# MICROCONTROLLERS AND EMBEDDED SYSTEMS (18CS44)
# MODULE – 02
# ARM PROGRAMMING USING ASSEMBLY LANGUAGE

## ASSEMBLY PROGRAMMING USING ASSEMBLY LANGUAGE

Embedded software projects often contain a few key subroutines that dominate system performance. By optimizing these routines you can reduce the system power consumption and reduce the clock speed needed for real-time operation. Optimization can turn an infeasible system into a feasible one or an uncompetitive system into a competitive one. For maximum performance you can optimize critical routines using hand-written assembly. Writing assembly by hand gives you direct control of 3 Optimization tools that you cannot explicitly use by writing C source.

**1. Instruction scheduling:** Reordering the instructions in a code sequence to avoid processor Stalls because of dependency.

**2. Register allocation:** Deciding how variables should be allocated to ARM registers or stack locations for maximum performance.

**3. Conditional execution:** Accessing the full range of ARM condition codes and conditional instructions.

When we take the time to optimize a routine, it has the side benefit of giving us a better understanding of the algorithm, its bottlenecks and dataflow. Common Optimization techniques, specific to writing assembly language is described. Thumb assembly is not covered specifically since ARM assembly will always give better performance when a 32 bit bus is available. Thumb is most useful for reducing the compiled size of C code that is not critical to performance and for efficient execution on a 16- bit data bus. Many of the principles covered here apply equally well to Thumb and ARM.

## WRITING ASSEMBLY CODE

Here we use ARM Macro Assembler ***armasm*** for example. You can also use the GNU assembler gas.

This section gives examples showing how to convert the C function to basic assembly code.

Example 1: This example shows how to convert a C function to an assembly function. Consider the simple C program main.c following that prints the squares of the integers from

0 to 9:

```
#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:

**Main1.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));
  }
}
```

**Square.s**

```
        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
```

- AREA directive names the area or code section that the code lives in.
- .text. : Read Only code | .text |: Alphanumeric character in prgm name
- EXPORT directive makes the symbol square available for external linking.
- The symbol "square" in the 6th line is considered as label. *armasm* treats non-indented text as a label definition.
- When "square" is called, 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 store the multiply result into r1 and then move to r0.

- END directive marks the end of the assembly file and 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
```

This example works only if we are compiling the C as ARM code. If we complile the C as Thumb code, then the assembly routine must return using a BX instruction.

**NOTE**: The *BX* instruction causes a branch to the address contained in Rm and exchanges the instruction set, if required:

*BX Rm* derives the target instruction set from bit[0] of *Rm*:

- If bit[0] of *Rm* is 0, the processor changes to, or remains in, ARM state.
- If bit[0] of *Rm* is 1, the processor changes to, or remains in, Thumb state.

Example 2: When calling ARM code from C compiled as THUMB, the only change required to the assembly in this example is to change the return instruction to a BX. BX will return to ARM or THUMB state according to bit zero 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. 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 of the THUMB C

code to call the ARM assembly code. We can see the following, commands to build this example:

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

**INSTRUCTION SCHEDULING**

- The time taken to execute instructions depends on the implementation pipeline.
- Here, we consider the ARM9TDMI pipeline timings.
- 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 32-bit words of memory such as LDR and LDM take N cycles to issue. If the instruction loads pc, then add two cycles.
    - Load instructions that load 16-bit or 8-bit data such as LDRB, LDRSB, LDRH, and LDRSH take one cycle to issue.

- Branch instructions take three cycles.
- Store instructions that store N values take N cycles.
- Multiply instructions take a varying number of cycles depending on the value of the second operand

**ARM Pipeline and Dependencies**

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:

1. Fetch
2. Decode
3. Execute (ALU)
4. LS1
5. LS2

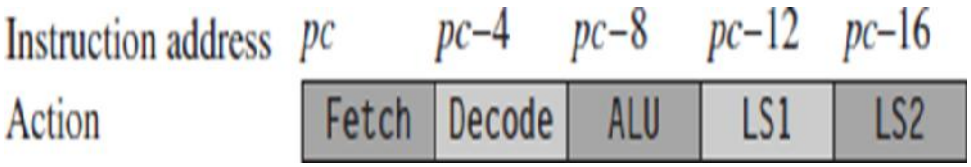


**Fig- ARM9TDMI Pipeline Executing in ARM state**

**Fetch**: Fetches instruction from memory using pc. The instruction is then loaded into the core and then processed 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 or if they are not available via one of the forwarding paths.

**Execute** (ALU): Executes the instruction that was decoded in the previous cycle.

This instruction was originally fetches the address from pc −8 (ARM state) or pc − 4 (Thumb state).

This stage calculates

- The answer for a data processing operation,
- The address for a load, store, or branch operation.

Some instructions take several cycles in this stage.

Example: multiply and register-controlled shift operations take several ALU cycles.

**LS1**: This stage 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 or bypassed.

**LS2**: 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.

After an instruction has completed the five stages of the pipeline, write the result of 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 the address PC - 8 in parallel with fetching the instruction at address PC. The following example shows 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.***

**Example1:** Case with No Interlock

```
ADD   r0, r0, r1   Cycle 1: r0←r1
ADD   r0, r0, r2   Cycle 2: r0←r2
```

This instruction pair takes two cycles.

**Example2:** Case with One-cycle interlock caused by load use.

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

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

Instruction pair takes three cycles. The ALU calculates the address r2 + 4 in the first cycle while decoding the ADD instruction in parallel. ADD cannot proceed on the second cycle because the load instruction has not yet loaded the value of r1. Therefore the pipeline stalls (pipeline bubble) for one cycle. Now that r1 is ready, the processor executes the ADD on the third cycle.

**Example3:** One-cycle interlock caused by delayed load use.

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

| 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 |

**Fig -Sequence progress through the processor pipeline**

Instruction triplet takes four cycles. *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 *ADD* doesn't cause any stalls since the *ADD* does not use *r1*, the result of the load.

**Example 4:** 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
```

| 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 | – | – |

Three executed instructions take a total of five cycles. The *MOV* instruction executes on the first cycle. In 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 pipeline drops the two instructions following the branch when the branch takes place.

**Scheduling of Load Instruction**

Load instructions occur frequently in the compiled code accounting for approximately one-third of all instructions. Careful scheduling of load instructions has to take place so that

pipeline stalls doesn't occur and 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.

Let's 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
```

- 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

## Cycle Count

```
str_tolower
    LDRB r2,[r1],#1              ; c = *(in++)          cycle-1
                                                        cycle-2 stall
                                                        cycle-3 stall
    SUB r3,r2,#0x41              ; r3 = c -'A'          cycle-4
    CMP r3,#0x19                 ; if (c <='Z'-'A')     cycle-5
    ADDLS r2,r2,#0x20            ; c +='a'-'A'          cycle-6
    STRB r2,[r0],#1              ; *(out++) = (char)c   cycle-7
    CMP r2,#0           ; if (c!=0)                     cycle -8
    BNE str_tolower     ; goto str_tolower              cycle-9
                                                        cycle-10 flush
                                                        cycle-11 flush

MOV pc,r14 ; return
```

- 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**

In this method, we Load the data required for the loop at the end of the previous loop, rather than at the beginning of the current loop. To get performance improvement with little increase in code size, we don't unroll the loop.

Example: This assembly applies preload method to the *str_tolower* function

```
out     RN 0    ; pointer to output string
in      RN 1    ; pointer to input string
c       RN 2    ; character loaded
t       RN 3    ; scratch register
        ; void str_tolower_preload(char *out, char *in)
        str_tolower_preload
        LDRB    c, [in], #1       ; c = *(in++)
loop
        SUB     t, c, #'A'        ; t = c-'A'
        CMP     t, #'Z'-'A'       ; if (t <= 'Z'-'A')
        ADDLS   c, c, #'a'-'A'    ;   c += 'a'-'A';
        STRB    c, [out], #1      ; *(out++) = (char)c;
        TEQ     c, #0             ; test if c==0
        LDRNEB  c, [in], #1       ; if (c!=0) { c=*in++;
        BNE     loop              ;             goto loop; }
        MOV     pc, lr            ; return
```

The scheduled version is one instruction longer than the C version, but we save two cycle for each inner loop iteration. This reduces the loop from 11 cycle per character to 9 cycle per character on an ARM9TDMI, giving a 1.22 time speed improvement.

The ARM architecture is particularly well suited to this type of preloading because instructions can be executed conditionally. Since loop i is loading the data for loop i+1 there is always a problem with the first and the last loops. For the first loop we can preload data by inserting extra load instructions before the loop starts. For the last loop it is essential that the loop does not read any data, or it will read beyond the end of the array. This could cause a data abort. With ARM we can easily solve this problem by making the load instruction

conditional. In the above example the preload of the next character only takes place if the loop will iterate once more. No byte load occurs on the last loop.

**Load scheduling by Unrolling**

This method of load scheduling works by unrolling and then interleaving the body of the loop. For example, we can perform loop iterations i, i + 1, i + 2 interleaved. When the result of an operation from loop i is not ready, we can perform an operation from loop i + 1 that avoids waiting for the loop i result.

```
out     RN 0    ; pointer to output string
in      RN 1    ; pointer to input string
ca0     RN 2    ; character 0
t       RN 3    ; scratch register
ca1     RN 12   ; character 1
ca2     RN 14   ; character 2
        ; void str_tolower_unrolled(char *out, char *in)
        str_tolower_unrolled
        STMFD   sp!, {lr}         ; function entry
loop_next3
        LDRB    ca0, [in], #1     ; ca0 = *in++;
        LDRB    ca1, [in], #1     ; ca1 = *in++;
        LDRB    ca2, [in], #1     ; ca2 = *in++;
        SUB     t, ca0, #'A'      ; convert ca0 to lower case
        CMP     t, #'Z'-'A'
        ADDLS   ca0, ca0, #'a'-'A'
        SUB     t, ca1, #'A'      ; convert ca1 to lower case
        CMP     t, #'Z'-'A'
        ADDLS   ca1, ca1, #'a'-'A'
        SUB     t, ca2, #'A'      ; convert ca2 to lower case
        CMP     t, #'Z'-'A'
        ADDLS   ca2, ca2, #'a'-'A'
        STRB    ca0, [out], #1    ; *out++ = ca0;
        TEQ     ca0, #0           ; if (ca0!=0)
        STRNEB  ca1, [out], #1    ;   *out++ = ca1;
        TEQNE   ca1, #0           ; if (ca0!=0 && ca1!=0)
        STRNEB  ca2, [out], #1    ;   *out++ = ca2;
        TEQNE   ca2, #0           ; if (ca0!=0 && ca1!=0 && ca2!=0)
        BNE     loop_next3        ;   goto loop_next3;
        LDMFD   sp!, {pc}         ; return;
```

This loop is more efficient and requires seven cycles per character on ARM9TDMI. This gives a 1.57 time speed increase over the original str_tolower. However the improvement in this example does have some costs. The routine is more than double the code size of the original

implementation. It is assumed that we can read upto 2 characters beyond the end of the input string, which may not be true if the string is right at the end of the available RAM, where reading of the end will cause a data abort. Also, performance can be slower for very short strings because (1) stacking lr causes additional function call overhead and (2) the routine may process upto 2 characters pointlessly, before discovering that they lie beyond the end of the string.

This form of scheduling by unrolling is used for time critical parts of an application where you know the data size is large. If the size of the data is also known at compile time we can remove the problem of reading beyond the end of the array.

**REGISTER ALLOCATION**

- 14 of the 16 visible ARM registers can be used to hold general-purpose data.
- The other two registers are: 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 we 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
```

- r12 is also stacked just to keep the stack eight-byte aligned.
    - Address starts from 0 and ends at 28 for r4-r11 (28 is not multiple of 8)
    - Address starts from 0 and ends at 32 for r4-r12 (32 is multiple of 8)
- r12 need not be stacked 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.
- For ARMv4T processor, i.e) if the routines are called from Thumb code, 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 Members**

- It is best to use alternate names to the registers, rather than explicit register numbers.
- Benefits of using alternate names:
    - Allows you to change the allocation of variables to register numbers easily.
    - Allows using 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. 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.

For an example of how to allocate registers when writing assembly, suppose we want to shift an array of N bits upwards in memory by k bits. For simplicity assume that N is large

and a multiple of 256. Also assume that 0≤k<32 and that the input and output pointers are word aligned. This type of operation is common in dealing with the arithmetic of multiple precision numbers where we want to multiply by 2k. It is also useful to block copy from one bit or byte alignment to a different bit or byte alignment.

For example, the C library function ***memcpy*** can use the routine to copy an array of bytes using only word accesses.The C routine ***shift_bits*** implements the simple k-bit shift of N bits of data. It returns the k bits remaining following the shift.

```
unsigned int shift_bits(unsigned int *out, unsigned int *in,
                        unsigned int N, unsigned int k)
{
  unsigned int carry=0, x;

  do
  {
    x = *in++;
    *out++ = (x<<k) | carry;
     carry = x>>(32-k);
     N -= 32;
   } while (N);

   return carry;
}
```

The obvious way to improve efficiency is to unroll the loop to process eight words of 256 bits at a time so that we can use load and store multiple operations to load and store eight words at a time for maximum efficiency. Before thinking about register numbers, we write the following assembly code:

```
shift_bits
        STMFD   sp!, {r4-r11, lr}       ; save registers
        RSB     kr, k, #32              ; kr = 32-k;
        MOV     carry, #0
loop
        LDMIA   in!, {x_0-x_7}          ; load 8 words
        ORR     y_0, carry, x_0, LSL k  ; shift the 8 words
        MOV     carry, x_0, LSR kr
        ORR     y_1, carry, x_1, LSL k
        MOV     carry, x_1, LSR kr
        ORR     y_2, carry, x_2, LSL k
        MOV     carry, x_2, LSR kr
        ORR     y_3, carry, x_3, LSL k
        MOV     carry, x_3, LSR kr
        ORR     y_4, carry, x_4, LSL k
        MOV     carry, x_4, LSR kr
        ORR     y_5, carry, x_5, LSL k
        MOV     carry, x_5, LSR kr
        ORR     y_6, carry, x_6, LSL k
        MOV     carry, x_6, LSR kr
        ORR     y_7, carry, x_7, LSL k
        MOV     carry, x_7, LSR kr
        STMIA   out!, {y_0-y_7}         ; store 8 words
        SUBS    N, N, #256              ; N -= (8 words * 32 bits)
        BNE     loop                    ; if (N!=0) goto loop;
        MOV     r0, carry               ; return carry;
        LDMFD   sp!, {r4-r11, pc}
```

For register allocation, the input arguments do not have to move registers, so that we can immediately assign.

```
out     RN 0
in      RN 1
N       RN 2
k       RN 3
```

For the load multiple to work correctly, we must assign x0 through x7 to sequentially increasing register numbers, and similarly for y0 through y7. Notice that we finish with x0 before starting with y1. In general, we can assign xn to the same register as yn+1. Therefore, assign,

```
x_0     RN 5
x_1     RN 6
x_2     RN 7
x_3     RN 8
x_4     RN 9
x_5     RN 10
x_6     RN 11
x_7     RN 12
y_0     RN 4
y_1     RN x_0
y_2     RN x_1
y_3     RN x_2
y_4     RN x_3
y_5     RN x_4
y_6     RN x_5
y_7     RN x_6
```

We are nearly finished, but there is a problem. There are two remaining variables carry and kr, but only one remaining free register lr. There are several possible ways we can proceed when we run out of registers:

- Reduce the number of registers we require by performing fewer operations in each loop. In this case we could load four words in each load multiple rather than eight.
- Use the stack to store the least-used values to free up more registers. In this case we could store the loop counter N on the stack.
- Alter the code implementation to free up more registers. This is the solution we consider in the following text.

We often iterate the process of implementation followed by register allocation several times until the algorithm fits into the 14 available registers. In this case we notice that the carry value need not stay in the same register at all! We can start off with the carry value in y0 and then move it to y1 when x0 is no longer required, and so on. We complete the routine by allocating kr to lr and recoding so that carry is not required.

This assembly shows our final ***shift_bits*** routine. It uses all 14 available ARM registers:

```
kr   RN lr

shift_bits
            STMFD   sp!, {r4-r11, lr}      ; save registers
            RSB     kr, k, #32             ; kr = 32-k;
            MOV     y_0, #0                ; initial carry
loop
            LDMIA   in!, {x_0-x_7}         ; load 8 words
            ORR     y_0, y_0, x_0, LSL k   ; shift the 8 words
            MOV     y_1, x_0, LSR kr       ; recall x_0 = y_1
            ORR     y_1, y_1, x_1, LSL k
            MOV     y_2, x_1, LSR kr
            ORR     y_2, y_2, x_2, LSL k
            MOV     y_3, x_2, LSR kr
            ORR     y_3, y_3, x_3, LSL k
            MOV     y_4, x_3, LSR kr
            ORR     y_4, y_4, x_4, LSL k
            MOV     y_5, x_4, LSR kr
            ORR     y_5, y_5, x_5, LSL k
            MOV     y_6, x_5, LSR kr
            ORR     y_6, y_6, x_6, LSL k
            MOV     y_7, x_6, LSR kr
            ORR     y_7, y_7, x_7, LSL k
            STMIA   out!, {y_0-y_7}        ; store 8 words
            MOV     y_0, x_7, LSR kr
            SUBS    N, N, #256             ; N -= (8 words * 32 bits)
            BNE     loop                   ; if (N!=0) goto loop;
            MOV     r0, y_0                ; return carry;
            LDMFD   sp!, {r4-r11, pc)
```

Other ways to allocate variables to register numbers for register-intensive tasks, and to use more than 14 local variables, and to make the best use of the 14 available registers are as follows:

- ***Using More Than 14 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.

- ✓ For example, you can store a loop counter and a shift in one register. You can also storemultiple pixels in one register.

**CONDITIONAL EXECUTION**

- The processor core can conditionally execute most ARM instructions.
- By combining conditional execution and conditional setting of the flags, it is possible to implement simple if statements without any need for branches.
- This improves efficiency since branches can take many cycles and also reduces code size.

**Example 1: Converts an unsigned integer 0 ≤ *i* ≤ *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. Section 6.3.1 shows a similar use of conditional execution to convert to lowercase. Conditional execution is even more powerful for cascading conditions.

**Example 2: 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 3: 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 $0 \le c \le$ 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
```

**LOOPING CONSTRUCTS**

- Most routines critical to performance will contain a loop.
- Note that, ARM loops are fastest when they count down towards zero.
- Different types of looping constructs available in ARM are:
    1. Decremented Counted Loops
    2. Unrolled Counted Loops

3. Multiple Nested Loops
4. Other Counted Loops

- Here we understand to implement count down loops efficiently in assembly and 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.
- 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
```

- 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
```

**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.

**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*.

**Use of Looping Structures that count in different patterns.**

- **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
```