# Coding for the Cortex™-R4(F)

Document number: ARM DAI 0240A



ARM®

## Coding for the Cortex™-R4(F)



### Release information

The following table lists the changes made to this document.

**Change history**

| Date | Issue | Change |
|---|---|---|
| 15 October 2010 | A | First release |

### Proprietary notice



### Confidentiality status

This document is Non-Confidential. This document has no restriction on distribution.

### Feedback on this application note

If you have any comments on content then send an e-mail to errata@arm.com. Give:

- the document title
- the document number
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

### ARM web address

http://www.arm.com

# Table of Contents

# 1 Introduction

ARM Cortex-R real-time processors offer high-performance computing solutions for deeply embedded systems with demanding real-time response constraints. This document introduces the main features of the Cortex-R4 and Cortex-R4F processors. It also discusses C coding considerations when porting code targeted for an ARM946E-S™ to the Cortex-R4(F).

## 1.1 Scope

This document assumes a familiarity with the ARM946E-S or ARM1156T2(F)-S™ processors, and the ARMv5TE architecture. It also assumes familiarity with C and assembler programming. It discusses issues related to porting code from an ARM946E-S to the Cortex-R4(F). It does not discuss general programming issues that apply equally to both processors.

## 1.2 Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, http://infocenter.arm.com., for access to ARM documentation.

### 1.2.1 ARM publications

The following documents contain information relevant to this document:

- *Cortex-R4 and Cortex-R4F Technical Reference Manual* (ARM DDI 0363)
- *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* (ARM DDI 0406)
- ARM Architecture Reference Manual Performance Monitors v2 Supplement (ARM DDI 0457)
- RealView® Compilation Tools Compiler Reference Guide Version 4.0 (ARM DUI 0348B)
- RealView Compilation Tools Compiler User Guide Version 4.0 (ARM DUI 0205I)

### 1.2.2 Other publications

This section lists relevant third-party websites and documents:

- *ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*
- *http://*www.xvid.org

# 2 Cortex-R4(F) Overview

This chapter gives an overview of the main features of the Cortex-R4 and Cortex-R4F processors. Later chapters will discuss how C compilers may make use of these features.

## 2.1 ARMv7-R

The Cortex-R4(F) implements the ARMv7-R architecture, which provides a number of extensions and improvements over ARMv5TE. Many of the issues to be considered when porting code to the Cortex-R4(F) are architectural differences/enhancements, so would also apply to other Cortex-R processors.

### 2.1.1 Thumb-2

The Thumb-2 ISA is a major extension to the Thumb ISA. It is a mixed 16-bit and 32-bit instruction set, which aims to provide the code density advantages of Thumb with the flexibility and performance of the ARM instruction set.

Thumb-2 extends Thumb to enable access to system registers (e.g. the CPSR) and coprocessors. The architecture also enables the processor to be configured to take exceptions in Thumb state.

### 2.1.2 Instruction set changes

The ARMv6 and ARMv7-R architectures added a number of additional instructions. These include instructions for mode changing, handling mixed endian data, and SIMD operations.

ARMv7-R added support for hardware divide, the UDIV and SDIV instructions. These are only available in the Thumb-2 instruction set.

The behavior of data processing instructions that write the PC has changed. Previously data processing instructions could not cause a state change, except when returning from an exception. In ARMv7-R any data processing instruction that writes to the PC can cause a state change, based on bit 0 of the address.

### 2.1.3 Mixed Endian Support

ARMv6 improved support for mixed endian systems. The Cortex-R4(F) processor supports Little Endian (LE) and Byte Invariant Big Endian (BE-8) for data accesses. Data endianness is controlled by the CPSR.E and SCTLR.EE bits. The CPSR.E bit can easily be controlled using the SETEND instruction.

Also provided are instructions for converting the format of data held in registers. These include the REV and REV16 instructions.

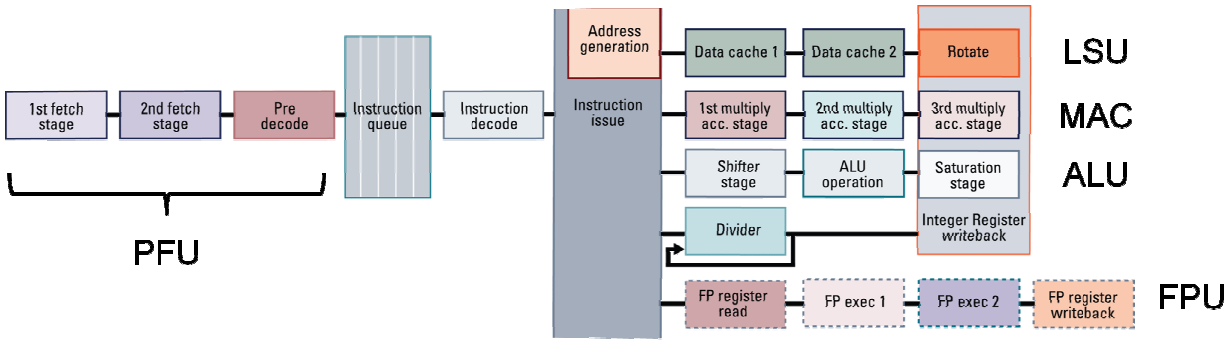
### 2.1.4 ARMv7-R PMSA

The Cortex-R4(F) processor implements the ARMv7-R Protected Memory System Architecture. This provides a weakly ordered memory model, allowing greater optimization of memory accesses by the processor.

Note: Memory management is outside of the scope of this document. However, memory management can have a very significant impact on overall performance.

## 2.2 Pipeline

The Cortex-R4(F) processor has an in order, dual-issue, eight stage pipeline:




The Pre-Fetch Unit (PFU) fetches, decodes and queues instructions ready to issue, while the latter stages execute instructions in one of four execution paths for load-store (LSU), multiply-accumulate (MAC), arithmetic logic unit (ALU) and divide, and also in the floating point unit (FPU) if it is present.

### 2.2.1 Pre-fetching and Dual issuing

The PFU has 64 bit interfaces to the bus, TCMs and instruction cache. This allows up to two ARM instructions, or four Thumb instructions, to be fetched per cycle. The pipeline provides buffering between the pre-decode and decode stages, and in the issue stage. This allows variations in the instruction flow (for example due to an instruction cache miss) to be smoothed out.

The pipeline is also capable of issuing two instructions in a single cycle. The instructions must match a number of criteria, such as using different backend pipes and not having a data dependency.

### 2.2.2 Branch prediction

Longer pipelines are common on modern high performance processors. Longer pipelines assist in reducing the length of critical paths within the design, enabling the processor to run at higher frequencies.

A side effect of longer pipelines is that a pipeline flush becomes more costly. Previous processors, such as the ARM946E-S, would flush the pipeline on every branch. If this were the case on the Cortex-R4(F) processor branches would be very costly in terms of cycles.

To avoid unnecessary pipeline flushes the processor attempts to recognize branches early, and begins to fetch from the destination of the branch.

Many branch instructions are conditional. For conditional branches whether the branch will be taken cannot be determined until the instruction is executed. The processor makes a prediction about whether the branch will be taken and fetches based on the prediction. The processor must also be able to detect when it gets the prediction wrong, and re-fetch from the correct location.

It is only when the pipeline is unable to predict, or mis-predicts, a branch that the pipeline is flushed.

Note: It is occasionally desirable to force a pipeline flush. As branches are no longer guaranteed to cause a pipeline flush, the ARMv7-R architecture provides the ISB (Instruction Synchronization Barrier) instruction.

Branches can be split into two categories, direct and indirect. Direct branches are PC relative, and branch to an offset from the current address. The range of a direct branch is limited (for example, +/-32MB for an ARM B or BL instruction). Examples of direct branches are:

```
B    <label>
BL   <label>
BLX  <label>
TBB  [Rn, Rm]
TBH  [Rn, Rm]
```

Indirect branches perform an absolute branch, so can branch to any location in the address space. However, the destination cannot be easily predicted by the processor. Examples of indirect branches:

```
BX   <Rd>
LDR  pc, [Rd]
ADD  pc, Rn, Rm
```

The Cortex-R4(F) processor supports two forms of branch prediction: Dynamic Branch Prediction and Return Stack Prediction. The Dynamic Branch Predictor is used to predict direct branches. Typically, loops and function calls in C code.

The Return Stack is used to predict a limited number of indirect branches. The Return Stack consists of a four entry LIFO (Last In First Out) buffer. When a function call instruction is executed, the generated LR is pushed onto the LIFO. When a function return is detected, the pipeline predicts that the destination is the top entry of the LIFO.

Recognized function calls:

```
BL   immediate
BLX  immediate
BLX  Rm.
```

Recognized function returns:

```
POP    {..,pc}
LDMIB  Rn{!}, {..,pc}
LDMDA  Rn{!}, {..,pc}
LDMDB  Rn{!}, {..,pc}
LDR    pc, [sp], #4
BX     Rm
```

These are instructions that are usually associated with function returns.

Not all branches are predictable. For example, the return stack does not recognize "`MOV pc, lr`" which may be used in some legacy code.

Note: Unlike some other Cortex processors, the Cortex-R4(F) has branch prediction enabled at Reset.

## 2.3 Level 1 Memory System

The Cortex-R4(F) processor supports Harvard level 1 instruction and data caches. Unlike the ARM1156T2(F)-S and ARM946E-S, these caches support both read and write allocation.

The Cortex-R4(F) processor provides two Tightly Coupled Memory (TCM) interfaces: ATCM and BTCM. Both the ATCM and BTCM can be used to store code and data. The BTCM can be implemented with dual ports, allowing two accesses in a single cycle (for example an instruction fetch and a data read).

Note: The ARM946E-S did not support instructions fetches from the DTCM (data TCM). This restriction is no longer present, and the naming of the TCMs was changed from I and D to A and B to reflect this.

The Cortex-R4(F) processor has a single 64-bit AXI master interface. An AXI slave interface is also provided, which gives external masters access to the TCMs and caches. This can be used to pre-populate the TCMs.

Whether the L1 caches and TCMs are present, and their sizes, is an implementation decision. Please refer to the datasheet for the specific part being used.

## 2.4 Floating point unit (FPU)

The Cortex-R4F includes an optional FPU, which implements the ARM VFP v3-D16 architecture and Common VFP Sub-Architecture v2. It provides floating-point computation functionality that is compliant with the ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, referred to as the IEEE 754 standard. The FPU supports all data-processing instructions and data types in the VFPv3 architecture as described in the *ARM Architecture Reference Manual*.

The FPU provides 16 double precision (64-bit) registers. It fully supports single-precision and double-precision operations. However, it is optimized for single-precision operation.

Unlike the ARM1156T2F-S and VFP9, no support code is needed by the Cortex-R4F for IEEE754 compliance.

The vector operations supported in the VFPv2 (ARM1156T2F-S and VFP9) are not supported, and attempting to use them on the Cortex-R4F will result in an UNDEFINED exception.

## 2.5 Error Detection/Correction

The Cortex-R4(F) processor includes a number of optional features related to error detection and correction. These include parity checking and ECC support for the caches and TCM interfaces.

# 3 Measuring performance with the PMU

The Cortex-R4(F) includes a PMU (Performance Monitor Unit). The PMU is a powerful feature for measuring and analyzing the performance of the processor.

## 3.1 PMU overview

The Cortex-R4(F)'s PMU provides four counters: a dedicated cycle counter (CCNT) and three configurable counters (PMN0, PMN1 and PMN2).

CCNT counts processor clock cycles. It can be configured to counter every clock cycle, or every 64$^{th}$ cycle.

The configurable counters count events, with each counter configured separately. The events that can be countered are given in the *Cortex-R4 and Cortex-R4F Technical Reference Manual*, and include events such as branch mis-predictions.

Each of the counters is 32-bits wide. Overflows can be checked for in software, and the PMU can be configured to signal an interrupt on overflow. This signal is an output from the processor, and the system designers must connect it to the interrupt controller to use it.

The PMU is accessed through a number of CP15 registers (c9). At reset these registers are only accessible in privileged modes, but User mode access can be enabled. This allows the system to choose whether to make the PMU accessible to unprivileged tasks.

## 3.2 PMU limitations

The PMU is ideal for analyzing code at a block or function level. Although, some thought must be given to the size of block being monitored.

With a very large block some events could overflow the 32-bit counters (e.g. number of instructions executed or cycles). Ideally this would be avoided, but the overflows should still be checked. For a loop events could be counted per iteration and then averaged. Beware that this additional code could alter the timing of the loop (e.g. due to changes in alignment).

With small blocks the counters may not be truly representative.

You must also carefully consider what the event definitions are – and what they tell you about the processor's execution. For example, event 0x10 is defined as:

> "Branch mispredicted or not predicted. This counts for every pipeline flush as a result of misprediction from the program flow prediction resources that might have predicted correctly within the core."

Forms of branch not supported by the branch predictor will not therefore be recorded. As will be shown later (section 4.4.1) a compiler can use "`ADD pc,pc,Rd,LSL #<imm>`" or "`TBB [pc,Rd]`" to implement a switch statement. Of these instructions the `TBB` is recognized by the branch predictor, but the `ADD` is not. Therefore code using `ADD` to perform a branch could show fewer mispredictions, while actually performing worse.

It is often necessary to look at a number of events in combination to get a full view of the pipeline's behavior. For example, in order to estimate cache efficiency the number of cache misses and the total number of accesses is required. For the data cache this would involve recording events 0x03 and 0x04.

## 3.3 Example code

ARM provides example PMU code for the Cortex-R and Cortex-A family of processors. This example code provides assembler functions, with C headers, to access the PMU registers. The example code helps you to get started using the PMU:

```
enable_pmu();              // Enable the PMU

reset_ccnt();              // Reset the CCNT (cycle counter)
reset_pmn();               // Reset the configurable counters

pmn_config(0, 0x03);       // Set counter 0 to count event 0x03

enable_ccnt();             // Enable (start) CCNT
enable_pmn(0);             // Enable (start) counter 0

//
// Code being analyzed
//

disable_ccnt();            // Stop CCNT
disable_pmn(0);            // Stop counter 0

counter0    = read_pmn(0); // Read counter 0
cycle_count = read_ccnt(); // Read CCNT
```

The example code is available from:

http://infocenter.arm.com/

From this location, navigate to:

→ ARM Technical Support Knowledge Articles

→ Processor Cores

→ Cortex processors

→ Cortex-R4

→ How do I use the Performance Monitoring features of my ARM11 / Cortex core to benchmark my code?

# 4 C Coding Considerations

## 4.1 Tools options

Part of generating efficient code is using the appropriate compiler, assembler and linker options when building the code. This document concentrates on the ARM Compiler toolchain, but the principles also apply to tools from other vendors.

The ARM Compiler toolchain will use default values for options which are not specified manually:

```
armcc –c –-debug my_file.c
```

This would compile “my_file.c” as:

- ARMv4T architecture
- ARM code
- No VFP (library code used for floating point operations)
- ARM9 pipeline optimizations
- With debug information (--debug)

While ARMv4T code is compatible with the Cortex-R4(F), it is not optimal. To make use of the additional features available in ARMv7-R the appropriate compiler options must be set:

```
armcc –c --debug --cpu=Cortex-R4F --thumb my_file.c
```

The file is now compiled as:

- ARMv7-R architecture
- Thumb-2 code
- VFPv3-D16 for floating point operations
- Cortex-R4F pipeline optimizations
- With debug information (--debug)

Note: --cpu=Cortex-R4 should be used if the VFP option is not implemented.

Setting the correct build options can have a significant effect on performance. For example, the table below shows the performance figures for the open source xvid codec that has been compiled using different compiler options. The figures are based on decoding 10 frames.

| | Code Size (Bytes) | Average cycles for Decode |
|---|---|---|
| `--cpu=4T` | 282,004 | 23,727,997.50 |
| `--cpu=ARM946E-S` | 278,180 | 23,604,788.40 |
| `--cpu=Cortex-R4` | 274,468 | 21,946,961.10 |

Using RVCT 4.0 build 870, with –O2 optimization level. All three builds were run on a Cortex-R4F with 64KB caches (caches enabled). The memory system was clocked at 1/5 of the core clock frequency, with zero wait state memory.

Note: XVID is very data intensive and data accesses dominate the performance figures.

## 4.2 Assembler vs C

The ARM and Thumb instructions sets include a number of specialist instructions that do not map easily to C operations. These can be system control instructions, such as enabling or disabling interrupts, or specialist data processing instructions.

An example of a specialist data processing instruction would be the REV instruction. This instruction reverses the order of bytes in a register, changing the endian format. Since the concept of endianness is not understood by C directly, it is difficult to express in C that this is what the programmer wishes to do. The compiler attempts to recognize common code fragments which implement such behavior, and use the specialist instructions. This is known as idiom recognition. For example:

| C | --cpu=v5TE | --cpu=Cortex-R4F |
|---|---|---|
| `uint32_t rev(uint32_t a)`<br>`{`<br>`return`<br>`((a<<24) & 0xFF000000U) \|`<br>`((a<< 8) & 0x00FF0000U) \|`<br>`((a>> 8) & 0x0000FF00U) \|`<br>`((a>>24) & 0x000000FFU);`<br>`}` | `rev PROC`<br>`MOV r1,#0xff0000`<br>`AND r1,r1,r0,LSL #8`<br>`MOV r2,#0xff00`<br>`ORR r1,r1,r0,LSL #24`<br>`AND r2,r2,r0,LSR #8`<br>`ORR r1,r1,r2`<br>`ORR r0,r1,r0,LSR #24`<br>`BX lr`<br>`ENDP` | `rev PROC`<br>`REV r0,r0`<br>`BX lr`<br>`ENDP` |

Using RVCT 4.0 build 870, with –O2 optimization level

However, the compiler may not recognize all cases where a specialist instruction may be used. This is particularly the case for system control operations.

In these cases the code could be re-written in assembler or using compiler intrinsics. An intrinsic tells the compiler to use a specific instruction. For example:

```
__usat()
```

This kind of optimization is quite intensive in terms of a developer's time. Typically such effort can only be justified for heavily used code, or code that forms part of a critical section of the application. The PMU can be used to help identify such sections.

## 4.3 ARM vs Thumb-2

The Cortex-R4(F) supports the ARMv7-R ARM and Thumb-2 instruction sets. Which one to use for a given section of code is an important development decision.

With ARMv5TE processors, such as the ARM946E-S, ARM state has to be used for particular operations. These included system control, exception handling and accessing the FPU. Thumb-2 supports all these operations.

One of the main advantages of the Thumb-2 instruction set compared to ARM is code density. Thumb-2 is a mixed 32 and 16-bit instruction set. Typically a large proportion of compiled code will use 16-bit instructions, giving smaller code size. Any code size savings will be code dependant. As an example here are the figures for the open source XVID codec:

| ARM code size (bytes) | Thumb code size (byte) |
|---|---|
| 271,816 | 197,756 |

Using RVCT 4.0 build 870, with –O2 optimization level and with —cpu=Cortex-R4. Quoted sizes include libraries and minimal initialization code.

Improved code density not only saves space in non-volatile memory, but can also lead to greater cache efficiency.

The hardware divide instructions (SDIV and UDIV) are only available in the Thumb-2 instruction set.

## 4.4 Branching

As discussed previously, flushing the pipeline leads to a performance penalty. To avoid unnecessary pipeline flushes the Cortex-R4(F) processor includes branch prediction logic.

In compiled C, branching will be used for function calls, function returns and loops. Less obviously, conditional statements such as IF…ELSE… may also be compiled using branch instructions.

Some forms of branching predict better than others. Consider the two examples below.

**Example 1:**

```
for (i=10; i > 0; i--)
{
  doSomething(i);
}
```

**Example 2:**

```
while (TRUE == pMyPeripheral->uiFlagReg)
{
  doSomething();
}
```

In Example 1 the for() loop will behave in a predictable pattern. For nine iterations the code loops (branches), and on the 10$^{th}$ iteration it will not branch. The call to doSomething() is not conditional. This arrangement can be learnt by the processor, and predicted.

Example 2 the number of iterations is dependent on an external condition. Unless this external condition follows a regular pattern, this loop is likely to predict poorly.

The purpose of these examples is not to say that you should never write code similar to example 2. Rather, that you should be aware of its affect on branch prediction.

### 4.4.1 Switch Statements in ARM and Thumb

The compiler will handle switch statements differently for ARM and Thumb code generation. Thumb includes the TBB (Table Branch Byte) and TBH (Table Branch Halfword) instructions, which can be used to efficiently compile switch statements. The example below shows typical code that would be generated for a switch statement with nine possible outcomes.

| C Code | ARM | Thumb-2 |
|---|---|---|
| switch (i)<br>{<br>case 0:<br>foo();<br>break;<br><br>….<br>case 8:<br>bar();<br>break;<br><br>default:<br>foobar();<br>break;<br>} | …<br>ADD pc,pc,r5,LSL #2<br>B \|L1.220\|<br>B \|L1.124\|<br>B \|L1.132\|<br>B \|L1.140\|<br>B \|L1.152\|<br>B \|L1.164\|<br>B \|L1.176\|<br>B \|L1.192\|<br>B \|L1.200\|<br>B \|L1.208\|<br>\|L1.124\|<br>MOV r0,r4<br>BL foo<br>B \|L1.240\|<br>…<br>\|L1.232\|<br>MOV r0,r4<br>BL foobar<br>\|L1.240\|<br>… | …<br>TBB [pc,r5]<br>DCB 0x05,0x07,0x09,0x0c<br>DCB 0x10,0x14,0x19,0x1b<br>DCB 0x1d,0x00<br>BL foo<br>B \|L1.152\|<br>…<br>BL bar<br>B \|L1.152\|<br>BL foobar<br>\|L1.152\|<br>… |

Using RVCT 4.0 build 870, with –O2 optimization level and with —cpu =Cortex-R4.

The Thumb implementation gives an improvement in code density. There is also a less obvious benefit. The branch predictor does not recognize the "`ADD pc, pc, r5, LSL #2`" used in the ARM implementation as a branch. It can therefore not be predicted.

### 4.4.2 Switch vs If

Switch and if statements are commonly used to select what action to take based on a state variable. Although there are differences in the definitions, a given section of code can normally be written using either of these statements.

When writing C it is worth considering what instructions will be generated by the compiler. Below is an example written using both approaches. In each case there are nine possible outcomes.

```
…
if (i == 0)
  foo()
else if (i == 1)
  bar()
….
else if (i == 8)
  foobar();
else
  barfoo();
…
```

```
  CMP       r4,#0
  BNE       |L1.116|
  BL        foo
  B         |L1.280|
|L1.116|
  CMP       r4,#1
  BNE       |L1.136|
  BL        bar
  B         |L1.280|
  …
|L1.256|
  CMP       r4,#8
  MOV       r0,r5
  BNE       |L1.276|
  BL        foobar
  B         |L1.280|
|L1.276|
  BL        barfoo
|L1.280|
```

Using RVCT 4.0 build 870, with –O2 optimization level and with —cpu =Cortex-R4.

```
…
switch (i)
{
   case 0:
    foo()
    break;

   ….
   case 8:
    foobar();
    break;

   default:
    barfoo();
    break;
}
…
```

```
  TBB       [pc,r5]
  DCB       0x05,0x09,0x0d,0x11
  DCB       0x15,0x19,0x1d,0x21
  DCB       0x25,0x00
  MOV       r0,r4
  BL        foo
  B         |L1.152|
  …
  BL        foobar
  B         |L1.152|
  BL        barfoo
  B         |L1.152|
|L1.146|
```

Using RVCT 4.0 build 870, with –O2 optimization level and with —cpu=Cortex-R4.

For the if statement the generated code checks each condition in sequence. Therefore the cycles required to call the required function depends on the number and order of the conditions. The cycles required when i==0 will be significantly fewer than when i==9.

The switch statement uses a table, with the condition as an index. This implementation requires the same number of cycles for each possible outcome. Making it more deterministic than the if. However, the cycles required when i==0 will typically be higher than in the if example.

Which is the best choice is therefore very much dependent on the requirements of the code. Typically, the greater the number of possible outcomes, the more attractive the switch statement becomes.

One thing that the compiler cannot consider, but you can, is the relative probability of given outcomes. It may be that i will be 0 on 75% of evaluations. In this scenario you could use if and switch in combination. For example:

```
if (choice == 0)
    foo()
else
{
  switch (i)
  {
     case 1:
     …
```

### 4.4.3 Nested Loops

Nested loops are another common code construct. They can be used, for example, when accessing multidimensional arrays. There are cases where the order of the accesses is important, and must be preserved to ensure correct operation. However, in many cases the ordering is unimportant.

We will concentrate on the case when the order is not important and where one dimension is larger than the other(s). In this case there is a choice whether the larger number of iterations should be in the outer or inner loop. For example:

```
for (i=100; i > 0; i--)
{
  for (j=10; j > 0; j--)
    doSomething(i, j);
}
```

```
for (j=10; j > 0; j--)
{
  for (i=100; i > 0; i--)
    doSomething(i, j);
}
```

The code generated by the compiler is, unsurprisingly, very similar:

```
  …
  MOVS      r5,#0x64
|L1.48|
  MOVS      r4,#0xa
|L1.50|
  MOV       r0,r5
  MOV       r1,r4
  BL        doSomething
  SUBS      r4,r4,#1
  BNE       |L1.50|
  SUBS      r5,r5,#1
  BNE       |L1.48|
  …
```

```
  …
  MOVS      r5,#0xa
|L1.68|
  MOVS      r4,#0x64
|L1.70|
  MOV       r1,r5
  MOV       r0,r4
  BL        doSomething
  SUBS      r4,r4,#1
  BNE       |L1.70|
  SUBS      r5,r5,#1
  BNE       |L1.68|
  …
```

Where the loops fit within the instruction cache and the number of iterations is low, there is little difference between the two approaches.

When the loop code is large, it is usually advisable to have the higher number iterations in the inner loop.

Where one dimension is relatively small (<2^5) and one large (>2^5), then it can be better to put the smaller number of iterations in the inner loop. The branch predictor can learn about the exit condition for a loop. For this example, the branch predictor

uses an 8-bit shift register, which switches to a 5-bit counter on saturation. For a loop with more iterations than this, the branch predictor cannot learn the exit condition.

## 4.5 Float vs Double

The Cortex-R4F processor includes a FPU that implements the VFPv3-D16 architecture. This provides support for both single and double precision. However, the implementation is optimized for single precision.

Therefore, where possible it is recommended to use single precision (floats) in preference to double precision (doubles).

## 4.6 Memory Management

Memory management is outside of the scope of this document. However, memory management will have a significant impact on the performance of your system. In particular, modern processors such as the Cortex-R4(F) are optimized for execution from the caches or TCM. Operating from non-cached external memory will often have a greater impact than the proportional difference in core and memory clock speeds.

For example, decoding a single frame using the open source XVID codec:

| | **Cycle count for decoding frame 0** | **Cycle count for decoding frame 1** |
|---|---|---|
| With caches enabled (64K I and D) | 39,663,416 | 21,757,269 |
| With caches and TCMs disabled | 204,963,576 | 114,034,034 |

Memory system running at 1/5 of processor clock, with zero wait state memory.

The comparison between cache and un-cached memory is an extreme one. A more likely scenario is deciding between different caching strategies, for example:

- read-allocate or write-allocate
- write-through or write-back

Write-back, write allocate is often a good choice for stack space. Data is written first (pushed) and then read back (popped), which fits a write-allocate strategy. Since the memory is also frequently re-written, write-back prevents unnecessary updates of memory. However, when using parity checking on the caches write-back is not supported.

# 5 Summary

In software development there is very rarely a “magic bullet” which can solve all a developer’s problems. Improvements in performance and functionality require developer time and effort.

This document has introduced the major features of the Cortex-R4(F), and explored how these features are used by the compiler. Understanding how these features are used can help a developer to make informed decisions when writing C targeted at the Cortex-R4(F).

We have also introduced the PMU. The PMU provides a valuable tool for analyzing the performance of code sections. When trying to understand why code does not provide expected performance, the PMU can be invaluable in investigating and improving execution.

# 6 Glossary

The following table describes some of the terms used in this document.

**Glossary terms**

| Term | Description |
|---|---|
| PMU | Performance Monitor Unit |
| LSU | Load Store Unit |
| FPU | Floating Point Unit |
| BIU | Bus Interface Unit |
| PFU | Pre-Fetch Unit |
| PMSA | Protected Memory System Architecture |