Assembly Language Primer for the Absolute Beginner

Doug Dingus (potatohead)
opengeek@gmail.com
01/12/09 Version 08

Table of Contents

Introduction

Welcome to the binary world! Computers, despite appearing far more complex, really do only two things:

-**add numbers together, or perform logic operations on them (operate)**

,and

-**copy them around**.

Additionally, the only numbers computers really understand are 1 and 0. These are known as on and off, set and reset, high and low, etc... Something is either happening, or it isn't.

That's it! (Well mostly it. There are the bit logic operators: AND, OR, NOT, XOR, etc... we will get into those later on.)

The reality is most computers do these things very fast and in combination. The computer you are using right now is more than just cool hardware. It's contains lots of code–specific combinations of ones and zeros, chained together to perform the many tasks we normally take for granted. This code, is the sum of many man years of thought applied to what appear to be the simplest of problems. Most of these problems are simple actually. It is the combinations that are hard.

Assembly language is the art of working right at the computers level. It's just one step up from entering in ones and zeros to get stuff done. Ultimately, everything your computer does, boils down to assembly language instructions. It's computing in the raw. It's also where all the real fun is at, for those in the know.

I'm writing this because I want to share some of the fun that comes with learning computing at this level. I'm also writing it because it's not all that easy to find anymore.

Computing today happens at the higher levels. We have languages, programs, means and methods of doing things, that translate down to assembly language and this is good. But, where does it all come from? How does one just build a computer from the ground up? Why are some computers slow, some fast? What makes the graphics appear on the screen, or that sound come out of the speaker?

To someone, who has not yet looked at computing at this assembly level, it all seems to be a bit of voodoo. You do stuff and other stuff happens.

Reading this guide will change all of that. Ideally, you will come to appreciate what computers do in a way that's just not so common anymore. Assembly is becoming something of a black art, forgotten, ignored in lieu of higher level things that appear to work just fine.

This guide is aimed at the Parallax Propeller Micro controller. Before you ask, I'll just get "what is a micro controller?" out of the way right now. The Propeller is a little computer on a single chip. There are other books, some with that exact title, that answer the question better than I will. For the purposes of this guide, single chip computer will serve just fine.

Single chip computers have a number of advantages over their bigger and far more complex siblings; namely, personal computers. Where assembly language programming is concerned, the primary advantage is there being absolutely nothing between you and the computer. This means it's going to do exactly what you tell it to, every time you ask it to.

More complex computers are following instructions that come from many sources. The Operating System is a set of instructions, as is the system firmware, the programs you run and use, along with the driver software that connects the various devices to the Operating System to form a whole.

By the time you, the user, get to doing anything, the computer has already been very busy!

Sometimes these instructions from others are conflicting, incomplete, or maybe just don't do exactly what you are looking for. Many people will look for new instructions (programs), or maybe start to learn to program in any number of "programming languages", which really are more instructions that help you write your own instructions.

Assembly is nothing like any of that. In assembly language, it's you and the CPU. Actually, on the Propeller, it's you and 8 CPU's, also known as COG's. Most micro-controllers are a single CPU affair. (Propeller specific terminology and core concepts are covered just a bit farther into the document!)

There is another implication here as well. In assembly language, the computer does exactly what you tell it to. This means, if your program does not work, it's your fault!

The Propeller, like most any computer, comes with some instructions that help to make programming easier. This is the SPIN language you may already be working with. Soon the Propeller will get a real C complier, which takes higher level programs and auto-generates assembly instructions for you.

If you are reading this, you are looking to get the most from your Propeller. That means you need to be able to work at the assembly level for one and only one reason:

you can't afford to have anything get between you and the Propeller.

With computers, making things easier often takes time. Specifically, computer time – cycles, potential to get work done, etc...

Programming in assembly language is as fast as it gets. Ask the computer to do something,

and it does it right then, no delays. Send your Propeller a SPIN program and it will do all sorts of things before it actually does anything you want it to do. None of this is bad. It is however, a trade off worth noting.

This document is now complete, with one basic assembly language program fully detailed. The related concepts, instructions, and terminology are explained in a detailed and conversational fashion. Updates will happen, from time to time as well. These will most often be in the form of an addendum that compliment the core introductory text.

The reader will be left with a solid foundation understanding of the elements of assembly language programming good enough to consume more advanced texts and progress from there, just as the author is doing!

Enjoy the ride!

Core Concepts

Before we get to doing any real assembly language programming, we've got to get some ground covered first. Computer math is like ordinary math, for the most part. Most of the trouble comes from having to learn different number representations, along with some terminology.

We also need to cover the basic math computers do. It is this basic math we use to do higher level things, in a fashion not unlike how LEGO toys combine together to form larger structures.

Let's jump right in!

Number Representations

The reality is we share the same numbers computers do. The big difference is in how we represent them! Skip this section, if you understand number representation basics. The take away here is that binary 1 (%), octal, or base 8 (%%1), decimal 1, and hexadecimal $1 are all the same value; namely, one!

Our representation of choice is base 10. One character, for each finger, arranged in digits, each representing greater powers of ten, building to the left thus:

$1234 = (1 \times 10^3) + (2 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$

We depend on the powers of 10, for this representation to work. These are:

$10^0 = 1$
$10^1 = 10$
$10^2 = 100$
$10^3 = 1000$

Two very common numerical representations we use to interact with computers are:

Binary (base 2 1 - 0) -- '%'

$\%1101 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 13$

Binary depends on the powers of 2, just as base 10 depends on the powers of 10. Each binary digit is known as a “bit”.

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$

$2^5 = 32$
$2^6 = 64$
$2^7 = 128$

Hexadecimal (base 16 FEDCBA9876543210) -- "$"

Each hex digit can hold 16 values, instead of just two or 10. They are represented by the characters 0 – F, noted above, where A = 10 and F = 15, with the other digits in sequence.

$\$F0E4 = (F \times 16^3) + (0 \times 16^2) + (E \times 16^1) + (4 \times 16^0) = 61668$

Like binary, hexadecimal depends on a different base set of powers:

$16^0 = 1$
$16^1 = 16$
$16^2 = 256$
$16^3 = 4096$

The hexadecimal number above can be broken down thus:

$(15 \times 1096) + (0 \times 256) + (14 \times 16) + (4 \times 1)$

These numerical representations are an essential part of assembly language programming, and for understanding how computers do things, why some numbers are faster than others, etc...

It is an extremely good idea to memorize the first 16 powers of two. They double, so it's not a hard task. With these, one can easily visualize binary numbers in decimal terms, and convert from binary to hexadecimal, by hand, if necessary.

Knowing the first 8 powers of 16 is a nice bonus!

There are other numerical representations, such as octal (base 8), not covered here.

The important thing to remember is a given value, may be represented many different ways, yet it is the same value. If we were to count up to the number 5, we would count:

1,2,3,4,5

In binary, this is:

1, 10, 11, 100, 101

In both cases, we are at the value represented by 5 or %101.

Before we move onto sizes and other core math elements, it is important to note that

computers deal with values the same way we do. However, when counting or indexing things, computers start with zero, where we start with one. This will be explained in more detail later.

Computer number sizes

Computers handle specific sized numbers. Unlike us, they cannot just add a digit, or take one away for clarity or convenience. They are, after all machines, and machines work in specific and logical ways.

The most common number sizes are: (in binary form)

Bit

%0 - This is a single binary digit, or bit. It's value is either one or zero.

Nibble

%0000 - Nibbles are half a byte, or four binary bits. Their value ranges from 0 to 15. Also known as one hexadecimal digit. Eg: $A

Byte

%11110000 – Bytes consist of 8 bits or two hexadecimal digits. They range from 0 to 255

The byte above is $F0 in hexadecimal.

Word

%1111111100000000 - Words are two bytes, 16 bits, 4 hexadecimal digits.

The word above is $FF00

Long

%11111111000000001111111100000000 – Longs are two words.

The long above is: $FF00FF00

From now on, the notations for hexadecimal and binary "$ %" will be used, along with the number size notation. Now we get to talk bits, bytes, nibbles, hexadecimal and binary. We are now ready to begin learning about the computer, having learned something about it's language.

Common computer numerical units

Kilobyte (Kb) = 1024 bytes = 2^9 = %1000000000

Megabyte (Mb = 1024 Kilobytes

There are some new units being introduced that are based off of 1000, not 1024. They are kib and mib, for kilobyte and megabyte. I don't think these make a lot of sense, largely because they do not leverage the powers of two, found everywhere in computers. I'm putting them here because you may bump into them, not because they have any real value.

Pages

A page refers to a discrete unit of size, generally based on one of the core powers of two, and or number size. A page of memory, for example, might be defined as being 256 bytes – just the exact amount one byte can differentiate. Pages can be anything though. 16 bytes, 65536 bytes (64Kbytes, or 64K), etc...

In modern computers, a page is generally much larger than that. For now, just know a page refers to a chunk of sequential memory addresses, almost always keyed to a power of two.

Personification

For a lot of reasons, it's easier to personify the computer. Treating it like a partner, instead of just some arbitrary thing, helps both with writing and to facilitate communication. This text will follow that convention, so don't be alarmed when you see it, instead of “the computer” all the time.

Personification, in general is a useful tool. Our brains are wired to relate to other people better than anything else. When we think in terms of people, often difficult concepts become easier to grasp in a manner similar to how difficult social situations are more easily grasped when one role plays, or identifies with one or more participants.

It's a learning tool, nothing more.

Computer Concepts -- “What is it? How does it work? What are the parts of it?”

CPU

The core of any computer is it's Central Processing Unit, or CPU. This is where the work gets done. Instructions and data come in, different data and instructions come out. It's all just ones and zeros. It all happens very fast however, and that's where the power is.

Think of the CPU like a big calculator. In fact, it's helpful to imagine you being the CPU,

and your calculator being one part of what a CPU does. For a computer, the CPU is the brain. It can make decisions, follow directions, read and write data and do math computations.

RAM

Brains and CPU's need memory. Memory is where instructions are stored, along with results and other data. Most CPU's are connected to a quantity of Random Access Memory, or RAM. The RAM is addressed, in sequence from the first storage location “0” to the last one, say $7FFF in the case of the Propeller.

Given the above units and terminology, we then can say the Propeller has 32kb of RAM. This is 32 * 1024 bytes = 32768 = $7fff

You might have noticed that $7fff does not equal 32768! It's really 32767, so why is that?

This is the first example of where a quantity differs from an address or index. Remember, computers count from zero. So the first element of anything is zero, not one like we are accustomed to.

When talking about the amount of ram, we use a quantity. That's the 32768 number. When talking about addresses, and in this case specifically, which addresses address the RAM for us, we then start from 0 and arrive at 32767, or $7FFF.

In the Propeller, we have what is called 16 bit address space. This gives us addresses that lie between $0000 and $FFFF. Half of that address space is RAM, ending at $7FFF. The other half, starting from $8000 and ending at $FFFF is ROM.

ROM

Read only memory is memory that is addressed just like RAM is, but it does not store anything new. One cannot put anything into ROM, only read from it. Of course, the question comes up, “what good is that?”.

Computers power up each time knowing nothing. In fact, their RAM is often filled with whatever numbers happen to result from everything powering up. ROM then is stable memory that contains instructions for the computer to follow before doing anything else.

In the case of the Propeller, it needs to get started looking for programs to run, clean up it's RAM, and do some other basic things for us. The ROM holds those instructions, plus some handy data, like characters, conversion tables and other goodies we might find handy to use in our own programs.

The ROM also contains the interpreter for the SPIN language you were likely using before

deciding to try assembly language.

I/O

On the propeller, it gets its input and sends it's output to and from the outside world, via it's 32 I/O pins. Each of these pins can be in a binary state, or depending on how you instruct the computer. There are other states too, like those used for the video, but those are beyond the scope of this document.

For now, this is enough to continue to learn about how assembly language works. Later we will explore the Propeller and it's 8 CPU's (COGs), the HUB and other on-chip capabilities.

Flags

CPUs note things while they are computing. If an operation results in a zero, or perhaps has a bit to carry over, these things are stored in the flags. Think of it like this. You are the CPU, and you are operating on numbers with your calculator.

So you read the number, that's like reading from RAM or ROM memory. You input it to the calculator, then you input another number and perform the addition. If that result ended up being zero, you would make a mental note of that. If that operation ended up being bigger than your calculator can handle, you note the overflow and carry that bit over to another addition maybe.

That's what the status flags are all about.

On the Propeller, we've got a **zero flag** and a **carry flag**.

There is also another context for the term 'flag', and that is all about preserving transient states that may need to be acted upon later, after the state has come and gone. In this context, a flag then is set, or reset to note an event or result for processing later on.

Anything can be a flag. Pin states, values –in fact any values, not just true or false, but just values, such as: $50, %00010101, or $FFFF. A flag, used in this context, is compared to a known flag state, then decisions are made based on the result.

Take video as an example. Video displays happen scan line by scan line. If there was a counter, ticking off the scan lines as they are drawn, it can act like a flag! All the programmer needs to do is instruct the computer to watch that scan line counter and make a decision when it's state matches a known one. Eg: branch, if scan line counter = $50

Opcodes (Instructions)

Each assembly language instruction is associated with a number. For sanity, we use an assembler to associate those numbers with mnemonics, such as **MOV ($101000)**. This instruction example moves a number from one place to another. There is a list of all the Propeller opcodes, along with verbose descriptions of their behavior in the Assembly Reference section of the Propeller Manual. Opcodes definitions will not be repeated here, other than for clarity.

At some level, everything we do with a computer is mathematical. Computers are number manipulation machines. This means we need to get good at representing things with numbers, so we can instruct the computer to manipulate them in useful ways. Having covered enough core concepts, it's time to talk some math.

Computer Math

As discussed before, computers really only add. They do lots of additions on lots of numbers very quickly to perform the tasks we require of them. If you are reading this on a computer, or near a computer, that computer is likely performing billions of adds per second, if not more.

On a side note, back when I first started computing, we were talking millions of adds every second. The progression of computer speed is staggering, given the short time we have been actually computing.

The Propeller chip does millions of adds per second. This is one of the things that differentiate it from a full on Personal Computer. Don't worry though. Millions of adds per second is still fast enough to do amazing things!

Be patient please! We've covered a fair amount of ground. It's almost time to begin writing some assembly language programs! Soon after that, we will be running them on the actual Propeller chip and my purpose here will have been accomplished. It's easy to skip the core basics and just start doing stuff. It's also harder to go back and figure out what went wrong that way too.

If you just have to skip forward and do stuff, do it! Then come back and continue with the core stuff here.

Addition (ADD)

This is the core operation. For our general sanity, typical CPU's provide a number of operations, such as: subtract, shift, and if you are lucky an multiply. All of these depend on addition, so it makes sense to cover that first.

Binary addition is just like ordinary addition, but for the fact we've really only got two digits! This makes things really easy actually. For any column of bits to be added, the result will either be 0, 1, 10 or 11 (0,1,2,3) with two and three needing a carry to over to the next column thus:

```
  1          1      <---- Carry
  23         010
 +49       + 011
  --         ---
  72         101    <----- Sum
```

Let's take a close look at the example above. On the left, we have ordinary decimal addition, just like we've all done on paper at one time or another. To get to the solution, we start with the right hand column, add the 3 and the 9 and get 12. That's two digits! More than will fit in that column, right?

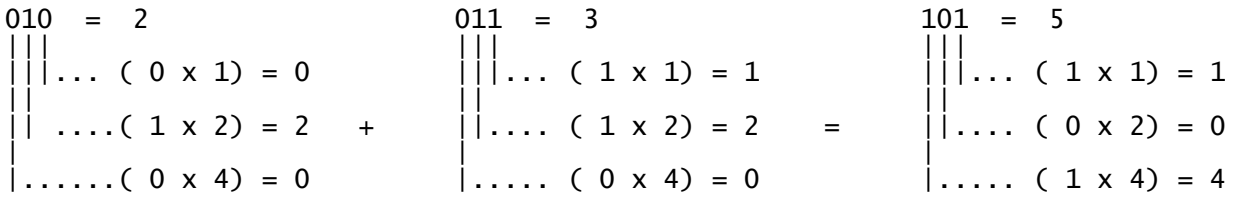
So, we carry the 1, note the 2 in the sum, then continue working on the next column.

That's a 2 plus 4, plus the 1, carried over, for a total of 7. We then can read the sum as being 72.

The binary example is no different! It's actually a lot easier because we've only got 0, 1, 10 or 11 to worry about. That's only four possibilities.

Working from the right then, we've got 0 plus 1, for a 1 in the sum. The next column over is 1 plus 1, so that's 10 –a carry. Carry the 1 to the third column, note the 0 in the sum, and add the last column 0 plus 0 plus 1, equals a 1, which goes in the last sum column, for a total of 101.

What did we add? Let's take a quick look at converting binary to decimal with the powers of two:



Essentially, all you need to do is take the powers of two, check to see if there is a one in their column, then add them all up. Either that power of two is represented in the number or it isn't! This is the beauty of binary. In the example above, we added 2 + 3 = 5.

One more example:

%10010011

Let's say the bits are numbered 0 to 7, starting at the right most bit. The conversion then looks like this:

1 + 2 + 0 + 0 + 16 + 0 + 0 + 128 = 147

We number bits, from right to left, according to the power of two, they represent. The right most bit is called least significant digit and the left most is called most significant digit. (**LSD & MSD**) Significance is all about the value any one digit represents. The farther left you go, the more significant it is.

> *Most personal computers have a calculator program that will do these number conversions for you. Use it, but only after you have worked through some addition and conversion on paper to understand what the digits mean, remember your powers of two and when to carry. These things are important going forward.*
>
> *A great exercise is to take some common numbers, convert them to binary,*

then convert them to hexadecimal, then back to decimal.

On a Microsoft Windows computer, you choose view, scientific from the program menu.

More on addition (The carry)
What happens when the result exceeds the number of columns in an addition? For us people, we just add another column and finish out the operation thus:

```
  11         11      <---- Carry
   23         110
 + 99       + 111
   --         ---
  122        1101    <----- Sum
```

When working on paper, we can do the same for binary numbers. Computers work a bit differently however. Computers work with fixed number sizes. If we are adding bytes, we are really adding numbers that range from 0 to 255. If the result is bigger than that, an extra digit, or carry is required. This can easily happen and has some interesting effects that will lead us to subtraction and modulo operations.

Let's start with a byte addition example:

```
   %11001110
 + %01100001
   ---------
 1  00101111
```

In this case, we've got a result that is bigger than one byte can hold, so where does the extra digit go?

Depending on what you've asked the computer to do, that extra digit might end up in a special location, called a flag, for use in a subsequent addition operation, or decision (branch) operation, or it is just ignored!

Both of these cases are meaningful and useful. Let's take the example where the extra digit is stored in the **carry flag**. On the Propeller, the CPU maintains two flags, to be discussed in more detail later. For now, all we need to know is that the carry flag will hold that overflow digit for us, if we want to.

In the example above, we are gonna need two bytes to hold the result. That looks like this:

```
          1 <------
%00000000                    %11001110
%00000000      Carry Flag  + %01100001
---------                    ---------
```

```
%00000001          1 <-----    %00101111
```

In this example, the next byte to be added is zero. The carry then just “carries” over to the least significant digit, and the addition begins again. The result of our addition then is a two byte number: %00000001_00101111

Delimiters

The underscore is a thing called a delimiter. It's ignored by the Propeller programming tool, and exists for clarity. Big computer numbers are hard for us to process. Actually, big numbers in general are hard for us to process.

Over time, it has been found that an ordinary person feels good about 3 – 5 digits. Past that, they've got to look the number over a few times and may get confused. That's what the delimiter is for.

The number of digits problem is why hexadecimal is so handy. Each digit is powers of 16, so we can represent very large numbers with few digits.

We would use the comma “,” like we do ordinary decimal numbers, but that character symbol sees a lot of use in computer programming. For a lot of reasons, it's just easier to use something else. The Propeller tool actually lets you pick. In this document, the underscore will be used. It's also the most common seen in the wild.

Back to the math...

The carry example above opens the door to adding really big numbers. The Propeller can handle up to long sized binary numbers. That's 32 bits, or a range of: 0 to 4294967296!

For most tasks, a 32 bit number is just fine. It's range is plenty for computation, indexing, etc...

However, there will come times when it's just not big enough, so we need to briefly expand on the carry example and add two discretely sized numbers together. For clarity, bytes will be used. It all works the same however, no matter what sizes are actually in play. Adding a one byte number to a two byte one looks like this:

```
          1    1111
    %00110000_11001110
+             %10010011
    ------------------
    %00110001_01100001
```

Note the carry over to the most significant byte above.

If the computer were only able to work with byte sized numbers, two adds would be required, and that looks familiar:

```
          1 <---
%00110000                    %11001110
%00000000      Carry Flag  + %10010011
---------                    ---------
%00110001          1 <-----  %01100001
```

Another important concept comes into play here, and that is extending a number. In the case of adding one byte, to a two byte number, we really perform two adds, and that means having two operands for the addition.

Picture it like this:

```
          1   1111
    %00110000_11001110
+   %00000000_10010011
    ------------------
    %00110001_01100001
```

Adding two byte numbers is really no different, because we extend the smaller one anyway. In the example above, we really could just put some ones in there and be adding two bytes.

> *It is important to note that adding same sized numbers (byte to byte - word to word), will result in an overflow -that extra carry digit. This is always true, no matter what size is worked with.*
>
> *Anytime you are adding numbers, you need to decide what you want to do with the carry bit. Either ignore it, make a decision based on it, or add it to the remainder of the addition being performed.*

That brings us back to case number two above, where we ignore the carry. What happens then?

Modulo (or wraparound)

This is a very important assembly language concept. When you perform an operation, whose result is larger than the number size being worked with, that over flow goes into the carry flag, or is ignored.

This causes numbers to wraparound and start again at zero! Let's say we are working with nibbles, which are half a byte.

Nibbles hold the numbers 0 through 15. If we add one to a full nibble, we end up with an empty one back at 0! It wraps around.

Another way to look at it is when counting, the least significant digits go through a regular pattern. They do their thing over and over. In decimal, a great example is the odometer on your car. Most cars have two of these. One very large one to record total miles and one smaller one to note smaller distances.

If both of these are set to zero, and the smaller one holds three digits, it will wraparound back to zero, when the larger one continues on to note 1000. Computer numbers work the same way, wrapping around when all their bits are set to one.

```
0000 + 1 = 0001      1110 + 1 = 1111
0001 + 1 = 0010      1111 + 1 = 0000  (carry a 1)
```
That's how modulo works.

Computer numbers then, because of their discrete sizes are all modulo numbers. This is not a bug, but a feature! A useful one too.

This leads us to....

Subtraction

On paper, subtraction works just like addition does, only we've got a borrow instead of a carry. There is a short cut however and that shortcut takes advantage of the modulo / wraparound concept discussed above.

Really, we subtract by adding. This involves changing one of the operands so that when it is added to the other one, the result wraps around to end up correct, just as if we actually subtracted in the ordinary way. This is called complimenting a number and should be discussed briefly in the context of subtraction.

Complements

First, let's try it with decimal. Let's say we are working with a single digit:

```
  4              4
 -2             +8 <----  8 is the compliment of 2
  -              -
  2            (1)2
```

The key here is to ignore the carry and only look at the modulo / wraparound result. In this simple example, 8 is the compliment of two, because when added it does the same thing as subtracting a two would do.

Now let's look at some binary:

```
   %1001          %1001
 - %0110        + %1010
    ----          -----
    0011        (1)0011
```

Both of these give the same result, if we ignore the carry. Subtraction then can be done through addition. For bytes, one could add $ff to subtract 1, by way of another example.

The Propeller has a subtract instruction, so we really can just offer it two numbers, ask it to subtract and move on. Compliments do crop up all over the place however. For now, just know you can express any subtraction as an addition, and understand how that ties in with the modulo and wraparound concepts.

Multiplication

Like addition, multiplication works the same way addition does. The manual steps work the same as they do for decimal multiplication. Here is a quick example:

Note: 0*0 = 0 1 * 0 = 0 1 * 1 = 1 (that's it!)

```
   %101
 x % 11
   ----
    101
 + 101
  -----
   1111
```

Where decimal multiplication makes multiply by 10 really easy, binary works the same way, but for multiplying by 2. Just tack a zero on the end, and you've multiplied by two!

Division

Here is a quick division example.

```
      10001
    ________
  11)110011
    -11
     --
      000
    - 000
      ---
        011
        011
```

Decimal to Binary conversions

Remember those powers of two? They come in extremely handy for this task. Basically you need to find the greatest power of two, that can be subtracted from the decimal number and start from there, repeatedly subtracting powers of two, keeping track of those you've subtracted up along the way. When there is no more subtraction being done, you have your binary number.

Here is an example:

27 =

32 is the 6th binary digit, but it's too big. 16 is the next largest power of two, and it's less than 27, so we start there.

27 – 16 = 11

Now the process repeats. 8 is the next largest power of two that works, so we subtract that.

11 – 8 = 3

4 is too big, so we subtract two.

3 – 2 = 1

That leaves 1, and we subtract that as well.

1 – 1 = 0

All done! Now we tally up the powers of two we subtracted and arrive at our binary number:

Each power of two is a digit. Where we didn't use one, we place a zero, where we did, we place a 1.

```
% 0   1   1   0   1   1     ---->    %011011

  |   |   |   |   |   |

  32  16  8   4   2   1
```

The leading zero can be left off, leaving us with %11011

Negative or signed numbers

So far, all we've worked with is positive integers. It's time now to discuss signed numbers. Essentially, we've got to encode the sign into whatever bits we've got to work with. This breaks down a few ways:

Signed magnitude

This is where we use the upper most bit as a sign indicator. A zero is positive and a one is negative. If our number size is byte, then we've got 8 bits, with the left most bit being the sign thus:

```
%00000101   (+5)          becomes     %10000101   (-5)
```

We trade the highest number represented (255) for having a sign (+127 – 128)

One's compliment

Basically, you just flip all the bits! This leaves the upper most bit as the sign, just as in the example above. This is just a different representation.

```
%00000101   (+5)          becomes     %11111010   (-5)
```

Two's compliment

Start with ones compliment, then just add one.

```
%00000101   (+5)          becomes     %11111011   (-5)
```

It's important to note which representation was used. Without this, one cannot know what the original value was.

Most computers today operate directly on Two's compliment signed numbers. This includes the Propeller.

Example math operations

If this is your first pass through this document, you can skip this section, then come back after having worked through the program example. Sometimes one needs to know something, along with something else, for a concept to become clear. Too much at the wrong time can cause trouble and confusion.

Having started to read this section, you may experience this. Feel free to move on to the other foundation concepts and return later! These example bits of code are not meant to be run directly. They do perform their function, but are really intended to be consumed then used in the context of a greater program.

Adding two unsigned 32 bit numbers:

```
DAT
                        ORG 0          'Begin at Cog RAM addr 0
                        add NUM2, NUM1

NUM1        long %0001000_00000111_11111111_00000000
NUM2        long %1111000_00110110_11100011_00000001

NUM2        long %0000000_00111110_11100010_00000001  (Carry set)
```

After the add instruction executes, NUM2 will contain the second result shown, and the carry flag will be set, as the result is larger than 32 bits can hold.

The key here, to remember is the Destination contains the result of the math operation.

Set flag states (Zero and Carry flags)

Some operations require a known flag state to work properly. There are no instructions in the Propeller to directly manipulate the flag states. The easiest way to accomplish this is to simply perform an operation with a known result. Here are some known operations that will set or clear the carry and zero flags. They depend on two COG memory locations being defined as all ones or all zeros.

```
              ORG 0                    'Begin at Cog RAM addr 0

Carry       xor _TRUE, #0  nr, wc      'Clear Carry, Even Parity = 0
            xor _TRUE, #1  nr, wc      'Set Carry, Odd Parity = 1

Zero        xor _TRUE, _TRUE nr, wz    'Set Zero, Result = 0
            xor _TRUE, _FALSE nr, wz   'Clear Zero, Result = not zero


_TRUE       long $FFFFFFFF  'all ones
_FALSE      long $00000000  'all zeros
```

The instructions have two effects specified. The 'nr' effect says, “do the operation, but don't write it to COG memory.” The 'wz' or 'wc' effect says, “apply the result to the flag”. These two combined result in only the flag state changed, nothing else.

Adding with carry to represent larger numbers of different size, unsigned

in the example above, the result is larger than 32 bits. Here is an unsigned addition of a 32 bit number to a 64 bit one, to fully capture that result:

```
              ORG 0              'Begin at Cog RAM addr 0
Start         add   NUM2, NUM1   'add NUM1 to NUM2, carry
              addx  NUM2a, #0    'add the carry to NUM2a

NUM1       long %0001000_00000111_11111111_00000000 '32 bit source
NUM2       long %1111000_00110110_11100011_00000001 'lower 32 bits
NUM2a      long %0000000_00000000_00000000_00000000 'upper 32 bits


_TRUE       long $FFFFFFFF  'all ones
_FALSE      long $00000000  'all zeros
```

We start with the add instruction, because it will ignore the state of the carry, perform the add, then set the carry state in preparation for subsequent addx instructions. In the second addition, the #0 is used to represent the empty upper 32 bits of NUM1, to complete the addition. This is like the extension discussed in the addition section earlier.

Add 64 bit number to 64 bit number, unsigned

```
              ORG 0                 'Begin at Cog RAM addr 0
Start         add   NUM2, NUM1      'add NUM1 to NUM2, set carry
              addx  NUM2a, NUM1a    'add the carry to NUM2a

NUM1       long %0001000_00000111_11111111_00000000 'lower 32 bits
NUM1a      long %0001000_00000001_00000000_00000000 'upper 32 bits
NUM2       long %1111000_00110110_11100011_00000001 'lower 32 bits
NUM2a      long %0000000_00000000_00000000_00000000 'upper 32 bits
```

This is really no different than doing the 32 bit to 64 bit. Since both numbers are the same size, the '#0' is replaced with 'NUM1a' indicating the upper 32 bits of NUM1 are to be added.

The 64 bit result ends up in NUM2 and NUM2a.

Propeller Specific Core Concepts

Having covered the elementary math and terminology necessary, it's time now to start working on the actual instructions, memory, flags and how they all interact.

It's getting time to start thinking about the Propeller specific things we are going to need in order to build assembly language programs. This is perhaps the biggest difference between assembly language and other means and methods of interacting with the computer. Understanding both the hardware and the logic is necessary.

The propeller is an interesting design, in that it actually has 8 CPU's, known as COGs, working together. For the sake of simplicity, that will be ignored for this section, so that instructions and memory can be discussed.

There are some new concepts and terms to learn in this section. Please refer to the Propeller Block Diagram , figure 1, found in the ***Propeller™ P8X32A Preliminary Datasheet, available from*** ***http://www.parallax.com***

Clock

The Propeller runs from a system clock, common to all the COGs and the HUB. The speed of this clock depends on the attached oscillator and whether or not the Internal clock, or an external reference is used. See the data sheet or Propeller Manual for more detail.

The reference clock implementation is 80Mhz.

System Counter (CNT)

The system counter is a global, read only counter, that increments once every system clock cycle.

HUB

The HUB has a 16 bit address space that is divided into RAM and ROM, discussed earlier. All cogs share this memory space and must coordinate their activities in this space. **($0000 - $7fff)** The HUB ROM contains the instructions necessary to get the Propeller started running your programs. Programs can come from an attached EEPROM memory, directly or loaded into HUB Ram via the Propeller Tool IDE.

HUB memory is addressable on a byte, word or long basis.

The HUB does not execute anything. It is a data and communications facility only. This

means there is no central processor. All COGs perform identically. COG 0 does have some significance, in that it is the first COG started with your programs, after that a COG is just a COG and an I/O pin is just an I/O pin.

Moving information to and from the HUB takes some time that depends on where the COG doing the work is in the round-robin access sequence. Average access is ~15 cycles. This translates into 4 instructions worth of time for a hub access.

Data alignment

As discussed, computer number sizes are bytes, words and longs on the Propeller. When these number sizes are stored in HUB memory, the issue of alignment comes into play. It works like this:

The least significant two bits (bits 0 and 1) of an address determines an alignment. For bytes, both digits are significant. In other words, bytes can just be anywhere in the HUB memory. Either digit can be a zero or one, allowing for any address.

For words, the last digit is always set to zero.

For longs, the last two digits are set to zero.

Eg: Let's say we are looking at the address $1003. Where are properly aligned addresses?

```
byte $fe  --->         $1003   (no alignment issue)

word $eedd     ---->   $1002   (least significant binary digit a zero)

long $FFFF0000 ---->   $1000   (least two significant digits are zero)
```

COG

The 8 COGs are all connected to the central HUB and get access to it in round-robin fashion. All of the COGs are identical and feature their own counters, video generator and RAM memory space. Additionally, all COG share access to the 32 I/O pins. COGs execute their instructions in parallel. Only the access to the HUB is round-robin.

COG memory is only addressable on a long basis. The 2kb of COG memory **divides into 512 longs**, addressed from **$0 to $1ff**

COG memory addressing is a frequent source of confusion. Going forward, pay special attention to where you are working with data to mitigate this difference.

When a COG is started with the **COGNEW** instruction, it's memory space is copied from HUB memory, prior to executing whatever instructions are contained therein.

COG memory is not associative to HUB memory. Once loaded from the HUB, the HUB memory it came from may be changed without affecting the program and data on the running COG. This makes it possible to get the most use out of the 32KB of shared HUB memory.

COG 0

COG 0 is somewhat different from the other COGs, because it is the startup COG. When the Propeller chip starts, COG 0 runs the **Boot Loader** program. Your programs start from a small bit of SPIN language code, at a minimum, executed by the SPIN language interpreter, after the Boot Loader completes it's tasks.

For more detail please refer to the Propeller Manual, or Data sheet. Some of this material will be reproduced here for clarity or context. Going forward, it is assumed you have one or both of the Propeller Manual and Data Sheet.

For a significant portion of this guide going forward, the discussion will focus on a single COG, or perhaps two COG's. One that will run a SPIN program, and another running an assembly program.

Instructions

All propeller assembly language instructions are one Long in size. They have binary bit fields in the form:

100000	001i	1111	ddddddddd	sssssssss
Instruction	ZCRi	Condition	Destination (d)	Source (s)

- Instruction = The actual binary opcode associated with the mnemonic

- ZCRI = Effects (Instruction modifiers)
  - Z = Write Zero Flag (wz)
  - C = Write Carry Flag (wc)
  - R = Write result (wr) (Destination Register Modified)
    - 1 = Modified (wr)
    - 0 = Not Modified (NR)
  - i = Immediate addressing
    Operative value is in the instruction itself, rather than in the COG memory location normally associated with the value in the Source (s) field.

- Conditions
  These are conditional execution bits that can be added to most any

instruction. Whether or not the instruction executes, then depends on the states of the two flags. Typical conditions are:

- if_z - If the Zero Flag = 1, then execute the instruction
- if_nc - If the Carry Flag = 0, then execute the instruction

Propeller **assembly instructions normally consume 4 system clock cycles**. Branches take either 4 or 8, depending on if they are taken or not. Hub memory access takes more cycles and the exact amount is determined by where the COG happens to be in the round robin access scheme.

To keep things simple right now, instructions are 4 cycles. With an 80Mhz clock, this means 20 million instructions per second, per COG.

A instruction that interacts with the hub, consumes the equivalent of 4 COG only instructions, for a quick rule of thumb reference.

Little Endian

The Propeller is a little endian design. This is what you need to know:

Values and Indexes

Numbers are stored according to their Most Significant Byte **(MSB)**, in sequence to their Least Significant Byte **(LSB)**.

Here is an example:

```
$1A_2B_3C_4D          $1000 - $4D
                      $1001 - $3C
                      $1002 - $2B
                      $1003 - $1A
```

The MSB is stored lower in memory than the LSB.

The rule here is if you are operating on the data, you can store it however you want to store it. If you are asking the Propeller to operate on the data, then the endian design matters.

Instructions in RAM

Say an instruction assembles into the following binary long:

%101010_001i_1111_ddddddddd_sssssssss

It ends up stored in memory like this:

%ssssssss_ddddddds_1i1111dd_10101000

There is little need to directly manipulate instructions in memory, but you may find this useful if you are looking directly at HUB memory with another program or process for debugging.

Three instructions are available to directly modify the opcode bit fields in memory, making this process simple and clear.

The Propeller can only execute instructions out of COG RAM. Self modifying code is the norm on the Propeller and is frequently used for indexing tasks. Unlike most other CPUs, the Propeller has no index registers, meaning you the programmer must modify instructions directly to accomplish this task. They are:

movi - This instruction modifies the opcode bit field. You use it to change the actual **instruction opcode value** stored at the target COG address. An instruction could be fetched from a table, based on a value, for example.

movs - You use movs to directly modify the **source value** of the target instruction. This one typically is used in a loop where a value is read based on an index value.

movd - Is the compliment of movs, in that it's purpose is to modify the **destination value** of the target instruction. Typical use here is to store a value based on an index value.

Data (Strings, lookup tables, etc...)

You are completely free to organize data however you want to. Endian issues only apply to values directly manipulated by the Propeller. Eg:

```
DAT   Byte   $3D, $4E, $5F        $1F00 - $3D
                                  $1F01 - $4E
                                  $1F02 - $5F
```

Program Counter

Each COG has a program counter that keeps track of where in memory it is. The Program Counter works upward through RAM as instructions are completed. Branch and Jump instructions essentially load the Program Counter with a new target address, allowing for program control. The Program Counter always counts up.

SPIN

Spin is the higher level, interpreted language, native to the Propeller. All propeller programs contain at least the bare minimum SPIN program required to start the Propeller running, and hand control off to an assembly language program.

In this guide, we will work with simple assembly language programs that operate as a sub-program to a main SPIN program, followed by assembly language programs that only need SPIN to get started.

It's easy to work with assembly in small bits, then build up as understanding grows. It won't be necessary to fully understand how the SPIN programs presented work, in order to understand the assembly language portions.

There is an excellent introduction to SPIN in the Propeller Manual.

What is a Register?

On a lot of CPU's there are these memory storage areas defined inside the CPU itself. If the CPU goes to do something with an element of memory, it loads that into a register, then proceeds to operate on it. Let's pretend for a moment that you are the CPU and you are going to add two numbers in your head. So, you acquire both numbers, then operate on them, holding the sum in your head also.

This is what registers do.

Now that sum really isn't any good, unless recorded somewhere. Of course it might also be an intermediate value as well. If you write the number down somewhere, that's like a memory storage operation. If you just keep the sum in your head, perhaps to be added to something else, that's like keeping the value in a register.

CPU's that have registers only have so many of them. Additionally, they operate on their registers, which requires them to fetch **data, better known as operands**, from memory, as well as write them back to memory when operations are complete. Like us, CPU's can only keep track of so many things at any given time. That's why we have memory storage. Memory is there for when things get too complex for registers alone.

Long ago, on the 6502, I worked with it's registers. They were called the Accumulator, X index Register and Y index register. There were others too.

The take away here is that registers are differentiated largely by how they are addressed, and that they are part of operations. On many CPU's registers are memory locations that are not addressed like RAM or ROM storage is. They have names, like accumulator, program counter, etc...

On your typical CPU then, we've got memory (RAM or ROM) and registers as places where

numbers may be copied to, from and operated on. On CPU's with well defined registers, their purpose is often sharply defined as well, meaning one has to worry about what register is being used for what, and when and how. Considerable time and effort is required to make best use of the limited registers.

On the Propeller, memory locations are the registers, where operations are concerned! It is not necessary to load an operand into a register, operate on it, then store it again, as with many CPUs. This all happens within the operation instruction, which works directly to and from memory.

This has lead to somewhat confusing nomenclature as memory will be referred to as a register, when it's a part of an operation and memory when it's holding values. In my opinion, this classification is not all that correct, but it's out there, so I'm including it here, so that you may understand what people are referring to when they use 'register' in this fashion.

The Propeller does have **internal registers**, in the sense of the word expressed above. The program counter is one such register. As discussed before, it's job is to maintain a reference to the next instruction to be executed. There are other internal registers that perform operations, shifting, etc... These are not directly addressable in most cases however.

Another use of the term register refers to specific CPU functionality being exposed as memory address space. This is called a **memory mapped register**. Instead of being just RAM or ROM, it's an active memory location. If one writes values to, or reads values from a memory mapped register, one is communicating to and from other internal parts of the Propeller, not just making use of storage. The special COG memory registers are a good example of this behavior.

Assembly language programs run in COGs. The COG memory is addressed as 512 longs. Each COG long can be an instruction, memory storage location, or register for use in an operation. What you call them is more about the context of what you happen to be doing, than it is anything else.

I refer to the Propeller as a memory to memory design. This is because it really does operate to and from memory instead of from memory to a register, operate, then from register back to memory, in terms of the instructions we give it. It's a simpler way to go, and very efficient as well I've no real idea if that is an official term. It's just how I choose to differentiate CPUs.

Pre-Defined Constants

These are labels that point to the addresses of things. They work a lot like an instruction mnemonic does. CNT = $1f1, for example. Here is a table of some constants that identify **COG memory registers**. These are presented for context only, you really only need to

know what each constant does for you, going forward.

```
PAR  $1f0      Read-Only Boot Parameter
CNT  $1f1      Read-Only System Counter
INA  $1f2      Read-Only Input States for P31 - P0
INB  $1f3      Read-Only Input States for P63- P321
OUTA $1f4      Read/Write Output States for P31 - P0
OUTB $1f5      Read/Write Output States for P63 – P321
DIRA $1f6      Read/Write Direction States for P31 - P0
DIRB $1f7      Read/Write Direction States for P63 - P321
CTRA $1f8      Read/Write Counter A Control
CTRB $1f9      Read/Write Counter B Control
FRQA $1fA      Read/Write Counter A Frequency
FRQB $1fB      Read/Write Counter B Frequency
PHSA $1fC      Read/Write Counter A Phase
PHSB $1fD      Read/Write Counter B Phase
VCFG $1fE      Read/Write Video Configuration
VSCL $1ff      Read/Write Video Scale
```

Software Setup

It's time to load some software. GEAR will be used for some explanations presented here. Other examples will do real things that can be seen on running Propeller hardware.

Propeller Hardware Setups Differ!

This is both a good and bad thing. Flexibility is good, and the Propeller really excels at this. It is possible to have a Propeller, up and running, on just a breadboard, battery and small number of discrete components. I happen to be running a HYDRA right now, meaning the code will be setup for a HYDRA system. Other setups range from complex custom boards to the Parallax Reference hardware.

This does not mean you can't work through this stuff, only that you may have to make a few edits to some examples, in order to get them working properly on whatever hardware you may have. Depending on what ends up being a part of this introduction to Assembly Language, some conversion hints may be warranted.

Conversions between propeller setups largely involve changes to pin specifications, clock speed parameters and other similar things. Core code changes normally are not required, given both setups have the same hardware components attached. The examples in this guide will stick to more or less standard things.

The assumption is made that you understand some basic electronics. If this is not the case, I highly recommend obtaining an already assembled Propeller system.

GEAR (The Propeller Debugger / Emulator)

You may or may not have a working Propeller environment. GEAR is an excellent learning tool, that can be used together with the Propeller IDE to run many Propeller programs. GEAR is a simulation that runs on your NET 2.0 capable PC. It's speed is nowhere near that of a real Propeller, but does offer memory views, TV and VGA screen emulation, etc...

Working environment or no, I highly recommend downloading this tool. Setup is really easy. Just unpack GEAR into a directory of your choosing, and run the executable from there. I like to put GEAR into one of my Parallax Tool sub folders and work that way.

GEAR can be found at:
http://soft.java-virtual-machine.net/virtual-machines/gear-parallax-propeller-debugger.html

You want the executable package, unless you plan on modifications to GEAR itself:
Gear-1.11.0.0.zip

GEAR output will be used in portions of this guide, and will continue to serve as a debug

environment going forward as well.

Build your Setup

It is a good idea to verify your hardware setup is working properly. Trouble shooting this is beyond the scope of this guide. I refer you to the friendly and helpful and active group of Propeller users and enthusiasts found at http://forums.parallax.com Register for an account, ask some questions, and get your setup running. Ideally, make a few friends along the way. Should you get up and running, consider returning the favor in the future.

Here is a quick software setup sanity check, using some of the example code that comes with the Propeller IDE.

1. **Download the Installer packages** for both the Propeller IDE and GEAR. The Propeller IDE comes with the Propeller Manual and it contains an excellent tutorial. GEAR has no such reference document, so I'll go through some basics here, as warranted.

   The Propeller IDE can be found at:
   http://www.parallax.com/propeller/downloads.asp

2. **Install the Propeller Tool**, taking the defaults. The only reason for not doing this, would be lack of disk space on your system disk (normally C:), or a preference for where programs are stored. This guide will assume the defaults, so translate accordingly, should you make other choices!

   Propeller Tool Default Information:
   The main, or root, program directory is:
   C:\Program Files\Parallax Inc\Propeller Tool v1.05.5

   I like to place new project folders under here in the 'Examples' directory.
   ...\Examples\Your Project Folder
   ...\Examples\Gear

3. **Create the GEAR folder** in the Examples directory as shown above

4. **Open the compressed GEAR file**. Therein, you will find a release folder. **Copy / extract the contents of this folder into the GEAR folder** created in step 3 above.

5. **Run the Propeller Tool**
   It should come up with a blank program work area tab active, 'Untitled1' that occupies the right hand portion of the Propeller Tool window.

6. **Run GEAR** by double clicking on **Gear.exe**
   (This takes a moment to start)
   You should see an empty window titled, “GEAR: Parallax Propeller Emulator”

Verify your Setup

1. **Start the Propeller Tool**

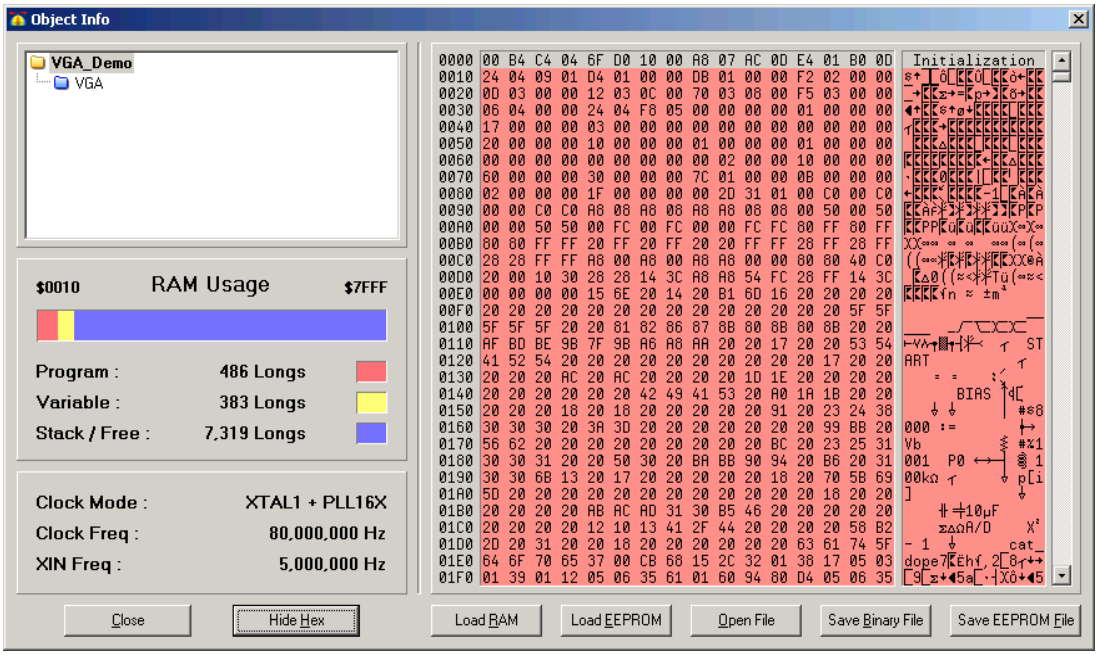
2. **Using the file tree on the left, navigate to** the ...\examples\Library directory and double click on **VGA_Demo.spin** program file you find there.

   You should see the program load into the main window.

3. **Hit the F8 key**

   This will compile the program and present you with a stats dialog.

4. On that dialog, **select the 'show hex' button** to arrive at the screen shown below:



5. **Select 'Save EEPROM File'** and navigate **to where you put the GEAR executable files** and hit the save button, modify the file name if you want, then complete the save.

6. **Run GEAR**

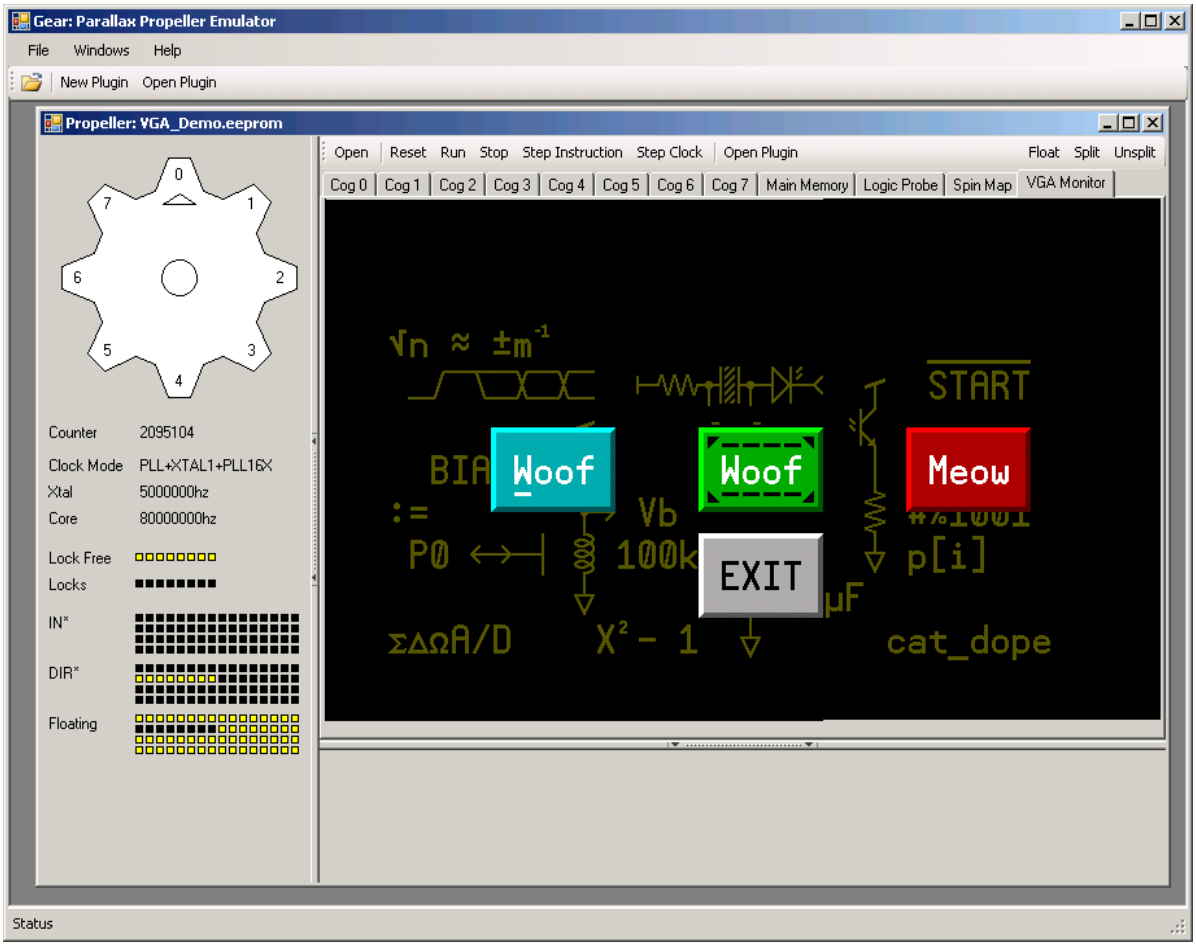
7. **From the upper most program menu, choose file, open** and select the file you just saved.

The file should open inside the GEAR window, showing you some code in the main window, the picture of the HUB and some menu and tab options.

8. **From the program window menu, not the main GEAR menus** or buttons, **select "Open Plugin"** and load **vgamonitor.xml**

9. **A new tab will appear, titled "VGA Monitor" Select it, then hit "Run"** from the program window.

   Graphics should appear, eventually rendering a screen that looks like the one shown below:



All of these things can happen with or without an actual Propeller chip being connected to your computer, leaving you free to accomplish that however makes best sense for you.

If you have a working setup, feel free to use it! This information is provided in the spirit of speaking to the very beginner.

Once you get the Propeller Tool and GEAR up and running, feel free to run some of the demos provided. You can start, stop and step the propeller, examine memory, see states of locks, where HUB access is currently at, etc... Spend a little time with the GEAR tool to understand what it does.

The EEPROM file is a binary that equals what would be in the Propeller RAM, had it been loaded by the Boot Loader. GEAR simulates the Propeller very well from there. GEAR has a few bugs, none of which should significantly impact the material presented here. Many have used GEAR to simulate complex Propeller programs.

When stepping through instructions, there is a lot of steps to go through when a new COG starts. At times, it may be necessary to just run something in GEAR, then look at the end result. The newest versions of Gear do support breakpoints, where execution can be stopped at a particular instruction. At this time, the basic example here does not cover the use of a breakpoint. Be sure and check the gear documentation.

Gear acts like a running Propeller in software. It's speed is less than 1/10 that of a real Propeller, running at 80Mhz.

The slow build of the graphics is actually a look at what the monitor does, line by line. This is why you won't see a full image at first. It takes some time for a few frames to draw, thus completing the image.

On my system, this takes a quarter of a minute.

Dissecting some Assembly Language Code

This really is a mixture of SPIN code and Assembly language code. The SPIN discussion will be kept light, but is necessary as all assembly language programs will have a small SPIN component. This example has the minimum amount of SPIN language required to get an assembly language program up and running.

The example code, shown below, toggles a specific pin on and off, endlessly, with a specific amount of time between toggles. The parameters given below, assume the reader will be using GEAR to watch the output. If real hardware is being used, longer delay times might make better sense. Try 6_000_000 for a delay, if you are using a blinking LED, for example.

The page break here is just for the sample program to appear in one, unbroken segment for clarity.

And now, the code: (finally!)

```
{{ AssemblyToggle.spin }}

'From Page 340 of the Propeller Manual
'With some small edits, for this purpose!
'opengeek@gmail.com

CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


PUB Main
{Launch cog to toggle P16 endlessly}

  cognew(@Toggle, 0) 'Launch new cog


DAT
{Toggle P16}

                    ORG 0                 'Begin at Cog RAM addr 0
Toggle              mov dira, Pin         'Set Pin to output
                    mov Time, cnt         'Calculate delay time
                    add Time, #$f         'Set initial delay here
:loop               waitcnt Time, Delay   'Wait
                    xor outa, Pin         'Toggle Pin
                    jmp #:loop            'Loop endlessly


Pin     long |< 16                        'Pin number
Delay   long 600                          'Clock cycles to delay

Time    res 1                             'System Counter Workspace
```

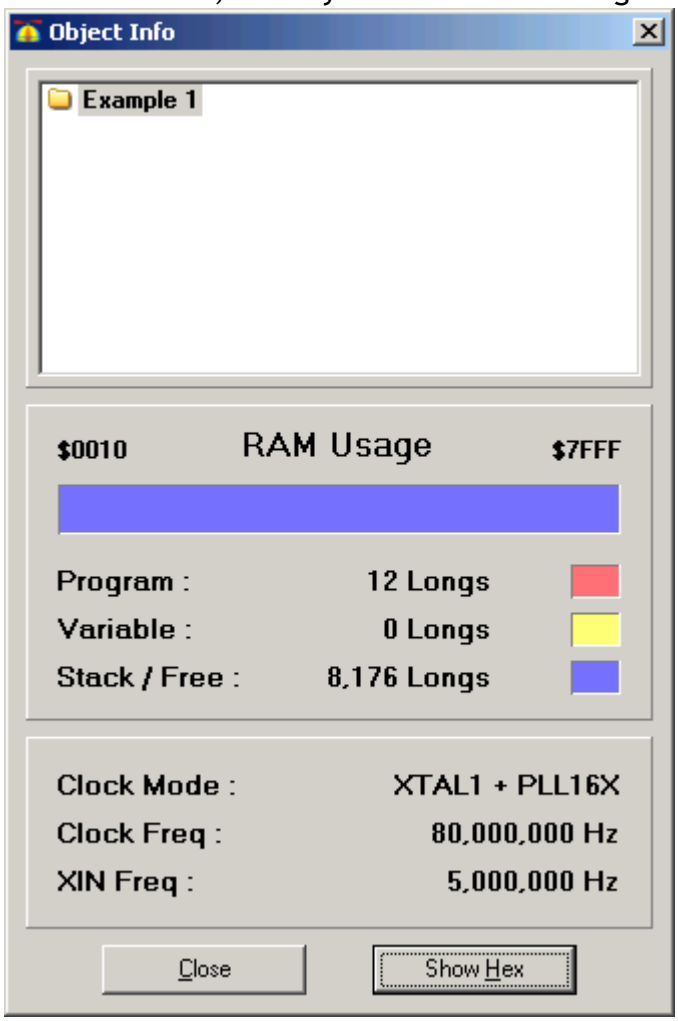
Getting the code into the Propeller Tool

Before anything else is discussed, it's a good idea to make sure this code is running properly. To do this, you can type it into the Propeller Tool. I highly recommend doing this, as you will get familiar with both the Propeller Tool and formatting at the same time. It's also good for exercising your memory.

Most adults learn best with a multi-sensory approach. This means read it, say it, write it, type it, watch it. That's old school, but I know it works. My day job involves adult learning on a regular basis.

The other alternative is to cut and paste the code above into the propeller tool.

Either way is fine, just get it in there. You can check your work, if typing, by using the F8 key. This will cause the Propeller Tool to evaluate the code, and try and build a working binary image of it.



If you see a screen like the one shown at right, everything is looking good for the GEAR simulation to follow.

If not, review what you have input, and try again.

This info screen tells you the size of your program, relative to the Propeller resources.

Only 12 longs are used for this program! That's a mere 48 bytes! Assembly language programs typically are very small compared to programs written in other forms.

Remember to hit the “Show_Hex” button and save your EEPROM file for use in GEAR.

If you are running real hardware, you can edit the pin number, make sure your LED, or other device is connected to that pin, and hit F10 to run the program directly from the Propeller RAM.

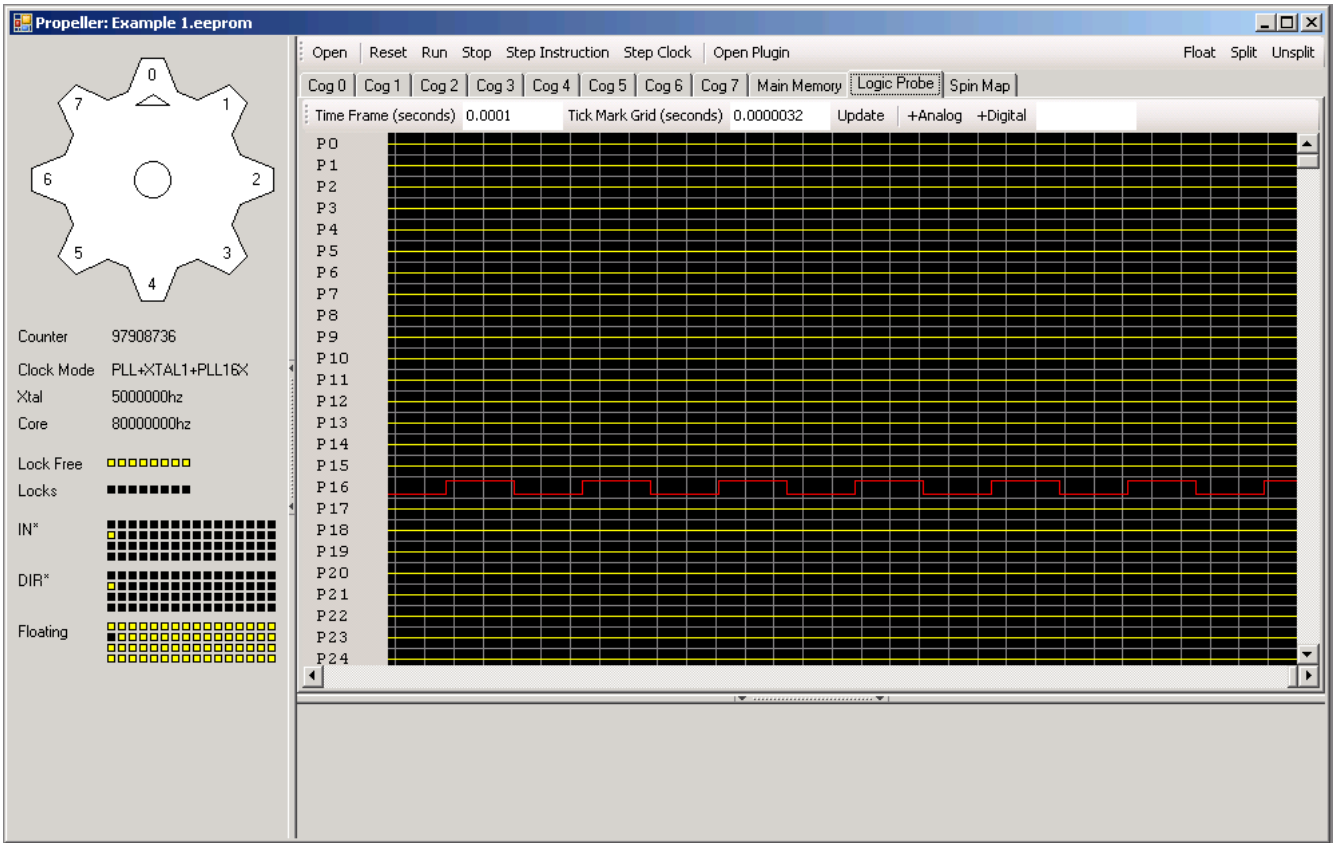
Gear Simulation

It is necessary to make sure this program is running properly before any further discussions, changes, additions happen.

Again, hit F8 to arrive at the Info screen. This will catch any gross errors in formatting and syntax. Select the “Show Hex” button, followed by the “Save EEPROM File” button to save the memory image GEAR needs. It's not a bad idea to just do all of this in the GEAR folder to avoid having to re-select directories.

Run Gear and open the EEPROM image you just created.

Select “Run”, from the window that appears, then select the logic view tab to see the pin toggle in the simulation. Your screen should look something like the one below:



As shown pin 16 is colored red, for output. At the left, you can also see the pin state registers and the state of pin 16 there too.

Hit the stop button, to freeze the output, run to continue and step instruction to go through it one instruction at a time. There is also step clock. Comparing these two is useful. Step instruction directs GEAR to just simulate what it takes to get to the next instruction state. Step clock means just advance the simulation one clock at a time.

Stepping through this with the clock is a lot of clicks! We have a delay of 600 clocks per pin toggle. That's actually not very long at 80Mhz, but seems very long, one clock at a time.

Another solution for this dilemma is a called a breakpoint. It's a pre-defined place in the program where it can stop running, while being examined. GEAR does not have this facility, which will limit some of the discussion examples.

Program Sections

The Propeller IDE highlights the different program sections in color. Each section has a specific purpose, detailed below:

AN IDE stands for "Integrated Development Environment" This is a short way of referring to the software tool, designed to work with the Propeller.

Formatting

The first thing we see is lots of formatting elements. Truth is, the Propeller IDE and the computer itself does not care about any of this stuff. It's for us humans, who need lots of context, if anything is to make any real sense.

It is a really good idea to get in the habit of formatting your code you write. Might seem like an extra hassle right now, but soon it will become habit and you will thank yourself when going back to look at code you wrote years ago. Others will thank you too.

I've colored the first example in a fashion similar to what the Propeller Tool does. Going forward, plain text will be used.

(unlabeled) Pre-program comment area

Some people refer to this as the **Program Header.** If you have a particular program license in mind, want to express credit for other contributors code, or maybe just want to explain what the program does, it's inputs, outputs, etc... this is where you do it. Comments can take many forms. A few are detailed here in this section.

```
{{ AssemblyToggle.spin }}
'From Page 340 of the Propeller Manual
'With some small edits, for this purpose!
'opengeek@gmail.com
```

Comments

Single quote comments " ' "

This one is quick and easy. Essentially, anything to the right of the single quote character "'" is then considered a comment.

Comments are explanatory in nature. They will not be considered as part of the program at all. Comments are used by programmers to communicate intent, so that it is not forgotten over time, or must be re-parsed.

Comments may appear as part of program lines. Look at the DAT section below this one, for some examples of that.

Bracket comments " { } "

Large comments may be defined with the open bracket “{“ and close bracket “}” characters thus:

```
{
     If a comment is substantial, perhaps contains a table,
etc...  the bracket method is a far better option.  Here
is an example table:

     parameter_array[0] – Address of something
     parameter_array[2] – Address of something else

Finally, we end the large comment with a close bracket!

}
```

Bracket comments are also a quick way to exclude segments of code from the program.

CON (Constants)

```
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000
```

You can define constants here. A constant is a value assigned to a text label. Eg: PI = 3.14159 This is done to make life easy. Instead of repeating the value every where it is needed, one just repeats the label. Additionally, changing the label definition, then makes that change global –meaning everywhere that label appears. This is powerful in that making lots of manual changes gets considerably easier when constants are used.

Constants do not consume program space in and of themselves. They are for you to communicate meaning to the Propeller Tool, or other assembler program.

Constants do take program space when specified in the main program. They take the same amount of space just putting the value there would. There are three program lines in this section.

Constants are operated on by the Propeller tool. They are not run time program elements. While it is possible to perform math operations with constants, the end result is a static value to be used by the running program. When used this way, the idea is to make pre-calculating some values needed by the program available to it as results only.

These constants define the clock mode, source and frequency the system clock is derived from. More detail on this can be found in the Propeller Manual, starting on Page 180.

PUB (Public)

```
PUB Main
{Launch cog to toggle P16 endlessly}

  cognew(@Toggle, 0) 'Launch new cog
```

SPIN programs are broken down into small components, called procedures. Any such procedures in the PUB area, are exposed to other SPIN programs. (There is also a PRI, meaning private, area not shown in this example. Those procedures would not be shared.) In this case, the procedure name is **Main**, and is the only procedure. Even though it is small, there is a lot to understand here.

Procedure Main only has one SPIN instruction! That's the **cognew** command, shown in bold above. This line tells the Propeller to load and start running another COG with the assembly language program, located in HUB memory by the label Toggle. The '@" operator means "pointer to", and will be discussed in more detail shortly. For now, "@Toggle" means the location in HUB memory where the assembly program begins.

Cognew has two arguments being supplied here. One is the HUB memory location of the program to be loaded into the COG. The other is a single value, to be passed to the program running inside the COG. That's the ", 0" portion. This particular program does not require a passed value, so anything actually works here, but a 0 is typically used when nothing needs to be passed to the COG.

It is worth noting that COG 0 is actually running the SPIN procedure main. Another COG will end up running the assembly language program, to be started by COG 0. This is a significant point to understand!

So, how does this all flow together anyway?

When the Propeller Tool sees this program example, it parses the program (with parse meaning: "To break down to core elements of meaning") to build up all the binary instruction code, interpreted SPIN code, data and other program elements, such as memory arrays, initialized values, etc...

It then builds up a HUB memory image containing all of these things, fit together in an organized way. That HUB memory image contains all of the executable program code in this example. That's the 48 bytes seen on the F8 screen above.

When the Propeller starts, it looks for a program image to fill it's RAM with. That's actually identical to the EEPROM images we've created to run with GEAR. Having filled up the RAM, it then loads the SPIN interpreter, from it's internal ROM, into COG 0, which then gets started running the SPIN program it finds in HUB memory. That's how Main is eventually executed.

The SPIN interpreter sees the cognew instruction, in the procedure Main, and also knows

the address associated with the label “Toggle”. It takes that address and directs the Propeller to start up a new COG, in this case COG 1.

At this point, the assembly language program gets copied from the HUB, to COG 1. Really, there are two copies of the assembly language program. One is the HUB, and is necessary to get a COG up and running. The other is actually in the COG, running!

COG's need to fill their program space with whatever it is they are going to be doing, before they actually run anything. That's 2Kb of RAM, 512 Longs, per COG. This happens even if the actual assembly language is very small.

At this point, COG 0 is essentially done. The only task it was given was to load up COG 1, and that's it.

The assembly program then exists in COG 1, and begins to run. That's the next program section!

DAT (Data)

This is where the assembly program lives. Lots of things can go into this section, but I'm going to focus on the assembly language program for now.

```
DAT
```

The first line, the DAT line, defines the start of a DAT section. Unlike the Highlander, there can be more than one! (That is to say, you may have more than one instance of the program section types.)

```
{Toggle P16}
```

This is a comment line, detailing the idea that we are toggling pin number 16.

```
                        ORG 0 'Begin at Cog RAM addr 0
```

ORG 0 essentially means, “Start of COG Memory”, which is location 0. This helps the Propeller Tool see we are telling it about an assembly language program and not some other data. The Propeller tool then can lay out this program in terms of HUB memory for loading it into the Propeller, while also encoding the instructions for execution in the COG memory, when that time comes.

If there was another assembly language program to be run in another COG, another ORG 0 statement and another label, like “Toggle” would be all that is needed to set that program up.

Toggle then has two meanings!

In the HUB, it's some memory location in the HUB memory space $0000 to

$7fff, such as $002E. (That's not the real location, just an example of low HUB memory.) That meaning is necessary to insure the COG is loaded with the right program data residing in the HUB.

In the COG, it's actually address 0, or the beginning of COG memory. Remember, all COG memory is addressed as longs. For assembly language this makes things really easy.

The first instruction, in this case, is location 0. The second one will be location 1, and so on... [shown below]

```
Toggle        [0]         mov dira, Pin 'Set Pin to output
              [1]         mov Time, cnt 'Calculate delay time
              [2]         add Time, #$f 'Set minimum delay here
:loop         [3]         waitcnt Time, Delay 'Wait
              [4]         xor outa, Pin 'Toggle Pin
              [5]         jmp #:loop 'Loop endlessly


Pin       [6]       long |< 16      'Pin number
Delay     [7]       long 600        'Clock cycles to delay

Time      [8]       res 1           'System Counter Workspace
```

A two COG example

Let's say we wanted to toggle two pins, instead of just one. There are 8 COGs in the Propeller, why not just fire up another COG? Right now, we know almost enough to do just that.

Here's the example code for two COG's. Get it into the Propeller Tool, and simulate it in GEAR. The different program elements are shown in bold.

```
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


PUB Main
{Launch cog to toggle P16 endlessly}

  cognew(@Toggle, 0) 'Launch new cog
  cognew(@Toggle1, 0) 'Launch another new cog

DAT
{Toggle P16}
```

```
                ORG 0                   'Begin at Cog RAM addr 0
Toggle          mov dira, Pin           'Set Pin to output
                mov Time, cnt           'Calculate delay time
                add Time, #$f           'Set minimum delay here
:loop           waitcnt Time, Delay     'Wait
                xor outa, Pin           'Toggle Pin
                jmp #:loop              'Loop endlessly


Pin     long |< 16              'Pin number
Delay   long 600                'Clock cycles to delay

Time    res 1 'System Counter Workspace


{Toggle P17 ------------------------------------------------}

                ORG 0                   'Begin at Cog RAM addr 0
Toggle1         mov dira, Pin           'Set Pin to output
                mov Time, cnt           'Calculate delay time
                add Time, #$f           'Set minimum delay here
:loop           waitcnt Time1, Delay1   'Wait
                xor outa, Pin1          'Toggle Pin
                jmp #:loop              'Loop endlessly


Pin1     long |< 17             'Pin number
Delay1   long 400               'Clock cycles to delay

Time1    res 1                  'System Counter Workspace
```

There are two ways to do this. The better way is to write a smart enough assembly language program, so that one copy of it can be in the HUB memory. This is done with passed parameters (arguments, pointers...), such that each time it is started up on a COG, it knows to look for some information that differentiates what it is doing from what it's other instances are doing on their COGs.

Then there is the easy way, shown here. Really all that happened was a cut 'n paste of the original assembly language program. From there, new labels were created by adding a 1 onto all of the existing ones, finally some comments were changed, along with the pin and delay values.

Back up in Main, another cognew command starts up COG 2 with the second assembly language program.

The only real new piece of information is that labels must be unique! That's why the ones were added onto the ends of the labels. The Propeller Tool must look through all the code, parse it and organize it. If it sees two of a label, it has no way to differentiate them like we do.

There are additional rules for labels, well covered in the Propeller Manual.

Hopefully, this example reinforces HUB memory and COG memory, labels, and such...

From here, it's onto parsing the assembly code.

Assembly Program Detail

I've reproduced the assembly program here again for clarity. For now, it's back to the single COG example. The instruction addresses are in brackets. One thing to remember is that each instruction is a long, so is a register, or memory storage location. It's all longs, addressed one long at a time. *Really, any given COG memory address is what you say it is. As long as you are clear and consistent with your purpose, for each location, you will not have problems with this.*

2Kbytes of COG memory then breaks down into 512 – 16 special purpose registers, of addressable COG memory. ($000 - $1EF)

This section will focus on the instructions used. Many of the instructions have a lot of variations that will be ignored, in the hopes that just detailing how they are used in this particular program ends up being helpful. Having learned about specific uses, one can then more easily explore other options! It is not a bad idea to crack open the Propeller Manual and read about the different instruction options, once you have their purpose, presented here, clear in your mind.

Discussion

Program

The first 6 COG memory locations hold the program. In the context of the, “What is a register?” discussion earlier, these memory locations are just storage right now. They hold program instructions, to be differentiated from the data consumed / manipulated by the program. They are not really 'registers' in the most often used sense of the word.

That's really the only differentiator between software (programs) and data. It can get somewhat bizarre with programs consuming other programs, or a program changing itself. However, if you are there asking those questions, you've progressed past needing this guide! Congratulations.

The point to remember here is that any COG memory location can play any of these roles. How a memory storage location is used really is up to the programmer, with the rest just being semantics.

Data

The last three then are data 'registers', in that they hold values to be used by the program, and one of them [8], defines a workspace used by the program.

There are two differentiators here:

Initialized data

These are values that are loaded with the program. Think of them as givens, from which a greater solution or task may be completed.

Uninitialized

Essentially, working memory locations. The program will use these for in-process storage. Working data, in other words. Their starting value is not important.

Again, these are not hard and fast rules. For example, one could define a memory location with some starter data, then decide to use it for temporary storage later, once the need for the initial data has passed. Again, semantics. The key to this stuff is to follow the intent of the program. From there, these differences will wash out, leaving you knowing what is actually happening in the memory. From there, call it what you want to call it!

```
Toggle        [0]         mov dira, Pin 'Set Pin to output
              [1]         mov Time, cnt 'Calculate delay time
              [2]         add Time, #$f 'Set minimum delay here
:loop         [3]         waitcnt Time, Delay 'Wait
              [4]         xor outa, Pin 'Toggle Pin
              [5]         jmp #:loop 'Loop endlessly


Pin       [6]       long |< 16     'Pin number
Delay     [7]       long 600       'Clock cycles to delay

Time      [8]       res 1          'System Counter Workspace
```

Instruction [0] mov [destination COG address], [source COG address]

```
Toggle        [0]         mov dira, Pin
```

The **mov** instruction copies the contents of one COG memory location to another one. The end result of a mov will be two memory locations having the same content. That's it! Pretty easy overall. It's ok to think of mov as actually being copy, because that is exactly what it does.

So, which locations are being moved and why?

That's what the labels are for. In this case, we've got the '**Pin**' label referring to COG memory location number 6. That memory location is initialized with the value necessary to identify pin 16 for some operation. (That will get detailed under that instruction header)

All we need to know, at this stage, is that whatever is contained in memory location [6] will do the job of identifying pin 16.

The pre-defined label **dira**, refers to one of the special COG memory registers. These are active memory locations, whose contents impact the behavior of the Propeller itself. The dira register defines the input / output direction for the first 32 i/o pins. In simple terms, each pin needs a direction. When more than one COG is working with a pin, it gets more complex than that however.

(See the Propeller Manual, page 23 for more on the matter of pin states.)

In this example, we've got one COG, working with one PIN, so a zero means that pin is an input pin to be read by running programs. A one means that pin is an output pin, whose state is changed by running programs.

To toggle the pin, we need to change it's state, so it's gotta be set to output.

That is exactly what this first mov instruction does. It sets the direction of pin numbered 16 to be in the output direction.

Instruction [1] mov [destination COG address], [source COG address]

```
              [1]         mov Time, cnt
```

Another mov instruction, with the same form as the first one. We know something is getting copied, but what? Again, the labels tell the story here.

Predefined label **cnt** refers to the active COG memory register that refers to the global system counter, shared by all the COGs. Global means literally, “everywhere”, and shared means “readable by everything”.

The label Time, is the working register. It's purpose is to provide a place for computations, intermediate results, etc... Whatever the counter value is, at the time this instruction is executed, will be copied into the COG memory location the label Time refers to.

This instruction then notes the value of the system counter, for use later.

Instruction [2] add [destination operand COG address], [#immediate source value operand to be added]

```
              [2]         add Time, #$f
```

Here we've got both a new instruction, and a new addressing form! (That's the pound sign! " # ") The new instruction is an add instruction. It works like this:

The unsigned sum of both operands will end up in the COG memory register specified as the destination.

The source value is interesting because it is contained in the instruction itself, not a COG memory location! This deserves some explanation, before we move on.

Time = Time + $f

Immediate, or literal addressing

An immediate address is essentially one where the instruction itself has the operand, or data to be operated on, within it, and not contained somewhere else in the COG memory. Propeller instructions have two 9 bit fields, called the destination and source fields. Normally, these refer to any of the 512 longs in the COG.

As an option, the programmer can just choose to put a value into the instruction, thus saving a COG memory location for something else. Immediate addressing is only available for the source field (the second one), not the destination field.

That's what is happening here. Small values (0 - $1ff) can be immediate values, where it makes sense. Larger ones won't fit into instructions and must be addressed in the usual way; namely, by reference to their COG memory address, as in the mov instruction above.

What we have here then is the contents of the COG memory location, referred to by the label Time, gets added to the immediate value $F, with the result being stored back to the COG memory the label Time points to. In this case, that happens to be memory location number [8].

This instruction then adds a short amount of additional time to the system count. This is just enough time to get through the rest of the initial instructions, without missing the counter! More instructions = more initial delay time. Not enough delay time and the program has to wait for the counter to wraparound. On an 80 mhz system, this takes a little under one full minute.

Instruction [3] waitcnt [cog memory location containing counter value to watch for] [cog memory location containing delay to be added in preparation for next watch]

```
:loop           [3]          waitcnt Time, Delay
```

This instruction also has a label, ":loop". The label means we are going to be coming back to this instruction later on. Rather than calculate the COG memory address to return to,

it's a whole lot easier to just slap a label on the return point and refer to it later. This is what ":loop" is all about here. It is a destination label for a jump instruction somewhere else in the program. **The two together form a program loop.**

The **waitcnt** instruction pauses the COG program, until the system counter value matches the COG memory location specified in the first part of the instruction. In this case, that's the one pointed to by the label Time. So we've got:

Pause until cnt = Time

Additionally, every instruction takes some time to execute. At 80 Mhz, this is 50ns of time actually. This is why a small value was added to Time in the instruction above. It is necessary to account for instruction execution time when setting up a waitcnt.

If this is not done properly, the waitcnt instruction is executed after the counter and COG memory values happen to match up. The program then will pause for some 53 seconds, with an 80Mhz clock, waiting for the counter to wraparound and then match up a second time.

This is a propeller red flag, BTW. If you've got a program that takes about a minute to do anything, you've hosed up a waitcnt for sure.

That's not all for this instruction though. Really, this is like two instructions! The first one is all about waiting until the counter matches some COG memory location. Once that match has occurred, the second part of the instruction comes into play.

Now, the COG memory location referred to by the label **Delta**, gets added to the contents of COG memory location Time.

Time = Time + Delta

Two instructions in one! The cog stops and waits for the system counter to reach a known value, then adds some other value to the value being watched for, then continues on.

Instruction [4] xor [destination COG memory address] [source COG memory address]

```
              [4]         xor outa, Pin 'Toggle Pin
```

Now it's time to take a digression. To understand what this instruction does, we've got to explore the bit operators. These are logical operations that can be applied to numbers the same way math operations are applied.

The xor instruction works like the add instruction does. The source will be operated on together with the destination, with the result of the operation being written to the destination. In this case, the source is COG memory associated with the label **Pin**. The

destination is another pre-defined label that points to the register controlling the Pin states. Since pin 16 is defined as an output pin, this instruction then affects it's state, high or low, depending on the logic contained in the bit operator, which is xor (exclusive or).

If you don't know what bit logic operators are, consider reading that section below; otherwise it's on to the last program instruction.

Instruction [5] jmp [#immediate COG memory location to jump to]

```
        [5]     jmp  #:loop
```

This one is really simple. It's a lot like the mov instruction in that some values get copied around. The immediate addressing mode is used, so everything about this instruction is contained in the instruction. Additionally, there is only one data field specified!

The jmp instruction copies a number into the Propeller program counter! That's it. Once the Program Counter has been changed, program execution continues at the new location.

Here we see the label loop being used in an immediate mode. We know that loop resolves to the COG memory location #3. So, a jmp #3 would work here nicely. Having the label is handy however. If additional instructions were placed before the :loop label, it's absolute memory location would change, and that would mean changing the jmp too.

Nobody wants to worry about that, so the Propeller Tool evaluates loop, then puts that value into the jmp, at the time the program is prepared for loading onto the Propeller.

Note: If the pound sign is missing, then the contents of COG memory address #3 would then contain the value to be loaded into the Program Counter! That would not have the desired effect at all, since location number 3 is an instruction, it's going to point the Program Counter somewhere other than where we want it to be.

In this case, that would be outside the program, meaning we've lost control of the COG! It's going to just continue executing whatever it finds in it's memory oblivious to anything we intend for it to do.

I call this "Happy Fun Memory Land"!

If we wanted to jmp this way, we would have to load a COG memory location with the value of :loop, then the jmp without the pound sign would work as intended. That's extra work, not required in this case. That is also called an indirect address, where a memory location contains an address. Important for subroutines, and other structured jumps.

Program Discussion

That's it. We've more or less parsed through this program. We now know it does the following things over and over:

(in pseudo english form)

-Spin procedure “Main” launches assembly program in COG 1

- set the direction state of pin 16 to be output
-look at the system counter and store it in the working memory location
-add a small minimum, “get things started” delay time to it
:loop -wait for the system counter to catch up, when it does, add delay time
-toggle the pin
-go back to the task identified by the label **:loop**

Additionally, there are things we can change. Those things are the immediate value used for the initial delay. That's the **#$f** in instruction number [2]. The pin to be toggled can be changed by redefining the initialized value, identified by the label **Pin**, which currently is memory location [6]. Finally, the delay between toggles is contained in COG memory location [7], associated with label **Delay**.

Finally, some understanding about the level and scope of the task breakdown should be understood at this point. Each instruction has a well defined contribution to the task as a whole.

I recommend some time spent making these changes, along with some time watching the results in GEAR, or better, on your Propeller hardware setup. If you want, use the two COG example so you have something to compare with.

This is also a great template program, for trying other instructions out. Some suggestions are:

1. Change the pattern of the pin toggle. Have it be a repeatable sequence, long, short, long, long...

This can be done with a longer loop and some additional waitcnt instructions.

Eg:

```
    waitcnt Time, Delay
    xor outa, Pin
    waitcnt Time, Delay1
    xor outa, Pin
```

2. Toggle more than one pin. Apply the bit operation info presented to build different pin masks and operate on them.

Bit Logic and Manipulation Operators

Near the beginning of this document, the idea that computers only do two core things was introduced. That's somewhat of an oversimplification. There is one additional core area of operation to be covered in this section. These operators could be included near the beginning with the math. Maybe they should. The balance for new information was weighed with actually getting through an assembly language program. Having done that, it's time to really explore these powerful tools. They are as essential as the math is.

Computers process logic in the same way they process math. In fact, the two ideas are closely related as will be explained below. When two binary numbers are combined, or operated on, rules are applied. So far, the math oriented rules have been presented. In this section, logic rules will be presented.

Each operator will be covered. There are some new concepts to be learned in this section. They will be introduced in context, in a fashion similar to how the math section is done. The operator, “OR” is used to introduce source, destination, etc... Other operators will assume that material is understood. Additionally, where the operators apply to the example program, that will be discussed in context as well, with the goal of a conversational style that ties the whole together, albeit in a somewhat less organized fashion.

Truth Tables

A truth table is simply a listing of combinations and results for a given rule. For any set of binary numbers, the number of unique combinations is the sum of the powers of two it's number of digits represent, plus one because we count from one and not zero.

```
Eg:       %1111 = $0f (+1) = 16 = number of combinations
     %1111_1111 = $ff (+1) = 256
```

A single binary digit has two states, on and off, high and low, true (1), false (0), etc... Operating on two numbers then reveals the basic truth table size of four possible states, two for each digit involved.

Truth tables for the various logic operators will be given below, along with explanatory text. Truth tables break down the digit combinations, apply the rule and summarize the result as being either true (1) or false (0).

OR (Bitwise OR)

The rule for this one is simple. If either bit is set, or true, then the result is true, otherwise false. Here is the truth table:

```
    OR
-----------
1 or 1  = 1  <---Result of OR rule applied
1 or 0  = 1
0 or 1  = 1
0 or 0  = 0
```

Some examples are in order here, along with some possible use scenarios. Consider the binary numbers below:

```
          %11000011  <--- Operand 0          %0000     %1001
     OR   %10011001  <--- Operand 1       OR %1010  OR %0000
          ---------                         -----     -----
          %11011011  <--- Logical Result    %1010     %1001
```

This one is pretty easy! If there is a one anywhere, the result is one. The direction of the operands does not matter either. One is either performing an OR operation, or one is not. It's worth taking a look at how the propeller does it:

```
DAT                     org 0

            [0]         or  Result, Operand0
            [1]         or  Result, #%0001_1101

'Pre OR instruction COG memory contents:
Result        long   %00000000_00001111_00000000_00000000
Operand0      long   %00001100_10000001_00011100_00010000

{

Post instruction contents, shown for each instruction.

Result  [0]   long   %00001100_10001111_00011100_00010000
Result  [1]   long   %00001100_10001111_00011100_00011101

Note that Operand0 never changed!

}
```

For the moment, let's pretend this bit of code has just been loaded into a COG, and the Program Counter is at [0], pointing to the first or instruction. At this moment, both Result and Operand0 are initialized data, residing in the COG memory.

When instruction [0] executes, the contents of Operand0 are OR'ed together with the contents of Result. The result of this operation is written back to the COG memory associated with label Result. This can be seen in the comment section above.

Now the next OR instruction runs, but this time one of the operands is immediate. That's the pound sign " # ". That operand, %0001_1101 gets OR'ed with Result and the result of

that ends up back in the COG memory, shown in the comment section above [1].

On the Propeller, bit operation instructions follow this general convention:

```
OR  [Result Operand 1], [Source Operand 0]
```

Remember, the result of the operation will end up being stored where Operand 1 originally was. This is a destructive operation, in that when it is completed, Operand 1 no longer exists in it's original form! Having this behavior is actually a good thing, just be sure you understand it.

What about the Carry? (Parity)

Unlike the math operations, bitwise operations don't carry over. They operate on each pair of digits in a discrete fashion. The carry does serve a useful purpose however. After a bit operation, the number of ones in the result is computed.

This is called parity, and it's simply an expression of the number of digits containing a one, being either even or odd.

If they are even, that's a parity result of 0. If they are odd, then that's a parity result of 1.

eg:

```
%11000011 = Parity Even = 0    %10011001 = Parity Odd = 1
```

This result will go into the carry flag, unless you instruct the Propeller to ignore the operation parity with the “NC” condition.

```
OR   Result, #ff  NC
```

It is also possible to compute the parity only, and not store the result at all! Here is an instruction to do just that:

```
OR   Result, %1101_0011  NR
```

Parity is used all over the place to perform very simple error checking. If a stream of bits is moving from computer to computer, or even from one part of a computer to another, the chance exists that one or more of those bit states will end up changed in the process.

Quite simply, stuff happens.

Parity can catch these in a high number of cases, because the sum total of ones, being even or odd, will change with a single bit change. A parity computation then serves as a general sanity check.

What else can I do with OR?

The OR instruction is great for setting a specific bit to a set, true, or one state without impacting the other bits surrounding it. This is done with a **bit mask.**

Bit Masking

A bit mask is an operand setup with the idea of it being applied to another operand to achieve a specific and predictable result. It's often used when the patterns of digits to be affected vary widely, but the general result does not.

Without bit masks, most programs would grow to be very lengthly as many branches and storage locations would be required to handle varied digits and their manipulations.

In the example program above, a specific pin direction state was set to the output state. (1) What if another pin state needed to be set, and we were never sure what other states may have been set in other cogs, or even by other parts of the program itself?

In this case we need to set a single digit to a one, without impacting the other digits. This is an OR operation.

Another example is graphics. A typical graphics screen has a bitmap display. The binary numbers are read and transformed into pixels on the screen. Unless the number of bits per pixel is a byte, word or long, it is necessary to manipulate bits directly to set and reset pixels.

Let's say the video sub-system is operating in a mode where one bit is used per pixel...

```
DEST %00000000      ........   SOURCE %11111111  DEST ********
     %00111100      ..****..          %10000001       *.****.*
     %00011000      ...**...   OR     %10000001   =   *..**..*
     %00000000      ........          %11111111       ********


        OR  DEST, SOURCE
```

In the example above, the video screen source bytes contain an existing image on screen. The source bytes are somewhere else, and are to be drawn on top of whatever is already on the graphics screen. In this case, the OR instruction lets us do something that will only add pixels to the screen.

Other bit operations work like OR does. Essentially, there are two operands. One is the source, and is not changed by the instruction. The other is the destination, and is changed by the instruction, unless the programmer has set the NR effect to inhibit this from happening.

Other Truth Tables

XOR (Exclusive Or)

The exclusive or operation is the toggle, in the program above. It's truth table looks like this:

```
     XOR
-----------
1 xor 1  = 0  <---Result of XOR rule applied
1 xor 0  = 1
0 xor 1  = 1
0 xor 0  = 0
```

This rule can be summed up thus: If the first operand or the second operand, not both, is one, the result is one; otherwise the result is zero.

Consider this truth table in the context of the example program above. This is how a toggle is achieved with one instruction. The COG memory, identified by label Pin, contains the following binary bit mask:

%00000000_00000001_00000000_00000000

This number identifies pin 16, and is used to change it's state. The state of all the pins can be obtained in the **outa** register. If one reads this register, one gets the state of all the pins. If one writes to this register, one then changes said states.

The xor instruction:

```
              [4]         xor outa, Pin 'Toggle Pins
```

take the bit mask at Pin, for it's source, and operates on the pin states themselves as a destination. Apply the truth table to see the toggle happen thus:

```
        outa %xxxxxxxx_xxxxxxx0_xxxxxxxx_xxxxxxxx

 xor    Pin  %00000000_00000001_00000000_00000000

Result  outa %xxxxxxxx_xxxxxxx1_xxxxxxxx_xxxxxxxx


        outa %xxxxxxxx_xxxxxxx1_xxxxxxxx_xxxxxxxx

 xor    Pin  %00000000_00000001_00000000_00000000  (Source bit mask)

Result  outa %xxxxxxxx_xxxxxxx0_xxxxxxxx_xxxxxxxx  (Destination)
```

Notice the x in the results above? The x means, “doesn't matter”. Looking at the xor truth table, it can be seen that applying an xor operation, with a source operand digit of 0, does not change the destination operand ever. If a one is specified however, the destination operand state will be toggled to the opposite state.

Using a bit mask then localizes the impact of the xor operation to the bit controlling the state of ping 16.

If it was necessary to toggle pin 17 also, the only change would be to the Pin bit mask, leaving the rest of the program as is.

The “|< 16” is a shortcut that basically, sets a one, then shifts it over 16 times. The pin state could also be specified like this:

```
Pin       [6]       long %00000000_00000001_00000000_00000000
```

There is no difference. The short hand method, is computed by the Propeller IDE tool. The longer method is keyed in by the programmer. Either is fine, so long as the mask correctly identifies the desired pins to impact.

Suggestion: Modify the bit mask and view the result in GEAR.

AND (Bitwise AND)

Here's the truth table:

```
          AND
-----------
1 and 1  = 1  <---Result of AND rule applied
1 and 0  = 0
0 and 1  = 0
0 and 0  = 0
```

The and rule is very simple: The result is only a one, when both operands are one.

ANDN (Bitwise AND of one number, with NOT of another)

This instruction involves two truth tables, and is really two operations packed into one. A NOT operation is performed on the source operand, with the result of that being ANDed with the destination. It looks like this:

```
                    ANDN
-------------------------------------------------
         Source %00000000_00000001_00000000_00000000

NOT      Source %11111111_11111110_11111111_11111111
```

```
AND Destination %00101100_00111001_00010000_00000000

Destination --> %00101100_00111000_00010000_00000000(Result written)
```

NOT (Bitwise NOT)

Unlike the other operations, the NOT rule only accepts one operand. Here is the truth table:

```
     NOT
-----------
1 not = 0
0 not = 1
```

The rule for NOT is inverse. Make all the ones into zeros, and all the zeros into ones. That's it! Not is part of the Propeller instruction set. The same result can be obtained with XOR, using a source mask of all ones. ($FFFFFFFF) Not is listed here, as part of the explanation for the ANDN instruction, which performs a NOT on it's source operand.

These are the essential bit logic operators. Other instructions, such as mux, are combinations of these aimed at specific purposes and efficiency. They are covered in the Propeller Manual.

Assembly language programming, as seen now, is about these core things:

-understanding the hardware with a high degree of fidelity

can't control that which is not understood!

-breaking the task down into small elements, that leverage said understanding

if you have problems, you probably don't understand exactly what the hardware is really doing

-organization!

you can make as big of a mess as you want to -Keep It Simple Stupid [KISS] applies here as much as it does anywhere.

I

Comments, feedback, successes, failures, stories, all appreciated. From time to time, I roll these up into a new version of this document. Thank you for the excellent feed back so far.

Alphabetical Index

Addendum A

Parsing Assembly Language Programs

Like any programming language, there are times when it's good to go through other programs and understand what is happening when and why. There is a good starter process for this I'm going to detail here, in an appendix, largely because it's a topic that's relevant to the beginner, and because it just doesn't fit within the overall document flow so far.

Assembly language programs are differentiated from higher level ones, where parsing to understand is concerned, by their simple “building block” structure, and the need to consider the hardware functionality in the context of the program to a higher degree than is often required for higher level programs. Working through a program then consists of understanding what each instruction does, and why that matters given the hardware at hand.

The core elements of the Propeller itself are well documented in the manual and schematics. Because the scope of this document is all about core understanding of the software side of things, I'm going to limit the discussion on the hardware side accordingly.

This is a process that has always worked very well for me. There are others approaches to this and by all means consider and use them if they work for you.

Assembly language instructions all boil down to three core types:

Data movement

The data movement types essentially get data from one place to another. During the course of program execution, values end up in various places as various times. These are the instructions that get that done.

Operators

Essentially, these instructions change bits. Some examples of operators are add, subtract, bitwise OR, AND, shift left, shift right, rotate left, rotate right, etc...

Program Flow

It's rare that a program is just a linear progression of instructions. Often things need to be done multiple times, and the sequence of events necessary to complete the task will vary depending on the state of the data too. Program flow instructions handle these aspect of things. They also help with the overall structure of things, where there is program and data elements present, program flow instructions then keep the flow of execution in the program areas. Nobody wants to execute their data, unless they are building self-modifying code!

(In that case, the data to be manipulated is part of the program itself. This happens on the propeller often for pointers.)

It's very handy to have a chart of your available instructions organized into these rough categories. Here is one below, you can start with today. You will see some instructions end up being categorized in more than one way. The reason for this is those instructions bundle functions together.

Eg: DJNZ This instruction decrements a specific COG memory location, and performs a JMP (jump to) operation, if that COG memory location is not yet equal to zero. That's a couple of operators (decrement and compare to zero) and a program flow element present, all in one instruction! (jump to, if not zero) I've got those in bold.

Data Transport	**Operators**	**Operators**	**Program Flow**
CLKSET	ABS	NEG*	CALL
COGID	ABSNEG	NEGC	COGINIT
	ADD	NEGNC	COGSTOP
HUBOP	ADDABS	NEGNZ	
	ADDS	NEGZ	**DJNZ**
LOCKCLR	ADDSX		
LOCKNEW	ADDX	OR	JMP
LOCKRET	AND		JMPRET
LOCKSET	ANDN	RCL	
		RCR	NOP
MOV	CMP	REV	
MOVD	CMPSUB	ROL	RET
MOVI	CMPSX	ROR	
MOVS	CMPX		TJNZ
		SAR	TJZ
RDBYTE	**DJNZ**	SHL	
RDLONG		SHR	WAITCNT
RDWORD	MAX	SUB	WAITPEQ
	MAXS	SUBABS	WAITPNE
WAITVID	MIN	SUBS	**WAITVID**
WRBYTE	MINS	SUBSX	
WRLONG	MUXC	SUBX	
WRWORD	MUXNC	SUMC	
	MUXNZ	SUMNC	
	MUXZ	SUMNZ	
		TEST	
		XOR	

From the chart above, it's clear that most of the instructions are operators. Where parsing a program is concerned, this means starting with the program flow and data movement instructions helps put all the operators in context. Not a bad place to start! You get a lot of structure with out having to handle too many instructions at one time.

Each instruction does a fairly simple thing. Maybe it's just moving a number to a location. What that number does is often as much about the hardware as it is anything else. Moving a number to set a pin state, for example, might light an LED, or trigger some other device to do something.

Moving that same number to a different location in memory may just provide some information necessary to complete a math operation in another part of the program too.

The Process

Having gone through that, it's time to take a look at the process. Generally speaking, it's good to go through and understand which instruction does what, according to the gross characterizations above. So, is it a data move, operation, or program flow instruction?

From there, taking it a pass at a time, you break the program down into program areas, data areas, inputs and outputs, and other such things that help you to understand the structure of things. It's not a bad idea to print things out, and make notes to reinforce what you are learning.

Finally, that structure and understanding of what the instructions do, when they do it, and on what they do it to, all combine to form higher level intent. Having arrived here, you should be able to put together a plain English description of what that program does.

It's worth noting at this point, changing the program, or writing a similar one is really this process in reverse! You go from higher level descriptions back down to the granular, instruction by instruction tasks necessary for it to happen on whatever hardware you are writing for.

The bulk of the main document is about the computing foundations necessary to understand what instructions do and what programs look like. This appendix is really about taking that and applying it to other programs to speed the learning and use process.

Consider the following bit of code:

```
ORG 0

mov             Counter_Time_To_Wait_For, CNT
add             Counter_Time_To_Wait_For, #$20   'Initial_Wait_Time
waitcnt         Counter_Time_To_Wait_For, Additional_Wait_Time
```

In the small bit of code above, there is one of each instruction type. They are colored, as if highlighted. If this were a larger program, some time would have been spent locating program flow instructions, looking at the labels to see what blocks of code execute when

and where, followed by a few more passes to locate movement and operators.

In this case, it's enough to just properly characterize each instruction and move on. It's just a code segment to illustrate the process in general.

The **mov** instruction is a transport / movement instruction. Really, all this one does is move some bits from one COG memory location to another. This does not tell much of the story however, and that's where the hardware part of the picture comes into play. CNT happens to be a predefined constant that points to the COG memory location where the current value of the system counter can be obtained. The move instruction takes whatever that value is and stores it in a COG memory location, so that the other parts of the program can work with it.

In plain English, this instruction just notes what the value of the counter is at the time it is executed, for use later on. That's it!

In SPIN, that might look like this:

Counter_Time_To_Wait_For := CNT

> *Note how the labels convey some of the story too. Often what the labels are will offer up clues as to what is happening. Sometimes this is the case and sometimes it isn't. For program segments with seemingly cryptic labels, it's often worth it to make label notes, or do some digging in the hopes to see what the label might mean.*

We know now there is a system counter, it's value is important to this program, and that value has been noted for use later on.

The **add** instruction is an operator one. Essentially, this boils down to some bits getting changed. In this case, the immediate value $20 is added to the counter value already noted, with the result of that addition stored in the COG memory location Counter_Time_To_Wait_For.

In English then, we understand the program is adding a value to the one noted above and storing it where the original value used to be.

In SPIN, that's very similar to this:

Counter_Time_To_Wait_For := Counter_Time_To_Wait_For + $20

Now we are down to the last instruction, **waitcnt**. This one is tricky because it's really two instructions in one! The wait part of the instruction is program flow related. However, it's also an operator in that it performs an addition after it's done waiting!

Waitcnt pauses until the system counter, CNT, matches a value. In this case, that value is

specified by the label, Counter_Time_To_Wait_For.

In English this is:

Pause until the system counter is equal to Counter_Time_To_Wait_For.

That's the program flow part of the instruction. The operator type is straightforward in that the value identified by the label, Additional_Wait_Time, will be added to the value the program was waiting for.

In English then, this is:

When the counter matches the target value, add Additional_Wait_Time to the target and move on.

Looking it all over, it then becomes clear the program is doing the following, expressed in plain English:

1. Look at the counter

2. Note that value

3. Add on some known additional value to compensate for time consumed to get things done (execute other instructions)

4. Wait on that result, allowing the counter to catch up.

5. Having caught up, add in the next wait time.

> *There are some hardware considerations here, that I glossed over. The reason some time is added to the counter value is because instructions take time to execute. If that time is not accounted for, the program will miss the target counter value and hang for about a minute as the counter counts all the way through and wraps around.*

That's basically how it goes from bottom to top. From here, we know some things about the program and can make changes. The initial delay time is the $20, value in the add instruction. The other additional wait time can be changed in the waitcnt instruction as well. These changes only make sense once the overall intent behind the program is well understood.

Writing new programs, or maybe just changing this one then works in reverse! Formulate the high level statement that details what needs to happen, break it into discrete hardware related pieces, apply the instructions that can perform the task, and sequence them into the program flow.

Addendum B

Propeller Memory Addressing

There are two distinct address spaces in the Propeller. They are COG and HUB memory addressing. The Propeller is fairly, if not totally unique in it's method of loading the COG from HUB memory to run assembly language programs. One artifact of this is program instructions only execute from COG RAM.

HUB

The HUB is byte addressed, meaning each numerical address corresponds to a byte in the HUB storage area. $0000 = the first addressable byte. $0001 = the second addressable byte, etc... Addressing is generally expressed in terms of the most granular address specification possible. HUB addresses operate on a byte, word and long basis.

COG

COG memory is long addressable only. $00 = the first long, which is actually 4 bytes. $01 = the second long, etc... Instruction source and destination bit fields are 9 bits, just enough for the 512 possible COG addresses.

Bit, byte, word and instruction bit field addressing must either be done with masks and shifts, or through the instructions provided for directly manipulating program instruction bit fields. Those are movi, movd, movs, and are briefly covered here and in the terminology portion of this document.

Additionally, there are two primary addressing forms, or modes as well. They are direct and indirect. The octothorpe character '**#**' is used to differentiate these in your assembly language program. The octothorpe specifies that a value is an immediate value to be used directly as the source value, instead of as the address of a COG memory location where a source value could be found.

Indirect addressing

Indirect addressing is generally the norm on the Propeller. Indirect addresses are those addresses where a value contained in an instruction bit field is the address of a COG memory storage location that contains the value that is the actual address.

Immediate addressing '#'

Things are simpler with Immediate addressing. This is often called direct addressing also, and that is generally the convention I will use here. It's simple to contrast direct and

indirect.
A value contained in the source instruction bit field is the target address. Source address values are the only values that can be specified in the immediate mode.
Most of the assembly language instructions you will write for the propeller will be using the indirect addressing form. A few common exceptions follow.

ADD [destination] [#source] direct or immediate addressing eg: add counter, #5

In plain English, this instruction means add the value 5 to whatever value is contained at the cog address associated with the label “counter”. Let's say that “counter” = 7 and that COG storage location 7 contains 25. Here is what happens in greater detail:

The Propeller COG executing the instruction reads the destination bit field, sees the number 7, and understands that the target address is then COG storage memory 7 will be the target address.

The COG then looks at the source instruction bit field and sees the value 5. It also looks at the state of the immediate mode instruction bit, and if it's a 1, understands that immediate mode addressing is in effect. The value to be added is then used directly for the addition.

After the addition is performed (25 + 5), the flags are set, if warranted, and the sum is written to the destination address. COG storage location 7 then contains the value 30.

ADD [destination], [source] indirect addressing ` eg: add counter, 5

In plain English, this instruction means add the value contained in COG storage location 5, to the value contained in the COG storage location associated with the label counter. Again, “counter” = 7, COG storage location 7 contains 25. In addition, let's also say that COG storage location 5, contains the value 50.

The Propeller COG executing this instruction reads the destination bit field, sees the value 7, and understands the target address is COG memory 7. Nothing different here.

When the COG looks at the source instruction bit field, it sees the same value 5, only this time the immediate mode instruction bit is now a 0. This means the source address is an indirect one! The COG then looks at COG memory location 5, sees the value 50, and performs the addition.

After the addition is performed (25 + 50), the flags are set, if warranted, and the sum is written to the destination address. COG storage location 7 then contains the value 75.

[more examples to come]

The special case of implied addressing

There is another form of addressing on the Propeller that I have not seen formally defined in the documentation, and that is implied addressing. An implied address is one where the instruction itself contains the address!

On the Propeller there are instructions created specifically for the purpose of directly modifying the bit fields of an instruction contained in a COG storage memory location. The implied address is the actual bit field of an instruction. This addressing form is used to build self-modifying code. Self-modifying code is a form of programming where the program modifies it's own instructions.

There are no index registers on the Propeller, meaning it is up to the programmer to modify instructions for this purpose.

We generally think of an index as a pointer to a value in a series of values. Indexes are easiest, if done as simple numbers. The first value in the table is associated with an index value of 0. The second one is associated with index value 1, and so on.

Programmers generally like counting from 0. Have you ever wondered why this is? This is why! In the assembly language realm, an index is **ADD**ed to the address of the table to be indexed. If we indicate the index of the first value in the table with a zero, then when it's added to the table address, the resulting effective address points directly to the first value in the table.

If we do this with a 1, then we must either offset the address of the table by one, or pad the table with a dummy value that won't ever get used. Neither of these options make a lot of sense, so it's easier to just start counting at zero.

[program example to come]