

Scott R. Armstrong
sarmstr2
CS433 Project – Final Submission
Due May 8, 2007

1. Project Description

This project describes the performance improvements of a brute-force prime number calculator running in the eCos ARM execution environment. I evaluated several approaches to improve the performance of this application, including register renaming to replace indexed references to variables, branch alternatives, predicated instructions and loop unrolling.

This program offered some interesting challenges because of the irregularity of prime numbers occurring within a sequential set of numbers and the subsequent difficulty in taking advantage of repeating patterns in the execution of the code.

One of the more evident opportunities for performance improvements was the use of separate registers to hold global variables instead of repeated loads and stores to and from an indexed frame pointer. Initially, each use and modification of a variable required a load instruction, an arithmetic instruction (the work) and a store instruction. The relatively small number of variables in this program allowed the global assignment of registers, replacing frame pointer indices.

Another effective opportunity for performance improvements required some fairly significant changes to the branch structure, making use of predicated instructions to further reduce the number and complexity of branches from the original two versions of the assembly code.

Though I did unroll the innermost loop in one version of the prime number calculator, this program didn't afford performance improvement opportunities as described during our lessons because each of the duplicated loop bodies was not likely to take the same amount of time to execute and there were no loads within the loop that could be handled as multiple loads.

In order to look more closely at the program logic, I removed all I/O instructions. The most recent prime numbers is stored in register r8, allowing us to confirm that this program functions correctly. Additionally, to avoid the challenges of linking functions in the eCos environment, I inserted simple Divide code in place of the Modulo function call and checked for the existence of a remainder to determine whether a number was a non-prime number.

I also replaced more complicated Divide instructions with a simple shift instruction to get an integer value of $y/2$, effectively producing $y \text{ DIV } 2$. This program doesn't need to use fractions in this portion of the logic, so this change eliminated several instructions.

2. Summary of Performance Enhancements

The initial version of the program (described in previous deliverables for this project, also available via a Web page link at the bottom of section 3) included load and store instructions to frame pointer indices that were unrelated to I/O, program setup or program termination. All load and store instructions except for the program setup and termination instructions were removed from the final version of this program by using dedicated registers for variables and by renaming all registers except for the scratch registers r0 through r3.

The intermediate version of this program included 13 branch labels (two branch labels were excess) and 11 branch statements. The final, restructured version of this program (Appendix A) included four branch labels and four branch statements (three conditional branch statements and 1 unconditional branch statement), excluding the Modulo code inserted to replace the MOD function call. The use of three predicated, non-branch instructions made it possible to eliminate several branches, and in one case, a single compare instruction was used to set the appropriate bits for use by both a predicated instruction and a conditional branch instruction.

I wanted to unroll the innermost loop (Appendix B) to look for opportunities for performance improvements similar to those described in the lectures, but this program didn't lend itself to instruction reordering within unrolled loops. Features of the ARM environment allow for multiple loads, but this program as refined doesn't need to load from memory in the main part of the loop. The unrolled loop, though, did expose some obvious patterns of processing, namely the set of non-prime numbers that are divisible by two or five. By unrolling the loop and checking prime-ness for numbers ending in 1, 3, 7 and 9 (for numbers greater than 10), six unnecessary iterations are avoided out of every ten. Since the more interesting computation involves numbers as they get larger, I didn't create a preamble to the unrolled loop to test the numbers between 2 and 10. Though the code size of the program with the unrolled loop is quite a bit larger, this version will confirm the prime-ness of more numbers faster than any of the previous versions.

3. Summary:

Though the processing done by this brute-force prime number calculator does not transform large amounts of data in a regular fashion, fairly significant performance gains can be realized using a coordinated approach in taking advantage of language-specific performance features. Register renaming to remove unnecessary load and store instructions, careful attention to program branches and the use of predicated instructions to eliminate unneeded branches simplified this program significantly and resulted in generous performance gains.

Additionally, by unrolling the main processing loop, it was possible to analyze and modify the program logic, and remove six unnecessary loops from every set of ten loops. Though it wasn't possible to take advantage of the "unroll-and-jam" technique described in the lessons, this approach also resulted in the savings of one branch statement per unrolled loop. The cost of unrolling the loop, though, is an increase in the size of the program.

Digital copies of all project deliverables as well as the code versions used for each deliverable are available at <http://www.sneetches.net/scott/cs433/>.

Appendix A.

Program listing for ARM assembly code of the brute-force prime number calculator (the original source code and initial assembly code are detailed in the previous deliverables for this project).

```
#####
@
@ Scott R. Armstrong
@ UIUC CS433 Spring 2007
@ ASM Performance Improvement Project
@ Due by May 8, 2007
@
#####
        .file    "primes.c"
        .section .rodata
        .text
        .align  2
        .global main
        .type   main, %function
main:
        @ args = 0, pretend = 0, frame = 28
        @ frame_needed = 1, uses_anonymous_args = 0
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #28

        @@ set up z (limit of numbers to check for prime-ness)
        mov     r3, #99328      @compiler split z into 99328+672
        add     r3, r3, #672    @uses shifted values to fit IMM field
                                @ or load a different IMM into r6
                                @ assigned here to replace I/O for
                                @ the purposes of this project
        mov     r6, r3          @@ move value into z
        mov     r5, #2          @@ y, initialized to 2

.L1:                                @@ y-loop-start
        cmp     r5, r6
        bgt     .L1_END
        mov     r7, #1          @@ prime, initialized to true
        mov     r4, #2          @@ x, initialized to 2

.L2:                                @@ x-loop-start
        mov     r2, r5, asr #1  @@ y/2 (y DIV 2, no need for fractions)
        cmp     r4, r2          @@ Compare r2 (y/2) with x
        bgt     .L2_END

@ DIV/MOD function - based on ARM Developer Suite 1.1 Divide Function
@ used here instead of linking a MOD function

        mov     r9, r4
        mov     r10, r5         @ r10 will contain any remainder
.L_DIV:
        mov     r2, r9
        cmp     r2, r10, lsr #1
.L_DIV_1:
```

```
        movls    r2, r2, lsl #1@
        cmp     r2, r10, lsr #1
        bls    .L_DIV_1
        mov     r0, #0           @ Initialize quotient
.L_DIV_2:
        cmp     r10, r2
        subcs   r10, r10, r2
        adc     r0, r0, r0
        mov     r2, r2, lsr #1
        cmp     r2, r9
        bhs    .L_DIV_2

@ End DIV/MOD function

        cmp     r10, #0           @ if no remainder
        moveq   r7, #0           @ set prime to false
        cmp     r7, #1
        addeq   r4, r4, #1       @ if prime, increment x
        beq    .L2              @ ...and branch to L2 (repeat)

.L2_END:
        cmp     r7, #1
        moveq   r8, r5           @ store last prime number in r8
        add    r5, r5, #1
        b      .L1

.L1_END:
        sub     sp, fp, #12
        ldmfd  sp, {fp, sp, pc}
        .align 2
        .size  main, .-main
        .ident "GCC: (GNU) 4.1.0"

.end
```

Appendix B.

Program listing of the ARM assembly code of the brute-force prime number calculator with an unrolled inner loop and always-nonprime numbers excluded from being checked.

```
#####
@
@ Scott R. Armstrong
@ UIUC CS433 Spring 2007
@ ASM Performance Improvement Project
@ Due by May 8, 2007
@
#####
.file "primes.c"
.section .rodata
.text
.align 2
.global main
.type main, %function
main:
@ args = 0, pretend = 0, frame = 28
@ frame_needed = 1, uses_anonymous_args = 0
mov ip, sp
stmfd sp!, {fp, ip, lr, pc}
sub fp, ip, #4
sub sp, sp, #28

@@ set up z (limit of numbers to check for prime-ness)
mov r3, #99328 @ compiler split z into 99328+672
add r3, r3, #672 @ uses shifted values to fit IMM field
@ or load a different IMM into r6
@ assigned here to replace I/O for
@ the purposes of this project
mov r6, r3 @ move value into z
mov r5, #10 @ y, initialized to 10
@ to demonstrate loop unrolling

.L1: @@ y-loop-start
cmp r5, r6
bgt .L1_END

@@ Start of unrolled-loops section
@@ n+1 loop

add r5, r5, #1 @ n + 1

mov r7, #1 @@ prime, initialized to true
mov r4, #2 @@ x, initialized to 2

.L2_n1: @@ x-loop-start
mov r2, r5, asr #1 @@ y/2 (y DIV 2, no need for fractions)
cmp r4, r2 @@ Compare r2 (y/2) with x
bgt .L2_END_n1

@ DIV/MOD function - based on ARM Developer Suite 1.1 Divide Function
@ used here instead of linking a MOD function
```

```

        mov     r9, r4
        mov     r10, r5           @ r10 will contain any remainder
.L_DIV_n1:
        mov     r2, r9
        cmp     r2, r10, lsr #1
.L_DIV_1_n1:
        movls   r2, r2, lsl #1@
        cmp     r2, r10, lsr #1
        bls     .L_DIV_1_n1
        mov     r0, #0           @ Initialize quotient
.L_DIV_2_n1:
        cmp     r10, r2
        subcs   r10, r10, r2
        adc     r0, r0, r0
        mov     r2, r2, lsr #1
        cmp     r2, r9
        bhs     .L_DIV_2_n1

@ End DIV/MOD function

        cmp     r10, #0           @ if no remainder
        moveq   r7, #0           @ set prime to false
        cmp     r7, #1
        addeq   r4, r4, #1       @ if prime, increment x
        beq     .L2_n1           @ ...and branch to L2 (repeat)

.L2_END_n1:
        cmp     r7, #1
        moveq   r8, r5           @ store last prime number in r8

@@ n+3 loop

        add     r5, r5, #2       @ n + 3

        mov     r7, #1           @@ prime, initialized to true
        mov     r4, #2           @@ x, initialized to 2

.L2_n3:
                                @@ x-loop-start
        mov     r2, r5, asr #1   @@ y/2 (y DIV 2, no need for fractions)
        cmp     r4, r2           @@ Compare r2 (y/2) with x
        bgt     .L2_END_n3

@ DIV/MOD function - based on ARM Developer Suite 1.1 Divide Function
@ used here instead of linking a MOD function

        mov     r9, r4
        mov     r10, r5         @ r10 will contain any remainder
.L_DIV_n3:
        mov     r2, r9
        cmp     r2, r10, lsr #1
.L_DIV_1_n3:
        movls   r2, r2, lsl #1@
        cmp     r2, r10, lsr #1
        bls     .L_DIV_1_n3
        mov     r0, #0         @ Initialize quotient
.L_DIV_2_n3:

```

```

        cmp     r10, r2
        subcs  r10, r10, r2
        adc   r0, r0, r0
        mov   r2, r2, lsr #1
        cmp   r2, r9
        bhs   .L_DIV_2_n3

@ End DIV/MOD function

        cmp     r10, #0           @ if no remainder
        moveq  r7, #0           @ set prime to false
        cmp     r7, #1
        addeq  r4, r4, #1       @ if prime, increment x
        beq    .L2_n3          @ ...and branch to L2 (repeat)

.L2_END_n3:
        cmp     r7, #1
        moveq  r8, r5           @ store last prime number in r8

@@ n+7 loop

        add    r5, r5, #4       @ n + 7

        mov    r7, #1           @@ prime, initialized to true
        mov    r4, #2           @@ x, initialized to 2

.L2_n7:
                                @@ x-loop-start
        mov    r2, r5, asr #1   @@ y/2 (y DIV 2, no need for fractions)
        cmp    r4, r2           @@ Compare r2 (y/2) with x
        bgt    .L2_END_n7

@ DIV/MOD function - based on ARM Developer Suite 1.1 Divide Function
@ used here instead of linking a MOD function

        mov    r9, r4
        mov    r10, r5         @ r10 will contain any remainder
.L_DIV_n7:
        mov    r2, r9
        cmp    r2, r10, lsr #1
.L_DIV_1_n7:
        movls  r2, r2, lsl #1@
        cmp    r2, r10, lsr #1
        bls   .L_DIV_1_n7
        mov    r0, #0         @ Initialize quotient
.L_DIV_2_n7:
        cmp    r10, r2
        subcs  r10, r10, r2
        adc   r0, r0, r0
        mov   r2, r2, lsr #1
        cmp   r2, r9
        bhs   .L_DIV_2_n7

@ End DIV/MOD function

        cmp     r10, #0           @ if no remainder
        moveq  r7, #0           @ set prime to false
        cmp     r7, #1

```

```

        addeq    r4, r4, #1      @ if prime, increment x
        beq     .L2_n7         @ ...and branch to L2 (repeat)

.L2_END_n7:
        cmp     r7, #1
        moveq   r8, r5         @ store last prime number in r8

@@ n+9 loop
        add     r5, r5, #2      @ n + 9

        mov     r7, #1         @@ prime, initialized to true
        mov     r4, #2         @@ x, initialized to 2

.L2_n9:                                @@ x-loop-start
        mov     r2, r5, asr #1  @@ y/2 (y DIV 2, no need for fractions)
        cmp     r4, r2         @@ Compare r2 (y/2) with x
        bgt     .L2_END_n9

@ DIV/MOD function - based on ARM Developer Suite 1.1 Divide Function
@ used here instead of linking a MOD function

        mov     r9, r4
        mov     r10, r5        @ r10 will contain any remainder
.L_DIV_n9:
        mov     r2, r9
        cmp     r2, r10, lsr #1
.L_DIV_1_n9:
        movls   r2, r2, lsl #1@
        cmp     r2, r10, lsr #1
        bls     .L_DIV_1_n9
        mov     r0, #0         @ Initialize quotient
.L_DIV_2_n9:
        cmp     r10, r2
        subcs   r10, r10, r2
        adc     r0, r0, r0
        mov     r2, r2, lsr #1
        cmp     r2, r9
        bhs     .L_DIV_2_n9

@ End DIV/MOD function

        cmp     r10, #0        @ if no remainder
        moveq   r7, #0         @ set prime to false
        cmp     r7, #1
        addeq   r4, r4, #1     @ if prime, increment x
        beq     .L2_n9         @ ...and branch to L2 (repeat)

.L2_END_n9:
        cmp     r7, #1
        moveq   r8, r5         @ store last prime number in r8

@@ End of unrolled-loops section

        add     r5, r5, #1     @ add 1 to y, r5 adds total 10
        b       .L1

.L1_END:

```

```
sub    sp, fp, #12
ldmfd sp, {fp, sp, pc}
.align 2
.size  main, .-main
.ident "GCC: (GNU) 4.1.0"

.end
```