UNIVERSITY OF THE PEOPLE

Tuition-Free Online University

Computer Systems

Daniel J. Taipala, Ph. D.

University of the People
225 S. Lake Ave., Suite 300
Pasadena, CA 91101

www.uopeople.edu

Tel. +1 626 264 8880

dan.taipala@uopeople.rg

F I R S T   E D I T I O N

Table of Contents

Table of Figures

Preface

This book has been developed as a companion text to the University of the People course CS1104 Computer Systems. The University of the People is an accredited online university that offers programs in business administration and computer science. The university is unique in that it has no tuition, is tuition free, non-profit, and serves students from around the world.

The computer systems course takes the student on a fascinating journey that begins by learning the properties of conductors and semiconductors, through an understanding of how these properties are used to create transistors as switches. It continues with the concepts of logic gates, which are assembled into circuits, and eventually ends with the student developing their own computer system and writing assembly language programs for the computer system they design.

The text is in a tutorial format and is intended to be used in conjunction with a course text. The course text is available from the author's web site at:

Computer Organization and Design Fundamentals, David L. Tarnoff, Copyright (C) 2005-2007. All Rights Reserved. Text used with permission of author. Available from http://faculty.etsu.edu/tarnoff/138292/

Equally important is the use of the *Logisim* simulation tool. *Logisim* is an educational tool for designing and simulating digital logic circuits. With its simple toolbar interface and simulation of circuits as you build them, it is simple enough to facilitate learning the most basic concepts related to logic circuits. With the capacity to build larger circuits from smaller sub circuits, and to draw bundles of wires with a single mouse drag, *Logisim* is used to design and simulate entire CPUs for educational purposes.

Logisim can be downloaded from
http://ozark.hendrix.edu/~burch/logisim/download.html

Audience

This text has been primarily written as a companion text for students in the University of the People's CS1104 Computer Systems course. This course is offered in both an open format and as a credit earning course that is a part of the universities computer science curriculum. The open format of the course is offered to anyone as a MOOC (massively online open course). Anyone can register to be a student of the open course. There is no fee for participating in the course in keeping with the mission of the University, which is to provide high quality tuition-free education. Students interested in the open version of the course should contact the University of the People administrative staff through the website uopeople.edu.

The course does have some recommended pre-requisites, however. The successful student will be one who is prepared for undergraduate study by having strong communication skills in English, and math preparedness comparable to a high school graduate who has completed algebra. A further prerequisite would be either the completion of an introductory course in programming using a high-level imperative programming language such as C++, Python, Java, JavaScript, or the equivalent. The ability to evaluate a computing problem, break it down into computable steps, and select the appropriate statements to solve the problem using the syntax of the programming language is the skill-set that is required.

Although the text has been written primarily for students of the University of the People, any student with the necessary prerequisite knowledge and skills can take advantage of this text.

Approach and Organization

This book has been developed as a companion text to the Computer Systems course offered by the University of the People. The course follows a 9-week term that includes 8 weeks of instruction and a final week for the comprehensive Final Exam. Each of the eight chapters of the book corresponds to each of the eight instructional weeks during the course term.

Chapter 1: Introduces transistors and the circuits that form logic gates. This chapter also introduces basic concepts of digital signals.

Chapter 2: Introduces number systems including binary, decimal, and hexadecimal. The concepts of data encoding are discussed along with binary addition and subtraction using two's complement. Finally, Boolean algebra is presented.

Chapter 3: Introduces combinational logic and covers half and full adder circuits, decoder, multiplexor, and de-multiplexor circuits and finally puts these circuits together as a simple ALU (Arithmetic Logic Unit).

Chapter 4: In chapter 4, we explore sequential logic, introduce the idea of feedback to maintain state within a logic circuit and then examine important sequential logic circuits including the D-Latch, memory register, binary counter, and the divide by two circuit.

Chapters 5 and 6: In chapters 5 and 6, we take the concepts and circuits that we have learned about, built, and put them together to form a functioning computer system implementing the von Neumann architecture.

Chapters 7 and 8: In chapters 7 and 8, we look at how the collection of circuits that form the computer can be programmed to perform specific computational tasks. In chapter 7, we learn the relationship between machine instructions and the control unit and ALU within our computer system. We learn to create programs from machine instructions and we are introduced to assembly language. In chapter 8, we explore assembly language for our computer system, we learn about the assembler, and the difference between one and two pass assemblers.

Chapter Exercises

Throughout the text there are exercises defined as learning tools to build upon the knowledge gained from the chapter. These assignments are designed to promote experiential learning and directly relate to the CS1104 Computer Systems course content. The assignments are assigned within the course and solutions are provided as part of the course content.

Because the assignments and their associated solutions are a part of the University of the People course, the solutions to the assignments are not provided within the text, nor will they be provided outside of the online course room.

The course is offered as an Open Course in addition to being offered as part of the computer science curriculum, which means that anyone is welcome to enroll in the course. Interested parties should visit the University of the People web page www.uopeople.edu for more details.

(This page intentionally left blank)

Logic circuits and Digital Signals

We begin this chapter by learning about chemistry. If you ever wondered why you needed to take that high school chemistry course, perhaps it was for this class. Following is a refresher on the atomic structure.

Chemistry?

Within the atom are protons (positive charge) and neutrons (no charge), which make up the nucleus of the atom, and electrons (negative charge) that orbit the nucleus. The electrons reside in consecutive orbits that move outward from the nucleus.

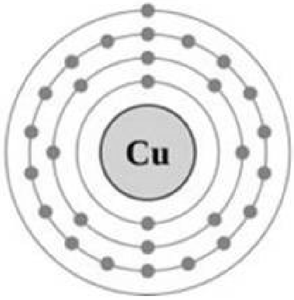


Figure 1.1 Copper Atom

The outermost of these orbits of electrons is called the *valance shell* and it is this shell that we are interested in. We know that metals such as copper are good *conductors* of electricity. A conductor is a material through which electricity can flow. In contrast, some materials are not conductors but insulators that restrict the flow of electricity. Ceramic materials are good examples of materials that are insulators that restrict the flow of electricity.

In copper (Figure 1.1), we notice that the shell has an unequal number of electrons. This imbalance of electrons means that copper can easily gain and lose electrons. When a copper atom forms a bond with another copper atom, they tend to share the electrons in their valance shells. The bond that is formed between atoms of copper is called metallic bonding and results in conduction electrons, which means that these shared electrons move freely between the atoms. Metallic bonds occur in metals that are conductors of electricity. The sharing and movement of electrons between the atoms is what we observe as electricity.

Copper is a conductor, but silicon is not. This should not be a surprise as the principle component in glass is silica (silicon dioxide) and we know that glass is an excellent insulator. Silicon (Figure 1.2) has balanced electrons in its shell.

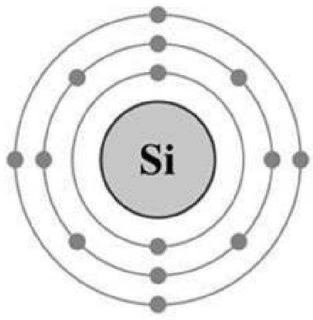


Figure 1.2 Silicon Atom

This means that Silicon does not share electrons easily between atoms in the way that copper does, and Silicon does not facilitate the flow of electricity.

In computer circuits, we need to be able to control of the flow of electricity; we need to be able to turn it on and off at will. This is where the idea of a semi-conductor comes in. Silicon is NOT a conductor, but we can alter silicon to make it conduct electricity under the right conditions.

We do this by doping the silicon. Silicon as an element forms a crystalline structure. This means that it has strong bonds and the atoms align in a rigid pattern. This atomic structure gives silicon its shiny metallic appearance. In doping, we introduce elements other than silicon into the crystalline structure.

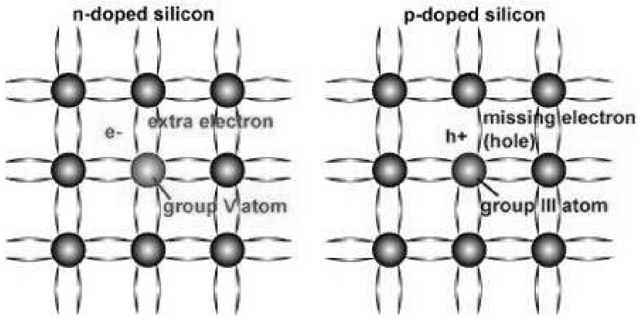


Figure 1.3 P-Type and N-Type Silicon

As we can see in Figure 1.3, doped silicon has an impurity (atoms other than silicon) in it and these non-silicon atoms give silicon the ability to become a conductor under the correct circumstances.

There are two kinds of doped silicon called p-type and n-type. P-type silicon is doped with substances that can readily gain electrons (boron, gallium, aluminum) while N-type silicon is doped with substances that can readily lose electrons (arsenic, phosphorus, antimony).

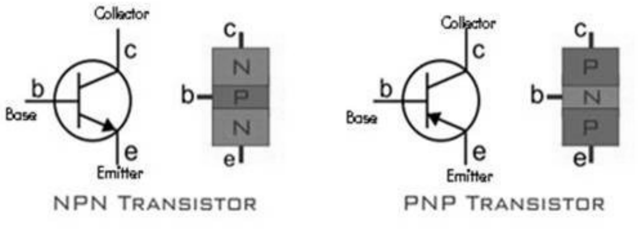


Figure 1.4 NPN and PNP Transistors

By making a sandwich of p-type and n-type silicon a device is created that can control the flow of electricity through the device.

We essentially create the ability to switch ON or OFF the flow of electricity through the silicon.   These sandwiches of p and n type silicon are called transistors and they form the foundation of all computer circuits.

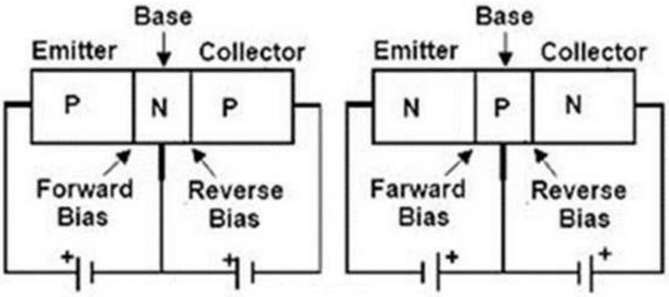


Figure 1.5 Transistor Electrical Properties

There are two types of transistors. Each type receives its name based upon the layers of silicon used to construct them. A transistor that has p-type silicon between two layers of n-type silicon is called a NPN transistor. A transistor that has n-type silicon between two layers of p-type silicon is called a PNP transistor.

Because the properties of p-type and n-type silicon differ in terms of their ability to easily gain or lose electrons, the transistors that are created using these layers require different polarities in the circuits designed to use these transistors. In figure 1.5, we see that the current to the base of a PNP transistor is negative while the current to the base of a NPN transistor is positive. This requirement for a positive base current is the reason that most logic gates employ NPN transistors. It will become clear why this is, as we begin to examine the logic circuits created using these transistors.

If we look at the diagram in figure 1.5 of both the NPN and PNP transistors, we see that current cannot flow through the transistor because the “meat of the sandwich” type of silicon in the middle restricts the flow of electricity, because it has different properties than the outer layers. The transistor functions as a switch when a current is applied to this middle layer. This middle layer in the transistor is called the base and is represented in electronic schematics as a vertical bar and often abbreviated "b" as we can see in the Figure 1.4. When the appropriate current flows to the base, current (electricity) is allowed to flow through, thus turning on the switch. When the current to the base is turned off, the transistor switches off.

Transistor Operation Illustrated

The following diagrams provide a visual illustration of the functioning of a transistor. The first circuit (Figure 1.6) has a battery connected to the base of the transistor turning it on. We can see that the transistor is turned on, because we see the current flowing through it on the voltmeter, which indicates 5 volts. The second circuit (Figure 1.7) has had the battery removed from the base circuit turning the transistor off, as we see in the voltmeter, which registers 0 volts.

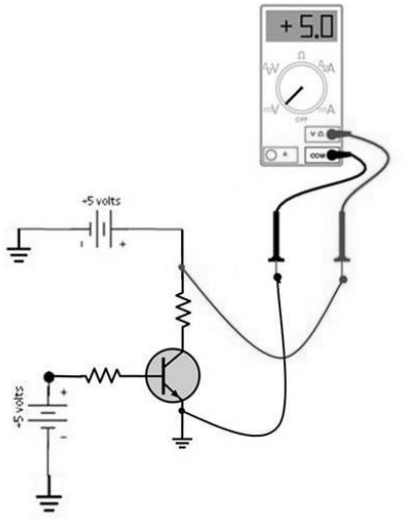


Figure 1.6 Transistor Circuit Turned On

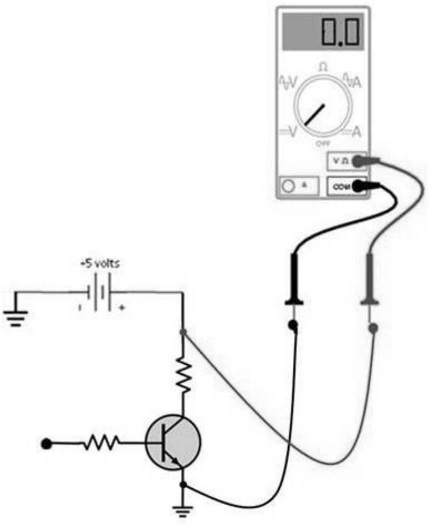


Figure 1.7 Transistor Circuit Turned Off

From Transistors to Gates

The transistor we now realize acts as a switch capable of turning current on and off in a circuit, but how does this relate to computer systems? To answer this question, we need to look at how transistors are used to form logic gates. While considering each of the following examples, remember, 1) that the bar is the base and, 2) when a current is applied to the base, that current flows from the collector (the line that goes into the base) through the emitter (the arrow that is leaving the bar).

This transistor structure is illustrated in Figure 1.4. In order for the transistor to be turned "on," a positive voltage must be applied to the base as seen in the examples illustrated in Figure 1.6 and Figure 1.7. When the positive end of the battery was connected to the base of the transistor, the circuit was turned on. When the battery was removed, the circuit was turned off.

You can think of this battery or positive voltage as a logical 1, and the lack of voltage (0 voltage when the battery is removed) as logical 0. The inputs in each of the examples represent the signal inputs into the gate and the output represents the signal output.

Later we will see that each of these gate circuits has a symbol to represent it and these symbols have inputs and outputs, which relate to the inputs in our circuits (the base of each transistor is an input) and the output is how we determine whether voltage is flowing through the circuit. The first gate that we will look at is the AND gate.

The AND Gate Circuit

The following diagram (Figure 1.8) illustrates the circuit for the AND gate. We can see that in order for current to flow through the circuit, both transistors must be turned on by applying a voltage (logical 1) to each transistor's base. The output is tested after the second transistor.

Output is the presence of an electrical current or lack of an electrical current present at the "output" point. Imagine that you had a circuit tester. If the gate was turned "on," then we would be able to measure a voltage at the point of output as illustrated in Figure 1.8.

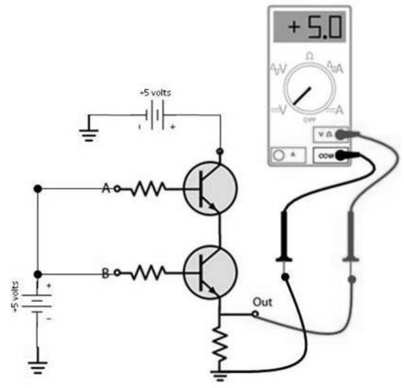


Figure 1.8 AND Gate Turned On

Notice that in this circuit we have applied a 5 volt current to the base of each transistor (the symbol with the + and – signs on it represents a source of electricity (direct current to be precise such as the current provided by a battery). We are using the voltmeter to measure the current and we can see that we have +5 volts at the output. The circuit is turned on because we have attached the battery to the bases of both transistors. If the battery were not attached to either or to neither of the transistor bases, the circuit would be "off" and the voltmeter would register 0 volts.

In this next example circuit, no current is being applied to the base, because the batteries that are not connected to the bases of the transistors and the circuit is now turned off, because the voltage at the point of output reads 0 volts.

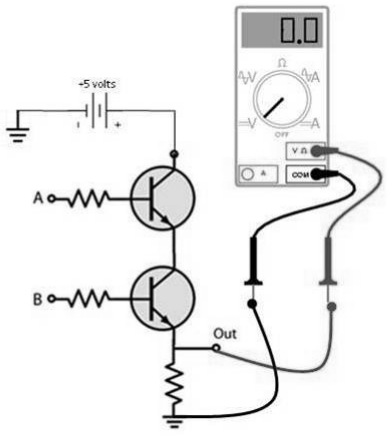


Figure 1.9 AND Gate Turned Off

The AND gate is normally "off," meaning that the output has a 0 voltage *UNLESS* both of the transistors are turned on by applying a logical 1 to the base of each transistor.

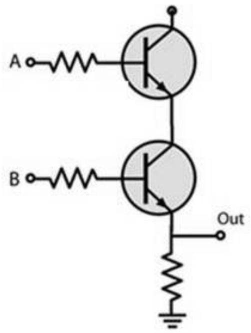


Figure 1.10 Transistor AND Gate

The NAND Gate Circuit

The NAND gate looks almost exactly like the AND gate with the exception that in the NAND gate the output is measured before the transistors. What this does is make the NAND gate have an output of logical 1 *UNLESS* both of the transistors are turned on. In this case, the voltmeter applied to the output would show +5 volts unless both transistors had a voltage applied to them in which case the output would show 0 volts.

The NAND gate takes on what is called *computational completeness*. This means that *ANY* Boolean function can be modeled using exclusively NAND gates. If you had a large enough pile of NAND gates, you could build a computer system from them.

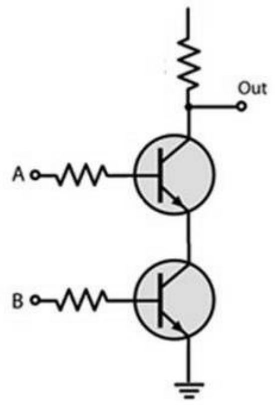


Figure 1.11 Transistor NAND Gate

The OR Gate Circuit

In the OR gate circuit we see that the gate can have a logical output of 1 if either of the transistors are turned on or if both transistors are turned on. If neither transistor base has a voltage applied, then the circuit will be “off” (logical 0).

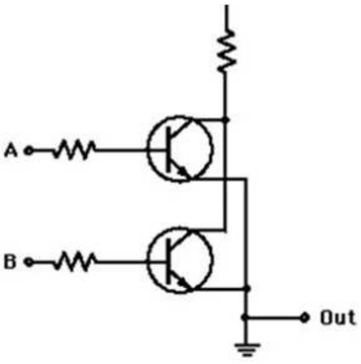


Figure 1.12 Transistor OR Gate

The NOR Gate Circuit

The NOR gate circuit is again the opposite or the negation of the OR gate, because it has a logical output of 1; unless, either or both of the transistors are turned on by applying a logical 1 to the base of either or both transistors. In this case, the gate will have an output of logical 0.

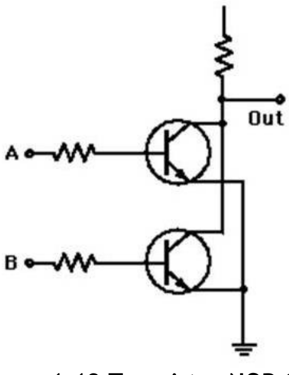


Figure 1.13 Transistor NOR Gate

The NOT Gate (Inverter) Circuit

The NOT gate is sometimes called an inverter because the output is simply the opposite value as the input.

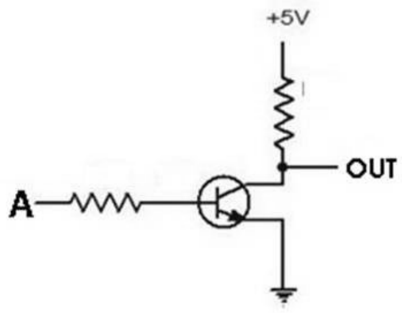


Figure 1.14 Transistor NOT Gate (Inverter)

Each of these gate circuits has a behavior with respect to its inputs and outputs. A voltage applied to the base of the transistor will turn on the transistor. This positive voltage is represented as a logical 1.

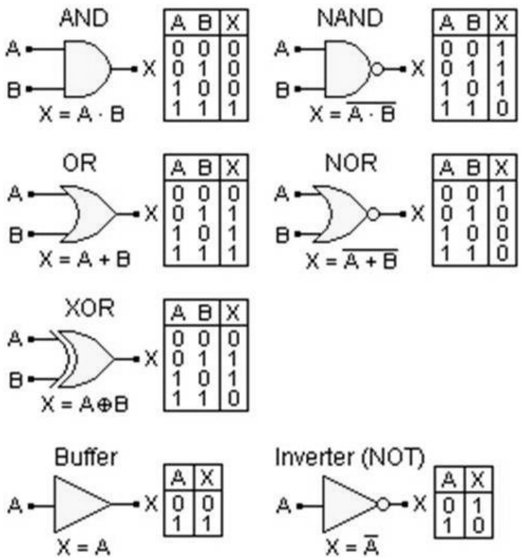
The lack of a positive voltage is represented as a logical 0. Throughout the rest of this material, it is not important that we keep track of the transistors in logic gates, because we can simply use the symbols that have been created for each gate. However, it is important that we realize the relationship between transistors and the logic gates that are made out of the transistors. It is important to realize that it is these transistor circuits that make the logic gates, behave the way they do.

Moving forward, instead of looking at the transistors in the logic gate circuits to determine when the gate will be turned on or off, we can simply look at a *truth table*. The truth table is simply a table that shows what the outputs will be for any given combination of inputs using logical values 0 or 1.

This shorthand method of determining the behavior of a gate becomes increasingly important as we begin to combine gates together with other gates to make more complex circuits.

Figure 1.15 shows the truth tables and symbols for all of the gates that we have just learned about (along with a few additional gates). Notice that the A and B inputs are simply the inputs to the transistors and the truth table shows us what the output behavior of each gate will be as the transistors are turned on, and off.

Each of the symbols is meant to represent the circuit. The logic symbol is shorthand for drawing logic circuits without the need to draw all of the components such as the batteries, transistors, ground, or resistors. All of these components are assumed present and correctly wired within the logic symbol.



AND

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

NAND

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

OR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

XOR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

Buffer

A	X
0	0
1	1

Inverter (NOT)

A	X
0	1
1	0

Figure 1.15 Logic Gates and Truth Tables

Digital Signals

In this chapter, we also introduce the concept of digital signals. The concepts of digital signals and logic gates are closely associated.

We have already discussed the fact that transistors operate as "switches" in logic gate circuits. When a current is applied to the base circuit of the transistor, the transistor is turned on. When the current is taken away from the base circuit of the transistor, the transistor is turned off.

This process of turning a circuit on or off forms a pattern referred to as a digital signal. Each transition from logic 0 (off) to logic 1 (on) or from logic 1 to logic 0 is called a cycle.

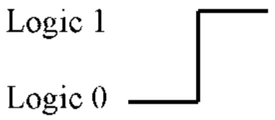


Figure 1.16 Digital Signal Cycle

In the clock circuit that will be discussed in more detail in chapter four, the cycle pattern of moving from logic 0 to logic 1 is consistent. Each cycle is referred to as a hertz (often abbreviated as Hz). The frequency of a signal is the measurement of hertz or the number of hertz that occur in a time interval, which is usually 1 second.

A signal with a speed of 1 kilohertz (KHz) refers to 1,000 cycles occurring in 1 second. A computer with a signal speed of 1 GHz means that 1,000,000,000 cycles occur in 1 second.

The cycles in the clock are consistent, because these cycles are controlled by the vibrations of a quartz crystal. The crystal only vibrates at a specific frequency. These regular cycles or "pulses" that result from the crystal's vibrations are referred to as periodic pulses because they occur at regular and precise intervals.

Figure 1.17 Quartz Clock Crystal

We know that logic gates can turn on or turn off, based upon the signal values to the inputs. The computer system relies on the ability of logic gates to output high or low logic values as signals. A low value (0 voltage) represents a logic 0 and a high value (positive voltage) represents a logic 1. These high values and low values are how we carry information in the computer system's circuits. These pulses are NOT regular, because the state (high or low) carries information in binary bits as shown in the following diagram (Figure 1.18).

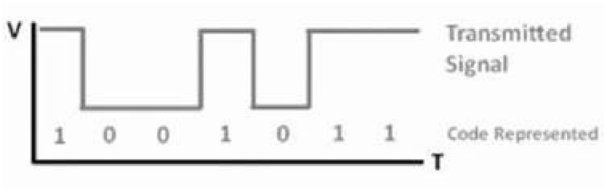


Figure 1.18 Non Periodic Pulses

These irregular signals are called non-periodic pulses. Both periodic and non-periodic pulses are essential in a computer system. The periodic pulses are tied to the clock and they provide the regular cadence of signal pulses that are used to control the execution of instructions on the computer. The non-periodic pulses carry the information that the computer system will process.

Chapter 1 Exercise 1

Using the following two timing diagrams, construct both the truth table *AND* a circuit composed *ONLY* of the primary logic gates (AND, OR, NAND, NOR, XOR, NOT, etc.) to implement each of these timing diagrams. To aid in your interpreting of the diagrams, please note that in Timing Diagram 1, the initial values for A, B, and C are 0, 0, and 1 where A and B are inputs and C represents the output. In Timing Diagram 2, the initial values for A, B, and C are 0, 0, and 0 where A and B are inputs and C represents the output.

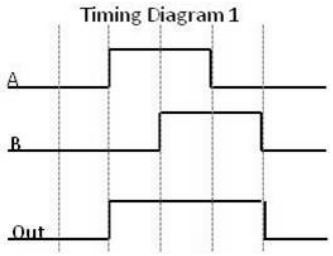


Figure 1.19 Timing Diagram 1

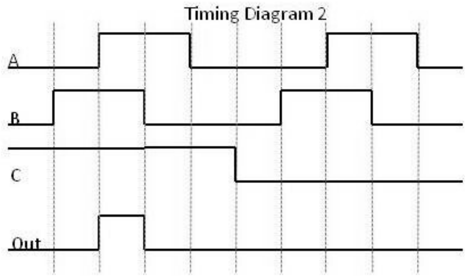


Figure 1.20 Timing Diagram 2

Chapter 1 Exercise 2

For the second exercise, download and install the *Logisim* software. Execute *Logisim* to ensure that it is working properly.

If you have issues getting the software installed or working correctly, please refer to the instructions and the help files located on the *Logisim* website.

The *Logisim* site provides both a tutorial explaining the use of *Logisim*, as well as a reference that provides details for each of the gates and other components within *Logisim* that you can use to build your circuits. The reference is available at the following URL:

http://ozark.hendrix.edu/~burch/*Logisim*/docs/2.3.0/libs/index.html

When you have successfully installed and executed *Logisim*, or accessed *Logisim* from the virtual computing lab, use *Logisim* to simulate the following transistor circuits and each of the following gates. This means that you should create the gate within *Logisim*, assign both input, and output pins to it, and then experiment with the circuits to understand their properties.

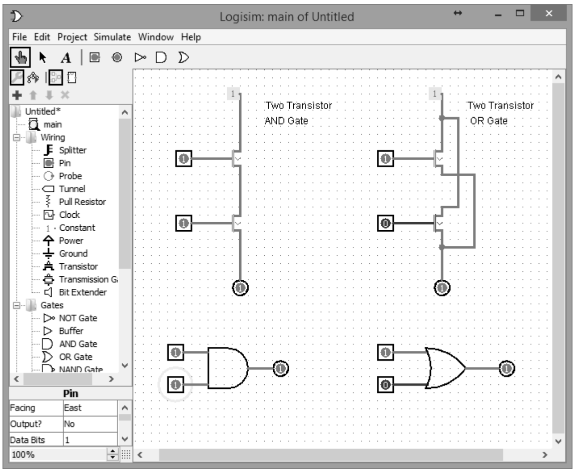


Figure 1.21: Transistor Circuits for AND and OR Gates

The following picture (Figure 1.22) illustrates the simulation of the NAND gate:

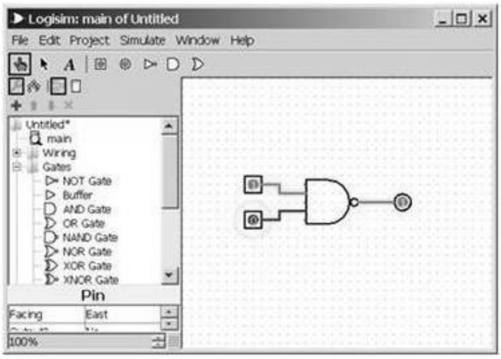
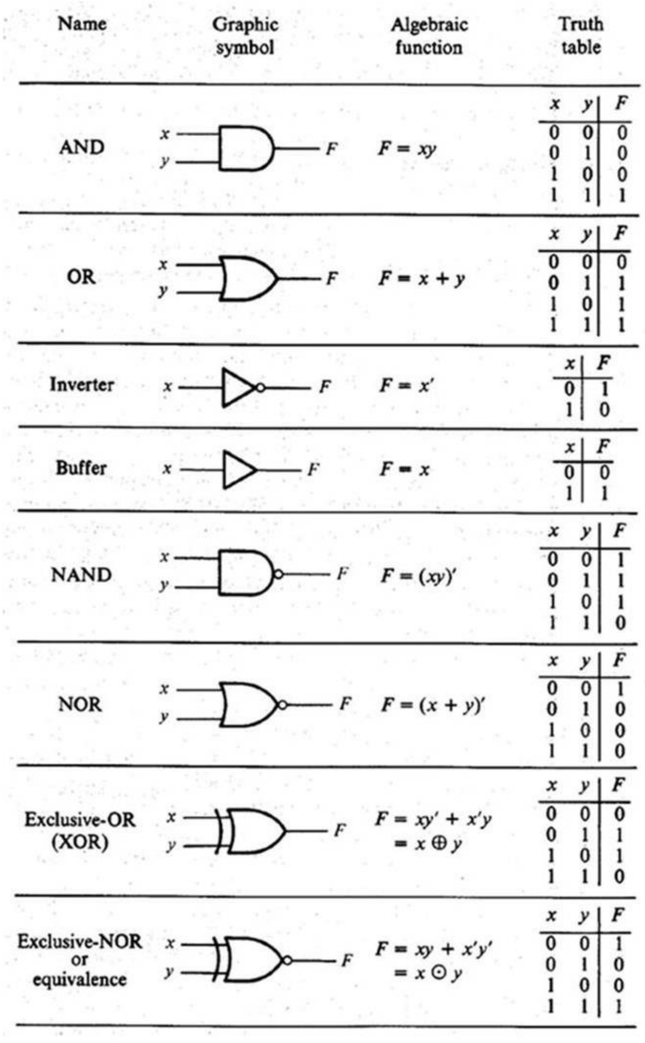


Figure 1.22 NAND Gate Simulation

In this example, we see the behavior of the NAND gate, which has an output of 1 until both of the inputs are 1, when the output becomes 0. As part of this exercise, simulate each of the gates in the following diagram and validate their truth tables using the *Logisim* simulation.

Please note that by clicking on the input pins you can change their value from 0 to 1 or back to 0 when in simulation mode. *Logisim* is in simulation mode when the Red Hand (see upper left part of the screen in the figure above) has been selected. In order to build the circuit, the arrow (to the right of the red hand) must be selected. In order wire the logic gate into a circuit, you can select input pins and place them on the screen. You can place the gates in the same manner by selecting a gate from the menu and clicking on the white portion of the screen to place it there. By clicking on a component (input pin, gate, or other component) and dragging, a wire will be created connecting the components together.



Name	Graphic symbol	Algebraic function	x	y	F
AND		$F = xy$	0	0	0
			0	1	0
			1	0	0
			1	1	1
OR		$F = x + y$	0	0	0
			0	1	1
			1	0	1
			1	1	1
Inverter		$F = x'$	0		1
			1		0
Buffer		$F = x$	0		0
			1		1
NAND		$F = (xy)'$	0	0	1
			0	1	1
			1	0	1
			1	1	0
NOR		$F = (x + y)'$	0	0	1
			0	1	0
			1	0	0
			1	1	0
Exclusive-OR (XOR)		$F = xy' + x'y = x \oplus y$	0	0	0
			0	1	1
			1	0	1
			1	1	0
Exclusive-NOR or equivalence		$F = xy + x'y' = x \odot y$	0	0	1
			0	1	0
			1	0	0
			1	1	1

Figure 1.23 Logic Gates and Truth Tables

The preceding diagram details most of the common logic gates along with their functional definitions and their respective truth tables. Using *Logisim*, simulate each of these gates (in addition to creating and simulating the transistor circuits for the AND and OR gates) and verify that their operation matches the truth tables in the graphic.

(This page intentionally left blank)

Binary Arithmetic and Boolean algebra

In the previous chapter, we discussed the transistor as a switch and the fact that we can construct gates from transistors and these gates have specific output behaviors based upon the inputs to the gate. We also learned about the idea of a digital signal, which is a signal that moves from a logical 0 to a logical 1 and back again. The ability to associate a signal voltage with a value of 0 or 1 is the method that enables computation in a computer system.

We saw in chapter one that we have the ability to create electrical circuits that represent numbers. Unfortunately, the only numbers we can use are 0 and 1.

Number Systems

We all learned about numbers in school. We learned to count and we learned that when counting we had a certain quantity of numerals that we could use to count with. Our counting began with the numeral 0 and continued with 1, 2, 3, 4, 5, 6, 7, 8, and 9. When reaching 9 we discovered that we only had ten numerals, so when we wanted to count to a number greater than 9, we had to combine some of the numerals. The next number beyond 9 combined the numerals 1 and 0. Essentially, we moved a 1 into the next digit space and began counting with our numerals again.

This system of counting and numbers is called the decimal system because ten symbols or numerals represent numbers. Referring to the ten symbols used for counting, the root of the word, “decimal” is from the Latin, “decem,” meaning “ten.” The decimal system is the most widely employed system for representing numbers and numeric values, but it is not the only system.

Consider “octal” derived from the Latin, “octo,” eight, or hexadecimal where “hexa” is derived from the Greek, sixteen. Another way of expressing these different number systems is by describing the number of symbols that are used in the counting system. Decimal has 10 symbols so we describe it as base 10, Hexadecimal has 16 symbols so we describe it as base 16, and finally there is base 2, which we commonly describe as the binary number system.

The prefix “bi,” derived from the Latin, refers to two. Thus, the binary or base 2 number system only 2 symbols to represent a value.

The hexadecimal system has 16 symbols that are typically defined to be: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, A, B, C, D, E, and F.

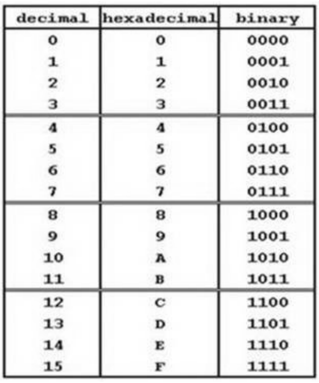
In decimal, there are 10 symbols defined as 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

In octal, there are eight symbols defined as 0, 1, 2, 3, 4, 5, 6, and 7.

Do you see a pattern here?

In binary, there are 2 symbols defined as 0, 1.

Because the binary system only requires only 2 symbols (0 and 1) and because we defined the input and output of circuits using the values of 0 and 1, it should become clear that we can use logic circuits to represent information using binary numbers.



decimal	hexadecimal	binary
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

Figure 2.24 Decimal, Hexadecimal, and Binary Numbers

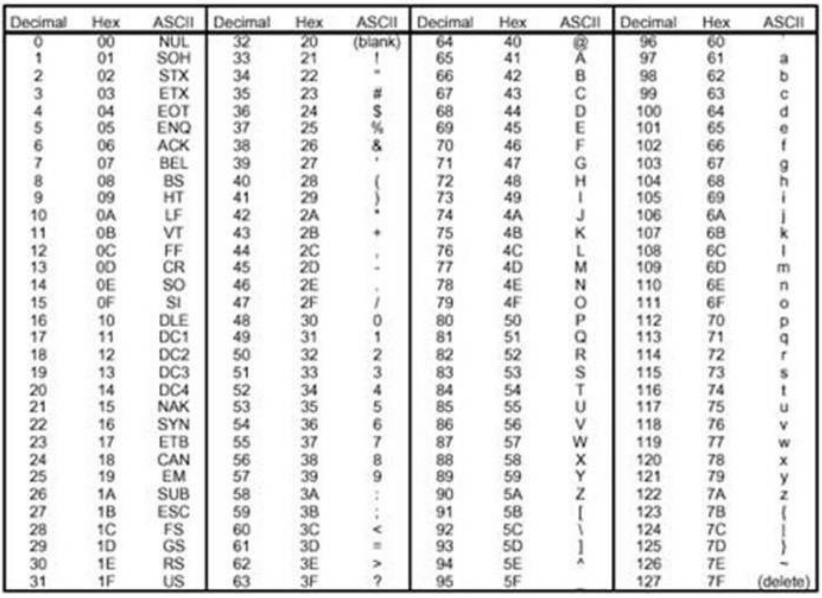
As you can see from Figure 2.1, we can convert from hexadecimal to decimal and binary.

ASCII and Unicode Encoding

It should be clear that we can use binary to represent numerical values and that we can convert something that we know and understand such as the number 13 into its equivalent in binary, which is 1101. How do we represent other information? Numbers are important, but we might also want to represent letters or words.

The answer to this is found in what we call ASCII and Unicode coding. The ASCII (American Standard Code for Information Interchange) code was based originally on the English language. It essentially took both numbers and characters commonly used in the English language and assigned an 8-bit number to each. For example, the name, "Dan," is comprised of the uppercase letter "D," which in ASCII is represented by the decimal number 65. The lower case "a" is represented by the decimal number 97 and the lower case "n" is represented by the decimal number 110. Each of these can be converted to a binary equivalent as well. The decimal number 65 in binary is 01000001. The decimal number 97 becomes 01100001 in binary, and finally, the decimal number 110 becomes 01101110.

Therefore, the binary equivalent of Dan is 01000001 01100001 01101110. However, that would not look right on a business card! Figure 2.2 shows all of the symbols and their representative numeric values for the ASCII code.



Decimal	Hex	ASCII	Decimal	Hex	ASCII	Decimal	Hex	ASCII	Decimal	Hex	ASCII
0	00	NUL	32	20	(blank)	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	(delete)

Figure 2.25 ASCII Encoding

ASCII was a useful code but had a problem. The problem was that ASCII was designed around English. It had no ability to accommodate other languages or the accent characters that exist in many languages that are based upon the Latin character set. The biggest challenge that ASCII faced was its size. ASCII is based upon 8-bits (that means 8 binary digits) and could only accommodate 255 different numbers and of course 255 corresponding characters. Languages such as Chinese have thousands of characters, so an 8-bit (also known as a byte) based coding scheme would not work.

The solution was Unicode. Unicode utilizes up to 4 bytes to represent each character, which means that Unicode has the potential to support millions of different characters.

Although the subject of coding systems is an important topic in Computer Science, it could fill many books. The key to understand is that all forms of information can be reduced to a numeric format and those numbers can all be represented in any number system including binary with its 2 numerals of 0 and 1.

Binary Addition

The binary (base 2) number system like the decimal system can be used for mathematics. The rules and procedures of addition, subtraction, multiplication, and division in binary are much the same as the rules that we know in the decimal system.

In binary addition like in the decimal system, we can add two numbers together.

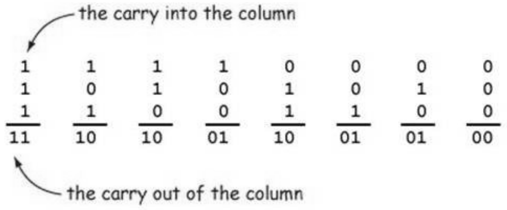


Figure 2.26 Binary Addition with Carry

In decimal addition, when we add two digits that exceed 9, we need to carry a portion of the result to the next place. This same rule is true when adding in binary. For example, if we were to add 1+1, the result would be 2. However, we do not have a 2 numeral in binary, so we need to carry to the next place. In binary, the result of 1+1 is 10. The most fundamental computation that a computer can do is ADD two binary numbers together, and this one small ability results in the complex computer systems that we have today.

If you are viewing this book with a handheld eBook reader, a or tablet, or using a smartphone or computer, it may seem absurd to consider deriving these capabilities from the addition of two numbers in binary. However, by the time that you get to the end of this book (and the end of CS1104 Computer Systems if you are using this text as part of the class), it will seem sensible to you. However, before getting into proving this statement, we need to learn a bit more about the mathematics (addition) that can be performed within the computer. Covering the concept of binary addition has been relatively simple, because it is essentially, what most of us have learned about addition of decimal numbers.

Addition is the most important thing that the computer does. One of the reasons that this statement can be made is because addition is integral to other mathematical operations. For example, in subtracting one number from another, you are really adding a negative number.

$5-6 = 5 + (-6)$

To multiply two numbers we simply add numbers together, the same number of times as the multiplier.

$5 \times 6 = 5+5+5+5+5+5$

To divide using binary, we use a technique of shift and subtract in which the divisor is repeatedly aligned with the dividend and subtracted. Of course this technique uses subtraction which is based upon addition.

Subtraction Using Two's Complement

It has been established that addition and subtraction are needed to perform all mathematical operations and it has been demonstrated that subtraction is the addition of a negative number, but how does that work in binary? We subtract two binary numbers by adding them together. The trick is that one of the numbers must be converted into two's complement format first. Two's complement transforms it into a negative number.

To convert a binary number into two's complement; we must first convert it into one's complement. One's complement is simply taking every bit and inverting it. For every 0, you turn it into a 1 and for every 1 you turn it into a 0.

The NOT gate (which is also called the inverter) inverts a bit. We already know how to convert a binary number into one's complement with a circuit: simply invert each bit with the NOT gate!

The second step in the process is nearly as easy: adding binary one to the one's complement number results in two's complement—simple!

Two's complement numbers have a unique feature. If the addition of a binary number with a two's complement number causes the last bit in the calculation to carry, then the resulting number is positive; otherwise it is negative.

If the resulting number is negative, then we need to put the number through the same two's complement conversion process again to get a binary number for which we can determine the value.

Following are a couple of examples of two's complement used to subtract binary numbers.

```
0 1 1 1  - The value of 7 in binary
1 0 0 0  - The value of 7 converted into one's complement
+     1  - Add 1 to the one's complement number
1 0 0 1  - Two's complement of 7 in binary
```

Now, subtract 7 from 8 by adding the two's complement of 7 to 8. This is the same as (-7) + 8.

```
  1001
+1000
10001
```

Notice that we have a carry bit of 1 (the digit to the far left). When we have a carry bit of 1, it means that the resulting number is positive. If the carry bit were 0 (no carry bit), then the resulting number would be negative.

In this case the number that we are left with is 0 0 0 1, which of course is binary for 1, the expected result of 8-7.

Now, try another example. In this case, subtract 7 from 5 in binary, which should result in a value of negative 2.

```
  1001
+0101
01110
```

Notice in this case that there is a carry bit of 0, which means that the result of the operation is a negative number. The problem is that the resulting number is still in two's complement, so it needs to be converted to determine what the value is. Execute the same procedure to convert from two's complement. Convert into one's complement and then add 1 to the result.

```
1 1 1 0 – The two's complement value in binary
0 0 0 1 – Converted to One's complement
+     1 – Add 1 to the One's complement
0 0 1 0 – The value 2 in binary
```

It seems to work like magic. You can use any combination of numbers to add and subtract in binary. Try it. It always works!

Binary Multiplication

Binary multiplication is nearly as easy as binary addition. In order multiply two numbers in binary, we need to know just a couple of rules. Although we will get into Boolean Algebra in more detail a bit later in this chapter, we need to remember that multiplication of two bits is essentially the same as an AND operation. We know that if we multiply 1*0 the output is 0 and if we multiply 1*1 the output is 1. In binary, the multiplication of two bits can be accomplish with the AND gate. Try it! The output of 0 AND 1 is 0 and the output of 1 AND 1 is 1 the same as in multiplication. We can use this fact to develop a circuit to multiply two numbers together.

We also need to know how to shift a bit. When we shift a bit to the left, it simply means that we move all the bits one position to the left as in the following example:

Left shifting 00000001 by 1 bit would result in 00000010

We can also shift bits to the right as in the following example:

Right shifting 10000000 by 1 bit would result in 01000000

Now that we know the rules and the process of bit shifting, we can look at how to multiply two numbers in binary. Assume that we wanted to multiply binary 1010 (10 in decimal) by 0010 (2 in decimal) of course we know that the result of this operation should be decimal 20.

```
     1010 Multiplicand
x    0010 Multiplier
---------
```

In our example, the multiplicand (the number on the top) will be multiplied by the multiplier (the number on the bottom).

Typically, to multiply two numbers we would multiply the top number by the first digit in the bottom number. In this case, we would multiply 01010 by 0. The result of this we would shift one bit to the left. Next, we multiply by the second big in the bottom number, in this case 1 and of course shift again to the left. This process is completed until all of the bits in the multiplier have

```
      1010
x     0010
----------
      0000
     1010
    0000
   0000
   -------
   0010100
```

Of course, our result is 0010100 binary, which is 20 in decimal. When implementing binary multiplication remember that after multiplying each bit, the resulting product is shifted 1 bit to the left and this number is then used as the multiplicand in the next iteration. Also keep in mind that the multiplication of two bits is nothing more than an AND operation.

Binary Division

We have already suggested that dividing numbers in binary will require two operations, shift and subtract. In division, the number to be divided is called the dividend, the number that you will divide into the dividend and the result of the division operation is called the quotient.

The division process is relatively simple. The divisor is compared with the first digit of the dividend and if it is larger then add a 0 to the quotient and then shift to the right until the dividend is larger than the dividend. Subtract the divisor from the dividend, add a 1 to the quotient and process all over again as illustrated in the following.

```
                 101  Quotient
 Divisor   10|   1010 Dividend
                -10
                  -10
                   10
                   00
```

This process is repeated until there are no more digits in the dividend to be divided. Any amount remaining from the last subtraction is the remainder and the accumulated quotient bits is the quotient of division.

Encoding Floating Point Numbers

Representing both characters and integer numbers in binary has been rather straight forward. However, we cannot rely upon the fact that we can always use an integer when representing numeric data. We often will need to represent numeric values that are fractions and expressed using a decimal point.

Consider the following number:

100.25

This number might represent a price in US dollars where the portion to the left of the decimal point represents the total number of whole dollars the portion to the right of the decimal point representing the factional amount of a dollar. In US currency, this portion to the right of the decimal point is called cents and each cent is 1/100 of a dollar.

The problem with such decimal numbers is the fact that need at least two pieces of information to convey the amount. First, we need the value that is to the left of the decimal point. Then we need the portion that is to the right of the decimal point and of course, if the value represents a negative amount then we also will need to have some way to keep track of the sign so that we know if the value is positive or negative.

Of course, we could simply keep three binary numbers one for the portion to the left of the decimal point, one for the portion to the right of the decimal point and one for the sign of the number. The problem with this approach is that sometime we need to account for very large or very small numbers.

In mathematics, we typically represent such numbers with an exponent. For example if we had a very large number such as 3,600,000,000,000 we could use a shorthand known as scientific notation where we keep just the relevant portion of the number which is 3.6 and then define how many 0's follow it with an exponent such as $3.6x10^{12}$. This means that there are 12 digits that really appear to the right of the decimal point. Very small numbers can also be represented in the same way except that the exponent is a negative number, which means that you move the decimal point to the left instead of the right.

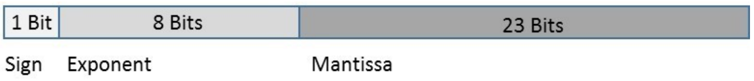
When encoding fraction numbers, which we often refer to as floating point number because of the fact that the decimal point can move (or float) to the left, or the right based upon the exponent value we need to keep three pieces of information.

First, we need to keep the sign. Since the sign of a number can only be positive or negative we can represent this with a single bit. If the value of the sign bit is 1, then the number of positive. If the value of the sign bit is 0, then the number is negative. Ok that was pretty easy.

Next, we need to keep the value of the exponent. In our example of 3.6 trillion, which we represented as $3.6x10^{12}$, the value of the exponent was 12. The size of the exponent varies based upon the data type that is being used. When we are using a data size that is 32 bits (4 bytes) then the exponent portion will often occupy 8 bits. This means that we can represent a very large number with an exponent up to 255. Think about a number with 1 followed by 255 zeros … that is a very large number indeed!

The final piece of information that we need is called the *mantissa*. The mantissa simply contains the relevant portion of the number to which we will add zeros to either the left or the right with the exponent. In our case of 3.6 Trillion, the mantissa would be 36 (with a corresponding exponent of 11 and a sign bit of 1).

All of these pieces of information are put together by convention so that the information can be accurately extracted. The way the pieces are typically put together is as follows. The example show us the structure for a 32 bit number but this could change when we are using 8, 16, or 64 bit numbers.



Boolean algebra

Boolean algebra provides us with mathematical shorthand to represent the functions of logic gates and circuits. Consider the following:

s	d_1	d_0	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

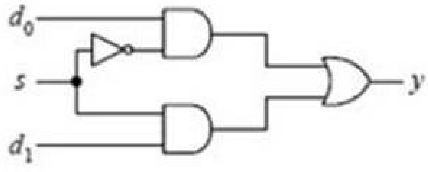


Figure 2.27 Logic Circuit and Truth Table

This is a simple circuit and yet it becomes quite complex to represent this circuit using either a diagram of logic gates or the truth table of its behavior. Boolean algebra provides us with a way to represent our logic circuits mathematically, using the symbols of Boolean algebra. The first symbol (Figure 2.5) is a way to represent AND. The logic gate representation of AND is on the left and the Boolean algebra expression on the right. Obviously, the expression on the right is much easier to write.

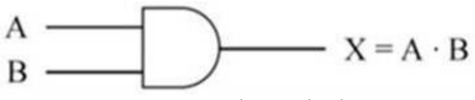


Figure 2.28 Boolean algebra AND

The AND gate is represented in Boolean algebra as the multiplication of A and B. The following are the Boolean algebra expressions equivalent to all of the logic gates that we have learned about.

AND	$A \bullet B = X$
OR	$A + B = X$
NOT	$X = \overline{A}$
NAND	$X = \overline{A \cdot B}$
XOR	$X = A \oplus B$
XNOR	$X = \overline{A \oplus B}$

In application, we can combine these expressions to represent any logic circuit. For example, consider the following circuit.

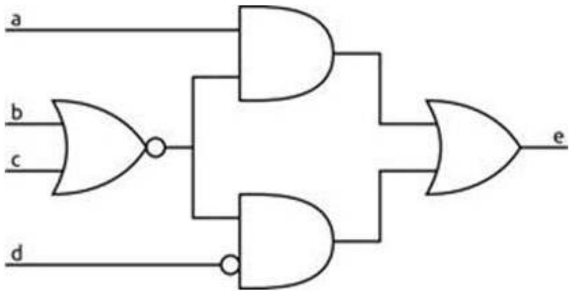


Figure 2.29 Logic Circuit

Evaluating this we have $\overline{(B+C)}$ and the output of this AND with A resulting in $A \cdot \overline{(B+C)}$. We also have $\overline{(B+C)}$ AND with NOT D which is $\overline{(B+C)} \cdot \overline{D}$. Both of these terms are then added (OR) to get our result $A \cdot \overline{(B+C)} + \overline{(B+C)} \cdot \overline{D}$. Using the Distributive law of Boolean algebra, we can simply the expression as $(A+\overline{D}) \cdot \overline{(B+C)}$.

Boolean algebra has many of the same laws as we have learned about in algebra including:

Commutative Law: Which can be described as either

$A+B = B+A$ or $A \cdot B = B \cdot A$

Associative Law: Which can be described as either

$A+(B+C)=(A+B)+C$ or $A\cdot(B\cdot C)=(A\cdot B)\cdot C$

Distributive Law: Which can be described as

$A\cdot(B+C)=A\cdot B+A\cdot C$

DeMorgan's Theorem

DeMorgan's theorem recognizes the relationship that exists between the truth tables of the NAND and NOR gates. Essentially DeMorgan's theorem recognizes the fact that if you invert the output of the NAND gate it is a NOR gate and if you invert the output of the NOR gate it is the equivalent of a NAND gate.

NAND

A	B	$\overline{A\cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

Inverted Inputs Becomes $\longleftrightarrow$

NOR

A	B	$\overline{A+B}$
0	0	0
0	1	1
1	0	1
1	1	1

Figure 2.30 DeMorgan's Theorem

What DeMorgan's theorem represents is the distribution an inverter in the output of an NAND or NOR gate back to its inputs.

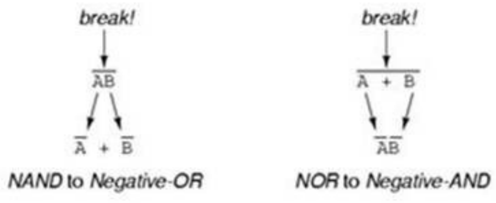


Figure 2.31 DeMorgan's Theorem for NAND and NOR

DeMorgan’s theorem is an important and valuable tool because it allows us to simplify our logic circuits. The ability to convert a NAND into a NOR (and vice versa) can often be used to simplify circuits making them faster and requiring fewer components.

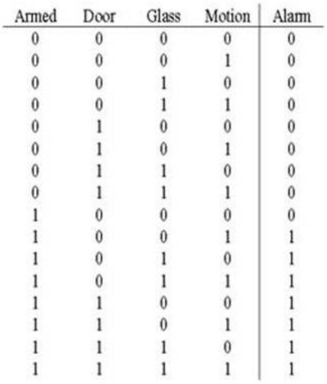
Chapter 2 Exercise

For the Chapter 2 exercise, you must complete all three of the following assignments:

First assignment: Develop a circuit using combinational logic (putting together two or more logic gates) for an alarm system. The following truth table describes the operation of the logic circuit.

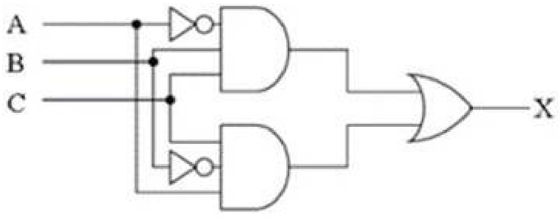
If the alarm is Armed (value of 1) and any of the following occurs: the door opens, the glass is broken, or motion is detected (all indicated by a value of 1 when any of these items are true), THEN the value of Alarm is 1, meaning that the alarm will be sounded.

Challenge Question: If you want to try something a bit more challenging to test yourself, try to make the circuit enable the alarm *ONLY* if two or more of the following events occur (door opens, glass breaks, and motion is detected).



Armed	Door	Glass	Motion	Alarm
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Second Assignment: Given the following circuit, determine what the truth table is and document it in truth table format. Your truth table must be formatted in the same way that the truth table above is formatted.



Third Assignment: Construct the truth table and *Logisim* circuit for a three input Exclusive NOR gate.

Do not use either the XNOR or the XOR gates for this assignment; you must build the functionality using other gates. For this assignment, complete both the truth table and the circuit.

Combinational Logic and the ALU

In chapter three, we will be learning about combinational logic circuits. Combinational logic circuits are essentially circuits that we build by combining logic gates together. Combinational logic circuits have inputs and outputs, the same as logic gates do. In chapter one, we looked at the truth table for individual logic gates. Chapter 2 introduced Boolean algebra as a way of evaluating more complex logic expressions. In this chapter, we develop combinational logic circuits where we leverage what we have learned about logic gates with our ability to evaluate the Boolean logic of circuits developed using multiple gates.

Combinational logic circuits are simply circuits built from logic gates that are designed to evaluate a Boolean algebra expression.

In the chapter 2 development project, we had an assignment to develop a circuit using logic gates to evaluate the truth table for an alarm system. The alarm system had a series of rules that dictated under what conditions the alarm would be triggered.

One of those conditions was that the alarm system had to be enabled. A second condition was that any two of the triggers (door, glass, motion) had to be triggered. The alarm would be triggered only when both of these conditions were met.

We can see how this problem can be reduced into a Boolean logic expression. We know that the alarm must be armed *AND* two of the triggers had to be triggered. If we consider the problem of determining when two of the triggers have been triggered we realize that we will need to use a combination of both AND and OR expressions. For example, we could state the following Boolean expression to capture this condition.

((door OR glass) AND motion) OR ((door OR motion) AND glass)

Combinational logic circuits are used to evaluate these kinds of Boolean expressions. One of the characteristics of a combinational logic circuit is that it does not hold any form of *state*. What this means is that when we apply inputs to the circuit it will produce some output. When we take away the input, the outputs are also removed. This might seem absurd, but as we will learn in the next chapter, there are also sequential logic circuits where an input will create an output and the value of the output is preserved, even when the input values are removed.

Throughout this chapter, we will look at three key topics. The first is combinational logic circuits that are used for control. These include decoders, multiplexors, and de-multiplexors. Second we will look at the half and full adder circuits. Finally, we will look at the structure of the ALU (Arithmetic Logic Unit) which will utilize both the adder circuits and the control circuits.

Adder Circuits: Half Adder, Full Adder

If someone told you that the primary thing that your computer does is add, you might have a hard time believing them, however as we explore the design of computer systems further you will come to realize that this is true. Consider that the basic operation of all mathematics is to add. To subtract we merely add a negative number. To multiply we add the number of times the value of the multiplier. The key operation in each case is, "add."

More than anything else, what makes the computer possible is the ease of adding binary numbers using logic circuits. Consider the following circuit (Figure 3.1).

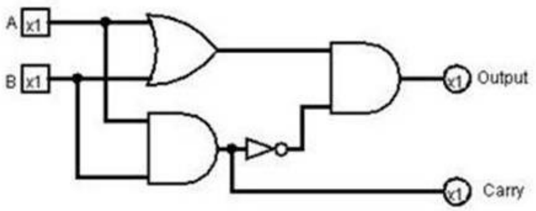


Figure 3.32 Half Adder Circuit

The truth table for the circuit is as follows (Figure 3.2). In the truth table we see that when input A=0 and B=0, then the output is 0. Think of this as adding input A and B. When either A=1 or B=1, then the output is 1. Finally, if A=1 and B=1 we would assume that since we are adding binary numbers and since we only have 2 numerals, 0 and 1, adding 1+1 would equal 2. However, since we do not have the 2 numeral we will need to carry it to the next digit, which is what occurs.

A	B	Output	Carry
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Figure 3.33 Half Adder Truth Table

The circuit in Figure 3.6 is an adder because what it does is add two binary numbers each 1 digit in length. This is excellent! We can build a circuit using transistors to form logic gates and this circuit can add two single digit binary numbers.

This example shows that we can add numbers in binary with just a few simple gates. Of course, we need to be able to add more than 1 binary digit. In order to do this, we need to be able to carry the value from one digit to the next. To accomplish this we extend the half-adder to make it a full-adder.

Before we get into looking at the full-adder circuit however, we need to review the truth table for the exclusive OR logic gate.

Recall that the characteristics of the OR gate were such that if either or both of the inputs had a value of logic 1 then the output was logic 1. There is a modification to the behavior of the OR gate that is defined as the Exclusive OR gate. In the exclusive OR gate, the output will be a logical 1 if one or the other input is logical one, *BUT NOT BOTH*. In Figure 3.3 we see both the symbol for the exclusive OR gate on the left and the exclusive OR circuit built with AND, OR, and NOT gates.

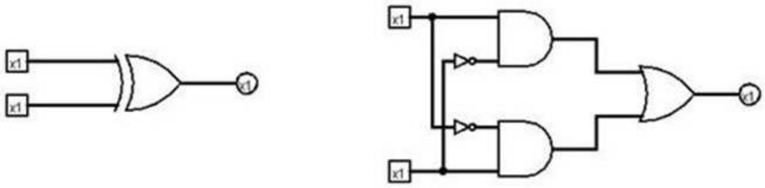
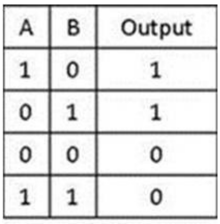


Figure 3.34 Exclusive OR Circuit

The following table illustrates the truth table of the exclusive OR gate (XOR).



A	B	Output
1	0	1
0	1	1
0	0	0
1	1	0

Figure 3.35 Exclusive OR (XOR) Truth Table

The reason that we covered the operation of the XOR gate is because the XOR gate simplifies the circuit of the full adder.

The full adder circuit is shown in the next diagram (Figure 3.5). The full adder has the additional feature over the half adder that the carry from operation can be brought into the next digit, and multiple full adder circuits can be constructed in series to accommodate a number for any quantity of bits required.

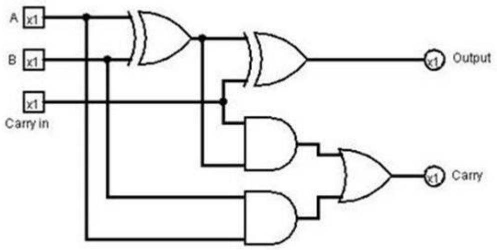


Figure 3.36 Full Adder Circuit

Binary Subtractor Circuit

Now that we know how to add binary numbers using logic circuits, what about subtract? The answer is relatively simple. Remember our discussion of one's and two's complement? To subtract two numbers we only need to convert the subtrahend into two's complement form and then add the two operands together.

Getting a number into two's complement we know is easy, as we simply need to invert the bits and then add 1 to the resulting one's complement number.

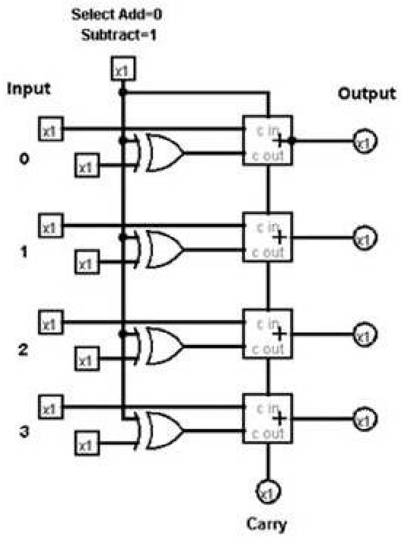


Figure 3.37 Subtractor Circuit

This can be accomplished several ways, but the circuit illustrated in figure 3.6 is a simple but elegant way to implement subtraction in binary.

In figure 3.6, we see four full adder circuits in series. The full adder is represented by the box with the + sign on it. We know that the full adder circuit has three inputs and two outputs all of which are represented in the full adder component. The full adder has an A and B input representing the two bits that will be added together. It also has a carry in input. There are two outputs, the sum bit, and the carry out. In the full adder component, the two inputs on the left of the box represent inputs A and B. The input coming into the top of the box is the carry in flag and the line coming out of the bottom of the box is the carry out flag. Finally, the line coming out the right side of the box is the sum bit.

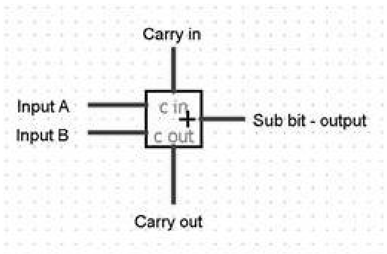


Figure 3.38 Full Adder Component

We know that to subtract two numbers we need to convert one of them into two's complement which involves inverting the bits and then adding 1 to the result.

As we examine the circuit in Figure 3.6, the first thing we notice is the subtract / add input near the top of the circuit. When this value is 1, which means that the circuit should subtract, a value of logic 1 is sent to the input of *EACH* of the XOR gates on the B inputs.

This exclusive OR has the effect of inverting all of the bits in the B input, which of course will convert the B input into one's complement. To finish the process we simply need to add one to the result. We accomplish this elegantly by taking the same subtract input and sending it into the carry in on the first full adder circuit. Essentially this adds binary 1 to the number.

Intuitively, it may not seem correct, because we are adding the 1 as we add the two numbers together. However, recall the associative law which states that A+(B+C) = (A+B)+C, and realize that we can complete each of these operations (invert bits, add 1, add operands) in any order.

Control Circuits: Decoder, Multiplexor, and De-Multiplexor

Control circuits are important logic circuits in that they allow us to control where and when we send digital signals. We have been learning about logic circuits and we know that these circuits are constructed from logic gates and we know that these logic gates are constructed from transistors. Although we may represent digital circuits using logic gates, the reality is that they are simply electrical circuits and as such, this imposes certain limitations in their design.

One limitation is that we can never have two (2) inputs into the same gate input that are active at the same time. The following diagram, Figure 3.8 should make it clear why this cannot be allowed. Assume that input A was to have a value of 1. How are we to evaluate the resulting logical expression?

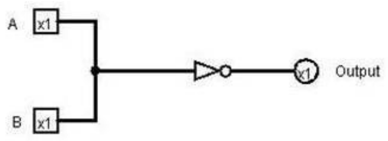


Figure 3.39 Two Inputs to NOT gate

Assume that input A were to have a value of 1, how are we to evaluate the resulting logical expression?

The problem that we have is that the expression is no longer valid. A Boolean expression cannot be evaluated if the inputs are inconsistent. If input A is 1 and input B is 0, then what really is the input? Again, this discontinuity is invalid and cannot be implemented in logic circuits. Within *Logisim*, this circuit would be flagged with an error.

Another limitation is that we must make sure that if we open a circuit it does not create a *loop* where the output of a circuit becomes its input as well (Figure 3.9). This problem is actually related to the first one in that it creates a situation where there are two potential inputs into the same logic component.

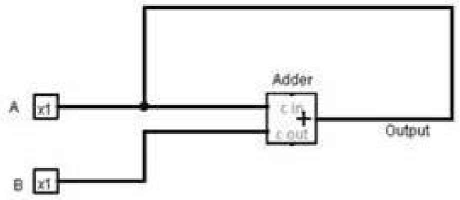


Figure 3.40 Loop Circuit

Later in this chapter, we will look at the structure of the Arithmetic Logic Unit (ALU) and one of the things that we will see is that the design of the ALU is based upon the potential ability to route the output of the ALU back to its inputs. As we will see, we need some careful planning to prevent this from violating the two input rule and the way that we will be able to prevent having this issue is by carefully controlling what signals are allowed to impact which circuits and when. Important groups of circuits that can help us to accomplish this are the Decoders, Multiplexors, and De-Multiplexors.

Multiplexor

The multiplexor is a simple concept. It has multiple inputs and one output. The basic idea is that you use a multiplexor whenever you need to make a choice between different inputs. The way that the multiplexor works is that it has two types of inputs. The first is a data input. In the diagram below, we can see eight different lines coming into the multiplexor. Each of these lines represents an input signal.

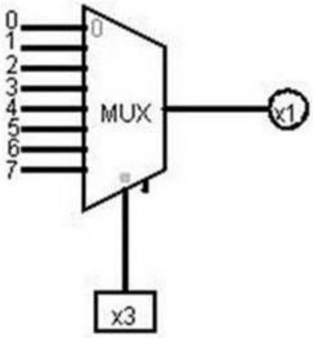


Figure 3.41 Multiplexor

One point that we should emphasize at this stage in the book is the fact that we can assume that we have 1 or more bits as an input. We have dwelt with logic gates that have had a single input (or single bit). Imagine that we have 8, 16, or 32 of the same gate in series, one for each bit of information that we need to carry. It would get very cumbersome to draw a circuit diagram with each of these gates on it, so the *Logisim* tool provides us with a shortcut because we can specify the number of bits for the input of any gate. This will be very important as we construct an ALU circuit for an 8-bit computer system that uses a 16-bit instruction.

In the multiplexor circuit, each of the eight inputs can carry a signal value but we want to select just one of the inputs. This is where the second type of input comes in. The second input is represented in the diagram above (Figure 3.10) as the line extending down from the multiplexor gate. These are called the select bits. The select bits will allow us to specify which input to select for output. In this multiplexor, we have three select bits, which means that the multiplexor can support up to eight inputs. The select bits control a series of gates that either enable or disable a particular input. The following diagram (Figure 3.11) details a multiplexor circuit built using AND and OR gates. This example shows a multiplexor with two select bits, which means it can support up to four inputs.

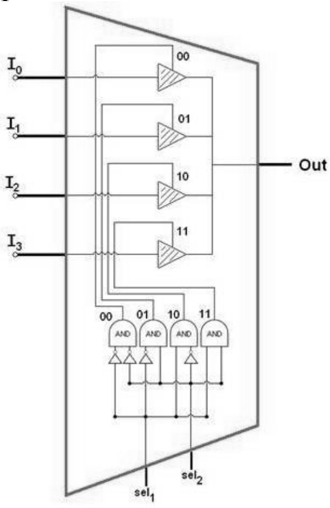


Figure 3.42 Inside the multiplexor

Notice that when both of the select bits are 0, the inverters on the first AND gate will both have a value of 1, which sends a signal to the controlled buffer turning on input one and sending it to the output. The controlled buffer is not a gate that we have used yet, but its operation is simple.

When a logical 1 signal is applied to the control bit then the buffer will pass the input signal to the output.

The symbol for the controlled buffer is shown in the following diagram (Figure 3.12) and the transistor circuit that implements it is shown in Figure 3.13. You have probably noticed that the circuit looks a lot like the circuit for the inverter, which it does, with the exception that the use of the inputs and outputs is different.

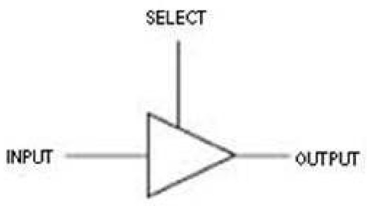


Figure 3.43 Controlled Buffer

The buffer has an input and an output. The gate will not pass the signal unless the control bit (represented by select) has a logical value of 1.

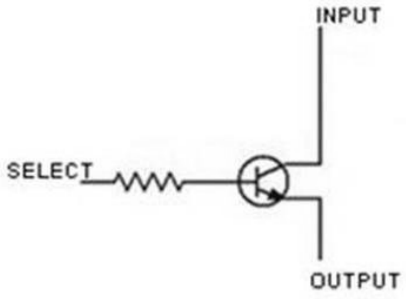


Figure 3.44 Controlled Buffer Circuit

Control buffers like other gates can have multiple bit inputs. This way we can use the buffer as a way of turning on and turning off a particular signal path. When used within a multiplexor or de-multiplexor and when used in conjunction with them, the goal of controlling when (and where) a signal is allowed to pass from or to, can be realized.

De-Multiplexor

The de-multiplexor is much like the multiplexor with the exception that instead of selecting one of many inputs to pass to the output, the de-multiplexor is used to select one of many outputs to send a single input signal through.

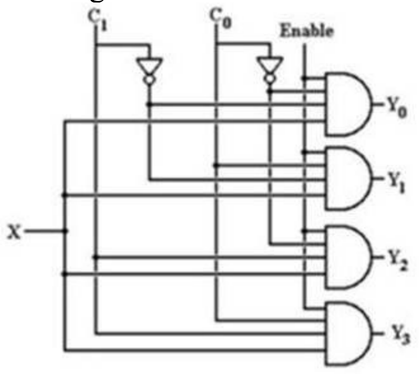


Figure 3.45 De-multiplexor

The de-multiplexor has select bits just as the multiplexor does and they work in exactly the same way. Both the multiplexor and de-multiplexor are capable of supporting signals that contain more than one bit. As such, these gates are often used as part of the computer bus to send data bits represented as signals, across many wires from one location in the computer system to another.

The ability to carry more than 1 bit of information is one of the things that differentiates the multiplexor (and de-multiplexor) from the decoder.

Decoder

The decoder circuit is a bit different from the multiplexor and de-multiplexor circuits. The objective of the decoder is to use some input value to generate or *decode* one or more outputs.

Decoders are often described using terms such as 1-to-2 decoder or 2-of-4 decoder. These terms describe how the decoder maps input values into a set of signals as output. When we say that the decoder maps an input value into a set of signals as output, it means that the output of the decoder is typically a single bit signal that has a 1-True or 0-False value.

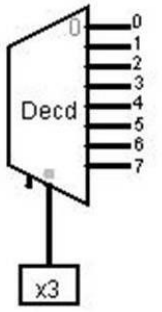
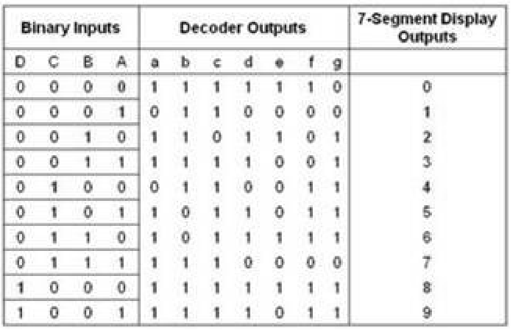


Figure 3.46 Decoder

For example, the input signal might be the number 2 in binary (X3 in Figure 3.13 is the input bits) so the function of the decoder would be to enable the signal on output number 2 of the decoder.

Other decoders may have more sophisticated logic. For example you could use a decoder to convert a binary number into the LED's that must be lighted on a seven segment display, to represent a decimal or hexadecimal number that is the equivalent of the binary input.



Binary Inputs				Decoder Outputs							7-Segment Display Outputs
D	C	B	A	a	b	c	d	e	f	g	
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	1	0	1	1	9

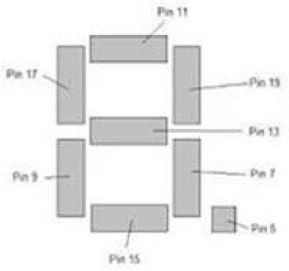


Figure 3.47 Decoding the 7 Segment Display

The ALU – Arithmetic Logic Unit

Throughout this chapter, we have learned about some important combinational logic circuits including multiplexors, de-multiplexors, decoders, and of course adders. We will now look at one way to put these circuits together to create the heart of a computer system, the ALU.

The ALU or Arithmetic Logic Unit is the portion of the computer system that actually performs computations. Many ALU's can have relatively sophisticated sets of instructions, but we are going to keep it simple and explore the design of a very simple ALU circuit that performs addition, subtraction, bitwise AND, and bitwise OR operations for two binary numbers.

Although we do not have a standard gate that we can use for an ALU, a symbol is often used to represent the ALU. The symbol is shown in the diagram (Figure 3.17) below.

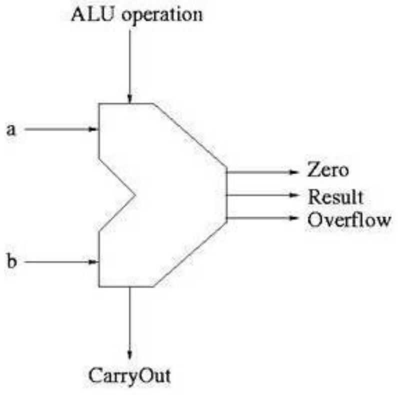


Figure 3.48 ALU (Arithmetic Logic Unit)

By looking at this diagram, we can see the common inputs and outputs of the ALU. First, the ALU has two data inputs that are labeled A and B. These two inputs are multi-bit inputs that contain binary numbers. The ALU will perform its functions against these two operands. If the ALU is performing an ADD operation, it will add the value of B to the value of A. If a Subtract operation is called for, then the value of B will be subtracted from A.

The ALU operation either is the input to a multiplexor or de-multiplexor. These select bits are used to select which operation is to be performed. If our ALU can perform ADD, Subtract, AND, or OR operations, then these bits will be used to select which *ONE* operation will be executed.

There are four outputs identified. The first is the result, which is a multi-bit binary number, and this output contains the result of the ALU operation. Some of the other outputs are called flags, which provide us with information about the operation performed by the ALU. For example, if you recall our discussion on the use of two's complement to add two binary numbers together, you will recall that we needed to be able to check the value of the carry bit to determine if the resulting number was positive or negative. The carry out flag is essentially this carry bit.

Another flag is the zero flag, which determines if the output of the ALU operation is zero. Finally, we see an overflow flag. This can be useful in addition operations to determine if an operation was requested that resulted in a number that was too large to represent in the number of bits available and resulted in an overflow.

These flags will become important to understand in chapter 5, when we look at how to implement N, P, and Z functionality. N,P,Z refers to flags that our computer system will need to support. This abbreviation stands for Negative, Positive and Zero. When the ALU computes a mathematical operation (add or subtract) we will determine if the result of that operation was a negative number, positive number, or zero, and set the appropriate Negative, Positive, or Zero flags. These flags are simply signals that contain either a logical 0 or logical 1. If the result is positive, then the positive flag will have a logical 1, if the result is negative, then the negative flag will have a logical 1, and if the result of the operation is 0, then the zero flag will have a value of logical 1.

We can create a simple ALU circuit using *Logisim* that matches the functionality that we have just discussed. The circuit will look very much like the following.

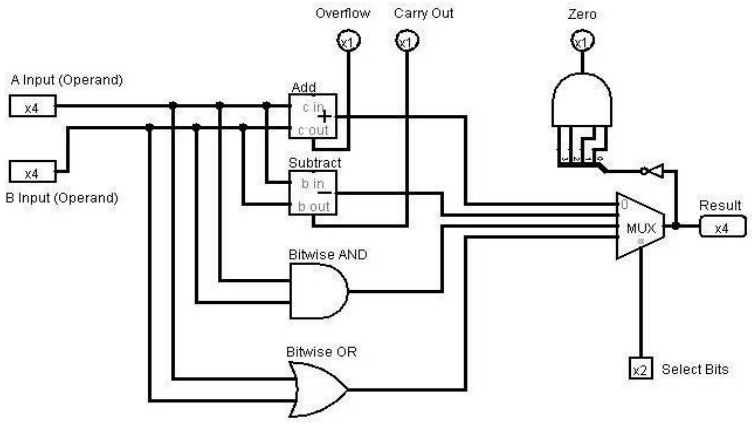


Figure 3.49 ALU Circuit with Outputs

In this circuit (a 4-bit ALU) we have an adder circuit, subtract circuit, an AND circuit, and an OR circuit.   We see the source of the flags, such as the overflow flag and the carry out flag.

The Add component and the subtract component are represented as square boxes with the + sign for Add and the – sign for subtract. As you design your ALU circuit in the chapter exercise, you should use these components.

We also see an implementation of the zero flag. In this example, we took the output (result) of the ALU, then inverted the bits and ANDed it. If all of the bits are 0, then the zero flag will be 1.

Notice that we have two inputs, operand A and operand B. These inputs go to every operation, but we use a multiplexor to send the result of *ONLY* the operation that has been selected. The particular operation is selected using the select bits for the multiplexor. In this example, bits 00 select ADD, bits 01 select subtract, bits 10 select bitwise AND, and bits 11 select bitwise OR.

A bitwise AND is simply where every bit of operand A and every bit of operand B are used as inputs to an AND gate. Essentially the output of each bit is the output of the AND operation on each pair of bits (one from A and one from B). A bitwise OR is the same thing except that each pair of bits are used as inputs to an OR gate.

Assuming two 8-bit binary numbers, the bitwise AND would work like the following example:

$$\begin{array}{ll} 01011001 & \text{AND} \\ \underline{11110111} & \\ 01010001 & \end{array}$$

Assuming two 8-bit binary numbers, the bitwise OR would work like the following example:

01011001 OR
<u>11110111</u>
11111111

In subsequent chapters, we will learn to use this simple ALU, and add additional functionality to it, such as the ability to pass the value of the A operand to the result or, to pass the value of the B operand to the result and integrate this ALU functionality with control functionality to create a complete computer system.

What is relevant is that an ALU circuit as simple as this one is capable of supporting just about any type of computation, as we will learn in subsequent chapters of this course.

Chapter 3 Exercise 1

For the first part of the Chapter 3 exercise, create a circuit using *Logisim* that implements a Full Adder circuit capable of adding 2 – 4-bit binary numbers and subtracting 2 – 4-bit binary numbers.

The circuit must use a multiplexor that will select between the add and subtract operations.
The subtract circuit will convert the second of your two input numbers into 2's complement format and then add the resulting binary number to the first number as this will effectively subtract the second binary number from the first binary number.

Your circuit should look similar to the following diagram with the two binary numbers to be added on the left, a bit selector that will select the operation to be performed. When it is 0, the adder circuit should be selected and when 1, the subtraction circuit should be selected. The output of the computation should be on the right.

Your circuit should follow this basic format, but you must design all of the actual circuits to perform addition and subtraction. You must design the actual connections between components. The representation below is an idea of where the inputs, outputs belong, and the processing that occurs between them. You can only basic logic gates including AND, OR, NAND, NOR, NOT, XOR, and XNOR, and the multiplexor, de-multiplexor, or decoder components within *Logisim* to construct your circuits.

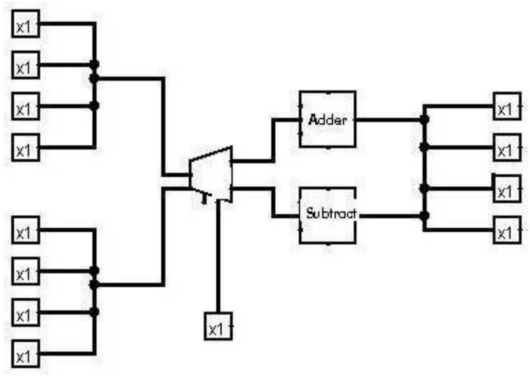


Figure 3.50 ALU Circuit

Chapter 3 Exercise 2

In the second part of the Chapter 3 exercise, you will construct an ALU (Arithmetic Logic Unit) circuit. At a minimum, your ALU must support the following functions. You are welcome to add additional functionality; however, this may require additional work in future assignments or you may need to modify your ALU circuit to be consistent with the remainder of the course assignments.

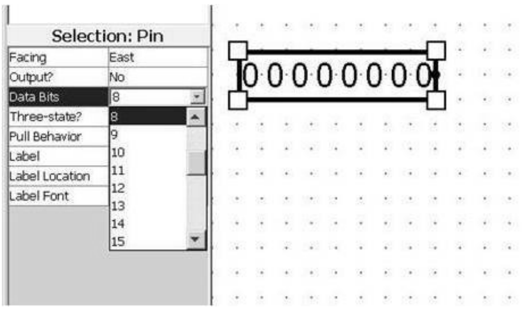
Required ALU Functionality

- ADD
- Subtract
- Bitwise AND
- Bitwise OR

- Pass through Register A
- Pass through Register D

Your ALU must support two operands. For this assignment, you should use the input pin tool within *Logisim*. The input pin tool will look like the following.

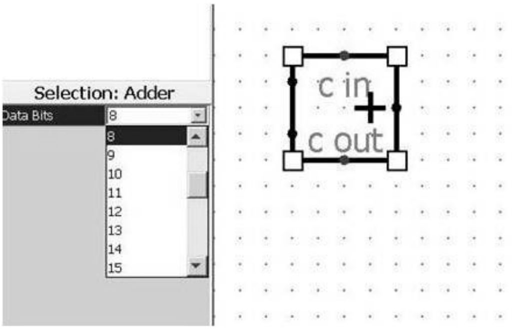
Your ALU must support operating on 8-bit numbers. This means that you will need to select the bits on the gates and components that you use within *Logisim*. The following is an example of selecting 8-bits on the Adder gate, input pin, and the AND gate.



Selection: Pin

Facing	East
Output?	No
Data Bits	8
Three-state?	
Pull Behavior	
Label	
Label Location	
Label Font	

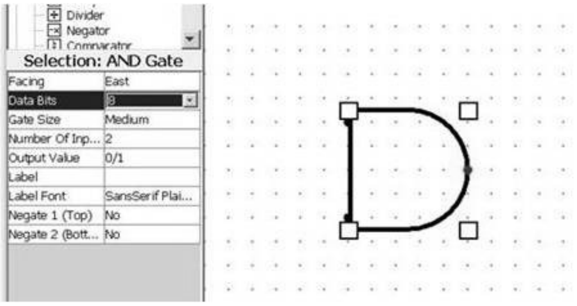
Selecting 8-bits for the input pin component.



Selection: Adder

Data Bits	8

Selecting 8-bits for the Adder component



Selection: AND Gate

Facing	East
Data Bits	8
Gate Size	Medium
Number Of Inp...	2
Output Value	0/1
Label	
Label Font	SansSerif Plai...
Negate 1 (Top)	No
Negate 2 (Bott...	No

Selecting 8-bits for the ADD gate

As part of your ALU circuit, you must use at least one of the following components: decoder, multiplexor, or de-multiplexor to control which operation (Add, Subtract, AND, OR, pass through A input, pass through B input) is executed.

With the exception of the two pass-through operations (a pass-through simply passes the input value to the output without doing anything to it), the rest of the operations must all be applied to two operands. Assume that these two inputs will be identified as input (operand) A and input (operand) B.

(This page intentionally left blank)

Sequential Logic: Registers, Memory, Counters

In the previous chapter, we learned about the ALU and its ability to perform simple computations such as ADD, subtract, bitwise AND, and bitwise OR. The ALU is part of what is called the execute cycle. In a computer system, there are many stages of execution. Modern CPU's have relatively complicated cycles and instructions can often span multiple cycles.

For the purposes of what we will learn in this class however, we will only consider a computer system that has a single cycle design, which means that an instruction can be completed within a single cycle.

Processing Execution Stages

Within that cycle, at least the following four stages must be completed. The first stage is ***fetch***.

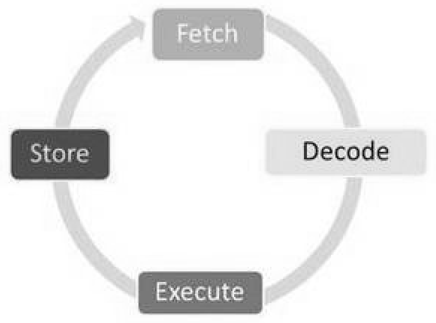


Figure 4.51 Fetch-Decode-Execute-Store Stages

The basic idea of the fetch cycle is that an instruction is retrieved from ROM memory. We know that in most modern architectures, there is no actual ROM memory. Rather the program instructions are stored in an area of RAM that is mapped as the ROM memory space.

However, for the purposes of this class we will assume there is a separate ROM memory where the program instructions are stored. These instructions are just binary numbers where each bit or group of bits has a specific meaning.

In the previous chapter, we learned how we could select the particular operation to be performed in the ALU with the select bits. Well, these select bits actually come from the instruction that is located in ROM memory!

The fetch cycle essentially loads an address into the ROM address register and positions to read a value at that address from ROM memory.

Following the fetch stage, we need to decode the instruction that we just fetched from memory. This is called the ***decode*** stage. In the decode stage, we must extract the groups of bits that make up a machine language instruction, and use those bits to control which signals are active, and in so doing, we can control what function in the ALU will be executed, where the results of the computation will be sent, and whether to execute a jump instruction.

Although some of these terms may not make sense right now, rest assured that as we proceed through the book, they will make perfect sense.

The decode stage allows the instruction fetched from ROM to set the correct selection bits to control the execution of the ALU. The execution of a particular function within the ALU is known as the ***execution*** stage.

Finally, when the ALU has computed a result from the instruction we need to know where to send the results of the computation. This is known as the ***store*** stage.

A computer's design must take advantage of both the clock signal and the design of the logic circuits to control each of these stages. Some stages such as the fetch, decode and execution stages are typically done first, so one way to manage the timing is to ensure that they are executed on the rising edge of the clock signal. The store phase can only be completed after the ALU has computed the instruction, so the store phase have to rely upon the falling edge of the cycle to ensure that each stage is completed in the proper order.

Remember that the *rising edge* of the cycle is when the clock cycles from logical 0 to logical 1 and the *falling edge* of the cycle is when the clock moves from logical 1 back to logical 0.

Preserving State

The problem that we will quickly encounter is the challenge of ensuring that we have the right set of signals in the right place at the right time. We need to ensure that we can maintain *state*, or maintain a set of signals until we can process them. Further, we need to have some way to implement structures such as ROM (read only memory) and RAM (random access memory) which must maintain a set of signals, indefinitely.

The purpose of sequential logic circuits is to maintain such state. We have learned that combinational logic circuits can take inputs and produce outputs based upon the design of the logic within the circuit, but we also know that the outputs persist only as long as the inputs are present. So how do we maintain state? In this context, state refers to the ability to maintain a set of signals indefinitely, even after the inputs that generated the signals are no longer present.

We can see how this might be useful. Imagine that we have an instruction that we fetch, decode, and send to the ALU to be computed. The resulting value will be lost the moment that the inputs change, perhaps when we move to execute the next instruction.

Therefore, we need to maintain the *state* of the signals. The secret to doing this is called feedback. Imagine creating a circuit that when a value is applied, this value is immediately fed back as an input into the gate, thus sustaining the *state* of the value. That is exactly what we do with the D-Latch, which we see in the following diagram.

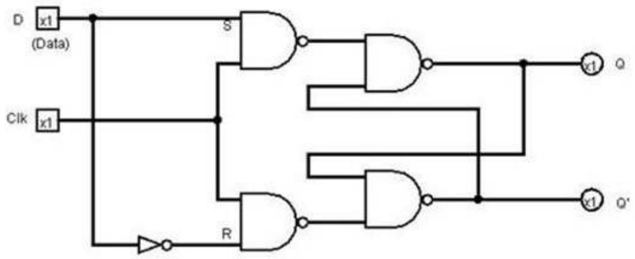


Figure 4.52 D-Latch

In this D-Latch circuit (figure 4.2), we see that we can apply a signal to S (store) and when toggling the R (reset), the value is maintained. In the D-Latch, we refer to the S input as the data input and the R input as Clock. The behavior is that the signal on the Data when the clock is toggled from 0 to 1 is stored in the latch. The Q value will contain the data from the D input and Q will contain the complement (inverse) of the data.

Throughout the remainder of this book, we will be using the memory gate in *Logisim* to represent a single bit D-Latch. The inputs and outputs of this gate are detailed in the following diagram.

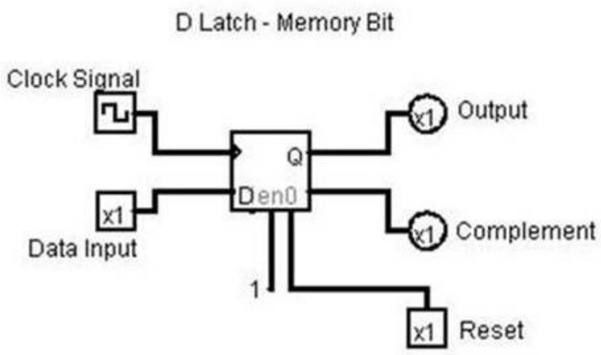


Figure 4.53 D-Latch 1 Bit Memory Cell Implemented in *Logisim*

The clock signal provides timing for the memory circuit. Essentially the value in the memory gate or cell can ONLY be updated when the clock signal has a value of logical 1. This signal can be used to control when the value in the memory cell can be updated.

The data input is the value to be stored in the memory cell. Of course, the only time that this value will be stored is when the clock signal is *high* or has a value of 1.

The output is the value that is stored in the memory cell and the complement is the complement of what is stored in the cell. For example, if the memory cell were to store the value of 1, then the complement would be 0; and if the cell stored 0, then the complement would be 1.

This memory D-latch circuit relies on feedback to maintain the state, or in other words, to store a value. We may need at times to *interrupt* this feedback circuit, which will cause the memory cell to be reset. This is what occurs when a value of logical 1 is applied to the reset input. The feedback circuit is interrupted and the memory cell is reset.

Although important, a memory cell that stores one bit is limited in its use. We need to be able to store numbers larger than a single bit in order to build effective computer systems. The solution to this problem is to put several of these 1-bit memory cells (D-Latches) together. One example of putting a series of D-Latches together is the register. In the register, there is one D-Latch for each bit required. Consider an example like the one below where we have 8-bits of information. The register would be built by putting eight D-latches together in series.

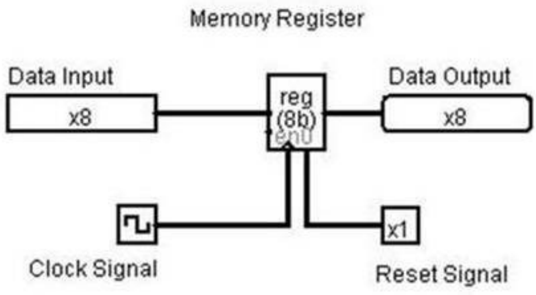


Figure 4.54 Memory Register

The register gate in *Logisim* looks like the device in the above diagram (Figure 4.4). It has three inputs and one output.

The data input is a binary number that has 1 or more bits. In the above example, we see a device that has eight input bits (represented by the x8). The signals on these eight bits will be stored in the register only when the value of the clock signal is logical 1.

The clock signal input is a single bit signal that toggles between 0 and 1. When the value rises to logic 1, the data input is stored in the register.

The reset signal that provides the interruption in the feedback circuit is required to reset the D-Latch. When this input has a value of logic 1, the circuit is interrupted and the register is reset. This sets the value in the register bits to all zeros.

Finally, we have one output which is the data output. This group of signals contains the value stored in the memory register.

There is an important concept that we need to understand when looking at the register circuit. We see that both the input and the output is a group of bits. In the case of the example that we see above, there are 8-bits in the input and 8-bits in the output. You have to realize that the one input or output line actually represents eight different wires to carry the eight different 1-bit signals.

This idea of having a bundle of wires that can be used to carry multiple bits of information simultaneously into or out of a gate or circuit, is called a ***BUS***. As we continue to design our computer system throughout this course, we will be using many such busses. We will need a bus to carry signals from memory to the ALU and to carry the result of the ALU to memory, to the ALU registers (A Register or D Register), or the address registers.

We have explored a single bit of memory and the register that has several bits of memory in series. We should also recognize larger memory structures such as ROM (Read Only Memory) and RAM (Random Access Memory). We can think of ROM and RAM as being similar to the register in that we have a group of memory cells that are in series. However, unlike the register, RAM and ROM have a second dimension typically, which is like having an entire array of registers.

D-RAM Memory

RAM Memory, and in particular the D-RAM that is used in most modern computers, does not rely upon the D-Latch circuit to store a bit of memory. The D-Latch requires at least 9 transistors to store a single bit of memory. This number of transistors can add up quickly. Imagine a computer with 1 gigabyte of memory. A gigabyte is 1 billion (1,000,000,000) bytes, each byte has 8-bits, and using D-Latch circuits, each of those 8-bits has at least nine transistors, which totals 77,309,411,328 transistors. Additional transistors would be required for decoding and addressing, but we will not consider this now.

The alternative to the D-Latch circuit is an innovative approach that couples a transistor with a capacitor.   A capacitor is an electronic component that can store an electrical charge between two conductor plates.   In circuit diagrams with a capacitor, two parallel lines represent the two conductor plates separated by an insulator. The capacitor acts as a battery, storing an electrical charge. When reading the memory, this stored electrical charge provides the signal current. Writing into the memory is storing the electrical charge. The circuit for the D-Ram bit of memory is shown in Figure 4.5

You might notice in Figure 4.5 that the transistor looks different from the ones previously studied. This transistor *is* different; it does not have the arrow representing the emitter or the line representing the collector.

This is because D-Ram cells use a special kind of transistor called a Field Effect Transistor, which is often abbreviated FET.

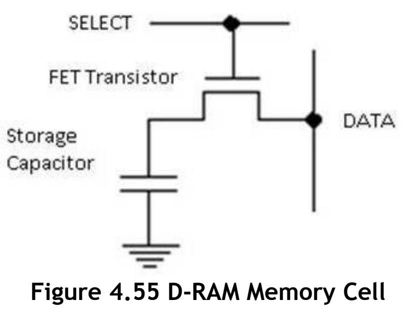


Figure 4.55 D-RAM Memory Cell

Although we will not cover it here, you might want to do some further research on the internet to understand the properties of field effect transistors.

The benefit of the D-Ram cell is clearly the reduction in the number of transistors. The D-Latch required at least nine and the D-Ram circuit requires only one.

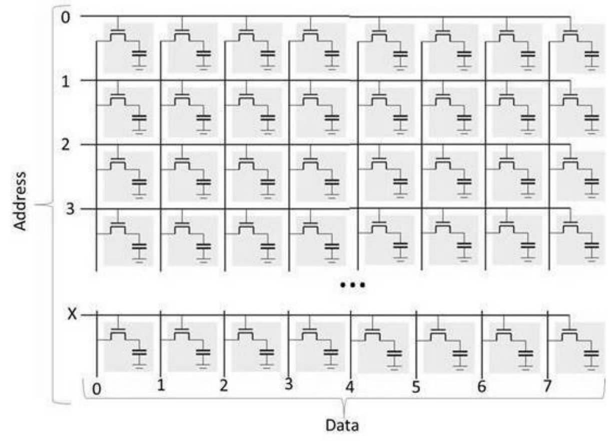


Figure 4.56 D-RAM Memory Structure

Figure 4.6) provides an illustration of the structure of random access memory. We see that the memory cells are arranged in a matrix. The dimension across is the number of bits of data. The dimension down represents locations in memory. Each row of memory cells make up a memory location. We access memory by accessing a specific row. This row in memory is the memory address. Each row represents 8-bits, or one byte of memory.

Historically, in order to save larger data structures in memory, the data would need to be separated into bytes. For example, a number that required 32-bits would have to be broken up into 4 bytes. In older computer system architectures, the order of these bytes in memory was very important.

Big and little endian

Once upon a time in the land of Endian there were two races of people. One was the Big endians and the other was the Little endians.

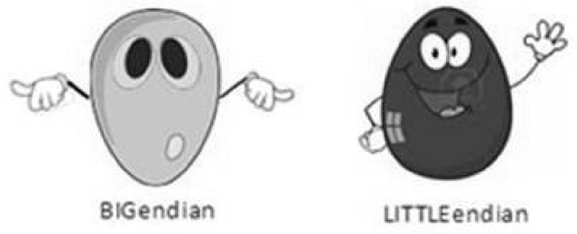


Figure 4.57 Big and little Endian

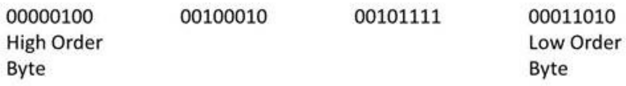
Actually, big-endian and little-endian are terms used to describe the order in memory by which values are stored. If all values that we stored in memory were only one byte, 8-bits in size, then there would not be a problem, however, that is not the case.

Traditionally, computer architectures have adopted an approach which breaks down any data item to be stored that is *LARGER* than one byte, into byte-sized pieces. In the little endian approach, the less significant byte appears first in memory and the most significant byte appears last.

For example, assume that we need to store in memory, a very large number, such as 69,349,146. This large number requires 32-bits to store, because the binary equivalent is 00000100001000100010111100011010. How would we store this number?

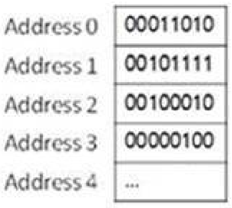
We need to take the number and break it down into bytes. The byte at the far left is called the high order or most significant byte.

The byte at the far right is called the low order or least significant byte.



00000100	00100010	00101111	00011010
High Order Byte			Low Order Byte

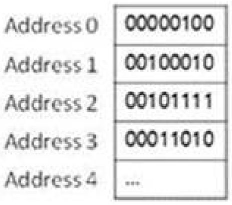
In little endian, the byte 00011010 is the least significant byte, so it will appear in memory first. When we say it "appears in memory first," we mean that it is stored at the lowest address. The address is the row in the D-RAM array where the byte will be stored.



Address 0	00011010
Address 1	00101111
Address 2	00100010
Address 3	00000100
Address 4	...

Figure 4.58 Little endian format

Big endian is the opposite. In big endian, the most significant byte appears first and the least significant byte appears last.



Address 0	00000100
Address 1	00100010
Address 2	00101111
Address 3	00011010
Address 4	...

Figure 4.59 Big endian format

This concept of endian-ness was a big issue when different formats of files or programs supported only one type of endian-ness. Most modern computer architectures no longer struggle with this issue and many can support memory stored in either direction.

If you were wondering where the term "endian" came from, it was from the author, Jonathan Swift. Jonathan Swift was a satirist (he poked fun at society through his writings). His most famous book is *Gulliver's Travels*, a story about a mythical land where certain people prefer to eat their hard-boiled eggs from the little end first (little endian), while others prefer to eat from the big end (big endian) and how this led to wars between these people. For years, the computer industry warred over big-endian and little-endian much like the characters in Jonathan Swift's novel.

ROM Memory = Program Memory

The idea of memory addressing is now simple to understand, as the memory address is simply the row of memory that we either must update or from which we retrieve a value. Typically, the address or row in memory is selected by using a decoder circuit. The decoder enables a particular row within the memory array from which only the values from the selected row of memory cells are sent to the output. This is another great example of the use of the decoder circuit or the controlled buffer to enable or disable a particular group of signal wires.

In the *Logisim* tool, we have been provided with two additional components that we will use. The first is the ROM component, which represents read-only memory and the second is the RAM component, which represents random-access memory.

In the projects specified within this text, we will use ROM as the memory where we place the instructions to be executed (the program). In many computer systems, an area of RAM is actually mapped to be ROM, or more accurately, is the place where our program instructions are stored. As we progress through the text, we will develop an understanding of how the program counter is used to point to the next instruction that the computer must process. In our simple computer system where we have all of our instructions in a ROM memory model, this is quite simple to implement. However, in the typical computer system with an operating system and multiple programs, this becomes more complex.

One of the roles of the operating system is to help coordinate the flow of instructions to be processed. When memory is mapped to be ROM, or the location of program instructions, an *offset* is applied to all of the instructions that utilize addresses. Instead of going to line 5 in memory and executing the instruction, since the program may start at position 22000 in memory, the instructions and addresses will be remapped to start at 22000. The instruction to go to line 5 would become an instruction to go to line 22005.

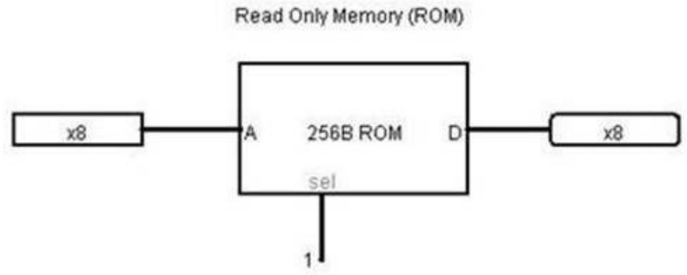


Figure 4.60 Read Only Memory

The *Logisim* component for ROM memory is rather simple as we can see in the above diagram. It supports one input and one output. The input is an address (A on the component) which simply points to the row of memory bits that should be enabled to send output. The address is nothing more than a simple offset. The first row has an address of 0, the second row an address of 1 and so on.

The output (D on the component) is the bus that contains the value in ROM memory that is pointed to by the address. We will learn in subsequent chapters that this ROM address is typically tied to a structure called the program counter and the combination of the program counter pointing to the ROM address is the mechanism used to fetch the next instruction from ROM memory to execute.

RAM Memory = Data Memory

The RAM memory component is a bit more complicated in that not only do we need to be able to address and read the RAM memory much like we have seen with the ROM memory, but we also need the ability to load values into memory and store them.

The RAM memory is where we store data values.   The following diagram details all of the inputs and outputs of the RAM memory component that is available in *Logisim*.

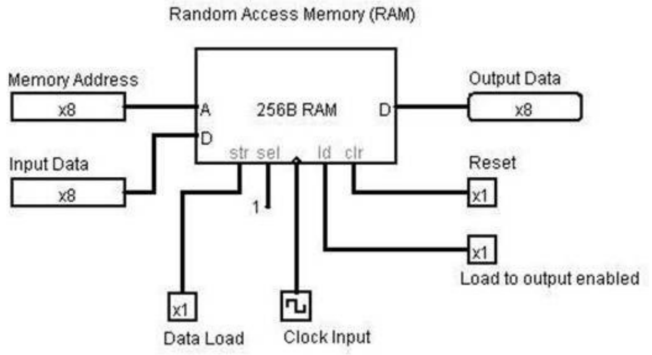


Figure 4.61 Random Access Memory

For inputs, we have a memory address, input data, data load, clock input, reset, and load to output, which operate as follows.

The memory address for RAM works the same way as the address did for ROM. This is simply a bus that contains a binary number. A decoder in the RAM component will select one of the rows in the array of memory cells to activate, based upon this address. In the ROM component however, we could only read the memory that was in the component, while in RAM, we can either read the value already in memory or change the value.

The input data is a bus that contains a binary number to be stored into memory. The value will be stored into the memory location pointed to by the memory address. Data is stored into RAM only when the Data Load input has a value of logical 1 and when the clock input is cycled (moves from logical 0 to logical 1).

The clock input is typically tied to the system clock. Memory values can only be retrieved (loaded to output) or updated during a clock cycle change.

The reset input performs the same function for RAM as it did for memory bit cells and registers when it has a value of logical 1, the contents of memory are cleared.

The load to output enabled controls *WHEN* a value in memory can be made visible as output data. This can be an important functionality in terms of controlling when data signals are moved within a computer system. We will find that this is important as we begin building our own control system.

Binary Counter Circuit

We briefly mentioned how we use the output of the program counter as the address to ROM memory to control the execution of instructions in a computer system. If we think of ROM memory as an array where each row in the array is an instruction to be executed, then executing a program is as simple as sequentially pointing to each row, fetching the instruction from ROM memory and executing it via the ALU.

However to do this, we need an important circuit, which is the program counter. The program counter does not use the adder circuit. This might be a bit surprising, but there is actually a more efficient way to count in our computer system.

The binary counter uses the principal that each D-Latch circuit stores both a number and its complement. If we take the complement from a memory bit and feed it back into the data input of the memory bit *AND* use this same signal to toggle the input on the next memory bit in sequence, it has the effect of counting in binary.

The circuit for such a binary counter is shown in the following diagram (Figure 4.12).

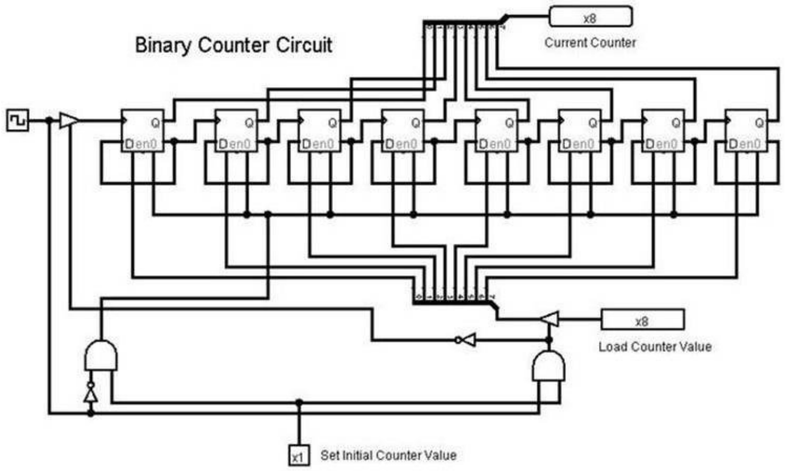


Figure 4.62 Binary Counter Circuit

Notice in the circuit that the system clock is used as an input to the binary counter. This means that every time the clock executes a cycle (moves from logical 0 to logical 1); the counter will be incremented by one.

The output of this counter (current counter value) is sent via a bus to the address register for the ROM memory creating a simple control system to execute instructions on our computer. You might be asking the question "what happens if I need to loop or branch?"

That is a good question, and it has a relatively simple answer. If the counter simply points to an instruction in ROM memory (one of the rows) and if we could put a new value in the counter, then it would point to a new address, allowing us to *jump* to any location we within our program. One thing we could do with this is implement a loop or a branching circuit. In figure 4.12, we see how the ability to load a new value into the counter is implemented. The new value is set from the load counter value. This value would typically come from the jump address register. These signals are directed to the data input on the D-Latch. You will notice that we use a couple of AND gates and NOT gates which essentially disable the clock while the new value is being loaded and enable the signal on the D-Latch to load a new value.

Divide by Two Circuit

We have discussed circuits that all operate within a single clock cycle. We will be building and discussing a computer system that is designed to be able to complete the fetch-decode-execute-store stages within a single cycle of the clock.

Most modern computers, however, are not designed to have such a simple single cycle design. In many cases, there are operations that may span two, four, or more cycles.

We need to be able to control and manage the timing of these types of computer systems and one technique that we can employ is the use of divide-by-two circuits.

The divide-by-two circuit is designed to cut the clock frequency in half. By cutting the frequency in half, it takes twice as long to complete one cycle. Consider the following circuit example:

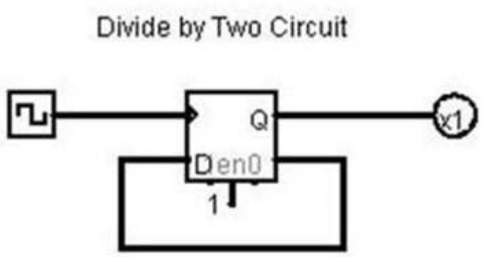


Figure 4.63 Divide by Two Circuit

Notice how we are using a D-Latch and we are feeding the complement value (complement of the value stored in the memory cell) back to the data input value. If you run this circuit using *Logisim*, what you will see is that this feedback loop will only allow the output value x1 to be logical 1 every second clock cycle. You should recognize that this circuit operates on the same principle as the counter circuit.

We call this a divide-by-two circuit because it reduces the frequency of the clock signal by ½. We can create other clock cycle frequencies by feeding the output of the circuit into another divide-by-two circuit to divide-by-four; adding another circuit will divide by eight, and so on.

By using the divide-by-two circuit, we can create a timing signal that will generate a logical 1 as output, every 2, 4, or more clock cycles.

Chapter 4 Exercise

For the Chapter 4 exercise, you must create a circuit using *Logisim* that implements a memory register capable of storing a 4-bit binary number. Your register circuit must be able to support the inputs detailed in the following diagram:

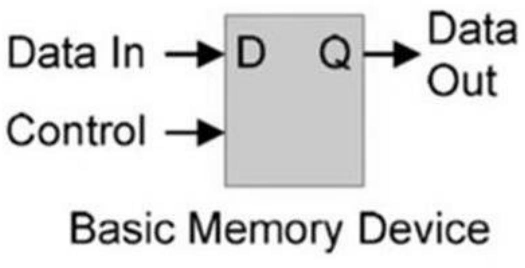


Figure 4.64 D-Latch Memory Cell

Each bit of the register circuit must support a data in, a data out, and a control. The control functions such that the data value will only be changed when the control bit is toggled on.

The value in the register must be persistent and can only be changed when the control input has been toggled (cycled from logic 0 to logic 1).

You should develop and test the 4-bit register using *Logisim*. *YOU CAN ONLY USE THE BASIC GATES* including AND, OR, NAND, NOR, NOT, Exclusive OR, and Exclusive NOR when building your circuit.

When you have successfully developed a functioning 4-bit register, you should duplicate the circuit and add it to the two inputs of the ALU circuit that you developed as part of the assignment from the previous chapter.

(This page intentionally left blank)

Control System: Clock, Counter, and NPZ

In this chapter, we will begin to put together the components of a computer architecture. We will look specifically at elements of the control unit such as data and signal busses, the system clock, program counter and the N,P,Z functionality, but before we can get into those topics we need to develop an understanding of what a computer architecture is and in particular the von Neumann architecture.

Introduction to the von Neumann architecture

John von Neumann and the computer architecture that bears his name incorporate the basic idea of a flexible general purpose computing device (computer) where both instructions and the data they operate upon are stored in memory. This is called a *stored-program* computer. The basic von Neumann computer had memory, an ALU (arithmetic logic unit), and a control unit, to control the execution of a program loaded into memory. It is the responsibility of the control unit to fetch instructions from memory and load them into the ALU for processing.

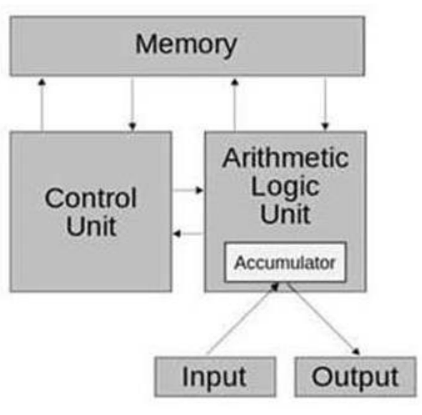


Figure 5.65 von Neumann Architecture

The arrows in von Neumann's architecture refer to the data and signal busses that the control unit uses to control the execution of the computer.

The von Neumann architecture was a revolutionary idea because it made possible modern general purpose computing. Prior to the development of the von Neumann architecture, all computing devices had to be hard wired to perform a specific task. The innovation of von Neumann was the use of memory and a control system so that both instructions and data could be stored and retrieved from memory and sent to the appropriate place for processing along a data bus. The idea that the program or the set of computations that the computer performed were simply instructions that were loaded into memory and could be retrieved automatically using signal and data busses, transformed the computer from a specific purpose device capable of a single computational task into general purpose computing devices that could be programmed to complete any computing task.

Data and Signal Busses

One of the key innovations introduced by von Neumann was the data and signal busses to move data and instructions around the computer architecture. The basic difference between a data bus and a signal bus is in how they are used. A data bus is a group of 1 or more signal wires used to carry the data of the computer system between memory, registers, the ALU, and input/output devices. Data busses typically have the same number of bits (wires to carry a signal value) as the inputs to the ALU, registers, and word size in memory. A computer system that has an ALU designed to process 32-bit words will need to have a 32-bit data bus.

Perhaps at this point it might make sense to define a word as it relates to computer architecture. A word is the largest unit within a computer system comprised of bits. It is the number of bits used in a particular operation and transferred using a bus. For example if the ALU of a computer system operated using a 32 bit operand, then the word size of the system would be 32 bits.

Consider the following diagram. In this diagram, we see a simple example of ROM memory, and Instruction decoding. Not only can we learn how to fetch an instruction from ROM and decode it from this circuit, but we also see examples of both Data and Signal busses.

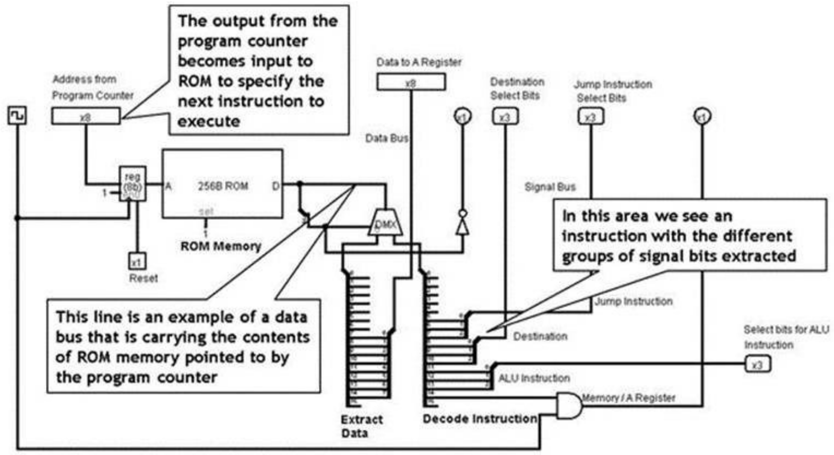

The output from the program counter becomes input to ROM to specify the next instruction to execute
Address from Program Counter
Data to A Register
Destination Select Bits
Jump Instruction Select Bits
Data Bus
256B ROM
ROM Memory
Reset
Signal Bus
In this area we see an instruction with the different groups of signal bits extracted
Jump Instruction
Destination
Select bits for ALU Instruction
ALU Instruction
Memory / A Register
This line is an example of a data bus that is carrying the contents of ROM memory pointed to by the program counter
Extract Data
Decode Instruction

Figure 5.66 Instruction Fetch and Decoding

We see in this circuit, that the data bus carries a value in binary. The size of the bus (number of bits) is typically defined by the architecture of the ALU. In this case, the size of the ROM is 16-bits wide. Each *row* in the ROM memory contains a 16-bit number that we can fetch. We also can see that this binary number must be *decoded* in order for us to use it. Decoding simply means that we need to extract the information in the binary word to set the correct signals to process the instruction.

In this example, which is the architecture to which we will be building our computer systems, each location in ROM can be either a data value or an instruction. We can see that each is processed differently. Data values are extracted and sent to the A register (we will see how to integrate registers into this shortly), while instructions are decoded and used to control the computer's processing.

The term decode is used because most computer systems use decoder circuits to extract the information from an instruction for processing. To simplify the circuit in the above example, we are using a fan-out. The fan-out takes the bus, which appears as a single line, then splits out each bit (wire) that is in the bus.

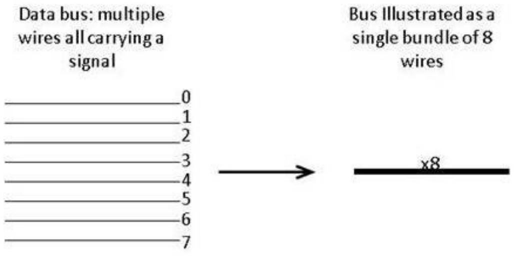

Data bus: multiple wires all carrying a signal
0
1
2
3
4
5
6
7
Bus Illustrated as a single bundle of 8 wires
x8

Figure 5.67 Bus Illustrated as Single Wire

We see from the circuit that when we have split out the relevant bits, we use those bits as control signals (signal bus). Examples in this circuit of signal busses include the bits that are selected out to specify the jump instruction, destination, or the ALU instruction.

If you recall in the previous chapter where we used a de-multiplexor to select which operation (Add, Subtract, AND, OR) the ALU would perform, we now see where those select bits to choose the operation come from. In this case, we extract the bits from an instruction fetched from ROM and extract and decode the correct bits of data to create a signal bus.

The System Clock and Program Counter

The system clock coupled with the program counter, form the heartbeat of the computer system, which brings it to life. As we have already learned, the system clock is not a clock in the sense of a device that keeps track of and communicates time in terms of hours, minutes, and seconds, but rather is a device that generates a digital signal cycle that occur during regular periodic intervals.

We use this signal for a number of purposes within the computer system, the most important of which, is the program counter. In the following diagram (figure 5.4), we see a simple circuit that combines a system clock with a counter. Each cycle of the clock sends a signal to the counter, which causes the counter to increment by one.

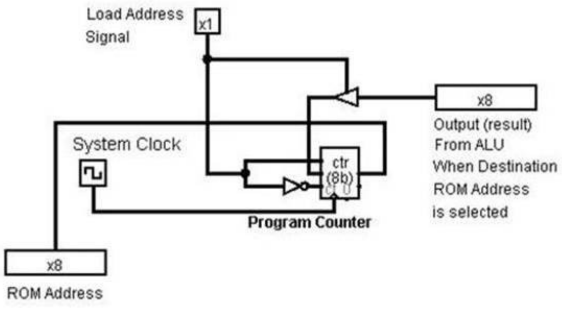
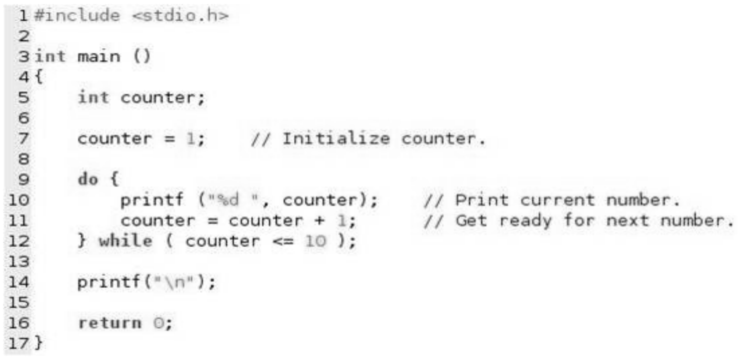


Figure 5.68 System Clock and Program Counter

The counter, which in this case we are calling the program counter, sends the output value of the counter to the ROM memory as its address.



```
 1 #include <stdio.h>
 2
 3 int main ()
 4 {
 5     int counter;
 6
 7     counter = 1;    // Initialize counter.
 8
 9     do {
10         printf ("%d ", counter);     // Print current number.
11         counter = counter + 1;       // Get ready for next number.
12     } while ( counter <= 10 );
13
14     printf("\n");
15
16     return 0;
17 }
```

Figure 5.69 Short Program in the C language

This address in ROM points to each instruction that the computer must execute in sequence. Imagine the following short program (Figure 5.5).

It is easy to imagine the program counter beginning at 0, receiving the clock signal and incrementing to 1. The program counter now points to the first instruction in ROM, which is then executed. As the counter increments to 2, 3, 4, 5, and so on, each line of code is executed.

Of course, our computer system will be executing machine language instructions and *NOT* the C code that we see in this example, but it does help us to understand how the program counter works to facilitate the execution of a program.

Looking at the code in the preceding figure (Figure 5.5), we see that there is a loop in lines 9 through 12 of the code. You might be wondering how our simple program counter circuit can accommodate a loop or conditional logic, such as an if-then statement.

The answer is rather simple. A loop is nothing more than causing the program counter to be set back to some value. If our program reached the end of the loop in line 12 and needed to go back to line

9 and execute the loop again, all that we would need to do is load 9 into the program counter.

We see how this can be accomplished in the circuit (Figure 5.4) by simply sending a value to the counter circuit, in this case the ROM address or the line in ROM memory to which we need to move. In order to load a value into the counter we need to send a signal to load and not increment, which is what the load address signal does. We see that if the load address signal is logic 0 (meaning that we do not want to load an address now), we are sending a value of logic 1 using an inverter to the signal that enables the counter to increment. Otherwise, if the load address signal has a value of logic 1, then the load signal on the counter circuit is enabled, and the increment is disabled allowing a new value to be loaded into the counter.

Using De-Multiplexors, Multiplexors, and Decoders for Control

One of the important innovations introduced by von Neumann was the use of data and signal busses to control the flow of data and signals. Of course these busses are subject to the issues that we discussed in chapter three where we cannot have two inputs active to any input of a gate and that we also must avoid the situation where a loop is formed where the output of one gate can also become the input to the same gate

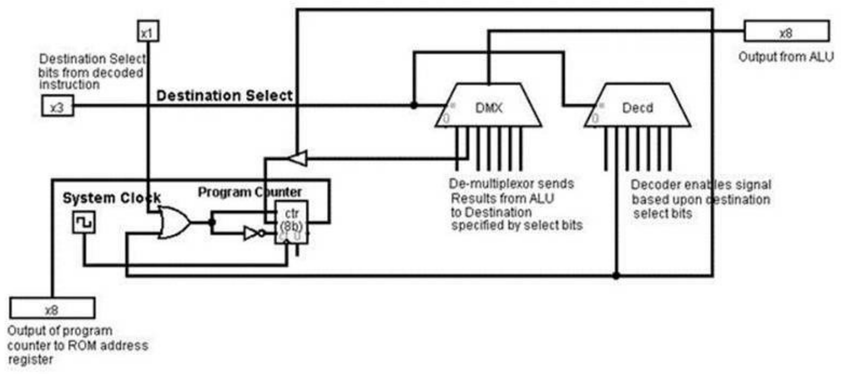

x1
Destination Select bits from decoded instruction
x3
Destination Select
x8
Output from ALU
DMX
Decd
De-multiplexor sends Results from ALU to Destination specified by select bits
Decoder enables signal based upon destination select bits
System Clock
Program Counter
ctr (8b)
x8
Output of program counter to ROM address register

Figure 5.70 Control Circuits using De-multiplexor and Decoder

Implementing N,P,Z functionality

If you are wondering what N,P,Z functionality is, let's begin by defining N,P,Z as Negative, Positive, and Zero. We have been studying the various circuits and components that together make up a computer system. We have discovered that it is relatively easy to add numbers together in binary using logic gates, and we have discovered that since addition is the basis of other mathematical operations such as Subtraction, Multiplication, and Division that our ability to do ***addition*** provides us with the ability to compute almost anything.

What we have *NOT* seen so far is how we can accommodate the idea of conditional logic or expressions within the design of our computer system.

We all know from our experience with programming languages that we need to be able to evaluate conditional expressions. For example, most programming languages will have an if statement that looks something like the following:

```
If (a < 5) then
    a = a+1;
else
    a = a-1;
```

This is a classic example of a conditional expression. If the value contained in variable ***a*** is less than 5, we want to compute one thing, and if the value is equal to 5 or greater than 5, we want to compute something else.

The question is, how do we create a circuit that can help us to evaluate this expression?

The answer is that we cannot. No circuit can evaluate this kind of conditional expression. What we need to do is change this problem into something that we can do well with our computer system—Math.

Let us assume for a moment that the value contained in the variable ***a*** was 4. One way that we could determine if 4 was less than 5 would be to subtract 4 from 5 and if the resulting value was a positive number, then 4 would be smaller than 5. If the result of subtracting the value of ***a*** from 5 was 0, then we would know that the value in ***a*** would have to be *EQUAL* to 5, and likewise, if we subtracted the value of ***a*** from 5 and the result was negative, then we would know that the value of ***a*** was larger than 5.

We have just discussed three cases where we evaluated expressions and based upon whether the result was Negative, Positive or Zero we could take the appropriate action. This is the purpose of the N, P, Z flags.

Considering our if-then-else statement, the way to implement it using our ALU and control unit design, is by taking the expression to be evaluated, turning it into a mathematical operation and, based upon the N,P,Z of the result of this operation, we would jump to the appropriate place in the code to continue executing the program.

In our program, the “then” portion of the code would be at one memory location and the “else” part of the code would be at another, and we would simply use our ability to load the appropriate location into the program counter to “jump” to that location, thereby implementing a conditional expression.

Now that we understand *WHY* we need to have N,P,Z flags and how we can use them, we now need to learn how to create these flags.

Consider the following circuit diagram (figure 5.7). In this circuit, we have included the ALU and the gates that we are using in our example, UoPeople computer system to implement the N,P,Z functionality. You will notice that the entire N,P,Z system is used only to determine when to execute a *jump*. A jump is nothing more than sending a new address to the program counter to move to a new position (row) in ROM memory and using this as the next instruction to execute.

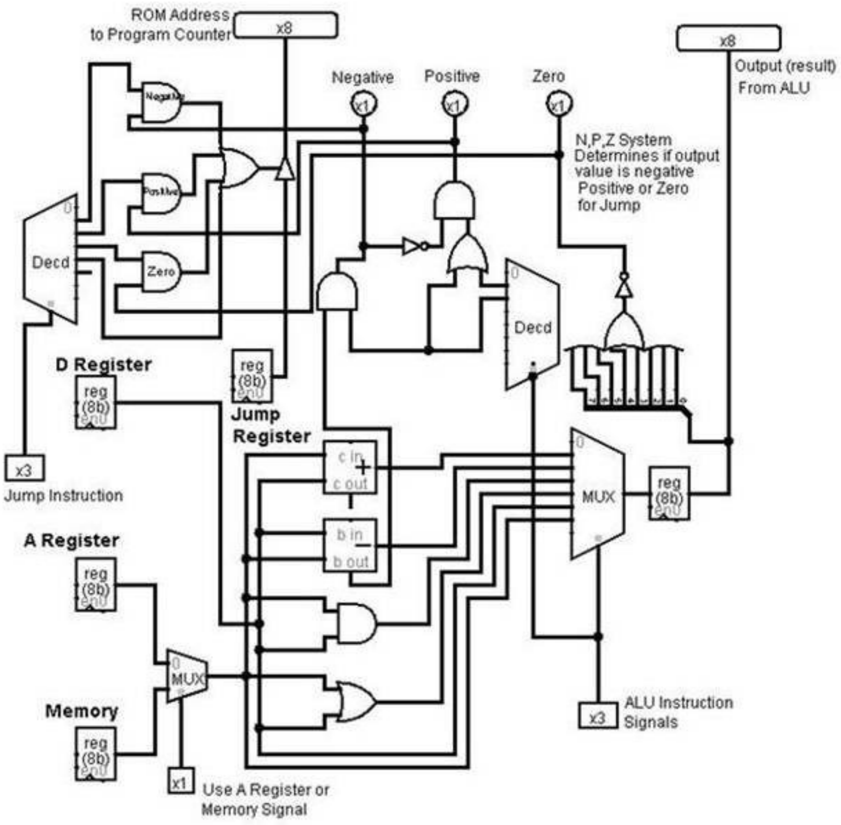


Figure 5.71 N,P,Z, and the ALU

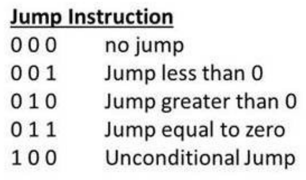
You will notice that there are two distinct types of N,P,Z functionality. On one hand, we have the functionality to determine if the result of a computation in the ALU is Negative, Positive or Zero. Notice that we use a decoder circuit to enable checking for Negative or Positive results only when the operation of the ALU is either ADD or SUBTRACT.

You will see that we take the carry bit signal from the Subtract gate to check if the result is negative. Remember from chapter 2, that we discovered that when subtracting two binary numbers using 2's complement that the value of the carry bit tells us if the result is either negative (Logic 0) or Positive (Logic 1). This carry bit is used in the subtract gate to generate the signal value that is the negative flag. If the result is not negative then an inverter creates the positive flag.

For the zero flag, we take the output of the ALU into an OR gate and invert the output of this gate. If any bit into the OR gate is logic 1 then the zero flag is NOT set.

The second type of N,P,Z functionality is represented by the decoder and the AND gates in the upper left hand portion of the circuit. What this portion of the circuit is doing is implementing a jump based upon the N,P,Z flags.

The decoder in this circuit is decoding the jump instruction bits, which are as follows:



Jump Instruction	
0 0 0	no jump
0 0 1	Jump less than 0
0 1 0	Jump greater than 0
0 1 1	Jump equal to zero
1 0 0	Unconditional Jump

When you examine the circuit you will notice that the 0 output on the decoder is not used, which means that if 000 is specified, no jump will occur.

The next three outputs from the decoder are interesting. The first output (select bits 0 0 1) or position 1 on the decoder, becomes the input to an AND gate and a second input into the gate comes from the Negative flag.

This operates as follows. When the select bits for the decoder are 0 0 1, which we can see is defined as jump less than 0, then output 1 on the decoder will have a value of logical 1. If the Negative flag is also set (has a value of logical 1) then the bus that takes the value in the jump address register is enabled and the address is sent to the program counter effectively causing it to point to a new location in ROM memory, in other words, executing a jump. The AND gate evaluates the condition that both the jump less than 0 signal has a logical 1 and the Negative flag has a logical 1.

We see a similar design for select bits 0 1 0, or position 2 on the decoder, which are ANDed with the Positive flag implementing the "jump greater than 0" instruction.

We also see that when the select bits are 0 1 1, (position 3 on the decoder) the decoder output is ANDed with the Zero flag implementing the "jump equal to zero" instruction.

The final output from the decoder has no conditions. If the select bits are 1 0 0 (position 4 on the decoder) for Unconditional jump, then the address in the jump address register is sent to the program counter. This final jump instruction is called an unconditional jump because it will always jump, as it has no conditions on it.

Chapter 5 Exercise

For the Chapter 5 exercise, you will begin by downloading the provided circuits (if you are enrolled in the CS1104 course) or constructing them by referring to the figure in which they appear:

- the *Logisim* ALU (Arithmetic Logic Unit) circuit file (Figure 5.7)
- System Clock and Program Counter Circuit File (Figure 5.4)
- Instruction Decoder Circuit File (Figure 5.2)

As part of this assignment, first determine how to integrate the functionality of the Instruction decoder circuit with the ALU circuit and System Clock and Program Counter circuit.

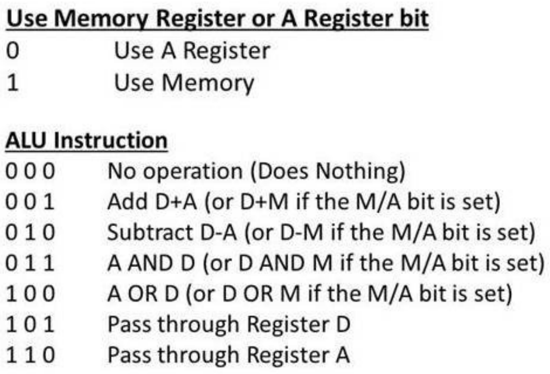
Next, develop a circuit that implements the N,P,Z functionality.

Implement both a ROM memory and a RAM memory

Finally, ensure that all of these circuits are integrated together and controlled through both data busses and signal busses.

Your system should be able to implement the following machine instructions, which means that your decoder circuit must be able to leverage the following signals extracted from a 16 bit word (from the ROM memory component) and used to implement destination functionality, jump functionality, and the ALU instructions.

Keep in mind that the first bit of each instruction will specify if the instruction is an A instruction meaning that it simply loads a value into the A register or a C instruction which specifies a computation to be performed.   You can look ahead to the next chapter if you want more details.



Use Memory Register or A Register bit

0	Use A Register
1	Use Memory

ALU Instruction

0 0 0	No operation (Does Nothing)
0 0 1	Add D+A (or D+M if the M/A bit is set)
0 1 0	Subtract D-A (or D-M if the M/A bit is set)
0 1 1	A AND D (or D AND M if the M/A bit is set)
1 0 0	A OR D (or D OR M if the M/A bit is set)
1 0 1	Pass through Register D
1 1 0	Pass through Register A

Destination

0 0 0	No operation (does nothing)
0 0 1	A Register
0 1 0	ROM Address (points to location in ROM memory)
0 1 1	D Register
1 0 0	Memory (memory location in RAM)
1 0 1	RAM Address (points to location in RAM Memory)
1 1 0	Jump Address Register (sends to register which contains the jump address that will be used in a subsequent operation)
1 1 1	Output Register (the register for output where we can send the output of a computation to examine)

Jump Instruction

0 0 0	no jump
0 0 1	Jump less than 0
0 1 0	Jump greater than 0
0 1 1	Jump equal to zero
1 0 0	Unconditional Jump

Computer Architecture

We all know some of the common computer architectures. Most of us recognize names such as Intel, Advanced Micro Devices (AMD), ARM, SPARC, Integrity, Motorola, or Power. Each of these CPU architectures defines the architecture of a computer system. In this course, we have been working towards building the University of the People (UoPeople) architecture, which is a simplified CPU architecture that supports a simple machine language.

At this point, we have either studied or created the circuits using *Logisim* that implement core functionality including:

- the ALU (Arithmetic Logic Unit)
- the N, P, Z flag system and the circuits to implement jump capabilities
- the system clock and program counter, the ROM memory system and an instruction decoder
- the multiplexors, de-multiplexors, and decoders necessary to direct the results of the ALU to a destination such as memory, RAM address, ROM address, registers, or the jump address register

All of these assignments and learning exercises have been building to the point where we put these elements together into a complete computer architecture, which of course we have called the University of the People architecture (After all we would not want Intel, HP, and ARM to have ALL the fun).

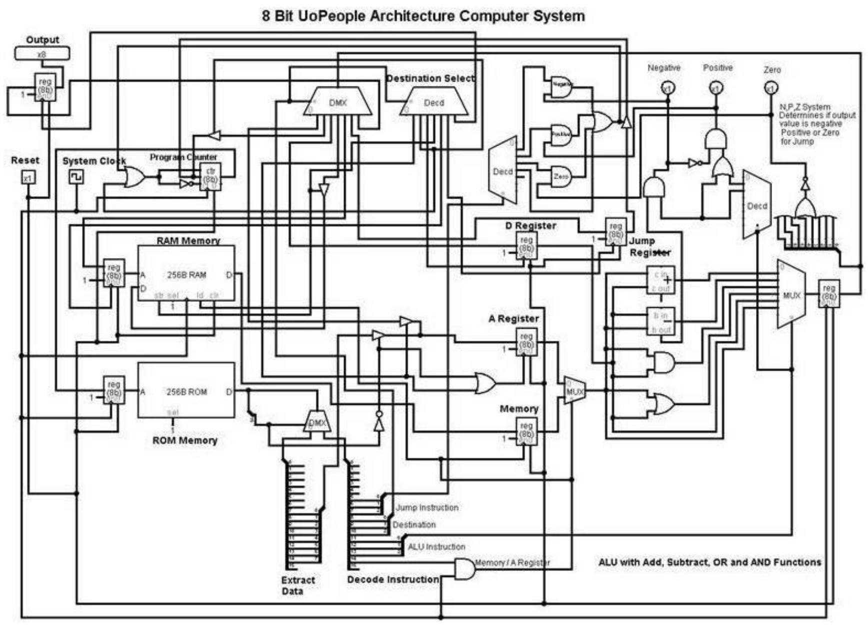


Figure 6.72 Complete Circuit for UoPeople Computer System

In the preceding diagram (Figure 6.1), we see a completed circuit that takes these elements and puts them together into a complete computer architecture. The basic idea behind this architecture is that it fills in the details to show how we get from Adder Circuits, Memory Circuits, and Registers. It also helps in understanding multiplexors, de-multiplexors, and decoders, by putting these elements together with the clock and program counter circuits, the N, P, Z functionality, and the rest of the essential control circuitry to create a functioning computer.

Some of these circuits should be familiar, as we have reviewed them in previous chapters. Others might be new. Together they form a functioning computer system that can be used to develop and execute programs.

In this chapter, we will be learning a couple of important concepts. First, we will learn to distinguish which circuits are typically included within the CPU (Central Processing Unit) and which are a part of the remaining *chip set* of the computer architecture.

Further, we will begin to understand how this circuit functions to execute general-purpose programs by exploring how each element of the Fetch-Decode-Execute-Store cycle is implemented within the circuit.

Finally, we will establish the foundation that will enable you to build *YOUR OWN* computer system circuit. Although it would be possible to recreate the circuit that appears above, it is recommended that you attempt to wire up your own computer system using the circuits provided and the knowledge that you have gained so far. Doing this will enhance your understanding of exactly how this computer system functions as well as how every von Neumann based computer architecture functions.

The University of the People Architecture

To get us acquainted with the various components of a computer architecture, we will examine a few elements of the UoPeople architecture using color codes. For those who are reading this text on a device that only supports greyscale, each color has been labeled so that you can identify the sections.

In the diagram below (Figure 6.2), we see a section colored in light grey. This section contains two things, the system clock, and the program counter. We have already discussed the role of the system clock and we know that it produces a digital signal cycle on a regular periodic basis. During each cycle, the clock will change state from 0 to 1 and then from 1 to 0. You will see that there is a need for such timing.

Some of the registers that are being used need to know when to update the value in the register, and the clock signal (when the value goes from 0 to 1) is used to signal the register to load a new value. The RAM component, the ALU output register, and the ROM address register in particular, rely upon the rising and falling signals of the clock

You might also notice that there is an output register in this grey shaded area. Essentially this register demonstrates the integration of input and output into a computer system. In this case, we can send a value that has been computed to the output register where it will be displayed until it has been changed by moving another value into the register. This output device is useful when we want to compute some value and then see what the result of the computation is, to ensure that the computation was performed successfully.

You will notice that this output register along with the A, D, and, Memory registers are not controlled by the clock as we want to make sure that values are loaded into the registers to be consistent with the processing of instructions. These registers are triggered when the clock input on the register (the small triangle on the bottom of the component) is toggled from Logic 0 to Logic 1. By controlling the *clock update* on the register, we can control when to load a new value.

If you study the circuit carefully, you will notice that we are using signals from the decoded instruction and in particular the *destination* bits of the instruction. When a particular register or memory is the destination for the output from the ALU, we send the data via a data bus to the input of the register and a signal from the decoder to the clock input on the register component to update simultaneously, the value in the register. This offers a great example of the use of both data and signal busses in our computer architecture.

The RAM and ROM memory components are located in the green shaded area. You will notice that we are using a register to hold the address of each type of memory so that it is consistent between clock cycles. Essentially this register ensures that the address for the memory component will not change until a new address has been set.

You will see that the address to the ROM memory is actually just the output from the program counter. The ROM is the memory that contains our program instructions.

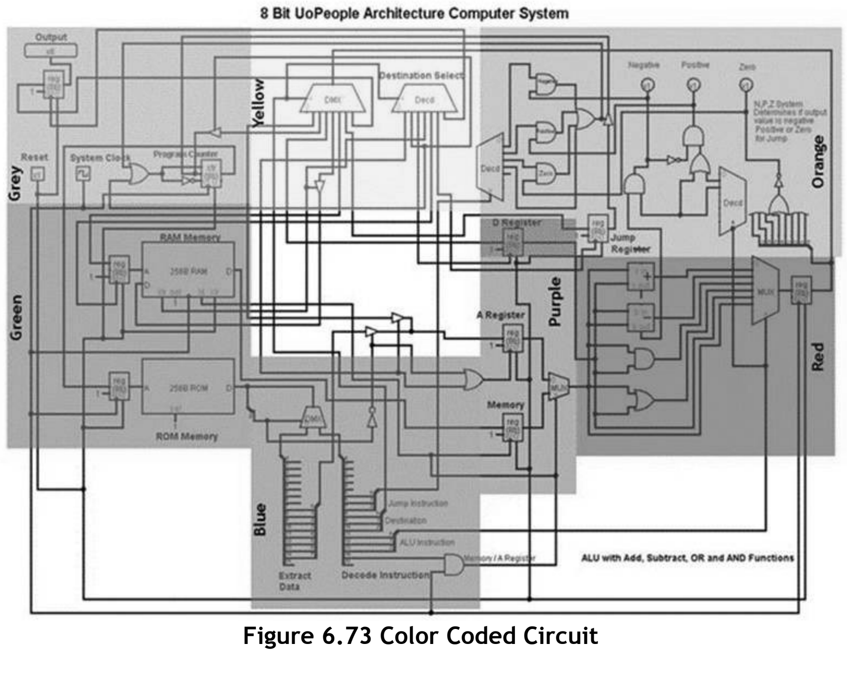


Figure 6.73 Color Coded Circuit

The program is created by taking the machine instructions that the computer understands (in binary) converting these binary instructions to a hexadecimal format and putting them in a file that can be loaded into the ROM memory.

Speaking of machine instructions, the section in blue is the functionality that we have in the computer to determine if the current instruction fetched from ROM is an A instruction or C instruction and what to do with it.   In the case of the A instruction, we can see that we strip off the 8-bits of data following the first bit and send it to the A register.   We use the fan out in *Logisim* to do this, which allows us to break out the bits and decide what to do with each bit or group of bits.   In the case of the C instruction, we can see how we split out the bits for each component of the instruction (M/A, instruction, destination, jump) and use these groups of bits as the signal busses to control the ALU instruction, destination decoder, and jump functionality.

It is interesting to note that if you look at a magnified view of a modern CPU (Figure 6.3) you can begin to see the similarity with our color-coded circuit in Figure 6.2 above. We can distinguish the CPUs, which contain the ALU, registers, and control unit, we can also see cache memory, and other I/O devices. We can also see the wires connecting the different components together and these wires form the basis of both data and signal busses within the CPU. These busses are extended out of the CPU in the computer system and implemented using the rest of the chip set, however, this view does help us to understand that what we see in the Figure 6.23 circuit is implemented in an actual modern CPU.

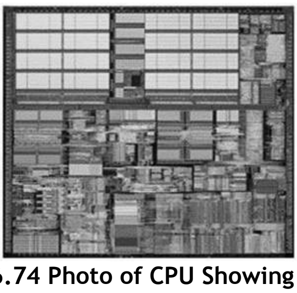


Figure 6.74 Photo of CPU Showing Circuits

The destination bits from the blue section are split out and sent up to the section in yellow where we have a de-multiplexor and a decoder. These two components work together and what they do is take the data coming from the output of the ALU and direct it to the proper destination. The de-multiplexor simply directs the data down different paths while the decoder provides us with an ability to turn circuits on or off to receive the data. An example of this is that the decoder might send a value of 1 to the clock input on the D register because this tells the register to update the value. The de-multiplexor sends the value and the decoder sends the signal that enables the value to be updated.

The key registers in the computer system,, which include the A register, D register, and the Memory register are in the purple section. A register for memory is not necessary, but it makes the operation easier to understand by staying consistent with the A and D registers. All operations performed by the ALU will operate on the D register AND either the A register or the Memory Register. The second bit in the machine instruction determines if the A register or Memory Register will be used.

The section in red should be familiar because it is the ALU and it should look like the ALU that you designed as part of the assignment in chapter three. You will see that our ALU has four functions including ADD, SUBTRACT, bitwise AND, and bitwise OR. The ALU also supports two pass through operations that will allow the contents of the D register to pass through the ALU or that will allow either the Memory Register or the A Register to pass through the ALU and to the destination specified by the destination select bits.

Finally, the section in orange implements the N, P, Z functionality for our system. The way that we can implement conditional logic within our computer system is with N,P,Z. These stand for negative, positive, and zero. Assume that we want to compare two numbers. The first number is larger than the second is and we want to branch to a location in our program. We implement this is by subtracting the first number from the second. Consider the following example. To test if 5 is greater than 4, subtract 4 from 5 and if the output of this operation is positive, then the expression is true; if the output is 0 or negative, then the expression is false.

Using this process of arithmetic with two numbers and with the knowledge of whether the output is Positive, Negative, or Zero, we can take action. This action is implemented with the jump functionality. What happens is that we load an address into the jump address register, test our number, and based upon the positive, negative, ,or zero outcome of that operation, we can implement a jump.

Implementing the Computer Architecture

A complete computer system is built from the CPU, Memory, and the supporting components that make up the chip set of the computer system. Many of the components that we have been designing reside directly within the CPU. Other components, such as RAM Memory, ROM
Memory, input and output devices, and elements of the bus exist outside of the CPU. In the following section, we will identify those components that are typically found *WITHIN* the CPU.

The CPU or Central Processing Unit is the heart of any computer system. The CPU typically has the ALU or Arithmetic Logic Unit, Control Unit, Instruction Decoder, and Registers. Many modern CPU's also include components that we have not explored such as Cache Memory, the Stack and a variety of registers that we have not implemented within our computer system.

The stack is a useful structure and one that we will explore in a bit more detail in this chapter. To summarize the stack, it is RAM memory dedicated within the CPU and its purpose is to provide a place to temporarily store some work while the processor completes some other work.

The Cache Memory has become a common element of all modern CPU's. We have previously discussed the fact that in many modern computer systems, some operations may require more than one clock cycle to complete. For example, it can often take more than one clock cycle to fetch data from memory. The cache is a very fast form of RAM memory that is located right on the CPU chip. Access to the cache memory tends to be much faster than access to the main memory of the computer. The way that the cache memory is used is that the cache might be loaded up with the instructions that need to be executed or the data that is required in an operation. Once the instructions are loaded to the cache, the CPU can access them much faster (often within a single cycle) and this improves the overall performance of the CPU and the computer system.

ALU Arithmetic Logic Unit

We have already described the ALU as the heart of the computer system. Much like the heart is the pump that gives us life by pumping blood through it and throughout the body. The ALU is the heart of the computer system because all data flows through it and to the rest of the computer system. Some of the data flowing through the ALU is subjected to computation and we have implemented an ALU circuit that is capable of computing Add, Subtract, Bitwise AND, and Bitwise OR operations.

Modern ALU designs will incorporate additional, but similar functions. For example, the ALU could implement an inverter, which inverts all of the bits of an input number. Perhaps one could add a circuit that increments or decrements an input value by 1. Others may implement various comparative circuits. While all of these features are possible, the core capability of the ALU is the Add operation.

In the project from chapter three, we implemented a relatively simple ALU design in which we use signal bits to enable a single operation against one or both input operands. An alternative to this simple approach that many ALU designs employ is to use consecutive operations. For example, the ALU might add the two input operands and then have the ability to invert the result or perhaps decrement the result or perform any of a number of operations upon the results of the basic operation. In these cases, the ALU design can be thought of as a processing pipeline where a number of computations are applied within the ALU, and not the single operation that we have modeled in our ALU design.

Control Unit

As you may have realized at this point, the control unit is not a specific component within the computer architecture but rather the collection of busses and circuits that control the operation and execution of the computer system.

The control unit is what coordinates the Fetch-Decode-Execute-Store cycle within the computer system. The control unit coordinates the fetch of an instruction, and decoding of the instruction, which is simply the process of setting the appropriate control signals to facilitate the execution of the instruction. Included in the execution of the instruction is the need to activate the appropriate function within the ALU, send the control signals and data to load registers, direct the output of the ALU computation, and enable the execution of a jump operation when required.

Instruction Decoder

Instruction decoding is a complicated sounding term, but in practice, it is quite simple. Our computer system instruction decoding is accomplished by splitting out the various control bits or groups of control bits that select the function within the ALU, determining the use of the Memory register or the A Register, directing the result of the ALU computation to a destination register, and enabling jump capabilities.

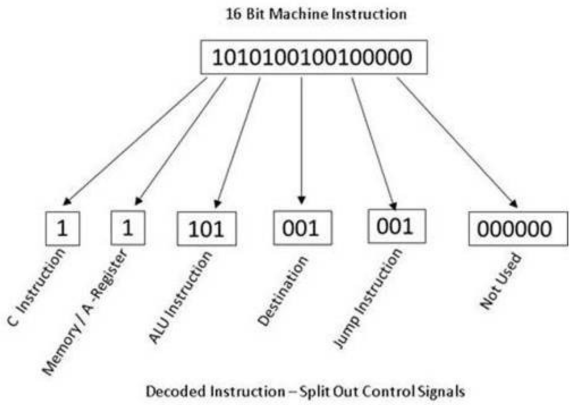


Figure 6.75 Decoded Instruction

In modern computer architectures, there are a variety of different instruction decoding techniques used including the use of Microcode and other techniques developed to make CPU's more efficient and improve performance. For our purposes, however, we will consider decoding as the process of retrieving an instruction from ROM and using this instruction to activate the control signals that facilitate the operation of the computer.

Memory Registers

Registers are memory devices designed to buffer the end of a bus. We know that data and control signals exist only as long as the signal is being sent. They are not persistent. Registers are essentially memory devices that can maintain the state of a bus until a new state is provided.

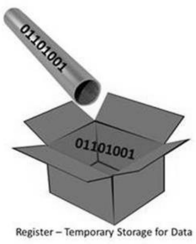


Figure 6.76 Memory Register

It is sometimes helpful to think of a register as a box at the end of a pipe. The data flows down the pipe and into the box where it says until it is emptied out.

Although the number, size, and use of registers varies from one computer architecture to the next, some registers are present in all computer systems, including the one that we created. These include:

The Program counter (PC) – The program counter is an incrementing counter that keeps track of the memory address of the instruction that is to be executed. The program counter is essentially a memory circuit very similar to other registers with the exception that it can increment its value with each cycle of the clock.

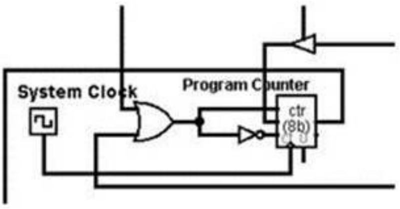


Figure 6.77 Program Counter

Memory address register (MAR) – The MAR holds the address of a memory block to be read from or written to. The MAR in our computer system is identified as the register that is located just to the left of the RAM component as we can see in Figure 6.7.

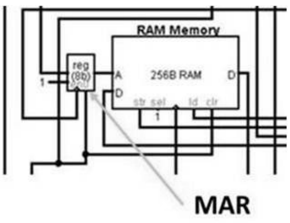


Figure 6.78 Memory Address Register

Memory data register (MDR) - register that holds data fetched from memory (and ready for the CPU to process). We see this register in Figure 6.8. The M/A bit in the instruction determines whether to use this register or the A Register in ALU computations.

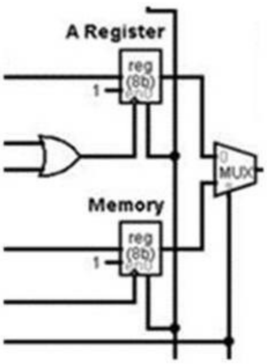


Figure 6.79 Memory Data Register

Instruction register (IR) - a temporary holding ground for the instruction that has just been fetched from memory. Our computer system design has not implemented the IR. An IR would be required in a design that requires more than 1 cycle to complete the processing of an instruction.

The D register - The D register stores the value that is used as one of the two operands that are used as input to the ALU. The D register is the stationary register as it is always one of the operands to the ALU while the other one can be either the Memory register or the A Register. Although we will not cover this in detail until chapter seven, be aware that the D register is also called the accumulator in the von Neumann architecture.

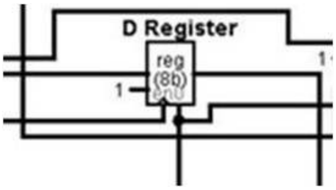


Figure 6.80 D-Register

The Jump address register is used to hold an address that is used in conjunction with the jump functionality. When one of the jump instructions are executed, they essentially activate the signals that send the value from the jump address register to the program counter resetting the position in ROM memory which controls which instruction is fetched to executed.

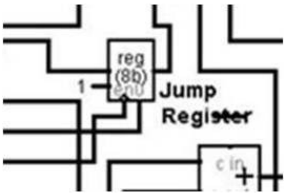


Figure 6.81 Jump Address Register

The A register is the register that can be used to load data into the computer system. The A instruction in our computer system loads a value into the A register. The A register can be used as an operand with the D Register as input into the ALU.

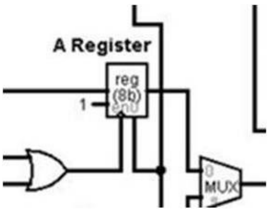


Figure 6.82 A-Register

The memory register is the register that can be used to load data values from RAM memory as an operand that can be used with the D Register. Essentially the Memory or A-Register bit in the machine instruction controls whether the A-Register or the Memory register is used in ALU operations.

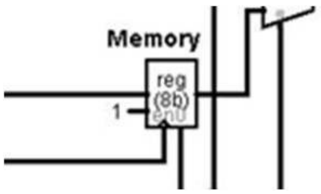


Figure 6.83 Memory Register

The Stack

The stack is an advanced concept that we have not implemented in our computer system, however it is important that we understand how the stack functions and is used because most modern CPU architectures do employ a stack.

The stack is a memory structure that is used to store information temporarily. A stack is a data structure that operates as last in, first out (LIFO).

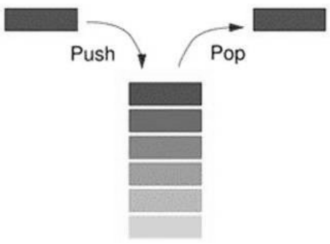


Figure 6.84 Stack Operation

Think of a stack of plates. You can place a plate on the stack and then take it back off. In the case of the stack within the CPU, the stack is used to push memory items on the stack and then pop them back off as required.

The way that the stack is used varies, but one use of the stack is in maintaining state. There are many conditions where some processing on the CPU needs to be suspended while other things are processed.

One example might be subroutines. Imagine you are running a program and in the program, you call a subroutine. Executing the subroutine suspends the current process. While the code in the subroutine is occupying the CPU, we need to have a way to keep track of what the program was doing, where it left off, to save the values in the registers and perhaps the position of the program counter. The stack provides an excellent solution to address this problem. As processes interrupt the CPU, current work is pushed on the stack while the CPU addresses the more pressing processing. When the subroutine or other urgent processing is completed, the state is popped off the stack, restoring the point where processing left off and processing continues where it left off.

The use of the stack is one of the features that makes modern computer systems capable of running multi-user, multi-tasking operating systems that are capable of servicing many programs and users.

Input and Output

The final topic that we will address in this chapter is the subject of I/O (input and output). Most of us are familiar with a typical computing model which has input and output devices such as secondary storage (disk drives, USB drives), output devices such as graphical displays, input devices such as the keyboard, mouse, camera, and devices that facilitate network connectivity such as LAN interfaces, Bluetooth devices, wireless interfaces, and serial based devices (USB).

It might come as a surprise, but to the computer system, all of these devices are memory. So far, we have implemented both ROM memory and RAM memory devices in our circuits. We have discussed the fact that in most general-purpose computer systems, ROM memory or the memory that we use to store program instructions is simply a region of memory that has been reserved for program code.

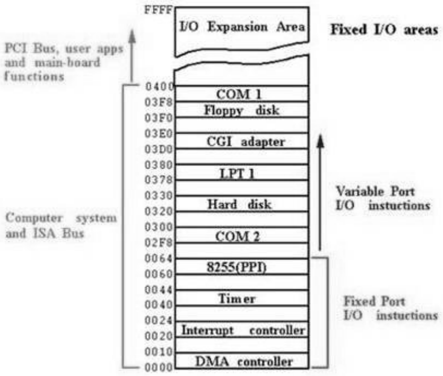


Figure 6.85 Memory Addressing Structure

What we quickly discover is that all memory in a computer system tends to be addressed using a common addressing scheme (memory starts at zero and extends to the limit of memory). Some of these addresses do not contain RAM memory but rather are addresses that provide an interface to the various I/O devices. The diagram above (Figure 6.14) provides an example of how different I/O devices are mapped into the memory addressing scheme.

In figure 6.14, when memory location 0320 (hexadecimal) is accessed we are accessing the hard disk. When we put data into location 0378, it is being sent to the printer.

The graphics system of your personal computer, phone, or tablet is another example, as all of the data is mapped as memory addresses in this system. Each pixel of the graphics display has both color and intensity information that is stored in memory) is mapped as memory addresses in the computer..

In our computer system circuit from Figure 6.1, we only have a single register that we use to display output. However, in most modern computer systems all of the I/O devices are mapped as memory addresses in the memory address scheme.

Machine Language, Instruction Decoding, and Execution

Our computer system, like any computer system, has a set of instructions that it can execute. We now know that these instructions are actually a set of bits in a binary word that are used to send control signals to the ALU to select the different computations that the ALU can perform to control the destination of the result of the ALU computation or to implement jump functionality. A particular instruction in our computer system, is made up of a number of elements including specifying whether to use the Memory Register or the A register in an operation, and the ALU computation to perform. We also specify the destination (A Register, D Register, ROM Address, RAM Address, and Memory at the current RAM address) to send the results of the operation, and the conditions under which a jump should be executed.

We will be using the ROM component within *Logisim* as the location to store our program instructions. *Logisim* requires that the contents of both the RAM and ROM components be loaded using Hexadecimal codes. In our design, we are using a 16-bit binary number to specify an instruction, and 16-bits in binary require four digits in hexadecimal. The inputs and outputs of these devices are in binary, but *Logisim* uses hexadecimal because it can represent each location with fewer digits as we can see in the following diagram (figure 7.1), which shows us the ROM module.

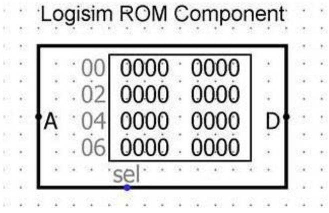


Figure 7.86 *Logisim* ROM Component

Machine Instruction Types

Each of the 4-digit hexadecimal numbers in our ROM component is either an A instruction, which is a number to be loaded into the A register or a C instruction, which is a machine instruction that the system can execute. You will notice that each instruction is 16-bits in length, but we are not using all of the bits. The A instruction for example, uses the first bit to indicate that it is an A instruction (has value of 0), the next 8-bits hold the value to be loaded into the A register and the rest of the bits are ignored and not used.

A	Address/Value	Not Used
0	00000000	0000000

We should realize by now that this means that our computer system has an 8-bit data architecture, so all of our data busses should have 8-bits in them. The A instruction is very important because it is the primary means to get data into the computer system. The only way that we can introduce a data value is by loading it using the A instruction, which will load the data value into the A register. We can then move this data value to wherever it is required.

The data value could be a number that is being used in a computation or it could be an address to either RAM or ROM that will allow us to reference a particular location in memory to either retrieve (in the case of RAM or ROM) or to update (in the case of RAM).

The C instruction, on the other hand, is a computation instruction.

C	M/A	Instruction	Dest	JMP	Not Used
1	0	000	000	000	00000

The C instruction uses the first bit to indicate that it is a C instruction (has value of 1). The next bit indicates whether the computation should use the memory register or the A register. The next 3-bits define which instruction the ALU should perform.

Following this, we have three bits to specify the destination to which the output of the ALU computation should be sent. Finally, we have 3-bits to define the conditions under which a jump is to be executed. The remaining 5-bits in the instruction are ignored and not used.

Instruction Decoding

Our computer system has two types of instructions: the A instruction and the C instruction. The A instruction is used to load data into the A register. The C instruction is used to specify the computations that our computer system will execute. In the example computer system that we reviewed in Chapter 6 (Figure 6.1), we can easily see that we have these two types of instructions when we examine the instruction decoding section of the circuit.

In the expanded view of this section of the computer system circuit (Figure 7.2), we clearly see the instruction carried on a data bus that is output from the ROM memory component (the data output is represented by D on the ROM component).

We then see a *Logisim fan out* component used to extract the first bit in the instruction and this bit is used as the select bit on a De-Multiplexor. What this bit is doing is selecting between the left hand portion of the circuit which decodes the A instruction and the right hand portion of the circuit that decodes the C instruction.

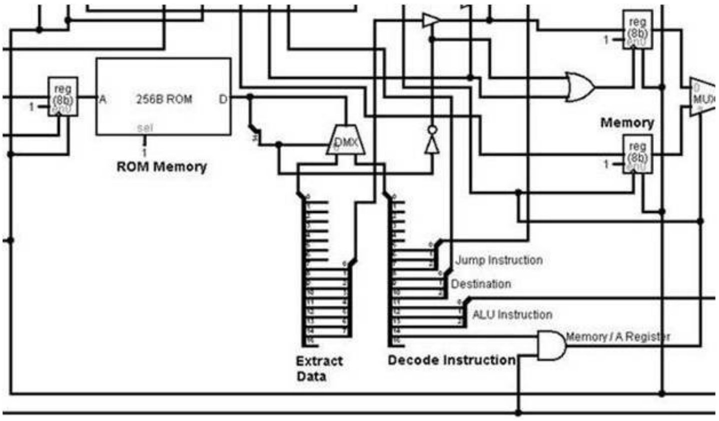


Figure 7.87 ROM and Instruction Decoding

In this example, our decoder is relatively simple. We are using the fan out to select the groups of bits that we need either populate a data bus, which is the case in the A instruction, or to populate the various signal busses for the ALU instruction, Destination, Jump Instruction or the bit, which selects to use either the Memory register or the A register.

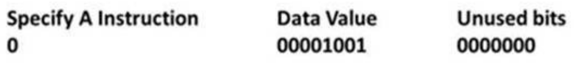
Constructing a Machine Instruction

We now turn our attention to how a machine instruction is actually constructed. We know that we have a variety of bits in an instruction that control different functions within our computer system. The process of creating a machine instruction is simply one of determining which functionality we want to execute for a particular instruction and then constructing a 16-bit binary word that contains the required bits.

For the A instruction this is relatively easy. The first bit will always be a “0.” Next, we add the value we want to load to the A Register as an 8-bit binary word. Finally, we simply add the rest of the digits as zeros. As an example, assume we want to load the value of decimal 9 to the A register. What would the A instruction look like? First, we need to convert decimal 9 into its binary equivalent.

You should be able to do this conversion manually, however, there is also a simple online converter that you could use that is available at http://www.binaryhexconverter.com/decimal-to-binary-converter

The binary equivalent of decimal 9, is 1001. We need an 8-bit binary number for our data so we would need to add some zeros to the beginning of this, which would result in the binary number, 00001001. Now we can construct the instruction as follows



Specify A Instruction	Data Value	Unused bits
0	00001001	0000000

The resulting instruction would then be:
0000010010000000

Of course, in order to load this instruction into the computer system's ROM component, we need to convert it into hexadecimal format. You should be able to do this conversion manually as well, however, there is a simple online converter that you could use available at http://www.binaryhexconverter.com/binary-to-hex-converter

The resulting instruction in hex is 0480, which could be entered into the ROM module of our computer system circuit and executed.

The A instruction is rather easy, so we will now tackle something a bit more difficult, the C instruction.

For the C instruction, the first bit will always be a "1." To this, we will add a number of different signal bits. The options include:

Use Memory Register or A Register bit

0	Use A Register
1	Use Memory

ALU Instruction

0 0 0	No operation (Does Nothing)
0 0 1	Add D+A (or D+M if the M/A bit is set)
0 1 0	Subtract D-A (or D-M if the M/A bit is set)
0 1 1	A AND D (or D AND M if the M/A bit is set)
1 0 0	A OR D (or D OR M if the M/A bit is set)
1 0 1	Pass through Register D
1 1 0	Pass through Register A

Destination

0 0 0	No operation (does nothing)
0 0 1	A Register
0 1 0	ROM Address (points to location in ROM memory)
0 1 1	D Register
1 0 0	Memory (memory location in RAM)
1 0 1	RAM Address (points to location in RAM Memory)
1 1 0	Jump Address Register (sends to register which contains the jump address that will be used in a subsequent operation)
1 1 1	Output Register (the register for output where we can send the output of a computation to examine)

Jump Instruction

0 0 0	no jump
0 0 1	Jump less than 0
0 1 0	Jump greater than 0
0 1 1	Jump equal to zero
1 0 0	Unconditional Jump

Notice that a value is *REQUIRED* in the C Instruction for each of these different sections. If we do not want to enable any of one of these different sections functionality, we can simply specify the bits “000” which in each case is a “No Operation” instruction.

It might be easier to understand if we worked through an example. Imagine that we have a value in the D register and we have loaded a value in the A register. Imagine that we wanted to add the contents of the A register to the contents of the D register and we wanted to send the results of this operation back to the D register. How would we specify this instruction?

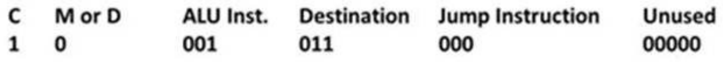
First, we know that the first bit will always be a “1” because it is a C instruction. Second we know that we want to use the A register as opposed to a value in the Memory register for the computation, so the “Memory Register or A Register” bit would need to be set to “0.”

Next, we get to the ALU instruction. We know that we want to execute an ADD operation between the A Register and the D Register so looking at the codes for the ALU instruction we see that the ADD instruction is specified by the bits “001.”

For the next set of bits, destination, we know that we want to send the results to the D register so we look at the table and discover that the bits for the D Register destination are “011”.

Finally, we get to the Jump Instruction. For this operation, we do NOT want to execute a jump so we want to specify a “No Operation.” Looking at the list of jump instructions, we see that the bits “000” will disable any jump operation.

Putting the entire instruction together looks like this:



C	M or D	ALU Inst.	Destination	Jump Instruction	Unused
1	0	001	011	000	00000

The resulting instruction would then be:
1000101100000000

Of course, in order to load this instruction into the computer systems ROM component, we need to convert it into hexadecimal format. The resulting instruction in hex is 8B00, which can be entered into the ROM module of our computer system circuit and executed.

We have now seen how to put together machine instructions in binary and convert them into hexadecimal format that can be loaded into the ROM component within a *Logisim* circuit.

A group of such instructions put together form a program. During the next chapter, we will learn how we can represent these Machine instructions using symbols to make programming easier and how these symbols can be converted directly into the machine code that we can load into the computer circuit using a special program called an assembler. For now, however, we need to explore different types of functionality in the computer to solve common programming problems. In particular, we will look at how we can load data, store values in memory, implement loops, and make decisions using conditional logic, and do all of these things with the simple machine instructions that our computer system can process.

Programming the Computer System

Now that we have learned how to define machine instructions for the UoPeople computer system, we can begin to develop programs for our computer. To understand how to develop programs for the computer, we need to have a working knowledge of how the computer operates.

The diagram in Figure 7.3, details the operation of the UoPeople computer system in a block diagram featuring the registers, memory, and the ALU of the computer, connected with grey paths that represent the data busses.

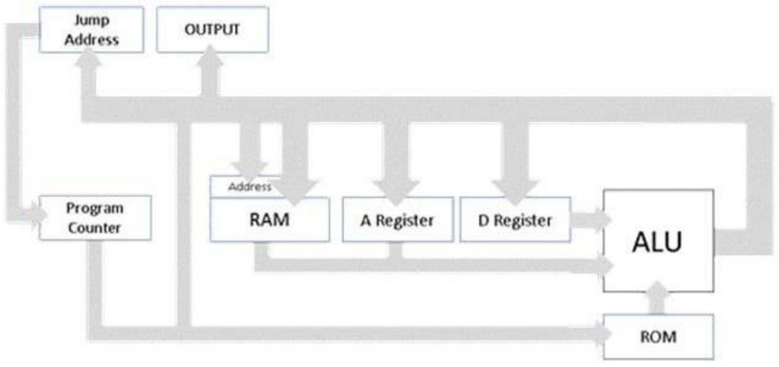


Figure 7.88 UoPeople Computer Data Flow

In the diagram, we can visualize the computer as a loop through which data flows. Data flows into the ALU from the D Register and either the A Register or the Memory register and flows out of the ALU back to the various destinations.

We can see why the ALU is the focal point, as everything flows through it. The ROM supplies instructions, the D Register and either A Register or Memory register supplies data and the ALU sends its results back out to the rest of the computer system via the bus.

This diagram helps us to realize that ALU has three different kinds of functions that it performs. First, the ALU ***computes data***. The ALU in our computer system can compute add, subtract, AND, and OR of two binary numbers.

Second, the ALU ***moves data*** within the system. One of the characteristics of the von Neumann architecture is the fact that the bus and movement of data and instructions is central to the design. The innovation of von Neumann was the use of memory for both instructions and data. In order to implement this innovation, von Neumann required a great deal of flexibility to move data and instructions between memory, registers, and the ALU. You will find as you become more proficient at programming with machine and assembly language, that a large percentage of what you, do as the programmer, do is specify the movement of data in the form of addresses, data values, and instructions between memory, registers, and the ALU.

Finally, the ALU ***controls operations*** of the computer system. Although the control system is principally responsible for control, the ability to move an address directly into ROM provides direct control from the ALU over the execution of programs within the computer system.

It is important to understand these three functions of the ALU because all of the instructions that you will implement within your programs will be used to implement one of these three tasks. You will find that the ability of the ALU to move data within the system is the task that will demand most of the ALU's time and will be the focus of most of your programming efforts.

Consider the following examples. Assume that within your program you want to store a value in memory. To accomplish this you would first need to load the address in memory where the value needs to be stored and MOVE it through the ALU and to the RAM address register. Second, you would need to load the value to be stored and then move this value through the ALU to Memory. Since we previously stored the RAM address register, the value will be stored into that location in memory.

Consider another example. Assume that we want to add together two numbers, which were both stored in memory. How would we accomplish this? Again, we would need to load the address in memory of the first number into the A Register and then move this address to the RAM address register. Next, we would move the contents of that memory location to the D Register. We move this to the D register for two reasons. First, the D Register is the register that is always used in ALU operations. Second, we need to store the value somewhere temporarily while we do the processing to get the second value. To get the second value we would simply repeat the process by loading into the A Register the address in RAM of the second number. This address is moved to the RAM address register. Finally, we would execute an instruction to add the contents of the Memory register, which of course is pointing to the second number to the D Register, which is still storing the first number. The result of this computation would then need to be moved to either a register or other memory location.

What is important to see in this example is that we had one instruction to add two numbers, but five instructions to load and move values to the proper registers so that we could actually perform the required computation.

The key to machine and assembly language programming that should be clear from these examples is our ability to plan out all of the steps necessary to load and move values to get them positioned in the ALU for the desired computation.

von Neumann's Accumulator

We have mentioned a couple of times the importance of the D Register in our computer system. The D Register is essentially the primary register for the ALU. When we are using the add operation within the ALU, we are adding something to the D Register. When we are subtracting, we are subtracting something from the D Register. To explain the importance of the D Register we need to go back to the diagram of the von Neumann architecture that we first saw at the beginning of Chapter 5.

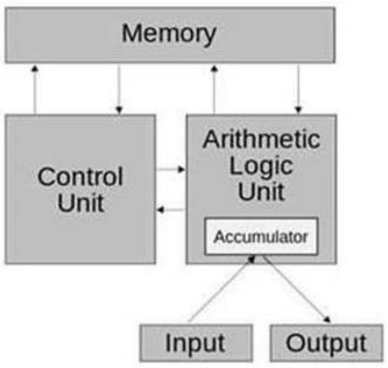


Figure 7.89 von Neumann Architecture

The diagram (figure 7.4) details the key components of the von Neumann architecture. We have addressed most of these components in detail. In chapter three, we learned about the design of the ALU. In chapter four, we studied memory and registers. In chapter five, we learned about the bus and the control unit and in chapter six, we learned how Input and Output devices are mapped as memory addresses. We have also learned how to use registers to get data into and out of the ALU. What we have not covered is the white box in Figure 7.4 called the accumulator.

The von Neumann architecture was designed around a register called the accumulator. The basic idea of the accumulator was that it was the register against which all ALU operations are performed. In our ALU design the D Register is in fact the accumulator and compute operations are performed against the accumulator. We add a number to the accumulator, we subtract a number from the accumulator, we AND or we OR a number to the accumulator. In every case, the operation is conducted against the accumulator.

By the way, hopefully you have recognized this, but when we see the input box in the von Nuemann architecture diagram it refers to the register that is used to operate AGAINST the accumulator. Consider the fact that we can either use the A Register or the Memory register. Further, we have already stated that all input and output devices in the computer system are in fact treated as memory devices as they are all mapped into the memory addressing scheme. This insight should help us to understand all forms of input and output that occur within the computer system.

Loading Data and Storing Values in Memory

We have already discussed the fact that in order to load new data into the computer system, we need to use the A instruction which will load the specified value into the A register. However, in order to make use of that value, we will need to do something with it.

We could execute an ALU instruction against the value in the A register, but suppose that all we wanted to do was to load a value into memory.

The solution to this problem would be to use an A instruction to load the value into the A register and then use a pass through instruction with a specified destination to move the value to one of the destination registers.

Loading data consists of two instructions. The first instruction is used to load the data into the A register and the second instruction is a C instruction which is used to move the value to the appropriate destination register. Assuming that we needed to store the value 21 into memory location 10, we would need to load 10 using an A instruction, move the value to the Memory address register, Load 21 into the A register and then move the value 21 to the memory destination. All of this will take four machine instructions as follows:

```
0000010100000000 : CNST #10
1011010100000000 : MOVA RAM
0000101010000000 : CNST #21
1011010000000000 : MOVA MEM
```

You might notice in this list of instructions both the machine language instructions are on the left, and something unfamiliar is on the right. On the right in this listing is the assembly language equivalent of the machine instructions. It should be clear that writing and reading the version on the right is easier. We can almost figure out what it means because it appears similar to English. For example, CNST is an abbreviation of the word constant and this specifies the instruction to load a value into the A register. CNST #10 generates an A instruction containing the value decimal 10.

The next instruction, MOVA RAM, also seems to make sense if we read this as MOVe register A to the RAM memory address. It should be apparent that using assembly language is much easier than writing machine instructions. With assembly language, we write our machine instructions using simple mnemonic symbols for machine instructions and then use a program called an assembler to translate these symbols into their machine instruction equivalents. We will learn more about assembly language later in this chapter and all of Chapter 8 is dedicated to assembly language.

Implementing Conditional Logic

Conditional statements in higher level programming languages include statements such as the "if-then-else" statement, the "switch-case" statement and of course the conditional statements that are parts of loops. Examples include while, do-while, and the test in the form statement.

If we carefully consider what these statements are doing, we realize that they are essentially conditional branches that will branch to different blocks of code based upon an evaluation of the expression that is a part of the statement.

We should take a moment to consider two terms here, *branch* and *code block*. A branch is an instruction that transfers control to another place within the program code.

In the past, we had a language called BASIC, and one of the statements in BASIC was GOTO. These days GOTO is considered a very bad statement because it promoted the development of messy code that was difficult to understand, follow, and maintain. However, we are going to refer to it because it helps us make sense of how we need to implement branches using machine instructions. Consider the following short program written in the BASIC language.

```
10 PRINT "MODFUNC"
20 INPUT A
30 INPUT B
40 IF A<=B THEN GOTO 70
50 A=A-B
60 GOTO 40
70 PRINT A
80 END
```

One of the first things you might notice, is the use of line numbers. Each program statement exists on a line number. The program is executed by moving from one line number to the next.

This processing sequence can be altered with statements such as the IF statement that we see in line 40. The IF statement executes a branch, which in this case is implemented with the GOTO statement. The IF statement indicates that if the value in variable A is less than or equal to the value in the variable B, then branch or transfer control to line 70 in the program. If it makes itIt may be easier to understand, thinking of this as "going to" line 70.

The way that we implement a branch using machine instructions is very similar to this. All that we need to do is to load the line number that we need to GOTO into the A register, move this value to the jump address register and then execute a jump.

Incidentally, at line 60 in the BASIC program we see another goto statement. This one goes to line 40 where we test if A is still less than or equal to B. What is interesting is that we are seeing here an implementation of a while statement. Again, this simple BASIC program gives us insight into how we need to implement common programming problems with our machine instructions.

We now understand how to implement a branch, but how do we evaluate a conditional expression? The answer is simple. If we think about the expression if A<=B, we realize that what we are looking at is an inequality. We can solve this inequality using a bit of math by subtracting one term from the other and then examining the result.

For example, let us assume that B has a value of 5 and A has a value of 4.

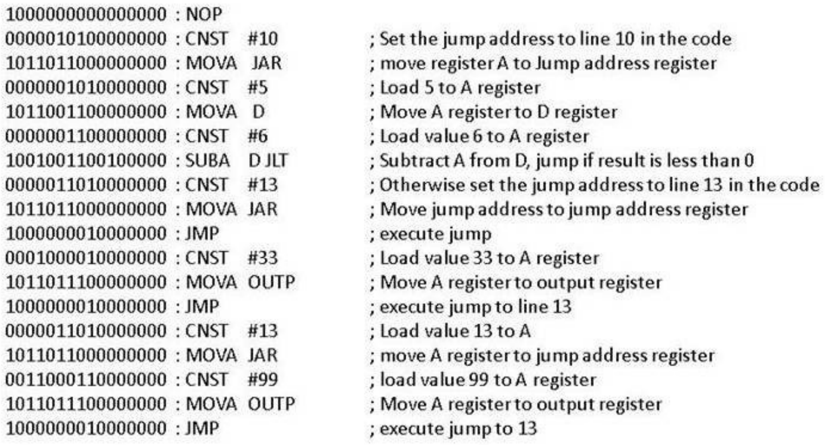
If we subtract 4 from 5, we get 5-4 = 1. The result is a positive number.

If B has a value of 5, and A has a value of 5, then subtracting 5-5 = 0. The result is zero.

Finally if B has a value of 5 and A has a value of 6, then subtracting 5-6 = -1. The result is a negative number.

You may see now where we are going with this. To evaluate a conditional expression using machine instructions, we need to evaluate it using a mathematical operation. In the preceding example the expression A<=B would be true UNLESS the negative flag from the ALU were to be set to a logical 1. As such, all that we need to do to evaluate the expression and execute the appropriate branch, is to subtract one operand from the other, set the jump address, and then execute a jump if the result of the subtract operation has a value less than 0.

The following code would implement this branch:



```
1000000000000000 : NOP
0000010100000000 : CNST   #10        ; Set the jump address to line 10 in the code
1011011000000000 : MOVA   JAR        ; move register A to Jump address register
0000001010000000 : CNST   #5         ; Load 5 to A register
1011001100000000 : MOVA   D          ; Move A register to D register
0000001100000000 : CNST   #6         ; Load value 6 to A register
1001001100100000 : SUBA   D JLT      ; Subtract A from D, jump if result is less than 0
0000011010000000 : CNST   #13        ; Otherwise set the jump address to line 13 in the code
1011011000000000 : MOVA  JAR         ; Move jump address to jump address register
1000000010000000 : JMP               ; execute jump
0001000010000000 : CNST   #33        ; Load value 33 to A register
1011011100000000 : MOVA  OUTP        ; Move A register to output register
1000000010000000 : JMP               ; execute jump to line 13
0000011010000000 : CNST   #13        ; Load value 13 to A
1011011000000000 : MOVA  JAR         ; move A register to jump address register
0011000110000000 : CNST   #99        ; load value 99 to A register
1011011100000000 : MOVA  OUTP        ; Move A register to output register
1000000010000000 : JMP               ; execute jump to 13
```

In this program, we are evaluating if 6 < 5. If not, it branches to line 10 in the program. Otherwise, it branches to line 13. Notice that we are simply outputting the value 33 to the output register if we branch to line 10 and we are outputting the value of 99 to the output register if we branch to line 13. This is just a useful way in our computer system to determine that our "if" expression worked correctly.

You could take each of these binary machine instructions, convert them into their hexadecimal form, place them all in a text file, and then load and execute this program using the computer circuit that is defined in Figure 6.1.

Loops with For and While Characteristics

A loop, if you think about it, has three parts. The first part is a conditional expression that tests whether to terminate looping. The second is a branching mechanism that is executed when the end of the loop is reached it returns to the beginning of the loop to execute it again. Finally, a code block forms the basis of the loop as well as the instructions that are executed within it.

Consider this simple example in pseudo code:

```
i=0
while (i <= 5) {
            i++
            print i
}
```

This simple loop initializes a counter variable, enters the loop, tests whether the value of the variable is less than 5, then executes the code block. In this case, the code block increments the counter variable and prints the value of the variable to output.

Each of these instructions can be implemented using our machine instructions. We have already learned how to execute a branch and we have learned to evaluate an expression. The only thing that we need to add is the ability to increment a value by adding one to it. The following code shows an implementation of the loop using our machine instructions.

```
1000000000000000 : NOP
0000001100000000 : CNST   #6
1011010100000000 : MOVA  RAM
0000000000000000 : CNST  #0
1011001100000000 : MOVA   D
1010110000000000 : MOVD  MEM
1111011100000000 : MOVM  OUTP
0000101010000000 : CNST  :LOOP2
1011011000000000 : MOVA  JAR
0000001100000000 : CNST  #6
1011010100000000 : MOVA  RAM
0000000010000000 : CNST   #1
1011001100000000 : MOVA  D
1100110000000000 : ADDM  MEM
0000001010000000 : CNST  #5
1011001100000000 : MOVA   D
1101000001100000 : SUBM   JEQU
1111011100000000 : MOVM OUTP
0000001110000000 : CNST  :LOOP1
1011011000000000 : MOVA  JAR
1000000010000000 : JMP
0000101010000000 : CNST  :LOOP2
1011011000000000 : MOVA  JAR
1000000010000000 : JMP
```

The following instructions in this example are implementing the increment of a variable value.

```
CNST   #6
MOVA  RAM
CNST   #1
MOVA  D
ADDM  MEM
```

Simplifying the Process with Assembly Language

Although we have reserved Chapter 8 to dig deeper into the concepts of Assembler language, it makes sense to provide an initial introduction to assembly language programming at this point, as well as to the assembler language that we have defined for our computer system.

An assembler is a program designed to translate symbols representing machine language instructions into the machine instruction format, and output them in the appropriate format to execute on a computer system.

In our case, the assembly language symbols are converted first into their binary equivalents and then into the hexadecimal format that is required for input into the ROM component within our *Logisim* circuit.

You can access the assembler program for the University of the People architecture directly at the following URL and, of course, the assembler program is available from within the Moodle course page for those of you who are registered for the course.
http://uopeopleweb.com/js/uopeopleassembler.html

With any language, we need to know the syntax and grammar of that language. For our assembler language, the syntax or structure of statements follows the design of our machine instruction where we will have tokens to represent ALU instructions, destinations, and jump instructions.

An example of a statement that has all three of these elements would be:
ADDA MEM JEQU

In this statement, we are instructing the computer to take the value in the A register, add the value to the value in D register (hence add a) and send the results of the computation to RAM memory. If the result of the computation is equal to zero, then execute a jump to the location that has already been loaded into the jump address register.

It is important to understand that ALL operations that involve two operands will feature an operation between Register D and either the Memory Register or the A Register.   In this case, we see that the instruction ADDA has an A on the end of the ADD instruction, which means that the instruction will add the value in Register A to value in Register D.   If the instruction were to be slightly altered to ADDM it would mean that the value at the location currently loaded into the Memory register would be added to the value in Register D.

The following is the full list of Assembly language instructions supported by the assembler, which the assembler can compile into machine instructions.

Assembler	Description
;xxxx	Anything in the assembler that begins with a ; is a comment and should be ignored
:label	Anything beginning with a : is a label for a location that will be evaluated and stored in the symbol table during the first pass of the compiler. The actual address in ROM will be replaced with the symbol during second pass processing. NOTE that 'label' should be replaced with a label of your choosing

	such as loop1, start, end or anything that makes sense in your program.
LOADA	Move data from A to current location in memory
LOADD	Move data from D to current location in memory
CNST :label	Lookup the label in the symbol table and replace the label with the actual address.
CNST #num	Load the number following the # into the A register
ADDM	Add contents of D register to Memory
ADDA	Add contents of D register to contents of A register
SUBM	Subtract contents of Memory from D register
SUBA	Subtract contents of A register from D register
ANDM	Bitwise AND D register with contents of Memory
ANDA	Bitwise AND D register with contents of A register
ORM	Bitwise OR D register with contents of Memory
ORA	Bitwise OR D register with contents of A register
MOVD	Move contents of D register to destination
MOVA	Move contents of A register to destination
MOVM	Move contents of the location in Memory (that is currently being pointed to by the memory address register) to destination
A	Destination Register A
D	Destination Register D
ROM	Destination ROM Address register
RAM	Destination RAM Address register (sets the location pointed to within memory)
MEM	Destination MEM (updates the location pointed to by the RAM Address register)
JAR	Destination Jump Address register
OUTP	Destination Output register … convenient way to output a number so that you can see the result of a computation
JGT	Jump if output of operation is greater than 0
JLT	Jump if output of operation is less than 0
JEQU	Jump if output of operation is equal to 0
JMP	Jump unconditionally

As you begin to write your own assembler programs for the UoPeople computer system, you should keep the following rules in mind. First, it is not required to have a jump instruction. The jump instruction can always be left off the command if it is not desired. In addition, jump instructions are *ONLY* processed when the ALU operation is either an ADD or a Subtract (ADDM, ADDA, SUBM, SUBA). After all, the jump functionality is based upon the output of the N,P,Z flags which are set based upon the output of an add or subtract operation.

Another point to keep in mind is that the addition of two positive binary numbers will never be negative. As such the JLT jump instruction will only be effective when used with a subtract operation.

The Assembler is not very smart and has limited (or no) error-checking features, so it will attempt to assemble what you give it. If your code is not correct, it simply will not execute the way that you might expect.

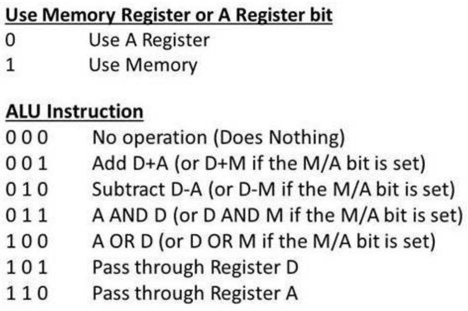
The JMP instruction may be executed on its own, because it does not rely upon the results of an ALU computation to determine when to execute the jump. The JMP instruction is called an unconditional jump because it will *ALWAYS* attempt to jump when executed.

Finally, although valid, an instruction that does not contain a destination will execute, but the results of the computation will not go anywhere. You can use this to execute jumps particularly if you do not want to store the results of the calculation used to initiate the jump.

Chapter 7 Exercise

For the Chapter 7 exercise, you will be provided with a series (four pseudo-code segments that represent typical code and data structures that one might implement with a language such as Java or Python.

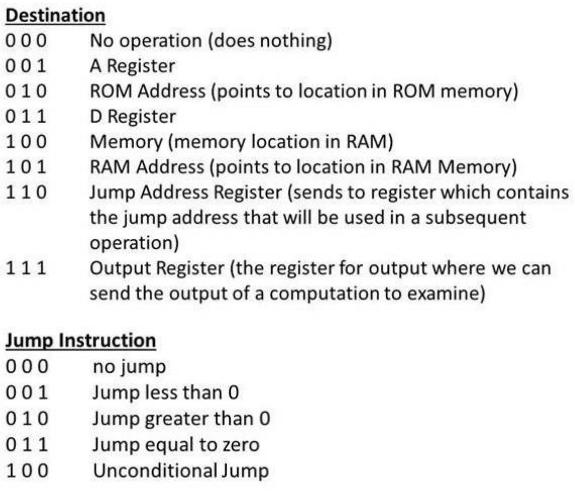
For your assignment, devise a strategy to implement these code and data structures using only the machine language for the computer system that we have created. You will be able to use the following machine language instructions to complete this assignment.



Use Memory Register or A Register bit	
0	Use A Register
1	Use Memory

ALU Instruction	
0 0 0	No operation (Does Nothing)
0 0 1	Add D+A (or D+M if the M/A bit is set)
0 1 0	Subtract D-A (or D-M if the M/A bit is set)
0 1 1	A AND D (or D AND M if the M/A bit is set)
1 0 0	A OR D (or D OR M if the M/A bit is set)
1 0 1	Pass through Register D
1 1 0	Pass through Register A

In addition, you can take advantage of the jump and destination functionality that is represented in the following two tables:



Destination	
0 0 0	No operation (does nothing)
0 0 1	A Register
0 1 0	ROM Address (points to location in ROM memory)
0 1 1	D Register
1 0 0	Memory (memory location in RAM)
1 0 1	RAM Address (points to location in RAM Memory)
1 1 0	Jump Address Register (sends to register which contains the jump address that will be used in a subsequent operation)
1 1 1	Output Register (the register for output where we can send the output of a computation to examine)

Jump Instruction	
0 0 0	no jump
0 0 1	Jump less than 0
0 1 0	Jump greater than 0
0 1 1	Jump equal to zero
1 0 0	Unconditional Jump

A couple of tips that might be useful as you develop your program:
Comparisons: Your program will need to compare two numbers to determine if one is larger than the other is. In our computer system, we do not have an if expression that allows us to test the equality or inequality of two variables. This capability can be

implemented by subtracting the second number from the first number and then executing a jump based upon the result. For example, if A and B are our two numbers, then the rules for A-B are as follows:

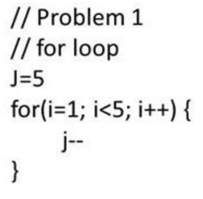
If A-B = 0, then A = B.
If A-B > 0, then A > B.
If A-B < 0, then A < B.

If the result of the operation is 0, then we can execute the 011 "jump equal to zero" instruction which means jump if the output of the operation is equal to zero (see jump instruction table above).

In the algorithm, we need to be able to test if a variable is less than, greater than, or equal to either another variable or a constant value. Each of these tests can be accomplished by subtracting the two numbers and then using the appropriate jump instructions such as:

0 0 1 – jump less than 0
0 1 0 – jump greater than 0
0 1 1 – jump equal to 0

The following four problems are the pseudo-code routines that you must implement using the University of the People computer system.



```
// Problem 1
// for loop
J=5
for(i=1; i<5; i++) {
      j--
}
```

```
// Problem 2
// if - then  - else
i=4
if (i < 5) then
       j = 3
else
       j = 2
```

```
// Problem 3
//while loop
i = 0
while(i==0) {
       j++
       if j = 5 then
              i = j
}
```

```
// Problem 4
// load and  traverse an array
A[0] =5
A[1]=4
A[2]=3
A[3]=2
A[4]=1
A[5]=0

for (i=0; i<=5; i++) {
      if A[i] == 0 then
             A[i] = 5;
}
```

(This page intentionally left blank)

Assembly Language

In Chapter 7, we had a short introduction to assembly language programming. Assembly language can be a difficult subject to grasp initially, so it makes sense that we introduce this same content again.

An assembler is essentially a program designed to translate symbols that represent machine language instructions into the machine instruction format and output them in the appropriate format to execute on a computer system.

In our case, the assembly language symbols are converted first into their binary equivalents and then into the hexadecimal format that is required for input into the ROM component within our *Logisim* circuit.

You can access the assembler program directly at the following URL and it is available from within the Moodle course page.
http://uopeopleweb.com/js/uopeopleassembler.html

With any language, we need to know the syntax and grammar of that language. For our assembler language, the syntax or structure of statements follows the design of our machine instruction where we will have tokens to represent ALU instructions, destinations, and jump instructions.

An example of a statement that has all three of these elements would be:
ADDA MEM JEQU

In this statement, we are instructing the computer to take the value in register D, add that value to the value in register A and send the results of the computation to RAM memory. If the result of the computation is equal to zero then execute a jump to the location that is currently in the jump register.

It is important to understand that ALL operations that involve two operands will feature an operation between Register D and either the Memory Register or Register A. In this case, we see that the instruction ADDA has an A on the end of the ADD instruction, which means that the instruction will add the value in Register D with the value in Register A. If the instruction were to be slightly altered to ADDM it would mean that the value in Register D would be added to the value at the location currently pointed to by the Memory address register and loaded into the Memory register.

The following is the full list of assembly language instructions supported by the assembler, which the assembler can compile into machine instructions.

Assembler	Description
;xxxx	Anything in the assembler that begins with a ; is a comment and should be ignored
:label	Anything beginning with a : is a label for a location that will be evaluated and stored in the symbol table during the first pass of the compiler. The actual address in ROM will be replaced with the symbol during second pass processing. NOTE that 'label' should be replaced with a label of your choosing such as loop1, start, end or anything that makes sense in your program.
LOADA	Move data from A to current location in memory
LOADD	Move data from D to current location in memory
CNST :label	Lookup the label in the symbol table and replace the label with the actual address.
CNST #num	Load the number following the # into the A register
ADDM	Add contents of D register to Memory
ADDA	Add contents of D register to contents of A register
SUBM	Subtract contents of Memory from D register
SUBA	Subtract contents of A register from D register
ANDM	Bitwise AND D register with contents of Memory
ANDA	Bitwise AND D register with contents of A register

ORM	Bitwise OR D register with contents of Memory
ORA	Bitwise OR D register with contents of A register
MOVD	Move contents of D register to destination
MOVA	Move contents of A register to destination
MOVM	Move contents of the location in Memory (that is currently being pointed to by the memory address register) to destination
A	Destination Register A
D	Destination Register D
ROM	Destination ROM Address register
RAM	Destination RAM Address register (sets the location pointed to within memory)
MEM	Destination MEM (updates the location pointed to by the RAM Address register)
JAR	Destination Jump Address register
OUTP	Destination Output register … convenient way to output a number so that you can see the result of a computation
JGT	Jump if output of operation is greater than 0
JLT	Jump if output of operation is less than 0
JEQU	Jump if output of operation is equal to 0
JMP	Jump unconditionally

As you continue to write your own assembler programs for the UoPeople computer system, you should keep the following rules in mind. First, it is not required to have a jump instruction. The jump instruction can always be left off the command if it is not required. Further jump instructions are ONLY processed when the ALU operation is either an ADD or a Subtract. After all, the jump functionality is based upon the output of the N,P,Z flags which are set based upon the output of an add or subtract operation.

Another point to keep in mind is that the addition of two positive binary numbers will never be negative. As such the JLT jump instruction will only be effective when used with a subtract operation.

The Assembler is not very smart and has limited (or no) error-checking features, so it will attempt to assemble what you give it.

If your code is not correct, it simply will not execute the way that you might expect.

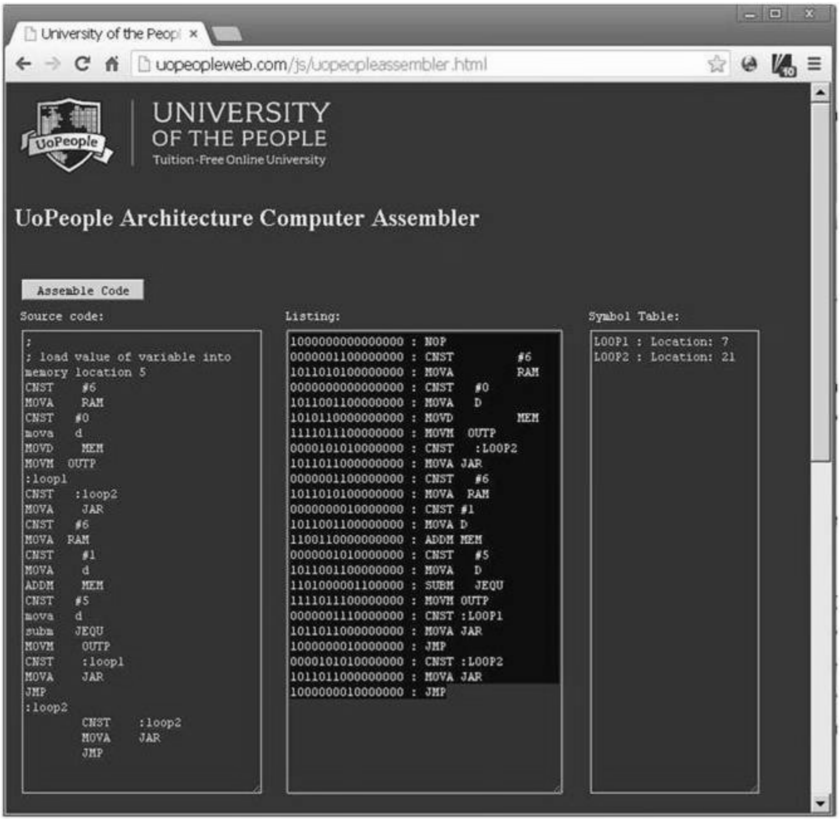
The JMP instruction may be executed on its own, because it does not rely upon the results of an ALU computation to determine when to execute the jump. The JMP instruction is called an unconditional jump because it will *ALWAYS* attempt to jump when executed.

Finally, although valid, an instruction that does not contain a destination will execute, but the results of the computation will not go anywhere. You can use this to execute jumps particularly if you do not want to store the results of the calculation that is used to initiate the jump.

Operating and Understanding the Assembler

The assembler is a JavaScript program that has been designed to evaluate the assembly language code for our UoPeople computer system and generate output in the form of the hexadecimal codes that are required as input into the ROM memory module that we use in *Logisim* to store the programs that our computer system will execute.

The following figure (Figure 8.2) is an example of the top portion of the assembler page:


University of the Peopl
uopeopleweb.com/js/uopeopleassembler.html
UoPeople
UNIVERSITY
OF THE PEOPLE
Tuition-Free Online University
UoPeople Architecture Computer Assembler
Assemble Code

Source code:

```
;
; load value of variable into
memory location 5
CNST    #6
MOVA    RAM
CNST   #0
mova   d
MOVD    MEM
MOVM  OUTP
:loop1
CNST   :loop2
MOVA    JAR
CNST   #6
MOVA  RAM
CNST    #1
MOVA    d
ADDM    MEM
CNST   #5
mova   d
subm   JEQU
MOVM    OUTP
CNST    :loop1
MOVA    JAR
JMP
:loop2
        CNST    :loop2
        MOVA    JAR
        JMP
```

Listing:

```
1000000000000000 : NOP
0000001100000000 : CNST         #6
1011010100000000 : MOVA         RAM
0000000000000000 : CNST   #0
1011001100000000 : MOVA   D
1010110000000000 : MOVD         MEM
1111011100000000 : MOVM  OUTP
0000101010000000 : CNST   :LOOP2
1011011000000000 : MOVA JAR
0000001100000000 : CNST   #6
1011010100000000 : MOVA  RAM
0000000010000000 : CNST #1
1011001100000000 : MOVA D
1100110000000000 : ADDM MEM
0000001010000000 : CNST   #5
1011001100000000 : MOVA   D
1101000001100000 : SUBM   JEQU
1111011100000000 : MOVM OUTP
0000001110000000 : CNST :LOOP1
1011011000000000 : MOVA JAR
1000000010000000 : JMP
0000101010000000 : CNST :LOOP2
1011011000000000 : MOVA JAR
1000000010000000 : JMP
```

Symbol Table:

```
LOOP1 : Location: 7
LOOP2 : Location: 21
```

Figure 8.90 Assembler Screen Part 1

In this diagram, we see that the assembler has three text panes. Each has specific information. The first pane, which has the title “Source Code,” is where you enter your assembler source code. You can either type your assembly language instructions directly into this textbox, or you can write your assembler code with your favorite editor and then cut and paste it into the source-code text-box.

The second pane that we see is labeled “Listing.” The listing pane is populated when you have entered all of your source code and then clicked on the “Assemble code” button.

What we see in the listing textbox are the assembly language instructions on the right hand side and the binary machine language equivalent of the instruction on the left hand side. You will probably notice that the listing does not exactly match what you entered into the source code text box. The reason for this is that during processing, the assembler will strip out all of the comments and will process all of the labels. When you put a label into your code, such as the :loop1 and :loop2 as in this example, the assembler will replace these labels with the actual location in the program that the label refers to.

We can tell which labels were processed and where the labels actually appear in the machine language program by looking at the third pane, which is called “Symbol table.”

You will notice that the symbols’ that we defined, which in this case are the labels that we can use to determine position in the source code, are listed in the symbol table, and to the left there is a location with a number. The number indicates the line in the machine language code where the label appeared.

These labels make it easier to program because we can simply refer to the label when writing our assembler language program instead of trying to determine at what line number of the code the label appears. Many assemblers also feature the ability to define and use variables in the assembler code. This feature is not currently supported in the UoPeople assembler.

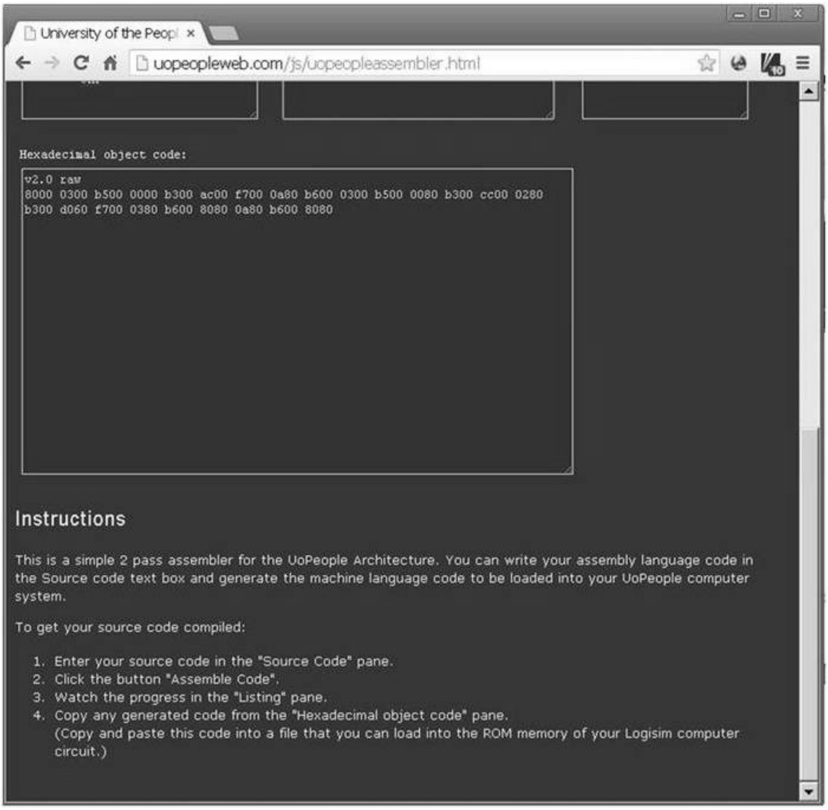
Three panes of information are populated by the assembler when the “Assemble Code” button is clicked. We have discussed the listing pane and the symbol table. The third pane that is populated is the “Hexadecimal object code” pane.

An example of this pane is captured in Figure 8.3. We see in Figure 8.3 that the Hexadecimal object code pane contains the object code that is required for the UoPeople computer system.

The ROM component within the *Logisim* tool is designed to load a program that is in this hexadecimal format. The ROM memory contains 16-bit instructions. Our computer system uses 8-bit addressing meaning that the number that we use to point to a location in either ROM memory or RAM memory is specified by an 8-bit number, which means that we can have a maximum of 255 locations in memory. Therefore, we could have a program with a maximum of 255 instructions. Each of these instructions is represented by a four-digit hexadecimal number.

To use the code generated in the "Hexadecimal object code" pane, you should select the contents of the pane, copy this content, and then paste it into a text file using your favorite text editor. For Windows users, Notepad would be a good choice. You should not use a word processing program such as Microsoft Word as these programs will typically add additional formatting information into the file. You should ensure that when you save your hexadecimal codes that you save it as an unformatted text file.

The following is an example of the bottom portion of the assembler page:



Hexadecimal object code:

```
v2.0 raw
8000 0300 b500 0000 b300 ac00 f700 0a80 b600 0300 b500 0080 b300 cc00 0280
b300 d060 f700 0380 b600 8080 0a80 b600 8080
```

Instructions

This is a simple 2 pass assembler for the UoPeople Architecture. You can write your assembly language code in the Source code text box and generate the machine language code to be loaded into your UoPeople computer system.

To get your source code compiled:

1. Enter your source code in the "Source Code" pane.
2. Click the button "Assemble Code".
3. Watch the progress in the "Listing" pane.
4. Copy any generated code from the "Hexadecimal object code" pane.
   (Copy and paste this code into a file that you can load into the ROM memory of your Logisim computer circuit.)

Figure 8.91 Assembler Screen Part 2

Assembly language mnemonics

According to Wikipedia (n.d.), a mnemonic is a learning device that helps with information retention, tending to be simple symbols that help us to remember things that are more complex.

> *Mnemonics aim to translate information into a form that the human brain can retain better than its original form. Even the process of merely learning this conversion might already aid in the transfer of information to*

long-term memory. Commonly encountered mnemonics are often used for lists and in auditory form, such as short poems, acronyms, or memorable phrases, but mnemonics can also be used for other types of information and in visual or kinesthetic forms. Their use is based on the observation that the human mind more easily remembers spatial, personal, surprising, physical, sexual, humorous, or otherwise 'relatable' [sic] information, rather than more abstract or impersonal forms of information (Wikipedia, n.d.).

The machine instructions comprised of 1 and 0 certainly qualify as an impersonal form of information. The objective of assembly language is to define simple memorable symbols to represent the more complex sequences of 1's and 0's that make up machine language.

When we look at the assembly language statements for the UoPeople computer system we can get an idea of what operation the computer will perform when the statement is executed.

For example:

ADDM
This instruction adds the current value in the Memory register to the contents of the D Register. The ADD gives us the clue that it will perform and ADD operation and the M tells us that it will use the Memory register to add to the D register.

MOVD
This instruction tells us that we will be moving something from the D register to whatever destination we specify.

MEM
This instruction is obviously defining a destination to send the results of a computation to. In this case the destination will be memory.

JGT
This instruction specifies a jump condition. It instructs the system to jump IF the operation that it is associated with resulted in a computation whose output was *GREATER THAN* zero.

JAR
This instruction specifies the Jump Address Register as a destination.

JMP
This instruction instructs the system to execute a jump.

As a programmer sitting down to write an assembler program, you have a good chance that you will be able to remember many of these instructions. It is also a good bet that you would not be able to remember the complex 16-bit binary number that these instructions represent.

The simplification makes programming much easier, faster, and less prone to errors. It is relatively easy to scan the assembler code and look for errors or to be able to determine the functionality of the assembler program. The same could not be said of the machine instructions, which are much more difficult to understand or debug.

One Pass and Two Pass Assemblers

The assembler that we are using in this chapter's assignments to *assemble* or *compile* our Assembly Language programs into the machine instructions that we can run on our computer system is a simple JavaScript program. This program reads the source code, interprets the assembly language instructions, and converts them into machine instructions.

Our assembler is very simple. It does not do a lot of error checking and it does not incorporate any functionality to deal with variables. It does however, provide some simple two pass functionality.

A one-pass assembler or compiler will read source code and convert the instructions that it identifies into the object code that can be executed by the computer system. For example, when the assembler program encounters the following command:

ADDA D JLT

It converts this into the machine instruction:

1000101100100000

We can refer this as the second pass in a two-pass assembler process. The first pass is designed to identify and replace all symbols.

In our assembly language, we can specify a label that indicates a position in the code that we want to jump to in the case of a loop or a branch. Instead of having to figure out what line number in the machine code to jump to for a branch or loop, we leave this up to the assembler to determine, which it does by the location of the label.

As the assembler reads the code in the first pass, it identifies any of these labels, determines the line number in the code where they appear and puts this information into what we call a symbol table. During the second pass, when we find one of these labels in a command, the label is replaced with the correct line number.

In our assembler, we are using this very simple approach to managing symbols to keep track of labels within the code.

However, in most modern assemblers we can do the same thing for variables.

As we write our assembler programs, we need to keep track of where we store values in memory. From the perspective of the machine language, there is only an address in memory, and the address is simply an offset from the beginning of memory.

Most modern two pass assemblers will allow us to define a variable in our assembler programs using a name, a symbol of some form that represents a memory location. As the assembler goes through the first pass it will identify all of those symbols and assign a memory location for them.

This is the same concept that we use to keep track of locations within the code (the location in the code is just an address or offset into ROM memory). In addition to replacing the symbol (or in this case, the variable name) with a memory location, the assembler in many cases will generate the code necessary to access a particular variable.

Consider the case where we want to store a number in memory and as part of the process, we want to retrieve the number from memory, increment the number by adding one to it, and then store it back into memory.

We know that retrieving a value from memory can have several steps. For example we would need to define a constant and load it into the A register that specifies the address of the item in memory.

We would then need to move the value from the A register to the memory address register. Finally, we could retrieve the value from memory.

What the assembler will often do in two pass assemblers is provide us with 1 simple assembler instruction which is then converted into each of these steps. So perhaps we would have an assembler instruction such as:

```
GETA  @var1
```

This one instruction would then be converted into a set of machine instructions such as:

```
0111111010000000 : CNST   #253
1011010100000000 : MOVA  RAM
1111001100000000 : MOVM  D
```

In this example, we see that the assembler had identified a free memory location at location 253 in memory and replaced our variable name with this location. Keep in mind that every time we use the symbol @var1 this set of instructions will be added into our machine language program.

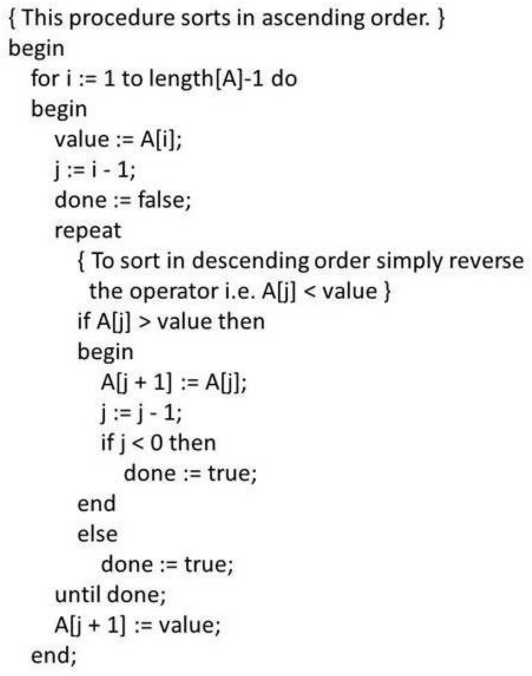
We also see that the assembler has done some of the work for us because it has automatically generated the machine instruction to load the memory address register, retrieve the value from memory, and store it in the D register to be used.

The use of two pass assemblers further simplify assembly language programming making it easier for the programmer and reducing the errors that might occur when attempting to manually keep track of specific memory addresses.

Chapter 8 Exercise

For the final exercise, you are provided with the code for an insertion sort algorithm written in pseudo-code (see code below).

Your task is to implement this insertion sort using the assembly language of our UoPeople computer system.



```
{ This procedure sorts in ascending order. }
begin
   for i := 1 to length[A]-1 do
   begin
      value := A[i];
      j := i - 1;
      done := false;
      repeat
         { To sort in descending order simply reverse
           the operator i.e. A[j] < value }
         if A[j] > value then
         begin
            A[j + 1] := A[j];
            j := j - 1;
            if j < 0 then
               done := true;
         end
         else
            done := true;
      until done;
      A[j + 1] := value;
   end;
```

The insertion sort defines and uses a number of variables including:

i	– a counter variable for the for loop
value	– a temporary variable used to hold a value to compare
A	- an array which holds the values to be sorted
j	- an index variable calculated from i
done	- a status variable indicating when the sort is complete

Use the following table of values to populate your array A. With assembler, this can be accomplished through the use of the CNST statement to load the value into register A and then the value can be moved into a location in memory.

12, 3, 52, 7, 1, 9, 16, 11, 5, 2

The array will have 10 elements meaning that you will need to load each value into a consecutive location in RAM memory.

The remaining variables can each be a single location in memory of your choosing.

The insertion sort algorithm features two loops. In the outer loop we see an example of a for loop as the loop counts from one to the length of the array. The inner loop is more like a while loop that will continue until the condition of the while loop is met. In this case, the condition is the done flag. When done has a value of 1, then exit the while loop. You can identify the while loop in the pseudo-code because it uses the keyword "repeat" to begin the loop and "until done" to end the loop.

You should be able to reuse much of the code that you developed during the Chapter 7 assignment because we implemented array traversal, the for loop, the while loop, and conditional expression (if-then-else).

Bibliography

Wikipedia. (n.d.). Retrieved September 5, 2013, from http://en.wikipedia.org /wiki/Mnemonic

Index