

Author: Jeremy Leach Email: [ukc802139700@btconnect.com](mailto:ukc802139700@btconnect.com)

April 2006

## Introduction

This article describes a 370-byte PICAXE code 'module' that allows simple maths to be performed to 32-bit precision. The code executes maths expressions, which are stored as simple 'strings' in data EEPROM.

When a PICAXE is evaluating an expression, like  $AvWindSpeedWord = WindWord1 + WindWord2 + WindWord3 / 3$ , the onboard integer maths is performed to 16-bit resolution. The PICAXE has a 16-bit Accumulator. This means that any intermediate result that exceeds 16bits will be corrupted, and the overall result corrupted.

So by providing 32-bit maths, this code allows expressions to be evaluated without the normal overflow problems.

The whole code module is based around a 'Virtual CPU' having a 32-bit Accumulator. The code is experimental and it's unclear whether it will really prove to be useful.

Rich-text code is available from my web-site:  
[http://home.btconnect.com/PicAxe\\_Projects/Home.htm](http://home.btconnect.com/PicAxe_Projects/Home.htm)



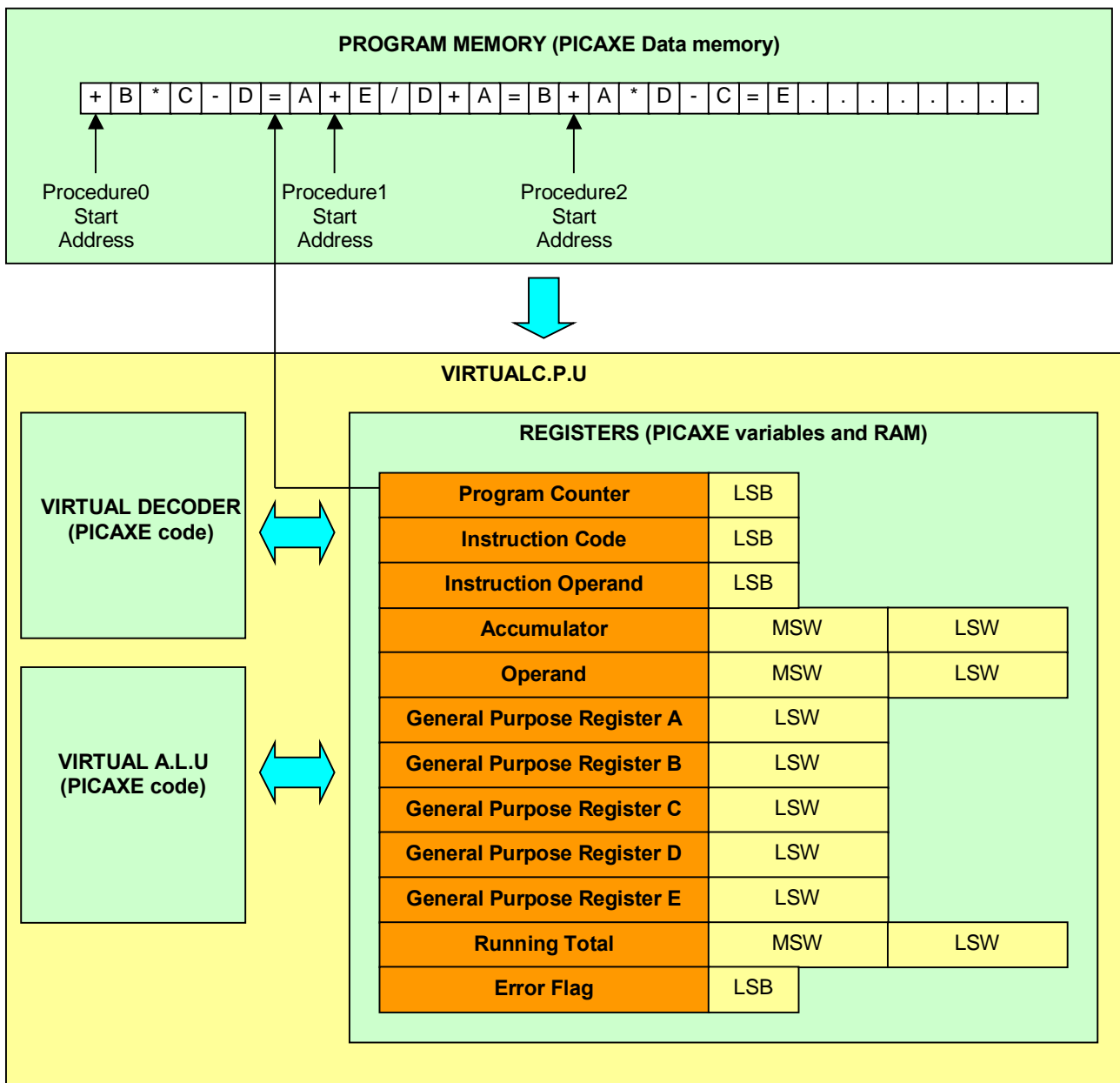
## IN THIS ARTICLE

- 2 A Virtual CPU
- 3 Using the CPU
- 3 The Machine Language
- 4 The Virtual ALU
- 5 Speed Trials
- 5 Code Module
- 5 PICAXE Source Code
- 6 Code Listing

## ACKNOWLEDGEMENTS

- PICAXE is a trademark of Revolution Education Ltd.
- PICAXE Internet Forum, a constant source of inspiration, is at: <http://www.reved.co.uk/picaxe/forum/>

Figure 1 : Virtual CPU with 32-Bit Accumulator



## A Virtual CPU

The code models a virtual CPU with a 32-bit Accumulator, which is shown in outline in Figure 1. All parts of the CPU are 'virtual' because the CPU isn't a physical device!

A set of registers is implemented as a mixture of PICAXE variables and locations in PICAXE RAM. It would have been nice to only use PICAXE variables, but it wasn't possible.

The 'Program' that the CPU executes is stored in PICAXE Data memory and accessed via Read commands. The Program consists of a set of user-defined procedures, each procedure a set of

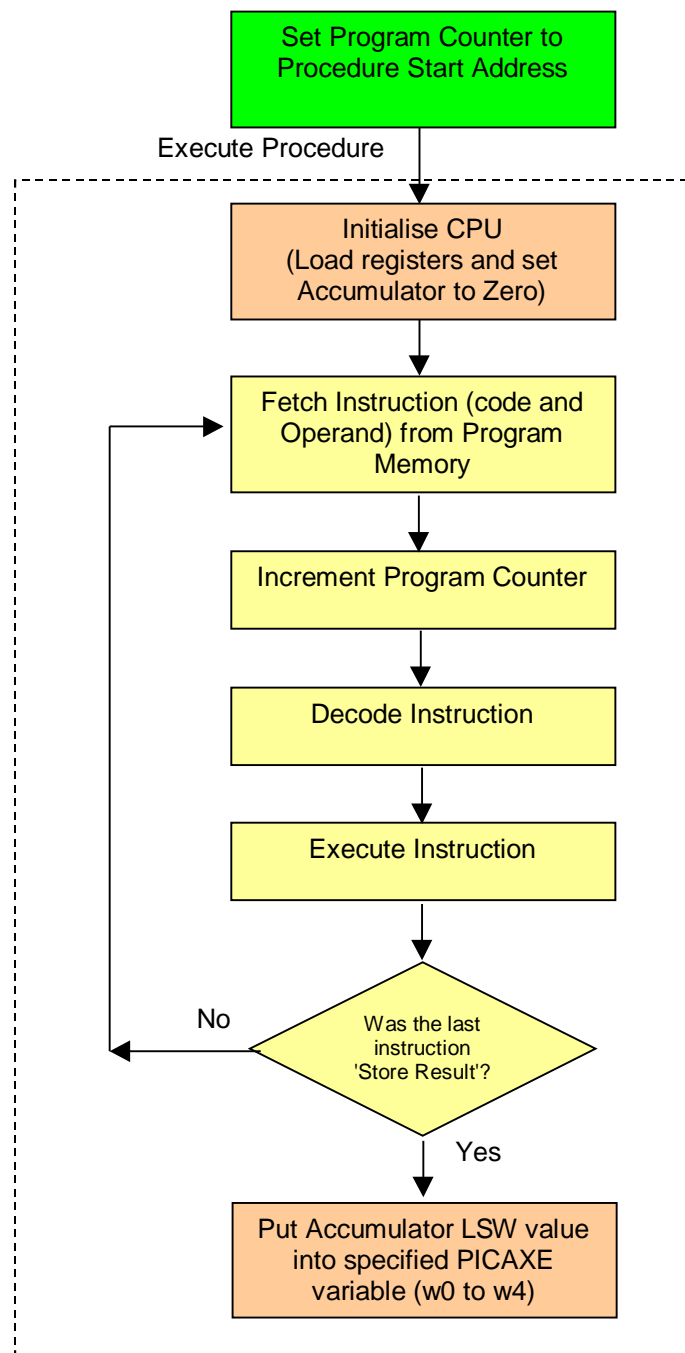
instructions.

Because this virtual CPU is being used for the purpose of 32-bit maths, the procedures are in reality mathematical expressions only. However there's nothing to stop this idea being expanded and other high-level languages written.

A virtual Decoder interprets the instructions and is implemented in quite straightforward PICAXE code. The instructions are written in a simple 'machine language'.

A virtual Arithmetic and Logic Unit (ALU) performs the basic mathematical functions of addition, subtraction, multiplication and division. Again this is implemented in PICAXE code, and is the most

Figure 2 : Calling a Procedure (evaluating a maths expression)



complex part of the overall system. The code has been carefully written to optimise both speed and storage as much as possible.

## Using the CPU

The idea behind having this virtual CPU is that you only call upon it when necessary, to perform 32-bit maths. To make the CPU run a specific procedure you simply set the Program Counter to point to the start address of the relevant procedure, then *Gosub ExecuteProcedure*. The CPU then executes the

specified procedure and exits when the result of the maths expression has been stored.

The flow is shown in Figure 2. The core tasks are a typical CPU fetch-execute cycle.

## The Machine Language

The language used is very simple. All maths is performed on the General Purpose Registers A to E.

A traditional maths expression might be:

$$A = B + C / D$$

In this language this expression is written as:

$$+ B + C / D = A$$

This means: Add B to 0. Then perform other maths operations. Then store the result in A.

The advantage of writing the instructions like this is that the procedure is now a serial set of pairs of <Instruction Code><Instruction Operand>. For Example <+><B>. This simplifies the implementation of the virtual Decoder and saves a few bytes of PICAXE program space. Also the variable to store the result in doesn't have to be remembered at the start of the expression.

Because a 'Store Result' instruction is always the last in a procedure, there is no need to have an 'end of procedure' instruction.

The variables A to E actually correspond to PICAXE Words w0 to w4. Before running the command string, w0 to w4 must be initialised to the values you want for A to E. The result of the calculation is returned to the specified variable.

## The Virtual ALU

In the world of a PICAXE there's nothing to manipulate at 32-bit (double-Word) level directly. So to be able to perform the basic maths functions of addition, subtraction, multiplication and division the maths routines have to rely on the 'currency' units available, i.e. Bits, Bytes or Words.

It's sensible to use the largest currency you can, because the code is then likely to be faster and more compact. So the code I've written manipulates Word variables.

All the A to E variables used in maths expressions are Word values. However the Accumulator is a double-Word value. This means that interim results can exceed the Word limit of 65535 - the whole point of this idea!

Because the variables A to E hold Word values, the Operand in any calculation is a Word value. The only exception is in the Addition routine, because this is also called from the multiplication and division routines when the Operand can be a double-Word.

All maths routines act on the value in the Accumulator and put the result of the calculation back into the Accumulator. For instance calling 'Add' means Accumulator = Accumulator + Operand. 'Divide' means Accumulator = Accumulator /

Operand etc.

### Addition

This sounds trivial: Add Word values together and use a 'carry' indicator. Although addition is simple to achieve there is some subtlety needed when using Word values:

PICAXE Basic doesn't allow access to a carry flag, so how do we know if the addition of two Words has resulted in a carry? If we were working with Bytes and adding the lower bytes of two Words it would be easy: we would just need to inspect the upper Bytes of the Result Word.

There's a simple observation that makes it easy to identify a carry: In decimal, if we add "5" to "6" the result is 1 carry 1. In general the value the result digit holds is ALWAYS greater or equal to the values of either operand digits, UNLESS there is a carry. For instance, in this example the result digit is "1" which is less than either Operand digit, and so indicates a Carry has taken place. However, if we add "5" to "4" we get "9" which is greater than the Operand digits and there's no Carry.

This principle will work in any number base and so this is the simple method I use to identify carry when adding Word values.

### Subtraction

Here we identify borrow with a similar rule: If the result digit value is greater than the Operand value being subtracted from then borrow must have occurred.

### Multiplication

This routine uses the special PICAXE operator \*\*, which allows the overflow, if the result exceeds a Word value, to be accessed. It also uses the Addition routine to sum the products.

### Division

This is by far the most complex routine and the principle I've worked out requires some explanation:

Imagine the division of Numerator by Denominator:  
 $N_3N_2N_1N_0 / D_1D_0$

Where  $N_0$  to  $N_3$  are the Numerator digits, and  $D_0$  and  $D_1$  the Denominator digits. This is the situation with this ALU where the Accumulator holds 4 byte 'digits' and the Operand 2 byte 'digits'.

Note the blue colour-coding to indicate that the digits are expressed as a decimal number (e.g  $N_3N_2N_1N_0 = '1234'$ ).

This division can't be done directly, so the idea is to manipulate the basic division formula to give terms that can be evaluated using the other maths functions available. The following explanation is using base 10 maths, however the principle applies to any number base. Word maths is to base 65536.

If the digits were in base 10 then the division could be modified as follows:

$$\begin{aligned} \text{Result} &= N_3N_2N_1N_0 / D_1D_0 = [(N_3 * 1000) / D_1D_0] + [(N_2 * 100) / D_1D_0] + [N_1N_0 / D_1D_0] \\ &= [N_3(10 + 990) / D_1D_0] + [(N_2(1+99) / D_1D_0) + [N_1N_0 / D_1D_0] \\ &= [10N_3(99 / D_1D_0)] + [10N_3 / D_1D_0] + [N_2 / D_1D_0] + [99N_2 / D_1D_0] + [N_1N_0 / D_1D_0] \\ &= [(99 / D_1D_0)*(10N_3 + N_2)] + [(10N_3 + N_2 + N_1N_0) / D_1D_0] \dots \text{Eqn1} \end{aligned}$$

The whole and remainder parts of  $(99 / D_1D_0)$  can be evaluated easily (and in base 65536 would just be done using Word maths). Call the whole and remainder parts S and T:

$$99 / D_1D_0 = S + (T / D_1D_0) \dots \text{Eqn2}$$

Substituting Eqn2 into Eqn1 gives :

$$\text{Result} = [S*(10N_3 + N_2)] + [([(T + 1)*(10N_3 + N_2)] + N_1N_0) / D_1D_0]$$

But  $10N_3 + N_2$  is actually the top two digits of the Numerator expressed as a decimal number. Ie  $N_3N_2$ . For instance, if the Numerator was decimal 1234 then  $N_3N_2 = 12$ . So:

$$\text{Result} = S * N_3N_2 + [([(T + 1)* N_3N_2] + N_1N_0) / D_1D_0] \dots \text{Eqn3}$$

Eqn3 is in two parts:

$$S * N_3N_2 \dots \text{Term1}$$

Term1 can be evaluated immediately (using Word maths in base 65536).

$$[(T * N_3N_2) + N_3N_2 + N_1N_0] / D_1D_0 \dots \text{Term2}$$

Term2 is another division.

These two terms allow an iterative way of getting the final result. The procedure is:

**Step0:** Load the Accumulator with the Numerator.

**Step1:** Determine S and T, based purely on the

Divisor  $D_1D_0$ . The values of S and T are constant throughout the division process.

**Step2:** Set Running-Total to zero.

**Step3:** Evaluate Term1 and add to Running-Total.

**Step4:** Evaluate the Numerator of Term2 and load into the Accumulator.

If the Numerator is a value greater than two digits, then *Goto Step3* else evaluate Accumulator /  $D_1D_0$ . In base 65536 maths this just uses Word maths (because Accumulator value is now only two-digits). Add to Running-Total to get final result.

So this iterative method will give the result of  $N_3N_2N_1N_0 / D_1D_0$ . It uses normal PICAXE Word maths and the 32-bit addition routine.

In reality it won't take too many iterations to complete a division instruction, although this is an intensive and slow routine compared to the others.

## Speed Trials

The execution speed of each command depends heavily on the exact calculations being performed. However here are some example test results, based on: B = 64321, C = 1052, D = 761, E = 4523:

Expression	Approximate Execution Time
+B=A	60ms
+B-C=A	75ms
+B-C*D=A	100ms
+B-C*D/E=A	300ms

## Code Module

The routines for this virtual CPU fit into around 370 bytes of PICAXE Program memory and only a few bytes of RAM and data memory. So with, say, a PICAXE 18X it would be possible to include this code as a 'module' alongside your normal code. By stripping out the Decoder code and just calling the maths routines directly, you can just squeeze the code onto a PICAXE 08M!

## PICAXE Source Code

The following section gives the PICAXE code, which was developed fully using my PICAXE Simulator before hitting a real PICAXE.



## PICAXE 32-Bit Maths

---

```
!*****
!*      32-Bit (unsigned) maths on a PICAXE      *
!*      J.Leach 2006                               *
!*****

!*****
!****  PROCEDURE POINTERS  ****
!*****

Symbol Procedure0StartAddress = 0

!*****
!****  VIRTUAL PROGRAM MEMORY  ****
!*****

Eeprom 0, ("B-C*D/E=A")

!*****
!****  PICAXE VARIABLES  ****
!*****

'Word 0 (b0 and b1)
  Symbol OperandLSW = w0
  Symbol OperandLSWLSB = b0
  Symbol OperandLSWMSB = b1
'Word 1 (b2 and b3)
  Symbol T = w1
  Symbol Temp2Word = w1
  Symbol InstructionOperand = b2
'Word 2 (b4 and b5)
  Symbol AccumulatorLSW = w2
  Symbol AccumulatorLSWLSB = b4
  Symbol AccumulatorLSWMSB = b5
'Word 3 (b6 and b7)
  Symbol AccumulatorMSW = w3
'Word 4 (b8 and b9)
  Symbol S = w4
  Symbol Temp1Word = w4
'Word 5 (b10 and b11)
  Symbol OperandMSW = w5
  Symbol Address = b10
  Symbol Address1 = b10
  Symbol Address2 = b11
'Word 6 (b12 and b13)
  Symbol InstructionCode = b12
  Symbol Temp1Byte = b12
  Symbol ProgramCounter = b13
  Symbol Temp2Byte = b13
  Symbol ErrorFlag = b13
  Symbol Index = b13

!*****
!****  CONSTANTS  ****
!*****

Symbol LCDOutPin = 6 'LCD used for alerting, but other methods can be used.

'Addresses
Symbol RTStartAddress = 80
Symbol RTEndAddress = 83
Symbol DenominatorLSBAddress = 84
Symbol DenominatorMSBAddress = 85
Symbol ErrorFlagAddress = 86
Symbol GPRStartAddress = 87
Symbol ProgramCounterAddress = 97
'Error constants
```

## PICAXE 32-Bit Maths

---

```
Symbol ERROR_Overflow = 0
Symbol ERROR_NegativeResult = 1
Symbol ERROR_DivideByZero = 2
'Other
Symbol RTStartLessb4Address = 26 '(80 - 54)
Symbol GPRStartAddressLessb0Address = 37 '(87 - 50)

'*****
'**** MAIN ****
'*****

Main:
Serout LCDOutPin,N2400,(254,1) 'Clear LCD. NOT really part of this module.
Pause 5000

'Example calculation: (64321 - 1052 * 761 / 4523 )
w1 = 64321
w2 = 1052
w3 = 761
w4 = 4523
ProgramCounter = Procedure0StartAddress
Gosub ExecuteProcedure
End

'*****
'**** VIRTUAL ARITHMETIC AND LOGIC UNIT (ALU) ****
'*****

Add:
'Performs : Accumulator = Accumulator + Operand
'Jumps to error routine on overflow
'Uses both words of Operand (because used by Multiply and Divide routines)
'ON EXIT: Operand is same as on entry

Poke ErrorFlagAddress,ERROR_Overflow

'Add LSW
AccumulatorLSW = AccumulatorLSW + OperandLSW
If AccumulatorLSW >= OperandLSW Then Add_1
'Add Carry to MSW and jump to error routine if overflow
AccumulatorMSW = AccumulatorMSW + 1
If AccumulatorMSW = 0 Then CPU_Error

'Add MSW
Add_1:
AccumulatorMSW = AccumulatorMSW + OperandMSW
'Jump to error routine if overflow
If AccumulatorMSW < OperandMSW Then CPU_Error
Return

Subtract:
'Performs : Accumulator = Accumulator + OperandLSW
'Jumps to error routine if the result is less than zero
'Uses only the LSW of Operand
'ON EXIT: Operand is same as on entry

Poke ErrorFlagAddress,ERROR_NegativeResult

'Subtract LSW
Temp1Word = AccumulatorLSW
AccumulatorLSW = AccumulatorLSW - OperandLSW

If Temp1Word >= AccumulatorLSW Then Subtract_1
```

## PICAXE 32-Bit Maths

---

```
'Borrow from MSW and jump to error routine if this will make the overall result
'negative.
If AccumulatorMSW = 0 Then CPU_Error
AccumulatorMSW = AccumulatorMSW - 1

'Note: No need to Subtract MSW as only OperandLSW is being used
Subtract_1:
Goto Fetch

Multiply:
'Performs : Accumulator = Accumulator * OperandLSW
'Jumps to error routine on overflow
'Uses only the LSW of Operand
'ON EXIT: Operand is corrupted

Poke ErrorFlagAddress,ERROR_Overflow

'Calculate the higher multiple and keep in Accumulator
Temp1Word = AccumulatorLSW
Temp2Word = AccumulatorMSW
AccumulatorLSW = 0
AccumulatorMSW = OperandLSW * Temp2Word
Temp2Word = OperandLSW ** Temp2Word
'Check for overflow
If Temp2Word > 0 Then CPU_Error

'Calculate the lower multiple and put in Operand
OperandMSW = Temp1Word ** OperandLSW
OperandLSW = Temp1Word * OperandLSW

'Add the multiples to get the final result
Gosub Add
Goto Fetch

Divide:
'Performs : Accumulator = Accumulator / OperandLSW
'Jumps to error routine if divide by zero
'Uses only the LSW of Operand
'ON EXIT: Operand is corrupted

Poke ErrorFlagAddress,ERROR_DivideByZero

'Check for error
If OperandLSW = 0 Then CPU_Error

'Zero the Running Total
For Address = RTStartAddress To RTEndAddress
    Poke Address,0
Next

'Calculate the quotient and remainder for 65535/OperandLSW
S = 65535 / OperandLSW
T = 65535 // OperandLSW

'Store the denominator
Poke DenominatorLSBAddress,OperandLSWLSB
Poke DenominatorMSBAddress,OperandLSWMSB

Divide_1:
'Calculate S * AccumulatorMSW in the Operand, and add to Running Total.
'Note: uses variables very carefully !
OperandLSW = AccumulatorMSW
```

## PICAXE 32-Bit Maths

---

```
Gosub SwapAccumulatorWithRT
OperandMSW = S ** OperandLSW
OperandLSW = S * OperandLSW
Gosub Add
'Update the running total
Gosub SwapAccumulatorWithRT

'Calculate the new Numerator
OperandMSW = 0
OperandLSW = AccumulatorLSW + AccumulatorMSW
If OperandLSW > AccumulatorMSW Then Divide_2
OperandMSW = 1
Divide_2:
AccumulatorLSW = AccumulatorMSW * T
AccumulatorMSW = AccumulatorMSW ** T
Gosub Add

'Check to see if the new numerator is a single word. Loop back if it isn't
If AccumulatorMSW > 0 Then Divide_1

Temp1Word = AccumulatorLSW
'Retrieve the running total
Gosub SwapAccumulatorWithRT
'Retrieve the denominator
Peek DenominatorLSBAddress,OperandLSWLSB
Peek DenominatorMSBAddress,OperandLSWMSB
'Calculate AccumulatorLSW/Denominator and store in Operand
OperandLSW = Temp1Word / OperandLSW
OperandMSW = 0
'and add to the Running total to give the final result
Gosub Add
Goto Fetch

SwapAccumulatorWithRT:
'Swaps the Accumulator value with the Running Total value
For Address1 = RTStartAddress To RTEndAddress
    Address2 = Address1 - RTStartLessb4Address
    Peek Address1,Temp1Byte
    Peek Address2,Temp2Byte
    Poke Address1,Temp2Byte
    Poke Address2,Temp1Byte
Next
Return

StoreAccumulatorLSW:
'ON EXIT : The specified w0-w4 variable is loaded with the LSW of the
Accumulator.
'The 'Return' statement will end the use of the Virtual CPU
'and return to 'normal' PICAXE code.

Address = InstructionOperand - "A" * 2 + 50
Poke Address,AccumulatorLSWLSB
Address = Address + 1
Poke Address,AccumulatorLSWMSB
Return

'*****
'**** VIRTUAL CPU ROUTINES ****
'*****

CPU_Error:
    Peek ErrorFlagAddress,ErrorFlag
```

## PICAXE 32-Bit Maths

---

```
    Serout LCDOutPin,N2400,(254,128,"ERROR: ",#ErrorFlag," ")
End

ExecuteProcedure:
    'ON ENTRY: ProgramCounter has been set to the start of the Procedure

    'Save the Program Counter
    Poke ProgramCounterAddress,ProgramCounter 'Save the ProgramCounter

    'Load General Purpose registers with corresponding w0 to w4 values
    For Address1 = 50 To 59
        Address2 = Address1 + GPRStartAddressLessb0Address
        Peek Address1,Temp1Byte
        Poke Address2,Temp1Byte
    Next

    'Set the Accumulator to 0
    AccumulatorMSW = 0
    AccumulatorLSW = 0

    '*****
    '**** VIRTUAL DECODER ****
    '*****

Fetch:
    'Fetch an Instruction code and operand from Program Memory
    Peek ProgramCounterAddress,ProgramCounter 'Retrieve the ProgramCounter
    Read ProgramCounter,InstructionCode
    ProgramCounter = ProgramCounter + 1
    Read ProgramCounter,InstructionOperand
    ProgramCounter = ProgramCounter + 1
    Poke ProgramCounterAddress,ProgramCounter 'Save the ProgramCounter

    'Load the contents of the specified Register into OperandLSW
    Address = InstructionOperand - "A" * 2 + GPRStartAddress
    Peek Address,OperandLSWLSB
    Address = Address + 1
    Peek Address,OperandLSWMSB
    OperandMSW = 0

Execute:
    'Execute the loaded Instruction

    Lookdown InstructionCode,("+","-","*","/","="),Index
    Branch Index,(Instruction_Add,Subtract,Multiply,Divide,StoreAccumulatorLSW)

Instruction_Add:
    Gosub Add
    Goto Fetch
```