ARM Programming and Optimisation Techniques

Vijaya Sagar Vinnakota (vijaya.sagar@wipro.com)
Wipro Technologies

Abstract

This tutorial paper presents some optimisation techniques for programming with ARM processors, addressing both memory (footprint) and execution time optimisation. For purposes of illustration, the ARM7TDMI is chosen as a representative of the ARM family. The 32-bit ARM instruction set is chosen as the vehicle to convey various optimisation techniques while programming references to the 16-bit Thumb instruction set are minimal.

The tutorial does not assume prior knowledge of the ARM architecture. Experience in programming with any RISC assembly language makes understanding easier, though.

Keywords

ARM, RISC, CISC, footprint, Thumb, pipeline, stall, auto-indexing, profiling, loop unrolling

Introduction

From being the first commercial implementation of RISC architecture, the ARM processor family has come a long way over the past decade in becoming the primary choice for low-power (yet powerful) embedded applications.

ARM follows the standard load-store (read as 'no memory-memory operations') and fixed-length (not necessarily single-cycle) instruction architecture commonly seen in all RISC processors. It uses an instruction pipeline to improve the processor performance.

Since pure RISC implementations are prone to induce program size inflation, the ARM designers chose to borrow a few CISC concepts in carefully chosen areas. For instance, multiple load / store, stack addressing and auto-indexing are all new to the RISC philosophy. At the same time, a few RISC concepts were left out of the architecture for power, die-size and performance considerations. For instance, register-files and single cycle execution have been dropped off the ARM architecture.

For the footprint conscious system designers and developers, the ARM architects provided a novel solution by way of a restricted 16-bit instruction set (as against the standard 32-bit) known as 'Thumb'. While a 50% reduction program size over ARM instruction set may not be feasible, the gains are seldom less than 30%.

These novel designs in the ARM architecture resulted in demonstrated gains such as die-size reduction, low gate-complexity, high MIPS/watt and increased code density.

The ARM architecture

To write efficient programs, a thorough understanding of the target processor architecture is a

prerequisite. As shall be seen in the course of the tutorial, even the best of the compilers have their limitations. There is always scope for improvement in the code that they generate, for they are tools meant for generic use. They know very little (if at all) of the specific program instance to be compiled. Only the programmer[1] knows his program best – and hence stands the best chance of converting it into code that is most optimised for the target processor, provided its architecture is well understood. With this as the motivation, let us take a quick tour of the ARM architecture.

Building blocks

An ARM processor is made up of the following blocks:

- o *The register bank* – contains 37 registers[2] visible to the programmer
- o *The barrel shifter* – for shifting and rotating data
- o *The ALU*
- o *The instruction decoder*
- o *The control logic* – for generating necessary signals such as memory-read/write
- o *The address and data registers* - to serve as latches (these are not 'programmable')
- o *The CP15* – ARM system control coprocessor which serves, among other things, as an MMU and cache controller

Some of these blocks are of direct interest to the programmer and would be discussed in greater detail later.

Pipelined execution

The ARM architecture has a 3-stage[3] instruction pipeline for improving the

---

[1] Though referred to only as a male - purely for writing convenience, the author acknowledges many a fine female programmer, right since the days of Lady Ada.
[2] 20 of these registers are banked and are not directly visible to the programmer

processor performance and utilisation of its functional units.
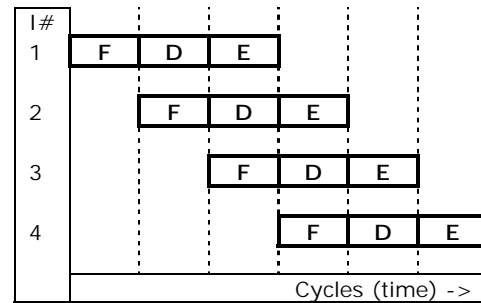

Fig.1 – steady state ARM pipeline

F: Fetch   D: Decode   E: Execute

In the *fetch* stage, the instruction is brought into the pipeline from memory (cache / ROM / RAM).

In the *decode* stage, the instruction is decoded and control signals are made ready for activities such as fetching the next-but-one instruction and memory address calculation.

In the *execute* stage, the instruction takes effect, primarily in the form modification to one or more of the registers (even in the case of a load/store instruction as memory interaction is through the invisible address and data latch registers).

Fig.1 depicts an ideal case, steady state scenario for the ARM pipeline, where in each instruction takes exactly three cycles to complete (i.e., each stage of the instruction needs exactly one cycle). For example, consider a simple 'add' instruction:

```
ADD   r0, r0, r1  ; r0 = r0 + r1
```

This instruction goes through the following stages:

1. Fetch the instruction from memory
2. Decode the instruction and identify that it is an ADD

---

[3] Since ARM8, pipelining schemes other than the simple "fetch-decode-execute" have been implemented

instruction (of type register-register)

3. Pass the current values of the operand registers (r0 and r1) to the ALU, obtain their sum and store it in the destination register (r0)

However, this need not always be the case. For example, consider the case of a 'load' instruction:

```
LDR   r0, [r1], #0 ; r0 = *r1
```

This instruction goes through the following stages:

1. Fetch the instruction from memory
2. Decode the instruction and identify that it is an LDR instruction
3. Compute the memory address which contains the data to be loaded, using the base register (r1) and offset (#0)
4. Fetch the data word from memory into the data latch register
5. Transfer the data word from the latch register to the destination register (r0)

It can be seen that 'add' takes 1-cycle for its 'execute' stage whereas 'load' needs 3-cycles for the same (shown in a grey shade). See also, fig.2.

The ARM instruction set designers have made good use of this extra latency by adding *auto-indexing* capability to 'load'/'store' instructions. Since the ALU and barrel-shifter would otherwise be idle during the data-word cycle (#4) of the 'execute' stage, they can be used to add an offset (index) to the base register. This feature is frequently used in the programs illustrated in this paper.

PC runs ahead! If instruction #1 in fig.1 were to refer to PC, it would find that the register value is not the same as the instruction address. This is

understandable because #1 would 'see' the value of PC in its 'execute' stage, by which time/cycle the PC has been incremented twice (for fetching #2 and #3). Thus, PC runs ahead of the current instruction and holds the address of the next but one instruction. Assembly language programmers must factor this while directly accessing the PC value.

Pipeline stall: Instructions such as load/store that take multiple cycles to complete the 'execute' stage adversely affect the pipeline throughput. This is due to the fact that such instructions occupy the processor functional units for more than one cycle in the 'execute' stage, thus stopping the progress of subsequent instructions in the pipeline. This condition is described as a *stall* in the pipeline.
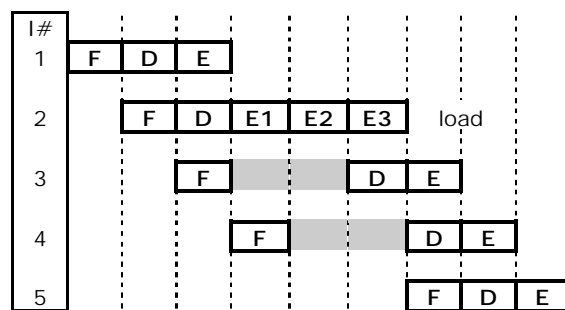

Fig.2 – ARM pipeline with stalls

Fig.2 depicts a stall condition when a 'load' (#2) is mixed with simple instructions such as 'add' and 'sub'. 'En' stands for cycle-#n in the execution stage of the load instruction described in detail earlier.

The decode stage of the subsequent instruction (#3) cannot happen along with E1 since the decode logic is occupied for generating data-fetch related control signals.

Instruction#3 cannot enter its decode stage along with E2 either, but for a different reason altogether. For even if it were allowed, the control signals generated cannot be used in the immediate next cycle because of E3

being in progress. Another way to view this is to understand that during E2, the control logic was busy generating signals for transferring data from the latch register to its destination in the register bank.

For obvious reasons, instruction #4 also experiences the effects of this stall.

Pipeline breaks: Another throughput limiting condition arises when a branch instruction enters the pipeline.
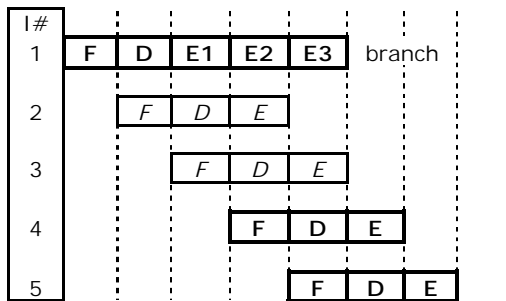

Fig.3 – ARM pipeline with a branch

As shown in fig.3, a branch instruction (#1) takes three cycles in its 'execute' stage of the pipeline. This view holds both for unconditional and condition-satisfied branch instructions.

In E1, the branch target address is computed and fed to the addressing logic. In E2, if a 'branch and link' has been requested, the link register is updated with the return address. During the same time, the branch target instruction is fetched into the pipeline. In E3[4], as the pipeline continues to be filled, the link register is adjusted to offset the PC-run-ahead effect.

Instructions #2 and #3 (which entered the pipeline as #1 was still in its

decode / execute stage) are discarded without getting executed.

Thus, branch instructions impose a significant overhead and disrupt the smooth flow in the pipeline. This is not to say that branches should not be used (for it is impractical), but only to emphasise the fact that knowledge of the pipeline behaviour guides a better program design, which leads to lesser number of branches.

ARM instruction set

The ARM instruction set can be organized into the following broad categories[5]:
1. Data transfer instructions
2. Data processing instructions
3. Branch instructions
4. Coprocessor instructions
5. Miscellaneous instructions

The purpose of this tutorial is served better if, instead of exhaustively listing all the instructions, the most frequently used ones are explained in the context of common programming tasks.

ARM assembly examples

Simple arithmetic

```
int   a, b, c;
```

Say, variables **a**, **b** and **c** are in registers **r0**, **r1** and **r2** respectively.

Addition:

a = b + c;
ADD   r0, r1, r2 ; r0 = r1 + r2

a += b;
ADD   r0, r0, r1

---

[4] E3's primary purpose is to generate control signals for fetching instruction #5. Then on, the pipeline is back to steady state.

[5] This categorisation is not from the instruction set design viewpoint but only from that of an assembly programmer.

Subtraction:

a = b – c;
SUB   r0, r1, r2 ; r0 = r1 – r2

a -= b;
SUB   r0, r0, r1

a = 25 – b;
RSB   r0, r1, #25 ; r0 = 25 – r1 ; RSB = reverse subtract

Multiplication:

a = b * c;
MUL   r0, r1, r2 ; r0 = r1 * r2

a *= b;
MUL   r0, r0, r1 ; illegal !

MUL cannot be used with the same register for the multiplicand and the product. In practice, this restriction does not pose a significant handicap as multiplication is commutative. The instruction can simply be re-written as:

```
MUL    r0, r1, r0  ; r0 = r1 * r0
```

Multiply and accumulate:

a += b * c;
MLA   r0, r1, r2, r0   ; r0 = r1 * r2 + r0

Conditional execution – a digression

Before going through any further instructions, let us take a quick look at *conditional execution* – arguably the most powerful feature in the ARM instruction set.

Every single ARM instruction can be executed conditionally. In fact, every single ARM instruction *is* executed conditionally. To achieve this, the instruction set designers have set aside 4-MSbits out of the 32-bit instruction word for specifying a condition code. This condition code is related to the status of four flags in the CPSR (current program status register) – **N**egative, **Z**ero, **C**arry, o**V**erflow. Some of these codes are:

code	description	condition
MI	minus	N is set
EQ	equals [to zero]	Z is set
LO	unsigned lower	C is clear
VC	no overflow	V is clear

These two letter condition codes are to be suffixed to an instruction mnemonic to affect the desired conditional execution.

Now, if such were indeed the case, how is it that the arithmetic examples listed earlier used no such suffixes? The answer lies in the default condition code – *AL* (always) which implies execution irrespective of the condition flags' status. The net effect is one of unconditional execution! The previous examples could as well have been written as:

```
ADDAL r0, r1, r2  ; r0 = r1 + r2
SUBAL r0, r1, r2  ; r0 = r1 – r2
MULAL r0, r0, r1  ; illegal ;^)
```

However, for the sake of readability, AL is better left unspecified.

While the 15 condition codes[6] alone lend a lot of utility to the ARM instruction set, the flexibility allowed in setting the CPSR flags (N, Z, C, V) combined with persistence of those flag values (until further modification) makes conditional execution all the more powerful. As a rule:

An instruction can be conditionally executed, based on the CPSR flags set in a previous (already executed) instruction after zero or more intermediate instructions that have not modified the flags.

This implies that unlike instructions of many other processors, most ARM

---

[6] Of the 16 possible values, the code '1111' (binary) is reserved.

instructions have a choice of not disturbing the CPSR flags. By default, but for a few instructions such as CMP (compare) and TST (test), ARM instructions do not modify the flags. In order to affect such modification, the letter 'S' must be suffixed to instruction mnemonics.

Armed with this knowledge of conditional execution, let us move ahead with further code sequences and programming examples.

Data processing

Moving data:

```
a = b;
MOV    r0, r1        ; r0 = r1
```

```
a = b << 5;
MOVS   r0, r1, LSL #5
       ; r0 = r1 << 5
       ; 'S' forces flags update
```

```
a = ~b;
MVN    r0, r1        ; r0 = ~r1
```

Logical operations:

```
a = b | c;
ORR    r0, r1, r2   ; r0 = r1 | r2
```

```
a = b & ~c;
BIC    r0, r1, r2   ; r0 = r1 &~r2
       ; BIC = bit clear
```

Comparators:

```
(a </>/==/<=/>= b);
CMP    r0, r1
; CMP updates CPSR flags based
; on the result of 'r0 – r1'
```

```
if (a > 13)
       a = b << c;
CMP    r0, #13      ; r0 – 13
MOVGT r0, r1, LSL r2
       ; GT: greater than
```

```
(a == b);
TEQ    r0, r1
; TEQ updates CPSR flags based
; on the result of 'r0 ^ r1'
```

```
if (a == 13)
       if (b == 12)
             c = c & 74;
TEQ   r0, #13
TEQEQ r1, #12
ANDEQ r2, r2, #74
```

Memory access

Let us assume the following definitions:

```
int    *pa = &a, *pb = &b;
```

And also that 'pa' and 'pb' are held in r3 and r4 respectively.

Loads:

```
b = *pa;
LDR    r1, [r3]     ; r1 = *r3
```

```
c = *(pb + 1);
LDR    r2, [r4, #4]
       ; r2 = *(r4 + 4)
       ; sizeof(int) = 4
       ; pre-indexing
```

```
c = *pb++;
LDR    r2, [r4], #4;
       ; r2 = *r4, r4 += 4
       ; auto (post) indexing
```

```
c = *++pb;
LDR    r2, [r4, #4]!
       ; r2 = *(r4 + 4), r4 += 4
       ; auto (pre) indexing
       ; '!' to update index r4
```

Stores:

```
*pb = a;
STR    r0, [r4]     ; *r4 = r0
```

```
pa[1] = c;
STR    r2, [r3, #4]
; *(r3+4) = r2
```

```
*pa++ = c;
STR    r2, [r3], #4
       ; *r3 = r2, r3 += 4
```

```
*++pa = c;
```

```
STR    r2, [r3, #4]!
       ; *(r3 + 4) = r2, r3 += 4
```

Branching

Unconditional branch:
```
goto  label_1;
```
```
B      label_1
       ; |label_1 – PC| <= 32MB
```

Function call:
```
foo();
```
```
BL     _foo
; branch and link(L)
; r14 = return addr, goto _foo
```

Conditional branches:
```
if (!a)
       goto label_1;
```
```
TEQ    r0, #0
BEQ    label_1
```

```
if (b & c)
       foo();
```
```
TST    r1, r2
BLNE   _foo
; TST is similar to ANDS, but
; does not modify any register
```

Miscellaneous

No Operation:
```
NOP   ; typically a 'MOV r0, r0'
```

Load an address:
```
pa = &a;
```
```
ADR    r3, a
; 'a' being the label for the
; variable's storage
;
; ADR is a pseudo-instruction
; that translates to a ADD/SUB
; PC/register relative address
```

Swap:
```
c = *ap; *ap = b; b = c;
/* swap '*ap' and 'b' */
```
```
SWP    r1, r1, [r3]
; SWP swaps data between a
; register and a memory location
```

Optimisation techniques

Here on, program fragments in C are listed alongside their equivalent ARM assembly code. Concepts specific to the ARM instruction set and relevant optimization techniques are introduced at appropriate places.

Note:
All ARM assemblers by convention use r13 as the stack pointer. The architecture supports r14 as the link register.

A simple search example:

```c
#include <stdio.h>

int    main(void)
{
       int    a[10] = {7, 6, 4, 5, 5, 1, 3, 2, 9, 8};
       int    i;
       int    s = 4;

       for (i = 0; i < 10; i++)
             if (s == a[i])
                    break;

       if (i >= 10)
             return 1;    /* 1 => not found */
       else
             return 0;    /* 0 => found */
}
```

This C-program hand-translated to assembly, without any optimisation may be similar to the listing given below.

```
          .text
   ; Program labels are terminated by ':' for readability
   ; Stack 'grows' downward, caller saves registers.

1  ; Make 48bytes of space for a[], i and s on the stack

   ADD   r13, r13, #-48     ; r13 is SP
   ; a[]: (sp + 12) .. (sp + 48), i: (sp + 8), s: (sp + 4)

   ; Assume that a run-time library routine initialises a[] and s with
   ; the required values

   MOV   r0, #0             ;
   STR   r0, [r13, #8]      ; for (i = 0; ...)

2  loop_start:
   ; loop entry condition check
   LDR   r0, [r13, #8]      ; load 'i'
   CMP   r0, #10            ; for (...; i < 10; ...)
   BGE   loop_end

3  LDR   r1, [r13, #4]              ; load 's' (already initialised to 4)

   ; get a[i]
   MOV   r2, #4             ; sizeof(a[i])
   MUL   r3, r0, r2         ; r3 = i * 4 (serves as an index in a[i])
   ADD   r3, r3, #12        ; adjust r3 relative to base of a[]
   LDR   r4, [r13, r3]      ; r4 = *(r13 + r3) i.e., a[i]

   TEQ   r1, r4
   BEQ   loop_end           ; if (s == a[i]) break;

   ADD   r0, r0, #1         ; for (...; i++)
4
```

```
         STR    r0, [r13, #8]      ; update i

         B      loop_start         ; next iteration

 5       loop_end:
         LDR    r0, [r13, #8]      ; load 'i'
         CMP    r0, #10
         BGE    return_1           ; if (i >= 10)

         MOV    r0, #0             ; by convention, r0 holds the return-value
         B      prog_end

         return_1:
         MOV    r0, #1

         prog_end:
         ADD    r13, r13, #48      ; pop the function frame off the stack
         MOV    r15, r14           ; load LR into PC (r15) [causing a return]
```

Albeit deliberately under-optimised, this assembly listing gives scope for non-trivial optimisations. But firstly, let us remove some glaring redundancies.

Unnecessary loads/stores:
The variables 'i' and 's' need not be stored on the stack. Registers can be used instead, to server their purpose. With this simple change, we save:
   o 8-bytes on the stack (see grey-shaded #1)
   o 20-bytes (5-instructions) in the program space (see grey-shaded #2-5)
   o 3 of the load/store instructions off the loop, which in the worst-case scenario
      (of the element being searched for being the last one in the array) saves
      them from being executed 9 times (i.e., a minimum of 126-cycles[7])

The compiler is good at tasks such as register allocation. But before we look at a compiler generate code, let us attempt utilising the knowledge we have acquired of the ARM instruction set.

Loop invariants:

'r2' is initialised to '#4' in the loop, but never modified. So, the initialisation can be moved out of the loop, say ahead of 'loop_start'. This, in the worst-case scenario saves it from being executed 9 times (i.e., a minimum of 27-cycles)

Conditional execution:

   o The 'BEQ' ahead of the grey-shaded region #4 can be eliminated by making
      the succeeding 'ADD' and 'B' conditional on 'NE' (saving of 4-bytes and a
      worst-case 45-cycles)
   o The 'BGE' and 'MOV' after the grey-shaded region #5 can be replaced with a
      single 'MOVLT' (saving of 4-bytes)
   o The unconditional branch to 'prog_end' can be removed if the succeeding
      'MOV' instruction is replaced with a 'MOVGE' (saving of 4-bytes)

Let us re-visit the assembly listing after incorporating these optimisations.

---

[7] Four cycles per store and five per load

```
        .text

; Stack 'grows' downward, caller saves registers.

; Make 40bytes of space for a[] on the stack

ADD   r13, r13, #-40
; a[]: (sp + 4) .. (sp + 40)

; Assume that a run-time library routine initialises a[] with
; the required values

MOV   r0, #0              ; for (i = 0; ...)
MOV   r1, #4              ; s = 4

MOV   r2, #4              ; sizeof(a[i])

loop_start:
; loop entry condition check
CMP   r0, #10             ; for (...; i < 10; ...)
BGE   loop_end

; get a[i]
MUL   r3, r0, r2          ; r3 = i * 4 (serves as an index in a[i])
ADD   r3, r3, #4          ; adjust r3 relative to base of a[]
LDR   r4, [r13, r3]       ; r4 = *(r13 + r3) i.e., a[i]

TEQ   r1, r4              ; s == a[i] ?

ADDNE r0, r0, #1          ; for (...; i++) (if 's' not found)
BNE   loop_start          ; next iteration (if 's' not found)

loop_end:
CMP   r0, #10

MOVLT r0, #0              ; if (i < 10) ...
MOVGE r0, #1              ; else

prog_end:
ADD   r13, r13, #40       ; pop the function frame off the stack
MOV   r15, r14            ; load LR into PC (r15) [causing a return]
```

This seems to be as good as it can get. Yet, there is one little trick left – eliminating the multiplication – to be tried out.

Shift to multiply:

We have already seen the second operand of ARM load/store instructions being used for auto-indexing. It can also be used to specify a register with an optional shift as follows:

```
LDR/STR{cond} Rd, [Rn, {-}Rm{,shift}]{!}  ; pre-indexing
-OR-
LDR/STR{cond} Rd, Rn, [{-}Rm{,shift}]     ; post-indexing
```

where *shift* can be one of:

ASR #n   arithmetic shift right n bits; n = [1..32]
LSL #n   logical shift left n bits; n = [0..31]
LSR #n   logical shift right n bits; n = [1..32]
ROR #n   rotate right n bits; n = [1..31]
RRX   rotate right one bit with carry (extend)

The multiplication in the listing can now be replaced with a simple left shift:

```
        .text

; Stack 'grows' downward, caller saves registers.

; Make 40bytes of space for a[] on the stack

ADD   r13, r13, #-40
; a[]: (sp + 4) .. (sp + 40)

; Assume that a run-time library routine initialises a[] with
; the required values

MOV   r0, #0              ; for (i = 0; ...)
MOV   r1, #4              ; s = 4

ADD   r2, r13, #4         ; r2 = &a[0]

loop_start:
; loop entry condition check
CMP   r0, #10             ; for (...; i < 10; ...)
BGE   loop_end

; get a[i]
LDR   r3, [r2, r0, LSL #2]    ; r3 = *(r2 + r0*4) i.e., a[i]

TEQ   r1, r3              ; s == a[i] ?

ADDNE r0, r0, #1          ; for (...; i++) (if 's' not found)
BNE   loop_start          ; next iteration (if 's' not found)

loop_end:
CMP   r0, #10

MOVLT r0, #0             ; if (i < 10) ...
MOVGE r0, #1             ; else

prog_end:
ADD   r13, r13, #40      ; pop the function frame off the stack
MOV   r15, r14          ; load LR into PC (r15) [causing a return]
```

After a string of optimisations, we achieved a program length of 15-words (as compared to the original size of 24-words). We have also been able to significantly reduce the execution time. This is better than the code generated by an optimising

compiler[8] (even after making an allowance for 'a[]' initialization code). This improvement can primarily be attributed to our understanding of the application on hand. For instance, it is difficult to imagine a compiler generate the 'TEQ, ADDNE, BNE' sequence for this program!

Caution! **You cannot replace a compiler:**

It is very tempting to hand code a program in assembly as the returns are rewarding enough, especially in the case of small programs. But any non-trivial program should first be run through a respectable compiler. Further optimisation can then be attempted on the generated assembly code. Not only does this cut the development effort by an order of magnitude but also greatly reduces the chances of defects creeping in due to oversight and weariness that sets on the programmer (on the second sleepless night when the coffee maker runs out of the refill). For, modern compilers incorporate many advanced optimisation techniques and are good at applying them tirelessly, over and over, to large chunks of code.

To explore ARM optmisation further, let us now move on to 'block copy' - an example no ARM programming tutorial can do without:

An optimised bcopy:

```
void  bcopy(char *to, char *from, unsigned int nbytes)
{
      while (nbytes--)
            *to++ = *from++;
}
```

Translation:

```
_bcopy:

; by procedure call convention, the arguments to this function are
; passed in r0, r1, r2

TEQ   r2, #0             ; nbytes == 0 ?
BEQ   bcopy_end

bcopy_start:
SUB   r2, r2, #1         ; nbytes--

; *to++ = *from++
LDRB  r3, [r1], #1       ; LDRB/STRB loads/stores a byte
STRB  r3, [r0], #1       ; auto indexing for post increment (++)

B     bcopy_start        ; next iteration

bcopy_end:
MOV   r15, r14           ; PC = LR i.e., return
```

---

[8] I make this claim after verifying the 'release' mode code generated by two popular compilers for ARM

There seems to be hardly any scope for optimization at the outset. Yet, in a task that involves a condition check, we hardly seem to be using any conditional execution. This gives us a clue for optimisation.

Optimisation-1:

```
_bcopy:

; rewriting '(nbytes--)' as '(--nbytes >= 0)', we get:
SUBS        r2, r2, #1          ; set CPSR flags on --nbytes

LDRPLB      r3, [r1], #1        ; PL: condition code for 'PLus'
STRPLB      r3, [r0], #1        ; 'PLus' stands for 'positive or zero'
BPL         _bcopy              ; next iteration

MOV         r15, r14            ; return
```

We were able to save 2 instructions out of 7 and bettered a compiler, again[9].

Now let us move our focus from size to performance, as a 30% reduction does not really mean much when the original footprint is only 28-bytes.

bcopy's execution profile:

As obvious from the listing, bcopy spends its entire lifetime in a loop. The branch instruction contributes 25% to the size of the loop. More importantly, it takes up 30% of the execution time (5-cycles out of every 17-cycles). This overhead is unacceptable to any non-trivial and performance sensitive application. This understanding drives our further attempts at optimisation.

We apply the popular loop-unrolling technique to reduce the percentage of time taken by the branch instruction as compared to overall loop time.

Optimisation-2:

```
_bcopy:

; For simplicity of illustration, assume 'nbytes' is a multiple of 4

; Unrolling the loop, to copy four bytes per iteration, we get:

SUBS        r2, r2, #4

LDRPLB      r3, [r1], #1        ; copy byte-1
STRPLB      r3, [r0], #1

LDRPLB      r3, [r1], #1        ; copy byte-2
STRPLB      r3, [r0], #1

LDRPLB      r3, [r1], #1        ; copy byte-3
STRPLB      r3, [r0], #1
```

---

[9] Yes. With O2 optimisation for space in 'release' mode!

```
LDRPLB        r3, [r1], #1        ; copy byte-4
STRPLB        r3, [r0], #1

BPL           _bcopy              ; next iteration

MOV           r15, r14            ; return
```

By adding 6 more instructions, we have been able to reduce the share of 'BPL' from 30% to 14% (5 out of 44-cycles). Yet, this gain is highly deceptive. For we could as well have used a load/store-word combination in place of the four load/store-byte instructions, thereby increasing the effective throughput of the original loop without incurring a size/cycle penalty. That way we only need 17-cycles to transfer four bytes (in spite of 'BPL' usurping 30% of the cycles)!

Caution! Each nail needs a different hit[10]

The de-optimisation seen above is due to a blind application of the 'loop unrolling' technique. And such cases are not unique to this technique alone. Each technique needs to be tailored to the task on hand.

Optimisation-3:

```
_bcopy:

; For simplicity of illustration, assume 'nbytes' is a multiple of 16

; Unrolling the loop, to copy four words per iteration, we get:

SUBS        r2, r2, #16

LDRPL r3, [r1], #4      ; copy word-1
STRPL r3, [r0], #4

LDRPL r3, [r1], #4      ; copy word-2
STRPL r3, [r0], #4

LDRPL r3, [r1], #4      ; copy word-3
STRPL r3, [r0], #4

LDRPL r3, [r1], #4      ; copy word-4
STRPL r3, [r0], #4

BPL   _bcopy            ; next iteration

MOV   r15, r14          ; return
```

With this, the throughput has increased to 16bytes per 44-cycle iteration (a gain of 600% as compared to the original 1byte per 17-cycle iteration), with 'BPL' taking 14% of the execution time.

---

[10] An old saying goes 'When you have a hammer in the hand...'

Is further optimisation possible? Ignoring unreasonable options such as unrolling the word-copy loop many more times, there hardly seems to be any technique left that can be used to achieve a significant gain in performance. Well, we will re-visit this example if we find one.

Multiple-load/store instructions – a detour

Consider a subroutine which needs to call other subroutines and also uses all the available general-purpose registers for local use. Assuming a 'callee-saves' protocol, a stack that grows from low to high address and a stack pointer that always pointing to the next free word available, the subroutine's entry and exit code looks similar to this:

```
_foo:                               ; exit code start
                                    ; restore all registers
; entry code start
; save all registers                LDR   r14, [r13, #-4]!
; (r0 – r3 need not be saved        LDR   r12, [r13, #-4]!
; as are for parameter passing)     LDR   r11, [r13, #-4]!
STR   r4, [r13], #4                 LDR   r10, [r13, #-4]!
STR   r5, [r13], #4                 LDR   r9, [r13, #-4]!
STR   r6, [r13], #4                 LDR   r8, [r13, #-4]!
STR   r7, [r13], #4                 LDR   r7, [r13, #-4]!
STR   r8, [r13], #4                 LDR   r6, [r13, #-4]!
STR   r9, [r13], #4                 LDR   r5, [r13, #-4]!
STR   r10, [r13], #4                LDR   r4, [r13, #-4]!
STR   r11, [r13], #4
STR   r12, [r13], #4                ; exit code ends
STR   r14, [r13], #4
; entry code ends                   MOV   r15, r14     ; return
```

To a non-ARM RISC programmer, this listing is familiar and normal. For, each and every instruction is very much relevant and essential to the task on hand. Only, an ARM programmer would simply have written this *equivalent* code:

```
_foo:

STMEA      r13!, {r4 – r12, r14}   ; entry code

;
; body of _foo
;

LDMEA      r13!, {r4 – r12, r15}   ; exit code
```

There is no mysterious magic here. Any non-trivial programming task, whether driven by structured, object oriented or functional design approach, involves subroutines. And most programming languages use a stack based solution for maintaining the activation of records of called (currently active thread) subroutines. Given the limited number of registers on any processor, saving and restoring registers across subroutine calls is inevitable. The special non-RISC-like ARM

instructions seen above (LDM&STM) are an explicit acknowledgement from the ARM architecture of the frequency and importance of such multiple-register load/store activity. These instructions can be used multiple ways such as:

STMEA & LDMEA: push(store)/pop(load) multiple-registers to/from an 'empty ascending' stack i.e., the stack pointer points to the next free word on the stack as it grows from low to high address. The suffix 'EA' can be replaced with a more generic mnemonic 'IA' (**i**ncrement **a**fter). E.g., STMEA r13!, {r3-r5, r11, LR}

STMFD & LDMFD: By replacing the suffix 'EA' with 'FD' you get a 'full descending' stack which is exactly opposite in behaviour to an 'EA' stack. 'FD' has a semantically equivalent name 'DB' which stands for 'decrement before'. E.g., LDMDB r0!, {r1-r10}

Other combinations such as 'IB' and 'DA' are also possible.

Obviously, these instructions cannot be taking the same number of cycles as an LDR or an STR. The real gains are in program space, reduced chances of making coding mistakes and enhanced readability.

Optimisation-4:

As you might have guessed by now, the time has come to rejoin the main road and revisit the bcopy example. The previous throughput of 16bytes per iteration can now be achieved on a smaller footprint by replacing the four pairs of LDRPL/STRPL with a single LDMPL/STMPL combination such as:

```
LDMPL r1!, {r3 – r6}     ; load r3-r6 from [r1], advance r1 by 16 bytes
STMPL r0!, {r3 – r6}     ; store r3-r6 starting at [r0], advance r0
```

The blue print for a high throughput (close to 40byte) bcopy is as follows:

```
_bcopy:

; For every iteration, attempt to transfer 40bytes. The modulus value
; remaining (remainder of the division of 'nbytes' by 40) should be
; treated as a separate lesser throughput bcopy loop. Shown here is
; only the main 40byte throughput loop:

; Unrolling the loop, to copy ten words per iteration, we get:
SUBS        r2, r2, #40

BMI         copy_rest          ; 'MI' stands for minus/negative

LDMPL       r1!, {r3 – r12}    ; load r3-r12 from [r1]
STMPL       r0!, {r3 – r12}    ; store the loaded words at [r0]

BPL   _bcopy                   ; next iteration

; copy the residual bytes
copy_rest:
; completed copying

MOV   r15, r14                 ; return
```

By saving r13 and r14 on the stack before the start of the loop, the throughput can be pushed closer to the maximum possible 48bytes per iteration. This is an example of how two of ARM's best features - conditional execution and multiple load-stores put together[11] improve the program characteristics by an order of magnitude.

Thumb – the final footprint solution

Before calling it a day, a brief[12] overview of the 16-bit Thumb instruction set would be appropriate. All the ARM listings seen earlier used the normal 32-bit ARM instructions. However, many embedded systems designers show an inclination to trade performance[13] for footprint – provided there is an option. Not until the advent of ARM did a single processor offer a solution of simultaneously executing both 32-bit and 16-bit code with such little overhead (in terms of additional silicon, programming complexity and cost).

The Thumb is a 16-bit instruction set architecture that is functionally complete but relatively restricted in variety as compared to that of the regular 32-bit ARM instruction set. Notable differences include 2-address format, unconditional updation of CPSR flags for (almost) all instructions and less flexible 'second operand'. The Thumb architecture is cleanly implemented in silicon by way of including an on-the-fly instruction de-compressor functional unit in the processor that translates 16-bit ARM instructions into their 32-bit equivalents that are understood by the rest of the processor. It must be noted though that it is only the instruction length that is halved and not the register sizes themselves. As a side effect, the number of usually visible registers is reduced by five[14].

The programmer is not constrained to use a single instruction set throughout his program. Complete freedom is given to freely intermix 32-bit and 16-bit code and switch between them using the BX (branch and exchange) instruction. Whenever the Thumb is found to be inadequate and restrictive for a particular functionality / module / computation, the ARM instruction set can be used as a special case (or the other way around, if at certain places the power of 32-bit seems an overkill[15]). It is this flexibility which makes the ARM processor a very attractive option for an embedded systems designer / programmer.

Conclusion

This paper made an attempt at introducing the ARM architecture to an embedded systems designer / programmer by providing an overview of its functional units and instruction set. ARM assembly optimisation techniques were introduced along with a couple of examples in a graded exercise like fashion. This being targeted at those who are new to the architecture, most fine grain details were left out of this paper for fear of losing reader interest. However, the techniques presented herein should

---

[11] Chess enthusiasts can liken this to an active Queen-Knight combination
[12] Brief, not because its instruction length being only half as long as its more powerful 32-bit cousin, but because of it requiring a completely dedicated tutorial to do justice.
[13] Even in performance critical cases such as an RTOS, the emphasis is usually on predictability and bounded response times rather than on searing speed.
[14] r0-r7 known as 'low' registers are always visible while the high 'r8-r12' are visible only in certain instructions in restricted formats.
[15] If the reader wondered why there was a switch between a 2-column and single column mode, the answer should now be evident ; ^)

be sufficient enough to venture into serious software design and programming with the ARM processor(s). The references provided towards the end of this paper can be used to further hone ARM assembly programming skills.

References

- Dave Jagger (editor), *ARM Architecture Reference Manual,* Prentice Hall
- Steve B. Furber, *ARM System Architecture,* Addison-Wesley, 1996, ISBN 0-201-40352-8
- *http://www.arm.com/* - data sheets, instruction set summaries, white papers, application notes and architecture references for the ARM family of processors
- Rupesh W. Kumbhare, *Optimization Techniques for DSP based Applications* – A highly detailed paper on optimisation techniques for the TMS320C54x family of DSPs. It describes with code-samples, concepts that are applicable to non-DSP programming as well. Interested readers are can reach the author (rupesh.kumbhare@wipro.com) for a copy of the same

---