ARM PROGRAMMING USING ASSEMBLY LANGUAGE

WRITING ASSEMBLY CODE:

This section gives examples showing how to write basic assembly code. Also, this section uses the *ARM macro assembler* ***armasm*** for examples.

Example 1:

This example shows how to convert a *C* function to an assembly function—usually the first stage of assembly optimization. Consider the simple *C* program *main.c* following that prints the squares of the integers from 0 to 9:

```c
#include <stdio.h>

int square(int i);

int main(void)
{
  int i;

  for (i=0; i<10; i++)
  {
    printf("Square of %d is %d\n", i, square(i));
  }
}

int square(int i)
{
  return i*i;
}
```

Let's see how to replace square by an assembly function that performs the same action. Remove the *C* definition of square, but not the declaration (the second line) to produce a new *C* file *main1.c*. Next add an *armasm* assembler file *square.s* with the following contents:

```
        AREA    |.text|, CODE, READONLY

        EXPORT  square

        ; int square(int i)
square

        MUL     r1, r0, r0      ; r1 = r0 * r0
        MOV     r0, r1          ; r0 = r1
        MOV     pc, lr          ; return r0
        END
```

- The *AREA* directive names the area or code section that the code lives in. If you use non-alphanumeric characters in a symbol or area name, then enclose the name in vertical bars. Many non-alphanumeric characters have special meanings otherwise. In the previous code we define a read-only code area called *.text*.
- The *EXPORT* directive makes the symbol square available for external linking. At line six we define the symbol square as a code label. Note that *armasm* treats non-indented text as a label definition.
- When square is called, the parameter passing is defined by the *ARM-Thumb procedure call standard (ATPCS)*. The input argument is passed in register *r0*, and the return value is returned in register *r0*. The multiply instruction has a restriction that the destination register must not be the same as the first argument register. Therefore we place the multiply result into *r1* and move this to *r0*.
- The *END* directive marks the end of the assembly file. Comments follow a semicolon.

The following script illustrates how to build this example using command line tools.

```
armcc -c main1.c
armasm square.s
armlink -o main1.axf main1.o square.o
```

Example 1 only works if you are compiling your *C* as ARM code. If you compile your *C* as Thumb code, then the assembly routine must return using a *BX* instruction.

Example 2: When calling ARM code from *C* compiled as Thumb, the only change required to the assembly in *Example 1* is to change the return instruction to a *BX*. *BX* will return to ARM or Thumb state according to bit *0* of *lr*. Therefore this routine can be called from ARM or Thumb. Use *BX lr* instead of *MOV pc, lr* whenever your processor supports *BX* (*ARMv4T* and above). Create a new assembly file *square2.s* as follows:

```
        AREA    |.text|, CODE, READONLY

        EXPORT  square

        ; int square(int i)
square
        MUL   r1, r0, r0    ; r1 = r0 * r0
        MOV   r0, r1        ; r0 = r1
        BX    lr            ; return r0

        END
```

With this example we build the *C* file using the Thumb *C* compiler *tcc*. We assemble the assembly file with the interworking flag enabled so that the linker will allow the Thumb *C* code to call the ARM assembly code. You can use the following commands to build this example:

```
tcc -c main1.c
armasm -apcs /interwork square2.s
armlink -o main2.axf main1.o square2.o
```

Example 3: This example shows how to call a subroutine from an assembly routine. We will take Example 1 and convert the whole program (including main) into assembly. We will call the *C* library routine *printf* as a subroutine. Create a new assembly file *main3.s* with the following contents:

```
        AREA    |.text|, CODE, READONLY

        EXPORT  main

        IMPORT  |Lib$$Request$$armlib|, WEAK
        IMPORT  __main    ; C library entry
        IMPORT  printf    ; prints to stdout
i       RN 4

         ; int main(void)
  main
         STMFD   sp!, {i, lr}
         MOV     i, #0
  loop
         ADR     r0, print_string
         MOV     r1, i
         MUL     r2, i, i
         BL      printf
         ADD     i, i, #1
         CMP     i, #10
         BLT     loop
         LDMFD   sp!, {i, pc}

  print_string
         DCB      "Square of %d is %d\n", 0

         END
```

- The *IMPORT* directive is used to declare symbols that are defined in other files.
- The imported symbol *Lib$$Request$$armlib* makes a request that the linker links with the standard ARM *C* library.
    - The *WEAK* specifier prevents the linker from giving an error if the symbol is not found at link time. If the symbol is not found, it will take the value zero.
- The second imported symbol____*main* is the start of the *C* library initialization code.

You only need to import these symbols if you are defining your own main; a main defined in *C* code will import these automatically for you. Importing *printf* allows us to call that *C* library function.

- The *RN* directive allows us to use names for registers. In this case we define *i* as an alternate name for register *r4*.

- Using register names makes the code more readable. It is also easier to change the allocation of variables to registers at a later date. Recall that *ATPCS* states that a function must preserve registers *r4* to *r11* and *sp*. We corrupt *i* (*r4*), and calling *printf* will corrupt *lr*. Therefore we stack these two registers at the start of the function using an *STMFD* instruction. The *LDMFD* instruction pulls these registers from the stack and returns by writing the return address to *pc*.

- The *DCB* directive defines byte data described as a string or a comma-separated list of bytes.

To build this example you can use the following command line script:

```
armasm main3.s
armlink -o main3.axf main3.o
```

Note that Example 3 also assumes that the code is called from ARM code. If the code can be called from Thumb code as in Example 2 then we must be capable of returning to Thumb code. For architectures before *ARMv5* we must use a *BX* to return. Change the last instruction to the two instructions:

```
LDMFD   sp!, {i, lr}
BX      lr
```

Example 4: This example defines a function *sumof* that can sum any number of integers. The arguments are the number of integers to sum followed by a list of the integers. The *sumof* function is written in assembly and can accept any number of arguments. Put the *C* part of the example in a file *main4.c*:

```
#include <stdio.h>

/* N is the number of values to sum in list ... */
int sumof(int N, ...);

int main(void)
{
   printf("Empty sum=%d\n", sumof(0));
   printf("1=%d\n", sumof(1,1));
   printf("1+2=%d\n", sumof(2,1,2));
   printf("1+2+3=%d\n", sumof(3,1,2,3));
   printf("1+2+3+4=%d\n", sumof(4,1,2,3,4));
   printf("1+2+3+4+5=%d\n", sumof(5,1,2,3,4,5));
   printf("1+2+3+4+5+6=%d\n", sumof(6,1,2,3,4,5,6));
}
```

Next define the *sumof* function in an assembly file *sumof.s*:

```
        AREA    |.text|, CODE, READONLY

        EXPORT  sumof

N       RN 0    ; number of elements to sum
sum     RN 1    ; current sum

        ; int sumof(int N, ...)
sumof
        SUBS    N, N, #1        ; do we have one element
        MOVLT   sum, #0         ; no elements to sum!
        SUBS    N, N, #1        ; do we have two elements
        ADDGE   sum, sum, r2
        SUBS    N, N, #1        ; do we have three elements
        ADDGE   sum, sum, r3
        MOV     r2, sp          ; top of stack
loop
        SUBS    N, N, #1        ; do we have another element
        LDMGEFD r2!, {r3}       ; load from the stack

        ADDGE   sum, sum, r3
        BGE     loop
        MOV     r0, sum
        MOV     pc, lr          ; return r0

        END
```

The code keeps count of the number of remaining values to *sum*, *N*. The first three values are in registers *r1*, *r2*, *r3*. The remaining values are on the stack (Recall that *ATPCS* places the first four arguments in registers r0 to r3. Subsequent arguments are placed on the stack). You can build this example using the commands –

```
armcc -c main4.c
armasm sumof.s
armlink -o main4.axf main4.o sumof.o
```

PROFILING AND CYCLE COUNTING:

- ✓ The first stage of any optimization process is to identify the critical routines and measure their current performance. A ***profiler*** is a tool that measures the proportion of time or processing cycles spent in each subroutine. You use a profiler to identify the most critical routines.

- ✓ A ***cycle counter*** measures the number of cycles taken by a specific routine. You can measure your success by using a cycle counter to benchmark a given subroutine before and after an optimization.
- ✓ The ARM simulator used by the *ADS1.1 debugger* is called the *ARMulator* and provides profiling and cycle counting features.
  - o The *ARMulator profiler* works by sampling the program counter *pc* at regular intervals. The profiler identifies the function the *pc* points to and updates a hit counter for each function it encounters. Another approach is to use the trace output of a simulator as a source for analysis.
  - o The accuracy of a *pc*-sampled profiler is limited, as it can produce meaningless results if it records too few samples.
- ✓ ARM implementations do not normally contain cycle-counting hardware; so to easily measure cycle counts you should use an ARM debugger with ARM simulator.
  - o You can configure the *ARMulator* to simulate a range of different ARM cores and obtain cycle count benchmarks for a number of platforms.

INSTRUCTION SCHEDULING:

The time taken to execute instructions depends on the implementation pipeline. For this section, we assume *ARM9TDMI* pipeline timings. The following rules summarize the cycle timings for common instruction classes on the *ARM9TDMI*.

Instructions that are conditional on the value of the ARM condition codes in the *cpsr* take one cycle if the condition is not met. If the condition is met, then the following rules apply:

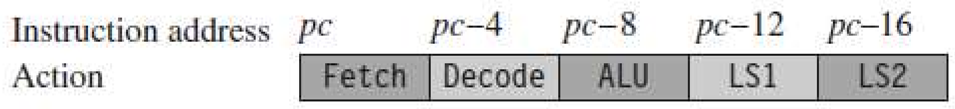
- ✓ *ALU* operations such as addition, subtraction, and logical operations take one cycle.
- ✓ This includes a shift by an immediate value. If you use a register-specified shift, then add one cycle. If the instruction writes to the *pc*, then add two cycles.
- Load instructions that load *N* 32-bit words of memory such as *LDR* and *LDM* take *N* cycles to issue, but the result of the last word loaded is not available on the following cycle.
  - o The updated load address is available on the next cycle. This assumes zero-wait-state memory for an un-cached system, or a cache hit for a cached system. An *LDM* of a single value is exceptional, taking two cycles. If the instruction loads *pc*, then add two cycles.
  - o Load instructions that load 16-bit or 8-bit data such as *LDRB*, *LDRSB*, *LDRH*, and *LDRSH* take one cycle to issue. The load result is not available on the following two cycles. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an un-cached system, or a cache hit for a cached system.
- Branch instructions take three cycles.

- Store instructions that store *N* values take *N* cycles. This assumes zero-wait-state memory for an un-cached system, or a cache hit or a write buffer with *N* free entries for a cached system. An *STM* of a single value is exceptional, taking two cycles.
- Multiply instructions take a varying number of cycles depending on the value of the second operand in the product.

To understand how to schedule code efficiently on the ARM, we need to understand the ARM pipeline and dependencies. The *ARM9TDMI* processor performs five operations in parallel:

- ***Fetch:*** Fetch from memory the instruction at address *pc*. The instruction is loaded into the core and then processes down the core pipeline.
- ***Decode:*** Decode the instruction that was fetched in the previous cycle. The processor also reads the input operands from the register bank if they are not available via one of the forwarding paths.
- ***ALU:*** Executes the instruction that was decoded in the previous cycle. Note this instruction was originally fetched from address $pc - 8$ (ARM state) or $pc - 4$ (Thumb state).
    - Normally this involves calculating the answer for a data processing operation, or the address for a load, store, or branch operation.
    - Some instructions may spend several cycles in this stage. For example, multiply and register-controlled shift operations take several ALU cycles.
- ***LS1:*** Load or store the data specified by a load or store instruction. If the instruction is not a load or store, then this stage has no effect.
- ***LS2:*** Extract and zero- or sign-extend the data loaded by a byte or half-word load instruction. If the instruction is not a load of an 8-bit byte or 16-bit half-word item, then this stage has no effect.

The following Figure shows a simplified functional view of the five-stage *ARM9TDMI* pipeline.



Note that multiply and register shift operations are not shown in the figure.
After an instruction has completed the five stages of the pipeline, the core writes the result to the register file. Note that *pc* points to the address of the instruction being fetched. The ALU is executing the instruction that was originally fetched from address $pc - 8$ in parallel with fetching the instruction at address *pc*.

How does the pipeline affect the timing of instructions? Consider the following examples. These examples show how the cycle timings change because an earlier instruction must complete a stage before the current instruction can progress down the pipeline.
If an instruction requires the result of a previous instruction that is not available, then the processor stalls. This is called a *pipeline hazard* or *pipeline interlock*.

Example 5: This example shows the case where there is no interlock.

```
ADD   r0, r0, r1
ADD   r0, r0, r2
```

This instruction pair takes two cycles. The ALU calculates *r0* + *r1* in one cycle. Therefore this result is available for the ALU to calculate *r0* + *r2* in the second cycle.

Example 6: This example shows a one-cycle interlock caused by load use.

```
LDR   r1, [r2, #4]
ADD   r0, r0, r1
```

This instruction pair takes three cycles. The ALU calculates the address *r2* + *4* in the first cycle while decoding the *ADD* instruction in parallel. However, the *ADD* cannot proceed on the second cycle because the load instruction has not yet loaded the value of *r1*. Therefore the pipeline stalls for one cycle while the load instruction completes the *LS1* stage. Now that *r1* is ready, the processor executes the *ADD* in the ALU on the third cycle.
The following Figure illustrates how this interlock affects the pipeline.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	...	ADD	LDR	...	
Cycle 2		...	*ADD*	LDR	...
Cycle 3		...	ADD	—	LDR

The processor stalls the *ADD* instruction for one cycle in the ALU stage of the pipeline while the load instruction completes the *LS1* stage. Figure denotes this stall by *italic ADD*. Since the *LDR* instruction proceeds down the pipeline, but the *ADD* instruction is stalled, a gap opens up between them. This gap is sometimes called a pipeline ***bubble***. We've marked the bubble with a *dash*.

Example 7: This example shows a one-cycle interlock caused by delayed load use.

```
LDRB   r1, [r2, #1]
ADD    r0, r0, r2
EOR    r0, r0, r1
```

This instruction triplet takes four cycles. Although the *ADD* proceeds on the cycle following the load byte, the *EOR* instruction cannot start on the third cycle. The *r1* value is not ready until the load instruction completes the *LS2* stage of the pipeline. The processor stalls the *EOR* instruction for one cycle. Note that the *ADD* instruction does not affect the timing at all. The sequence takes four cycles whether it is there or not! The following Figure shows how this sequence progresses through the processor pipeline. The *ADD* doesn't cause any stalls since the *ADD* does not use *r1*, the result of the load.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	EOR	ADD	LDRB	...	
Cycle 2	...	EOR	ADD	LDRB	...
Cycle 3		...	*EOR*	ADD	LDRB
Cycle 4		...	EOR	—	ADD

Example 8: This example shows why a branch instruction takes three cycles. The processor must flush the pipeline when jumping to a new address.

```
        MOV   r1, #1
        B     case1
        AND   r0, r0, r1
        EOR   r2, r2, r3
        ...
case1
        SUB   r0, r0, r1
```

The three executed instructions take a total of five cycles. The *MOV* instruction executes on the first cycle. On the second cycle, the branch instruction calculates the destination address. This causes the core to flush the pipeline and refill it using this new *pc* value. The refill takes two cycles. Finally, the *SUB* instruction executes normally. The following Figure illustrates the pipeline state on each cycle. The pipeline drops the two instructions following the branch when the branch takes place.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	AND	B	MOV	...	
Cycle 2	EOR	AND	B	MOV	...
Cycle 3	SUB	—	—	B	MOV
Cycle 4	...	SUB	—	—	B
Cycle 5		...	SUB	—	—

Scheduling of Load Instructions:

Load instructions occur frequently in compiled code, accounting for approximately one-third of all instructions. Careful scheduling of load instructions so that pipeline stalls don't occur can improve performance. The compiler attempts to schedule the code as best it can, but the aliasing problem of *C* limits the available optimizations. The compiler cannot move a load instruction before a store instruction unless it is certain that the two pointers used do not point to the same address.

Consider an example of a memory-intensive task. The following function, *str_tolower*, copies a zero-terminated string of characters from *in* to *out*. It converts the string to lowercase in the process.

```
void str_tolower(char *out, char *in)
{
  unsigned int c;

  do
  {
    c = *(in++);
    if (c>='A' && c<='Z')
    {
      c = c + ('a' -'A');
    }
    *(out++) = (char)c;
  } while (c);
}
```

```
str_tolower
        LDRB    r2,[r1],#1    ; c = *(in++)
        SUB     r3,r2,#0x41   ; r3 = c -'A'
        CMP     r3,#0x19      ; if (c <='Z'-'A')
        ADDLS   r2,r2,#0x20   ;    c +='a'-'A'
        STRB    r2,[r0],#1    ; *(out++) = (char)c
        CMP     r2,#0         ; if (c!=0)
        BNE     str_tolower   ;    goto str_tolower
        MOV     pc,r14        ; return
```

The compiler generates the above compiled output. Notice that the compiler optimizes the condition (*c >= 'A' && c <= 'Z'*) to the check that *0 <= c-'A' <= 'Z'-'A'*. The compiler can perform this check using a single unsigned comparison.

Unfortunately, the *SUB* instruction uses the value of *c* directly after the *LDRB* instruction that loads *c*. Consequently, the *ARM9TDMI* pipeline will stall for two cycles. The compiler can't do any better since everything following the load of *c* depends on its value.

However, there are two ways you can alter the structure of the algorithm to avoid the cycles by using assembly. We call these methods load scheduling by *preloading* and *unrolling*.

» *Load Scheduling by Preloading & Load Scheduling by Unrolling – Self Study*.

REGISTER ALLOCATION:

You can use 14 of the 16 visible ARM registers to hold general-purpose data. The other two registers are the *stack pointer, r13,* and the *program counter, r15*. For a function to be *ATPCS* compliant it must preserve the callee values of registers *r4* to *r11*. *ATPCS* also specifies that the stack should be eight-byte aligned; therefore you must preserve this alignment if calling subroutines. Use the following template for optimized assembly routines requiring many registers:

```
routine_name
        STMFD sp!,        {r4-r12, lr}       ; stack saved registers
          ; body of routine
          ; the fourteen registers r0-r12 and lr are available
        LDMFD sp!,        {r4-r12, pc}       ; restore registers and return
```

The only purpose in stacking *r12* is to keep the stack eight-byte aligned. You need not stack *r12* if your routine doesn't call other *ATPCS* routines. For *ARMv5* and above you can use the preceding template even when being called from Thumb code. If your routine may be called from Thumb code on an *ARMv4T* processor, then modify the template as follows:

```
routine_name
        STMFD sp!,       {r4-r12, lr}       ; stack saved registers
          ; body of routine
          ; registers r0-r12 and lr available
        LDMFD sp!,       {r4-r12, lr}       ; restore registers
        BX          lr                      ; return, with mode switch
```

Allocating Variables to Register Numbers:

When you write an assembly routine, it is best to start by using names for the variables, rather than explicit register numbers. This allows you to change the allocation of variables to register numbers easily. You can even use different register names for the same physical register number when their use doesn't overlap. Register names increase the clarity and readability of optimized code.

For the most part ARM operations are orthogonal with respect to register number. In other words, specific register numbers do not have specific roles. If you swap all occurrences of two registers *Ra* and *Rb* in a routine, the function of the routine does not change.

However, there are several cases where the physical number of the register is important:

- ✓ *Argument registers:* The *ATPCS* convention defines that the first four arguments to a function are placed in registers *r0* to *r3*. Further arguments are placed on the stack.
    - o The return value must be placed in *r0*.
- ✓ *Registers used in a load or store multiple:* Load and store multiple instructions *LDM* and *STM* operate on a list of registers in order of ascending register number. If *r0* and *r1* appear in the register list, then the processor will always load or store *r0* using a lower address than *r1* and so on.
- ✓ *Load and store double word:* The *LDRD* and *STRD* instructions introduced in *ARMv5E* operate on a pair of registers with sequential register numbers, *Rd* and *Rd + 1*. Furthermore, *Rd* must be an even register number.

Using More Than 14 Local Variables:

If you need more than 14 local 32-bit variables in a routine, then you must store some variables on the stack. The standard procedure is to work outwards from the innermost loop of the algorithm, since the innermost loop has the greatest performance impact.

Making the Most of Available Registers:

On load-store architecture such as the ARM, it is more efficient to access values held in registers than values held in memory. There are several tricks you can use to fit several sub-32-bit length variables into a single 32-bit register and thus can reduce code size and increase performance.

CONDITIONAL EXECUTION:

The processor core can conditionally execute most ARM instructions. This conditional execution is based on one of 15 condition codes. If you don't specify a condition, the assembler defaults to execute always condition (AL). The other 14 conditions split into seven pairs of complements. The conditions depend on the four condition code flags *N*, *Z*, *C*, *V* stored in the *cpsr* register.

By default, ARM instructions do not update the *N*, *Z*, *C*, *V* flags in the ARM *cpsr*. For most instructions, to update these flags you append an *S* suffix to the instruction mnemonic.

Exceptions to this are comparison instructions that do not write to a destination register. Their sole purpose is to update the flags and so they don't require the *S* suffix.

By combining conditional execution and conditional setting of the flags, you can implement simple if statements without any need for branches. This improves efficiency since branches can take many cycles and also reduces code size.

Example 17: The following *C* code converts an unsigned integer $0 \leq i \leq 15$ to a hexadecimal character *c*:

```
if (i<10)
{
  c = i + '0';
}
else
{
  c = i + 'A'-10;
}
```

We can write this in assembly using conditional execution rather than conditional branches:

```
CMP      i, #10
ADDLO    c, i, #'0'
ADDHS    c, i, #'A'-10
```

The sequence works since the first *ADD* does not change the condition codes. The second *ADD* is still conditional on the result of the compare.

Conditional execution is even more powerful for cascading conditions.

xample 18: The following *C* code identifies if *c* is a vowel:

```
if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
{
    vowel++;
}
```

In assembly you can write this using conditional comparison:

```
TEQ     c, #'a'
TEQNE   c, #'e'
TEQNE   c, #'i'
TEQNE   c, #'o'
TEQNE   c, #'u'
ADDEQ   vowel, vowel, #1
```

As soon as one of the *TEQ* comparisons detects a match, the *Z* flag is set in the *cpsr*. The following *TEQNE* instructions have no effect as they are conditional on *Z = 0*. The next instruction to have effect is the *ADDEQ* that increments vowel. You can use this method whenever all the comparisons in the if statement are of the same type.

Example 19: Consider the following code that detects if c is a letter:

```
if ((c>='A' && c<='Z') || (c>='a' && c<='z'))
{
    letter++;
}
```

To implement this efficiently, we can use an addition or subtraction to move each range to the form $0 \leq c \leq limit$. Then we use unsigned comparisons to detect this range and conditional comparisons to chain together ranges. The following assembly implements this efficiently:

```
SUB     temp, c, #'A'
CMP     temp, #'Z'-'A'
SUBHI   temp, c, #'a'
CMPHI   temp, #'z'-'a'
ADDLS   letter, letter, #1
```

Note that the logical operations *AND* and *OR* are related by the standard logical relations as shown in the following Table. You can invert logical expressions involving *OR* to get an expression involving *AND*, which can often be useful in simplifying or rearranging logical expressions.

Inverted expression	Equivalent
`!(a && b)`	`(!a) || (!b)`
`!(a || b)`	`(!a) && (!b)`

LOOPING CONSTRUCTS:

Most routines critical to performance will contain a loop. Note that, ARM loops are fastest when they count down towards zero. This section describes how to implement these loops efficiently in assembly. We also look at examples of how to unroll loops for maximum performance.

Decremented Counted Loops:

For a decrementing loop of *N* iterations, the loop counter *i* counts down from *N* to *1* inclusive. The loop terminates with *i = 0*. An efficient implementation is

```
        MOV i, N
loop
        ; loop body goes here and i=N,N-1,...,1
        SUBS i, i, #1
        BGT  loop
```

The loop overhead consists of a subtraction setting the condition codes followed by a conditional branch. On *ARM7* and *ARM9* this overhead costs four cycles per loop. If *i* is an array index, then you may want to count down from *N−1* to *0* inclusive instead so that you can access array element zero. You can implement this in the same way by using a different conditional branch:

```
        SUBS i, N, #1
loop
        ; loop body goes here and i=N-1,N-2,...,0
        SUBS i, i, #1
        BGE  loop
```

In this arrangement the *Z* flag is set on the last iteration of the loop and cleared for other iterations. If there is anything different about the last loop, then we can achieve this using the *EQ* and *NE* conditions. For example, if you preload data for the next loop, then you want to avoid the preload on the last loop. You can make all preload operations conditional on *NE*.

There is no reason why we must decrement by one on each loop. Suppose we require *N/3* loops; rather than attempting to divide *N* by three, it is far more efficient to subtract three from the loop counter on each iteration:

```
        MOV i, N
loop
        ; loop body goes here and iterates (round up)(N/3) times
        SUBS i, i, #3
        BGT  loop
```

Unrolled Counted Loops:

Loop unrolling reduces the loop overhead by executing the loop body multiple times. However, there are problems to overcome.

Multiple Nested Loops:

How many loop counters does it take to maintain multiple nested loops? Actually, one will suffice—or more accurately, one provided the sum of the bits needed for each loop count does not exceed 32. We can combine the loop counts within a single register, placing the innermost loop count at the highest bit positions.

Other Counted Loops:

You may want to use the value of a loop counter as an input to calculations in the loop. It's not always desirable to count down from *N* to *1* or *N −1* to *0*. For example, you may want to select bits out of a data register one at a time; in this case you may want a power-of-two mask that doubles on each iteration.

The following subsections show useful looping structures that count in different patterns. They use only a single instruction combined with a branch to implement the loop.

Negative Indexing: This loop structure counts from *−N* to *0* (inclusive or exclusive) in steps of size *STEP*.

```
        RSB     i, N, #0        ; i=-N
loop
        ; loop body goes here and i=-N,-N+STEP,...,
        ADDS    i, i, #STEP
        BLT     loop            ; use BLT or BLE to exclude 0 or not
```

Logarithmic Indexing: This loop structure counts down from 2^N to *1* in powers of two. For example, if $N = 4$, then it counts 16, 8, 4, 2, 1.

```
          MOV     i, #1
          MOV     i, i, LSL N
loop
          ; loop body
          MOVS    i, i, LSR#1
          BNE     loop
```

The following loop structure counts down from an *N-bit* mask to a one-bit mask. For example, if *N = 4*, then it counts 15, 7, 3, 1.

```
          MOV     i, #1
          RSB     i, i, i, LSL N  ; i=(1<<N)-1
loop
          ; loop body
          MOVS    i, i, LSR#1
          BNE     loop
```