



Visvesvaraya Technological University (VTU)

Subject Code:21CS43

Subject : MICROCONTROLLER AND EMBEDDED SYSTEM

MODULE-03**C Compilers and Optimization and ARM Programming using Assembly Language****Chapter-I****C Compilers and Optimization****3.1. Structure Arrangement**

- The way you lay out a frequently used structure can have a significant impact on its performance and code density.
- There are two issues concerning structures on the ARM:
 1. Alignment of the structure entries
 2. the overall size of the structure.
- For architectures up to and including ARMv5TE, load and store instructions are only guaranteed to load and store values with address aligned to the size of the access width.
- ARM compilers will automatically align the start address of a structure to a multiple of the largest access width used within the structure and align entries within structures to their access width by inserting padding.

For example, consider the structure

```
struct {
    char a;
    int b;
    char c;
    short d;
}
```

For a little-endian memory system, the compiler will lay this out adding padding to ensure that the next object is aligned to the size of that object:

Address	+3	+2	+1	+0
+0	pad	pad	pad	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	d[15,8]	d[7,0]	pad	c

Load and store alignment restrictions for ARMv5TE.

Transfer size	Instruction	Byte address
1 byte	LDRB, LDRSB, STRB	any byte address alignment
2 bytes	LDRH, LDRSH, STRH	multiple of 2 bytes
4 bytes	LDR, STR	multiple of 4 bytes
8 bytes	LDRD, STRD	multiple of 8 bytes

The following rules generate a structure with the elements packed for maximum efficiency:

- Place all 8-bit elements at the start of the structure.
- Place all 16-bit elements next, then 32-bit, then 64-bit.
- Place all arrays and larger elements at the end of the structure.
- If the structure is too big for a single instruction to access all the elements, then group the elements into substructures.
- The compiler can maintain pointers to the individual substructures.

Thumb load and store offsets.

Instructions	Offset available from the base register
LDRB, LDRSB, STRB	0 to 31 bytes
LDRH, LDRSH, STRH	0 to 31 halfwords (0 to 62 bytes)
LDR, STR	0 to 31 words (0 to 124 bytes)

Efficient Structure Arrangement

- Lay structures out in order of increasing element size. Start the structure with the smallest elements and finish with the largest.
- Avoid very large structures. Instead use a hierarchy of smaller structures.
- For portability, manually add padding into API
- structures so that the layout of the structure does not depend on the compiler.
- Beware of using enum types in API structures. The size of an enum type is compiler dependent.

3.2. Bit-Fields

- Bit-fields are probably the least standardized part of the ANSI C specification.
- The compiler can choose how bits are allocated within the bit-field container.
- For this reason alone, avoid using bit-fields inside a union or in an API structure definition.
- Different compilers can assign the same bit-field different bit positions in the container.
- It is also a good idea to avoid bit-fields for efficiency.
- it-fields are structure elements and usually accessed using structure pointers; consequently.

Example

```
void dostageA(void);
```

```
void dostageB(void);
```

```
void dostageC(void);
```

```
typedef struct {
```

```
    unsigned int stageA : 1;
```

```
    unsigned int stageB : 1;
```

```
    unsigned int stageC : 1;
```

```
} Stages_v1;
```

```
void dostages_v1(Stages_v1 *stages)
{
    if (stages->stageA)
    {
        dostageA();
    }if (stages->stageB)
    {
        dostageB();
    }
    if (stages->stageC)
    {
        dostageC();
    }
}
```

Here, we use three bit-field flags to enable three possible stages of processing. The example compiles to

```
dostages_v1
    STMFD    r13!,{r4,r14}      ; stack r4, lr
    MOV      r4,r0              ; move stages to r4
    LDR      r0,[r0,#0]         ; r0 = stages bitfield
    TST      r0,#1              ; if (stages->stageA)
    BLNE     dostageA           ;     {dostageA();}
    LDR      r0,[r4,#0]         ; r0 = stages bitfield
    MOV      r0,r0,LSL #30      ; shift bit 1 to bit 31
    CMP      r0,#0              ; if (bit31)
    BLLT     dostageB           ;     {dostageB();}
    LDR      r0,[r4,#0]         ; r0 = stages bitfield
    MOV      r0,r0,LSL #29      ; shift bit 2 to bit 31
    CMP      r0,#0              ; if (!bit31)
    LDMLTFD  r13!,{r4,r14}      ;     return
    BLT      dostageC           ; dostageC();
    LDMFD    r13!,{r4,pc}       ; return
```

You can also use the masks to set and clear the bit-fields, just as easily as for testing them.

The following code shows how to set, clear, or toggle bits using the STAGE masks:

```
stages |= STAGEA; /* enable stage A */
```

```
stages &= ~STAGEB; /* disable stage B */
```

```
stages A= STAGEC; /* toggle stage C */
```

Bit-fields

- Avoid using bit-fields. Instead use #define or enum to define mask values.
- Test, toggle, and set bit-fields using integer logical AND, OR, and exclusive OR operations with the mask values.
- These operations compile efficiently, and you can test, toggle, or set multiple fields at the same time.

3.3. Unaligned Data and Endianness

- Unaligned data and endianness are two issues that can complicate memory accesses and portability.
- The ARM load and store instructions assume that the address is a multiple of the type you are loading or storing.
- If you load or store to an address that is not aligned to its type, then the behavior depends on the particular implementation.
- The core may generate a data abort or load a rotated value. For well-written, portable code you should avoid unaligned accesses.
- C compilers assume that a pointer is aligned unless you say otherwise.
- If a pointer isn't aligned, then the program may give unexpected results.
- This is sometimes an issue when you are porting code to the ARM from processors that do allow unaligned accesses.
- For armcc, the __packed directive tells the compiler that a data item can be positioned at any byte alignment.
- This is useful for porting code, but using __packed will impact performance.

Example:-simple routine

```
int readint(__packed int *data)
{
return *data;
}
```

This compiles to

```
readint
    BIC    r3,r0,#3           ; r3 = data & 0xFFFFFFFF
    AND    r0,r0,#3           ; r0 = data & 0x00000003
    MOV    r0,r0,LSL #3       ; r0 = bit offset of data word
    LDMIA  r3,{r3,r12}        ; r3, r12 = 8 bytes read from r3
    MOV    r3,r3,LSR r0       ; These three instructions
    RSB    r0,r0,#0x20        ; shift the 64 bit value r12.r3
    ORR    r0,r3,r12,LSL r0   ; right by r0 bits
    MOV    pc,r14             ; return r0
```

Little-endian configuration.

Instruction	Width (bits)	b31..b24	b23..b16	b15..b8	b7..b0
LDRB	8	0	0	0	B(A)
LDRSB	8	S(A)	S(A)	S(A)	B(A)
STRB	8	X	X	X	B(A)
LDRH	16	0	0	B(A+1)	B(A)
LDRSH	16	S(A+1)	S(A+1)	B(A+1)	B(A)
STRH	16	X	X	B(A+1)	B(A)
LDR/STR	32	B(A+3)	B(A+2)	B(A+1)	B(A)

Big-endian configuration.

Instruction	Width (bits)	b31..b24	b23..b16	b15..b8	b7..b0
LDRB	8	0	0	0	B(A)
LDRSB	8	S(A)	S(A)	S(A)	B(A)
STRB	8	X	X	X	B(A)
LDRH	16	0	0	B(A)	B(A+1)
LDRSH	16	S(A)	S(A)	B(A)	B(A+1)
STRH	16	X	X	B(A)	B(A+1)
LDR/STR	32	B(A)	B(A+1)	B(A+2)	B(A+3)

Notes:

B(A) : The byte at address A.

S(A) : 0xFF if bit 7 of B(A) is set, otherwise 0x00.

X: These bits are ignored on a write.

Example:

These functions read a 32-bit integer from a byte stream pointed to by data.

```
int readint_little (char *data)
```

```
{
```

```
    int a0,a1,a2,a3;
```

```
    a0 = *(data++);
```

```
    a1 = *(data++);
```

```
    a2 = *(data++);
```

```
    a3 = *(data++);
```

```
    return a0 | (a1 << 8) | (a2 << 16) | (a3 << 24);
```

```
}
```

```
int readint_big (char *data)
{
    int a0, a1, a2, a3;
    a0 = *(data++);
    a1 = *(data++);
    a2 = *(data++);
    a3 = *(data++);
    return (((a0 << 8) | a1) << 8) | a2 << 8) | a3;
}
```

- If speed is critical, then the fastest approach is to write several variants of the critical routine.
- For each possible alignment and ARM endianness configuration, you call a separate routine optimized for that situation.

Endianness and Alignment

- Avoid using unaligned data if you can.
- Use the type `char *` for data that can be at any byte alignment.
- Access the data by reading bytes and combining with logical operations.
- Then the code won't depend on alignment or ARM endianness configuration.
- For fast access to unaligned structures, write different variants according to pointer alignment and processor endianness.

3.4 .Division

- The ARM does not have a divide instruction in hardware.
- Instead, the compiler implements divisions by calling software routines in the C library.
- There are many different types of division routine that you can tailor to a specific range of numerator and denominator values.
- Division and modulus (/ and %) are such slow operations that you should avoid them as much as possible.

```
offset = (offset + increment) % buffer_size;
```

Instead it is far more efficient to write

```
offset += increment;
```

```
if (offset >= buffer_size)
```

```
{
```

```
    offset -= buffer_size;
```

```
}
```

Example: In `getxy_v2`, the quotient and remainder operation only require a single call to a division routine:

```
point getxy_v2(unsigned int offset, unsigned int bytes_per_line)
```

```
{
```

```
    point p;
```

```
    p.x = offset % bytes_per_line;
```

```
p.y = offset / bytes_per_line;  
  
return p;  
  
}
```

```
getxy_v2  
    STMFD    r13!,{r4, r14}    ; stack r4, lr  
    MOV      r4,r0              ; move p to r4  
    MOV      r0,r2              ; r0 = bytes_per_line  
    BL       __rt_udiv          ; (r0,r1) = (r1/r0, r1%r0)  
    STR      r0,[r4,#4]         ; p.y = offset / bytes_per_line  
    STR      r1,[r4,#0]         ; p.x = offset % bytes_per_line  
    LDMFD    r13!,{r4,pc}      ; return
```

3.4.1. Repeated Unsigned Division with Remainder

- Often the same denominator occurs several times in code.
- In the previous example, bytes_per_line will probably be fixed throughout the program.
- If we project from three to two cartesian coordinates, then we use the denominator twice:

$$(x, y, z) \rightarrow (x/z, y/z)$$

3.4.2. Converting Divides into Multiplies

We'll use the following notation to distinguish exact mathematical divides from integer divides:

- n/d = the integer part of n divided by d , rounding towards zero (as in C)
- $n\%d$ = the remainder of n divided by d which is $n - d(n / d)$
- $\frac{n}{d} = nd^{-1}$ = the true mathematical divide of n by d

The obvious way to estimate d^{-1} , while sticking to integer arithmetic, is to calculate $2^{32}/d$. Then we can estimate n/d

$$(n(2^{32}/d)) / 2^{32}$$

Example : The following routine, `scale`, shows how to convert divisions to multiplications in practice.

```

void scale(
    unsigned int *dest,      /* destination for the scale data */
    unsigned int *src,       /* source unscaled data */
    unsigned int d,          /* denominator to divide by */
    unsigned int N)          /* data length */
{
    unsigned int s = 0xFFFFFFFFu / d;
    do
    {
        unsigned int n, q, r;

```

```

    n = *(src++);
    q = (unsigned int)((((unsigned long long)n * s) >> 32));
    r = n - q * d;
    if (r >= d)
    {
        q++;
    }
    *(dest++) = q;
} while (--N);
}

```

3.4.3. Unsigned Division by a Constant

- The idea is to use an approximation to $d-1$ that is sufficiently accurate so that multiplying by the approximation gives the exact value of n/d .

We use the following mathematical results:

If $2^{N+k} \leq ds \leq 2^{N+k} + 2^k$, then $n/d = (ns) \gg (N+k)$ for $0 \leq n < 2^N$.

If $2^{N+k} - 2^k \leq ds < 2^{N+k}$, then $n/d = (ns + s) \gg (N+k)$ for $0 \leq n < 2^N$.

Example:

In practice d will be a fixed constant rather than a variable.

You can precalculated s and k in advance and only include the calculations relevant for your particular value of d.

```
unsigned int udiv_by_const(unsigned int n, unsigned int d)
```

```
{
```

```
    unsigned int s,k,q;
```

```
    /* We assume d!=0 */
```

```
    /* first find k such that (1 << k) <= d < (1 << (k+1)) */
```

```
        for (k=0; d/2>=(1u << k); k++);
```

```
        if (d==(1u << k)
```

```
{
```

```
    /* we can implement the divide with a shift */
```

```
        return n >> k;
```

```
}
```

```
/* d is in the range (1 << k) < d < (1 << (k+1)) */
```

```
    s = (unsigned int)((((1ull << (32+k))+(1ull << k))/d);
```

```
    if (((unsigned long long)s*d >= (1ull << (32+k))))
```

```
{
```

```
    /* n/d = (n*s) >> (32+k) */
```

```
        q = (unsigned int)((((unsigned long long)n*s) >> 32);
```

```
        return q >> k;
```

```
}
```



```

/* n/d = (n*s+s) >> (32+k) */
    q = (unsigned int)((((unsigned long long)n*s + s) >> 32);
    return q >> k;
}

```

3.4.3. Signed Division by a Constant

o handle signed constants as well. If $d < 0$, then we can divide by $|d|$ and correct the sign later, so for now

we assume that $d > 0$

$$n/d = (ns) \gg (N + k) \text{ for all } 0 \leq n < 2^N$$

$$n/d = ((ns) \gg (N + k)) + 1 \text{ for all } -2^N \leq n < 0$$

Example: The following routine, `sdiv_by_const`, shows how to divide by a signed constant `d`.

```

int sdiv_by_const(int n, int d)
{
    int s,k,q;
    unsigned int D;

    /* set D to be the absolute value of d, we assume d!=0 */
    if (d>0)
    {
        D=(unsigned int)d; /* 1 <= D <= 0x7FFFFFFF */
    }
}

```

```

Else
{
    D=(unsigned int) - d; /* 1 <= D <= 0x80000000 */
}

/* first find k such that (1 << k) <= D < (1 << (k+1)) */
for (k=0; D/2>=(1u << k); k++);
if (D==1u << k)
{
    /* we can implement the divide with a shift */
    q = n >> 31; /* 0 if n>0, -1 if n<0 */
    q = n + ((unsigned)q >> (32-k)); /* insert rounding */
    q = q >> k; /* divide */
    if (d < 0)
    {
        q = -q; /* correct sign */
    }

    return q;
}

/* Next find s in the range 0<=s<=0xFFFFFFFF */

/* Note that k here is one smaller than the k in the equation */
s = (int)(((1ull << (31+(k+1)))+(1ull << (k+1)))/D);
if (s>=0)

```

```
{  
    q = (int)((((signed long long)n*s) >> 32);  
}  
else  
{  
    /* (unsigned)s = (signed)s + (1 << 32) */  
    q = n + (int)((((signed long long)n*s) >> 32);  
}  
  
    q = q >> k;  
  
    /* if n<0 then the formula requires us to add one */  
    q += (unsigned)n >> 31;  
  
    /* if d was negative we must correct the sign */  
    if (d<0)  
    {  
        q = -q;  
    }  
  
    return q;  
}
```

3.5. Floating Point

- The majority of ARM processor implementations do not provide hardware floating-point
- which saves on power and area when using ARM in a price-sensitive, embedded application
- With the exceptions of the Floating-Point Accelerator (FPA) used on the ARM7500FE
- the Vector Floating Point accelerator (VFP) hardware.
- the C compiler must provide support for floating point in software.
- the C compiler converts every floating-point operation into a subroutine call.
- the C library contains subroutines to simulate floating-point behavior using integer arithmetic.
- This code is written in highly optimized assembly.
- floating-point algorithms will execute far more slowly than corresponding integer algorithms.
- If you need fast execution and fractional values, you should use fixed-point or block- floating algorithms.

3.6. Inline Functions and Inline Assembly.

- You can remove the function call overhead completely by inlining functions.
- many compilers allow you to include inline assembly in your C source code.
- Using inline functions that contain assembly you can get the compiler to support ARM instructions and optimizations that aren't usually available.
- The inline assembler is part of the C compiler.
- The C compiler still performs register allocation, function entry, and exit.
- The main benefit of inline functions and inline assembly is to make operations that are not usually available as part of the C language accessible in C.
- It is better to use inline functions rather than #define macros because the latter doesn't check the types of the function arguments and return value.

Example:

```
__inline int qmac (int a, int x, int y)
{
    int i;

    i = x*y; /* this multiplication cannot saturate */

    if (i>=0)
    {
        /* x*y is positive */

        i = 2*i;
    }
}
```

```
    if (i<0)

{

/* the doubling saturated */

    i = 0x7FFFFFFF;

}

    if (a + i < a)

{

/* the addition saturated */

    return 0x7FFFFFFF;

}

return a + i;

}

/* x*y is negative so the doubling can't saturate */

if (a + 2*i > a)

{

/* the accumulate saturated */

    return - 0x80000000;

}

    return a + 2*i;
```

```
}
```

We can now use this new operation to calculate a saturating correlation. In other words,

we calculate $a = 2x_0y_0 + \dots + 2x_{N-1}y_{N-1}$ with saturation.

```
int sat_correlate(short *x, short *y, unsigned int N)
```

```
{
```

```
int a=0;
```

```
do
```

```
{
```

```
    a = qmac(a, *(x++), *(y++));
```

```
} while (--N);
```

```
    return a;
```

```
}
```

- Use inline functions to declare new operations or primitives not supported by the C compiler.
- Use inline assembly to access ARM instructions not supported by the C compiler. Examples are coprocessor instructions or ARMv5E extensions.

3.7. Portability Issues.

The Char Type:

On the ARM, char is unsigned rather than signed as for many other processors.

The int type.

Some older architectures use a 16-bit int, which may cause problems when moving to ARM's 32-bit int type although this is rare nowadays.

Unaligned data pointers.

Some processors support the loading of short and int typed values from unaligned addresses.

Endian assumptions.

C code may make assumptions about the endianness of a memory system, for example, by casting a char * to an int *.

Function prototyping.

The armcc compiler passes arguments narrow, that is, reduced to the range of the argument type.

Use of bit-fields.

The layout of bits within a bit-field is implementation and endian dependent.

Use of enumerations.

Although enum is portable, different compilers allocate different numbers of bytes to an enum.

Inline assembly.

Using inline assembly in C code reduces portability between architectures.

Chapter-II

ARM programming using Assembly language

3.8. Writing Assembly code

- this section gives examples showing how to write basic assembly code.
- We assume you are familiar with the ARM instructions
- We also assume that you are familiar with the ARM and Thumb procedure call standard.

Example: 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:

```
#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 nonalphanumeric characters in a symbol or area name, then enclose the name in vertical bars.
- Many nonalphanumeric 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.

3.9. 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.
- A cycle counter measures the number of cycles taken by a specific routine.
- The ARM simulator used by the ADS1.1 debugger is called the ARMulator and provides profiling and cycle counting features.
- 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.
- Be sure that you know how the profiler you are using works and the limits of its accuracy.
- A pc-sampled profiler can produce meaningless results if it records too few samples.
- You can even implement your own pc-sampled profiler in a hardware system using timer interrupts to collect the pc data points.

3.10. Instruction Scheduling

- ALU operations such as addition, subtraction, and logical operations take one cycle.
- 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.
- 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.

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.

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).

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 halfword load instruction.

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

ADD r0, r0, r1

ADD r0, r0, r2

3.10.1 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.

3.10.2 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.

3.11. 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.
- 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

3.11.1 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.

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.
- **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.
- **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.

EXAMPLE This assembly shows our final `shift_bits` routine. It uses all 14 available ARM registers.

6.11

```

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}

```

Register Allocation

- ARM has 14 available registers for general-purpose use: r0 to r12 and r14. The stack pointer r13 and program counter r15 cannot be used for general-purpose data.
- If you need more than 14 local variables, swap the variables out to the stack, working outwards from the innermost loop.
- Use register names rather than physical register numbers when writing assembly routines. This makes it easier to reallocate registers and to maintain the code.
- To ease register pressure you can sometimes store multiple values in the same register. For example, you can store a loop counter and a shift in one register.

3.12. Conditional Execution

- The processor core can conditionally execute most ARM instructions. This conditional execution is based on one of 15 condition codes.
- 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.

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. Section 6.3.1 shows a similar use of conditional execution to convert to lowercase. ■

Conditional execution is even more powerful for cascading conditions.

Conditional Execution

- You can implement most if statements with conditional execution. This is more efficient than using a conditional branch.

Inverted logical relations

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

- You can implement if statements with the logical AND or OR of several similar conditions using compare instructions that are themselves conditional.

3.13. Looping Constructs

This section describes how to implement these loops efficiently in assembly.

3.13.1 Decremental 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
```

3.13.2. Unrolled Counted Loops

- This brings us to the subject of loop unrolling.
- Loop unrolling reduces the loop overhead by executing the loop body multiple times.

Cycles taken for each range of N values.

N range	Cycles taken
$0 \leq N < T_1$	$640N_h + 20N_m + 5N_l + 6$
$T_1 \leq N < T_2$	$160N_h + 5N_m + 5N_l + 17 + 5Z_l$
$T_2 \leq N$	$36N_h + 5N_m + 5N_l + 32 + 5Z_l + 5Z_m$

3.13.3. 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.
- This section gives an example showing how to do this.
- We will ensure the loops count down from $\text{max} - 1$ to 0 inclusive so that the loop terminates by producing a negative result.