Introduction

Last time we covered Strings, Methods, Loops, Control Structures, and the beginnings of Object Oriented Programming. Today we are going to take a detour away from Java and talk about The Binary System, Two's Complement, Sign Mantissa form, Circuits, Assembly, and Computer Organization.

The Binary System

When you were growing up you were taught to count and perform mathematical operations with the Decimal System. It turns out there are many more numbering systems that exist, usually denoted as Base-# so Decimal is base 10 since there are ten digits used to express values, these being 0-9. When we started creating computers we needed a new system to work with so Base 2 or Binary was created and it is what Computers use to process data. As the Base 2 probably gives away there are only 2 valid digits for binary, those being 0 and 1. So the following is an example of a binary number:

```
10101101
```

Converting from Binary to Decimal and Vice Versa

An important skill to have is the ability to translate between binary numbers and decimal numbers and the reverse too. To go from binary to decimal you simply start from the right of the number and add up powers of two whenever there is a 1 in its place with the right most value being 2^0 and the power increasing every time we move to the left. Let's try a couple of examples!

First let's convert the following binary numbers to decimal:

```
10101101 -> 173

Starting from the rightmost bit and adding the powers of two
where there is a 1 we get:
```

$2^0+2^2+2^3+2^5+2^7 = 173$

11000101 -> 197

Same process as the last one we get:
$2^0 + 2^2 + 2^6 + 2^7 = 197$

Now let's go from a Decimal Number to Binary! The process is a little different but still straightforward. First find the value n that gives closest power of 2 greater than or equal to your number so if 63 the value will be n = 6 if the value is 65 then n = 7, this value n denotes the number of bits your binary number will have and after this you find the biggest power of 2 that fits in your number and make its equivalent bit a 1 and continue this process until the number is 0. Let's try a couple examples:

117 -> 01110101 (the leading zero is only there to keep our 8-bit values consistent)

Using the process described above we first must find n, which in this case is 7 since 2^7 is the closest power of 2 that is greater than or equal to our number. The closest power of 2 that will fit into our number is 2^6 so we will turn the 7th bit from the right to a 1 117-64 is 53 with the next power of 2 being 2^5 so same process, the 6th bit from the right is turned into a 1 and 53-32 is 21 repeat until the value reaches 0 and you get the value above.

127 -> 01111111 (the leading zero is only there to keep our 8-bit values consistent)

Using the process described above we first must find n, which in this case is 7 since 2^7 is the closest power of 2 that is greater than or equal to our number. The closest power of 2 that will fit into our number is 2^6 so we will turn the 7th bit from the right to a 1 127-64 is 63 with the next power of 2 being 2^5 so same process, the 6th bit from the right is turned into a 1 and 63-32 is 31 repeat until the value reaches 0 and you get the value above.

Two's Complement

Originally when we first started using binary representation the leftmost bit denoted the sign of the binary number with a 1 denoting a negative number and a 0 denoting a positive number. The problem with this is that there were now two different zeroes 1000 being -0 and 0000 being +0 which is fundamentally incorrect. One trick to getting around this was a system known as Two's Complement. Two's Complement is a simple concept and it's easy to grasp:

Start at 000 (or however many bits you have I am using 3 for the simplicity) the amount of values you can represent after the 0 is 2^3-1 which is 7 take that number divide it into 2 and give the extra value to the negative subset of values so we get the following:

```
000 -> 0  | 001 -> +1 | 010 -> +2 | 011 -> +3 | 100 -> -4 |
101 -> -3 | 110 -> -2 | 111 -> -1
```

In order to give the two's complement representation of a negative number, all we do is start from the positive version of the number in binary, then we invert all the bits and simply add 1. Let's try a couple of examples!

```
Give the Two's Complement Representation of -124
First we must write out the binary representation of 124 as
practiced above:
                           01111100
Now we invert all the bits: 10000011 and then we add 1:
10000100 so our final answer is: 10000100

Give the Two's Complement Representation of -35
First we must write out the binary representation of 35 as
practiced above:
                           0100011
Now we invert all the bits: 1011100 and then we add 1:
1011101 so our final answer is: 1011101
```

Sign-Mantissa Form for Fractional Numbers

Eventually we will have had to mention fractional numbers and whether or not they can be represented in binary and the answer is yes, there is just a bit more work involved. The first step of which is to convert the number to scientific notation: $\pm M \times B^{\pm E}$ We will detail this process using the value +5.75 and also assume that we will use 16 bits to represent the value with 10 bits allocated for representing the mantissa and 6 bits for the exponent.

Let's start with converting 5 to binary: 101 now to add on the fractional bit we use the exact same logic insert a binary point and continue on, except we now have the halves, quarters and eighths places rather than the tenths, hundredths and thousandths place

.75 = .11 since (½ + ¼ = ¾) so our final value for 5.75 is 101.11 but we do not want the decimal point there so we must normalize the number so the first significant digit is immediately to the right of the binary point

$$\begin{aligned} 5.75 &= 101.11 \times 2^0 \\ &= 10.111 \times 2^1 \\ &= 1.0111 \times 2^2 \\ &= .10111 \times 2^3 \end{aligned}$$

We now have our exponent which is 3 and we have our mantissa which is 10111 now using the placeholders described above with the leftmost bit of each fragment representing the sign we get the following:

0|101110000|0|00011

I have added the bars to express the different segments here, the first segment being the sign of the mantissa, in this case it is positive so it has a 0 in its spot. The second segment is the mantissa itself. All you do is plop the number down there and add zeros after it until there is a total of 9 bits in that segment. The third segment is the sign of the exponent which in this case is positive so a 0 is in its spot. The final segment is the value of the exponent and this you treat like a regular binary value, starting from the right and moving left.

Circuits, Logic Gates and Truth Tables

Circuits are an integral part of Computer Engineering and theoretical Computer Science and as such it is important for you to understand how circuits work and how to construct them. Circuits have 4 key components, those being inputs, wires, logic gates, and the output.

Let's focus on logic gates first. There are plenty of logic gates within the realms of Computer Science and Computer Engineering but here are the three most important ones to know:

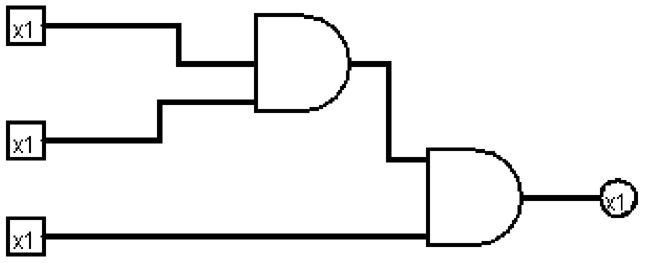
1. AND: Represented by && in Java
2. OR: Represented by || in Java
3. NOT: Represented by ! in Java

All of these should be familiar to you at this point as you've used all of them in prior assignments. All of this is part of Boolean Logic and an addition to Boolean Logic that we will discuss now are Truth Tables which allows us to display the result of boolean expressions. Let's try one now!

```
Give the truth table for the following:(x || z) && (y || !z)
First write out all columns needed then break the boolean
into chunks and solve them from there, the following is how
you should approach this problem:
```

x	y	z	x || z	y || !z	(x || z) && (y||!z)
0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	1	1	1
0	1	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1
0	0	0	0	1	0

Now we can go back to Circuit design, below is an example of a constructed circuit, let’s dissect the components and then do some practice converting booleans to circuits and vice versa.



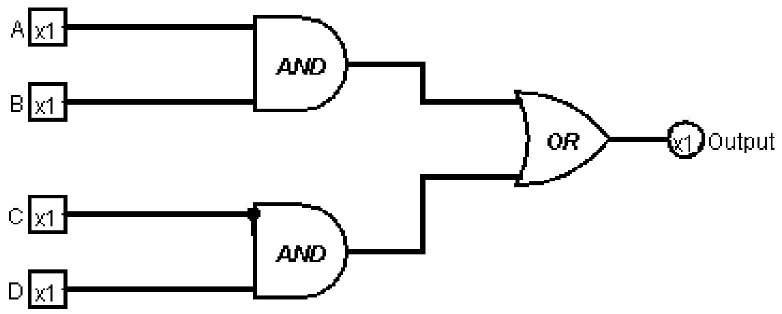
Equivalent Boolean Expression: ((A&&B)&&C)

In this schematic we can see 3 square nodes which represent our inputs, a single circular node that represents the output and two logic gates, these two gates are AND gates. Let’s practice converting between circuits and booleans and vice versa:

```
Given the following boolean expression construct a circuit
representation:  (A&&B) || (C&&D)
```

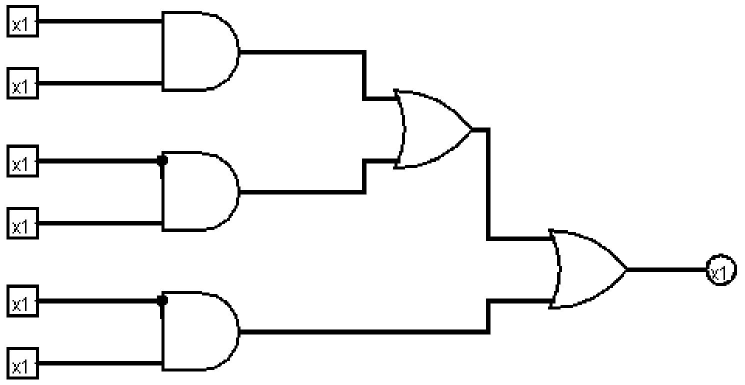
1. `First Step: Determine the number of Inputs you have (In this case we have 4: A, B, C, and D)`
2. `Second Step: Determine what logic gates you’ll need (In this case we will need 2 AND Gates and an OR gate)`
3. `Third Step: Split into smaller circuits, construct and then combine results together`

```
Using this logic lets see what the final circuit will look
like!
```

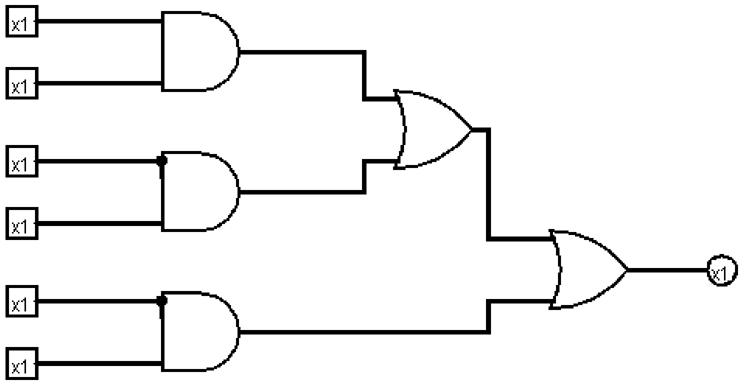


Equivalent Boolean: (A&&B) || (C&&D)

Now it is to go in reverse, given the following circuit provide the proper boolean expression:



Step 1.) Recognize there are 3 pairs of inputs: A,B,C,D,E, and F
Step 2.) Connect each pair with their logic gate: A&&B C&&D E&&F
Step 3.) Connect each connections with their logic gate: (((A&&B) || (C&&D)) || (E&&F))



Equivalent Boolean Expression: (((A&&B) || (C&&D)) || (E&&F))

Assembly Language

Assembly is a low level language that was used early on in the development of Computers and Computer Programming, there are many different versions of assembly language each with their own set of commands the list of which is below along with a snippet of code of what they do:

1. LOAD X - Loads the value at address x to Register R
2. STORE X - Stores the value in Register R to address X

3. MOVE X,Y - Copies the contents of X and puts them in Y
4. ADD X,Y,Z - Performs X+Y and stores on Z
5. ADD X,Y - Adds X to the value at Y
6. ADD X - Adds X to the value currently in Register R
7. COMPARE X,Y - Compares the values at X and Y
8. JUMP X - Performs an unconditional jump to location X
9. JUMPGT/LT/EQ/GE/LE/NEQ - Performs a conditional jump
10. HALT - Program Stops

These commands can allow you to write a lot of programs, and we will use these commands to translate Java code to Assembly language. Let's try a couple of examples. (v,w,x,y,z = 200,201.202,203, and 204 respectively)

Write out the Assembly Instructions for the following code segment: if(x < y) then set v to z else set v to y

Memory Location	OP Code	Address Field	Comment
50	COMPARE	202, 203	Compares the values of x and y
51	JUMPLT	54	Jump to Location 54 if x < y
52	MOVE	203, 200	Copies contents of y onto v
53	JUMP	55	Jump to location 55
54	MOVE	204, ,200	Copies the contents of z onto v

Write the Assembly code instructions for the following code segment: while(y >= z) set y to the value of w+y set z to the value of z+y End of Loop

Memory Location	OP Code	Address Field	Comment
50	COMPARE	203, 204	Compares the values of y and z
51	JUMPLT	59	Jump to Location 59 if y < z
52	LOAD	203	Register R now has the contents of y
53	ADD	201	Register R now has the contents of y+w
54	STORE	203	The value has been stored to y
55	LOAD	204	Register R now has the contents of z
56	ADD	203	Register R now has the contents of z+y
57	STORE	204	The value has been stored to z
58	JUMP	50	Jump to location 50 to repeat loop
59	….	…	….

Computer Organization: Von Neumann Architecture

The organization and structure of all modern computer systems are based on a theoretical model called the Von Neumann architecture. The Von Neumann architecture is based on the following principles:

1. The stored program concept, in which the instructions to be executed by the computer are represented as binary and stored in memory
2. Four major subsystems, memory, Input/Output, Arithmetic Logic Unit, and the Control Unit
3. The sequential execution of instructions, in which one instruction is taken from memory and passed to the control unit where it is decoded and executed.

Now each of the four major subsystems will be described, more information can be found in Chapter 5 Section 2 of the Invitation textbook.

Memory

Memory is the functional unit of a computer that stores and retrieves instructions and data. All information stored in memory is represented internally using binary. Computer memory uses an access technique called random access, you know it as random access memory (RAM) which has the following characteristics:

- The time it takes to store or retrieve contents of a single cell is the same for all cells in memory
- All accesses to memory are to a specific address, you must always store or retrieve a complete cell.
- Memory is divided into fixed-size units called cells and each cell is associated with an unique identifier known as an address which comprises unsigned integer values.

Memory is made up of cells that contain a fixed number of binary digits, the number of bits per cell is called the memory width and we denote this with a *W* the standard width for a cell nowadays is 8 with there being 8 bits in a byte! When considering how many possible addresses there can be in memory let's consider N to be the number of bits available to represent the address, that would mean you have 2^N-1 possible addresses you have to subtract 1 since 0 is considered a valid address. The memory unit contains two special registers, the MAR which holds the address of the cell to be accessed, and the MDR which contains the data value being stored or retrieved the size of the MDR are usually multiples of *W* usually being n*W* traditionally n*W* holds the values of 32 and 64 (hopefully these trivial values ring some bells)

Input/Output and Mass Storage

The input/output (I/O) units are the devices that allow a computer system to communicate and interact with the outside world as well as store information for the long term. Of all the components of a Von Neumann machine, the I/O and mass storage subsystems are the most ad hoc and the most variable. Input/output devices come in two basic types: those that represent information in human-readable form for human consumption and those that store information in machine-readable form for access by a computer system. The former includes such well-known I/O devices as keyboards, both physical and virtual, screens, and printers. The latter group of devices includes flash memory, hard drives, DVDs, and streaming tapes.

Mass storage devices themselves come in two distinct forms: direct access storage devices (DASDs) and sequential access storage devices (SASDs).

A direct access storage device (DASD) is one in which requirement number 2, equal access time has been eliminated. That is, in a direct access storage device, every unit of information still has a unique address, but the time needed to access that information depends on its physical location and the current state of the device A magnetic disk stores information in units called sectors, each of which contains an address and a data block containing a fixed number of bytes. A fixed number of these sectors are placed in a concentric circle on the surface of the disk, called a track.

A sequential access storage device (SASD) does not require that all units of data be identifiable via unique addresses. To find any given data item, we must search all data sequentially.

One of the fundamental characteristics of many (although not all) I/O devices is that they are very, very slow when compared with other components of a computer.

The Arithmetic Logic Unit

The arithmetic/logic unit (ALU) is the subsystem that performs such mathematical and logical operations as addition, subtraction, and comparison for equality. The one-bit adder that was in the review session and the one-bit subtractor that was completed on the problem set last week, are just a fraction of what makes up the ALU.

The ALU is made up of three parts: the registers, the interconnections between components, and the ALU circuitry. Together these components are called the data path.

The Control Unit

The most fundamental characteristic of the Von Neumann architecture is the stored program—a sequence of machine language instructions stored as binary values in memory. It is the task of the control unit to (1) fetch from memory the next instruction to be executed, (2) decode it—that is, determine what is to be done, and (3) execute it by issuing the appropriate command to the ALU, memory, or I/O controllers. These three steps are repeated over and over until we reach the last instruction in the program in your case this command is HALT

Together the Control Unit and the ALU construct what we all know as the Central Processing Unit or CPU

Conclusion

That is the end of the notes session, next time we will discuss good design habits for your classes and methods, and provide a brief introduction to arrays. See you next time!