Laboratory Manual and Supplementary Notes

ECE 458: Computer Systems Laboratory

Version 1.0

Dr. Sol Rosenstark
Dr. John Carpinelli

Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, New Jersey

Contents

Very Important General Notes

Procurement of Components

Each student has to take the initiative to obtain the following items, which will be needed for experiments 5, 6 and 7.

- One GAL such as the Lattice 16V8.
- One 2816A EEPROM.
- One 74374 octal D-type latch (or CMOS equivalent).
- One 74298 quad 2-to-1 multiplexer with latched outputs.
- One debounced push button switch to simulate the system clock.
- Two DIP switches.
- Three LEDs and limiting resistors.
- Pencilbox kit (optional).
- Additional 74-series ICs based on the student's design.

Programming GALs and EEPROMs

In the \SP\BIN subdirectory type SP<ENT>.

- Type <F9> to select the programming of GALs or EEPROMs, and then <esc> to get out.
- Type <F7> and use < $*$ > <ENT> to select the device manufacturer, and then <esc> to get out.
- Type <F8> and use < $*$ > <ENT> to select the device number, and then <esc> to get out.

The proper data should now appear in the right middle of the screen. Use the FILE menu to load the file that you use to program the chip. Use the BUFFER, EDIT feature to verify the data that has been loaded. Go to the DEVICE screen to program the chip. It's self explanatory from here.

EXPERIMENT 1: Introduction to the MC68000 Educational Computer Board and a Program For Data Input

Objectives

This experiment consists of two parts. In the first part, the student is introduced to the operation of the Motorola 68000 CPU based educational computer board (ECB), the public domain communications package PROCOMM, and the Antonakos assembler. In the second part, a program has to be written which will accept data from the PC keyboard.

Equipment Needed

- MC68000 Educational Computer Board with RS232 communication cable, with 25 pin to 9 pin transition cable, as well as the *SBC68K User's Manual* by Arnewsh Inc. These items are all available from the stockroom.
- The students should have in their possession, from the prerequisite Microprocessor course, the textbook *The 68000 Microprocessor, Hardware and Software Principles and Applications*, by James Antonakos, 4th Edition, Prentice Hall, 1999.

Background

The Motorola MC68000 Educational Computer Board includes:

- A 68000 microprocessor acting as the central processing unit.
- 16K-bytes of read only memory (ROM) containing a monitor program.
- 32K-bytes of read/write memory (RAM).
- Two serial RS232-C compatible communications ports, one for a terminal and one for a host.
- An audio tape serial I/O port and a parallel port.
- Reset and Abort function switches.
- The firmware has monitor, debug, one line assembly capability and a multiple line disassembly function.
- User programs can be uploaded to and downloaded from a laboratory PC.

The MC68000 ECB will be used with a PC as a host. To set up communications between the PC and the MC68000 use the following procedure.

1. Connect the communications cable to the RS232 port COM1 of the PC and port 1 of the MC68000 ECB. Set the SBC serial communications speed to 9600 or 19200 bits/second.
2. Use a DOS window on the PC and find the subdirectory C:\PROCOM.
3. Type P<ENT> to bring it up. You can use the CTRL-Z function to get a help menu.
4. Use ALT-P to set PROCOMM to the desired bit rate, no parity, 8 bits and 1 stop bit.
5. To make sure that the SBC does not ruin your HEX files when they are loaded, use ALT-S to set the ASCII TRANSFER OPTIONS for a character pacing of 500 millisec. All other parameters should be 0 and all other options should be answered in the negative. Save the settings when you exit.
6. Plug in the SBC and you should then have the prompt **TUTOR 1.3** > which indicates that you have established communications between the PC and the ECB.

7. To upload a HEX file from the PC, at the SBC command line type LO1<ENT>. The SBC will now be waiting for the HEX file transfer to begin. Use the PgUp key to tell PROCOMM to send the file. Choose the ASCII upload protocol and type, in the little box, the subdirectory and the file name. That's all there's to it. (Question: Why **do** we use ASCII file transfer ?)

Using The MC68000 Onboard Assembler

Though normally your programs will be assembled by the assembler on a PC in room 211F, the MC68000 has an onboard assembler. It is a line-at-a-time assembler which means the programmer must compute all addresses. This assembler should be used for making small changes in a program. The user is referred to the *SBC68K User's Manual* for additional information on the use of this assembler.

Using The Antonakos ASM68K Assembler

The students should have acquired a reasonable familiarity with this assembler in the microprocessor course. This assembler should be found in the C:\ASM68K subdirectory on the computers in room 211F.

Preparing Your Programs For Assembly

Use any ASCII editor to prepare your program. Document type word processors such as Word or WordPerfect are not editors that produce ASCII files and should therefore be avoided. DOS Edit or Notepad will do the job well enough.

A sample program, TEST, is given below:

```
; The SBC uses TRAP #14 to access many functions in the
; MONITOR program which operates it. You have to inform
; the TRAP #14 service routine what function you want it
; to perform by putting a specific code-byte into D7.
; These equates explain some of these code bytes.
inchar  equ     247 ;code for getting a keyboard char
outchar equ     248 ;code to send a char to the screen
exit    equ     228 ;code for a graceful exit to TUTOR
        org     $2000
; This program inputs a character from the
; keyboard until '.' is typed. The character
; is then displayed on the monitor. The
; characters typed are then stored in memory.
getput  lea.l   storend,a6 ;point A6 to char storage
next    move.b  #inchar,d7 ;Get a character into D0.B
```

```
        trap    #14        ;using a MONITOR routine
        move.b  #outchar,d7;Put it on the screen
        trap    #14
        move.b  d0,-(a6)   ;Store it going backwards
        cmpi.b  #'.',d0    ;See if char is a '.'
        bne     next       ;No, then loop for more
        move.b  #exit,d7   ;Done, so exit gracefully
        trap    #14        ;to the MONITOR prompt
; Data storage area. We reserve only 20 bytes in RAM.
stor    ds.b    20
storend
        end     getput     ;This goes at the very end
```

Save this program in a file with extension .ASM, for example TEST.ASM.

Assembling Your Program

On the DOS prompt simply type A:>C:\ASM68K TEST. If errors are signaled at the end of assembly then examine the file TEST.LST to see what went wrong. Correct the errors in the TEST.ASM file and repeat the above procedure till all the errors are eliminated. The file TEST.HEX is an ASCII file that can be downloaded to the SBC for testing and can also be emulated with the Livadas and Ward simulator SIM68K.EXE.

The Experiment

1. Review Chapters 1–3, in Antonakos's book.
2. What are the possible addressing modes of the MC68000 ? Give one example of each addressing mode in the form of an assembly language instruction.
3. Read TRAP #14 for parameters #247 and #248 and #228 in the *SBC68K User's Manual*.
4. Create the file for the program TEST.ASM using an ASCII (non-document) editor. Assemble it using Antonakos's ASM68K.EXE assembler. Find the Livadas and Ward simulator SIM68K.EXE. It is readily recognizable when it is loaded into memory because it produces the sign-on display appearing below.

```
C:\>sim68k
MC68000/ECB Simulator.
Copyright (C) Livadas and Ward, 1992.  Author Wayne Wolf
Version 2.3
SIM68000  2.3 >
```

5. This simulator is case sensitive and works with capital letter commands only. The commands are the same as for the ECB board as explained in the *SBC68K User's Manual* by Arnewsh Inc. The only difference is that to load a file you simply type LO <TEST.HEX>. (For the SBC you would type LO1 and then tell PROCOMM to upload the file.)
6. Observe that when you emulate the file TEST, the program crashes at a certain point. When does this happen and why ?
7. Modify the program TEST to include a test to make sure that the stored characters do not endanger the program itself. The program should, at that point, stop accepting any more characters and put out a message on the screen stating that an overflow has occurred.
8. Demonstrate the working version of the modified program TEST.
9. Using the MC68000 ECB on board assembler, write a delay loop for a delay of 1 sec. Only relative branch (Bcc) instructions should be used. The starting address of this code should be at 2000H.
10. Write routines to add, subtract and multiply two single byte numbers which the user places on the stack using the TUTOR. The programs should POP the parameters off the stack and place the computed result in register D0.
11. Write a program that reads in data from the PC keyboard. The data consists of two parts:

    (a) Name of a student. This may contain up to six alphabetic characters. The program must filter out all non-alphabetic characters and all characters after the first six alphabetic characters.

    (b) ID number of a student. This must contain exactly nine digits. The program must filter out all non-numeric characters and all characters after the first nine digits.

    The data is to be stored in the memory as follows:

    SMITH>883925333(0D)(0A) *next record*

    and so on. This program should be capable of inputting data until the user presses a 'hot key'. A 'hot key' can be any one of the function keys, control keys or ESC, and pressing it indicates that the data entry is terminated. The program should then display all the data that has been inputted before it exits. TRAP #14 handlers should be used for the input and output of data. Upload the data on your disk for later use in Experiment 2 and show the data to your instructor.

EXPERIMENT 2: Data Record Sorting

Objectives

In this experiment, the MC68000 ECB is used to sort the records which were created in Experiment 1. The program should be able to sort the records both alphabetically (according to name) or numerically (according to number).

Equipment Needed

- MC68000 Educational Computer Board with RS232 communication cable, with 25 pin to 9 pin transition cable, as well as the *SBC68K User's Manual* by Arnewsh Inc. These items are all available from the stockroom.
- The students should have in their possession, from the prerequisite Microprocessor course, the textbook *The 68000 Microprocessor, Hardware and Software Principles and Applications*, by James Antonakos, 4th Edition, Prentice Hall, 1999.

Background

Data sorting is commonly done by computers for better presentation and structuring of the data to facilitate its search and retrieval. Sorting is a function included in all database management programs. In Experiment 1, data records were entered into the ECB through the keyboard of the laboratory computer. It is desired to arrange this data in a logical order.

Chapter 6 of Antonakos's book contains explanations of bucket sorting and bubble sorting algorithms. We are interested in one that can be implemented easily. Efficiency is not an important criterion since our database has fewer than twenty records

The Experiment

1. Write a commented assembly language program to implement the data sorting operation. It should be capable of sorting both alphabetically (according to name) and numerically (according to number).

2. Assemble and debug the program the so that it will sort the records created in Experiment 1. The sort should be done in two ways: The program should sort the records in ascending order. It should display both outputs, that is records sorted alphabetically and records sorted numerically. Assemble and test your program. Get a printout of the output and attach it to your report.

3. Change the program so that it sorts data in descending order instead of ascending order. Get a printout of this and attach it to your report.

EXPERIMENT 3: A Basic Mathematical Package for Multibyte Numbers

Objectives

In this experiment we develop a basic mathematical package for multibyte BCD numbers. The operations which will be possible using this package will be addition and multiplication. This experiment consists of two parts. In the first part a program for addition will be implemented. In the second part a program for multiplication will be implemented.

Equipment Needed

- MC68000 Educational Computer Board with RS232 communication cable, with 25 pin to 9 pin transition cable, as well as the *SBC68K User's Manual* by Arnewsh Inc. These items are all available from the stockroom.
- The students should have in their possession, from the prerequisite Microprocessor course, the textbook *The 68000 Microprocessor, Hardware and Software Principles and Applications*, by James Antonakos, 4th Edition, Prentice Hall, 1999.

Background

A mathematical package is normally developed for microprocessor based systems as a library of subroutines. Each subroutine has a different function, such as addition, subtraction, multiplication and division. There are different sets of subroutines for integers, real numbers and floating point numbers. Subroutines can also be developed for other functions, such as sines, cosines, tangents etc.

These subroutines, once developed, can then be called from any program. This cuts down the development time of the programs. These subroutines should be well documented and the registers and memory used for parameter passing and return of results should be similar for all such subroutines.

The Experiment

1. List the instructions used for mathematical operations on BCD numbers for the MC68000.

2. Write commented assembly programs and include flow charts for the implementation of routines for addition and multiplication of multibyte BCD positive integers. The number of bytes in a number are stored in one register and a pointer to the start of the number is stored in another register. The second number is also similarly stored with different registers for the number of bytes and the pointer. All these routines should return the answer in a similar format in two registers. This problem is greatly simplified if you use a similar structure for all these routines. Multiplication can be done by any algorithm, the simplest being multiplication by iterative addition. Minimize the number of scratch registers. Save and restore all registers you use.

3. Assemble and debug the routines for addition and multiplication. Test the routines and attach two examples of each operation.

4. Attach an example for addition where a carry is generated.

EXPERIMENT 4: An MC68000 Based Rudimentary Calculator

Objectives

In this experiment a rudimentary calculator is developed. The subroutines developed in the earlier experiment can be used with modifications if necessary. The operations which will be performed by this calculator are addition, subtraction and multiplication.

Equipment Needed

- MC68000 Educational Computer Board with RS232 communication cable, with 25 pin to 9 pin transition cable, as well as the *SBC68K User's Manual* by Arnewsh Inc. These items are all available from the stockroom.
- The students should have in their possession, from the prerequisite Microprocessor course, the textbook *The 68000 Microprocessor, Hardware and Software Principles and Applications*, by James Antonakos, 4th Edition, Prentice Hall, 1999.

Background

This experiment is a practical application of the techniques learned in the earlier experiments. The mathematical package developed in experiment 3 will be streamlined for ease of user input and output. TRAP #14 handlers will be used for data input and output.

The Experiment

1. Write a program to implement a rudimentary calculator for positive BCD numbers. The calculator should work as follows.

   (a) All the keyboard entries described below should be echoed on the monitor.

   (b) The first operand should be typed in from the keyboard. It should be followed by a space typed on the keyboard.

   (c) The operation to be carried out should be typed next followed by a space. This operation can be $+$, $-$ or $*$.

   (d) The second operand should be typed in followed by <ENT>. The calculation should now be performed and the result should be displayed on the next line.

   An example is shown below.
   2138$< SP > + < SP >$ 9528 $< ENT >$
   11666

   The operands will be positive integers up to 15 digits long.

2. Your code should handle illegal key intries in a sensible way.

3. Get the printout of one example of each operation.

4. Explain briefly how you would implement a function like $\sin(x)$, $\cos(x)$ and $\tan(x)$, where x is in degrees.

EXPERIMENT 5: Sequential Circuit Using Programmable Logic

Objectives

The objective of this lab is to take the student through the design of a sequential circuit in an implementation utilizing a programmable logic device.

Preparation

Review the material pertaining to sequential logic in *Logic and Computer Design Fundamentals*, by M. Morris Mano and Charles R. Kime, Prentice Hall, 1997.

Requisite Equipment and Reference

- One GAL such as the Lattice 16V8-25.
- The dice-tosser jig for testing the GAL.
- *Logic and Computer Design Fundamentals*, by M. Morris Mano and Charles R. Kime, Prentice Hall, 1997.

Background

Introduction to GAL

When designing digital circuits, the question is often asked: “Is it possible to reduce the chip count by utilizing a more advanced IC technology?” The technology in question is generally referred to as ASIC (Application Specific Integrated

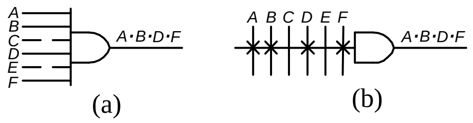

Figure 5.1: Conventional logic notation (a) contrasted with the more compact ASIC notation (b).

Circuits). The answer is in most instances affirmative, but certain techniques are not suitable for applications involving small production runs because some devices require an IC foundry for implementation of the chip, hence a minimum number of chips is required per order to make this process economical. There is also substantial turn-around time of more than one week. On the other hand techniques involving erasable programmable read only memory (EPROM), programmable array logic (PAL) and generic array logic (GAL) devices can be used at the design bench even by small operators. We, in the academic community, are the latter, hence we will concentrate our attention on these chips.

Before we begin the explanation of PALs and GALs we need to become familiar with some new notation. Figure 5.1(a) shows the logic diagram of a six input AND gate of which four are used. Since a PALs and GALs can have AND gates with many inputs (32 in one case), the notation has to be modified as shown in figure 5.1(b). Each of the variables A, B, C, D, E, F can be an input to the AND gate if its fuse is left intact. An intact fuse is indicated by a $\times$ symbol. Clearly, variables C and E are not inputs to this AND gate. It may be noted that any unconnected AND gate inputs are automatically pulled up, hence they do not affect the output.

The first fusible link technology was applied to memory devices such as field programmable read only memory. The PROM can be used to implement combinatorial logic. Some pin saving can be obtained by using a chip design which is not quite as general as that of the PROM. That is the function of the PAL and the GAL, henceforth referred to jointly as PAL/GAL. To get an understanding of the difference between the two chip architectures, we carefully examine figure 5.2.

In the PROM, the AND array inputs are fixed and the OR array inputs are programmable. The sixteen (16) AND gates shown produce all possible minterms of (I_3, I_2, I_1, I_0). During the programming phase the designer has the freedom to choose which minterms are ORed into each of the four output OR gates. The programming determines which of the connections to the OR gates are removed, leaving in place only those desired. Suppose we need one combinatorial PROM output given by

$$O_3 = I_0 \tag{5.1}$$

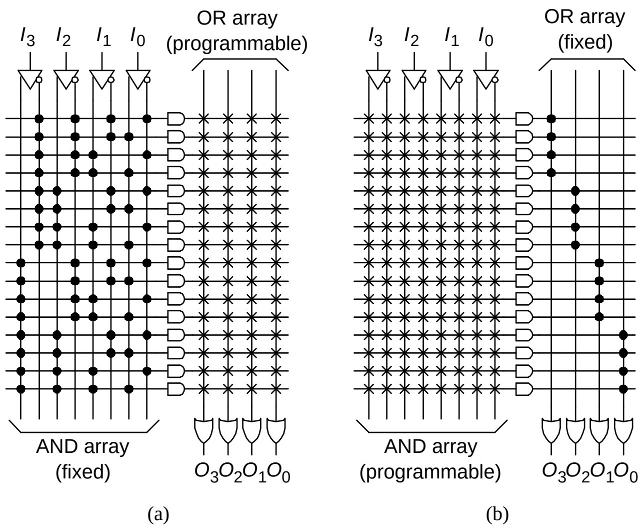

Figure 5.2: PROM architecture (a) compared with PAL/GAL architecture (b).

This has to be implemented by using the sum of the eight minterms

$$\begin{aligned} O_3 &= \overline{I_3} \cdot \overline{I_2} \cdot \overline{I_1} \cdot I_0 + \overline{I_3} \cdot \overline{I_2} \cdot I_1 \cdot I_0 + \overline{I_3} \cdot I_2 \cdot \overline{I_1} \cdot I_0 + \overline{I_3} \cdot I_2 \cdot I_1 \cdot I_0 \\ &+ I_3 \cdot \overline{I_2} \cdot \overline{I_1} \cdot I_0 + I_3 \cdot \overline{I_2} \cdot I_1 \cdot I_0 + I_3 \cdot I_2 \cdot \overline{I_1} \cdot I_0 + I_3 \cdot I_2 \cdot I_1 \cdot I_0 \quad (5.2) \end{aligned}$$

There is absolutely no point in reducing a logic function to its minimum number of terms, since the output is a sum of minterms in any event.

As can be seen in figure 5.2(b), the PAL/GAL has a fixed OR array and a programmable AND array. To get the function O_3 described in 5.1, it is necessary to program the inputs to the topmost AND gate. This is done by removing all the links used for ANDing normal and complemented versions of I_3, I_2 and I_1. In addition, the link for ANDing $\overline{I_0}$ is removed.

All the links are left intact for the three AND gates below the topmost AND gate. For those AND gates every variable is presented to the input in both normal and complemented form, so these AND gates will have outputs of zero. This way they will have no effect on the OR whose output is O_3.

In the PROM realization we have utilized eight 8-input AND gates and one 8-input OR gate to get the function O_3. In the PAL/GAL realization we need only four 8-input AND gates and one 4-input OR gate. This saving in IC hardware can be put to good use in implementing other desirable functions on the chip. Since the PAL/GAL is not specifically a memory storage device, PAL/GAL designers can put all kinds of features into them making them very flexible for the user. The kind of features shown in figure 5.3 are incorporated into the 16L8 *combinatorial* PAL. It allows the enabling of the output tri-state buffer to be controlled by a logic expression. It also allows for feedback from any output terminals, in normal or complemented form, to the inputs of any AND gates.

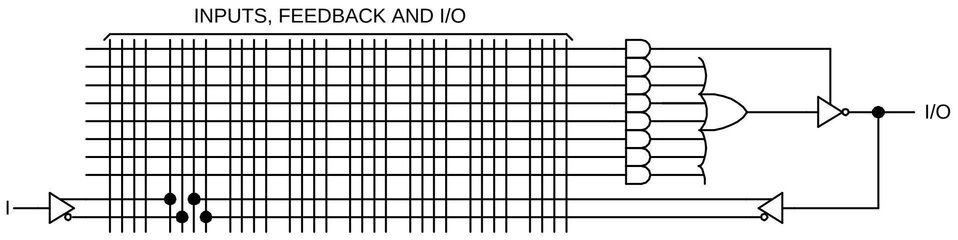

Figure 5.3: A block of combinatorial logic of a 16L8 PAL.

To give the designer additional capabilities, registers in the form of D flip-flops are included into *registered* PALs such as the 16R8. This is illustrated in figure 5.4. It is beyond the scope of this presentation to discuss every conceivable PAL/GAL architecture. It is hoped that once the student gains familiarity with the use of one specific PAL/GAL architecture, the knowledge will be transferable to other devices without much difficulty.

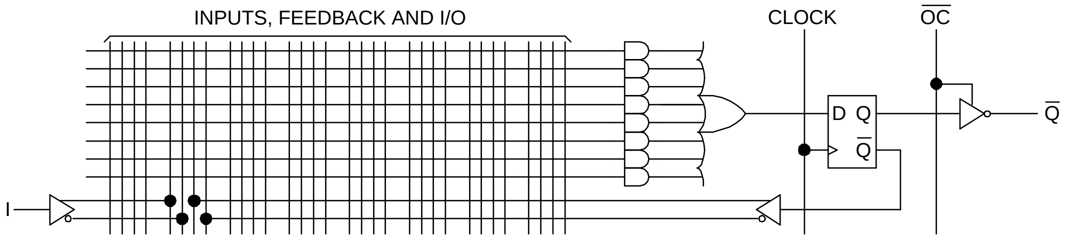

Figure 5.4: A block of registered logic of a 16R8 PAL.

In our implementation we have chosen to use GALs because they, unlike PALs, can be reprogrammed numerous times. In addition, the 16V8 GAL has a more general architecture than the 16R8 PAL. The D flip-flop registers can be either included or excluded in the final realization, according to the designer's choosing. As a consequence a 16V8 GAL can be used to replace a great number of PALs. It can replace the purely combinatorial 16L8 PAL as well as the mixed combinatorial and registered 16R4 and 16R6 PALs. In our case the convenience is that it can also replace the fully registered 16R8 PAL.

Table 5.1: The state transition table of one die. W is the wakeup state and A forces the transition of the second die.

Die	Present State				Next State					
Toss	Q_4	Q_3	Q_2	Q_1	Q_4	Q_3	Q_2	Q_1	A	$\overline{A}$
W	1	1	1	1	1	0	1	1	1	0
1	1	0	1	1	1	1	0	1	0	1
2	1	1	0	1	0	0	1	1	0	1
3	0	0	1	1	0	1	0	1	0	1
4	0	1	0	1	0	0	0	1	0	1
5	0	0	0	1	0	1	0	0	0	1
6	0	1	0	0	1	0	1	1	1	0

A Simple Example - A Dice Tosser

The example of the dice tosser which will be presented is based on a design which appeared in the "PAL Programmable Array Logic Handbook," Third Edition, 1983, Monolithic Memories Inc. (MMI), Santa Clara, CA 95050.

It is desired to implement a dice tosser with as little hardware as possible. Each die is to be made up of LEDs driven by sequential logic. The design is to be similar to that shown in figure 5.5. The design shown there uses conventional 74XXX logic chips and would require three 7408 quad 2-input AND gate chips, one 7411 triple 3-input AND gate chip, one 7432 quad 2-input OR gate chip, one 744075 triple 3-input OR gate chip, and two 74175 quad D flip-flops. That's a total of eight chips. We wish to reduce the chip count to unity. We will see presently how such a design is obtained.

We want to use a single pulse train to advance the dice. To do this, the second die will not change state while the first die goes through the count 1, 2, 3, 4, 5, 6. The second die is advanced when the first die reaches the end of its count. Both dice will have the same state transition table, hence table 5.1 contains the data for the transitions for the first die.

On the first design attempt we will simplify the design by *ignoring the first line of the state transition table for the A variable.* We use K-maps to minimize terms for the variables appearing in table 5.1, including consideration of "don't care" states, to obtain the following state equations for the first die:

$$Q_1 := Q_2 + Q_3; \quad Q_2 := \overline{Q_1} + Q_3Q_4; \quad Q_3 := \overline{Q_3}; \quad Q_4 := \overline{Q_1} + Q_2Q_4 \qquad (5.3)$$

The symbol := is used to show how the next state of Q_n is dependent on the previous states appearing on the right of each equation.

If the second die is to use the same sequence, then its equations can be obtained from 5.3 by adding four (4) to every subscript. This produces the result

$$Q_5 := Q_6 + Q_7; \quad Q_6 := \overline{Q_5} + Q_7Q_8; \quad Q_7 := \overline{Q_7}; \quad Q_8 := \overline{Q_5} + Q_6Q_8 \qquad (5.4)$$

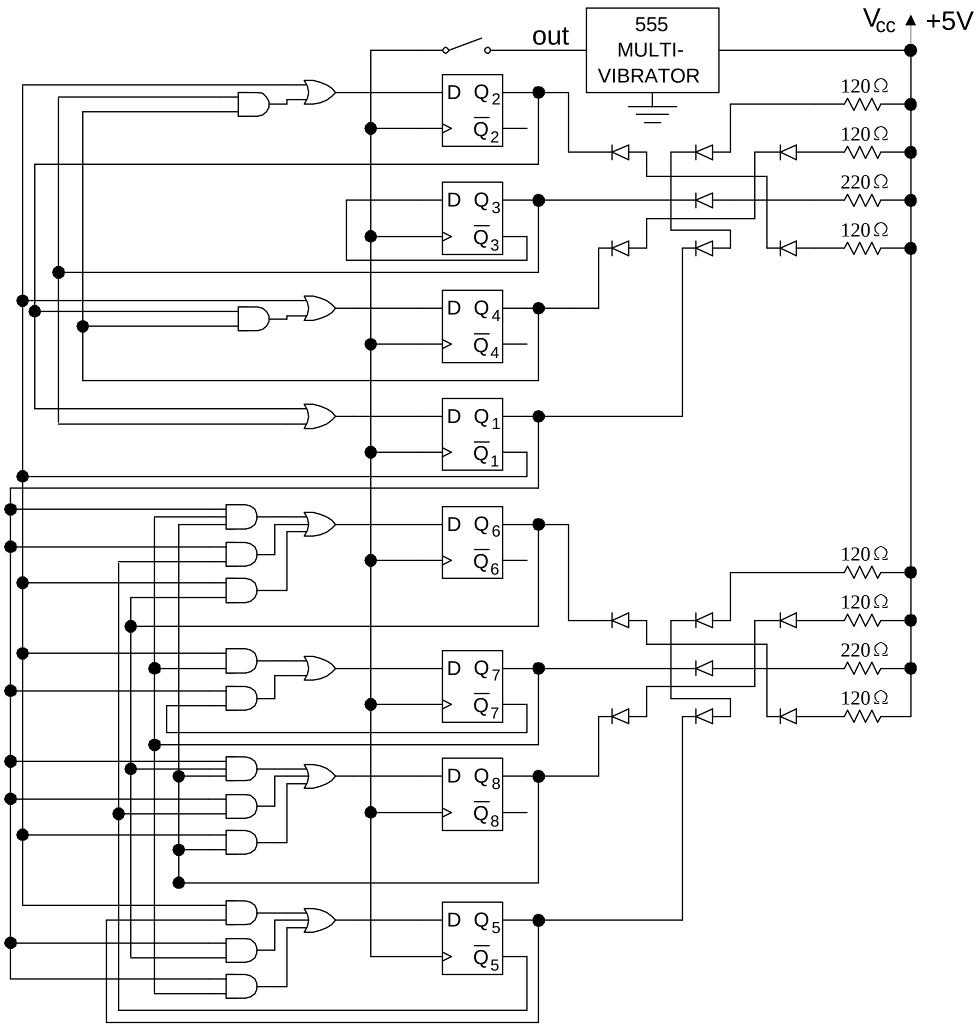

Figure 5.5: Conventional implementation of a dice tosser. This circuit requires three 7408 quad 2-input AND gate chips, one 7411 triple 3-input AND gate chip, one 7432 quad 2-input OR gate chip, one 744075 triple 3-input OR gate chip, and two 74175 quad D flip-flops for a total of eight chips.

Again, we ignore the first line of table 5.1, and obtain the equation for the transition variable A,

$$A = Q_1 \tag{5.5}$$

And while we are at it, we perform an independent determination for its complement $\overline{A}$ to obtain,

$$\overline{A} = \overline{Q_1} \tag{5.6}$$

Since we do not intend to use two separate clocks, we will simply have the first die count up from 1 to 6 then back to 1 again. When the first die goes from state 6 back to state 1, the second die will be incremented. We want Q_n to stay in its current state when $A = 0$ and to make a transition when $A = 1$. Below, the state equations given previously for the second die have been modified to reflect this fact.

In summary the state equations for both dice are:

$$\begin{aligned}
Q_1 &:= Q_2 + Q_3 && (5.7)\\
Q_2 &:= \overline{Q_1} + Q_3Q_4 && (5.8)\\
Q_3 &:= \overline{Q_3} && (5.9)\\
Q_4 &:= \overline{Q_1} + Q_2Q_4 && (5.10)\\
Q_5 &:= \overline{A}Q_5 + A(Q_6 + Q_7) && (5.11)\\
Q_6 &:= \overline{A}Q_6 + A(\overline{Q_5} + Q_7Q_8) && (5.12)\\
Q_7 &:= \overline{A}Q_7 + A(\overline{Q_7}) && (5.13)\\
Q_8 &:= \overline{A}Q_8 + A(\overline{Q_5} + Q_6Q_8) && (5.14)
\end{aligned}$$

The implementation shown in figure 5.6 is based on the above equations. The design requires a substantial amount of combinatorial logic as well as eight D-type flip-flops. An examination of PAL specifications reveals that the 16R8 has the requisite 8 D-type flip-flops. The logic diagram for this PAL is shown in figure 5.7. We should be able to implement this design with that PAL. As has been mentioned earlier, GALs can be used as a very convenient replacement for PALs. Accordingly the final design for the dice tosser, shown in figure 5.6, has a 16V8 GAL in place of a 16R8 PAL. With this design we have reduced the chip count to unity.

In order to program a PAL to do something specific (and useful), we need to generate a bit map which will inform the PAL programmer which fuses to blow and which to leave alone. We need a compiler program to convert (or compile) logic statements into a JEDEC (.JED) file which is used to program (burn) the PAL or GAL. The next section explains the use of such a compiler.

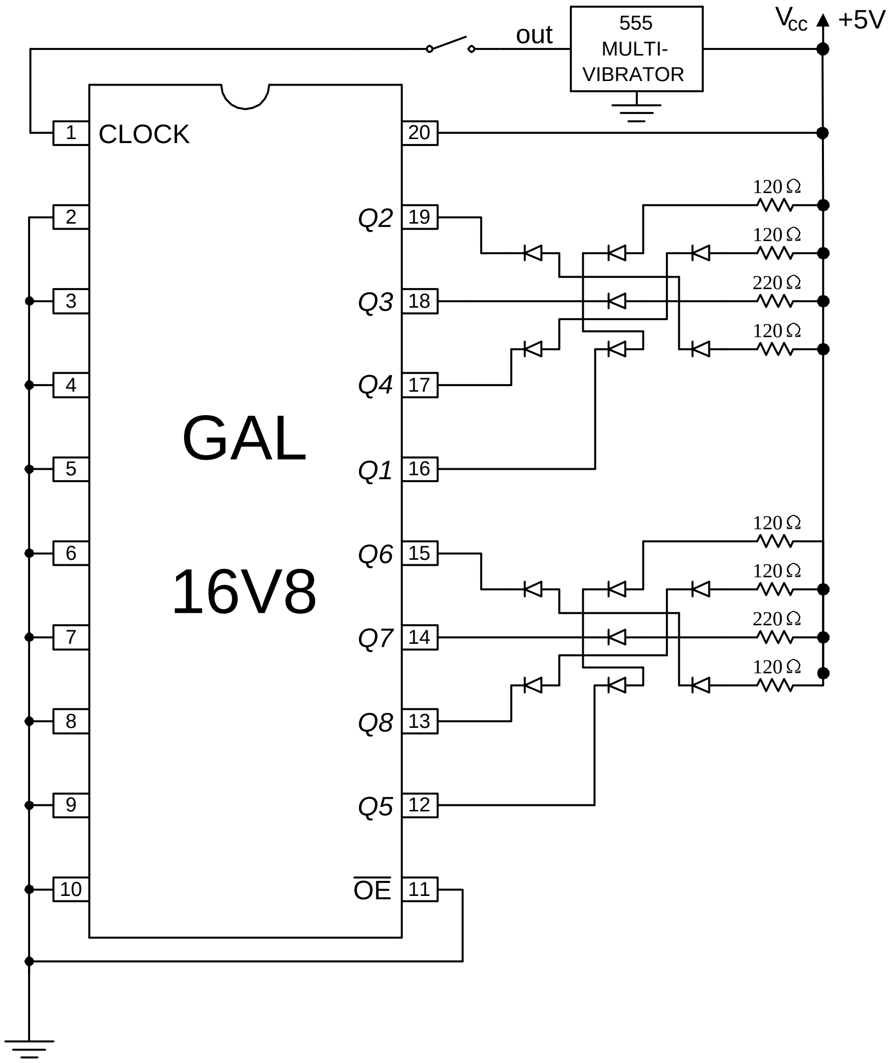

Figure 5.6: Final circuit diagram for the dice tosser.

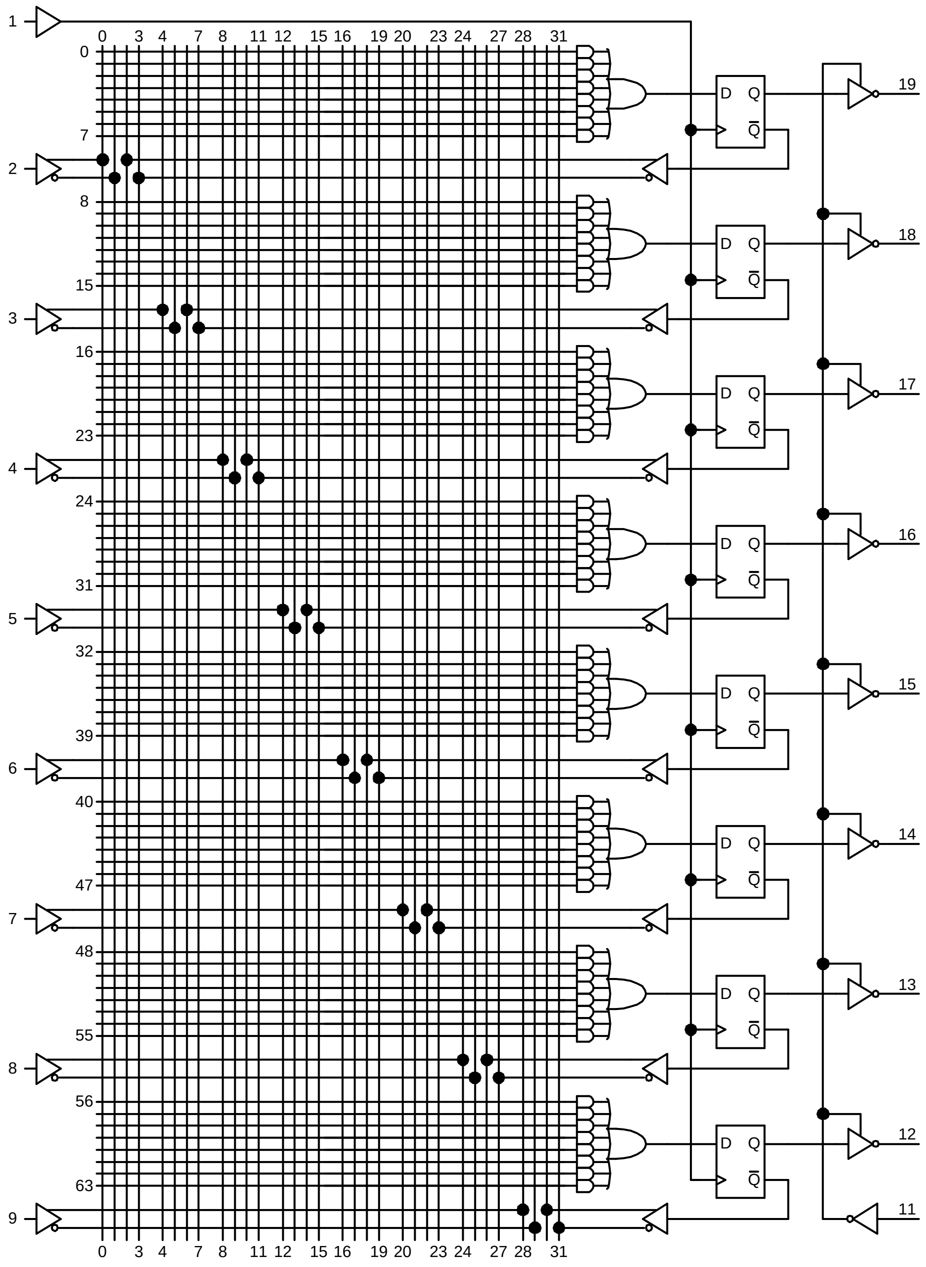

Figure 5.7: Logic diagram for the 16R8 PAL. All fuses are absent in this diagram.

How to use the National Semiconductor ispEXPERT System Project Navigator ABEL compiler

Acknowledgements

I am grateful to Tim Schnettler, of National Semiconductor, for making ispEXPERT System Project Navigator available to NJIT. In addition I wish to thank Dan Gardner, also of National Semiconductor, for helping me become acquainted with this version of the ABEL compiler.

Introduction

In the past we have used AMD's PALASM compiler to obtain the files necessary for programming the Generic Array Logic (GAL) chips needed in some of our experiments. PALASM worked in a DOS environment. The old PCs which we had in our laboratories operated with true DOS under Windows 3.1. The new PCs, recently installed in our laboratories, use the Windows NT operating system. This operating system makes an emulated DOS available. Unfortunately PALASM will not operate in this new DOS environment. No help could be expected from AMD, as that company had sold off its GAL division to Lattice Semiconductor, and it therefore no longer supported PALASM, hence a new GAL compiler had to be found.

CUPL and ABEL are two other well known GAL compilers. Calls to Lattice Semiconductor produced the offer of an ABEL compiler in the form of the ispEXPERT System Project Navigator. This is the compiler which will be used in our labs in the foreseeable future. It is different from PALASM and consequently requires a new set of instructions for its use.

The Source File

I've chosen as an example a student who has the incredible name Dice Toss. If that is indeed your name the you could create a temporary subdirectory \DiceToss for this work. But at the end of the work session you should save your files on a diskette and get rid of this subdirectory so that the computer hard drive does not become cluttered.

The next thing to do is create the source file for the GAL design. Again, if your name is Dice Toss, you could call the file DiceToss.abl and also call the module (see the first line of the sample below) by the same name. (I prefer to use names that are no longer than eight characters.) A sample file DiceToss.abl is shown below and this particular version can be found in the subdirectory C:\DICETOSS.

```
MODULE DiceToss // Change this for a suitable name.
TITLE 'A dice tosser source file for ABEL. These
       equations are for the dice waking up in the,
       not so interesting, (0,1) state.'

// Comments go behind // or ".
// This is an example file for creating a dice tosser
// in which the dice advance in the straight order 1, 2,
// 4, 5, 6. Obviously, the equations and the test vectors
// have to be changed for other sequences.
//      Dr. S. Rosenstark, 1999.

DECLARATIONS
//***************** input pins ******
CLOCK                PIN 1;
PINOE                PIN 11;
//***************** output pins *****
Q8..Q6               PIN 13..15 istype 'reg_d';
Q4..Q2               PIN 17..19 istype 'reg_d';
Q5                   PIN 12     istype 'reg_d';
Q1                   PIN 16     istype 'reg_d';
// Substitution variables also go with DECLARATIONS
A      = !Q1;      // ! is complement
comp_A = Q1 ;
// Definition of variable used in test vectors
PIPS      = [Q8..Q1];

EQUATIONS
// The two following lines you can leave alone
PIPS.C  = CLOCK;   //Hardwired register clock
PIPS.OE = !PINOE;  //Hardwired OE, active low

Q1 := Q2 # Q3;     // # is logic OR
Q2 := !Q1 # Q3&Q4; // & is logic AND
Q3 := !Q3;
Q4 := !Q1 # Q2&Q4;
Q5 := comp_A&Q5 # A&(Q6 # Q7);
Q6 := comp_A&Q6 # A&(!Q5 # Q7&Q8);
Q7 := comp_A&Q7 # A&(!Q7);
Q8 := comp_A&Q8 # A&(!Q5 # Q6&Q8);

// All D FFs wake up in reset mode. After the first upward
// clock transition, we expect to get the dice toss 0,1 then
// 0,2 and so on. The binary values were simply copied from
// transition table 5.1.
TEST_VECTORS
    ( [CLOCK] -> [PIPS] )
      [  0  ] -> [^b11111111]; // Power up
      [ .C. ] -> [^b11111011]; // 0, 1
      [ .C. ] -> [^b11111101]; // 0, 2
      [ .C. ] -> [^b11110011]; // 0, 3
      [ .C. ] -> [^b11110101]; // 0, 4
      [ .C. ] -> [^b11110001]; // 0, 5
      [ .C. ] -> [^b11110100]; // 0, 6

      [ .C. ] -> [^b10111011]; // 1, 1
      [ .C. ] -> [^b10111101]; // 1, 2
      [ .C. ] -> [^b10110011]; // 1, 3
      [ .C. ] -> [^b10110101]; // 1, 4
      [ .C. ] -> [^b10110001]; // 1, 5
      [ .C. ] -> [^b10110100]; // 1, 6

      [ .C. ] -> [^b11011011]; // 2, 1
      [ .C. ] -> [^b11011101]; // 2, 2
      [ .C. ] -> [^b11010011]; // 2, 3
      [ .C. ] -> [^b11010101]; // 2, 4
      [ .C. ] -> [^b11010001]; // 2, 5
      [ .C. ] -> [^b11010100]; // 2, 6

      [ .C. ] -> [^b00111011]; // 3, 1
      [ .C. ] -> [^b00111101]; // 3, 2
      [ .C. ] -> [^b00110011]; // 3, 3
      [ .C. ] -> [^b00110101]; // 3, 4
      [ .C. ] -> [^b00110001]; // 3, 5
      [ .C. ] -> [^b00110100]; // 3, 6

      [ .C. ] -> [^b01011011]; // 4, 1
      [ .C. ] -> [^b01011101]; // 4, 2
      [ .C. ] -> [^b01010011]; // 4, 3
      [ .C. ] -> [^b01010101]; // 4, 4
      [ .C. ] -> [^b01010001]; // 4, 5
      [ .C. ] -> [^b01010100]; // 4, 6

      [ .C. ] -> [^b00011011]; // 5, 1
      [ .C. ] -> [^b00011101]; // 5, 2
      [ .C. ] -> [^b00010011]; // 5, 3
      [ .C. ] -> [^b00010101]; // 5, 4
      [ .C. ] -> [^b00010001]; // 5, 5
      [ .C. ] -> [^b00010100]; // 5, 6

      [ .C. ] -> [^b01001011]; // 6, 1
      [ .C. ] -> [^b01001101]; // 6, 2
      [ .C. ] -> [^b01000011]; // 6, 3
      [ .C. ] -> [^b01000101]; // 6, 4
      [ .C. ] -> [^b01000001]; // 6, 5
      [ .C. ] -> [^b01000100]; // 6, 6

END
```

Getting Started with the National Semiconductor ispEXPERT System

Now that you have created the subdirectory **DiceToss** and have put the file **DiceToss.abl** into it, you are ready start the ispEXPERT System. You can find it in the Programs menu, under Lattice, after clicking on the bottom left START icon, unless it already has an icon on the desktop. If the program comes up with anything but a blank screen then select **File** ⇒ **Close Project**.

To open a new project select **File** ⇒ **New Project**. In the **Create New Project** window change directories to **DiceToss**. In the **File name** box type **DiceToss.syn**. In the **Project type** box stay with **ABEL/Schematic**. Click on the **Save** button, and you'll be back to the ispEXPERT System Project Navigator screen. Double click on the **Untitled** name and change its name to **DiceToss**. Double click on the **ispLSI1032E-125LT100** chip designation and change to the GAL chip designation GAL16V8/Z/ZD.

You still have to inform the ispEXPERT System Project Navigator of your source file. You do this by selecting **Source** ⇒ **Import**. In the top left **Import File** box type **DiceToss.abl** then click on the **OK** button and you're ready to proceed with the compilation.

Compiling with the National Semiconductor ispEXPERT System

Compiling is very simple. In the left window highlight **DiceToss(dicetoss.abl)** and now highlight the topmost item in the right window. Now click on **Start** to proceed with the compilation. When that is finished repeat for the other items in the right window. If no errors are signaled then all is well and you can proceed to the next step. If errors are indicated during the particular operation then click on the **VIEW** icon to see where the errors occurred. The .abl file can be reedited by double clicking on DiceToss(dicetoss.abl) and recompiling.

If this procedure is successful then highlight DiceToss-vectors and compile in the right window from top to bottom. When you get to JEDEC Simulation

Waveform, you can view it by adding waveforms using the Edit, followed by the Show commands to add waveforms to your display. That way if your chip does not work properly you can look to see which waveform is incorrect and from this determine which equation needs to be corrected.

When all of the above steps are successful you'll have the file DiceToss.jed in your subdirectory and you are ready to program your GAL. To do that, consult the section on Programming GALs and EEPROMs at the beginning of this manual.

If you are convinced that the .abl file has no errors at all then you can take a shortcut and simply highlight the GAL16V8/Z/ZD block in the left window and then select the JEDEC File icon in the right window. Clicking on **Start** will produce the JEDEC file.

The Implementation of the Dice Tosser Design

1. Having read the design example, students can now carry out the steps needed to successfully complete the design of their own dice tosser. Each student will be given a unique dice sequence to implement.

2. Before the final implementation the student should pay attention to the results of the compilation to become convinced that the design is correct.

3. After programming the GAL, the student can use the test jig to demonstrate to the class instructor that the device is indeed carrying out the steps which it was designed to carry out.

4. The GAL wakes up with all D flip-flops set. This means that all Q_n outputs will be high, so no LEDs will light on the dice tosser. A single clock pulse is generated so the dice immediately go to the (0,1) state.

   A second part of the experiment is to modify the design to make that the dice tosser wakes up in the (1,1) state when power is applied to the test jig. This is a good time to review the state table and get a better understanding of the variable A. It is used to cause transitions in the second die.

   The 16V8 GAL is being used in this laboratory to obtain performance identical to the 16R8 PAL. It should be readily apparent from the logic diagram for the 16R8, that each equation can consist of the sum of no more than 8 product terms. If in your redesign one of your equations ends up with the sum of more than 8 terms, then it is possible you did not take proper care of the "don't care" terms in you design. If you are convinced that you did everything right, then speak to your instructor. If the instructor is convinced that your dice sequence cannot be redesigned to start with the (1,1) state, you may get permission to change the sequence cyclically. A cyclic shift moves all the numbers around in a circle by one state, or if need be, by more than one state. For example 1, 2, 3, 4, 5, 6 becomes 2, 3, 4, 5, 6, 1 after one shift or 3, 4, 5, 6, 1, 2 after two shifts.

5. Write the laboratory report in the same style as this text was written. It should describe each step in the design procedure. It can include a critique of this design, with suggestions for its improvement.

Appendix: Program for Selecting Random Dice Face Sequences

The instructor can use this program for generating dice sequences for the students. If you come up with a better program do let us know.

```
'Program to produce state order (dice face order) for implementation
'by student in the Dice Tosser experiment. The current date can be
'used as a random number generator seed.

  INPUT "Any 6 digit number such as todays's date  ", DATE
  RANDOMIZE (DATE)  'Current date used as a seed
  OPEN "dice.out" FOR OUTPUT AS #1

  PRINT #1,
  PRINT #1, "Student #     Dice Face Order"
  FOR J = 1 TO 40  'Adjust for the number of students. Check
                   'output file DICE.OUT for duplicate sequences.

  FOR I = 1 TO 6
    p(I) = 0    'Start with dice face sequence nulled
  NEXT I

  y = 0     'Dice face counter

  DO UNTIL y = 6
    x1 = RND        'Get a random number
    x = 6 * x1 + 1  'Scale it to lie between 1 and 7
    x% = INT(x)     'Truncate the number
    x5 = (x% = p(1) OR x% = p(2) OR x% = p(3))
    x6 = (x% = p(4) OR x% = p(5) OR x% = p(6))
    'Make sure you don't store duplicate numbers
      IF NOT (x5 OR x6) THEN y = y + 1: p(y) = x%
    LOOP

  'Sort the numbers to lead with unity.
```

```
 DO UNTIL p(1) = 1
   z = p(1)
     FOR I = 1 TO 6
       p(I) = p(I + 1)
     NEXT I
   p(6) = z
 LOOP
 'Print them out.

 PRINT #1, USING "    ##"; J;
 PRINT #1, "      ";
   FOR I = 1 TO 6
     PRINT #1, p(I);
   NEXT I
   PRINT #1,
NEXT J
CLOSE #1
```

EXPERIMENT 6: Event Driven Circuit

Objectives

The objective of this lab is to familiarize the student with design techniques for event driven sequential circuits, and to introduce the student to applications involving ROMs.

Preparation

Review chapters 4 through 6 of *Logic and Computer Design Fundamentals*, by M. Morris Mano and Charles R. Kime, Prentice Hall, 1997.

Requisite Equipment

- Pencilbox kit
- One 2816A EEPROM
- One 74374 octal D-type latch (or CMOS equivalent).
- One debounced push button switch to simulate the system clock
- One DIP switch for simulating the inputs
- Three LEDs with current limiting resistors

References

- *Logic and Computer Design Fundamentals*, by M. Morris Mano and Charles R. Kime, Prentice Hall, 1997.

Background

Introduction

Event driven sequential circuits differ from combinatorial circuits in that the outputs of the circuit depend not only on the present state of the inputs but also on the past history of the inputs. Thus a sequential circuit has memory. Memory is provided for each bit needed to define a state by using a bit storage device such as a flip-flop. Any type of flip-flop is suitable, but with some designs the J-K type may be better in the sense that the combinatorial logic required for the feedback path is usually (but not always) minimized by this choice. The combinatorial part of the circuit can be implemented using gates, MUXs or any method that is capable of providing the necessary feedback logic. In our case an EEPROM will be used. This will have the added benefit of familiarizing the student with the procedures of working with these very common (and presently inexpensive) read-only digital-memories. The storage device that will be used will be an octal D-type latch, and it will be seen that the entire design has a chip count of two.

In this experiment an event driven sequential circuit will be implemented. This type of circuit is free running in the sense that the output must respond to an input change in a very short time. It is clearly the input change which drives the circuit, so the circuit is called event driven. Other names commonly used are nonpulse circuits or asynchronous circuits.

A Simple Example

A simple example was chosen in order to acquaint the student with the type of design presented in this experiment. It does not necessarily correspond to anything practical, but was chosen for the simplicity needed to get the requisite familiarity. The state diagram of this system is shown in figure 6.1.

Each state is defined by the two bits WZ and each state produces an output defined by the three bits PQR. The inputs KL needed to make the transitions between the states are indicated on the directed branches connecting the state balloons.

To gain a familiarity with the use of EEPROMs, the two-chip design of the type shown in figure 6.2 was chosen.

The EEPROM contains the combinatorial logic and the edge triggered octal D-type latch contains the needed bit storage. Both devices contain more logic than is needed for this design, so expansion should be possible. It is decided to pass the inputs KL through the latch for two reasons. This way a race condition is avoided by having a stable address input to the EEPROM, and in addition all events can be controlled by the system clock, facilitating the testing of the system. The output bits PQR are derived combinatorially from the state bits WZ. Since the EEPROM has a substantial amount of unused logic we can put

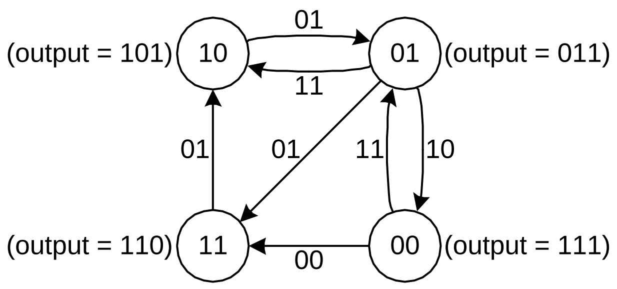

Figure 6.1: The state diagram of a simple event-driven sequential circuit.

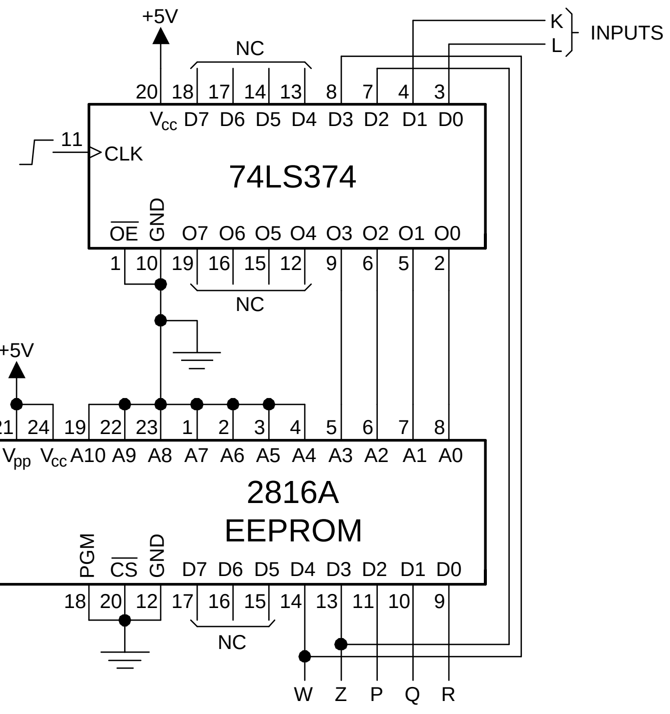

Figure 6.2: Hardware implementation for the simple event-driven sequential circuit of figure 6.1.

Table 6.1: The state table of the simple event-driven sequential circuit.

Present State				Next State				
State		Input		State		Output		
W	Z	K	L	W	Z	P	Q	R
0	0	0	0	1	1	1	1	0
0	0	0	1	0	0	1	1	1
0	0	1	0	0	0	1	1	1
0	0	1	1	0	1	0	1	1
0	1	0	0	0	1	0	1	1
0	1	0	1	1	1	1	1	0
0	1	1	0	0	0	1	1	1
0	1	1	1	1	0	1	0	1
1	0	0	0	1	0	1	0	1
1	0	0	1	0	1	0	1	1
1	0	1	0	1	0	1	0	1
1	0	1	1	1	0	1	0	1
1	1	0	0	1	1	1	1	0
1	1	0	1	1	0	1	0	1
1	1	1	0	1	1	1	1	0
1	1	1	1	1	1	1	1	0

it to good use by using three of the EEPROM output lines for the output bits PQR. We determine the bit storage pattern for the EEPROM by creating a state table as shown in 6.1. Note that the present state and inputs supply the address to the EEPROM, and the EEPROM's data outputs provide the next state and system outputs.

It just remains for us to program the EEPROM to complete the design. To put programs, or data, into EEPROMs we must create a source file which can then be assembled into a .HEX file. This file can then be used for programming of the above devices. The ASCII source code must be created with an editor in *non-document mode*. This means that you have to avoid word processors such as Word and WordPerfect. You can use DOS EDIT or NOTEPAD. Let us suppose that the file that you want to create will be named TEST.ASM. If, when you are finished, you type it to the screen using the DOS line command C:>TYPE TEST.ASM and you get strange looking characters then you haven't created an ASCII file. Your can designate the storage of bytes using the simple DC.B directive appearing in the example file below.

```
        ORG     0
        DC.B    $41,$A3,$3F,$29,$5B,$7D,$09,$64
; As many lines similar to the above as needed
        END
```

Your file will be small because you will be using only a small part of the EEPROM memory. The bits stored in the rest of the EEPROM are irrelevant, and the EEPROM programmer program will make some assumption about them and then go about its business. Once the source file is edited it can be assembled using Antonakos's assembler ASM68K which can be found in the subdirectory \ASM68K. You must be operating in DOS. If you are logged to the A drive then, to assemble, simply type:

```
C:\ASM68K\ASM68K TEST<ENT>
```

When the assembly is finished you will have a MOTOROLA S-type of HEX file TEST.HEX, as discussed in the Microprocessor Course. This can then be used to program the EEPROM using the programming instructions provided in the beginning of this manual under the title **Programming GALs and EEPROMs**. If errors are signaled then look at your TEST.LST file to find out what went wrong.

An Event Driven Annunciator System

The state diagram for the sequential circuit that we wish to design is shown in figure 6.3. It is a two alarm system which might be used in a factory to signal that various levels of faults are occurring on the production line. One application may be in process control, where the fault could indicate a high pressure. A flashing amber light would indicate the first stage of pressure change, a potential hazard. A large change in the system, shown by a flashing red light would indicate an emergency condition.

The annunciator has 6 proper states, so we will need a 3 bit sequential circuit. Each state is described by the 3 bits UVW. The annunciator monitors the factory for the presence of a low level alarm signal LLA and a high level alarm signal HLA. The high level alarm signal HLA cannot occur if the low level alarm signal LLA has not occurred first. The state diagram has the states labeled A through F. The instructor will assign the binary code of each state to every student.

The operation of the circuit is best described by the state diagram of figure 6.3. The circuit operates in the following manner:

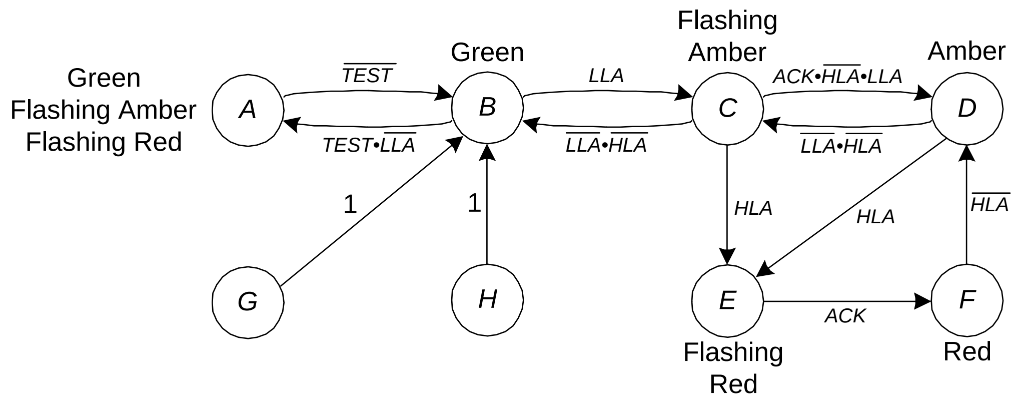

Figure 6.3: State diagram of the annunciator circuit.

1. With no fault-signal present the system is stable, it is in state B, and the GREEN light is on.

2. When the signal LLA is present, indicating a minor fault, the state changes to C, a FLASHING-AMBER light comes on and the GREEN light goes off. If the fault disappears ($\overline{LLA}$), the annunciator returns directly to the normal GREEN state.

3. When the system is in the minor-fault (FLASHING-AMBER) state, an operator can intervene to clear the minor fault by pushing the acknowledge button which contains a momentary contact switch. The presence of the ACK signal for a mere fraction of a second changes the system to the steady-AMBER state, telling supervisory personnel that someone is trying to clear the fault. If the minor fault is cleared ($\overline{LLA}$) then the annunciator returns to the normal (GREEN) state after 2 clock pulses.

4. If the system is in the AMBER or FLASHING-AMBER state and the major fault signal (HLA) is received, the system signals a major fault by changing to the FLASHING-RED state. Even if this signal is only momentary, this condition will be maintained indefinitely. The presence of the ACK signal for a mere fraction of a second causes a transition to the steady-RED state, telling supervisory personnel that someone is trying to clear the major fault. If the major fault is cleared ($\overline{HLA}$), the annunciator starts on its path to the normal state and finally attains it if there is no low level alarm ($\overline{LLA}$).

5. A test pushbutton is included to check the condition of all the lights. On $TEST$, the GREEN, FLASHING-AMBER and FLASHING-RED lights should be on. This is the A state.

The Experiment

1. All the students should obtain their own unique 3-bit designation for the states $ABCDEFGH$ from the class instructor.
2. Having read the design example the student can now carry out the steps needed to successfully complete the design of the event driven annunciator system. Each student is responsible for prototyping the design for testing purposes.
3. Demonstrate to the class instructor that the device implements the design specification.

State order program

The instructor can use this program for generating the state order for the students. If you come up with a better program do let us know.

```
‘Program to produce state order for implementation by student
‘in the Event Driven Asynchronous Circuit. The current date can
‘be used as a random number generator seed.

  INPUT "Any 6 digit number such as todays’s date  ", DATE
  RANDOMIZE (DATE)  ‘Current date used as a seed
  OPEN "EEPROM.OUT" FOR OUTPUT AS #1

   PRINT #1,
   PRINT #1, "Student #     State Sequence Order"
   PRINT #1, "             A  B  C  D  E  F  G  H"
  FOR J = 1 TO 50  ‘Adjust for the number of students.  Check
                   ‘output file EEPROM.OUT for duplicate sequences.
  FOR I = 1 TO 8
    p(I) = 10   ‘Start with  sequence set to impossible numbers
  NEXT I

  y = 0     ‘Event sequence counter
  DO UNTIL y = 8
    x1 = RND        ‘Get a random number
    x = 8 * x1      ‘Scale it to lie between 1 and 9
    x% = INT(x)     ‘Truncate the number
‘Check to see if x% is equal to any of the previously assigned #s.
‘Unassigned #s are all 10, so that shouldn’t be a problem.
    x5 = (x% = p(1) OR x% = p(2) OR x% = p(3))
    x6 = (x% = p(4) OR x% = p(5) OR x% = p(6))
    x7 = (x% = p(7) OR p(8))
    ‘Make sure you don’t store duplicate numbers
      IF NOT (x5 OR x6 OR x7) THEN y = y + 1: p(y) = x%
    LOOP

‘Print them out.
   PRINT #1, USING "    ##"; J;
   PRINT #1, "      ";
    FOR I = 1 TO 8
       PRINT #1, p(I);
    NEXT I
     PRINT #1,
 NEXT J
 CLOSE #1
```

EXPERIMENT 7: Microsequencer Design

Objectives

To learn the basic operating principles of microsequencers.

Preparation

Review Chapter 7 of *Computer Systems Organization and Architecture*, by John D. Carpinelli, Addison Wesley, 2001.

Equipment needed

- Pencilbox kit
- One 2816A EEPROM
- One 74298 quad 2-to-1 multiplexer with latched outputs
- One debounced push button switch to simulate the system clock
- Two DIP switches
- Three LEDs and limiting resistors
- Additional 74-series ICs based on the student's design

References

- Any textbook used in the prerequisite computer systems architecture course.

Background

Microsequencers

The control unit is responsible for coordinating actions within the CPU. The control unit decodes the instruction and issues a set of control signals that result in the instruction's execution. These control signals cause data to be routed correctly within the CPU, generate correct external control signals, such as $\overline{RD}$ and $\overline{WR}$ in the 8085, and cause the ALU to perform the correct operation on its data.

There are two methods used to generate these control signals. *Hard-wired* control units implement the control signals using combinatorial logic. The inputs to this control logic are generated by the control unit and also taken from the instruction opcode. *Microprogrammed* control units, or *microsequencers*, are another approach. In this method, control signals are the output of *control memory*; the control unit generates the control signals by stepping through the correct sequence of control memory locations. Each word of control memory is called a *microinstruction*. The sequence is initiated based on the instruction to be executed; the next microinstruction is generated from the current microinstruction and external conditions. The general configuration is shown in figure 7.1. The sequence of operations is as follows:

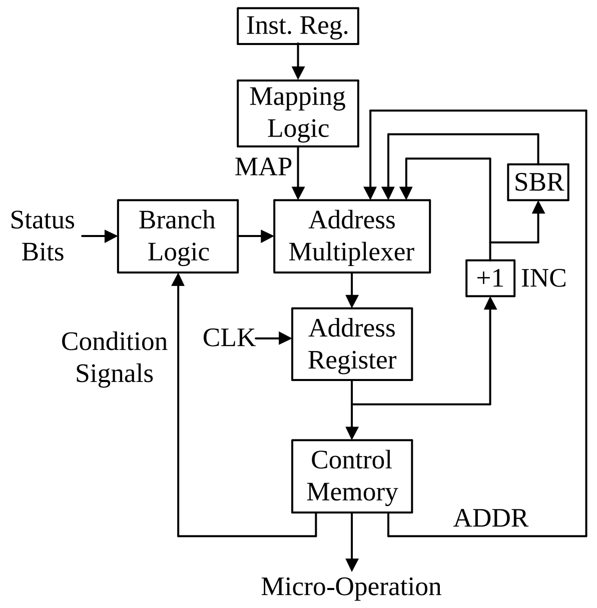

Figure 7.1: General microsequencer configuration

1. The address of the next microinstruction to be executed is loaded into the address register. This becomes the input address to the control memory.
2. The control memory decodes its input address and puts out the microinstruction. This microinstruction has two functions. First, it makes available the control signals to sequence execution of the instruction. Note that the microsequencer does not actually execute the instruction; it generates control signals which cause other hardware within the CPU (such as the ALU) to execute the instruction. Second, it generates address and condition signals.
3. Based on the condition signals, the multiplexer generates the address of the next microinstruction. Possible next addresses are discussed later in this experiment.

Address multiplexer and mapping logic

The address multiplexer is used to select one out of a number of possible addresses for the next microinstruction to be executed. There are four standard possibilities.

INCR The next address in control memory. This is often used to execute a series of microinstructions that occupy consecutive locations in control memory.

ADDR The address specified by control memory. This is used to specify jumps within control memory. This is typically used to jump to the opcode fetch routine when finished executing the instruction. It is also combined with INCR to perform conditional jumps within the microcode, i.e. if the jump is taken, the next microinstruction address is taken from ADDR, otherwise it is taken from INCR.

SBR The address stored in the microsubroutine register. Just as high-level and assembly language programs may have subroutines which can be invoked from different locations within the program, microcode may also use microsubroutines. These microsubroutines usually perform functions, such as indirect addressing, that are common to many instructions. The microsubroutine register contains the return address of the microsubroutine. The register is loaded when a microcall is executed and is read when a microreturn is reached. If nested microsubroutines are used in the system, the register must be replaced with a hardware stack.

MAP The address based on the instruction to be executed. The mapping logic reads the opcode of the instruction to be executed and generates the microaddress of the first microinstruction used to execute the instruction. The logic used to generate the output depends on the placement of microcode within the control memory.

The address to be used is determined by the *branch logic*. This is combinatorial logic that selects the next address based on the current microinstruction and external status bits. The branch logic also generates the signal used to load SBR.

In this experiment, INCR and SBR are not used; only ADDR and MAP can supply the next microaddress. Furthermore, all branches are unconditional, so the branch logic defaults to a single control signal derived from the control memory.

Microaddress register

This is a single register which holds the address of the microinstruction. The clock used to load the register is actually the system clock.

In this experiment, the register is incorporated within the multiplexer chip. Instead of the oscillator normally used as the system clock, you will use a **DEBOUNCED** push button switch to allow you to single step through the microcode.

Control memory

The control memory is typically a small ROM. For this experiment, you will use a 2816A EEPROM. You will enter the microcode using the DB assembler directive. Ground all unused address lines.

Other hardware

For this experiment, two additional pieces of hardware are needed. The instruction register, which normally holds the opcode for input to the mapping logic, will be replaced by two DIP switches. This is done to simplify hardware and facilitate debugging. Also, three LEDs with current-limiting resistors will be connected to signals Z_1, Z_2 and Z_3 of the control memory. Since this experiment does not actually use the other components of the CPU, the LEDs will be used to monitor the control signals. Note that all LEDs must be driven by active low logic.

Control memory format

All eight outputs of the control memory are used in this experiment. The format is as follows.

Bit	Function
D_7	Address Multiplexer Control
	0=Address from ROM, 1=Address from MAP
D_6	Control for LED Z_1, active low
D_5	Control for LED Z_2, active low
D_4	Control for LED Z_3, active low
$D_3 - D_0$	Address from ROM, ADDR

There are three instructions which this control unit can execute. They are summarized below.

Instruction	Opcode	LEDs lit during			
		T_1	T_2	T_3	T_4
I_1	01	Z_1	Z_2	Z_3	Z_1, Z_2, Z_3
I_2	10	Z_1, Z_2	Z_2, Z_3	Z_3, Z_1	Z_1, Z_2, Z_3
I_3	11	Z_1	Z_1, Z_2	Z_1, Z_2, Z_3	None

Each of these instructions must have its own microcode routine.

A fourth microroutine is needed to implement the opcode fetch. This takes four microinstructions. The first three microinstructions would normally sequence the fetching of the opcode from memory; since this is not done here, use null code which steps through these microcode locations. The fourth microinstruction must map to the correct microroutine. All LEDs must be **OFF** during the opcode fetch. If the opcode fetch reads an opcode of 00, the microcode should map back to the opcode fetch routine.

Pre-lab Assignments

1. Design the microcode layout, i.e. which instructions reside in which ROM locations.
2. Based on your microcode layout, design the mapping logic.
3. Based on the microcode specifications given in this experiment and the microcode layout, write the microcode.

Experimental Procedure

1. Burn the microcode into the 2816A EEPROM.
2. Wire and test the circuit. Use the debounced push button switch for the system clock, and the toggle switches for the instruction register. Test the system and demonstrate it to the instructor.