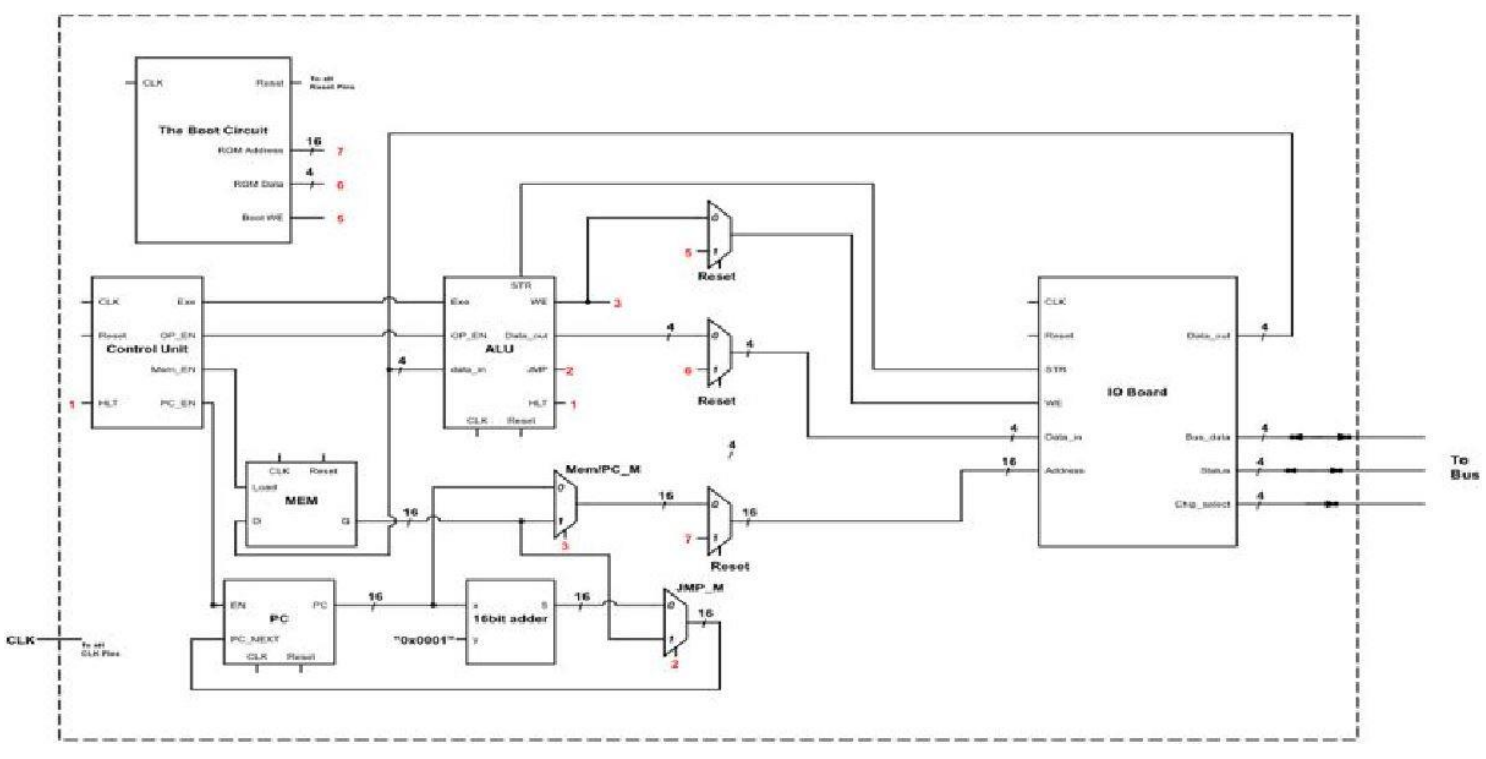
A Computer from Ground Up



A Nibble Knowledge Story

Table of Contents

1. **Preface 4**
   - 1.1. Preface 4
   - 1.2. Nibble Knowledge Project Summary 6
   - 1.3. Nibble Knowledge Project Motivation and Objective 7
   - 1.4. Acknowledgments 9
2. **Introduction to Computers 10**
   - 2.1. Overview of Computer Organization 10
   - 2.2. Operating Systems – A High Level Introduction 17
   - 2.3. Number Systems: Decimal, Binary, Hexadecimal, and Addition, Signed Binary 20
   - 2.4. Languages: Machine, Assembly and Instruction – A High Level Introduction 40
   - 2.5. Memory – A High Level Introduction 42
3. **Combinational Logic and Sequential Logic 46**
   - 3.1. Logic Gates 46
   - 3.2. Boolean Algebra and Equations 53
   - 3.3. Hardware Reduction Techniques 60
   - 3.4. Timing and Delays 64
   - 3.5. Multiplexers 69
   - 3.6. Flip-Flops and Latches 75
4. **Arithmetic Circuits 79**
   - 4.1. Addition 79
   - 4.2. Subtraction 86
   - 4.3. Counters 87
   - 4.4. Multiplication/Division – High Level 96
   - 4.5. Comparator 100
   - 4.6. Registers: Shift Registers 104
   - 4.7. ALU 106
5. **Computer Architecture 115**
   - 5.1. Instruction Sets 115
   - 5.2. Assembly Language 119
6. **Nibble Knowledge Computer High Level Design 124**

7. The CPU **129**
8. Audio Controller **169**
9. IDE Controller **189**
10. PS/2 Keyboard Controller **199**
11. Serial Communication RS232 Controller **211**
12. Software **222**
13. VGA Controller **234**
14. Peripheral and CPU Integration **243**
15. Glossary **248**
16. Appendix **254**
16.1. Oscillator Circuits – Alternate Design 254
16.2. IDE Controller – Arduino Test Code 257
16.3. PS/2 Keyboard Controller – Main VHDL Code 264
16.4. PS/2 Keyboard Controller – Internal Multiplexer VHDL Code 268
16.5. Software – Complete CUTE Basic Compiler 269
16.6. Software – Complete Macro Assembler 295
16.7. Library Function Example – 32 Bit Multiplication 304
16.8. References 309

1. Preface

1.1. Preface

Computers are one of the most fascinating machines ever built by man. They exhibit behaviour that no other machine can imitate - they can make decisions based on external input. This decision making ability allowed us to construct devices of unimaginable functionality just 50 years ago - we have computers that display simulations of our physical world, that can provide immersive fantasy worlds, and we even have computers that can simulate the behaviour of living organisms like you or me (though that one is a lot harder).

Computers are also one of the most complex machines humanity has ever built - a modern desktop processor has on the order of 10 billion tiny switches called “transistors”, and newer graphics cards have closer to 20 billion transistors. These have to be organized within tenths of millionths of centimeters of each other, and turn on and off four hundred million times faster than you can perceive changes with your eyes. They are, in fact, so complex that humans are no longer capable of designing new computers alone - enormous teams of people use computers to design the next generation of computers together. This is not new - the last desktop processor to be designed entirely by humans (with computers merely for drawing or simulation of the new processor) was over thirty years ago.

So where does this book fit in? Well, we’re not here to teach you about modern processor design techniques - far from it. Instead, we’re going the opposite way. We’re going to show you how a computer works by walking you through a processor that was designed by a team of just three people. It uses around one millionth of the transistors. It switches about a thousand times slower. It’s made of big pieces you can assemble with your hands. It is one of the simplest and slowest computers you will encounter in your life - it’s probably even slower than your calculator!

You may be wondering at this point - what’s the point? Why teach me about a computer that is so much slower and less complicated than the other computers? Well, there’s a really good answer to that actually - it’s called Turing Equivalence. Turing Equivalence is a rather complicated concept to explain with complete precision (and I will not attempt to), but it boils down to a very simple thing: any computer, with enough memory and time, can do anything that any other computer can

do. This simple computer, with enough memory and time, can indeed run the newest video games (though it would take an extraordinary amount of effort and the time needed to even launch the game would be simple astronomical) or run the same simple programs you can write on your personal computer. The most important thing that it means though is that if you understand how *this* computer works you are know very strongly equipped to understand how *every* computer works because they all have to act in a similar fashion.

We hope you enjoy. Computers are really enjoyable to tinker with, and getting them to do what you want is a challenging but a deeply rewarding process. Don't give up, and have fun!

1.2. Nibble Knowledge Project Summary

The Nibble Knowledge team consists of fifteen students nearing the completion of an Engineering degree in Software, Electrical, or Computer engineering. The main goal of the team was to design a 4-bit educational computer kit that is aimed at revolutionizing the way Canadian population understands and perceives computer literacy. Even though computers have become a mainstream resource, the complexity has caused the masses to not have the drive, or adequate resources to further improve their understanding of a computer. For that reason, this kit has been designed for people with no prior computer architecture or programming knowledge.

The documentation outlines the methodology and design of the peripheral bus, controllers and devices, CPU, and Software. The five peripherals are Audio Controller, Serial Communication, IDE controller, PS/2 controller, and VGA controller. The audio controller can input and output sound, serial communication is a standard method of communicating between computers and peripherals, the IDE controller in a mass storage device, the PS/2 controller takes in user input, the VGA controller can display a visual output to a monitor while the CPU, controls all these processes. The computer uses Cute BASIC programming language, which is translated to macro assembly language by a macro assembler.

The success in the construction of the kit has been determined by comparing the initial product scope, with the final product scope. Technical specifications of the final product highlight the features and degree of usability of each device/peripheral. The methods used for testing have been described in detail to showcase how resulted were validated and success was measured. The documentation is concluded with a list of tools, materials, supplies and costs so users of the kit are aware of what is needed to build a 4-bit computer. All the code written for this 4-bit computer is shown in the appendix, as well as referenced to the team GitHub account.

Since this computer kit is provided with openly accessible resources and documentation, content can be self-taught, and users can assemble the kit using easily accessible materials. The ICT sub-sector in Canada has approximately $160 billion in revenue, which shows there’s a large potential for growth for people invested in this sector.

1.3. Nibble Knowledge Project Motivation and Objectives

Project Motivation:

In the current day and age, the ability to understand how a computer works and the ability to use a computer is crucial in almost all walks of life. All professions and businesses now include some computer use, and as such, improving computer literacy is now a very important educational goal. Computers are a ubiquity and a necessity in the modern world. However, in the Albertan secondary education system, dedicated computer classes are a relative rarity, even though interest in computers is at an all-time high. Countries such as the UK and China have successfully pushed ahead with dedicated computer courses to give their students an educational advantage. We wanted to give Canada a chance to do the same. Moreover, due to globalization, the entire planet is now interconnected and this has mainly been achieved due to the advancements in computer technologies. Continuously, more and more jobs are being created in the fields related to computers, and thus learning how a computer works is highly valuable. It is common knowledge that a computer is a very complicated instrument and we realized that younger students (High School and early post-secondary) do not understand the technical details of a computer. With these ideas in mind, we decided to build a platform that could be used to teach or learn about processors. We came across many platforms such as Raspberry Pi, Arduino and BeagleBone Black, which have already been designed and marketed for a similar purpose. However, we realized that one may purchase any of the above or similar platforms, but none of the platforms describe how one can actually build a CPU. This is how our project motivation originated. From an educational standpoint, we wanted to create a complete package that would teach a person the internal functionalities of a CPU, all the way up to interfacing with Peripherals and basic programming.

The project objective is to create a platform that would teach an individual how to make a simple computer. The problem we are attempting to solve is that no computer kit is available on the market that actually allows students to delve into the inner workings of the hardware, and thus there is a knowledge gap when it comes to understanding computer functionality from an educational perspective. We aim to create a product that fills this market niche. Our objective is to design a 4-bit computer system package (kit) that comprises of discrete logic components on breadboards, along with an operating system, a compiler, and associated documentation for all components describing exactly how all the circuits and code works. All the circuit diagrams, programming

code, and documentation will be available online as open source for free; allowing anyone to understand and develop the system. Furthermore, this allows the Internet community to naturally help develop the system and help translate the components into more natural languages. The computer will be compatible with peripherals such as Serial Communication, IDE Hard Drives, PS/2 Keyboards, VGA Monitors, and will also have an audio output. The two main sellable components would be the kit itself (which is assembled, tested, then packaged and sold) and hard copies of the documentation.

1.4. Acknowledgments

First and foremost, we thank the University of Calgary and the Schulich School of Engineering for developing and having a wonderful Department of Electrical and Computer Engineering. Along with that, we thank Dr. Denis Onen and Dr. Steven Norman of the University of Calgary, for their incredible support as Academic Advisors for the Nibble Knowledge Project team. Their advice, suggestions, their technical assistance and their belief in our team have made this project a success, and hence, this textbook possible.

Furthermore, we thank the magnificent Technicians in the Electrical and Computer engineering department, without whom, this project would not have manifested. We thank Warren. F, Richard. G, Garwin. H, Andrew Michael. L, John. S, Christopher. S and Rob. T for their help and assistance throughout the project. The technician's combined experience of over 100 years is engrained in this project. The Technicians also made it possible to acquire and use all the necessary tools at the team's convenience. We sincerely thank you all.

We also appreciate the course coordinator of the project course in the Department of Electrical and Computer engineering, Dr. Hamidreza Zareipour and the Teaching Assistant Mark Li, for their continuous efforts, support and understanding throughout the year.

2. Introduction to Computers

2.1. Overview of Computer Organization

How would you describe a computer to someone? A software engineering might begin to describe a computer from a software perspective, while, an electrical engineering might being to explain it from a hardware perspective. Let’s look at the simplest definition of a computer. In simple terms, a computer is a machine that performs tasks and calculations based off a set of instructions, or program operations. Electronic computers were introduced in the 1940s and they were huge machines that actually required a team of individuals to operate. The first computer was invented by an English Mechanical Engineer and polymath in the early 19th century. Charles Babbage, originated the concept of a programmable computer. Considered the "father of the computer", he conceptualized and invented the first mechanical computer in the early 19th century.

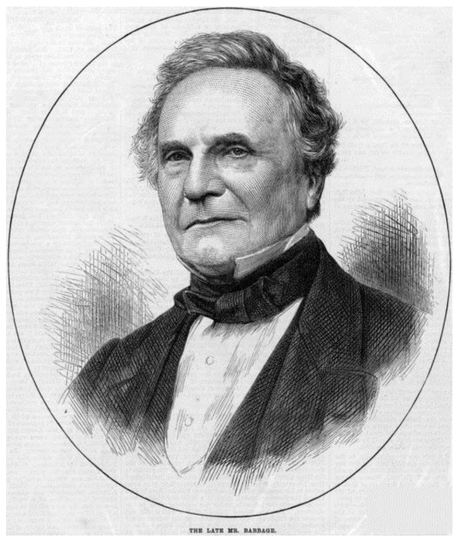


Figure 2.1.1: Charles Babbage (1791-1871)

After working on his revolutionary difference engine, designed to aid in navigational calculations, in 1833, he realized that a much more general design, an Analytical Engine, was possible. The input of programs and data was to be provided to the machine via punched cards, a method being used at the time to direct mechanical looms such as the Jacquard loom. For output, the machine would have a printer, a curve plotter and a bell. The machine would also be able to punch numbers onto cards to be read in later. The Engine incorporated an arithmetic logic unit, control flow in the form of conditional branching and loops, and integrated memory, making it the first design for a general-purpose computer that could be described in modern terms as Turing-complete.

In contrast to early computer systems, today's computers are vastly smaller and amazingly powerful. Not only are they thousands of times faster but they are small enough to fit on your desk, on your lap, or even in a pocket.

In a nutshell, computers work through an interaction of physical hardware components and software instructions. Hardware is the actual physical parts of a computer that you can see with your eyes or touch with your hands. Software is basically a collection of code that makes up a computer program. The hardware in turn carries out instructions by a computer program on a complex level using electricity. Other hardware items such as a monitor, keyboard, mouse, printer, video card, sound card, or other physical piece are often called hardware devices or even just devices for short. When looking at software, it is the actual instructions, or programs, that tell the hardware components what to do. Without software the hardware is essentially useless. A computer game, word processing program, or Internet browser are some common types of computer software.

The most important piece of software for a computer system is the operating system. It is responsible for managing the entire computer and all the devices connected to it. Popular operating systems include Windows XP, Windows Vista, Mac OSX, and Linux among others that you might be familiar with.

A computer is organized in four main sections. These sections are the Central Processing Unit (CPU), the Memory, Input Devices and Output Devices.

The **Central Processing Unit (CPU)** is where the decisions are made. It takes cares of all the computations and the input/output requests are handled.

The **Memory** is what stores the information being processed by the CPU. More on this later.

The **Input Devices**, such as your PS/2 Keyboard, allows users to provide information to the computer, while, the **Output Devices**, such as an Audio Speaker, allows users to receive

information from the computer, in this case, in the form of a pleasant sound, hopefully. Below is a figure that illustrates the interactions between the four sections of the computer.

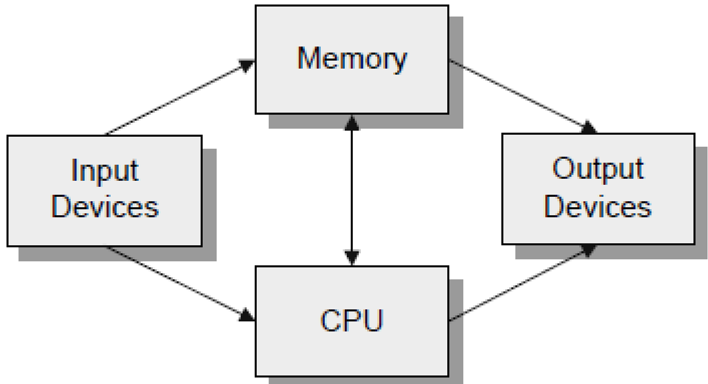


Figure 2.1.2: Different components of the computer

CPU

In simple terms, the CPU is the brains of the computer. It has a unit called the Arithmetic Logic Unit (ALU) that performs all the arithmetic calculations. The arithmetic calculations are of the binary number system, which is discussed in details in the next section. A Control Unit within the CPU decodes and executes instructions that are supplied to the CPU by the software. A detailed look into the CPU is provided in Chapter 7 of the textbook.

The Program Counter (PC), holds the memory address of the instruction. The Control Unit fetches the instruction that the program counter is pointing at, then the program counter gets incremented and the instruction that was fetched gets executed. This cycle repeats as per the demands of the software.

The Input and Output Devices

The input and output devices allow the computer to perform tasks that the user of the computer likes. The input and output devices receive information for processing, store necessary information and return the results of processing. Most commonly used devices are the Monitor, Speakers, Mouse, Keyboard, Printers and etc.

Below is a diagram that illustrates the connections between the CPU, the memory and the input/output (I/O) devices.

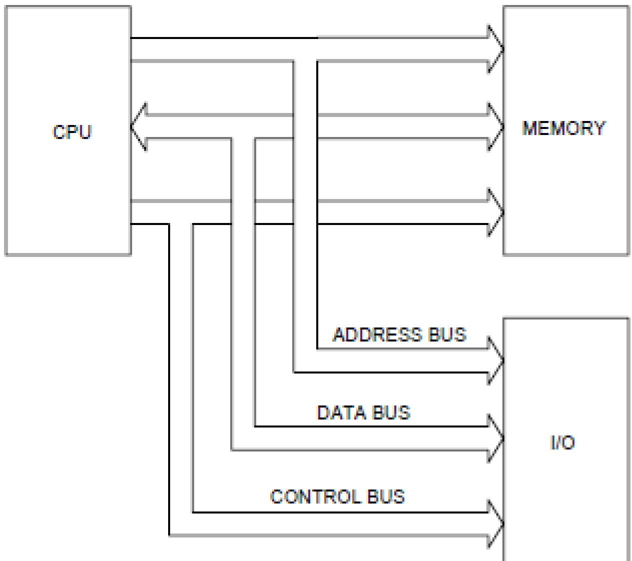


Figure 2.1.3: The CPU connected with memory and I/O devices

Software

The above information brings us to the Software aspect of a computer. First, let's look at the history of software development.

First programming languages

The first programming languages that were designed to communicate instructions to a computer were written in the 1950s. An early high-level programming language designed for a computer was Plankalkül, which was developed by the Germans for the functional program controlled Turing-complete Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000.

Figure 2.1.4: Konrad Zuse (1910-1995)

Short Code

John Mauchly developed the first Short Code in 1949, which was one of the first high-level languages ever developed for an electronic computer. Unlike Machine Code (section 2.4), Short Code statements represented mathematical expressions in understandable form. However, the program had to be translated into machine code every time it ran, making the process much slower than running the equivalent machine code.

Auto code

Auto Code is a programming language used by a compiler to automatically convert the language into machine code. The first code and compiler was developed in 1952 for the Mark 1 computer at the University of Manchester and is considered to be the first compiled high-level programming language.

The first major languages

In 1957, the first of the major languages appeared in the form of FORTRAN. Its name stands for Formula Translating system. The language was designed at IBM for scientific computing. The components were very simple, and provided the programmer with low-level access to the computers insides.

Algol

The Algol language was created by a committee for scientific use in 1958. It's major contribution is being the root of the tree that has led to such languages as Pascal, C, C++, and Java. It was also the first language with a formal grammar.

Now, that a little bit of software history is covered, let's consider the application of software. There are two main types of software: Application software and System Software.

Application Software

Application software consists of programs designed to perform specific tasks that are visible to the user. It is due to this type of software that the use of computers has become so popular. Most common application software are word processing like Microsoft Word, spreadsheets like Microsoft Excel, drawing programs, publishing programs and presentation programs like Microsoft PowerPoint to name a few.

System Software

System software consists of programs that support the execution and development of other programs. They two major types are Translation Systems and Operating Systems.

Translation system are set of programs used to develop software. As the names states, the key factor in translation system is the translator, which could be a compiler or liner for instance. A compiler converts one language to another, while a linker combines resources. An example of this is the Microsoft Visual C++. The compliers and linkers work towards editing, compiling, linking object and library files, loading, executing and observing how a program reacts. The compilation process of the Nibble Knowledge Computer Cute BASIC language is as follows.

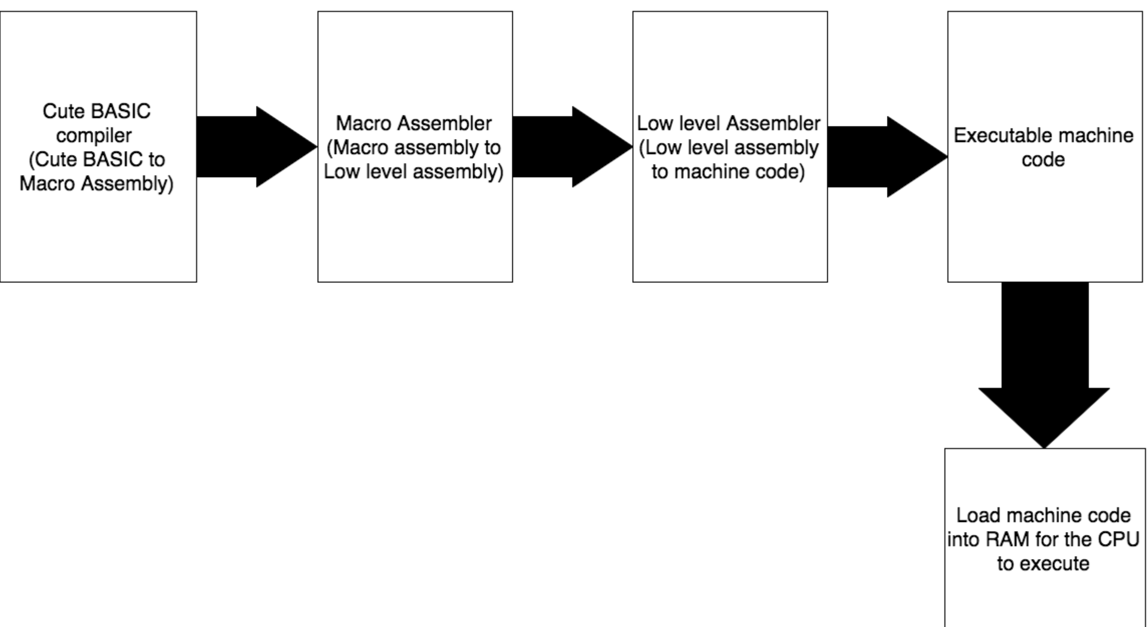


Figure 2.1.5: Compilation process for Cute BASIC

The operating system controls and manages the computational resources of the computer. Some examples of operating systems are UNIX, Mac OS X and Windows. Operating systems provide file systems which consist of directories, folder and files. Moreover, operating systems manage the programs that are running on the computer along with managing how the computer interacts with the I/O devices. More on operating systems in section 2.2 below.

2.2. Operating Systems – A High Level Introduction

Computers are a part of almost every application, so a discussion on operating systems continues to teach the fundamental building blocks of a computer. In fact, operating systems are a crucial part of any computing system.

An operating system is a software program that manages computer hardware. It is essentially an intermediary between the user of a computer and the computer hardware. The main purpose of an operating system is to provide a user-friendly environment where programs can be executed in a convenient and efficient manner.

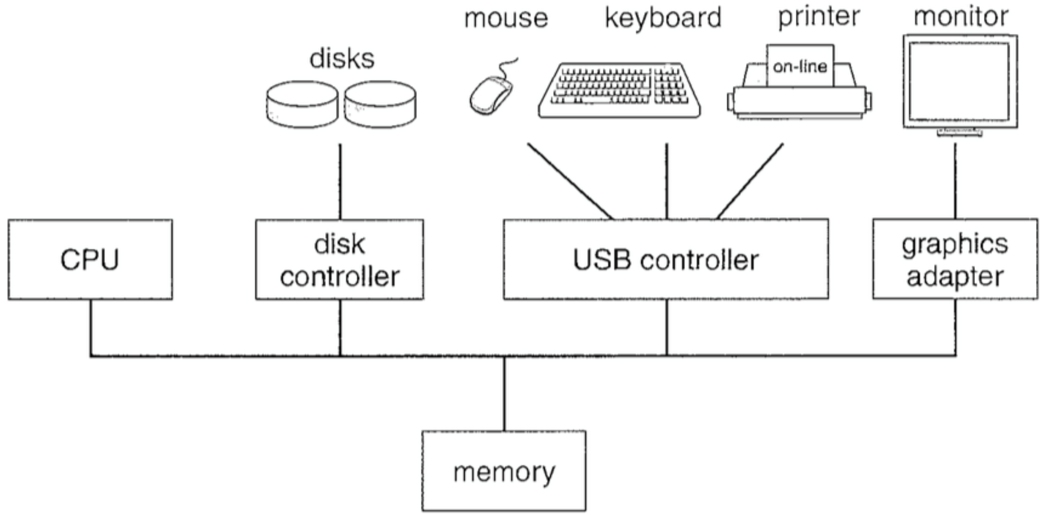


Figure 2.2.1: Overview of a Computer

Even though the operating system is designed to manage/control actions of the hardware; alternatively, hardware must also provide the needed mechanisms to allow and ensure appropriate computer system operation. The hardware must disable user programs from interfering with the needed operation of the system.

Due to the complexity of the tasks an operating system has to perform, it is vital that it is created in bits. Each section should be defined in detail in terms of functions, inputs and outputs. The detailed definitions of the system form the basis of choices amongst various strategies and algorithms. This allows each piece to be a proper delineated section of the system. The internal structure of an operating system varies greatly since it is organized along various lines. This proves designing an operating system is a huge task.

Operating System Performance

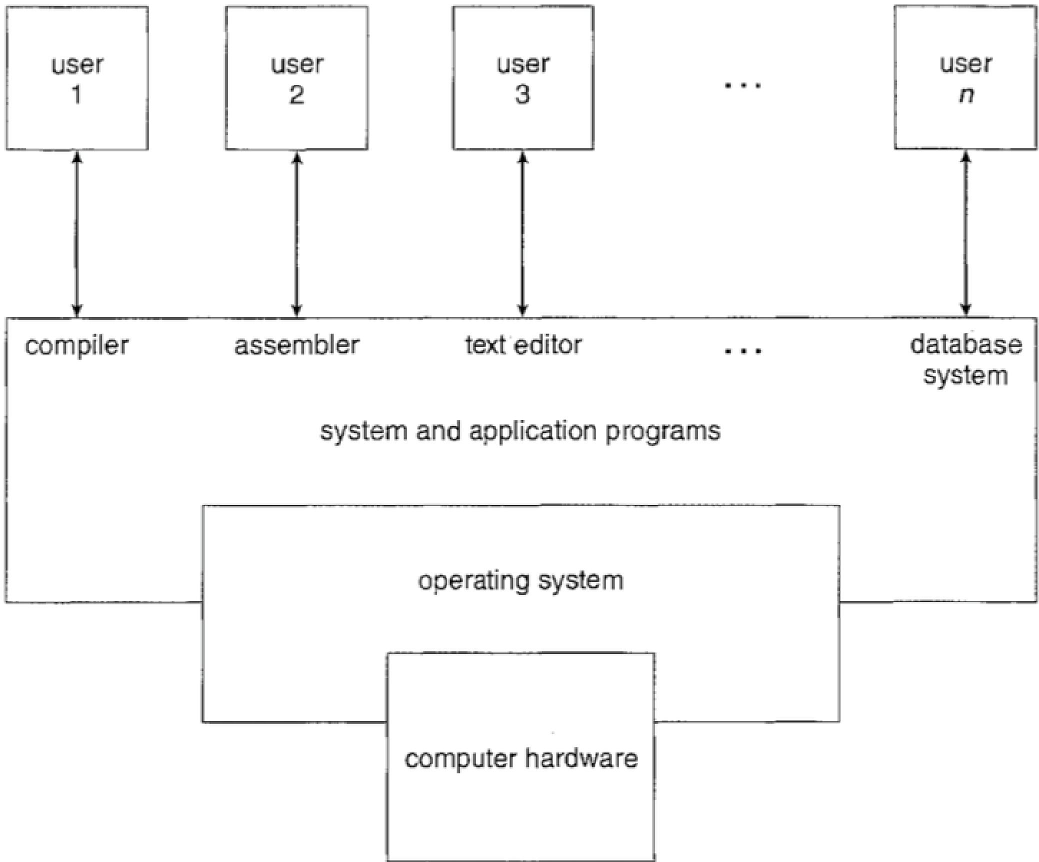


Figure 2.2.2: Operating System Components

As shown above, a computer system can be split into four components. The hardware is composed of the CPU, memory and I/O devices. The operating system acts as the controlling element between the hardware, and all the applications in a computer. The operating system can be split into the user and system perspective.

From a User Perspective

The view of the user changes in accordance with the computer interface being used. For example, the operating system designed for a personal computer would be something aimed towards easy usage, and moderate performance levels. In some cases, users even sit at a terminal connected to a mainframe or minicomputer. For this situation, the operating system is designed to maximize resource utilization. Operating systems for embedded computers are designed to run with no user interruptions. It is evident that certain features of an operating system are enhanced based upon the type of usage.

From a System Perspective

From the view of the actual computer itself, as mentioned earlier, the operating system needs to be the most involved with the actual hardware. The operating system essentially acts as a resource allocator.

Examples of some well-known operating systems are Windows, Linux, AIX and OS/400.

2.3. Number Systems: Decimal, Binary, Hexadecimal, and Addition, Signed Binary

Throughout your elementary, secondary and high school years, the number system that you are accustomed to is known as the Decimal number systems. It consists of many different types of number, such as integers, real number and so on and so forth. However, digital systems consist of only 1's and 0's (ones and zeros), which are primarily referred to as Binary numbers.

Decimal Numbers

Primarily, there are ten decimal numbers or often referred to as digits: 0, 1, 2, 3... 9. These primary decimal numbers, when joined together, form longer decimal numbers. Just as there are ten primary decimal numbers, each column of a decimal number has ten times the weight of the column before it. For example, the decimal number 1000, from right to the left has columns that weigh 1, 10, 100 and 1000. Each column weighs ten times the previous column and therefore, the decimal number system is also referred to as the base 10. What does base 10 mean? For example, a decimal number 1234 has a one's column, a ten's column, a hundred's column and a thousand's column from right to left respectively. Therefore, the number 1234 in base 10 can be represented as $1234_{10} = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$. Usually, the base is indicated by a subscript after the number, as in 1234_{10} for base 10. It is also worth noting that for an N digit decimal number, there are 10^N different numbers. Hence, a three digit decimal number, that is, 10^3, where N = 3, there are 1000 different possibilities that range from 0 to 999. Or, the possibilities range from 0 to $10^N - 1$.

5678_{10}	=	5×10^3	+	6×10^2	+	7×10^1	+	8×10^0
Decimal		Five thousand		Six hundred		7 tens		8 ones

Example of decimal number representation

How do we represent a decimal number that has a fraction part? Consider the example below:

12.34_{10}	=	1×10^1	+	2×10^0	.	3×10^{-1}	+	4×10^{-2}
Decimal		One tens		Two ones		3 tenths (1/10)		4 hundredths (1/100)

In general, any decimal number, A, could be expressed as a power series representation:

$$A = a_N\, 10^N + a_{N-1}\, 10^{N-1} + \ldots + a_0\, 10^0.\ a_{-1}\, 10^{-1} + \ldots + a_{-N+1}\, 10^{-N+1} + a_{-N}\,^{10-N}$$

Everything in front of the decimal point (left of decimal point) is the integer part and everything after the decimal point (right of decimal point) is the fractional part.

Binary Numbers

Binary numbers consists only of two digits, 0 and 1. These digits are more commonly referred to as bits. A bit is either 0 or 1. The term Binary means composed of two pieces or two parts. Binary numbers are base 2, where each column of binary number has twice the weight of the previous column. In binary, the smallest weight possible is 1, which due to $2^0 = 1$. The next number after 1 would be 2, which is twice the value of 1, after which, you would have 4, which is twice the value of 2. From right to the left, the values of binary numbers would be 1, 2, 4, 8, 16, 32, and 64, which represents 2^0, 2^1, 2^2, 2^3, 2^4, 2^5 and 2^6 respectively and so on. As stated above, the bits 0 and 1 could be combined to create numbers of any magnitude. For example, the number 12_{10} could be expressed in binary as 1010_2. In the previous example, a base 10 number was converted into a base 2 number. An N-bit binary number has 2^N different numbers. Hence, a three bit binary number, that is, 2^3, where N = 3, has 8 different possibilities that range from 0 to 7. Or, the possibilities range from 0 to $2^N - 1$.

12_{10}	=	1010_2	=	1×2^3	+	0×2^2	+	1×2^1	+	1×2^0
Decimal		Binary		1 eight		0 four		1 two		0 one

Decimal Number	1 Bit Binary Number	2 Bit Binary Number	3 Bit Binary Number
0	0	00	000
1	1	01	001
2		10	010
3		11	011
4			100
5			101
6			110
7			111

Table 2.3.1: Decimal Numbers and different Binary Numbers

In the above table, it can be noted that the maximum decimal number that could be represented by a 3 bit binary number (111) is 7: $2^2 + 2^1 + 2^0 = 7$ (4+2+1). In the case of a 4 bit binary number, the maximum decimal number that could be represented would be 15. In binary, 1111 could be

expressed as $2^3 + 2^2 + 2^1 + 2^0 = 15$ (8+4+2+1). In other words, $2^4 – 1 = 15$ from the possibilities of $2^N – 1$.

The conversion of a decimal number to a binary number is challenging at first; however, it becomes easy with a little bit of practice. The integer part of a decimal number could be converted to binary using successive division by 2 and the fraction part of a decimal number could be converted to binary using successive multiplication by 2. For Binary numbers, the successive division or multiplication is done by 2; for a different number type, replace the division or multiplication with the appropriate conversion system.

Example: Convert the decimal number 14_{10} to a binary number (base 2).

Solution: As the decimal number only has an integer part (no fractions), successive division is used to determine the binary number. If there is a Binary number B, the solution will be of B_3 B_2 B_1 B_0 type. We proceed from the right to the left when determining the conversion.

Decimal Division	Quotient	Remainder	Binary Value
14 ÷ 2	7	0	B_0 = 0 (remainder)
7 ÷ 2	3	1	B_1 = 1 (remainder)
3 ÷ 2	1	1	B_2 = 1 (remainder)
1 ÷ 2	0	1	B_3 = 1 (remainder)

Therefore, the binary number B = [B_3 B_2 B_1 B_0] = 1110_2. To verify, use the power series summation.

1110_2	=	1 x 2^3	+	1 x 2^2	+	1 x 2^1	+	0 x 2^0	=	14_{10}
Binary		1 eight		1 four		1 two		0 one		Decimal

Example: Convert the fractional number 0.125_{10} to a binary number (base 2).

Solution: As the decimal number only has a fractional part, successive multiplication is used to determine the binary number. If there is a Binary number B, the solution will be of B_{-1} B_{-2} B_{-3} B_{-4} type. We proceed from the left to the right when determining the conversion.

Decimal Multiplication	Product	Integer	Binary Value
0.125 × 2	0.25	0	B_{-1} = 0 (integer)
0.25 × 2	0.50	0	B_{-2} = 0 (integer)
0.50 × 2	1.00	1	B_{-3} = 1 (integer)

Therefore, the binary number B = [B_{-1} B_{-2} B_{-3} B_{-4}] = 0.001_2. To verify, use the power series summation.

0.001_2	=	0×2^{-1}	+	0×2^{-2}	+	1×2^{-3}	=	0.125_{10}
Binary		0 half (1/2)		0 quarter (1/4)		1 eighth (1/8)		Decimal

It is worth mentioning here that the solution for the method of successive multiplication is determined by first placing the decimal point. Then, the decimal number is multiplied by 2; if the integer part of the result is even, then a zero is written after the decimal point of the binary fraction. If the integer part of the result is odd, a one is written after the decimal point of the binary fraction. The decimal number is again multiplied by 2, and the process is repeated as often as necessary, until there is no longer a fraction part in the decimal number.

The process of successive multiplication could get very lengthy, for instance, the conversion of 0.1_{10} to binary is 0.0001100110011001100110011001100110011 … Many fractional conversions generate a lot of digits, because a lot of decimal fractions cannot be represented exactly. Do not worry, you will not be expected to compute such fractions. The above information was shared to give the reader an insight into binary computations.

For instance, if the decimal number 14.125_{10} had to be converted to binary, one would imply the method of successive division and multiplication separately for the integer and fraction part respectively and then bring the binary numbers together, separated by the decimal point. 14.125_{10} = 1110.001_2.

Conversion from decimal to binary could be done in many different ways. With practice, the conversion becomes more natural and faster. Below are some examples with alternate methods.

Example: Convert the decimal number 100_{10} to a binary number (base 2).
Solution: Ideally, determine closest largest 2^N operation that would yield the desired decimal number, where N is a positive whole number (N = 0, 1, 2 …). Usually, working from the left and starting from the largest power of 2 equal to or less than number is preferred. In this case, 2^8 = 256_{10}, which is too high, $2^7 = 128_{10}$, which is also too high. $2^6 = 64_{10}$, which is the closest largest 2^N operation to the decimal number 100_{10} i.e. $64 < 100$. To determine the next number, take the

remainder, which is 100 – 64 = 36 and compare it with the next smaller binary number than 2^6, which is $2^5 = 32$. The sum of 64 + 32 = 96, which is less than 100 i.e. 96 < 100. Therefore, 2^5 is the next binary digit (the next 1 or high). So far, there are 2 binary digits that are high: 1 1 0 0 0 0 0, where the first two ones are 2^6 are 2^5 respectively. Now, 100 – 96 = 4. Is there a binary computation that would result in the decimal number 4 that is of 2^N magnitude? Yes, $2^2 = 4$. In this example, we did not check the magnitude of 2^4 and 2^3 because it was quiet obvious that both 2^3 and 2^4 are higher than 4. This results in $100_{10} = 64 + 32 + 4 = 2^6 + 2^5 + 2^2$. Therefore, in binary notation, the result is $100_{10} = 1100100_2$.

1100100_2	=	1 x 2^6	+	1 x 2^5	+	0 x 2^4	+	0 x 2^3	+	1 x 2^2	+	1 x 2^1	+	0 x 2^0
Binary		1 sixty - four		1 thirty - two		0 sixteen		0 eight		1 four		1 two		0 one

Example of decimal to binary conversion

Example: Convert the binary number 11011_2 to decimal number (base 10).

Solution: Going from the right to the left, count, starting from zero, the number of digits the binary number has. In this example, starting from the right, we have zero, one, two, three and four (OR number of columns – 1 (5-1=4) to account for the 2^0 term). Now, multiply the binary digit, either 1 or 0 with the respective power of 2 term for that column. To obtain, the final decimal number, sum all the terms.

11011_2	=	1 x 2^4	+	1 x 2^3	+	0 x 2^2	+	1 x 2^1	+	1 x 2^0	=	27_{10}
Binary		1 sixteen		1 eight		0 four		1 two		1 one		Decimal

Example of binary to decimal conversion

Octal Numeral System

Students do not commonly know the Octal Numeral System because binary and hexadecimal numbers (next section) are more common. The Octal numeral system is a base 8 system, where each column of octal number has eight times the weight of the previous column. In octal, the smallest weight possible is 1, which due to $8^0 = 1$. The next number after 1 would be 8, which is eight times the value of 1, after which, you would have 64, which is eight times the value of 8. From right to the left, the values of octal numbers would be 1, 8, 64, 512, 4096, 32786, which represents 8^0, 8^1, 8^2, 8^3, 8^4, and 8^5 respectively and so on. For example, the number 100_{10} could be

expressed in octal as 144_8. That is, $1 \times 8^2 + 4 \times 8^1 + 4 \times 8^0$. In the previous example, a base 10 number was converted into a base 8 number. An N-decimal octal number has 8^N different numbers. Hence, a three digit octal number, that is, 8^3, where N = 3, has 512 different possibilities that range from 0 to 511. Or, the possibilities range from 0 to $8^N - 1$.

Binary, Octal and Hexadecimal numbers are closely related because Octal and Hexadecimal are groups of bits. An Octal digit is a group of 3 binary bits ($2^3 = 8$) and a Hexadecimal digit is a group of 4 binary bits ($2^4 = 16$). The process of converting binary to octal or hexadecimal and vice versa will be discussed shortly.

Example: Convert the decimal number 100_{10} to octal number (base 8).

Solution: As the decimal number only has an integer part (no fractions), successive division is used to determine the octal number. If there is an octal number O, the solution will be of $O_3\ O_2\ O_1\ O_0$ type. We proceed from the right to the left when determining the conversion. In the case of Binary number, the division was conducted with the number 2 (Binary), but for octal; the decimal number is divided by 8.

Decimal Division	Quotient	Remainder	Binary Value
$100 \div 8$	12	4	$O_0 = 4$ (remainder)
$14 \div 8$	1	4	$O_1 = 4$ (remainder)
$1 \div 8$	0	1	$O_2 = 1$ (remainder)

Therefore, the Octal number O = [$O_3\ O_2\ O_1\ O_0$] = 144_8. To verify, use the power series summation.

144_8	=	1×8^2	+	4×8^1	+	4×8^0	=	100_{10}
Octal		1 sixty-four		4 eights		4 ones		Decimal

Converting from Octal to Decimal is quiet easy as seen above; power series summations is the easiest way to go about it. Below is another example: Convert 123_8 to a decimal number:

123_8	=	1×8^2	+	2×8^1	+	3×8^0	=	83_{10}
Octal		1 sixty-four		2 eights		3 ones		Decimal

Example: Convert the octal number 123_8 to a binary number (base 2).

Solution: As it was mentioned on the above page, an octal digit is a group of 3 binary bits ($2^3 = 8$). Take each digit of the octal number 123_8 and split it into three columns of binary spaces, then, fill

those binary spaces with either 0 or 1 to equal the magnitude of the octal digit. Please refer to the table below for the explanation:

123_8	=	1 x 8^2	+	2 x 8^1	+	3 x 8^0	=	83_{10}
Octal Number	1		2		3			Decimal
Empty Binary Spaces	_ _ _		_ _ _		_ _ _			
Spaces filled with binary equivalent magnitude	001		010		011		=	001010011_2 Binary
Drop the leading 0s	1010011_2 Binary							
Bring everything together	$123_8 = 1010011_2$ (Octal to Binary Equivalent)							

In the above explanation, the leading zeros were dropped or not needed because they do not add any value to the number.

Example: Convert the binary number 1010011100_2 to an octal number (base 8).

Solution: An octal digit is a group of 3 binary bits ($2^3 = 8$). Starting from the right, start grouping the binary number into groups of three binary digits. If, at the left most point of the binary number, a group of three bits is not possible, than add leading zeros to make the group of three bits. Please refer to the table below for explanation:

Binary Number	Binary number split into groups of 3 bits from right to the left			
1010011100_2	1	010	011	100
Add Leading 0s	001	010	011	100
Convert to Decimal Equivalent	001 = 1 x 2^0 = 1	010 = 1 x 2^1 = 2	011 = 1 x 2^1 + 1 x 2^0 = 2 + 1 = 3	100 = 1 x 2^2 = 4
Write the Decimal # as Octal #	1_8	2_8	3_8	4_8
Bring Everything Together	$1010011100_2 = 1234_8$ (Binary to Octal Equivalent)			

Example: Convert the octal number 12.34_8 to a decimal number (base 10).

Solution: The power series summation method would be the easiest method to go about solving the example.

12.34_8	=	1 x 8^1	+	2 x 8^0	.	3 x 8^{-1}	+	4 x 8^{-2}	=	10.4375_{10}

Octal		One eights		Two ones		3 eighths (1/8)		4 sixty-fourths (1/64)		Decimal

To convert the decimal number above, 10.4375_{10} to an octal number, one would have to use the method of successive division to solve the integer part and the method of successive multiplication to solve the fraction part; then, the answers from the two methods would have to be brought together, separated by the decimal point.

Hexadecimal Numbers

Hexadecimal numbers are a group of four bits of binary numbers. A hexadecimal number represents 16 different possibilities from 0 to 15 as $2^4 = 16$, thus, known as base 16 numbers. They make it easier to write lengthy binary numbers and also reduce error in writing length binary numbers. But, how would one write in base 16 notation? There are only 0 to 9 decimal numbers, but here there are 16 possibilities. Hexadecimal numbers use the digits 0 to 9 and letters A to F, where letter A = 10, B = 11, C = 12, D = 13, E = 14 and F = 15 respectively to represent digits 10 to 15. In hexadecimal, the smallest weight possible is 1, which due to $16^0 = 1$. The next number after 1 would be 16, which is sixteen times the value of 1, after which, you would have 256, which is sixteen times the value of 16. From right to the left, the values of hexadecimal numbers would be 1, 16, 256, 4096, 65536 which represents 16^0, 16^1, 16^2, 16^3, 16^4, and 16^5 respectively and so on. For example, the number 1234_{10} could be expressed in hexadecimal as $4D2_{16}$. That is, $4 \times 16^2 + 13 \times 16^1 + 2 \times 16^0$. In the previous example, a base 10 number was converted into a base 16 number. An N-decimal hexadecimal number has 16^N different numbers. Hence, a three digit hexadecimal number, that is, 16^3, where N = 3, has 4096 different possibilities that range from 0 to 4095. Or, the possibilities range from 0 to $16^N – 1$.

Decimal Number	Hexadecimal Digit	Binary Digits
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000

9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Example: Convert the decimal number 678_{10} to hexadecimal number (base 16).

Solution: As the decimal number only has an integer part (no fractions), successive division is used to determine the hexadecimal number. If there is a hexadecimal number H, the solution will be of H_3 H_2 H_1 H_0 type. We proceed from the right to the left when determining the conversion. In the case of Binary and Octal numbers, the division was conducted with the number 2 (Binary) and number 8 (Octal) respectively. In the case of hexadecimal numbers, you will have to divide the decimal by 16.

Decimal Division	Quotient	Remainder	Binary Value
678 ÷ 16	42	6	H_0 = 6 (remainder)
42 ÷ 16	2	10 = A	H_1 = A (remainder)
2 ÷ 16	0	2	H_2 = 2 (remainder)

Therefore, the hexadecimal number H = [H_3 H_2 H_1 H_0] = $2A6_{16}$. To verify, use the power series summation.

$2A6_{16}$	=	2×16^2	+	10×16^1	+	6×6^0	=	678_{10}
Hex		2 two-hundred-fifty-sixes		10 sixteen		6 ones		Decimal

Converting from hexadecimal to Decimal is quiet easy as seen above; power series summations is the easiest way to go about it. Below is another example: Convert $B3C_{16}$ to a decimal number:

$B3C_{16}$	=	11×16^2	+	3×16^1	+	12×16^0	=	2876_{10}
Hex		11 two-hundred-fifty-sixes		3 sixteen		12 ones		Decimal

Example: Convert the hex number $B3C_{16}$ to a binary number (base 2).

Solution: A hex (hexadecimal) digit is a group of 4 binary bits ($2^4 = 16$). Take each digit of the hex number $B3C_{16}$ and split it into four columns of binary spaces, then, fill those binary spaces

with either 0 or 1 to equal the magnitude of the hex digit. As per hexadecimal notation, B = 11_{16} and C = 12_{16}. Please refer to the table below for the explanation:

$B3C_{16}$	=	11 x 16^2	+	3 x 16^1	+	12 x 16^0	=	2876_{10}
Hex Number	11			3		12		Decimal
Empty Binary Spaces	_ _ _ _			_ _ _ _		_ _ _ _		
Spaces filled with binary equivalent magnitude	1011			0011		1100	=	101100111100_2 Binary
Drop the leading 0s	There are no leading 0s = 101100111100_2Binary							
Bring everything together	$B3C_{16}$ = 101100111100_2 (Hex to Binary Equivalent)							

In the above explanation, there were no leading zeros. If there are leading zeros, then drop the leading zeros because they do not add any value to the number.

Example: Convert the binary number 1010011100_2 to a hex number (base 16).
Solution: A hex digit is a group of 4 binary bits (2^4 = 16). Starting from the right, start grouping the binary number into groups of four binary digits. If, at the left most point of the binary number, a group of four bits is not possible, than add leading zeros to make the group of four bits. Please refer to the table below for explanation:

Binary Number	Binary number split into groups of 3 bits from right to the left		
1010011100_2	10	1001	1100
Add Leading 0s	0010	1001	1100
Convert to Decimal Equivalent	0010 = 1 x 2^1 = 2	1001 = 1 x 2^3 + 1 x 2^0 = 8 + 1 = 9	1100 = 1 x 2^3 + 1 x 2^2 = 8 + 4 = 12
Write the Decimal # as Octal #	2_{16}	9_{16}	12_{16} = C_{16}
Bring Everything Together	1010011100_2 = $29C_{16}$ (Binary to Hexadecimal Equivalent)		

Example: Convert the hex number $A2.F4_{16}$ to a decimal number (base 10).
Solution: The power series summation method would be the easiest method to go about solving the example.

To convert the decimal number above, 162.953125_{10} to a hex number, one would have to use the method of successive division to solve the integer part and the method of successive multiplication

to solve the fraction part; then, the answers from the two methods would have to be brought together, separated by the decimal point.

$A2.F4_{16}$	=	10×16^1	+	2×16^0	.	15×16^{-1}	+	4×16^{-2}	=	162.953125_{10}
Hex		Ten sixteen		Two ones		15 sixteenth (1/16)		4 two-hundred-fifty-sixth (1/256)		Decimal

What are Bytes?

A group of eight bits is called a byte. The term byte is most commonly used for describing the size in computer memories or hard drives; for example, 1 Mb (Mega byte). It represents one of 2^8 numbers. Two hexadecimal numbers store one byte. 1 byte is equal to 8 bits. In most computers, 1 character, e. g. "a", is one byte. Now, the prefix mega stands for 10^6 and the prefix kilo stands for 10^3; a kilobyte is 1024 bytes (2^{10}) and a megabyte is 1024 kilobytes (2^{20}), which is 1,048,576 byes.

What are Nibbles?

A group of four bits is called a nibble. This is similar to a hexadecimal number, a group of four. In fact, one hexadecimal digit stores one nibble. Therefore, a hexadecimal number $4D2_{16}$ is 3 nibbles. The term byte is more commonly used.

Now you know what Nibbles are; think how that relates to the name of the project, "Nibble Knowledge".

What are Most and Least Significant Bits?

When there are a group of bits, for example, 110100_2, the right most bit is called the least significant bit and the left most bit is called the most significant bit. For binary number 110100_2, the least significant bit is 0 (right most bit) and the most significant bit is 1 (left most bit). The abbreviation for least significant bit is LSB and the abbreviation for most significant bit is MSB.

You may be wondering why is it important to know what LSB and MSB are? The most and least significant bits are important in binary addition, overflow determination and many other applications.

What are unsigned binary numbers?

Unsigned binary numbers are numbers that represent only positive quantities, that is, a number that is zero or larger in magnitude. In fact, in binary presentation, there are two types of zeros, positive zero and negative zero (-0_2 and $+0_2$). Information about negative binary numbers will be covered in the section under signed numbers.

So far, the binary numbers that have been considered have been unsigned binary numbers; from 0, 1, 2, 3 to all the way up to positive infinity.

Unsigned Binary Addition

Adding binary numbers is very similar to adding decimal numbers. For many students, binary addition may even be easier than adding decimal numbers. In this text, for simplicity, only two binary will be added at a time. It is also common practice to only add two binary numbers at a time. A computer also usually adds two binary numbers at a time, but it does it very quickly. Computers are very good at computing smaller tasks very quickly, thus, computers do extremely well when solving complex mathematic problems. Computers run programs written by humans to perform specific well-defined tasks; mathematical computations are easily well defined and can be computed by a computer very efficiently.

Adding two binary numbers is very simple because there are only five distinct possibilities. Binary bits are represented in either zero or one (0 or 1), therefore the four out of five possibilities are:

Binary Bit		Binary Bit		Carry Bit	Sum Bit	Together	Decimal Value
0	+	0	=	0	0	00_2	0_{10}
0	+	1	=	0	1	01_2	1_{10}
1	+	0	=	0	1	01_2	1_{10}
1	+	1	=	1	0	10_2	2_{10}

The fifth possibility is that if 1 + 1 is added, then the carry bit will be shifted to the next column on the left, and, if, in the next column, 1 + 1 is to be added, then the summation would be 1 + 1 + 1 = 3_{10}, (carry bit + binary bit + binary bit), which has a carry bit of 1 and a sum bit of 1 as well:

Previous Carry		Binary Bit		Binary Bit		Carry Bit	Sum Bit	Together	Decimal Value
1	+	1	+	1	=	1	1	11_2	3_{10}

While adding binary numbers, move from the right most columns to the left most columns, just like adding decimal numbers.

A carry bit occurs when the summation cannot be written in one binary bit, either zero or one, and the carry bit is placed on top of the next column on the left. For instance, adding $5_{10} + 6_{10} = 11_{10}$, which cannot be fit into a single decimal digit because 11_{10} is larger than 9_{10}, therefore, the digit 1_{10} in the tenth column is carried over to the column on the left. The same concept applies with binary addition.

Example: Add two binary numbers $0111_2 + 0101_2$.
Solution: Starting from the right most column, $1 + 1 = 2_{10} = 10_2$. The carry bit is 1 and the sum bit is 0. For the next column on the left, add the carry bit + binary bit + binary bit: $1 + 1 + 0 = 2_{10} = 10_2$. Again, the carry bit is 1 and the sum bit is 0. For the next column on the left, add the carry bit + binary bit + binary bit: $1 + 1 + 1 = 3_{10} = 11_2$. Lastly, the carry bit is 1 and the sum bit is 1. For the next column on the left, add the carry bit + binary bit + binary bit: $1 + 0 + 0 = 1_{10} = 01_2$. The carry bit for the last addition is zero, and thus has no carry bit. The result is 1100_2. Does the result make sense? Well convert the binary numbers to decimal numbers and verify the result. In decimal, $0111_2 = 7_{10}$ and $0101_2 = 5_{10}$. $7_{10} + 5_{10} = 12_{10} = 1100_2$. The results are the same. Please refer to the table below:

Carry Bit	0	1	1	1	
Binary Bit A		0	1	1	1
Binary Bit B	+	0	1	0	1
Sum		1	1	0	0

The same summation could also be displayed as:

```
0111        Carry Bit
  0111      Binary # A
+0101       Binary # B
----------
 1100       Sum
```

In the above example, we added two 4-bit numbers, or you may say we added two Nibbles together. We know that the maximum value a nibble can hold is $1111_2 = 15_{10}$, but what is the summation results a solution that is larger than 15_{10} or 1111_2? In such cases, the addition has overflown or reaches an overflow condition. All it means is that 4-bits are not enough to represent the results and a 5th bit is needed to write the correct solution. If the number is limited to 4-bits, than the solution is incorrect. For unsigned binary addition, an overflow condition is determined if the carry bit of the most significant bit (MSB) is 1. Consider the example below:

```
Overflow ->  1111        Carry Bit
              0111       Binary # A
             +1011       Binary # B
             ----------
              0010       Sum
```

There is an overflow condition above because the carry bit of the most significant bit is 1, which indicates that the number is too large to be represented in 4-bits. To verify, add the numbers in decimal format. 0111_2 = 710 and $1011_2 = 11_{10}$; the addition of $7_{10} + 11_{10} = 18_{10}$, but the result obtained from the binary addition is $0010_2 = 2_{10}$. It is quite obvious that solution is incorrect and there is an overflow. If, a fifth bit were allowed, then the result would be 10010_2, which equals 18_{10} and would be the correct solution. Note again, that only 4-bits were allowed above, which resulted in an overflow condition and an incorrect result.

What are signed binary numbers?

It is quiet useful to be able to represent both positive and negative numbers. There are two common methods used to represent negative binary numbers: sign/magnitude method and two's complement. For a signed number, the size of the number must always be states, for example, a 3-bit number or a 6-bit number, etc.

The two's complement method is preferred over sign/magnitude because it makes binary addition easier and the result makes sense.

In the sign/magnitude method, the most significant bit is used to represent the sign of the number, either positive or negative, where the sign bit of 0 indicates positive and the sign bit of 1 indicates negative number.

Example: Convert 12_{10} to -12_{10} as 5-bit sign/magnitude number.

Solution: First, first the magnitude of the decimal number in binary and then either add 0 or 1 to indicate the sign. $12_{10} = 1100_2$ or 01100_2 to be more precise, however, leading zeros are usually dropped. To convert to -12_{10}, simple change the zero in the most significant bit to 1. This results in $11100_2 = -12_{10}$.

There are a few downsides to the sign/magnitude system because there are two ways to show 0 in the sign/magnitude system: -0 and +0. Both of these show zero and this causes problems because there are two representations of the same number.

Also, adding sign/magnitude numbers does not work. For instance, adding 12_{10} + -12_{10} should result in zero, or in 5-bits, 00000. But does it?

```
111                Carry Bit
  01100            Binary # A
+11100             Binary # B
----------
 101000            Sum
```

Clearly, the solution is incorrect. Decimal numbers 12_{10} + -12_{10}, do not equal to 101000_2, which is equal to 40_{10}. For this reason, two's complement numbers are preferred.

Two's Complement Numbers

Two's complement numbers are the most commonly used numbers in the binary world to represent signed numbers. The benefit of using two's complement is that mathematical computation are similar to unsigned numbers. There is only one way to write zero in two's complement number, which is 0_2, and addition works exactly the same way as unsigned binary addition. A positive number is presented exactly the same way as unsigned binary number and a negative number has a 1 in the most significant bit position to show the negative number. For a given binary number size, the most positive number would have zero as the most significant bit and ones everywhere else. For instance, a 5-bit number to have maximum positive value, the number would look like 01111_2 or $2^{N-1} - 1$, where N is the number size (5 in this case). And, for a given binary number size, the most negative number would have one as the most significant bit and zeros everywhere

else. For instance, a 5-bit number to have maximum negative value, the number would look like 10000_2 or -2^{N-1}, where N is the number size (5 in this case).

A quick method to find the two's complement of negative decimal number, one could employ a quick calculation of $2^N - |A|$, where N is the size of the binary number and |A| is the magnitude of the negative decimal number. For example, to convert -12_{10} to a 5-bit two's complement number, simply calculate $2^N - |A|$, where N = 5 and $|A| = 12_{10}$, which results in $2^5 - |12| = 20_{10} = 10100_2$.

Converting a negative decimal number to two's complement representation requires three quick steps: first, find the binary representation of the magnitude of the negative number, secondly, invert each bit of the number, that is, zeros become ones and ones becomes zeros; this is also known as taking the complement of the number, hence the term two's complement. Lastly, add 1 to the complemented (inverted) number. Following the three steps above results in the two's complement representation of a negative decimal number.

Example: Convert the negative decimal number -12_{10} to a 5-bit two's complement number.
Solution: To find the magnitude of the decimal number in binary, one may use successive division or simply power series presentation if the number is small and simple. The number -12_{10} is simple and the magnitude may be converted to binary quiet easily. However, let's review the method of successive division:

Decimal Division	Quotient	Remainder	Binary Value
12 ÷ 2	6	0	$B_0 = 0$ (remainder)
6 ÷ 2	3	0	$B_1 = 0$ (remainder)
3 ÷ 2	1	1	$B_2 = 1$ (remainder)
1 ÷ 2	0	1	$B_3 = 1$ (remainder)

Therefore, $12_{10} = 01100_2$ in binary 5-bits. This concludes step number one. Now, step number two is to complement (invert) the each bits, which results in 10011_2, here, 0 became 1 (0 -> 1)and 1 became 0 (1 -> 0). Lastly, add 1 to the inverted number:

```
  0011            Carry Bit
  10011           Inverted #
+     1           Add 1
----------
 10100            Sum
```

The result of the summation is the representation of the negative decimal number -12_{10} in to binary two's complement number, which is 10100_2. It may be noted that the result matches the solution derived from solving $2^N - |A|$ above.

Example: Convert the positive decimal number 15_{10} to a 5-bit two's complement number.
Solution: The solution for this is very simply because it is a positive number. In two's complement method, a positive number is presented in the exactly the same manner as an unsigned binary number. Again, let's employ the method of successive division for practice:

Decimal Division	Quotient	Remainder	Binary Value
15 ÷ 2	7	1	B_0 = 1 (remainder)
7 ÷ 2	3	1	B_1 = 1 (remainder)
3 ÷ 2	1	1	B_2 = 1 (remainder)
1 ÷ 2	0	1	B_3 = 1 (remainder)

The result is $15_{10} = 01111_2$. Steps two and three are not required because this is a positive number.

Two's Complement Addition

Adding two's complement numbers is the same as adding unsigned binary numbers. However, the overflow conditions will be different and will be discussed later in the section. Adding two binary numbers is very simple because there are only five distinct possibilities, which were described in the unsigned binary addition section.

The only extra work involved in adding two complement's numbers is that if there is a negative number, it has to be converted to its negative representation in two's complement using the three steps mention above.

Example: Add -12_{10} and 15_{10} as 5-bit two's complement numbers.

Solution: From the previous exercise, it is known that the binary two's complement equivalent of $-12_{10} = 10100_2$ and $15_{10} = 01111_2$. Now, all that is left to do is add:

```
111              Carry Bit
  10100          -12₁₀
+ 01111          15₁₀
----------
  00011          Sum
```

The 5-bit solution of the addition is $00011_2 = 3_{10}$. Does that add up? $-12_{10} + 15_{10} = 3_{10}$. It is important to note that the last carry bit or the most significant carry bit of 1 does not indicate an overflow. In two complement's addition, ignore the last carry bit as it may or may not indicate whether an overflow condition has occurred or not.

In two's complement addition, overflow never occurs when adding two numbers of opposite sign or adding a positive and negative number. A method to recognize whether an overflow has occurred in two's complement addition is to look at the last two carry bits. If the last two carry bits are the same, that is, if the last two carry bits are either '00' or '11', than overflow has not occurred. If the last two carry bit are opposite, that is, if the last two carry bits are either '01' or '10', than overflow has occurred. In the example above, overflow did not occur because two numbers with opposite signs were added and the last two carry bits were the same ('11').

Example: Add -10_{10} and -12_{10} as 5-bit two's complement numbers. Also, check for overflow and explain your reasoning.

Solution: Begin by converting the two negative decimal numbers into two's complement numbers. Find the magnitude of the numbers in binary first, the convert to two's complement presentation.

Decimal Division	Quotient	Remainder	Binary Value
$10 \div 2$	5	0	$B_0 = 0$ (remainder)
$5 \div 2$	2	1	$B_1 = 1$ (remainder)
$2 \div 2$	1	0	$B_2 = 0$ (remainder)
$1 \div 2$	0	1	$B_3 = 1$ (remainder)

$10_{10} = 01010_2$ in 5-bits binary. Now, invert (complement) the number: 10101_2 and lastly, add 1:

```
00001        Carry Bit
  10101      Inverted #
+     1      Add 1
----------
 10110       Sum
```

Therefore, $-10_{10} = 10110_2$ in two's complement binary. Repeat the steps for -12_{10}.

Decimal Division	Quotient	Remainder	Binary Value
12 ÷ 2	6	0	B_0 = 0 (remainder)
6 ÷ 2	3	0	B_1 = 0 (remainder)
3 ÷ 2	1	1	B_2 = 1 (remainder)
1 ÷ 2	0	1	B_3 = 1 (remainder)

Therefore, $12_{10} = 01100_2$ in binary 5-bits. This concludes step number one. Now, step number two is to complement (invert) the each bits, which results in 10011_2. Lastly, add 1 to the inverted number:

```
  0011       Carry Bit
  10011      Inverted #
+     1      Add 1
----------
 10100       Sum
```

Therefore, $-12_{10} = 10100_2$ in two's complement binary. Now, add the 2 two's complement numbers together:

```
10100        Carry Bit
  10110      -10₁₀
+ 10100      -12₁₀
----------
  01010      Sum
```

The 5-bit result of the addition $-10_{10} + -12_{10} = 01010_2$. Does the solution make sense? The solution does not make sense because two negative numbers were added and the result is a positive number (MSB = 0). Of course, that is not correct. The solution is a positive number 01010_2, if that is converted to decimal, it equals 10_{10}. The addition of $-10_{10} + -12_{10} \neq 10_{10}$. There is an overflow condition here because the last two carry bit are opposite ('10').

If the computation had been conducted using six bits, the result would have been $101010_2 = -22_{10}$, which is the correct solution. How does $101010_2 = -22_{10}$? Apply the two's complement conversation method; invert the bits and add 1.

101010_2 inverted (complemented) = 010101_2. Now add 1.

101010	Carry Bit
010101	Inverted #
+ 1	Add 1

010110	Sum

0101102 = $0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 16_{10} + 4_{10} + 2_{10} = 22_{10}$. In the 6-bit solution the most significant bit of 101010_2 is 1, which means it is a negative number, this add the negative sign to the magnitude to give -22_{10}. Therefore, $101010_2 = -22_{10}$.

2.4. Languages: Machine, Assembly and Instruction – A High Level Introduction

We use languages to communicate with each other, likewise, computers also use their version of language to communicate with its architecture. When programming, one is working with the software and hardware level of the computer. The programmer is working with the computer's architecture. To understand the architecture of a computer, one has to study the language a computer understands, how it executes its tasks and where it stores the information. The language is the instruction set and the operators and operands are the ALU, registers and memory of the computer. ALU, registers and memory are discussed in different chapters.

It is wise to study the computer's language (Instruction Set) to understand the computer's architecture. To describe it as an analogy, the vocabulary is the instruction set and the words are the instructions. Each and every single program that is running on the computer uses the same instructions and the instruction set. A computer is faster than humans in the sense that it is capable of conduction very simple tasks very quickly. This holds true even when complex programs are being run in the computer. For instance, applications like Microsoft Word, Excel or PowerPoint, which are some programs that most people familiar with get compiled down to simple instructions via the use of macro and low-level assemblers. Most common instructions in a computer are add, subtract, jump and no operation. It was learned in section 2.3 that the computer is binary, it only understands 1s and 0s. Instructions that encoded as binary numbers are called Machine Language. Computers use machine language, which consists of many 1s and 0s to decipher what to execute. Imagine reading hundreds to thousands of 1s and 0s all the time. That is a very tedious task and humans get very tired of reading these binary numbers to figure out what is happening or to direct the computer to execute a specific task. Thus, the instructions to the computer are represented symbolically, using what is called Assembly Language. The complier in the translating software converts the symbols to binary numbers that the computer understands.

Different architectures have different assembly language. Each type of CPU will have its own machine and assembly language. However, most architectures have common instructions such as add, subtract and jump, which makes it easy to understand other architectures as well.

Assembly Language

In simple terms, assembly language is the readable version of machine language that humans can understand. The assembly language is composed of instructions that specify the operation to perform and where to conduct that operation. An Assembler, which is program in the computer that takes the assembly language and converts it into machine code that the computer will understand and execute. The Nibble Knowledge Computer uses an assembly language that was developed only for the Nibble Knowledge CPU.

Instructions

Instructions are like the words used to describe an operation. For example, if you wanted to subtract two number, say X – Y = Z, then in English, you would say, Z equals the number that is a result of Y being subtracted from X. In assembly language, the translation is similar.

Subtraction of X – Y = Z in Assembly Language is:

```
LOD   addressY      ; Put Y into A
NND   n15           ; Negate Y
ADD   n1            ; Add 1 to Negated Y (Finished Two's Compliment)
ADD   addressX      ; Add X to Negated Y (X-Y)
```

It seems more complicated, but with practice, it is very easy to understand. One may read the assembly into English equivalent as "put Y into memory location A, negate y from it, then compute the binary subtraction by adding 1 to the negated Y, which is doing the Two's Complement (discussed in section 2.3 above), and then add X to the negated Y value. Assembly language will be discussed in more details in chapter 5.

2.5 Memory – A High Level Introduction

In today's day and age, there are so many types of electronic memory that have integrated into our lives. Almost every electronic device used today has some sort of memory. For example, cell phones, computers, PDAs, car stereo systems, TVs, gaming consoles and so many more. This section will focus on what computer memory looks like and what it does.

Computer Memory

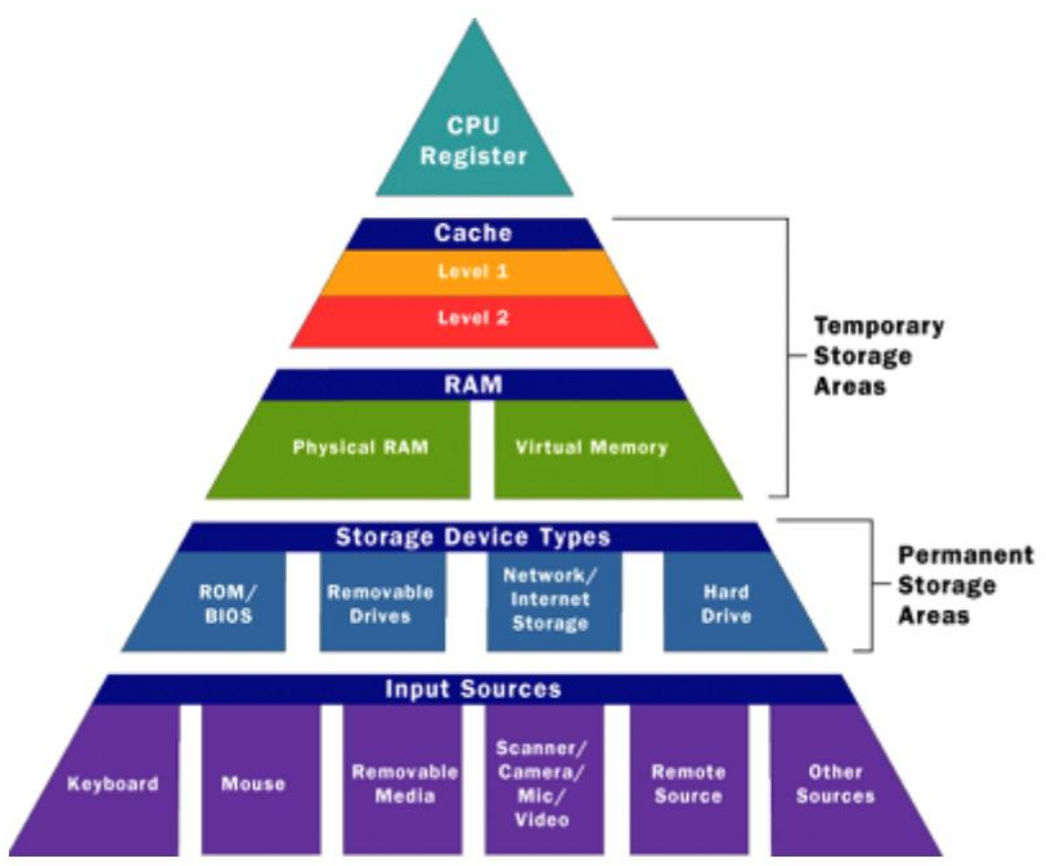


Figure 2.5.1: Memory Accessed by the CPU

This diagram shows that memory is accessed by the CPU in accordance with a distinct hierarchy. Computer speeds were drastically improved when Temporary Storage areas were introduced to the memory hierarchy. If a CPU has to access the hard drive every time a piece of data is needed, the computer would function very slowly. Data usually comes from an input source, or a storage device, and gets stored in the RAM. The CPU then stores frequently accessed data in the cache, and special instructions are maintained in the register. In general, from the time a computer is

turned on, the CPU is constantly accessing memory. Since it is so frequently accessed, an overview of a modern computer memory is show in the figure below.

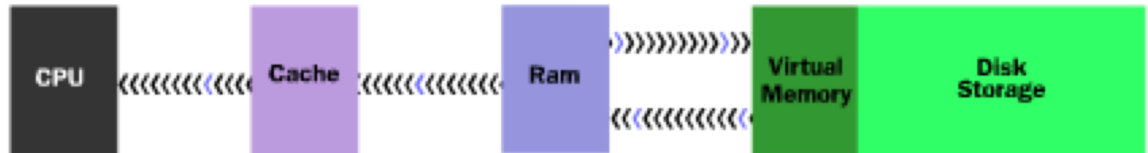


Figure 2.5.2: Computer's Cache, RAM and Virtual Memory and Disk Storage

The figure shows that a typical computer has level 1 and 2 caches, RAM, virtual memory, and hard disk. It makes you wonder why so many tiers of storage is required, doesn't it? The answer is simple! The speed and power expected from computers today means CPUs need fast and easy access to any amount of data to be able to maximize performance.

<u>What does data in memory look like?</u>

Computers represent all data in the form of numbers. For that reason, the various tiers of computer memory need to be able to store and retrieve (read and write) numbers. Computer memory is made up of memory cells, and each cell contains of one number. So, a memory cell can essentially be visualized as a box that contains a single number.

1st Dig.	Last Digit 0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0
...					...					

Figure 2.5.3: Memory Cell Structure

The data in a memory cell can be accessed/read infinite number of times. However, once the data in a memory cell is modified or deleted, the old data will no longer be accessible.

A typical computer has millions of memory cells, and every memory cell has a distinct "name," which is commonly known as the address. Different notations are adapted when talking about a memory cell address, and the data in a memory cell. This will become clearer once assembly language is introduced.

It is often confusing to visualize how letters, and special characters are stored as numbers. In general, CPUs use what's called the ASCII character set table.

```
+--------+-------------------------------------------------+
| First  |                   Last Digit                    |
| Digits |  0    1    2    3    4    5    6    7    8    9  |
+--------+-------------------------------------------------+
|    x   | NUL  SOH  STX  ETX  EOT  ENQ  ACK  BEL  BS   HT  |
|   1x   | LF   VT   NP   CR   SO   SI   DLE  DC1  DC2  DC3 |
|   2x   | DC4  NAK  SYN  ETB  CAN  EM   SUB  ESC  FS   GS  |
|   3x   | RS   US   ` '  `!'  `"'  `#'  `$'  `%'  `&'  `'' |
|   4x   | `('  `)'  `*'  `+'  `,'  `-'  `.'  `/'  `0'  `1' |
|   5x   | `2'  `3'  `4'  `5'  `6'  `7'  `8'  `9'  `:'  `;' |
|   6x   | `<'  `='  `>'  `?'  `@'  `A'  `B'  `C'  `D'  `E' |
|   7x   | `F'  `G'  `H'  `I'  `H'  `K'  `L'  `M'  `N'  `O' |
|   8x   | `P'  `Q'  `R'  `S'  `T'  `U'  `V'  `W'  `X'  `Y' |
|   9x   | `Z'  `['  `\'  `]'  `^'  `_'  ``'  `a'  `b'  `c' |
|  10x   | `d'  `e'  `f'  `g'  `h'  `i'  `j'  `k'  `l'  `m' |
|  11x   | `n'  `o'  `p'  `q'  `r'  `s'  `t'  `u'  `v'  `w' |
|  12x   | `x'  `y'  `z'  `{'  `|'  `}'  `~'  DEL           |
+--------+-------------------------------------------------+
```

Figure 2.5.4: ASCII Character Set table

Every character is converted to a number using the table shown in the figure above. Similarly, text and sound is converted into a sequence of numbers. Images are also converted to a sequence of numbers, but each number corresponds to a pixel, which represents a small part of a picture. So, a sequence of these numbers would display an entire picture.

3. Combinational and Sequential Logic

3.1. Logic Gates

3.1.1 Introduction

At the simplest level, a logic gate is a device that executes a logical operation on two or more inputs to produce one logical output. However, in order to have an understanding of logic gates one must first have an understanding these logical operations and the arithmetic behind them. These logical operations were invented the mid 1800's by George Boole then refined and perfected the late 19th century until it reached what we know today as Boolean Algebra.

In Boolean algebra, there are only two different values either True or False represented in binary as "1" and "0" respectively. These values are used to execute logical operations of which the solutions are organized into what is called a Truth Table, which will show every possible combination of inputs and the given output of that operation. An example of a truth table is shown in the table below.

Input A	Input B	Output Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.1.1: An example of a truth table

Logic Gate: A device that executes a logical operation, on two or more inputs to produce one logical output.

3.1.2 Basic Logic Gates

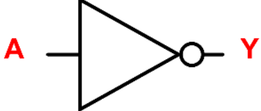
In Boolean algebra, there are three basic operations or "gates" that can be used either by themselves or in complex Boolean expressions to get a desired truth table. However, let's start with an overview of each of the basic gates, their symbols, and how they operate.

Symbols	Meaning
*	AND
+	OR
‘	NOT
$\oplus$	XOR

Table 3.1.2: Boolean symbols and their meanings

NOT Gate

The NOT gate is the most simple logic gate to understand. It takes a single input and simply inverts or switches the value of the input to the opposite value. For example, if the input of the gate is a "1" the output of the gate is now "0".



Boolean Expression:
Y = A'

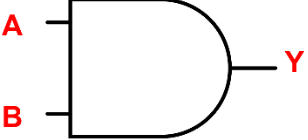
Figure 3.1.1: NOT Gate Symbol and Equation

Input A	Output Y
0	1
1	0

Table 3.1.3: NOT Gate Truth Table

AND Gate

The AND gate takes two or more inputs and compares their values. If all of the inputs are true the output will also be true otherwise, the result is false. Therefore, the output of the gate can only be 1 if all of the inputs are 1.



Boolean Expression:
Y = A * B

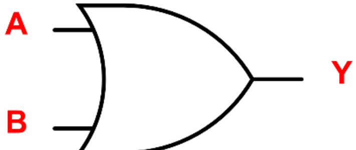
Figure 3.1.2: AND Gate Symbol and Equation

Input A	Input B	Output Y
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.1.4: AND Gate Truth Table

OR Gate

The OR gate also takes two or more inputs and compares their values. Unlike the AND gate however, the output of an OR gate is true if any of the inputs are true no matter what the remaining input values are. In other words if any one of the inputs is a 1, the output is also 1.



Boolean Expression:
Y = A + B

Figure 3.1.3: OR Gate Symbol and Equation

Input A	Input B	Output Y
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.1.5: OR Gate Truth Table

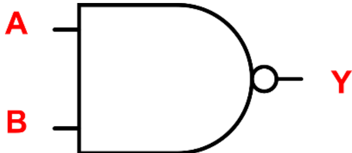
3.1.3 Complex Basic Logic Gates

While slightly more complex than the first three gates, these four gates are still considered to be part of the basic building blocks that make up Boolean algebra.

NAND and NOR Gates

These two gates are very important, as you will see in a later section. These gates are made by taking an ordinary AND or OR gate and inverting the output with a NOT gate to get an inverted truth table. Although locally you'd assume these gates more complex than AND's, OR's and NOT's the truth is actually the opposite in hardware. NAND's and NOR's in hardware are in fact simpler and are considered the building blocks of all other gates.

NAND Gate: Created by taking an AND gate and inverting the output with a NOT gate



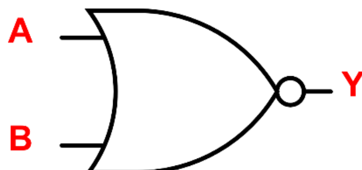
Boolean Expression:
Y = A' * B'

Figure 3.1.4: NAND Gate Symbol and Equation

Input A	Input B	Output Y
0	0	1
0	1	1
1	0	1
1	1	0

Table 3.1.6: NAND Gate Truth Table

NOR Gate: Created by taking an OR gate and inverting the output with a NOT gate



Boolean Expression:
Y = A' + B'

Figure 3.1.5: NOR Gate Symbol and Equation

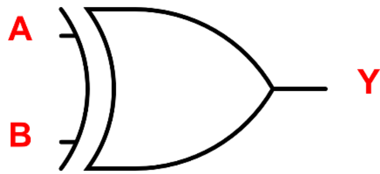
Input A	Input B	Output Y
0	0	1
0	1	0
1	0	0
1	1	0

Table 3.1.7: NAND Gate Truth Table

Exclusive OR and Exclusive NOR

Last, but definitely not least, of the basic logic gates are the exclusive OR and exclusive NOR, shortened to XOR and XNOR respectively. The XOR gate takes two or more inputs and compares their values. An XOR gate's output is similar to a normal OR except for the output becomes false if all of the inputs are true at the same time. The XNOR gate just adds a NOT gate on top of this causing the gate to only be true if the inputs are either all 0's or all 1's.

XOR Gate: Similar to an OR gate except the output is false if all the inputs are true.



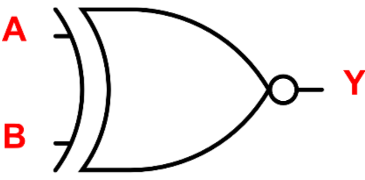
Boolean Expression:
Y = (A * B') + (A' * B)

Figure 3.1.6: XOR Gate Symbol and Equation

Input A	Input B	Output Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.1.8: XOR Gate Truth Table

XNOR Gate: Created by taking an XOR gate and adding a NOT gate. Output is true if inputs are either all true or all false



Boolean Expression:
Y = (A * B) + (A' * B')

Figure 3.1.7: XNOR Gate Symbol and Equation

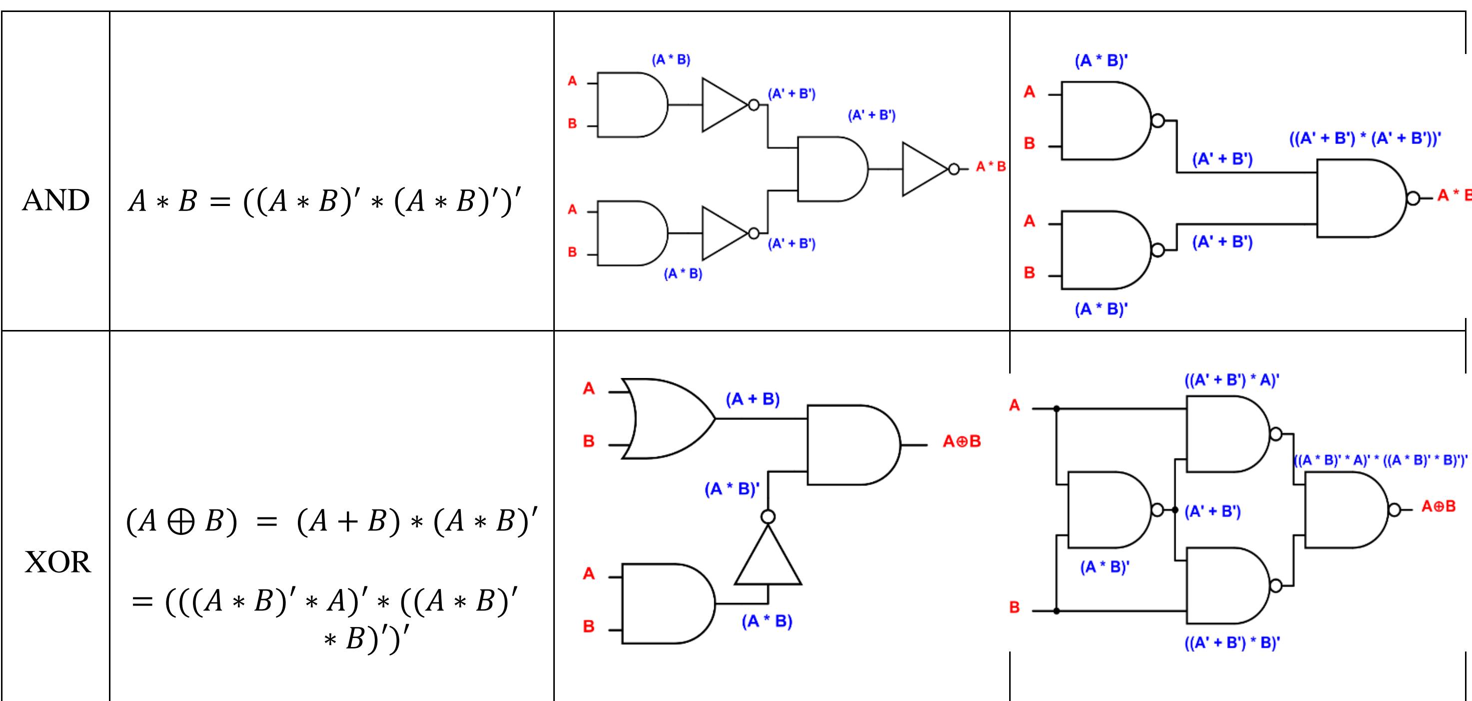
Input A	Input B	Output Y
0	0	1
0	1	0
1	0	0
1	1	1

Table 3.1.9: XNOR Gate Truth Table

3.1.4 Universal Gates

NAND gates and NOR gates are known as **Universal Gates**. This means that using only NAND's, for example, you can achieve the functionality of any other gate. NAND's are the preferred universal gates, as they require less area to implement and have a shorter propagation delay

Logic	Expression	Expression Implementation	NAND Implementation
NOT	$A' = (A * A)'$		
OR	$A + B = ((A * A)' * (B * B)')'$		



AND	$A * B = ((A * B)' * (A * B)')'$		
XOR	$(A \oplus B) = (A + B) * (A * B)'$ $= (((A * B)' * A)' * ((A * B)' * B)')'$		

Table 3.1.10: Universal NAND Gate Interpretations

The Nibble Knowledge CPU, for example, makes use of this and has only a NAND gate to execute all logical operations.

3.2. Boolean Algebra and Equations

3.2.1 Boolean Algebra

Just like the algebra that students have learned helps to simplify mathematical expression, Boolean algebra helps to simplify Boolean expressions. The term Boolean originated from the work of George Boole who introduced binary variables and the fundamental logic operations of Not, OR and AND. Boolean means either true or false, or 1 or 0. Just like mathematics, Boolean algebra is based on postulates that are assumed to be correct. A postulate is also known as Axiom. An axiom is defined as a statement that is highly evident and well established that it is accepted as being true universally without controversy. For example, 1 x 0 = 0, it is accepted as being true. In the case of Axioms, Axioms might be not provable, but are accepted as being correct. Inherently, Boolean axioms are based on the three most basic Boolean expressions: NOT, AND & OR. Below are the eight most common axioms:

Number	Name	Axiom/Postulate
1	Binary	$A = 0$ if $A \neq 1$ or $A = 1$ if $A \neq 0$
2	NOT	$0' = 1$ or $1' = 0$
3	AND	$0 \circ 0 = 0$
4	AND	$1 \circ 1 = 1$
5	AND	$0 \circ 1 = 1 \circ 0 = 0$
6	OR	$0 + 0 = 0$
7	OR	$1 + 1 = 1$
8	OR	$1 + 0 = 0 + 1 = 1$

Table 3.2.1: Boolean Equation Postulates

The first axiom, which is the binary axiom, is quiet simple to explain; a variable is 1 (high) if it is not 0 (low) and vice versa. The second axiom states that the NOT of 1 (high) is 0 (low) and vice versa. It is called NOT-ing a variable or taking the compliment or inverse. Axioms 3, 4 and 5 are exactly the same as normal mathematical product expressions: 0 x 0 = 0, 1 x 1 = 1 and 0 x 1 or 1 x 0 = 0. Axioms 6, 7 and 8 all relate to the OR operation. It is easiest to understand these Axioms if written out as a phrase. If there are two inputs, A and B, then if A is zero or B is zero, then the output will also be zero (0 + 0 = 0). If A is one or B is one, the output would still be one.

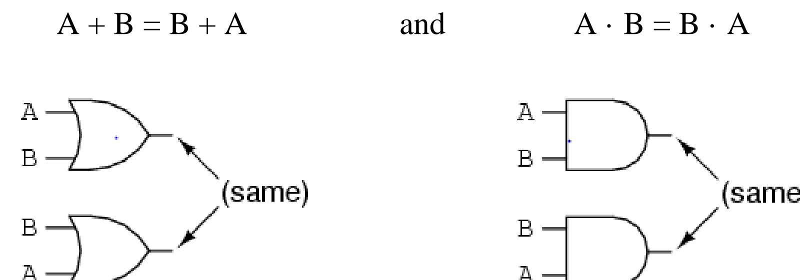
Most Common Properties of Boolean algebra:

1. (Closure)

$$A + B \in S \quad \text{and} \quad A \cdot B \in S$$

2. (Commutativity)

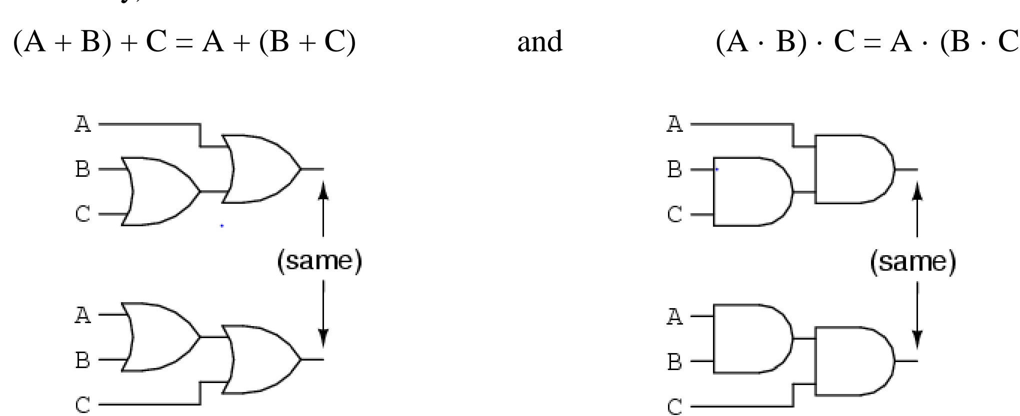
$$A + B = B + A \quad \text{and} \quad A \cdot B = B \cdot A$$



The commutative property says that we can reverse the order of variables that are either added together or multiplied together without changing the truth of the expression. Binary operations AND and OR may be applied left to right or right to left.

3. (Associativity)

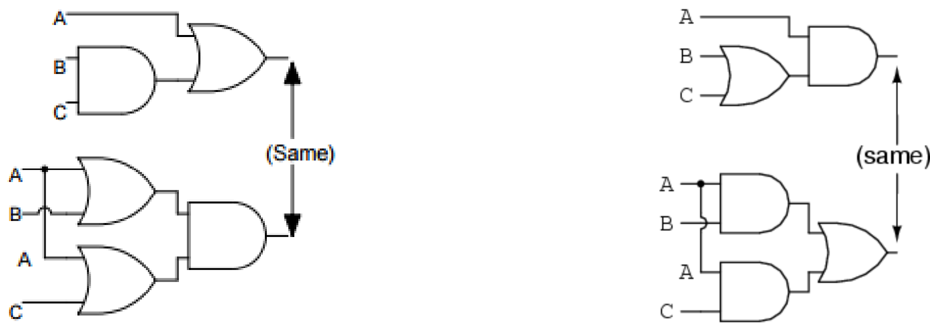
$$(A + B) + C = A + (B + C) \quad \text{and} \quad (A \cdot B) \cdot C = A \cdot (B \cdot C)$$



The associative property says that we can associate groups of added or multiplied variables together with parentheses without altering the truth of the equations. Given three Boolean variables, they may be AND or, OR may be applied right to left or left to right.

4. (Distributive)

$$A + (B \cdot C) = (A + B) \cdot (A + C) \quad \text{and} \quad A \cdot (B + C) = (A \cdot B) + (A \cdot B)$$



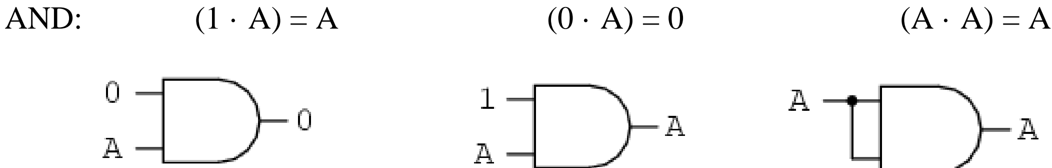
The distributive property that illustrating how to expand a Boolean expression formed by the product of a sum, and in reverse shows us how terms may be factored out of Boolean sums-of-products. Given three Boolean variables, the first AND the result of the second OR the third is the same as the first AND the second OR the first AND the third. Also, the first OR the result of second AND the third is the same as the first OR the second AND the result of the first OR the third.

5. (Identity)

OR: $A + 0 = A$ $A + 1 = 1$ $A + A = A$

The first identity property says that the sum of anything and zero is the same as the original "anything." Second says no matter what the value of A, the sum of A and 1 will always be 1. Third property says adding A and A together which is the same as connecting both inputs of an OR gate to each other and activating them with the same signal. We cannot say that A + A = 2A, since a quantity of "2" has no meaning in Boolean algebra, only 1 and 0.

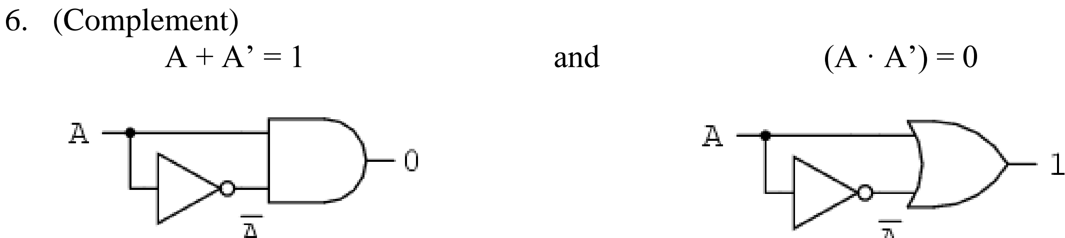
AND: $(1 \cdot A) = A$ $(0 \cdot A) = 0$ $(A \cdot A) = A$



The first identity property says that the product of anything and 1 is the same as the original "anything." Second says no matter what the value of A, the product of A and 0 will always be 0. Third property says adding A and A together which is the same as connecting both inputs of an OR gate to each other and activating them with the same signal. We cannot say that A x A = A2, since the concept of "square" implies a quantity of 2, which has no meaning in Boolean algebra.

6. (Complement)

$A + A' = 1$ and $(A \cdot A') = 0$



The complement property says that there must be one "1" value between any variable and its complement, and since the sum of any Boolean quantity and 1 is 1, the sum of a variable and its complement must be 1. Second property says that there must be one "0" value between any variable and its complement, and since the product of any Boolean quantity and 0 is 0, the product of a variable and its complement must be 0.

3.2.2 De Morgan's Law

Augustus De Morgan was the 19th-century British mathematician who invented what is known as De Morgan's laws.1

These rules are expressed as follows:

- The negation of a conjunction is the disjunction of the negations.
- The negation of a disjunction is the conjunction of the negations.

Or in algebra can be written as:

not(A and B) = (not A) or (not B) **(A*B)' = A'+B'**

- not (A or B) = (not A) and (not B) **(A+B)' = A'*B'**

This is often used to simplify expressions or to translate a theoretical circuit into a hardware circuit.

3.2.3 Boolean Equations

The term Boolean originated from the work of George Boole who introduced binary variables and the fundamental logic operations of Not, OR and AND. Boolean means either true or false, or 1 or 0. Boolean equations can be of two forms: Sum of Products form and Products of Sum form. Before the details of Boolean equations, some basic definitions have to be discussed. For example, a variable A (this is not the same A in hexadecimal) has an inverse, that has the notation of A'; it is the complement of the variable. A variable by itself is called a literal. A literal could be a variable

or a complement. Usually, a variable A is in it's true form and the inverse is the complementary form.

In Boolean algebra, to AND or "ANDing" means taking the product of one or more literals. For instance, the product of variables A and B, which can be written as AB is the "ANDing" of A and B. To OR, or "ORing" means taking the sum of one of more literals. Taking the sum of A+B is the same as "ORing" them.

When making calculations in decimal numbers, one considers the order of operation to determine which part of the expression to calculate first. Likewise, in the binary computations, one must also consider the order of operation. The order of operation plays a significant role in understanding Boolean equations. In Boolean equations, NOT has the highest precedence, followed by AND, then OR. It is similar to decimation computation, where products are performed before summations. For instance, if there is an expression, Y = AB + CD, then the expression could be read as Y equals (A AND B) OR (C AND D).

Sum of Products

The term sum-of-products itself explains the meaning: taking the summation (OR) of products (ANDs). Recalling from the previous explanation, summation is the OR function and product is the AND function in Boolean equations. If, there is an expression, where there are more than one product terms that result in the Boolean expression becoming TRUE (of high or 1), than those products can be summed to give one Boolean equation. For instance, there is a circuit that has 2 inputs, A and B, and 1 output Y. If the output of the circuit is TRUE when the following conditions are met: A is logic high and B is logic low (B'), or A is logic high and B is logic high, in other words, AB' or AB, the output could be written as a Boolean equation as Y = AB' + AB. In this case, one if summing the products or sum-of-products. Y = AB' + AB could be written as Y equal (A and B') OR (A AND B). The output is logic high in these two conditions.

Products of Sums

The Term products-of-sum itself explains the meaning: taking the product (AND) of summations (ORs). However, there is a twist here. In the sum of products form, one has to sum product terms that are TRUE (logic high or logic 1), however, the products-of-sums considers terms that result in the output being FALSE. For instance, considering the same example from sum of products, there is a circuit that has 2 inputs, A and B, and 1 output Y. If the output of the circuit is FALSE when the following conditions are met: A is logic low (A') or B is logic high, and A is logic low (A') or B is logic low (B'), in other words, A' OR B AND A' OR B', the output could be written as a Boolean equation as Y = (A + B) (A+B'). Here, (A+B) gives Y = 0 for A = 0, B= 0, also Y = 0 for A = 0, B = 1. The Key here is to find the FALSE logic.

Both the sum of products and products of sum forms are correct and equivalent, however, for most students, the sum-of-products form is easier to use.

3.2.4 Advanced Arithmetic – Introduction

Now that we have a basic understanding of the basic gates and logic arithmetic we can begin to use these tools to create complex logical equations to suit our needs. This is a very important concept in digital design and is used extensively in the Nibble Knowledge computer. For an example, let's look at the logic in the CPU's Control Unit that determines which phase the CPU is in and sends out the necessary signals. It is not necessary to understand the nature of these signals that will be discussed in later chapters for now just try to understand the arithmetic behind the signals.

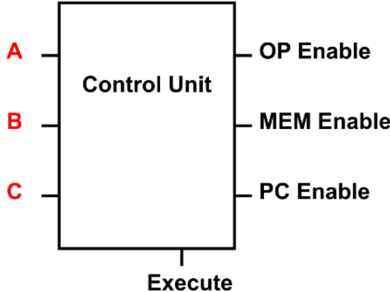


Figure 3.2.1: Control Unit of Nibble Knowledge Computer

First let's look and the truth table for the Control Unit Decode:

ABC	OP enable	MEM Enable	Execute	PC Enable
000	0	1	0	0

001	1	0	0	0
010	1	0	0	0
011	1	0	0	0
100	1	1	1	1

Table 3.2.2: Control Unit Decode Truth Table

Now let's determine the equations for each phase. Looking first at the OP enable column we can see that the output is only 0 when A, B, and C are all false this gives the equation of:

$$OP\ enable\ =\ A'B'C\ +\ A'BC'\ +\ A'BC\ +\ AB'C'$$

Using the rules we learned before we can reduce this equation to something more manageable.

$$OP\ enable\ =\ A'(B'C\ + BC'\ +\ BC)\ +\ AB'C'$$
$$OP\ enable\ =\ A'(\ C(B' + B)\ +\ BC'\ )\ +\ AB'C'$$
$$OPenable\ =\ A'(\ C\ +\ BC'\ )\ +\ AB'C'$$
$$OP\ enable\ =\ A'(\ (C + B)(C + C'))\ +\ AB'C'$$
$$OP\ enable = \ A'(C + B)\ +\ AB'C'$$
$$OP\ enable\ =\ A'(B + C)\ +\ A(B + C)'$$
$$OP\ enable\ =\ A\ + B\ +\ C$$

This can easily be checked by looking at the truth table and seeing that OP enable is only false when A, B, and C are all false.

Exercise: Try to get the other equations on your own for practice:

$$MEM\ enable\ =\ B'(A'C'\ +\ AC)$$
$$Execute\ =\ PC\ enable\ =\ AB'C$$

3.3. Hardware Reduction Techniques

After you design a circuit to have the correct functionality, you then should adjust it to use less hardware. Often this means applying De Morgan’s theorem (or Bubble pushing) to reduce the number of different logic gates you use or to reduce the number of total gates needed to accomplish a task. For instance, if you look at the circuit below, it can be reduced to one logic gate by using bubble pushing.

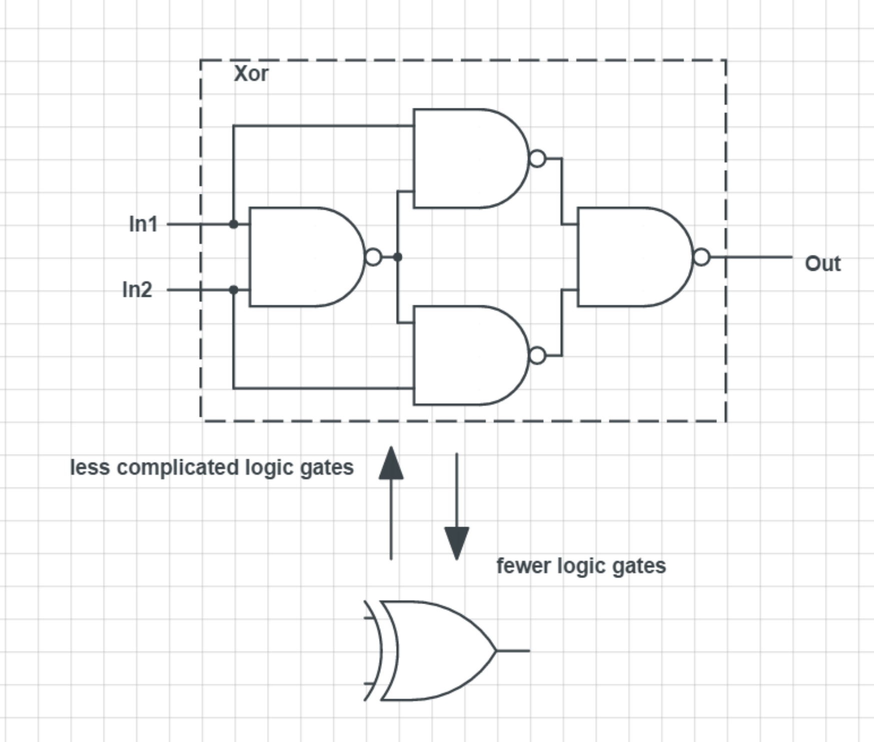


Figure 3.3.1: Hardware Reduction due to Bubble Pushing

Bubble Pushing

Bubble pushing is a technique to apply De Morgan's theorem directly to the logic diagram. Although the algebraic representation is used more often, Bubble pushing can be used to visually apply De Morgan’s theorem and is used when you need to make diagrams closer represent the gates you have available or when reducing hardware.

The bubbles on the inputs/outputs of gates shown in the figure below represent a not logic operation on those inputs/outputs.

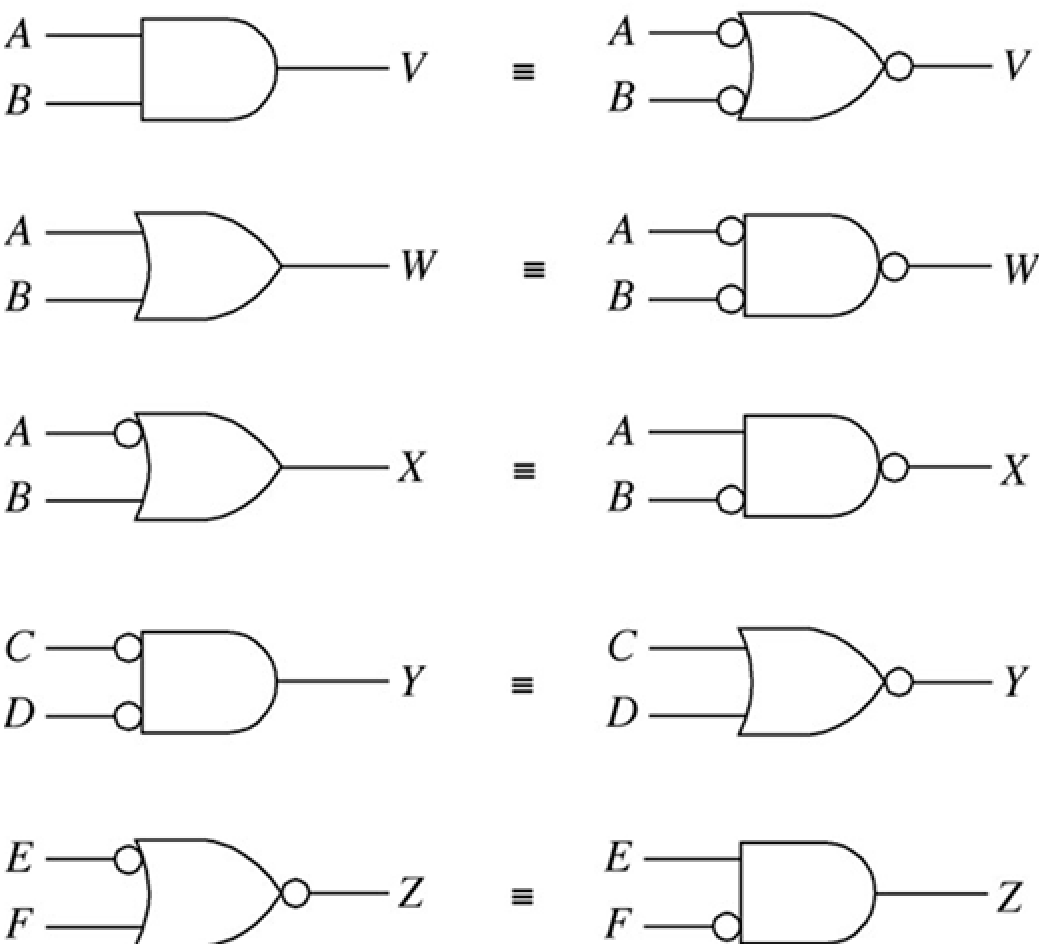


Figure 3.3.2: Logic gates transformed through Bubble Pushing

Below is are examples to illustrate the theory. It is not a very practical example, however, it illustrates the theory well. It shows how it is possible to manipulate logic by applying Bubble Pushing and change the gates used in the circuit.

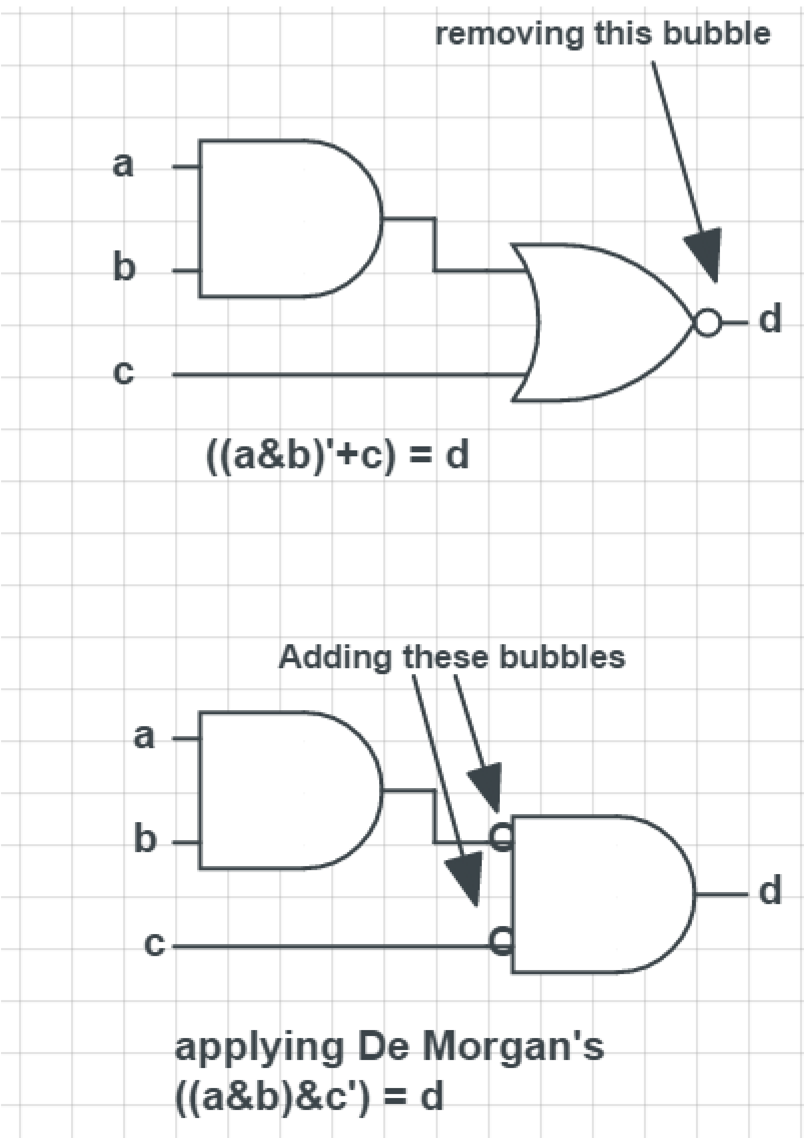


Figure 3.3.3: Example 1 of Bubble Pushing

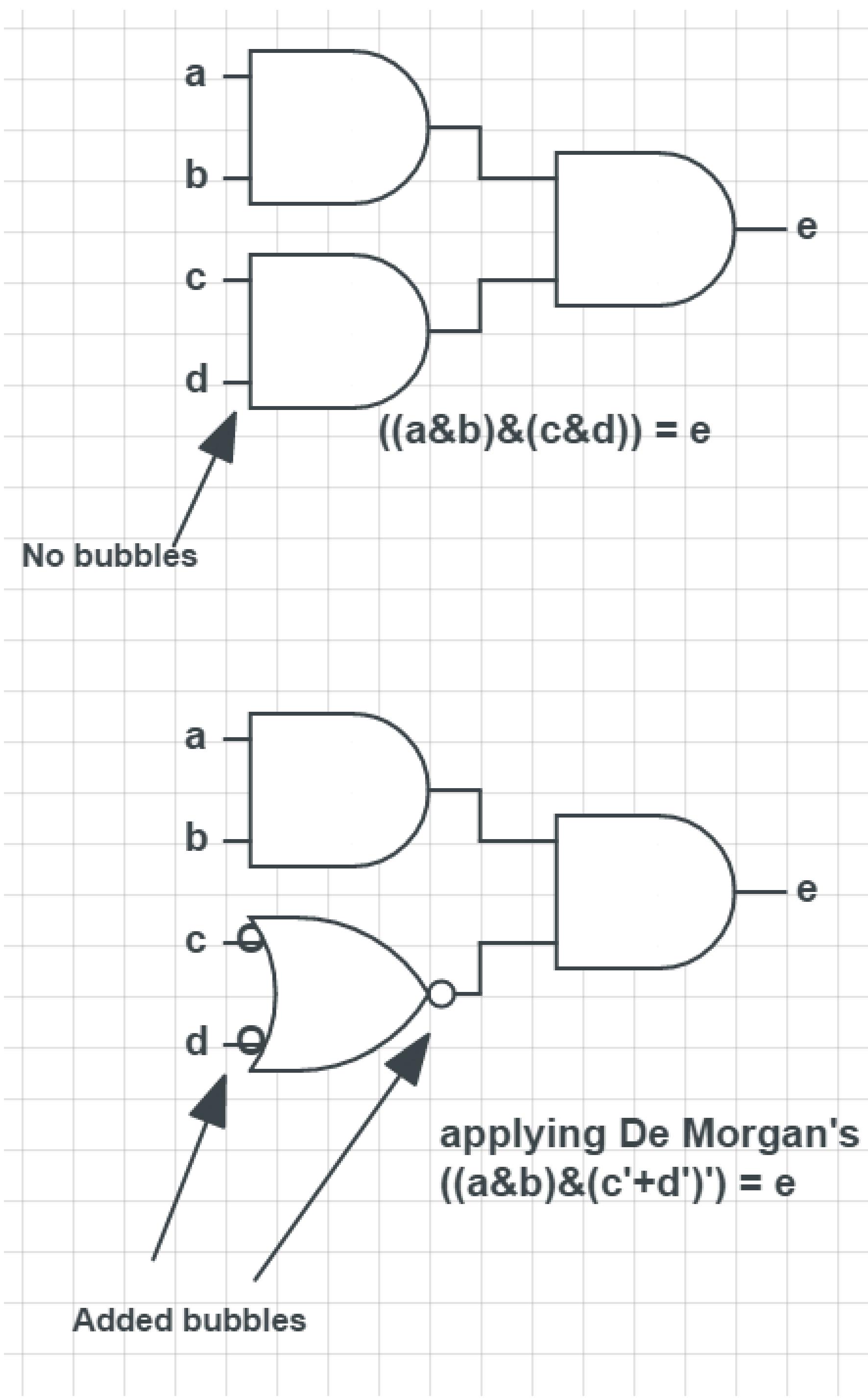


Figure 3.3.4: Example 2 of Bubble Pushing

3.4. Timing and Delays

As you have learned by now, electronic circuits rely heavily on timing. In sequential circuits, if a signal is changing from one state to another at the same time as that signal is being read into a flip flop many issues could arise. For this reason, we must look deeper into some theory behind timing and delay in sequential circuits.

Propagation and Contamination Delay

There are two main types of delay that we need to worry about: propagation delay, and contamination delay. Propagation delay is the time between the last input change, and the last output change (input at steady state, to output at steady state), while contamination delay is the time between the first input change, to the first output change (input is contaminated, to output is contaminated). They are very similar, but it is important to know the difference. We will mainly focus on propagation delay, because that is what is shown on data sheets when buying chips. These delays are shown in a timing diagram below. The box between a, and b is a combinational logic circuit. From Figure 3.4.2, "tcab" is the contamination delay from a to b, and "tdab" is the propagation delay from a to b. A common propagation delay for an LS chip is 20ns (AND, OR, NAND, NOR, gates are all this).

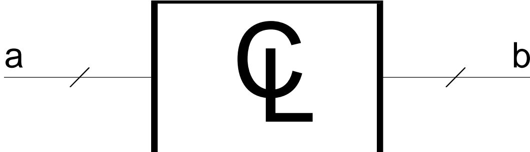


Figure 3.4.1: Combinational Element

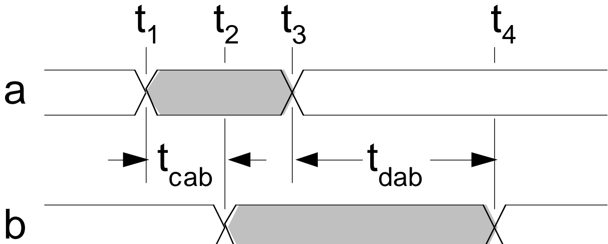


Figure 3.4.2: Timing of Combinational Element

When designing a sequential circuit, it is important to make sure that the propagation delay of all elements in-between clock elements is less than the period of the clock. For example, if there were flip flops on either side of Figure 3.4.1, then the period of the clock must be greater than tdab. To

add to this, if there were multiple paths between clocked elements, the longest path must be less than the period of the clock.

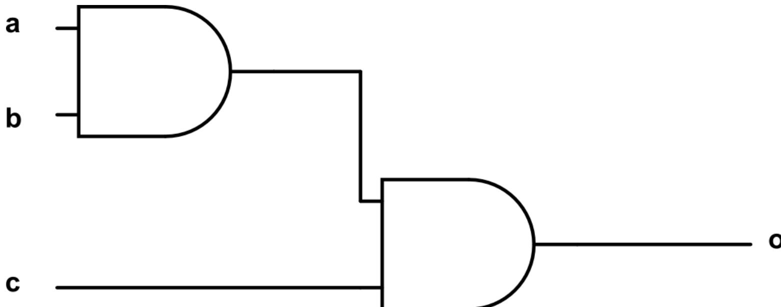


Figure 3.4.3: Small Combinational Circuit

Looking at Figure 3.4.3, there are two paths through the circuit: from a/b to o, or from c to o. The longest path is obviously from a or b through to o, so this is the one we would want to measure the delay of.

Steps for Solving

1. Identify the clock driven elements
2. Identify all paths between clock elements, then identify the longest path
3. Calculate the delay of the longest path
4. Check that this delay is less than the clock period

Example 1

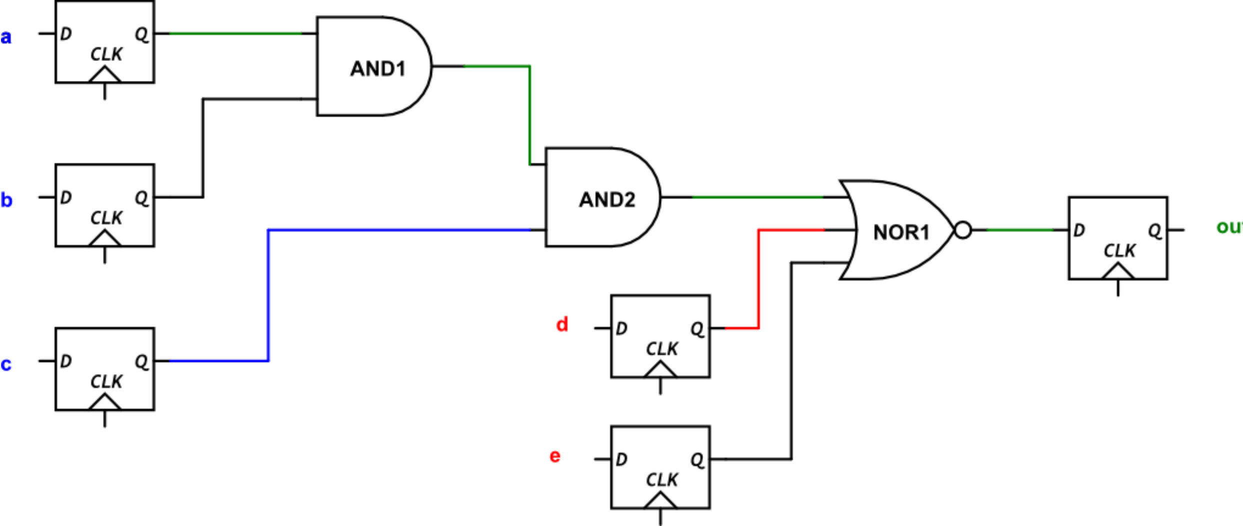


Figure 3.4.4: Circuit for example 1

Above is the circuit we will be analyzing to determine the longest path.

Given information:

- t_{pdAND} = 20ns, t_{pdNOR} = 50ns
- T_{clock} = (clock period) = 200ns

Now let us follow the steps for solving to determine the longest delay.

1. *Identify the clock driven elements.*

   It is quite easy in this case to find all the clock driven elements because they are the only ones with a "CLK" input on them. You can also find them as the elements with the V shape below "CLK". This tells the viewer that the element is a rising edge triggered device.

   There are six clock driven elements, the three with blue inputs, the two with red, and the output one with a green output.

2. *Identify all paths between clock elements, then identify the longest.*

   There are three possible paths that we could take through the circuit that give us different results. The top path shown in green, the middle path shown in blue, and the bottom path shown in red. It is important to note that the bottom path begins from a completely different clock driven component than the other two paths. As long as a path starts and ends at a clock component, it can be a potential longest path. The top green and black path are identical. Looking at the delays of each gate, the top green path has the longest delay.

3. *Calculate the delay of the longest path.*

   Using the green path, and the delays given above we can calculate the $t_{longest\ path}$.

   $t_{longest\ path} = (2 * t_{pdAND}) + t_{pdNOR}$ = 90ns

4. *Check that this delay is longer than the clock period.*

   90ns < 200ns

   The delay is well under the clock period so this circuit should have enough time to process the combinational circuitry before the clock edge saves the output to the final flip flop on the right of Figure 3.4.4.

Example 2

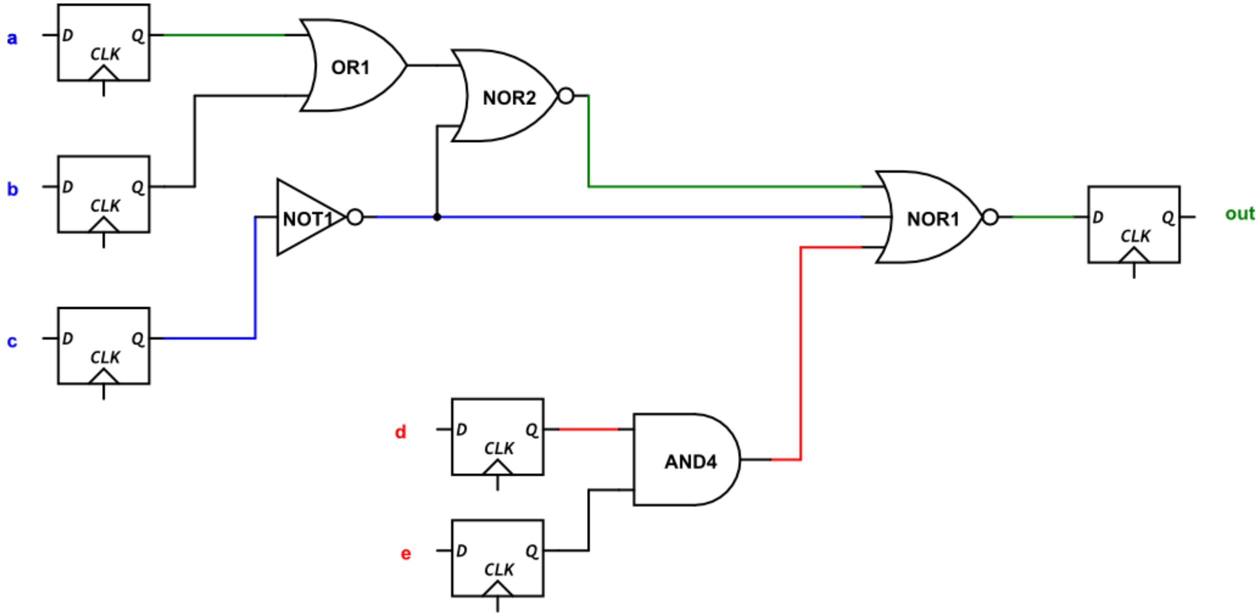


Figure 3.4.5: Circuit for example 2

In this example the final goal is a little different. Instead of comparing to the clock period in step 4, let us find what the minimum clock period is, in order to find the maximum frequency of the circuit.

Given information:

- t_{pdAND} = 50ns, t_{pdNOR1} = 50ns (3-input NOR labeled NOR1), t_{pdNOR2} = 15ns, t_{pdOR} = 15ns, t_{pdNOT} = 10ns

Let us again follow the steps:

1. *Identify the clock driven elements.*

   This step is the same as in example 1, there are 6 clock driven elements.

2. *Identify all paths between clock elements, then identify the longest.*

   There are many different paths in this example, three of them are shown in the diagram but there is another one that starts by following the blue path, but then goes up through the NOR. Since the delays of the OR and the NOT are identical, this path and the top green path end up being the same.

   In this problem it may be beneficial to calculate multiple paths in this step, because it may not be obvious by just looking at the circuit.

   - Green path: $t_{green} = t_{pdOR} + t_{pdNOR2} + t_{pdNOR1}$ = 15ns + 15ns + 50ns = 80ns

- Blue path: $t_{blue} = t_{pdNOT} + t_{pdNOR1} = 10ns + 50ns = 60ns$
- Red path: $t_{red} = t_{pdAND} + t_{pdNOR1} = 50ns + 50ns = 100ns$

In this case, the high delay on the AND gate actually made the red path the longest even though it has fewer gates than the green.

3. *Calculate the delay of the longest path.*

   This was already done in the previous step.

4. *Check that this delay is longer than the clock period.*

   This time this step will be a little different. We do not know the clock period, instead we are asked to find the maximum frequency of the circuit. Knowing that the longest delay is 100ns, we use that as the minimum period of the clock. This gives us a frequency of 1/100ns = 10 MHz. This number is a little unrealistic because there would be absolutely no wriggle room, but it is the theoretical maximum frequency that the circuit can run at.

3.5. Multiplexer (MUX)

Multiplexer (MUX) is a combination logic circuit that selects one of multiple input signals and directs it to a single output. A 2^N multiplexer allows you to select one output from N inputs. The multiplexer fundamentally acts as a multiple input, single output switch, and the signals can be either analog or digital signals.

Multiplexer is generally used in combination with de-multiplexer (DEMUX). The multiplexer is first used to combine multiple input into a single output, which can be sent to the de-multiplexer. The de-multiplexer receives the single stream of data and splits it into the original multiple signals. In this case, using the multiplexer and de-multiplexer greatly reduced the amount of channels required to send the data. Instead of requiring N channels to send the data, by using 2^N multiplexer and de-multiplexer, only one channel is required.

Common size of Multiplexer are 2-to-1, 4-to-1, 8-to-1 and 16-to-1. Since digital logic uses binary values, powers of 2 are used (4, 8, and 16) to maximally control a number of inputs for the given number of selector inputs.

2-to-1 Multiplexer

$$Y = X_0\bar{C} + X_1C$$

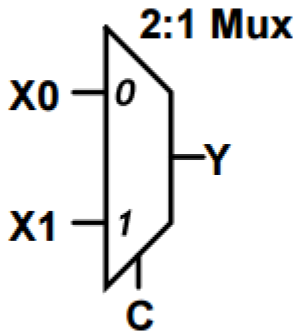


Figure 3.5.1: 2:1 Multiplexer

Input		Select	Output	
X_0	X_1	C	Y_0	Y
0	0	0	0	X_0
0	0	1	0	X_1
0	1	0	0	X_0
0	1	1	1	X_1
1	0	0	1	X_0
1	0	1	0	X_1
1	1	0	1	X_0
1	1	1	1	X_1

Table 3.5.1: Truth Table of 2:1 Multiplexer

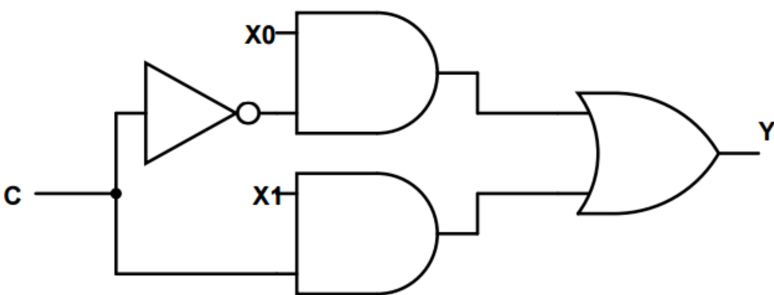


Figure 3.5.2: Circuit Diagram of the 2:1 Multiplexer

4-to-1 Multiplexer

$$Y = X_0\overline{C_1C_0} + X_1\overline{C_1}C_0 + X_2C_1\overline{C_0} + X_3C_1C_0$$

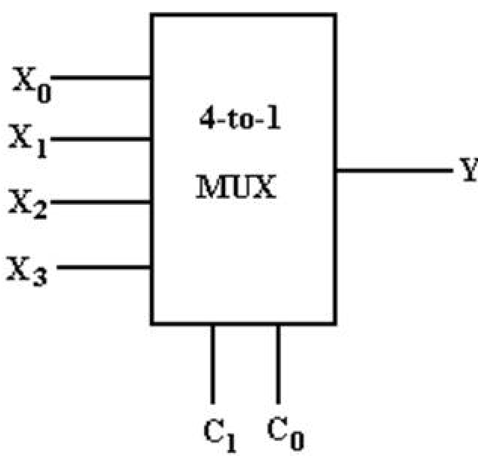


Figure 3.5.3: 4:1 Multiplexer

Select Data Input		Output
C_1	C_0	Y
0	0	X_0
0	1	X_1
1	0	X_2
1	1	X_3

Table 3.5.2: Truth Table of 4:1 Multiplexer

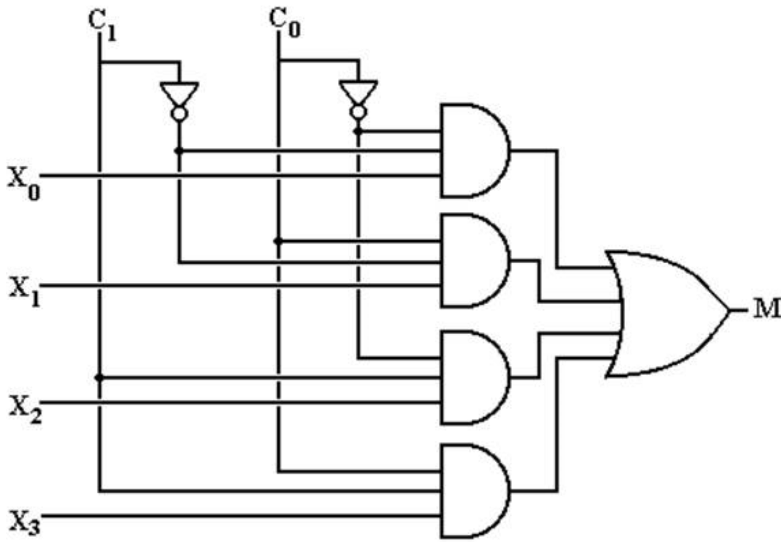


Figure 3.5.4: Circuit Diagram of 4:1 Multiplexer

This truth table and diagram shows when both select input equal to 0 then output Y=X_0 when both select input equal to 1 then output Y=X_3. When select input C_1=0, C_0=1 then output Y=X_1, and when select input C_1=1, C_0=0 then output Y=X_2.

There is an alternative way to implement a 4-to-1 Mux is using three 2-to1 Mux. In addition, this scenario works for 8-to-1 Mux and 16-to-1.

Special case: 4-to-1 Mux from 2-to-1 Mux

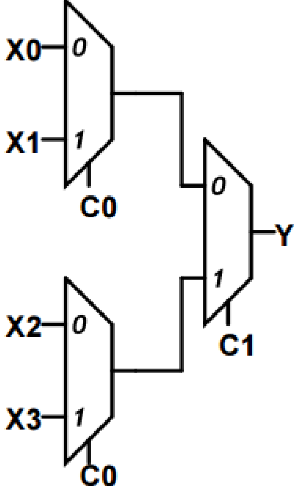


Figure 3.5.5: Special Case of the 4:1 Multiplexer

8-to-1 Multiplexer

$$Y = \overline{C_2 C_1 C_0} X_0 + \overline{C_2 C_1} C_0 X_1 + \overline{C_2} C_1 \overline{C_0} X_2 + \overline{C_2} C_1 C_0 X_3 + C_2 \overline{C_1 C_0} X_4 + C_2 \overline{C_1} C_0 X_5 + C_2 C_1 \overline{C_0} X_6 + C_2 C_1 C_0 X_7$$

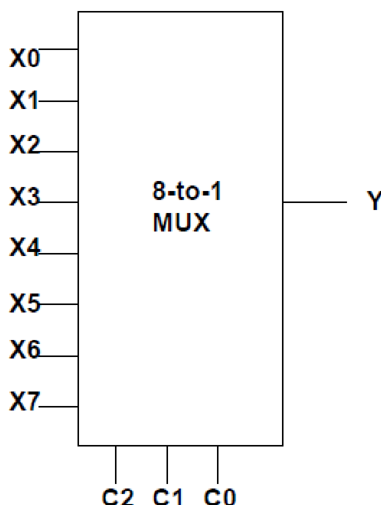


Figure 3.5.6: 8:1 Multiplexer

Select Data Input			Output
C_2	C_1	C_0	Y
0	0	0	X_0
0	0	1	X_1
0	1	0	X_2
0	1	1	X_3
1	0	0	X_4
1	0	1	X_5
1	1	0	X_6
1	1	1	X_7

Table 3.5.3: Truth Table of 8:1 Multiplexer

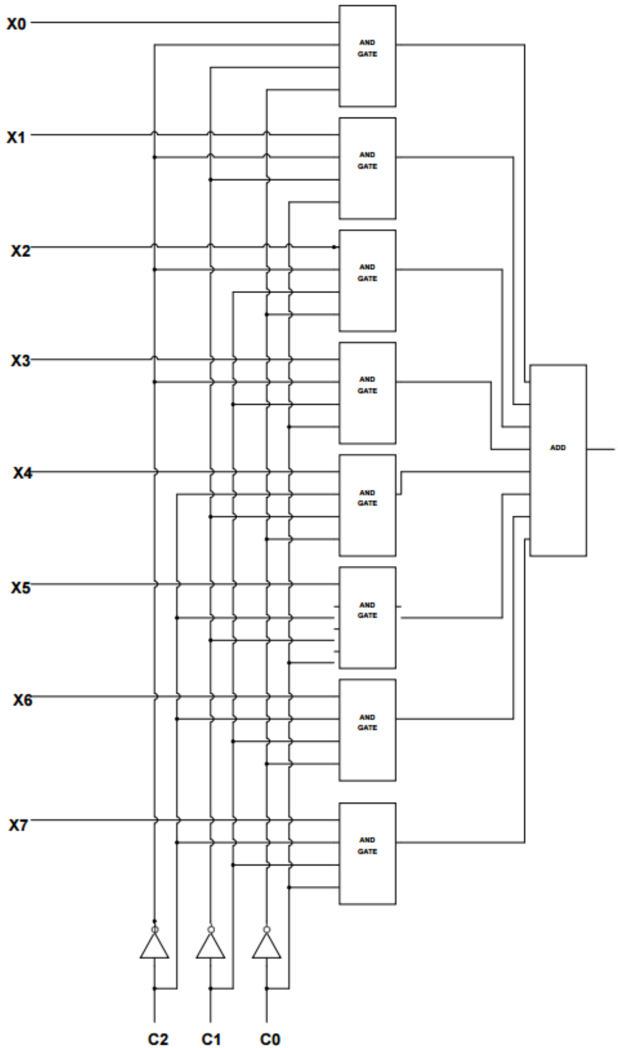


Figure 3.5.7: Circuit Diagram of 8:1 Multiplexer

This truth table and diagram shows when all select input equal to 0 then output Y=X_0 when all select input equal to 1 then output Y=X_7. When select input C_2=0, C_1=0, C_0=1 then output Y=X_1, when select input C_2=0, C_1=1, C_0=0 then output Y=X_2, when select input C_2=0, C_1=1, C_0=1 then output Y=X_3, when select input C_2=1, C_1=0, C_0=0 then output Y=X_4, when select input C_2=1, C_1=0, C_0=1 then output Y=X_5, and when select input C_2=1, C_1=1, C_0=0 then output Y=X_6.

There are many advantages to using a multiplexer. It can be used to increase the amount of data that can be sent within a fixed time or bandwidth. In addition, multiplexer reduces the number of wires required in a system, which in turn lowers the complexity of the circuit and the cost.

3.6. Flip-Flops and Latches

Flip flop and latch are the basic building block of the sequential logic circuit, they are two state (Set/Reset) synchronous device with feedback path that can be used to store one bit of data. The primary difference between flip flop and latch is that flip flop is edge triggered and latch is pulse triggered. This means the flip flop output changes only during the brief instances where the clock input changes from high to low or low to high. The latch checks all its inputs continuously and change its output according to input at any period of time. Latches are generally faster, which makes them useful for high speed designs. In addition, since they're pulse triggered, they require less power. However, latches are less predictable, which makes it harder for designer to perform timing analysis. The advantage of using flip flop is that it helps the circuit designer to maintain better control over the timing in the circuit and perform accurate timing analysis.

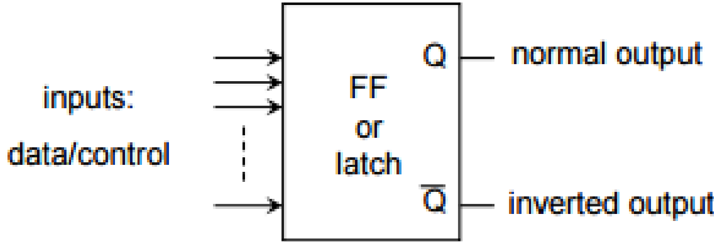


Figure 3.6.1: Block Diagram of a Flip-Flop or a Latch

From the above figure:

Q=1 is the SET state

Q=0 is the RESET state

Flip flop and latches normally have 2 complementary outputs which usually donated as Q and Q'.

S-R Latch

SR latch can be created with two NOR gates that have a cross-feedback loop.

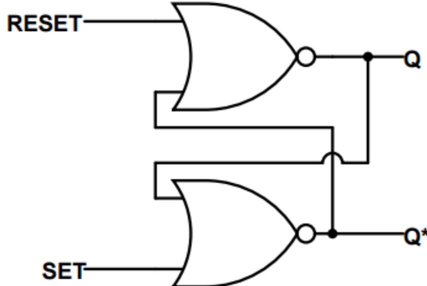
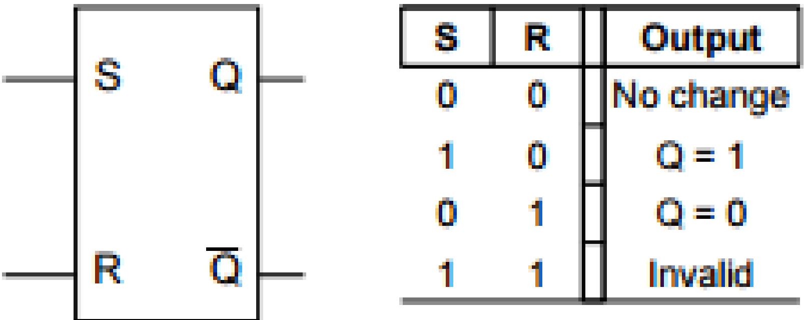


Figure 3.6.2: Circuit Diagram of S-R Latch



S	R	Output
0	0	No change
1	0	Q = 1
0	1	Q = 0
1	1	Invalid

Figure 3.6.3: Block Diagram and Truth Table of S-R Latch

When a high is applied to the Set line of an SR latch, the Q output goes high (and Q low). The feedback mechanism, however, means that the Q output will remain high, even when the S input goes low again. This is how the latch serves as a memory device. SR latches can also be made from NAND gates, but the inputs are swapped and negated.

D Latch (transparent latch)

A D latch is like an S-R latch with only one input: the "D" input. Activating the D input sets the circuit, and de-activating the D input resets the circuit.

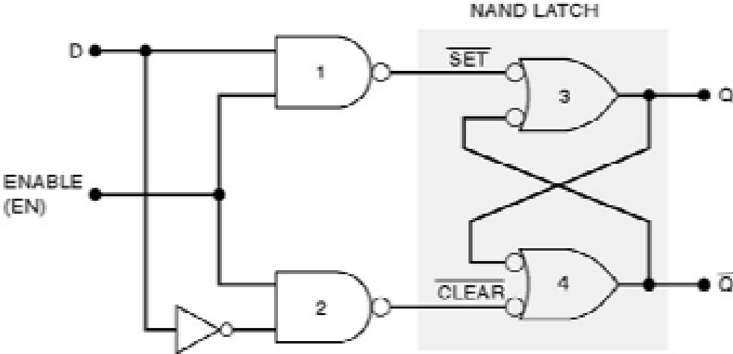
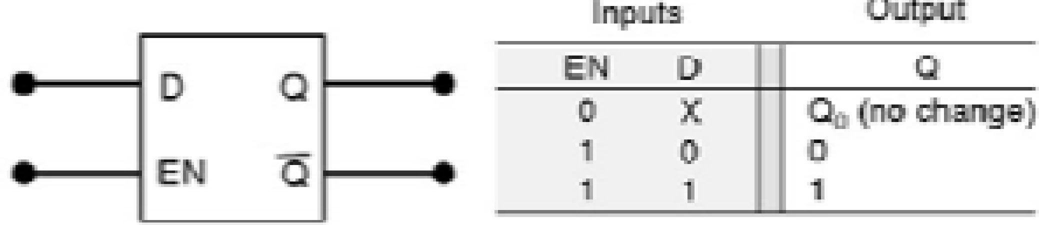


Figure 3.6.4: Circuit Diagram of the D Latch



Inputs		Output
EN	D	Q
0	X	Q_0 (no change)
1	0	0
1	1	1

Figure 3.6.5: Block Diagram and Truth Table of the D Latch

The D latch is used to capture, or 'latch' the logic level which is present on the Data line when the clock input is high. If the data on the D line changes state while the clock pulse is high, then the output, Q, follows the input, D. When the CLK input falls to logic 0, the last state of the D input is trapped and held in the latch.

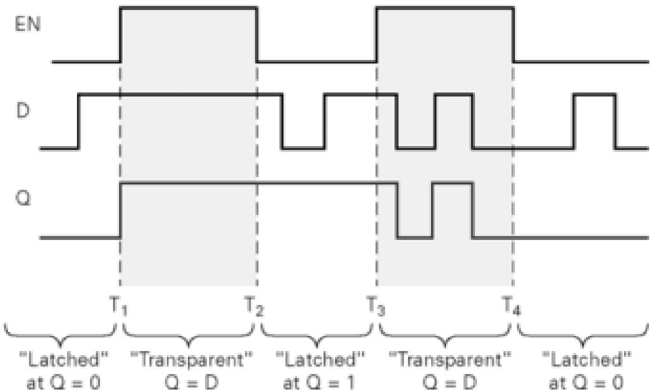


Figure 3.6.6: Timing Diagram of the D Latch

D Flip-Flop

A D flip-flop, known as a data or delay flip-flop, captures a value and only propagates it on a rising clock edge. A rising edge is when a value goes from a 0 logic level to a 1 logic level.

The working of D flip flop is similar to the D latch except that the output of D Flip Flop takes the state of the D input at the moment of a positive edge at the clock pin (or negative edge if the clock input is active low) and delays it by one clock cycle. That's why, it is commonly known as a delay flip flop. The D Flip Flop can be interpreted as a delay line or zero order hold.

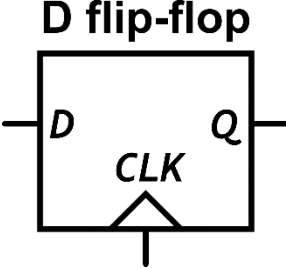


Figure 3.6.7: Block Diagram of the D Flip-Flop

Clock	D	Q
Rising edge	0	0
Rising edge	1	1
Non-Rising	X	Q

Table 3.6.1: D Flip-Flop Truth Table

Table 3.6.1 shows that on a rising edge the inputs propagated to the output but when there isn't a rising edge the output remains at the last value that was propagated.

Note: An X is known as a "don't care" symbol showing that the result doesn't change whether this value is a 1 or a 0.

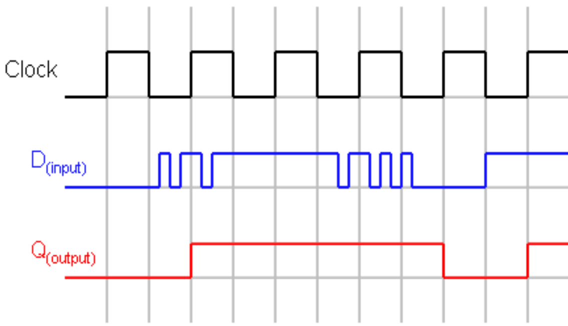


Figure 3.6.8: Timing Diagram of the D Flip-Flop

Although there are many clock driven devices the DFF is the simplest and often a building block for other clock driven devices such as a register. The advantage of the D flip-flop over the D-type "transparent latch" is that the signal on the D input pin is captured the moment the flip-flop is clocked, and subsequent changes on the D input will be ignored until the next clock event.

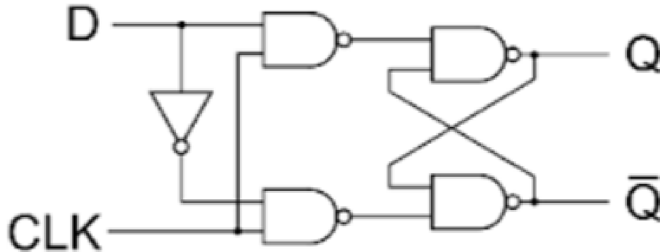


Figure 3.6.9: Circuit Diagram of the D Flip-Flop

4. Arithmetic Circuits

4.1. Addition

Adders in digital circuits take two binary numbers as inputs to the circuit and produce the sum of the two numbers as the output with the addition of a carry if necessary. These circuits have are essential in most digital logic circuits and have a wide variety of used including subtraction, multiplication, division, as well as its main purpose of adding two numbers together.

In the Nibble Knowledge CPU there are two adders in the hardware. The first one is located in the ALU and is used in the add instruction more abstractly through multiple instructions it also allows the CPU to do all other forms of arithmetic as mentioned above. The second adder is located in the program counter loop. This adder has the single purpose of adding 1 to the address held in the PC register. This means that one input of the adder is always set to 1 while the other is set to the program counter address.

4.1.1 The Half Adder

A half adder is also known as a binary adder, it adds together two or more 1-bit binary numbers. A half adder has two inputs (A and B) and two outputs (Sum and Carry out). The half adder is incomplete in the sense that it does not do anything with the carry, as such if the input to the half adder has a carry it will be neglected and will add only the A and B bits. As such, the arithmetic operations are incomplete for half adders. First let us take a basic look at how a binary adder works at the most basic level the 1-bit half adder. This is the addition of two bits together to produce a single output (the sum) with a carry out signal. The truth table for this is shown in the table below.

Inputs		Outputs	
A	B	Cout	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 4.1.1: Truth Table for a Half Adder

The half adder generates the following Boolean expression:

Cout = A * B

Sum = (A’ * B) + (A * B’)

Using this truth table we can see that the sum is only true when either A or B are true but false when they are both true. The carry out signal however is only true when A and B are both true. Using this logic we get the following logic diagram.

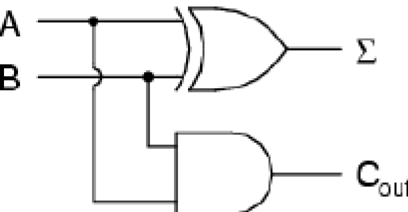


Figure 4.1.1: Circuit Diagram of the 1-Bit Half Adder

Where the inputs A and B are send into an XOR gate to produce the sum output and into a AND gate to produce the carry out signal. The 1-Bit Half Adder circuit could be further broken down into the following circuit.

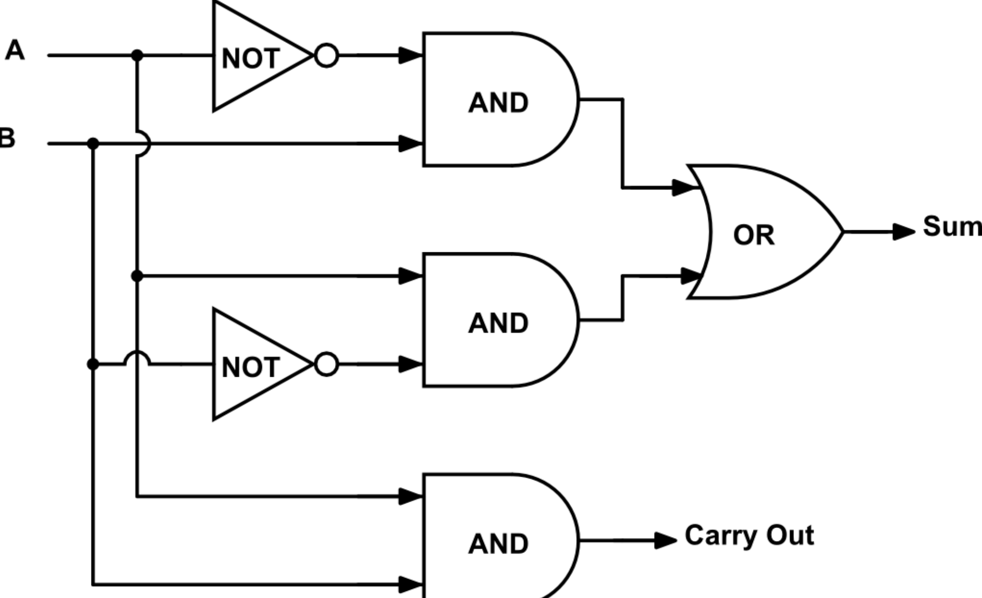


Figure 4.1.2: Complete Half Adder Circuit

4.1.2 The Full Adder

The difference between a full adder and half adder is that a full adder had three inputs instead of just two. There inputs are 1-bit data A, 1-bit data B, and a Carry-in. The carry-in allows the adder to receive the carry from a previous stage. Now that we have looked at the basic building block of an adder with the half adder we will now take a look at the full adder which expands the half adder to also have a carry into the adder. This creates a general block that as we will see later can be

cascaded to create bigger adders. The truth table and the logic diagram for a full adder are shown in the figures below.

Inputs			Outputs	
A	B	Cin	Result	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 4.1.2: Truth Table for the Full Adder

We then need to simplify the full Adder's truth table to determine the logic gates need for the full adders design. So, we convert the above truth table into a logical expression. Let's start by separating the table based on the Cout value.

Cout = 1		
A	B	Cin
0	1	1
1	0	1
1	1	0
1	1	1

Table 4.1.3: Values for inputs A, B and Cin when Carry Out is a 1

Next we take this simplified table and generate a logical expression for the Carry Out component of the full adder.

Cout = (B * Cin) + (A * Cin) + (A * B) + (A * B * Cin)

If the last term is true, (A * B * Cin) = 1, then all the other terms must also be true as such the last term can be eliminated as it is redundant.

Cout = (B * Cin) + (A * Cin) + (A * B)

From this expression we can design the adder component that generates the Cout output signal.

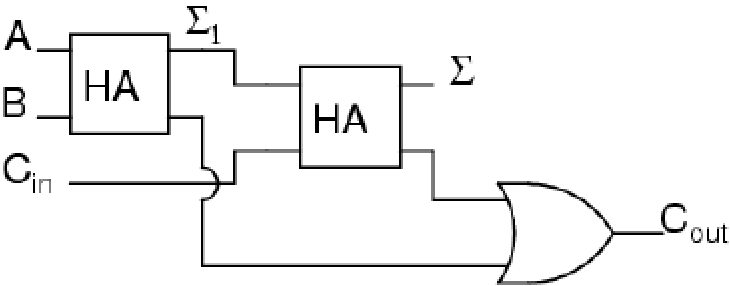


Figure 4.1.3: Circuit Diagram of the Full Adder

As you can see in the figure a full adder consists of the cascading of two half adders. The first one adds the two bit to be added together and generates the sum and necessary sum and carry out bits. The sum of the first adder is then added to the carry in signal of the full adder in the second half adder block. This second half adder generates the final sum of the full adder. The carry outs of the two half adders are then sent into an OR gate to generate the carry out for the full adder.

Now that we know what it takes to make a full adder we can put it into a black box that has 3 inputs of input A, input B, and carry in, and 2 outputs of sum, and carry out as shown in the figure below.

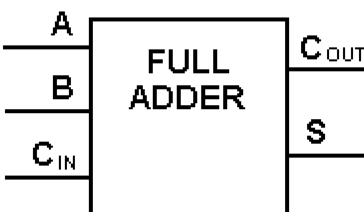


Figure 4.1.4: Full Adder Black Box

Sum = (A * B * Cin’) + (A’ * B * Cin’) + (A’ * B’ * Cin) + (A * B * Cin)

Exercise: Confirm the Boolean expression for the Result output signal.

Exercise: Design the Sum component of the full-adder using the appropriate logic gates.

4.1.3 The Ripple-Carry Adder

Using the block above, we can now cascade the adder together to create adders that can handle as many bits as necessary for the application required. This is done by connecting the carry out of the previous bit’s full adder into the carry in of the next full adder as shown in the diagram below. This is known as a ripple carry counter. The ripple carry adder allows you to add two n-bit

numbers. To construct a ripple carry adder we use half and full adders and add them one at a time. However, it is possible to generate a ripple carry adder using only full adders.

An n-bit ripple carry adder connects the carry out of the less significant bit to the carry in of the most significant bit and cascades down until it reaches the last, nth, full adder. Below is an example of a 4-Bit Ripple Carry Adder

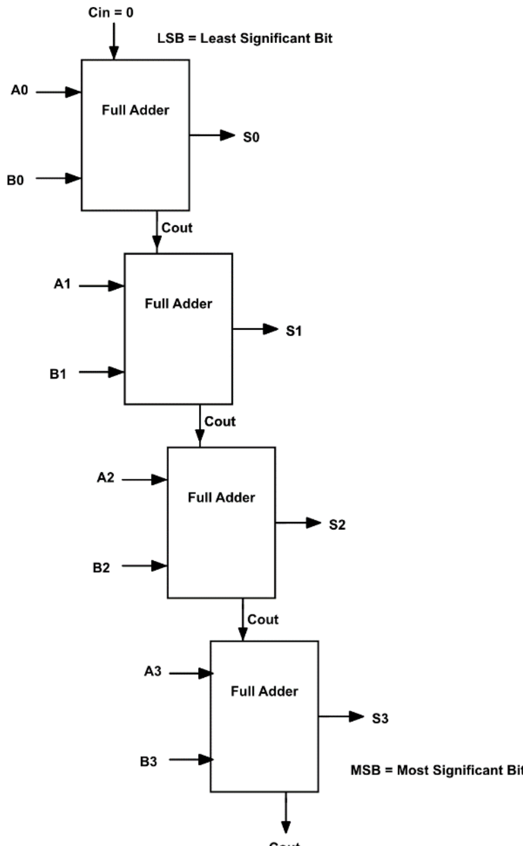


Figure 4.1.5: A 4-Bit Ripple Carry Adder

The Ripple Carry Adder is not limited to any specific number of bits. It is possible to add as many bits as possible.

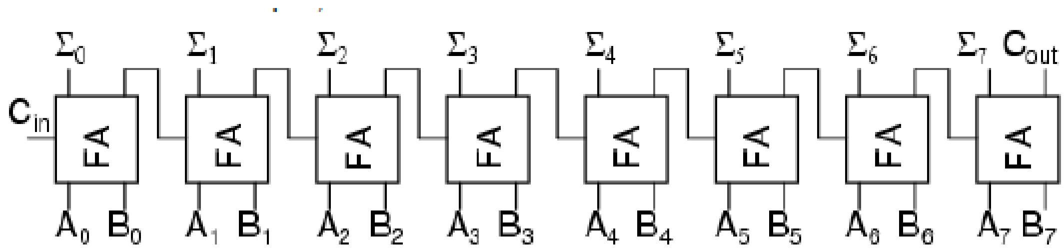


Figure 4.1.6: Example of larger Ripple Carry Adder

Using this method however can have drawbacks however. Since each carry in depends on the carry out of previous bit which in turn depends on the carry out of the bit before that, the adder becomes increasingly slow and cumbersome to deal with in a high speed circuit with the addition of each additional bit as the propagation delays of each previous full adder are added in series creating extremely long delay times.

The Carry-Lookahead Adder

One solution to the problem of increasing delay times in the cascading of full adders is the carry-lookahead adder. This adder is based on the fact that a carry signal is generated in two cases:

1. When both input bits are 1, or
2. When one of the two bits is 1 and the carry-in is 1

Implementing these two cases in an adder circuits gives us the logic diagram as shown below.

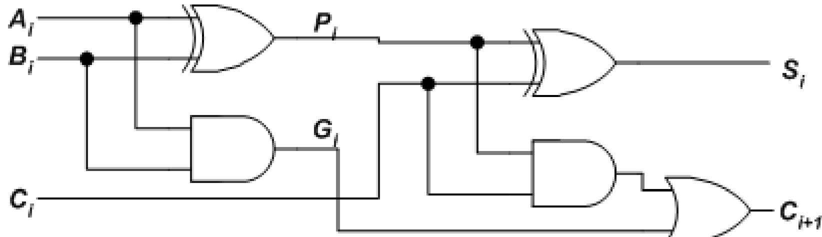


Figure 4.1.7: Logic Expression for Carry-Lookahead Adder cases

Where the two intermediate signals shown on the diagram are defined as:

Carry Generate (G) - generated when both of the input bits of a single adder are both '1' regardless of the carry in value

Carry Propagate (P) – This signal is generated by the XOR of the input bits. The purpose of this is to carry the carry in of the addition through to the carry out immediately if one of the input bits is '1' because if that is true then there is directly dependent on the carry in.

Looking at the 4-bit carry-lookahead adder it consists of three levels of logic:

1. First level: The first level generates all of the P and G signals in the adder. This means one set for each of the four levels.
2. Second level: This is the Carry-lookahead block which consists of four 2-level implantations of logic. It generates the carry signals for each bit of the adder as defined by these equations

$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0)$$
$$= G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$
$$C_4 = G_3 + P_3C_3$$
$$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

3. Third level: Four XOR gates which generate the sum signals for the P and carry in of each of the four levels of the adder

With this set up the delay of the carry-lookahead adder is 4τ. Comparing this with a 4-bit ripple counter that has a delay of $(2n+1)\ \tau$ where n is the number of bits in the adder. This means that for a 4-bit ripple counter it takes 9τ compared to the 4τ of the carry-lookahead counter.
The disadvantage of carry-lookahead adders however, is that once they are constructed beyond 4-bits the carry equations needed for the second level of logic become overly complicated and cumbersome. This is why most carry-lookahead counters are implemented in 4-bit modules and cascaded similar to ordinary full adders.

4.2. Subtraction

The in order implement a subtraction operation, we need to ensure that the arithmetic operation being performed is in two's complement as done with the addition operation.

As such, to subtract B from A, we invert input B and then add 1 to the inverted B value to generate a two's complement expression for the A-B operation. Then the two's complement of B is added to A in the adder to find the result.

So the final expression for the subtract operation is:

Sum = A + (B' + 1)

To account for the addition of 1 to B we can take the circuit for a full adder and add a mux to select a Carry input of 1 to the Least Significant bit, when a control signal is sent the mux will select between the Cin of 0 (for addition) or 1 (for subtraction). As such the circuit for the subtraction operation is simply the addition of two mux's and a not gate.

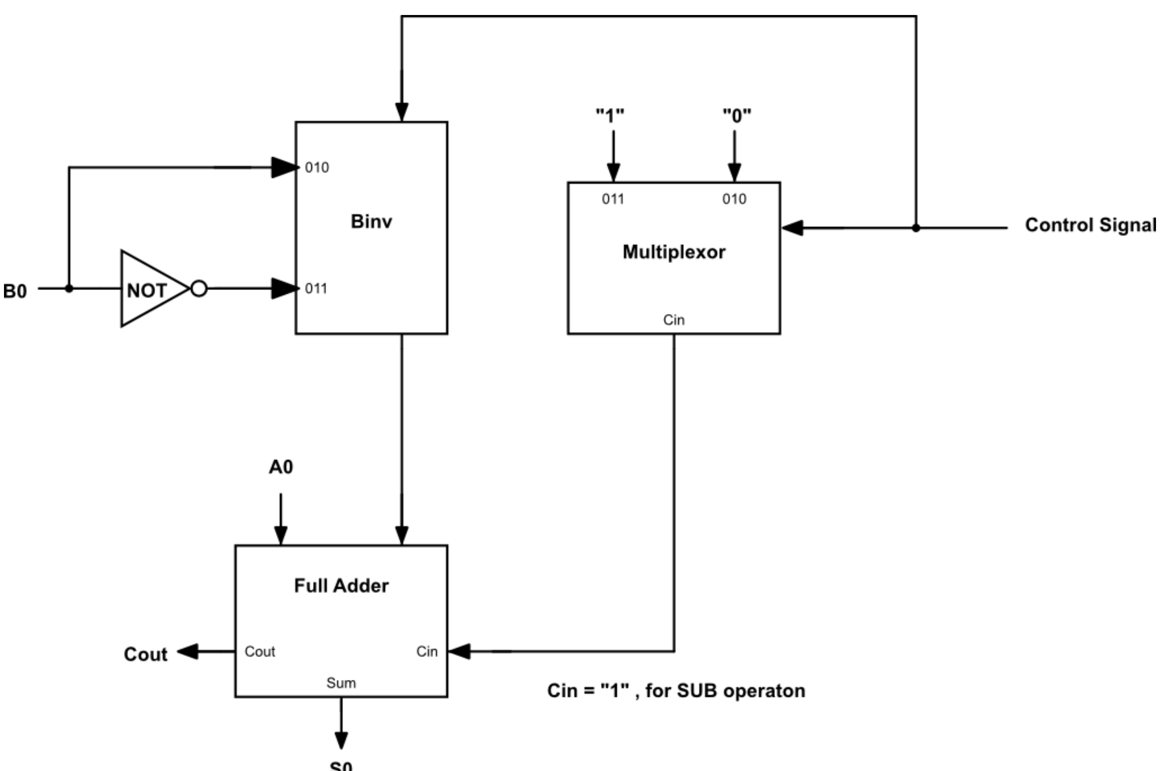


Figure 4.2.1: Algorithmic Logic Unit that performs Subtraction

4.3. Counters

Unlike combinational circuits where the system produce an output based on input variables only, sequential circuits use input variables and pervious input variables by storing the information and putting in back into the circuit on the next clock cycle (activation edge). This type of circuit uses pervious input, output, clock and a memory element

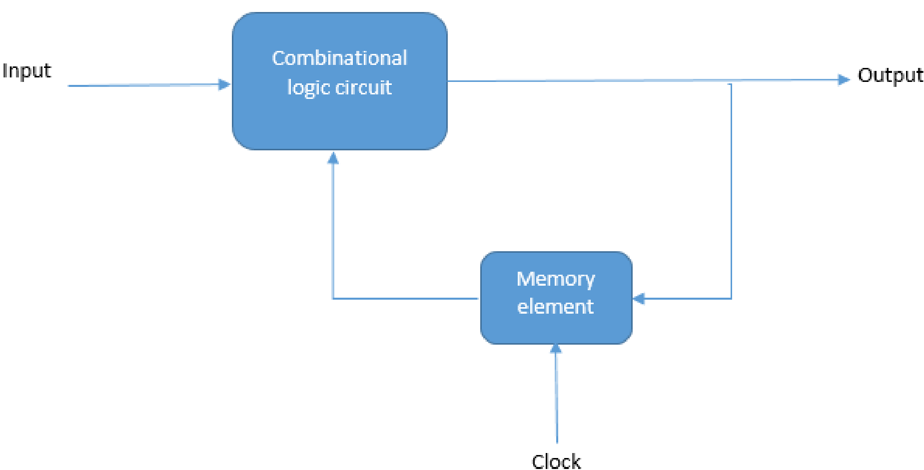


Figure 4.3.1: Sequential Logic derived from Combinational Logic

The figure above shows a theoretical view of how sequential logic are made up from combinational logic and some memory storage element.

There are two types of inputs to the combinational logic:

- External inputs: come from outside the circuit and they are not controlled by the circuit
- Internal inputs: function of a pervious output states

Synchronous:

In synchronous circuits the input are pulses or levels to drive the circuit. Synchronous circuits are known as clocked sequential circuits which uses flip-flops or gate latches as digital storage (memory elements). There is a (synchronizing) periodic clock connected to the clock inputs of all the memory elements of the circuit, to synchronize all internal changes of state. Therefore, the operation of the entire circuit is controlled and synchronized by the periodic pulses of the clock.

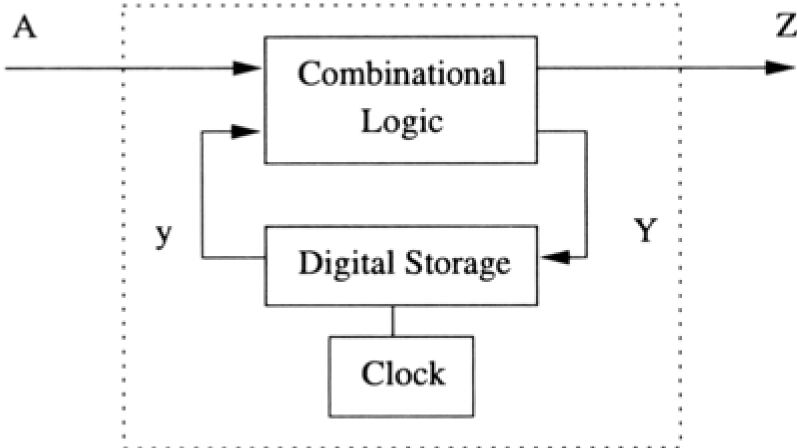


Figure 4.3.2: **B**lock diagram of a synchronous sequential circuit

Asynchronous:

In asynchronous sequential circuits the inputs are levels and there are no clock pulses, only the change in inputs drive the circuit. In other words, if the circuit does not employ a periodic clock signal to synchronize its internal changes of state is considered to be asynchronous. In general, an asynchronous circuit does not need the timing control of the synchronous type.

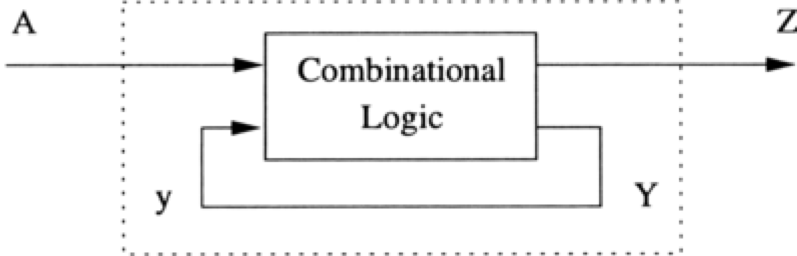


Figure 4.3.3: Block diagram of a synchronous sequential circuit

To recap, in sequential circuits the output depends, not only on the combination of logic states at its inputs, but also on the logic states that existed previously. In other words the output depends on a SEQUENCE of events occurring at the circuit inputs. Examples of such circuits include clocks, flip-flops, bi-stables, counters, memories, and registers. In this chapter, the concept of these circuits will be introduced. Also, we will discuss some examples of sequential circuit that were used in our Nibble Knowledge computer kit.

What are Counters?

A counter is a device that stores the amount of times an event has occurred in relation to a clock signal. Sequential digital logic circuits are the most common type of counter. They include an input line called a 'clock' and multiple output lines. These output lines usually represent a number in either binary or BCD. When a pulse is applied to the input line (or a clock signal) the counter will either increment or decrement depending on how you have set up your counter. Counters are easily designed using a cascade of flip-flops ("Counter (digital)", n.d., para. 1-2).

Why are Counters Used?

Counters are generally used to keep time and control the length of processes. Counters have many applications, some of which include:

- They keep time in alarm clocks
- They count to a predefined number when you set a delay timer on your camera
- The frequency of flashes for signal lights on vehicles are controlled by counters

Although counters have a vast variety of applications in many industries the role they have in computers is paramount.

Where are they used in the Computer?

Without the use of counters a computer would not work. The two most important roles that counters have within a computer are:

I. Dividing the clock frequency, since some elements of a computer run at a lower frequency than others
II. Controlling the length of processes within various elements of the computer

Example:

When a PS/2 Keyboard button is pressed it sends an 11-bit code along with a clock signal to the PS/2 keyboard controller (the intermediate interface between the keyboard and the CPU). The code sent from the keyboard is shifted into a serial in parallel out shift register. A counter is used to count the pulses of the clock signal sent by the keyboard and halts the shift register when eleven clock pulses have been counted. The counter is vital in this application because it avoids the loss of any data transferred from keyboard to keyboard controller.

Theory

There are two classes of counters in sequential circuit design: Asynchronous and Synchronous. The latter of the two has one distinct clock (i.e. all flip-flops within the counter are triggered by the same clock). This is not the case for the former. Synchronous counters are the preferred counter used in computers since the asynchronous counter has some timing skew which leads to issues when working together with synchronous logic.

Asynchronous Counters

A cascade of JK flip-flops can be used to design an asynchronous counter. As seen in previous sections of the textbook, when both inputs of a JK flip-flop are set to 'high' the flip-flop will enter toggle mode. When a flip-flop is in toggle mode it will toggle from 'low' to 'high' at the rising edge of clock for a positive edge triggered flip-flop. By taking the inverted output of one flip-flop (i.e. Q') and using it as an input to the clock for the next flip-flop it will cut the clock frequency in half. Therefore, the second flip-flop will only change values when the first flip-flop has toggled from 'low' to 'high' and back to 'low'. This behavior is precisely why this design works so great for binary counting.

Counters are able to count in a variety ways (i.e. BCD, binary, decimal, etc.). How they count all depends on how the flip-flops are hooked together. We will outline how asynchronous counters count in binary below. If you refer to Figure 4.3.4 below, which demonstrate up counting, whenever a transition from 'low' to 'high' is made for a significant bit a transition from 'low' to 'high' is made in the next most significant bit. This demonstrates that in the way the flip-flops are hooked up they can be used as a binary up counter.

0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1

Figure 4.3.4: Logic behind Binary up Asynchronous Counter

As you can see in the Figure 4.3.5 below, Q1 transitions from 'low' to 'high' when Q0' transitions from 'low' to 'high' (i.e. Q0 transitions from 'high' to 'low'). This is the case because, Q0' is hooked into the input line of the next flip-flop. This pattern continues with Q2 & Q1' and Q3 and Q2', which completes the four bit up counter.

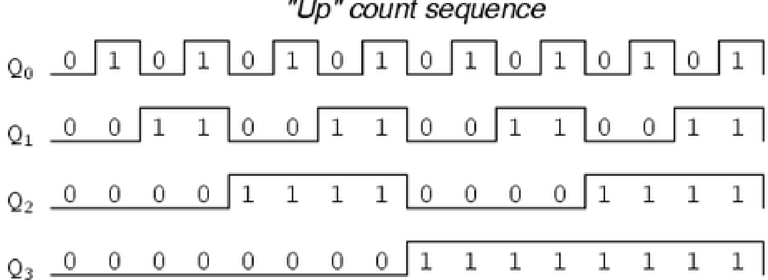


Figure 4.3.5: Up count sequence

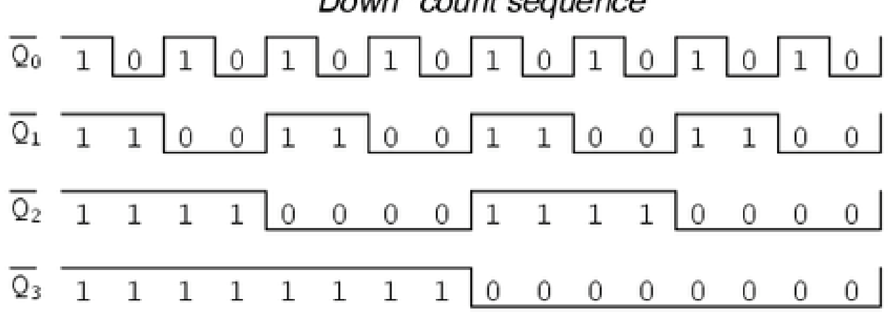


Figure 4.3.6: Down count sequence

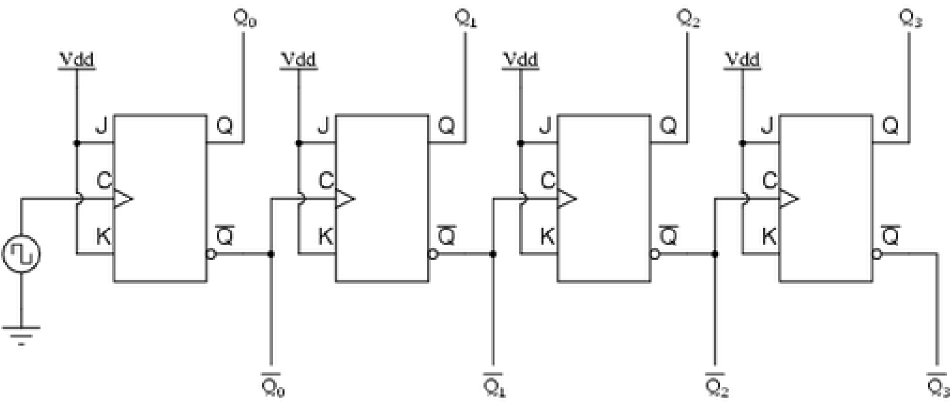


Figure 4.3.7: Circuit Diagram of the Up and Down Counter

Synchronous Counters

As the asynchronous four-bit up counter, the synchronous four-bit up counter is also designed using a cascade of JK flip-flops, although a few modifications are made. All of the clock inputs are this time connected together and therefore change states simultaneously. Also the following adjustments are made to the J and K inputs of the flip-flops:

I. FF0: J and K are set to 'high'
II. FF1: J and K are set to Q0
III. FF2: J and K are set to (Q0 AND Q1)
IV. FF3: J and K are set to (Q0 AND Q1 AND Q2)

The outputs of the synchronous counter are taken from Q. Just like the asynchronous up counter. The four-bit synchronous up counter exploits the fact that before a bit toggles, all preceding bits must first be 'high'. This is why the extra logic is needed in the design of the synchronous counter. Other than FF0 (which will always be in toggle mode) FF1-FF3 will only be in toggle mode when all preceding flip-flop outputs are 'high'. The flip-flops will therefore behave in the following manner:

I. FF0: Will always toggle with the clock
II. FF1: Will only toggle when Q0 is 'high'
III. FF2: Will only toggle when Q0 AND Q1 are 'high'
IV. FF3: Will only toggle when Q0 AND Q1 AND Q2 are 'high'

Therefore, the circuit will behave as a four bit binary up counter as outlined in the figure below.

0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1

Figure 4.3.8: Logic behind Binary up Synchronous Counter

When examining the four-bit synchronous down counter another useful observation can be made. Before a flip-flop will toggle all preceding bits must be set to 'low'. Therefore, the following adjustments are made to the inputs J and K of the flop-flops:

I. FF0: J and K are set to 'high'
II. FF1: J and K are set to Q0'
III. FF2: J and K are set to (Q0' AND Q1')
IV. FF3: J and K are set to ((Q0' AND Q1' AND Q2')

The flip-flops will therefore behave in the following manner:

I. FF0: Will always toggle with the clock
II. FF1: Will only toggle when Q0' is 'high'
III. FF2: Will only toggle when Q0' AND Q1' are 'high'
IV. FF3: Will only toggle when Q0' AND Q1' AND Q2' are 'high'

The outputs for the synchronous four-bit down counter are also taken from Q, similar to the four-bit synchronous up counter. The counter will now behave as a four-bit synchronous down counter as outlined in Figure 4.3.9 below.

0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1

Figure 4.3.9: Logic behind Binary down Synchronous Counter

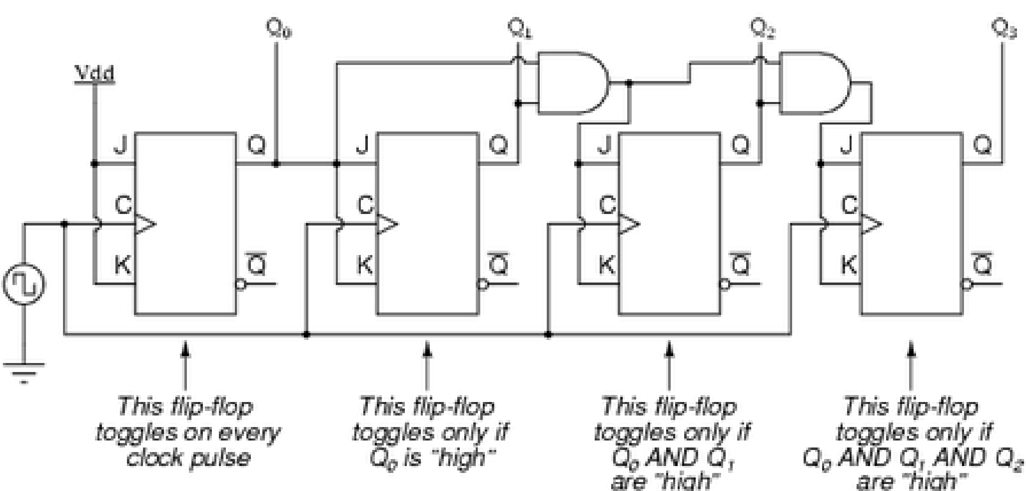


Figure 4.3.10: Circuit Diagram of the 4-Bit Synchronous Up Counter

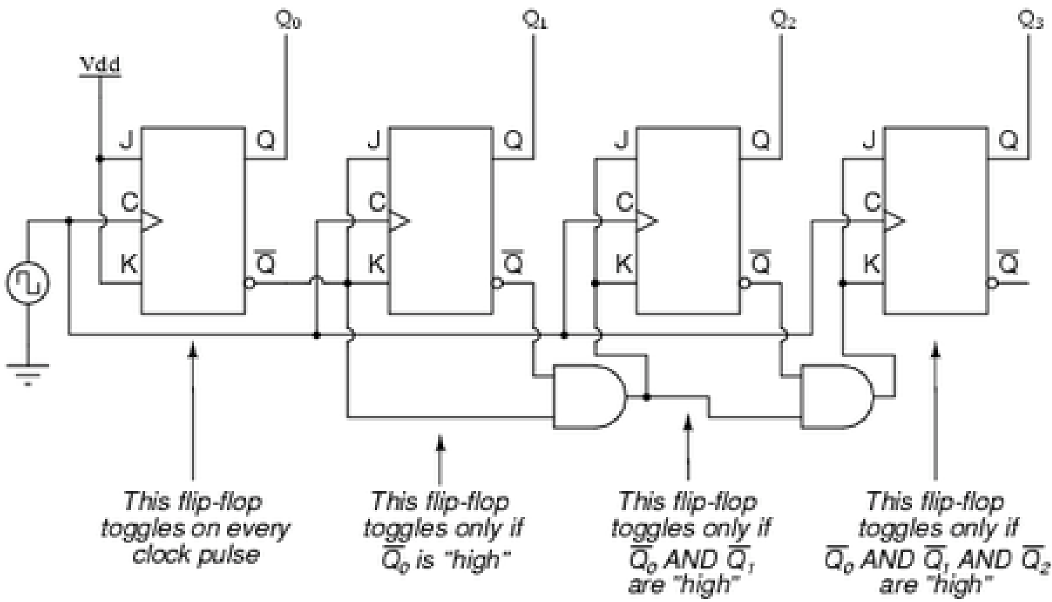


Figure 4.3.11: Circuit Diagram of the 4-Bit Synchronous Down Counter

4.4 Multiplication/Division – High Level

In general, multiplication and division circuits are complicated and will only be introduced in this textbook. Extensive details will not be provided.

Multipliers as the name implies take two values, be they binary or decimal, and multiplies them together. The process of multiply digital values (binary) is very similar to that of decimal multiplication. They are used in digital process systems (DSP), some applications include digital filtering, digital communications and spectral analysis.

Decimal Multiplication

```
      123
x     264
      492
     7380
    24600
    32472
```

Binary Multiplication

```
Multiplicand        1011
Multiplier    x     0110
                    0000
                   10110
                  101100
                 0000000
Product          1000010
```

The process involves shifting the multiplicand and adding zero to that multiplicand. Each multiplier bit determines whether a zero is added or a shift version of the multiplicand.

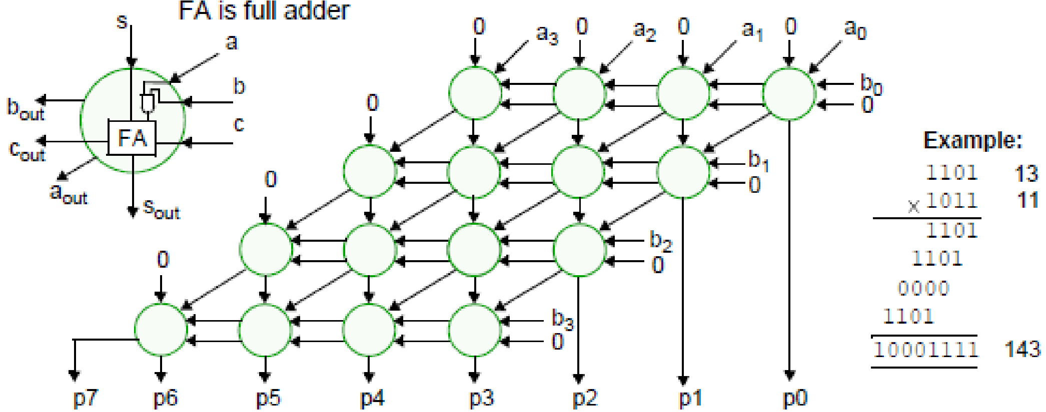


Figure 4.4.1: Example of Binary Multiplication with the Circuit Model

The AND gate in the Full Adder performs the selection for each bit. The AND Gate is the bit multiplier. The above design does not work for signed two's complement numbers.

There are many types of sequential circuits that accomplish multiplying:

- Shift and Add Multiplier
- Bit-serial Multiplier
- Array Multiplier

Shit and ADD Multiplier

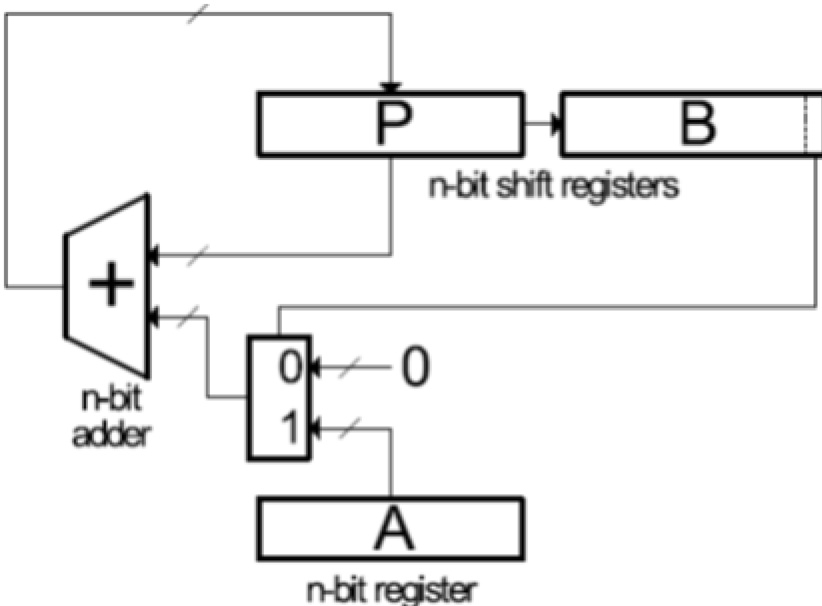


Figure 4.4.2: Shit and ADD Multiplier block Diagram

The partial product is summed, one at a time. Each partial product is shifted versions of A or 0.

Signed multiplication:

Two’s complement: MSB (most significant bit) has negative weight.

For multiplication

a) Subtract final partial product

b) Sign-extend partial products

Modifications to shift and circuit

a) Add/Subtract

b) Sign-extender on P shifter register

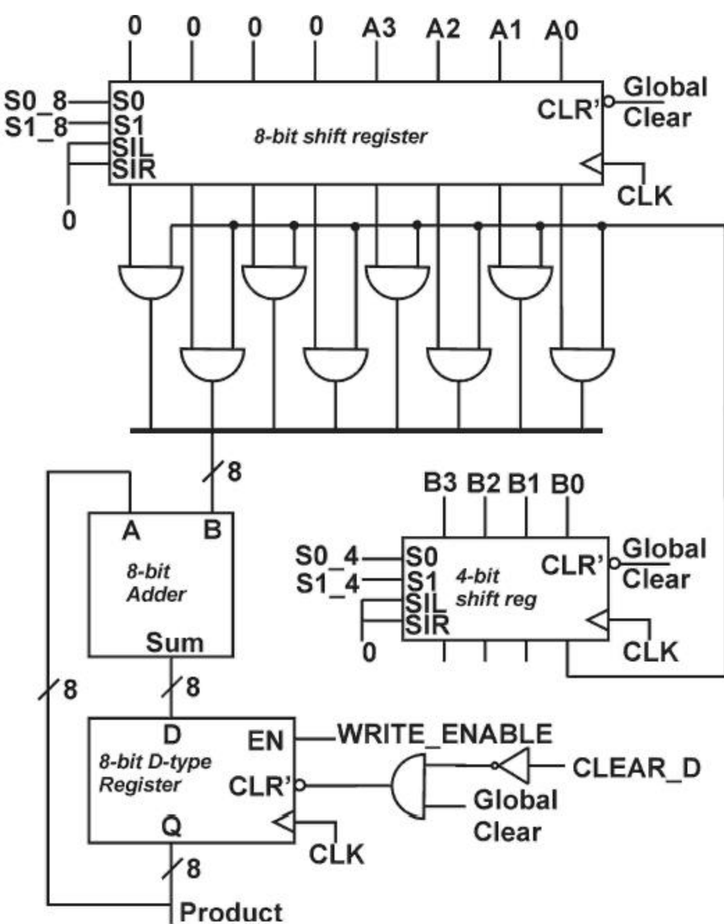


Figure 4.4.3: Circuit Diagram of the Multiplier

Division

The “comparison requires a subtract, the sign of the results is examined; if the result is negative, the divisor must be added back

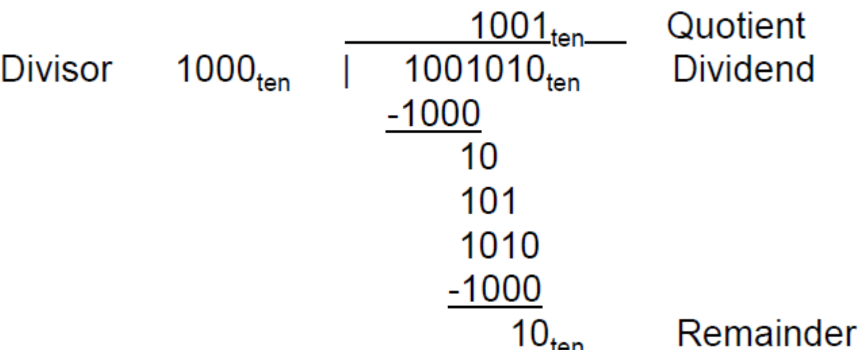


Figure 4.4.4: Example of Division

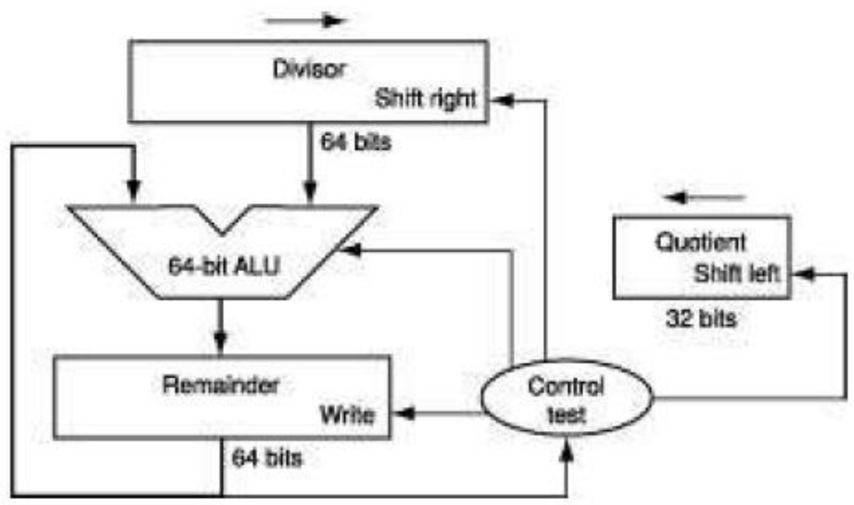


Figure 4.4.5: Block Diagram of Division

4.5 Comparator

In general, comparator circuits are complicated and will only be introduced in this textbook. Extensive details will not be provided.

Binary comparators, which are also called digital comparators or logic comparators, are combinational logic circuits that are used for checking whether the value represented by one binary word is greater than, less than, or equal to the value represented by another binary word. It applies the same principles as equality/inequality comparisons that are performed in everyday arithmetic. There are two basic types of comparators that are most commonly implemented in the digital circuits.

- Equality comparators.
- Magnitude comparators.

Equality Comparators

An equality comparator circuit that is shown in Fig 4.6.1 below is the simplest multibit logic comparator. This equality comparator could be used for real life applications like electronic locks and security devices where a binary password consisting of multiple bits is inputted to the comparator and has to be compared with another preset word.

In Figure 4.6.1, a logic 1 (high) will be present at the output if the two inputs match, otherwise the output remains at 0. Therefore there is only one input combination that is correct, and the more bits the input words possesses, the more possible wrong combinations there are. The circuit of the equality comparator consists of an exclusive NOR gate (XNOR) per pair of input bits. If the two inputs are the same, either both are 1s or both are 0s, an output of logic 1 is obtained.

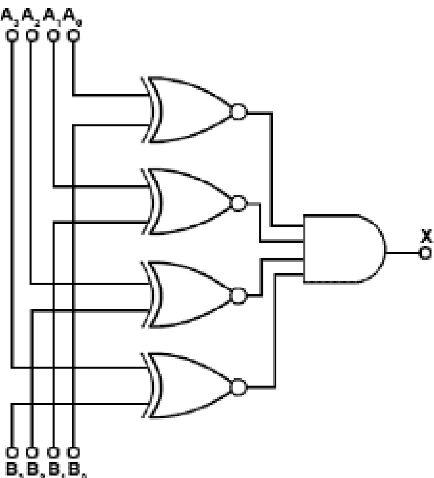


Fig. 4.5.1: Four Bit Equality Comparator Circuit

The outputs of the XNOR gates are then combined in an AND gate, the output of which will be 1, only when all the XNOR gates indicate matched inputs.

Magnitude Comparator

Magnitude comparator is a combinational circuit that compares two numbers and determines their relative magnitude. The magnitude comparator indicate equality, but has further two outputs, one that is logic 1 when word A is greater than word B, and another that is logic 1 when word A is less than word B. Magnitude comparators therefore, form the basis of decision making in logic circuits. Any logical problem can be reduced to one or more yes/no decisions based on a pair of compared values.

A comparator block diagram is shown in figure 4.6.2. The output of comparator is usually 3 binary variables indicating:

I. A>B
II. A=B
III. A<B

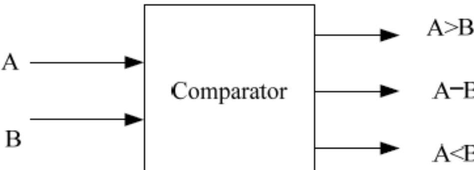


Fig. 4.5.2: Block Diagram of Magnitude Comparator

The Circuit diagram of the above comparator is shown in Figure 4.6.3 below. Gate 1 produces the function A>B and gate 3 gives A<B, while gate 2 is an XNOR gate giving an equality output. This basic circuit for a magnitude comparator may be extended for any number of bits but the more bits the circuit has to compare, the more complex the circuit gets.

We shall now construct a 2-Bit comparator by analyzing the logic required to construct the design. Below is the block diagram of a 2-Bit comparator.

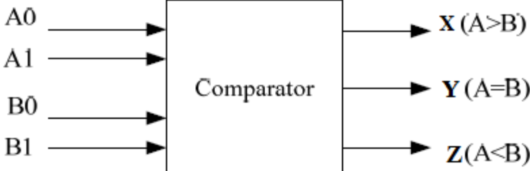


Figure 4.5.3: Block Diagram of 2-Bit Comparator

We shall imply a simpler method to find the three different cases shown above, X, Y and Z.

Case 1: We know that A=B if all Ai= Bi

Ai	Bi	Xi
0	0	1
0	1	0
1	0	0
1	1	0

Table 4.5.1: Truth Table for Case 1

It means $X0 = A0B0 + A'0B'0$ and $X1 = A1B1 + A'1B'1$

If X0=1 and X1=1 then A0=B0 and A1=B1

Thus, if A=B then X0X1 = 1 it means $X = (A0B0 + A'0B'0)(A1B1 + A'1B'1)$ since $(x \oplus y)' = (xy + x'y')$

$X = (A0 \oplus B0)' (A1 \oplus B1)' = ((A0 \oplus B0) + (A1 \oplus B1))'$

It means for X we can NOR the result of two exclusive-OR gates.

Even though this analysis is simple, it is beyond the scope of this textbook. Case 2 and Case 3 we derived by following a similar approach. The above magnitude comparator can be constructed by the logic circuit below.

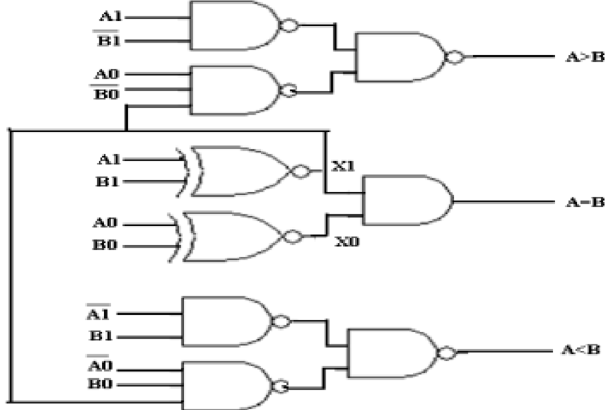


Figure 4.5.4: Circuit Diagram of 2-Bit Comparator

The circuit diagram of a 4-Bit Magnitude Comparator is much more complicated. It is illustrated in the figure below. The circuits, equations and theory in this section are referenced to Dong. X., Peng. M., Al-Khalili. A. (2015). Design of a 4-Bit Comparator. Project Report for COEN6511: ASIC Design. Department of Electrical and Computer Engineering. Concordia University, Montreal, Quebec

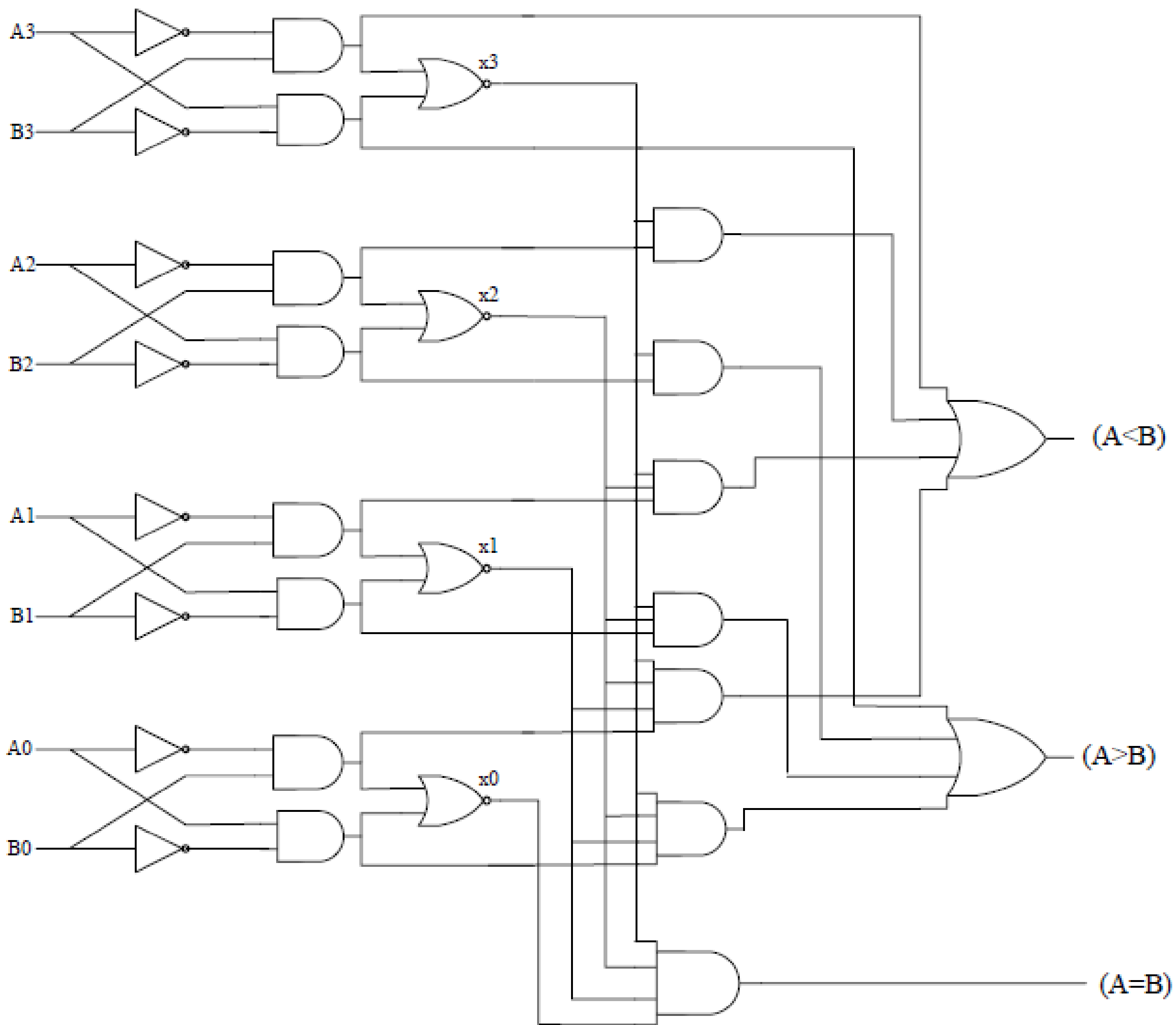


Figure 4.5.5: Circuit Diagram of 4-Bit Magnitude Comparator

4.6 Registers: Shift Registers

The computer processor is made up of many different components. The main component we will be focusing on for this section is the register. Registers are a type of sequential logic circuit with a main purpose of storing digital data. A register may hold instructions that come in differentiating sizes. For example, a 32-bit instruction computer must have a register 32 bits in length in order to hold a large enough instruction. Even though the Nibble knowledge kit have a different set of bit instructions, registers are still used in the same way and for the same purpose. Furthermore, there are many different computer designs. Some computer designs may only require half registers for shorter instructions. This all depends on the computer and processor design. You will learn more about the 4-bit computer and its processor design in advanced sections of this book.

Registers act like the memory system. They are the top of the memory hierarchy and can effectively speed up the manipulation of data. A single register consists of a group of flip-flops and gates. The flip-flops contain the information and the gates control how new information moves into the register. Figure 4.6.1 displays how a 4-bit register is composed of 4 flip-flops that is triggered by the common clock input.

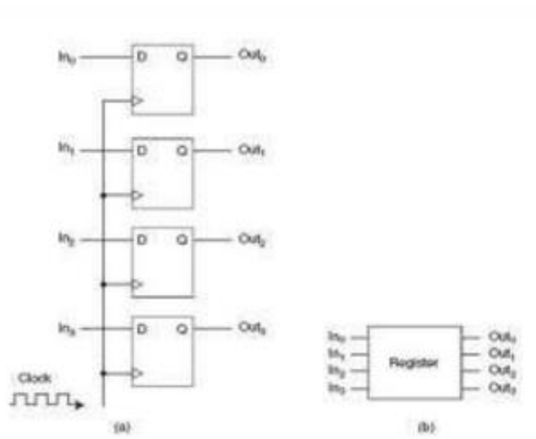


Figure 4.6.1: Circuit Diagram of a 4-Bit Register

Parallel Load Register

The transfer of new information into a register is referred to as loading the register. If there is a common clock pulse and all the bits of the register are loaded simultaneously, then the loading is done in parallel. Load input determines whether or not the next clock pulse will leave the existing information in the register or it will accept new information.

Shift Registers

Shift registers consist of a group of flip-flops set in a linear fashion that consist of inputs and outputs connected together that data shifts down the line when the circuit is activated. The simplest shift register uses only flip-flops. Figure 4.6.2 displays how data moves down in a linear fashion and still follows a common clock pulse. In this case the data will be shifted right from the least significant bit ‘Q1’ to the most significant bit ‘Q4’. Register can have different configuration to shift left or even in both direction as we are going to learn later in this section.

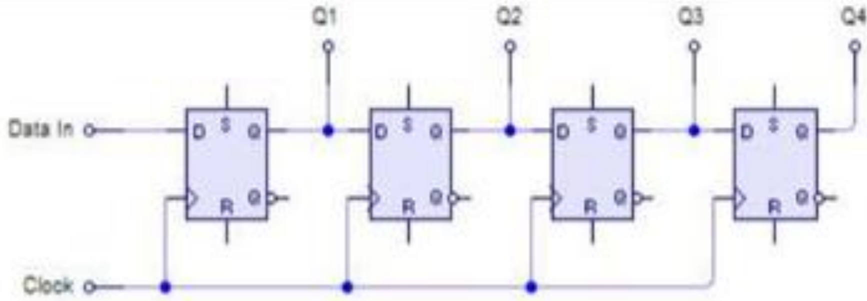


Figure 4.6.2: Data flow through a shift register

General shift registers have many capabilities such as; parallel load, control state, input for clock pulse, shift right and shift left operations. A serial input determines what enters into the furthest to the left position during the shift. Serial output is taken from the output of the flip-flop furthest to the right. It is important to note that the clock remains common to all flip-flops. A unidirectional shift registers is capable of shifting in only one direction. Therefore, bidirectional shift registers can shift in both directions (left and right). Figure 1.3 displays a bidirectional shift register.

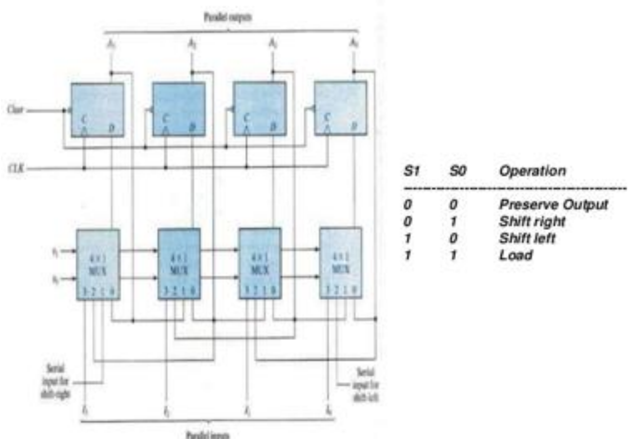


Figure 4.6.3: Bidirectional Shift Register with Parallel Load

4.7. ALU

The two fundamental components of a CPU (computer processing unit) are the control unit, and the ALU. The control unit basically directs operations within a computer processor and decodes instructions into commands for the ALU and determines the sequence of operations.

An ALU stands for Arithmetic-Logic Unit. It is a component of the computer processor that carries out arithmetic and logic operations on the operands and combines them into a single unit as computer instruction words.

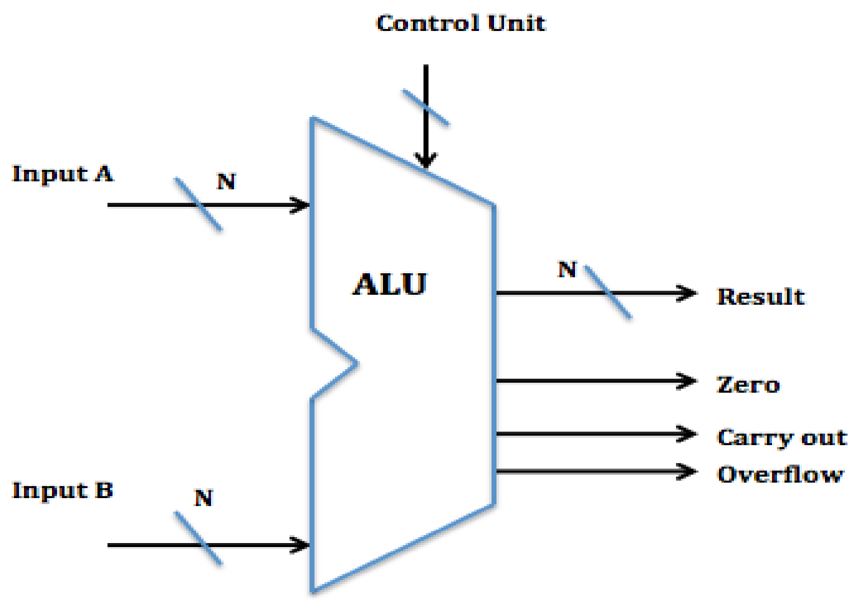


Figure 4.7.1: Block Diagram of the ALU

The components of an ALU consists of:

Inputs A/B: These are data inputs that are used in the ALU computation each input is N bits in size.

Result: The result of the ALU computation is an output value of N bits.

Control Unit: The control unit sends a control signal to the ALU to tell it which operation to perform.

The following are known as status flags:

Zero: Indicates if the result of an operation is zero, by determining if two values are equal.

Carry out: Indicates if the operation resulted in a carry.

Recall that a carry can occur if the result is greater than or equal to 2^{32}

Overflow: Indicates that the result of an operation is too large.

Recall that an overflow occurs if the result is greater than or equal to 2^{31} or less than -2^{31}.

The first step in determining the ALU operation is to understand the different control signals that indicate the function to be performed.

For example, the following control signals may be used as input to the ALU to instruct it on which function to perform:

Function	Control Signal
AND	000
OR	001
ADD	010
SUBTRACT	110

Table 4.7.1: Control Signals for functions AND, OR, ADD and SUBTRACT

Next comes the combination of multiplexers and adders to design basic 1-Bit ALU's.

The multiplexor is used to choose from one of the two inputs coming into the ALU based on the control signal, and outputs the result.

4.7.2 1-Bit ALU: AND, OR and Addition

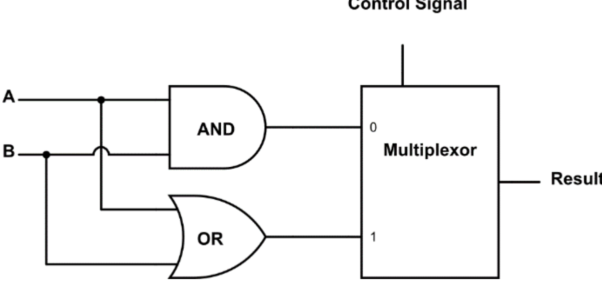


Figure 4.7.2: 1-bit ALU for logical functions AND and OR

Here, the control signal has two possible inputs:

Function	Control Signal
AND	0
OR	1

Table 4.7.2: Control Signals for functions AND & OR

The signal from the control unit determines whether the result (output) will come from the AND gate or the OR gate. For ADD and SUBTRACT arithmetic functions we need to implement an

adder circuit. The addition and Subtraction circuits have been explained in detail in the previous sections above, 4.1 and 4.2. In order to design an ALU that performs multiple operations we simply need to change the type of multiplexer being implemented to select the appropriate result. As such a 1-Bit ALU that performs three functions such as AND, OR, and ADD will implement a 4-input multiplexer that outputs a single result. The control signals for this ALU can be assigned as follows:

Function	Control Signal
AND	000
OR	001
ADD	010
SUBTRACT	011

Table 4.7.3: Control Signals for functions AND, OR, ADD, and SUBTRACT

The result is the following 1- bit ALU:

Control Signal
A
B
Carry In
AND
OR
000
Multiplexor
001
010
Result
+
(1-Bit Adder)
Carry Out

Figure 4.7.3: 1-bit ALU that performs AND, OR and addition

4.7.3 4- Bit ALU: Overflow and Zero Flag Detection

From section 2.3, you may recall that overflow occurs when the size of the inputs is such that there is a carry, which changes the most significant sign bit. As such, overflow indicates when the sign of the result is different from the sign of the inputs.

Overflow detection is for add/subtract operations, as it is bound to occur if the addition or subtraction result does not fit into the n-bits of the final result output of the ALU.

To detect overflow a XOR gate is added to the last adder of an n-bit ripple carry adder. Its compares the carry in to the carry out values and detects overflow if they are different.

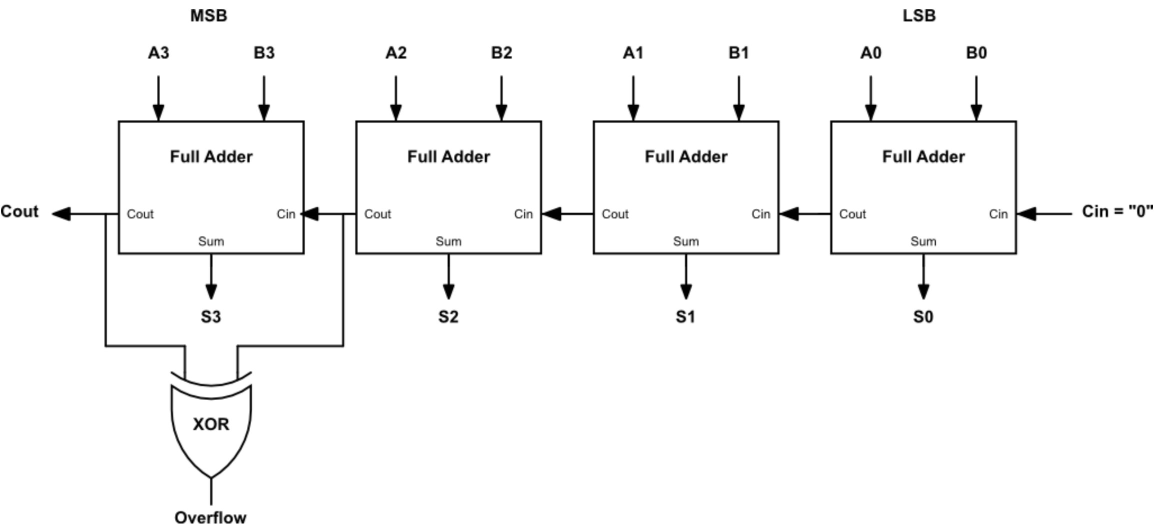


Figure 4.7.4: 4-bit ripple carry adder with overflow detection.

Example: The XOR gate has overflow detection with addition operation

Case	Cin	Cout	Sum
Add two negative numbers (A_{n-1}= 1, B_{n-1}=1)	0	1	0
Add two positive numbers (A_{n-1}= 0, B_{n-1}=0)	1	0	1

Table 4.7.4: XOR gate overflow detection addition cases; 0 = positive, 1 = negative

Zero detection basically checks the result and determines if all the result bits are zero. If the result is zero, i.e. all bits are zero, then the OR gate will output a 0 and the NOT gate will invert that result to a 1, to indicate a zero has been detected. If the result is not zero, i.e. not all bits are zero, then the OR gate will output a 1 and the NOT gate will invert that result to a 0, to indicate a zero has not been detected.

The implementation of zero detection is shown in the figure below.

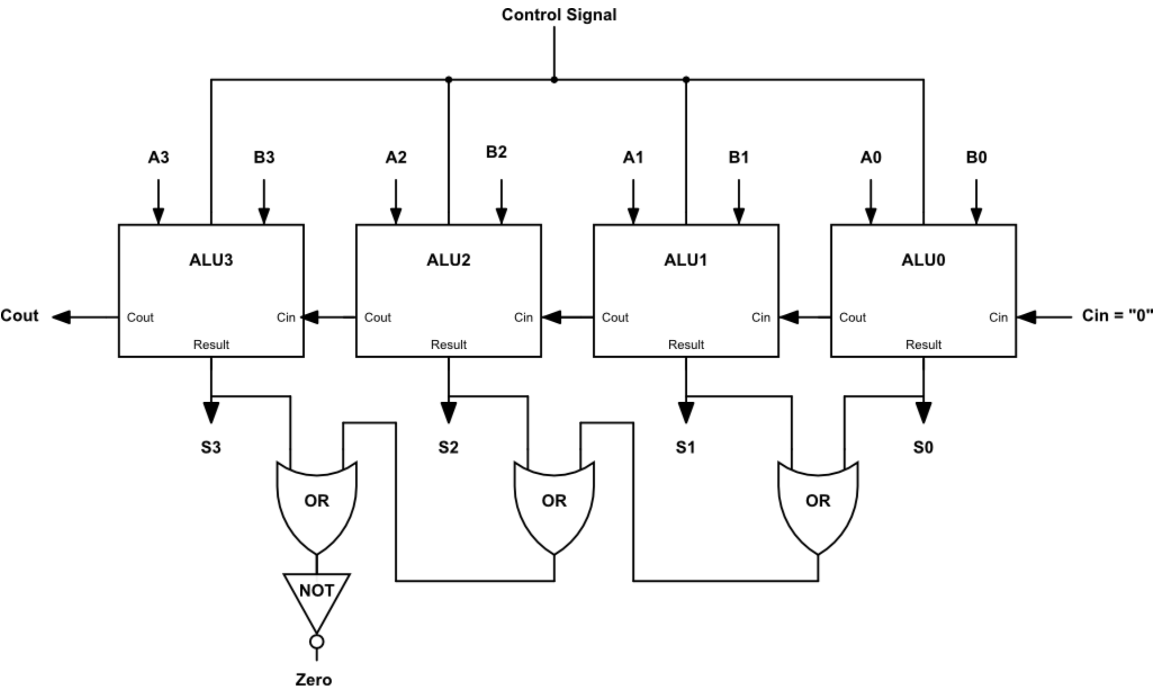


Figure 4.7.5: 4-bit ALU with Zero detection.

4.7.4 The Nibble Knowledge ALU

The ALU or Arithmetic Logic Unit is where instructions are carried out in the CPU. It firsts determines what to do with the data as specified by the OP code and performs the required arithmetic and sends out the proper control signals to various parts of the CPU.

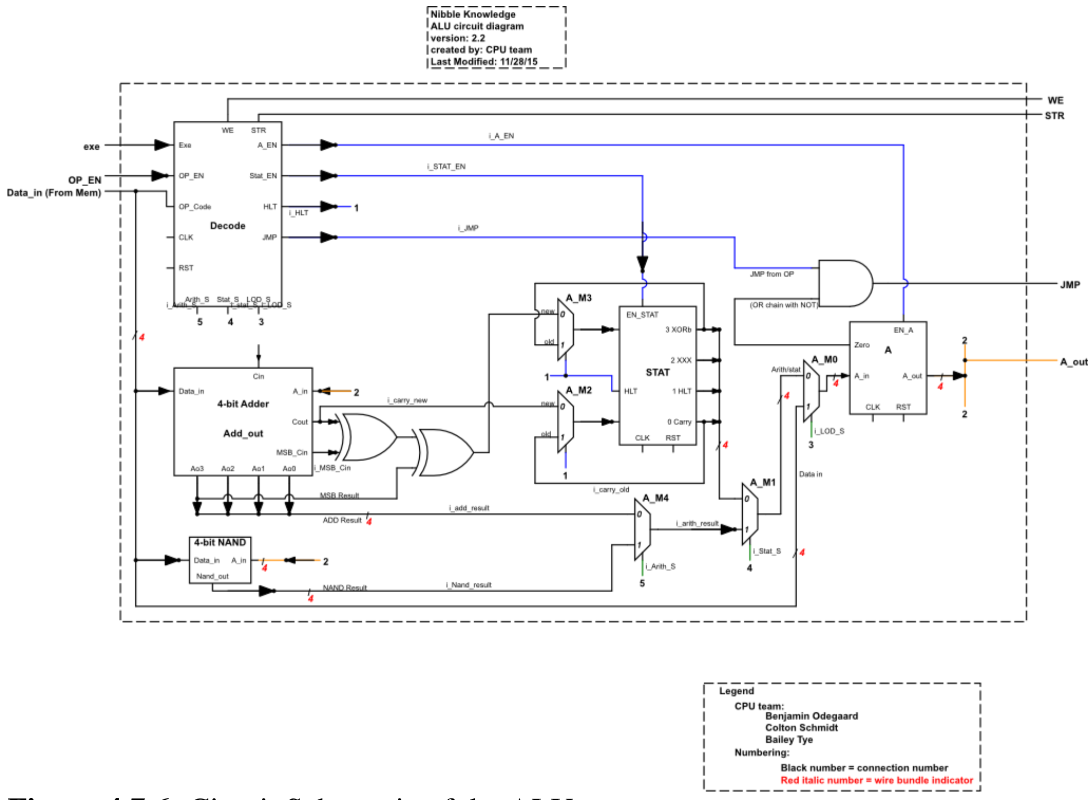


Figure 4.7.6: Circuit Schematic of the ALU

ALU Decode

The ALU decode is what tells the rest of the ALU components what to do. On the first cycles of the 6 cycles it takes to run an instruction, the opcode is saved inside the decode unit (this will be on the data in line from main memory). The chart below shows which signals each instruction requires. The only clock driven components in the ALU are the A, opcode (inside the ALU decode), and STAT registers. Everything is combinational and will basically be running even when the decoder signals are not specifically assigned. During cycle 6, the execute phase enables and select signals are turned on based on the opcode, and the instruction is complete.

Instruction	WE	HLT	A_EN	STAT_EN	JMP	Arith_S	Stat_S	LOD_S
HLT	0	1	0	0	0	X	X	X
LOD	0	0	1	0	0	X	X	1
STR	1	0	0	0	0	X	X	X
ADD	0	0	1	1	0	0	0	0

NOP	0	0	0	0	0	X	X	X
NND	0	0	1	0	0	1	0	0
CXA	0	0	1	0	0	X	1	0
JMP	0	0	0	0	1	X	X	X

Table 4.7.5: Low Level Design of Top CPU Module

Using the logic developed in above table, equations were developed to create the right signals out of logic for implementation. These equations are:

Aen = C'	JMP = ABC'
Sen = A + B' +C'	ArithS = A
HLT = A + B + C	StatS = AB
WE = A + B' + C	LodS = A'B'

These are the equations used in the implementation of the ALU

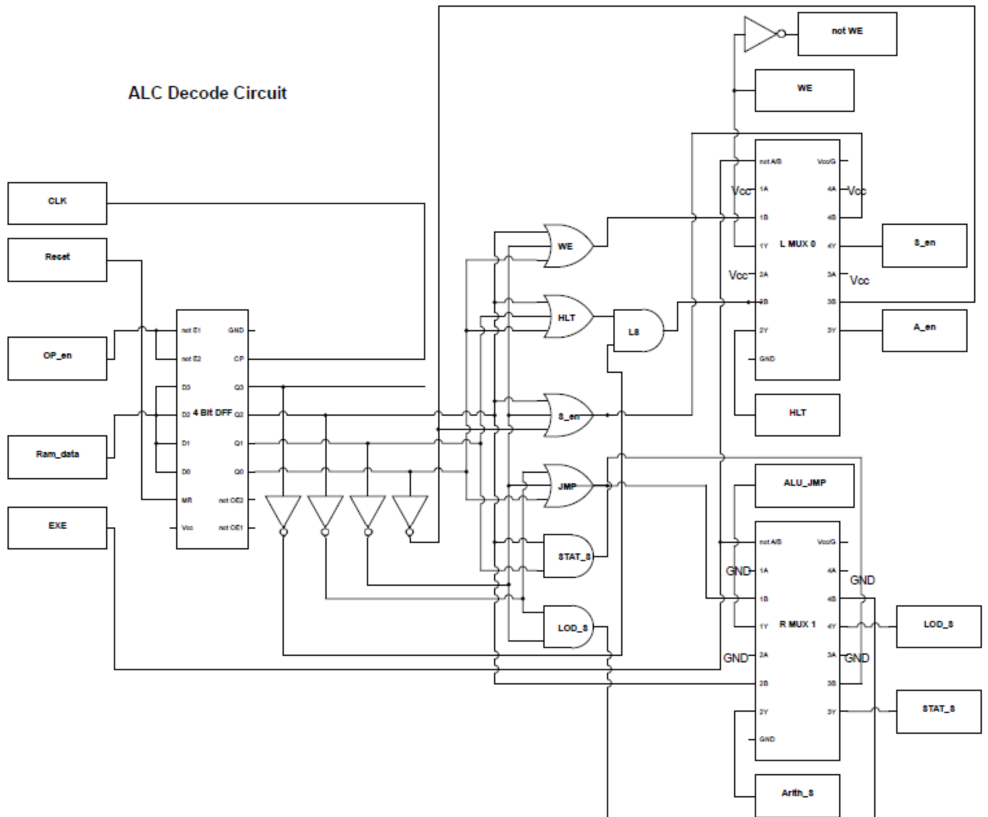


Figure 4.7.7: ALU Decode Circuit Schematic

ALU Logical/Arithmetic Operations

The ALU is capable of two operations: a bitwise NAND of the A register with a memory location, and a 4-bit add. These are combinational circuits as stated above, so they will be running even if the instruction does not require them.

MUXs/XORs/ANDs

The two XOR gates in the middle of the ALU are used for overflow detection. Even though there is only an ADD, and most operations will be unsigned, the overflow detection for comparisons was implemented. After which, XOR the overflow bit with the MSB of the sum and this will determine whether a number is greater than the other, or less than.

The Bus Interface and Memory

The IO Board holds the CPU's main memory and the registers for the memory mapped I/O addresses. This means that this section manages the RAM and the input and output to the bus. This is accomplished by first taking the 16 bit address line coming into the block and determining if it was in the lowest three addresses by a chain of NAND gates the output of this logic block is then used as the enable for a decoder of which the bottom two bits are sent as inputs. This causes the right DFF corresponding to the bottom two bits to be enabled if the top 14 bits of the address line are '0' else they are disabled from propagating. Whether the address is in a memory IO spot the data is always sent to the RAM chip no matter what. To get data from the I/O board the same bottom two address bits are sent to a 4 to 1 multiplexer that selects either the Data register, Stat register, Chip Select, or the RAM chip to output back to the CPU.

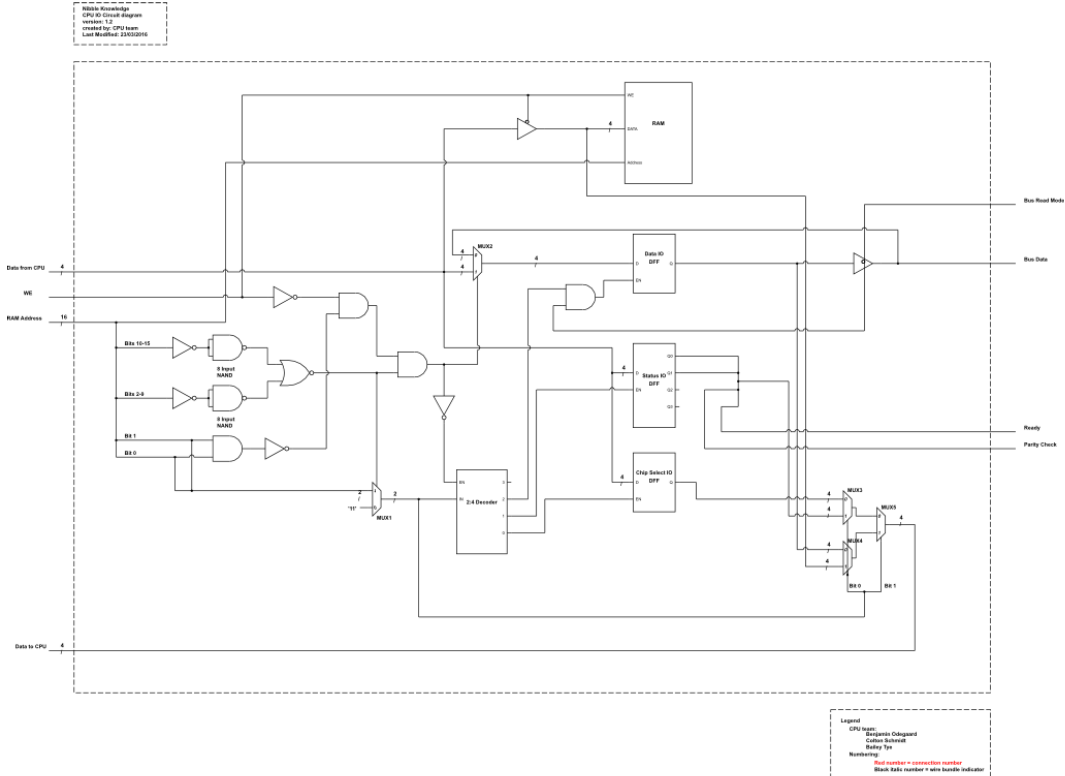


Figure 4.7.8: Circuit Schematic of the CPU I/O

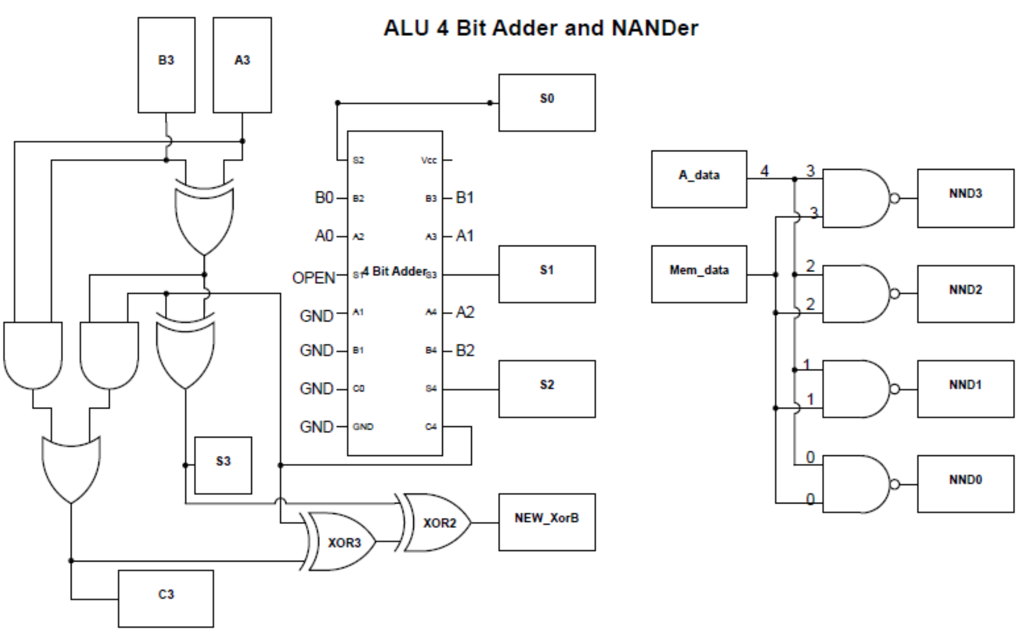


Figure 4.7.9: ALU 4-Bit Adder and NANDer Circuit Schematic

5. Computer Architecture

5.1 Instruction Set

The formal definition of the instruction set is that it is a group of commands for a CPU in machine language. The term can refer to all possible instructions for a CPU or a subset of instructions to enhance its performance in certain situations. The language that the computer understands is the instruction set and the operators and operands are the ALU, registers and memory of the computer respectively. It is wise to study the computer's language (Instruction Set) to understand the computer's architecture. To describe it as an analogy, the vocabulary is the instruction set and the words are the instructions. Each and every single program that is running on the computer uses the same instructions and the instruction set. The instruction set provides commands to the processor, to tell it what it needs to do. The instruction set consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external I/O. All CPUs have instruction sets that enable commands to the processor directing the CPU to switch the relevant transistors. However, the instruction set of the Nibble Knowledge Computer is simpler. The Nibble Knowledge computer has been designed with hardware simplicity in mind. The RAM of the CPU is very small compared to your average computer. The RAM is 65536 nits in size and the memory cell size is 4 bits, a nibble, hence the name Nibble Knowledge.

Registers

The Nibble Knowledge computer has only four different registers, which hold the addresses of instructions, data and information for arithmetic operations. Registers hold values for the CPU to use for various purposes. Modern day processors have many of these registers for convenience in computation and programing. The Nibble Knowledge computer however, only has a few essential registers. The table below describes the registers.

Register	Width (bits)	Comments
PC	16	Program counter; holds current address of next instruction to be loaded
A	4	Accumulator register used for arithmetic and other operations
MEM	16	Holds the address being used by an instruction at any given time.

STAT	4	layout: (signed overflow bit) XOR with MSB of the A register, EMPTY, HLT, Carry bit

Table 5.1.1: Registers and their operations

Instruction Set

The table below describes the instruction set of the Nibble Knowledge computer. These eight instructions are supported by the architecture of the CPU and their machine code format and description. You will later see how this instruction sets determines the layout of the entire CPU in chapter 7 on CPU.

Instruction	**Format**	**Description**
HLT	0000-0000-0000-0000-0000	Halts the CPU
LOD	0001-16-bit address to load from	Loads a value from location specified into the A register
STR	0010-16-bit address to store to	Stores the value in the A register into memory location specified
ADD	0011-16-bit address to add from	Adds the value at the location specified, plus the existing carry flag, into the value in A. Sets carry and overflow flags as required
NOP	0100-0000-0000-0000-0000	No operation
NND	0101-16-bit address to NAND from	Bitwise NANDS the value at the location specified into the value in A
JMP	0110-16-bit address to jump to	Jumps to location specified by setting the program counter to that address
CXA	0111-0000-0000-0000-0000	Copies the STAT register into the A

Table 5.1.2: Machine Code Format and Description of Instruction Set

The Instruction Set above, could further be developed into a microinstruction set. The table below describes the microinstruction set of the computer.

Instruction	Type of Micro	Comments
A->IR	TransferRtoR	Transfers the contents of A into IR

Carry->A0	TransferRtoR	Transfers the carry bit into the A and shifts it to the LSB
IR->A	TransferRtoR	Transfers IR into A
IR->MEM1	TransferRtoR	Moves bits from the IR to the LSBs of MEM register (0 to 3)
IR->MEM2	TransferRtoR	Moves bits from the IR to the LSBs of MEM register (4 to 7)
IR->MEM3	TransferRtoR	Moves bits from the IR to the LSBs of MEM register (8 to 11)
IR->MEM4	TransferRtoR	Moves bits from the IR to the LSBs of MEM register (12 to 15)
MEM->PC	TransferRtoR	Moves the contents of MEM into PC
XOR->XORb	TransferRtoR	Transfers the 0 bit of IR into the 0 bit of STAT (Used after XOR of Carry and MSB in A)
XORb->A3	TransferRtoR	Transfers the XORb bit (3 in STAT) into bit 1 of A
Zero->A1	TransferRtoR	Transfers the halt (which will be zero if running) into bit 1 or A
Zero->A2	TransferRtoR	Transfers the halt (which will be zero if running) into bit 2 or A
Load	MemoryAccess	Loads the 4 bit value stored at address “MEM” into the A
Load Instruction	MemoryAccess	Loads 4 bit value stared at address “PC” into IR
Store	MemoryAccess	Stores the 4 bit A value into address location “MEM”
Add*	Arithmetic	Adds the 4 bit value stored at IR to the 4 bit value in the A and assigns the carry bit in STAT if necessary
Halt	SetCondBit	Sets the halt bit in STAT to 1
PC+1	Increment	Adds 1(cell) to the current address in PC
Not	Logical	Negates the current value in A (NANDs A with itself)
NAND*	Logical	NANDs IR and A and stores result in A

XOR	Logical	XORs STAT and A to get the XOR of carry bit and MSB of A
If A ==0	Test	If A is 0000 then skip the next 14 microinstructions (See JMP)
Decode	Decode	Decodes the OPcode (value in IR) of the instruction

Table 5.1.3: Description of Microinstruction Set

For the microinstructions, it is always assumed that the programmer loads a value into a register beforehand, or, the value must already be in the A register. How to use these instructions is described in section 5.2 below.

5.2 Assembly Language

In simple terms, assembly language is the readable version of machine language that humans can understand. The assembly language is composed of instructions that specify the operation to perform and where to conduct that operation. An Assembler, which is program in the computer that takes the assembly language and converts it into machine code that the computer will understand and execute. The Nibble Knowledge Computer uses an assembly language that was developed only for the Nibble Knowledge CPU.

Instructions

Instructions are like the words used to describe an operation.

For example, if you wanted to subtract two number, say X – Y = Z, then in English, you would say, Z equals the number that is a result of Y being subtracted from X. In assembly language, the translation is similar.

Subtraction of X – Y = Z in Assembly Language is:

```
LOD   addressY      ; Put Y into A
NND   n15           ; Negate Y
ADD   n1            ; Add 1 to Negated Y (Finished Two's Compliment)
ADD   addressX      ; Add X to Negated Y (X-Y)
```

Implementation of the Subtraction Instruction with X – Y = Z is as follows:

1. Load value Y into A from memory
2. Negate the Bits in A (flip Y)
3. Add 1 to A
4. Add value X from memory to A
5. The Value in A should now be Z and the Carry and XOR Bit of the STAT Register Set

One may read the assembly into English equivalent as “put Y into memory location A, negate y from it, then compute the binary subtraction by adding 1 to the negated Y, which is doing the Two’s Complement (discussed in section 2.3 above and reviewed below), and then add X to the negated Y value.

Two's Complement Comparison

Comparison is achieved by subtracting the two values in question.

- If the subtraction equals zero the values are equal
- If the result is non-zero, XOR the MSB of the result and the carry bit. If the XOR is 1 the subtraction was less than zero resulting and when the XOR is 0 the result is positive

Example: Two's Complement Comparison

Implementation of Comparison: Comparison of X & Y

1. Transfer value X to the IR from the A
2. Load value Y into register A
3. Negate the bits in register A (flip Y)
4. Add 1 to A
5. Add the IR to A (X + -Y)
6. XOR Bit 3 of A and Bit 0 of Stat
7. Store Result in Bit 3 of Stat
8. Check the XOR bit (Bit 3 of Stat)
9. If the XOR bit is 0   X >= Y
10. If the XOR bit is 1   X <   Y

Comparison of X & Y in Assembly:

```
LOD   addressY    ; Put Y into A
NND   n15         ; Negate Y
ADD   n1          ; Adds 1 to Negated Y (Finished Two's Compliment)
ADD   addressX    ; Add X to Negated Y (X-Y)
CXA               ; Copies Carry and XORb to A
NND   n8
NND   n15         ; ANDs with 0010 to remove carry bit and leave XORb
JMP   TAG         ; Jumps to tag if X >= Y else continue
```

Logic Functions

Here, it must be assumed that the value of variable X is already in register A.

NOT X = X’

1. NAND with n15 in data table
2. The value in A is now X’

Assembly:

```
NND   n15      ; X = X’
```

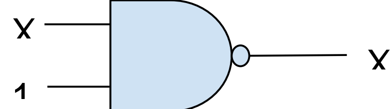


Figure 5.2.1: NOT X = X’

X AND Y = Z

1. NAND with the address of Y
2. NAND with n15 in data table
3. The Value in A is now Z

Assembly:

```
NND   addressY        ; NAND with Y
NND n15               ; Negate A = Z
```

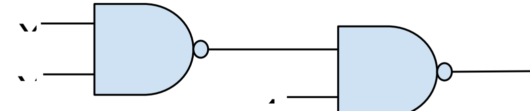


Figure 5.2.2: X AND Y = Z

X OR Y =Z

1. NAND with n15 in data table
2. The value in A is now X’
3. Save X’ into memory
4. Load Y into A
5. NAND with n15 in data table
6. The value in A is now Y’
7. NAND with the address that stores X’
8. The value in A is now Z

Assembly:

```
NND n15            ; X = X’
STR    addressX’   ; Store X’ in Mem
LOD    addressY    ; Load Y
NND    n15         ; Y = Y’
NND    addressX’   ; Y’ NAND X’ = Z
```

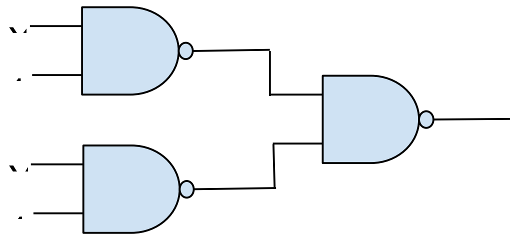


Figure 5.2.3: X OR Y = Z

X XOR Y = Z

1. Save X into memory
2. NAND with the address of Y
3. The value in A is now intermediate value 1
4. Save the intermediate value into memory

5. NAND with the address that stores X
6. The value in A is now intermediate value 2
7. Save intermediate value 2 into memory
8. Load intermediate value 1 into A
9. NAND with the address that stores Y
10. The value in A is now intermediate value 3
11. NAND with the address that stores intermediate value 2
12. The value in A is now Z

Assembly:

```
STR   addressX        ; Store X into Mem
NND   addressY        ; X NAND Y = Inter1
STR   addressInter1   ; Store Intermediate 1 in Mem
NND   addressX        ; Inter1 NAND X = Inter2
STR   addressInter2   ; Store Intermediate 2 in Mem
LOD   addressInter1   ; Load Intermediate 1 into Mem
NND   addressY        ; Inter1 NAND Y =Inter3
NND   addressInter2   ; Inter3 NAND Inter2 = Z
```

Note: Carry into MSB XOR with carry out XOR with MSB of sum

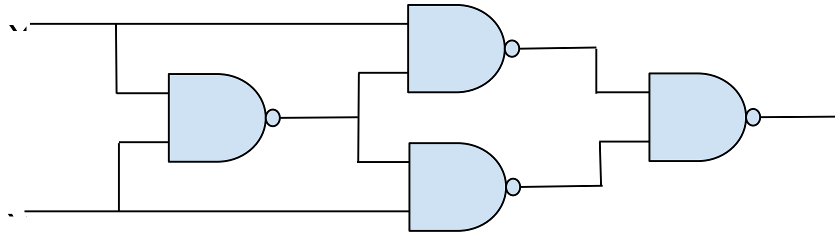


Figure 5.2.2: X XOR Y = Z

There are endless possibilities of what can be done with the Assembly language. With the assembly language, the programmer is in command of the processor running the computer.

6. Nibble Knowledge Computer High Level Design

The project encompasses the design of a 4-bit computer system that holds an educational value, where an individual would be able to learn the internal functionalities of a CPU. The high level design is divided into four subsections to clearly explain the implementation of the preferred solution. This will enable us to show how the various components of the solution are related and interact with each other.

Overall Design: CPU and Connected Devices

The overall design consists of CPU, Memory, Peripheral Controllers, Peripheral Devices and a Peripheral Bus. The five peripheral devices are the Serial Communication RS232 Controller, VGA Monitor Controller, IDE Hard Drive Controller, PS/2 Keyboard Controller and the Audio Controller. The Serial network will be used to communicate with other computers and the local area network; for example to generate a response to tweets. The VGA Monitor will display a visual output, while the speaker will output audio waveforms. The PS/2 keyboard will be able to take user input. The CPU will control all the above processes. Each of the peripheral devices will be connected to their own controller device, which is connected by a 4-bit data bus to the CPU. The controllers will govern the interaction process between the peripheral devices and the CPU, for instance, data transfer, error detection, communication and timing control. Both RAM and a boot ROM will to store data for execution. The figure below illustrates the overall design.

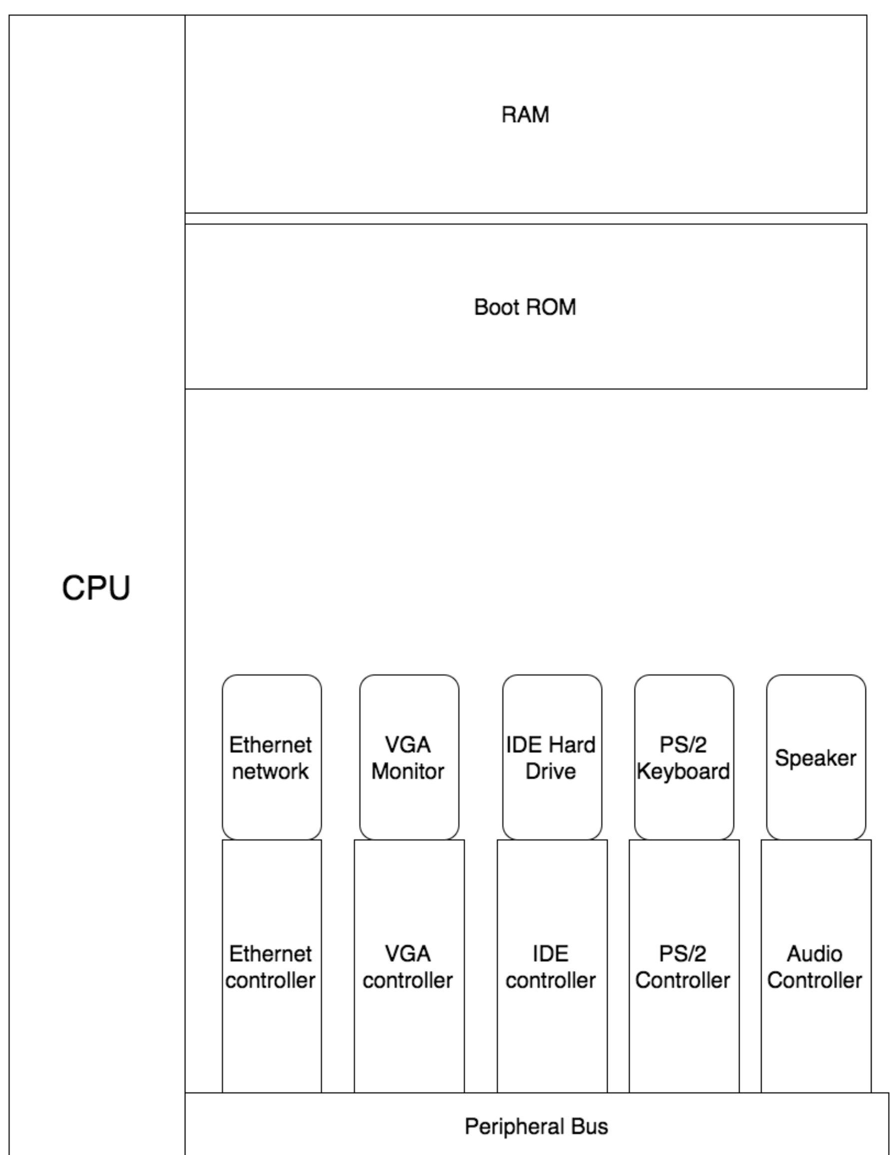


Figure 6.1: A diagram of CPU and connected devices.

Connection between CPU and an External Device

The CPU interfaces with the peripheral device controllers using a technique known as "memory mapped I/O", where the peripheral bus is simply treated as a readable and writable location in memory. There are three 4-bit outputs from the CPU that are used to communicate with the peripheral device controllers. Address 0, the chip select, is a 4-bit input to a de-multiplexer that sends an enable signal to the selected device, allowing it to interact with the CPU and utilize the data and status buses. Address 1, the status bus, is a 4-bit output that contains the information for

device synchronization and tells the peripheral controller whether the CPU is attempting to send or receive data, and allows the controller to tell the CPU if it is ready for such action. Address 2 is the data bus, where data is exchanged between the controller and the CPU, 4-bits at a time. Figure 6.2 visually illustrates the connection between CPU and an external device.

Figure 6.2: A diagram illustrating the connection between CPU and an external device.

Address Spaces in Memory

The total address space required for a 16-bit wide address would be 2^16, which is from 0 to 65535 bits. The following illustrates the general layout of memory, and numbers are used to comparable relative sizes. In the final design, the exact sizes will likely change. The first 3 address spaces (0-2) will be allocated for peripheral input and output for chip select, status and data respectively. Boot ROM is allocated to address spaces from 3 to 1024 bits. Address spaces 3 to 18 will be allocated for the numbers 0 to 15 (as this aids in certain elementary operations in the CPU), address space 19 to 512 will be for the IDE Driver, as the system must boot from information stored on the hard drive; and address space 513 to 1024 are for initialization code which is used to boot the computer up. The remaining address spaces will be allocated to the RAM, from 1025 to 65355 bits. Zero Page Operating System functionality, which reloads the command prompt after completion of a user space process, will be from address space 1025 to 1537 bits. Within RAM, the user space process will be allocated in memory from address space 1037 to 65355 bits.

Operating system libraries, which include drivers and anything necessary to talk to the hardware, will be allocated memory from 1538 to 11788 bits and standard library for user space programming, which includes things such as the functions to print strings to the screen and certain advanced math functions, will be allocated memory from 11779 to 22019 bits. The remaining memory will be assigned for the executable code of the user space program, from address space 22020 to 65535 bits. Figure 6.3 below, describes the 16-bit address space with a user space program in memory.

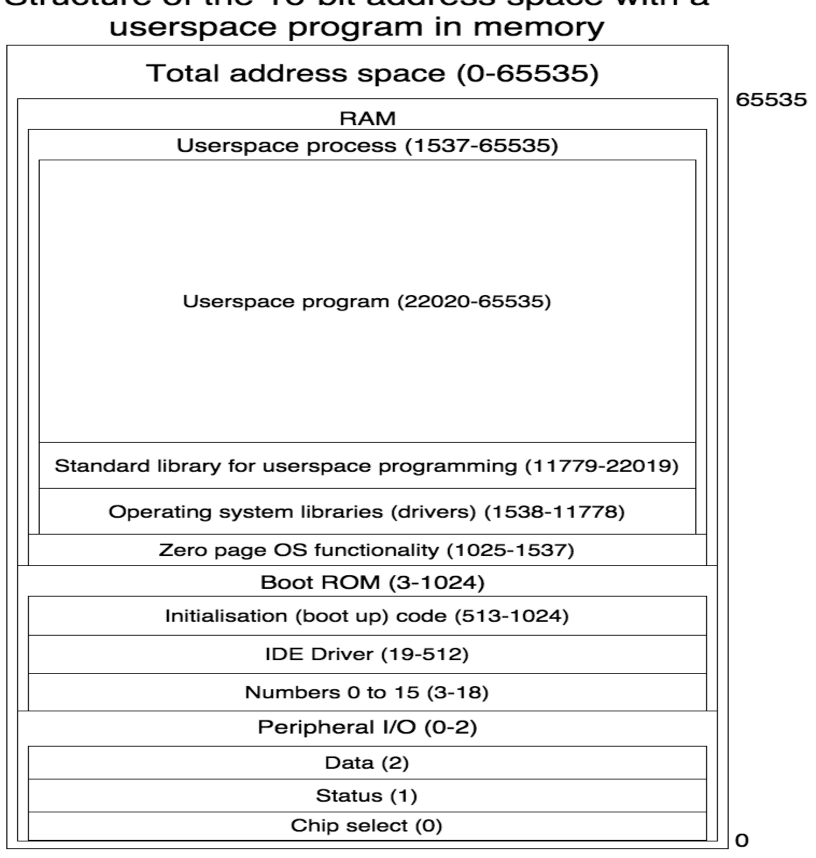


Figure 6.3: Diagram illustrating the structure of 16-bit address space with a user space program in memory

Code Compilation process for Cute BASIC

The programming language is a simple BASIC-like statement based procedural language. Cute BASIC would be translated to machine code through the following process. The Cute BASIC compiler will translate Cute BASIC to macro assembly language. Thereafter, a macro assembler will convert the macro assembly language to a low level assembly language. A macro assembly

language is needed as the actual assembly language of the CPU, containing so few instructions that even the simplest operations require substantial amounts of instructions and as such, the complexity of the compiler itself can be reduced by offloading some of the compilation to a program explicitly designed to detail with the peculiarities of the architecture. The assembly language will consist of only 8 instructions to ensure electrical simplicity. The low level assembly language will be converted to machine code by the low level assembler. This will result into executable machine code, which will then be loaded into RAM for the CPU to execute. Figure 6.4 illustrates the process visually.

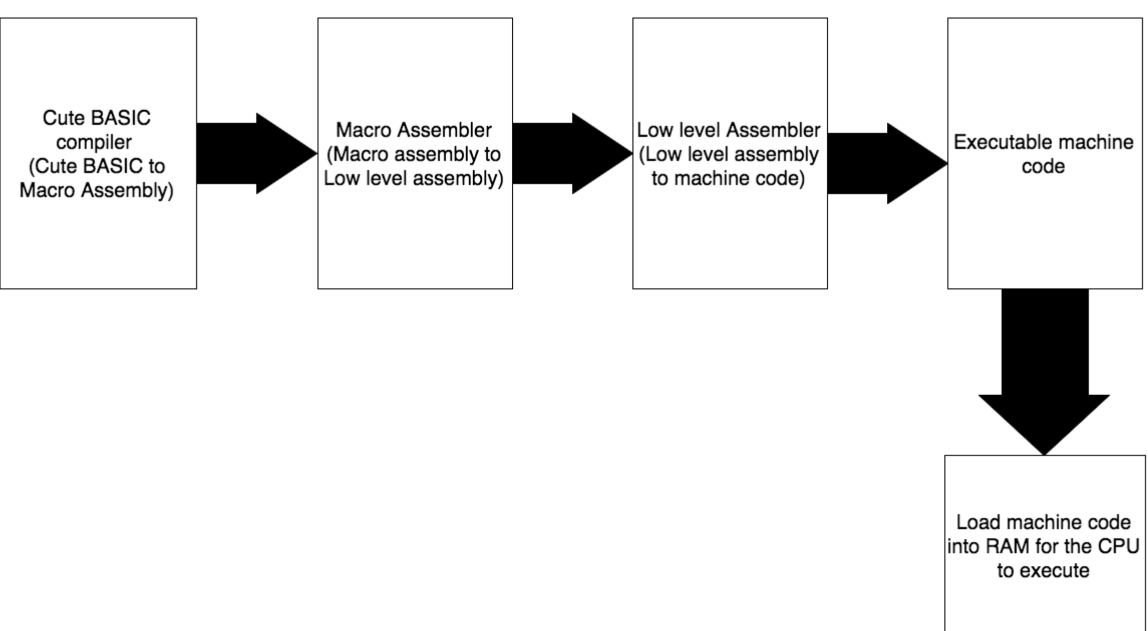


Figure 6.4: A diagram illustrating the compilation process for Cute BASIC

7. CPU

7.1 Introduction

The Central Processing Unit or CPU of a computer is considered the heart and brains of a computer and it is tasked with carrying out the instructions laid out by computer programs. The first CPUs were constructed out of vacuum tubes and relays, which had to be manually programmed by hand; a far cry from today's high powered microprocessors capable of high clock speeds and parallel processing.

A computer processor executes computer programs by performing basic arithmetic, logic, and input/output operations specified by the programs. In order to execute this, the code must be broken down into language and instructions that the CPU can understand, this is called "machine code". Every processor has a different set of instructions, which it is capable of carrying out and the machine code must reflect those instructions or the processor will not work correctly.

Today's modern CPUs are located on single chips with billions of transistors packed inside them, doing thousands of instructions in the blink of an eye. The Nibble Knowledge CPU is a very simplified version of these chips, but the main components and methodologies remain the same as with any other computer processor. It opens the black box and provides a detailed look at each individual component of what makes up a computer processor and the purpose and reasoning behind each decision.

7.2 Instruction Set and Architecture

Every computer processor has its own architecture and instruction set. The reason being that this is how the hardware itself is structured and cannot be changed after it is implemented. This means that even the Nibble Knowledge CPU has its own architecture as described in this section below.

Memory

The CPU has an on board memory in the form of a 64 kB RAM chip with a 4 bit memory cell size. For the initial boot up of the CPU, there is an 8 K boot ROM located in the reset circuit that loads the initial program into the RAM.

Registers

Registers hold values for the CPU to use for various purposes. Modern day processors have many of these registers for convenience in computation and programing. The Nibble Knowledge computer however, only has a few essential registers. The table below lists the Registers located in the CPU and their functions.

Register	Width (bits)	Comments
PC	16	Program counter; holds current address of next instruction to be loaded
A	4	Accumulator register used for arithmetic and other operations
MEM	16	Holds the address being used by an instruction at any given time.
STAT	4	layout: (signed overflow bit) XOR with MSB of the A register, EMPTY, HLT, Carry bit

Table 7.1: Registers Located in the CPU and Their Functions

Instruction Set

The table below lists the eight instructions supported by the architecture of the CPU and their machine code format and description. This instruction set was developed and testing for the completeness of the instruction set was done by using a program called CPUsim found at: http://www.cs.colby.edu/djskrien/CPUSim/

Instruction	Format	Description
HLT	0000-0000-0000-0000-0000	Halts the CPU
LOD	0001-16-bit address to load from	Loads a value from location specified into the A register
STR	0010-16-bit address to store to	Stores the value in the A register into memory location specified
ADD	0011-16-bit address to add from	Adds the value at the location specified, plus the existing carry flag, into the value in A. Sets carry and overflow flags as required
NOP	0100-0000-0000-0000-0000	No operation
NND	0101-16-bit address to NAND	Bitwise NANDS the value at the location

	from	specified into the value in A
JMP	0110-16-bit address to jump to	Jumps to location specified by setting the program counter to that address
CXA	0111-0000-0000-0000-0000	Copies the STAT register into the A

Table 7.2: Machine Code Format and Description of Instruction Set

7.3 The Virtual Machine

The Virtual Machine is a digital simulation of the CPU and its architecture is set to clock accurate speeds. This was done in order to test the completed peripherals while the discrete portion of the CPU was being completed, as well as to test the code compiled through the software toolchain to see how it operates before using it on the discrete CPU. This enabled troubleshooting as it made it possible to find errors in the macro assembly, logic errors in the peripheral drivers and physical controllers.

There are two versions of the virtual machine: first is an implementation on a Raspberry PI running Arch Linux as the main operating system. Written in low level C, the Raspberry PI virtual machine is a command line program used to gain access to the Raspberry PI board’s GPIO pins to interface with discrete components and run machine instructions loaded from binary files created by the assembler. The second version is a windows program with a full user interface written in C# language, where users can edit and compile macro assembly and test the resulting machine code. This version is mostly used for software testing and behavioral analysis of the CPU.

7.3.1 The Raspberry PI Virtual Machine

In order to get the virtual machine set up and running on the Raspberry PI, Arch Linux must be installed first.

Setting up Arch Linux

Note: Arch Linux could also be set up through Noobs!

The implemented set-up process requires more work, and a lot more setup time. However, it does provide more barebones experience, which should help with accuracy of timings, and accuracy GPIO lines switching. Also, these instructions are assuming that there is another Linux computer available for the setup. It is possible to use windows, following this link:

1. With your SD card plugged in, open a terminal on your linux machine and type lsblk, this shows you the name of the SD card which you will need.
2. Follow the instructions here: http://archlinuxarm.org/platforms/armv7/broadcom/raspberry-pi-2
3. Once the Pi is on and in the terminal here are the suggested items to install:
    a. Install Sudo: pacman -S sudo
    b. Create a regular user account: useradd -m -G wheel -s /bin/bash newuser
    c. Add user password: passwd newuser
    d. You can now logout and login as that user
    e. Install most developer tools:sudo pacman -Sy base-devel
        i. Verify version of gcc with: gcc -v
        ii. Verify make version with: make -v
    f. Install Git: sudo pacman -Sy git
    g. For nano syntax highlighting (Raspbian way may work, but if not try this)
        i. Type: sudo nano /etc/nanorc
        ii. Scroll down to near the bottom and find: “include /path/to/syntax_file.nan0rc”
        iii. Type: include /usr/share/nano/c.nanorc
include /usr/share/nano/asm.nanorc
4. Required repositories for running the virtual machine
    a. sudo pacman -Sy doxygen
    b. sudo pacman -Sy graphviz
    c. sudo pacman -Sy links
    d. sudo pacman -Sy hexedit
    e. sudo pacman -Sy openssl

To Change Keyboard Default:

If you are using the Pi with a screen and keyboard and the keyboard is incorrect:

sudo nano /etc/default/keyboard

*go to the thing and make it -> XKBLAYOUT=”us”

Setting up SSH Server for Remote Access (Optional)

1. Download PuTTY from http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html.
2. On the Raspberry Pi, start the SSH Server by typing "sudo systemctl start ssh" into a terminal window. (If that doesn't work type in raspi-config and navigate to the SSH settings in the menu and enable)
3. On the raspberry pi, get the IP address by typing "ip addr" in the terminal.
4. Open PuTTY and in the "Host Name (or IP address)" put the address you just found.
5. Save the session then "Open." Some error might come up, ignore it.
6. Login with the user you created with Arch Linux

Once Arch Linux is set up the VM can be installed by pulling from the Nibble Knowledge virtual machine repository on GitHub found at https://github.com/Nibble-Knowledge/cpu-vm or using a program such as WinSCP to manually place the files onto the Raspberry PI.

To run the virtual machine on the PI, compile it by typing "make" while in the directory, then run it using "sudo ./vm4". Then follow the onscreen prompts from the program to operate.

7.3.2 The Windows Virtual Machine

The windows virtual machine is an exact copy of the Raspberry Pi virtual machine, except, it includes a graphical user interface (GUI) to make it easier to manage. However, it does not allow for pin outputs. It is more a tool for software to step through the code and observe what happens to the registers, I/O Ports, Bus, Memory, and even Peripherals. As of this report, the only peripheral that was virtually added was the hard drive. Hard drive commands could be sent in the same manner as commands sent to a real hard drive and observe what outputs it returns.

The GUI is built using Microsoft's Visual Studio in C#. The full project can be found on GitHub. Currently the features included are as follow:

- Write, edit, and assemble NK4 Assembly and Macro Assembly
- Step through the code at your own pace
- See the CPU Registers, and Main Memory update as you step through the code
- Simulate the NK4 CPU running through code with a given period

- Attach a virtual hard drive that can be written to, and read from (Currently only 1000 sectors in size)

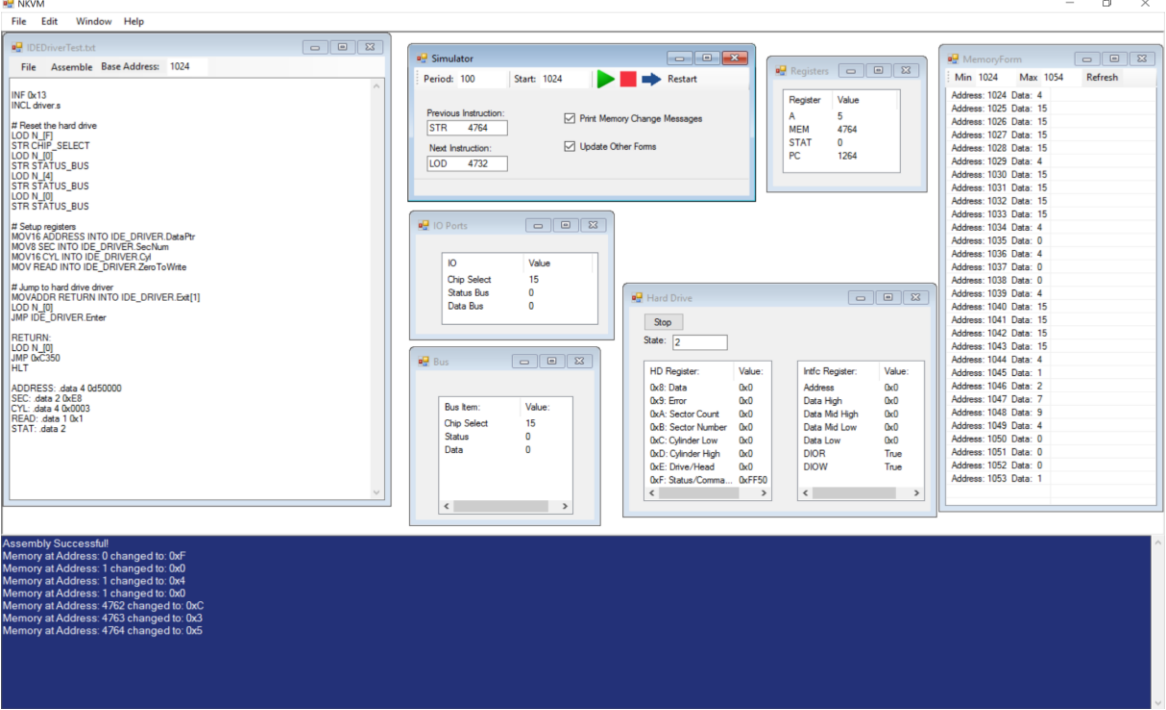


Figure 7.1: The CPU GUI

The GUI was used to test and debug the IDE driver and saved the endless rewriting of ROM chips to test on the real CPU. It was also a great visual tool to show instructions running and what the effect of each instruction was. This was a last minute tool that was created in March 2016 and was not a part of the original scope. It is not complete, merely a work in progress.

7.4 FPGA Implementation

All parts of the CPU were first designed and simulated using FPGA's before moving into discrete components. This allowed to confirm logic and architecture for each distinct block of the CPU and fix bugs before it was made into a discrete circuit. The main methodologies were consistent between the two phases and the circuits generated in VHDL were used to create the discrete versions.

Example 1: A four bit counter for the control unit

```
-- Counter
process(clk, reset)
```

```
begin

	if rising_edge(clk) then
		--if reset = '1' then
			--cycle_counter <= "000";
		if cycle_counter = "101" then
			cycle_counter <= "000";
		else
			cycle_counter <= cycle_counter + '1';
		end if;
	end if;

	if reset = '1' then
		cycle_counter <= "000";
	end if;
end process;
```

This process counts up by one every clock cycle effectively making 4 bit counter. However since there are only 6 cycles in the instruction cycle and a 4 bit counter will go up to 15 it needs to be reset to 0 on the next clock cycle after reaching 5 thus restarting the counter for a new instruction. This is accomplished by the if else statement inside the rising edge detector to either add ‘0001’ to the current count or reset to ‘0000’ if the count is at 5.

Example 2: ALU Decode

```
process(clk)
begin
	if clk' event and clk = '1' then
		--Write op_code into temp storage
		if OP_EN = '1' then
			stored_OP_Code <= OP_Code;
```

```
			end if;
		end if;
end process;
-- Decoder Combinational Logic --
-- Active Low WE
WE <= '0' when ( stored_OP_code = "0010" and exe = '1' and clk = '1') else '1';
STR <= '1' when (stored_OP_code = "0010" and exe = '1') else '0';

-- HLT
HLT <= '1' when( stored_OP_code = "0000" and exe = '1') else '0';

-- A_EN
A_EN <= '1' when( (stored_OP_code = "0001" or stored_OP_code = "0011" or stored_OP_code
= "0101" or stored_OP_code = "0111") and exe = '1' ) else '0';

-- STAT_EN
STAT_EN <= '1' when( stored_OP_code = "0011" and exe = '1') else '0';

-- JMP
JMP <= '1' when( stored_OP_code = "0110" and exe = '1') else '0';

-- Arith_S
Arith_S <= '1' when( stored_OP_code = "0101" and exe = '1') else '0';

-- Stat_S
Stat_S <= '1' when( stored_OP_code = "0111" and exe = '1') else '0';

-- LOD_S
LOD_S <= '1' when( stored_OP_code = "0001" and exe = '1') else '0';
```

This is the entirety of the ALU decode block in the VHDL Implementation. The process at the beginning of the code acts as the OP code DFF and will store the incoming OP code when the OP_en signal is high. Whatever is stored in the stored_OP_code variable is then used to calculate the WE, HLT, A_EN, STAT_EN, JMP, ARITH_S, STAT_S, and LOD_S signals in the combinational logic block without the need for a clock. This simulates the signal running through logic gates in the discrete circuit following the truth table found in Table ## in the above section.

High Level of the CPU

On a high level, the CPU can be divided into six distinct blocks. These blocks are the Boot Circuit, the Control Unit, the MEM Register, the ALU, the Program Counter and the I/O circuit. Each one of these serves an essential purpose in the CPU and each of the six blocks will be explained in detail further in this section. The high level view of the CPU architecture is shown below.

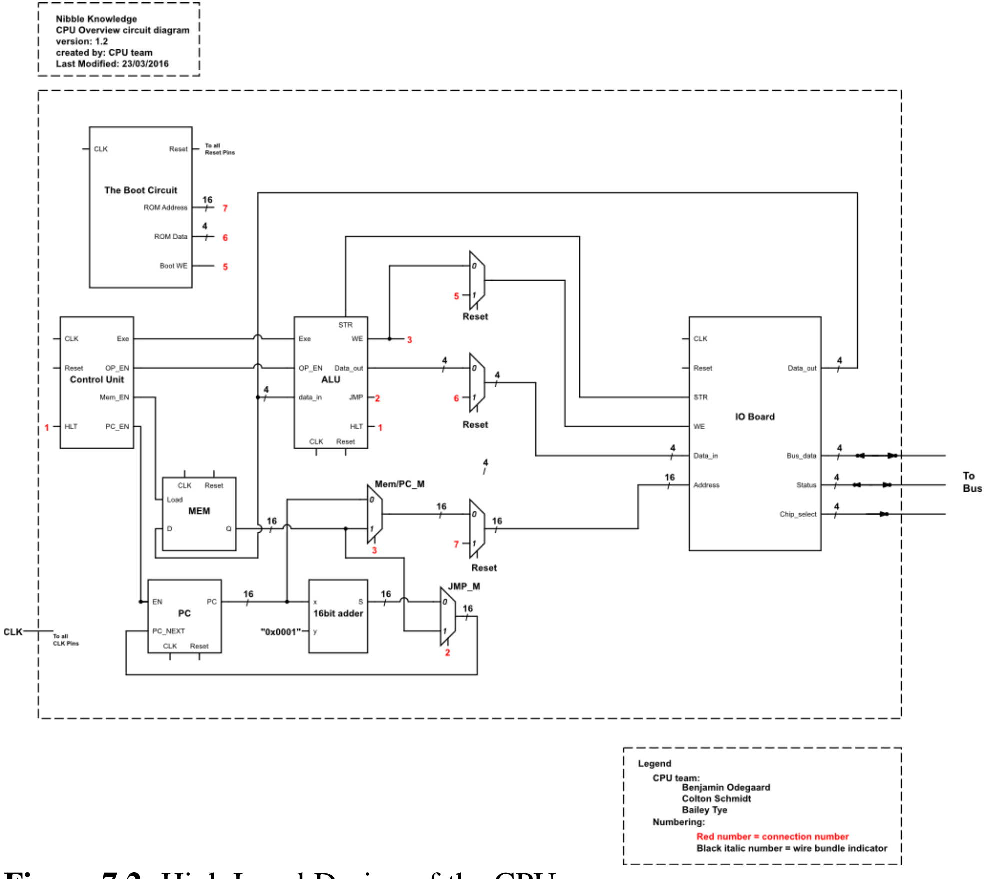


Figure 7.2: High Level Design of the CPU

The Cycle Control Unit

The CPU completes an instruction every six clock cycles and in order for the CPU to keep track of where it is in the instruction, it needs a way to count the steps while operating. This is the job for the Cycle control unit. These 6 cycles are separated into three distinct phases of operation, the “OP Code” phase, the “Address” phase and the “Execute” phase.

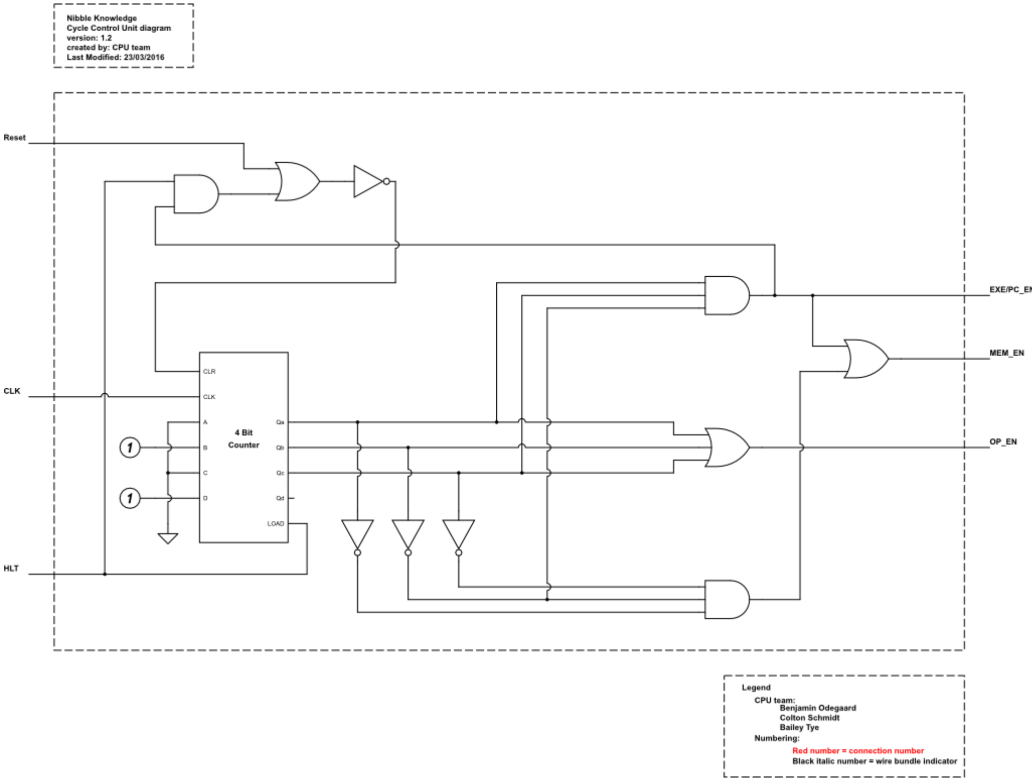


Figure 7.3: Cycle Control Unit of the CPU

<u>OP Code Phase</u>

The OP code phase lasts the first clock cycle of the six cycle sequence and it’s function is to load the first nibble of the instruction (the OP code) into the register located in the ALU decode.

<u>Address Phase</u>

The next four clock cycles are the Address phase, which loads the next four nibbles of the instruction (the memory address) into the MEM register.

<u>Execute Phase</u>

The final clock cycle executes the loaded instruction by doing a couple of important operations in the CPU. First, it disables the program counter so that it does not increment on the next rising edge

of the clock. Next, it switches the address accessing RAM from the PC to the address loaded into the MEM register. Finally, it sends the EXE signal to the ALU to perform the required instruction operation. The table below is an overview of the signals. AL denotes an Active Low signal and AH denotes an Active High signal

Cycle	OP_en (AL)	MEM_en (AL)	PC_EN (AL)	EXE (AH)
1	0	1	0	0
2	1	0	0	0
3	1	0	0	0
4	1	0	0	0
5	1	0	0	0
6	1	1	1	1

Table 7.3: Overview of Cycle Control Unit Signals

Using the table above, logic equations were developed for the proper output of each signal; these equations are:

$OPen = A + B + C$

$EXE = PCen = AB'C$

$MEMen = A'B'C' + AB'C = B'(A'C'+AC)$

The Program Counter

The Program Counter or PC is a register that holds the address of the current instruction in memory. The PC register in itself is just a normal 16-bit flip-flop. The output of which splits and is outputted to the memory select mux and also to a 16-bit full adder that increments the PC by one.

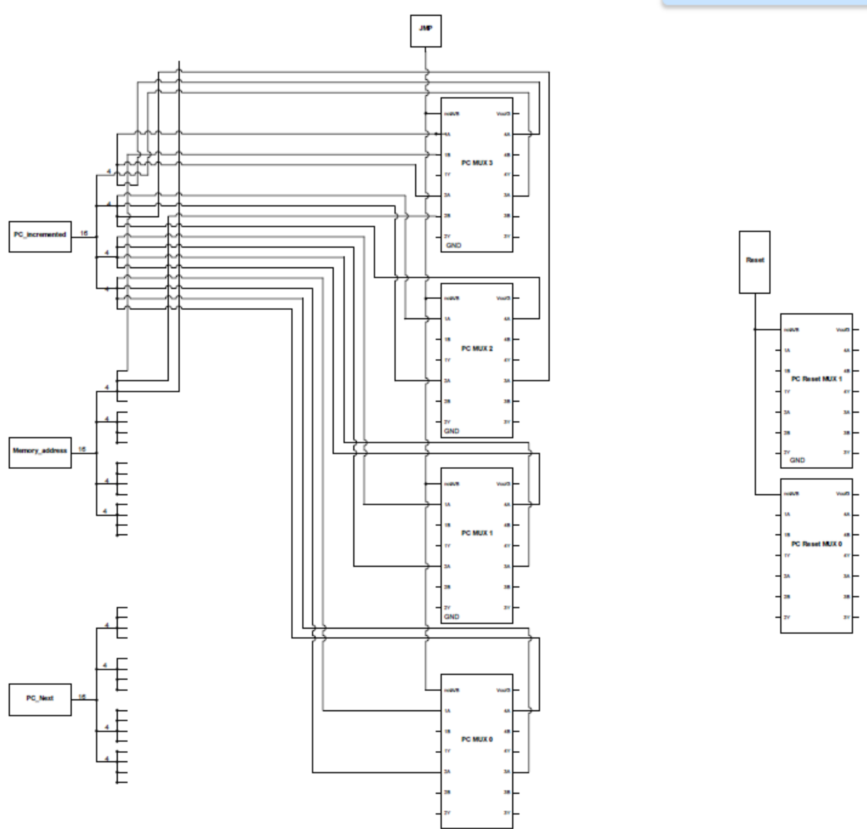


Figure 7.4: The Program Counter Circuit

The Memory Register

Since each instruction is based on an OP code and a 4 nibble memory address, the address of the instruction needs to be stored in order for the CPU to execute instructions. This is accomplished by the MEM Register. The MEM register block is a modified version of a regular flip-flop register. Due to the limitations of the memory bus from RAM, this special register had to be designed in order to load a 16-bit address one nibble at time. When the load input of the register transitions to logic high, it loads a 4-bit nibble onto the 16-bit Q line starting at the most significant nibble and moving down a nibble every clock cycle. This process is illustrated in the figure below.

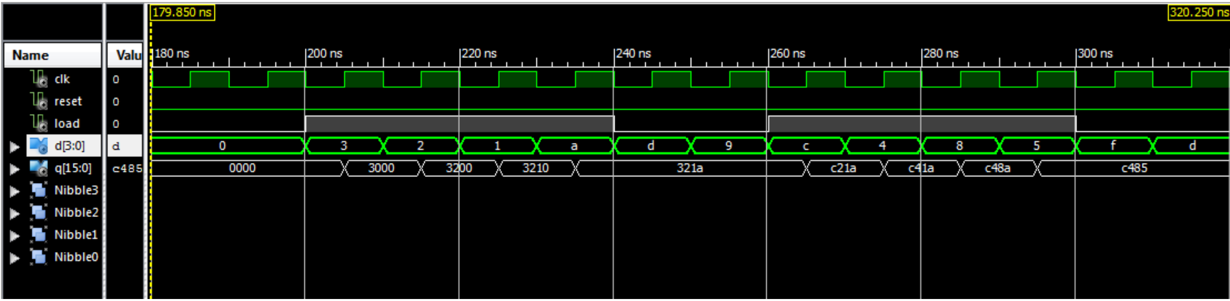


Figure 7.5: Testbench of Two Instruction Cycles of the CPU

As shown by the white waveforms, when load goes to logic high, the MEM register starts to load nibbles into the register from top nibble to bottom nibble. Load then goes to logic low for two cycles to allow for the Execute and OP code phases to pass before becoming high again for the next address.

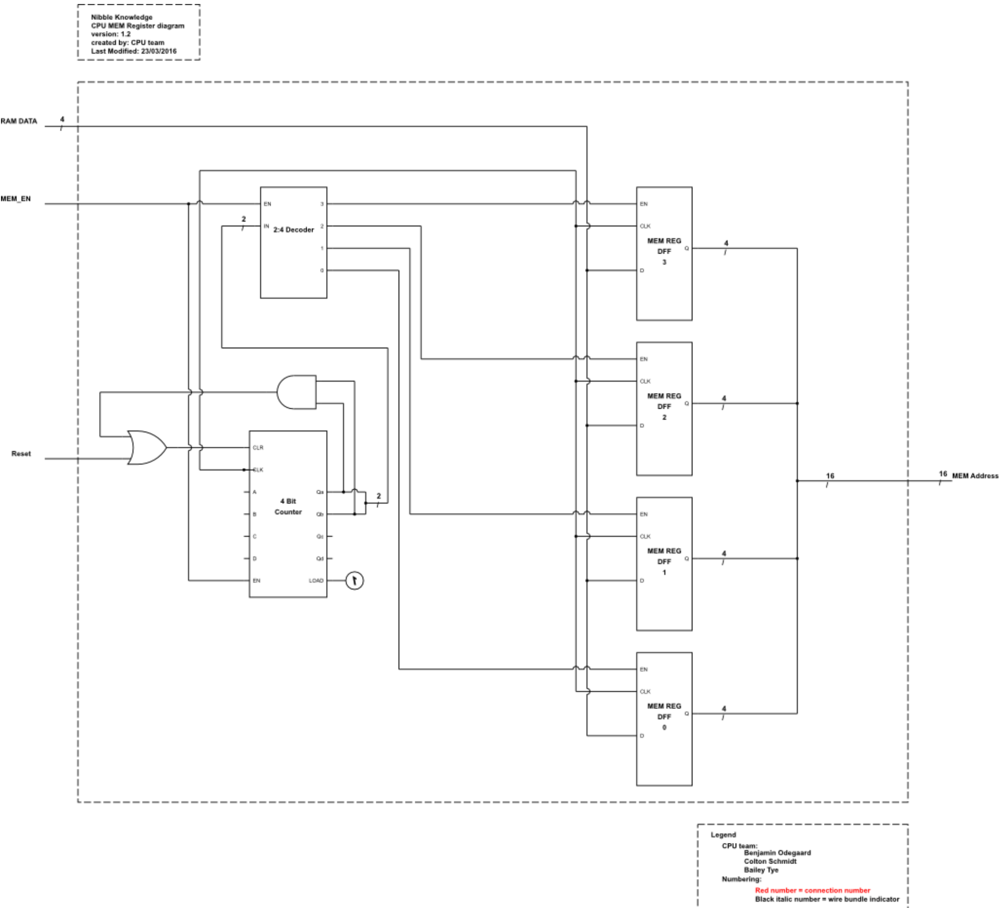


Figure 7.6: Circuit Schematic of the Memory Register

The ALU

The ALU or Arithmetic Logic Unit is where instructions are carried out in the CPU. It firsts determines what to do with the data as specified by the OP code and performs the required arithmetic and sends out the proper control signals to various parts of the CPU.

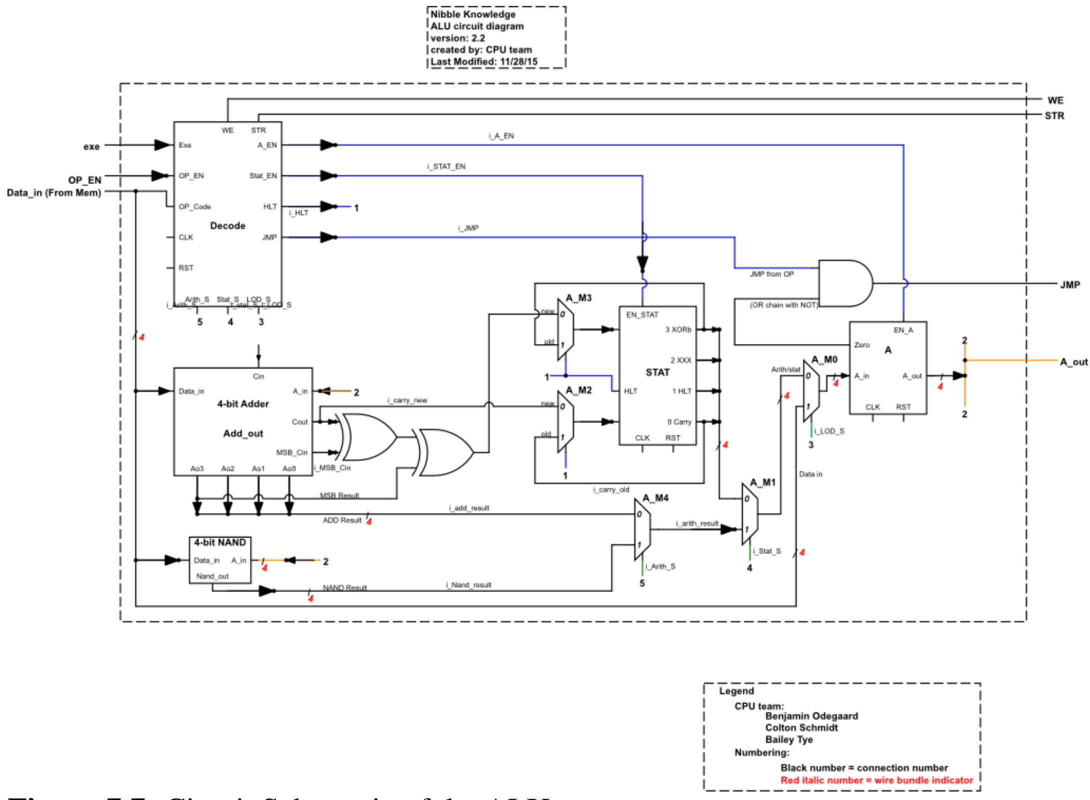


Figure 7.7: Circuit Schematic of the ALU

ALU Decode

The ALU decode is what tells the rest of the ALU components what to do. On the first cycles of the 6 cycles it takes to run an instruction, the opcode is saved inside the decode unit (this will be on the data in line from main memory). The chart below shows which signals each instruction requires. The only clock driven components in the ALU are the A, opcode (inside the ALU decode), and STAT registers. Everything is combinational and will basically be running even when the decoder signals are not specifically assigned. During cycle 6, the execute phase enables and select signals are turned on based on the opcode, and the instruction is complete.

Instruction	WE	HLT	A_EN	STAT_EN	JMP	Arith_S	Stat_S	LOD_S
HLT	0	1	0	0	0	X	X	X
LOD	0	0	1	0	0	X	X	1
STR	1	0	0	0	0	X	X	X
ADD	0	0	1	1	0	0	0	0

NOP	0	0	0	0	0	X	X	X
NND	0	0	1	0	0	1	0	0
CXA	0	0	1	0	0	X	1	0
JMP	0	0	0	0	1	X	X	X

Table 7.4: Low Level Design of Top CPU Module

Using the logic developed in above table, equations were developed to create the right signals out of logic for implementation. These equations are:

$Aen = C'$ $\quad$ $JMP = ABC'$

$Sen = A + B' + C'$ $\quad$ $ArithS = A$

$HLT = A + B + C$ $\quad$ $StatS = AB$

$WE = A + B' + C$ $\quad$ $LodS = A'B'$

These are the equations used in the implementation of the ALU

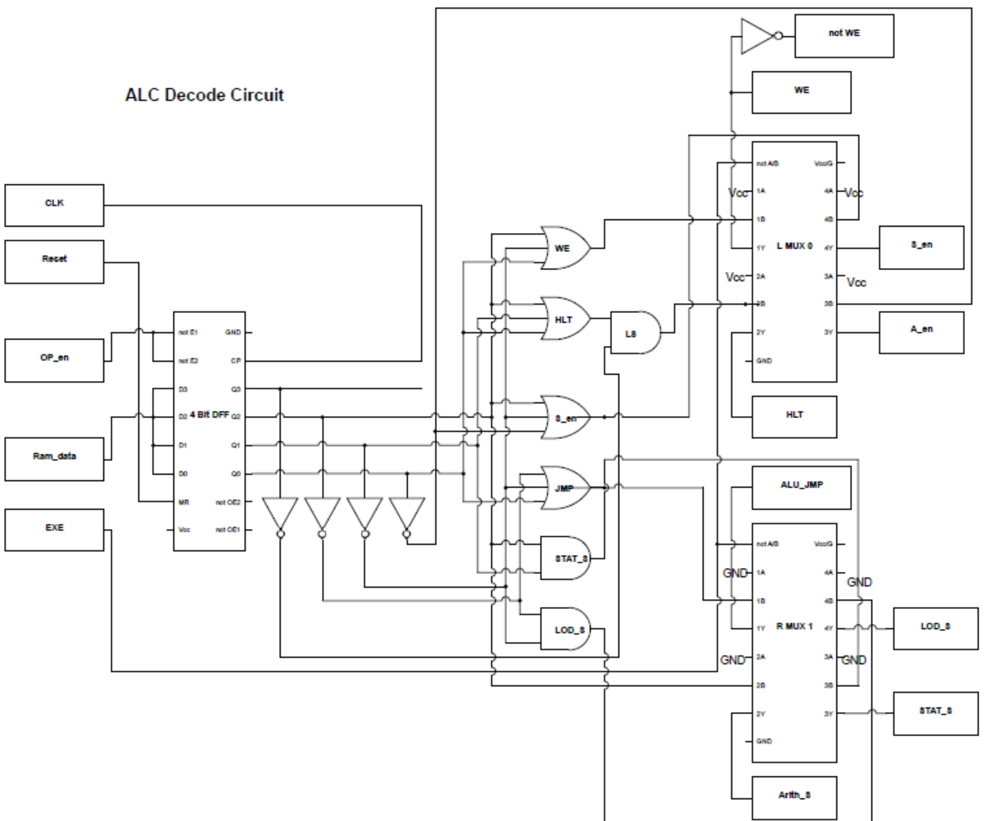


Figure 7.8: ALU Decode Circuit Schematic

ALU Logical/Arithmetic Operations

The ALU is capable of two operations: a bitwise NAND of the A register with a memory location, and a 4-bit add. These are combinational circuits as stated above, so they will be running even if the instruction does not require them.

MUXs/XORs/ANDs

The two XOR gates in the middle of the ALU are used for overflow detection. Even though there is only an ADD, and most operations will be unsigned, the overflow detection for comparisons was implemented. After which, XOR the overflow bit with the MSB of the sum and this will determine whether a number is greater than the other, or less than.

The Bus Interface and Memory

The IO Board holds the CPU's main memory and the registers for the memory mapped I/O addresses. This means that this section manages the RAM and the input and output to the bus. This is accomplished by first taking the 16 bit address line coming into the block and determining if it was in the lowest three addresses by a chain of NAND gates the output of this logic block is then used as the enable for a decoder of which the bottom two bits are sent as inputs. This causes the right DFF corresponding to the bottom two bits to be enabled if the top 14 bits of the address line are '0' else they are disabled from propagating. Whether the address is in a memory IO spot the data is always sent to the RAM chip no matter what. To get data from the I/O board the same bottom two address bits are sent to a 4 to 1 multiplexer that selects either the Data register, Stat register, Chip Select, or the RAM chip to output back to the CPU.

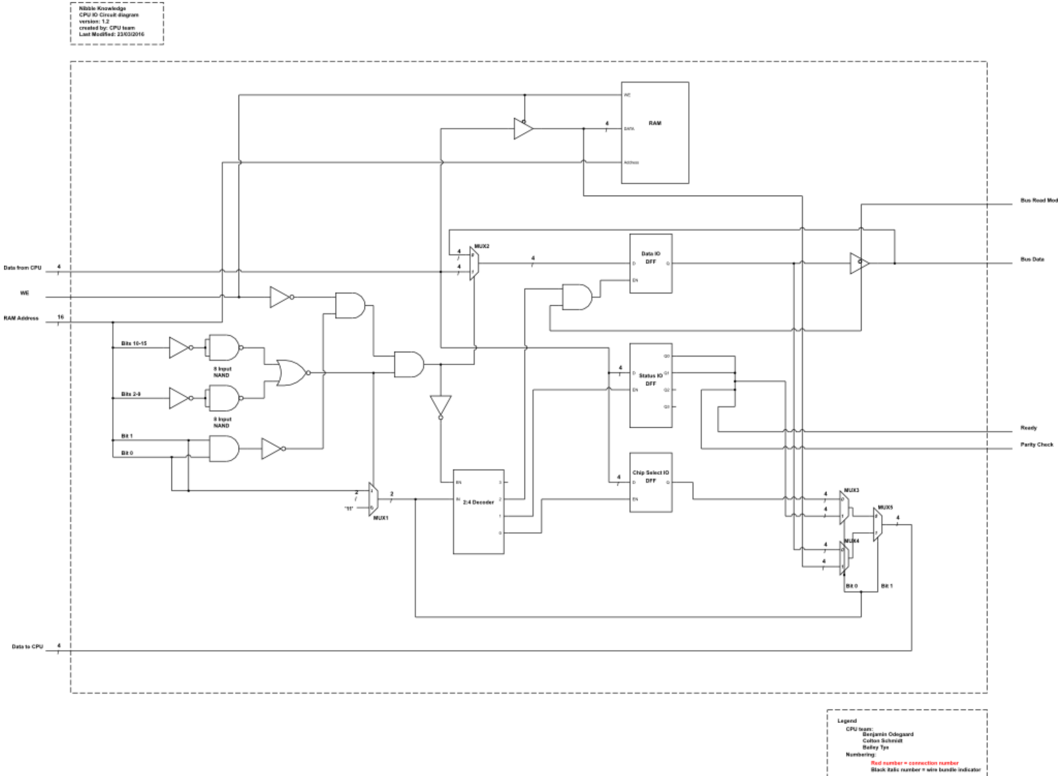


Figure 7.9: Circuit Schematic of the CPU I/O

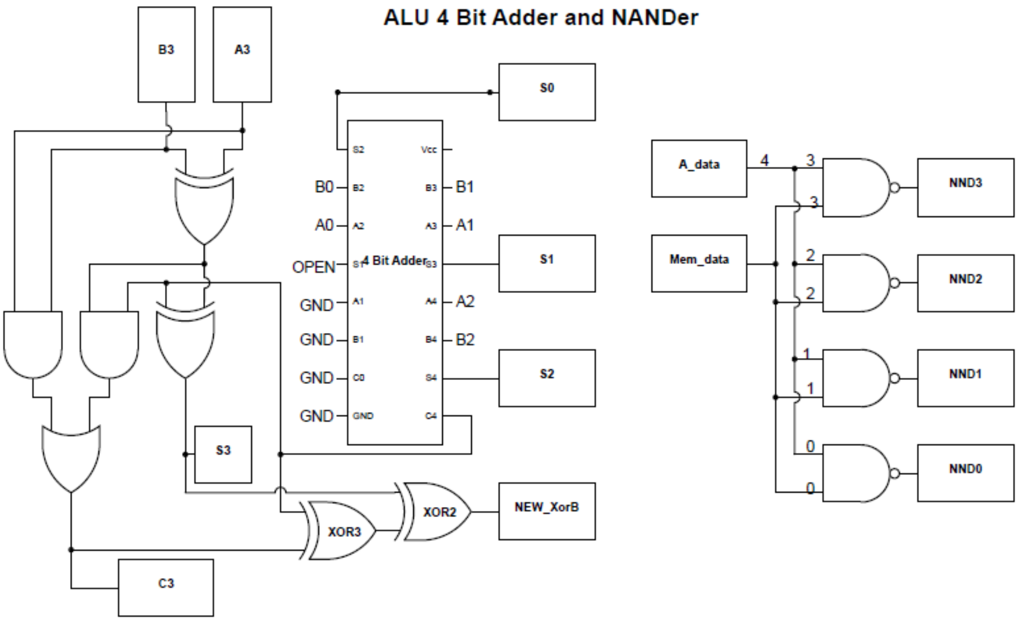


Figure 7.10: ALU 4-Bit Adder and NANDer Circuit Schematic

The Reboot Circuit

The reboot circuit works as a way to set the CPU in a known state by sending out a reset signal to the rest of the CPU while bypassing the hardware to load the initial boot program from the ROM

into the RAM on the IO board. This is accomplished by using a counter that is one bit bigger than the address lines on the ROM chip. The output of the counter is fed into the ROM and also sent out to RAM in order to transfer the data at the same address. This continues with the DFF inverting every clock cycle to simulate the pulsing of WE signal and advancing the counter every two clock cycles. When the counter reaches the top bit that is not used by the ROM the boot circuit effectively turns off by isolating itself and the CPU begins to operate normally beginning at a PC of 19.

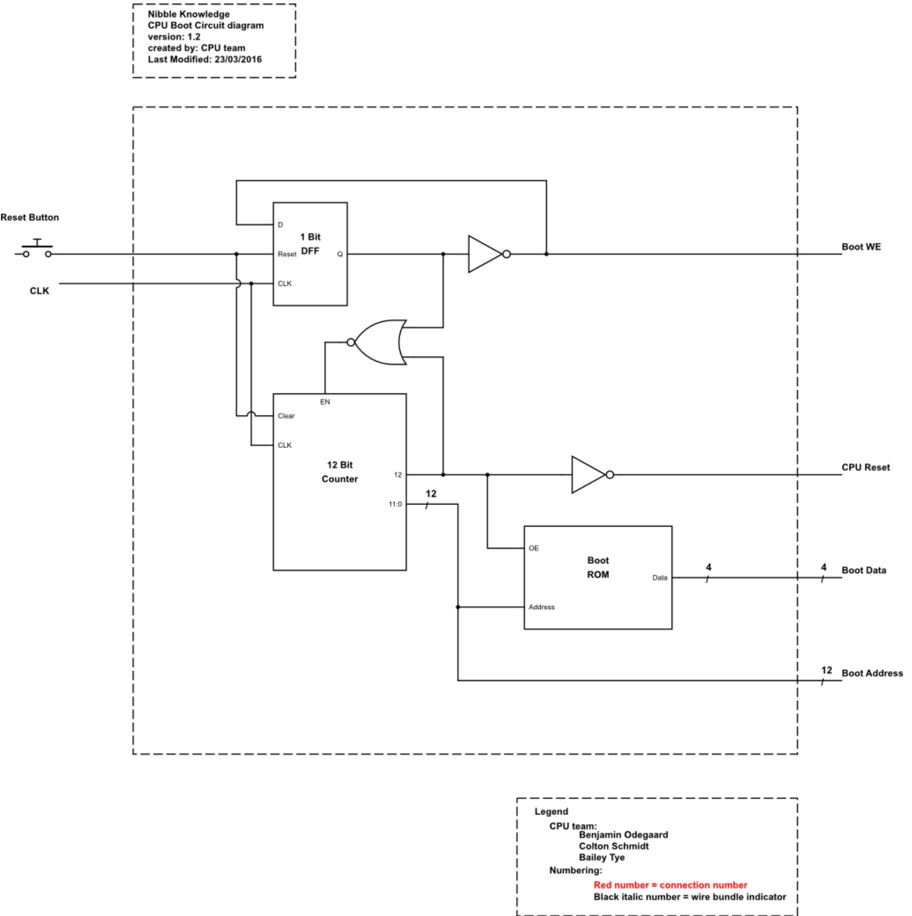


Figure 7.11: CPU Boot Circuit Schematic

7.5 Discrete Implementation of the CPU

Discrete Implementation used the circuit diagrams developed in the FPGA stage as discussed above to create the discrete circuits. The circuits were wired using 74LS series IC chips and minor adjustments had to be made to accommodate the chips available. An example of this would be the cascading of 4 bit counters in order to make the 12 bit counter necessary for the boot circuit to operate properly.

Consider the above example more in depth and take a look at how to wire it:

1. First take a look at the data sheet for a 4 bit counter (SN74LS163N) and look at the pin out diagram as shown below. Taken from http://www.ti.com/lit/ds/symlink/sn74ls163a.pdf
2. Now since it is a 12 bit counter we need 4 of these chips so put 4 of them down one row of the breadboard.
3. Looking at the pin out diagram we can that the pins for Vcc and GND are pins 16 and 8 respectively. Attach the Vcc pins on chip to the red rail along the side of the breadboard lanes and attach the GND pins to the blue rail. Do this for each of the 4 chips. These wires will power the chips.
4. Since we are not using the LOAD functionality of the chip we must wire that pin to the red power rail on the breadboard so it is a constant high value
5. Next wire the CLR pins together between each chip this will serve as the common reset
6. Since there is a common clock signal to all the counter chips wire the CLK pins together also
7. Now since the need to cascade the counters so they count properly when moving on to the next one we need to make use of the two enables present on the chip. When a counter is full it makes the ripple carry out or RCO pin go into a high state. This means that this pin should be attached to an enable pin (either ENP or ENT) of the next two chips above it.
8. If all of the previous steps are down correcting the clock and reset signals can now be hooked up and the counter should operate as a full 12 bit counter outputting on the Q pins of each chip.

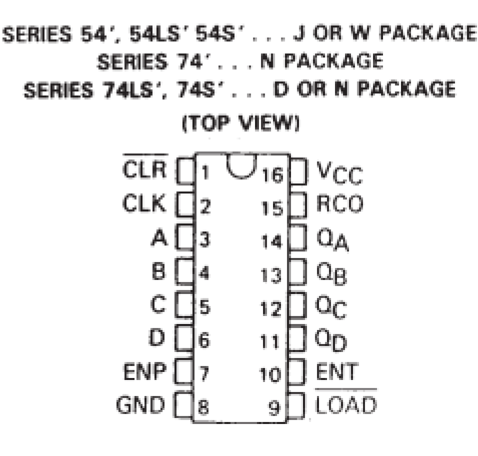


Figure 7.12: Top View of Series 74LS Chip

7.6 CPU Clock Generator

The clock controls the synchronization of the events of all sequential circuits in CPU, providing the CPU with a reference to time needed for transferring data, and command execution. A clock generator outputs a constant frequency which is a usually a square wave that is then distributed to other devices that need it. The clock oscillates between two different states: low and high at the specific frequency making it a periodic signal. From math, we determine that the period (T) of the clock signal by computing $\frac{1}{f}$.

The ideal waveform of the clock signal looks like the figure below

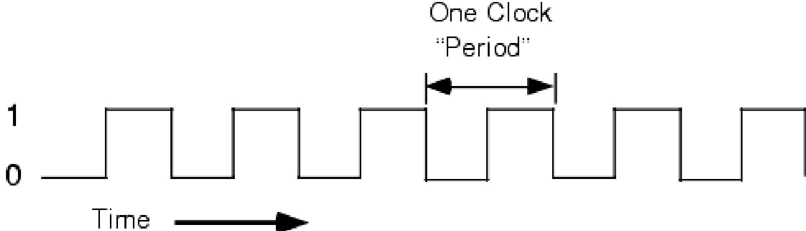


Figure 7.13: Ideal Clock Waveform

The realistic waveform of the clock signal looks like

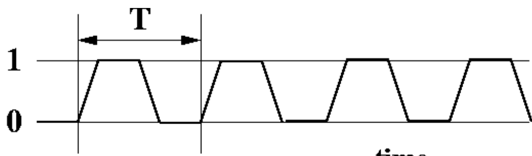


Figure 7.14: Realistic Clock Waveform

The transition of the clock from 0 to 1 is called the rising edge/ positive edge, while the transition from 1 to 0 is called the falling edge/ negative edge. Devices are synchronized to perform tasks at either their falling or rising edges.

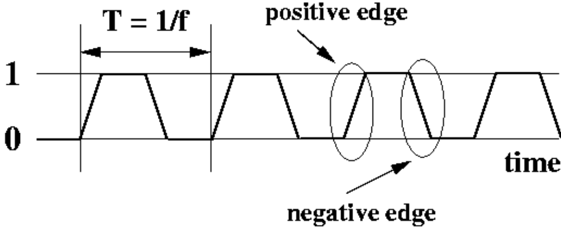


Figure 7.15: Positive and Negative Edge of Clock Waveform

Modern computers run at much higher frequencies than the Nibble Knowledge four bit computer. The Nibble Knowledge computer operates at four different frequencies: 1 HZ, 10 Hz, 100 kHz and

5.00 MHz. The slower frequencies are mainly for visual purposes, so that if a person were to hook up light emitting diodes (LEDs) to the clock output, one would see lights flashing at that frequency. Since there was no way to build an oscillator using just one circuit, there are two separate clock generator circuits controlled by a master switch that alternates between the two.

555 Timer with Select Frequencies

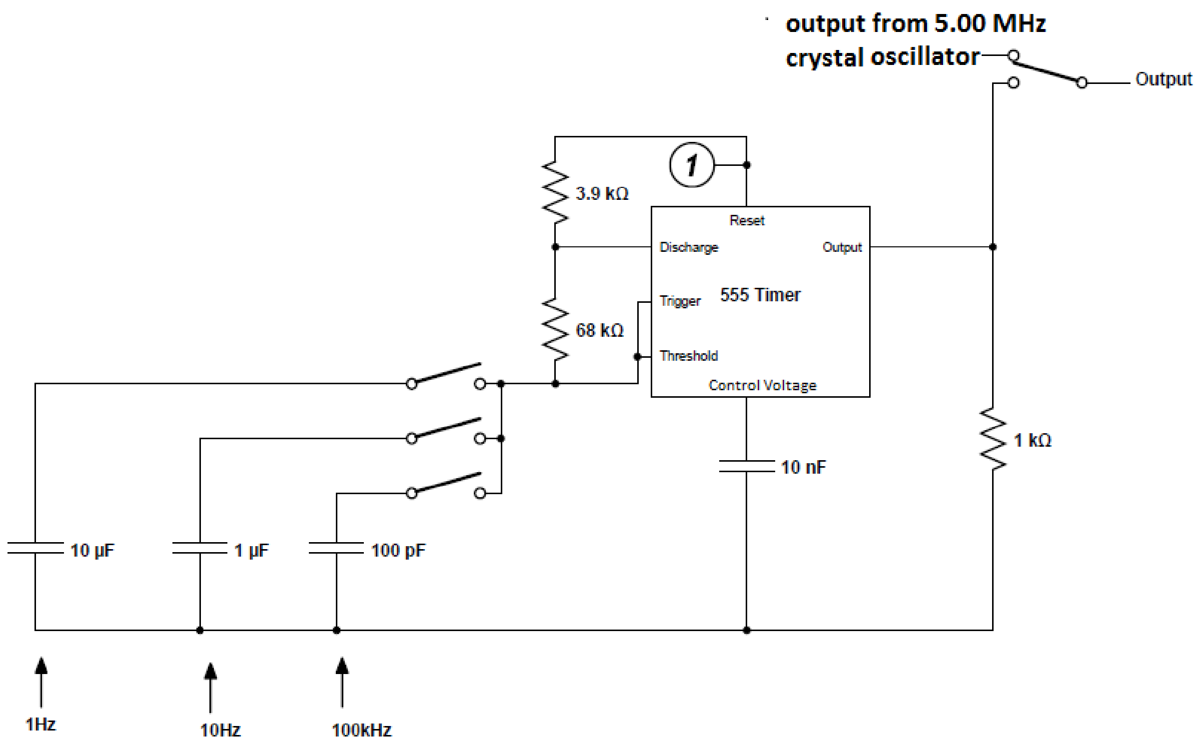


Figure 7.16: Timer Circuit with Select Frequencies

The 555 timer outputs a select frequency based on a different value of capacitor as shown in the above figure (i.e. if a frequency of 1 Hz is desired we would connect the 10.0 µF capacitor). The 555 timer operates in a-stable mode, outputting a square way with that respective frequency. The trigger and reset pins on the chip are active low. Trigger starts the 555 timer, when it is triggered, pin 3 goes high. It is triggered when voltage on this pin is reduced to below one-third of the power supply voltage. The discharge pin discharges a capacitor; it is used to control the timing interval. The threshold monitors the voltage across the capacitor, when the voltage reaches two-thirds the power voltage, the cycle is finished and the output pin goes low. The control voltage is connected to a small capacitor; the purpose of the capacitor is to reduce any distortion that arises from the

power supply. The three switches controlling the capacitors are called dip switches and their diagram is shown to the side.

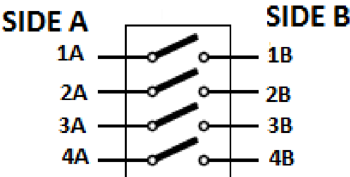


Figure 7.17: The Three Switches Controlling the Capacitor

Vcc is connected to +5.00 V, while GND is connected to ground. Reset pin on the 555 is also connected to +5.00 V.The output (pin 3) is connected to one of the pins of the toggle switch. One end of the 10 nF capacitor is connected to the control voltage while the other end is connected to ground. Tigger and threshold pins are connected together at the same node and a dip switch is connected between those pins and the capacitor.

- 10.0 µF capacitor is connected to 1A; 1B is connected to either trigger or threshold.
- 1.00 µF capacitor is connected to 2A; 2B is connected to either trigger or threshold.
- 100.0 pF capacitor is connected to 3A; 3B is connected to either trigger or threshold.
- 4A, 4B are connected to the crystal oscillator

A 68.0 kΩ resistor is connected between discharge and trigger. The 3.90 kΩ resistor is connected between reset and discharge.

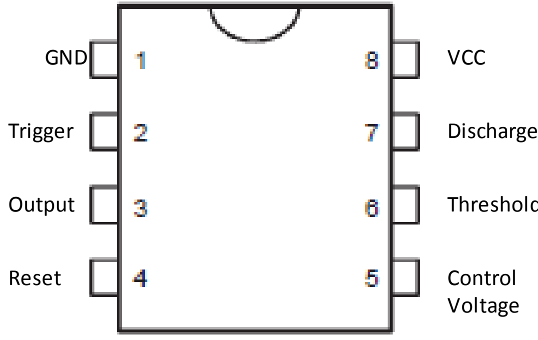


Figure 7.18: Top View of SA555N Timer Chip

Crystal Oscillator

A crystal is a piece of electrical circuitry that has piezoelectric properties. Piezoelectric means that when the shape of the crystal is changed (subject to mechanical stress) an electric potential

(voltage) is formed across the crystal's terminals. Piezoelectric devices can be classified as transducers because they convert energy from one form to another. There are many different types of crystals that can be used as oscillators but one of the most commonly used is the Quartz Crystal because of its great mechanical strength.

The fundamental frequency of the crystal is determined by its physical properties: the size and thickness of the element. This frequency is also called the characteristic frequency. Once the crystal is cut, it can only be used for one frequency and only that one.

Figure 7.19: Image of a Crystal Oscillator

The equivalent circuit diagram for a crystal is shown below:

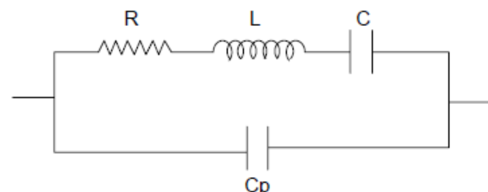


Figure 7.20: Equivalent Diagram of the Crystal

Figure 7.21: Circuit Symbol of the Crystal

5.00 MHz Crystal Oscillator

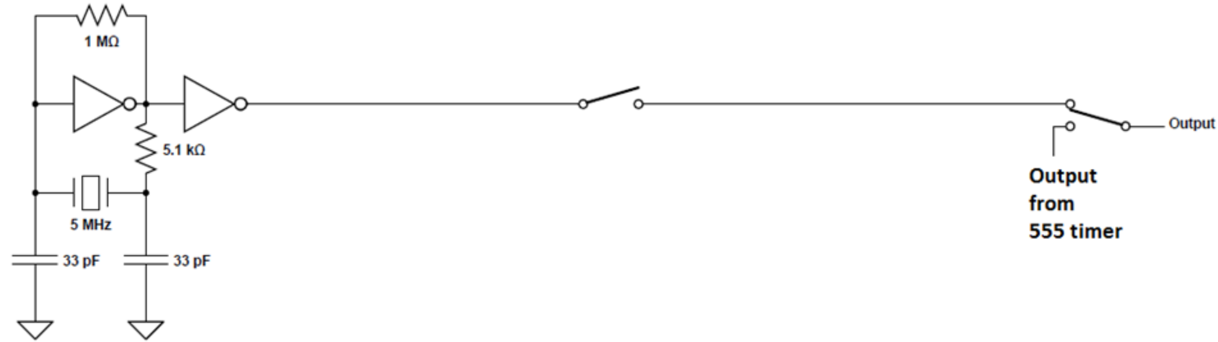


Figure 7.22: 5.00 MHz Crystal Oscillator Circuit

The configuration is known as a Pierce Oscillator. In this configuration the crystal operates in parallel resonance mode. For a crystal to oscillate correctly there are two criteria that must be satisfied. The criteria, also known as Barkhausen criteria are as follow:

1. The closed loop gain ≥ 1
2. The phase shift around the loop has to be 0 or 360n, where n is an integer multiple.

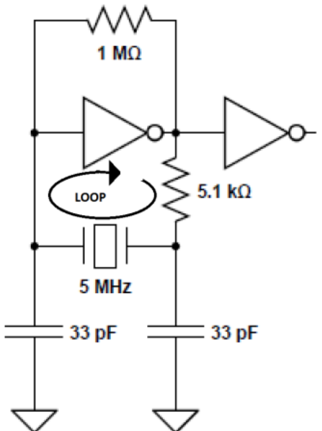


Figure 7.23: Crystal Oscillator is Parallel Resonance Mode

The purpose of the 5.10 kΩ resistor is to prevent overdriving the crystal by limiting the output. If a crystal is overdriven there exists a chance that it might become permanently physically damaged. The feedback resistor, 1.00 MΩ, turns the first inverter into an analog amplifier, to be precise a Class AB amplifier. The phase shift provided by the first inverter is 180 degrees, the other 180 degrees is provided by the two capacitors and the 5.10 kΩ resistor. The inverter at the output helps sharpen edges making the waveform more square like.

The V_{cc} is powered at +5.00 V while GND is grounded. One side of the 1.00 MΩ resistor is connected to pin A1 of the 74HC4049 inverter the other end is connected to pin Y1. A 33.0 pF capacitor is connected from pin A1 to ground. One side of the crystal is connected to pin A1, and the other end is connected to a free node on the breadboard. The 5.10 kΩ resistor is connected to both pin Y1 of the inverter and the end of the crystal that is connected to a free node. A 33.0 pF capacitor goes from this node to ground. Pins Y1 and A2 are connected to one another. Y2 is connected to 4A on the dip switch; 4B is connected to one of the pins of the toggle switch. Both NC pins are not connected to anything. A3, A4, A5 and A6 are all grounded.

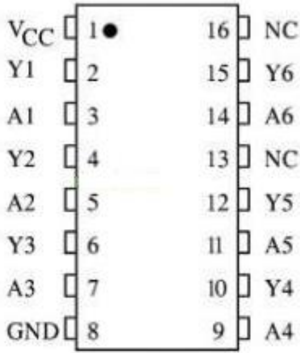


Figure 7.24: Top View of 74HC4049 Hex Inverter

Figure 7.25: Toggle Switch

Complete Clock Circuit Schematic

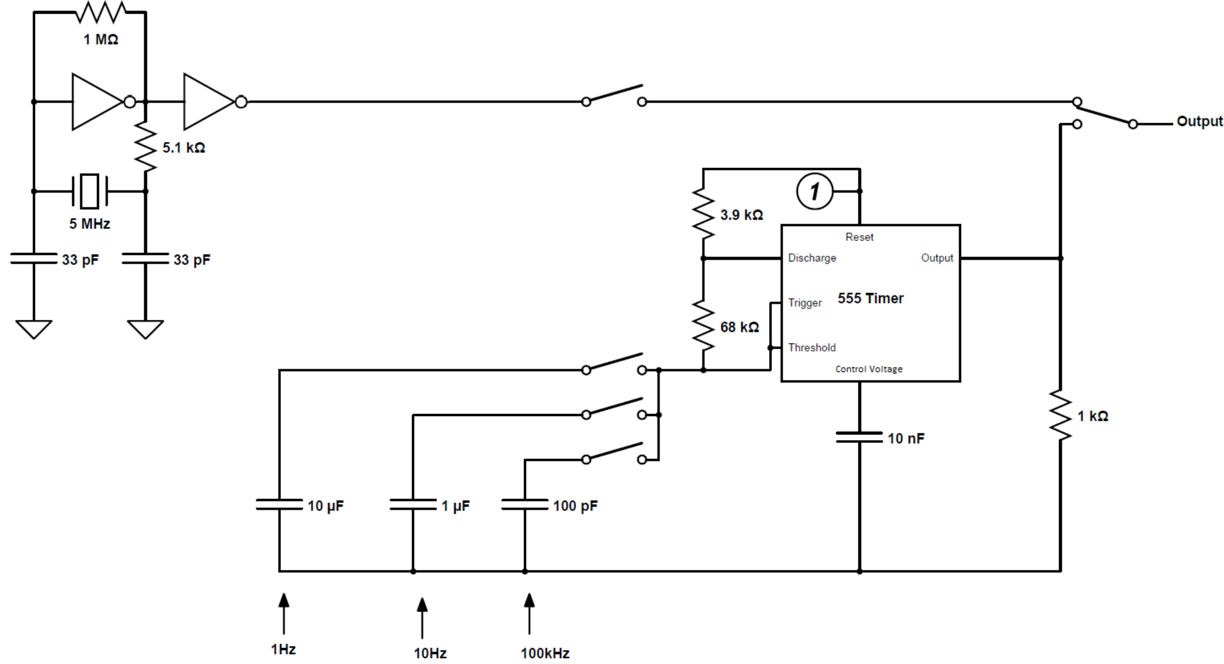


Figure 7.26: Complete Clock Circuit Schematic

The output, marked on the diagram, of this circuit is connected to the clock input of the CPU.

7.7 Raspberry PI VM and FPGA Code

Below are segments of the Raspberry PI Virtual Machine Code and the FPGA Implementation Code. These are only snippets and the entire code is attached in the Appendix, section A.2 CPU.

Raspberry PI Virtual Machine Code

```
#ifndef _VM4_H_
#define _VM4_H_
#define _POSIX_TIMERS
#define __USE_POSIX199309
#if __STDC_VERSION__ >= 199901L
#define _XOPEN_SOURCE 600
#else
#define _XOPEN_SOURCE 500
#endif /* __STDC_VERSION__ */
#include <stdio.h>
```

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <limits.h>
#include <unistd.h>
#include <inttypes.h>
#include <time.h>
#include <fcntl.h>
#ifdef RPI
#include <sys/mman.h>
#endif
#include <sched.h>
#define VERSTR "VM4 v0.3"
#define BOOTBIN "test.bin"
#define IOMEM 3
#define BOOTMEM 1024
#define RESERVEMEM (1 + IOMEM + BOOTMEM)
#define NOERROR 0
#define MEMALLOCERROR 1
#define UNKNOWNINSTRUCTIONERROR 2
#define UNKOWNERROR -1
#define MEMADDRSIZE 16
#define MEMSIZE 65536
#define BIGEND -1
#define LITTLEEND 1
#define MEMMODSIZE 4
#define PLATEND LITTLEEND
#define NUMREG 4
#define PC 0
#define A 1
```

```
#define MEM 2
#define STAT 3
#define HLT 0
#define LOD 1
#define STR 2
#define ADD 3
#define NOP 4
#define NND 5
#define JMP 6
#define CXA 7
```

CPU Instruction Set FPGA Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
entity alu_decode is
   Port ( exe : in  STD_LOGIC;
          OP_EN : in  STD_LOGIC;
          clk : in  STD_LOGIC;
                    clk_fast: in std_logic;
          rst : in  STD_LOGIC;
                    OP_Code : in  STD_LOGIC_VECTOR(3 downto 0);
          WE : out  STD_LOGIC;
          A_EN : out  STD_LOGIC;
          STAT_EN : out  STD_LOGIC;
```

```
        HLT : out  STD_LOGIC;
        JMP : out  STD_LOGIC;
        Arith_S : out  STD_LOGIC;
        Stat_S : out  STD_LOGIC;
        LOD_S : out  STD_LOGIC;
                STR : out STD_LOGIC);
end alu_decode;
architecture Behavioral of alu_decode is
signal stored_OP_Code : STD_LOGIC_VECTOR(3 downto 0);
signal i_WE : std_logic;

begin
process(clk)
--ADD A RESET THAT SETS OPCODE TO NOP
begin
  if clk' event and clk = '1' then
      --Write op_code into temp storage
      if OP_EN = '1' then
            stored_OP_Code <= OP_Code;
      end if;
  end if;
end process;

--process(clk)
--begin
--  --Execute instruction
--
--  if rst = '1' then
--      i_WE <= '1';
--      HLT <= '0';
--      A_EN <= '0';
```

```
--        STAT_EN <= '0';
--        JMP <= '0';
--
-- elsif exe = '1' then
--        --HLT
--        if stored_OP_Code = "0000" then
--                i_WE <= '1';
--                HLT <= '1';
--                A_EN <= '0';
--                STAT_EN <= '0';
--                JMP <= '0';
--
--        --LOD
--        elsif stored_OP_Code = "0001" then
--                i_WE <= '1';
--                HLT <= '0';
--                A_EN <= '1';
--                STAT_EN <= '0';
--                JMP <= '0';
--                LOD_S <= '1';
--
--        --STR
--        elsif stored_OP_Code = "0010" then
--
--
--                --if i_WE = '1' then
--                --        i_WE <= '0';
--                --else
--                --        i_WE <= '1';
--                --end if;
--
```

```
--			i_WE <= '0';
--			HLT <= '0';
--			A_EN <= '0';
--			STAT_EN <= '0';
--			JMP <= '0';
--
--		--ADD
--		elsif stored_OP_Code = "0011" then
--			i_WE <= '1';
--			HLT <= '0';
--			A_EN <= '1';
--			STAT_EN <= '1';
--			JMP <= '0';
--			Arith_S <= '0';
--			Stat_S <= '0';
--			LOD_S <= '0';
--
--		--NOP
--		elsif stored_OP_Code = "0100" then
--			i_WE <= '1';
--			HLT <= '0';
--			A_EN <= '0';
--			STAT_EN <= '0';
--			JMP <= '0';
--
--		--NND
--		elsif stored_OP_Code = "0101" then
--			i_WE <= '1';
--			HLT <= '0';
--			A_EN <= '1';
--			STAT_EN <= '0';
```

```
--              JMP <= '0';
--              Arith_S <= '1';
--              Stat_S <= '0';
--              LOD_S <= '0';
--
--      --CXA
--      elsif stored_OP_Code = "0111" then
--              i_WE <= '1';
--              HLT <= '0';
--              A_EN <= '1';
--              STAT_EN <= '0';
--              JMP <= '0';
--              Stat_S <= '1';
--              LOD_S <= '0';
--
--      --JMP
--      elsif stored_OP_Code = "0110" then
--              i_WE <= '1';
--              HLT <= '0';
--              A_EN <= '0';
--              STAT_EN <= '0';
--              JMP <= '1';
--
--      --Unknown - halt the CPU
--      else
--              i_WE <= '1';
--              HLT <= '1';
--              A_EN <= '0';
--              STAT_EN <= '0';
--              JMP <= '0';
--      end if;
```

7.8 Testing the CPU

The CPU operates as per the original scope; it operates at 5MHz and is integrated with the audio peripheral and keyboard peripheral. Tests were conducted to validate each operation and instruction performed by the CPU. The final CPU design also implements the custom made nibble knowledge architecture and instruction set, and interfaces with the bus.

7.8.1 FPGA Testbench: MEM Register Block

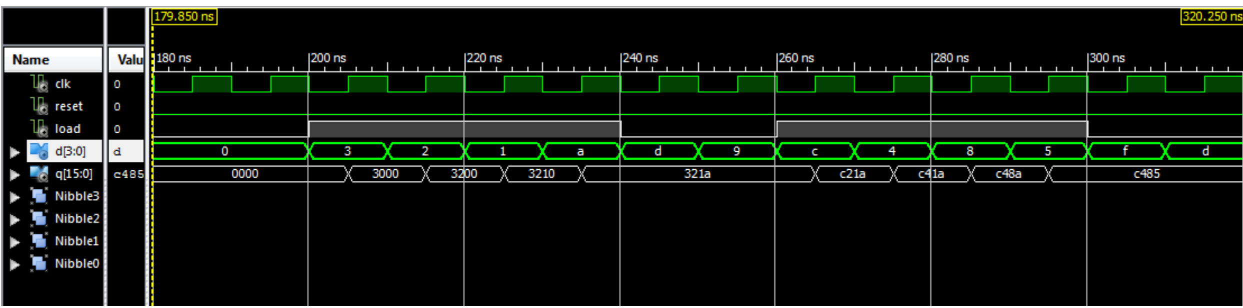


Figure 7.27: Testbench for CPU MEM Register Block

This testbench shows two instruction cycles of the CPU, depicted by the white waveforms.

1. When the load goes high, the register starts to load nibbles in into the register from the top nibble to the bottom nibble.
2. The load then goes low for two cycles. This allows for the Execute and OP code phases to pass before becoming high again for the next address.

Note: The cycles are controlled by the Control Unit, the register has no way of knowing if the cycles are correct. Also, the address becomes garbage while in the process of loading all for nibbles.

7.8.2 FPGA Testbench: Control Unit Block

Figure 7.28: Testbench for CPU Control Unit Block

The 16-bit adder is working as expected when the inputs change. Note, this does not have overflow detection.

7.8.4 FPGA Testbench: CPU Top Level Testbench

Test 1 - add 2 looped

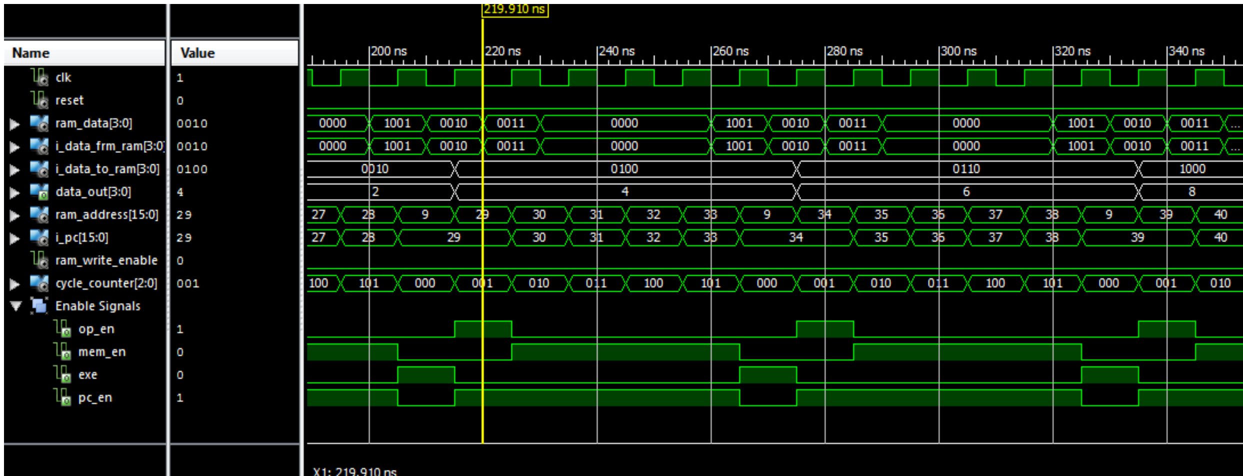


Figure 7.29: Testbench for CPU Top Level Block

This test validates that the Accumulator output is increasing by 2 every 6 cycles as expected. Note: The A is highlighted in white.

7.8.5 FPGA Test Programs

Test programs were loaded onto a ram chip using a loader developed using the Raspberry PI. This tested the full functionality and the developed logic of the CPU design, using real machine instructions. It also proved to be a test for the assembly and macro assembly languages; to see if they work before moving on to discrete circuits. Here is a sample of some of the test programs and outcomes achieved.

Test 1:

START:

LOD 3

STR 1025

LOD 4

STR 1026

LOD 5

STR 1027

LOD 6

STR 1028

LOD 7

STR 1029

LOD 8

STR 1030

LOD 9

STR 1031

LOD 10

STR 1032

LOD 11

STR 1033

LOD 12

STR 1034

LOD 13

STR 1035

LOD 14

STR 1036

LOD 15

STR 1037

LOD 16

STR 1038

LOD 17

STR 1039

LOD 18

STR 1040

LOD 3

JMP START

Expected Results: Test 1

The values 1-15 should be in the locations 1025-1040. This was observed after multiple test runs at 10 Hz, 100Hz, 1 kHz, and 100 kHz

Test 2 (allinst.bin)

```
19      LOD 4
24      STR 1025
29      ADD 18
34      STR 1026
39      CXA
44      STR 1027
49      LOD 13
54      ADD 11
59      STR 1028
64      CXA
69      STR 1029
74      LOD 14
79      NND 18
84      STR 1030
89      LOD 3
94      JMP OVER
99      LOD 18          ; should never get here
104     STR 1000
        OVER:
109     LOD 15
114     STR 1000
119     HLT
124     LOD 18          ; should never get here
129     STR 1000
```

Expected Results: Test 2

1000: 12 (Stored 12)

1025: 1

1026: 0

1027: 1 (Just carry is on)

1028: 2 (Result from adding 8 to 10)

1029: 9 (XOR and carry on)

1030: 4 (Result of NAND 11 is 4)

Results matched what was expected after many trials.

Test 3 (whileloop.bin)

```
int  i = 0;
while( i < 8){
        i++;
}
```

```
;int i = 0
LOD 18          ;Loads 15 into A        1025
        STR 1200        ;Stores i at 1200       1030
LOOP:           LOD 1200        ;Loads i into A                 1035
                ADD 4           ;Adds 1                 1040
                STR 1200        ;Stores i at 1200       1045
        ;if (7 >= i)
                NND 18          ;Negates i              1050
                ADD 4           ;Twos-Compliment i      1055
                ADD 10          ;Add 7                  1060
                CXA                                     1065
                NND 11                                  1070
                NND 18          ;AND with 8             1075
        ;Jump if (7 >= i)
                JMP LOOP        ;Jump if above true     1080
                HLT             ;End function           1085
```

Expected Results: Test 3

1200: 8

Results matched what was expected.

7.8.6 Discrete Testing

Most of the discrete testing for the CPU was done by creating a program and making sure the outputs matched. The Raspberry Pi was used as a ROM loader to load programs onto the ROM chip, and then put that chip back into the discrete circuit. The Pi also had a checker that read back in the ROM and confirmed if the program was correct before it was used in testing. This eliminated the possibility that the program loaded incorrectly. To test the programs at slow speeds, an Arduino was used as the CPU clock. This sped up the boot up (around 16000 clock cycles) and allowed stepping through the program as it was run; the Arduino outputted a clock pulse every time the enter key was hit on the connected computer. The code for this is shown below. The code for the ROM loader can be found on GitHub with the name "FPGARAM".

```
int Delay = 1;
void setup() {
  // put your setup code here, to run once:
  pinMode(12, OUTPUT);
  pinMode(5, INPUT);
  digitalWrite(12,LOW);
  Serial.begin(9600);
}
void loop() {
  // Reset the CPU by sending a clock pulse then
  // informing user to release the reset button:
    digitalWrite(12, HIGH);
    delay(1);
    digitalWrite(12,LOW);
    Serial.println("LET GO");
    delay(4000);
```

```
  //Skip through boot up
 for(int i = 0 ; i < 16383 ; i++){
  digitalWrite(12, HIGH);
  delay(Delay);
  digitalWrite(12,LOW);
  Serial.println(i);
 }
 //Used to step though additional clock cycles if need be
 for(int i = 0; i < 0; i++){
  digitalWrite(12, HIGH);
  delay(1);
  digitalWrite(12,LOW);
  Serial.println(i);
 }
 //Pulse clock when any key pressed
 while(1){
  Serial.flush();
  while(!Serial.available()) ;
  digitalWrite(12, HIGH);
  delay(Delay);
  digitalWrite(12,LOW);
  delay(Delay);
  Serial.read();
 }
}
```

To ensure everything matched expectations, the above “allinst.bin” program was run; it checked the outputs with a multi-meter or LEDs. This took a long time, and there were many wires that were plugged in backwards, or one hole off.

The peripherals and bus were integrated, once it was confirmed that the CPU ran every instruction. Audio was tested first, as it was an easy driver to write; simply send a value and a sound would

play. There were a few issues with this peripheral. The keyboard was then added to play a sound based on the key pressed. Similar to the audio, the keyboard was not too difficult to implement. However, adding peripherals beyond that became much more challenging. The hard drive required very time and level sensitive data, and VGA itself was very inconsistent. The hard drive worked occasionally, and the VGA worked well. For more information, refer to their respective sections. Serial was tested with an FPGA, it successfully received data from the discrete CPU. Due to time constraints, the discrete implementation was not integrated.

Many other programs were tested to verify that other parts of CPU worked. For example, a program was loaded at address 50000 of the RAM to make sure this would work and found two high address lines crossed. Programs were loaded that jumped from low to high addresses. These worked as expected.

7.10 CPU Specifications

Component	Specifications
Clock Frequency	5 M Hz with variable 1 Hz, 10 Hz, 100 kHz
Data Bus	4-bits
Chip Select	4-bits
Parity Check	1-bit
Ready	1-bit
Write	1-bit
Read	1-bit
Register A	4-bits
Register STAT	4-bits
Register MEM	16-bits
Register PC	16-bits
Register OP	4-bits
Memory	64 kB
Input/output	Memory Mapped I/O
Bus Protocol	Master-Slave Protocol
Clock Circuit Voltage	5.00 V

Table 7.5: CPU Specifications

8. Audio Controller

8.1. Introduction

The Personal Computer (PC) Audio Controller, which is also known as the sound card in a modern day computer, is a removable computer expansion card, which, under the control of computer programs can input and output sound. The purpose of the sound card is to provide the audio for applications such as games, movies and music. The audio controller in the case of the Nibble Knowledge computer consists of two separate circuits, whereas a sound card only consists of one mode. The typical audio controller/sound card consists of a digital-to-analog converter (DAC), and the signal is led to a connecter, where an amplifier can be plugged in. Prior to the invention of the sound card, the only PC software that could produce music and sound was the internal PC speaker, which could only output beeps.

An audio controller was first developed and marketed by Adlib in 1989; the other producer was Creative Labs, a company which is still the best-known sound card manufacturer in the world. Creative Labs produced a line of cards known as the sound blaster, which have set the standard for sound cards today. The first developed sound cards were huge, and the volume controls were on the cards itself. The user had to reach behind the PC to change the volume. In modern computers, the same technology used to build these sound expansion cards is directly integrated within the computer itself.

The Low-Level design of the Audio Controller could be broken down into five major components as shown in Figure 8.1 below. These components are the Latch, Digital to Analog Converter (DAC), Variable Clock, Amplifier and lastly, the Speaker. The Bus is the interface that enables communication between the CPU and the Audio Controller, along with ensuring appropriate communication standards are met. The CPU and the BUS are not the domains of the Audio Controller

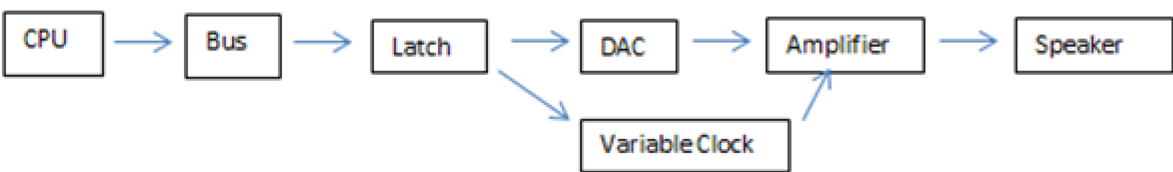


Figure 8.1: High Level overview of the Audio Controller

8.2 The Latch

As learned in the earlier chapter of Combinational and Sequential Logic, latches are very useful devices. A latch is an array of D flip-flops. One of the many applications of a latch is to act as a holding register that stores incoming data from the CPU's data bus. The latch is used to ensure that the correct data is sent down the bus at the correct time, which is every 125 microseconds (μs).The latch used in the Nibble Knowledge audio controller is CD4508BE. CD4508BE is a CMOS 4-bit dual latch, hence, for the purposes of the 4-bit computer, only one side of the latch will suffice. The diagram below shows the used pins of the integrated circuit. The red rectangles indicate the necessary pins while the green rectangle indicates the three pins that are grounded in the actual circuit.

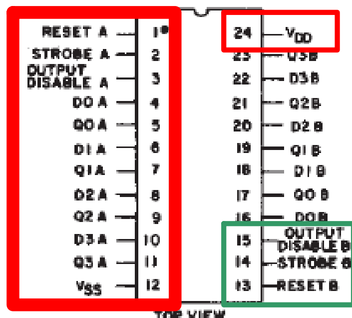


Figure 8.2: Top View of the CD4508BE Latch Chip

All digital components of the audio controller circuit have a positive voltage of 5.00 V, thus the latch is also powered with 5.00 V. A capacitor of 100 nF is included in the circuit schematic between VDD and VSS to filter out noise from power supplies. VDD is powered with 5.00 V while VSS is connected to the ground. The data lines plug into pins D0A, D1A, D2A, and D3A respectively, while the output of the latch are Q0A, Q1A, Q2A, Q3A respectively.

The table below is a truth table for the CD4508BE Chip. It shows that for Side A (left side of the chip in Figure 8.2) of the latch to work, disable must be connected to the ground while strobe is connected to the positive output of one of the two-input AND gates. The following logic ensures that the latch is being strobed at the correct time and data is being sent further down the circuit. The green shaded row shows that to disconnect the B side of the latch, the reset, disable and strobe must be connected to the ground.

Truth Table for the CD4508BE				
Reset	**Disable**	**Strobe**	**D Input**	**Q Output**
0	0	1	1	1
0	0	1	0	0
0	0	0	X	LATCHED
1	0	X	X	0
X	1	X	X	Z

Table 8.1: Truth Table for the CD4508BE Latch chip.

The Chip Select (CS) is connected to 1A of the 7404 chip, the output, 1Y is connected to 1B of the AND chip. All other inputs are connected to the ground. V_{CC} is connected to +5.00 V, while GND is connected to ground.

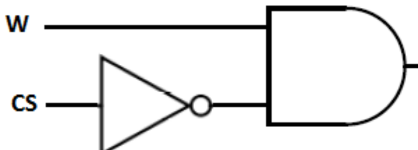


Figure 8.3: Circuit Schematic of Chip Select and Write Control Line

1A is connected to the write control line (W). In the Hex Inverter 7404 Chip, pin 1Y is connected to 1B on the 7408 AND chip. All other inputs are connected to the ground.

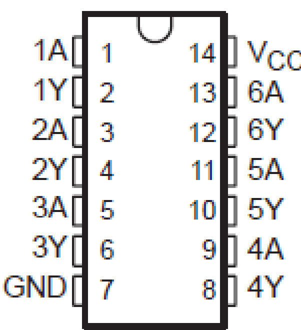


Figure 8.4: Top chip view of the 7404 Hex Inverter Chip

In the 7308 AND chip, the 1Y pin on the AND chip is connected to strobe on the CD4508BE Latch Chip.

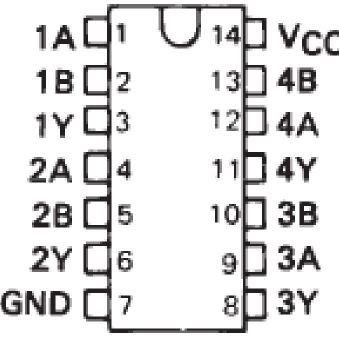


Figure 8.5: Top chip view of the 7408 AND Chip

The Reset can be hot-wired to ground, or one can also manually reset the circuit by adding a simple circuit shown below and connect it to the reset pin on the latch.

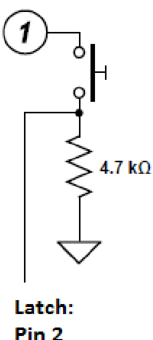


Figure 8.6: A Simple Reset Circuit

8.3 Digital to Analog Converter (DAC)

A digital-to-analog converter or DAC are devices that take digital data (usually binary) and convert that data into analog signals (current, charges or voltage). DACs are found in audio devices (CD players, digital music players and PC sound cards), and video devices.

One of the very first DACs were not electronic, in fact, it were hydraulic. First developed in the time of the Ottoman Empire in Turkey, DACs built to meter water; functionally, it was an 8-bit DAC. Instead of taking digital signals as input, it took a manual input, and a “wet output.” It is believed to be the world’s oldest digital-to-analog converter.

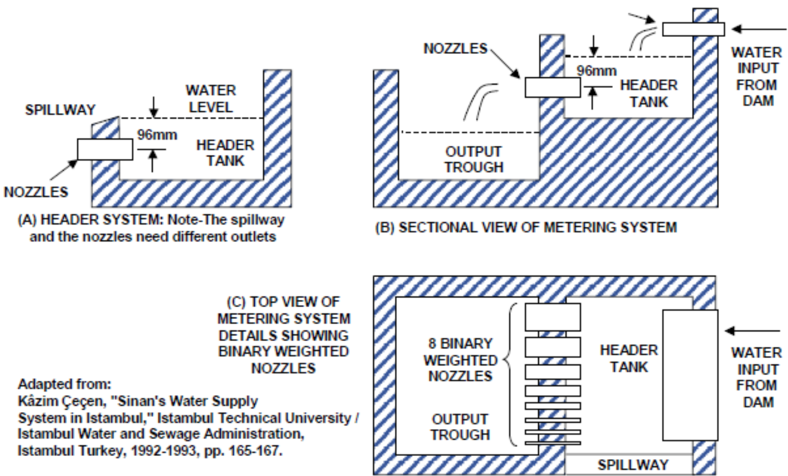


Figure 8.7: The First Digital to Analog Converter System

The primary driving force for the invention of converters was in fact the digital computer. Prior to the 1950s, converters were developed and only used for specialized tasks such as message encryption systems of World War II. The only technology available to build the converters was vacuum tubes, making the converters expensive, huge and inefficient. There was no commercial usage. However, with the invention of the transistor in 1947, the converter's capability increased drastically, making it smaller, efficient and more powerful. Commercial usage of the DAC became more prevalent, with applications in measurements, medical imaging, audio, computer graphics, even modems and wireless infrastructure. There are many types of digital-to-analog converters, one of which is mentioned in this text.

Generally, DAC devices do not contain clocks. The clock is associated with the CPU; the CPU sends both the clock and the digital data to the DAC. According to Nyquist's Theorem of Digitization, in order to properly replicate the signal: $f_S = 2 f_m$, where f_m is the maximum frequency of the signal and f_S is the sampling frequency. Since the quality of the system is not required to be very high, the maximum frequency of a telephone which is around 4 kHz is sufficient to meet the objective. Thus, in this audio controller system $f_S \cong 8$ kHz will suffice in recreating the signal without significant data loss. This results in the sampling time being $t_S = 125$ µs (microseconds).

The circuit used to implement the DAC is the binary-weighted resistor. It is a simple design with an operational amplifier, (in short referred as op-amp) already integrated into the design. The op-

amp amplifies the signal which is then further amplified by the amplifier circuit at the output of the op-amp. One of the major disadvantages of the binary-weighted resistor circuit is that there needs to be high precision in small resistor values, two ways around that would be to use resistors with 1% tolerance and potentiometers, rather than using resistors with 5% tolerance. This disadvantage would be more prevalent in higher resolution systems, but for the current design objective, the difference is negligible. Note: the resistor values were chosen arbitrarily.

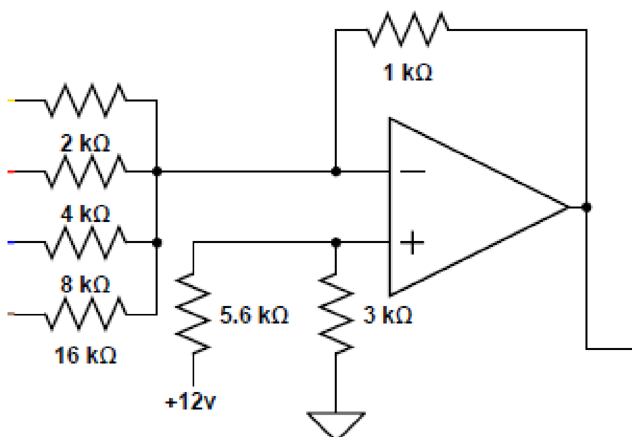


Figure 8.8: Circuit Schematic of the DAC

A standard LM741 Op-amp was used in the circuit implementation. Typically the op-amp would be powered with a positive and negative voltage (+12V and – 12V). In this case, V+ is powered at +12.0 V while V^- is directly connected to the ground. To compensate for the single-ended power supply, a virtual ground had to be constructed, and it was must to include resistors at the non-inverting input.

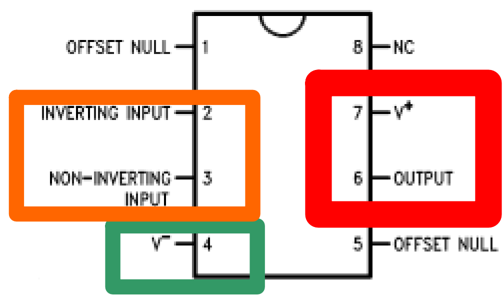


Figure 8.9: Top View of LM741 Operational Amplifier Chip

Below is a table that represents digital bits and the corresponding analog values of the digital-to-analog converter.

Digital				**Analog**
D_3	D_2	D_1	D_0	V_{out} (-V)
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Table 8.2: Digital bits and the corresponding analog values

The smallest resistance is the most significant bit, which is 2.00 kΩ (2.00 kΩ is the Q3A output from the latch).

The advantages of this DAC are that it is very simple from a hardware implementation perspective and the conversions are fast. The disadvantages are that the large range of resistor values require high precision and the circuit has small switch resistance.

8.4 Amplifier Circuit

As the name suggests, an amplifier is a device that increases the volume of the sound so that it could be heard through a speaker. It is the last step in the process of sound moving from the input to the output. Some applications of amplifiers are in public address systems and concerts. A man

by the name of Lee De Forest developed the very first audio amplifier in 1906. It used a triode vacuum tube which consisted of three elements. The triode was designed to modulate the sound by adjusting the movement of electrons moving from a filament to a plate.

Introduced in 1946, the Williamson amplifier, which uses vacuum tubes was considered top of the edge, producing higher quality sound compared to its competition. Most modern audio amplifiers are called solid state transistors, they offer high efficiency and convenience, the only downside is that they could not replicate the quality of those amplifiers made with vacuum tubes. This was due to distortion, called intermodulation distortion, caused by the rapid increase of voltage in the output device. Intermodulation distortion is no longer a concern in audio amplifiers.

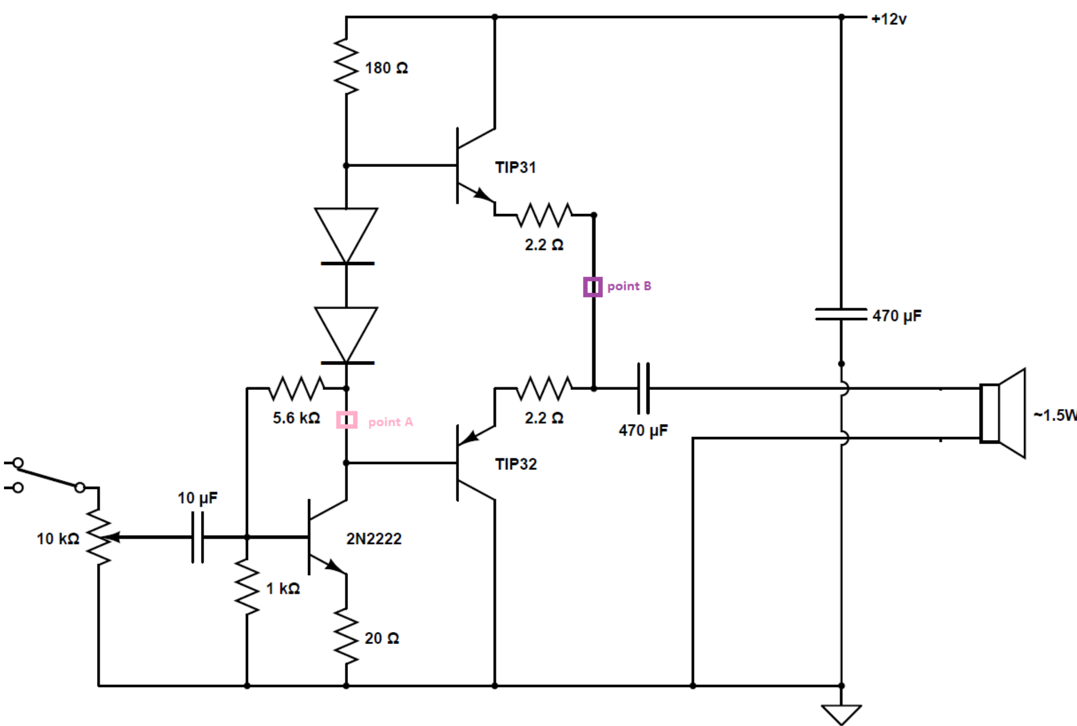


Figure 8.10: Circuit Schematic of the Amplifier Circuit

The amplifier circuit is used for amplifying the sound intensity. The amplifier circuit being implemented is a Complementary Power Amplifier circuit with Driver and Auto-bias. This circuit has an advantage of offering outstanding thermal compensation. Changing the 10.0 kΩ potentiometer value, it is possible to increase or decrease the intensity of the sound. The amplifier circuit consists of a complementary pair of NPN and PNP transistors. In this case, TIP31 and TIP32 were the chosen transistors but any complementary pair (similar to the TIP31 and TIP32) would

suffice. The 2N2222 transistor acts as pre-amplifier driver for the complimentary pair of transistors. All three transistors in the amplifier must operate in the active region because the output signal must follow the input signal, multiplied by amplification constant. If a transistor happens to go into saturation mode even for a brief period of time, the output signal (at the speaker) will be clipped and distorted, which means the output does not properly follow the input signal. The design incorporates 2.20 Ω resistors, which are bigger than resistors that are normally used because the resistors have to be able to tolerate a higher current. Focusing on a small section of the circuit shown below it may be assumed that the speaker impedance is approximately equal to 4.00 Ω. The power going through the resistor can then be calculated as:

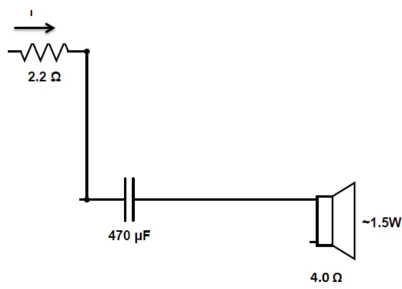


Figure 8.11: Enlarged Section of the Amplifier Circuit

$R_T = 2.2 + 4$

$R_T = 6.6\,\Omega$

$V = I \times R$

$I = \frac{V}{R}$

$I = 1.00\,A$

$P = I \times V = I^2 R$

$P = 1.0^2 \times 2$

$P = 2.0\,W$

The entire amplifier circuit is powered by a single ended power supply, when there is no signal at the DC component at Point A (Figure 8.10), the voltage should be close to $V_{cc}/2$ V. Because it is a single-ended supply a 470 µF capacitor must be used to block the DC current from passing through the speaker. The 470 µF capacitor that is connected between V_{cc} and the ground is used to block noise from the power supply. As shown in the enlarged section of amplifier circuit, the

470 µF capacitor connects to the orange wire in the below image. The black wire is connected to ground.

Figure 8.12: Capacitor and Ground Wire connected with the Speaker

The driver and speaker should have the same impedance to ensure maximum energy transfer. In reality, for the Nibble Knowledge project, it is not critical to ensure maximum energy transfer, because at the end, the objective is to only hear some sound and maximum energy efficiency is not a concern. Point B (Figure 8.10) moves either towards V_{cc} or ground to ensure that the speaker is provided with a continuous alternating signal.

The three transistors have the following configuration:

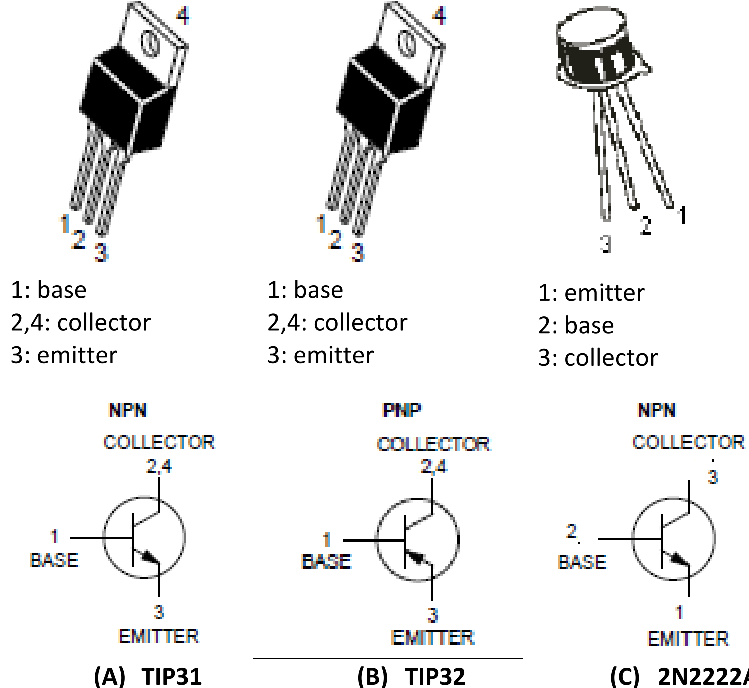


Figure 8.13: Configuration and Circuit Diagrams of Three Transistors

8.5 Variable Clock

Variable Clock Oscillator is a very simple circuit that is designed to change the frequency of the sound. It requires a constant input from the bus. It consists of a circuit that sets a controlled frequency, a down counter and another frequency divider made using a chip. The controlled frequency is set at a value of 22.4 kHz that is sent to the counter and based on the data bits sent to the counter, the frequency is 'divided' by the value of the bits. It may be said that the counter is preset by the value of the 4-bit data lines. The frequency is then 'divided' in half, this is to ensure the pitch is within the audible range, so a person can hear; it also serves the function to make the signal have a 50% duty cycle. A 50 % duty cycle means that the high and low levels of the signal are equal in time duration. The variable clock circuit is controlled by a switch that is located at the top of the potentiometer. When the toggle switch is turned in one direction, the other mode is connected (i.e. when the switch is turned to the DAC side, the variable clock oscillator is connected to the rest of the circuit.)

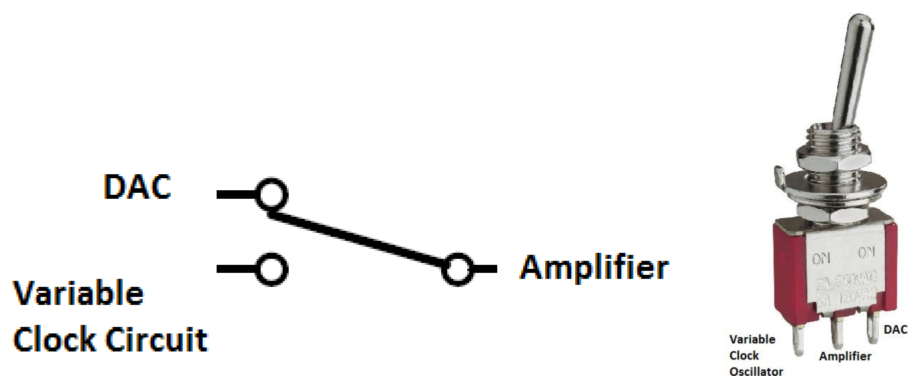


Figure 8.14: Toggle Switch Configuration

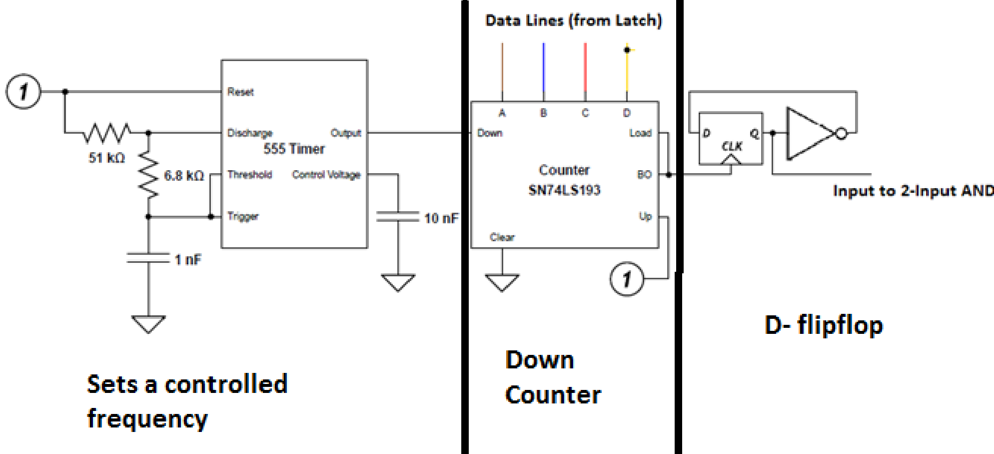


Figure 8.15: Controlled Frequency and Down Counter Circuit

Wires have to be soldered onto the pins of the switch. Vcc is connected to +5.00 V, while GND is connected to ground. Reset pin on the SA555 is also connected to +5.00 V. The output (pin 3) is connected to the down pin on the SN74193 down counter. One end of the 10 nF capacitor is connected to the control voltage while the other end is connected to ground. Tigger and threshold pins are connected together at the same node and a 1nF capacitor is connected between those pins and ground. A 6.80 kΩ resistor is connected between discharge and trigger. The 51.0 kΩ resistor is connected between reset and discharge.

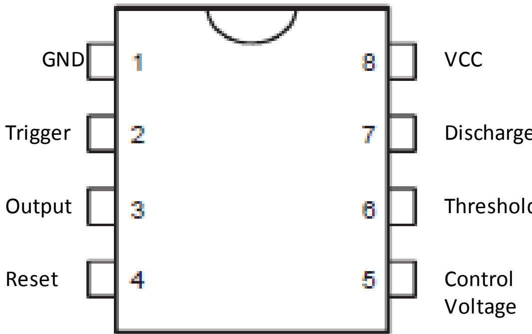


Figure 8.16: Top View of the SA55N Timer Chip

Vcc is connected to +5.00 V, while GND is connected to ground. Up is also connected to +5.00 V. The outputs; pins QA, QB, QC, and QD are left floating. The down pin is connected to the output pin of the 555 timer. Clear is connected to ground. BO and load are connected together and the load is connected to the clock of the D flip-flop. The inputs to the chip are A, B, C and D are they are connected to their respective data lines.

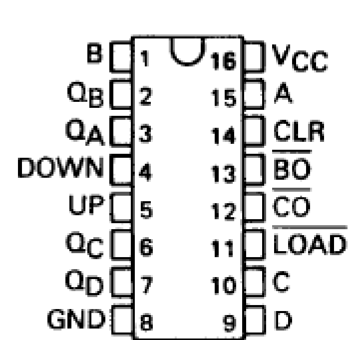


Figure 8.17: Top View of the SN74193 Preset Counter Chip

In the SN74193 Preset Counter Chip, the data lines are connected to A, B, C and D as follows:

- A = Q0
- B = Q1
- C = Q2
- D = Q3

Vcc is connected to +5.00 V, while GND is connected to ground. The load of the counter chip is connected to the clock of the D flip-flop (1CLK). $1\overline{Q}$ is connected to pin 1D. 1Q is connected to one of the inputs of the 2-input AND. $1\overline{CLR}$ and $1\overline{PRE}$ are both connected to high. $2\overline{CLR}$ and $2\overline{PRE}$ are both connected to low.

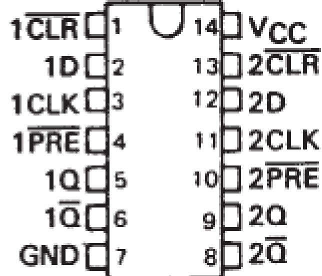


Figure 8.18: Top View of the 7474 D Flip-Flop Chip

The table below shows the bit pattern and the calculated frequency for that bit pattern.

Bits	Calculated Frequency (Hz)
1111	746.7
1110	800
1101	861.5
1100	933.3
1011	1.018 kHz
1010	1.120
1001	1.244
1000	1.4
0111	1.6
0110	1.867
0101	2.24
0100	2.8
0011	3.733
0010	5.6
0001	11.2

0000	10.2

Table 8.3: Bit Pattern and Corresponding Frequency

<u>Sample calculation:</u>

Using a base value of 22.4 kHz, following formula could be used to calculate the frequency at the output of the d flip-flop. Below, the calculation uses bit value of 16.

$$= \frac{base\ frequency}{2 \times bits}$$

$$= \frac{22400}{2 \times 16}$$

$$= \frac{22400}{32}$$

= 700 Hz

The zero bits can be heard, but it is a very high-pitched sound, so by adding a simple circuit as shown below, it is possible to simply shut off the audio controller completely. The OR gate passes 1 when at least 1 bit is a 1 and a zero whenever all bits are zero. The AND in front of the OR gate passes a zero whenever the output from the OR is a zero, this is the same as OFF, otherwise it passes a 1 when both the output from the flip-flop is a 1 and the output from the OR is also a one.

Data Lines (from Latch)

4 input OR

Q output (from D Flip-Flop)

Figure 8.19: Data Lines from the Latch to the Output D Flip-Flop Circuit

Vcc is connected to +5.00 V, while GND is connected to ground. Pins 4A and 4B are connected to ground. Output from latch (Q0) is connected to 1A; output from latch (Q1) is connected to 1B. Output from latch (Q2) is connected to 2A; output from latch (Q3) is connected to 2B. 1Y is connected to 3A, and 2Y is connected to 3B. 3Y is connected to one of the inputs to the AND gate. Pins 4A and 4B are connected to ground.

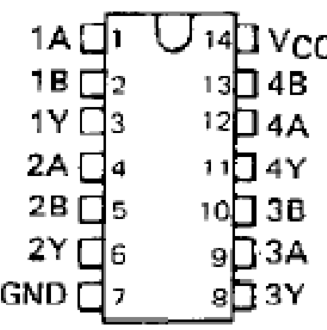


Figure 8.20: Top View of the 7432 OR Chip

Vcc is connected to +5.00 V, while GND is connected to ground. 1A is connected to pin 3Y on the OR chip. The Q output (D flip-flop mentioned above) is connected to 1B. 1Y being the output of the variable clock circuit is connected to the switch. Pins 2A, 2B, 3A, 3B, 4A and 4B are all connected to ground.

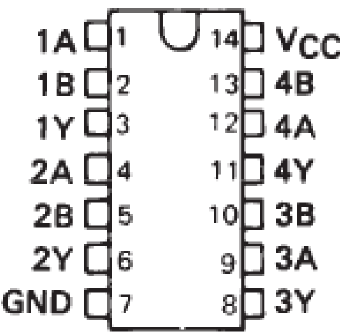


Figure 8.21: Top View of the 7408 AND Chip

8.6 The Final Product

The Audio Controller final circuit schematic consisting of the Latch, Digital to Analog Converter (DAC), Variable Clock, Amplifier and lastly, the Speaker is shown below:

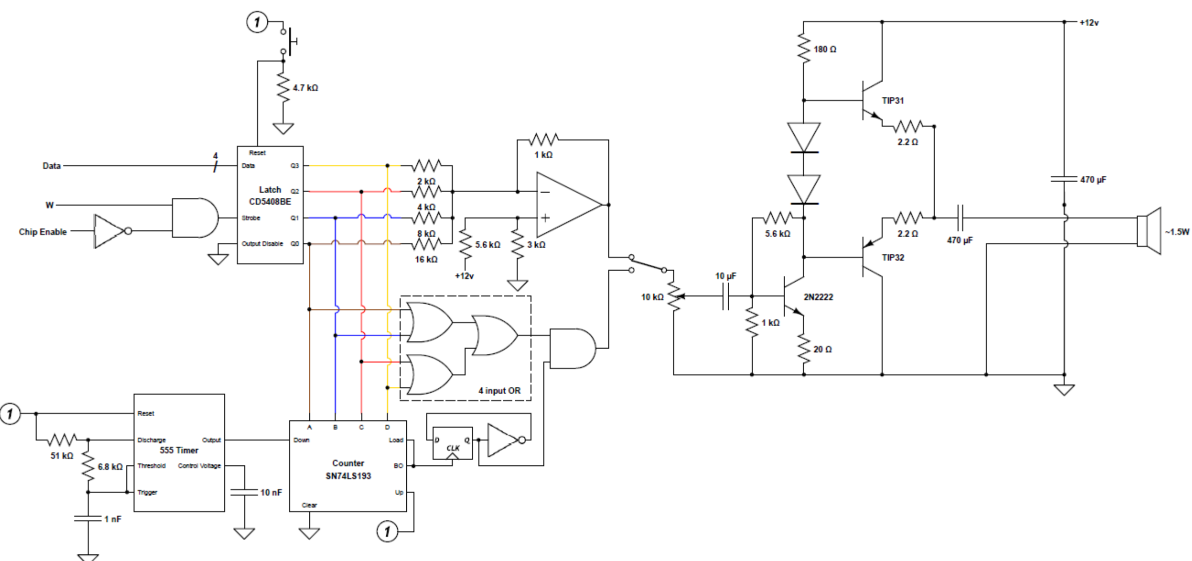


Figure 8.22: Final Circuit Schematic of the Audio Controller

As the name implies, the audio controller outputs sound based on inputs from the CPU. It has the capability of not only outputting simple beeps but also melodies such as He's a Pirate from Pirates of the Caribbean, the Star Wars theme song and Vangelis' Chariots of Fire. For the audio controller to play complex melodies, data must be stored in computer memory and then sent to the latch, otherwise for the debugging circuit it is not necessary for data to be stored.

For the debugging mode of the audio controller, the product while capable of changing pitch, does not increase the volume of the sound automatically because the pitch is not encoded in the bits. It must be changed manually by adjusting the potentiometer. For the sound card mode, both the pitch and the intensity of the sound are encoded in the bits from the CPU.

The current design is one of the simplest ways to design the audio controller. There are other ways to design the DAC, and the binary-weighted resistor is the easiest and most straightforward. There are different ways to make the input circuit, however, the functionality changes. If the circuit were to be changed to add a few more components, the intensity of the sound could be changed.

8.7 Testing the Controller

The measure of success for the audio controller is its ability to successfully communicate with the CPU and output sound through the speaker. The audio peripheral is connected to the CPU via three 4-bit lines, which are the data lines, status lines and chip select lines. In the final design, the audio controller circuits are implemented and verified. The audio controller was verified through discrete implementation and it successfully outputted sound through speakers.

Tests that confirm the discrete implementation of the CPU, audio and PS/2 keyboard are considered to be relevant. Specifically, tests were conducted to verify the capturing of data from the keyboard and sending it to the audio through the CPU. Tests that use the CPU to send data to the audio are also relevant, as they showcase the audio controller's ability to output sound.

Over twenty tests were performed on the audio controller. The first few tests were done on components as the circuit was built. The component tests were conducted using a 9.00 V battery as the power supply.

The most significant tests performed are listed as follows:

- The first test was testing the amplifier circuit by itself. An mp3 player was hooked to the circuit and a song was played over the cable. The test confirmed that the amplifier worked as intended.
- The second test was performed on the op-amp and amplifier circuits.
- The third test was the last test performed with the 9 V battery; at this point the entire circuit was completed. The audio circuit itself wouldn't work without an input from the "data bus". As such, an oscillator circuit was constructed using a SA55N timer circuit and 4520 counter. The complete oscillator circuit is shown in the Appendix A1. Prior to testing with the power supply, which has its own separate +5.00 V line, a voltage regulator chip was included to reduce the +12.0 V to +5.00 V.
- The third last test was performed using a PC ATX power supply. The oscillator circuit was used to send a signal into the audio circuit. For this test, capacitors were connected in parallel to remove noise generated by the power supply. The capacitors were not necessary for the battery tests because the battery is stable. The output on the oscilloscope should look like a saw-tooth waveform, and this was verified.
- The next test removed the oscillator from the circuit, and hooked up a virtual machine made with a raspberry pi. The strobe input pin of the latch was connected to the clock pin of the raspberry pi. Data lines from audio were connected to the data pins on the raspberry pin. Code one in Appendix A1 was used to test only the audio peripheral. It worked as well.
- Testing the audio circuit separately, an Arduino was used as an input to the input circuit. The program was a counter designed to count up to 15. It was supposed to test the variable

clock circuit that has been designed to change the pitch of the sound. The digital pins 2 – 5 for the data lines, 7 as enable, 8 as parity, and 9 as write were used. This circuit worked as well.

- Using an Arduino, two peripherals were hooked up: the audio and the keyboard. The test was setup to emit sound when a button was pressed; when another button was pressed, another pitch was heard. This test also worked.
- The hard drive and the audio peripheral were hooked up. Bit patterns were stored on the hard drive, and then the hard drive was read and patterns were sent to the audio peripheral. This was performed using an Arduino. This test also worked.
- The finals test were conducted on the discrete versions of the audio controller and the CPU; a program was loaded onto the ROM chip. The program outputted data through the data lines and into the audio; and both the CPU and the audio controller performed their tasks accordingly. This test was performed several times to verify the consistency of the results. The final result confirmed it worked correctly.
- The last test also tested the discrete version of the audio controller, the CPU and the keyboard. The program loaded onto the ROM read inputs from the keyboard and sent the corresponding bits to the audio. Every time a different key was pressed, there was a different pattern of bits sent. This test was also performed several times to test the consistency of the results. The final result confirmed it worked correctly.

8.8 Controller Specifications

Component	Specifications
Digital Component Power Supply	5.00 V
Analog Component Power Supply	12.00 V

Table 11: Audio Controller Specifications

9. IDE Controller

9.1 Introduction

A hard drive is a non- volatile mass storage device. It can be used to store files without the risk of losing them when power is turned off. It is used for storing and retrieving digital information, and is responsible for storing all content of a computer from files, programs, to operating systems. Hard drives are much slower than other types of memory such as RAM/ROM, but have significantly higher capacities. The cost of hard drives is also much lower per GB, compared to other types of memory. Currently, a 1TB hard drive is around $60 CAD, and a 4GB RAM chip is $50 CAD.

The disadvantage of a hard drive is its memory. CPU caches and RAM have a high speed memory that is much faster than the magnetic storage of a hard drive. However, it would be extremely expensive to build a high capacity drive out of such memory. In the case of the Nibble Knowledge CPU, the latency of a magnetic disk is actually fast enough to keep up with the slow speed that our CPU runs at.

9.2 The IDE Cable and Pin Layout

Integrated Drive Electronics IDE), is also commonly known as PATA, Parallel Advanced Technology Attachment. The IDE/Parallel ATA cable is a 40 pin cable that transfers data in parallel to the hard drive. It is an interface standard used for the connection of storage devices in computers, and is usually used as an internal computer storage interface.

The pin layouts for the IDE cable are shown below, we will ignore the signals that are not used for simplicity's sake.

Figure 9.1: IDE 40 Pin Cable

Below is a table that describes the 40 Pins on the IDE, along with the Signal and usage

Pin	Signal	Use
1	Reset	Low enabled reset
2	Ground	
3-18	Data lines 0-15, (Not in that order)	Used to transfer data over the interface
19	Key	Empty to show user the orientation of cable connector

20	Ground	
21	DDRQ	Not used in NK4
22	Ground	
23	DIOW	Pulse to write data
24	Ground	
25	DIOR	Pulse to read data
26	Ground	
27	IORDY	Not used in NK4
28	Cable Select	Not used in NK4
29	DMACK	Not used in NK4
30	Ground	
31	INTRQ	Not used in NK4
32	IOCS16	Not used in NK4
33	DA1	Register address 1
34	PDIAG	Not used in NK4
35	DA0	Register address 0
36	DA2	Register address 2
37	CS1FX	Chip select, used for high bits of register address
38	CS3FX	Chip select, used for high bits of register address
39	DASP	Not used in NK4
40	Ground	

Table 9.1: IDE 40 Pin Assignments and Description

9.3 The IDE High Level Design Overview

The interface between the Nibble CPU and the hard drive takes four bits at a time and saves each one until there are five nibbles saved, and then executes the DIOR/DIOW pulse. This is implemented with five 4-bit registers, and a timing control unit that enables one of them at a time. For a read, it was necessary to hold the DIOR line low until all four nibbles had been sent to the CPU, and then raise it again. The top level design is shown below.

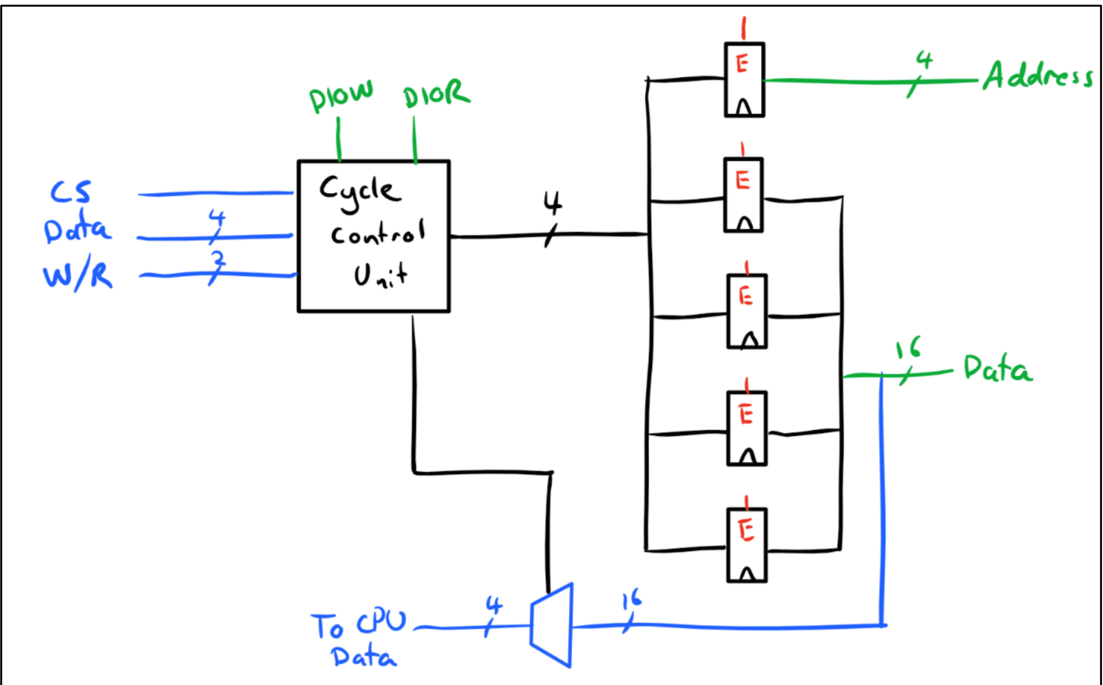


Figure 9.2: IDE High Level Circuit Schematic

In the figure above, the blue line indicates a signal that leads to the CPU and the green line indicates signals going to the hard drive. The bottom multiplexer (MUX) controls which 4-bits are coming from the hard drive on a read and have to be sent to the CPU, if it is a write, then this is turned off. The enables coming from the cycle control unit are not shown, but they would connect to each enable on the five registers. The cycle control unit is essentially a decoder, a counter, and a bit of combinational circuitry. It takes in the CPU bus signals and decides which enable to activate and whether or not the command is a read or a write from register command. The discrete circuit shown below refers to more specific information on the IDE interface implementation.

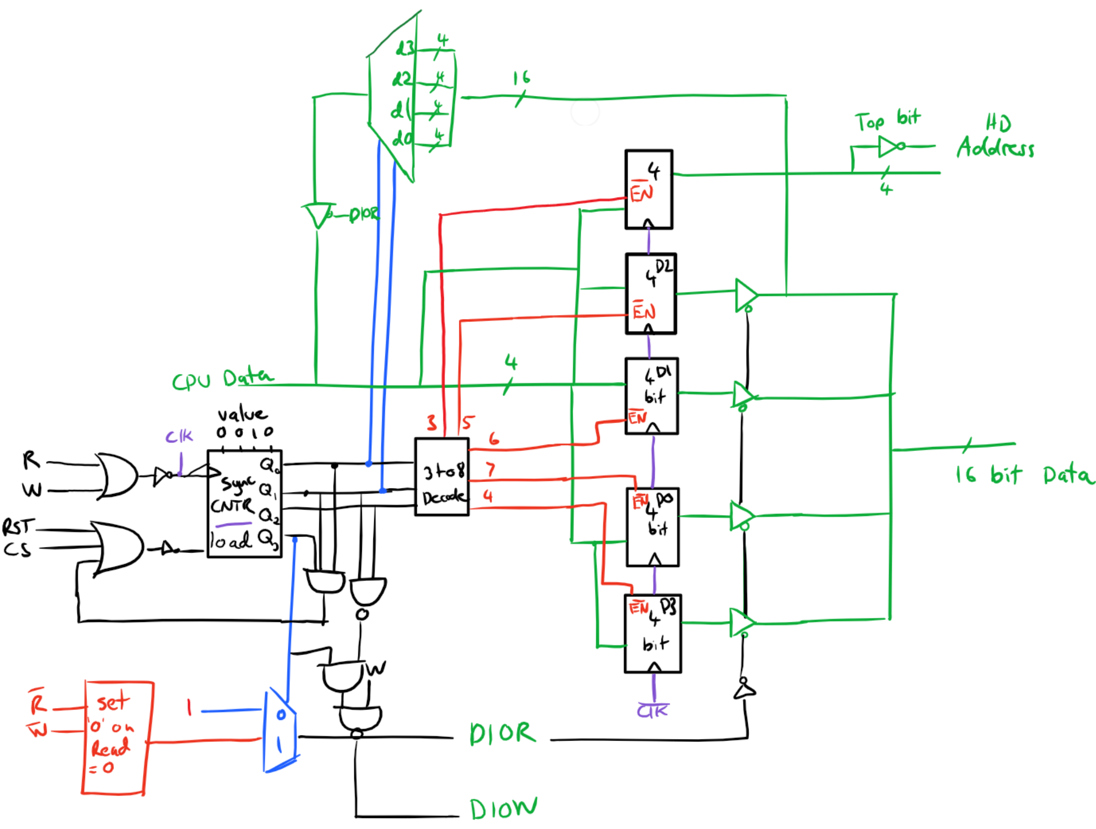


Figure 9.3: Low Level Discrete IDE Controller

9.4 IDE Controller and Hard Drive Communication

Communicating with the hard drive over the IDE cable has very specific requirements as per the ATA Standards. If the timing is incorrect, or the combination of signals sent do not match what the hard drive expects, an error occurs. The hard drive works by having a bank of communication registers that the user talks to over the IDE interface. This bank of registers serve as a barrier to the physical disk to prevent a computer from doing things it shouldn't. There are two types of registers: command block, and control block registers. In this design, control blocks will be ignored since they are never used by the IDE driver. The registers are described in the table below.

Command Block Registers						
Register #	Register Name	/CS1	/CS0	A2-A0	Usage	Register Details
$0	16 bit I/O data register	1	0	000	Used for all data block transfers from and to the IDE device	Read and write sector data words to

						and from this address
$1	Error register	1	0	001	Contains the status from the last command executed by the drive. It can be used to read Error code	It is only valid or only reads this register when an error is indicated "ERR=1" in the IDE Status register
$2	Sector Count	1	0	010	Number of sectors to transfer	This is hard coded as 1 in the IDE driver
$3	Sector Number	1	0	011	Contains sector address (0:7) of the LBA	Address lines used to specify where to read a sector from
$4	Cylinder Low	1	0	100	Sector bits (8:15) of the LBA	
$5	Cylinder High	1	0	101	Contains bits (16:23) of LBA	
$6	Drive/ Head			110	Sector address LBA (24:27)	
$7	Status Register	1	0	111	Contains either the drive status or the controller status. It Sends command to the drive when written to. And provides drive status when it is read.	

Table 9.2: IDE Command Block Register Description

There are 20 signal lines between the hard drive and the IDE interface. Four register address lines, (the CS1/CS0 are never on at the same time so they are just the NOT of each other), and 16 data lines. To perform a register read or write you must follow the following steps:

1. Set the 4 register address lines to the desired register to read/write from
2. If you are writing to this register, set the data lines to the desired value. If you are reading you can ignore this step
3. Pulse the DIOR signal for a read, or the DIOW signal for a write. While the DIOR signal is low, the data in the register shown on the address lines is forced onto the data lines.

These steps become a little more complicated when you add in the fact that you can only send 4 bits of data at a time as will be explained later.

9.5 IDE FPGA Implementation

The above design was implemented on an FPGA to test whether the idea would work in practice. It was completed and test benched, but due to logic level differences between the FPGA and the CPU (3.3V to 5V), and time constraints on making bi-directional level shifters, we were not able to test properly with the CPU. Since the design was relatively simple, we instead rushed to discrete to begin testing that. The entire VHDL code can be found on the Nibble Knowledge GitHub page. Below is the VHDL code for the cycle control unit. As seen in this code, the process counts when a falling edge of ‘R’ or ‘W’ from the CPU is detected. It also resets when it has finished a complete cycle, i.e. when the counter reaches ‘100’. When the VHDL code was written, the IDE interface was only using 8 data lines, so there were only 3 registers to load: the address bits, d_high, and d_low. The signal i_ready is used to tell the rest of the system that the registers have been loaded and the DIOR/DIOW signals can be pulsed. The last three lines of the if statement (prev_W <=W… wr_prev <= prev_W & prev_R) are used as a falling edge detector. The previous values of ‘R’ and ‘W’ from the CPU are compared to the current values to check if prev = ‘1’, and current = ‘0’. The cycle_counter value is used to determine which register enables are activated. Below is a segment the FPGA Implementation Code. This is only a snippet and the entire code can be found on the Nibble Knowledge GitHub page.

```
process(clk, reset, R, W)
begin
            if rising_edge(clk) and CS = '0' then
                  if prev_W = '1' and W = '0' then
                        cycle_counter <= cycle_counter + '1';
                        if cycle_counter = "100" then
                              cycle_counter <= "000";
                        end if;
                  elsif prev_R = '1' and R = '0' then
                        cycle_counter <= cycle_counter + '1';
                        if cycle_counter = "100" then
                              cycle_counter <= "000";
                              i_ready <= '0';
                        end if;
                  elsif prev_R = '0' and R = '1' then
                        if cycle_counter = "011" then
```

```
                        i_ready <= '1';
                    end if;
                end if;

                prev_W <= W;
                prev_R <= R;
                wr_prev <= prev_W & prev_R;
            end if;



        if reset = '1' then
            cycle_counter <= "000";
        end if;
```

9.6 IDE Discrete Implementation

The discrete version of the IDE interface follows the high level diagram that was shown in Figure 9.2 above. The only differences are that a few more parts were added since one discrete component cannot complete all the tasks required from the control unit box. The discrete circuit was built entirely from SN74 LS chips. Propagation delays were all measured and are well within required time for the circuit to work without race conditions. As described earlier, the circuit works by receiving 5 nibbles of data and then pulsing DIOR/DIOW depending on which is required.

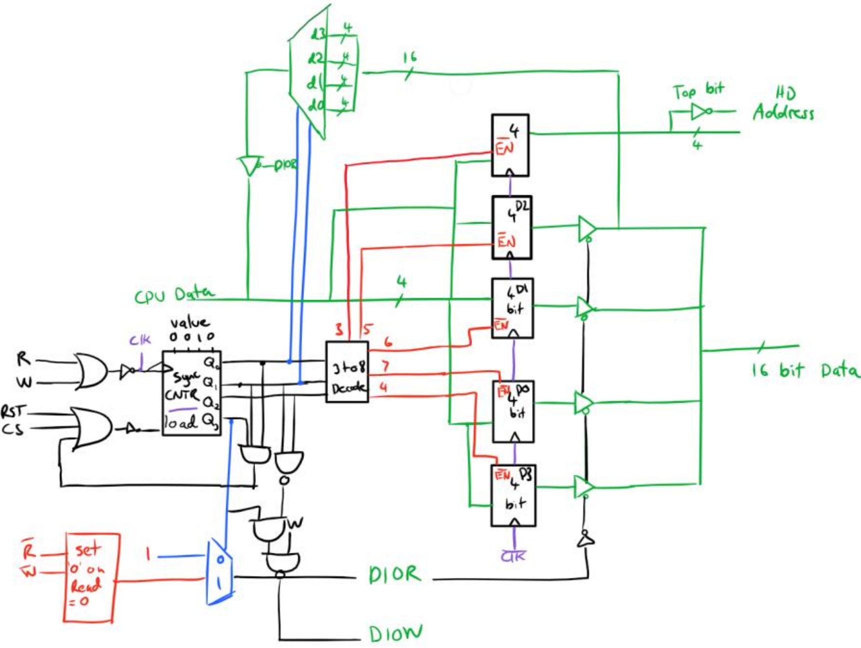


Figure 9.4: Low Level Discrete IDE Controller – Repeated from above

The communication signals that the CPU sends the interface are shown in Figure 9.5 below.

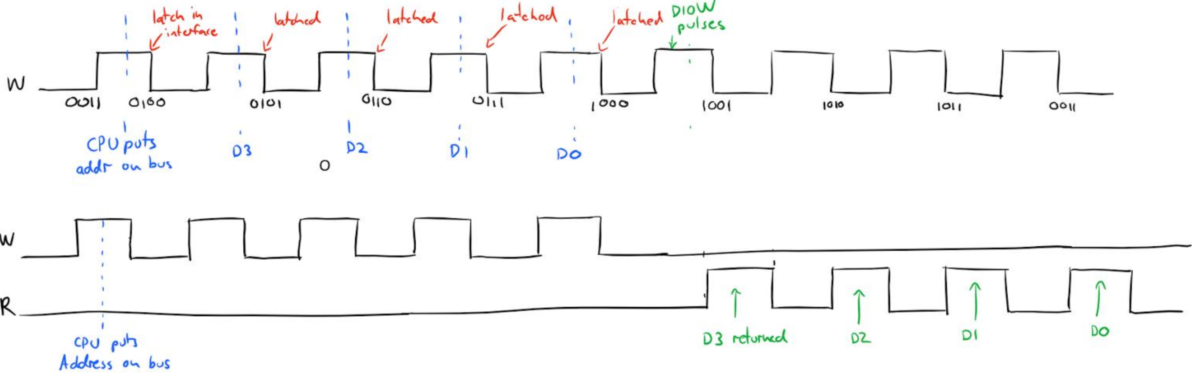


Figure 9.5: Signals of Communication between CPU and Interface

It is evident that the low-level design is very similar to the high level diagram. An important thing to note is that DIOR/DIOW must only pulse once on the read or write of a hard drive register. This is a requirement in the hard drive ATA specifications. The red box in the bottom left hand corner of the discrete circuit is what allows the DIOR to hold its value for the 4 cycles it must wait to output all four nibbles of data from the CPU. It essentially captures when the read signal turns on, and then leaves it output as a ‘0’ until write becomes a ‘1’. It is evident why this is required from the bottom signal in Figure 9.5. The data from the hard drive is received during the last four pulses of read. During this time DIOR must stay low.

Figure 9.5 describes the signals required from the CPU to correctly communicate with the hard drive. The top signal shows what must happen to write to a register, and the bottom two show the read from a register. It takes nine R/W pulses from the CPU in order to do a single read/write from the hard drive. This is very slow.

The entire circuit is clocked from the asynchronous signal NOT (R OR W). This allows the controller to be rate limited from the CPU, meaning it cannot continue into the next state without the CPU telling it. The CPU could send the address and D3 values to the controller, wait for a long time, and then resume without any consequences.

To read or write from/to a register, the CPU must follow these steps below, also refer to Figure 9.5. To read or write to an entire sector of the hard drive one must follow the steps below.

Some information before continuing:

- A sector is the smallest block that the hard drive can read or write to. In the case of the IDE 20-40GB hard drives used to test this peripheral, a sector is 512 bytes. (This is why there is a loop of 256 when reading/writing)
- The DIOR/DIOW single pulse is how the hard drive knows that 16 bits have been read, and it can move to the next data. If multiple pulses of DIOR happened per register read the hard drive would spit out the D3 from one 16-bit value, D2 from another, and so on.
- Register values in the section below are assuming that the top bit (CS1) is a part of the address. For example, in table 2 where it lists all hard drive registers, the data register with addresses of: CS1 = 1, DA2-0 = 0, is shown below as register 0x8. Every register used below has CS1 = 1 because these are the command registers.

Reading a Sector of Data

1) Using above steps for writing to a register, do the following:
   a) Write number of sectors to be read to Reg 0xA (recommended 1, max 8)
   b) Write bits 7-0 of LBA address to Reg 0xB (sector number)
   c) Write bits 15-8 of LBA address to Reg 0xC (Cylinder low)
   d) Write bits 23-16 of LBA address to Reg 0xD (Cylinder high)
   e) Write 0xE<bits 26-24> of LBA address to Reg 0xE (Drive/Head) (Ex. 0xE0 recommended)
2) Write the read command(0x20) to cmd Reg(0xF)
3) Read the status Reg (0xF, yes it’s the same reg as cmd) a few times, the more the better (5 is recommended by websites, ill test less when I get it on the CPU)
4) Repeat 256 times:
   a) Write 16 bits of data to data Reg(0x8)
5) Optional: Read status and check for error (mask = 0x1 of dlow)/ make sure DRQ is low (mask = 0x8 of dlow), maybe check BSY is low too (mask = 0x8 of dhigh).
6) Optional: to be sure you can read the sector count Reg(0xA), there should be 0 in there if the data was transferred correctly.

Writing a Sector of Data

1) Using above steps for writing to a register, do the following:
   a) Write number of sectors to be read to Reg 0xA (recommended 1, max 8)
   b) Write bits 7-0 of LBA address to Reg 0xB (sector number)
   c) Write bits 15-8 of LBA address to Reg 0xC (Cylinder low)
   d) Write bits 23-16 of LBA address to Reg 0xD (Cylinder high)
   e) Write 0xE<bits 26-24> of LBA address to Reg 0xE (Drive/Head) (Ex. 0xE0 recommended)
2) Write the write command (0x30) to cmd Reg(0xF)
3) Do not read status here
4) Repeat 256 times:
   a) Read 16 bits of data from Reg(0x8)
5) NOT OPTIONAL: Read status (0xF) a few times (5) to clear interrupt
6) Optional: steps 5, 6 of read can also be done.

These outlines are followed in the IDE driver that is written in macro assembly. This code can be found in the Nibble Knowledge GitHub Folder.

Wiring the IDE Controller

A few examples of wiring of the LS chips that were used to create the hard drive interface follow.

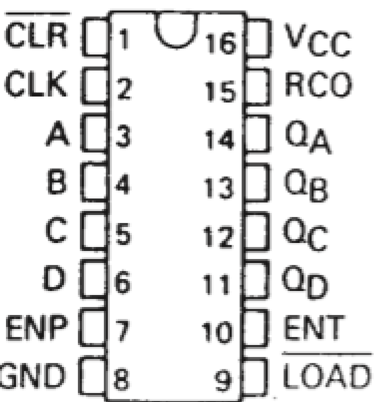


Figure 9.6: Top View of SN74LS163AN Counter Chip

This is the counter chip that is used as the main component of the cycle control unit. As with most 74 series chips, V_{cc} is in the top right, and ground is in the bottom left. These are wired directly into the high and low rails on a bread board. This chip has some of the low enables that were originally planned to be active high. In the next section we talk about how this affected the design. A, B, C, D are the inputs if you want to load a starting value into the counter. Since this design requires a reset value of 3, these values were used along with the active low signal LOAD to reset to the required value. The Q values directly across the chip are the counter output values that are used in combinational circuitry and fed into a decoder to enable each register one by one. When the counter reaches the max value of 1011, the LOAD signal, is pulsed low so it reset the chip.

9.7. Testing the IDE Controller

The FPGA version was strictly in test bench. The discrete version, however, was thoroughly testing in many different ways.

The components were tested while being built. Each time a new chip was added, input signals were provided to verify performance. During this phase, the important discoveries were:

1. Many chips had active low enables instead of high. It usually worked out okay because they all had active low so NOT gates were not needed.
2. Unused inputs must be grounded to avoid noise and unintended outputs.

Once built, the discrete circuit was tested using an Arduino since this is 5V logic. These tests were successful and allowed editing of sectors. A problem occurred later due to the maximum achievable speed of the Arduino being 165Hz.

Validation tests included:

1. Creating a text file on a separate computer with blank text, then using the IDE interface to write text to it. We could then check the results on the other computer. This was successful.
2. Using the Arduino as a controller, reading and writing to/from the same sector with the IDE interface. This was successful.
3. Using the Arduino with the bus circuit, to ensure design could read from the hard drive; and send the read values to the sound controller in-between reads. This was successful.

Tests were then conducted using Nibble Knowledge CPU. This was very inconsistent, mainly due to the high speeds. We were able to read a program from sector 0, and run it at 5MHz for a while,

but eventually it stopped working. We did this by using a separate computer to write the binary file onto sector 0 of the hard drive. The CPU would then use the interface to read the program into location 50000 of the RAM. The CPU would jump to 50000 and run a keyboard and sound program, which played a different sound based on the key pressed. It was found that the voltage supply required very high precision for it to function properly; as a result the next time the computer was set up, we could not get it to run. Due to this we removed the hard drive from the final demo.

Note: There is extensive amount of code that has been developed to design the Nibble Knowledge Computer. All of the code for different aspects of the IDE Controller is located in an online, web-based Git repository hosting service. The Code is accessible through the link below:

https://github.com/Nibble-Knowledge

9.8 IDE Controller Specifications

Component	**Specifications**
Clock Frequency	5 MHz CPU Clock
Controller Voltage Logic	5.00 V
Input Lines from CPU	4 Data lines
Input Lines from CPU	3 Status lines (W,R,PC)
Output Lines to Hard Drive	16 Data lines
Output Lines to Hard Drive	5 Address lines
Output Lines to Hard Drive	DIOR and DIOW
Connector	(4-pin) AMP1-480424-0
Contacts	(piece) AMP 60619-4
Contacts	(strip) AMP 61117-4
ATA Standard	ANSI X3.221-199x

Table 9.3: IDE Controller Specifications

10. PS/2 Keyboard Controller

10.1 Introduction

The PS/2 Keyboard Controller is the interface between a PS/2 Keyboard and the CPU. Making it one of the crucial elements in ensuring the words or numbers that are typed on the keyboard end up on the monitor. The main function of the controller is to evaluate the signals transmitted from both the PS/2 Keyboard and the CPU. Once the controller is done evaluating these signals, it will then transmit signals back to either the keyboard or the CPU based on the information the controller initially received.

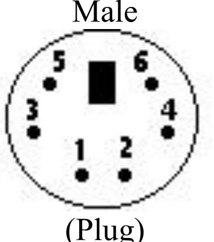
The purpose of the PS/2 keyboard controller is to facilitate communication between the keyboard and the CPU. The controller converts electrical signals transmitted by the keyboard (when a button is pressed on the keyboard) into valuable information that is read by the CPU when it is convenient for it. In order to allow the CPU to finish its processes with other peripherals, the data from the keyboard is stored within the controller until the CPU is ready to read it.

10.2 PS/2 Port

The PS/2 port is the location where the keyboard plugs into the controller. All communication is transmitted through this port. The protocol between the PS/2 keyboard and the PS/2 Keyboard Controller is synchronous, binary and serial based. There are two important signals sent through the PS/2 port; the PS/2 keyboard generated clock and data signals.

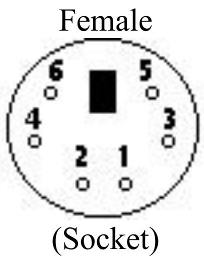
The PS/2 Port pin-outs are as follows:

Male



(Plug)

Female



(Socket)

6-pin Mini-DIN (PS/2):
1 - Data
2 - Not Implemented
3 - Ground
4 - Vcc (+5V)
5 - Clock
6 - Not Implemented

Figure 10.1: PS/2 Port Plug and Pin Assignment

10.3 Keyboard to Controller Transmission

The data from the PS/2 keyboard to the PS/2 keyboard controller is sent serially in 11-bit frames. The order of data transmission is as follows:

1. A start bit (which is always zero).

2. Eight data bits starting with LSB first are shifted out one-by-one.
    a. These are the bits of interest that will be transmitted from the PS/2 Keyboard Controller to the CPU.
3. An odd parity bit (used for error detection).
    a. If the number of 1's in the data byte is even the parity is set to 1, but if the number 1's is odd the parity is set to 0.
4. A stop bit (which is always one).

The PS/2 data transmission is synchronized to the PS/2 Keyboard clock signal, using a frequency of 12.3 kHz. The keyboard writes a bit on the PS/2 Keyboard data line on the rising edge of the clock. On the falling edge of the clock the bit is shifted into the PS/2 Keyboard controller's shift register.

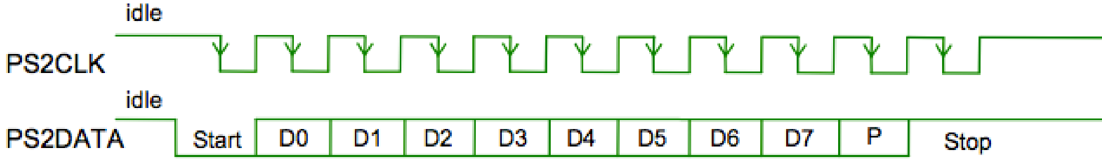


Figure 10.2: PS/2 Data Transmission Synchronized with Clock Signal

Although not implemented, the CPU can transmit to the PS2 keyboard through the controller for the purpose of turning LED's on, transmitting an acknowledgement bit, etc. This process initiates when the controller drives the clock line low for a period of 100 microseconds, disabling any data being sent from the keyboard. The controller then drives the data line low, which tells the keyboard the controller is ready to transmit data. The controller then releases the clock line and after a short delay the keyboard starts clocking. The keyboard reads each bit on the rising edge and the controller sends each bit on the falling edge. This continues until the stop bit is sent and the controller releases the data line. The keyboard then sends an acknowledgement bit in the form of driving the data line low.

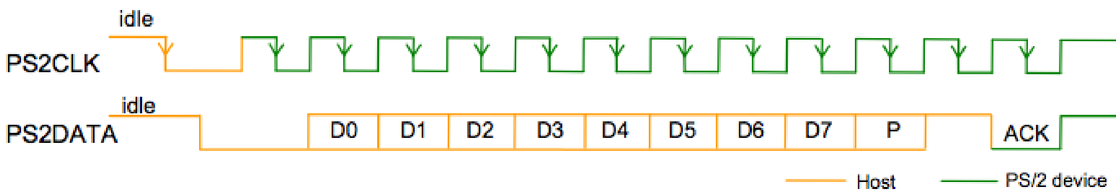


Figure 10.3: CPU Data Transmission to PS/2 Controller

10.4 Scan Codes

The keyboard sends a scan code to the controller whenever a key is being pressed, released or held down. The data codes for the various keys on the keyboard are shown in Figure 10.4 below. Two types of scan codes are sent to the controller: “make codes” and “break codes”. The make code is sent when a key is pressed or held down; and a break code is sent after a key is released. We have chosen to ignore the break codes in this design.

Each key has its own specific make and break codes. The make code is a single byte, unless an extended key is pressed (i.e. Ctrl, left arrow, etc.). If an extended key is pressed the make code is preceded by an “E0” byte. A break code is the keys make code preceded by either a “F0” for normal keys or an “E0” and an “F0” for extended keys. In this design, instead of sending a make code, a break code and another make code for every key pressed to the CPU, the peripheral only sends the initial make code. If a key is pressed and held down that key becomes typematic. The keyboard will continue to send that keys make code until another key is pressed or the key is released.

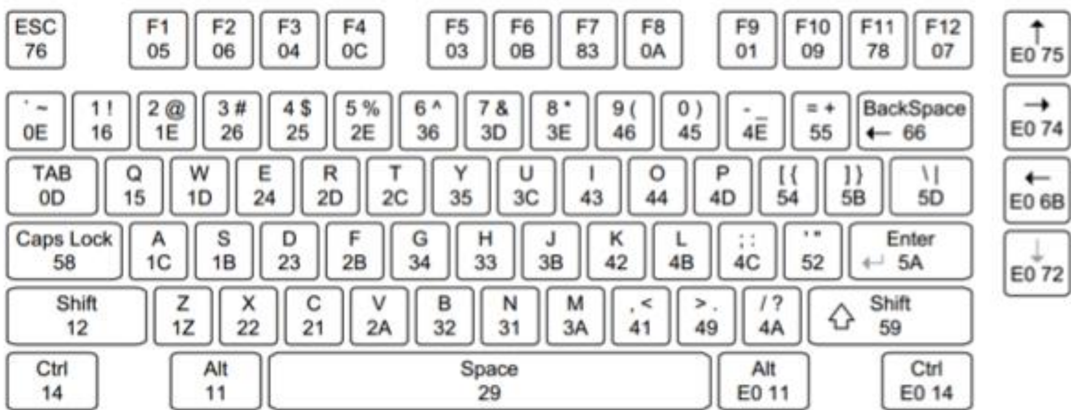


Figure 10.4: PS/2 Keyboard Scan Codes

The concept of scan codes is best explained through an example:

1. Letter ‘S’ key is pressed. The scan code for this key is 1B in hexadecimal or 00011011 in binary. The odd parity bit for this scan code is ‘1’.
2. The make code for this transmission is then 0 1101 1000 1 1 (start bit, data byte with LSB first, parity bit, and stop bit).
3. Once the ‘S’ key is released a break code is sent. This break code is essentially the make code with an F0 byte added in front of it. Therefore, the break code sent is 0 0000 1111 1

0, 0 0011 1000 0 1 (the stop bit for the F0 byte is 0 since the transmission is not complete). In this educational kit we will completely ignore the break code and focus solely on the make code. Once the make code has been transmitted into the shift register the shift register will discontinue to shift freely and ignore any new incoming data until the make code has been transmitted to the CPU.

10.5 PS/2 FPGA Implementation

The FPGA implementation of the peripheral serves as an intermediate step in the development of the discrete PS/2 keyboard controller. This implementation allowed a behavioral approach in designing the peripheral. Additionally, it simplified the task of creating a discrete circuit by transforming the VHDL code into discrete logic chips. These are only snippets and the entire code is attached in the Appendix, section A.2 PS/2 Controller

Example 1: VHDL code to discrete logic chips conversion

VHDL code: if (cpu_read = '0' and parity_check = '1' and low_nibble_out = '1' and end_of_trans <= '0')

Logic chips: A 4 IN – 1 OUT AND gate

- Inputs: NOT cpu_read, parity_check, low_nibble_out and NOT end_of_trans
- Output: the result of the following 4 signals AND gated together

Example 2: VHDL code to discrete logic chips conversion - Multiplexer Selector

VHDL code:

```
if (no_data = '1') then
        out_nibble <= "ZZZZ";
else
        if (s = '1') then
                out_nibble <= high_nibble;
        else
                out_nibble <= low_nibble;
        end if;
end if;
```

Logic chips: 8x4 Multiplexer chip

- Inputs: output of an AND gate (which determines if the output of the Mux should be the high or low nibble due to the state of signals coming from the CPU and PS/2 Keyboard)

- Output: if the input to the select pin was high the mux will output the high nibble otherwise it will output the low nibble

10.6 PS/2 Discrete Implementation

Detecting End of Transmission

When a key is pressed on the PS/2 keyboard the data is shifted into the 11 shift registers located within the PS/2 keyboard controller. Once the 11-bits have been transmitted the shift registers stop shifting through the use of a counter and some additional combinational logic.

8 x 4 Multiplexer

The CPU is informed that data is ready to be read once the end of transmission has been detected. This is accomplished by setting the ready bit to logic high. Bits (8 to 1) of the shift registers are passed into an 8 x 4 multiplexer. The select for the mux is controlled with the CPU read, parity and chip select lines along with some additional sequential and combinational logic. Initially, when end of transmission has been determined the select is set to '1' and the multiplexer will output the high nibble (bits 8 to 5). The CPU will then read the data from the controller and set its read line to logic low, which sets the multiplexer select to '0'. The output of the multiplexer will now be the low nibble (bits 4 to 1). The CPU reads the data from the controller and sets its read line to logic low once the read has completed. Now that the high and low nibbles have been successfully sent from the controller to the CPU, the controller will set the ready line back to logic low until another key has been pressed on the keyboard.

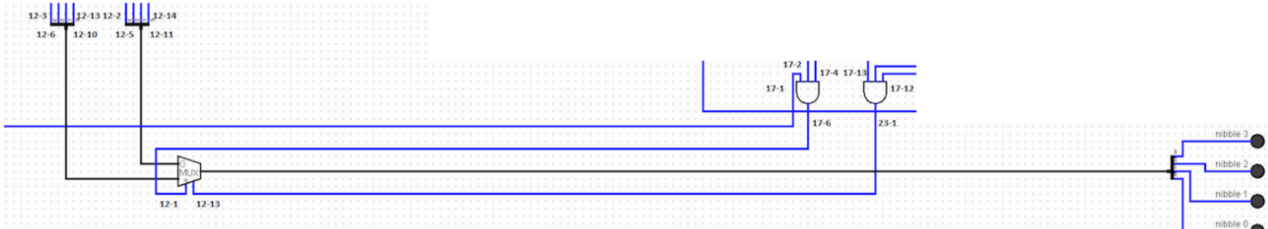


Figure 10.5: The 8 x 4 Multiplexer

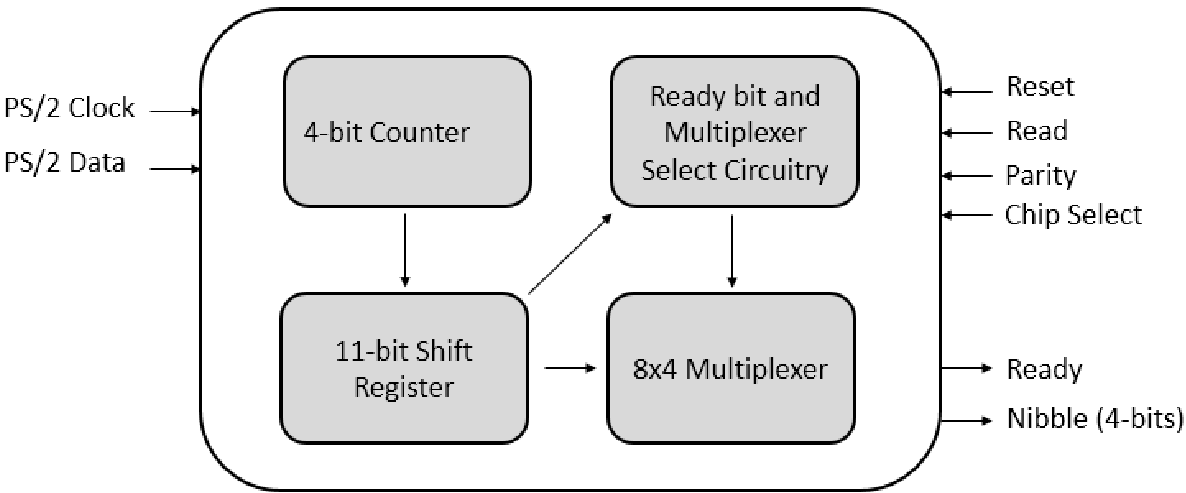


Figure 10.6: High Level Block Diagram of PS/2 Controller

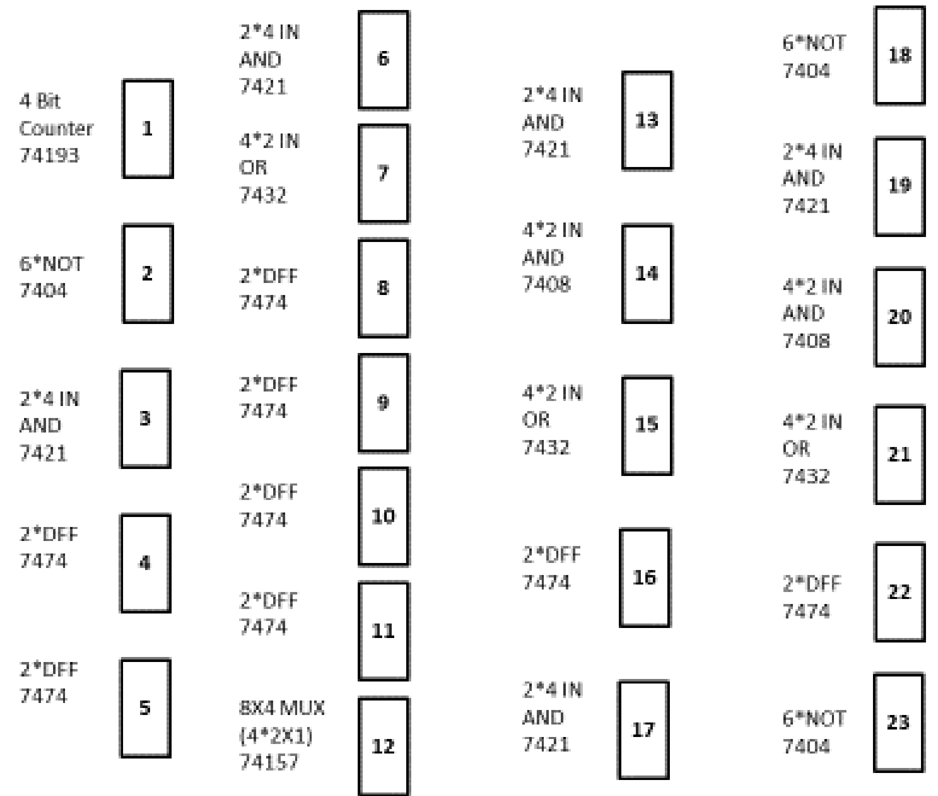


Figure 10.7: PS/2 Controller Breadboard Circuit Layout

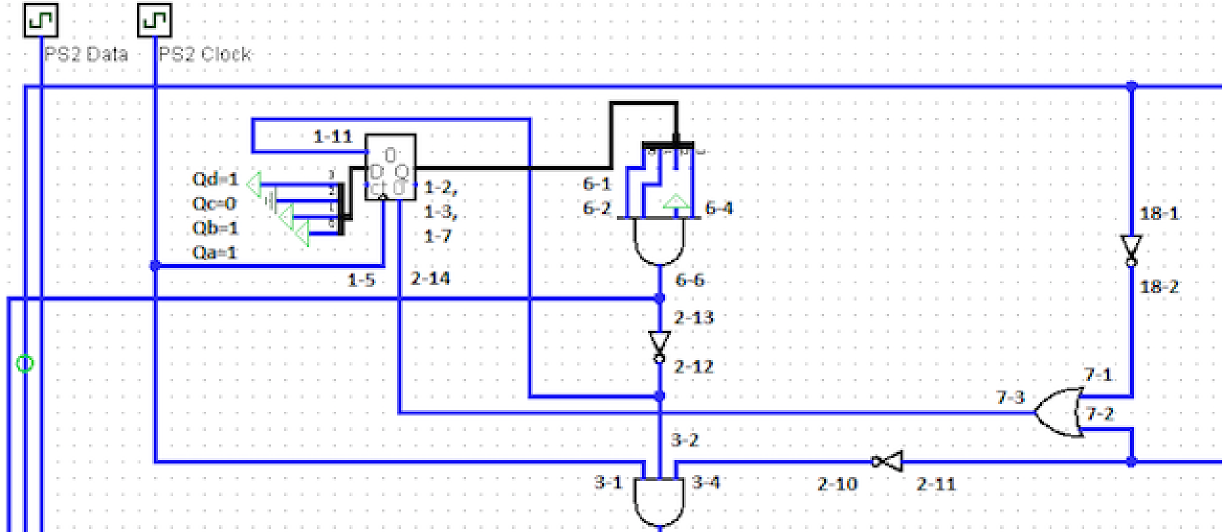


Figure 10.8: PS/2 Controller Counter Circuit

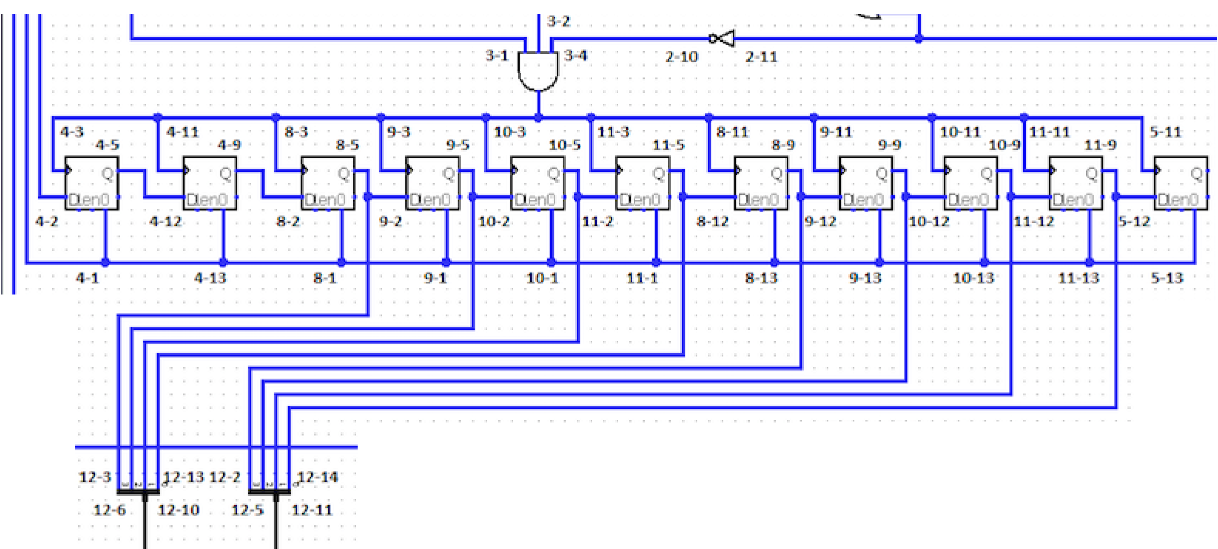


Figure 10.9: PS/2 Controller Shift Register

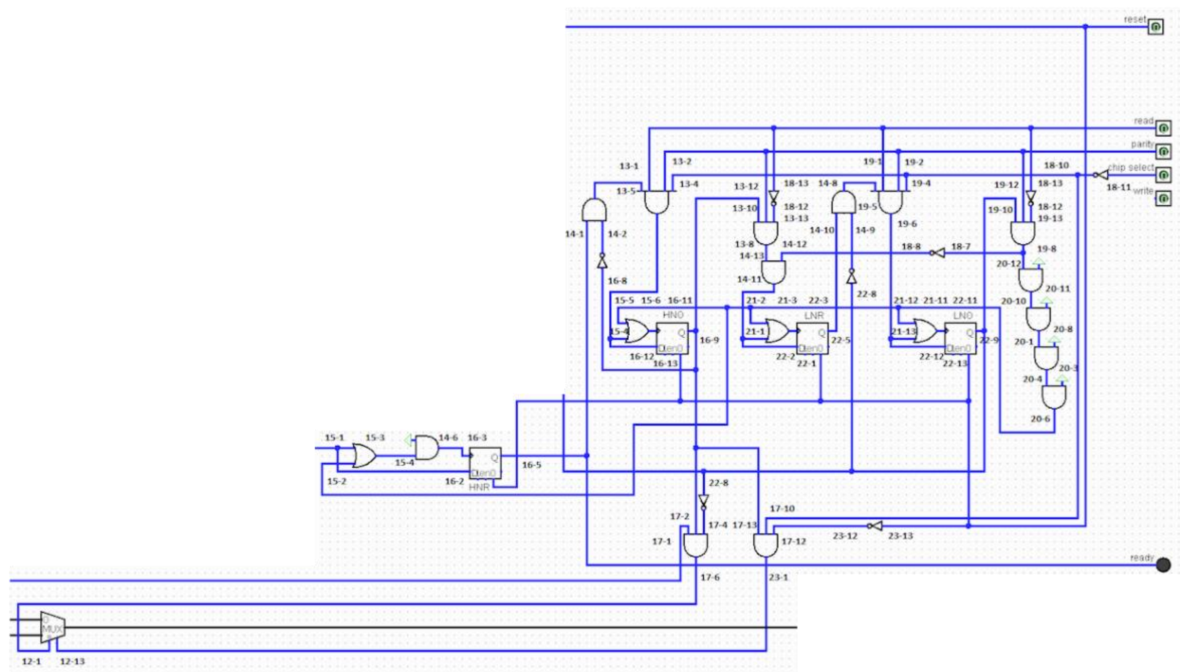


Figure 10.10: PS/2 Controller Ready Bit and Multiplexer Select Circuit

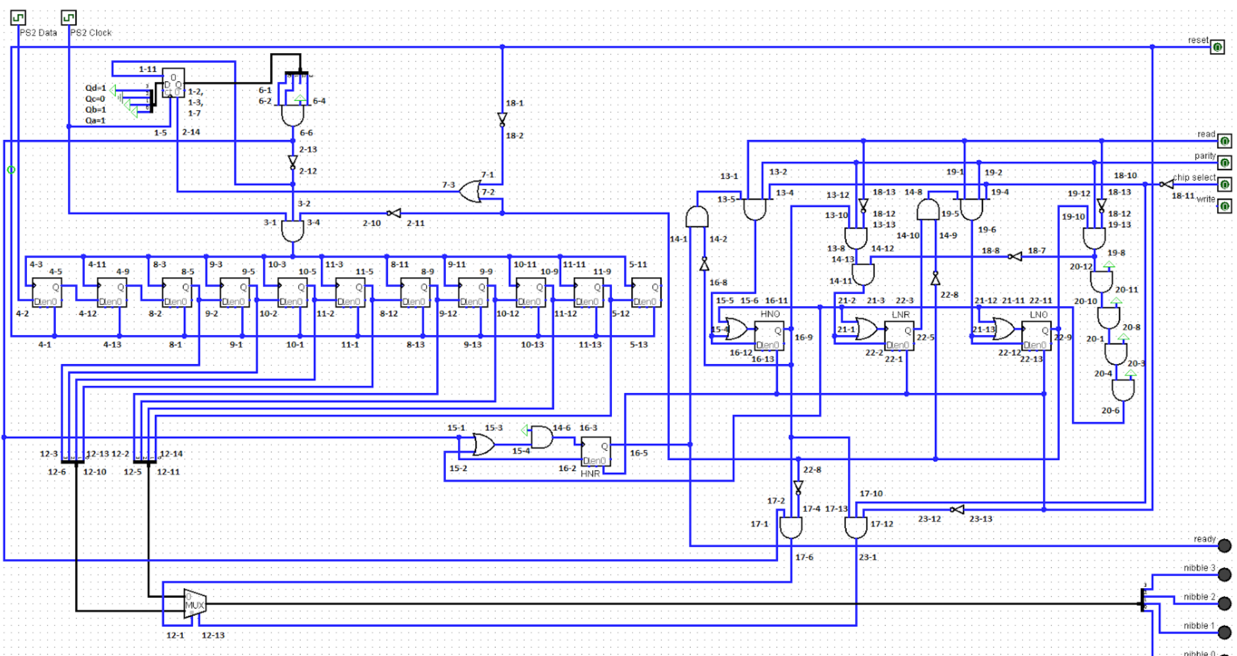


Figure 10.11: PS/2 Controller Entire Low Level Circuit

Circuit Wiring Protocol

Example: Explaining the number-number convention

Chip 2 – Pin 10 is connected to Chip 3 – Pin 4 (i.e. Output of the NOT gate is connected to one of the inputs of the 3 IN AND gate.

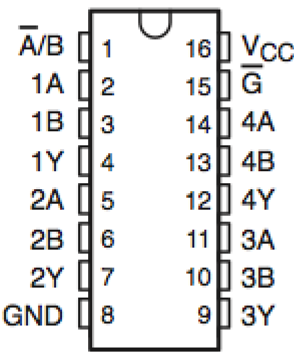


Figure 10.12: PS/2 Controller SN74LS 8 x 4 Multiplexer

Pin 1: Chip 12 - Pin 1 (Mux selector - If high will output high nibble to bus data line and if low will output low nibble to bus data line)

Pin 2: Chip 8 - Pin 9 (Shift Register - Bit 4) Low Nibble Bit 3

Pin 3: Chip 8 - Pin 5 (Shift Register - Bit 8) High Nibble Bit 3

Pin 4: Bus Data line bit 3

Pin 5: Chip 9 - Pin 9 (Shift Register - Bit 3) Low Nibble Bit 2

Pin 6: Chip 9 - Pin 5 (Shift Register - Bit 7) High Nibble Bit 2

Pin 7: Bus Data line bit 2

Pin 8: Wired to ground

Pin 9: Bus Data line bit 1

Pin 10: Chip 10 - Pin 5 (Shift Register - Bit 6) High Nibble Bit 1

Pin 11: Chip 10 - Pin 9 (Shift Register - Bit 2) Low Nibble Bit 1

Pin 12: Bus Data line bit 0

Pin 13: Chip 11 - Pin 5 (Shift Register - Bit 5) High Nibble Bit 0

Pin 14: Chip 11 - Pin 9 (Shift Register - Bit 1) Low Nibble Bit 0

Pin 15: Chip 12 - Pin 13 (Mux low enabled reset)

Pin 16: Wired to 5 V

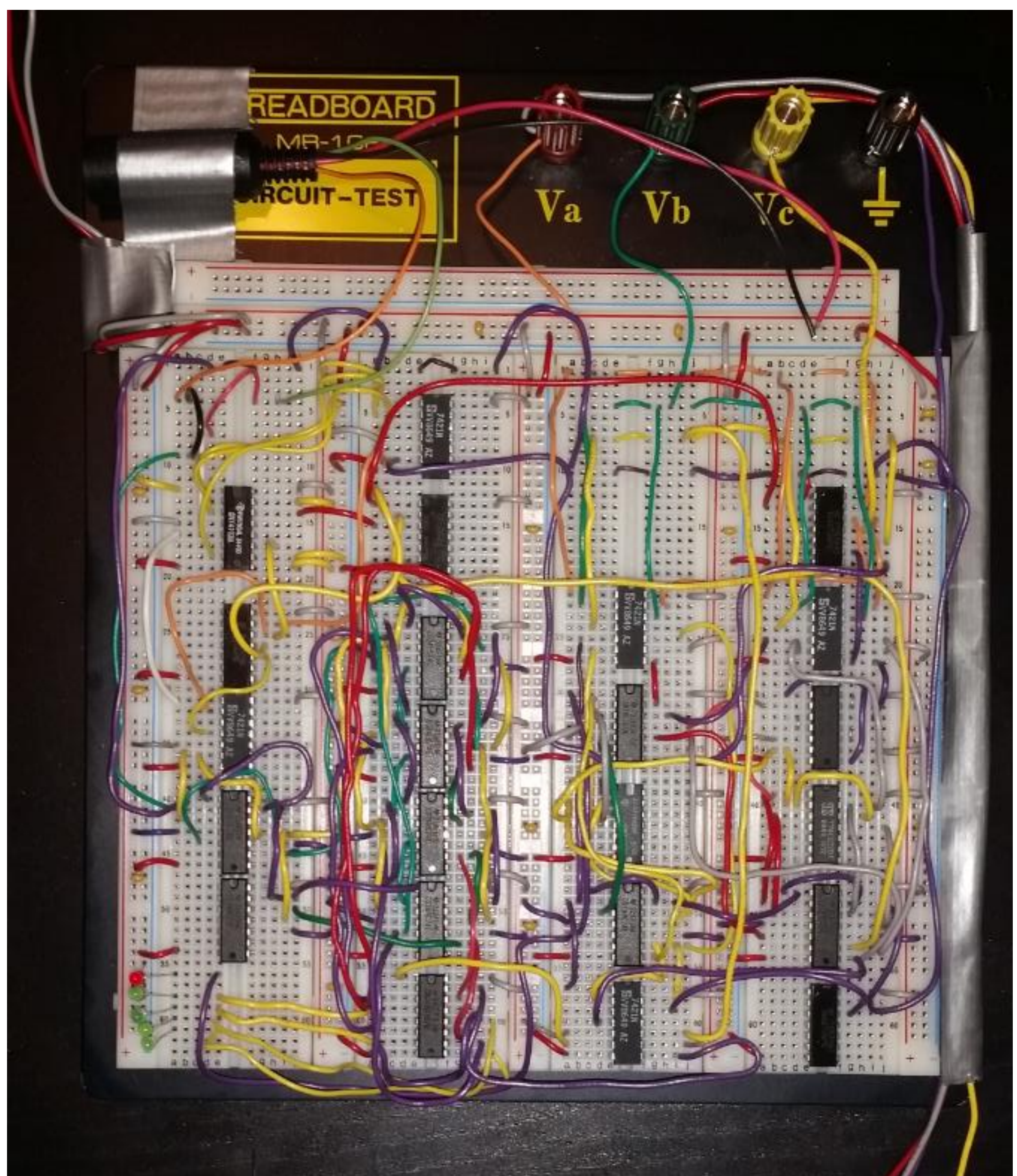


Figure 10.13: Fully wired PS/2 Keyboard Controller

10.7. The Final Product

The PS/2 keyboard controller is able to convert serial input from the PS/2 keyboard (i.e. the 11 bit make code from a key that is pressed on the keyboard) to two four bit nibble outputs, which are transmitted at the CPU's convenience. The final PS/2 keyboard can also transmit to the CPU through the controller for the purpose of turning LED's. Due to the simplicity of our computer

design it was decided we omit any CPU to keyboard transmission (i.e. illuminating the CAPS or NUM lock LED's).

Controller to Keyboard Transmission

Although not relevant to the implementation of this Keyboard Controller, the controller can be a bidirectional device. It was in the best interest of hardware simplicity to avoid transmitting from the CPU to the keyboard because this functionality is not required for the purpose of this educational kit and it would add unneeded complexity to the design. The CPU can transmit to the PS2 keyboard through the controller for the purpose of turning LED's on, transmitting an acknowledgement bit, etc. This process initiates when the controller drives the clock line low for a period of 100 microseconds, disabling any data being sent from the keyboard. The controller then drives the data line low, which tells the keyboard the controller is ready to transmit data. The controller then releases the clock line and after a short delay the keyboard starts clocking. The keyboard reads each bit on the rising edge and the controller sends each bit on the falling edge. This continues until the stop bit is sent and the controller releases the data line. The keyboard then sends an acknowledgement bit in the form of driving the data line low.

The PS/2 keyboard controller ignores the start, parity and end bits of the make code and only relays the eight data bits which are unique to each key on the keyboard. We also chose to ignore the parity bit sent from the keyboard to the controller and deemed it as an unnecessary addition to our design.

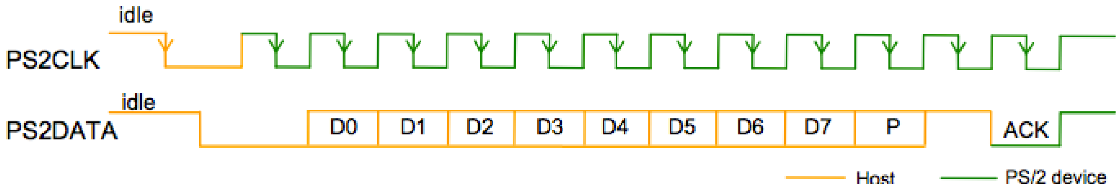


Figure 10.14: CPU Data Transmission to PS/2 Controller

Parity Bit

A parity bit check that includes data sent from the keyboard to the controller is a handy tool to detect any errors during transmission. It was determined to exclude this functionality in the kit since transmission errors between the keyboard and controller are very rare and the extra logic to

include the parity check will take up needed real estate on our breadboards. The parity bit is the second last bit to be sent to the controller followed by the stop bit. A parity bit will also be computed within the controller and the two bits will then be compared to prove no data was corrupted during transmission. The parity bit within the controller is computed with the use of an XNOR gate. All data bit are connected together at the input of the gate and the parity bit is given at the output.

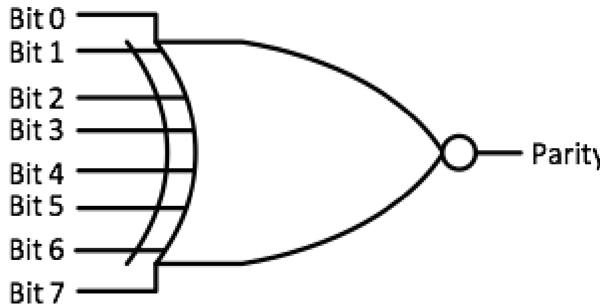


Figure 10.15: Parity Bit XOR Gate

The controller is designed this way to achieve the simplest circuit possible while upholding the functionality needed to transmit data from the keyboard to the CPU. Therefore, it was decided to leave out features like CPU to Keyboard data transfer (i.e. to illuminate LED's for CAPS or NUM lock) and an internal parity check for further protection against data corruption.

10.8 PS/2 Controller Specifications

Component	Specifications
Controller Voltage Logic	5.00 V
Clock Frequency	12.3 kHz
CMOS/TTL Chips V_{cc}	5.00 V
CMOS/TTL Chips VIH	2 – 3.15 V
CMOS/TTL Chips VIL	0.8 – 1.53 V

Table 10.1: PS/2 Keyboard Controller Specifications

11. Serial Communication RS232 Controller

11.1 Introduction

Serial communication, also known as RS232, is a standard method of communicating between computers and peripherals. The devices are connected with a serial cable, most commonly with DB9 connectors. The standard bit transfer rate is 9600 baud (where a baud is approximately one bit per second), but it can be anywhere from 110 baud to almost a mega-baud.

Although the standards specify how information is to be transmitted across the wire, it leaves it up to the user to verify the integrity of the data. Thus, features such as handshaking and parity checks depend on how the hardware and software were implemented, and there is no one standard.

Data Equipment

Serial communication is used between two devices, one which is called the data terminal equipment (DTE) and the other being the data circuit-terminating equipment (DCE). Normally, the DTE is the main computer, while the DCE is the peripheral to which the DTE sends commands, such as a modem. In Nibble Knowledge serial controller, the DTE is considered the PC or other device connected to the serial peripheral, and the DCE is the peripheral itself, which is connected to the CPU.

DB9 Connector

A DB9 connector is a 9-pin jack to which a serial cable connects. Not all pins need to be used, but at least three of those have to be connected: one for transmitting data, one for receiving, and one grounded. The full pinout and listing of the pins used by the peripheral are described in Figure 11.1 and Table 11.1 below.

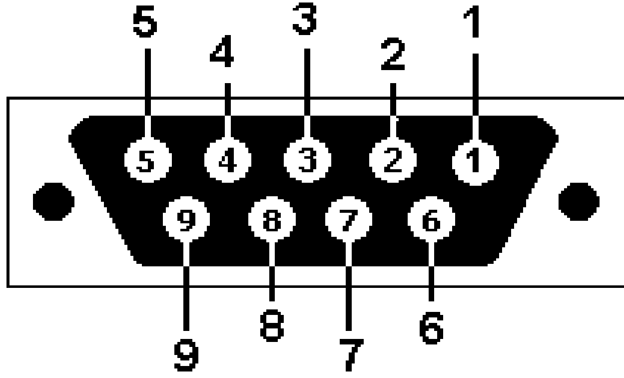


Figure 11.1: DB9 Connector – Female. Male Connector flipped horizontally

Pin	Name	Direction	Description	Used in Peripheral?
1	Data Carrier Detector (DCD)	DTE←DCE	Is the DCE (e.g. a modem) connected to the remote host	NO
2	Received Data (RD)	DTE←DCE	Data sent from peripheral to PC	YES
3	Transmitted Data (TD)	DTE→DCE	Data sent from PC to peripheral	YES
4	Data Terminal Ready (DTR)	DTE→DCE	DTE is connected and ready to transmit/receive data	NO
5	Ground (GND)			YES
6	Data Set Ready (DSR)	DTE←DCE	DCE is connected and ready to transmit/receive data	NO
7	Request to Send (RTS)	DTE→DCE	DTE wants to send data to the DCE	YES
8	Clear to Send (CTS)	DTE←DCE	DCE is ready to accept data from DTE	NO
9	Ring Indicator (RI)	DTE←DCE	DCE (e.g. modem) has detected a ring on the phone line	NO

Table 11.1: DB9 Connector Pinout in Serial Controller

Data Transmission Protocol

Data is sent on the RD or TD lines, depending on the direction, bit by bit at the specified baud rate (peripheral uses 9600 baud). The first signal sent is a transition from low (0V) to high (5V), indicating to the other side that a message is being sent. This is followed by 8 bits of data, least significant bit (LSB) first, where a '1' is represented as 0V and a '0' as 5V. Finally, a low signal is sent to indicate the end of the message. Afterwards, another message may be sent, starting with the low → high transition. Figure 11.2 visually shows the signal being sent on the wire.

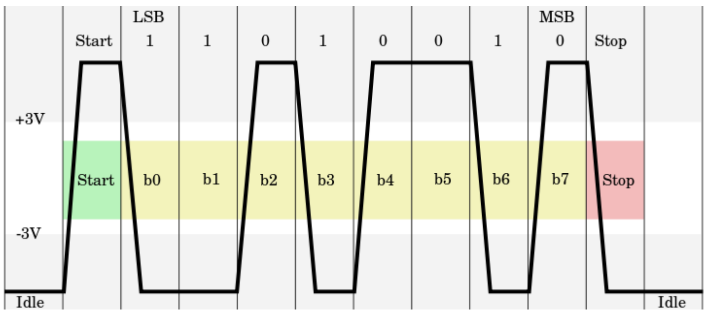


Figure 11.2: Serial Controller Sample Data Transmission

Communicating with the CPU

When reading from the peripheral, the CPU needs to raise the read signal high. If the peripheral does not raise the ready signal high, it means that no new data arrived from the PC. Otherwise, the peripheral raises the ready signal high and puts the four most significant bits (MSB) of the data on the bus, until the CPU lowers the read signal. To get the four LSBs, the CPU must once again raise the read signal.

When writing to the peripheral, the CPU needs to raise the write signal high. If the peripheral does not raise the ready signal high, the peripheral is busy transferring older data to the PC and is not currently ready to receive any new data. Otherwise, the CPU needs to place the four MSBs of the byte on the bus, and keep them until lowering the write signal. To write the four LSBs, the CPU needs to once again raise the write signal. Only after the CPU transmits all eight bits to the peripheral is when the byte will be actually transmitted to the PC.

11.2 Serial Controller High Level Design

The peripheral can be considered as made out of four components: PC to peripheral, peripheral to CPU, CPU to peripheral, and peripheral to PC. The former two components are separated by a RAM cell, used to buffer incoming data, and the latter two separated by an 8-bit buffer for outgoing data. Figure 11.3 illustrates this design.

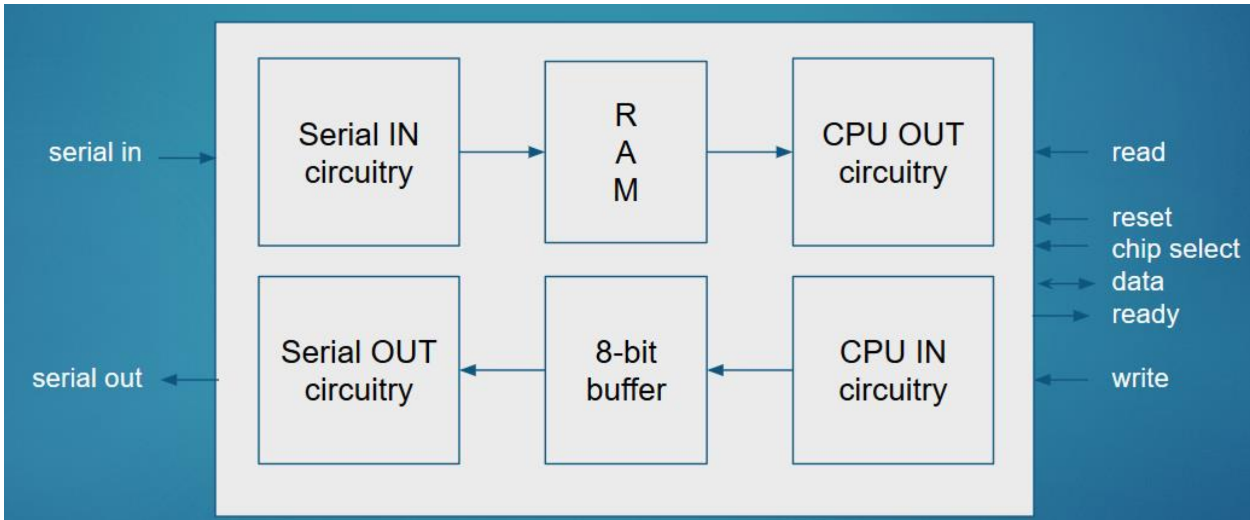


Figure 11.3: Serial Controller High Level Design

Connections on the left are connected to the DB9 connector, specifically pin 3 for serial in and pin 2 for serial out, and signals on the right are connected to the CPU.

The RAM cell is required when receiving data from the PC because, in the current implementation, there is no way to tell the PC when to send data. As such, the peripheral needs to buffer every incoming byte or risk dropping data. Although the CPU is faster than incoming data, the CPU may not be constantly polling the peripheral for more data. A RAM cell is not required for outgoing data, instead being replaced with a simple 8-bit buffer, because the peripheral can indicate to the CPU when it's allowed to send data.

11.3 Serial Controller FPGA Implementation

The top-level design described above has been implemented and tested on an FPGA. Like the top-level design, it is composed of a top-level implementation and contains the four components described above. Due to the limitations of the FPGA, an external RAM chip must be wired up to the FPGA. Furthermore, since the serial signal operates at 5V and the FPGA operates at 3.3V, a couple CMOS chips were used as voltage translators.

The code used for a Papilio One FPGA is available on the Nibble Knowledge GitHub page. In Figure 11.4 below, an example wiring diagram is provided with Papilio One FPGA, IDT71256SA 32KB RAM cell, and CD4007 CMOS chips.

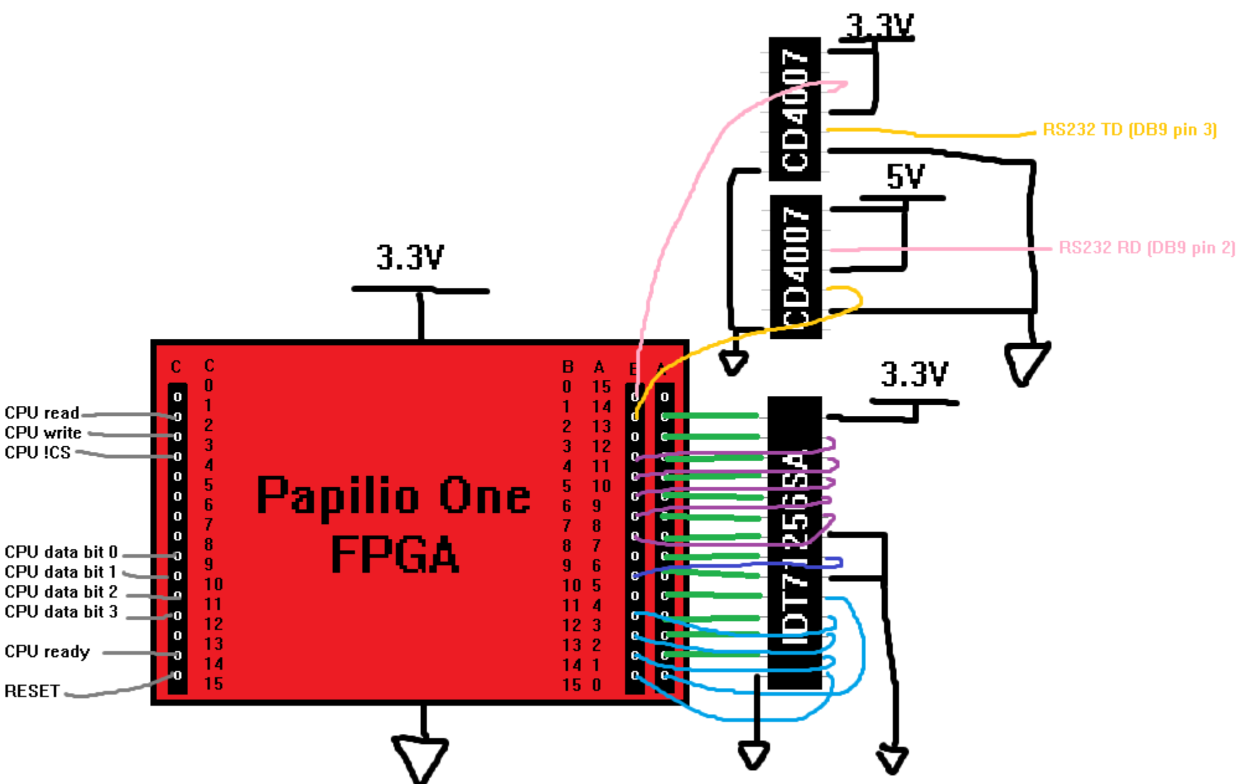


Figure 11.4: Papilio One FPGA Wiring Diagram

11.4 Serial Controller Discrete Implementation

Quick Wiring Guidelines

The chips illustrated above are referred to as dual in-line packages (DIP). DIP chips are primarily used on breadboards and are inserted in the middle, above the center groove. Figure 11.5 below shows how and how not to insert DIPs.

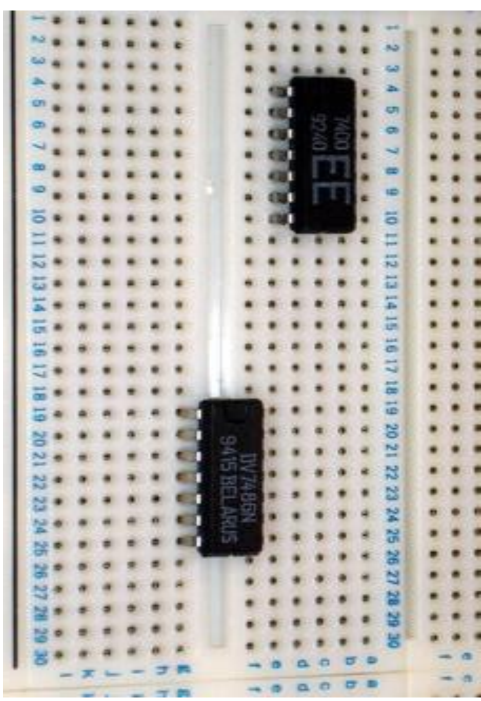


Figure 11.5: DIP Chips on Breadboard

Insert the chips as on the bottom. NEVER insert like the on the top. Note the engraving on the side of the Chip, indicating the top side of it.

DIP chips usually have an engraved semicircle, indicating the top side of the chip. All chips should be facing the same way on the breadboard.

Pin numbering begins on the top-left of the chip and continues counterclockwise, reaching the bottom-left corner, continuing on the bottom-right, and ending on the top-right. Looking on the chips in Figure 11.5, there are 14 pins: 1 through 7 on the left side, counting downwards, and 8 through 14 on the right side, counting upwards.

To connect one pin of a chip to another, insert a wire to a breadboard pin on the same row as the chip pin. Normally, breadboards have dedicated rails on the sides for ground and power. Chips requiring ground and power (which should be all of them) need to be wired up to the corresponding rail. For example, in Figure 11.5, the top CD4007 chip has 6 connections, and the pins should be connected as follows: pins 7 (bottom left) and 9 connected to the ground rail; pins 11 and 14 (top

right) to the power rail, ensuring the voltage is at 3.3V; pin 10 connected to the DB9 connector; and pin 12 connected to the B0 pin on the FPGA.

To simplify the implementation of the discrete serial peripheral, the VHDL code used for programming the FPGA has been converted into a circuit. Since VHDL is used to describe a real digital circuit, translation is relatively simple. All finite state machines (FSM) were converted to a mux-DFF-decoder combination: mux (multiplexer) to determine the next state, DFF (D flip-flop) to hold the current state, and a decoder to convert a state number into an active signal.

For simplicity, the discrete implementation is split up into the top-level design and the four components. Below, figure 11.6 described the top-level discrete implementation, while figure 11.7 describes the components.

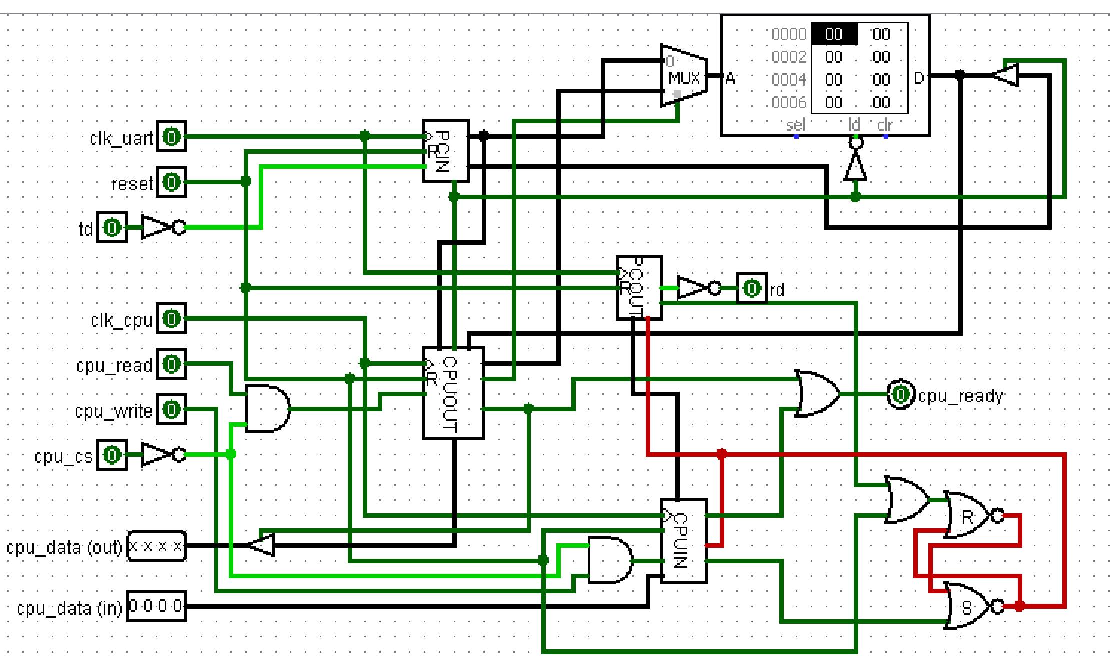


Figure 11.6: Low Level Design of the Serial Controller

The device at the top-right is the RAM cell. The pair of NOR gates at the bottom-right can (and should be) replaced with an SR-latch. Although the implementation requires a clock signal from the CPU, it may be replaced with the read signal from the CPU.

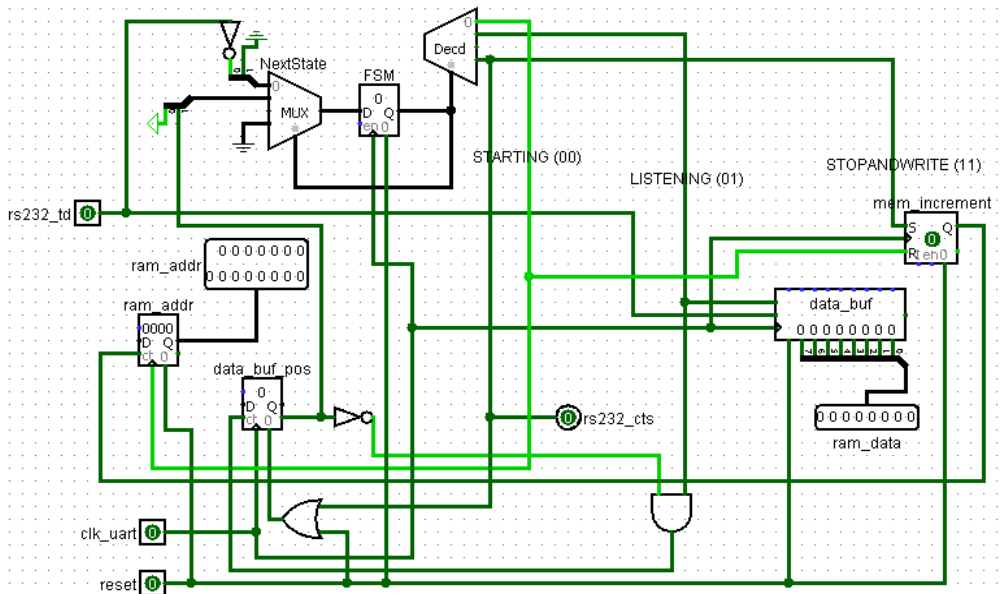


Figure 11.7: PCIN Component of the Serial Controller

A finite state machine (FSM) that puts incoming data into a shift register, transferring the data to the RAM cell once full and incrementing the RAM address. The clk_uart is a clock signal that has the same frequency as the baud rate (9600Hz).

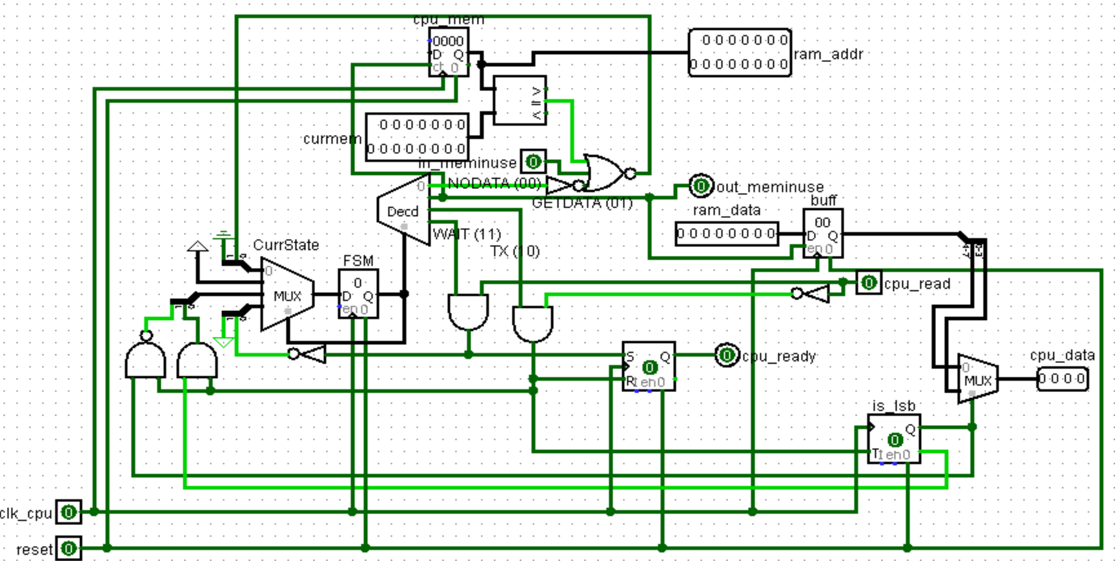


Figure 11.8: CPU OUT Component of the Serial Controller

An FSM which grabs data from the RAM when its RAM address counter no longer matches PCIN's RAM address, increments its own counter, and transmits data to the CPU when ready.

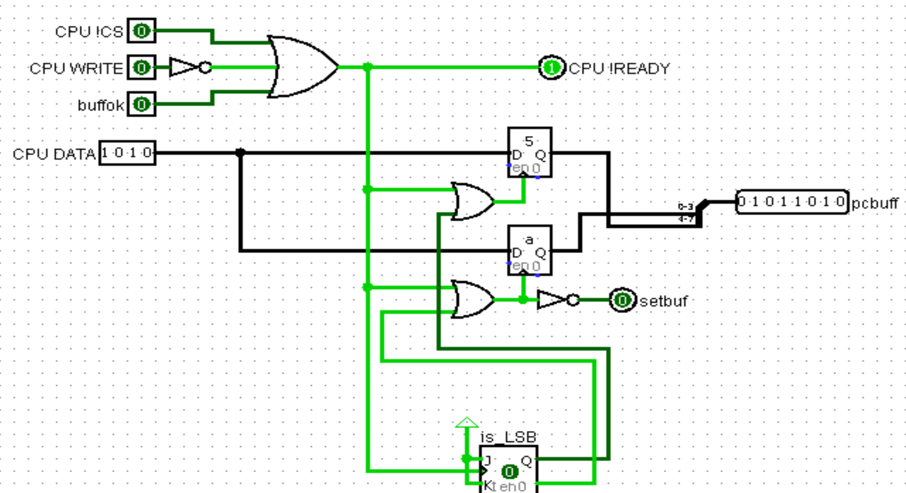


Figure 11.9: CPU IN Component of the Serial Controller

Takes data from the CPU and places into two 4-bit registers, ready to be used by the PCOUT component.

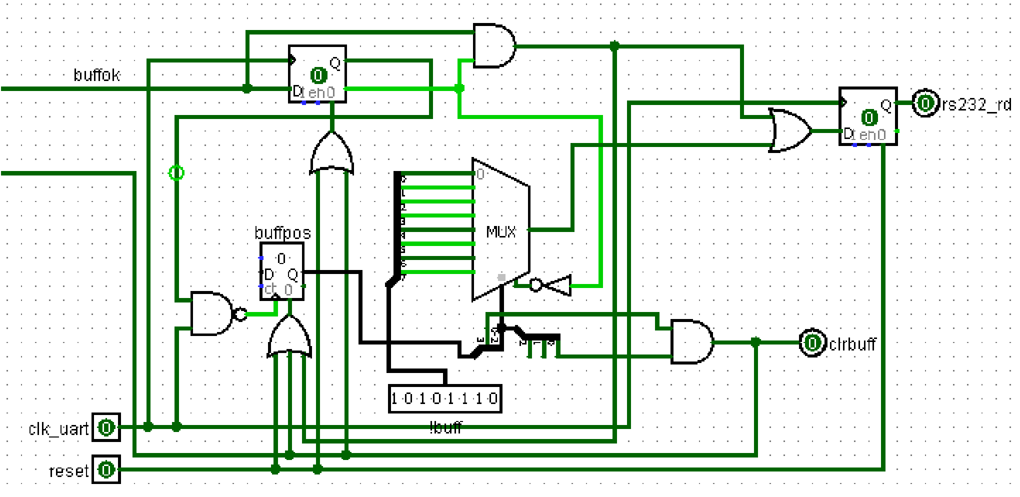


Figure 11.10: PC IN Component of the Serial Controller

Takes data from the registers in the CPUIN component and transmits, bit by bit, to the PC.

11.5. Testing the Serial Communication RS232 Controller

Serial communication is successful between a PC and the Nibble Knowledge CPU, as it provides a way to send arbitrary raw data. The peripheral allows a PC to input commands directly to the CPU; it also allows programmers to write programs that can send arbitrary data back and forth between the PC and a CPU.

9.5.2 Validation Tests and Results

Testbench files were written to test the VHDL code before being programmed to the FPGA. The testbenches simulated the input data, such as pulses coming in from the serial and data coming in from the CPU. To ensure correct results, the waveforms were analyzed to monitor the output. These tests allowed for simpler debugging, and all bugs were eliminated prior to programming. A snippet of the waveforms produced by the test is illustrated in figure 11.11 below.

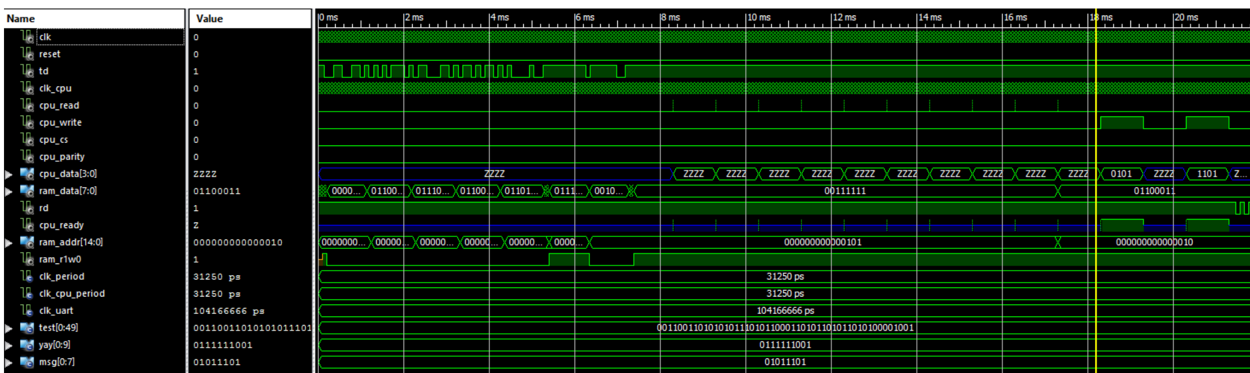


Figure 11.11: Serial Input VHDL Testbench Response

After programming, the FPGA was wired up as described in Figure 11.11. The serial's pins were soldered to the DB9 connector, and a serial-to-USB cable was used to connect to the PC. All CPU connections were connected to a breadboard, all inputs were connected to buttons and the output (CPU ready) was connected to the LED. Using the Tera Term program on the PC, bytes were sent from the PC to the peripheral; using the buttons, we determined if the same input would be displayed on the breadboard. Similarly, the buttons were used to send data to the peripheral; Tera Term was used to confirm that the correct data was sent through. The results confirmed that the FPGA was programmed correctly and the implementation of the serial protocol was correct.

After wiring the discrete implementation of the peripheral, it was tested in a similar fashion as above, being wired up to the breadboard. The tests determined that neither sending nor receiving data was working correctly. Due to a shortage of time, further testing or debugging was not conducted.

Finally, the FPGA implementation was tested with the discrete bus, communicating with the discrete CPU. The serial side was wired up to the same PC, the CPU side was disconnected from the breadboard and connected to the bus. Due to lack of time, the signals were shared with the ones from the VGA peripheral; which has a similar driver to the CPU. Tera Term was used to confirm

that the peripheral correctly sends data to the PC. Data receiving was not tested with the discrete bus.

Note: There is extensive amount of code that has been developed to design the Nibble Knowledge Computer. All of the code for different aspects of the Serial Communication RS232 Controller is located in an online, web-based Git repository hosting service. The Code is accessible through the link below:

https://github.com/Nibble-Knowledge

11.6 Serial Communication RS232 Controller Specifications

Component	Specifications
Clock Frequency	9600 Hz
Serial Logic Voltage	0.00 V to 5.00 V
Controller Voltage (FPGA)	0.00 V to 3.30 V
Data Rate	9600 baud
Communication Standard	RS 232 Standard

Table 11.2: Serial Communication RS232 Controller Specifications

12. Software

12.1 Cute BASIC

Cute BASIC is a BASIC like programming language developed for the Nibble Knowledge System. BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code. BASIC languages are built to be lightweight and easy to learn. Many programmable calculators such as the Texas Instruments brand use some variant of a BASIC language. It is a simple, general purpose high-level programming language. A high-level programming language is a programming language which uses natural language elements and is abstracted from the details of the computer. These languages are focused on usability and deal with things like variables, arrays, Boolean expressions, loops, functions and other abstract concepts rather than translating directly to the computer's opcode like a low-level assembly language would. Low-level languages will be described in the following chapter. These languages focus on usability rather than on efficiency.

It makes programming a lot easier and faster, as well as provides a translatable instruction set so that it can be easily learned for any spoken language instead of just English. Instructions in Cute BASIC are transformed into multiple lower level instructions which are understood and executed by the computer. These lower level instructions are difficult to read and maintain, and can be very time consuming writing out manually. Some of the more extreme cases will have one line of Cute BASIC being turned into dozens of lines of opcode which makes programming a much more efficient task. The downside to using Cute BASIC is that it can't perfectly optimize your program for computational efficiency - this can lead to unnecessary or redundant instructions for the computer to perform, however in majority of cases this is not an issue that needs to be addressed. It works by taking an expected line of code based on the language specification, or the language syntax and turning it into something the computer can understand. The syntax is the set of rules which the programming language uses to check if a document is correctly structured and understood. These rules vary based on each language. Each line of Cute BASIC code is translated into a series of Macro Assembly instructions, which themselves are turned into assembly instructions which are able to be understood by the computer, these topics will be covered further on.

The following is the language specification for Cute BASIC:

Declaration

LET (variable name)

Assigns value of var2 to var1. UNSIGNED keyword is implied. var2 can be a literal.

LET (variable name) BE (number)

Creates a variable called (variable name) and assigns (number) to it

LET (variable name) AS (size in nibbles)

Creates a variable called (variable name) with a size in nibbles of (size in nibbles)

LET (variable name) AS (size in nibbles) BE (number)

Creates a variable called (variable name) with a size in nibbles of (size in nibbles) and assigns (number) to it

Assignment

(var1) BE (var2)

Assigns value of var2 to var1. UNSIGNED keyword is implied. var2 can be a literal.

(var1) BE (var2) UNSIGNED

Assigns value of var2 to var1. Truncates or zero-extends

(var1) BE (var2) SIGNED

Assigns value of var2 to var1. Truncates (maintaining value of MSB) or sign-extends

Note for assignment and declaration: Values are stored big-endian

Unary math/logical/bitwise operators

(var0) BE NEGATE (var1)

Var0 evaluates to -(var1). In 2's complement, so (COMPLEMENT var1) ADD 1

(var0) BE INCREMENT (var1)

Var0 evaluates to (var1 + 1)

(var0) BE DECREMENT (var1)

Var0 evaluates to (var1 - 1)

(var0) BE NOT (var1)

Var0 evaluates to logical negation of var1

(var0) BE COMPLEMENT (var1)

Var0 evaluates to bitwise complement of var1

Binary math/logical/bitwise operators

(var0) BE (var1) ADD (var2)

Var0 evaluates to sum of var1+var2

(var0) BE (var1) SUBTRACT (var2)

Var0 evaluates to difference of var1-var2

(var0) BE (var1) MODULUS (var2)

Var0 evaluates to remainder of var1/var2

(var0) BE (var1) MULTIPLY (var2)

Var0 evaluates to product of var1*var2

(var0) BE (var1) DIVIDE (var2)

Var0 evaluates to product of var1/var2

(var0) BE (var1) ALSO (var2)

Var0 evaluates to 1 if both var1 and var2 are TRUE

(var0) BE (var1) EITHER (var2)

Var0 evaluates to 1 if either var1 or var2 are TRUE

(var0) BE (var1) AND (var2)

Var0 evaluates to bitwise AND of variable

(var0) BE (var1) OR (var2)

Var0 evaluates to bitwise OR of variables

(var0) BE (var1) XOR (var2)

Var0 evaluates to bitwise XOR of variables

(var0) BE (var1) NAND (var2)

Var0 evaluates to bitwise NAND of variables

(var0) BE (var1) NOR (var2)

Var0 evaluates to bitwise NOR of variables

(var0) BE (var1) XNOR (var2)

Var0 evaluates to bitwise XNORof variables

(var0) BE (var1) LSHIFT (var2)

Var0 var2 should be small; left-shifts by var2, fills right with zeros, truncates anything that falls off left

(var0) BE (var1) RSHIFT (var2)

(var0) BE (var1) LROTATE (var2)

Var0 var2 should be small; left-rotates by var2, anything that falls off left gets put back on right

(var0) BE (var1) RROTATE (var2)

Logical Comparison

(var0) BE (var1) EQUALS (var2)

(var0) BE (var1) NOTEQUALS (var2)

(var0) BE (var1) GREATER (var2)

(var0) BE (var1) GREATEREQUALS (var2)

(var0) BE (var1) LESS (var2)

(var0) BE (var1) LESSEQUALS (var2)

Pointers

(var0) BE CONTENTOF (var1)

Var0 evaluates to dereferenced value pointed at by var1

(var0) BE ADDRESSOF (var1)

Var0 evaluates to (16-bit) address of var1

Loops and Conditionals

LOOPWHILE (var1)

ENDLOOP

IF (var1)

ELSE IF (var2)

ELSE

ENDIF

LOOPAGAIN

Return to top of loop, as "continue"

EXITLOOP

Exit innermost loop, as "break"

EXITIF

Exit innermost "if" block, as "break"

Labels and Gotos

LABEL (name)

GOTO (name)

Functions

FUNCTION (funcName) RETURNS (size in nibbles) TAKES [(size in nibbles) (paramName)] ASWELL…

FUNCTION (funcName) TAKES ... RETURNS (size in nibbles)

FUNCTION (funcName) TAKES [(size in nibbles) (paramName)]

FUNCTION (funcName) RETURNS (size in nibbles)

FUNCTION (funcName)

RETURN (var1)

Exits function, returns var1

CALL (funcName) (var1) ASWELL (var2)...

Var1, var2, are parameters. Evaluates to return type of funcName; evaluates to TRUE if nothing returned

I/O

NOHEADER

Remove CUTE BASIC header output

CALL GETCHAR (variable) [AT (index)] (file name or STDIN if blank) [AT (index)] [FOR #]

Gets a character from file or STDIN(keyboard) for # chars

CALL PUTCHAR (variable) [AT (index)] (file name or STDOUT if blank) [AT (index)] [FOR #]

Writes a character to file or STDOUT(screen) for # chars

12.2 Macro Assembler

The Macro Assembly is a set of macro instructions that work with the CPU assembly language. A macro instruction is an assembly language pseudo instruction that can be directly expressed as a sequence of normal assembly language instructions. In essence, it is a shorthand way of writing several normal instructions. We use these macro instructions to express common procedures that are not directly supported by the CPU assembly language.

Why do we use it?

Our CPU assembly language is exceptionally simple. It was designed to be the minimal set of instructions that are necessary to do all of the basic operations common to computing. Unfortunately, it being possible to do a basic operation does not mean it is simple to figure out how to do that basic operation. Take, for instance, the macro instruction definition for a 4-bit Exclusive-or (XOR) operation:

```
binMac["XOR"] = """
LOD $op1
NND $op2
STR macro[0]
NND $op1
STR macro[1]
LOD macro[0]
NND $op2
NND macro[1]
STR $dest"""
```

The procedure above is nine instructions, and requires use of two memory locations as temporary variables. Writing it out longhand every time it is needed would not only be an incredible waste of time, but would also increase the number of errors that make it into our code. It would be very easy to mess up and put a “0” where we actually need a “1”, or switch a NND for an STR. It would also be very hard to figure out where this mistake was made. So, to avoid this problem, we decided to write the procedure once, and make sure that it was correct that one time.

Another benefit of writing things down once and referring back to them is that we can be very clever with how we define our macro instructions, and make them very efficient. Often times, there exist several ways of doing something, and the most obvious or direct method is not always the fastest or best. For instance, the standard textbook procedure for doing a two's complement comparison operation is to:

1. Bitwise complement the second operand
2. Add 1 to the complemented second operand, to finish negating it
3. Add the first operand to the negated second operand
4. Fetch the two's complement comparison flag

In our system, following this procedure exactly would produce the following code:

```
NOT $op2 INTO macro[0]
INC macro[0]
ADD macro[0] $op1 INTO macro[1]
GETCMP ACC
```

This works out to nine instructions per nibble, plus the three instructions of GETCMP ACC. By artfully combining instructions, the standard procedure we actually use for comparison operations works out to just three instructions per nibble. This means that a one-time optimization effort allows all future programmers to have code that is just one third the size of the "most obvious" solution!

The third benefit of the Macro Assembly is that we can deal with large numbers in a transparent way. The CPU can only count to 15 (unsigned), or the range from -8 to +7 (signed). We might need to count to twenty someday, or maybe twenty million. By creating a set of macros that deal with multi-nibble computations, we can allow a programmer to deal with numbers up to sixty-four bits long, without them needing to fully understand how to propagate the computation from one nibble to the next.

How does it work?

The macro assembler itself is made of three main parts:

- The front end script
- The macro expansion core
- The macro definition libraries

The front end script is the simplest part. It opens and manages files, and reads the command-line arguments it is given. It also directly handles file inclusion, by stacking the program code and data sections into our stack-of-pancakes structure.

The macro expansion core is a couple of files of python. It is given input one line at a time. For each line, it determines if that line is a macro instruction, a "fall-through line" (such as a CPU assembly instruction or a comment), or an error. If the line is a macro instruction, it looks up the definition of that instruction, and works out the labels and offsets needed to apply that definition. Finally, it passes each line of the resulting code to itself, in case it contains even more macro instructions. All of the macro definitions eventually expand down to CPU assembly, so this passing code to itself is guaranteed to end.

The macro definition libraries are the largest component of the macro assembler. It creates a set of Python "Dictionary" objects that allow the macro expansion core to look up the definition of each macro instruction. Some of the macro definitions refer to other macros, but we were very careful to ensure that there were never any "cycles" of macro A referring to macro B referring back to macro A, because these would make the macro expansion core get lost in an infinite recursion. Another careful design aspect of the macro definitions is the use of macro scratch space memory. This is a block of memory reserved specifically for the macro instructions, but they need to coordinate how they use it so that they do not start overwriting each other's data. Fortunately, macros only use memory inside the set of instructions they expand to, so when using macro memory, you only need to consider the macro memory use of the other macros your macro uses. Sounds a bit complex, but it's relatively easy in practice.

12.3 CPU Assembler

The Nibble Knowledge CPU Assembler interfaces directly with the CPU and effectively converts assembly language code made up of 12 recognized language constructs and converts them into binary which can be run on the CPU. An assembler is a program which takes low-level assembly language programming and converts it into a pattern of bits which is recognized and executed by the computer. Assembly is a low-level programming language, that means that is has little or no abstraction from the computer's instruction set and that commands map closely to processor instructions. Due to how close this relationship it low-level code is generally non-portable - meaning that it's ability to run depends on if it is compatible with the CPU hardware.

Why do we use it?

It translates our 12 instructions into a series of bits which the CPU recognizes and can run. It is how we make the computer do what we want it to do, rather than just sit there and be a fancy box. In other words, if we want the computer to run any programs, we need a functioning assembler. If used directly it can make programs extremely efficient by programming them in a low-level assembly language, as it is possible to make it so that every instruction is purposeful and there is zero redundancy or repetition unlike that which gets added when using high-level languages. In most situations this is not a necessity, but in extremely long or time sensitive projects this optimization can make a massive difference, especially if it is in a subroutine which will be called and executed many times.

How does it work?

It works by reading in an assembly file line by line, and outputting a pattern of bits which are recognised by the CPU. These patterns tell the CPU what to do and depend on the architecture of the CPU. The Nibble Knowledge CPU recognises only a small number of patterns in order to keep its complexity relatively small. The patterns recognised by Nibble Knowledge will be different to those recognised by an ARM or x86 processor and thus binary code created for one system will not usually translate to another system. Like a higher-level language assembly languages each have their own syntax as well, and you must make sure you are using the correct syntax for the language you are programming in. Each pattern will cause the CPU to perform a specific action, and when chained together, these actions allow the computer to function as seen in everyday life - from things such as starting up, running a program, or typing on the keyboard.

12.3.1 8 Instructions

The Nibble Knowledge CPU has 8 instructions which are split into two different types:

- Solitary instructions that do not require a memory address (has the format INST)
    - NOP: No operation
    - HLT: Halt CPU
    - CXA: Copy the overflow bit and the XOR of that bit and the MSB of the accumulator.
- Binary instructions that require a memory address (has the formal INST ADDR)
    - ADD: Add the nibble at the specified memory address to the accumulator.
    - NND: NAND the nibble at the specified memory address to the accumulator.
    - JMP: Jump to the specified memory address if the accumulator is 0.
    - LOD: Load the accumulator with the nibble at the specified address.
    - STR: Store the nibble in the accumulator to the specified memory address.

12.3.2 Pseudo Instructions

To aid in disassembly a new metadata format for binary files is now included in the assembler as of v1.1.0.

- INF - the information section must start with this, and this should be the first instruction of any file.
- PINF - the start of the executable information section. Any unknown tuples within the executable information section are treated as pseudo instructions with a data field.
- BADR - The base address of the binary file. Must be in the executable data section.
- EPINF - the end of the executable information section.
- DSEC - the memory location of the data section which should succeed the text section. Should be a label so that it is modified by the base address and can be reliably disassembled. If there is no data section, this should point to the end of the file.
- Data section descriptors:
    - DNUM - the amount of data sections of the following size
    - DSIZE - the size

    - There should be as many DNUM/DSIZE pairs as there are unique groups of data sections. The example below is illustrative. These pairs must be in the same order as the data sections themselves.
- EINF - end of the information section.

12.3.3 Two Data Types

AS4 recognizes two inbuilt data types:

- Numerical values. Format: ".data SIZE INITIALVALUE"
    - .data can also be used to create a static reference to a label. The SIZE must be 4. The format is ".data SIZE LABEL" or ".data SIZE LABEL [OFFSET]". This will save the static 16-bit memory location pointed to by LABEL or LABEL + OFFSET to the .data section.
- Strings, both plain and zero terminated. Format: ".ascii "String"" or ".asciiz "String""
    - Strings must start and end with double quotes.
    - AS4 recognises standard escape characters

12.3.4 Labels

Labels are of the format "NAME:”. They are used to refer to memory locations without having to memorize or calculate number. An example of usage would be "number: .data 1 2", which is using the label "number" to point to a data element of 1 nibble in size with the initial value of 2.

Labels when referenced in instructions can be used in two forms:

- INST LABEL
    - Where the instruction INST simply references the memory location pointed to by LABEL
- INST LABEL[OFFSET]
    - Where the instruction INST references the memory location pointed to by LABEL + OFFSET. OFFSET is usually a hexidecimal value, optionally preceded by "0x". To use a binary value, prefix with "0b". For an octal value, prefix "0" or "0o". For a decimal value, prefix "0d".

An example of usage would be "LOD sum [F]", which loads the memory address pointed to by "sum" plus the offset of "F" (15 in decimal) into the accumulator.

12.3.5 Address of Operations

As all programs built by AS4 are assumed to be static, non-relocatable binary files, the addresses pointed by labels can be calculated at assemble time and used statically. The form is below:

- &(LABEL[OFFSET])[ADDRESS_OFFSET]
    - LABEL[OFFSET] is the same as for the standard label usage. &() indicates this is an address of operation, and [ADDRESS_OFFSET] is what 4-bit portion of the 16-bit address you want. Both [OFFSET] and [ADDRESS_OFFSET] are optional, without them an offset of zero is assumed.

This loads a corresponding value from the table of static values - for example, if the address of the label "Carmen" is 0x00FE, and you use the address of operation LOD &(Carmen)[1], you would load "0xF" into the accumulator; which is the same as using the instruction LOD N_[F] when the N_ static number series is defined.

12.3.6 Comments

Comments in AS4 start with a semicolon, ";" or an octothorp, "#".

12.3.7 Numbers

AS4 accepts binary, octal, hexadecimal and binary numbers. Binary numbers must always be preceded by "0b" and octal by "0" or "0o". The rules for decimal and hexadecimal vary depending on use case.

- When used for an offset, in the form LABEL[OFFSET], it is assumed the default is hexadecimal. Thus, hexadecimal numbers can be written with or without a preceding "0x". Decimal numbers must be written with a preceding "0d".
- In all other cases, decimal is assumed to be the default. "0d" can optionally precede the decimal number. Hexadecimal numbers must be written with a preceding "0x".

Note: There is extensive amount of code that has been developed to design the Nibble Knowledge Computer. All of the code for different aspects of the Software design is located in an online, web-based Git repository hosting service. The Code is accessible through the link below:

- **https://github.com/Nibble-Knowledge**

13. VGA Controller

13.1 Introduction

A VGA Controller is the interface between a VGA cable and the CPU. It is one of the most crucial elements in ensuring the words or numbers you type on the keyboard end up on the monitor. Signals transmitted from both the CPU will be evaluated within the controller and an appropriate sequence of pixels will be stored in the controller's memory. The controller will then transmit these pixels to the screen for display in the specified order.

The purpose of the VGA controller is to facilitate communication between the screen and the CPU. The reason for the use is to turn the electrical signals transmitted by the CPU into valuable data that is read by the screen when it is convenient for it. This allows the VGA controller to run at its required minimum speed (25.175 MHz) without the need for the CPU to be operating at the same frequency.

VGA Port

The VGA port is the location where the VGA cable plugs into the controller. All communication from the controller to the screen is transmitted through this port. The protocol between the VGA cable and the VGA Controller is partly analog and semi-synchronous. There are five important signals sent through the VGA port: The Vsync, Hsync, Red, Green and Blue signals. First the graphics card inside the monitor analyzes the Vsync and Hsync signals to determine the resolution being requested. In our case this is 640x480 at 25.175MHz (the minimum requirement). Then based on the resolution (specifically on the resulting frequency) it samples the analog signals Red, Green and Blue to determine the color being displayed for a particular pixel. When Hsync is low the screen repositions itself to begin displaying the next row of pixels. When Vsync is low the screen repositions itself to begin the cycle over again. The VGA Port pin-outs are described in the figure below.

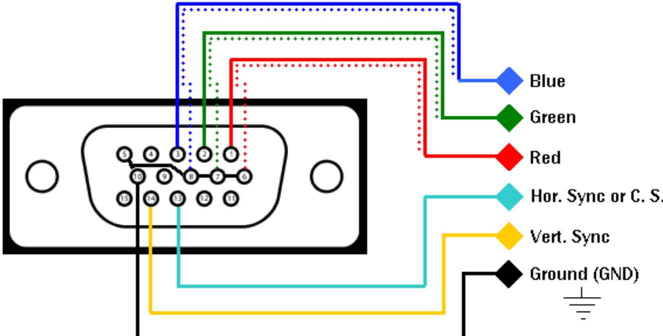


Figure 13.1: VGA Port Pin-Outs

In the Nibble Knowledge design, all other pins besides the five mentioned above were grounded as the intention was to imply the minimum requirements and thus did not need the other pins.

Controller to VGA port Transmission

The controller is made of three major parts: the input block, the RAM/ROM and the output block. The input block is not used for controller to VGA port transmission but the other two blocks are.

The output block consists of a series of counters that use a signal generated by a 25.175MHz crystal oscillator to create three signals: Hsync, Vsync and Blank. Hsync and Vsync are used as above, Blank is used to prevent pixels from being sent while Hsync and Vsync are resetting the screen. One set of counters increment every rising edge of the oscillator and when they reach the value 640 (the width of the screen) activate the Blank signal to prevent pixels from being sent when they are not needed. Once the counters reach 656 Hsync begins, when they reach 753 Hsync ends and once they reach 800 Blank is turned off, the counters are reset and a second set of counters is incremented. This second set is used to activate Vsync and Blank in a similar fashion, once they reach the values of 480, 490, 492 and 525 the Blank signal is activated, Vsync Begins, Vsync ends and the Blank signal is deactivated respectively.

Inside of the RAM/ROM are two memory banks: a character buffer and a pixel map. When the CPU is not attempting to send data to the controller the controller is in read mode. While in read mode the controller constantly cycles through the character buffer, sending the resulting ASCII code to the pixel map. The pixel map then looks up the ASCII code and outputs the corresponding pixels associated with whichever row is required of the ASCII character being requested.

CPU to Controller Transmission

The input block is set up as a finite state machine with four states, data1, data2, write and wait. By receiving a falling edge on the write signal sent by the CPU and confirming that the parity and select lines are high and low respectively the machine cycles through the four states. In the data states, data being presented on the line is sampled and stored in either the top 4 bits or the bottom 4 bits of an 8 bit DFF array. Once these two states have occurred, the next state (write) ignores the data on the bus but simply informs the RAM/ROM to enter write mode and grab the data. Finally, the wait state is used as a "cooldown" period where nothing happens in order to allow for time for the memory in the RAM/ROM to update. In the RAM/ROM the controller enters write mode after receiving the above mentioned signal. While in this mode the character buffer takes the values stored in the DFF's of the input block and stores them in memory. Then the controller enters read mode again.

13.2 FPGA Implementation

As an intermediate step in the development of the discrete VGA controller we created a FPGA implementation of the peripheral. It was a great first step because it allowed us to use a behavioral approach at designing the peripheral. Most importantly it simplified the task of creating a discrete circuit because we were able to take the VHDL code and transform it into discrete logic chips.

Example: VHDL code to discrete logic chips conversion

VHDL code: if (cpu_write = '0' and parity_check = '1')

Logic chips: A 2 IN – 1 OUT AND gate

- Inputs: NOT cpu_write, parity_check
- Output: the result of the following 2 signals AND gated together

13.3 Discrete Implementation

Receiving Signals from the CPU

As stated above, when the CPU has data for the VGA controller it sends four "pulses" along the write line while selecting the VGA controller. During these pulses a falling edge detector will cause an update to a finite state machine causing either the reception of 4 bits of data from the

CPU or the transmission of 8 bits of data to the character buffer memory. After the data has been received the controller goes back to read mode.

The main components of this write cycle are a system of DFF's, counters and RAM. One counter is activated on the falling edge of the write signal (provided that the parity check and chip select match) and is used to determine the current state of the finite state machine. Its output is sent to a demultiplexer which outputs signals to either the top 4 bits of a DFF array or the bottom 4 bits of the same array or to the RAM/ROM.

Inside the RAM/ROM, there a system of counters used to keep track of the current character being written. When a signal is sent from the input block these counters increment and then pass their data as the address to the character buffer where the data being sent from the input block is then stored.

Sending Signals to the VGA port

As stated above, the Hsync and Vsync values are determined by two systems of counters as well as a series of AND Gates and two DFF's being fed the inverse of their output value (making them small, 1 bit counters). The other signal created is Blank which simply sets the output to zero when it is on and does not allow a new character to be processed. When Blank is not on, the output block sends a pulse every eight clock cycles (through the same process as HSYNC) to the RAM/ROM to update to the next character. Every time HSYNC occurs it also sends a signal to inform the RAM/ROM to write the next row of each character.

In the RAM/ROM like during the write cycle a system of counters is used to keep track of the current character being read. This character is then passed to the pixel memory where it is combined with the current row being used and the appropriate set of data is sent to the output block to be used as a set of pixels.

High Level Block Diagram

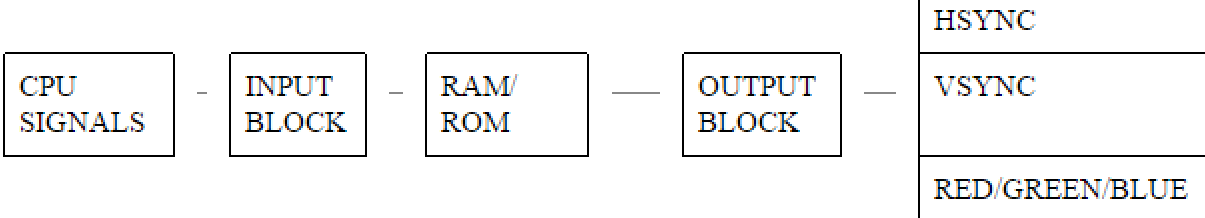


Figure 13.2: VGA Controller High Level Block Diagram

Overview of the entire Discrete Circuit

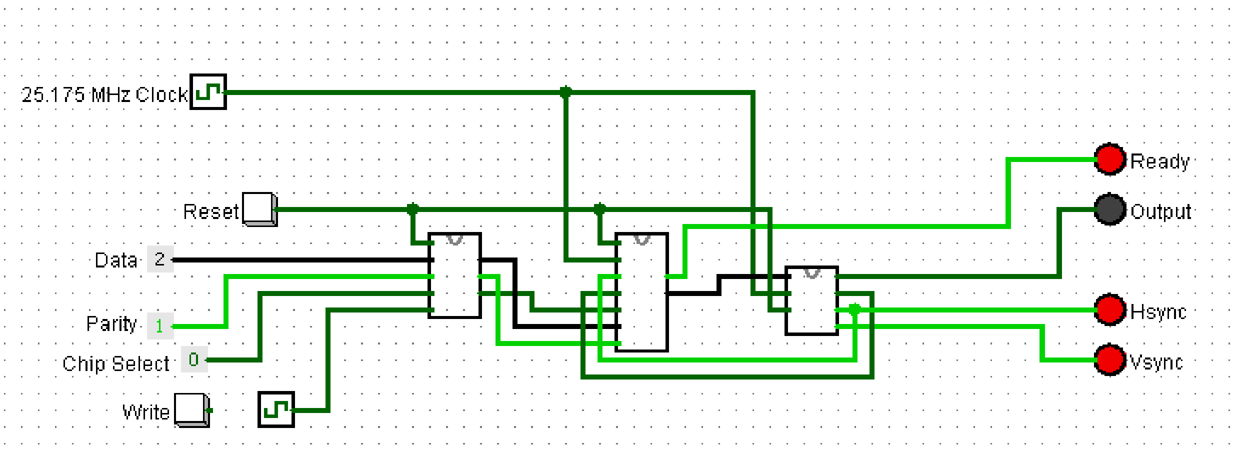


Figure 13.3: VGA Controller Discrete Overview

The Input Block Discrete Circuit

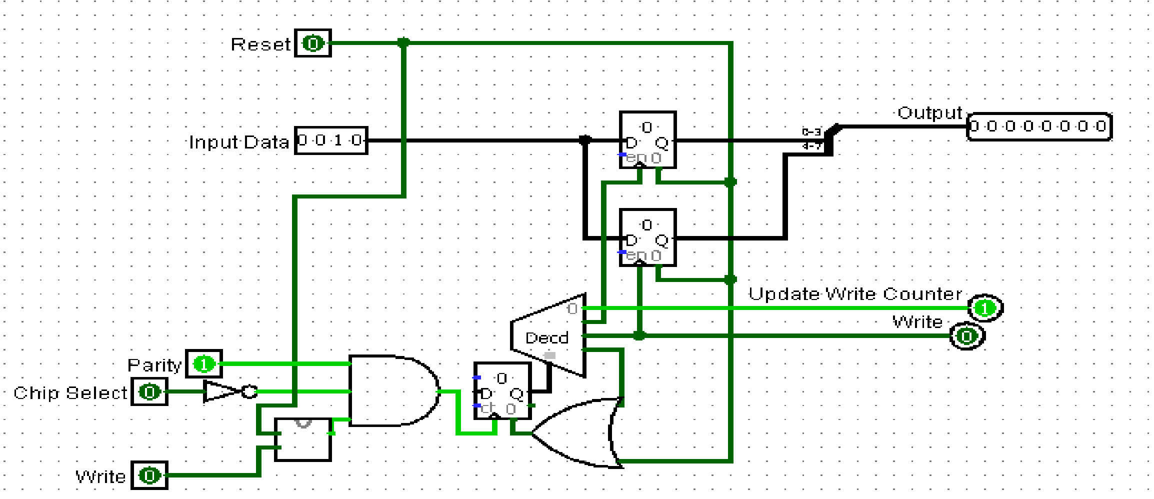


Figure 13.4: VGA Controller Input Block Circuit

The RAM / ROM Discrete Circuit

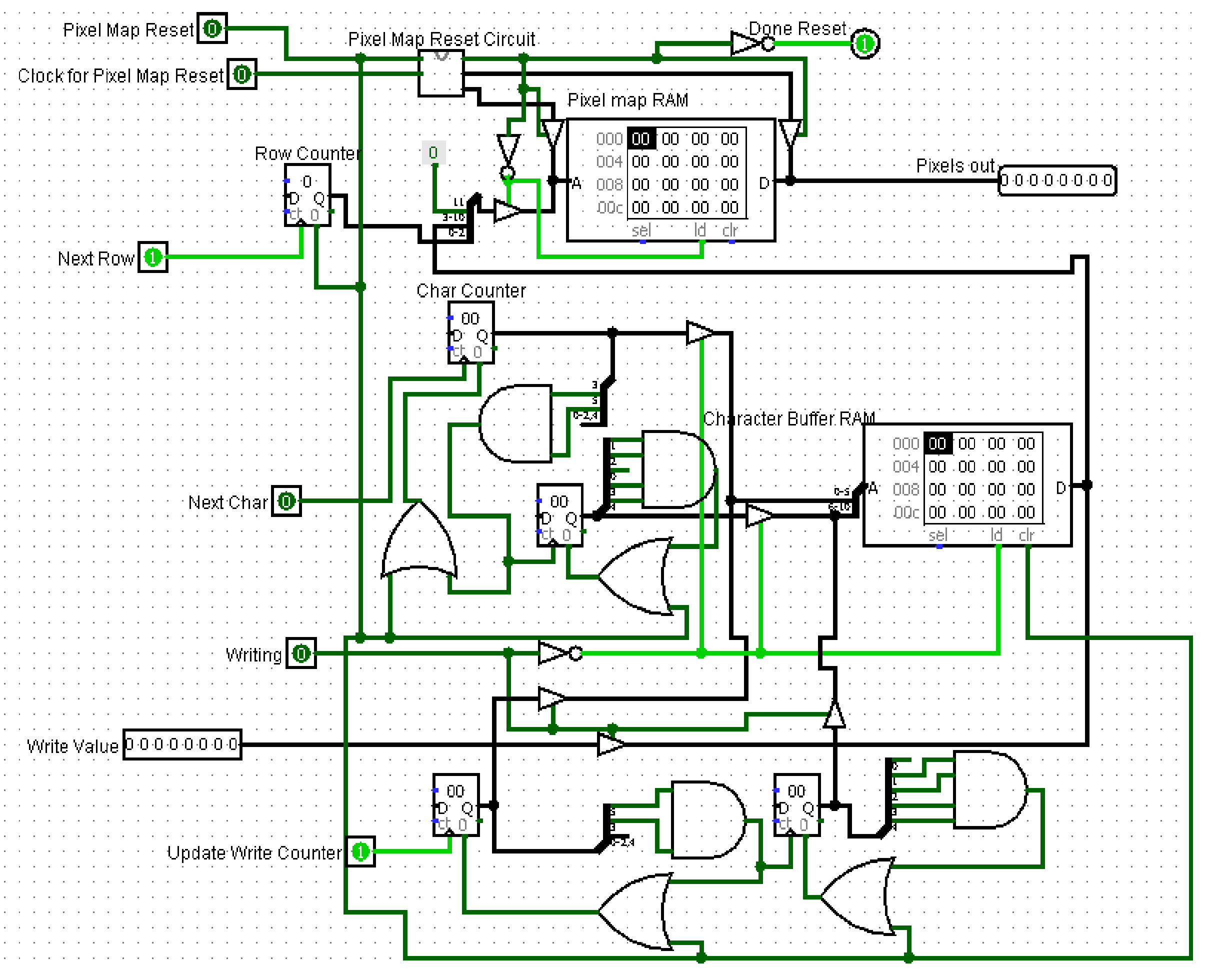


Figure 13.5: VGA Controller RAM / ROM Circuit

The Output Discrete Circuit

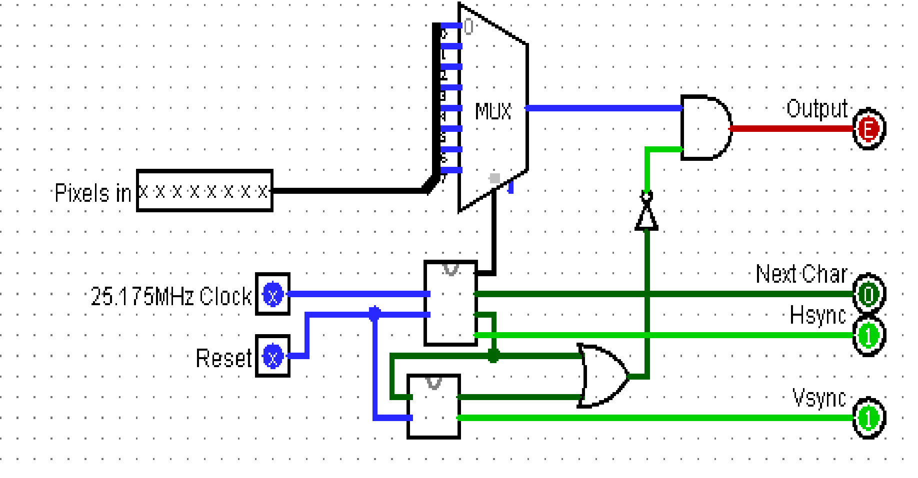


Figure 13.6: VGA Controller Output Block Circuit

The HSYNC Discrete Circuit

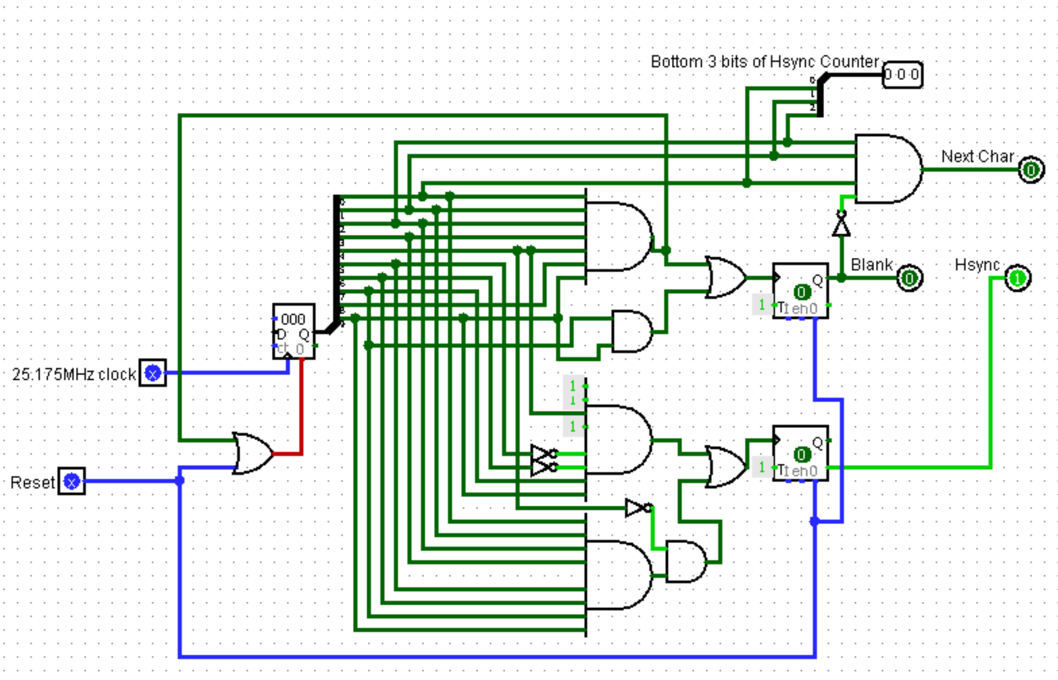


Figure 13.7: VGA Controller HSYNC Circuit

The VSYNC Discrete Circuit

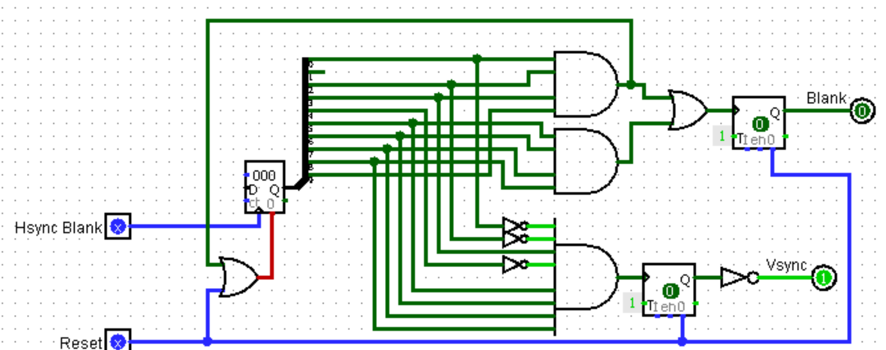


Figure 13.8: VGA Controller VSYNC Circuit

13.4 Testing the VGA Controller

The validation tests relevant to the VGA controller verified that the:

- VGA design successfully worked virtually
- VGA design worked with push button inputs
- Discrete design worked with push button inputs and as a fully functional discrete peripheral

Although validation was a great tool for verifying the successfulness of our design it also provided us with the information and tools needed to move from an FPGA implementation to a fully functional discrete implementation of the VGA Controller.

Validation Testing Details/Procedures and Results

The validation tests completed for this peripheral include:

1. Test bench testing (waveform simulation) at a software level to verify the behavior of the VHDL code written for the FPGA implementation of the VGA keyboard controller.
2. LED testing for FPGA implementation of the VGA keyboard controller with push button inputs for CPU signals
3. Full system testing for FPGA implementation of the VGA keyboard controller with push button inputs for CPU signals
4. Virtually testing the designed discrete VGA controller within LogiSim with virtual push button inputs and LED outputs to verify behavior
5. LED testing for discrete implementation of the VGA controller with push button inputs for CPU signals

6. Full system testing with discrete CPU and VGA controller

During the assembling of the Discrete VGA Keyboard Controller a multi-meter was used to trace specific signals from input to output to verify if the circuit was working correctly. Also, a probe kit was used to trace multiple signals at once when necessary. This was an excellent method to detect any errors in the construction of the discrete circuit. We are quickly able to identify any bugs and take the appropriate measures to remedy the problems. Overall validation tested was valuable to not only ensure our controller worked as intended but it also helped in the designing process when moving from a FPGA to a discrete implementation.

Note: There is extensive amount of code that has been developed to design the Nibble Knowledge Computer. All of the code for different aspects of the VGA Controller is located in an online, web-based Git repository hosting service. The Code is accessible through the link below:
https://github.com/Nibble-Knowledge

13.5 VGA Controller Specifications

Component	**Specifications**
Controller Voltage Logic	5.00 V
CMOS/TTL Chips V_{cc}	5.00 V
CMOS/TTL Chips VIH	2 – 3.15 V
CMOS/TTL Chips VIL	0.8 – 1.53 V
Port Red, Green, Blue Lines	0.00 V to 0.70 V
Clock Frequency	25.175 MHz minimum

Table 13.1: VGA Controller Specifications

14. CPU and Peripheral Integration

The project encompasses the design of a 4-bit computer system that holds an educational value, where an individual would be able to learn the internal functionalities of a CPU. The high level design is divided into four subsections to clearly explain the implementation of the preferred solution. This will enable us to show how the various components of the solution are related and interact with each other.

Overall Design: CPU and Connected Devices

The overall design consists of CPU, Memory, Peripheral Controllers, Peripheral Devices and a Peripheral Bus. The five peripheral devices are the Serial Network, VGA Monitor, IDE Hard Drive, PS/2 Keyboard and Speaker. The Serial Communication will be used to communicate with a local area network. The VGA Monitor will display a visual output, while the speaker will output audio waveforms. The PS/2 keyboard will be able to take user input. The CPU controls all the above processes. Each of the peripheral devices will be connected to their own controller device, which is connected by a 4-bit data bus to the CPU. The controllers will govern the interaction process between the peripheral devices and the CPU, for instance, data transfer, error detection, communication and timing control. Both RAM and a boot ROM will to store data for execution.

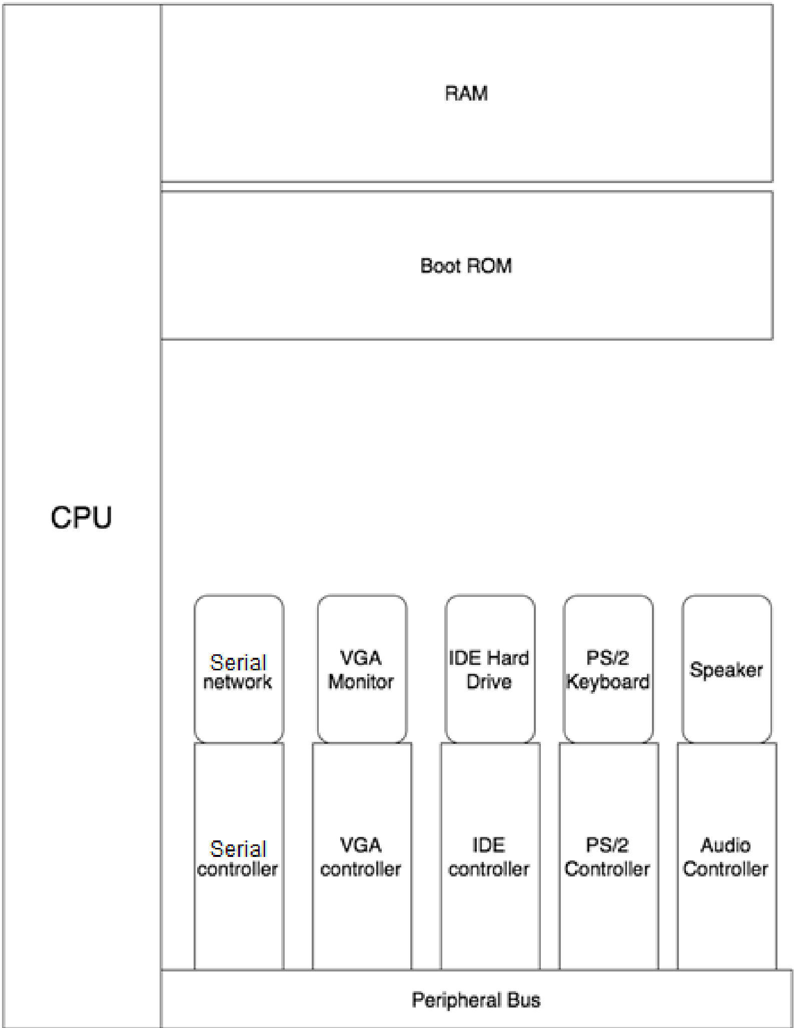


Figure 14.1: A Diagram of CPU and Connected Devices.

The Bus – Connection between CPU and an External Device

A computer bus is a component that transfers information between different devices. Different protocols can be used for a bus such as master-slave, taking turns, or open communication. Either in the device interface or in the protocol itself you'll usually find safeties in place so that the information is not corrupted in transit and that the correct device is sending or receiving information.

Communication Protocols

In everyday life we experience many protocols such as saying hello before talking to someone. These protocols allow us to interact in a predictable manner so that we make sure the other person is listening or that you don't all talk at once.

Three examples in the computer world are as follows:

Master-slave

This protocol says that a device is declared the master and decides who gets to talk on the bus and who gets to listen.

Taking turns

This protocol requires that a token is passed around. If you have the token, you can speak and in this type of bus everyone listens and decides independently if the information is pertinent for them.

Open communication

In this protocol everyone is talking and listening at the same time. This requires that you have ways to differentiate one device from the next. Ways to do this might include a name or frequency selection. This can be difficult as it is easy to interfere with other simultaneous transmissions. Think of trying to talk in a loud room.

The Nibble Knowledge Bus

The Nibble Knowledge Bus uses a master-slave protocol and the CPU chooses which peripheral that it communicates with at all times.

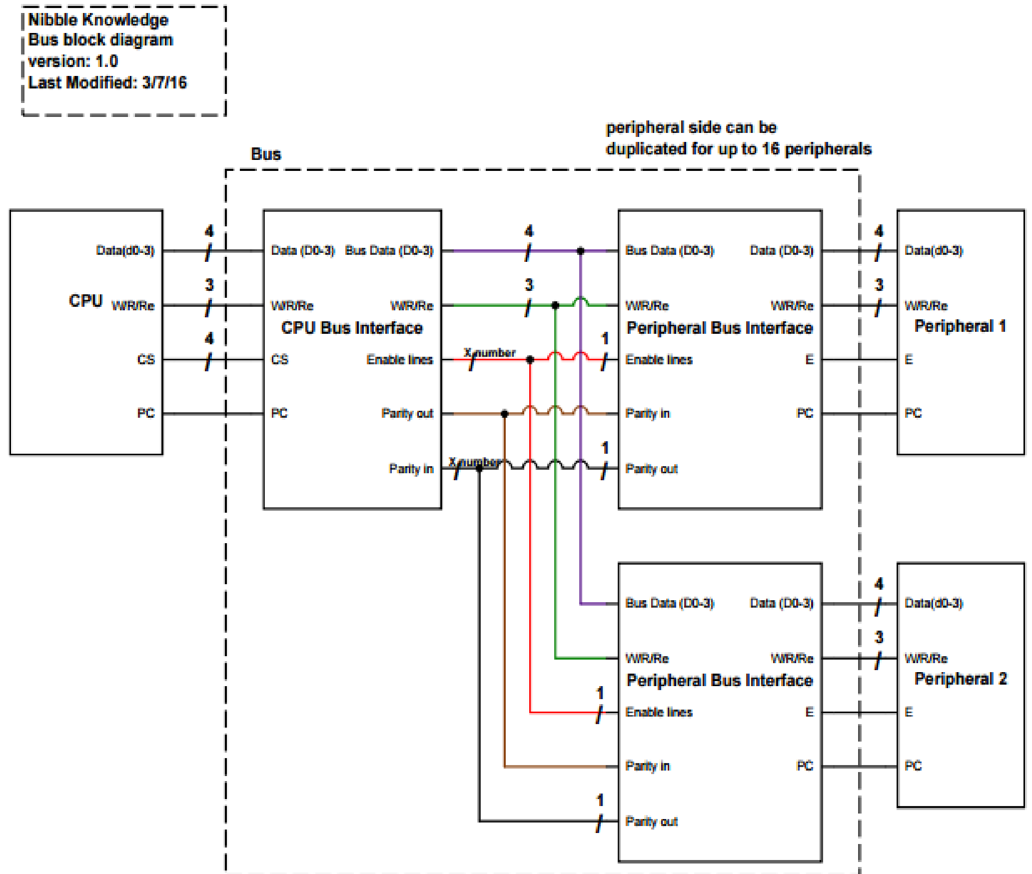


Figure 14.2: High Level Diagram of the Bus

The Nibble Knowledge bus uses the Chip Select (CS) lines to select which peripheral it wants to communicate with. This connects the data lines and the Write (W), Read (R), and Ready (Re) to their counterparts on the correct peripheral. If Read is high then the CPU is requesting data from the peripheral, however, if Write is high the CPU is wanting to send data to the peripheral. Ready is what the peripheral uses to tell the CPU that it has sent or received the data. At any time if the Parity Check (PC) is not high then there was a transmission error and the devices must independently decide what to do.

An example of communication between the CPU and a peripheral would look like the following:

CPU Write to Peripheral

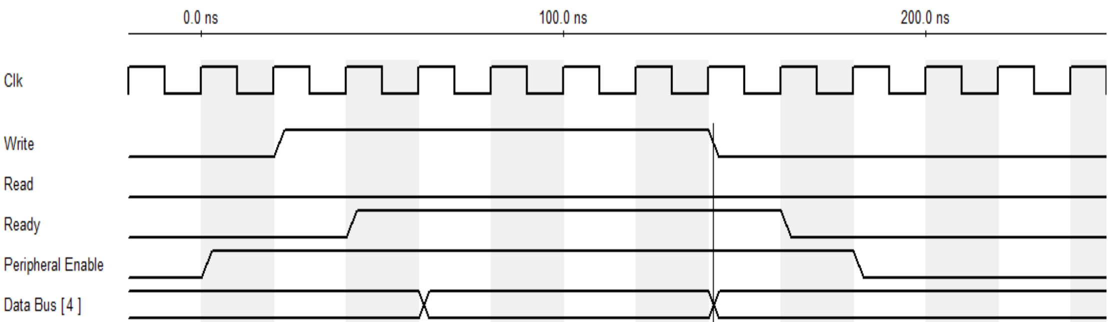


Figure 14.3: Communication – CPU Writes to Peripheral

CPU Reads from Peripheral

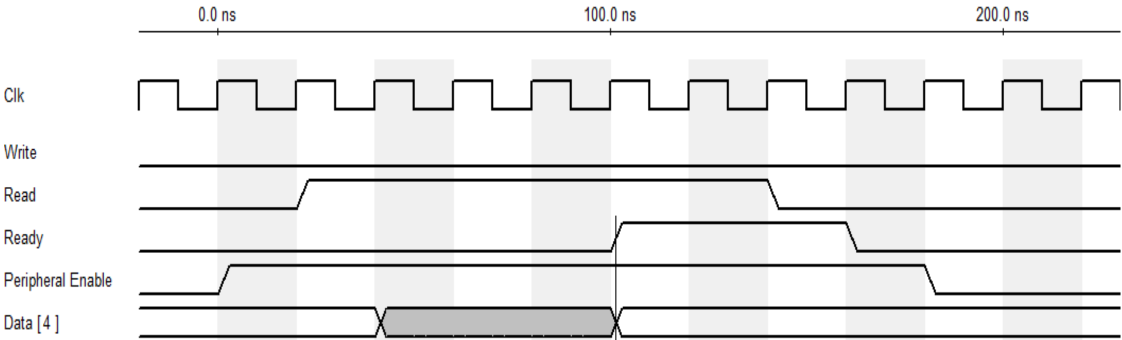


Figure 14.4: Communication – CPU Reads from Peripheral

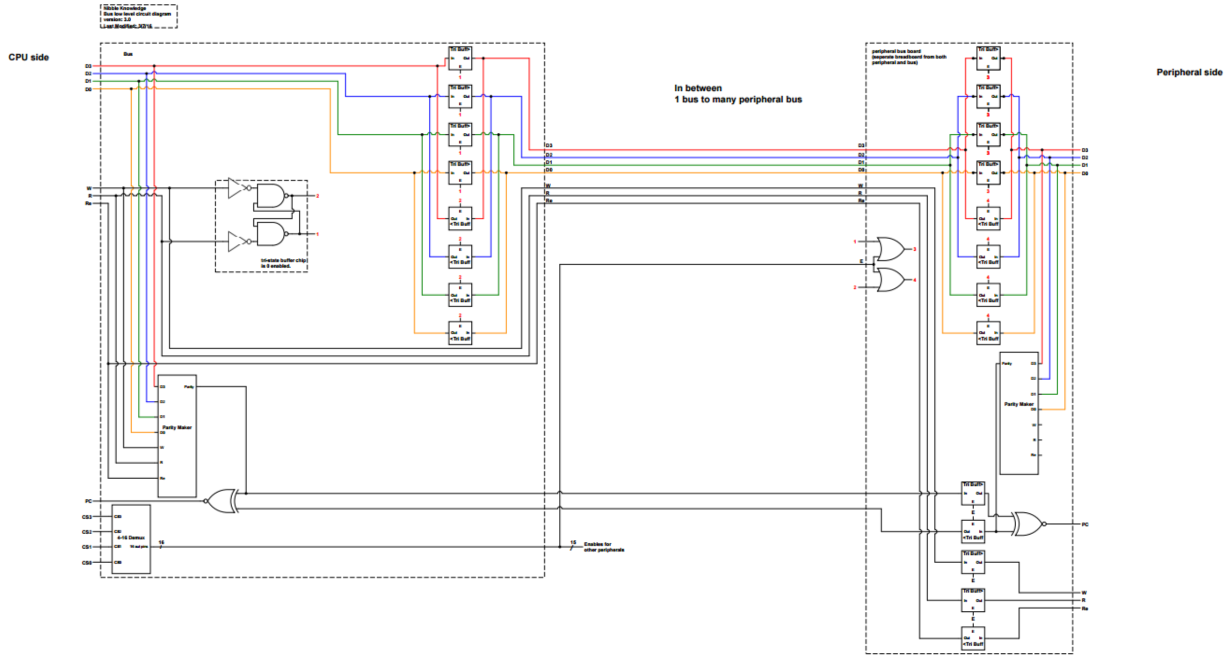


Figure 89: Low Level Diagram of the Bus

15. Glossary

Adder (electronics): An adder is a digital circuit that performs the addition of numbers.

Arduino: Arduino is an open source electronic prototyping platform that allows creating electronics projects that are controlled by a microcontroller. It is based on an AVR microcontroller and includes an IDE and a C-like language for programming the device.

Assembly Language: An assembly language at its lowest level is a set of mnemonics for the machine code of a CPU. It may be augmented by macros and labels to ease the programming of the device.

BASIC Code: BASIC languages are a family of procedural programming languages that share a similar syntax derived from the original Dartmouth BASIC language, and are developed usually to be learned easily by people who are new to programming while also being easily to implement on the computer.

BeagleBone Black: BeagleBone Black is an open hardware, single-board computer and community-supported development platform for developers and hobbyists.

Binary File: A compiled executable (ready-to-run) file that contains data ready to be used by a program.

Bitwise: A shift operator that takes two operands and shift the first one a number of bits that is specified by the second operand. The operator used controls the direction of the shift.

Boot ROM: Boot ROM is a small piece of non-volatile memory that contains the executable code used by the computer system to initialize devices and load necessary constructs into memory on power-on.

Bus: A bus is a physical data connection shared between multiple different components or devices that are connected to a CPU.

C Language: A high-level and general purpose programming language that mainly uses logical and mathematical operators to develop firmware and portable applications.

C # Language: An object-oriented programming language that is based on C++ language, and it combines the computing power of C++ with Visual Basic programming.

Clock: A particular type of signal that behaves like a square wave. It has only two levels, which is zero (low) and one (high) where mostly the duration of a high level is the same as the zero level.

Clock (To Clock, Action): To open and close digital paths, allow or stop a process in timely manner. In general, clocking a digital circuit means provide timing for the circuit.

Control Unit: Establishes a method for the CPU to determine what information is stored in the data coming out of the memory.

CMOS: It stands for Complementary metal-oxide-semiconductor. It is a technology used for building digital logic and low power integrated circuits.

Compiler: A special program that translates statements in a high level programming language into machine language so that it can be executed.

Control Unit: Establishes a method for the CPU to determine what information is stored in the data coming out of the memory.

Cyclic Redundancy Check (CRC): CRC is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. Blocks of data entering these systems get a short check value attached, based on the remainder of a polynomial division of their contents.

Digital to Analog Converter (DAC): DAC is an electronic circuit consisting of Operational Amplifiers and common circuit elements such as resistor and capacitors that is used to convert an analog signal to a digital signal.

Debouncer Circuit: A debouncer circuit is a simple circuit that counts clock cycles and resets if the input changes while it is counting the clock cycles.

Demultiplexer: Demultiplexer is a circuit device (could be a chip) that has one input and more than one output. It takes a single input signal and selects one of many data output lines via another set of inputs. The amount of outputs is 2 to the power of the number of selection inputs. It is abbreviated as DEMUX.

Ethernet: Ethernet is a computer networking technology most commonly used for local area networks (LAN). It uses twisted pair copper cables as the physical media, has speeds between 10 megabits and 40 gigabits, and is standardized in IEEE 802.3.

FAT32: FAT32 is a simple file system based on a File Allocation Table, which is used to keep track of what files are allocated to what blocks on the hard drive. FAT12 was the original specification, and FAT32 extends this with directories and 32-bit addressing of blocks allowing for larger files and hard drives.

Flip-Flop: Flip Flop is device (chip), that is a circuit based on bipolar junction transistors that has two stable states and can be used to store state information. The circuit is described as sequential logic.

FPGA: Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing.

Graphical User Interface (GUI): It is a type of interface, which allows the user to interact with the computer through graphic elements instead of text characters.

Hard Drive: Hard Drive is a non-volatile data storage device used for storing and retrieving digital information based on a rotating magnetic platter.

Holding Register: A Holding register is a digital device that stores incoming data from a given data bus.

IDE Controller: IDE stands for Integrated Drive Electronics. It is the original standard of connecting to hard drives, originating the idea of standardizing the interface between the computer and the hard drive by including sufficient electronics on the hard drive to directly control the mechanical movement of the hard drive and only requiring the computer to implement the IDE messaging standard to communicate with a wide variety of hard drives.

Instruction Set: Also known as Instruction Set Architecture (ISA). It is a group of commands for a CPU in a machine language that a computer or a microprocessor understand.

Kit: Kit in the sense of this document refers to the discrete 4-bit computer kit that will be comprised of all the parts necessary to build the 4-bit computer.

Latch: A Latch is an array of D flip-flops that acts as a holding register that stores incoming data from the CPU's data bus. The Latch is used to ensure the correct data is sent down the bus at the correct time.

Logical Block Addressing (LBA): LBA is a common scheme used for specifying the location of blocks of data stored on a computer storage device.

Logic Gates: Logic Gates are electronic devices (chips) that make logical decisions based on the different combinations of digital signals present on its inputs. A device that executes a logical operation, on two or more inputs to produce one logical output.

Machine Code: Machine code is a sequence of machine instructions that is loaded into the memory of a CPU and executed by the CPU.

Macro Assembler: A higher level representation of an assembly language that includes structures and instructions that abstract away certain details of the lower level assembly language to ease the effort of programming the device.

Macro Instruction: A macro instruction is a group of assembly instructions that have been compressed into a simpler form and appear as a single instruction.

Microprocessor: An integrated circuit that contains all the functions of a CPU.

Microsoft's Visual Studio: An integrated development environment used to develop computer programs and applications**.**

Multiplexer: Multiplexer is a circuit device (could be a chip) that has greater inputs than outputs. It switches one of many inputs through a single common output by the application of control signals. The amount of inputs is 2 to the power of the control signals.

OP Code: A single machine language instruction that can be performed by the CPU.

Operational Amplifier: An Operational Amplifier, commonly referred as op-amp, is a voltage-amplifying device designed to use with external feedback components. Originally designed for analogue computers where they performed mathematical operations, op-amps are now popular for their versatility and high amplification potential.

Papilio One FPGA: The Papilio One FPGA is a brand of FPGA that is designed by Gadget Factory, which is an Open Source Hardware development community. It is an expandable development board for the design and prototyping of digital circuit projects. It has a Xilinx Spartan 3E FPGA chip to provide all digital logic required for the design.

Parser: A program that can receive its inputs in various forms and break them up into parts.

Parity Check: A parity check bit is a bit added to the end of a string of binary code that indicates whether the number of bits in the string with the value one is even or odd. Parity bits are used as the simplest form of error detecting code.

PC Register: The PC register is a 16-bit flip-flop. The output of the flip-flop is split and outputted to the memory selected mux and also to a 16-bit full adder that increments the PC by one.

Peripherals: In the context of this report, the term peripheral refers to five external devices which are interfacing with: the IDE Hard Drive, VGA Monitor, Ethernet Network, PS/2 Keyboard and Audio Speaker.

Potentiometer: A two or three terminal resistor with a sliding or rotating contact that can form an adjustable voltage divider or a variable resistor.

PS/2 Controller: PS/2 controller is used to communicate with a PS/2 keyboard. It provides an interface to the PS/2 protocol, handling the data transmission, error detection and the timing control.

PS/2 Keyboard: A PS/2 keyboard is a keyboard that uses the PS/2 serial interface to send the information on key presses from the keyboard to the PS/2 controller.

RAM: RAM stands for Random Access Memory. It is a type of computer memory that can be accessed randomly; that is, any cell of memory can be directly addressed at any time.

Raspberry Pi: Raspberry Pi is an ARM-based computer that usually is used with Linux. It is product designed to teach programming through do-it-yourself projects.

ROM: ROM stands for Read-Only Memory. Originally, these devices were memory devices where the data was set at construction time, and could not be altered; but now ROM devices are a family of non-volatile memory components that are usually defined by their use in being rarely written to.

Shift Register: In digital circuits, a shift register is a cascade of flip flops, sharing the same clock, in which the output of each flip-flop is connected to the input data of the next flip-flop in a chain, resulting in a circuit that shifts by one position.

Sound Card: The Personal Computer (PC) sound card is a removable computer expansion card, which, under the control of computer programs can input and output sound. The purpose of the sound card is to provide the audio for applications such as games, movies and music.

Strobe (Signal): To strobe the signal means to set the signal to logic low or 0 for a predetermined amount of time and then setting the signal back to logic high or 1.

Testbench: Testbenching refers to the method employed in VHDL programming to test and troubleshoot a design before building the actual circuit.

TTL Flip-Flop: Transistor-Transistor Logic Flip Flop is device (chip), that is a circuit based on bipolar junction transistors that has two stable states and can be used to store state information. The circuit is described as sequential logic.

TTL Logic Gates: Transistor-Transistor Logic Gates are electronic devices (chips) that are based on bipolar junction transistors and make logical decisions based on the different combinations of digital signals present on its inputs.

Tweet: A tweet is a posting made on the social media website Twitter.

Unary: A type of mathematical operator that has only one operand

User Space: User space is the portion of system memory in which processes run that directly interact with the user.

VGA: VGA is a standard for transmitting information for a monitor to display images based on an analogue signal representing color and a set of two clock signal that determine what pixel is being addressed.

VGA Controller: A VGA controller handles the low-level details of communicating with a monitor over the VGA protocol.

Virtual Machine (VM): It is a copy of a certain hardware or computer system that can be installed on a software that can imitates that system to give users the same experience as they would have on the actual system.

16. Appendix

16.1 Oscillator Circuit – Alternate Design

A 16.0 kHz frequency is needed for the oscillator to ensure all 16 bits are properly passed into the counter. Vcc is connected to the +5.0 V.

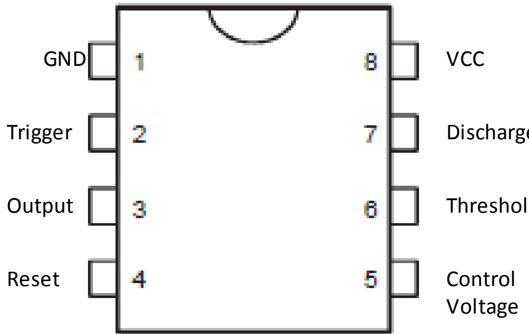


Figure 16.1.1: Appendix A1 Top View of SA555N Timer Chip

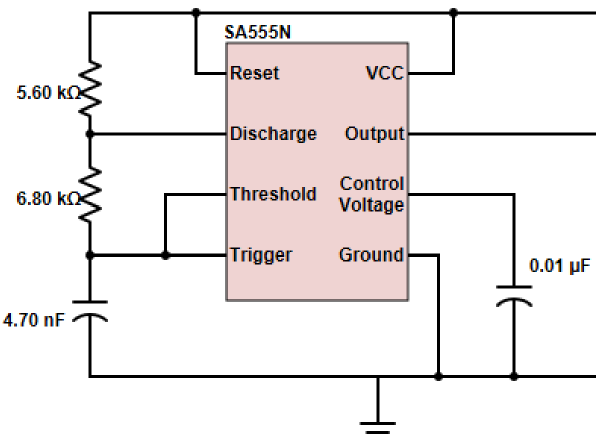


Figure 16.1.2: Appendix A1 Timer Circuit Connections

The counter is a CD4520. It is a 4-bit dual up counter so only one side of the counter integrated circuit is necessary. The figure on the left shows the green grounded pins. Since this circuit is also a digital circuit, VDD is powered at +5.00 V. VSS is grounded. Q1-Q4 pins are connected to the d0-d3 pins on the 4508 latch. As shown below, enable is connected to +5.00 V, while reset is connected to the ground. The output from the 555 timer integrated circuit is connected to the clock of the counter.

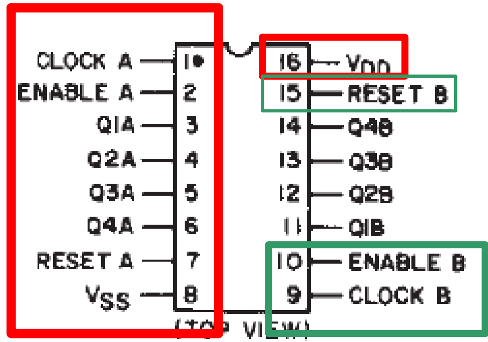


Figure 16.1.3: Appendix A1 Top View of CD4520BE Counter

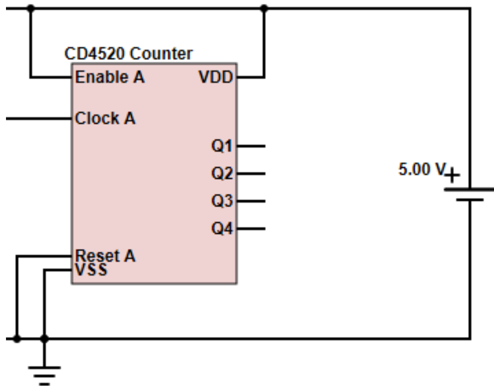


Figure 16.1.4: Appendix A1 CD4520BE Counter Connections

Complete Oscillator Circuit

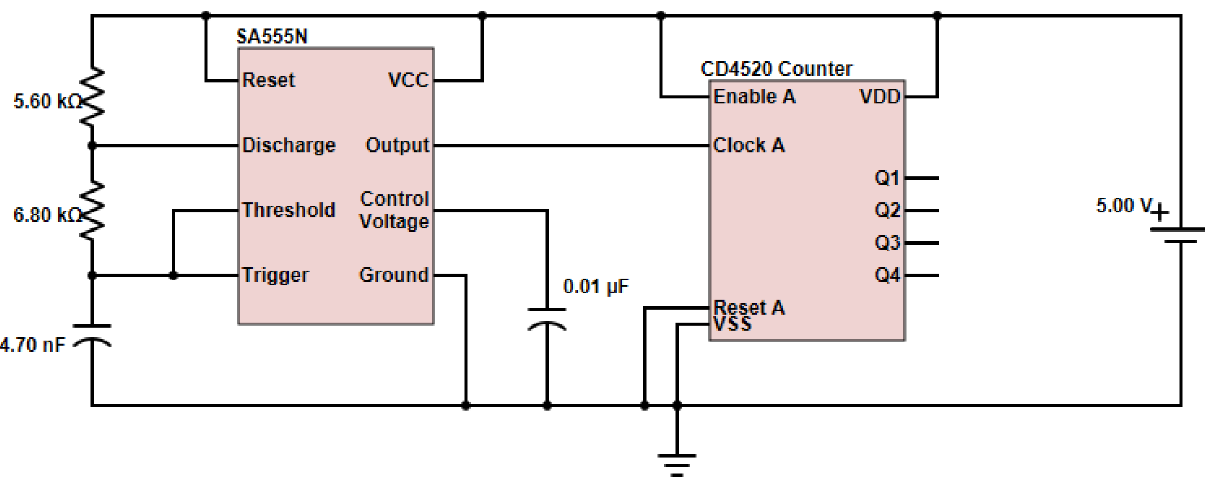


Figure 16.1.5: Appendix A1 Oscillator Circuit

Table Four: Truth Table for the CD4520BE

Clock	Enable	Reset	Action
↑	1	0	Increment counter
0	↓	0	Increment counter
↓	X	0	No change
X	↑	0	No change
↑	0	0	No change
1	↓	0	No change
X	X	1	Q1-Q4 = 0 (low state)

Table 16.1.1: Appendix A1 Truth Table for CD4520BE Chip

16.2 IDE Controller – Arduino Test Code

Arduino Code for Testing

```
short int R = 9;
short int W = 8;
short int CS = 7;
short int data[4] = {2,3,4,5}; //5 is d0
int period = 1;
char value;
void setup() {
  // Set R as output
  pinMode(R,OUTPUT);
  pinMode(W, OUTPUT);
  pinMode(CS, OUTPUT);
  pinMode(10, INPUT);

  digitalWrite(CS, LOW);
  digitalWrite(R, LOW);
  digitalWrite(W, LOW);
  digitalWrite(CS, HIGH);
  digitalWrite(R, HIGH);
  delay(period);
  digitalWrite(R, LOW);
  delay(period);
  digitalWrite(CS, LOW);
  //Set data as output to begin
  for(int i = 0; i < 4; i ++){
    pinMode(data[i], OUTPUT);
  }
  Serial.begin(9600);
}
void loop() {

//    readHD(0xF);
//    readHD(0xF);
//
//  //Set up Write
//  Serial.println("/-------------Begining of Write to drive-------------/\nSetup:");
//  writeHD(0xA, 0, 0, 0, 1);
//  writeHD(0xB, 0, 0, 0, 0);
//  writeHD(0xC, 0, 0, 0, 0);
//  writeHD(0xD, 0, 0, 0, 0);
//  writeHD(0xE, 0, 0, 0xE, 0);
//  Serial.println("\nSending write command:");
//  writeHD(0xF, 0, 0, 0x3, 0x0);
//
```

```
//  Serial.println("\n/-----------------Writing to HD-----------------/\n");
//
//  for(int i = 0; i < 64; i++){
//    writeHD(0x8, 0x4, 0x9, 0x4, 0x8);
//    writeHD(0x8, 0x4, 0x2, 0x2, 0x0);
//    writeHD(0x8, 0x4, 0xE, 0x4, 0x5);
//    writeHD(0x8, 0x2, 0x0, 0x2, 0x0);
//  }
//  Serial.println("\n\n/-------------Finished writing to HD-----------/\n");
//
//  for(int i = 0; i < 5; i ++){
//    readHD(0xF);
//  }

  Serial.println("/-------------Begining of Read from drive----------------/\nSetup:");
  writeHD(0xA, 0, 0, 0, 1);
  writeHD(0xB, 0, 0, 0, 0);
  writeHD(0xC, 0, 0, 0, 0);
  writeHD(0xD, 0, 0, 0, 0);
  writeHD(0xE, 0, 0, B1110, 0);

  for(int i = 0; i < 5; i ++){
    readHD(0xF);
  }

  Serial.println("\n/----------Sending read command----------/");
  writeHD(0xF, 0x0, 0x0, 0x2, 0x0);

  Serial.println("\n/-----------------Reading from HD-----------------/\n");
  for(int i = 0; i < 5; i ++){
    readHD(0xF);
  }

  for(int i = 0; i < 256; i++){
    readHD(0x8);

//    Serial.println("/--------------Sending Value to Sound-------------/");
//    digitalWrite(CS, HIGH);
//    digitalWrite(W, HIGH);
//    writeData(value);
//    digitalWrite(W, LOW);
//    digitalWrite(CS, LOW);
//    Serial.println("/-------------Finished Writing to Sound-------------/");
  }

  Serial.println("\n\n/-------------Finished Reading from HD-----------/\n");
```

```
  for(int i = 0; i < 5; i ++){
    readHD(0xF);
  }

  readHD(0xA);
  readHD(0xB);
  readHD(0xC);
  readHD(0xD);
  readHD(0xE);
  readHD(0x9);

  while(1){

  }
}

//Reads data from the 4 bit data lines
void readData(){
  //Set data as input to read
  for(int i = 0; i < 4; i ++){
    pinMode(data[i], INPUT);
  }

  char c = 0;

  for(int i = 0; i < 4; i++){
    if(digitalRead(data[i]) == HIGH)
    {
      c |= (1 << (3-i));
      //Serial.print("1");
    }
    //else
      //Serial.print("0");
  }
  Serial.print(c, HEX);
  Serial.print(" ");

  value = c;
}


//Writes data to the 4 bit data lines
void writeData(int val){
    //Set data as output to write
  for(int i = 0; i < 4; i ++){
```

```
    pinMode(data[i], OUTPUT);
  }

  int check = 1;
  for(int count = 3; count >= 0; count--){
    if((val & check) == check)
      digitalWrite(data[count], HIGH);
    else
      digitalWrite(data[count], LOW);
    check = check << 1;
  }

}

//Complete write to a HD register
void writeHD(int cmd, int d3, int d2, int d1, int d0){

  Serial.print("Writing: 0x");
  Serial.print(d3, HEX);
  Serial.print(d2, HEX);
  Serial.print(d1, HEX);
  Serial.print(d0, HEX);
  Serial.print(" to register: 0x");
  Serial.print(cmd, HEX);
  Serial.print("\n");

 // digitalWrite(CS, LOW);
  delay(period);

  digitalWrite(W, HIGH);
  delay(period);
  writeData(cmd);
  delay(period);
  digitalWrite(W, LOW);
  delay(period);

  digitalWrite(W, HIGH);
  delay(period);
  writeData(d3);
  delay(period);
  digitalWrite(W, LOW);
  delay(period);

  digitalWrite(W, HIGH);
  delay(period);
  writeData(d2);
```

```
  delay(period);
  digitalWrite(W, LOW);
  delay(period);

  digitalWrite(W, HIGH);
  delay(period);
  writeData(d1);
  delay(period);
  digitalWrite(W, LOW);
  delay(period);

  digitalWrite(W, HIGH);
  delay(period);
  writeData(d0);
  delay(period);
  digitalWrite(W, LOW);
  delay(period);

  digitalWrite(W, HIGH);
  delay(period);
  digitalWrite(W, LOW);
  delay(period);

  digitalWrite(W, HIGH);
  delay(period);
  digitalWrite(W, LOW);
  delay(period);

  digitalWrite(W, HIGH);
  delay(period);
  digitalWrite(W, LOW);
  delay(period);

  digitalWrite(W, HIGH);
  delay(period);
  digitalWrite(W, LOW);
  delay(period);

 // digitalWrite(CS, HIGH);
}

//Complete Read of a HD register
void readHD(int cmd){

  Serial.print("Reading from register: 0x");
  Serial.print(cmd, HEX);
```

```
 Serial.print("  Data read: ");

// digitalWrite(CS, LOW);
 delay(period);

 digitalWrite(W, HIGH);
 delay(period);
 writeData(cmd);
 delay(period);
 digitalWrite(W, LOW);
 delay(period);

 digitalWrite(R, HIGH);
 delay(period);
 digitalWrite(R, LOW);
 delay(period);

 digitalWrite(R, HIGH);
 delay(period);
 digitalWrite(R, LOW);
 delay(period);

 digitalWrite(R, HIGH);
 delay(period);
 digitalWrite(R, LOW);
 delay(period);

 digitalWrite(R, HIGH);
 delay(period);
 digitalWrite(R, LOW);
 delay(period);

 digitalWrite(R, HIGH);
 delay(period);
 readData();
 digitalWrite(R, LOW);
 delay(period);

 digitalWrite(R, HIGH);
 delay(period);
 readData();
 digitalWrite(R, LOW);
 delay(period);

 digitalWrite(R, HIGH);
 delay(period);
```

```
  readData();
  digitalWrite(R, LOW);
  delay(period);

  digitalWrite(R, HIGH);
  delay(period);
  readData();
  digitalWrite(R, LOW);
  delay(period);

  Serial.print("\n");
//  digitalWrite(CS, HIGH);
}
```

Note: There is extensive amount of code that has been developed to design the Nibble Knowledge Computer. All of the code for different aspects of the IDE Controller is located in an online, web-based Git repository hosting service. The Code is accessible through the link below:

https://github.com/Nibble-Knowledge

16.3 PS/2 Keyboard Controller VHDL Code: Main

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity keyboard is
	PORT (
		-- CPU (bus) inputs/outputs
		clk: in STD_LOGIC; -- CPU 5 MHz clock
		cpu_read:in STD_LOGIC; -- Needs to be logic high in order for the cpu to read
from the PS/2 Controller
		cpu_write:in STD_LOGIC;	-- Unused
		parity_check:in STD_LOGIC;		-- Dealt with by the Parity check circuit
		chip_select:in STD_LOGIC; -- Needs to be logic high (PS/2 Controller selected by
CPU) in order for the CPU to read data from the PS/2 Controller
		ready: out STD_LOGIC := '0'; -- Goes to logic high when data is ready to be sent
to CPU
		nibble: out STD_LOGIC_VECTOR(3 downto 0); -- 4 bit data line from PS/2
Controller to CPU

		-- DC Power Source & Ground is also wired to the Keyboard Controller

		-- Keyboard inputs
		ps2_clk: in STD_LOGIC; -- PS/2 Keyboard Clock signal
		ps2_data: in STD_LOGIC; -- PS/2 Keyboard Data signal

		-- Reset input
		reset: in STD_LOGIC
	);
end keyboard;

architecture Behavioral of keyboard is

-- PS/2 CLK Rising Edge Determination:
signal ps2_clk_dffs: STD_LOGIC_VECTOR(1 downto 0);

-- PS/2 DATA signals:
signal ps2_word: STD_LOGIC_VECTOR(10 downto 0);
signal ps2_code: STD_LOGIC_VECTOR(7 downto 0);
signal ps2_select: STD_LOGIC := '0'; -- Default to high nibble

-- PS/2 IDLE signals:
signal high_nibble_ready: std_logic := '0'; -- High nibble is ready to be read by CPU
signal high_nibble_out: std_logic := '0'; -- High nibble has been passed through 8x4 Mux
signal low_nibble_ready: std_logic := '0'; -- Low nibble is ready to be read by CPU
```

```
signal low_nibble_out: std_logic := '0'; -- Low nibble has been passed through 8x4 Mux
signal end_of_trans: std_logic := '0'; -- transmission is completed
signal bit_counter: integer range 0 to 11;
signal bits_shifted_in: std_logic := '0';

-- DCM 32 to 5 MHz (Digital Clock Manager - reducing clock speed from 32 to 5 MHz)
signal bf_clk: std_logic;

-- Declare 8 to 4 mux component

component eight_to_four_mux is
	port(
		high_nibble : in std_logic_vector (3 downto 0);
		low_nibble : in std_logic_vector (3 downto 0);
		s	: in	std_logic;
		out_nibble : out std_logic_vector (3 downto 0);
		no_data : in std_logic
		);
end component;

--Declare Digital clock manager (32 MHz to 5 Mhz)

COMPONENT DCM_32to5
	PORT(
		CLKIN_IN : IN std_logic;
		CLKFX_OUT : OUT std_logic;
		CLKIN_IBUFG_OUT : OUT std_logic;
		CLK0_OUT : OUT std_logic
		);
	END COMPONENT;

begin

-- Instantiate the 8 to 4 Mux component

NIBBLE_SELECTOR: eight_to_four_mux
	port map (
		high_nibble => ps2_code(7 downto 4),
		low_nibble => ps2_code(3 downto 0),
		s	=> ps2_select,
		no_data => chip_select,
		out_nibble => nibble);

--Instantiate the Digital Clock Manager

Inst_DCM_32to5: DCM_32to5
```

```
	PORT MAP(
		CLKIN_IN => clk,
		CLKFX_OUT => bf_clk,
		CLKIN_IBUFG_OUT => open,
		CLK0_OUT => open
	);

-- Determine end of transmission and output results in two - four bit nibbles
process(bf_clk)
begin
	if (reset = '1') then
		ready <= '0';
		low_nibble_out <= '0';
		high_nibble_out <= '0';
		low_nibble_ready <= '0';
		high_nibble_ready <= '0';
		end_of_trans <= '0';
		ps2_clk_dffs <= "00";
		ps2_word <= "00000000000";
		bit_counter <= 0;
		bits_shifted_in <= '0';
	elsif (rising_edge(bf_clk)) then

-- Syncronizing circuitry (shift serial bits into 11 bit shift register on rising edge of PS2 clock signal)

		if (bits_shifted_in = '0') then -- if all 11 bits haven't been shifted in, continue to shift
		ps2_clk_dffs(0) <= ps2_clk;
		ps2_clk_dffs(1) <= ps2_clk_dffs(0);
		end if;

-- Detecting end of transmission counter and 11 bit serial in, 8 bit parallel out shift register

		if ((ps2_clk_dffs(0) = '0') and (ps2_clk_dffs(1) = '1') and bit_counter /= 11) then
			ps2_word <= ps2_data & ps2_word(10 downto 1);
			bit_counter <= bit_counter + 1;
		elsif (bit_counter = 11 and end_of_trans = '0' and high_nibble_ready <= '0') then
			ps2_code <= ps2_word(8 downto 1);
			high_nibble_ready <= '1';
			ready <= '1'; -- Data is available on PS/2 keyboard controller peripheral (CPU can read in two nibbles)
		end if;

-- Split 8 bit word into 2 - 4 bit nibbles and control the ready line

		-- High nibble is placed on bus and CPU is free to read data
```

```
		if (cpu_read = '1' and chip_select = '0' and parity_check = '1' and high_nibble_ready
= '1' and high_nibble_out <= '0') then
			ps2_select <= '1';
			high_nibble_out <= '1';
		end if;

		-- CPU sets it's 'read' signal to zero, high nibble read complete
		if (cpu_read = '0' and parity_check = '1' and high_nibble_out = '1' and
low_nibble_ready <= '0') then
			low_nibble_ready <= '1';
		end if;

		-- Low nibble is placed on bus and CPU is free to read data
		if (cpu_read = '1' and chip_select = '0' and parity_check = '1' and low_nibble_ready
= '1' and low_nibble_out = '0') then
			ps2_select <= '0';
			low_nibble_out <= '1';
		end if;
		-- CPU sets it's 'read' signal to zero, low nibble read complete
		if (cpu_read = '0' and parity_check = '1' and low_nibble_out = '1' and end_of_trans
<= '0') then
			end_of_trans <= '1';
			ready <= '0'; -- One make code has successfully been read by CPU
		end if;
		-- PS/2 keyboard controller resets all DFF's and prepares for new data to be sent
from PS/2 keyboard
		if (end_of_trans = '1') then
			low_nibble_out <= '0';
			high_nibble_out <= '0';
			low_nibble_ready <= '0';
			high_nibble_ready <= '0';
			end_of_trans <= '0';
			bit_counter <= 0;
			bits_shifted_in <= '0';
		end if;
end if;
end process;
end Behavioral;
```

Note: There is extensive amount of code that has been developed to design the Nibble Knowledge Computer. All of the code for different aspects of the PS/2 Keyboard Controller is located in an online, web-based Git repository hosting service. The Code is accessible through the link below:

https://github.com/Nibble-Knowledge

16.4 PS/2 Keyboard Controller – Internal Multiplexer VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity eight_to_four_mux is

 port(
		high_nibble : in std_logic_vector (3 downto 0);
		low_nibble : in std_logic_vector (3 downto 0);
		s	: in	std_logic;
		no_data : in std_logic;
		out_nibble : out std_logic_vector (3 downto 0)
		);

end eight_to_four_mux;

architecture Behavioral of eight_to_four_mux is

begin
process(s, no_data)
begin
	if (no_data = '1') then
		out_nibble <= "ZZZZ"; -- no data put on data lines on bus (i.e. make code has been read or no key has been pressed on the PS/2 keyboard)
	else
		if (s = '1') then
			out_nibble <= high_nibble; -- multiplexer selects the high nibble first (i.e. bits 8-5 from 11 bit code sent to shift register)
		else
			out_nibble <= low_nibble; -- multiplexer selects the low nibble second (i.e. bits 4-1 from 11 bit code sent to shift register)
		end if;
	end if;

end process;
end Behavioral;
```

Note: There is extensive amount of code that has been developed to design the Nibble Knowledge Computer. All of the code for different aspects of the PS/2 Keyboard Controller is located in an online, web-based Git repository hosting service. The Code is accessible through the link below:

https://github.com/Nibble-Knowledge

16.5 Software – Complete CUTE Basic Compiler

The Cute BASIC Compiler takes a Cute BASIC file as an input and outputs a macro assembly file. It parses through the CB file, line by line, and writes the appropriate macro instructions to a buffer. Each line is checked for certain keywords until a match is found; when matched the program loads a function appropriate for whatever keyword was found, translates the current line into a series of macro instructions, and then goes to the next line. The macro is written to in the same order as it is in the CB file, the only exception is for variables which are stored to a separate buffer as they must be stored at the bottom of a macro file. When the end of the file is reached the instruction buffer is output to the macro file, followed by the variable buffer.

```
#!/usr/bin/python
import sys
import os

#stack for keeping track of IF statements - holds line numbers of IF
ifStatementStack = []
endifStack = []
#stack for keeping track of LOOPWHILE statements - holds like numbers of LOOPWHILE
loopStatementStack = []
#giant buffer holding output
output = []
data = []
#list of names of variables

#format: VARNAME SIZE SIGN
variables = []

varnames = []
varsizes = []
varvals = []

funcnames = []
funcstack = []
returns = 0

signed = []
files = []

funcVars = []

baseaddress = -1
```

```
identifiers = ["START" , "END", "SIGNED", "LET", "BE", "AS", "COMPLEMENT",
"NEGATE", "INCREMENT", "DECREMENT",
"NOT", "ADDRESSOF", "CONTENTOF", "IF", "ELSEIF", "ELSE", "ENDIF",
"LOOPWHILE", "ENDLOOP", "LOOPAGAIN", "EXITLOOP",
"EXITIF", "EARLYEND", "LABEL", "GOTO", "ADD", "SUBTRACT", "MODULUS",
"MULTIPLY", "DIVIDE", "ALSO", "EITHER", "AND",
"OR", "XOR", "NAND", "NOR", "XNOR", "LSHIFT", "RSHIFT", "LROTATE", "RROTATE",
"ADDRESSOF", "CONTENTOF", "FUNCTION",
"ENDFUNCTION", "RETURN", "RETURNS", "TAKES", "ASWELL"]


#Added by Naomi to allow specific static pointers to be used
validPointers = ["*CHIP_SELECT", "*STATUS_BUS", "*DATA_BUS"]

#Keeps track of which files have been included
included = []

#flags
hasstart = 0
hasend = 0
multicomment = 0
asmFunc = 0



#Checks the nibble length of the variable
#Takes name of variable
#Returns length of variable
def varLength(varname):
	if varname in varnames:
		index = varnames.index(varname)

		if(len(varsizes) > index):
			length = varsizes[index]
		#Assuming literals should be 4?
		else:
		#if length == None:
			length = 4
		return int(length)


#Check the length of two lines and return the difference between them (-# means var 1 is smaller
by # nibs, 0 means even, +# means var 1 is bigger by # nibs)
#Takes 2 variables
#Returns an integer (differnece in size)
def compareLength(varname1, varname2):
```

```
	if(varname1 in varnames and varname2 in varnames):
		length1 = varLength(varname1)
		length2 = varLength(varname2)
		return length1 - length2

#Takes two variables
#Returns the length of the longer variable
def longer(var1, var2):
	longer = varLength(var1)
	if longer == None:
		longer = 4

	longer2 = varLength(var2)
	if longer2 == None:
		longer2 = 4

	if longer < longer2:
		longer = longer2
	return longer

#Checks if a value is a variable or literal, and adds the N_[] if its a literal
#Takes a potential variable
#Returns the same var, or an encased literal
def literal(var):
	if var not in varnames:
		var = "N_[" + var + "]"
	return var

#Writes the ASM code needed to perform signed or unsigned extension depending on variable
#Takes a variable and a size to extend to
#Returns the extended value label
def extend(var, difference):
	#Number of nibbles needed to extend by
	size = int(difference)
	cnt = 0
	length = varLength(var)
	total = length + difference
	#Sign extend
	if var in signed:
		extendedVar = var + str(total*4) + "signed"
		if(extendedVar not in varnames):
			data.append(var + str(total) + "signed: .data " + str(total) + " 0")
			varnames.append(extendedVar)
```

```
			op = multinibOp(length, "MOV")
			output.append(op +  " " + var + " INTO " + extendedVar + "[" + str((total - length
)) + "]")

			#TODO Not sure if im using signex right
			output.append("LOD " + var)
			output.append("SIGNEX ACC")

			#output.append(op + " ") Just taking this out for now, it messes up stuff.

	#Unsigned extend
	else:
		extendedVar = var + str(total) + "unsigned"
		if(extendedVar not in varnames):
			data.append(var + str(total) + "unsigned: .data " + str(total) + " 0")
			varnames.append(extendedVar)

		op = multinibOp(length, "MOV")
		output.append(op +  " " + var + " INTO " + extendedVar  + "[" + str((total - length
)) + "]")

		output.append("LOD N_[0]")

	#Fill the remaining values with the extended value

	#Wrong endianness
	#cnt = 0
	#while cnt < size:

	cnt = size - 1
	while cnt >= 0:
		output.append("STR "+ extendedVar + "[" + str(cnt) + "]")
		cnt -= 1

	return extendedVar

#Takes an operation and a size and returns the oepration with that size included
def multinibOp(length, op):
	if str(length) == "1":
		return op
	else:
		return op + str(int(length)*4)
		#return op + str(int(length))

#Checks if we are in a function, if we are change all labels to be the local version of the label
#Takes the splitline
```

```
#Returns: the unaltered splitline or the splitline with function names appended
def localVars(splitline):
	if len(funcstack) > 0:
		cnt = 0
		#Check each word in splitline
		for word in splitline:
			num = word.isdigit()
			#If word is not an identifier or a number it is a label - therefore add funcname_ before it
			if word not in identifiers and num == False:
				splitline[cnt] = funcstack[-1] + "_" + word
			cnt += 1
	return splitline



#Checks if a comment exists on the line
#Takes: a line from readlines()
#Returns: 1 if a comment is found, 0 otherwise
def checkComment(line):
	global multicomment
	continueBool = 0
	if len(line) > 1 and (line.find("#/") != -1):
		multicomment = 0
		continueBool = 1

	elif multicomment == 1:
		continueBool = 1

	elif len(line.strip()) > 1 and line.strip()[0] == "/" and line.strip()[1] == "#":
		multicomment = 1
		continueBool = 1

	elif len(line.strip()) > 0 and line.strip()[0] == "#":
		continueBool = 1
	return continueBool


#Checks if line contains start or end
#Takes: a line
def startEnd(line):
	cont = 0
	global hasstart
	global hasend

	if line.strip() == "START":
```

```
		if hasstart == 1:
			print "START already declared"
			exit(1)
		else:
			hasstart = 1
			output.append("START:")

		cont = 1

	elif line.strip() == "END":
		if hasend == 1:
			print "END already declared"
			exit(1)
		else:
			hasend = 1
			output.append("END:")
		cont = 1
	return cont


#Checks if a line is asm or passthroc, if it is put it directly into output as is
#Takes: a splitline
#Returns: 1 if asm or passthroc is found, 0 otherwise
def asm_passthroc(splitline, linenum):
	global asmFunc
	cont = 0
	if splitline[0] == "ASM" or splitline[0] == "PASSTHROC" or asmFunc == 1:
		#If its a special variable function type
		if (len(splitline) > 1 and splitline[1] == "FUNCTION") or asmFunc == 1:
			asmFunction(splitline, linenum)
		else:
			del splitline[0]
			newline = ""
			for i in splitline:
				newline += i
				newline += " "
			output.append(newline)
		cont = 1

	return cont


#Checks if variable declaration is being performed
#Takes: a splitline
def variableDeclaration(splitline, linenum):
	cont = 0
```

```
	if splitline[0] == "LET":
		#LET VAR - this should default a size and value
		#LET VAR AS SIZE- this should default a value
		#LET VAR BE VAL (SIGNED)- this should default a size, and if no signed default
unsigned
		#LET VAR AS SIZE BE VAL (SIGNED) - if no signed, default unsigned

		if splitline[1] in identifiers:
			print "Error line " + str(linenum) + ": can not use identifier " + splitline[1]
+ " as a variable name"


		#Disallow variable names starting with '*' unless we have them in our pointer list
(at top of file)
		#Also, don't allow pointers to have non-default size, or any initialisation value.
		if splitline[1][0] == '*' and (splitline[1] not in validPointers or len(splitline) > 2):
			print "Error line " + str(linenum) + ": special character * can only be used
with valid static labels"

		#Check if variable already exists
		for i in varnames:
			if (len(splitline) > 1 and i == splitline[1] ):
				if(len(funcstack) > 0 and funcstack[-1] + "_" + splitline[1] not in
varnames):
					a = 0 #do nothing
				else:
					print "Error line " + str(linenum) + ": Variable name " +
splitline[1] + " already declared!"
					quit(1)


		# The variable name will be the second whitespace delimited field

		varnames.append(splitline[1])
		#print varnames
		temp = ''
		temp += splitline[1]

		defaultSize = 4 #4 nibble default
		defaultVal = 0

		#LET VAR
		if len(splitline) == 2:
			temp +=": .data " + str(defaultSize) + " " + str(defaultVal)
			varsizes.append(defaultSize)
```

```
		#LET VAR AS or LET VAR AS BE
		elif splitline[2] == "AS":
			#LET VAR AS
			#Check that valid nib size
			if (splitline[3] not in ["1", "2", "4", "8", "16"]):
				print "Error on line: " + str(linenum) + ": Invalid size in nibbles, size must be 1, 2, 4, 8 or 16"
				exit(1)

			if len(splitline) == 4:
				temp += ": .data " + splitline[3] + " " + str(defaultVal)
				varsizes.append(splitline[3])

			#LET VAR AS BE
			else:
				if(len(splitline) > 5 and splitline[5][0] == "\""):
					temp += ": .ascii " + splitline[5]
					varsizes.append(splitline[3])
				else:
					temp += ": .data " + splitline[3] + " " + splitline[5]
					varsizes.append(splitline[3])

		#LET VAR BE
		elif splitline[2] == "BE":
			if(splitline[3][0] == "\""):
				temp += ": .ascii " + splitline[3]
				varsizes.append(len(splitline[3]) * 8) #TODO not sure if this value is correct
			else:
				temp += ": .data " + str(defaultSize) + " " + splitline[3]
				varsizes.append(defaultSize)

		#Invalid input
		else:
			print "Error on line " + str(linenum) + ": Invalid variable declaration " + str(splitline)
			exit(1)
		#If last input is signed	#TODO figure out what this should do
		if splitline[-1] == "SIGNED":
			signed.append(splitline[1])

		#Add the data declaration - unless it's a pointer!
		if (splitline[1] not in validPointers):
			data.append(temp)
		cont = 1
	return cont
```

```
#Checks if unary math operations are being performed with assignments as the first thing on the line
#Takes: a splitline
def unarymath(splitline, linenum):
	cont = 0
	if (len(splitline) > 3 and (splitline[0] in varnames) and (splitline[2] in ["COMPLEMENT", "NEGATE", "INCREMENT", "DECREMENT", "NOT", "ADDRESSOF", "CONTENTOF"])):

		length = varLength(splitline[3])
		minnib = 1
		suffix = str(length*4)
		#unary math/bit/logic operators
		if splitline[2] == "COMPLEMENT":
			if length == minnib:
				output.append("NOT " + splitline[3] + " INTO " + splitline[0])
			else:
				output.append("NOT" + suffix + " " + splitline[3] + " INTO " + splitline[0])

		elif splitline[2] == "NEGATE":
			if length == minnib:
				output.append("NEG " + splitline[3] + " INTO " + splitline[0])
			else:
				output.append("NEG" + suffix + " " + splitline[3] + " INTO " + splitline[0])

		elif splitline[2] == "INCREMENT":
			if length == minnib:
				output.append("INC " + splitline[3] + " INTO " + splitline[0])
			else:
				output.append("INC" + suffix + " " + splitline[3] + " INTO " + splitline[0])

		elif splitline[2] == "DECREMENT":
			if length == minnib:
				output.append("SUB " + splitline[3] + " N_[1] INTO " + splitline[0])
			else:
				output.append("SUB" + suffix + " " + splitline[3] + " N_[1] INTO " + splitline[0])


		elif splitline[2] == "NOT":
			output.append("LOGNOT ACC")
			output.append("MOV " + splitline[3] + " INTO " + splitline[0])
```

```
				#TODO Dont think this is right but have to double check - only accumualtor
lognot?

			elif splitline[2] == "ADDRESSOF":
				output.append("MOVADDR "+ splitline[3] +" INTO " + splitline[0])
				#TODO not sure if these parts are right
			elif splitline[2] == "CONTENTOF":
				if length == minnib:
					output.append("MOV " + splitline[3] + " INTO " + splitline[0])
				else:
					output.append("MOV" + suffix + " " + splitline[3] + " INTO " +
splitline[0])
			cont = 1
	return cont

#Checks if binary math operations are being performed with assignments as the first thing on the
line
#Takes: a splitline, linenumber
def binarymath(splitline, linenum):
	cont = 0
	if ((splitline[0] in varnames) and len(splitline) > 4 and splitline[2] != "CALL"):
		lengthDif = compareLength(splitline[2], splitline[4])

		#Get length of longer value
		length = longer(splitline[2], splitline[4])
		#print str(length) + "\n"
		#Lengths are the same or a val is a literal
		if lengthDif == 0 or lengthDif == None:
			var1 = literal(splitline[2])
			var2 = literal(splitline[4])

		#Extends the first or second value as needed
		#first var is smaller
		elif lengthDif < 0:
			diff = -lengthDif
			var1 = extend(splitline[2], diff)
			var2 = splitline[4]

		#First var is larger
		elif lengthDif > 0:
			diff = lengthDif
			var1 = splitline[2]
			var2 = extend(splitline[4], diff)


		minnib = 1
```

```
		suffix = str(length*4)

		count = 0
		if splitline[3] == "ADD":
			if length == minnib:
				output.append("ADD " + var1 + " " + var2 + " INTO " + splitline[0])
			else:
				output.append("ADD" + suffix + " " + var1 + " " + var2 + " INTO "
+ splitline[0])

		elif splitline[3] == "SUBTRACT":
			if length == minnib:
				output.append("SUB " + var1 + " " + var2 + " INTO " + splitline[0])
			else:
				output.append("SUB" + suffix + " " + var1 + " " + var2 + " INTO "
+ splitline[0])

		elif splitline[3] == "MULTIPLY":
			#output.append("MULT " + var1 + " " + var2 + " INTO " + splitline[0])
			if length == 16:
				print "Error On line " + str(linenum) + ": Multiplication can not
handle larger than 32 bit variables."
			multiply(var1, var2, splitline[0], length, linenum)
		elif splitline[3] == "DIVIDE":
			output.append("DIV " + var1 + " " + var2 + " INTO " + splitline[0]) #TODO
This doesnt work yet
		elif splitline[3] == "MODULUS":
			output.append("MOD " + var1 + " " + var2 + " INTO " + splitline[0])
#TODO this doesnt work yet

		#binary bitwise operators
		elif splitline[3] == "AND":
			if length == minnib:
				output.append("AND " + var1 + " " + var2 + " INTO " + splitline[0])
			else:
				output.append("AND" + suffix + " " + var1 + " " + var2 + " INTO "
+ splitline[0])

		elif splitline[3] == "NAND":
			if length == minnib:
				output.append("NAND " + var1 + " " + var2 + " INTO " +
splitline[0])
			else:
				output.append("NAND" + suffix + " " + var1 + " " + var2 + " INTO
" + splitline[0])
```

```
		elif splitline[3] == "OR":
			if length == minnib:
				output.append("OR " + var1 + " " + var2 + " INTO " + splitline[0])
			else:
				output.append("OR" + suffix + " " + var1 + " " + var2 + " INTO " +
splitline[0])

		elif splitline[3] == "NOR":
			if length == minnib:
				output.append("NOR " + var1 + " " + var2 + " INTO " + splitline[0])
			else:
				output.append("NOR" + suffix + " " + var1 + " " + var2 + " INTO "
+ splitline[0])

		elif splitline[3] == "XOR":
			if length == minnib:
				output.append("XOR " + var1 + " " + var2 + " INTO " + splitline[0])
			else:
				output.append("XOR" + suffix + " " + var1 + " " + var2 + " INTO "
+ splitline[0])

		elif splitline[3] == "XNOR":
			if length == minnib:
				output.append("XNOR " + var1 + " " + var2 + " INTO " +
splitline[0])
			else:
				output.append("XNOR" + suffix + " " + var1 + " " + var2 + " INTO
" + splitline[0])

		#TODO Not sure if this is right, or how to do right shifts/rots at all
		#binary bitshifting
		elif (splitline[3] == "RROTATE"):
			a = ''
			#RROT [label1] into [label2]
		elif (splitline[3] == "LROTATE" ):
			while (count < int(splitline[4])) :
				output.append("LROT" + suffix + " "+ var1 + " INTO " +
splitline[0])
				count += 1
			#RROT [label1] into [label2]

		elif (splitline[3] == "RSHIFT"):
			a = ''
		elif (splitline[3] == "LSHIFT"):
			while (count < int(splitline[4])) :
```

```
                output.append("LSHIFT" + suffix + " "+ var1 + " INTO " + 
splitline[0])
                count += 1

        #binary logical operators
        elif splitline[3] == "ALSO":
            #Not sure if this is correct or not TODO (for both also + either)
            #Check if first val = 0 - if it is jump to fail
            output.append("JMPEQ N_[0] " + var1 + " TO FALSE" + str(linenum))
            #check second val = 0, if it is jump fail
            output.append("JMPEQ N_[0] " + var2 + " TO FALSE" + str(linenum))
            #If made it here both are non0 so true
            output.append("MOV N_[1] INTO " + splitline[0])
            output.append("LOD N_[0]")
            output.append("JMP ENDALSO" + str(linenum))
            #Go here if are false
            output.append("FALSE" + str(linenum) + ":")
            output.append("MOV N_[0] INTO " + splitline[0])
            output.append("ENDALSO" + str(linenum) + ":")

        elif splitline[3] == "EITHER":
            #Check if first val is not 0 - if it is jump to true
            output.append("JMPNE N_[0] " + var1 + " TO TRUE" + str(linenum))
            #check second val = 0, if it is jump fail
            output.append("JMPNE N_[0] " + var2 + " TO TRUE" + str(linenum))
            #If made it here both are 0 so false
            output.append("MOV N_[0] INTO " + splitline[0])
            output.append("LOD N_[0]")
            output.append("JMP ENDEITHER" + str(linenum))
            #Go here if either are true
            output.append("TRUE" + str(linenum) + ":")
            output.append("MOV N_[1] INTO " + splitline[0])
            output.append("ENDEITHER" + str(linenum) + ":")

        cont = 1
    return cont

#Calls the appropriate multiply library call based on the given values - writes the appropriate asm
#Takes- 2 variables, the result label, and the length of the largest variable, and the linenumber
def multiply(var1, var2, result, length, linenum):
    multType = str(length * 4)
    file = "asm-library/Mult" + multType + ".s"

    if file not in included:
        output.append("INCL " + file)
        included.append(file)
```

```
	length = str(length)
	#print var1
	#print var2
	#print result
	#print length
	op = multinibOp(length, "MOV")
	output.append(op + " " + var1 + " INTO Mult" + multType + "_Op1[1]")
	output.append(op + " " + var2 + " INTO Mult" + multType + "_Op2[1]")
	output.append("MOVADDR Return" + str(linenum) + " INTO Mult" + multType + 
"_RetAddr[1]")
	output.append("LOD N_[0]")

	#unsgined

	if(var1 not in signed and var2 not in signed):

		if int(length) == 4:
			output.append("STR Mult4_Op1[0]")
			output.append("STR Mult4_Op2[0]")
		else:
			output.append("STR Mult" + multType + "_Op1Ex")
			output.append("STR Mult" + multType + "_Op2Ex")
		output.append("JMP Mult" + multType + "_UnsignedEntry")

	#signed
	else:
		output.append("JMP Mult" + multType + "_SignedEntry")

	output.append("Return" + str(linenum) + ":")
	output.append("NOP 0")
	resLength = varLength(result)
	op = multinibOp(resLength, "MOV")
	output.append(op + " Mult" + multType + "_Ans INTO " + result)



#Checks if assigning a variable to another variable
#Takes: a splitline
def assignment(splitline, linenum):
	cont = 0
	if ((splitline[0] in varnames) and ((len(splitline) == 3) or (splitline[2] == "CALL") or 
splitline[3] == "SIGNED")):
		length = varLength(splitline[0])
		#If assigning a variable to antoehr variable
		if(splitline[2] in varnames):
```

```
            output.append("MOV " + splitline[2] + " INTO " + splitline[0])
        #Function return value
        elif(splitline[2] == "CALL"):
            line = splitline[2:]
            functionCall(line, linenum)
            output.append("MOV " + splitline[3] +  "_RetVal INTO " + splitline[0])

        #Literal value assignment
        else:
            output.append("MOV N_[" + splitline[2] + "] INTO " + splitline[0])

        #Add to signed list
        if(splitline > 3 and [3] == "SIGNED" and splitline not in signed):
            signed.append(splitline[0])
        cont = 1
    return cont

#Checks a line for any conditionals
#Takes: a splitline, linenum
def conditionals(splitline, linenum):
    cont = 0
    if(splitline[0] in ["IF", "ELSEIF", "ELSE", "ENDIF", "LOOPWHILE", "ENDLOOP", 
"LOOPAGAIN", "EXITLOOP", "EXITIF", "EARLYEND"]):

        label = splitline[0]+str(linenum)+"jump"
        output.append("LOD N_[0]")
        if splitline[0] in ["IF", "ELSEIF", "LOOPWHILE"]:
            lengthDif = compareLength(splitline[1], splitline[3])

            #Get length of longer value
            length = longer(splitline[1], splitline[3])
            #print str(length) + "\n"
            #Lengths are the same or a val is a literal
            if lengthDif == 0 or lengthDif == None:
                var1 = literal(splitline[1])
                var2 = literal(splitline[3])

            #Extends the first or second value as needed
            #first var is smaller
            elif lengthDif < 0:
                diff = -lengthDif
                var1 = extend(splitline[1], diff)
                var2 = splitline[3]

            #First var is larger
            elif lengthDif > 0:
```

```
            diff = lengthDif
            var1 = splitline[1]
            var2 = extend(splitline[3], diff)
        minnib = 1
        suffix = str(length*4)


        staycond = splitline[2]
        if staycond == "EQUALS":
            jumptype = "JMPNE"
        elif staycond == "NOTEQUALS":
            jumptype = "JMPEQ"
        elif staycond == "GREATER":
            jumptype = "JMPLE"
        elif staycond == "GREATEREQUALS":
            jumptype = "JMPL"
        elif staycond == "LESS":
            jumptype = "JMPGE"
        elif staycond == "LESSEQUALS":
            jumptype = "JMPG"

        if length != minnib:
            jumptype = jumptype + str(suffix)

        splitline[1] = literal(var1)
        splitline[3] = literal(var2)


        if splitline[0] == "IF":
            ifStatementStack.append(label)
            endifStack.append("endif" + str(linenum))
            output.append(jumptype + " " + splitline[1] + " " + splitline[3] + "
TO " + label)

        elif splitline[0] == "ELSEIF":
            output.append("JMP "+str(endifStack[-1]))
            output.append(str(ifStatementStack.pop()) + ":")
            ifStatementStack.append(label)
            output.append(jumptype + " " + splitline[1] + " " + splitline[3] + "
TO " + label)

        elif splitline[0] == "LOOPWHILE":
            output.append("start" + label + ":")
            output.append(jumptype + " " + splitline[1] + " " + splitline[3] + "
TO " + label)
            loopStatementStack.append(label)
```

```
		elif splitline[0] == "ELSE":
			output.append("JMP "+str(endifStack[-1]))
			output.append(str(ifStatementStack.pop()) + ":")
			ifStatementStack.append(label)

		elif splitline[0]== "ENDIF":
			output.append(str(ifStatementStack.pop()) + ":")
			output.append(str(endifStack.pop()) + ":")

		elif splitline[0] == "ENDLOOP":
			label = str(loopStatementStack.pop())
			output.append("JMP start"+label)
			output.append(label + ":")

		#GOTO derivatives
		elif splitline[0] == "LOOPAGAIN":
			output.append("JMP start"+str(loopStatementStack[-1]))
		elif splitline[0] == "EXITLOOP":
			output.append("JMP " + str(loopStatementStack[-1]))
		elif splitline[0] == "EXITIF":
			output.append("JMP "+ str(endifStack[-1]))

		#Earlyend
		elif splitline[0] == "EARLYEND":
			output.append("JMP END")

		cont = 1
	return cont

#goto and labels
#Takes: splitline
def goto(splitline):
	cont = 0
	cont = 0
	if(splitline[0] in ["GOTO", "LABEL"]):
		if splitline[0] == "LABEL":
			output.append(splitline[1] + ":")
		elif splitline[0] == "GOTO":
			#apparently load 0 before uncoditional jumps
			output.append("LOD N_[0]")
			output.append("JMP " + splitline[1])
		cont = 1
	return cont
```

```
#Checks if the line contains any function properties
#Takes: splitline
def functionProperties(splitline, linenum):
	cont = 0

	global returns
	if splitline[0] == "FUNCTION":
		#Check if function exists:
		if splitline[1] in funcnames:
			print "Error line " + str(linenum) + ": Function " + splitline[1] + " already
declared"

		#Add to funcnames (plus current funcnames length)
		funcnames.append(splitline[1])
		funcNum = len(funcnames)
		funcvars = ''

		#must match declaration sig
		name = splitline[1]
		funcstack.append(name)
		output.append("#" + name + " function begins here")
		#If you ge to here something went wrong so jump to end
		output.append("LOD N_[0]")
		output.append("JMP END")

		output.append(name + "Entry:")
		nextparam = 0
		paramnum = 0
		if len(splitline) > 2:
			#just takes
			if splitline[2] == "TAKES":
				nextparam = 2
				# Using 0 as default val for everything
				while len(splitline) > nextparam + 1:
					data.append(name + "_" + splitline[nextparam + 2] +": .data
" + splitline[nextparam + 1] + " 0")
					varnames.append(name + "_" + splitline[nextparam + 2])
					varsizes.append(splitline[nextparam + 1])
					funcvars += splitline[nextparam + 2] + " "
					nextparam += 3
					paramnum += 1

			elif splitline[2] == "RETURNS":
				#Jsut returns - defaults value to 0
				data.append(name + "_RetVal: .data " + str(splitline[3]) + " 0")
				if len(splitline) > 4:
```

```
							#returns and takes
							nextparam = 4
							while len(splitline) > nextparam + 1:
								data.append(name + "_" + splitline[nextparam + 2] 
+": .data " + splitline[nextparam + 1] + " 0")
								varnames.append(name + "_" + splitline[nextparam 
+ 2])
								varsizes.append(splitline[nextparam + 1])
								funcvars += splitline[nextparam + 2] + " "
								nextparam += 3
								paramnum += 1

				else:
					print "Error  on line " + str(linenum) + ": Not a valid function " 
+str(splitline[1])
					exit(1)

				#Add variable names to funcVars
				funcVars.append(funcvars)

			cont = 1

		elif splitline[0] == "ENDFUNCTION":
			name = funcstack.pop()
			if returns == 0:
				output.append(name + "_RetAddr:")
				output.append("LOD N_[0]")
				output.append("JMP 0000")
			returns = 0
			cont = 1

		elif splitline[0] == "RETURN":
			name = funcstack[-1]
			output.append("MOV " + splitline[1] + " INTO "+ name + "_RetVal")
			output.append("LOD N_[0]")
			output.append(name + "_RetAddr:")
			output.append("JMP 0000")
			returns = 1
			cont = 1
		return cont


#Checks for and performs function calls
#Takes: splitline, current line number
def functionCall(splitline, linenum):
```

```
	cont = 0
	if splitline[0] == "CALL":

		#For this part to work we need to specifiy that functions go at top of program?
		if splitline[1] not in funcnames:
			print "Error on line " + str(linenum) + ": Function " + splitline[1] + " does
not exist"

		#Fill the correct var
		funcVarLoc = funcnames.index(splitline[1])
		funcvars = funcVars[funcVarLoc].split()
		varcnt = 0

		#Has at least one variable
		if len(splitline) >= 3:
			var = literal(splitline[2])
			#output.append("MOV " + var + " INTO " + splitline[1] + "_Param0")
			output.append("MOV " + var + " INTO " + splitline[1] + "_" +
funcvars[varcnt])
			varcnt += 1

			#Has 1+ ASWELL in it
			if len(splitline) > 3:
				paramnum = 1
				nextparam = 3
				while (len(splitline) > nextparam + 1):
					var = literal(splitline[nextparam+1])
					#output.append("MOV " + var + " INTO " + splitline[1] +
"_Param" + str(paramnum))
					output.append("MOV " + var + " INTO " + splitline[1] + "_"
+ funcvars[varcnt])
					varcnt += 1
					nextparam += 2
					paramnum += 1


		output.append("MOVADDR " + splitline[1] + "_Return" + str(linenum) + " INTO
" + splitline[1] + "_RetAddr[1]")
		output.append("LOD N_[0]")
		output.append("JMP " + splitline[1] + "Entry")
		output.append(splitline[1] + "_Return" + str(linenum) + ":")
		output.append("NOP 0")

		cont = 1
	return cont
```

```
#TODO compelte this
#Checks if a special asm function is being created - creates the required output
#Takes a splitline
def asmFunction(splitline, linenum):
	global asmFunc
	#First call - initialize function params
	if asmFunc == 0:
		#get rid of ASM
		funcProp = splitline[1:]
		functionProperties(funcProp, linenum)
		asmFunc = 1

	#End function
	elif len(splitline) > 2 and splitline[1] == "ENDFUNCTION":
		asmFunc = 0
		funcProp = splitline[1:]
		functionProperties(funcProp, linenum)

	#Inside a ASM function
	else:
		temp = ''
		#ASM should fall through and labels should be replaced
		for word in splitline:
			#TODO i think im misinterpreting what Ryan/Naomi want me to do here
with the special params?
			if word in varnames:
				temp += "$" + word + " "
			else:
				temp +=  word + " "
		output.append(temp)



#Checks for included files and does the appropriate stuff for them- can be of type .CB or .s  TODO
is it .s?
#Takes a splitline
def includeFile(splitline, linenum):
	cont = 0
	cont = 0
	if len(splitline) == 3 and splitline[0] == "INCLUDE":
		if splitline[1] == "CUTEBASIC":
			files.append(splitline[2])

		elif splitline[1] == "ASM":
			asmFile(splitline[2])
```

```
		else:
			print "Error on line: " + str(linenum) + " Not a valid CB or macro asm file"
			exit(1)
		cont = 1
	return cont

def baseaddr(splitline, linenum):
	if splitline[0] == "BASEADDRESS":
		output.append("INF " + splitline[1])
		baseaddress = int(splitline[1])
		return 1

#Pretty much the same as the main loop except it uses included files instead, it deletes them from the lsit when they are read in
def inlineFile():
	linenum2 = 1
	inputf2 = open(files[-1], 'r')
	name = files[-1]
	#This loop is the compiler section - converts all lines of CB into lines of Macro ASM
	for line2 in inputf2.readlines():
		#Check for comments
		continueBool = checkComment(line2)
		if continueBool == 1:
			linenum2 += 1
			continue

		#Check if line is START or END
		startEnd(line2)

		splitline2 = line2.split()

		#Check for empty line
		if len(splitline2) == 0:
			output.append("") #preserves empty lines, an be changed if unwanted
			linenum2 += 1
			continue

		#Check for asm or passthroc, if found then line is passed through unchanged
		continueBool = asm_passthroc(splitline2)
		if continueBool == 1:
			linenum2 += 1
			continue

		#Check what the line is
		variableDeclaration(splitline2, linenum2)
```

```
			unarymath(splitline2, linenum2)
			binarymath(splitline2, linenum2)
			assignment(splitline2, linenum2)
			conditionals(splitline2,linenum2)
			goto(splitline2)
			functionProperties(splitline2, linenum2)
			functionCall(splitline2, linenum2)
			#files(splitline,linenum)
			includeFile(splitline2, linenum2)
			linenum2 += 1
		files.remove(name)


#Takes an ASM file and puts it into our file sepeariting the data and non data sections
#Takes an ASM file
def asmFile(file):
	#This as been replaced by naomi creating a way to handle this in the macro assembly
	#inputf = open(file, 'r')
	#for line in inputf.readlines():
	#	splitline = line.split()
		#Data type
	#	if(len(splitline) > 1 and splitline[0][-1] == ":"):
	#		data.append(line)
		#Non data
	#	else:
	#output.append(line)
	if(file not in included):
		output.append("INCL " + file)
		included.append(file)
	else:
		print "File " + file + " already included\n"


#******************************************************************************
*******
MAIN******************************************************************************
********************
def main():
	global output
	global data
	#data.append("########## DATA SECTION BELOW HERE ##########")
	if len(sys.argv) < 2:
		print 'No file specified'
		quit()
```

```
if len(sys.argv) == 3:
	outputf = open(sys.argv[2], 'w')
linenum = 1

files.append(sys.argv[1])
while(files):
	inputf = open(files[0], 'r')

	#This loop is the compiler section - converts all lines of CB into lines of Macro
ASM
	for line in inputf.readlines():
		#Check for comments
		continueBool = checkComment(line)
		if continueBool == 1:
			linenum += 1
			continue

		#Check if line is START or END
		continueBool = startEnd(line)
		if continueBool == 1:
			linenum += 1
			continue


		splitline = line.split()
		#print splitline
		splitline = localVars(splitline)
		#print splitline

		#Check for empty line
		if len(splitline) == 0:
			output.append("") #preserves empty lines, an be changed if
unwanted
			linenum += 1
			continue

		#Check for asm or passthroc, if found then line is passed through unchanged
		continueBool = asm_passthroc(splitline, linenum)
		if continueBool == 1:
			linenum += 1
			continue

		#Check what the line is
		continueBool = variableDeclaration(splitline, linenum)
		if continueBool == 1:
			linenum += 1
```

```
        continue

continueBool = unarymath(splitline, linenum)
if continueBool == 1:
        linenum += 1
        continue

continueBool =          binarymath(splitline, linenum)
if continueBool == 1:
        linenum += 1
        continue

continueBool = assignment(splitline, linenum)
if continueBool == 1:
        linenum += 1
        continue

continueBool = conditionals(splitline,linenum)
if continueBool == 1:
        linenum += 1
        continue

continueBool = goto(splitline)
if continueBool == 1:
        linenum += 1
        continue

continueBool = functionProperties(splitline, linenum)
if continueBool == 1:
        linenum += 1
        continue

continueBool = functionCall(splitline, linenum)
if continueBool == 1:
        linenum += 1
        continue

#files(splitline,linenum)
continueBool = includeFile(splitline, linenum)
if continueBool == 1:
        linenum += 1
        continue

continueBool = baseaddr(splitline, linenum)
if continueBool == 1:
        linenum += 1
```

```
                    continue
                #Testing a way to inline include instead of adding it just to end(because then
we get issues of stuff being after END)
                while(len(files) > 1):
                    inlineFile()
                    linenum += 1
                    continue
                print "Error - Unkown Values in line: " + str(linenum)
                exit(1)
                #linenum = linenum + 1
            inputf.close()
            del files[0]

    if hasstart == 0:
        print "No START declared!"
        quit(1)
    if hasend == 0:
        print "No END declared!"
        quit(1)
    if hasend == 1 and hasstart == 1:
        if(baseaddress == -1):
            prepend = []
            prepend.append("INF 0x400")
            output = prepend + output
        output = output + data
        for i in output:
            if(len(sys.argv)) == 3:
                outputf.write(i + "\n")
            #print i
    #for i in varsizes:
    #   print i
    #for i in varnames:
    #   print i

    quit(0)

# This if only runs if this file is called as a script - if it is included, it doesn't
if __name__ == "__main__":
    main()
```

Note: There is extensive amount of code that has been developed to design the Nibble Knowledge Computer. All of the code for different aspects of the software is located in an online, web-based Git repository hosting service. The Code is accessible through the link below:

https://github.com/Nibble-Knowledge

16.6 Software – Complete Macro Assembler

The macro assembler receives a macro assembly file as an input, and outputs a non-label replaced assembly file. It goes through the macro assembly file line by line and translates macro instructions into their appropriate base assembly instructions. It is setup with a variety of dictionaries which check for keywords in the macro assembly, and store the translated assembly instructions in a buffer. After all of the lines have been traversed in the macro file the buffer is output as a fully functional assembly file which can be handled by the label replacer or the assembler.

```
#!/usr/bin/python

# Naomi Hiebert coded this

#import our data structures
from accMacDict import accMac
from unaMacDict import unaMac
from binMacDict import binMac
from jmpMacDict import jmpMac
from asmDicts import opcodes, unaryOpcodes, dataTypes

#import global variables
import globalVars

# The real heart of the operation - identifies macros anywhere,
# including inside other macros! Tends to get called recursively
# since macros inside macros need to be expanded inside macros.

#Takes: a split line
#Returns: a list of (joined) lines

def expandline(splitline):
	expLine = []

	if isFallthroughLine(splitline): #the base case - encompasses several other cases
		expLine.append(" ".join(splitline))

	elif isINFLine(splitline):
		getINFValue(splitline)

	elif isIncludeStatement(splitline):
		expLine.append(handleInclude(splitline))

	elif isAccMacro(splitline):
		expLine.extend(expandAccMacro(splitline))
```

```
	elif isUnaryMacro(splitline):
		expLine.extend(expandUnaryMacro(splitline))

	elif isBinaryMacro(splitline):
		expLine.extend(expandBinaryMacro(splitline))

	elif isJumpMacro(splitline):
		expLine.extend(expandJumpMacro(splitline))

	else:
		syntaxfail(splitline)

	return expLine

#boolean functions - for identifying macros and syntax errors

#all the base cases return true on this line
def isFallthroughLine(splitline):
	if isBlankOrComment(splitline):
		return True
	elif isNativeASM(splitline):
		return True
	elif isSoleLabel(splitline):
		return True
	elif isData(splitline):
		return True
	else:
		return False

#INF header lines get grabbed
def isINFLine(splitline):
	if len(splitline) == 2 and splitline[0] == "INF":
		return True
	else:
		return False


#blank or comment lines fall through unchanged
def isBlankOrComment(splitline):
	if len(splitline) == 0 or splitline[0][0] == '#' or splitline[0][0] == ";":
		return True
	else:
		return False

#checks if it's native ASM.
def isNativeASM(splitline):
```

```
	if len(splitline) == 2 and splitline[0] in opcodes:
		if globalVars.DataFields:
			structurefail(splitline)
		return True
	elif len(splitline) == 1 and splitline[0] in unaryOpcodes:
		if globalVars.DataFields:
			structurefail(splitline)
		return True
	else:
		return False

#checks if it fits the standard label syntax, alone on a line
def isSoleLabel(splitline):
	if len(splitline) == 1 and splitline[0][-1] == ':':
		return True
	else:
		return False


#checks if it's a data declaration, possibly with label
def isData(splitline):
	if len(splitline) > 1 and splitline[0] in dataTypes:
		globalVars.DataFields = True
		return True
	elif len(splitline) > 2 and splitline[1] in dataTypes:
		globalVars.DataFields = True
		return True
	else:
		return False

#detects INCL statements
def isIncludeStatement(splitline):
	if splitline[0] == "INCL" and len(splitline) == 2:
		return True
	else:
		return False

#detects accumulator-based macros
def isAccMacro(splitline):
	if len(splitline) == 2 and splitline[0] in accMac and splitline[1] == "ACC":
		return True
	else:
		return False

#detects unary operation macros
def isUnaryMacro(splitline):
```

```
	if len(splitline) == 4 and splitline[0] in unaMac and splitline[2] == "INTO":
		return True
	elif len(splitline) == 2 and splitline[0] in unaMac and splitline[1] != "ACC":
		return True
	else:
		return False

#detects binary operation macros
def isBinaryMacro(splitline):
	if len(splitline) == 5 and splitline[0] in binMac and splitline[3] == "INTO":
		return True
	elif len(splitline) == 3 and splitline[0] in binMac:
		return True
	else:
		return False

#detects jump macros
def isJumpMacro(splitline):
	if len(splitline) == 5 and splitline[0] in jmpMac and splitline[3] == "TO":
		return True
	else:
		return False

#complains when it can't figure out what you're saying
def syntaxfail(errorline):
	raise Exception("Syntax Error!", " ".join(errorline))

#complains when you put data in front of instructions
def structurefail(errorline):
	raise Exception("Structural Error: Instructions cannot be placed after data fields!",
		" ".join(errorline))

#complains when you ask for the fifth or greater nibble of an address
def addroffsetfail(errortoken):
	raise Exception("Addressing Error: Addresses are only four nibbles long!", errortoken)


#replacement functions - expand those macros!


# The simplest expasion function, since no acc macro
# takes any arguments. Some might contain other macros
# though, so we still need to check

#Takes: a split line (as list of single-word strings)
#Returns: a list of (joined) lines (possibly a single-element list)
```

```
def expandAccMacro(inMac):
	outlines = []

	for line in accMac[inMac[0]].splitlines():
		countMacroUsage(line.split())
		outlines.extend(expandline(line.split()))
	return outlines

# Really the only difference between unary and binary is
# the number of arguments. That's why the functions are
# almost identical.

#Takes: a split line
#Returns: a list of (joined) lines
def expandUnaryMacro(inMac):
	outlines = []
	op1 = inMac[1]

	#Assume in-place operation if no dest given
	if len(inMac) == 4:
		dest = inMac[3]
	else:
		dest = op1

	for line in unaMac[inMac[0]].splitlines():
		splitline = line.split()

		#replace our placeholder labels with the input ones
		splitline = replaceLabels(splitline, "$op1", op1)
		splitline = replaceLabels(splitline, "&op1", op1)
		splitline = replaceLabels(splitline, "$dest", dest)
		countMacroUsage(splitline)

		#recursively expand the resulting line
		outlines.extend(expandline(splitline))
	return outlines

#Takes: a split line
#Returns: a list of lines
def expandBinaryMacro(inMac):
	outlines = []
	op1 = inMac[1]
	op2 = inMac[2]

	if len(inMac) == 5:
		dest = inMac[4]
```

```
	else:
		dest = op1

	for line in binMac[inMac[0]].splitlines():
		splitline = line.split()

		#replace our placeholder labels with the input ones
		splitline = replaceLabels(splitline, "$op1", op1)
		splitline = replaceLabels(splitline, "$op2", op2)
		splitline = replaceLabels(splitline, "$dest", dest)
		countMacroUsage(splitline)

		#recursively expand the resulting line
		outlines.extend(expandline(splitline))
	return outlines

# Frankly, this is no different from the operation macros.
# I just split them into different dictionaries for ease of
# coding and maintenance. The only cost of that decision was
# having to write this function, which is basically identical
# to the functions above.

#Takes: a split line
#Returns: a list of lines
def expandJumpMacro(inMac):
	outlines = []
	op1 = inMac[1]
	op2 = inMac[2]
	dest = inMac[4]

	for line in jmpMac[inMac[0]].splitlines():
		splitline = line.split()

		#replace our placeholder labels with the input ones
		splitline = replaceLabels(splitline, "$op1", op1)
		splitline = replaceLabels(splitline, "$op2", op2)
		splitline = replaceLabels(splitline, "$dest", dest)
		countMacroUsage(splitline)

		#recursively expand the resulting line
		outlines.extend(expandline(splitline))
	return outlines

# One of the more complex bits of code in this script, if only
# because of the amount of string operations involved. Takes
# macros and part of their context, and replaces the $-marked
```

```
# placeholder tokens in the macros with the actual labels they
# should hold. Also does math on memory offsets, so we don't
# have to define a new label for each nibble of memory. Finally,
# keeps up the counter on the amount of memory used internal to
# the macros we're using. This allows us to declare only as much
# macro scratch space as we need.

#Takes: a split line,
#		the placeholder (starts with $ or maybe &) label to replace
#		the new label (maybe with [offset]) to replace it with
#Also note that the placeholder in the line may also have an offset
#Returns: a split line
#Edits: global "memUsed" variable, if necessary
def replaceLabels(splitline, oldlabel, replabel):
	outline = []

	for token in splitline:
		#put it in the output line, adapted
		if token.startswith(oldlabel) and "$" in oldlabel:
			outline.append(reptoken(token, replabel))
		elif token.startswith(oldlabel) and "&" in oldlabel:
			outline.append(repaddress(token, replabel))
		else:
			#not the label we're looking for
			outline.append(token)
	return outline


def countMacroUsage(outline):
	#check if we're using macro memory. If so, we might need to
	#expand our macro memory bank.
	#We can get away with only checking the last token on the line
	#because macro memory is always assigned to before it is used,
	#and assignment is always to the last label on a line.
	if "macro[" in outline[-1]:
		macoffset = outline[-1][outline[-1].index('[') + 1 : outline[-1].index(']')]
		macoffset = int(macoffset, 16)
		macoffset += 1
		if macoffset > globalVars.memUsed:
			globalVars.memUsed = macoffset


#Takes:		The token to replace (maybe with offset, starts with $)
#		The new label to replace things with
#Returns:
#		A new token, with calculate labels
```

```
#Assumes:
#		If replabel or token uses the "&" syntax, it already has
#		the trailing [] present, as &(label[A])[B] but definitely
#		not &(label[A]). This is always the case if this program
#		applied the "&" syntax itself; users might break things.
def reptoken(token, replabel):
	#default values, if no offset found
	oldoffset = 0
	repoffset = 0

	#get the offset from the replacement, if necessary
	if '[' in replabel and ']' in replabel:
		repoffset = replabel[replabel.rindex('[') + 1 : replabel.rindex(']')]
		repoffset = hexSmartInt(repoffset)

	#and from the old label, if necessary
	if '[' in token and ']' in token:
		oldoffset = token[token.rindex('[') + 1 : token.rindex(']')]
		oldoffset = hexSmartInt(oldoffset)

	#add them together
	newoffset = oldoffset + repoffset

	#smash together the new token
	if '[' in replabel:
		newtoken = replabel[:replabel.rfind('[')] + '[' + hex(newoffset)[2:] + ']'
	else:
		newtoken = replabel + '[' + hex(newoffset)[2:] + ']'

	if '&' in newtoken and newoffset > 3:
		addroffsetfail(newtoken)

	return newtoken

#Takes:		A token to replace (maybe with offset in [0:4], starts with &)
#			A label to replace it with (completely unrelated offset, not already using &)
#Returns:	A token formed as &(replabel[repoffset])[tokenoffset]
def repaddress(token, replabel):
	repoffset = 0
	addroffset = 0

	#get the offset from the replacement, if necessary
	if '[' in replabel and ']' in replabel:
		repoffset = replabel[replabel.index('[') + 1 : replabel.index(']')]
		repoffset = int(repoffset, 16)
		replabel = replabel[:replabel.find('[')]
```

```
		#and from the old label, if necessary
		if '[' in token and ']' in token:
			addroffset = token[token.index('[') + 1 : token.index(']')]
			addroffset = int(addroffset, 16)
			token = token[:token.find('[')]

		#assemble new token
		newtoken = "&(" + replabel + '[' + hex(repoffset)[2:] + "])[" + hex(addroffset)[2:] + ']'
		return newtoken

#deal with include statements by adding them to the FList queue
def handleInclude(splitline):
	if splitline[1] not in globalVars.FList:
		globalVars.FList.append(splitline[1])
		return ";Included " + splitline[1]
	else:
		return ";Ignored repeated include: " + splitline[1]

#deal with INF statements by, if they're in the first file,
#setting out output INF statement to have the given value.
#If there's no statement in the first file, use the default
#(1024).
def getINFValue(splitline):
	if globalVars.FIndex != 0:
		return
	else:
		globalVars.BAddr = int(splitline[1], 0)

#Like int(token, 0) but defaults to hexadecimal.
def hexSmartInt(token):
	if token[0] == '0' and not token.isdigit():
		if len(token) > 2 and token[1] == 'd':
			return int(token[2:], 10)
		else:
			return int(token, 0)
	else:
		return int(token, 16)
```

Note: There is extensive amount of code that has been developed to design the Nibble Knowledge Computer. All of the code for different aspects of the software is located in an online, web-based Git repository hosting service. The Code is accessible through the link below:

https://github.com/Nibble-Knowledge

16.7 Library Function Example – 32-Bit Multiplication

This is the macro assembly library function for performing 32-bit multiplications. It takes in two numbers and returns the 32-bit multiplied value of them. It can perform signed or unsigned 32-bit, or unsafe 64-bit multiplication. It must be included in any assembly file that wishes to call it, and the return address must be updated before being called using self-modifying code, which is described in the macro assembler.

```
#This function can be called three ways:
#	-signed 32-bit multiplication
#	-unsigned 32-bit multiplication
#	-unsafe 64-bit multiplication

#to call Mult32 as signed 32 multiplication:

MOV32 $Op1 INTO Mult32_Op1
MOV32 $Op2 INTO Mult32_Op2
MOVADDR Return INTO Mult32_RetAddr[1]
LOD N_[0]
JMP Mult32_SignedEntry
Return:
NOP 0

#to call Mult32 as unsigned 32 multiplication:

MOV32 $Op1 INTO Mult32_Op1
MOV32 $Op2 INTO Mult32_Op2
MOVADDR Return INTO Mult32_RetAddr[1]
LOD N_[0]
STR Mult32_Op1Ex
STR Mult32_Op2Ex
JMP Mult32_UnsignedEntry
Return:
NOP 0

#to call Mult32 as (sign-agnostic) 64-bit multiplication:

MOV64 $Op1 INTO Mult32_1full
MOV64 $Op2 INTO Mult32_2full
MOVADDR Return INTO Mult32_RetAddr[1]
LOD N_[0]
JMP Mult32_64Entry
Return:
NOP 0
```

```
#64-bit answer is now in Mult32_Ans.
#The answer will be determined according
#to the same signedness as the call.
#If you only want a 32-bit answer,
#look in Mult32_Ans[8]

Mult32_SignedEntry:

;Get sign-extension bit patterns
LOD Mult32_Op1[0]
SIGNEX ACC
STR Mult32_Op1Ex[0]
LOD Mult32_Op2[0]
SIGNEX ACC
STR Mult32_Op2Ex[0]

Mult32_UnsignedEntry:

;appropriately extend Op1
LOD Mult32_Op1Ex[0]
STR Mult32_1full[0]
STR Mult32_1full[1]
STR Mult32_1full[2]
STR Mult32_1full[3]
STR Mult32_1full[4]
STR Mult32_1full[5]
STR Mult32_1full[6]
STR Mult32_1full[7]

;appropriately extend Op2
LOD Mult32_Op2Ex[0]
STR Mult32_2full[0]
STR Mult32_2full[1]
STR Mult32_2full[2]
STR Mult32_2full[3]
STR Mult32_2full[4]
STR Mult32_2full[5]
STR Mult32_2full[6]
STR Mult32_2full[7]
;64-bit version needs no extensions
Mult32_64Entry:
LOD N_[0]
STR Mult32_Ans[0]
STR Mult32_Ans[1]
STR Mult32_Ans[2]
```

```
STR Mult32_Ans[3]
STR Mult32_Ans[4]
STR Mult32_Ans[5]
STR Mult32_Ans[6]
STR Mult32_Ans[7]
STR Mult32_Ans[8]
STR Mult32_Ans[9]
STR Mult32_Ans[A]
STR Mult32_Ans[B]
STR Mult32_Ans[C]
STR Mult32_Ans[D]
STR Mult32_Ans[E]
STR Mult32_Ans[F]
STR Mult32_loopCount[0]
LOD N_[1]
STR Mult32_mask[0]
;Outer loop
Mult32_outerLoopStart:
LOD Mult32_2full[F]
STR Mult32_o2nib

;Inner loop
Mult32_innerLoopStart:
LOD Mult32_o2nib
NND Mult32_mask
NND N_[F]

;Add, if necessary
JMP Mult32_doneAdd
ADD64 Mult32_1full Mult32_Ans INTO Mult32_Ans

;Mess with operands appropriately
Mult32_doneAdd:
LSHIFT64 Mult32_1full INTO Mult32_1full
LROT Mult32_mask INTO Mult32_mask

;Leave inner loop, if it is time to do so
UCLC ACC
LOD Mult32_mask
ADD N_[F]
JMP Mult32_doneInner
LOD N_[0]
JMP Mult32_innerLoopStart
;Done inner loop. Do outer loop stuff.
Mult32_doneInner:
LOD Mult32_2full[E]
```

```
STR Mult32_2full[F]
LOD Mult32_2full[D]
STR Mult32_2full[E]
LOD Mult32_2full[C]
STR Mult32_2full[D]
LOD Mult32_2full[B]
STR Mult32_2full[C]
LOD Mult32_2full[A]
STR Mult32_2full[B]
LOD Mult32_2full[9]
STR Mult32_2full[A]
LOD Mult32_2full[8]
STR Mult32_2full[9]
LOD Mult32_2full[7]
STR Mult32_2full[8]
LOD Mult32_2full[6]
STR Mult32_2full[7]
LOD Mult32_2full[5]
STR Mult32_2full[6]
LOD Mult32_2full[4]
STR Mult32_2full[5]
LOD Mult32_2full[3]
STR Mult32_2full[4]
LOD Mult32_2full[2]
STR Mult32_2full[3]
LOD Mult32_2full[1]
STR Mult32_2full[2]
LOD Mult32_2full[0]
STR Mult32_2full[1]

UCLC ACC
LOD Mult32_loopCount
ADD N_[1]
STR Mult32_loopCount

;Return, if it is time
Mult32_RetAddr:
JMP 0000
LOD N_[0]
JMP Mult32_outerLoopStart

Mult32_Ans:         .data 16
Mult32_1full:       .data 8
Mult32_Op1:         .data 8
Mult32_2full:       .data 8
Mult32_Op2:         .data 8
```

```
Mult32_loopCount:    .data 1
Mult32_mask:         .data 1
Mult32_o2nib:                .data 1
Mult32_Op1Ex:                .data 1
Mult32_Op2Ex:                .data 1
```

Note: There is extensive amount of code that has been developed to design the Nibble Knowledge Computer. All of the code for different aspects of the software is located in an online, web-based Git repository hosting service. The Code is accessible through the link below:

https://github.com/Nibble-Knowledge

16.8 References

Note: References are listed using the APA Format.

1. Harris, D., & Harris, S. (2007). Digital design and computer architecture (2nd ed., p. 9, 79, 109-171, 239-252, 295-369). Amsterdam: Morgan Kaufmann.

2. Sedra, A., & Smith, K. (2013). Microelectronic Circuits (6th Ed.). New York: Oxford University Press.

3. Roth, C. (2008). Digital systems design using VHDL (2nd Ed.). Boston: PWS Pub.

4. Rabaey, J., & Chandrakasan, A. (2003). Digital integrated circuits: A design perspective (2nd Ed.). Upper Saddle River, N.J.: Pearson Education.

5. Zilog, Inc. (2015). Z80 Microprocessor, Z80 CPU User Manual. UM008007-0715. Zilog Inc.: Author: Zilog, Inc.

6. ZX Spectrum 48k Service Manual (n.d.). Retrieved October 12, 2015, from http://www.1000bit.it/support/manuali/sinclair/zxspectrum/sm/service.html

7. Intel Corporation. Moore's Law 40th Anniversary. (1965). Retrieved October 12, 2015, from http://www.intel.com/pressroom/kits/events/moores_law_40th/

8. CMOS 4-bit latch. (2003, June 1). Texas Instrument Inc. Retrieved November 2, 2015, from http://www.ti.com/lit/ds/symlink/cd4508b-mil.pdf

9. CMOS Dual Up-counters. (2004, March 1). Texas Instrument Inc. Retrieved November 2, 2015, from http://www.ti.com/lit/ds/symlink/cd4520b.pdf

10. CMOS Dual Up-counters. (2004, March 1). Texas Instrument Inc. Retrieved November 2, 2015, from http://www.ti.com/lit/ds/symlink/cd4518b.pdf

11. Dual Positive-Edge-Triggered D-Type Flip-Flops With Clear And Preset. (1993, October 1). Texas Instrument Inc. Retrieved November 2, 2015, from http://www.ti.com/lit/ds/symlink/sn54f74.pdf

12. LM741 Operational Amplifier. (2015, October 1). Texas Instrument Inc. Retrieved November 2, 2015, from http://www.ti.com/lit/ds/symlink/lm741.pdf

13. Quadruple 2-Input Positive-NAND Gates. (2003, October 1). Texas Instrument Inc. Retrieved November 2, 2015, from http://www.ti.com/lit/ds/symlink/sn74ls00.pdf

14. TRIPLE 3-INPUT POSITIVE-NAND GATES. (2003, April 1). Texas Instrument Inc. Retrieved November 2, 2015, from http://www.ti.com/lit/ds/symlink/sn74ls10.pdf

15. Van Roon, T. (2009, December 18). Transistor Tutorial: Power Amplifiers, Part 4. Retrieved November 22015, from http://www.sentex.ca/~mec1995/tutorial/xtor/xtor4/xtor4.html

16. Yanushkevich. S. (2014). Lab 4: PS/2 Keyboard. ENEL 453: Digital Systems Design. Department of Electrical and Computer Engineering. University of Calgary, Calgary, AB

17. Lamers. L (1994). AT Attachment Interface. American National Standards of Accredited Standards Committee X3. Washington DC.

18. IEEE 802.3™-2012 – IEEE Standard for Ethernet. (2012, December 28). Retrieved November 2, 2015, from https://standards.ieee.org/about/get/802/802.3.html

19. Ickes, N. (2004, April 29). VGA Video (6.111 labkit). Retrieved November 2, 2015, from http://web.mit.edu/6.111/www/s2004/NEWKIT/vga.shtml

20. Valcarce, J. (2015, April 5). VGA Video Signal Format and Timing Specifications.

Retrieved November 2, 2015, from http://www.javiervalcarce.eu/html/vga-signal-format-timming-specs-en.html

21. Walker, J. (2006, January 10). How Many Dots Has It Got? Retrieved November 2, 2015, from http://www.fourmilab.ch/documents/howmanydots/

22. Yanushkevich. S. (2014). Lab 3: VGA Display. ENEL 453: Digital Systems Design. Department of Electrical and Computer Engineering. University of Calgary, Calgary, AB

23. Chapweske, A. (2005, September 3). The PS/2 Mouse/Keyboard Protocol. Retrieved from http://www.computer-engineering.org/ps2protocol/

24. Chapweske, A. (2004, January 3). The PS/2 Keyboard Intervace. Retrieved from http://www.computer-engineering.org/ps2keyboard/scancodes2.html

25. Department of Electrical and Computer Engineering. (2015). Digital system design lab 4 manual and pre-lab exercises. Calgary, AB: University of Calgary.

26. PS2 - PS/2 Controller. (2009, February 12). Retrieved from http://valhalla.altium.com/Learning-Guides/PS2-PS2_Controller.pdf

27. Parallel ATA. (n.d.). In Wikipedia. Retrieved Aprril 10, 2016, from https://en.wikipedia.org/wiki/Parallel_ATA

28. Counter (digital). (n.d.). Retrieved April 14, 2016, from https://en.wikipedia.org/wiki/Counter_(digital)

29. Asynchronous Counters. (n.d.). Retrieved April 14, 2016, from http://www.allaboutcircuits.com/textbook/digital/chpt-11/asynchronous-counters/

30. Synchronous Counters. (n.d.). Retrieved April 14, 2016, from http://www.allaboutcircuits.com/textbook/digital/chpt-11/synchronous-counters/

31. Dong. X., Peng. M., Al-Khalili. A. (2015). Design of a 4-Bit Comparator. Project Report for COEN6511: ASIC Design. Department of Electrical and Computer Engineering. Concordia University, Montreal, Quebec

32. JPC., JWD. (2002). Basics: Machine, software, and program design. McGraw-Hill, Inc.

33. Tyson, J. (2000, August 20). How Computer Memory Works. Retrieved March 4, 2016, from http://computer.howstuffworks.com/computer-memory3.htm

34. Gramlich, W. C. (1993). Introduction to Computer Memory. Retrieved March 27, 2016, from http://gramlich.net/projects/computer_tutorial/memory.html

35. Silberschatz, Galvin, & Gagne. (2009). Operating System Concepts. Retrieved March 3, 2016, from http://iit.qau.edu.pk/books/OS_8th_Edition.pdf

36. Gramlich, W. C. (1993). Everything is a Number. Retrieved March 27, 2016, from http://gramlich.net/projects/computer_tutorial/numbers.html