Lab manual corrected as of 11/25/2016. I have made minor clarifications to workload requirements, to lab 6 and to the schematic entry requirements in hopes of reducing errors on your written reports. The procedure in Section 10 for programming the CPLD boards for lab 3, 4, 5, 7, 8, and A **has been revised to reflect changes to the Xilinx software package** and how it is supported on the Engineering network. The manual is only complete through Lab D. I hope to add a lab E on SDRAM and the SPI bus and the beginning of that lab is included. Reza and I are still working on preparing the lab so most of the details are still TBD.

wrp

*Brown University*
*School of Engineering*

# Engineering 1630 - Digital Electronics System Design

## Semester I - 2016-17

## *The Student Lab Manual*

Wm. R. Patterson

# Table of Contents


**1. GENERAL INTRODUCTION ..... 5**

**2. EVALUATION ..... 8**

2.1. GRADES ..... 9
2.2. PARTIAL CREDIT ..... 10
2.3. INCOMPLETES ..... 10
2.4. THE FAULT TOLERANCE QUESTION (FTQ) ..... 11
2.5. DOCUMENTATION ..... 13
2.6. DEADLINES ..... 13
2.7. THE ENGN1630 EXCLUSION PRINCIPLE ..... 14
2.8. LAB SCHEDULE ..... 15
2.9. TEACHER EVALUATION ..... 15
2.10. COLLABORATION ..... 15

**3. GETTING STARTED ..... 16**

**4. CIRCUIT CONSTRUCTION GUIDELINES ..... 17**

4.1. GENERAL HINTS ..... 17
4.2. THE 163 LAB ENVIRONMENT ..... 18
4.3. LOGIC PROBES ..... 19
4.4. POWER SUPPLIES ..... 19
4.5. BYPASSING ..... 20
4.6. DEBOUNCING ..... 20

**5. DESIGN AND TROUBLESHOOTING ..... 22**

5.1. DESIGN ..... 22
5.2. TROUBLESHOOTING ..... 23

**6. HOMEWORK, LECTURES, AND TEXTBOOKS ..... 25**

6.1. HOMEWORK ..... 25
6.2. LECTURES ..... 26
6.3. TEXTBOOKS ..... 26

**7. SCORECARD ..... 28**

**8. KIT INVENTORY ..... 29**

**9. THE LAB CHALLENGES ..... 30**

**9.1. DISPLAYS AND LIGHT EMITTING DIODES ..... 30**

**9.2. THE NUMBERED LABS ..... 34**

9.2.1. LAB ZERO ..... 34
9.2.2. LAB ONE ..... 36
9.2.3. LAB TWO ..... 38
9.2.4. LAB THREE ..... 48
9.2.5. LAB FOUR ..... 51
9.2.6. LAB FIVE ..... 53
*9.2.6.1. Verilog Implementation of a Counter – An Example ..... 56*
*9.2.6.2. Assigning Pin Numbers to Programmable Devices ..... 58*
*9.2.6.3. Editing and Compiling Verilog Files for Xilinx Parts ..... 59*
*9.2.6.4. Alternative Tutorial for Xilinx CPLD Programming with Verilog ..... 61*

STAGE 1, SETTING UP YOUR PROJECT ..... 61
STAGE 2, PROGRAMMING VERILOG: THE VERY, VERY BASICS. ..... 62
STAGE 3, COMPILING YOUR CODE AND LOADING IT ONTO A CPLD ..... 62
9.2.7. LAB SIX ..... 63
9.2.8. LAB SEVEN ..... 72
9.2.9. LAB EIGHT ..... 79
9.2.10. LAB NINE ..... 85
**9.3. THE LETTERED LABS ..... 87**
9.3.1. LAB A: ..... 87
9.3.2. LAB B: ..... 89
9.3.3. LAB C: ..... 94
9.3.4. LAB D: ..... 100
9.3.5. LAB E: ..... 114
9.3.6. LAB F: ..... 119
**9.4. THE LOGIC ANALYZER TOOL ..... 120**
**9.5. REQUIREMENTS FOR A SCHEMATIC DRAWING ..... 122**
**10. THE ENGINEERING 163 CPLD-II BOARD ..... 127**
*Figure 10.2: Silkscreen Pattern and Connector Functions for the CPLD Board ..... 130*
*Figure 10.3: Schematic Diagram of the Engineering 163 CPLD Board ..... 134*
**11. SCHEMATICS AND TIMING DIAGRAMS ..... 136**
11.1. GENERAL DOCUMENTATION ..... 136
11.2. TIMING DIAGRAMS ..... 137
*Figure 11.2: Timing diagram example. ..... 138*
11.3. USING THE DXDESIGNER SCHEMATIC CAPTURE SOFTWARE ..... 138
**12. COMPONENT DATA INDEX ..... 148**

## 1. General Introduction

*Course Overview:*

This is primarily a lab-based course. It starts with a series of simple labs getting you acquainted with basic gates and flip-flops and how they are connected to do such tasks as single-bit addition, error correction, bus multiplexing, etc. Few real systems are assembled this way anymore but there are two virtues to these labs. First, they are a simple way to let you learn the basics through your fingers. Second, you learn something about the logic of debugging circuits including finding mechanical problems. (Real systems have lots of design and reliability issues from mechanical problems. There are always difficult design problems in building neat, compact packaging and unexpected reliability issues from lack of durability or from signal errors due to crosstalk, electromagnetic interference (EMI), etc.)

Somewhat more complex logic operations are feasible for you if you build the circuit using programmable logic, one example of which is the complex programmable logic device (CPLD). The one you will use for many of the labs is the Xilinx XC9572XL, which is roughly equivalent to a few hundred basic gates. We have prototyping boards that allow you to embed these devices into larger more practical systems that you must design. Finally in the last few labs, you use an even more powerful programmable device, a Xilinx Spartan 3E™ field programmable gate array (FPGA). While the device itself has been purchased embedded in an evaluation board with a lot of support components, I have built input-output add-on boards to simplify having you design real non-trivial logic systems. Please be patient with us as I revise and augment these labs as I try to do every year.

*Software, Debugging, and Simulation Tools:*

As part of learning the practice of the profession, we require you to use some *industrial strength* CAD software. You will learn schematic capture in *DxDesigner*, a design tool suite from Mentor Graphics Corp. This tool is a widely used starting point for printed circuit board design, the only feasible means of solving those pesky interconnection problems! On the other hand, we no longer use *DxDesigner* to do circuit simulation of discrete circuits because small scale logic is generally not complex enough to warrant logical simulation while the real issues in such designs are electrical not logical, *e.g.*, transmission line reflections, propagation time effects, crosstalk, etc. These effects are not captured by logic simulators but require specialized simulators, one of which, *Hyperlinx*, is part of the *DxDesigner* tool suite.

You will only do logic simulation for your programmable devices, the complexity of which requires sophisticated simulation capabilityYou do labs using Xilinx XC9572XL CPLDs and XC3S500E FPGAs. For their simulation, we will probably use a mix of Xilinx software to program the devices and Aldec Active-HDL software to simulate them. There are continuing changes to both the hardware and software for this latter part of the course, so I expect that this material will not be integrated without a certain amount of difficulty. I ask your indulgence.

It may seem peculiar to you that I do not recommend simulating the first couple of circuits that you build with discrete gates. My feeling is that these are sufficiently simple that you should

have confidence in the logical functions you derive. For those of you who really want to do such simulation, there is a collection of free DIY software that includes a couple of very simple logic simulators at http://www.edn.com/design/diy/4430357/2/Top-free-DIY-tools-every-EE-needs . *Logisim* is one of these and was used in CSCI0310 for several years.

What is usually more problematic in the first labs is not the logical design but the physical interconnection. Making a *DxDesigner* schematic is not mandatory at the beginning of the course, only being due near the end of the semester. However, I think you would find it quite useful to design using a schematic to capture the interconnections. Knowing from a permanent, visual record where the wires have to go can make the wiring a lot easier and require a lot less debugging.

In the same vein, there is a requirement that you eventually use a logic simulator to show waveforms and time relationships in one of your labs. Again, the requirement only has to be fulfilled later in the course but you would get a lot more benefit from using that instrument on labs 5, 7, and 8 as you do the lab and find any need for debugging. Logic analyzers are great for that purpose because they expose both physical errors, that is, interconnection mistakes, and logical errors.

*Through-Hole Hardware:*

In Engineering 1630, you start by designing small systems with TTL integrated circuits, building them, and then fiddling until they work. The fiddling can go from minor rewiring to complete redesign as one gets accustomed to the subtlety of the problem. This method can be very satisfying and great fun. (Please do not assume that "fiddling" alone is the right way to design. Care in analysis and simulation is the only route to larger systems. Fiddling is simply a necessary step in learning to ask the right questions.) This is not a practical way to build larger systems so the later parts of the course introduce hardware description languages as the first element of more modern approaches.

Your kits are largely made from "TTL" or Transistor Transistor Logic parts because these are the main commercial source of very small blocks of logic, they are still available in convenient packages, and they are quite inexpensive. But it is important to realize that TTL is not the only or even the dominant expression of logical systems even at the very small systems level. In fact, the main driver for my changing the simpler labs is that the components are so obsolete they are disappearing from the market altogether. (If you decide to keep your kit at the end of the course, you will have a pre-made antique collection.)

The dominant manufacturing technology for digital circuits is CMOS - the Complementary Metal Oxide Semiconductor process, which we will examine briefly in class in explaining how gates actually do their work. For labs two and six in which you measure gate properties we will supply some TTL-compatible, low voltage CMOS parts from Texas Instruments' SN74LVCxx series. Devices in this family are intended to supply small logic blocks for systems using power supply voltages less than the 5 volts that were standard for many years for TTL. (Most discrete logic today functions on 3.3 volts or lower.) Their working input signal range is 1.8 to 5.0 volts and they will function with supply voltages over a similar range but are optimized for 3.3 volts supply or less. You will look at their speed and power performance over the full supply range.

*Programmable Hardware:*

Complex programmable logic devices (CPLDs in the trade jargon) are a competitive area of development. They are generally smaller and less flexible than FPGAs, but they are much less expensive, require many fewer support components, and have fast, well-controlled propagation delays. (FPGAs have benefited from the most advanced manufacturing processes. As a result their speeds are now faster. The basis of choice is usually the size of the logic - FPGAs implement larger systems, albeit at higher prices. CPLDs, however, do have the curious properties of more efficiently generating simple Boolean products of many inputs and of having propagation times more or less independent of logical function. They are also non-volatile, that is, you do not have to reprogram every time you turn the circuit on.)

You start to do labs with a CPLD at lab 3 and may use one again in Labs 4, 5, 7, A, and B with some restrictions. The ninth lab is on timing simulation, thus introducing the problem of verifying designs before they are built. Labs C through F are done with an FPGA.

You can program both CPLDs and FPGAs either using schematic entry or a hardware description language (HDL) or you can mix the two methods. In all cases you are just converting your thoughts into terms that software can convert to bit patterns for the switches and floating gate transistors in the devices. In this course you will use the Verilog HDL for design entry and will mix that with a schematic for the FPGA labs. In the early labs, labs 3, 4, and 5, you are limited in the syntax you are allowed to use because the point of the labs is the logic not the elegance of how well you can express intent in an HDL language. In the later labs, you are freer to exploit the full language flexibility. Dally and Harting have limited coverage of Verilog syntax and the older textbook by Wakerly (sects. 5.4, 7.13, and examples in chapter 6) has very good coverage of the basics of the Verilog language and there are many other tutorials on the web. I will put a copy of Wakerly on course reserve at the Sciences Library. Some TAs have found the site http://www.asic-world.com/verilog/index.html to be useful.

*Class Philosophy:*

The emphasis throughout Engineering 1630 is on the design, construction, and verification of digital circuits. In class, we will discuss the conceptual and physical building blocks available to the designer, and show how these tools and techniques are used in various design situations. In the lab, you will take this design theory and put it to use by building actual hardware-based solutions. This may well be your only exposure to full responsibility for making a product work ever! (Engineering, like Computer Science, is a surprisingly social endeavor - most products come from development teams, not isolated geniuses. Most continuing education to keep one's skill set fresh is peer-to-peer.)

Reflecting the fact that this is primarily a hands-on laboratory course, many of the customary requirements of engineering courses have been scaled down. There are only two lab reports and one schematic to compose, and these will be graded on a satisfactory/must be redone basis. The computer simulation lab and the logic analyzer exercise will be at least partially self-documenting. There are only minimal fixed deadlines for individual assignments. While there will be a midterm and a final examination, they are intended to encourage you to think of the lab material in a more general framework. Your grade will be calculated as an average of your lab grade

and the two exam grades. We expect the exams to test how well you respond to freshly posed problems of a similar nature to those found in the labs.

In the lab you design and build a particular circuit and then demonstrate that system to one of the TAs. **You must be prepared to show a schematic of your system drawn with your own hand or through your own schematic entry and to explain it to the TA. The TA must be satisfied with the quality of that documentation before he/she will examine the rest of your work**. After studying your approach, the TA will ask you a question about its performance or its design; if you answer the question correctly, the TA signs your scorecard, and you are free to move on to the next lab challenge.

In this course there are no specified times when you have to appear in the lab. You may design and build your circuit wherever you like. We will provide the necessary equipment to test and debug your circuit and will also provide a group of well-qualified teaching assistants to help you with any unusual problems. If you run into problems with malfunctioning equipment or software, please bring it to the attention of a TA or to me.

The liberalized format of ENGN1630 presents you with a number of challenges quite apart from actually building each lab assignment. By the time you take this course (typically in the junior or senior year) we expect that you have learned to think creatively, to work consistently, and to budget your time. The minimal deadlines established for completion of the labs force you to set goals for your own performance. While many design techniques are discussed in class, it is your responsibility to put broad concepts into practice. Making choices is part of the process.

Class will meet on Wednesdays and Fridays, from 3:00 PM until about 4:20. The lab will be open day and night most of the week with TA coverage for about 36 hours per week, with convenient evening hours at least twice a week. The lab will be in the Hewlett Electronics Laboratory, room 196 of the Giancarlo addition, affectionately known by most as "The Fishbowl".

## 2. Evaluation

The evaluation policy used in ENGN1630 changes with the convictions and whims of the instructor in charge. Some take a rather liberal approach, allowing your circuits alone to represent the depth and breadth of your digital logic design knowledge. Some lower the number of labs and their weighting to make room for formal homework assignments. Others of us use points-oriented grading schemes of Baroque complexity that make your final grade dependent on how well you score on mid-term and final examinations, in addition to the number of design exercises you complete. Each of these approaches has its benefits, and each has its drawbacks.

I follow a grading policy based on a point system. There will probably be a total of 165 points available, 65 points from exams and 100 points allocated for completed lab work as tabulated below. To some extent the exact number of points will have to be adjusted based on whether I can get an additional lab functional in time. I reserve the right to change the total number and their weight depending on how much I get done. Passing performances on both exams and lab work **separately** are required to pass the course, but a curve will be applied to your total grade to determine the letter grade reported to the Registrar. I find that determining the break points of that

curve is sometimes difficult and I try to use criteria that call for significantly above average performance on at least one of the two components of the grade is necessary for an A.

## 2.1. Grades

Your lab grade is based on the number of lab challenges you complete. There is no partial credit on the lab grades. Instead you will receive points for each successfully completed lab challenge or piece of written lab work. There are 13 hexadecimally numbered labs now. I continue to hope that I will be able to add at least one and perhaps two more. With the points for using the schematic drawing software and the logic analyzer, there are a total of around 100 possible lab points. The distribution of points is:

| | |
|---|---|
| Lab 0 through Lab 5: | 5 points each |
| Lab 6 through Lab 8 and software simulation Lab 9: | 7 points each |
| Lab A and B: | 7 points each |
| Lab C: | 8 points |
| Lab D | 10 points total<br>4 pts for first test (I/O)<br>6 pts for instructions |
| A schematic for either Lab 1, 5, 7, 8, or A: | 6 points |
| Logic analyzer measurement challenge (see section 9.4): | 5 points |

To pass the course, however, you must have passing grades on **BOTH** the examination and lab parts of the course. A passing grade on the examination part of the course is 48% of the sum of the maximum number of points possible on the exams. A passing grade on the lab part of the course is **57 points**, which **must include credit** for Labs 1 and 2; for one lab from the set 7 and 8; for Lab 9 and for one lab from the B through F series; for the Logic Analyzer Challenge, and for a schematic of either lab 1, 7, 8, or A. **THERE WILL BE NO EXCEPTIONS TO THIS POLICY.** Please note that the use of a logic analyzer is mandatory. The laboratory has a superb collection of instruments and their use is to hardware what a debugger is to software. You don't know much about either discipline without knowing its fundamental test tools.

A word of advice: labs 7 and 8 involve systems with mixed analog and digital signals. Many students find these labs time consuming because they find it difficult to be systematic about dealing with the analog part. The letter series labs may also take significant amounts of time. However, they are purely digital. If you find that the first analog-to-digital converter lab that you do takes too long, then do the letter labs next. Only do the other A/D lab if you have time toward the end of the course.

Scuttlebutt variously suggests that either lab 7 or 8 is much easier than the other. I believe they are about equal difficulty. (I have changed some of the parts and requirements on both labs to make them easier to do than they used to be.) A careful and systematic approach is critical to success within reasonable time limits on either of them. It is not uncommon for some students to take 30 or more hours to do one of these labs, primarily because they do not think systematically about how to test what is going on in the prototype. The use of simulation for the digital section in advance can save enormous amounts of time on these labs and will provide a basis for logic analyzer measurements comparing expected and actual performance. Similarly, comparing the logic analyzer picture to your design intent can also make a big difference in the amount of time the lab takes. Analyzers save time despite their having a certain learning curve. Since you have to use an analyzer eventually, it makes sense to take the time to learn its use early enough to use it here. If I were taking the course, I would make it a point to learn the logic analyzer at the fifth lab on simple state machines.

Section 7 of this manual contains a scorecard for recording your performance on each of the labs; each time you successfully complete a lab, a TA will sign the line associated with that challenge and make an entry on our backup spreadsheet. I recommend you check that the TA makes this spreadsheet entry. To receive your grade, you must turn in your scorecard at the end of the semester, which is defined as no later than 5:00 PM of the day of the scheduled final examination. **Do not lose your scorecard!!!** I will give credit automatically for whatever is in the duplicate records spreadsheet, but this backup tends to be incomplete, and there are **NO guarantees without the signed scorecard.** People who arrive empty-handed at the end of the semester will receive No Credit for the course.

## 2.2. Partial Credit

The nature of ENGN1630 precludes the assignment of partial credit for any of the Labs. If, for example, you are told to build a circuit that displays the digits 0, 1, 2, 3, 4, 5, 6, and 7 in a particular order, and you offer a circuit that displays all but the digit 3, you get no credit for the Lab until you troubleshoot the circuit and get it working as requested. This policy just reflects the nature of digital system design, and it is one from which no deviations will be made.

## 2.3. Incompletes

No grade of *Incomplete* for unfinished labs will be given unless you present evidence from a physician or a Dean of a serious and protracted medical problem. Because of the nature of the course, dealing with students after the end of the semester is difficult. The TAs are gone, the lab space is reorganized and used by other courses and other students, and all parts must be checked, cleaned, and inventoried. If you need to pass ENGN1630 to complete the requirements for a specific Brown degree, make sure you *at least* qualify for a grade of C. I have no sympathy for someone who shows up at the end of the semester with a nearly-blank scorecard in hand, and claims that he or she needs this course to graduate. The time to consider the relative importance of ENGN1630 to your educational experience is ***now.***

## 2.4. The Fault Tolerance Question (FTQ)

When you demonstrate a circuit for a TA, you and the TA will spend a couple of minutes verifying that the circuit meets all requirements. **You must then show a schematic and explain the operation**. (NOTE: This procedure is relatively recent change.) The schematic does not have to be machine generated although that would be preferable. It does have to be both neat and accurate. If you are asked where a particular connection point on the schematic is and reply that well you changed that, then you get to go away and do the documentation over again. The TAs get to decide whether the drawing is neat enough to understand. The TA will then check your breadboard serial number. Remember that you must do your own work with your own kit and substituting another person's breadboard for yours may lose you credit for the lab and risks academic penalties. Finally, the TA will ask a Fault Tolerance Question (FTQ). These questions may take one of three forms, at the option of the TA. For the first type of question, the TA will choose one wire in your circuit and ask what, if anything, will go wrong with your circuit after the wire is removed. For the second type of FTQ, the TA will introduce a problem into your circuit, leaving you the responsibility of finding and correcting the problem. The third type of FTQ will allow the TA to ask you general questions about the operation of your circuit, such as "What will happen to the output of the Q1 flip-flop if both inputs to this NAND gate are grounded?" It is a TA's prerogative to ask for documentation before an FTQ.

These notions of an FTQ are based on TTL implementations of small systems. The labs based on programmable devices (CPLD or FPGA) pose special problems for framing good FTQs. In preparation for an FTQ or even for certification of compliance with specifications, **you must present full documentation of your design** including the wiring schematic and any programming file whether Verilog programming code, Xilinx schematic file or Aldec block diagram. The form of that documentation is at the discretion of the TA who may ask for it as a file in your computer account, a printed copy for easier reading, or even an electronic copy. Most likely the TA will prefer a paper schematic and an on-screen file. At the least, the TA will examine this and ask how you designed it. She may ask variants on the hardware FTQs based on changes to operation of the circuit that will be caused by a change to the schematic, Verilog, or Boolean description and may expect you to compile and demonstrate the expected result on the spot. Another possibility is that you will simply be asked to explain the logic of your system and answer questions (plural) about why you did it the way you did.

If you answer the FTQ correctly, the TA will sign your scorecard, congratulate you, and you can celebrate! (As a precautionary measure, you might want to be sure the TA enters your success on the backup sheets just in case you lose your scorecard.) If you do not answer the FTQ correctly, the TA will be obliged to ask you **two more** FTQ's. For every FTQ that you miss, two more must be asked. If several FTQ's are answered incorrectly, we will begin to wonder how you can know so little about a circuit that you designed, built, and tested yourself. We will begin to wonder about plagiarism. If you miss three FTQ's in a row, your circuit breadboard will be temporarily confiscated and you will be referred to the head TA for further consultation. If you are guilty of demonstrating a circuit that someone else designed, you will automatically lose credit for that lab. (If this is a required lab, you may not be able to pass the course.) If a second offense occurs, standard academic discipline procedures will be initiated.

We are aware that a certain amount of shopping around is done by students looking for a soft-hearted TA. To limit this behavior, no one can have more than five labs checked by any one TA. All the TAs will have been told that we regard FTQ's as serious matters, and no TA has to be a patsy.

For most of the labs, once an FTQ (or a series of FTQ's) have been answered correctly, you have met all the requirements for that lab. There are a few exceptions to this rule. Labs 2 and 6 involve the measurement of important properties of your chips, such as power dissipation, logic levels, propagation delay, etc. They require data and the answers to some questions to be handed in as a report in lieu of an FTQ. These write-ups will be graded only on a *satisfactory/must-be-corrected* basis and will be handed back only after all reports for that lab were due. **You must both take the data and write the report individually!** It is perfectly okay to give each other advice and information, but you are responsible for doing your own work.

Similarly lab 9 on simulation and the schematic drawing by computer design entry require documentation in lieu of FTQs. All documentation is handed in through a box in room 196. We will mark that box with an appropriate sign but you can't miss it.

An FTQ can be fairly difficult sometimes. We will try not to choose bizarre wires or concoct impossible-to-analyze situations. In any event, you are responsible for giving a **complete** account of impending faults in your circuit *before the wire is pulled or the software program modified.* If you mention that X and Y will go wrong, but only X goes wrong when the wire is pulled, your answer is incorrect. On the other hand, if you mention that X and Y will go wrong, and X and Y *and Z* actually go wrong, then your answer is also incorrect. In some cases, you may want to list contingencies: "If the input goes HIGH, then X will occur, but if the input goes LOW, Y will occur." This may be particularly important for CMOS circuits for which an uncommitted input is not in a well-defined state. Vague answers, like "the circuit will not work" or "it would take a scientist to figure it out" are not acceptable. Neither is going to lunch; once an FTQ is asked, you must answer it during the same session. Any interruption in the FTQ process between a question and an answer will be recorded as a question failure. You may not break off a losing streak of questions without leaving your breadboard and a record of the missed questions with the TA for passing on to the next TA who takes up the questioning. You may only break off such a streak for good cause, and ignorance is not a good cause. Be sure you understand your system before you ask for an FTQ. The TA may pass you on to his or her successor if there is a shift change during your questioning.

It is the privilege of the TA to choose an appropriate FTQ. If you feel that you have been given an unfair question, first try to work out an answer. If you fail the question, you may bring your complaint to me. If I feel your complaint is justified, you will be asked a single replacement question; otherwise you will be referred back to the original TA who will ask the normal two-question FTQ follow-up.

Because an FTQ may be non-trivial, take a reasonable amount of time to answer a question. Allocate *at least* fifteen minutes to this process. Do not let the atmosphere of the lab make you anxious - getting the first question correct can save you a lot of time. We expect that over the course of the semester you will spend from 2 to 4 hours answering FTQs.

*In Summary:*

- Generally perform the following steps to have a standard lab recorded:

    1. Show the TA a neat schematic of the lab, and explain its operation.
        a. For programmable labs, similarly share the code/block diagram.
    2. Exhibit the operation of the circuit to show that it meets all requirements.
    3. Have the TA check your board number.
    4. Answer an FTQ of the TA's choosing.
        a. For each unsuccessful FTQ answer, you must answer two more FTQs.
        b. FTQs must be answered completely and correctly, and cannot be delayed without good cause.
    5. Have the TA sign your scorecard, and record the completion in the excel sheet.

- A TA may check off only 5 of your labs.
- **Academic honesty is taken very seriously,** in both designing and building the circuits and in taking and reporting data.

## 2.5. Documentation

Proper documentation is an essential aspect of the design process; I expect you to get in the habit of specifying your circuit completely while you are devising and building it. Circuit diagrams, written circuit descriptions, timing diagrams, truth tables, etc., are the counterparts in hardware design to the documentation of computer programs which is so much a part of satisfactory programming. To encourage good habits in this area, we are requiring a carefully done schematic for one lab you have completed. (You may not hand in documentation for a lab that does not work! We may reject a schematic for a bad design even though you may have gotten it past a TA. For example, some people carefully adjust components to get a poor design to work for a while. That may fool a TA sometimes, but it may not fool the one - very likely me - who grades a stack of schematics.) The schematic must be done using the *DxDesigner* computer aided design (CAD) package. **You must also use the same tool to generate a Bill of Materials for the circuit including adding information on the particular parts in the circuit at the level of commercial availability.** See section 9.5 for the full set of requirements. We may even **require redrawing** if your schematic has a **poor esthetic layout** even if the interconnections are correctly expressed. Please see section 11.2 for information on timing diagrams for lab 9 and section 11.3 for further information on how *DxDesigner* works.

Also, you cannot expect much professorial or TA help with a non-working breadboard unless your working drawings are reasonably complete and legible. The TAs have been instructed *not* to answer questions about circuits unless they see a reasonable amount of documentation and to require a schematic as part of the evaluation process.

## 2.6. Deadlines

Many years ago, I experimented with deadlines for general groups of labs. As a result of its success in retaining the sanity of both students and TAs during the course, I continue the tradition. The lab groups, and associated completion dates, are:

*Group one:*

Labs 0 through 3 may be checked off only until October 12, 2016.

*Group two:*

Labs 4 through 9 may be checked only until November 21, 2016.

*Group three:*

Labs A and B and the schematic diagram requirement may be checked off only until December 7, 2016.

You may obtain credit for the labs within each group *only* up to the posted date. (Note that labs may be completed as soon as you wish, but they must be completed by the given enpumpkination date.) The last week before the final exam will be a grace period. During this week, you may have up to one additional lab from each group checked off. You can actually have the lab checked earlier but the credit date must be in exam week. You can only get one lab a day credited in exam week.

Labs C through E may be checked off at any time until the last day of the semester, the day of the final exam. This year the ENGN1630 exam is on the morning of Saturday, Dec. 17, 2016. I will allow labs to be checked off until the Tuesday after the exam since that is the latest that grades can be integrated before I have grades due. However, keep in mind that **TA assistance and availability for FTQ's decreases during exam period** for the obvious reason.

ENGN1630 is allocated a fixed amount of lab space, a small number of TAs, and a finite amount of circuit testing, troubleshooting, and demonstrating equipment. It has been our experience that most people (including me!) tend to procrastinate when not confronted with fixed deadlines. Without deadlines, some students fall hopelessly behind by the end of the semester.

Note that students who thrive on the excitement generated by trying to complete the course at the last possible moment will especially appreciate and benefit from this deadline approach, since there are now four “last possible moments” to contend with instead of only one!

## 2.7. The ENGN1630 Exclusion Principle

For each day that the lab is open, **you may have only one lab signed off**. This will still allow ample time to complete as many labs as you wish - after all, the typical academic semester has at least 70 class days but only 15 or so labs have been defined for the entire course. The Exclusion Principle has two purposes: first, it prevents one individual from monopolizing TA time, and second, it provides more incentive for you to learn to work efficiently and steadily rather than in bursts. Deviations to this policy will not be considered. Please note that this has implications for how much you can hope to get done in the end-of-semester grace period.

## 2.8. Lab Schedule

The ENGN1630 lab will be in Giancarlo room 196, the Hewlett Electronics Laboratory. It will be open every day except Saturday while the University is in session. Use the door nearest the ramp out of the building as its electric lock will be unlatched. Please do not leave the other door open or unlocked. TAs will be available for some 35 plus hours a week including at least two evenings during the week, probably Wednesday and Thursday.

We will try to keep TA schedules posted in the lab and on the web, telling when each TA will be available for troubleshooting help and for checking off labs. No TA will be assigned to work when class is in session. Also, although we will try to keep most lab slots staffed with at least one TA, there will undoubtedly be times when the lab has a TA scheduled but he or she does not turn up. The TAs have job interviews and other personal obligations and while I ask them to arrange for substitutes, this is not always practical. Do not schedule yourself so tightly that you miss getting a lab signed off because a TA overslept. The responsibility for getting labs checked off in a timely fashion is yours, not the TA's.

You are encouraged to come to the lab at times when the crowd is small (such as 9 AM) you will get more attention and have shorter waits for lab sign-offs.

## 2.9. Teacher Evaluation

Besides my obligation to evaluate you, you will have an opportunity to evaluate me at the end of the semester. The School of Engineering arranges for web-based evaluations of teaching. The results are used for class assignment, salary adjustment, and (occasionally) tenure review. Some of the evaluation is designed to measure whether I have met the course goals for ABET. (Those goals are listed on the handout you receive the first day of class.) Finally, I distribute forms for the *Critical Review*. I take these reviews, both for the School and for the *Critical Review*, quite seriously, and would appreciate your doing the same. (By the way, I do not read any of them until after the final exam. The Dean's office retains all forms and tabulations until after exams are over and grades are submitted.)

## 2.10. Collaboration

Unlike some courses, you are welcome to collaborate with your fellow students during the semester. Discussing problems with a friend is a good way to troubleshoot your circuit, and can be a source of self-improvement for both of you. Be sure that you completely understand any advice offered by anyone, however, since this advice may be wrong - providing even more frustration than the original problem that you yourself devised. Make sure that you will not be stumped on an FTQ and doing this will help with exams too.

In general, two kinds of collaboration can be distinguished: *soft collaboration* and *hard collaboration*. Soft collaboration is a perfectly acceptable process of seeking and offering advice on the design of a particular lab solution. We assume that you will understand any advice given, and hence will benefit from the information exchange. Note that while you may get design help from a

friend, **you must use your own kit to construct the solution, you must generate your own documentation, you must be ready to explain your system with understanding, and you must answer your own FTQ.**

Hard collaboration means the use of another person's circuit board, wiring or documentation during your FTQ. It also encompasses blind copying of another's design without sufficient understanding to pass the FTQ process. In labs requiring computer files, you must generate your own files individually, making your own choice of variable names, comment records, indentation, etc. **All the data and reports for labs 2 and 6 and the schematic must be done individually.** I will take a very dim view of nearly identical reports or **duplicate data**. If you copy something from another source as part of an explanation in your report, such as a figure or paragraph from Wikipedia, you must show explicit attribution in a footnote with appropriate bibliographic citation. (For the material in these labs, you really should be capable of writing the explanations yourself.) Copying is hard collaboration! None of the forms of hard collaboration will be tolerated. If you feel compelled to use a friend's design for a given lab challenge, you must at least go through the process of making your own schematic from scratch not by Xeroxing, wiring up the solution (including going through the computer entry process for CPLD or FPGA designs), and debugging it yourself! All the wiring and troubleshooting associated with lab challenges must be performed on the circuit boards issued to you at the beginning of the semester. TAs will be required to match the numbers found on your scorecard with those on the back of your boards. If you are found using someone else's board, you will automatically lose credit for the lab you are trying to demonstrate. (If this occurs during one of the labs required to pass, you cannot possibly pass the course.) You may also get your friend in trouble for loaning his or her boards. You may be referred to the appropriate Dean for action by the Academic Code Committee and will probably receive a grade of NC for the course. Penalties for tampering with someone else's board or kit will be dealt with even more stringently.

Improper collaboration, besides wasting the thousands of dollars you pay for taking this course, is a serious matter and will not be tolerated.

## 3. Getting Started

You will receive your own chip set along with this lab manual. Look over Lab 0, work out a solution on paper and test it conceptually until you are convinced it will work. Then assemble and test it. Be especially careful about connections to the power supply. The easiest way to damage a chip is to get the power supply connections backwards. The +5 volt supply lead must go to the VCC lead of each device, and the ground or 5 volt return lead must go to the GND pin of each device. If you are using LVTTL parts, be careful not to apply an overvoltage. Failure to be careful may result in wholesale damage to your chips. We realize that some of your chips may be defective as a result of prior use (unusual but possible), but we will not replace components damaged by carelessness without some charge. Another point to be especially careful about is not to connect the output of a chip to +5 volts or even worse connect any pin of a logic chip to plus or minus 12 or minus five volts. These voltages are available on the supplies and may get used in labs 7 or 8. (Chips usually are more tolerant of outputs accidentally connected to ground.) **In the labs that use the CPLDs, which is half the labs and most of the required one, you must program the device before you connect it to other wiring. We have special cables on the power supplies so**

**you can program the XC9572XL. Then turn off the power supply, wait a moment for the supply voltage to disappear, remove the power cable and substitute your cable.**

***Hot-swapping cables can damage CPLDs too.*** Always turn the power off and wait a few seconds before removing a cable to the CPLD board and never connect the cable with the power supply turned on.

**Nota Bene:** Massive destruction of chips or melted or burnt areas on protoboards or keypads will result in appropriate adjustments to the **price at which we will buy back your kit.** Your keypads are mechanically very robust, but they can be ruined easily by using them to short out the power supply, *i.e.*, to connect VCC to GND. This melts the plastic film under the buttons, causing them to stick. **You must use limiting or pull-up resistors** with the keypad. We are not sympathetic to damaged keypads.

When you are sure the circuit works properly, find a TA, demonstrate it, explain it, and answer an FTQ. With this signoff, you are well underway! Remember the deadline system, and remember that the lab gets busier and the challenges harder as the semester progresses.

If Lab 0 has you baffled, if you do not know who Herr Georg Ohm was (and why he is remembered -- his lab equipment is carefully and reverently preserved in the Deutsches Museum in Munich), if you do not know the difference between voltage and ground or capacitance and resistance, do not despair. The graduate TAs and I want to be helpful and can explain a lot if asked. Throughout the course you will have every opportunity to have your fears explained away; your responsibility is to take advantage of the help available.

## 4. Circuit Construction Guidelines

### 4.1. General Hints

Be a little compulsive. Organize the components in your chip set so that you do not waste time looking through the entire lot each time you need a particular chip. If you sort them in numerical order, you will easily find any part you need. The black foam that holds your kit chips is static resistant and a place of safe storage. If you want to store them in other materials, you should lay down a piece of aluminum foil to press onto the pins of the CMOS chips; this will short their pins together and prevent the chip from being destroyed by high-voltage static electricity. (CMOS parts including CPLDs are the most vulnerable parts, but TTL parts can be damaged too. The foam we use for kits is pretty good without aluminum foil, but be careful if you substitute other materials.)

Another suggestion is to make a set 3x5 index cards that summarize the pin connections and logical functions of each of the chips in your Kit. This may cost a bit of time now, but it will save time in the long run. These cards will serve as handy references on how each chip behaves, and will allow you to avoid leafing through your lab manual each time you need to know how the chip works (especially during the answering of FTQ's!). The automated form of this is to make use of the *DxDesigner* libraries and draw everything on-line. If you don't want to do index cards or look through the back of the manual, you can find the full data sheets on the class web site.

In general, you should start a lab challenge by working through the requirements of each subcircuit. Construct truth tables, if appropriate, showing the relationships among the various inputs and outputs of your system. **Try to decompose one large problem into several smaller problems**, keeping in mind the limitations of your chip selection and the need to integrate each of the pieces into a whole. Once you understand what needs to be done, work out a complete pencil-and-paper solution. Simulate the circuit by hand to see what happens during various combinations of inputs. Make sure that most of your logic errors are caught while you are still working in a pencil-and-paper mode; it is easier to erase a connection than to fix a circuit. You **may wish to use *DxDesigner*** to turn your hand sketch into a full schematic. This will help with accuracy of pin assignments, with finding missing signals, and with legibility. The *DxDesigner* software offers a design rule checker that finds outputs connected together, single pin nets, misnamed nets, etc. Further testing by simulation at the design stage is standard practice and is much recommended for the Xilinx-based designs. With practice, the time to draw and simulate a circuit is short enough that it is completely offset by the time saved in construction and debug. Remember that neat drawings help you work out ideas and build things with minimum errors. Per the comments above, this effort will also guarantee better help from TAs or from me.

After designing a solution, build your circuit gradually. Isolated sub-sections of the design can then be tested apart from the rest of the solution. If your circuit evolves in a step-wise manner, you will be able to understand its operation better, and you will be less likely to be baffled by a hard FTQ.

When you assemble your circuit, do it neatly with wires that are not so long that they rise above the breadboard in a confused mass. Make sure that only a minimum of bare wire is exposed throughout your circuit; this will cut down on intermittent problems that result from improper insulation. **Do not make wires so short** that they loop tightly over the top of individual chips, however, since this will prevent you from exchanging a chip if it fails. (This also prevents testing it or attaching a logic analyzer to it if you *suspect* it has failed. Logic analyzer connections are usually made with a clip that fits over the chip. Wires over the chip prevent doing this.) The CPLD board has a special connector to make connections to the logic analyzer. Assigning pins to make that connection give a proper analyzer display can help too. For example, if you have a bus, order the bits so they go high to low when displayed on the analyzer since that is the only format the analyzer will display.

When you wire a circuit, use a consistent color-coding scheme. For example, make all +5v lines red, all ground lines brown (or any other Earth tone, for that matter), etc. To aid in troubleshooting, use a variety of wire colors throughout your circuit so that co-located wires can be easily distinguished. For parallel lines of information (busses) you may want to use wire colors corresponding to the resistor color code.[1]

### 4.2. The 163 Lab Environment

Besides being the place where your circuits will be tested by TA's, room 196 is where you can go to use a power supply, scope and logic analyzer for troubleshooting, to talk with fellow students about the lab challenges, and generally to bask in a high-tech atmosphere. Do not, however,

[1] The resistor color code is a series of color and digit assignments. See the back of the data sheet section of this manual for the code's definition.

make a nuisance of yourself! No loud social gatherings; no smoking; no drugs; no food fights or practical jokes. Observe normal lab courtesy. Turn off equipment when you are done. Return all wires and cables to their normal places. Clean up little wires or scraps of paper. Do not remove anything from the lab (except your lab kit) without permission. If you break something, or notice something is broken, notify a TA.

Check the whiteboards for the latest news concerning the course. If there is a change in the lecture schedule, or if some particularly useful lab information is discovered (such as the opening or closing of the lab at non-standard times), it will be posted there and on the class web-site.

For some labs, you will want to use an oscilloscope to monitor rapidly changing voltages. If you do not know how to use one, please ask a TA. He or she can explain time bases, vertical calibration, triggering, dual trace operation and other features. Because TAs are only a year ahead of you, they may not know all the details. The manuals for scopes are on the class website -- be persistent. Please be careful when using the scopes – be especially cautious with the probes. These probes are easy to damage. Similarly the logic analyzers have delicate probes that need care in handling. There are a number of manuals in the lab for the analyzers that discuss the instruments in detail. Check the drawers in the back of the room. We do have a good supply of leads and clips, and you shouldn't have to jury rig everything. Ask the TAs to get more leads from me if the supply is getting low.

There are 16 PCs on the workbenches in the lab to support the labs and they run from the same server and SAN that the Computing Facility uses. There is software on the computers that will remove anything that you leave on the C:\ drive and some directories on the D:\ drive. Use the space allocated for you on the U:\ drive.

If you are unfamiliar with any other pieces of equipment in the lab, ask a TA for assistance.

### 4.3. Logic Probes

Most of the power supplies are equipped with logic probes that can quickly tell you the voltage levels of any point in your circuit. The LOW LED is on for voltages less than 0.8v; the HIGH LED is on the voltages greater than 2.4v. If the logic probe measures voltages between 1.0v and 2.2v (or if it is unconnected) no lights will glow. A third LED will light if the probe detects a pulse train.

### 4.4. Power Supplies

Room 196 has approximately twelve digital test stations. The heart of each of these stations is the power supply. To activate your circuit, connect a wire from the +5v terminal or from the -5v or +3.3v terminal as appropriate to the appropriate slot on your breadboard and a wire from the ground terminal to your ground pins. Always double check that you have the proper polarity of power *before* turning on your circuit. **Reversing the power supply leads could damage all of the chips on your breadboard!**

For your own reasons, you may wish to avoid the bustling atmosphere of the 163 lab and instead test your circuits in the privacy of your own room. The advantages of this approach are that you will be able to control the noise level around you, and you will also (presumably) be able to control the size of the crowd waiting in line for you to finish. The disadvantages of this approach are that there will typically be no TA to offer advice, and that an alternate source of 5 volt power is necessary. There are a number of solutions to this latter problem.

Because supply voltage fluctuations cause circuits to malfunction and because TTL circuits draw large currents, any approach based on batteries is a marginal solution to your problem. The parts for a simple power supply can be purchased from Radio Shack for about $20. If you are interested in building one, I can give you a circuit diagram and some guidance on how to do it. For more money, it is possible to purchase a fully assembled 5v power supply that will deliver 1 or more amps of current. For example, a 5v/5A supply with auxiliary +-12-15v outputs costs under $ 40.

## 4.5. Bypassing

No power supply is perfect in the sense that a constant potential difference of exactly +5v is present between every pair of connections to VCC and GND on every extension of every wire of the power-to-ground system. The relative immunity of digital systems to extraneous potential variations (noise!) is a major reason for the ubiquity and power of digital logic, but that immunity is not infinite. Because of small series resistances and inductances associated with the power supply and its wiring, noise, that is, transient current spikes generated by pulse circuits or by TTL gates changing state, can cause voltage changes on the power wiring and make your logic circuits malfunction. Sequential circuits -- counters, flip-flops, etc. -- are especially sensitive. Other noise may come from equipment turned off and on somewhere else in the lab or from the 60 cycle magnetic field noise generated by fluorescent lights, or from *reflections* of signals sent over long wires in your circuit. (In fact, a major subdiscipline has grown up around designing out problems with power and signal distribution. It goes by the name of signal integrity analysis and design.)

To protect your circuit from transient voltage spikes on the power supply lines, you can "bypass" or "decouple" the power supply by placing capacitors directly from power to ground at key places in your circuit. Key places are typically clock pulse generating or clock pulse receiving chips, such as oscillators, multivibrators, counters, flip flops or other state machines. You have several 0.1μF capacitors among your parts that can be used for bypass. More are available if necessary. Because it takes an appreciable amount of charge to change the voltage across such a capacitor, these will divert some of the current causing voltage transients. **Be aware of the possibility of bypass problems when troubleshooting your circuits!** $V_A$ If a counter seems to skips values or a flip-flop changes state for no logical reason, a bypass capacitor from VCC to GND at that chip may be necessary. Please be aware too that the 74ACT04 chip may be a source of extra noise and may require its own bypass capacitor. All except the simplest labs require at least a couple of bypass capacitors. One other possibility to check when troubleshooting an erratic circuit is that the supply voltage **at the chips** may be less that the 4.75 volts required for proper operation. Measure it directly at the chip with a multimeter.

## 4.6. Debouncing

Mechanical switches, such as those on your keyboard, are another source of unwanted pulses. Suppose you press one of your keyboard buttons. Initially, the two internal wires are separated (or form an "open circuit"). Pressing the button will close the circuit, and cause contact between the two wires. Instead of making a single continuous contact, however, the two wires actually make several bouncing contacts on a microscopic scale of millisecond duration. Looking at the event with an oscilloscope you may see a series of pulses instead of the single pulse you desire. Figure I illustrates this problem and shows a method of *debouncing* a 2 wire push button.

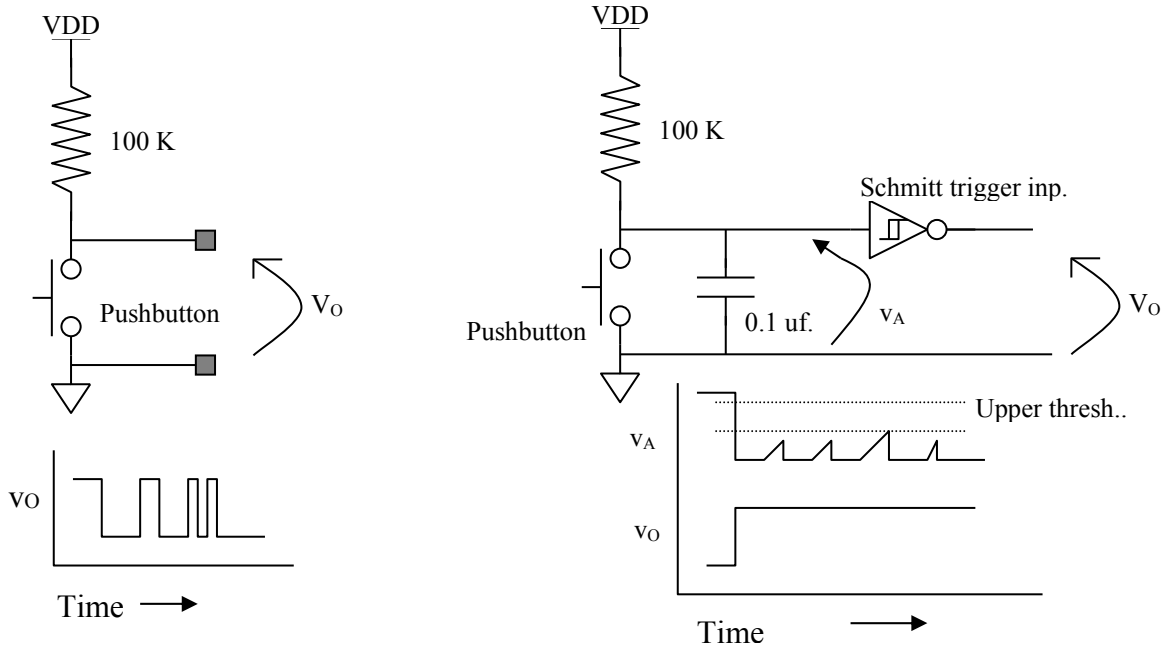



**Figure I: Debouncing a single-contact mechanical switch or button.**

The solution idea here is similar to the solution of the bypass problem. Use a capacitor to absorb some of the current associated with the unwanted pulse activity. In Fig. I, we have a large resistance connected from the power supply to the gate of an inverter. If there is no other current at that logic gate input, the capacitor will remain charged to the power supply voltage. If the gate input is grounded, as it can be with the push of a button, the gate voltage will change immediately. If mechanical bounce occurs, and the contacts momentarily part, the capacitor will recharge with a time constant of about $R \cdot C$. The actual time this circuit takes from the opening of the switch contacts until the signal $V_A$ crosses the inverter threshold depends on the type of inverter. In lab 2 you will find that TTL gates have substantial current flowing thru input terminals. These currents charge the capacitor faster than the resistor alone would. With 74LSxx series gates, C must be several microfarads to get a 10 ms. hold time. If the gate is a CMOS gate, then it usually has no steady input current. In that case, an RC time constant of 10 ms as in Fig Eq II would result in a hold time of 3 milliseconds. [The exact result is $\Delta t = -R \cdot C \ln\left[(V_{CC} - V_{THG})/(V_{CC})\right]$ .] If, well before the 3 milliseconds elapse, mechanical contact is made again, the gate voltage will again drop to 0v and be kept *below the threshold* that causes a change of state for the gate. Thus, the output of the gate will remain low in spite of small changes going on at the input due to mechanical bounce. You may first encounter the need to put this theory into practice in Lab 4.

When the mechanical signal source is a switch instead of a pushbutton, an alternative method of debouncing the switch signal is available. Figure II shows two variants on the technique, one using the preset/clear lines on a flip-flop and the other using two NAND gates connected as an RS flip-flop. The idea is that the single pull double throw (SPDT) switch bounces on the make or break of the moving contact with either of the two stationary ones but does not bounce back and forth from the one to the other. The flip-flop responds to the first time each contact is made and ignores subsequent bounces until the other contact is closed again.

From your point of view, what we hope you will notice about the devices in your kit are the input currents, and the power versus frequency relations. One of the important results of the difference in input currents is that TTL devices have uncommitted inputs which are always high, while CMOS devices have undetermined values for uncommitted inputs. Also, this behavior and the high power supply noise of the newer CMOS parts can make them not interchangeable with TTL despite having nominally identical interface specifications. **It is good construction practice never to leave an input floating; connect it to either VDD or GND**. Good design tools will flag such open pins as errors during design checks. Also with your CMOS parts, as with the TTL parts, it is very important to bypass power connections.

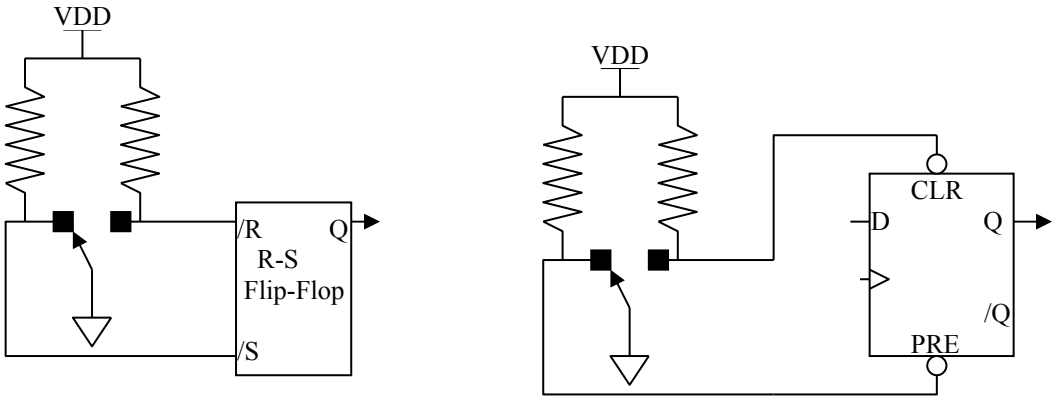



**Figure II: Using flip-flops to debounce an SPDT switch.**

## 5. Design and Troubleshooting

### 5.1. Design

You may find that inspiration is not enough to keep you working on lab challenge designs. Develop a moderate weekly schedule for steady progress. Your goal could initially be to design or build a lab solution during each session, although later labs may require considerably more time and effort. Do not find out the hard way that necessity is not always the mother of invention, especially at 4:30 pm on the last day you can get a certain lab checked off. However well you organize your time, you may still be frustrated by problems. The problems will come from incomplete circuit design, sloppy construction, or unsystematic troubleshooting.

Design problems can be approached in one of two ways. First, you can wait for inspiration to strike. This may involve re-reading a lab challenge, and then taking a walk in the rain. (A

shower may be substituted on those occasions that the sun shines for several successive days in Providence.) At the end of your walk, you may have subconsciously thought about the problem enough to have a good first approximation for a solution. A second approach may be to sit down somewhere and think very hard about the nature of the problem and how it might be solved. You can read the recommended texts for the course. You could talk to friends. You might even pay attention during lecture! Another good approach might be to study the actions of various chips in your kit, and see if this provides a clue to how a problem may be solved.

Try to break the problem into several smaller ones. For example, if part of your assignment calls for a display, design and build the display portion of your circuit (being careful not to design yourself into a corner!). Seeing a portion of your assignment work may provide the incentive to explore other aspects of the challenge. Suppose you have tried to think of *some* solution for a couple of hours, and still are getting nowhere. Try (in the following order) these steps:

1) dreaming,
2) asking another student for help – soft collaboration,
3) consulting a TA for a hint,
4) seeking an appointment with the professor,
5) going to the Emerald City for an appointment with the Wizard of Oz.

Help from step 4 is guaranteed to provide you with a basis for hope.

### 5.2. Troubleshooting

Suppose you have a design that, on paper, looks as if it should work and yet does not give proper outputs after you have hooked it up and tested it. Recheck your wiring. Are the power supply connections made properly? Are all the inputs to each gate accounted for? Do all of the outputs go somewhere? Next, check the power supply. Are you getting the proper voltage? Is the polarity correct? If the power supply has been reversed, all the chips in the circuit may be damaged. Is there an adequate number of bypass capacitors? Does one chip seem like the culprit? Take it out and test it by itself; you may have a defective chip that you should exchange for a good one. The CPLDs are very easy to damage! While we try to keep on top of replacing them, sometimes the class destroys them faster than we can keep up. Are you having problems with clock signals? Sometimes with a heavy load and poor wiring, the NE555 will give two clock edges at one nominal low to high or high to low transition. If there is a switch in the system, you may need to bypass or debounce its signal as described in the Construction Guidelines.

Use the logic analyzer to test activity at multiple pins simultaneously. (The analyzer is really not that hard to use and is very powerful in displaying the action of a system. Use it early and often rather than leaving the Logic Analyzer Challenge to the last possible moment.) There are clips in the lab that go over DIP integrated circuits and make it easy to connect the logic analyzer to all the pins at once. The CPLD and FPGA boards have logic analyzer plugs that connect 16 pins to the analyzer in one push. As of the start of labs, every logic analyzer has a plug that fits the ones on these boards and a second cable that has individual wires that can go on the DIP clips. In assigning pins of the CPLD to signals, lay them out to simplify the use of the analyzer, *e.g.*, if there is a bus, get the pin order correct for the order of bus signals to be correct on the analyzer display. If you have timing signals, sketch what you expect the timing waveforms to be and make sure that the

proper events happen in the correct sequence. (Remember that simulation can be part of your paper design and the comparison of what is predicted and what is measured is a wonderful diagnostic method.) You may want to single step through the states of your circuit. Examining timing waveforms with an oscilloscope, you may discover noise on a clock signal or discover a signal that does not meet the specifications for a HIGH or LOW. Particularly in labs that use sequential circuits, you should use the analyzer to see if one event is happening before or after it ought to according to your design. You may have to introduce or remove delays from your circuits. Don't be afraid to use the right tool even if it takes a little while to get used to it. You designed the circuit and ought to know the sequence of events it should display. Developing the ability to compare design to reality is the chief reason to require labs rather than doing everything in simulation.

You may want to double-check interfaces between one kind of chip and another. For example, if you have an output from a gate driving an LED, the voltage across the LED in the HIGH state may be clamped lower than the HIGH logic levels required for other connections to that output. You need a resistor in series with that LED. Or, you may have a CMOS chip fanning out to too many TTL chips. Or, you may have a TTL chip unable to reach threshold voltage on a CMOS input. Is your problem intermittent? You may want to make sure that the wiring has been done in a neat manner. Wiggle the wires in their breadboard socket holes to make sure there are no inadvertent open circuits. Bare wires may cause unwanted short circuits. Do not hesitate to trim the ends of wires and strip them again as worn tin plate can lead to bad connections.

Let us suppose, however, that you have checked all these possibilities and your circuit still does not work. What should you do? The next thing to do is sleep on it. We are not talking about a Rip Van Winkle solution here - just overnight, or at most forget it for the weekend. You hope that a subconscious integration of the paradoxes and dilemmas of your case will lead to an intuitive insight - dream therapy, if you like!

Now suppose it is the next day, and the consolation of sleep did not help. Seek out the help of another student or a TA. You may know someone who has already done the lab or know a student from last year's 1630 course, or you may have a sympathetic friend to whom you can explain your situation out loud. Often giving such an explanation helps you realize the problem or the friend or TA may be able to spot an oversight. Of course, you will want to restrict your give-and-take with other students to "soft collaboration." Remember that any advice you get from anyone may be tested by the Fault Tolerance Question when you finally demonstrate your circuit!

One important requirement in talking to a TA (and probably to a friend too) is you must **be prepared with legible documentation**. (Sloppy documentation is very frustrating for both TAs and me. Do not try to transfer your frustration to one of us!) A TA may also set up some test, the results of which you both will study. She may want to swap a couple of replacement chips in and out. If she finds you have made a trivial error, she will point it out to you and you will be home free. If you are way off base though, the TA's are instructed not to give you a complete answer to your problem. The TA will suggest ideas that should help you do the design yourself. During this time you must keep in mind that the lab challenges are not simply homework problems that you are doing as practice for exams; the challenges are the primary basis of your grade itself, and as such, we and the TA's constrain ourselves not to spoon-feed solutions to you.

Suppose you take the ideas of the TA and attempt to implement them. You may modify your circuit; you may start over and build another one. But it is possible you will continue to fail. You are convinced the TA is a bozo. Things are worse now. You will have slipped to *stage five frustration.* Why might this be? The TA may not have understood your design, or may not have been willing to give you enough information for you to navigate out of your particular maze, or you may have simply been too depressed or anxious to appreciate the advice, or the TA *may really be a bozo!* You will now want to seek out a Professor for troubleshooting advice. I am always available after class. Also I am glad to answer questions if you find me in my office (Room 449) or labs (Rooms 195, 325 or 703). For more extended advice, make an appointment. When you do come, however, make sure that you come prepared with legible documentation of your design and have your circuit wired neatly. Be willing to describe your problem and the attempts you have made to solve it. I will try to understand your circuit enough to make a judgment about it. Be willing to endure a Socratic dialogue. I will tell you whether I think your design can work, and if I think it cannot, I will suggest some sort of redesign. Ask as many questions as you like. Do not leave until you feel at least a little optimistic that further work will be profitable. Remember that I cannot tell whether you understand something unless you tell me honestly if you do or don't. Don't try to hide your confusion when I say something mysterious. Make your feelings known if you think your progress has been halted entirely!

A final word on frustration: if the results of tests on your circuit indicate it, don't be afraid to acknowledge that you may have designed yourself into a dead end and that the best strategy may be to pull out all your wires and chips and start with a fresh idea. Engineers, who have become too enamored of a particular design that will never work, tend to become pretty bitter. They drift from anger to cynicism to apathy. They become blind to the faults of the circuit at hand. They view it as possessed by supernatural forces beyond the understanding of student logic. They kill time by watching movies like "The Exorcist" when all they really need is a good EXOR gate. In professional life such tendencies are a devastating handicap.

This advice about frustration may seem silly as you proceed with the early labs. We hope the advice will still seem silly as you continue on, but experience indicates that this may not be the case. You should be aware that the later labs, particularly 7, 8, A, B, and C, may require much more attention to detail and knowledge of chips.

## 6. Homework, Lectures, and Textbooks

### 6.1. Homework

ENGN1630 is largely a laboratory course, and hence has very little written homework. (I will give two optional problem sets but these are recycled from year to year. I will neither collect nor grade the answers.) It does have exams. The purpose of the mid-semester is to give you practice in answering abstract design questions; think of it as a structured means of studying for the Final exam. The mid-semester exam will actually come fairly late, probably in early November. I will grade and return it within a week or two. The Final Examination will also be graded, but will not be returned immediately. (You will have a chance to look at it as soon as I grade it.) If you want additional practice doing problems, try the ones at the end of each chapter of the textbook.

The major "homework" component of the course is the design and testing of the laboratory challenges. These labs are for your benefit, and should be done by you alone. Beat-the-clock cookbook labs are gone. No more lab partners with the IQ of a tree on one hand or the experience of Thomas Edison on the other. No more lab reports graded from 1 to 10 on neatness and precognition.

Unfortunately, some participants feel that the entire goal of the course is to complete the seven or nine or fourteen labs required to get a specific grade. The exams are intended to discourage this, but perhaps this is an unavoidable by-product of my general approach. I hope you will spend the time to expand your understanding of design in general, and to obtain an appreciation of the subtleties of digital design. One way to do this is to attend lecture and to read the readings. Another is to allow yourself to worry about larger questions, not just "can something other than a lecture help me design lab 3 in the shortest amount time."

## 6.2. Lectures

The sequence of lecture topics from the course syllabus are not a guaranteed sequence but are roughly what I plan.

1. Digital signals, discrete symbols and voltage level standards, etc.
2. Boolean algebra, logic minimization, SOP and POS forms, K-maps
3. MOSFETs, static logic circuits, noise margins, signal levels, open-drain and three state circuits, flip-flops, switch models of gates
4. Sequential circuits and finite state machines with counters as special cases
5. Floating gate devices and their use to build PALs and CPLDs; logic programming in Verilog.
6. Verilog hardware description language as used to synthesize simple circuits and some rules for good coding style
7. Quick introduction to analog to digital and digital to analog conversion.
8. CAD for schematic capture and programmable device programming
9. Semiconductor memory: SRAM, DRAM, ROM – asynchronous and synchronous
10. Field Programmable Gate Arrays (Xilinx examples) for logic implementation
11. Arithmetic: adders, fast carry methods, simple multiply, number representation (2's complement integers, offset binary, IEEE 754 floating point standard)
12. Register transfer machines and very simple models of computer structure.

Some of you may decide that the lectures are too simple or too sophisticated and not attend. If, however, you seek troubleshooting advice from a TA for a problem that had been explained in a lecture you did not attend, the *TA may give your problem a low priority* and urge you to look over another student's lecture notes. Furthermore, **ALL** the material covered in the lectures is potential subject matter for questions on the examinations. For example, I believe that you should know at least a little about transistor circuits because you cannot do chip-level logic design without it and all logic design today tends ultimately to be done on chip.

## 6.3. Textbooks

The **suggested** textbook for ENGN1630 is *Digital Design: A Systems Approach* by William J. Dally and R. Curtis Harting (Oxford Univ. Press, 2012). This book concentrates on application specific circuit design in Verilog and is quite well done for the topics it does cover. It does not cover some of what I talk about very well and is weak in its discussion of Verilog syntax and nuance. It is not a very thorough language reference. The textbook I have used other years, *Digital Design: Principles and Practices, 4th Edition*, by John F. Wakerly (Prentice Hall, 2006) is nearer to my style. Wakerly is comprehensive and well written, as good a text as any I know of and works as a good language reference for both Verilog and VHDL. Still he does not cover A/D conversion at all and covers several things lightly, *e.g.*, memory and FPGAs. The book has become exorbitantly expensive and is also showing its age particularly discussing the state of the art in memory or programmable logic. I will try to put a copy of it on course reserve in the Sciences Library.

**7. Scorecard**

Name_____ Board #1_____ Board #2 _____ Board #3_____

| *Lab* | *Title* | *Date Completed* | *Checker's Initials* |
|---|---|---|---|
| 0 | One Bit Full Adder with NAND & EXOR | | |
| 1 | Error Detection and Correction in 4 Bit Data * | | |
| 2 | Logic Family Properties and Logic Voltage Levels * | | |
| 3 | Two Digit Common Cathode Multiplexed Display | | |
| 4 | 16 Button Matrix Keyboard Encoding | | |
| 5 | Counter with External Control | | |
| 6 | Propagation Delay Measurement Using a Scope | | |
| 7 | Dual Slope A/D Converter ** | | |
| 8 | Successive Approximation A/D Converter system ** | | |
| 9 | Software simulation of either Lab 5, 7, 8, A, B, or C (Please indicate which lab → ) * | | |
| A | 4-Bit x 4-Bit Hexadecimal Multiplier | | |
| B | Music Box (Xilinx FPGA based logic) *** | | |
| | DxDesigner Schematic and BOM for either Lab 1, 5, 7, 8, or A. * | | |
| | Logic analyzer applied to Lab 5, 7, A, B, or C. (Please indicate which lab → ) * | | |
| C | Tweet++ Scrolling Electric Sign - FPGA controller driving 14-segment, 8-character display *** | | |
| D | A brain-damaged small processor built within the logic of an FPGA with dual-port memory for I/O. Has 16-bit instructions and data with a VERY limited set of instructions and no interrupts. *** | | |
| E | DRAM Controller with SPI I/O interface Implemented in Xilinx Array – Note: This lab is still vaporware; I only reserve the right to introduce it.*** | | |
| F | TBD *** | | |

* Mandatory to pass
** Mandatory choice of labs 7 or 8
*** Mandatory choice of one lab from B, C, D, E, or F to pass

## 8. Kit Inventory

| Item | Qty | Part Number | Cost |
|---|---|---|---|
| NAND gates, two input TTL, four gates/chip | 2 | SN74LS00N | $0.58 |
| Hex inverters; CMOS, 6 inverters per package | 1 | SN74ACT04N | 0.56 |
| Hex inverters; TTL, open collector, high voltage | 1 | SN7406N | 0.75 |
| Hex inverters; Schmitt trigger inputs | 1 | SN74LS14N | 0.39 |
| D Flip-Flops; two FFs/chip | 4 | SN74LS74AN | 1.56 |
| EXOR gates; 4/chip | 3 | SN74LS86AN | 1.38 |
| One-shot (monostable multivibrator); 2/chip | 1 | SN74LS123N | 0.59 |
| 1 of 4 Decoder/Demultiplexers; two/chip | 1 | SN74LS139AN | 0.59 |
| 8 -> 1 Multiplexer | 1 | SN74LS151N | 0.72 |
| 2-> 1 Multiplexers; four/chip | 1 | SN74LS157N | 0.75 |
| Counter, binary up/down, Sync. PRE & CLR | 1 | SN74LS161AN | 0.60 |
| Shift register, serial in-parallel out, 8-bits | 1 | SN74LS164N | 0.88 |
| D-FF latches; six/chip | 1 | SN74LS174N | 0.70 |
| Inverters; TRI-STATE | 1 | SN74LS240N | 0.80 |
| Analog switch/multiplexor; four/chip | 1 | DG202BDJ-E3 | 1.45 |
| Comparator, open collector output | 1 | LM311N | 0.36 |
| Operational Amplifiers; two/chip, BIFET | 1 | TLV272IP | 0.45 |
| Timer, astable or monostable | 1 | NE555N | 0.65 |
| Light-emitting diodes | 1 | NSL5056 | 0.80 |
| Matrix keyboard | 1 | 11KS121 | 4.95 |
| Miniature toggle switch SPDT | 1 | 1MS1T1B1M1QE-EVX | 2.60 |
| Eight position DIP switch | 1 | | 2.10 |
| 25K potentiometer | 1 | 201XR253B | 0.46 |
| Resistors: 100(2), 510(1), 1K(6), 10K(6), 100K(3), 1M(2) | | | 0.92 |
| 10M(1), 39K(1), 5.1K(2); (1/4 W.) | | | |
| Resistors: 20K (precision 1% ¼ W, for A/D labs) | 6 | P20.0KCACT | 0.39 |
| Capacitors: 0.001μF(2), 0.01(2), 0.1(6), 1(+/- 20%), 51pf(2) | | | 0.92 |
| Wire cutter/stripper (Miller) | 1 | 84N1038 | 3.56 |
| Solderless Breadboards, each individually numbered | 2 | EXP-300E | 7.90 |
| Cable for protoboard to CPLD board connection | 1 | In-house assembly | 5.65 |
| Total | | | $44.01 |

## 9. The Lab Challenges

### 9.1. Displays and Light Emitting Diodes

### The Care and Feeding of Displays and LEDs

**WARNING:** It is fairly easy to burn out part of a hexadecimal display or to destroy an LED. We look with disfavor on such actions as they usually mean you have been careless. This section describes the electrical characteristics of these devices and how to avoid problems with them. There is also information on wiring the two digit displays so that both digits are usable at the same time. This process, called multiplexing, is needed for the first time in lab 3.

How can you burn out a segment of a display? *By allowing too much current to pass through the segment.* How much is too much? The data sheet for the display gives several pieces of information: under absolute maximum ratings is IFP = 60mA., which means "Peak forward current per segment not to exceed .06 amperes" (60 "milliamperes" ). Also under absolute maximum ratings is $I_F$ = 20 mA., which means that for long term operation with more than one segment lit, the average current through a segment must not exceed .02 amperes. In testing a display with one segment lit for short periods of time, it is probably safe to use a current between these two numbers, but nearer to the latter. A maximum of 30 mA is safe choice. There is also a typical rating of $V_F$ = 2.1 volts @ 20 mA., meaning that about 2.1 volts must appear across a segment in order to pass 20 mA through it. Thus, the potential difference across R in Fig. Disp-i is $V$ = 5 – 2.1 =2.9 volts. According to Ohm's law,

$$V = IR = 2.9v = (30mA.) \times R$$

thus

$$\mathrm{R} = \frac{2.9}{30 \cdot 10^{-3}} = 97\Omega$$

so a resistance less than 97 ohms in series with 5v and a forward biased (operating) segment may cause irreversible damage.

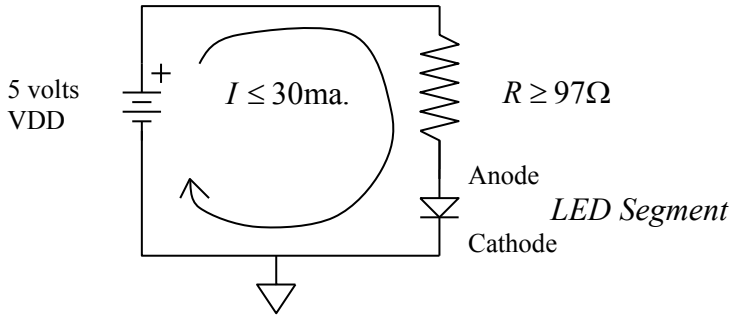



**Figure Disp-i: LED biasing.**

When you are probing pairs of pins to determine the pinout of the display you will be safe if you use a 100Ω (brown-black-brown) resistor in series with +5v and ground. Later, after you have

determined which pins connect to the segment anodes (and which connect to the common digit cathodes) you will be using the displays on the CPLD-II or BUXUSP-II boards, and these have current limiting resistors built-in.

If you place *too much* resistance in series with a segment (> 1K) the segment will be too dim for good visibility.

Each segment is a light-emitting diode (LED), with an output wavelength of 656nm, which corresponds to red light. The electrical characteristics of diodes are discussed in chapter 3 (sec. 3.9) of Wakerly, and in Taub and Schilling (T&S) chapter 1. Both books will be on reserve at the Sciences library for this course. The I-V plots of a silicon diode and of an LED are shown superposed in Fig. Disp-ii. The primary difference in their electrical characteristics is the difference in the voltage required to produce appreciable current flow through the diode, which is called the turn-on voltage $V_{on}$.

Note that a diode does not obey Ohm's law! For one direction of applied voltage, there will be no current flow or light output. In the other direction (called the "forward" direction) the current flow and light output will rise rapidly once the potential exceeds $V_{on}$. For a regular p-n junction silicon diode $V_{on} \approx 0.7$V; for our LEDs $V_{on} \approx 1.7$V. If a diode is placed across the output of a logic chip (i.e. between the output pin and ground) the chip output will be clamped to $V_{on}$ when the chip output is driven to a "HIGH" state! A value of 1.7 volts is *less than the minimum high output* expected by other chip inputs. Loading chip outputs with LEDs can be a source of problems *unless* a "big enough" resistor is placed in series with the diode to increase the voltage across the combination. Depending on the driver chip characteristics, "big enough" may be more than 100 Ω.

What does it mean that your display is a "common cathode" device? Cathode and Anode are terms that refer to the two ends of a diode. Common cathode means that all the cathode ends of the seven segment diodes (and the decimal point) are connected together at *one* pin, which normally should have a pathway to ground (perhaps through a transistor or a gate, if both digits are to be displayed "simultaneously"). See Figure Disp-iii. .

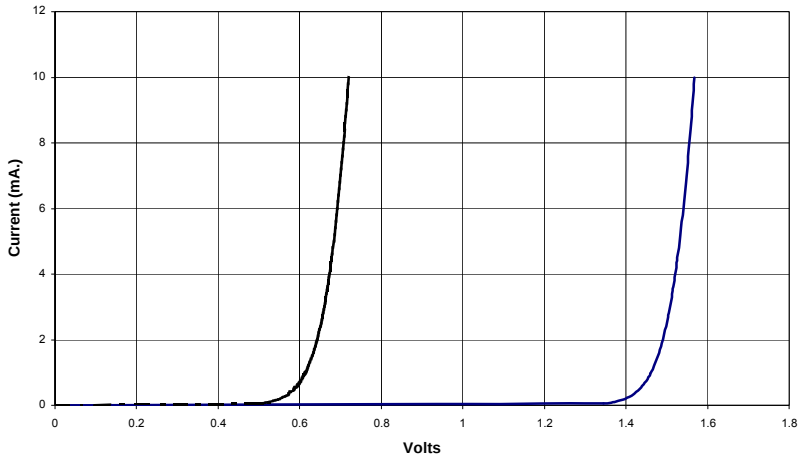



**Figure Disp-ii: Diode I-V plots.**
**(The left curve is a silicon diode and the right one is a red LED.)**

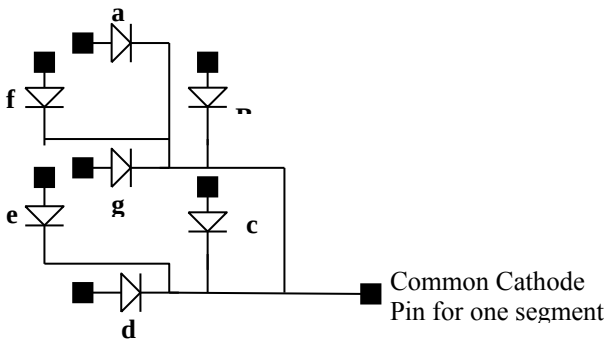



**Figure Disp-iii: Common cathode display internal wiring. (Segments are labeled a – g. Black rectangles are pin connections.)**

What is a "multiplexed" display? It is one in which two or more digits time-share a common set of wires and decoding logic. Multiplexing is used to save wiring and cost. It is done by connecting the anode of one segment of anode of the same segment of every other digit in the display. All the cathodes of each digit are then wired to a line unique to that digit. Thus, for a two digit display there is one connection to each pair of segment anodes (eight pair in all, counting the decimal point). See Figure Disp-iv.

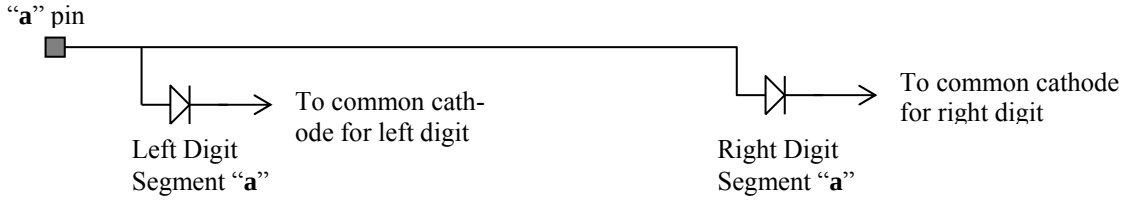



**Figure Disp-iv: Display segment multiplexing.**

The signals for the left and right digit segments must share these common points. The segment information must be *multiplexed* (i.e. it must share, in time, the common set of anode wires). Data for the right segment is applied to the anode lines when the right segment cathode is grounded and left data when the left cathode is grounded. By rapidly changing back and forth between these two conditions, say at a rate of several hundred times a second, one can make both digits appear to display the appropriate data continuously.

The displays on the CPLD boards are common cathode (with the decimal points connected to the same common cathodes). While each segment anode has a separate pin, the anodes are connected together pair wise between digits to make a display that can be multiplexed. There are current limiting resistors in series with the anodes. This saves pins and wires, probably the most expensive parts of most simple systems.

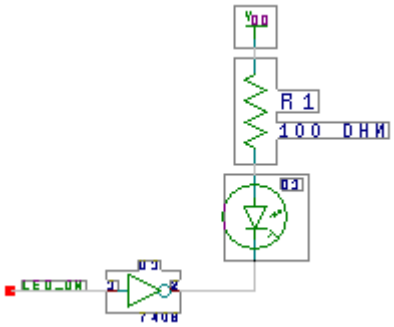



**Figure Disp-v: Driving an LED from a logic gate.**

The other LEDs in your kit, two NSL5056's, must also be protected from too much current - with a current limiting resistor ($\geq$100 ohms). Placing these diodes directly across +5v to ground will burn them out. A common requirement is to turn on an LED from a logic signal. It happens for reasons we will discuss in class that almost all logic gates can conduct more current from output to ground in the LOW state than from VDD to output in the HIGH state. For this reason and a couple of others, the circuit shown in Figure Disp-v is the preferred arrangement. Note that the LED lights for a 0 on the output or a HIGH on the input of the inverter. The inverter can be an open collector or open drain type such as the 7406 though standard inverters of sufficient drive capability can also be used.

## 9.2. The Numbered Labs

### 9.2.1. Lab Zero

### One Bit Full Adder

**Requirements:** Design and build a one bit full adder using only **one** 74LS00 (quad NAND) and **one** 74LS86 (quad EXOR) for logic. The adder is to have *four* outputs, two of them are the usual **Sum** and **Carry-out** signals; the other two are outputs normally used by fast carry-look-ahead logic. The latter two depend only on A and B and are called **Propagate** and **Generate**. The three signals, "Carry-out", "Propagate", and "Generate" are related to each other as

$$C_{out} = C_{IN} \cdot P + G$$

The Generate signal is partially listed in the table below to get you started. This equation is not enough to define *P* unambiguously. However the further constraint that

$$P \cdot G = 0$$

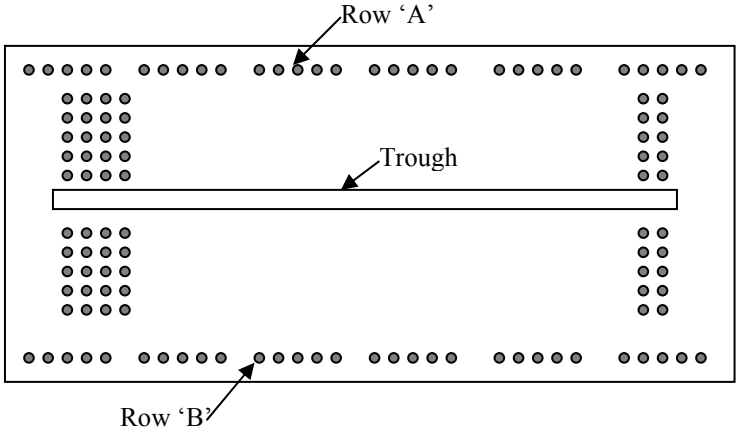
does define it sufficiently. Another way to describe the Propagate signal is that it is HIGH if and only if the values of *A* and *B* are such that both $C_{IN}$ = '1' would imply $C_{OUT}$ = '1' and $C_{IN}$ = '0' would imply $C_{OUT}$ = '0'. (Note that one implication of this statement is that *P* and *G* are never simultaneously true. When *G* is HIGH, then $C_{OUT}$ is HIGH even if $C_{IN}$ is LOW.) This definition of *P* is deliberately slightly different than some common definitions of this signal. Initially, the Sum and Carry-out signals must be displayed with two LEDs. Be prepared to connect the LEDs to the *P* and *G* outputs when asked to do so by the TA. Set up your DIP switch contacts 1 to 3 to be *A*, *B*, and $C_{IN}$ respectively. (WARNING: be sure to do this in such a way as not to damage the switch and so as to get full noise margins. **Use pull-up resistors.** See comments about pushbuttons under "Debouncing" in the introduction for how the circuit would look.)

| Inputs<br>Outputs . | | | | | | |
|---|---|---|---|---|---|---|
| Carry In | A | B | Sum | Carry Out | Generate | Propagate |
| 0 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 1 | 1 | 0 | 0 | |
| 0 | 1 | 0 | 1 | 0 | 0 | |
| 0 | 1 | 1 | 0 | 1 | | |
| 1 | 0 | 0 | 1 | 0 | | |
| 1 | 0 | 1 | 0 | 1 | | |
| 1 | 1 | 0 | 0 | 1 | | |
| 1 | 1 | 1 | 1 | 1 | | |

**Discussion:** The full adder is discussed in chapter 6.10 of Wakerly. The truth table is given here. You should be able to fill in the *P* and *G* columns from the description given above.

Of course, with lab 0 (as with all demonstrations of working labs) you will be asked a Fault Tolerance Question: "What, and only what, will go wrong with your circuit \ if such-and-such a wire is removed?" To achieve credit for Lab 0 you must answer a Fault Tolerance Question correctly. In preparation for the question, you must have a legible schematic.

The holes in your breadboard are connected as shown in Fig. 0-i. Normally you connect +5v to row *A*, GND to row *B*, and straddle the chips across the trough to have each pin contact a different column $C_i$. Determine the LED anode and cathode pins by testing with a 100Ω resistor connected between the diode and VCC. Truth tables for NAND and EXOR are shown below:




All holes in row 'A' are connected. Likewise all holes in row 'B' are connected, but neither row connects to anything else. These rows are normally for VDD and Gnd.

Each set of 5 holes in a column are connected, but column on opposite sides of the trough are not connected.

**Figure 0-i: Breadboard interconnections**

| Inputs | NAND | EXOR |
|---|---|---|
| **0 0** | **1** | **0** |
| **0 1** | **1** | **1** |
| **1 0** | **1** | **1** |
| **1 1** | **0** | **0** |

Designing lab 0 is largely the clever application of DeMorgan's theorems, not necessarily to minimize the logic but to push it into a form that fits the two chips. For further help come to class and consult Wakerly chapter 4 or any text that discusses combinational logic.

The Propagate and Generate outputs are used in systems that calculate $C_{OUT}$ by different means. These systems go under the names of carry-look-ahead and Manchester carry adders. It is unlikely that one would use both $C_{OUT}$ and Propagate/Generate logic in the same adder. Nonetheless it can be done within the limits of your logic, and an actual system might use this circuit and prune it according to which type of block was needed at a certain point.

### 9.2.2. Lab One

**Error Detection and Correction for 4 Bit Data**

**Requirements:** Set up one of your DIP switches to simulate seven bits of parallel data within a communication or storage system requiring error correction. The four least significant bits shall be the data bits $Q_A$ through $Q_D$. ($Q_A$ is the least significant bit.) The next three bits are error correction bits assumed to have been derived from the Venn diagram of Figure 1-i with each circle having even parity. $ECC_1$ is the fifth significant bit, $ECC_2$ the sixth, etc. Assuming that no more than one bit can be in error, devise a circuit that continuously displays the correct (or corrected) hexadecimal representation of the data bits $Q_A$ through $Q_D$ on a LED seven segment display. We have preprogrammed a CPLD board that it will display the hexadecimal value of four bits connected to it. (**BE CAREFUL not to "HOT PLUG or UNPLUG" these boards**. You can kill the CPLD doing that and then a TA has to program a new one.) See discussion below for the wiring of that board. Also light a separate LED if there is an error in any of the seven bits.

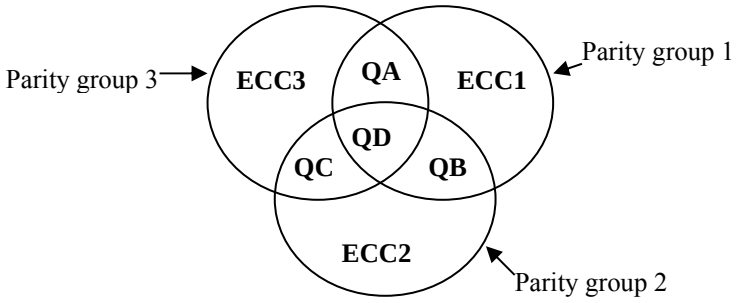



**Figure 1-i: Venn diagram of error correction bit assignment.**

**Discussion:** There are many situations in digital systems in which data bits cannot be counted upon to be completely accurate. One such situation commonly occurs in communication systems where noise on a channel can occasionally cause one of the bits in a data stream to be interpreted incorrectly as its complement by the receiver. Memory systems are also subject to soft errors (e.g. alpha particles from trace radioactive contaminants in the packaging materials hitting a memory chip can cause a single location to change bit polarity). All of these sources of error are to a greater or lesser extent unavoidable. Typically many of these sources of error are random in occurrence and relatively infrequent (say one error in every $10^5$ bits or more). For this reason, single bit errors are the most common, and two nearby bits in error are very infrequent. However, with the high transmission rates and large storage capabilities of current machines, random bit errors, even at very low rates of occurrence, would render the systems useless if it were not possible to detect and correct most of them.

Errors in the data bits can be detected by introducing additional "error correction" bits which have a specific, known relationship to the data bits. The parity-based version of error correction in this lab was invented by the great applied mathematician Richard Hamming at Bell Labs.

The data bits are divided into groups, and error correction bits are assigned to each group to give the group a particular *parity*. A group is said to have "even" parity if it has an even number of '1s', and odd parity if the number of '1s' is odd. For four bit parallel data, there is a way to add three correction bits to form three parity groups from which single bit errors can be both detected and corrected. Figure 1-i shows the simple Venn diagram on which the technique is based.

Each circle represents a parity group. The ECC bit in each circle is the added correction bit needed to make the parity of all the bits in that circle even. For example, parity group 1 contains $ECC_1$ and the data bits $Q_A$, $Q_B$, and $Q_D$; if $Q_DQ_BQ_A$ = 001, then $ECC_1 = 1$. The value of $ECC_1$ is chosen to make the parity of this entire group even. Similarly the values of $ECC_2$ and $ECC_3$ are chosen to make groups 2 and 3 have even parity.

If any single bit is changed, the parity of one or more of the groups will be changed. You can convince yourself that an error in any one of the seven bits will give rise to a unique parity pattern, so that one can detect the specific bit that is in error. For example, an error in $Q_B$ causes odd parity in groups 1 and 2 but not 3. There are three parity groups yielding $2^3 = 8$ error states: one for no error and one for each of the seven possible single bit errors. Since one can discover which bit is in error, one can correct it, so this scheme can both detect and correct all *first order* errors (i.e. no more than one bit altered). For more information, you might consult R.J. McEliece, *Scientific American*, January 1985, pp. 88.

Incidentally, if two bits of the seven are altered, this scheme will misdiagnose the error, and the correction scheme will typically make things worse. There is, however, a way to add a fourth correction bit which (through the use of additional parity checks) will detect the existence of a second order error (two bits altered). Unfortunately, this scheme does not provide enough information to correct the second order error, but it does allow one to detect the existence of an unrecoverable error. **You might want to think** about how this could be done.

We have programmed a few of the CPLD boards described in section 10 to use as displays for this lab. (You only use these displays for this lab – in later labs you program the display yourself if it is needed.) You connect a display board to your protoboard with the ribbon cable in your kit. The pinout for those boards refers to the 24-pin DIP plug on the protoboard end of that cable. Pin numbers are stamped on the top of the DIP plug. The pin assignments for the display are:

| Pin | Function |
|---|---|
| 24 | VDD = + 5 volts (be sure this connection is made before turning on the power.) |
| 10 | GND = 0 volts |
| 23 | Q3 - MSB of hex display (Use pull-up resistors for high – do NOT connect directly to VDD.) |
| 22 | Q2 |
| 21 | Q1 |
| 20 | Q0 - LSB of hex display |
| 11 | Cathode of display - MUST CONNECT TO GND |

### 9.2.3. Lab Two

### Logic Family Properties and Logic Voltage Levels

**Notes**: Lab 2 is one of only two *measurement labs*; the other labs are synthesis (design) exercises. The *raisson d'etre* for labs 2 and 6 is that the properties you measure are the absolute minimum electrical characteristics that a digital-system designer must understand to make intelligent decisions. Because this is a different type of experiment, *you hand in a report* instead of demonstrating a circuit for Lab 2. Be prepared to answer questions about your measurement procedures, which we may ask you if the report is not completely satisfactory.

To prepare for this lab, read chapter 3 of Wakerly. Also, read the lab description carefully and be sure you understand the procedures and explanations *before* you come in to the lab. There are only a limited set of meters for this lab, so please do not waste time by attempting to do it unprepared!

Organize your report in the spirit of Computer Science documentation standards or Engineering core-course labs. There is a box on the rear bench in room 196 for handing in your report in room 196. I ask that you not hand things to me. I lose them. The report must be **typeset** and contain three basic parts. First is a brief introduction stating **your** understanding of the importance of interface standards and of the large noise margins incorporated into such standards. Second, tabulate your measured data with units and measurement conditions clearly labeled. Finally, answer the questions that follow the measurement requirements. The report will be graded and returned somewhat after the deadline for doing this lab. The report will be judged satisfactory and you will receive automatic credit for it or else some specified part must be done again or must be explained better. If only minor corrections are wanted, the TAs will judge your response to the required corrections and will check off your scorecard in the usual way. Reports needing more substantial corrections will have to be turned in again. Write-ups that do not require corrections will be recorded automatically. Your returned report will indicate which process to follow.

### Measurement Requirements:

1) **Logic Levels and Noise Margins - TTL Family:** For each of your four, standard-TTL inverter chips 74LS14, 7406, 74LS240, and 74ACT04, graph the voltage out as a function of the input voltage over the range $0 < V_{IN} < 5v$. (The 7406 will not act as an inverter without a pull-up resistor; for standardization, use 1 K ohm.) Take sufficient measurements to be able to plot the output transition from high to low with good accuracy. Pay particular attention to the gate threshold levels and to the values of input that produce .8 and 2.4 volt outputs. To make it easy for you to do this, we have several sawtooth waveform generators that plug into the power supplies in the lab. (Please do not use the Agilent function generators for this purpose – they are in short supply and are way too easy to damage. Susceptibility to damage is one reason there are now so few of them.) The homemade generators produce a roughly 1 KHz sawtooth wave that goes from .1 volts to 4.5 volts. You apply

Connect this signal to the input of the gate **with a small (51 pf) capacitor** also connected directly from input to ground. (The reason for this capacitor is to prevent any interaction of the gate with the source, especially for Schmitt trigger devices.) Use a scope with two identical 10X probes to measure the input and output simultaneously. Set the scope in X-Y mode with the gate input on the X axis. The resultant plot is the input-output relation directly. Take enough data that you can reproduce this curve accurately in your report. There are four new Agilent DSOX2012A oscilloscopes, one on each bench, that have USB plugs. You can use them for the measurement and copy the data onto a flash drive. Ask a TA how to use them. (The manual for the scope is in the Handouts page of the ENGN1630 web site and more specific instructions are on the Lab Hints and Comments hyperlink on the same Handouts page.) The 74LS14 and 74LS240 are Schmitt trigger devices and show two thresholds with hysteresis. When you draw the figure for your report be sure to capture this effect. Mark the transition lines for the rising and falling inputs clearly. There may be some difficulty getting good data on the 74ACT04. This is because it is a very fast, high-gain device that may actually oscillate in your breadboard when the input is near threshold. If this happens to you, the line on the scope connecting high and low levels will be blurry and unstable. You can represent that in your report with a fat, blurry line. Be sure to use a bypass capacitor directly between VCC and GND on all devices being measured! On the 74ACT04, be sure to ground any unused inputs too.

2) **Low Voltage Device Logic Levels:** We will have some 74LVC04A inverters available on adapters in the lab. (These adapters are new this year and feature a clamshell socket so the 74LVC04A can be replaced. They are wired for lab 6 so please see Fig. 6-iv for that wiring. Because we only have six of them, I am asking the TAs to distribute them and collect them back as needed.) The main feature of the 74LVC series of parts is that they are designed specifically for use with power supply voltages from 2.0 to 3.3 volts as part of the trend to lower supply voltages, but they may be used with VDD up to 5.0 volts. Repeat the logic level measurement that you did above for the 74ACT04 device except use a variable power supply. Measure the input-output voltage curve for VDD equal to 3.3 volts and again for VDD = 2.0, 2.5, and 5.0 volts. (If you set up correctly for this, the measurement times are trivial.) Plot the gate threshold voltage from this measurement against supply voltage. (You don't have to plot all the I-V curves, just the threshold versus VDD.)

3) **Open Collector Outputs:** You have two chips with open collector outputs: 7406 and LM311. Normally these devices require "pull-up" resistors on their outputs. By measuring the 7406, we explore why this is necessary. With your 7406 powered by +5v, measure the output voltage of an inverter with no pull-up resistor on the output for the input grounded (logical '0') and again for the input connected to +5 volts (logical '1'). Repeat these two measurements with a "pull-up" resistor tied from the output pin to +5v.

What happens to the output voltage when the pull-up resistor is tied to +12v? (DO NOT POWER THE CHIP WITH +12v THROUGH PIN 14! KEEP PIN 14 AT +5v!) .

Can one of the 7406 inverter gates drive another TTL inverter gate properly without a pull-up resistor at the interconnection between them? Try it by measuring the logical state of the output of one inverter of the 7406 with a pull-up resistor when its input is driven by another section without one. Apply logic LOW and HIGH to the input of the first inverter, measur-

ing the output of the second gate. Also measure the input voltage of the second gate (output of the first inverter) for the same conditions. (This data is needed to answer one of the later questions.)

4) **Tri-state outputs:** You have two chips with Tri-state outputs - 74LS240 and 74LS169. With your 74LS240 gates *disabled*, measure the voltages on an output pin when the corresponding input is HIGH and LOW. (Be sure to power the chip with +5v and GND!) Measure how the two output voltages change when you *enable* the gate.

5) **Input DC Currents and the Effect of Unconnected Inputs:** Frequently, systems have unused gate inputs which need to be either '0' or '1' at all times. Making no connection to a pin may seem an economical way to get such a condition, but that can cause unanticipated problems. The purpose of these measurements is to determine the logic level of an unconnected input, if indeed it is determined at all, for a 74LS14 inverter and for a 74LVC04A (CMOS) inverter. The actual input current of an unconnected pin is identically zero. Begin by measuring the input current at full LOW and HIGH levels to see whether an open pin might be at either level. To do this for the 74LS14, connect a 510-ohm resistor from the input to GND and then reconnect it from input to VCC. Measure the voltage across the resistor with the multimeter. Be sure to observe the sign of the voltage (the direction of the current) as well as its value. (DO NOT MEASURE WITH A CURRENT RANGE OF THE MULTIMETER. USE A SERIES RESISTOR AS SHOWN IN FIG. 2-iii AND MEASURE THE VOLTAGE ACROSS IT. Invariably you will end up damaging the meter if you use its current range.) One can also measure the logic level associated with an open connection by measuring the *output* of a gate with no input connected and inferring the input state. For the 74LS14, also measure the voltage at its input with nothing but the voltmeter connected to that input as a rough measure of the unconnected voltage. It is sufficient for the 74LS14 to do these measurements only once.

   The 74LVC04A and similar CMOS chips act rather differently because their DC input currents are much smaller. Also the 74LVC04A is not available in a DIP package so I have had some of them mounted on special sockets and wired in advance for Lab 6. Please see Fig. 6-iv of that lab for the schematic of the wiring. One gate of the 74LVC04A is available by itself for measurements. The other five gates are wired in tandem so connect the input of the first gate of the string to ground for the duration of your work.

   First determine if an uncommitted input on your *particular device* has an apparent definite value. Is it HIGH or LOW? Record the measurement you do to measure it and describe your procedure in your writeup.

   Then, as for the 74LS14, measure the input current of the gate with the input HIGH and LOW. Use VDD = 3.3V, which is a recommended operating condition. The current will be low enough that you will need to use a 10 Meg sense resistor. In combination with the DMM's 10 megohm input resistance, this is an effective sensing resistance of 5 Megohms. (You may also leave out the sense resistor altogether and rely on the 10 Megohms of the DMM.)

Place a 0.1 uf capacitor from input to ground. Connect a meter to read the gate ***output*** voltage. Momentarily connect the input to VDD and see how long it takes the output to change from 0 to 1. Be careful not to touch the input connection when you remove the connection to VDD. (Don't wait too long!) Repeat the process touching the input to ground and wait for it to go HIGH. (Note: the effect you will see is what is exploited to make DRAMs.)

6) **Power Consumption:** Measure the power supply current and calculate the power consumption for your 74LSL14, 74ACT04 and 74LVC04A inverter chips. For the 74LS14 and 74ACT14, do two cases - one with all inputs high, the other with all inputs low. For the 74LVC04A just measure for the single gate, but measure with VDD = 3.3V and again with 5V. As with the device input currents, you will need to use very different values of sense resistor for the 74LS14 on the one hand and the two CMOS chips on the other. I think 100 ohms should be sufficient for the 74LS14 but use a 1 Megohm resistor with the 74ACT04 and 74LVC04A.

7) **Power and Uncommitted Inputs with CMOS Gates:** Connect your 25K potentiometer so it can set the input voltage to the single isolated gate of the 74LVC04A hex inverter using the circuit in Figure 2-iv. Ground the input to the chain of other gates in that package. (See comments above and in Lab 6 on the wiring of this device.) Use the 3.3 volt supply for this measurement. This setup allows you to adjust the input voltage to any value from 0 to VDD and to measure the power dissipation at the same time. Use a sense resistor of 100 ohms. Rotate the pot slowly from one end to the other and record the maximum current into the gate. Make a crude plot of current, IDD, versus input voltage. Estimate the input voltages that make IDD approximately half of the peak value.

8) **Fanout Limitations:** Figure 2-i shows a simplified model of the output of an inverter based on an electrically controlled switch and two resistors. The resistors mimic the change in output voltage with output current. This model is not very good for TTL gates because their transistors do not obey Ohm's law. However, it is a reasonable approximation for CMOS gates. $R_L$ and $R_H$ are the on-resistances of the NMOS and PMOS output transistors respectively. The model is good enough that we will use it for you to find an upper limit on fanout from your 74ACT04 to TTL gates like the 74LS14.

   Measure $R_L$ and $R_H$ for one section of a 74ACT04. Probably the most reliable method is to connect a resistor ($\approx$510 Ω) from output to ground, put the output in the logic HIGH state, and measure the potential difference between the $V_{CC}$ and the output pins. From this you calculate $R_H$ using Ohm's law. A similar arrangement but with the output LOW and the resistor between the output and $V_{CC}$ will give $R_L$. The internal resistances shown in Figure 2-i will limit the amount of current which your chip can handle before noise margins are compromised.

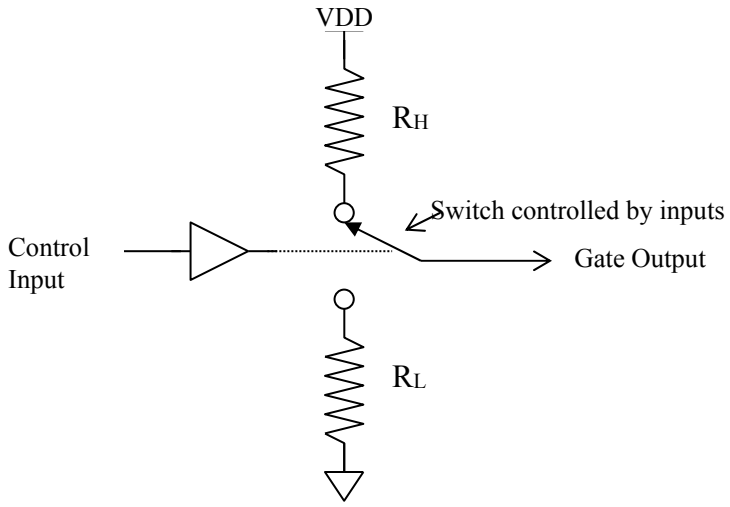



**Figure 2-i: Simplified switch model for the output of a gate.**

## Questions and Interpretation:

1.) From the data from part (1) of your measurements, what is the gate-threshold voltage, $V_{THG}$, for each standard 5-volt gate type? (The gate threshold is the input voltage for which the output voltage has the same value. For Schmitt trigger devices, 74LS14 and 74LS240, there are two such thresholds.)

2.) Assume that the required input voltages for logic HIGH and LOW are $V_{IH}$ = 2.0 and $V_{IL}$ = 0.8 volts respectively. (These are the data sheet standards.) Based on the actual $V_{OL}$ and $V_{OH}$, what are the high and low level noise margins, $\Delta V_H$ and $\Delta V_L$ , for each chip? (Here we are asking about the actual rather than the very conservative guaranteed margins. In this sense, the noise margin is the difference between what is required at an input to which the output of the gate might be connected and the actual value of the gate output when its own input is at $V_{IH}$ or $V_{IL}$. The latter is part of your measurements in part 1.) Finally, for the 74LVC04 operating at 3.3 volts, what are the noise margins if $V_{IH} = 0.3\ V_{DD}$ and $V_{IL} = 0.3 V_{DD}$?

3.) Why are the output voltages of the 7406 so changed by the pull-up resistor? Please be explicit and write a short paragraph in clear English. We are looking for an indication that you understand the roles of the resistor, power supply, and transistor switch. Do not be afraid of Ohm's law as part of the explanation.

4.) This question looks at the properties of the open-collector 7406: What does the data sheet for the 7406 claim is the maximum allowable output voltage? What does the data sheet claim is the maximum allowable *current* that can flow through the output transistor before it may be DESTROYED? How much current flows through a 1K pull-up resistor when it is tied to +12v and $V_{OUT}$ = LOW? What is the lowest value of pull-up resistor that can be used with the 7406 and a +12v pull-up supply?

5.) What logic function is realized if all six 7406 outputs are connected to the *same* 1K pull-up resistor? Use DeMorgan's theorems to express this result in two forms.

6.) If an open collector device with no pull-up resistor drives a TTL input, does the circuit work at all? What is the noise margin of such an interconnection when the interconnection point itself (the output of open-collector inverter and input of the second device) is in the HIGH state? (To answer this sensibly, you need the measurements of this situation that you did for part (2) above. The answer is the difference between the measured input without a pull-up resistor and the measured input that will make the output 0.8 volts.)

7.) Was the state of an unconnected input for your particular device well determined? Did it remain so when you put a small capacitor across the input? Is the state of an unconnected CMOS input generally well determined in a product when the product may be built with components from different vendors? Why? (Remember that a MOSFET gate draws almost no steady current. The only current at a CMOS chip input is the small, random, temperature-sensitive current of its static protection circuit. The direction and magnitude of this current depends on manufacturer, input voltage, temperature, and conditions during manufacture and is not guaranteed.)

8.) The power used by the 74LVC04A as measured in part 7 depended on the voltage at its input. Why? Please be explicit and refer to the usual inverter circuit to explain this very large effect. How much power did it draw in one gate for the worst-case input? How much was the static input power, that is, the power with the input either grounded or connected to VDD? What's the ratio of the two conditions? Is it generally a good idea to leave CMOS inputs unconnected when the input might drift to something in the middle of the input voltage range?

9.) Draw a circuit diagram showing how to configure your 74LS240 as a 4-bit wide, 2-to-1, multiplexer.

10.) For the worst-case input and output currents listed in the 74LS14 data sheet, calculate the maximum fanout for a 74LS14 output projecting to other 74LS14 inputs.

11.) How many regular LS-TTL inputs can one of your 74ACT04 (CMOS) outputs drive high *and* low? Assume that the output has to be limited to 0.2 volts for low and 3.5 volts for high. Use the worst-case input currents from the data sheet for the 74LS14 and your measured values of $R_H$ and $R_L$ for the 74ACT04. (You may well find the answer is quite large. Other factors including capacitive loading and wiring inductance generally make it unwise to try that large a fanout.) As a *Challenge to the Bored*, use the fact that the circuit must work at 150 deg. C and that the resistances of the CMOS gate increase as $T^{1.5}$, to calculate the fanout limit over temperature. ($T$ is the absolute temperature and at 25 deg. C. $T$ =300 deg. absolute.)

**Discussion:** You should be able to make most of the required measurements with one of the digital multimeters available in the lab. (If the multimeter you pick up has a dead battery, please bring it to the attention of a TA. Turn off meters that are not in use.) The I/O voltage relationships require one of the 100 MHz scopes. Four of these scopes are set up so that a trace can be recorded and stored in an EXCEL file on a flash drive to make recording easier.

The current edition of your text, Wakerly, has a nice discussion of the properties of various gate circuits in chapter 3 including extensive coverage of TTL gates in section 3.10. We will spend much of the class time available for discussing gate characteristics in talking about CMOS circuits.

**Noise Margins:** All logic systems have significant amounts of noise - unwanted and unintended voltages added to the real signals of the system. The noise is usually induced by electromagnetic coupling between the different signal wires. While it can be minimized, it cannot be eliminated. Much of the appeal of digital systems comes from their relative immunity to such noise. Figure 2-ii shows a circuit model for what is going on as a noise voltage adds to the output of a gate, U1, before the signal reaches the input of gate U2. With well-designed gates, the voltage at the input of U2 which is necessary for U2 to recognize a logic LOW, $V_{IL}$, is more than the output of U1 in its LOW state, $V_{OL}$. For U2 to function properly we must have

$$V_{OL} + |V_{NOISE}| < V_{IL}$$

The maximum noise which will still result in this inequality being met is the *low level noise margin*, $\Delta V_L = V_{IL} - V_{OL}$ . There is a similar consideration when the node is in the HIGH state for a *high level noise margin*,

The standard specification for TTL circuits assumes that $V_O$ will always be less than $V_{OL}$ = 0.4 volts in the low state, that $V_{IL} \geq 0.8$ volts, and that $\Delta V_L$ is then 0.4 volts. Similarly the output high voltage is 2.4 volts minimum, the input high level is 2.0 volts and $\Delta V_H$ is 0.4 volts too. Now all these numbers are very conservative since they have to reflect the worst possible variations of manufacturing, temperature changes, power supply variations, etc. In this experiment you are looking at how much extra margin there may actually be in more typical cases. (This is not to suggest that it is okay to build things with more than rated noise levels!)

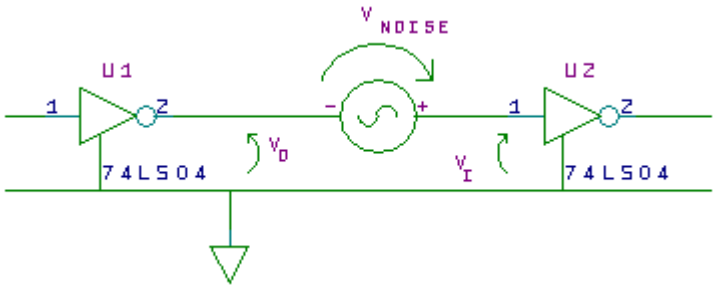



**Figure 2-ii: Model of how noise adds to a signal between gates**.

**Threshold Voltage:** Except for gates with Schmitt trigger inputs like the 74LS14, there is an input voltage for which the output voltage is equal to the input. This is the "gate threshold voltage" $V_{TH}$ or $V_{THG}$. (In practice this may be hard to measure if the circuit oscillates for inputs near threshold as your 74ACT04 may.) It is useful as a rough measure of the value of input at which the output is most dependent on the input. It or an average data sheet value for it is often used as a reference level for timing measurements on the assumption that this is the value of input at which most electrical activity inside a gate actually happens. Nothing much happens inside a gate until its input approaches this level, so it makes sense not to count the time it takes for input to rise to this level as

part of the propagation time through that gate. Similarly, its output must change to a value near gate-threshold before anything will happen in a subsequent gate. The propagation time through the gate should not be counted as over until the next gate begins to respond to its input. Therefore the time between input and output threshold crossings is a good simple measure of propagation time.

The 74LS14 and 74LS240 have what are called "Schmitt-trigger input circuits". They do not have an equilibrium condition with equal input and output voltages. Instead internal circuitry sets two different levels of input at which the output will go abruptly from high-to-low or vice versa. The two levels are distinguished by the initial state of the gate. Thus there are effectively two different thresholds depending on the transition direction. The advantages of such an arrangement are a much higher noise margin and an ability to respond sensibly to slowly changing input signals. These devices are often used as interfaces between slow and fast systems.

**Fanout:** Fanout is the number of connections to other devices from a given gate output. There is a maximum number of such connections that can be made if the output logic levels and the transition speed are not to be affected by the load. With TTL gates, the primary limitation is the DC current drawn by the gate inputs. Suppose you are attempting to keep an output at a high level. The more inputs that one output projects to (fanout) the more current is required to leave the output pin to supply the various inputs. The addition of too many output loads (i.e., inputs to which the output is connected) will cause so much current to pass from the power supply through the gate that the voltage at the output will drop *below the high level required by the various inputs* or at least below an acceptable noise margin. A similar argument with $I_{OL}$ and $I_{IL}$ currents can be used to calculate a fanout for the *low level* output voltage. The lesser of the two fanout numbers is the "worst case" fanout. This value is a direct and rigid limitation on the number of inputs a designer can connect to one output.

Since more current is required to change a logic level than to maintain it, fanout is often limited to less than DC measurements might suggest. (The reason for the difference between the static and dynamic currents is the input and wiring capacitances which have to be charged or discharged. The charge in those capacitances contributes to the load current of the gate only as it is changed, "i.e." during transitions.) In fact, chips which are built from MOSFET transistors (such as your 74ACT04 and MCM6268P55 chips) have negligible static input currents, and so the only limitation on fanout comes from the dynamic load, the current associated with changing output states. In this case, fanout is limited by the acceptable speed of the system.

**Input currents:** Gate currents of MOSFETS are invariably below $10^{-14}$ amps. If you measure *any* significant input current, it is likely to be from circuits that protect the input from being zapped by electrostatic discharge during handling rather than the gate input current itself. These input currents are very temperature sensitive, are not specified in the data sheet, and may actually go either direction. Different manufacturers make nominally the same device with different protection circuits. The input currents vary in magnitude and direction from vendor to vendor. Given the current can go either direction, is the input " well determined" in the sense of being the same for all devices regardless of manufacturer?

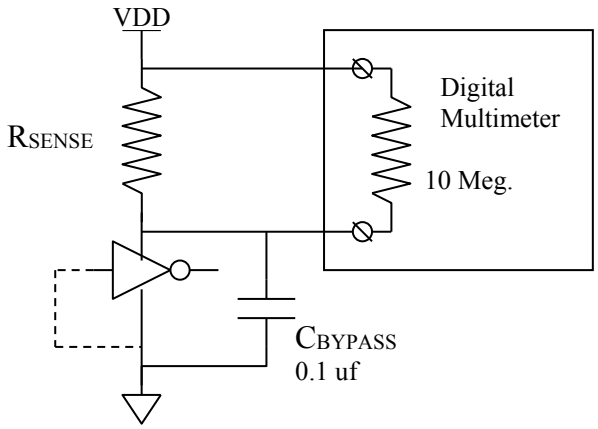



**Figure 2-iii: Scheme for measuring the supply current of a gate/integrated circuit.** The size of the sense resistor has to be chose so that the voltage drop across the resistor is less than 0.2 volts but great enough to be measured. The multimeter has 10 megohms input resistance so combined with a 10 megohm sense resistor the current is really passing through 5 megohms equivalent.

**Power:** The power used within a chip is the product of the current *through* the chip from its $V_{DD}$ or $V_{CC}$ terminal to its GND terminal times the voltage *across* those terminals, that is: Power = (current through) x (voltage across). This is the rate at which the power supply is doing work (mostly heating the chip) on the circuit. A flow of current can only be measured ***by routing it through something***, typically the ammeter part of a DMM. However, we have found that invariably students damage meters they try to use in this way. Instead for this lab you measure current by putting a resistor "in series" with the supply and the $V_{CC}$ or $V_{DD}$ terminal. Be sure to have a bypass capacitor across the integrated circuit itself as shown in Fig. 2-iii. The CMOS chips require a very large resistor, 10 Megohms. However, if you connect a 10 Megohm resistor in series with the $V_{DD}$ terminal, as shown in Figure 2-iii, and use the DVM on a *voltage* range to measure the potential drop across the resistor, then Ohm's law can be used to find the current. The resistance of the meter itself when used as a voltmeter is 10 Megohms. Since this is not negligible compared to the 10 Megohm sensing resistor, one has to use the effective resistance of the combination of the two in parallel, namely 5 Megohm. The corresponding voltage across the chip is the power supply voltage minus the drop across the series-measuring resistor. **WARNING:** Do not try to measure the voltage across the inverter and infer the current through it by Kirchoff's voltage law. The multimeter current is not negligible compared to the CMOS current.

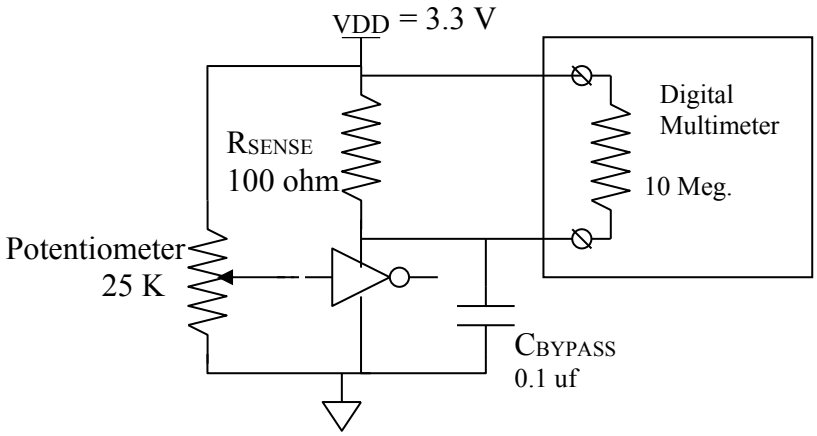



**Figure 2-iv: Measuring Power Supply Current and Power as a Function of Gate Input Voltage.** Turning the potentiometer sets the input voltage to any value between 0 and VDD. There will be a maximum current for some value of input in the excluded middle.

### 9.2.4. Lab Three

**Multiplexed Display**

**Requirements:** Program one of the Engineering 163 CPLD boards to act as a seven-segment decoder for its on-board display. (Please see the Appendix section 10 and subsections of lab 5 for information on this board and for notes on the programming procedure for the XC9572XL on the board.) The resulting characters should have unique proper forms for all 16 possible inputs as shown in Fig. 3.i. Using the discrete parts from your kit, wire your 8-bit DIP switch so that the upper and lower sets of four switches control the left and right hexadecimal character of the on-board display. Use pull-up resistors as appropriate. (You may NOT use pull-downs!) Arrange a circuit with a toggle switch such that in one position of the toggle switch the display shows the two numbers set on the DIP-switch with no flicker in either digit. The display should update immediately when the DIP-switch setting changes. When the toggle switch is in the other position, the display shall show "A" on the left digit and the right digit shall be off entirely. I realize that in principle **this multiplexing could be done in the CPLD but that is not allowed** for this lab, which aims to show you how bus multiplexing is customarily done. The only connections to the user cable of the CPLD board are to the two cathodes, the four data lines, and two power connections. If you need more pull-up resistors than are available in your kit, you can get those in the lab.

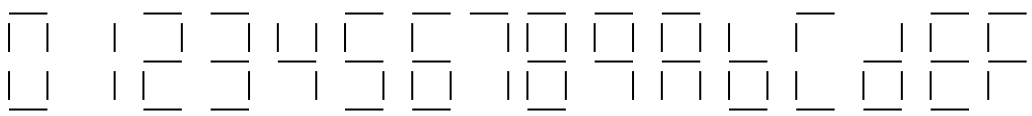


**Figure 3.i: Hexadecimal characters on a 7-segment display**

As with all labs requiring programming, you **must have a hard copy of the program** for your device ready for the TA. FTQ's may require you to predict the effect of a change to that code, and you must show mastery of the programming sufficient to carry the suggested change through to a demonstration on your hardware.

**Discussion:** Figure 3.ii shows how the toggle switch connects between terminals in its two positions.

To use the CPLD Board's display, you need to determine the pinout of the display chip and how they are connected to the CPLD itself. The way to do this is to consult the schematic for the CPLD board in section 10 of this manual and infer the pinout from that.

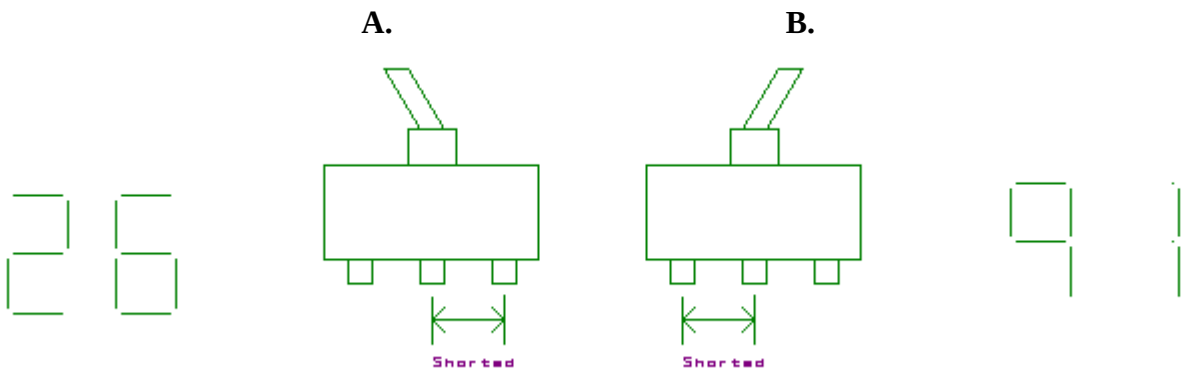



**Figure 3-ii: Toggle positions and resulting connections.**

**<u>WARNING!!</u> When you start to use a CPLD board for the first time, you *<u>MUST PROGRAM IT BEFORE YOU CONNECT IT</u>* to your protoboard. The CPLDs are easily damaged if an output pin is connected to a signal source, as it might be if a prior user had different pinouts programmed than you use. To power up the board for programming there are special power cables that connect to the ribbon cable plug on the board. Ask a TA!**

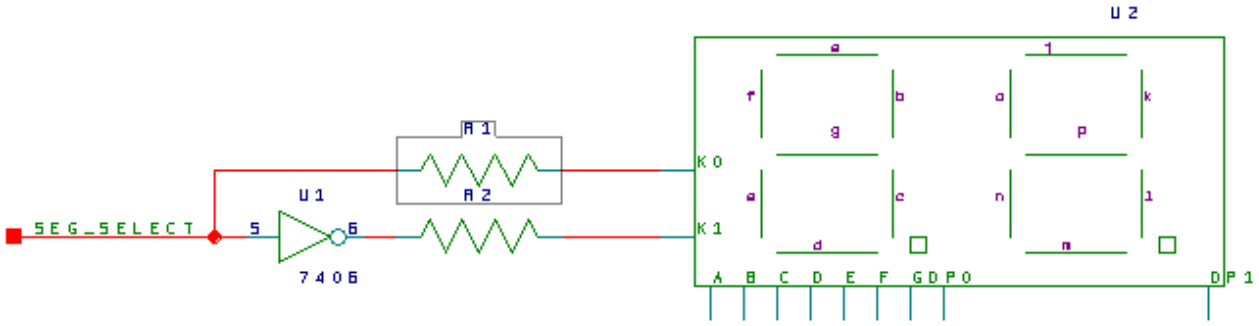



**Figure 3-iii: Cathode drive circuit with current limiting resistors; R1 and R2 are already on the CPLD Board. The 7406 is not on the board, and the reason for its use is that the required display current is larger than the XC9572XL output rating.**

You want a flicker-free (>50 Hz) rate of switching back and forth between the halves of the display. In one phase the left digit will be on for a few milliseconds while the right digit is off, in the second phase the right digit will be on while the left is off. Remember that a display digit is ON when the cathode pin for that side is grounded by a LOW output on the logic gate driving it. Also remember that the current required to drive all segments of the display is too large for a simple TTL gate and is marginal for the XC9572XL. I show a 7406 inverter in this application because they are capable of sinking 40 ma and the display requires 40 ma for an "8" with decimal point. To control switching, you can build a pulse generator using the NE555 timer chip to drive one side. An oscillator example circuit is included in the data sheets at the back of this manual. The output transistor of the NE555 can "sink" the current from one digit, while an inverter derives the signal to switch the other digit.

Once you are able to switch the two digits off and on rapidly, you will need to coordinate that action with the proper signals for the segments. You will need to *multiplex* the four inputs to the XC9572XL with three different signals. You will want the oscillator and the output of your toggle switch to control that multiplexing.

See Lab 5 and section 10 of this manual for more information on the CPLD-II board, the Xilinx schematic capture process, and downloading procedures.

### 9.2.5. Lab Four

**Keyboard Encoding**

**Requirements:** Arrange a circuit such that each of 16 different buttons on your keyboard represents a different hexadecimal number on one digit of the LN524RK display on a CPLD board. (Consult Lab 3 for further information about these displays.) It is not necessary that the displayed digit correspond to the key marking, but that would be nice. Have only the decimal point and nothing else light when no button is pressed and have the decimal point go off when anything is pressed. You may use any amount of combinational logic in the CPLD that you wish, but any flip-flops you need must be external. (This circuit may use a counter of some kind and I want you to assemble that in some fairly primitive way to learn a bit about counters.) As usual, you must have a current schematic and a hard copy of any program you use when you ask for evaluation. You must design your circuit so that it is safe for any number of buttons to be pressed simultaneously. Remember that pressing more than one button at a time may connect gate outputs together, a condition called output contention. Such contention can cause unacceptable current and power levels when gates pull against each other. That must be avoided. The TA will want to know how you have prevented contention by design as a separate issue from any FTQ.

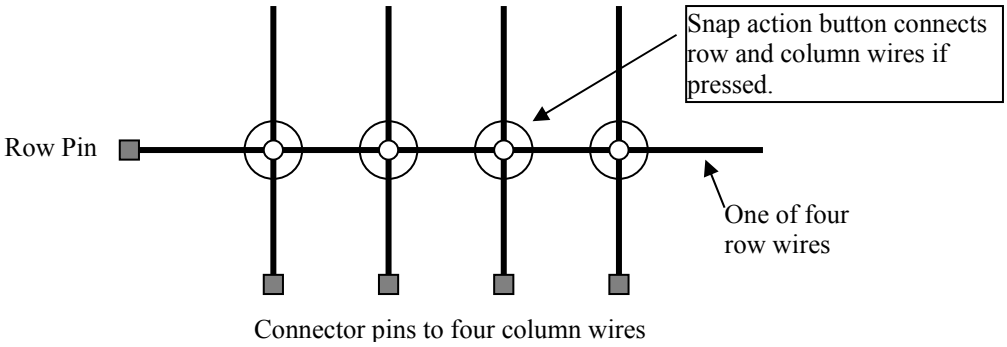



**Figure 4-i: Connecting the row and column wires via a button press.**

**Discussion:** Your keyboard has a matrix of 4 by 4 wires (8 wires total). When a button is pressed a unique pair of row and column wires is connected (shorted) together. (See Figure 4-i.) You will need to devise a method for determining which pins go with which row or column. One possibility is to use the ohmmeter of the DVM from the Lab 2 set-up.

You may want to build a circuit that *scans* the keyboard by sending test signals to the columns while the outputs of the rows are analyzed. For the fault tolerance question we may ask you to predict what response your circuit will give when *two* buttons are pressed simultaneously and why! Read section 5.7 of Wakerly about multiplexing and demultiplexing.

A word of warning: the keypads are easily destroyed by melting the button supports with too much current. The commonest way this happens is to use a button to short the power supply from VCC to GND or to a TTL output that is low. **DO NOT CONNECT A KEYPAD WIRE TO VCC EVER!** Use 1 K pull-up resistors on either the row or column wires and let the keypad pull down to GND or to the output of a gate.

### 9.2.6. Lab Five

### Counter with External Control

**Requirements:** This lab must be built with the Xilinx XC9572XL logic chip on the class CPLD-II board. This time you use the flip-flops inside that device. (Programmable logic is very important in prototyping and small-volume production. Here is where you start to learn its full capabilities. I have chosen the Xilinx parts because they are inexpensive, Xilinx supports us with software very well, and these devices are currently popular ones of their kind. Other vendors, particularly Altera, Vantis, and Lattice, make similar product.

Design and build a counter circuit that is clocked by a switch or pushbutton with no skipped steps. The output of the counter should be displayed on the seven-segment display on the CPLD board. The system has the following features: There are two control signals that you can generate with two sections of your DIP-switch. One such signal is called "Abbreviate" and the other is called "Halt." When the clock switch is flipped back and forth with *no* control signal asserted, it is in a "continue" mode, and the display steps through the count 2, 3, 1, 6, 0, 2, 3, 1, 6, 0, 2, etc on the right display digit. When the "Abbreviate" signal is asserted while the counter is clocked, the sequence cycles normally up through the number "2". On the next clock, the counter should begin the 'abbreviated' sequence 6, 7, 6, 7, 6, etc. The abbreviated sequence, including the first 6 should display on the left digit! If the system is displaying anything except 2 on the right digit at the time "Abbreviate" is asserted, it should continue through the entire normal sequence on the right digit until it reaches 2 before going to the abbreviated sequence on the left display digit. If the Abbreviate signal is deasserted in state 7, the system should go to 6 on the left digit before resuming its normal sequence. If the Abbreviate signal is not asserted in state 6 on the left digit, the system should go to 0 (on the right digit) next and continue its normal sequence. While the "Halt" signal is asserted, the counter should not change state when toggling the clock switch any number of times. Similarly, if the clock switch is left in one position, the output of the counter should not change when the "Halt" signal is toggled, regardless of which position the clock switch is in.

**Challenge to the Bored:** If you accept the challenge, the 6, 7, 6, 7 abbreviated sequence should count slowly (about once per second) by itself without your needing to pump the toggle switch. The toggle switch will now have no effect. When the abbreviate signal is asserted and the counter has toggled up to 6, the first 6 on the left digit must display for at least a second, maybe longer, and not be skipped as the steady automatic counting starts. "Halt" also stops the automatic counting in the abbreviated count mode. (The point of this subsidiary exercise is to get you thinking about the problems of switching clock sources. You may exploit the relative slowness of the clocks in this particular problem.)

You ***may NOT use the STATE MACHINE*** facilities of the Verilog language or the programming software for this lab. Within Verilog, you may not use "if" statements or the "?" operator; "case()" may only be used for truth tables for the display segment decoding; and "assign" statements should be used for the state logic. (One of the major pedagogical purposes of the lab is to make clear to you what state machines are all about. Automating the synthesis completely

would defeat that goal.) You must have a printout of your data entry file for the XC9572XL for the TA. The TA may base her FTQ on a change to that file. (Remember: that file must be your own work -- original design and actual typing. Your fingertips!) You will still have to wire up the input circuitry (including the debouncing circuit for the toggle switch) and the cathode drive connections. (And any circuitry to support the Challenge to the Bored if you are bored.)

## WARNING!! The CPLD boards are easily damaged by two things:

## 1. Connecting an output pin to a signal source, as it might be if a prior user had different pinouts programmed than you use.
## 2. Connecting an input or output pin to +5 volts when there is no power on the VDD pin of the CPLD. This can happen if you plug or unplug the board with power on.

**As a result you must use these rules:**

**1. WARNING!! When you start to use a CPLD board for the first time, you *MUST PROGRAM IT BEFORE YOU CONNECT* it to your protoboard. The CPLDs are easily damaged if an output pin is connected to a signal source, as it might be if a prior user had different pinouts programmed than you use. There are special power cables on every supply in the lab to power up the board just for programming. Connect that cable to the ribbon cable plug on the board. Ask a TA!**

2. **Be SURE THAT VDD IS CONNECTED before turning on power. (The VDD pin is pin 24 on the DIP connector.)**

**3. Do NOT UNPLUG the cpld board before the power has been turned off and the output capacitor is discharged. This takes about 5 seconds after you turn off the power.**

**4. Do NOT CONNECT INPUTS TO VDD = +5 volts directly!! If you want a HIGH on a pin, use a pullup resistor of 1K to 20 K. Works fine and limits current into the pin to harmless levels. You have plenty of resistors.**

**5. Do NOT USE A SWITCH TO PULL AN INPUT UP TO VDD! Always use a pullup resistor and use the switch to pull down.**

**Discussion**: Debounce your toggle switch or clock button so the counter does not skip numbers as it sequences. The introduction of this manual describes two possible procedures for preventing false clocking by the switch. Read chapters 7, 8, and section 9.1 of Wakerly for a discussion of flip-flops and counters. Do your design systematically with the usual tools for finite state machine design. *Ad hoc* methods may work, but you should avoid them as they are basically dead ends. As usual, we will ask you a Fault Tolerance Question and it may be based on the code in your CPLD. You **must have a hard copy** of that code available to the TA at the time of evaluation and that copy must be current.

Programming the CPLD requires telling appropriate software how the flip-flops and gates of the device are to be configured. The two largest vendors of programmable logic, Altera and Xilinx both have software that accepts schematic diagrams or files of VHDL or Verilog hardware description language designs. In the United States, the most common HDL is Verilog and so we will use that here.

The point of this lab is to understand what is being done in constructing a state machine and so we insist that you limit the use of Verilog to simple Boolean statements and some D-flip flop instantiations. The mainstay of your next state logic should be "assign" statements. You must have your code available when getting the lab checked off and the TAs are asked to check that you are not using any "if" or "?" constructs. You may only use "case" statements for Boolean truth tables for purposes other than next-state logic.

The following example shows the syntax of a Verilog file and is annotated to explain some of its features. It is compiled into JEDEC files (".jed" extension) to encode the pattern of gate charges needed to program the device. That data is downloaded into the XC9572XL through a JTAG port while the device is actually hooked up to your protoboard. (Lab C is about the JTAG standard and the ideas behind it and there are references given in that lab that explain the idea further.) Suggestions for HDL data entry come after the example.

The inclusion of a comment block at the beginning with your name, the date and a circuit name is *NOT OPTIONAL*. You must include such a section properly filled in. Again, the TAs are asked to check this.

### 9.2.6.1. Verilog Implementation of a Counter – An Example

```
//-------------------------------------------------------------------------------
//
// Title      : ctrcpld2010
// Design     : ctrcpld2010
// Author     : William R Patterson
// Date       : 9/12/2010
//
// Description : Example of a 2-bit up/down binary counter with halt written in Verilog
//                    Written with simple boolean statements  and d-ffs only.
//-------------------------------------------------------------------------------

// module statement is standard beginning of a block
module ctrcpld2010 ( mclk ,oe_n ,halt ,up ,rolling ,going ,ctrcnt );

output [1:0] ctrcnt ;  // Output port connections
wire [1:0] ctrcnt ;
output going, rolling ;
wire going;
reg rolling;          // “reg”  required because “rolling” originates in an "always" block

input mclk ; // Input port connections
wire mclk ;
input oe_n ;
wire oe_n ;
input halt ;
wire halt ;
input up ;
wire up ;

// Signals used within module
wire [1:0] ctr_d; // Plain vanilla signals on wires
reg [1:0] count;  // Signal type "reg" is not generally the output of a register, but it is in this case.
          // The "reg" type is required for any signal originating in an "always" block of sequential logic.

assign ctrcnt = (oe_n) ? 2'bz : count;  // Tristate output drivers. To be synthesized, the pin
          // assignment for oe_n must be on a tristate enable pin.  Xilinx .ucf file sets pin numbers
assign going = (oe_n) ? 1'bz : ~halt;// Tristate outputs only available at the package pin level

assign ctr_d[0] = (halt & count[0]) | (~halt & ~count[0]); // Counter boolean logic
assign ctr_d[1] = (halt & count[1]) | (~halt & ((up & (count[1]^count[0])) | (~up) & ((~count[1] & ~count[0]) |
(count[1] & count[0]))));

// d-ff's defined by behavior
always @(posedge mclk) count <= ctr_d;

// Combinational "always" block using case() to implement a truth table – not allowed for next-state logic in lab 5
always @*          // Implicit sensitivity list
          case(count)
                    2'b11 : rolling = 1'b1;
                    default : rolling = 1'b0;
          endcase

endmodule
```

An alternative way to write the module port names and types is:

```
// module statement is standard beginning of a block.  This time using a more C-like format

module ctrcpld2010 ( input mclk ,
            input oe_n ,
            input halt ,
            input up ,
            output reg rolling ,
            output going ,
            output [1:0] ctrcnt );          // "wire" is the default type of ports without a type
```

### 9.2.6.2. Assigning Pin Numbers to Programmable Devices

There is no provision for assigning pin numbers on a programmable device within Verilog. The language started as a simulation language and grew to encompass synthesis, but the synthesis is used for all kinds of systems, many of which don't have pins *per se*. The ways pins are assigned depend on decisions by the device manufacturers who generally have to supply routing software because the internal bit patterns for these devices are proprietary data.

Xilinx recommends that users make a ".ucf" file (Universal Constraint File) for each of their devices in your design. These files are simple ASCII text files that can be used to assign pin locations, timing requirements, logic voltage level assignments, and several other properties. The Xilinx tool suite provides a number of different pieces of software to generate these files. However, for CPLDs like the XC9572XL you use in this lab that only need pin numbers, the file syntax is trivial and a file entered manually works very well.

Use a standard text editor (Notepad and Wordpad on Microsoft machines work perfectly well) to create the file. Save the file in the directory with your Verilog file, usually a subdirectory of your U:\Xilinx directory. The instructions below for creating and downloading your design include how to link the UCF file to the design. The syntax of the UCF file for this lab is trivial. Here is an example from the file I made for the display used in lab 1.

```
# Beginning of ".ucf" file for the display driver used in Lab 1
# "#" is the comment marker in a UCF file
# LOC is the attribute for pin position
NET "q(3)" LOC = P11;
NET "q(2)" LOC = P14;
NET "q(1)" LOC = P13;
NET "q(0)" LOC = P12;
NET "seg(6)" LOC = P33;
NET "seg(5)" LOC = P29;
NET "seg(4)" LOC = P26;
NET "seg(3)" LOC = P27;
NET "seg(2)" LOC = P28;
NET "seg(1)" LOC = P25;
NET "seg(0)" LOC = P24;
# End file
# Addendum comment: When a signal is to be a clock on a clock
# network in the device, the line:
# NET <my_clock_name> BUFG=CLK;
# will guarantee to assign that net to a clock network.
```

### 9.2.6.3. Editing and Compiling Verilog Files for Xilinx Parts

The very first step in preparing to use the Xilinx XC9572XL parts is to create the directory system for storing and manipulating the necessary files on your U:\ drive on the Engineering computing facilities. DO NOT STORE ANY DESIGN FILES ON THE D:\ drives in the lab. Such files are may be discarded at any time. They are also public and you should not be sharing electronic files that way.

Keep your Xilinx files in a working directory at U:\Xilinx_Projects. Create that directory before you begin to use the software by either using the File/New menu command in Windows Explorer or the instruction: "mkdir U:\Xilinx_Projects" in a command prompt window.

We have three Verilog-aware editing tools, two of them from Xilinx and Altera, are aimed at their respective products while the third from Aldec, Inc. is designed to target any programmable logic. We will ask you to use the Aldec software for the simulation in Lab 9 and recommend it for the later FPGA labs because of its block diagram entry system and state machine tool. However, for the first few labs using the XC9572XL parts, it is simpler to do all design entry within the Xilinx ISE tool suite. The steps are:

1) From the Start menu of Win 7, choose "All Programs/Xilinx ISE …./ISE Design Tools/64-Bit Project Navigator". In *Project Navigator*, select the File/New menu sequence or hit the "New Project" softbutton to open a project dialog. Supply your Verilog file name (without the ".v" extension) as the project name, change the project directory name to "U:\Xilinx_Projects" and select "HDL" as the "Top-level Source Type. Remember that the file name must match the Verilog module name and that the module name is CASE SENSITIVE. The use of uppercase letters in file and module names is deprecated for Verilog usage.
2) Click the Next softbutton and set up the project's configuration as: Product Category = General Pupose; Device Family = XC9500XL CPLDs; Device = XC9572XL; package type = PC44; Speed Grade = -10; Synthesis Tool = XST (Verilog/VHDL); and Simulator Tool: Other. Hit "Next".
3) If you have not already created your Verilog file, hit the "New Source" softbutton on the next screen. Enter the name of your module and use the table to enter the names of your input and output nets. Buses require checking the "bus" box for the port and adding the range of net numbers. ALWAYS list wire numbers from high to low.
4) Write the ".ucf" file for assigning pin numbers with Notepad or Wordpad. Store it in your Xilinx working directory. In the next dialog, attach that file to your design by hitting the "Add Source" softbutton and browsing to it. Hit Finish several times to bring you back to the basic Navigator page.
5) There will be a set of panes on the left side of the page with tabs beneath them. Be sure the "Design" tab is selected so that the panes show Hierarchy and Processes. In the Hierarchy pane, double click on the name of your Verilog module, not the UCF file. This opens a Verilog-aware editor in which you enter your actual code. Edit in your logic description.
6) By default the Xilinx software enters the port names of your input and output connections using a "C"-like format. This must be modified to add the "reg" attribute to any output that requires it. Alternatively, you can change the port name list to the format used in the exam-

ple in section 9.2.6.1. The lines for internal net names and properties are still required in the format of 9.2.6.1.

7) When you have entered your Verilog code, click on the name of your module in the Hierarchy pane and then in the Processes pane expand "Implement Design" and double click on "Synthesize XST". This will start the compilation. Correct any errors that the compilation finds by appropriate editing and rerunning the synthesis.
8) When you have successfully run the synthesis, double click on the "Implement Design" entry itself. This will complete the process of mapping the design to the CPLD.
9) Implementation should include running Generate Program File. This step produces a JEDEC industry standard file (with file extension ".jed") describing how the device will be programmed. At this point, the next step is to use the downloading cable in the lab to put your program into a CPLD board. See section 10., particularly for download instructions and Table 10.1 for connectivity information.

### 9.2.6.4. Alternative Tutorial for Xilinx CPLD Programming with Verilog

**Stage 1, Setting up your project**

1. Open the ISE Project Navigator (Start -> All Programs -> Xilinx ISE Design Suite 13 -> ISE -> Project Navigator)
2. Close any current project (File -> Close Project)
3. Create a new project (File-> New Project...)
    a. Choose a name for your project (ex. lab03Verilog)
    b. Make sure the location is on your "U:\" drive (the "D:\" Drive gets wiped from time to time and is public readable so DO NOT save your work there) (ex. "U:\Lab03Verilog\")
    c. The working directory can be the same as the location (ex. "U:\Lab03Verilog\")
    d. Top-Level source type should be set to "HDL"
    e. Click Next

4. Set up the properties (some of these may work with other values, but these are the settings I know work)
    a. Product Category: All
    b. Family: XC9500XL CPLDs
    c. Device: XC9572XL
    d. Package: PC44
    e. Speed: -10
    f. Synthesis Tool: XST (VHDL/Verilog)
    g. Simulator: Other Verilog
    h. Preferred Language: Verilog
    i. Property Specification in Project File: Store All Values
    j. Manual Compile Order: unchecked
    k. Enable Enhanced Design Summary: checked
    l. Enable Message Filtering: unchecked
    m. Display Incremental Messages: unchecked
    n. Click Next
5. Create a new source file (This will be the file in which you write your verilog code)
    a. Click "New Source..."
    b. Select Verilog Module and give it a name (ex. lab03Verilog.v)
    c. Click Next
    d. Click Next (you can fill things in here, but it is not necessary, more on this later)
    e. Click Finish
    f. If prompted to create a directory, select Yes
    g. Click Next
6. Add a source file for constraints. (This is the file that stores your pin assignments.)

- Open your favorite text editor (Notepad will do, but be sure to save this file with the extension .ucf not .txt.)
- Create a line for every pin you need to assign on the chip with the format as follows: NET "<Verilog Variable>" LOC = P<Pin Number> Where <Verilog Variable> is replaced with your variable name. Do not include the greater/less than signs. And Where <Pin Number>

is replaced with the CPLD's pin number you want to associate with that variable (Note: this is not the same as the ribbon cable pin number, see page 111 in this lab manual for a conversion chart. Use the columns DIP Plug (Protoboard) and Pin on XC9572) See the end of Lab 5 for more details on the pin assignment file.

- Save this file in your project directory (in this example it would be "U:\Lab03Verilog\") with a file name of your choosing (ex. "Lab03PinAssingment.ucf")
- Close Notepad and Return to the ISE Project Navigator
- Click Add Source
- Select the file you just created in notepad (in this example "U:\Lab03Verilog\Lab03PinAssignment.ucf")
- Click Next
- Click Finish
- Wait a while
- Click OK if a dialog box about adding source files comes up
- Wait
- Smile, you have set up your project, and you are ready to program your Verilog!

**Stage 2, Programming Verilog: The very, very basics.**

1. Double click on your Verilog file
2. Add your variable names and types inside the parenthesis after the module name see page 55 for an example. For the early labs you will only need outputs and inputs. "Reg" refers to registers (or flip flops) brackets are used to define a bus (an array of ports or variables).
3. Add the logic you want to compute between the module statements:
   | = or
   & = and
   ^ = xor
   ~ = not
   Make statements like "assign d =( a & (~b)) | c;" where a, b, *c,* and d are variables, a.k.a. ports. Statements end with a semicolon.
4. Save your work
5. See page 57 and 58 of this lab manual for details
6. NOTE: if you need to update your pin description file, just do it in notepad because I can't seem to get it to open in the ISE editor. Any updates in notepad will reflect through to the editor so don't worry about re-importing or closing and reopening or anything.

**Stage 3, Compiling Your Code and Loading it onto a CPLD**

1. Select the Verilog file in the design pane (which is on the left if you haven't messed with the default view)
2. Below the file list in the design pane, click on the + next to Implement Design
3. Right click on Implement Design and click "Rerun All"
4. Wait a while. This can take a bit so sit back and relax for a moment
5. If anything else appears (e.g., a yellow ! or a red X), go through the error messages in the console window and debug your code until it compiles properly.
6. Right click on "Configure Target Device" and select "Run".
7. Click OK for the warning that comes up.

### 9.2.7. Lab Six

**Propagation Delay Measurements**

**Requirements**: Using a high-speed oscilloscope, measure the timing properties and power dissipation of several basic logic gates, including two from your bag-of-chips. Also, probe the behavior of an SN74LVC04 chip as a function of power supply voltage so you will see what lowering that voltage does to speed and power. The SN74LVC04 chips are only available in surface mount packages so they cannot be wired on your breadboards. Instead we have mounted them in sockets pin connectors that plug into your boards. Those leads can be connected to a power supply and scope via your breadboard. See the discussion section below for the wiring details.

*Hand in a report* describing your results, in the spirit of Lab 2. The report must be **typeset** and should include three things: first, an introduction explaining in your own words why these measurements are of interest to a system designer. In your discussion, distinguish between your measurements and the *worst-case* conditions for the same values. Second, give the results of your measurements - clearly labeled and tabulated with units. Finally, answer the questions that follow the measurements. Make sure you see a TA if you are unfamiliar or uncomfortable with the use of an oscilloscope.

There are several instruments that you may choose from for this lab. The 100 MHz digital oscilloscopes at each workstation are quite satisfactory for most purposes but may make some transient events appear slower than they really are. There are now also four 200 MHz digital oscilloscopes spread around the first three benches. These are probably the most satisfactory for the purpose since they are fast enough to see most of the circuit behavior and allow to record your results to a USB flash drive easily. There is also a 500 MHz scope on a rolling cart that will resolve even more interesting behavior in your circuit but it is more delicate and can be difficult to hook up without introducing artifacts. Finally, there are scopes built into some of the logic analyzers and these have time resolution similar to the 500 MHz stand-alone scope. The stand-alone instruments have advantages in dynamic range and storage depth but the logic analyzers let you do triggering on logic conditions. You are free to choose among these tools, but I **strongly recommend using the 200 MHz scopes** for the setup and hold measurements. Manuals for everything are in the lab, and the TAs will do their best to help. However, they will be little better off than you as far as prior knowledge of these instruments is concerned because they probably do not have much experience.

**Measurements:** First, measure the propagation delay for both possible directions of input transition ($t_{LH}$ and $t_{HL}$) on three of your chips: 74LS14, 74LS240, and 74LVC04. Also measure the output rise and fall times for these gates. Use the 3.3 volt fixed power supply when doing these measurements on the 74LVC04. Delay times are measured between "gate threshold" voltage levels and you should take those values for the LS chips from your measurements for Lab 2. The reference level for the 74LVC04 is .5*VDD or 1.65 volts. There are several ways one might define the rise and fall times. As shown by $t_r$ and $t_f$ in Fig. 6-i (b), we will define them in terms of the time to go between the defined HIGH and LOW voltages. For the LS parts, these values are 2.4 and 0.8 volts. The 74LVC04 parts use .7*VDD and .3*VDD levels. With the 74LS240 also measure the delay

from the enable input to the gate output for both directions of the enable input transition. To make the change of state visible use a 1 K ohm pull-up resistor on the output and tie the gate input to ground so the output will be LOW when enabled. Such a circuit makes the relation between the enable input and the output similar to that between the input and output of an inverter.

Figure 6-i (a) shows the measurement arrangement. Adjust the oscilloscope so that the horizontal sweep is triggered **only by the input** waveform. Then superpose the input and output waveforms, being careful to match the zero levels exactly. Measure both $t_{LH}$ and $t_{HL}$ between the threshold crossing points of the input and output waveforms, using the threshold voltages determined from Lab 2 as the reference levels for the time measurement. In your report, please be sure to specify what level you used. See Fig. 6-i (b) for a sketch of what the scope trace should look like. These delay times will be quite short, typically a few nanoseconds. To measure them, you will need to use the fastest horizontal sampling rates possible and will find the measurement resolution is coarse. Nonetheless, these are not negligible times for any serious attempt at a fast system.

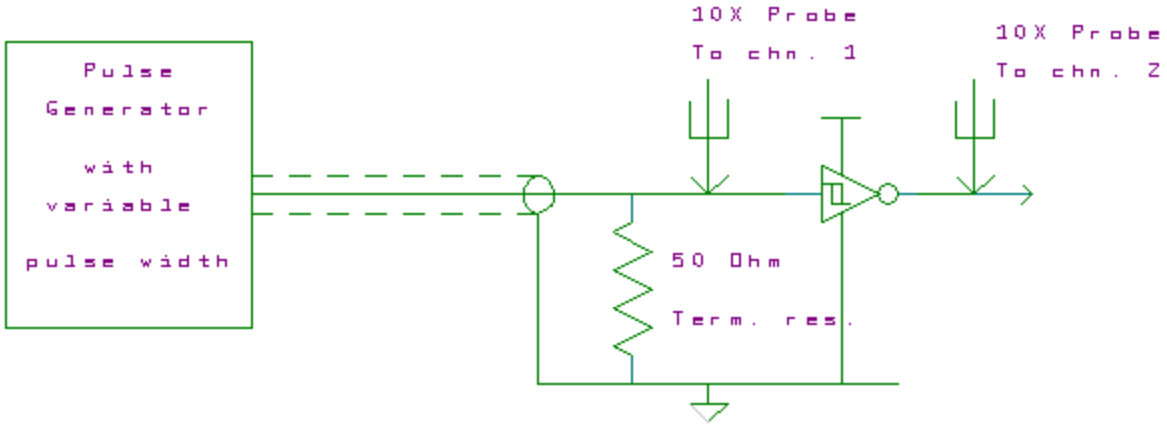



**Figure 6-i(a): Test setup for propagation delay**

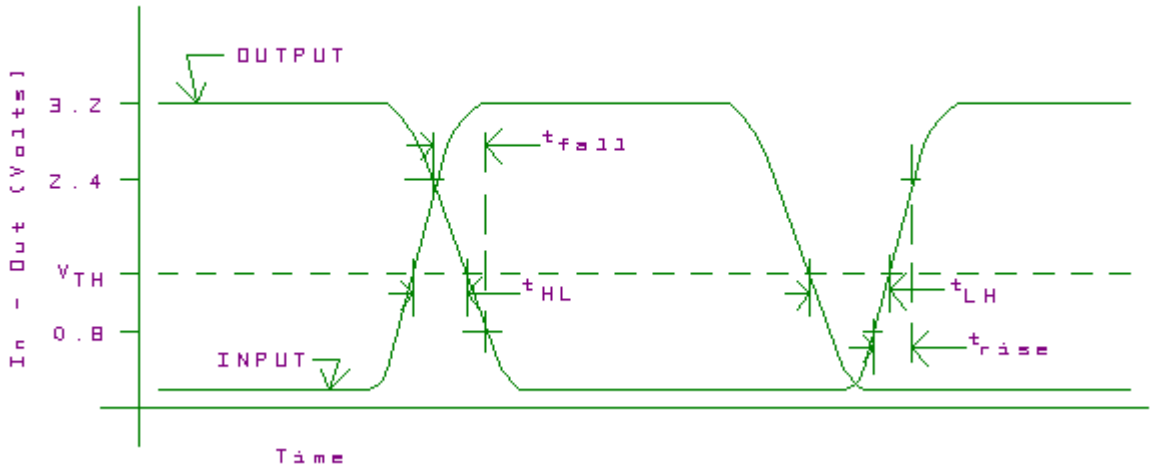



**Figure 6-i(b): Superposed, delayed waveforms**

Second, connect five of the six inverters in your 74LVC04 in tandem to form a ring oscillator, as shown in Figure 6-ii. (An oscillator produces a continuously changing voltage, in this case a rounded square wave. The circuit has not equilibrium voltage distribution so it swings back and forth between possible output states.) This time use a **variable** power supply and make separate measurements with 2.0, 2.5, 3.3 and 5.0 volts on VDD. For each supply voltage, determine the frequency of oscillation. Connect a 50 pf. capacitor from the output to ground. What are the new frequencies of oscillation?

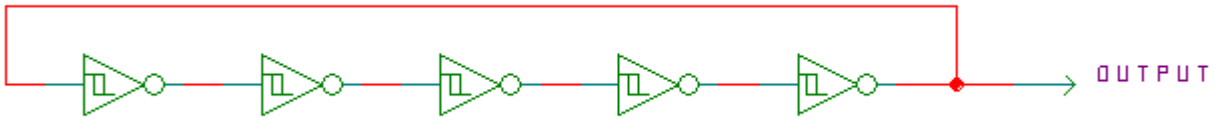



**Figure 6-ii: Ring oscillator**

Third, measure the power consumed by your ring oscillator at the same set of supply voltages. You should have a bypass capacitor between the VCC and GND pins on your chip. Use a sense resistor and measure the voltage across it as you did for current measurements in Lab 2. Probably need 10 to 20 ohms resistance. (Check that your resistor does not change the frequency and amplitude of the output appreciably.) Also measure the **change** in power dissipation and frequency of the ring oscillator if you load one gate with a 50 pf. capacitor from output to ground.

Fourth, build the circuit shown in Figure 6-vi using extra parts that the TAs have available. (We have put together kits with a resistor, two caps, and a crystal. Please put those components back in their envelopes when you are done with them.) These parts include a quartz crystal mounted on wires that form the equivalent of a resonant RLC circuit. (The equivalent circuit is shown on the right in Fig. 6-vi.) At slightly above the resonant frequency of the crystal, the crystal has the impedance of a coil or inductor. If the inductive reactance of the crystal matches (resonates with) the two 22 pf capacitors, the circuit will oscillate. Measure the current drawn by the inverter using a 100 ohm sense resistor. Also measure the input and output signals of the inverter, noting both their amplitudes and their shapes.

The reason the circuit oscillates is that the 330 K resistor forces the inverter to settle in the middle of the excluded middle with the input and output DC voltages equal. At that condition, as we discussed in class, the inverter has very high gain (~X50). The resonant circuit formed by the crystal and the two capacitors has the interesting property that at resonance a signal imposed at one side comes out the other side inverted. The inverter then amplifies and inverts that signal, reinforcing the output. Soon the output grows until it is limited to the power supply capability and begins to look like a digital square wave rather than a single frequency sinusoid. The effect is like the positive-feedback squeal you hear when a microphone can detect the output of a speaker that is driven

by it. This is how clock circuits are designed!

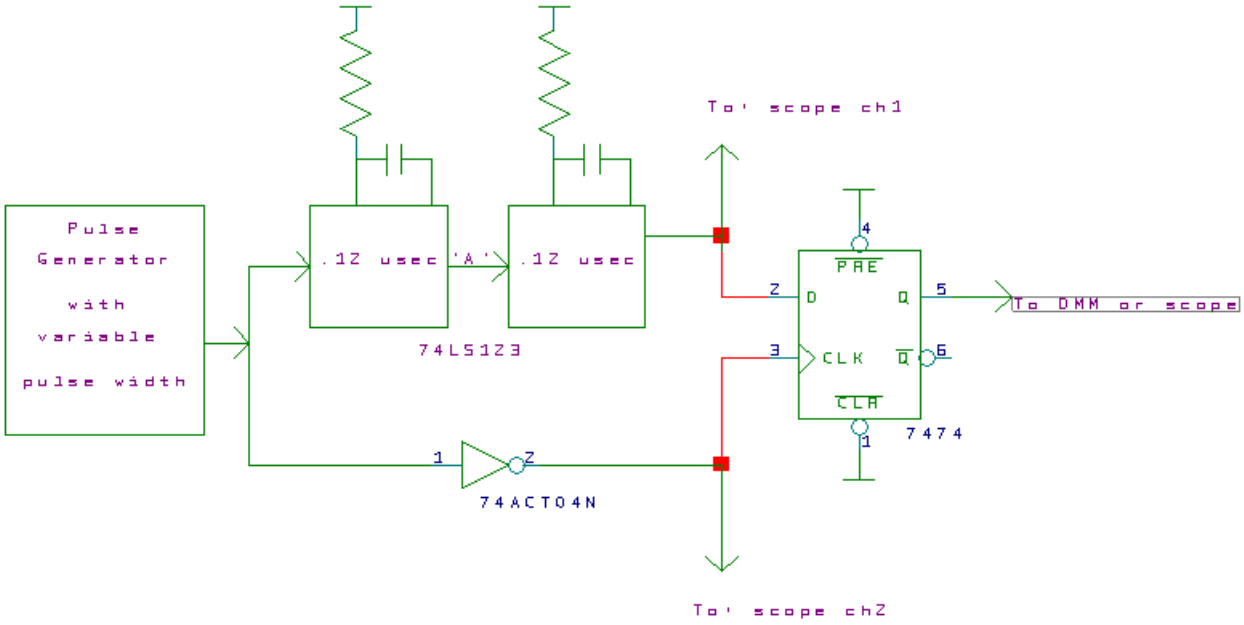



**Figure 6-iii(a): System for measuring setup and hold times**

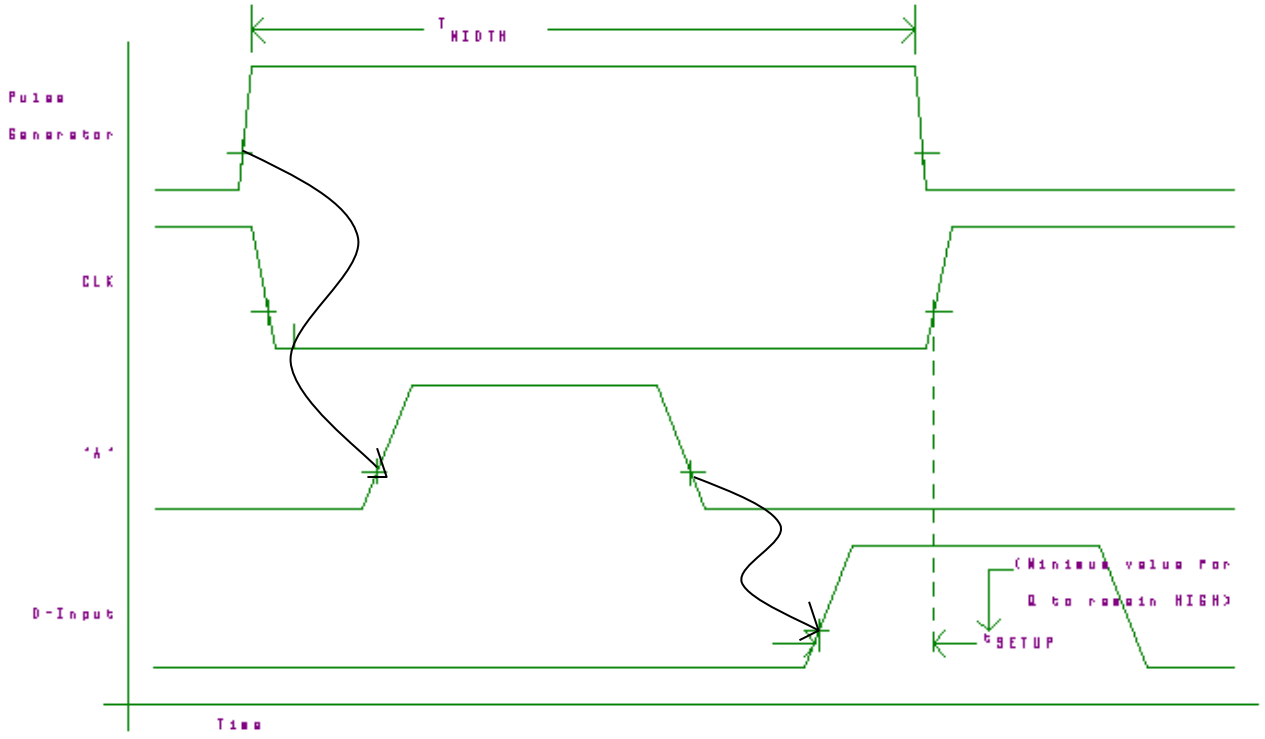



**Figure 6-iii(b): Timing diagram for measurement of setup time**

Finally, measure the setup and hold times for the D input of one half of one of your 74LS74 dual D-flip flop chips. Make measurements for two cases, first for the D input high before the rising clock edge and then for it low before the clock edge. (The first case corresponds to a low-to-high transition on the output; the second case to a high-to-low transition.) Figure 6-iii shows one way to do this using a variable width pulse generator as a signal source. (We have some home-made pulse generators that allow very fine adjustment of the width of a pulse with a very fast edge. There are also some new Agilent pulse generators. You may choose between them, but I believe the home-made ones may still be the best for this particular task.) The two halves of the 74LS123 should be set to give pulses at the D input of the 74LS74 which are about .15 microseconds wide starting about .25 microseconds after the leading edge of the input clock. (I call your attention to the data sheet limitation that the timing resistor used with the 74LS123 must not be smaller than 5 kilohms.) Please note that the **second section of the 74LS123 is triggered by the trailing edge** of the first section as shown by the lines of causality in the timing diagram. Be sure to connect the two sections together in such a way as to get that relationship. By varying the pulse width of the incoming pulse, you can move the rising edge of the 74LS74 clock relative to the pulse on the D input. The setup pictured in the timing diagram in Fig. 6-iii is for the D input driven by the Q output from the 74LS123 and so measures the setup time for a low-to-high transition of the flip-flop. In this figure, the setup time is simply the minimum time lag between the leading (rising) edge of the D pulse and the clock edge for which the 74LS74 output will remain HIGH, that is, will match the D pulse. Similarly the hold time is the minimum time lag between the falling edge of the D pulse and the clock edge for which the 74LS74 output will remain HIGH. The second case for a HIGH to LOW transitions on the output of the 74LS74 is measured by changing the polarity of the pulses to the D input. You can do the inversion by changing the connection from the 74LS123 from $Q$ to $\overline{Q}$. In this case the times are the minimums that keep the 74LS74 output LOW.

The setup and hold times are very fast. Setup time is likely to be in the 1 to 5 nanoseconds range and hold time may sometimes even be negative. (Negative hold time just means that the D signal does not have to persist all the way up to the clock transition to be recognized!) You will need to be very careful to keep the zero levels of the two channels of the scope the same and to use a consistent threshold level around 1.3 volts.

**Questions:**

1.) What delay would you measure if you put an even number of identical inverters in tandem and measured the total delay of the circuit? Suppose the number of inverters is $2 \cdot N$. Express your answer in terms of $N$, and the times $t_{LH}$ and $t_{HL}$.

2.) Why is the reference voltage level for the delay measurements chosen to be $V_{THG}$?

3.) Which of your rise, fall, or delay measurements in the first measurements of this lab are seriously limited by the speed of response of the oscilloscope? (The apparent rise time of a very fast pulse on the 100 MHz scopes in room 196 is about 3.9 ns. There is a very good match between the speeds of the two vertical channels.)

4.) In your ring oscillator measurements, what determines the period of oscillation? How does it relate to the measurements of $t_{LH}$ and $t_{HL}$?

5.) As power supply voltage changes, what happens to the speed and power dissipation of a gate (or system)? (Show a plot from your ring oscillator data.) Does the power dependence on supply voltage match predictions from class? (I expect a quantitative comparison to an actual formula.) The industry trend is toward lower supply voltages. Does this make sense for speed? Why? Why do people lower this voltage if they are not concerned about saving fossil fuel?

6.) A CMOS gate input appears to the output of the gate which is driving it as a capacitor to ground. Based on what happened with the 50 pf. capacitor and based on the typical data sheet value of the gate input capacitance, what should be the effect of connecting each output of the ring oscillator to two additional gates? (This is a total of 10 gates beyond the 5 in the oscillator itself. See the data sheet for the typical capacitance per input gate connection.) This is the effect of fanout in a system. Give a qualitative answer for speed and a quantitative one for power.

7.) How does the power dissipation of the operating ring oscillator compare to power measured for the same chip in Lab 2? Why?

8.) What is the *effective* capacitance charged and discharged by one gate on each transition of its output? Calculate the dissipation capacitance, $C_{DISP}$, of a single inverter gate of the 74LVC04A. In doing so, you may use the typical value of the input capacitance of the gate from the data sheet. Remember that if you are measuring the frequency of oscillation with a scope at the same time as the power, then you need to include the capacitance of the scope probe (14 pf) in your calculations.

9.) Is the *change* in power dissipation of the ring oscillator when you add the 50 pf. capacitor to the circuit consistent with the theoretical value? (Calculate a theoretical value for the change and compare it with the measured value.)

10.) Does the frequency of the crystal oscillator match the number marked on its case? Is the wave shape at the input of the inverter different from that at the the output?

**Discussion:** In all measurements it will probably prove necessary to **place a bypass capacitor** (0.1 μfd.) directly between the VCC/VDD and GND pins of the chip being measured. For best results, keep the capacitor leads short.

The pulse generators we suggest you use in this lab were home-built to give fast rise times and finely adjustable pulse widths. Both the generator and the coaxial cable used to connect it to your circuit require that you terminate the output with a resistor between 47 and 51 ohms mounted at the circuit end of the cable as near as possible to the gate being tested. (See Figure 6-i (a).) Our pulse generators usually have the terminating resistor plugged onto a tee at the circuit end of the cable, and no extra resistor is necessary on your board. However, be sure that the pulse generator you use does have a tee with both leads and a terminator on it. Also keep the leads from the tee as short as possible and keep them close together. These comments about terminating cables and wiring to your board are equally true of the Agilent pulse generators. If you choose to use them, you

will still have to be careful of the cabling. They are also not as easy to adjust as the home-built units and have slower rise and fall times.

Use two **identical** 10X probes to monitor the input and output waveforms. Signals travel down the oscilloscope cables at about eight inches per nanosecond. If the cables are not the same length, then the delays you measure will be off by the difference in the times it takes the two signals to go down their respective probe cables. There are a couple of different types of 10X probes in the lab because we have bought them at different times. Be sure that the pair you use are matched. Please be careful with the scope probe tips! (They break easily and are very expensive.)

Because the bandwidth of the oscilloscope is finite, there is a limit to how fast a signal the scope can display accurately. This limits the apparent fastest rise time to a little under $\frac{1}{\pi F_{BW}}$ where $F_{BW}$ is the bandwidth of the scope. For this reason the 100 MHz scopes are limited to about 3.9 ns apparent rise time, and the scope time resolution is limited around 1.0 ns. The 200 and 500 MHz scopes are proportionately faster.

We have six printed circuit jigs (see Figure 6-v) that simplify measurements on the SN74LVC04A inverters and Figure 6-iv shows their wiring for the inverters themselves. These jigs can plug into your protoboard as shown on the left in Fig. 6-v. Use the isolated gate (pins 12, 13 of the chip and 5, 6 of the edge connector) for the single-gate measurements and wire together the ends of the string to make the ring oscillator. Lines terminating in arrowheads are connections available for your protoboards. Please be careful working with these jigs as they are delicate.

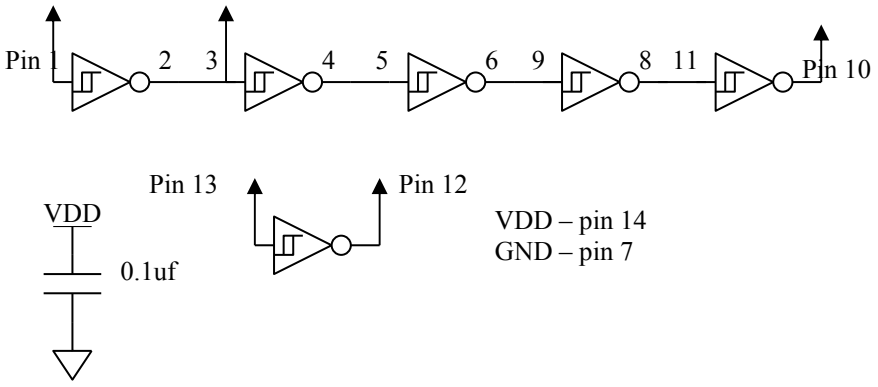



**Figure 6-iv: Prewired SN74LVC04A Connections: The bypass capacitor is on the socket and arrowheads mark the wires available for your connections. Pin numbers are the pins of the 74LVC04A. See Fig. 6-v for the way the pins connect to your breadboard.**

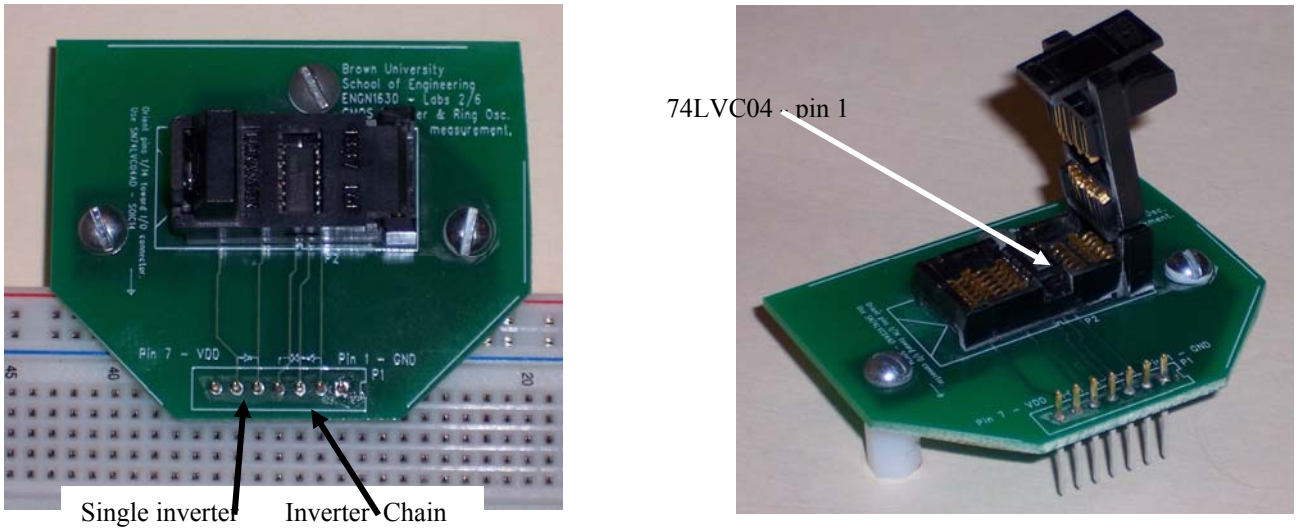



**Figure 6-v: Printed Circuit Jig for Measurements of a Socketed SN74LVC04A. Seven pins plug into your protoboard and you connect pin 7 of that connector to VDD and pin 1 to ground. Pin 6 is single inverter input and 5 the output of that inverter. The five inverter chain extends from pin 4 input to pin 2 final output**.

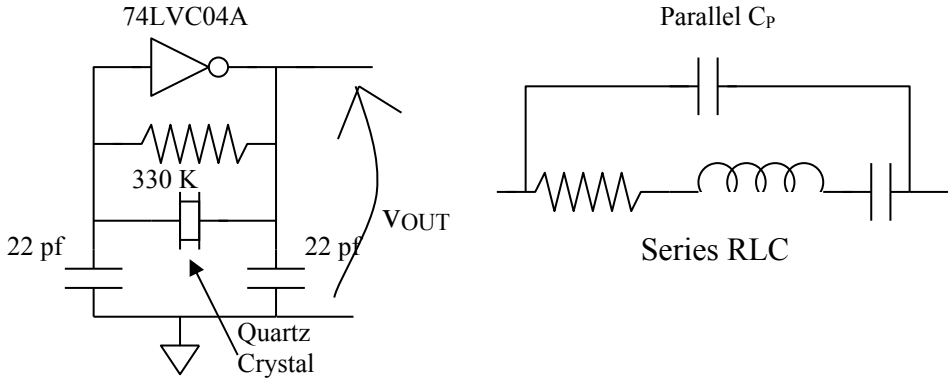



**Figure 6-vi: Quartz Crystal Oscillator. Left: the actual circuit with one inverter from an SN74LVC04A hex inverter. The resistor turns the inverter into an amplifier and the quartz crystal forms a resonant circuit with the 22 pf capacitors to make the circuit oscillate. Right: The equivalent circuit of the quartz crystal. The resonance is the result of electrically generated ultrasonic vibrations.**

### 9.2.8. Lab Seven

**Dual Slope A/D Converter**

**Requirements:** Design and build a 5-bit A/D converter based on the dual slope integrator technique. Generate a variable input signal in the range of zero to plus 5 volts with your potentiometer. Display the answer on one digit of your display plus an LED for your MSB. The circuit should update the display two to four times a second or so, so that it follows the manual rotation of the potentiometer. For Lab 7 to be signed off, we must see all the digits in proper sequence as the pot rotates. It is possible in testing the lab that either 0 or 31 will not appear because of mismatched resistors or because of offset in the comparator. This is an allowed deviation from the full sequence of digits.

Figure 7-i shows the recommended system in block form. The placement of the analog switches in the circuit is dictated by the requirement that the potential connected to any input of the particular switches you are using must be between +5V and ground. If you wish to try any variation on the placement of these switches, please keep this constraint in mind.

You will need at least a five bit binary counter. The four least significant bits may be realized with the 74LS169 counter or may be inside the CPLD. (I would like you to learn about counter chips.) If you put the counter inside the CPLD, you may not implement the counter using any of the Verilog language beyond the limitations of lab 5. (I want you to see the logic of binary counting if you use HDL for that.) Use the CPLD board for any control logic, for the display, and to extend the counter to five bits if you need to. The total number of cycles of the master clock in a conversion cycle may not exceed 66. The TA may check this by looking at the period of C1 with an oscilloscope.

**A**rrange your clock frequency to minimize the sensitivity of the circuit to 60Hz noise. This is done simply by making the signal integration period an integer multiple of the period of 60 Hz line noise, i.e. a multiple of 1/60th of a second, so that the integral of any noise injected from the AC power lines is zero.

**Discussion:** Dual slope analog to digital converters are widely used whenever relatively slow conversion rates are required. One application that you have seen already, probably without realizing it, is the digital voltmeters used in Lab 2. Other applications include thermocouple readouts, and appliance controls. The reason for this popularity is that the method allows very low cost, high precision conversion. Single-chip, decimal-coded, 14 bit converters are available for less than a dollar in quantity. If used with care, the technique is capable of realizing very high precision. It has been used, for example, to build high precision DC voltmeters with more than 20 bit resolution.

Labs 7 and 8 make use of operational amplifiers (op-amps to the initiated), which are discussed at length in the textbooks used for Engineering 520. Figure 7-iii shows a conceptual view of such a device. An amplifier is simply a circuit that has an output signal that is *proportional* to its input but larger in magnitude. It differs from a logic device in that the relation between input and output is linear (straight line) rather than deliberately distorted as is done in digital systems to

exploit the HIGH/LOW nature of digital signals. An opamp is a particular type of amplifier which has a large gain ($A > 10^5$) and a high input impedance (many megohms). It is a widely used component because it can be tailored to a wide range of uses with external resistors and capacitors. You have two op amps in your kit in the TL272IP dual opamp chip. In these labs they are used as *interfaces* between the digital and analog domains.

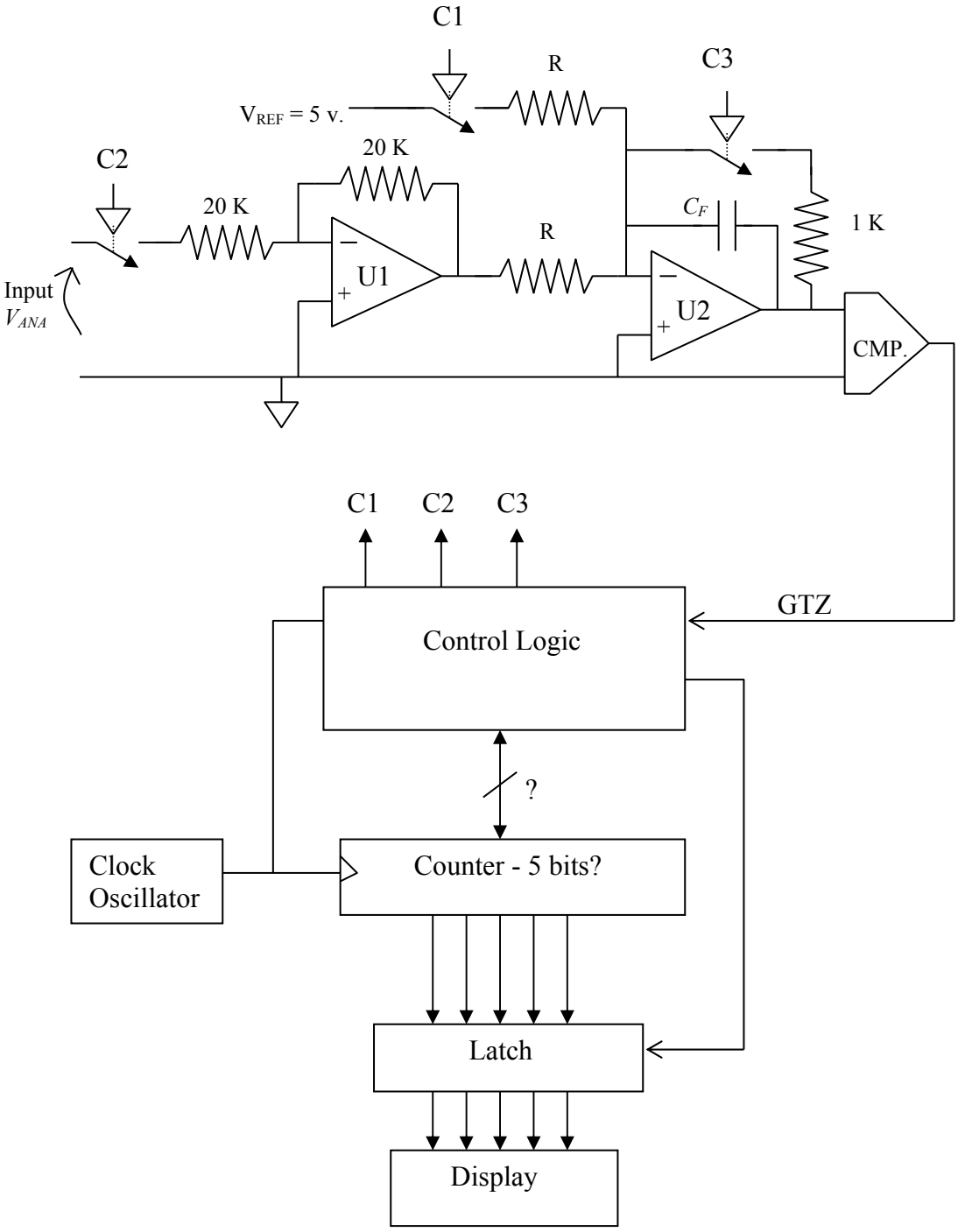



**Figure 7-i: Block diagram of a dual-slope A/D converter**

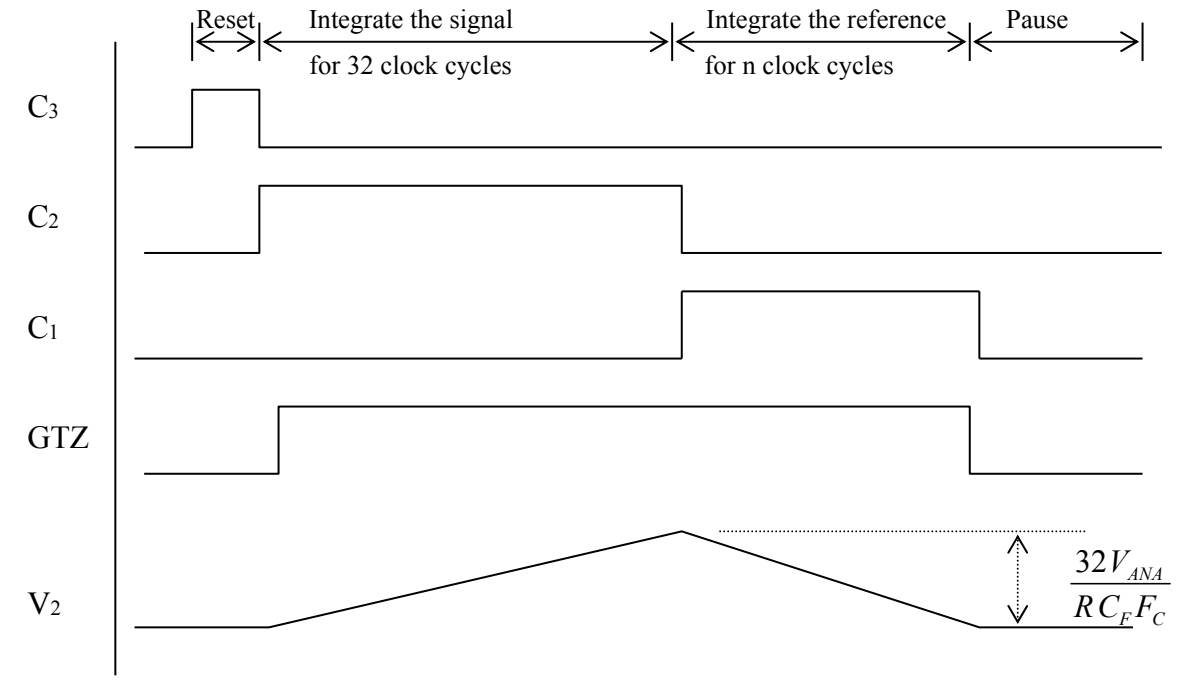



**Figure 7-ii: Timing diagram of the dual-slope A/D converter**

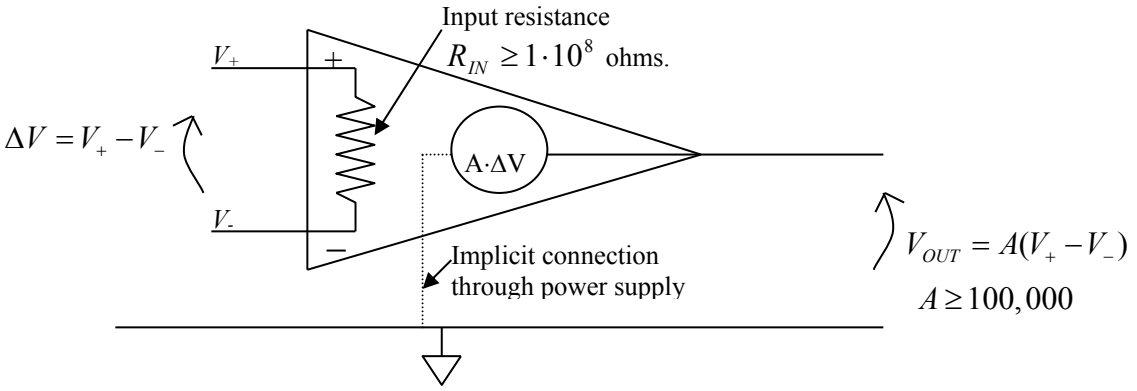



**Figure 7-iii: Operational amplifier conceptual view**

The TL272IP requires +5.0 V on pin 8 and -5.0 V on pin 4, but it does not have a separate ground pin. In the configuration of Figure 7-iv, an op amp provides a *negative linear* gain for output in the range $-5_V < V_{OUT} < +5_V$. The magnitude of this gain is $\frac{-Rf}{Rs}$, and depends only on the

resistor values. If $V_{IN}$ and/or the gain $\frac{-Rf}{Rs}$ is too large, the op amp output will "saturate" at nearly -5 or +5 volts. (Saturation of an amplifier is simply a condition in which the output is unresponsive to the input because it is already as high or low as the available power supply will let it get. Such a condition roughly corresponds to the HIGH or LOW state of a digital output.) Amplifier U1 in your ADC (Fig. 7-i) is connected in precisely this configuration with $Rf = Rs$. Thus the voltage at the output of U1 is equal to minus the analog input when SW2 is closed and zero when the switch is open.

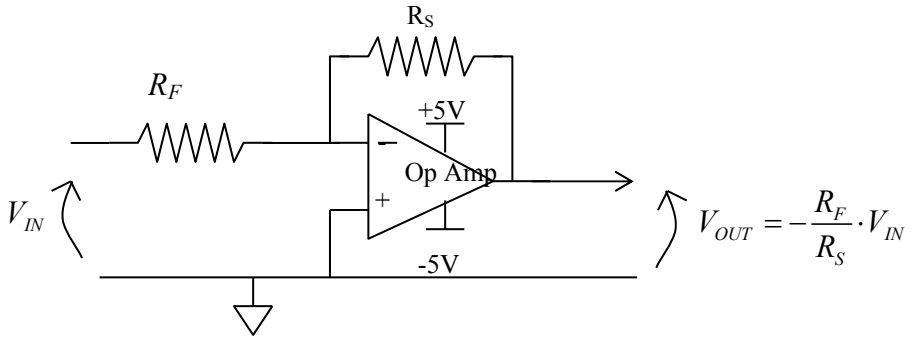



**Figure 7-iv: Opamp as a negative gain, wideband amplifier**.

The term "operational amplifier" comes from the fact that these devices can be made to do arithmetic operations on analog signals, including the operation of integration with respect to time. In your system, the second op amp, U2, is wired as a resettable integrator; figure 7-v shows that part of the circuit separately. When the switch SW3 is closed, the output of the amplifier is connected directly to its inverting input. The output potential $V_2$ is zero, and there is no charge stored in capacitor $C_F$. (Remember the relation between charge in a capacitor and the potential across it is: $Q = C_{\dot{F}}\text{V}$ .) Because the gain, A, of the amplifier is very large, the potential across the input terminals, which is $\frac{V_2}{A}$, is always nearly zero so long as the output is not in saturation. When SW3 opens, say at t = 0, the current flow through resistor R must go either into the amplifier input or into the capacitor $C_F$. Since the amplifier input resistance is very large and its input voltage is very small, most of the current will go into the capacitor. The current is given by Ohm's law as $\frac{V_{sig}}{R}$, and is the rate of flow of charge into the capacitor, i.e. $\frac{dQ}{dt}$. Thus

$$\frac{dQ}{dt} = \frac{V_{sig}}{R} = -C_F \cdot \frac{dV_2}{dt}$$

and

$$V_2 = \frac{-1}{(R \cdot C_F)} \int_0^t V_{sig}\, dt$$

To see how this might work, consider applying a constant voltage at the input *Vsig*. The current through R is simply a constant, $\frac{V_{sig}}{R}$. At t = 0 the output voltage will begin to go negative starting from zero; the rate of change will be $\frac{-V_{sig}}{(R \cdot C_F)}$ volts per second. The output will look like a linear ramp, which attains a value of $-V_{sig} \cdot \frac{T}{(R \cdot C_F)}$ at time T.

The dual slope A/D converter works by coupling the resettable integrator with switches that change the input to the integrator, with a comparator which determines whether the output of the integrator is positive or negative, with a counter that measures out time intervals and with some logic that executes a simple control algorithm. The process begins with a short period in which SW3 closes to reset the integrator. This time is shown on the timing diagram, Fig. 7-ii, where $C_3$ is "high." Such a **closure of SW3 must occur once each** conversion cycle regardless of what the comparator signal does. This period ends when the counter clocks into its zero state. Then the process proceeds the following way:

(1) SW3 opens and SW2 closes for N clock cycles; in this case N = 32. During this time, U2 integrates the input signal $V_{ANA}$ after it has been inverted by U1. By the argument given above, the output of the integrator at the end of this time will be $V_{ANA} \cdot \frac{N}{(R \cdot C_F \cdot F_C)}$ where $F_C$ is the clock frequency since $\frac{N}{F_C}$ is the integration time, i.e. the length of time of this part of the conversion. This situation is shown on the waveform of $V_2$ in the timing diagram.

(2) SW2 opens and SW1 closes as the counter counts over to zero again. The integrator then integrates the constant +5V reference voltage, and its output voltage decreases linearly with time. When the output reaches zero as determined by the comparator, the contents of the counter are latched into the output register as the digital output of the system, and SW1 opens to prevent the integrator output from going excessively negative and damaging SW3. The change in the integrator output during this part of the cycle is:

$$\Delta V_{INT} = 5 \cdot \frac{n}{(R \cdot C_F \cdot F_C)}$$

where n is the number of clock cycles in this period, i.e. the number latched into the output register. Equating this change to the voltage at the beginning of this part of the cycle, gives:

$$V_{ANA} = 5 \cdot \frac{n}{N}$$

volts. Thus n is a digital representation of the original analog voltage.

(3) The system then counts with switches SW1 and SW2 open until a convenient time to restart the cycle. (SW3 may be open or closed in this time at your option.) Since

there are often advantages to equally spaced samples, you might make the overall cycle length of your converter constant.

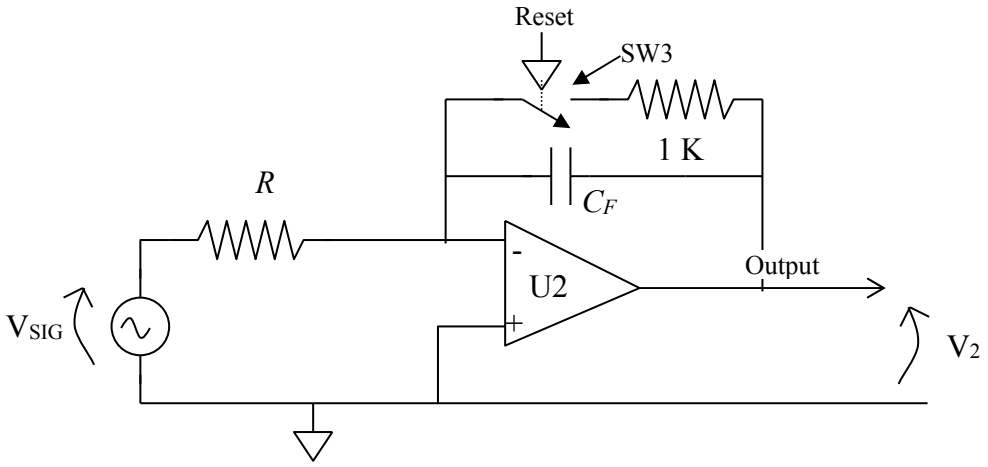



**Figure 7-v: Opamp as a resettable integrator**.

In addition to the op amps and comparator, the other new components being introduced in this lab are the bilateral analog switches. There are four of these CMOS switches in one DG202BDJ integrated circuit. Their operation is quite straightforward. Two pins of the package look like an electrical switch in series with about a 100 ohm resistor; the switch can be opened or closed in substantially less than a microsecond by a digital signal applied to a control pin. The switch is closed when this pin is "HIGH." The DG202BDJ has three power pins labeled GND, V+ and V-. Wiring the ground (GND) pin is obvious. It serves as the voltage reference for the digital input signals. The V+ power pin must have a higher voltage than any input pin and similarly the V- pin must have lower voltage than any input pin. Since the power connections to the opamps and comparator set the maximum voltages to the switches, connect V+ to the same +5 volts as the opamps and connect V- to the –5 volts that feeds the negative voltage terminals of the opamps and comparator. The 1K ohm resistor shown in series with SW3 is there to protect SW3 should the integrator output go above below a supply voltage due to miswiring or bad supply connections. The analog switches and the comparator are easily damaged by excessive current on an input pin, so please be careful and keep this in mind during debugging.

The use of the comparator in this lab poses some simple **problems**. The comparator in your kit, the LM311, can easily withstand input voltages up to +18 volts above ground regardless of the voltage on its VCC power line. However, in the negative direction the device can be destroyed instantly by voltages that make an input go negative with respect to its negative power line (pin 4). Since the op-amp requires a -5 volts on its VEE power line (pin 4), use the comparator with -5 volts on pin 4 too. This will reduce the chance of an accidental burnout. Both the +5 V and -5V power lines should have bypass capacitors (.1 μfd.) connected to ground. Those capacitors must be physically near the comparator. You may want to use additional bypass capacitors near the TL272IP chip. One other problem with comparators is that they tend to oscillate, that is their output goes up and down from logic '0' to logic '1' and back, rapidly and periodically when the inputs are nearly equal. This can be stopped but it is difficult to do, particularly with protoboard construction. It is better to recognize this possibility while designing the logic to control the counter and latch, so that

spurious transitions of the comparator output are ignored. **Do not use** the unconditioned output of the comparator to drive the display latch directly.

There are several design choices you must make in this lab. First you need to select a clock frequency, $F_c$ this is done on the basis of needing to update the output a few times a second. An optimum choice would also make the period of integration of the signal an integer multiple of $\frac{1}{60}$ second in order to maximize the rejection of 60Hz noise in the analog input signal. (60 Hz is the frequency of the power mains in this country and there is a tendency for noise at that frequency to appear in many systems.) Second, you need to choose values for R and $C_F$ such that with full scale, *i.e.*, +5V on the input, the output of the integrator will go up to more than 2 volts but less than 5 volts. This insures that the amplifier will not saturate but that the comparator will have a reasonable signal to measure. Finally you need to devise a logic circuit which will generate the necessary control signals for the conversion. Although this can be designed on an *ad hoc* basis, we believe you would be better off if you were systematic about applying standard finite state machine design techniques to this problem.

**Summary of Hints:**

(1) Be sure C3 will go high at least once each conversion cycle regardless of the comparator output (which may never go LOW!). Remember, if you use a CPLD, then the maximum clock cyles per conversion is 66.

(2) Be systematic in designing the logic; use finite state machine techniques.

(3) This is an excellent example of when a logic analyzer makes quick work of debugging. Display all three clock signals, the GTZ and the master clock on the analyzer and usually the source of your trouble will be apparent. Use a standard scope to see that your opamp circuits are working. Spend the time to learn the tools - it will save a lot more time spent in random poking around.

(3) Use bypass capacitors liberally.

(4) Use -5.0 volts from the power supply for the $V_{EE}$ pins of both the comparator and the op-amp.

(5) The comparator output may be either '0' or '1' at the instant upwards integration starts.

(6) Do not drive the output latch directly with the comparator output. Spurious transitions on the comparator signal will cause malfunctions.

NOTE: The opamp used in this lab, the TL272IP, is a change from earlier editions of the manual that used an LF353. This changes the power supply requirements from +/- 12 V to +/- 5 V. DO NOT USE THE 12 VOLT SUPPLY TERMINALS FOR ANYTHING IN ENGN1630 LABS.

## 9.2.9. Lab Eight

### Successive Approximation A/D Converter (ADC) System with Sample and Hold

**Requirements:** Design and build a 4-bit ADC which uses the successive approximation method to search for a digital representation of an analog signal. The system shall include a sample and hold circuit which can stabilize the analog signal, *i.e.*, "hold" the converter input constant, during a conversion. The system shall accept a "start of conversion" (SOC) signal from an external circuit. On the positive going edge of that signal, the analog signal will be held and the conversion started. After no more than 100 microseconds, the conversion will be complete and the result latched into a register. At that time, the sample and hold circuit can return to tracking the analog signal. The converter will be expected to respond properly to SOC pulses with repetition rates up to 9 KHz. The SOC pulse width will lie between 10 and 30 microseconds. Figure 8-i is a block diagram of the system. The circuit should be designed for a maximum analog input voltage range of between 2 and 3 volts peak to peak.

To do this, you need to build a D/A converter (DAC) as shown in Figure 8-ii. As noted on this figure, the operation of the ladder the ladder being terminated in resistances of 2R ohms. The practical implication of this is shown in Figure 8-iii, in which both an output through an op-amp and one directly into a comparator are shown. Your kit contains precision 10K and 20K resistors from which to build your network. With these, you should encounter no problems with your ladder being inaccurate or non-monotonic. I will discuss the R-2R ladder briefly in class.

To test the output of your D/A converter, you should measure the output voltage levels with a voltmeter to verify that the output shifts according to the weighted binary influence. When the DAC is driven by TTL chips, as it is in this lab, the output of the DAC ladder network will range between +.1 volts and +2.0 volts. The outputs of all our test fixtures and of most of the function generators in the lab are symmetric about zero volts, i.e. in your case varies between +1.0 and - 1.0 volts. Therefore, it is necessary to offset the signal from zero by about 1.0 volts, if the DAC is to be able to approximate it. A technique to do this is shown as part of the sample and hold circuit shown in Fig. 8-v; it consists of a capacitor and resistor network between the signal source and the sampling switch. This supplies a fixed offset of about 1.4 volts and eliminates any offset the signal might have.

The operation of counters and comparators can be adversely affected by signals on the power supply connections. Please reread the comments on bypassing in the introduction to this manual and remember that they may apply especially to this lab.

To test your circuit for the TA, you will connect it to one of the ADC test fixtures we will have in the lab. These fixtures consist of a triangle wave generator and a DAC which is connected to an oscilloscope. The triangle generator supplies SOC pulses synchronized to a 2 volt peak to peak triangle wave. If you connect the output of your latch to the DAC and connect the analog input to the ramp signal, the oscilloscope should show a "staircase" approximation to the ramp, with each "step" corresponding to a digital code from your converter. All sixteen codes (0 through F)

must be visible in the correct order for your lab to be checked off. Fig. 8-iv shows an actual picture of such a waveform. In the same test, the TA will also check the output of your sample and hold circuit.

We have only three or four test fixtures for this lab. This means that there will be times when the demand for the use of these fixtures exceeds the supply. Please **test your circuit without the fixture** before you ask for a final test run. In testing your circuit before trying it on one of the test fixtures, you might find it useful to derive SOC pulses by counting down the free running successive approximation register (SAR) clock with another counter chip. Then look to see that all your clock signals are what you designed for, that the analog switch is working, and that the SAR output is reasonable.

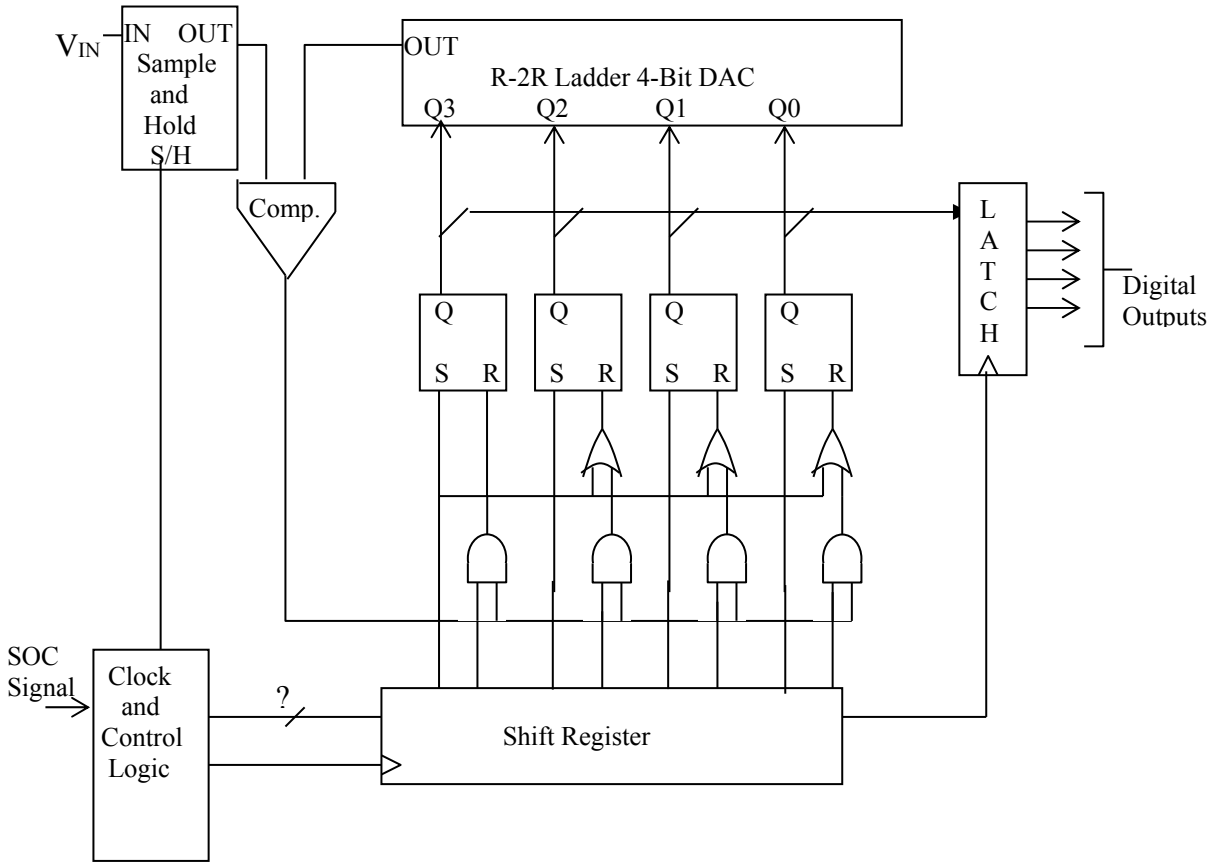



**Figure 8-i: Block diagram of a successive approximation ADC system**

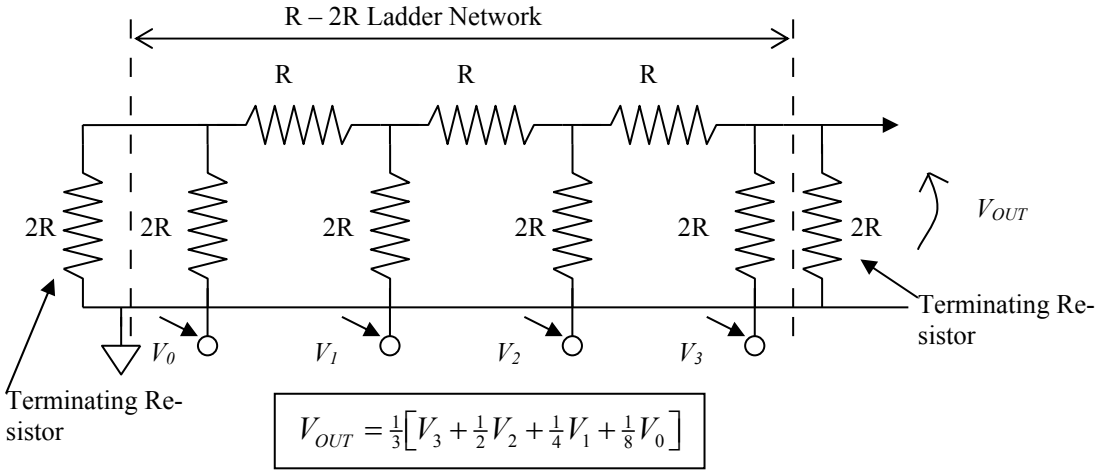
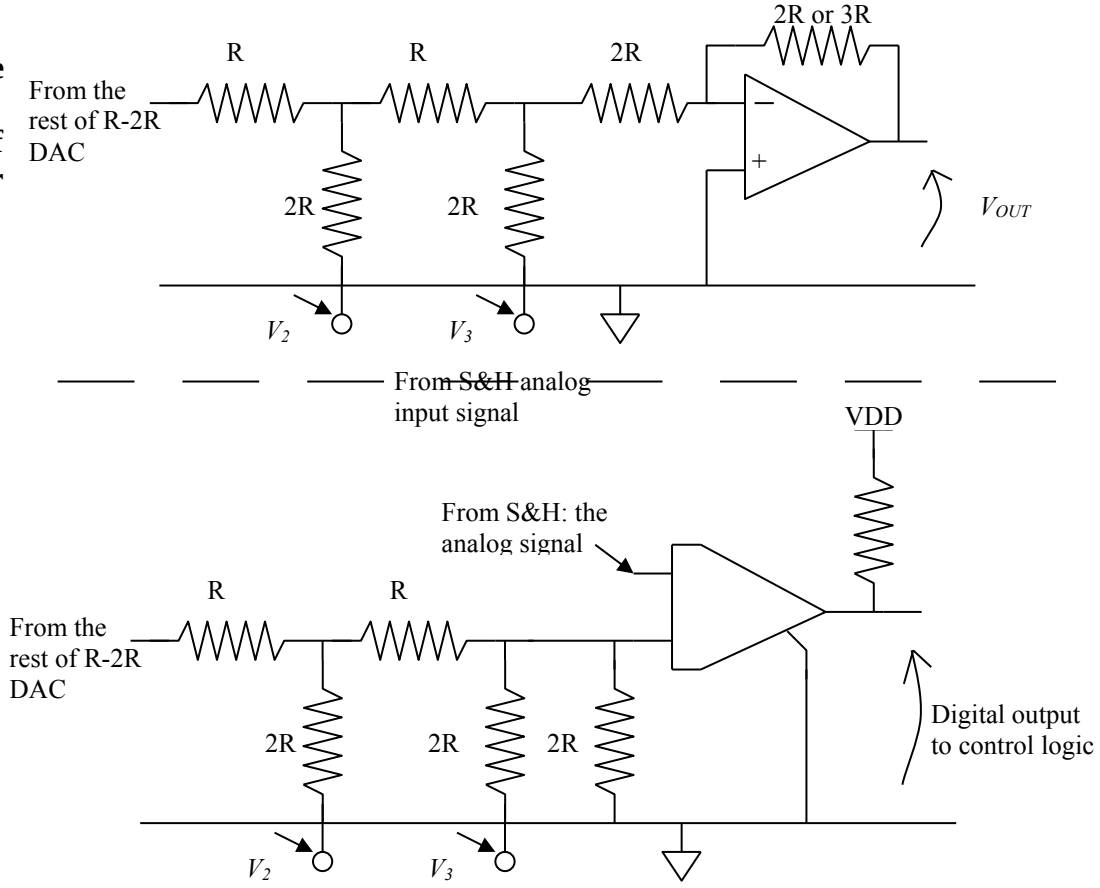



**Figure 8-ii: R-2R DAC.**

**Figure 8-iii: Connection of ladder to an op-amp comparator**




**WARNING:** the one even moderately challenging part of the lab is the "Clock and Control Logic" block. Do NOT use monostable multivibrators (one-shots) to build this. They can never work well in this application. Also you may NOT use the CPLD board for this lab. You have plenty of TTL parts to do the job, and the wiring is straightforward and not appreciably eased by using a CPLD.

**Discussion:** Successive approximation A/D converters are widely used for medium speed, moderate accuracy applications. They are generally somewhat more expensive and much faster than dual slope methods but slower than flash converters. With present design techniques, the conversion time per bit generally lies between .01 and .5 microseconds per bit, depending on the total number of bits. (High accuracy converters must generally allow longer times for their DACs to settle than do units with fewer bits.) Up to about 16 bit converters are readily available. Typically as of 2011 a 10 Msps 12-bit single-chip SAR converter costs about $ 6.00. (Even 16 bit converters have come down to modest prices recently – 1 input, 100 Ksps sampling for $ 7.00.)

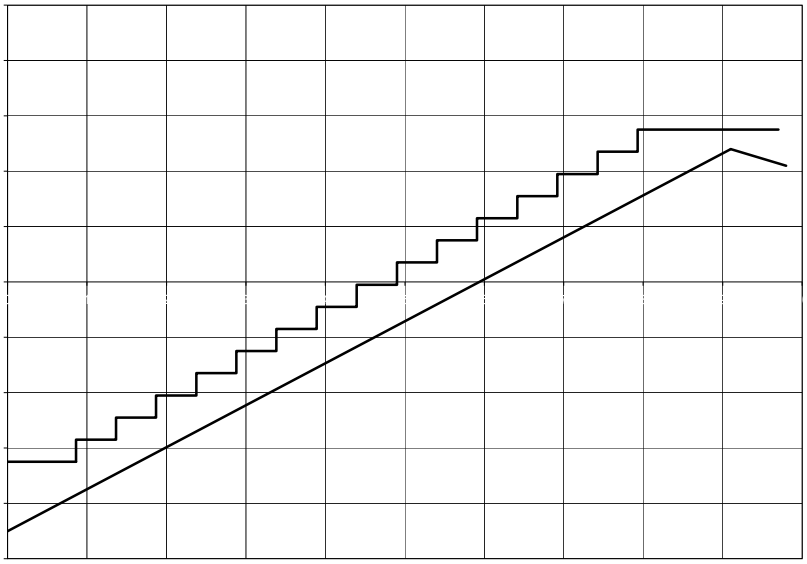


**Figure 8-iv: Oscilloscope waveform at the output of test fixture for a ramp input**

Figure 8-v shows the general form of a multi-purpose sample and hold circuit that can be built from your chip set using the DG202BDJ switch chip. (Please see lab 7 for information on using these switches.) The purpose of a sample and hold circuit is to prevent the signal which the converter is trying to digitize from changing during the time that the converter is operating on it. For the converter to digitize the signal accurately, the signal on which it operates must not change during the conversion by more than one half the amount corresponding to one least significant bit change in the digital output. However, in most situations, the incoming signal changes by a large fraction of the total scale between samples, and the sample separation is only slightly longer than the conversion time.

The sample and hold circuit works by charging a capacitor ($C_S$ in Fig. 8-v) through a switch from the input signal so that as long as the switch is closed, the capacitor voltage is the same as the signal. When the conversion is about to start, the switch opens; since there is now no mechanism by which the charge in the capacitor can change, the voltage across the capacitor remains constant until the end of conversion. The principle is the same as that used in the dynamic RAM storage cell. At the end of conversion the switch closes again so that the capacitor will "track" the incoming signal until the next conversion cycle begins.

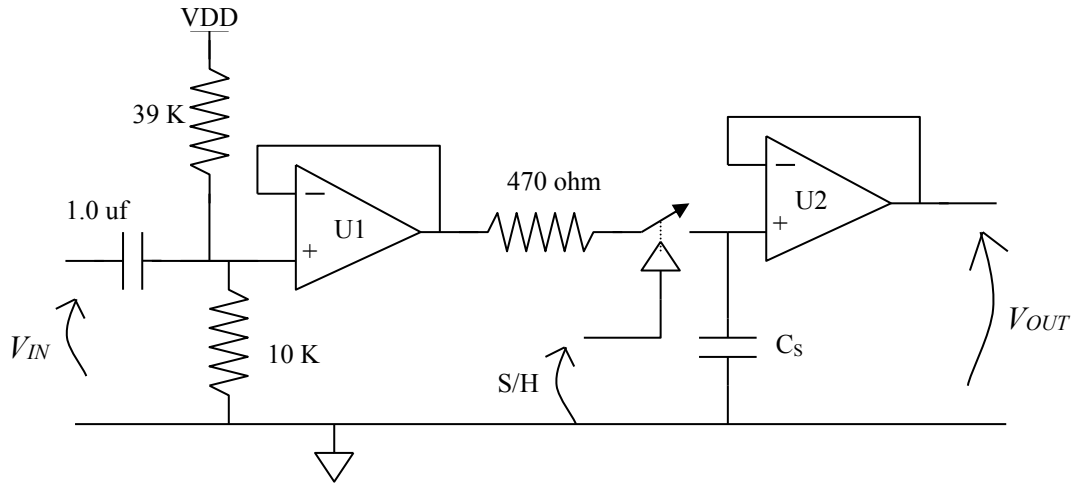



**Figure 8-v: Sample and Hold circuit with input offset.**

The circuit of Fig. 8-v serves a second purpose peculiar to the constraints of this lab. The capacitor (1 μfd.) at its input together with the two resistors and op amp U1 to which the capacitor attaches is used to introduce a DC offset into the incoming signal so that signal seen by the converter has the same range of voltage as the DAC output. The function of the resistor between U1 and the sampling switch is to protect the switch should the potential at the output of U1 exceed the allowed input range of the switch chip. The amplifier U2 is wired to have a gain of 1. Its function is to isolate the hold capacitor from any current drain and hence voltage drift during the hold time due to current drawn by the input of the comparator.

The choice of the sampling capacitor, $C_S$, is limited by the need for the signal on it to follow the incoming signal with sufficient accuracy when the sampling switch is closed. If the capacitor is too large, its voltage will tend to lag behind the input signal and will tend to vary with time less than the input signal does because not enough current can be passed by the switch to change the capacitor's charge rapidly enough. The switch current is limited by the resistance of the switch itself plus that of the protection resistor. The 10 KHz converter you are designing has worst-case only 10 microseconds for the sample and hold circuit to do its job. For the circuit to work properly for a four-bit ADC in this sample time, the product of $(R_{protect} + R_{SW}) \cdot C_S$ must be less than 2 microseconds. The value of $R_{SW}$ is about 50 ohm. One needs to select $C_S$ to be as large as possible consistent with this requirement.

The operation of the successive approximation converter itself requires a series of single pulses. The first of these sets the most significant flip-flop in the successive approximation register (SAR) high and all the others low. (The SAR is the set of flip flops connected to the DAC.) The second pulse resets this flip flop if the input signal is less than the DAC output. The third pulse sets the next most significant bit in the SAR, and the process continues through the rest of the bits until all have been set and conditionally reset. After the SAR has settled to its final value, it is necessary to latch that answer into an output register and to return the sample and hold circuit to the sample mode. Although there are several ways to generate the necessary sequence of pulses, we suggest you use a shift register as the basis for a simple scheme.

This system is an example of synchronizing a clocked system to an external asynchronous event, namely the SOC pulse. Any of the standard methods of doing this will work. Again this circuit is likely to be susceptible to power supply noise and will require bypass capacitors.

The practice of connecting the DAC inputs, *i.e.,* the 2R resistors in the DAC network, directly to the SAR flip-flop outputs is not very satisfactory. The output voltages of the gates are not well matched and are not independent of temperature. It is possible to do so in this lab because of the relatively small number of bits to be converted. A small but significant improvement in the circuit performance is possible within the constraints of your kit by using the fact that CMOS chips generally have output signal voltage levels which are much closer to the power supply and ground levels and which are more reproducible than are the output levels of TTL chips. For this reason, you could use your 74ACT04 inverter chip between the *Q* outputs of the SAR and your DAC network. This would result in more evenly spaced DAC levels and in a slightly wider range of output voltage for the DAC.

Finally, please reread the sections of lab 7 that deal with the connection of power to the comparator and analog switches. You will want to use -5 V on the V- line (pin 4) of both and +5V on their positive supply pins. You will need to bypass both power lines to ground.

### 9.2.10. Lab Nine

**Software Simulation of a Logic System:**

**WARNING:** We have installed new versions of all software this fall and they only became functional the first week of classes. While the Aldec Active-HDL software has been tested and explained in a separate handout, there may still be some operational problems. Do not hesitate to ask me or the TAs about anything that doesn't seem to run properly.

**Requirements:** Simulate a portion of either Lab 5, 7, A, B, or C. Labs 5, 7, and A use an XC9572XL CPLD for most of their logic and the simulation of these labs only studies the operation of the CPLD. That simulation uses the *Active-HDL* software (www.aldec.com) to turn a simulation file written in Verilog by the Xilinx ISE package into a full timing diagram. Labs B and C will use the Xilinx XC3S500E field programmable gate array and may also be simulated in the Aldec package. Instructions for using the Active-HDL tool for the CPLD are already posted on the class website. Instructions on simulating the XC3S500E FPGA parts will have to wait until later.

The exact requirements for what to simulate in each possible lab and for the data and time scale to display are given below. To receive credit for this lab, you hand in printed copies of the required timing diagrams and supporting documentation. Whatever method of file generation you use, the supporting documentation includes all original source files - .v, .ucf, etc. (You do not have to include the <>_timesim.v/sdf files – they are too prolix to be useful on paper.)

**WARNING:** We reserve the right to reject an unworkable design even if it is one you have gotten past a TA for lab credit. Something that works with things tweaked just right is not a properly robust design. Done right, a simulation can show how such a circuit fails.

**GENERAL REQUIREMENT:** any bus that is part of your simulation must be displayed on your timing diagram with hexadecimal radix. Use sufficient print space so that the hex notation is readable. You may also put the individual bit lines of the bus on the diagram, as you choose. Try to have the order of the signals on the diagram make some sort of sense, *e.g.*, put the clock at the top, then inputs and then outputs in some reasonable order going down the page.

**For Lab 5:** Treat the clock as a continuous, free-running, square-wave with period of 1.0 microseconds. Make the "Abbreviate" and "Halt" signals arbitrarily specified so that the system executes one complete cycle of the long count from 0 back to 0, then halts for two clock cycles, and finally executes the abbreviated sequence from 0 to 6 and on through 7, 6, and 7. Make the "Halt" signal bounce between clock edges to show it has no effect on the state of the counter. Do this once with the clock HIGH and again with the clock LOW.

As outputs, please show the state bits as a single bus and the segment and decimal point drive lines. Put the clock, Abbreviate, and Halt lines at the top. Prepare two hardcopies: the first should show the sequences matching the requirements of the lab. If necessary for clarity, you may split this printout onto two pages. The second printout is to be on an expanded scale that show the

propagation of the clock signal to the change in outputs out of the 0 state such that one can see the delay times.

**For Lab 7:** Treat the free running clock oscillator signal and the comparator output, **GTZ**, as inputs to your system. If you used the CPLD, simulate that with *Active-HDL*. If not, simulate the control logic, the five bit counter, and the latch with *ViewSim*. (See Figure 7-i.) While it is possible to simulate both the analog and digital sections of the system by coupling Active-HDL or ViewSim to a SPICE simulation, this is too complicated for the time available in this course. Therefore, you should simply treat **GTZ** as an externally specified signal and make it do what one would expect in a real conversion. Simulate two cycles of the A/D converter, one with an analog input corresponding to about one-third full scale and one with an input exceeding full scale. For the run at one-third full scale, make the GTZ signal changes to signal integration back to zero at a down count of 12. Then make the **GTZ** line oscillate or bounce back and forth from 0 to 1 and back a few times between when the reference integrates back to zero and when the conversion cycle begins again. Include at least one time with the **GTZ** line high at a clock edge well after the signal integrates back to zero. This must show that the latched data remains stable. The run simulating an over-range input will have **GTZ** high throughout the down integration and must show the generation of a reset signal at the appropriate time.

As outputs, please show the control signals C1, C2, and C3; the latch clock line; and the five data output lines. If your simulation is of a discrete version of the system, it is sufficient to show the binary counter outputs to the display logic rather than the display segment signals, that is, you don't have to show the display decoder logic. Do include segment signals for a CPLD simulation. Put the two input lines (clock and GTZ) at the top of your timing diagram. Use a free running clock period equal to your design value.

Prepare three hardcopy outputs. Two of these are to be on the most compressed time scale which will still reliably show the free running clock. Both should show the data being latched. (One is for the overrange condition, the other for one-third scale conversion.) The third should be done with a highly expanded time scale to show the time delays during the one-third full-scale conversion between the first edge of the comparator signal and the resultant change of state of the C2, C3, and data signals.

**For Lab A:** Simulate the operation of the CPLD alone. The simulation must show clearing internal registers, loading the multiplier, and completing the registered product. Multiply "1010" by "0110" and display the 8-bit product. Also display the 7-segment drive lines. Both the product lines and display segment lines should use hexadecimal notation on the timing diagram.

**For Labs B: t**he requirements are still TBD.

**For Lab C:** simulate the circuit in such a way that you reproduce the timing diagram of Fig. C-iii for writing data into the SRAM. Also provide a separate timing diagram for the output data for the digit select and segment drive signals. Make the output run long enough to see one advance of the character output address.

## 9.3. The Lettered Labs

### 9.3.1. Lab A:

### 4-Bit by 4-Bit Positive Hexadecimal Multiplier

**Requirements:** Design and build a circuit that multiplies two 4-bit numbers together and displays the eight-bit result. Consider the two inputs as positive hexadecimal numbers, *not 2's complement.* One of the input numbers must be entered via your keyboard, the other should be entered with one of your DIP switches. Use pull-up resistors and the switches to pull down. The keyboard may be used as the source of an undecoded, four bit number, i.e. four buttons can be pressed in binary combinations to make one input. However, as a challenge to the bored, you should know that it is possible to decode the keyboard fully as well as do the multiply within the constraints of your chip set. The computation (not including the loading of one of the numbers) must be done in less than ten clock pulses. **Your may not use a simple lookup table to do this lab.** You must choose an arithmetic algorithm and there are a couple of ways to do that including shift and add. There are even a couple of variations on the shift and add method.

| $P_{3:0} \bullet Q_{3:0}$ | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | | | | | | | | | | | | | | |
| 2 | 0 | 2 | 4 | | | | | | | | | | | | | |
| 3 | 0 | 3 | 6 | 9 | | | | | | | | | | | | |
| 4 | 0 | 4 | 8 | C | 10 | | | | | | | | | | | |
| 5 | 0 | 5 | A | F | 14 | 19 | | | | | | | | | | |
| 6 | 0 | 6 | C | 12 | 18 | 1E | 24 | | | | | | | | | |
| 7 | 0 | 7 | E | 15 | 1C | 23 | 2A | 31 | | | | | | | | |
| 8 | 0 | 8 | 10 | 18 | 20 | 28 | 30 | 38 | 40 | | | | | | | |
| 9 | 0 | 9 | 12 | 1B | 24 | 2D | 36 | 3F | 48 | 51 | | | | | | |
| A | 0 | A | 14 | 1E | 28 | 32 | 3C | 46 | 50 | 5A | 64 | | | | | |
| B | 0 | B | 16 | 21 | 2C | 37 | 42 | 4D | 58 | 63 | 6E | 79 | | | | |
| C | 0 | C | 18 | 24 | 30 | 3C | 48 | 54 | 60 | 6C | 78 | 84 | 90 | | | |
| D | 0 | D | 1A | 27 | 34 | 41 | 4E | 5B | 68 | 75 | 82 | 8F | 9C | A9 | | |
| E | 0 | E | 1C | 2A | 38 | 46 | 54 | 62 | 70 | 7E | 8C | 9A | A8 | B6 | C4 | |
| F | 0 | F | 1E | 2D | 3C | 4B | 5A | 69 | 78 | 87 | 96 | A5 | B4 | C3 | D2 | E1 |

**Figure A-i: 4-Bit Hex Multiplication Table**

In preparing the jed file for your circuit, you **may not use any of the arithmetic constructs in Verilog**. The most obvious example is you may not use the "+" operator as addition. (If you

use the default set of operators, then "+" will do arithmetic addition on two multi-bit operands as if they were unsigned binary numbers, *e.g.*, OUTPUT = INPUT + [0,0,1,0]. The point of this lab is to get you to think about the structure of arithmetic hardware, not to learn the syntax of comprehensive languages.)

If you use an "add-and-shift" scheme, then the multiplier has to be converted to a serial bit stream. While you may do this within the XC9572, you must show the conversion to a serial stream by routing the output of the serializer to a pin and back in on another. The TA may require you to pull that connection and use a HIGH/LOW on the input pin to show the serialization. You may use another pin to load that chip. Whether the keypad or dip switch operand is the multiplier is immaterial.

Your circuit must include a continuous clock so that the answer appears to be displayed continuously as the dip switch or push buttons are changed. The display should not flicker. You should also build an external single-step clock into your circuit so you can troubleshoot by observing intermediate "add-and-shift" results and so you can demonstrate the number of pulses to do the multiply. Such a feature may make the Fault Tolerance Question easier to answer, since you will not have to compute by hand *all* intermediate results for your altered circuit. See Figure B-i for the hex multiplication table that your circuit must realize.

**Discussion:** Consider an add-and-shift scheme for your design, perhaps an extension of the serial adders discussed in your text. (Wakerly also discusses the add-and-shift multiplier as an example of a synchronous system design. His example has more detail but may be less immediately applicable.)

### 9.3.2. Lab B:

### Music Box/Greeting Card Sound from a Xilinx FPGA

**Requirements:** We will have three or four evaluation boards for Xilinx XC3S500E field programmable gate arrays in the lab. I have posted a choice of two project files, each having much of the wiring to make the FPGA into an electronic music box, on the class web site. (The difference between the two projects is the method of frequency synthesis and you may choose either.) We will also supply you with a small loudspeaker complete with amplifier already wired to the evaluation board. Figure B-i shows how the system is supposed to work in block diagram form. As discussed below, each project is missing parts of its code. Your task is to reconstruct, enter, debug, and download that code so the system plays a song. The test of whether the circuit works is whether the tune, which is in four-part harmony, is recognizable!

There is a master clock at 50 MHz on the board and that is counted down to generate clocks for the various components of the system. The prescaler block counts the 50 MHz down 16:1 to 3.125 MHz. There are two reasons for this. First, it lowers the number of bits needed in subsequent counters. Second, it lowers the clock rate of subsequent counters to reduce the power required. In a real product one would try to design with as low as practical a clocking rate to extend battery life.

The "beat_clock", which sets the rate at which notes are played, is one output of a second counter and is determined so the song completes in roughly 70 seconds. There are 1692 entries in the song PROMs. Design the code for the beat counter such that a quick change of a parameter in the Verilog code can speed up or slow down the beat. (One of the FTQs may be to show how you expressed that parameter and to make a change to it and estimate the resulting duration of the song.) The second function of the "beat clock generator" is to provide the clock for the frequency synthesis blocks. The project files make the top-level interconnections using a schematic instead of a Verilog file. (The Xilinx tool turns the schematic into a structural Verilog file with the extension ".vf.") Look at that schematic drawing to see how the clocks are set up for your choice of frequency synthesis.

The note frequency synthesis may be done either by preloading a counter with the contents of a PROM and counting down to zero or by an adder/accumulator method. The two methods are shown in block form in Fig. B-ii and are discussed at greater length below. You are free to choose the method but be sure to download the appropriate project file set. It is primarily the contents of the PROM memory blocks in each project that change with the frequency synthesis method.

To provide for rests in the music and a distinct attack on the notes, the msb of the PROM outputs is used to gate the output. When the bit is HIGH, the output of the frequency synthesizer block will stay constant. Because your boss once had trouble gating a free-running oscillator, she insists that in your design the gating be done by freezing the count so any pause in the counting will be an integer number of the underlying clock cycles. The TAs may check this by examining your Verilog code, so have that ready when you ask for a check-off. (Eventually I hope to have logic analyzer connections that could check this more thoroughly.)

There are four switches on the lower right corner of the FPGA board and they are wired so they can provide simple HIGH/LOW signals to the FPGA. Your program must use two of these switches for special functions. Switch SW0 is the restart switch. When HIGH, the box stops completely and when set LOW again, the song starts from the beginning. Switch SW1 is a counter reset switch. Partly it is included to make simulation easier. However, its actual function is to pause the music, that is, the song stops when the switch is LOW and takes up again where it left off when the switch goes HIGH.

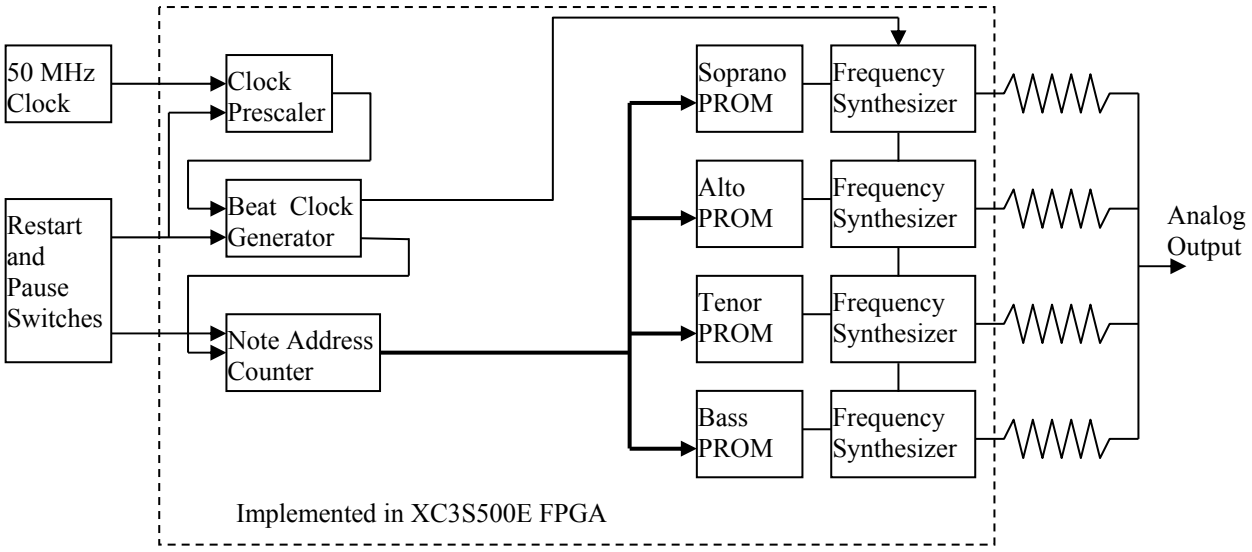



**Figure B-i: Block Diagram of Music Box System.**

**Discussion:** Most of you have probably seen and heard one of the musical greeting cards that are especially popular at Christmas. You open one, and it starts to play a tinny version of *Jingle Bells* or *O Tannenbaum*. These are really a remarkable *tour de force* of manufacturing. For a manufacturing cost that has to be well under one dollar, each card includes an on/off switch, a battery, a programmable oscillator with enough ROM for the tune, and a speaker (actually a transducer). The whole system has to be wired and shrunk to fit inside the card. Despite its cost, the battery seems to last for an hour or so of playing. In this lab, we ask you to imagine being in the early stages of designing a new product of this kind, say a music box that is to have slightly better sound quality than a greeting card. Your management thinks that an essential feature will be four-part harmony. One of the first questions you want to ask is what minimum algorithm will get adequate sound quality. At this stage, the cost minimization is not critical (that comes later). As is usual, development time is important, so the use of an FPGA with some block memory seems justified for exploration. To get the hang of things, you will start off with a primitive scheme that produces gated square waves and does a very primitive digital to analog conversion. (The DAC operation is the purpose of the four resistors on the right in Fig. B-i.)

The music produced by your system is very simple. It has four lines of notes without much subtlety of intonation or timbre. All one has to do is generate a signal with the appropriate variation of

frequency and with breaks for rests and for the start of each note. The result sounds MIDI-like. The function of the PROMs is to contain the sequence and duration of the notes. The thirteen or fifteen least significant bits of the data specify the frequency directly as the bit pattern needed to run a frequency synthesizer. There is no decoding necessary such as one might use to reduce the required amount of memory. The most significant PROM bit is used to gate the output clocking and so specifies rests and breaks. The duration of notes is determined by how many times the same frequency data is repeated in the PROM. Thus the PROM address is simply incremented at a constant rate proportional to the beat of the music. Sixteenth notes in the music are repeated four times in the PROM, so a quarter note (4/16 th's) occupies 16 PROM locations. Music is normally played at between 50 and 150 quarter notes per minute, which sets the required modulus of the counter in the beat generator of Fig. B-i. The total capacity of the PROMs is more data than I had the patience to enter. Therefore, the soprano PROM's most significant nibble has an 0xF entry at the end of the song. The note address counter is to recognize this nibble as a cue to restart the tune.

FPGAs have both configurable logic blocks and dedicated blocks for such things as memory and multipliers. The dedicated block are wired into a design by the same methods as the logic but their functions are preprogrammed and are not synthesized the way logic is. To use these resources requires ad hoc, vendor dependent procedures. To hid that from the user, Xilinx provides a tool that makes symbols for memory blocks that you can wire into a "schematic". The logic synthesis tool looks at these blocks as "black boxes" and simply passes their properties and connections on to the layout tool. When one of these memory blocks is loaded with data as part of the device configuration and the mechanism to write the memory is locked, you have effectively converted the block to a PROM. We have provided the symbols and PROM contents appropriate to the frequency synthesis of each project.

There are several ways a digital system can convert a binary number into the frequency of a chain of pulses. Two such ways are depicted in Fig. B-ii. The simplest method is the reloadable counter shown on the left in the figure. A fixed frequency oscillator drives the counter. Every time it counts down to zero, it is reloaded on the next clock cycle with the number specified by the PROM bits. The zero condition generates a carry-out pulse and these pulses occur with constant frequency. If *N* is the number to be loaded, then the output pulse rate is simply

$$f = \frac{f_c}{N+1}.$$

The frequency range is from $f_c / M$ to $f_c / 2$ where M is the modulus of the counter. However, the spacing between possible frequencies is not uniform. (It is obviously also possible to use an up-counter and to reload on detecting all ones. The result is basically the same.)

The second method for frequency synthesis, shown on the right in Figure B-ii, uses a combination of an adder and an accumulator. At each cycle of the clock oscillator, the value of the PROM output (considered as an unsigned binary number) is added to the contents of the register and the new sum is stored in the register. Carry-outs from the adder are the output pulses of the synthesizer. While the time between such pulses is not constant, the *average* rate of the pulses is

$$f = \frac{f_c \cdot N}{M}$$

where $M$ is now the modulus of the adder and $N$ must be in the range of $1 \le N \le M/2$. Again the frequency range is from $f_c / M$ to $f_c / 2$, but the spacing in this case is uniform. The uniformity of frequency spacing makes this method an attractive one for commercial frequency synthesizers, which use a variety of tricks to even out the spacing between pulses. (It would be well worth your while to be sure you understand how the adder system works. Try seeing what will happen with an adder of modulus eight, i.e. 3 bits. Considering the $N$ = 1,2,…5 cases should prove the formula as well as show the reason for the constraint for $N \le M/2$. The *M/2* comes from the fact that if there is overflow on two successive clock pulses, there is only one output pulse, not two.)

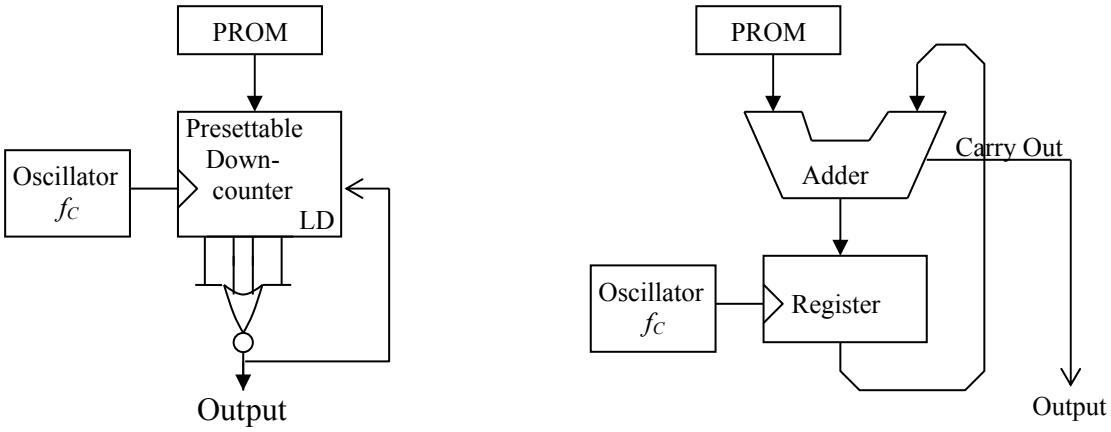



**Figure B-ii: Two Methods of Frequency Synthesis**

If you write this frequency synthesizer in behavioral Verilog, there is a small problem in getting the carry-out of the adder. The simple way to do this is to make the adder one-bit wider than the addition requires. Concatenate a zero as the most significant bit of each of the two adder inputs. Have the accumulator register save only the required number of bits and then the most significant bit of the augmented adder will effectively be the carry-out. The PROM for this method has been written assuming that it holds 13 bits of frequency data and the accumulator register is also 13 bits wide. For example, if PROMOUT[12:0] is one adder input, then make the adder 14 bits wide and that input be {1'b0, PROMOUT}. Form the input to the other side of the adder similarly. Adder bit 13 (the 14$^{th}$ bit) is the carry-out.

You are free to choose whichever method, counter or accumulator, appeals to you. To make a rational choice, consider the advantages of each: the counter method uses the minimal number of components and gives a pulse train with absolutely equal spacing. This means a pure tone. The adder method operates with a much lower clock rate for a given resolution at the upper end of its range. (You might consider the implications of the clock rate for the power used by your music box.) Moreover, the variation of time from pulse to pulse acts as a kind of frequency modulation, which will give some timbre to the notes. Frequency modulation as a way of generating

timbre has been used in a number of music synthesizer systems. Whether the effect in this system is either noticeable or attractive or not is an empirical matter.

The PROM contents have been loaded on the assumption that the range of output frequency will be approximately from the second A below middle C (A2 = 110 Hz. on the American Standard pitch scale) to the second A above middle C (A5 = 880 Hz.). Neither frequency will be generated exactly, but both will be fairly close. The PROM data value, *N*, for the 880 Hz. note will be decimal 1774 for the counter method and 550 for the adder method.

In addition to the synthesizing circuit, you will need counters before and after the synthesizer. You will be starting from the crystal-controlled 50 MHz clock on the Spartan 3E development board. The prescaler brings this down to 3.125 MHz. You will need to scale that frequency down further with a counter in the "beat generator" block to derive the clock for the synthesizer. Since the output pulses from both synthesis methods have a short duty cycle and the signal to the speaker is to be a square wave, you will also need to divide the output by two. In the case of the adder circuit, you make the second counter a divide-by-four counter so that the short- term variations in the output frequency will be smaller.

Begin by downloading a project from the class website and unzip them it into its own directory on your U:\ drive. These began as fully functional music box designs with which I tested the hardware. I then stripped certain parts of the design out. These include:

- The symbol for the soprano part PROM from the schematic. (The symbol itself is still in the file.)
- The Verilog code of the frequency synthesizer block, the note address generator, and the beat clock generator
- Some pin references for the switches on the FPGA board in the user constraint file. (I supply the .ucf file for the rest of the connections. The manual for the FPGA board is also posted on the class web site and has the pin assignment information for the switches.)

You need to restore the missing information and demo the music box to a TA.

### 9.3.3. Lab C:

**Build the *Tweet*++ Electric Sign Using Verilog Code to Program a Xilinx FPGA**

**NOTE: This revised writeup is current as of 12/2/2011.**

**Requirements:** Program the Xilinx evaluation board FPGA to make an 8-character LED sign that displays a message by scrolling it across an 8-character display character-by-character sufficiently slowly that you can read it. Tweets are limited to 140 characters, but you will make your *Tweet*++ display to accommodate 141 characters! All eight characters display at once and the whole message streams across, right to left, at about one to two characters per second. There are multiple examples of this style of sign in Times Square.

This lab was newly revised for AY 2012/13 and there are only three or four setups for it, so please be considerate about how long you occupy the system. (There are three currently functional setups and one more that I need to repair.) The evaluation board for the Xilinx Spartan 3E500 FPGA has a second board plugged into it that I designed and built. Among other things, it has a set of four 2-digit, 14-segment displays that can be driven from the FPGA directly. The display has a common-anode structure but its operation is similar to the two-digit, common-cathode, 7-segment display of lab 3.

Figure C-i shows a block diagram of the system. Both the display and data bus circuitry are built onto my custom board, the schematic diagram of which is posted on the class web site along with the user manual for the FPGA board. The board includes MOSFET drivers for the display anodes and a demultiplexer for their gates. The board plugs into the expansion connector of the FPGA board. The pin connections to the FPGA for the display and data bus lines are tabulated in a ".ucf" file for this lab posted on the class web site.

The implementation of the sign controller is shown in Fig. C-ii. A PROM inside the FPGA that is loaded with a look-up-table (LUT) when the FPGA is programmed generates the segment drive signals. The input to the table comes from a dual-port SRAM that is filled with text data represented by ASCII-coded bytes from a host program on a PC over an 8-bit parallel data bus. The PROM only translates the keyboard characters stored in the SRAM into segment signals.

The SRAM has two ports that can read and write data independently from each other. Thus, there is an output address controller that continuously reads from one port of the SRAM and displays the text. A second controller transfers input data from the data bus into the second port of the SRAM. In Figure C-ii, the output bus, clock signal and address bus on the right side of the SRAM block are the output port and the similar set of signals on the left represent the input port. Except for your needing to reset the output address right after new data is loaded into the SRAM, the dual-port arrangement makes display and downloads independent of each other. (The connection for the reset is not shown in Fig. C-ii.)

Figure C-iii shows the timing diagram for transferring data into the character SRAM. There are four control signals and eight data lines. Seven data lines encode the text characters in the ASCII system. The eighth line is a flag line to reset the output address to zero at the end of the

clock cycle during which it is high for the current character. The host program on the PC sets the eighth bit of the last character of the message. If the output counter is also reset when new data is entered, this assures that the full message will recycle with no extra characters from prior messages. Writing data into the SRAM is done to the input side of the memory and has no special requirements on timing relative to the output side. However, the SRAM is synchronous and so each port requires a clock signal that registers the data of its respective input or output data bus. Very likely you will use a clock derived from the free-running "mclk" signal to clock both ports of the SRAM and the finite state machine that handshakes with the data source and host PC.

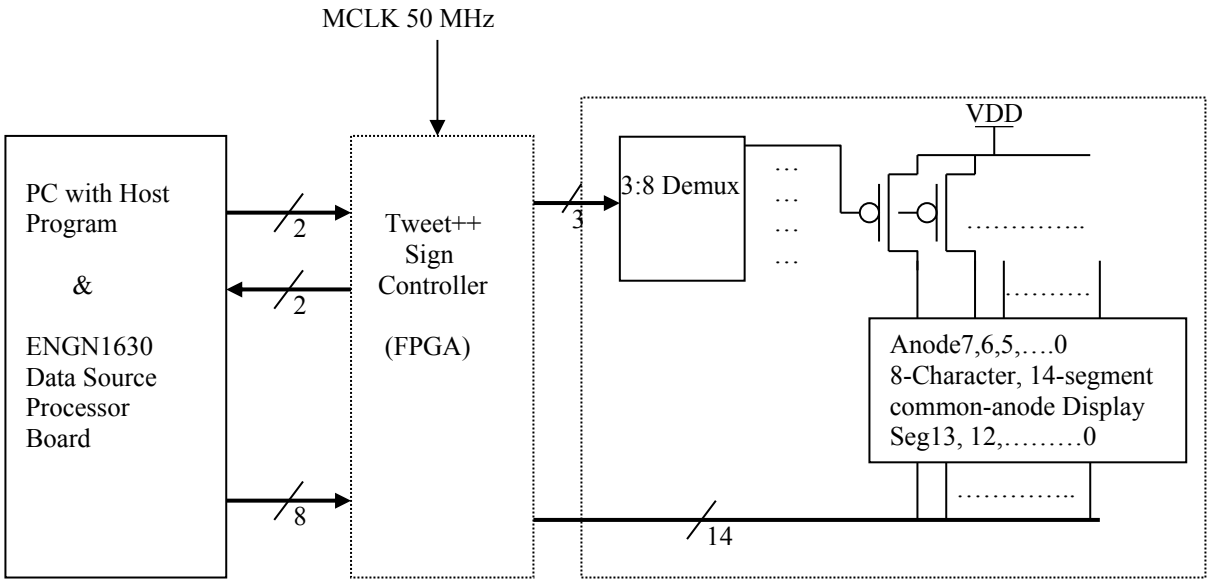



**Figure C-i: Block Diagram of the Tweet++ Scrolling Sign Electronics.** The large block on the right is the 14-segment x 8 digit display on the ENGN1630 custom printed circuit board. The left block is software on a PC feeding a small microprocessor on the same ENGN1630 board to make a parallel data bus to the sign controller. You program the FPGA to store the message and display it continuously.

As shown in the timing diagram, the rules for data transfer are:

- The transfer of a new message to the sign begins when the source board asserts the WR_ENB line HIGH. In response the FPGA controller must assert its WR_RDY line HIGH within 2.0 microseconds.
- The data bus is set up in advance of the DAT_CLK rising edge and should be written to the SRAM starting at address zero on the first rising edge of DAT_CLK after the WR_RDY line goes HIGH.
- In response to a rising edge of the DAT_CLK and after the data has been written to memory, the sign controller asserts WR_ACK HIGH. This must happen within 2.0 microseconds after the DAT_CLK edge.
- Similarly when DAT_CLK goes LOW during data transfer, WR_ACK must be asserted LOW within 2.0 microseconds later. Data will not change on the data bus until the clock is acknowledged.

- Data continues to be written to SRAM on each rising DAT_CLK edge until either the WR_ENB signal goes low ***or*** 141 characters have been written to memory.
- When WR_ENB is asserted low to end communication, the sign controller must assert WR_RDY low within 2.0 microseconds.
- When WR_ENB goes LOW, the controller should also reset the output address counter so that the new message begins from its beginning. (This prevents further display of any old message if the old message were longer than the new.)
- The state of the DAT_CLK signal is not determined while the WR_ENB is deasserted. The clock may remain HIGH or LOW or toggle and its state on assertion of WR_ENB is not determined.

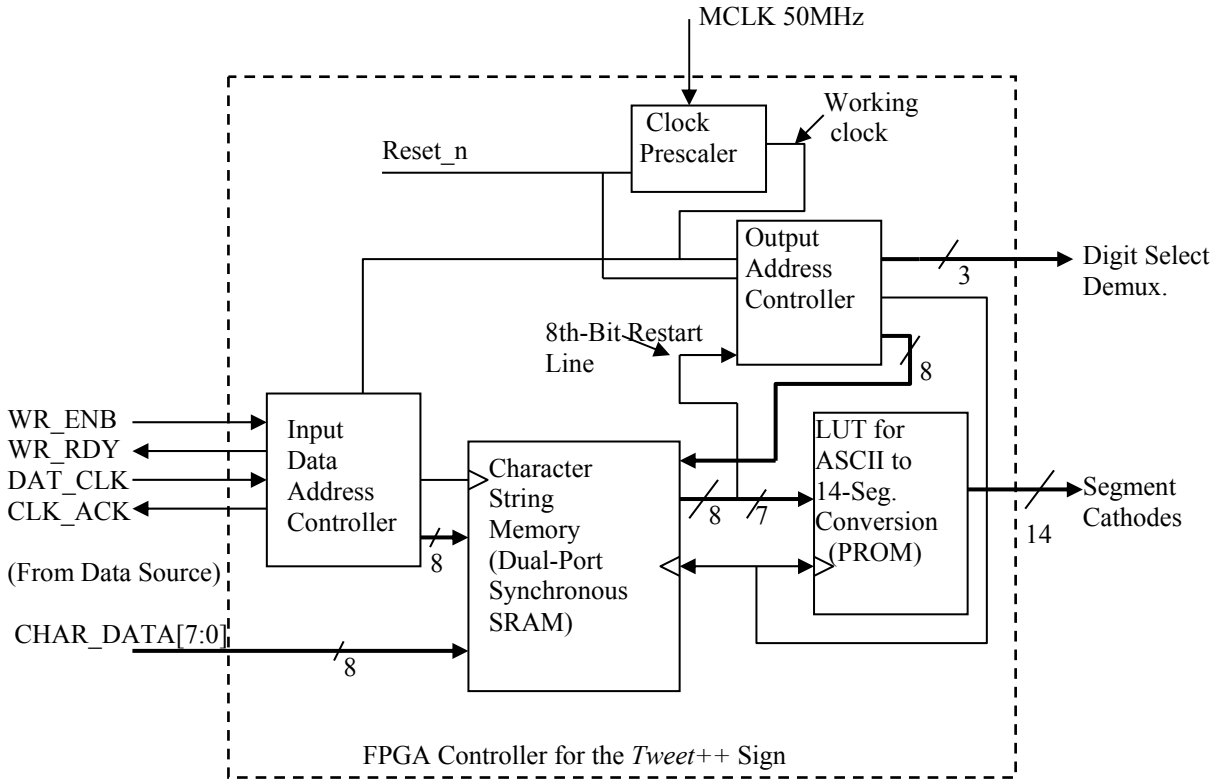



**Figure C-ii: Block Diagram of the FPGA Controller Section of the *Tweet*++ Sign**. The dual-port SRAM and the look-up table PROM are standard blocks within the FPGA. They are located on its periphery rather than being part of the configurable logic array. They are already instantiated on the schematic as are the I/O pin drivers. The clock prescaler is optional but is on the schematic with its Verilog code properly linked.

As with lab B, I have posted a set of project files from which I have deleted almost all the Verilog code. The files include a very simple schematic that already instantiates the PROM for converting the ASCII bytes to the segment drive signals. I have also included all the I/O pin drivers so they match the .ucf file. Dual-port, synchronous SRAMs are standard building blocks from the Xilinx library so I instantiated one of those for you too. (The PROM is also a dual-port SRAM but it is preloaded and its write lines are disabled.) It is possible to have the instantiation code initialize the SRAM and I have used that capability to put a very short message in the text RAM too. If the logic for transferring data from the SRAM to the display is correct, there will be a four-word

English message on the display even before the host computer tries to write to the sign. It is a good idea to design and test in modular steps, testing the output logic before trying to debug the input logic.

I have left out the address controller block for transferring data into the SRAM entirely so you will have to add it to the schematic as well as writing the Verilog code for it. I suggest adding a prescaler for the clock so that you do not need as many stages of counting in each of the address controller. The one used in lab B that counts down to 3.125 MHz is probably a good choice and I put the code and symbol for it into the project already. You are free to modify it if you like.

I have removed some of the schematic wiring but I hope what is missing is obvious. If it isn't, ask me about it. (One problem with new labs is that what I think is obvious is sometimes not so obvious to you.) There are a couple of pin numbers missing in the .ucf file but there is plenty of data here to fill in the values.

**NOTE:** There is a 4-position DIP switch in the upper right corner of the display board. These switches should be in the ON position for this lab. **If the end switch is turned off, the display is turned off.**

**Synchronizer:** The DAT_CLK signal comes from an oscillator on my custom board while the master clock comes from another oscillator on the evaluation board. The two clocks are not synchronized at all and so your control and data signals are crossing a clock domain boundary. As we discussed in class, it is necessary to synchronize the two clocks so that the finite state input controller in the FPGA will be reliable. You must include at least one stage of synchronizer in your design. The TA may ask to see your synchronizer code as part of an FTQ.

**Host Software:** There is a program on the host PC that you use to test the sign. It is in the D:\TestSoftware\en163 directory. When it runs, it asks you for a string to display, downloads that string to your board, and displays any error messages that the interface hardware returns. You are ready for the TA when your display shows the message you sent and the code says there were no errors in the handshaking. The error messages should be self-explanatory. All the signals to the FPGA are available on one header, P2 on the custom printed circuit board. This helps debugging and makes it possible to do the logic analyzer exercise with this lab.

**Discussion:** The sign you build in this lab is an extension of what you did for the multiplexed display in lab 3. The display digits have common anodes and all eight have their segment pins wired in parallel to form an eight-character multiplexed display. To conserve connector pins, the anodes are driven by a TTL 3:8 demultiplexer and the select inputs of that chip are driven by the FPGA. These three select lines essentially address the digits and the display board is laid out with the address zero digit on the right and with increasing addresses going to the left.

The PROM address has to increment as the anodes of successive digits are turned on so that the proper data is on the segment pins for the digit that is active at that instant. To eliminate flicker, the display digits have to be clocked at 300 Hz or higher, cycling through the eight characters of the message currently being displayed. Overall the addresses have to advance at a rate of about one or two characters every second to scroll the display. The most significant bit of the SRAM sets the final message length. That bit is HIGH when the last character of the message is selected and that

signal is an input to the address generator that resets the address counter so the message begins again. (I have padded the messages with spaces to make the effect of the end of line look right.)

The address controllers are primarily collections of counters with some additional small state machine blocks. In an FPGA, any collection of flip-flops that must be clocked synchronously, such as a counter or register, must get its clock signal from one of the FPGA's special **clock distribution networks.** This is the only way to control clock skew between flip-flops that would otherwise cause timing errors. In the project files I have made for you, the use of a buffer of type "bufg" on the schematic for the 50 MHz free running clock called "mclk" causes the mapping and place and route software to put that signal on an appropriate network. I did the same thing for the clk3_25MHz net that carries the prescaled clock signal. (There are eight global clock networks and 16 other clock networks that only reach part of the logic array in the XC3E500 FPGA.) You must be careful to use only clocks from such a network. If you need another clock, you must explicitly force your signal onto another clock network. You can do that with an additional entry in the ".ucf" file. They ".ucf" syntax is, "NET "xxxxx" BUFG=CLK;" where xxxxx is the name of the net.

The PROM in this design as in Lab B is actually a dual-port synchronous static RAM that is preloaded with initialization data and left with the write enable lines and one port permanently disabled. Xilinx builds a number of such synchronous SRAM blocks along with integer array multipliers and several other useful specialized features around the periphery of the array, just inside the I/O pad circuits. All these peripherals can be wired into the logic array. This allows for more efficient implementations of some common digital building blocks without compromising the flexibility of the logic. Because these memory and arithmetic blocks are unique to a given FPGA, the vendors supply special ways to integrate them into the hardware generated from the HDL. Thus you do not instantiate them from simple Verilog but use tools in the Xilinx ISE suite. I have done that part for you.

It is good practice to provide a reset line that forces counters and finite state machines to a known condition when asserted. I have provided a line on the schematic for that purpose and ran its connection on the Xilinx schematic to both the clock prescaler and the output address generator. It is easier for you to debug and to assure reliable startup if you build that line into your design. That line must be HIGH when trying to run the lab as it will freeze the clock if it is low. (One advantage of this is that it stops the music so you can get the TA without having everyone driven crazy by the tune while waiting for him or her.)

Here is a table of the connections between the signals defined in the circuit specification, the pins of the logic analyzer port and the connector to the Digilent evaluation board. The peculiar order of the data pins is a result of a hardware problem. The reset line is still switch on the Xilinx board that was used for that function in lab B. Together with the tables of pin assignment from the FPGA to the extension port of the evaluation board in the user manual there is enough information to rebuild the ".ucf" file.

To get credit for the logic analyzer exercise, you must do two things. First, set up the analyzer with properly labeled signal names and triggering to show the complete handshake and data transfer for a message of some 10 to 30 characters. This is essentially reproducing the timing diagram of Fig. C-iii. Second, change the analyzer connection to P3 and measure the time between a digit select

signal and a segment select signal. The three anode select signals are the three lsb's of P3 and the segment lines are the next msb's.

| Signal | Extension Connector | Logic Port P2 |
|---|---|---|
| CHAR_DATA[7:4], [3:2], [1:0] | FX2_[8:5],[15:14],[2:1] | SIGNAL[7:4], [14:13], [1:0] |
| WR_ENB | FX2_9 | SIGNAL8 |
| WR_RDY | FX2_10 | SIGNAL9 |
| CLK_DAT | FX2_11 | SIGNAL10 |
| CLK_ACK (redundant) | FX2_13, FX2_12 | SIGNAL[12:11] |

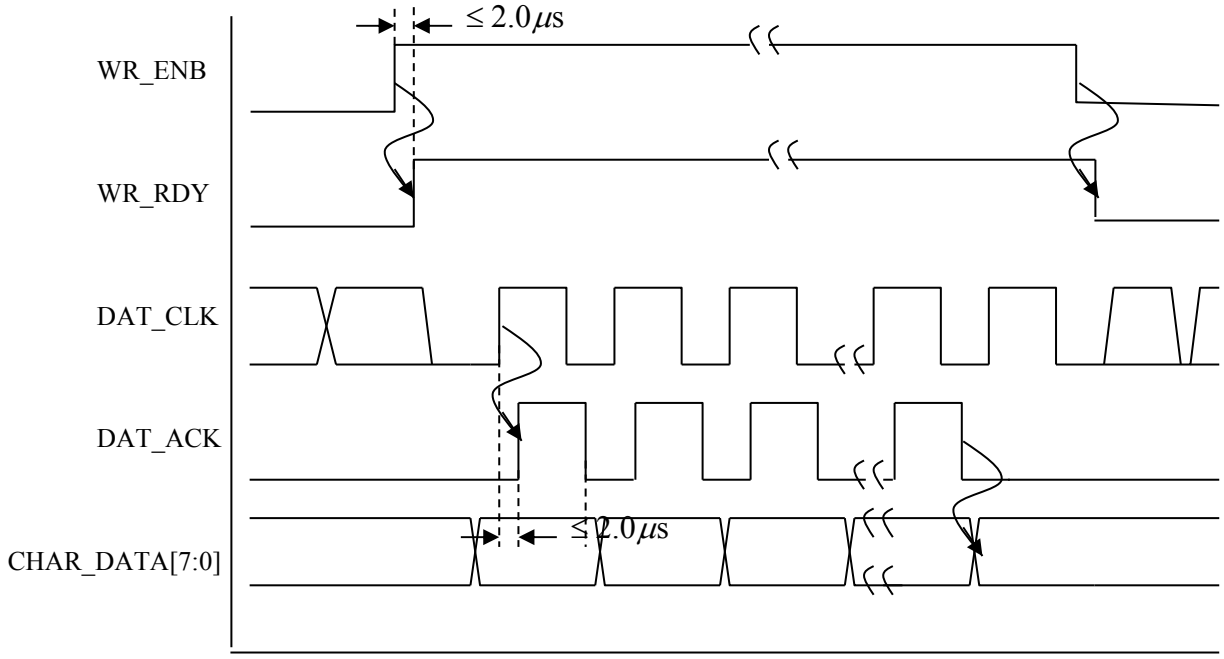



**Figure C-iii: Timing Diagram for the Transfer of ASCII-Code Bytes into the Character SRAM Memory.** Timing for the last character transfer as shown is the end of the 141st character. Alternatively, communication may be ended by WR_ENB going LOW.

### 9.3.4. Lab D:

### A Brain-Damaged Microprocessor Implemented in an FPGA

**Notice:** In this lab you design a large part of a small microcontroller for which I have done the "system" design. I have tried to keep what you do fairly simple but there is a lot of material to read. The point of the lab is not to strain your capacity to read but to show you a basic example of the stored program computer that is the ultimate expression of the logic that you study in this class. Some of you will not study computer architecture formally but all of you should understand the concept of a Von Neumann machine, one of the most consequential of 20th century ideas. You may wish to begin reading at the Discussion section and come back to the detailed description of the lab itself later.

**Table of Contents:**

**Notice:** Rationale for the lab and suggested order for reading sections of the lab description
**Requirements:** Detailed discussion of the Microprocessor and what Verilog code is needed
**What You Do**: Explicit list of required code writing separate from machine operation; DO NOT MODIFY any other modules.
**Tools:** List of available tools to help you work: assembler, code construction spreadsheet, and simulation tool
**Discussion:** Additional design and implementation information including the full instruction set of my design
**The Way It Works:** Simplified block diagram and less detailed description of functionality

**Requirements:** Build the microprocessor shown in the block diagram of Figure D-i in the FPGA on the same evaluation board as used in Lab C. As in labs B and C, I am supplying a schematic in which the component blocks are implemented using Verilog code and a set of Verilog files to implement the blocks, both of which are on the class web site. I have removed most of the code from some of the Verilog files and you have to rewrite them to complete the lab. Don't get intimidated by the complexity of this diagram. Much of that complexity is from the simple multiplexors that implement shared buses. Look at the simplified block diagram of Fig. D-iv for a better sense of what's going on.

The circuit is a full-fledged microprocessor, the principal input and output of which is through a dual-port SRAM. As with the microcontrollers you may have seen in Arduino projects, there are also some output pins for controlling anything you might want. One limitation of this design is that it only has 4 output pins and no input ones. Testing the resulting processor is entirely by simulation software. I provide two Verilog "testbenches" that act as wrappers around your Verilog code. When you simulate the testbench, it loads data into your microprocessor and monitors the resultant operation. If that operation matches the results we have for a known working version of the system, then you get credit for the lab. We are still refining those benches and will have a detailed writeup on how to use them on the class website before Thanksgiving.

Figure D-i shows the system block diagram. All input and output of this "processor" goes through a dual-port SRAM that is shown at the top left except for the output pin data that goes to an output register just above the dual port SRAM. In the schematic file, the SRAM section of the design is

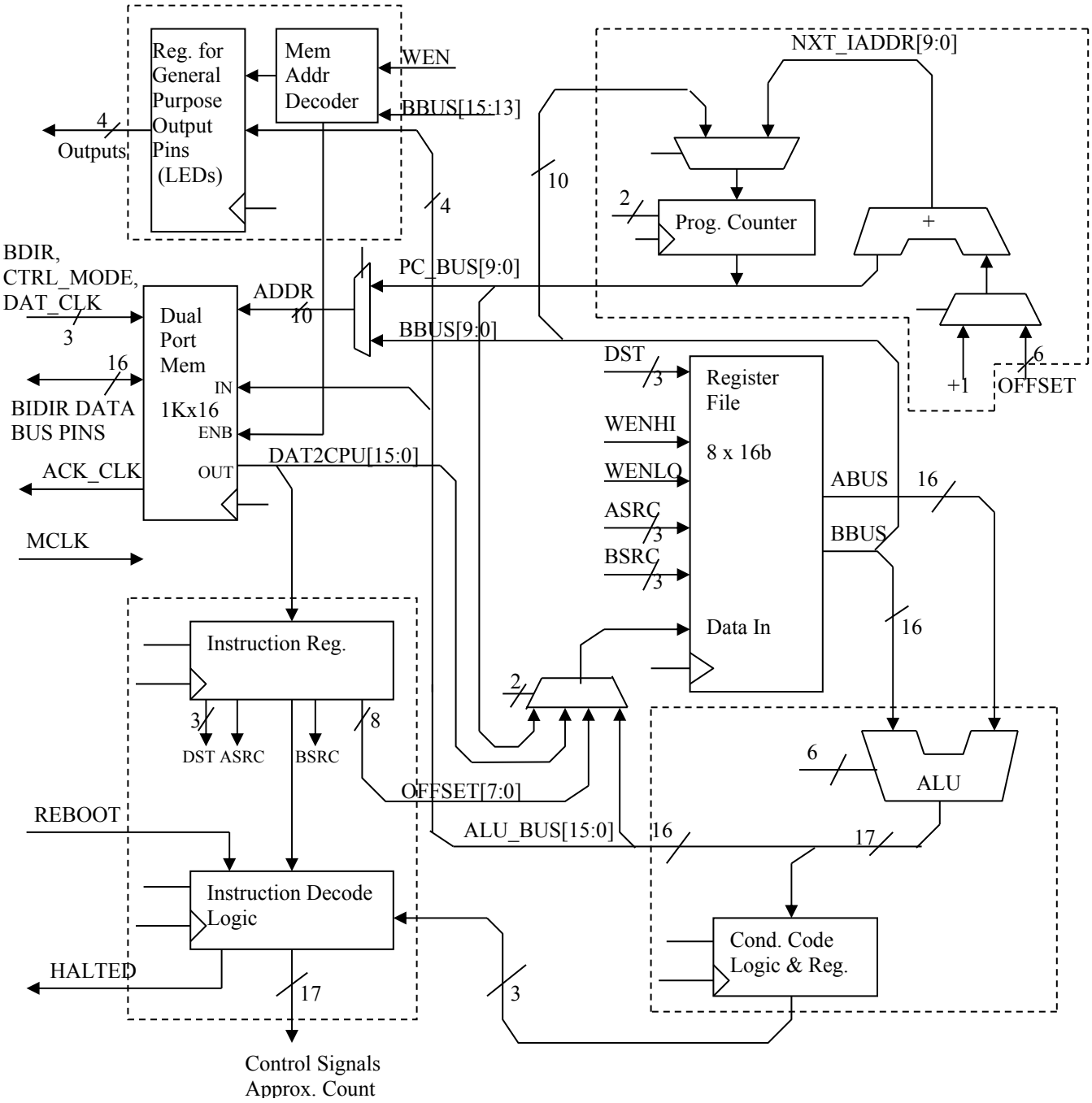



**Figure D-i: Block Diagram of the Full Microprocessor System**

on sheet 3 and consists of the SRAM itself and a state-machine controller. You are responsible for filling in the Verilog code for that state machine, something very similar to lab C. In Xilinx nomenclature the two ports are designated A-side and B-side with the A-side being the I/O side on the left going to my test system. All tests are done by reading and writing data to this A-side port and looking at LEDs connected to the pin outputs.

Some of the blocks in Figure D-i have been grouped together as single Verilog objects on the schematic. On the block diagram these groups have dashed lines around them. For example, the block containing the Program Counter in the upper right also has an adder for incrementing the program counter and multiplexers that allow the contents of the program counter to be replaced with data for jumps or branches in the program. The dashed box around those components marks the block boundary. On the schematic this whole group is in a single Verilog file called "prog_ctr.v". Similarly, the arithmetic logic unit (ALU) and condition code register are grouped in the file "alu_ccr.v" and are also bounded by a box with a dashed line. The output pin register and memory address decoder are grouped in the file "gpo_reg.v." Finally, the components for the instruction register and decoder are together in the file "IR_REG.v".

The control signals that determine the functionality of all the processor units while a program is running appear on the block diagram in two places. They originate in the instruction register block (IR_REG.v). Some of the instruction register bits go directly to different points and these are labeled with names that suggest their functions on both ends of the connection. The places where the other control signals are used appear as short stub lines by each controlled block. For example, six of the control lines go to the ALU to set the arithmetic operation and appear as a stub marked as a 6-line bus on the left side of the ALU itself.

Because I have kept a large part of the instruction register and decoder Verilog intact, the signals in that block already have names that cannot be changed. My capitalization conventions have ended up giving the control signals different names that would appear on the higher level block diagrams. **Table D-VI has an explicit list of the signal names within the IR_REG.v module and their functionality and, if pertinent, their names on the block diagrams.**

The register file is simply a set of eight fast 16-bit registers with a common input bus and two independent read buses. There are multiplexers that select which registers connect to each of the two output buses, ABUS and BBUS. A demultiplexer selects which register is selected for writing, and two write enable lines, WENHI and WENLO, determine whether the high or low or both bytes are written. Separating the high and low byte write enable signals makes it possible to implement the load immediate instructions (LDIMML and LDIMMH). **The register file itself and its wiring to the rest of the system are complete in the files you download and do not require any changes.**

Figure D-iiA shows an expanded view of the interface between the processor and my test board through the dual port SRAM. In the Xilinx schematic file this section of the system is on sheet 3. **There is a finites state machine in the io_control.v module that you will design.** That FSM supplies the bus enable, write enable and address signals for the A-side of the memory which connects to the test system through a simple bidirectional bus driver. Asserting reboot HIGH halts the processor and zeros the program address counter. The testbenches we supply emulate the parallel bus of the hardware test system.

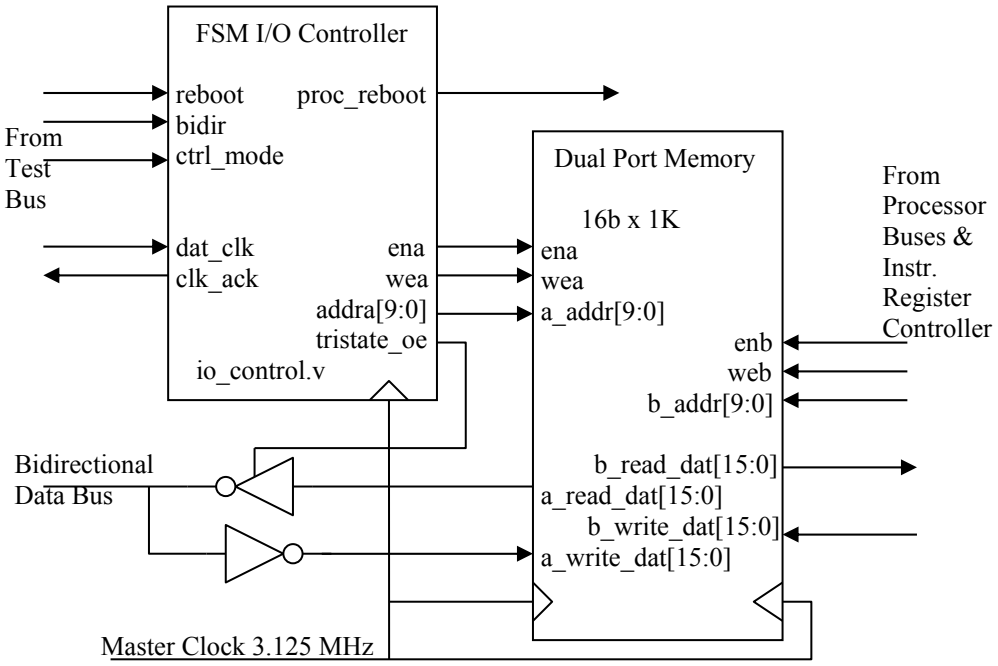



**Figure D-iiA: Block diagram of Input/Output Dual Port Memory Subsystem**

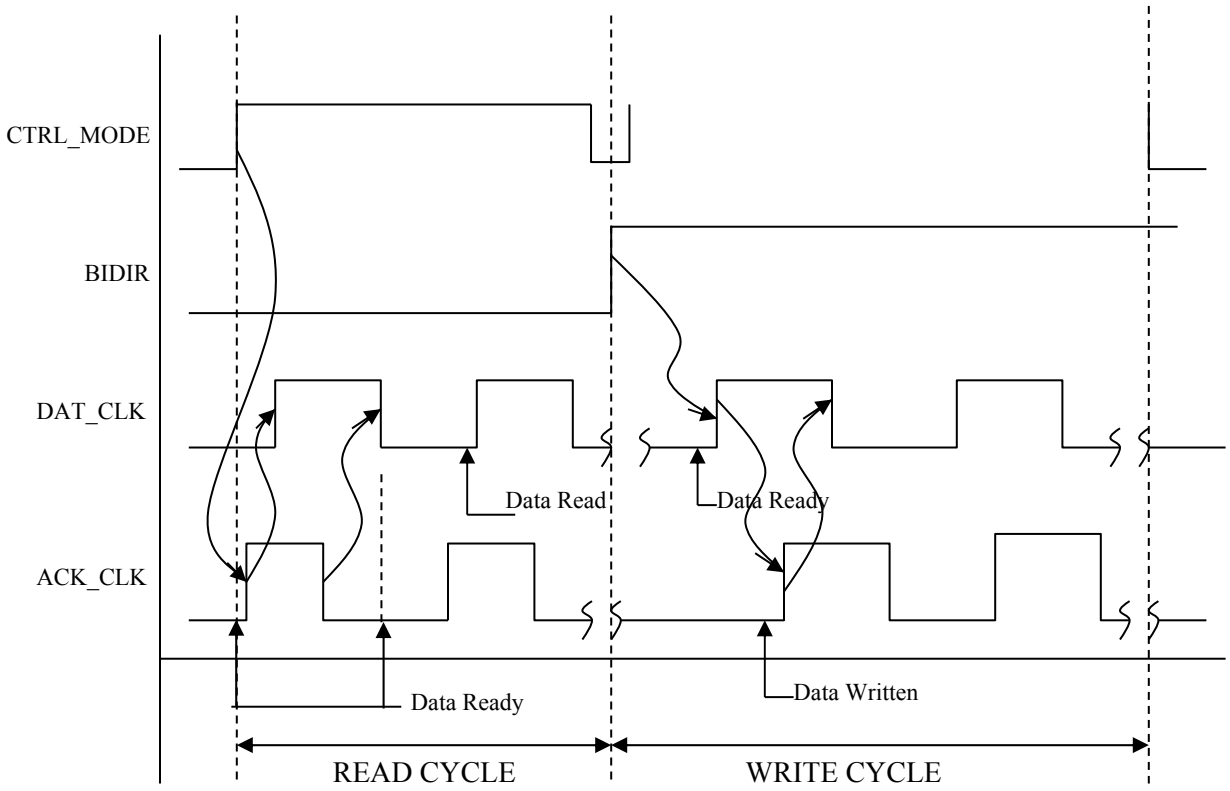



**Figure D-iiB: Timing Diagram for Input/Output between the Test System and the Processor**

Figure D-iiB shows the timing diagram for how data is exchanged between the SRAM and the test system data bus. When the ctrl_mode signal is asserted HIGH, the processor is stopped, the A-side

address counter for the SRAM is zeroed, and the data port of the A-side of the SRAM is connected to the bus from the test system. The pin assignments for the pin connections are already made in the .ucf constraints file associated with the project files on the class web site. The BDIR signal determines whether subsequent operations are reads or writes as shown on the timing diagram. BDIR will only change state when ctrl_mode is LOW. The signals DAT_CLK from the test system and ACK_CLK from the bus controller handshake to guarantee correct data transfer. When all data has been transferred, ctrl_mode is deasserted and the processor resumes operation after the reboot signal is deasserted. More precise definitions of the handshake signal properties are in Table D-I.

**Table D-I: Input and Output Control Signals**

| Name | Direction | Function |
|---|---|---|
| BIDIR | To FPGA (IN) | Bus direction: if HIGH then transfer is from PC to FPGA; LOW is FPGA to PC; signal is ignored if CTRL_MODE is not asserted. This signal may change state only when both CTRL_MODE is not asserted. |
| CTRL_MODE | To FPGA (IN) | HIGH enables read/write to the I/O side of dual port memory. |
| REBOOT | To FPGA (IN) | HIGH holds the FPGA mcu in reboot by continuously resetting the PC to ZERO and inhibiting all writes; LOW runs the program. |
| HALTED | From FPGA (OUT) | HIGH indicates that the FPGA mcu has finished its program and has executed a HALT instruction. |
| DAT_CLK | To FPGA (IN) | Rising edge clocks data into the FPGA or reads data out of the FPGA. (See timing diagram.) |
| ACK__CLK | From FPGA (OUT) | Rising edge clocks read data to the A-side or test system port or acknowledges a write to the FPGA dual port memory. |

Because the implementation of the ALU has to be encoded in a particular way to work with my test programs, the ALU block requires more comment. Figure D-iii shows the internal structure of the unit. The ALU implements add and subtract operations, some shift operations and some bitwise operations. Two shared bits of the ALU function signals (bits 3:2 of funct(5:0) on the schematic) select the operations of the three arithmetic blocks that do these three classes of operation. Table D-II shows the operations of each arithmetic block as a function of bits 3:2. Multiplexers, the select lines of which are wired to particular bits of the funct(5:0) bus, determine the operand choices (funct(5:4)) and the output selection (funct(1:0)). The particular selection pattern is shown in Fig. D-iii as binary select numbers within each multiplexer.

**Table D-II: Function Bits for ALU Blocks**

| Operation: Arithmetic Ops<br>NOTE: ALU operations on the A bus are really on the output of the A multiplexer, ASEL_BUS[15:0] | Funct[3:2] | CY may be non-zero |
|---|---|---|
| A + B | 00 | Y |
| A - B | 01 | Y |
| Pass B | 10 | N |
| Pass A | 11 | N |
| | | |
| Operation: Bitwise Ops | | |
| A AND B | 00 | N |
| A OR B | 01 | N |

| | | |
|---|---|---|
| A XOR B | 10 | N |
| A XNOR B | 11 | N |
| | | |
| Operation: Shifter | | |
| ASR | 00 | N |
| LSR | 01 | N |
| LSL | 10 | Y |
| ROTR | 11 | N |
| | | |

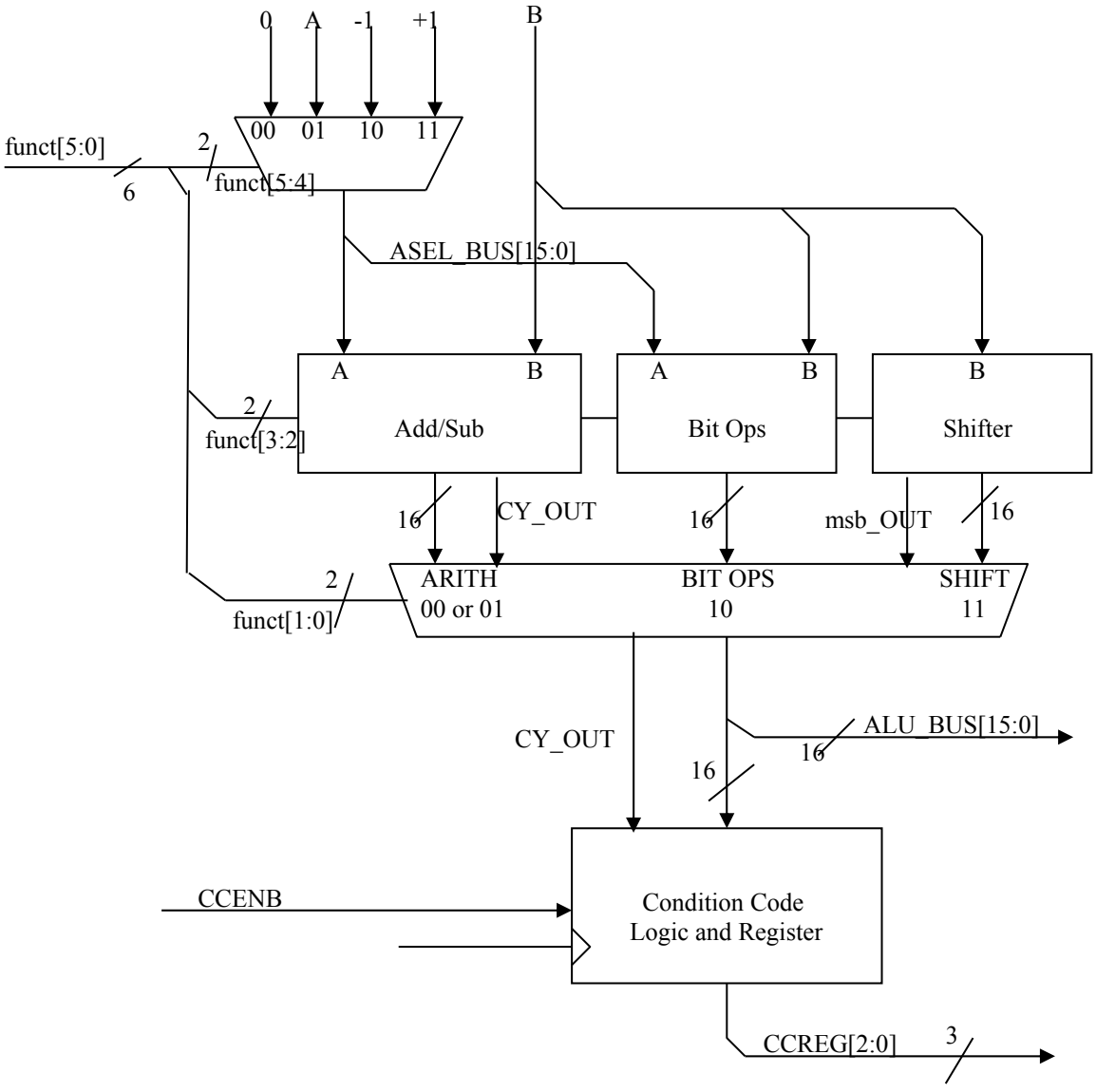



**Figure D-iii: Block Diagram of the Arithmetic Logic Unit (ALU)**

The most difficult part of the design is probably working out the logic for instruction decoding. There are some 17 to 20 control signals that have to be derived by logic from the five opcode bits. **To simplify that process there is a design spreadsheet (labD_codes.xlsx) on the class web site that has a table of all instruction mnemonics, their opcodes, columns for the value of each control signal bit, and a code column.** The control bit values are empty but the code column provides blocks of Verilog to assign values to everything when you fill in the bit values. When the Verilog for all your instructions is set up in the spreadsheet, you can cut and paste the entire column into a suitable 'case' statement in IR_REG.v. That spreadsheet also has tables with the ALU function encoding and is the source of all the tables for lab D in this manual.

**There are only nine required instructions, most of which may be conditionally executed.** Two of these, the LDIMML and HALT instructions are already filled in as example lines in the spreadsheet that you use to turn control line requirements into Verilog so you only really have to write seven instructions. All arithmetic operations operate exclusively on the eight 16-bit registers in the register file block. These are denoted in descriptions of the instructions as R0, R1,…R7 or when one wants to indicate a generic register as "Ra". The notation of (Rb) in the load and store instructions means that the contents of Rb is to be used as an address to the memory and not as data itself. The address reaches the memory by the appropriate selection of the multiplexer between the BBUS and the PC in the upper left of the block diagram. In assembly language the eight required instructions are:

- ADD[c] Rn, Ra, Rb – add the contents of Ra to the contents of Rb and put the result in Rn.
- SUB[c] Rn, Ra, Rb – subtract the contents of Rb from the contents of Ra and store the result in Rn; for this purpose, numbers are presumed to be 2's complement.
- CLR[c] Rn – replace the contents of Rn with all zero's.
- LDR[c] Rn, (Rb) – retrieve a 16-bit word from the I/O memory using the contents stored in Rb as the address and place this word in register Rn.
- LDIMML Rn – load the 8 least significant bits of the instruction word into the lowest 8-bits of register Rn.
- LDIMMH Rn – load the 8 least significant bits of the instruction word into the highest 8-bits of register Rn. This is the complement of LDIMML and between them allows loading a full register.
- STR[c] Ra, (Rb) – store the contents of Ra into the I/O memory using the contents stored in Rb as the address.
- JMP[c] (Rb) – jump to the instruction at address stored in Rb
- HALT – stop all processor operations and signal the I/O system by asserting the HALTED signal HIGH.

I designed the processor with a much richer and more complete set of instructions. The full set is in a table in the *Discussion* section of this lab and the labD_codes.xlsx spreadsheet for this lab on the class web site has the full instruction set and the entire machine language bit assignment for how these instructions are encoded for the processor to use. I lobotomized the instruction set in setting the requirements for the lab to make doing it easier and plan eventually to offer variable credit for the lab based on how many instructions you make work.

Condition codes are the bits in the condition code register that indicate whether the result of the last operation in the ALU met certain conditions. For this processor there are three such conditions: result is zero, negative, or had a carry out. Carry out is generally set to zero except for the instructions listed in Table D-Ia. The condition codes are set for **all operations that write to the register file** except for the three load operations: LDR, LDIMML and LDIMMH.

The notation "[c]" in some of the instructions means that those instructions may or may not execute depending on the results of the last arithmetic operation as recorded in the condition code register. Preventing execution is done by forcing the write control bits to the memory, register file and condition code register to be zero. Two bits ([7:6]) in the instruction encode this conditionality. If those bits are both zero, the instruction always executes. The condition codes signal whether the last arithmetic or load operation result was ZERO, NEGATIVE or had a CARRY_OUT. There is also an assembler spreadsheet (labD_assembler.xlsm) on the class website and that assembler looks for a single character on the end of the opcode mnemonic to determine if an operation is conditional. For example, the operation 'ADDZ' executes an addition operation if the last result was zero. The bit encoding for this conditionality is:

**Condition Field Codes:**

| [c] in assembly language | Conditions for executing an instruction having a condition field | Instruction Bit Pattern [7:6] |
|---|---|---|
| <> | Always execute | 00 |
| Z | Only if CC: zero = 1 | 01 |
| N | Only if CC: negative = 1 | 10 |
| CY | Only if CC: carry out = 1 | 11 |

I am making the lab easier by leaving a substantial part of the Verilog for the decoder in place. **To fill in the missing parts, you need to know how the instructions relate to the bits in the instruction word.** The most significant bit of the instruction determines whether the instruction does arithmetic (IR[15] = 0) or is a load/store/jump operation (IR[15] = 1). The next most significant four bits (IR[14:11]) are an "op-code" that determines the particular instruction. The register numbers are coded in 3-bit groups as Rn = IR[10:8], Ra = IR[5:3], Rb = IR[2:0]. As mentioned above the condition for execution is encoded in IR[7:6]. The non-arithmetic operations have the same coding for op-code and destination register but vary in the least significant eight bits. The general formats are:

| | |
|---|---|
| Bit encoding for arithmetic instructions: | [0][4-bit opcode][Rn][cc][Ra][Rb] |
| Bit encoding for control flow instructions: | [1][4-bit opcode][Rn][cc][Ra][Rb] |
| | [1][4-bit opcode][Rn][cc][000][Rb] |
| | [1][4-bit opcode][Rn][cc][6-bit immediate] |
| | [1][4-bit opcode][Rn][8-bit immediate] |

**What you actually have to do:** As with labs B and C, I supply you with a Xilinx project that has a top level schematic and a set of underlying Verilog modules for each schematic block. You can download the project from the class website on the Handouts page. Certain modules are missing

code that you have to supply. When that code is correct, you simulate the processor operation by adding a Verilog testbench to your project and running the simulation it generates. Testbenches are non-synthesizable Verilog that generate signals and use them to drive your processor module. They report discrepancies between your operation and a known-good run. There is a very short description of what a testbench is in your textbook, Dally and Harting, on page 137. To get credit for the lab, you need the two simulations to run without flagging errors and then need to explain how you did it to a TA.

Table D-III shows the opcodes for the required instructions and the ALU function value required for the arithmetic operations. The ALU function bus value is generated in the instruction decoder based on the value of IR[15: 11] as are all the other control signals. A more complete table with the same information for the entire instruction set is in the design spreadsheet on the class web site.

**Table D-III: Instruction Opcodes and ALU function bus values**

| Mnemonic - assembly language | IR[15] | Opcode | ALU funct[5:0] |
|---|---|---|---|
| ADD[c] Rn, Ra, Rb | 0 | 0000 | 010000 |
| SUB[c] Rn, Ra, Rb | 0 | 0001 | 010100 |
| CLR[c] Rn | 0 | 0011 | 001100 |
| LDR[c] Rn, (Rb) | 1 | 0100 | 001000 |
| STR[c] Ra, (Rb) | 1 | 0111 | 011100 |
| LDIMML Rn, <8-bit imm> | 1 | 0110 | 001000 |
| JMP (Rb) | 1 | 1010 | 001000 |
| HALT | 1 | 1111 | 001000 |

This is the first incarnation of this lab so I have tried to limit what I ask of you to a subset of blocks that may teach you the basics of the idea without taking an exorbitant amount of time. The blocks with missing code are:

1. The PC counter block (prog_ctr.v) is empty. Use the block diagram of Fig. D-i as the guide in writing that code. The program counter reset is synchronous and the entire block is principally multiplexers and a register.

2. The state machine for writing to the I/O side of the dual-port memory (io_ctrl.v) is empty. This is effectively writing out the timing diagram of Fig. D-iiB and its operation is similar to the unidirectional transfers of lab C. The difference is the need for two-way transfer and a slightly different clocking scheme to reflect that difference. NOTE: despite the appearance of the symbol for tristate enable on the Xilinx schematic, the tristate_oe line turns the driver on when that signal is LOW.

3. The ALU block (alu_ccr.v) is missing the code for the three arithmetic logic blocks. The multiplexers are still there, so this is largely a matter of completing a couple of ‘case’ statements.

4. The Instruction Decoder (IR_REG.v) is the hardest part of the system to design since it puts out a large number of control signals. I have removed almost all the logic for deriving

control signals from opcodes but have left the logic for controlling the sequence of actions on startup and for the instruction register itself. To make constructing the control signals tractable I have included an ***Excel*** spreadsheet (LabD_codes.xlsx) on the class website that will compose the Verilog for you if you fill in the values of the control signals for the required instructions. Open the spreadsheet and go to the "IR Verilog Maps" tab. On that sheet, there is a line for each instruction and a column for each control signal. **If you fill in 0 or 1 for a column, the corresponding Verilog is written in a column at the right of the spreadsheet. You just cut and paste into an appropriate "CASE" structure.** The work is to figure out what control signals are needed for each operation.

**Tools:** It would not be wise to attempt to develop a system of this complexity without having simulation available and it would be pointless to build a processor and have no way to program it. To make these activities available to you, I am making the tools I put together for myself at your disposal. The following tools should simplify your activity:

- Spreadsheet "LabD_codes.xlsx" has extensive information about signal codes and has a sheet called "IR Verilog maps" on which you can form the Verilog case statement code for the Instruction Register decoder by specifying the state of each control signal for a given mnemonic instruction.

- The spreadsheet "labD_assembler.xlsm" has two functions. It compiles assembly language instructions that you enter on one sheet ('Entry Page' ) into an assembler listing on another sheet ('Assembler Listing') and the binary text file for download to the processor on a third sheet ('Binary'). The second function is to generate a testbench file for simulation of your design. Testbenches are one of two or three standard methods for simulating small to medium sized systems. See your textbook for a discussion and look at the output that I have left on the sheet named "tb_labd". This is one of the two testbenches you use in simulation to test the processor operation. If you find it amusing, you can write your own program for this processor and run it with a testbench.

- The macro that generates the testbench in "labD_assembler.xlsm" creates a Verilog file called "up_architecture_tb.v". We have created two of these, one for testing the dual-port memory input output on the A-side and one that tests a suite or processor operations. When you run the iSIM simulator, it will show you what your processor does. The run will also give error messages if your device does not do the same thing as ours. (See separate handout for using iSIM.)

**Discussion:** I call this a brain-damaged processor because it lacks a good I/O facility, has no interrupts, handles only 16-bit data, is fairly slow, and has very few, probably non-optimal instructions. I invented the instruction set by looking at the ARM architecture and fitting the features that amused me most into a 16-bit idiosyncratic version. It follows the usual rubrics of Reduced Instruction Set Computer (RISC) architecture by having a uniform instruction size and by having largely single clock cycle instruction execution. More complete processors with more sensible instruction sets are available as intellectual property (IP) from various vendors including Xilinx themselves for implementation in one of their FPGAs where one needs processing power amidst a logic system. Altera, Lattice and Microsemiconductor do the same for their FPGA devices. Many commercial variants on this type of processor are sold widely and used as stand-alone microcon-

trollers. The basic operations really summarize the principles of all general-purpose stored-program, Von Neumann machine computers. When you program your Arduino toy, what you are actually doing is breaking its operation into instructions of this granularity, turning them to binary memory contents and pointing the processor's program counter to the starting point. All else is mere complication!

**The way it works:** Figure D-iv is a simplified version of the block diagram of the system. Look at it first.

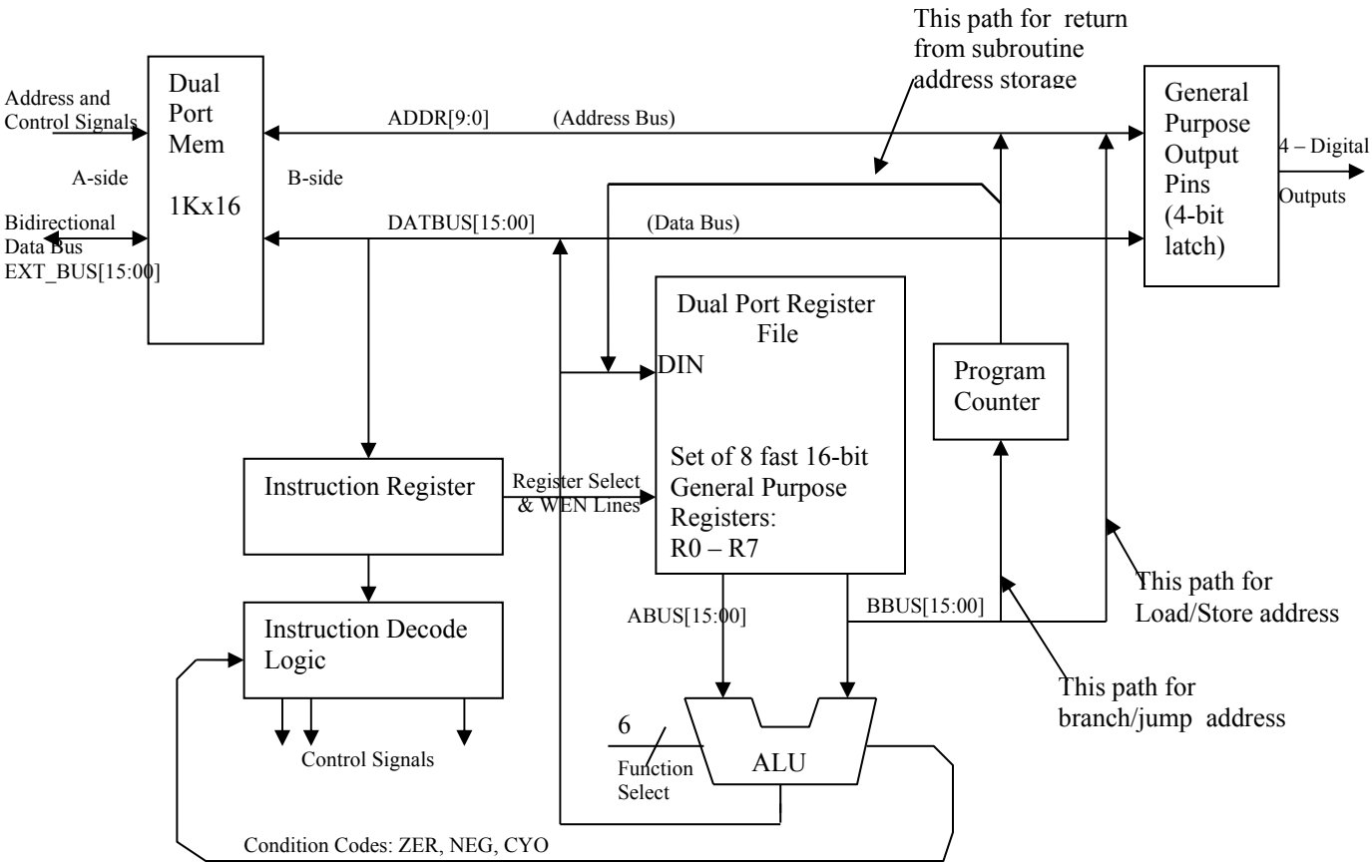



**Figure D-iv:** Simplified Block Diagram of the Brain-Damaged Microprocessor

To understand how this circuit works, consider how a single arithmetic operation goes through the process of being fetched from memory and executed. Arithmetic operations and the LDIMML operation all take two clock cycles to go through the processor but they overlap with the prior instruction in such a way that one instruction completes each clock cycle. Suppose on one rising edge of the clock (MCLK on Fig. D-i) the program counter increments so that it now holds the address of a new instruction in the dual-port RAM. The address multiplexer in front of the SRAM will put the program counter value on the B-side SRAM address lines so at the next clock edge, the memory content on the B-read port, that is, the instruction we want to execute, will be latched into the instruction register. (For the IR to latch the data, it has to be enabled by a control signal generated by the instruction decoder, which is still decoding the previous instruction. The decoder enables the IR when the current instruction will complete on the next clock edge.) When the new instruction is

latched into the IR, the decoder sets all the control lines to do whatever the instruction requires including enabling the target registers if the instruction is to be executed. Finally, on the second rising clock edge after the instruction starts, its results are latched into its destination register or registers.

Table D-IV shows a simple test program that begins at address 0x000 in the I/O SRAM. The first instruction zeros R5 and takes two clock cycles to do so because as the first instruction of the program after booting there is no previous instruction being executed. The program then clears R1, loads the lower byte of R1 with the binary 0x20 and adds that value to itself to put 0x40 into R1. Finally, it loads the 0x40 into the SRAM at memory address 0x20 and halts. My test program would look to see that the SRAM location 0x20 had 0x40.

**Table D-IV: A Short Test Program**

| SRAM Address | Binary Instruction | Operation | Operands | Clock cycles | Writes to: | ALU funct(5:0) |
|---|---|---|---|---|---|---|
| 0 | 0001110100000000 | CLR | R5 | 2 | reg. file | 001100 |
| 1 | 0001100100000000 | CLR | R1 | 1 | reg. file | 001100 |
| 2 | 1011010100100000 | LDIMML | R5, 0x20 | 1 | reg. low byte | 001000 |
| 3 | 0000000100101101 | ADD | R1, R5, R5 | 1 | reg. file | 010000 |
| 4 | 1011100000001101 | STR | R1, (R5) | 2 | SRAM | 011100 |
| 5 | 1111100000000000 | HALT | | inf. | n.a. | 001000 |

Load and Store instructions take an extra cycle during which incrementing the program counter and loading the instruction register are inhibited while the data is taken from or written to the SRAM. To make the processor more useful as a microcontroller there is a set of outputs that go to some output pins of the FPGA to which in turn there are LEDs connected. You write to those pins rather than to the dual port memory by setting the uppermost bits of the address register word to be Rb = 0xe000 (bbus[15:13] = 0x7) and pointing Ra to a register, the least significant nibble of which is <n>, the desired pin values. (Ra = 0xXXX<n>.) With those upper address bits set, there will be no write to the memory and without them no write to the GPO port.

The overall design of this processor is not really quite as brain-damaged as it might seem. I actually designed it to have a much larger instruction set that is close to adequate - not quite but close. The primary defects are poor I/O that is a function of the FPGA format, a data size restriction to all 16-bit words (no bytes, no long 32-bit words) that is the result of trying to put in features of a 32-bit instruction set without the bits required to implement a full version of such a set, and no interrupts. The lack of interrupts is mitigated by the fact that this processor will never run long enough to be worth interrupting. (An interrupt is a special signal that causes a processor to stop the current flow of instructions and start at some new, preprogrammed address. They are used for such things as alerts for new mouse or keyboard inputs. The use of a periodic interrupt is the basis for multitasking and multi-threading in most complex processors.)

You might suggest that the dual port SRAM is not a very good representative of a practical external interface but it actually does mimic several such standard interfaces. Many microcontrollers have

separate data and program memories and the program memory is filled through a JTAG serial port, more or less the way this lab uses a parallel bus to load the SRAM. The SRAM here acts as both program and data memory. Many larger microcontrollers and microprocessors use external SRAM or SDRAM for program and data but time share a single port for input output and program operation. In use that gives similar latency considerations as the dual port system.

**Table D-V: Full Instruction Opcodes and ALU function bus values**

| Mnemonic - assembly language | IR[15} | Opcode | ALU funct(5:0) | Action |
|---|---|---|---|---|
| ADD[c] Rn, Ra, Rb | 0 | 0000 | 010000 | Add |
| SUB[c] Rn, Ra, Rb | 0 | 0001 | 010100 | Subtract |
| CMP[c] Ra, Rb | 0 | 1101 | 010100 | Compare (Subtract but do not store result) |
| CLR[c] Rn | 0 | 0011 | 001100 | Clear register |
| INC[c] Rn, Rb | 0 | 1100 | 110000 | Increment |
| DEC[c] Rn, Rb | 0 | 0010 | 100000 | Decrement |
| AND[c] Rn, Ra, Rb | 0 | 1000 | 010010 | Bitwise AND |
| ORR[c] Rn, Ra, Rb | 0 | 1001 | 010110 | Bitwise OR |
| XOR[c] Rn, Ra, Rb | 0 | 1010 | 011010 | Bitwise XOR |
| XNOR[c] Rn, Ra, Rb | 0 | 1011 | 011110 | Bitwised XNOR |
| ASR[c] Rn, Rb | 0 | 0100 | 000011 | Arithmetic shift right |
| LSR[c] Rn, Rb | 0 | 0101 | 000111 | Logical shift right, fill with 0 |
| LSL[c] Rn, Rb | 0 | 0110 | 001011 | Logical shift left, fill with 0, carry original msb out |
| ROTR[c] Rn, Rb | 0 | 0111 | 001111 | Rotate right |
| MOV[c] Rn, Rb | 0 | 1111 | 001000 | Move (copy) |
| LDR[c] Rn, (Rb) | 1 | 0100 | 001000 | Load register from memory |
| STR[c] Ra, (Rb) | 1 | 0111 | 011100 | Store register into memory |
| LDIMMH Rn, <8-bit imm> | 1 | 0101 | 001000 | Load high byte with 8-bit immediate |
| LDIMML Rn, <8-bit imm> | 1 | 0110 | 001000 | Load low byte with 8-bit immediate |
| JMP[c] (Rb) | 1 | 1010 | 001000 | Jump to address |
| JSR[c] Rn, (Rb) | 1 | 1110 | 001000 | Jump to address but store current PC in Rn |
| BRI[c] <6-bit imm. Signed> | 1 | 0010 | 001000 | Branch immediately by adding 6-bit signed number to next PC address |
| HALT | 1 | 1111 | 001000 | Stop |
| NOP | 1 | 0000 | 001000 | Do nothing for a cycle |

Table D-V shows the full instruction set as I designed it. My intention eventually is to let students get more credit if their lab implements more of the instructions. My other dream is to give credit

for actually programming the processor to do something - maybe multiply 8-bit numbers. I may make that another lab – lab E?

| **Table D-VI: Instruction Register Control Bit Functionality** | | |
|---|---|---|
| **IR_REG.v Verilog Signal Name** | **Functionality** | **Name on Block Diagram** |
| Cycle | Prevents instruction update when asserted. Internal to the IR block, it is used to generate a second cycle for instructions with two-cycle latency. | |
| cndexe | High implies that the instruction may be conditionally executed | |
| ir_enb | Enables write to the instruction register | |
| wpc | Enables write to the program counter register | |
| funct[5:0] | Sets ALU function | |
| whi | Enables write to high byte of register file | WENHI |
| wlo | Enables write to low byte of register file | WENLO |
| wmem | Enables write to B-side of dual port or to GPO register depending on the address | WEN |
| wcnd | Enables writing the condition code register | |
| mem_addrsel | Controls the memory address bus multiplexer; see multiplexer code for polarity. | |
| file_wrsel | Two bit bus that selects the source for writes to the register file through a multiplexer from the ALU output, the B side of the dual port memory, the program counter bus or the low byte of the instruction register. | |
| pc_jump | Control signal to the program counter module to enable the JMP/JSR instructions or normal operation | |
| branch | Control signal to the program counter module to enable the BRI instruction or the normal increment operation | |

### 9.3.5. Lab E:

**Data Transfer to SDRAM from a Serial Interface**

**Requirements:** One of the commonest digital operations is the transfer of data into and out of memory. The Xilinx FPGA on the same evaluation board used in Labs B, C, and D has a single SDRAM (Synchronous Dynamic Random Access Memory) chip connected to dedicated pins that support full speed operation of that device. It is capable of holding a 512 megabit (32 Mw x 16) set of data. I have programmed the microcontroller on the ENGN1630 support board attached to the FPGA evaluation board as a source and sink for serial data. As in the earlier labs I have provided a schematic with empty Verilog modules for you to design and fill in. To get credit for the lab, you will demo to a TA that a test program on the PC can write and read data from the SDRAM. Because SDRAMs require refresh to retain data, you have to provide that too. You will also show a simulation of the SDRAM connection using a test bench that we supply and you run as is required in lab D. Finally you will hook the logic analyzer to a plug on the support board and will capture a serial data transfer event. This is the threefold form of analysis, design and test required for any hardware project of some complexity.

The operation of the SDRAM uses dedicated FPGA resources that cannot be directly invoked by Verilog functions. To solve that problem, the Xilinx software includes proprietary modules that handle the details of generating the SDRAM control signals. To be sure that you pay some attention to how the SDRAM works, the simulation displays those signals and the FTQ for the lab will be a question on how the SDRAM works and how it responds to those controls.

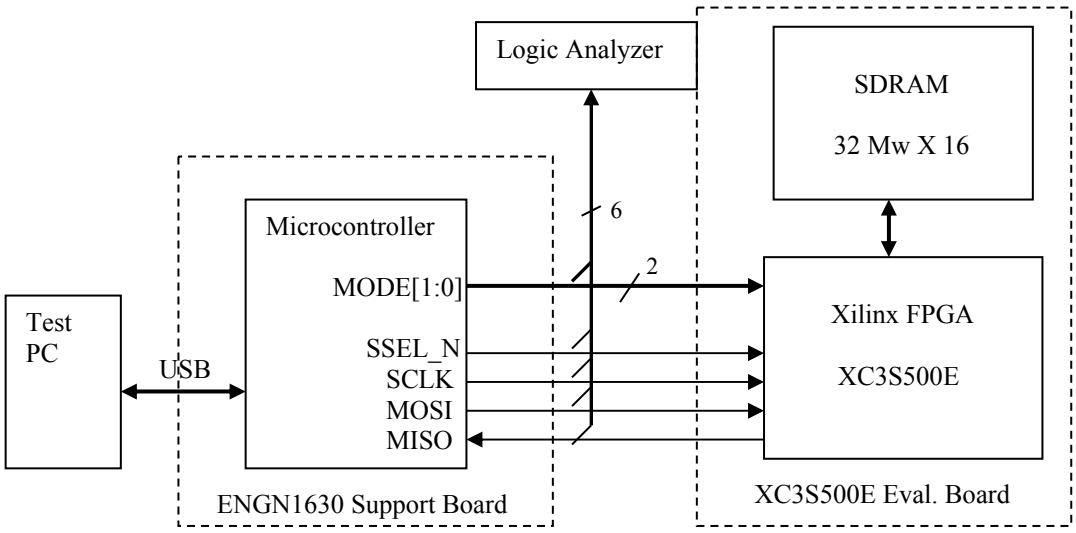



**Figure E-i:** Overall PC to SDRAM Data Storage System with Serial Data Link

Figure E-i shows the overall system in block diagram form. A test program on the PC moves data to and from the synchronous dynamic RAM by a 6 wire interface between the auxiliary microcontroller and the FPGA that is the host to the SDRAM. Fours wires (SSEL_N, SCLK, MOSI, and MISO) implement an industry standard serial interface called the Serial Peripheral Interface (SPI) bus. In the terminology of the bus, the microcontroller is the Master and the FPGA module is the Slave.

The two remaining wires are control signals that are used by the memory control blocks to determine what should be done with data written transferred to the system by SPI or what memory data should be returned through the SPI bus. The coding of those signals, MODE[1:0], is given in Table E-i.

| **Table E-1: Data Transfer Operations for Different Mode Settings** | |
|---|---|
| **MODE[1:0]** | **Operation** |
| 00 | Send high-order 5bits of the address as right justified data in a 16-bit word. Note: the two msb's are the Bank address. |
| 01 | Send the low-order address bits as a 16-bit word. At the end of this transfer, the data will be moved to the address counter and will become the starting address for subsequent WRITE and READ operations. |
| 10 | Write words to the SDRAM |
| 11 | Read words from the SDRAM |

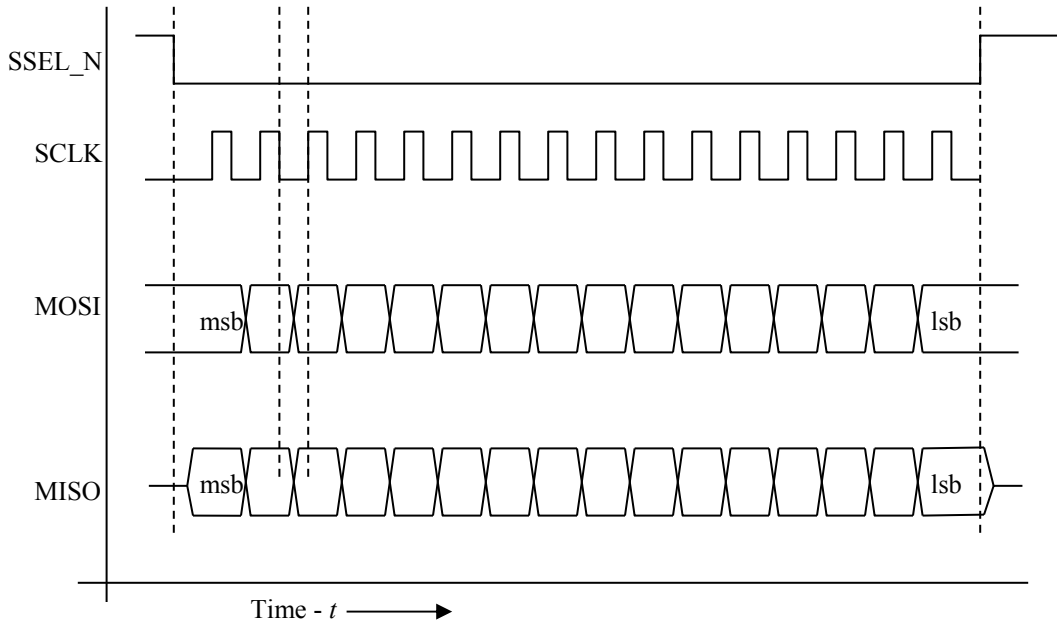



**Figure E-ii:** Timing Diagram of the SPI Bus

Your implementation of the SPI bus transfers all data as 16-bit serial words with the most significant bit first as shown in the timing diagram in Figure E-ii. Within the FPGA there is a single control block that handles all SPI transfers. This is instantiated in the Verilog module "spi_ctrl.v" as shown in the block diagram of the FPGA subsystem in Figure E-iii. The SPI controller has two simple internal registers, one for data from the external system customarily called the Rx register and one for data to be returned to the master called the Tx register. On each SPI transfer for addressing or writing to memory, new data is written into the Rx register. A pair of handshake signals notifies the memory controller that fresh data is available to be put either into the address counter or into the memory at the next memory address. During both Read and Write operations, the address register increments immediately after the operation.

When the mode command is to read data from the memory and the tx_empty signal is HIGH, the memory controller places the next word from memory onto the Tx_data[15:0] bus and signals that data is available by asserting the tx_rdy line HIGH. The SPI controller latches that data into its Tx register, sets the tx_empty line LOW and on the next bus transfer moves the contents of the Tx register to the master over the MISO line. When that transfer ends, the controller sets the tx_empty signal HIGH again. The memory controller removes the tx_rdy signal, increments the address register, places new data on the Tx_data[] bus and resets the tx_rdy signal HIGH.

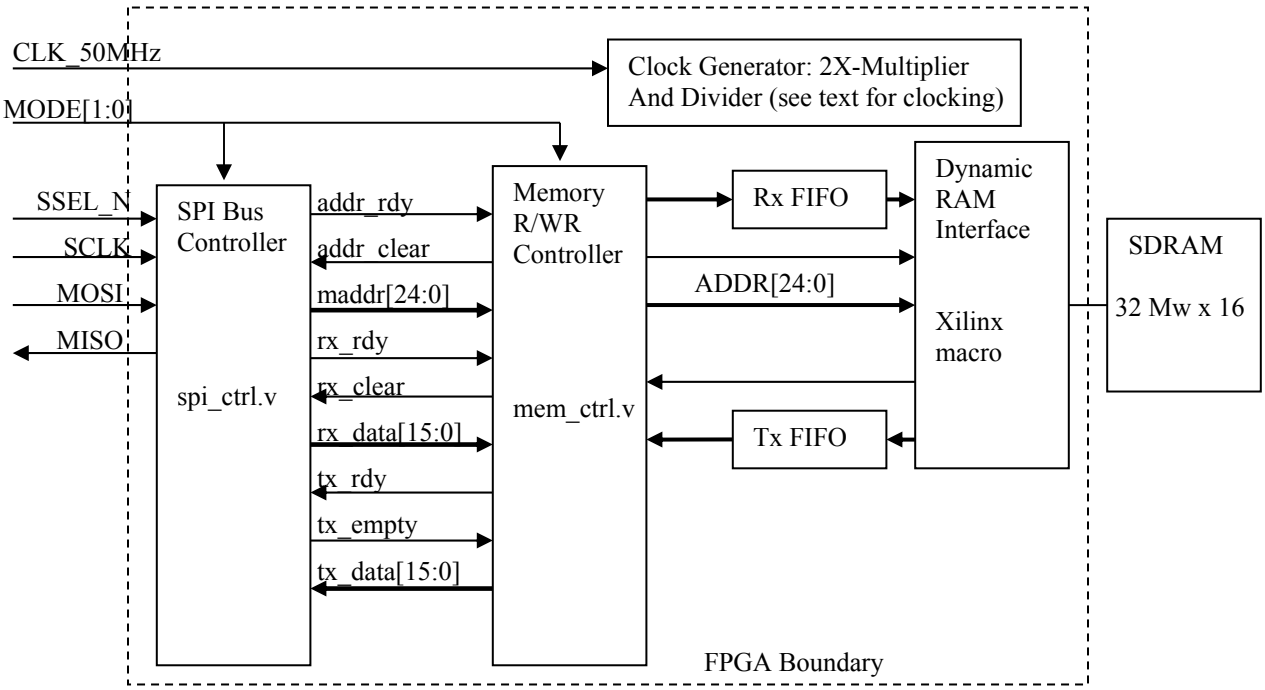



**Figure E-iii:** Block diagram of the FPGA system for data read and write by SPI to SDRAM

The exact operation of the SPI bus is as follows:

- An SPI transfer begins when the SSEL_N (Slave SELect negative true signal) makes a HIGH to LOW transition. At that time the SCLK (clock) is LOW and the most significant bit of the output of the master has already settled. That msb may be latched on the rising edge of SCLK.

- The MOSI line will not change until **after** the SCLK has gone LOW. MOSI will then settle to the next data bit before the SCLK line goes HIGH again.
- After 16 successive bits have been transferred, the MOSI line will remain stable while SCLK goes LOW and the SSEL_N goes HIGH.
- For WRITE to memory transfers, the SPI Bus Controller shall signal the Memory R/WR Controller that data has been received and that the Rx register has been filled by asserting the rx_rdy signal HIGH. When the Memory R/WR Controller acknowledges use of the data by asserting rx_clear HIGH, the SPI Bus Controller shall return the rx_rdy line LOW.
- When rx_rdy goes LOW, the Memory R/WR Controller shall return the rx_clear signal to LOW.
- NOTE: The data on the rx_data[] bus shall only change once each transfer or handshake cycle and shall do so just before the rx_rdy signal goes HIGH. This implies a need for a separate shift register to handle deserialization of incoming data.
- If the MODE command is to set the high or low-order memory address (MODE[1:0] == 2'b00 or 2'b01;), the SPI controller executes the same data capture as for a WRITE data command but transfers the incoming data to the appropriate location in a 25-bit memory address register. (This register drives the maddr[24:0] bus shown in Fig. E-iii above.) If the high address is written over, its data transfer will precede any low address data update.
- When the low address data is written, the addr_rdy handshake signal will be asserted HIGH. It will remain asserted until the memory R/WR controller asserts the addr_clear signal HIGH. The SPI Bus Controller will then remove the addr_rdy signal and the memory controller will deassert the addr_clear signal. This is exactly the same sequence of signals as those for the rx_rdy/rx_clear handshake except using the signals addr_rdy/addr_clear.
- During the handshake for a low address update, the memory controller will replace the current SDRAM address with the data on the maddr[24:0] bus.
- If the MODE command is a READ operation (MODE[1:0] = 2'b01;) and the tx_empty line is HIGH, then the Memory R/WR Controller shall place the DRAM word from the current memory address on the tx_data[15:0] bus and assert tx_rdy HIGH. The SPI Bus Controller shall store that datum in its Tx register, return tx_empty to LOW, and wait until the next SPI READ transfer.
- When that transfer is has no more need for the data in the tx register, the SPI controller signals the end of transaction by asserting tx_empty HIGH. The memory controller responds by removing the tx_rdy signal, incrementing the address register, placing the next datum on the Tx_data[15:0] bus and reasserting the tx_rdy signal. This cycle is repeated as long as the Master performs READ operations.
- To avoid incorrect data in the first READ after an address change or write operation, the SPI controller shall set the tx_empty flag when the MODE command changes the SPI controller mode to READ.
- Because a READ operation does not use the data from the master, the SPI controller shall omit the rx_rdy/rx_clear handshake operation for READ operations.
- For all SPI transfers except the READ transfers, that is, for WRITE and address initialization, the SPI controller shall return the word received from the MOSI line on the MISO line with the timing shown in Fig. E-ii. To meet timing requirements, the returned data must be placed on MISO immediately on start of a transfer before the first rising clock edge.
- MISO is a tristate output because the SPI bus is usually shared between several peripherals. The output is enabled when the SSEL_N is LOW.

Still to go:
Memory controller operation
Xilinx DDR-SDRAM interface
Refresh operations
Clock generation for the SDRAM, the SPI state machine, etc.
FIFOs

### 9.3.6. Lab F:

### JTAG Boundary Scan Register Implemented in an FPGA

## 9.4. The Logic Analyzer Tool

### Learn to Use a Logic Analyzer

We have a dozen logic analyzers purchased at a very deep discount from and courtesy of Agilent Technology Inc, the successor to Hewlett-Packard's Test and Measurement Instrument Division. I should like to express my gratitude to them for the gift. These instruments are essentially multi-channel oscilloscopes especially designed to display signals from logic systems. Depending on the particular model, these will display 34, 68, or 102 inputs simultaneously. Input signals are converted to logic levels and may be displayed either as timing diagrams or as state tables. Synchronization is extremely flexible, being software-selectable on such logic events as the occurrence of a particular word or the first edge of a given signal after a certain word. Seven of the instruments also have a two-channel digital oscilloscope that can be triggered from the logic analyzer section. You may use that capability in lab 6.

**Requirements:** We are requiring you to learn to use State/Timing section of this instrument to characterize one of the labs that are open to simulation in lab 9 because it is the fundamental tool for logic circuit analysis and debugging. You must demonstrate to a TA that you have the same signal pattern as you would be required to simulate for lab 9. The signals you display and the trigger points and time scales to demonstrate are those called out in lab 9. You also **must set up the screen labels of the analyzer** to show the proper names of the signals, that is, the same mnemonic names that appear on your schematic. (The schematic must have sensible mnemonics!) Ideally, you should plan on doing both the software simulation and the logic analyzer measurement on the same lab. Direct comparison of the simulation printout with the analyzer screen would certainly be the simplest way to convince the TA of your accomplishment. When the TA is satisfied with your measurements, she will sign off your scorecard in the usual way. I reserve the right to change the way this requirement is met if I am able to get the analyzers to print their results out on the printers in the computing facility.

Attaching an analyzer to your circuit should be fairly straightforward. The CPLD board has a header on it and there are 20-pin receptacle pods you can attach to the logic analyzer to measure 16 separate signals at once. Consult the table of CPLD board connections in section 10 of this manual. There are also multi-pin clips in the lab that fit over DIP integrated circuit packages to contact all pins at once. We have 16, 20, 28, and 40 pin versions of these. If you space your devices astutely, then you will not have a problem with 14, 18, or 24 pin packages as well.

The one thing I ask is that you respect the equipment. The analyzer leads are fragile and we expect some breakage. They are also expensive -- more expensive than we can afford to replace unless you use some care. Most of the time only the end breaks off and we can put it back on. We can only do that if we have both the end and the wire. **IF A LEAD BREAKS, PUT ALL PIECES OF IT IN ONE OF THE MARKED PLASTIC BOXES FOR BROKEN LEADS!**

Well-written manuals on how to set the controls are with the machines in the lab – check the drawers in the back bench. The TAs can be of some help, but some of this you will have to work out by trial and error because the TAs have not had a lot of experience with the analyzers.

Many of you will use the analyzer with relatively slow clock rates, *e.g.*, with lab 7, and may find that the logic analyzer does not want to display a very long (by its standards) set of data. To extend that time, select the Waveform screen, the one on which you are viewing your timing diagram. On the top edge of the screen select the control box labeled "Acq. Control." This will open a new screen with four control boxes, one of which is labeled "Sample Period." When selected, that will show a period of 4 ns by default. Change it to 10 microseconds by either mouse clicking or knob turning. Close the several boxes and make your measurement.

## 9.5. Requirements for a Schematic Drawing

### Schematic Capture Including Physical Part Information

The purpose of a schematic is to specify how an electronic system is assembled. It is not usually a tool for logic design. Even though this course is primarily about logic design, we want you to see at least a little of the more extended design problem.

A schematic is most obviously a netlist of the circuit interconnections. Properties attached to the symbols carry information on the physical package of the part and on the cost and vendor part numbers. Automated systems such as the DxDesigner suite that is discussed in section 11.3 take information from the schematic and prepare Bills of Materials, simulation netlists, and augmented netlists for printed circuit board design. The schematic along with layout information can be the front-end for tools in the same suite that analyze signal propagation. When programmable logic such as FPGAs are used, the tools optimize the use of the programmable logic pins to lower the cost and improve the functionality of the printed circuit.

You are required to draw a schematic using DxDesigner that could serve as a starting point for a PCB design. Before you can use this package, you may need to set up appropriate directories using the instructions found in a handout on the class website. You must attach a minimal set of properties to certain components so that the tool also provides a bill-of-materials (BOM). Also remember that power supply connections **generally require bypass capacitors and at least one of these** should appear on your diagram. Here are the general guidelines:

1. **Initial Setup:** You must begin each drawing by enclosing it in a frame, called a "sheet", which corresponds to selecting a paper size for the full-size picture. (Although your system may fit a single sheet, a more complex system may run over many sheets.) The frame also sets up some crude position coordinates and has a block in the lower right corner for title information. You must fill in that block with a title that is consistent over all sheets of the schematic. You may add a subtitle unique to each sheet for multiple sheet drawings. Also fill in the date, your name, and a version number. There is a place for mandatory sheet numbering. (Even a single-sheet schematic must be marked "SH 1 of 1".) The sheet numbering is supposed to be automatic but check to see that it is correct.

2. **Title Block Setup:** I have tried to set the system up so the sheet is there when you open your project. Should that not be the case, you will have to add the sheet frame. Sheets are treated as standard components and are found in the "Mechanical" component library as the symbols: "csheet_bu.1", and "csheet_bu_rev.1". A single "csheet" should be about right for your first drawings. The first sheet must always use the "cheet_bu_rev.1" symbol. Using that symbol will put a revision line at the top of the documentation block. Repeat the version number there and add a line of text indicating that this is the first version of the particular system.

3. **Components and Symbols:** The part symbols for a drawing are invariably drawn from a symbol library. In this way, symbols are standardized, and much other information such as

the package type, vendor part number, simulation model parameters, etc. is bound to the schematic at the same time with no extra effort from the designer. Most of the parts you will need are in the "en163" library. This library includes all the integrated circuits from your kit as well as resistors, capacitors, mechanical switches, etc. The connections for power and ground are in the "Mechanical" symbol folder. Notice that there are different types of capacitor in the symbol set and you must use the ones that match your components. There are also multiple power symbols for different voltages. "VDD" is the symbol for the generic +5 volt source. (VDD3_3 is the symbol for a 3.3 VDC source, etc.)

4. **Signal Flow:** When practical, signals should flow from left to right and top to bottom. This is not always possible and should not be regarded as an ironclad rule, but it does make reading a schematic easier. (We reserve the right to reject a schematic that violates this rule in a confusing way.) The commonest exception is that jacks and plugs, which contain both inputs and outputs, may be drawn with all connections together on either side of the page. Bidirectional lines may approach an integrated circuit from either side of the block.

5. **System-Level Interconnections:** Connections between points in a system are made by drawing a wire, called a "net," between them. When parts are too far apart or appear on separate sheets (pages), you connect separate instances of wires by giving all of them a common label. It is good practice to label even a signal that only runs on a single line if it will be referred to in simulation or may be important to probe in debug. [See also the discussion of "labeling" in section 11.3.] Often, signals have to leave your system through some external connection. Sometimes this is through an actual connector, but you will sometimes do it with just wires from your breadboard to instrumentation. In the latter case, bring the dangling ends of these nets near one another at a side of the sheet and label each one appropriately. These labels are also a guide for the human reader. To help him or her understand the purpose better, add a text description of the set of connections.

6. **Connectors:** Connections through real physical connectors use appropriate symbols. Like other actual components, connectors also have information about their physical structure in their library descriptions. You will probably not have any connectors in your early labs because you only use solderless breadboards. In lab A and the Xilinx labs C through F, there are computer and test jig connectors that you can put on a schematic from the en163 library. An example of such usage is embedding the CPLD board in a schematic. For example, the symbol for the DIP-24 plug from the CPLD-II board is in the en163 library as "cpld_board_dip.1".

7. **Labeling:** The drawing program lets you label a signal wire or net when you select the line with the left mouse button, right click "Properties" to open the properties pane, and modify the Name line in the properties list. The label is an alphanumeric text string that identifies the net in a readable form. In both simulations and netlists, a wire is referred to by its label. (Simulations use the label as the name of the signal carried by that net. Unlabeled nets are automatically given unintelligible but unique names.) The label will follow a wire no matter how far it is extended in a drawing or how much it is moved. Labels even cross from sheet to sheet within a given project. All nets in a project that have the same label name are assumed to be connected together. Assigning **a label is NOT** the same as just placing a piece of text next to a wire! **Do NOT use the Text button** to add labels.

At the very least, **you must label** all intersheet signals, all wires which come from outside the system, all signals that are to be simulated, all bus wires, and the principal control signals. Any signal that is named in the requirements of a lab should have that name as a label in the *DxDesigner* schematic. How many of the remaining wires are labeled is a matter of taste. The goal is a balance between a well-annotated drawing and a cluttered one -- an aesthetic judgment. It is probably better to err on the side of too many rather than too few labels.

8. **Bus Notation:** Frequently several different but closely related signals are connected along roughly parallel paths between a particular set of chips. Such a set of lines is called a bus. Typical examples include data, address, and timing buses in computers. The example diagrams have several buses on them. To save drawing area and to emphasize the parallelism of their function, one usually draws all of these wires as a single line, representing a cable or a multiconductor set. *DxDesigner* offers a special command for this purpose, on the menu "Add/Bus", which draws a bus as thick line. All buses must be labeled, preferably with a very short abbreviation of its function followed by a pair of numbers in brackets. The numbers tell the range of subscripts that distinguish individual wires. Each wire connecting to a bus must be labeled with the bus name concatenated with a signal number within the range of subscripts. For example, ADDR15, ADDR14, ..... ADDR02, ADDR01, and ADDR00 all connect to the bus ADDR[15:00]. (The order of the numbers, that is [15:00] versus [00:15] is important when joining bus segments. To minimize chances of errors, industry usage is high to low, *i.e*., [15:00] is preferred.) *DxDesigner* allows one to label wires with automatic generation of a number suffix in certain circumstances. [See the *Viewdaw Hints* section.]

Remember it is only appropriate to use the bus convention for **closely related** signals. It is not correct to use it just to simplify drawing unrelated signals that happen to be going in roughly the same direction across the screen or paper. Use the bus drawing convention sparingly until you thoroughly understand the idea.

9. **Power Connections:** The five-volt power supply net automatically has the name VDD. (The label VCC is not used for power at all even though that is the usual TTL pin label unless the designer makes that arrangement herself.) Similarly the ground connection is labeled GND. Power connections to purely digital ICs are generally not shown but are handled by symbol "SIGNAL" properties that assign certain pins to the appropriate power nets. Analog and mixed function chips have their power connections shown explicitly on the part symbol, and these must be explicitly connected to the appropriate power nets. There are additional power symbols in the "Mechanical" library for most common supply voltages. (Only certain names are legal for power networks.) Bypass capacitors on power connections are necessary in almost all systems and so should show on your diagram too.

10. **Component Orientation:** By convention, the orientation of some symbols is fixed. For example, you may not rotate the power and ground symbols because an upside down ground symbol becomes a positive power supply symbol and an upside down VDD symbol becomes one for a negative supply. There are clear advantages to being able to have passive components, e.g. resistors, capacitors, etc., sitting either vertically or horizontally. You do not want the accompanying text to rotate with the device. To make this easy, all such

components have both a vertical and horizontal view in the library. You find the alternative by hitting on the list box at the bottom of the symbol viewer and selecting the alternate view to the default.

11. **Reference Designators and Component Identification:** Every physical component, whether integrated circuit, switch, resistor, transistor, etc., must have a reference designator. This is a string that uniquely identifies the particular component in the system. There are standard conventions on the choice of such names. Typically an integrated circuit would have the form U1 or U22, for example. A resistor would be R3 or R4, meaning the third or fourth resistor on the drawing. Each component in the library comes with a reference designator. For example, all integrated circuits have the reference designator U? in the library. The first character or characters are the standard prefix characters for that part type. You have to replace the question mark with a number unique to particular part. You assign the reference designator by selecting the component with the left mouse button (you may have to enter "Select Mode" first), right-click on Properties, and scroll down to REFDES to enter the new name. Then hit <Enter> to fix the new value. Remember that the first character of the reference designator should be the same as the library default character.

Do **NOT** try to distinguish different sections of the same component with a suffix, *e.g.*, U3A or U3B. *DxDesigner* will consider these to be two different components altogether. Parts with multiple sections, like the 74LS00, are given the same reference designator for all sections, making sure that pin numbers of all sections are different. On these parts, you can change the section for a given instantiation by TYPING "<space>slot #" where # is the section number you want to use. For example, if you select a section of an SN74LS00AN chip and type "<space>slot 1<Ent>", the gate section will use pins 1, 2 and 3 while "<space>slot 4<Ent>"will result in a gate using pins 13, 14 and 15.

12. **Passive Components:** Your schematic is not complete unless it shows all passive components, i.e., resistors, capacitors, switches, connectors, etc., that are required for full functionality. For example, if you use a dip switch to enter data as in lab 1, then both the switches and the pullup resistors required for reliable logic levels must be drawn. The resistors must be tied to VDD explicitly with the appropriate symbol. Similarly, bypass capacitors must be shown with their power and ground connections. The capacitor type must match the kind of device in your kit. It is sufficient for the course to show one bypass capacitor (0.1 μfd) for every two logic packages but there must be one for every integrated circuit with memory, i.e., flip-flops, RAM, counters, etc. The resistance and capacitance values of symbols from the library are bogus and you must reset them to the appropriate values on your schematic. See the section 11.3 on setting component properties.

Every symbol in the en163 library should have properties for unit cost, supplier, and vendor part number. For simplicity assume the supplier is always Digikey, Inc. (www.digikey.com). You must also set the exact part number and if necessary the unit cost for your passive components from the DIGIKEY website. (You get the part number and unit price from the DIGIKEY website and add it to the symbol the same way you add a resistance or capacitance value.)

13. **Bill of Materials:** When the schematic is finished, you must also print out a Bill of Materials that was generated automatically from the schematic. On the Tools menu of DxDesign-

er, select “Create Partlist (Common)”. The dialog should be set up properly so you only need to Run it. This will leave a file called “<your schematic name>.txt” in your U:\wv directory. This is a text file that does not print well directly and does not have cost calculations. There should also be an Excel file called "ProcessBOM.xlsm" in your U:\PADS_WDIR directory. Open that Excel file, enable macro processing, and run the macro called Process BOM. That will open a dialog for you to browse to your text BOM. When you select that text file, the spreadsheet will format the data properly. Rename the spreadsheet, save it, and print the processed file. Print landscape style and set margins on the page so the printing is neat and all columns are printed within a single line.

## 10. The Engineering 163 CPLD-II Board

**Introduction:** I have built a small printed circuit assembly that I call the Engineering 163 CPLD-II Board as a partial response to the obsolescence of the construction style that you use in the lab. The integrated circuits in your kit are in DIPs or Dual In-line Packages that can plug into proto-boards. They are increasingly difficult to get because they really do not represent current packaging technology. Still, if you actually want to design anything electronic, you will not find it in a single package already – otherwise it would not be a "design" problem for you! My goal in this course is not simply to teach Boolean algebra but to develop the mindset that can partition a system, design its pieces, assemble it, and understand it well enough to debug the final product, whether it has simple soldering problems or more complex oversights of design specification or implementation. To achieve that goal within a hands-on context requires finding ways to use modern packaging while still allowing you to assemble and debug the product.

The main problem with conventional gates such as those you use for labs 0 and 1 is that they are not very flexible without extensive wiring. Programmable Logic Devices or PLDs attempt to solve this problem by building large AND-OR circuits that can do SOP logic and by connecting the logic outputs to simple registers. The gates themselves use NOR gate logic of the kind we first looked at in class, namely N-channel MOSFETs in parallel and connected to a pull-up device. Their flexibility comes from using floating gate transistors for the pull-down devices in the AND plane part of the NOR gates. By suitably stressing these transistors with a relatively high voltage, one can raise or lower their device threshold voltage, effectively wiring them into the circuit or taking them out. (The programming voltage is generated on-chip. I talk about how these devices work in class.) This is wiring flexibility without physical rewiring. That "rewiring" is form of programming and is most easily done from compiled text descriptions of the logic in the Verilog or VHDL languages. Once programmed, the circuit principles are the same as other gates, but the choice of an N-MOSFET circuit rather than CMOS for the logic sections has consequences. In comparison to CMOS static gates like the 74ACT04 chips in your kit, PLDs can:

- Implement multiple SOP expressions with large numbers of inputs but usually with a limited numbers of product terms. In comparison, CMOS gates generally do only one product term or a simple OR of a small number of inputs.
- PLDs are a little slower than discrete gates and much slower than gates that are interconnected within a chip. Discrete gates pay a very large speed penalty for connecting off-chip, the only way that you can do external wiring without changing circuit manufacture.
- PLDs draw some power continuously because of their pull-up devices. By comparison, static CMOS gates draw almost no power unless they are changing output state. Almost all power in static CMOS gates is dynamic power from charging and discharging stray capacitance. (As with many blanket statements about electronic technology, this one has to be qualified. The newest CMOS technologies at the smallest feature sizes do have significant currents from junction leakage.)

With the CPLD Board, you can embed a current generation Complex Programmable Logic Device (CPLD) into a breadboard system and make it interact with other components. This chip, a Xilinx XC9572XL, is capable of implementing substantial amounts of logic including state machines and comes in a 44-pin plastic leadless chip carrier (PLCC) that cannot be hooked to a protoboard without something like my new printed circuit board.

CPLDs are one of two classes of device that attempt to solve the problem of providing a flexible means to build large amounts of custom-tailored logic without the development costs of custom or semi-custom integrated circuits. The other class of such devices is the Field Programmable Gate Array (FPGA), one of which is used as the basis of labs C through F. They address markets for prototype development and small volume manufacturing. (By small volume I generally mean something under 20,000 to 100,000 units per year. The non-recurring costs for integrated circuit development are quite high and generally limit the use of application specific circuits to products with large unit sales.)

Of course, there are tradeoffs to be made in deciding whether to implement a system with a CPLD or an FPGA. Programmable logic devices are generally built using the same technology as flash memories and other electrically programmable, read-only-memories. (The basis of all these devices is the floating gate MOSFET that I discuss at least briefly in class.) Most of the deciding factors in the choice between the two approaches are set by the properties of the floating gate technology. The main qualitative features of the CPLD are:

- The logic pattern you program is non-volatile, that is, once programmed it does not have to be reprogrammed whether or not you turn off the power. (Charge stored on a floating gate is generally stable for decades.)
- The circuit topology allows the formation of Boolean products with very large numbers of inputs at no speed penalty.
- Propagation delays from pin-to-pin are relatively uniform and easy to predict.
- Some devices, including the one used on our board, can be programmed while in the circuit at very low cost, allowing optional design features and upgrades and further lowering the cost of initial product.
- One disadvantage is that compared to FPGAs, CPLDs offer only moderate speed and density. (For example, the one on our board has a 10ns nominal delay and 72 flip flops whereas the comparable generation FPGA of the same price had 4ns best-case delay and about 144 flip flops.) This is the result of more graceful scaling of FPGA architectures.

**The Board Itself:** Figure 10.1 shows a block diagram of what I have built. The actual board encompasses the things inside the dotted rectangle. There is a 26-pin ribbon cable connector on the board and the cables we provide have a 24-pin DIP plug on the user end. (This is the rectangle on the left in the figure outside the dotted box.) You can insert that plug into a protoboard just as you would a DIP integrated circuit. It affords what I hope is a quick and convenient way to make up to 18 logic I/O connections to the XC9572XL CPLD. The board also has a two-digit, seven-segment display with current limiting resistors wired to 8 other pins of the CPLD. The display has a common-cathode configuration with the two cathode terminals also going to the DIP plug. Three pins carry power and ground connections and one more of the 24 pin DIP connector pins is an I/O line that can be dedicated as a global reset line.

Every member of the Xilinx XC9500XL family of CPLDs has an industry-standard JTAG test port inside it. The device can be programmed *in situ* through that port with a suitable programming cable attached to a dedicated connector on the board.. (The JTAG idea and the standards to implement it are covered in https://en.wikipedia.org/wiki/JTAG. This industry standard provides a scheme for testing completed boards and circuits using a four-wire serial interface.)

**<u>WARNING!!</u> When you start to use a CPLD board for the first time, you *<u>MUST PROGRAM IT BEFORE YOU</u>* connect it to your protoboard. The CPLDs are easily damaged if an output pin is connected to a signal source, as it might be if a prior user had different pinouts programmed than you use. To power up the board for programming there are special power cables that connect to the ribbon cable plug on the board. Ask a TA!**

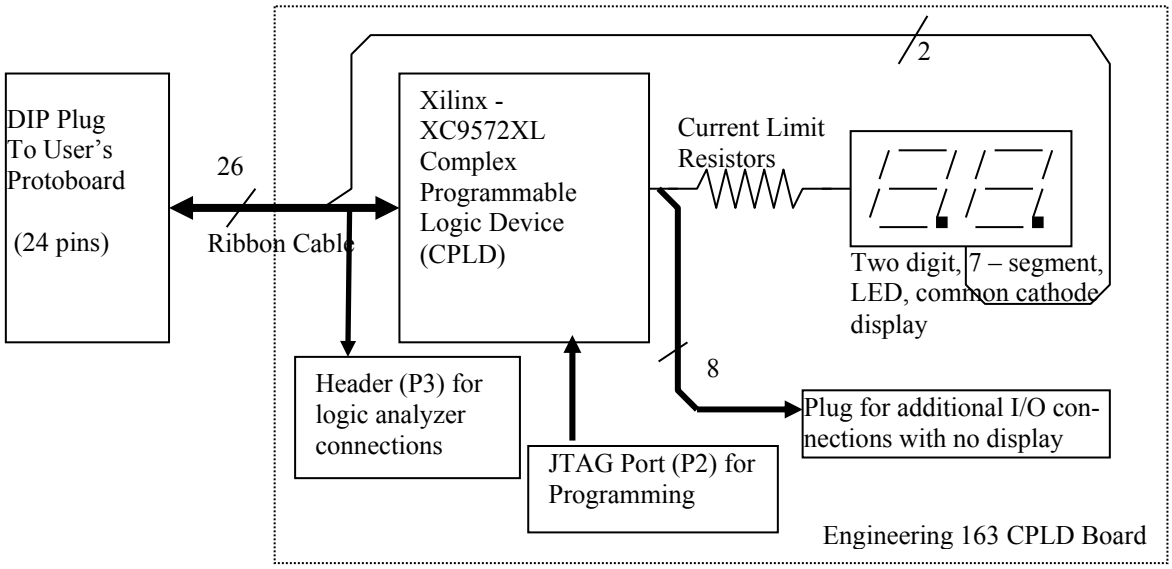
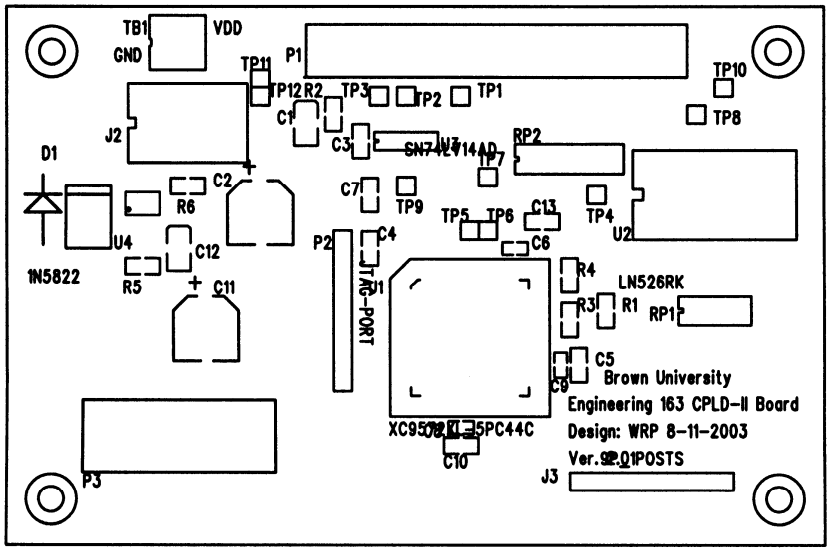



**Figure 10.1: Block Diagram of the Engineering 163 CPLD Board based on the Xilinx XC9572XL**

Figure 10.2 shows the top silkscreen pattern for the board along with a table identifying the various connectors. The user cable connector P1 is at the top and the JTAG connector P2 at the bottom. I added a connector compatible with the logic analyzers so that 16 channels of logic analyzer connections can be made with one cable to the first 16 signals in the user cable. The connections to the display also come to a connector so you can probe them during debug or, if the display is not needed, they can be used for additional I/O connections. When I was done with the basic functionality of the board, I noticed extra resources available in spare inverters and extra I/O terminals. I brought all these to test point patterns on the printed wiring so one could use them if needed. Some also go to another connector that allows jumper configuration changes. I do not anticipate that you will need to change these.

The full schematic for the board is shown in Figure 10.3. The XC9572XL requires a 3.3 volt power supply while you usually use a 5.0 volt supply. (The reason for the choice is cost and probable lifetime. Xilinx has raised the cost of their 5-volt parts to almost 3 times the cost of the 3.3 volt parts. Also, they are clearly signaling that they regard the 5-volt parts as obsolete and they may remove them from production soon.) Therefore, I included a regulator chip (U4) on the board with its supporting components to generate 3.3 volts from 5.0 volts. The XC9572XL will tolerate 5-volt input signals (it can be destroyed by +12 volts or – 5 volts, so be careful). However, its output is only 3.1 volts or so, so while it is compatible with TTL parts, it may give trouble with some CMOS parts. In addition to the components already mentioned, there is the usual complement of bypass capacitors. In using the board you have to supply power and ground through the ribbon cable.

There is a large diode (D1) connected from VDD to GND that I hope will partially protect the board if you connect power incorrectly. (Please don't do that anyway!) The XC9572XL has certain pins that are the only ones you can program to be global clocks, something you need to do for state machines and counters. Because ribbon cable connections are notoriously poor for fast clock connections and because clock edge problems are both obscure and difficult to fix, I have sent the signal for the GCK1 clock on pin 5 through an RC filter and two stages of Schmitt trigger inverters. This preserves the sense of the clock signal while smoothing it somewhat and exploiting the noise margin advantages of a Schmitt trigger device. (The Schmitt trigger sections are in the 74LVC14 hex inverter – U1.) This, of course, means that pin 5 of the CPLD cannot be an output. There are similar limitations on the placement of three-state output enables and of a global reset line. I have made sure that pins for three-state enable and for reset are available on the 24-pin DIP plug.




| Ref. | Function |
|---|---|
| P1 | Ribbon cable to user |
| P2 | JTAG program cable |
| P3 | Logic Analyzer plug |
| J2 | Jumpers (leave alone) |
| J3 | Optional I/O points |

**Figure 10.2: Silkscreen Pattern and Connector Functions for the CPLD Board**

**HINTS:** Xilinx recommends against doing what I told you not to do too, namely leaving unused inputs unconnected. It is good practice that uncommitted pins should be tied HIGH or LOW with a resistor. (You can tie a bunch of such pins together and then through a resistor to ground or VDD. This prevents trouble in case one gets accidentally programmed as an output due to a typo.) Also there is a set of 4 pins tied to test points [TP4, TP6, TP7, and TP8 on the schematic] to which I have added individual pull-up resistors. I recommend using those pins for outputs you don't need to connect elsewhere but which you might want to probe with the scope or logic analyzer during debug. FSM state bits that are not system outputs are an example of such use. There is also a 20-pin header that mates with a plug on a logic analyzer cable. When you want to do this, get the appropriate adapter plug from a TA. (I try to keep one such connector in place on each analyzer already.) This can connect 16 analyzer inputs to 16 of the CPLD inputs as shown in the pin assignment Table 10.1.

**Wiring:** The 44-pin package of the XC9572XL, the ribbon cable, and the users' DIP plug all use different numbering conventions. (We have included this adapter cable in every kit. We recommend that you minimize pulling the DIP plug in and out of your board, as the pins are easy to break. Leave it plugged in between labs.) As a result, figuring out how to make connections on the protoboard is non-trivial. The pin numbers of the DIP plug follow the same conventions as the through-hole integrated circuits in your kit. The ribbon cable, the connector for which is the only part of the protoboard connections to the XC9572XL actually shown on the schematic, uses still another convention. To simplify your task, Table 10.1 below shows how those numbers line up with the pins of the ribbon cable and the pins of the XC9572XL. For wiring your protoboards from the DIP plug, you may ignore the ribbon cable column. There is also a header, P3, on the board that you can use to plug into a logic analyzer and the table shows the logic analyzer channel associated with each CPLD/protoboard net. The table also points out those pins with special functionality, such as possible global clock or output enables in the CPLD. Complete pinout details are given in the data sheets at the end of this manual.

The connections to the display are shown on the schematic. Notice that the Xilinx pins 33, 29,28, 27, 26, 25, and 24 each connect to two segments of the display, one from the left and one from the right. This makes multiplexing the display trivial by if you drive the two display cathode connections to ground alternately. When most of the display segments are lit, the cathode load current exceeds the maximum pin current of the XC9572XL, so we recommend using a 7406 for driving the cathodes. See lab 3 for a further discussion.

**The XC9572XL CPLD:** A partial data sheet for the XC9572XL itself is in this manual and the full set of data from Xilinx on this family of parts is available both on their web site and on the Engineering 163 site. The parts are built using floating gate N-channel MOSFETs to implement non-volatile programmable logic in the form of AND-OR trees. I will talk about the transistors and the logic trees in class. This part can form up to 72 output signals that can be either registered or not. (However, the package only allows 34 of these to connect to physical package pins. The other nodes are "buried" but may be used for hidden logic variables.) Each output can have 5 Boolean product terms with up to 36 variables in a term. More terms are available if one forms fewer outputs. There is a large switch matrix before the AND-OR trees that gives very flexible feedback logic for state machines. The architecture implies some complicated rules for how logic is formed, but much of the burden of those details is taken care of in the software.

**Programming the Xilinx CPLDs:** As a practical matter, generating instructions for modifying the floating gate transistors to realize your logic can only be done through a suitable CAD tool. There is too much data that has to be put into proprietary formats to do the job by hand. Thus what you see when you go to use these devices is some description determined by the software rather than the CPLD's own architecture. With the Xilinx software, there are two possible forms for specifying the desired logic:

1. VHDL or Verilog HDL's
2. Schematic entry through Xilinx's own tool or block diagram entry through a third-party vendor such as Aldec.

Schematic entry does not make a lot of sense for small circuits that are heavy on the large Boolean expressions that are common in CPLDs and are characteristic of labs 3 - 7. You will use Verilog as

the entry language as that is the most commonly used language in the United States. You can do the early labs with just single Verilog modules, single files with a single logical block. See Lab 5 for an example Verilog file, an example of assigning pin numbers, and some discussion of what tools to use to write these files.

| Table 10.1: Pin Assignments Between XC9572 and User Connections | | | | |
|---|---|---|---|---|
| DIP Plug (Protoboard) | Ribbon Cable | Pin on XC9572 | Logic Anal. Channel | Function |
| 1 | 2 | 5 | 4 | Input to clock buffer and global clock pin 5 of XC9572XL. |
| 2 | 4 | 1 | 0 | |
| 3 | 6 | 2 | 1 | |
| 4 | 8 | 3 | 2 | |
| 5 | 10 | 4 | 3 | |
| 6 | 12 | 6 | 5 | |
| 7 | 14 | 7 | 6 | |
| 8 | 16 | 8 | 7 | |
| 9 | 18 | 9 | 8 | |
| 10 | 20 | | | Ground |
| 11 | 22 | | | Display Cathode |
| 12 | 24 | | | Ground |
| 13 | 23 | | | Display Cathode |
| 14 | 21 | 39 | | I/O or global reset |
| 15 | 19 | 42 | | I/O or global tristate 1 |
| 16 | 17 | 22 | | |
| 17 | 15 | 20 | | |
| 18 | 13 | 19 | 14 | |
| 19 | 11 | 18 | 13 | |
| 20 | 9 | 12 | 10 | |
| 21 | 7 | 13 | 11 | |
| 22 | 5 | 14 | 12 | |
| 23 | 3 | 11 | 9 | |
| 24 | 1 | | | VDD |
| | 26 | | | Ground |
| | 25 | 40 | 15 | I/O or global tristate 2 |

**Downloading Instructions:**

To program your circuit, find a machine in Room 196 that has a Xilinx JTAG downloading cable on it. There are more boards in the lab than there are JTAG cables, but once loaded your part will retain its contents even if you disconnect it from power to move it to another machine or to put it aside. Each PC with a JTAG cable will also have a ribbon cable connected to an adjacent power supply for **powering your CPLD board during initial programming. This is important because it prevents destruction of CPLDs by mismatches between your pin allocations and those used by the last person who used the board.**

1) The JTAG cable connects to the CPLD boards by a short flat ribbon cable that is keyed to prevent you from putting it on the wrong way. There are two styles of such cables depending on the particular CPLD board. All red programmer cables support both styles of JTAG connector through an adapter box. Hook up the board to the power supply with **JUST the dummy power-only cable and to the JTAG cable**. Do not hook the board to your breadboard until after it is programmed. Ask a TA if you have any doubts about how to make connections. With these two connections, you can turn on power and check that the LED on the JTAG cable turns from yellow to green.
2) From the start menu choose /Start/ All Programs/Electrical/Xilinx Design Tools/ISE Design System ##/ISE Design Tools/64-bit Tools/iImpact to open the downloading tool. (The symbol "##" is the current version number of the Xilinx software on our machines.) When iImpact opens, it will say there is no iImpact Project file. Cancel this and it will ask if you want to create a Project. Cancel the second dialog too.
3) In the left pane of the window there will be an entry for "Boundary Scan". Double click on that and the right pane will turn white with a blue legend in the middle.
4) Select "Output" from the main menu bar and within that click on "Cable Auto Connect". All you should see is a brief flash on screen.
5) Right click on the text to the right and select "Initialize Chain". Cancel any request to specify a configuration file or project file. You should end up with an icon for your XC9572XL part.
6) Left click on the icon and choose "Configuration File". Browse to the jed file for your design and select it.
7) Right click on the icon again and select "Program" to do the actual download. In the download dialog, make sure to check both Erase and Verify before hitting OKAY.
8) If the *iImpact* program says "Programmed Successfully" then you are ready to test your system.
9) If this is your first download for a lab, TURN OFF THE POWER SUPPLY AND WAIT FOR THE GREEN LED ON THE XILINX CABLE TO TURN YELLOW AGAIN. Then connect the cable to your board in place of the initial power cable.

## Figure 10.3: Schematic Diagram of the Engineering 163 CPLD Board

This page left blank. In the printed manual it is replaced by a foldout copy of the schematic.

**Figure 10.3 (Continued): Schematic Diagram of the Engineering 163 CPLD Board**

# 11. Schematics and Timing Diagrams

## 11.1. General Documentation

### Documentation of Circuits: Schematics and Timing Diagrams

Engineers earn their pay by devising ways to make things and by supervising manufacturing. Commonly, electrical engineers design devices that may range in complexity from five dollar dining room light dimmers to multimillion dollar satellites and utility systems. Almost never do they actually make anything themselves for reasons ranging from economics (they get paid too much...) to reliability (technicians and robots are usually better craftsmen...). An obvious consequence of this division of labor is that engineers must both communicate the results of their efforts and understand such communications. Thanks to the work of St. Cyril, Gutenberg, Pordof, Xero, and others, most of this information is in written or pictorial form. An IEEE survey a few years ago found that the average electrical engineer spends about thirty percent of his or her time generating this documentation.

There are two general types of documentation providing different kinds of information, one set of data for users and managers and another set for manufacturing. We are primarily concerned with the latter. Manufacturing documentation is diverse and includes at least such things as:

1. Block diagrams
2. Schematic diagrams
3. HDL design and testbench files
4. Parts lists (Bill of Materials – BOM)
5. Net lists (lists of interconnection for printed circuit fabrication)
6. Programmable device design files and/or ASIC (Application Specific Integrated Circuits, *i.e.,* custom chips) specifications
7. PCB (printed circuit board) plots of interconnections, particularly silk-screen and PCB assembly drawings showing part locations and annotation
8. Layout and assembly drawings
9. Circuit descriptions
10. Simulation documentation
11. Subsystem timing diagrams
12. Test specifications for verification of functionality and reliability.
13. Reports of calculations that insure some requirements are met by design. This is not a requirement in this course but is certainly a common one in most environments.

Subsystem timing diagrams are the subject of Lab 9 and are discussed in more detail in section 11.2.

The schematic diagram shows all the interconnections in a system in forms that are easy to relate to their functions. It is the commonest and earliest product of the design process. Of all the forms of documentation mentioned above, the schematic is the most standardized and the least sub-

ject to idiosyncratic company customs or to other situation-specific rules. We are asking you to draw schematics of your labs to this more or less universal form.

Since the 1970's, professional practice has been to draw schematics with appropriate Computer-Aided-Design (CAD) programs, a process called schematic capture. The reason for this development is that a schematic file can be used both to simulate the circuit and to generate the engineering documentation for building it more or less automatically with less engineering effort and fewer mistakes. The process is non-trivial and requires adding design information at each additional step. However, it guarantees consistency of part types and their interconnections between the schematic, the simulation, and the implementation whether on a Multi-chip Module or on a printed circuit. Changes to a system can be made rapidly at low cost to fix mistakes or to add new features. The computer-drawn schematic will be the basic design expression for digital systems for some years to come. Eventually even higher, more abstract levels of system description will displace it, and it too will be a derived product.

Given this reality, we have decided that you must turn in one schematic using the Mentor Graphics Corporation *DxDesigner* software system. It is available on the workstations in the Instructional Computing Facility and in room 196. It is also available through the Engineering VPN network. There is a very brief set of instructions on the class website that tells you how to gain access to it.

Unfortunately, learning to use a complex system like *DxDesigner* takes some time. There will be a couple of help sessions covering this material. There is on-line documentation within the program, but trying out the example schematic in the hints section of this manual is probably the fastest way to learn the system. As an example to guide your drawings, you might look at the schematic for the CPLD-II board in section 10.

**Handing in schematics:** You can print your schematics on a laser printer in the instructional computing lab. Hand in printed copies of your schematics to the specifications given below through the box in Room 196, the same procedure as for labs 2 and 6.

## 11.2. Timing Diagrams

Lab 9 involves software simulation of a circuit's performance. Most simulations will be just for the circuitry in one or another programmable device. Xilinx software, which is used to turn the netlist into programming code for those chips, also produces a VHDL file that is the basis for simulation. [You have to supply netlist information by doing schematic capture or by using synthesis from an HDL before trying the simulation.] You will transfer that file to Aldec's Active HDL software for simulation. The printout from that program is a timing diagram. In our experience such a diagram is the best tool for designing small timing systems such as those in Labs 7, 8 and A. Usually one makes the diagram of the necessary timing signals before trying to design the clock circuitry and then checks that the final circuit will realize this specification.

Such diagrams are frequently used in system specifications, particularly interface specifications. Figure S-ii has an example of such a drawing. The drawing is part of a data specification from a Cypress Semiconductor VMEBus controller.

It is useful and usual to annotate timing diagrams with indications of causal relations. A curved line extending from an edge of one signal to an edge of another represents a causal link and the edges will be separated by the signal propagation delay time. If that time is short compared to the scale of the time axis of the drawing, the edges may appear synchronous, but that is not actually the case. Two transitions that appear synchronous and are not connected by a "cause and effect line" probably have a common cause and are indeed roughly synchronous. I have used this convention in the timing diagrams that accompany several of the labs in this manual.

The details of using the Aldec *Active HDL* software are covered in a separate handout.

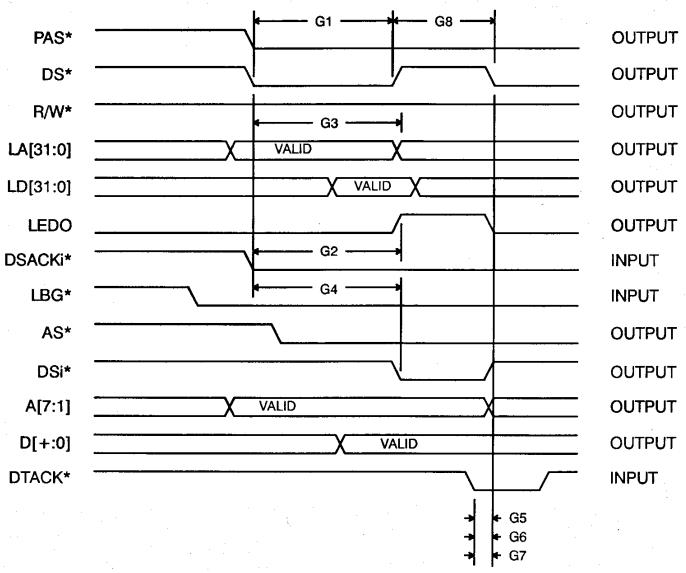



**Figure 11.2: Timing diagram example.**

## 11.3. Using the DxDesigner Schematic Capture Software

### Hints on the Use of the *DxDesigner* Schematic Capture System

The simplest schematic only requires placing preexisting symbols and wiring them up. These notes should help you do this using *DxDesigner*. These notes were written assuming that you will use the workstations in the Hewlett Computing Facility and in 196. There are two sets of Mentor Graphics documentation available. There is an illustrated user guide for the whole PADS/DxDesigner tool suite available from the Start menu as Start/All Programs/Electrical/Mentor Graphics SDD/PADS Evaluation Guide.pdf. There is an even more exhaustive guide available as on-line documentation. (On-line documentation is in the Help pull-down menu on the right side of the menu bar as is customary in Windows-based software.) Since the software is also available in room 196, you can make changes while working on the lab. This should be useful in the CPLD or

FPGA-based labs if you need to make minor updates. I have set up an ENGN1630 parts library, and you will want to take most of your symbols from this.

The designers of *DxDesigner* have used certain common engineering terms in slightly specialized ways. Understanding this vocabulary usage is important to minimize frustration. **Components,** for example, include not just integrated circuits, resistors, etc. Even the drawing frame that determines the size of the page and provides a place for descriptive text is a component. (Following usual drafting practices *DxDesigner* libraries have sheets labeled in standard sizes A, B, C, D.... Often these are called something like "csheet_bu.1." You will use C-size sheets.) Connections to power or ground use symbols found in the "Mechanical" component library even though these are symbols of connections rather than actual parts. (The most common symbol libraries are stored in the P:\Programs\Mentor Graphics\PCBLibraries directory should you want to copy a symbol file.) Symbol libraries may offer several variants on a symbol with different drafting practices for each; the variations are marked by different file extensions, for example "xxxx.1" or "xxxx.2" will be different forms of the same part "xxxx". **A Net** is a wire interconnecting two or more points. It extends however far one specifies a common connection. It is drawn with the command "Add/Net" from the pull-down menu. To extend it from one sheet to another, the two segments must have a common "**Label**". Labels are attached to nets by selecting the net with the left mouse button, right-clicking on Properties, and changing the net Name in the properties pane at the right of the screen. Remember that labels are different from simple text, which only adds English annotation but does not affect connections. **Buses** appear as thick lines on the schematic and **must be labeled** so that the individual wires in the bus can be distinguished. Normally, the bus label is an alphabetic prefix followed by a bracketed indication of the number of lines in the bus. For example, an address bus with 16 lines might be labeled ADDR_BUS[15:00]. (Labels are attached to buses the same way as to single net wires.) Subsequently, individual wires attaching to the bus must have labels of the form ADDR_BUS01 for the second wire in the group. (The first is ADDR_BUS00.)

Every component, symbol, net, etc. has a set of non-graphical information that is recorded and tracked during the schematic drawing. This information includes such things as package type for printed circuit layout, logic information and electrical data for simulation, section or pin data for multi-part components (e.g., which of the 4 sections of a 7400 Quad NAND gate is to be used at a given location), power connections, etc. Collectively, this information is known as the **properties** of the object. Usually this information comes from the library. You only need to change it when the default information is incorrect or insufficient. (For example, resistors and capacitors need values added to them and need the vendor part number revised so the bill of materials will be correct.) The one exception to this rule that **everyone** has to take care of is the **Reference Designator**. Library parts come with reference designators that are not unique, and *DxDesigner* does not generate unique ones as components are placed. The default designators are usually a prefix followed by a question mark. The prefix indicates the class of part for simulation and BOM generation and **must not be** changed. The question mark does have to be changed to a number so that the components can be tracked uniquely. For example, all integrated circuits have the default designator "U?". You would change this to something like "U2", keeping the "U" as a prefix but numbering the ICs uniquely. The procedure to do so is in the tutorial example below.

*DxDesigner* runs on the machines in the instructional computing laboratory and on the machines in Room 196 as well. Everyone enrolled in the course has their own account and should customize its password once they begin to use the machines. Please do not share accounts or put

off getting your account going. The first time you wish to use the *DxDesigner* toolset, please follow the brief procedure to set up directories and initialization files that is posted on the class web site as a separate handout. Note that this procedure is executed only once annually.

**How DxDesigner is organized:**

*DxDesigner* has a huge number of commands available because there is a lot of potential detail in a schematic. Since different users have different fine motor skills, *DxDesigner* provides several ways to do the commoner tasks. To help you get a working understanding of the program quickly, we will explain the command entry process first, how you select and move objects, change screen views, and move between sheets and levels. Finally there is a tutorial guide to drawing a simple schematic.

**The Mouse:** Like most graphics programs with extensive user input, this program is mouse-driven for menu choices, parts placement, node selection, wire routing, etc. For most commands, the mouse behaves the same way. You use the **Left Button** to select menus, to hit soft buttons, and to select and move objects. If an object has properties, selecting it and right-clicking on Properties will open a pane on the right side in which you can change properties such as reference designators, component values, speed grades, etc. The properties list also provides for labeling and marking signals as negative true, *i.e.*, inverted. The **Right Button** opens a short menu of commonly used commands for whatever is selected. If nothing is selected at that time, this menu still offers a chance to move to any other sheets in the project.

**Command Entry**: The mouse and the keyboard are used together to enter commands in five different ways.

(1) The mouse can activate pull-down menus located at the top of the drawing window. They are labeled: File, Edit, View, Setup, Add, Format, Simulation, Tools, Window, and Help. In the command tables below, the first word of the menu choice column is the name on the pull-down menu bar.

(2) Certain keys can be used as single stroke commands. These are listed in the command tables in the "Single Key" column. If marked "<Ctrl>x", it means hold down the <Control> key then hit the "x" key simultaneously. You will often find that this is the most efficient way to enter frequently used commands.

(3) Many of the most used commands are also available as soft-buttons activated by single clicking the left mouse button on icons around the edge of the drawing. In the tables below, the icon buttons are labeled by position assuming the buttons are clustered around the working area. (This layout corresponds to how I have setup the screen if you followed the initialization procedure on the class website.) The buttons referred to as "Tx," where "x" is a number, are on the top left edge of the screen. The number specifies position counting from the left to right along that bar. "Lx" denotes a button on the left side counting top to bottom; and "Rx" a button on the top right counting top to bottom.

(4) Entering more complex commands, or frequent commands that do not have single keystroke or icon equivalents is done by hitting the <Space> bar to activate a command line at the upper left edge of the main window. (This line is called a "dockable" window because you can move it by clicking on its frame and pushing with the mouse. I myself like it where the designers put it.) Then type in a command. Again, the keywords for the commonest commands are included in tables II and III. This mode of entry is most useful for commands requiring that you type in strings, such as changing grid spacing or package slots.

(5) The right mouse button opens a menu that changes entries to anticipate the most the probable next operations.

**Selecting and moving objects:** If you want to change anything about something already on your drawing, whether its position, value, labels, properties, etc., you first have to select it. To do so, you have to be in "select mode," which is reached by pushing the top left softbutton. (It's marked by an up-arrow. Alternately, the single key "s" will do the same thing.) Then put the mouse cursor on the object and click the left button. A box will appear around the object or it will be highlighted. To move it, just continue to hold the button down and drag the object by moving the mouse. To select more objects at the same time, hold the <Control> key down while clicking on successive elements. Any command to cut, copy, or paste will affect all the objects selected at that time. To change values, labels, or other properties, select the object or label, right click on the object, and enter the new information in the Properties list on the right side of the screen.

**Changing Views:** There is usually more detail in a schematic than can be clearly displayed at once. To make features legible, one zooms in and out and changes the displayed area. One can zoom with the mouse scroll key or use the special function keys at the top of the keyboard to do these operations. Table I lists the particular function of each of these keys along with the alternate ways to do these commands.

Table I: Screen View Control – Zoom, Pan, and Refresh

| Key | Softbutton | Pull-Down Menu | Function |
|---|---|---|---|
| <Home> | R1 | View/Fit All | Return to the full of Home view of drawing |
| <Insert> | | View/Fit Selected | Move the drawing view so that the current cursor position or last selected component is centered on screen. |
| <F7> | R2 | View/ Zoom Out | Zoom out by one step |
| <F8> | R3 | View/Zoom In | Zoom in by one step |
| z | R4 | View/Zoom Area | Make a rectangular area selected by the Mouse the full screen view. Push left mouse Button and drag the mouse cursor over the Region to be the new view. |

**Multiple sheets and other levels:** While Engineering 163 drawings tend to be fairly small, most diagrams require more than on sheet or page to hold all their parts. In the Mentor Graphics system, each sheet is a separate schematic file and is numbered by a simple numerical extension, for example, "my_junk.3" would be the third page of a schematic for the "my_junk" project. Within a schematic, each symbol links to other files having physical data and simulation models. There is a group of commands that enable you to move around within this hierarchy and Table II summarizes them.

| Table II: Moving Between Levels | | | |
|---|---|---|---|
| Softbutton | Command Line | Right Mouse Button | Function |
| | psym | Edit Library Symbol | Open the symbol file for a selected object for Editing. You generally will not have to do this. |
| | Psch | Schematic | Push to the underlying schematic for a selected object. Often an underlying schematic tells how a device is to be simulated. |
| R6 | psheet <sheet #> | Next Page –or- Go to Page | Push (go to) the next sheet. (With R10 or the Go To Page command you can go to an arbitrary sheet.) |

**An Example Schematic:**

The following steps will guide you through a simple example. Try them to get a feeling for how the software works. We have included the resulting schematic as Figure 11.3.

(1) Open *DxDesigner* (Start/All Programs/Electrical/Mentor Graphics SDD/Design Entry/Dx Designer) and go to the File menu. Choose Open/Project/ and browse to the Engineering Classes.prj file in the wv\Engineering Classes directory. That should open the screen with all the tool panes showing.

(2) In the Navigator pane on the left side of the screen, expand the menu tree at Board. There will be a yellow symbol for a board named "example1". Double click that and a drawing page will open with a sheet frame already in place. If you wish, you can use this sheet for either a test drawing or your required schematic.

(3) If you don't want to use the example1 schematic as your basis, then from the File pull-down menu, select "New/Board" and a new "Board#" and Schematic will appear in the Board tree in the Navigator. Right click on each name in turn and rename them both to something appropriate for their use. You will now have a blank, screen for drawing and it should have a sheet frame already.

(4) If the sheet frame is missing, you will have to add it. Go to the "DxDataBook" pane at the lower left corner of the screen and look for the Symbol View box with two columns for Partition and Symbol. (If the DxDataBook is not open, open it with menu View/DxDataBook.) Scroll down in that window to the "Mechanical" library and expand the symbol list that

goes with it. From that symbol list, select the "csheet_bu_rev.1" with the left mouse button. Just to the right is a window with thumbnail sketch of the selected component. Place the cursor over the sketch and holding the left mouse button down, drag a copy of the component onto the schematic. When the sheet frame is centered on the working area, release the mouse button. Should you need to move the sheet after you have placed it, go to the lower left corner of the frame and select a box around that corner. Hit the 'm' key and press the left mouse button to drag and drop the symbol. Congratulations, you have just laid down a sheet frame.

(5) Position the cursor just above and to the left of the title block on the lower right corner. Hit the special function key <*F9*>; holding the left mouse button down, drag the cursor down and to the right until the title block is surrounded by a new rectangle. Release the button and now the title block will fill the screen.

(6) From the top menus select "Add Text." Place the mouse cursor next to the "Designer:" annotation, click the left mouse button, and enter your initials. Accepting the entry will mark the drawing as yours. To complete the rest of the title information move the mouse cursor to each entry, click the left button, and type the entry. [NOTE: filling in the title block is not optional. You must fill it in completely including a revision line at the top of the annotation block. This is the very minimal documentation control that I impose. Industrial requirements are far more stringent.]

(7) Return to the full screen view by hitting the special function key <*Home*>.

(8) Again go to the "DxDataBook" pane at the lower left corner of the screen and look for the Symbol View box with two columns for Partition and Symbol. (If the DxDataBook is not open, open it with menu View/DxDataBook or with the second softbutton from the top on the left side of the schematic frame.) Scroll down in that window to the "en163" library and expand the symbol list that goes with it. Select the "en163" library. Move the cursor to the "74ls00.1" selection, hold down the left button on the mouse over the thumbnail sketch of the selected component and drag the gate to the right of center on the drawing. Should you need to move the gate after you have placed it, select it (left mouse button) hit the 'm' key. Then press the left mouse button to drag and drop the symbol.

(9) Repeat the process selecting the "74ls169.1" chip. Place it along the same line but to the left by an inch or two from the NAND gate. Reposition the components as needed.

(10) Zoom in on the two integrated circuits by putting the mouse cursor above and to the left of the 74ls169. Hit <*F9*>, drag the mouse down and to the right past the NAND gate, and release the button. Next select the 74ls169 by entering "select mode" and clicking on the 74ls169 symbol. Keeping the cursor on the symbol, press left mouse button and hold it down. Drag the symbol with the mouse until the QD pin is on the same horizontal line as the top input pin of the NAND gate.

(11) From the menus, select the command "Add/Net". Put the cursor on the QD pin of the 74ls569, hold down the left mouse button, and drag the cursor over to the upper input pin of the NAND gate. A line will appear representing a wire, part of a net. When you reach the

pin, release the button. Repeat the mouse action to connect the output of the gate to the *LOAD* pin of the 74ls169. Start again with the cursor on the lower input of the NAND gate a wire left and down. Releasing the button will make a temporary break in the net for later connection.

(12) Now we will fix the reference designators of the two parts. Enter "select mode" and select the NAND with the mouse. (The shortcut to "select mode" is the "s" key or use the top left softbutton around the schematic.) If necessary open the device properties pane by right click on the part. In the properties pane, left click on the value of REFDES and change the value from U? to U1. Repeat the process for the 74ls169, naming it U2. [Note: reference designators are a mandatory part of a drawing, not optional. However, you have some freedom in assigning them as long as you keep the prefix for the part and you make the designator unique to each component package. Parts that have multiple sections, such as the 74ls00 must have the same reference designator for all four sections but different pin sets or "slots" for each.

(13) Next we need to add another two input NAND gate. There are two ways to do this, namely returning to DxDataBook to retrieve another section or copying the existing gate. For variety, let us try the latter. Enter "Select Mode" again (icon L1 or keystroke "s"), and click on the first 74ls00 section already on the drawing. From the top pull-down menu, choose "Edit Copy" then "Edit Paste." (The usual keystrokes Crtl-C and Crtl-V also work.) Click the left mouse button on the schematic to make the part appear and drag it until it is positioned somewhat above the original part. Releasing the button locks it into place. By default, its pinout will be the same as the first gate, and so it must be changed.

(14) Now suppose we wish to make this gate the one using pins 8, 9, and 10. Select the new gate with the mouse select button. Use the command line entry "<Space bar>slot <space>3<Enter>". Finally, change the reference designator on this part to U1 to show it is part of the same package as the first gate. (An alternate method to copy a part is to select it, hold down the <Ctrl> button, and drag a copy where you want it. This method will automatically change the slot number of the new device but it may still not be the slot you want.)

(15) Wire the inputs of this gate to QC and QD of the 74ls169 with the "Add Net" command. It may be easier to attach QC to pin 9 of U1 first. You may also have to release and repress the mouse button once as you drag the wire to make the wire bend where you want it to to.

(16) Now we will label one of our lines. Enter "Select Mode" (keystroke "s"), place the mouse cursor on the top segment of the net connecting the pin 3 output of U1 to the 74ls169 synchronous load pin, and click once to select that wire. In the properties pane on the right side of the screen, change the Name of the net to "CTR_RELOAD" and check that the box for visibility is checked. This signal is actually negative true, that is, the counter resets when this signal is LOW. To indicate this on the label, change the "Name Inverted" property from false to true.

(17) Finally let us add a bus to the schematic and connect a wire or two to it. From the menus, select the command for "Add Bus". Put the cursor an inch above the 74ls169, hold down

the left mouse button and drag the cursor to the right for about three inches. Release the button to end the segment. Label the new bus with the same procedure as for the CTR_RELOAD line, except call it "CTR_DATBUS[7:0]", implying an eight wire bus. If the label is obscured by the bus itself, you can reposition it by selecting just the label itself and dragging it with the mouse. Using the usual "Add Net" command, add a wire from QA of U2 to the bus. Label this wire "CTR_DATBUS0", meaning it is the lowest order bit line of the 8 bit counter bus.

(18) The net label may be awkwardly oriented. To rotate it, select just the label itself (NOT the net). From the menus, select "Edit Rotate". With the cursor on the lower left corner of the label press the execute button. You can then reposition it by dragging with the mouse.

(19) Connect nets from QB, QC, and QD to the CTR_DATBUS bus. It often happens that one has a group of lines like this attached to a bus, and they have to be labeled sequentially. Select the line extending from the bus to QB of the 74ls169. Label it the same way as for CTR_DATBUS0 except use the label string CTR_DATBUS[1:3]. The result will be a label for CTR_DATBUS1. Now select the line attached to QC, hit the right mouse button, and select "Next Label". This will label that line CTR_DATBUS2. Repeat for QD to make it CTR_DATBUS3.

(20) Now add, position, and wire the components that will be U3, R1, and VDD as shown on the drawing. Change their reference designators and label the dangling output wire. The last thing to do is to set the resistor value. Select the resistor and edit the "Value" property line in the properties pane to be 1K Repeat the process to update the "Vendor" property of the resistor with the Digikey stock number for this value.

(21) It is not necessary to store your work as that is done automatically and fully as you go along. Please consult the Mentor on-line documentation for how to rollback a schematic to an earlier version.

(22) To run a check for proper interconnections, use the menu command "Tool/Verify" or the 10th button on the top toolbar, the one with a blue checkmark. Some data about the schematic will be shown either at the bottom of the screen or in a new window. It should say that there are no errors or warnings. If there are any, then the window information will include a list of warnings and errors.

(23) Now you may wish to print what you have drawn. Pull down the File menu and choose "Print Setup". Make sure that the mode is set for Landscape printing and NOT Portrait. Then again pull down the File menu and select "Print". The result will appear on one of the laser printers in the computing lab.

(24) Some people prefer to work with a black background for their schematic. To change the color scheme, use the menu command Setup/Display/Objects and hit the Load Scheme softbutton. Browse to C:|Mentor Graphics\9.4 PADS\SDD_HOME\Standard. There will be several color scheme files and you might choose "Classic" or "Expedition". Accept the choice and the colors change.

After you have used the menu commands a few times, you will realize that pulling down a menu and making selections is a cumbersome method of changing between the commonest operating modes. Depending on the command, you will want to use the single keystroke command, the icon soft button, or the command line word entry. The correspondence of keystroke, icon, and word commands to menu selections for the commonest choices is given in Table III. More comprehensive but less accessible lists are given in the Help menu.

Table III: Correspondence of Typed Commands with Menu Choices

| Menu Path | Soft Button | Command Entry | Single Key | Function |
|---|---|---|---|---|
| Edit Select | L1 | | s | Enter Select Mode in which the left mouse button selects objects for editing – moving changing properties, etc. |
| View DxDataBook | L2 | | | Open DxDatabook to get symbols |
| Add Net | L3 | net | n | Connect a wire between two or more points. |
| Add Bus | L4 | bus | b | Add a selection of a bus to the schematic. |
| | | label | | Attach a label to a net or bus. Alternate is to edit the Name entry in the net properties pane. |
| Add Box | L7 | box | | Draw a rectangular box; mouse sets corner positions. |
| Add Circle | L8 | circle | | Draw a circle; mouse sets center, radius. |
| Add Line | L9 | line | | Draw a straight line; NOT the same as adding a wire. (Use Add Net for wiring.) |
| Add Text | L10 | text | | Put some text on the drawing; Not the same as a label. |
| Edit Copy | T4 | copy | <Ctrl>c | Make a duplicate of a selected object already on the screen. |
| Edit Cut | T3 | bcut | <Ctrl>x | Cut selected section and put in buffer. |
| Edit Paste | T5 | bpaste | <Ctrl>v | Paste buffer onto screen at cursor. |
| Edit Delete | L12 | del | <Del> or d | Delete selected objects. |
| Edit Reflect | L14/15 | reflect | | Mirror reflect a selected object about a line set by the mouse cursor. Softbutton (icon) method separates vertical and horizontal reflections onto two buttons. |
| Edit Rotate | L13 | rotate | | Turn an object about the cursor position by 90 degrees. The rotate command entry also requires a mouse click to activate. |

| Table III Continued: Correspondence of Typed Commands with Menu Choices | | | | |
|---|---|---|---|---|
| Menu Path | Soft Button | Command Entry | Single Key | Function |
| | | slot | | Change the section of a multipart component used for a selected location. See discussion in text. |
| File Print | T2 | | <Ctrl>p | Print to the laser printer. (Be sure to have the print view set to landscape and NOT portrait before printing.) |
| File Exit | | | | End the DxDesigner session. |

Note: <Ctrl>x means hold the Control key and simultaneously hit the next character, “x”.

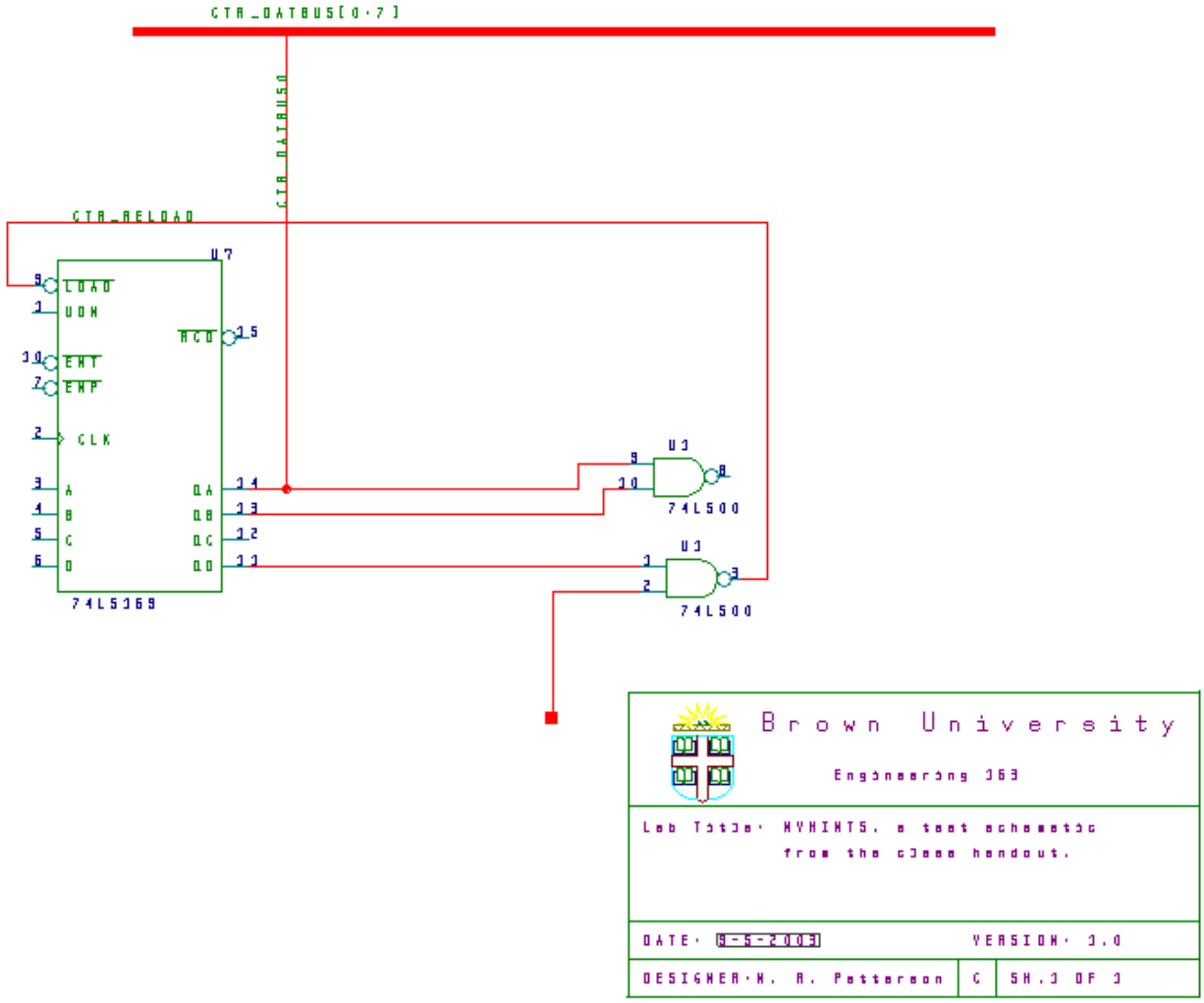



**Figure 11.3: Schematic of the tutorial example.**

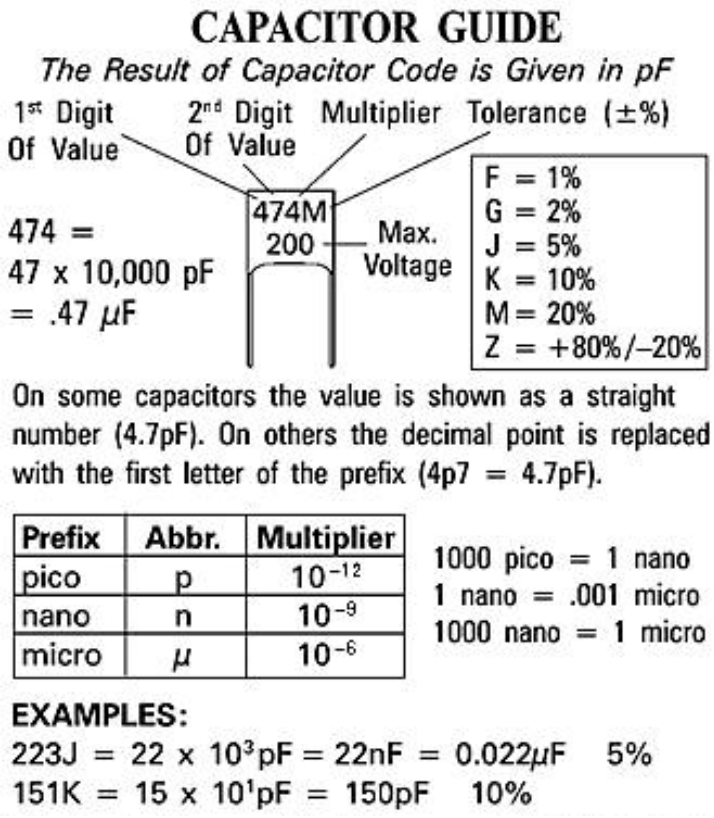
## 12. Component Data Index

# Manufacturers' Device Data Sheets

Here is the first page of each data sheet for the active devices in your Bag-of-Chips as well as some material on other components. First pages give the functionality and pinout of the devices. For several labs you will need more data on the devices. I have posted a zip file with the full data sheets that you can download from the class web site. The descriptions are ordered by type of device (e.g. TTL, CMOS, etc) and within each group the devices are ordered by type number. The groupings are as follows:

1. 74' series TTL, LVC, and ACT logic.
2. Xilinx CPLD and FPGA pinouts and overview.
3. Analog integrated circuits and switches (DG202BDJ-E3, LM311, TLV272IP, NE555).
4. Selected passive components.

Capacitor Guide:



As an added convenience, here is a table of logic analyzer cable connections that you can use to monitor the data moving from the PC to the FPGA and back again. Monitoring these connections is very useful in diagnosing the dual port memory interface to the outside world.

| Signal Function | Signal Name | Header # | Logic Analyzer Bits |
|---|---|---|---|
| Data bus | DatBus[] | P2 | [15:00] |
| Transfer direction: 0 is read to PC, 1 is write to FPGA | BDIR | P4 | SIG2 |
| HIGH runs I/O | CTRL_MODE | P4 | SIG3 |
| Bit from test port to force processor reboot when HIGH | REBOOT | P3 | SIG3 |
| Returned HIGH from FPGA when processor executes a HALT instruction | HALTED | P3 | SIG4 |
| From test port to clock data per timing diagram above | DAT_CLK | P3 | SIG2 |
| From FPGA to clock data per timing diagram above | ACK_CLK | P3 | SIG5 |

- The macro that generates the testbench in "labD_assembler.xlsm" creates a Verilog file called "up_architecture_tb.v" that you add to your Xilinx design with "simulation" checked as the design function. (The macro algorithm in the spreadsheet tries to save the file in your source directory to make it easy to change the simulation. You may have to modify its guess as to where that Xilinx directory is in your directory tree. Change cell "C1" as needed.) When you run the iSIM simulator, it will show you what your processor does. (See separate handout for using iSIM.)