Chapter 5 Efficient C Programming

The aim of this chapter is to help you write C code in a style that will compile efficiently on the ARM architecture. We will look at many small examples to show how the compiler translates C source to ARM assembler. Once you have a feel for this translation process, you can distinguish fast C code from slow C code. The techniques apply equally to C++, but we will stick to plain C for these examples.

We start with an overview of C compilers and optimization, which will give an idea of the problems the C compiler faces when optimizing your code. By understanding these problems you can write source code that will compile more efficiently in terms of increased speed and reduced code size. The following sections are grouped by topic.

Sections 5.2 and 5.3 look at how to optimize a basic C loop. These sections use a data packet checksum as a simple example to illustrate the ideas. Sections 5.4 and 5.5 look at optimizing a whole C function body, including how the compiler allocates registers within a function and how to reduce the overhead of a function call.

Sections 5.6 through 5.9 look at memory issues, including handling pointers and how to pack data and access memory efficiently. Sections 5.10 through 5.12 look at basic operations that are usually not supported directly by ARM instructions. You can add your own basic operations using inline functions and assembler.

The final section summarizes problems you may face when porting C code from another architecture to the ARM architecture.

5.1 Overview of C Compilers and Optimization

This chapter assumes that you are familiar with the C language and have some knowledge of assembly programming. The latter is not essential, but is useful for following the compiler output examples. See Chapter 3 or Appendix A for details of ARM assembly syntax.

Optimizing code takes time and reduces source code readability. Usually, it's only worth optimizing functions that are frequently executed and important for performance. We recommend you use a performance profiling tool, found in most ARM simulators, to find these frequently executed functions. Document nonobvious optimizations with source code comments to aid maintainability.

C compilers have to translate your C function literally into assembler so that it works for all possible inputs. In practice, many of the input combinations are not possible or won't occur. Let's start by looking at an example of the problems the compiler faces. The `memclr` function clears `N` bytes of memory at address `data`.

```
void memclr(char *data, int N)
{
  for (; N>0; N--)
  {
    *data=0;
    data++;
  }
}
```

No matter how advanced the compiler, it does not know whether `N` can be 0 on input or not. Therefore the compiler needs to test for this case explicitly before the first iteration of the loop.

The compiler doesn't know whether the `data` array pointer is four-byte aligned or not. If it is four-byte aligned, then the compiler can clear four bytes at a time using an `int` store rather than a `char` store. Nor does it know whether `N` is a multiple of four or not. If `N` is a multiple of four, then the compiler can repeat the loop body four times or store four bytes at a time using an `int` store.

The compiler must be conservative and assume all possible values for `N` and all possible alignments for *data*. Section 5.3 discusses these specific points in detail.

To write efficient C code, you must be aware of areas where the C compiler has to be conservative, the limits of the processor architecture the C compiler is mapping to, and the limits of a specific C compiler.

Most of this chapter covers the first two points above and should be applicable to any ARM C compiler. The third point will be very dependent on the compiler vendor and compiler revision. You will need to look at the compiler's documentation or experiment with the compiler yourself.

To keep our examples concrete, we have tested them using the following specific C compilers:

- *armcc* from ARM Developer Suite version 1.1 (ADS1.1). You can license this compiler, or a later version, directly from ARM.
- *arm-elf-gcc* version 2.95.2. This is the ARM target for the GNU C compiler, *gcc*, and is freely available.

We have used *armcc* from ADS1.1 to generate the example assembler output in this book. The following short script shows you how to invoke *armcc* on a C file `test.c`. You can use this to reproduce our examples.

```
armcc -Otime -c -o test.o test.c
fromelf -text/c test.o > test.txt
```

By default *armcc* has full optimizations turned on (the `-O2` command line switch). The `-Otime` switch optimizes for execution efficiency rather than space and mainly affects the layout of `for` and `while` loops. If you are using the *gcc* compiler, then the following short script generates a similar assembler output listing:

```
arm-elf-gcc -O2 -fomit-frame-pointer -c -o test.o test.c
arm-elf-objdump -d test.o > test.txt
```

Full optimizations are turned off by default for the GNU compiler. The `-fomit-frame-pointer` switch prevents the GNU compiler from maintaining a frame pointer register. Frame pointers assist the debug view by pointing to the local variables stored on the stack frame. However, they are inefficient to maintain and shouldn't be used in code critical to performance.

5.2 Basic C Data Types

Let's start by looking at how ARM compilers handle the basic C data types. We will see that some of these types are more efficient to use for local variables than others. There are also differences between the addressing modes available when loading and storing data of each type.

ARM processors have 32-bit registers and 32-bit data processing operations. The ARM architecture is a RISC load/store architecture. In other words you must load values from memory into registers before acting on them. There are no arithmetic or logical instructions that manipulate values in memory directly.

Early versions of the ARM architecture (ARMv1 to ARMv3) provided hardware support for loading and storing unsigned 8-bit and unsigned or signed 32-bit values.

Table 5.1 Load and store instructions by ARM architecture.

Architecture	Instruction	Action
Pre-ARMv4	`LDRB`	load an unsigned 8-bit value
	`STRB`	store a signed or unsigned 8-bit value
	`LDR`	load a signed or unsigned 32-bit value
	`STR`	store a signed or unsigned 32-bit value
ARMv4	`LDRSB`	load a signed 8-bit value
	`LDRH`	load an unsigned 16-bit value
	`LDRSH`	load a signed 16-bit value
	`STRH`	store a signed or unsigned 16-bit value
ARMv5	`LDRD`	load a signed or unsigned 64-bit value
	`STRD`	store a signed or unsigned 64-bit value

These architectures were used on processors prior to the ARM7TDMI. Table 5.1 shows the load/store instruction classes available by ARM architecture.

In Table 5.1 loads that act on 8- or 16-bit values extend the value to 32 bits before writing to an ARM register. Unsigned values are zero-extended, and signed values sign-extended. This means that the cast of a loaded value to an `int` type does not cost extra instructions. Similarly, a store of an 8- or 16-bit value selects the lowest 8 or 16 bits of the register. The cast of an `int` to smaller type does not cost extra instructions on a store.

The ARMv4 architecture and above support signed 8-bit and 16-bit loads and stores directly, through new instructions. Since these instructions are a later addition, they do not support as many addressing modes as the pre-ARMv4 instructions. (See Section 3.3 for details of the different addressing modes.) We will see the effect of this in the example `checksum_v3` in Section 5.2.1.

Finally, ARMv5 adds instruction support for 64-bit load and stores. This is available in ARM9E and later cores.

Prior to ARMv4, ARM processors were not good at handling signed 8-bit or any 16-bit values. Therefore ARM C compilers define `char` to be an unsigned 8-bit value, rather than a signed 8-bit value as is typical in many other compilers.

Compilers *armcc* and *gcc* use the datatype mappings in Table 5.2 for an ARM target. The exceptional case for type `char` is worth noting as it can cause problems when you are porting code from another processor architecture. A common example is using a `char` type variable `i` as a loop counter, with loop continuation condition `i` $\geq$ `0`. As `i` is unsigned for the ARM compilers, the loop will never terminate. Fortunately *armcc* produces a warning in this situation: *unsigned comparison with 0*. Compilers also provide an override switch to make `char` signed. For example, the command line option `-fsigned-char` will make `char` signed on *gcc*. The command line option `-zc` will have the same effect with *armcc*.

For the rest of this book we assume that you are using an ARMv4 processor or above. This includes ARM7TDMI and all later processors.

Table 5.2 C compiler datatype mappings.

C Data Type	Implementation
`char`	unsigned 8-bit byte
`short`	signed 16-bit halfword
`int`	signed 32-bit word
`long`	signed 32-bit word
`long long`	signed 64-bit double word

5.2.1 Local Variable Types

ARMv4-based processors can efficiently load and store 8-, 16-, and 32-bit data. However, most ARM data processing operations are 32-bit only. For this reason, you should use a 32-bit datatype, `int` or `long`, for local variables wherever possible. Avoid using `char` and `short` as local variable types, even if you are manipulating an 8- or 16-bit value. The one exception is when you want wrap-around to occur. If you require modulo arithmetic of the form $255 + 1 = 0$, then use the `char` type.

To see the effect of local variable types, let's consider a simple example. We'll look in detail at a checksum function that sums the values in a data packet. Most communication protocols (such as TCP/IP) have a checksum or cyclic redundancy check (CRC) routine to check for errors in a data packet.

The following code checksums a data packet containing 64 words. It shows why you should avoid using `char` for local variables.

```
int checksum_v1(int *data)
{
  char i;
  int sum=0;

  for (i=0; i<64; i++)
  {
    sum += data[i];
  }
  return sum;
}
```

At first sight it looks as though declaring `i` as a `char` is efficient. You may be thinking that a `char` uses less register space or less space on the ARM stack than an `int`. On the ARM, both these assumptions are wrong. All ARM registers are 32-bit and all stack entries are at least 32-bit. Furthermore, to implement the `i++` exactly, the compiler must account for the case when `i` = 255. Any attempt to increment 255 should produce the answer 0.

Consider the compiler output for this function. We've added labels and comments to make the assembly clear.

```
checksum_v1
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v1_loop
        LDR     r3,[r2,r1,LSL #2]   ; r3 = data[i]
        ADD     r1,r1,#1            ; r1 = i+1
        AND     r1,r1,#0xff         ; i = (char)r1
        CMP     r1,#0x40            ; compare i, 64
        ADD     r0,r3,r0            ; sum += r3
        BCC     checksum_v1_loop    ; if (i<64) loop
        MOV     pc,r14              ; return sum
```

Now compare this to the compiler output where instead we declare `i` as an `unsigned int`.

```
checksum_v2
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v2_loop
        LDR     r3,[r2,r1,LSL #2]   ; r3 = data[i]
        ADD     r1,r1,#1            ; r1++
        CMP     r1,#0x40            ; compare i, 64
        ADD     r0,r3,r0            ; sum += r3
        BCC     checksum_v2_loop    ; if (i<64) goto loop
        MOV     pc,r14              ; return sum
```

In the first case, the compiler inserts an extra AND instruction to reduce `i` to the range 0 to 255 before the comparison with 64. This instruction disappears in the second case.

Next, suppose the data packet contains 16-bit values and we need a 16-bit checksum. It is tempting to write the following C code:

```
short checksum_v3(short *data)
{
  unsigned int i;
  short sum=0;

  for (i=0; i<64; i++)
  {
    sum = (short)(sum + data[i]);
  }
  return sum;
}
```

You may wonder why the `for` loop body doesn't contain the code

```
sum += data[i];
```

With *armcc* this code will produce a warning if you enable implicit narrowing cast warnings using the compiler switch `-W+n`. The expression `sum+data[i]` is an integer and so can only be assigned to a `short` using an (implicit or explicit) narrowing cast. As you can see in the following assembly output, the compiler must insert extra instructions to implement the narrowing cast:

```
checksum_v3
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v3_loop
        ADD     r3,r2,r1,LSL #1     ; r3 = &data[i]
        LDRH    r3,[r3,#0]          ; r3 = data[i]
        ADD     r1,r1,#1            ; i++
        CMP     r1,#0x40            ; compare i, 64
        ADD     r0,r3,r0            ; r0 = sum + r3
        MOV     r0,r0,LSL #16
        MOV     r0,r0,ASR #16       ; sum = (short)r0
        BCC     checksum_v3_loop    ; if (i<64) goto loop
        MOV     pc,r14              ; return sum
```

The loop is now three instructions longer than the loop for example `checksum_v2` earlier! There are two reasons for the extra instructions:

- The `LDRH` instruction does not allow for a shifted address offset as the `LDR` instruction did in `checksum_v2`. Therefore the first `ADD` in the loop calculates the address of item `i` in the array. The `LDRH` loads from an address with no offset. `LDRH` has fewer addressing modes than `LDR` as it was a later addition to the ARM instruction set. (See Table 5.1.)
- The cast reducing `total+array[i]` to a `short` requires two `MOV` instructions. The compiler shifts left by 16 and then right by 16 to implement a 16-bit sign extend. The shift right is a sign-extending shift so it replicates the sign bit to fill the upper 16 bits.

We can avoid the second problem by using an `int` type variable to hold the partial sum. We only reduce the sum to a `short` type at the function exit.

However, the first problem is a new issue. We can solve it by accessing the array by incrementing the pointer *data* rather than using an index as in `data[i]`. This is efficient regardless of array type size or element size. All ARM load and store instructions have a postincrement addressing mode.

Example 5.1 The `checksum_v4` code fixes all the problems we have discussed in this section. It uses `int` type local variables to avoid unnecessary casts. It increments the pointer `data` instead of using an index offset `data[i]`.

```
short checksum_v4(short *data)
{
  unsigned int i;
  int sum=0;

  for (i=0; i<64; i++)
  {
    sum += *(data++);
  }
  return (short)sum;
}
```

The `*(data++)` operation translates to a single ARM instruction that loads the data and increments the `data` pointer. Of course you could write `sum += *data; data++;` or even `*data++` instead if you prefer. The compiler produces the following output. Three instructions have been removed from the inside loop, saving three cycles per loop compared to `checksum_v3`.

```
checksum_v4
        MOV      r2,#0              ; sum = 0
        MOV      r1,#0              ; i = 0
checksum_v4_loop
        LDRSH    r3,[r0],#2         ; r3 = *(data++)
        ADD      r1,r1,#1           ; i++
        CMP      r1,#0x40           ; compare i, 64
        ADD      r2,r3,r2           ; sum += r3
        BCC      checksum_v4_loop   ; if (sum<64) goto loop
        MOV      r0,r2,LSL #16
        MOV      r0,r0,ASR #16      ; r0 = (short)sum
        MOV      pc,r14             ; return r0
```

The compiler is still performing one cast to a 16-bit range, on the function return. You could remove this also by returning an `int` result as discussed in Section 5.2.2. ■

5.2.2 Function Argument Types

We saw in Section 5.2.1 that converting local variables from types `char` or `short` to type `int` increases performance and reduces code size. The same holds for function arguments. Consider the following simple function, which adds two 16-bit values, halving the second, and returns a 16-bit sum:

```
short add_v1(short a, short b)
{
  return a + (b>>1);
}
```

This function is a little artificial, but it is a useful test case to illustrate the problems faced by the compiler. The input values `a`, `b`, and the return value will be passed in 32-bit ARM registers. Should the compiler assume that these 32-bit values are in the range of a `short` type, that is, −32,768 to +32,767? Or should the compiler force values to be in this range by sign-extending the lowest 16 bits to fill the 32-bit register? The compiler must make compatible decisions for the function caller and callee. Either the caller or callee must perform the cast to a `short` type.

We say that function arguments are passed *wide* if they are not reduced to the range of the type and *narrow* if they are. You can tell which decision the compiler has made by looking at the assembly output for `add_v1`. If the compiler passes arguments wide, then the callee must reduce function arguments to the correct range. If the compiler passes arguments narrow, then the caller must reduce the range. If the compiler returns values wide, then the caller must reduce the return value to the correct range. If the compiler returns values narrow, then the callee must reduce the range before returning the value.

For *armcc* in ADS, function arguments are passed narrow and values returned narrow. In other words, the caller casts argument values and the callee casts return values. The compiler uses the ANSI prototype of the function to determine the datatypes of the function arguments.

The *armcc* output for `add_v1` shows that the compiler casts the return value to a `short` type, but does not cast the input values. It assumes that the caller has already ensured that the 32-bit values `r0` and `r1` are in the range of the `short` type. This shows narrow passing of arguments and return value.

```
add_v1
        ADD     r0,r0,r1,ASR #1     ; r0 = (int)a + ((int)b>>1)
        MOV     r0,r0,LSL #16
        MOV     r0,r0,ASR #16       ; r0 = (short)r0
        MOV     pc,r14              ; return r0
```

The *gcc* compiler we used is more cautious and makes no assumptions about the range of argument value. This version of the compiler reduces the input arguments to the range of a short in both the caller and the callee. It also casts the return value to a `short` type. Here is the compiled code for `add_v1`:

```
add_v1_gcc
        MOV      r0, r0, LSL #16
        MOV      r1, r1, LSL #16
        MOV      r1, r1, ASR #17         ; r1 = (int)b>>1
        ADD      r1, r1, r0, ASR #16     ; r1 += (int)a
        MOV      r1, r1, LSL #16
        MOV      r0, r1, ASR #16         ; r0 = (short)r1
        MOV      pc, lr                  ; return r0
```

Whatever the merits of different narrow and wide calling protocols, you can see that `char` or `short` type function arguments and return values introduce extra casts. These increase code size and decrease performance. It is more efficient to use the `int` type for function arguments and return values, even if you are only passing an 8-bit value.

5.2.3 Signed versus Unsigned Types

The previous sections demonstrate the advantages of using `int` rather than a `char` or `short` type for local variables and function arguments. This section compares the efficiencies of `signed int` and `unsigned int`.

If your code uses addition, subtraction, and multiplication, then there is no performance difference between signed and unsigned operations. However, there is a difference when it comes to division. Consider the following short example that averages two integers:

```
int average_v1(int a, int b)
{
  return (a+b)/2;
}
```

This compiles to

```
average_v1
        ADD      r0,r0,r1             ; r0 = a + b
        ADD      r0,r0,r0,LSR #31     ; if (r0<0) r0++
        MOV      r0,r0,ASR #1         ; r0 = r0>>1
        MOV      pc,r14               ; return r0
```

Notice that the compiler adds one to the sum before shifting by right if the sum is negative. In other words it replaces $x/2$ by the statement:

```
(x<0) ? ((x+1)>>1): (x>>1)
```

It must do this because x is signed. In C on an ARM target, a divide by two is not a right shift if x is negative. For example, $-3 \gg 1 = -2$ but $-3/2 = -1$. Division rounds towards zero, but arithmetic right shift rounds towards $-\infty$.

It is more efficient to use unsigned types for divisions. The compiler converts unsigned power of two divisions directly to right shifts. For general divisions, the divide routine in the C library is faster for unsigned types. See Section 5.10 for discussion on avoiding divisions completely.

SUMMARY **The Efficient Use of C Types**

- For local variables held in registers, don't use a `char` or `short` type unless 8-bit or 16-bit modular arithmetic is necessary. Use the `signed` or `unsigned int` types instead. Unsigned types are faster when you use divisions.
- For array entries and global variables held in main memory, use the type with the smallest size possible to hold the required data. This saves memory footprint. The ARMv4 architecture is efficient at loading and storing all data widths provided you traverse arrays by incrementing the array pointer. Avoid using offsets from the base of the array with `short` type arrays, as `LDRH` does not support this.
- Use explicit casts when reading array entries or global variables into local variables, or writing local variables out to array entries. The casts make it clear that for fast operation you are taking a narrow width type stored in memory and expanding it to a wider type in the registers. Switch on *implicit narrowing cast* warnings in the compiler to detect implicit casts.
- Avoid implicit or explicit narrowing casts in expressions because they usually cost extra cycles. Casts on loads or stores are usually free because the load or store instruction performs the cast for you.
- Avoid `char` and `short` types for function arguments or return values. Instead use the `int` type even if the range of the parameter is smaller. This prevents the compiler performing unnecessary casts.

5.3 C Looping Structures

This section looks at the most efficient ways to code `for` and `while` loops on the ARM. We start by looking at loops with a fixed number of iterations and then move on to loops with a variable number of iterations. Finally we look at loop unrolling.

5.3.1 Loops with a Fixed Number of Iterations

What is the most efficient way to write a `for` loop on the ARM? Let's return to our checksum example and look at the looping structure.

Here is the last version of the 64-word packet checksum routine we studied in Section 5.2. This shows how the compiler treats a loop with incrementing count `i++`.

```
int checksum_v5(int *data)
{
  unsigned int i;
  int sum=0;

  for (i=0; i<64; i++)
  {
    sum += *(data++);
  }
  return sum;
}
```

This compiles to

```
checksum_v5
        MOV     r2,r0              ; r2 = data
        MOV     r0,#0              ; sum = 0
        MOV     r1,#0              ; i = 0
checksum_v5_loop
        LDR     r3,[r2],#4         ; r3 = *(data++)
        ADD     r1,r1,#1           ; i++
        CMP     r1,#0x40           ; compare i, 64
        ADD     r0,r3,r0           ; sum += r3
        BCC     checksum_v5_loop   ; if (i<64) goto loop
        MOV     pc,r14             ; return sum
```

It takes three instructions to implement the `for` loop structure:

- An `ADD` to increment `i`
- A compare to check if `i` is less than 64
- A conditional branch to continue the loop if `i` < 64

This is not efficient. On the ARM, a loop should only use two instructions:

- A subtract to decrement the loop counter, which also sets the condition code flags on the result
- A conditional branch instruction

The key point is that the loop counter should count down to zero rather than counting up to some arbitrary limit. Then the comparison with zero is free since the result is stored in the condition flags. Since we are no longer using i as an array index, there is no problem in counting down rather than up.

EXAMPLE 5.2 This example shows the improvement if we switch to a decrementing loop rather than an incrementing loop.

```
int checksum_v6(int *data)
{
  unsigned int i;
  int sum=0;

  for (i=64; i!=0; i--)
  {
    sum += *(data++);
  }
  return sum;
}
```

This compiles to

```
checksum_v6
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0x40            ; i = 64
checksum_v6_loop
        LDR     r3,[r2],#4          ; r3 = *(data++)
        SUBS    r1,r1,#1            ; i-- and set flags
        ADD     r0,r3,r0            ; sum += r3
        BNE     checksum_v6_loop    ; if (i!=0) goto loop
        MOV     pc,r14              ; return sum
```

The SUBS and BNE instructions implement the loop. Our checksum example now has the minimum number of four instructions per loop. This is much better than six for checksum_v1 and eight for checksum_v3. ■

For an unsigned loop counter i we can use either of the loop continuation conditions i!=0 or i>0. As i can't be negative, they are the same condition. For a signed loop counter, it is tempting to use the condition i>0 to continue the loop. You might expect the compiler to generate the following two instructions to implement the loop:

```
SUBS   r1,r1,#1   ; compare i with 1, i=i-1
BGT    loop       ; if (i+1>1) goto loop
```

In fact, the compiler will generate

```
SUB   r1,r1,#1       ; i--
CMP   r1,#0          ; compare i with 0
BGT   loop           ; if (i>0) goto loop
```

The compiler is not being inefficient. It must be careful about the case when `i = -0x80000000` because the two sections of code generate different answers in this case. For the first piece of code the `SUBS` instruction compares `i` with 1 and then decrements `i`. Since `-0x80000000` < 1, the loop terminates. For the second piece of code, we decrement `i` and then compare with 0. Modulo arithmetic means that `i` now has the value `+0x7fffffff`, which is greater than zero. Thus the loop continues for many iterations.

Of course, in practice, `i` rarely takes the value `-0x80000000`. The compiler can't usually determine this, especially if the loop starts with a variable number of iterations (see Section 5.3.2).

Therefore you should use the termination condition `i!=0` for signed or unsigned loop counters. It saves one instruction over the condition `i>0` for signed `i`.

5.3.2 Loops Using a Variable Number of Iterations

Now suppose we want our checksum routine to handle packets of arbitrary size. We pass in a variable `N` giving the number of words in the data packet. Using the lessons from the last section we count down until `N` = 0 and don't require an extra loop counter `i`.

The `checksum_v7` example shows how the compiler handles a `for` loop with a variable number of iterations `N`.

```
int checksum_v7(int *data, unsigned int N)
{
  int sum=0;

  for (; N!=0; N--)
  {
    sum += *(data++);
  }
  return sum;
}
```

This compiles to

```
checksum_v7
        MOV    r2,#0            ; sum = 0
        CMP    r1,#0            ; compare N, 0
        BEQ    checksum_v7_end  ; if (N==0) goto end
```

```
checksum_v7_loop
        LDR     r3,[r0],#4          ; r3 = *(data++)
        SUBS    r1,r1,#1            ; N-- and set flags
        ADD     r2,r3,r2            ; sum += r3
        BNE     checksum_v7_loop    ; if (N!=0) goto loop
checksum_v7_end
        MOV     r0,r2               ; r0 = sum
        MOV     pc,r14              ; return r0
```

Notice that the compiler checks that `N` is nonzero on entry to the function. Often this check is unnecessary since you know that the array won't be empty. In this case a `do-while` loop gives better performance and code density than a `for` loop.

Example 5.3 This example shows how to use a `do-while` loop to remove the test for `N` being zero that occurs in a `for` loop.

```
int checksum_v8(int *data, unsigned int N)
{
  int sum=0;

  do
  {
    sum += *(data++);
  } while (--N!=0);
  return sum;
}
```

The compiler output is now

```
checksum_v8
        MOV     r2,#0               ; sum = 0
checksum_v8_loop
        LDR     r3,[r0],#4          ; r3 = *(data++)
        SUBS    r1,r1,#1            ; N-- and set flags
        ADD     r2,r3,r2            ; sum += r3
        BNE     checksum_v8_loop    ; if (N!=0) goto loop
        MOV     r0,r2               ; r0 = sum
        MOV     pc,r14              ; return r0
```

Compare this with the output for `checksum_v7` to see the two-cycle saving. ■

5.3.3 Loop Unrolling

We saw in Section 5.3.1 that each loop iteration costs two instructions in addition to the body of the loop: a subtract to decrement the loop count and a conditional branch.

We call these instructions the *loop overhead.* On ARM7 or ARM9 processors the subtract takes one cycle and the branch three cycles, giving an overhead of four cycles per loop.

You can save some of these cycles by *unrolling* a loop—repeating the loop body several times, and reducing the number of loop iterations by the same proportion. For example, let's unroll our packet checksum example four times.

EXAMPLE 5.4 The following code unrolls our packet checksum loop by four times. We assume that the number of words in the packet N is a multiple of four.

```
int checksum_v9(int *data, unsigned int N)
{
  int sum=0;

  do
  {
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
    N -= 4;
  } while ( N!=0);
  return sum;
}
```

This compiles to

```
checksum_v9
        MOV r2,#0    ; sum = 0
checksum_v9_loop
        LDR          r3,[r0],#4         ; r3 = *(data++)
        SUBS         r1,r1,#4           ; N -= 4 & set flags
        ADD          r2,r3,r2           ; sum += r3
        LDR          r3,[r0],#4         ; r3 = *(data++)
        ADD          r2,r3,r2           ; sum += r3
        LDR          r3,[r0],#4         ; r3 = *(data++)
        ADD          r2,r3,r2           ; sum += r3
        LDR          r3,[r0],#4         ; r3 = *(data++)
        ADD          r2,r3,r2           ; sum += r3
        BNE          checksum_v9_loop   ; if (N!=0) goto loop
        MOV          r0,r2              ; r0 = sum
        MOV          pc,r14             ; return r0
```

We have reduced the loop overhead from 4N cycles to (4N)/4 = N cycles. On the ARM7TDMI, this accelerates the loop from 8 cycles per accumulate to 20/4 = 5 cycles per accumulate, nearly doubling the speed! For the ARM9TDMI, which has a faster load instruction, the benefit is even higher. ■

There are two questions you need to ask when unrolling a loop:

- How many times should I unroll the loop?
- What if the number of loop iterations is not a multiple of the unroll amount? For example, what if N is not a multiple of four in `checksum_v9`?

To start with the first question, only unroll loops that are important for the overall performance of the application. Otherwise unrolling will increase the code size with little performance benefit. Unrolling may even reduce performance by evicting more important code from the cache.

Suppose the loop is important, for example, 30% of the entire application. Suppose you unroll the loop until it is 0.5 KB in code size (128 instructions). Then the loop overhead is at most 4 cycles compared to a loop body of around 128 cycles. The loop overhead cost is 3/128, roughly 3%. Recalling that the loop is 30% of the entire application, overall the loop overhead is only 1%. Unrolling the code further gains little extra performance, but has a significant impact on the cache contents. It is usually not worth unrolling further when the gain is less than 1%.

For the second question, try to arrange it so that array sizes are multiples of your unroll amount. If this isn't possible, then you must add extra code to take care of the leftover cases. This increases the code size a little but keeps the performance high.

Example 5.5 This example handles the checksum of any size of data packet using a loop that has been unrolled four times.

```
int checksum_v10(int *data, unsigned int N)
{
  unsigned int i;
  int sum=0;

  for (i=N/4; i!=0; i--)
  {
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
  }
  for (i=N&3; i!=0; i--)
  {
      sum += *(data++);
    }
    return sum;
  }
```

The second `for` loop handles the remaining cases when `N` is not a multiple of four. Note that both `N/4` and `N&3` can be zero, so we can't use `do-while` loops. ■

SUMMARY **Writing Loops Efficiently**

- Use loops that count down to zero. Then the compiler does not need to allocate a register to hold the termination value, and the comparison with zero is free.
- Use unsigned loop counters by default and the continuation condition `i!=0` rather than `i>0`. This will ensure that the loop overhead is only two instructions.
- Use `do-while` loops rather than `for` loops when you know the loop will iterate at least once. This saves the compiler checking to see if the loop count is zero.
- Unroll important loops to reduce the loop overhead. Do not overunroll. If the loop overhead is small as a proportion of the total, then unrolling will increase code size and hurt the performance of the cache.
- Try to arrange that the number of elements in arrays are multiples of four or eight. You can then unroll loops easily by two, four, or eight times without worrying about the leftover array elements.

5.4 Register Allocation

The compiler attempts to allocate a processor register to each local variable you use in a C function. It will try to use the same register for different local variables if the use of the variables do not overlap. When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called *spilled* or *swapped out* variables since they are written out to memory (in a similar way virtual memory is swapped out to disk). Spilled variables are slow to access compared to variables allocated to registers.

To implement a function efficiently, you need to

- minimize the number of spilled variables
- ensure that the most important and frequently accessed variables are stored in registers

First let's look at the number of processor registers the ARM C compilers have available for allocating variables. Table 5.3 shows the standard register names and usage when following the ARM-Thumb procedure call standard (ATPCS), which is used in code generated by C compilers.

Table 5.3 C compiler register usage.

Register number	Alternate register names	ATPCS register usage
r0	*a1*	Argument registers. These hold the first four function arguments on a function call and the return value on a function return. A function may corrupt these registers and use them as general scratch registers within the function.
r1	*a2*	
r2	*a3*	
r3	*a4*	
r4	*v1*	General variable registers. The function must preserve the callee values of these registers.
r5	*v2*	
r6	*v3*	
r7	*v4*	
r8	*v5*	
r9	*v6 sb*	General variable register. The function must preserve the callee value of this register except when compiling for *read-write position independence* (RWPI). Then *r9* holds the *static base* address. This is the address of the read-write data.
r10	*v7 sl*	General variable register. The function must preserve the callee value of this register except when compiling with stack limit checking. Then *r10* holds the stack limit address.
r11	*v8 fp*	General variable register. The function must preserve the callee value of this register except when compiling using a frame pointer. Only old versions of *armcc* use a frame pointer.
r12	*ip*	A general scratch register that the function can corrupt. It is useful as a scratch register for function veneers or other intraprocedure call requirements.
r13	*sp*	The stack pointer, pointing to the full descending stack.
r14	*lr*	The link register. On a function call this holds the return address.
r15	*pc*	The program counter.

Provided the compiler is not using software stack checking or a frame pointer, then the C compiler can use registers *r0* to *r12* and *r14* to hold variables. It must save the callee values of *r4* to *r11* and *r14* on the stack if using these registers.

In theory, the C compiler can assign 14 variables to registers without spillage. In practice, some compilers use a fixed register such as *r12* for intermediate scratch working and do not assign variables to this register. Also, complex expressions require intermediate working registers to evaluate. Therefore, to ensure good assignment to registers, you should try to limit the internal loop of functions to using at most 12 local variables.

If the compiler does need to swap out variables, then it chooses which variables to swap out based on frequency of use. A variable used inside a loop counts multiple times. You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

The `register` keyword in C hints that a compiler should allocate the given variable to a register. However, different compilers treat this keyword in different ways, and different architectures have a different number of available registers (for example, Thumb and ARM). Therefore we recommend that you avoid using `register` and rely on the compiler's normal register allocation routine.

Summary **Efficient Register Allocation**

- Try to limit the number of local variables in the internal loop of functions to 12. The compiler should be able to allocate these to ARM registers.
- You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

5.5 Function Calls

The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers. The more recent ARM-Thumb Procedure Call Standard (ATPCS) covers ARM and Thumb interworking as well.

The first four integer arguments are passed in the first four ARM registers: *r0, r1, r2*, and *r3*. Subsequent integer arguments are placed on the full descending stack, ascending in memory as in Figure 5.1. Function return integer values are passed in *r0*.

This description covers only integer or pointer arguments. Two-word arguments such as `long long` or `double` are passed in a pair of consecutive argument registers and returned in *r0, r1*. The compiler may pass structures in registers or by reference according to command line compiler options.

The first point to note about the procedure call standard is the *four-register rule*. Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments. For functions with four or fewer arguments, the compiler can pass all the arguments in registers. For functions with more arguments, both the caller and callee must access the stack for some arguments. Note that for C++ the first argument to an object method is the *this* pointer. This argument is implicit and additional to the explicit arguments.

If your C function needs more than four arguments, or your C++ method more than three explicit arguments, then it is almost always more efficient to use structures. Group related arguments into structures, and pass a structure pointer rather than multiple arguments. Which arguments are related will depend on the structure of your software.

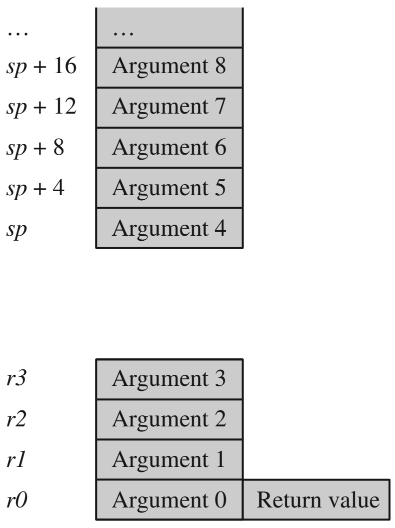

Figure 5.1 ATPCS argument passing.

The next example illustrates the benefits of using a structure pointer. First we show a typical routine to insert `N` bytes from array `data` into a queue. We implement the queue using a cyclic buffer with start address `Q_start` (inclusive) and end address `Q_end` (exclusive).

```
char *queue_bytes_v1(
  char *Q_start,        /* Queue buffer start address */
  char *Q_end,          /* Queue buffer end address */
  char *Q_ptr,          /* Current queue pointer position */
  char *data,           /* Data to insert into the queue */
  unsigned int N)       /* Number of bytes to insert */
{
  do
  {
    *(Q_ptr++) = *(data++);

    if (Q_ptr == Q_end)
    {
      Q_ptr = Q_start;
    }
  } while (--N);
  return Q_ptr;
}
```

This compiles to

```
queue_bytes_v1
        STR     r14,[r13,#-4]!   ; save lr on the stack
        LDR     r12,[r13,#4]     ; r12 = N
queue_v1_loop
        LDRB    r14,[r3],#1      ; r14 = *(data++)
        STRB    r14,[r2],#1      ; *(Q_ptr++) = r14
        CMP     r2,r1            ; if (Q_ptr == Q_end)
        MOVEQ   r2,r0            ;    {Q_ptr = Q_start;}
        SUBS    r12,r12,#1       ; --N and set flags
        BNE     queue_v1_loop    ; if (N!=0) goto loop
        MOV     r0,r2            ; r0 = Q_ptr
        LDR     pc,[r13],#4      ; return r0
```

Compare this with a more structured approach using three function arguments.

EXAMPLE 5.6 The following code creates a `Queue` structure and passes this to the function to reduce the number of function arguments.

```
typedef struct {
  char *Q_start;        /* Queue buffer start address */
  char *Q_end;          /* Queue buffer end address */
  char *Q_ptr;          /* Current queue pointer position */
} Queue;

void queue_bytes_v2(Queue *queue, char *data, unsigned int N)
{
  char *Q_ptr = queue->Q_ptr;
  char *Q_end = queue->Q_end;

  do
  {
    *(Q_ptr++) = *(data++);

    if (Q_ptr == Q_end)
    {
      Q_ptr = queue->Q_start;
    }
  } while (--N);
  queue->Q_ptr = Q_ptr;
}
```

This compiles to

```
queue_bytes_v2
        STR     r14,[r13,#-4]!   ; save lr on the stack
        LDR     r3,[r0,#8]       ; r3 = queue->Q_ptr
        LDR     r14,[r0,#4]      ; r14 = queue->Q_end
queue_v2_loop
        LDRB    r12,[r1],#1      ; r12 = *(data++)
        STRB    r12,[r3],#1      ; *(Q_ptr++) = r12
        CMP     r3,r14           ; if (Q_ptr == Q_end)
        LDREQ   r3,[r0,#0]       ;    Q_ptr = queue->Q_start
        SUBS    r2,r2,#1         ; --N and set flags
        BNE     queue_v2_loop    ; if (N!=0) goto loop
        STR     r3,[r0,#8]       ; queue->Q_ptr = r3
        LDR     pc,[r13],#4      ; return
```

■

The `queue_bytes_v2` is one instruction longer than `queue_bytes_v1`, but it is in fact more efficient overall. The second version has only three function arguments rather than five. Each call to the function requires only three register setups. This compares with four register setups, a stack push, and a stack pull for the first version. There is a net saving of two instructions in function call overhead. There are likely further savings in the callee function, as it only needs to assign a single register to the `Queue` structure pointer, rather than three registers in the nonstructured case.

There are other ways of reducing function call overhead if your function is very small and corrupts few registers (uses few local variables). Put the C function in the same C file as the functions that will call it. The C compiler then knows the code generated for the callee function and can make optimizations in the caller function:

- The caller function need not preserve registers that it can see the callee doesn't corrupt. Therefore the caller function need not save all the ATPCS corruptible registers.
- If the callee function is very small, then the compiler can inline the code in the caller function. This removes the function call overhead completely.

Example 5.7 The function `uint_to_hex` converts a 32-bit unsigned integer into an array of eight hexadecimal digits. It uses a helper function `nybble_to_hex`, which converts a digit `d` in the range 0 to 15 to a hexadecimal digit.

```
unsigned int nybble_to_hex(unsigned int d)
{
  if (d<10)
  {
    return d + '0';
  }
  return d - 10 + 'A';
}

void uint_to_hex(char *out, unsigned int in)
{
  unsigned int i;

  for (i=8; i!=0; i--)
  {
    in = (in<<4) | (in>>28); /* rotate in left by 4 bits */
    *(out++) = (char)nybble_to_hex(in & 15);
  }
}
```

When we compile this, we see that `uint_to_hex` doesn't call `nybble_to_hex` at all! In the following compiled code, the compiler has inlined the `uint_to_hex` code. This is more efficient than generating a function call.

```
uint_to_hex
        MOV     r3,#8             ; i = 8
uint_to_hex_loop
        MOV     r1,r1,ROR #28     ; in = (in<<4)|(in>>28)
        AND     r2,r1,#0xf        ; r2 = in & 15
        CMP     r2,#0xa           ; if (r2>=10)
        ADDCS   r2,r2,#0x37       ;    r2 +='A'-10
        ADDCC   r2,r2,#0x30       ; else r2 +='0'
        STRB    r2,[r0],#1        ; *(out++) = r2
        SUBS    r3,r3,#1          ; i-- and set flags
        BNE     uint_to_hex_loop  ; if (i!=0) goto loop
        MOV     pc,r14            ; return
```

The compiler will only inline small functions. You can ask the compiler to inline a function using the `__inline` keyword, although this keyword is only a hint and the compiler may ignore it (see Section 5.12 for more on inline functions). Inlining large functions can lead to big increases in code size without much performance improvement.

SUMMARY **Calling Functions Efficiently**

- Try to restrict functions to four arguments. This will make them more efficient to call. Use structures to group related arguments and pass structure pointers instead of multiple arguments.

- Define small functions in the same source file and before the functions that call them. The compiler can then optimize the function call or inline the small function.
- Critical functions can be inlined using the `__inline` keyword.

5.6 Pointer Aliasing

Two pointers are said to *alias* when they point to the same address. If you write to one pointer, it will affect the value you read from the other pointer. In a function, the compiler often doesn't know which pointers can alias and which pointers can't. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

Let's start with a very simple example. The following function increments two timer values by a step amount:

```
void timers_v1(int *timer1, int *timer2, int *step)
{
  *timer1 += *step;
  *timer2 += *step;
}
```

This compiles to

```
timers_v1
        LDR     r3,[r0,#0]          ; r3 = *timer1
        LDR     r12,[r2,#0]         ; r12 = *step
        ADD     r3,r3,r12           ; r3 += r12
        STR     r3,[r0,#0]          ; *timer1 = r3
        LDR     r0,[r1,#0]          ; r0 = *timer2
        LDR     r2,[r2,#0]          ; r2 = *step
        ADD     r0,r0,r2            ; r0 += r2
        STR     r0,[r1,#0]          ; *timer2 = t0
        MOV     pc,r14              ; return
```

Note that the compiler loads from `step` twice. Usually a compiler optimization called *common subexpression elimination* would kick in so that `*step` was only evaluated once, and the value reused for the second occurrence. However, the compiler can't use this optimization here. The pointers `timer1` and `step` might alias one another. In other words, the compiler cannot be sure that the write to `timer1` doesn't affect the read from `step`.

In this case the second value of `*step` is different from the first and has the value `*timer1`. This forces the compiler to insert an extra load instruction.

The same problem occurs if you use structure accesses rather than direct pointer access. The following code also compiles inefficiently:

```
typedef struct {int step;} State;
typedef struct {int timer1, timer2;} Timers;

void timers_v2(State *state, Timers *timers)
{
  timers->timer1 += state->step;
  timers->timer2 += state->step;
}
```

The compiler evaluates `state->step` twice in case `state->step` and `timers->timer1` are at the same memory address. The fix is easy: Create a new local variable to hold the value of `state->step` so the compiler only performs a single load.

EXAMPLE 5.8 In the code for `timers_v3` we use a local variable `step` to hold the value of `state->step`. Now the compiler does not need to worry that `state` may alias with `timers`.

```
void timers_v3(State *state, Timers *timers)
{
     int step = state->step;

     timers->timer1 += step;
     timers->timer2 += step;
}
```

You must also be careful of other, less obvious situations where aliasing may occur. When you call another function, this function may alter the state of memory and so change the values of any expressions involving memory reads. The compiler will evaluate the expressions again. For example suppose you read `state->step`, call a function and then read `state->step` again. The compiler must assume that the function could change the value of `state->step` in memory. Therefore it will perform two reads, rather than reusing the first value it read for `state->step`.

Another pitfall is to take the address of a local variable. Once you do this, the variable is referenced by a pointer and so aliasing can occur with other pointers. The compiler is likely to keep reading the variable from the stack in case aliasing occurs. Consider the following example, which reads and then checksums a data packet:

```
int checksum_next_packet(void)
{
  int *data;
  int N, sum=0;
  data = get_next_packet(&N);

  do
  {
    sum += *(data++);
  } while (--N);

  return sum;
}
```

Here `get_next_packet` is a function returning the address and size of the next data packet. The previous code compiles to

```
checksum_next_packet
        STMFD   r13!,{r4,r14}        ; save r4, lr on the stack
        SUB     r13,r13,#8           ; create two stacked variables
        ADD     r0,r13,#4            ; r0 = &N, N stacked
        MOV     r4,#0                ; sum = 0
        BL      get_next_packet      ; r0 = data
checksum_loop
        LDR     r1,[r0],#4           ; r1 = *(data++)
        ADD     r4,r1,r4             ; sum += r1
        LDR     r1,[r13,#4]          ; r1 = N (read from stack)
        SUBS    r1,r1,#1             ; r1-- & set flags
        STR     r1,[r13,#4]          ; N = r1 (write to stack)
        BNE     checksum_loop        ; if (N!=0) goto loop
        MOV     r0,r4                ; r0 = sum
        ADD     r13,r13,#8           ; delete stacked variables
        LDMFD   r13!,{r4,pc}         ; return r0
```

Note how the compiler reads and writes `N` from the stack for every `N--`. Once you take the address of `N` and pass it to `get_next_packet`, the compiler needs to worry about aliasing because the pointers `data` and `&N` may alias. To avoid this, don't take the address of local variables. If you must do this, then copy the value into another local variable before use.

You may wonder why the compiler makes room for two stacked variables when it only uses one. This is to keep the stack eight-byte aligned, which is required for `LDRD` instructions available in ARMv5TE. The example above doesn't actually use an `LDRD`, but the compiler does not know whether `get_next_packet` will use this instruction.

SUMMARY **Avoiding Pointer Aliasing**

- Do not rely on the compiler to eliminate common subexpressions involving memory accesses. Instead create new local variables to hold the expression. This ensures the expression is evaluated only once.
- Avoid taking the address of local variables. The variable may be inefficient to access from then on.

5.7 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: alignment of the structure entries and 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. Table 5.4 summarizes these restrictions.

For this reason, ARM compilers will automatically align the start address of a structure to a multiple of the largest access width used within the structure (usually four or eight bytes) 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`

Table 5.4 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

To improve the memory usage, you should reorder the elements

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

This reduces the structure size from 12 bytes to 8 bytes, with the following new layout:

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

Therefore, it is a good idea to group structure elements of the same size, so that the structure layout doesn't contain unnecessary padding. The *armcc* compiler does include a keyword `__packed` that removes all padding. For example, the structure

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

will be laid out in memory as

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

However, packed structures are slow and inefficient to access. The compiler emulates unaligned load and store operations by using several aligned accesses with data operations to merge the results. Only use the `__packed` keyword where space is far more important than speed and you can't reduce padding by rearragement. Also use it for porting code that assumes a certain structure layout in memory.

The exact layout of a structure in memory may depend on the compiler vendor and compiler version you use. In API (Application Programmer Interface) definitions it is often a good idea to insert any padding that you cannot get rid of into the structure manually. This way the structure layout is not ambiguous. It is easier to link code between compiler versions and compiler vendors if you stick to unambiguous structures.

Another point of ambiguity is enum. Different compilers use different sizes for an enumerated type, depending on the range of the enumeration. For example, consider the type

```
typedef enum {
  FALSE,
  TRUE
} Bool;
```

The *armcc* in ADS1.1 will treat Bool as a one-byte type as it only uses the values 0 and 1. Bool will only take up 8 bits of space in a structure. However, *gcc* will treat Bool as a word and take up 32 bits of space in a structure. To avoid ambiguity it is best to avoid using enum types in structures used in the API to your code.

Another consideration is the size of the structure and the offsets of elements within the structure. This problem is most acute when you are compiling for the Thumb instruction set. Thumb instructions are only 16 bits wide and so only allow for small element offsets from a structure base pointer. Table 5.5 shows the load and store base register offsets available in Thumb.

Therefore the compiler can only access an 8-bit structure element with a single instruction if it appears within the first 32 bytes of the structure. Similarly, single instructions can only access 16-bit values in the first 64 bytes and 32-bit values in the first 128 bytes. Once you exceed these limits, structure accesses become inefficient.

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.

Table 5.5 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)

SUMMARY **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 (that would appear implicitly) 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.

5.8 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. Bit-fields are structure elements and usually accessed using structure pointers; consequently, they suffer from the pointer aliasing problems described in Section 5.6. Every bit-field access is really a memory access. Possible pointer aliasing often forces the compiler to reload the bit-field several times.

The following example, `dostages_v1`, illustrates this problem. It also shows that compilers do not tend to optimize bit-field testing very well.

```
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
```

Note that the compiler accesses the memory location containing the bit-field three times. Because the bit-field is stored in memory, the `dostage` functions could change the value. Also, the compiler uses two instructions to test bit 1 and bit 2 of the bit-field, rather than a single instruction.

You can generate far more efficient code by using an integer rather than a bit-field. Use `enum` or `#define` masks to divide the integer type into different fields.

EXAMPLE 5.9 The following code implements the `dostages` function using logical operations rather than bit-fields:

```
typedef unsigned long Stages_v2;

#define STAGEA (1ul << 0)
```

```
#define STAGEB (1ul<<1)
#define STAGEC (1ul<<2)

void dostages_v2(Stages_v2 *stages_v2)
{
  Stages_v2 stages = *stages_v2;

  if (stages & STAGEA)
  {
    dostageA();
  }
  if (stages & STAGEB)
  {
    dostageB();
  }
  if (stages & STAGEC)
  {
    dostageC();
  }
}
```

Now that a single `unsigned long` type contains all the bit-fields, we can keep a copy of their values in a single local variable `stages`, which removes the memory aliasing problem discussed in Section 5.6. In other words, the compiler must assume that the `dostage`*X* (where *X* is `A`, `B`, or `C`) functions could change the value of `*stages_v2`.

The compiler generates the following code giving a saving of 33% over the previous version using ANSI bit-fields:

```
dostages_v2
        STMFD     r13!,{r4,r14}   ; stack r4, lr
        LDR       r4,[r0,#0]      ; stages = *stages_v2
        TST       r4,#1           ; if (stage & STAGEA)
        BLNE      dostageA        ;    {dostageA();}
        TST       r4,#2           ; if (stage & STAGEB)
        BLNE      dostageB        ;    {dostageB();}
        TST       r4,#4           ; if (!(stage & STAGEC))
        LDMNEFD   r13!,{r4,r14}   ; return;
        BNE       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 ^= STAGEC;           /* toggle stage C */
```

These bit set, clear, and toggle operations take only one ARM instruction each, using `ORR`, `BIC`, and `EOR` instructions, respectively. Another advantage is that you can now manipulate several bit-fields at the same time, using one instruction. For example:

```
stages |= (STAGEA | STAGEB);          /* enable stages A and B */
stages &= ~(STAGEA | STAGEC);         /* disable stages A and C */
```

Summary **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.

5.9 Unaligned Data and Endianness

Unaligned data and endianness are two issues that can complicate memory accesses and portability. Is the array pointer aligned? Is the ARM configured for a big-endian or little-endian memory system?

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.

To illustrate this, look at the following simple routine, `readint`. It returns the integer at the address pointed to by `data`. We've used `__packed` to tell the compiler that the integer may possibly not be aligned.

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

This compiles to

```
readint
        BIC     r3,r0,#3            ; r3 = data & 0xFFFFFFFC
        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
```

Notice how large and complex the code is. The compiler emulates the unaligned access using two aligned accesses and data processing operations, which is very costly and shows why you should avoid `_packed`. Instead use the `type char *` to point to data that can appear at any alignment. We will look at more efficient ways to read 32-bit words from a `char *` later.

You are likely to meet alignment problems when reading data packets or files used to transfer information between computers. Network packets and compressed image files are good examples. Two- or four-byte integers may appear at arbitrary offsets in these files. Data has been squeezed as much as possible, to the detriment of alignment.

Endianness (or byte order) is also a big issue when reading data packets or compressed files. The ARM core can be configured to work in *little-endian* (least significant byte at lowest address) or *big-endian* (most significant byte at lowest address) modes. Little-endian mode is usually the default.

The endianness of an ARM is usually set at power-up and remains fixed thereafter. Tables 5.6 and 5.7 illustrate how the ARM's 8-bit, 16-bit, and 32-bit load and store instructions work for different endian configurations. We assume that byte address A is aligned to

Table 5.6 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)

Table 5.7 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.

the size of the memory transfer. The tables show how the byte addresses in memory map into the 32-bit register that the instruction loads or stores.

What is the best way to deal with endian and alignment problems? If speed is not critical, then use functions like `readint_little` and `readint_big` in Example 5.10, which read a four-byte integer from a possibly unaligned address in memory. The address alignment is not known at compile time, only at run time. If you've loaded a file containing big-endian data such as a JPEG image, then use `readint_big`. For a bytestream containing little-endian data, use `readint_little`. Both routines will work correctly regardless of the memory endianness ARM is configured for.

EXAMPLE 5.10 These functions read a 32-bit integer from a bytestream pointed to by `data`. The bytestream contains little- or big-endian data, respectively. These functions are independent of the ARM memory system byte order since they only use byte accesses.

```
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.

EXAMPLE 5.11 The `read_samples` routine takes an array of `N` 16-bit sound samples at address `in`. The sound samples are little-endian (for example from a `.wav` file) and can be at any byte alignment. The routine copies the samples to an aligned array of `short` type values pointed to by `out`. The samples will be stored according to the configured ARM memory endianness.

The routine handles all cases in an efficient manner, regardless of input alignment and of ARM endianness configuration.

```
void read_samples(short *out, char *in, unsigned int N)
{
  unsigned short *data; /* aligned input pointer */
  unsigned int sample, next;

  switch ((unsigned int)in & 1)
  {
    case 0: /* the input pointer is aligned */
      data = (unsigned short *)in;
      do
      {
        sample = *(data++);
#ifdef __BIG_ENDIAN
        sample = (sample>>8) | (sample<<8);
#endif
        *(out++) = (short)sample;
      } while (--N);
      break;

    case 1: /* the input pointer is not aligned */
        data = (unsigned short *)(in-1);
        sample = *(data++);
#ifdef __BIG_ENDIAN
         sample = sample & 0xFF; /* get first byte of sample */
#else
         sample = sample>>8;     /* get first byte of sample */
#endif
         do
         {
         next = *(data++);
         /* complete one sample and start the next */
#ifdef __BIG_ENDIAN
         *out++ = (short)((next & 0xFF00) | sample);
         sample = next & 0xFF;
#else
         *out++ = (short)((next<<8) | sample);
         sample = next>>8;
#endif
         } while (--N);
         break;
    }
}
```

The routine works by having different code for each endianness and alignment. Endianness is dealt with at compile time using the `__BIG_ENDIAN` compiler flag. Alignment must be dealt with at run time using the `switch` statement.

You can make the routine even more efficient by using 32-bit reads and writes rather than 16-bit reads and writes, which leads to four elements in the switch statement, one for each possible address alignment modulo four. ■

SUMMARY **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.

5.10 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. We look at assembly division routines in detail in Chapter 7. The standard integer division routine provided in the C library can take between 20 and 100 cycles, depending on implementation, early termination, and the ranges of the input operands.

Division and modulus (/ and %) are such slow operations that you should avoid them as much as possible. However, division by a constant and repeated division by the same denominator can be handled efficiently. This section describes how to replace certain divisions by multiplications and how to minimize the number of division calls.

Circular buffers are one area where programmers often use division, but you can avoid these divisions completely. Suppose you have a circular buffer of size `buffer_size` bytes and a position indicated by a buffer `offset`. To advance the offset by `increment` bytes you could write

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

Instead it is far more efficient to write

```
offset += increment;
if (offset>=buffer_size)
{
   offset -= buffer_size;
}
```

The first version may take 50 cycles; the second will take 3 cycles because it does not involve a division. We've assumed that `increment < buffer_size`; you can always arrange this in practice.

If you can't avoid a division, then try to arrange that the numerator and denominator are unsigned integers. Signed division routines are slower since they take the absolute values of the numerator and denominator and then call the unsigned division routine. They fix the sign of the result afterwards.

Many C library division routines return the quotient and remainder from the division. In other words a free remainder operation is available to you with each division operation and vice versa. For example, to find the `(x, y)` position of a location at `offset` bytes into a screen buffer, it is tempting to write

```
typedef struct {
  int x;
  int y;
} point;

point getxy_v1(unsigned int offset, unsigned int bytes_per_line)
{
  point p;
  p.y = offset / bytes_per_line;
  p.x = offset - p.y * bytes_per_line;
  return p;
}
```

It appears that we have saved a division by using a subtract and multiply to calculate p.x, but in fact, it is often more efficient to write the function with the modulus or remainder operation.

EXAMPLE 5.12 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;
}
```

There is only one division call here, as you can see in the following compiler output. In fact, this version is four instructions shorter than getxy_v1. Note that this may not be the case for all compilers and C libraries.

```
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
```

5.10.1 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:

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

In these situations it is more efficient to *cache* the value of `1/z` in some way and use a multiplication by `1/z` instead of a division. We will show how to do this in the next subsection. We also want to stick to integer arithmetic and avoid floating point (see Section 5.11).

The next description is rather mathematical and covers the theory behind this conversion of repeated divisions into multiplications. If you are not interested in the theory, then don't worry. You can jump directly to Example 5.13, which follows.

5.10.2 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

$$\left(n(2^{32}/d)\right)/2^{32} \tag{5.1}$$

We need to perform the multiplication by n to 64-bit accuracy. There are a couple of problems with this approach:

- To calculate $2^{32}/d$, the compiler needs to use 64-bit `long long` type arithmetic because 2^{32} does not fit into an `unsigned int` type. We must specify the division as $(1\mathrm{ull} \ll 32)/d$. This 64-bit division is much slower than the 32-bit division we wanted to perform originally!
- If d happens to be 1, then $2^{32}/d$ will not fit into an `unsigned int` type.

It turns out that a slightly cruder estimate works well and fixes both these problems. Instead of $2^{32}/d$, we look at $(2^{32} - 1)/d$. Let

```
s = 0xFFFFFFFFul / d;  /* s = (2^32-1)/d */
```

We can calculate s using a single `unsigned int` type division. We know that

$$2^{32} - 1 = sd + t \text{ for some } 0 \le t < d \tag{5.2}$$

Therefore

$$s = \frac{2^{32}}{d} - e_1, \quad \text{where } 0 < e_1 = \frac{1+t}{d} \le 1 \tag{5.3}$$

Next, calculate an estimate q to n/d:

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

Mathematically, the shift right by 32 introduces an error e_2:

$$q = ns2^{-32} - e_2 \text{ for some } 0 \leq e_2 < 1 \tag{5.4}$$

Substituting the value of s:

$$q = \frac{n}{d} - ne_1 2^{-32} - e_2 \tag{5.5}$$

So, q is an underestimate to n/d. Now

$$0 \leq ne_1 2^{-32} + e_2 < e_1 + e_2 < 2 \tag{5.6}$$

Therefore

$$n/d - 2 < q \leq n/d \tag{5.7}$$

So $q = n/d$ or $q = (n/d) - 1$. We can find out which quite easily, by calculating the remainder $r = n - qd$, which must be in the range $0 \leq r < 2d$. The following code corrects the result:

```
r = n-q * d;      /* the remainder in the range 0 <= r < 2 * d */
if (r >= d)       /* if correction is required */
{
  r -= d;         /* correct the remainder to the range 0 <= r < d */
  q++;            /* correct the quotient */
}
/* now q = n / d and r = n % d */
```

EXAMPLE 5.13 The following routine, `scale`, shows how to convert divisions to multiplications in practice. It divides an array of `N` elements by denominator `d`. We first calculate the value of `s` as above. Then we replace each divide by `d` with a multiplication by `s`. The 64-bit multiply is cheap because the ARM has an instruction `UMULL`, which multiplies two 32-bit values, giving a 64-bit result.

```
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);
}
```

Here we have assumed that the numerator and denominator are 32-bit unsigned integers. Of course, the algorithm works equally well for 16-bit unsigned integers using a 32-bit multiply, or for 64-bit integers using a 128-bit multiply. You should choose the narrowest width for your data. If your data is 16-bit, then set $s = (2^{16} - 1)/d$ and estimate q using a standard integer C multiply. ■

5.10.3 Unsigned Division by a Constant

To divide by a constant c, you could use the algorithm of Example 5.13, precalculating $s = (2^{32} - 1)/c$. However, there is an even more efficient method. The ADS1.2 compiler uses this method to synthesize divisions 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:[1]

$$\text{If } 2^{N+k} \leq ds \leq 2^{N+k} + 2^k, \text{ then } n/d = (ns) \gg (N+k) \text{ for } 0 \leq n < 2^N. \tag{5.8}$$

$$\text{If } 2^{N+k} - 2^k \leq ds < 2^{N+k}, \text{ then } n/d = (ns + s) \gg (N+k) \text{ for } 0 \leq n < 2^N. \tag{5.9}$$

1. For the first result see a paper by Torbjorn Granlund and Peter L. Montgomery, “Division by Invariant Integers Using Multiplication,” in *proceedings of the SIG-PLAN PLDI’94 Conference*, June 1994.

Since $n = (n/d)d + r$ for $0 \le r \le d - 1$, the results follow from the equations

$$ns - (n/d)2^{N+k} = ns - \frac{n-r}{d}2^{N+k} = n\frac{ds - 2^{N+k}}{d} + \frac{r2^{N+k}}{d} \tag{5.10}$$

$$(n+1)s - (n/d)2^{N+k} = (n+1)\frac{ds - 2^{N+k}}{d} + \frac{(r+1)2^{N+k}}{d} \tag{5.11}$$

For both equations the right-hand side is in the range $0 \le x < 2^{N+k}$. For a 32-bit unsigned integer n, we take $N = 32$, choose k such that $2^k < d \le 2^{k+1}$, and set $s = (2^{N+k} + 2^k)/d$. If $ds \ge 2^{N+k}$, then $n/d = (ns) \gg (N + k)$; otherwise, $n/d = (ns + s) \gg (N + k)$. As an extra optimization, if d is a power of two, we can replace the division with a shift.

EXAMPLE 5.14 The `udiv_by_const` function tests the algorithm described above. In practice d will be a fixed constant rather than a variable. You can precalculate 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;
}
```

If you know that $0 \le n < 2^{31}$, as for a positive signed integer, then you don't need to bother with the different cases. You can increase k by one without having to worry about s overflowing. Take $N = 31$, choose k such that $2^{k-1} < d \le 2^k$, and set $s = (s^{N+k}+2^k-1)/d$. Then $n/d = (ns) \gg (N+k)$.

5.10.4 Signed Division by a Constant

We can use ideas and algorithms similar to those in Section 5.10.3 to 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 $d > 0$. The first mathematical result of Section 5.10.3 extends to signed n. If $d > 0$ and $2^{N+k} < ds \le 2^{N+k} + 2^k$, then

$$n/d = (ns) \gg (N+k) \text{ for all } 0 \le n < 2^N \qquad (5.12)$$

$$n/d = ((ns) \gg (N+k)) + 1 \text{ for all } -2^N \le n < 0 \qquad (5.13)$$

For 32-bit signed n, we take $N = 31$ and choose $k \le 31$ such that $2^{k-1} < d \le 2^k$. This ensures that we can find a 32-bit unsigned $s = (2^{N+k} + 2^k)/d$ satisfying the preceding relations. We need to take special care multiplying the 32-bit signed n with the 32-bit unsigned s. We achieve this using a `signed long long` type multiply with a correction if the top bit of s is set.

Example 5.15 The following routine, `sdiv_by_const`, shows how to divide by a signed constant d. In practice you will precalculate k and s at compile time. Only the operations involving n for your particular value of d need be executed at run time.

```
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;
}
```

■

Section 7.3 shows how to implement divides efficiently in assembler.

SUMMARY **Division**

- Avoid divisions as much as possible. Do not use them for circular buffer handling.
- If you can't avoid a division, then try to take advantage of the fact that divide routines often generate the quotient `n/d` and modulus `n%d` together.
- To repeatedly divide by the same denominator d, calculate $s = (2^k - 1)/d$ in advance. You can replace the divide of a k-bit unsigned integer by d with a $2k$-bit multiply by s.
- To divide unsigned $n < 2^N$ by an unsigned constant d, you can find a 32-bit unsigned s and shift k such that `n/d` is either $(ns) \gg (N + k)$ or $(ns + s) \gg (N + k)$. The choice depends only on d. There is a similar result for signed divisions.

5.11 Floating Point

The majority of ARM processor implementations do not provide hardware floating-point support, 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 and the Vector Floating Point accelerator (VFP) hardware, the C compiler must provide support for floating point in software.

In practice, this means that 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. Even so, 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. Fractional values are most often used when processing digital signals such as audio and video. This is a large and important area of programming, so we have dedicated a whole chapter, Chapter 8, to the area of digital signal processing on the ARM. For best performance you need to code the algorithms in assembly (see the examples of Chapter 8).

5.12 Inline Functions and Inline Assembly

Section 5.5 looked at how to call functions efficiently. You can remove the function call overhead completely by inlining functions. Additionally 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. For the examples of this section we will use the inline assembler in *armcc*.

Don't confuse the inline assembler with the main assembler *armasm* or *gas*. The inline assembler is part of the C compiler. The C compiler still performs register allocation, function entry, and exit. The compiler also attempts to optimize the inline assembly you write, or deoptimize it for debug mode. Although the compiler output will be functionally equivalent to your inline assembly, it may not be identical.

The main benefit of inline functions and inline assembly is to make accessible in C operations that are not usually available as part of the C language. 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.

Let's consider as an example the saturating multiply double accumulate primitive used by many speech processing algorithms. This operation calculates $a + 2xy$ for 16-bit signed operands x and y and 32-bit accumulator a. Additionally, all operations saturate to the nearest possible value if they exceed a 32-bit range. We say x and y are Q15 fixed-point integers because they represent the values $x2^{-15}$ and $y2^{-15}$, respectively. Similarly, a is a Q31 fixed-point integer because it represents the value $a2^{-31}$.

We can define this new operation using an inline function `qmac`:

```
__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 $a = 2x_0y_0 + \cdots 2x_{N-1}y_{N-1}$ 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;
}
```

The compiler replaces each qmac function call with inline code. In other words it inserts the code for qmac instead of calling qmac. Our C implementation of qmac isn't very efficient, requiring several if statements. We can write it much more efficiently using assembly. The inline assembler in the C compiler allows us to use assembly in our inline C function.

Example 5.16 This example shows an efficient implementation of qmac using inline assembly. The example supports both *armcc* and *gcc* inline assembly formats, which are quite different. In the *gcc* format the "cc" informs the compiler that the instruction reads or writes the condition code flags. See the *armcc* or *gcc* manuals for further information.

```
__inline int qmac(int a, int x, int y)
{
  int i;
  const int mask = 0x80000000;

  i = x*y;
#ifdef __ARMCC_VERSION      /* check for the armcc compiler */
  __asm
  {
    ADDS i, i, i            /* double */
    EORVS i, mask, i, ASR 31 /* saturate the double */
    ADDS a, a, i             /* accumulate */
    EORVS a, mask, a, ASR 31 /* saturate the accumulate */
  }
#endif
#ifdef __GNUC__ /* check for the gcc compiler */
  asm("ADDS %0,%1,%2        ":"=r" (i):"r" (i)   ,"r" (i):"cc");
  asm("EORVS %0,%1,%2,ASR#31":"=r" (i):"r" (mask),"r" (i):"cc");
  asm("ADDS %0,%1,%2        ":"=r" (a):"r" (a)   ,"r" (i):"cc");
  asm("EORVS %0,%1,%2,ASR#31":"=r" (a):"r" (mask),"r" (a):"cc");
#endif

  return a;
}
```

This inlined code reduces the main loop of `sat_correlate` from 19 instructions to 9 instructions. ■

EXAMPLE 5.17 Now suppose that we are using an ARM9E processor with the ARMv5E extensions. We can rewrite `qmac` again so that the compiler uses the new ARMv5E instructions:

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

  __asm
  {
    SMULBB i, x, y /* multiply */
    QDADD a, a, i /* double + saturate + accumulate + saturate */
  }
  return a;
}
```

This time the main loop compiles to just six instructions:

```
sat_correlate_v3
        STR     r14,[r13,#-4]!   ; stack lr
        MOV     r12,#0           ; a = 0
sat_v3_loop
        LDRSH   r3,[r0],#2       ; r3 = *(x++)
        LDRSH   r14,[r1],#2      ; r14 = *(y++)
        SUBS    r2,r2,#1         ; N-- and set flags
```

```
SMULBB   r3,r3,r14     ; r3 = r3 * r14
QDADD    r12,r12,r3    ; a = sat(a+sat(2*r3))
BNE      sat_v3_loop   ; if (N!=0) goto loop
MOV      r0,r12        ; r0 = a
LDR      pc,[r13],#4   ; return r0
```

Other instructions that are not usually available from C include coprocessor instructions. Example 5.18 shows how to access these.

EXAMPLE 5.18 This example writes to coprocessor 15 to flush the instruction cache. You can use similar code to access other coprocessor numbers.

```
void flush_Icache(void)
{
#ifdef __ARMCC_VERSION /* armcc */
  __asm {MCR p15, 0, 0, c7, c5, 0}
#endif
#ifdef __GNUC__ /* gcc */
  asm ( "MCR p15, 0, r0, c7, c5, 0" );
#endif
}
```

SUMMARY **Inline Functions and Assembly**

- 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.

5.13 Portability Issues

Here is a summary of the issues you may encounter when porting C code to the ARM.

- *The* `char` *type.* On the ARM, `char` is unsigned rather than signed as for many other processors. A common problem concerns loops that use a `char` loop counter `i` and the continuation condition `i` ≥ `0`, they become infinite loops. In this situation, *armcc* produces a warning of unsigned comparison with zero. You should either use a compiler option to make `char` signed or change loop counters to type `int`.

- *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. Note that expressions are promoted to an `int` type before evaluation. Therefore if `i = -0x1000`, the expression `i == 0xF000` is true on a 16-bit machine but false on a 32- bit machine.
- *Unaligned data pointers.* Some processors support the loading of `short` and `int` typed values from unaligned addresses. A C program may manipulate pointers directly so that they become unaligned, for example, by casting a `char *` to an `int *`. ARM architectures up to ARMv5TE do not support unaligned pointers. To detect them, run the program on an ARM with an alignment checking trap. For example, you can configure the ARM720T to data abort on an unaligned access.
- *Endian assumptions.* C code may make assumptions about the endianness of a memory system, for example, by casting a `char *` to an `int *`. If you configure the ARM for the same endianness the code is expecting, then there is no issue. Otherwise, you must remove endian-dependent code sequences and replace them by endian-independent ones. See Section 5.9 for more details.
- *Function prototyping.* The *armcc* compiler passes arguments narrow, that is, reduced to the range of the argument type. If functions are not prototyped correctly, then the function may return the wrong answer. Other compilers that pass arguments wide may give the correct answer even if the function prototype is incorrect. Always use ANSI prototypes.
- *Use of bit-fields.* The layout of bits within a bit-field is implementation and endian dependent. If C code assumes that bits are laid out in a certain order, then the code is not portable.
- *Use of enumerations.* Although `enum` is portable, different compilers allocate different numbers of bytes to an `enum`. The *gcc* compiler will always allocate four bytes to an `enum` type. The *armcc* compiler will only allocate one byte if the `enum` takes only eight-bit values. Therefore you can't cross-link code and libraries between different compilers if you use `enums` in an API structure.
- *Inline assembly.* Using inline assembly in C code reduces portability between architectures. You should separate any inline assembly into small inlined functions that can easily be replaced. It is also useful to supply reference, plain C implementations of these functions that can be used on other architectures, where this is possible.
- *The* `volatile` *keyword.* Use the `volatile` keyword on the type definitions of ARM memory-mapped peripheral locations. This keyword prevents the compiler from optimizing away the memory access. It also ensures that the compiler generates a data access of the correct type. For example, if you define a memory location as a `volatile short type`, then the compiler will access it using 16-bit load and store instructions `LDRSH` and `STRH`.

5.14 Summary

By writing C routines in a certain style, you can help the C compiler to generate faster ARM code. Performance-critical applications often contain a few routines that dominate the performance profile; concentrate on rewriting these routines using the guidelines of this chapter.

Here are the key performance points we covered:

- Use the `signed` and `unsigned int` types for local variables, function arguments, and return values. This avoids casts and uses the ARM's native 32-bit data processing instructions efficiently.
- The most efficient form of loop is a `do-while` loop that counts down to zero.
- Unroll important loops to reduce the loop overhead.
- Do not rely on the compiler to optimize away repeated memory accesses. Pointer aliasing often prevents this.
- Try to limit functions to four arguments. Functions are faster to call if their arguments are held in registers.
- Lay structures out in increasing order of element size, especially when compiling for Thumb.
- Don't use bit-fields. Use masks and logical operations instead.
- Avoid divisions. Use multiplications by reciprocals instead.
- Avoid unaligned data. Use the `char *` pointer type if the data could be unaligned.
- Use the inline assembler in the C compiler to access instructions or optimizations that the C compiler does not support.