Application Note 55

Floating-Point Performance

Document number: ARM DAI 0055A

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

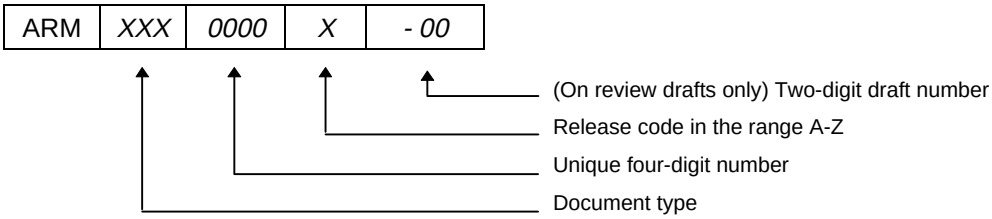
ARM
Advanced RISC Machines

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

There are a number of different floating-point options on ARM systems, including:

- Hardware coprocessors that execute a floating-point instruction set.
- Software emulation of these hardware coprocessors.
- A software library that implements floating-point arithmetic functions.

These represent various different trade-offs between floating-point performance, system cost and system flexibility. The purpose of this Application Note is to detail these trade-offs and provide an indication of the performance available from each option.

Some guidance is also provided about performance tuning of critical sections of floating-point code, with an example.

Other documentation about ARM floating- and fixed-point systems can be found in:

- *Software Development Toolkit User Guide* (ARM DUI 0040) ***Chapter 15 Floating-Point Support***
- *Application Note 23: The ARM Floating-Point Emulator* (ARM DAI 0023)
- *Application Note 33: Fixed-Point Arithmetic on the ARM* (ARM DAI 0033)
- *Application Note 40: Configuring the FPA Support Code/FPE* (ARM DAI 0040)

2 Floating-Point Systems

This section details the floating-point systems that can be used with an ARM processor, and the trade-offs that they make between floating-point performance, system cost and system flexibility.

2.1 Hardware floating-point systems

A hardware floating-point system typically consists of a hardware coprocessor for the ARM plus some associated “support code”. The latter handles exceptional conditions which occur too rarely or are too complex for the cost of handling them in hardware to be justified, or which require software intervention for other reasons (for example, if an error message is to be produced on floating-point overflow).

The main advantage of a hardware floating-point system is its high performance relative to software floating-point systems. A typical floating-point operation takes the order of two to ten machine cycles on a hardware coprocessor, compared with 50–100 or even more machine cycles for a software floating-point system. At identical clock rates, therefore, a hardware floating-point system is likely to be considerably faster than a software floating-point system.

The main disadvantages of a hardware floating-point system are:

- The silicon cost and power consumption of the coprocessor.
- Due to the lack of coprocessor instructions in the Thumb instruction set, all functions that are to use hardware floating-point must be compiled to ARM code. The resulting larger code size can increase system cost. Also, if a 16-bit or 8-bit memory system is being used, there will be some loss of integer performance associated with the use of ARM code.
- A hardware floating-point coprocessor has a maximum clock speed and requires a particular electrical interface to the ARM processor. Both of these factors can reduce the ability of a system using hardware floating-point to take advantage of improved ARM processors.

Hardware floating-point systems can therefore be characterized as very fast, but rather costly and inflexible compared with software floating-point systems.

Only one hardware floating-point system is currently available, namely the FPA11 floating-point coprocessor. It is incorporated into the ARM7500FE processor.

2.1.1 The FPA11 hardware floating-point system

The FPA11 (and its predecessor, the FPA10) is a floating-point hardware coprocessor, originally designed to work with the ARM2, ARM3 and ARM6 processors. It is also compatible with the subsequently developed ARM7 processor and its derivatives (including ARM7TDMI), but not with ARM8, ARM9 or StrongARM, which use different hardware interfaces to their coprocessors. Its maximum clock speed is roughly the same as that of the ARM7 processor on any particular process—the exact limit depends on the manufacturing process used.

The coprocessor instruction sequencing and hardware coprocessor interface used on ARM2–ARM7 constrains the speed at which floating-point data can be loaded or stored, to one single precision value per two cycles or one double precision value per three cycles. The FPA11 can achieve this speed provided certain hardware interlocks are observed; if they are not, an extra cycle can be taken. Typical arithmetic operations such as additions and multiplies take two to eight cycles, again with some interlock requirements; rarer arithmetic operations such as divisions can take up to 70 cycles. In addition, the FPA11 is capable of performing load/store operations and arithmetic operations in parallel with each other provided there are no data dependencies between them.

The cycle counts are designed to be “balanced” with respect to each other, for typical floating-point code containing about 60% load/store instructions and 40% arithmetic instructions. There are two main forms of this balance:

- In code that has not been scheduled to avoid interlocks and other effects due to data dependencies, the number of cycles used by the arithmetic instructions is roughly the same as the number used by the load/store instructions. This means that while it would be possible to improve the floating-point performance of such code by speeding up the arithmetic hardware, such attempts encounter a “law of diminishing returns”. For example, arithmetic hardware that was twice as fast would only improve the floating-point performance of such a system by about 30%.
- In code that has been scheduled to avoid interlocks, the number of cycles used by the arithmetic instructions is roughly the same as the number used by the load/store instructions, plus the number required for the ARM to issue the arithmetic operations to the FPA11 hardware. With the two being performed in parallel, this means that a speed-up of a factor of up to two is available from this scheduling. It also means that speeding up the arithmetic hardware produces little improvement in floating-point performance on most such code: it typically results in the arithmetic hardware standing idle while waiting for its operands to be loaded and its results stored.

2.2 Software floating-point systems

Floating-point can be performed in software by invoking code which breaks down the floating-point arithmetic operations into a sequence of integer operations, which are performed using ARM (or Thumb) instructions. There are typically 10–100 integer operations involved in this sequence (sometimes more), so this normally involves accepting lower floating-point performance than a hardware floating-point system could provide. The advantages of a software floating-point system are primarily the reduced silicon cost and the ability to benefit more quickly from ARM processor developments. (For example, when a new, faster ARM processor is developed, software floating-point systems benefit immediately. Hardware floating-point systems may have to wait until a faster coprocessor has also been developed.)

Software floating-point systems come in two main varieties: hardware emulators and floating-point libraries. They differ in the way in which the ARM software is invoked which performs the floating-point arithmetic.

2.2.1 Hardware emulators

Hardware emulators replace an absent floating-point coprocessor and its associated support code. The absence of the coprocessor causes the ARM processor to take its undefined instruction trap for all instructions belonging to that coprocessor, and this trap causes the floating-point software to be invoked. The hardware emulator therefore consists primarily of a trap handler which:

- Saves away the register values and other processor state associated with the code containing the floating-point instruction.
- Loads and decodes the floating-point instruction which caused the undefined instruction trap to be entered.
- Emulates that instruction, using the saved processor state to provide any register values and other state that it needs, and modifying the saved state if, for example, the instruction is supposed to modify a register value.
- Restores the processor state from the saved state and returns to the instruction following the floating-point instruction.

The main advantage of a floating-point hardware emulator is that it executes precisely the same code as the corresponding hardware floating-point system. Code can be compiled, assembled, linked and so on without committing to the precise floating-point system to be used. This can make it a good solution in cases where fast floating-point is not to be included in a basic system, but hardware-accelerated floating-point is to be provided as an optional extra.

The main drawback of a floating-point hardware emulator is that it is intrinsically slow. All the state-saving, instruction decoding and so on add up to a substantial overhead per floating-point instruction emulated. Additionally, each floating-point arithmetic operation in the original program typically has a number of associated load/store instructions. For example, the statement in the performance-critical inner loop of the Linpack benchmark contains two arithmetic operations (a multiply and an addition), but requires five FPA11 instructions to be executed (two loads, a multiply, an addition and a store). The net result is that the already-large overhead is multiplied up further: each floating-point arithmetic operation in the Linpack benchmark has an overhead equal to about 2.5 times the instruction emulation overhead.

The FPE (Floating-Point Emulator) is a hardware emulator for the FPA11, and is included in the ARM Software Development Toolkit.

2.2.2 Floating-point libraries

An alternative software floating-point solution is to make the application code invoke the floating-point software directly, via library function calls. This requires compilers and assembler code writers to generate function calls rather than floating-point instructions when floating-point arithmetic is required, and thus means that different binaries must be produced than those for hardware coprocessors or emulators for them. This lack of compatibility with floating-point hardware is the main disadvantage of floating-point libraries.

Note *In principle, it would be possible to address this partially with a floating-point library which performs its floating-point arithmetic by invoking the hardware coprocessor, rather than with a sequence of integer operations. However, the procedure entry/exit instructions involved would create a substantial overhead and a large percentage of the hardware's performance would be lost—enough to make it very doubtful whether the silicon cost of the hardware is justified. This solution has therefore not yet been used or implemented.*

The main advantage of a floating-point library is that it is substantially faster than a hardware emulator. This is due to three main factors:

- A function call involves much less overhead than entering an undefined trap handler.
- Fewer function calls need to be made than the number of instructions which would have had to be emulated—normally, only one function call is required per floating-point arithmetic operation.
- The fact that fewer function calls are made also means that the floating-point work is being done in larger “packages”. This can make optimizations available to floating-point library code that cannot easily be exploited by a hardware emulator.

Between them, these factors can cause an improvement in floating-point performance by a factor of two or more between a system using a hardware emulator and one using a floating-point library.

There are also typically some benefits in code size from using a floating-point library. This is because armlink automatically links in only those library routines that the application code actually uses. Including a hardware emulator in a system typically means that the emulation routines for all of that hardware’s functionality are included, regardless of whether all of the functionality is actually used.

A floating-point library is therefore the recommended floating-point system in most cases where hardware floating-point performance is not required. A floating-point library is included in the ARM Software Development Toolkit, and is the default floating-point system in the toolkit as supplied.

2.3 Floating-point system selection when compiling

The C compiler needs to be told which floating-point system to compile code for, as it needs to know whether to generate floating-point instructions or floating-point library calls when floating-point arithmetic is required. This is done via qualifiers to the `-apcs` option:

- Use the `/hardfp/fpe3` qualifiers in order to compile for the FPA11 or the FPE.
- Use the `/softfp` qualifier in order to compile for the floating-point library.
- Do not use the `/hardfp/fpe2` qualifiers unless you need compatibility with pre-FPA10 implementations of the FPA11 instruction set.

3 Hardware, Software and Benchmarks Used

The hardware system used for the performance measurements in ***4 Floating-Point Performance of Selected Systems*** (on page 8) contained an ARM7500FE processor, running at 32 MHz. The ARM7500FE contains an FPA11 macrocell, which was enabled for the FPA11 measurements by installing the FPA11 Support Code. For the FPE measurements, the FPE was installed instead and the FPA11 was therefore left disabled. For the floating-point library measurements, its use was selected at compile time, as described in ***2.3 Floating-point system selection when compiling*** on page 6.

The hardware's memory consisted of EDO RAM, also running at 32 MHz. All measurements should scale according to the clock speed, up to the maximum clock speed at which the hardware works.

All compilation, linking and so on was performed using version 2.11 of the ARM Software Development Toolkit.

The main benchmark used was a C version of the Linpack benchmark. This benchmark's inner loop consists of code of the form:

```
for (i = 1; i <= n; i++)
  dy[i] = dy[i] + da * dx[i];
```

All floating-point variables are double precision.

For the FPA11 coprocessor and its hardware emulator (the FPE), this code involves two loads (of `dy[i]` and `dx[i]`), a multiplication, an addition and a store (of `dy[i]`) per iteration. The value `da` is kept in a floating-point register throughout the loop, so does not need to be loaded within the loop. Profiling of various floating-point programs has shown that this mix of floating-point operations is fairly typical.

For the floating-point library, this code involves one procedure call each to the double precision arithmetic routines `_dadd` and `_dmul`, plus various ARM instructions to access the arrays of floating-point values.

A second version of the Linpack benchmark was also used, in which the inner loop shown above is optimized for the FPA11. This optimization primarily consists of rewriting the C code in such a way as to induce the compiler to avoid data dependencies between adjacent instructions. This enables the FPA11 to avoid interlocks and take advantage of its ability to perform load/store and arithmetic instructions in parallel. The results of this version of the benchmark are primarily of interest because they indicate what sort of additional performance the FPA11 is capable of if code is scheduled well.

For more details of how this code was optimized for the FPA11, see ***6 Example: Optimizing the Linpack Inner Loop*** on page 12.

4 Floating-Point Performance of Selected Systems

The following table shows floating-point performance figures obtained in practice from the systems and benchmarks described in ***3 Hardware, Software and Benchmarks Used*** on page 7.

Floating-point system	Linpack performance	FPA11-optimized Linpack performance
FPA11 + support code	1.83 Mflops	2.75 Mflops
FPE (FPA11 emulator)	56 kflops	53 kflops
Floating-point library	150 kflops	150 kflops

These figures illustrate the main points made about floating-point performance in ***2 Floating-Point Systems*** on page 3. The hardware floating-point system is the fastest, and is more than 30 times faster than its hardware emulator (rising to 50 times faster if the code is optimized for the hardware). However, if the use of floating-point hardware is not anticipated and a pure software floating-point system is wanted, the floating-point library offers a significant speed advantage over the use of a hardware emulator (by a factor of about 2.7 for this particular program).

Note *The exact cause of the slight drop in FPE performance when the benchmark is optimized for the FPA11 is unknown, but is probably a cache effect caused by reduced locality in its use of data and/or the FPE code.*

5 Improving Floating-Point Performance

This section gives a number of suggestions for improving the performance of a program that uses floating-point arithmetic. It addresses this issue at the "micro-optimization" level; that is, it assumes that more global optimizations like selecting the right algorithm for the job have already been done, and that all that needs to be done at this point is to make the program use the chosen floating-point system as efficiently as possible.

It is strongly recommended that the sort of optimizations described in this section are only performed at a late stage in development, after ensuring that all sensible algorithmic optimizations have already been performed. Trying to apply these optimizations too early in a development can lead to a lot of wasted effort if the algorithms are subsequently changed.

As always with optimization work, it is important to profile the code being optimized, to find out which code sections are using significant amounts of time, and to concentrate the optimization efforts on those sections. This is particularly true for micro-optimization work, since it is typically labor-intensive and often system-specific: unless it is carefully targeted at the areas which will really benefit from it, such work can consume a lot of effort for very little return.

The system-specific nature of micro-optimization work also means that it is usually worth keeping a record of which code sections were optimized and of the pre-optimization code, to aid with porting to new floating-point systems. In C, a simple technique for doing this involves the use of conditional compilation of the following form:

```
#if defined(OPTIMIZED_FOR_FPA11)
  ... code optimized for FPA11 ...
#elif defined(OPTIMIZED_FOR_FPE)
  ... code optimized for FPE ...
#else
  ... original unoptimized code ...
#endif
```

Further system-specific optimizations can easily be added to this structure, and can be based on the unoptimized code (as opposed to having to undo the optimizations for one of the other systems before starting to optimize for the new system).

The rest of this section describes some floating-point micro-optimization techniques that should be of benefit on all floating-point systems, and some system-specific techniques for particular floating-point systems. (No system-specific section exists for the floating-point library, as the optimization techniques that apply to it are covered in the general section.)

5.1 General techniques for optimizing floating-point code

It is often worth examining the assembler output of the compiler (using the `-S` option) for inefficiencies in the code it has generated. One particular point to bear in mind when generating code for a floating-point coprocessor (or an emulator of one) is that ARM coprocessor instructions do not include register-indexed loads and stores. This makes indexed accesses to arrays of floating-point numbers take at least two instructions—one to calculate the correct address, the other to perform the memory access.

The coprocessor load/store instructions do, however, include forms which update their base register by a constant. In code which scans through an array sequentially, this means that indexed array accesses can often be replaced profitably with pointer accesses, together with increments or decrements of the pointers. (For an example of this, see ***6 Example: Optimizing the Linpack Inner Loop*** on page 12.)

Try to avoid divisions where possible: they are significantly slower than multiplications and additions on most floating-point systems. For example, if many numbers are to be divided by the same value, it is often worth calculating the reciprocal of the divisor and multiplying by it instead:

```
for (i = 0; i < n; i++)             /* Original code      */
  data[i] = data[i] / scalefactor;

reciprocal = 1.0 / scalefactor;     /* Faster alternative */
for (i = 0; i < n; i++)
  data[i] = data[i] * reciprocal;
```

Overall, this replaces N divisions by one division and N multiplications, and produces a substantial gain in performance on all ARM floating-point systems to date.

However, take care with this optimization in code which is sensitive to rounding errors. There are two rounding errors involved in the calculation of each final value of `data[i]` (one when calculating the reciprocal, the other in the multiplication). This makes its maximum overall rounding error larger than that of the original code.

5.2 Optimizing floating-point code for the FPA11

There are three important aspects of the FPA11 that lead to opportunities to optimize floating-point code:

- Interlocks due to pipelining effects: most instructions will take an extra cycle or two if their result is used by the immediately following instruction.
- Parallel execution: FPA11 instructions which involve no data communication with the ARM processor (arithmetic instructions, but not load/store instructions, FIXes, FLTs, compares and FPSR transfers) only require a single ARM cycle each to issue them to the FPA11. All the remaining cycles of their execution can occur in parallel with ARM instructions. They can also occur in parallel with floating-point load/store instructions, provided that:
  - Store instructions do not try to store the results of the arithmetic instruction: if they do, they will interlock until the result of the arithmetic operation is ready.
  - Load instructions do not overwrite the operands of the arithmetic instruction too soon: if they do, they may have to interlock until it is known that the operand is not required for subsequent exception processing.
- Speculative execution: once an instruction has entered the ARM's pipeline, the FPA11 can start executing it. The speculative execution is restricted to execution that does not change the coprocessor state: for example, if a multiply instruction is executed speculatively, the multiplication itself will occur, but the result will not be written back to the register file until the instruction reaches the execute stage of the ARM's pipeline. (If the instruction does not reach the execute stage of the ARM's pipeline, due for example to an intervening branch, the result is discarded.)

Producing code which exploits these features of the FPA11 optimally is quite hard. However, observing the following guidelines usually leads to results that are close to optimal:

- Try to interleave load/store instructions with arithmetic instructions as far as possible.
- Try to avoid cases where two consecutive floating-point instructions use the same floating-point registers.
- Try to execute other instructions (ARM instructions or floating-point load/store instructions) in parallel with arithmetic instructions. This rule is particularly important for multiply instructions, because:
  - Unlike most simpler arithmetic instructions (such as additions), they take enough cycles (five for FML or eight for MUF) that it is usually worthwhile finding more than one instruction that can execute in parallel with the multiply.
  - Unlike the more complex arithmetic instructions (primarily divisions), they do not take so long that it is hard to find enough instructions to execute in parallel with them.
- When placing instructions that are to be executed in parallel with an arithmetic instruction, it is normal to place them after the arithmetic instruction. However, speculative execution means that multi-cycle instructions can be placed one or two instructions before the arithmetic instruction and still effectively be executed in parallel with them.

An example which uses most of these guidelines can be found in ***6 Example: Optimizing the Linpack Inner Loop*** on page 12.

5.3 Optimizing floating-point code for the FPE

The FPE contains an optimization to avoid some of the overhead associated with every emulation of a floating-point instruction. Before returning from emulating one instruction, it checks whether the next instruction in memory is also a floating-point instruction. If so, it emulates that instruction as well, then looks at the next instruction. If not, it returns normally.

The net result is that the overhead of entering and leaving the FPE is only paid once per sequence of consecutive floating-point instructions, not once per floating-point instruction. Other aspects of the FPE's overhead (such as loading and decoding the instruction) still have to be paid once per instruction, but a significant gain in performance (of the order of 10–20%) can be obtained by trying to ensure that the floating-point instructions occur in "clusters".

Unfortunately, in some cases, this conflicts with the wish to intersperse ARM and floating-point instructions to take advantage of the FPA11's parallel execution ability. When this occurs, it is necessary to decide whether FPA11 or FPE performance is more important for the application concerned.

6 Example: Optimizing the Linpack Inner Loop

This section describes how the Linpack benchmark's inner loop was changed to produce the optimized version described in ***3 Hardware, Software and Benchmarks Used*** on page 7. The code was optimized for the FPA11; there were some slight negative side effects on its performance on the FPE.

The original C code for the inner loop was:

```
for (i = 1; i <= n; i++)
  dy[i] = dy[i] + da * dx[i];
```

In this code, `i` and `n` are integers, `da` is a double precision floating-point number and `dx[]` and `dy[]` are arrays of double precision floating-point numbers. It is also known from an earlier test in the function concerned that `n` is greater than 0: this allows some of the optimizations below to be expressed more simply. (The compiler does not know this, however, so some of the code it generates contains redundant tests of whether `n` is less than 1.)

Compiling this for the FPA11 with the compiler's `-S` option led to the following assembly code.

Note *All comments have been inserted by hand after compilation, rather than being generated by the compiler, and the variable `da` is in register `f0` at the start of the code—the same is true for all other assembler code in this section.*

```
        MOV     a2,#1                   ; i = 1
        CMP     a1,#1                   ; Test for n < 1, and return
        LDMLTIA sp!,{v1-v4,pc}          ;  if so (won't ever happen)
|L0007d8.J21.daxpy|
        ADD     a3,lr,a2,LSL #3         ; Address dy[i]
        LDFD    f2,[a3,#0]              ;  and load dy[i] into f2
        ADD     ip,a4,a2,LSL #3         ; Address dx[i]
        LDFD    f1,[ip,#0]              ;  and load dx[i] into f1
        MUFD    f1,f1,f0                ; Calculate da * dx[i],
        ADFD    f1,f2,f1                ;  then dy[i] + da * dx[i]
        STFD    f1,[a3,#0]              ;  and store back to dy[i]
        ADD     a2,a2,#1                ; i++
        CMP     a2,a1                   ; Test for i <= n
        BLE     |L0007d8.J21.daxpy|     ;  and loop if so
```

The first thing to do to this code is to apply the techniques described in ***5.1 General techniques for optimizing floating-point code*** on page 9. The code does not contain any divisions, but does contain some array accesses. These result in two ADD instructions to calculate the addresses of the required elements.

Since this code scans through the `dx[]` and `dy[]` arrays sequentially, it is easy to add two suitable pointer variables and rewrite the code to use pointer arithmetic:

```
double *xptr, *yptr;

xptr = &dx[1];
yptr = &dy[1];
for (i = n-1; i >= 0; i--)
{
  *yptr = *yptr + da * (*xptr);
  yptr++;
  xptr++;
}
```

The loop variable `i` is no longer used as an array index, and is now only needed for the purpose of ensuring that the loop is executed the right number of times. This allows the original loop variable, which incremented from one to `n`, to be replaced by one which decrements from `n-1` to zero. This results in more efficient ARM code, as the comparison with zero can be incorporated into the loop index subtraction.

This results in the following assembler code from the compiler:

```
        ADD     a2,a4,#8              ; xptr = &dx[1]
        ADD     a3,lr,#8              ; yptr = &dy[1]
        SUBS    a1,a1,#1              ; i = n-1
        LDMMIIA sp!,{v1-v4,pc}        ; (redundant) if n<1, return
|L0007dc.J21.daxpy|
        LDFD    f1,[a2],#8            ; Load *xptr, then xptr++
        MUFD    f2,f0,f1              ; Calculate da * (*xptr)
        LDFD    f1,[a3,#0]            ; Load *yptr
        ADFD    f1,f2,f1              ; Calc. *yptr + da * (*xptr)
        STFD    f1,[a3],#8            ; Store to *yptr, then yptr++
        SUBS    a1,a1,#1              ; i--
        BPL     |L0007dc.J21.daxpy|   ; Loop if i >= 0
```

Note *As a side effect of these changes, the floating-point instructions occur in a single cluster of five instructions per iteration, rather than as an isolated instruction and a cluster of four instructions. In principle, this should slightly improve the code's performance on the FPE (see* ***5.2 Optimizing floating-point code for the FPA11*** *on page 10); unfortunately, other negative side effects (probably cache-related) of the changes appear to have outweighed this.*

This code is now reasonably efficient as far as the ARM code is concerned. However, on the FPA11 there are a number of inefficiencies:

- The MUFD instruction uses the result of the first LDFD instruction, resulting in an interlock between them. There are similar interlocks between the second LDFD and the ADFD, and between the ADFD and the STFD.
- The only instruction that can be executed in parallel with the MUFD instruction is the second LDFD instruction. This does not provide enough cycles of parallel execution to cover the latency of the MUFD instruction, with the result that the ADFD has to wait for the result of the MUFD, and so will interlock with it.
- With the ADFD instruction depending on the result of the instruction before it and having its result used by the next instruction, there is no possibility of any code being executed in parallel with the ADFD.
- The "loop overhead" instructions (the SUBS and BPL at the end of the loop) are executed at a time when the FPA11 has nothing to do.

As a result, the "critical path" through the loop code passes through the LDFD, MUFD, ADFD, STFD, SUBS, BPL instructions, with each floating-point instruction depending on the previous one. This means that the "latency" cycle counts for the FPA11 are appropriate, leading to a cycle count of about 4+4+9+4+1+3 = 25 cycles per loop iteration. (The original code took three cycles more, or about 28 cycles per loop iteration.)

To improve this, the first step is to use some extra double precision variables in the C code to break the calculation up into the separate load, multiply, add and store operations:

```
double *xptr, *yptr, x_loaded, y_loaded, product, sum;

xptr = &dx[1];
yptr = &dy[1];
for (i = n-1; i >= 0; i--)
{
  x_loaded = *xptr;
  product = da * x_loaded;
  y_loaded = *yptr;
  sum = y_loaded + product;
  *yptr = sum;
  yptr++;
  xptr++;
}
```

This makes no real difference to the compiled code (a few register allocations change), but permits the next step, which is to re-arrange the loop so that the loop overhead instructions occur just after the multiplication. This involves taking one complete iteration out of the loop, with the multiply and all instructions preceding it being taken out of the start of the loop, and all instructions following it being taken out of the end of the loop:

```
double *xptr, *yptr, x_loaded, y_loaded, product, sum;

xptr = &dx[1];
yptr = &dy[1];
x_loaded = *xptr;
product = da * x_loaded;
y_loaded = *yptr;
for (i = n-2; i >= 0; i--)
{
  sum = y_loaded + product;
  *yptr = sum;
  yptr++;
  xptr++;
  x_loaded = *xptr;
  product = da * x_loaded;
  y_loaded = *yptr;
}
sum = y_loaded + product;
*yptr = sum;
yptr++;
xptr++;
```

It is no longer clear that the loop will be executed at least once: this means that the compiler-generated start-of-loop test is no longer redundant. However, benefit is still gained from the original knowledge that the loop was executed at least once—otherwise, this code would need to be surrounded with a test that n is greater than zero.

After eliminating the final `yptr++` and `xptr++` statements on the grounds of redundancy, this compiles to the following assembler code:

```
        ADD     a2,a4,#8              ; xptr = &dx[1]
        ADD     a3,lr,#8              ; yptr = &dy[1]
        LDFD    f0,[a2,#0]            ; x_loaded = *xptr
        MUFD    f0,f1,f0              ; product = da * x_loaded
        LDFD    f2,[a3,#0]            ; y_loaded = *yptr
        SUBS    a1,a1,#2              ; i = n-2, and branch past
        BMI     |L000804.J22.daxpy|   ;  loop if i already < 0
|L0007e8.J21.daxpy|
        ADFD    f0,f2,f0              ; sum = y_loaded + product
        STFD    f0,[a3],#8            ; *yptr = sum, then yptr++
        LDFD    f0,[a2,#8]!           ; xptr++; x_loaded = *xptr
        MUFD    f0,f1,f0              ; product = da * x_loaded
        LDFD    f2,[a3,#0]            ; y_loaded = *yptr
        SUBS    a1,a1,#1              ; i--
        BPL     |L0007e8.J21.daxpy|   ; Loop if i >= 0
|L000804.J22.daxpy|
        ADFD    f0,f2,f0              ; sum = y_loaded + product
        STFD    f0,[a3,#0]            ; *yptr = sum
```

Now the MUFD instruction is executed in parallel with an LDFD instruction and the loop overhead instructions. There are still some interlocks, but the critical path through the code has been reduced to the LDFD/MUFD pair at the end of one iteration followed by the ADFD/STFD pair at the start of the next iteration, or about 4+9+4+4 = 21 cycles.

One final optimization is possible in most circumstances: the operations can be reordered in the loop so as to reduce the amount of interlocking and further increase the number of instructions that can be executed in parallel with the floating-point arithmetic instructions. To do this, the order of the STFD instruction and the following LDFD instruction must be reversed. This is legitimate provided that the two instructions are not accessing the same double precision number—in this example, that `dx[i+1]` and `dy[i]` are not the same location. This can normally be expected to be the case, and can in fact be verified as always being true in the Linpack benchmark.

Having interchanged the STFD with the following LDFD, the operations in the loop can be rearranged almost at will. Note in particular that you can always interchange the STFD with the other LDFD, since they access different elements of the same `dy[]` array. A reasonable final order for the resulting code is:

```
double *xptr, *yptr, x_loaded, y_loaded, product, sum;

xptr = &dx[1];
yptr = &dy[1];
x_loaded = *xptr;
product = da * x_loaded;
y_loaded = *yptr;
for (i = n-2; i >= 0; i--)
{
  xptr++;
  x_loaded = *xptr;
  sum = y_loaded + product;
  y_loaded = *(yptr+1);
  product = da * x_loaded;
  *yptr = sum;
  yptr++;
}
sum = y_loaded + product;
*yptr = sum;
```

This compiles to the following code:

```
        ADD     a2,a4,#8              ; xptr = &dx[1]
        ADD     a3,lr,#8              ; yptr = &dy[1]
        LDFD    f0,[a2,#0]            ; x_loaded = *xptr
        MUFD    f0,f2,f0              ; product = da * x_loaded
        LDFD    f1,[a3,#0]            ; y_loaded = *yptr
        SUBS    a1,a1,#2              ; i = n-2, and branch past
        BMI     |L00080c.J22.daxpy|   ;  loop if i already < 0
|L0007f0.J21.daxpy|
        LDFD    f4,[a2,#8]!           ; xptr++; x_loaded = *xptr
        ADFD    f3,f1,f0              ; sum = y_loaded + product
        LDFD    f1,[a3,#8]            ; y_loaded = *(y_ptr+1)
        MUFD    f0,f2,f4              ; product = da * x_loaded
        STFD    f3,[a3],#8            ; *yptr = sum, then yptr++
        SUBS    a1,a1,#1              ; i--
        BPL     |L0007f0.J21.daxpy|   ; Loop if i >= 0
|L00080c.J22.daxpy|
        ADFD    f0,f1,f0              ; sum = y_loaded * product
        STFD    f0,[a3,#0]            ; *yptr = sum
```

This code comes close to obeying all the guidelines given in ***5.2 Optimizing floating-point code for the FPA11*** on page 10: the only breach of them within the loop is that the ADFD instruction is followed by an LDFD instruction that loads one of the ADFD's operands. Determining how many cycles it takes per iteration is quite a complex modeling job and beyond the scope of this Application Note; it can however be expected to lie close to the higher of the two minima implied by the ARM instruction cycle counts and by the FPA11's arithmetic unit. The former minimum is 3+1+3+1+3+1+3 = 15 cycles for the seven instructions in the loop; the latter is 2+8 = 10 cycles for the ADFD and MUFD instructions. This code can therefore be expected to take something of the order of 15–17 cycles per iteration.

Compared with the original cycle count of 28 cycles per iteration, this represents a performance increase by a factor in the range 1.65–1.85. After taking account of the fact that this loop does not contain all the floating-point operations in the Linpack benchmark, and that the optimized code is relatively more affected by overheads such as cache misses, this calculated increase is a reasonable match to the performance increase by a factor of 1.5 actually observed in practice (see ***4 Floating-Point Performance of Selected Systems*** on page 8).

Further increases in performance are possible by techniques such as loop unrolling. These may however require somewhat more serious trade-offs between code size and performance than the optimizations performed so far. Despite the large increase in C source code size, the optimizations above have only resulted in a net increase in the code size by three instructions (from 13 originally to 16 at the end).