Application Note 34

Writing Efficient C for ARM

Document number: ARM DAI 0034A

Issued: January 1998

ENGLAND

Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@arm.com

GERMANY

Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@arm.com

JAPAN

Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@arm.com

USA

ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

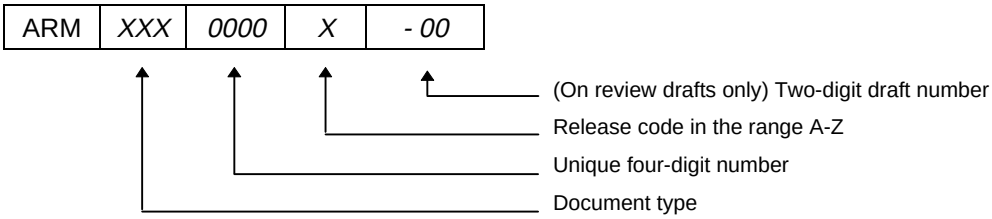
ARM
Advanced RISC Machines

Proprietary Notice

Key

Document Number

This document has a number which identifies it uniquely. The number is displayed on the front page and at the foot of each subsequent page.



Document Status

The document's status is displayed in a banner at the bottom of each page. This describes the document's confidentiality and its information status.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above and to the staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A	January 1998	SKW	Released

Table of Contents

1 Introduction

The ARM and Thumb C compilers are mature, industrial-strength ANSI C compilers which are capable of producing high quality machine code. However, when writing source code, it is always worthwhile to use programming techniques which work well on RISC processors such as ARM. This Application Note describes some of the techniques that can be useful. It also explains to some extent how the ARM compiler works, and how to use the C language efficiently. These techniques and knowledge will enable programmers to increase execution speed and/or lower code density.

Note *Most of the techniques discussed in this Application Note are equally applicable to both* `armcc` *and* `tcc`*. If a technique is only applicable to ARM or Thumb, this is highlighted. In principle, many of the described techniques are also applicable to other languages and other compilers.*

Note *The code examples given in this Application Note have been compiled and disassembled using tools supplied with ARM Software Development Toolkit version 2.11. If you are using another version of the toolkit, your output may differ slightly, though the principles highlighted should still hold, and should continue to hold in future releases.*

2 Setting Compiler Options

2.1 Selecting processor/architecture

You must select the correct processor, as this enables the compiler to make full use of instructions which are supported by the processor, and also perform processor-specific optimizations. If the processor is not directly supported, use the most appropriate base processor in a family. For example, use `-proc ARM7` for the ARM7 derivatives ARM710, ARM7100, and use `-proc StrongARM1` for the SA-1100. Note that when an ARM7 processor is selected you must specify whether the processor supports the Thumb instruction set (for example, `proc ARM7T`). This enables the compiler to use halfword instructions, making 16-bit accesses more efficient.

Alternatively, when a program is compiled to run on different ARM processors, choose the most appropriate architecture using `-arch N`. If possible, choose architecture 4 or 4T, as this enables the halfword instructions.

2.2 Debugging options

When setting the debugging options, it is important to know that enabling any of them will affect both codesize and performance *significantly*. The reason is that for debugging a varying level of optimizations is disabled. This is necessary because some optimizations produce code that cannot be described in debugging tables (so, for example, variables may be displayed incorrectly, or it becomes impossible to set breakpoints at certain places). Therefore, it is best to switch off all debugging when codesize and/or performance are important.

The compiler supports the following debugging options:

- `-g` high-level debugging, all optimizations disabled, simple register allocation
- `-gr` high-level debugging, some optimizations enabled, simple register allocation
- `-go` high-level debugging, most optimizations enabled, full register allocation

The following sections indicate which optimizations are disabled for debugging.

The option `-go` is the best option to use if debug information is really needed, as it enables the most important optimizations. It typically increases codesize by 7–15%, and decreases performance even more.

Debugging table size can be reduced by using ASD format in combination with `-gtp`. This provides the same level of debug information as Dwarf 1.0.

2.3 Optimization options

The compiler provides two optimization options:

- `-Otime` causes the compiler to optimize mainly for speed
- `-Ospace` causes the compiler to optimize mainly for codesize

Currently the difference between selecting `-Otime` and `-Ospace` is small. As this may change in later compiler releases, choose the most appropriate one. The most important differences are described in the rest of the document.

Note *It is not guaranteed that for small functions `-Otime` will always generate faster code, and `-Ospace` will always generate shorter code. A compiler cannot predict the exact outcome of a particular optimization until all optimizations have been applied, at which point it is too late to undo any optimizations.*

2.4 APCS options

The recommended APCS options are:

- `-apcs /nofp/noswst/softfp/narrow`
  for systems without hardware floating-point support
- `-apcs /nofp/noswst/hardfp/fpe3/fpr/narrow`
  for processors with a hardware floating-point coprocessor

The use of `/nofp/noswst` means that two extra registers are freed for allocation. Stack checking is not required, since the MMU can be programmed to do this. Alternatively, for systems without a MMU, stack requirements can often be calculated beforehand. Using a framepointer has no advantage (except when debugging is enabled: some debugging formats require a framepointer).

When `/hardfp` is used, the `/fpr` option passes the first four floating-point arguments in floating-point registers, which is faster than passing them in integer registers.

Using `/narrow` mainly results in float values being passed as floats, not as doubles. This is advantageous to both software and hardware floating-point. It also passes chars and shorts more efficiently. Note that using `/narrow` always results in higher performance, and in most cases codesize is decreased.

When compiling functions which do not need interworking, do not use `/interwork`, as interworking affects codesize and performance adversely.

Note *tcc version 1.05 as supplied with SDT2.11 does not support `/hardfp` or `/narrow`. A future version will support `/narrow` to speed up floating-point emulation.*

To show how the compiler options influence both codesize and performance, the Dhrystone program has been compiled with `armcc` and executed on a 40 MHz ARM7TDMI processor with a set of options. The default options used were:

```
-proc ARM7TDMI -apcs /nofp/noswst/softfp/narrow -bi.
```

Dhrystone options	Codesize in bytes	Dhrystones / second
`-Ospace`	40788	57971
`-Otime`	40816	61538
`-Otime -apcs /wide`	40820	60060
`-Otime -go`	40940	53476
`-Otime -gr`	41236	46404
`-Otime -g`	41308	46296
`-Otime -apcs /fp`	41392	59347
`-Otime -apcs /swst/fp`	42484	58651

The table shows that:

- `-Ospace` gives the shortest possible code, but at a high cost in execution time (about 6%)
- `/narrow` is faster than `/wide`
- `-go` is the best debugging option (15% slower)
- `/fp` and `/swst` result in the largest codesize.

3 Division and Remainder

The ARM instruction set does not provide integer division. Divisions are typically implemented by calling a C-library function (`__rt_sdiv` for signed and `__rt_udiv` for unsigned division). Depending on the numerator and denominator, a 32-bit division takes 20–140 cycles. The division function takes a constant time plus a time for each bit to divide:

Time(numerator / denominator)

= C0 + C1 * $\log_2$(numerator / denominator) =

= C0 + C1 * ($\log_2$(numerator) – $\log_2$(denominator)).

The current version takes about 20 + 4.3N cycles.

As division is an expensive operation, it is desirable to avoid it where possible. Sometimes expressions can be rewritten so that a division becomes a multiplication. For example, `(x / y) > z` can be rewritten as `x > (z * y)` if it is known that `y` is positive and `y * z` fits in an integer.

It is best to use unsigned division by ensuring that one of the operands is unsigned, as this is faster than signed division. This applies both to the division subroutines and to divisions by a power of two (see ***3.2 Division and remainder by powers of two***).

In the following sections several methods are discussed to improve division efficiency.

3.1 Combining division and remainder

In some cases both the dividend (x / y) and remainder (x % y) are needed. In such cases the compiler can combine both by calling the division function once (as it always returns both dividend and remainder). In order for this to work both expressions need to be close together. For example:

```
int combined_div_mod (int a, int b)
{  return (a / b) + (a % b);
}
```

This compiles into:

```
combined_div_mod
        STMDB    sp!,{lr}
        MOV      a3,a2
        MOV      a2,a1
        MOV      a1,a3
        BL       __rt_sdiv
        ADD      a1,a1,a2
        LDMIA    sp!,{pc}
```

3.2 Division and remainder by powers of two

If the divisor in a division operation is a power of two, the compiler uses a shift to perform the division. Therefore you should always arrange, where possible, for scaling factors to be powers of two (for example, 128 rather than 100).

This can be seen by examining the following piece of code:

```
typedef unsigned int uint;

uint div16u (uint a)
{  return a / 16;
}
int div16s (int a)
{  return a / 16;
}
```

The following code is produced:

```
div16u
        MOV     a1,a1,LSR #4
        MOV     pc,lr

div16s
        CMP     a1,#0
        ADDLT   a1,a1,#&f
        MOV     a1,a1,ASR #4
        MOV     pc,lr
```

Notice that while both divisions avoid calling the division function, unsigned division takes fewer instructions than signed. In many cases the `shift` instruction can be combined with following instructions. Signed division needs additional instructions because it rounds towards zero, while a shift rounds towards minus infinity.

3.3 Alternatives to remainder for modulo arithmetic

One reason for using the remainder operator is to provide modulo arithmetic. However, it is sometimes possible to rewrite the code to make use of `if` statement checks.

Consider the following two sample routines:

```
uint counter1 (uint count)
{  return (++count % 60);
}

uint counter2 (uint count)
{  if (++count >= 60)
       count = 0;
   return (count);
}
```

The following code is produced:

```
counter1
        STMDB   sp!,{lr}
        ADD     a2,a1,#1
        MOV     a1,#&3c
        BL      __rt_udiv
        MOV     a1,a2
        LDMIA   sp!,{pc}

counter2
        ADD     a1,a1,#1
        CMP     a1,#&3c
        MOVCS   a1,#0
        MOV     pc,lr
```

From this it is clear that the use of the `if` statement, rather than the remainder operator, is preferable, as it produces much faster code. Note that the new version only works if it is known that the range of count on input is 0–59.

3.4 Division by a constant

It is possible to write functions which implement divisions by a certain constant much faster than the general divide function. The ARM C-library contains two such functions, namely signed and unsigned division by 10 (to speed up decimal printing).

If division by other constants is commonly used in your particular application, it may be worthwhile writing specific routines to implement them.

Both ARM and Thumb versions of a division by a constant generator program are provided in `examples\explasm\div.c` and `examples\thumb\div.c` subdirectories of the toolkit.

3.5 Using lookup tables

See ***11 Using Lookup Tables*** on page 30 about lookup tables.

4 Conditional Execution

Note *This section is applicable to `armcc` only.*

Note *Conditional execution is disabled for all debugging options.*

All ARM instructions are conditional. Each instruction contains a 4-bit field which is a *condition code*; the instruction is only executed if the ARM flag bits indicate that the specified condition is true. Typically a conditionally executing code sequence starts with a compare instruction setting the flags, followed by a few conditionally executed instructions. For example:

```
CMP   x, #0
MOVGE y, #1
MOVLT y, #0
```

This saves two branch instructions and on average 2.5 ARM7 cycles.

Conditional execution reduces the number of branch instructions, and therefore improves codesize and performance. However, when more than about four instructions are conditional, performance could be worse in some cases (since branches take three cycles or less on ARMs). The compiler therefore limits the number of conditionally executed instructions. In SDT2.11 this limit is three instructions. In future compilers the limit will depend on whether `-Otime` or `-Ospace` is used.

Conditional execution is applied mostly in the body of `if` statements, but it is also used while evaluating complex expressions with relational (<, ==, > and so on) or boolean operators (&&, !, and so on). Conditional execution is disabled for code sequences which contain function calls, as on function return the flags are destroyed.

It is therefore beneficial to keep the bodies of `if` and `else` statements as simple as possible, so that they can be conditionalized. Relational expressions should be grouped into blocks of similar conditions.

The following example shows how the compiler uses conditional execution:

```
int g(int a, int b, int c, int d)
{  if (a > 0 && b > 0 && c < 0 && d < 0)  /* grouped conditions */
       return a + b + c + d;
   return -1;
}

g
        CMP     a1,#0
        CMPGT   a2,#0
        BLE     |L000024.J4.g|
        CMP     a3,#0
        CMPLT   a4,#0
        ADDLT   a1,a1,a2
        ADDLT   a1,a1,a3
        ADDLT   a1,a1,a4
        MOVLT   pc,lr
|L000024.J4.g|
        MVN     a1,#0
        MOV     pc,lr
```

Because the conditions were grouped, the compiler was able to conditionalize them.

5 Boolean Expressions

5.1 Range checking

A common boolean expression is used to check whether a variable lies within a certain range, for example to check whether a graphics co-ordinate lies within a window:

```
bool PointInRect1(Point p, Rectangle *r)
{  return (p.x >= r->xmin && p.x < r->xmax &&
           p.y >= r->ymin && p.y < r->ymax);
}
```

This compiles into:

```
PointInRect1
        LDR     a4,[a3,#0]
        CMP     a1,a4
        BLT     |L000034.J5.PointInRect1|
        LDR     a4,[a3,#4]
        CMP     a4,a1
        BLE     |L000034.J5.PointInRect1|
        LDR     a1,[a3,#8]
        CMP     a2,a1
        BLT     |L000034.J5.PointInRect1|
        LDR     a1,[a3,#&c]!
        CMP     a2,a1
        MOVLT   a1,#1
        MOVLT   pc,lr
|L000034.J5.PointInRect1|
        MOV     a1,#0
        MOV     pc,lr
```

There is a faster way to implement this: `(x >= min && x < max)` can be transformed into `(unsigned)(x-min) < (max-min)`. This is especially beneficial if `min` is zero. The same example after this optimization:

```
bool PointInRect2(Point p, Rectangle *r)
{  return ((unsigned) (p.x - r->xmin) < r->xmax &&
           (unsigned) (p.y - r->ymin) < r->ymax);
}

PointInRect2
        LDR     a4,[a3,#0]
        SUB     a1,a1,a4
        LDR     a4,[a3,#4]
        CMP     a1,a4
        LDRCC   a1,[a3,#8]
        SUBCC   a1,a2,a1
        LDRCC   a2,[a3,#&c]!
        CMPCC   a1,a2
        MOVCS   a1,#0
        MOVCC   a1,#1
        MOV     pc,lr
```

Future versions of the compiler will perform this optimization automatically.

5.2 Compares with zero

The ARM flags are set after a compare (CMP) instruction. The flags can also be set by other operations, such as MOV, ADD, AND, MUL, which are the basic arithmetic and logical instructions (the *dataprocessing* instructions). If a dataprocessing instruction sets the flags, the N and Z flags are set the same way as if the result was compared with zero. The N flag indicates whether the result is negative, the Z flag indicates that the result is zero. For example:

```
ADD R0, R0, R1
CMP R0, #0
```

This produces identical N and Z flags as:

```
ADDS R0, R0, R1
```

The N and Z flags on the ARM correspond to the signed relational operators `x < 0`, `x >= 0`, `x == 0`, `x != 0`, and unsigned `x == 0`, `x != 0` (or `x > 0`) in C.

Each time a relational operator is used in C, the compiler emits a compare instruction. If the operator is one of the above, the compiler can remove the compare if a data processing operation preceded the compare. For example:

```
int g(int x, int y)
{  if (x + y < 0)
       return 1;
   else
       return 0;
}

g
        ADDS     a1,a1,a2
        MOVPL    a1,#0
        MOVMI    a1,#1
        MOV      pc,lr
```

If possible, arrange for critical routines to test the above conditions (see ***6.1 Loop termination*** on page 12). This often allows you to save compares in critical loops, leading to reduced code size and increased performance.

The C language has no concept of a carry flag or overflow flag so it is not possible to test the C or V flag bits directly without using inline assembler. However, the compiler supports the carry flag (unsigned overflow). For example:

```
int sum(int x, int y)
{  int res;
   res = x + y;
   if ((unsigned) res < (unsigned) x)      /* carry set? */
       res++;
       return res;
}

sum
        ADDS     a2,a1,a2
        ADC      a2,a2,#0
        MOV      a1,a2
        MOV      pc,lr
```

6 Loops

Loops are a common construct in most programs; a significant amount of the execution time is often spent in loops. It is therefore worthwhile to pay attention to time-critical loops.

6.1 Loop termination

The loop termination condition can cause significant overhead if written without caution. You should always write count-down-to-zero loops and use simple termination conditions. Take the following two sample routines, which calculate n!. The first implementation uses an incrementing loop, the second a decrementing loop.

```
int fact1 (int n)
{  int i, fact = 1;
   for (i = 1; i <= n; i++)
      fact *= i;
   return (fact);
}

int fact2 (int n)
{  int i, fact = 1;
   for (i = n; i != 0; i--)
      fact *= i;
   return (fact);
}
```

The following code is produced:

```
fact1
        MOV     a3,#1
        MOV     a2,#1
        CMP     a1,#1
        BLT     |L000020.J5.fact1|
 |L000010.J4.fact1|
        MUL     a3,a2,a3
        ADD     a2,a2,#1
        CMP     a2,a1
        BLE     |L000010.J4.fact1|
 |L000020.J5.fact1|
        MOV     a1,a3
        MOV     pc,lr

 fact2
        MOVS    a2,a1
        MOV     a1,#1
        MOVEQ   pc,lr
 |L000034.J4.fact2|
        MUL     a1,a2,a1
        SUBS    a2,a2,#1
        BNE     |L000034.J4.fact2|
        MOV     pc,lr
```

You can see that the slight recoding of `fact1` required to produce `fact2` has caused the original ADD/CMP instruction pair to be replaced a single SUBS instruction. This is because a compare with zero could be optimized away, as described in ***5.2 Compares with zero*** on page 11.

In addition to saving an instruction in the loop, the variable n does not need to be saved across the loop, so a register is also saved. This eases register allocation, and leads to more efficient code elsewhere in the function (two more instructions saved).

This technique of initializing the loop counter to the number of iterations required, and then decrementing down to zero, also applies to `while` and `do` statements.

6.2 Loop unrolling

Small loops can be unrolled for higher performance, with the disadvantage of increased codesize. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears. The ARM compilers currently do not unroll loops automatically, so any unrolling should be done in the source code.

Population count—counting the number of bits set

This routine efficiently tests a single bit by extracting the lowest bit and counting it, after which the bit is shifted out. The second routine was first unrolled four times, after which an optimization could be applied by combining the four shifts of n into one. Unrolling frequently provides new opportunities for optimization.

```
int countbit1(uint n)
{  int bits = 0;
   while (n != 0)
   {
      if (n & 1) bits++;
      n >>= 1;
   }
   return bits;
}

int countbit2(uint n)
{  int bits = 0;
   while (n != 0)
   {
      if (n & 1) bits++;
      if (n & 2) bits++;
      if (n & 4) bits++;
      if (n & 8) bits++;
      n >>= 4;
   }
   return bits;
}
```

On the ARM7, checking a single bit takes six cycles when using the first version. The code size is only nine instructions. The unrolled version checks four bits at a time, taking on average only three cycles per bit. The cost is larger codesize: 15 instructions.

7 Switch Statement

A switch statement is translated by the ARM compiler as follows:

If the switch is *dense* the compiler uses a table lookup to jump to the code of the selected case label. A switch is dense if case labels comprise more than half the range spanned by the labels with the minimum and maximum values.

- For `armcc` the table is a branch-table with one word per entry, while `tcc` uses an offset table using only 8 or 16 bits per entry. `tcc` uses the 8-bit table when the number of case labels is less than 32. However, when the code in the switch statement is large, extra branches are needed to jump to the case labels.
- If the case labels are not dense, the compiler splits the case labels, and applies the same rules on each part recursively until all case labels have been processed.
- In order to improve the code size of switch statements, they should be as dense as possible, and for `tcc` both the code and the number of case labels should be kept small.

7.1 Switch statement vs. lookup tables

The switch statement is typically used for one of the following reasons:

- To call to one of several functions
- To set a variable or return a value
- To execute one of several fragments of code.

If the case labels are dense, in the first two uses of switch statements they could be implemented more efficiently using a lookup table. For example, two implementations of a routine that disassembles condition codes to strings:

```
char * ConditionStr1(int condition)
{
    switch(condition)
    {
           case 0: return "EQ";
           case 1: return "NE";
           case 2: return "CS";
           case 3: return "CC";
           case 4: return "MI";
           case 5: return "PL";
           case 6: return "VS";
           case 7: return "VC";
           case 8: return "HI";
           case 9: return "LS";
           case 10: return "GE";
           case 11: return "LT";
           case 12: return "GT";
           case 13: return "LE";
           case 14: return "";
           default: return 0;
    }
}
```

```
char * ConditionStr2(int condition)
{
   if ((unsigned) condition >= 15) return 0;
   return
      "EQ\0NE\0CS\0CC\0MI\0PL\0VS\0VC\0HI\0LS\0GE\0LT\0GT\0LE\0\0" +
      3 * condition;
}
```

The first routine needs a total of 240 bytes, the second only 72 bytes.

8 Register Allocation

Note *Register allocation is less efficient when the `-gr` or `-g` options are used. This is to ensure that variables are always displayed correctly in the debugger.*

The most important optimization supported by the ARM compilers is called *register allocation*. This is a process where the compiler allocates variables to ARM registers, rather than to memory. This has the advantage that those variables can be accessed quickly whenever needed, without needing instructions to transfer them from/to memory. As a result of register allocation, most variables are kept in registers, resulting in dramatic improvement in codesize and performance. You can write code which enables the compiler to achieve a more optimal register allocation.

8.1 Register allocatable variables

All basic integer, pointer and floating-point types can be allocated to registers. Fields of structures and complete structures can also be kept in registers. The following rules define when a variable is considered for allocation in a register:

A variable may be allocated to a register if:

- it is a local variable or a function parameter, *and*
- its address is never taken, *or* its address is taken, but not assigned to another variable.

A field in a structure may be allocated to a register if:

- it is declared locally or a function parameter, *and*
- the structure is not assigned directly with the result of a function call, *and*
- neither the address of the structure nor any of its fields is taken, or if any of these addresses is taken, it is not assigned to another variable.

8.2 Aliasing

Pointers are a powerful part of the C language. However, they must be used carefully or poor code may result. If the address of a variable is taken, the compiler must assume that the variable can be changed by any assignment through a pointer or by any function call, making it impossible to put it into a register. This is also true for global variables, as they might have their address taken in some other function. This problem is known as *pointer aliasing*, because the pointer is known as an alias of the variable it points to.

Note *Some C compilers offer an "ignore pointer aliasing" option, which tells the compiler to ignore the fact that other functions could be accessing local variables which have their address taken. This can cause problems if this is not the case, resulting in bugs which are difficult to trace. ARM does not offer this option because it contradicts with ANSI/ISO standard for C compilers.*

The negative effects which pointer aliasing has on performance can be reduced by using the following techniques:

- Avoid taking the address of local variables.
- Avoid global variables.
- Avoid pointer chains.

8.2.1 Local variables

It is often necessary to take the address of variables, for example if they are passed as a reference parameter to a function. This means that those variables cannot be allocated to registers. A solution is to make a copy of the variable, and pass the address of that copy.

In the following example, `test1` shows the conventional way of taking the address of the local variable, resulting in inefficient code. `test2` uses a dummy variable whose address is taken. The value is then copied to a local variable `i` (whose address is not taken). This allows the variable `i` to be allocated to a register, which reduces memory traffic.

```
void f(int *a);
int g(int a);

int test1(int i)
{  f(&i);
   /* now use 'i' extensively */
   i += g(i);
   i += g(i);
   return i;
}

int test2(int i)
{  int dummy = i;
   f(&dummy);
   i = dummy;
   /* now use 'i' extensively */
   i += g(i);
   i += g(i);
   return i;
}

test1
         STMDB    sp!,{a1,lr}
         MOV      a1,sp
         BL       f
         LDR      a1,[sp,#0]
         BL       g
         LDR      a2,[sp,#0]
         ADD      a1,a1,a2
         STR      a1,[sp,#0]
         BL       g
         LDR      a2,[sp,#0]
         ADD      a1,a1,a2
         ADD      sp,sp,#4
         LDMIA    sp!,{pc}

test2
         STMDB    sp!,{v1,lr}
         STR      a1,[sp,#-4]!
         MOV      a1,sp
         BL       f
         LDR      v1,[sp,#0]
         MOV      a1,v1
         BL       g
         ADD      v1,a1,v1
         MOV      a1,v1
         BL       g
         ADD      a1,a1,v1
         ADD      sp,sp,#4
         LDMIA    sp!,{v1,pc}
```

The first routine allocates `i` on the stack, and four memory accesses are needed for `i`. The second uses two memory accesses for `dummy`, and none for `i`.

Note *There are some exceptions where the compiler is able to determine that the address is not really used. For example:*

```
int f(int i)
{  return *(&i);
}
```

Here the compiler detects that the address is only taken inside the expression, and never assigned to another variable or passed to a function.

8.2.2 Global variables

Global variables are never allocated to registers (unless the `__global_reg` feature is used). Global variables can be changed by assigning them indirectly using a pointer, or by a function call. Hence the compiler cannot cache the value of a global variable in a register, resulting in extra (often unnecessary) loads and stores when globals are used. You should therefore not use global variables inside critical loops.

If a function uses global variables heavily, it is beneficial to copy those global variables into local variables so that they can be assigned to registers. This is possible only if those global variables are not used by any of the functions which are called.

For example:

```
int f(void);
int g(void);

int errs;

void test1(void)
{  errs += f();
   errs += g();
}

void test2(void)
{  int localerrs = errs;
   localerrs += f();
   localerrs += g();
   errs = localerrs;
}

test1
         STMDB    sp!,{v1,lr}
         BL       f
         LDR      v1,[pc, #L00002c-.-8]
         LDR      a2,[v1,#0]
         ADD      a1,a1,a2
         STR      a1,[v1,#0]
         BL       g
         LDR      a2,[v1,#0]
         ADD      a1,a1,a2
         STR      a1,[v1,#0]
         LDMIA    sp!,{v1,pc}
 L00002c
         DCD      |x$dataseg|
```

```
test2
        STMDB    sp!,{v1,v2,lr}
        LDR      v1,[pc, #L00002c-.-8]
        LDR      v2,[v1,#0]
        BL       f
        ADD      v2,a1,v2
        BL       g
        ADD      a1,a1,v2
        STR      a1,[v1,#0]
        LDMIA    sp!,{v1,v2,pc}
```

Note that `test1` must load and store the global `errs` value each time it is incremented, whereas `test2` stores `localerrs` in a register and needs only a single instruction.

8.2.3 Pointer chains

Pointer chains are frequently used to access information in structures. For example, a common code sequence is:

```
typedef struct { int x, y, z; } Point3;
typedef struct { Point3 *pos, *direction; } Object;

void InitPos1(Object *p)
{
   p->pos->x = 0;
   p->pos->y = 0;
   p->pos->z = 0;
}
```

However, this code must reload `p->pos` for each assignment, because the compiler does not know that `p->pos->x` is not an alias for `p->pos`. A better version would cache `p->pos` in a local variable:

```
void InitPos2(Object *p)
{  Point3 *pos = p->pos;
   pos->x = 0;
   pos->y = 0;
   pos->z = 0;
}
```

Another possibility is to include the `Point3` structure in the `Object` structure, thereby avoiding pointers completely.

8.3 Live variables and spilling

As the ARM has a fixed set of registers, there is a limit to the number of variables that can be kept in registers at any one point in the program. With the recommended options, there are 14 integer registers available. For hardware floating-point, eight separate floating-point registers are available. For software floating-point, the integer registers are used to hold floating-point variables.

The ARM compilers support *live-range splitting*, where a variable can be allocated to different registers as well as to memory in different parts of the function. The *live-range* of a variable is defined as all statements between the last assignment to the variable, and the last usage of the variable before the next assignment. In this range the value of the variable is valid, thus it is *alive*. In between live ranges, the value of a variable is not needed: it is dead, so its register can be used for other variables, allowing the compiler to allocate more variables to registers.

The number of registers needed for register-allocatable variables is at least the number of overlapping live-ranges at each point in a function. If this exceeds the number of registers available, some variables must be stored to memory temporarily. This process is called *spilling*. The compiler spills the least frequently used variables first, so as to minimize the cost of spilling. Spilling of variables can be avoided by:

- Limiting the maximum number of live variables. This is typically achieved by keeping expressions simple and small, and not using too many variables in a function. Subdividing large functions into smaller, simpler ones might also help.
- Using `register` for frequently-used variables. This tells the compiler that the register variable is going to be frequently used, so it should be allocated to a register with a very high priority. However, such a variable may still be spilled in some circumstances.

9 Variable Types

The C compilers support the basic types `char`, `short`, `int` and `long long` (signed and unsigned), `float` and `double`. Using the most appropriate type for variables is important, as it can reduce code and/or data size and increase performance considerably.

9.1 Local variables

Where possible, it is best to avoid using `char` and `short` as local variables. For the types `char` and `short` the compiler needs to reduce the size of the local variable to 8 or 16 bits after each assignment. This is called *sign-extending* for signed variables and *zero-extending* for unsigned variables. It is implemented by shifting the register left by 24 or 16 bits, followed by a signed or unsigned shift right by the same amount, taking two instructions (zero-extension of an unsigned char takes one instruction).

These shifts can be avoided by using `int` and `unsigned int` for local variables. This is particularly important for calculations which first load data into local variables and then process the data inside the local variables. Even if data is input and output as 8- or 16-bit quantities, it is worth considering processing them as 32-bit quantities.

Note *If you do this you must ensure that the results will be identical.*

Consider the following three functions:

```
int wordinc (int a)
{  return a + 1;
}

short shortinc (short a)
{  return a + 1;
}

char charinc (char a)
{  return a + 1;
}

wordinc
        ADD     a1,a1,#1
        MOV     pc,lr

shortinc
        ADD     a1,a1,#1
        MOV     a1,a1,LSL #16
        MOV     a1,a1,ASR #16
        MOV     pc,lr

charinc
        ADD     a1,a1,#1
        AND     a1,a1,#&ff
        MOV     pc,lr
```

9.2 Use of shorts/signed bytes on ARM

ARM processors implementing an ARM Architecture earlier than version 4 do not have the ability to load or store halfwords (shorts) or signed bytes (signed char) directly to or from memory. These operations are implemented using load/store byte operations and shifts, and may take up to four instructions. On Architecture 4 and later, these operations only take a single instruction.

Therefore, if possible, select Architecture 4 processors for applications which use shorts or signed chars heavily.

The following example illustrates the effect of using shorts and the `-arch 4` option.

```
VARTYPE array [2000];
void varsize (void)
{  int loop;
   for (loop = 0; loop < 2000; loop++)
          array[loop] = loop;
}
```

This gives the following results:

Compiler options	Code size	ZI data size
`-DVARTYPE=int`	32	8000
`-DVARTYPE=short -arch 3`	44	4000
`-DVARTYPE=short -arch 4`	36	4000

9.3 Space occupied by global data

If your code uses C global variables, you are advised to read *Application Note 36: Declaring C Global Data* (ARM DAI 0036) for details of how the compilers map such variables onto memory and how this can be controlled.

10 Function Design

In general, it is a good idea to keep functions small and simple. This enables the compiler to perform other optimizations, such as register allocation, more efficiently.

10.1 Function call overhead

Function call overhead on the ARM is small, and is often small in proportion to the work performed by the called function.

- The minimal call-return sequence is `BL ... MOV pc, lr`, which is extremely fast (four cycles on StrongARM1, six cycles on ARM7).
- The multiple load and store instructions, `LDM` and `STM` (`PUSH` and `POP` in the Thumb instruction set) reduce the cost of function entry and exit when some registers need to be saved.

Under the ARM Procedure Call Standard (APCS), up to four words of arguments can be passed to a function in registers. These arguments can be integer-compatible (char, shorts, ints and floats all take one word), or structures of up to four words (including the 2-word doubles and long longs).

If more arguments are needed, then the fifth and subsequent words are passed on the stack. This incurs the cost of storing these words in the calling function and reloading them in the called function.

Consider the following sample code:

```
int f1(int a, int b, int c, int d)
{  return a + b + c + d;
}

int g1(void)
{  return f1(1, 2, 3, 4);
}

int f2(int a, int b, int c, int d, int e, int f)
{  return a + b + c + d + e + f;
}

ing g2(void)
{  return f2(1, 2, 3, 4, 5, 6);
}
```

The fifth and sixth parameters are stored on the stack in g2, and reloaded in f2, costing two memory accesses per parameter.

10.1.1 Minimizing parameter passing overhead

To minimize the overhead of passing parameters to functions:

- Try to ensure that small functions take four or fewer arguments. These will not use the stack for argument passing.
- If a function needs more than four arguments, try to ensure that it does a significant amount of work, so that the cost of passing the stacked arguments is outweighed.
- Pass pointers to structures instead of passing the structure itself.
- Put related arguments in a structure, and pass a pointer to the structure to functions. This will reduce the number of parameters and increase readability.
- Minimize the number of long long parameters, as these take two argument words. This also applies to doubles if software floating-point is enabled.
- Avoid functions with a parameter that is passed partially in a register and partially on the stack (*split-argument*). This is not handled efficiently by the current compilers: all register arguments are pushed on the stack.
- Avoid functions with a variable number of parameters. Varargs functions effectively pass all their arguments on the stack.

10.1.2 Using __value_in_regs

A structure can be used to return multiple return values from a function. Normally, structures are returned on the stack. However, the compiler provides the `__value_in_regs` specifier, which can be used to return structures of up to four words in registers. This reduces memory traffic, enables the structure to be allocated to registers, and can give considerable reduction in code size.

For example, the ARM Software Floating-Point Library uses `__value_in_regs` to return structures containing intermediate results, which reduces code size and increases speed significantly. Long longs and doubles are returned by default using `__value_in_regs.`

Consider the following example:

```
typedef struct { int hi; uint lo; } int64; // low word unsigned!

__value_in_regs int64 add64(int64 x, int64 y)
{  int64 res;
   res.lo = x.lo + y.lo;
   res.hi = x.hi + y.hi;
   if (res.lo < y.lo) res.hi++;     // carry from low word
   return res;
}

void test(void)
{  int64 a, b, c, sum;
   a.hi = 0x00000000; a.lo = 0xF0000000;
   b.hi = 0x00000001; b.lo = 0x10000001;
   sum = add64(a, b);
   c.hi = 0x00000002; c.lo = 0xFFFFFFFF;
   sum = add64(sum, c);
}
```

```
add64
        ADDS    a2,a2,a4
        ADC     a1,a3,a1
        MOV     pc,lr

test
        STMDB   sp!,{lr}
        MOV     a1,#0
        MOV     a2,#&f0000000
        MOV     a3,#1
        MOV     a4,#&10000001
        BL      add64
        MOV     a3,#2
        MVN     a4,#0
        LDMIA   sp!,{lr}
        B       add64
```

By using `__value_in_regs`, the code size is 52 bytes (compared with 160 bytes otherwise). Note how the result from the first call to add64 is returned in r0 and r1, so these registers are already prepared for the second call to add64. The compiler can use the ADC instruction, resulting in optimal code.

`__value_in_regs` can also be an efficient way of interfacing C to assembler when you wish to return more than one result. You can define a structure to hold the return values and then write a C function declaration using `__value_in_regs`:

```
typedef struct
{  int x, y, z;
} Point3;

/* this is an ARM assembler function which takes a coord in r0-r3
 * and returns a transformed coordinate in r0-r3
 */
__value_in_regs Point3 TransformCoord(Point3 a);
```

10.2 Leaf functions

A function which does not call any other functions is known as a *leaf* function. In many applications, about half of all function calls made are to leaf functions. Leaf functions are compiled very efficiently on the ARM, as they often do not need to perform the usual saving and restoring of registers:

- With `armcc`, if the function needs up to five registers, it will use r0–r3 and r12, and thus does not need to save/restore registers.
- With `tcc`, there is no save/restore overhead if the function is simple enough to compile using just r0–r3.

When registers have to be saved, STM is the most efficient method. In a leaf function, all the relevant registers are saved by a single `STMFD sp!,{reg_list,lr}` on entry and a matching `LDMFD sp!,{reg_list,pc}` on exit (or the equivalent `PUSH {reg_list,lr}` and `POP {reg_list,pc}` instructions in Thumb code).

In general, the cost of pushing some registers on entry and popping them on exit is very small compared to the cost of the useful work done by a leaf function that is complicated enough to need more than four or five registers.

Overall, you should expect a leaf function to carry virtually no function entry / exit overhead, and at worst, a small overhead, most likely in proportion to the useful work done by the function.

If possible, you should try to arrange for frequently-called functions to be leaf functions. The number of times a function is called can be determined by using the profiling facility (refer to the *Software Development Toolkit User Guide* (ARM DUI 0040) for more details).

There are several ways to ensure that a function is compiled as a leaf function:

- Avoid calling other functions. This includes any operations which are converted to calls to the C-library (such as division, or any floating-point operation when the software floating-point library is used).
- Use `__inline` for small functions which are called from it (see ***10.5 Inline functions*** on page 28 for more details).

10.3 Tail continued functions

Note *This section applies only to `armcc`.*

Note *Tail continuation is disabled for all debugging options.*

If a function ends with a call to another function, the call can be converted to a branch to that function. This is called *tail continuation*. It usually saves stackspace and a branch. This optimization can be applied to any function that calls another just before returning, if none of the parameters is the address of a local variable.

Consider, for example:

```
extern int func2(int);

int func1 (int a, int b)
{   if (a > b)
        return (func2(a - b));
    else
        return (func2(b - a));
}
```

The following code is produced:

```
func1
        CMP     a1,a2
        SUBLE   a1,a2,a1
        SUBGT   a1,a1,a2
        B       func2
```

Here there is no function entry or exit overhead, and the function return has disappeared entirely: `func2` returns directly to `func1`'s caller. In this case, the basic call-return cost for the function pair has been reduced from:

```
BL + BL + MOV pc,lr + MOV pc,lr
```

to:

```
BL + B  +            MOV pc,lr
```

which works out as a saving of 25%. Many of the other examples given in this Application Note also benefit from this optimization.

10.4 Pure functions

Note *The pure function optimization is disabled for the -g option.*

Pure functions are those which return a result which depends only on their arguments. They can be thought of as mathematical functions: they always return the same result if the arguments are the same. Therefore they must not have any side-effects, where a different value could be returned, even though the parameters are the same. For example, a pure function may use the stack for local storage, and read its parameters from the stack. However, a pure function cannot read or write global state by using global variables or indirecting through pointers. To tell the compiler that a function is pure, use the special declaration keyword `__pure`.

Consider the following sample code:

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

int f(int n)
{  return square(n) + square(n);
}

square
        MOV     a2,a1
        MUL     a1,a2,a2
        MOV     pc,lr

f
        STMDB   sp!,{lr}
        MOV     a3,a1
        BL      square
        MOV     a4,a1
        MOV     a1,a3
        BL      square
        ADD     a1,a4,a1
        LDMIA   sp!,{pc}
```

If the code is modified so that `square` is defined as pure:

```
__pure int square(int x)
{  return x * x;
}

f
         STMDB   sp!,{lr}
         BL      square
         MOV     a1,a1,LSL #1
         LDMIA   sp!,{pc}
```

Note that `square` is now only called once. This is because the compiler has detected that it is a *common subexpression* (CSE). This optimization can only be performed on pure functions as they do not have side-effects.

Pure functions can also improve the code in other ways, as they cannot have read or written any memory locations. This means, for example, that values which are allocated to memory can be safely cached in registers, instead of being written to memory before a call and reloaded afterwards.

Another way to tell the compiler that a function is pure is to place the following pragmas around its definition:

```
#pragma no_side_effects
   /* function definition */
#pragma side_effects
```

10.5 Inline functions

Note *Function inlining is disabled for all debugging options.*

The ARM compilers support inline functions with the keyword `__inline`. This results in each call to an inline function being substituted by its body, instead of a normal call. This results in faster code, but it adversely affects code size, particularly if the inline function is large and used often.

```
__inline int square(int x)
{  return x * x;
}

#include <math.h>

double length(int x, int y)
{  return sqrt(square(x) + square(y));
}

length
        STMDB    sp!,{lr}
        MUL      a3,a1,a1
        MLA      a1,a2,a2,a3
        BL       _dflt
        LDMIA    sp!,{lr}
        B        sqrt
```

There are several advantages to using inline functions:

- No function call overhead.
  As the code is substituted directly, there is no overhead, like saving and restoring registers.
- Lower argument evaluation overhead.
  The overhead of parameter passing is generally lower, since it is not necessary to copy variables. If some of the parameters are constants, the compiler can optimize the resulting code even further.
- More optimizations possible.
  The compiler was able to combine an ADD and MUL instruction into a multiply-accumulate (MLA) instruction.

The big disadvantage of inline functions is that the code sizes increases if the function is used in many places. This can vary significantly depending on the size of the function, and the number of places where it is used. It is for this reason that `armcc` and `tcc` do not offer a compiler option to inline functions automatically.

In general, it is wise to only inline a few critical functions. Note that when done wisely, inlining may *decrease* the size of the code: A call takes usually a few instructions, but the optimized version of the inlined code might translate to even less instructions. The above examples show this: a call to square takes between one and three instructions, while an inlined square always takes a single instruction.

10.6 Function definitions

Note *This section applies only to `armcc`, as `tcc` does not currently support this optimization.*

Placing function definitions before their use can sometimes produce better code as it allows the compiler to see the register usage of the called function. This is a simple form of *interprocedural optimization* (where optimizations are carried out between functions).

```
int square(int x);

int sumsquares1(int x, int y)
{  return square(x) + square(y);
}

/* square might be defined here, or in another source file */
int square(int x)
{  return x * x;
}

int sumsquares2(int x, int y)
{  return square(x) + square(y);
}
```

```
sumsquares1
        STMDB    sp!,{v1,v2,lr}
        MOV      v1,a2
        BL       square
        MOV      v2,a1
        MOV      a1,v1
        BL       square
        ADD      a1,v2,a1
        LDMIA    sp!,{v1,v2,pc}

square
        MOV      a2,a1
        MUL      a1,a2,a2
        MOV      pc,lr

sumsquares2
        STMDB    sp!,{lr}
        MOV      a3,a2
        BL       square
        MOV      a4,a1
        MOV      a1,a3
        BL       square
        ADD      a1,a4,a1
        LDMIA    sp!,{pc}
```

By putting the `square` definition before the `sumsquares` definition, the compiler knows that a3 and a4 are not used. It is able to use these registers, knowing that they will not be corrupted by the `square` function, instead of being forced to use v1 and v2 by storing their values on the stack. Fewer memory accesses result in higher speed.

11 Using Lookup Tables

A function can often be approximated using a lookup table, which increases performance significantly. A table lookup is usually less accurate than calculating the value properly, but for many applications this does not matter.

Many signal processing applications (for example, modem demodulator software) make heavy use of sin and cos functions, which are computationally expensive to calculate. For real-time systems where accuracy is not very important, sin/cos lookup tables might be essential.

When using lookup tables, try to combine as many adjacent operations as possible into a single lookup table. This is faster and uses less space than multiple lookup tables.

Example code which demonstrates the use of lookup tables can be found on the ARM Web site.

12 Floating-Point Arithmetic

The ARM core does not contain any actual floating-point hardware. Instead there are three options for an application which needs floating-point support.

- Floating-Point Accelerator (FPA) hardware coprocessor.
  This implements a floating-point instruction set using a number of ARM coprocessor instructions. However this does require the FPA hardware to exist within the system as a coprocessor.
- The Floating-Point Emulator (FPE).
  This emulates in software the instructions that the FPA executes. This means that there is no need to recompile code for systems with or without the FPA.
- The Floating-Point Library (FPLib).
  Floating-point operations are compiled into function calls to library routines rather than floating-point instructions. Although this is slower than using a FPA it is typically two or three times faster than using the FPE. (The FPE emulates the FPA instruction set, which means there is some overhead per instruction.) The overall code size of the system is also smaller because only the required library routines are included, rather than the whole of the FPE. The floating-point library is therefore the route that ARM recommends for use in embedded systems and is the default.

Note *The Thumb instruction set does not have instruction space for coprocessor instructions. Thus the Thumb compiler will always provide floating-point functionality using the Floating-Point Library.*

The recommended compiler options give the best results in terms of performance and code size. However, when writing floating-point code, keep the following things in mind:

- Floating-point division is slow.
  Division is typically twice as slow as addition or multiplication. Rewrite divisions by a constant into a multiplication with the inverse (for example, `x = x / 3.0` becomes `x = x * (1.0/3.0)`—the constant is calculated during compilation).
- Use floats instead of doubles.
  Float variables consume less memory and fewer registers, and are more efficient because of their lower precision. Use floats whenever their precision is good enough.
- Avoid using transcendental functions.
  Transcendental functions, like sin, exp and log are implemented using series of multiplications and additions (using extended precision). As a result, these operations are at least ten times slower than a normal multiply.
- Simplify floating-point expressions.
  The compiler cannot apply many optimizations which are performed on integers to floating-point values. For example, `3 * (x / 3)` cannot be optimized to `x`, since floating-point operations generally lead to loss of precision. Even the order of evaluation is important: `(a + b) + c` is not the same as `a + (b + c)`. Therefore, it is beneficial to perform floating-point optimizations manually if it is known they are correct.

For more information about optimizing floating-point performance, see *Application Note 55: Floating-Point Performance* (ARM DAI 0055).

However it is still possible that the floating performance will not reach the required level for a particular application. In such a case the best approach may be to change from using floating-point to fixed point arithmetic. When the range of values needed is sufficiently small, fixed-point arithmetic is more accurate and much faster than floating-point arithmetic. See *Application Note 33: Fixed Point Arithmetic* (ARM DAI 0033) for more details.

13 Cross Jump Optimization

Note *Cross jump optimization is disabled for all debugging options and for* `-Otime`.

One of the optimizations carried out by the ARM compilers is known as *cross jump optimization*. This is a space-saving strategy whereby identical sequences of code are identified. It is mainly used in if- and switch-statements. However, in a switch-statement this can lead to extra branches being executed which may decrease performance in some cases. Therefore cross jumping is switched off when optimizing for speed (`-Otime`). Cross jump optimization can be switched on or off by using `#pragma -j1 / -j0`, or at the command line by using `-zpj1` / `-zpj0`.

Consider the following section of C code:

```
extern void g(int, int);

int f(int a, int b)
{
   if (a > b)
   {
      a += b;
      a--;
      g(a, b);
   }
   else
   {
      a += b;
      b--;
      g(a, b)
   }
   return a + b;
}

f
        STMDB    sp!,{v1,v2,lr}
        MOV      v2,a2
        CMP      a1,a2
        ADD      v1,a1,a2
        SUBLE    v2,v2,#1
        SUBGT    v1,v1,#1
        MOV      a2,v2
        MOV      a1,v1
        BL       g
        ADD      a1,v1,v2
        LDMIA    sp!,{v1,v2,pc}
```

The final three instructions in the then and else part have been combined, resulting in shorter and faster code.

14 Portability of C Code

Many of the C optimizations described here apply to other processors. However, some of the optimizations rely on pragma statements or special function declaration keywords (such as `__inline`). Many other compilers also support these, although they have a different syntax.

To maintain code portability to other platforms, these ARM-specific keywords could be made into macros:

```
#ifdef __arm
#  define INLINE        __inline
#  define VALUE_IN_REGS __value_in_regs
#  define PURE          __pure
#else
#  define INLINE
#  define VALUE_IN_REGS
#  define PURE
#endif
```

The code can make use of `INLINE`, `VALUE_IN_REGS`, and so on.

```
INLINE int square(int x) {
  return x*x;
}
```

Note `__arm` *is defined by both* `armcc` *and* `tcc`
`__thumb` *is defined by* `tcc`.

15 Further Information

For further information, refer to:

- *Software Development Toolkit User Guide* (ARM DUI 0040): ***Chapter 7 Benchmarking, Performance Analysis and Profiling***
- *Application Note 33: Fixed Point Arithmetic on the ARM* (ARM DAI 0033)
- *Application Note 36: Declaring C Global Data* (ARM DAI 0036)
- *Application Note 55: Floating-Point Performance* (ARM DAI 0055)