

# Welcome

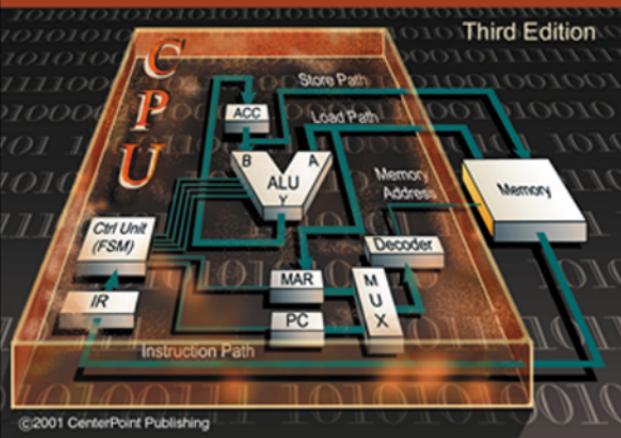
Digital Design for the Laboratory

Daniel J. Tylavsky, Ph.D.

## Digital Design for the Laboratory

Hardware & Simulation (using LogicWorks™ 4) Ver. 3

ABET 2000  
Compliant



©2001 CenterPoint Publishing

## Preface

## Report Writing Guideline

## Hardware Labs

## Simulation Labs

# Simulation

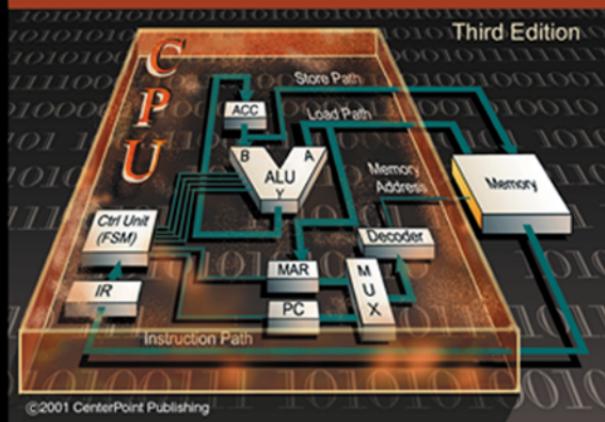
Digital Design for the Laboratory

Daniel J. Tylavsky, Ph.D.

## Digital Design for the Laboratory

Hardware & Simulation (using LogicWorks™ 4) Ver. 3

ABET 2000  
Compliant



## LogicWorks Tutorial

### Simulation Labs

### Appendices

Back

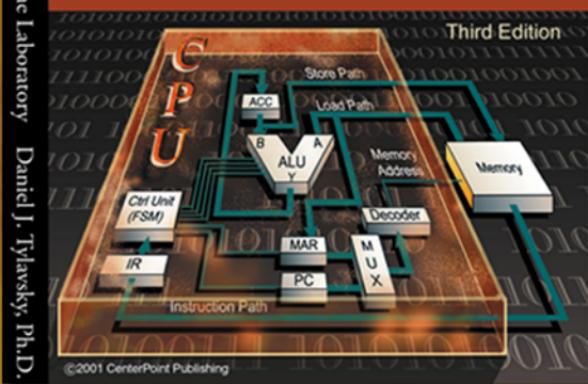
# Simulation Labs

Digital Design for the Laboratory

## Digital Design for the Laboratory

Hardware & Simulation (using LogicWorks™ 4) Ver. 3

ABET 2000  
Compliant



Hardware  
Labs

Back

Simulation Lab 1

Simulation Lab 2

Simulation Lab 3

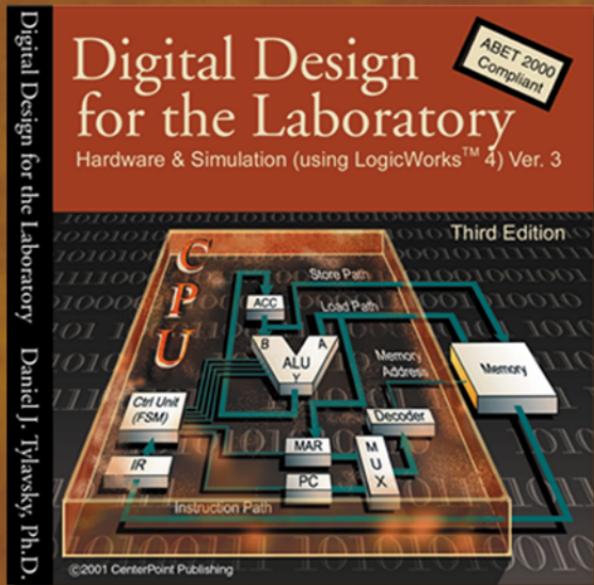
Simulation Lab 4

Simulation Lab 5

Simulation Lab 6

Simulation Lab 7

# Hardware Labs



Simulation  
Labs

Back

Hardware Lab 0

Hardware Lab 1

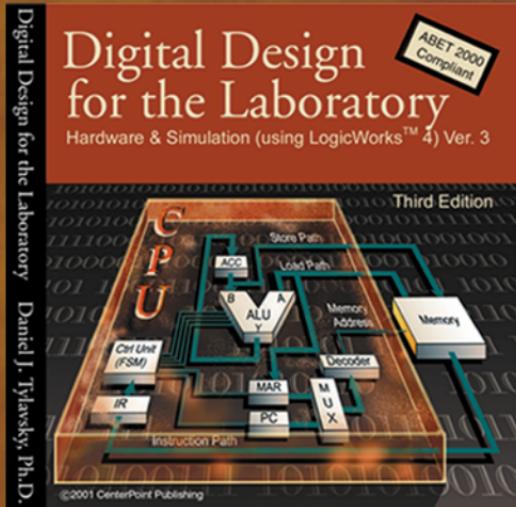
Hardware Lab 2

Hardware Lab 3

Hardware Lab 4

Hardware Lab 5

# Appendices



Back

## Appendix A: LogicWorks Symbols

## Appendix B: PROM-Based Synchronized Mealy Machines

## Appendix C: Controller Design Using Classical Design Techniques

## Appendix D: Creating a PROM Device in LogicWorks 4 for Windows

# Digital Design for the Laboratory

Hardware & Simulation (using LogicWorks™ 4)

Version 3

Third Edition

Daniel J. Tylavsky, Ph.D.

CenterPoint Publishing  
Tempe, Arizona 85284



**CenterPoint Publishing**

2301 E. Balboa Drive  
Tempe, AZ 85282

---

Copyright © 2001 by CenterPoint Publishing

**All rights reserved.** No part of this book may be reproduced, stored in a retrieval system, or transcribed in any form or by any means – electronic, mechanical, photocopying, recording, or otherwise – without the prior written permission of CenterPoint Publishing.

*For more information, contact:*

**CenterPoint Publishing**  
**2301 E. Balboa Drive**  
**Tempe, AZ 85282**

Cover Designer: Cheryl F. Johnson, P.E.

Printed and produced in the United States of America

ISBN: 0-9662944-3-2

Library of Congress Cataloging-in-Publication Data

Tylavsky, Daniel J.

Digital Design for the Laboratory: Hardware & Simulation (using LogicWorks™ 4) version 3, / Daniel J. Tylavsky – 3<sup>rd</sup> ed.

ISBN 0-9662944-3-2 (Multimedia)

1. Logic circuits 2. Logic design. 3. Logic circuits – Design and construction

I. Title. II. Tylavsky, Daniel J.

# PREFACE

---

## Rationale

It is traditional in most universities to have an introductory digital design course with a hardware laboratory component. In this manual, I selected traditional digital-logic hardware experiments that emphasize the types of experiences we all wish our students to get from their laboratory experience: digital logic design, construction, testing, and debugging.

There is, however, a limit to what students can build in the hardware laboratory. The digital circuits that students can reasonably build with hardware are limited by the time constraints imposed by the course format and the equipment constraints imposed by the laboratory facilities. To supplement hardware experiments, digital-logic simulators, such as LogicWorks™, with schematic capture capability, can be used to pedagogical advantage. By carefully selecting and designing complementary simulation laboratory exercises, I have tried to create exercises that will allow students to experiment with circuits that reinforce lecture concepts but are too complex to build with hardware. One advantage of using digital logic simulators, such as LogicWorks™, is that circuit simulations can be built upon, week after week, without tying-up laboratory resources and without the problems that occur as students carry their hardware circuits back and forth between home and the laboratory.

One challenge of using a digital logic simulator is that, with the appearance of each new version, the lab manual used with the simulator must be rewritten to accommodate the new interface, menus and submenus. My goal is to produce an updated version of this manual with the release of each new version of LogicWorks™.

## Audience

The hardware and simulation laboratory experiments contained in this manual are designed for an introductory course in digital design. Each experiment consists of a core set of tasks to challenge freshman and sophomore students, with additional tasks and experiments to challenge more mature students. The hardware and simulation experiments are integrated so that the results and learning objectives of each are mutually reinforcing. The first items the students reads at the beginning of each laboratory experiment are the list of prerequisite skills needed before beginning the laboratory experiment, designed learning objectives, and learning outcomes. These lists help the students understand what knowledge they need before beginning the experiment, and what they will learn by successfully completing the tasks comprising the experiment.

## Hardware Laboratory Experiments

The hardware laboratory exercises are designed to complement the simulation exercises contained in this manual. The hardware laboratory experiments start students with basic experiments that emphasize common laboratory measuring equipment and debugging techniques then progress to more sophisticated experiments that allow the students to demonstrate the design skills they have acquired in the lecture portion of the class.

The emphasis of the hardware lab experiments roughly progresses from electrical measurements, electrical characteristics, and debugging, to the use of classical design procedures as shown in Table 0-1. The order of tasks within each experiment is chosen to build understanding in a coherent way. The early laboratory experiments are written to be largely self-contained. Later experiments rely more heavily on material from the lecture portion of the course. Later hardware labs also include tasks that require simulations of the hardware circuits students construct in the hardware laboratory. The objective of this mix, is to allow students to discover, the role that simulation plays in the prototyping of complex engineering systems.

**Table 0-1. Main Performance Outcomes and Pedagogical Emphases in the Hardware Laboratory Experiments.**

Lab Title	Outcome: At the completion of the laboratory exercise the student will be able to:	Emphases
<a href="#">Hardware Lab 0: Using a Prototype Board, Logic Probe &amp; Voltmeter</a>	Use a breadboard, logic probe and voltmeter.	Electrical measurement.
<a href="#">Hardware Lab 1: Debugging a Half and Full Adder</a>	Build a half and full adder and debug combinational logic.	Electrical measurement, Debugging.
<a href="#">Hardware Lab 2: TTL Characteristics, Three-State Buffers, Open-Collector Buffers</a>	Use three-state and open-collector buffers.	Electrical characteristics, Debugging.
<a href="#">Hardware Lab 3: Latches, Flip-Flops, Registers and Counters</a>	Build and debug latches, counters, and registers.	Sequential circuit performance, Debugging.
<a href="#">Hardware Lab 4: Vending Machine Controller Design</a>	Build a Vending Machine Controller using MSI circuits. Use cut-and-try design principles.	Use of MSI circuits, Principles of design, Role of circuit simulation in design.
<a href="#">Hardware Lab 5: Capstone Design Project</a>	Design, simulate and build an arbitrarily complex synchronous sequential machine.	Principles of design, Classical design procedures, Circuit simulation.

There are more tasks included in many experiments than students can complete in the laboratory time allotted for most digital design courses. We have found that by selecting a subset of cohesive tasks that vary from semester to semester we allow students to rely on their own understanding of the material rather than that of students from previous semesters.

The order of the experiments is chosen to be in synchronism with the order of topics covered by most textbooks on introductory digital logic design. One challenge I encountered when designing these

experiments was to estimate the pace of most college courses, so that the experiments contained in this manual could be performed in consecutive weeks. Any feedback you care to give on how successfully this was achieved would be appreciated. You can reach me via email at: [tylavsky@asu.edu](mailto:tylavsky@asu.edu).

## Simulation Laboratory Experiments

The simulation laboratory exercises are designed to complement the hardware exercises contained in this manual. Upon the successful completion of the first five simulation exercises, students will be able to:

- Build and execute basic circuit simulations.
- Build upon these circuits until they have constructed the simulation of an elementary microprocessor.
- Create their own instruction set.
- Use their simulation to execute a program stored in simulated ROM.

The initial experiments emphasize using LogicWorks™ to build and modularize elementary circuits. Early laboratory experiments also emphasize the different debugging and building techniques needed when using schematic-capture digital-hardware simulators. (See Table 0-2.) The modularized circuits built in simulation labs 1 and 2 are combined in the third simulation exercise to construct an ALU. The remaining experiments build onto this ALU to construct an elementary microprocessor. We find that students derive a great sense of accomplishment from having built a simulation of a microprocessor. We also find that this experience helps when they move on to the study of microprocessor architectures and assembly language programming.

There were many design decisions made that resulted in the microprocessor architecture and circuit simulations you will find in this manual. Trade-offs between simplicity and authenticity were made at many stages in the design. A relatively simple architecture was chosen so that students' efforts could be concentrated on the fundamental principals behind the workings of a microprocessor, rather than on the intricacies required for a high performance design. If I were to redesign this microprocessor four times, there is no doubt that I would arrive at four different designs. I have second-guessed many design decisions, as no doubt will you; however, my experience with these lab experiments (with over 4,000 students who have completed these experiments), is that they achieve what they were designed for; at the completion of the simulation labs, students have a fundamental understanding of the operation of a microprocessor (and brag to their friends about how they “built a baby Pentium™”). It is my hope that you will have the same response from your students when using this laboratory manual.

**Table 0-2. Main Performance Outcomes and Pedagogical Emphases in the Simulation Laboratory Experiments.**

Lab Title	Outcome: At the completion of the laboratory exercise the student will be able to:	Emphases
<a href="#"><u>Simulator Tutorial: Using LogicWorks™ 4</u></a>	Use LogicWorks™.	Using schematic capture features of a digital logic simulator.
<a href="#"><u>Simulation Lab 1: Half Adder, Increment &amp; Two's Complement Circuit</u></a>	Create circuits to perform half adder, increment, and two's complement functions.	Building circuit simulations, Debugging, Creating modularized subcircuits.
<a href="#"><u>Simulation Lab 2: 4-Bit Full Adder, Multiplexer &amp; Decoder</u></a>	Create circuits to perform full adder, multiplexer, and decoder functions.	Circuit design, Debugging, Creating modularized subcircuits.
<a href="#"><u>Simulation Lab 3: Arithmetic and Logic Unit</u></a>	Create complex circuits using subcircuit blocks. Describe one design for an ALU.	Hierarchical circuit design concepts, ALU concepts.
<a href="#"><u>Simulation Lab 4: The Brainless Microprocessor</u></a>	Act as a controller for a microprocessor comprised of ROM, RAM, ALU, program counter, memory address register, and accumulator.	Hierarchical circuit design concepts, Microprocessor control, Writing to and reading from memory.
<a href="#"><u>Simulation Lab 5: The Complete Microprocessor</u></a>	Build, and debug a controller for a microprocessor. Develop and implement an instruction set. Program the microprocessor and execute the program.	ROM based synchronous machine design, Breakdown of instruction into microinstructions, Microprocessor control, Top-down design.
<a href="#"><u>Simulation Lab 6: The Jump Instruction</u></a>	Describe the architectural elements and functional blocks needed to implement a jump procedure.	Top-down design, Address-circuit design, ROM based synchronous machine design, Breakdown of instruction into microinstructions.
<a href="#"><u>Simulation Lab 7: The Status Register and the Conditional Jump</u></a>	Describe the architectural elements and functional blocks needed to implement a conditional jump procedure. Describe the function of a status register. Build a status register.	Microprocessor addressing architecture, Conditional microprocessor control, Role of a status register and status logic, Top-down design.

## Selection of LogicWorks™

I chose to use LogicWorks™, rather than another simulator, because the simulation-laboratory exercises, out of which this manual grew, were originally designed using LogicWorks™ 1.0. The simulation exercises matured as LogicWorks™ matured. With over 4,000 students using various versions of this manual and LogicWorks™ over nine years, I learned about many of the stumbling blocks students experience when using LogicWorks™ and felt that the experience I gained through my students could be used to advantage when writing this manual.

As you read the simulation exercises, you will notice, from time-to-time, instructions that are LogicWorks™ specific. These instructions are included because they answer recurrent questions my students have had when working through these exercises with LogicWorks™. The microprocessor design described by the simulation exercises can be built using other digital hardware simulators; although, I have never tried to use it with a different simulator. If you choose to do so, I would like to hear about your experience.

## Report Writing Guidelines

It is well known that students new to engineering have difficulty writing laboratory reports; yet there is precious little we can do within the time constraints of our courses to instruct students on acceptable report writing practices. This manual has a section on report-writing guidelines that can help students to write audience-centered, rather than writer-centered, lab reports. These guidelines help students to understand the context of report writing, and the major sections of a report. The guidelines also provide example schemes for handling section numbers.

In the report-writing section, students are introduced to two different report organization strategies: ‘task-oriented’ report organization and ‘top-down-oriented’ report organization. The task-oriented-report organization accommodates experiments consisting of many loosely related or unrelated tasks. A top-down description accommodates experiments where subcircuits are combined to achieve a whole. A recommendation for the report style that best fits each laboratory experiment is contained in the introductory material of each laboratory experiment. Of the lab reports required by the experiments in this manual, roughly half fit the task-oriented report style more naturally, while the other half are more naturally handled with a top-down style.

The report writing guidelines contained in this manual are learning oriented rather than research oriented. For example, the last section of most research reports is labeled, ‘conclusions’ and contains a summary of the measurements made along with the inferences the measurements will support. The report guidelines contain in this manual suggest labeling the last section as: ‘What Was Learned.’ In this section, I ask the students to report on what they learned while performing the experiment. The goal of this strategy is three fold. First, by reporting what they learned, students reinforce what they learned. Second, this section allows the instructor to view what the student believes they learned. Third, this synopsis of what students believe they have learned can be used during ABET reviews to provide evidence of achieved objectives.

## Assessment Guidelines and EC 2000

The last pages of each laboratory assignment have two tables that are designed to aid in assessing student performance. The first table lists each task and each facet of report writing. The maximum possible points

awarded for each line item are to be filled in by the instructor and made available to the students *a priori* so that they will know how their report will be graded. This document may be reproduced and distributed in class or via the Internet. The second table is a self-assessment worksheet that is designed to be completed by the student. This sheet is not to be graded. It is designed to provide a measure, by the students, of whether they believe the designed learning outcomes of each assignment were achieved. Both sheets are assessment vehicles in an EC 2000 compliant form.

## Order of Laboratory Assignments

The hardware and simulation experiments are designed to be mutually reinforcing. We have had success assigning hardware and simulation laboratory exercises in alternating weeks. This allows students to build a hardware circuit one week then reinforce the concepts with a circuit simulation of the same or related experiment the next week. We have experimented with assigning the simulation exercises first, then the related hardware experiments second, and vice versa, but have observed no difference in student performance.

A typical introductory semester-length course at the freshman and sophomore level will cover a subset of tasks in all five of the hardware laboratory experiments and all of the first five of the simulation experiments. An introductory semester-length course at the junior level will typically cover all five hardware laboratory experiments and all seven simulation experiments; although, in order to cover seven simulation experiments within the time allotted it may be necessary to assign two experiments to be conducted during the same week. If this is the case, it is possible to assign simulation labs 6 and 7 simultaneously, and, if necessary, to assign simulation labs 1 and 2 simultaneously; although, since simulation labs 1 and 2 require a substantial amount of work, these should be assigned simultaneously only after careful consideration.

## Errors

With a manual of this length, there are bound to be errors. If you find errors or have constructive criticisms, I would like to know about them. Also, my plan is to add to this manual as time permits. If you have suggestions for additional hardware experiments that have worked particularly well for you, or if you have suggestions for additional tasks in any simulation exercise, or for additional simulation exercises, I would like to know about them. You can contact me at [tylavsky@asu.edu](mailto:tylavsky@asu.edu) or via mail at:

Daniel Tylavsky  
Department of Electrical Engineering  
Arizona State University  
Tempe, AZ 85287-5706.

## Acknowledgements

This laboratory manual could not have been produced without the help of many people. Cheryl Johnson proofread the original manuscript and its many revisions and provided critical technical assistance and criticism of the work contained here. She did much of the artwork, and formatted the text to make its appearance more pleasing. Cheryl also designed the CD cover and constructed the Acrobat Reader formatted copy of each version of this manual, including all the links and control screens that make it easy to navigate through this manual. Her editorial comments helped immeasurably in improving the quality of this work and her technical assistance was invaluable. Srinivasan Devarajan donated much of his time to draw many of the figures, and provided invaluable technical and editorial feedback. He made many suggestions to improve both the presentation and content and was the guru in charge of creating the master CD. Ross Martin worked on the original design of the microprocessor simulation exercises. His original work is gratefully acknowledged. Philip Santeyan, Dana Griswold, Joel Barr, and Jennifer Johnson read the final manuscript to identify errors that we had missed and make suggestions from the undergraduate's point of view. Dr. Bassam Matar was the first professor, outside of Arizona State University, to teach out of various editions and versions of this manual. For his numerous suggestions, which led to many improvements, I am grateful. The undergraduate students at Arizona State University have worked through various versions of both the hardware and simulation exercises and continue to provide valuable feedback. I am grateful for all of their suggestions and the many things they have taught me.

Daniel J. Tylavsky  
Tempe, Arizona  
January 2002

# REPORT WRITING GUIDELINES

---

**Caveat emptor:** *There are several versions of this laboratory manual available from the publisher. Before you begin conducting your first experiment, check with your instructor to be sure that you have the correct version of this manual.*

## Preparing to Write a Report about a Digital Circuit Simulation

The best way to prepare for writing your simulation laboratory report is to ‘document as you go’. This means testing each of the circuits you simulate and recording the results of those tests in a laboratory notebook. Even if you have access to the circuits at the time you write your report, you will find that by methodically recording the results of your tests in a laboratory notebook ‘as you go’, you will be more likely to thoroughly test each circuit that you build and therefore be more likely to produce circuits with no hidden bugs. A hidden fault in a circuit you create may lie undetected until it and several other circuits (with their own potentially hidden faults) are combined into a complex design. The complex design may (and often does) subject each piece of the design to untested input combinations, revealing a hidden bug when the complex circuit is tested. Because the simulation exercises in this manual combine simpler circuits from early exercises to create complex designs in later exercises, your complex circuits are likely to expose any hidden faults in the circuits you create. Finding the one faulty component in the complex designs you create can be very difficult. Finding several faulty components is exasperating! Testing and recording the results of the tests ‘as you go’ will minimize the number of hidden faults in your circuits and save you much debugging time as your designs become more complex.

## Preparing to Write a Report about a Digital Hardware Experiment

The best way to prepare for writing a hardware lab report is to ‘document as you go’ while you are conducting the experiment. ‘Documenting as you go’ means keeping a thorough and informative laboratory notebook as you perform each task. In this notebook, you should record, at minimum, the schematic diagrams of the circuits you build and the input/output pairs you measure for each circuit. At this stage in your career, sitting at a laboratory bench and writing (as opposed to measuring) may seem like a waste of time; however, it is a habit that will pay dividends down the road. In school, it is unlikely that you will ever be asked to repeat a measurement months later. In the real world, you often simulate then build a system prototype, measure its performance, learn from these measurements, do a second-generation design, and then several months later repeat the earlier measurements on the new system. Detailed descriptions of experimental procedures used in performing measurements on the first design are essential

if an accurate comparison is to be made between the first- and second-generation designs. Unfortunately, it is not easy to predict exactly which measurements you are going to have to repeat and therefore it is accepted engineering practice to record all of your measurements in a laboratory notebook.

## **Hardware and Simulation Reporting Differences**

There are some obvious differences between using computer software to simulate a digital circuit and building the circuit using hardware in a laboratory. These differences affect the information you need to provide to the reader of your report if you are to provide them with information sufficient to duplicate your results. For example, in a digital-circuit simulation, the input/output measurements you make on your circuit will all be reported to you in the same way: by a value on your computer screen. In contrast, in the hardware laboratory, you will make measurements with several different instruments (e.g., voltmeter, logic probe, etc.); therefore you must report to the reader which instrument they must use if they are to duplicate your measurements.

Also, in the hardware laboratory, the circuit you make measurements upon will be disassembled and its parts returned to a common area long before you begin to write your report. Therefore, you must rely on your recorded measurements and observations to write your laboratory report. As the primary source of the data and observations for your report, your data sheets are the only trustworthy source of data; they are not tainted by the vagaries that time visits upon memory. By including your original data sheet in your report, you give your reader a document against which they can check the accuracy of the data in your report. You also give yourself one document that contains both the analysis of your experiment and your original notes and observations. By contrast, for simulation labs, we assume you have access to your circuit simulations at the time you write your report and can recreate measurements at will. Consequently, there is not as strong of a need to include your notebook data sheets with your simulation laboratory report - although including these original sheets and observations is never bad idea.

## **Lab Data Sheets**

Your original data sheets upon which you record your measurements and observations in the laboratory should be a part of your laboratory report. The following is a list of standard engineering practices for recording the results of laboratory experiments:

1. Lab data sheets can be handwritten on any kind of paper, although most engineering students use engineering paper.
2. You must print your name and date on each lab data sheet.
3. All individuals must write their ***own*** lab data sheets. (This means both you and any lab partner(s) must have individual lab data sheets. Using photocopies of another person's lab data sheet in your lab report is considered plagiarism.)

4. On your lab data sheet you should record all measurements and observations performed in the lab as well as the instruments you used to make the measurements. On this data sheet also record any truth tables that you verified.

Remember that one component of your target audience is your instructor/grader. Check with your target audience to see what information they want on your data sheet.

## Audience

Once you are finished conducting your experiment (and assuming that you ‘documented as you went,’) the next step, before beginning to write your report, is to answer the question, “Who is my audience?” Producing a report that is audience-centered, rather than writer-centered, requires analyzing your audience to determine what they know, how sophisticated they are, what their perspectives are, and then writing your report so that it is accessible to them. Typically, the knowledge, sophistication and perspectives of the individuals within your target audience will span a wide range. Your challenge is to write a report that is accessible to all of these individuals. Regardless of how many reports you write, you will find that writing your report so that it is accessible to your audience, will be a continuing challenge – it is for all writers.

The audience for the lab reports you write here is your grader. Typically, your grader will want you to write your lab report assuming your target audience is someone with a basic understanding in digital logic, such as another undergraduate who has taken an introduction-to-digital-design class. If you want to see how successful you are at writing to this target audience, give your lab report to a colleague and, (this is the best advice I can give you) listen to the feedback you get. You will find that your colleagues will not understand passages you have written that you believe should be understandable to a ten-year-old. When this happens to me, my first reaction is to want to explain to them why, the passage is clear. What I do, however, with very few exceptions, is to take the offending passage and rewrite it, trying to take their criticism into account. It is my experience, and the experience of many others that, if some passage is unclear to someone, then it is guaranteed to be unclear to others as well.

The reason our writing seems so clear to us, and not to others, is that we read our own words with an *a priori* understanding of what it is we are trying to say. Our understanding of our own writing is affected by neither the words on the page nor the organizing strategy we choose. Because of our familiarity with the subject and with the words we pen, we lose our ability to see our writing through the eyes of our audience. Our audience doesn’t have the same firsthand knowledge that we as authors have; therefore, they can provide feedback from the unique perspective of inexperience, a perspective from which our experience has eternally banished us. Accepting critical feedback on your writing and using it to improve your work is the mark of a maturing writer and will reward you in many ways.

Keep in mind that your grader is part of your audience. This means, as you write, you must keep in mind what your grader is looking for. It is hard to know what a grader will look for until you get your first

graded report returned. Most graders will look for proper syntax, punctuation, spelling, and will penalize you for not following any mandatory guidelines, such as title page format and report organization method.

It may seem that having two audiences (your grader and your colleagues) is unusual and that it only occurs in an educational setting. This is not the case. Once you graduate, you may be asked to complete memoranda to other organizations within your company. Your supervisor(s) within the company will check your memo first to ensure that they agree with its contents and then send it to the other organization, possibly for further comments; hence target audience for these memoranda will be both the other organization and your supervisor(s).

Writing technical reports and memoranda is not challenging. Writing them well is. As you practice writing and, especially, as you listen to the feedback you get, your writing will improve. I wish you luck with your writing as you progress through your education and career.

## **Scheduling Time to Write Your Report**

Most students find that the time spent writing a report is between one to two times the amount of time spent performing the experiment. Use this estimate to schedule time for your first few laboratory reports. Once you have written a few reports you'll be better able to estimate the time you'll need.

## **Formatting Guidelines**

Your instructor is likely to require that a report be submitted at the end of each laboratory exercise. Your instructor may also require that your report be produced using a word processor and that all pages be numbered. Staple your report securely. It is recommended that you place your name on every page so that any loose pages can be identified. Another way of keeping all of the pages together is to enclose your report in a large envelope.

Your lab report will consist of the following items:

0. Title page.
1. An introduction describing the educational objectives and expected outcomes of the laboratory experiments.
2. A main body in which you describe the results of your experiment.
3. A statement of which learning outcomes were achieved and which were not.
4. Laboratory data sheet(s). (Where necessary.)

## Report Organization

### Title Page

A typical title page will include the following information:

- Your name and ID.
- The course abbreviation and number.
- The name of your instructor and/or grader.
- The laboratory exercise number and title. (Identify the lab as either a hardware or simulation exercise.)
- The time your lecture meets. For example: MWF 10:30 a.m. or TTH 1:15 p.m.
- The date on which the lab report was submitted.

This title page organization needs to be tailored to fit your audience. The format provided here is the most general form needed for a course that is broken into many class sections. If the course that accompanies this laboratory has only one class-section, the information you need to provide on the title page to your audience (your instructor/grader) may be simpler. Check with your course instructor and/or grader to determine what they require.

### Introduction

The first major section of your report should be an introduction. Because the goal of these laboratory exercises is educational, you should list the learning objectives and outcomes of the assigned tasks in your introduction. If you are assigned to complete only a portion of the tasks contained in the lab experiment, be sure to list only those outcomes that you think the assigned tasks support. You may need to reword the objectives to fit with the point of view used in writing your report. For example, one of the outcome statements for hardware lab assignment 2 is:

“When you have completed this laboratory exercise you will be able to:

- “Make and interpret electrical voltage measurements.”

You may wish to rewrite this outcome as follows:

The lab exercises were designed so that, upon completion, the student (or ‘I’, or ‘the experimenter’) will be able to:

- “Make and interpret electrical voltage measurements [1].”

To avoid plagiarism, put in quotation marks any objective or outcome statements that you have taken verbatim and use the following to reference the source:

[1] D. J. Tylavsky, *Digital Design for the Laboratory: Hardware & Simulation (Using LogicWorks™)*  
Version 3, 3<sup>rd</sup> Edition, CenterPoint Publishing, Tempe, AZ, 2001.

## Main Body

The next major section of your report is the main body. The organization of the main body varies; it depends on the audience you are writing to and the information you want to convey. The reports you are asked to write for these laboratory exercises may be organized in many ways. Two organization styles discussed in this manual are the ‘task-oriented’ and the ‘top-down-oriented’ organizations. Regardless of the organization, there are two goals you will want to achieve: you must provide sufficient details to allow the reader to reproduce your work; and you must convince the reader that the design of your experiment and the measuring methods you choose support any conclusions that you draw.

### Reproducibility

To accomplish the goal of reproducibility, you must completely describe the circuits built or simulated, and their performance. This means, for each circuit you simulate or build, include a table in your report listing the measurements (i.e., input/output pairs) you observe. Also include in your report a schematic diagram corresponding to the circuit from which you recorded measurements. Each figure and table you use should be of your own construction; copying material from this lab manual and including it in your report is considered plagiarism. You should have a separate table and schematic diagram in your report for each circuit you build or simulate. Each table and schematic diagram should appear in the section of your report in which you discuss the corresponding circuit measurements. Including a table of measurements and the corresponding schematic diagram is easy to do if you use a laboratory notebook and ‘document as you go’; it is difficult to do if you don’t.

### Using Figures

The easiest, and hence universal, way of describing a circuit that you build, or a measurement that you make, is to supplement your prose description with a schematic diagram of the circuit<sup>1</sup>. Any figure you include in your report should be of your own creation. Copying a figure from this lab manual and including it in your report offers no evidence of experimental results and constitutes plagiarism. Each schematic should have a figure number and a title

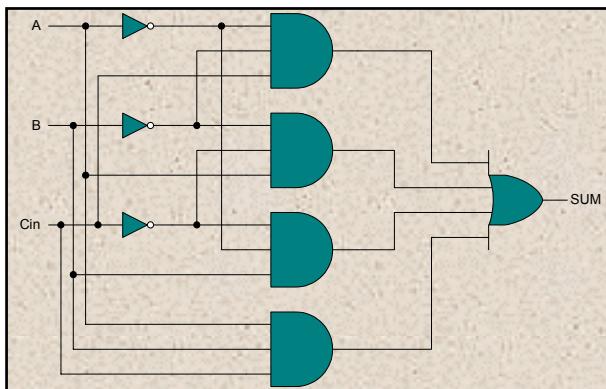


Figure 1. SOP implementation of the sum

<sup>1</sup> **Tip:** If you use LogicWorks™ for performing digital logic simulations, you can create simple hardware circuit schematics using LogicWorks then copy-paste them into a word-processed document. It is easier and much faster than drawing circuits by hand or by using most conventional drawing packages. Some students find that cutting and pasting from LogicWorks™ takes up a lot of memory. If this is a concern, you may find it more convenient to print out the schematics directly from LogicWorks™ on separate pages, assign figure numbers to each schematic and insert the pages into your lab report.

located (by convention) below the figure, as shown in Figure 1. If a figure is worth providing in your report, then it is worth discussing. You need to explain what the reader can learn from examining it<sup>2</sup>. At a minimum, every figure used must be referred to in the text of your report; unless the reader is told in prose to refer to a diagram, they have no reason to do so. For the schematics you create, use the same set of symbols and conventions that are used in this manual or in your textbook. (The symbols used in this manual are accepted conventions.) If the symbol you need is not used in this manual or in your textbook, use the symbols given in other professional reference texts.

### Using Tables

The easiest, and hence universal, way of describing a circuit's performance is to report the measurements you make in a table<sup>3</sup>. Any table you include in your report should be of your own creation. Copying a table from this lab manual and including it in your report offers no evidence of experimental results and constitutes plagiarism.

As an example of how to use a table, suppose you have a circuit schematic contained in Figure Y of your report. You might include in your report a table, called Table 1, containing the following data that you measured. (Presumably, the A, B, and F measurement points are clearly marked in Figure Y.)

**Table 1. Input-Output Characteristics for Circuit of Figure Y.**

Input (logic values)		Output (logic values)
A	B	F
0	0	0
0	1	-*
1	0	-*
1	1	1

\*“-” Represents logically indeterminate output values

Note that this table has a number and title located (by convention) at the top of the table. If a table is worth providing in your report, then it is worth discussing. You need to explain to the reader what the table contains. While the columns and/or rows of the table should have headings so that the reader can understand the contents of the table without referring to the text, these headings do not serve as a replacement for the prose in your report that describes the table's contents. Each row or column, where appropriate, should list the units of the data contained in the row/column, e.g., Volts<sup>4</sup>, logic values<sup>5</sup>, Amps,

---

<sup>2</sup> **Tip:** You may wish to observe the way figures are used in your textbook or in this manual and mimic it.

<sup>3</sup> **Tip:** You may wish to observe the way tables are used in your textbook or in this manual and mimic it.

<sup>4</sup> Units named after people, such as Volta or Ampere, should be capitalized. Other units, such as meter, or liter, should not be capitalized.

etc. At a minimum, every table used must be referred to in the text of your report; unless the reader is told in prose to refer to a table, they have no reason to do so.

### **Believability**

The second primary goal, of convincing the reader that the design of your experiment and the measuring methods you chose were consistent with the conclusions that you drew, is somewhat more difficult. The issue of believability has been largely solved for you because we have ensured that the circuit design, choice of the measuring instrument (e.g., logic probe, circuit simulation output etc.) and method for making the measurements are appropriate for each task. You must still, however, include your circuit design (and measurement techniques for hardware experiments) in your report since (you must assume) your reader will not have access to this lab manual when reviewing your report. Be sure to record your hardware laboratory measurement techniques (i.e., voltmeter, logic probe, LED, etc.) in your lab notebook since it may be days between the time you make a measurement and the time you begin to write your lab report.

Note the important role played by a lab notebook in the above discussions. A general rule is: students who take complete notes in a lab notebook find report writing is easier and faster. This will be true in every laboratory course you take.

### **What I Learned**

Label your last section ‘What I Learned’ and describe in this section what you learned by completing the experiment<sup>6</sup>. By writing about what you learned, you will be reinforcing your lab experience and helping to integrate it into the understanding of digital-design techniques that you are acquiring in your lectures.

The beginning of each laboratory experiment description includes a list of ‘learning outcomes’. You can use these as a starting point for discussing what you learned. Do not, however, limit yourself to only the list of ‘learning outcomes.’ During your experimental work, you will learn many things about

---

<sup>5</sup> When dealing with logic values, it is permissible in your laboratory report to leave the units unspecified provided the units are obvious from the context of the discussion.

<sup>6</sup> The last section of many reports is labeled ‘Conclusions.’ In this section, it is traditional to provide a summary of the results of your measurements and what these measurements allow you to infer about the circuits you built. Since these laboratory experiments are meant to be educational, the result of each experiment will be some bit of acquired knowledge or experience (instead of a collection of inferences); hence it is recommended that you label your last section ‘What Was Learned.’

designing, simulating, building, testing, and/or debugging digital logic circuits that are not specifically listed in the learning outcomes, such as:

- how the trainer board worked,
- debugging techniques that worked,
- test procedures that were effective,
- design techniques that were helpful,
- circuit failure modes that were unexpected,
- idiosyncrasies of the digital logic simulator,
- measurements that correlated (or did not correlate) well with expectations,
- the different debugging techniques needed when building and simulating a circuit,
- the role of circuit simulation in digital circuit design,
- an understanding of how a particular digital function could be implemented.

These bits of experience will be useful to you when you conduct subsequent experiments in digital logic in this course and in subsequent courses. Be sure to comment on them in this section. There is no single right answer here; what you learn from each experiment may be different from what your lab partner learns.

You might also comment in this section on things that did not work as expected, hypothesize as to why they did not work, and suggest improvements to the experiment design to eliminate the problems you encounter. You might also comment on other things you would like to explore if you had more time and resources. (For example, you might write that you would like to explore a way of building a function with NAND-NAND logic rather than AND-OR logic.)

If you did not learn anything from a particular task or experiment, report why you believe learning did not occur and suggest improvements that would allow desirable learning objectives to be met.

### Laboratory Data Sheets

Your laboratory data sheets should be included at the end of your hardware lab report as an appendix and given a title such as: ‘Appendix A: Hardware Laboratory Data Sheet(s)’. This appendix should be referred to somewhere in your report. Because the appendix is located at the end of your report, if the reader is not told of the existence of an appendix, they may never become aware of it.

### Laboratory Self-Assessment Table

Also include, as part of your report, a copy of the ‘self-assessment’ table, found at the end of each laboratory assignment. Label this section of your report ‘Appendix B: Self Assessment Worksheet.’ In this table, report your estimate of how effectively the learning outcomes set out at the beginning of the lab were achieved by the assigned laboratory tasks. Your responses in the ‘self-assessment’ table will be used by your instructor to gauge if the laboratory experiments designed in this manual are effective at achieving the stated learning outcomes. The responses you enter in the table **WILL NOT BE GRADED**; however, your instructor may require that this table be included and may award points for its presence.

Below this table is a space where you can offer suggestions for improving the laboratory experiments. If you learned nothing from a particular task, or if the results of the task were ambiguous or difficult to achieve/understand, tell us what you think needs to be improved and offer a suggestion (if you have one) for improving the task/task description.

## **Grading Guidelines**

At the end of each lab you will find a grading sheet containing a detailed breakdown of each report-writing task to which points may be assigned. Ask your instructor if these sheets will be used to grade your lab reports. If they are to be used, your instructor will provide you with a list of the points assigned to each report-writing task. You can maximize your grade by checking your lab report against the grading sheet to ensure that you have completed all tasks. The number of points assigned to each report-writing objective will depend on the learning objectives your instructor has selected for the course as a whole.

# TASK-ORIENTED REPORT WRITING GUIDELINES

---

**Caveat emptor:** *Before writing your report, check with your instructor to see if guidelines different from those stated here are to be used.*

There are many ways of organizing the main body of a report so that the information is accessible to your target audience. The organization depends on the information you have to convey. One organization style, the task-oriented style, accommodates experiments consisting of many loosely related or nonrelated tasks. The introductory laboratory assignments, in both the hardware and simulation portions of this laboratory manual, are comprised of loosely related tasks; hence, the task-oriented style works well for these experiments. In this style, each task is described as a stand-alone entity. No attempt is made to connect the results of these tasks in the mind of the reader.

## Main Body

The main body of a task-oriented report provides the following for each task:

- Task Statement: A statement of work in which you assign a unique number to each assigned task.
- Work Performed: A description of the work you performed while completing each task. (This section should include the methods and techniques you used for constructing any circuits and for making any measurements. This section should also include a discussion of the tests you performed to show that the circuit performs as your analysis and/or understanding predicted.)
- What Was Learned: Answers to any questions posed in the task and a discussion of what you learned from performing the task.

## Task Statement

Preparing a statement of work is very simple: simply retype the statement of the tasks assigned from this lab manual. If the task statement in the lab manual is long, you will want to shorten it by rephrasing in your own words what is written in the task. Another technique for shortening the task is to eliminate the discussions contained in each task that comment on the task but do not directly describe the circuit to be built or the measurements to be made. To avoid plagiarism, put the task statements that are taken verbatim in quotes and reference this laboratory manual giving the reference given [above](#).

## Description of Work Performed

In this section, describe the circuits you build, the measurements you make, and the instruments you use for making these measurements. Remember the goals you have in describing your work; first, you must

provide sufficient detail to allow the reader to reproduce your work; second, you must convince the reader that your quoted results can be believed. As you write this section, keep in mind the results of the audience analysis you performed.

## What I Learned

In this section list what you learned from the task. For a more detailed description, click on the blue text, [What I Learned](#).

## Numbering Scheme

Technical reports are (almost universally) organized using an outline-type numbering scheme. There are many such schemes to choose from. It is suggested, for the purposes of your lab report, that you use the following organization scheme: Create major report sections for ‘Introduction’, ‘Experimental Results’, and ‘What Was Learned.’ Within the ‘Experimental Results’ section, assign a separate subsection to each task assigned. Within each ‘Experimental Results’ subsection, assign a separate sub-subsection for:

- Task Statement.
- Work Performed.
- What Was Learned.

For example, in Hardware Laboratory Experiment 1 your list of headings might look like:

### Report for Hardware Lab # 1: Half Adder, Increment & 2's Complement Circuit

#### 1.0 Introduction

*(Here you would list the learning objective and outcomes.)*

#### 2.0 Experimental Results

##### 2.1 Task 1: Building the 1-Bit Half-Adder

###### 2.1.1 Task Statement

*(Here you would include a restatement of the work you were asked to perform.)*

###### 2.1.2 Description of Work Performed

*(Here you would include a description of the work you performed including a circuit schematic that depicts the circuit you build. Describe problems (if any) that you found and how you solved them.)*

###### 2.1.3 What Was Learned

*(Describe here what you learned, if anything, by performing this task.)*

- 2.2 Task 2: Test the SUM & CRY of the 1-Bit Half-Adder with LED
    - 2.2.1 Task Statement

*(Include the task statement.)*
    - 2.2.2 Description of Work Performed

*(Here describe the measurements that you made, bugs that you corrected. Include a table of the measurements that you made. Tell the reader whether your measurements indicate that the circuit is working correctly as it was designed to work.)*
    - 2.2.3 What Was Learned

*(Describe here what you learned, if anything, by performing this task.)*
  - 2.3 Task 3: Debugging
    - 2.3.1 Task Statement
    - 2.3.2 Description of Work Performed

*(Describe here the bug your lab partner introduced and how you tracked down the error. Include circuit schematic diagrams and tables that help the reader to understand what you did.)*
    - 2.3.3 What Was Learned

*(Describe here what you learned, if anything, by performing this task.)*
  - 2.3 Task 4: Completely Test Your Half Adder Circuit Using a Logic Probe
  - Etc.
- 3.0 What I Learned
- (Include here a summary of what you learned in performing this lab exercise.)*
- Appendix A: Laboratory Data Sheet
- Appendix B: Self Assessment Worksheet
- (Include here a copy of the table, found at the end of each laboratory assignment, giving your estimate of how effectively the learning outcomes, listed at the beginning of the simulation assignment, were achieved by the assigned laboratory tasks.)*
- As a second example, in Simulation Laboratory Experiment 1, your list of headings might look like:
- Report for Simulation Lab # 1: Debugging a Half and Full Adder**
- 1.0 Introduction

*(Here you would list the learning objective and outcomes.)*
  - 2.0 Experimental Results

- 2.1 Task 1: Building and Test the 1-Bit Half-Adder
  - 2.1.2 Task Statement

*(Here you would include a restatement of the work you were asked to perform.)*
  - 2.2.2 Description of Work Performed

*(Here you would include a description of the work you performed including a circuit schematic that depicts the circuit you build and a table containing the measurements you made. Describe problems (if any) that you found and how you solved them.)*
  - 2.2.3 What Was Learned

*(Describe here what you learned, if anything, by performing this task.)*
- 2.3 Task 2: Imbed the 1-Bit Half Adder in a Subcircuit
  - 2.3.3 Task Statement

*(Include the task statement.)*
  - 2.3.4 Description of Work Performed

*(Here you might state that you imbedded your circuit in a subcircuit. You could include a figure that shows the labeling of the input and output lines of the subcircuit. You would also describe in this section measurements that you made (including these measurements as a table in your report) and tell the reader whether your measurements correlated well with the results of task 1.)*
  - 2.3.5 What Was Learned

*(Describe here what you learned, if anything, by performing this task.)*
- 2.4 Task 3: Build a 4-Bit Increment Circuit
  - 2.4.1 Task Statement
  - 2.4.2 Description of Work Performed

*(Use a figure as an aid when you describe the design you proposed as a 4-bit increment circuit. Discuss any problem you encountered with the design or construction of your circuit and how you overcame them.)*
  - 2.3.3 What Was Learned

*(Describe here what you learned, if anything, by performing this task.)*
- 2.4 Task 4: Test the 4 Bit Increment Using Hex Numbers
  - Etc.
- 4.0 What I Learned

*(Include here a summary of what you learned in performing this lab exercise.)*

Appendix A: Laboratory Data Sheet

Appendix B: Self Assessment Worksheet

*(Include here a copy of the table, found at the end of each laboratory assignment, giving your estimate of how effectively the learning outcomes, listed at the beginning of the simulation assignment, were achieved by the assigned laboratory tasks.)*

**Caveat emptor:** *Before writing your report, check with your instructor to see if guidelines, different from those stated here, are to be used.*

# TOP-DOWN REPORT WRITING GUIDELINES

---

**Caveat emptor:** *Before writing your report, check with your instructor to see if guidelines different from those stated here are to be used.*

## Introduction

There are many ways of organizing the main body of a report so that the information is accessible to your target audience. The organization depends on the information you want to convey. One organization style, the top-down-oriented style, accommodates experiments where subcircuits are combined together to achieve a whole. The later laboratory assignments, in both the hardware and simulation portions of this laboratory manual, combine subcircuits, or MSI circuits, to create complex circuits that function in a coordinated way; hence, the top-down-oriented style works well for reporting on these experiments.

## Top-Down Description

In a top-down description, the goals of the report are the same as in a task-oriented description. That is, you must, somewhere in the report, describe the work you have done in sufficient detail so that:

- Your work is reproducible.
- Your results are believable.

You are also required to describe what you learned and what learning outcomes were achieved; however, the report is organized in a way that will allow the reader to more easily organize in their minds the way your design functions as a whole. It is organized so that first, a description of the whole you created is presented, then the whole is broken down into its major parts and the function and operation of the parts is presented. This process of breaking the parts down into their constituent parts continues until you reach the smallest part that you constructed in your exercise.

For example, in [Simulation Lab 3: Arithmetic and Logic Unit](#), you build an ALU by pasting together the NOT/NEG circuit and the ‘XOR/ADD circuit with pass-through’. The NOT/NEG circuit turn is made from the increment circuit (built and reported on in a previous lab) and some additional elementary logic gates. The ‘XOR/ADD with pass through’ you build using a 4-bit full adder and two 4-bit 2-to-1 multiplexers (built in previous labs) and some additional elementary logic gates.

The report you produce for this lab might have the following outline:

1.0 Introduction

2.0 ALU

    2.1 ALU Functional Description and Test Results

*(Here you describe the input/output characteristics of your ALU, using figures and tables to present your results. In addition, report on the test you made on the ALU that confirm it works and report what you learned from building the ALU.)*

    2.2 ALU Building Blocks

*(Here you might provide a schematic block diagram showing the major sub-blocks of the ALU, the NOT/NEG circuit and the XOR/ADD circuit with pass-through. And discuss their interaction.)*

        2.2.1 NOT/NEG Circuit Functional Description and Test Results

*(Here you might describe the input/output characteristics of the NOT/NEG circuit, using figures and tables to present your results. You would also report on the test you made on the NOT/NEG circuit that confirms that it works correctly. Also provide a schematic diagram showing the major building blocks that make up the NOT/NEG circuit and how these blocks interact to perform the NOT/NEG operation. [Since all of these building blocks were reported upon in a previous lab report, you could reference tables and figures in that earlier lab report to document what these sub-blocks do.] Also describe what you learned from building, testing, and debugging the NOT/NEG circuit.)*

        2.2.2 XOR/ADD circuit with pass-through

*(Here you repeat the same type of description you gave in the section 2.2.1, but apply it to the XOR/ADD circuit with pass-through.)*

        2.2.3 Calculation of Propagation Delay

*(Here describe the longest path in your ALU – possibly using a diagram – and show how you calculated the propagation delay time.)*

3.0 What I Learned

*(Include here a summary of what you learned in performing this lab exercise.)*

Appendix A: Self-Assessment Worksheet

*(Include here a copy of the table, found near the end of each laboratory assignment, giving your estimate of how effectively the learning outcomes, listed at the beginning of the simulation assignment, were achieved by the assigned laboratory tasks.)*

For [Simulation Lab 4: The Brainless Microprocessor](#) you build a brainless microprocessor consisting of a CPU, addressing circuitry, memory, and three buses: the address bus, control bus, and data bus. The highest-level description of your report would describe how these blocks were designed to interact and the test results you recorded when acting as the controller.

The report you produce for this lab might have the following outline:

## 1.0 Introduction

## 2.0 Brainless Microprocessor

### 2.1 Brainless Microprocessor Functional Description and Test Results

*(Here you describe the architecture of your brainless microprocessor, aided by schematic diagrams. Provide a schematic block diagram showing the architecture and major sub-blocks of the brainless microprocessor (the CPU, memory, and addressing circuitry), discuss their interaction, and provide the results of the tests you conducted to confirm that it works. Report also what you learned from building the brainless microprocessor.)*

### 2.2 CPU Functional Description and Test Results

*(Here you might provide a schematic block diagram showing the architecture and major sub-blocks of the CPU (the ALU, accumulator and buffer), discuss their interaction, and provide the results of the tests you conducted.)*

#### 2.2.1 Accumulator Circuit Functional Description and Test Results

*(Here you might describe the input/output characteristics of the accumulator register circuit, using figures and tables to present the results of your tests. You would provide a schematic diagram showing the internal circuitry of the register circuit and briefly describe how the register works. You should describe what you learned from building, testing, and debugging the register circuit.)*

#### 2.2.2 Buffer Circuit Functional Description and Test Results

*(Here you repeat the same type of description you gave in the previous section, but apply it to the CPU circuit.)*

#### 2.2.3 ALU Circuit Functional Description and Test Results

*(Because the ALU was described in a previous lab report, simply provide the input/output characteristics of the ALU and refer the reader to your previous lab report.)*

### 2.3 Memory Types Simulated

*(Give an introduction to the types of memory used in your system.)*

- 2.3.1 RAM Circuit Functional Description and Test Results
- 2.3.2 ROM Circuit Functional Description and Test Results
- 2.3.3 Output Port Circuit Functional Description and Test Results

#### 2.4 Addressing Circuitry Functional Description and Test Results

### 3.0 What I Learned

*(Include here a summary of what you learned in performing this lab exercise.)*

### Appendix A: Self-Assessment Worksheet

*(Include here a copy of the table, found at the end of each laboratory assignment, giving your estimate of how effectively the learning outcomes, listed at the beginning of the simulation assignment, were achieved by the assigned laboratory tasks.)*

**Caveat emptor:** *Before writing your report, check with your instructor to see if guidelines different from those stated here are to be used.*

# SIMULATOR TUTORIAL: USING LOGICWORKS™ FOR WINDOWS®

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Access LogicWorks™ for Windows®.
- Know how to use the Windows® interface and start application software.
- Know how to count using the binary number system.
- Have some familiarity with hexadecimal numbers. (Helpful but not necessary.)
- Create a truth table for a three-variable logic function.

**Equipment:** Personal computer with Windows® operating system.

**Objectives:** When you have completed this tutorial, you will be able to:

- Identify the various LogicWorks™ windows and describe their function.
  - Construct and execute a digital logic circuit simulation using the LogicWorks™ interface.
  - Apply names to signals and devices in a digital logic simulation.
  - Save an existing circuit simulation and open an existing circuit simulation.
  - Create, save, and use a subcircuit using the device editor.
  - Create a library.
  - Create bus lines and breakouts.
  - Use the hex keypad and hex display.
- 

## Introduction

This is a tutorial on LogicWorks™ for Microsoft Windows®. The purpose of this tutorial is to get you acquainted as quickly as possible with LogicWorks™. For a more thorough tutorial and reference information refer to the LogicWorks™ User's Manual.

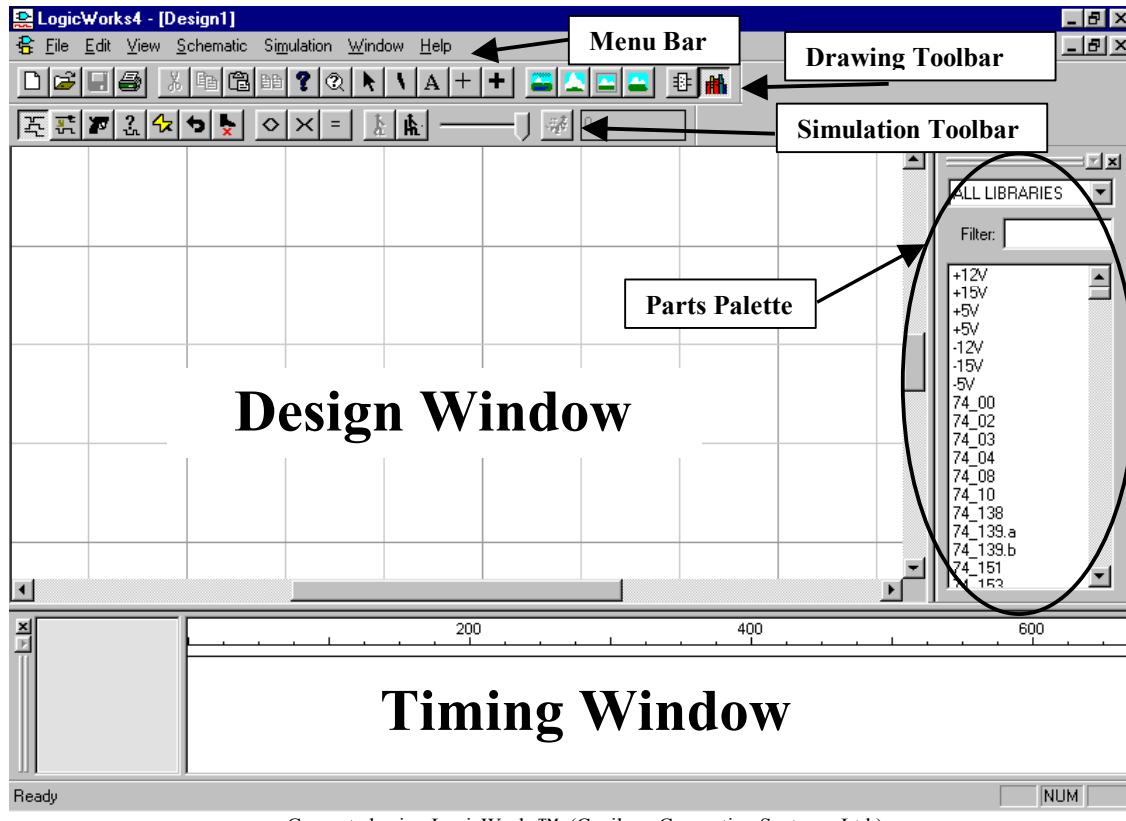
## Windows

Begin by starting LogicWorks™ on your computer. (If you have purchased LogicWorks™ follow your manual's instructions for loading and starting the software. If you are using a copy of LogicWorks™

supplied by an educational institution, check with your instructor or laboratory assistant for directions on getting access to and starting LogicWorks™.)

The instructions in this tutorial are consistent with only the latest version to LogicWorks™, version 4.06 for windows (as of this writing). (The LogicWorks™ version number is found by looking under the **Help > About** menu.) If you purchased LogicWorks™, you will need to download the latest patch for LogicWorks™ from their web site, <http://www.logicworks4.com/lw400/lw4update.html>. If you don't download the latest version, there are some features of LogicWorks™ that will not work correctly; for example you will not be able to cut and paste between LogicWorks and a word processing package – such as Word™. If you are using a copy of LogicWorks™ supplied by an educational institution, check with your instructor or laboratory assistant to determine that the version you will be using is current.

Upon starting LogicWorks™, you will see one window with several different sections as shown in [Figure 1](#).



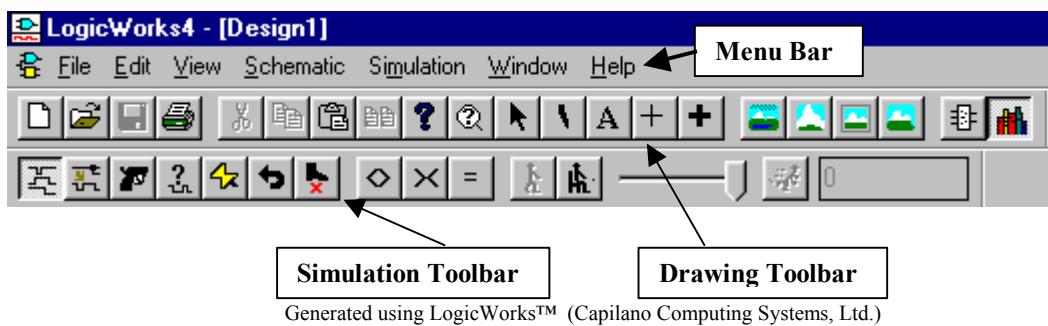
**Figure 1.** LogicWorks™ startup window.

The **design** window, shown in [Figure 1](#), is the area where you will draw your circuits.

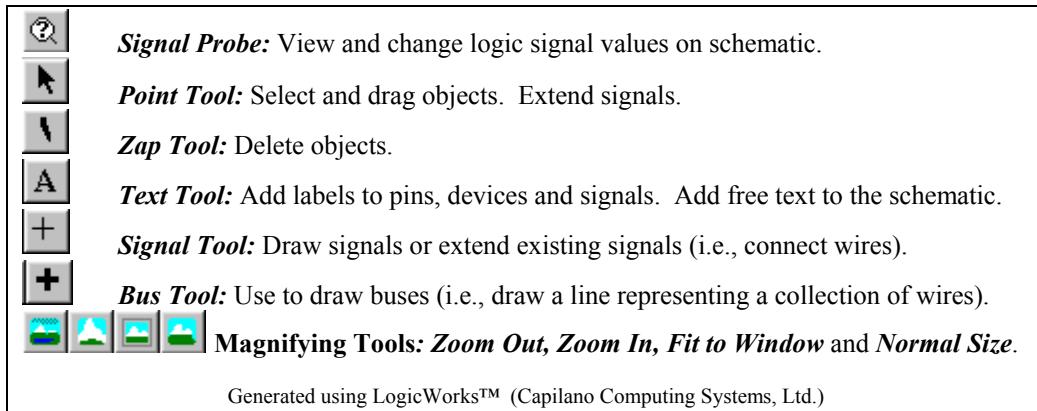
The **timing** window, also shown in [Figure 1](#), displays the values of different signals as they change with time when you run the simulator. If the **timing** window is not visible, go to the **main menu** bar and select **View→Timing Window**.

The **LogicWorks™ Menu Bar, Drawing Toolbar and Simulation Toolbar** are shown in [Figure 1](#) and shown enlarged in [Figure 2](#). These are taskbars from which tools and windows can be selected. If the **Drawing** and/or **Simulation Toolbars** are not visible, select **View→Drawing Toolbar** or **View→Simulation Toolbar** from the **main menu**.

A synopsis of the function of each tool in the **Drawing Toolbar** is contained in [Figure 3](#). You will use these tools to draw your circuit in the **design** window. Refer to the user's manual for further information regarding the **Drawing Toolbar**.



**Figure 2.** LogicWorks™ menu bar and toolbars.

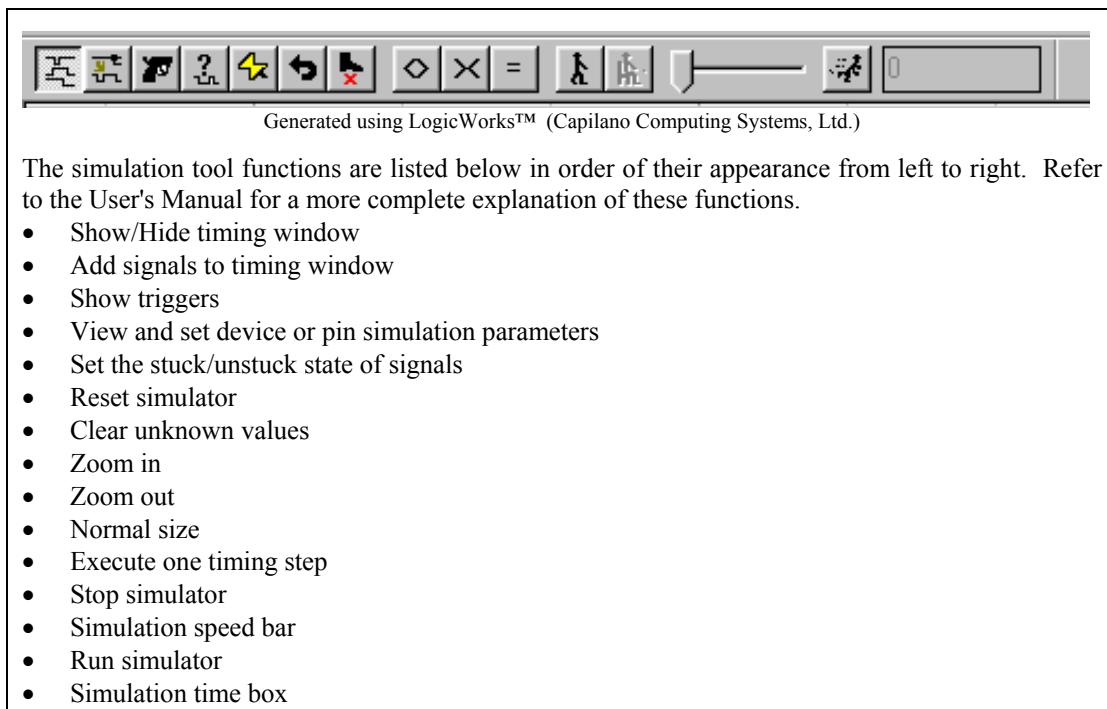


**Figure 3.** LogicWorks™ drawing tools.

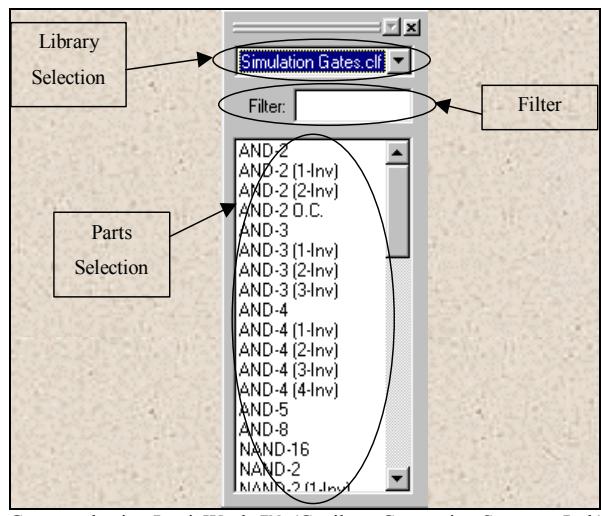
The **Simulation Toolbar** is shown in [Figure 2](#). The **Simulation Toolbar** is used to control the **timing** window parameters like speed, scaling etc. The simulation tools that you will use are described in [Figure 4](#).

The **Parts Palette**, shown in [Figure 1](#) and shown enlarged in [Figure 5](#), is used to select libraries of parts from which you select individual devices, switches, subcircuits, etc. that you will use to construct your circuits. If the **Parts Palette** is not visible, go to the **main menu** and select **View→Parts Palette**. Try

selecting different libraries by clicking on the down arrow in the **library selection** area and then clicking on a different library name (libraries end with the extension **.clf**). After selecting a library, the parts available in that library will appear in the **parts selection** area. You may show only a subset of the parts in a library by typing a keyword in the **filter** field. For example, to see only AND-2 parts of the Simulation Gates Library, type, "AND-2" in the **filter** field. Later in this tutorial, you will create parts and store them in a library that you will also create.



**Figure 4.** LogicWorks™ simulation tools.

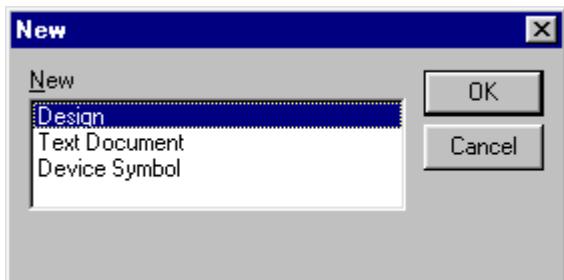


**Figure 5.** Parts palette.

## Dragging and Dropping Parts

LogicWorks™ uses a drag and drop interface. You select gates, switches, etc. from the parts palette, move them into the design window then paste their circuit symbols at the location you want. By performing the following steps, you will now learn how to place parts in the **design** window. After learning this basic skill, you will build a simple circuit and generate a subcircuit from it. You will be using these skills frequently in subsequent simulation labs. Let's get started.

1. Configure the LogicWorks™ "look". If you haven't started LogicWorks, do so now. You should see the window described in the previous section of this tutorial and shown in [Figure 1](#). You may close the **timing window** or the **parts palette** to reduce the clutter on the screen. (Click on the small  located within the boundary of a window to close it). Keep in mind that you will be using the **design** window and the **parts palette**. (To reopen or view a specific window, use the view menu commands as described earlier, e.g., **View→Parts Palette**.)
2. Confirm that your **design** window is open. If it is not open, open a new **design** window by selecting **File→New** from the **menu bar** and then selecting **Design** from the **New** window as shown in [Figure 6](#). Move the mouse cursor so that it is within this **design** window. Notice that the mouse cursor is a pointer or an arrow. This indicates that LogicWorks™ is in **point mode**. This is the default mode of LogicWorks™.



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 6.** LogicWorks™ new window.

3. Place a 2-input AND gate in the **design** window. Begin by selecting **Simulation Gates.clf** library from the **library selection** window of the **parts palette**. Find the part labeled "**AND-2**" in the **parts selection** window of the **parts palette**. "**AND-2**" means an AND gate with two inputs. Double click on **AND-2** and move the cursor into the **design** window. An AND-gate symbol will follow your mouse movement. Move the AND gate to any position in the **design** window.

If the orientation of the AND gate is not as you wish, you may change its orientation. You do this by pressing one of the **arrow** keys located on your keyboard. Try sequentially pressing all four of the

arrow keys and observing how the part's orientation changes. Notice that the output pin points in the direction indicated by the arrow key used.

Once you are satisfied with the gate's orientation and position, place the gate by clicking once with your **left** mouse button. As you move the mouse cursor away from the gate that you have just placed, you will notice that the cursor is still an AND-gate symbol. This means that the cursor is still in "insert-AND-gate" mode. If you click your **left** mouse button again, you will place another AND gate. This feature of LogicWorks™ allows you to place multiple parts without having to reselect the part from the library each time you need it. Let's try it.

4. Place a second AND gate in your **design** window. To do this, click your **left** mouse button again. Now return your mouse to **point mode** by pressing the **spacebar** or **escape** key on your keyboard.
5. Move a placed part in your **design** window. Select one of the AND gates that you placed by positioning the pointer anywhere on the gate then click and hold down the **left** mouse button. Move the symbol by dragging the mouse. Release the **left** mouse button when you are ready to place the part. (Note that the AND gate remains selected (highlighted) even after you have placed it. The gate will become deselected automatically when you perform another task. You can click on an empty area of the **design** window to deselect the part manually.)
6. Place a NOT gate. Scroll down the list in the **parts selection** area of the **parts palette** and locate the NOT part. Place this part in the **design** window. Press the **spacebar** to return to **point mode**.
7. Place additional parts. Place the following parts anywhere in the **design** window - we will arrange them to our liking later:
  - a. OR-2 part from Simulation Gates.clf.
  - b. Binary Switch from Simulation IO.clf. Click on the binary switch after it has been placed. Notice when you do this that the switch toggles between 0 and 1. To move a binary switch after it has been placed, press and hold down the shift key while performing the drag operation like you would on any other part.

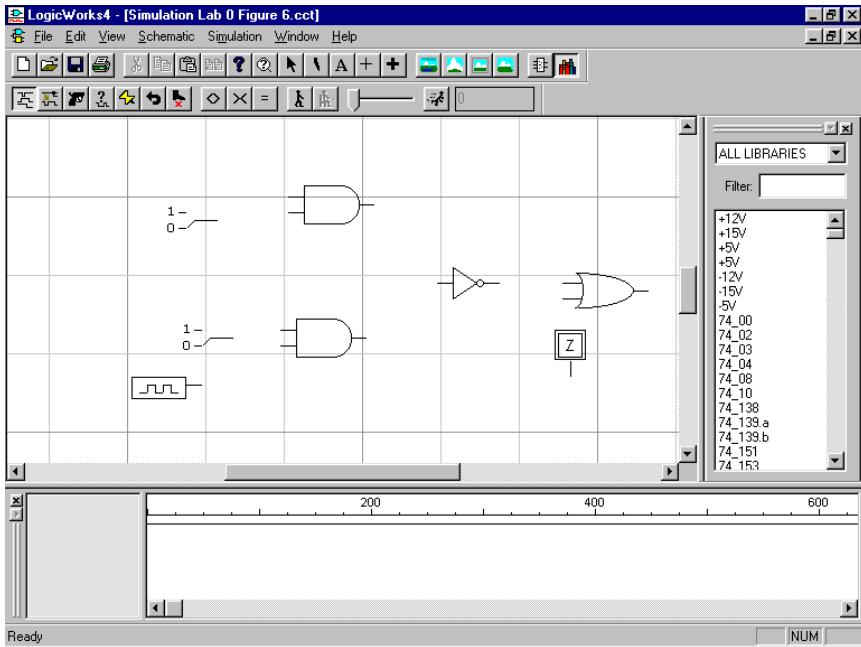
**LogicWorks™ Tip:** The switch will not toggle if the simulator controls have been set to **Stop**.  
To correct this problem, click the **Run Simulator** button on the **Simulation Tool Bar**.

- c. **Binary Probe** from **Simulation IO.clf**. You can use the binary probe on any pin or signal to check its value.
- d. **Clock**<sup>1</sup> from **Simulation IO.clf**.

---

<sup>1</sup> The clock is a periodic signal source used to provide a synchronizing signal simultaneously to many parts in a synchronous digital machine – such as the computer you are working on.

At this point, your circuit may appear similar to [Figure 7](#).



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 7.** Sample circuit design window.

## Drawing Signal Connections

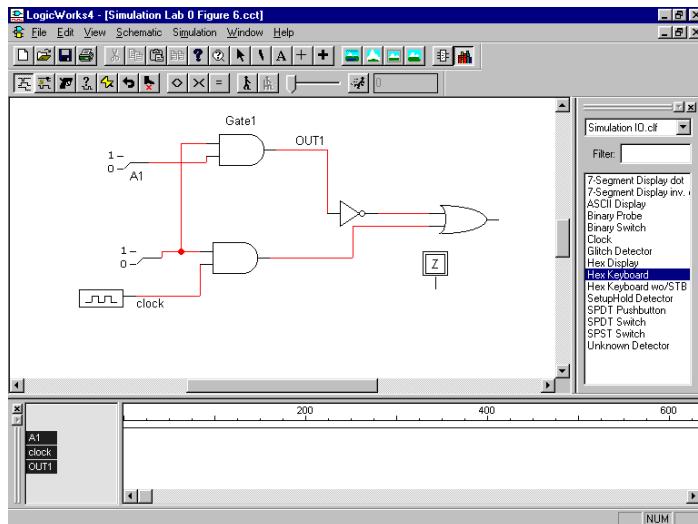
1. Draw a signal connection. Return to the ***point mode*** by pressing the ***spacebar***. Click and hold the ***left*** mouse button while the cursor is located directly over the end of one of the pins of an AND gate. Drag the mouse to draw a signal connection, which will appear in black. When you release the mouse button to anchor the signal line, the line will turn red.
2. Draw other signal lines from other pins. Click and hold the left mouse button while the cursor is located directly over the end of another pin and drag the mouse at a 45° angle. The signal line will bend at a 90° angle. Before releasing the ***left*** mouse button, alternately, press and hold down the ***ctrl*** and then the ***tab*** keys and then the ***ctrl*** and ***tab*** keys simultaneously and notice the change in the line routing. If you change your mind about drawing a line (before you anchor it), press the ***escape*** key. It will remove the line's image and change the cursor to the ***Point Tool***.
3. Connect different pins together. Using the method of drawing signal connections described in steps 1 and 2, try connecting the binary probe to the ***Clock*** and observe the display of the probe. The binary probe should be changing from 1 to 0 rapidly. If it is not, click the ***Run Simulator*** button on the ***Simulation Toolbar***. (Also try adjusting the ***Simulation Speed Bar*** on the ***Simulation Toolbar*** and observe the change in speed at which the binary probe toggles.) Try making other pin-to-pin connections as shown in [Figure 8](#).

## Naming Signals and Devices

1. Name a signal. Select the **text** tool from the **Drawing Toolbar**. Position the tip of the pencil-shaped cursor anywhere along one of the signal lines that you drew. Press and hold down the **left** mouse button. The cursor will change to an I-beam shape. While continuing to depress the left mouse button, move the I-beam to the point where you would like the signal name to be located. Release the mouse button. A blinking insertion marker inside of a rectangle will appear. Type any name for the signal, e.g. A1, RESET, ENABLE, AARON, etc. *Be careful not to leave any blank spaces before or after the name.* LogicWorks™ will interpret these spaces as part of the name. Chances are, when you use the name later on, you will have forgotten about the blank spaces and will be frustrated by a name that appears correct to you but unrecognizable by LogicWorks™. Press the **return** or **enter** key when you are finished. After hitting the enter key, the name you typed should appear magenta. If the lettering is black *after hitting the enter key*, this means that the cursor was not properly placed over a signal line or a device. If this is the case, try again. Once you are finished assigning names, press the **spacebar** to return to the **point mode**.

**ADVISORY:** If the cursor was not placed over a signal line or a device, pressing **enter** will add a new line to your text block. In this case, press the **esc** key and begin again.

Experiment by naming other signal lines. The names you entered should appear in the left portion of the **timing** window automatically. If you closed the timing window earlier, open it now to see if your signal names appear. If the names do not appear in the **timing window**, make sure **Simulation → Add Automatically** is checked. You can also select each name or line associated with the name by clicking on the name or line with the pointer and choose **Add to timing** from the menu bar or **Simulation Toolbar**. (You will know that you have selected a line or symbol when the line or symbol and its name become highlighted.) Refer to [Figure 8](#) for a reference.



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 8.** Naming signals and timing window.

2. Name the Clock signal. Using the **text** tool, name the signal connecting the *Clock* to the AND gate. Observe the *Clock's* signal in the **timing** window. If you do not see any changes occurring, try clicking the **Run Simulator** button on the **Simulation Toolbar**.
3. Name a part / device. You can also name devices with the **text** tool. Name one of the AND gates by clicking the pencil-shaped cursor on a gate and typing a name, e.g., U1, BRITT, etc., in the text box. Press the **enter** key after typing the name, then press the **spacebar** to return to **point mode**.

## Erasing Parts and Signal Wires

1. Delete the Clock symbol. Select the **zap** tool from the **Drawing Toolbar**. Click with the tip or bottom of the **zap** tool on the *Clock* symbol to delete it from the schematic. Press the **spacebar** to return to the point mode. Try an alternate method of deleting a part by selecting the part with the arrow pointer and then pressing the delete key (**Del**). (You will know that you have selected a part when it changes color.)
2. Delete signal lines. Select the **zap** tool again and delete some of the signal lines that you drew.
3. Undo. If you delete a part by mistake, immediately click **Edit→Undo** from the **main menu** or type **ctrl+z** to undo your last action. If you continue to use the undo, successively earlier changes will be reversed. The limit on how many steps can be undone or reversed varies from computer to computer but is on the order of 5-10.

## Saving, Closing, and Opening A Design

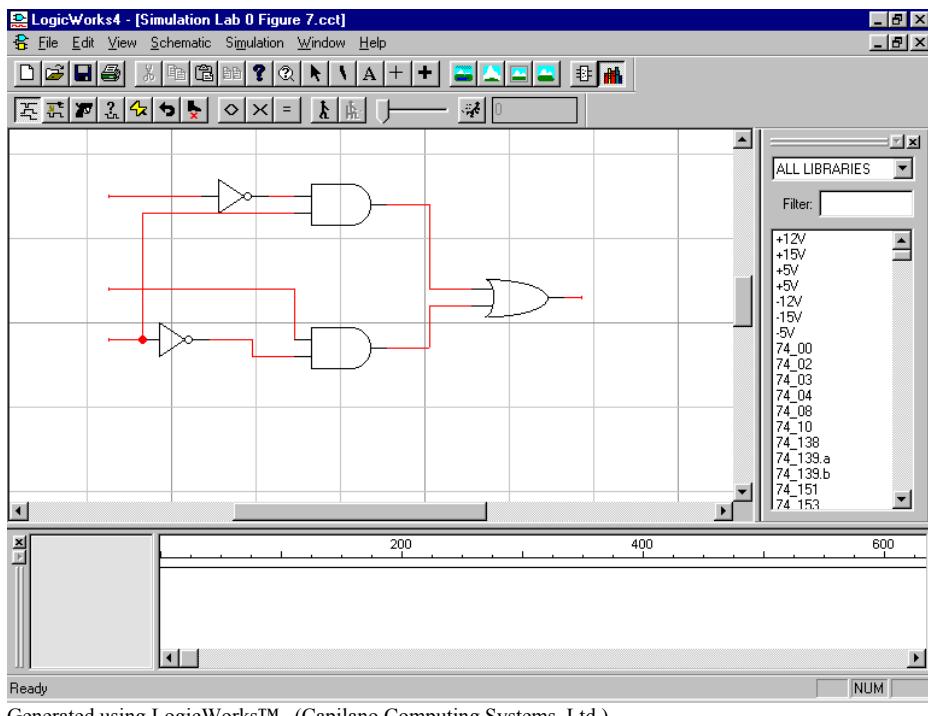
1. Save the drawing you have just made. Select **File→Save As...** from the **main menu**. Save your file with a **.cct** extension in any directory of your choice.
2. Close the design by selecting **File→Close** from the **main menu**. The **design** window should no longer appear on your screen if only one design was open.
3. Open the design you saved. Select **File→Open....** Select the file that you saved.

Refer to the LogicWorks™ user's manual if you want to learn more about LogicWorks™. A good way to learn to use any software package is to practice using it often. Try simulating circuits you learn about in class. Experimentally verify the truth tables of different devices. Simulating the hardware laboratory experiments before you build them in the hardware lab will help you gain skill with LogicWorks™ and help you understand the way a hardware lab experiment should behave before you begin to build a hardware realization and wrestle with all of the things that can go awry in such an experiment. The more you use the software, the more proficient you will be in its use.

## Creating a Subcircuit

In later labs, you will build a simple microprocessor using a hierarchical design philosophy. This means that you will build simple blocks and combine these blocks to build more and more complex blocks. The following procedure takes you step-by-step through the process of creating a *subcircuit*.

1. Close all open designs. Open a new design by selecting **File → New → Design** from the **main menu**.
2. Create the circuit shown in Figure 9 using the techniques described earlier in this tutorial.



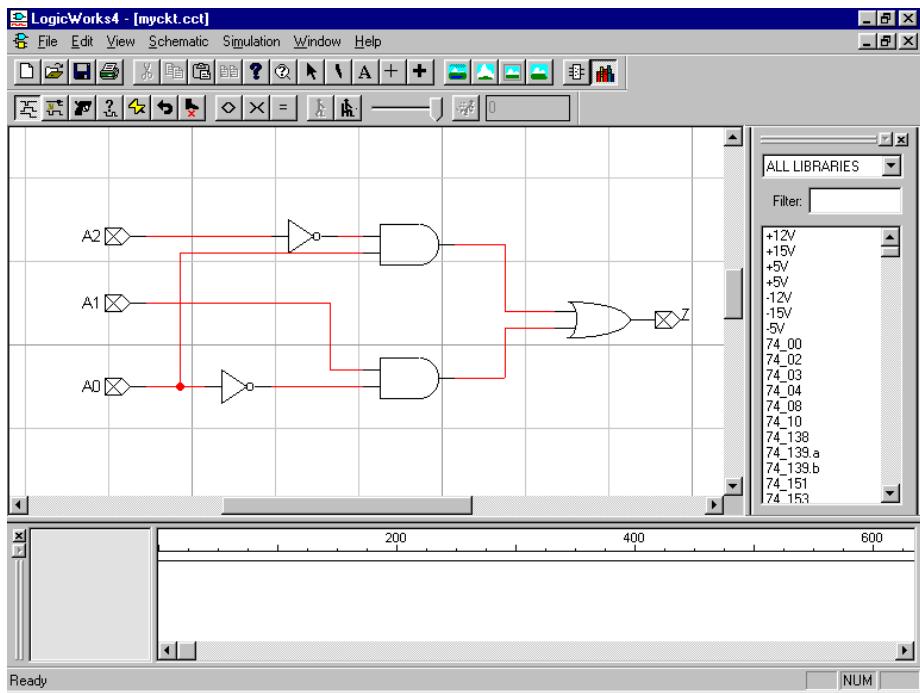
Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 9.** Sample subcircuit schematic.

3. **Exercise:** Before we can use this circuit to create a subcircuit, we need to verify that the circuit works correctly. If we assign in our minds, A, B, and C to the top, middle and bottom inputs and F to the output, the output will be  $F = \overline{A}C + \overline{B}C$ . (Do not label the wires with the text tool.) Create a truth table for this function. Using switches to control the inputs to the circuit and observing the output (by connecting it to a binary probe), methodically try every input combination and compare the results with that of your truth table. If your circuit does not perform correctly, use the signal probe from the **tool palette** and the debugging techniques described in [Hardware Lab 1: Debugging a Half and Full Adder](#) to track down the problem. (The advantage of using a simulator is that there are significantly fewer types of errors that can cause your circuit to malfunction.) If you can't find the errors in your circuit, ask a laboratory assistant for help.

4. Once you are convinced that the circuit works properly, delete any binary probes and switches that you placed during the above exercise. **Save** the design as ***myckt.cct*** in any directory of your choice.
5. To create a subcircuit you need to define the ***input*** and ***output*** pins of the subcircuit. To do this, you will attach LogicWorks™ input and output port symbols, to the input and output signal lines of the above circuit. Select the ***connect.clf*** library from the ***library selection*** area of the parts window. Find and double click the ***Port In*** part. This is the input port symbol. Place a ***Port In*** part on each of the three input signal wires as shown in Figure 10.

Press the ***spacebar*** to return to ***point mode***. Find and double click on the ***Port Out*** part and place one ***Port Out*** part on the output signal line as shown in [Figure 10](#). Press the ***spacebar*** to return to the ***point mode***.



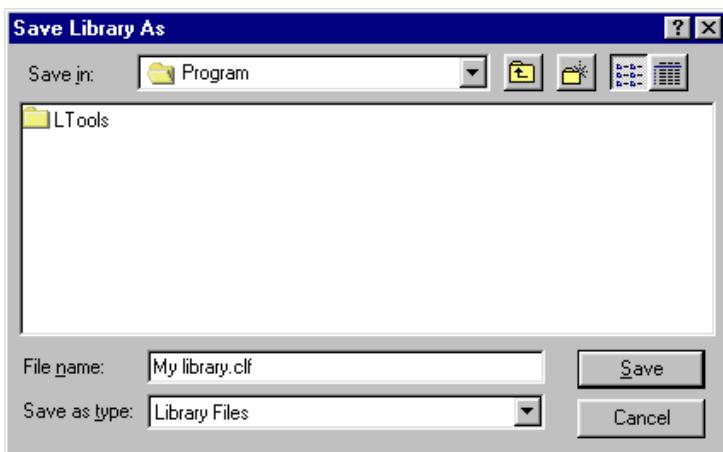
Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 10.** Sample subcircuit with input and output ports.

**Remember** that you can use the arrow keys to change the orientation of the ***Port In*** and ***Port Out*** parts.

6. Name the three input ports, A2, A1, and A0 beginning from the top. Name the output port, Z. To check that you have labeled the ports and not the signal wires, select a name by clicking on the port with your mouse pointer. The port should become highlighted simultaneously with the name if the name is linked to it. If you made a mistake, delete the label and try again. Be careful not to leave any spaces before or after your label.

7. Save your circuit by selecting **File → Save** from the **main menu**. Your circuit should look similar to [Figure 10](#).
8. You will now create your own **library**. Begin by positioning the mouse pointer anywhere within the **parts selection** area of the **parts palette**. Then, click the **right** mouse button. A submenu will pop up with the following options: **Edit Part**, **New Lib...**, **Open Lib...**, **Close Lib...** and **Lib Maintenance**. Click on **New Lib...** with the **left** mouse button.
9. A **Save Library As** window will open as shown in [Figure 11](#). Change to the directory in which you want to save your library. In the **File name** field, type in a name for the library, for example, **My library.clf** ([Figure 11](#)). Click **Save** when finished.



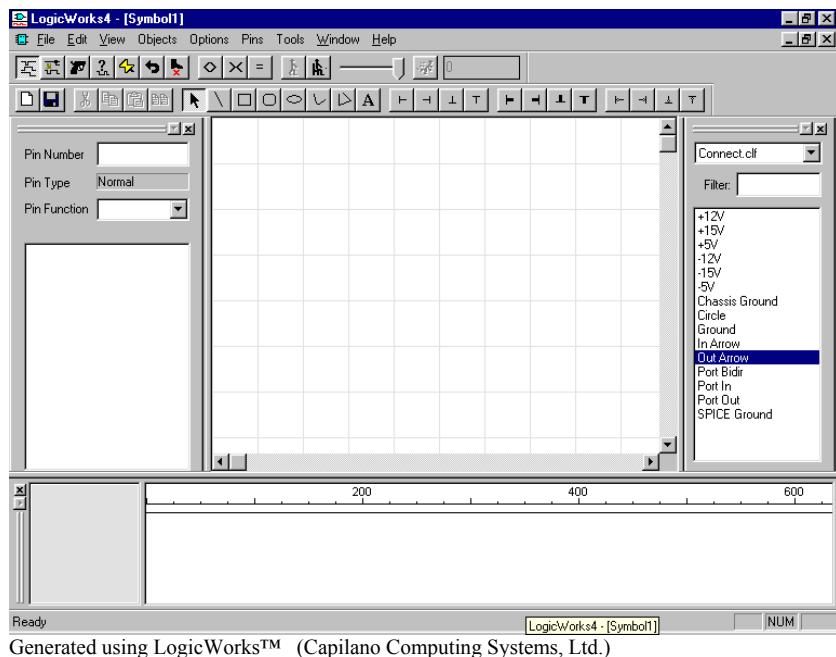
Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 11.** Save a new library.

10. Your library name will now appear in the **library selection** area of the **parts palette**. You will store all of the subcircuits that you will create in this library.

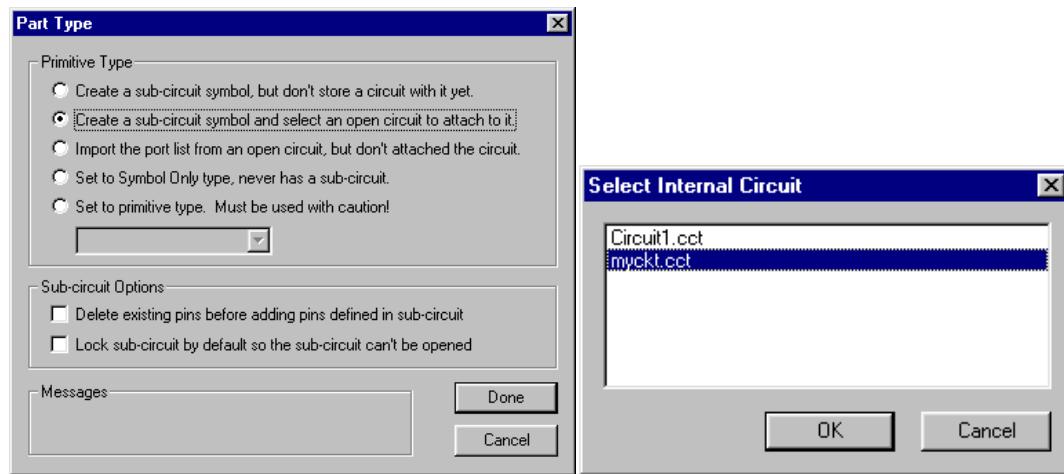
**LogicWorks™ Tip:** When you start LogicWorks™ later, your library name will not appear automatically in the **library selection** area of the **parts palette**. To open your library, position the mouse pointer anywhere within the parts selection area of the **parts palette** and click the **right** mouse button. Select **Open...** with the **left** mouse button. Browse until you locate the directory in which you saved your library. Click on your library file, e.g. **My library.clf**. Then click the **Open** button.

11. Next, you will create a subcircuit from the circuit that you just drew. Begin by selecting **File → New** from the **main menu**. The **New** window (see [Figure 6](#)) will be activated. Select **Device Symbol** from the **New** dialog box. The LogicWorks™ window will change and appear as shown in [Figure 12](#).



**Figure 12.** Device symbol window.

12. From the **main menu** select **Options → Subcircuit/Part Type**. A **Part Type** window will open as shown in the left side of [Figure 13](#). Select the second option, “*Create a subcircuit symbol and select an open circuit to attach to it.*” This will cause the **Select Internal Circuit** window to open as shown on the right side of [Figure 13](#).



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

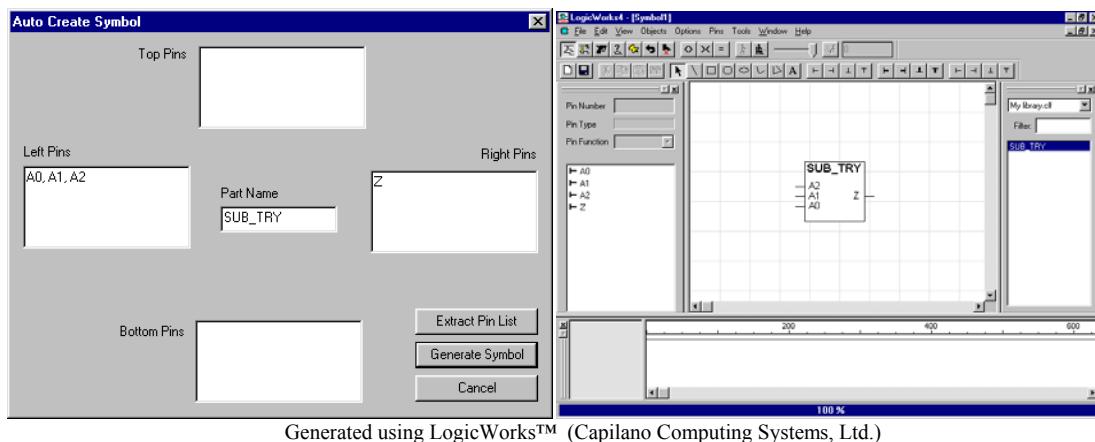
**Figure 13.** Selecting internal subcircuit.

**LogicWorks™ Tip:** The circuit choices displayed in the **Select Internal Circuit** window will consist only of those that are currently open in a **design** window. You will not be able to select circuits that are closed and saved on a disk.

Click on the circuit filename that you want placed into a subcircuit, in this case, ***myckt.cct***, and then click **OK**. This will take you back to the **Part Type** window. Click on **Done**.

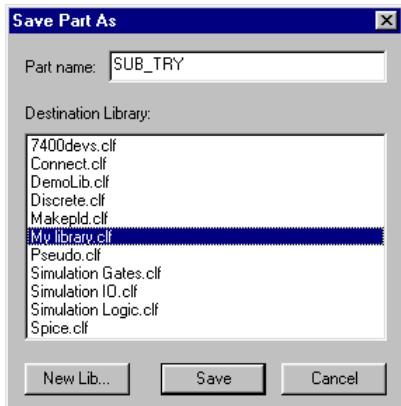
13. Select **Options → Auto Create Symbol...** from the **main menu**. An **Auto Create Symbol** window will appear as shown in the left side of [Figure 14](#). Click within the box under **Left Pins** to enter the names for the circuit's left pins. Type the following text for the three input pin names: **A0, A1, A2**. These are entered in order from the bottom of the circuit to the top of the circuit. The commas in the input string tell LogicWorks™ to put a standard unit of space between the input pins of your subcircuit. To add more space between pins, simply add more consecutive commas.

Now click within the box under **Right Pins** and type **Z** for the output pin name. In the **Part Name** field, type **SUB\_TRY**, this will be the name for this subcircuit. Click on the **Generate Symbol** button to create your subcircuit. Your screen should now appear similar to the screen shown in the right portion of [Figure 14](#).



**Figure 14.** Auto generation of subcircuit symbol.

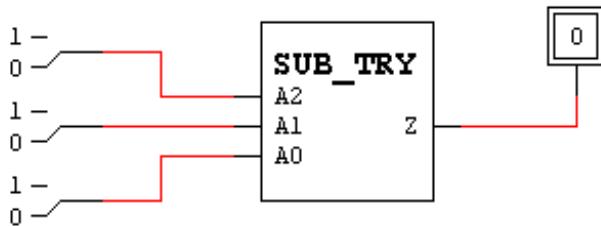
14. You have now created a subcircuit and associated a symbol with it. To save your subcircuit from the **main menu**, select **File → Save**. The **Save Part As** window should appear as shown in [Figure 15](#). Select the library that you created earlier (**My library.clf**) and click **Save**. SUB\_TRY will be stored in your library. When you open your library later, you should see SUB\_TRY listed as an available part. Close this window by clicking **File → Close**. Also close the design you used for the subcircuit by clicking **File → Close** again.



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 15.** Save Part As window.

15. **Exercise:** Next, we will place your newly created subcircuit in a new design. Open a *new* design by clicking **File→New** from the **main menu**. Place the subcircuit that you created, SUB\_TRY, in the **design** window by selecting your library in the **parts palette** and then selecting SUB\_TRY. Connect binary switches to A0, A1, and A2 as shown in [Figure 16](#). Label the switches C, B and A respectively. Connect a binary probe to Z. Verify that the truth table for your subcircuit is the same as the one you derived earlier for the circuit that you created in steps 1-9. (Save this circuit as **subdemo.cct**. You will need this circuit if you are asked to complete the demos at the end of this lab exercise.)



**Figure 16.** Circuit in a box **subdemo.cct**.

16. You can double click on the subcircuit part to view the subcircuit internals. When you do this, a new **design** window will open with the subcircuit internals in it. Close all **design** windows.

## Breakouts and Bus Lines

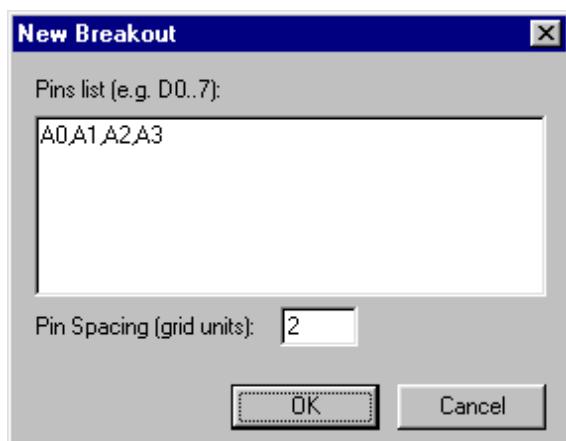
**Bus** lines are a collection of wires (connections) grouped together. Using buses simplifies large circuit diagrams by reducing the number of signal wires cluttering the design window. The simpler your circuit diagram is, the less likely you are to make mistakes. Bus lines are not essential or even desirable in many design layouts; however, using bus lines will greatly simplify the circuit schematics you will build and simulate when you complete the microprocessor simulation exercises contained in this laboratory manual.

Bus lines can be drawn with the **bus tool**, , from the **drawing toolbar**. To get access to anyone of signals within a bus, you will need to use a **breakout**.

Let's practice drawing bus lines and breakouts.

1. Close all open designs and open a new design.
2. Insert a breakout by selecting **Schematic** → **New Breakout...** from the **main menu**. The **New Breakout** dialog box will appear as shown in [Figure 17](#). We will have four pins in our breakout. Enter the pin list, A0, A1, A2, A3, as shown in Figure 17. When you are finished, click the **OK** button. When you move the mouse cursor into the **design** window, you should now see a breakout symbol that follows your mouse movement. Before clicking the **left** mouse button to place the breakout, try rotating the breakout by pressing the **arrow** keys on the keyboard. If you press the same **arrow** key twice, the pins will remain on the same side of the bus, but their orientation will reverse. Click the **left** mouse button to place the breakout. Press the **spacebar** to return to **point mode**.

**Tip:** If you wish the bus connection points on the breakout graphic to be spread further apart, enter a number greater than 2 in the **Pin spacing (grid units)** field of the **Breakout** dialog box.

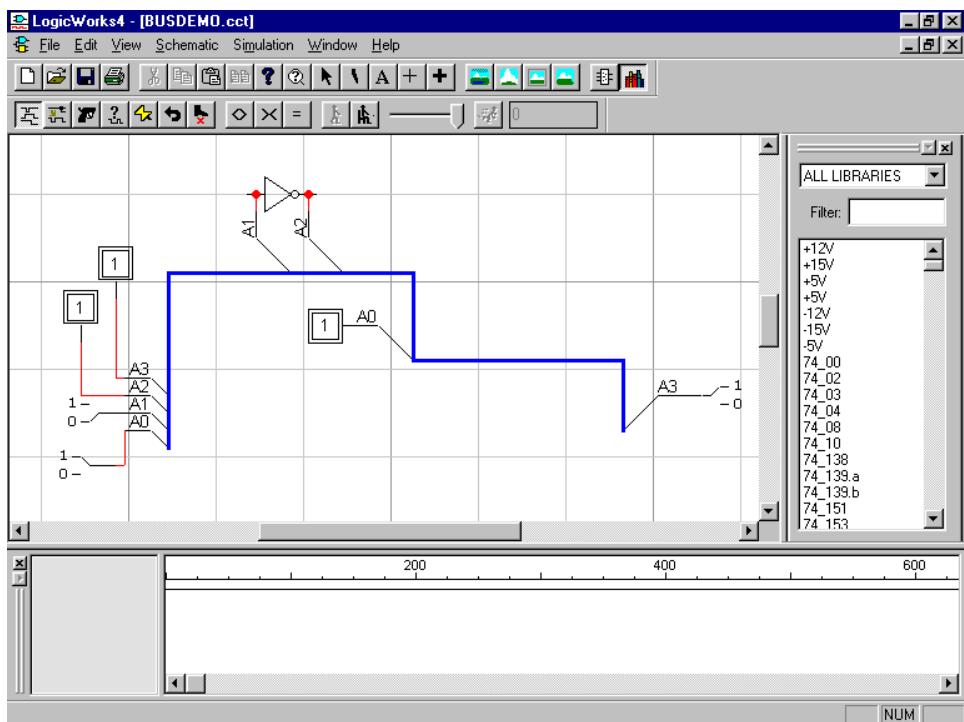


Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 17.** New Breakout dialog box.

3. Duplicate the schematic shown in Figure 18. Select the **bus tool** from the **drawing toolbar**. Connect a bus to the bus line of the breakout by positioning the **bus tool**, which appears as an X, at one of the ends of the breakout bus. Click and release the **left** mouse button and drag the mouse to draw the bus. As with signal lines, you can use the **ctrl** and **tab** keys to change the “bend point” of the bus lines while drawing them. Click the **left** mouse button again to anchor the bus and then either continue drawing the bus or press the **spacebar** if you are finished. As mentioned before, the bus is a collection of wires. Our bus contains four wires, A0 through A3.

4. Place other breakouts on the bus as shown in [Figure 18](#). Simply create new breakouts as described above, name the breakouts as shown in [Figure 18](#), then drag them so that the breakout bus portion is located directly over the existing bus line and click on the left mouse button. If you make a mistake, you can delete the breakout and insert a new one. Attach switches and probes to the bus breakouts as shown in [Figure 18](#). Place and connect an inverter so that bus signal A2 is always the complement of bus signal A1.
5. LogicWorks™ will attach the identically named pins of breakouts to the same “wire” within the bus. Test this feature by toggling the switch at one A0 breakout. The value displayed by the logic probe at the other A0 breakout should change accordingly.



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 18.** Bus example circuit.

## Using the Hex Keyboard and Hex Display

In the design of digital systems, we will often use buses containing many signal lines. One way to provide a signal to those lines and display the values on each line is to connect each to a binary switch and a logic probe. This can be a tedious process. The tedium can be reduced, if we use devices that can supply and display user-specified groups of signals. Devices that can generate or display signals in groups of four are known as hexadecimal keyboards and hexadecimal displays, respectively. To use these devices appropriately, you must be able to translate between hexadecimal and binary number representations. If

you already know how to translate between these two number systems, skip the next section. If you don't, the following section will give you enough background to be able to complete this lab tutorial.

### Hexadecimal (Hex) Numbers

There are three number systems that we will be using in the laboratory manual: binary, decimal, and hexadecimal. We are all familiar with the decimal or base-ten number system; it uses the symbols 0 through 9. In the binary number system, the symbols 0, and 1 are used. The correspondence between the representations of these number systems is shown in [Table 1](#).

The hexadecimal number system is a base-16 number system and is particularly useful when dealing with binary systems that use many bits of precision. This system uses the 16 symbols, 0 through 9 and A through F. [Table 1](#) shows the 16 hexadecimal numbers with their binary and decimal equivalents. For example, the hex (short for hexadecimal) number '4' is equivalent to binary '0100' and hex number 'B' is equivalent to binary '1011'.

**Table 1. Hexadecimal (Hex) Numbers and their Binary Equivalents.**

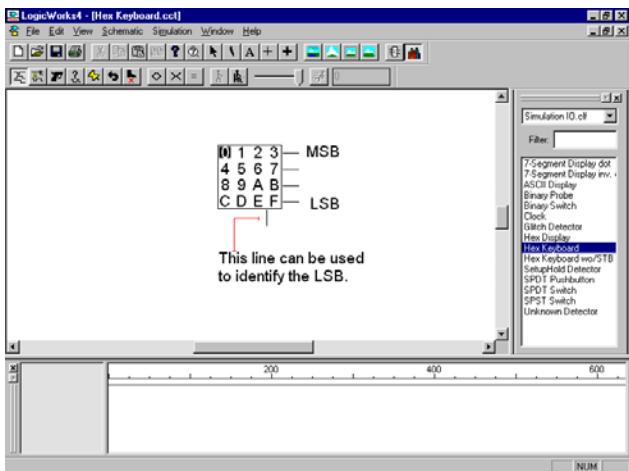
Decimal	Hex	Binary Equivalent									
0	0	0000	4	4	0100	8	8	1000	12	C	1100
1	1	0001	5	5	0101	9	9	1001	13	D	1101
2	2	0010	6	6	0110	10	A	1010	14	E	1110
3	3	0011	7	7	0111	11	B	1011	15	F	1111

By definition, the left-most binary digit (*bit*) is called the **MSB (Most Significant Bit)** and the right-most bit is called the **LSB (Least Significant Bit)**. Just as with decimal numbers, the order of the four binary bits is important; the binary number **0111B** is equivalent to the hexadecimal 7H, while the binary number **1110B** is represented by hexadecimal EH. LogicWorks™ provides two parts (or devices) to facilitate working with hexadecimal numbers. One part is called a **Hex Keyboard** and the other is called a **Hex Display**.

**Tip:** The **most common mistake** made by students in the simulation exercises is to incorrectly identify the least significant pin on the hex keyboard or display as the most significant and vice versa. When this happens, an input that the user thinks is 7H will be interpreted by their circuit as EH and vice versa. I have known many students who believed they correctly understood how to use the hex keyboard and display then spent **hours** looking elsewhere to find the bug in their LogicWorks™ simulation when their **only** problem was the incorrect use of the hex keyboard and/or hex display. Carefully read the next section on the use of these devices and your care will pay dividends the first time you use them.

## Hex Keyboard

The hex keyboard is found in the *Simulation IO.clf* library. It is shown in [Figure 19](#). The keyboard has 16 numbers (0 through F) that you can select with the mouse. The keyboard allows the user to control 4 output lines by entering one hex number on the keyboard. **The keyboard is designed such that the pin near the F is the LSB and the pin near the 3 is the MSB. Also, the side with the single output line<sup>2</sup> (between E and F in Figure 19) is always proximate to the pin supplying the LSB signal.**



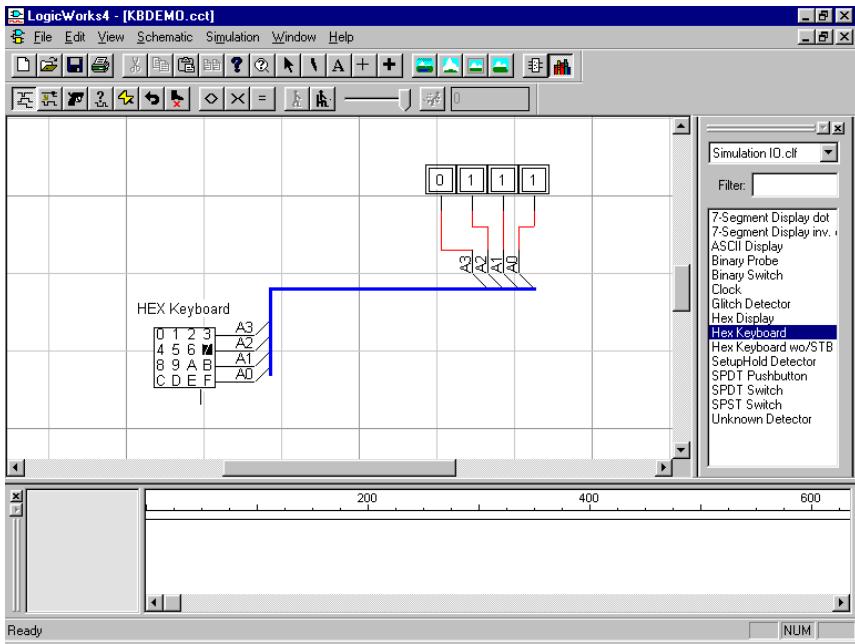
Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 19.** Hex keyboard.

Build the circuit shown in [Figure 20](#). If you want to move the hex keyboard after you place it, press and hold down the shift key while performing the drag operation like you would on any other part. Test the circuit by using all the entries in [Table 1](#) and observe the value displayed on the binary probe. You can change the connections of the keyboard and observe what happens when you make a wrong connection. This exercise may provide you with helpful experience in debugging circuits. (Save this circuit as *kbdemo.cct*. You will need this circuit if you are asked to complete the demos at the end of this lab exercise.)

---

<sup>2</sup> This output line goes high momentarily when you use the mouse to "depress" a key e.g., by positioning the tip of the arrow cursor over a key and clicking. We will not be using this line with any of the simulations described in this manual.



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

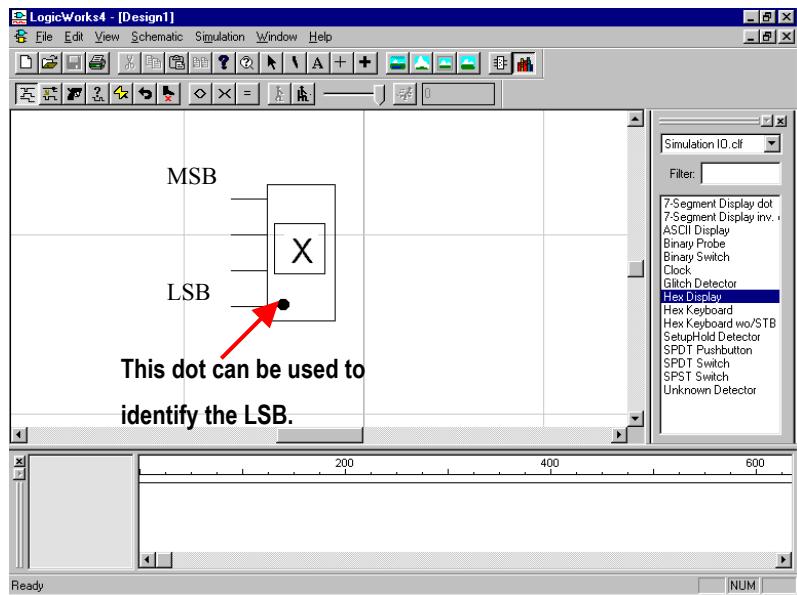
**Figure 20.** Hex keyboard example circuit *kbdemo.cct*.

### Hex Display

The hex display is also found in the *Simulation IO.clf* library. It is shown in [Figure 21](#). The hex display is used to view the signals on four lines as an equivalent hex number. As you saw in the previous section, without the hex display *four* binary probes are needed to view these signals. **The display is designed such that the signal supplied to the pin nearest the “dot” is interpreted as the LSB. The signal supplied to the pin furthest from the “dot” is interpreted as the MSB.**

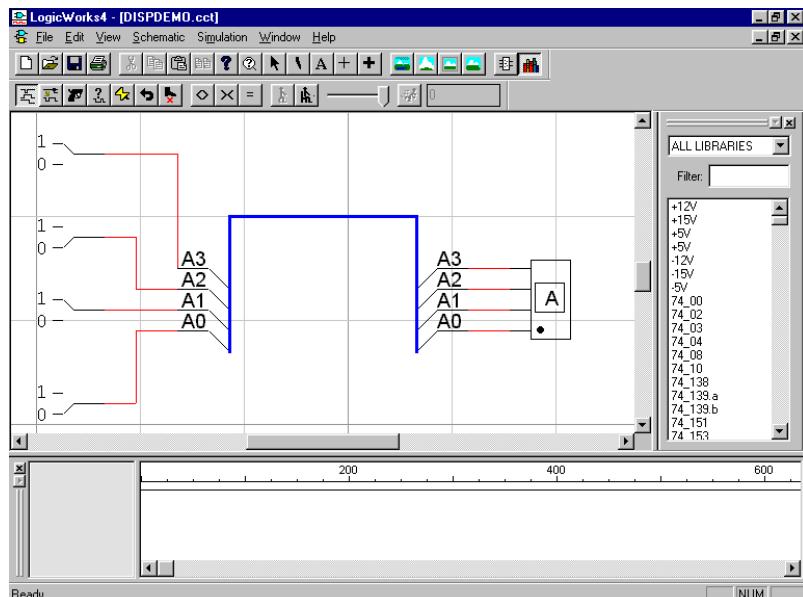
Build the circuit shown in [Figure 22](#). Test the circuit by keying in all the entries from [Table 1](#). Again, you should change the connections of the display and observe what happens when you make a wrong connection. This will help you down the road when you are debugging the microprocessor simulation circuits contained in these laboratory exercises. As a final test of your use of the hex keyboard, replace the four binary switches in [Figure 22](#) with a hex keyboard, and observe that the symbol you select on the keyboard is the same as the symbol you view in the hex display. (Save this circuit as *dispdemo.cct*. You will need this circuit if you are asked to complete the demos at the end of this lab exercise.)

Congratulations, you have completed the tutorial! You will use the skills you learned here to build a simple microprocessor in the subsequent simulation laboratory exercises. Good luck!



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 21.** Hex display.



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure 22.** Hex display example circuit *dispdemo.cct*.

## Demonstration

There are three circuits you built in the tutorial that you will use repeatedly in the laboratory experiments contained in this manual<sup>3</sup>. In the next three tasks, you are asked to demonstrate to a laboratory assistant three of the circuits you built in this tutorial. We have found (after years of assigning these laboratory exercises) that students who successfully complete the demos required below, save themselves frustrating hours of fruitless debugging in the labs where these skills are required. In task A, you will demo the subcircuit that you created and saved in this tutorial. In tasks B and C, you will demonstrate how to correctly connect hex keyboards and hex displays using the circuits you created and saved in this tutorial. Save all three circuits assigned in the tasks below on a floppy disk and take them to the laboratory assistant, who will instruct you on how to complete the demo. All three of the tasks below have to be demonstrated at one time.

### Task A: Creating a Subcircuit

Open *subdemo.cct* that you built earlier in this tutorial as shown in [Figure 16](#). Test it again to make sure it works correctly. Demonstrate to the laboratory assistant that your circuit works properly.

### Task B: Hex Keyboard

Open *kbdemo.cct* that you built earlier in this tutorial as shown in [Figure 20](#). Test it again to make sure it works correctly. Demonstrate to the laboratory assistant that your circuit works properly.

### Task C: Hex Display

Open *dispdemo.cct* that you built earlier in this tutorial as shown in [Figure 22](#). Test it again to make sure it works correctly. Demonstrate to the laboratory assistant that your circuit works properly.

---

<sup>3</sup> Note to Instructor: We have found that unless students are asked to demonstrate the completion of these circuits via a live demo or lab report, many students don't complete the tutorial and spend hours in later labs debugging incorrectly constructed subcircuits and hex keyboard/display circuits. We have found that this introductory demonstration also helps students understand the design project demonstration format of Hardware Experiment 5.

# SIMULATION LAB 1: HALF ADDER, INCREMENT & TWO'S COMPLEMENT CIRCUIT

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Use LogicWorks™.
- Interpret and construct schematic diagrams from Boolean algebraic expressions using AND/OR/NOT logic gates.

**Equipment:** Personal computer and LogicWorks™.

**Objective:** In this experiment, you will gain some experience building and debugging combinational logic circuits using LogicWorks™.

**Outcomes:** When you have completed the tasks in this experiment you will be able to:

- Describe the truth tables that characterize the addition of two single bit numbers.
- Write the Boolean algebraic expressions that characterize the sum and carry functions for the half adder.
- Build and debug a simulation of a circuit that will perform the half-adder operation.
- Build and debug a simulation of a circuit that will perform the 4-bit increment operation.
- Build and debug a simulation of a circuit that will perform the 2's-complement operation.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab, it is recommended that you use a task-oriented organization for your lab report. (If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Task-Oriented Report Writing Guidelines](#).)

---

## Prologue

The exercises contained in the simulation portion of this digital-design manual are meant to lead you step-by-step through the construction of an elementary microprocessor. At this point in your digital-design

course, this may seem like an immense feat, but don't worry, we'll take it one step at a time, and before you know it, you'll have constructed the complete simulation. Some people feel overwhelmed by the prospect of having to "build a complete microprocessor." Let me assure you that more than 4,000 students, with no knowledge of digital circuits, have started these laboratory exercises (while taking a course in digital design) and completed them successfully – and you will too!

To complete the microprocessor design, you will need to complete the first five simulation labs in this manual. In each of the first five laboratory exercises, you will either build pieces that will make up your microprocessor, or put the pieces together to form major subunits of your microprocessor. In the first two laboratory exercises, you will systematically design and simulate some elementary functions from AND, OR, NAND, NOR, and NOT gates that will be building blocks for subsequent exercises. In the third laboratory exercise, you will combine the circuits built in exercise 1 and 2 to form an Arithmetic and Logical Unit (ALU), which handles all the data manipulation operations in a microprocessor. In laboratory exercise 4 you will add memory and other simple circuitry to your ALU, to form a 'brainless' microprocessor. You will act as the brains of this microprocessor and control it to perform actions on stored data. In the fifth laboratory exercise, you will add control circuitry to complete your microprocessor design.

Beyond the first five exercises, there are two additional simulation laboratory exercises included in this manual. These are more advance exercises that allow you to enhance the functionality to your microprocessor.

As you work your way through these simulation exercises, notice that a large fraction (greater than 95%) of all the devices used are AND/NAND, OR/NOR, or NOT gates. At the completion of these exercises it is hoped that you will realize that all digital computers are merely complex combinations of these simple primitive logic devices.

In these exercises you are asked to build a simulation of a microprocessor rather than a hardware realization. Building a hardware realization of the microprocessor as prescribed in these exercises from discrete IC's (integrated circuits) requires several hundred IC's. The complexity of wiring together several hundred IC's puts a hardware realization beyond the scope of an introductory laboratory exercise. Further, when any circuit of this degree of complexity is to be constructed, a simulation of it is always completed first. Simulating a design before building it with hardware allows the conceptual design to be easily, reliably, and quickly built, tested, and modified. In practice, all complex systems are simulated before they are built. It is hoped that as you build the simulations in the subsequent laboratory experiments you will define for yourself the role that simulation plays in the construction of any large complex system.

## Introduction

In this laboratory exercise you will be building three circuits from primitive logic gates that will provide part of the functionality of a complete microprocessor. These circuits will allow you add two 1-bit<sup>1</sup> numbers (i.e., the half-adder function), increment a 4-bit number<sup>2</sup>, and perform the two's complement operation on a 4-bit number. In this exercise, you will also be modularizing the circuits you build so that they can be easily used and their functions easily understood. Creating modules (a.k.a. subcircuits) from complex circuits and using these modules to create more complex modules is a powerful strategy that will allow you to create complex circuits whose modularized diagrams can easily be interpreted and used.

## Preparation

The best way to prepare for a simulation laboratory experiment is to first make sure that you have the prerequisite skills listed on the first page of each experiment. Next, read the entire experiment, and complete the designs of the circuits you are asked to design. Your design procedure should include drawing schematic diagrams of the circuits you wish to build. Good luck!

## Pre-Lab Knowledge

In your lectures you have been introduced to primitive logic devices such as AND, OR, NAND, NOR, and NOT gates. If you are still unsure about the input/output relationships associated with these gates or how to construct a schematic diagram from a Boolean algebraic expression, refer to [Hardware Lab 1: Debugging a Half and Full Adder](#) or your textbook.

## Half Adder

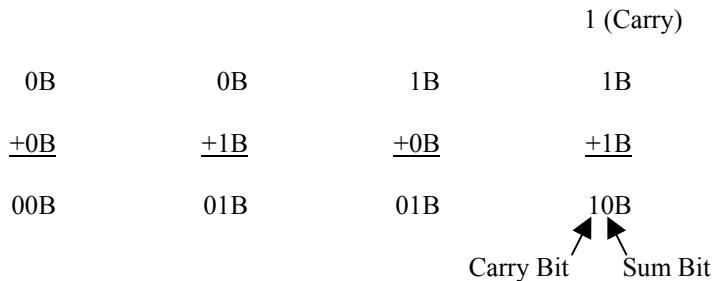
The microprocessor we wish to design will need the capability of adding binary numbers. In this laboratory exercise we will build a simple arithmetic circuit known as a half adder. The half adder is used to add two 1-bit binary numbers and produce a sum and carry output. (If you have completed Hardware Lab 1 or have studied how to create a schematic diagram for a half adder, you may wish to skip the next section on Binary Addition and begin reading the section on [Building the Half Adder](#).)

## Binary Addition

Using "B" to denote binary numbers (as opposed to "D" for decimal numbers), the addition of two 1-bit binary numbers is defined by the following equations:

---

<sup>1</sup> Bit is the shortened form of the expression "Binary Digit."



The only equation above that may be difficult to justify is  $1B + 1B = 10B$  – let's justify it. In the binary system, the symbol with the largest value is 1B, in the base-ten system the largest symbol is 9D. When we add two numbers in the base-ten system whose sum exceeds our largest symbol, we generate a carry of 1D to the next more significant position. Similarly, when we add two numbers in the binary number system whose sum exceeds our largest symbol, 1B, we generate a carry of 1B. This carry is added to the numbers residing in the more significant position (which are 0's in our example). If we assign the most significant bit of our 1-bit addition to be the carry (CRY) bit and the least significant bit to be sum (SUM) bit, we can represent the addition operation of two one-bit operands, A and B, using the binary-valued, addition-definition table of Figure 1-1. With digital logic gates we simulate truth values, not binary numbers; however, (and this is a subtle yet important isomorphism), if we reinterpret the binary values of Figure 1-1 as truth (or logical) values, then we get the truth table of Figure 1-2 – and we do know how to simulate truth values with digital logic gates!

A	B	A+B
0B	0B	00B
0B	1B	01B
1B	0B	01B
1B	1B	10B

Figure 1-1. Half-adder definition table.

A	B	CRY	SUM
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 1-2. Half-adder truth table.

<sup>2</sup> Your completed microprocessor will operate on 4-bit numbers or operands. Typical microprocessors operate on 8-, 16-, or 32-bit operands. We choose 4-bit, rather than 32-bit, operands because there is no conceptual insight gained by using large (e.g., 32-bit) operands. There is, however, significant advantage gained when using 4-bit circuitry; it is much less tedious to construct and debug.

To build a circuit that will add two 1-bit numbers, A and B, and give a 2-bit result, we will need to build the two digital logic functions defined in Figure 1-2, the SUM and CRY. There are two common canonical<sup>3</sup> forms we can use to realize these two functions: Sum-of-Products (SOP) and Product-of-Sums (POS) canonical form.

Let's use the SOP canonical form to realize the SUM function of Figure 1-2. To do this, define two auxiliary functions, K and L, whose values are 1 for one and only one of the input combinations that cause the SUM function to be 1. These auxiliary functions are shown in Figure 1-3. If we OR the K and L functions together we get the result K+L<sup>4</sup> shown in the truth table of Figure 1-3. By inspection we can see that SUM = K+L. This means that if we can realize the K and L functions, we can realize the SUM function as the OR combination of K and L. To realize the K function we observe that we want K to be 1 when A=0 and B=1. Clearly, this will be the case if we set  $K = \overline{A} \bullet B$ . Further, K must be equal to 1 ONLY when A=0 and B=1. The function  $K = \overline{A} \bullet B$  is consistent with that requirement; hence K can be realized using the circuit shown in Figure 1-3. Similarly, the function L must equal 1 when and only when A=1, and B=0. By inspection, the function  $L = A \bullet \overline{B}$  meets this requirement and can be realized using the circuit of Figure 1-3. If we connect the functions K and L to the inputs of a 2-input OR gate of Figure 1-3, we get the SUM function that we want. This circuit is the hardware equivalent to the Boolean algebraic expression:  $\text{SUM} = \overline{AB} + AB$ .

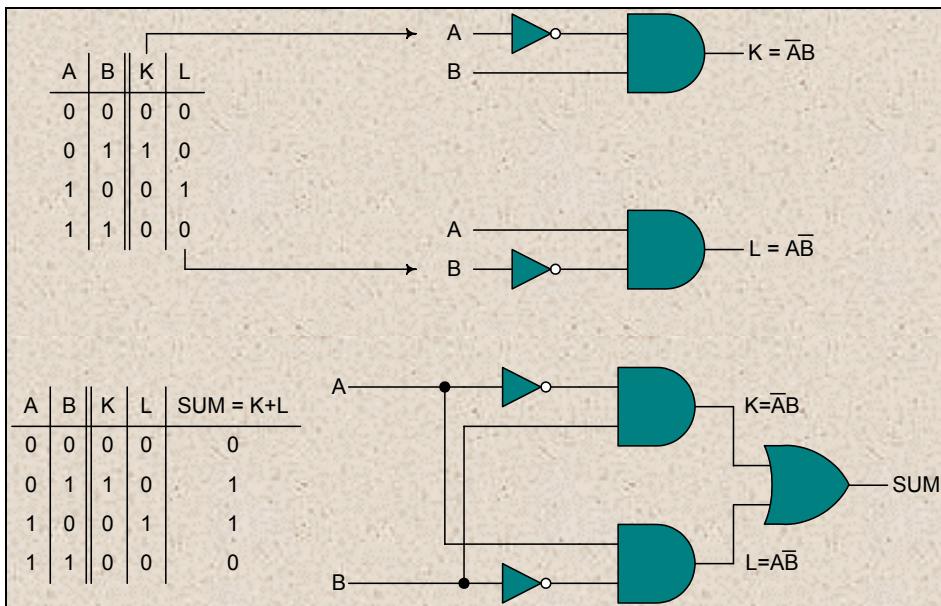


Figure 1-3. SOP implementation of sum for 1-bit half adder.

<sup>3</sup> A canon is a fundamental law or axiom. A canonical form is a form that is in some sense fundamental. The forms we use here are canonical in the sense that each AND or OR term contains every variable used in defining the function.

<sup>4</sup> We will use the standard Boolean algebraic notation in which “+” is used to indicate the OR operation, “•” is used to represent the AND operation, and an over-bar “—” is used to indicate a NOT (inverting or complementing) operation.

Similarly the CRY (carry) output can be expressed using the canonical SOP and canonical POS forms. The SOP form, as shown in Figure 1-4, requires only one logic (AND) gate for its realization. (Verify for yourself that this schematic diagram describes a circuit that could be used to realize CRY.)

A	B	CRY
0	0	0
0	1	0
1	0	0
1	1	1

Figure 1-4. SOP implementation of CRY for 1-bit half adder.

## Building the Half Adder

The circuit realizing the SUM and CRY functions that define the 1-bit half adder is shown in Figure 1-5.

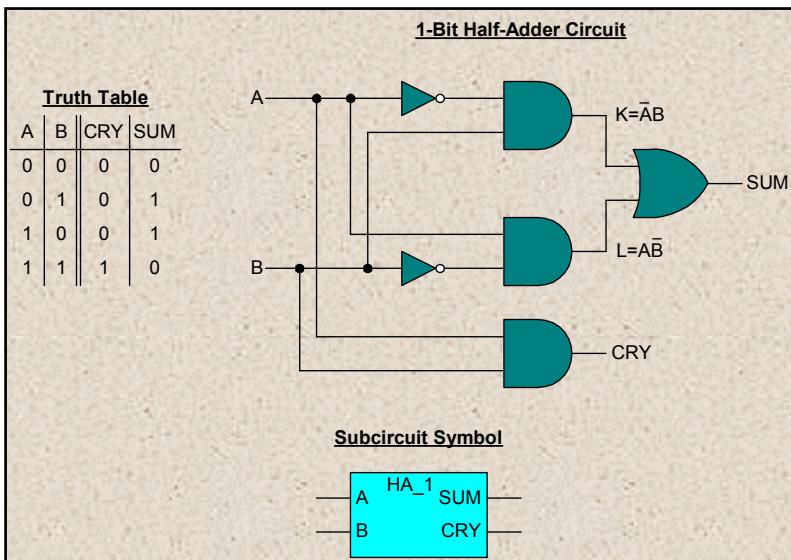


Figure 1-5. HA\_1: 1-bit half adder.

## Task 1-1: Build and Test the 1-Bit Half-Adder

Use LogicWorks™ to build and test the 1-bit half-adder circuit as shown in Figure 1-5<sup>5</sup>. (If you are not sure how to use LogicWorks™ refer to [Simulator Tutorial: Using LogicWorks™ for Windows®<sup>6</sup>](#).) To test this circuit, place switches on the inputs (A and B) and put binary probes on the outputs (SUM and CRY). Toggle the input switches through every (A, B) combination and compare the SUM and CRY outputs observed from the binary probe with those prescribed by the truth table of Figure 1-5. Record the results of these tests in your notebook. If the outputs your circuit generates agree with the outputs specified by the

<sup>5</sup> There are other circuits that will realize the half adder. For now, build the assigned circuits using the designs contained in the schematics provided. In many cases, we will be building upon and modifying these circuits. If you create designs of your own, you may find it difficult to incorporate later modifications. As you progress in these simulation exercises, you will have many opportunities to create your own designs.

<sup>6</sup> If you are using a Macintosh®, you can get the access to the equivalent tutorial via the web site: <http://www.eas.asu.edu/~cse120/>.

truth table of Figure 1-5, then your circuit is working correctly. If they do not agree, you have a problem with your circuit and you need to check that the connections have been made correctly.

The most common problem encountered when debugging a LogicWorks™ circuit is that the ‘wire’ drawn on the screen does not make contact with the intended devices at both ends. A useful trick is to click on a wire. The entire wire, along with the input/output pins of any devices to which it is attached, should become highlighted. If parts of it are not, you do not have the wire properly connected. The debugging procedures discussed in [Hardware Lab 1](#) may also be of use here if the results of your circuit tests indicate a problem. If you cannot detect the source of an error in your circuit after using the debugging techniques of Hardware Lab 1, see a laboratory teaching assistant (laboratory assistant) for help. Once you are convinced that your circuit is working properly, save your file. Give the file a meaningful name such as *ha\_1.cct*. This will help you keep track of your files.

**TIP:** Always keep backup copies of all your circuits and any libraries you create<sup>7</sup>.

Having created the half-adder circuit, the next step is to *document your work*. Do this by removing the switches and probes from the circuit. Then carefully label the inputs and outputs. (Refer to [Simulator Tutorial: Using LogicWorks™ 4 for Windows®](#) for instructions on labeling wires and pins.) You can test your label by selecting the wire, i.e., clicking on it. Selecting the wire will highlight the wire and everything connected to it. If the label is properly attached to the wire, the label will also be highlighted. You can name inputs and outputs anything you want except "0" and "1". LogicWorks™ interprets "0" and "1" as denoting logic levels. Re-save your circuit after you have labeled the inputs and outputs.

## Task 1-2: Imbed the 1-Bit Half Adder in a Subcircuit

After you have built and tested the circuit, the next task is to create a module by imbedding your circuit in a subcircuit. Imbed your half-adder circuit in a subcircuit using the subcircuit symbol shown in Figure 1-5. (To do this, you will have to remove the binary switches and probes you used in testing your circuit [if you have not done so already] and place properly configured input/output port symbols on the input/output lines, respectively. Then follow the directions for creating a subcircuit described in [Simulator Tutorial: Using LogicWorks™ 4 for Windows®](#).) Label your subcircuit ‘HA\_1’. ‘HA’ stands for ‘half adder’ and the ‘1’ indicates that the circuit operates on 1-bit operands. (In subsequent laboratory exercises you will build circuits that operate on binary numbers with multiple bits; hence, it is important to specify the bit

---

<sup>7</sup> Remember hard drives and networks crash often. If for some reason, you lose your files on your hard disk, you may have to recreate them in a short time to meet the laboratory report deadline; hence you should keep updated back-up copies of all of the LogicWorks™ circuits and libraries you create in these laboratory exercises.

<sup>8</sup> If you are using a Macintosh®, you can get the access to the equivalent tutorial via the web site: <http://www.eas.asu.edu/~cse120/>.

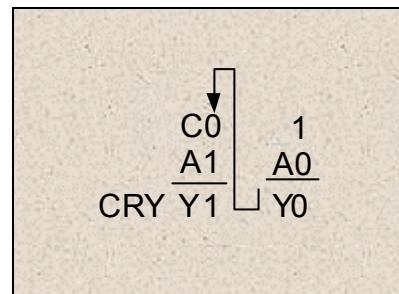
length in labeling each subcircuit so that the length of the operands accepted by each subcircuit can be determined simply by observing the label.)

After you have created the subcircuit, test the circuit using the same procedure described above. (I.e., place binary switches on the inputs of the subcircuit, binary probes at the outputs, and compare the output values obtained from your circuit with those contained in the truth table of Figure 1-5 for every combination of A, B inputs.) If the subcircuit does not work correctly, retrace your steps and try to debug your circuit. Ask a laboratory assistant for help if you need it. Once you are convinced that your circuit is working correctly, add the subcircuit to your library<sup>9</sup>.

Creating subcircuits is an important technique universally used in the creation of digital circuits. A subcircuit allows us to suppress the details of how a circuit operates and instead understand the subcircuit at the function level. By suppressing the internal details, subcircuits also greatly simplify schematic diagrams. We will see that as our circuits grow in complexity, using subcircuits is not only important but also necessary. For complex circuits, getting an overview of how the circuit functions without the use of subcircuits is next to impossible. The subcircuit symbols we create in this exercise will be used in schematics in subsequent laboratory exercises. If you keep your notation the same as we use here, you will have no problem using and combining your subcircuits using the schematic diagrams in subsequent laboratory exercises. So, for your own sake, follow the notation used in this lab manual. [Appendix A: LogicWorks™ Subcircuit Symbols](#) gives a summary of the notation used with the different subcircuits that you will be creating in these laboratory exercises.

## The Increment Circuit

An arithmetic operation that our microprocessor will use in its program-counter circuitry is the increment operation. The increment circuit we will need must add 1 to a 4-bit number<sup>10</sup>. Let's first design the simpler 2-bit increment function. Represent a 2-bit binary number A, with the most and least significant bits being A1 and A0 respectively. To add 1 to A, we need to add 1 to A0, getting the result Y0, plus a carry, C0, as shown in Figure 1-6. (This carry bit, C0, may be a 1 or a 0.) The carry is added to A1 to yield the result Y1 and another carry CRY. Since we are incrementing a 2-bit number, the carry, CRY, represents the most significant bit of the 3-bit result.



**Figure 1-6.** Incrementing a two-bit number.

<sup>9</sup> Recall you created your own library when you completed the [Simulator Tutorial: Using LogicWorks™ for Windows®](#).)

<sup>10</sup> Remember that we're building the simulation of a microprocessor that will operate on 4-bit operands and use 4-bit address values.

In practice, when we use this increment operation, sometimes we will be incrementing A by 0. Since  $A+0=A$  (here the ‘+’ signs means plus), any circuitry we design to perform the increment operation must give the correct result whether we are incrementing by 1 or 0.

To implement the 2-bit increment function, we will divide the operation conceptually into two functions: we will use one subcircuit to implement the addition of the increment value and the least significant bit, A0, and another subcircuit to add C0 and A1. The subcircuit that implements each of these functions is the half adder created in Task 1-2. Justify to yourself that connecting the half-adder subcircuits as shown in Figure 1-7 will perform the 2-bit increment function, where the INC input is the increment value.

In a similar fashion we can use four half adders to implement a 4-bit increment circuit. A 4-bit increment circuit will accept a 4-bit binary number as its operand and be controlled by the increment control, INC. (Label the increment control as INC in your design). Your circuit will produce a 4-bit binary number and a carry as output. If the input control, INC, is low, the output number will be the same as the input number. If the input control is high, the output will be one more than the input.

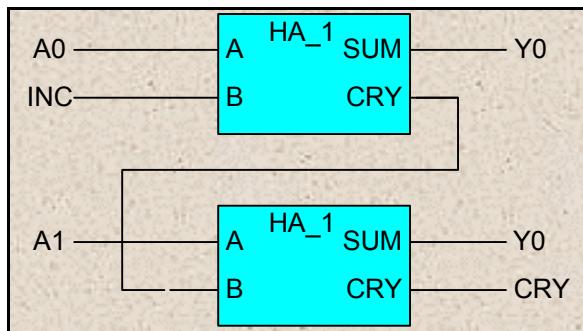


Figure 1-7. 2-bit increment circuit.

### Task 1-3: Build a 4-Bit Increment Circuit

Using your understanding of the increment function and the 1-bit half adder, build a 4-bit increment circuit using four 1-bit half adders. (Once you finish constructing the increment circuit using half-adder subcircuits, imagine what your circuit would look like if you did not construct it from subcircuit blocks. Your design is a good example of how we will use simple subcircuits to create more complicated circuits.)

### Task 1-4: Test the 4-Bit Increment Using Hex Numbers

By now you may have noticed the pattern of tasks used in creating circuits: first build the circuit, next test the circuit and (in many cases) imbed the circuit in a subcircuit. Let's test the 4-bit increment circuit you constructed in Task 1-3. Determining the minimum number of tests needed to guarantee that a circuit is working properly is a difficult problem – and beyond the scope of an introductory course. An alternative

testing technique is to observe the output for *every* input combination. Successful completion of such an exhaustive test applied to any circuit will guarantee that it is operating correctly.

The increment circuit you built in Task 1-3 operates on a 4-bit operand. In LogicWorks™, a convenient way to simulate a 4-bit operand and observe a 4-bit output is to use the hex keyboard and the hex display. Test your 4-bit increment circuit by connecting the four output lines of the hex keyboard to the four inputs of your 4-bit increment circuit. Similarly, connect the hex display to the four output bits of the 4-bit increment circuit and verify that the 4-bit increment circuit works in the desired fashion using hex arithmetic to check your results<sup>11</sup>. Note that this requires 32 tests. (Why?) Record your test results and include them in your report. (If you're not sure how to use the hex keyboard or hex display, review [Simulator Tutorial: Using LogicWorks™ 4 for Windows®](#).)

If you are not getting the results you expect, there may be one of two problems: either your circuit is constructed incorrectly, or you are using the hex keyboard or display incorrectly. The latter is usually the case. Remember that the side of the hex keyboard with the single output line<sup>12</sup> is adjacent to the LSB. Also remember that the hex display is designed such that the signal supplied to the pin nearest the “dot” is interpreted as the LSB.

### Task 1-5: Imbed 4-Bit Increment Circuit in a Subcircuit

Once your circuit works correctly, delete the hex keyboard and hex display. Then put the 4-bit increment circuit in a subcircuit using the symbol shown in Figure 1-8, label it “INC\_4” and include it in your library.

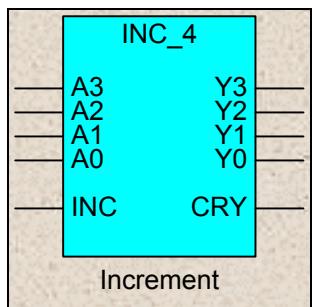


Figure 1-8. 4-bit Increment subcircuit symbol.

### Task 1-6: Use INC\_4 to Perform Two's-Complement Operation

In the course associated with this laboratory, you will learn about using the two's-complement number system to represent both positive and negative binary numbers. In the two's-complement number system, the sign of the number becomes an integral part of the representation of the number. Most computer

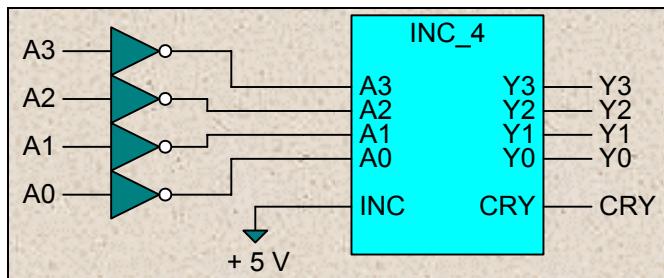
---

<sup>11</sup> If you are not familiar with the hexadecimal number system, review the Hexadecimal Numbers section in [Simulator Tutorial: Using LogicWorks™ 4 for Windows®](#).)

<sup>12</sup> Remember that we will not be connecting this output line to any devices in the simulations described in this manual

arithmetic is performed using numbers in two's-complement form – our microprocessor will be no different. In our microprocessor, we will want the capability of negating two's-complement numbers; If you have not yet learned about two's complement numbers in your course, don't worry; for now all you need to know is that negation of two's-complement numbers is handled by complementing each bit of the number and adding 1 to it. The circuit shown in Figure 1-9 uses the increment subcircuit you created in Task 1-5 along with four inverters to perform the two's-complement operation.

Note that by connecting the 'INC' input in Figure 1-9 to +5 Volts, the circuit will always perform the two's-complement operation. Build the circuit in LogicWorks™ and verify that it performs the two's-complement operation. In our earlier test of the increment circuit, we exhaustively tested every input combination. As you gain experience building and testing circuits you will realize that once a circuit works correctly for certain input combinations, it is *likely* to work correctly for all input combinations. It is left to you to decide what constitutes a sufficient set of tests to make it likely that the two's-complement circuit is operating correctly. The test should be thorough enough to prove that the circuit works beyond reasonable doubt. One set of test inputs is not enough to test the circuit. Justify in your report why successful completion of your tests makes it likely that your circuit is operating correctly. (Hint: Be sure to test the carry). Remember to record the details of your tests in your report.



**Figure 1-9.** A circuit that performs the two's-complement operation.

Once you have tested this circuit and have shown that it works correctly, save it. In Simulation Lab 3 you will be modifying this circuit and using it as one part of the arithmetic and logic unit for our microprocessor.

### Task 1-7: Backup Your Files

Over the years many students have had their libraries and files destroyed by many causes. Because these simulation exercises build upon one another, some students have needed to redo three or more simulation labs just to be able to perform the most recent one assigned. Take a moment and back up your libraries and circuit files on a new floppy disk and mark it 'SimLab#1'. Do this using a different floppy disk after every lab and you just may save yourself much unnecessary work.

# SIMULATION LAB 1: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 1-1: Build and Test the 1-Bit Half-Adder		
Task 1-2: Imbed the 1-Bit Half Adder in a Subcircuit		
Task 1-3: Build a 4-Bit Increment Circuit		
Task 1-4: Test the 4-Bit Increment Using Hex Numbers		
Task 1-5: Imbed 4-Bit Increment Circuit in a Subcircuit		
Task 1-6: Use INC_4 to Perform Two's-Complement Operation		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

<b>Caveat emptor:</b> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used
• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.
• Include lab data sheets for all labs. Use lab data sheets to document the lab as you perform it. It saves a lot of time when writing the report.
• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.
• <b>Label</b> all figures or circuits. (E.g. Figure 1. Schematic of . . . , Figure 2. . . , etc.)
• Refer in the text of your report to all circuits or figures that you include in the body of your report.
• Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

Put an 'X's' in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use '5' to indicate that you 'strongly agree', '3' to indicate that you are 'neutral', and '1' to indicate that you 'strongly disagree'. Use 'NA', 'Not Applicable', when the tasks you performed did not elicit this outcome. Credit may be given for including this worksheet with your lab report; however, your responses will not be graded. They are for your instructor's information only.

**Table \_\_: Self-Assessment of Outcomes for Simulation Lab 1: Half Adder, Increment & Two's Complement Circuit.**

<b>After completing the assigned tasks and report, I am able to:</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>NA</b>
Describe the truth tables that characterize the addition of two single bit numbers.						
Write the Boolean algebraic expressions that characterize the sum and carry functions for the half adder.						
Build and debug a simulation of a circuit that will perform the half-adder operation.						
Build and debug a simulation of a circuit that will perform the 4-bit increment operation.						
Build and debug a simulation of a circuit that will perform the 2's-complement operation.						
Backup my data files.						

Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

# SIMULATION LAB 2: 4-BIT FULL ADDER, MULTIPLEXER & DECODER

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Use LogicWorks™.
- Use Karnaugh maps.
- Have completed [Simulation Lab 1: Half Adder, Increment & Two's Complement Circuit](#).

**Equipment:** Personal computer and LogicWorks™.

**Objectives:** In this experiment, you will build and debug combinational logic subcircuits that perform arithmetic operations and data routing using LogicWorks™.

**Outcomes:** When you have completed the tasks in this experiment you will be able to design, build, test, debug, and imbed in a subcircuit, the following:

- A 1-bit full adder.
- A 4-bit full adder.
- A 2-to-1 multiplexer.
- A 4-bit, 2-to-1 multiplexer.
- A 1-to-2 decoder.
- A 2-to-4 decoder.
- A 4-to-16 decoder.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab, it is recommended that you use a task-oriented organization for your lab report. (If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Task-Oriented Report Writing Guidelines](#).)

---

## Introduction

In [Simulation Lab 1](#), you developed the increment circuit (which you will use in the program-counter portion of the microprocessor design) and the two's-complement circuit (which you will use as part of the arithmetic and logic unit [ALU] of the microprocessor design.) In this laboratory exercise you will continue constructing modules that will eventually be used in assembling the microprocessor. Our concern in this laboratory exercise is with circuits that can perform binary addition (the adder) and with circuits that control the flow of data through our system (the multiplexer and decoder). You will eventually use the data-flow-control circuits you create in this lab exercise (a 2-to-1 multiplexer, a 4-to-2 multiplexer, and a 1-to-2 decoder) to make the microprocessor self-capable of routing data to appropriate locations. The binary-addition circuitry you will create (which is a 4-bit full adder) will contribute another piece to the ALU. Each circuit you build will be modularized by imbedding it in a subcircuit and these modules will be used in subsequent labs to create more complex circuits. Using modules to create more complex modules is a powerful strategy that we will use throughout these simulation laboratory exercises.

As we progress through these laboratory exercises, you will notice that the burden of circuit design will be shifted from this manual to you. Instead of giving you a circuit schematic and asking you to build, debug and modularize it, we will slowly begin to ask you to do more of the design work yourself. Any design work you are asked to do will be well within your capabilities and will be supported by the material in this laboratory manual, your textbook, and your innate ability to recognize patterns.

In the next section we will look at a systematic method for designing a half adder. Using the methodology described in that section you will be asked to design, build, and debug a full adder. If you are familiar with canonical methods for designing circuits, skip the next section. (Skip the next section if you have completed [Hardware Lab 1: Debugging a Half and Full Adder](#).)

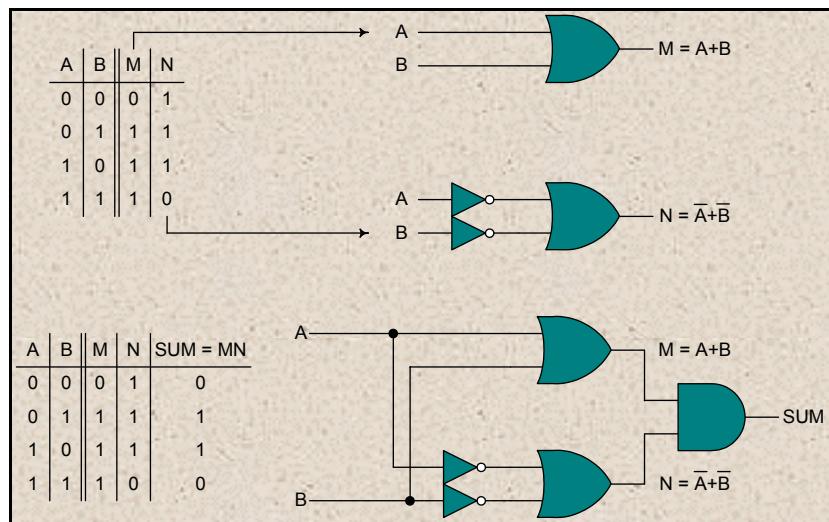
## Design of a Half-Adder Using Canonical Product of Sums

In [Simulation Lab 1](#), we used the canonical Sum-of-Products (SOP) technique for creating Boolean algebraic expressions and circuit schematics for a half adder. In this lab exercise, we will review the canonical Product-of-Sums (POS) method for creating the Boolean algebraic expressions for a half adder.

A	B	CRY	SUM
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 2-1. Half-adder truth table.

In Simulation Lab 1, we created the truth table for a half adder as shown in [Figure 2-1](#), where SUM and CRY represented the sum and carry outputs, respectively. In Simulation Lab 1, we used this truth table to create the SOP realization of the half adder. To design the half adder using a POS form, let's define two auxiliary functions, M and N, whose values are 0 for one and only one of the input combinations that cause the SUM function to be 0. These auxiliary functions are shown in [Figure 2-2](#). If we AND M and N together we get the result  $M \bullet N$  shown in the table of Figure 2-2. By inspection we can see that  $SUM = M \bullet N$ . This means that if we can realize the M and N functions, we can realize the SUM function as the AND combination of M and N. By inspection, we can see that we want M to be 0 when and only when  $A=0$  and  $B=0$ . Clearly, this will be the case if we set  $M = A + B$ . The function M can then be realized using the circuit shown in Figure 2-2. Similarly, the function N must equal 0 when and only when  $A=1$ , and  $B=1$ . By inspection, the function  $N = \bar{A} + \bar{B}$  meets this requirement and can be realized using the circuit of [Figure 2-2](#). If we connect the functions M and N to the inputs of a 2-input AND gate of Figure 2-2, we get the SUM function that we want. This circuit is the hardware equivalent to the Boolean algebra expression  $SUM = (A + B) \bullet (\bar{A} + \bar{B})$ .



**Figure 2-2.** POS implementation of SUM for 1-bit half adder.

Using this same technique, the POS implementation of the CRY function can be created as shown in [Figure 2-3](#). (Verify for yourself that the circuit of Figure 2-3 also will realize the CRY function.) Clearly the POS form requires many more logic gates and connections than the SOP form shown in [Figure 2-4](#). (See [Simulation Lab 1: Half Adder, Increment & Two's Complement](#) if you have forgotten how the circuit of Figure 2-4 was created.) Which would you rather build in the lab, the SOP or POS form for the CRY? Most of us would rather build the form that takes less time to construct and less time to debug. Industry

likewise prefers the form with the fewest gates<sup>1</sup> and circuits with fewer gate connections because it can be made smaller. Circuits with fewer connections and gates have fewer pieces that can become defective. Consequently, these circuits will perform more reliably.

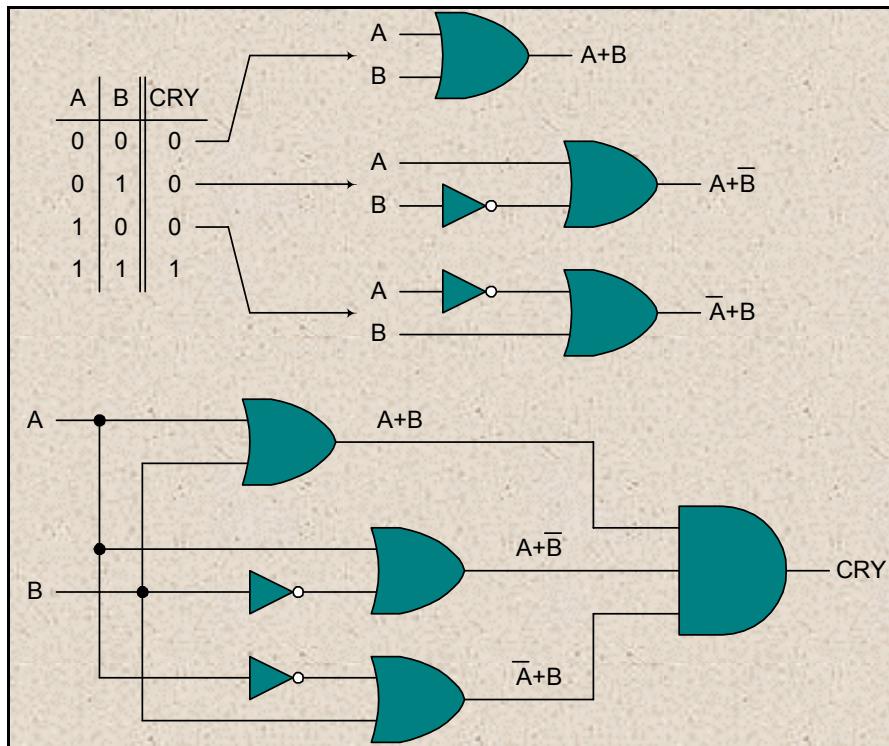


Figure 2-3. POS implementation of CRY for 1-bit half adder.

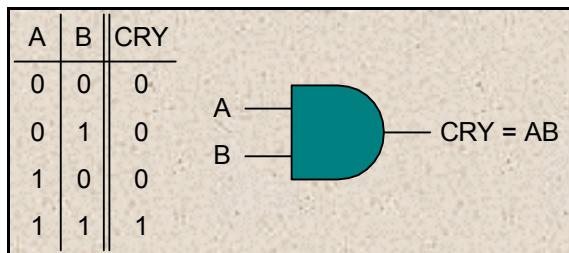


Figure 2-4. SOP implementation of CRY for 1-bit half adder.

If we wish to design a circuit using a minimal number of gates, is there a way of knowing *a priori* whether the SOP or POS canonical forms will yield the more minimal design? The answer is yes. If the function column of the truth table has fewer 1's than zeros, a canonical SOP form will be more minimal. If the function column of the truth table has the same number of 1's and 0's, both canonical SOP and POS forms require the same number of gates; otherwise the POS form is more minimal.

<sup>1</sup> Circuits are not always built using the realization that requires the fewest gates for a variety of reasons. Sometimes circuits are constructed using one gate type, and such circuits will often require more gates. Sometimes circuits are designed so that all of their

Your textbook describes ways of obtaining a minimal two-level<sup>2</sup> (non-canonical) realization for logic functions. There is no simple way of knowing *a priori* whether the minimal (non-canonical) SOP or POS form will require less logic.

## DeMorgan's Laws

DeMorgan's laws, shown in Figure 2-5, are two of the most useful laws in Boolean algebra for building circuits using only one gate type. Also shown in Figure 2-5 are the schematic equivalents to DeMorgan's laws, which we refer to as 'gate equivalency' rules.

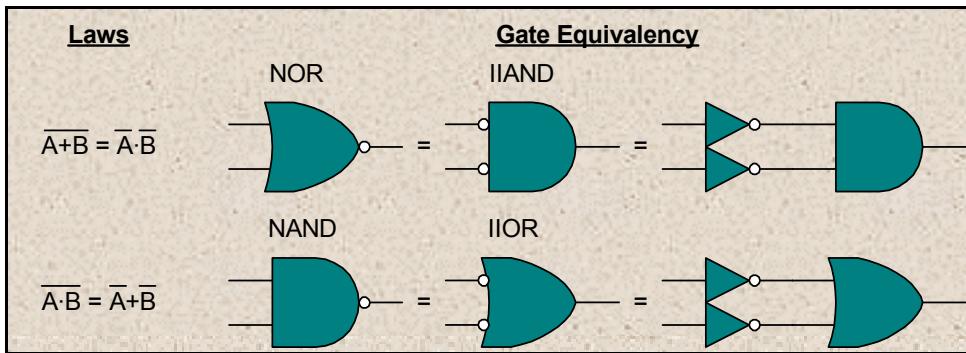


Figure 2-5. DeMorgan's laws and gate equivalency.

What the *gate equivalency* rules mean are:

- The performance of a NOR gate is *equivalent* to the performance of an AND gate with inverted inputs (shown in Figure 2-5 as an AND gate with bubbles on its inputs). This means we can replace the traditional NOR gate schematic symbol with an 'inverted inputs AND gate' (or IIAND), symbol (and *vice versa*) when such a change in the schematic, facilitates our understanding of the circuit. (We will see shortly how this symbol change can be beneficial.)
- Similarly, the performance of a NAND gate is *equivalent* to the performance of an OR gate with inverted inputs (shown in Figure 2-5 as an OR gate with bubbles on its inputs). This means we can replace the traditional NAND gate schematic symbol with an 'inverted inputs OR gate' (or IIOR) symbol (and *vice versa*) when such a change in the schematic, facilitates our understanding of the circuit.

Why are DeMorgan's laws and the gate equivalency rules so important? When we are manipulating Boolean algebraic expressions algebraically, DeMorgan's laws allow us to reduce them to a simpler form. When we are constructing schematic diagrams, gate equivalency rules allow us to easily sketch digital

capabilities are testable; this may require more gates. Also, circuit designs that avoid glitches, (see your textbook for the technical definition of glitch) require extra gates.

<sup>2</sup> All canonical POS and SOP realizations are two-level realizations; that is, the input signal must travel through two AND/OR gates before its effect on the output can be detected. (The presence or absence of inverters is ignored in this definition.)

circuits that can be built using *only one type* of gate. For example, we can build all of our circuits using only NOR or NAND gates. Sometimes this reduces the number of IC chips required to physically build a circuit with hardware. Also, at the transistor level, NAND and NOR gate are smaller (than AND's and OR's) and operate with less delay time. NAND's and NOR's are smaller and faster because, at the transistor level, AND gates and OR gates are built by adding inverters to the output stage of NAND and NOR gates respectively<sup>3</sup>.

## Designing Combinational Logic Circuits Using NOR/NOR Logic

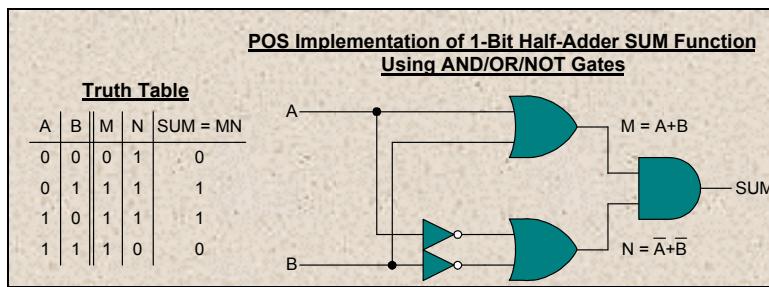
There are general rules that will allow you to realize any Boolean algebraic expression using only NAND or using only NOR gates. This set of general rules can be reduced to a very simple set if we restrict our interest to two special cases: implementing SOP forms using NAND gates, implementing POS forms using NOR gates.

In the next laboratory task you will be asked to implement a POS form using only NOR gates. The rules for performing this design are:

- Construct a traditional POS schematic for the circuit using AND/OR/NOT gates,
- In this schematic, replace all AND and OR gates with NOR gates,

And you're done!

Let's see why this works using an example. Consider the POS implementation of the SUM function of a half-adder using AND/OR/NOT gates as shown in Figure 2-6. Let's replace the OR gates of Figure 2-6 with NOR gates using the NOR-form schematic symbol and let's replace the AND gate with a NOR gate using the IIAND schematic symbol. Making these changes results in the schematic of Figure 2-7.

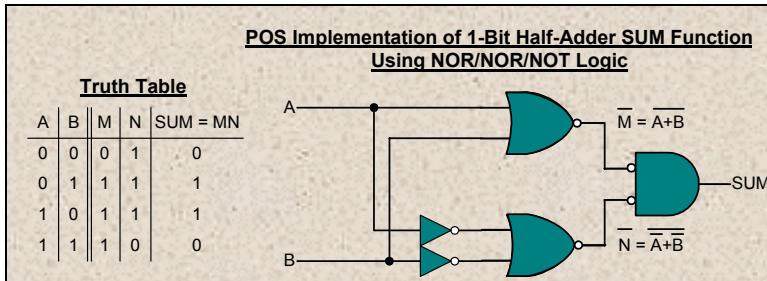


**Figure 2-6.** POS schematic for the SUM function of a half adder.

Recalling the Boolean algebraic law which states that double complementing a variable is equivalent to not complementing the variable (i.e.,  $\bar{\bar{x}} = x$ ), we see that the negation at the output of the NOR-form schematic symbol in Figure 2-7 is effectively canceled by the negation at the input of the IIAND schematic

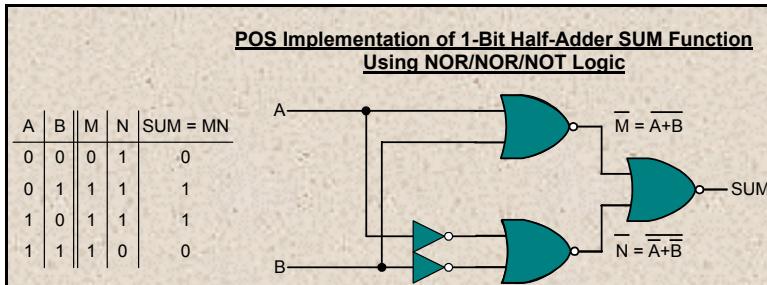
<sup>3</sup> This is true for both CMOS (Complementary Metal Oxide Semiconductor) and TTL (Transistor-Transistor Logic) digital logic families.

symbol; this leaves us effectively with what we started with: AND/OR logic – but constructed using NOR gates.



**Figure 2-7.** POS schematic for the SUM function using only NOR gates and gate-equivalency rules.

Once we understand how this works, we need not use the IIAND schematic symbol for a NOR gate and may draw the schematic diagram that we will implement, Figure 2-8, directly.



**Figure 2-8.** POS schematic for the SUM function using only NOR gates and NOR gate symbols.

### Task 2-1: Design a Full Adder Using NOR/NOR Logic

Using the full-adder function definition table, Table 2-1, write down the **canonical POS** expressions for the  $C_{out}$  and SUM functions of a full adder. Using **these POS canonical** expressions, build, test and debug the circuits that realize the  $C_{out}$  and SUM functions using **only NOR/NOR logic** with LogicWorks™. (Remember: you will need to design two circuits: one for the SUM function and one for the  $C_{out}$  function. Both functions should share the same A, B and  $C_{in}$  inputs. Where you need the complement on a variable, you may use inverters, i.e., NOT gates, rather than using NOR gates connected as inverters.) Record the results of your validation tests in the form of a truth table in your report.

**Table 2-1. Full Adder Function Definition Table.**

Address			Functions	
C <sub>in</sub>	A	B	SUM	C <sub>out</sub>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	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

### Design of a Full Adder Using a Minimal POS Form

If you have studied Karnaugh maps in the lecture portion of your course, you will be able to derive a minimum **POS** form for the full adder. Refer to your textbook or prove, using the Karnaugh map in Figure 2-9 that the C<sub>out</sub> function is given by:

$$C_{out} = (A + B) \bullet (B + C_{in}) \bullet (C_{in} + A)$$

		C <sub>OUT</sub>				
		AB	00	01	11	10
C <sub>in</sub>	AB	00	01	11	10	
	0	0	0	1	0	
1	0	1	1	1	1	

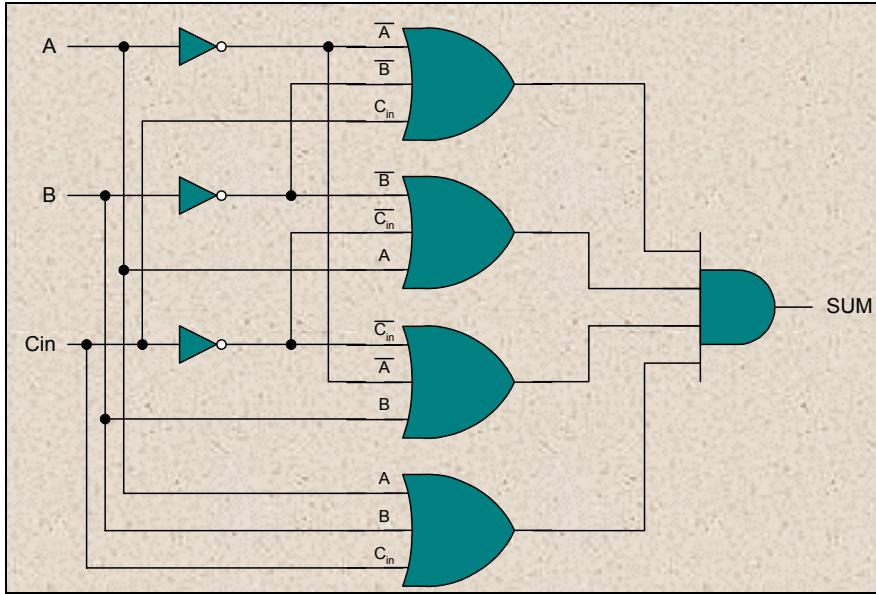
**Figure 2-9.** Karnaugh map for C<sub>out</sub> function.

		SUM				
		AB	00	01	11	10
C <sub>in</sub>	AB	00	01	11	10	
	0	0	1	0	1	
1	1	0	0	1	0	

**Figure 2-10.** Karnaugh map for SUM function.

If you look in [Figure 2-10](#) at the Karnaugh map attempt to obtain a minimum POS expression for the SUM function, you see that we obtain no reduction beyond the canonical POS form:

$$SUM = (A + B + C_{in}) \bullet (A + \bar{B} + \bar{C}_{in}) \bullet (\bar{A} + \bar{B} + C_{in}) \bullet (\bar{A} + B + \bar{C}_{in})$$

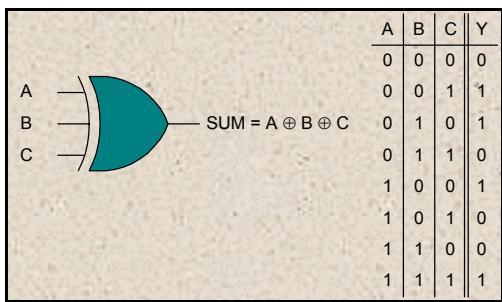


**Figure 2-11.** Minimal and canonical POS implementation of SUM function.

Such a realization for the SUM function requires four 3-input OR gates and one 4-input AND gate, and a few inverters as shown [Figure 2-11](#). Fortunately, there is a more economical method for implementing the SUM function. Refer again to [Table 2-1](#) and make the following observation: the SUM function is 1 **if and only if** the row address (i.e., input variable values) contains an odd number of 1's. This is precisely the function performed by an exclusive OR (XOR) gate (shown in [Figure 2-12](#)) , provided we supply the XOR inputs with variables that represent each bit of the address; hence an economical way to implement the SUM function is to use the XOR operator, i.e.,

$$\text{SUM} = A \oplus B \oplus C_{in}.$$

The SUM function could be built using either one 3-input XOR gate (see Figure 2-12) or two 2-input XOR gates. (To see this latter implementation, see [Hardware Lab 1: Debugging a Half and Full Adder](#).)



**Figure 2-12.** 3-Input XOR schematic symbol and truth table.

Using the 3-input XOR gate to realize the SUM function and the minimal POS form to realize the  $C_{out}$  function, the combinational logic that could be used to realize a full adder is shown in Figure 2-13.

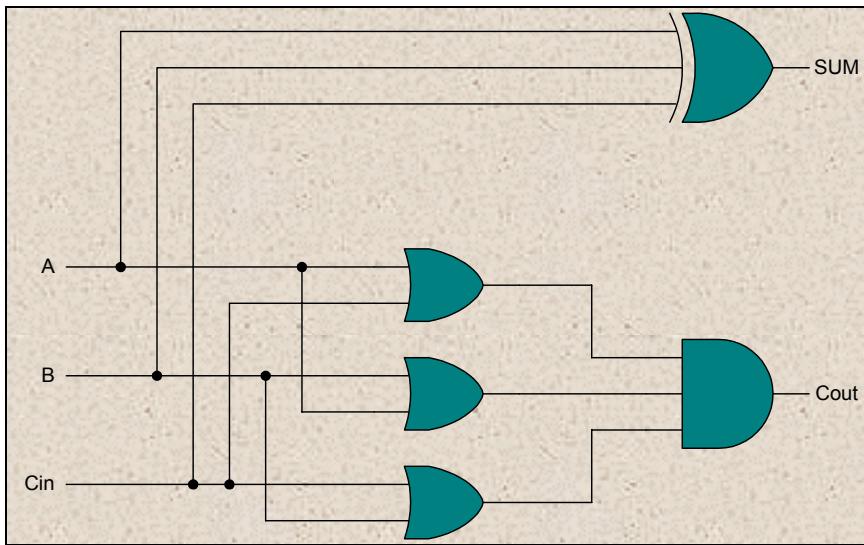


Figure 2-13. POS implementation of 1-bit full adder.

### Task 2-2: Build, Debug and Test a 1-Bit Full Adder

Build, debug and test a 1-bit full-adder circuit. Use NOR/NOR logic to implement your minimal POS form for the  $C_{out}$  function (use Figure 2-13 only as a guide) and construct the SUM function using either 2, 2-input XOR gates or a 4 input XOR; LogicWorks™ has no 3-input XOR gate. (If you use a 4-input XOR, give consideration to the proper way to connect the unused input.) Imbed your 1-bit full adder in a subcircuit (see [Figure 2-14](#)), label the subcircuit FA\_1, and add it to your library. Test your subcircuit. Record the results of these tests (in the form of a truth table) in your laboratory notebook and include the table in your report.

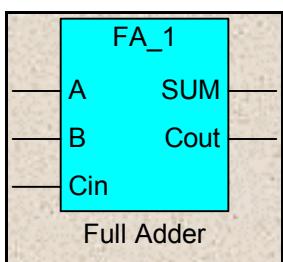


Figure 2-14. Subcircuit symbol for a 1-bit full adder.

### A 2-Bit Full Adder

The 1-bit full-adder subcircuit you constructed in the previous task is the kernel around which we will build the adder portion of the ALU for the microprocessor we are designing. The ALU we will design will operate on 4-bit numbers, so we need to extend our adder from a 1-bit design to a 4-bit design. A 2-bit design will be described here; it will be left to you to extend this to a 4-bit design.

Consider the addition of two numbers A, and B, each of which has two bits of precision. Let the postscripts 0 and 1 refer to the least and most significant bits of numbers, respectively. The addition of A0 and B0 yields a sum Y0 and carry C0, as shown in Figure 2-15. The carry, C0, is added to the sum of the most significant bits, A1 and B1, to yield a sum bit, Y1 and a carry, C<sub>out</sub>.

Using the 1-bit full-adder subcircuit you designed in the previous task, a 2-bit full adder can be constructed as shown in Figure 2-16. Verify for yourself that the 2-bit adder schematic will perform correctly. Note that the 2-bit adder circuit was designed for the possibility of a carry input. (If the carry input to the adder processing the least significant bit is not used, it may be permanently connected to ground.)

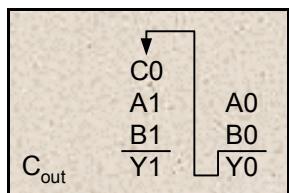


Figure 2-15. Two-bit number addition.

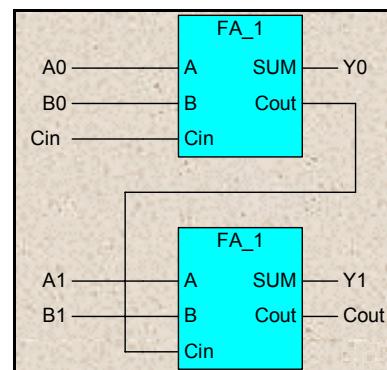


Figure 2-16. 2-bit full adder.

### Task 2-3: Design, Build and Test a 4-Bit Full Adder

Using Figure 2-16 as a guide, design a 4-bit full adder. The 4-bit full adder should accept two 4-bit numbers and a carry as input, and give one 4-bit sum and a 1-bit carry as output. Build, test and debug the 4-bit full adder. Test the circuit using hex keyboards and a hex display. Include the test results in the form of a truth table in your report. **DO NOT** test all input combinations. It is left to you to decide what constitutes a sufficient set of tests to make it likely that the 4-bit full-adder circuit is operating correctly. (One set of test inputs is not enough.) Justify in your report why successful completion of your tests proves beyond a reasonable doubt that your circuit is operating correctly. Once you are satisfied that the circuit is working correctly, imbed the 4-bit adder in a subcircuit, label it “FA\_4”, and test the subcircuit. (The subcircuit symbol we’ll use for a 4-bit full adder is shown in Figure 2-17.) Once you are satisfied that it is working correctly, add it to your library. (Remember to record the results of all of your tests for inclusion in your lab report.)

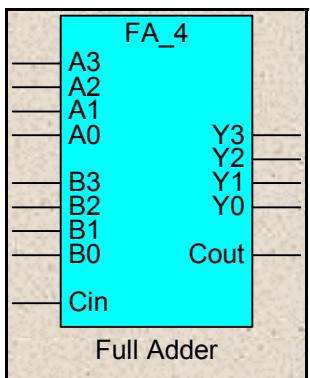


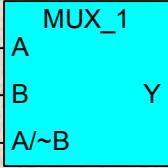
Figure 2-17. Symbol for 4-bit full adder.

## Notation

The notation used in this manual for representing complemented variables conforms with that used by most textbooks that is, the complement of A is denoted by  $\bar{A}$ . This notation is also used by many textbooks to represent a variable that is active when it is low, e.g.,  $\overline{\text{Arith}}$  is understood to be active when it is low. LogicWorks™ does not support using this over-bar notation; hence, the naming convention we have (and will) use for active low signals is an alpha-numeric character string preceded by the character string: ‘/~’. Using this notation, the active low signal  $\overline{\text{Arith}}$  would be represented by ‘/~Arith’. Where a signal performs one operation when high (operation Y) and another operation when it is low (Z) we will label the signal ‘Y/~Z’. Using this notation, the complement of an active low signal is represented as:  $\overline{/ \sim \text{Arith}}$ .

## A Multiplexer

When we discuss the architecture of the microprocessor in later laboratory experiments, we will find that hardware is needed that allows the executing program to select the route along which data flows. One component that we will need to perform this data routing function is the multiplexer, or MUX. A MUX is a device that can be controlled to route one of its many input signals and to its sole output. If a MUX has  $2^n$  data inputs, then it must have n control (or select) inputs (which have  $2^n$  different possible settings) to be capable of routing each input to the output. By changing the control-input values we can select which input data stream is connected to the output. For a MUX with two data inputs, one control/select input is necessary. A truth table and symbol for the 1-bit MUX you will build is shown in Figure 2-18. The control/select input, ‘A/~B’, indicates that the output is identical to the A input when the select signal is high (1) and identical to the B input when the select signal is low (0). This MUX is called a 2-to-1 MUX because it routes one of 2 inputs to the one output under select line control.



A/~B	A	B	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2-18. Truth table and symbol for 1-bit MUX.

Y				
AB	00	01	11	10
A/~B	0	1	1	0
1	0	0	1	1

Figure 2-19. Karnaugh map for a 2-1 MUX.

### Task 2-4: Design, Build and Test a MUX Using NOR/NOR Logic

Using the Karnaugh Map methods you learned in class, design, build and test a **NOR/NOR** implementation of the 1-bit MUX. You will want to base your design on the POS form of the equations that define the MUX output. You may wish to use the Karnaugh Map of [Figure 2-19](#) as an aid in getting the form of the equation that you want. Test the circuit and record the results as a truth table in your report. Once you are convinced that the circuit is working correctly, imbed it in a subcircuit, test the subcircuit, label it “MUX\_1”, and add it to your library.

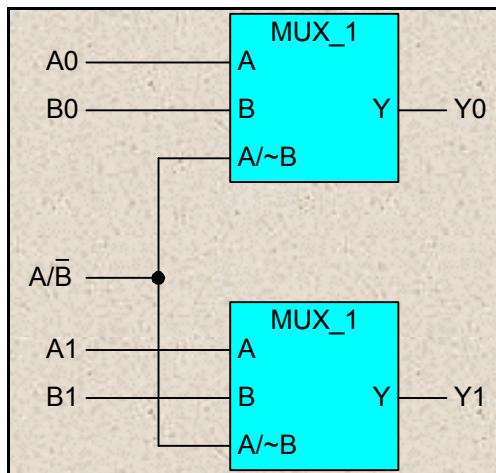


Figure 2-20. 2-bit, 2-to-1 MUX.

### Task 2-5: Build a 2-Input 4-Bit Multiplexer

Because our microprocessor operates on 4-bit numbers, it will be necessary to construct a 4-bit, 2-to-1 MUX. The 4-bit MUX should use a single control/select line to select one of two 4-bit numbers and the selected 4-bit number should appear on the output of the MUX. (A 2-bit 2-to-1 MUX is shown in Figure

2-20. Expand on this figure to design your 4-bit MUX.) Build and test the 4-bit MUX. Once you are satisfied that it is working correctly, imbed it in a subcircuit, label the subcircuit “MUX\_4” (Figure 2-21), test the subcircuit, and add it to your library.

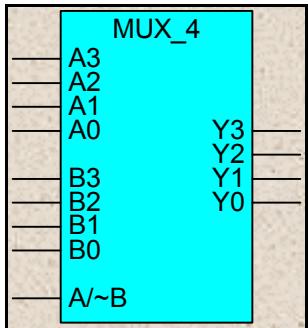


Figure 2-21. 4-bit, 2-to-1 MUX.

### Demultiplexer/Decoder

The other piece of hardware our microprocessor will need to perform data routing is a demultiplexer, or a demux. A demux, in some sense, performs the inverse operation of the MUX; it routes one input to one of many outputs. A demux has one input data line, n control inputs, and  $2^n$  output lines. Each of the  $2^n$  different settings of the control lines causes the input data to be routed to a different output line. The truth table and subcircuit symbol for the 1-bit demux you will build is shown in Figure 2-22. This demux is called a 1-to-2 demux because it uses one control line ( $Y/\sim Z$ ) to route the input data (D) to one of two outputs (Y or Z). (Using this notation, a demux with 16 output lines would be called a 4-to-16 demux, because we need four control/select lines with  $2^4=16$  unique settings to allow us to route the input to each of 16 output lines<sup>4</sup>.) When the  $Y/\sim Z$  input is high, the data input, D, should appear at the Y output and D should appear at the Z output when the  $Y/\sim Z$  input is low.

We could design each of the Y and Z output functions of the demux of Figure 2-22 using a 2-variable Karnaugh map (using input variables D,  $Y/\sim Z$ ). For large demultiplexers, the Karnaugh map approach would work as well, but it becomes a bit unwieldy; for a 4-to-16 MUX, we would need to specify 16 outputs for each of the 32 input combinations – a total of 512 values! (There are 32 input combinations because the device has a total of 5 input lines; four of these are control input lines, and one is a data input line.)

<sup>4</sup> The notation we are using here is not unique. It is also common in the literature to find a demultiplexer with 16 output lines called a 1-of-16 demultiplexer.

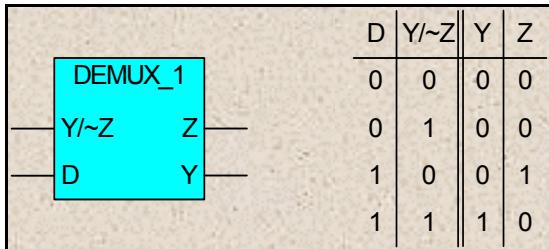


Figure 2-22. Truth table and symbol for 1-bit DEMUX.

Let's look at a simpler approach for designing the demultiplexer that relies on our understanding of canonical forms. For a 1-to-2 demux, we want the Z output to be 1 ONLY when the input D is 1 and the select line (call it A0 for the moment instead of Y/~Z<sup>5</sup>) is 0; hence the Z function is given by:

$$Z = D \bullet A0$$

Similarly, we want the output Y to be 1 ONLY when both the input D and select input, A0, are 1; hence Y is given by:

$$Y = D \bullet A0$$

### Task 2-6: Build and Test a 1-to-2 Demultiplexer Using NOR/NOR Logic

Build, and test the 1-bit, 1-to-2 demux using only NOR/NOR logic. (Save the circuit before you imbed it in a subcircuit.) Imbed your circuit in a subcircuit labeled “DEMUX\_1” (using the circuit symbol of Figure 2-22), test the subcircuit, and add it to your library. Hint: Remember the gate equivalency rules and think of a NOR gate as an IIAND gate. (Where you need the complement on a variable or a function, you may use inverters, i.e., NOT gates, rather than using NOR gates connected as inverters.)

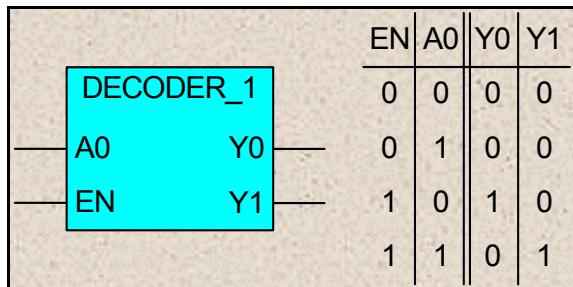
#### Decoder Versus Demultiplexer

The device you built above we refer to as a demux. This device may also be called a decoder. What is the difference? There is no difference in the construction of the device. What we call it, either a decoder or a demultiplexer, depends on how we use it. When we use the device to route data, we call it a demultiplexer. When the input bit (D) is thought of as a control signal (active high in our design), and when the output line is being used as a control input to another device, we call the device a decoder. Because we think of the decoder input data bit as an enabling signal (rather than a data bit), the notation we will use for a decoder is different from that we use for a demux. Let's repackage our demux as a decoder using a notation that is more traditional for a decoder.

<sup>5</sup> We have changed the notation from Y/~Z to A0 to facilitate writing the output equations. The notation Y/~Z will be used in the subcircuit symbol, as it is more meaningful when used in a schematic diagram.

### Task 2-7: Repackage the 1-to-2 Demux as a 1-to-2 Decoder

Open the 1-to-2 demux circuit (not subcircuit) you created in [Task 2-6](#). Imbed this circuit in a subcircuit using the notation shown in Figure 2-23. Test the subcircuit and add it to your library. Note that when the enable input to the decoder is inactive (i.e., 0), all outputs are inactive (i.e., 0), or disabled. (It is easy to make a mistake here if you are not careful. Be sure to assign the demux line ‘D’ to the EN input and the demux line ‘Y/~Z’ to the A0 input of your decoder subcircuit.)



**Figure 2-23.** Symbol and truth table for 1-bit decoder.

### Building a 2-to-4 Decoder

The microprocessor you will build will need a larger decoder than that constructed above. Let's start by building a 2-to-4 decoder. A 2-to-4 decoder has the function definition table shown in Figure 2-24. The postscript notation used in the decoder symbol and truth table is almost universal: the output line that becomes active is labeled with the decimal equivalent of the applied binary address (with A1 being the most significant bit - MSB). This means that if we apply the address (A1, A0) = 10B = 2D, the Y2 output line should become active. To design this decoder, we will use the same approach as used for the 1-to-2 decoder (demux).

Let's first design the output function Y0. We want the output line Y0 to be equal to the enable signal (EN) **only** when the applied address is (A1, A0) = 00B; hence Y0 is given by,

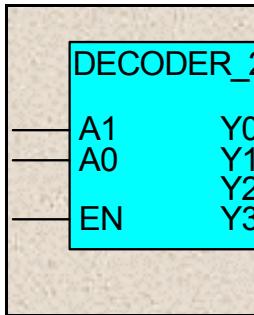
$$Y0 = EN \bullet \overline{A1} \bullet \overline{A0}$$

Similarly, we want Y1, Y2, and Y3 to be equal to EN **only** when the decimal equivalent of the applied address is equal to the respective subscript, i.e.,

$$Y1 = EN \bullet \overline{A1} \bullet A0$$

$$Y2 = EN \bullet A1 \bullet \overline{A0}$$

$$Y3 = EN \bullet A1 \bullet A0.$$



	A1	A0	Y0	Y1	Y2	Y3
	0	0	EN	0	0	0
	0	1	0	EN	0	0
	1	0	0	0	EN	0
	1	1	0	0	0	EN

Figure 2-24. Function definition table and symbol for 1-bit 2-to-4 decoder.

### Task 2-8: Build and Test a 2-to-4 Decoder Using NOR/NOR Logic

Build, and test the 1-bit, 2-to-4 decoder using only **NOR/NOR** logic. Imbed your circuit in a subcircuit labeled “DECODER\_2”, test the subcircuit, and add it to your library.

#### Building Larger Decoders

An alternate scheme for building the 2-to-4 decoder using primitive logic gates is to build the decoder using the 1-to-2 decoders built in Task 2-7. Justify to yourself that the circuit of Figure 2-25 will function as a 2-to-4 decoder and that the notation used is consistent with the convention of having the subscript of the activated output line be the decimal equivalent of the applied binary address. Also prove to yourself that when the EN input line is inactive (i.e., 0), all output lines will be inactive.

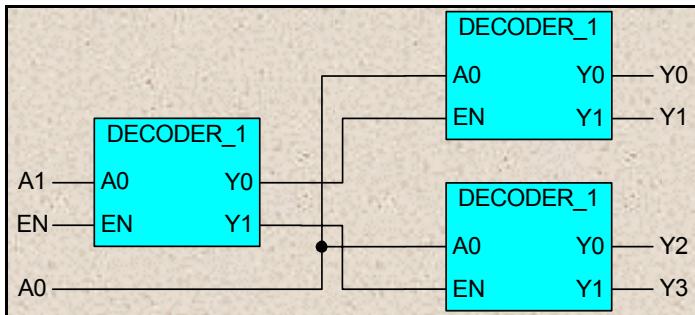


Figure 2-25. Schematic for a 2-to-4 decoder constructed from 1-to-2 decoders / demultiplexers.

### Task 2-9: Design, Build & Test a 4-to-16 Decoder Using 2-to-4 Decoders

Using the technique illustrated in Figure 2-25, design, build, and test a 4-to-16 decoder using only the 2-to-4 decoder subcircuits you constructed in [Task 2-8](#). Imbed your circuit in a subcircuit labeled “DECODER\_4” (see Figure 2-26 for the subcircuit symbol to use), test the subcircuit, and add it to your library. Be sure that your decoder’s input/output characteristics conform to the conventional notation: the output line that becomes active is labeled with the decimal equivalent of the applied binary address, as

shown in Figure 2-26 where A3 is the MSB and A0 is the LSB. For example, if  $(A_3, A_2, A_1, A_0) = 1110$  (fourteen decimal), then the Y14 output line should be active and all other lines inactive.)

**Figure 2-26.** Subcircuit symbol and I/O definition for a 4-to-16 decoder.

## **Task 2-10: Backup Your Files**

Take a moment and back up your libraries and circuit files on a new floppy disk and mark it ‘SimLab#2’. Do this using a different floppy disk after every lab and you just may save yourself much unnecessary work.

# SIMULATION LAB 2: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 2-1: Design a Full Adder Using NOR/NOR		
Task 2-2: Build, Debug and Test a 1-Bit Full Adder		
Task 2-3: Design, Build and Test a 4-Bit Full Adder		
Task 2-4: Design, Build and Test a MUX Using NOR/NOR Logic		
Task 2-5: Build a 2-Input 4-Bit Multiplexer		
Task 2-6: Build and Test a 1-to-2 Demultiplexer Using NOR/NOR		
Task 2-7: Repackage the 1-to-2 Demux as a 1-to-2 Decoder		
Task 2-8: Build and Test a 2-to-4 Decoder Using NOR/NOR		
Task 2-9: Design, Build & Test a 4-to-16 Decoder Using 2-to-4		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>		
Points Lost		
Late Lab		
Lab Score		

## Report Writing Reminders:

<i>Caveat emptor:</i> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used
• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.
• Include lab data sheets for all labs. Use lab data sheets to document the lab as you perform it. It saves a lot of time when writing the report.
• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.
• <b>Label</b> all figures or circuits. (E.g. Figure 1. Schematic of . . ., Figure 2 . . ., etc.)

- Refer in the text of your report to all circuits or figures that you include in the body of your report.
- Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

Put 'X's' in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use '5' to indicate that you 'strongly agree', '3' to indicate that you are 'neutral', and '1' to indicate that you 'strongly disagree'. Use 'NA', 'Not Applicable', when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your responses will not be graded. They are for your instructor's information only.

Table \_\_: Self-Assessment of Outcomes for Simulation Lab 2: 4-Bit Full Adder, Multiplexer & Decoder.

<b>After completing the assigned tasks and report, I am able to design, build, test, debug, and imbed in a subcircuit:</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>NA</b>
A 1-bit full adder.						
A 4-bit full adder.						
A 2-to-1 multiplexer.						
A 4-bit, 2-to-1 multiplexer.						
A 1-to-2 decoder.						
A 2-to-4 decoder.						
A 4-to-16 decoder.						

Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

# SIMULATION LAB 3: ARITHMETIC AND LOGIC UNIT

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Use LogicWorks™.
- Perform a 1's complement operation on a multi-bit operand.
- Have completed [Simulation Lab 1: Half Adder, Increment & Two's Complement Circuit](#).
- Have completed [Simulation Lab 2: 4-Bit Full Adder, Multiplexer & Decoder](#).

**Equipment:** Personal computer and LogicWorks™.

**Objectives:** In this laboratory exercise you will learn how to construct an arithmetic and logic unit (ALU) using subcircuits you built with LogicWorks™ in simulation labs 1 and 2.

**Outcomes:** When you have completed the tasks in this experiment you will be able to:

- Build, test, debug, and imbed in a subcircuit, an elementary arithmetic and logic unit (ALU).
- Describe the arithmetic and logical operations of which the ALU is capable.
- Describe the input control line values that correspond to each arithmetic and logical operation of the ALU.
- Write a top-down description of a complex logic circuit.
- Calculate the approximate propagation delay time of a combinational-logic circuit.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab, it is recommended that you use a top-down format when organizing your lab report. (If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Top Down Report Writing Guidelines](#).)

---

## Introduction

In simulation labs 1 and 2 you built combinational logic circuits of increasing complexity. In this laboratory exercise we will build upon these modules and combine them to form a more complex combinational logic subcircuit: the arithmetic and logic unit (ALU). The ALU in a microprocessor is the unit that performs all of

the arithmetic operations (such as add, subtract, negate, etc.) and all of the logical operations (such as 1's complement, AND, OR, etc.) Because this is an introductory class, we will limit the number of operations our ALU will perform so that we can limit the complexity of the design. Conceptually, however, the ALU we will design will be no different from the ALU in the personal computer you use for performing these digital-logic simulations.

Our ALU, like all ALU's, will be a combinational logic circuit. It will have two data input ports (each of which will accept a 4-bit binary data values<sup>1</sup>) and it will have a carry input. These data values are known as operands. Our ALU will be capable of operating on either one operand (e.g., performing a 1's complement operation), or two operands (e.g., performing the sum, A+B) and will produce a 4-bit result (plus a possible carry). The ALU will be controlled by a set of input control signals. These input control signals will determine which operation the ALU will perform.

In this laboratory exercise you will construct the ALU<sup>2</sup>. It will be left as an exercise at the end of this lab to create a table that lists the operations the ALU performs for each set of input control signals. As you work through this laboratory assignment, if you focus on how each subcircuit works, and how they interact, then you will find that constructing the table at the end is much easier. The modules we create will be capable of performing either logical or arithmetic operations. It may help you in organizing your understanding if you divide the ALU operations into these two types: logical and arithmetic.

## Design of a NOT/NEG Circuit

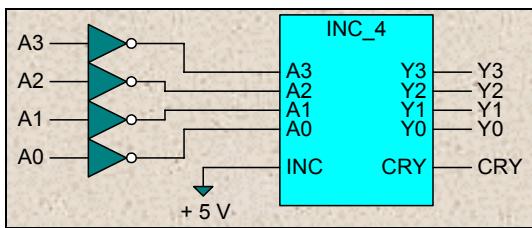
The first module we build will be capable of performing two different operations on one operand. Which operation we perform will be determined by the control signals we apply to the module. If we use one set of control signals and apply an operand, say A, we'll get a 4-bit negative (two's complement) A out. If we use a different set of control signals, our module will perform a bit-wise complement of A, giving the 1's complement of A; that is, if A = 1001, our output will be 0110.

In [Simulation Lab 1](#), you designed the two's complementing circuit shown in [Figure 3-1](#). This circuit is capable of performing only the 2's-complement operation; it has no control inputs that allow us change the function that it performs. Starting with this circuit, you will modify it as shown in [Figure 3-2](#).

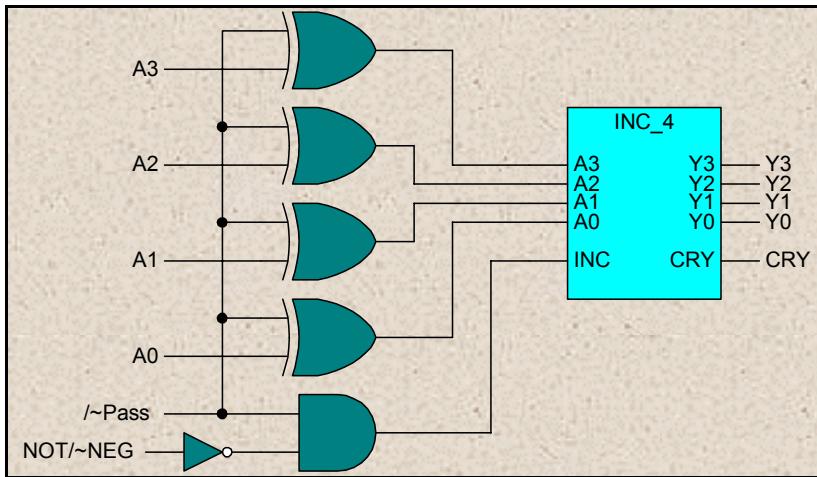
---

<sup>1</sup> We choose 4-bit (rather than 32-bit) operands because the 4-bit circuitry is much less tedious to construct and debug. This simplicity will allow you to spend your time understanding the concepts we introduce rather than tediously building 32-bit versions of the subcircuits we will need.

<sup>2</sup> For an overview of this laboratory refer to the PowerPoint lecture slides contained on this CD in the file labeled 'ALU's.'



**Figure 3-1.** Two's complement circuit.



**Figure 3-2.** NOT/NEG circuit.

The modifications shown in Figure 3-2 allow this circuit to do three things: it can perform the 2's-complement operation (i.e., negate arithmetically), perform the 1's complement (i.e., logically complement each bit, also known as performing the NOT of each bit) or allow the input argument to pass through unscathed. In Figure 3-2, two control signals have been added to effect these controls: the  $\sim\text{Pass}$  and the  $\text{NOT}/\sim\text{NEG}$ . The notation chosen for these signals communicates two things: which operation the signal controls, and whether the signal is high or low when active. The ' $\sim$ ' notation means that the signal is low (0) when active. If the ' $\sim$ ' is not present, the signal is high when active. (If you understand the notation we are using, in many cases you will be able to determine the functions being performed by each circuit simply by inspecting the notation.) The  $\sim\text{Pass}$  and  $\text{NOT}/\sim\text{NEG}$  signals work together to produce the following operations:

- $\sim\text{Pass} = 0$ . **Pass-Through Operation**

With  $\sim\text{Pass} = 0$ , the output of the AND gate in Figure 3-2 must be 0, preventing the INC\_4 subcircuit from incrementing the incoming signal. The  $\sim\text{Pass} = 0$  is also fed into each XOR gate. Recall from the definition of the XOR gate that  $0 \oplus A = A$ ; hence the A input is passed through the XOR gates unscathed and then through the incrementer unscathed. Thus with  $\sim\text{Pass} = 0$ , the A operand is passed unscathed through the NOT/~NEG circuit as indicated in Figure 3-3.

- $\sim\text{Pass} = 1, \text{NOT}/\sim\text{NEG} = 0$ .      **Arithmetic Negate Operation**

With  $\sim\text{Pass} = 1$  and  $\text{NOT}/\sim\text{NEG} = 0$ , both inputs to the AND gate are 1's; hence the output of the AND gate is high, causing the INC\_4 circuit to increment the incoming signal. The  $\sim\text{Pass} = 1$  signal is supplied as one input to each of the XOR gates. Recall from the definition of the XOR gate that  $1 \oplus A = \overline{A}$ ; that is, when one input to an XOR gate is a 1, the XOR acts on the other input as an inverter would; hence the input to the INC\_4 circuit is the bit-wise complement of A, (or  $\overline{A}$ ). Incrementing  $\overline{A}$  by 1 is the definition of the two's-complement operation. Thus this control scheme arithmetically negates the A operand as indicated in [Figure 3-3](#).

- $\sim\text{Pass} = 1, \text{NOT}/\sim\text{NEG} = 1$ .      **One's Complement Operation**

Under this control scheme the lower input to the AND gate is zero. This causes the output of the AND gate to be low. This in turn causes the INC\_4 circuit to ***pass through*** the incoming 4-bit operand signal unscathed. The incoming 4-bit operand is  $\overline{A}$  (see argument above); hence the output of the INC\_4 circuit is the bit-wise complement, or one's complement of A as indicated in [Figure 3-3](#).

### Task 3-1: Build the NOT/NEG Circuit

Build the circuit shown in Figure 3-2, imbed it in a subcircuit using the notation of Figure 3-3, test it, and add it to your library.

NOT/NEG		/~Pass	NOT/~NEG	FUNCTION
A3	Y3	0	0	PASS-THROUGH
A2	Y2	0	1	PASS-THROUGH
A1	Y1	1	0	TWO'S COMPLEMENT
A0	Y0	1	1	ONE'S COMPLEMENT
/~Pass	CRY			
NOT/~NEG				

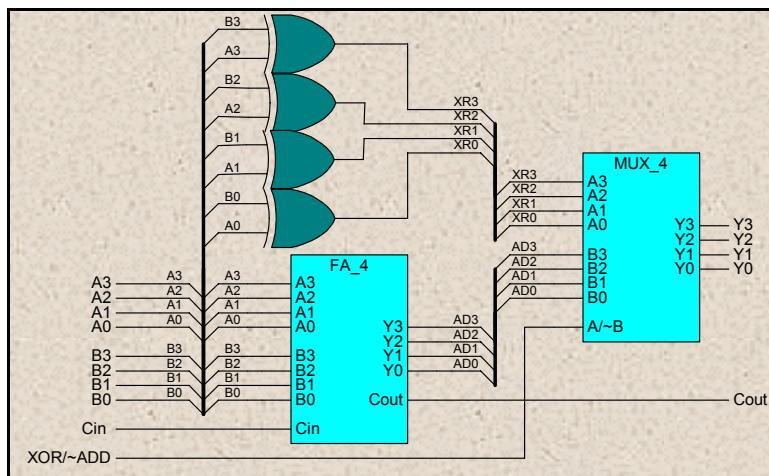
**Figure 3-3.** NOT/NEG function definition table and subcircuit symbol.

Each of the circuits we design in this lab will play a key role in your microprocessor; so it is important to select your tests to ensure that each of these circuits work correctly. Briefly mention in your lab report how you chose to test this circuit. It is not necessary to describe in your report all of the input and output values you tested; instead include a function table similar to the table shown in [Figure 3-3](#).

### Design of an XOR/ADD Circuit

The NOT/NEG circuit is a very simple version of an ALU, so simple that few would call it an ALU. Yet, it can do either an arithmetic operation, the NEGATE, or a logical operation, the NOT of 4 bits. It can also pass data through without operating on it. To justify calling our circuit an ALU, we will want to add more functions than just the three listed in [Figure 3-3](#). In particular, we would like to be able to perform an ADD

and an XOR of two 4-bit numbers – and we would like to retain the pass-through capability. In your previous laboratory exercises, you constructed all of the subcircuits we will need to build this circuit. Let's build this circuit in two stages; first let's build the XOR/ADD circuit; then let's modify the circuit to gain the pass-through capability.



**Figure 3-4.** XOR/ADD circuit.

Figure 3-4 shows the schematic of the XOR/ADD circuit. (This figure uses the LogicWorks™ bus concept and notation<sup>3</sup>. If you are not familiar with this notation look over the section titled Breakouts and Bus Lines in Simulator Tutorial: [Using LogicWorks™ 4 for Windows®](#).) This circuit accepts two 4-bit operands, A and B, as input, which are fed to the 4-bit full-adder subcircuit (FA\_4 found in your library) and to the four 2-input XOR gates. The 4-bit output of the XOR gates and adder is fed into a 2-to-1 4-bit mux (MUX\_4 found in your library). This mux is used to select which result (ADD or XOR) is routed to the output. (This is a typical example of how multiplexers are used in digital systems.) To control which result is routed to the output, the mux selector input is controlled by the *XOR/~ADD* signal. The selector signal notation is consistent with the notation described earlier: when the *XOR/~ADD* signal is low, the (arithmetic) addition result is routed to the output. When the *XOR/~ADD* signal is high, the (logical) XOR result is routed to the output.

To add the data pass-through capability to the circuit of [Figure 3-4](#), we add a 4-bit 2-to-1 mux as the output stage as shown in Figure 3-5. One input to the output-stage mux is the XOR/ADD-circuit output, the other is the A operand. The selector input of the output stage mux is controlled by the */~Pass* signal. When */~Pass* = 0, the A operand appears at the output unscathed. When */~Pass* = 1, the output function, either ADD or XOR, depends on the value of the *XOR/~ADD* signal as shown in the table of [Figure 3-6](#).

---

<sup>3</sup> Use of a bus simplifies the construction and presentation of the circuit schematic by eliminating the need to provide a unique route for each signal line.

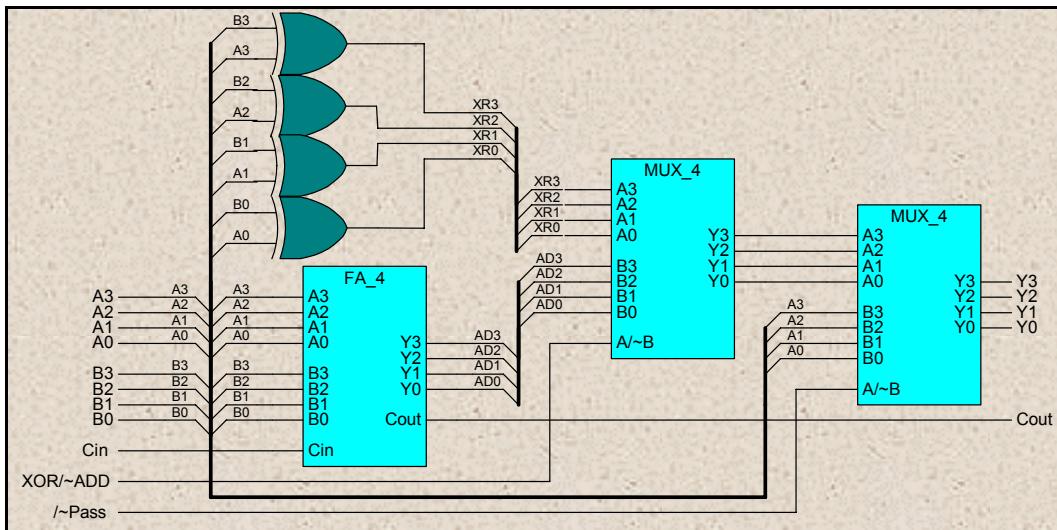


Figure 3-5. XOR/ADD circuit with pass-through.

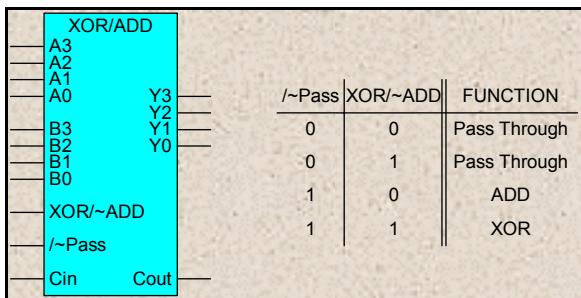


Figure 3-6. XOR/ADD function table and symbol.

### Task 3-2: Build and Test the XOR/ADD Circuit

Using the subcircuits contained in your library, build the XOR/ADD circuit with pass-through shown in Figure 3-5. Create a subcircuit for this circuit using the symbol in Figure 3-6. Test the subcircuit and add it to your library. Briefly mention the procedure you used to test it. Justify why the results of the tests you chose make it likely that your circuit functions correctly.

### Design of an ALU

The last stage in building the ALU is to combine the NOT/NEG and the XOR/ADD subcircuits as shown in [Figure 3-7](#) to create the complete ALU. This ALU contains three input control signals:

- The */~Invert* signal, when active, indicates that some type of inversion, either a one's or two's complement is taking place.
- The */~A\_Only* signal, when active, indicates that only the A operand is being acted upon by the ALU.

- The *Logic/Arith* signal operates so that when it is low, an arithmetic operation is performed. When *Logic/Arith* = 1, a logical operation is performed by the ALU.

Once this notation is understood, you may be able to fill out the function definition table for this circuit even without reference to Figure 3-7. Consider the example filled-in in Table 3-1:

- $\sim A\_Only = 1$  implies that the A and B operands are combined in some fashion.
- $\sim Invert = 1$  implies that A is not inverted (neither a one's nor a two's complement is performed.)
- $Logic/Arith = 0$  implies the operation is arithmetic.

The only two-operand, noninverting arithmetic operation of which our ALU is capable is the sum,  $A + B$ .

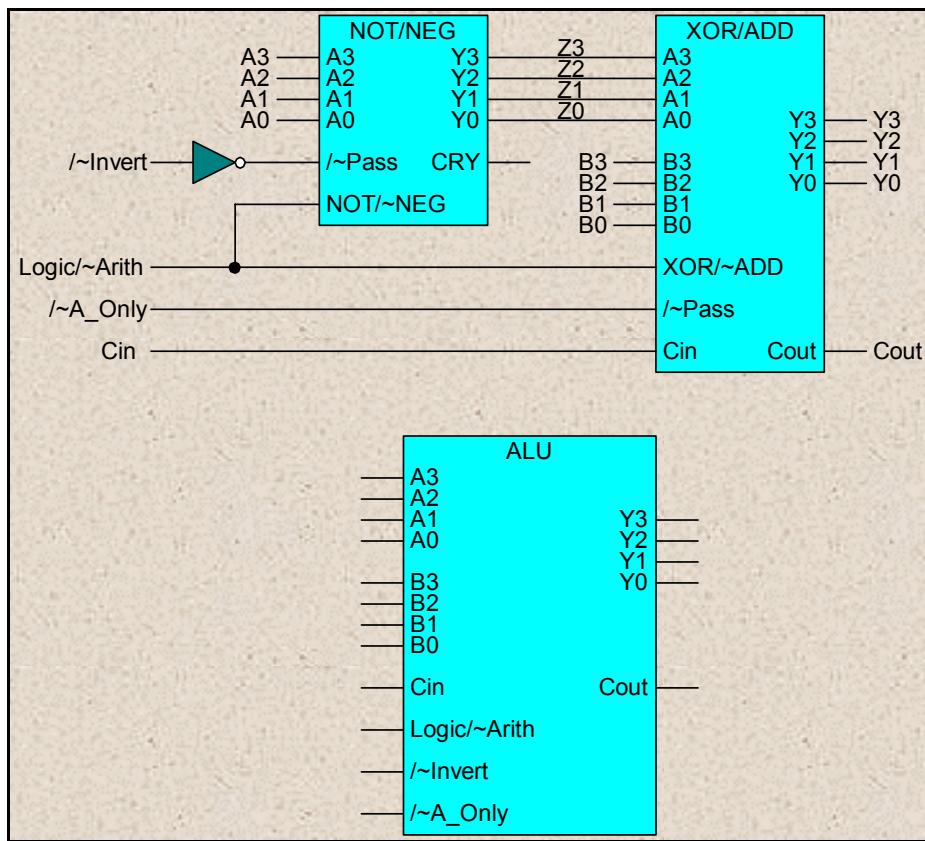


Figure 3-7. The complete ALU.

### Task 3-3: Build and Test the ALU Circuit

Build the complete ALU circuit shown in [Figure 3-7](#). Complete the ALU function definition table, Table 3-1 and include it in your report. Test the ALU and then imbed the ALU in a subcircuit using the notation shown in Figure 3-7. Test the subcircuit then save the subcircuit in your library. Describe in your report briefly how you tested the circuit. The ALU you have saved in your library is the one around which we will construct a microprocessor.

**Table 3-1. ALU Function Definition Table.**

<i>/~A_Only = 0</i>		
<i>Logic/~Arith</i>	<i>/~Invert</i>	<b>Function</b>
0	0	
0	1	
1	0	
1	1	

<i>/~A_Only = 1</i>		
<i>Logic/~Arith</i>	<i>/~Invert</i>	<b>Function</b>
0	0	
0	1	Arithmetic Sum, A+B
1	0	
1	1	

### The ALU Design Methodology

The design methodology we have used to construct the ALU has followed the same pattern we established early on in these laboratory exercises: create subcircuits from complex circuits, then use these subcircuits to create more complex subcircuits, and so on to ever greater levels of complexity. This made designing the ALU easy. Imagine how difficult it would be to design an ALU using only the symbols for NOR gates and to manage the resulting schematic diagram!

In spite of the complexity of laying out large complex circuits at the gate level, many ALU's (or at least parts thereof) are designed just that way today. Rather than use a hierarchical design approach, a truth table is constructed of the complex function to be implemented and sophisticated software programs, using techniques like Karnaugh methods, find the optimum design. (Optimization must balance many competing criteria. Speed, as measured by propagation delay time, is one of these.) Whether or not hierarchical techniques are used in designing a component depends on a number of factors including the type of component to be designed, the need for optimization, and the availability of the necessary software to manage the optimization. In this laboratory manual, we will continue to use hierarchical design techniques because they are easier to use. Also, the resultant designs are easier maintain and easier to understand by ourselves and by others who may work with them. When we come to a component that may benefit from using more sophisticated design techniques, we'll point that out.

### Propagation Delay Time of the ALU

Let's take a short break from microprocessor design and look at a practical problem: propagation delay times. The technical information provided by the manufacturer for all IC's includes the propagation delay from

every input to every output. This propagation delay time is measured by bringing all inputs to a fixed value then changing the one input of interest and measuring the time it takes the output of interest to reach and stay at its final value. The propagation delay time for manufactured circuits depends on many factors (more than we wish to delve into here); however a significant factor in the propagation delay time for any path is the number of gates the path passes through, since propagation delays of gates are larger than that of the links between gates; hence in estimating the propagation delay of our circuit, we will limit ourselves to considering the delays caused by logic gates.

To estimate the propagation delay from every one of our inputs to every one of our outputs, we will need to determine the path length, measured in gate delays, between every input and every output and then multiply the path length by the propagation delay of a gate. By path length, we mean the number of gates that a signal would traverse when traveling from the input to the output.

Determining the path length between every input and output is an intricate task, even for our simple ALU. Fortunately, we have done much of the work for you. Table 3-2 and Table 3-3 show the path lengths from each input to each output for the NOT/NEG and XOR/ADD circuits, respectively, of Figure 3-7. Table 3-2 shows, for example, that there are 6 gate delays between the  $A_0$  input and the  $Z_3$  output signal of the NOT/NEG circuit. Table 3-2 also shows that the  $A_3$  signal has no effect on  $Z_0$ , so the path length for this path is undefined. Note that the path lengths leading to the  $CRY$  and  $Cout$  outputs of the NOT/NEG and XOR/ADD circuits are not included in our tables. Also the input signal  $Cin$  is not included in our table. These signals will play no role in the microprocessor design that you will initially build; so we will ignore their delays for now.

**Table 3-2. Path Lengths for NOT/NEG Circuit.**

Output \ Input	Z3	Z2	Z1	Z0
A3	2	--	--	--
A2	4	3	--	--
A1	5	4	3	--
A0	6	5	4	3
Logic/~Arith	6	5	4	3
/~Invert	6	5	4	3

**Table 3-3. Path Lengths for XOR/ADD Circuit.**

Output \ Input	Y3	Y2	Y1	Y0
Z3	6	--	--	--
Z2	7	6	--	--
Z1	9	7	6	--
Z0	11	9	7	6
B3	6	--	--	--
B2	7	6	--	--
B1	9	7	6	--
B0	11	9	7	6
Logic/~Arith	4	4	4	4
/~A_Only	2	2	2	2

Your task is to use the information in Table 3-2 and Table 3-3 along with the interconnection information contained in Figure 3-7 to find the path length from each input of the ALU to each output and then fill in Table 3-4. We have filled in the information of Table 3-4 that is identical to that in Table 3-3. For example, the path lengths from the B inputs to the ALU output are not affected by the presence of the NOT/NEG circuit; hence these values are identical in Table 3-3 and Table 3-4.

Once Table 3-4 is filled in, complete Table 3-5, listing the input and output signals corresponding to the longest path within each input group (*A*, *B*, and *Control*), the length of the path, and total propagation delay time assuming the propagation delay time of one gate is 5 nanoseconds. For example, from the information already contained in Table 3-4, you can observe that the longest path between input group *B* and the output is 11 gate delays and occurs for the path between the input *B*0 and the output *Y*3. (We are only interested in the longest path from each input group to the output. The longest path within each input group will dictate the delay associated with that group because, when our design is completed, all of the input signals in each group will become valid simultaneously.)

**Table 3-4. Path Lengths for the ALU.**

Output Input \	<b>Y3</b>	<b>Y2</b>	<b>Y1</b>	<b>Y0</b>
<b>A3</b>		--	--	--
<b>A2</b>			--	--
<b>A1</b>		10		--
<b>A0</b>				
<b>B3</b>	6	--	--	--
<b>B2</b>	7	6	--	--
<b>B1</b>	9	7	6	--
<b>B0</b>	11	9	7	6
<b>Logic/~Arith</b>		12		
<b>/~Invert</b>				
<b>/~A_Only</b>	2	2	2	2

**Table 3-5. Propagation Delay Between Inputs (*A*, *B*, *Control*) and Outputs (*Y*).**

Input Group	Long. Path Input	Long. Path Output	# Gate Delays	Propagation Delay (ns)
A				
B	B0	Y3	11	
Control				

To fill in any entry in Table 3-4, you need to consider all paths between any input and any output. For example, consider the path length between *A*1 and *Y*2. From Table 3-2 you can see that there is a path between input/output pairs (*A*1,*Z*1), (*A*1,*Z*2), and (*A*1,*Z*3) for the NOT/NEG circuit. Similarly, Table 3-3 shows that there is a path between input/output pairs (*Z*1,*Y*2) and (*Z*2,*Y*2) for the XOR/ADD circuit. Since there is no path between *Z*3 and *Y*2 in the NOT/NEG circuit, we need not consider the *A*1 to *Z*3 path in the XOR/ADD circuit. This leaves us with two parallel paths between *A*1 and *Y*2: *A*1 → *Z*1 → *Y*2 and

$A1 \rightarrow Z2 \rightarrow Y2$ . To fill in the path length entry between  $A2$  and  $Y2$  in Table 3-4 you must chose the longest of the calculated parallel paths lengths:

$$A1 \rightarrow Z1 \rightarrow Y2 = 3 \text{ (from Table 3-2)} + 7 \text{ (from Table 3-3)} = 10$$

$$A1 \rightarrow Z2 \rightarrow Y2 = 4 \text{ (from Table 3-2)} + 6 \text{ (from Table 3-3)} = 10$$

This calculation requires that we carefully identify all possible paths, calculate their path lengths, and then chose the longest one. To help in visualizing this calculation, we can rotate Table 3-2 clockwise by 90°, then juxtapose it with Table 3-4 so that the  $Z$  outputs rows of the NOT/NEG circuit portion of Table 3-6 are aligned with the  $Z$  input rows of the XOR/ADD circuit portion of Table 3-6. This alignment reflects the interconnection information of Figure 3-7; i.e., that the  $Z$  outputs of the NOT/NEG circuit provide the  $Z$  inputs of the XOR/ADD circuit.

**Table 3-6. Visualization of the Longest Path Calculation.**

NOT/NEG							XOR/ADD				
/~Invert	Logic /~Arith	A0	A1	A2	A3	Input Output	Output Input	Y3	Y2	Y1	Y0
6	6	6	5	4	3	Z3	Z3	6	--	--	--
5	5	5	4	3	--	Z2	Z2	7	6	--	--
4	4	4	3	--	--	Z1	Z1	9	7	6	--
3	3	3	--	--	--	Z0	Z0	11	9	7	6
							B3	6	--	--	--
							B2	7	6	--	--
							B1	9	7	6	--
							B0	11	9	7	6
							Logic/~/Arith	4	4	4	4
							/~A_Only	2	2	2	2

To calculate of the  $A1$ -to- $Y2$ -path length, we simply add the respective  $A1$  column entries of the NOT/NEG circuit (outlined in Table 3-6) to the corresponding  $Y2$  column entries of the XOR/ADD circuit (outlined in Table 3-6):

5		
4	+	--
3		6
--		7
		9
	=	
		10
		10
		--

The path length for this path is taken as the largest value in the right-hand-side array. Note that in this calculation the absence of path entries, ‘—’, either from  $A1$  to  $Z$  or from  $Z$  to  $Y2$ , cause the path length for those particular paths to be meaningless.

As another example of using this approach, we can calculate the path length from  $Logic/\sim Arith \rightarrow Y2$  as 12. This value is the larger of 4 (which is the longest path length for the  $Logic/\sim Arith \rightarrow Y2$  path through only the XOR/ADD circuit [See Table 3-3]) and the longest path length for the  $Logic/\sim Arith \rightarrow Y2$  path through both the NOT/NEG and the XOR/ADD circuit, calculated as:

6		
5	+	--
4		6
3		7
--		9
	=	
		11
		11
		12

### Task 3-4: Calculate the Propagation Delay Times of the ALU

Complete Table 3-4 and Table 3-5. (Remember the propagation delay time for each gate is 5 nanoseconds.) Given what you know about your circuits, are your longest paths the ones you would have guessed were the longest paths before you did any calculations? In other words, does your result make sense? (Checking your results against what your common sense tells you is a technique most engineers use to eliminate obviously gross errors. For example, suppose you are trying to calculate an estimate of the mass of a fly,  $M_f$ , by multiplying the mass per unit volume of water,  $M_w$  by your estimate of the volume of a fly,  $V_f$ ; that is,  $M_f = M_w \times V_f$ . If your answer is 10 kg (22 lb.), your common sense tells you that something is obviously wrong with your calculation. If your answer is 0.1 gram, then your common sense tells you that at least your result is not obviously wrong.)

### Task 3-5: Backup Your Files

Take a moment and back up your libraries and circuit files on a new floppy disk and mark it ‘SimLab#3’. Do this using a different floppy disk after every lab and you may just save yourself much unnecessary work.

# SIMULATION LAB 3: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Top-Down Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Work Performed</b>		
ALU: Functional Description and Test Results		
NOT/NEG Circuit: Functional Description and Test Results		
XOR/ADD Circuit with Pass Through: Functional Des. & Test Results		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

### Report Writing Reminders:

<p><b>Caveat emptor:</b> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used</p> <ul style="list-style-type: none"><li>• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.</li><li>• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.</li><li>• <b>Label</b> all figures or circuits. (E.g. Figure 1. Schematic of . . ., Figure 2. . ., etc.)</li><li>• Refer in the text of your report to all circuits or figures that you include in the body of your report.</li><li>• Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.</li></ul>
--

# SIMULATION LAB 3: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Task Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 3-1: Build the NOT/NEG Circuit		
Task 3-2: Build and Test the XOR/ADD Circuit		
Task 3-3: Build and Test the ALU Circuit		
Task 3-4: Calculate the Propagation Delay Times of the ALU		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

<b>Caveat emptor:</b> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.
• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.
• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.
• <b>Label</b> all figures or circuits. (E.g. Figure 1. Schematic of . . . , Figure 2 . . . , etc.)
• Refer in the text of your report to all circuits or figures that you include in the body of your report.
• Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

Put “X’s” in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use ‘5’ to indicate that you ‘strongly agree’, ‘3’ to indicate that you are ‘neutral’, and ‘1’ to indicate that you ‘strongly disagree’. Use ‘NA’, ‘Not Applicable’, when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your **responses will not be graded**. They are for your instructor’s information only.

Table \_\_: Self-Assessment of Outcomes for Simulation Lab 3: Arithmetic and Logic Unit.

After completing the assigned tasks and report, I am able to:	5	4	3	2	1	NA
Build, test, debug, and imbed in a subcircuit, an elementary arithmetic and logic unit (ALU).						
Describe the arithmetic and logical operations of which the ALU is capable.						
Describe the input control line values that correspond to each arithmetic and logical operation of the ALU.						
Write a top-down description of a complex logic circuit.						
Calculate the approximate propagation delay time of a combinational-logic circuit.						

Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

# SIMULATION LAB 4: THE BRAINLESS MICROPROCESSOR

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Use LogicWorks™.
- Perform a 1's complement operation.
- Have completed [Simulation Lab 1: Half Adder, Increment & Two's Complement Circuit](#).
- Have completed [Simulation Lab 2: 4-Bit Full Adder, Multiplexer & Decoder](#).
- Have completed [Simulation Lab 3: Arithmetic and Logic Unit](#).

**Equipment:** Personal computer and LogicWorks™.

**Objectives:** In this laboratory exercise you will complete the design of three types of memory and the communication bus architecture of a simple microprocessor. Then you will act as its controller to cause it to execute a simple program.

**Outcomes:** When you have completed the tasks in this experiment you will be able to:

- Build and debug a simulation of a 4-bit register.
- Build and debug a simulation of a 4-bit buffer.
- Build, debug and control a simulation of a central processing unit (CPU).
- Build, debug and control a simulation of a ROM, RAM and an output port.
- Build, debug and control a simulation of an address decoding circuit.
- Build and debug a simulation of a microprocessor that is absent a controller.
- Act as the controller for an elementary microprocessor.
- Calculate the maximum clock frequency that may be applied to a synchronous circuit.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab, it is recommended that you use a top-down format when organizing your lab report.

(If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Top Down Report Writing Guidelines](#).)

---

## Introduction

In simulation labs 1, 2, and 3, we built almost all of the combinational logic circuits we'll need for our microprocessor. In this laboratory exercise, our aim is to build a few more circuits that we'll need and to use the subcircuits we have already created to build a complete microprocessor – almost; the only piece of the microprocessor we'll be missing is the controller – and we will be building that in the next laboratory exercise<sup>1</sup>. In this lab exercise we want to accomplish several things. First we wish to build registers that our microprocessor will use to store data temporarily. Second we want to get an overview of how a microprocessor accesses memory and we want to design a LogicWorks™ memory circuit that will allow us to use the mouse to store and change values in memory. (These memory circuits are what we will eventually use to store a program.) Finally, we will assemble our microprocessor (without the controller) and call it the **brainless microprocessor**. You will act as the ‘brains’ (or controller) for this processor and manually manipulate the signals controlling the ALU, registers, decoders, and memory, to cause the microprocessor to perform a series of operations we will eventually call an instruction, and a series of instructions we will eventually call a program.

The first circuits we wish to build are registers. To build a register you will need to know something about sequential logic circuits: specifically latches and flip-flops. If you are not familiar with latches review the section, [The Latch](#) in Hardware Lab 3. If you are already familiar with flip-flops, [skip the discussion in the next section](#); otherwise, let's learn about flip-flops.

## Flip-Flops

A flip-flop is a memory device, built from AND/OR/NOT gates, which requires a clocking event to cause it to store a value consistent with its synchronous input values. When no clocking event occurs, the flip-flop's state does not change. The clocking event is either a positive or negative clock edge (i.e., a  $0 \rightarrow 1$  or  $1 \rightarrow 0$  clock transition, respectively). A positive-edge-triggered device responds to its input only when the clock signal input is in transition from low to high. A negative-edge-triggered device responds to its input when the clock signal input is changing from high to low. Edge-triggered devices are desirable because they can be easily controlled to sample their input values at a precise instant in time.

In this laboratory exercise we will be using the LogicWorks™ J-K flip-flop. It is a negative-edge-triggered device that has the LogicWorks™ schematic symbol shown in [Figure 4-1\(a\)](#) (compared with typical

---

<sup>1</sup> For an overview of this laboratory exercise refer to the PowerPoint lecture slides contained on this CD in the file labeled ‘Brainless μP & Design Project.’

textbook schematic symbol shown in Figure 4-1(b) ) and has the function definition table shown in Table 4-1. (Note that the symbol  $\overline{\square}$  used in Table 4-1 indicates a negative clock edge.) The characteristics of the LogicWorks™ J-K flip-flop are consistent with those described in the textbook you use for your course.

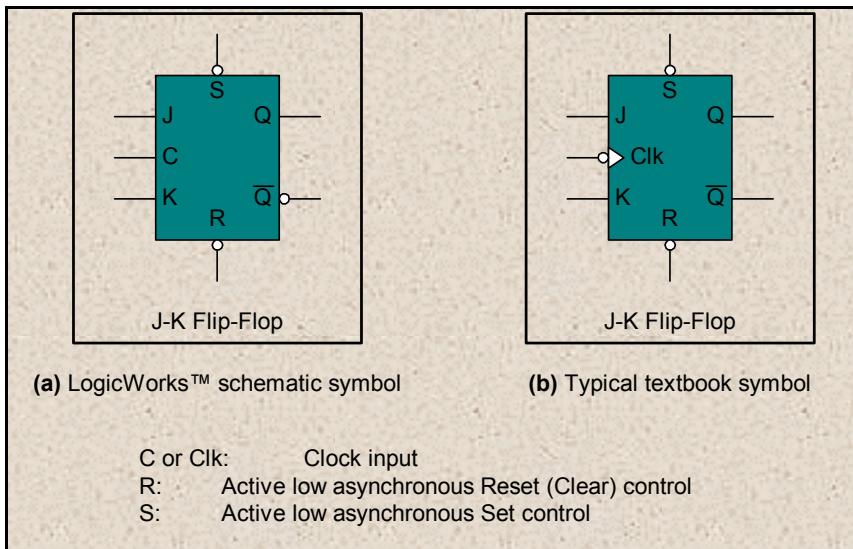


Figure 4-1. Schematic symbols for negative-edge-triggered J-K flip-flops.

Table 4-1. Function Definition for J-K Flip-Flop.

S	R	Clk	J	K	Q
1	1	$\overline{\square}$	0	0	Last Q
1	1	$\overline{\square}$	0	1	0
1	1	$\overline{\square}$	1	0	1
1	1	$\overline{\square}$	1	1	$\overline{\text{Last Q}}$
0	1	X	X	X	1
1	0	X	X	X	0
0	0	X	X	X	Invalid

Upon a clocking event, the J input (when high) acts as a set input, the K input (when high) acts as a reset input, and when  $J=K=0$ , the flip-flop retains the previously stored value or state. When both J and K are high, the flip-flop toggles on every clocking event, i.e., it changes state on every negative clock-signal edge. The asynchronous set and reset inputs shown in Figure 4-1 use the nearly universal graphic symbol for an active-low input,  $\neg\circ$ . When low, Table 4-1 indicates that the S input causes the flip-flop output to be set to 1 asynchronously, i.e., the flip-flop output is set immediately to 1, regardless of the value of the clock input. Similarly, when the R input is brought low the flip-flop output is reset (brought to 0) asynchronously. These inputs are called asynchronous inputs since they do not rely on a clock signal to effect their action. The X's in Table 4-1 indicate that when S or R are low, the action of the flip-flop is independent of the values of other

inputs, i.e., the other inputs are ‘don’t cares’. When R and S are both brought low, they attempt to asynchronously force the device into a set and reset state simultaneously. The result of applying such contradictory control signals puts the device into an invalid state, as indicated in Table 4-1.

The R and S inputs need not be connected when using TTL J-K flip-flops in the hardware lab because unconnected inputs in TTL circuits behave as if they were connected to a high voltage level, or +5 Volts – although leaving these inputs unconnected is poor practice. (See [Hardware Lab 2](#) for an investigation of the effects of leaving inputs unconnected.) When using the LogicWorks™ J-K flip-flops, the R and S pins need to be connected to some source of digital signal. LogicWorks™ does not make the assumption that unconnected input values, including R and S values, act as if supplied with +5 Volts. If you do not connect the R and S values to a valid logic signal, the LogicWorks™ flip-flop output will appear as an ‘X’ when viewed using a binary probe; this ‘X’ means that the output is indeterminate. It is indeterminate (in this case) because the values of some of the inputs to the flip-flops are not defined. If you do not plan to use the R and S values during any of the tasks described below, simply connect them to +5 V.

## Registers

Registers are arrays of flip-flops that share common set and reset control inputs and a common control signal for preventing the flip-flops from responding on an input-clocking event. The circuit for a 1-bit register built around a J-K flip-flop is shown in [Figure 4-2](#). This device operates as follows: When the ENABLE signal, (abbreviated EN or IE, which stands for input enable), is inactive (low) the outputs of both AND gates in this figure are 0; this causes the J-K flip-flop to retain the previously stored value (or state) on the next clock pulse. When EN is active (high) the input supplied to J is DATA and to K is  $\overline{\text{DATA}}$ ; when DATA = 1, the flip-flop is set (to 1) on a negative clock edge and when DATA = 0, the state of the flip-flop is reset (to 0) on a negative clock edge; hence, for the information in the register to change (i.e. for data to be ‘loaded’ into it), the input enable must be a 1, and the clock input signal must simultaneously experience a  $1 \rightarrow 0$  transition. When both of these events occur, a snapshot of the data is loaded into the register. Notice that the device of Figure 4-2 functions as a D flip-flop - except that it has an extra control input, the enable.

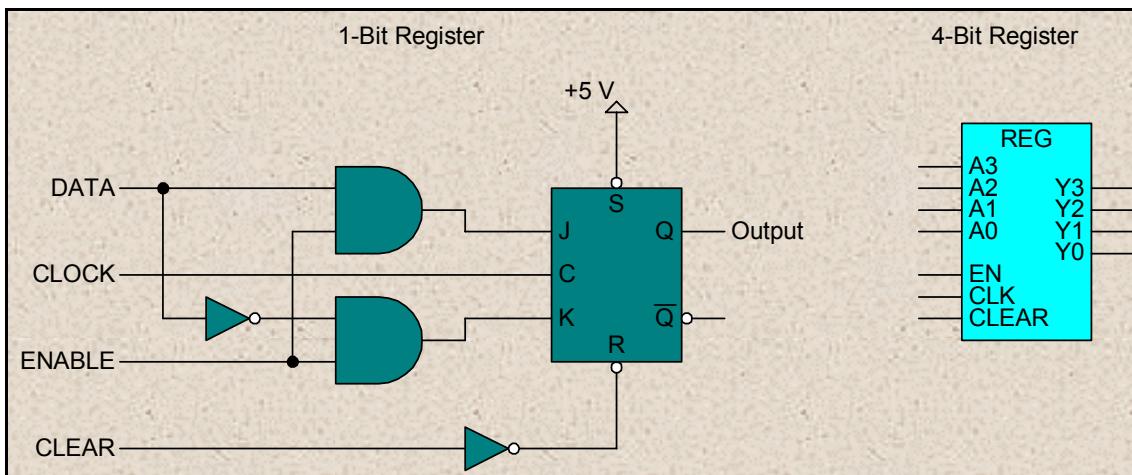
The 1-bit register shown in Figure 4-2 holds a single bit of information. When four of these 1-bit registers are combined so that they share common enable, clock, set, and reset signals, we have a 4-bit register, whose subcircuit symbol is shown in Figure 4-2.

**LogicWorks™ Note:** The J-K flip-flop can be found in the ‘**Simulation Logic**’ library.

### Task 4-1: Build the 1-Bit and 4-Bit Registers

Build the 1-bit register circuit shown in [Figure 4-2](#). Test this circuit and add it to your library. Next, combine four 1-bit registers into a 4-bit register. The 4-bit register should have one data input and one data

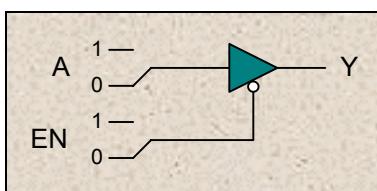
output for each of the four 1-bit registers comprising your register. The 4-bit register should have one common clear (also known as reset), one common enable, and one common clock input supplied (in common) to all of the four 1-bit registers. After completely constructing the 4-bit register, determine what tests you need to perform to ensure that the register is working. Test your circuit and add it to your library. Enclose the 4-bit register in a subcircuit using the symbol shown in [Figure 4-2](#). Test the subcircuit and add it to your library. Be sure to record the results of all tests in your lab notebook and to include them as part of your lab report. In your report, comment on the tests you chose and on why you believe that your tests are a reliable indicator that your circuit is operating correctly.



**Figure 4-2.** 1-bit register and symbol for 4-bit register.

## Introduction to Buffers

A buffer is a circuit with a data and a control input. The control input controls whether the data input signal is allowed to propagate to buffer's output. The buffer circuit uses the three-state device described in [Hardware Lab 2](#). If you have completed Hardware Lab 2 or are familiar with three-state (a.k.a. tri-state) devices, [skip the remainder of this section and perform Task 2](#).



**Figure 4-3.** Active-low enabled three-state device schematic.

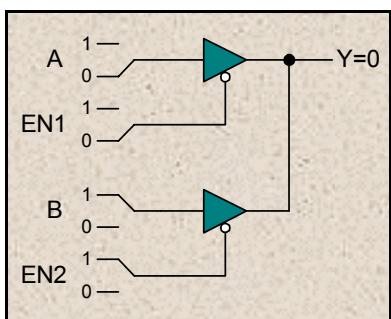
**Table 4-2. Function Definition Table for a Three-State Device.**

EN	Output
0	A
1	'Z'

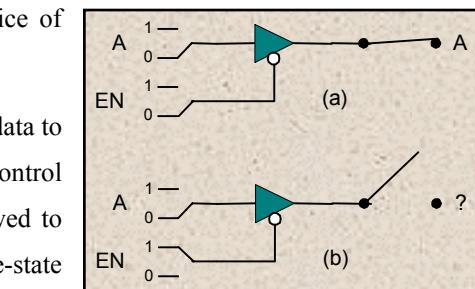
The three-state buffer symbol shown in Figure 4-3 has two inputs: a data input (A) and a control/enable input (EN). As its name implies, it has three output states. When the control/enable input of the three-state buffer is active, the output is the same as the input 'A' (as shown in [Figure 4-4\(a\)](#)): either a 0 or 1. If EN is inactive, the device enters its third state, a high impedance state. In this state, the voltage

measured at the output is not related to the ‘A’ input value, as shown schematically in [Figure 4-4\(b\)](#). In the high-impedance state, the output voltage is controlled by the downstream devices not by the buffer’s input signal. The function definition table for the three-state device of Figure 4-3 is shown in Table 4-2.

Three-state devices are used to allow different sources of data to supply data signals to a common point while allowing a control signal to control which of the various source signals is allowed to propagate to the common point. A typical application of three-state buffers is shown in [Figure 4-5](#). This circuit is used to allow signals A and B to supply a signal to point Y; when EN1=0 (while EN2 is inactive), the signal supplied to point Y is A when EN2=0 (while EN1 is inactive), the output of this circuit is B. When EN1=EN2=1 (i.e., when both control inputs are inactive), both three-state devices are in a high impedance state, and the output of the circuit (indicated by ‘Z’ on a LogicWorks™ binary probe) is unrelated to values applied to either input. When both EN1 and EN2 are active (low) there is a potential for a data conflict. A data conflict means that a common point, in this case Y, is being driven by two signal sources that have opposite logic values. If A and B have opposite logic values and EN1=EN2=0, a data conflict at Y exists and LogicWorks™ indicates this with a ‘C’ when viewed using a binary probe. If A and B have the same logic values then no data conflict can exist, regardless of the value of EN1 and EN2. A summary of the output value for all combinations of enable values for the circuit shown in [Figure 4-5](#), is given in Table 4-3.



**Figure 4-5.** Typical three-state buffer circuit.



**Figure 4-4.** Effect of EN input on input/output of an active low enabled three state buffer.

**Table 4-3. Function Definition Table of Circuit of Figure 4-5.**

EN2	EN1	Output
0	0	‘C’*
0	1	B
1	0	A
1	1	‘Z’

\* There is a potential for a data conflict when EN1=EN2=0. Whether an actual data conflict exists depends on the input values of A and B.

What we have shown in Figure 4-5 are two 1-bit buffer circuits. These circuits allow two 1-bit signal sources, A and B, to share a common communication bus. (By bus, we mean a signal line, or collection of signal lines, that are connected to more than one signal source by buffer circuits.) In our microprocessor, we will need to have more than two sources share a common communication bus. If the outputs of many three-state buffers are wired together and all but one are in the high-impedance state, then the active buffer

will control the value measured at the output; however, if the outputs do not ‘take turns’ properly, (i.e., if more than one of the three-state buffers is active) the potential exists for a data conflict. Therefore, it is necessary when using three-state devices to be sure that all outputs but one are in the high-impedance state. In our microprocessor, we will use similar buffer circuits to allow eight 4-bit memory locations to share a common communication bus. To allow these memory circuits to share a common 4-bit communication bus, we will need to construct a 4-bit buffer circuit made of four three-state devices that share a common enable signal as described in the next task.

### Task 4-2: Build a 4-Bit Buffer

The three-state buffer in LogicWorks™ has a single active low enable. For our microprocessor design we will need a 4-bit buffer with two active high enables; that is, both of the enables must be high to enable the four three-state outputs. You can design such a control scheme by using a 2-input NAND gate and routing its output to the active low enable inputs of each of four three-state devices. Design this circuit, enclose it in a subcircuit (see Figure 4-6 for the subcircuit symbol to use), test it, and add it to your library. Mention briefly in your report how you tested your subcircuit and comment on why you believe that your tests are a reliable indicator that your circuit is operating correctly.

**LogicWorks™ Note:** The three-state buffer is located in the ‘*Simulation Logic*’ library and is called ‘*buffer-1 T.S.*’.

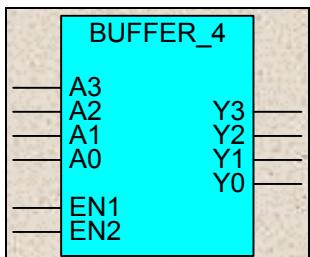


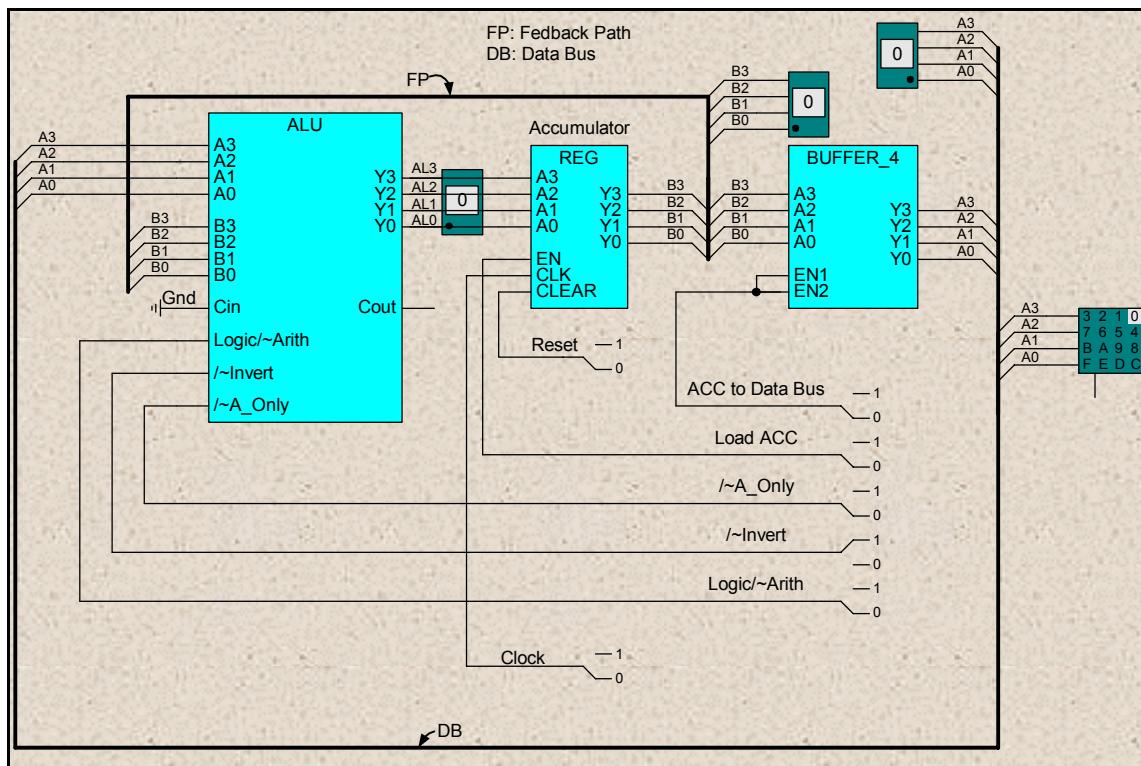
Figure 4-6. Subcircuit schematic for the 4-bit buffer circuit.

### Task 4-3: Build the Brainless Central Processing Unit

Using the ALU, register and buffer subcircuits you built in this and previous laboratory exercises build the brainless microprocessor central processing unit (CPU) shown in Figure 4-7. Name all signals as shown in Figure 4-7. (We will be using these names in Task 11.) Be sure to assign the name ‘FP’ to the Feedback Path bus and ‘DB’ to the Data Bus. Note how the use of the LogicWorks™ bus notation simplifies the schematic of Figure 4-7. Also note the liberal use of the hex displays to get access to the information of the various buses. Make sure you understand which are the most and least significant pins of the hex displays before you paste them into the circuits. If you are unsure how to use the hex display or the hex keyboard, see ‘Hex Display’ or ‘Hex Keyboard’ in Simulator Tutorial: [Using LogicWorks™ 4 for Windows®](#).

Once you complete construction, you may observe that the hex displays at the output of the accumulator register and/or on the data bus may display ‘X’s. An ‘X’ displayed on the accumulator hex display means that either the accumulator register was not initialized or one of the input signals to the register is undefined. If you have constructed the accumulator circuit correctly, you can correct the ‘X’s problem by clearing the accumulator register with the *Reset* switch. An ‘X’ displayed on the data-bus hex display is most likely to mean that you have a data conflict problem. A data conflict in our architecture can occur on the data bus (DB) when the buffer is enabled while the hex keyboard is driving the data bus. To eliminate this problem, reset (to 0) the *ACC to Data Bus* switch. (In general, to identify whether a data conflict exists, connect binary probes to each of the signal lines that comprise the problematic bus; if one or more of the binary probes displays a ‘C’, then there is a data conflict.)

The presence of the accumulator register (controlled by the clock signal) in Figure 4-7 transforms the circuit we have been building in a fundamental way: it transforms it from a combinational-logic circuit into a synchronous circuit. We are getting very close to completing the design of a microprocessor CPU. In the next task, you are asked to test and understand the workings of the partial CPU you just constructed.



**Figure 4-7.** Schematic of brainless central processing unit.

## Task 4-4: Test and Control the Brainless Central Processing Unit

Because this brainless CPU contains an ALU and a register for storing information, we can now perform meaningful arithmetic and logical operations. To test this circuit, you will add two numbers together and store them in the accumulator.

Start by setting the hex keyboard that controls the data bus values to any hex number. Observe this number on the data-bus hex display. Next set the switches connected to the ALU control lines so that the ALU implements the pass-through operation. (You may want to refer to the ALU function definition table you constructed in the previous lab exercise.) Set the *Load ACC* switch to 1 so that the accumulator register is enabled and make sure that the *Reset* switch is reset to 0 so that the register can store data under synchronous control. Toggle the *Clock* switch twice so that you lock the output of the ALU into the accumulator with a negative clock edge. Next, set the hex keyboard to any hex number that, when added to the one in the accumulator is less than F. (You can set it to any number but getting a sum less than F simplifies checking your result.) Set the switches connected to the ALU control lines so that the ALU implements the ADD operation. Observe the correct sum at the output of the ALU and then lock it in to the accumulator by toggling the *Clock* switch twice. (Make sure the *Reset* switch remains reset to 0.) Observe that the correct sum appears at the output of the accumulator.

Test a few other additions and any other operations you believe necessary. Once you are convinced that your circuit is working correctly, remove the hex keyboard, set the *ACC to Data Bus* control switch to 1, and observe that the output of the accumulator appears on the data bus. When this output appears on the data bus, how does that change the output of the ALU? If the hex keyboard is not removed and the *ACC to Data Bus* switch is set to 1, what would you expect to see in the hex display attached to the data bus? Why do you think the register at the output of the ALU is called the ‘accumulator’?

Play with this circuit until you are sure you understand how it works. It is important that you understand how this circuit works because it forms the kernel of our processor.

### The Clock Signal

We have been acting as the clock for our processor. We will continue to ‘manually’ toggle the clock switch to simulate a clock signal since we will want to observe how the state of our microprocessor changes with each clock pulse. In practice, high-speed clock signals are used to allow a microprocessor to execute instructions rapidly. The clock signal in a typical microprocessor has the form shown in [Figure 4-8](#). In this figure, a clocking event is any negative clock edge that causes the registers in our microprocessor to change state. The function of a clock is to set the pace at which instructions are executed and synchronize all registers so that they change state at the same time. A 1000-megahertz clock will go through a complete 0-1-0 cycle one-thousand-million times per second. That means that a system with a 1000-megahertz clock could execute 1000 million instructions per second if each instruction took only one

clock cycle to execute. Typically, instructions take more than one clock cycle to execute. Further, there are often long periods of time spent waiting for data to arrive from memory before useful operations can be carried out; so the pace at which useful instructions are executed is much slower than a naïve calculation would indicate.

For most of us ‘faster is better’, so we would all like a computer with a fantastically high clock speed; however the speed of the clock is limited by the time it takes for a single-cycle instruction to execute. Let us assume that, if we were to realize our microprocessor design in hardware, this time would be dictated by the propagation delay time of the ALU. If the longest propagation delay time of our ALU is 100 nanoseconds, then  $100 \times 10^{-9}$  sec. is the shortest time span between the time the ALU receives a valid input and the time it produces a valid output - if we are to guarantee that the output is valid regardless of the operation being executed. If our clock signal is designed so that, at each clocking event, a potentially new valid result is available from the ALU, then  $100 \times 10^{-9}$  sec. is the shortest practical time span between two clocking events. Such a clock signal would go through  $(100 \times 10^{-9})^{-1}$  cycles in one second or, equivalently, operate at a speed of 10 megahertz. If we want a processor with a higher speed, we’ll simply need to build our ALU in such a way that the propagation delay time of the ALU is shorter<sup>2</sup>. We can do this by using faster components and by putting the components closer together so that the time it takes the signal to propagate between components is smaller.

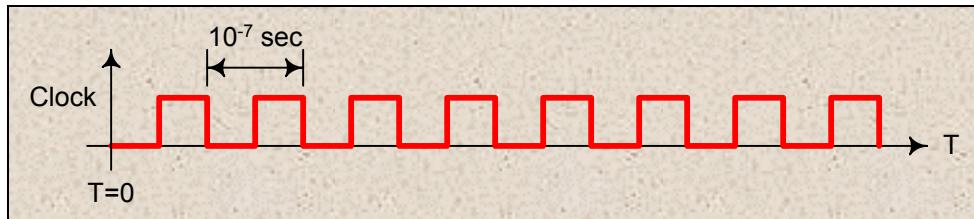


Figure 4-8. Clock signal.

### Task 4-5: Calculating the Maximum Clock Frequency

Using the ALU delay times you calculated in Simulation Lab 3 and entered into [Table 3-5](#), calculate the maximum frequency of a clock that you could use to drive your microprocessor as shown in [Figure 4-7](#). You may assume that the *A* operand and control signals become valid simultaneously at the falling edge of the clock signal. You may also assume that the propagation delay of the accumulator register is 4-gate delays. (Hint: You will need to adjust the ‘*B* input to output’ delay information you calculated in Simulation Lab 3 to account for the added delay caused by the accumulator register. Does this change in delay make a difference?)

<sup>2</sup> Creating a processor with a high clock frequency involves additional considerations that are beyond the scope of this discussion.

## Introduction to Microprocessor Memory

Having built a partial brainless microprocessor, our next step is to add memory to the system and test that we can control the CPU and memory combination. First, let's understand more about typical memory systems used with microprocessors.

We can classify the memory we find in a microprocessor system as: cache memory, which is memory on the processor IC itself, and external memory, which comes in various varieties. (On the processor IC, there are also registers used to temporarily store operands and addresses—such as the accumulator register you used in your partial CPU—but we usually refer to this type of memory using the specific term register, rather than the general term, memory.) The personal computer used for performing these simulation exercises probably has a relatively small amount of cache memory, perhaps 640k (k means thousand) bytes (a byte is 8 bits), and many millions of bytes of external memory (perhaps 128 megabytes). Because cache memory is located on the central processor unit (CPU) IC and external memory is located further away, the data stored in cache memory requires less time to travel to the processor's ALU than external memory. (The time that it takes to transfer data between memory and a memory register in the CPU, is a function of many things, one of these is distance. The speed with which data can travel is limited by the speed of light [which travels about 1 foot or 1/3 meter each nanosecond] and other characteristics of the transmission line through which it travels. Hence, the closer the memory is to the CPU, the faster the CPU can get access to the data.) We will not simulate transmission line delays and memory access time; hence we will not need to distinguish between cache and external memory; but this distinction is important when working with real systems. In these exercises, we will treat all memory as external memory.

Microprocessor external memory, shown schematically in Figure 4-9, can be grouped into three different types. The box labeled Read Only Memory (ROM) contains a read-only memory. As its name implies, data can be read from ROM but data cannot be written to it. Microprocessor ROM contains information that will not change for the life of the processor, such as the instructions that are used to boot-up your computer.

The box labeled RWM contains a read-write memory. This type of memory may be written to, and then read from when the stored data is needed. There are two basic types of RWM: random-access memory (RAM) and sequential access memory. In sequential access memory, we must scroll past the first  $N-1$  pieces of data to get access to the  $N$ th piece of data. An example of sequential access memory is a magnetic tape; we must mechanically move past the first five songs on the cassette to get to the sixth. With RAM, we can read from or write to memory locations in any order that we want; more precisely, the time needed to access any location in RAM is the same.

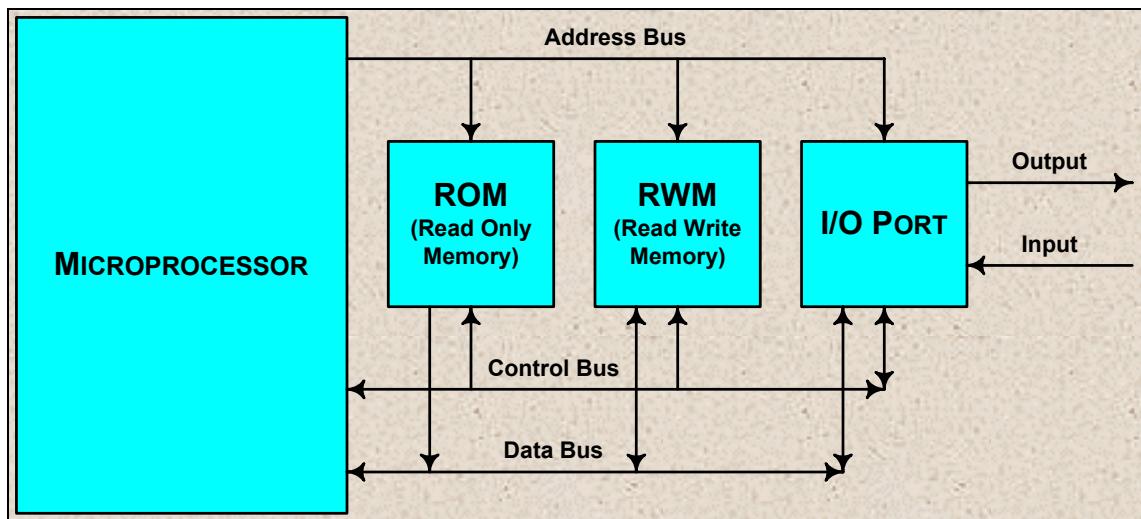


Figure 4-9. Block diagram of microprocessor memory system.

The input/output (I/O) ports have access to information from the external world. One I/O port may have access to modem data, while another may access the data entered via the keyboard, a third may drive the computer monitor. Some of these ports have one-way access, while others are bi-directional. For example, the port reading the keyboard information may not write data to the keyboard; hence it is unidirectional. Likewise, data communication with most video screens is unidirectional. By contrast, a modem port must be bi-directional; the modem must be able to send as well as receive data from the external world. One significant difference between the I/O port and the other types of memory, is that I/O port data arrives or is accessed by the external world asynchronously; the external world knows nothing of the synchronizing clock signal inside the processor. For example, you hit keys on the keyboard at your own whim, not at a time dictated by your computer.

To communicate with the memory shown in Figure 4-9, we will need three different buses: an address bus, a control bus, and a data bus. The data bus is bi-directional; data can either be sent to or received from the external memory via this bus. This bus is 4-bits wide, consistent with our 4-bit word design. The address bus, also 4-bits wide, supplies the address that uniquely specifies which memory location in our 4-bit address space is to be accessed. The control bus in our system will be 2-bits wide. It will contain a *Read* control line and a *Write* control line. The address and control buses are unidirectional; information flows from the CPU to external memory.

In the next tasks we'll build simulations of these various memory types. We'll also need to build the bus structure, addressing logic, and memory control logic we will need to access these various memory devices.

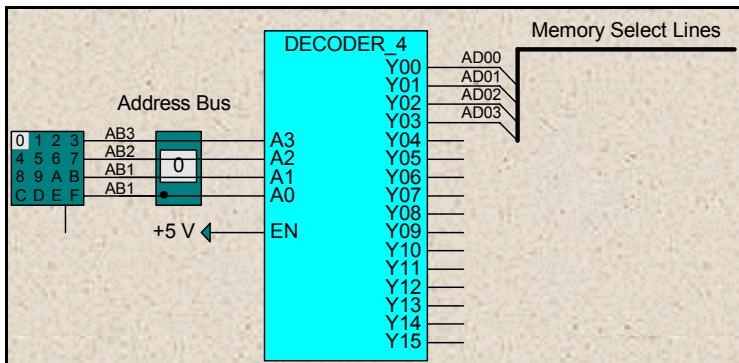


Figure 4-10. Addressing logic.

### Addressing Logic

The addressing logic we will need in our microprocessor is shown in Figure 4-10. We will simulate addresses in this laboratory exercise using the hex keyboard. These addresses, when applied to the decoder, will cause one of the decoder's output lines to become active (high). Each of these output lines will be routed (in a later task) to a different memory location and used to enable that memory location to make its contents available to our data bus.

### Task 4-6: Build the Addressing Logic

Build the addressing logic circuit shown in Figure 4-10 using a LogicWorks™ hex keyboard, a hex display and the 4-to-16 decoder you have stored in your library. Connect the first four output lines to a LogicWorks™ bus as shown. Test your circuit, record the tests you performed, then add it to the partial CPU circuit you built in the previous task. (Locate this circuit anywhere for now. We will place it in an appropriate location in a later task.)

### ROM Memory

We will simulate ROM in such a way that we can conveniently change its contents using our computer mouse and we will connect the output of this ROM to our data bus using a buffer circuit. We will use the LogicWorks™ hex keyboard as our ROM and control its connection to the data bus using the 4-bit buffer circuit you constructed earlier as shown in Figure 4-11. By ‘clicking’ on the hex keyboard you will be able to change the value ‘stored’ in the simulated ROM. It will not be possible to change the value stored in the hex keyboard by using control lines in our simulation or by feeding data from the data bus into the hex keyboard; hence this memory circuit truly functions as ROM.

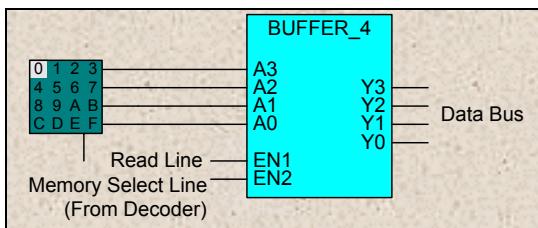


Figure 4-11. 4-bit ROM memory cell.

The value in ROM will be made available to the data bus when EN1 and EN2 inputs to the 4-bit buffer circuit are both high. When we add this memory to our circuit, we will drive EN1 of the 4-bit buffer circuit with the *Read* control line and EN2 with one of the memory select lines. With these two connections made, a ROM memory location will gain access to the data bus only when the *Read* line is high and when the memory select line (coming from the addressing logic) is high. (The *Read* line, which you will control in this exercise, will be high when we want to read a value from memory, and low when we want to write to memory.)

### Task 4-7: Build a 4-Bit ROM Memory Cell

Build the 4-bit ROM memory and buffer circuit shown in Figure 4-11, test it, and save it. Remember to record the results of your tests in your laboratory notebook.

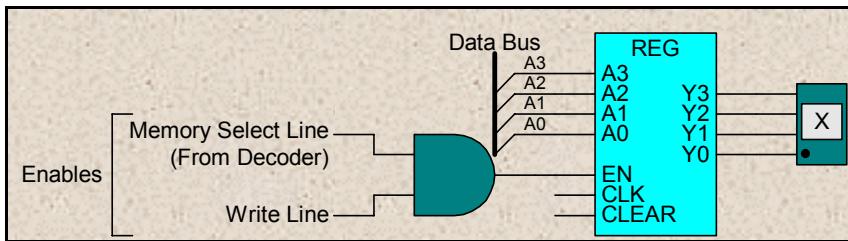
#### Output Port

The I/O port we will simulate will be unidirectional; we will be able to write data to the port (and view it using a LogicWorks™ hex display), but once written, the data will not be retrievable by the microprocessor; hence our port will be an output port. To control writing into this output port we will need to use the *Write* control line. (See [Figure 4-12](#).) The *Write* line contains a signal that you will control in this lab exercise – ultimately it will be controlled by the controller you will design in the next laboratory exercise. The *Write* signal is designed to be high when we wish to write into memory and zero otherwise.

The circuit we will use to simulate this port is shown in Figure 4-12. The hex display at the output of the 4-bit register will allow us to view the data written into the register. Just as in the ROM circuit described earlier, we will need an enable circuit that will permit data to be loaded into the output port (register) only when two signals are active simultaneously, when the *Write* line is high and when this memory location is selected by the decoder. The 2-input AND gate shown in Figure 4-12 implements this control strategy.

### Task 4-8: Build 4-Bit Output Port

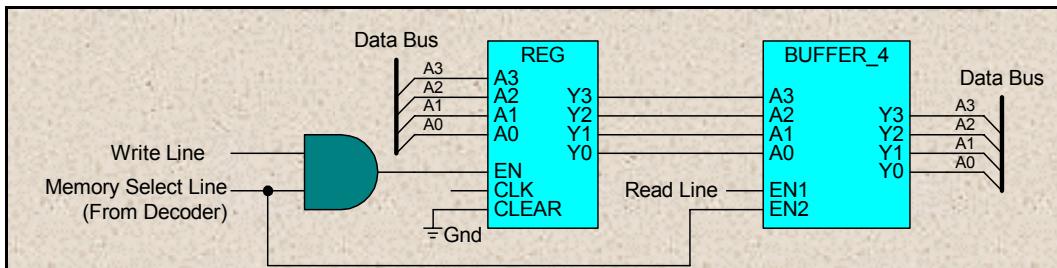
Build the 4-bit output port using the circuit shown in Figure 4-12, test it, and save it. Remember to record the results of your tests in your laboratory notebook.



**Figure 4-12.** 4-bit output device.

## RAM

The last type of external memory we will use in our microprocessor will be RAM. The simulated RAM circuit we will use is shown in Figure 4-13. We will use the same control philosophy for writing into the RAM register as we used with the output port: two control signals, the *Write* line and the connected memory select line (from the decoder), will need to be high to write into the RAM. To read from the RAM, we will use the same control philosophy we used with ROM: the memory select line (from the address decoder) and the *Read* line will need to be high; these two lines, when high, will cause the buffer to grant data bus access to the RAM output lines.



**Figure 4-13.** 4-bit random access memory (RAM; also called read-write memory.)

Notice in Figure 4-13 that the data bus is connected to both the RAM input and output. This type of connection is required if we wish to be able to write to and read from RAM using the same data bus.

## Task 4-9: Build the 4-Bit RAM Cell

Build the 4-bit RAM cell using the circuit shown in Figure 4-13, test it, and save it. Remember to record the results of your tests in your laboratory notebook.

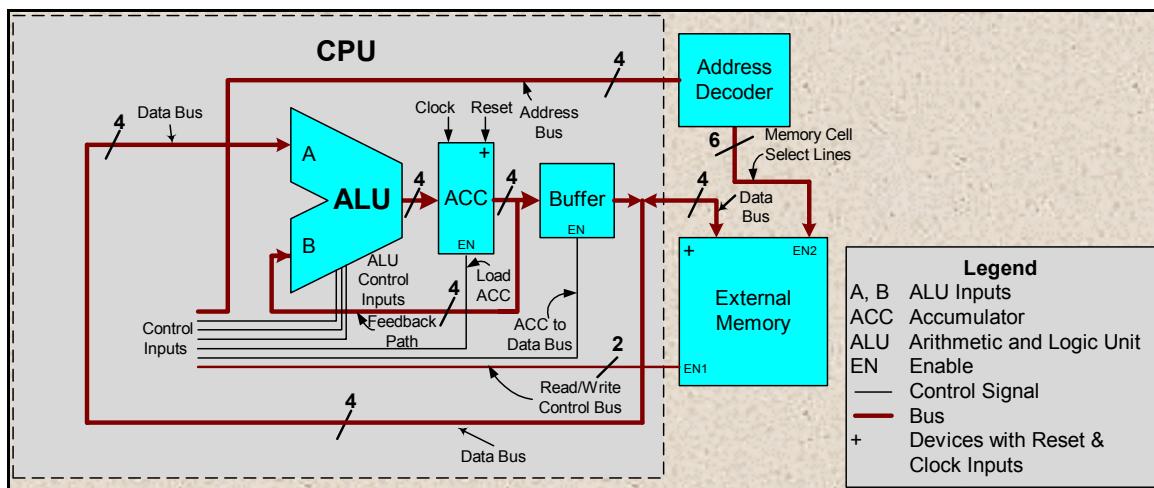
## Architecture of the Brainless Microprocessor<sup>3</sup>

You have now completed all of the pieces you will need to build the brainless microprocessor. Let's get an overview of the microprocessor architecture and then build the brainless microprocessor. The partial

<sup>3</sup> The following sections describe how the brainless microprocessor circuit operates. Some students find it easier to construct the brainless microprocessor, i.e. complete Task 4-10, first and then read the supporting material contained here.

architecture of the brainless microprocessor is depicted by the block diagram of Figure 4-14. By architecture, we mean the communication and control bus structure that facilitates the interactions between various circuit blocks. In Figure 4-14, a line with a slash (/) in it is used to signify a bus. The number next to the slash is the number of individual signal lines contained within the bus. The number is also called the bus width.

The portion inside the dashed box in Figure 4-14 is called the CPU (Central Processing Unit), which you built in [Task 4-3](#). Compare the block diagram of the CPU in Figure 4-14 with the circuit schematic in Figure 4-7. Be sure that you can identify the corresponding subcircuits and bus/control lines in each diagram.



**Figure 4-14.** Block diagram of the brainless microprocessor architecture.

Notice that the block diagram contains many fewer details and has a different layout than the circuit schematic, but provides information about the data flow and control-signal flow that is consistent with the circuit schematic. The advantage of the block diagram is that the absence of details helps the reader get an overview of the whole without the clutter and distractions attendant with the details of a circuit schematic. Block diagrams are something that can help you make your report reader-centered, rather than writer-centered.

The circuitry outside of the CPU in Figure 4-14 includes the address decoder and memory cells you built in Task 4-6 through Task 4-9. Note that the details of the memory you designed are absent in this diagram.

Using the block diagram of Figure 4-14 you can begin to understand the sequence of control operations and information flow needed to read from memory, and write to memory.

## Reading an Operand from Memory

When reading from memory, the first control line we need to check is the accumulator buffer control. We want the buffer to be in high impedance mode so that there can be no data-bus conflict between data in the

accumulator and data selected in memory. Next we need to select the memory address that we wish to access. We need to be careful to select a memory location that is compliant with reading. For instance, trying to read from an output port will not yield successful results. (Once you construct your memory circuit, you will know which addresses correspond to ROM, RAM, and the output port.) Having selected an appropriate address, the address decoder shown in Figure 4-14 will assert one of its output lines to activate EN2 of one of the memory location. To read from this location, we must bring the *Read* line high; this will make active EN1 of all readable memory.

At this point, the data from the memory location selected is driving the data bus and, via the load path, is driving the ‘A’ input of the ALU. To load this information into the accumulator (ACC), you must control the ALU to pass-through the ‘A’ operand to its output, ‘Y’, and consequently to the input of the accumulator. (You may want to refer to the ALU function definition table you constructed in the previous lab exercise to determine which ALU-control-input settings are needed to put the ALU in pass-through mode.)

The last step is to enable the accumulator through the line attached to its enable input, then cause the clock to output a negative edge to load the accumulator. Data has now been moved from the memory to the accumulator and is driving the ‘B’ input of the ALU.

## Performing an Operation with Two Operands

Moving one word from memory to the ACC is the first step in performing a two-operand operation. The second step is to allow another data word to be supplied from the memory to the ‘A’ input of the ALU via the data bus using the control sequence described above. Then the ALU can be controlled to perform an ADD, XOR, SUB, etc., on the operands at the A and B ports of the ALU. The result of the operation can be stored in the accumulator, replacing the existing operand and, if desired, written into memory through the store path.

## Writing to Memory

To write to memory, you must first ensure that the *Read* line is low; otherwise the ACC data and the data from the selected memory may cause a data-bus conflict. Next, you must enable the accumulator buffer so that the accumulator output lines have access to the data bus. At this point, the data from the accumulator is driving the data bus via the store path. Next you need to select the memory address to which we wish to write. You need to be careful to select a memory location that is compatible with writing. For instance, trying to write to ROM will not yield successful results. After selecting an appropriate address, the address decoder shown in Figure 4-14 will assert one of its outputs; this line will activate EN2 of one of the memory locations. To write to this location, you must bring the *Write* line high, which will assert EN1 of all write-capable memory cells. To load the information on the data bus into the memory, you must cause

the clock to output a negative edge. (Note that in Figure 4-14 the *Clock* signal driving the accumulator and memory are understood to be the same signal; there is only one clock signal synchronizing all operations of our microprocessor.)

## The Brainless Microprocessor

The LogicWorks™ schematic circuit of the brainless microprocessor is shown in Figure 4-15. The left side of the diagram contains the CPU you built in Task 4-3, the top center contains the addressing circuitry you built in Task 4-6, the bottom right contains four 4-bit ROM memory cells you built in Task 4-7, and the bottom center contains the output port and 4-bit RAM cell you built in Task 4-8 and Task 4-9. Notice that because one enable of each memory cell is connected to a different memory-cell-select line from the address decoder, each memory cell occupies a different address in the memory-address space. Verify for yourself that addresses 0H – 3H access ROM, address EH (14 decimal) accesses the output port and FH (15 decimal) accesses the RAM location.

## Task 4-10: Build the Brainless Microprocessor

Build the circuit shown in Figure 4-15. Do not panic when you look at this circuit! You already have everything you need to wire up this circuit. It is really not too hard to build, and it has some very promising capabilities.

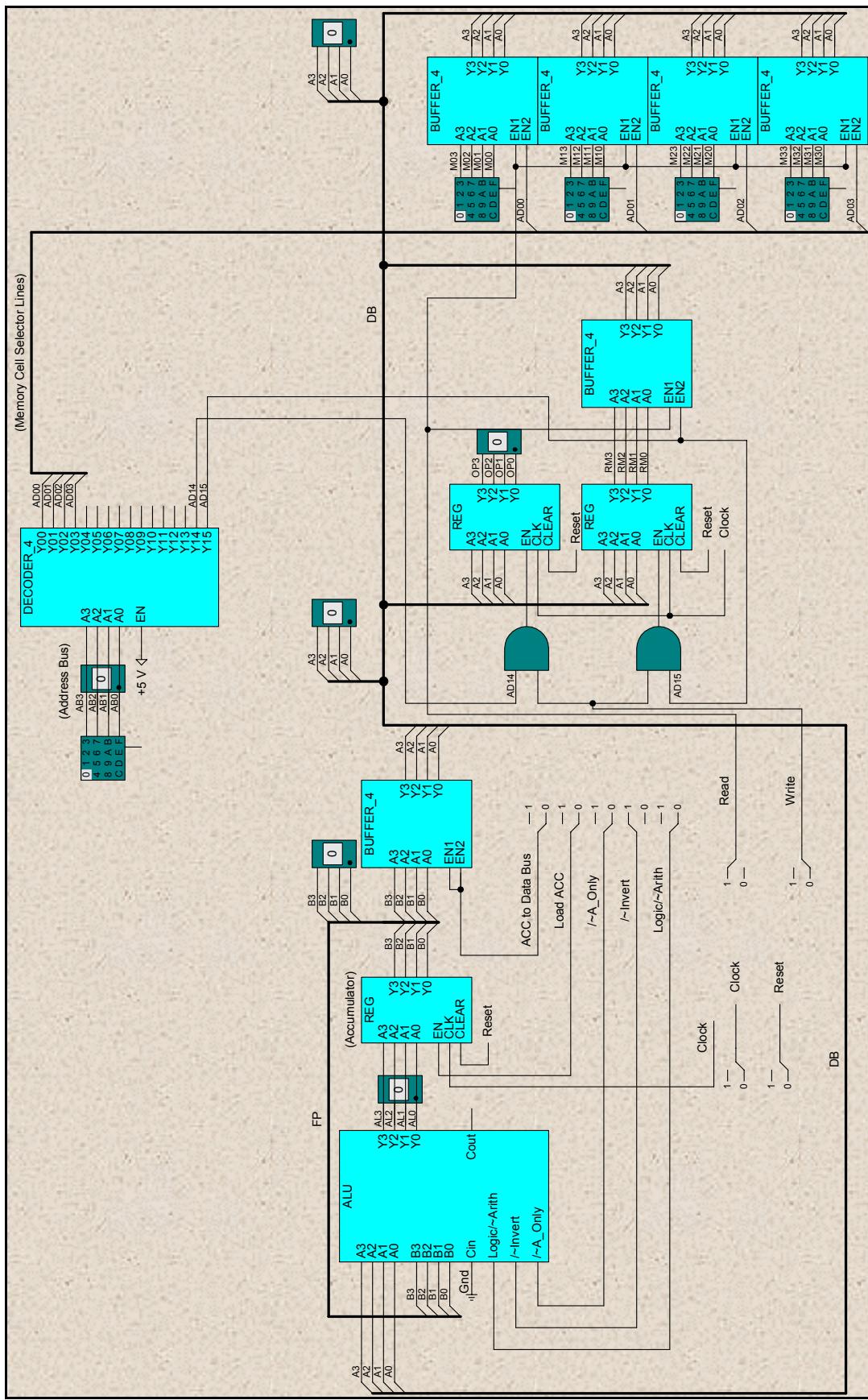


Figure 4-15. The brainless microprocessor.

## Controlling the Brainless Microprocessor

The circuit in Figure 4-15 is almost a microprocessor; it is only missing the ‘brains’, or more frequently called the controller or control circuitry. In the next laboratory exercise you will build a controller for your microprocessor; in this laboratory exercise, the objective is for you to get familiar with how the microprocessor is controlled, by acting as the controller, so that in the next laboratory exercise, the controller design will make more sense. For now, you will function as the ‘brains’ of the microprocessor, by manipulating the control lines to get it to process the data.

If you have ever wondered how a computer works, you will come very close to understanding how it works as you act as the controller when performing the next task.

## Task 4-11: Testing and Controlling the Brainless Microprocessor

In this task, you will test whether your brainless microprocessor functions as desired. Let’s conduct one test to validate its performance; let’s add two numbers, 3 and 5, and observe the sum, 8, in the accumulator. To complete this test, perform the following steps in order:

### 1. Entering Data in ROM Memory

- Enter 3 in ROM at address 0, and 5 in ROM at address 1. (Remember you do this using your mouse and clicking on the hex keyboards at these respective locations.)

### 2. Getting the First Operand (Load ACC with 3)

- Set the *ACC to Data Bus* line switch to 0 to prevent data bus conflicts.
- Bring the *Read* line high, while keeping the *Write* line low. (This will bring the EN1 input of all ROM cells high.)
- Enter 0 in the hex keyboard connected to the address bus. (This will assert EN2 of ROM memory cell 0. EN1 was already asserted when you brought the *Read* line high.) At this point you should observe the value ‘3’ on the hex display connected to the data bus.
- Put the ALU in ‘pass-through’ mode. (Use the same ALU control setting you used in Task 4-4.) At this point you should observe the value ‘3’ on the hex display connected to the output of the ALU.
- Set the *Load ACC* switch to 1. (This will enable the accumulator to be loaded.)
- Make sure the *Reset* switch is reset to 0. (This will allow the ACC [and RAM/Output Port] to accept data under synchronous control.)
- Click the *Clock* switch through one complete cycle to load the accumulator. You have now loaded the accumulator with ‘3’. The value ‘3’ should be viewable on the hex display connected to the output of the accumulator.

### 3. Getting and Adding the Second Operand (ADD 5 to ACC)

- Make sure the following switches remain in their previous positions:
  - *ACC to Data Bus* switch is reset to 0,
  - *Read* line is set to 1,
  - *Write* line is reset to 0,
  - *Load ACC* line is set to 1,
  - *Reset* switch reset to 0.
- Enter 1 in the hex keyboard connected to the address bus. (This will assert EN2 of ROM memory cell 1. EN1 was already asserted when you brought the *Read* line high.) At this point you should observe the value ‘5’ on the hex display connected to the data bus.
- Put the ALU in ADD mode. (You may want to refer to the ALU function definition table you constructed in [Simulation Lab 3](#).) At this point you should observe the value ‘8’ on the hex display connected to the output of the ALU.)
- Click the *Clock* switch through one complete cycle to load the accumulator. You have now completed an addition and loaded the accumulator with the result, 8. The value ‘8’ should be viewable on the hex display connected to the output of the accumulator.

### 4. Storing Result to Memory (Store ACC to RAM, or, Store ACC to Output Port)

- The last step is to store this result into memory. You can store to the output port, or the RAM. The steps needed to store this result into memory are left for you to discover.

## Your First Program

Congratulations! What you have done in the preceding steps is to act as the microprocessor’s controller to execute a three-instruction program:

- Load ACC with 3
- ADD 5 to ACC
- Store ACC to RAM (or Output Port).

Take a moment to revel in your own success and congratulate yourself!

After you celebrate, ask yourself where the program you just executed was written and how our microprocessor knew how to execute each of the instructions in our program. The answer, of course, is that the program we chose to execute resided in our heads and, because we also acted as the controller, we knew how to control all of the switches to execute each instruction. In the next laboratory exercise, we will design a controller that will be able to automatically view instructions (in the form of a program) stored in ROM and then automatically perform the operations needed to carry out each instruction.

## Other Tests

With the execution of the above program you have tested some of the capabilities of your brainless microprocessor. What other tests do you feel are necessary to provide evidence that your circuit operates as designed? Describe the tests you perform and enter the result in your laboratory notebook.

The following is the minimum set of tests you need to perform:

- Perform an XOR operation on two numbers instead of addition.
- Subtract two numbers by taking a two's complement of the first number and then adding it to the second number. (Do not worry about overflow or carries.)
- Test all of the control lines in your LogicWorks™ circuit and make sure they are delivering their signals to the appropriate subcircuit and that all subcircuits are responding correctly to the input control values. (Remember to record the results of all tests in your laboratory notebook.)

For all of the instructions you perform, e.g., Load ACC, Negate, XOR, ADD etc., create a table listing the values to which all control lines must be set during every clock cycle of each step of each instruction you test. (This information will be useful to you when you design your controller in the next laboratory exercise.)

It is important that you stop and make sure that you understand how this circuit works. If you do not understand how this circuit operates, you will not be able to implement the controller in the next laboratory exercise.

## Control/Display Panel

You have probably noticed that it is a bit inconvenient to be scrolling around your LogicWorks™ document to view different hex displays and toggle control switches. As we begin to use our microprocessor more and build on it, we will find it convenient to have all of the input control switches and bus/memory/register signals located in one location.

### Task 4-12: Creating a Control/Display Panel

Create the control/display panel for your microprocessor as shown in [Figure 4-17](#). Be sure to check that the names you apply to each hex display or switch signal in the display/control panel are the same as specified in schematic of [Figure 4-15](#) or [Figure 4-17](#). Remember: if you apply the same name to two different LogicWorks™ wires, LogicWorks™ makes an invisible connection between the two different wires. (If you are unsure how to name a wire, refer to Simulator Tutorial: [Using LogicWorks™ 4 for Windows®](#).) Also notice that you must label the bus breakouts after creating them from the *Schematic*→*New Breakout...* menu.

To change the orientation of labels you apply to pins to match that in the control panel shown in [Figure 4-17](#), right click on the pin name and select the ‘rotate left’ item in the drop-down menu. Next, right click on the pin name, select the ‘Justification’ menu item, click the middle-right pinhole in the ‘Text Justification Dialog Box’ as shown in [Figure 4-16](#), and then click OK. If you wish to enclose the control panel in a frame (as shown), use the bus tool to draw appropriately sized rectangles, and then drag the borders of these rectangles toward each other until they touch. (There is some empty space in this control/display panel. We will be modifying this panel in the next laboratory experiment and adding items to fill the empty space.)

**LogicWorks™ Note:** After you create the bus/hex-display item in the control/display box, you may need to stop then restart the simulator in order for the hex display to reflect the current hex value of the bus signals.

After you have completed the control/display panel, verify that it is operating correctly by trying to use your processor to add two numbers as you did in Task 4-11. You will find it much easier to control and view the operations of your microprocessor with this enhanced control panel.



Generated using LogicWorks™ 4 (Capilano Computing Systems, Ltd.)

**Figure 4-16.** Text justification dialog box.

### Task 4-13: Backup Your Files

Take a moment and backup your libraries and circuit files on a new floppy disk and mark it ‘SimLab#4’. Do this using a different floppy disk after every lab and you just may save yourself much unnecessary work.

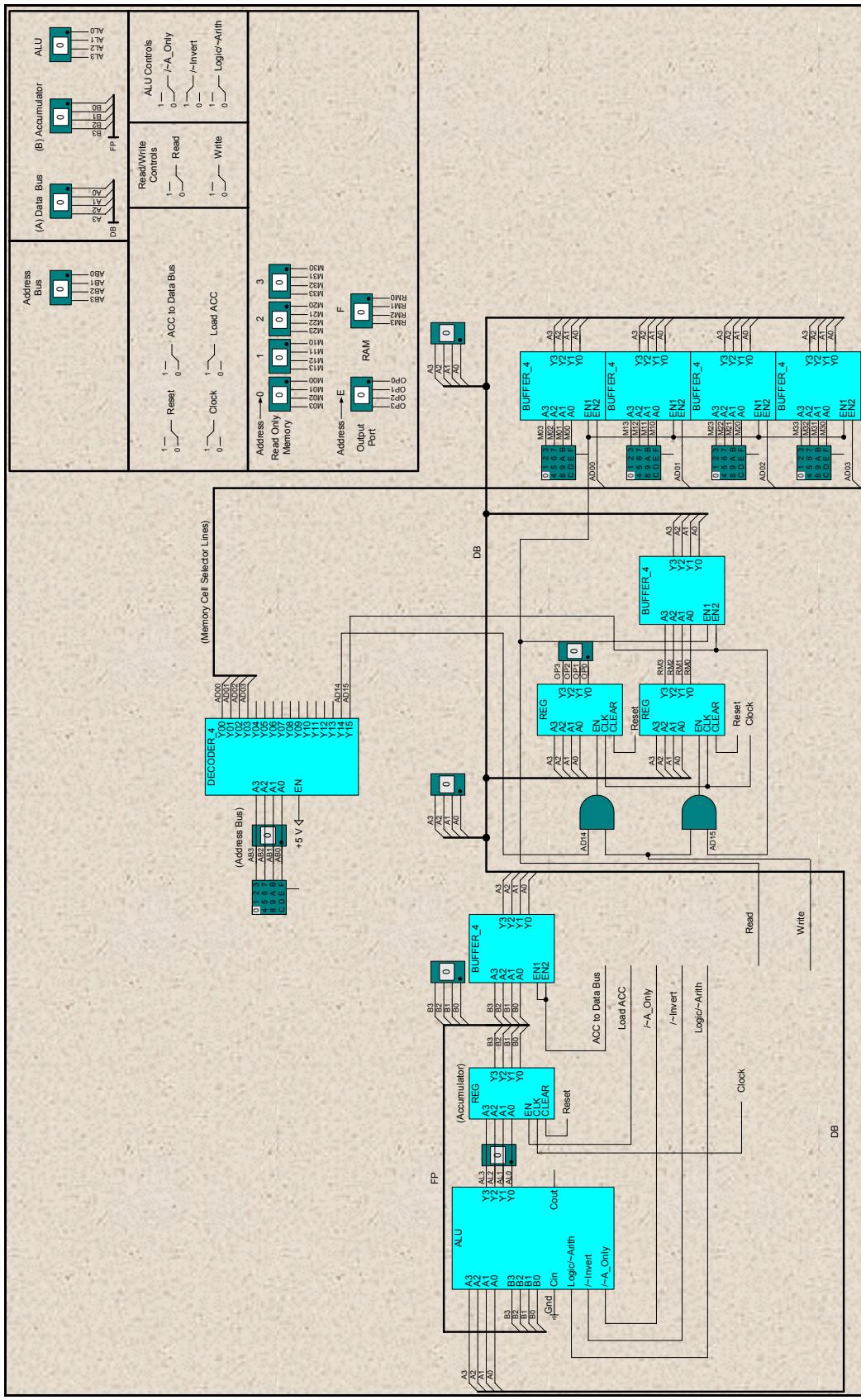


Figure 4-17. Brainless microprocessor with display/control panel.

# SIMULATION LAB 4: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Top-Down Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Top-Down Description of Work Performed</b>		
Brainless Microprocessor: Functional Description and Test Results		
CPU: Functional Description and Test Results		
Accumulator: Functional Description and Test Results		
Buffer Circuits: Functional Description and Test Results		
ALU: Functional Description and Test Results		
Memory: Functional Description and Test Results		
Addressing Circuit: Functional Description and Test Results		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>		
Points Lost		
Late Lab		
Lab Score		

## Report Writing Reminders:

<p><i>Caveat emptor:</i> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used</p> <ul style="list-style-type: none"> <li>• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.</li> <li>• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.</li> <li>• <b>Label</b> all figures or circuits. (E.g. Figure 1. Schematic of . . ., Figure 2. . ., etc.)</li> <li>• Refer in the text of your report to all circuits or figures that you include in the body of your report.</li> <li>• Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.</li> </ul>
--

# SIMULATION LAB 4: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Task Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 4-1: Build the 1-Bit and 4-Bit Registers		
Task 4-2: Build a 4-Bit Buffer		
Task 4-3: Build the Brainless Central Processing Unit		
Task 4-4: Test and Control the Brainless Central Processing Unit		
Task 4-5: Calculating the Maximum Clock Frequency		
Task 4-6: Build the Addressing Logic		
Task 4-7: Build a 4-Bit ROM Memory Cell		
Task 4-8: Build 4-Bit Output Port		
Task 4-9: Build the 4-Bit RAM Cell		
Task 4-10: Build the Brainless Microprocessor		
Task 4-11: Testing and Controlling the Brainless Microprocessor		
Task 4-12: Creating a Control/Display Panel		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

<p><b>Caveat emptor:</b> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used</p> <ul style="list-style-type: none"><li>• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.</li><li>• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.</li></ul>
--

- |  |
|--|
| • <b>Label</b> all figures or circuits. (E.g. Figure 1. Schematic of . . . , Figure 2 . . . , etc.)        |
| • Refer in the text of your report to all circuits or figures that you include in the body of your report. |
| • Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.          |

## SELF-ASSESSMENT WORKSHEET

Put 'X's' in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use '5' to indicate that you 'strongly agree', '3' to indicate that you are 'neutral', and '1' to indicate that you 'strongly disagree'. Use 'NA', 'Not Applicable', when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your **responses will not be graded**. They are for your instructor's information only.

Table \_\_: Self-Assessment of Outcomes for Simulation Lab 4: The Brainless Microprocessor.

After completing the assigned tasks and report, I am able to:	5	4	3	2	1	NA
Build and debug a simulation of a 4-bit register.						
Build and debug a simulation of a 4-bit buffer.						
Build, debug and control a simulation of a central processing unit (CPU).						
Build, debug and control a simulation of a ROM, RAM and an output port.						
Build, debug and control a simulation of an address decoding circuit.						
Build and debug a simulation of a microprocessor that is absent a controller.						
Act as the controller for an elementary microprocessor.						
Calculate the maximum clock frequency that may be applied to a synchronous circuit.						

Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

# SIMULATION LAB 5: THE COMPLETE MICROPROCESSOR

---

**Prerequisites:** Before beginning this laboratory experiment you must:

- Be able to use LogicWorks™.
- Be familiar with Mealy machine design techniques. (Necessary only for the classical approach.)
- Be able to describe the meaning of flip-flop set-up and hold times.
- Have completed [Simulation Lab 1: Half Adder, Increment & Two's Complement Circuit](#).
- Have completed [Simulation Lab 2: 4-Bit Full Adder, Multiplexer & Decoder](#).
- Have completed [Simulation Lab 3: Arithmetic and Logic Unit](#).
- Have completed [Simulation Lab 4: The Brainless Microprocessor](#).

**Equipment:** Personal computer and LogicWorks™.

**Objectives:** In this laboratory you will complete the design of a microprocessor, create an instruction set, enter a program into memory and execute the program.

**Outcomes:** When you have completed these laboratory experiments you will be able to:

- Design a PROM-based controller for an elementary microprocessor.
- Create an instruction set for an elementary microprocessor.
- Use the language of your instruction set to create a program and enter it into memory.
- Execute a program on your simulated microprocessor.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab it is recommended that you use a top-down format when organizing your lab report. (If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Top-Down Report Writing Guidelines](#).)

---

## Introduction

In [Simulation Lab 4](#), you completed testing the brainless microprocessor of Figure 5-1 by acting as the controller. In this lab exercise you will complete the microprocessor design by building a controller, defining an instruction set for the controller, and adding the controller to your brainless microprocessor.<sup>1</sup> You will then use the language inherent in your instruction set to create a simple program, enter the program in your microprocessor's memory and execute the program.

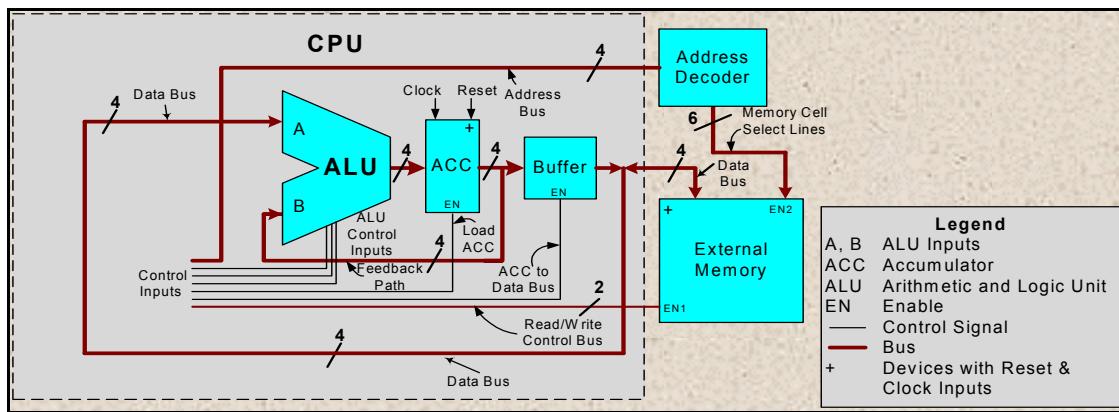


Figure 5-1. Brainless microprocessor architecture.

## Task 5-1: Add More Memory to Your Microprocessor

The microprocessor you designed in the previous laboratory exercise had four external ROM memory locations. To implement the program you will create in this laboratory exercise, you will need at least 8 external ROM memory locations. Add at least four more ROM locations that span address spaces 4H through 7H and add hex displays in your control/display panel that reflect the contents of the ROM locations as shown in Figure 5-2.

You may add ROM to a maximum of 15 ROM locations, addresses 0H thought EH. You cannot assign the address FH to a ROM cell without creating a potential bus conflict because the address FH is occupied by the simulated RAM cell. (Examine the connection between the address decoder and simulated RAM cell in Figure 5-2 to justify that this is the case.) Even though the address EH is occupied by the output port, you can still double assign it to a ROM cell without experiencing a conflict under either a write or a read operation. Recall that an external ROM memory cell has no input connection from the data bus; hence you can write to the output port without overwriting the data in the simulated ROM. Further, since the output port has no output connection to the data bus you can read from the simulated ROM without fearing a bus conflict from the output port. Regardless of the amount of ROM you add to your system, it is imperative that the address space spanned be contiguous because the program-addressing scheme we will

<sup>1</sup> For an overview of this laboratory exercise refer to the PowerPoint lecture slides contained on this CD in the files labeled 'ROM Based Synch Machines', 'Controller Design', and 'The Complete Microprocessor.'

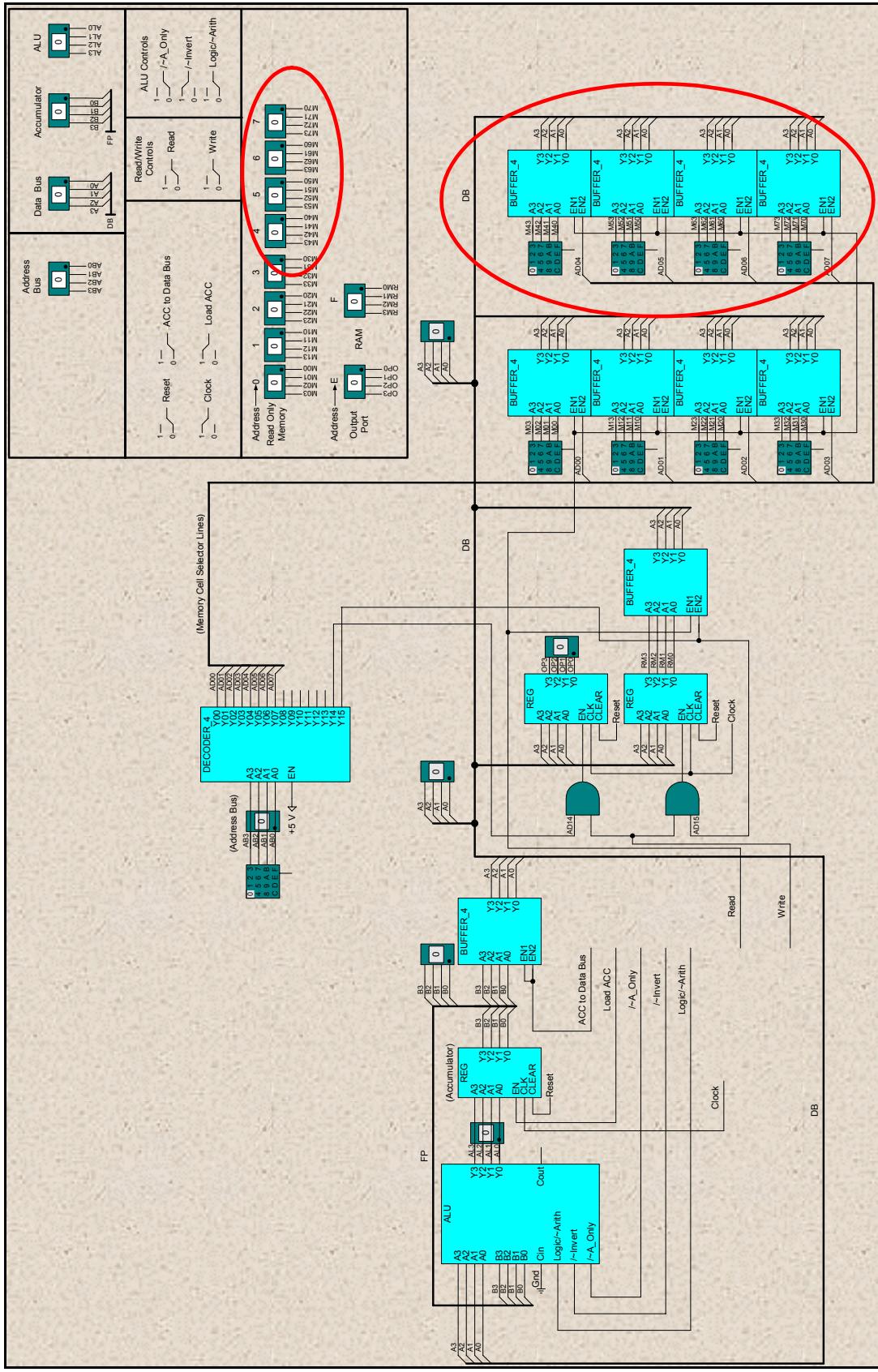


Figure 5-2. Schematic of brainless microprocessor showing added memory.

develop will assume that any program we create is laid out using consecutive memory addresses. That is, the first memory location you create should be accessible using address 0 and subsequent memory additions should be accessible using consecutively greater address values, 1, 2, 3, etc.

## Placement of Instructions and Data in Memory

The program you will enter into external ROM memory will be made up of two conceptually different types of entries: data and instructions. There are two traditional ways of storing these values in memory; each requires a different type of architecture to accommodate its needs. One storage method separates instructions from the data upon which they operate and stores them in separate memory blocks. With this scheme, data and instructions have independent paths by which they reach the central processing unit (CPU). The architecture that supports this communication scheme is known as the Harvard architecture. By contrast, in the Princeton architecture, data and instructions are stored in the same memory block; both data and instructions use the same path to the CPU. From your experience with our architecture, you may have already guessed that our design is based on the Princeton architecture. There is only one data bus in our architecture to carry information from memory to the CPU; hence in our microprocessor, instructions and the data upon which they operate will be intermingled in the same memory block.

When we talk about instructions and data, it is important to distinguish between the terms instruction, opcode (short for operation code or instruction code), and operand. An instruction consists of an opcode plus possibly one or more operands. An opcode is the portion of the instruction that uniquely identifies its function. The operand is the data or address that the opcode uses in performing its functions.

(a) Instruction with No Operand		(b) Instruction with One Operand	
Address	Contents	Address	Contents
Q	Opcode	Q	Opcode
Q+1	Next Opcode	Q+1	Operand
Q+2	Etc.	Q+2	Next Opcode
Q+3		Q+3	Etc.
Q+4		Q+4	

(c) Instruction with Two Operands

(c) Instruction with Two Operands	
Address	Contents
Q	Opcode
Q+1	Operand
Q+2	Operand
Q+3	Next Opcode
Q+4	Etc.

Figure 5-3. Possible memory layouts for instructions.

Some of the instructions you will enter into the external ROM will require no data to operate upon. For example, the ‘Stop’ instruction needs no operand. If we create a program consisting entirely of such instructions, the layout of the program in memory will be like that shown in Figure 5-3 (a). As shown in this figure, program instructions to be executed are entered into consecutive memory locations with successively greater addresses.

Some instructions, such as ‘Load ACC with 3’, use one piece of data to operate upon, namely ‘3’. A program that includes one-operand instructions will have the layout shown in Figure 5-3 (b). Assuming our program starts in location Q, the opcode that uniquely identifies the instruction to be performed will be located at address Q, and its operand will be located at the next greater address, Q+1. The opcode for the next instruction will be located at address Q+2, etc. Figure 5-3 (c) shows the opcode and data layout in memory for the case of two operands; the layout for three- or four-operand instructions uses the same philosophy.

## Program Counter and Address Generation

It is possible to design a computer architecture where instructions to be executed sequentially are located at arbitrary locations in memory. Each instruction in such a system is comprised of an opcode and at least one operand. The needed operand serves as a pointer to the memory address of the next instruction to be executed. Programs stored in this fashion require a large amount of memory because of the pointer operand that must be part of every instruction.

The more common way of storing instructions (to be executed sequentially) is to enter them into memory locations with successively greater addresses. When we use this strategy, we don't need to tell the controller where the next instruction is located. The controller assumes that the next instruction is located at the next sequential memory address—after accounting for the presence of any operands—and that any store operation is likewise directed at the next sequential memory location. This assumption is violated when we encounter jump or branch instructions, or when we wish to store the result of an operation to an arbitrary memory location; in those cases, we need to store a pointer with each branch instruction to point to the location to which the program should jump and we need to store a pointer with each store instruction to point to the location to which the result is to be written.

If we wish to allow our processor to be able to access memory locations both in and out of sequential order, our processor will need circuitry that can:

- Automatically increment memory addresses after each instruction or operand is retrieved by the CPU.
- Change the memory reference address so that we can store data to arbitrary memory locations.

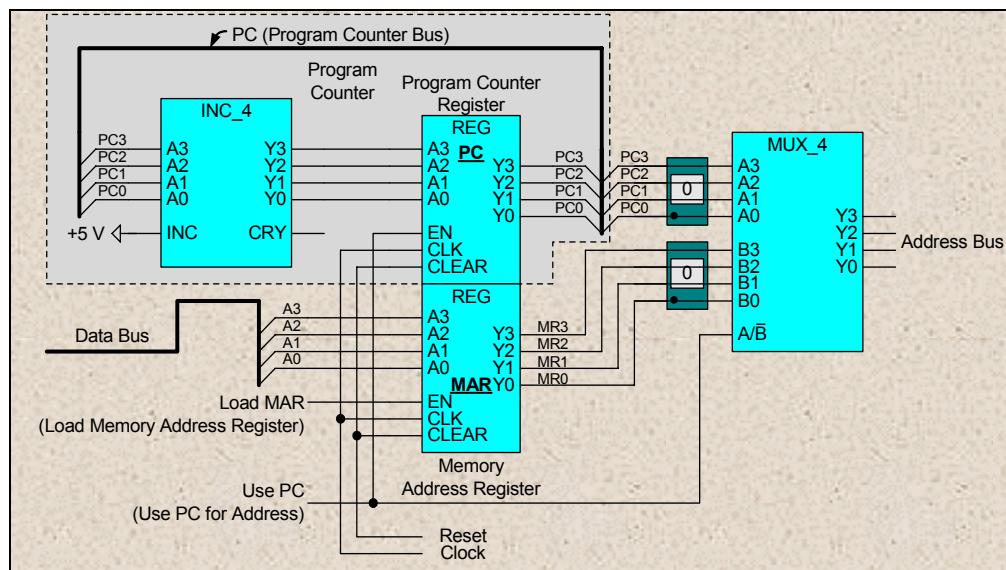
The circuit that can perform both types of address generation is shown in Figure 5-4. (This circuit does not allow our processor to execute a branch/jump instruction. Modifying this circuitry to allow a branch/jump to take place is the focus of Simulation Labs 6 and 7.)

The address generation circuit of Figure 5-4 is comprised of two independent pieces. One piece consists of a register and increment circuit and is known as the program counter (PC). The other piece, which accepts addresses from the data bus, is known as the memory address register (MAR). The output of the address generation circuit is a mux, which controls whether the PC or MAR drives the address bus. The control input to the mux, *Use PC* (short for *Use Program Counter for address*), is high when the PC values are to drive the address bus and low when the MAR contents are to be supplied to the address bus. Let's look in more detail at the operation of the PC and the MAR.

### The PC

As its name implies, the program counter (PC) is simply a counter; it increments on every clock pulse when enabled. This counter is made up of the 4-bit register and the 4-bit increment circuit that you added to your library in Simulation Labs 4 and 1, respectively. When enabled by the *Use PC* control signal, each clock pulse causes the PC register to be loaded with the output of the increment circuit, which is always greater, by 1, than the current contents of the PC register. Each time the counter increments, the address of the next sequential memory location will be generated and the next opcode or operand, in external memory, accessed.

The other input to the PC is the *Reset* control. By setting the *Reset* line briefly to 1, and then resetting it back to 0, we will be resetting the PC and MAR, thus driving 0 onto the address bus. This causes memory location 0 to be accessed. If we store our programs beginning at memory location 0, then the *Reset* control will provide a convenient method of initializing our microprocessor so that it executes our program starting with the program's first memory location.



**Figure 5-4.** Memory address generation circuit.

### The MAR

The lower register in Figure 5-4 is the memory address register (MAR). This register, when enabled by a high signal on the *Load MAR* (*Load Memory Address Register*) line, is loaded with data contained on the data bus upon receiving a *Clock* pulse. The contents of the MAR drive the address bus when the *Use PC* control line is low. The MAR and addressing-circuit architecture allows an executable instruction to access any desired address by first loading that address into the MAR and then driving the address bus directly with the MAR. For example, the instruction ‘Store ACC to EH’ must be stored in memory in the following way:

Address	Contents
Q	Opcode for ‘Store ACC’
Q+1	EH

When you design your ‘Store ACC’ instruction, you will make it sophisticated enough that after fetching the opcode for ‘Store ACC’ from program memory, your instruction will load the address operand EH into the MAR. (Your instruction will also cause the controller you will build to bring the *Use PC* line low so that the address bus is driven by the MAR contents.)

### Task 5-2: Build, Test and Debug the Memory-Address-Generation Circuit

Build the circuit shown in [Figure 5-4](#). Be sure to label the signals at the output of the PC and the MAR as shown. Also label the PC bus ‘PC’, as shown. Add switches to drive the lines containing the *Load MAR*, and *Use PC* lines. Use the LogicWorks™ hex keyboard to drive the MAR inputs from the data bus and test the circuit. Record in your notebook the results of your test. Save your circuit, but do not imbed it in a subcircuit.

### Task 5-3: Add the Memory-Address-Generation Circuit to the Processor

Add the memory address generation circuitry to your partially completed microprocessor as shown in Figure 5-5. Also add the PC and MAR displays and the *Load MAR* and *Use PC* controls to your control/display panel. (Remember to check that all of the signal names you use are assigned to the appropriate wires in the memory-address-generation circuit and in your control/display panel.)

Test your partial microprocessor to ensure that the memory-address-generation circuitry functions as desired. Record the results of your tests in your notebook. Remember to save your circuit when you are done with this task.

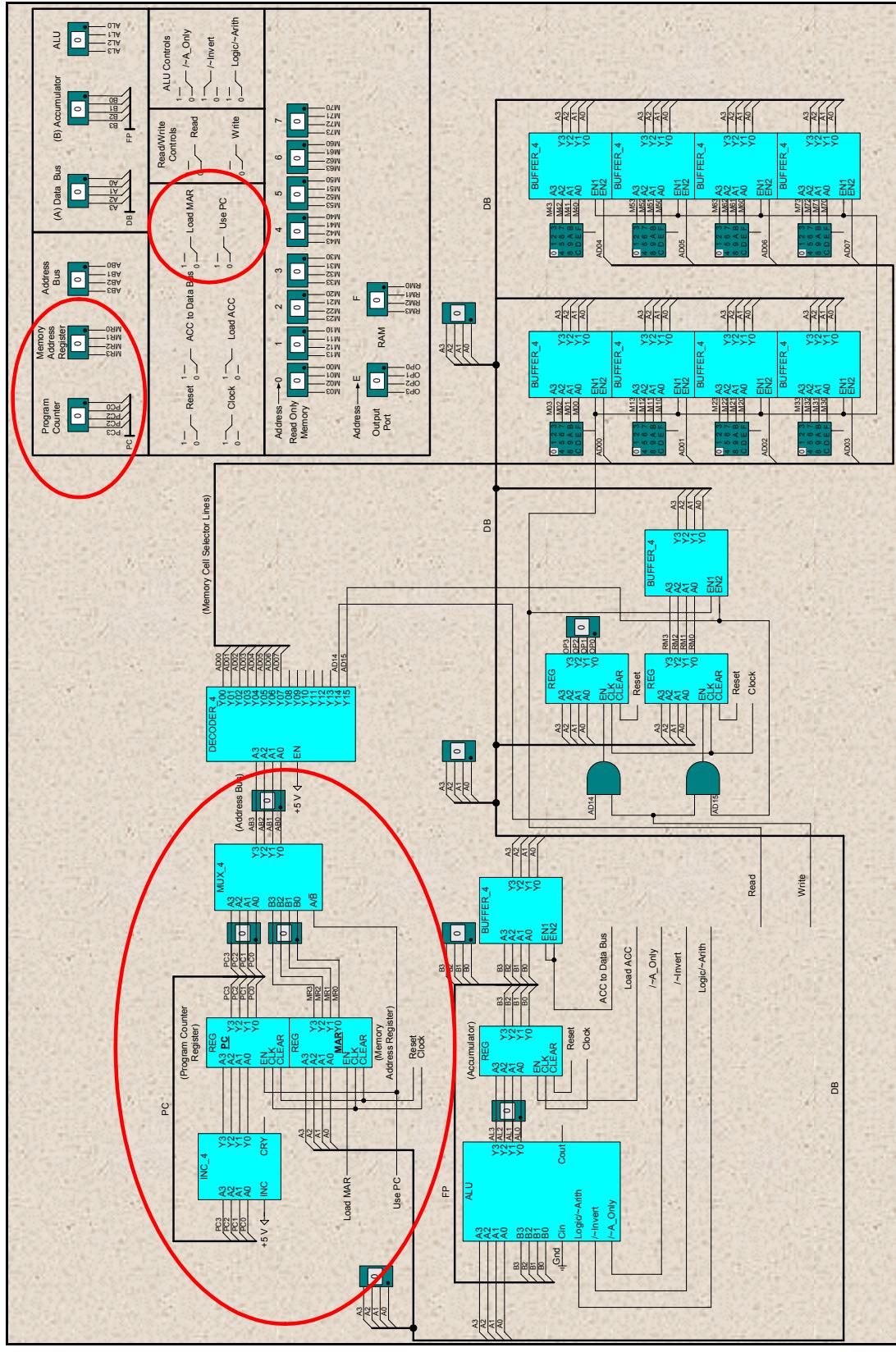


Figure 5-5. Schematic of a brainless microprocessor emphasizing memory-address-generation circuitry.

## Control Signals

### Controller Output Signals

Upon inspecting Figure 5-5, you will notice many toggle switches. These switches control signals that will be controlled by the controller we'll design - except for the *Clock* and *Reset* signal. (We'll continue to use toggle switches to manually control the *Clock* and *Reset* signals.) The signal names, their function, and their active values are listed in Table 5-1.

Table 5-1. Definition of Controller Output Signals.

Control Signal	Function	Active Value
Load IR <sup>2</sup>	Enables IR to be loaded with opcode from data bus.	High
Write	Enables memory to be written into.	High
Read	Enables memory to be read from.	High
ACC to Data Bus	Enables ACC contents to drive the data bus.	High
	Prevents ACC contents from driving the data bus.	Low
Load ACC	Enables ACC to be loaded with ALU output.	High
Load MAR	Enables MAR to be loaded from data bus.	High
Use PC	Selects PC to drive data bus.	High
	Selects MAR to drive data bus.	Low
/~A_Only	Causes ALU to operate on the A & B operands.	High
	Causes ALU to operate on operand at port A.	Low
/~Invert	Causes ALU to operate on A and/or B port operands <u>without inverting either</u> .	High
	Causes ALU to invert the A operand.	Low
Logic/~Arith	Causes ALU to perform a logical operation.	High
	Causes ALU to perform an arithmetic operation.	Low

<sup>2</sup> The *Load IR* control line will be discussed when the operation of the controller is described in detail.

### The Reset Signal

Everyone who has worked with a computer has at some time had the computer ‘lock up’ on them. By that we usually mean that no matter which key we push on the keyboard and no matter how feverishly we move the mouse, the image on the monitor does not change. Once we’ve tried all the alternatives we can think of, we often hit the reset switch on the front of the computer. This reset switch, among other things, sets the PC<sup>3</sup> of our personal computer to point to a location in ROM that contains the instructions that are part of the boot-up sequence. The microprocessor we are simulating has no operating system so we have no need for a boot-up sequence, but our *Reset* signal will perform the same function as the reset switch on your personal computer. By bringing the *Reset* signal high momentarily, we will be resetting our PC register to 0. (When we finish designing the controller we will see that the *Reset* signal will also initialize our *Use PC* signal to 0 so that the PC register will drive the address bus.) The PC will access ROM memory location 0 where the first instruction of our program (which could be a boot-up program if we needed one) will reside. Just as on a personal computer, we will manually control the *Reset* signal by using the toggle switch that drives the *Reset* line in Figure 5-5.

### Controller Design Preliminaries

The only piece of our microprocessor we have left to design is the controller, which will be a PROM-based synchronized Mealy finite-state machine. If you are unfamiliar with PROM-based synchronized Mealy machines, read [Appendix B: PROM-Based Synchronized Mealy Machines](#). This appendix takes you through the steps necessary to realize functions using a PROM rather than discrete components. It then explains how this technique can be applied to simplify the construction of Mealy machines and how the asynchronously changing outputs of a Mealy machine may be synchronized with the clock signal.

The next step is to understand the design requirements specific to our controller and then perform the design. There are two approaches to designing this controller.

- **Classical Approach:** One approach is to use classical design techniques. By classical design techniques, we mean state transition diagrams, and state transition tables. This approach is described in [Appendix C: Controller Design Using Classical Design Techniques](#). The purpose of describing this design approach is to show how the classical design techniques you learned in your course can be used to design our controller. The end result of this design approach will be identical to that described in the top-down approach below.
- **Top-Down Approach:** In the top-down approach, we use ad hoc methods similar to those used in hardware lab 4. We start by looking at our controller as an interaction of major functional blocks and develop a design specification that defines the way each of those blocks performs.

---

<sup>3</sup> In this document, PC will always stand for ‘Program Counter.’ When we want to refer to ‘personal computer’, we will spell it out.

Your instructor may prefer that you follow one or the other of these approaches, or perhaps that you become familiar with both; so check with your instructor before reading on. If you are to follow the classical approach, read [Appendix C](#) then return to this laboratory assignment and begin performing [Task 5-4](#). Otherwise, for the top-down approach, continue with the next section: [Microprocessor Controller Design Using a Top Down Approach](#).

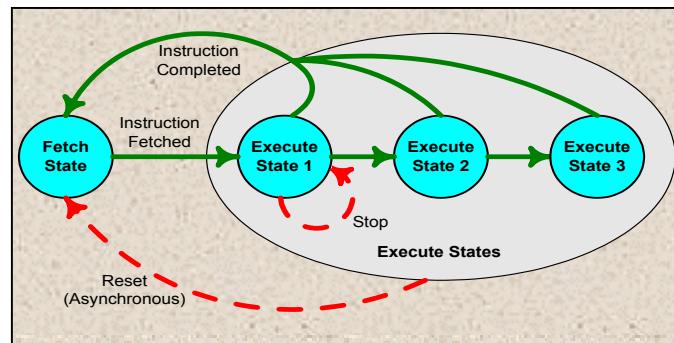
## Microprocessor Controller Design Using a Top Down Approach

### The Fetch-Execute Cycle

The controller we will design will do only three things:

- Fetch an instruction,
- Execute an instruction,
- Halt.

This does not seem like a lot of capability; yet most microprocessor controllers are limited to these three procedures. The capability of the microprocessor arises from the many different instructions it can execute. This cycle of operations is called the fetch-execute cycle and is shown in [Figure 5-6](#).



**Figure 5-6.** The basic instruction fetch-execute cycle.

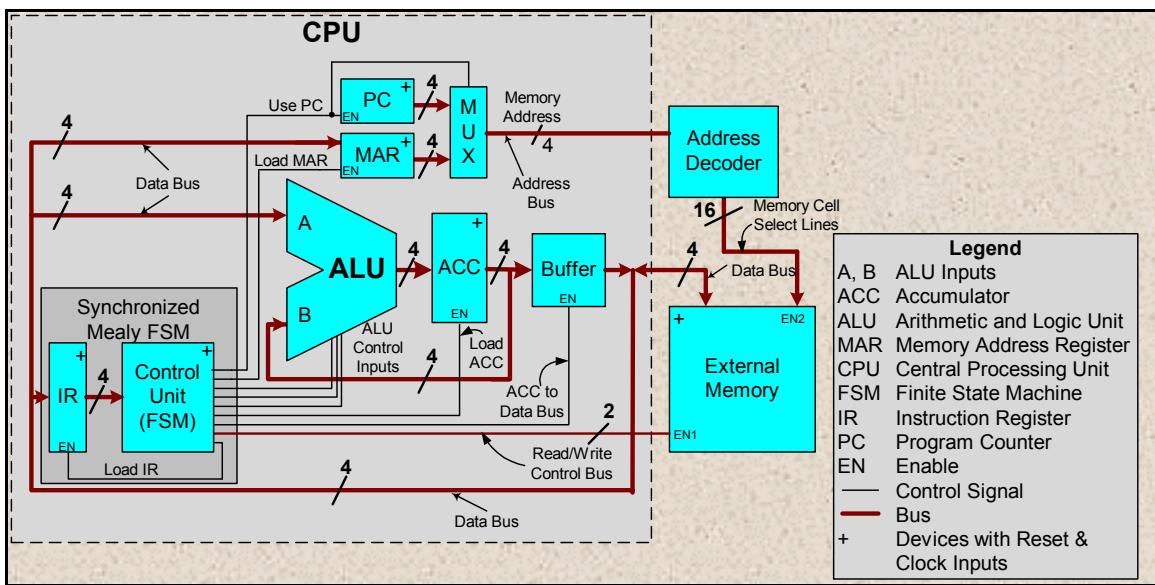
When the controller is in the fetch state, it

fetches the opcode of the next instruction to be executed from the externally stored program (in ROM or RAM). Once the opcode is fetched, the microprocessor enters an execute state. An execute state changes various memory units both internal and external to the CPU. Depending on the complexity of the instruction, the processor may pass through several execute states before the instruction is completed. After execution is completed, the microprocessor returns to the fetch state, unless it encounters the 'Stop' instruction, in which case it enters the Halt state. The 'Stop' instruction in our instruction set will be implemented by forcing the controller into an infinite loop; hence, in our implementation, the Halt state is not a state *per se*, but simply a repetitive indefinite execution of the execute-state 1 shown in Figure 5-6. After entering this infinite loop, (and at any time we wish to terminate execution) we simply assert the *Reset* signal to return the microprocessor to the fetch state.

Observe that to implement the instruction fetch-execute diagram of Figure 5-6, we need a controller with capabilities similar to those of a finite state machine (FSM). Recalling that the power of the microprocessor comes from the many different operations it can perform in each state, you may come to

the reasonable (and accurate) conclusion that the FSM needs to be a Mealy machine; that is, the (control) outputs associate with each execute state need to be functions of the (opcode) input<sup>4</sup>. If we further recognize that the Mealy machine input (i.e., opcode) needs to be constant throughout the entire cycle of states needed to execute a given instruction, we arrive at the block diagram of our controller as shown in Figure 5-7. This diagram shows that our controller is a synchronized, Mealy FSM. The instruction register (IR) acts to synchronize and store the opcode input (which it receives from the portion of external memory used to store a program), so that it can be held constant while the Mealy FSM cycles through the states needed to complete the operations consistent with the opcode.

Figure 5-7 hints, for the first time, at how the control unit interacts with the rest of the CPU. If you inspect the lines coming from the control unit, you will see that they touch every block that is part of the CPU. They control the registers, buffers, muxes, decoders, and the ALU of the CPU. They also control external memory operations through the *Read/Write* lines and through control of the addressing circuitry. You can see how central the controller is to the operation of our microprocessor.



**Figure 5-7.** Microprocessor architecture.

Notice that the clock signals, which load the registers at each clocking event, are not shown in Figure 5-7 to avoid the complexity that these lines add to the figure. Instead, the existence of a clock (and reset) input to each register is indicated by a '+' sign in the register block. Notice also that, because we are using the Princeton architecture, one data bus is used to move all data from memory to locations within the CPU.

Before we design the synchronized Mealy FSM controller, we need to understand how this controller performs the fetch-execute cycle. Specifically, let's look in detail at the control signals that must be active during

<sup>4</sup> We could design our controller as a Moore machine, but it would require ever so many more states.

the fetch operation. Then we will look at the signals that contribute to control of the execution part of the fetch-execute cycle.

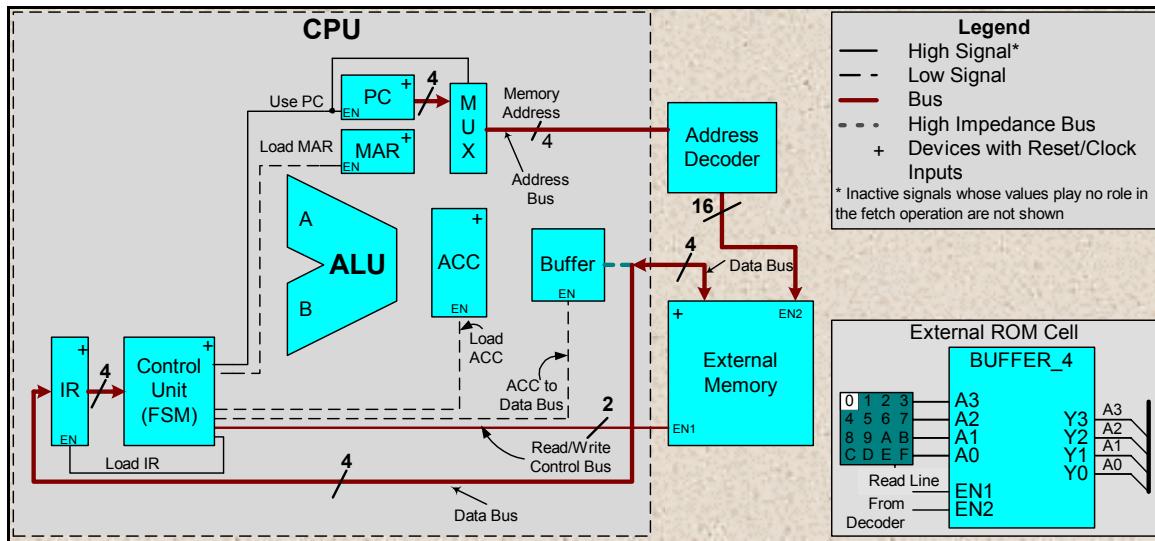
### Instruction Fetch State

In the fetch state, the controller fetches an opcode from external memory and places it on the data bus. Figure 5-8 shows the subset of active control and data lines used to fetch an opcode using the address stored in the PC. To fetch an opcode from memory, the address of the opcode to be fetched must have been stored in the PC by the execution of an earlier instruction. To fetch an opcode using the contents of the PC as an address our controller must:

- Bring the buffer into a high impedance state by bringing to 0 the *ACC to Data Bus* controller line. (Recall that the ACC output is connected to the data bus via this buffer. Without bringing this buffer into a high impedance state, data from the ACC would be driven on the data bus and would conflict with the data being driven onto the data bus from memory.)
- Bring the *Read* line high so that the EN1 input to each memory location is activated.
- Bring the *Write* line low so that no memory location is overwritten by any data driving the data bus.
- Bring the *Use PC* line high. This does two things. First it causes the address-generation mux to select the contents of the PC so that they are made available to the address decoder. The output of the decoder then asserts the EN2 input of the memory cell whose address is stored in the PC. With the EN1 (from the *Read* line) and EN2 inputs both active, the selected memory cell drives its content onto the data bus.

The Use PC line also causes the PC to be incremented on the clock pulse that ends the fetch state. Incrementing the PC at the conclusion of the fetch state allows it to point to the next location in memory, which may contain the next opcode or the operand of the fetched instruction.

- Bring the *Load IR* line high so that the opcode driving the data bus (from the accessed memory location) is loaded into the IR on the *Clock* pulse that **ends** the fetch cycle.
- Bring the *Load MAR* and *Load ACC* lines low so that data in the MAR and ACC are not overwritten by the opcode driving the data bus.
- Set the ALU control lines, */A\_Only*, */Invert*, and *Logic/Arith*, to any value since the results of the ALU will not be loaded into the ACC.



**Figure 5-8.** Fetch state of the microprocessor.

In general, we think of these control signals as performing two types of actions:

- Route data to the proper location. (For the fetch state described above, data is routed as follows: opcode from memory to the IR, and memory address from the PC to the Address Decoder.)
- Control registers so that they are either loaded or not loaded on the *Clock* pulse that **ends** the current execute cycle. (The PC and IR will be loaded in this case, while the contents of the MAR and ACC will be left unchanged by the cycle-**ending** clock pulse.)

### Instruction Execute State

An instruction execute state is entered at the time a *Clock* pulse first loads the opcode into the IR. During the execute state, the controller asserts control lines consistent with the opcode in the IR. Depending on the opcode fetched during the fetch cycle, the set of control lines asserted will vary greatly; hence, unlike the fetch state described earlier, there is no one set of control lines that is activated during every execute state.

Let's look at one example: the control lines active during the execute phase of the 'Add 5 to ACC' instruction. We will see that this instruction will be stored in memory in the form:

Address	Contents
Q	Opcode for 'Add to ACC'
Q+1	Operand = 5H

Let's assume in our discussion that the opcode for 'Add to ACC' has been fetched and resides in the IR. The result of the fetch operation will have incremented the PC so that it points to the operand 5H in memory. Further, we must assume that the value we wish to add to 5H has already been deposited into the

ACC by an earlier instruction. (We'll assume that the previously executed instruction was 'Load ACC with 3.') The controller has four major things to do, all of which it can do simultaneously.

- It must make the operand '5H' available to the data bus.
- It must control the ALU to perform the Add operation.
- It must enable the ACC so that it can be loaded with the sum when the Add is completed.
- It must prevent the MAR and IR from being overwritten by the operand on the data bus, 5H.

Let's look at the control lines that will play a role in this execute cycle.

### Make the operand '5H' available on the data bus

To make the operand available on the data bus the following control lines must be active: (Refer to [Figure 5-9.](#))

- The *Use PC* line must be high to:
  - Route the PC contents to the address decoder and, hence, activate the EN2 input of the selected memory location.
  - Enable the PC register to be incremented on the *Clock* pulse that **ends** the execute state. (This allows the PC to point to the next opcode in memory during the next clock cycle.)
- The *Read* line must be high to activate the EN1 signal to all memory cells, including the memory cell that contains the operand, '5H.' (Consequently the *Write* line must be low.)
- The *ACC to Data Bus* line must be low to prevent the ACC contents ('3H') from conflicting with the operand, '5H', being driven onto the data bus.

### Control the ALU to perform the Add operation

To control the ALU to perform the Add operation, the following control lines must be active:

- $\sim A\_Only = 1$ .
- $\sim Invert = 1$ .
- $Logic/\sim Arith = 0$ .

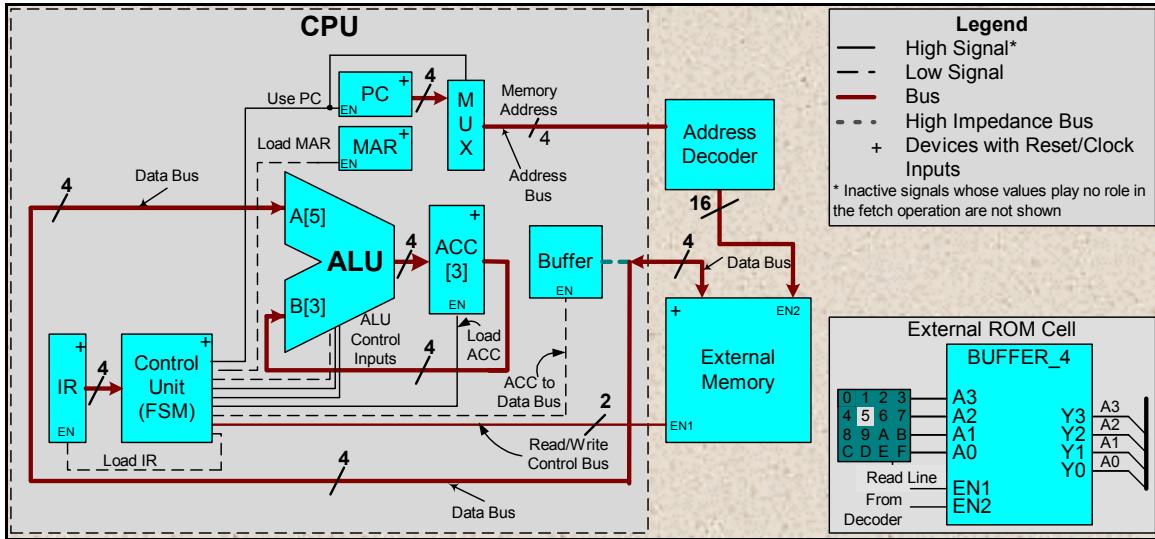
### Load the ACC with the result when the Add is completed

- The *Load ACC* line must be asserted to allow the *Clock* pulse that ends the execute state to load the output of the ALU into the ACC.

### Prevent the MAR and IR from being overwritten

- The *Load IR* line must be low to prevent the opcode in the IR from being overwritten by the data bus value on the *Clock* pulse that **ends** the execute state.

- The *Load MAR* line must be low to prevent any address in the MAR from being overwritten by the data bus value on the *Clock* pulse that **ends** the execute state.



**Figure 5-9.** Execute state of the microprocessor for instruction ‘Add 5H to ACC.’

The control lines active under this scenario, shown in Figure 5-9, perform two types of actions:

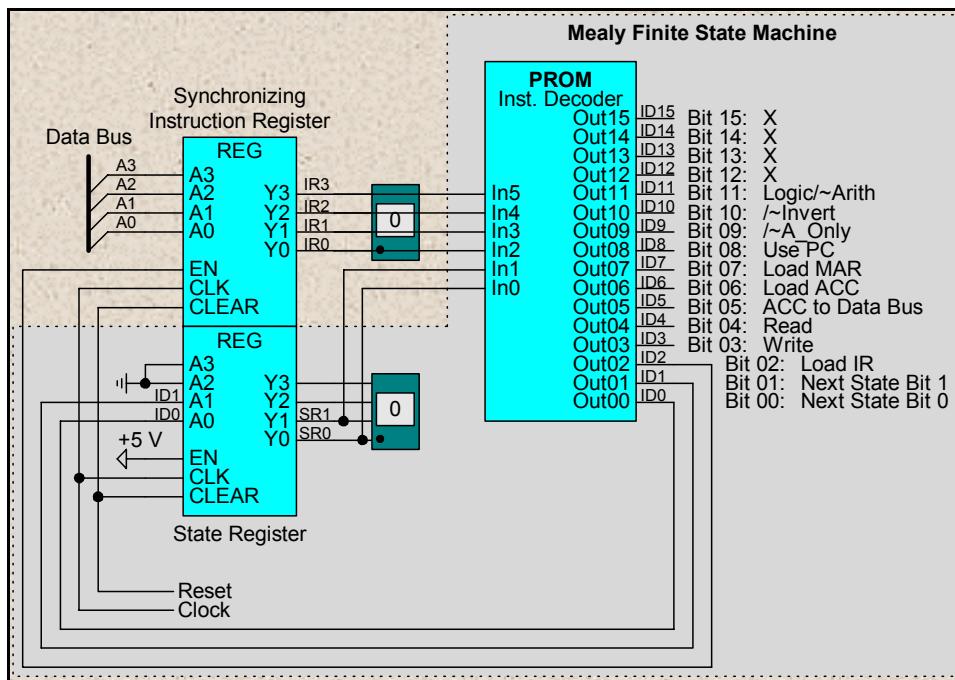
- Route data to the proper location. (For the control line settings described above, data is routed as follows: operand from memory to the ALU A-input port, data through the ALU while performing an Add operation, and memory address from the PC to the Address Decoder.)
- Control registers so that they are either loaded or not loaded on the *Clock* pulse that **ends** the current execute cycle. (The PC and ACC will be loaded in this case, while the contents of the MAR and IR will be left unchanged by the cycle-ending clock pulse.)

### Controller Architecture

Continuing under our assumption that the controller can be based on a synchronized Mealy-machine design similar in architecture to that of [Figure B-11](#), and having defined the controller’s inputs and outputs in [Figure 5-9](#), we are now in a position to justify the architecture of our microprocessor controller shown in [Figure 5-10](#). Review Figure 5-9, and observe that the external input to the Mealy machine controller consists of a 4-bit opcode. (Limiting our bus width and opcodes to four bits means that our processor’s instruction set is limited to at most 16 instructions. This is small, but will suffice for our purposes.) The architecture of Figure 5-10 shows that this 4-bit input is supplied by the data bus to the synchronizing instruction register.

The outputs from our Mealy machine controller will perform two conceptually distinct chores. One group of outputs will provide control inputs to the various blocks of our microprocessor as shown in [Figure 5-7](#). These 10 control signals, listed in [Table 5-1](#) (and shown in Figure 5-10), are the ones you have

heretofore manipulated manually to control the microprocessor. A second group of outputs from our Mealy machine controller are needed to drive the state flip-flop array to achieve the appropriate next state—given knowledge about the present state and external input. Limiting our controller to four states (as depicted in [Figure 5-6](#)) means that we need a two-flip-flop state array to hold the present-state information. Assuming that we use D flip-flops to store the present state, our controller needs two outputs to drive the state flip-flop array, as shown in Figure 5-10. Note that in this figure we are implementing the controller functions using a PROM rather than combinational logic. (Notice also that the PROM in this figure has 4 unused outputs. These outputs will be used in Simulation Labs 6 and 7.)



**Figure 5-10.** Microprocessor controller circuit.

Figure 5-10 is divided into two pieces. Once piece, laid on a flat gray background, is the classical PROM-based Mealy finite-state machine (FSM). In this part of the figure, the output functions of the Mealy-FSM PROM serve to drive the state flip-flop array and provide output functions that control inputs to the subcircuits of the microprocessor.

The other piece of Figure 5-10, the synchronizing IR, serves two functions. One function is to keep the opcode constant for the duration of clock cycles needed by the opcode to fulfill its task. The second function of the synchronizing IR is to ensure that the input to the controller changes in synchronism with the processor's clock. Without the synchronizing IR, the controller would perform erratically. For example, if the controller is executing an opcode, as shown in [Figure 5-9](#), the operand that is being operated upon is driving the data bus. If the IR is absent, then the controller interprets the value on the data bus as an opcode. This false ‘opcode’ causes the output lines of the controller to change, in turn changing (perhaps)

the value driving the address bus, which in turn changes the memory location accessed. This changes the ‘opcode’ the controller receives, which changes the controller’s outputs, etc. Clearly, the synchronizing IR is important to the proper functioning of the controller.

### Designing the Controller

Review [Figure 5-10](#) and notice that completing the design of the controller (at least for the fetch operation and Add instruction we discussed) means specifying the contents of the PROM so that its outputs control the ALU, muxes, decoders, and registers of the processor to perform the operations needed to fulfill the requirements of each opcode/instruction we desire. A tool that will be helpful in defining the outputs needed to perform each instruction is the microinstruction definition table, Table 5-2. We use the term microinstruction here to mean the PROM outputs associate with each state/opcode-input combination. We will classify microinstructions into one of two forms:

- Fetch Microinstruction: Microinstruction associated with state 0 used to fetch from memory the opcode of the next instruction to be executed.
- Execute Microinstruction: Microinstructions associated with states 1, 2, and 3 that perform operations peculiar to the opcode driving the PROM.

**Table 5-2. Fetch Microinstruction Definition Table.**

**Fetch Microinstruction**

		Present State Bits			
Output Bit	Control Line	00	01	02	03
1-0	Next State Bits	01			
2	Load IR	1			
3	Write	0			
4	Read	1			
5	ACC to Data Bus	0			
6	Load ACC	0			
7	Load MAR	0			
8	Use PC	1			
9	/~A_Only	X			
10	/~Invert	X			
11	Logic/~Arith	X			
12	X	X			
13	X	X			
14	X	X			
15	X	X			
Hex Equivalent		0115			

We use the term instruction to mean the sum total of operations performed by the microinstructions associated with each opcode. In these tables we will list, for each opcode of interest, the PROM outputs (contents) needed to effect the controls (i.e., microinstruction) that we desire.

Table 5-2 contains the PROM contents for the fetch state/microinstruction, which we have assigned to be state zero. The fetch state, or fetch microinstruction, is a microinstruction that each of our opcodes need, since one (final) task performed by every opcode is to fetch from memory the opcode of the next instruction to be executed.

Check that the values listed in Table 5-2 match those described in [Instruction Fetch State](#) section. Notice that the ALU control bits are listed in this table as ‘don’t cares’, as are the unused four MSB’s of the PROM. The next-state bits listed in this table are the PROM outputs for the functions that control which state will be entered upon a clock pulse. These bits show that after the fetch microinstruction is executed, the next microinstruction to be executed will be associated with execute-state 1.

To completely design our controller we must create a list of every instruction we want our microprocessor to be capable of, and then fill in the microinstruction definition table with the values that the controller’s PROM will need to complete each instruction. In the next sections, this manual will take you through the steps need to define three different instructions.

### The ‘Load ACC with [operand]’ Instruction

The first member of our instruction set, ‘Load ACC with [operand]’, will appear in program memory in the following way:

**Table 5-3. Program Storage Scheme for Instruction ‘Load ACC with [operand]’.**

Address	Contents	Example
Q	Opcode for ‘Load ACC’	0
Q+1	Operand	3

The opcode for the ‘Load ACC’ instruction can be chosen to be any number from 0H to FH<sup>5</sup>. Arbitrarily choosing the opcode to be 0H=0000B, the instruction ‘Load ACC with 3H’ is stored in program memory as shown in Table 5-3. The ‘Load ACC’ instruction requires only two of the possible four machine states/microinstructions: an execute microinstruction to execute the ‘Load ACC’ operation and a fetch microinstruction to fetch the opcode of the next instruction stored in memory (after the ‘Load ACC’ execute microinstruction has been executed). Refer to Figure 5-7 and prove to yourself that the ‘Load ACC’ execute microinstruction must:

<sup>5</sup> It is necessary to have some instruction assigned to the 0H opcode. Exactly why this is the case will become apparent when we discuss the reset operation.

- Bring the *Use PC* high so that:
  - The PC value (updated by the fetch microinstruction from a previous instruction) drives the address bus, allowing the instruction's operand in memory to access the data bus.
  - The PC is updated on the *Clock* pulse at the conclusion of execute microinstruction.
- Bring the *Read* line to 1 and the *Write* line to 0, so that the operand in memory being accessed by the PC is driven onto the data bus.
- Bring the *Load IR* line to 0 to prevent the opcode in the IR from being overwritten by the operand on the data bus at the *Clock* pulse that ends the execute state.
- Bring the *ACC to Data Bus* line to 0 to prevent the ACC contents from conflicting with the operand, being driven onto the data bus.
- Put the ALU in pass-through mode.
- Bring the *Load ACC* line to 1 to allow the *Clock* pulse that ends the execute state to load the output of the ALU into the ACC.

Confirm that the entries in Table 5-4 are consistent with these requirements.

**Table 5-4. ‘Load ACC with [operand]’ Microinstruction Definition Table.**

**Load ACC with [operand]: Opcode 0H**

<b>Output Bit</b>	<b>Control Line</b>	<b>Present State Bits</b>			
		00	01	02	03
1-0	Next State Bits	01	00		
2	Load IR	1	0		
3	Write	0	0		
4	Read	1	1		
5	ACC to Data Bus	0	0		
6	Load ACC	0	1		
7	Load MAR	0	0		
8	Use PC	1	1		
9	/~A_Only	X	0		
10	/~Invert	X	1		
11	Logic/~Arith	X	X		
12	X	X	X		
13	X	X	X		
14	X	X	X		
15	X	X	X		
Hex Equivalent		0115	0550		

Notice that the next-state bits associated with the execute microinstruction are 00B. This means that, at the conclusion of the execute microinstruction, the fetch microinstruction will be executed. This fetch microinstruction will fetch from memory the opcode of the next instruction to be executed.

### The ‘Add [operand] to ACC’ Instruction

The add instruction, ‘Add [operand] to ACC’ will appear in program memory as shown in Table 5-5.

Let’s choose 1H=0001B to be the opcode for ‘Add [operand] to ACC’. Using this opcode, ‘Add 5 to ACC’ is stored in memory as shown in Table 5-5. ‘Add [operand] to ACC’ requires using only two of the possible four machine macroinstructions; an execute microinstruction to execute the ‘Add [operand] to ACC’ opcode stored in the IR and a fetch microinstruction to fetch the opcode of the next sequential instruction stored in memory. Verify that the values of the PROM control lines needed to perform the Add operation, as specified in the [Instruction Execute State](#) section are consistent with those listed in Table 5-6. Notice that the next-state bits associated with the execute-state 1 microinstruction will cause the fetch microinstruction to be executed after the execute-state 1 microinstruction is completed.

**Table 5-5. Program Storage Scheme for Instruction ‘Add [operand] to ACC’.**

Address	Contents	Example
Q	Opcode for ‘Add to ACC’	1
Q+1	Operand	5

**Table 5-6. ‘Add [operand] to ACC’ Microinstruction Definition Table.**

**ADD [operand] to ACC: Opcode 1H**

Output Bit	Control Line	Present State Bits			
		00	01	02	03
1-0	Next State Bits	01	00		
2	Load IR	1	0		
3	Write	0	0		
4	Read	1	1		
5	ACC to Data Bus	0	0		
6	Load ACC	0	1		
7	Load MAR	0	0		
8	Use PC	1	1		
9	/~A_Only	X	1		
10	/~Invert	X	1		
11	Logic/~Arith	X	0		
12	X	X	X		
13	X	X	X		
14	X	X	X		
15	X	X	X		
	Hex Equivalent	0115	0750		

**The ‘Stop’ Instruction**

The ‘Stop’ instruction will appear in program memory as shown in Table 5-7. Choosing 2H=0010B to be the opcode for ‘Stop’, it will be stored in memory as shown in Table 5-7. ‘Stop’ will cause the microprocessor to cease executing new instructions. By creating a microinstruction whose next-state bits are the same as the present-state bits, as shown in Table 5-8, we force the controller into an infinite loop, making it repetitively execute the microinstruction associated with execute-state 1. Once our controller enters this loop, the only way we will have of regaining control of our processor is to use the *Reset* control line. The PROM output values for all bits other than the next state bits are shown as ‘0’s’ in Table 5-8. Many of these values can be specified as ‘X’s’. Can you determine which of these values may be defined as don’t cares? (There is more than one answer to this question.)

**Table 5-7. Program Storage Scheme for Instruction ‘Add [operand] to ACC’.**

Address	Contents	Example
Q	Opcode for ‘Stop’	3
Q+1	XXXXXXXXXXXXXX	X

**Table 5-8. Microinstruction Definition Table for the ‘Stop’ Instruction.****Stop: Opcode 2H**

		Present State Bits			
Output Bit	Control Line	00	01	02	03
1-0	Next State Bits	01	01		
2	Load IR	1	0		
3	Write	0	0		
4	Read	1	0		
5	ACC to Data Bus	0	0		
6	Load ACC	0	0		
7	Load MAR	0	0		
8	Use PC	1	0		
9	/~A_Only	X	0		
10	/~Invert	X	0		
11	Logic/~Arith	X	0		
12	X	X	X		
13	X	X	X		
14	X	X	X		
15	X	X	X		
Hex Equivalent		0115	0001		

**The ‘Add [3 operands] to ACC’ Instruction**

So far we have looked only at instructions that require two cycles to execute; one cycle to perform the execute microinstruction specified by the opcode fetched and one cycle to fetch from memory the opcode of the next instruction to be executed. Let’s look at a microinstruction that uses all four states. Consider

the instruction ‘Add [3 operands] to ACC’. This instruction is designed to add its three operands to a value already stored in the ACC. The instruction is stored in the program portion of the microprocessor’s memory as shown in [Table 5-9](#).

Let’s choose the opcode FH for this instruction. Using this opcode, ‘Add 5,2,3 to ACC’ is stored in memory as shown in [Table 5-9](#). Prove to yourself that the microinstruction definition table, Table 5-10 implements this instruction. Observe that constructing the

**Table 5-9. Storage Scheme for Instruction ‘Add [3 operands] to Acc.’**

Address	Contents	Example
Q	Opcode for ‘Add [3 operands] ACC’	F
Q+1	Operand 1	5
Q+2	Operand 2	2
Q+3	Operand 3	3

microinstruction definition table is straightforward; we start in state 0 with a fetch microinstruction and cycle through as many states/microinstructions as needed to perform the operations needed by our instruction—deciding which control lines to assert and how many cycles are needed is the only challenge.

**Table 5-10. ‘Add [3 operands] to ACC’ Microinstruction Definition Table.**

#### ADD [3 operands] to ACC: Opcode 1H

Output Bit	Control Line	Present State Bits			
		00	01	02	03
1-0	Next State Bits	01	10	11	00
2	Load IR	1	0	0	0
3	Write	0	0	0	0
4	Read	1	1	1	1
5	ACC to Data Bus	0	0	0	0
6	Load ACC	0	1	1	1
7	Load MAR	0	0	0	0
8	Use PC	1	1	1	1
9	/~A_Only	X	1	1	1
10	/~Invert	X	1	1	1
11	Logic/~Arith	X	0	0	0
12	X	X	X	X	X
13	X	X	X	X	X
14	X	X	X	X	X
15	X	X	X	X	X
Hex Equivalent		0115	0752	0753	0750

#### Revisiting Time Skewing

Before completing the design of our controller, let’s digress for a moment to understand how time skewing, caused by the presence of the synchronizing register, affects the operation of our controller. We predicted in the [‘Synchronizing a Mealy Machine’ section of Appendix B](#) that because of the presence of the synchronizing register in the controller, the opcode fetched by the fetch microinstruction (state 0) would

not be available to the Mealy machine until state 1, the clock cycle after the fetch is executed. We also predicted that this would mean that the last operation of each instruction would be the fetch microinstruction and that this microinstruction would fetch the opcode of the next instruction to be executed. To illustrate that these predictions have come to fruition, consider the timing diagram Figure 5-11, which corresponds to execute of the following program,

(Unspecified Instruction)  
Load ACC with 3  
Add 1,3,5 to ACC.

The timing diagram of Figure 5-11 clearly shows that the first microinstruction executed for each opcode is that associated with execute-state 1 and that the last microinstruction executed is the fetch microinstruction.

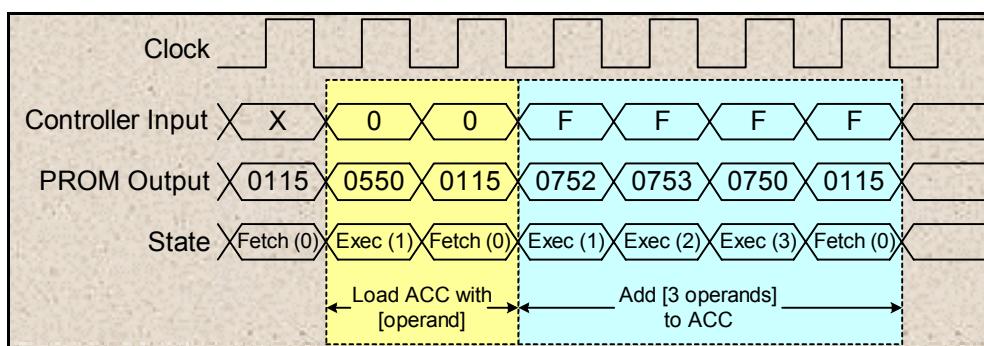


Figure 5-11. Timing diagram for execution of ‘Load ACC’ then ‘Add [3 operands] to ACC.’

### Defining the Contents of our PROM

The last step before we can build our PROM is to define its contents for all possible inputs. To do this we concatenate the microinstruction definition tables for each instruction we have defined ('Load ACC with [operand]', 'Add [operand] to ACC', and 'Stop'), starting with opcode 0, and replacing all 'don't cares' and undefined quantities with 0, as shown in Table 5-11. (This table does not include the 'Add [3 operands] to ACC' instruction.) The 16-bit hex numbers in the bottom row of this table are the control signal values that will be stored in our PROM.

### Task 5-4: Programming the Limited-Instruction-Set Instruction Decoder PROM

The first task in constructing the controller is to program the instruction decoder, or PROM. Program the instruction decoder with the limited instruction set we have defined as shown in Table 5-11: 'Load ACC', 'Add to ACC', and 'Stop'. For help in creating and programming a PROM with this instruction set, read [Appendix D: Creating a PROM Device in LogicWorks™ 4 for Windows](#)<sup>6</sup>.

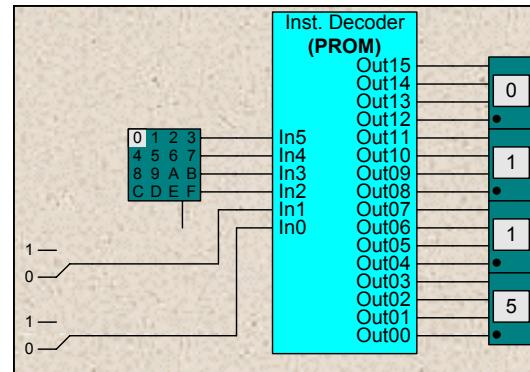
<sup>6</sup> If you are using a Macintosh®, you can get the access to the equivalent tutorial via the web site: <http://www.eas.asu.edu/~cse120/>.

**Table 5-11. Input-Output Data for Limited Instruction Set PROM.**

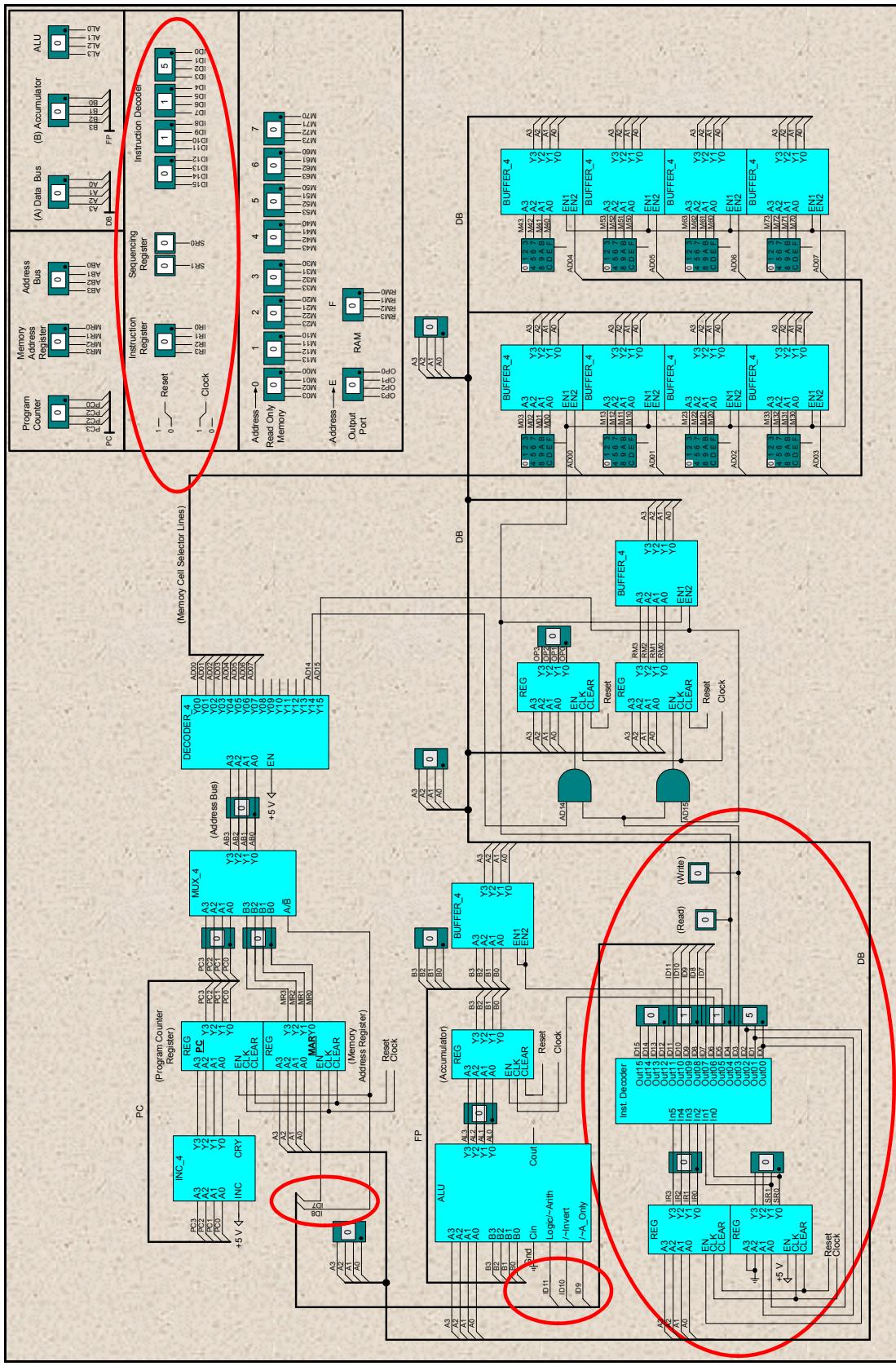
Instructions		Load ACC				Add to ACC				Stop			
Opcode		0				1				2			
Description	Pin Number	00	01	02	03	00	01	02	03	00	01	02	03
Next State Bits	1-0	01	00	00	00	01	00	00	00	01	01	00	00
Load IR	2	1	0	0	0	1	0	0	0	1	0	0	0
Write	3	0	0	0	0	0	0	0	0	0	0	0	0
Read	4	1	1	0	0	1	1	0	0	1	0	0	0
ACC to Data Bus	5	0	0	0	0	0	0	0	0	0	0	0	0
Load ACC	6	0	1	0	0	0	1	0	0	0	0	0	0
Load MAR	7	0	0	0	0	0	0	0	0	0	0	0	0
Use PC	8	1	1	0	0	1	1	0	0	1	0	0	0
/~A_Only	9	X	0	0	0	X	1	0	0	X	0	0	0
/~Invert	10	X	1	0	0	X	1	0	0	X	0	0	0
Logic/~Arith	11	X	X	0	0	X	0	0	0	X	0	0	0
X	12	X	X	0	0	X	X	0	0	X	X	0	0
X	13	X	X	0	0	X	X	0	0	X	X	0	0
X	14	X	X	0	0	X	X	0	0	X	X	0	0
X	15	X	X	0	0	X	X	0	0	X	X	0	0
Hex Equivalent		0115	0550	0000	0000	0115	0750	0000	0000	0115	0001	0000	0000

**Task 5-5: Testing the Instruction Decoder Circuit**

After you build your instruction decoder test it using the circuit shown in Figure 5-12. Verify as many stored entries as you feel necessary. Do not record detailed results in your lab notebook; however you will want to record which input/output combinations you checked in the event a problem develops with some aspect of the PROM later; this will save you the time of having to recheck those input/output pairs you have already validated. If any of your tests are not successful, check the data you have entered in your PROM and reprogram it if necessary.

**Figure 5-12.** Test circuit for instruction decoder.**Task 5-6: Build and Test the Controller**

Build and test the controller of Figure 5-10 using the instruction decoder you constructed in Task 5-4. Label all signals as shown in Figure 5-10. Record in your notebook the test conditions that result in successful results. If any of your tests are not successful, debug your circuit using the skills you've acquired throughout these laboratory exercises. (Debugging may include checking additional PROM entries.)



**Figure 5-13.** The complete microprocessor circuit.

## Task 5-7: Build the Complete Microprocessor Circuit

A term's worth of work is about to pay off! Insert the controller circuit into your brainless microprocessor using the schematic of Figure 5-13. Make sure all of the controller's lines are labeled as shown in Figure 5-13. Connect the controller's outputs to all of the lines that you have been controlling with switches (except *Reset* and *Clock*) and delete these switches from the display/control panel. Add elements to the control panel as shown.

## Task 5-8: Write and Execute a Simple Program for Your Microprocessor

Now that you have your microprocessor built, you may be very excited to try it out! As a test of your microprocessor, suppose we try the following program:

<u>Instruction</u>	<u>Comment</u>
Load ACC with 3	;Put the number 3 into the Accumulator
Add 5 to ACC	;Add 5 to the value stored in the Accumulator
Stop	;Stop executing new instructions

In memory, this program would look like this:

<u>Address</u>	<u>Contents</u>	<u>Comment</u>
0	0	;The ‘Load ACC’ Opcode
1	3	;The number ‘3’ to be loaded into the Accumulator
2	1	;The ‘Add to ACC’ Opcode
3	5	;The number 5 to be added to the Accumulator
4	2	;The ‘Stop’ Opcode

This program will add the numbers 3 and 5 and store the result in the accumulator. Enter the program in your ROM starting at address 0, and then reset your microprocessor using the *Reset* line.

### Checking and Debugging Your Program

Step through your program one step at a time by toggling the switch that drives the *Clock* signal line, and make sure every step of every instruction works correctly.

**LogicWorks™ Note:** When you step through a program by toggling the microprocessor *Clock*, it is important that you not toggle the *Clock* too fast. You need to give LogicWorks™ enough time to complete the simulation of all signal changes (after each clocking event) before you toggle the *Clock* again.

Record the operation of each step, i.e. record the values of the control lines, and record whether data is being moved properly according to those control line settings. Do you get ‘8’ stored in the accumulator when your program is done executing? If not, you have some debugging to do. Do not be too disappointed

if your microprocessor does not work the first time. It may take a little debugging to find wires that did not connect, and other minor problems. Problems at this stage are most likely to be caused by:

- faulty interconnections of major components (sometimes producing data conflicts),
- incorrect PROM programming, or
- incorrect control of the *Clock* or *Reset* signals.

One good approach to debugging your microprocessor is to check that the program you executed in Task 5-8 is correctly performing each of the sequence of microinstructions as listed in Table 5-12. This table shows that upon resetting, your controller accessed location 00H of your instruction decoder, which contained the fetch microinstruction<sup>7</sup>. The outputs of your decoder caused the first opcode of your program, 0H, to be made available at the input of the IR. The first clocking event caused the fetch microinstruction to load the first opcode of your program, 0H, into your IR. With this opcode loaded in the IR and the state register loaded with 01B, your controller began executing the execute microinstruction associated with the first instruction in your program, ‘Load ACC with 3’. The second clocking event loaded ‘3’ into the ACC and your controller began executing the fetch microinstruction associated with the ‘Load ACC’ instruction. Etc. If you can isolate any problem you have to the execution of one microinstruction, you are well on your way to solving the problem.

Table 5-12. Sequence of Microinstructions Executed for Program of Task 5-8.

		1	2	3	4	5	6	7	Time
Clocking Events		↓	↓	↓	↓	↓	↓	↓	→
Clock Cycle	Reset	1	2	3	4	5	6	Etc.	
Instruction	-----	Load ACC	Load ACC	Add to ACC	Add to ACC	Stop	Stop	Stop	
PROM Addr.	00H	01H	00H	11H	10H	21H	21H	21H	
Microinstr.	Fetch	Execute 1	Fetch	Execute 1	Fetch	Execute 1	Execute 1	Execute 1	
State	0	1	0	1	0	1	1	1	

If you do have problems, they are unlikely to be due to faulty major components because of the careful testing you have performed with every component you built; however, this occasionally happens, so don’t rule out the possibility. If you have problems, carefully track them down and try to isolate them to a particular microinstruction, component and/or interconnection. If any of the major components appear to be responding incorrectly, observe their input and output values and compare them with the test results you have recorded in this or earlier labs. Extract any malfunctioning components and test them outside of the circuit to eliminate the effects of your circuit acting on the component. You may begin, at this stage, to appreciate why we have put such emphasis on testing each of the circuits you built and recording the

<sup>7</sup> Using Mealy machine terminology we would express this as follows: upon resetting your microprocessor you entered state 0 with an input of 0H, (supplied by the IR).

results of those tests. If after repeated testing you encounter problems that befuddle you, see a laboratory assistant.

### Double Execution of the ‘Load ACC’ Fetch Microinstruction & Nonexecution of the ‘Stop’ Microinstruction

Notice in Table 5-12 that the fetch microinstruction associated with the ‘Load ACC’ instruction (i.e., PROM address 00H) is executed twice, yet it has only been included once in our program. Why does this occur? Well, when you reset your controller’s Instruction and State registers, you force the controller to access the microinstruction stored in your PROM at location 00H without regard to the first instruction contained in your program<sup>8</sup> – which coincidentally happens to be a ‘Load ACC’ instruction. The fetch microinstruction that your controller executes upon being reset, loads the IR with the ‘Load ACC’ instruction opcode, namely 0H, from program memory. Since this opcode is, coincidentally, the same value that the IR contained after the reset operation, the fetch operation that culminates the ‘Load ACC’ instruction uses the same fetch microinstruction as the reset operation triggered.

Finally, observe that the fetch instruction associated with the ‘Stop’ instruction is never executed. Because of this, you may be tempted to replace the PROM outputs for the ‘Stop’ fetch state with ‘don’t cares.’ You can do this provided you never reassign the ‘Stop’ instruction to have an opcode of 0H. If you reassign ‘Stop’ to have a 0H opcode, the first microinstruction must be an honest fetch instruction since, upon resetting the microprocessor, we need the controller to execute the fetch microinstruction first.

### Task 5-9: Add the ‘XOR’, ‘Zero’, ‘Subtract’, and ‘Store ACC’ Instructions

In this task you get to demonstrate your understanding of your microprocessor by adding instructions to your instruction set. Add the ‘XOR’, ‘Zero’, ‘Subtract’, and ‘Store ACC’ instructions to your microprocessor circuit.

- The ‘XOR’ instruction should perform a bit-wise XOR operation of the value in the ACC with an operand in memory and store the result in the ACC.
- The ‘Zero’ instruction should store the hex digit 0H in the accumulator by subtracting the accumulator from itself and storing the result back in the accumulator.
- The ‘Subtract’ instruction gets the next value in memory and subtracts it from the accumulator by forming its two’s complement, adding the result to the accumulator, then storing this result back into the accumulator.

---

<sup>8</sup> We were careful to design our PROM so that this location contained a fetch microinstruction so that our controller, upon being reset, would begin by fetching the opcode of the first program instruction you stored in external memory ROM location 0H.

- The ‘Store ACC’ instruction should write the contents of the ACC into a location in memory specified by the instruction’s operand.

Write and execute a few programs to test each of your instructions. When writing your program, ‘Store’ only to locations EH or FH, the output port and simulated RAM, respectively. (These are the only location that can store data under program control.) Include the programs and their outputs in your report along with the microinstruction definition tables and/or state transition diagram for each instruction.

### **Task 5-10: Invent Your Own Instruction**

Invent a new instruction and add it to your instruction set. Write and execute a program that tests your instruction. Include the program(s) and their outputs in your report along with the microinstruction definition table and/or state transition diagram for your instruction.

### **Task 5-11: Backup Your Files**

Take a moment and backup you libraries and circuit files on a new floppy disk and mark it ‘SimLab#5’. Do this using a different floppy disk after every lab and you just may save yourself much unnecessary work.

## **Language Hierarchies in Computer Programming**

The instruction set you created looks nothing like the high level language you are probably accustomed to using. For example, in your instruction set, the instruction ‘Load ACC with 3’ was entered into memory as 1H and 3H (if you used the same opcode we chose). Why are these instruction formats so different from what you are used to? It’s because the computer language(s) you are most likely familiar with, C, C++, Fortran, Basic, etc., are high-level languages. An expression ‘Load ACC with 3’ is a lower level language known as assembly language and ‘13H’ is known as machine language. One high-level language instruction such as A=5+3, is the equivalent of many assembly-language instructions. Each assembly-language instruction in turn is equivalent to one machine language instruction, which in turn is composed of several microinstructions.

To translate, from a high level language (the type of languages we like to use), to machine language (which the computer can use) we need a compiler. To translate from assembly language to machine language we need an assembler. We could develop an assembler that would translate a limited set of instructions from a assembly language, into machine language, but this would be quite a bit of work and we’d need to learn much more than we have time for here. Further, we will have no need for an assembler for our microprocessor because the programs we might develop are small and simple enough that machine language will suffice.

## Task 5-12: Finding Errors on Your CD Cover

When the CD cover was created, the artist was given license to sacrifice accuracy for the sake of aesthetics. There are errors of omission and commission on the cover. Which ones can you find?

### Congratulations!

When you started these laboratory exercises, it is likely that you knew nothing or little about digital systems. Over the time span of one course, you have created an ALU, built a microprocessor controller, defined your own instruction set, then entered a program in ROM and simulated its execution. You have gained a fundamental understanding of the major elements that any microprocessor must have and how each of these components are designed and controlled. Perhaps most important (and most impressive!), you now understand how a controller coordinates the activity of each component to perform arithmetic and logical operations and how a controller performs data transfers between the memory and the CPU. Now is the time to congratulate yourself; you have come a long way and learned a great deal. Also, let me congratulate you. Congratulations!

# SIMULATION LAB 5: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Top-Down Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Operation of the Microprocessor		
Operation of the Controller		
Operation of the Address Generation Circuit		
The Instruction Set		
How a Program is Stored in Memory		
Testing and Validation of the Instruction Set		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
	Points	
<b>Lab Score</b>	Late Lab	
	Lab Score	

## Report Writing Reminders:

*Caveat emptor:* Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.

- Title Page: Include **Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted** in all reports.
- Include **Truth Tables or Function Tables** as required to explain how circuits work and as proof of circuit tests.
- **Label** all figures or circuits. (E.g. Figure 1. Schematic of . . . , Figure 2. . . , etc.)
- Refer in the text of your report to all circuits or figures that you include in the body of your report.
- Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.

# SIMULATION LAB 5: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Task Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 5-1: Add More Memory to Your Microprocessor		
Task 5-2: Build, Test and Debug the Memory-Address-Generation Circuit		
Task 5-3: Add the Memory-Address-Generation Circuit to the Processor		
Task 5-4: Programming the Limited-Instruction-Set Instruction Decoder PROM		
Task 5-5: Testing the Instruction Decoder Circuit		
Task 5-6: Build and Test the Controller		
Task 5-7: Build the Complete Microprocessor Circuit		
Task 5-8: Write and Execute a Simple Program for Your Microprocessor		
Task 5-9: Add the ‘XOR’, ‘Zero’, ‘Subtract’, and ‘Store ACC’ Instructions		
Task 5-10: Invent Your Own Instruction		
Task 5-12: Finding Errors on Your CD Cover		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>		
Points		
Late Lab		
Lab Score		

## Report Writing Reminders:

**Caveat emptor:** Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.

- Title Page: Include **Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted** in all reports.
- Include **Truth Tables or Function Tables** as required to explain how circuits work and as proof of circuit tests.
- **Label** all figures or circuits. (E.g. Figure 1. Schematic of . . . , Figure 2. . . , etc.)
- Refer in the text of your report to all circuits or figures that you include in the body of your report.
- Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

Put an ‘X’s’ in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use ‘5’ to indicate that you ‘strongly agree’, ‘3’ to indicate that you are ‘neutral’, and ‘1’ to indicate that you ‘strongly disagree’. Use ‘NA’, Not Applicable, when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your *responses will not be graded*. They are for your instructor’s information only.

Table \_\_: Self-Assessment of Outcomes for Simulation Lab 5: The Complete Microprocessor.

After completing the assigned tasks and report, I am able to:	5	4	3	2	1	NA
Design a PROM-based controller for an elementary microprocessor.						
Create an instruction set for an elementary microprocessor.						
Use the language of your instruction set to create a program and enter it into memory.						
Execute a program on your simulated microprocessor.						

Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

# SIMULATION LAB 6: THE JUMP INSTRUCTION

---

**Prerequisites:** Before beginning this laboratory experiment you must have completed:

- [Simulation Lab 1: Half Adder, Increment & Two's Complement Circuit.](#)
- [Simulation Lab 2: 4-Bit Full Adder, Multiplexer & Decoder.](#)
- [Simulation Lab 3: Arithmetic and Logic Unit.](#)
- [Simulation Lab 4: The Brainless Microprocessor.](#)
- [Simulation Lab 5: The Complete Microprocessor.](#)

**Equipment:** Personal computer and LogicWorks™

**Objective:** In this laboratory experiment you will modify the design of a microprocessor so that you can implement a jump instruction. You will recreate your instruction set to be compatible with the new architecture and you will demonstrate your success by entering a program, which includes a jump instruction, into memory and showing that the program executes correctly.

**Outcomes:** When you have completed these laboratory experiments you will be able to:

- Modify an existing microprocessor architecture so that it has the capability of executing a jump instruction.
- Describe the control scheme needed with your architecture to implement a jump instruction.
- Describe the flow of information needed with your architecture to implement a jump instruction.
- Modify the existing instruction set to be compatible with the modified microprocessor architecture.
- Create a jump instruction for the modified microprocessor architecture.
- Demonstrate that the jump instruction functions correctly with the modified architecture.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab, it is recommended that you use a top-down format when organizing your lab report.

(If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Top-Down Report Writing Guidelines](#).)

## Prologue

The work you completed in the first five simulation laboratory assignments resulted in a simulation of a microprocessor. For most tasks, the circuits you simulated were given to you in the form of a schematic diagram. In the remaining laboratory exercises, you will be asked to rely more on your own understanding of digital design. In other words, you will do more of the design work yourself. For every design, you will be given at least a design specification and, perhaps, some schematic suggestions. Good luck!

## Introduction

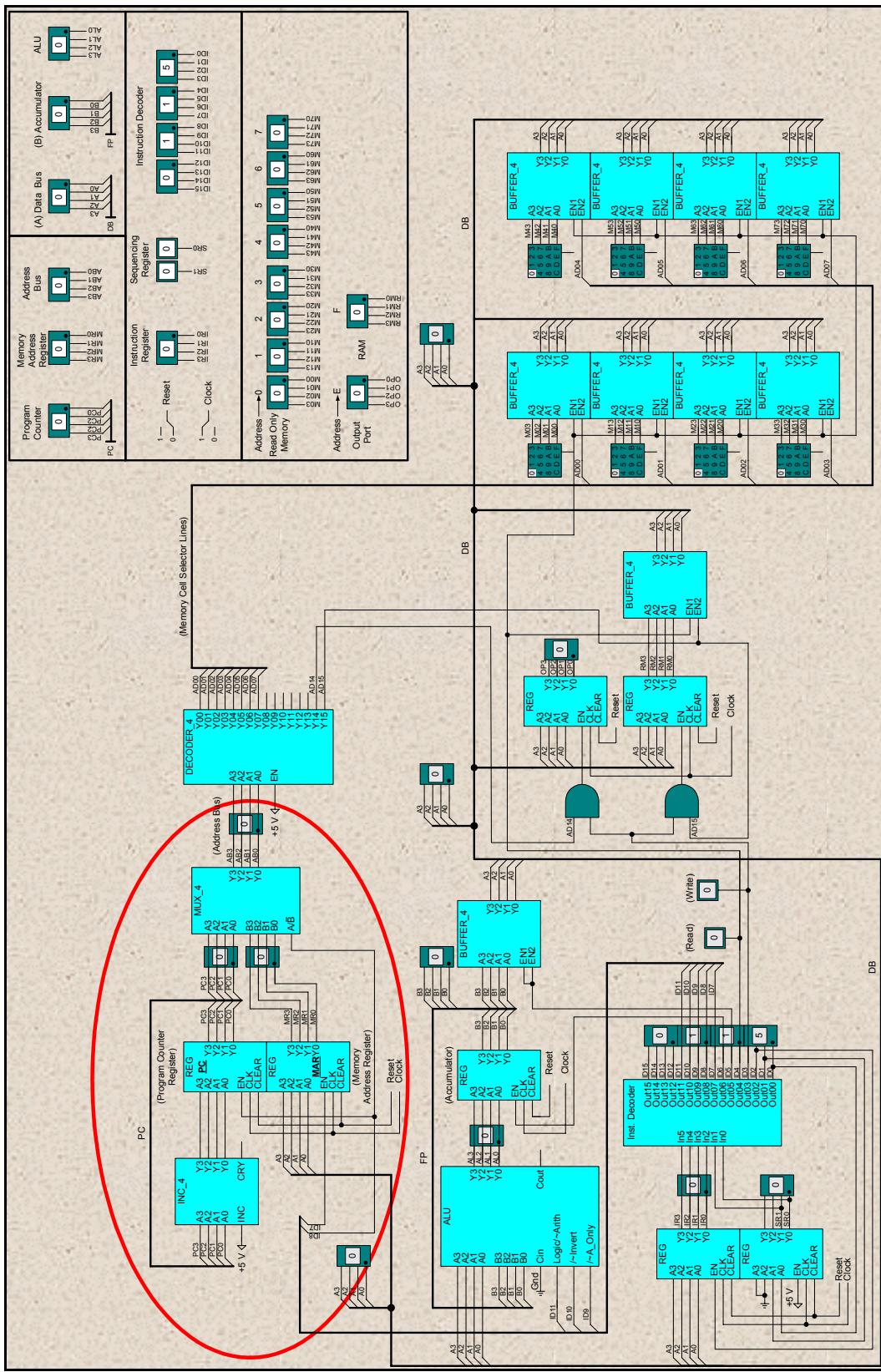
In the last simulation lab assignment you completed your microprocessor design. Congratulations! Before we celebrate too much, or file a patent disclosure, observe that our microprocessor has no way of implementing a jump instruction. From your experience with programming, you know that a powerful programming technique is to break a program into a set of subroutines, and, at the completion of each subroutine, jump back to the location in the main program from where the subroutine was called. In this laboratory exercise, you will modify the existing microprocessor architecture so that it is capable of executing a jump instruction. You will then modify your instruction set to be compatible with the new architecture and add a jump instruction to your instruction set. Your last tasks will be to test your new instruction set, including your jump instruction, to show that it works correctly.

## Modifying the Existing Architecture

To implement a jump instruction, it is necessary to be able to load the PC with an address directly from memory through the data bus. For example, assuming that the opcode for a jump instruction is ‘7’, an instruction that jumps to memory location 9, shown in Table 6-1, must cause ‘9’ to be loaded into the PC. The address-generation circuit of the microprocessor shown in Figure 6-1, can load the MAR through the data bus; however, there is no way of loading the PC register from the data bus.

**Table 6-1. Program Storage Scheme for Instruction ‘ADD [operand] to ACC’.**

Address	Contents	Example
Q	Opcode for ‘Jump’	7
Q+1	Address Operand	9



**Figure 6-1:** The complete microprocessor circuit.

## Task 6-1. Modify Existing Microprocessor Architecture and Instruction Set for Jump Capability

Modify the existing microprocessor architecture so that it has the capability of performing a ‘Jump’ instruction. The ‘Jump’ instruction should be able to jump to a user specified memory location and continue execution from there. (If you are having difficulty, consider the hint below.) Give the ‘Jump’ instruction an opcode and include it in your instruction set. Also update your instruction decoder so that each of the instructions you created previously are compatible with the new architecture.

**Hint:** One way to implement the jump instruction is to modify the architecture so that there are two ways to load the PC register: from the data bus (with the address operand when executing a jump) and from the incrementer (with the next sequential value of the PC when executing instructions sequentially). This requires using another control line, from the instruction decoder, to select whether the incrementer or the data bus is used to load the PC register. The circuit shown in Figure 6-2 may help you.

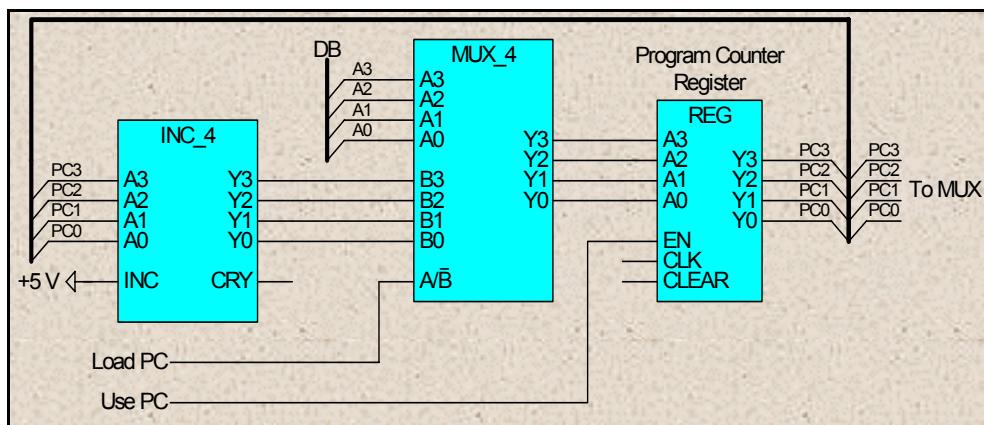


Figure 6-2. Architectural hint for the ‘Jump’ instruction.

## Task 6-2. Testing Your Architecture and Instruction Set

Write a program using your new instruction set and show that your instruction set works with your architecture to implement the ‘Jump’ instruction. Add more memory to your microprocessor if you need to. In your report be sure to describe the control scheme needed with your architecture to implement the jump instruction. Also describe the flow of information needed with your architecture to implement a jump instruction.

## Task 6-3: Backup Your Files

Take a moment and backup your libraries and circuit files on a new floppy disk and mark it ‘SimLab#6’. Do this using a different floppy disk after every lab and you just may save yourself much unnecessary work.

# SIMULATION LAB 6: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Top-Down Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Operation of the Address-Generation Circuit		
The Modified Instruction Set		
Testing and Validation of the Architecture and Instruction Set		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

<b>Caveat emptor:</b> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.
• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructor's Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.
• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.
• <b>Label</b> all figures or circuits. (E.g. Figure 1. Schematic of . . ., Figure 2. . . , etc.)
• Refer to all circuits or figures that you include in the text of your report.
• <b>Tip:</b> Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.

# SIMULATION LAB 6: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Task Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 6-1. Modify Existing Microprocessor Architecture and Instruction Set for Jump Capability		
Task 6-2. Testing Your Architecture and Instruction Set		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

<b>Caveat emptor:</b> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.
• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructor's Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.
• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.
• <b>Label</b> all figures or circuits. (E.g. Figure 1. Schematic of . . ., Figure 2. . . , etc.)
• Refer to all circuits or figures that you include in the text of your report.
• <b>Tip:</b> Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

Put X's in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use '5' to indicate that you 'strongly agree', '3' to indicate that you are 'neutral', and '1' to indicate that you 'strongly disagree'. Use 'NA', 'Not Applicable', when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your responses will not be graded. They are for your instructor's information only.

Table \_\_: Self-Assessment of Outcomes for Simulation Lab 6: The Jump Instruction.

After completing the assigned tasks and report, I am able to:	5	4	3	2	1	NA
Modify an existing microprocessor architecture so that it has the capability of executing a jump instruction.						
Describe the control scheme needed with your architecture to implement a jump instruction.						
Describe the flow of information needed with your architecture to implement a jump instruction.						
Modify the existing instruction set to be compatible with the modified microprocessor architecture.						
Create a jump instruction for the modified microprocessor architecture.						
Demonstrate that the jump instruction functions correctly with the modified architecture.						

# SIMULATION LAB 7: THE STATUS REGISTER AND THE CONDITIONAL JUMP

---

**Prerequisites:** Before beginning this laboratory experiment you must have completed:

- [Simulation Lab 1: Half Adder, Increment & Two's Complement Circuit.](#)
- [Simulation Lab 2: 4-Bit Full Adder, Multiplexer & Decoder.](#)
- [Simulation Lab 3: Arithmetic and Logic Unit.](#)
- [Simulation Lab 4: The Brainless Microprocessor.](#)
- [Simulation Lab 5: The Complete Microprocessor.](#)
- [Simulation Lab 6: The Jump Instruction.](#)
- The study of the two's complement number system.

**Equipment:** Personal computer and LogicWorks™

**Objective:** In this simulation experiment you will modify the design of your microprocessor so that you can implement a conditional-jump instruction. You will then recreate your instruction set to be compatible with the new architecture and you will demonstrate your success by writing and executing programs that test each of your conditional jump instructions.

**Outcomes:** When you have completed these laboratory experiments you will be able to:

- Modify an existing microprocessor architecture so that it has the capability of executing a conditional-jump instruction.
- Describe the control scheme needed with your architecture to implement a conditional-jump instruction.
- Describe the flow of information needed with your architecture to implement a conditional-jump instruction.
- Modify the existing instruction set to be compatible with the modified microprocessor architecture.
- Create seven conditional-jump instructions for the modified microprocessor architecture.
- Demonstrate that each of the conditional-jump instructions created, functions correctly.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab, it is recommended that you use a top-down format when organizing your lab report. (If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Top-Down Report Writing Guidelines](#).)

---

## Introduction

In the previous lab assignment you modified the existing microprocessor architecture so that you could implement a jump instruction and you modified your existing instruction set so that it was compatible with the new architecture.

In this laboratory exercise we will be adding seven new instructions to our instruction set. These instructions will be conditional-jump instructions; this means that whether the jump is made is determined by (conditioned upon) the results of a previous instruction. For example, consider the loop structure used when creating a computer program using a high-level language; the computer code within the loop is repeatedly executed using different input data until an exit condition is satisfied. Typical conditions that cause the loop to be exited are an index reaching a limit, or a specific value being encountered. When the exit condition is satisfied, the loop is exited, that is, the next instruction executed is not within the range of the loop and (chances are) the opcode of this next instruction is located somewhere in memory distant from the memory containing the loop instructions. When the higher-level language comprising this loop is processed by a compiler, the resulting equivalent assembly-language instruction that performs this jump is a conditional-jump instruction.

As a practical example of a conditional jump instruction, consider the case where we have a section of a routine that performs two's complement addition. After the two's complement addition we may wish to test whether an overflow has occurred; if an overflow has occurred, we will want our program to jump to a location in memory containing instructions that signal to the user that the result of the addition is invalid; otherwise we may wish to execute the next sequential instruction. Assuming our instruction set has an instruction ‘Jump on Overflow’, a program to appropriately handle the overflow condition may look something like that shown in Table 7-1.

The control signals, which determine whether the conditional jump of Table 7-1 is made, include information about the arithmetic result from the processor’s ALU. This information, along with information about the particular instruction being executed (e.g., ‘Jump on Overflow’), can then be used to control the address-generation circuitry so that a jump is either allowed to occur or prevented from occurring.

In this laboratory exercise, you will modify your existing computer architecture so that information from the ALU can be used to implement conditional-jump instructions. You will add a set of conditional-jump instructions to your instruction set and modify your existing instruction set so that it is compatible with the new architecture you create.

**Table 7-1. Example Use of a Conditional-Jump Instruction.**

Memory Address	Instruction	
00	Load ACC	← Beginning of program to perform two's complement addition.
•	•	
•	•	
•	•	
04	Jump on Overflow	← Jump to location 09H if overflow occurs
05	09	
06	Load ACC	← Continue performing program instructions if no overflow occurs
•	•	
•	•	
•	•	
09	Store	← Store overflow detect code
0A	F	← (taken as F in this example),
0B	To location 0EH	← to our output port, location 14D.
0C	Stop	
•	•	
•	•	

### Adding Status Capability to your ALU

The condition that determines whether or not a conditional-jump instruction causes a jump to occur is based on the result of a logical or arithmetic operation involving the ALU. The ALU that we have designed currently does not produce all of the information (available in a typical microprocessor) that characterizes the results produced by the ALU, information that programmers want and need to know. Currently, our ALU produces only one of the status signals we'll need, the carry-out signal, as shown in Figure 7-1. There are four other status values we'd like to have the capability of checking:

- whether the parity is even or odd, (We'll define parity shortly.)
- whether the ALU operation has resulted in an overflow or not,
- whether the result is zero or not zero,
- and whether the result of an operation is positive or negative.

The output bits that we'll need are shown schematically in [Figure 7-2](#) and defined in Table 7-2.

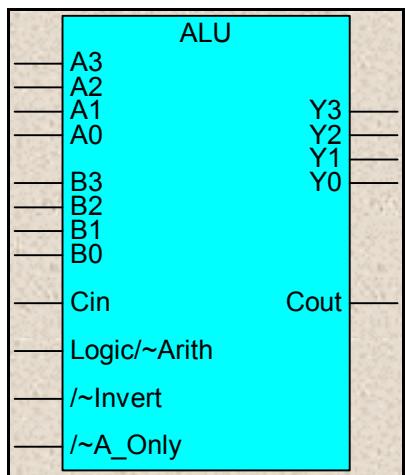


Figure 7-1. Existing ALU subcircuit symbol.

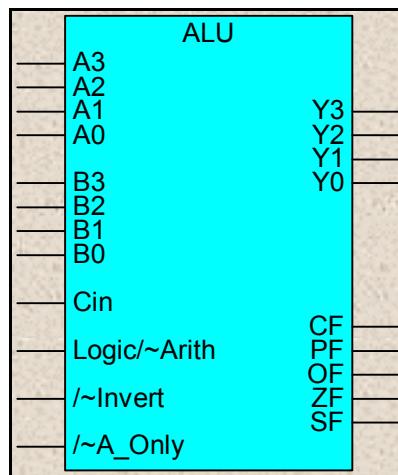


Figure 7-2. Modified ALU subcircuit symbol.

Table 7-2. ALU Status Output Definition.

ALU Status	Value	Interpretation: ALU Operation resulted in a(n):
CF <sup>1</sup>	1	Carry
	0	No Carry
PF	1	Parity Even
	0	Parity Odd
OF	1	Overflow
	0	No Overflow
ZF	1	Zero Value
	0	Non-zero Value
SF	1	Negative Value
	0	Positive Value

### Status Flags

The CF, PF, OF, ZF and SF outputs are known as status bits, or more commonly, status flags. These five status flags will be present in a typical microprocessor, along with a few others. The five status flags that we are including in our design are arguably the most used and will give you a good feeling about the types of jump instructions that can be found in most microprocessors. Also, since we have limited ourselves to adding seven conditional-jump instructions to our instruction set, (you'll see why shortly), five is about as many status bits as we can reasonable use.

<sup>1</sup> This is the same as  $C_{out}$  in Figure 7-1. We will be using CF to make the carry-flag label consistent with the other flags.

Your job, in the first task, is to modify your ALU so that it produces the PF, OF, ZF and SF status flags; it already produces the CF status flag, which was label  $C_{out}$  in the earlier lab exercises. The following explanations of the functions of the new status flags will be helpful when you begin to redesign your ALU.

**Parity Flag (PF):** The parity flag is high when the number of 1's, occurring in the ALU result, is an even number, and 0 when the number of 1's occurring in the ALU result is an odd number. For example, for our 4-bit ALU, the parity flag is high if the ALU result is 0110, and low if the result available to the accumulator is 1110.

**Overflow Flag (OF):** We want OF to be high when an addition or subtraction results in an overflow. An overflow occurs when two negative numbers are added to yield a positive number or when two positive numbers are added and yield a negative number.

There are many ways to express the OF flag function. Let's look at the most traditional way, with the help of an example. Consider the addition of two positive two's-complement 4-bit numbers using a 4-bit ALU as shown in Figure 7-3. (In this figure  $C_0$ ,  $C_1$ ,  $C_2$ , and CF are the carry bits produced from the addition in the LSB through MSB positions, respectively. Verify for yourself that, for the example of Figure 7-3,  $[CF, C_2, C_1, C_0] = [0,1,1,1]$ .)

CF=0	$C_2=1$	$C_1=1$	$C_0=1$
	0	1	1
+	0	0	0
	0	1	0

Figure 7-3. Addition of two, two's complement numbers showing an overflow.

The two's complement operands in Figure 7-3 are both positive numbers since the most significant bits of each are zero, but the result is negative since its most significant bit is a  $1^2$ ; hence an overflow has occurred. We will define our OF status flag to be high when an overflow occurs and low when no overflow occurs<sup>3</sup>. It can be shown that an overflow will occur when  $C_2$ , and CF differ. One realization of the overflow flag is based on the following equation<sup>4</sup>:

$$OF = \overline{CF} \bullet C_2 + CF \bullet \overline{C_2} = CF \oplus C_2$$

Note that the OF function is meaningful only when the ALU is manipulating two's complement numbers.

**Zero Flag (ZF):** The zero flag is to be high when all bits of the result produced by the ALU, excluding the carry flag, are zeros. Otherwise the zero flag is to be low.

**Sign Flag (SF):** The sign flag is identical to the most significant bit of the result supplied by the ALU. This flag typically is used when the ALU is manipulating two's complement numbers.

<sup>2</sup> Remember that in two's complement arithmetic, the carry-out, CF, is not part of the representation of the result.

<sup>3</sup> This definition is completely arbitrary. We could just as easily define the status flag to have logic values opposite to those we have chosen.

<sup>4</sup> You may find it curious that CF plays a role in defining the overflow condition even though (as you will find in your course text) it plays no part in the representation of the result. Indeed, the OF function can be constructed without using the CF bit; however, by using the CF bit, we can express the OF function with fewer terms.

## Task 7-1: Modify the ALU to Include Status Flags

Modify your ALU to produce the status flag outputs defined in Table 7-2 and embed the modified ALU in a subcircuit using the symbol shown in Figure 7-2. (You may want to re-label your existing ALU, along with the subcircuits used within your existing ALU, with a name such as ‘ALU\_Lab3’, to avoid overwriting your existing circuits with your new design.) Test your status-capable ALU with sufficient validation tests so that you can convince the reader of your report that your status-capable ALU is working properly. Remember to record the results of your test in your laboratory notebook. Also remember to save the subcircuit in your library.

Upon testing your ALU design you may notice that the CF, and OF flags (which we have defined only for arithmetic operations) provide logic values even if the ALU is controlled to perform a logical operation. This may be puzzling at first. After all, isn’t the OF flag meaningless when performing a logical operation? Indeed, this is one of many examples where the information conveyed by a status flag is meaningless. The meaning the status flags contain is a function of the instruction being executed; consequently, when using the status flags, it is the responsibility of the programmer to know which status flags are meaningful for which instruction; therefore, in your report, you must provide information (perhaps in a table) that allows the reader to determine, for each of your non-jump instructions, whether or not the interpretation of the status flags (specified in Table 7-2) is valid. Also, when you perform your validation tests, you only need to show that the status-flag results are correct for the instructions for which the status flags are meaningful.

### Status Register

Just as it is necessary to store the result of the ALU operation in the ACC for use by subsequent instructions, it is necessary to store the status flags associated with the ALU operations in a status register for later use.

## Task 7-2: Add a Status Register

Add a status register to store the status flags from the ALU as shown in Figure 7-4.

Connect the ‘EN’ input of the Status Register with the same signal that enables the ACC, as shown in Figure 7-4. This common enable signal allows the status flags to be always reflective of the ACC contents.

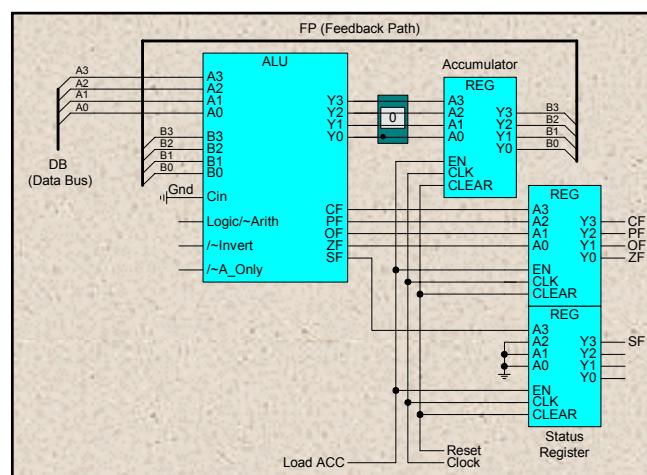


Figure 7-4. Schematic of ALU and status register.

## Conditional-Jump Instructions

Typical microprocessors have many instructions. Because our instruction register can store only 4 bits, it is only capable of storing 16 different opcodes; hence we are limited to 16 different instructions. Reserving 8 opcodes, 0000B through 0111B for the arithmetic and logical instructions you developed in Simulation Lab 5, we are left with 8 opcodes for implementing jump instructions. Reserving one of these opcodes for the unconditional-jump instruction you implemented in Simulation Lab 6, we have seven opcodes left for conditional-jump instructions.

Conditional-jump instructions are often found in pairs in typical instruction sets. For example, if the instruction JC (standing for ‘Jump on Carry’) is in the instruction set, then JNC (‘Jump on Not Carry’) will also be in the instruction set. Because we are limiting ourselves to eight jump instructions, our instruction set, shown in Table 7-3 along with abbreviations and opcodes<sup>5</sup>, does not include the complement of every conditional-jump instruction. To include the complement of every jump instruction we’d need to add JPO (‘Jump on Parity Odd’), JNS (‘Jump on Not Sign’), JNO (‘Jump on No Overflow’), JNZ (‘Jump on Not Zero’) and JLE (‘Jump on Less Than or Equal to’).

Table 7-3. Jump Instruction Set.

Opcode (Binary)	Abbr.	Instruction	Condition
1000	JMP	Unconditional Jump	Jump
1001	JC	Jump on Carry	Jump when CF=1
1010	JNC	Jump on Not Carry	Jump when CF=0
1011	JPE	Jump on Parity Even	Jump when PF=1
1100	JS	Jump on Sign	Jump when SF=1
1101	JO	Jump on Overflow	Jump when OF=1
1110	JZ	Jump on Zero	Jump when ZF=1
1111	JG	Jump on Greater	Jump when result of subtraction shows OP1 > OP2.

The first jump instruction in Table 7-3<sup>6</sup>, JMP, is the unconditional-jump you implemented in Simulation Lab 6. The remainder of the jump instructions in this table are all conditional-jump instructions. The ‘Condition’ column of Table 7-3 shows the conditions that cause a jump to be performed when each instruction is executed. All of these jumps are conditioned upon one status flag, except for the JG instruction.

<sup>5</sup> The opcodes assigned to instructions in Table 7-3 are only suggestions. You are free to assign any instruction to any opcode.

<sup>6</sup> The opcode shown in this table may be different from the one you used in your earlier implementation. When you implement JMP in this laboratory exercise you may wish to reassign it the opcode shown in Table 7-3, although that is not necessary.

### 'Jump on Greater' Instruction

The JG instruction is defined to cause a jump when the result of a subtraction shows that the first operand is strictly greater than the second operand. To use this instruction properly in a program, a subtraction must be executed first. For example:

Load ACC with [OP1]	;Move operand 1 into the ACC
Subtract [OP2] from ACC	;Subtract operand 2 from the ACC, $[OP1]-[OP2] \rightarrow ACC$
JG	;Jump if $[OP1] > [OP2]$ , i.e., $[OP1]-[OP2] > 0$

You can make using the JG instruction simpler if you create a new instruction for your instruction set, call it the ‘compare’ instruction, that performs the first two instructions above. In that case, the JG<sup>7</sup> instruction is used as follows:

Compare [OP1], [OP2]	;Compare operand 1 and operand 2, $[OP1]-[OP2] \rightarrow ACC$
JG	;Jump if $[OP1] > [OP2]$ , i.e., $[OP1]-[OP2] > 0$

Our task is to determine the values of the ALU status flags that indicate that operand 1 is strictly greater than operand 2, when  $[OP1]-[OP2]$  is evaluated. There are three flags we’ll need to use: SF, OF, and ZF.

To understand how the status flags are affected by different combinations of operands, let’s look at the range of results that can be obtained, using the binary circle of Figure 7-5 as an aid. The 4-bit binary circuit shown in Figure 7-5 shows all of the possible bit patterns and their decimal equivalents if these bit patterns are interpreted as two’s complement numbers. Recall that in the two’s complement number system, negative base-ten-number representations have a most significant bit of 1; positive numbers are those whose most significant bit is zero.

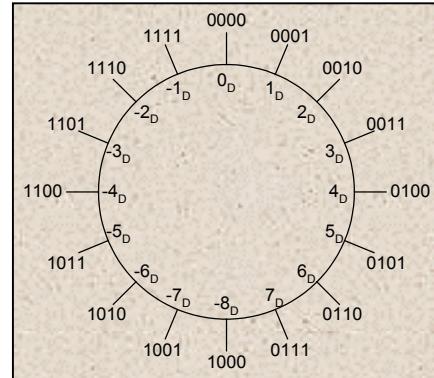


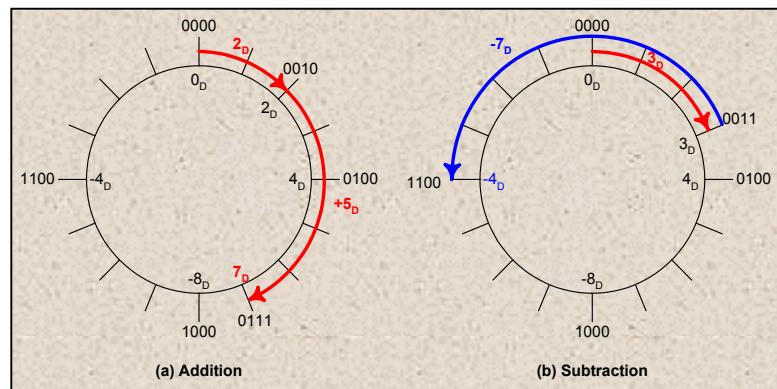
Figure 7-5. Four-bit binary circle.

An arithmetic operation performed by our 4-bit ALU will always produce a value that is located somewhere on the 4-bit binary circle; some of the results may be valid, others may not. The addition of a positive number is graphically equivalent to moving around the binary circle in a clockwise direction; the addition of a negative number (or subtraction) is graphically equivalent to moving in a counter-clockwise direction. For example, the graphical addition of 2D and 5D and the graphical subtraction, 3D – 7D, is shown in Figure 7-6 (a) and (b), respectively. Both of these two’s-complement arithmetic operations yield results whose two’s-complement representations are equivalent to the correct decimal result.

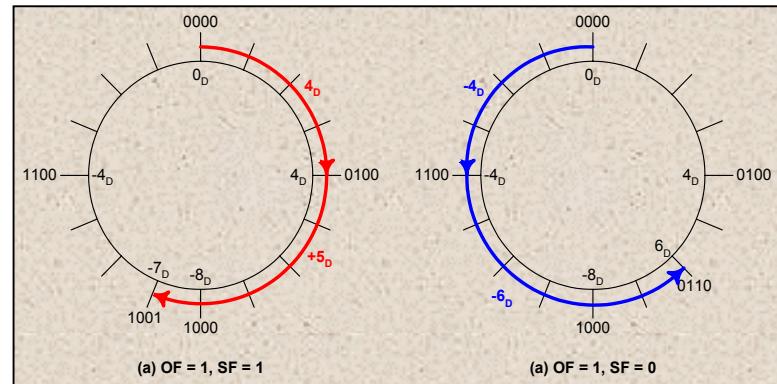
<sup>7</sup> Note that you could also abbreviate this instruction JLE, to represent ‘Jump on Less Than or Equal to’, and the instruction would cause a jump when operand 2 was strictly less than operand 1. Clearly, when you are designing your instruction set, you have flexibility when labeling each instruction and the responsibility to tell the users how to correctly use each instruction.

Not all arithmetic operations will yield valid results. The representation of a result will be invalid whenever an overflow occurs. Overflows occur when combining two positive or two negative numbers causes the result to cross the +/- boundary. Figure 7-7(a) shows that the two's complement addition of 4D and 5D using our 4-bit ALU erroneously yields -7D, with an overflow flag of OF=1, and a sign flag of SF=1. Figure 7-7(b) shows that the two's complement addition of -4D and -6D using our 4-bit ALU erroneously yields 6D, with flags OF=1 and SF=0.

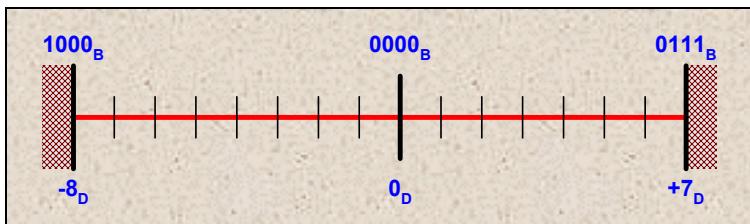
The 4-bit binary circle is useful for determining when an overflow occurs and for determining the 4-bit value that will result from a two's complement operation. For the purpose of determining the values of the status flags that will occur as the result of an arithmetic operation, we can use a simpler diagram, known as the 4-bit binary-range line shown in Figure 7-8. Whenever the right-most (+ to -) boundary is traversed by an arithmetic operation, a negative value and an overflow results, with OF=1, and SF=1. Whenever the left-most (- to +) boundary is traversed by an arithmetic operation, a positive value and an overflow results, with OF=1, and SF=0. If the arithmetic operation results in neither boundary being violated, then OF=0 and the value of SF is determined by the sign of the valid two's complement result.



**Figure 7-6.** Graphical interpretation of two's complement addition and subtraction.



**Figure 7-7.** Two's complement addition and subtraction examples in which overflows occur.



**Figure 7-8.** Four-bit binary-range line.

We want to determine the values of the status flags that the JG instruction should recognize as indicative of the case  $OP1 > OP2$ , after the subtraction  $OP1 - OP2$  has occurred. To aid in this discussion, let's assign  $OP2$  to be the  $A$  variable, and  $OP1$  to the  $B$  variable. We chose this assignment of variables  $A$  and  $B$  because of the ALU symbol shown in Figure 7-4. Recall that to form the operation  $OP1 - OP2$ , we must first load  $OP1$  into the ACC.  $OP1$  is then made available to the  $B$  input port of the ALU. Next,  $OP2$  is made available to the  $A$  input port of the ALU from memory and the subtraction is performed.

When we perform the arithmetic operation  $B - A$ , i.e.,  $[OP1] - [OP2]$ , there are three different scenarios that will cause different status flag patterns:  $B = A$ ,  $B < A$ ,  $B > A$ .

### Case 1: $B = A$ ( $OF = 0$ , $SF = 0$ , $ZF = 1$ )

If  $B = A$ , then the result of the subtraction operation is:  $B - A = 0D$ . No overflow will occur; hence  $OF = 0$ . The sign bit of the result,  $0D = 0H$ , is  $0B$ ; hence  $SF = 0$ . And the zero flag is set to 1 ( $ZF = 1$ ) indicating a zero result.

### Case 2: $B < A$ ( $OF \oplus SF = 1$ , $ZF = 0$ )

If  $B < A$ , then  $B$  must lie to the left of  $A$  on the 4-bit binary-range line as shown in Figure 7-9. The status flags that result from the operation  $B - A$  depend on the magnitude of the result:

Case 2a: If the value  $B - A$  is between  $-1D$  and  $-8D$  then no overflow occurs and the result is negative. ( $OF = 0$ ,  $SF = 1$ ).

Case 2b: If the value  $B - A$  is more negative than  $-8D$  (which might occur if  $B$  is negative and  $A$  is positive), then an overflow occurs and the sign of the result is positive. ( $OF = 1$ ,  $SF = 0$ ).

Comparing these two subcases, we observe that regardless of the magnitude of  $B - A$ ,  $OF$  always takes on the opposite value of  $SF$  and  $OF \oplus SF = 1$  for the case where  $B < A$ . Also since  $B \neq A$ ,  $B - A \neq 0$  and the zero flag must always be zero ( $ZF = 0$ ).

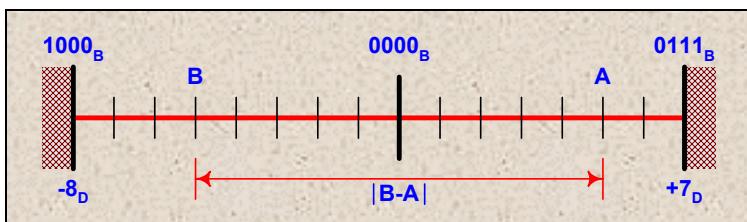


Figure 7-9. Four-bit binary-range line showing the case  $B < A$ .

### Case 3: $B > A$ ( $OF \oplus SF = 0$ , $ZF = 0$ ) (JG Jump Condition)

If  $B > A$ , then  $B$  must lie to the right of  $A$  on the 4-bit binary-range line as shown in Figure 7-10. The status flags that result from the operation  $B - A$  depend on the magnitude of the result:

Case 3a: If the value B-A is between +1D and +7D then no overflow occurs and the result is positive. (OF=0, SF =0).

Case 3b: If the value B-A is more positive than +7D (which might occur if B is positive and A is negative), then an overflow occurs and the sign of the result is negative. (OF=1, SF=1).

Comparing these two subcases, we observe that regardless of the magnitude of B-A, OF always takes on the same value as SF and hence OF  $\oplus$  SF=0 for the case where B>A. Also since B  $\neq$  A, B-A  $\neq$  0 and the zero flag must always be zero (ZF=0).

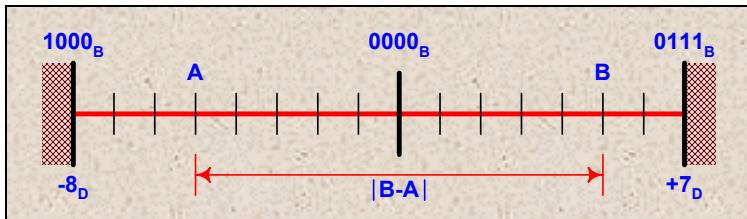


Figure 7-10. Four-Bit binary-range line showing the case B>A.

### The JG Condition

Observing the status flags of the above cases we see that we want the JG instruction to cause a jump (i.e., the JG function from our branch logic to be 1) when SF  $\oplus$  ZF=0 and when ZF =0. Using Boolean algebra, this condition becomes:

$$JG = \overline{OF \oplus SF} \bullet \overline{ZF} \quad (1)$$

Double complementing (1) and using DeMorgan's law we can rewrite JG in a form that will require fewer logic gates.

$$\begin{aligned} JG &= \overline{\overline{OF} \oplus \overline{SF}} \bullet \overline{\overline{ZF}} \\ &= \overline{OF} \oplus \overline{SF} + \overline{ZF} \\ &= OF \oplus SF + ZF \end{aligned} \quad (2)$$

We will use the JG function of (2) to control the active-high jump-control input signal to the address-generation circuit when the JG instruction is being executed.

### Branch Logic

Using the definition of the JG function in (2) and a understanding of how each instruction is to operate, we can construct the branch logic of Figure 7-11. This branch logic has one jump-enabling function for each conditional and unconditional instruction to be included in our instruction set. Each function is high when the results contained in the status register are such that the respective jump instruction should cause a jump

to be executed. For example, for the unconditional jump, we want the jump-enabling output function, (the JMP function), of the branch logic to be perpetually high, as it is in Figure 7-11. For the ‘Jump on Carry’ instruction, we want the jump-enabling output

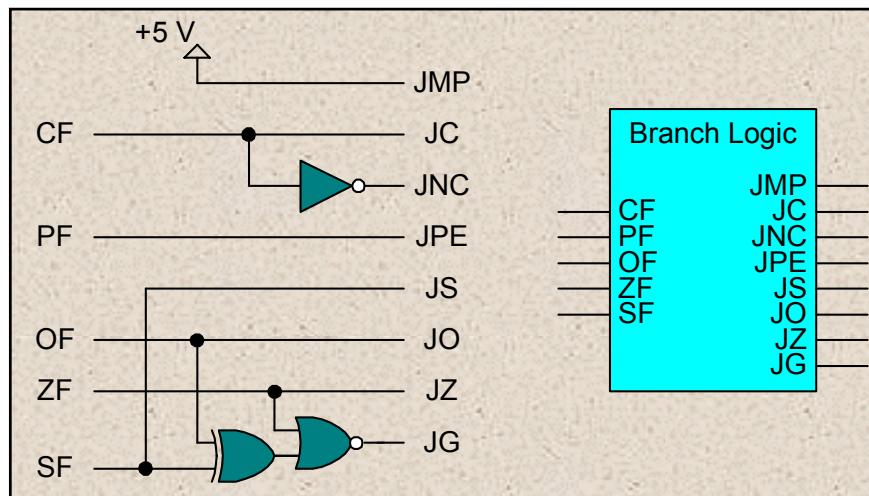


Figure 7-11. Branch control logic schematic.

function (the JC function) to be high whenever CF is high. For the ‘Jump on Not Carry’ instruction, we want the corresponding jump-enabling output function (the JNC function), to be high whenever CF is low. Justify to yourself that these conditional-jump functions, as well as the conditional-jump functions JPE, JS, JO, JZ and JG, whose schematics are shown in Figure 7-11, are consistent with the jump-condition definitions contained in Table 7-2, and equation (2).

### Task 7-3: Build the Branch Logic

Construct the branch logic and imbed it in a subcircuit as shown in Figure 7-11. Perform sufficient tests on your branch logic subcircuit so that you have confidence that it is working correctly. Remember to save the subcircuit in your library.

### Jump Control Circuitry

In Simulation Lab 6, you modified your address-generation circuit to implement an unconditional-jump instruction. One schematic for the address-generation circuit that you may have used is shown in Figure 7-12. Regardless of the design you created in Simulation Lab 6, your address-generation circuit design will have one input, let’s call it the jump-control input (*Load PC* in Figure 7-12), which controls whether or not a jump takes place. To implement conditional-jump instructions, we need to combine information about the result of the ALU operation, (provided by the branch logic), with information about which instruction is being executed (provided by the controller PROM) to control of the *Load PC* line (or jump-control input) of the address-generation circuit. We combine these pieces of information using the circuit shown in Figure 7-13. This circuit consists of an 8-to-1 mux whose data inputs are connected to the output of the branch logic, and whose address inputs are controlled by the instruction-decoder outputs (PROM) from the controller.

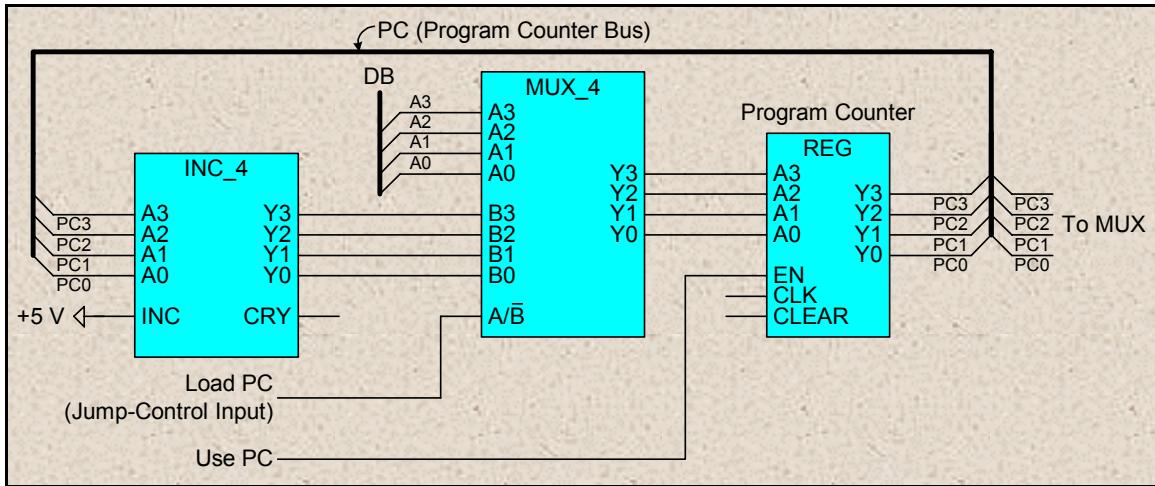


Figure 7-12. Schematic of address-generation circuit.

To implement eight jump instructions (seven conditional and one unconditional), we use four controller output lines. Three of the controller lines, J2, J1, and J0, control the 8-to-1 mux of Figure 7-13 to select the branch-logic function corresponding to the instruction being executed. One of the controller output lines, we'll call the *Jump Enable*, needs to be high for all jump instructions (during the appropriate microinstruction),

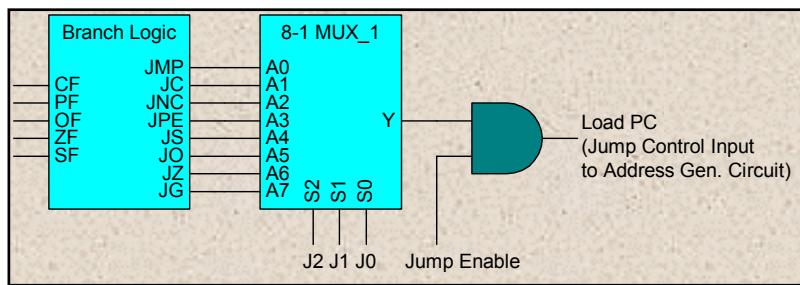


Figure 7-13. Jump-control circuit schematic.

and low for all other instructions. The *Jump Enable* signal, when high, will enable the AND gate of Figure 7-13 to allow the selected branch-logic function to control the address-generation-circuit jump-control input. When low, the *Jump Enable* control line from the controller, causes the address-generation circuit jump-control input to remain low. This prevents the jump from taking place.

### Task 7-4: Complete the Conditional-Jump Control Circuit

Complete the jump control logic of Figure 7-13. Validate your circuit's performance. Integrate your circuit into the microprocessor as shown Figure 7-14. Connect the AND gate output of the jump-control circuit to the jump-control input of the address-generation circuitry. Route the *Jump Enable*, J2, J1, and J0 controller output lines, shown in Figure 7-15, to the 8-1 mux shown in Figure 7-14.

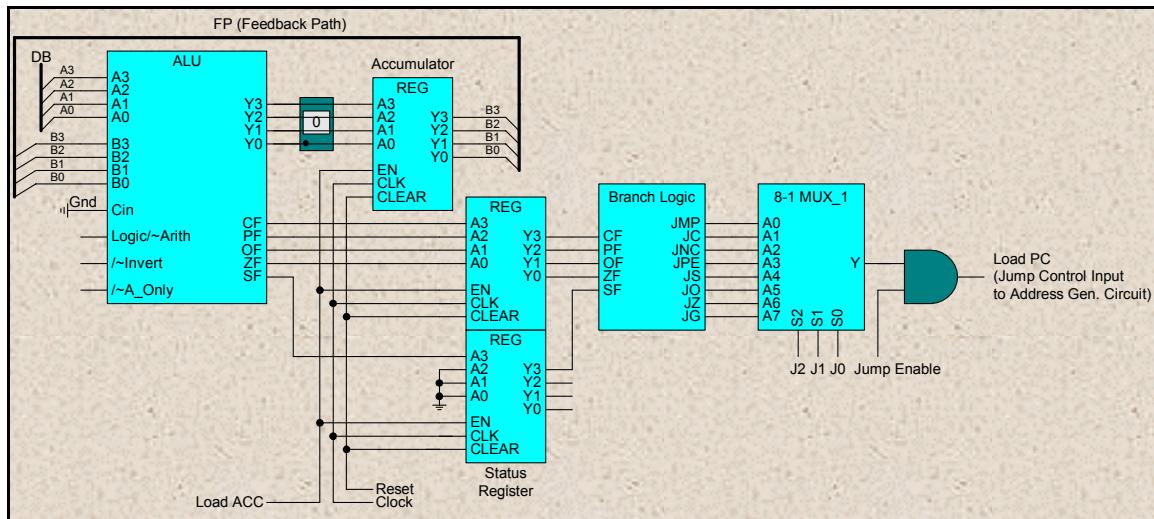


Figure 7-14. Schematic showing integration of jump-control circuitry.

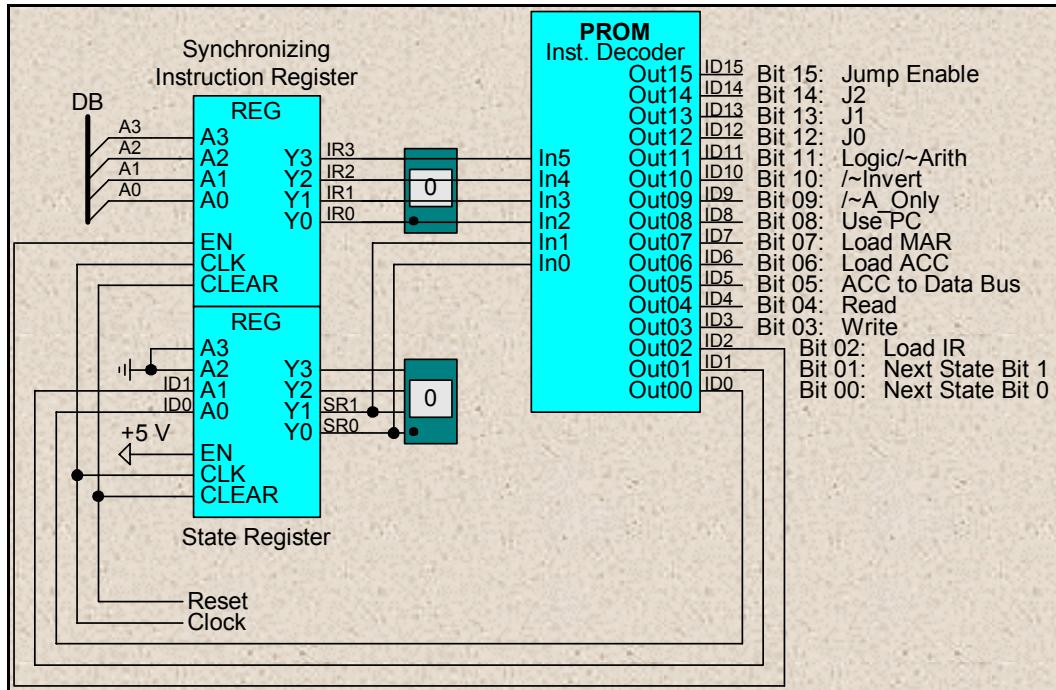


Figure 7-15. Assignment of instruction decoder output lines.

## Modifying Your Instruction Set

Since we have modified the architecture of your system it will be necessary to modify the contents of the instruction decoder. Each instruction in your instruction set will have to include appropriate values for instruction-decoder output bits 12 through 15. You will also need to add the jump instructions of Table 7-3 to your instruction set.

### **Task 7-5: Update Your Instruction Set**

Update your instruction set to account for your new architecture and include the jump instructions of Table 7-3 in your instruction set. (Remember: if you want to keep your original instruction decoder (PROM), you will need to save your existing version with a new name, such as ‘PROM1\_Lab5,’ or save your new PROM using a new name.)

### **Task 7-6: Validate Your Architecture and Instruction Set**

Write and execute programs that thoroughly test your instruction set and its interaction with your architecture. Include the programs and their outputs in your report along with the microinstruction definition tables for each of your instructions. Convince the reader of your report that the programs you’ve created thoroughly test your instruction set and architecture. In your report be sure to describe the control scheme needed with your architecture to implement the conditional-jump instruction. Also describe the flow of information needed with your architecture to implement a conditional-jump instruction.

### **Task 7-7: Backup Your Files**

Take a moment and backup your libraries and circuit files on a new floppy disk and mark it ‘SimLab#7’. Do this using a different floppy disk after every lab and you may just save yourself much unnecessary work.

### **Congratulations!**

Congratulations, you have completed the design and simulation of a fully functional elementary microprocessor! At this point you may have many ideas about how you might change your architecture or your instruction set to provide it with more functionality. Any changes you, or we, might make at this point would not require the introduction of new concepts; you have been introduced to all of the fundamental concepts used in the design of microprocessor architectures by completing these exercises.

Indeed, there are many things we can do to improve our microprocessor. We could enhance our instruction set, or make improvements in our architecture, or we could simply increase the functionality of the components within our existing architecture. Before we launch into the never-ending process of improving our design, or before we confess that there are other things we would rather do, let’s realize that we need to stop sometime. Let it be now.

# SIMULATION LAB 7: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Top-Down Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Modified CPU Architecture		
Overview of the Conditional Jump Operation		
Modified ALU and Status Register		
Jump Control Circuitry		
Branch Logic		
Capabilities of Modified Instruction Set		
Testing and Validation of Architecture and Instruction Set		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	<b>Points Lost</b>	
	<b>Late Lab</b>	
	<b>Lab Score</b>	

## Report Writing Reminders:

*Caveat emptor:* Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.

- Title Page: Include Your Name, ID, Class Time, Course Number, Your Instructor's Name, Laboratory Experiment Number and Title, and Date Submitted in all reports.
- Include Truth Tables or Function Tables as required to explain how circuits work and as proof of circuit tests.
- Label all figures or circuits. (e.g. Figure 1. Schematic of . . ., Figure 2. . . , etc.)
- Refer to all circuits or figures that you include in the text of your report.
- Tip: Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.

# SIMULATION LAB 7: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Task Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 7-1: Modify the ALU to Include Status Flags		
Task 7-2: Add a Status Register		
Task 7-3: Build the Branch Logic		
Task 7-4: Complete the Conditional-Jump Control Circuit		
Task 7-5: Update Your Instruction Set		
Task 7-6: Validate Your Architecture and Instruction Set		
What I Learned		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

<i>Caveat emptor:</i> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.
• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructor's Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.
• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.
• <b>Label</b> all figures or circuits. (e.g. Figure 1. Schematic of . . ., Figure 2. . . , etc.)
• Refer to all circuits or figures that you include in the text of your report.
• <b>Tip:</b> Cut and paste your digital logic schematics from LogicWorks™ into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

Put an “X’s” in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use ‘5’ to indicate that you ‘strongly agree’, ‘3’ to indicate that you are ‘neutral’, and ‘1’ to indicate that you ‘strongly disagree’. Use ‘NA’, ‘Not Applicable’, when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your responses will not be graded. They are for your instructor’s information only.

Table \_\_ : Self-Assessment of Outcomes for Simulation Lab 7: The Status Register and the Conditional Jump.

After completing the assigned tasks and report, I am able to:	5	4	3	2	1	NA
Modify an existing microprocessor architecture so that it has the capability of executing a conditional-jump instruction.						
Describe the control scheme needed with your architecture to implement a conditional-jump instruction.						
Describe the flow of information needed with your architecture to implement a conditional-jump instruction.						
Modify the existing instruction set to be compatible with the modified microprocessor architecture.						
Create seven conditional-jump instructions for the modified microprocessor architecture.						
Demonstrate that each of the conditional-jump instructions created, functions correctly.						

# HARDWARE LAB 0: USING A PROTOTYPE BOARD, LOGIC PROBE & VOLTMETER

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Describe the following concepts: voltage, current, ground, open circuit, and closed circuit.
- Understand simple Boolean Algebraic expressions.
- Recognize the schematic symbol for an AND gate.

**Equipment:** Digital Trainer Board, Logic Probe, Voltmeter

**Integrated Circuits:** You will need the following IC to complete this lab:

- (1) 7408 (Quad 2-input AND gate)

**Objective:** The objective of this laboratory exercise is to familiarize you with the operation of a typical prototyping board, digital logic probe and digital voltmeter (DVM).

**Outcomes:** When you have completed these laboratory experiments you will be able to:

- Insert an IC in the proper location in a breadboard.
  - Wire the IC to ground and the +5 Volt (+5V.) power supply.
  - Connect pins of an IC to realize a combinational logic circuit.
  - Use a logic probe to determine a signal's logic levels.
  - Use a voltmeter to determine a signal's voltage levels.
- 

## Introduction

The objective of this laboratory exercise is to help you familiarize yourself with the operation of a typical digital logic prototyping environment. You will need this familiarity in order to realize (i.e., design, build and debug) the digital-logic circuits that you learn about in class and that are assigned in this manual. To facilitate building these circuits we will be using measuring instruments as well as a connector board that will allow us to make rapid connections between circuits. We will refer to this board as a breadboard. Some breadboards, like the one we'll describe here, are integrated into a hardware development environment that includes voltage sources and measuring instruments. We'll refer to these environments as trainer boards, or a prototyping boards. In addition to the measuring devices on the trainer board (i.e.,

light-emitting diodes [LED's]), you will be using two instruments to measure the voltage values of the inputs and outputs of the circuits you build: a logic probe, and a voltmeter.

In this laboratory exercise, we will describe how to use the Elenco Electronics Inc. digital trainer board, a digital logic probe, and the Fluke 8840A digital multimeter. The equipment available to you in the lab may differ from that described here; however, because the operation of all trainer boards and voltage measuring instruments is similar, the instructions included with this exercise will help you understand how to operate the equipment you will find on your lab bench. Throughout your lab experience you will have questions about aspects of the lab equipment and experiments. The laboratory teaching assistant's (TA's) are your friends. Ask them for help.

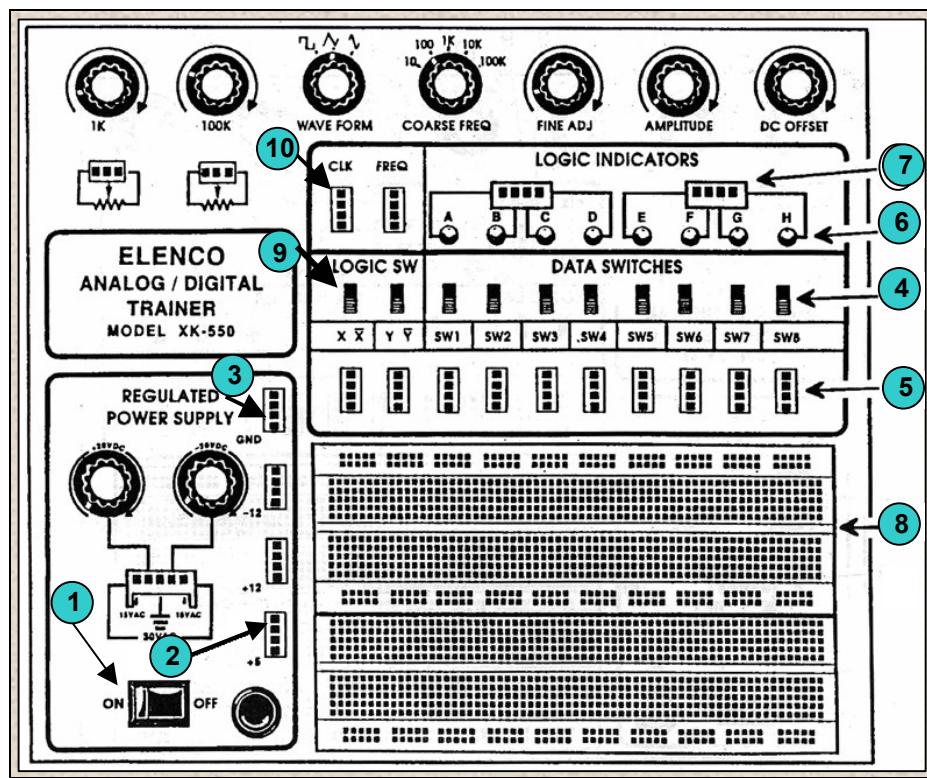


Figure 0-1. ELENCO Analog/digital trainer board. (Used with permission of Elenco Electronics Inc.)

## Breadboard Basics

A schematic of the Elenco digital trainer board is shown in [Figure 0-1](#). The trainer board has an ON/OFF switch at the lower left corner ①. It is wise practice to keep this switch turned off until you have finished wiring or re-wiring your circuit. You will be using the +5 V., ②, and the ground (GND), ③, connections to supply power to your digital devices. The square black boxes pointed to by ② and ③ in Figure 0-1 represent spring-loaded openings in the face of the trainer board into which you may insert the ends of

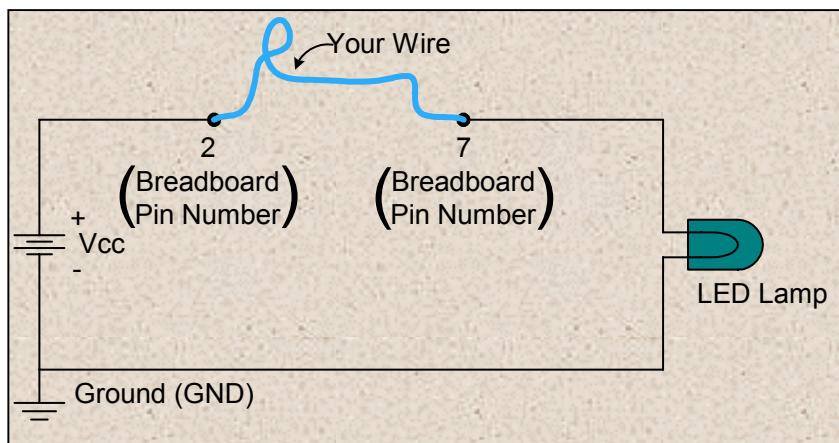
wires (stripped bare) to make connections to +5V. and ground, respectively. (You may ignore the +12 V. and -12 V. terminals, as you will not use them. There are other connections and knobs that you will also not be using. It is wise not to experiment with these until you are instructed in their proper use.)

The trainer board also contains 8 data switches (④ in [Figure 0-1](#)) for controlling inputs to your circuits. You get access to the signals these switches control by inserting wires into the holes (marked as ⑤ in [Figure 0-1](#)) located directly below each switch. When a switch, ④, is in the bottom (down) position the output from its corresponding 4-pin terminal, ⑤, is a signal whose voltage is 0 V.; this is also known as a logical false or digital 0. When a switch is in the top (up) position the output is +5 V.; this is also known as a logical true or digital 1.

Two additional logic switches labeled Logic SW (⑨ in Figure 0-1) are located to the left of the data switches. These are de-bounced switches. To learn more about de-bounced switches, see [Hardware Lab 3](#).

Also provided in the kit are 8 light-emitting diodes (LED's), ⑥. You will be wiring the output of your circuits to these LED via the holes, ⑦. This wiring strategy will allow the LED's to glow when the output of your circuit is +5 V., or remain dark when the output of your circuit is 0 V. Note that lines on the trainer board help you identify which hole gives you access to which LED.

You can test the LED by connecting a wire from the +5 V. supply (② in Figure 0-1) to the LED Lamp connection point (⑦ in Figure 0-1). When you make this connection properly, current will flow through the LED and it will glow. For the LED to light up, the LED must be part



**Figure 0-2.** Circuit schematic for LED lamp circuit.

of a closed circuit. How is it that one wire can form a closed circuit and allow current to flow through the lamp? The wire that you add supplies only the circuit completing leg of the closed circuit shown in [Figure 0-2](#); the other components that complete the circuit, shown in Figure 0-2, are hidden under the breadboard façade<sup>1</sup>.

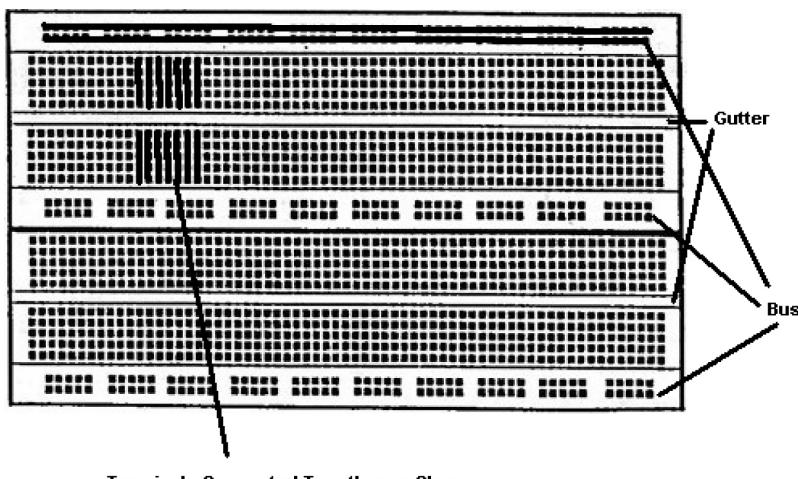
<sup>1</sup> Figure 0-2 is a simplified representation of the components hidden beneath the bread boards façade. For our needs, this simplified representation is sufficient.

The circuit connections hidden beneath the trainer board's façade are useful because they will allow you to form closed circuits with a minimal amount of wiring. The breadboard portion of the trainer board (⑧) in [Figure 0-1](#)) likewise has hidden connections as shown in Figure 0-3.

The holes that allow access to the six buses (see Figure 0-3) are connected horizontally beneath the breadboard's façade. When using the breadboard you will find it useful to connect at least one of the bus strips to +5 V. and at least one to ground (GND). There are many inputs to the circuits you will build that you will want to +5 V. and GND; the many access points available with the bus strips will allow you to make these connections easily.

The holes that allow access to the remainder of the board are connected vertically (see Figure 0-3). Each group of five vertically connected access holes is insulated from all other holes. All holes in the face of the breadboard (and trainer board) give you access to spring loaded terminals that allow you to make secure connections.

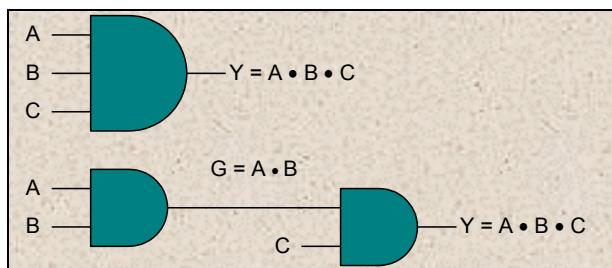
The last item noted in Figure 0-3 is the gutter. The gutter is the depression in the surface of the breadboard that acts to electrically insulate groups of holes from each other.



**Figure 0-3.** Breadboard connections. (Used with permission from Elenco Electronics Inc)

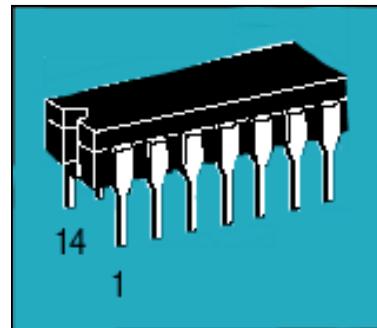
## Simple Logic Circuit

Let's use the trainer board to build a simple logic circuit. Consider the simple logic function:  $Y = A \bullet B \bullet C$ . Two ways of realizing this function are to use either one, 3-input AND gate or two, 2-input AND gates as shown in [Figure 0-4](#).



**Figure 0-4.** Simple logic circuit.

To gain experience in wiring digital logic circuits, let's look at realizing  $Y = A \bullet B \bullet C$  using two 2-input AND gates. AND gates are manufactured as integrated circuits or IC's. Because these IC's are cut from a silicon wafer, they are often referred to as *chips*. A typical 14-pin IC is shown in [Figure 0-5](#). A notch on the left end of the chip is part of the packaging design to help the user identify the pin numbers. If you hold the chip so that the notch is towards your left and the pins are pointing down, then the pins are numbered on the near side from left to right starting from 1, then continuing with the pins on the far side from right to left.

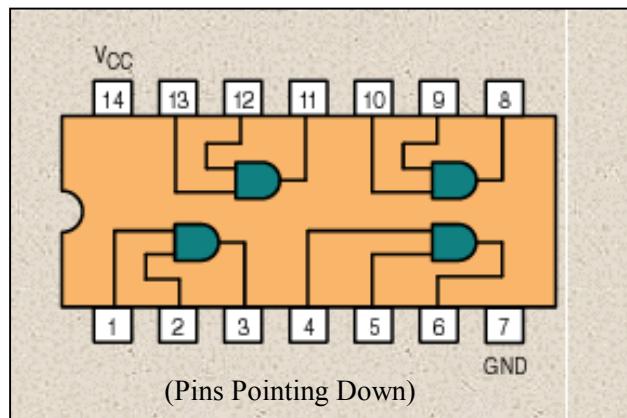


(Copyright of Motorola. Used with permission)

**Figure 0-5.** A 14-pin IC chip.

To work with the IC, our first task is to insert it into the breadboard. The chip should ***always*** be inserted so that it straddles the gutter as shown in [Figure 0-7](#). In that position, pins on opposite sides of the chip are electrically isolated from one another. The gutter isolates one side of the chip from the other. Once a chip has been placed on a breadboard, you can connect wires to any pin on the chip using holes vertically adjacent to that pin. To insert a chip in the breadboard, first make sure its pins are straight. Next, set it on the breadboard, and be sure the pins line up well with the holes they are to go in. Then slowly and carefully push the chip in. ***Be warned:*** it is possible to flip the chip and get the pins embedded in your finger if you are not careful! Extracting chips from the breadboard can also be more difficult than it seems. If you are not careful, it is again possible to bend the pins or flip the chip and get it embedded in your finger. One way to remove an IC from the breadboard is to slide a wire under the long axis of the chip so the chip is "looped" and raise the chip by pulling on both ends of the wire. Also slipping a screwdriver, key, or pencil under the chip and prying it out gently works well.

All the chips have numbers on them indicating what type of gates they contain. To build the simple logic circuit in [Figure 0-4](#), you will need the 7408 chip, a.k.a. quad two-input AND-gate IC. The quad means that there are 4 gates on one chip. A schematic of the chip and the pin numbering scheme is shown in Figure 0-6. (Similar [Pin-Out Diagrams](#) for other IC's are given at the end of each laboratory exercise.)



(Copyright of Motorola, Used with permission)

**Figure 0-6.** Motorola quad 2-input AND gate (MC74F08).

To build the circuit of Figure 0-4, follow the steps outlined below and shown in Figure 0-7.

1. Be sure the power is turned off on your trainer board.
2. Insert the chip over the gutter as shown in Figure 0-7. Make sure that the notch is to your left. (You can insert the chip with the notch on the left or right; however by keeping the notch on the left, your completed circuit will look the same as that of Figure 0-7 and the subsequent steps will be easier to follow.)
3. Connect the trainer board +5 V. supply to a bus (Wire-1). After you do this, the entire bus is energized to + 5 V. (Remember the buses are connected in a horizontal manner.)
4. Connect the trainer board GND to a bus (Wire-2).

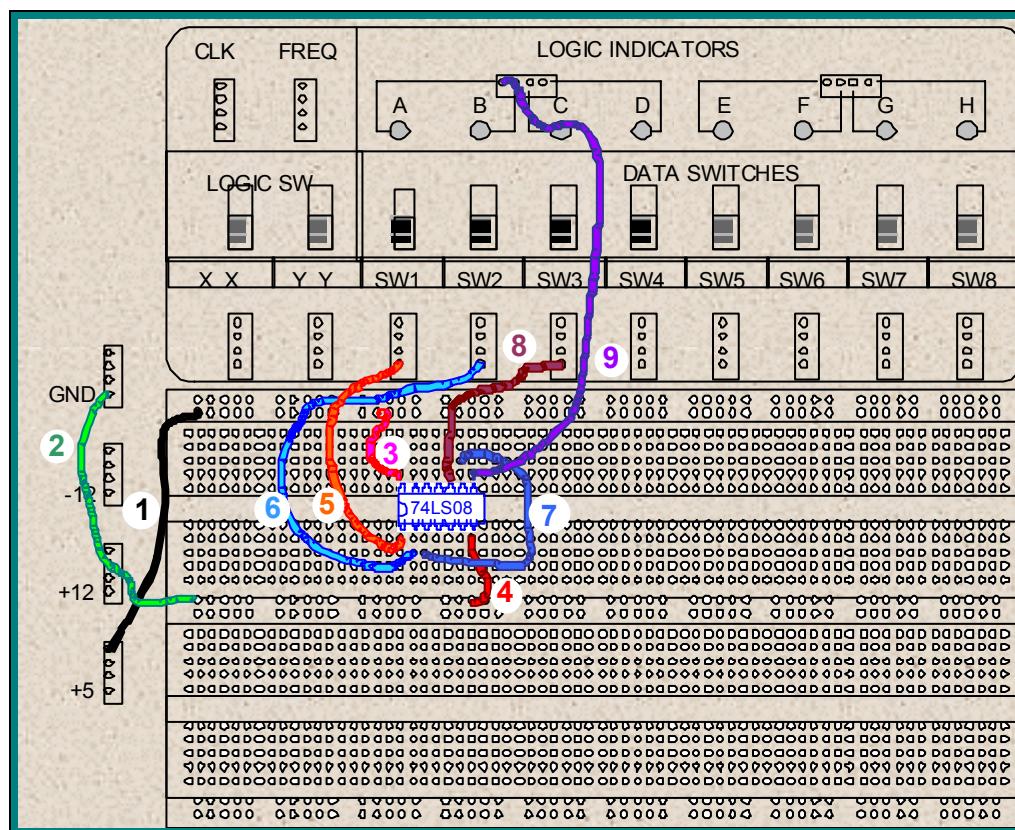


Figure 0-7. Trainer board wiring example for circuit of [Figure 0-4](#).

5. Now we will connect the 5 V. and ground buses to the appropriate chip pins. (You will find it a preferable habit to make the  $V_{CC}$ <sup>2</sup> and GND connections the first connections you make to each IC.) Connect the 5 V. bus to pin 14 (Wire-3) and the ground bus to pin 7 (Wire-4) as shown in

<sup>2</sup>  $V_{CC}$  is understood by convention to be the power supply connection point. (The notation  $V_{CC}$  comes from a convention used first in the analysis of the vacuum tube circuits and subsequently adopted for the transistor circuits. This convention was carried over and used in labeling the power supply pin of the IC schematic when transistor-transistor logic (TTL) circuits were integrated onto a single silicon chip.)

[Figure 0-7](#). (It is good lab practice to run wires around the sides of the IC rather than over the top of the IC. This will allow you to easily replace the IC if it proves to be defective.) By connecting the V<sub>CC</sub> and GND pins of the chip to the appropriate power supply connections, you have effectively ‘plugged-in’ the IC<sup>3</sup>. (Just as all of our home appliances have two-pronged plugs, ‘plugging in’ the IC requires two connections.)

6. The circuit in [Figure 0-4](#) has three inputs (A, B, and C). We will use three switches to control the values of these input variables. Connect Wire-5 from switch SW1 to Pin-1 on the IC. (Note from [Figure 0-6](#) that pins 1 and 2 are inputs to one AND gate and pin 3 is the output signal.) You will use switch SW1 to control the “A” variable input to your circuit. (You may use a portion of a PostIt™ note to label this switch as “A”.)
7. Similarly connect Wire-6 from switch SW2 to Pin-2. This switch will control the “B” input.
8. The output of this AND gate is the function G=A•B. Next, connect this output to the input of another AND gate; connect Wire-7 from Pin-3 (output of first gate) to Pin-9 (input of another gate).
9. Next connect input C. Connect Wire-8 from switch SW3 to Pin-10.
10. Pin-8 is the desired output function Y=A•B•C. To display the voltage value on pin 8, connect it, via Wire-9, to an LED.
11. Flip on the power switch on your trainer board, ①, push switches 1 through 3 into the up position and observe the LED to which you connected your output. It should light up indicating that the output is a digital 1. If it glows, try moving switches into the down position and observe the value of the function Y. Does the function Y behave as you expect when you toggle the values of A, B, and C?

**Debugging<sup>4</sup>:** If your circuit does not work, any number of things may be causing the problem. A voltmeter and/or a logic probe can help you debug your circuit, but before you learn how to use these instruments, you can try the actions on list below:

1. Make sure your trainer board power supply is turned on.
2. Check that the V<sub>CC</sub> and GND pins of the chip are properly connected.

---

<sup>3</sup> Even if not connected to a power supply, some IC's will appear to function correctly for some input signal values. The reason this occurs is beyond the scope of this manual. Suffice it to say that ignoring the power supply connection will eventually cause unexpected and distressing results from your circuit.

<sup>4</sup> The term "debugging" comes from the old days when digital devices were made from relays and vacuum tubes rather than transistors. To make logic functions, and hence computers, operate, relay contacts needed to open and close under program control. Insects, or bugs, would find their way between the relay contacts preventing the contacts from closing. To make the computer work properly, a technician would have to "get the bugs out" of all the relays – hence the term debugging.

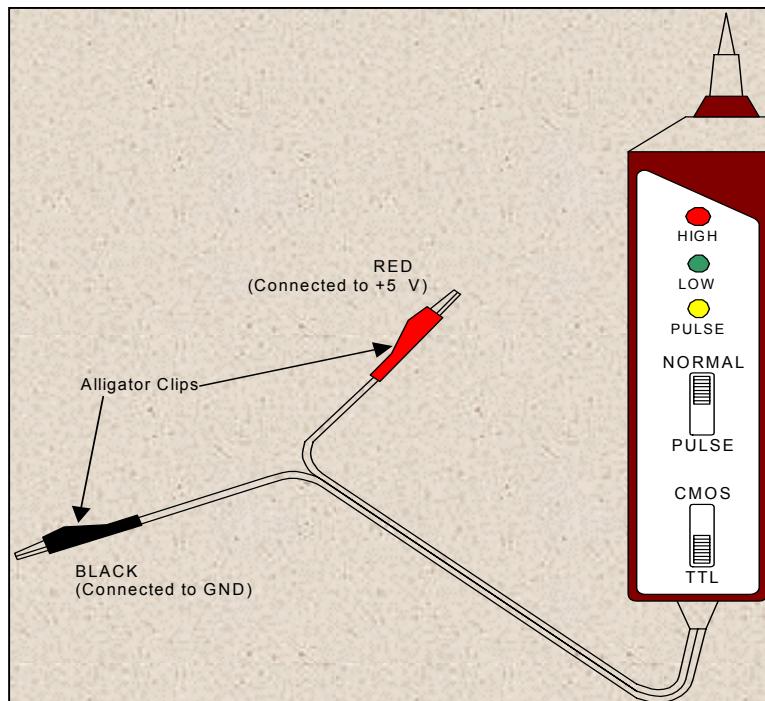
3. Connect your output to another LED. (Sometimes LED's burn out.)
4. Wiggle all of your wires and observe the results. (Sometimes the wires you insert may not make a proper connection.)
5. Move some wires to other holes in the same vertical group. (The spring-loaded mechanism inside the holes of the breadboard gets worn out over time. The connection may not be made properly inside the breadboard even though you shoved a wire in the hole.)
6. Use different AND gates on the IC. (Sometimes gates become defective.)
7. Replace the IC. (Sometimes IC's becomes defective.)
8. Replace the wires that you used. (Wires may break inside the insulated covering.)

When building complex circuits, it is good practice to test and debug as you go. This means building a reasonably small piece of the circuit, turning the power switch on to test it, then turning the power switch off to build the next piece. Testing and debugging as you go saves a tremendous amount of debugging time later. Your textbook may suggest other debugging techniques that may be helpful. Using these techniques will usually identify your problem; however, the problem with a faulty circuit can be determined more precisely and more quickly using either a logic probe or a voltmeter.

## Logic Probe

You can use a logic probe to check the digital value (1 or 0) of any lead or IC pin in your circuit. A schematic of one type of logic probe is shown in [Figure 0-8](#). (The logic probe you use may look different than this and be controlled in a different way. Check with the laboratory assistant or for the logic probe's user's manual for instructions on its use.)

Notice the high (red), low (green), and pulse (yellow) LED's on the logic probe. When the red LED lights up, it means that the probe detects a high digital value (1). When the green LED lights up, a low digital value (0) is detected.



**Figure 0-8.** Logic probe.

To use the probe, move the top switch to NORMAL and the bottom switch to TTL. Connect the *red* clip to V<sub>CC</sub> and the *black* clip to GND. (This provides power, i.e., ‘plugs in the probe,’ and provides the reference levels, i.e., +5 V. and 0 V. for maximum and minimum voltage measurements respectively.) If you touch the tip of the probe to +5 V. on the breadboard, the red LED will light up and you will hear a high pitched tone<sup>5</sup>. If you touch the tip to GND, the green LED will light up and you will hear a lower pitched tone.

Test the circuit that you have built using the logic probe. First make sure that power to your trainer board is turned on. Next, touch the probe to pin 14 of the 7408 IC. It should be high. Then test whether pin 7 of the IC has been connected to GND. (As mentioned above, the first step in debugging a circuit is to check that IC’s have their V<sub>CC</sub> and GND pins connected properly.) You will be making extensive use of the logic probe throughout your laboratory experience. If you have difficulties or questions with any instrument ask a laboratory assistant for help.

## Digital Volt-Meters (DVM)

The logic probe is convenient and easy to use; however it tells you only that a signal is a digital 1 or 0, it does not indicate how far from +5 V. or 0 V. the signal strays. A digital voltmeter (DVM), also called a digital multimeter (DMM), provides the voltage values maintained by each signal; hence the voltmeter is a debugging tool which provides more information than the logic probe. The voltage reading from the DVM can be used to detect such problems such as unconnected inputs, two or more outputs connected together, missing ground connections, and defective IC’s. In TTL chips, as in most types of digital logic, information is passed in terms of voltage. **The output voltage of a properly functioning TTL chip is guaranteed to be between 0 and 0.4 V. for a digital 0 and between 2.4 and 5 V. for a digital 1.** Similarly, to guarantee that a digital circuit responds correctly, input values must be between 0 and 0.8 V. to be interpreted as a digital 0, and 2.0 and 5.0 V. to be interpreted as a digital 1. Can you guess why the output and input specifications are different? Hint: uncontrollable stray electrical signals, known as electrical noise, can affect the output voltage in an unpredictable way.

Figure 0-9 shows a picture of the 8840A digital multimeter. It is called a multimeter because it can measure current, resistance, and frequency in addition to voltage. There are many styles of DMM’s. The one you use in your lab may differ from the one described here. Ask a laboratory assistant for help if you have difficulty using your DMM.

---

<sup>5</sup> When the lower switch on the logic probe is pushed down to the transistor-transistor logic (TTL) mode, the probe indicates a digital 0 or 1 when the voltage it measures is within the range (typically) 0-0.8V. and 2.2-5.0V., respectively. When the logic probe is pushed into the up position, the complementary metal oxide semiconductor (CMOS) mode, the probe indicates a digital 0 or 1 when the voltage it measure is within the range (typically) 0-1.5V. and 3.5-5.0V., respectively. The logic probe you use may have a slightly different range. Check the user’s manual of the probe if you are curious about your probe’s capabilities or specifications.



**Figure 0-9.** 8840A digital multimeter. (Reproduced with permission of FLUKE Corp.)

The top left corner of the 8840A is labeled “INPUT.” There are two input jacks marked “HI” and “LO.” You will connect the LO input to the ground reference (low) point for all voltage measurements, and the HI input to the signal whose voltage you wish to measure. (Recall that voltage is always measured between two points; one point [usually ground] serves as a reference.) To do this, first set up your voltmeter to measure DC<sup>6</sup> voltage by selecting “V DC” on the meter. Next insert cables into the HI and LO DMM terminals, connect an alligator clip to the end of each cable, and clip each alligator clip to one end of a bare wire. Insert the other end of the bare wire attached to the LO cable into the GND connector in your trainer board. (This connection will serve as the ground reference point for your voltage measurements. Keep this lead connected to ground for all measurements.) Now turn on your voltmeter and prototype board. To measure the actual value of the +5 V. supply delivered by your trainer board, insert the wire attached to the HI terminal cable into the trainer board’s +5 V. supply terminal. Your voltmeter should measure a value within one-tenth of a volt of 5.0 V. If your DMM reading is outside of the range 4.9–5.1 V., ask a laboratory assistant to check your set-up. Next try measuring some of the input and output signals of the circuit you built earlier. The values you measure for digital 1’s and 0’s should be close to +5 and 0 V. respectively. Are they as close as you expected?

Clearly the DMM provides more information about the signals at the input and output pins of digital IC’s. Why then use a logic probe? We use logic probes because they are less expensive instruments, are easier and faster to use, and the information they provide is sufficient in most cases.

You may find other instruments on your bench such as signal generators and power supplies. Connecting outputs from these devices to terminals on your trainer board may cause damage to the trainer board or the other instruments. Ask a laboratory assistant for help if you are curious about these other instruments.

<sup>6</sup> DC is the abbreviation for direct current. All digital devices operate with direct current. This stands in contrast to home appliances that use alternating current, or AC power.

## SELF-ASSESSMENT WORKSHEET

---

No laboratory report is required for this laboratory exercise; however you may be asked to submit this self-assessment worksheet so your instructor can determine whether the designed outcomes for this laboratory exercise were achieved.

Put an “X” in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use ‘5’ to indicate that you ‘strongly agree’, ‘3’ to indicate that you are ‘neutral’, and ‘1’ to indicate that you ‘strongly disagree’. Use ‘NA’, ‘Not Applicable’, when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your responses will not be graded. They are for your instructor’s information only.

**Table \_\_:** Self Assessment of Outcomes for Hardware Lab 0: Using a Prototype Board, Logic Probe & Voltmeter.

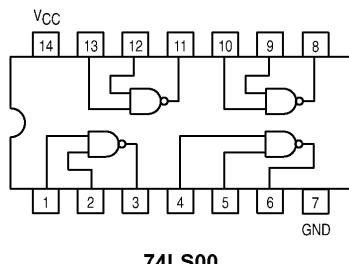
<b>After completing the assigned tasks and report, I am able to:</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>NA</b>
Insert an IC in the proper location in a breadboard.						
Wire the IC to ground and the +5 Volt (+5V.) power supply.						
Connect pins of an IC to realize a combinational logic circuit.						
Use a logic probe to determine a signal’s logic levels.						
Use a voltmeter to determine a signal’s voltage levels.						

---

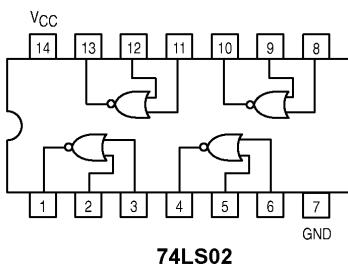
Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

## PIN-OUT DIAGRAMS

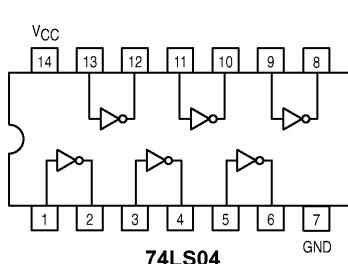
The pin-outs of several TTL devices are given below.



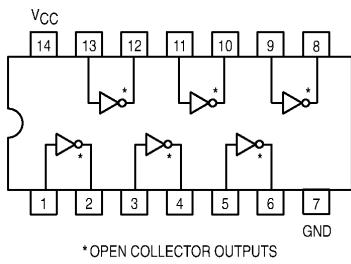
**74LS00**



**74LS02**

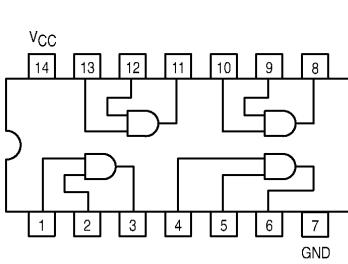


**74LS04**

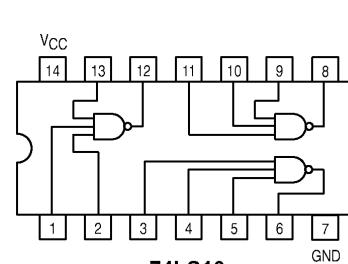


\* OPEN COLLECTOR OUTPUTS

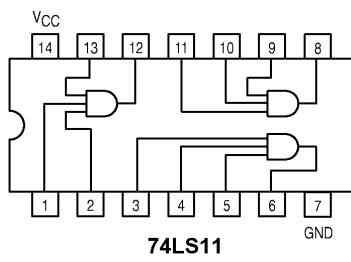
**74LS05**



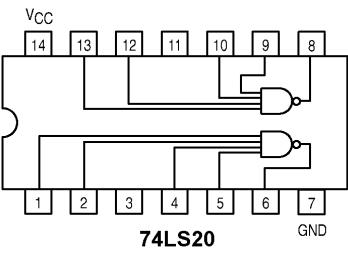
**74LS08**



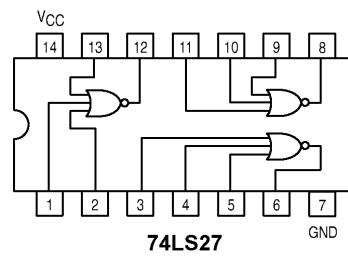
**74LS10**



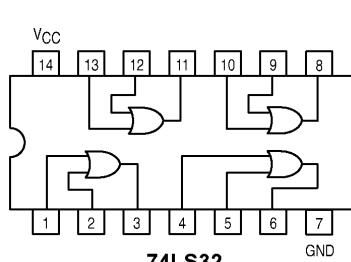
**74LS11**



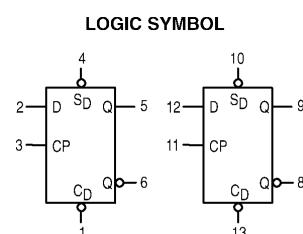
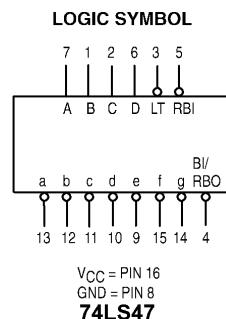
**74LS20**



**74LS27**

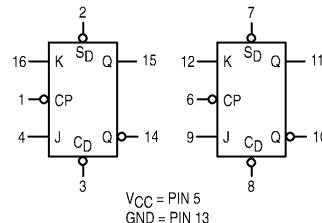


**74LS32**



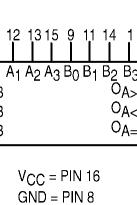
(Copyright of Motorola, Used with permission)

**LOGIC SYMBOL**

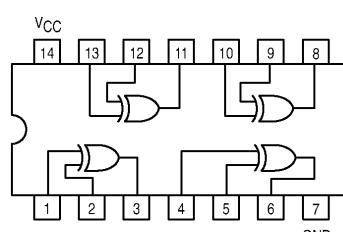


**74LS76**

**LOGIC SYMBOL**

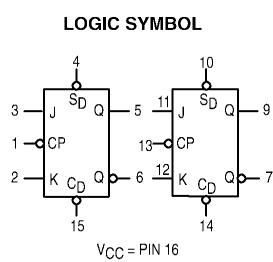


**74LS85**

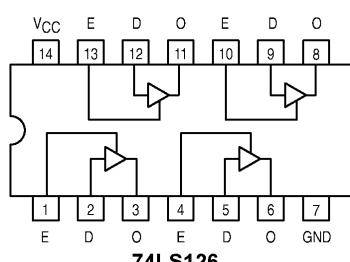


**74LS86**

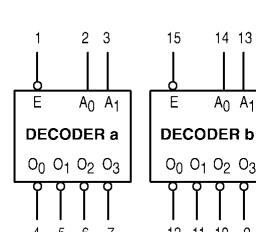
**LOGIC SYMBOL**



**74LS112**

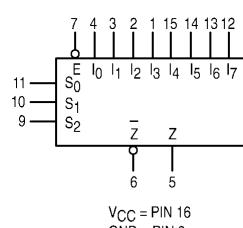


**74LS126**



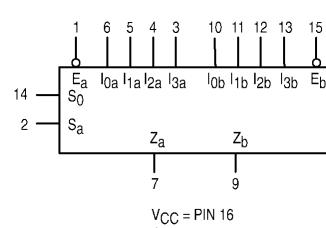
**74LS139**

**LOGIC SYMBOL**



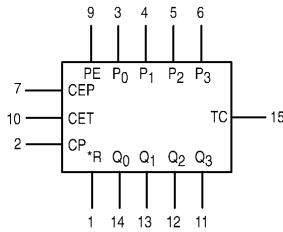
**74LS151**

**LOGIC SYMBOL**



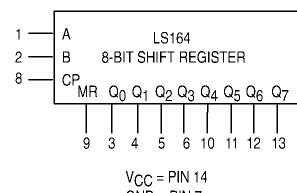
**74LS153**

**LOGIC SYMBOL**



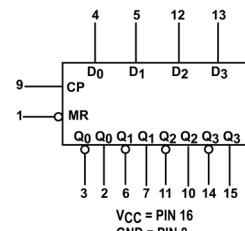
**74LS163**

**LOGIC SYMBOL**



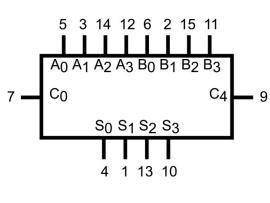
**74LS164**

**LOGIC SYMBOL**



**74LS175**

**LOGIC SYMBOL**



**74LS283**

(Copyright of Motorola, Used with permission)

# HARDWARE LAB 1: DEBUGGING A HALF AND FULL ADDER

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Use a prototyping board and a logic probe (see [Lab 0: Hardware Lab Basics](#) if you need help.)
- Interpret a truth table.
- Read Boolean algebraic notation.
- Recognize the Boolean algebraic expressions characterizing a full adder.

**Equipment:** Digital Trainer Board, Logic Probe

**Integrated Circuits:** You will need the following IC's to complete the tasks in this lab:

- (1) 7404 (Hex Inverters)
- (1) 7408 (Quad 2-input AND gate)
- (1) 7432 (Quad 2-input OR gate)
- (1) 7486 (Quad 2-input XOR gate)

**Objectives:** In this laboratory you will gain some experience building and debugging combinational logic circuits using TTL IC's. You will also gain experience using canonical sum-of-products and product-of-sums forms along with the exclusive OR (XOR) operator.

**Outcomes:** When you have completed these laboratory experiments you will be able to:

- Describe the truth tables that characterize the addition of two single bit numbers.
- Write the Boolean algebraic expressions that characterize the sum and carry functions for the half adder using both the product-of-sums and sum-of-products canonical forms.
- Realize the sum and carry functions using TTL hardware.
- Debug TTL circuits using an LED and Logic Probe as testing instruments.
- Write the Boolean algebraic expressions that characterize the sum and carry functions for the full adder.
- Build and debug a full adder.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab it is recommended that you use a task-oriented format when organizing your lab report. (If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Task-Oriented Report Writing Guidelines](#).)

---

## Prologue

The hardware labs are intended to give you hands-on experience in designing, building, and debugging simple logic circuits. Each phase of the realization process (designing, building, and debugging) is like a link in a chain; if one link breaks then the process breaks down. The skills you acquire by completing these laboratory experiments, (i.e., carefully executing sound circuit design procedures, building circuits carefully, intelligently tracking down your lab partner's mistakes, and completely documenting what you have done and how you did it), are particularly important as you progress to design ever more complicated circuits. As circuits become more complicated, they will test your skills more thoroughly and will uncover any weaknesses. The artifact of poor design and circuit-prototyping practices are circuit bugs, which can be a major problem for complex circuits. This lab exercise will start you in the process of building sound circuit design and hardware laboratory practices.

## Preparation

The best way to prepare for a hardware laboratory experiment is to first make sure that you have the prerequisite skills listed on the first page of each experiment. Next, read the entire experiment, and complete the designs of the circuits to be built. Then, draw schematics of the circuits you wish to build, labeling both ends of any connections with the pin numbers of the IC's that are to be interconnected. You will find that these diagrams (provided for you in Task 1-1) will help you complete the experiment more quickly and will minimize the time you spend debugging your design. Good luck!

## Pre-Lab Knowledge

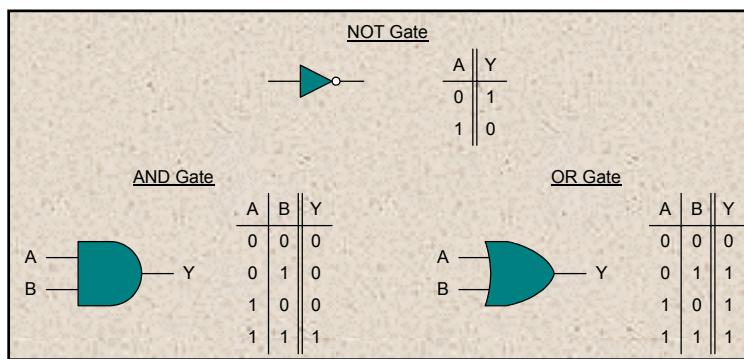
It is assumed that you have acquired certain fundamental knowledge about digital circuits from the lecture portion of your course. To understand this pre-lab material you will need to understand how to interpret a truth table and be familiar with Boolean algebraic notation. This pre-lab discusses the concepts that we will use in designing the circuits you will build in the experiments described in this laboratory manual. Reading through this material will help you understand these concepts so you can use them in subsequent lab experiments. Reading this material will also help you understand any differences between the notation

used in your course text and this manual. If you do not understand something in this experiment, see your laboratory-teaching assistants. They will be able to answer all of your questions.

If you are already familiar with the truth tables of AND/OR/NOT gates, skip the next section and go to [Half Adder Circuit](#).

## Primitive Gates

The most elementary logic device is the inverter or the NOT gate. It has the truth table and schematic symbol shown in Figure 1-1. This truth table shows that the inverter has one input and one output; if the input is 1, the



**Figure 1-1.** Basic logic gates and truth tables.

output is 0, and vice-versa. Next in simplicity are the AND and OR gates, whose truth tables and schematic symbols are also shown in Figure 1-1: both have two inputs and one output. The AND gate has an output of 1 only if both inputs are set to 1. The OR gate has an output of 0 only if both of its inputs are set to 0. Alternatively, an AND gate has an output of 0 if any one of its inputs is set to 0 and an OR gate has an output of 1 if any one of its inputs is set to 1.

## Half Adder Circuit

Suppose we have a truth table for a useful circuit that we want to build. How do we build it out of these elementary gates? Let's look at an example that you will also be examining when you perform the simulation experiments described in this manual, the half adder. (If you are familiar with the product-of-sums and sum-of-products designs for a half adder, skip to section: [Building A Circuit with TTL IC's](#); otherwise, let's look at building a half adder [HA].)

As you might guess from the name of the half-adder circuit, it performs an addition operation. Let the notation “B” designate a number that we wish to interpret as a binary number, and let “D” designate a decimal number. (A number without any designator is assumed to be a digital logic value.) The addition of two 1-bit<sup>1</sup> binary numbers is defined by the following equations:

$$\begin{array}{cccc}
 & & 1 \text{ (Carry)} & \\
 \begin{array}{c} 0B \\ +0B \\ \hline 00B \end{array} & \begin{array}{c} 0B \\ +1B \\ \hline 01B \end{array} & \begin{array}{c} 1B \\ +0B \\ \hline 01B \end{array} & \begin{array}{c} 1B \\ +1B \\ \hline 10B \end{array} \\
 & & \begin{array}{l} \text{Carry Bit} \\ \text{Sum Bit} \end{array} &
 \end{array}$$

<sup>1</sup> Bit is the shortened form of the expression “Binary Digit.”

(It is hard to believe you get post baccalaureate credit for adding  $0 + 1$  isn't it?) The only equation above that may be difficulty to justify is  $1B + 1B = 10B$  – let's justify it. In the binary system, the symbol with the largest value is  $1B$ , in the base-ten system the largest symbol is  $9D$ . When we add two numbers in the base-ten system whose sum exceeds our largest symbol, we generate a carry of  $1D$  to the next more significant position. Similarly, when we add two numbers in the binary number system whose sum exceeds our largest symbol,  $1B$ , we generate a carry of  $1B$ . This carry is added to the numbers residing in the more significant position (which are  $0$ 's in our example). If we assign the most significant bit of our 1-bit addition to be the carry (CRY) bit and the least significant bit to be sum (SUM) bit, we can represent the addition operation of two one-bit operands,  $A$  and  $B$ , using the binary-valued, addition-definition table of [Figure 1-2](#). With digital logic gates we simulate truth values, not binary numbers; however (and this is a subtle yet important isomorphism) if we reinterpret the binary values as truth (or logical) values, then we get the truth table of [Figure 1-3](#) – and we do know how to simulate truth values with digital logic gates!

A	B	A+B
0B	0B	00B
0B	1B	01B
1B	0B	01B
1B	1B	10B

**Figure 1-2.** Half-adder definition table.

A	B	CRY	SUM
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

**Figure 1-3.** Half-adder truth table.

To build a circuit that will add two 1-bit numbers,  $A$  and  $B$ , and give a 2-bit result, we will need to build the two digital logic functions defined in [Figure 1-4](#), the SUM and CRY. We will use two canonical<sup>2</sup> forms in realizing these two functions: Sum-of-Products (SOP) and Product-of-Sums (POS) canonical form.

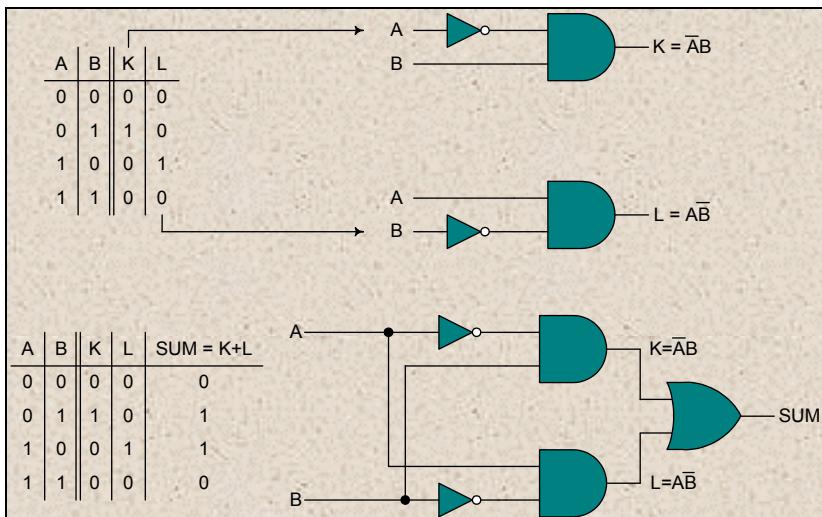
A	B	SUM	A	B	CRY
		0			0
0	1	1	0	1	0
1	0	1	1	0	0
1	1	0	1	1	1

**Figure 1-4.** Truth tables for SUM and CRY.

Let's use the SOP canonical form to realize the SUM function of Figure 1-4. To do this, define two auxiliary functions,  $K$  and  $L$ , whose values are 1 for one and only one of the input combinations that cause the SUM function to be 1. These auxiliary functions are shown in [Figure 1-5](#). If we OR the  $K$  and  $L$

<sup>2</sup> A canon is a fundamental law or axiom. A canonical form is a form that is in some sense fundamental. The forms we use here are canonical in the sense that each AND or OR term contains every variable used in defining the function.

functions together we get the result  $K+L^3$  shown in the truth table of [Figure 1-5](#). By inspection we can see that  $SUM = K+L$ . This means that if we can realize the  $K$  and  $L$  functions, we can realize the  $SUM$  function as the OR combination of  $K$  and  $L$ . By inspection we can see that we want  $K$  to be 1 when  $A=0$  and  $B=1$ . Clearly, this will be the case if we set  $K = \overline{A} \bullet B$ . Further,  $K$  must be equal to 1 ONLY when  $A=0$  and  $B=1$ , which is the case for the function  $K = \overline{A} \bullet B$ ; hence  $K$  can be realized using the circuit shown in Figure 1-5. Similarly, the function  $L$  must equal 1 when and only when  $A=1$ , and  $B=0$ . By inspection, the function  $L = A \bullet \overline{B}$  meets this requirement and can be realized using the circuit of Figure 1-5. If we connect the functions  $K$  and  $L$  to the inputs of a 2-input OR gate of Figure 1-5, we get the  $SUM$  function that we want. This circuit is the hardware equivalent to the Boolean algebra expression  $SUM = AB + A\overline{B}$ .



**Figure 1-5.** SOP implementation of sum for 1-bit half adder.

The other form we want to use for expressing the  $SUM$  function is the canonical Product-of-Sums (POS) form. Let's define two auxiliary functions,  $M$  and  $N$ , whose values are 0 for one and only one of the input combinations that cause the  $SUM$  function to be 0. These auxiliary functions are shown in [Figure 1-6](#). If we AND  $M$  and  $N$  together we get the result  $M \bullet N$  shown in the table of Figure 1-6. By inspection we can see that  $SUM = M \bullet N$ . This means that if we can realize the  $M$  and  $N$  functions, we can realize the  $SUM$  function as the AND combination of  $M$  and  $N$ . By inspection we can see that we want  $M$  to be 0 when and only when  $A=0$  and  $B=0$ . Clearly, this will be the case if we set  $M = A + B$ . The function  $M$  can then be realized using the circuit shown in Figure 1-6. Similarly, the function  $N$  must equal 0 when and only when  $A=1$ , and  $B=1$ . By inspection, the function  $N = \overline{A} + \overline{B}$  meets this requirement and can be realized using the circuit of Figure 1-6. If we connect the functions  $M$  and  $N$  to the inputs of a 2-input

<sup>3</sup> We will use the standard Boolean algebraic notation in which “+” is used to indicate the OR operation, “•” is used to represent the AND operation, and an over-bar “—” is used to indicate a NOT (inverting or complementing) operation.

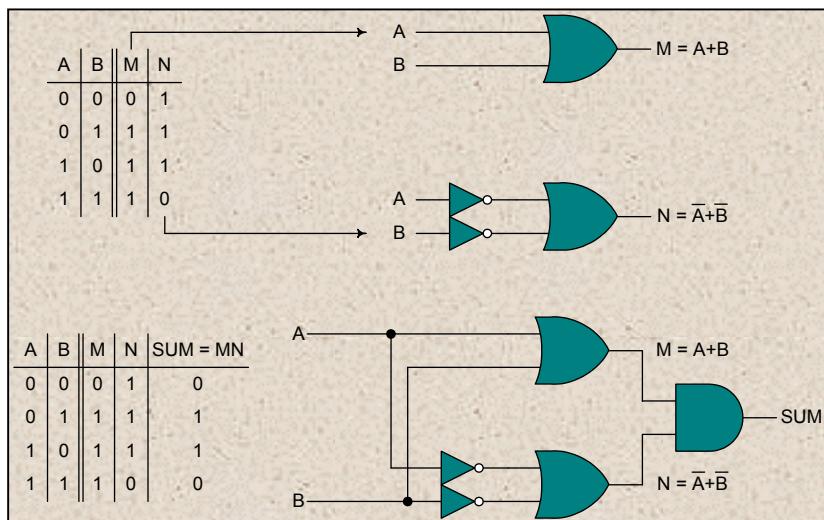


Figure 1-6. POS implementation of SUM for 1-bit half adder.

(AND) gate for its realization. (Verify for yourself that this schematic diagram describes one circuit for realizing the CRY function.)

The POS implementation is shown in [Figure 1-8](#). (Verify for yourself that this will also realize the CRY function.) Clearly the POS form requires many more logic gates and connections than the SOP form. Which would you rather build in the lab, the SOP or POS form for the CRY? Most of us would rather build the form that takes less time to construct and less time to debug. Industry likewise prefers the form with the fewest gates<sup>4</sup> and circuits with fewer gate connections because it can be made smaller. Circuits with fewer connections and gates have fewer pieces that can become defective. Consequently, these circuits will perform more reliably.

If we wish to design a circuit using a minimal number of gates, is there a way of knowing *a priori* whether the SOP or POS canonical forms will yield the more minimal design? The answer is yes. If the function column of the truth table has fewer 1's than zeros, a canonical SOP form will be more minimal. If the function column of the truth table has the same number of 1's and 0's, both canonical SOP and POS forms require the same number of gates; otherwise the POS form is more minimal.

AND gate of Figure 1-6, we get the SUM function that we want. This circuit is the hardware equivalent to the Boolean algebra expression  $\text{SUM} = (A + B) \bullet (\bar{A} + \bar{B})$ .

Similarly the CRY (carry) output can be expressed using the canonical SOP and canonical POS forms. The SOP form as shown in [Figure 1-7](#) requires only one logic

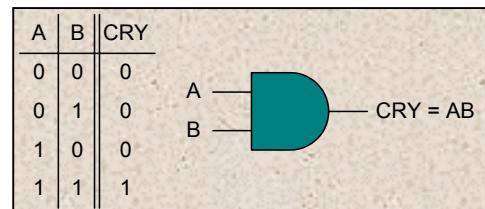


Figure 1-7. SOP implementation of CRY for 1-bit half adder.

The answer is yes. If the function column of the truth table has fewer 1's than zeros, a canonical SOP form will be more minimal. If the function column of the truth table has the same number of 1's and 0's, both canonical SOP and POS forms require the same number of gates; otherwise the POS form is more minimal.

<sup>4</sup> Circuits are not always built using the realization that requires the fewest gates for a variety of reasons. Sometimes circuits are constructed using only one gate type, which will often require more gates. Sometimes extra gates are added to circuits so that all of their capabilities are testable. Also, to avoid glitches, (see your textbook for the technical definition of glitch) circuit designs will require extra gates.

Your textbook describes ways of obtaining minimal two-level<sup>5</sup> (non-canonical) realizations for logic functions. There is no way of knowing *a priori* whether the minimal (non-canonical) SOP or POS forms will require less logic.

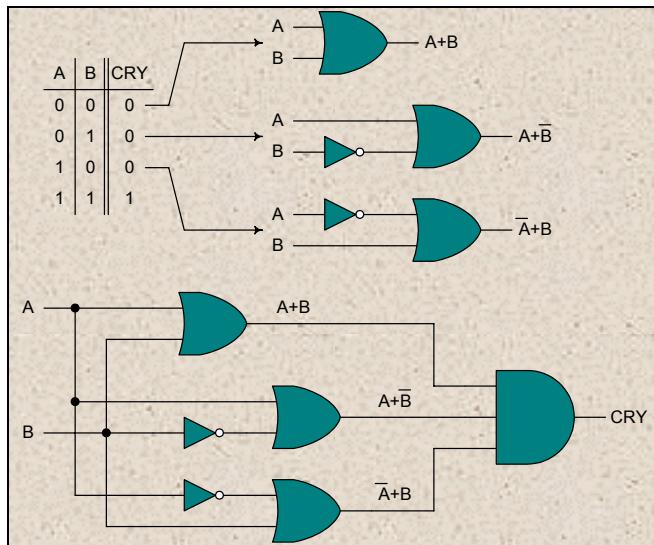


Figure 1-8. POS implementation of CRY for 1-bit half adder.

## Building a Circuit with TTL IC's

The first step in circuit design is to look through a catalog of IC's (provided free by digital logic chip manufacturers) to see which chips are available and determine what combination of IC's you will need to build the functions you want to build. In these laboratory exercises you will be using 7400 series TTL chips. The last pages of this (and each) lab contain the [Pin-Out Diagrams](#) for IC's used in these lab experiments. (Those pages also contain some additional IC's that you may find useful for other applications.) Peruse the Pin-Out Diagrams for the chips you need in order to build the half-adder circuits we have designed. The IC's you probably picked are the 7408 (AND), the 7432 (OR), and the 7404 (NOT).

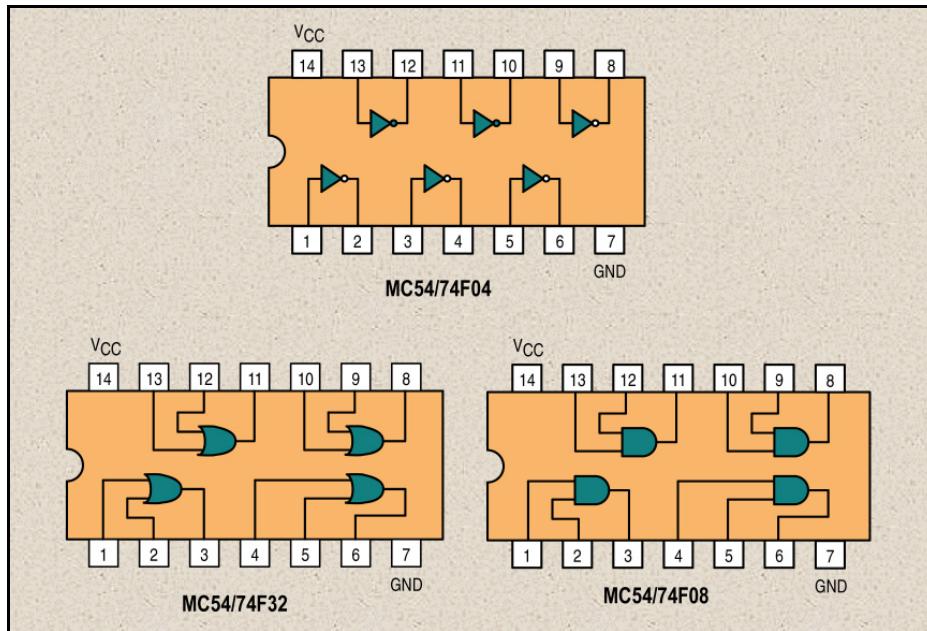
## Setting up

First, gather all of the equipment needed for this experiment as listed below:

- Prototype board
- Logic Probe
- (1) 7404 IC
- (1) 7408 IC
- (1) 7432 IC

<sup>5</sup> All canonical POS and SOP realizations are two-level realization; that is, the input signal must travel through two AND/OR gates

The IC's you need have the pin-outs shown in [Figure 1-9](#). (We will need this schematic to know which pins are to be connected to the +5 Volt supply (pin 14 labeled V<sub>CC</sub>) and ground (pin 7 labeled GND), and which pins are inputs and outputs. If you are not sure how to insert an IC in the trainer board, connect it to the power supply, and wire its inputs and outputs to realize a circuit, review the directions in [Hardware Lab 0](#).)



**Figure 1-9.** Pin-outs for the 7404, 7432 and 7408 chips. (Copyright of Motorola, Used with permission)

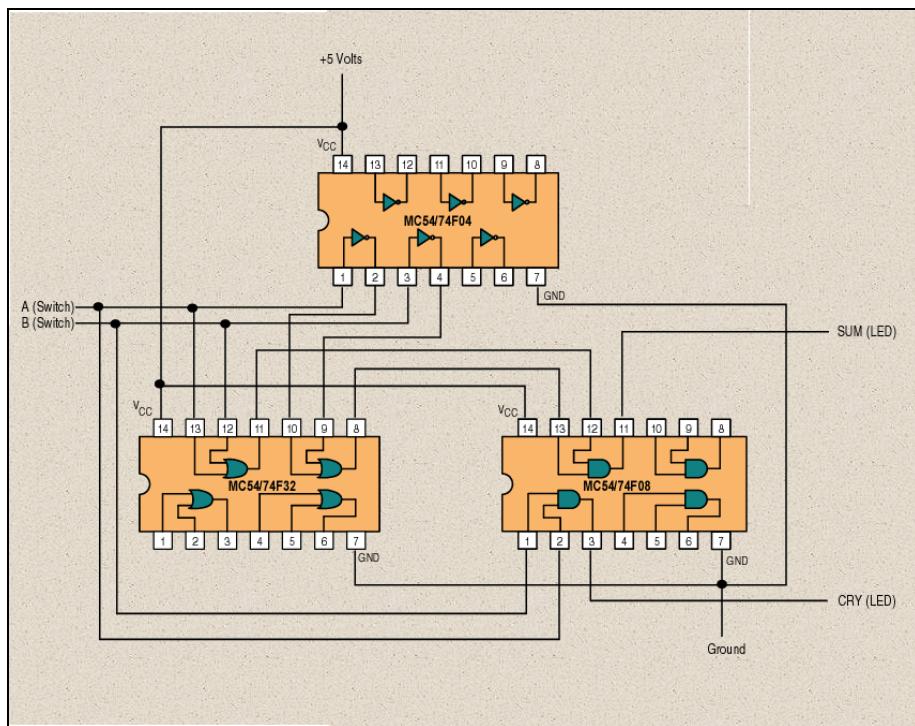
Using these IC's, we wish to build the half adder using the minimum number of gates necessary; hence we'll pick the canonical POS form for the SUM function and the canonical SOP form for the CRY function. One possible layout that will realize this circuit is shown in [Figure 1-10](#). Check the layout carefully to see if you agree that it is correct.

### Task 1-1: Build the 1-Bit Half-Adder

With the power off, build the half adder using the design given in Figure 1-10 on a digital prototyping board. Use switches to drive the inputs (A & B) of the circuit and LED's to view the output (SUM & CRY) voltages or digital logic levels. Be careful that you have the circuit set up correctly and have the V<sub>CC</sub> and GND connected correctly. Also, be sure to use the [effective wiring practices](#) described below.

---

before its effect on the output can be detected. (The presence or absence of inverters is ignored in this definition.)



**Figure 1-10.** Possible layout for 1-bit half adder.

## Effective Wiring Practices

When physically making wire connections on a breadboard or trainer board, there are some practices that will eliminate problems before they start.

- Strip off about  $\frac{1}{4}$  inch of insulation from each end of the wire. Don't strip too much or too little off of the ends of wires. If too short, a good connection may not be made when the bare end is inserted into the breadboard. If too long, the shafts of two bare wires protruding from the breadboard may come into contact making an unwanted connection.
- Route wires around IC's, not over IC's. By routing wires around the IC, you can eliminate the need to remove then replace these wires if the IC needs to be removed for any reason. IC's need to be removed when you suspect they are defective or when you want to validate the performance of an IC so you can eliminate it as a suspected cause of problems in a problematic circuit.
- Use the buses for V<sub>CC</sub> and GND connections. You will need to make many connections to V<sub>CC</sub> and GND. By making all of these connections to buses, you will find it easier to debug your circuits.
- Never use more than one wire in one hole. Two wires in a hole will quickly cause the spring mechanism inside the breadboard to lose its compression ability, leading to a poor and/or intermittent wire-to-breadboard connection when the hole is used again.

## Task 1-2: Test the SUM & CRY of the 1-Bit Half-Adder with LED

Now turn on the power, test each output (SUM & CRY) by trying all input combinations in the truth table ([Figure 1-4](#)), and record the results of your tests in your lab notebook. If it does not work as you expect, find the errors by debugging it as described in [Hardware Lab 0](#) and the next section. When it does work as you expect, move on to the next task.

## Debugging

In Lab 0, some techniques were listed to help you find problems with your circuits. As your sophistication in the realization of digital logic circuits grows you will learn that circuits are best debugged using a methodical technique. The ordered list below provides the steps professionals use in debugging their circuits.

## Steps Used in Debugging

1. Construct a truth table for each function you wish to realize. (It is impossible to know whether your circuit is operating correctly unless you know how it is supposed to operate.) This has been completed for you in this lab and is listed in Figure 1-4.
2. Measure the V<sub>CC</sub> and GND pin logic values for each IC used in your design. If you measure a value not consistent with a properly powered IC, correct it. A poor V<sub>CC</sub> or GND connection may be hard to detect with an LED or a logic probe. (In [Hardware Lab 2](#) we will examine the effects of incorrect power supply connections.) Visually trace all the wires supplying V<sub>CC</sub> and GND to all the chips and make sure that all V<sub>CC</sub> and GND connections are secure.
3. Methodically observe the output for every combination of input variables for each function. At the first wrong output, proceed to the next step. If the output values are correct for every input combination, congratulations! You have successfully debugged your circuit!
4. Using a layout of your circuit, such as shown in [Figure 1-10](#), write down the digital values that you would expect to measure at each connection point if the circuit was working correctly. The tables of [Figure 1-6](#) and [Figure 1-7](#) containing the auxiliary function values for each input combination may help; although you can easily determine these values by inspecting the schematic diagram and using your knowledge of how AND/OR/NOT gates operate.

5. Working from output back to input<sup>6</sup>, measure and record the logic values of all pins used in the path supplying the erroneous signal.<sup>7</sup> In addition, you may wish to make the measurement at least twice to check whether you have an intermittent fault<sup>8</sup>. When you find a single circuit element whose inputs are correct but whose output is incorrect, you have identified the area of the problem. At this point you need to look at that gate and connections to that gate to determine where the problem is. Concentrating your measurements near the circuit element(s) that you suspect are causing the problem, your measurements might indicate one of the following problems:

- If the signals on the generating and receiving end of a connecting wire differ, then either the wire connecting the two pins is broken within the insulation or the receiving-end pin is shorted to V<sub>CC</sub> or GND inside the IC. If replacing the wire does not help, try replacing the IC. If replacing the IC is successful, return to Step 3.
- If your lab partner has incorrectly wired a circuit, you will reach a point where you identify an input value that is incorrect or missing. Correct your lab partner's wiring error and return to Step 3.
- If a gate is defective, you will reach a point where the output of the gate is inconsistent with the input. Re-wire that part of the circuit using another gate on the IC or replace the IC, then return to Step 3.
- If you make a measurement that has a voltage level<sup>9</sup> that is neither within the 1 nor 0 voltage range<sup>10</sup> or is near a limit of one of these ranges, either the IC is defective or (more likely) you have two outputs connected to the same point. (For example, if you inadvertently connect the wire that should go to pin 13 of the AND gate to pin 12 of the AND gate in Figure 1-10 you would have two potential problems. First pin 13 would be unconnected. Second, pin 12 would have two outputs feeding it.) Correctly wire the circuit and return to Step 3.

Let's look at an example-debugging problem. Suppose you inadvertently connect the wire that should go to pin 13 of the AND gate to pin 12 of the AND gate in Figure 1-10. This wiring error causes the M and N functions to simultaneously drive one input of the AND gate while the other input is left unattached, or

---

<sup>6</sup> If you look at the POS and SOP schematics in this document, you will observe a tree-like structure. Problems with signals on the upstream (input side) of the tree, cause all of the downstream (output side) signals in the path to be corrupted. By working from output back to input, we hope to identify that branch of the tree whose input values are correct but whose output values are incorrect. When this occurs, we have isolated the circuit element causing the problems.

<sup>7</sup> When recording measurements at the pins of IC's, observe if there is a difference in the voltage/logic value between a FROM pin (i.e., the pin or switch generating the signal) and a TO pin (i.e., the down-stream pin receiving the signal).

<sup>8</sup> An intermittent fault is a circuit defect that appears and disappears for no apparent reason. These faults are difficult to detect even for professionals. The most likely cause is a loose breadboard connection or a wire broken within its insulation that is alternately making and then breaking contact. Loose connections within the IC itself may also cause such a problem. If you have an intermittent fault, jiggle all wires (including the V<sub>CC</sub> and GND connections) on your breadboard to see if this can recreate the problem. Then try to make sure all wires are connected securely. As a last resort, replace the IC's in your circuit. If all else fails, ask a laboratory assistant for help.

<sup>9</sup> You'll need a voltmeter to observe this measurement.

<sup>10</sup> The output voltage of a properly functioning TTL chip is guaranteed to be between 0 and 0.4 Volts for a digital 0 and between 2.4 and 5 Volts for a digital 1. You will need a voltmeter to accurately determine if the output voltage of a circuit is out of range.

left ‘hanging’ in the parlance of digital circuit designers. When you check the input conditions that cause both functions M and N to have opposite logic values, your LED and logic probe outputs may not show a low condition for the SUM function<sup>11</sup>. When we detect this problem with an LED or logic probe, we have identified an output that is inconsistent with a line of the truth table of the function – Step 3 above. Moving from output pin 11 upstream, we next measure the voltage/logic values on pins 12 and 13. At this point we will detect an anomaly and know that either the AND gate is defective or the wiring is defective. Since wiring problems are more common than defective AND gates, we check the input and output wiring first. At this point we should detect the problem. If we miss the problem, we isolate the AND gate from the rest of the circuit by moving the IC to a new location on the prototype board and test it independently. If the AND gate functions correctly under this test, we immediately suspect that the cause of the problem is the interaction between the gate and the circuit. We then retrace our steps and recheck the original wiring where we will find the problem. This may seem like a complicated process, but if you approach it methodically, it’s not too bad.

Sometimes, the input/output characteristics of a circuit element may erroneously appear to be defective, when the problem is actually caused by an undetected wiring problem. To determine if a potentially defective circuit element is in truth defective, you will need to isolate that circuit element from the remainder of the circuit, and test it by using switches to control its inputs and LED’s or a logic probe to observe its output. This method of isolating the circuit element in question allows you to positively identify it as the problem and eliminate the possibility that the problem is caused by the interaction of the circuit with another circuit element. If the problem is caused by the interaction of the circuit and other circuit elements, chances are the problem is caused by a wiring error such as two outputs driving a common input or one input driving too many outputs. When the problem is caused by the interaction of two subcircuits<sup>12</sup>, the same rules apply: isolate the subcircuits and independently test their responses (following the steps above) to independent inputs.

Debugging a circuit may seem like a complex problem and indeed it can be vexing at times. (It is at these times that you can avail yourself of the laboratory assistant’s help.) There can be any number of problems that can arise and it is impossible to list them all and all their indications; however most (but typically not all) of your circuit bugs are rather mundane. A list of common problems encountered in the lab are (roughly ordered from most to least frequent):

- $V_{CC}$  and/or GND not connected.
- Breadboard/power not turned on.
- Wires connected to the wrong pin.

---

<sup>11</sup> The voltage value of the SUM function output under this input condition depends on the circuit design internal to the IC and cannot be predicted without knowing the internal circuit design parameters.

<sup>12</sup>We use the term ‘subcircuit’ to mean any collection of logic devices that we group together and think of as a “whole.”

- Wires making a poor or intermittent connection with the breadboard.
- Two outputs driving a common input.
- IC not properly inserted.
- Defective IC's.
- Defective LED's.
- Defective breadboard connection.
- Wires broken inside of the insulation.

## Wiring Rules

In creating the wiring diagram of [Figure 1-10](#) we followed certain mandatory design rules. If you want your circuits to function properly, here are the rules you will want to follow as well:

- Never wire two outputs to one input. (If the outputs have different logic levels, there is no way to predict the output value of the gate receiving these conflicting signals.)
- Never leave an input pin unconnected. (Under some conditions and for certain logic families, the output may respond as desired, but leaving inputs ‘hanging’ will eventually cause you problems. You will explore more about this in [Hardware Lab 2: TTL Characteristics, Three-state Buffers, Open-Collector Buffers](#))
- Obey the fan-out properties of your logic family. (In the TTL family, it takes output current to drive the input of the gates down stream. If you ask a device to drive too many downstream gates, you may exceed the current capability of the device and experience sustained or intermittent problems. ‘Fan out’ refers to the maximum number of inputs one single output can drive without over taxing the IC. While you should check the fan-out capability of each IC, the fanout for most of the TTL devices, is 10 or greater.)

## Task 1-3: Debugging

Turn your back and have your lab partner introduce an error into the circuit. Practice finding the errors using the debugging techniques described above. Be sure to record in your notebook how the circuit was changed and how you used the debugging techniques to discover those changes. Real world digital systems are complicated. The debugging skills you acquire here will serve you well throughout the rest of your education and career.

## Task 1-4: Completely Test your Half Adder Circuit Using a Logic Probe

Using a logic probe, test each point in the circuit for different input combinations. Satisfy yourself that every part of the circuit is working as you think it should, i.e., the functions SUM, CRY and all intermediate outputs, M and N, are consistent with the truth tables shown in [Figure 1-6](#) and [Figure 1-7](#).

## Hardware Realization v. Simulation

In the simulation portion of this laboratory manual (laboratory exercise 2) of you will build the simulation of 4-bit full-adder circuit. In the next task you are asked to build a 1-bit full adder. The objective of asking you to build a hardware realization and simulation of a full adder is to allow you to compare the complexity of building and simulating a digital logic circuit. This will help you draw some conclusions about the role of simulation in the realization of complex digital logic circuits. By extrapolation, you may draw some conclusions about the role of simulation in the realization of any complex engineering system.

## Full Adder

(If you have completed Simulation Lab 2 or are familiar with the derivation that leads to [Figure 1-15](#), skip this section and go to directly to [Task 5](#).)

In the previous tasks you built, debugged and tested a half adder. In the next task you will build a 1-bit full adder. The difference between a half and full adder is that the full adder adds two 1-bit numbers and a carry input. The full adder, like the half adder produces a SUM and CRY output as defined by the function definition contained in [Table 1-1](#). Note that in constructing this table we have implicitly used the isomorphism that allows us to represent binary values as logic values and *vice versa*.

Before going on, make sure you can justify all of the SUM and CRY function entries in Table 1-1. Refer to our earlier discussion about the [addition of binary numbers](#) or refer to your course textbook.

**Table 1-1. Full Adder Function Definition Table.**

C <sub>in</sub>	A	B	SUM	CRY
0	0	0	0	0
0	0	1	1	0
0	1	0	1	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

You could realize the SUM and CRY circuits from this table using canonical SOP or POS functions. If you have studied Karnaugh maps in the lecture portion of your course you will be able to derive a minimum SOP form for the full adder. Refer to your textbook or prove, using the Karnaugh map below (Figure 1-11), that the CRY function is given by:

$$\text{CRY} = AB + BC_{\text{in}} + C_{\text{in}}A$$

		CRY			
		AB	00	01	11
Cin	AB	00	0	1	0
	Cin	00	0	1	0

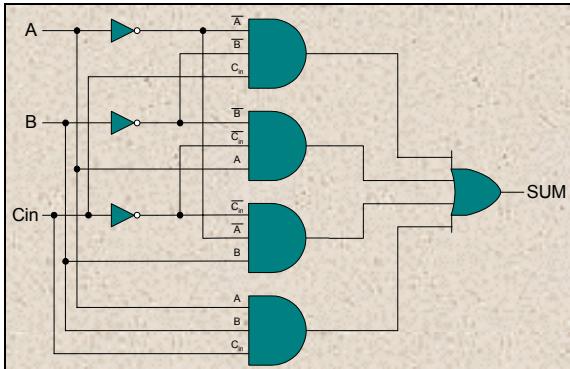
**Figure 1-11.** Karnaugh map for CRY function.

		SUM			
		AB	00	01	11
Cin	AB	00	0	1	0
	Cin	00	0	1	0

**Figure 1-12.** Karnaugh map for SUM function.

If you look in [Figure 1-12](#) at the Karnaugh map attempt to obtain a minimum SOP expression for the SUM function, you see that we obtain no reduction beyond the canonical SOP form:

$$\text{SUM} = \overline{A} \overline{B} C_{\text{in}} + \overline{A} B \overline{C}_{\text{in}} + A B C_{\text{in}} + A \overline{B} \overline{C}_{\text{in}}$$

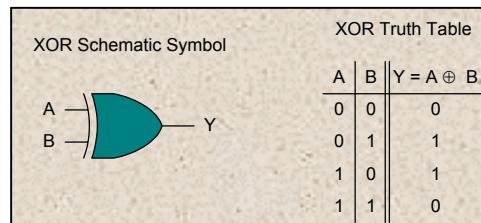


**Figure 1-13.** SOP implementation of SUM function.

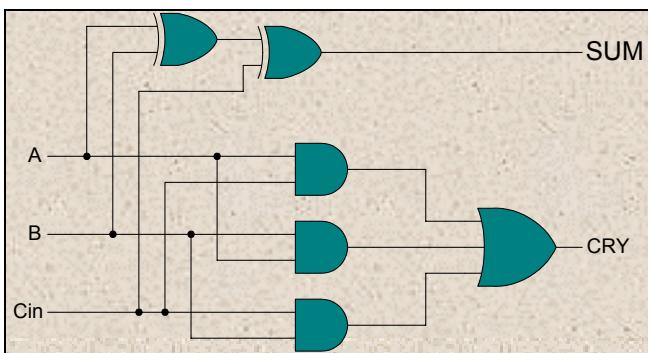
Such a realization for the SUM function would require four 3-input AND gates and one 4-input OR gate, and three inverters as shown [Figure 1-13](#). Fortunately there is a more economical method for implementing the SUM function. Refer again to [Table 1-1](#) and make the following observation: the SUM function is 1 **if and only if** the row address (i.e., input variable values) contains an odd number of 1's. This is precisely the function performed by an exclusive OR (XOR) gate (shown in [Figure 1-14](#)), provided we supply the XOR inputs with variables that represent each bit of the address; hence an economical way to implement the SUM function is to use the XOR operator, i.e.,

$$\text{SUM} = A \oplus B \oplus C_{\text{in}}$$

The SUM function could be built using either one 3-input XOR gate or two 2-input XOR gates from the 7486 quad XOR gate IC. This latter implementation is shown in [Figure 1-15](#).



**Figure 1-14.** XOR schematic symbol and truth table.



**Figure 1-15.** SOP implementation of 1-bit full adder.

### Task 1-5: Build, Debug and Test a TTL 1-Bit Full Adder

Build, debug and test a 1-bit full-adder circuit using the schematic shown in Figure 1-15. Provide a complete description of your circuit schematic as part of your lab. This means identifying which pins of which TTL IC's are used to implement each of the Boolean gates in your design. Select the measurements you make so that they can be used in your report to convince the reader that your circuit functions as a 1-bit full adder.

# HARDWARE LAB 1: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 1-1: Build the 1-Bit Half-Adder		
Task 1-2: Test the SUM & CRY of the 1-Bit Half-Adder with LED		
Task 1-3: Debugging		
Task 1-4: Completely Test your Half Adder Circuit Using a Logic Probe		
Task 1-5: Build, Debug and Test a TTL 1-Bit Full Adder		
What I Learned		
<b>Lab Data Sheets</b>		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

**Caveat emptor:** Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.

- Title Page: Include **Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted** in all reports.
- Include lab data sheets for all labs. Use lab data sheets to document the lab as you perform it. It saves a lot of time when writing the report.
- Include **Truth Tables or Function Tables** as required to explain how circuits work and as proof of circuit tests.
- **Label** all figures or circuits. (E.g. Figure 1. Schematic of . . . , Figure 2. . . , etc.)
- Refer in the text of your report to all circuits or figures that you include in the body of your report.
- Use LogicWorks™ to draw your circuit schematics and cut/paste them into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

Put an ‘X’ in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use ‘5’ to indicate that you ‘strongly agree’, ‘3’ to indicate that you are ‘neutral’, and ‘1’ to indicate that you ‘strongly disagree’. Use ‘NA’, ‘Not Applicable’, when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your responses will not be graded. They are for your instructor’s information only.

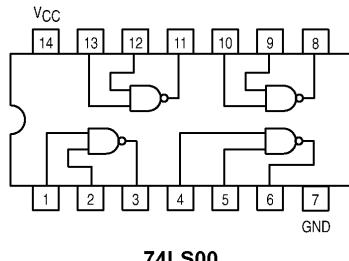
**Table \_\_: Self-Assessment of Outcomes for Hardware Lab 1: Debugging a Half and Full Adder.**

After completing the assigned tasks and report, I am able to:	5	4	3	2	1	NA
Describe the truth tables that characterize the addition of two single bit numbers.						
Write the Boolean algebraic expressions that characterize the sum and carry functions for the half adder using both the product-of-sums and sum-of-products canonical forms.						
Realize the sum and carry functions using TTL hardware.						
Debug TTL circuits using an LED and Logic Probe as testing instruments.						
Write the Boolean algebraic expressions that characterize the sum and carry functions for the full adder.						
Build and debug a full adder.						

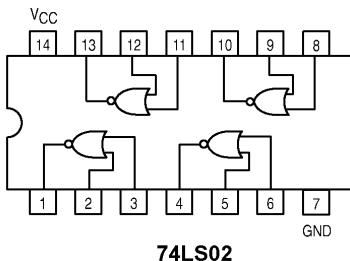
Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

## PIN-OUT DIAGRAMS

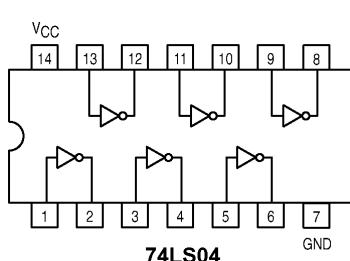
The pin-outs of several TTL devices are given below.



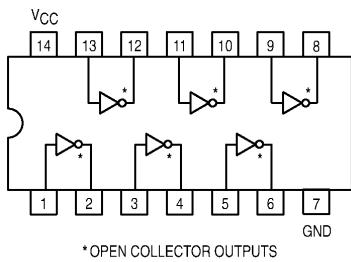
**74LS00**



**74LS02**

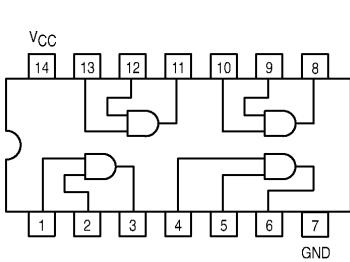


**74LS04**

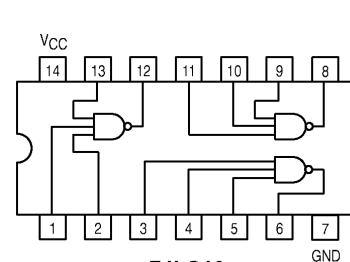


\* OPEN COLLECTOR OUTPUTS

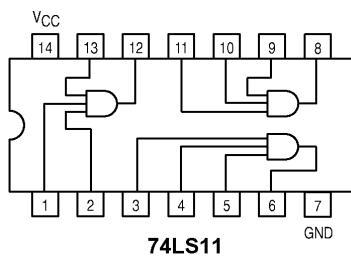
**74LS05**



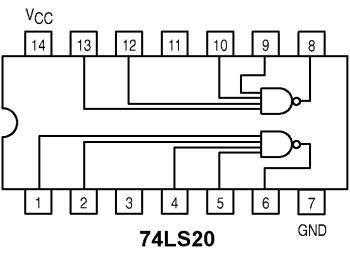
**74LS08**



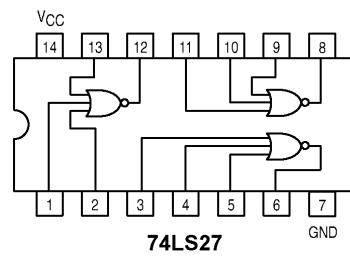
**74LS10**



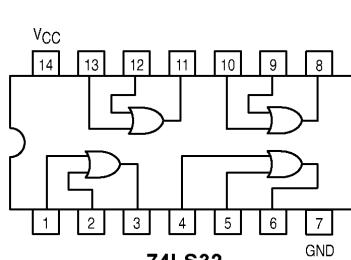
**74LS11**



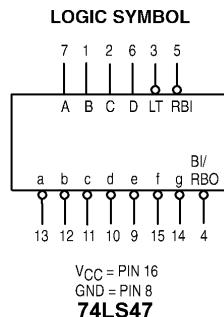
**74LS20**



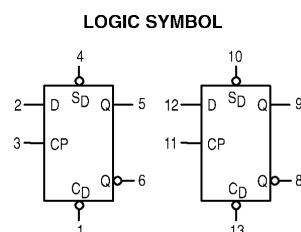
**74LS27**



**74LS32**



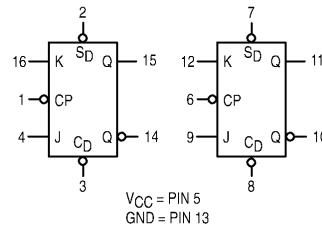
**74LS47**



**74LS74**

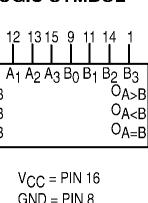
(Copyright of Motorola, Used with permission)

**LOGIC SYMBOL**

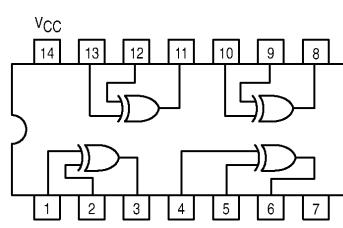


**74LS76**

**LOGIC SYMBOL**

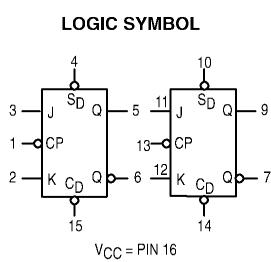


**74LS85**

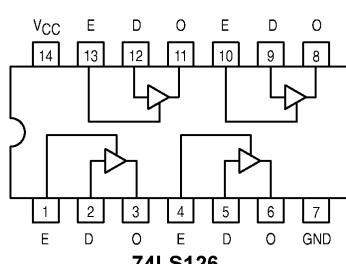


**74LS86**

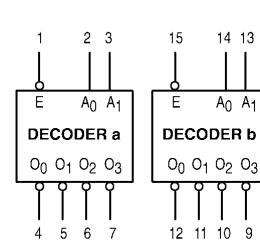
**LOGIC SYMBOL**



**74LS112**

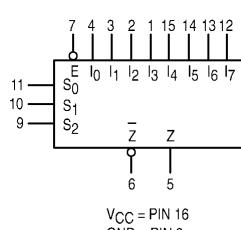


**74LS126**



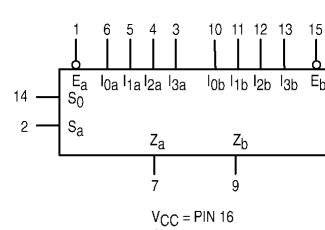
**74LS139**

**LOGIC SYMBOL**



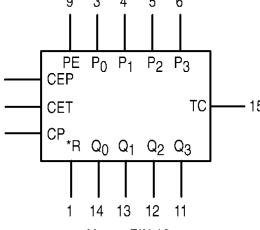
**74LS151**

**LOGIC SYMBOL**



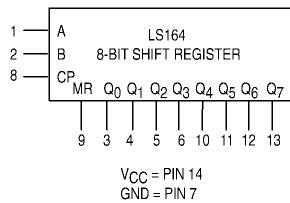
**74LS153**

**LOGIC SYMBOL**



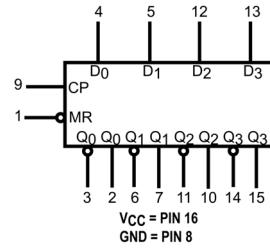
**74LS163**

**LOGIC SYMBOL**



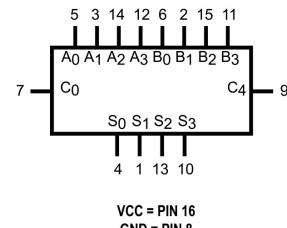
**74LS164**

**LOGIC SYMBOL**



**74LS175**

**LOGIC SYMBOL**



**74LS283**

(Copyright of Motorola, Used with permission)

# HARDWARE LAB 2: TTL CHARACTERISTICS, THREE-STATE BUFFERS, OPEN-COLLECTOR BUFFERS

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Use a prototype board, voltmeter, and logic probe.
- Describe the truth tables that characterize the AND, OR, XOR and NOT gates.

**Equipment:** Digital Trainer Board, Voltmeter, Logic Probe

**Circuit Components:** You will need the following circuits to complete the tasks in this lab:

- (2) 7404 (Hex Inverters)
- (1) 7405 (Hex Open-Collector Inverter Buffers)
- (1) 74LS126(Three-State Buffers) (Not 74HC126)
- (1) 1000 Ohm Resistor
- (1) 100 Ohm Resistor (Task 2-7 and Task 2-8 only.)
- (1) 5.6 V. Zener Diode (Task 2-7 and Task 2-8 only.)

**Objective:** In this laboratory exercise, you will learn to identify the electrical and thermal effects of questionable wiring practices. You will also learn how to electrically connect three-state and open-collector buffer circuits to drive a common communication bus.

**Outcomes:** When you have completed this laboratory exercise you will be able to:

- Make and interpret electrical voltage measurements.
- Detect and recognize the thermal and electrical effects of driving a common point with two conflicting output signals.
- Describe the different ways in which three-state and open-collector buffer circuits must be controlled to drive a common bus.
- Electrically connect three-state and open-collector buffer circuits to drive a common communication bus.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab it is recommended that you use a task-oriented format when organizing your lab report. (If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Task-Oriented Report Writing Guidelines.](#))

---

## Introduction

In Hardware Lab 1, we built and debugged a full-adder and a half-adder circuit. In Hardware Lab 2, we will look further at debugging techniques and we will investigate the three-state and common-collector buffers. In this lab, we will use another debugging tool, the voltmeter. (Refer to the [Hardware Lab 0: Using a Prototype Board, Logic Probe & Voltmeter](#) for a description of how to use a voltmeter.) In TTL chips, as in most types of digital logic, information is passed in terms of voltage. A value of +5 V. is used to represent a digital 1 (logical true) condition and a digital 0 (logical false) is represented by 0 V. (or ground.)

Because of design and manufacturing issues, it is impossible to guarantee that the output voltages of an IC or input voltages feeding an IC will be precisely these values; hence a designer designs IC's so that a certain range of input values will be interpreted as a digital 1 (or 0) and a certain range of output values will be produced by inputs within this range. It is important in the design that the guaranteed output values are a subset of the allowable input values; that way, slight inaccuracies in the output values will still produce an input value that falls within the acceptable range. To be more precise, let's define the following variables:

- $V_{OLmax}$  is the maximum output voltage value that a circuit with a low logic-level will produce.
- $V_{ILmax}$  is the maximum input voltage value that a circuit will interpret as a low-logic level.
- $V_{OHmin}$  is the minimum output voltage value that a circuit with a high-logic level will produce.
- $V_{IHmin}$  is the minimum input voltage value that a circuit will interpret as a high-logic level.

The relationship between these quantities is shown in Figure 2-1. Notice that  $V_{OLmax} < V_{ILmax}$  and  $V_{OHmin} > V_{IHmin}$ . Convince yourself that if the outputs of digital logic gates are to feed inputs of other gates and if the outputs of these gates are to feed other inputs, and so on, that this relationship must be the case. The strict inequality relationship (e.g. strictly ‘greater than’, rather than ‘greater than or equal to’) is necessary to provide a margin of safety so that even with the vagaries of the manufacturing process and the addition of electrical-noise voltage to signals (caused by stray electrical radiation), the digital logic circuits will work reliably.

The experiments in this lab manual are designed to be performed using TTL logic; however, many educational laboratories rely on donations from semiconductor manufacturers to stock their IC bins; hence you may be completing some of these experiments with IC's made with other technologies, specifically CMOS (Complementary-Metal-Oxide Semiconductor) IC's. With CMOS IC's the same voltage relationships hold, but the ranges are different as shown in Figure 2-2.

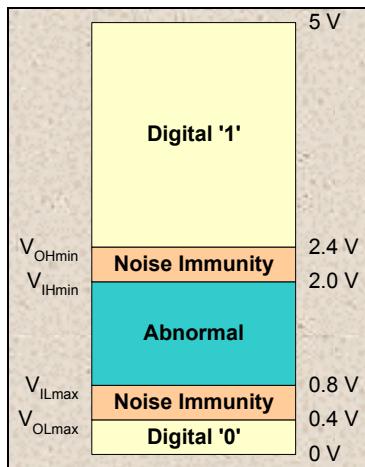


Figure 2-1. Logic levels in TTL IC's.

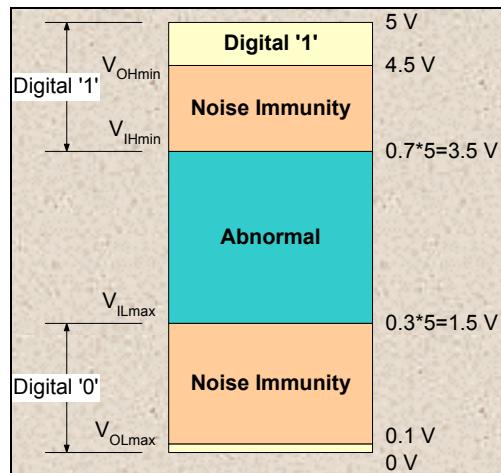


Figure 2-2. Logic levels in CMOS IC's.

In the following tasks, we will intentionally be connecting IC's in unusual, and sometimes undesirable, ways. The objective of this perverse approach is to allow you to measure the voltage values produced by TTL circuits when they are used with questionable wiring practices. Some practices that we have regarded heretofore as heresy will give surprisingly acceptable, even if unreliable, results. Some wiring practices will yield voltage measurements that violate the  $V_{OLmax}$ ,  $V_{ILmax}$ ,  $V_{OHmin}$ , or  $V_{IHmin}$  criteria. Once you learn to associate the wiring practices with the violations that they cause, you will be able to make the intuitive leap and associate the voltage violations you measure in the lab with the questionable wiring practices that cause them. Although this reverse association is not always correct (causally related), it may identify one possible cause of the voltage violation problems you encounter in the lab and provides at least a starting point for identifying your problem.

### Task 2-1: Effect of Missing Inputs to TTL Gates

Let's investigate the effects of having missing inputs on TTL gates. Specifically let's look at the effects of missing inputs on the inverters contained on the 7404 IC (hex inverters.) Insert a TTL 7404 IC (quad inverters) into your breadboard and connect its  $V_{CC}$  and GND pins. (Check with your lab laboratory assistant if you have any questions about which technology family your IC belongs to<sup>1</sup>.) Connect the input

<sup>1</sup> 7400 series gates have the nomenclature, 74XXXNNN, where 'XXX' is the family mnemonic such as S, LS, AS, F, ALS, (for TTL IC's) or HC, HCT, AC, or ACT (for CMOS IC's) and 'NNN' identifies the IC as a NAND, NOR, AND etc. gate; hence a 74LS04 is a TTL inverter and a 74HC04 is a CMOS inverter.

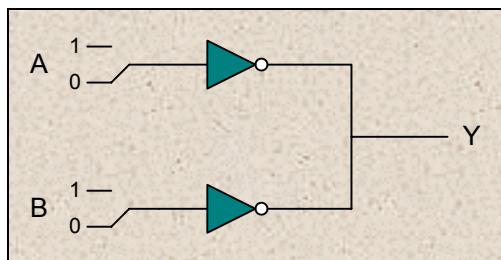
of one of the NOT gates to a data switch. Record the input and output voltages of the NOT gate for inputs of digital 0 and 1. Next disconnect the switch from the NOT gate. Now that there is no input to the NOT gate, record the voltages at the input and output of the NOT gate. What happens to a TTL gate when you forget to connect one of its inputs? In answering this question refer, as needed, to the standard TTL voltage levels discussed in the pre-lab discussion. Don't count on this answer to save you a little wiring. If you use a gate, you should always connect all of its inputs. It is important to know what a gate does when it has no input only so that you can more easily find this problem when debugging.

### Task 2-2: Observe How Hot the Chips Get

Be careful in the next procedure. No chips in this lab should get very hot, but malfunctions or wrong wiring can cause problems. Always take precautions that you do not burn yourself when feeling chips. Now feel the top of the 7404 to see how hot it is. You will probably find that the back of your finger is more sensitive to heat than your finger's tip. Jot down in your lab notebook a subjective assessment of how hot the chip is for later reference. If wires prevent you from feeling the top of the IC with your finger, move the wires. (Remember it is a preferable practice not to run wires directly over the top of a chip because you may need to take the chip out later if you find out it is defective.)

### Task 2-3: Gates with Common Outputs

Next, connect the circuit shown in [Figure 2-3](#) on a *second* 7404 IC on the same breadboard.



**Figure 2-3.** Inverter driving a common output.

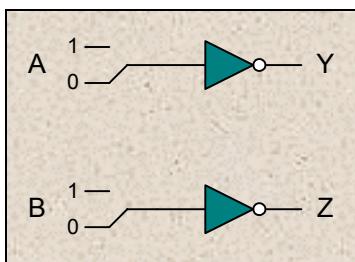
This circuit will allow you to see what happens when two outputs are connected together. For all four possible input combinations, record the output voltage in a table. Comment on the differences between these measurements and the ones you recorded in Task 2-1. Which input combinations caused the voltage readings to appear normal? Which input combinations gave voltage readings that were abnormal? What output voltage values will make you suspicious in the future that two outputs have been connected together? In digital computer design it is often desirable to have two different sources of information made

available to a single input. Can the connection of [Figure 2-3](#) be used to control this data sharing so that no data conflict occurs<sup>2</sup>? Explain your answer in your lab report.

Use a logic probe to determine the logic level of the output for each of the four input combinations. Add this information to your table. Can you identify the ‘two outputs connected together’ error as reliably with a logic probe as you can with a voltmeter? When you are done with your tests, (taking appropriate precautions) feel the top of the 7404 while A is driven with a 1 and B is driven with a 0. Does the IC get abnormally hot when connecting two conflicting outputs together?

### Task 2-4: Missing Ground and V<sub>CC</sub> Connections

Next, we will measure the effect on the output signals of removing the ground and V<sub>CC</sub> connections, one at a time. First, set up the circuit shown in [Figure 2-4](#), using two NOT gates on the same IC. Wire the IC so that the V<sub>CC</sub> pin is connected to +5 V, but leave the GND pin unconnected. Since we need a ground return path for the power supply to supply electrical power to the IC, the IC will get no power from the power supply; hence it seems reasonable to expect that the IC will not function properly. Let's see if this is true.



**Figure 2-4.** Missing ground connections.

Measure the voltage at Y for all four combinations of inputs A and B and enter them in a table in your lab notebook. Also determine the logic level at Y for all four input combinations using the logic probe. If switch B is set to 0, does inverter A appear to work? (In your discussion, mention the differences in the information provided by the voltmeter and the logic probe.) Can you detect this problem better with the voltmeter than with the logic probe? Now reconnect GND, and disconnect V<sub>CC</sub> from the 7404. Repeat the measurements of Y using a voltmeter and logic probe for all four input combinations of A and B.

The results for the above test may surprise you a little. It turns out that chips can appear to work when they are missing the ground connection. This happens because of the way TTL chips are built internally. If you connect any input of the chip to ground, the chip can use that as a ground return path. This problem can be difficult to detect, because the chip is almost working correctly. The problem may appear intermittently – and by now you know how hard intermittent problems are to debug. Luckily, when the problem manifests itself it can be detected with a voltmeter.

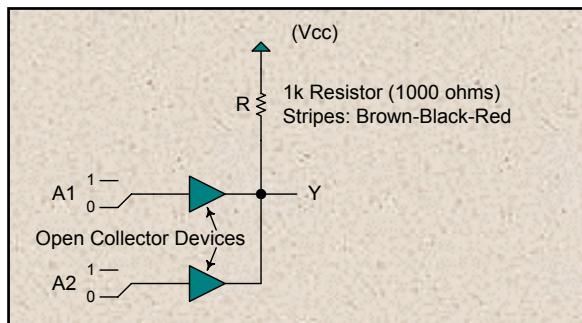
<sup>2</sup> A data conflict occurs when two different data sources supply signals to a common point simultaneously.

## Buffers

In [Task 2-3](#), we experimented with connecting the outputs of two inverters together and found that this created a data conflict under certain input conditions. There are many instances in computer design where we would like to have two (and often more) sources of information drive different data values onto a single data line at different times. As you progress in your course, you will learn that one way to route several sources of information to a common point without experiencing data conflicts is to use a multiplexer. A multiplexer is a digital circuit that allows multiple sources of data to ‘take turns’ so that only one source of data drives a single point at any one time. Data conflicts can also be eliminated by using circuits known as buffers. In the next tasks, we will experiment with three-state (a.k.a. tri-state<sup>3</sup>) and common-collector buffers.

## Common-Collector Buffers

A TTL circuit is often designed so that the collector terminal of the output transistor is connected to the  $V_{CC}$  pin via another transistor and a resistor. This configuration is called a ‘totem-pole output’ because the output schematic shows these circuit elements ‘stacked,’ one on top of the other. When the output connections of two such gates are wired together while one is being driven ‘high’ and the other driven low, the output current is enormous – by TTL standards. The designer can eliminate this large current under data-conflict conditions by changing the design slightly. In the new design, the top transistor is replaced with a resistor connected from the +5 V. to the collector terminal of the remaining output transistor (a technique used in RTL<sup>4</sup> designs.) This resistor must be sized so that the value of the output voltage is correct for various input conditions. The size of the resistor is determined by the number of gates whose outputs will be wired together. Because the manufacturer cannot know how many gates you intend to wire together, the gate is manufactured with the collector terminal left open circuited – hence the name open-collector outputs – and this collector is made available to the user through one of the IC pins.



**Figure 2-5.** Common-collector buffer.

When a single resistor (known as a ‘pull-up resistor’<sup>5</sup>) is used to connect the open-collector pins of several open-collector devices to +5 V. (as shown in **Figure 2-5**), we get a common-collector buffer. The output of the common-collector buffer has the following behavior: the output, Y, of the circuit of **Figure 2-5** is a 0 if any of the inputs is 0. This is the precisely the characteristic of an AND gate. Common-

<sup>3</sup> Tri-state is a trademark of National Semiconductor Corporation

<sup>4</sup> RTL – Resistor Transistor Logic – is an obsolete technology used in the early designs of bipolar-transistor-logic gates.

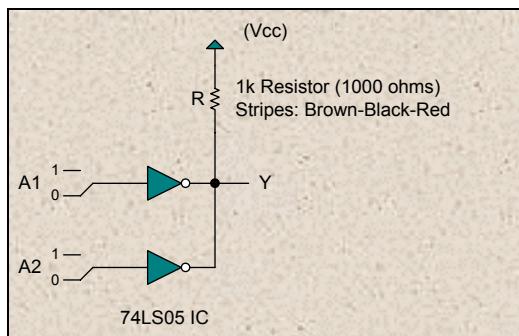
<sup>5</sup> This resistor must be sized according to the number of gate outputs (transistor collectors) that are to be wired to a common point. For this experiment, a 1000-Ohm resistor will be appropriate.

collector output circuits are consequently called ‘wired-ANDs’ because the AND function is produced by simply ‘wiring’ the outputs together. (Note that the absence of the circles at the outputs of the open-collector-buffer circuit symbol means that these are non-inverting buffers.) To function properly, the inputs of all open-collector buffers that are to be inactive must be driven to a 1 so that the active buffer can control the output to be either a 1 or 0. Sometimes, creating circuits to drive the inputs to high values when unused can be inconvenient. In such instances, a three-state buffer is used.

The open-collector buffers you will experiment with differ from those discussed above because they are inverting buffers. When active, the output of these buffers is the complement of the input.

### Task 2-5: Build and Test a Common-Collector Buffer Circuit

Build the circuit shown in [Figure 2-6](#), using the 7405 common-collector buffer. Connect the common output to an LED. (When you get the pull-up resistor from the parts bin, be sure to check its value by looking for the brown-black-red stripe pattern. Do not trust the label on the bin containing the resistor. IC's are often misplaced as well, so check the part number on the IC before you use it.)



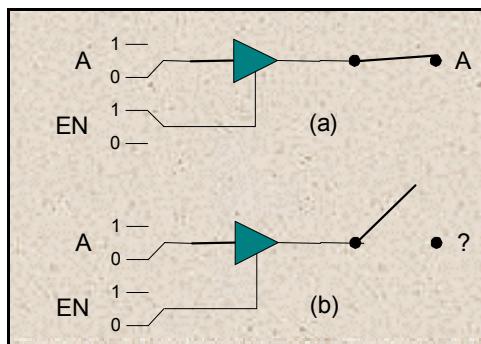
**Figure 2-6.** Schematic for common-collector inverter buffer (74LS05 chip) experiment.

Record the voltage and logic level (i.e. 0 or 1) of the output for all input combinations. What wired logic does the circuit perform (AND, OR, etc.)? (Hint: Remember that these are *inverting* buffers and their outputs will behave differently than discussed in the [Common-Collector Buffers](#) section.) Does the chip give the proper output voltages even though its outputs are connected together? Does the chip heat up from connecting its outputs together?

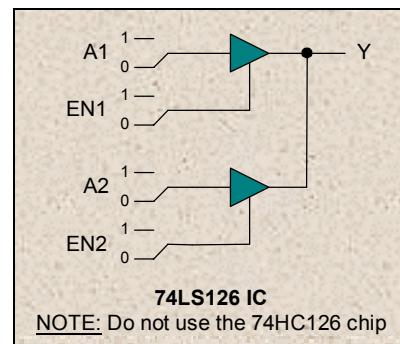
### Three-state Buffers

The three-state buffer has three output states, as its name implies. When the enable input of the three-state buffer, EN, is active, the output is the same as the input (as shown in Figure 2-7(a)): either a 0 or 1. If EN is inactive, the device enters its third state, a high impedance state. In this state, the voltage measured at the output is not related to the input value A, as shown schematically in Figure 2-7(b). In the high-impedance state, the output voltage is controlled by the downstream devices not by the buffer’s input signal. This

means that three-state buffers can have their outputs connected together (as shown in [Figure 2-8](#)) provided all but one are in the high-impedance state. Using this control scheme, only the active buffer controls the value measured at the output; however, if the outputs do not ‘take turns’ properly, (i.e., if more than one of the three-state buffers is active) the potential exists for a data conflict. Therefore, it is necessary when connecting together the outputs of several three-state devices to be sure that, at most, one of the outputs is active. By comparing Figure 2-6 and Figure 2-8 you may notice that one advantage the three-state buffer has over the open-collector buffer is that the three-state buffer requires no external pull-up resistor.



**Figure 2-7.** Effect of EN input on input/output of a three state buffer.



**Figure 2-8.** Schematic for three-state buffer experiment.

The three-state buffers shown connected together in Figure 2-8 have two inputs each. The enable (EN) input determines whether the device is in the active state or in the inactive high-impedance state. If  $EN_1 = 1$  while  $EN_2 = 0$ , the output is the same as the A input. If both  $EN_1$  and  $EN_2$  equal 0, both devices are in the high-impedance state and we can make no statement about the value of the output voltage, Y.

### Task 2-6: Build and Test a Three-State Buffer Circuit

Build the TTL three-state buffer circuit shown in [Figure 2-8](#). Use only a TTL IC, such as the 74LS126 chip. **DO NOT** use a CMOS IC, such as the 74HC126 chip.

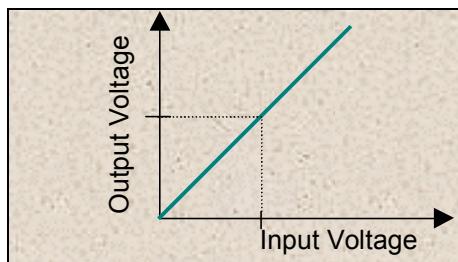
Before making your first measurement on this circuit, use the voltmeter to measure the voltage of the tip of the logic probe<sup>6</sup>. Record this value because it may help you explain errant measurements later. Next, using a voltmeter, measure the output voltages of the circuit for all combinations of inputs. (Remember to record all measured values in your lab notebook.) After you have finished voltage measurements for all input combinations, measure the logic levels for all input combinations using a logic probe set to **TTL**. It is **very important** to record the voltages **first** for **all** input combinations using the voltmeter and **then** the logic

<sup>6</sup> Remember to connect the logic probe properly to the power supply and set the logic probe switch(es) so that the probe responds to TTL, not CMOS, values.

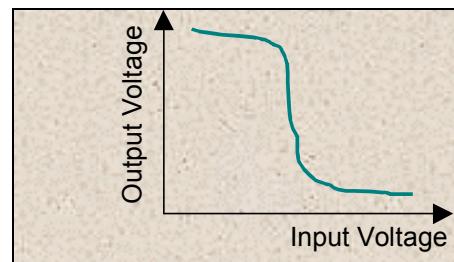
values using a logic probe for the same input combinations<sup>7</sup>. Note that in TTL designs an output of 0 is guaranteed to be between 0 to 0.4 V. and a 1 between 2.4 to 5 V. When the three-state buffer is in the high-impedance state, some voltmeters will register no reading; that is, their LCD displays will remain blank. Other voltmeters will retain their last reading - perhaps changing slightly. Be aware of this possibility when you record your measurements and observations in a table in your lab notebook. Using your table, answer the following questions: If the enable, EN, of one of the buffers is 0, does changing the input A of that gate change the output? If the enables of both gates are 0, what output do you get? If both EN values are 1, what outputs do you get for all combinations of A1 and A2? (Be careful you do not burn yourself when performing the experiment that answers the following question.) Does the IC heat up if both enables are 1 while A1 is 0 and A2 is 1?

### Prologue to [Task 2-7](#) and [Task 2-8](#) (Advanced Tasks)

In the exercises you have completed thus far, the logic gates have largely functioned as ideal devices. In the foregoing tasks we supplied the inputs with voltages that were within the TTL 1 and 0 design ranges; consequently our circuits produced output values that were within the appropriate TTL design ranges. The next tasks are designed to allow you to measure quantities that show that a logic gate is a highly nonlinear device.



**Figure 2-9.** Output v. input characteristics of a linear device.



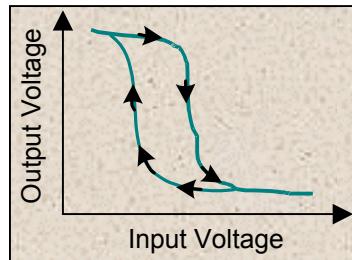
**Figure 2-10.** Output v. input characteristic of a nonlinear device.

In a linear device, if the input voltage is doubled, the output voltage changes by a factor of 2. A linear device always has the output v. input characteristics shown in [Figure 2-9](#). In a nonlinear device, if the input voltage is doubled, the output voltage may change by a factor much larger or smaller than 2. The nonlinear output v. input characteristic, shown in [Figure 2-10](#) for one nonlinear device, shows that the amount the

<sup>7</sup> The tips of some logic probes are elevated to a voltage of 2.5 V. (with respect to ground) when not being used to sample a value. When the logic probe is touched to the output of a three-state device in a high-impedance state, it will deliver sufficient electrical charge to the output to boost the output voltage to +2.5 V. Since this charge cannot drain off with the device in a high-impedance state the voltmeter will subsequently measure the value of the charge on the output terminal i.e., +2.5 V., rather than the natural high-impedance-state output voltage.

output changes (for a given input change) will depend in a complex way on the input voltage magnitude. The characteristics of your inverter circuit will look similar to that of [Figure 2-10](#).

Some inverter circuits have more complex output v. input characteristics than those shown in [Figure 2-10](#). Specifically Schottky TTL, (those IC's with 'S' in their name e.g., 74LS04), have output values that depend on their input and on the history of their input values. For example, the output v. input characteristic of the 74LS04 inverter has the form shown in Figure 2-11. The arrows indicate that the state transition, between digital values interpreted as 1 and 0, occurs for a greater input voltage when the input is low and increasing, than when the input voltage is high and decreasing. This effect is known a hysteresis. If you use a Schottky inverter in this experiment, you will have to account for the hysteresis effect in your results.



**Figure 2-11.** Output v. input characteristic of a Schottky inverter

In the next two tasks, you will measure the input-output characteristics of an inverter. You will show that the output v. input characteristics of an inverter is highly nonlinear. You will estimate then measure the realized minimum voltage that a specific gate will interpret as a high value at its input,  $V_{IH\min}$ , and the realized maximum voltage that a specific gate will interpret as a low value at its input,  $V_{IL\max}$ . The learning outcomes of the next two tasks are:

**Outcomes:** When you have completed the next two tasks you will be able to:

- Describe the approximate input/output characteristics of an inverter.
- Recognize that the inverter input/output curve is nonlinear.
- Recite a measured value of  $V_{IH\min}$  and  $V_{IL\max}$  for a typical TTL inverter.

You will need the following equipment to complete the following two tasks.

### Equipment:

A + 5 V. voltage supply, a variable voltage supply whose output can be varied between 0 V. and +5 V., at least one and preferably two voltmeters.

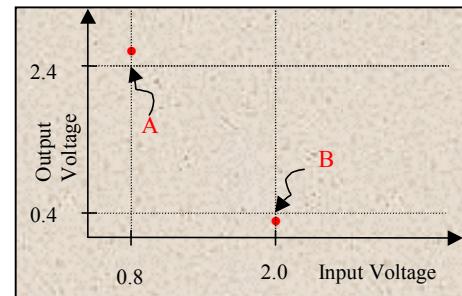
### Circuits Components:

- (1) 7404 (Hex Inverters)
  - 100 Ohm Resistor
  - 5.6 Volt Zener Diode
-

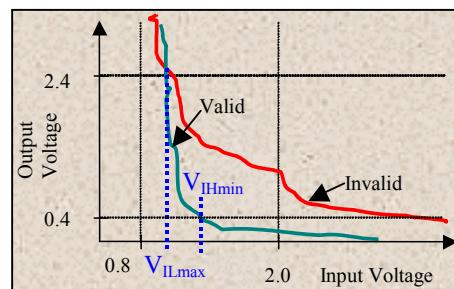
## Input/Output Characteristics of TTL Devices

In the introduction to these laboratory exercises, we observed that a TTL device will interpret a range of input voltages as a logic 0 and produce an output voltage that is within a specified range for the appropriate logic level. Specifically, for a TTL inverter, if the input voltage is at or below 0.8 V, then the output is guaranteed to be above 2.4 V. (Remember that an inverter inverts; hence a low input will produce a high output.) Similarly, if the inverter is supplied with an input value greater than or equal to 2.0 V., then the output is guaranteed to be below 0.4 V. Two data points consistent with this requirement are shown in the output v. input plot of [Figure 2-12](#).

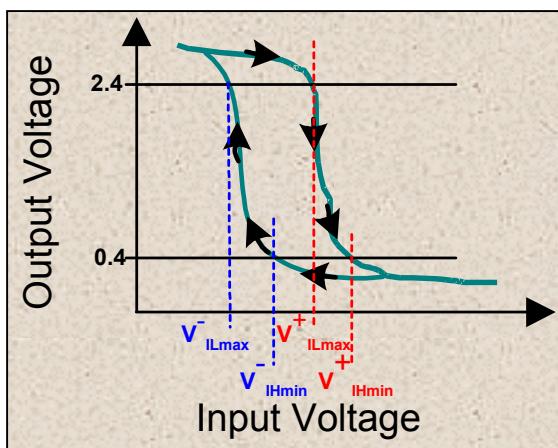
For a TTL input/output characteristic to be valid, the input/output curve must cross the 2.4 V. output line to the right of the 0.8 V. input value (point A of Figure 2-12) and it must cross the 0.4 V. output line to the left of the 2.0 V. input value (point B of Figure 2-12). Two input/output characteristic, one valid and one not valid, are shown in [Figure 2-13](#). The red curve of Figure 2-13 is invalid because it violates the criterion that the output for an inverter must be less than 0.4 V. for all input values greater than 2.0 V.



**Figure 2-12.** Two data points in the input/output characteristic of a TTL inverter.



**Figure 2-13.** Valid and Invalid TTL input/output characteristics for a TTL inverter.



**Figure 2-14.**  $V_{ILmax}$  and  $V_{IHmin}$  definitions for Schottky devices.

In [Task 2-7](#), you will be asked to construct a test circuit to measure the input/output characteristics of a TTL inverter. From a plot of that data, you will be asked to estimate the value  $V_{IHmin}$  (the realized minimum input voltage your inverter interprets as a logic 1) and  $V_{ILmax}$  (the realized maximum voltage that your inverter interprets as a low value.) Figure 2-13 shows that  $V_{ILmax}$  can be estimated as the input voltage that causes an output voltage of 2.4 V;  $V_{IHmin}$  can be estimated as the input voltage that causes an output voltage of 0.4 V.

If you are measuring a Schottky inverter, you will need to estimate two different values for  $V_{IHmin}$  and  $V_{ILmax}$ . In one test, you will start with a low value of input voltage and increase it to measure  $V^+_{ILmax}$  and  $V^+_{IHmin}$  as indicated in Figure 2-14. In a second test,

you will start with a high value of input voltage and decrease it to measure  $V_{IL,max}$  and  $V_{IH,min}$  as indicated in Figure 2-14.

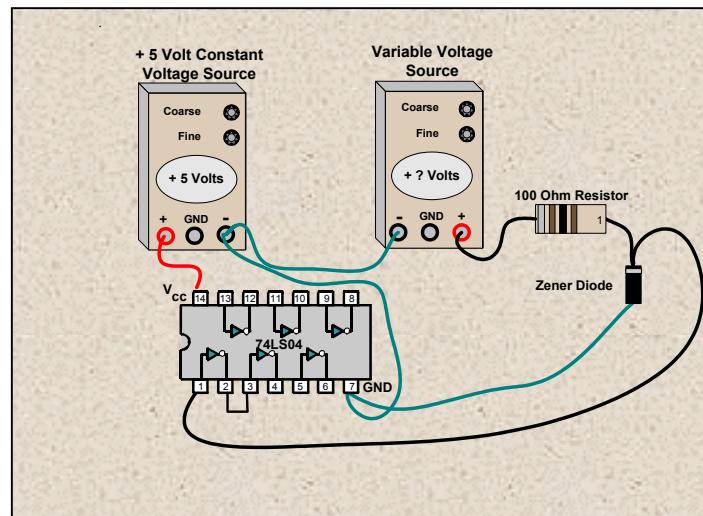
**WARNING:** It is easy to burn out IC's or elements of your prototype board if you connect external power supplies incorrectly to your prototype board. Follow the directions given in [Task 2-7](#) and [Task 2-8](#) carefully! Stop and ask a lab laboratory assistant for help if you are unsure about any of the directions.

### Task 2-7: Measuring the Input/Output Characteristic of a TTL Inverter.

Let's begin by estimating the input-output characteristic for a TTL inverter.

To make this measurement, we will need two voltage sources: one that produces a constant value of +5 V. and one that can easily be varied between 0 V. and +5 V<sup>8</sup>. Using these two sources, a 7404 hex inverter IC, a 100-Ohm resistor, and a 5.6 V. Zener diode, build the circuit shown in [Figure 2-15](#). The Zener diode in this circuit functions to limit the voltage supplied to the input of the inverter to 5.6 V. – even when the output of the variable voltage source exceeds 5.6 V. - and prevents

the variable voltage source from burning out the inverter IC by delivering a large voltage. The output of the inverter whose characteristics we'll measure is connected to another inverter in order to simulate the effects of load on the inverter characteristics. Once you have constructed this circuit, check that it is working correctly by measuring the output voltage (at pin 2 of the IC) as you set the variable input voltage (at pin 1 of the IC) to 0 then +5 V.



**Figure 2-15.** Schematic diagram of circuit used for measuring input v. output characteristic of TTL inverter.

**ADVISORY:** Measure the inverter-input voltage at the pin of the IC, not at the output of the voltage source. The 100-Ohm resistor and the Zener diode will always force the voltage at the input pin to be lower than that produced at the terminals of the variable voltage source.

<sup>8</sup> If you are using an Elenco or similar digital trainer board you can use the +5 V. supply available through the trainer board as the constant voltage source. You will need one additional voltage source that can be varied; otherwise you will need two voltage sources – one of which can be varied. See your course instructor or a lab TA if you need help in finding a variable voltage supply.

To make the measurements associated with this task use the experimental set up shown in Figure 2-16. If resources are available in your laboratory, use two voltmeters. Connect the voltmeters so that one continuously monitors the voltage input to the inverter while the other continuously monitors the output of the inverter as shown in Figure 2-16. (If you cannot get two voltmeters, simply monitor the output voltage until you get the desired value – to be described - then measure the input causing this value.)

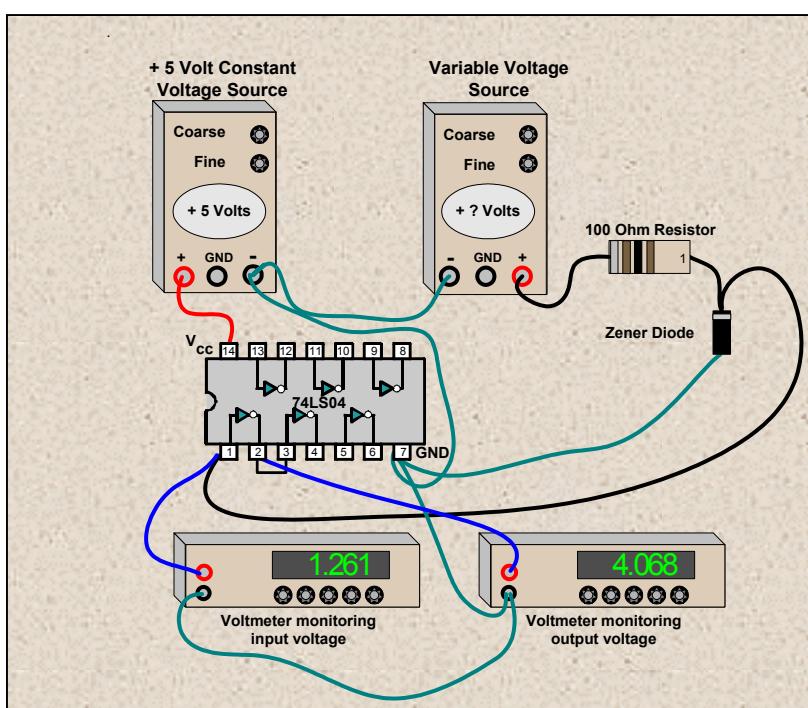


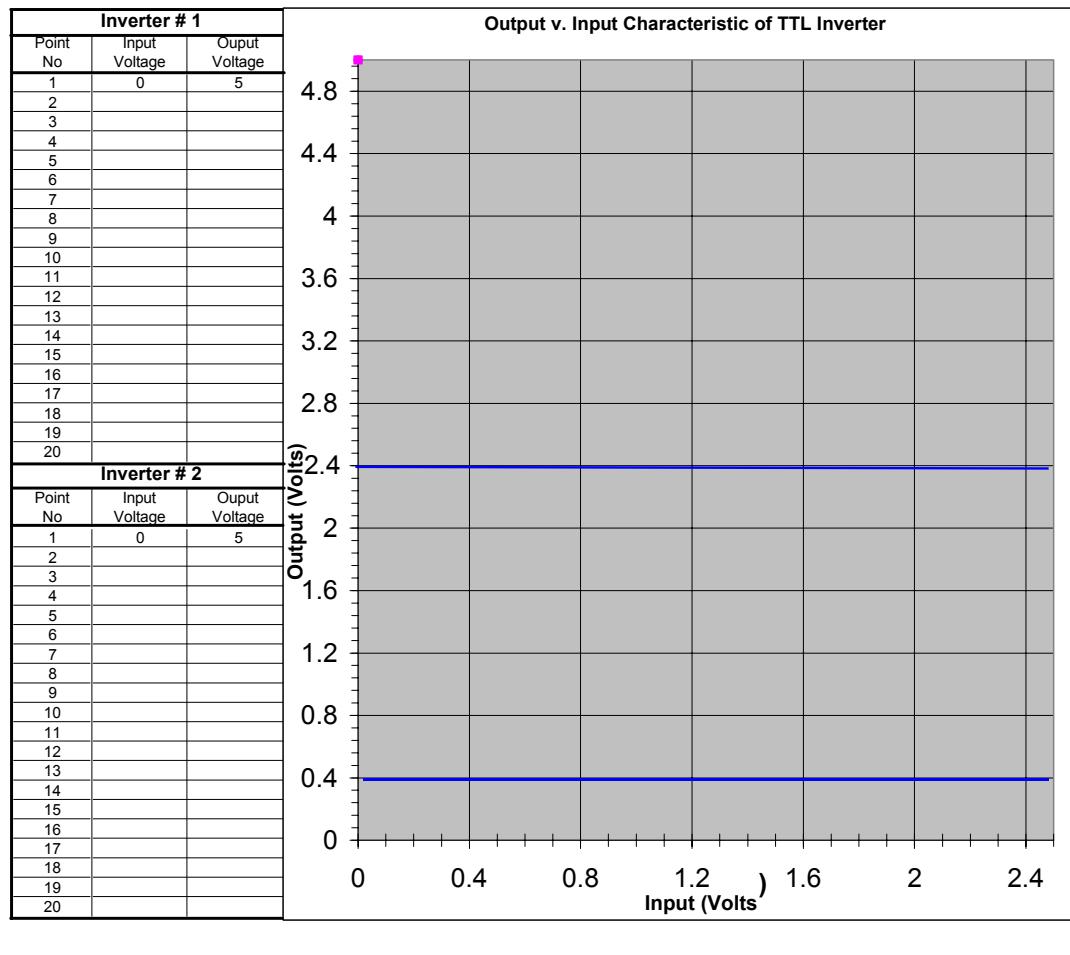
Figure 2-16. Measurement schematic for Task 7 and 8.

To estimate the input-output characteristics, SLOWLY increase and measure the inverter input voltage (starting at 0 V. and ending at 5.0 V.) while measuring the inverter output voltage. (If you measure voltages greater than 6.0 V. on the pins of any IC's get a laboratory assistant to check your circuit.) Record the measurements in your laboratory notebook and plot the inverter output voltage versus the inverter input voltage. You may use the worksheet given in Figure 2-17 as a guide.

**REMEMBER:** Because we are using an inverter for this experiment, to get the output voltage to increase you must decrease the input voltage.

While you are varying the input voltage, you will reach a point where the output voltage changes precipitously for a small change in input voltage. This extreme sensitivity of the output voltage to small perturbations in the input voltage is a desirable operating characteristic of a digital gate. Because of this sensitivity, duplicating input/output pair measurements reliably near this point is difficult – for the same (or practically the same) input, you may find that the output may be different the second, third and fourth times you make the measurement. Don't worry about these problems. Simply use your best estimate near the transition point for entries into your table. Also, remember that for Schottky devices, you will find that the transition point will occur at one input voltage level when you make your measurements by starting at a low voltage and raising it ( $V_{IL,max}^+$  and  $V_{IH,min}^+$ ), and at another input voltage level when you start your measurements at a high input voltage and lower it ( $V_{IL,max}^-$  and  $V_{IH,min}^-$ ).

Once you have taken data for one inverter on your IC, repeat your measurements for a second inverter on your IC or on another inverter in another IC, then plot the output v. input characteristics of your two inverters.



**Figure 2-17.** Input v. output characteristic worksheet.

From the plots you create, estimate the inverter-input voltage that causes the output voltage of each inverter to be 2.4 V. (for a TTL gate). These are good estimates of  $V_{IL,max}$  for each of your inverters. Your estimates for each should agree to within a few tenths of a Volt. (Check with your lab laboratory assistant if your estimates vary widely.) From your plots, also estimate the inverter-input voltage that causes the output voltage of each inverter to be 0.4 V. (for a TTL gate). This is a good estimate of  $V_{IH,min}$  for your particular IC<sup>9</sup>.

<sup>9</sup> Don't worry if you can't accurately sketch your plot while in the lab. We only need a rough estimate of  $V_{IL,max}$  and  $V_{IH,min}$  as a starting point for the next task.

If you are using a Schottky device, you will need to use your plot to estimate  $V_{ILmax}^+$ ,  $V_{IHmin}^+$ ,  $V_{ILmax}^-$  and  $V_{IHmin}^-$ . In the next task, we'll see how accurate your estimates really are.

### Task 2-8: Measuring $V_{ILmax}$ and $V_{IHmin}$ for a TTL Inverter.

For this task use the same experimental set up you used in Task 1. Set the inverter input voltage a little below your estimate of  $V_{ILmax}^+$  (use the voltmeter to measure that you have accomplished this), then SLOWLY vary the input voltage, while you monitor the inverter output voltage. Most voltage sources have a fine adjustment control that will allow you to slowly vary the voltage sources output voltage. When the circuits output voltage reaches 2.4 V. the corresponding input voltage is  $V_{ILmax}^+$ . If you are using a non-Schottky device,  $V_{ILmax}^+ = V_{ILmax}^-$ , and you are finished with this measurement.

If you are using a Schottky device, to get the  $V_{ILmax}^-$  value associated with decreasing input voltage, (shown in [Figure 2-14](#)) you will need to set the inverter input voltage a little above your estimate of  $V_{ILmax}^+$  and then SLOWLY decrease the input voltage until the output is 2.4 V.)

Next, set the inverter input voltage a little below your estimate of  $V_{IHmin}^+$  (use the voltmeter to measure that you have accomplished this), then SLOWLY increase the input voltage, while you monitor the inverter output voltage. When the output voltage reaches 0.4 V. the corresponding input voltage is  $V_{IHmin}^+$  for your particular IC.

If you are using a non-Schottky device,  $V_{IHmin}^+ = V_{IHmin}^-$ , and you are finished with this measurement. If you are using a Schottky device, to get the  $V_{IHmin}^-$  value associated with decreasing input voltage, (shown in [Figure 2-14](#)) you will need to set the inverter input voltage a little above your estimate of  $V_{IHmin}^+$  and then SLOWLY decrease the input voltage until the output is 0.4 V.

Record all of your measurements in your lab data sheet and compare them with your estimates and the TTL design values: 2.0 V. for  $V_{IHmin}$  and  $V_{IHmin}^+$ , 0.8 V. for  $V_{ILmax}$  and  $V_{ILmax}^-$ .

Because of the sensitivity of the output voltage of digital circuits to the input voltage changes near  $V_{IHmin}$  and  $V_{ILmax}$ , it may be hard to duplicate these measurements. Don't fret if you are having trouble duplicating your measurements; simply make the measurements several times and record your best estimates of  $V_{ILmax}$  and  $V_{IHmin}$ . In your laboratory report, be sure to discuss what you learned by doing this experiment.

# HARDWARE LAB 2: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 2-1: Effect of Missing Inputs to TTL Gates		
Task 2-2: Observe How Hot the Chips Get		
Task 2-3: Gates with Common Outputs		
Task 2-4: Missing Ground and Vcc Connections		
Task 2-5: Build and Test a Common-Collector Buffer Circuit		
Task 2-6: Build and Test a Three-State Buffer Circuit		
Task 2-7: Measuring the Input/Output Characteristic of a TTL Inverter.		
Task 2-8: Measuring VILmax and VIHmin for a TTL Inverter.		
What I Learned		
<b>Lab Data Sheets</b>		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

**Caveat emptor:** Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.

- Title Page: Include **Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted** in all reports.
- Include lab data sheets for all labs. Use lab data sheets to document the lab as you perform it. It saves a lot of time when writing the report.
- Include **Truth Tables or Function Tables** as required to explain how circuits work and as proof of circuit tests.
- **Label** all figures or circuits. (E.g. Figure 1. Schematic of . . ., Figure 2. . . , etc.)
- Refer in the text of your report to all circuits or figures that you include in the body of your report.
- Use LogicWorks™ to draw your circuit schematics and cut/paste them into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

---

Put an 'X' in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use '5' to indicate that you 'strongly agree', '3' to indicate that you are 'neutral', and '1' to indicate that you 'strongly disagree'. Use 'NA', 'Not Applicable', when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your **responses will not be graded**. They are for your instructor's information only.

**Table \_\_: Self-Assessment of Outcomes for Hardware Lab 2: TTL Characteristics, Three-State Buffers, Open-Collector Buffers.**

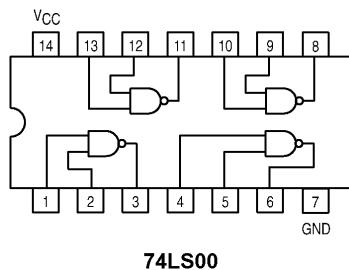
<b>After completing the assigned tasks and report I am able to:</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>NA</b>
Make and interpret electrical voltage measurements.						
Detect and recognize the thermal and electrical effects of driving a common point with two conflicting output signals.						
Describe the different ways in which three-state and open-collector buffer circuits must be controlled to drive a common bus.						
Electrically connect three-state and open-collector buffer circuits to drive a common communication bus.						
Describe the approximate input/output characteristics of an inverter. (Task 2-7and Task 2-8 only.)						
Recognize that the inverter input/output curve is nonlinear. (Task 2-7and Task 2-8 only.)						
Recite a measured value of VIHmin and VILmax for a typical TTL inverter. (Task 2-7and Task 2-8 only.)						

---

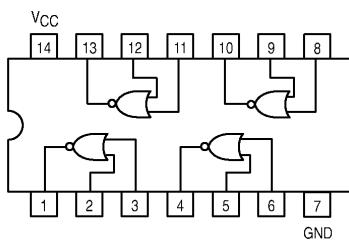
Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

## PIN-OUT DIAGRAMS

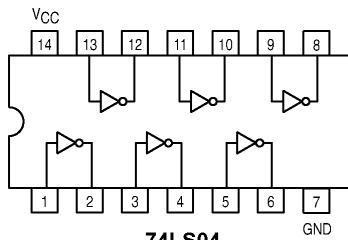
The pin-outs of several TTL devices are given below.



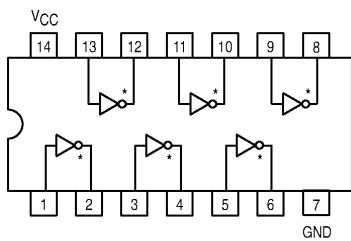
**74LS00**



**74LS02**

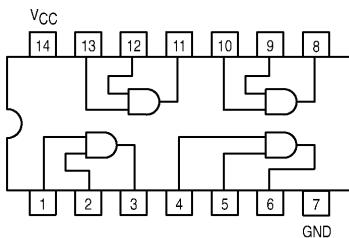


**74LS04**

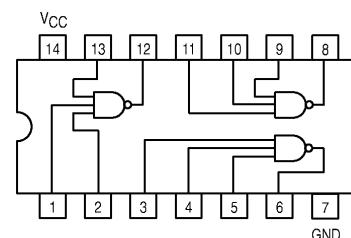


\*OPEN COLLECTOR OUTPUTS

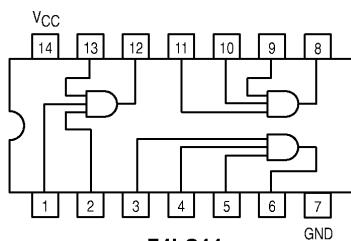
**74LS05**



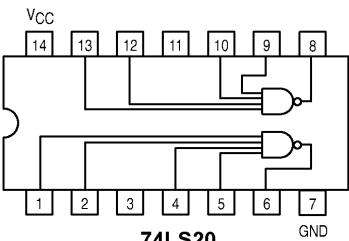
**74LS08**



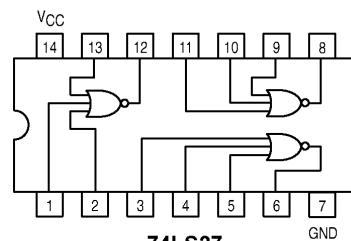
**74LS10**



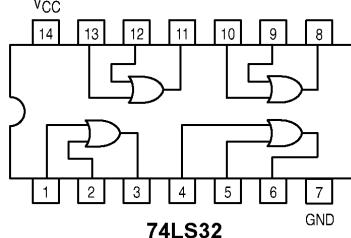
**74LS11**



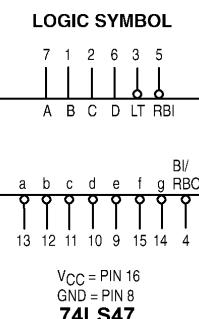
**74LS20**



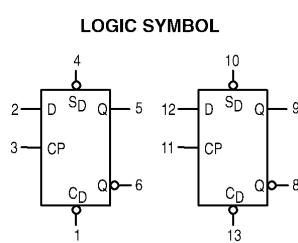
**74LS27**



**74LS32**

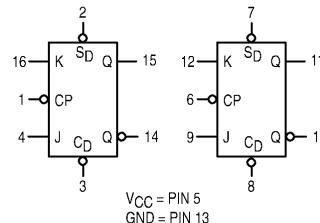


(Copyright of Motorola, Used with permission)



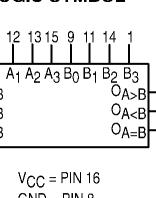
**74LS74**

**LOGIC SYMBOL**

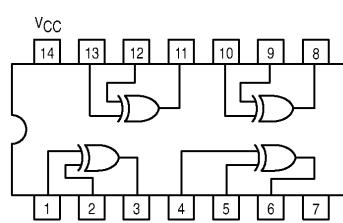


**74LS76**

**LOGIC SYMBOL**

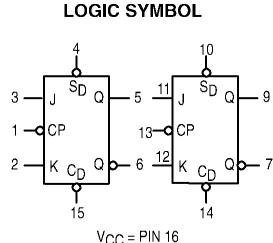


**74LS85**

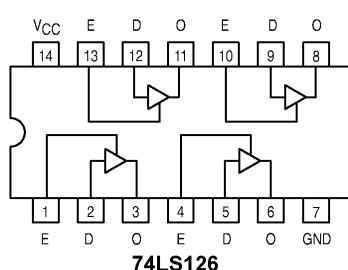


**74LS86**

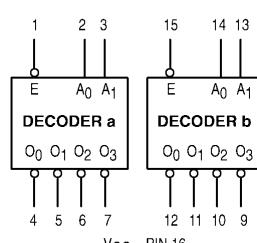
**LOGIC SYMBOL**



**74LS112**

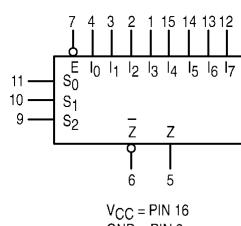


**74LS126**



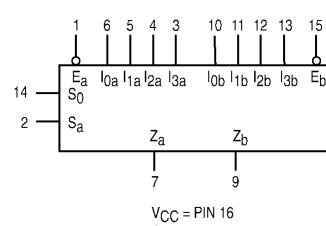
**74LS139**

**LOGIC SYMBOL**



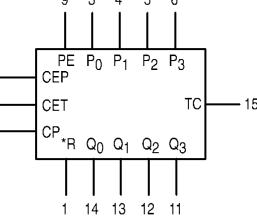
**74LS151**

**LOGIC SYMBOL**



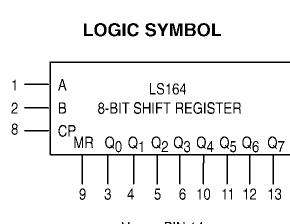
**74LS153**

**LOGIC SYMBOL**



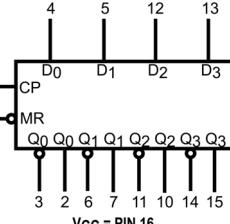
**74LS163**

**LOGIC SYMBOL**



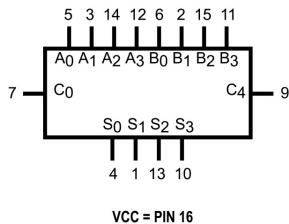
**74LS164**

**LOGIC SYMBOL**



**74LS175**

**LOGIC SYMBOL**



**74LS283**

(Copyright of Motorola, Used with permission)

# HARDWARE LAB 3: LATCHES, FLIP-FLOPS, REGISTERS AND COUNTERS

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Use a prototype board, voltmeter, and logic probe.
- Debug a digital circuit.
- Interpret the function definition table of a D and J-K flip-flop.
- Describe the binary up-counter counting sequence.
- (Task 3-11) Use an excitation table, transition table, and Karnaugh map to design a synchronous machine.

**Equipment:** Digital Trainer Board, Voltmeter, Logic Probe

**Circuit Components:** You will need the following circuits to complete the tasks in this lab:

- (1) 7400 (Quad 2-Input NAND)
- (1) 7402 (Quad NOR gates)
- (1) 7404 (Hex Inverters)
- (1) 7408 (Quad 2-Input AND gates)
- (1) 7474 (Dual D Flip Flops)
- (1) 74112 (Dual J-K Flip Flops)
- (1) 1000 Ohm Resistor

**Objective:** The objective of these laboratory assignments is to allow you to gain some experience in building and using latches, flip-flops and registers. You will also learn to apply your knowledge of the operations of these devices by building a 2-bit binary up-counter<sup>1</sup>.

**Outcome:** When you have completed these laboratory exercise you will be able to:

- Realize and describe the operation of an S-R and D latch.
- Describe the difference between an active-high and an active-low latch.
- Use a D and J-K flip-flop in a circuit.

---

<sup>1</sup> Instructor: I assign this experiment to my students before they cover finite-state machine design; therefore, design of the counter in this experiment is handled using an informal design procedure. If you cover counter design before assigning this experiment you may wish to reorder the tasks so that Task 3-11 follows immediately after Task 3-8

- Realize a de-bouncing circuit.
- Realize an arbitrarily large register using J-K flip-flops.
- Realize a two-bit binary up-counter using J-K flip-flops.
- Design a three-bit binary up-counter using J-K flip-flops.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab it is recommended that you use a task-oriented format when organizing your lab report. (If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Task-Oriented Report Writing Guidelines](#).)

---

## Introduction

In labs 1 and 2, we built combinational logic circuits. Combinational logic circuits have outputs whose values are a function of only the present inputs; the outputs do not depend on the value of the inputs from a second ago or a month ago. A sequential logic circuit, on the other hand, has memory; its outputs depend not only on its present inputs, but also on its past inputs, possibly going back far in time. In this lab, we will experiment with building and debugging sequential circuits.

## The Latch

The ability of a computer to perform complex operations is the result of the synergy of combinational logic circuits, with their ability to implement complex numerical and logical functions, and sequential logic circuits, with their ability to store information. The ability of sequential logical circuits to store information or, equivalently, a history of process they experience, has a fortuitous characteristic: sequential logic circuits can be constructed to act as a controller of a digital computer. In sequential logic circuits, the outputs of gates are fed back to act as their own inputs. This type of connection represents a ‘difference in kind’ from the way we have connected logic gates in the earlier labs. With this type of a connection, the outputs can affect the inputs, which then affect the outputs, etc. *ad infinitum*. Feeding the outputs back to the inputs will make the circuits more difficult to design and debug but will provide us with capabilities combinational logic circuits cannot.

Let’s start understanding these circuits by building a simple memory circuit. The most elementary storage device is the latch. [Figure 3-1](#) shows an active-high S-R latch. The latch is called ‘active-high’ because an input

must be high in order to change the value of the stored data. The stored output of the latch, also known as the state, is labeled Q. The latch also produces the complement of Q,  $\bar{Q}$ , as an output. By definition, a latch is operating properly only when Q and  $\bar{Q}$  are complements of each other.

The inputs that control Q are R and S; R stands for reset and S stands for set. Setting the reset control input to 1 (while S=0) causes the state Q to be reset to 0. Setting the set control input to 1 (while R=0) causes the state Q to be set to 1.

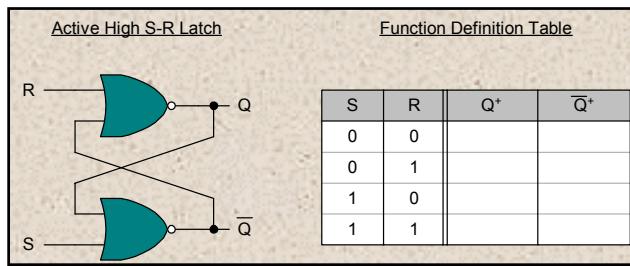


Figure 3-1. Active-high S-R latch and function definition table.

### Task 3-1: Build an Active-High S-R Latch

The first step in this task is to fill in the next-state ( $Q^+$ ,  $\bar{Q}^+$ ) entries in the active-high S-R latch function definition table of Figure 3-1. The entries in the table should be 0, 1, Q (meaning the next state is the same as the present state), or  $\bar{Q}$  (meaning the next  $\bar{Q}$  is the same as the present  $\bar{Q}$ ). Be sure to identify which input pair causes an invalid state, i.e., a state in which  $Q = \bar{Q}$ .

Build the active-high S-R latch shown in [Figure 3-1](#). Using a logic probe or LED, verify the function definition table you filled in. Set S=R=1 and observe the value of Q. Try saving this Q value by resetting simultaneously S=R=0. Do you get the same results when you perform this experiment (i.e., setting S=R=1 then resetting S=R=0 simultaneously) repeatedly?

One fortuitous feature of a storage device, like a latch, is that we can use Q and  $\bar{Q}$  (without the need of an inverter) to drive a combinational logic function that requires both the variable and its complement. Will S=R=1 create a problem for a combinational logic circuit that uses Q and  $\bar{Q}$  from the latch?

### The D Latch

From your answers to the questions of Task 3-1 you no doubt realize that the invalid state listed in the function definition table for the S-R latch can create problems. One way of avoiding this problem is to use the D latch design shown in [Figure 3-2](#).

### Task 3-2: Build a D Latch

Using your knowledge of how the active high S-R latch behaves, complete the function definition table of [Figure 3-2](#) then construct the D latch. How does the D latch avoid the invalid state? D in the name stands for ‘Data.’

Why do you suppose this latch is called a ‘Data’ latch? Using a logic probe or LED verify that the data you entered in the function definition table is correct.

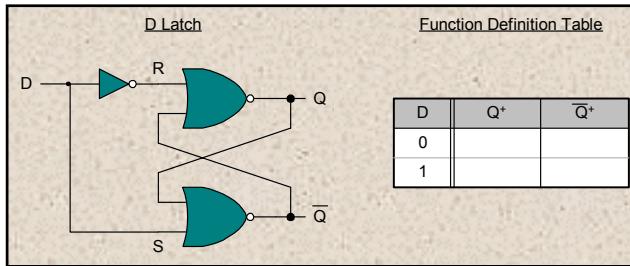


Figure 3-2. D latch schematic and definition table.

One of the limitations of the D latch is that the input D value must be held on the input, otherwise the stored data will change. This limitation can be eliminated by using the D latch with enable shown Figure 3-3.

### Task 3-3: Build a D-Latch with Enable

Using your knowledge of the operation of digital gates, fill in the function definition table for the D latch with enable. Build the D latch with enable shown in Figure 3-3.

Use a logic probe to test the operation of your device and correct your function definition table if necessary.

A circuit is considered to have an active-high enable signal if a value of 1 applied to the enable allows the circuit to change state. In a circuit with an active-low enable, a 0 applied to the enable allows the circuit to change state. Would you classify this as an active-high or an active-low enabled circuit? Why?

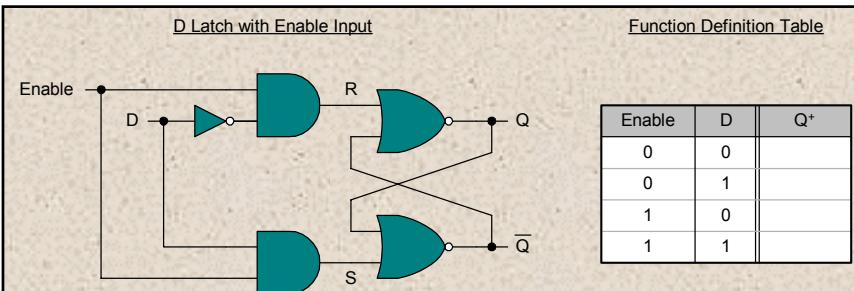


Figure 3-3. Schematic and function definition table for a D-latch with an enable input.

### Latches versus Flip-Flops

A latch is defined by most designers as a device that continuously responds to a change of inputs with or without the need for an enable. (When an enable signal is present as a control input, some textbooks may refer to the device as a flip-flop; consequently the device in [Figure 3-3](#) may be referred to as a flip-flop in your course textbook, although this is not standard.) The latch design shown in Figure 3-3 is usually referred to as a level-triggered device; when the enable or clock signal is in the active state, the latch continuously responds to changes in input values. When the clock signal is inactive, the device's state

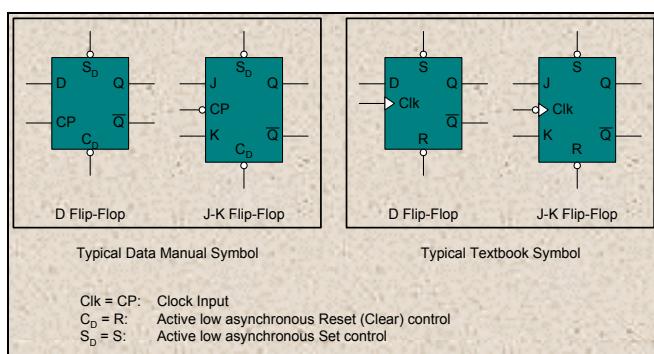
does not change. By contrast, flip-flops are edge-triggered devices. Edge-triggered devices are triggered by clock transitions rather than clock levels. This allows the flip-flop to sample and respond to its input at a precise instant of time. A positive-edge-triggered device will respond to its input only when the clock signal input is in transition from low to high. A negative-edge-triggered device will respond to its input when the clock signal input is changing from high to low. Edge-triggered devices are desirable because they can be easily controlled to sample their input values at a precise instant in time.

## Flip-Flops

The symbols for a positive-edge-triggered D and a negative-edge-triggered J-K flip-flop are shown in Figure 3-4, where ‘CP’ or ‘Clk’ are used to indicate the clock input. (Depending on the textbook or data manual you use as a reference, there are many different notations that you will encounter. Two of the more common notations are shown in Figure 3-4.) The function definition tables for the D and J-K flip-flops are shown in [Table 3-1](#) and [Table 3-2](#) respectively. (These tables use the ‘textbook’ notation shown in Figure 3-4.) The symbols  $\nearrow$  and  $\searrow$  are used to indicate a positive clock transition and a negative clock transition respectively.

The D flip-flop characteristics are the same as the D latch. The J-K flip-flop has characteristics, matching in many ways, the truth table of the active high S-R latch (with J analogous to S and K analogous to R), that you already built. The obvious difference between the function definition tables of the S-R and J-K devices is the replacement of the invalid state of the S-R latch with the toggle operation; when  $J = K = 1$ , Q toggles on each negative clock edge, i.e., takes on the complement of the last value of Q.

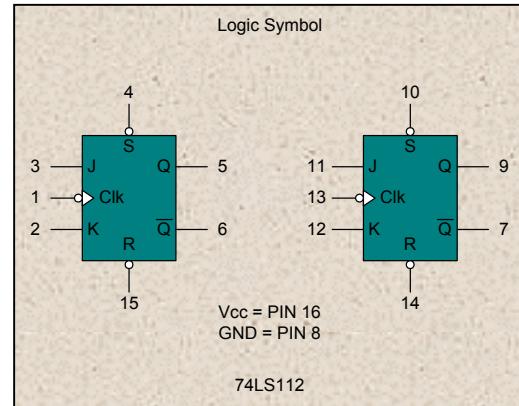
The set and reset inputs shown in [Figure 3-4](#) use the universal graphic symbol for an active low input,  $\neg\circ$ . When low, the S input causes the flip-flop output to be set to 1 asynchronously, i.e., if S is set to 0, the flip-flop output is set immediately to 1, regardless of the value of the clock input. Similarly, when the R input is brought low the flip-flop output is reset asynchronously. In the lab, these pins should be connected to +5 V. if they are not going to be used; otherwise they should be connected to reset logic that is activated appropriately.



**Figure 3-4.** Schematic symbols for positive-edge-triggered D and J-K flip-flops.

### Task 3-4: Verify the Function Definition Tables for D & J-K Flip-flop

Using the TTL 74112 (J-K) and 7474 (D) flip-flops, verify the function definition tables given in [Table 3-1](#) and [Table 3-2](#) for the J-K and D flip-flops. Refer to the pin-out diagrams in Figure 3-5. Note that the input symbol used to indicate the clock input to the 74112 (see Figure 3-5) is the active-low input symbol,  $\neg$ . This symbol, when placed at the clock input, indicates a negative-edge-triggered flip-flop; this means that the device will change state only when a high to low (1 to 0)



transition of the clock occurs. When testing the toggle mode of the J-K flip-flop, be sure to use a de-bounced switch to control the clock input. (To learn more about the need for de-bounced switches, read the next section.) If you're not sure whether the switches you are using are de-bounced, ask your laboratory assistant. If you are using the Elenco Digital Trainer Board, use the switches labeled 'Logic SW', shown as ① in Figure 3-6.

**Table 3-1. Function Definition for D Flip-Flop.**

S	R	Clk	D	Q
1	1	$\neg$	0	0
1	1	$\neg$	1	1
0	1	X	X	1
1	0	X	X	0

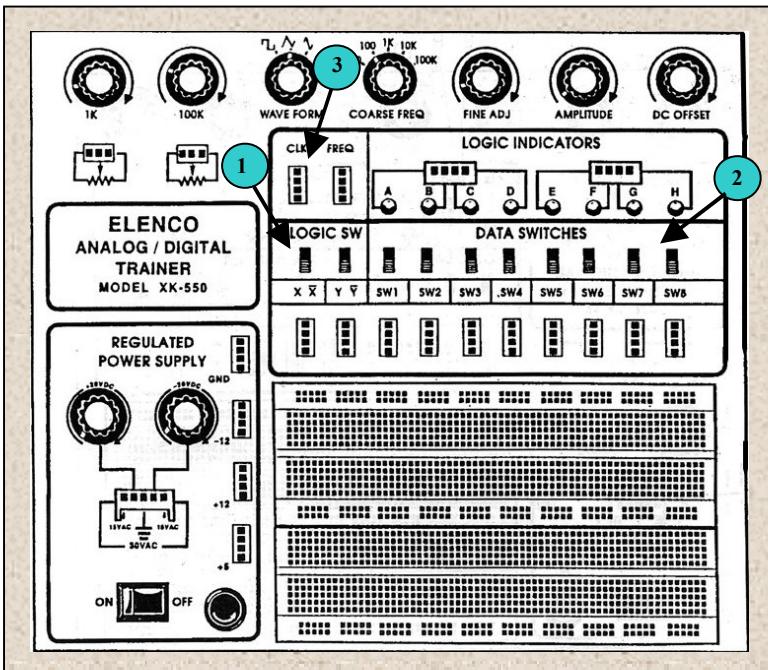
#### De-Bounced Switches

Just as a ball bounces when it is dropped on the floor, a closing switch will make and break contact many times before the mechanical mechanism comes to rest<sup>2</sup>. The mechanical bouncing motion of the switch mechanism dampens so quickly and the amplitude of the bouncing motion is so small that it can only be detected with the use of an electrical oscilloscope. A plot of the output voltage versus time of a switch closing into a +5 V. contact as seen on an oscilloscope might appear as shown in [Figure 3-7](#). Notice that there are a large number of positive and negative edges in this figure. Because the number of edges caused by a bouncing switch cannot be predicted, there is no way to know in advance how many times a

**Table 3-2. Function Definition for J-K Flip-Flop.**

S	R	Clk	J	K	Q
1	1	$\neg$	0	0	Last Q
1	1	$\neg$	0	1	0
1	1	$\neg$	1	0	1
1	1	$\neg$	1	1	Last $\bar{Q}$
0	1	X	X	X	1
1	0	X	X	X	0

<sup>2</sup> In addition to the electrical/mechanical contact caused by a mechanical bouncing motion of the switches contacts, electrical conduction between the contacts of a switch can occur even when no mechanical contact has occurred. As a switch's contacts get close, the electric field strength between the switch's contacts gets strong enough to break down the dielectric effect of the air and cause an arc across the gap. This effect is known as pre-strike. The arcing phenomena associate with pre-strike causes the output voltage to remain at +5 V. (when closing into a +5 V. supply) longer than the duration of the mechanical contact during each bounce.



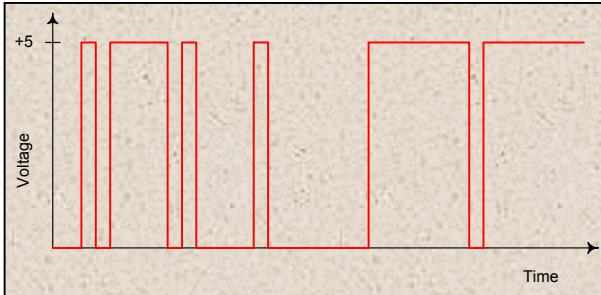
(Used with permission of Elenco Electronics Inc.)

**Figure 3-6.** ELENCO analog/digital trainer board.

### Task 3-5: Prove that the Bouncing Phenomenon is Real

The first task is to demonstrate that the bouncing phenomenon is real. If you have access to a non-de-bounced switch, (such as the data switches labeled ② in (Figure 3-6) use it to control the clock input of a J-K flip-flop in toggle mode, (i.e.,  $J=K=1$ ), and try toggling the flip-flop. If all of your switches are de-bounced, (or if the bouncing motion is not sufficient to cause the flip-flop to repeatedly change state) use a wire to alternately connect the clock input to +5 V.

and ground and show that the result is unreliable. You should observe that the change of state is unreliable. This may take a few tries, so be patient.



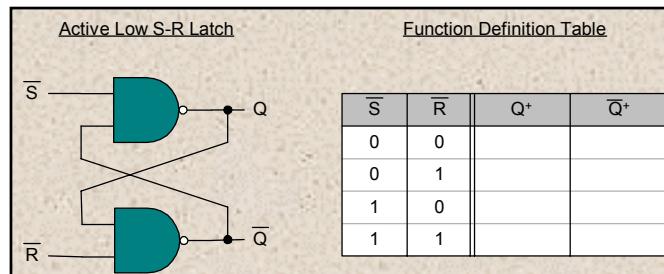
**Figure 3-7.** Plot of voltage v. time for a typical switch operation.

### Task 3-6: Build an Active-Low S-R Latch

In this task, you will build and debug a de-bouncing circuit. Our de-bouncing circuit is an active-low S-R latch shown in [Figure 3-8](#). Build the latch shown in [Figure 3-8](#). Connect the  $\bar{S}$  and  $\bar{R}$  inputs to data switches. Using a logic probe or LED fill in the function definition table for this latch. (Use one of the

J-K edge-triggered flip-flop will toggle (when  $J=K=1$ ); hence the performance in toggle mode will be unreliable. To eliminate this bouncing problem, a switch may be connected to a de-bouncing circuit; this circuit produces an output that has one edge regardless of the number of voltage transitions applied to the input. The next three tasks will take you through the steps needed to build a de-bouncing circuit and test it.

following four values in each square of the table: 0, 1, Q, or  $\bar{Q}$ .) Be sure to identify which input-pair causes an invalid output<sup>3</sup>. Why do you suppose this is called an active-low S-R latch?



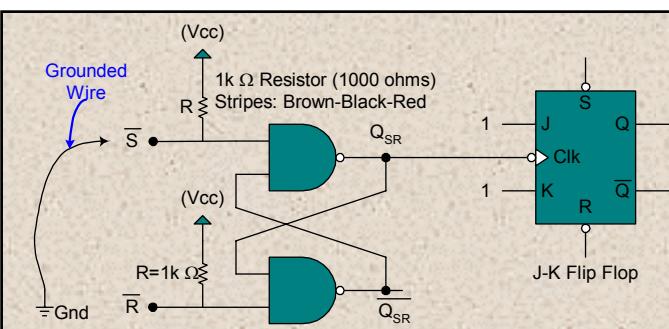
### Using an S-R Latch as a De-Bouncing Circuit

Figure 3-8. Active-low S-R latch schematic and function definition table.

From your experimentation, you noticed that whenever one input is low while the other is high, the output changes state; hence by bringing the inputs low one at a time (e.g., by touching a grounded wire to one of the inputs while the other is kept high through the ‘pull-up’ resistor of Figure 3-9), your output will become either a 1 or 0; further, when the output makes a transition it will make only one transition. To see why this is so, consider the case where the  $\bar{R}$  input is high while you touch a ground wire to the  $\bar{S}$  input. Under these conditions, with a bouncy  $\bar{S}$  input, the input pair driving the latch will take on the following sequence:  $(\bar{S}, \bar{R}) \Rightarrow (0,1) \Rightarrow (1,1) \Rightarrow (0,1) \Rightarrow (1,1) \Rightarrow \dots$  From your experimentation with this latch, you will recognize that the  $(\bar{S}, \bar{R}) = (0,1)$  condition causes the output to become 1. The next input in the sequence,  $(\bar{S}, \bar{R}) = (1,1)$  causes the input to remain at 1, and so on; hence once the output makes a transition it will remain at that value regardless of the input voltage swings on the  $\bar{S}$  input. Let's test your de-bounced switch.

### Task 3-7: Test a De-Bounced Switch

Use the active-low S-R latch you built in Figure 3-9 but disconnect any switches driving the  $\bar{S}$  and  $\bar{R}$  inputs then connect these switches to a  $1k\Omega$  pull-up resistors, as shown in Figure 3-9. Connect the Q output of the active-low S-R latch to the clock input of the negative-edge-triggered J-K flip-flop (74112) you experimented with in Task 3-4 as



shown in Figure 3-9. Set  $J = K = 1$  and alternately bring the  $\bar{S}$  then  $\bar{R}$  inputs to 0 by touching them with a grounded wire. If your de-bounced circuit is working correctly, each time you touch the  $\bar{R}$  input with the ground wire (after touching the  $\bar{S}$  input to ground) the state of the J-K flip-flop should change.

Figure 3-9. De-bounced switch driving a J-K flip-flop.

<sup>3</sup> Remember that an output is invalid when  $Q$  and  $\bar{Q}$  have the same logic value.

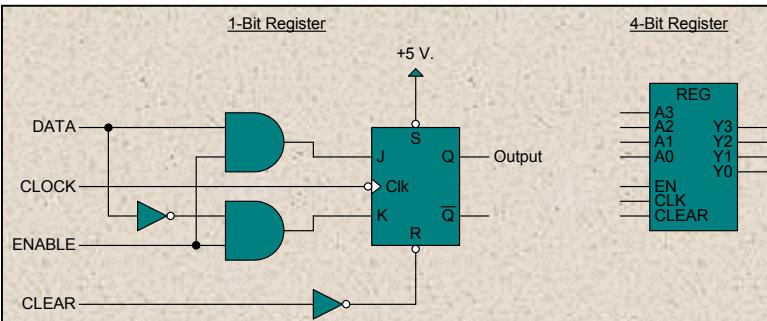
## Registers

A register is a collection of flip-flops and control circuitry used for data storage. The control circuitry in a typical register performs two functions:

- It provides a way to reset to 0 the value of all storage elements in the register.
- It provides a way to disable the register from reading data during a clock pulse.

The circuit for a 1-bit register is shown in [Figure 3-10](#). When the *clear* input is high, the register is reset asynchronously. When the *clear* input is low, the register may respond to the other inputs. When the *clear* input is low the *enable* functions as an active-high load enable input. When the *enable* input is low, the outputs from both of the AND gates are 0, and the state of the register will remain unchanged. When the *enable* input is high,  $J = \text{DATA}$  and  $K = \overline{\text{DATA}}$ ; then on a negative-clock edge the *Q* value of the flip-flop will become *DATA*.

A 1-bit register is of limited use. Typically, we want registers capable of storing multiple bit words. In the Simulation Lab portion of this manual, we are using 4-bit registers capable of storing the values taken from the 4-bit wide data bus, program counter bus, accumulator etc. A 4-bit register, is built by taking four 1-bit registers and connecting them so that they share common *enable* and *clear* input signals. In the Simulation Lab portion of this manual you will be building a simulation of a 4-bit register using this technique, then assigning it to a



**Figure 3-10.** Schematic diagram of a 1-bit register and symbol for 4-bit register.

comparison, why do you think most complex systems are simulated before they are build?

subcircuit using the symbol shown in [Figure 3-10](#). After building the 1-bit register in the next task, extrapolate to estimate the amount of work it would take you to build a 4-bit register. Then compare this with the amount of work it takes to build a simulation of a 4-bit register. Based on this

### Task 3-8: Build a One-Bit Register

Before you enter the laboratory, complete the function definition table for the 1-bit register of Figure 3-10. (Remember that the symbol,  $\overline{\text{L}}$ , in Table 3-3, is used to indicate a negative clock transition.) Your entries in the *Q* column should be one of the following values: 0, 1, or Last *Q*. Build the 1-bit register of Figure 3-10, measure the output for each input condition of [Table 3-3](#) using a logic probe or LED and compare the results with the values you enter in the function definition table. Correct any errors in the table.

## Counters

Counters are synchronous sequential circuits, built using flip-flops, that take on a specific sequence of states upon receiving a succession of clocking events. An important class of counters contains those that count up (i.e., 0, 1,

**Table 3-3. Function Definition Table for a Register.**

CLEAR	ENABLE	CLOCK	DATA	Q
1	X	X	X	
0	0	X	X	
0	1	—	1	
0	1	—	0	

2, etc.) or down (5, 4, 3, etc.). In your class, you may have studied how to use state diagrams, transition tables, and Karnaugh maps to design arbitrarily complex synchronous sequential circuits (See [Task 3-11](#)). To realize a 2-bit counter that counts up, i.e.,  $00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00$ , on successive clock pulses, we could use those techniques; however for a design problem as simple as this one, we will use a heuristic approach for designing the counter using J-K flip-flops. Consider the 2-bit up-counter count-sequence table shown in [Table 3-4](#) and make the following two observations:

- The least-significant bit (LSB) changes on every clock pulse.
- The most-significant bit (MSB) changes subsequent to any clock pulse that occurs when the LSB is 1. (The arrows in the table show this pattern.)

These observations beg the following conclusions:

- If we assign the state of J-K flip-flop A to represent the LSB of the count sequence, then by connecting the A flip-flop to toggle on every clock pulse, i.e.,  $J_A = K_A = 1$ , then  $Q_A$  will follow the count sequence shown in Table 3-4.
- If we assign the state of J-K flip-flop B to represent the MSB of the count sequence, then by connecting the B flip-flop to toggle on every clock pulse when the LSB = 1, i.e.,  $J_B = K_B = Q_A$ , then  $Q_B$  will follow the count sequence shown in Table 3-4.

**Table 3-4. Count Sequence Table**

Clock Pulse Number	Binary Count Value	
	MSB	LSB
Initial Value	0	0
1	0	1
2	1	0
3	1	1
4	0	0
5	0	1
6	Etc.	Etc.

These observations are used to construct the circuit of [Figure 3-11](#). Notice that the counter of this figure is a controlled counter. When the count control is high, the counter counts up because the inputs to the flip-flops are precisely as defined above. When the count control is low, the J and K inputs are low, preventing the flip-flops from changing state.

### Task 3-9: Build and Test a Controlled Two-Bit Binary Up-Counter

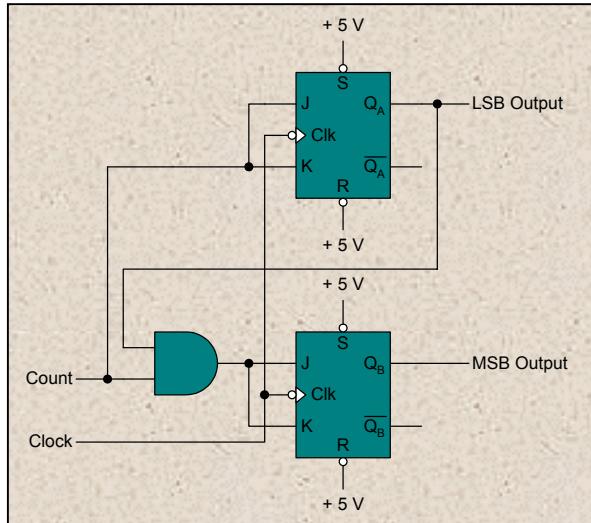
The first step of this task is to fill in the partial transition table, Table 3-5, for the circuit shown in Figure 3-11. The circuit shown in Figure 3-11 uses a count control input (AND'ed with the J and K inputs) to act as an active-high enable for the counter. Next, build and test the 2-bit binary up-counter shown in Figure 3-11 using the 74112 JK flip-flop. Be careful to connect all flip-flop set and reset inputs to +5 V.

Test your circuit using a de-bounced switch to act as the clock input and compare its response to the response you predicted with the truth table. Does it count as your table predicts? If not, correct your table or your circuit. Turn all power off to your circuit then turn the power on again.

**Table 3-5. Partial Transition Table for 2-Bit Binary Up-Counter of Figure 3-11.**

Count Control	Present Count (Binary)	Next Count (Binary)
0	00	
0	01	
0	10	
0	11	
1	00	
1	01	
1	10	
1	11	

Elenco Digital Trainer Board, use the Clock signal labeled ‘Clk’, shown as ③ in Figure 3-6.) Connect the *count* input to a data switch and connect the MSB and LSB outputs to LED's. Observe the clock and the two outputs of the counter on LED's. Does your counter count as expected?



**Figure 3-11.** Schematic of a 2-bit binary up-counter.

What is the initial count value after you turn the power on and before your circuit receives any clock pulses? Turn the power off and on several times and observe the initial state of the counter each time. Does the counter always have the same value when the power is turned on? What circuitry would you add to bring your circuit to a known count value? (Don't build this circuitry, just describe it in your lab report.)

If you have access to a variable frequency clock signal, connect the clock signal to the Clk input and to an LED and set the frequency to about 1 Hz. (If you are using the

### Task 3-10: Design A Three-Bit Counter Circuit

Extend the circuit in [Figure 3-11](#) to make a 3-bit counter using either the heuristic approach used here or traditional techniques for finite state machine design. **You do not have to build this circuit.** In your report:

1. Provide a circuit schematic diagram showing your design.
2. Explain how this circuit works.
3. Describe a TTL implementation of the circuit; e.g. identify IC numbers and the pin connections you would need to make to build the circuit.

### Designing a Controlled Counter Using Traditional Design Techniques

In [Task 3-9](#), the design of a 2-bit binary up counter was completed using a heuristic approach. In that task, you were asked to complete the partial transition table. In the next task you are asked to complete the design of a 2-bit up-counter using transition tables and Karnaugh maps.

### Task 3-11: Complete Design Procedure for 2-Bit Binary Up-Counter

Using traditional design techniques, complete the design of a 2-bit binary up counter. Your counter should count up when the *count* control value is high; otherwise, the count should remain unchanged. First, complete the transition table (Table 3-7) for a 2-bit binary up counter assuming that it will be built using J-K flip-flops. (The excitation table of the J-K flip-flop is shown in Table 3-6.) Then use the Karnaugh maps ([Table 3-8](#)) to show that the design arrived at heuristically in [Task 3-9](#) is consistent with the design arrived at using formal finite state machine design techniques.

Table 3-6. J-K Excitation Table.

Present State	Next State	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Table 3-7. Transition Table for 2-Bit Binary Up-Counter of Figure 3-11.

Count Control	Present State		Next State					
	Q <sub>A</sub>	Q <sub>B</sub>	Q <sub>A</sub>	Q <sub>B</sub>	J <sub>A</sub>	K <sub>A</sub>	J <sub>B</sub>	K <sub>B</sub>
0	0	0						
0	0	1						
0	1	0						
0	1	1						
1	0	0						
1	0	1						
1	1	0						
1	1	1						

Table 3-8. Karnaugh Maps for Use with Design of 2-bit Binary Up-Counter.

		J <sub>A</sub>						J <sub>B</sub>					
		Q <sub>A</sub> Q <sub>B</sub>	00	01	11	10			Q <sub>A</sub> Q <sub>B</sub>	00	01	11	10
Count		00					Count		00				
0		0					0		0				
1		1					1		1				
		K <sub>A</sub>						K <sub>B</sub>					
		Q <sub>A</sub> Q <sub>B</sub>	00	01	11	10			Q <sub>A</sub> Q <sub>B</sub>	00	01	11	10
Count		00					Count		00				
0		0					0		0				
1		1					1		1				

# HARDWARE LAB 3: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 3-1: Build an Active-High S-R Latch		
Task 3-2: Build a D Latch		
Task 3-3: Build a D-Latch with Enable		
Task 3-4: Verify the Function Definition Tables for D & J-K Flip-flop		
Task 3-5: Prove that the Bouncing Phenomenon is Real		
Task 3-6: Build an Active-Low S-R Latch		
Task 3-7: Test a De-Bounced Switch		
Task 3-8: Build a One-Bit Register		
Task 3-9: Build and Test a Controlled Two-Bit Binary Up-Counter		
Task 3-10: Design A Three-Bit Counter Circuit		
Task 3-11: Complete Design Procedure for 2-Bit Binary Up-Counter		
What I Learned		
<b>Lab Data Sheets</b>		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

<p><i>Caveat emptor:</i> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.</p> <ul style="list-style-type: none"> <li>Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.</li> <li>Include lab data sheets for all labs. Use lab data sheets to document the lab as you perform it. It saves a lot of time when writing the report.</li> </ul>
--

- Include **Truth Tables** or **Function Tables** as required to explain how circuits work and as proof of circuit tests.
- **Label** all figures or circuits. (E.g. Figure 1. Schematic of . . . , Figure 2. . . , etc.)
- Refer in the text of your report to all circuits or figures that you include in the body of your report.
- Use LogicWorks™ to draw your circuit schematics and cut/paste them into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

Put an 'X' in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use '5' to indicate that you 'strongly agree', '3' to indicate that you are 'neutral', and '1' to indicate that you 'strongly disagree'. Use 'NA', 'Not Applicable', when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your **responses** will not be graded. They are for your instructor's information only.

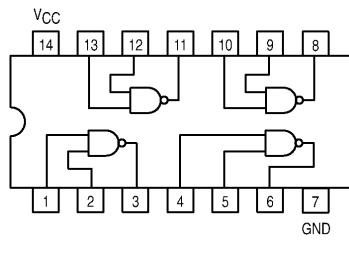
**Table \_\_: Self Assessment of Outcomes for Hardware Lab 3: Latches, Flip-Flops, Registers and Counters**

After completing the assigned tasks and report I am able to:	5	4	3	2	1	NA
Realize and describe the operation of an S-R and D latch.						
Describe the difference between an active-high and an active-low latch.						
Use a D and J-K flip-flop in a circuit.						
Realize a de-bouncing circuit.						
Realize an arbitrarily large register using J-K flip-flops.						
Realize a two-bit binary up-counter using J-K flip-flops.						
Design a three-bit binary up-counter using J-K flip-flops.						

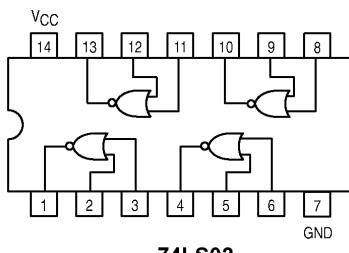
Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

## PIN-OUT DIAGRAMS

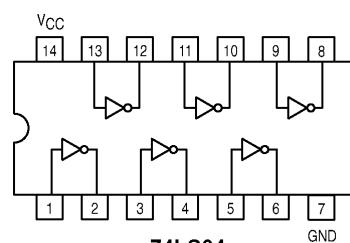
The pin-outs of several TTL devices are given below.



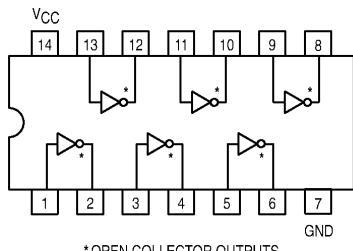
**74LS00**



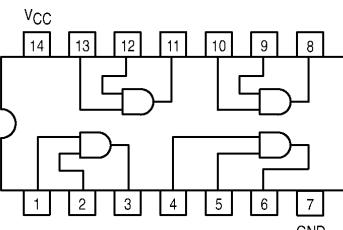
**74LS02**



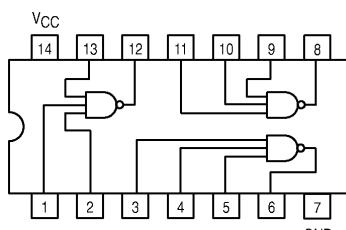
**74LS04**



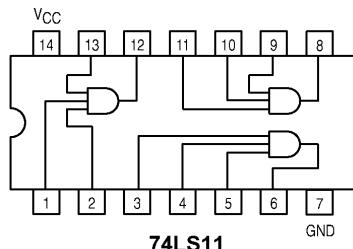
\* OPEN COLLECTOR OUTPUTS  
**74LS05**



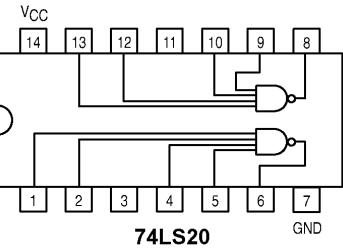
**74LS08**



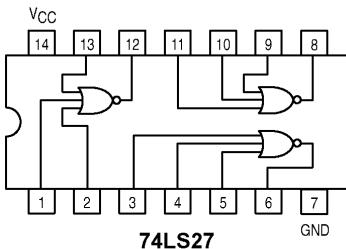
**74LS10**



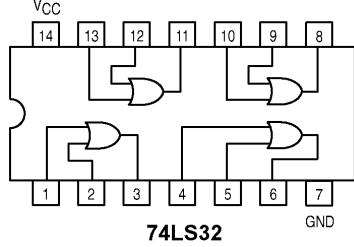
**74LS11**



**74LS20**

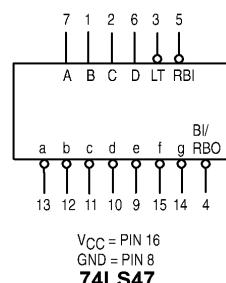


**74LS27**



**74LS32**

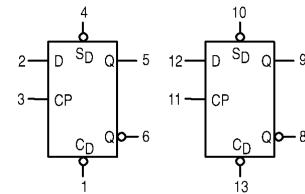
LOGIC SYMBOL



V<sub>CC</sub> = PIN 16  
GND = PIN 8

**74LS47**

LOGIC SYMBOL

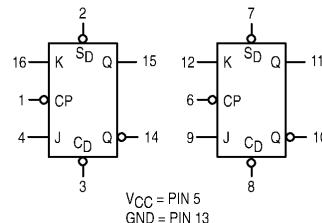


V<sub>CC</sub> = PIN 14  
GND = PIN 7

**74LS74**

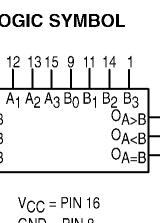
(Copyright of Motorola, Used with permission)

**LOGIC SYMBOL**

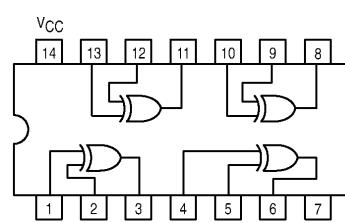


**74LS76**

**LOGIC SYMBOL**

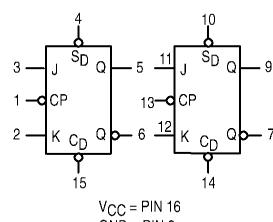


**74LS85**

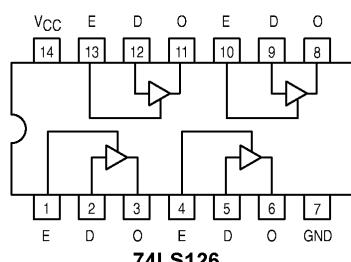


**74LS86**

**LOGIC SYMBOL**

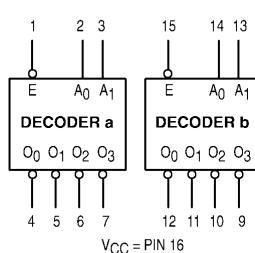


**74LS112**



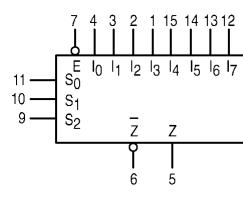
**74LS126**

**LOGIC SYMBOL**



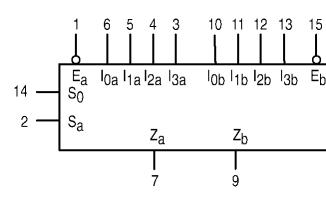
**74LS139**

**LOGIC SYMBOL**



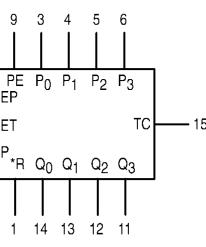
**74LS151**

**LOGIC SYMBOL**



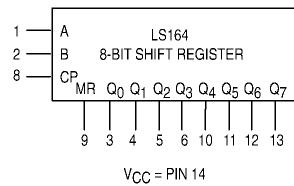
**74LS153**

**LOGIC SYMBOL**



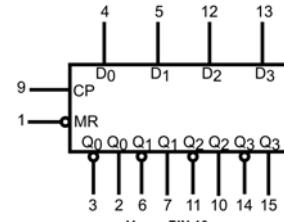
**74LS163**

**LOGIC SYMBOL**



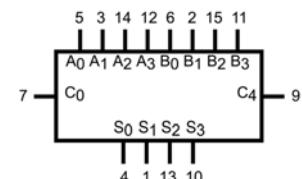
**74LS164**

**LOGIC SYMBOL**



**74LS175**

**LOGIC SYMBOL**



**74LS283**

(Copyright of Motorola, Used with permission)

---

# HARDWARE LAB 4: VENDING MACHINE CONTROLLER DESIGN

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Use a prototype board, voltmeter, and logic probe.
- Describe the truth tables that characterize the primitive logic gates (i.e., AND, OR, NOT, NAND, NOR.)
- Use DeMorgan's Law.
- Perform binary addition of multi-bit numbers.
- Convert between decimal and binary representations of numbers.
- Describe the function of a register.
- Use LogicWorks™.

**Equipment:** Digital Trainer Board, Voltmeter, Logic Probe.

**Circuit Components:** You will need the following circuits to complete the tasks in this lab:

- (1) 7402 (Quad 2-Input Nor Gates)
- (1) 7432 (Quad 2-Input Or Gates)
- (1) 74LS85 (4-Bit Magnitude Comparator) (NOT 74F85)
- (1) 74175 (Quad D Flip-Flop)
- (1) 74283 (4-Bit Binary Full Adder with Fast Carry)

**Objective:** In this laboratory exercise, you will gain some experience using medium-scale-integrated

(MSI) circuits to build a controller for a simple vending machine. You will gain an appreciation of the power of MSI circuits to simplify circuit designs. You will also gain an understanding of the design process, and the role of simulation in circuit design.

**Outcomes:** When you have completed this laboratory exercise, you will be able to:

- Describe the function of and use a 7485, 4-bit magnitude comparator.
- Describe the function of and use a 74283, 4-bit binary full adder.
- Describe the function of and use a 74175, quad D flip-flop.

- Interpret and use the function definition table of an MSI circuit.
- Simulate a controller for a simple vending machine.
- Build and debug a controller for a simple vending machine.
- Design and simulate an 8-bit controller for a vending machine.
- Appreciate the role of MSI circuits in the design of digital circuits.
- Describe the role of circuit simulation in prototype construction.
- Describe some aspects of the design process.

## Lab Report Guidelines

Before you begin this laboratory exercise, make sure you understand the reporting requirements your instructor has specified. If you understand the reporting requirements then you will know the observations that are important to record in your lab notebook and the tests that need to be performed on the circuits you construct. For this lab, it is recommended that you use a top-down format when organizing your lab report. (If your instructor is using the report writing guidelines contained in this manual, you can review them by clicking on the blue text: [Top-Down Report Writing Guidelines](#).)

---

## Prologue

The previous laboratory exercises contained in this manual have largely used what is known as small-scale integrated (SSI) circuits. SSI circuits are so called because each IC is built using relatively few components. (SSI circuit packages typically contain fewer than 6 gates, or 1 (or at most 2) flip-flops.) Packaged circuits using many logic gates in their construction, such as registers and adders, are known as medium-scale integrated (MSI) circuit packages. (MSI circuit packages typically contain more than 6 but less than 100 gates.) In this laboratory exercise, you will gain some experience using MSI circuits to build a digital controller for a vending machine. It is hoped that upon the completion of this laboratory experiment, you will appreciate the power of MSI circuits to simplify circuit designs.

In the previous hardware laboratory experiments the circuits that you designed and build were rather simple; hence any design errors you (or we) made could be easily corrected in the laboratory with some minor re-wiring. For more complex design, this is not the case. With complex designs, errors in the design can affect many aspects of the design, causing the designer to completely disassemble their hardware prototype and start from scratch building a new design. This process can be time consuming, especially if significant errors are found in the second and third generation designs. In order to save time, all complex circuits are simulated before they are built. With simulations, errors can be corrected easily and exhaustive tests conducted quickly to validate all aspects of the designs performance. Once the design meets all of the design specification, a hardware prototype is constructed, and measurements made to

validate its performance. In this laboratory exercise you are going to go through this same process. First you are going to simulate the circuit to prove that it will indeed perform as desired and meet all design specifications. Once satisfied, you will then construct the circuit in the hardware laboratory. It is hoped that when you complete this laboratory experiment you will have your own ideas about the role that simulation plays in digital-circuit design.

## Introduction

Depending upon how far along you are in your course material, you may (or may not) have studied the use of state diagrams, transition tables and Karnaugh maps to design synchronous sequential machines. These techniques are powerful and are used every day in the digital electronics industry to build both simple and complex synchronous machines. These techniques are used for two reasons: they are guaranteed to produce a design regardless of the complexity of the specification and they can be structured to yield a design that uses the minimum amount of digital logic. Designs that use the minimum number of logic gates are important in many contexts. For example, if you are building a widget that will be manufactured in the thousands or millions and you have to compete with other manufacturers, who are building similar widgets, minimizing the number of gates, and hence cost, may be of paramount importance. In this situation, the cost of production, rather than the cost of the design process, dominates the cost of your product.

There exist many situations where the resources consumed in designing, prototyping, and building a device are the dominant costs, not the costs of the logic gates used to build the device. These situations occur when you need a one-of-a-kind device or a device that will be replicated few times. In these situations you will be building these devices from IC's rather than creating a single IC to perform the task. You will be concerned less about the number of gates used and more about the number of IC's you need and the concomitant wiring demands. In these cases, you may be looking for a quick way to build a device with few MSI IC's rather than a truly minimal design that may require many SSI IC's. If we are so constrained, the design procedure used is much less formal. It reduces to:

- clearly understanding the given design specifications and discovering any additional requirements needed to complete the design specification,
- recognizing (from experience with digital IC's) which IC's provide the capabilities that can be used to realize the functions demanded of the design,
- forming a paper design that meets the design specification (usually with MSI circuits) using cut-and-try methods, then
- prototyping, debugging, and producing the design.

In this laboratory exercise, you will be taken through the design procedure that uses this approach to design a vending machine controller. Do not be concerned that you may have learned the state diagram and transition table techniques for naught; you will have ample opportunity in Hardware Lab 5 to use the formal design procedures you have learned.

## Vending Machine Controller Design

The function of a vending machine controller is to tally the money deposited into a vending machine, to return money when requested from the user and to allow the vending mechanism to operate once sufficient money has been deposited. The outputs of such a vending machine controller depend on its current inputs (such as the value of the inserted coin, or a request for return of money) as well as its past inputs (such as the tally of the previous coin deposits). A digital circuit whose output depends on its past as well as present inputs is called a sequential circuit or finite state machine (FSM). When the change of state of the FSM is synchronized with a clocking event, we give the machine a special name: synchronous FSM or synchronous machine for short. In this laboratory exercise, we will design a synchronous machine that will function as a controller for a vending machine.

The first step in any synchronous machine design is to obtain a functional specification. The purpose of the *functional specification* is to define all of the functions that the device must be capable of performing. This specification is informal many times and (more importantly) incomplete. By incomplete, we mean that it contains insufficient information to completely define our design.

For example, the functional specification of a vending machine controller might be given to you verbally as follows:

*"I'd like you to design a controller for a vending machine that dispenses only one type of product. The product vended will cost \$0.30 but we need to have the capability of raising the price. Your controller should allow users to request their money back if they change their minds. You should do this in your design by providing a digital output line that produces a +5 V. signal to enable the money return mechanism when a user presses the coin return lever on the front of the machine."*

*"We'll be using these controllers to replace old controllers in existing vending machines on the campuses of educational institutions. The existing vending machines already have an electronic circuit that produces a +5 V. signal on one line when the product is requested and a +5 V. signal on another line when the coin return lever is depressed. Your controller will have to produce a digital output signal that is high when \$0.30 has been deposited to enable the vending mechanism of the existing vending machine."*

*Any questions?"*

This initial functional specification statement, as we shall see, is indeed incomplete<sup>1</sup>. This means that as we proceed with the design, we will encounter many questions whose answers cannot be found in the informal functional specification quoted above. The sum total of the way we answer these questions will determine the complete *design specification* of the product. The *design specification* specifies in detail the values (in Volts, Amps, logic values, etc.) of all inputs to and outputs from our device and the operations that the device will perform (including any limit on propagation delay time) for every input condition.

To move from a functional specification to a design specification, we will need to answer many questions. In practice, when we encounter questions, we would ask them to the people responsible for using our design so that the end product we produce does everything the way they would want it had they thought through the design thoroughly. If there is no one to get answers from, we could examine one of the existing controllers our controller is meant to replace; chances are, our new design will have to perform many of the same functions - our controller will certainly have to use the same inputs and produce the same outputs. If we can't find an old controller and if there is no one to answer our questions (such as in this exercise) we need to make reasonable engineering assumptions using our own experience of the way vending machines operate and proceed based on those assumptions. Let's pose and answer some questions that we'll need to address to complete the vending machine controller design specification.

## Assumptions

**In the verbal functional specification, we are told that our design will need to accommodate a price change. Will the price change be given in increments of \$0.01, \$0.05, \$0.10, etc.?**

*Assumption:* It will probably work like all vending machines and have \$0.05 increments.

**What might the price rise to ultimately?**

*Assumption:* If the price is \$.30 cents today, and *assuming* a 5% inflation rate, the price will hit \$1.00 if we *assume* a life span of 25 years. Since the useful life of this product is probably less than 25 years, *assuming* an ultimate price of \$1.00 is reasonable.

**How will the price be made available to our controller?**

*Assumption:* Since our price will be a multiple of \$0.05, we will save on hardware if we *assume* that the price will be a binary value representing the number of \$0.05 increments

---

<sup>1</sup> The author has never worked on a project where the initial functional specification provided enough information to completely specify all aspects of the final design.

needed before we will vend our product. This means that if the price is 30 cents, the price input to our machine will be  $30D/5D = 6D = 0110B$ . Since the price might eventually increase to 100 cents, our binary price inputs must be capable of representing numbers at least as big as  $100D/5D = 20D = 10100B$ . This means that there must be at least five 1-bit price input lines to our controller.

**What should we do if someone puts in too much money?**

*Assumption:* This is a machine at an educational institution so we won't have to give change.

**When we plug in our vending machine and power is first supplied to the controller, how can we guarantee that the controller is initialized to a state consistent with zero money deposited?**

*Assumption:* Our design will need to contain a means to reset the controller so that when power is applied, it is in a state corresponding to a deposit of no money. The reset may need to be performed manually or automatically. Let's assume that an active-high external reset control signal is currently available in the existing vending machines for this reset function and that it will be triggered manually.

**How can we detect whether a nickel, dime or quarter has been inserted?**

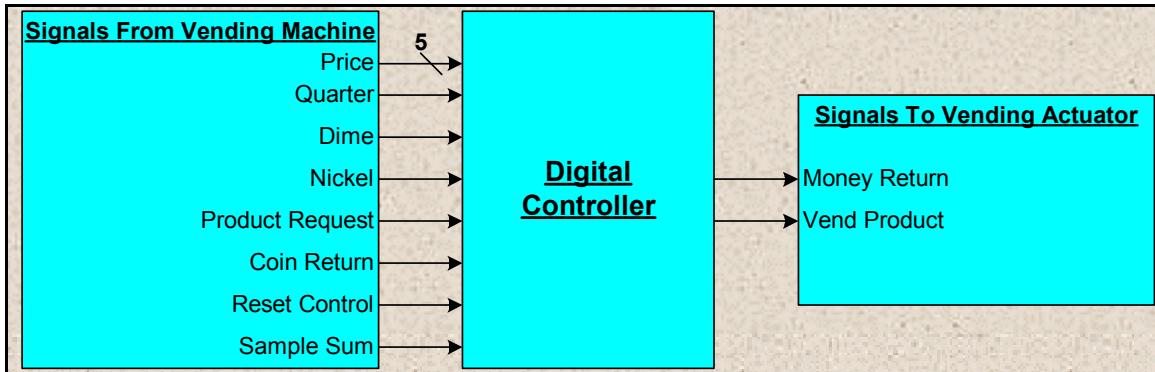
*Assumption:* The existing vending machine must already have a mechanism for detecting the denomination of the coins inserted and providing this information to the existing controller. We will assume that digital signals from the vending machine coin mechanism are available as inputs to our controller and will take on the following values:

- A *quarter* input line momentarily equals 1 if a quarter is inserted, 0 if no quarter is inserted.
- A *dime* input line momentarily equals 1 if a dime is inserted, 0 otherwise.
- A *nickel* input line momentarily equals 1 if a nickel is inserted, 0 otherwise.
- A *sample-sum* input line signal contains one  $0 \rightarrow 1 \rightarrow 0$  digital-valued pulse that occurs only during the time any coin input line (*quarter*, *nickel* or *dime*) is momentarily high. (This signal will be used to trigger a circuit that will sample the coin input lines when a coin is inserted.)

We'll also assume that only one of the coin-sensing lines (*quarter*, *nickel* or *dime*) is high at any one time.

By asking and answering questions about the design, we have obtained all of the information you will need to complete your design specification. (Including all of the inputs needed by and outputs needed

from our controller as is shown in [Figure 4-1](#).) Ideally the complete design specification should be completed before the design is begun, as you will do in Task 4-1. In practice, however, there are often things that are overlooked, so that additional specifications are added and inconsistencies in the design specification are uncovered and corrected as the design unfolds.



**Figure 4-1.** Input/output specification of a digital controller.

### Task 4-1. Create Your Design Specification

Using the functional specification and the answers to the questions above, create a design specification for the design we're creating. This specification should list all inputs and outputs and should specify that each signal is a logic value<sup>2</sup>. Also describe in a concise but thorough way, how this controller should respond to each of it's inputs.

### The Design Process

The design process consists of producing a detailed schematic diagram, which, when realized, satisfies the complete design specification. When the product to be designed is complex, this schematic diagram is usually constructed using a combination of what is known as a top-down process and a bottom-up process.

In a top-down approach, the design process starts with a high-level view of the completed product. The product to be designed is represented first as an interconnection of high-level interacting subsystems. The signals used to allow these subsystems to interact and specifications detailing the way each subsystem must respond to its inputs are precisely defined at this stage. Figure 4-1 along with the design specification you created in Task 4-1 is the first step in the top down design process.

---

<sup>2</sup> In real applications, some signals may not be logic values. In such cases we would need to design analog circuitry to convert these to logic signals.

Next, each subsystem is broken down into smaller interacting subsystems, a design specification for each of these subsystems is defined, and so on, until we reach a stage where the circuitry needed to construct a subsystem is obvious to the designer.

By contrast, in a bottom-up approach, existing IC's and gates are assembled to build subsystems. These subsystems are assembled to form bigger subsystems, and so on, until the requirements of the specification are satisfied.

The design we are embarking on is simple enough that the top-down and bottom-up approaches are almost the same. As we proceed, you will see aspects of both in the design process.

## A Top-Down and Bottom-Up Design

Two of the functions that the controller needs to perform are:

- Tally the amount of money deposited.
- Compare the amount deposited with the price.

There are MSI circuits that can implement these two functions: one is an adder, and the other is a comparator. Using these two MSI circuits, and a register for storage, a signal flow diagram of one possible partial design of our controller is shown in Figure 4-2.

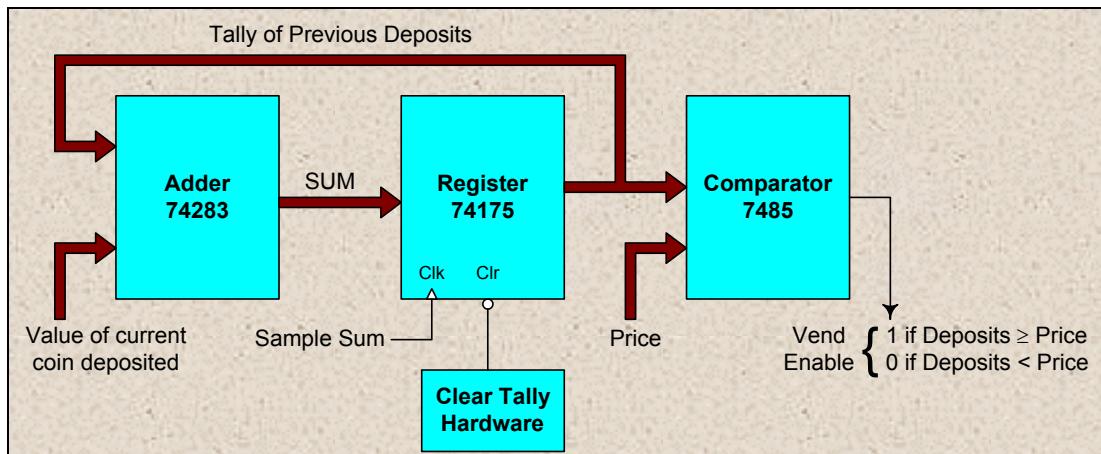


Figure 4-2. Signal flow diagram of vending machine controller.

In the design of Figure 4-2, the adder adds the value of the current coin deposited to the tally of the coins deposited since the last product was vended. This sum, stored in the 74175 register, represents the state of our finite-state machine. When the *sample-sum* (synchronizing) input has a positive edge, the 74175 register is loaded with the sum, thereby updating the state of the synchronous machine. (Recall our assumption that a *sample-sum* signal is available to sample the value of the adder output at the time a coin is inserted.) The output of the register is fed back to the 'A' inputs of the adder so that it may be

added with subsequent coin deposits. The output of the register is also fed to a comparator. The comparator is a digital circuit that compares its two inputs and produces several output signals. The output we'll need is the one that is active (high) when the tally input (sum) is greater than or equal to the price input. The *clear-tally* hardware is a circuit that we'll design to clear the tally under the appropriate conditions – such as once the product has been vended or when a customer wants their money back.

Our design approach has aspects of both a top-down and bottom-up design. It looks like a top-down approach because we have broken the controller of Figure 4-1 into the four interacting subsystems. By immediately recognizing that three of the subsystems in Figure 4-2 can be realized with available MSI IC's, the design process also has aspects of a bottom-up approach. Because the *clear-tally* hardware cannot apparently be realized using only one IC, it is left defined only as a functional block that will require further design work. Let's use a bottom-up design procedure to fill in the details of the *clear-tally* functional block.

## Bottom-Up Clear-Tally Hardware Design

From our own experience with vending machines, we will *assume* that the tally should be cleared when one of the following happens:

- Someone activates the coin return, OR
- We vend the product.

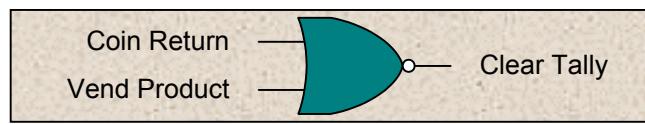


Figure 4-3. Partial clear-tally hardware design.

Since our *clear-tally* line is an active low signal, a simple NOR gate will implement this function (shown in [Figure 4-3](#)) provided the *coin-return* and *vend-product* signals are active high – which we know to be true from our original problem specification.

We must design the *vend-product* signal to be high (and actuate the vending mechanism) when the following is the case:

- The *product-select* line is high (i.e., the product has been selected), AND
- The *vend-enable* line (see Figure 4-2) is high, (i.e., the tally equals or exceeds the price.)

Using this understanding, the *vend-product* signal can be represented as  $(\text{product-select}) \bullet (\text{vend-enable})$ . The complete design of the *clear-tally* hardware that results is shown in Figure 4-4.

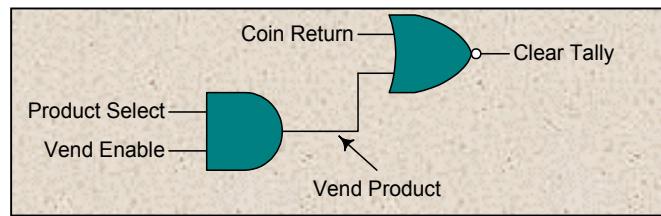


Figure 4-4. Clear-tally hardware design.

If we could look ahead at the completed design, we would see that the *only* AND and NOR gates we will need in the completed design will be

those used in the *clear-tally* hardware of [Figure 4-4](#). If we use the design of Figure 4-4, we will have to use two IC's for the *clear-tally* hardware: one for the AND gate and one for the NOR gate. We can minimize the cost of our design, if we build the *clear-tally* hardware using only one IC. We can do this using only a 7402 (quad 2-input NOR gates) and the laws of Boolean algebra.

The *clear-tally* hardware realizes the Boolean algebraic function:

$$\text{ClearTally} = \overline{\text{ProductSelect}} \bullet \overline{\text{VendEnable}} + \text{CoinReturn}$$

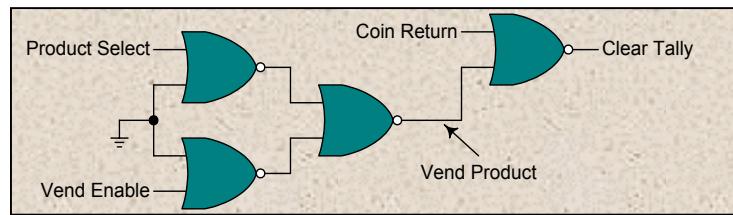
If we double complement the product term we get,

$$\text{ClearTally} = \overline{\overline{\text{ProductSelect}}} \bullet \overline{\overline{\text{VendEnable}}} + \text{CoinReturn}$$

Then applying DeMorgan's law to this term we get,

$$\text{ClearTally} = \overline{\text{ProductSelect}} + \overline{\text{VendEnable}} + \text{CoinReturn}$$

This equivalent form of the *clear-tally* function can be built using two NOR gates and two inverters. If we connect two of the four NOR gates of the 7402 to function as inverters



we can build the circuit of [Figure 4-4](#) **Figure 4-5.** Tally clear hardware using one 7402 IC.

using only one 7402 IC. Prove to yourself that the design [Figure 4-5](#) is equivalent to that of Figure 4-4.

## Adder Hardware Design

Let's design the adder hardware next. The first question we need to answer is how many input bits our adder will need and how many output bits it must produce. Using our assumption that the smallest coin accepted by the coin mechanism is a nickel, (and that all other coins are multiples of 5 cents) we can minimize the number of adder input (and output) lines if each adder increment represents 5 cents. This means that our adder will function by adding the effective value of each deposit (5 for a quarter, 2 for a dime and 1 for a nickel) to the existing sum. This also means that largest input to the adder from the coins sensing mechanism will be 0101, the binary equivalent of 5 for a quarter. The other input to the adder, the tally, must be capable of representing the ultimate price of our product, \$1.00, which, in 5-cent increments, is represented by the binary string, 10100; hence our adder must be capable of at least 5-bit arithmetic.

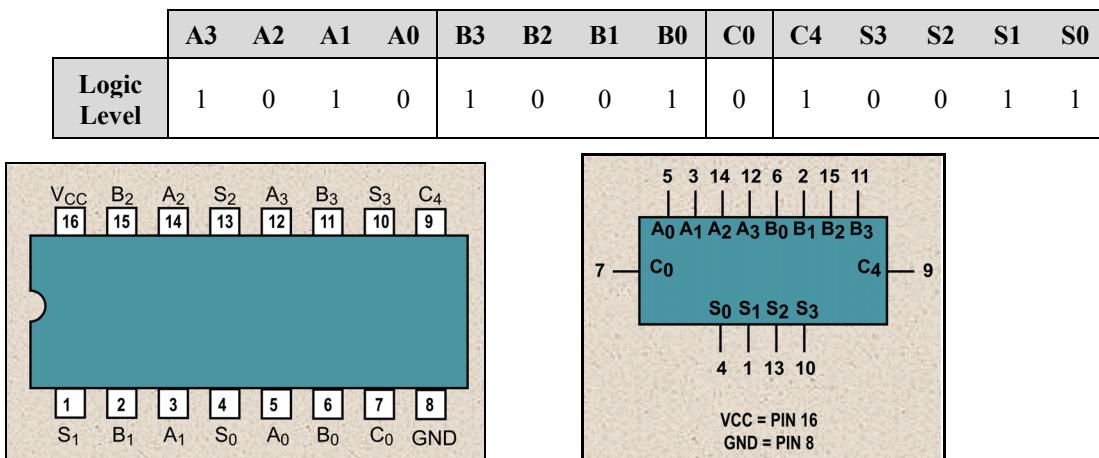
Realizing that MSI circuits allow us to do 4-bit arithmetic easily, but require a fair amount of extra work to do 5-bit arithmetic, let's revisit our assumption that the ultimate price of the product may increase to \$1.00. Recall that this assumption was based on a 25-year life time with a 5% inflation rate. If we assume either a 20 year life span or a 3.7% inflation rate, the ultimate price of our product will be \$0.75, which is represented in our adder by  $75D/5D = 15D = 1111B$  – which only requires 4-bit arithmetic! Let's *assume* that our revised assumption is reasonable. This means that we can design our adder to work using 4-bit arithmetic.

An MSI circuit which does 4-bit binary arithmetic is the 74283, 4-bit binary full adder (with fast carry). The connection (pin-out) diagram and logic symbol for the 74283 are shown in Figure 4-6<sup>3</sup>. The 74283 adder is capable of adding two 4-bit operands (supplied to the A and B input lines) to a 1-bit incoming carry (supplied to the C0 input line) and producing a 5-bit result on lines C4, S3, S2, S1, and S0. Using base-10 arithmetic, the input output relationship of the 74283 is given by,

$$C0 + (A0 + B0) + 2 * (A1 + B1) + 4 * (A2 + B2) + 8 * (A3 + B3) = S0 + 2 * S1 + 4 * S2 + 8 * S3 + 16 * C4$$

where, for this equation, + represents plus and \* represents multiplication. An example of the addition operation is shown in Table 4-1.

**Table 4-1. Example Operation of the 74283 4-Bit Binary Adder.**



**Figure 4-6.** Pin-out diagram and logic symbol for the 74283 4-bit binary full adder.

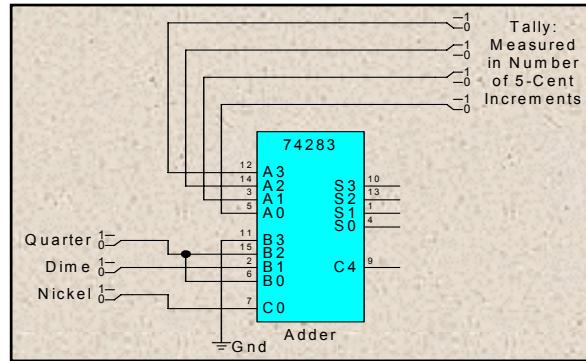
Let's use the A inputs, A3 – A0 input lines, to provide the present tally of previous deposits to the adder as shown in [Figure 4-7](#). (In the schematic diagram of Figure 4-7, the pin-label postscripts 3 and 0 indicate the most and least significant bits, respectively.). Let's use the B inputs, B3 - B0 input lines, to provide the value (divided by 5) of the present coin input (for quarters and dimes) and the carry input, C0, to act as the nickel input as shown in Figure 4-5. Using the [definition of the coin sensing input lines](#), prove to yourself that connecting these lines to the B inputs as shown in Figure 4-7 will provide the appropriate input bit patterns given in

<sup>3</sup> The pin labeling convention used for this IC is different in the data books supplied by different manufacturers.

**Table 4-2.** Also, prove to yourself that the schematic of Figure 4-7 will function to add the present deposit (*nickel*, *dime* or *quarter*), to the tally input, which has yet to be designed.

**Table 4-2. B and C Input Values as a Function of Coin Value**

Coin Deposit	B3	B2	B1	B0	C0
Quarter	0	1	0	1	0
Dime	0	0	1	0	0
Nickel	0	0	0	0	1



**Figure 4-7.** Adder Design using 74283.

### Task 4-2. Simulate the Adder Hardware Circuit

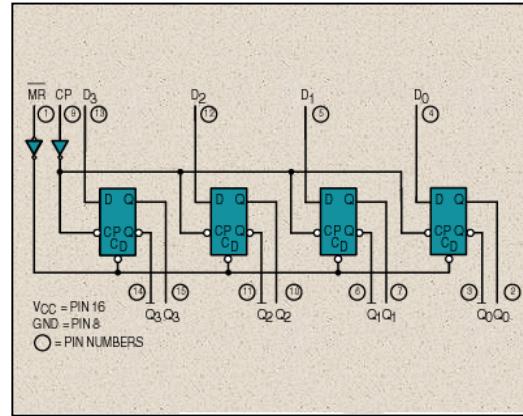
Using LogicWorks™, construct a simulation of the adder design shown in Figure 4-7. (The pin numbers in the diagram are provided to help you when testing this circuit in the hardware laboratory as described in Task 4-5.) You will find LogicWorks™ parts for the 7400 series IC's in the 7400devs.clf library. You get access to this library by using the library selection area of the parts palette. As of this writing, LogicWorks™ has no simulation of the 74283 IC<sup>4</sup>. You will find the 74283 on this CD contained in the LogicWorks™ Demos directory both as a circuit (74283.cct) and as a library (74283.clf).

To test the circuit of Figure 4-7, connect binary switches to the tally input lines and the coin-sensor input lines and verify that the circuit performs the correct addition operations. You do not need to test all 128 possible input combinations. For example, do you need to test that the sum output works correctly when two or three of the coin-sensor input lines are active simultaneously? (Explain your answer in your report.) Test a sufficient number of combinations so that you are convinced that the correct sum is calculated when a nickel, dime or quarter is input into your machine. What is the maximum amount of money that your adder's output can indicate? Will the adder's output be sufficient to represent our revised ultimate price (\$0.75) plus some overage due to excess deposits? Remember to record in your lab notebook all of the measurements that you make. You will be using these measurements to validate the performance of your hardware realization when you perform Task 4-5.

<sup>4</sup> LogicWorks™ version 4.10 (which is a beta release as of this writing) has the 74283 part available. So check for it first in the 7400devs.clf library of your current versions.

**Table 4-3. Pin Definition for 74175 Register.**

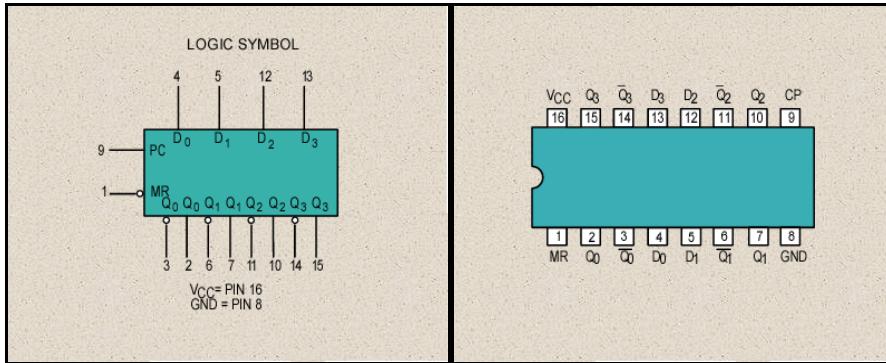
Pin Names	Definition
D0-D3	Data Inputs
CP	Clock Input (Pos.-Edge Triggered)
MR	Reset (Clear) Input (Active Low)
Q0 - Q3	State Outputs
$\overline{Q_0} - \overline{Q_3}$	Complement of State Outputs



**Figure 4-8.** Internal logic diagram of a 74175, quad D flip-flops.

### Register Hardware Design

In our controller design, we will use the 74175, a positive-edge triggered, quad D flip-flop register with active-low reset (clear) control, to store the controller's state, (i.e., sum output of the adder). The internal logic diagram of the 74175 is shown in Figure 4-8 and its pin-out (connection) diagram and logic symbol are shown in Figure 4-9. The definition of each of these inputs is shown Table 4-3. (Note that in this table and in Figure 4-8 and Figure 4-9 we are using the data book [rather than textbook] abbreviation for the flip-flops' clock input, CP, [as described in [Figure 3-4.](#).])<sup>5</sup>

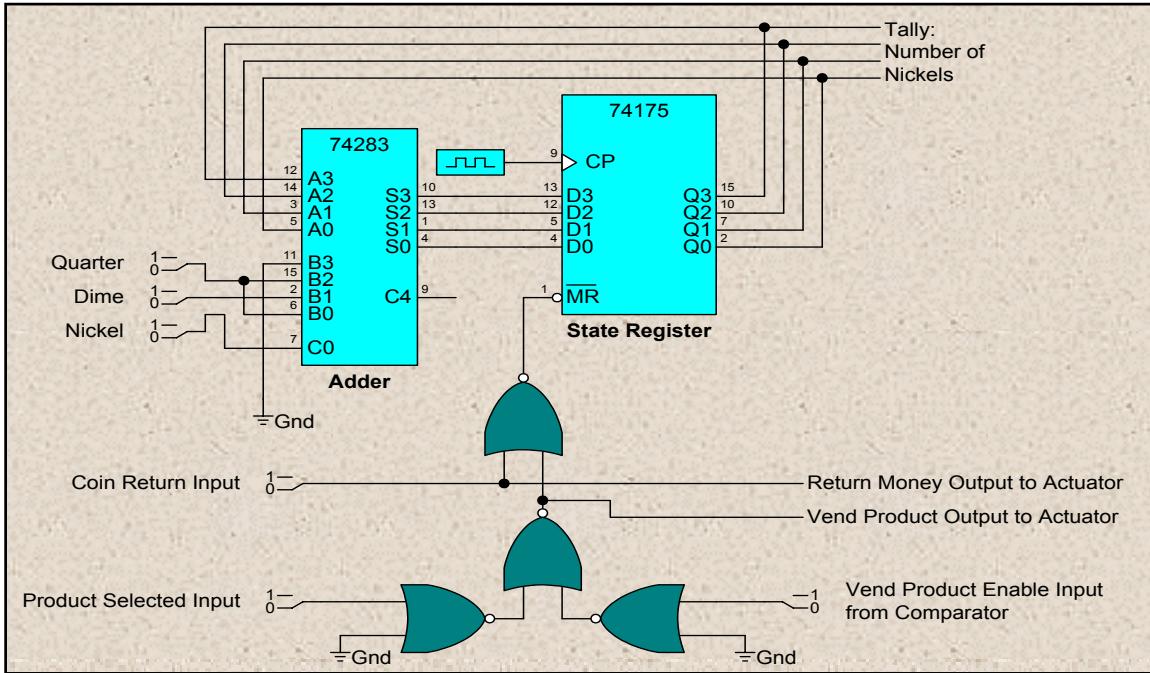


**Figure 4-9.** Logic symbol and connection diagram (pin-out) for a 74175, quad D flip-flops.

### Task 4-3. Simulate the Adder with Output Register & Clear-Tally Hardware

Using the 74175 register IC in the 77400devs.clf library of LogicWorks™, simulate the adder with register circuit shown in Figure 4-10. (Be aware that the 74175 IC schematic symbol shown in Figure 4-10 does not show the complement of the state variables as outputs – while the schematic symbol you get from the LogicWorks library will.) Test this circuit and record the results of your tests in your notebook.

<sup>5</sup> Earlier in this manual we used the typical textbook notation. Here we purposely use the data manual notation to help you familiarize yourself with it. The ability to use this notation will be important as you begin to design your own systems.

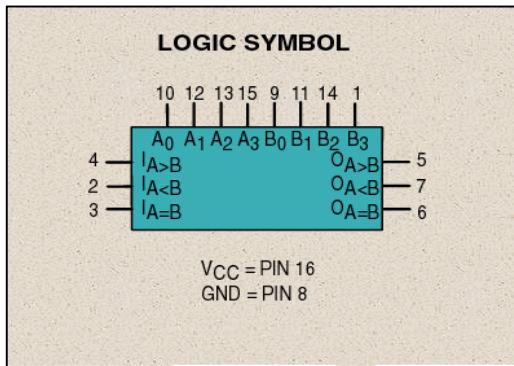


**Figure 4-10.** Adder with register and clear tally hardware.

(When building this circuit in the hardware laboratory (Task 4-5) be sure to disconnect the switches you used in Task 4-2 that drove the tally lines. Also, in the hardware laboratory, be sure to drive the clock (CP) input with a de-bounced switch. When testing this circuit in the hardware laboratory, you will also want to connect the Q3-Q0 state signals to LED's so that you can easily monitor the addition and verify that it is proceeding correctly. Use the records of the test you completed on your simulation to verify that your hardware implementation is working correctly.)

Next use a 7402 IC in from the 7400devs.clf to build the *clear-tally* hardware circuit. Test that it works correctly and record the results of these test in your notebook.

Next, add the clear-tally hardware circuit to your adder/register circuit, test the resultant circuit and record the results in your notebook. Be sure to use sufficient tests (which should include testing the *coin-return*, *product-select* and *vend-product* inputs) to insure that your circuit functions correctly beyond a reasonable doubt. Record in your notebook the results of these tests. (When you build this circuit in the hardware laboratory, you will use the results of the tests you performed on you simulation to verify that your hardware implementation is working correctly.)



**Figure 4-11.** Logic symbol for a 7485, 4-bit magnitude comparator.

**Table 4-4. Pin Definitions for a 7485.**

Pin Names	Definitions
A0-A3, B0-B3	Input Ports
O <sub>A=B</sub>	A Equal to B Output
O <sub>A&gt;B</sub>	A Greater than B Output
O <sub>A&lt;B</sub>	A Less than B Output
I <sub>A=B</sub> , I <sub>A&gt;B</sub> , I <sub>A&lt;B</sub>	Expander Inputs

## Adding the Comparator

A magnitude comparator is a device that compares the magnitudes of its binary inputs and provides outputs indicating whether the inputs are equal or not. The logic symbol and pin definitions are contained in Figure 4-11 and Table 4-4, respectively. The function definition table for the 7485 is shown in Table 4-5.

**Table 4-5. Function Definition Table for 7485, 4-Bit Magnitude Comparator.**

Comparing Inputs				Cascading Inputs <sup>6</sup>			Outputs		
A <sub>3,B<sub>3</sub></sub>	A <sub>2,B<sub>2</sub></sub>	A <sub>1,B<sub>1</sub></sub>	A <sub>0,B<sub>0</sub></sub>	I <sub>A&gt;B</sub>	I <sub>A&lt;B</sub>	I <sub>A=B</sub>	O <sub>A&gt;B</sub>	O <sub>A&lt;B</sub>	O <sub>A=B</sub>
A <sub>3&gt;B<sub>3</sub></sub>	X	X	X	X	X	X	H	L	L
A <sub>3&lt;B<sub>3</sub></sub>	X	X	X	X	X	X	L	H	L
A <sub>3=B<sub>3</sub></sub>	A <sub>2&gt;B<sub>2</sub></sub>	X	X	X	X	X	H	L	L
A <sub>3=B<sub>3</sub></sub>	A <sub>2&lt;B<sub>2</sub></sub>	X	X	X	X	X	L	H	L
A <sub>3=B<sub>3</sub></sub>	A <sub>2=B<sub>2</sub></sub>	A <sub>1&gt;B<sub>1</sub></sub>	X	X	X	X	H	L	L
A <sub>3=B<sub>3</sub></sub>	A <sub>2=B<sub>2</sub></sub>	A <sub>1&lt;B<sub>1</sub></sub>	X	X	X	X	L	H	L
A <sub>3=B<sub>3</sub></sub>	A <sub>2=B<sub>2</sub></sub>	A <sub>1=B<sub>1</sub></sub>	A <sub>0&gt;B<sub>0</sub></sub>	X	X	X	H	L	L
A <sub>3=B<sub>3</sub></sub>	A <sub>2=B<sub>2</sub></sub>	A <sub>1=B<sub>1</sub></sub>	A <sub>0&lt;B<sub>0</sub></sub>	X	X	X	L	H	L
A <sub>3=B<sub>3</sub></sub>	A <sub>2=B<sub>2</sub></sub>	A <sub>1=B<sub>1</sub></sub>	A <sub>0=B<sub>0</sub></sub>	H	L	L	H	L	L
A <sub>3=B<sub>3</sub></sub>	A <sub>2=B<sub>2</sub></sub>	A <sub>1=B<sub>1</sub></sub>	A <sub>0=B<sub>0</sub></sub>	L	H	L	L	H	L
A <sub>3=B<sub>3</sub></sub>	A <sub>2=B<sub>2</sub></sub>	A <sub>1=B<sub>1</sub></sub>	A <sub>0=B<sub>0</sub></sub>	X	X	H	L	L	H
A <sub>3=B<sub>3</sub></sub>	A <sub>2=B<sub>2</sub></sub>	A <sub>1=B<sub>1</sub></sub>	A <sub>0=B<sub>0</sub></sub>	H	H	L	L	L	L
A <sub>3=B<sub>3</sub></sub>	A <sub>2=B<sub>2</sub></sub>	A <sub>1=B<sub>1</sub></sub>	A <sub>0=B<sub>0</sub></sub>	L	L	L	H	H	L

<sup>6</sup> These inputs are provided to allow several comparators to work together to compare numbers with more than 4-bits.

### Task 4-4. Finish the Vending Machine Controller Design

Using the [logic symbol](#) and [truth table definition](#), prove to yourself that the circuit of [Figure 4-12](#) will compare the price with the tally and produce the appropriate outputs. Be sure that you can explain why the connections to the  $I_{A=B}$ ,  $I_{A>B}$ , and  $I_{A<B}$  inputs are as shown in Figure 4-12. Next simulate the circuit of Figure 4-12 and test it. (Record the results of your tests in your notebook so that they can be used to validate the performance of your hardware implementation in Task 4-5.) Set the price to a value of 6 (for \$.30) and verify that your circuit performs the functions consistent with the functional specification. Be careful that the tests you design to verify your controller's operation are sufficient to convince the reader of your report that you have indeed built a controller that meets all design specifications. Are there any design specifications that our design does not meet?

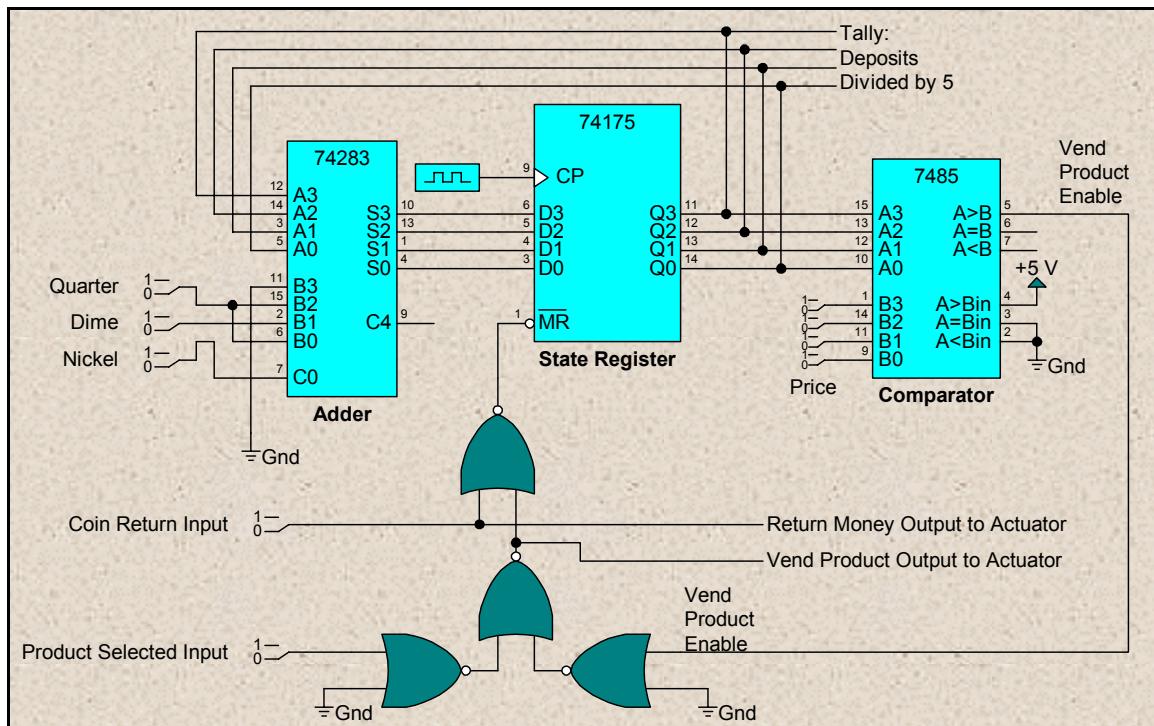


Figure 4-12. Complete vending machine controller design.

One of our functional specifications dealt with resetting the controller after power is applied so that it is in a state corresponding to a deposit of no money. How can this be accomplished with the design of Figure 4-12? What operations would you have the installer perform after plugging in the vending machine so that your controller does not allow product to be sold for less than \$0.30?

Another design specification is that the *vend-product-output-to-actuator* signal be active when the tally is greater than or equal to the price (provided the *product-select* signal is high). Will the vending machine

controller vend product under the equality condition or just when the tally is greater than the price; is our controller being fair, or is it gouging our customers? (You will need to carefully compare the connections made to the  $I_{A=B}$ ,  $I_{A>B}$ , and  $I_{A<B}$  inputs of Figure 4-12 with the information in Table 4-5 to answer this question.)

We have not used the traditional state diagram/transition table/Karnaugh map approach in this design process. Even so, the results of our process can be classified as either a Mealy or a Moore machine. Which is it? Be sure to justify your answer in your report.

### Task 4-5. Implement the Complete Controller Using Hardware

You have built a simulation of the controller using LogicWorks™. The purpose of this simulation was to verify that the design of Figure 4-12 meets the design specification. Once you are satisfied that it does, you will build this circuit in the hardware lab using the results of the tests you performed on the simulation to validate the performance of your hardware implementation.

*Tip:* Before you begin to build your hardware circuit, recall the way you built the simulation: you built one piece (the adder) then tested it; you added on a second piece (the state register) and tested the combination; you built a third circuit (the *clear-tally*), tested it independently, then added it to the partially completed circuit and tested that circuit, etc. We had you complete the circuit in this fashion because this is the way complex circuits are built; pieces are constructed and tested independently, then added to the partially completed design and then that partial design is tested.

There is a reason circuit construction is handled this way: if anything does not perform correctly, the bug is most likely to be found in the piece we most recently added to our partial design, or is caused by the way the piece we added interacts with our partial design. This limits where we need to search for an error and makes debugging relatively easy. If, by contrast, we construct the entire circuit before we begin debugging, we often have little clue as to where to begin our search for errors if the circuit does not perform as expected. When you construct this circuit in the hardware lab, you are likely to have the easiest time debugging if you build and test your circuit in the same way you built your simulation, by sequentially following the instructions of Task 4-2, Task 4-3, and Task 4-5, respectively.

In your report, compare the difficulty of building a hardware realization vis-à-vis a simulation of the circuit shown in Figure 4-12. Based on your observations, what role do you believe circuit simulation should play in the design of new digital circuits?

### Excess Money Deposits

In Task 4-3 we selected the 74283, 4-bit binary adder to add the tally of previous deposits to the current coin deposit. This adder produces a 5-bit output that is capable of representing a decimal value of 31,

which is equivalent to \$1.55 using our nickel-per-increment scheme. This is sufficient to meet the expected eventual price rise of \$0.75; however, there are two problems. First, because only the least significant 4 bits of this value are stored in the state register and fed back to the input to the adder, our addition/register circuit will only perform correctly for tallies less than or equal to \$0.75. That means that there is not room for excess deposits once the price reaches \$0.75. The second problem also deals with this 4-bit limitation of the state register: only the 4 least-significant bits of this stored sum can be supplied to the comparator. This means that the largest tally that the comparator can detect is \$0.75. Once \$0.80 is deposited, the state of the state register becomes 0000B, even though the output from the adder is 10000B.

There are several ways to handle excess deposits. One way is to return all deposited money once the tally exceeds \$0.75. Another way is to make the assumption that users are required to make their selection before they deposit money; then you could automatically vend the product whenever enough money was deposited. Another way might be to use a 5-bit state register (instead of a 4-bit state register) and store the most significant bit (C4) from the adder. To make this work correctly, you would need to redesign your adder and comparator circuits to accommodate 5-bit tally values. The advantage to this approach is that it would allow us to accommodate our initial estimate of the ultimate product price. Other solutions are certainly possible.

### Task 4-6. Account for Excess Money Deposits

Using the schematic of Figure 4-12 as a starting point, produce the schematic of a circuit that will cause the *return-money-output-to-actuator* signal to become high once the tally exceeds \$0.75. Describe another way that you might handle excess deposits and draw a schematic diagram of a circuit that would implement this approach. You are free to use one of the suggestions above or create your own idea. You are also free to make any reasonable assumptions about what inputs the vending machine might provide to you or what mechanisms exist in the vending machine that can be digitally controlled to effect your excess deposit scheme. Be sure to list in your report any assumption you make.

### Task 4-7. Simulate Your Design of Task 4-6

Simulate one of the designs you created in Task 4-6 using LogicWorks™. Test your circuit to insure that it works and record the results of your test. Describe in words in your report how your design accounts for excess money deposits.

## 8-Bit Arithmetic

If you review your design, you will see that it functions according to our functional specification, provided we can accept the revised assumption of a smaller inflation rate or a shorter life span to get an ultimate price of \$0.75. If this was a real application, we would discuss our revised assumption with people

knowledgeable about the vending business and decide whether a \$0.75 deposit limit was acceptable; however, even if this revised assumption is acceptable to experts in the vending business, we see that there are some undesirable limitations to our design; namely, we cannot accurately handle deposits over \$0.75. Also, if we are held to the \$1.00 eventual price rise, (and allowing for some overage in the buyer's deposit), we will need to enhance the design of [Figure 4-12](#) to account for larger deposits, including the eventual acceptance of dollar-bills. Assuming we need to account for dollar-bills, and allowing for a certain amount of overage, we'll need to re-design our circuit to account for deposits somewhat greater than \$1.00, or  $20 \times \$0.05$ . This means our adder must be able to yield valid sums for tallies somewhat greater than  $20D = 10100B$  and our comparator must be able to make comparisons of at least 5-bit numbers as well; hence our controller will need to be redesigned to account for greater precision. Because the 74283 is a 4-bit adder, any adder we build using multiple 74283 IC's will be capable of precision that is a multiple of 4 bits. The next tasks step you through the process of building a controller simulation that uses 8-bit arithmetic.

### Task 4-8. Redesign the Adder to Perform 8-Bit Arithmetic

Using multiple 74283 adder IC's, re-design the adder and state register circuits to perform 8-bit arithmetic and to allow dollar-bills to be input. The information contained in [Table 4-1](#) will help you understand the input/output relationships of the 74283. You may make the same assumptions regarding the dollar bill input as we made about the coin inputs. Show your schematic in your report. Do not build your design.

### Task 4-9. Complete the Controller Design for 8-Bit Precision

Using the results of Task 4-8 and the [logic symbols](#) and [truth table](#) definitions of the 7485, (the 4-bit magnitude comparator), design then draw the schematic circuit of a controller that is capable of a full 8-bit price/tally comparison. To complete this task you will need to understand how the cascading inputs,  $I_{A=B}$ ,  $I_{A>B}$ , and  $I_{A<B}$ , of the 7485 affect its output values. (See [Table 4-5](#).) You will also need to use the  $O_{A=B}$ ,  $O_{A>B}$ , and/or  $O_{A<B}$  outputs of one 7485 (which compares the least significant bits of your 8-bit value) to control the cascading inputs of another 7485, which will compare the most significant bits. Do not build this circuit. You may wish to use LogicWorks™ to help in constructing the schematic of your design. (Using LogicWorks™ here will also help you when you begin the next task.)

The 8-bit arithmetic capability of your controller allows you to accurately account for deposits up to \$12.75. (Your 8 bit adder can accurately account for  $FFH = 255D$  nickels, which corresponds to

$\$0.05 * 255 = \$12.75$ .) Given that the ultimate price of your product is \$1.00, you do not need to account for deposits in excess of \$12.75 in your design<sup>7</sup>.

### Task 4-10. Simulate and Verify the Design of Task 4-8 and Task 4-9

Using the 7400 series IC's available in LogicWorks™, build a simulation of the circuit you designed in Task 4-8 and Task 4-9 and verify that it works correctly. With the benefit of having completed this simulation, assess the relative difficulty of building, debugging, and modifying a simulation versus a hardware implementation. Comment in your report on any advantages you believe exist when you simulate a design before you build it using hardware.

### Report Writing Tips

When you write your report, break it into two major sections: one to deal with the 4-bit controller and one for the 8-bit controller. When discussing the 8-bit controller, you can avoid repeating the same information by referring the reader to the subsections in the 4-bit controller description that discuss the corresponding topics. For example, when discussing the design specification for the 8-bit controller, refer the reader to the design specification of the 4-bit controller, then describe only the differences between these two design specifications.

---

<sup>7</sup> You could try to use the C4 output of the most significant 74283 adder in your design as an input to a circuit that would return money once the deposits exceed \$12.75, but you will find that C4 has a glitch when the number of nickels goes from EFH to F0H. This glitch causes deposits in excess of \$11.95 to be returned – which is an acceptable solution, even if it is not the one anticipated. There are ways of returning excess deposits without encountering this glitch. You are free to explore them if you are so inclined.

# HARDWARE LAB 4: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Top-Down Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Top-Down Description of Work Performed</b>		
<b>The 4-Bit Controller Circuit</b>		
Function and Testing of the Controller Circuit (Incl. Design Spec.)		
Function and Testing of the Adder Circuit		
Function and Testing of the State Register Circuit		
Function and Testing of the Comparator Circuit		
Function and Testing of the Clear-Tally Circuit		
Description of the Excess Deposits Modification		
<b>The 8-Bit Controller Circuit</b>		
Function and Testing of the Controller Circuit (Incl. Design Spec.)		
Function and Testing of the Adder Circuit		
Function and Testing of the State Register Circuit		
Function and Testing of the Comparator Circuit		
Function and Testing of the Clear-Tally Circuit		
Function and Testing of the Excess Deposits Function		
What Was Learned		
<b>Lab Data Sheets</b>		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

**Caveat emptor:** Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used

- |  |
|--|
| <ul style="list-style-type: none"><li>• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.</li></ul> |
| <ul style="list-style-type: none"><li>• Include lab data sheets for all labs. Use lab data sheets to document the lab as you perform it. It saves a lot of time when writing the report.</li></ul>                       |
| <ul style="list-style-type: none"><li>• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.</li></ul>   |
| <ul style="list-style-type: none"><li>• <b>Label</b> all figures or circuits. (E.g. Figure 1. Schematic of . . . , Figure 2. . . , etc.)</li></ul>   |
| <ul style="list-style-type: none"><li>• Refer in the text of your report to all circuits or figures that you include in the body of your report.</li></ul>   |
| <ul style="list-style-type: none"><li>• Use LogicWorks™ to draw your circuit schematics and cut/paste them into your word-processed document.</li></ul>  |

# HARDWARE LAB 4: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Task Oriented

Grading Criteria	Max Points	Points Lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 4-1. Create Your Design Specification		
Task 4-2. Simulate the Adder Hardware Circuit		
Task 4-3. Simulate the Adder with Output Register & Clear-Tally		
Task 4-4. Finish the Vending Machine Controller Design		
Task 4-5. Implement the Complete Controller		
Task 4-6. Account for Excess Money		
Task 4-7. Simulate Your Design of Task 4-6		
Task 4-8. Redesign the Adder to Perform 8-Bit Arithmetic		
Task 4-9. Complete the Controller Design for 8-Bit Precision		
Task 4-10. Simulate and Verify the Design of Task 4-8 and Task 4-9		
What I Learned		
<b>Lab Data Sheets</b>		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

<p><i>Caveat emptor:</i> Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.</p> <ul style="list-style-type: none"> <li>• Title Page: Include <b>Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted</b> in all reports.</li> <li>• Include lab data sheets for all labs. Use lab data sheets to document the lab as you perform it. It saves a lot of time when writing the report.</li> <li>• Include <b>Truth Tables or Function Tables</b> as required to explain how circuits work and as proof of circuit tests.</li> </ul>
--

- **Label** all figures or circuits. (E.g. Figure 1. Schematic of . . . , Figure 2. . . , etc.)
- Refer in the text of your report to all circuits or figures that you include in the body of your report.
- Use LogicWorks™ to draw your circuit schematics and cut/paste them into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

Put an 'X' in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use '5' to indicate that you 'strongly agree', '3' to indicate that you are 'neutral', and '1' to indicate that you 'strongly disagree'. Use 'NA', 'Not Applicable', when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report; however, your **responses will not be graded**. They are for your instructor's information only.

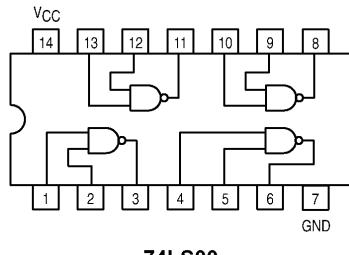
**Table \_\_: Self Assessment of Outcomes for Hardware Lab 4: Vending Machine Controller Design**

After completing the assigned tasks and report I am able to:	5	4	3	2	1	NA
Describe the function of and use a 7485, 4-bit magnitude comparator.						
Describe the function of and use a 74283, 4-bit binary full adder.						
Describe the function of and use a 74175, quad D flip-flop.						
Interpret and use the function definition table of an MSI circuit.						
Simulate a controller for a simple vending machine.						
Simulate a controller for a simple vending machine.						
Design and simulate an 8-bit controller for a vending machine.						
Appreciate the role of MSI circuits in the design of digital circuits.						
Describe the role of circuit simulation in prototype construction.						
Describe some aspects of the design process.						

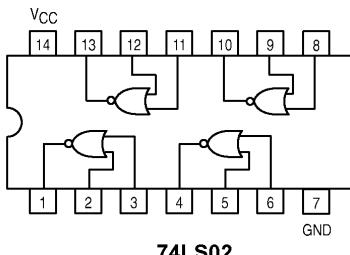
Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

## PIN-OUT DIAGRAMS

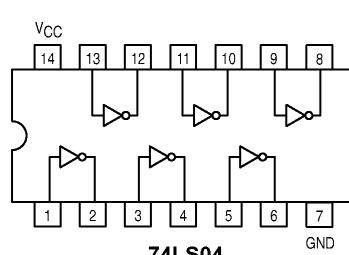
The pin-outs of several TTL devices are given below.



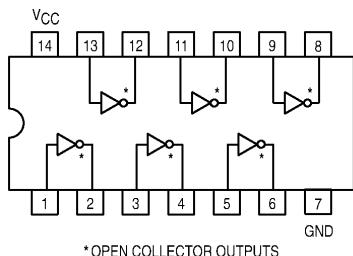
**74LS00**



**74LS02**

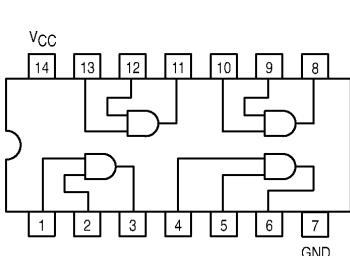


**74LS04**

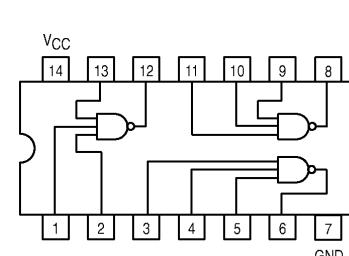


\* OPEN COLLECTOR OUTPUTS

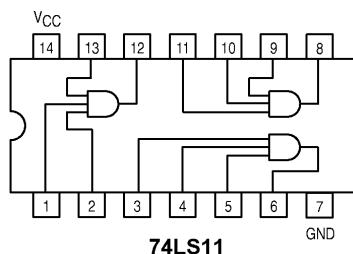
**74LS05**



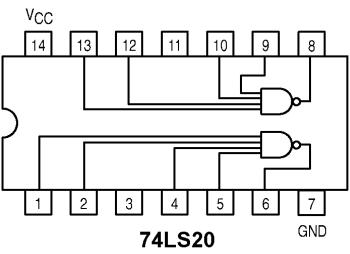
**74LS08**



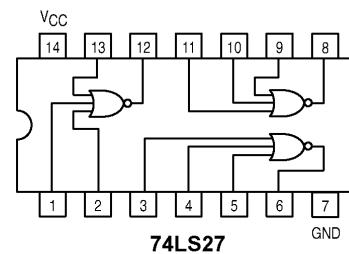
**74LS10**



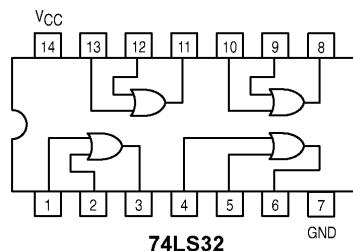
**74LS11**



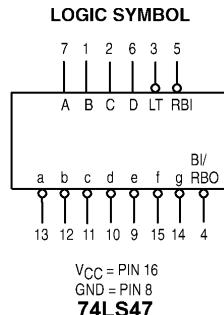
**74LS20**



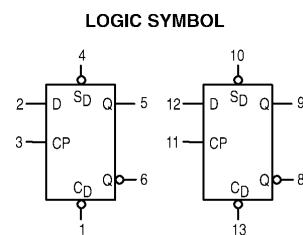
**74LS27**



**74LS32**



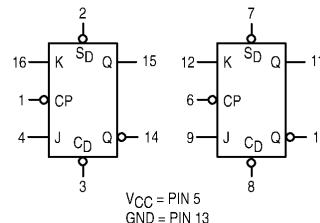
**74LS47**



**74LS74**

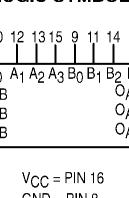
(Copyright of Motorola, Used with permission)

**LOGIC SYMBOL**

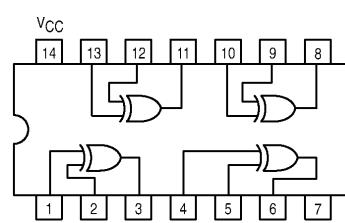


**74LS76**

**LOGIC SYMBOL**

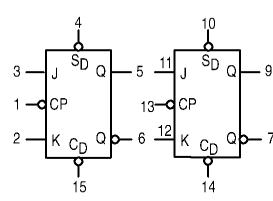


**74LS85**

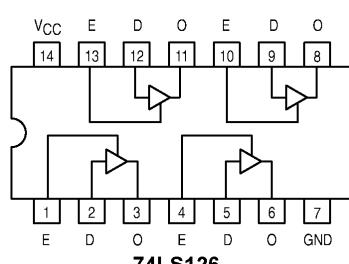


**74LS86**

**LOGIC SYMBOL**

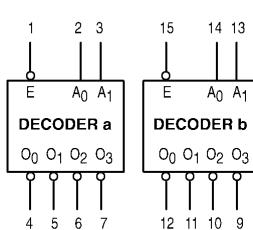


**74LS112**



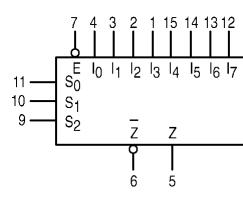
**74LS126**

**LOGIC SYMBOL**



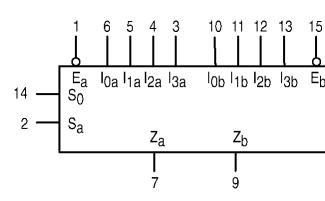
**74LS139**

**LOGIC SYMBOL**



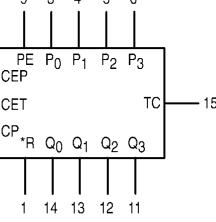
**74LS151**

**LOGIC SYMBOL**



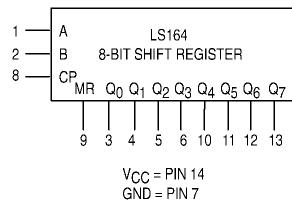
**74LS153**

**LOGIC SYMBOL**



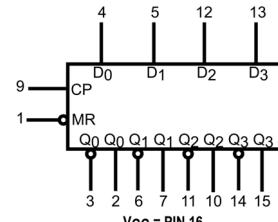
**74LS163**

**LOGIC SYMBOL**



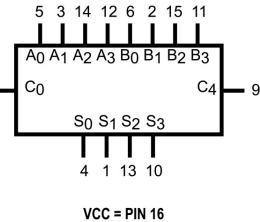
**74LS164**

**LOGIC SYMBOL**



**74LS175**

**LOGIC SYMBOL**



**74LS283**

(Copyright of Motorola, Used with permission)

# HARDWARE LAB 5: CAPSTONE DESIGN PROJECT

---

**Prerequisites:** Before beginning this laboratory experiment you must be able to:

- Design a Mealy and a Moore Machine.
- Use LogicWorks™ to simulate a synchronous machine.
- Build, and debug complex logic circuits.

**Equipment:** Digital Trainer Board, Voltmeter, Logic Probe, LogicWorks™.

**Circuit Components:** The circuit components you will need will depend on your design.

**Objective:** In this laboratory exercise, you will practice designing, simulating, and realizing a simple synchronous machine. You will also practice making assumptions to complete an incomplete functional specification.

**Outcomes:** When you have completed this laboratory exercise you will be able to:

- Use classical design techniques (i.e., state diagrams, state transition tables, and Karnaugh Maps), to design a synchronous sequential machine starting with a functional specification.
  - Make assumptions to complete an incomplete functional specification.
  - Write a complete functional specification.
  - Build, and debug a synchronous sequential machine.
  - Develop reasonable engineering criteria for comparing different designs.
  - Apply engineering criteria to select a ‘best’ design.
- 

## Introduction

In the previous laboratory exercises, we have designed most of the logic circuits for you and specified the measurements you need to make. In this laboratory assignment, you will design two finite state machines, perform simulations of each using LogicWorks™, and build them in the hardware laboratory.

## Task 5-1: Design of Synchronous Sequential Machines

You are to design a Mealy and a Moore synchronous machine that performs according to only one of the functional specifications listed below. (Your instructor will assign the functional specification you are to use.) In your write up, include the complete *design* specification that you developed, i.e., the functional specification listed below along with any assumptions you made to complete the functional specification and any additional assumption you made as the design developed. Include all state diagrams, state transition tables, and Karnaugh maps used in your design. Also include a state definition table that describes in plain English what each state in your machine means and what binary value you have assigned to represent each state.

## Task 5-2: Simulate Both Designs

Using the results of Task 5-1, use LogicWorks™ to simulate each design. Make and report sufficient observations to prove to the reader that your simulations perform according to the completed functional specification of Task 5-1. Demonstrate to a laboratory assistant that your designs meet your completed design specification.

## The Best Design

It is common in practice to find more than one design to meet any given specification. Usually, these designs are evaluated according to some criteria and the best design is selected for fabrication.

## Task 5-3: Determine Criteria and Weighting for Judging Your Designs

To determine a set of criteria for assigning a figure of merit to your designs, you will need to draw on all aspects of your life experience and education, including this course in digital design and the laboratory exercises you have completed. Using your background, create a list of criteria for judging your designs and include it in your lab report. You are free to choose any criteria, provided they are based on reasonable engineering or social values. A criterion that requires judging the expense of construction or complexity is a reasonable engineering value. A criterion that requires judging the ease of use is based both on reasonable engineering and social values. A criterion that says design number two is a better design because two is an even number is based on neither reasonable engineering nor social values.

When using a set of criteria to assess which design is better, a weighting system, implicit or explicit, is always used. Develop and describe the weighting system you will apply to the criteria. The simplest weighting system equally values all criteria. Typically though, some criteria are more important than others. If one of your criteria is more important than another criteria, develop a weighting system that shows the relative weights you will give each criterion. The weights do not have to be numerical, although numerical weights will allow you to arrive at a quantitative answer. If you want to arrive at a qualitative

judgement, you might choose a weighting system such as ‘very important’, ‘important’, or ‘somewhat important’.

### **Task 5-4: Apply the Criteria to Pick the Best Design**

Use the criteria and weighting system you developed and apply it to pick the better of your two designs. Be sure to explain clearly in your report how you have applied your criteria and weighting system.

### **Task 5-5: Build One Design in the Laboratory**

Pick one design to build in the laboratory. It does not have to be the ‘best’ design you picked earlier. Demonstrate to a laboratory assistant that your design works according to your completed specification.

### **Task 5-6: Produce a Report**

Produce a report sufficient to describe your results. Choose a report organization that will allow you to most naturally convey your results to your audience. You may choose to use the top-down approach described in [Top-Down Report Writing Guidelines](#). If you use this approach, the highest level description might include the assignment given, your completed design specification and an overview of your approaches to solving this problem. The next level might describe the two designs you produced, how they functioned, what your criteria was for selecting a preferred design and which design you chose as ‘best.’ The lowest level might include the detailed work you did to create these designs, including state-transition diagram, state tables, Karnaugh Maps, etc. You will also want to include a section in your report describing what you learned. There are many equally effective ways of organizing and writing this report. Good luck!

## **Functional Specifications**

Four different functional specifications for practical problems are included below. Your instructor will select one of these specifications for you to use as the basis of your capstone design project. All information to complete each design may not be specified. **Write down and report any assumptions that you make in your design.**

### **1. Vending Machine Controller**

Design a vending machine controller. There is only one product that is vended by this machine and it costs \$0.15. The controller should allow users to request their money back if they change their minds. The vending machine has a coin mechanism that has a dime and nickel sensor output line; each line becomes high when the appropriate coin is deposited, and is low otherwise. The machine also has an electronic circuit that produces a +5 Volt signal on one line when the coin return lever is depressed. Your controller

will have to produce one digital output signal that is high to enable the vending mechanism when \$0.15 has been deposited and another that is high when the coin return lever is pressed. You may assume that your vending machine will not have to give change. (You may not use the vending machine design of Hardware Lab 4.)

## 2. Win Announcer

There is a game called screwball. Every time a ball is put into play either player 1(P1) scores a point or player 2 (P2) scores a point. A winner is declared when one player is ahead by 2 points. Starting with a score of 0:0 for P1 v. P2, a game might progress as follows:

0:1        1:1        2:1        3:1 (P1 Wins!)

Design a finite state machine that uses information about which player scores a point at each stage of a game to determine when a player wins and which player wins. Your design should include one reset input to initialize your win announcer (this may be a synchronous or an asynchronous input, as you desire). The outputs from your designs should indicate BOTH when a win has occurred and which player is the winner.

## 3. Gas Pump Controller

A petroleum company is interested in reducing pollution in the atmosphere by preventing people from excessively ‘topping-off’ their gas tanks. They realize that when people overfill their gas tanks, gas is spilled and evaporates causing air pollution. To minimize this problem, their pumps currently turn off when the back pressure from the gas tank gets too large while the nozzle switch is activated (i.e., compressed).

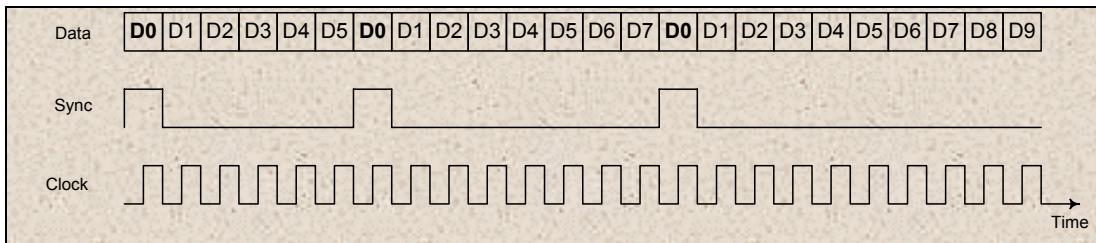
Design a controller that does the following: If the nozzle switch is depressed and the pump pressure sensor indicates low pressure, then allow gas to be pumped. If the pump pressure sensor indicates high-pressure when the nozzle switch is depressed, then turn the pump off. (The system clock pulses every two seconds and you may assume that once the pump shuts off, the pump pressure returns to normal within two seconds.) Once the pressure returns to normal, allow the pump to be controlled again by the nozzle switch. If the pump pressure sensor indicates high pressure a second time, your design will permanently shut down the pump until reset by the gas station attendant. (Note that there are two inputs to your synchronous machine: the nozzle switch position, and the pump pressure sensor indicator. Your synchronous machine has one output that controls the pump.)

## 4. Two's Complementer Circuit

### Introduction

A commonly used method of transmitting digital data is to do it serially. Transmitting data serially means we send data through a transmission line (e.g., coaxial cable) one bit per clock cycle. A second signal, sent

on a second line and called a *Sync* signal, tells us when each sequential data string starts and ends. Sample *Data*, *Sync* and *Clock* signals are shown Figure 5-1.



**Figure 5-1.** Clock, Synch, and Data versus time representation of serial data transmission.

While we could transmit the clock signal along a third line, there are difficulties with this approach. Instead, we build a more sophisticated receiving system that analyzes the received data stream and ‘recovers’ the clock signal<sup>1</sup>.

### Problem Statement

Data is generated remotely from a lander on the Mars surface. *Data* as well as a *Sync* signal is sent back to earth where it is converted into a digital signal. (A designer has already developed a system for recovering the clock signal, so that the above diagram represents the input to our system.) The string length (i.e. number of bits) of each word is unknown. Each word, demarcated by *Sync* signals, is a two’s complement number. Develop a synchronous sequential machine that uses the received data as input and produces as output the negative of the input number (by performing the two’s complement operation on the incoming data.) Your circuit should have two 1-bit inputs: *Data* and *Sync*, as shown in the above figure. Your circuit should have one 1-bit output: the two’s complement of *Data*. An example of the input and output of such a circuit is shown in Figure 5-2.

Recall that the most negative number in any finite bit representation cannot be negated to get a positive number (i.e., in a 4-bit computer the two’s complement operation performed on input 1000 yields 1000, which is not the negative of the input.) What assumption will you make regarding this string?

When designing your circuit it may be helpful to recall the short-cut rule for performing a two’s complement operation on a string of data:

*Starting with the LSB and moving toward the MSB, transcribe all zeros until you reach the first ‘1’; transcribe that ‘1’ then complement every bit after (more significant than) that.*

<sup>1</sup> This recovery process is made more difficult if the data involves long, contiguous sequences of 0's or sequences of 1's. Therefore, we often translate the data into a code that has many transitions between 0 and 1.

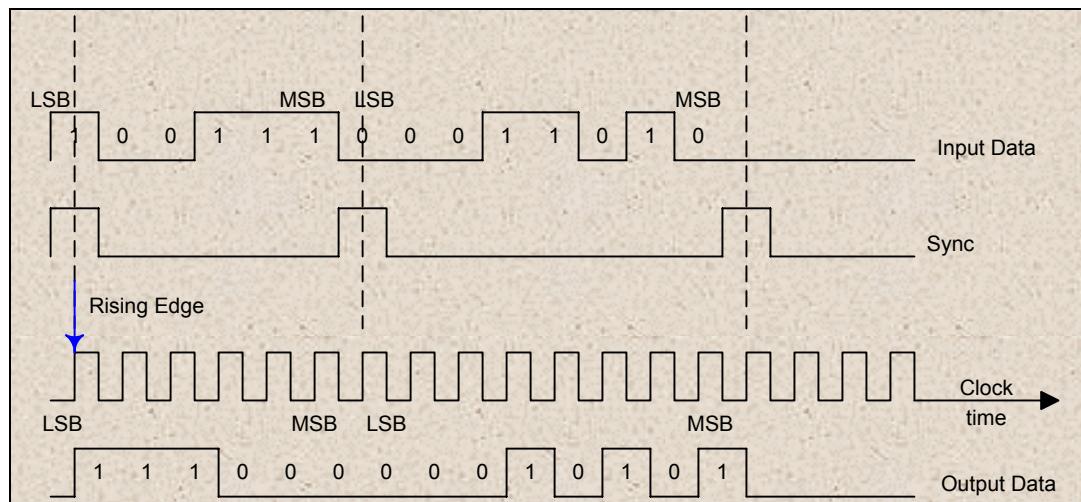


Figure 5-2. Sample output of a two's complementer circuit operating on serial input data.

# HARDWARE LAB 5: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Top-Down Oriented

Grading Criteria	Max Points	Points lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Work Performed</b>		
Design Specification		
Description of Design #1 (Incl. K-Maps, Transition Diagrams, etc.)		
Description of Design #2 (Incl. K-Maps, Transition Diagrams, etc.)		
Engineering Criteria for Picking the Best Design		
Applying Criteria to Pick Best Design		
Demonstrating Simulation of Design #1		
Demonstrating Simulation of Design #2		
Demonstrating One Design in the Hardware Laboratory		
What I Learned		
<b>Lab Data Sheets</b>		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

### Report Writing Reminders:

**Caveat emptor:** Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used

- Title Page: Include **Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted** in all reports.
- Include lab data sheets for all labs. Use lab data sheets to document the lab as you perform it. It saves a lot of time when writing the report.
- Include **Truth Tables or Function Tables** as required to explain how circuits work and as proof of circuit tests.
- **Label** all figures or circuits. (E.g. Figure 1. Schematic of . . ., Figure 2. . . , etc.)
- Refer in the text of your report to all circuits or figures that you include in the body of your report.
- Use LogicWorks™ to draw your circuit schematics and cut/paste them into your word-processed document.

# HARDWARE LAB 5: LAB REPORT GRADE SHEET

Name: \_\_\_\_\_

## Instructor Assessment: Task Oriented

Grading Criteria	Max Points	Points lost
<b>Report Writing</b>		
Complete Title Page		
Organization		
Neatness, Clarity, and Concision		
Statement of Learning Objectives and Outcomes		
<b>Description of Assigned Tasks, Work Performed &amp; Outcomes Met</b>		
Task 5-1: Design of Synchronous Sequential Machines		
Task 5-2: Simulate Both Designs		
Task 5-3: Determine Criteria and Weighting for Judging Your Designs		
Task 5-4: Apply the Criteria to Pick the Best Design		
Task 5-5: Build One Design in the Laboratory		
Task 5-6: Produce a Report		
What I Learned		
<b>Lab Data Sheets</b>		
<b>Self-Assessment Worksheet</b> (The content of the self-assessment worksheet will not be graded. Full credit is given for including the completed worksheet.)		
<b>Lab Score</b>	Points Lost	
	Late Lab	
	Lab Score	

## Report Writing Reminders:

**Caveat emptor:** Before writing your report, check with your instructor to see if grading guidelines different from those stated here are to be used.

- Title Page: Include **Your Name, ID, Class Time, Course Number, Your Instructors Name, Laboratory Experiment Number and Title, and Date Submitted** in all reports.
- Include lab data sheets for all labs. Use lab data sheets to document the lab as you perform it. It saves a lot of time when writing the report.
- Include **Truth Tables or Function Tables** as required to explain how circuits work and as proof of circuit tests.
- **Label** all figures or circuits. (E.g. Figure 1. Schematic of . . . , Figure 2 . . . , etc.)
- Refer in the text of your report to all circuits or figures that you include in the body of your report.
- Use LogicWorks™ to draw your circuit schematics and cut/paste them into your word-processed document.

## SELF-ASSESSMENT WORKSHEET

---

Put an ‘X’ in the table below indicating how strongly you agree or disagree that the outcomes of the assigned tasks were achieved. Use ‘5’ to indicate that you ‘strongly agree’ and ‘1’ to indicate that you ‘strongly disagree’. Use ‘NA’, Not Applicable, when the tasks you performed did not elicit this outcome. Credit will be given for including this worksheet with your lab report. However, your responses will not be graded, they are for your instructor’s information only.

**Table \_\_: Self-Assessment of Outcomes for Hardware Lab 5: Capstone Design Project.**

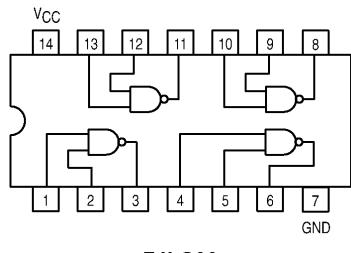
<b>After completing the assigned tasks and report I am able to:</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>NA</b>
Use classical design techniques (i.e., state diagrams, state transition tables, and Karnaugh Maps), to design a synchronous sequential machine starting with a functional specification.						
Make assumptions to complete an incomplete functional specification.						
Write a complete functional specification.						
Build, and debug a synchronous sequential machine.						
Develop reasonable engineering criteria for comparing different designs.						
Apply engineering criteria to select a ‘best’ design.						

---

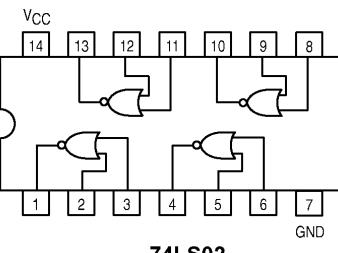
Write below any suggestions you have for improving this laboratory exercise so that the stated learning outcomes are achieved.

## PIN-OUT DIAGRAMS

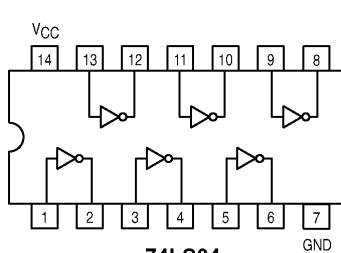
The pin-outs of several TTL devices are given below.



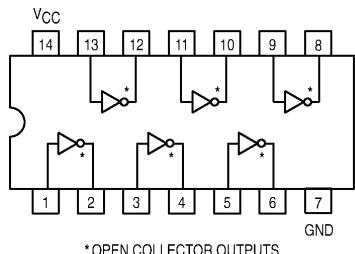
**74LS00**



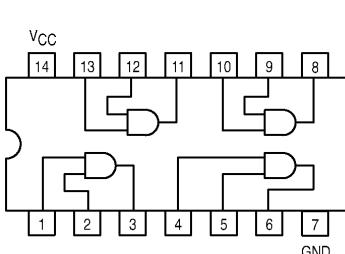
**74LS02**



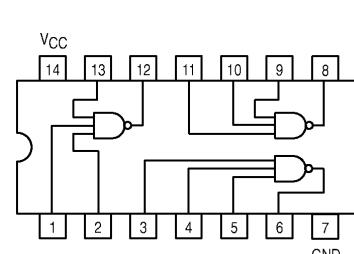
**74LS04**



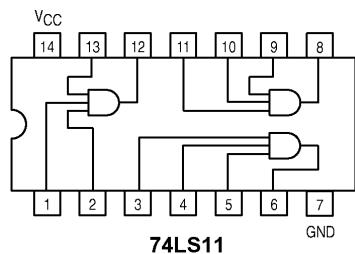
\* OPEN COLLECTOR OUTPUTS  
**74LS05**



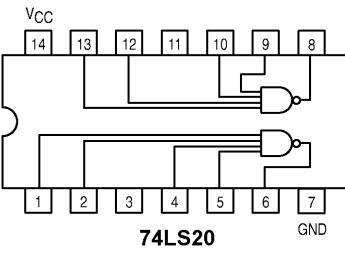
**74LS08**



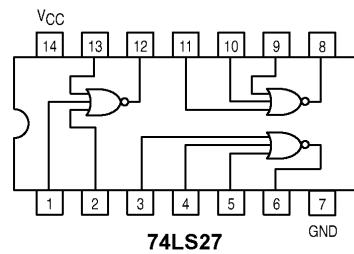
**74LS10**



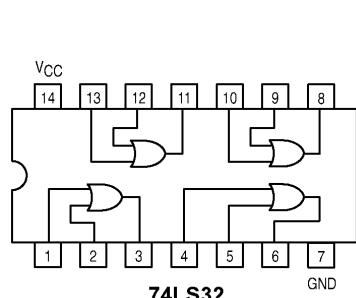
**74LS11**



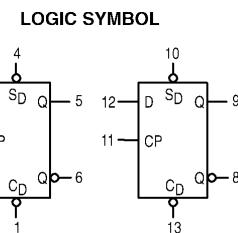
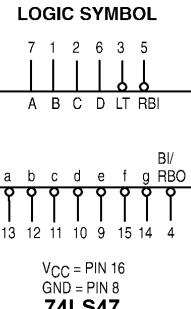
**74LS20**



**74LS27**



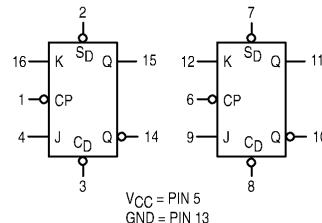
**74LS32**



V<sub>CC</sub> = PIN 14  
GND = PIN 7

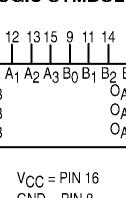
(Copyright of Motorola, Used with permission)

**LOGIC SYMBOL**

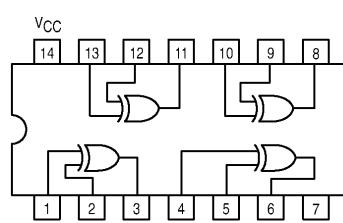


**74LS76**

**LOGIC SYMBOL**

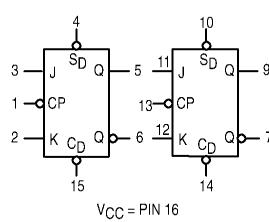


**74LS85**

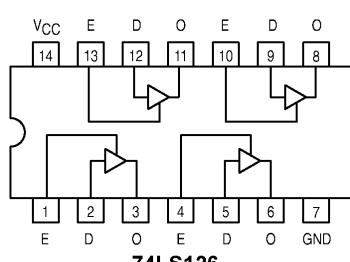


**74LS86**

**LOGIC SYMBOL**

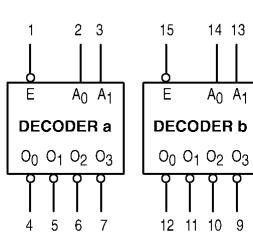


**74LS112**



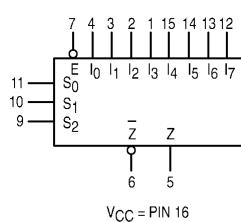
**74LS126**

**LOGIC SYMBOL**



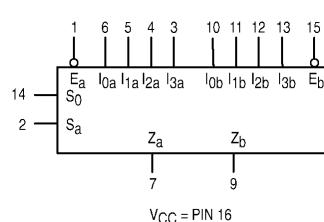
**74LS139**

**LOGIC SYMBOL**



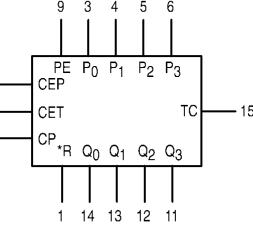
**74LS151**

**LOGIC SYMBOL**



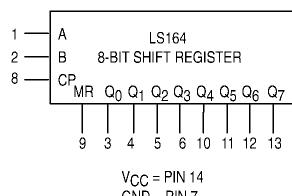
**74LS153**

**LOGIC SYMBOL**



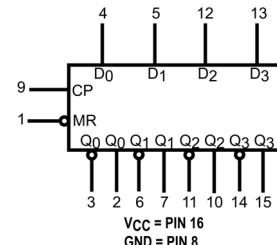
**74LS163**

**LOGIC SYMBOL**



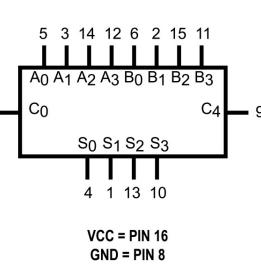
**74LS164**

**LOGIC SYMBOL**



**74LS175**

**LOGIC SYMBOL**



**74LS283**

(Copyright of Motorola, Used with permission)

## APPENDIX A: LOGICWORKS™ SUBCIRCUIT SYMBOLS

### Single Bit Data

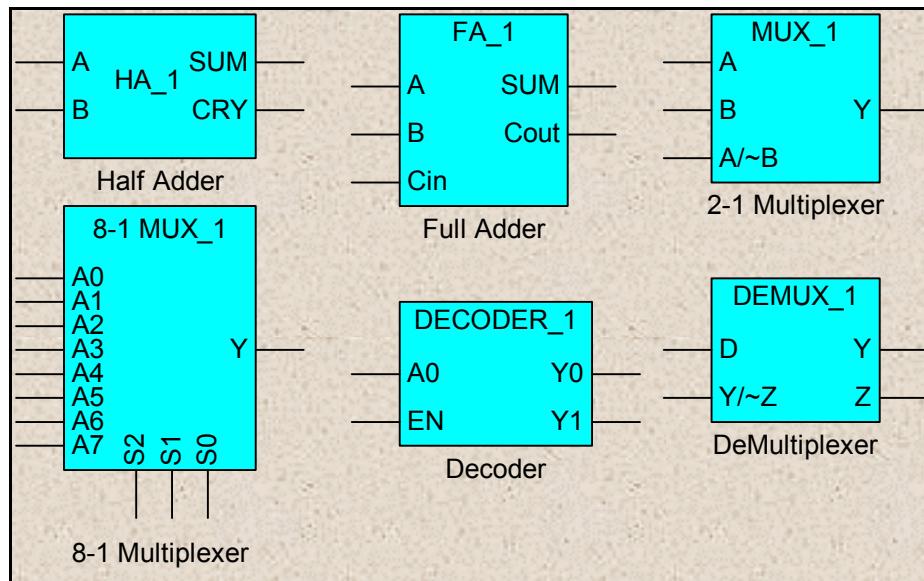


Figure A-1. Devices that operate on one-bit data words.

### Two Bit Data

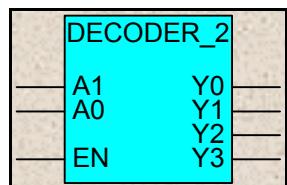


Figure A-2. Devices that operate on two-bit data words.

## Four Bit Data

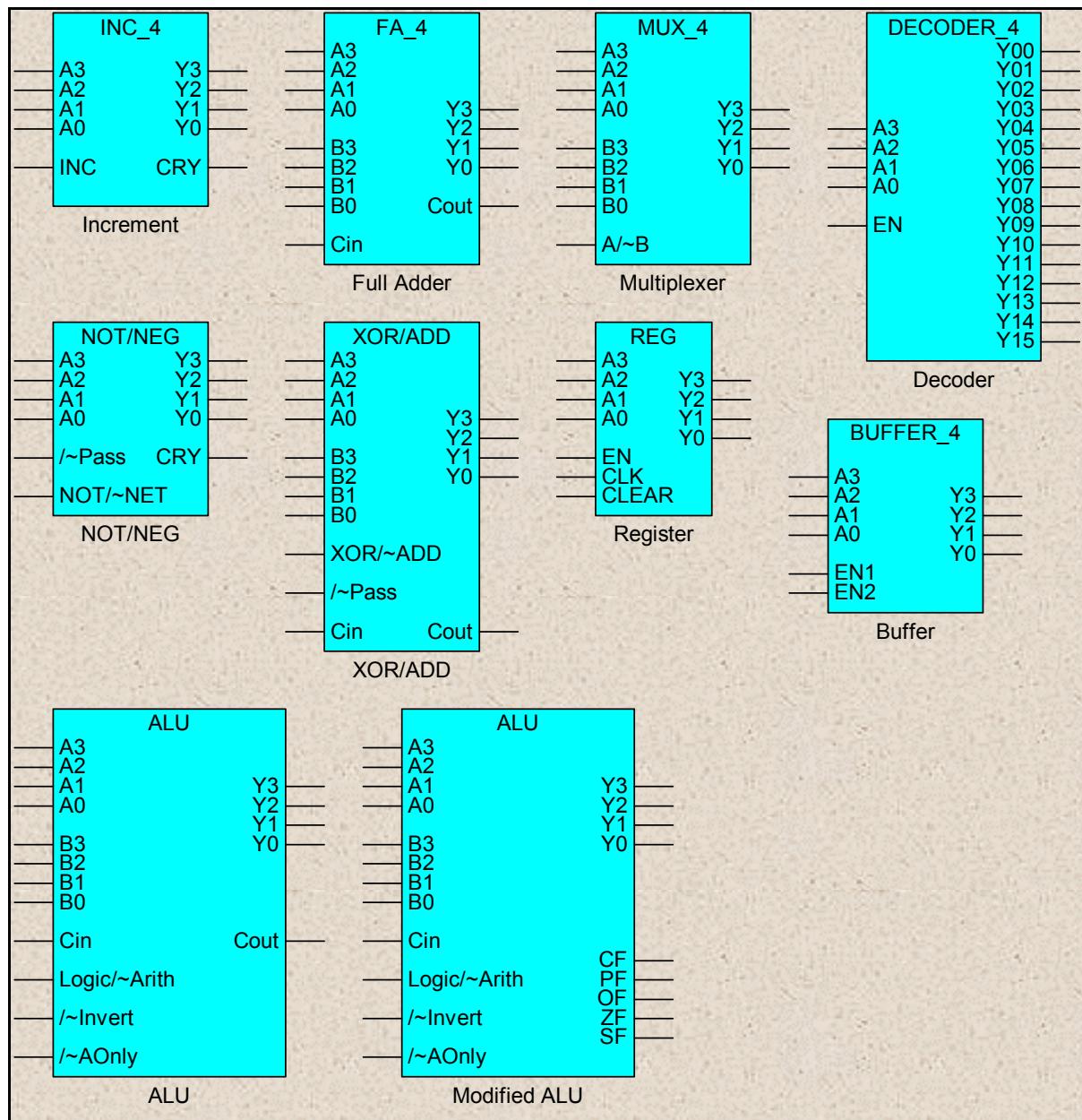
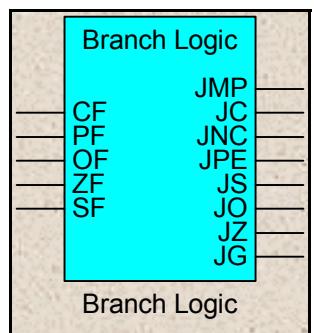


Figure A-3. Devices that operate on four-bit data words.

## Five Bit or More Data



**Figure A-4.** Devices that operate on five or more bit data words.

# APPENDIX B: PROM-BASED SYNCHRONIZED MEALY MACHINES

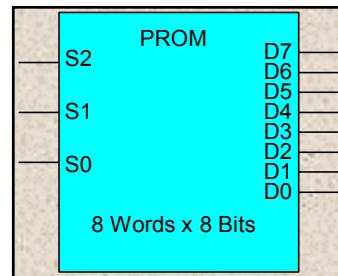
---

## Introduction

In this appendix we will describe how to design a PROM-based synchronized Mealy machine. The next section will take you through the steps necessary to realize functions using a PROM rather than discrete components. Subsequent sections explain how this technique can be applied to simplify the construction of Mealy machines and how the asynchronously changing outputs of a Mealy machine may be synchronized with the clock signal.

## Realizing Combinational Logic Functions with PROM

The combinational logic functions you have realized in these simulation lab exercises have been relatively simple; they have been functions of a few variables and have required a hand full of gates in their realization. The controller we will design will have six input variables and will require 12 output functions. We could build these functions by placing and wiring together primitive logic gates, but even if each function required only ten gates in its realization, we'd still need to wire together 120 gates and debug quite a complex circuit. In industry, circuits with this degree of complexity are often constructed using programmable logic devices (PLD's). Of the PLD's that we could chose for building our functions, we will chose a programmable read-only memory (PROM) because LogicWorks™ has the capability of simulating PROM's. A PROM is a ROM that can be programmed by the user after the manufacturing process is complete. By contrast, the data contained in a ROM is built in during the manufacturing process and can not be altered by the user.



**Figure B-1.** Schematic symbol for an 8-word by 8-bit ROM.

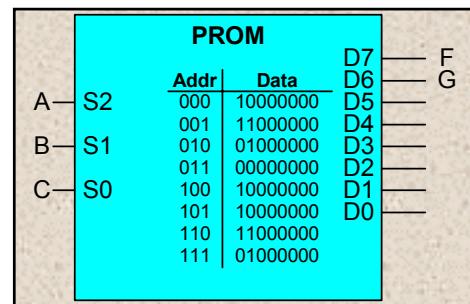
PROM's can be purchased with various numbers of input address lines but the number of output lines is, in general, a power of 2. One symbol we will use in this manual for a PROM is exemplified by the 8-word by 8-bit PROM shown in Figure B-1. The three address inputs S2, S1, and S0, from most to least significant respectively, can uniquely identify up to eight memory locations. The eight output lines give access to the eight bits stored at each addressed memory location, with D0 being the least significant bit.

This PROM, without the aid of external gating, can be used to implement up to 8 functions that share identically the same 3 input variables. Consider the modest goal of implementing the two 3-variable functions, F and G (shown in Table B-1), which share identical input variables. If we assign the A, B, and

C variables to the address inputs S<sub>2</sub>, S<sub>1</sub>, and S<sub>0</sub>, respectively, and assume we want the F and G to appear at the D<sub>7</sub> and D<sub>6</sub> output pins of the PROM, then the data that must be programmed into the PROM to give this result is shown in Table B-2. In this table, only the two most significant output bits, D<sub>7</sub> and D<sub>6</sub>, correspond to the F and G functions respectively; the least significant six bits are specified as ‘don’t cares’ in Table B-2 then arbitrarily assigned to be 0’s when programmed.

**Table B-1. Definition of 3-Variable Functions F and G.**

Variables			Functions	
A	B	C	F	G
0	0	0	1	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	1
1	1	1	0	1



**Figure B-2.** Implementation of functions F and G with an 8-word by 8-bit PROM.

**Table B-2. PROM Contents for Implementing 3-Variable Functions F and G.**

Address Inputs			Stored Data	
S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>	Specified	Programmed
0	0	0	10XXXXXX	10000000
0	0	1	11XXXXXX	11000000
0	1	0	01XXXXXX	01000000
0	1	1	00XXXXXX	00000000
1	0	0	10XXXXXX	10000000
1	0	1	10XXXXXX	10000000
1	1	0	11XXXXXX	11000000
1	1	1	01XXXXXX	01000000

Once the PROM is programmed with the data shown in Table B-2, the functions F and G are realized by connecting the A, B, and C variables to the S<sub>2</sub>, S<sub>1</sub>, and S<sub>0</sub> address inputs respectively and the output D<sub>7</sub> and D<sub>6</sub> are interpreted as the F and G functions as shown in [Figure B-2](#). By inspecting Figure B-2 you can see that once the PROM is programmed, implementing the F and G functions is trivial; all of the effort in building this circuit goes into defining what goes into the PROM and then programming it.

## Implementing a Mealy Machine with a PROM

The technique of implementing functions with PROM can be used in any application; we will use it to implement the functions necessary for realizing a Mealy machine. A Mealy machine, in its most general form, is described by the block diagram of Figure B-3. External inputs along with present-state information are fed into combinational logic to produce the desired output functions and the functions that drive the state flip-flop array. We can rearrange the block diagram of Figure B-3 to get that of Figure B-4, which emphasizes that the next state and output combinational logic functions share identical input variables. Realizing that the output functions and the functions that drive the state flip-flop array have identical input variables, we can implement both the output and next state functions using a single PROM as shown in [Figure B-5](#). Assumed in this figure is that our PROM uses  $2^N$  words, each of which is of length K.

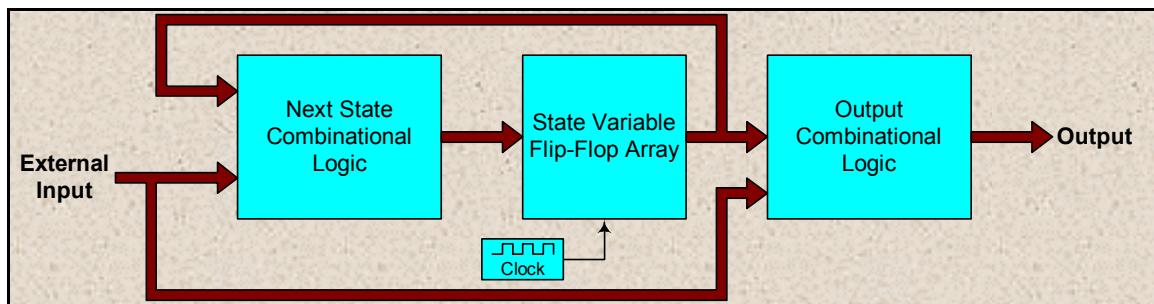


Figure B-3. Schematic block diagram of a Mealy finite-state machine.

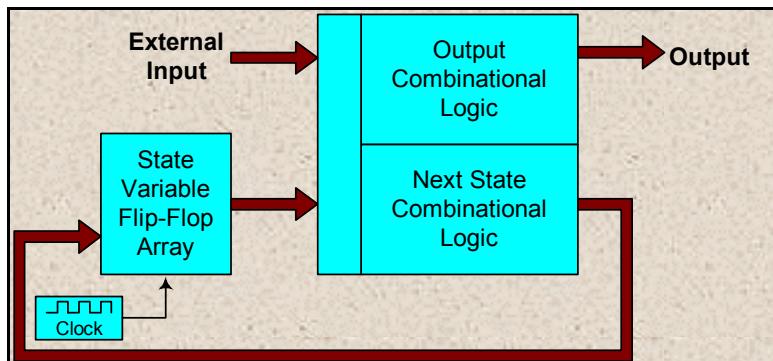


Figure B-4. Rearranged block diagram of a Mealy finite-state machine.

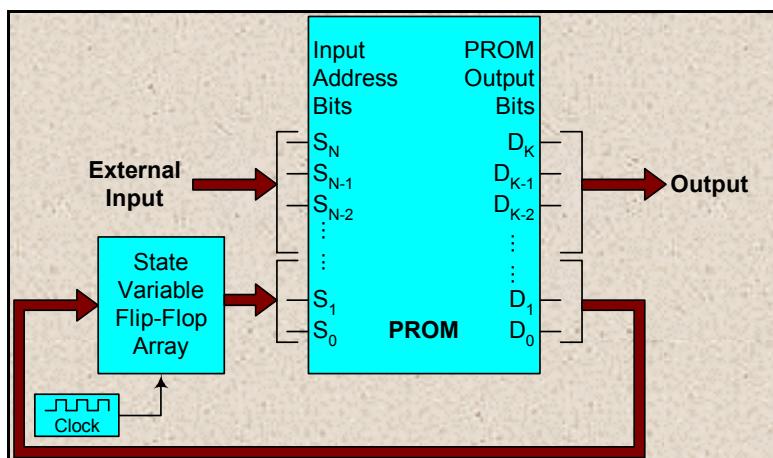
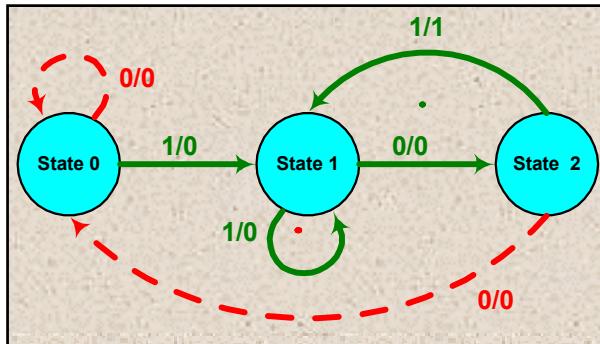


Figure B-5. Block diagram of a PROM-based Mealy finite-state machine.

Before we use this technique for realizing our controller, let's look at a small example. Consider, the state transition diagram of a Mealy machine of Figure B-6. This Mealy machine produces an output of 1 when it detects the key sequence '101'; otherwise its output is zero. (The meaning of each of the states in this figure is listed in Table B-3.) The green annotations accompanying the solid (green) state transitions in Table

B-3 indicate the paths associated with detection of the key sequence<sup>1</sup>; the red annotations associate with the



dashed (red) state transition arrows indicate paths associated with out-of-sequence bits. Using the state binary assignments shown in Table B-3 we can create the state transition table, shown as the nontextured gray portion of Table B-4 (a). This table shows the next-state, output, and D-flip-flop functions (for every combination of present state and input values) that are needed to implement the

**Figure B-6.** State transition diagram for detecting the ‘101’ key sequence.

**Table B-3. State Definition and Assignment for the ‘101’ Key-Sequence-Detecting Mealy Machine.**

State	Meaning	Binary Assignment
0	No part of valid sequence received / reset	00
1	“1” of valid sequence received	01
2	“10” of valid sequence received	10
3	Not Used	11

**Table B-4. State Transition Table and PROM Contents for Implementing the ‘101’ Key-Sequence Detector.**

PROM Address			Next State		PROM Contents			(b)	
S2	S1	S0	QA	QB	D2	D1	D0	PROM	
Input	Present State		Next State		Output	Flip-Flop Inputs			
Y	QA	QB	QA	QB	Z	DA	DB		
0	0	0	0	0	0	0	0		
0	0	1	1	0	0	1	0		
0	1	0	0	0	0	0	0		
0	1	1	X	X	X	X	X		
1	0	0	0	1	0	0	1		
1	0	1	0	1	0	0	1		
1	1	0	0	1	1	0	1		
1	1	1	X	X	X	X	X		

Address (Hex)	Contents (Hex)
0	0
1	2
2	0
3	0
4	1
5	1
6	5
7	0

To implement the sequence detector with a PROM we need to assign each external input and present state variable to a PROM address input and we need to assign the detector’s output and each D-flip-flop function to a PROM output signal line. Let’s make the following assignments:

<sup>1</sup> It is assumed in constructing this transition diagram that the Mealy machine will detect two occurrences of the key sequence ‘101’ if the sequence ‘10101’ is received.

- Assign the external input, Y, to drive S2, the most significant bit (MSB) of the PROM address. (See Table B-4 (a).)
- Assign the state variables QA and QB to drive S1 and S0, the next-to-least significant and least significant bits (LSB's), respectively, of the PROM address.
- Assign the function producing the output of the key sequence detector to the D2 output of the PROM as shown Table B-4 (a).
- Require the functions driving the D input of the A and B flip-flops to be produced by the D1 and D0 outputs, respectively, of the PROM.
- Assign the D3 output to be 0 for every input combination.

Using these input/output assignments (shown in the gold textured header row of Table B-4 (a)), the information that must be programmed at every location in PROM is shown in binary form in the three right-hand columns of Table B-4 (a). If we convert the binary notation used in Table B-4 (a) to hex notation, we get the PROM address/contents shown in Table B-4 (b). Notice that in performing the binary to hex conversion, all ‘don’t cares’ were arbitrarily replaced with 0’s.

Using the PROM contents in hex notation, an 8-word by 4-bit PROM is used to construct the ‘101’-key-sequence detector, as shown in Figure B-7. Verify for yourself that this design is correct.

Notice how the PROM-based design of Figure B-7 appears so easy to build; all of the work goes into deciding what goes into the PROM and programming it. The simplicity of constructing PROM-based designs is even more apparent for designs with more states, inputs and/or outputs, such as the controller design we will implement.

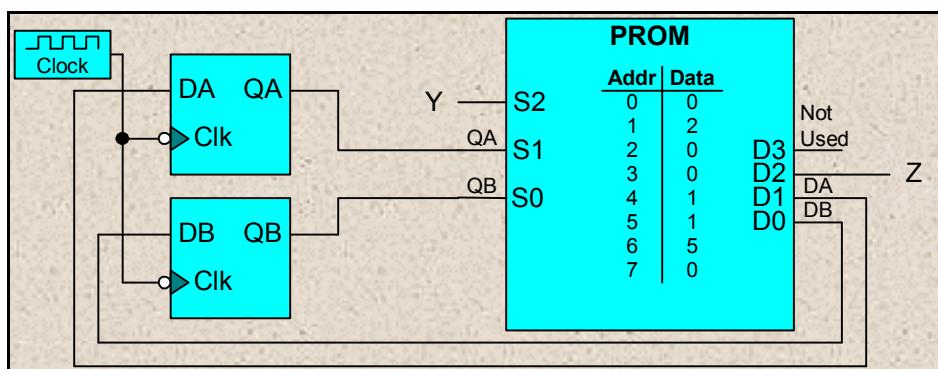


Figure B-7. PROM-based implementation of Mealy machine ‘101’ key-sequence detector.

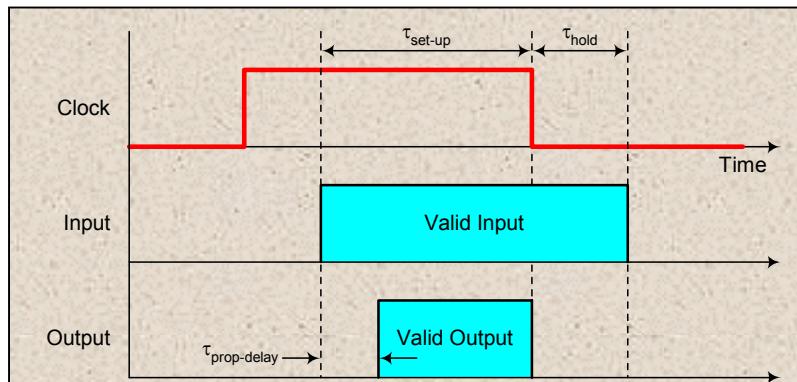
## Synchronizing a Mealy Machine

The advantage that Mealy machines have over Moore machines is that they require the same number or fewer states. The disadvantage of using a Mealy machine is that its output may change when the input changes, regardless of the state of the clock.

For example, consider the key-sequence detector whose state diagram is shown in Table B-3. Assume that the machine resides in state 2. The annotated state-transition arrows leaving state 2 indicate that the output will change from 1 to 0, as the input changes from 1 to 0, regardless of the state of the clock signal.

Evidence that the output changes with the input, without regard to the clock, is also contained (as it must be) in the PROM-based implementation shown in [Figure B-7](#). Assume, again, that the machine of Figure B-7 resides in state 2. While in state 2, the least significant bits of the applied address must be  $(S_1, S_0)=10$ . When the input  $Y=1$ , the address applied to the PROM is  $110B=6H$ , which accesses the memory location containing  $5H$ . This produces an output of 1 at the D2 output. If the input changes to  $Y=0$ , while the machine remains in state 2, the applied address is  $010B=2H$ , which accesses a memory location containing  $0H$ . This produces a 0 at the D2 output; hence while in state 2, as the input changes, the output changes asynchronously.

The observation that the output changes as the input changes implies that we must be careful when we observe the Mealy machine output. We want to observe the output of the Mealy machine for that input which contributes to the change of state. The input that contributes to the change of state is the value that meets or exceeds the set-up time,  $\tau_{\text{set-up}}$ , and the hold time,  $\tau_{\text{hold}}$ , (required by the flip-flops we use in our design) at the time of a clocking event. For a negative-edge-triggered flip-flop, the minimum time span over which the input must be valid is shown in the middle trace of [Figure B-8](#). Assuming that our input is valid for the minimum time span, the time span over which the output is guaranteed to be valid is shown in the bottom trace of [Figure B-8](#). Notice that the time span of the valid output is shorter than the time span over which the input must be held constant. This is because the Mealy machine output becomes valid only after the effect of the input change has propagated through the output circuitry, i.e., the output is delayed by the output-circuit propagation-delay time,  $\tau_{\text{prop-delay}}$ .



**Figure B-8.** Minimum time span over which the Mealy machine input and output must be valid.

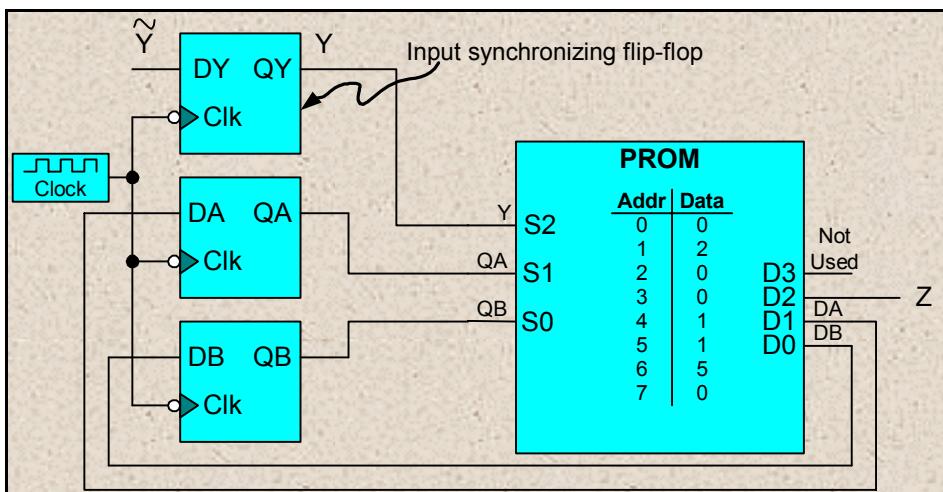
The time span of the valid output is bounded by the negative edge of the clock. Once the negative

edge of a clock signal occurs, the output begins to change quickly; so sampling the output after the negative edge gives unreliable results.

Figure B-8 shows that there are two potential drawbacks to a Mealy machine: the output may be valid for a very short period and (as pointed out in [our earlier discussion](#)) the output may change asynchronously. Is there any way of curing both of these ills? The answer, happily, is yes. If we synchronize the Mealy machine, the output will be valid for an entire clock cycle (less some propagation delay time) and the output will change in synchronism with the clock. There are several ways of synchronizing the output of the Mealy machine with the clock signal. We can use:

- (a) flip-flops to synchronize both the external input and the Mealy output,
- (b) flip-flops to synchronize the output only, or
- (c) flip-flops to synchronize the external input only.

Of these possibilities, we will choose the third; we will use a flip-flop to synchronize only the external input. For our ‘101’ sequence detector, this means sampling the external input,  $\tilde{Y}$ , with the clock signal by using a D flip-flop as shown in Figure B-9. The input synchronizing flip-flop of Figure B-9 causes the input to the PROM, Y, to be synchronized with the clock and to be equal to  $\tilde{Y}$ , but delayed by one clock cycle.



**Figure B-9.** Circuit and timing diagram showing input synchronization of the Mealy machine key-sequence detector.

The delay between the external input,  $\tilde{Y}$ , and the input that drives the Mealy machine, Y, is something that we will have to be aware of when we use this design; so let's examine the origin of this delay. Figure B-10 shows an example of an external-input-data stream,  $\tilde{Y}$ , supplied to the synchronizing flip-flop and the resulting output-data stream—which is the internal Mealy machine input Y. In this figure, it is assumed that  $\tilde{Y}$  changes a short time after the negative clock edge. (We make this assumption because this will be

the case with our controller design.) Observe that any change in  $\tilde{Y}$  will not be observed in  $Y$  until about one clock cycle after it occurs. Notice also in this figure that the output of the Mealy machine,  $Z$ , is synchronized with the internal input,  $Y$ , but is skewed by one clock cycle from the external input,  $\tilde{Y}$ . Since the  $Y$  and  $Z$  signals remain in synchronism, all of the design rules we like to use when designing a Mealy machine still apply for these signals; however when we use such a synchronized Mealy machine we will need to be aware of the time skewing between external input,  $\tilde{Y}$ , and Mealy output  $Z$ .

Cycle #	0	1	2	3	4	5	6	7
Clock	1	0	1	0	1	0	1	0
$\tilde{Y}$	1	0	0	1	0	1	1	0
$Y$	0	1	0	0	1	0	1	0
$Z$							1	

Figure B-10. Example of time skewing between external input,  $\tilde{Y}$ , internal input,  $Y$ , and Mealy output,  $Z$ .

(When we synchronize our Mealy-machine controller, we will find that the external input,  $\tilde{Y}$ , will be our instruction opcode, and the Mealy-machine output will be the control signals that fetch the opcode. Because of this time skewing problem, we will find that we will need to issue the control signals that fetch our opcode one clock cycle before we expect to use the opcode. This will mean that the last operation performed by each instruction we design will be to fetch the opcode of the next instruction to be executed.)

Incorporating the synchronizing scheme of Figure B-9 into the Mealy machine schematic of Figure B-7, while grouping the state flip-flops and input synchronizing flip-flops into registers (labeled as the state register and input synchronizing register, respectively), yields the schematic diagram of Figure B-11. This figure is conceptually identical to our controller. (If you want to look ahead, our controller design is contained in [Figure 5-10](#).) The only difference will be the number of PROM output bits (i.e., size of each word stored in PROM), the number of input address lines, and the data stored in the PROM. Also, when we discuss our controller, we will distinguish between the finite state machine (FSM) portion inside the dotted line of Figure B-11, and the input synchronizing register, which we will call the instruction register.

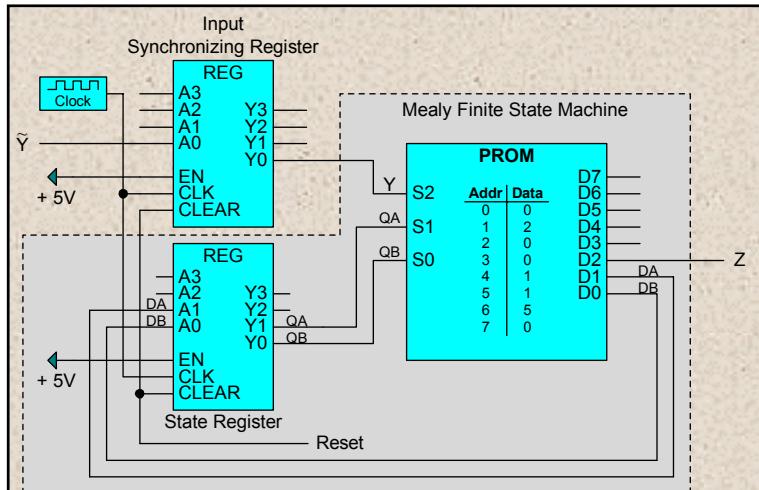


Figure B-11. Synchronized PROM-based Mealy machine for detecting the key sequence 101B.

# APPENDIX C: CONTROLLER DESIGN USING CLASSICAL DESIGN TECHNIQUES

---

## Introduction

In this appendix we will describe how to design the controller for our microprocessor using classical design techniques. By classical design techniques, we mean state transition diagrams, and state transition tables. When we use classical design techniques, any change in the design specification may necessitate completely redoing the design, starting with the first steps in the design process. We will avoid this drawback; for after we complete the design of the controller using classical design techniques, we will spend some time developing a higher-level understanding so that we can endow our controller with more capability, without the need to complete all of the steps required by a classical design process.

Our controller will be a PROM-based synchronized Mealy machine. If you are unfamiliar with PROM-based synchronized Mealy machines, you must first read [Appendix B: PROM-Based Synchronized Mealy Machines](#) before proceeding with the next section.

## Microprocessor Controller Design Using Classical Design Techniques

### The Fetch-Execute Cycle

The controller we will design will be a synchronized Mealy machine with a schematic structure like that shown in [Figure B-11](#). (If you want to look ahead, our controller design is contained in [Figure C-5](#).) The controller we will design will do only three things:

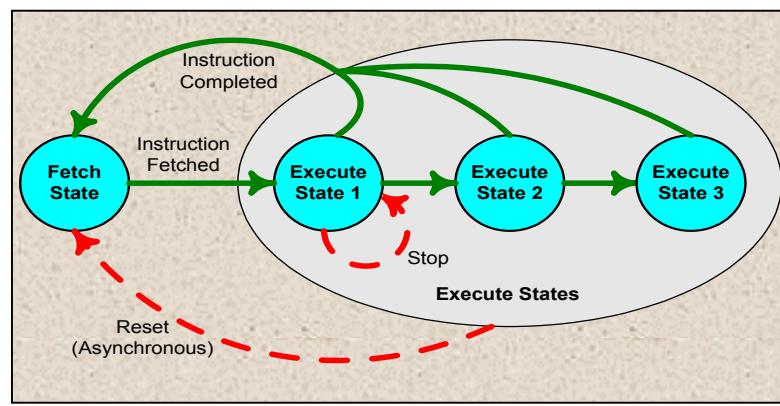
- Fetch an instruction,
- Execute an instruction,
- Halt.

This does not seem like a lot of capability; yet most microprocessor controllers are limited to these three procedures. The capability of the microprocessor arises from the many different instructions it can execute. This cycle of operations is called the fetch-execute cycle and is shown in [Figure C-1](#). When the microprocessor is in the fetch state, it must fetch the opcode of the next instruction to be executed from the externally stored program (in ROM or RAM). Once the instruction is fetched, the microprocessor enters an execute state. An execute state changes various memory units both internal and external to the CPU. Depending on the complexity of the instruction, the processor may pass through several execute states before

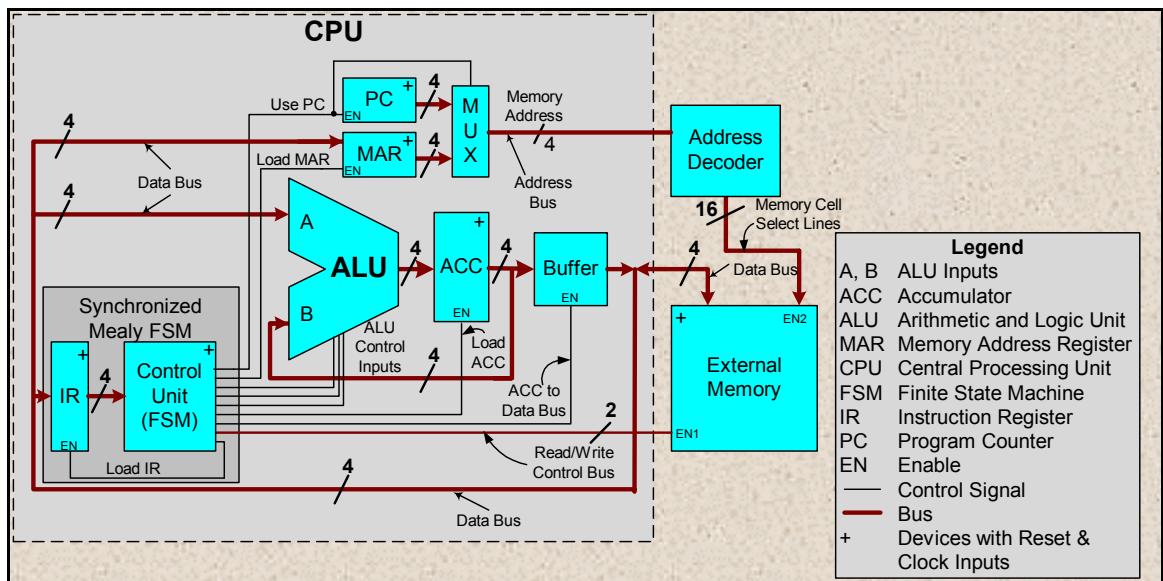
the instruction is completed. After execution is completed, the microprocessor returns to the fetch state, unless it encounters the ‘Stop’ instruction, in which case it enters the Halt state. The ‘Stop’ instruction in our instruction set will be implemented by forcing the controller into an infinite loop; hence, in our implementation, the

Halt state is not a state *per se*, but simply a repetitive indefinite execution of the execute-state 1 shown in Figure C-1. After entering this infinite loop, (and at any time we wish to terminate execution) we simply assert the *Reset* signal to return the microprocessor to the fetch state.

To illustrate how the controller carries out the fetch-execute cycle, we need to understand how the blocks in the architecture of the complete microprocessor, shown in Figure C-2, interact. The clock signals, which load the processor’s registers at each clocking event, are not shown in this figure to avoid the complexity that these lines add to the figure. Instead, the existence of a clock (and reset) input to each register is indicated by a ‘+’ sign in the block. Notice also that, because we are using the Princeton architecture, one data bus is used to move all data from memory to locations within the CPU.



**Figure C-1.** The basic instruction fetch-execute cycle.



**Figure C-2.** Microprocessor architecture.

The controller of the CPU in Figure C-2 is comprised of the Mealy FSM block as well as the synchronizing input register, which is called the instruction register (IR). Figure C-2 hints, for the first time, at how the control unit interacts with the rest of the CPU. If you inspect the lines coming from the control unit, you will see that they touch every block that is part of the CPU, and control memory operations through the *Read/Write* lines. You can see how central the controller is to the operation of our microprocessor.

To see how this controller performs the fetch-execute cycle, let's first look in detail at the control signals that must be active to execute the fetch operation. Then we will look at the signals that contribute to control of the execution part of the fetch-execute cycle.

### Instruction Fetch State

In the fetch state, the controller fetches an opcode from external memory and places it on the data bus. Figure C-3 shows the subset of active control and data lines used to fetch an opcode using the address stored in the PC. To fetch an opcode from memory, the address of the opcode to be fetched must have been stored in the PC by the execution of an earlier instruction. To fetch an opcode using the contents of the PC as an address our controller must:

- Bring the buffer into a high impedance state by bringing to 0 the *ACC to Data Bus* controller line. (Recall that the ACC output is connected to the data bus via this buffer. Without bringing this buffer into a high impedance state, data from the ACC would be driven on the data bus and would conflict with the data being driven onto the data bus from memory.)
- Bring the *Read* line high so that the EN1 input to each memory location is activated.
- Bring the *Write* line low so that no memory location is overwritten by any data driving the data bus.
- Bring the *Use PC* line high. This does two things. First it causes the address-generation mux to select the contents of the PC so that they are made available to the address decoder. The output of the decoder then asserts the EN2 input of the memory cell whose address is stored in the PC. With the EN1 (from the *Read* line) and EN2 inputs both active, the selected memory cell drives its content onto the data bus.

The Use PC line also causes the PC to be incremented on the clock pulse that ends the fetch state. Incrementing the PC at the conclusion of the fetch state allows it to point to the next location in memory, which may contain the next opcode or the operand of the fetched instruction.

- Bring the *Load IR* line high so that the opcode driving the data bus (from the accessed memory location) will be loaded into the IR on the *Clock* pulse that **ends** the fetch cycle.

- Bring the *Load MAR* and *Load ACC* lines low so that data in the MAR and ACC are not overwritten by the opcode driving the data bus.
- Set the ALU control lines, */A Only*, */Invert*, and *Logic/Arith*, to any value since the results of the ALU will not be loaded into the ACC.

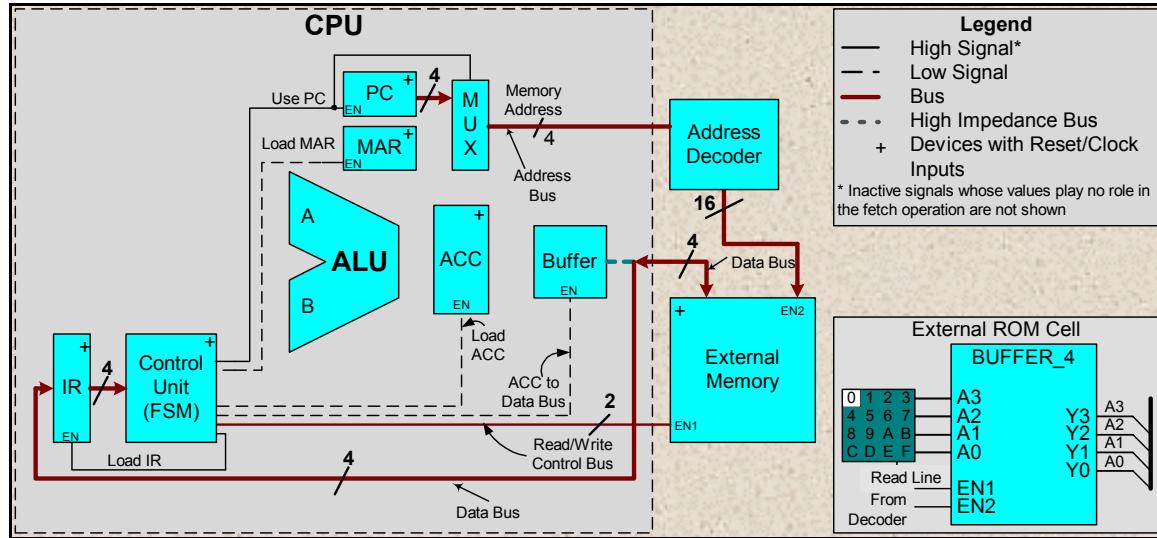


Figure C-3. Fetch state of the microprocessor.

In general, we can think of these control signals as performing two types of actions:

- Route data to the proper location. (For the fetch state described above, data is routed as follows: opcode from memory to the IR, and memory address from the PC to the Address Decoder.)
- Control registers so that they are either loaded or not loaded on the *Clock* pulse that **ends** the current execute cycle. (The PC and IR will be loaded in this case, while the contents of the MAR and ACC will be left unchanged by the **cycle-ending** clock pulse.)

### Instruction Execute State

An instruction execute state is entered at the time a clock pulse first loads the opcode into the IR. During the execute state, the controller asserts control lines consistent with the opcode in the IR. Depending on the opcode fetched during the fetch cycle, the set of control lines asserted will vary greatly; hence, unlike the fetch state described earlier, there is no one set of control lines that is activated during every execute state.

Let's look at one example: the control lines active during the execute phase of the 'Add 5 to ACC' instruction. We will see that this instruction will be stored in memory in the form:

Address	Contents
Q	Opcode for ‘Add to ACC’
Q+1	Operand = 5H

Let's assume in our discussion that the opcode for ‘Add to ACC’ has been fetched and resides in the IR. The result of the fetch operation will have incremented the PC so that it points to the operand 5H in memory. Further, we must assume that the value we wish to add to 5H has already been deposited into the ACC by an earlier instruction. (We'll assume that the previously executed instruction was ‘Load ACC with 3.’) The controller has four major things to do, all of which it can do simultaneously.

- It must make the operand ‘5H’ available to the data bus.
- It must control the ALU to perform the Add operation.
- It must enable the ACC so that it can be loaded with the sum when the Add is completed.
- It must prevent the MAR and IR from being overwritten by the operand on the data bus, 5H.

Let's look at the control lines that will play a role in this execute cycle.

### Make the operand ‘5H’ available on the data bus

To make the operand available on the data bus the following control lines must be active: (Refer to Figure C-4.)

- The *Use PC* line must be high to:
  - Route the PC contents to the address decoder and, hence, activate the EN2 input of the selected memory location.
  - Enable the PC register to be incremented on the *Clock* pulse that **ends** the execute state. (This allows the PC to point to the next opcode in memory during the next clock cycle.)
- The *Read* line must be high to activate the EN1 signal to all memory cells, including the memory cell that contains the operand, ‘5H.’ (Consequently the *Write* line must be low.)
- The *ACC to Data Bus* line must be low to prevent the ACC contents (‘3H’) from conflicting with the operand, ‘5H’, being driven onto the data bus.

### Control the ALU to perform the Add operation

To control the ALU to perform the Add operation, the following control lines must be active:

- $\sim A\_Only = 1$ .
- $\sim Invert = 1$ .
- $Logic/\sim Arith = 0$ .

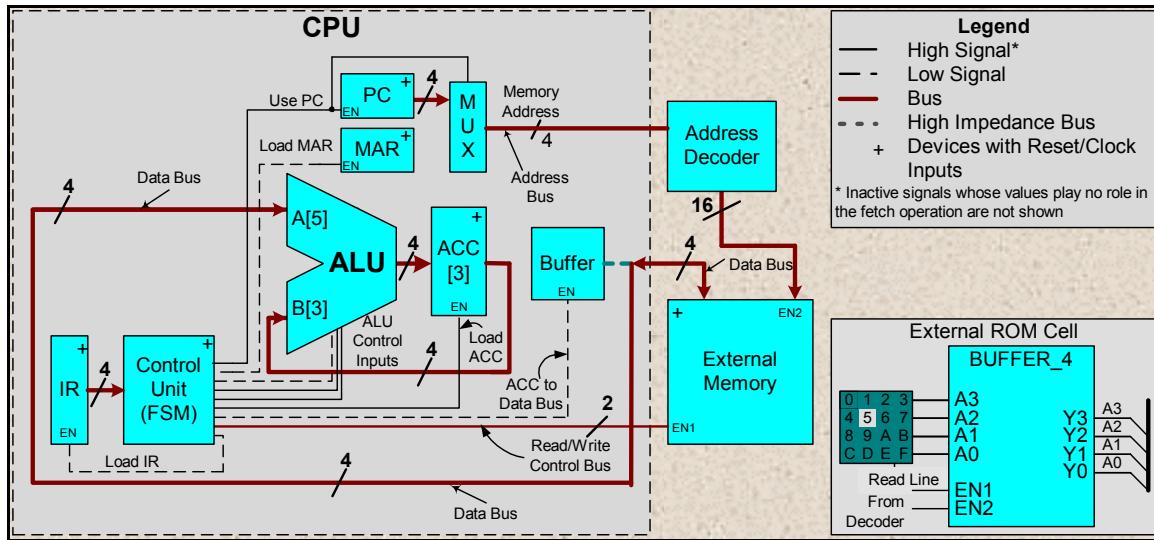
**Load the ACC with the result when the Add is completed**

- The *Load ACC* line must be asserted to allow the *Clock* pulse that **ends** the execute state to load the output of the ALU into the ACC.

## **Prevent the MAR and IR from being overwritten**

- The *Load IR* line must be low to prevent the opcode in the IR from being overwritten by the data bus value on the *Clock* pulse that ends the execute state.
  - The *Load MAR* line must be low to prevent any address in the MAR from being overwritten by the data bus value on the *Clock* pulse that ends the execute state.

The control lines active under this scenario are shown in Figure C-4.



**Figure C-4.** Execute state of the microprocessor for instruction ‘Add 5H to ACC.’

The control lines active under this scenario, shown in Figure C-4, perform two types of actions:

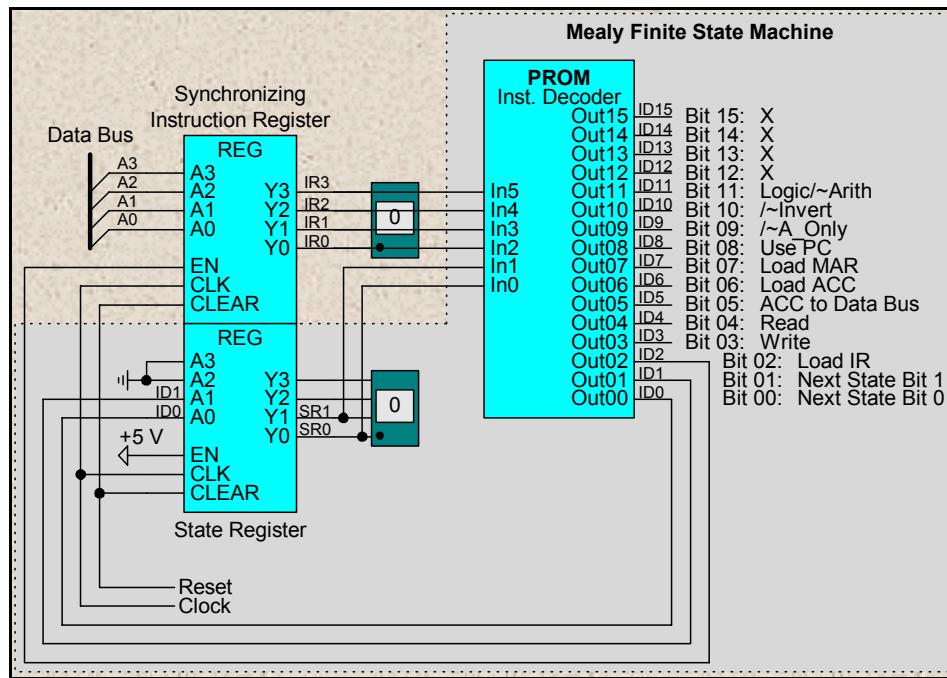
- Route data to the proper location. (For the control line settings described above, data is routed as follows: operand from memory to the ALU A-input port, data through the ALU while performing an Add operation, and memory address from the PC to the Address Decoder.)
  - Control registers so that they are either loaded or not loaded on the *Clock* pulse that ends the current execute cycle. (The PC and ACC will be loaded in this case, while the contents of the MAR and IR will be left unchanged by the cycle-ending clock pulse.)

## Controller Architecture

Taking it on faith that our controller can be based on a synchronized Mealy-machine design similar in architecture to that of [Figure B-11](#), and having defined the controller's inputs and outputs in Figure C-2, we are now in a position to justify the architecture of our microprocessor controller shown in [Figure C-5](#). By

reviewing [Figure C-4](#), you may observe that the external input to the Mealy machine controller will consist of a 4-bit opcode. (Limiting our bus width and opcodes to four bits means that our processor is limited to at most 16 instructions in its instruction set. This is small, but will suffice for our purposes.) The architecture of Figure C-5 shows that this 4-bit input is supplied by the data bus to the synchronizing instruction register.

The outputs from our Mealy machine controller will perform two conceptually distinct chores. One group of outputs will provide control inputs to the various blocks of our microprocessor as shown in [Figure C-2](#). These 10 control signals, listed in [Table 5-1](#) (and shown in Figure C-5), are the ones you have heretofore manipulated manually to control the microprocessor. A second group of outputs from our Mealy machine controller are needed to drive the state flip-flop array to achieve the appropriate next state—given knowledge about the present state and external input. Limiting our controller to four states (as depicted in [Figure C-1](#)) means that we need a two-flip-flop state array to hold the present-state information. Assuming that we use D flip-flops to store the present state, our controller needs two outputs to drive the state flip-flop array, as shown in Figure C-5. Note that in this figure we are implementing the controller functions using a PROM rather than combinational logic. (Notice also that the PROM in this figure has 4 unused outputs. These outputs will be used in Simulation Labs 6 and 7.)



**Figure C-5.** Microprocessor controller circuit.

[Figure C-5](#) is divided into two pieces. Once piece, laid on a flat gray background, is the classical PROM-based Mealy finite-state machine (FSM). In this part of the figure, the output functions of the

Mealy-FSM PROM serve to drive the state flip-flop array and provide output functions that control inputs to the subcircuits of the microprocessor.

The other piece of Figure C-5, the synchronizing IR, serves two functions. One function is to keep the opcode constant for the duration of clock cycles needed by the opcode to fulfill its task. The second function of the synchronizing IR is to ensure that the input to the controller changes in synchronism with the processor clock. Without the synchronizing IR, the controller would perform erratically. For example, if the controller is executing an opcode, as shown in [Figure C-4](#), the operand that is being operated upon is driving the data bus. If the IR is absent, then the controller interprets the value on the data bus as an opcode. This false ‘opcode’ causes the output lines of the controller to change, in turn changing (perhaps) the value driving the address bus, which in turn changes the memory location accessed. This changes the ‘opcode’ the controller receives, which changes the controller’s outputs, etc. Clearly, the synchronizing IR is important to the proper functioning of the controller.

### Designing the Controller Using Classical Design Techniques.

Now that we have defined the inputs that will drive our controller and the outputs that the controller will produce, we can apply classical design techniques to design our controller. The first step in designing our controller is to create a state definition table and assign binary state values to each of the machine’s states. From our previous discussion we know that we need a fetch state. We will also need at least one execute state. To allow for the possibility of creating complex instructions, which will need to activate different sets of control lines on successive clock cycles, we will need additional execute states. A total of three execute states will be sufficient for our controller and will give a total of four states for our machine—consistent with the number of states shown in [Figure C-1](#). We will refer to the operations performed by each of our states as microinstructions; hence, an instruction can be thought of as composed of up to four microinstructions; one fetches the opcode, the other microinstructions assert different sets of control lines to perform the operations required by the instruction. Using these assumptions and our definition for a microinstruction, we obtain the state definitions shown in Table C-1. The binary values assigned to each state in this table are selected arbitrarily.

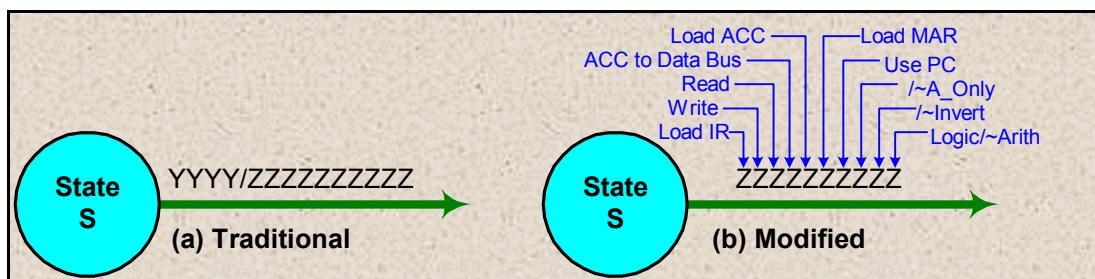
**Table C-1. State Definition and State Assignment a Simple Microprocessor Controller.**

State	Meaning	Binary Assignment
0	Fetch microinstruction	00
1	First-execute microinstruction	01
2	Second-execute microinstruction	10
3	Third-execute microinstruction	11

### Creating the Controller State Transition Diagrams

The traditional state-diagram symbol is shown in Figure C-6 (a) where ‘YYYY’ indicates the four external inputs (4-bit opcode in our design) and ‘/ZZZZZZZZZZ’ indicates the ten outputs the machine produces while in state S (with an input of ‘YYYY’). Because our machine has four input variables, we know there must be 16 arrows leaving every state—one arrow for each different input combination. With four states to our machine, there will be a total of 64 arrows on our state-transition diagram. That is a lot of arrows!

To make our diagram simpler, let’s draw a separate state diagram for each opcode input, or instruction, in our instruction set. We then need only show the output values associated with each state, as shown in Figure C-6 (b), since the opcode input is assumed to be the same for every transition within a single modified state diagram. The Mealy machine outputs we list on the arrow are consistent with the order defined in Figure C-5—but do not include the unused bits, ID15-ID12, or next state bits, ID0-ID1, which drive the state register.

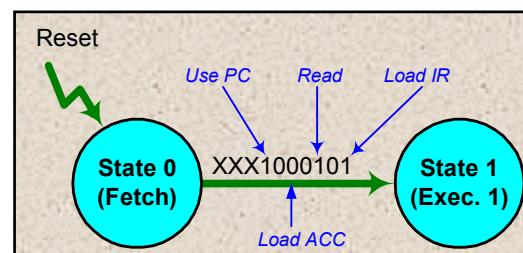


**Figure C-6.** Symbol definition to be used in construction of a state transition diagram.

The next step in creating a state transition diagram, completing a design specification for our controller, means completely defining our instruction set, that is, defining each of the instructions we want our controller to be able to carry out. We are free here to specify any instructions that are within the capabilities of the ALU and architecture we have created. Let’s create a simple one first.

Each instruction we create must pass through the fetch state, state 0, since, as its last function, it must fetch the opcode of the instruction that follows it. The portion of the state transition diagram corresponding to the fetch operation is shown in Figure C-7. From our [earlier description of the fetch state](#), we know that the only control outputs that need to be high during the fetch state are *Read*, *Use PC*, and *Load IR*, as shown in Figure C-7.

Notice in this figure that the ALU control inputs (the left-most 3 bits in this diagram) are ‘don’t cares’. (Since *Load ACC*=0 prevents the ACC from loading the ALU output, the operation performed by the ALU is irrelevant.) Notice also that the asynchronous *Reset* signal takes us to the fetch state. This is required if we want our *Reset* signal to initialize our controller so that its first act will be fetch the opcode of a program entered in ROM, starting a location 0.



**Figure C-7.** State-transition-diagram definition of the fetch state.

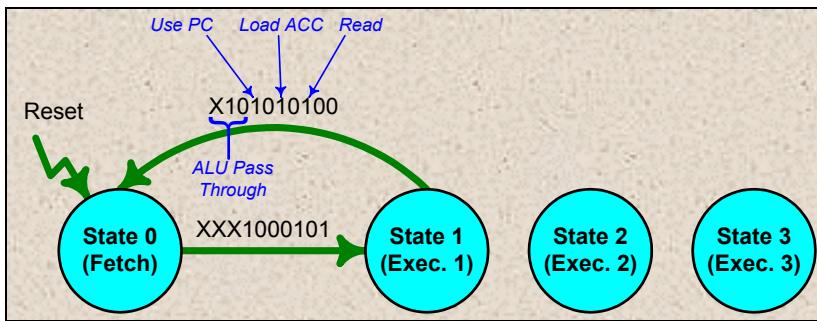
### The ‘Load ACC with [operand]’ Instruction

Let’s create as the first member of our instruction set the ‘Load ACC with [operand]’ instruction. This instruction will appear in program memory in the following way:

**Table C-2. Program Storage Scheme for Instruction ‘Load ACC with [operand]’.**

Address	Contents	Example
Q	Opcode for ‘Load ACC’	0
Q+1	Operand	3

The opcode for ‘Load ACC’ can be chosen to be any number from 0H to FH<sup>1</sup>. Arbitrarily choosing the opcode to be 0H=0000B, ‘Load ACC with 3H’ is stored in program memory as shown in Table C-2. The ‘Load ACC’ instruction requires only two of the possible four machine states; one state to execute the ‘Load ACC’ opcode contained in the IR and another state to fetch the opcode of the next sequential instruction stored in memory. To execute the IR opcode, the controller must use the updated PC (created by the fetch microinstruction) to access the instruction’s operand in memory and load it into the ACC. The instruction must also enable the PC so that it is updated on the *Clock* pulse that concludes execute state 1. Refer to Figure C-2 and prove to yourself that this can be done by bringing the *Read*, *Load ACC*, and *Use PC* control lines high and by putting the ALU in ‘Pass-Through’ mode. Using these control signal specifications, the state transition diagram for the ‘Load ACC’ instruction is shown in Figure C-8. Notice that no arrows are shown entering or leaving states 2 and 3 in [Figure C-8](#); this is because these states are not needed by the ‘Load ACC’ instruction.



**Figure C-8.** State transition diagram for ‘Load ACC with [operand]’ instruction with opcode (input) 0H=0000B.

### The ‘Add [operand] to ACC’ Instruction

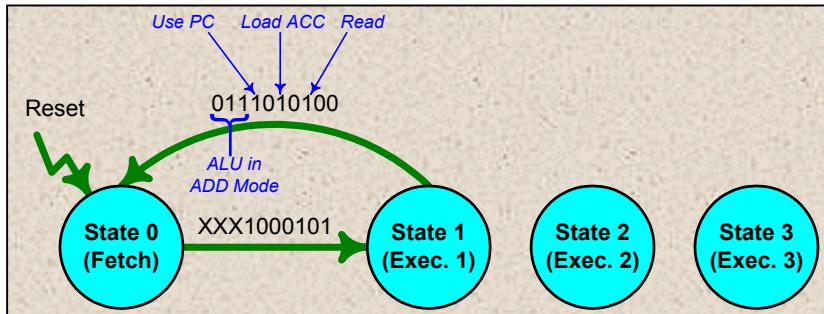
The second instruction we’ll add to our instruction set is ‘Add [operand] to ACC’. When this instruction is used, it will appear in program memory as shown in Table C-3.

<sup>1</sup> It is necessary to have some instruction assigned to the 0H opcode. Exactly why this is the case will become apparent when we discuss the reset operation.

Let's choose 1H=0001B to be the opcode for 'Add [operand] to ACC'. Using this opcode, 'Add 5 to ACC' is stored in memory as shown in Table C-3. 'Add [operand] to ACC' requires using only two of the possible four machine states; one state to execute the 'Add to ACC' opcode contained in the IR and another state to fetch the opcode of the next sequential instruction stored in memory. To execute this opcode, we showed in the [Instruction Execute State](#) section that the following lines must be high:

- *Read*
- *Load ACC*
- *Use PC*
- $\sim A\_Only$
- $\sim Invert$

The state transition diagram for the 'Add 5 to ACC' instruction is shown in Figure C-9.



**Figure C-9.** State transition diagram for 'Add [operand] to ACC' instruction with opcode (input) 1H=0001B.

### The 'Add [3 operands] to ACC' Instruction

So far we have looked only at instructions that require two cycles to execute; one cycle to perform the execute microinstruction specified by the opcode in the IR and one cycle to the fetch of the opcode of the next sequential instruction stored in memory. Consider the instruction 'Add [3 operands] to ACC', which uses all four states. This instruction adds its three operands to a value already stored in the ACC. The instruction is stored in the program portion of the microprocessor's memory as shown in [Table C-4](#).

Let's choose the opcode FH for this instruction. Using this opcode, 'Add 5,2,3 to ACC' is stored in memory as shown in Table C-4. Prove to yourself that the state diagram of Figure C-10 implements this instruction. Observe that constructing the state transition diagram is straightforward; we start in state 0 with a fetch instruction

**Table C-4. Storage Scheme for Instruction 'Add [3 operands] to Acc.'**

Address	Contents	Example
Q	Opcode for 'Add [3 operands] ACC'	F
Q+1	Operand 1	5
Q+2	Operand 2	2
Q+3	Operand 3	3

and cycle through as many states as needed to perform the operations needed by our instruction—deciding which control lines to assert and how many cycles are needed is the only challenge.

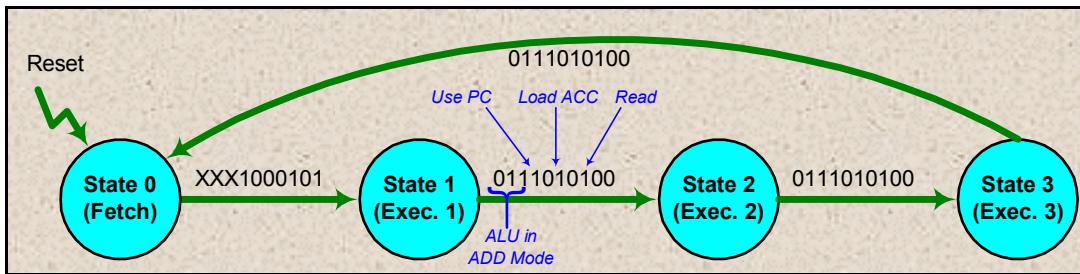


Figure C-10. State transition diagram for 'Add [3 operands] to ACC' instruction with opcode (input) FH=1111B.

### Creating the State Transition Diagram

So far we have created modified state transition diagrams for three instructions:

- Load ACC with [operand]
- Add [operand] to ACC
- Add [3 operands] to ACC

Let's merge these into a single transition diagram using hex (rather than binary) notation. First let's convert the modified state transition diagram of [Figure C-8](#) to a traditional state transition diagram by adding the input values (shown in blue in Figure C-11) to each transition arrow.

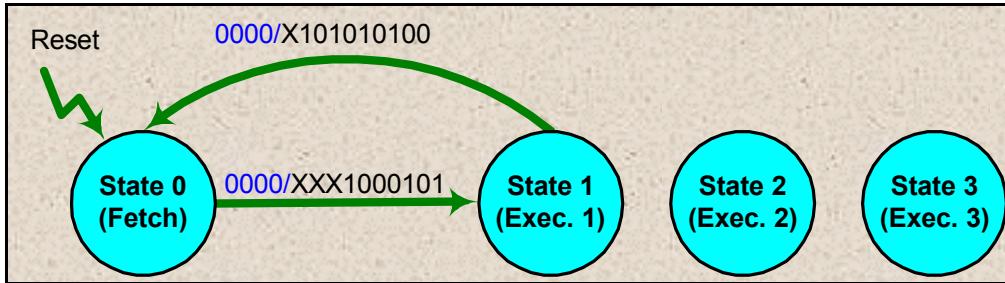
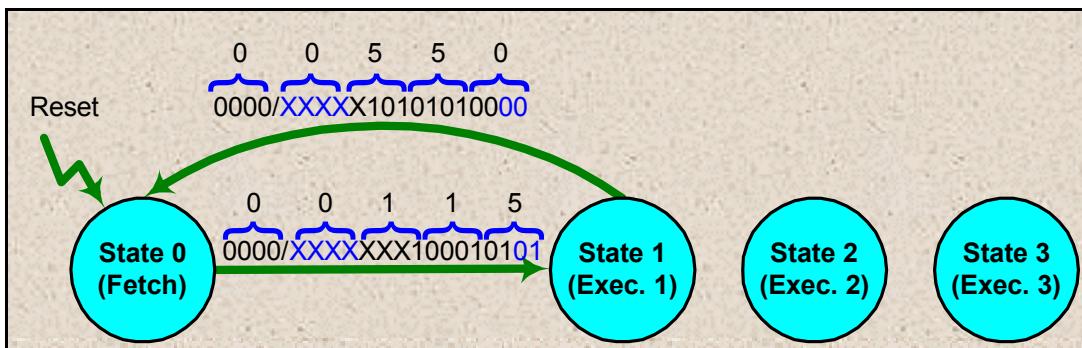


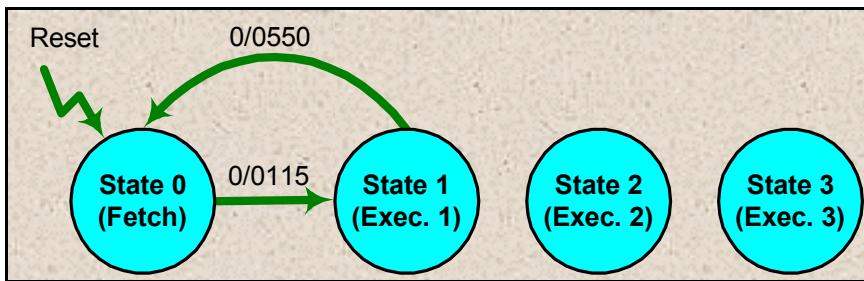
Figure C-11. Traditional state transition diagram for the 'Load ACC with [operand]' instruction.

Next let's modify the traditional diagram further to include the control outputs that drive the state flip-flop array to the appropriate next state. Since the state flip-flop array uses D flip-flops, and since the state representation is the binary representation of the state number, the next state bits are simply the binary value of the state to enter upon a clock pulse. These next state bits are shown in blue in Figure C-12. Notice that this diagram also contains 'don't cares' (shown in blue) for the most significant output bits of our controller PROM, which will not be used in this laboratory exercise, but must be specified (arbitrarily as 0's) when programming the 16-bit PROM we will use.



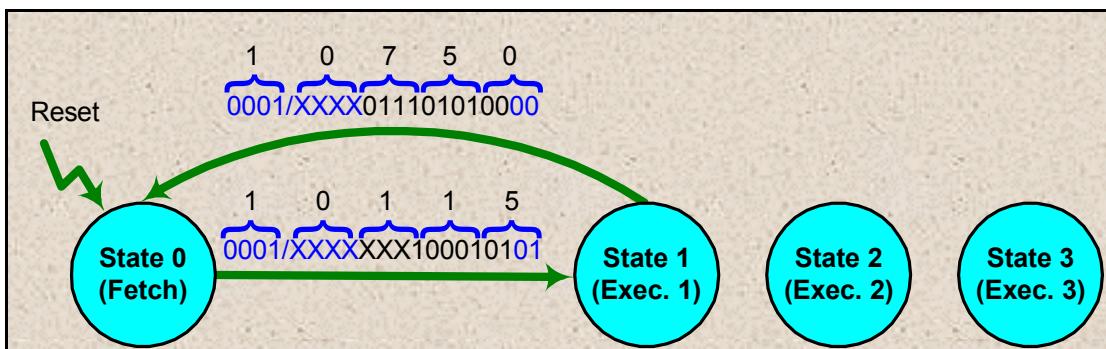
**Figure C-12.** Augmented state transition diagram for the 'Load ACC with [operand]' instruction.

The last step in the conversion process is to replace the 'don't cares' with 0's and convert the notation from binary to hex, as shown in Figure C-12 and Figure C-13.



**Figure C-13.** Augmented state transition diagram for 'Load ACC with [operand]' using hex notation.

Verify for yourself that applying this conversion process to the 'Add [operand] to ACC' and 'Add [3 operand] to ACC' instructions yields the augmented state transition diagrams of Figure C-14 and Figure C-15, respectively.



**Figure C-14.** Augmented state transition diagram for 'Add [operand] to ACC' using hex notation.

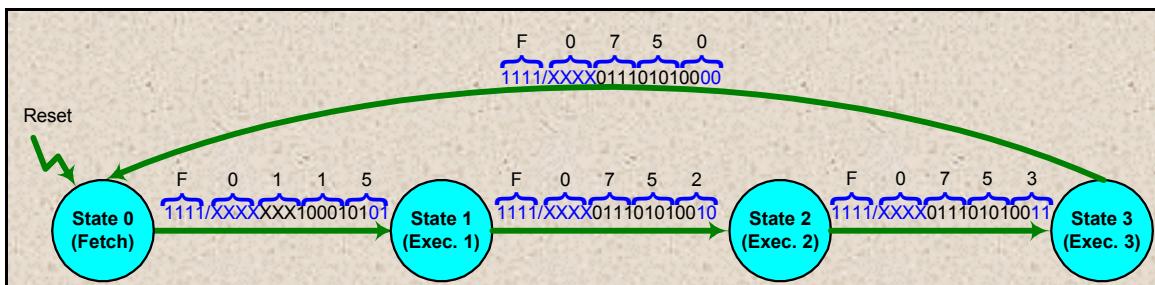


Figure C-15. Augmented state transition diagram for 'Add [operand] to ACC' using hex notation.

Combining these diagrams, we get the augmented state transition diagram of Figure C-16 for the Mealy machine controller that allows our processor to implement three instructions. Notice that the transition arrow from state 0 to state 1 shows the hex input as a ‘don’t care’ since, regardless of the input opcode, the fetch operation is always followed by execute state 1. As we add more instructions to our instruction set, our augmented state transition diagram will continue to grow more complex.

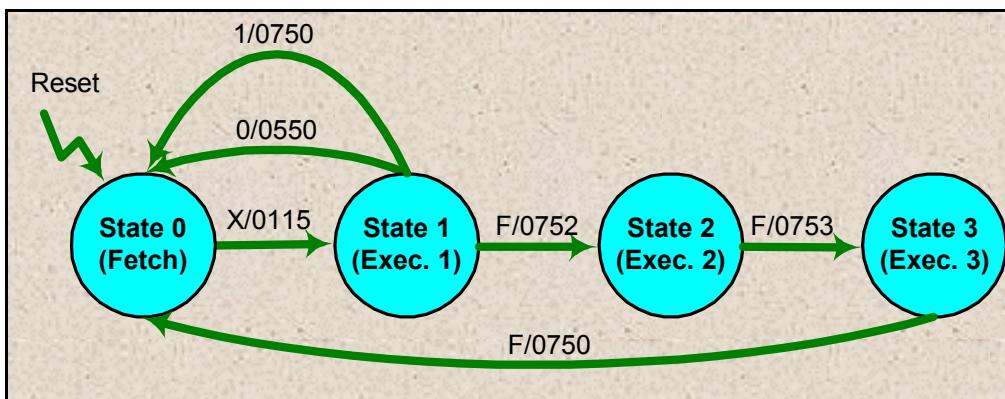


Figure C-16. Augmented state transition diagram using hex notation for 3-instruction instruction set.

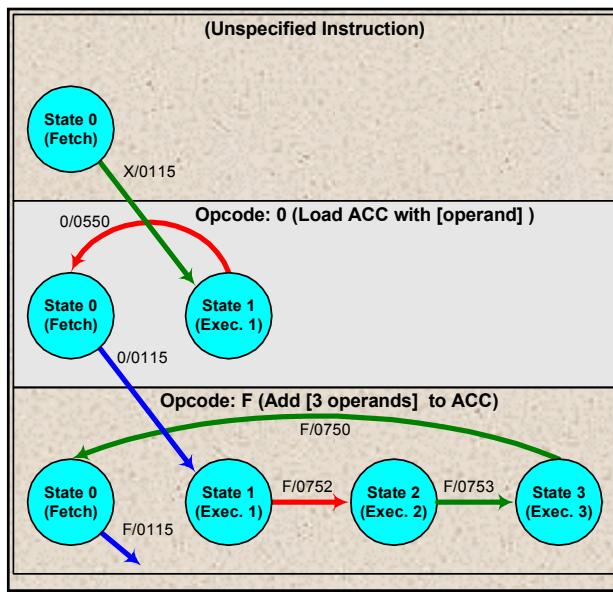
### Revisiting Time Skewing

Before completing the design of our controller, let’s digress for a moment to understand how time skewing, caused by the presence of the synchronizing register, affects the operation of our controller. We predicted in the ‘[Synchronizing a Mealy Machine](#)’ section of Appendix B that because of the presence of the synchronizing register in the controller, the opcode fetched by the fetch microinstruction (state 0) would not be available to the Mealy machine until state 1, the clock cycle after the fetch is executed. We also predicted that this would mean that the last operation of each instruction would be the fetch microinstruction and that this microinstruction would fetch the opcode of the next instruction to be executed. The Mealy state transition diagram of Figure C-16 shows these predictions to be true, but this figure does not show this in an obvious way. To illustrate that these predictions have come to fruition, let’s draw the state transition diagram showing only states entered and branches traversed for the following program,

(Unspecified Instruction)  
Load ACC with 3  
Add 1,3,5 to ACC.

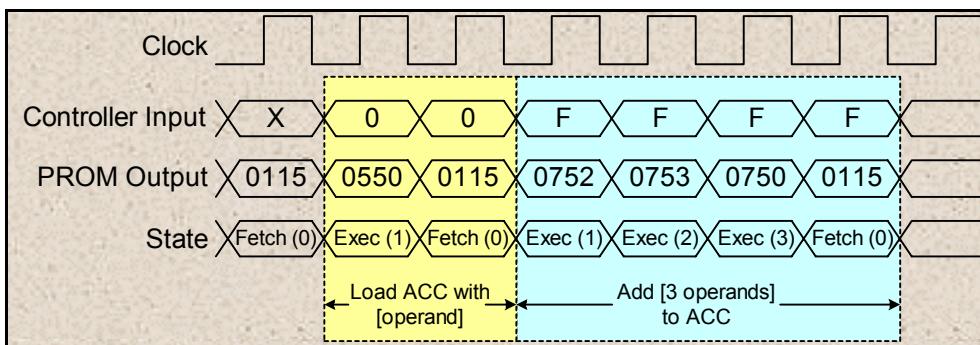
using the information and notation contained in Figure C-16. The resulting state transition diagram is shown in Figure C-17. This diagram is constructed so that all microinstructions/states driven by the same opcode inputs are shown aligned horizontally. This method of constructing the state diagram dictates that a separate copy of each state be generated in the diagram whenever a state has, as its input, a different opcode. Observe that the two predictions we made are demonstrated by this diagram:

- The blue state-transition arrows show that the last microinstruction executed by each instruction is the fetch microinstruction.
- The red state-transition arrows show that the opcode of the fetched instruction becomes available to the Mealy machine during the clock cycle following the fetch instruction.



**Figure C-17.** State transition diagram showing execution of 'Load ACC' then 'Add [3 operands] to ACC.'

The evidence of these observations may also be seen in the timing diagram of Figure C-18, which corresponds to the state transition diagram of Figure C-17, and shows the execution of 'Load ACC with [operand]' and 'Add [3 operands] to ACC'.



**Figure C-18.** Timing diagram for execution of 'Load ACC' then 'Add [3 operands] to ACC.'

### Creating the Partial State Transition Table

With this much of our instruction set defined, we can begin to construct the transition table for our controller by entering, in the gray area of Table C-5(a), the opcodes and next state information for the instructions: 'Load ACC with [operand]' and 'Add [3 operand] to ACC'. Verify for yourself that the partial

state transition table of Table C-5(a) is consistent with the transition diagrams developed so far. Notice in Table C-5(a) that because we will be implementing this machine using a 16-bit PROM, we will need to retain a place in this table for the unused four MSB outputs. Also note that the flip-flop inputs, ID1 and ID0, are defined to be identical to SR1<sup>+</sup> and SR0<sup>+</sup> in order to provide the proper state transition. The other outputs are defined by the labels on the transition arrows of [Figure C-8](#) and [Figure C-9](#).

To uniquely specify the contents of the PROM, we must assign the most and least significant input and output bits to be consistent with the order shown in [Figure C-5](#), then translate the entries in Table C-5 (a) into the entries in Table C-5 (b). This means we must assign the unused PROM outputs to MSB's (Out 15-Out12), the bits Out11-Out02 to the *Logic/Arith*, */~Invert*, */~A\_Only* . . . *Load IR* signals (in the order shown in [Figure C-5](#)), and ID1, ID0 to Out1, Out0. We arrive at the contents of the PROM Table C-5 (b) by arbitrarily assigning all X's to be zeros. Notice that the address of the PROM contents as specified in Table C-5 (b) uses a modified hex code. In the modified code, the least significant character is the hex equivalent of the 2-bit binary present-state value, In1 and In0. The most significant hex character is the hex equivalent of the opcode, In6 through In2. We use this modified hex code, because it allow us to more easily associate the address being applied to the PROM with the operation the PROM is carrying out. Once you have completed the translation from transition table entries to PROM contents, you may notice that the PROM contents could have likewise been read directly from [Figure C-16](#) had you chosen to do so.

**Table C-5. Partial State Transition Table and PROM Contents for Mealy Machine Controller.**

ROM Address					Output				(b)	
In6-In2	In1	In0			Out15-Out12	Out11-Out2	Out1	Out0	ROM	
Input (Binary)	Present State (Binary)		Next State (Binary)		Unused Output (Hex)	Output (Binary)	Flip-Flop Inputs		Address (Mod Hex)	Contents (Hex)
Opcode	SR1	SR0	SR1 <sup>+</sup>	SR0 <sup>+</sup>	Z	Z	ID1	ID0		
0000	0	0	0	1	X	XXX1000101	0	1	00	0115
0000	0	1	0	0	X	X101010100	0	0	01	0550
0000	1	0	X	X	X	X	X	X	02	0000
0000	1	1	X	X	X	X	X	X	03	0000
0001	0	0	0	1	X	XXX1000101	0	1	10	0115
0001	0	1	0	0	X	0111010100	0	0	11	0750
0001	1	0	X	X	X	X	X	X	12	0000
0001	1	1	X	X	X	X	X	X	13	0000

**Developing a Higher Level Understanding Using the Microinstruction Definition Table**

By continuing to use state transition diagrams you can build your own instruction set and define the contents of the PROM necessary to implement your instruction set. But the state transition diagrams are so simple to construct that we could just as easily fill in the transition table by inspection. Further, there is an alternative method of presenting the transition table data that we will find easier to work with and which will promote a higher-level understanding of our design. Compare the information presented in the first data line of Table C-5 with the information contained in the microinstruction definition table contained in Table C-6. Both of these tables contain the same information, but most people find Table C-6 easier to work with because it clearly identifies the values needed from each of the control lines. Justify to yourself that the information contained in Table C-6 through Table C-8 is consistent with that contained in Table C-5.

**Table C-6. Fetch Microinstruction Definition Table.****Fetch Microinstruction**

<b>Output Bit</b>	<b>Control Line</b>	<b>Present State Bits</b>			
		00	01	02	03
1-0	Next State Bits	01			
2	Load IR	1			
3	Write	0			
4	Read	1			
5	ACC to Data Bus	0			
6	Load ACC	0			
7	Load MAR	0			
8	Use PC	1			
9	/~A_Only	X			
10	/~Invert	X			
11	Logic/~Arith	X			
12	X	X			
13	X	X			
14	X	X			
15	X	X			
	Hex Equivalent	0115			

**Table C-7. ‘Load ACC with [operand]’ Microinstruction Definition Table.**

**Load ACC with [operand]: Opcode 0H**

<b>Output Bit</b>	<b>Control Line</b>	<b>Present State Bits</b>			
		00	01	02	03
1-0	Next State Bits	01	00		
2	Load IR	1	0		
3	Write	0	0		
4	Read	1	1		
5	ACC to Data Bus	0	0		
6	Load ACC	0	1		
7	Load MAR	0	0		
8	Use PC	1	1		
9	/~A_Only	X	0		
10	/~Invert	X	1		
11	Logic/~Arith	X	X		
12	X	X	X		
13	X	X	X		
14	X	X	X		
15	X	X	X		
Hex Equivalent		0115	0550		

**Table C-8. ‘Add [operand] to ACC’ Microinstruction Definition Table.**

**ADD [operand] to ACC: Opcode 1H**

<b>Output Bit</b>	<b>Control Line</b>	<b>Present State Bits</b>			
		00	01	02	03
1-0	Next State Bits	01	00		
2	Load IR	1	0		
3	Write	0	0		
4	Read	1	1		
5	ACC to Data Bus	0	0		
6	Load ACC	0	1		
7	Load MAR	0	0		
8	Use PC	1	1		
9	/~A_Only	X	1		
10	/~Invert	X	1		
11	Logic/~Arith	X	0		
12	X	X	X		
13	X	X	X		
14	X	X	X		
15	X	X	X		
Hex Equivalent		0115	0750		

**Creating the ‘Stop’ Instruction Using the Microinstruction Definition Table**

Let’s define the ‘Stop’ instruction (with opcode 2H=0010B) using the microinstruction definition table directly, instead of the state transition diagram. This instruction will cause the microprocessor to cease executing new instructions. More precisely, we will force our controller into an infinite loop by creating a microinstruction whose next-state bits are the same as the present-state bits, as shown in Table C-9<sup>2</sup>. Once our controller enters this loop, the only way we have of regaining control of our processor is to use the *Reset* control line. The PROM output values for all bits other than the next state bits are shown as ‘0’s’ in Table C-9. Many of these values can be specified as ‘X’s’. Can you determine which of these values may be defined as ‘don’t cares’? (There is more than one answer to this question.)

**Table C-9. Microinstruction Definition Table for the ‘Stop’ Instruction.**

		Present State Bits			
Output Bit	Control Line	00	01	02	03
1-0	Next State Bit	01	01		
2	Load IR	1	0		
3	Write	0	0		
4	Read	1	0		
5	ACC to Data Bus	0	0		
6	Load ACC	0	0		
7	Load MAR	0	0		
8	Use PC	1	0		
9	/~A_Only	X	0		
10	/~Invert	X	0		
11	Logic/~Arith	X	0		
12	X	X	X		
13	X	X	X		
14	X	X	X		
15	X	X	X		
Hex Equivalent		0115	0001		

**Defining the Contents of our PROM**

We have already seen how to determine the contents of the PROM using the transition diagram approach. If we use the microinstruction definition table directly, then defining the contents of the PROM may be handled as follows: First we need to concatenate the microinstruction definition tables for each instruction we have defined, starting with opcode 0. This has been done in Table C-10 for the 3-instruction instruction set (‘Load ACC with [operand]’, ‘Add [operand] to ACC’, and ‘Stop’) we have defined thus far. (This table does not include the ‘Add [3 operands] to ACC’ instruction.) Note that in this table the control

<sup>2</sup> There is no need for specifying the PROM values for present state 0 in this table since this state will never be entered under design conditions. Can you justify why this statement is true?

signals for all opcode-state combinations that are not used are set to zero. The 16-bit hex numbers in the bottom row of this table are the control signal values that will be stored in our PROM.

**Table C-10. Input-Output Data for Limited Instruction Set PROM.**

Instructions		Load ACC				Add to ACC				Stop			
Opcode		0				1				2			
Pres. State		00	01	02	03	00	01	02	03	00	01	02	03
Description	Pin Number												
Next State Bit	1-0	01	00	00	00	01	00	00	00	01	01	00	00
Load IR	2	1	0	0	0	1	0	0	0	1	0	0	0
Write	3	0	0	0	0	0	0	0	0	0	0	0	0
Read	4	1	1	0	0	1	1	0	0	1	0	0	0
ACC to Data Bus	5	0	0	0	0	0	0	0	0	0	0	0	0
Load ACC	6	0	1	0	0	0	1	0	0	0	0	0	0
Load MAR	7	0	0	0	0	0	0	0	0	0	0	0	0
Use PC	8	1	1	0	0	1	1	0	0	1	0	0	0
/~A_Only	9	X	0	0	0	X	1	0	0	X	0	0	0
/~Invert	10	X	1	0	0	X	1	0	0	X	0	0	0
Logic/~Arith	11	X	X	0	0	X	0	0	0	X	0	0	0
X	12	X	X	0	0	X	X	0	0	X	X	0	0
X	13	X	X	0	0	X	X	0	0	X	X	0	0
X	14	X	X	0	0	X	X	0	0	X	X	0	0
X	15	X	X	0	0	X	X	0	0	X	X	0	0
Hex Equiv.		0115	0550	0000	0000	0115	0750	0000	0000	0115	0001	0000	0000

Next, we need to identify the PROM addresses at which each of these 16-bit data words will be stored. Referring to Figure C-5, notice that the

PROM address is made up of the opcode (as the most significant hex character) and the present-state bits (as the least significant 2-bit hex character). (Recall that we mentioned [earlier](#) that we are using a modified hex numbering system to label each PROM memory location: the most significant hex character represents the opcode and the least significant hex character is the hex equivalent of the 2-bit present-state value). Using this addressing scheme and the information in Table C-10, we can create the table containing the PROM contents for our 3-instruction instruction set, as shown in Table C-11.

Your next task, [Task 5-4](#), is to use LogicWorks™ to create the PROM you will need to build your 3-instruction instruction-set controller. Turn now to [Task 5-4](#).

**Table C-11. PROM Contents for 3-Instruction Instruction Set.**

Address (Mod. Hex Code)	Contents (Hex)
00	0115
01	0550
02	0000
03	0000
10	0115
11	0750
12	0000
13	0000
20	0115
21	0001
22	0000
23	0000

# APPENDIX D: CREATING A PROM DEVICE IN LOGICWORKS™ FOR WINDOWS®

---

## Introduction

Creating a PROM device in LogicWorks™ involves three major steps:

- Defining what is to be entered into the PROM and determining the size of the PROM.
- Creating a text file containing the PROM contents.
- Using the PROM/RAM/PLA Wizard to create and store the PROM.

## Defining the PROM Contents

Within the context of this laboratory manual, the motivation for creating a PROM is to implement the logic functions needed to drive the synchronized Mealy-machine that implements the microprocessor controller function. The first step in creating a PROM in LogicWorks is to define what is to be entered in the PROM. The data to be entered into the PROM for the limited instruction set was defined in Simulation Lab 5 and is listed again in Table D-1.

The next step is to determine the size of the PROM. The number of words the PROM must store is  $2^{(\# \text{ of input variables})}$ . There are 6 input bits to our PROM, 4 bits representing the operation code of the instruction to be executed, and 2 bits to control the next state of the Mealy machine controller. Therefore the PROM must have  $2^6 = 64$  words. The length of each word is determined by the number of output bits. For the controller, the number of output bits is 2 (next state bits) + 10 (operation control bits) + 4 (bits to be used with later exercises<sup>1</sup>) = 16.

## Creating the instruction file

Once the size of the PROM has been determined and the contents of the PROM have been defined, the next step in creating a PROM is to enter the data in the appropriate format for use with LogicWorks™. LogicWorks™ for Windows accepts two formats for the instruction or data file. They are the Intel hex format and the raw hex format. This guide will focus on the raw hex format<sup>2</sup>. The raw hex format data

<sup>1</sup> It is coincidental that, with the 4 PROM output bits to be used with future exercises, our PROM has a word length of 16 bits. PROM's are commercially available with word lengths that are a power of 2. Since we use 12 bits in our application in Simulation Lab 5, we need to select the next largest PROM, which has a 16 bit word length.

<sup>2</sup> The advantage of using the raw hex format is that it is the simplest data entry method that allows a posteriori modification of the data.

file may be created in any word processing program or text editor that allows the file to be saved as text only, such as Microsoft Word™ or Notepad. Using the word processor you chose, enter the PROM contents (i.e., controller microinstructions), for each memory location address beginning with address 0. Enter each 16-bit word (microinstruction) in hex format with a non-hex character separating each word. Each microinstruction must be entered from the most-significant to the least-significant byte. When all microinstructions have been entered, save the file as a text only file with a *.hex* extension.

**Table D-1.** Input-Output Table for *.hex* File.

Instruction:		Load ACC				ADD				STOP <sup>3</sup>			
Op Code:		0				1				2			
State Bits:		00	01	10	11	00	01	10	11	00	01	10	11
Description	Pin Number												
Next State Bit	1,0	01	00	00	00	01	00	00	00	01	01	00	00
Load IR	2	1	0	0	0	1	0	0	0	1	0	0	0
Write	3	0	0	0	0	0	0	0	0	0	0	0	0
Read	4	1	1	0	0	1	1	0	0	1	0	0	0
ACC to Data Bus	5	0	0	0	0	0	0	0	0	0	0	0	0
Load ACC	6	0	1	0	0	0	1	0	0	0	0	0	0
Load MAR	7	0	0	0	0	0	0	0	0	0	0	0	0
Use PC	8	1	1	0	0	1	1	0	0	1	0	0	0
A Only	9	0	0	0	0	0	1	0	0	0	0	0	0
Invert	10	0	1	0	0	0	1	0	0	0	0	0	0
Arith/Logic	11	0	0	0	0	0	0	0	0	0	0	0	0
X	12	0	0	0	0	0	0	0	0	0	0	0	0
X	13	0	0	0	0	0	0	0	0	0	0	0	0
X	14	0	0	0	0	0	0	0	0	0	0	0	0
X	15	0	0	0	0	0	0	0	0	0	0	0	0
HEX Code:		0115	0550	0000	0000	0115	0750	0000	0000	0000	0001	0000	0000

As an example, let's create the raw hex format file for the Load ACC, ADD and STOP instructions shown in Table D-1. The hex codes for these instructions are listed at the bottom of the table. Entered these hex codes in your text editor exactly as they appear in this table. If you use a space as the non-hex delimiting character, the first line of your file should be:

**0115 0550 0000 0000**

(You may use any non-hex character that you prefer including line-breaks as a delimiter.) The ADD instruction codes could be placed on the same line as those for the Load ACC or on the next line. The same is true for the STOP instruction. The final raw hex file you create should appear similar to Figure D-1.

<sup>3</sup> The first microinstruction of the STOP instruction shown in this table may contain anything since it will never be executed; however, if the STOP instruction is assigned to reside first in the PROM memory (i.e., assigned an Op Code of "0") it's first microinstruction will be executed when the microprocessor is initialized. In such a case, the first microinstruction must be the fetch microinstruction.

```
0115 0550 0000 0000
0115 0750 0000 0000
0115 0001 0000 0000
```

**Figure D-1.** Raw hex format file.

After entering this data, save the file as a *text only* file with a *.hex* extension, for example, *PROM1.hex*. Note the location of the file since you will need to direct LogicWorks™ to this file in the next step of creating a PROM device.

Notice that the opcode for each instruction you enter in the PROM is determined by the order in which you enter the instruction; hence, opcode ‘0’ is defined by the microinstructions in the first 4 memory locations, opcode ‘1’ occupies the next four PROM locations, etc. If you chose to skip a certain opcode, you must still enter four 16-bit data words (perhaps all 0’s) in the locations in PROM that correspond to that opcode. For example, if we assign ‘0’, ‘1’, ‘3’ to the ‘Load ACC’, ‘Add to ACC’, and ‘Stop’ instructions, then the data that we enter in the hex file will look like:

```
0115 0550 0000 0000
0115 0750 0000 0000
0000 0000 0000 0000
0115 0001 0000 0000
```

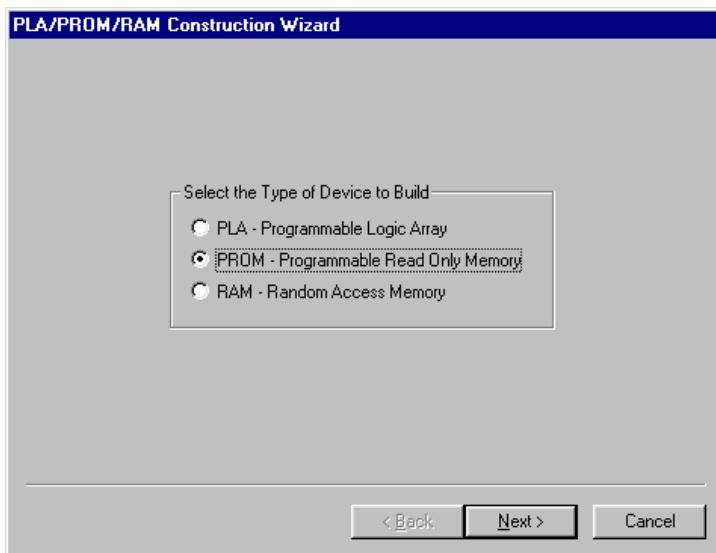
**Figure D-2.** Skip the ‘2’ opcode.

### Create a PROM Device with the LogicWorks™ PROM/RAM/PLA Wizard

Before opening the PROM/RAM/PLA Wizard, open a library that you created to store the PROM device you are developing. (You can open or create a library by clicking your right mouse button while pointing to the *parts selection area* of the *parts palette*. If you are not sure how to create a library, see the instructions for creating a sub-circuit in [Simulator Tutorial: Using LogicWorks™ for Windows](#).) It is advisable to store the PROM in a library that you created and not one of the LogicWorks™ supplied libraries. This protects your file from possibly being lost when upgrading LogicWorks™ in the future.

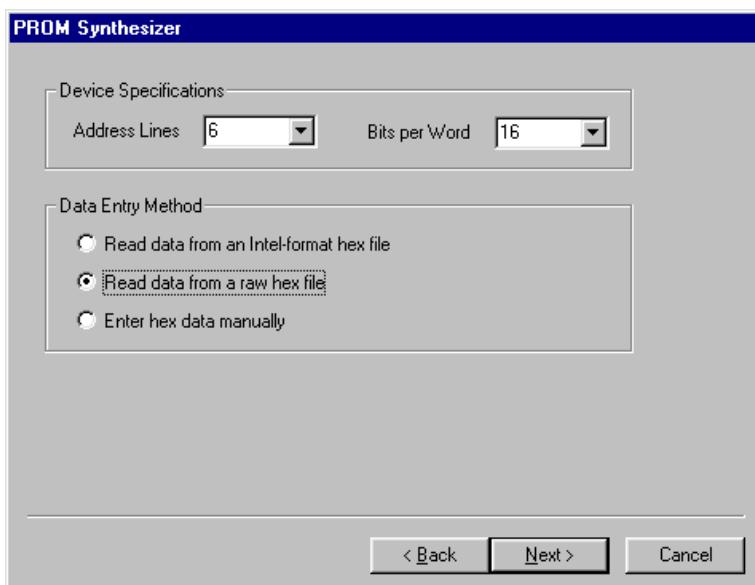
After you have opened your library, engage the PROM/RAM/PLA Wizard by selecting **Simulation→PROM/RAM/PLA Wizard** from the LogicWorks™ *Menu Bar*. The **PLA/PROM/RAM Construction Wizard** Dialog Box will appear as shown in Figure D-3.

Select **PROM - Programmable Read Only Memory** then click on the **Next** button. The next window to appear will be the **PROM Synthesizer** Dialog Box as shown in Figure D-4.



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

**Figure D-3.** PLA/PROM/RAM Construction Wizard dialog box.



Generated using LogicWorks™ (Capilano Computing Systems, Ltd.)

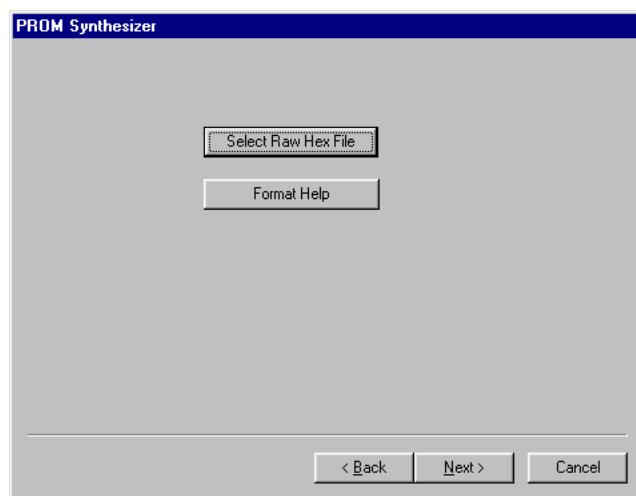
**Figure D-4.** PROM Synthesizer dialog box for entering number of input/output lines and data entry method.

You are required to enter data in two locations. The **Address Lines** entry field allows you to specify the number of inputs the PROM will have. (To build the PROM you will be using in the Simulation Lab 5 exercise, you will enter “6” in the **Address Lines** entry field.) The **Bits per Word** entry box is used to specify the PROM word length, which is synonymous with the number of PROM output pins. (For the PROM you are building for the Simulation Lab 5 exercise you will enter “16” in this field, since each microinstruction requires a 16 bit word.) Next, select the **Data Entry Method** to be used. In this guide, we

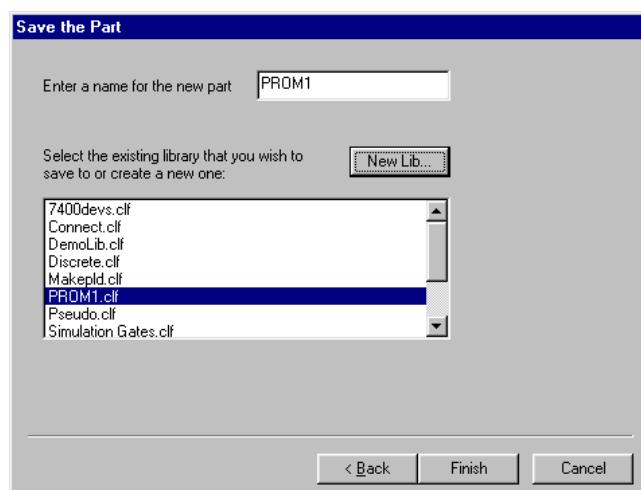
are using the raw hex format so select **Read data from a raw hex file** then click on the **Next** button<sup>4</sup>. This will take you to the second half of the **PROM Synthesizer** dialog, which appears in Figure D-5.

Click on **Select Raw Hex File** then select the .hex file that you created earlier. (If you used the file name suggested earlier, you will select the file, “PROM1.hex” from the directory in which you stored it.) Then click the **Next** button. This takes you to the final step of creating a PROM device, saving the device in a library.

The **Save the Part** dialog appears as shown in Figure D-6. Enter a name for your PROM device in the **Enter a name for the new part** field, for example, ‘PROM1’. Then click on the name of the library in which the PROM will be saved. If you did not open your own library earlier, you may create a new one at this point by clicking on the **New Lib** button and typing in a name for the new library. After completing these items, click on the **Finish** button and your new PROM part name will appear in the library you selected. To use your part, open your library, select the new ‘PROM1’ device and place it in the design window as you would place any LogicWorks™ part.



**Figure D-5.** PROM Synthesizer dialog box for selecting raw hex file.



**Figure D-6.** Save the Part dialog box.

<sup>4</sup> Selecting the “Enter hex data manually” allows you to enter data directly into the PROM without the use of a .hex file. The advantage to this approach is that you do not have to go to notepad to create a hex file. The draw back to this entry method is that once the PROM is created, there is no way to edit the data if/when changes need to be made; if you find an error you must enter ALL of the data again. As you create instructions for your microprocessor you will most likely find errors in your microinstructions and will need to reenter the data several times before you get each instruction working the way you want. For that reason it is recommended that you use the “Read data from a raw hex file” input method.