# How To Use This Book

The Brain Of The Computer is intended to provide you with a thorough explanation of basic computer operations. Computers can be hard to understand. Even some technical people have trouble doing so.

Almost everyone works with applications. But that requires only a limited degree of technical knowledge. Few people really understand or appreciate the essential underlying processes that make a computer function. I've divided the material into segments that will highlight the various elements of basic computer design and teach you how they work.

Near the end of the book - perhaps when you're doing the test programs - a light bulb should come on. You may even jump out of your seat and say, "Wow! I get it now!"

Resist the urge to skip material or you might deny yourself that "Eureka!" moment.

JAMES BUCHANAN

# Introduction

You might think, "I know to use Microsoft Windows®.  What more do I need to know?"

Well, whether you're going seek a career in the electronics industry or even just repair anything that contains any digital electronics, a good understanding of the basics will be crucial to your success.

Before a doctor can treat a patient, an accurate diagnosis must be made – one that's based upon a sound, comprehensive knowledge of how the human body works. And the same is true with electronic and computing devices.

In the 1980's, pinball machines evolved from making logic using relays to making logic using transistors.  We've seen the same situation develop in the field of automotive mechanics. Unlike the much simpler cars of years past, today's vehicles may contain as many as 20 separate computers.  Anyone who can't keep up with these developments isn't going to last long in that field.

It's not enough to know what a hard drive is, how to move files, how to access the help section and the many other ways you can use computers to your advantage. In truth, most people have no idea how a computer really functions.  The real operation is embedded deep in the motherboard, in the micro-processor, and the machine language that controls it.

Many who enroll in college to study computer science drop out because they're overwhelmed with information before they can really grasp the basics of how these important pieces of the puzzle interact:

Electricity
Binary numbers
Digital Circuits
Micro-processors
Instruction data
Program counter
Random Access Memory (RAM)
Read-Only Memory (ROM)
Digital logic
Machine language
Higher level computer language
Display circuits
Programming

In college, dropout rates for Science, Technology, Engineering and Math (STEM) majors, including, those studying computer science, hover around 50 percent (Rogers, 2013). Many of these students, although intellectually capable of learning the content, leave before they understand the fundamentals of all the related pieces. Even some who do make it through and enter the field professionally continue to function without a thorough grounding in basic computer operations.

In this book, I'm not going to weigh you down with a lot of details. This will make it easier for you to put the computer puzzle together.

The "mini-computers" of the 1960s and 1970s, had discrete components such as transistors, resistors and diodes mounted on circuit boards. Technology really advanced when those components could be put on a single part called the integrated circuit (IC). Computer engineers then used integrated circuits to simplify their design.

We are going to study and use the 6502 processor. Advances in metal oxide semiconductor technology in the late 1970s made it possible to take the entire 6502 circuit and put it on one IC chip. But the easiest way to understand the 6502 processor is to explain it with single logic ICs.

You might be wondering why this book focuses on a 6502 8-bit computer system, which is old technology. This book explains the 6502 and it's code because it's easier to learn than the newer central processing units (CPU). Hobbyists love the 6502 processor and are still programming today on old 8-bit computers.

Why would you want to learn 6502 assembly code? Because most other languages do not allow you to see what the computer is really doing. An example of this is the HTML language. HTML allows you to make web pages by using keyboard character commands to make structured documents with images, text, paragraphs and lists. Just a few lines of code in HTML can generate hundreds of machine codes. If you want to understand and control every step of the code, you must program in assembly language. Assembly code is basically the same as Machine code, but it has added features that make it easier to use.

Through the many years of working in the computer industry, I have pieced together all the important information that makes a computer operate. This information when learned, will allow the reader to completely understand a computer. I have simplified the process by:

1. Explain the basic design in the first few pages of this book with as little technical information as possible. I have taken a complicated issue that people are somewhat familiar with, and put it into simple terms. The reader won't need to wait until the end of the book to have a basic understanding of computers.

2. Describe only the most important details. This book is not meant to replace a college course, but to prepare the reader to understand what he is going to learn in that college course. The hobbyist who isn't taking a college course can use this information as well to increase the knowledge that they already know.

3. Provide hands-on computer code using an engineering-type computer development system with a monitor (web simulator). Using the hands on action of typing code into a simulator and seeing the results will give the reader a visual that will help to reinforce computer concepts.

Hard-to-understand concepts will be slowly introduced throughout the book, sometimes repeating them in different ways or comparing them to simple circuits or mechanical machines. The most important thing you'll find here that other books seem to avoid is a complete explanation of the binary numbers that computers use.

This book will provide three levels of computer understanding. The first is demonstrated by the example of someone purchasing a can of beans in a grocery store. Hardware and programming is explained at its most elemental level. Components are described in very simple terms. Programming is just a general idea. Much more is going on with a grocery store computer than is discussed.

The second level of understanding involves the Etch-a-Sketch® game. Components and hardware layout are explained in a little more detail and some new concepts are discussed that were not in the grocery store computer. Simulator instructions are demonstrated by a simple routine. Seeing the code at work will increase the reader's understanding.

The third and highest level begins after Chapter 2 (Etch-a-Sketch®). Hardware, discrete electronic components, digital circuits, timing circuits, and programming are explained to a much greater extent.

Essentially, the first two chapters are meant to be light, and the rest of the book is heavy. Many components described in the book are explained three times, with the level of complexity increasing as you gain greater understanding. The last chapters are the most detailed and are meant to be understood only after all the test routines are executed and you have reviewed earlier parts of the book a second time.

Finding out how something works always leaves me with a feeling of exhilaration. I hope that the information I've put in this book has the same affect upon you.

# Table Of Contents

# Chapter 1

## Going To the Grocery Store

Computers have changed how we do many things in our lives. But do computers change basic processes? Let's look at a grocery store and see how things were before computers and how they are now.

Before computers, grocery store customers would bring an item, such as a can of beans, to the check-out. The cashier would need to determine the price from the price label, enter the price into the cash register, calculate and add tax, total the bill, collect the money from the customer, calculate the customer's change, and count the change back to the customer to complete the transaction. If the beans did not have a price label, the cashier would need to look up the price from a price list. The calculations for tax and change would be done in the cashier's head, or on scratch paper.

Later, a stock clerk would need to count how many cans of beans were left on the shelf and in the stockroom, determine when to reorder, and add that information to the order list. When fresh stock arrived, a manager or stock clerk would need to calculate the retail markup on the beans, update the price list, and apply the price labels to the cans before stocking the shelves. The cash register – a mechanical device – was the only automated part of the process.

Now let's fast forward 50 years or so to see what happens. The customer takes the can of beans to the checkout counter. The cashier scans the bar code. The item name, manufacturer and price pops up on a display. If it's a sale item, the discounted price is automatically calculated. The customer gives the cashier the money, and the cashier types in the amount received. The computer then tells the cashier the correct amount of change to give to the customer. At that same time, the inventory and reorder lists are updated. In large corporate grocery stores, the reorder list goes automatically to the warehouse over the internet.

So if the basic processes of choosing an item, making a purchase, totaling the sale, making a payment, and tracking stock has not changed significantly, how does the consumer benefit from computerization? Here are some of the ways: customer checkout is quicker, pricing errors are minimized and overhead, including the number of employees is lowered, which greatly decreases the price of the items. In addition, outdated food gets taken off the shelf more quickly, and restocking is more accurate and efficient.

Now, let's take a look at what goes on "behind the scenes" as the computer performs all of those operations.

The first and most important thing for you to learn is that computers only use binary numbers. With decimal numbers, every tenth count adds a digit. But in binary, adding digits happens more often and they're added to the left, not the right. The book is going write binary numbers in two different ways. The first is with ones and zeros and the second is with hexadecimals, which is a series of letters and numbers. Hexadecimals are easier to use, because they don't have so many digits.

Every digit of a binary number when used in a computer is a one or zero. The common voltage for a zero is 0 volts and the common voltage for a one is 5 volts, but other voltages can be used. In this book, a zero is represented by "0" and a one is represented by "1".

The following shows you three kinds of numbers. Each of the three types of numbers below, have the same number value.

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

The second most important concept is all information that goes into a computer is converted into binary numbers. Groups of numbers that are used for a particular purpose must be kept separate and be easy to find. The main input device is the keyboard. The binary information within the computer must be converted back to letters and numbers in order to be displayed or printed.

Another of the important concepts is that a computer is basically a number machine. The other number machine that you might be familiar with is a programmable calculator. By adding the following parts and functions to a programmable calculator, you create a computer:

1. Keyboard
2. Monitor or printer
3. More memory
4. Programmable code that can be typed into memory
5. More functions (instructions), like compare, move, input, output, locate and store.
6. Instructions are executed sequentially by adding a part called the Program Counter. The Program Counter controls the order of the instructions.

The last computer concept is the functionality of the motherboard. For all practical purposes, the computer is separated into two parts: CPU and motherboard. The CPU performs the instructions and outputs numbers to the motherboard, where they can control things such as a monitor screen, audio circuit, and a USB bus. Numbers are also transmitted to motherboard connector slots, where controller circuit boards can plug in. As you can see, these components are being controlled by binary numbers from the CPU. Memory devices are also located on the motherboard.

Since a computer is a number machine, you'll need to know how to convert decimal numbers to binary and hexadecimal. It's also important that you keep separate in your mind the three uses and categories of binary numbers which appear to be the same but are totally different from one another. These categories are address, data and instruction. From here on, the book will try to stay with hexadecimal numbers to minimize confusion. Hexadecimal and binary numbers can represent the same number value, but they are written differently.

Numbers that can be changed are stored in Random Access Memory (RAM). Numbers that stay the same can be in "Read only Memory" (ROM). A hard drive works like a RAM or a ROM. Because hard drives are slow to read or write, the computer uses a RAM and a ROM as much as possible to speed things up.

**Computer Concepts and Terms**

Throughout this book, various concepts and terms will slowly be added. You will need to know these concepts and terms to understand this book.

You'll recall that in the non-computer bean transaction at the grocery store, the most important part was the cashier. The cashier decided what to do and the order in which to do it. "What to do" is called instruction. The binary coded numbers that cause it to be done are called "Operation (OP) code". The part that controls the order of instructions is the "program counter". The program counter is almost always sequential. That means the next thing you want do has been written next in code.

**Loops**

Loops are a computer code routine that is constantly looking for something to happen. Often, the computer has no idea when you want to do something. That's the main reason for loops. Keys are one example. In less than $1000^{th}$ of a second, all keys can be looked at to see if one has been pressed. It is hard to realize that a computer can only perform one function at a time, because it operates so fast.

**Display**

I am going to refer to a display that will be discussed later in this book. For now, I'll use it to help illustrate principles in a grocery store display. The main display diagram is the "Pixel Chart" (Figure 1). All video information you see on a display is made up of dots called pixels. Pixels can be on or off and have location numbers. A series of pixels make up all the letters and numbers on a display. For example, the number 1 would be five pixels and the letter E would be 11 pixels, as shown on Figure 1 of the Pixel Chart. Using the pixel chart, the shapes for all letters and numbers can be defined. This information is called font information and is stored in permanent memory. When writing letters or numbers on the screen, the computer sends binary numbers that are coded for letters and numbers to the font list. The font list then supplies the pixels that are needed to make the shape of the letter or number. You supply where the letter or number with all its pixels will be located on the screen.

**CPU**

The CPU (the brain of the computer), which is also called a micro-processor, is a single part that contains computer timing, the program counter, registers and the arithmetic logic unit (ALU).

**ALU**

The ALU performs addition, subtraction, multiplication, division, and several logic functions.

**Computer Timing Circuit**

The computer timing circuit tells binary number data to move from place to place. It also tells registers to input or output, the program counter to increment and the ALU what function to perform and when to do it.

**Registers**

Registers are small memory devices inside the CPU that hold a two-digit hexadecimal number for further processing. Registers can increment, decrement, hold a number for an addition or subtraction, and can input or output through the data bus.

**Program Counter**

The Program counter, using the address bus, locates the instructions in program ROM and sends them to the CPU using the Data bus.

**Address Bus**

The address bus transfers location numbers to the motherboard.

**Data Bus**

The CPU uses the data bus to transfer data numbers to and from the motherboard.

**Motherboard**

The Mother board contains ROM, RAM, hard drive, display circuit, key board, and barcode scanner input.

**Decoder Timing Control**

All circuits on the motherboard need to be accessed and turned on one at a time when needed. That is the purpose of the decoder timing control.

**ROM**

A ROM is a permanent binary number storage device. The stored binary numbers are instruction or data. The CPU, using the address bus, selects a location in the ROM. The data bus receives the stored number and sends it to the CPU.

**RAM**

The CPU can only read a number from a Rom, but the CPU can write a number into a Ram and can read it out.

**Hard Drive**

Hard drives can read or write numbers. They have a larger storage capacity than a RAM or ROM, but they read and write slowly because they a mechanical arm inside that moves like in a record player. Current hard drives often have no moving parts, which improves their read/write times.

This is only a very preliminary explanation of hardware terminology. It's enough to get you through this chapter but not sufficient for complete understanding.

We are now ready to see what this dumb computer machine can do. I call it "dumb" because you can do anything it can do, and often you can do it better. The computers advantage is, it is much faster. When you're a cashier people can be very grumpy when you're slow. The computer will make you fast, and customers will love you.

Let’s go back to the can of beans analogy with the can of beans at the check-out counter. All keys output a particular binary number when pressed. The computer is now doing a looping sequence of compare number instructions, waiting to see which key will be pressed. The price list (with barcode data attached to each item), and the font list are in ROM memory.

When the hand held barcode scanner trigger key gets pressed, the CPU sees the scanner key number and jumps to the load "A" register instruction to put the barcode data into the "A" register. The next CPU instruction says compare "A" register data with the contents of the selected memory address. The selected memory address location is the first item in the price list. The CPU is looking for the barcode bean number on the price list. The next series of instructions increments the price list memory location to keep reading items down the list until it finds the matching barcode bean number. Once a match is found, the CPU will put the location of the bean price into the X register. The price now needs to be displayed. The monitor will display "Acme Beans 85 cents". The pixels that are needed to display letters and numbers are in the font list. A series of load and store instructions will go to the price list, locate the bean message data using the X register and convert that data to letters and numbers using the font list. The letters and numbers then go to the display for the customer to see.

After the message is written, the CPU will go back to the key looping routine, but this time it will be looking for the number that is coded for the dollar sign key. Remember - every key is a particular binary number. The cashier pressing the dollar key on the keyboard will tell the computer that you are typing in the money the customer is giving you. The customer hands you the money and waits for change. The next series of instructions stores those typed-in money numbers into a RAM memory location. So you have the money he gave you in a RAM location and a bean price in a ROM location. The next series of instructions subtracts the contents of one location from the other to find the customer's change, using the ALU in the CPU to do the calculation. The CPU executes the next instruction which puts the change amount on the display. The cashier gives the customer their change, and the transaction is complete.

As you can see, the computer is doing the same thing that the grocery clerk did in the past. Actions that are the same as before computers are:

1. Choosing the grocery item
2. Looking up the price from the price list
3. Printing the item name and price on a receipt
4. Counting, adding, and subtracting to calculate the total sale and return change.
5. Creating inventory and re-order lists.
6. Dealing with sales tax and discounted items.

The difference in those processes is the computer. Many of the manual tasks - such as figuring tax, calculating change, and tracking inventory - are now handled automatically by the computer. The entire sequence of accounting in a sales transaction is now handled by a computer, using binary coded numbers to know what to do next. Except for actually choosing the grocery item, the computer now handles every step of the process.

Some people rail against computers without realizing how many ways they have made our lives easier. The Chinese used an abacus to control grocery store transactions, with beads sliding on wires to account for goods and money. It was a good idea at the time, but who would want to go back to it now?

We have covered a lot of information in just a few pages. The rest of the book will slowly explain in more detail what has been said. If you got the general idea of what you've read, you are doing really well.

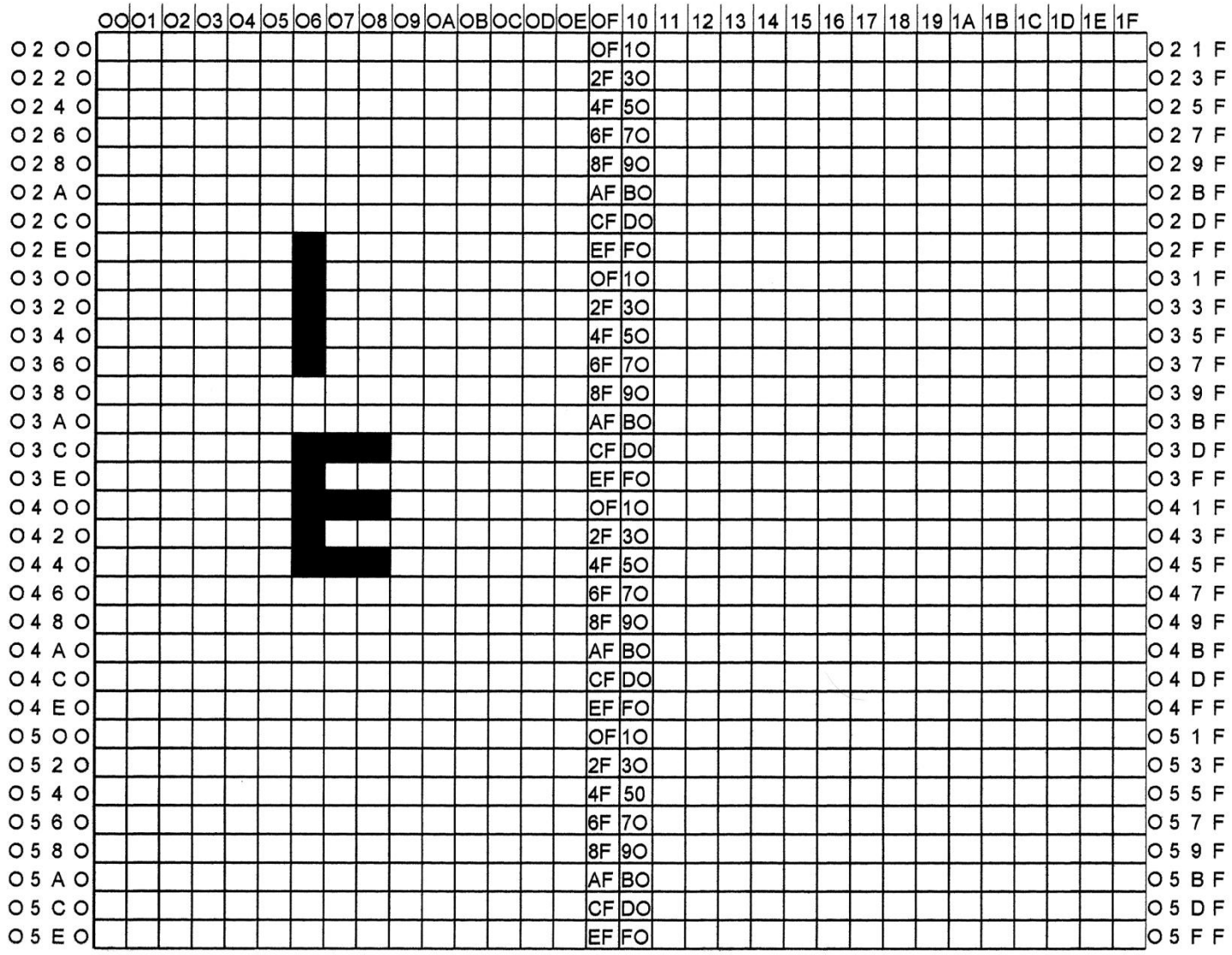



**Figure 1**. Pixel chart

## BLOCK DIAGRAM

The block diagram is used to illustrate the layout of the computer parts and the path that binary numbers take as they move from part to part. The other illustration is the schematic diagram. Both are from the same computer system. Parts inside the CPU are shown on the block diagram, but not on the schematic diagram.

The block diagram is broken down into the CPU side and the motherboard side. Dashes show the separation. The CPU is a single component containing many parts inside. Binary numbers need to go back and forth and to and from all CPU and motherboard parts.

The CPU puts location numbers on the address bus and sends them to the motherboard. Trace out the address bus to see where it goes on the diagram. The data bus sends and receives numbers from the CPU and motherboard. Trace the data bus also.

When the computer executes an instruction, CPU timing and control signals are used to control the parts that move the numbers around in the CPU for the instruction. The CPU timing and control signals on the block diagram only show one line to a CPU part, but that one line represents more than one timing signal. The CPU parts, including most registers, can input, output, decrement and increment. That would be four timing control lines, but only one signal line is shown. This is normal drawing procedures for block diagram drawings.

The address bus comes from the CPU to the ROM, RAM, hard drive, display circuit, and decoder timing control. The purpose behind of all these parts receiving and sending binary numbers will be explained when we do the programming. At this point, you are not going to totally understand the block diagram. I just want you to become familiar with it. Whenever you do test code in this book, look at the block diagram and trace the path of data.

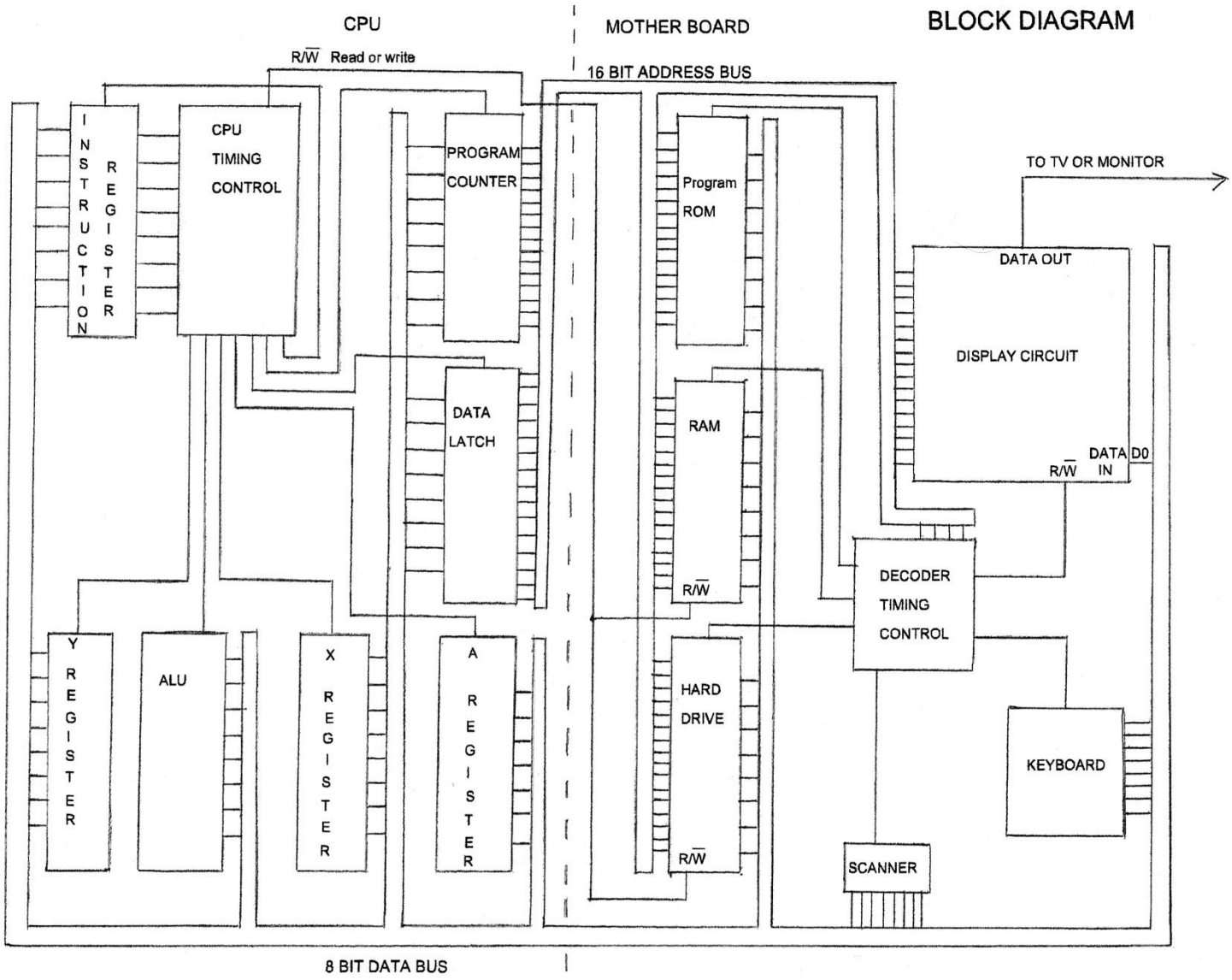



**Figure 2.** Block diagram

## SCHEMATIC DIAGRAM

Explaining the motherboard schematic and the code that controls it are fundamental to this book. This total circuit is the basic idea of all computers. The only things that need to be added are more input and output circuitry, which could be a USB integrated circuit chip connected to the data bus and the outside world. Note that the CPU is one part, and everything else is the motherboard. Individual address lines, data lines, and sync signal lines are not drawn in, because there would be too many lines on the diagram. The memory map on the diagram is a circuit number or a range of circuit numbers for each motherboard circuit. The CPU needs a circuit number to be able find and turn on a mother board circuit.

Just like the block diagram, the motherboard schematic and parts are not going to be fully understood until the end of the book. All hardware and software needs to be explained and the computer code must be demonstrated on the web simulator before you will able to understand everything. Also, the motherboard schematic, except for the missing keyboard, is the exact total hardware circuit that our web simulator is simulating. The Simulator will be used in the next chapter and will be explained in more detail in chapter 9.

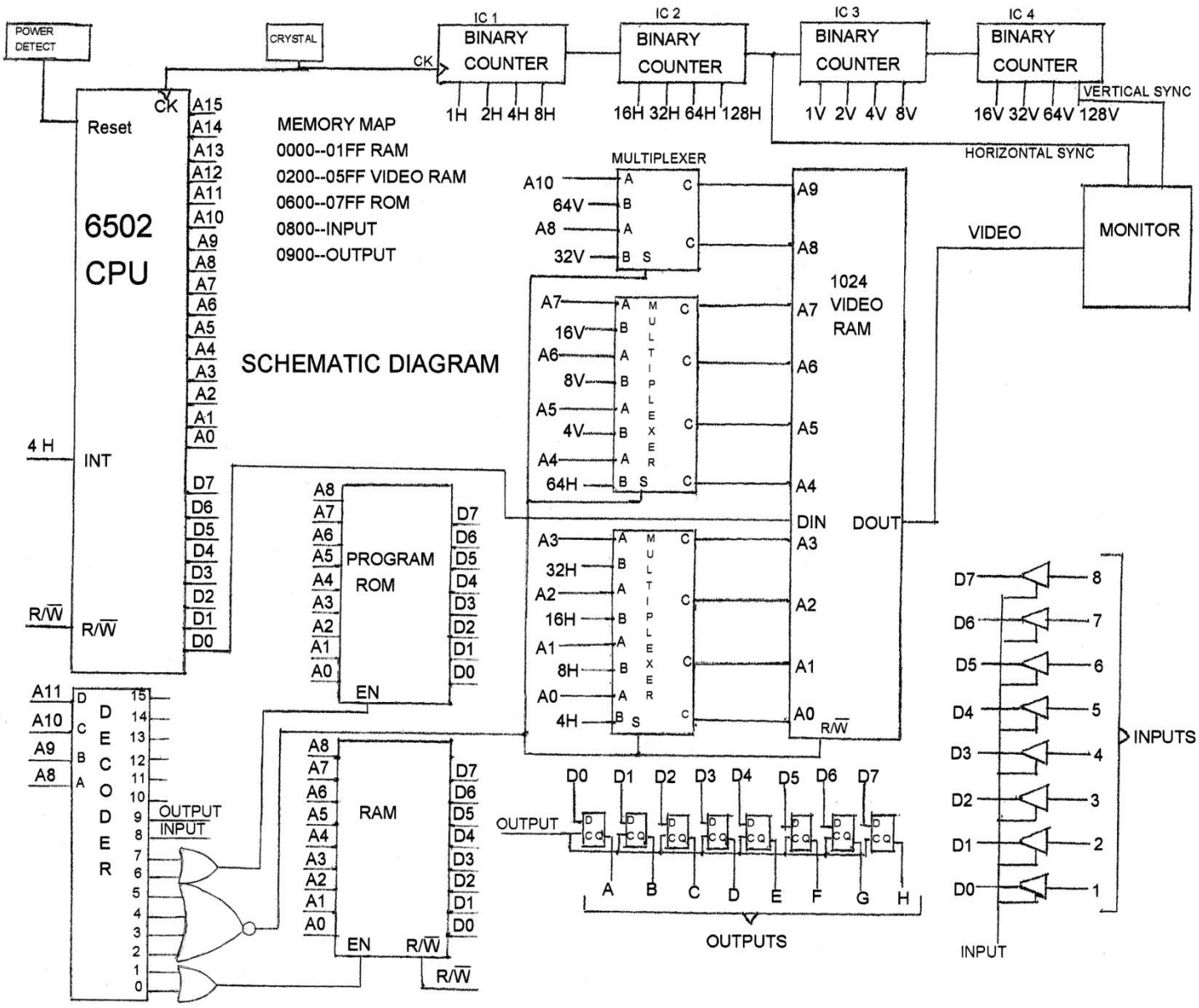



**Figure 3.** Schematic diagram

## MOTHERBOARD PARTS

Look at the schematic diagram. Again, we're getting familiar with the layout and the use of its parts. Motherboard parts will be explained in more detail later in the book.

### ROM

A ROM has binary numbers stored in it which are instructions and data. The address bus selects a location in the ROM and the data bus receives the stored number. A ROM only turns on when selected by the CPU, using the decoder on the motherboard to enable the ROM.

### RAM

A RAM operates the same as a ROM, except the CPU can write numbers into it and read numbers out of it. An example of using a RAM is incrementing a number. The CPU will take a number out of a RAM location, increment it, and put it back in.

### Input

The CPU, using the decoder enables eight input buffers to input numbers from the outside world.

### Output

The CPU, using the decoder enables eight flip-flops (a flip-flop stores a "1" or "0") to output numbers to the outside world.

**The crystal clock circuit** – A crystal clock has a thin quartz wafer that vibrates when energized with electrical power and outputs an oscillating signal. That signal is converted to a square wave and is used to run our CPU and our video binary counters. The thickness of the quartz wafer determines the frequency of the square wave clock. Quartz crystals also run digital watches and digital clocks.

**Binary counters-** ic1, ic2, ic3, and ic4 are used to count out the pixels that are stored in the video RAM. That pixel data is sent to the monitor to be displayed.

**The power detect circuit-** goes low "0" for one second every time computer power is turned on. That resets the CPU (6502 micro processor) to get a starting program address from two of the last four addresses in ROM. That starting address starts your program code when you power up your computer.

## Motherboard Decoder

The decoder circuit uses a binary to decimal decoder. Four of the upper addresses from the CPU will put out a binary number from 0000 - 1001. Each of those numbers will allow one of the decoder outputs to go high ("1"), which turns on one of the five circuits that the CPU can access on the Mother Board.

## Two input Multiplexer and Video RAM

There are three, two input multiplexers. When the select S signal on the multiplexers is "0", addresses from the CPU on A inputs go to C outputs. When the select S signal is "1", sync signals from the binary counters on B inputs go to C outputs. These two functions of the multiplexer are: 1. Using CPU addresses, select locations in the video RAM to write in pixels. 2. Using the sync counter signals, count out all the pixels in the video Ram and send them to the monitor to be displayed.

## Display Circuit

The monitor paints a picture by scanning a line from the upper left to the upper right. Then, one line down, it scans another line. It does this until the entire screen is filled. The horizontal sync signal starts the horizontal line at the left and stops the line at the right. The vertical sync starts the lines at the top and ends the lines at the bottom. The bright parts of each horizontal line are the pixels that are read out of the video RAM.

**6502 Micro Processor**

The 6502 is an 8 data bit 65536 address computer. It can take a number from a RAM or ROM and put it into one of its registers. That register number can then be added to another number, subtracted, compared, tested to determine if a bit is high, it can be incremented, decremented, stored to a location, and many other options. Instructions are needed for the CPU to perform those functions. The Program Counter finds the instructions in program ROM and sends them to the CPU to be executed. Note: There is a lot more to know about CPU operation.

# Chapter 2

## Making An Etch-A-Sketch® Game

Intel© made the first micro-computer. That wasn't their plan, but in effect that's what they did. Intel© originally created several different calculator models. The digital circuitry was different for every model. One of Intel's engineers got the idea to design one digital circuit for all of the calculators. That circuit had to be a computer. From that point on, engineers would just change the programming as new models were released. Of course, it wasn't long before their CPUs became more popular than their calculators.

Do you remember the Etch-A-Sketch® game? It's a framed screen filled with aluminum powder, and it's operated with two knobs that work together to move a plotter system around the screen to create images. To demonstrate how the functionality of a computer can change, this chapter describes programming it to function like the Etch-A-Sketch® game. The process of what you will do is explained here. Later in the chapter, you'll find out how to access and use the 6502 simulator to build and test the code.

The Etch-A-Sketch® game, when used on a computer, has four keyboard buttons: one for left, one for right, one for up, and one for down. Computers use pixels to make letters, numbers, pictures, video, and a moving line like in this game. A pixel is a dot. If you make a circle with large dots, it looks boxy. If you make a circle with many small dots, it looks normal. With high definition, the dots are so small you need a magnifying glass to see them. You are going to use big dots, so we can see what's happening.

The display we're going to utilize has 32 by 32 possible dots. If all 1024 dots were turned on, they would turn the whole screen white. A single dot can have 1024 different possible locations on the screen. The display location numbers in decimal starts with 0 at the upper left of the screen and ends with 31 at the upper right. Moving down one row, the count continues with location 32 at the left and ends with 63 at the right. So after every 32 counts, it moves down one vertical location until it gets down to the bottom of the screen. The completed screen has 32 vertical locations for each of the 32 horizontal lines. Please note that the pixel chart diagram (Figure 1) does not have decimal location numbers. It has hexadecimal and 20 in hex equals 32 in decimal.

Getting back to the game, you have four buttons, left, right, up, and down. The display needs a starting location, so you will pick a pixel location at the center of the screen and write it there. When you push the "right button", the computer adds 1 to the current pixel location number and displays another pixel to the right of the center one. When you press the button again, the computer adds another 1 to the current location number and three pixels are displayed. So pressing the "right button" keeps adding 1 to the current location number and writes pixels to the right. The left button subtracts 1 from the current location number and writes pixels to the left. Pressing the down button adds 32 to the current location number and a pixel gets written down 1 vertical location. The up button subtracts 32 from the current location number and a pixel gets written up 1 vertical location.

When you keep pressing one of the four buttons, the computer will keep adding or subtracting 1 or adding or subtracting 32 from the current location number, displaying a moving line in the direction of which key you pressed. When a key stays depressed, the line will move automatically if a key routine is added to the program. The test routines in Chapter 9 have that added feature.

Binary numbers need to be mentioned again here. Numbers sent from place to place need to be fast and reliable. Electricity meets the criteria if voltage is constant. A common computer voltage that represents a "1" or "0", is 5 volts and 0 volts. Many years ago, military computers were analog. Different voltages would stand for different numbers. These computers were somewhat unreliable. A wire is used to transmit or receive a digit of a binary number. Transmitting binary takes fewer wires than decimal. For example, to transmit 0 through 7 in decimal would require 8 wires, one for each number. To transmit 0 through 7 in binary (000, 001, 010, 011, 100, 101, 110, 111) requires only three wires. Given this efficiency of transmission, you can see why computers would never use decimal. If you want to work on computer controlled equipment, you must learn binary.

How do you make a binary number? The number is made by using a direct current power source (battery) and a switch.

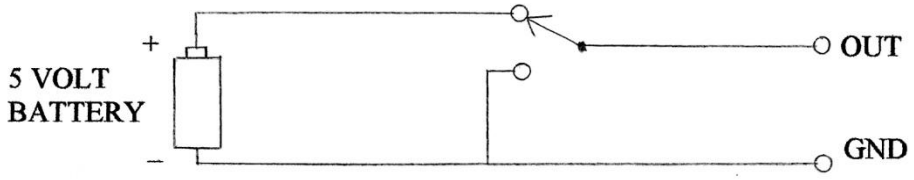



**Figure 4.** How to make a binary number

The switch is the arrow that can move either up or down. OUT is the binary number out. The return part of the circuit is GND, which is the negative side of the battery.

When the switch arrow is up, the circuit outputs 5 volts, which equals a binary "1". When the switch arrow is down, the circuit outputs 0 volts which equals a binary "0". If you have more than one binary digit, you would need two or more switches and more outputs. This would equal a bus. Our computer uses an 8-wire data bus and a 16-wire address bus. Imagine putting eight toggle switches on a piece of plywood with three electrical lugs on each switch: one lug power, one lug ground (GND), and one lug output (as shown in Figure 4). This would equal an 8-bit number. With a toggle switch up, that switch or bit would be a "1" and with the switch down, it would be a "0". Every possible on or off switch combination with eight switches would equal 256 different binary numbers.

The computer needs to store the electrical number signals. To accomplish this, each binary digit - a "1" or a "0" - must be stored. A two transistor circuit is needed. The circuitry is wired so if one transistor is on, the other is off. The transistor that is on is a "0" and the transistor that is off is a "1" and will be stored as long as there is power. This two transistor circuit (Figure 44) is explained in the basic electrical and digital section of the book.

Transistors in computers are used like relays. Relays can have digital logic just like transistor logic. This means it can make a decision. A relay uses an electro-magnet, which is a round steel bar with a wire wrapped around it. When the two ends of the wrapped wire connect to a power source, the steel bar becomes a magnet. A flat piece of spring-loaded steel metal sits about a 1/8 inch away from the electro-magnet. When power is applied to the electro-magnet, the spring metal pulls down and touches the steel bar. Every relay emits a click sound when that happens. That movement is attached to a blade switch. A blade switch is one or two flat bendable pieces of metal with contacts on the end that connect or open up when the electro-magnet turns on.

An "And Gate" tests if two inputs are a "1". This circuit tests if two switches are on.

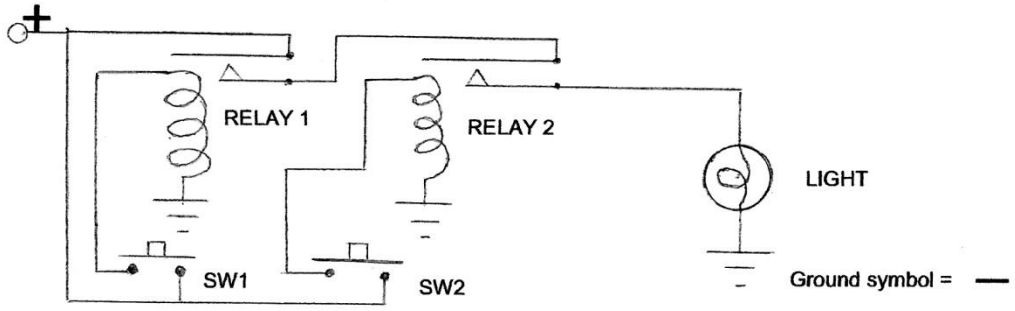



**Figure 5.** Relay "And" gate

On the diagram above the + sign is the positive side of a battery and the - sign is the negative side of a battery. When switch 1(SW1) is pressed, Relay 1 contact closes. When switch 2 (SW2) is also pressed, Relay 2 contact closes. With both relay contacts closed, the light turns on. An important fact about a relay is that a small amount of power is needed to energize the relay coil and a large amount of power can be turned on or off with the relay contact.

Transistors act like relays in computers. Transistors are used because they can be very small, work very fast, and last almost indefinitely. A relay computer would be the size of a house. A transistor has a very small gap on a piece of silicon wafer between the input and output. The gap is chemically doped with electrons and is connected to the base lead (B). When the base has power, electrons flow in the gap. When that happens, the electrons also start flowing from the emitter (E) to the collector (C). A small amount of power from the base (B) to the emitter turns on a large amount of power from the emitter to the collector. Too much power on the base will burn it out; therefore, a resistor (zig-zag line) is added to limit power on the base.

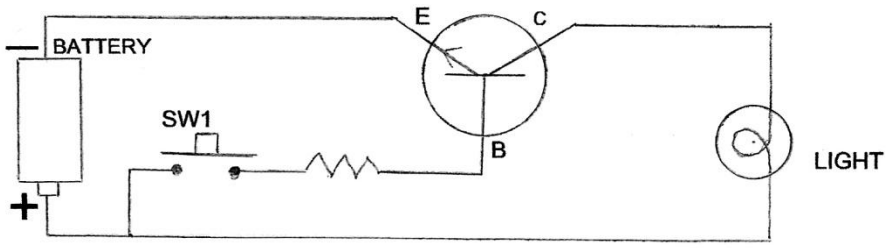



**Figure 6.** Transistor diagram

Looking at Figure 6 diagram, you'll see that when switch 1 is pressed, the base to emitter has power. The emitter and collector then connect and the light turns on. This circuit works the same as a relay circuit. Transistor logic circuits are explained in Chapter 6, Basic Electrical and Digital.

Everything the computer does is controlled by binary numbers, so storing them is an important priority. To store a "1" or "0" takes two transistors, but that's only one bit. You need to be able to store an 8- bit number (0-255). Memories have many addresses to store many numbers. As a programmer, you have to keep track of the numbers and put them in a place that is easy to find. It's also imperative to use the right instructions so the computer will do what you want it to do.

All binary numbers look the same, but their purposes are different and have different names. You can have several memories with different purpose numbers in each memory or put two or three different purpose numbers into one memory. With hardware, purpose numbers would be divided up by using different address lines on the memory for each purpose. With software, memory locations in a memory could be divided up for each purpose.

Some of the purpose numbers are:

1. Program memory numbers - computer instructions
2. RAM memory numbers - numbers that can change, data, or variables
3. Computer key numbers
4. Program counter numbers
5. Address bus numbers
6. Motherboard circuit numbers
7. Micro-coded CPU timing numbers
8. CPU register numbers
9. Data bus numbers

Explaining what all those numbers do is going take this whole book. Some of the software concepts and some the hardware concepts have been covered. More will come later in the book.

I am going to stop explaining computer concepts for now and give you a preview of computer code. Computer code will be examined in more detail later in the book.

The Etch-A-Sketch® game, discussed earlier in this chapter shows how the computer draws pixels to the left, right, up and down. In this part of the chapter we will demonstrate two different programming code methods for drawing pixels to the right.

**Method 1**

Look at Method 1 code diagram, Figure 7. The program address numbers are listed on the left, the assembly instructions are listed in the middle and the machine codes are listed on the right. Program addresses are where the instructions are located and the instructions that are executed are the Machine codes. Assembly codes are used as an easier way to write the machine codes (an assemble program changes assembly codes to machine codes). One by one the 6502 simulator executes instructions down the program list, from 0600 to 0619. The first instruction at program address 0600 is a load "A" register instruction. That instruction takes number 01 that is attached to its instruction code (LDA #$01) and loads it into the "A" register. The second instruction at 0602 is - Store "A" to a memory location. The store "A" instruction takes 01 out of the "A" register along with 020F that is attached to the store "A" instruction code (STA $020F) and sends them to the display circuit. The display circuit uses 020F for a pixel location number, and 01 is used to turn on that pixel. Look at the pixel chart diagram, Figure 1. The top line of the pixel chart goes from location 0200 on the left to location 021F on the right. The center top pixel is 020F, which has been turned on. The third instruction (BRK) at 0605 stops the computer code. Every time you hit "run" you do the next instruction which is another store "A" code with an attached address location number that is plus one more than the last number. Five more store "A" codes, gives you five more pixels to the right. This whole scenario is not practical, because you need another store instruction for every new pixel. The better way to create the code is to have the computer increment the video location for you. To accomplish this, store the video location number in RAM and have the computer increment that stored number. Method 2 does that.

**Method 2**

Method 2 begins with the same starting location, 020F, but it will be put in RAM. Our computer RAM can only store a 2 digit hexadecimal number, so two different load and store instructions are needed to store 020F into the RAM. At program address 0602, "0F" is stored into RAM address 40. At program address 0606, "02" is stored into RAM address 41. So the numbers stored in RAM at address 41 and 40 put together is 020F, which is the first pixel location to be displayed. The load "A" instruction at address 060A, will load 01 into the "A" register. Then the store "A" instruction at address 060C { STA ($40),y} takes 01 from the "A" register along with 020F that is stored in RAM address 41 and 40, and send those numbers to the display circuit to turn on the pixel at location 020F. The next instruction at 060E increments the location stored in RAM by one (020F to 0210). At 0610, the computer code says stop. Now hit "run" again. You will now be at program address 0611, and the jump instruction there says go to program address 060A and execute the code again to write another pixel to the right. Every time you hit run a new pixel will appear next to the last one. Remember that the pixel location address is being incremented in the RAM, every time you hit "run".

Now let's see this code work. Using a PC or Mac computer, go to internet search and type "Easy 6502 by Skilldrick" and enter. Choose any of the small black screens that have a big square box next to it and erase the code. Then type in Method 1 assembly code (Figure 7). After typing each instruction, hit "enter". After all the codes are typed, hit "assemble". Then keep hitting "run" to execute the code. If you typed something wrong, "assemble" and "run" will not work. If you want to see the code work more than once, hit "reset" and run again.

PLEASE NOTE: A SPACE IS REQUIRED AFTER THE OP CODE ON ASSEMBLY INSTRUCTIONS. IN ASSEMBLY, THE OP CODE IS THE FIRST THREE LETTERS OF AN INSTRUCTION. THE SPACE IS A LITTLE HARD TO SEE ON THE CODE DIAGRAMS.

After erasing the previous code, type in Method 2 assembly code (Figure 8). Then execute that code. Try changing the starting location at 0600 from LDA #$0F to LDA #$00. It will now start at the left top of the screen. Look at the pixel chart (Figure 1); you could start anywhere from 0200 to 05FF.

The instruction numbers that you just typed into the web simulator are Assembly programming code. When you click "Assemble", the assembly codes are changed into machine code. Program address numbers and machine code numbers are not displayed on the simulator unless you click "Disassemble". Program address numbers, are the location numbers for each instruction and machine code numbers are the final code that the computer will use. On the next two code diagrams, I added program address numbers and machine code numbers along side of the assembly code.

# METHOD 1

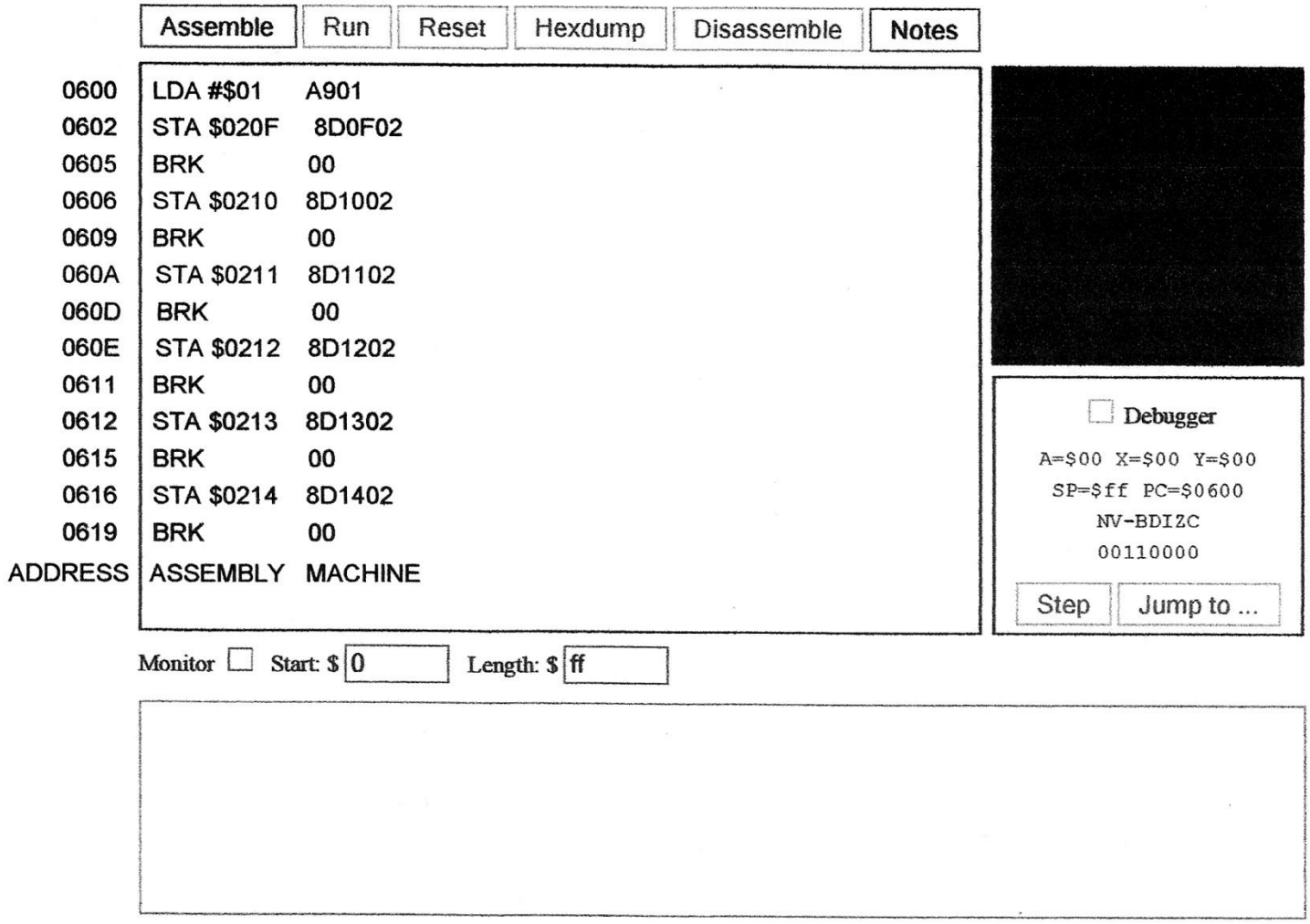



**Figure 7.** Method 1 code

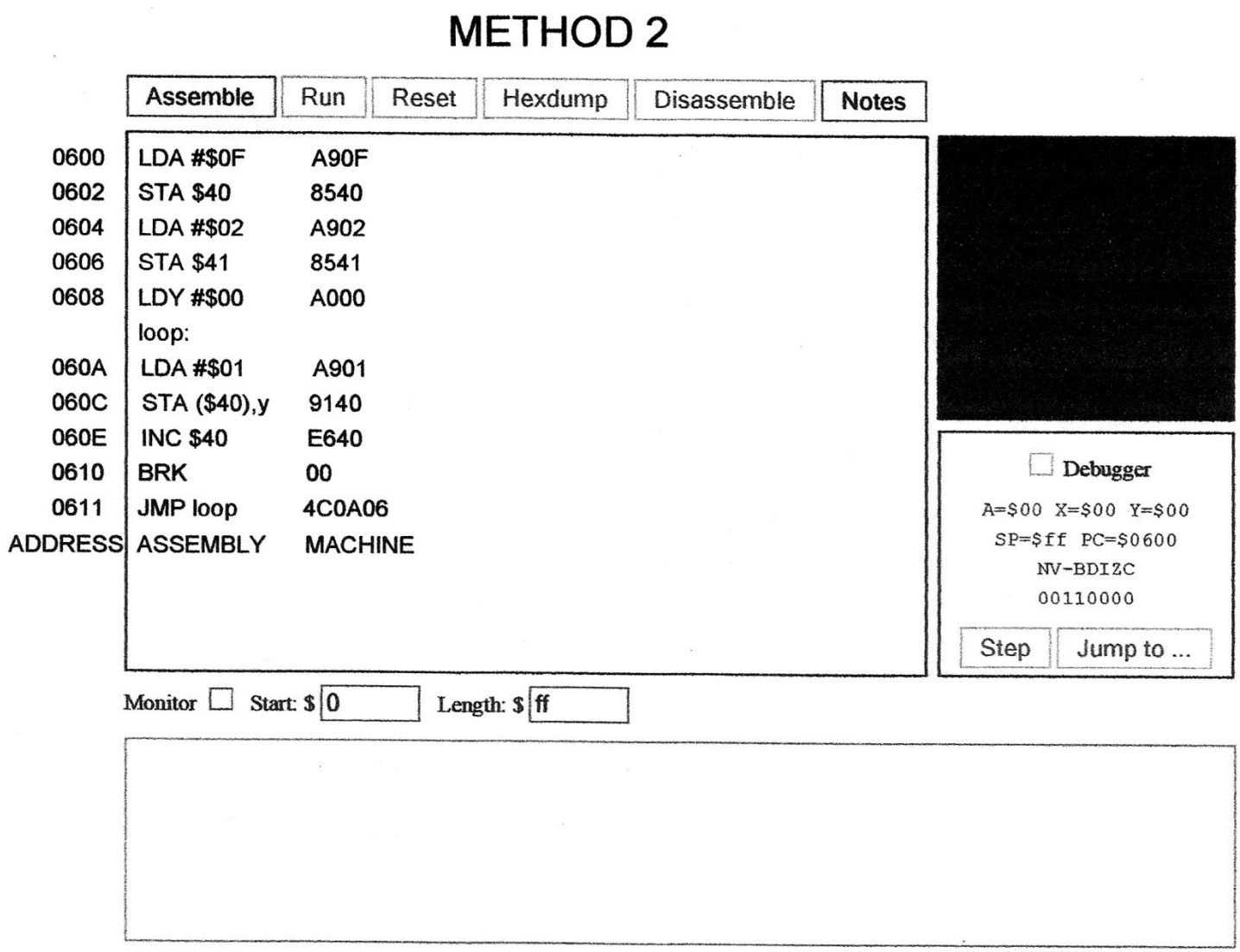



**Figure 8.** Method 2 code

# Chapter 3

## Programming

A programmer can choose one of many high-level languages to program, but they are all converted to machine language, because computers only use the machine language. A machine code instruction can only perform one simple operation. But if you put many machine code instructions together (called a routine) you can do something more complicated. High-level languages use machine code routines for instruction codes. A high-level language will then be able to perform a more complicated instruction which will reduce the number of instructions a programmer needs to make a computer program. Some of the names of high-level languages are: basic, C, C++, DOS, java, visual basic and many others. Because it is hard to figure out what the machine codes are doing in a high-level language, this book will stay with assembly programming, so we can see what the computer is really doing. Assembly code is a version of machine language that is easier to use, but it still needs to be assembled to machine codes.

With 6502 machine language, a two-digit hex number (00 through FA), called an OP code, will choose all instructions and instruction variations. Variations have to do with how an instruction gets its data or address. Most of these variations are called immediate, absolute, direct, and indirect. After the OP code the next 2- or 4-digit hex number is usually data or address for that instruction. The program counter keeps track of the location of each instruction and increments to the next instruction. An OP code with data or addresses is considered one instruction. Because 8-bit computers are only capable of storing a 2 digit hex number, one instruction could use up to 3 memory locations.

Machine code has a few drawbacks compared to assembly code.

1. Machine code numbers are hard to remember, while assembly numbers are not because they have letters you can recall easily.

Example:

Store A register = STA or 85 in machine code
Load A register = LDA or A9 in machine code

2. If you wanted to add an instruction in the middle of a routine in machine code, the program address numbers after that instruction would all be off by two, four, or six program counts. You could deal with this when programming in machine code by adding NOP codes in the middle of software routines. A NOP is an instruction that does nothing except add a program count. If you needed to add an instruction, just take away that NOP and add your new instruction at that program address. In assembly, when you add an instruction in the middle of a routine, it moves everything down automatically and renumbers the program addresses correctly.

3. With some instructions in machine language, if you wanted to jump somewhere else in the code, you would count how many lines of code you need to jump backward or forward, convert it to hex, and add that number to the instruction. In assembly, you just pick a name and put that same name somewhere else in the code and it will jump there.

4. In assembly, if you misspell something, or code something not legal, it will tell you. Machine code will not.

5. Machine code has no symbols. Assembly has many symbols like # $ ( ) , : and others. If you get those symbols wrong you have a mess.

If you want to do more programming than what is in this book, a 6502 programming book will be helpful and can be obtained from an online retailer such as Amazon.com©. Free information from the web is usually not complete or very understandable. It is a good idea to have a 6502 programming book on hand, because it gives you a thorough description of each instruction and examples of how to use them.

The rest of this chapter lists and explains all of the instructions we will be using in the book. Most have data or address locations that you want to use included in the instruction. The first two digits in all machine instructions is the "op" code, which is what the instruction wants to do. The rest of the digits are for location numbers or data numbers that the instruction is going to use.

In the following instructions, xx or xxxx is the place you put your chosen address or data that the instruction will use. The machine codes below are written right after the assembly code. Remember, we are working with assembly code, but when we assemble them with an assembler program, everything gets converted to machine codes. The simulator does not list the machine code numbers like I do here. To be able to see machine codes on the simulator, click "disassemble". Here are the instructions used in this book:

1. LDA #$xx - A9xx - This instruction is called Load "A" register immediate. The number loaded is any 2-digit hex number you choose to put at xx.

2. LDA $xxxx - Adxxxx - This instruction is called load "A" register absolute. A memory location xxxx has the stored data you want to load.

3. STA $xx or $xxxx - 85xx or 8Dxxxx - This instruction takes the contents of the "A" register and stores it to a memory location, using a 2- or 4-digit hex address number that you choose to put in xx or xxxx.

4. LDY #$xx - A0xx - Load Y immediate. The loaded number is any 2-digit hex number you choose to put in xx.

5. STA ($xx), y - 91xx - This instruction is called - store "A" register indirect post indexed with Y. The final end result is to take a 2 digit-hex number that is in the "A" register and store it to a 4 digit-hex memory location that is stored in two consecutive RAM addresses. The first consecutive RAM address is put in xx and the second consecutive address is xx plus one. We will illustrate this instruction by using real numbers.

"A" register = 01. Y register = 00. The RAM locations that will store the memory location that we will use, are in address 40 (lower 2 hex number) and address 41 (upper 2 hex number). Stored at address 40 = 00, and stored at address 41 = 02. The instruction is {STA ($40), y} and the machine code is 9140.

Now get the data from location 40 and 41and put it together and use it as an address to store what is in the "A" register. The Y register is used for offset, but when you put 00 into it, you disable it.

The final address that "A" register data (01) gets stored at is 0200. This might seem like a crazy way to get an address, but it can be used as a variable (changeable address) throughout all of a program's code. This instruction is very useful.

6. LDX #$xx - A2xx - Load x immediate. The loaded number is any 2-digit hex number you choose to put in xx.

7. DEX - CA – Decrements, the 2-digit hex number in the X register by 1.

8. CMP #$xx - C9xx - Compare immediate with the "A" register and set the zero flag if equal. xx is the number you choose to compare and "A" register has the number you want to compare with.

9. BNE + a name or $xx - D0xx - The "Branch if not equal to zero" instruction jumps plus or minus locations by putting a number in xx or by adding a name in the instruction, and then write that same name somewhere else in code. The restriction is +128 addresses or -128 addresses. If branch is equal to zero, nothing happens and the next instruction is executed.

10. JMP + a name or $xxxx address - 4Cxxxx - With the jump instruction, you can choose a name and put that same name somewhere else in code and it will jump to that name, or jump to a 4-digit hex address that you put into xxxx.

11. INC $xx or $xxxx - E6xx or EExxxx - Increment memory at a RAM location. Choose a 2 or 4 digit hex address and the data stored at that address will be incremented.

12. DEC $xx or $xxxx - C6xx or CExxxx - Decrement memory at a RAM location. Choose a 2- or 4-digit hex address and the data stored at that address will be decremented.

13. SEC - 38 - Set carry - so the add instruction will work right.

14. CLC - 18 - Clear carry - is the same as reset borrow so the subtraction instruction will work right.

15. BRK - 00 - Stops the computer code.

The programming guide on the next 16 pages is called "Assembly in one step". It was written in 1980 and is very popular on the web. It is a little complicated for beginning programmers.

After completing this book you should be able write some code for yourself. A 6502 programming book and this programming guide would be enough to get you started. A novice programmer is constantly trying to figure out how to logically put instructions together to perform a certain task or operation. Many times I have used this program guide to help me with programming examples.

If you understand this book and you are able to navigate its programming examples, a career in this line of work might be for you.

# Programming guide

Assembly In One Step
RTK, last update: 23-Jul-97
A brief guide to programming the 6502 in assembly language. It will introduce the 6502 architecture, addressing modes, and instruction set. No prior assembly language programming is assumed, however it is assumed that you are somewhat familiar with hexadecimal numbers. Programming examples are given at the end. Much of this material comes from 6502 Software Design by Leo Scanlon, Blacksburg, 1980.

## The 6502 Architecture

The 6502 is an 8-bit microprocessor that follows the memory oriented design philosophy of the Motorola 6800. Several engineers left Motorola and formed MOS Technology which introduced the 6502 in 1975. The 6502 gained in popularity because of it's low price and became the heart of several early personal computers including the Apple II, Commodore 64, and Atari 400 and 800.

### Simplicity is key

The 6502 handles data in its registers, each of which holds one byte (8-bits) of data. There are a total of three general use and two special purpose registers:

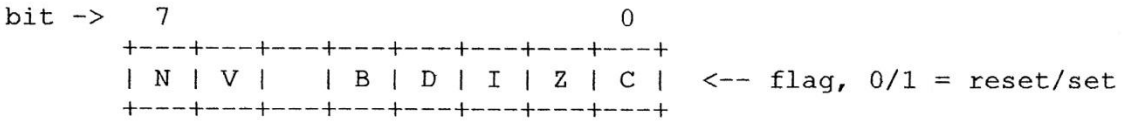
| | | |
|---|---|---|
| accumulator (A) | - | Handles all arithmetic and logic. The real heart of the system. |
| X and Y | - | General purpose registers with limited abilities. |
| S | - | Stack pointer. |
| P | - | Processor status. Holds the result of tests and flags. |

### Stack Pointer

When the microprocessor executes a JSR (Jump to SubRoutine) instruction it needs to know where to return when finished. The 6502 keeps this information in low memory from $0100 to $01FF and uses the stack pointer as an offset. The stack grows down from $01FF and makes it possible to nest subroutines up to 128 levels deep. Not a problem in most cases.

### Processor Status

The processor status register is not directly accessible by any 6502 instruction. Instead, there exist numerous instructions that test the bits of the processor status register. The flags within the register are:




N = NEGATIVE. Set if bit 7 of the accumulator is set.

V = OVERFLOW. Set if the addition of two like-signed numbers or the subtraction of two unlike-signed numbers produces a result greater than +127 or less than -128.

B = BRK COMMAND. Set if an interrupt caused by a BRK, reset if caused by an external interrupt.

D = DECIMAL MODE. Set if decimal mode active.

I = IRQ DISABLE. Set if maskable interrupts are disabled.

Z = ZERO. Set if the result of the last operation (load/inc/dec/ add/sub) was zero.

C = CARRY. Set if the add produced a carry, or if the subtraction produced a borrow. Also holds bits after a logical shift.

### Accumulator

The majority of the 6502's business makes use of the accumulator. All addition and subtraction is done in the accumulator. It also handles the majority of the logical comparisons (is A > B ?) and logical bit shifts.

### X and Y

These are index registers often used to hold offsets to memory locations. They can also be used for holding needed values. Much of their use lies in supporting some of the addressing modes.

## Addressing Modes

The 6502 has 13 addressing modes, or ways of accessing memory. The 65C02 introduces two additional modes.

They are:

| mode | assembler format |
|---|---|
| Immediate | #aa |
| Absolute | aaaa |
| Zero Page | aa |
| Implied | |
| Indirect Absolute | (aaaa) |
| Absolute Indexed,X | aaaa,X |
| Absolute Indexed,Y | aaaa,Y |
| Zero Page Indexed,X | aa,X |
| Zero Page Indexed,Y | aa,Y |
| Indexed Indirect | (aa,X) |
| Indirect Indexed | (aa),Y |
| Relative | aaaa |
| Accumulator | A |

Note:

aa = 2 hex digits as $FF

aaaa = 4 hex digits as $FFFF

Can also be assembler labels

(Table 2-3. _6502 Software Design_, Scanlon, 1980)

## Immediate Addressing

The value given is a number to be used immediately by the instruction. For example, LDA #$99 loads the value $99 into the accumulator.

## Absolute Addressing

The value given is the address (16-bits) of a memory location that contains the 8-bit value to be used. For example, STA $3E32 stores the present value of the accumulator in memory location $3E32.

## Zero Page Addressing

The first 256 memory locations ($0000-00FF) are called "zero page". The next 256 instructions ($0100-01FF) are page 1, etc. Instructions making use of the zero page save memory by not using an extra $00 to indicate the high part of the address. For example,

```
LDA $0023   -- works but uses an extra byte
LDA $23     -- the zero page address
```

## Implied Addressing

Many instructions are only one byte in length and do not reference memory. These are said to be using implied addressing. For example,

```
CLC  -- Clear the carry flag
DEX  -- Decrement the X register by one
TYA  -- Transfer the Y register to the accumulator
```

## Indirect Absolute Addressing

Only used by JMP (JuMP). It takes the given address and uses it as a pointer to the low part of a 16-bit address in memory, then jumps to that address. For example,

```
JMP ($2345)   -- jump to the address in $2345 low and $2346 high
```

So if $2345 contains $EA and $2346 contains $12 then the next instruction executed is the one stored at $12EA. Remember, the 6502 puts its addresses in low/high format.

## Absolute Indexed Addressing

The final address is found by taking the given address as a base and adding the current value of the X or Y register to it as an offset. So,

```
LDA $F453,X  where X contains 3
```

Load the accumulator with the contents of address $F453 + 3 = $F456.

## Zero Page Indexed Addressing

Same as Absolute Indexed but the given address is in the zero page thereby saving a byte of memory.

## Indexed Indirect Addressing

Find the 16-bit address starting at the given location plus the current X register. The value is the contents of that address. For example,

```
LDA ($B4,X)  where X contains 6
```

gives an address of $B4 + 6 = $BA. If $BA and $BB contain $12 and $EE respectively, then the final address is $EE12. The value at location $EE12 is put in the accumulator.

## Indirect Indexed Addressing

Find the 16-bit address contained in the given location ( and the one following). Add to that address the contents of the Y register.

Fetch the value stored at that address. For example,

```
LDA ($B4),Y  where Y contains 6
```

If $B4 contains $EE and $B5 contains $12 then the value at memory location $12EE + Y (6) = $12F4 is fetched and put in the accumulator.

## Relative Addressing

The 6502 branch instructions use relative addressing. The next byte is a signed offset from the current address, and the net sum is the address of the next instruction executed. For example,

```
BNE $7F   (branch on zero flag reset)
```

will add 127 to the current program counter (address to execute) and start executing the instruction at that address. SImilarly,

```
BEQ $F9   (branch on zero flag set)
```

will add a -7 to the current program counter and start execution at the new program counter address.

Remember, if one treats the highest bit (bit 7) of a byte as a sign (0 = positive, 1 = negative) then it is possible to have numbers in the range -128 ($80) to +127 (7F). So, if the high bit is set, i.e. the number is > $7F, it is a negative branch. How far is the branch? If the value is < $80 (positive) it is simply that many bytes. If the value is > $7F (negative) then it is the 2's compliment of the given value in the negative direction.

### 2's compilment

The 2's compilment of a number is found by switching all the bits from 0 -> 1 and 1 -> 0, then adding 1. So,

```
$FF  =  1111 1111   <-- original
        0000 0000   <-- 1's compliment
     +          1
        ---------
        0000 0001   <-- 2's compliment, therefore $FF = -1
```

Note that QForth uses this for numbers greater than 32768 so that 65535 = -1 and 32768 = -32768.

In practice, the assembly language programmer uses a label and the assembler takes care of the actual computation. Note that branches can only be to addresses within -128 to +127 bytes from the present address. The 6502 does not allow branches to an absolute address.

## Accumulator Addressing

Like implied addressing, the object of the instruction is the accumulator and need not be specified.

## The 6502 Instruction Set

There are 56 instructions in the 6502, and more in the 65C02. Many instructions make use of more than one addressing mode and each instruction/addressing mode combination has a particular hexadecimal opcode that specifies it exactly. So,

```
A9  =  LDA #$aa   Immediate addressing mode load of accumulator
AD  =  LDA $aaaa  Absolute addressing mode load of accumulator
etc.
```

Some 6502 instructions make use of bitwise logic. This includes AND, OR, and EOR (Exclusive-OR). The tables below illustrate the effects of these operations:

```
AND   1  1  ->  1    "both"
      1  0  ->  0
      0  1  ->  0
      0  0  ->  0

OR    1  1  ->  1    "either one or both"
      1  0  ->  1
      0  1  ->  1
      0  0  ->  0

EOR   1  1  ->  0    "one or the other but not both"
      1  0  ->  1
      0  1  ->  1
      0  0  ->  0
```

Therefore, $FF AND $0F = $0F since,

```
      1111 1111
  and 0000 1111
      ---------
      0000 1111  = $0F
```

AND is useful for masking bits. For example, to mask the high order bits of a value AND with $0F:

```
$36 AND $0F  =  $06
```

OR is useful for setting a particular bit:

```
$80 OR $08   =  $88

since  1000 0000  ($80)
       0000 1000  ($08)
    or ---------
```

```
          1000 1000  ($88)
```

EOR is useful for flipping bits:

```
   $AA EOR $FF  =  $55

   since  1010 1010  ($AA)
          1111 1111  ($FF)
      eor ---------
          0101 0101  ($55)
```

Other 6502 instructions shift bits to the right or the left or rotate them right or left. Note that shifting to the left by one bit is the same as multipling by 2 and that shifting right by one bit is the same as dividing by 2.

The 6502 instructions fall naturally into 10 groups with two odd-ball instructions NOP and BRK:

- Load and Store Instructions
- Arithmetic Instructions
- Increment and Decrement Instructions
- Logical Instructions
- Jump, Branch, Compare and Test Bits Instructions
- Shift and Rotate Instructions
- Transfer Instructions
- Stack Instructions
- Subroutine Instructions
- Set/Reset Instructions
- NOP/BRK Instructions

## Load and Store Instructions

LDA - LoaD the Accumulator
LDX - LoaD the X register
LDY - LoaD the Y register

STA - STore the Accumulator
STX - STore the X register
STY - STore the Y register

Microprocessors spend much of their time moving stuff around in memory. Data from one location is loaded into a register and stored in another location, often with something added or subtracted in the process. Memory can be loaded directly into the A, X, and Y registers but as usual, the accumulator has more addressing modes available.

If the high bit (left most, bit 7) is set when loaded the N flag on the processor status register is set. If the loaded value is zero the Z flag is set.

# Arithmetic Instructions

ADC - ADd to accumulator with Carry
SBC - SuBtract from accumulator with Carry

The 6502 has two arithmetic modes, binary and decimal. Both addition and subtraction implement the carry flag to track carries and borrows thereby making multibyte arithmetic simple. Note that in the case of subtraction it is necessary to SET the carry flag as it is the opposite of the carry that is subtracted.

Addition should follow this form:

```
CLC
ADC ...
.
.
ADC ...
.
.
.
```

Clear the carry flag, and perform all the additions. The carry between additions will be handled in the carry flag. Add from low byte to high byte. Symbolically, the net effect of an ADC instruction is:

A + M + C --> A

Subtraction follows the same format:

```
SEC
SBC ...
.
.
SBC ...
.
.
.
```

In this case set the carry flag first and then do the subtractions. Symbolically,

A - M - ~C --> A , where ~C is the opposite of C

Ex.1
---

A 16-bit addition routine. $20,$21 + $22,$23 = $24,$25

```
CLC         clear the carry
LDA $20     get the low byte of the first number
ADC $22     add to it the low byte of the second
STA $24     store in the low byte of the result
LDA $21     get the high byte of the first number
ADC $23     add to it the high byte of the second, plus carry
```

```
        STA $25     store in high byte of the result

        ... on exit the carry will be set if the result could not be
            contained in 16-bit number.
```

Ex.2
---

A 16-bit subtraction routine. $20,$21 - $22,$23 = $24,$25

```
        SEC         clear the carry
        LDA $20     get the low byte of the first number
        SBC $22     add to it the low byte of the second
        STA $24     store in the low byte of the result
        LDA $21     get the high byte of the first number
        SBC $23     add to it the high byte of the second, plus carry
        STA $25     store in high byte of the result

        ... on exit the carry will be set if the result produced a
            borrow
```

Aside from the carry flag, arithmetic instructions also affect the N, Z, and V flags as follows:

```
   Z = 1  if result was zero, 0 otherwise
   N = 1  if bit 7 of the result is 1, 0 otherwise
   V = 1  if bit 7 of the accumulator was changed, a sign change
```

## Increment and Decrement Instructions

```
INC  - INCrement memory by one
INX  - INcrement X by one
INY  - INcrement Y by one

DEC  - DECrement memory by one
DEX  - DEcrement X by one
DEY  - DEcrement Y by one
```

The 6502 has instructions for incrementing/decrementing the index registers and memory. Note that it does not have instructions for incrementing/decrementing the accumulator. This oversight was rectified in the 65C02 which added INA and DEA instructions. The index register instructions are implied mode for obvious reasons while the INC and DEC instructions use a number of addressing modes.

All inc/dec instructions have alter the processor status flags in the following way:

```
  Z = 1  if the result is zero, 0 otherwise
  N = 1  if bit 7 is 1, 0 otherwise
```

## Logical Instructions

AND - AND memory with accumulator
ORA - OR memory with Accumulator
EOR - Exclusive-OR memory with Accumulator

These instructions perform a bitwise binary operation according to the tables given above. They set the Z flag if the net result is zero and set the N flag if bit 7 of the result is set.

## Jump, Branch, Compare, and Test Bits

JMP - JuMP to another location (GOTO)

BCC - Branch on Carry Clear, C = 0
BCS - Branch on Carry Set, C = 1
BEQ - Branch on EQual to zero, Z = 1
BNE - Branch on Not Equal to zero, Z = 0
BMI - Branch on MInus, N = 1
BPL - Branch on PLus, N = 0
BVS - Branch on oVerflow Set, V = 1
BVC - Branch on oVerflow Clear, V = 0

CMP - CoMPare memory and accumulator
CPX - ComPare memory and X
CPY - ComPare memory and Y

BIT - test BITs

This large group includes all instructions that alter the flow of the program or perform a comparison of values or bits.

JMP simply sets the program counter (PC) to the address given. Execution proceeds from the new address. The branch instructions are relative jumps. They cause a branch to a new address that is either 127 bytes beyond the current PC or 128 bytes before the current PC. Code that only uses branch instructions is relocatable and can be run anywhere in memory.

The three compare instructions are used to set processor status bits. After the comparison one frequently branches to a new place in the program based on the settings of the status register. The relationship between the compared values and the status bits is,

| | N | Z | C |
|---|---|---|---|
| A, X, or Y < Memory | 1 | 0 | 0 |
| A, X, or Y = Memory | 0 | 1 | 1 |
| A, X, or Y > Memory | 0 | 0 | 1 |

The BIT instruction tests bits in memory with the accumulator but

changes neither. Only processor status flags are set. The contents of the specified memory location are logically ANDed with the accumulator, then the status bits are set such that,

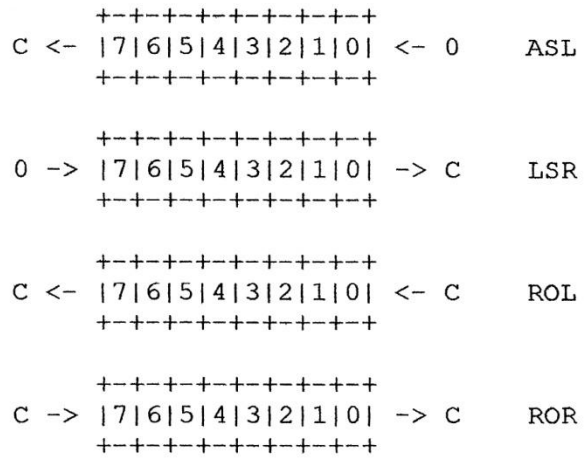
* N receives the initial, un-ANDed value of memory bit 7.
* V receives the initial, un-ANDed value of memory bit 6.
* Z is set if the result of the AND is zero, otherwise reset.

So, if $23 contained $7F and the accumulator contained $80 a BIT $23 instruction would result in the V and Z flags being set and N reset since bit 7 of $7F is 0, bit 6 of $7F is 1, and $7F AND $80 = 0.

## Shift and Rotate Instructions

ASL - Accumulator Shift Left
LSR - Logical Shift Right
ROL - ROtate Left
ROR - ROtate Right

Use these instructions to move things around in the accumulator or memory. The net effects are (where C is the carry flag):




Z is set if the result it zero. N is set if bit 7 is 1. It is always reset on LSR. Remember that ASL A is equal to multiplying by two and that LSR is equal to dividing by two.

## Transfer Instructions

TAX - Transfer Accumulator to X
TAY - Transfer Accumulator to Y
TXA - Transfer X to accumulator
TYA - Transfer Y to Accumulator

Transfer instructions move values between the 6502 registers. The N and Z flags are set if the value being moved warrants it, i.e.

```
LDA #$80
TAX
```

causes the N flag to be set since bit 7 of the value moved is 1, while

```
LDX #$00
TXA
```

causes the Z flag to be set since the value is zero.

## Stack Instructions

```
TSX  - Transfer Stack pointer to X
TXS  - Transfer X to Stack pointer

PHA  - PusH Accumulator on stack
PHP  - PusH Processor status on stack
PLA  - PulL Accumulator from stack
PLP  - PulL Processor status from stack
```

TSX and TXS make manipulating the stack possible. The push and pull instructions are useful for saving register values and status flags. Their operation is straightforward.

## Subroutine Instructions

```
JSR  - Jump to SubRoutine
RTS  - ReTurn from Subroutine
RTI  - ReTurn from Interrupt
```

Like JMP, JSR causes the program to start execution of the next instruction at the given address. Unlike JMP, JSR pushes the address of the next instruction after itself on the stack. When an RTS instruction is executed the address pushed on the stack is pulled off the stack and the program resumes at that address. For example,

```
   LDA #$C1   ; load the character 'A'
   JSR print  ; print the character and it's hex code
   LDA #$C2   ; load 'B'
   JSR print  ; and print it
   .
   .
   .
print JSR $FDED  ; print the letter
      JSR $FDDA  ; and its ASCII code
      RTS        ; return to the caller
```

RTI is analagous to RTS and should be used to end an interrupt routine.

## Set and Reset (Clear) Instructions

```
CLC  - CLear Carry flag
CLD  - CLear Decimal mode
CLI  - CLear Interrupt disable
CLV  - CLear oVerflow flag

SEC  - SEt Carry
SED  - SEt Decimal mode
SEI  - SEt Interrupt disable
```

These are one byte instructions to specify processor status flag settings.

CLC and SEC are of particular use in addition and subtraction respectively. Before any addition (ADC) use CLC to clear the carry or the result may be one greater than you expect. For subtraction (SBC) use SEC to ensure that the carry is set as its compliment is subtracted from the answer. In multi-byte additions or subtractions only clear or set the carry flag before the initial operation. For example, to add one to a 16-bit number in $23 and $24 you would write:

```
LDA $23      ; get the low byte
CLC          ; clear the carry
ADC #$02     ; add a constant 2, carry will be set if result > 255
STA $23      ; save the low byte
LDA $24      ; get the high byte
ADC #$00     ; add zero to add any carry that might have been set above
STA $24      ; save the high byte
RTS          ; if carry set now the result was > 65535
```

Similarly for subtraction,

```
LDA $23      ; get the low byte
SEC          ; set the carry
SBC #$02     ; subtract 2
STA $23      ; save the low byte
LDA $24      ; get the high byte
SBC #$00     ; subtract 0 and any borrow generated above
STA $24      ; save the high byte
RTS          ; if the carry is not set the result was < 0
```

## Other Instructions

```
NOP  - No OPeration (or is it NO oPeration ? :)
BRK  - BReaK
```

NOP is just that, no operation. Useful for deleting old instructions, reserving room for future instructions or for use in careful timing loops as it uses 2 microprocessor cycles.

BRK causes a forced break to occur and the processor will immediately start execution of the routine whose address is in $FFFE and $FFFF. This address is often the start of a system monitor program.

## Some simple programming examples

A few simple programming examples are given here. They serve to illustrate some techniques commonly used in assembly programming. There are doubtless dozens more and I make no claim at being a proficient assembly language programmer. For examples of addition and subtraction see above on CLC and SEC.

### A count down loop

```
;
; An 8-bit count down loop
;

start LDX #$FF     ; load X with $FF = 255
loop  DEX          ; X = X - 1
      BNE loop     ; if X not zero then goto loop
      RTS          ; return
```

How does the BNE instruction know that X is zero? It doesn't, all it knows is that the Z flag is set or reset. The DEX instruction will set the Z flag when X is zero.

```
;
; A 16-bit count down loop
;

start LDY #$FF     ; load Y with $FF
loop1 LDX #$FF     ; load X with $FF
loop2 DEX          ; X = X - 1
      BNE loop2    ; if X not zero goto loop2
      DEY          ; Y = Y - 1
      BNE loop1    ; if Y not zero goto loop1
      RTS          ; return
```

There are two loops here, X will be set to 255 and count to zero for each time Y is decremented. The net result is to count the 16-bit number Y (high) and X (low) down from $FFFF = 65535 to zero.

### Other examples

** Note: All of the following examples are lifted nearly verbatim from the book "6502 Software Design", whose reference is above.

```
; Example 4-2.  Deleting an entry from an unordered list
;
; Delete the contents of $2F from a list whose starting
; address is in $30 and $31.  The first byte of the list
; is its length.
;

deluel  LDY #$00    ; fetch element count
        LDA ($30),Y
        TAX         ; transfer length to X
        LDA $2F     ; item to delete
nextel  INY         ; index to next element
        CMP ($30),Y ; do entry and element match?
        BEQ delete  ; yes. delete element
        DEX         ; no. decrement element count
        BNE nextel  ; any more elements to compare?
        RTS         ; no. element not in list. done

; delete an element by moving the ones below it up one location

delete  DEX         ; decrement element count
        BEQ deccnt  ; end of list?
        INY         ; no. move next element up
        LDA ($30),Y
        DEY
        STA ($30),Y
        INY
        JMP delete
deccnt  LDA ($30,X) ; update element count of list
        SBC #$01
        STA ($30,X)
        RTS
```

```
; Example 5-6.  16-bit by 16-bit unsigned multiply
;
; Multiply $22 (low) and $23 (high) by $20 (low) and
; $21 (high) producing a 32-bit result in $24 (low) to $27 (high)
;

mlt16   LDA #$00    ; clear p2 and p3 of product
        STA $26
        STA $27
        LDX #$16    ; multiplier bit count = 16
nxtbt   LSR $21     ; shift two-byte multiplier right
        ROR $20
        BCC align   ; multiplier = 1?
        LDA $26     ; yes. fetch p2
        CLC
        ADC $22     ; and add m0 to it
        STA $26     ; store new p2
        LDA $27     ; fetch p3
        ADC $23     ; and add m1 to it
```

```
align   ROR A         ; rotate four-byte product right
        STA $27       ; store new p3
        ROR $26
        ROR $25
        ROR $24
        DEX           ; decrement bit count
        BNE nxtbt     ; loop until 16 bits are done
        RTS



; Example 5-14.  Simple 16-bit square root.
;
; Returns the 8-bit square root in $20 of the
; 16-bit number in $20 (low) and $21 (high). The
; remainder is in location $21.

sqrt16  LDY #$01      ; lsby of first odd number = 1
        STY $22
        DEY
        STY $23       ; msby of first odd number (sqrt = 0)
again   SEC
        LDA $20       ; save remainder in X register
        TAX           ; subtract odd lo from integer lo
        SBC $22
        STA $20
        LDA $21       ; subtract odd hi from integer hi
        SBC $23
        STA $21       ; is subtract result negative?
        BCC nomore    ; no. increment square root
        INY
        LDA $22       ; calculate next odd number
        ADC #$01
        STA $22
        BCC again
        INC $23
        JMP again
 nomore STY $20       ; all done, store square root
        STX $21       ; and remainder
        RTS
```

This is based on the observation that the square root of an integer is equal to the number of times an increasing odd number can be subtracted from the original number and remain positive.  For example,

```
  25
- 1          1
  --
  24
- 3          2
  --
  21
- 5          3
  --
  16
```

# Chapter 4

## Binary Numbers

Computers use binary numbers because they're the most dependable way of communicating. The binary digits are on or off, "1" or "0". The common voltage for a "0" is 0 volts. The common voltages for a "1" are 5.0 volts 3.3 volts and 1.8 volts.

The numbers below will count from 0 to 16 in binary, hexadecimal, and decimal. Each of the three types of numbers below, have the same number value.

| Binary | Hexadecimal | Decimal |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | A | 10 |
| 1011 | B | 11 |
| 1100 | C | 12 |
| 1101 | D | 13 |
| 1110 | E | 14 |
| 1111 | F | 15 |
| 10000 | 10 | 16 |

Everything on a computer is represented exclusively by binary numbers. This includes data, instructions, program counter numbers, inputs, outputs, memory locations, addresses and circuit identifier numbers.

All of the following are represented by a binary number.

1. Every key on the keyboard
2. Every location on the monitor screen
3. Every color
4. Every measurement
5. Every letter or number on a monitor screen is made up with a series of pixels that are binary numbers

All inputs must be converted to binary numbers and all outputs must be converted from binary numbers to a letter or a number or some sort of a signal. An example of using binary numbers could be illustrated by shaking a robots hand. When you shake the robot's hand the movement is changed into binary numbers by using an analog to digital circuit. Those binary numbers are sent to a computer. The computer analyses the numbers and sends new binary numbers back to the robot. The new numbers are changed into electric power by using a digital to analog circuit. The electric power then energizes motors in the arm and hand and the robot shakes your hand in return.

Binary numbers are difficult to work with. Separating the numbers into bits and bytes improves the process. Anything you can do with decimal, you can do with binary or hexadecimal. You can add, subtract, multiply, divide, compare, decrement, and increment.

This is an example of an 8-bit binary number:

Byte 1 Byte 0
Binary = 00011111

Bits 8 Bits – 4 Bits in each byte

Hex = 1 F (each hex number or letter is a byte)

Decimal 1, 15 (each byte)

Decimal 31 (both bytes)

Use the binary conversion chart to help you convert numbers.

# Conversion Table

## Decimal - Hexadecimal - Binary

| Dec | Hex | Bin | Dec | Hex | Bin | Dec | Hex | Bin | Dec | Hex | Bin |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 00000000 | 64 | 40 | 01000000 | 128 | 80 | 10000000 | 192 | c0 | 11000000 |
| 1 | 1 | 00000001 | 65 | 41 | 01000001 | 129 | 81 | 10000001 | 193 | c1 | 11000001 |
| 2 | 2 | 00000010 | 66 | 42 | 01000010 | 130 | 82 | 10000010 | 194 | c2 | 11000010 |
| 3 | 3 | 00000011 | 67 | 43 | 01000011 | 131 | 83 | 10000011 | 195 | c3 | 11000011 |
| 4 | 4 | 00000100 | 68 | 44 | 01000100 | 132 | 84 | 10000100 | 196 | c4 | 11000100 |
| 5 | 5 | 00000101 | 69 | 45 | 01000101 | 133 | 85 | 10000101 | 197 | c5 | 11000101 |
| 6 | 6 | 00000110 | 70 | 46 | 01000110 | 134 | 86 | 10000110 | 198 | c6 | 11000110 |
| 7 | 7 | 00000111 | 71 | 47 | 01000111 | 135 | 87 | 10000111 | 199 | c7 | 11000111 |
| 8 | 8 | 00001000 | 72 | 48 | 01001000 | 136 | 88 | 10001000 | 200 | c8 | 11001000 |
| 9 | 9 | 00001001 | 73 | 49 | 01001001 | 137 | 89 | 10001001 | 201 | c9 | 11001001 |
| 10 | a | 00001010 | 74 | 4a | 01001010 | 138 | 8a | 10001010 | 202 | ca | 11001010 |
| 11 | b | 00001011 | 75 | 4b | 01001011 | 139 | 8b | 10001011 | 203 | cb | 11001011 |
| 12 | c | 00001100 | 76 | 4c | 01001100 | 140 | 8c | 10001100 | 204 | cc | 11001100 |
| 13 | d | 00001101 | 77 | 4d | 01001101 | 141 | 8d | 10001101 | 205 | cd | 11001101 |
| 14 | e | 00001110 | 78 | 4e | 01001110 | 142 | 8e | 10001110 | 206 | ce | 11001110 |
| 15 | f | 00001111 | 79 | 4f | 01001111 | 143 | 8f | 10001111 | 207 | cf | 11001111 |
| 16 | 10 | 00010000 | 80 | 50 | 01010000 | 144 | 90 | 10010000 | 208 | d0 | 11010000 |
| 17 | 11 | 00010001 | 81 | 51 | 01010001 | 145 | 91 | 10010001 | 209 | d1 | 11010001 |
| 18 | 12 | 00010010 | 82 | 52 | 01010010 | 146 | 92 | 10010010 | 210 | d2 | 11010010 |
| 19 | 13 | 00010011 | 83 | 53 | 01010011 | 147 | 93 | 10010011 | 211 | d3 | 11010011 |
| 20 | 14 | 00010100 | 84 | 54 | 01010100 | 148 | 94 | 10010100 | 212 | d4 | 11010100 |
| 21 | 15 | 00010101 | 85 | 55 | 01010101 | 149 | 95 | 10010101 | 213 | d5 | 11010101 |
| 22 | 16 | 00010110 | 86 | 56 | 01010110 | 150 | 96 | 10010110 | 214 | d6 | 11010110 |
| 23 | 17 | 00010111 | 87 | 57 | 01010111 | 151 | 97 | 10010111 | 215 | d7 | 11010111 |
| 24 | 18 | 00011000 | 88 | 58 | 01011000 | 152 | 98 | 10011000 | 216 | d8 | 11011000 |
| 25 | 19 | 00011001 | 89 | 59 | 01011001 | 153 | 99 | 10011001 | 217 | d9 | 11011001 |
| 26 | 1a | 00011010 | 90 | 5a | 01011010 | 154 | 9a | 10011010 | 218 | da | 11011010 |
| 27 | 1b | 00011011 | 91 | 5b | 01011011 | 155 | 9b | 10011011 | 219 | db | 11011011 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 28 | 1c | 00011100 | 92 | 5c | 01011100 | 156 | 9c | 10011100 | 220 | dc | 11011100 |
| 29 | 1d | 00011101 | 93 | 5d | 01011101 | 157 | 9d | 10011101 | 221 | dd | 11011101 |
| 30 | 1e | 00011110 | 94 | 5e | 01011110 | 158 | 9e | 10011110 | 222 | de | 11011110 |
| 31 | 1f | 00011111 | 95 | 5f | 01011111 | 159 | 9f | 10011111 | 223 | df | 11011111 |
| 32 | 20 | 00100000 | 96 | 60 | 01100000 | 160 | a0 | 10100000 | 224 | e0 | 11100000 |
| 33 | 21 | 00100001 | 97 | 61 | 01100001 | 161 | a1 | 10100001 | 225 | e1 | 11100001 |
| 34 | 22 | 00100010 | 98 | 62 | 01100010 | 162 | a2 | 10100010 | 226 | e2 | 11100010 |
| 35 | 23 | 00100011 | 99 | 63 | 01100011 | 163 | a3 | 10100011 | 227 | e3 | 11100011 |
| 36 | 24 | 00100100 | 100 | 64 | 01100100 | 164 | a4 | 10100100 | 228 | e4 | 11100100 |
| 37 | 25 | 00100101 | 101 | 65 | 01100101 | 165 | a5 | 10100101 | 229 | e5 | 11100101 |
| 38 | 26 | 00100110 | 102 | 66 | 01100110 | 166 | a6 | 10100110 | 230 | e6 | 11100110 |
| 39 | 27 | 00100111 | 103 | 67 | 01100111 | 167 | a7 | 10100111 | 231 | e7 | 11100111 |
| 40 | 28 | 00101000 | 104 | 68 | 01101000 | 168 | a8 | 10101000 | 232 | e8 | 11101000 |
| 41 | 29 | 00101001 | 105 | 69 | 01101001 | 169 | a9 | 10101001 | 233 | e9 | 11101001 |
| 42 | 2a | 00101010 | 106 | 6a | 01101010 | 170 | aa | 10101010 | 234 | ea | 11101010 |
| 43 | 2b | 00101011 | 107 | 6b | 01101011 | 171 | ab | 10101011 | 235 | eb | 11101011 |
| 44 | 2c | 00101100 | 108 | 6c | 01101100 | 172 | ac | 10101100 | 236 | ec | 11101100 |
| 45 | 2d | 00101101 | 109 | 6d | 01101101 | 173 | ad | 10101101 | 237 | ed | 11101101 |
| 46 | 2e | 00101110 | 110 | 6e | 01101110 | 174 | ae | 10101110 | 238 | ee | 11101110 |
| 47 | 2f | 00101111 | 111 | 6f | 01101111 | 175 | af | 10101111 | 239 | ef | 11101111 |
| 48 | 30 | 00110000 | 112 | 70 | 01110000 | 176 | b0 | 10110000 | 240 | f0 | 11110000 |
| 49 | 31 | 00110001 | 113 | 71 | 01110001 | 177 | b1 | 10110001 | 241 | f1 | 11110001 |
| 50 | 32 | 00110010 | 114 | 72 | 01110010 | 178 | b2 | 10110010 | 242 | f2 | 11110010 |
| 51 | 33 | 00110011 | 115 | 73 | 01110011 | 179 | b3 | 10110011 | 243 | f3 | 11110011 |
| 52 | 34 | 00110100 | 116 | 74 | 01110100 | 180 | b4 | 10110100 | 244 | f4 | 11110100 |
| 53 | 35 | 00110101 | 117 | 75 | 01110101 | 181 | b5 | 10110101 | 245 | f5 | 11110101 |
| 54 | 36 | 00110110 | 118 | 76 | 01110110 | 182 | b6 | 10110110 | 246 | f6 | 11110110 |
| 55 | 37 | 00110111 | 119 | 77 | 01110111 | 183 | b7 | 10110111 | 247 | f7 | 11110111 |
| 56 | 38 | 00111000 | 120 | 78 | 01111000 | 184 | b8 | 10111000 | 248 | f8 | 11111000 |
| 57 | 39 | 00111001 | 121 | 79 | 01111001 | 185 | b9 | 10111001 | 249 | f9 | 11111001 |
| 58 | 3a | 00111010 | 122 | 7a | 01111010 | 186 | ba | 10111010 | 250 | fa | 11111010 |
| 59 | 3b | 00111011 | 123 | 7b | 01111011 | 187 | bb | 10111011 | 251 | fb | 11111011 |
| 60 | 3c | 00111100 | 124 | 7c | 01111100 | 188 | bc | 10111100 | 252 | fc | 11111100 |
| 61 | 3d | 00111101 | 125 | 7d | 01111101 | 189 | bd | 10111101 | 253 | fd | 11111101 |
| 62 | 3e | 00111110 | 126 | 7e | 01111110 | 190 | be | 10111110 | 254 | fe | 11111110 |
| 63 | 3f | 00111111 | 127 | 7f | 01111111 | 191 | bf | 10111111 | 255 | ff | 11111111 |

# Chapter 5

## Display Circuit

The display circuit consists of three circuits put together.

1. Horizontal and vertical binary counters
2. Multiplexer (selector)
3. RAM

The monitor paints a picture by scanning a line from the top left to the top right. Then one line down it scans another. It keeps scanning lines to bottom until the entire screen is filled. Horizontal sync signal starts the horizontal line at the left and stops the line at the right. Vertical sync signal starts the lines at the top of the screen and ends the lines at the bottom. Without these two signals, the vertical and horizontal oscillators in the monitor will free run and the video would float all over the screen. The bright and dark of each horizontal line make up the picture. To demonstrate that concept, take a picture photograph and cut it into small horizontal strips with scissors. Connect the strips end to end in one long chain and send them to a location. When the chain gets to that location the strips are disconnected from the chain and glued back together onto a piece of cardboard. When the strips are reassembled, you'll have the picture back. That process can be compared to serial data being transmitted to a TV set, the internet, or a computer screen.

The Display Circuit timing uses the following counters. See Figure 3.

a. Horizontal timing. Binary counters IC 1 and IC 2.
b. Vertical timing. Binary counters IC 3 and IC 4.

It is important to note that when the horizontal counters (IC 1, IC 2) get to its final count, it starts the vertical counters (IC3, IC4) to move the line down one vertical location to so the horizontal counters can draw another line. More importantly, there are in-between count bits from the counters (1H-128H and 1V-128V) that are in synchronization with the monitor and count in binary. A combination of those signals count out the stored pixels that are in the video RAM so they can be sent to the monitor to be displayed.

Our 1-bit video RAM has a data bit in and a data bit out. The data bit in (Din) is connected to bit D0 of the data bus and receives a "1" or "0" from the CPU. A "1" on Din turns on a pixel and a "0" on Din turns off a pixel. The first address in the video RAM refers to the upper left pixel position on the monitor. The last address refers to the lower right pixel position of the monitor. If you write a "1" in the first video address and a one in the last address, you will get a pixel (a small white square) in the top left and the bottom right of the screen. If you write a "1" in every video address, the entire screen will turn white.

The main idea of the display circuit is to write a pixel into a video RAM location and then read it out and send it to the monitor. The address location you put the pixel in determines where on the screen you will see it. As you can see, it is a two-step process. Look at Figure 3.

**Step 1. Write to the RAM**

When the processor wants to write to the display circuit to turn on a pixel, it outputs an address from 0200 to 05FF and puts a "1" on data bit D0. The decoder and the 4-input NOR gate detects that address range and outputs a "0" to the Video RAM R/$\overline{W}$ input, which puts the video RAM into the write mode (R/$\overline{W}$ from the CPU to the RAM is a separate signal). The 4-input NOR gate output ("0") is also sent to the S select inputs of the multiplexers. That causes the multiplexers to select "A" inputs to go to C outputs. The end result is, CPU addresses connect to video RAM addresses and data bit D0 ("1"), from the CPU is written into the selected video Ram address.

**Step 2. Read the RAM**

The CPU stops writing to the video RAM and the decoder outputs turn off ("0"). The 4- input NOR gate output changes to "1" which is connected to the S select signal on the multiplexers and the R/$\overline{W}$ signal on the video RAM. The multiplexers now select B inputs to go to C outputs. This means the binary counters, which are always counting, connect to the address inputs of the Video RAM. The R/$\overline{W}$ signal on the video RAM which is now "1", puts the video RAM into the Read mode. The binary counters count all the video RAM addresses and the pixel that was written into the video Ram at Din comes out at Dout and gets sent to the monitor to be displayed.

The display circuit has two different modes or times, on-screen or off-screen. You do not want to write pixels during on-screen time or you would see the pixels writing one by one. Movie film projectors have a shutter that covers and stops the image while changing picture frames for that reason. During off-screen time (called blanking) the CPU writes one or more pixels to the display circuit. During on-screen time, the video binary counters count out all the pixels in the video RAM over and over to keep them displayed on the monitor screen.

My sync generator circuit on Figure 3 with four counters is a simplified circuit for explanation purposes and does not have all the circuitry needed for correct timing. The real sync counter circuit, taken from a video game on Figure 9 has the correct timing for:

1. Horizontal sync - After starting a line, at what count are you going to stop?
2. Vertical sync - How many horizontal lines are you going to count vertically?
3. Horizontal blank - After drawing a horizontal line, how many counts do you wait until you draw another?
4. Vertical blank - After the last horizontal line at the bottom, how many counts do you wait until you start back at the top of the screen again?
5. Frames per second - How many?

The counting (timing) of these signals is different in various parts of the world. National Television System Committee (NTSC) is the standard in the Americas and the Phase Alternation By Line (PAL) is more common in Europe. These are legal timing signals. The timing circuit on Figure 9 has all the added parts and circuitry needed to get the exact timing for all of the sync signals.

A popular troubleshooting method is to disconnect the video input to the monitor, and use it as a probe to look at video signals. There are eight probe video picture signals on the following pages for some of the vertical and horizontal counter outputs. All video signals can be looked at using this probe method. The tool used to look at all the digital signals is an oscilloscope.

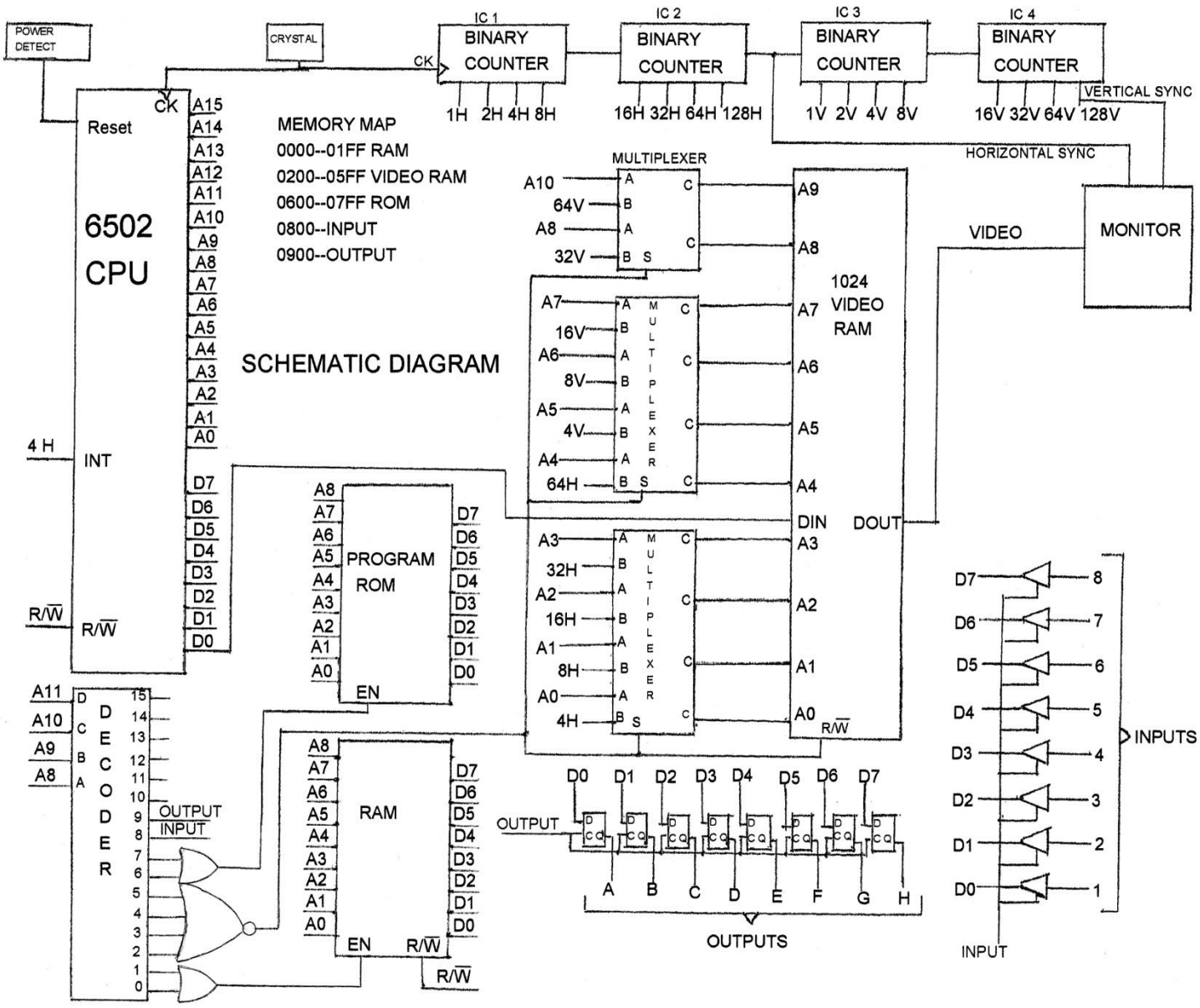



**Figure 3.** Schematic Diagram

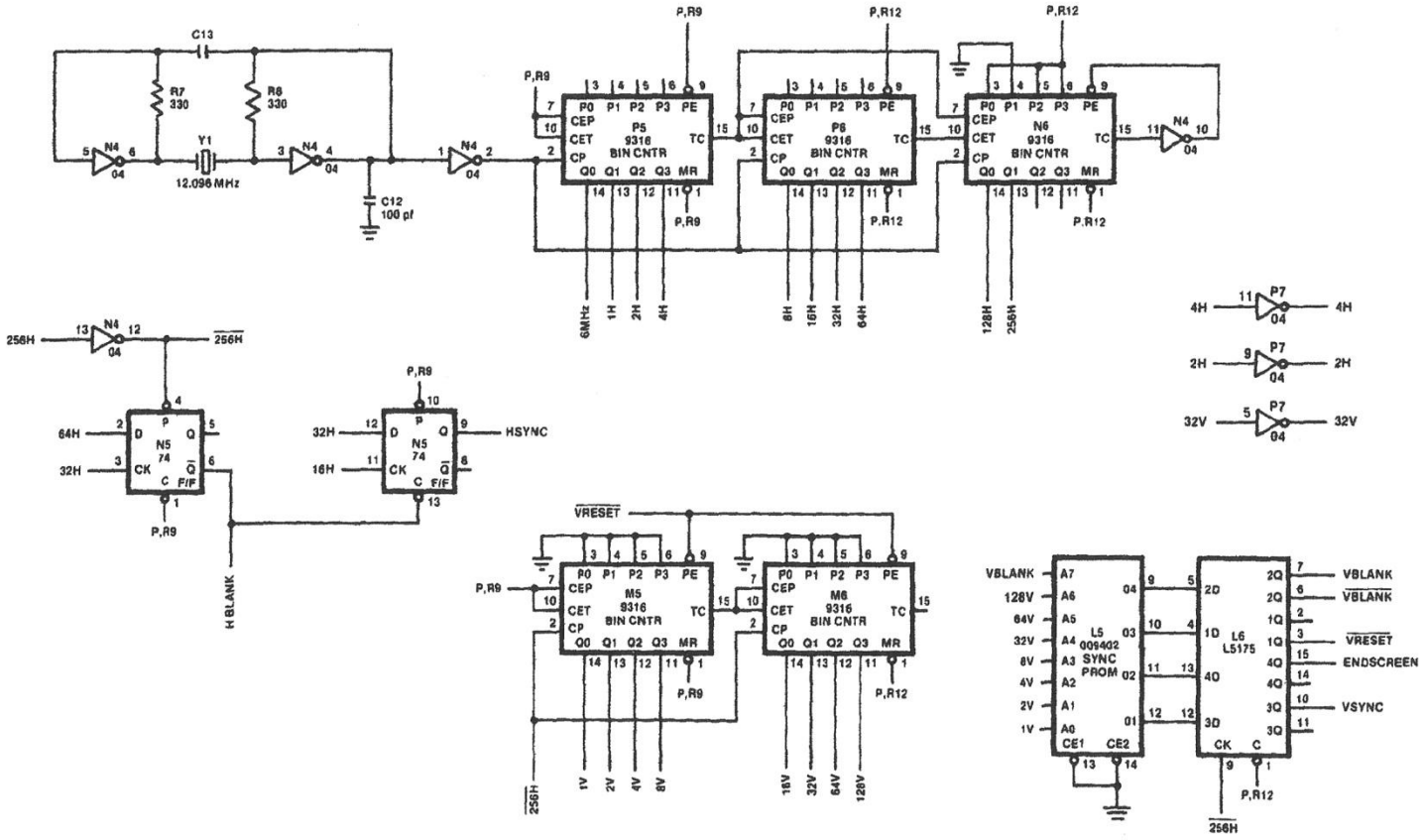



**Figure 9.** Sync counter schematic

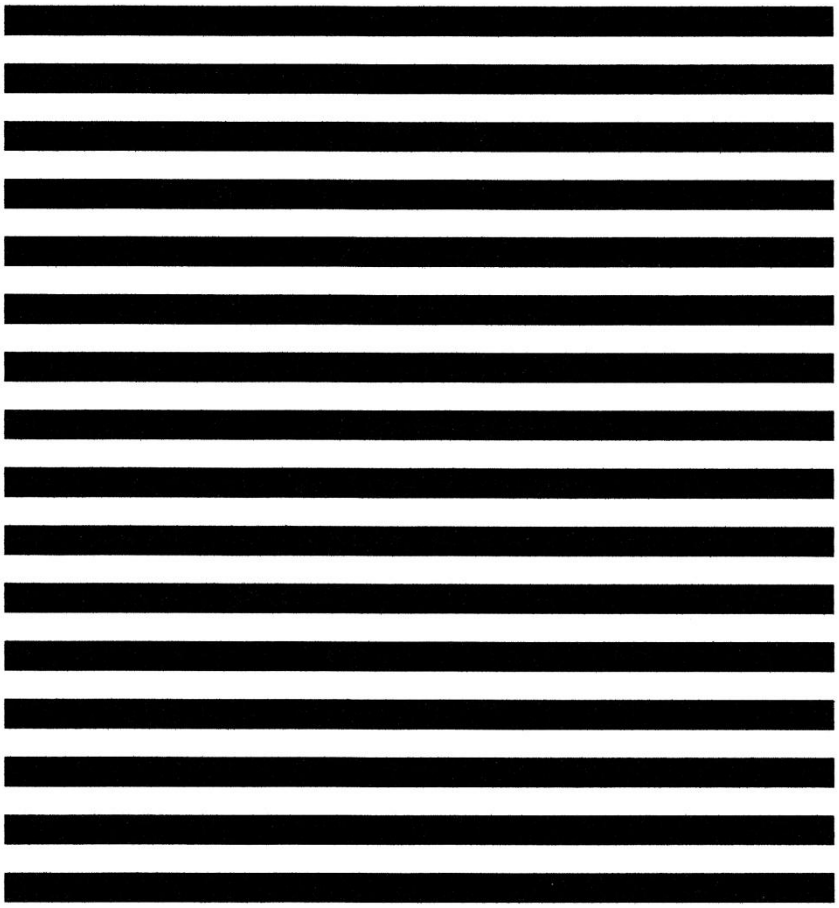


**Figure 10.** 16V vertical counter

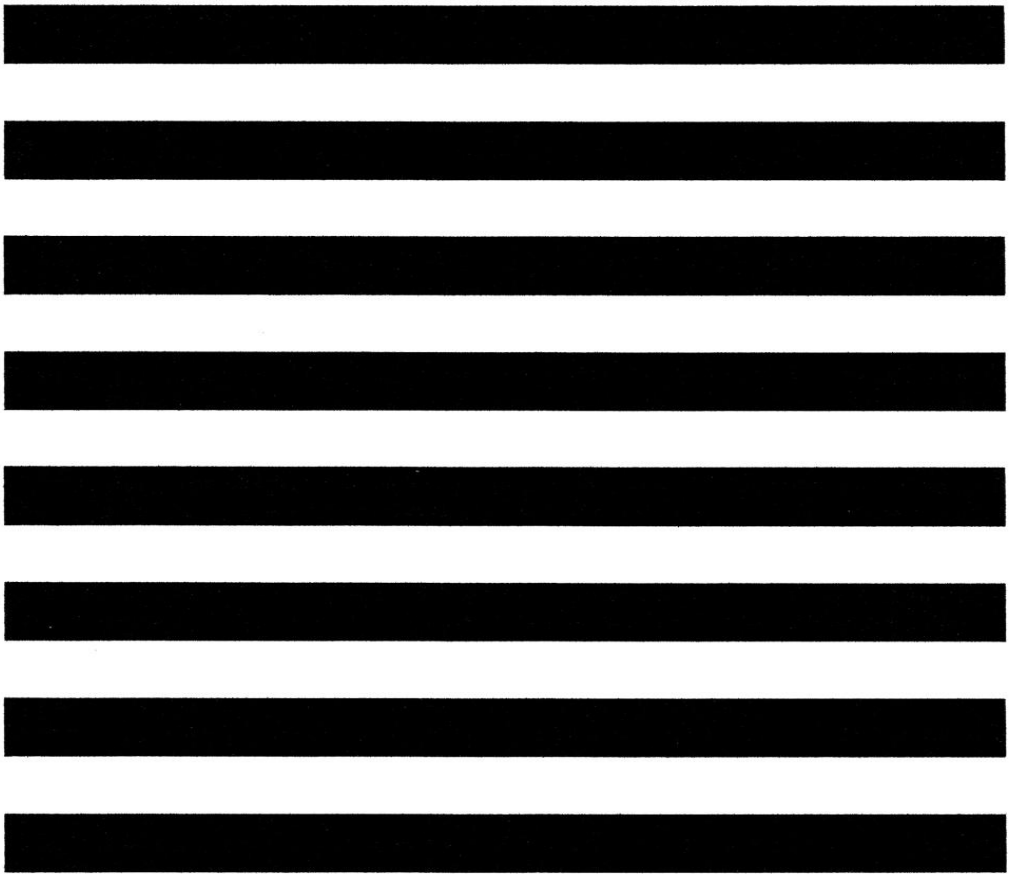


**Figure 11**. 32V vertical counter

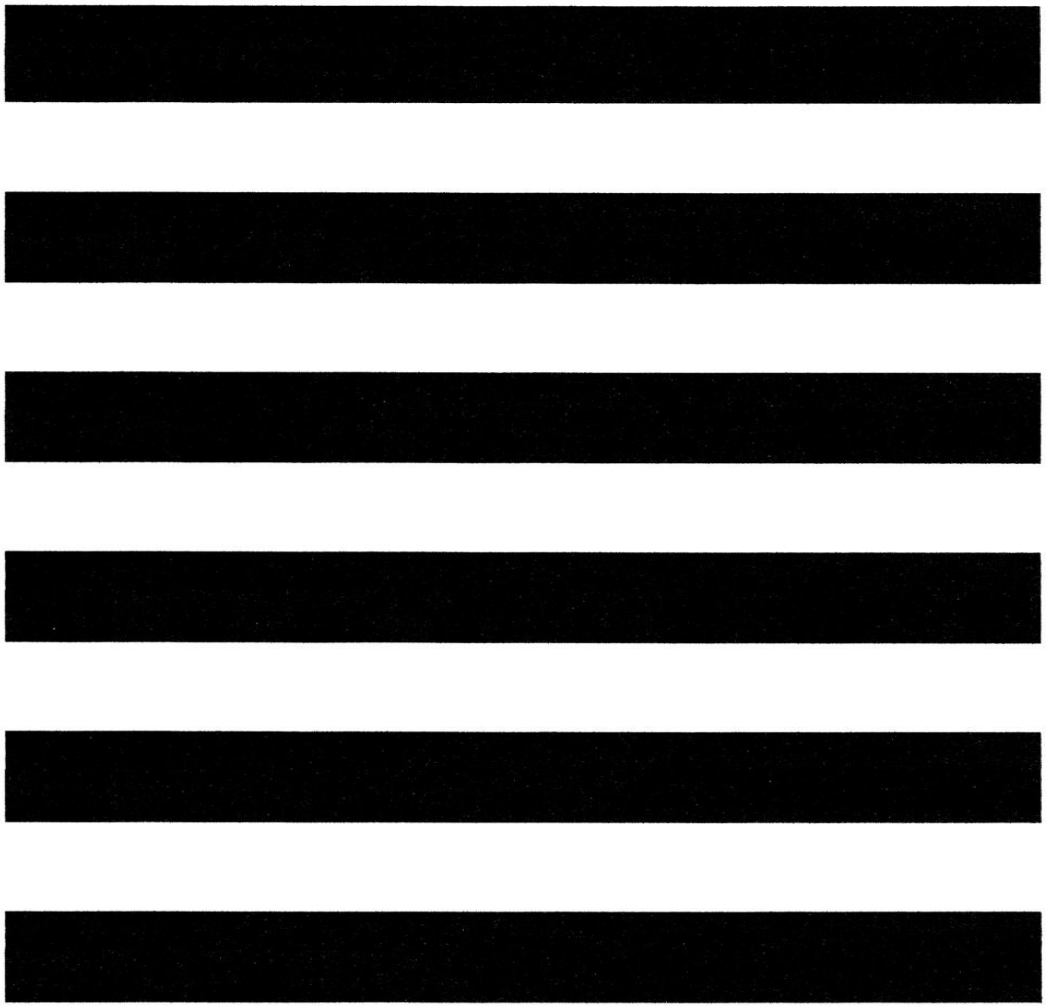


**Figure 12.** 64V vertical counter

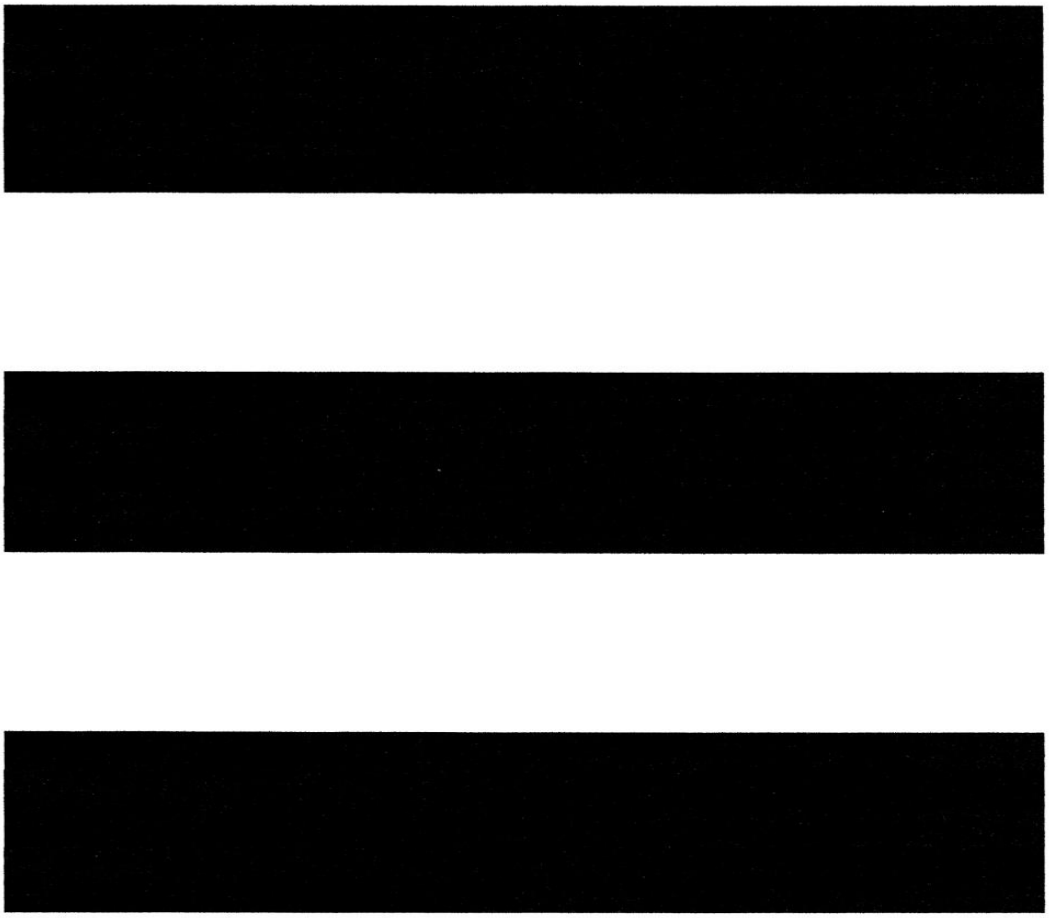


**Figure 13.** 128V vertical counter

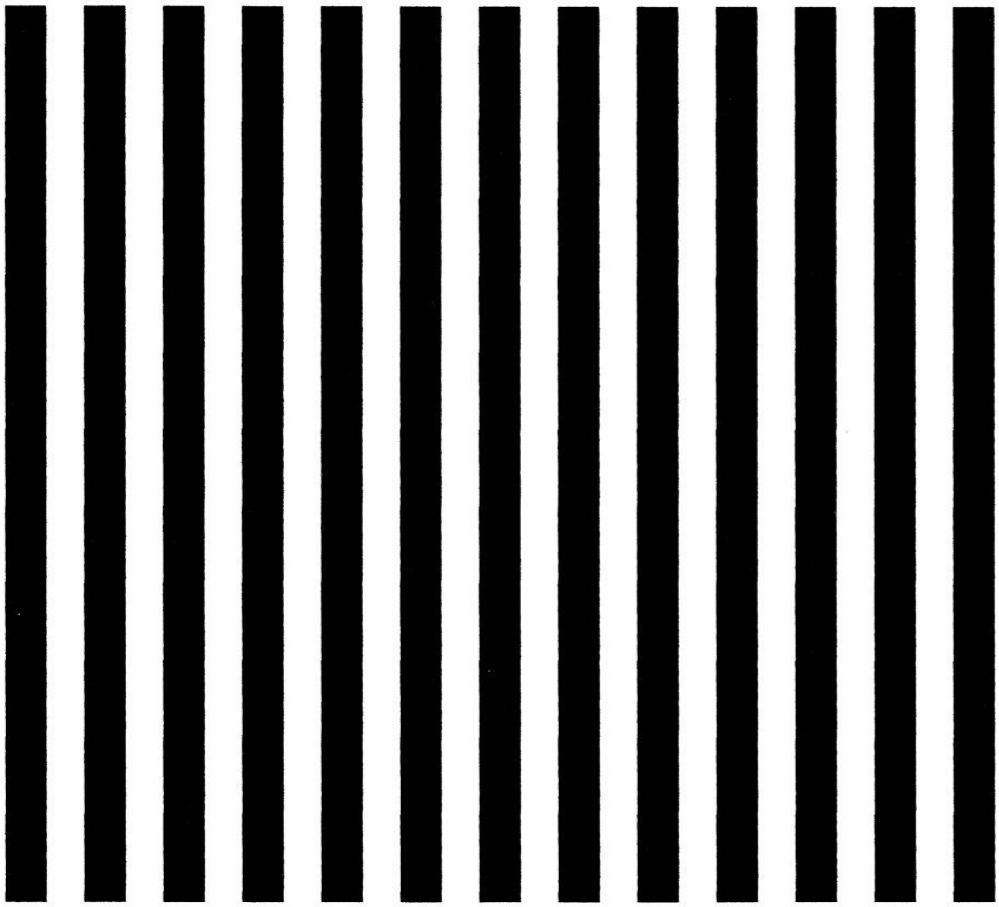


**Figure 14.** 16H horizontal counter

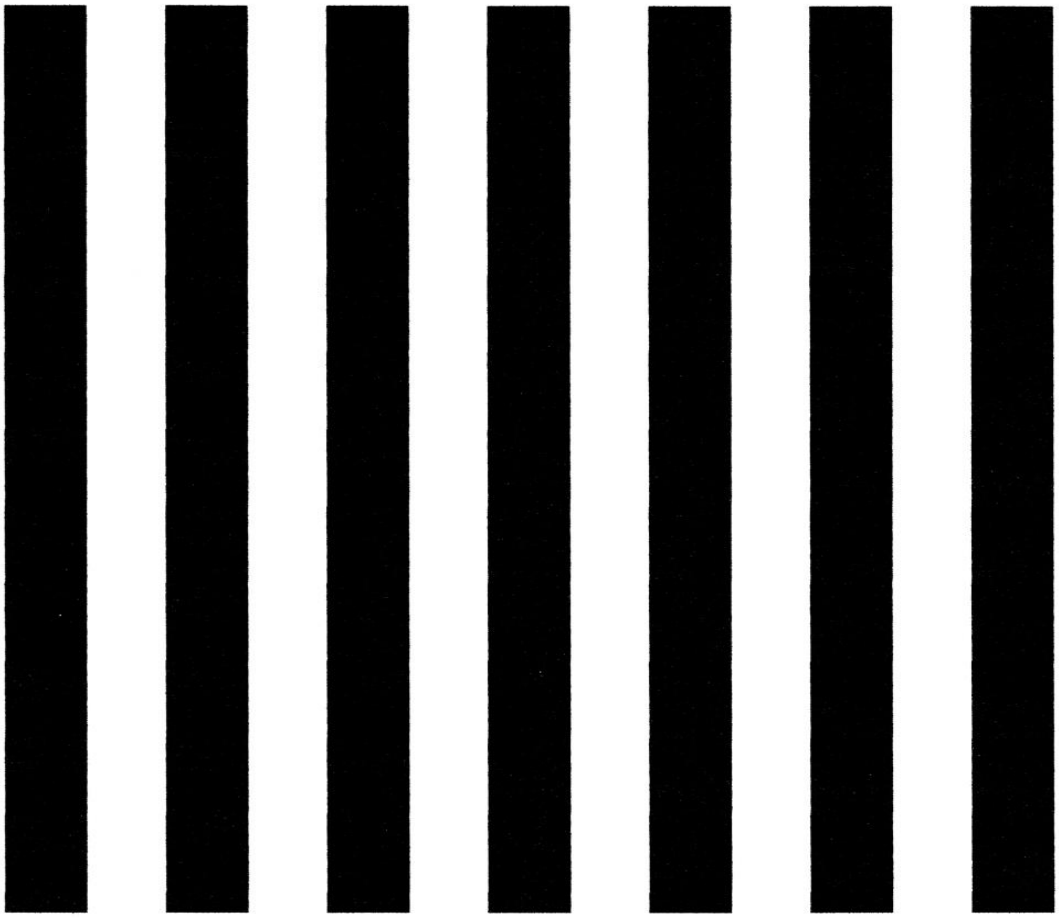


**Figure 15.** 32H horizontal counter

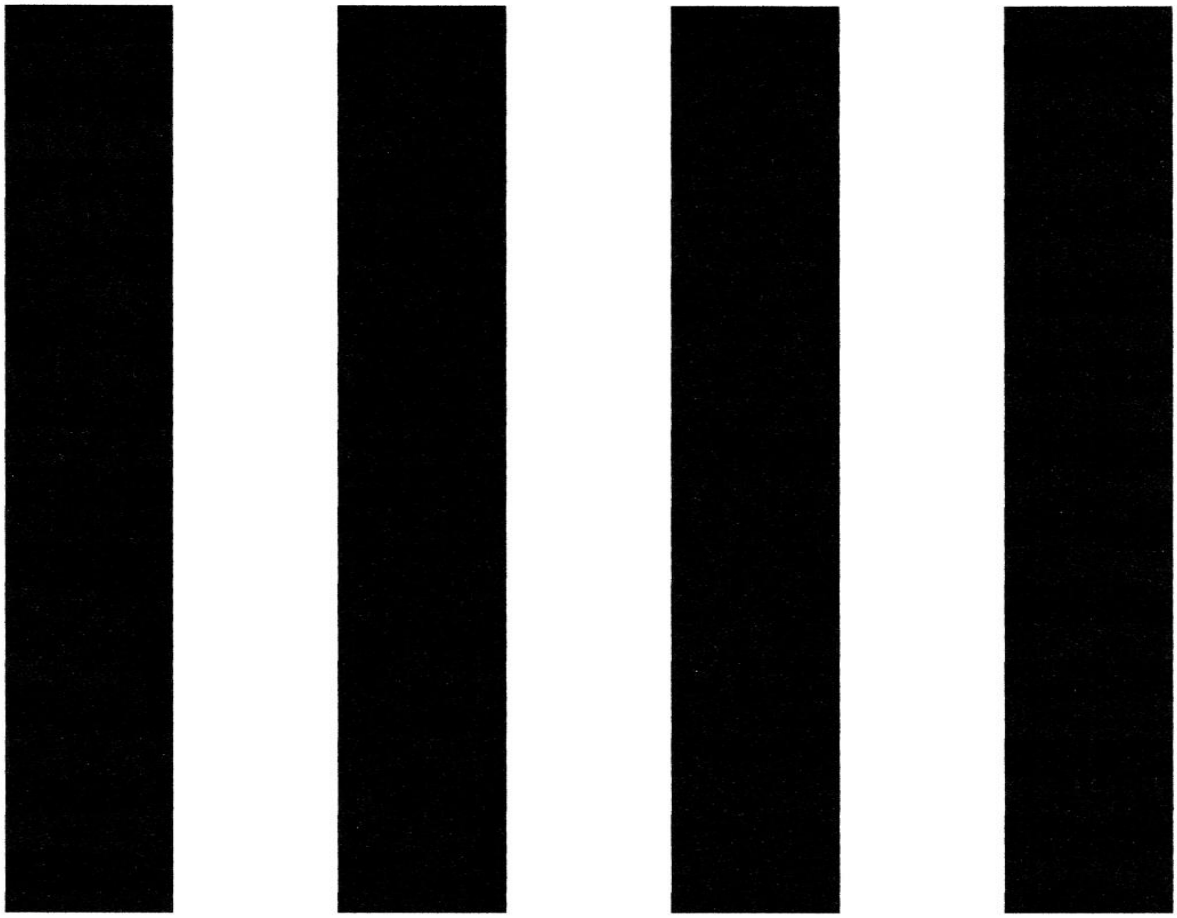


**Figure 16.** 64H horizontal counter

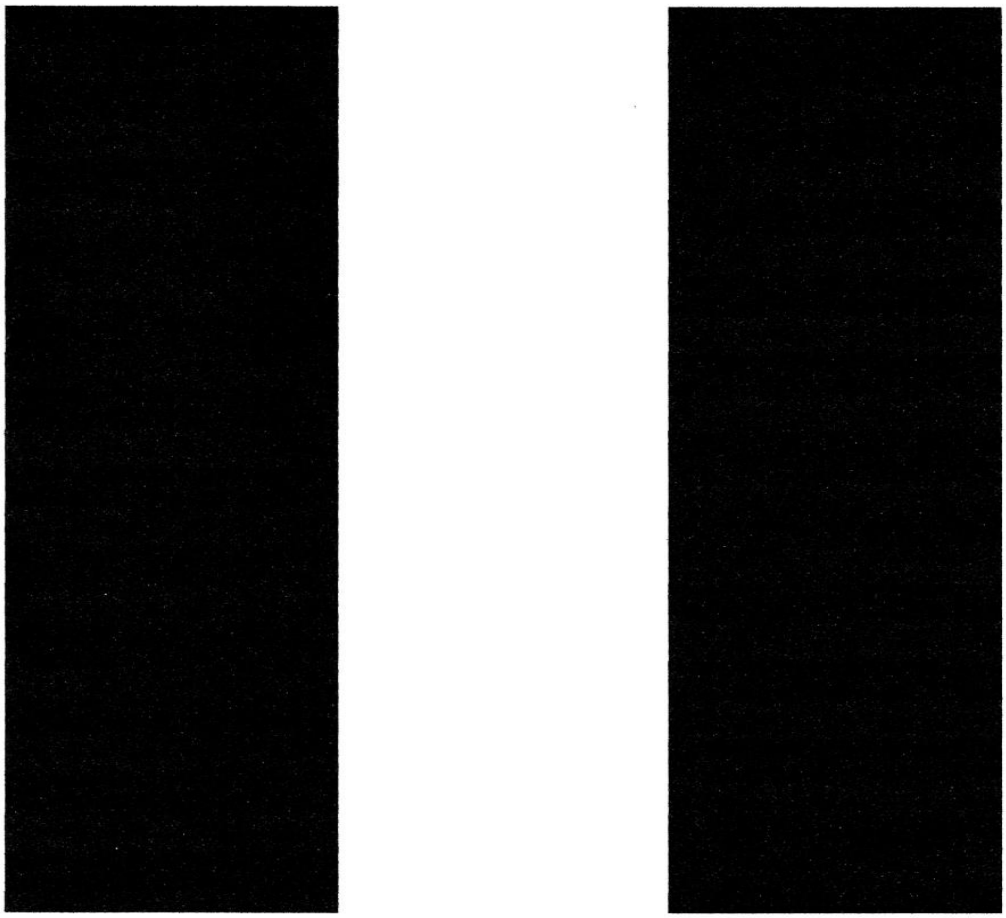


**Figure 17.** 128H horizontal counter

# Chapter 6

## Basic Electrical and Digital

Basic electrical and digital information is an area that is sometimes neglected in training. Simply learning electronics from a book is not very effective. A mechanic who has gone to school and passed the course work still needs to have hands-on training and experience to be able to fix cars. The same is true with computer technicians, engineers, and hobbyists. Anyone who works on computer equipment that lacks experience should take a lab training course. You could get that hands-on experience yourself by buying an electronic circuit training kit. That type of kit has electrical parts and wires that plug into a circuit board to demonstrate different kinds of circuits. Working with a kit like that for a few weeks would go a long way toward helping you become familiar with the electrical circuitry in anything, including computers.

You need to know the following basic concepts to understand computers.

1. Electricity is generated from the flow of electrons resulting in a positive or negative charge. You cannot see electrons; you can only see their effects.

2. Electricity has voltage and current. Because you cannot see electrons, consider a water hose for comparison. If you have a small opening at the end of a garden hose, you would have pressure. The distance the stream of water would squirt would be a good measurement of the water pressure. Consider that to be the measurement of voltage. Now, say you have a large opening at the end of the hose and you fill up a swimming pool. How fast you fill up the pool is flow, which can be compared to electric current (also called amperage).

3. If you multiply voltage times current you get power. Power is expressed as watts. If you read the labels on your home appliances, you'll see voltage, amps and power.

4. Circuits are connected in series, parallel, or the combination of both.

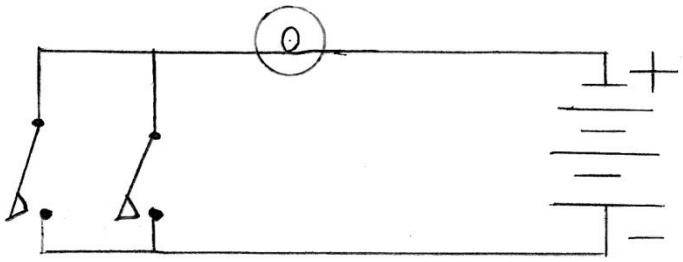
5. A wire is a garden hose of electrons.

6. Switches turn electricity on or off. It can be toggle switch or a press switch which makes a connection when you are pressing it.

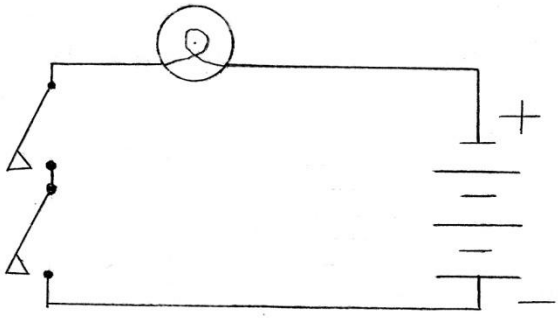
7. Batteries come in different sizes, voltages, currents, and power. Generally speaking, little batteries have small power and big batteries have large power. Lithium batteries have a lot of power for their size. If you want full power from a battery, or other power sources, connect the + to the – (direct current power), or short the two AC (Alternating Current) plug prongs that go into a wall socket at home. But you never want full power. Full power will smoke, burn, get red, catch fire, blow fuses or cause your battery to go dead.

8. Resistors come in different sizes and values. They are used to limit power in circuits.

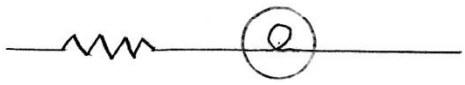
The two most important circuits ever invented are series and parallel. If they were patented, they would be worth billions of dollars.



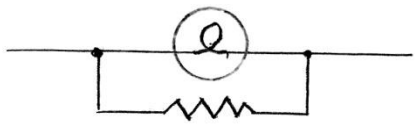

9. **Figure 18.** Parallel Circuit - Either of the switches can turn on the light




10. **Figure 19**. Series Circuit - Both switches need to be on to turn on the light



11. **Figure 20.** Resistor in series - A resistor in series with a light bulb would limit current and dim the light of the bulb.

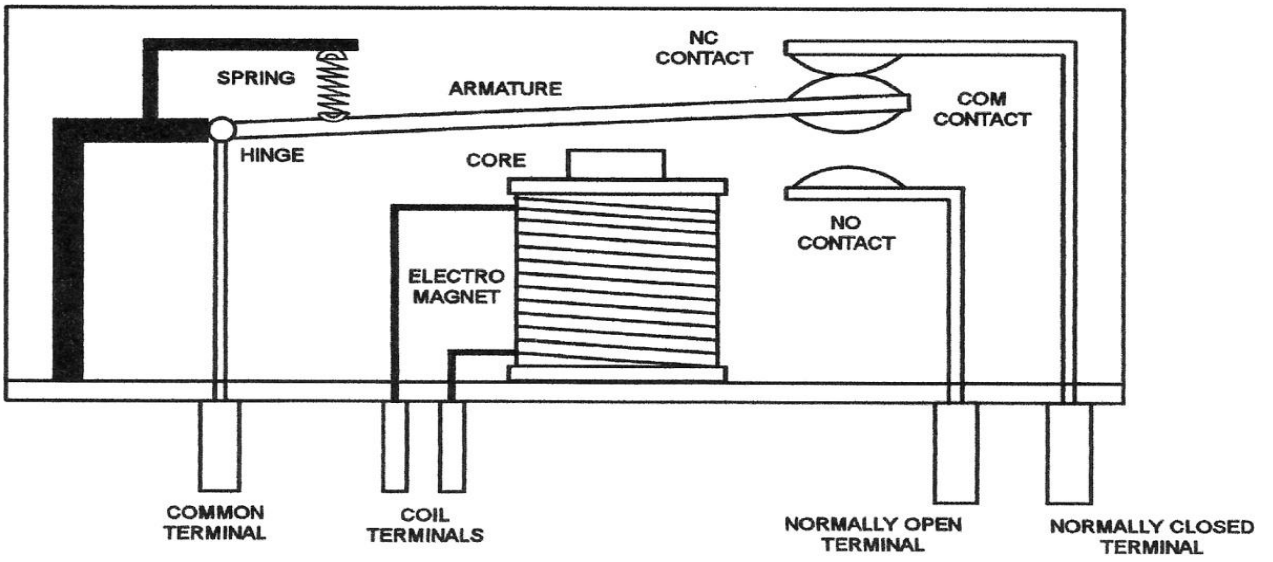


12. Figure 21. Resister in parallel - A resistor in parallel would increase the current and the bulb brightness would stay the same.

13. A fuse is a certain size wire that burns apart when the current rating is exceeded, thus opening the circuit. A ¼ amp fuse would be a small diameter wire and a 100 amp fuse would be a larger diameter wire.

14. A relay is an electrically controlled switch. It works by an electro-magnet moving a piece of steel that turns on or off a switch.

**Figure 22**. Relay diagram



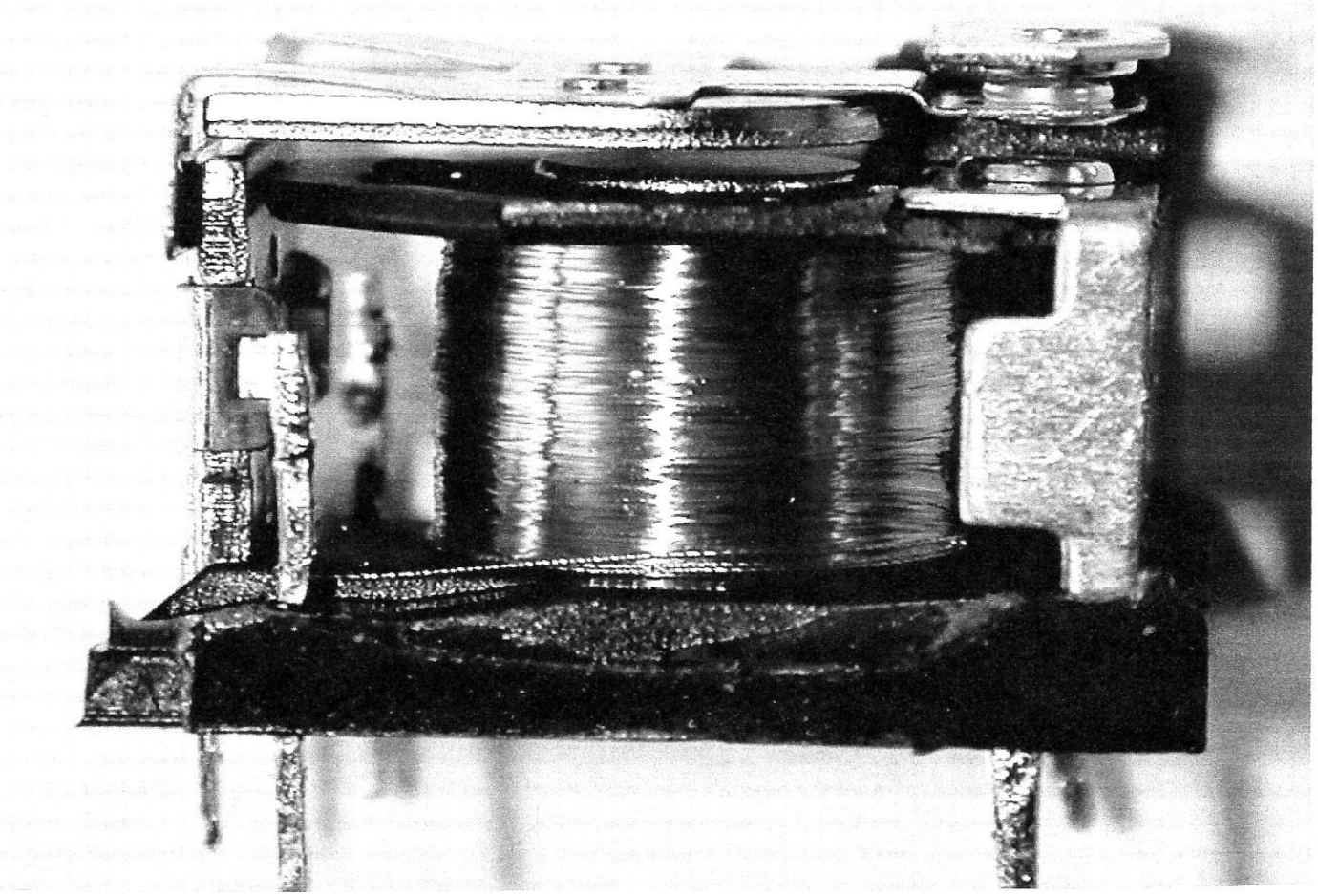



**Figure 23.** Relay image

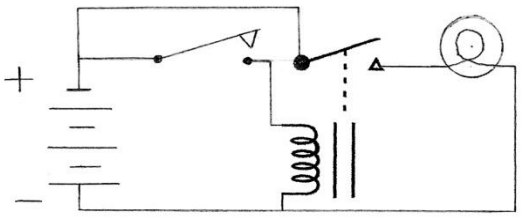


**Figure 24**. Relay circuit

15. On figure 24, when the switch is turned on, power from the battery energizes the coil in the relay. The coil changes the metal in the coil (two vertical lines) into a magnet. Magnetism pulls the metal arm down and the switch contacts in the relay turn on the light.

16. A transistor can work like an amplifier or a relay. Amplifiers are in radios and TV but not discussed in this book. A relay has an input and output and so does a transistor.

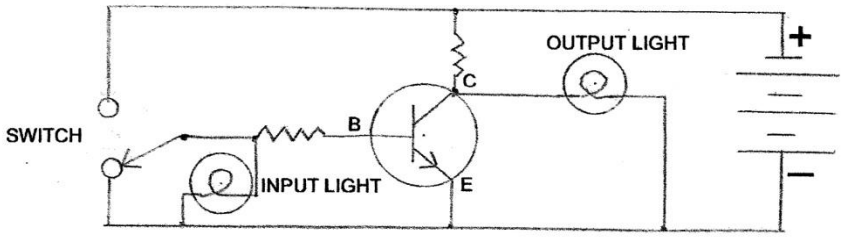



**Figure 25.** Transistor circuit

A transistor connects the emitter (E) to the collector (C) when the base (B) has electrical power. When the switch is down, the base has no power and the transistor is off. Power goes through the upper resistor and turns on the output light. When the switch is up, the base has power and the collector and the emitter connect. The collector shorts out the power through the upper resistor to the negative of the battery, which removes the power from the output light.

17. Inverter - When you flip the switch up on Figure 25, the input light comes on, indicating a "1", and the output light is off, indicating a "0". When you flip the switch down, the input light goes off, indicating a "0", and the output light turns on, indicating a "1". Figure 25 is an inverter and Figure 26 is the symbol for it.

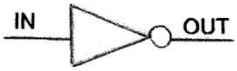



**Figure 26.** Inverter symbol

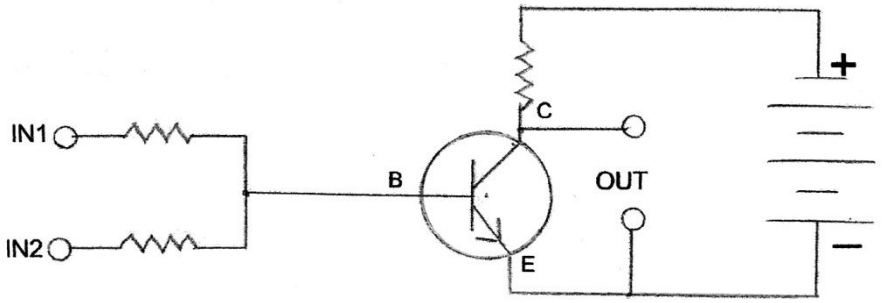



**Figure 27.** Nor gate

18. This Nor Gate circuit is similar to the parallel circuit with 2 switches and a light bulb. If either Nor Gate input is a "1" (a "1" = 5 volts), or both inputs are "1", the output will be "0". That means the transistor is on, with C and E connected. If both IN1 and IN2 are "0", then the transistor is off and the output is “1”.

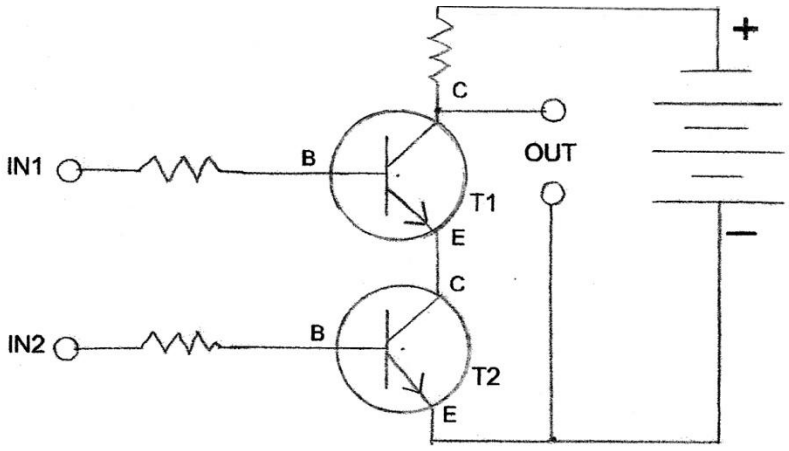



**Figure 28.** Nand gate

19. This Nand Gate circuit is similar to the series circuit with two switches and a light bulb. A "1" on IN1 turns on T1, and a "1" on IN2 turns on T2. With both inputs, a "1" the output would be a "0". Any other combination gives you a "1". Figure 29 is the diagram symbol for it.

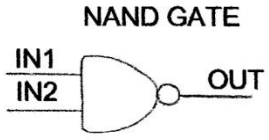



**Figure 29.** Nand gate symbol

20. Simple memory - This is an example of a fuse memory that has 2 bits with 4 binary numbers.

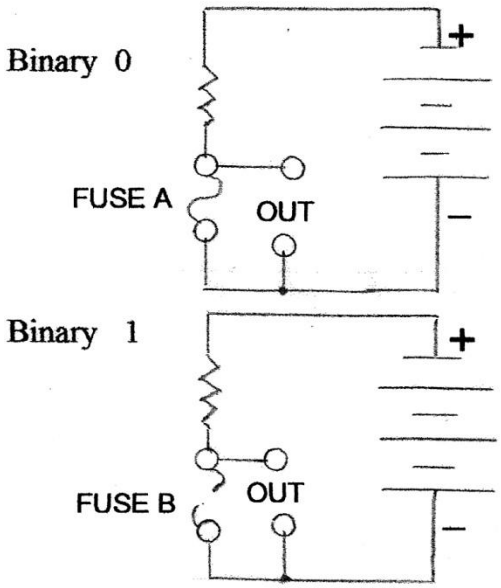



**Figure 30.** Simple memory

With this type of memory,

A fuse blown = "1"
A fuse good = "0"

All possible output combinations are:

| Fuse B | Fuse A |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

According to the chart above, this simple memory can store four different binary numbers. With power off, you will not lose memory. PROMs are memories that have fuses inside just like you see here on Figure 30. A programming machine burns the fuses. A ROM operates the same as a PROM, but ROMS are only programmed at the factory.

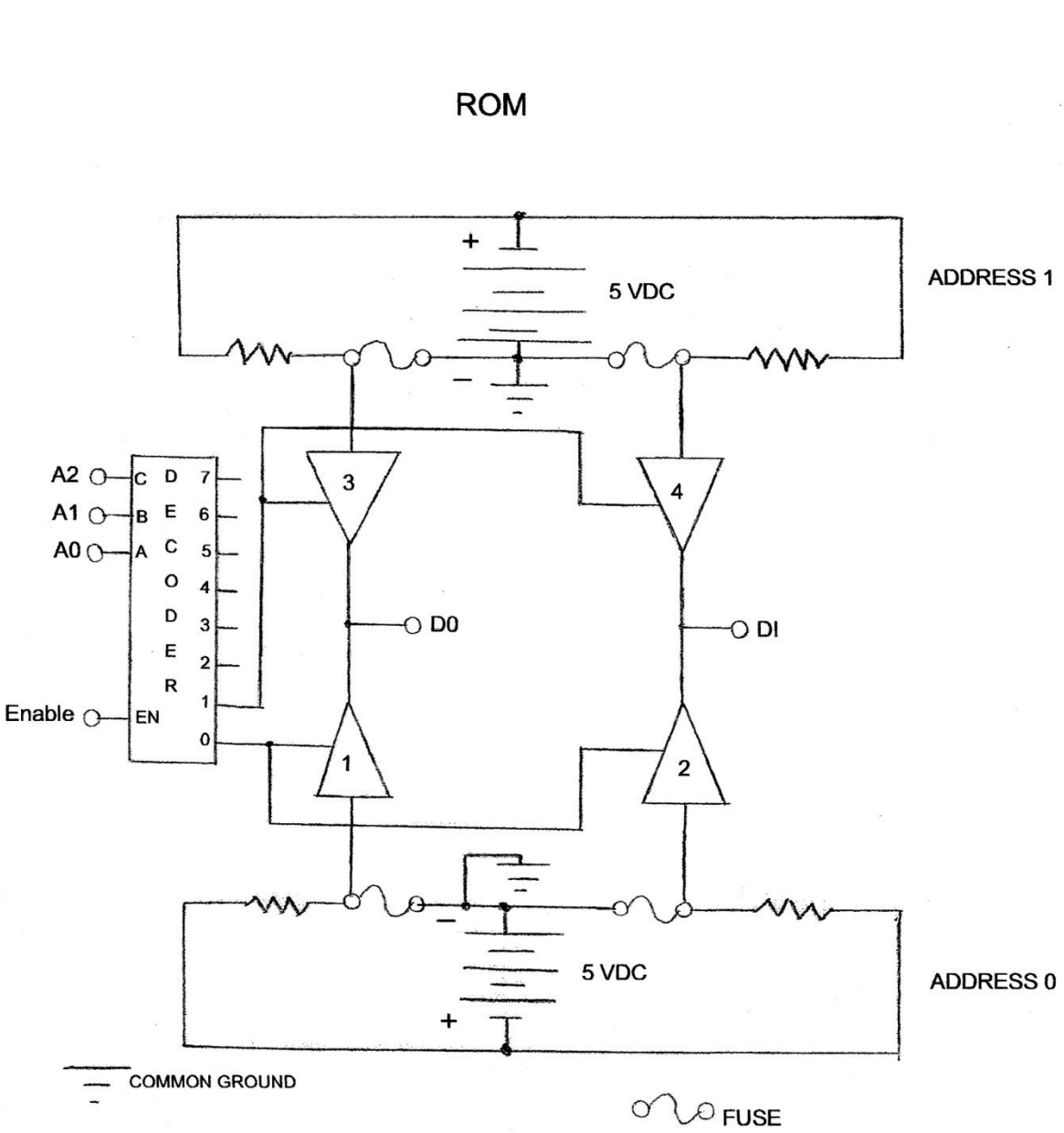



**Figure 31.** Fuse ROM

21. This fuse Rom has 2 addresses and 2 data bits. The circuit explanation is:

Parts 1, 2, 3, and 4 are enabled buffers. A buffer will pass a "1" or "0" from its input to its output when it receives a "1" on its enable input. The part that enables the buffers is the decoder. When the decoder is enabled (a "1" on enable), the binary address input number on A0, A1, and A2 will select a decoder output. This Rom has only two addresses, address 000 and address 001. When the Rom has address 000 on its inputs, decoder output 0 will output a "1" which enables the lower buffers, (parts 1 and 2) to output lower fuse data to D0 and D1. When the Rom has address 001 on its inputs, decoder output 1 will output a "1" which enables the upper buffers, (parts 3 and 4) to output upper fuse data to D0 and D1.

If we wanted a larger ROM with 8 addresses and 8 data bits, you would need a total of 64 fuses, 64 resisters and 64 buffers. Each decoder output (0, 1, 2, 3, 4, 5, 6, 7) would enable one of the 8 addresses. At each address, 8 buffers would output an 8 bit data number.

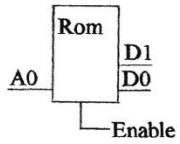



**Figure 32.** Final diagram for two addresses and two data bits

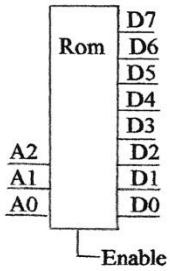



**Figure 33.** Final diagram for eight addresses and eight data bits

22. Nand gate flip-flop. A flip-flop is used to store a "1" or "0". The most basic flip-flop is a 2 Nand gate flip-flop. A Nand gate will have an output of "0" when both inputs are "1". Use that principle to understand how the Nand gate flip-flop operates.

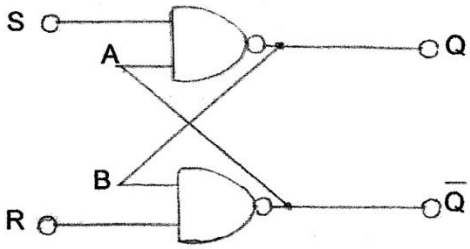



**Figure 34.** Nand gate flip-flop

To store a "0" on Q, S = "1" and R = "0". The circuit explanation is, since R = "0", one input of that Nand gate is "0", so $\overline{Q}$ must be "1". Q must be "0" because S = "1" and A = "1". To store a "1" on Q, S = "0" and R = "1".

The Nand gate flip-flop can be re-written, as an S R flip flop.

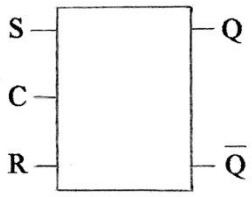



**Figure 35.** S R flip-Flop

The difference with this rewritten circuit is the addition of a clock. When the clock goes from "0" to "1", S and R data gets inputted into the flip-flop.

23. This conversion changes a SR flip flop into a JK flip flop by adding "And" gates to the S and R inputs. If J and K are "1", the flip-flop toggles after every positive clock. The clock could be a push button switch and a resistor or a continuous digital clock. We are going to use this flip-flop in our next circuit.

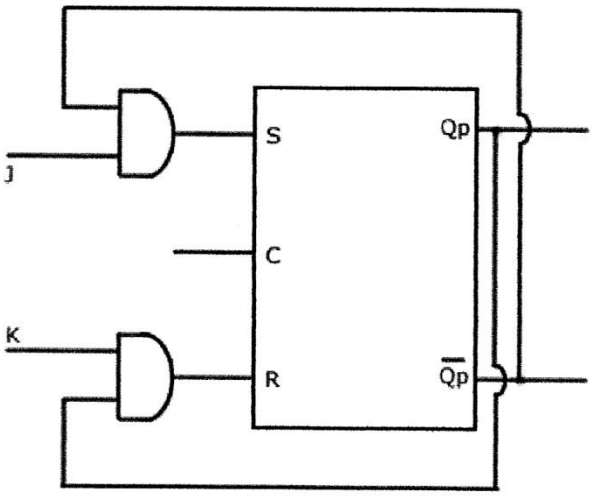



**Figure 36**. S R to J K flip-flop

24. The two "AND" gates that are outside of the flip-flop on figure 36 are now inside on figure 37.

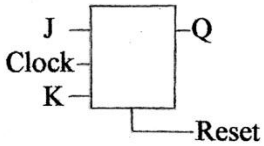



**Figure 37.** J K final diagram. The progression from a Nand gate flip-flop to a JK flip-flop has now been shown.

25. Making a Counter Circuit.

We are going to design a counter circuit. The circuit will count from 0 to 15 in binary. It will use four JK flip-flops and two "And" gates to take care of carry. When you count in decimal, you add one to every count until you get up to 9. With decimal 10, you get a carry. When counting in binary, carry happens more often. For example, when counting from 0 to 4 in binary, (0, 1, 10, 11, and 100) you will receive a carry at 10 and at 100. The final counter circuit is figure 38.

Now let's operate the counter (figure 38) to see how it works.

JK Flip-flops will not toggle unless J and K inputs are both "1". Flip-flop "A" will toggle after every clock (step), because J and K inputs are connected to a resistor and 5 volts, which equals a "1" on both inputs. Below is the counting process:

| | Counter outputs = | D | C | B | A |
|---|---|---|---|---|---|
| 1. | Press Reset | 0 | 0 | 0 | 0 |
| 2. | Press Clock | 0 | 0 | 0 | 1 |
| 3. | Press Clock | 0 | 0 | 1 | 0 |
| 4. | Press Clock | 0 | 0 | 1 | 1 |
| 5. | Press Clock | 0 | 1 | 0 | 0 |
| 6. | Press Clock | 0 | 1 | 0 | 1 |
| 7. | Press Clock | 0 | 1 | 1 | 0 |
| 8. | Press Clock | 0 | 1 | 1 | 1 |
| 9. | Press Clock | 1 | 0 | 0 | 0 |

1. Pressing reset- resets all 4 flip-flops to 0000.
2. Press clock – Flip-flop A toggles because J and k has a "1". The result= 0001
3. Flip-flop A- j and k inputs are "1" and Flip-flop B- j and k inputs are "1". Press clock. Flip-flop A and B toggles. The result = 0010.
4. Flip-flop A- j and k inputs are "1". Press clock. Flip-flop A toggles. The result = 0011
5. Flip-flop A- j and k inputs are "1", Flip-flop B- j and k inputs are "1" and because of "And" gate E, Flip-flop C- j and k inputs are "1". Press clock. Flip-flop A, B, and C will toggle. The result= 0100.
6. Further pressing of clock will make the counter count up to 15 or 1111 in binary.

Please notice that the binary output numbers just recorded go from right to left (D C B A), and the outputs on diagram 38 go left to right (A B C D).

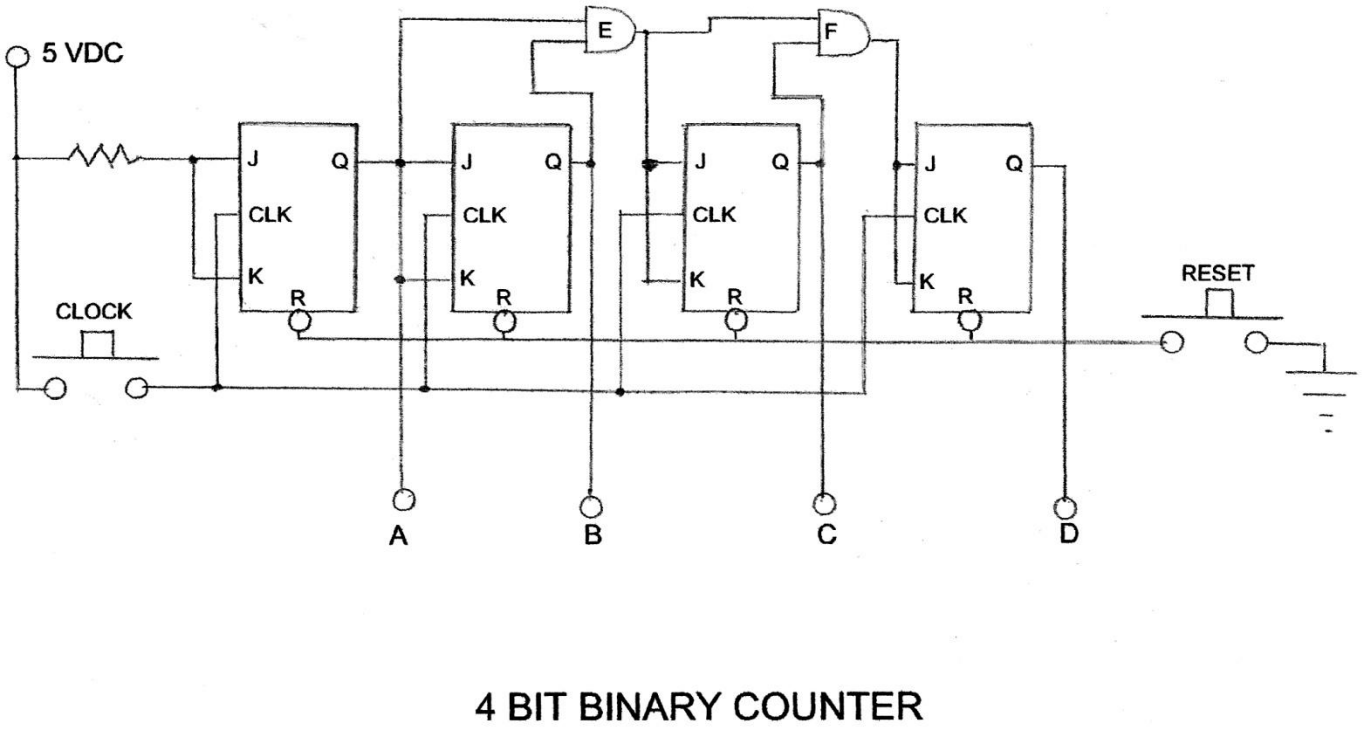



**Figure 38.** 4-bit binary **co**unter

I will explain this counter a second time in a different way. The end result will still be the same, the counter counting in binary from 0000 to 1111. As before, the Flip-flops will not toggle unless J and K inputs are both "1". A clock (step) is required to toggle the Flip-flops, and "And Gates" are used as a carry to the next digit. The sequence of operation said differently is:

Flip-flop A- Toggles with every clock.
Flip-flop B- Toggles every other clock.
Flip-flop C- Toggles only when Flip-flop A and B are "1".
Flip-flop D- Toggles only when Flip-flop A, B and C are "1".

Note: 1. Counter outputs A and B are connected to "And" gate E inputs. "And" gate E enables Flip-flop C to toggle.
2. "And" gate E output and counter C output are connected to "And" gate F inputs. "And" gate F output enables Flip-flop D to toggle.

26. RAM-1 address 1 data bit Computer memory.

We are going to design a computer memory. It starts with a Nand gate flip-flop. When you add a clock to the Nand gate flip-flop, you get an "S R" flip-flop. When you add an inverter from S to R and you get a D flip-flop. We now have the flip-flop that we are going to use for our computer memory.

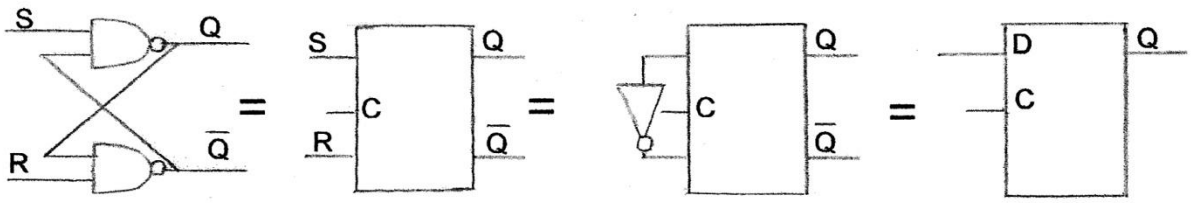



**Figure 39.** S R flip-flop conversion to D flip-flop

The way a D flip-flop operates is, a "0" on D input goes to Q output, after a positive clock or a "1" on D input goes to Q output, after a positive clock. For our computer memory one new part needs to be added. It is the Enabled Buffer.

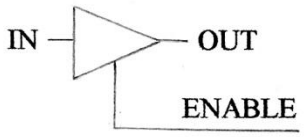



**Figure 40.** Enabled buffer

When the buffer has a "1" on enable, a "0" on the input gives a "0" on the output, or a "1" on the input gives a "1" on the output. When the buffer has a "0" on enable, the output is the same as not connected. A shared bus requires this, like the data bus on a computer.

The example below is our computer memory put together as a circuit:

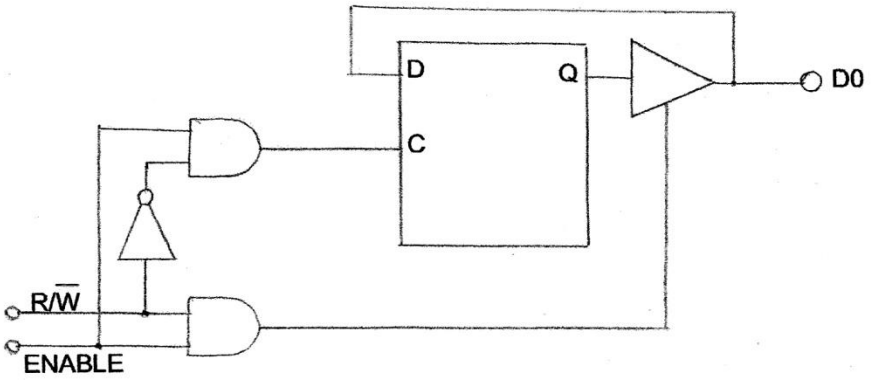



**Figure 41.** Computer memory

**Circuit explanation**

To make a computer memory you need an Enable to be able to turn it on or off. You also need to be able to select read or write, and the memory must to be able to be used on a shared bus that can input or output. Note: An "And" gate will output a "1" if both inputs are "1".

The top "And Gate" is for writing. When writing, R/$\overline{W}$ is a "0". It is changed to a "1" with the inverter. The inverter output ("1") and plus the "1" on Enable make the top "And" gate output a "1". That clocks the flip-flop to input data from D0 ("0" or "1").

The bottom "And Gate" is for reading. When reading, R/$\overline{W}$ is a "1". A "1" on R/$\overline{W}$ plus the "1" on Enable make the bottom "And" Gate output a "1". That turns on the buffer to output flip-flop data ("0" or "1") to D0.

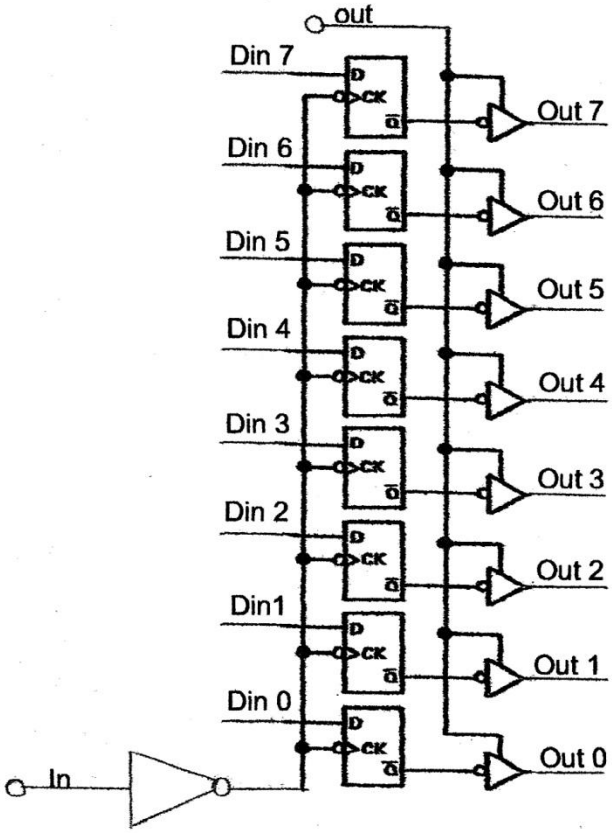



**Figure 42.** 8-bit register

When the IN signal goes from "0" to "1", 8 flip-flops will clock and store an 8-bit binary number. A "1" on the OUT signal will allow you to read out the number that was stored by enabling the 8 output buffers. This register is basically the same circuit that is on the last page (figure 41), except there are 8 bits and no read write signal. The enabled buffers on this part are tri-state, which means that when they're not enabled, the outputs are the same as not connected. This register is essentially a one address, 8 data bit computer memory.

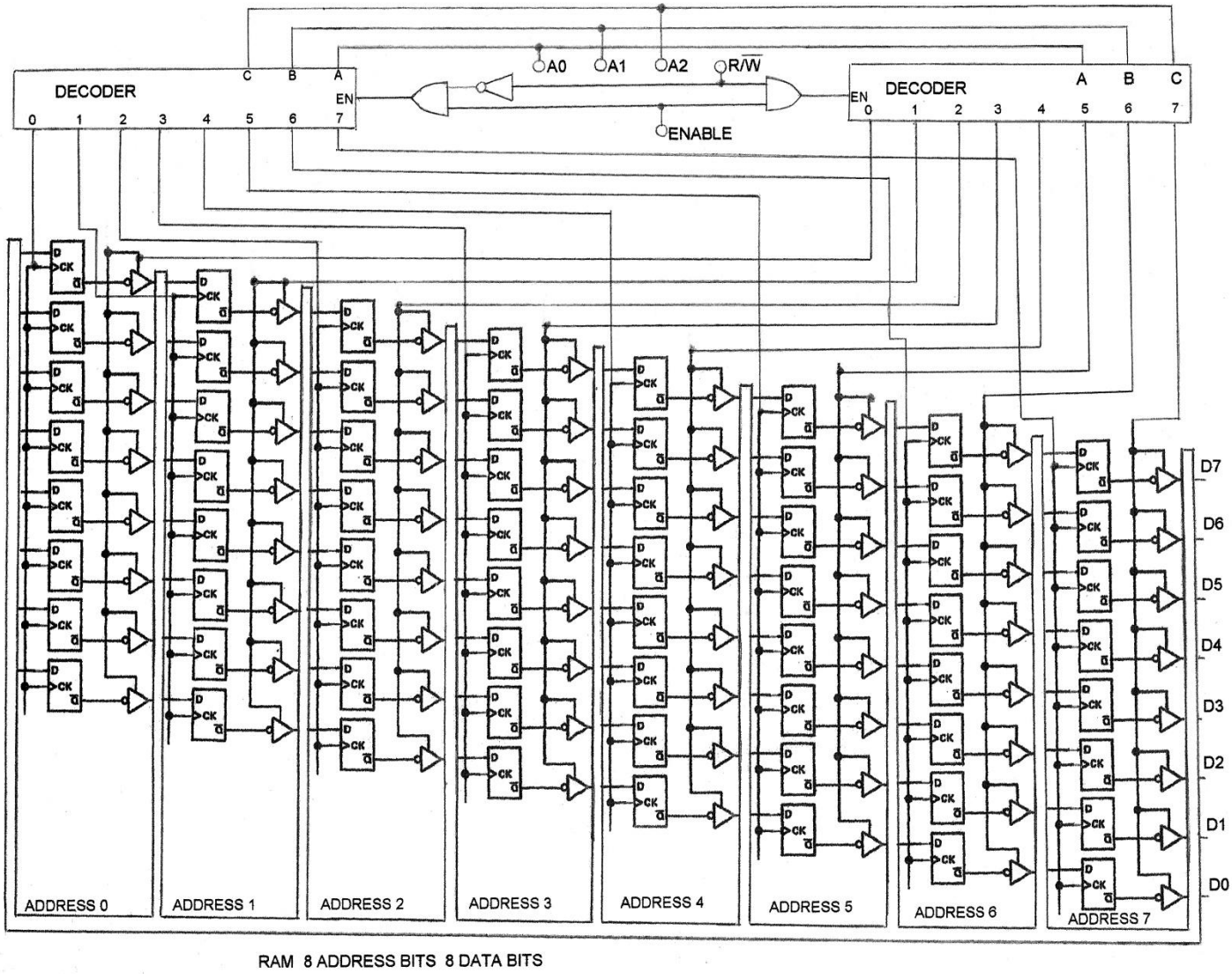



**Figure 43.** RAM eight addresses and eight data bits

Address inputs on every RAM are in binary. Binary numbers can be changed back to decimal numbers and that is what both of the decoders do on this RAM. A0, A1, and A2 are the binary address numbers in and 0, 1, 2, 3, 4, 5, 6, 7 are the decimal equivalent outputs. Each decoder output is for each of the 8 addresses and each address has 8 data bits that can input or output. The left decoder is for write and is enabled when R/$\overline{W}$ is "0" and enable is "1". A selected binary number on the decoder inputs will select one of the decoder outputs, to output "1". That will clock and store an 8 bit number into 8 flip-flops. The decoder on the right is for read and is enabled when the enable is "1" and R/$\overline{W}$ is "1". A selected binary number on the decoder inputs will select one of decoder outputs, to output a "1". That will enable 8 buffers to output an 8 bit stored number. This whole circuit is the same as the 1 address 1 bit RAM (figure 41), except for more addresses and more data bits.

The number of parts for only eight addresses is large. It took many years for memory size to increase to what it is today. The number of parts in a RAM is in the millions. Manufacturers are always designing larger memories.

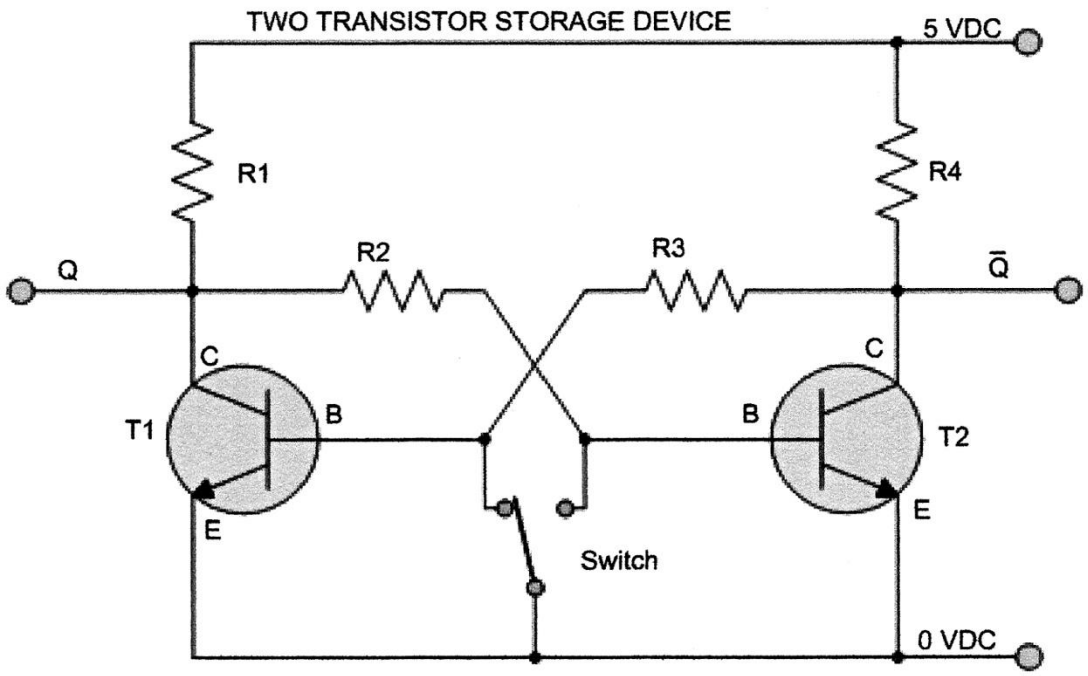



**Figure 44**. Two transistor storage device

The switch is the input to the memory and Q and $\overline{Q}$ is the output. One transistor stores a binary one and the other transistor stores a binary zero. The resistors R1, R2, R3, and R4 limit power so nothing burns out. As you'll remember from the theory at the beginning of the book, when Base (B) to Emitter (E) has power, the Emitter (E) to Collector (C) connects. The explanation is:

1. The switch is moved to left terminal (simulates a "1" on the input).
   A. Power from 5 VDC to R4 to R3 to the base of T1 is cut off by the switch.
   B. Transistor 1 is off.
   C. Power also goes from 5 VDC to R1 to Q output to R2 then to base of T2
   D. T2 has power from base to emitter.
   E. T2 turns on and Emitter to Collector connects.
   F. Output $\overline{Q}$ is a zero.
   G. Output Q is a one.

2. The switch is moved to right terminal (simulates a "0" on the input).
   A. Power from 5 VDC to R1 to R2 to the base of T2 is cut off by the switch.
   B. Transistor 2 is off.
   C. Power also goes from 5 VDC to R4 to $\overline{Q}$ output to R3 then to the base of T1.
   D. T1 has power from the base to the emitter.
   E. T1 turns on and Emitter to the Collector connects.
   F. Output $\overline{Q}$ is a one.
   G. Output Q is a zero

30. D flip-flop

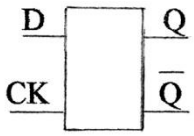



**Figure 45.** D flip-flop

The purpose of a D flip-flop is to store a "1" or "0". A "0" on the D input makes Q output a "0" after a positive clock. A "1" on the D input makes Q output a "1" after a positive clock.

31. And Gate

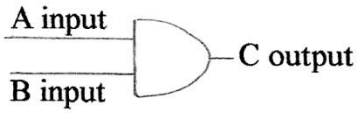



**Figure 46**. And gate

If A and B are "1" then C is a "1". C= "0" with any other combination

32. Or Gate

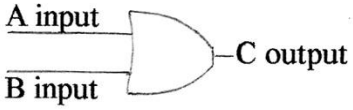



**Figure 47.** Or gate

If either A or B is a "1" or both are "1" then C = 1. C= "0" with any other combination

33. Binary Counter that counts from 0 to 15

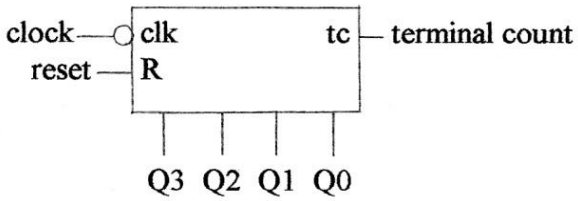



**Figure 48**. 0-15 Binary counter

This is what the counter (Figure 48) signals look like on an oscilloscope.

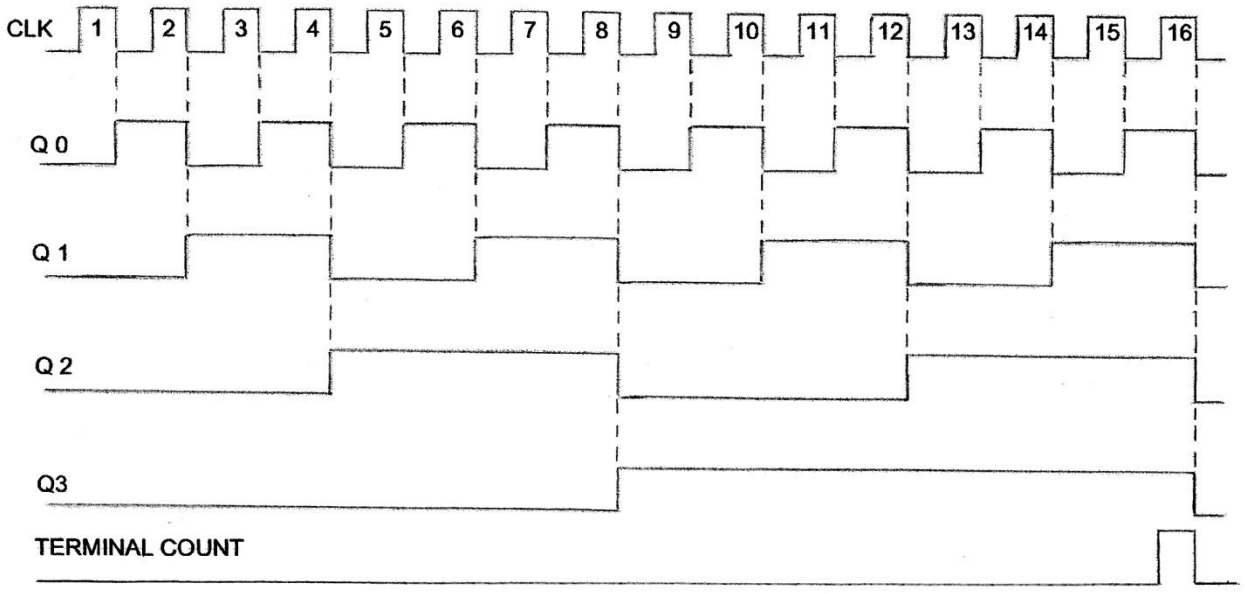



**Figure 49.** Counter oscilloscope signals

When the clock on figure 48 goes from "1" to "0", the counter will increment. The outputs after each clock are:

| | 3210-Q outputs | | |
|---|---|---|---|
| 0. | 0000-after Reset | | |
| 1. | 0001-starts counting | 8. | 1000 |
| 2. | 0010 | 9. | 1001 |
| 3. | 0011 | 10. | 1010 |
| 4. | 0100 | 11. | 1011 |
| 5. | 0101 | 12. | 1100 |
| 6. | 0110 | 13. | 1101 |
| 7. | 0111 | 14. | 1110 |
| | | 15. | 1111 ends here with terminal count a "1" |
| | | 16. | 0000 Count 16 makes the counter go back to zero |

34. RAM - how to read and write

Memories come in different sizes and bit widths. A RAM is a rewritable memory that can store binary numbers. When a location number is supplied to the addresses on a RAM, data numbers can be written in or read out by using the R/$\overline{W}$ input. If you select an address on a Ram, put a "0" on R/$\overline{W}$, and a "1" on enable, you will write a data number into that Ram address. If you select an address on a Ram, put a "1" on R/$\overline{W}$, and a "1" on enable, you will read out the data that was stored in that Ram address.

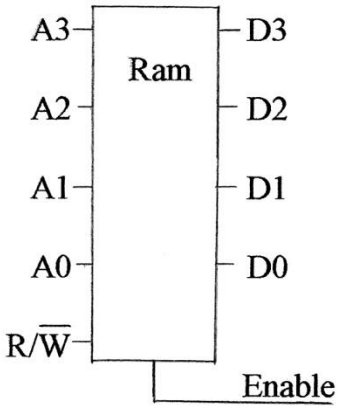



**Figure 50.** RAM - 16 addresses and 4 data bits

Selecting a RAM address, deciding whether to read or write, and enabling the RAM must become routine. I am going to use 16 addresses on Figure 50 Ram to write in 16 random data numbers. I will select an address, put a "0" on R/$\overline{W}$, choose a random data number, and then put a "1" on enable. I will go through this process a total of 16 times. The random data numbers to be stored at each address are:

| Decimal address | Binary address A3 A2 A1 A0 | Decimal Random numbers | Binary random numbers D3 D2 D1 D0 |
|---|---|---|---|
| 0 | 0 0 0 0 | 5 | 0 1 0 1 |
| 1 | 0 0 0 1 | 1 | 0 0 0 1 |
| 2 | 0 0 1 0 | 3 | 0 0 1 1 |
| 3 | 0 0 1 1 | 3 | 0 0 1 1 |
| 4 | 0 1 0 0 | 10 | 1 0 1 0 |
| 5 | 0 1 0 1 | 15 | 1 1 1 1 |
| 6 | 0 1 1 0 | 4 | 0 1 0 0 |
| 7 | 0 1 1 1 | 7 | 0 1 1 1 |
| 8 | 1 0 0 0 | 6 | 0 1 1 0 |
| 9 | 1 0 0 1 | 1 | 0 0 0 1 |
| 10 | 1 0 1 0 | 2 | 0 0 1 0 |
| 11 | 1 0 1 1 | 7 | 0 1 1 1 |
| 12 | 1 1 0 0 | 9 | 1 0 0 1 |
| 13 | 1 1 0 1 | 8 | 1 0 0 0 |
| 14 | 1 1 1 0 | 11 | 1 0 1 1 |
| 15 | 1 1 1 1 | 14 | 1 1 1 0 |

Note: The numbers above are recorded in two different numbering systems. The decimal address numbers have the same number value as the binary address numbers. The decimal random data numbers have the same number value as the binary random data numbers.

I'll read what I wrote into figure 50 Ram, by choosing an address, putting a "1" on R/$\overline{W}$, and then a "1" on Enable. I'll go through this process three times to read data out at three addresses, but I could have read out all 16 addresses. The results are:

| Address Decimal | Address Binary A3 | A2 | A1 | A0 | Data numbers read out of the RAM. Decimal | Binary D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 0 | 1 | 0 | 1 | 15 | 1 | 1 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 12 | 1 | 1 | 0 | 0 | 9 | 1 | 0 | 0 | 1 |

The bottom line is, what I wrote into a particular address can be read out of that address at a later time. But if you turn off the power, all random data will be lost.

35. ROMs

Computers can read numbers from a ROM or a RAM but computers cannot write new numbers into a ROM. Since a ROM can only be programmed at the factory it is ideal for storing data that never need to change. Personal Computers (PCs) do not have ROMs. Instead, they have hard drives. Hard drives can read or write. If a PC had ROMs, every update of Windows®, or every new app would require a new ROM to be plugged into the computer. Hard drives, just like ROMs will not lose data when powered off.

36. 8-bit binary to Decimal Decoder

This part inputs a 3-bit binary number and has eight decimal outputs that represent which binary number was input. When an output is on, its output is a "1" with all other outputs at "0".

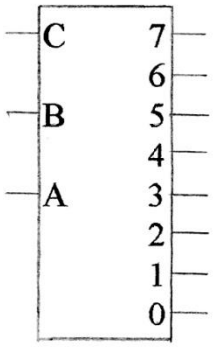



**Figure 51**. 8-bit binary to decimal decoder

| Inputs | Outputs |
|---|---|
| CBA | 01234567 |
| 000 | 10000000 |
| 001 | 01000000 |
| 010 | 00100000 |
| 011 | 00010000 |
| 100 | 00001000 |
| 101 | 00000100 |
| 110 | 00000010 |
| 111 | 00000001 |

37. The binary counter below has added features. Two or more of these counters could be linked together to make a program counter or a computer register.

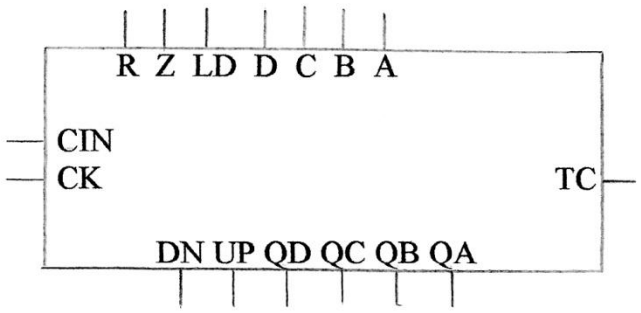



**Figure 52.** Counter with added features

This counter's features include:

1. The LD signal loads a starting number into D, C, B, and A inputs.
2. The UP or DN signal is used to count up or to count down the counter.
3. The CIN signal is a carry input from other counter TC outputs.
4. The CK signal clocks the counter outputs (QD, QC, QB, QA) to increment or to decrement.
5. The R signal resets the counters to 0000.
6. The Z signal tells when the counter number is 0000.
7. The TC signal is a terminal count output that can be linked to the next counter carry input (CIN).

All of the major digital components on the Block Diagram have now been explained in this chapter. You are one step closer to knowing how a computer works. Separate digital integrated circuit parts put on a circuit board, wired, tested, and made into a custom chip was the old method of design. The new way is to program individual digital circuits, which are identified with a code, into a part called the FPGA (field programmable gate array). This book explains a computer system that has separate logic ICs. Many of parts today have so much logic inside, it is impossible to know exactly what is wrong when there is a problem. Many years ago when circuits were simple, computers were easier to understand. Computers have had many improvements over the years but the basic design is still the same.

# Chapter 7

## Timing Circuits

Look at the music box figure 53 on the next page. On the back, out of view, is a wind up key. The spring that gets wound up is on the inside of the cover labeled "Romance". That spring has stored power. The power from the spring needs to be regulated so the drum will turn at a slow constant speed. This is accomplished by gears rotating a fan blade that creates drag. The fan blade is hard to see in the picture, it's directly below the spring cover. At the center bottom, there's a 45-degree wire with a pull lever on the back that stops the fan blade from turning, which stops the music. The tone prongs are the 18 vertical strips of metal that are held on with two screws. When the drum turns, the bumps on the drum hit the tone prongs, and the music plays. Each tone prong is one note of the song. A completed song turns the drum around once.

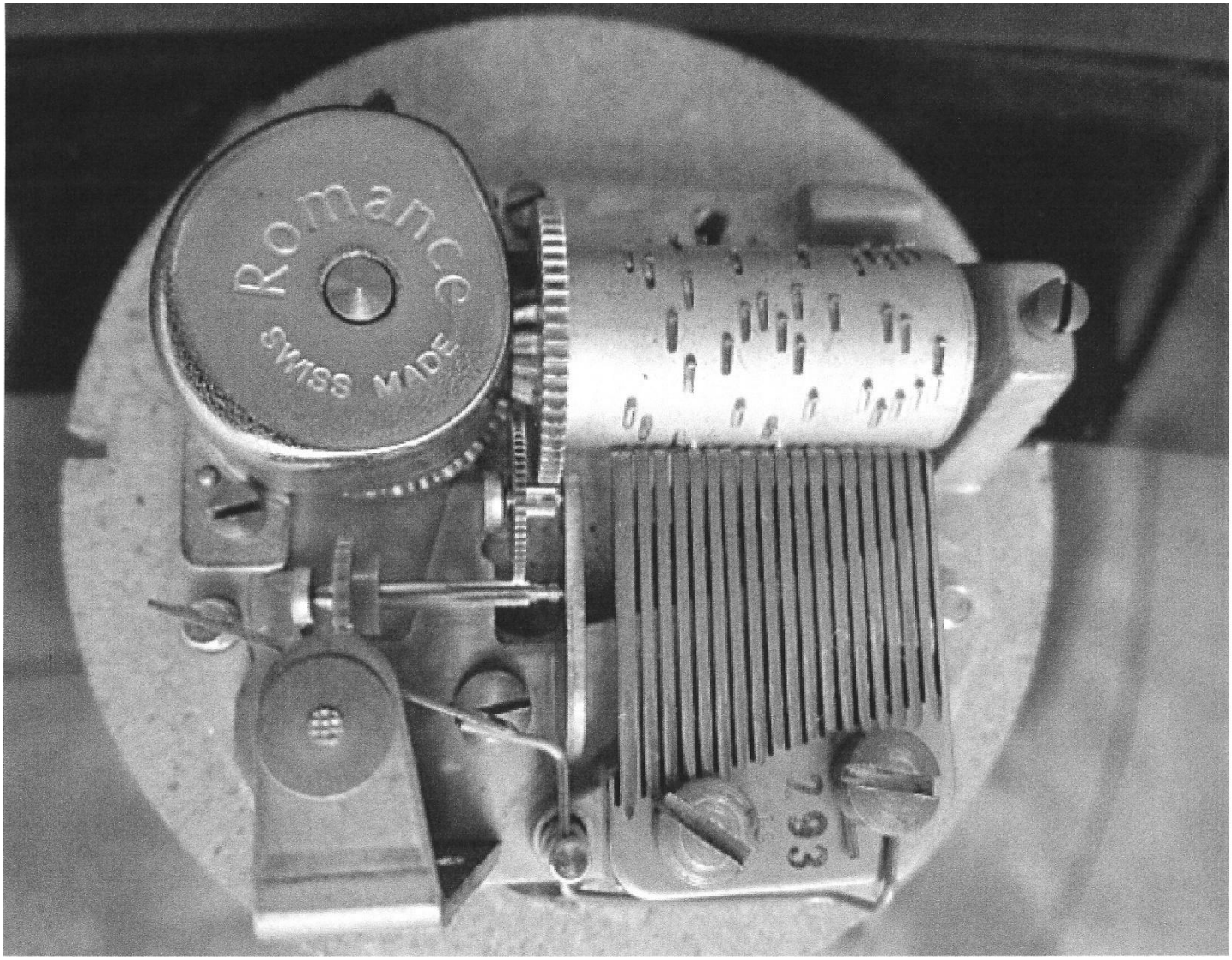



**Figure 53.** Music box picture

**Understanding Computers**

People have trouble understanding computers for two main reasons:

1. They don't know how to count in the binary number system.
2. They can't see electricity work because electricity is invisible.

It's easier to understand a concept when you're able to see it work. The mechanics of a music box is a good example. The design is very complex, yet people can understand it quickly. The electronic computer was based on the music box design. If you can understand a music box, you can understand computers.

The music box parts use different terminology than those of a computer, but they are the same:

| Music Box | Computer |
|---|---|
| a. Wind up key | Clock |
| b. Reduction gears that turn the drum | Counter |
| c. Drum with bumps | Memory with stored numbers |
| d. Bump that lines up with one tone prong | Data bit that is a one |
| e. A bump or no bump | "1" (one) or "0" (zero) |
| f. Total possible bumps at 1 tone time | 18 Data bits |
| g. One tone time | 1 address |
| h. Tone prong | CPU Timing and control signal |
| I. Bump designer | Computer programmer |
| j. Bump location at each tone time | Binary address number on a memory |

The prongs on a music box, supplies the notes for the music. On a guitar the strings that you pluck supplies the notes and the person who plucks the strings is a music programmer. If you placed the music box drum with its bumps next to the strings of a guitar, the guitar would play itself. Likewise, if you let the music box drum bumps hit the keys on a calculator it would essentially become a computer. This timing part of a computer is what I call the timer controller. When you understand the timer controller, the rest of the computer is easy.

So let's find a way to explain it. Consider a washing machine. It is a simple machine that doesn't have a lot of functions. Remember that the washing machine started out all mechanical, and it was only later that technology was added to it.

The following pages will compare four timer controllers.

1. Electro-mechanical washing machine timer controller that turns on a function (water, drain, agitator, or spin) and controls how long the function will stay on or off.

2. Electronic washing machine controller that turns on or off a function (water, drain, agitator, or spin), but does not control how long the functions will stay on or off.

3. Electronic washing machine timer controller that starts a function (water, drain, agitator, or spin), and controls how long it will stay on or off.

4. 6502 Computer timer controller. You will finally use the timer controller to control a computer, which is the whole mission of this chapter. The CPU timer controller outputs instruction timing signals, to control digital parts in the CPU. Those digital parts move numbers around in a certain order to complete an instruction.

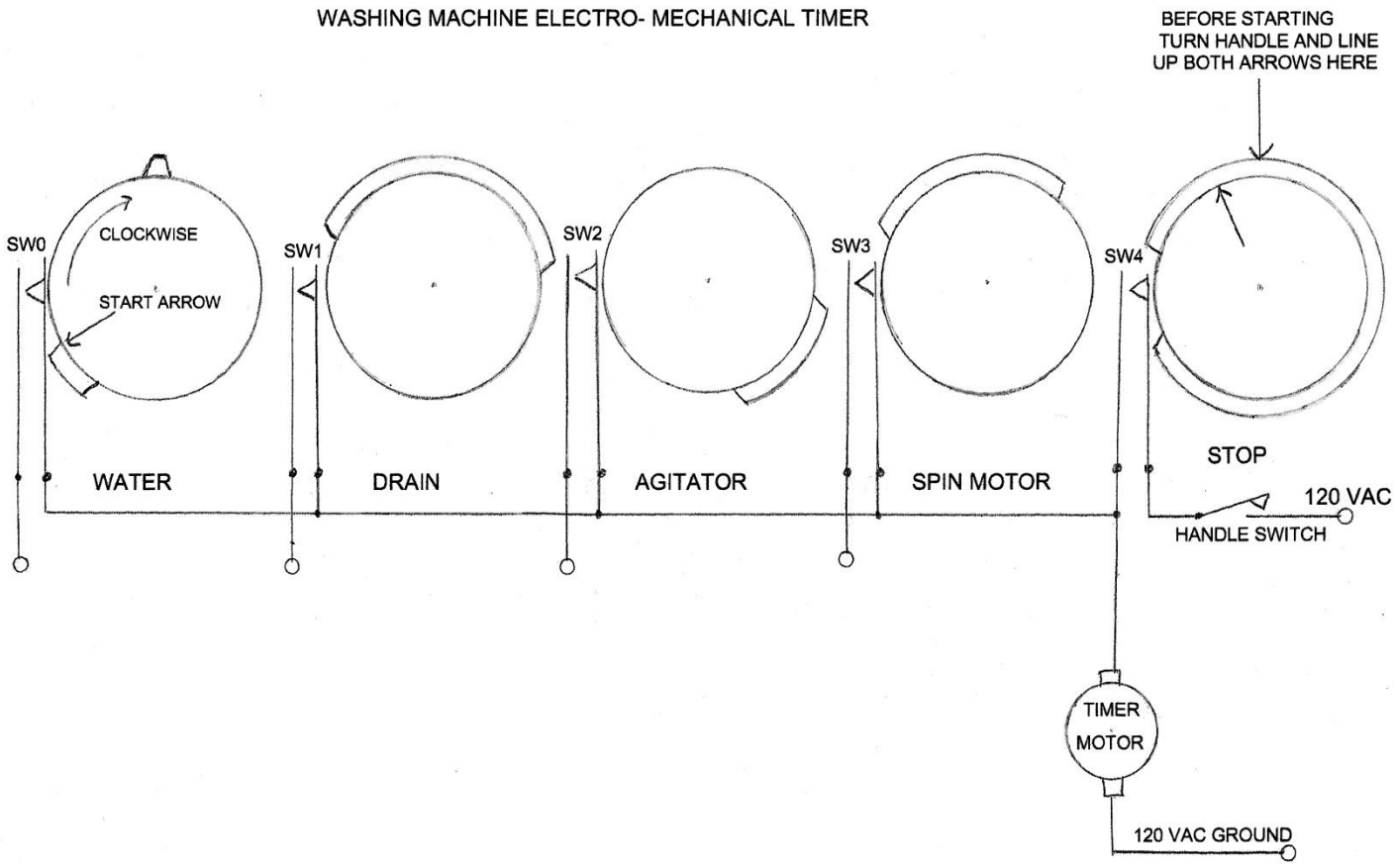



**Figure 54.** Washing machine electro-mechanical timer diagram

**Timing Control Circuit**

Our first timer controller controls an electro-mechanical washing machine. It is very similar to the music box. On the music box, the bumps on the drum hit tone prongs. On a washing machine timer controller, the bumps turn on switches.

The washing machine timer controller is an electro-mechanical CPU and is usually right behind the selector dial. The washing machine timer controller operates by a small motor, with reduction gears, slowly turning plastic wheels with bumps. Those bumps turn on switches. There is a wheel and switch for the water valve, a wheel and switch for the drain valve, a wheel and switch for the agitator motor, and a wheel and switch for the spin motor. And there's one more wheel and switch that turns the timer motor off. The location of the bumps on the plastic wheels will determine when the switch turns on, and how long the bump is, will determine how long the switch will stay on.

Things to remember:

a. Plastic wheels are all connected together

b. Wheels rotate about 350 degrees and stop. It takes about 20 minutes to rotate 350 degrees

c. A manual selector is connected to the plastic wheels. On some washing machines you can see the manual selector handle turn as washing progresses.

d. There are short bumps and long bumps. Long bumps make the switches stay on for a longer period of time.

e. The switches are two bendable flat strips of metal with silver contacts on the end. The bumps bend the metal and the contacts touch each other, which turns on a function (spin, or agitator, or drain, or water).

f. The stop switch contacts open up when the washing is done, which turns the timer motor off.

Sequence of operation:  Look at the electro-mechanical diagram in figure 54 and the time line diagram Figure 57.

1. Turn the selector pointer to start.
2. Pull handle, input power will go to all switches and the timer motor will turn on
3. Water turns on
4. Four minutes later- water turns off
5. ½ minute later- agitator starts
6. Six minutes later- agitator turns off
7. ½ minute later- drain turns on
8. ½ minute later- spin turns on.
9. 3 ½ minutes later - water turns on.
10. ½ minute later- water turns off.
11. 3 minutes later- drain and spin turns off.
12. ½ minute later- the timer motor stops and the clothes are done.

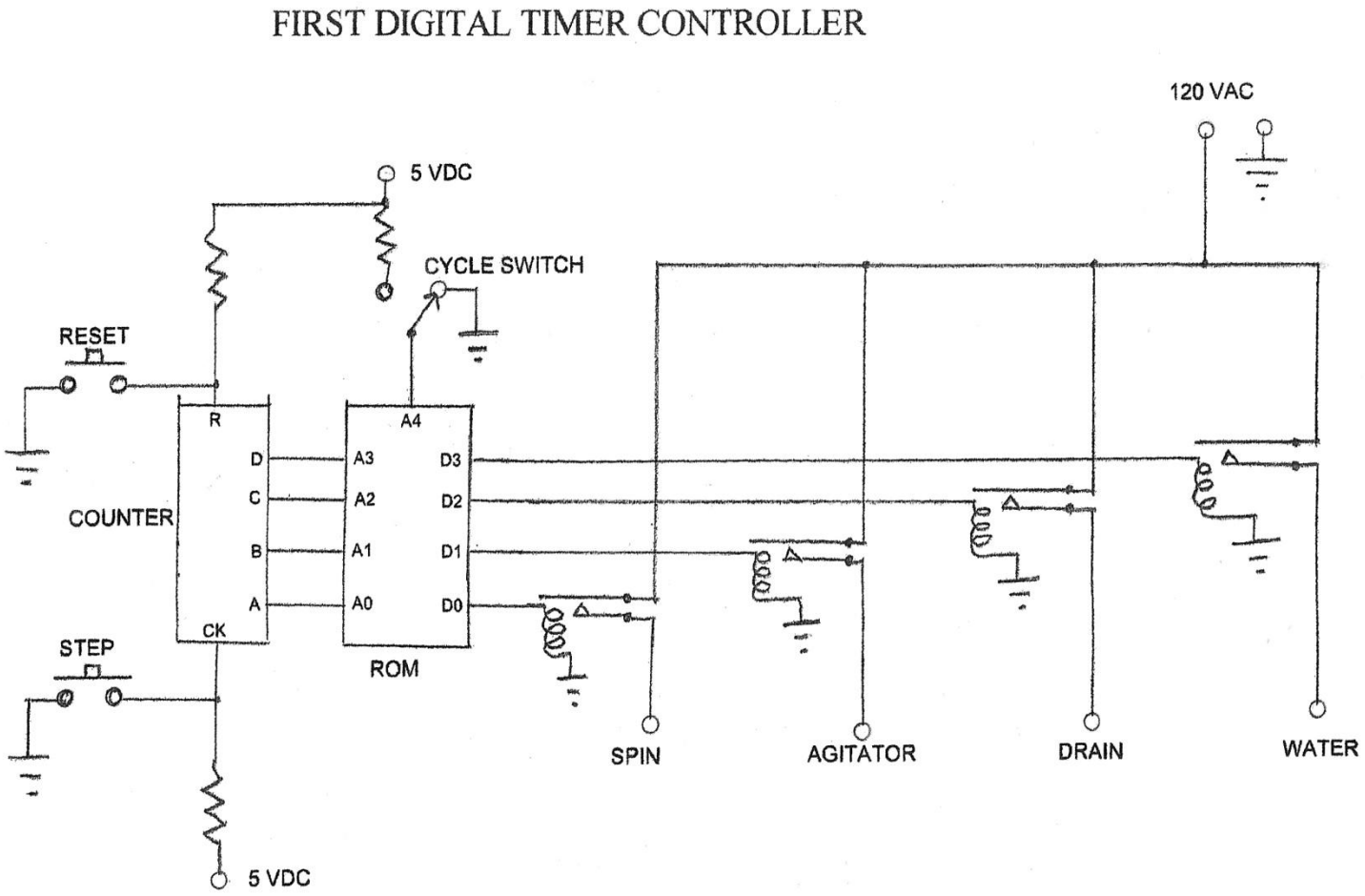



**Figure 55.** First digital timer controller

## FIRST DIGITAL TIMER CONTROLLER

The first digital timer controller circuit (Figure 55) turns on or off washing machine function relays. How long they stay on or off is not controlled. The circuit has a step switch that clocks a binary counter, which increments the addresses on a ROM. The ROM has 4 data bits that are connected to 4 relays that can turn on water, drain, agitator, or spin. The programming at each address of the Rom will decide which data bit will output a "1". That bit will turn on a function relay (water, drain, agitator, or spin). Up to 16 addresses on this ROM can be programmed to turn function relays on in a certain order. The order of the functions programmed into the 16 addresses of the ROM, will be the order that they come out.

**Relays**

Relays are used to turn on high current devices and will be used to turn on our washing machine motors and valves. The power that a “1” has out of a ROM data bit is not enough to activate a relay. A transistor driver circuit would be needed to increase the power on each data bit. I left those circuits out so the timer controller would be less complicated.

Each list number below has the functions that will be stored on each ROM address. After the Rom is programmed, each time you press the step switch you will do what is next on this list.

1. Turn on the water to fill the tub
2. Agitate
3. Drain water
4. With drain on, spin the clothes to remove the water.
5. While spinning with the drain valve on, add water
6. Turn water off and Spin with the drain valve on.
7. Turn everything off

Spin = Data bit D0 of ROM
Agitator = Data bit D1 of ROM
Drain = Data bit D2 of ROM
Water = Data bit D3 of ROM

Binary "1" on one the bits listed above will turn on the function and binary "0" will turn off the function. The bits with a "1", plus the bits with a "0" have a binary number value. Those binary value numbers are used to program the ROM.

To start the washing machine, press reset. Then increment the counter addresses on the ROM by pressing the step (clock) switch. Every time you press step, allow enough time for each washing machine task to complete, before pressing again. Program the ROM with these numbers. Note: Each address of the Rom can select more than one relay function to turn on.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | A | | | |
| | | | | | | | G | | | |
| | | | | | | | I | | | |
| | | | | | W | D | T | | | |
| | | | | | A | R | A | S | | |
| | | | | | T | A | T | P | | |
| | | | | | E | I | O | I | | |
| | | | | | R | N | R | N | | |
| ROM addresses | | | | | | | | | | |
| Hex | A3 | A2 | A1 | A0 | | Hex | D3 | D2 | D1 | D0---Data bits with a "1" |
| 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 reset |
| 1 | 0 | 0 | 0 | 1 | | 8 | 1 | 0 | 0 | 0 water on |
| 2 | 0 | 0 | 1 | 0 | | 2 | 0 | 0 | 1 | 0 agitator on |
| 3 | 0 | 0 | 1 | 1 | | 4 | 0 | 1 | 0 | 0 drain on |
| 4 | 0 | 1 | 0 | 0 | | 5 | 0 | 1 | 0 | 1 drain on spin on |
| 5 | 0 | 1 | 0 | 1 | | D | 1 | 1 | 0 | 1 water on drain on spin on |
| 6 | 0 | 1 | 1 | 0 | | 5 | 0 | 1 | 0 | 1 drain on spin on |
| 7 | 0 | 1 | 1 | 1 | | 0 | 0 | 0 | 0 | 0 all off |

Looking at the programming list, you will see hexadecimal numbers and binary numbers. The address numbers for the ROM are on the left, and the data numbers to be stored are on the right. When operating the circuit, the first row of numbers is after you press reset. The second row is after pressing the step switch. It keeps going down the list every time you press step. Up to 16 addresses can be used with this timer circuit. If you wanted two totally different washing cycles, just double the size of memory, add the new programming, and switch between each half of the memory. On diagram figure 55, I added that switch and a ROM with 32 addresses instead of 16.

Because there is no provision for on or off function time, manually stepping the ROM functions on this washing machine circuit is the only way the washing cycle progresses. This circuit would be good for a CPU timer controller but not for a washing machine controller. The next timer controller will take this circuit, add a clock, another counter, and a larger ROM so the washing machine can count time and be automatic.

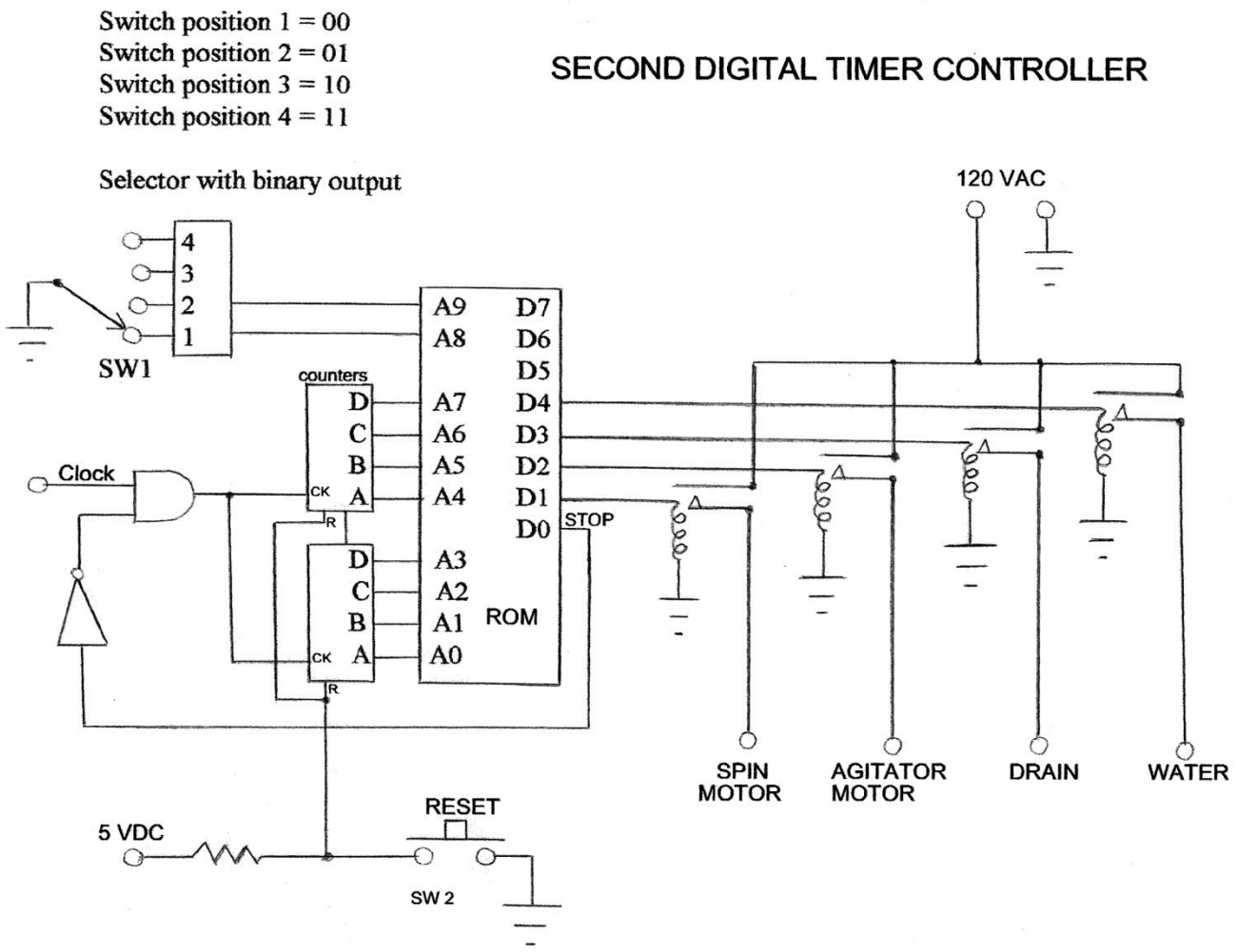



**Figure 56.** Second digital timer controller

Now let's change the electro-mechanical timer controller to the second complete electronic one. Obviously, our digital timer needs something that can count time, not just select a function. A clock that changes every 4.5 seconds and a larger binary counter counting time are needed. The bumps on each of the plastic wheels need to be remembered. Each plastic wheel is for each data bit of the ROM. A bump on a wheel equals a "1" and no bump equals a "0". The duration of a bump decides how for many multiple addresses a "1" is stored, which determines how long the function will stay on. For example, by choosing a starting address, choosing a data bit to store a "1", and deciding for how many addresses that bit will be stored, determines when a washing machine function will turn on, which function will turn on and how long it will stay on. The mechanical washing machine with a regular cycle had five plastic wheels. If you wanted to add a permanent press cycle, you would need to add four more plastic wheels which would be a total of 9 wheels.

With an electronic timer, you would just double the size of the memory, add the new programming and switch between each half of memory. This circuit diagram quadrupled the size of memory and uses a switch to select 4 different cycles.

Let's explain the circuit. Look at the time line diagram (Figure 57) and the second digital timer controller diagram (Figure 56), but remember that the counts on the time line are the addresses numbers on the Rom and the addresses numbers are recorded in decimal.

Note: Data Bits that Turn on functions = "1". Data bits that turn off functions = "0".

The washing machine will start when you press reset.

1. At 1 count D4 turns water on.
2. At 52 counts D4 turns water off
3. At 57 counts D2 turns on agitator.
4. At 137 counts D2 turns off agitator.
5. At 143 counts D3 turns on drain.
6. At 156 counts D1 turns on spin.
7. At 200 counts D4 turns on water.
8. At 208 counts D4 turns off water.
9. At 247 counts D1 turns off spin and D3 turns off drain
10. At 251 counts D0 turns off the clock. The timer stops and the clothes are done.

Look at the time line diagram (Figure 57). It has the same timing as the mechanical timer.

This is how the timer controller operates:

1. When the reset button is pressed, the binary counters reset the addresses on the ROM to zero and data bit Do from the Rom will start the washing machine by enabling the "And" gate to input the clock to the counters.

2. Every 4.5 seconds the clock changes and the timer counters increment, which increments the addresses on the ROM.

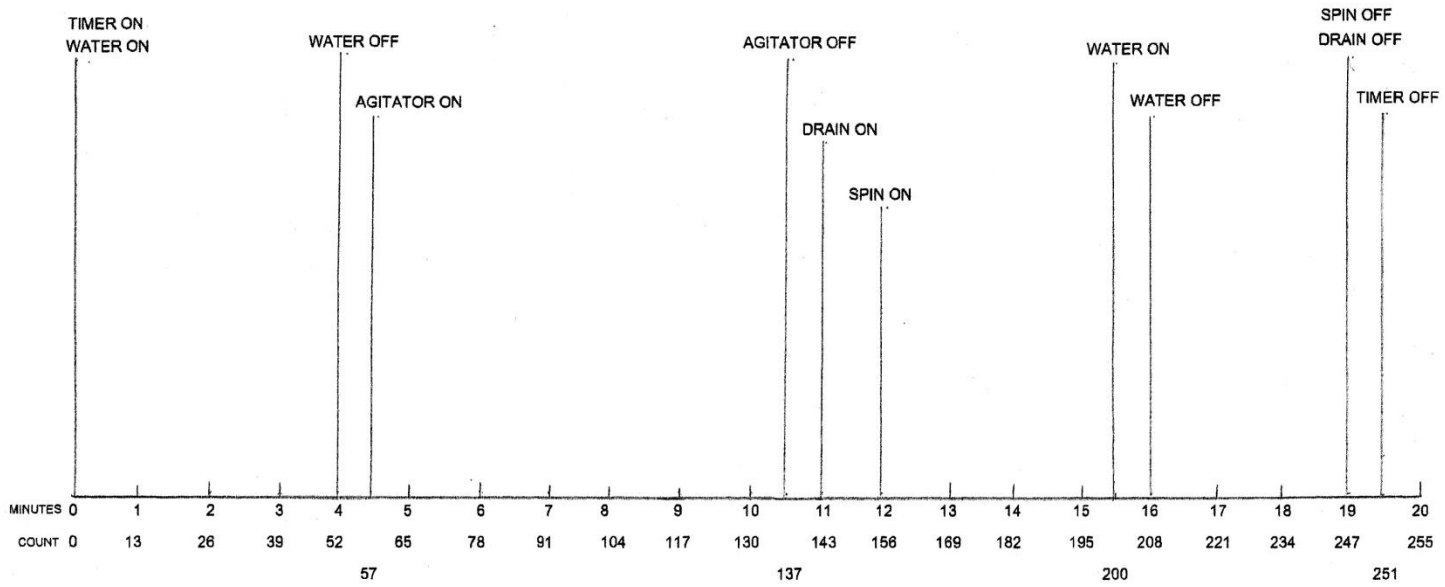
3. At each address of the ROM, a stored data bit with a "1" will turn on a relay function (water, drain, agitator, or spin) and a "0" will turn off a relay function.

To program the ROM, we will choose a starting address and an ending address for each function. This will select the start time and end time for each function. Also at those addresses, we will choose which data bits need a "1". This will select which relay circuits need to turn on. Remember that many multiple addresses with a one ("1") will turn on a function for a long time, and many multiple addresses with zero ("0") will turn off a function for a long time. A single address with a "1" on the stop bit will make everything stop, by the “And” gate turning off the clock.

Programming - Please note that address numbers are in decimal and data numbers are in binary.

| | | WATER D4 | DRAIN D3 | AGITATOR D2 | SPIN D1 | STOP D0 |
|---|---|---|---|---|---|---|
| Address | 0 | 0 | 0 | 0 | 0 | 0 |
| Address | 1 through 51 | 1 | 0 | 0 | 0 | 0 |
| Address | 52 through 56 | 0 | 0 | 0 | 0 | 0 |
| Address | 57 through 136 | 0 | 0 | 1 | 0 | 0 |
| Address | 137 through 142 | 0 | 0 | 0 | 0 | 0 |
| Address | 143 through 155 | 0 | 1 | 0 | 0 | 0 |
| Address | 156 through 199 | 0 | 1 | 0 | 1 | 0 |
| Address | 200 through 207 | 1 | 1 | 0 | 1 | 0 |
| Address | 208 through 246 | 0 | 1 | 0 | 1 | 0 |
| Address | 247 through 250 | 0 | 0 | 0 | 0 | 0 |
| Address | 251 | 0 | 0 | 0 | 0 | 1 |

Note: The clock circuitry that steps the counters every 4.5 seconds has not been shown.




TIME LINE FOR ELECTRO- MECHANICAL AND DIGITAL WASHING MACHINE

**Figure 57.** Time line diagram

**6502 Timer Controller**

We are going to add address bits and data bits to the first digital timer controller and use it for the 6502 timer controller. A fast running clock will also be added. To be able to execute all the instruction in the 6502 Micro-processor the CPU needs 64 timing signals to control the digital parts inside the CPU. A timing signal is a binary digit that goes from "0" to "1". The 64 timing signals that are needed for the CPU come from the 64 data bits on the timer controller (I) ROM. Each data bit on the I ROM is a timing signal. 16 timing signals for each instruction are programmed into the I ROM. Each timing signal comes from one address in the I ROM selecting one of the 64 data bits to output a "1". The instruction register selects the starting address in the I ROM for the instruction and the clock increments the addresses on the I ROM 16 times to get the 16 timing signals. The 16 timing signals control digital parts in the CPU to move numbers around to complete the instruction. Some of the 64 timing signal names are: input X register number, increment the program counter number, output "A" register number, output data latch number, input Y register number, and input the instruction register number. If the instruction does not require all 16 timing signals to complete its instruction, a bit from the I ROM will reset the counter when the instruction is complete and the next instruction will start. Those 64 timing signals are represented on the block diagram (Figure 2) by 8 output lines from the CPU timing control. Note: the 6502 timer controller (Figure 58) only shows 34 of the 64 timing signals. To see all 64 timing signals, look at the "real 6502 diagram in detail" on Figure 60 at random control logic outputs.

Every instruction is allocated 16 timer addresses (A0, A1, A2, and A3), with up to 16 timing signals, for each instruction. Each one of the 250 instructions (56 instructions plus variations) selecting 16 address, takes 4000 addresses in the I ROM. This is accomplished by the instruction register using addresses A4, A5, A6, A7, A8, A9, A10, and A11 to select each of those 250 instructions.

Status register – On some instructions, if previous data has changed the timing sequence for that instruction needs to change. An example of that would be the "Branch if equal to zero" instruction. This instruction starts by sending I ROM data select signals to the 16 to 1 data selector. 0001 on data selector inputs "D, C, B, A" will select the zero part of the status register. The "0" or "1" that is stored in that register is sent through the data selector to I ROM address 12. If the zero status is a zero flag, a "1" will be sent to Address 12 and the I ROM will select the timing signals needed for the branch instruction to branch to a new location. If the zero status is not a zero flag, then a "0" is sent to address 12 and the instruction will end and the next instruction will start. The zero flag is in the status register, which is in the CPU.

To understand the 6502 I ROM circuit more completely, slowly read Chapter 8.

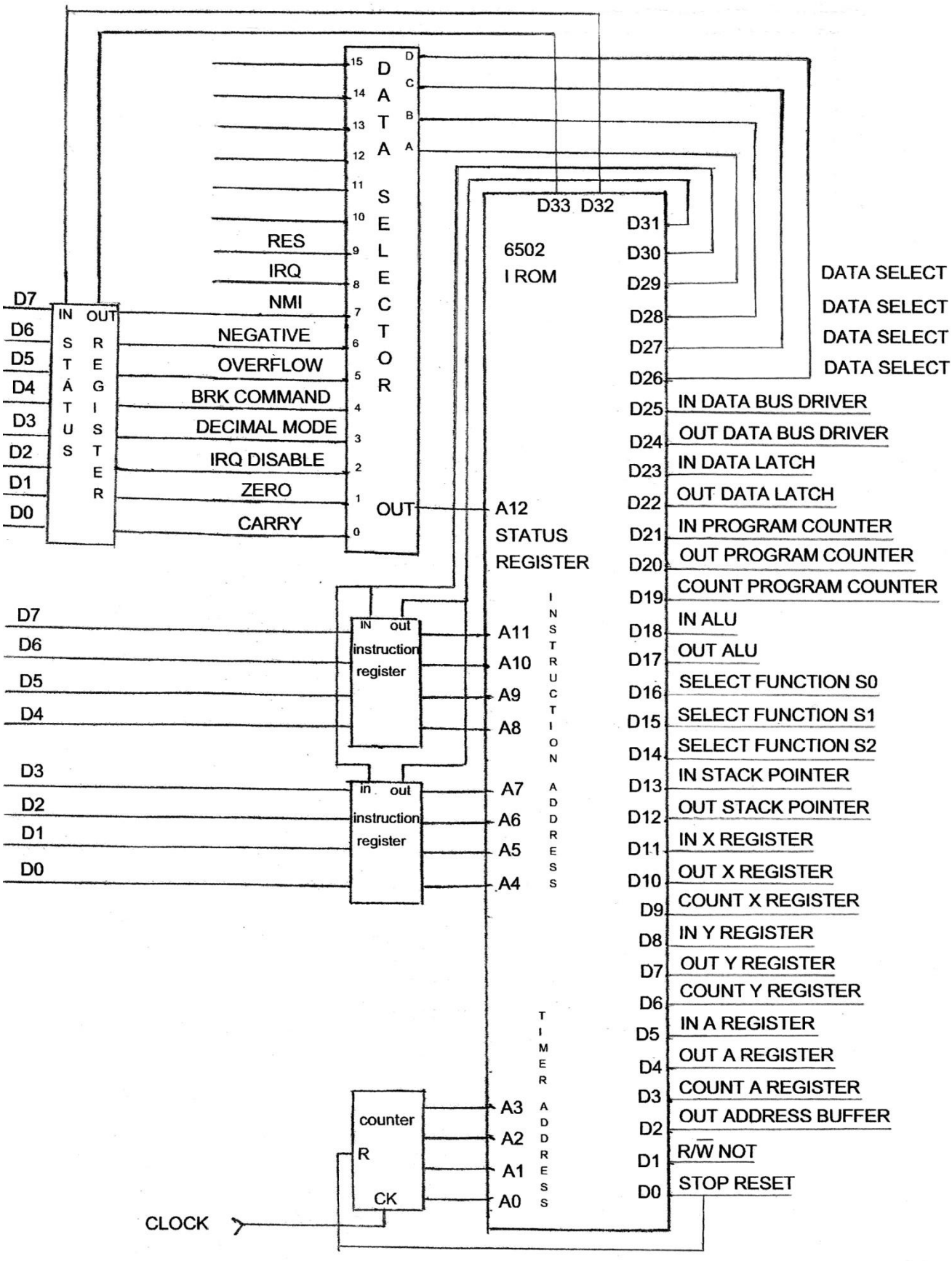



**Figure 58.** 6502 Timer Controller

**How a digital music box operates.**

We're going to change the mechanical music box to an electronic music box by using the first digital timer controller.

Music box comparisons

A. Drum = ROM.
B. Turning of the drum = Binary Counter.
C. Bumps on the drum = Data bits with a "1" that are programmed into the timing Rom.

If the mechanical music box could be stepped, every time you press the step button a tone would be heard. You would see the drum turn a small amount every time you press step. The music box has one song. That song has about 90 notes, or 90 steps. Each step has a possible of 18 different tone prongs. The right bump hitting the right tone prong in the right order 90 times gives you one whole song. If you step it fast, you will hear the song.

Look at the first digital timer controller diagram (Figure 55). If we added more addresses (now 16) and more data bits (now 4 data bits and 4 relays) you would still have the same circuit, but with more capacity. Change the number of ROM addresses from 16 to 90, the number of data bits from 4 to 18, and the number of relays from 4 to 18. Now instead of a relay pushing on a switch, have the relays hit each of the 18 tone prongs. Guess what you get - an electronic music box. If the timing ROM was programmed for music (selecting which data bit has a "1" at each ROM address) and a clock signal was used instead of a step switch, you would have a music box. The mechanical and the electronic music box would sound the same. If you doubled the size of the memory, added more programming and had a switch that was connected to the highest address line, you could select two different songs. You could use a much larger memory with more address lines and more switches and choose hundreds of songs.

**Basic CPU operation**

The first thing to notice about the 6502 is all data to or from the data bus is 8 bits. The data bus can input or output. The address bus is a 16 bit CPU output bus connected to the motherboard and is used to address motherboard circuits. It can address a pixel location on the display circuit. It can address an address location in a ROM or RAM. It can select and control a sound board or anything plugged into the motherboard. Most addresses from the CPU come from the 16-bit program counter. The program counter is used to keep track of program instructions. It does this by incrementing the addresses that go to the motherboard program ROM, thus selecting the next program code. The upper byte of the program codes go to the instruction register in the CPU for timing and control. The timing ROM then counts out instruction timing signals that control registers in the CPU to move numbers around to complete an instruction. During the execution of some instructions, the next two addresses in Program ROM after the "op code" is a new address. When that happens, the program counter addresses the next two addresses in Program ROM and sends that data to the CPU input/output data latch. The data latch then outputs the new address to the address bus. Motherboard circuits need to be turned on one at a time and accessed. The CPU puts identifying circuit numbers for those circuits on the upper addresses of the address bus. When the decoder detects a circuit number, it turns on the chosen motherboard circuit.

To figure out what's happening with all the instructions in the CPU, look at the description for each instruction in a programming book. It will tell you which registers and digital parts are used and in what order they are used.

One more thing: My timing ROM (called I ROM) circuit uses micro-code. That is not exactly the same circuit that is used on the real 6502, which has a ROM and random logic control for the timing signals. Random logic circuits use ands, ors, and inverter gates to make logic, which is designed using Boolean Algebra. Micro-code is easier to understand than Boolean Algebra. I hope you can see that the designer who made up the instructions for the 6502 also made the programming code for the ROM. If you look at all of the instructions, you will notice that they only do simple things. To do something complicated, you need many instructions. That is the main reason why computer programs keep getting larger and why your home computer keeps getting slower.

Figures 59 and 60 are the real diagrams for the 6502. Figure 59 block diagram is almost the same as our block diagram. The second diagram (Figure 60) has all the details. I threw in those diagrams to show you what we're doing is real.

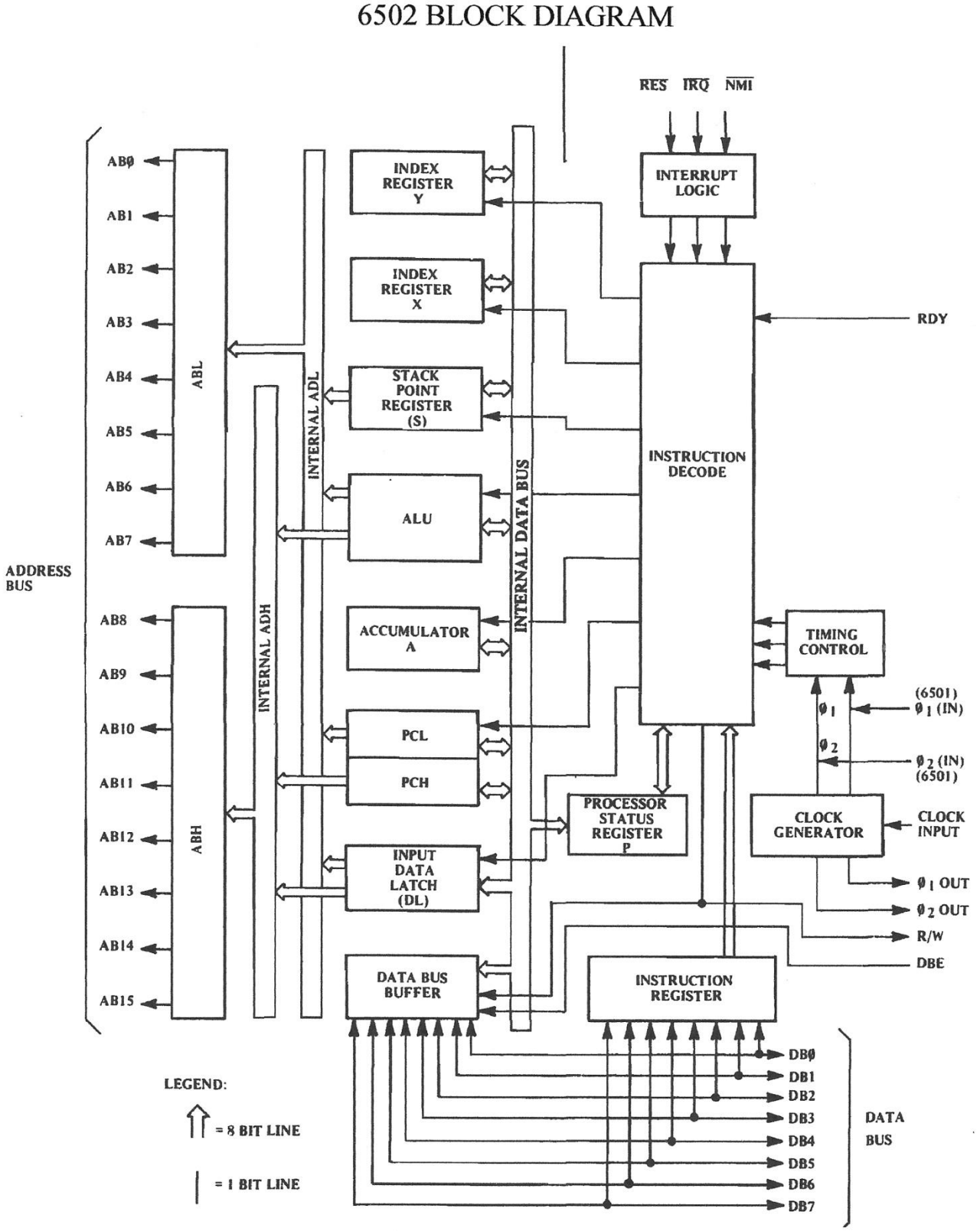



**Figure 59.** Real 6502 block diagram

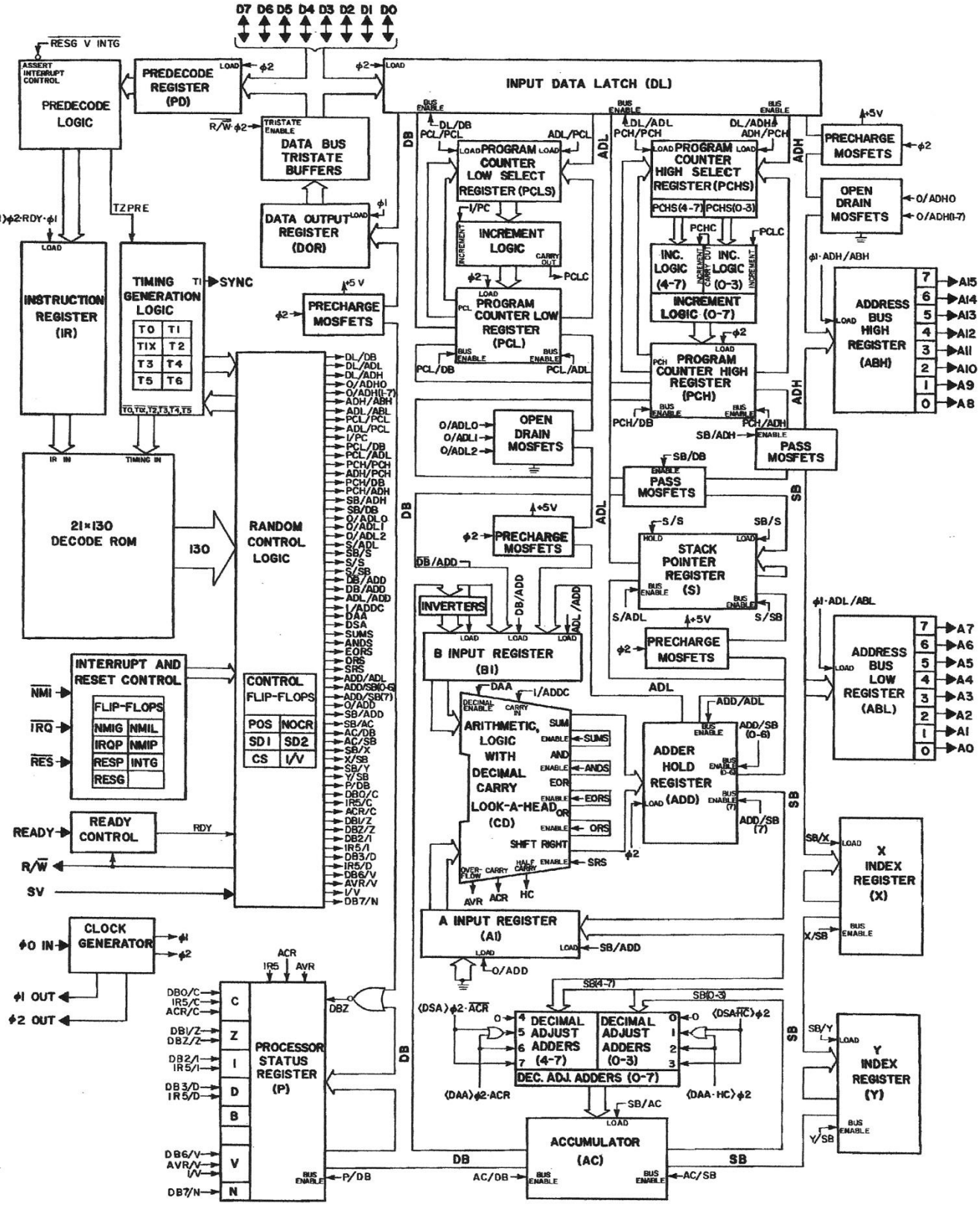



**Figure 60.** Real 6502 block diagram in detail

# Chapter 8

## One Instruction

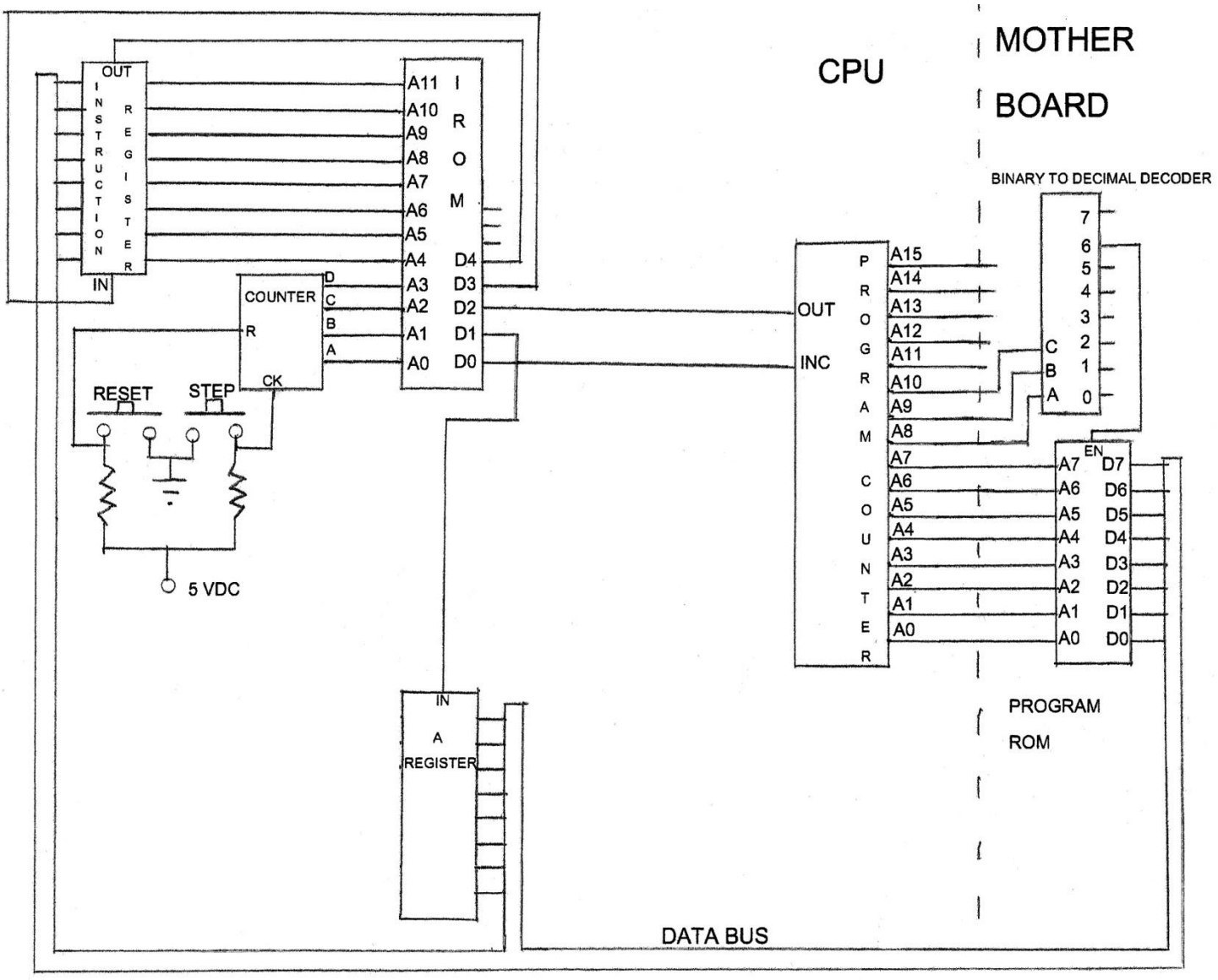



**Figure 61**. One instruction diagram

The purpose of this exercise is to show the parts of a 6502 computer system that are only used with the load "A" register instruction. Before explaining this circuit in detail, I want to give you a general idea of what's happening. There are five data bits out of the I Rom. Those bits are the step instruction signals for the load "A" register instruction. This timing circuit outputs these step instruction signals in this order:

1. D2 outputs the program counter number to the address bus.
2. D3 takes the instruction number from the data bus and puts it into the instruction register.
3. D4 takes the instruction number out of the instruction register and sends it to the addresses of the I ROM.
4. D0 increments the Programmer counter.
5. D2 outputs the program counter number to the address bus.
6. D1 Takes the data number from the data bus and puts it into the "A" register.

The first digital timer controller data outputs were for spin, agitator, drain, and water. The timer outputs now are the five I just listed. The "one instruction" circuit will help you understand the 6502 I Rom circuit. Now the details:

The "A" register is a small memory in the CPU that stores a two-digit hex number. Our instruction is load "A" register. The machine code is A901. It's stored in Program ROM on two consecutive addresses. A9 tells the CPU it's a load "A" register instruction with its data from the next address in Program ROM.

Every time you power up your computer, the program counter is loaded with a starting address, from two of the last four addresses in Program ROM. The starting address is always 0600 in our book and it will be the starting program address for this code. The explanation for our code is:

1. Go to the first address in Program ROM and send its data (A9) to the CPU instruction register.

2. The instruction register outputs "A9" (OP code number) to the addresses of the I ROM. The I ROM data bits then supply the timing signals to go to the second address in Program ROM, take out the data (01) and put it into the A register.

That is the whole process, except that we're going to step the I ROM addresses to read out the timing signals. Before executing the instruction, all data is in hex and the instruction register has 00 (hex) in it.

1. 0600 is in program counter

2. Reset the counter. Counter goes to 00, I ROM address = 000-- Data Bits D0, D1, D2, D3, and D4 = "0"

3. Step counter. I ROM address = 001-- Data = 04--Bit D2 = "1".
   D2 outputs program counter number 0600 to the address bus. The decoder reads the upper 2 digits from the address bus (06) and enables the Program Rom. The Program ROM address is 00 and the data from the Rom (A9) outputs to the data bus. Note: A9 is the OP code.

4. Step counter. I ROM address = 002--Data = 08--Bit D3 = "1"
   D3 loads the instruction register with data (A9) from the data bus.

5. Step counter. I ROM address = 003--Data = 10--Bit D4 = "1"
   D4 outputs A9 from the instruction register to I ROM addresses.
   I ROMs new address = A93--Data = 01--Bit D0 = "1"
   D0 increments program counter to 0601

6. Step counter. I ROM address = A94-- Data = 04--Bit D2 = "1"
   D2 outputs 0601 from the program counter to the address bus. The decoder reads the upper 2 digits from the address bus (06) and enables the Program ROM. The Program Rom address is 01 and the data from the Rom (01) outputs to the data bus.

7. Step counter. I ROM address = A95--Data = 02--Bit D1 = "1"
   D1 takes 01from the data bus and loads it into the "A" register.

The instruction has now been completed.

I ROM programming code for Load A register

| ADDRESS | | | | | | | | | | | | | DATA | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hex | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | Hex | D4 | D3 | D2 | D1 | D0 |
| 000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | 0 | 0 | 0 | 0 | 0 |
| 001 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 04 | 0 | 0 | 1 | 0 | 0 |
| 002 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 08 | 0 | 1 | 0 | 0 | 0 |
| 003 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 10 | 1 | 0 | 0 | 0 | 0 |
| A93 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 01 | 0 | 0 | 0 | 0 | 1 |
| A94 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 04 | 0 | 0 | 1 | 0 | 0 |
| A95 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 02 | 0 | 0 | 0 | 1 | 0 |

Data bit functions.

D0 = increment program counter
D1 = input "A" register
D2 = output program counter
D3 = input instruction register
D4 = output instruction register

# Chapter 9

## 6502 Simulator Codes

These are the things we are going to do with the 6502 assembler/simulator:

1. Show you all the controls and what they do.

2. Put a pixel on the screen by selecting a video memory location and telling the computer to write a pixel there.

3. Write a video memory location into RAM memory. Then tell the computer to get that location that you stored and use it for a pixel location on the display.

4. Key routine. The key routine inputs a keystroke from the web simulator and sends it out as a label. The label can be put anywhere in our code, and is used for jumping to another routine. The key routine also repeats a keystroke if a key stays pressed.

5. Type in the code for increment memory and watch a line draw to the right.

6. Type in the code for decrement memory and watch a line draw to the left. Add an additional decrementing code or two and watch missing spaces when the line is drawn to the left.

7. Type in the code for moving a pixel. Watch it work and understand how it works.

8. Type in the code for moving a line upward.

## 1. Controls

Go to the simulator web site by typing search (Easy 6502 by Skilldrick). Choose one of the small black screens with a big square box next to it. The typed-in letters and numbers in the box are 6502 assembly computer code. Put your cursor at the last code and back space to erase all of the code. Choose a test routine to type in. After typing in each line of code, hit "enter". After typing all the codes, hit "assemble" so the assembly codes can be changed into machine codes. Then hit "run" to execute the test routine.

After you hit "assemble", you can choose to hit "debugger" and step the program at default address 0600 or choose a program address to go to by clicking the "jump to" box or just hit "run" which is the normal procedure. If you hit "debugger" and use step, the right side of the screen will monitor registers A, X, Y, the stack pointer, the program counter, and the status register. Below the instruction area is a memory dump. To turn it on, hit the monitor check box. The starting address and length can also be selected. The memory dump is used to see what is in simulated memory. The display screen is a 32x32 pixel screen. Moving pixels up or down this screen requires adding and subtracting instructions. 6502 8-bit adding and subtracting instructions are not capable of writing to the whole screen, so extra instructions are needed to link two registers worth of data together for 16 bits. The pixels will then move up or down past each ¼ section of the screen. I only use 16 bit addition and subtraction, on the completed game at the end of the book.

## 2. Store Direct

Erase all instructions and type in this code

LDA #$01
STA $0200

Hit "assemble". That will change the assembly codes into machine codes.

Hit "disassemble" and a window will pop up. It will show the machine codes for LDA #$01 and STA $0200. Listed below is each program address and each machine code byte.

| Address | Data | |
|---|---|---|
| 0600 | A9 | OP code - Load "A" Register data from the next memory location |
| 0601 | 01 | Data to be put into the "A" Register |
| 0602 | 8D | Op code - Store "A" register to the location in the next 2 bytes |
| 0603 | 00 | Memory location- lower byte |
| 0604 | 02 | Memory location- higher byte |

Note: The load "A" instruction (A901) is the complete machine code, with data (01) included in the instruction. And the Store "A" register instruction (8D 00 02) is the complete machine code, with the memory location (0200) included in the instruction.

If you hit "run", both instructions will execute and the routine will complete. Instead, hit the debugger check box and hit "step" instead of run. Step executes one instruction every time you hit it. Stepping a real 6502, steps only part of one instruction.

Before stepping, look at the debugger indicators and you will see A = 00 ("A" Register) and PC= 0600 (Program counter)

Hit debugger and Step once- A= 01 PC=0602- You just did a Load "A" Register instruction.
Hit "step" again - A= 01 PC =0605- You just did a Store "A" Register instruction.

If you look at the screen, a small white pixel will be in the upper left corner.

Change the memory location from STA $0200 to STA $05FF.

Hit "assemble".

Hit "debugger" and Step once- A=$01 PC=0602
Hit "step" again- A=$01 PC=0605- There will be a pixel at the bottom right corner.

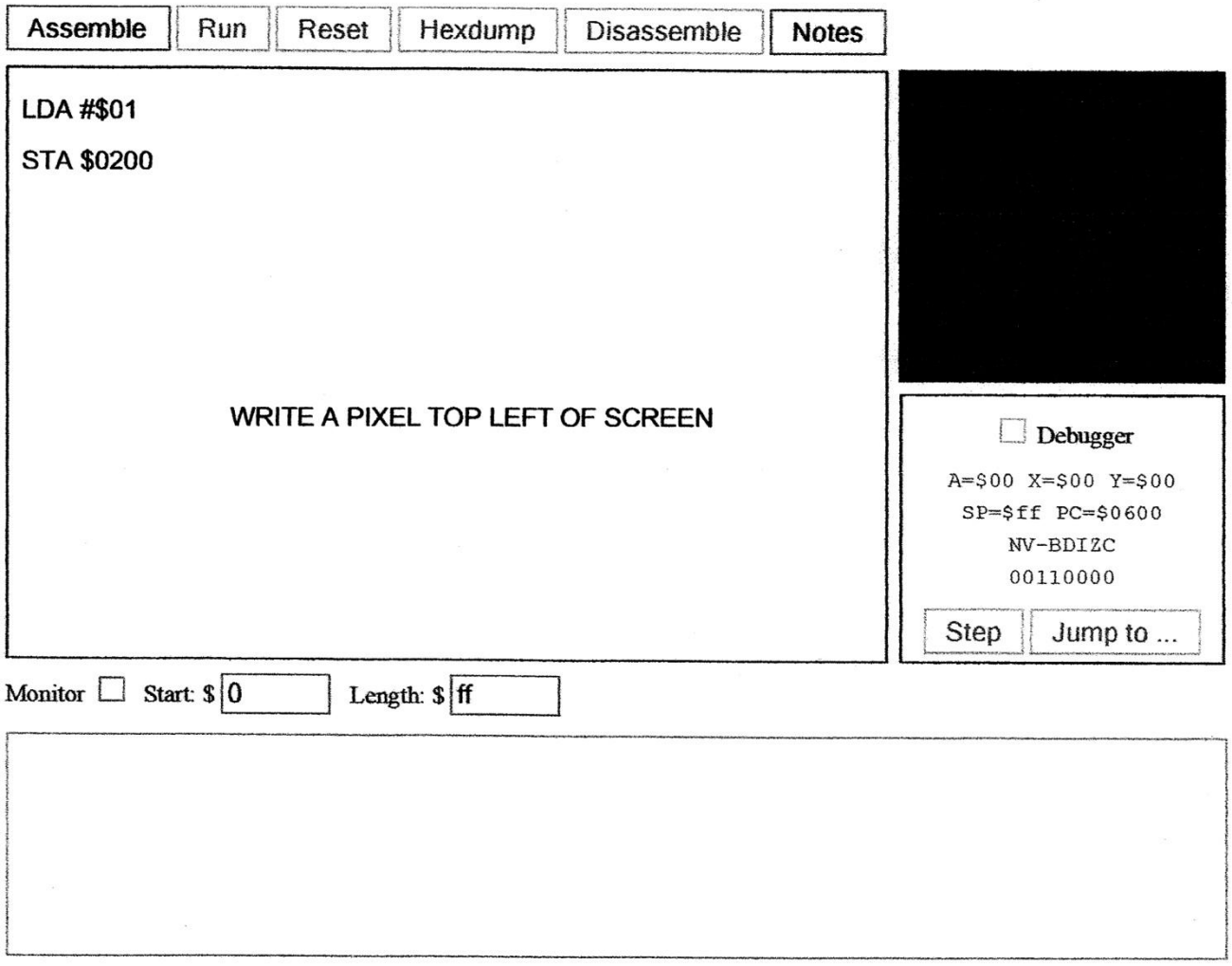



**Figure 62.** Write a pixel top left code diagram

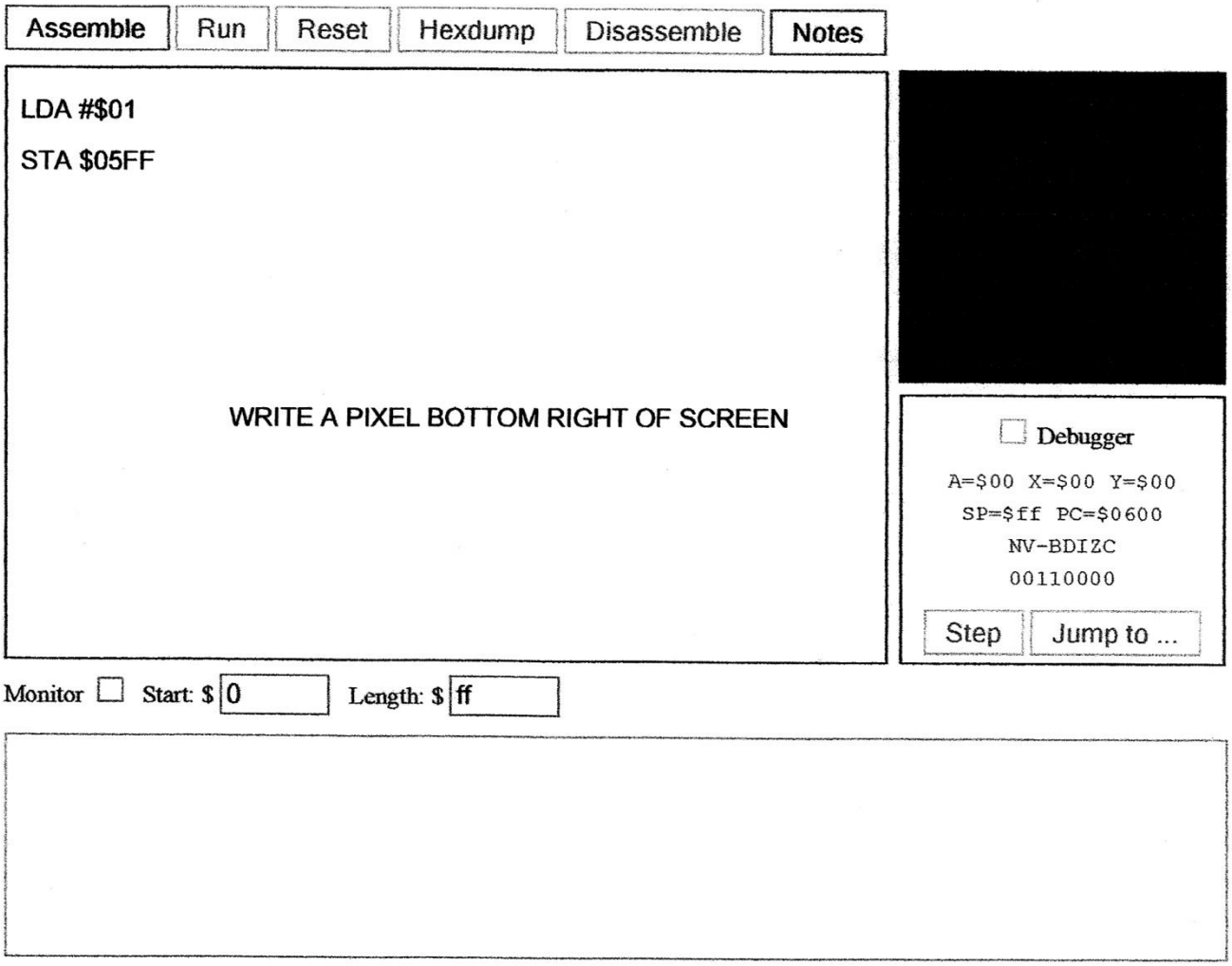



**Figure 63.** Write a pixel bottom right code diagram

## 3. Store Indirect

With the proceeding store direct instructions, the video location was attached to the store instruction code. We are now we are going to use a store indirect instruction. Its video location will be stored in two addresses of the RAM. The reasoning behind this is, location data in a RAM can be changed and can be used anywhere in code. It's called a variable. The lower 2 digits of the video location number (00) will be put in RAM address 40, and the upper 2 digits (02) will be put in RAM address 41. Put the stored data together (0200) and use that number for the display pixel location. The load "A" instruction loads 01 into the "A" register. Then the Store "A" indirect instruction takes 01 from the "A" register, along with video location 0200 from RAM addresses 41 and 40, and sends them to the display to write a pixel.

Note: The store "A" indirect instruction {STA ($40),y} only comes with Y register offset. To disable the offset, 00 was loaded into the Y register by adding instruction "LDY #$00" to the routine below.

Now let's write the routine:

```
LDA #$00
STA $40
LDA #$02
STA $41
LDY #$00 to disable offset
LDA #$01
STA ($40),y
```

Run or step this program. You will get a pixel in the upper left part of the display. To change the location of this pixel, write different location data into RAM addresses 40 and 41. That would be the first instruction and the third instruction. Try it!

Note: The first five instructions are called initialization.

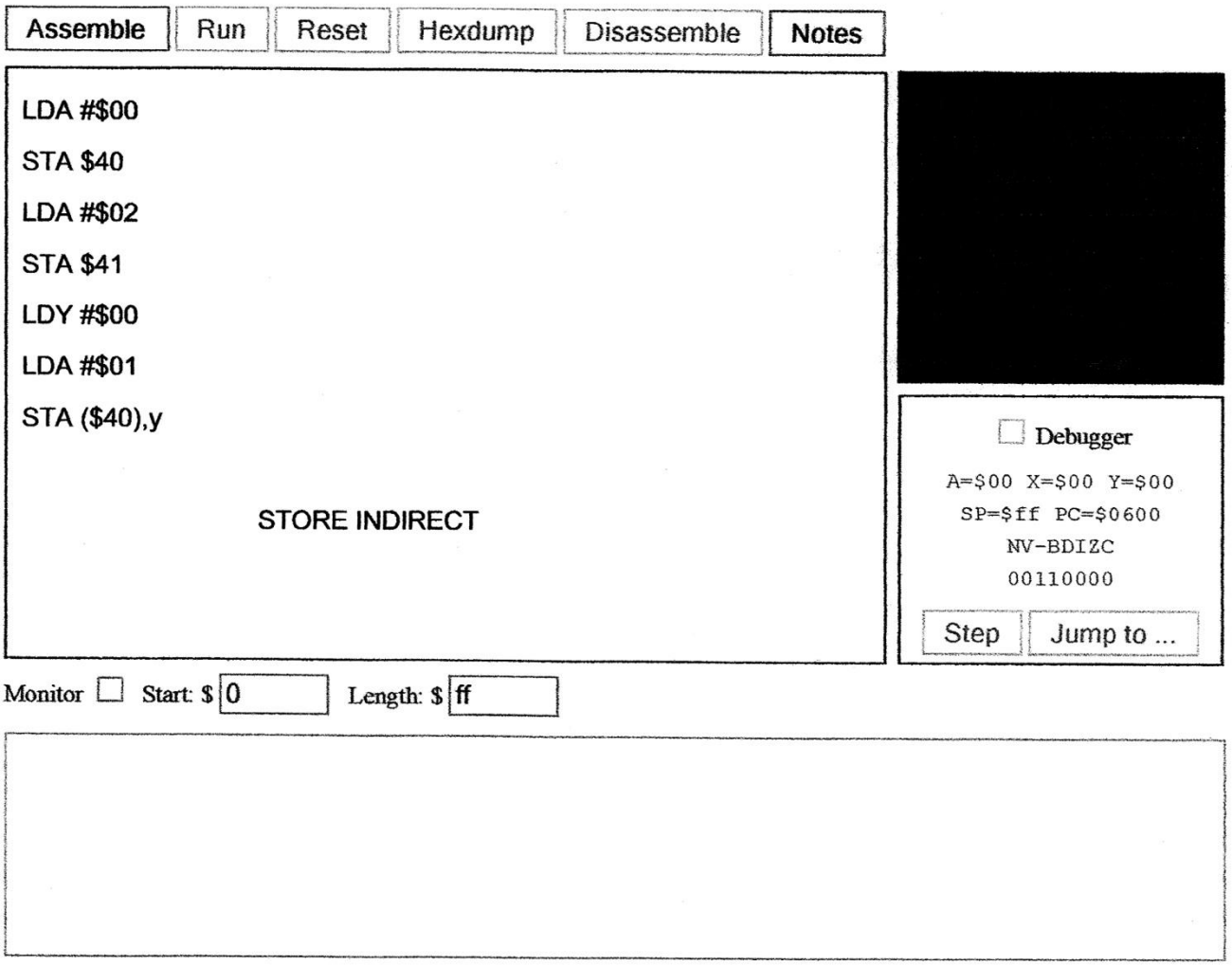



**Figure 64.** Store indirect code diagram

## 4. Key routine and initialization routine

The purpose of the key routine is to automatically repeat a key stroke if a key stays pressed, and to start our test programs with labels. Labels can be put anywhere in code and the key routine will make it jump there.

The simulator stores the last key pressed from the keyboard in memory location 00FF. The first thing to be done is to clear that location so we can wait for a new key to be pressed. Loading 00 into that location will clear it.

The clear instructions are LDA #$00 and STA $FF. Now we want a delay. The loop delay instructions are- Loop:, DEX, and BNE Loop. The first thing needed is to put a delay number into the X register. That number gets decremented by 1 and if the result is not zero, it jumps back to Loop: and decrements again. It does this over and over until the X register is zero. When the X register is zero, it does the next instruction. The reason for the delay is to repeat the key stroke at a slower rate. Without this delay, the line drawing across the screen will draw too fast.

Next on the key routine is - get the key number from location 00FF (LDA $00FF) on the simulator and put it into the "A" register. Then compare that "A" register number to 31 (CMP #$31), if not equal to 31, compare it to number 32 (CMP #$32), if it is not equal to 32, jump back to loop1 and start looking for those two numbers all over again. It's one big loop looking for key 1 or 2 to be pressed. When one of those keys is pressed, the routine will go to a location I named pointer or pointer 1. All keyboard keys have a binary number. Keyboard key 1 = 31 and keyboard key 2 = 32. The output labels of those two keys are called pointer or pointer 1. Those labels can be put anywhere in code. Pressing key 1 or key 2 will make the computer jump to its label to do more code.

We are going to test the key routine. Start by clearing the instruction screen. Type in the initialization routine, then the key routine and hit "assemble". Hit the check box for monitor, write 1FF into the length box, and hit reset. Data will appear on the lower part of the screen. Scroll down to see 0100 (the key location) and hit "run". Press key 1 and 31 will be displayed at location 0100. Hit "reset" and "run" again. Press key 2 and 32 will be displayed at location 0100. If this series of actions complete, the key routine is good.

For the rest of the book, write the test routines after the initialization and key routine. Hitting "run" starts the program address at 0600, which is the beginning of the initialization routine. The test programs use keyboard key number 1. Keyboard key 2 is not used.

The initialization routine stores a 4-digit hex video location in RAM. The upper two digits are stored in RAM address 41 and the lower two digits are stored in RAM address 40. The rest of the test routines from here on will be using an indirect store instruction with Y offset. The last instruction in the initialization routine loads 00 into the Y register to disable the offset.

Initialization and Key Routine Codes. (Extra copy of Figure 65. codes)

```
LDA #$6F
STA $40
LDA #$03
STA $41
LDY #$00
loop1:
LDA #$00
STA $00FF
LDX #$FF
loop:
DEX
BNE loop
LDA $FF
STA $0100
LDA $0100
CMP #$31
BNE loop2
JMP pointer
loop2:
CMP #$32
BNE loop3
JMP pointer1
loop3:
JMP loop1
```

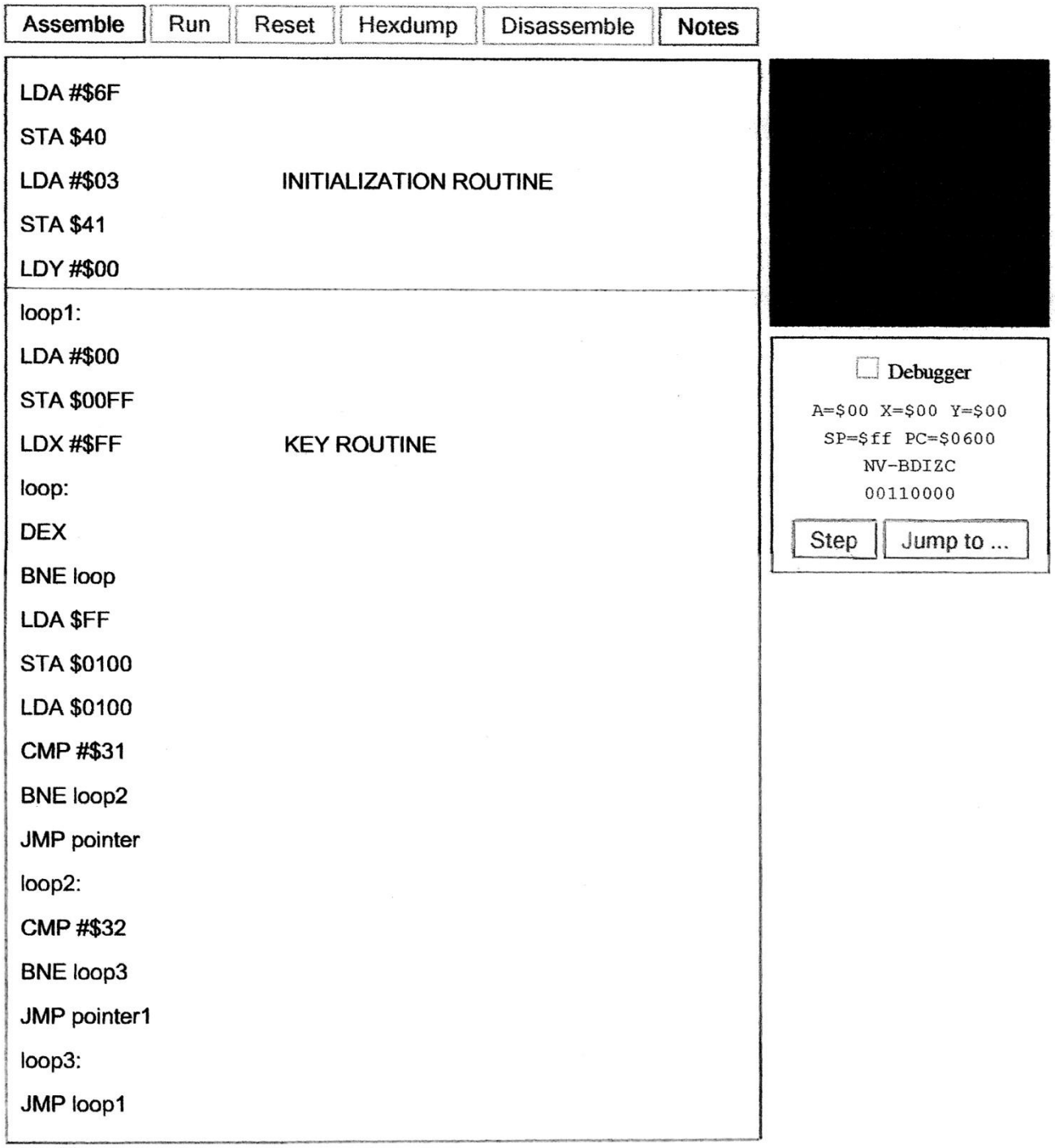



**Figure 65.** Key code diagram

## 5. Increment memory

The increment memory routine will draw a line to the right.
Type in these codes after the initialization and key routine.

```
pointer:
LDA #$01
STA ($40),y
INC $40
JMP loop1
```

When keyboard key 1 gets pressed, the key routine looks for the label named pointer: and jumps here to do this routine.

The increment memory code begins with the initialization routine. It is located (Figure 65) just before the key routine and has a 4- digit hex video location stored in RAM. The higher two digits are stored in RAM address 41 and the two lower digits are stored in RAM address 40. The stored address is 036F, which is our chosen starting video address.

The load "A" register instruction {LDA #$01} will load 01 into the "A" register. Then the store "A" indirect instruction {STA ($40),y} will take 01 from the "A" register along with pixel location 036F that is stored in RAM addresses 40 and 41 and sends them to the display to turn on a pixel. The next instruction (INC $40) will increment the starting video location by 1, so the next pixel will appear to the right of the last one. The last instruction is JMP loop1. That jump instruction jumps back to the key routine to write another pixel if keyboard key 1 stays pressed. When key 1 stays pressed, the key routine jumps back here over and over which keeps pixels incrementing across the screen to the right. If you add another increment code right after the first increment code, you would see a dotted line go across the screen.

It's important to remember that starting this routine begins with the initialization routine at default address 0600 by hitting "run". The rest of the routines in this book will start like this one. Erase the old test routines before writing in the next new routine, but do not erase the initialization and key routine.

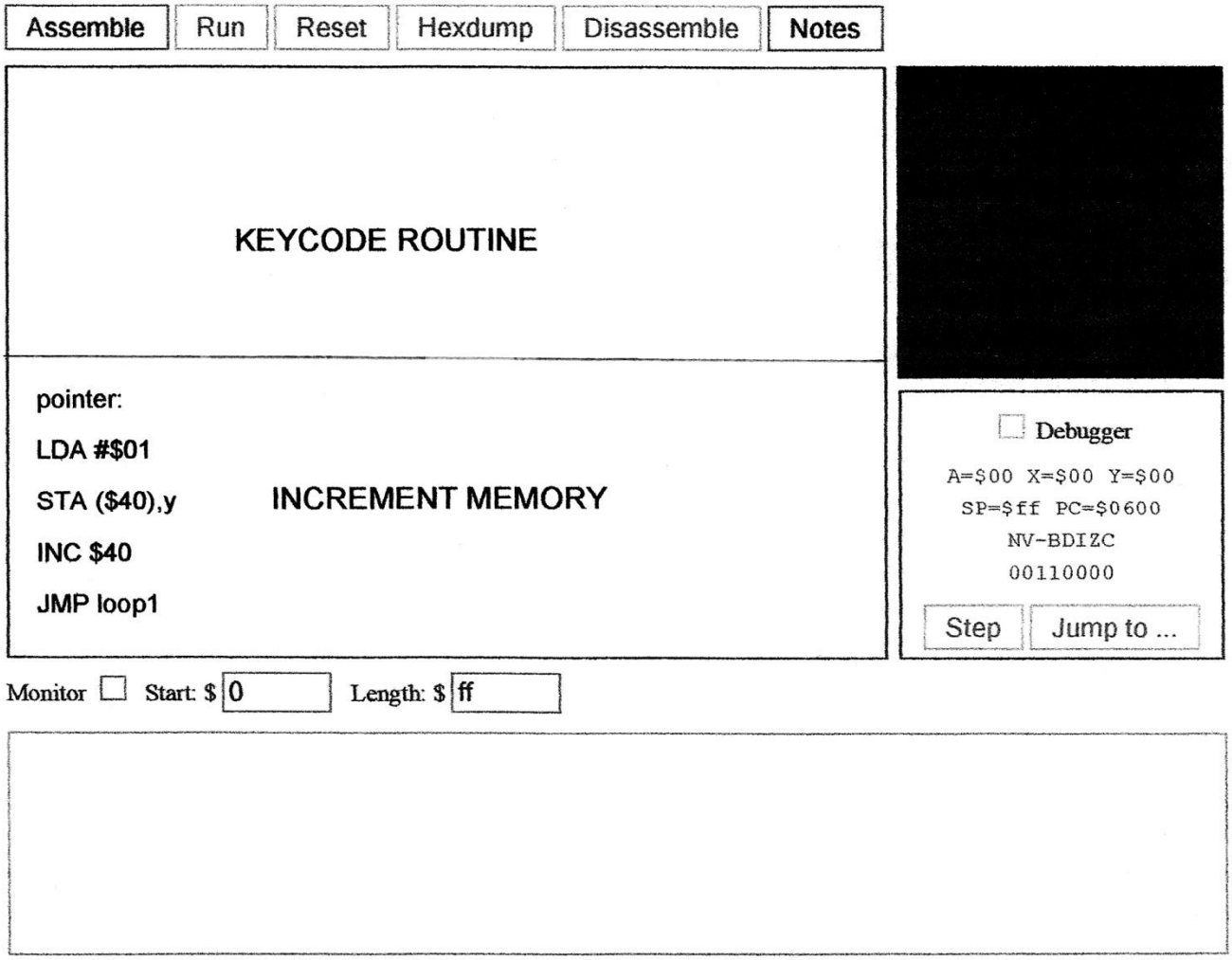



**Figure 66.** Increment memory code diagram

## 6. Decrement Memory

pointer:
LDA #$01
STA ($40),y
DEC $40
JMP loop1

This routine does the same thing as increment memory routine except that it has a decrement memory instruction. The line will draw to the left.

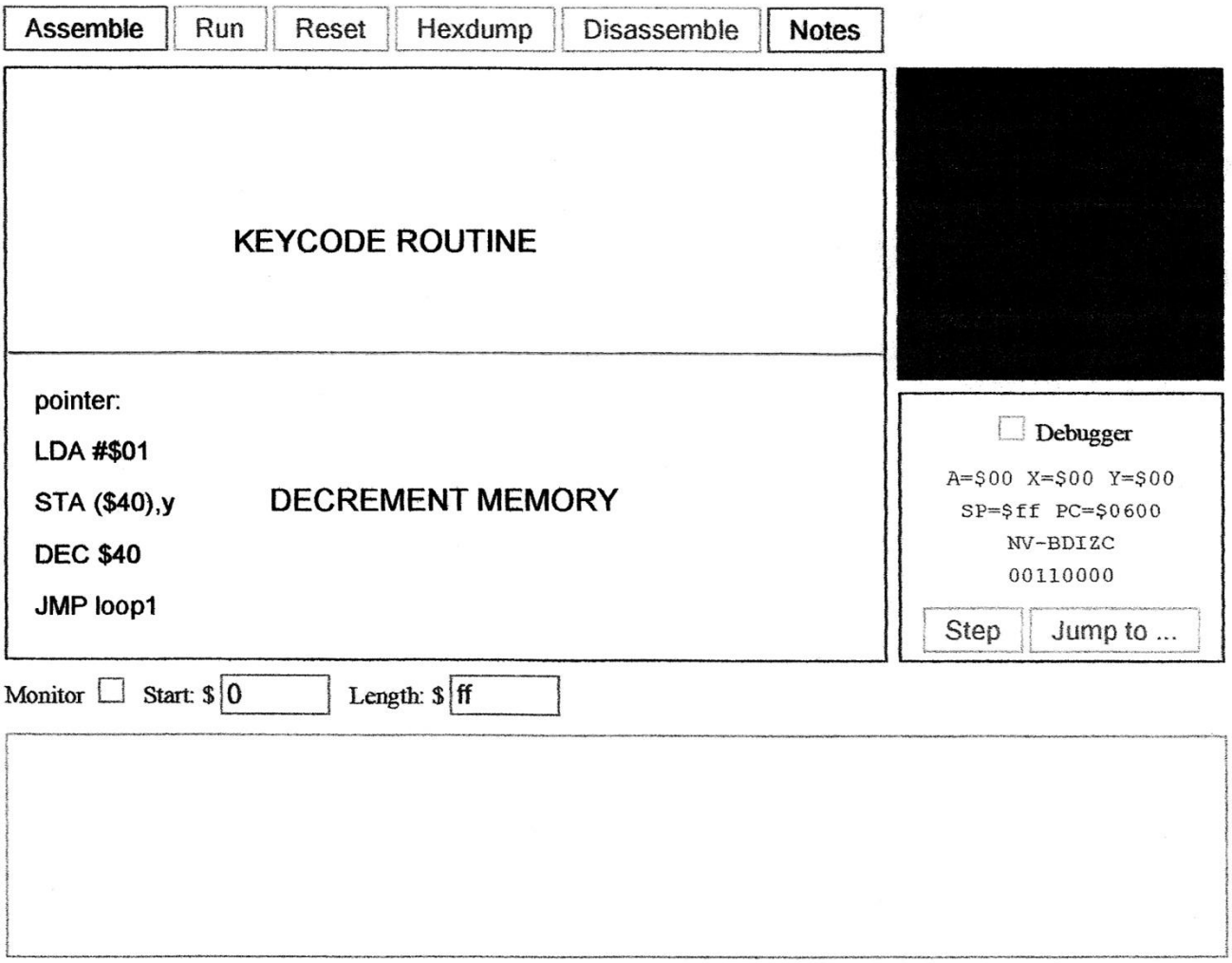



**Figure 67.** Decrement memory code diagram

## 7. Moving Dot

Erase the last instructions and type these.

```
pointer:
INC $40
LDA #$01
STA ($40),y
DEC $40
LDA #$00
STA ($40),y
INC $40
JMP loop1
```

The thing to be noticed about this routine is there are two pixel store "A" instruction {STA ($40),y} codes. The first store "A" instruction using Data 01 turns on a pixel and the second store "A" instruction using Data 00 turns off the pixel that is one address before. So after writing a new pixel, you decrement that location and erase the old pixel. Then you increment once to get back to the original location and go to the key routine to make another pixel if key 1 is still pressed.

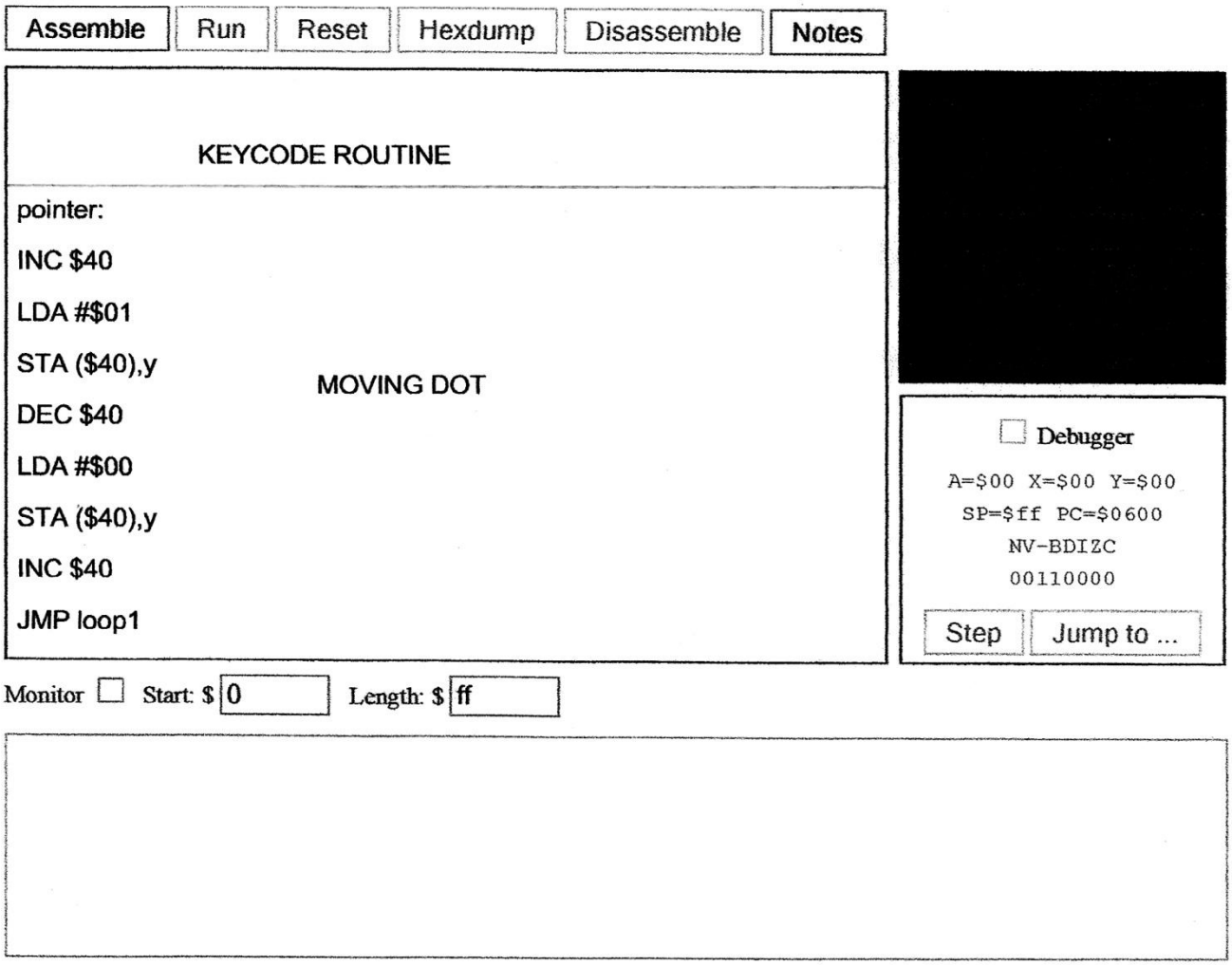



**Figure 68.** Moving dot code diagram

## 8. Moving a line up

Moving a line up or down requires adding or subtracting 20 hex counts. You could write a pixel up one vertical location if you decremented the current location 20 hex times (32), but that would take too much code. (The Pixel Chart Figure 1 shows that it takes 20 hex counts to move a pixel up or down). This routine is going to write pixels up by subtracting 20 Hex from the current location. Before doing this routine, change the starting address to 030F in the initialization routine. This is accomplished by changing the data in the first initialization instruction from LDA #$6F to LDA #$0F. The initialization routine is just before the key routine. See figure 65.

Here is the routine to type in:

```
pointer:
LDA $40
SEC
SBC #$20
STA $40
LDA #$01
STA ($40),y
JMP loop1
```

This routine starts by taking the video location that is stored in the RAM at address 40 and putting it into the "A" register. Set carry (SEC) to take care of carry problems. Then subtract the from the "A" register 20 hex counts and store that data back into RAM location 40. The load "A" instruction will load 01 into the "A" register. Then the store "A" instruction {STA ($40),y} will take 01 from the "A" register along with the new location that is stored in RAM addresses 40 and 41 and sends them to the display to turn on a pixel. Every time a pixel goes up, 20 Hex is being subtracted from the current location. The current location is in Ram at address 40 and 41.

Change the SBC #$20 instruction to SBC #$40 and you will then see the pixels going up with spaces.

You can execute this routine one pixel at a time by hitting key 1 momentarily.

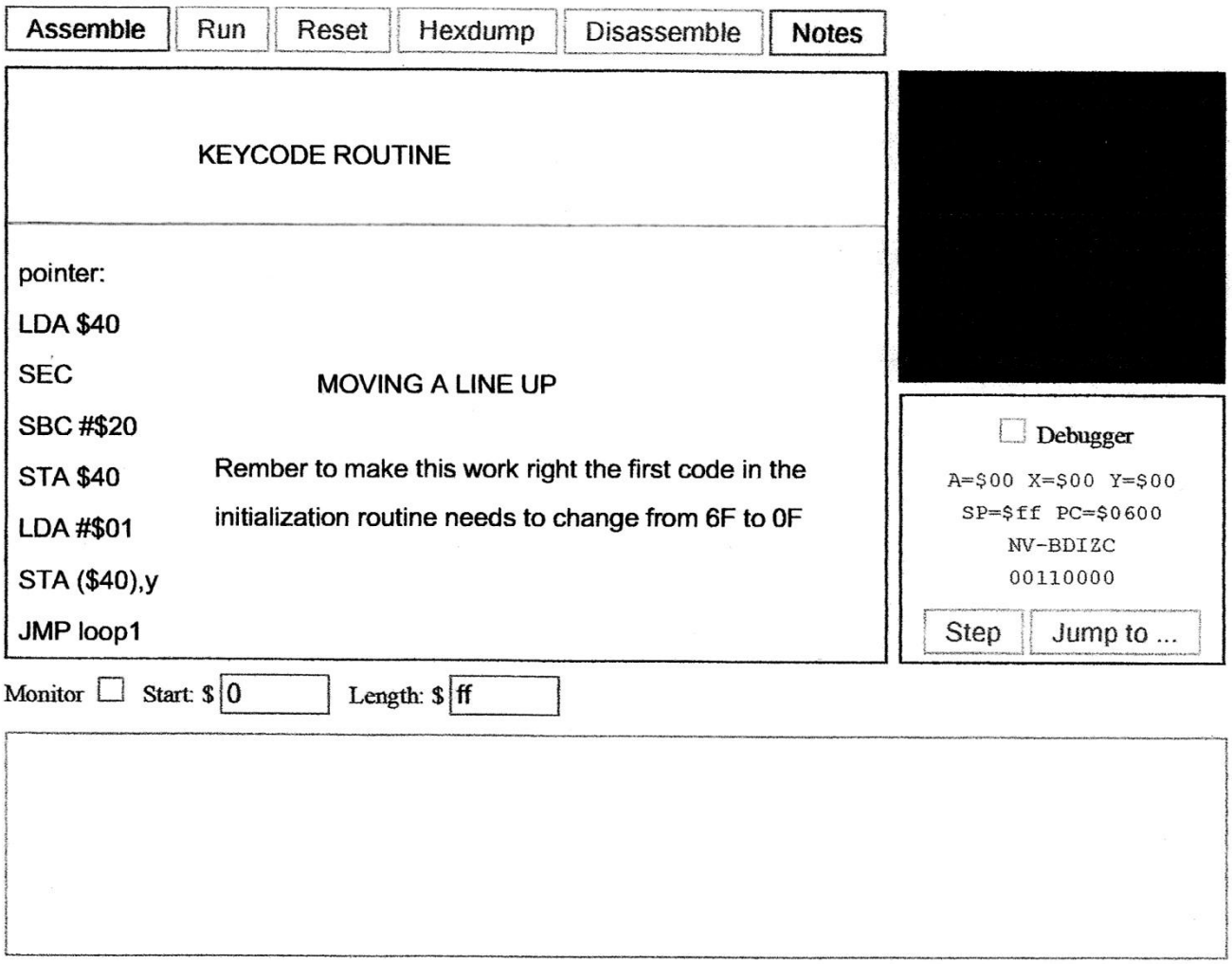



**Figure 69.** Moving a line up code diagram

**Putting the pieces together**

So how does a computer work? The 6502 CPU can do 250 different things, but I only showed you one. I have 249 more to go. This book is about computer hardware. The software in the book is to show you how the hardware works. It is the hardware that has been neglected in training today. If you are a programmer, you will be a better programmer if you know how hardware works. If you are a technician, you will have easier time repairing digital equipment if you know how hardware works.

Chapter 7 shows you how the hardware works for the load "A" register instruction. But you need to be able to understand all of the 250 instructions. The CPU timer controller parts on the one instruction diagram (figure 61) are the instruction register, the counter, and the Rom. The controlled part is the "A" register. But more controlled parts are needed for the other 249 instructions. The timer controller on the block diagram (figure 2) controls these parts: the "A" register, the "X" register, the "Y" register, the ALU, the data latch, and the program counter. The real 6502 diagram on figure 59 shows more of the controlled parts and the complete real 6502 diagram on figure 60 shows all the controlled parts. It takes 64 timing signals to control all of the controlled parts. When an instruction is executed, the timer controller selects the timing signals that are needed for the instruction and then outputs them one by one to the CPU controlled parts.

The one instruction diagram is only good for the load "A" instruction. But if we add the "X" Register to the diagram, and we add another data timing bit to the Rom (D5), and we add the programming in the Rom for the "X" register instruction, and we connect data bit D5 to the input control of the "X" Register, the whole circuit would be able to do two instructions (load "A" register and load "X" register). If we added a total of 64 data bits to the Rom, programmed the Rom for every instruction, and added all the CPU controlled parts, we would be able to execute every instruction. The completed CPU could be used in any computer system. I think you the reader, with the help of a 6502 programming book, could figure out how to add more instructions to the one instruction diagram. If this explanation is not clear to you, go back and review the timing section again.

**Switch inputs**

The switch inputs that are used on the simulator come from the keyboard keys of your laptop computer. Each keyboard key is a press switch. A "1" is applied to one side of the switch and the other side of the switch receives a "1" when pressed. A key not pressed is a "0". Inside your laptop computer the keyboard outputs go to the input port. The input port outputs the key information to the data bus. When the computer wants to read the keys, it enables the input port to send the key information to the data bus. The binary number value on the data bus tells the computer which key is pressed. The same concept applies to input port hardware on the schematic diagram (figure 3). That input port has 8 inputs. If 8 button switches were connected to the input port, the hardware would be able to read the 8 switches. The output port has 8 outputs. Each of those outputs can select a bank of 8 switches. By inputting 8 banks of switches, one bank at a time, the hardware could read 64 button switches.

**Monitor**

I am not going to explain monitors or the type of video signals that they need. You will be able to find that information, if you search the web.

Clock

I did not include very much information about clocks, how they are generated and how they can be divided up. You can easily find information, about clocks.

Pixel color

The web simulator can generate 16 pixel colors. My hardware on figure 3 can only generate black and white. If my real hardware had four one bit Rams instead of one, 16 colors could be generated. Each ram would be a color bit. Four color bits in binary equal 16 different colors.

Throughout this book I have explained most of the code for the Etch-a-Sketch® game. But extra code was needed for more keys, the flashing red pixel, 16 bit addition and subtraction, and border control. If you write pixels past the left or right border of the screen, it will write on the other side of the screen. If you try to write pixels past the bottom border of the screen, the game program will crash. I added software to fix those border problems.

In conclusion, there's a lot of technical information in this book. I tried to get it all perfect so you wouldn't become confused. But the real value of the book is not in the details. It was written to give you a good idea of how computers work. There is a lot more to know about computers, but the fundamentals that you now know can take you anywhere. Don't forget that whatever your career goals might be, unless you show enough initiative you will be limiting your opportunities for advancement.

I didn't go to school for engineering, but on my own, I made two devices in the early 90s using the 6502. One was a scrolling LED sign on the back window of my truck, with a remote to select a message. The other was a home telephone restrictor that could limit time and phone numbers. Dialing a key code could turn it off.

Being able to create the software and hardware all by myself on those two projects, and knowing that people struggle when they try to understand computers, is what motivated me to write this book.

Some of the diagrams in this book have instructional videos that I put on YouTube. This is the link to those videos - https://goo.gl/S3qfX7

Using a PC computer is the easiest way to try the software routines in this book. The simulator website is - http://skilldrick.github.io/easy6502/

When you finish this book, please go to my YouTube book site and leave comments.

My YouTube site - https://goo.gl/RyB1Ft

Here is the “etch-a-sketch®” game that goes with my book. Keyboard key 1 = left, key 2 = right, key 3 = up, key 4 = down. Key 5 erases one pixel. After erasing a pixel, the pixel location is still in memory. This means you might need to rewrite a pixel. I know this sounds confusing but you will see that it is easy to figure out. I added a flashing red pixel to show you where you are at on the screen. Use the erase key to erase all unwanted pixels on the screen. Start the game by pressing one of the four move keys. The simulator can be found by searching - easy 6502 by skilldrick - on the web. Copy this code from the bottom to the top and paste this program code to one of the simulator windows.

```
LDA #$6F
STA $40
LDA #$03
STA $41
LDY #$00
loop1:
LDA #$00
STA $00FF
LDX #$FF
loop:
DEX
BNE loop
LDA $FF
STA $0100
LDA $0100
CMP #$31
BNE loop2
JMP pointer
loop2:
CMP #$32
BNE loop3
JMP pointer1
loop3:
CMP #$33
BNE loop4
JMP pointer2
loop4:
CMP #$34
BNE loop5
JMP pointer3
loop5:
CMP #$35
BNE loop30
JMP pointer4
loop30:
JMP loop1
pointer:
LDA $40
CMP #$00
BNE loop10
JMP loop1
```

```
loop10:
CMP #$20
BNE loop11
JMP loop1
loop11:
CMP #$40
BNE loop12
JMP loop1
loop12:
CMP #$60
BNE loop13
JMP loop1
loop13:
CMP #$80
BNE loop14
JMP loop1
loop14:
CMP #$A0
BNE loop15
JMP loop1
loop15:
CMP #$C0
BNE loop16
JMP loop1
loop16:
CMP #$E0
BNE loop17
JMP loop1
loop17:
DEC $40
LDA #$0A
STA ($40),y
LDX #$FF
loop31:
DEX
BNE loop31
LDA #$01
STA ($40),y
JMP loop1
pointer1:
LDA $40
CMP #$1F
BNE loop18
JMP loop1
loop18:
CMP #$3F
BNE loop19
JMP loop1
loop19:
CMP #$5F
```

```
BNE loop20
JMP loop1
loop20:
CMP #$7F
BNE loop21
JMP loop1
loop21:
CMP #$9F
BNE loop22
JMP loop1
loop22:
CMP #$BF
BNE loop23
JMP loop1
loop23:
CMP #$DF
BNE loop24
JMP loop1
loop24:
CMP #$FF
BNE loop25
JMP loop1
loop25:
INC $40
LDA #$0A
STA ($40),y
LDX #$FF
loop32:
DEX
BNE loop32
LDA #$01
STA ($40),y
JMP loop1
pointer2:
LDX #$00
STX $43
LDA $40
SEC
SBC #$20
STA $40
LDA $41
SBC #$00
STA $41
LDA $40
CLC
ADC #$20
STA $40
LDA $40
SEC
SBC #$20
```

```
STA $40
LDA #$0A
STA ($40),y
LDX #$FF
loop33:
DEX
BNE loop33
LDA #$01
STA ($40),y
JMP loop1
pointer3:
LDX $43
CPX #$01
BNE loop8
JMP loop1
loop8:
LDA $40
CLC
ADC #$20
STA $40
LDA $41
ADC #$00
CMP #$06
STA $41
BNE loop6
LDX #$01
STX $43
JMP loop1
loop6:
LDA #$0A
STA ($40),y
LDX #$FF
loop34:
DEX
BNE loop34
LDA #$01
STA ($40),y
JMP loop1
pointer4:
LDA #$00
STA ($40),y
JMP loop1
```