I have been trying to learn PASM off and on for a while. After reviewing many tutorials and much of the Parallax forums, I found it not easy to get basic information about just simply communicating with PASM. Everybody wants to blink a light. That is great but how does one do simple math, an array and other tasks that are relatively simple in SPIN or Prop C.

My project involves GPS and other sensors. I decided that I would tackle the project in PASM. So, while attempting to learn to code in assembly I got some jump starts from David and Jeff at parallax which was a great help, scoured the forums and despite finding many broken links and digging through some older tutorials, I found some information. Still everyone wants to blink a light.

I wrote a version of the tutorial that is in the LEARN section for creating Prop C libraries and was encouraged by the compliments, Thank you all.

My approach to that rewrite was from the aspect of a teacher not an engineer as I am a flight instructor and an aircraft mechanic instructor at a college in the Los Angeles area. So, I attempted to not be too geeky with the tutorial so as to appeal to the inexperienced and those who are really techy.

So here is my attempt at a PASM tutorial.

No Blinky lights in the beginning!!!!

The first thing one will need is a copy of the propeller manual that is in the propeller tool and can be found here: https://www.parallax.com/product/122-32000.

Here is a link to Jeff Martin's webinar I uploaded to YouTube:
https://www.youtube.com/watch?v=OZHuWYW3o1A

The first exercise will encompass passing variables from a spin method to a pasm method and back.

This the first piece of code that I came up with. There may be better ways to do this so bear with me.

I setup two global variables one for the spin method and the other for the pasm method. A five second waitcnt is used so as to have time to open the serial terminal when launching the code.

In order to launch the pasm code into a new cog this command is needed:

cognew(@asm,@datavar). The cognew means open the next cog, the @asm is the beginning of the assembly routine and the @datavar is the address of the first global variable.

```


{{ Tutorial 1 how to pass a number variable from spin to pasm and back, this works for numbers
from 0 to 256, bigger numbers in a later tutorial}}


CON
          _clkmode = xtal1 + pll16x
         '_xinfreq = 6_250_000   'MY BOARD AT 100MHZ  DIFFERENT CRYSTAL
          _xinfreq = 5_000_000  'QUICKSTART 80 MHZ  NORMAL CRYSTAL



obj

        pst:"parallax serial terminal"

var
        long datavar
        long answervar


```

The next steps are to start the serial terminal wait five seconds to allow one to open the serial terminal and then launch the cog. The code will then take the value in data var and print on the terminal. Now to the PASM method:

```
pub main
        datavar:= 25              'assign a value to datavar


        pst.start(115000)        'start the serial terminal object

        waitcnt(clkfreq*5 +cnt) 'hold five sec to open the

        cognew(@asm,@datavar)    ' open a new cog for pasm. where it starts "asm" and
                                 ' the address of the first variable
        waitcnt(clkfreq+cnt)     ' hold for a second

                                 ' print routine

        pst.str(string("answer:"))
         pst.newline
        pst.dec(answervar)
        pst.newline

```

The datavar is assigned a value, in this case 256 which is the maximum pasm will handle without extra work. I will tackle that at a later time. We want to keep it simple at this time. This is also because many of the other tutorials I have seen get really complicated very quickly and do not take it in baby steps. I want to make sure that everybody can grasp the concept before getting into complicated code and get lost.

```
dat


  asm        org      0    'This is the starting point for PASM

                 {{ The first item is to move the address of the parameter register "PAR" into
                 a temporary variable and assigne it to the variable in which we will read the in
                 this case the value of datavar in the spin method. }}
                 mov temp_var, par
                 {{ Now we are going to assign the pasm variable, data_var, the address of datavar in
                 the spin method. }}
                 mov data_var, temp_var

                 {{ Now we have to move over to the next long to get the address of answervar in the
                 spin object and assign it to answer_var in the pasm code.}}
                 add temp_var, #4
                 {{ Now assign this address to answer_var. }}
                 mov answer_var,temp_var

                 {{ Next read the value of datavar (spin object)  into the pasm data_var. }}
                 rdlong data_var, par


                 {{ Finaly write it to the answer_var which is spin's answervar for printing. }}
                 wrlong data_var, answer_var

{{ Reserved variables reserved for PASM's use. }}

  data_var res 1
  answer_var res 1
  temp_var res 1
```

The pasm code starts in a “dat” section of spin. The “asm”   “org”  “0” indicates the beginning of the pasm code. In the cognew there is also an @datavar expression. This tells the pasm code the address of the first variable and that address will be stored in the “par” value. “par” from what I have found means parameter.

There is a very nice webinar done by Jeff Martin in 2009 that explains a lot of information regarding pasm code. I uploaded it to YouTube: https://www.youtube.com/watch?v=OZHuWYW3o1A.

Starting at:

mov temp_var, par

This is the  mov instruction description:

# MOV

**Instruction:** Set a register to a value.
MOV ***Destination***, □#□ ***Value***
**Result:** *Value* is stored in *Destination*.

***Destination*** (d-field) is the register in which to store *Value*.

***Value*** (s-field) is a register or a 9-bit literal whose value is stored into *Destination*.
Explanation

MOV copies, or stores, the number in *Value* into *Destination*.

If the WZ effect is specified, the Z flag is set (1) if *Value* equals zero. If the WC effect is specified, the C flag is set to *Value's* MSB. The result is written to *Destination* unless the NR effect is specified.

So, our first instruction directive will take the address of the spin code datavar variable in the registers and pass it to a temporary variable that we can manipulate. The code is commented so as to follow the progression and I am using full words instead of abbreviations so as one could more easily follow the progression.

```

asm         org       0     'This is the starting point for PASM

                  {{ The first item is to move the address of the parameter register "PAR" into
                  a temporary variable and assigne it to the variable in which we will read the in
                  this case the value of datavar in the spin method. }}
                  mov temp_var, par
                  {{ Now we are going to assign the pasm variable, data_var, the address of datavar in
                  the spin method. }}
                  mov data_var, temp_var
```

Now we have the address of the data_var which corresponds to datavar in the spin method.

```
                  {{ Now we have to move over to the next long to get the address of answervar in the
                  spin object and assign it to answer_var in the pasm code.}}
                  add temp_var, #4
                  {{ Now assign this address to answer_var. }}
                  mov answer_var,temp_var
```

As you can see, we move over and get the address of the spin code answervar variable and assign it's address to the pasm code answer_var variable. This is done by adding 4 to the temporary variable. Adding 4 moves to the next adjacent long where the answer var is located in the hub.

We are next going to use the rdlong and wrlong directives. The rdlong directive will read from a location and copy the value into a destination field as is shown in the propeller manual listing.

RDLONG ***Value***, □#□ ***Address***
**Result:** Long is stored in *Value*.

***Value*** (d-field) is the register to store the long value into.

***Address*** (s-field) is a register or a 9-bit literal whose value is the main memory address to read from.

The rdlong goes from right to left. We are reading the value that is in the par register which has the location of datavar and it's contents.

```

                   {{ Next read the value of datavar (spin object)  into the pasm data_var. }}
                   rdlong data_var, par

```

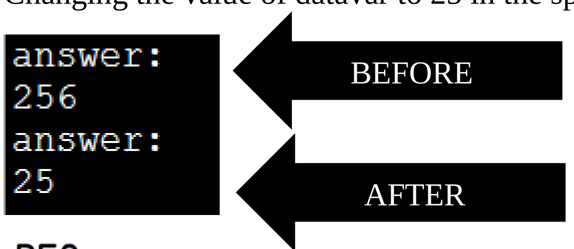
Lastly, we are going to write the value to the answer_var location that corresponds with answervar in the spin method and then print the results in a new variable. Note: wrlong works from ***<u>left to right.</u>***

```
                {{ Finaly write it to the answer_var which is spin's answervar for printing. }}
                wrlong data_var, answer_var
```

You should get a value on the serial terminal. I used 256 as this is the largest value for a single long, which is four bytes in size.

```
answer:
256
```

Changing the value of datavar to 25 in the spin method to verify.




## RES

**Directive:** Reserve next long(s) for symbol.

⟨***Symbol***⟩ RES ⟨***Count***⟩

- ***Symbol*** is an optional name for the reserved long in Cog RAM.
- ***Count*** is the optional number of longs to reserve for *Symbol*. If not specified, **RES** reserves one long.

RES: We need to reserve space for the pasm variables this is self-explanatory.

Now we can manipulate two variables and print them in succession. This is the new code:

```


{{ Tutorial 2 how to pass two number variables from spin to pasm and back, this works for numbers
from 0 to 256, bigger numbers in a later tutorial}}

CON
          _clkmode = xtal1 + pll16x
         '_xinfreq = 6_250_000   'MY BOARD AT 100MHZ  DIFFERENT CRYSTAL
          _xinfreq = 5_000_000  'QUICKSTART 80 MHZ  NORMAL CRYSTAL

obj

          pst:"parallax serial terminal"

var
          long datavar          {{each of these are one long apart. Have to move over one long
                                  so as to access them}}
          long answervar
          long datavar2
          long answervar2

```

```
pub main
        datavar:= 21             'assign a value to datavar
        datavar2 := 29

        pst.start(115000)       'start the serial terminal object

        waitcnt(clkfreq*5 +cnt) 'hold five sec to open the serial terminal

        cognew(@asm,@datavar)   ' open a new cog for pasm. where it starts "asm" and
                                ' the address of the first variable
        waitcnt(clkfreq+cnt)    ' hold for a second

                                ' print routine

        pst.str(string("answer:"))
        pst.newline
        pst.dec(answervar)
        pst.newline
        pst.str(string("answer:"))
        pst.newline
        pst.dec(answervar2)
        pst.newline

```

```

dat

  asm         org       0     'This is the starting point for PASM

              {{ The first item is to move the address of the parameter register "PAR" into
              a temporary variable and assigne it to the variable in which we will read the in
              this case the value of datavar in the spin method. }}
              mov temp_var, par
              {{ Now we are going to assign the pasm variable, data_var, the address of datavar in
              the spin method. }}
              mov data_var, temp_var
              rdlong data_var, temp_var
              {{ Now we have to move over to the next long to get the address of answervar in the
              spin object and assign it to answer_var in the pasm code.}}
              add temp_var, #4
              {{ Now assign this address to answer_var. }}
              mov answer_var,temp_var
               {{write the value to the answervar in spin}}
              wrlong data_var, answer_var
               {{go back and get the par address to access the next variable}}
              mov temp_var, par
```

```
                {{jump over two longs to get the address of datavar2 in the spin method}}
                add temp_var, #8
                 {{assign the address}}
                mov data_var2, temp_var
                {{read the value}}
                rdlong data_var2, temp_var
                {{skip over one long to get answervar in spin}}
                add temp_var, #4
                 {{assign the address}}
                mov answer_var2,temp_var
                {{now write the value to answervar2 in spin}}
                wrlong data_var2, answer_var2


{{ Reserved variables reserved for PASM's use. }}

   data_var res 1
   data_var2 res 1
   answer_var res 1
   answer_var2 res 1
   temp_var res 1
```

We have added a couple of items. First a new datavar named datavar2 and a new answervar named answervar2 as well as their counterparts in the pasm method. In the print area answervar2 has been added also.

```
var
        long datavar         {{each of these are one long apart. Have to move over one long
                               so as to access them}}
        long answervar
        long datavar2
        long answervar2

```

Note the order of the global variables. This will make it easy to find them in the pasm method.

The pasm routine begins just like before and we get the location of datavar from par into the temporary variable and assign the location to data_var and read the value from par to data_var.

Now we have to move over a couple of longs to get the new variables and values:

```
                mov temp_var, par
                {{jump over two longs to get the address of datavar2 in the spin method}}
                add temp_var, #8
                 {{assign the address}}
                mov data_var2, temp_var
                {{read the value}}
                rdlong data_var2, temp_var
```

Now we can write the value to the second answer_var. Remember wrlong is from left to right as opposed to rdlong and other directives which are right to left.

```
{{move over one long to get answervar in spin}}
add temp_var, #4
 {{assign the address}}
mov answer_var2,temp_var
{{now write the value to answervar2 in spin}}
wrlong data_var2, answer_var2
```

This is what you should see on the serial terminal:

```
answer:
21
answer:
29
```

Changing the two datavar's values:

```
answer:
150
answer:
256
```

It works.

Now that we can get in and out of spin and pasm, I will present some examples of simple math.

I am trying to avoid the jump to really complicated programs with the assumption that the reader has a total comprehension of coding in assembly language of any type. I have found many tutorials do that.

These tutorials were good but confusing when they jump ahead and get very complex. Since I am a teacher, I teach flying and aircraft mechanics, I have to assess the background of each student. Academic learning can be difficult and painful, so if the instructor keeps it simple and explains the concept with easy examples that build up slowly, the student has a better chance of understanding and correlating the subject matter.

That results in a much better outcome. First addition, note the global variable name change. We are going to repeat the above code and make some changes:

## ADD

**Instruction:** Add two unsigned values.

**ADD *Value1*, ⟨#⟩ *Value2***

**Result:** Sum of unsigned *Value1* and unsigned *Value2* is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to add to *Value2* and is the destination in which to write the result.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is added into *Value1*.

```
{{basic addition in pasm using the add directive. Page259 propeller manual}}

CON
_clkmode = xtal1 + pll16x
'_xinfreq = 6_250_000    'MY BOARD AT 100MHZ  DIFFERENT CRYSTAL
_xinfreq = 5_000_000  'QUICKSTART 80 MHZ  NORMAL CRYSTAL

var
   'VARIABLE IN THE PAR ADDRESS TO BE PASSED
   long x
   long y
   long product
obj

pst:"parallax serial terminal"

pub main
      x := 30
      y := 45
pst.start(115000)
    waitcnt(clkfreq*5 +cnt)'hold five sec to open the
     'serial terminal and enable it
cognew(@asm,@x)'start cog at the first variable address
      waitcnt(clkfreq*2 +cnt) 'give pasm time to do the work

      pst.str(string("product:"))
      pst.dec(product~)
      pst.newline
```

```

dat

asm        org

            mov tempvar, par 'get the address of x from par
            mov xvar, tempvar 'assign the address to the xvar in pasm
            rdlong xvar, tempvar 'read the value that is in x
            add tempvar, #4  'move over one long to get y's address
            mov yvar, tempvar 'assign that address to yvar
            rdlong yvar, tempvar 'read the value that is in y
            add tempvar, #4 'move over one long to get the address of product
            mov productvar, tempvar 'assign the address to productvar
            add xvar,yvar   'add x and y together answer will be in x
            wrlong xvar, productvar 'write x into the product variable and print


tempvar long 0
xvar long 0
yvar long 0
productvar long 0
flag long 0
```

```
product:75
```

Subtraction:

```



{{ Tutorial on how to pass a number variable  and perform subtraction
with the sub directive
from spin to pasm and back, this works for numbers
from 0 to 256, bigger numbers in a later tutorial}}


CON
          _clkmode = xtal1 + pll16x
         '_xinfreq = 6_250_000   'MY BOARD AT 100MHZ  DIFFERENT CRYSTAL
          _xinfreq = 5_000_000  'QUICKSTART 80 MHZ  NORMAL CRYSTAL
obj

        pst:"parallax serial terminal"

var     'global variables
        long datavar
        long answervar
        long subvar

pub main
        datavar:= 25             'assign a value to datavar
        subvar := 10

        pst.start(115000)       'start the serial terminal object

        waitcnt(clkfreq*5 +cnt) 'hold five sec to open the

        cognew(@asm,@datavar)    ' open a new cog for pasm. where it starts "asm" and
                                 ' the address of the first variable
        waitcnt(clkfreq+cnt)     ' hold for a second

                                 ' print routine

        pst.str(string("results:"))
         pst.newline
        pst.dec(answervar)
        pst.newline
```

```
41
42 dat
43
44
45   asm         org      0    'This is the starting point for PASM
46
47               {{ The first item is to move the address of the parameter register "PAR" into
48               a temporary variable and assigne it to the variable in which we will read the in
49               this case the value of datavar in the spin method. }}
50               mov temp_var, par
51               {{ Now we are going to assign the pasm variable, data_var, the address of datavar in
52               the spin method. }}
53               mov data_var, temp_var
54
55               {{ Now we have to move over to the next long to get the address of answervar in the
56               spin object and assign it to answer_var in the pasm code.}}
57               add temp_var, #4
58               {{ Now assign this address to answer_var. }}
59               mov answer_var,temp_var
60               add temp_var,#4  'move over to the next long and get the subtraction variable address
61               mov sub_var, temp_var 'assign the address to the variable
62               rdlong sub_var,temp_var  'read the value in that address
63               {{ Next read the value of datavar (spin object)  into the pasm data_var. }}
64               rdlong data_var, par  'go back and get the value from the data variable that is in the par register
65               sub data_var,sub_var  ''perform the subtraction data-subvar= xxx
66
67               {{ Finaly write it to the answer_var which is spin's answervar for printing. }}
68               wrlong data_var, answer_var
69
```

```
70 {{ Reserved variables reserved for PASM's use. }}
71    sub_var res 1
72    data_var res 1
73    answer_var res 1
74    temp_var res 1
```

What we have done is simply, at lines 60 and 61, added a new variable as well at line 71, these will be the subtraction variables. Next perform the subtraction and then write to our answer variable.

You should get this:

```
results:
15
```

25-10=15

Change subtraction variable to 12.

```
results:
13
```

25-12=13

Multiplication this is from the propeller manual page 380:

```
{{Multiplication based on the propeller manual page 380}}

CON
_clkmode = xtal1 + pll16x
'_xinfreq = 6_250_000   'MY BOARD AT 100MHZ  DIFFERENT CRYSTAL
_xinfreq = 5_000_000  'QUICKSTART 80 MHZ  NORMAL CRYSTAL

var

   'VARIABLE IN THE PAR ADDRESS TO BE PASSED
   long x
   long y
   long product
obj

pst:"parallax serial terminal"

pub main
      x := 3
      y := 27
pst.start(115000)
    waitcnt(clkfreq*5 +cnt)'hold five sec to open the
    'serial terminal and enable it
cognew(@asm,@x)'start cog at the first variable address
      waitcnt(clkfreq*2 +cnt) 'give pasm time to do the work

      pst.str(string("product:"))
      pst.dec(product~)
      pst.newline
```

```


dat
'' Multiply x[15..0] by y[15..0] (y[31..16] must be 0)
' on exit, product in y[31..0]
'
asm                    org

                       mov temp_var, par  'move par to a temporary variable
                       mov x_var, temp_var 'find the x variable
                       rdlong x_var, temp_var 'read in the value from top object
                       add temp_var, #4      'jump to next long which is the address of the
                                              ' next variable
                       mov y_var, temp_var  'repeat assignment and read in value
                       rdlong y_var, temp_var
                       add temp_var, #4      'jump again to assign the product variable address
                       mov product_var, temp_var
```

```
47
48                      multiply shl x_var,#16 'get multiplicand into x[31..16]
49                      mov t,#16 'ready for 16 multiplier bits
50                      shr y_var,#1 wc 'get initial multiplier bit into c
51 :loop                if_c add y_var,x_var wc 'if c set, add multiplicand to product
52                      rcr y_var,#1 wc 'put next multiplier in c, shift prod.
53                      djnz t,#:loop 'loop until done
54                      wrlong y_var, product_var    'write the product from y[31..0] to the
55                                                   'product variable for the top object
56
57                      'multiply_ret ret 'return with product in y[31..0] 'this would be a subroutine
58                      ' when used in a program
59 temp_var res 1
60 x_var res 1
61 y_var res 1
62 product_var res 1
63 t res 1
```

product:81 3*27=81

Change 27 to 9.

product:27 3*9=27

Basically, we are doing multiplication by addition:

27+27+27=81

3+3+3+3+3+3+3+3+3=27

The first operation is to shift left, the multiplicand into x[31..16], line 48.

## SHL

**Instruction:** Shift value left by specified number of bits.

**SHL *Value*, 〈#〉 *Bits***

**Result:** *Value* is shifted left by *Bits*.

- ***Value*** (d-field) is the register to shift left.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits to shift left.

Next because this is 16 bit multiplication, so we are going to load a variable with the number 16, line 49: mov t,#16 'ready for 16 multiplier bits.

We are going to shift the carry into y by 1 each time we add the variables. So, on line 50 the first iteration will be loaded. This is done by shifting y right by one to get the carry flag set with the first number that will eventually be the result of the multiplication.

SHR: There is a shift right and shift left these are self-explanatory in the propeller manual as shown. The code will shift left or right by the number specified.

Line 50: shr y_var,#1 wc 'get initial multiplier bit into c

## SHR

**Instruction:** Shift value right by specified number of bits.

**SHR *Value*, ⟨#⟩ *Bits***

---

**Result:** *Value* is shifted right by *Bits*.

- ***Value*** (d-field) is the register to shift right.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits to shift right.

Now we are going to ask if the carry flag is set when we add x and y. this will loop until the carry flag is not set an we will loop back and perform the operation again. Each addition will be counted until finished. When completed the carry will be the result of the multiplication. The carry will be discussed in the "if" conditional in the next paragraphs.

Now the loop:

If the carry flag is set, we will loop back and perform an add instruction and check the carry flag after each iteration. This conditional jump will be performed by the DJNZ directive what will evaluate the carry. If the carry in this case is set it will jump back to the beginning of the loop where the RCR instruction will rotate the carry flag, RCR, over into y at the end the value in y will be the answer. Basically, it adds up the carry bits. . If the carry is not set it will NOP, NO OPERATION, and drop out of the loop and go to the next instruction which in this case is to write the results to the variable, product_var and will be printed.

Which in the end of the loop, would be the answer if one did multiplication via the addition process.

RCR:

## RCR

**Instruction:** Rotate C right into value by specified number of bits.

**RCR Value, ⟨#⟩ Bits**

---

**Result:** *Value* has *Bits* copies of C rotated right into it.

- ***Value*** (d-field) is the register in which to rotate C rightwards.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits of *Value* to rotate C rightwards into.

CONDITIONAL STATEMENTS:

## IF_x (Conditions)

Every Propeller Assembly instruction has an optional "condition" field that is used to dynamically determine whether or not it executes when it is reached at run time. The basic syntax for Propeller Assembly instructions is:

〈*Label*〉 〈*Condition*〉 ***Instruction Operands*** 〈*Effects*〉

The optional *Condition* field can contain one of 32 conditions (see Table 3-3) and defaults to IF_ALWAYS when no condition is specified. The 4-bit **Value** shown for each condition is the value used for the **-CON-** field in the instruction's opcode.

This feature, along with proper use of instructions' optional *Effects* field, makes Propeller Assembly very powerful. Flags can be affected at will and later instructions can be conditionally executed based on the results. Here's an example:

```
        test   _pins, #$20      wc
        and    _pins, #$38
        shl    t1, _pins
        shr    _pins, #3
        movd   vcfg, _pins
if_nc   mov    dira, t1
if_nc   mov    dirb, #0
if_c    mov    dira, #0
if_c    mov    dirb, t1
```

The first instruction, `test _pins, #$20  wc`, performs its operation and adjusts the state of the C flag because the WC effect was specified. The next four instructions perform operations that could affect the C flag, but they do not affect it because no WC effect was specified. This means that the state of the C flag is preserved since it was last modified by the first instruction. The last four instructions are conditionally executed based on the state of the C flag that was set five instructions prior. Among the last four instructions, the first two mov instructions have if_nc conditions, causing them to execute only "if not C" (if C = 0). The last two mov instructions have if_c conditions, causing them to execute only "if C" (if C = 1). In this case, the two pairs of mov instructions are executed in a mutually exclusive fashion.

When an instruction's condition evaluates to FALSE, the instruction dynamically becomes a NOP, elapsing 4 clock cycles but affecting no flags or registers. This makes the timing of multi-decision code very deterministic.

DJNZ:

## DJNZ

**Instruction:** Decrement value and jump to address if not zero.

DJNZ ***Value***, 〈#〉 ***Address***

**Result:** *Value*-1 is written to *Value*.

- ***Value*** (d-field) is the register to decrement and test.
- ***Address*** (s-field) is the register or a 9-bit literal whose value is the address to jump to when the decremented *Value* is not zero.

This directive allows for repetition while decrementing a particular value of choice and when the result is not zero jump to a particular point in the code until the result is zero. At that point the code will drop down to the next instruction in line.

We run the loop until the carry flag is empty. This is repeated addition. Jeff and Dave at Parallax told me that there are many ways to do this. I am working on this myself. Basically, it is repetitive addition and that can be done in a loop until the number of iterations required are completed.

Division:

```
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000          'QUICKSTART 80 MHZ  NORMAL CRYSTAL

var
   long dividend          'VARIABLE IN THE PAR ADDRESS TO BE PASSED
   long divisor
   long quotient
   long remainder

obj
  pst : "parallax serial terminal"

pub main
   dividend := 211
   divisor := 6
   pst.start(115200)
   waitcnt(clkfreq*5 + cnt)  'hold five sec to open the
                             'serial terminal and enable it
   cognew(@asm,@dividend)    'start cog at the first variable address
   waitcnt(clkfreq + cnt)    'give top object time to catch up to pasm


   pst.str(string("quotient:"))
   pst.dec(quotient)
   pst.newline
   pst.str(string("remainder:"))
   pst.dec(remainder)
   pst.newline


dat


asm        org

           mov tempvar, par       'get the par address into the temporary variable
           rdlong x, tempvar      'read the value into the dividend
           add tempvar, #4        'move over to the next long to get the divisor variable
           rdlong y, tempvar      'read the value of the divisor into the variable
           add tempvar, #4        'move over to the next long to get the quotient address


' Divide x[31..0] by y[15..0] (y[16] must be 0)
' on exit, quotient is in x[15..0] and remainder is in x[31..16]
'
divide     shl y,#15              'get divisor into y[30..15]
           mov t,#16              'ready for 16 quotient bits
:loop      cmpsub x,y    wc       'y =< x? Subtract it, quotient bit in c
           rcl x,#1    'rotate c into quotient, shift dividend
           djnz t,#:loop          'loop until done

'  quotient in x[15..0],  ;return if used as a subroutine
'  remainder in x[31..16]
```

```
55
56          mov     quotientvar,x
57          and     quotientvar,andvar2     'isolate lower 16 bits
58          wrlong  quotientvar,tempvar     'write into Spinvar 'quotient'
59
60          mov     remaindervar,x
61          shr     remaindervar, #16       'isolate higher 16 bits
62          add     tempvar,#4              'incr pointer to remainder address
63          wrlong  remaindervar,tempvar    'write into Spinvar 'remainder'
64
65 andvar2       long $ffff
66 tempvar       res 1
67 x             res 1
68 y             res 1
69 quotientvar   res 1
70 remaindervar  res 1
71 t             res 1
72
```

```
quotient:35
remainder:1
```

The division will be a continued subtraction algorithm that will subtract the divisor from the dividend until the divisor is either zero or there is a remainder less than the divisor. The answer will now be in the quotient the low bits, with the remainder in the high bits.

On line 47 we are going to shift left the divisor by 15 bits to get it into the high end of y. Then move the number 16 into t because t will be our iterations for the DNJZ directive which will perform the loop function 16 iterations. Now the compare and subtract, cmpsub, will subtract y from x and see if it is zero, the carry flag will answer the condition. At each iteration we will rotate carry left, RCL, by one. At the end of all operations x will have the quotient and y will have the remainder.

## CMPSUB

**Instruction:** Compare two unsigned values and subtract the second if it is lesser or equal.

CMPSUB ***Value1***, 〈**#**〉 ***Value2***

**Result:** Optionally, *Value1* = *Value1* – *Value2*, and Z and C flags = comparison results.

- ***Value1*** (d-field) is the register containing the value to compare with that of *Value2* and is the destination in which to write the result if a subtraction is performed.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is compared with and possibly subtracted from *Value1*.

## RCL

**Instruction:** Rotate C left into value by specified number of bits.

RCL *Value*, 〈#〉 *Bits*

**Result:** *Value* has *Bits* copies of C rotated left into it.

- *Value* (d-field) is the register in which to rotate C leftwards.
- *Bits* (s-field) is a register or a 5-bit literal whose value is the number of bits of *Value* to rotate C leftwards into.

The AND operation takes $FFFF and masks off high bits so as to get the quotient, we later shift the naked remainder by 16 to get the remainder.

# AND – Assembly Language Reference

## AND

**Instruction:** Bitwise AND two values.

AND *Value1*, 〈#〉 *Value2*

**Result:** *Value1* AND *Value2* is stored in *Value1*.

- *Value1* (d-field) is the register containing the value to bitwise AND with *Value2* and is the destination in which to write the result.
- *Value2* (s-field) is a register or a 9-bit literal whose value is bitwise ANDed with *Value1*.

Counting up and down:

```
{{counting up example, have to slow pasm. Introducing conditionals
and jmp command}}



CON
_clkmode = xtal1 + pll16x
'_xinfreq = 6_250_000    'MY BOARD AT 100MHZ
_xinfreq = 5_000_000  'QUICKSTART 80 MHZ

var

   long count

obj

pst:"parallax serial terminal"

pub main

pst.start(115000)
    waitcnt(clkfreq*5 +cnt)'hold two sed to open the
     'serial terminal and enable it
cognew(@asm,@count)

```

```
26       repeat
27        pst.dec(count~)'post clear p 157
28        pst.newline
29        waitcnt(clkfreq +cnt)
30
31 dat
32
33 asm       org
34
35            mov addr, par
36  loop      add value,#1   'counting variable
37  wait      rdlong prev, addr  wz 'what is in par??
38            if_nz   jmp #wait  'if the value in
39            'par is zero continue to next command
40            'if the value in par "addr" has not been cleared
41            'meaning the value that was put in "value" from
42            ' addr which has the address of par "parameter"
43            '
44            wrlong value, addr
45            'now write the value to the addr which has been assigned
46            'the same address as par and where the address of count in
47            'memory where the spin program can read it then jump back
48            'to the top of the loop and continue after the variable
49            ' called count has been cleard to zero
50            jmp #loop
51
```

```
52
53 addr long 0
54 value long 0
55 prev  long 0
```

Since spin is much slower than pasm, we have to interrupt pasm so spin can keep up. With that in mind we are going to look at line 27 and  37 to 51.

Line 27:

pst.dec(count~)'post clear p 157.

Y := X~ + 2
The Post-Clear operator in this example clears the variable to 0 (all bits low) after providing its current value for the next operation. In this example if X started out as 5, X~ would provide the current value for the expression (5 + 2) to be evaluated later, then would store 0 in X. The expression 5 + 2 is then evaluated and the result, 7, is stored into Y. After this statement, X equals 0 and Y equals 7.

Since Sign-Extend 7 and Post-Clear are always assignment operators, the rules of Intermediate Assignments apply to them (see page 147).

```
37  wait      rdlong prev, addr   wz
38            if_nz   jmp #wait
```

So, if the line 27 instruction has not cleared, pasm will jump back to the loop until it is cleared then pasm will perform the operation again.

Change line: 36 add to sub and you will have a continuous loop of subtraction.

```
36  loop      add value,#1   'counting variable
```

```
36  loop      sub value,#1   'counting variable
```

Arrays:

We are now able to add, subtract, multiply and divide. Basic math skills that we will now take to a next level but in a slow process. Next let’s create an array and do some math while learning to populate the array and print selected arrays cells.

Simple array.

```

{{basic array populate the array do some simple math }}


CON
_clkmode = xtal1 + pll16x
'_xinfreq = 6_250_000   'MY BOARD AT 100MHZ  DIFFERENT CRYSTAL
_xinfreq = 5_000_000  'QUICKSTART 80 MHZ  NORMAL CRYSTAL

var


  long data
  long array[10] 'global variable array 10 cells long array[0]..array[9]


obj

pst:"parallax serial terminal"


pub main
      data := 16

pst.start(115000)
    waitcnt(clkfreq*5 +cnt)'hold five sec to open the
    'serial terminal and enable it
   cognew(@asm,@data )'start cog at the first variable address



      pst.str(string("array:"))
      pst.dec(array[1])   'print the second cell first

      pst.newline

      pst.str(string("array:"))
      pst.dec(array[0])  'print the first cell second

      pst.newline
```

```
44
45
46
47 dat
48
49
50  asm        org   0
51
52            mov tempvar,par    'get the par address into a temporary variable
53
54            mov datvar,tempvar 'assign the address to the datvar in pasm
55
56            rdlong datvar,tempvar 'read in the value of the data variable from spin
57
58            add tempvar,#4    'move over and get the beginning of the array
59
60            mov arrayvar,tempvar 'assign the beginning of the array
61
62            wrlong datvar,arrayvar  'write the value from spin to array[0]
63
64            add arrayvar,#4 'move over to the next array cell
65
66            add datvar, #10  'add 10 to the value in in the data variaable from spin
67            'in this case 16 + 10 = 26
68
69            wrlong datvar,arrayvar 'write the product to the second array cell array[1]
70
71
72
73
74
75 tempvar long 0
76 datvar long 0
77
78 arrayvar long 10  'global variable array 10 cells long array[0]..array[9]
```

We are going to start as before and this time have two global variables. One is the data to be passed with a value from spin to pasm. The other is an array that is 10 cells long. That means that each cell will be a long in size.

As you can see in the spin method and the pasm method both are declared. Standard entry to get the addresses and values entered.

The line 56, read in the value to the datavar variable.

Line 62 write it to the first array cell, array[0].

Now to access the second array cell, array[1], we have to move over to the next long, line 64 by adding 4 bytes. Now for a little math to make it interesting we are going to add the littoral number 10 to the variable that is stored in the datavar which is 16. So 16+10=26.

The spin method is going to print them in reverse order which shows that we can manipulate the array.

```
array:26
array:16
```