How to create your own virtual machine! Part I

Presented by: Alan L. Bryan
A.k.a. Icemanind
Questions? Comments? Email me at
icemanind@yahoo.com

Please leave feedback if you enjoyed this tutorial. The more feedback I get, the more it'll make me want to write Part II

Introduction

Welcome to my tutorial on virtual machines. This tutorial will introduce you to the concept of a virtual machine and then we will, step by step, create our own simple virtual machine in C#. Keep in mind that a virtual machine is a very complicated thing and even the simplest virtual machine can take years for a team of programmers to create. With that said, don't expect to be able to create your own language or virtual machine that will take over .NET or Java overnight.

In this tutorial, we will first layout the plan for our virtual machine. Then we will create a very simple *intermediate language*. An intermediate language is the lowest level language still readable by humans. It is comparable to assembly language, which is also the lowest level language on most computers. The first program we will create will be a very simple intermediate compiler that will convert our intermediate language to *bytecode*. Bytecode is a set of binary instructions that our virtual machine will be able to directly execute. It is comparable to machine language, which is a set of binary or machine instructions that all computers and CPUs understand. This virtual machine will be our second project. It will be a virtual machine, created from scratch in C# that will execute our bytecode. It will be very simple at first, but then we will expand it by adding threading support and dual screen outputs (you'll find out what I'm talking about later).

All of the code in this tutorial is created using Visual Studio 2008 Professional, targeting the .NET Framework 2.0. Since I'm targeting the 2.0 framework, you should be able to use Visual Studio 2005 as well. Since creating a virtual machine really does dive down into the nuts and bolts of how computers work, I am assuming the reader of this has a pretty good, or a basic knowledge of, programming, hexadecimal and binary number systems, and threading. It would also really help to know something about assembly language, although I will try to help you understand things on a need-to-know basis.

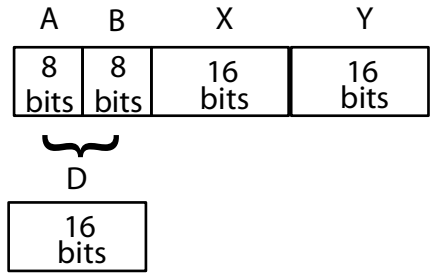
If I haven't scared you off and you're still interested in how to make a virtual machine, then let's begin!

Planning it out

As described in the introduction, the first thing we will want to do is draw out a rough blue print of what our machine will be able to do. I have decided to call our machine, B32 (Binary 32), although, for simplicity's sake it will not be a 32-bit machine. It will be a 16-bit machine. B32 will have 64K of memory and it can be addressed anywhere from $0000 - $FFFF. A B32 executable program can access any part of that memory. Along with a 64K memory space, we will introduce 5 *registers* into our virtual machine. All CPU's and all virtual machines have what's called registers. A register is similar to a variable. Registers hold numbers and depending on how large the register is, determines how large of a number it can hold. Unlike variables, however, registers do not take up memory space. Registers are "built into" CPUs. This will make more sense once you see an example, which is coming up real soon.

To keep things simple, we will only implement 5 registers into our virtual machines. These registers will be called A, B, D, X and Y. The A and B registers are only 8 bits in length, which means each register can hold any number between 0 and 255 unsigned or between -128 to 127 signed. For now, we are going to worry only about unsigned integers. We will get into signed later and we will briefly touch on floating point numbers later. The X, Y and D registers will be 16 bits in length, capable of storing any number between 0 and 65,535 unsigned or between -32768 to 32767 signed. The D register will be something of a unique register. The D register will hold the concatenated values of the A and B registers. In other words, if register A has $3C and register B has $10, than register D will contain $3C10. Anytime a value in the A or B register is changed, then the value in the D register is also changed. The same is true if a value in the D register is changed, the A and B registers will be changed accordingly. You will see later why this is handy to have.

This has been a lot of dry talk, but here is a picture to represent our B32 registers:



Hopefully this makes sense to you. If not, you will catch on as we progress through the tutorial.

Earlier when I told you that our virtual machine had 64K of free memory for an executable to use, that was not entirely true. Really it's only 60K because 4000 bytes must be reserved for screen

output. I've chosen to use $A000 - $AFA0. This area of memory will map to our screen. In most CPUs and most virtual machines, this memory is mapped inside the video card memory, however, for simplicity; I am going to share our 64K of memory with our video output. This memory will give us an 80x25 screen (80 columns, 25 rows). You may be thinking right now, "I think your math is off dude. 80 times 25 is only 2000". This is true; however, the extra 2000 bytes will be for an *attribute*.

For those of us old enough to remember programming assembly language, back in the old DOS days, will already be familiar with an attribute byte. An attribute byte defines the foreground and background color of our text. How it works is the last 3 bits of the byte make up the RGB or Red, Green, Blue values of our foreground color. The 4th bit is an intensity flag. If this bit is 1 then the color is brighter. The next 3 bits make up the RGB values of our background color. The last bit is not used (back in DOS days, this bit was used to make text blink, but in B32, it is ignored). You will see later how colors are created using this method.

The final part of this section will define the mnemonics and the bytecode that make up a B32 executable. Mnemonics are the building block of our assembly language code that will be assembled to bytecode. For now, I am only going to introduce enough for us to get started and we will expand on our list throughout this tutorial. The first mnemonic we will introduce is called "LDA". "LDA" is short for "Load A Register" and what it will do is assign a value to the A register. Now in most CPUs and virtual machines, you have what's called *addressing modes*. Addressing modes determine how a register gets its value. For example, is the value specified directly on the *operand* (an operand is the data that follows the mnemonic) or does it pull a value from somewhere in memory or is loaded from a value assigned to another register? There can be dozens of addressing modes, depending on how complex of a virtual machine you want to create. For now, our virtual machine will only pull data directly specified in the operand. We will assign this mnemonic a bytecode value of $01. Since we decided earlier that the A register can only hold an 8 bit value, we now that the entire length of a "LDA" mnemonic that pulls direct data from the operand will be 2 bytes in length (1 byte for the mnemonic and 1 byte for the data).

The next mnemonic we will discuss will be called "LDX". "LDX" is short for "Load X Register" and, just like "LDA", it will load a value directly into the X register from the operand. Another difference between "LDX" and "LDA" is the length. Since our X register can hold 16 bits of data, that means the total length of the bytecodes will be 3 bytes instead of 2 (1 byte for the mnemonic and 2 bytes for the data). We will assign this mnemonic a bytecode of $02. If I lost you guys, keep reading and I promise this will make sense when we look at some examples.

The next mnemonic we will discuss now will be called "STA". "STA" is short for "Store A Register" and its function will be to store the value contained in the A register into a location somewhere in our 64K memory. Unlike our load mnemonics, which pulls the value directly from the operand, our store mnemonic will pull its data from the value stored in one of the 16 bit registers. We will assign this mnemonic a bytecode of $03.

The final mnemonic we will discuss is call "END". "END" will do exactly that. It will terminate the application. All B32 programs must have an END mnemonic as the last line of the program. The operand for the END mnemonic will be a label that will point to where execution of our B32 program will begin.

Here is a summary of our mnemonics:

Mnemonic	Description	Example	What will this example do?
LDA $01	Assigns a value to our A register	LDA #$2A	Assigns the hex value $2A to the A register
LDX $02	Assigns a value to our X register	LDX #16000	Assigns the number 16,000 to the X register
STA $03	Stores the value of the A register to a memory location	STA ,X	Stores the value of the A register to the memory location pointed to by the X register
END $04	Terminates the B32 program	END START	Terminate the program and tell our assembler that execution of our program should start at the START label

You may be wondering what the pound sign '#' means in my examples above. The pound sign will tell our assembler to use "this value", that is, the value immediately following the pound sign. We will introduce other forms of LDA, LDX and STA later in this tutorial, but for now, this is enough to get us started.

For those of you who may also be wondering what the dollar sign '$' means, it is a prefix that will tell our assembler that the value is in hexadecimal format. If there is no dollar sign present, then the assembler will assume the number is a regular integer number.

One final mnemonic that we will introduce is called "END". This is not really a mnemonic though. This is an assembler command that will tell our assembler "this is the end of our program". All B32 programs we created must have at least 1 and only 1 END statement and it should be the last line of the program. The operand for our END statement will be a label that points to the part of our program where execution will begin. We will discuss labels and execution points later in the tutorial.

One final piece of business we need to discuss before we get our hands dirty and start writing our assembler is the file format of our B32 executables. To keep things simple, our file format will be as follows:

Data	Length	Description
"B32"	3 Bytes	Our magic header number
<Starting Address>	2 Bytes	This is a 16-bit integer that tells our virtual machine where, in memory, to place our program.
<Execution Address>	2 Bytes	This is a 16-bit integer that tells our virtual machine where to begin execution of our program.
<ByteCode>	?? Bytes	This will be the start of our bytecode, which can be any length.

Most binary file formats have a “magic number” as a header. A magic number is one or more bytes that are unique to that file format. For example, all DOS and Windows executables start with “MZ”. Java binary class files have 4 bytes for its magic number and start with $CAFEBABE. Our B32 executables will start with “B32”. There are two main purposes for this “magic number”. The first is, our virtual machine can check to be sure the file it’s trying to execute is, indeed, a B32 binary file. The second purpose for have magic numbers is some operating systems, such as Linux for example, can automatically execute files by looking at this magic number in a database, then calling the appropriate program.

B32 Assembler

Finally! It’s time to get our hands dirty and start working on our assembler. The goal of the assembler will be to translate our B32 mnemonics into a B32 binary. Our assembler is going to expect input to be in the following format:

[Optional Label:]
<white space><mnemonic><white space><operand>[Optional white space]<newline>

A label starts with a letter and is composed of any number of letters or numbers, followed by a colon. As far as the assembler’s concerned, a label will simply be translated into a 16-bit value defining an area of memory. A white space is any number of spaces or tabs. Each mnemonic MUST be preceded by at least 1 space or 1 tab; otherwise, our assembler will treat the mnemonic as a label instead of as a mnemonic. Likewise, each mnemonic must also have at least 1 space or 1 tab after the mnemonic. To demonstrate this, we are going to create our very first B32 assembly language program right now. Open up notepad or some other text editor of your choosing and type the following program EXACTLY as you see it below and don’t forget to put a single space before each mnemonic and also be sure to end the last line with a newline:

```
START:
 LDA #65
 LDX #$A000
 STA ,X
 END START
```

Five points if you can guess what this program will do! That’s right! This program doesn’t do much except put a capital letter ‘A’ in the upper left hand corner of the screen. The first line is our label. This is where our execution will begin. The next line loads our A register with the value of 65. The ASCII value of ‘A’ is 65. The following line loads our X register with hex value $A000. If you remember from our previous discussion, we said that our video memory will start at $A000, thus defining the upper left

hand corner. The next line is the action line that actually stores the value in register A (65, which is the letter 'A' in ASCII) at the location pointed to by register X (which is $A000). The final line ends our program and tells the assembler to start execution at our START label. Hopefully this all makes sense to you. If not, scroll back up and reread the "Planning it out" section carefully.

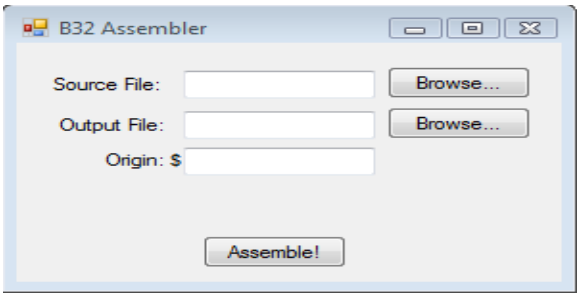
Save this file somewhere on your computer. Call it "Test.asm". We will now create the assembler that will be able to translate this code into B32 bytecode. Our assembler will work by assembling in two phases. First the assembler will load the program into memory. Then it will start phase one of the assemble process. This phase will scan for all labels we have in the program. Each time the assembler encounters a label, it will store the label as a key in a hash table and the current location in the program as its value. A hash table is a type of .NET collection that stores values based on unique keys. This is a perfect collection to use for gather labels, since each label name must be unique in our program. Once this is complete, the assembler will move onto phase two. Phase two will actually translate our mnemonics into bytecode.

Okay, fire up Visual Studio and create a new C# Windows Form project called "B32Assembler". Target the 2.0 framework or higher. Open up Form1 and change the name to frmMainForm. Resize it so that the width is 300 and the height is 207. Add the following controls to the form:

Control Type	Name	Location	Size	Other properties
Label	Label1	X = 16 Y = 23	Autosize	Text = "Source File:"
Label	Label2	X = 20 Y = 52	Autosize	Text = "Output File:"
Label	Label3	X = 44 Y = 77	Autosize	Text = "Origin:"
Label	Label4	X = 77 Y = 77	Autosize	Text = "$"
TextBox	txtSourceFileName	X = 87 Y = 20	W = 100 H = 20	Text = ""
TextBox	txtOutputFileName	X = 87 Y = 49	W = 100 H = 20	Text = ""
TextBox	txtOrigin	X = 87 Y = 75	W = 100 H = 20	Text = ""
Button	btnAssemble	X = 97 Y = 138	W = 75 H = 23	Text = "Assemble!"
Button	btnSourceBrowse	X = 193 Y = 17	W = 75 H = 23	Text = "Browse..."
Button	btnOutputBrowse	X = 193 Y = 46	W = 75 H = 23	Text = "Browse..."
OpenFileDialog	fdDestinationFile	N/A	N/A	Filter = "B32 Files|*.B32" DefaultExt = "B32" Filename = ""

				CheckFileExists = False
OpenFileDialog	fdSourceFile	N/A	N/A	Filter = "B32 Assembly Files|*.asm" DefaultExt = "asm" Filename = ""

Your form should look like the following:



Now double click the top browse button to create a "click" event handler for the button, then type in the following code:

```
        private void btnSourceBrowse_Click(object sender, EventArgs e)
        {
            this.fdSourceFile.ShowDialog();
            this.txtSourceFileName.Text = fdSourceFile.FileName;
        }
```

Now go back to designer view and double click the second browse button, then type in the following code:

```
        private void btnOutputBrowse_Click(object sender, EventArgs e)
        {
            this.fdDestinationFile.ShowDialog();
            this.txtOutputFileName.Text = fdDestinationFile.FileName;
        }
```

What this will do is allow the user to browse for a source and output file. If you run the program now and click one of the browse buttons, it should pop up with a dialog box allowing you to find and choose a file. Once you select a file and click OK, the filename should pop into the appropriate text box. The origin will be where, in our 64K memory region, you want the program to be placed.

Now that we got our interface wired up, let's add the main functionality. Double click on the "Assemble!" button to create a click event handler. Before coding the event handler though, add the following class members to our class:

```
    public partial class frmMainForm : Form
    {
        private string SourceProgram;
```

```
        private System.Collections.Hashtable LabelTable;
        private int CurrentNdx;
        private ushort AsLength;
        private bool IsEnd;
        private ushort ExecutionAddress;
```

The `SourceProgram` variable will hold our program in memory. The B32 assembler will read our source file and dump the contents into the `SourceProgram` variable. As described earlier, `LabelTable` is a hash table that will hold our labels. The hash table will be populated during the first stage of the assembly. `CurrentNdx` will be an integer variable that will be an index pointer to the current location in the file. `AsLength` will be an unsigned 16-bit variable that will keep track of how big our binary program is. `IsEnd` is simply a flag to determine if the end of the program has been reached. Finally, `ExecutionAddress` will hold the value of our execution address. If some of this doesn't make sense yet, it will as we code our program.

We are also going to need an enumeration that will store our registers. Add the following enumeration just below the code you just entered:

```
    private enum Registers
        {
            Unknown = 0,
            A = 4,
            B = 2,
            D = 1,
            X = 16,
            Y = 8
        }
```

We will put this enumeration to use later on when we start coding our helper functions. We will create a function that will read a register from our program and return an enumeration type representing the register.

Finally, before we start coding our Assemble button handler, add the following lines to the frmMainForm class constructor. These lines will automatically initialize our variables we added earlier, assign a default origin, and allocate memory for our hash table:

```
        public frmMainForm()
        {
            InitializeComponent();

            LabelTable = new System.Collections.Hashtable(50);
            CurrentNdx = 0;
            AsLength = 0;
            ExecutionAddress = 0;
            IsEnd = false;
            SourceProgram = "";
            txtOrigin.Text = "1000";
        }
```

Okay, now add the following code to the Assemble click handler that we made earlier. Type in the code as you see it and then I will explain what it does:

```
        private void btnAssemble_Click(object sender, EventArgs e)
        {
            AsLength = Convert.ToUInt16(this.txtOrigin.Text, 16);

            System.IO.BinaryWriter output;
            System.IO.TextReader input;
            System.IO.FileStream fs = new
System.IO.FileStream(this.txtOutputFileName.Text, System.IO.FileMode.Create);

            output = new System.IO.BinaryWriter(fs);

            input = System.IO.File.OpenText(this.txtSourceFileName.Text);
            SourceProgram = input.ReadToEnd();
            input.Close();

            output.Write('B');
            output.Write('3');
            output.Write('2');
            output.Write(Convert.ToUInt16(this.txtOrigin.Text, 16));
            output.Write((ushort)0);
            Parse(output);

            output.Seek(5, System.IO.SeekOrigin.Begin);
            output.Write(ExecutionAddress);
            output.Close();
            fs.Close();
            MessageBox.Show("Done!");
        }
```

The first thing we are doing is grabbing our origin address and converting it into an unsigned 16 bit value and assigning that to `AsLength`. Next we are creating a BinaryWriter stream for our output and a TextReader stream for our input, then we are opening the output stream, creating the file if it does not already exist or overwriting it if it does exist. Next, we are opening the source file and read the entire contents and storing it into `SourceProgram`, then closing the input buffer. The next 3 lines create the header and magic numbers for our B32 binary file format, as we discussed earlier. Also, as we discussed earlier, our file header will contain the string 'B32', followed by the starting address and the execution address. You can see that we writing the stringing address to the file, however you may be confused as to why we are writing zero as the execution address. This simply serves as a placeholder for now, since we do not yet know the execution address. We will come back to this spot after we parse the source file and write the correct address. Next we call the Parse() function. We have not written this function yet, so I will hold off on discussing the details for that function. Finally, as promised, we are seeking back to the execution address and writing the correct address, then we close the buffers and we are done! Pretty simple, huh? Well I am hiding a lot of details, so let's move on and discuss those details in depth.

The Parse() function will also be a pretty simple function. It will simply scan our file for labels, then scan our file again and compile it (2 phases, as discussed earlier):

```
        private void Parse(System.IO.BinaryWriter OutputFile)
        {
            CurrentNdx = 0;
            while (IsEnd == false)
                LabelScan(OutputFile, true);

            IsEnd = false;
            CurrentNdx = 0;
            AsLength = Convert.ToUInt16(this.txtOrigin.Text, 16);

            while (IsEnd == false)
                LabelScan(OutputFile, false);
        }
```

Pretty simple. It first resets the `CurrentNdx` to zero, then enters a loop that calls LabelScan() until the end of the file has been reached. It then resets the `IsEnd` flag to false, `CurrentNdx` back to zero and `AsLength` back to the starting address. Finally it starts on the second pass and actually writes the bytecode for our output file.

Next, we need to write the code for our LabelScan() function:

```
        private void LabelScan(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {

            if (char.IsLetter(SourceProgram[CurrentNdx]))
            {
                // Must be a label
                if (IsLabelScan) LabelTable.Add(GetLabelName(), AsLength);
                while (SourceProgram[CurrentNdx] != '\n')
                    CurrentNdx++;
                CurrentNdx++;
                return;
            }
            EatWhiteSpaces();
            ReadMneumonic(OutputFile, IsLabelScan);
        }
```

Our LabelScan() function starts out by checking the first character in the line. Remember earlier I told you that in our source file, each mnemonic MUST be preceded with a space. This is why. Our assembler looks at the first character and if it's not a space, it assumes it's a label. The program then decides if it's on pass 1 or pass 2 (determined by our `IsLabelScan` flag variable) and if it's on pass 1, it adds the label to our `LabelTable` hash table. The GetLabelName() function is one of many helper functions we will create later. For now, just know that GetLabelName() will simply retrieve the name of the label. After our assembler finds a label, it continues to basically "eat" characters till it finds a newline character (because remember that after the label, there should be nothing else on the line). If our LabelScan() function does not find a label, then it calls the EatWhiteSpaces() function (another helper function) to "eat" the white spaces. It then calls the next function we are going to code, ReadMnemonic().

The ReadMnemonic() function does exactly what it sounds like. It reads in the next mnemonic waiting to be read. This function is presented here now:

```
        private void ReadMneumonic(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            string Mneumonic = "";

            while (!(char.IsWhiteSpace(SourceProgram[CurrentNdx])))
            {
                Mneumonic = Mneumonic + SourceProgram[CurrentNdx];
                CurrentNdx++;
            }

            if (Mneumonic.ToUpper() == "LDX") InterpretLDX(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "LDA") InterpretLDA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "STA") InterpretSTA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "END") { IsEnd = true;
DoEnd(OutputFile,IsLabelScan); EatWhiteSpaces(); ExecutionAddress =
(ushort)LabelTable[(GetLabelName())]; return; }

            while (SourceProgram[CurrentNdx] != '\n')
            {
                CurrentNdx++;
            }
            CurrentNdx++;
        }
```

The ReadMneumonic() function should be pretty self explanatory. It reads the mnemonic, then compares it against several `if` statements. These `if` statements call various functions to interpret the mnemonic. After interpreting the mnemonic, it eats characters till if finds the end of the line.

Of the three interpreting functions we have referenced so far, the first one I am going to introduce here is the InterpretLDA() function:

```
private void InterpretLDA(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                byte val = ReadByteValue();
                AsLength += 2;
                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x01);
                    OutputFile.Write(val);
                }
            }
        }
```

Again, this function is pretty simple to figure out. First it eats all white spaces. Then it checks to see if the operand begins with a pound sign '#' (recall earlier, I said that whenever our assembler encounters a pound sign in the operand, it means to use "this value", or in better terms, you the value

immediately following the pound sign). If it does, then it increments pass the pound sign, then calls the ReadByteValue() function. This is another helper function that reads the 8-bit value immediately after the pound sign and assigns it to `val`. It then increments our length pointer by 2. Remember earlier I said that the LDA mnemonic would consume 2 bytes of memory; one for the mnemonic itself and another for the actual byte value. If we are on phase 2 and not just scanning for labels, then the InterpretLDA() function with then write a $01 byte and the value we loaded into the register (recall that we said earlier that $01 will be the bytecode we assign to the LDA mnemonic).

A similar function is presented here now, InterpretLDX():

```
        private void InterpretLDX(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                ushort val = ReadWordValue();
                AsLength += 3;
                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x02);
                    OutputFile.Write(val);
                }
            }
        }
```

Similar to its counterpart InterpretLDA(), InterpretLDX() works almost exactly the same. The only differences are, for one, we are reading a 16-bit word value instead of an 8-bit byte value. Second, we incrementing our length by 3 instead of by 2 since the X register can load a 16-bit value. Also, notice we are writing $02 instead of $01 since $02 is the bytecode assigned to LDX.

The next of the interpret functions is InterpretSTA():

```
        private void InterpretSTA(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == ',')
            {
                Registers r;
                byte opcode = 0x00;

                CurrentNdx++;
                EatWhiteSpaces();
                r = ReadRegister();
                switch (r)
                {
                    case Registers.X:
                        opcode = 0x03;
                        break;
                }
                AsLength += 1;
```

```
                if (!IsLabelScan)
                {
                    OutputFile.Write(opcode);
                }
            }
        }
```

Remember our format for storing the value of the A register to a memory location pointed to by the X register "STA ,X". The first thing this function does is check for a comma. It then eats any white space then calls the ReadRegister() function. This is a helper function that will return the appropriate register enumeration. It then writes the bytecode and increments our `AsLength` variable by one.

The last of our interpret functions (for now) is DoEnd:

```
        private void DoEnd(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            AsLength++;
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x04);
            }
        }
```

Again, this is pretty simple. It simply increments our `AsLength` variable and writes a $04 byte.

Our assembler is almost done. All we need to do now is define our helper functions. First, here is the code to the ReadRegister() function:

```
        private Registers ReadRegister()
        {
            Registers r = Registers.Unknown;

            if ((SourceProgram[CurrentNdx] == 'X') ||
                (SourceProgram[CurrentNdx] == 'x')) r = Registers.X;
            if ((SourceProgram[CurrentNdx] == 'Y') ||
                (SourceProgram[CurrentNdx] == 'y')) r = Registers.Y;
            if ((SourceProgram[CurrentNdx] == 'D') ||
                (SourceProgram[CurrentNdx] == 'd')) r = Registers.D;
            if ((SourceProgram[CurrentNdx] == 'A') ||
                (SourceProgram[CurrentNdx] == 'a')) r = Registers.A;
            if ((SourceProgram[CurrentNdx] == 'B') ||
                (SourceProgram[CurrentNdx] == 'b')) r = Registers.B;

            CurrentNdx++;
            return r;
        }
```

This function simply reads the next character in the input and returns the appropriate enumeration. Simple enough to figure out how it works.

Next, we will define our ReadWordValue() function:

```
        private ushort ReadWordValue()
        {
            ushort val = 0;
            bool IsHex = false;
            string sval = "";

            if (SourceProgram[CurrentNdx] == '$')
            {
                CurrentNdx++;
                IsHex = true;
            }

            while (char.IsLetterOrDigit(SourceProgram[CurrentNdx]))
            {
                sval = sval + SourceProgram[CurrentNdx];
                CurrentNdx++;
            }
            if (IsHex)
            {
                val = Convert.ToUInt16(sval, 16);
            }
            else
            {
                val = ushort.Parse(sval);
            }

            return val;
        }
```

This function is also pretty simple to figure out. It first checks for a dollar sign '$'. The dollar sign signals to the function that the number about to be read in is a hexadecimal number and not an integer. It then reads in the number and converts it to an unsigned short and returns the value.

The next function is the sister function to ReadWordValue(), called ReadByteValue():

```
        private byte ReadByteValue()
        {
            byte val = 0;
            bool IsHex = false;
            string sval = "";

            if (SourceProgram[CurrentNdx] == '$')
            {
                CurrentNdx++;
                IsHex = true;
            }

            while (char.IsLetterOrDigit(SourceProgram[CurrentNdx]))
            {
                sval = sval + SourceProgram[CurrentNdx];
                CurrentNdx++;
            }
            if (IsHex)
            {
                val = Convert.ToByte(sval, 16);
            }
```

```
            else
            {
                val = byte.Parse(sval);
            }

            return val;
        }
```

The code is almost identical, except the final value is converted into a byte instead of an unsigned short.

The next helper function is our EatWhiteSpaces() function:

```
        private void EatWhiteSpaces()
        {
            while (char.IsWhiteSpace(SourceProgram[CurrentNdx]))
            {
                CurrentNdx++;
            }
        }
```

This function is very easy to follow. It simply increments our source code pointer till it points to a character that's not a white space.

Finally, the last helper function we need, GetLabelName():

```
        private string GetLabelName()
        {
            string lblname = "";

            while (char.IsLetterOrDigit(SourceProgram[CurrentNdx]))
            {
                if (SourceProgram[CurrentNdx] == ':')
                {
                    CurrentNdx++;
                    break;
                }

                lblname = lblname + SourceProgram[CurrentNdx];
                CurrentNdx++;
            }

            return lblname.ToUpper();
        }
```

This function returns the name of the label and converts it to upper case (since B32 code is case insensitive).

Congratulations! Our primitive assembler is now complete. Go ahead and try to run the program. It should compile and run without any errors. For the source file, browse to the Test.ASM file we created earlier, and then choose an appropriate output file. Click the "Assemble!" button and it should assemble your program without any problems. After it is done, it should have created you a B32 executable file. Feel free to examine this file with a hex editor. If you do, you will notice that our B32 header is in there, along with 2 bytes for our starting address, 2 bytes for our execution address and the

remaining bytes making up our program. Feel free to play around with the assembler as much as you want. When you are ready to move on, I will show you how to make a virtual machine. The virtual machine will execute our B32 binary code. Before moving on, assemble the Test.ASM file and have Test.B32 ready to play with.

B32 Virtual Machine

Now that we got our assembler finished and we can make simple programs, it's time to create the virtual machine that will run those programs. Keeping with the KISS (Keep It Simple Stupid) philosophy, the virtual machine we are going to create here is a simple one. Basically, it will load our program into the appropriate 64K memory area, and then it will interpret each instruction. You should know however, that real life virtual machines work differently. Most modern virtual machines use a method called *dynamic recompilation*. Dynamic recompilation is a method that, rather than interpreting the bytecode, recompiles the program in the native machine's format. For example, one of the lines in our program is "LDA #65". Using dynamic recompilation, this would recompile into something like "MOV AX,65" on the x86 processors. The reason for doing this is because dynamic recompilation is much faster since the processor is actually running native code. Dynamic recompilation is way beyond the scope of this tutorial though (maybe one day, I will write a virtual machine tutorial that dynamically recompiles, depending on the demand I get for it).

Fire up Visual Studio and create a new C# Windows form project called "B32Machine". So far in this tutorial I have assumed you are familiar with C# and Visual Studio. In keeping with that assumption, there are certain things I am going to tell you do that I assume you know how to do. If following along with this tutorial turns out to be too hard, you can always download the completed source code to both the assembler and the virtual machine and just follow along with that. The source code should be available for download at the same place this tutorial was downloaded.

Okay, the first thing you will want to do is rename Form1 to MainForm. Resize the form so the width is 660 and the height is 394. Next, right click on your project and add a new user control. Call this control "B32Screen". This control will represent the output screen of the B32 Virtual Machine. Recall earlier I mentioned that we are going to use 4,000 bytes of our 64K memory for screen output. This control will represent those 4,000 bytes. If you need to refresh your memory on how we decided our virtual machine screen will work, review that now.

Set the background color of the control to Black and resize it so the width is 429 and the height is 408. From the design stand point, we are done with the screen! See how easy that was? Now comes the hard part, the code. Now really hard though. Okay, switch over to the code view and add the following private members and property:

```
public partial class B32Screen : UserControl
{
    private ushort m_ScreenMemoryLocation;
```

```
        private byte[] m_ScreenMemory;

        public ushort ScreenMemoryLocation
        {
            get
            {
                return m_ScreenMemoryLocation;
            }
            set
            {
                m_ScreenMemoryLocation = value;
            }
        }
```

The ScreenMemoryLocation property and the `m_ScreenMemoryLocation` member store the address of where, in our 64K memory, our screen will reside. As you will see in the constructor, presented below, it will default to $A000. By making this dynamic, in the form of a property, rather than hard coding $A000, it will allow us to use multiple screens, as we will demonstrate later in this tutorial. The `m_ScreenMemory` variable will be a byte array that holds the text and attributes.

Next, add the following to the B32Screen constructor:

```
        public B32Screen()
        {
            InitializeComponent();
            m_ScreenMemoryLocation = 0xA000;
            m_ScreenMemory = new byte[4000];

            for (int i = 0; i < 4000; i += 2)
            {
                m_ScreenMemory[i] = 32;
                m_ScreenMemory[i + 1] = 7;
            }
        }
```

As promised, I am first initializing the default screen location to $A000. Next, I am allocated 4,000 bytes to our array and finally, I am initializing the values in the array. I am using 32 (which is ASCII for a blank space) for the character and an attribute of 7. An attribute of 7 will produce gray text on a black background, just like the old DOS computers used to do. Essentially, I am clearing the screen. Review our discussion on attributes up above, if this is confusing.

Next, we are going to add a public method called “Poke”. Poke will load a value into the area specified in memory, within the range of our screen address. It is presented here:

```
        public void Poke(ushort Address, byte Value)
        {
            ushort MemLoc;

            try
            {
                MemLoc = (ushort)(Address - m_ScreenMemoryLocation);
            }
            catch (Exception)
```

```
            {
                return;
            }

            if (MemLoc < 0 || MemLoc > 3999)
                return;

            m_ScreenMemory[MemLoc] = Value;
            Refresh();
        }
```

This method first tries to create an offset and returns if there is an error (if there is an error, it's because the address being poked is not within our screen range), then it checks to make sure it's in range, then "pokes" to value into the memory location. Finally, the control refreshes itself to update.

Poke's sister method is called "Peek". Peek will return the byte value stored in any video memory location. Add this method to our program:

```
        public byte Peek(ushort Address)
        {
            ushort MemLoc;

            try
            {
                MemLoc = (ushort)(Address - m_ScreenMemoryLocation);
            }
            catch (Exception)
            {
                return (byte)0;
            }

            if (MemLoc < 0 || MemLoc > 3999)
                return (byte)0;

            return m_ScreenMemory[MemLoc];
        }
```

This method is similar in the sense that it first tries to create an offset, validates the range, then returns the appropriate byte.

The final thing we need to do to finish our B32Screen user control is to add code for the paint handler. Switch back to designer mode, then create an event handler for the B32Screen Paint event. The code for the paint handler is a little long, but after I present it, I will explain in detail how it works. Here it is:

```
        private void B32Screen_Paint(object sender, PaintEventArgs e)
        {
            Bitmap bmp = new Bitmap(this.Width, this.Height);
            Graphics bmpGraphics = Graphics.FromImage(bmp);
            Font f = new Font("Courier New", 10f, FontStyle.Bold);
            int xLoc = 0;
            int yLoc = 0;

            for (int i = 0; i < 4000; i += 2)
```

```
            {
                SolidBrush bgBrush = null;
                SolidBrush fgBrush = null;

                if ((m_ScreenMemory[i + 1] & 112) == 112)
                {
                    bgBrush = new SolidBrush(Color.Gray);
                }
                if ((m_ScreenMemory[i + 1] & 112) == 96)
                {
                    bgBrush = new SolidBrush(Color.Brown);
                }
                if ((m_ScreenMemory[i + 1] & 112) == 80)
                {
                    bgBrush = new SolidBrush(Color.Magenta);
                }
                if ((m_ScreenMemory[i + 1] & 112) == 64)
                {
                    bgBrush = new SolidBrush(Color.Red);
                }
                if ((m_ScreenMemory[i + 1] & 112) == 48)
                {
                    bgBrush = new SolidBrush(Color.Cyan);
                }
                if ((m_ScreenMemory[i + 1] & 112) == 32)
                {
                    bgBrush = new SolidBrush(Color.Green);
                }
                if ((m_ScreenMemory[i + 1] & 112) == 16)
                {
                    bgBrush = new SolidBrush(Color.Blue);
                }
                if ((m_ScreenMemory[i + 1] & 112) == 0)
                {
                    bgBrush = new SolidBrush(Color.Black);
                }

                if ((m_ScreenMemory[i + 1] & 7) == 0)
                {
                    if ((m_ScreenMemory[i + 1] & 8) == 8)
                    {
                        fgBrush = new SolidBrush(Color.Gray);
                    }
                    else
                    {
                        fgBrush = new SolidBrush(Color.Black);
                    }
                }

                if ((m_ScreenMemory[i + 1] & 7) == 1)
                {
                    if ((m_ScreenMemory[i + 1] & 8) == 8)
                    {
                        fgBrush = new SolidBrush(Color.LightBlue);
                    }
                    else
                    {
```

```
                    fgBrush = new SolidBrush(Color.Blue);
                }
            }

            if ((m_ScreenMemory[i + 1] & 7) == 2)
            {
                if ((m_ScreenMemory[i + 1] & 8) == 8)
                {
                    fgBrush = new SolidBrush(Color.LightGreen);
                }
                else
                {
                    fgBrush = new SolidBrush(Color.Green);
                }
            }

            if ((m_ScreenMemory[i + 1] & 7) == 3)
            {
                if ((m_ScreenMemory[i + 1] & 8) == 8)
                {
                    fgBrush = new SolidBrush(Color.LightCyan);
                }
                else
                {
                    fgBrush = new SolidBrush(Color.Cyan);
                }
            }

            if ((m_ScreenMemory[i + 1] & 7) == 4)
            {
                if ((m_ScreenMemory[i + 1] & 8) == 8)
                {
                    fgBrush = new SolidBrush(Color.Pink);
                }
                else
                {
                    fgBrush = new SolidBrush(Color.Red);
                }
            }

            if ((m ScreenMemory[i + 1] & 7) == 5)
            {
                if ((m_ScreenMemory[i + 1] & 8) == 8)
                {
                    fgBrush = new SolidBrush(Color.Fuchsia);
                }
                else
                {
                    fgBrush = new SolidBrush(Color.Magenta);
                }
            }

            if ((m_ScreenMemory[i + 1] & 7) == 6)
            {
                if ((m_ScreenMemory[i + 1] & 8) == 8)
                {
                    fgBrush = new SolidBrush(Color.Yellow);
```

```
                    }
                    else
                    {
                        fgBrush = new SolidBrush(Color.Brown);
                    }
                }

                if ((m_ScreenMemory[i + 1] & 7) == 7)
                {
                    if ((m_ScreenMemory[i + 1] & 8) == 8)
                    {
                        fgBrush = new SolidBrush(Color.White);
                    }
                    else
                    {
                        fgBrush = new SolidBrush(Color.Gray);
                    }
                }
                if (bgBrush == null)
                    bgBrush = new SolidBrush(Color.Black);
                if (fgBrush == null)
                    fgBrush = new SolidBrush(Color.Gray);

                if (((xLoc % 640) == 0) && (xLoc != 0))
                {
                    yLoc += 11;
                    xLoc = 0;
                }
                string s =
System.Text.Encoding.ASCII.GetString(m_ScreenMemory, i, 1);
                PointF pf = new PointF(xLoc, yLoc);

                bmpGraphics.FillRectangle(bgBrush, xLoc+2, yLoc+2, 8f, 11f);
                bmpGraphics.DrawString(s, f, fgBrush, pf);
                xLoc += 8;
            }

            e.Graphics.DrawImage(bmp, new Point(0, 0));
            bmpGraphics.Dispose();
            bmp.Dispose();
        }
```

The code is quite long, but not really hard to figure out. The first thing I'm doing is creating a bitmap object to match the size of the control. In order to avoid flashing and/or flickering, I am creating the text on a bitmap, and then transferring the bitmap to the screen. This will avoid all flashing and flickering issues when a redraw() is needed. Next, I am creating a font for the text. I decided to use Courier New for two reasons. One, almost all computers have a Courier font installed and two; it is a *mono spaced* font. A mono spaced font is a font in which all the characters are the same width. In other words, if I type a sentence that's 20 characters long, then type another sentence below it that's also 20 characters long, the text will start and end at the same exact spot. This differs from a font you may use to type a letter for example. If you got Microsoft Word, open it up and type 2 sentences below each other and you will see the spacing is different. A capital 'O' may take up more space than a lower case 'l' for example.

The next thing I am doing is starting a loop. The 'if' statements that follow determine what foreground and background brush to use, based on the attribute. I am then using FillRectangle() to change the background behind the letter and DrawString() to draw the actually letter. Finally, once the loop completes and all the characters are drawn, the bitmap is then copied to the screen. That is all there is to our B32Screen user control. Now we can switch back to our MainForm and continue the work necessary to finish our virtual machine.

Now that we are done with our B32Screen control, lets test it out and see if it works as expected. Switch back to MainForm and add a B32Screen control to MainForm. Change the Dock property on the B32Screen control to "Fill", that way it will occupy the entire form. Switch to code view and add the following line to the MainForm constructor:

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
        b32Screen1.Poke(0xa000, 65);
    }
}
```

What we are doing is "poking" 65 ('A' in ASCII) to memory location $A000, which is the start of our screen memory. Go ahead and run the program and you should see an 'A' in the upper left hand corner in a light-gray color. If you do not, recheck the program for mistakes. If all works as expected, add the following 2 lines:

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
        b32Screen1.Poke(0xa000, 65);
        b32Screen1.Poke(0xa002, 66);
        b32Screen1.Poke(0xa004, 67);
    }
}
```

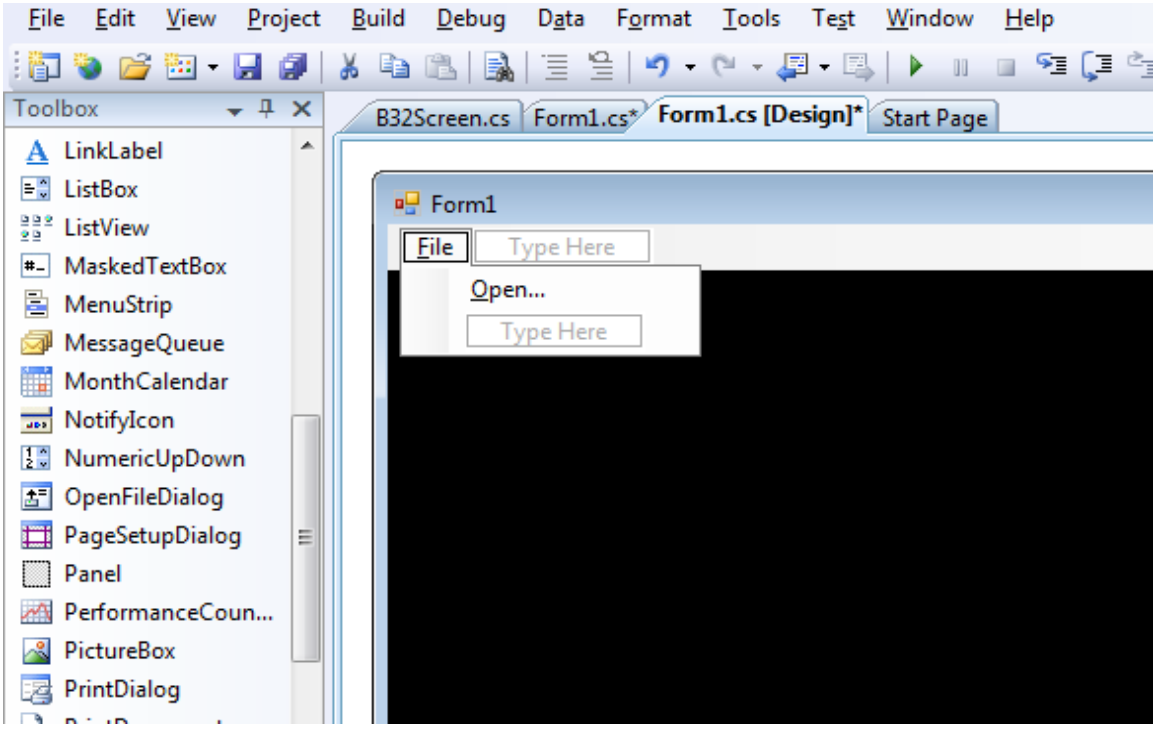
What I am doing is poking 66 ('B' in ASCII) and 67 ('C' in ASCII). When you run the program now, it should display 'ABC' in the upper left hand corner. Notice, in the program, that I am "poking" incrementally by 2. Remember that the odd bytes (the ones I'm not poking) control the attribute. Speaking of which, modify the program as follows:

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
        b32Screen1.Poke(0xa000, 65);
        b32Screen1.Poke(0xa002, 66);
        b32Screen1.Poke(0xa004, 67);
```

```
            b32Screen1.Poke(0xa001, Convert.ToByte("00011111", 2));
            b32Screen1.Poke(0xa003, Convert.ToByte("01001111", 2));
            b32Screen1.Poke(0xa005, Convert.ToByte("00101111", 2));
        }
    }
```

Run the program now and you'll see our text is now bright white, housed inside a blue, red and green background. Feel free to mess around and add some lines of your own. Write your own name, if you wish. If you are confused as to how the colors are determined from the given binary pattern, refer to the earlier discussion about attributes and colors and how they work.

Now that we tested our screen and we know it works, go ahead and remove all the lines we added in the constructor (do not remove the InitializeComponent() call), along with any lines you may have added. Switch back to design view. Add a menustrip control to MainForm. It should dock to the top. This menu strip will provide us with a way to open a B32 bytecode file. Change the name of the menu strip to "msMainMenu". For now, just add a "&File" menu header to the strip and under that, add "&Open…". It should like similar to the following:



Next, add a panel to the form. Change the dock to "Bottom" and change the name to "pnlRegisters". Make the height of the panel 54 pixels long. This panel will be a "diagnostic" panel that will monitor the state of our registers and display the values of them accordingly. This way, you can see that the B32 program we created is indeed executing as it should, without any tricks.

Drag an OpenFileDialog control onto the form. This dialog will popup when we click Open on our menu. This will allow the user to search for and select a B32 binary file to be executed. Change the

"DefaultExt" property to "B32" and the "Filter" property to "B32 Files | *.B32". Also, remove the text from the "Filename" property, so it's blank.

One final thing to add before we start coding. Drag a label control onto the panel. Make sure the label is a child control of the parent. Change the autosize property to "false" and the dock to "Fill" and change the textalign property to "MiddleLeft". Next, change the font to "Courier New", the size to 10 and set bold to "true". Finally, change the name to "lblRegisters".

Now it's time to start coding. Switch over to the code view of MainForm and lets add some member fields:

```
public partial class MainForm : Form
{
    private byte[] B32Memory;
    private ushort StartAddr;
    private ushort ExecAddr;
    private ushort InstructionPointer;
    private byte Register_A;
    private byte Register_B;
    private ushort Register_X;
    private ushort Register_Y;
    private ushort Register_D;
```

The first member field, `B32Memory`, will be our 64K memory, in the form of an array. We will initialize that later, in the constructor. The `StartAddr` will contain the starting address of our B32 binary. Likewise, the `ExecAddr` will be the execution address of our B32 binary. The InstructionPointer will contain an address of where, in our 64K memory, that our next bytecode to be executed is at. The remaining lines hold the values of our register.

Our constructor will initialize all our fields to default values:

```
    public MainForm()
    {
        InitializeComponent();

        B32Memory = new byte[65535];
        StartAddr = 0;
        ExecAddr = 0;
        Register_A = 0;
        Register_B = 0;
        Register_D = 0;
        Register_X = 0;
        Register_Y = 0;
        UpdateRegisterStatus();

    }
```

Our B32Memory is initialized with a 64K byte array. Everything else is set to 0. UpdateRegisterStatus() is a function we will code in a minute that will update the register label in our panel to display the appropriate contents of our registers. Here is that function now:

```
        private void UpdateRegisterStatus()
        {
            string strRegisters = "";

            strRegisters = "Register A = $" +
Register_A.ToString("X").PadLeft(2, '0');
            strRegisters += "     Register B = $" +
Register_B.ToString("X").PadLeft(2, '0');
            strRegisters += "     Register D = $" +
Register_D.ToString("X").PadLeft(4, '0');
            strRegisters += "\nRegister X = $" +
Register_X.ToString("X").PadLeft(4, '0');
            strRegisters += "   Register Y = $" +
Register_Y.ToString("X").PadLeft(4, '0');
            strRegisters += "   Instruction Pointer = $" +
InstructionPointer.ToString("X").PadLeft(4, '0');

            this.lblRegisters.Text = strRegisters;
        }
```

When this function is called, it takes the current values of the registers, converts them into hexadecimal, then it sticks the text into our label. Should be pretty simple to follow this code. The PadLeft() functions make the values consistent length, so that "$01" is displayed instead of "$1".

Switch back over to design view and create an event handler for our File → Open menu item. Type the following code into the event handler, then we will analyze it more in depth:

```
        private void openToolStripMenuItem_Click(object sender, EventArgs e)
        {
            byte Magic1;
            byte Magic2;
            byte Magic3;

            openFileDialog1.ShowDialog();

            System.IO.BinaryReader br;
            System.IO.FileStream fs = new
System.IO.FileStream(openFileDialog1.FileName, System.IO.FileMode.Open);

            br = new System.IO.BinaryReader(fs);

            Magic1 = br.ReadByte();
            Magic2 = br.ReadByte();
            Magic3 = br.ReadByte();

            if (Magic1 != 'B' && Magic2 != '3' && Magic3 != '2')
            {
                MessageBox.Show("This is not a valid B32 file!", "Error!",
MessageBoxButtons.OK, MessageBoxIcon.Error);
                return;
            }

            StartAddr = br.ReadUInt16();
            ExecAddr = br.ReadUInt16();
            ushort Counter = 0;
            while ((br.PeekChar() != -1))
```

```
            {
                B32Memory[(StartAddr + Counter)] = br.ReadByte();
                Counter++;
            }

            br.Close();
            fs.Close();

            InstructionPointer = ExecAddr;

            ExecuteProgram(ExecAddr, Counter);
        }
```

The first thing I'm doing is declaring 3 variables to hold our magic header numbers. Remember earlier, we said that all B32 binaries will have a "B32" as the magic header? I am then displaying our open file dialog so that the user can select the B32 file. We then use a BinaryReader stream to open the file. We then read in the file and check to make sure it has our "B32" header. If it doesn't we display a message box informing the user that this is not a valid file. We then read in the starting address and the execution address, which is part of our B32 header. Next, we read in the bytecode and store it, beginning at our start address. Finally, we close the stream, point our instruction pointer to the execution address, then we call a function to execute our program.

Our ExecuteProgram() function does most of the "real" work. Here it is:

```
        private void ExecuteProgram(ushort ExecAddr, ushort ProgLength)
        {
            ProgLength = 64000;
            while (ProgLength > 0)
            {

                byte Instruction = B32Memory[InstructionPointer];
                ProgLength--;
                if (Instruction == 0x02) // LDX #<value>
                {
                    Register_X = (ushort)((B32Memory[(InstructionPointer +
2)]) << 8);
                    Register_X += B32Memory[(InstructionPointer + 1)];
                    ProgLength -= 2;
                    InstructionPointer += 3;

                    UpdateRegisterStatus();

                    continue;
                }
                if (Instruction == 0x01) // LDA #<value>
                {
                    Register_A = B32Memory[(InstructionPointer + 1)];
                    SetRegisterD();
                    ProgLength -= 1;
                    InstructionPointer += 2;

                    UpdateRegisterStatus();

                    continue;
```

```
                }
                if (Instruction == 0x03) // STA ,X
                {
                    B32Memory[Register_X] = Register_A;
                    b32Screen1.Poke(Register_X, Register_A);
                    InstructionPointer++;

                    UpdateRegisterStatus();

                    continue;
                }
                if (Instruction == 0x04) // END
                {
                    InstructionPointer++;
                    UpdateRegisterStatus();
                    break;
                }
            }
        }
```

The first think we do is read in an instruction bytecode. Then, using a series of "if" statements, we "execute" that bytecode. After each bytecode is interpreted, we make a call to UpdateRegisterStatus() to update our register label panel. We need only one last function to be able to execute and try out our new virtual machine. Remember that whenever a value in the 'A' register or the 'B' register is modified, the 'D' register needs to also automatically update, and same thing if a value is modified in the 'D' register; we need to update the 'A' and 'B' registers accordingly. Here is the SetRegisterD() function that updates the 'D' register:

```
        private void SetRegisterD()
        {
            Register_D = (ushort)(Register_A << 8 + Register_B);
        }
```

This should be a pretty simple function to wrap your head around. I am pulling the value of the A register, shifting it 8 bits to the left to get it into the upper significant bits, then I am adding the value of the B register. Finally, I am casting the result to an unsigned short (16-bit value).

Go ahead and run our program now. Click on the File menu and go to open. Open the B32 test file we assembled earlier. You should see an 'A' appear in the upper left hand corner of the B32 screen, along with the status of the registers at the bottom of the window! Congratulations! We have just created a working virtual machine. Albeit, it is a pretty simple and limited virtual machine, but it does work! And just to prove it, we are going to expand our test program from earlier. Exit the virtual machine and open your test.asm file we made earlier. Change the file so it looks like the following:

```
Start:
 LDA #65
 LDX #$A000
 STA ,X
 LDA #66
 LDX #$A002
 STA ,X
```

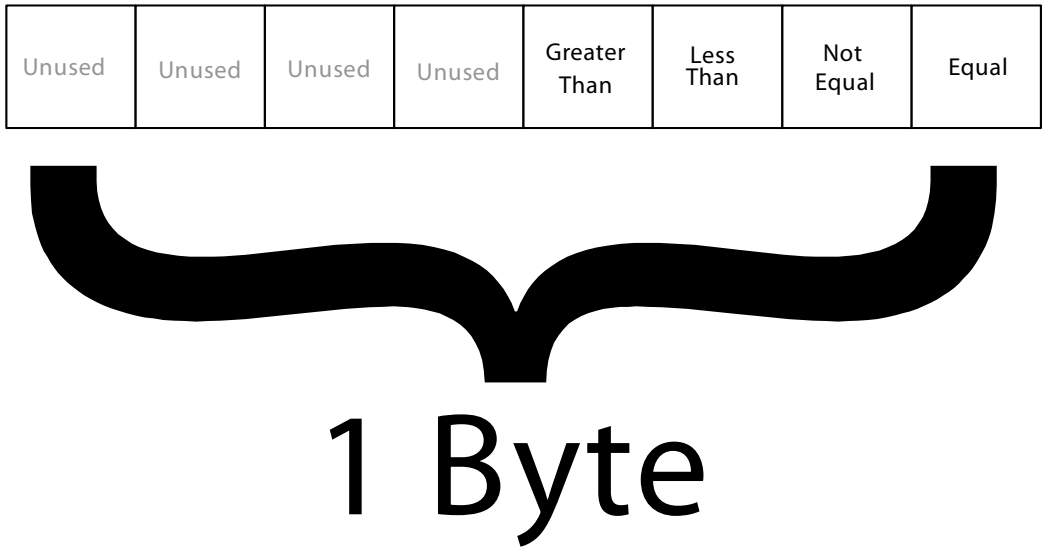
```
LDA #67
LDX #$A004
STA ,X
END Start
```

When typing this, don't forget to add at least one space before each mnemonic. Also, don't forget to end the last line with a carriage return.

Open the B32 assembler and assemble "Test.ASM", the new program you just created. Set the output to "Test.B32", then run the B32 virtual machine again, this time selecting the Test.B32 file we just assembler. This time, it should display the letters ABC in the upper left hand corner. How cool is this? Technically, you could write a B32 assembly file to do whatever you want, assuming you just use the 4 mnemonics that we have programmed so far. This is pretty limiting though, and other then some text being written to the screen, you couldn't create much of an application. We are going to spend the remainder of this tutorial improving upon the virtual machine and assembler that we created here. For now, if you'd like, feel free to create some test ASM files and trying them out. Try to write your name, centered, on the screen. Make it appear with a blue background and bright white text (hint: revisit the attributes section we discussed earlier).

Revisiting the Drawing Board

Before we jump back into coding, it's time to do some more planning. Yes, that's right, more dry talk. But I promise to keep it brief and to get back into the code side soon. We are going to add several more mnemonics to our B32 language to make it more useful. The first set of mnemonics I'd like to add are comparator mnemonics. To keep it simple, comparator mnemonics are instructions that change comparator flags (we will talk about these in a second) based on the evaluation of a register and some value. They are B32's way of implementing an "if" statement, like C# has. Comparator flags are individual bits of a byte that get set or reset depending on the evaluation of the register. The following are the comparator flags we are going to use for B32:



Whenever we perform a "compare" operation, internally, each bit, or flag, gets set. Typically, either the last bit will get set, indicating the register and the value compared against are equal, or else the "not equal" flag will get set, along with one other flag, indicating if the register value was greater than or less than the value compared against. Once we can do some sort comparison in B32, we can then cause program execution to jump to another spot based on that comparison. Once we implement this, we can start to do some more interesting stuff.

With all that said, our plan will be to implement the following mnemonics:

Mnemonic	Description	Example	What will this example do?
CMPA $05	Compares the value of the 'A' register	CMPA #$20	Compares the value of the 'A' register with $20 and sets our internal "compare registers" appropriately
CMPB $06	Compares the value of the 'B' register	CMPB #$20	Compares the value of the 'B' register with $20 and sets our internal "compare registers" appropriately
CMPX $07	Compares the value of the 'X' register	CMPX #$A057	Compares the value of the 'X' register with $A057 and sets our internal "compare registers" appropriately
CMPY $08	Compares the value of the 'Y' register	CMPY #$A057	Compares the value of the 'Y' register with $A057 and sets our internal "compare registers" appropriately
CMPD $09	Compares the value of the 'D' register	CMPD #$A057	Compares the value of the 'D' register with $A057 and sets our internal "compare registers" appropriately

These mnemonics do us no good though if we have no way to act upon the result of the compare. As a result, we are also going to implement these mnemonics:

Mnemonic	Description	Example	What will this example do?
JMP $0A	Jumps to a specific location in memory and resumes execution	JMP #$3000	Jumps to memory location $3000 and resumes execution at this location
JEQ $0B	Jumps to a specific location in memory ONLY if the result of the last compare was equal	CMPA #$6A JEQ #$3000	Compares the value of the 'A' register to $6A and if it's equal, the program jumps to memory location #$3000 and resumes execution
JNE $0C	Jumps to a specific location in memory ONLY if the result of the last compare was NOT equal	CMPA #$6A JNE #$3000	Compares the value of the 'A' register to $6A and if it's NOT equal, the program jumps to memory location #$3000 and resumes execution
JGT $0D	Jumps to a specific location in memory ONLY if the result of the last compare was greater than the value	CMPA #$6A JGT #$3000	Compares the value of the 'A' register to $6A and if 'A' is greater than $6A, the program jumps to memory location #$3000 and resumes execution
JLT $0E	Jumps to a specific location in memory ONLY if the result of the last compare was less than the value	CMPA #$6A JLT #$3000	Compares the value of the 'A' register to $6A and if 'A' is less than $6A, the program jumps to memory location #$3000 and resumes execution

This may seem like it's going to be a lot of work to implement all these mnemonics, but you'll soon see that adding new mnemonics to our assembler and virtual machine is very easy to do.

Now that we have defined our goals and what we want to do, it's time to make a test assembly file that will test our new mnemonics. Open "Test.asm" in notepad again and delete all the lines and type in the following for our test program (remember to put a space before each mnemonic and a carriage return after the last line):

```
Start:
 JMP #Spot1
 LDA #65
 LDX #$A000
 STA ,X
 JMP #EndSpot
Spot1:
 LDA #72
 LDX #$A002
 STA ,X
 CMPA #72
 JEQ #Spot2
```

```
 JMP #EndSpot
Spot2:
 LDA #73
 LDX #$A004
 STA ,X
 CMPA #99
 JNE #Spot3
 JMP #EndSpot
Spot3:
 LDA #74
 LDX #$A006
 STA ,X
 CMPA #107
 JLT #Spot4
 JMP #EndSpot
Spot4:
 LDA #75
 LDX #$A008
 STA ,X
 CMPA #12
 JGT #Spot5
 JMP #EndSpot
Spot5:
 LDA #76
 LDX #$A00A
 STA ,X
 CMPA #92
 JEQ #Spot6
 JMP #EndSpot
Spot6:
 LDA #77
 LDX #$A00C
 STA ,X
EndSpot:
 END Start
```

At first, this may seem a bit intimidating. But try to follow the logic. Our program execution will start at the "Start" label. The first mnemonic after that is a "JMP". In our jump mnemonic examples, we used an actual address, but you will almost always want to use a label instead. Labels make it easier; for example, it's easier to say "JMP #Spot5" then it is to try to figure out what address "Spot5" is. First, we jump to Spot1. Then we write an 'H' as the second character in the upper left hand corner of the screen. Then we do our first compare. We compare 'A' register to 72. Since the value in the 'A' register does indeed equal 72, the next line (the 'JEQ' mnemonic) will jump to Spot2. These same kind of comparisons continue on in the program, to "Spot2", to "Spot3", to "Spot4", and finally, to "Spot5". At "Spot5", we do another compare to see if 'A' Register is equal to 92. Since it won't be, the program will NOT jump to "Spot6" and instead will continue on, hitting the next "JMP #EndSpot", which will cause the program to jump to "Endspot". The program finally terminates at our 'END' mnemonic. Notice there are some parts of the code that will never even get executed. Normally, you wouldn't write a program like this, but I am doing it this time to demonstrate how our new mnemonics will work.

Okay, go ahead and save the file as "Test.ASM". We are now ready to make the modifications to the assembler that are necessary to assemble our file.

Revisiting our Assembler

Open Visual Studio and reload our assembler solution. The first thing we need to do is modify one of our functions that we created earlier. Switch over to the code side and scroll down to the ReadWordValue() function. The problem with this function right now is, it doesn't know how to read the value of a label. The function, as it stands, simply reads in a hexadecimal or decimal number. Our jumps, however, will almost always be a label value, so we need to modify this function to equate a label with a value. Doing this is pretty simple. Add the following highlighted lines:

```
        private ushort ReadWordValue()
        {
            ushort val = 0;
            bool IsHex = false;
            string sval = "";

            if (SourceProgram[CurrentNdx] == '$')
            {
                CurrentNdx++;
                IsHex = true;
            }

            if ((IsHex == false) &&
(char.IsLetter(SourceProgram[CurrentNdx])))
            {
                val = (ushort)LabelTable[GetLabelName()];
                return val;
            }

            while (char.IsLetterOrDigit(SourceProgram[CurrentNdx]))
            {
                sval = sval + SourceProgram[CurrentNdx];
                CurrentNdx++;
            }
            if (IsHex)
            {
                val = Convert.ToUInt16(sval, 16);
            }
            else
            {
                val = ushort.Parse(sval);
            }

            return val;
        }
```

Pretty simple change. We test to see if the value is going to be hexadecimal. If it's not and the current character is not a number, we can safely assume it's a label. We then access our `LabelTable` hash table to retrieve the value and cast it to ushort.

The next function we need to change is the ReadMneumonic() function. Find that function, then add the following lines:

```
        private void ReadMneumonic(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            string Mneumonic = "";

            while (!(char.IsWhiteSpace(SourceProgram[CurrentNdx])))
            {
                Mneumonic = Mneumonic + SourceProgram[CurrentNdx];
                CurrentNdx++;
            }
            if (Mneumonic.ToUpper() == "LDX")  InterpretLDX(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "LDA")  InterpretLDA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "STA")  InterpretSTA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "CMPA") InterpretCMPA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "CMPB") InterpretCMPB(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "CMPX") InterpretCMPX(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "CMPY") InterpretCMPY(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "CMPD") InterpretCMPD(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "JMP")  InterpretJMP(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "JEQ") InterpretJEQ(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "JNE") InterpretJNE(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "JGT") InterpretJGT(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "JLT") InterpretJLT(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "END") { IsEnd = true;
DoEnd(OutputFile, IsLabelScan); EatWhiteSpaces(); ExecutionAddress =
(ushort)LabelTable[(GetLabelName())]; return; }

            while (SourceProgram[CurrentNdx] != '\n')
            {
                CurrentNdx++;
            }
            CurrentNdx++;
        }
```

This should be pretty self explanatory. We are simply adding more “if” statements to test for the presence of our jump and compare mnemonics. This works no differently then when we added LDA, STA, END or LDX.

Now all we need to do is add the appropriate functions to do the work. First, here are the compare functions:

```
        private void InterpretCMPA(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                byte val = ReadByteValue();
                AsLength += 2;
                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x05);
                    OutputFile.Write(val);
                }
            }
        }

        private void InterpretCMPB(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                byte val = ReadByteValue();
                AsLength += 2;
                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x06);
                    OutputFile.Write(val);
                }
            }
        }

        private void InterpretCMPX(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                ushort val = ReadWordValue();
                AsLength += 3;
                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x07);
                    OutputFile.Write(val);
                }
            }
        }

        private void InterpretCMPY(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
```

```
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                ushort val = ReadWordValue();
                AsLength += 3;
                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x08);
                    OutputFile.Write(val);
                }
            }
        }

        private void InterpretCMPD(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                ushort val = ReadWordValue();
                AsLength += 3;
                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x09);
                    OutputFile.Write(val);
                }
            }
        }
```

Each of these functions is almost identical. The only difference is the bytecode value that gets written. There should be no mystery to understanding how each of these functions work. Now, here are the jump functions:

```
        private void InterpretJMP(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                AsLength += 3;
                if (IsLabelScan) return;
                ushort val = ReadWordValue();

                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x0A);
                    OutputFile.Write(val);
                }
            }
        }

        private void InterpretJEQ(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
```

```
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                AsLength += 3;
                if (IsLabelScan) return;
                ushort val = ReadWordValue();

                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x0B);
                    OutputFile.Write(val);
                }
            }
        }

        private void InterpretJNE(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                AsLength += 3;
                if (IsLabelScan) return;
                ushort val = ReadWordValue();

                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x0C);
                    OutputFile.Write(val);
                }
            }
        }

        private void InterpretJGT(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                AsLength += 3;
                if (IsLabelScan) return;
                ushort val = ReadWordValue();

                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x0D);
                    OutputFile.Write(val);
                }
            }
        }

        private void InterpretJLT(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
```

```
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                AsLength += 3;
                if (IsLabelScan) return;
                ushort val = ReadWordValue();

                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x0E);
                    OutputFile.Write(val);
                }
            }
        }
```

These are almost identical to our compare statements. The biggest difference is, notice that, if we are on Pass 1, doing the label scan, we return early. The reason is, because the any forward labels we use will not yet be defined, causing an error.

That's it! See how simple it is to expand our assembler? We just added 10 brand new mnemonics and it took all of 15 minutes to do it! Go ahead and run the assembler and try to assemble the "Test.ASM" file you created earlier. It should we assemble just fine, producing an 82 byte B32 bytecode file. Now it's time to expand our virtual machine so that it will be able to interpret the new bytecodes.

Revisiting our Virtual Machine

The first part of our bytecode that we will be implementing will be the compare codes. In order to do this, we need to first setup an internal private member to act as our compare flag. Fire up Visual Studio and open the B32Machine solution. Add the following two lines:

```
public partial class MainForm : Form
{
    private byte[] B32Memory;
    private ushort StartAddr;
    private ushort ExecAddr;
    private ushort InstructionPointer;
    private byte Register_A;
    private byte Register_B;
    private ushort Register_X;
    private ushort Register_Y;
    private ushort Register_D;
    private byte CompareFlag;

    public MainForm()
    {
        InitializeComponent();
```

```
            CompareFlag = 0;
            B32Memory = new byte[65535];
            StartAddr = 0;
            ExecAddr = 0;
```

We define a byte to act as our compare flag and then we give it an initial value of 0 in the constructor.

The next change we need to make is to add lines to our ExecuteProgram() function. Scroll and find this function and add the following new lines:

```
                if (Instruction == 0x04) // END
                {
                    InstructionPointer++;
                    UpdateRegisterStatus();
                    break;
                }
                if (Instruction == 0x05) // CMPA
                {
                    byte CompValue=B32Memory[InstructionPointer+1];

                    CompareFlag = 0;

                    if (Register_A == CompValue) CompareFlag =
(byte)(CompareFlag | 1);
                    if (Register_A != CompValue) CompareFlag =
(byte)(CompareFlag | 2);
                    if (Register_A < CompValue)  CompareFlag =
(byte)(CompareFlag | 4);
                    if (Register_A > CompValue)  CompareFlag =
(byte)(CompareFlag | 8);

                    InstructionPointer += 2;

                    UpdateRegisterStatus();

                    continue;
                }
                if (Instruction == 0x06) // CMPB
                {
                    byte CompValue = B32Memory[InstructionPointer + 1];

                    CompareFlag = 0;

                    if (Register_B == CompValue) CompareFlag =
(byte)(CompareFlag | 1);
                    if (Register_B != CompValue) CompareFlag =
(byte)(CompareFlag | 2);
                    if (Register_B < CompValue) CompareFlag =
(byte)(CompareFlag | 4);
                    if (Register_B > CompValue) CompareFlag =
(byte)(CompareFlag | 8);

                    InstructionPointer += 2;
```

```
                    UpdateRegisterStatus();

                    continue;
                }
                if (Instruction == 0x07) //CMPX
                {
                    ushort CompValue =
(ushort)((B32Memory[(InstructionPointer + 2)]) << 8);
                    CompValue += B32Memory[(InstructionPointer + 1)];

                    CompareFlag = 0;

                    if (Register_X == CompValue) CompareFlag =
(byte)(CompareFlag | 1);
                    if (Register_X != CompValue) CompareFlag =
(byte)(CompareFlag | 2);
                    if (Register_X < CompValue) CompareFlag =
(byte)(CompareFlag | 4);
                    if (Register_X > CompValue) CompareFlag =
(byte)(CompareFlag | 8);

                    InstructionPointer += 3;

                    UpdateRegisterStatus();

                    continue;
                }
                if (Instruction == 0x08) //CMPY
                {
                    ushort CompValue =
(ushort)((B32Memory[(InstructionPointer + 2)]) << 8);
                    CompValue += B32Memory[(InstructionPointer + 1)];

                    CompareFlag = 0;

                    if (Register_Y == CompValue) CompareFlag =
(byte)(CompareFlag | 1);
                    if (Register_Y != CompValue) CompareFlag =
(byte)(CompareFlag | 2);
                    if (Register_Y < CompValue) CompareFlag =
(byte)(CompareFlag | 4);
                    if (Register_Y > CompValue) CompareFlag =
(byte)(CompareFlag | 8);

                    InstructionPointer += 3;

                    UpdateRegisterStatus();

                    continue;
                }
                if (Instruction == 0x09) //CMPD
                {
                    ushort CompValue =
(ushort)((B32Memory[(InstructionPointer + 2)]) << 8);
                    CompValue += B32Memory[(InstructionPointer + 1)];

                    CompareFlag = 0;
```

```
                    if (Register_D == CompValue) CompareFlag =
(byte)(CompareFlag | 1);
                    if (Register_D != CompValue) CompareFlag =
(byte)(CompareFlag | 2);
                    if (Register_D < CompValue) CompareFlag =
(byte)(CompareFlag | 4);
                    if (Register_D > CompValue) CompareFlag =
(byte)(CompareFlag | 8);

                    InstructionPointer += 3;

                    UpdateRegisterStatus();

                    continue;
                }
            }
        }
```

Most of these functions work identical to each other. The code should be pretty simple to follow. We are setting the compare flag to zero each time a comparison is made, and then toggling the appropriate bit depending on weather the comparison is equal, not equal, greater than, or less than.

Well if you thought that was easy, you're going to love the jump functions. Those are even easier! Add the following lines to the ExecuteProgram() function:

```
                if (Instruction == 0x09) //CMPD
                {
                    ushort CompValue =
(ushort)((B32Memory[(InstructionPointer + 2)]) << 8);
                    CompValue += B32Memory[(InstructionPointer + 1)];

                    CompareFlag = 0;

                    if (Register_D == CompValue) CompareFlag =
(byte)(CompareFlag | 1);
                    if (Register_D != CompValue) CompareFlag =
(byte)(CompareFlag | 2);
                    if (Register_D < CompValue) CompareFlag =
(byte)(CompareFlag | 4);
                    if (Register_D > CompValue) CompareFlag =
(byte)(CompareFlag | 8);

                    InstructionPointer += 3;

                    UpdateRegisterStatus();

                    continue;
                }
                if (Instruction == 0x0A) // JMP
                {
                    ushort JmpValue = (ushort)((B32Memory[(InstructionPointer
+ 2)]) << 8);
                    JmpValue += B32Memory[(InstructionPointer + 1)];

                    InstructionPointer = JmpValue;
```

```
                    UpdateRegisterStatus();

                    continue;
                }
                if (Instruction == 0x0B) // JEQ
                {
                    ushort JmpValue = (ushort)((B32Memory[(InstructionPointer
+ 2)]) << 8);
                    JmpValue += B32Memory[(InstructionPointer + 1)];

                    if ((CompareFlag & 1) == 1)
                    {
                        InstructionPointer = JmpValue;
                    }
                    else
                    {
                        InstructionPointer += 3;
                    }
                    UpdateRegisterStatus();

                    continue;
                }
                if (Instruction == 0x0C) // JNE
                {
                    ushort JmpValue = (ushort)((B32Memory[(InstructionPointer
+ 2)]) << 8);
                    JmpValue += B32Memory[(InstructionPointer + 1)];

                    if ((CompareFlag & 2) == 2)
                    {
                        InstructionPointer = JmpValue;
                    }
                    else
                    {
                        InstructionPointer += 3;
                    }
                    UpdateRegisterStatus();

                    continue;
                }
                if (Instruction == 0x0D) // JGT
                {
                    ushort JmpValue = (ushort)((B32Memory[(InstructionPointer
+ 2)]) << 8);
                    JmpValue += B32Memory[(InstructionPointer + 1)];

                    if ((CompareFlag & 4) == 4)
                    {
                        InstructionPointer = JmpValue;
                    }
                    else
                    {
                        InstructionPointer += 3;
                    }
                    UpdateRegisterStatus();
```

```
                    continue;
                }
                if (Instruction == 0x0E) // JLT
                {
                    ushort JmpValue = (ushort)((B32Memory[(InstructionPointer
+ 2)]) << 8);
                    JmpValue += B32Memory[(InstructionPointer + 1)];

                    if ((CompareFlag & 8) == 8)
                    {
                        InstructionPointer = JmpValue;
                    }
                    else
                    {
                        InstructionPointer += 3;
                    }
                    UpdateRegisterStatus();

                    continue;
                }
            }
        }
```

Again, most of these functions work almost the save, jumping to an appropriate place in memory depending on the bit state of our compare flags.

One last thing I want to do before we try out our new virtual machine. Scroll and find the function we created earlier called UpdateRegisterStatus(). Add the following line:

```
            strRegisters += "\nRegister X = $" +
Register_X.ToString("X").PadLeft(4, '0');
            strRegisters += "   Register Y = $" +
Register_Y.ToString("X").PadLeft(4, '0');
            strRegisters += "   Instruction Pointer = $" +
InstructionPointer.ToString("X").PadLeft(4, '0');
            strRegisters += "\nCompare Flag = $" +
CompareFlag.ToString("X").PadLeft(2, '0');

            this.lblRegisters.Text = strRegisters;
```

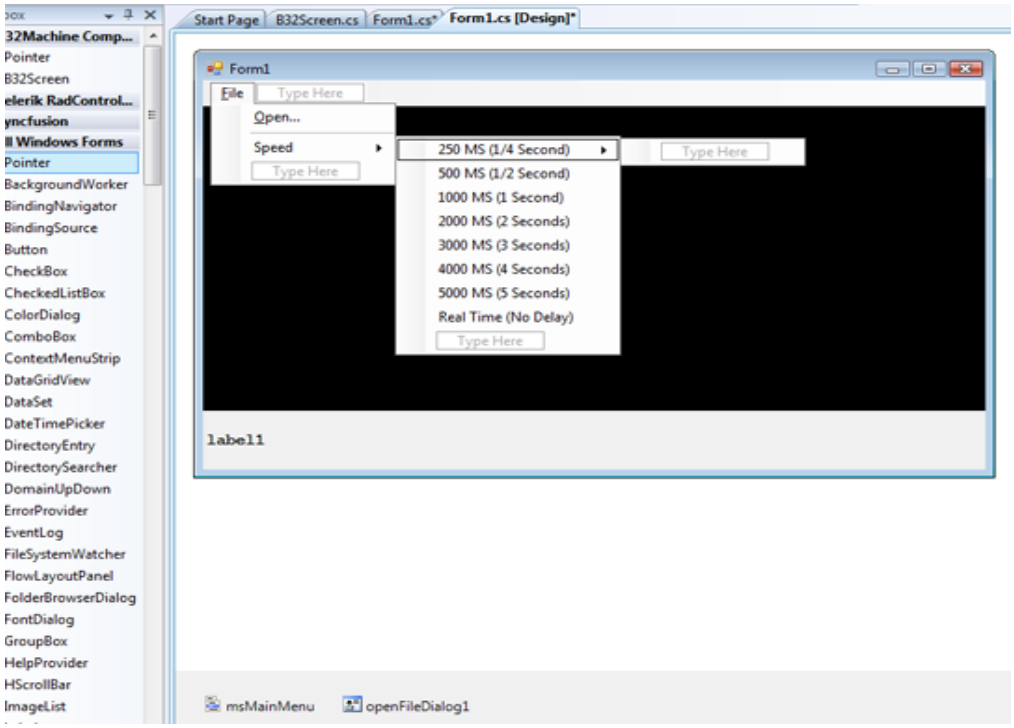
This will add a new line to our register status bar and show us the value of the compare flag. Go ahead and run the virtual machine and then run out B32 bytecode binary. In the upper left hand corner of the screen it should read “ HIJKL”. Look again at the “Test.asm” file and you should be able to figure out what is going on. Notice that “Spot6” is never hit. If “Spot6” had been hit, your display would say “ HIJKLM” instead of “ HIJKL”. If you were to change that last “JEQ #Spot6” to “JNE #Spot6”, re-assembled and re-ran our program, then you would get “ HIJKLM” as the output. Hopefully everything up to this point makes sense. Do not continue on with this tutorial unless you have a firm understanding of everything we learned up to this point. In the next section, we are going to implement some timing functionality to our virtual machine.

Timing is Everything!

Some virtual machines, especially ones that emulate an older machine from the 80's or early 90's, try to target a fixed clock cycle. Most modern day computers run at 2 GHz or 3 GHz or even 4 GHz, while older machines only 2 MHz (that's MHz or Megahertz). If we wanted to emulate a 6502 microprocessor, for example, the way we created our virtual machine now, it would execute 6502 instruction codes WAY too fast. The way to properly introduce precision timing in our virtual machine would be to find a hardware guide to the 6502 processor, find out how many clock cycles an instruction will take to execute (which will probably be different for each instruction), then introduce some kind of delay in between instruction execution commands.

For our virtual machine in this tutorial, we are not trying to target a specific clock cycle timing. I am, however, demonstrating timing here because I think its important, as a learning tool, to see the register values and flags change with each instruction. Our virtual machine does have a register bar that we created; however, our program executes way too fast to see each and every change occur per instruction. So in this section, we are going to improve our virtual machine so that we can adjust the speed at which our virtual machine executes instructions. It's worth mentioning that the method we will use here to delay is a static function found inside the System.Threading.Thread class. This function is called Sleep() and is used to pause our program for XXX number of milliseconds. The Sleep() function is defined as guaranteeing that our process will Sleep() for AT LEAST XXX number of milliseconds, however it is considered inaccurate because it will not guarantee that our process will Sleep() for PRECISLY XXX number of seconds. It is also not a good choice because sometimes you'll want to delay for less than a millisecond, which Sleep() does not support. Most "real" emulators will use some kind of high precision and accurate timer, such as the timers found in DirectX. Since we are simply trying to learn about virtual machines and not trying to actually make a full blown commercial product, Sleep() will suffice for us in this tutorial.

To begin, open the B32Machine project inside Visual Studio, if you don't already have it open. Switch over to designer view and we are going to add some more menu items. Add a menu item under Open called "Speed", and then under "Speed", we are going to add 8 different speeds. We are going to add a 1 second, 2 second, 3 second, 4 second, 5 second, ½ second, ¼ second, and finally a "Real Time" options, which will be a speed with no delay. Your menu should look like the following:



If you have named the menu items exactly the same way as I named them, then the IDs should be the same as used in the code below. Create a click handler for each of the 8 menu items you created and place the following code in the appropriate handler:

```
        private void mS14SecondToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            UncheckAll();
            ((ToolStripMenuItem)sender).Checked = true;
            SpeedMS = 250;
        }

        private void mS12SecondToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            UncheckAll();
            ((ToolStripMenuItem)sender).Checked = true;
            SpeedMS = 500;
        }

        private void mS1SecondToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            UncheckAll();
            ((ToolStripMenuItem)sender).Checked = true;
            SpeedMS = 1000;
        }

        private void mS2SecondsToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            UncheckAll();
```

```
            ((ToolStripMenuItem)sender).Checked = true;
            SpeedMS = 2000;
        }

        private void mS3SecondsToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            UncheckAll();
            ((ToolStripMenuItem)sender).Checked = true;
            SpeedMS = 3000;
        }

        private void mS4SecondsToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            UncheckAll();
            ((ToolStripMenuItem)sender).Checked = true;
            SpeedMS = 4000;
        }

        private void mS5SecondsToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            UncheckAll();
            ((ToolStripMenuItem)sender).Checked = true;
            SpeedMS = 5000;
        }

        private void realTimeNoDelayToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            UncheckAll();
            ((ToolStripMenuItem)sender).Checked = true;
            SpeedMS = 0;
        }
```

Basically, each handler does almost the same thing. First, we uncheck all the menu items. Then we check the menu item that was selected. Finally, we set the variable `SpeedMS` (which we did not create yet) to the number of milliseconds we want to delay.

Go ahead and the SpeedMS field to the beginning of the class and initialize it in the constructor:

```
    public partial class MainForm : Form
    {
        private byte[] B32Memory;
        private ushort StartAddr;
        private ushort ExecAddr;
        private ushort InstructionPointer;
        private byte Register_A;
        private byte Register_B;
        private ushort Register_X;
        private ushort Register_Y;
        private ushort Register_D;
        private byte CompareFlag;
        private ushort SpeedMS;
```

```
        public MainForm()
        {
            InitializeComponent();

            CompareFlag = 0;
            SpeedMS = 0;
            realTimeNoDelayToolStripMenuItem.Checked = true;
            B32Memory = new byte[65535];
            StartAddr = 0;
            ExecAddr = 0;
```

We are simply defaulting it to 0 milliseconds (which is real time) and we are defaulting a check mark in the RealTime menu strip, so that when we first run the program, this will be the default menu option selected.

One more function we need to add before we can test the user interface:

```
        private void UncheckAll()
        {
            mS12SecondToolStripMenuItem.Checked = false;       // 1/2 second
            mS14SecondToolStripMenuItem.Checked = false;       // 1/4 second
            mS1SecondToolStripMenuItem.Checked = false;        // 1 second
            mS2SecondsToolStripMenuItem.Checked = false;       // 2 seconds
            mS3SecondsToolStripMenuItem.Checked = false;       // 3 seconds
            mS4SecondsToolStripMenuItem.Checked = false;       // 4 seconds
            mS5SecondsToolStripMenuItem.Checked = false;       // 5 seconds
            realTimeNoDelayToolStripMenuItem.Checked = false; // real time
        }
```

This function un-checks all the menu items under the Speed menu. Go ahead and run the program now and test the menu items out. Whenever you select a speed under the "Speed" menu item, it should place a checkmark next to it and uncheck the previous one that was checked.

Now we will add the functionality needed to actually make the speed options work. Believe it or not, this is as simple as adding just 1 line of code. Find the ExecuteProgram() function and the following line of code:

```
        private void ExecuteProgram(ushort ExecAddr, ushort ProgLength)
        {
            ProgLength = 64000;
            while (ProgLength > 0)
            {
                byte Instruction = B32Memory[InstructionPointer];
                ProgLength--;
                System.Threading.Thread.Sleep(SpeedMS);

                if (Instruction == 0x02) // LDX #<value>
```

As discussed earlier, this line makes a call to the "System.Threading.Thread.Sleep" static function. Go ahead and run the program. Set the speed to 250 MS (1/4 of a second) and then open our B32 binary test program that we used earlier. Notice how " HIJKLM" are slowly placed on the screen?

There are probably a couple of questions you are asking yourself right now. First of all, how come it seems like the letters are placed on the screen every second or so, rather than every 250ms? The answer to that question is simple. Remember that the delay is added prior to executing any BYTECODE, not just before writing to our B32 screen. Remember that one an "LDA" or an "STA" or an "LDX" or a "JMP", or any other bytecode is encountered, that's 250ms the processor delays for. Since our example does an "LDX", an "LDA", a "STA", that's 750ms right there, plus in our example, most of this is proceeded with some kind of jump statement, which adds another 250ms – 1 second!

The second thing you might have noticed is, as our program runs, our register bar does not change. In addition, our user interface "locks up". The reason for this is because our virtual machine is running on a single thread. And that single thread is running our program, thus it cannot update our register bar or respond to user interface requests. To fix this problem, we need to make our virtual machine multithreaded. We will do add that improvement to our virtual machine in the next section coming up.

Making our Virtual Machine Multithreaded

Multithreading is a very complex topic and entire books have been devoted to the topic. Multithreading, in a nutshell, is creating and executing 1 or more threads. A thread is a portion of code that is executing at the same time other threads are executing. What we are going to do is have our virtual machine execute our B32 programs in a separate thread. By doing this, you'll see realtime updates on our register bar and our user interface will be responsive, allowing us to add a "Start", "Stop" and "Restart" option to our user interface.

Adding multithreading support to our application really isn't difficult. The tricky part comes when having to update the user interface. Because several threads running concurrently could inadvertently access the same resources (such as the keyboard, the mouse, the screen, the printer, etc etc), it is generally not an accepted (nor is it allowed) practice to directly update the screen from a thread. So what we are going to do instead is create a delegate function that will update the screen for us. We will call this delegate from our thread.

With that said, the first thing we want to do is add the following 2 lines just after our field definitions, at the top of our class:

```
private ushort Register_D;
        private byte CompareFlag;
        private ushort SpeedMS;

        delegate void SetTextCallback(string text);
        delegate void PokeCallBack(ushort addr, byte value);

        public MainForm()
        {
            InitializeComponent();

            CompareFlag = 0;
```

This defines two delegates, one called SetTextCallback() for setting the text on our register bar and the second called PokeCallBack(), which takes care of updating our B32Screen object. As of right now, those are the only two visual objects in our interface that gets updated by a B32 bytecode file executing.

The next step is to create a new thread and initialize it with the parameters needed for ExecuteProgram(). Find the event handler for our File → Open menu item and add the following 2 lines and comment out 1 line at the bottom of the function:

```
            br.Close();
            fs.Close();

            InstructionPointer = ExecAddr;

            //ExecuteProgram(ExecAddr, Counter);
            System.Threading.Thread prog = new
System.Threading.Thread(delegate() { ExecuteProgram(ExecAddr, Counter); });
            prog.Start();
        }
```

What we did here was first, comment out the line that calls ExecuteProgram(). We no longer want to directly call this. Instead, on the next line, we created a new thread called `prog` and for the construction parameters, we passed it a delegate to our ExecuteProgram() function and also passed along the appropriate parameters to the function. Finally, the last new line we added starts the thread.

Believe it or not, we are actually almost done. We need to add three short functions to our program:

```
        private void ThreadPoke(ushort Addr, byte value)
        {
            if (b32Screen1.InvokeRequired)
            {
                PokeCallBack pcb = new PokeCallBack(Poke);
                this.Invoke(pcb, new object[] { Addr, value });
            }
            else
            {
                Poke(Addr, value);
            }
        }

        private void Poke(ushort Addr, byte value)
        {
            lock (b32Screen1)
            {
                b32Screen1.Poke(Addr, value);
            }
        }

        private void SetRegisterText(string text)
        {
            lblRegisters.Text = text;
        }
```

The ThreadPoke() function will a new function to "poke" data into a B32Screen without having to reference the `b32Screen1` object that our program currently uses. How this works is, all objects that are inherited from the `Object` class in .NET (which is all CLR objects) have a property called `InvokeRequired`. This property is a Boolean property that is set to true if the object is being access by a thread. If the main program is accessing the object (not through a child thread), the property is set to false. If the property is true, then we invoke the object through a delegate function, which is what we are doing above. If it's false, then we can access the object directly. The Poke() function is accessed indirectly through a delegate, via our Invoke() method, which is a CLR method common to all .NET objects. The "lock" statement inside our Poke() function tells our thread to "lock" the `b32Screen1` object. By locking this object, it prevents any other thread from accessing it. This is always a good (and sometimes necessary) practice when multithreading. This will definitely become necessary when we have 2 B32Screens with two separate threads accessing both screens (which will happen later in this tutorial). The SetRegisterText() function will also be accessed through an invoke(), coming up next.

The next thing we need to is make a small change in the ExecuteProgram() function. Since STA is our only byte code so far that actually can write to the screen, we need to change it so that is will use our new ThreadPoke() function. Find the ExecuteProgram() function, then make the following changes:

```
                if (Instruction == 0x03) // STA ,X
                {
                    B32Memory[Register_X] = Register_A;
                    //b32Screen1.Poke(Register_X, Register_A);
                    ThreadPoke(Register_X, Register_A);
                    InstructionPointer++;

                    UpdateRegisterStatus();

                    continue;
                }
```

What we did was comment out the old line and added our call to ThreadPoke(). The final change is to the UpdateRegisterStatus() function and its shown here:

```
        private void UpdateRegisterStatus()
        {
            string strRegisters = "";

            strRegisters = "Register A = $" +
Register_A.ToString("X").PadLeft(2, '0');
            strRegisters += "     Register B = $" +
Register_B.ToString("X").PadLeft(2, '0');
            strRegisters += "     Register D = $" +
Register_D.ToString("X").PadLeft(4, '0');
            strRegisters += "\nRegister X = $" +
Register_X.ToString("X").PadLeft(4, '0');
            strRegisters += "   Register Y = $" +
Register_Y.ToString("X").PadLeft(4, '0');
            strRegisters += "   Instruction Pointer = $" +
InstructionPointer.ToString("X").PadLeft(4, '0');
            strRegisters += "\nCompare Flag = $" +
CompareFlag.ToString("X").PadLeft(2, '0');
```

```
            //this.lblRegisters.Text = strRegisters;
            if (lblRegisters.InvokeRequired)
            {
                SetTextCallback z = new SetTextCallback(SetRegisterText);
                this.Invoke(z, new object[] { strRegisters });
            }
            else
            {
                SetRegisterText(strRegisters);
            }
        }
```

We are doing the same thing here that we did in ThreadPoke(). We are simple checking InvokeRequired to see if we need to do an invoke and if so, we are creating a delegate to SetRegisterText().

Go ahead and build and run our solution now. Choose a speed of 250 milliseconds or 500 milliseconds and run our test B32 bytecode program. Notice the register bar accurately updates as the program is running. Also notice that our interface doesn't freeze. We can move the window around, pull down our "File" menu and even change the speed of the program dynamically, as its running.

One annoyance you may have noticed earlier on when we first developed our B32 virtual machine is that each time we go to open a B32 bytecode program and run it, the screen does not reset. Essentially, the data in our 64K memory retains its values every time we load a new program or the same program. I would like to fix this annoyance now.

Bring up the code for the B32Screen control. Make the following changes to the constructor and add the following function:

```
        public B32Screen()
        {
            InitializeComponent();
            m_ScreenMemoryLocation = 0xA000;
            m_ScreenMemory = new byte[4000];

            //for (int i = 0; i < 4000; i += 2)
            //{
            //    m_ScreenMemory[i] = 32;
            //    m_ScreenMemory[i + 1] = 7;
            //}
            Reset();
        }

        public void Reset()
        {
            for (int i = 0; i < 4000; i += 2)
            {
                m_ScreenMemory[i] = 32;
                m_ScreenMemory[i + 1] = 7;
            }
            Refresh();
        }
```

All we did here was comment out our earlier initialization code and copy it to a public method called Reset(). We are then calling our Reset() function in the constructor to do a first time reset to the screen.

The final change we are going to make to fix our annoyance will be to the code behind in the main application. Open the event handler for our File → Open menu click event. Make the following changes:

```
        private void openToolStripMenuItem_Click(object sender, EventArgs e)
        {
            byte Magic1;
            byte Magic2;
            byte Magic3;
            DialogResult dr;

            dr = openFileDialog1.ShowDialog();

            if (dr == DialogResult.Cancel) return;

            lock (b32Screen1)
            {
                b32Screen1.Reset();
            }

            System.IO.BinaryReader br;
```

I added a local variable called `dr`. This variable will determine if the user canceled or closed the dialog box. If so, we just return. Otherwise, we lock the b32Screen1 object (just in case the user clicked "open" while a B32 binary was running) and perform a Reset().

Go ahead and run the program. Run our test B32 bytecode file, then change the speed to something like 3000 milliseconds, then run our test file again. You will notice, this time, it does, indeed reset the screen for us.

There is one final piece of business I'd like to do before I close out our discussion of multithreading. I want to implement a "Restart" and a "Pause/Resume" feature to our virtual machine. Since our program is threaded now, it makes it a lot easier to implement these sort of functions.

Earlier we defined a local variable called `prog`. This variable was a System.Threading.Thread object that executed our program. We need to change the scope of this variable so that it can be accessed from anywhere in the class. So to do that, add the following line to the member declaration section at the top of the class:

```
    public partial class MainForm : Form
    {
        private byte[] B32Memory;
        private ushort StartAddr;
        private ushort ExecAddr;
        private ushort InstructionPointer;
        private byte Register_A;
        private byte Register_B;
        private ushort Register_X;
        private ushort Register_Y;
```

```
        private ushort Register_D;
        private byte CompareFlag;
        private ushort SpeedMS;

        private System.Threading.Thread prog;

        delegate void SetTextCallback(string text);
        delegate void PokeCallBack(ushort addr, byte value);
```

Next, give it a default value in the constructor:

```
        public MainForm()
        {
            InitializeComponent();

            prog = null;
            CompareFlag = 0;
            SpeedMS = 0;
            realTimeNoDelayToolStripMenuItem.Checked = true;
            B32Memory = new byte[65535];
```

Finally, comment out our declaration from earlier and add the following line to the "File" → "Open" event click handler:

```
            br.Close();
            fs.Close();

            InstructionPointer = ExecAddr;

            //ExecuteProgram(ExecAddr, Counter);
            //System.Threading.Thread prog = new
System.Threading.Thread(delegate() { ExecuteProgram(ExecAddr, Counter); });
            prog = new System.Threading.Thread(delegate() {
ExecuteProgram(ExecAddr, Counter); });
            prog.Start();

        }
```

Okay, running and executing the program now should do exactly what it did before. The only change is the thread is now class-wide.

Even though all Thread objects have methods for pausing and resuming thread executing (Suspend(), Resume()), it is not a good idea to use these. Microsoft provided us with them for the purpose of debugging and not to use on production code. The reason these shouldn't be used is because there is no way of knowing what the process is doing at the time of suspension. If the thread was processing a block of code inside one of our lock blocks, then the variable locked because suspended indefinitely, causing what's called a "dead lock".

Because of this, we are going to approach this a different way. Create a new instance member at the top of the class:

```
public partial class MainForm : Form
```

```
    {
        private byte[] B32Memory;
        private ushort StartAddr;
        private ushort ExecAddr;
        private ushort InstructionPointer;
        private byte Register_A;
        private byte Register_B;
        private ushort Register_X;
        private ushort Register_Y;
        private ushort Register_D;
        private byte CompareFlag;
        private ushort SpeedMS;

        private System.Threading.Thread prog;
        private System.Threading.ManualResetEvent PauseEvent;

        delegate void SetTextCallback(string text);
        delegate void PokeCallBack(ushort addr, byte value);
```

Our thread will check the state of this PauseEvent and if its set, our program will wait indefinitely until its reset. While you are near the top of the class, add the following two lines to the constructor:

```
        public MainForm()
        {
            InitializeComponent();

            prog = null;
            CompareFlag = 0;
            SpeedMS = 0;
            realTimeNoDelayToolStripMenuItem.Checked = true;
            resumeProgramToolStripMenuItem.Enabled = false;
            pauseProgramToolStripMenuItem.Enabled = true;
            B32Memory = new byte[65535];
            StartAddr = 0;
```

These 2 lines will default the "Resume" menu item to disabled (since there is no program running currently paused when we first run our virtual machine) and the "Pause" menu item to enabled.

The next change we need to make is in the click event handler for our File → Open menu item. Go to the bottom of the function and add the following line of code:

```
            br.Close();
            fs.Close();

            InstructionPointer = ExecAddr;

            //ExecuteProgram(ExecAddr, Counter);
            //System.Threading.Thread prog = new
System.Threading.Thread(delegate() { ExecuteProgram(ExecAddr, Counter); });
            prog = new System.Threading.Thread(delegate() {
ExecuteProgram(ExecAddr, Counter); });
            PauseEvent = new System.Threading.ManualResetEvent(true);
            prog.Start();
              }
```

What this will do is create a new ManualResetEvent. As you will see in just a minute, we check this with each iteration of our code processing loop and our loop process will pause depending on weather this is reset or set.

The next change we need to make is in the ExecuteProgram() function. Add the following line right after our Sleep() call:

```
        private void ExecuteProgram(ushort ExecAddr, ushort ProgLength)
        {
            ProgLength = 64000;
            while (ProgLength > 0)
            {
                byte Instruction = B32Memory[InstructionPointer];
                ProgLength--;
                System.Threading.Thread.Sleep(SpeedMS);
                PauseEvent.WaitOne(System.Threading.Timeout.Infinite);

                if (Instruction == 0x02) // LDX #<value>
                      {
```

The WaitOne() method will pause indefinitely, if our PauseEvent is reset. Create a click event handler for our “Pause” menu item and add the following code to it:

```
        private void pauseProgramToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            resumeProgramToolStripMenuItem.Enabled = true;
            pauseProgramToolStripMenuItem.Enabled = false;
            PauseEvent.Reset();
        }
```

Now create a click event handler for our “Resume” menu item and add the following:

```
        private void resumeProgramToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            resumeProgramToolStripMenuItem.Enabled = false;
            pauseProgramToolStripMenuItem.Enabled = true;
            PauseEvent.Set();
        }
```

Go ahead and run the program. Set the speed to 3000 milliseconds, then open our test B32 bytecode file. As its running, go ahead and try to pause our program.

The last thing we need to do is get our “Restart” working. Create a click handler for our “Restart” click event and add the following code:

```
        private void restartProgramToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            if (prog != null)
```

```
            {
                prog.Abort();
                prog = null;
            }
            InstructionPointer = ExecAddr;

            resumeProgramToolStripMenuItem.Enabled = false;
            pauseProgramToolStripMenuItem.Enabled = true;

            prog = new System.Threading.Thread(delegate() {
ExecuteProgram(ExecAddr, 64000); });
            PauseEvent = new System.Threading.ManualResetEvent(true);
            b32Screen1.Reset();
            prog.Start();
        }
```

The first thing we do here is check to see if there is already a thread running. If so, we abort it and set `prog` to null. Next, we point our InstructionPointer to the beginning of the execution address. Finally, we create a new thread and a new PauseEvent and then we reset the B32 Screen and start the thread.

Go ahead and run the program again and open our test B32 file. After it executes, you should be able to restart it as many times as you wish. This wraps up our multithreaded discussion. As we expand our virtual machine, we will do it, keeping in mind the stuff we learned here. The next section, we will continue to improve our assembler and virtual machine by adding some "Pseudo-mnemonics" and a couple more regular mnemonics as well as some flags.

Planning More Stuff Out

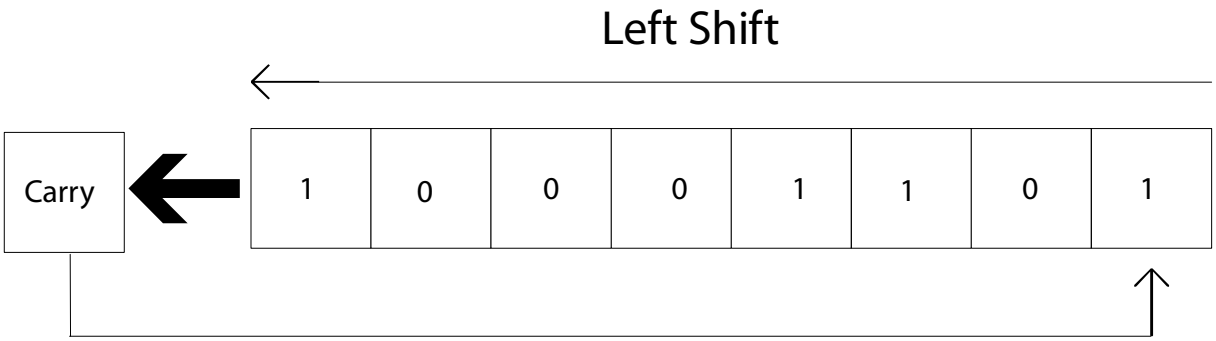
Most processors (I may even venture to say ALL processors) have an accessible byte (or sometimes 2 bytes) called "flags". We've seen this already when we made our compare flags. The flags we make in this section will be different. They will provide us with feedback on certain mathematical operations. For our virtual machine, we are going to make at least 2 flags. The first flag will be called an overflow flag. An overflow flag is a flag that is set whenever the result of a mathematical operation "overflows". Overflowing occurs when you try to hold a number larger then it can handle. For example, if register 'A' contains $FF and we try to add 5 to this, an overflow would occur. Register 'A' would contain $04 and the overflow flag would flip on. This is because $FF or 255 is the highest number an 8-bit register can contain. So the register rolls over to 0. You can actually see this in action in the .NET runtime. Create a real simple Windows Forms solution and insert the following code somewhere in the program, in the constructor or Load event:

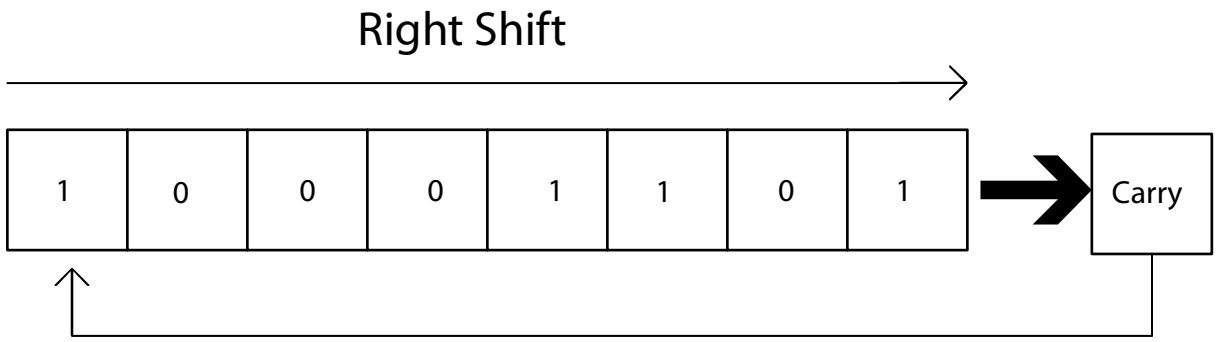
```
            byte j = 0xff;

            unchecked { j += 5; }
```

If you set a breakpoint and follow the value of "j", you will see that it is first assign the value 255. After the next line is hit, the value of "j" is 4. We had to put this in an "unchecked" code block, otherwise .NET

would not do the operation and instead, throw a runtime error. Our virtual machine will mimic this behavior, except instead of throwing an error; it will set our overflow flag to 1.

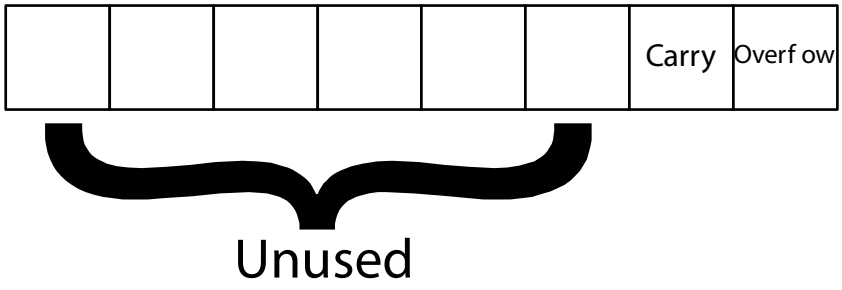
The second flag we are going to introduce is called the "carry" flag. This flag, in most processors, have multiple uses. In the case of our Virtual Machine, for now, the purpose will be limited to bit manipulation routines. We are going to introduce something called a "rotate". A rotate will shift the bits of a number either to the left or to the right. What makes it different from a "logical shift" is that each bit is carried through our carry flag. The following drawings will make it clearer:





You'll see why this is useful, later.

We are going to add a new byte, called 'F' which holds the flags for our new 3 flags. The byte will look like this:



In addition to the new flag byte, we are going to add the following new mnemonics:

Mnemonic	Description	Example	What will this example do?
INCA $0F	Increment the value in the 'A' register by 1	INCA	Increment the value of the 'A' register by 1
INCB $10	Increment the value in the 'B' register by 1	INCB	Increment the value of the 'B' register by 1
INCX $11	Increment the value in the 'X' register by 1	INCX	Increment the value of the 'X' register by 1
INCY $12	Increment the value in the 'Y' register by 1	INCY	Increment the value of the 'Y' register by 1
INCD $13	Increment the value in the 'D' register by 1	INCD	Increment the value of the 'D' register by 1
DECA $14	Decrement the value in the 'A' register by 1	DECA	Decrement the value of the 'A' register by 1
DECB $15	Decrement the value in the 'B' register by 1	DECB	Decrement the value of the 'B' register by 1
DECX $16	Decrement the value in the 'X' register by 1	DECX	Decrement the value of the 'X' register by 1
DECY $17	Decrement the value in the 'Y' register by 1	DECY	Decrement the value of the 'Y' register by 1
DECD $18	Decrement the value in the 'D' register by 1	DECD	Decrement the value of the 'D' register by 1
ROLA $19	Rotate 'A' register to the left	ROLA	Rotates 'A' register to the left
ROLB $1A	Rotate 'B' register to the left	ROLB	Rotates 'B' register to the left
RORA $1B	Rotate 'A' register to the right	RORA	Rotates 'A' register to the right
RORB $1C	Rotate 'B' register to the right	RORB	Rotates 'B' register to the right
ADCA $1D	Adds 1 to the value in 'A' register, IF carry flag is set	ADCA	Adds 1 to the value in 'A' register, IF carry flag is set
ADCB $1E	Adds 1 to the value in 'B' register, IF carry flag is set	ADCB	Adds 1 to the value in 'B' register, IF carry flag is set
ADDA $1F	Adds a value to the 'A' register	ADDA #$30	Adds $30 to the value in 'A' register, storing the value in 'A' register
ADDB $20	Adds a value to the 'B' register	ADDB #$30	Adds $30 to the value in 'B' register, storing the value in 'B' register
ADDAB $21	Adds the value of the 'A' register to the value of the 'B' register and stores the result in the 'D' register	ADDAB	Adds the value of the 'A' register with the value of the 'B' register, storing the result in the 'D' register
LDB $22	Loads a value into the 'B' register	LDB #$A0	Places $A0 in the 'B' register
LDY $23	Loads a value into the 'Y' register	LDY #$89FF	Places $89FF in the 'Y' register

This looks like a lot new stuff, but really its only 3 different functions repeated for all the registers. Notice I added an 'LDY' and a 'LDB' mnemonic. We created an 'LDA' mnemonic earlier, but failed to make an 'LDB' mnemonic or 'LDY' mnemonic. Truth be told, we had no use for it till now.

Open Test.ASM in notepad and type the following program (remember to precede each mnemonic with a space and end the last line with a carriage return):

```
Start:
 LDX #$A000
 LDY #8
 LDA #48
 LDB #$81
Loop1:
 ROLB
 ADCA
 STA ,X
 LDA #48
 INCX
 INCX
 DECY
 CMPY #$00
 JNE #Loop1
 END Start
```

Big, Big points if you can figure out what this program will do. It might stump you at first, but once I explain in, it should make sense. What it will do is output the 8-bit binary equivalent of the number loaded in the 'B' register. How it works is, it starts by rotating the 'B' register to the left. The left most bit is "dropped" into the carry flag. Register 'A' is preloaded with 48. This is ASCII for '0'. ASCII for '1' is 49. So after the rotate, we are adding the value in the 'A' register (which is 48) with the value in the carry flag. If the carry flag is 0, then register 'A' stays 48. If the carry flag is 1, then register 'A' becomes 49. This value is then outputted to the screen. We then increment the 'X' register twice (remember twice, not once to skip the attribute byte). We then decrement the 'Y' register and check to see if it's 0 yet. If not, it loops back to our Loop1 label and repeats the process till all 8 bits are printed.

Time to implement these new mnemonics. Open Visual Studio and open your B32Assembler project. The easiest ones to implement will be 'LDB' and 'LDY', so let's do those first. Add the following two lines to the ReadMneumonic() function:

```
            if (Mneumonic.ToUpper() == "JNE") InterpretJNE(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "JGT") InterpretJGT(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "JLT") InterpretJLT(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "LDY") InterpretLDY(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "LDB") InterpretLDB(OutputFile,
IsLabelScan);
```

```
            if (Mneumonic.ToUpper() == "END") { IsEnd = true;
DoEnd(OutputFile, IsLabelScan); EatWhiteSpaces(); ExecutionAddress =
(ushort)LabelTable[(GetLabelName())]; return; }
```

Next, add the following functions to our class:

```
        private void InterpretLDB(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                byte val = ReadByteValue();
                AsLength += 2;
                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x22);
                    OutputFile.Write(val);
                }
            }
        }

        private void InterpretLDY(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                ushort val = ReadWordValue();
                AsLength += 3;
                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x23);
                    OutputFile.Write(val);
                }
            }
        }
```

It should be no mystery how these work. They work identical to InterpretLDA() and InterpretLDX(). In fact, the only change that is different between the functions is the bytecode value that's written.

Next, we are going to write the increment and decrement code. Add the following lines to the ReadMneumonic() function:

```
            if (Mneumonic.ToUpper() == "LDY") InterpretLDY(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "LDB") InterpretLDB(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "INCA") InterpretINCA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "INCB") InterpretINCB(OutputFile,
IsLabelScan);
```

```
            if (Mneumonic.ToUpper() == "INCX") InterpretINCX(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "INCY") InterpretINCY(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "INCD") InterpretINCD(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "DECA") InterpretDECA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "DECB") InterpretDECB(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "DECX") InterpretDECX(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "DECY") InterpretDECY(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "DECD") InterpretDECD(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "END") { IsEnd = true;
DoEnd(OutputFile, IsLabelScan); EatWhiteSpaces(); ExecutionAddress =
(ushort)LabelTable[(GetLabelName())]; return; }
```

And now, add the supporting functions to the class:

```
        private void InterpretINCA(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x0F);
            }
            AsLength++;
        }

        private void InterpretINCB(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x10);
            }
            AsLength++;
        }

        private void InterpretINCX(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x11);
            }
            AsLength++;
        }

        private void InterpretINCY(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
```

```
                OutputFile.Write((byte)0x12);
            }
            AsLength++;
        }

        private void InterpretINCD(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x13);
            }
            AsLength++;
        }

        private void InterpretDECA(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x14);
            }
            AsLength++;
        }

        private void InterpretDECB(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x15);
            }
            AsLength++;
        }

        private void InterpretDECX(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x16);
            }
            AsLength++;
        }

        private void InterpretDECY(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x17);
            }
            AsLength++;
        }

        private void InterpretDECD(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
```

```
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x18);
            }
            AsLength++;
        }
```

All these function are pretty simple and they all work the same way. They write the bytecode, increment the `AsLength` counter and that's it.

We will now finish up by adding the last of our mnemonics. Add the following lines to the ReadMneumonic() function:

```
            if (Mneumonic.ToUpper() == "DECB") InterpretDECB(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "DECX") InterpretDECX(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "DECY") InterpretDECY(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "DECD") InterpretDECD(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "ROLA") InterpretROLA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "ROLB") InterpretROLB(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "RORA") InterpretRORA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "RORB") InterpretRORB(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "ADCA") InterpretADCA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "ADCB") InterpretADCB(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "ADDA") InterpretADDA(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "ADDB") InterpretADDB(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "ADDAB") InterpretADDAB(OutputFile,
IsLabelScan);
            if (Mneumonic.ToUpper() == "END") { IsEnd = true;
DoEnd(OutputFile, IsLabelScan); EatWhiteSpaces(); ExecutionAddress =
(ushort)LabelTable[(GetLabelName())]; return; }
```

And now add the supporting functions to the class:

```
private void InterpretDECD(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x18);
            }
            AsLength++;
        }
```

```
        private void InterpretADCA(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x1D);
            }
            AsLength++;
        }

        private void InterpretADCB(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x1E);
            }
            AsLength++;
        }

        private void InterpretADDA(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                AsLength += 2;
                if (IsLabelScan) return;
                ushort val = ReadByteValue();

                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x1F);
                    OutputFile.Write(val);
                }
            }
        }

        private void InterpretADDB(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            EatWhiteSpaces();
            if (SourceProgram[CurrentNdx] == '#')
            {
                CurrentNdx++;
                AsLength += 2;
                if (IsLabelScan) return;
                ushort val = ReadByteValue();

                if (!IsLabelScan)
                {
                    OutputFile.Write((byte)0x20);
                    OutputFile.Write(val);
                }
            }
```

```
        }

        private void InterpretROLB(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x1A);
            }
            AsLength++;
        }

        private void InterpretROLA(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x19);
            }
            AsLength++;
        }

        private void InterpretRORB(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x1C);
            }
            AsLength++;
        }

        private void InterpretRORA(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x1B);
            }
            AsLength++;
        }

        private void InterpretADDAB(System.IO.BinaryWriter OutputFile, bool
IsLabelScan)
        {
            if (!IsLabelScan)
            {
                OutputFile.Write((byte)0x21);
            }
            AsLength++;
        }
```

Go ahead and run the assembler and assemble our new Test.asm file. It should assemble without any issues.

Now that our assembler is complete, we will turn our focus back to the virtual machine. The first thing we want to do is make a member variable for our flags. Add the following line to the member definitions section of the class and another line in the constructor to initialize it:

```
        private ushort SpeedMS;
        private byte ProcessorFlags;

        private System.Threading.Thread prog;
        private System.Threading.ManualResetEvent PauseEvent;

        delegate void SetTextCallback(string text);
        delegate void PokeCallBack(ushort addr, byte value);

        public MainForm()
        {
            InitializeComponent();

            prog = null;
            ProcessorFlags = 0;
            CompareFlag = 0;
            SpeedMS = 0;
```

Just like we did with the assembler, we will implement the 'LDB' and 'LDY' mnemonics first. Add the following lines to the ExecuteProgram() function:

```
                if (Instruction == 0x02) // LDX #<value>
                {
                    Register_X = (ushort)((B32Memory[(InstructionPointer +
2)]) << 8);
                    Register_X += B32Memory[(InstructionPointer + 1)];
                    ProgLength -= 2;
                    InstructionPointer += 3;

                    UpdateRegisterStatus();

                    continue;
                }

                if (Instruction == 0x23) // LDY #<value>
                {
                    Register_Y = (ushort)((B32Memory[(InstructionPointer +
2)]) << 8);
                    Register_Y += B32Memory[(InstructionPointer + 1)];
                    ProgLength -= 2;
                    InstructionPointer += 3;

                    UpdateRegisterStatus();

                    continue;
                }

                if (Instruction == 0x01) // LDA #<value>
                {
                    Register_A = B32Memory[(InstructionPointer + 1)];
                    SetRegisterD();
                    ProgLength -= 1;
```

```
                InstructionPointer += 2;

                UpdateRegisterStatus();

                continue;
            }

            if (Instruction == 0x22) // LDB #<value>
            {
                Register_B = B32Memory[(InstructionPointer + 1)];
                SetRegisterD();
                ProgLength -= 1;
                InstructionPointer += 2;

                UpdateRegisterStatus();

                continue;
            }
```

These functions work identical to their counterparts, 'LDA' and 'LDX'. Next, we are going to implement the increment and decrement functions. These are easy and the code should be pretty straightforward. Add the following lines to the ExecuteProgram() function:

```
            if (Instruction == 0x0E) // JLT
            {
                ushort JmpValue = (ushort)((B32Memory[(InstructionPointer
+ 2)]) << 8);
                JmpValue += B32Memory[(InstructionPointer + 1)];

                if ((CompareFlag & 4) == 4)
                {
                    InstructionPointer = JmpValue;
                }
                else
                {
                    InstructionPointer += 3;
                }
                UpdateRegisterStatus();

                continue;
            }

            if (Instruction == 0x0F) // INCA
            {
                if (Register_A == 0xFF)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 1);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
                }

                unchecked { Register A++; }
                SetRegisterD();
                InstructionPointer++;
```

```
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x10) // INCB
            {
                if (Register_B == 0xFF)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 1);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
                }

                unchecked { Register_B++; }
                SetRegisterD();
                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x11) // INCX
            {
                if (Register_X == 0xFFFF)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 1);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
                }

                unchecked { Register_X++; }
                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x12) // INCY
            {
                if (Register_Y == 0xFFFF)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 1);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
                }

                unchecked { Register_Y++; }
                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x13) // INCD
```

```
            {
                if (Register_D == 0xFFFF)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 1);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
                }

                unchecked
                {
                    Register_D++;
                    Register_A = (byte)(Register_D >> 8);
                    Register_B = (byte)(Register_D & 255);
                }

                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x14) // DECA
            {
                ProcessorFlags = (byte)(ProcessorFlags & 0xFE);

                unchecked { Register_A--; }
                SetRegisterD();
                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x15) // DECB
            {
                ProcessorFlags = (byte)(ProcessorFlags & 0xFE);

                unchecked { Register_B--; }
                SetRegisterD();
                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x16) // DECX
            {
                ProcessorFlags = (byte)(ProcessorFlags & 0xFE);

                unchecked { Register_X--; }
                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x17) // DECY
            {
                ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
```

```
                unchecked { Register_Y--; }
                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x18) // DECD
            {
                ProcessorFlags = (byte)(ProcessorFlags & 0xFD);

                unchecked
                {
                    Register_D--;
                    Register_A = (byte)(Register_D >> 8);
                    Register_B = (byte)(Register_D & 255);
                }

                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }
```

These should be pretty straight forward functions. We are adding 1 for the increment functions and subtracting 1 for the decrement functions and setting our new flags appropriately. Next, we are going to add our rotate functions. Add these functions just below where you added the functions above:

```
            if (Instruction == 0x19) // ROLA
            {
                byte OldCarryFlag = (byte)(ProcessorFlags & 2);

                if ((Register_A & 128) == 128)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 2);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFD);
                }
                Register_A = (byte)(Register_A << 1);

                if (OldCarryFlag > 0)
                {
                    Register A = (byte)(Register A | 1);
                }

                SetRegisterD();
                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x1A) // ROLB
            {
                byte OldCarryFlag = (byte)(ProcessorFlags & 2);
```

```
                if ((Register_B & 128) == 128)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 2);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFD);
                }
                Register_B = (byte)(Register_B << 1);

                if (OldCarryFlag > 0)
                {
                    Register_B = (byte)(Register_B | 1);
                }

                SetRegisterD();
                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x1B) // RORA
            {
                byte OldCarryFlag = (byte)(ProcessorFlags & 2);

                if ((Register_A & 1) == 1)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 2);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFD);
                }
                Register_A = (byte)(Register_A >> 1);

                if (OldCarryFlag > 0)
                {
                    Register_A = (byte)(Register_A | 128);
                }

                SetRegisterD();
                InstructionPointer++;
                UpdateRegisterStatus();
                continue;
            }

            if (Instruction == 0x1C) // RORB
            {
                byte OldCarryFlag = (byte)(ProcessorFlags & 2);

                if ((Register_B & 1) == 1)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 2);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFD);
```

```
            }
            Register_B = (byte)(Register_B >> 1);

            if (OldCarryFlag > 0)
            {
                Register_B = (byte)(Register_B | 128);
            }

            SetRegisterD();
            InstructionPointer++;
            UpdateRegisterStatus();
            continue;
        }
```

To make these functions work, first we are saving the carry bit status to a temporary local variable. Then we are setting the carry bit, according to either the leftmost bit or rightmost bit (weather it's a left rotate or a right rotate). We are then shifting the bits and finally, setting the appropriate bit to a 1 or 0, which we got from the old carry flag.

Only thing left now is to add the functions that perform arithmetic addition. Add these functions just below the functions you just added:

```
        if (Instruction == 0x1D) // ADCA
        {
            if ((byte)(ProcessorFlags & 2) == 2)
            {
                if (Register_A == 0xFF)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 1);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
                }

                unchecked { Register_A++; }
                SetRegisterD();
            }
            InstructionPointer++;
            UpdateRegisterStatus();
            continue;
        }

        if (Instruction == 0x1E) // ADCB
        {
            if ((byte)(ProcessorFlags & 2) == 2)
            {
                if (Register_B == 0xFF)
                {
                    ProcessorFlags = (byte)(ProcessorFlags | 1);
                }
                else
                {
                    ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
                }
```

```
                unchecked { Register_B++; }
                SetRegisterD();
            }
            InstructionPointer++;
            UpdateRegisterStatus();
            continue;
        }

        if (Instruction == 0x1F) // ADDA
        {
            byte val = B32Memory[(InstructionPointer + 1)];

            if (Register_A == 0xFF && val > 0)
            {
                ProcessorFlags = (byte)(ProcessorFlags | 1);
            }
            else
            {
                ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
            }

            unchecked { Register_A += val; }
            SetRegisterD();

            InstructionPointer += 2;
            UpdateRegisterStatus();
            continue;
        }

        if (Instruction == 0x20) // ADDB
        {
            byte val = B32Memory[(InstructionPointer + 1)];

            if (Register_B == 0xFF && val > 0)
            {
                ProcessorFlags = (byte)(ProcessorFlags | 1);
            }
            else
            {
                ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
            }

            unchecked { Register_B += val; }
            SetRegisterD();

            InstructionPointer += 2;
            UpdateRegisterStatus();
            continue;
        }

        if (Instruction == 0x21) // ADDAB
        {
            if ((255 - Register_A) > (Register_B))
            {
                ProcessorFlags = (byte)(ProcessorFlags | 1);
            }
            else
```

```
                    {
                        ProcessorFlags = (byte)(ProcessorFlags & 0xFE);
                    }

                    unchecked { Register_D = (ushort)(((ushort)Register_B) +
((ushort)Register_A)); }

                    Register_A = (byte)(Register_D >> 8);
                    Register_B = (byte)(Register_D & 255);

                    InstructionPointer++;
                    UpdateRegisterStatus();
                    continue;
                }
            }
```

Go ahead and run the program. Open the test file we made and run it. It should print an 8-bit binary number to the screen. This is the binary representation of the hexadecimal number we loaded in the 'B' register. Feel free to change this number, re-assemble, than rerun our binary against it.

A Fond Farewell

I am closing out this document here. I am writing a part 2 that will pick up from this point exactly and continue on. I hope you enjoyed this tutorial and you learned something from it. There is sooooo much that can be added on. More mnemonics, keyboard input, graphics and/or multimedia extensions, etc. can all be added to this. I would love to see examples of B32 programs. Creative programs. Feel free to write me about questions or comments at icemanind@yahoo.com. I always look forward to emails.