

Scott R. Armstrong
sarmstr2
CS433
ILP Project
Functional Performance Improvement

1. Project Progress

After generating and reviewing the ARM assembly code for a brute-force prime number calculator from the original C++ code, I decided to rewrite the program in C (Appendix A) and remove most of the I/O instructions to simplify the generated assembly code (Appendix B). This would make it easier to focus more on the core functionality of the program. The performance improvements to the ARM assembly code (Appendix C) so far are limited to replacing location references with dedicated registers within the program to reduce or eliminate load instructions within the loop iterations. I will probably remove the rest of the I/O instructions for the final version and I will still need to link the library for the Modulo operation at the heart of the program. I was able to run the modified instructions in the UIUC eCos environment by removing the I/O and the unlinked modulo operation to confirm that the use of registers resulted in predictable program behavior.

2. Initial Performance Enhancements

This program consists of a while() loop within a for() loop within another while() loop. Instead of prompting the user for an upper limit of the numbers to check for prime-ness, I created variable z and initialized its value to 100000 to make sure that the loops would iterate many times.

The innermost loop (labels L7 and L8) checks for a zero Modulo return value for variable y as x goes from 2 to $y/2$. If the return value is not zero, x is incremented and the loop runs again (label L8). If the return value is zero, the number is determined to be non-prime and control is given back to the for() loop (label L7). This innermost loop can iterate as many as $y/2$ times, so as y increases, so does the potential number of iterations. This loop is inside the for() loop (label L18), which itself can iterate as many as y times. The outer loop was originally written to run the prime number calculator multiple times or exit the program—this feature has been disabled in the C version of the code with minimal I/O.

The gcc-generated ARM assembly code includes up to 3 load instructions (depending on the branch) and 1 store instruction using the frame pointer (fp, also r11) offsets within the innermost loop, and two (regardless of the branch) loads using the frame pointer within the middle loop. Throughout the rest of the program, only the scratch registers (r0 – r3) were used to store values. All variables were accessed using frame pointer offset references.

Modifications to any variable's value would require loading the value from the referenced location into a scratch register, and any updates to a variable would involve a store to the

referenced location. Another instruction is required to do the actual work using the value in the scratch register or registers.

Assuming that loads require more clock cycles than moves, I decided to reduce the number of loads, especially those within the innermost and middle loops. Because of the small number of variables in the program (x, y, z, prime, repeat, true, false), I was able to dedicate a register to each of the variables (r4, r5, r6, r7, r8, for x, y, z, prime, and repeat, respectively). For the purposes of this project, I didn't dedicate registers for true and false—the true or false values in the prime and repeat variables are manually assigned using the appropriate move-immediate instruction. This isn't ideal and would complicate the debugging process if the wrong immediate value was used in the code, but I used this method only because of the relatively small program size.

By using dedicated registers instead of frame pointer offsets for the variables, I was able to replace each of the load instructions and subsequent processing and store instructions (usually groups of two or three instructions) in the loops with just the needed instruction. For example, the innermost loop now has no load instructions, and label L8 consists of a single add instruction to increment x.

3. Notes on inline comments:

Inline comments with no space following the '@' character are for description and reference. Inline comments in the modified ARM assembly code file with a space following the '@' character indicate a manually modified portion of the code.

4. Remaining efforts:

I still plan to explore the possibility of finding performance gains by unrolling the loops, but with most of the load instructions replaced by simpler move instructions, this approach may not prove to be as effective as I thought earlier.

I have noticed that I might be able to eliminate some of the branches, especially those that now execute a single instruction and fall through to the next label in the program.

I still plan to compare similar performance modifications to the PowerPC assembly code for this program. My familiarity with the ARM assembly code should help me pick up some of the syntax differences between ARM and PowerPC assembly code, but I'll need to look closely at the PowerPC instruction set for other variations of performance modifications.

Appendix A.

```
// *****  
// Scott R. Armstrong  
// June, 2003 and April 2007  
//  
// No error checking is included to avoid errors caused by  
// non-numeric or negative input when prompted for a value.  
//  
// Programming environment: Borland C++, Version 5.5, Win32  
//  
// Rewritten in C for assembly language project  
// UIUC CS433, Spring 2007  
//  
// Much of the I/O has been removed to simplify the assembly  
// code generated by the gcc compiler.  
//  
// *****  
  
#include <stdio.h>          // I/O preprocessor header file  
#include <math.h>          // Arithmetic  
  
int main()  
{  
  
// declaration of variables  
//   bool    prime, repeat;  
//   char    ch;  
   int      x, y, z, prime, repeat, true, false;  
  
       z = 100000;        // initialized for now - request user input  
  
       true = 1;  
       false = 0;  
  
   repeat = true;        // preset the boolean value  
   while ( repeat )  
   {  
       printf("\n");  
       printf("Prime Number Calculator\n");  
       printf("=====\n");  
       //printf("Enter a positive integer:  "\n);  
  
       // input z here  
  
       for ( y = 2; y <= z; y++)  
       {  
           prime = 1;  
           x = 2;  
           while ( (x <= y/2 ) && (prime == true) )  
           {  
  
               if ((y % x) == 0 )  
               {  
                   prime = false;  
               }  
           }  
       }  
   }  
}
```

```
        else
        {
            x++;
        }
    }

    if (prime == true)
    {
        printf("%d", y);
        printf(" is a prime number.\n");
    }
}

printf("\n");
printf("Do you want to continue (Y or N)? ");
//      repeat logic here
      repeat = false; // temp

    if ( !repeat )
        printf("\nQuitting the Prime Number Calculator\n");

} // end of while loop

return (0);
}
```

Appendix B.

Program listing for ARM assembly code of the program in Appendix A.

```
.file "primes.c"
.section .rodata
.align 2

.LC0:
.ascii "Prime Number Calculator\000"
.align 2

.LC1:
.ascii "=====\000"
.global __modsi3
.align 2

.LC2:
.ascii "%d\000"
.align 2

.LC3:
.ascii " is a prime number.\000"
.align 2

.LC4:
.ascii "Do you want to continue (Y or N)? \000"
.align 2

.LC5:
.ascii "\012Quitting the Prime Number Calculator\000"
.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
    mov    r3, #99328
    add    r3, r3, #672
    str    r3, [fp, #-32]
    mov    r3, #1
    str    r3, [fp, #-20]
    mov    r3, #0
    str    r3, [fp, #-16]
    ldr    r3, [fp, #-20]
    str    r3, [fp, #-24]
    b     .L19

.L3:
    mov    r0, #10
    bl    putchar
    ldr    r0, .L20
    bl    puts
    ldr    r0, .L20+4
    bl    puts
    mov    r3, #2
    str    r3, [fp, #-36]
    b     .L4
```

```
.L5:
    mov     r3, #1
    str     r3, [fp, #-28]
    mov     r3, #2
    str     r3, [fp, #-40]
    b       .L18

.L7:
    ldr     r3, [fp, #-36]
    mov     r0, r3
    ldr     r1, [fp, #-40]
    bl     __modsi3
    mov     r3, r0
    cmp     r3, #0
    bne     .L8
    ldr     r3, [fp, #-16]
    str     r3, [fp, #-28]
    b       .L6

.L8:
    ldr     r3, [fp, #-40]
    add     r3, r3, #1
    str     r3, [fp, #-40]

.L6:
.L18:
    ldr     r2, [fp, #-36]
    mov     r3, r2, lsr #31
    add     r3, r3, r2
    mov     r3, r3, asr #1
    mov     r2, r3
    ldr     r3, [fp, #-40]
    cmp     r2, r3
    blt     .L10
    ldr     r2, [fp, #-28]
    ldr     r3, [fp, #-20]
    cmp     r2, r3
    beq     .L7

.L10:
    ldr     r2, [fp, #-28]
    ldr     r3, [fp, #-20]
    cmp     r2, r3
    bne     .L12
    ldr     r0, .L20+8
    ldr     r1, [fp, #-36]
    bl     printf
    ldr     r0, .L20+12
    bl     puts

.L12:
    ldr     r3, [fp, #-36]
    add     r3, r3, #1
    str     r3, [fp, #-36]

.L4:
    ldr     r2, [fp, #-36]
    ldr     r3, [fp, #-32]
    cmp     r2, r3
    ble     .L5
    mov     r0, #10
    bl     putchar
    ldr     r0, .L20+16
```

```
    bl    printf
    ldr   r3, [fp, #-16]
    str   r3, [fp, #-24]
    ldr   r3, [fp, #-24]
    cmp   r3, #0
    bne   .L2
    ldr   r0, .L20+20
    bl    puts
.L2:
.L19:
    ldr   r3, [fp, #-24]
    cmp   r3, #0
    bne   .L3
    mov   r3, #0
    mov   r0, r3
    sub   sp, fp, #12
    ldmfd sp, {fp, sp, pc}
.L21:
    .align 2
.L20:
    .word .LC0
    .word .LC1
    .word .LC2
    .word .LC3
    .word .LC4
    .word .LC5
    .size main, .-main
    .ident "GCC: (GNU) 4.1.0"
```

Appendix C.

Program listing of the ARM assembly program with performance modifications as described earlier.

```
.file "primes.c"
.section .rodata
.align 2

.LC0:
.ascii "Prime Number Calculator\000"
.align 2

.LC1:
.ascii "=====\000"
.global __modsi3
.align 2

.LC2:
.ascii "%d\000"
.align 2

.LC3:
.ascii " is a prime number.\000"
.align 2

.LC4:
.ascii "Do you want to continue (Y or N)? \000"
.align 2

.LC5:
.ascii "\012Quitting the Prime Number Calculator\000"
.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
    mov r3, #99328 @compiler split z into 99328+672
    add r3, r3, #672 @uses shifted values to fit IMM field
    mov r6, r3 @@ move value into z
    mov r3, #1 @manually move 1 to a register for True?
    str r3, [fp, #-20] @fp-20 is value of true
    mov r3, #0
    str r3, [fp, #-16] @fp-16 is value of false
    mov r8, #1 @@ initialize repeat with true
    b .L19 @.L19 starts the while(repeat) loop

.L3:
    mov r0, #10
    bl putchar
    ldr r0, .L20
    bl puts
    ldr r0, .L20+4
    bl puts
    mov r5, #2 @@ y, initialized to 2
    b .L4

.L5:
```

```

    mov     r7, #1          @@ prime, initialized to true
    mov     r4, #2          @@ x, initialized to 2
    b       .L18

.L7:      @@ convert load-stores to Register moves
    mov     r0, r5          @set up r0 with y
    mov     r1, r4          @set up r1 with x (for y mod x)
    bl     __modsi3         @Modulo function, need to link
    mov     r3, r0          @r0 contains results of (y mod x)
    cmp     r3, #0
    bne    .L8
    mov     r7, #0          @@ prime gets false
    b       .L6

.L8:      @x++
    add     r4, r4, #1      @@ x++ (removed load and store)

.L6:
.L18:     @@ remove loads
    mov     r3, r5, lsr #31 @@ y
    add     r3, r3, r2
    mov     r3, r3, asr #1
    mov     r2, r3
    cmp     r2, r4          @@ Compare r2 with x
    blt    .L10
    ldr     r3, [fp, #-20]
    cmp     r7, r3          @@ Compare prime with true
    beq    .L7

.L10:    @@ remove loads
    cmp     r7, #1          @@ compare prime with true
    bne    .L12
    ldr     r0, .L20+8
    mov     r1, r5          @@ setup y in r1 for printf
    bl     printf
    ldr     r0, .L20+12
    bl     puts

.L12:
.L4:      add     r5, r5, #1  @@ increment y
    @@ remove loads
    cmp     r5, r6          @@ compare y with z
    ble    .L5
    mov     r0, #10
    bl     putchar
    ldr     r0, .L20+16
    bl     printf
    ldr     r3, [fp, #-16]
    str     r3, [fp, #-24]
    ldr     r3, [fp, #-24]
    cmp     r3, #0
    bne    .L2
    ldr     r0, .L20+20
    bl     puts

.L2:
.L19:    cmp     r8, #0          @ compare repeat and false, remove load
    bne    .L3
    mov     r3, #0
    mov     r0, r3
    sub     sp, fp, #12
    ldmfd  sp, {fp, sp, pc}

```

```
.L21:
      .align 2
.L20:
      .word  .LC0
      .word  .LC1
      .word  .LC2
      .word  .LC3
      .word  .LC4
      .word  .LC5
      .size  main, .-main
      .ident "GCC: (GNU) 4.1.0"
```