SpinLMM

Version 1.0a

October 1, 2010

Note: This version of the SpinLMM document corrects an error in Version 1.0 where pcurr was used instead of dcurr to reference the Spin stack pointer.

Introduction

SpinLMM is an enhancement to the Spin interpreter that provides the capability to execute PASM instructions using the large memory model (LMM). LMM was created by Bill Henning to provide a way to run PASM programs from the hub memory of the Propeller. The large memory model uses an interpreter that runs in a cog to fetch an instruction from the hub RAM and execute it in place. This is feasible because the carry and zero flags are not normally modified unless they are explicitly defined in the PASM instruction.

SpinLMM can be used to implement PASM routines that normally run in a seperate cog from the Spin program. There are many existing applications where PASM routines use up an entire cog, but they are called infrequently and/or seqeuentially, and the parallel execution feature of a seperate cog is not needed. Some examples of this may be I2C access, floating-point operations or half-duplex serial operation.

Programming for SpinLMM requires a basic understanding of PASM. A beginner programmer should first become familiar with programming in PASM using multiple cogs before working with SpinLMM. However, in some respects SpinLMM is simpler than multi-cog PASM because of the direct interface to the Spin code through the stack. SpinLMM uses the @@@ and Bytecode extensions to the Spin language that are available in the BST and homespun compilers. The Parallax Spin Tool does not support these extensions, so either BST or homespun must be used with SpinLMM.

The Basics

An example of a simple SpinLMM program that returns a value of 5 is given below.

```
OBJ
  lmm : "SpinLMM"

CON
  reg0  = lmm#reg0
  FRETX = lmm#FRETX

PUB main
  lmm.start                          ' Start the LMM PASM interpreter

  result := lmm.run0(@GetFive)       ' Run the LMM PASM routine located at GetFive

DAT
GetFive  mov    reg0, #5            ' Move 5 into the reg0 register
         jmp    #FRETX              ' Return value of reg0 to the Spin interpreter
```

The call to the lmm.start method installs the LMM PASM interpreter into the Spin interpreter. The lmm.run0 method invokes the LMM PASM interpreter starting at GetFive. The **jmp #FRETX** instruction pushes the value of the reg0 register onto the stack and returns control to the Spin interpreter.

Parameters are passed to LMM routines by pushing them onto the Spin stack. The LMM PASM routine must pop them off of the stack to access them. The following program shows a routine that adds two numbers and returns the sum to the Spin

program.

```
OBJ
  lmm : "SpinLMM"

CON
  reg0  = lmm#reg0
  reg1  = lmm#reg1
  dcurr = lmm#dcurr
  FRETX = lmm#FRETX

PUB main
  lmm.start                          ' Start the LMM PASM interpreter

  result := lmm.run2(@AddTwo, 1, 2) ' Run the LMM PASM AddTwo using two parameters

DAT
AddTwo   sub    dcurr, #4           ' Decrement the stack pointer
         rdlong reg0, dcurr         ' Get the first parameter
         sub    dcurr, #4           ' Decrement the stack pointer
         rdlong reg1, dcurr         ' Get the second parameter
         add    reg0, reg1          ' Add them together
         jmp    #FRETX              ' Return the result to the Spin interpreter
```

The stack address is contained in the dcurr register that is used by the Spin interpreter. The lmm.run2 method pushes the two parameters onto the stack, and the LMM routine pops them off of the stack by decrementing the dcurr register and reading the values. The registers reg0, reg1 and dcurr are all pre-defined in the SpinLMM interpreter. A complete list of the available registers is given later in this document.

Branching

LMM PASM programs cannot branch the same way a normal PASM program branches because it is executed by an interpreter and not directly by the processor. Instruction addresses in an LMM PASM program refer to hub memory, and not cog memory. The LMM interpreter uses a small loop to execute each PASM instruction one at a time. The loop looks like this.

```
lmm_loop       rdlong  instruct, lmm_pc
               add lmm_pc, #4
instruct       nop
               jmp #lmm_loop
```

An instruction is read from the address in the lmm_pc register, and then executed after incrementing the lmm_pc register by four. The interpreter then jumps back to the beginning of the loop to fetch the next instruction. The instructions in this loop total to 5 instruction cycles, or 20 system cycles. However, three additional instruction cycles are required to align with the memory access window of the cog. This results in a total of eight instruction cycles, or 32 system cycles to execute each LMM PASM instruction. This is an 8:1 speed reduction compared to straight PASM.

This loop could be unrolled several times to approach an average of 16 cycles per instruction, with only a 4:1 reduction in speed of straight PASM. However, the SpinLMM interpreter only uses the four-instruction loop. There is actually no speed penalty for hub-access instructions when using the four-instruction loop versus an unrolled loop. This is because hub-access instructions, such as rdlong and wrlong can execute within the extra cycles needed for hub access window alignment.

Branching in an LMM PASM program is performed by manipulating the program counter, or lmm_pc register in the interpreter. The interpreter adds 4 to the lmm_pc immediately after the instruction is read. At the time when an LMM PASM instruction is executed the lmm_pc register is pointing to the next long location. An example of a routine that performs branching is shown below. This routine computes the sum of the integer numbers from 1 to N using a small loop.

```
DAT
```

```
SumIt    sub    dcurr, #4          ' Decrement the stack pointer
         rdlong reg1, dcurr        ' Get the value of N
         mov    reg0, #0           ' Initialize sum to zero
LoopIt   add    reg0, reg1         ' Add number to sum
         sub    reg1, #1       wz  ' Decrement the loop counter
   if_nz rdlong lmm_pc, lmm_pc     ' Set program counter to value of next long if not zero
         long   @@@LoopIt          ' Absolute address of beginning of loop
         jmp    #FRETX             ' Return the result to the Spin interpreter
```

This program performs a jump back to the beginning of the loop by loading the address of LoopIt into the lmm_pc register. Notice that the addess is stored immediately after the **rdlong lmm_pc, lmm_pc** instruction by using **long @@@LoopIt**. The **@@@** operator generates the absolute address of LoopIt. This operator is only supported by the BST and homespun compilers. The Parallax Spin Tool will not be able to correctly compile this code.

There are various branching instructions that cannot be efficiently implemented directly in LMM PASM such as the djnz and call instructions. Special psuedo-ops have been created to facilitate these branches. LMM psuedo-ops are just small routines within the interpreter that the LMM PASM program can jump to. FRETX is a psuedo-op that returns control to the Spin interpreter. Other psuedo-ops are defined in the next section.

Pseudo-ops

FJMP

The FJMP pseudo-op causes the execution to jump to the location specified in the next long after the FJMP instruction. The FJMP pseudo-op sets the LMM program counter to this value by executing a **rdlong lmm_pc, lmm_pc** instruction. It can be used with the jmp instruction, and it is especially useful in conjuction with the djnz instruction as shown below.

```
LoopIt   add    reg0, reg1         ' Add number to sum
         djnz   reg1, #FJMP        ' Decrement number and loop if not zero
         long   @@@LoopIt          ' Absolute address of beginning of loop
```

FCALL

FCALL is used to implement a subroutine call. The FCALL pseudo-op saves the program counter in the dedicated lmm_ret register, and then jumps to the FJMP pseudo-op. If the called routine also uses the FCALL pseudo-op it must save the lmm_ret register so that it can be used later when returning. A return is performed by copying the lmm_ret register to the program counter, lmm_pc. An example of using the FCALL pseudo-op and performing a return is shown below.

```
              jmp #FCALL          ' Call the routine at sub1
              long @@@sub1

sub1          .....

              mov lmm_pc, lmm_ret ' Return to the caller
```

ICALL

The ICALL pseudo-op is used to perform an indirect, or indexed jump. It is normally used in conjunction with a calling table. It is similar to FCALL except that the value of the reg7 register is added to the contents of the next long to determine the jump address. It is implemented with a **rdlong lmm_pc, lmm_pc** instruction followed by an **add lmm_pc, reg7**, which is then followed by another **rdlong lmm_pc, lmm_pc**. An example of using ICALL is shown below.

```
              mov reg7, #(2*4)     ' Index to call_table[2]
              jmp #ICALL           ' Execute the ICALL pseudo-op
              long @@@call_table
```

```
                ...

call_table      long @@@sub1
                long @@@sub2
                long @@@sub3          ' Calls sub3 at this address
                long @@@sub4
```

This code will cause sub3 to be called. Note that the index address is the sum of the long after the jmp #ICALL instruction and the contents of register reg7. Therefore, it would also work to store the address of the call table in reg7, and store the table offset after the jmp #ICALL instruction. This is useful for driver code that implements a set of basic functions, but each instance of the driver uses different code for the basic functions.

The ICALL peusdo-op can be used to perform a simple indirect call if either the immediate value or reg7 is always zero. In this case, the call table is just a single long that contains the indirect address.

IJMP

IJMP is similar to ICALL except that a return address is not stored in lmm_ret. The IJMP pseudo-op performs indexed jumps in the same way that ICALL does.

FRET

This pseudo-op returns control back to the Spin interpreter. It does not push a return value onto the stack.

FRETX

This pseudo-op pushes the contents of reg0 onto the stack and returns control back to the Spin interpreter.

FCACHE

The FCACHE pseudo-op is used to load a section of code into the cog's memory and then jump to it. The cache area consists of 16 consecutive long locations, and it uses the same space as reg8 through reg23. reg8 is aliased with the label cache_addr, which can be used to set the cog address with the PASM org instruction. The instructions to be cached must be terminated by a zero long value, and it must contain a jump back to the LMM or Spin interpreters.

Note, the FCACHE is the only pseudo-op that changes the status flags. The FCACHE sets the zero flag before jumping to the cache. The carry flag is unaffected. The cache area is 16 longs in size, and the cached loop must fit within this space. This includes the terminating zero at the end of the loop. An example using the FCACHE pseudo-op with the SumIt routine is shown below.

```
DAT
SumIt    sub    dcurr, #4          ' Backup the stack pointer
         rdlong reg1, dcurr        ' Get the value of N
         mov    reg0, #0           ' Initialize sum to zero
         jmp    #FCACHE            ' Jump to the FCACHE pseudo-op code
         org    cache_addr         ' The code origin must start at cache_addr
LoopIt   add    reg0, reg1         ' Add number to sum
         djnz   reg1, #LoopIt      ' Use a normal PASM jump instruction in cached code
         jmp    #FRETX             ' Must exit cached code with a jump
         long   0                  ' Terminating zero long
```

LMM_LOOP

This is the starting address of the LMM interpreter loop. An FCACHE program can continue normal execution by jumping to this location.

Registers

There are several registers defined in the Spin and LMM interpreters. Most of the registers are used for general operations. Some of the registers have special functions, such as reg0, reg7, dcurr, lmm_pc and lmm_ret. The registers are defined below.

lmm_pc Program Counter
lmm_ret Return Address set by **FCALL** and **ICALL**
dcurr The Spin stack pointer
reg0 General use and used with **FRETX**
reg1 General use
reg2 General use
reg3 General use
reg4 General use

reg5 General use
reg6 General use
reg7 General use and used with **ICALL** and **IJMP**
reg8 General use and **FCACHE**
reg9 General use and **FCACHE**
reg10 General use and **FCACHE**
reg11 General use and **FCACHE**
reg12 General use and **FCACHE**
reg13 General use and **FCACHE**
reg14 General use and **FCACHE**
reg15 General use and **FCACHE**
reg16 General use and **FCACHE**
reg17 General use and **FCACHE**
reg18 General use and **FCACHE**
reg19 General use and **FCACHE**
reg20 General use and **FCACHE**
reg21 General use and **FCACHE**
reg22 General use and **FCACHE**
reg23 General use and **FCACHE**
