Lab 1

Link for Demo: https://drive.google.com/file/d/1okRNbf4ZVbNO28t4aHiBsXWM_rYctE5x

Group 4

Andrew Branum, Cade Garrett, Ankit Kanotra, Shawn Witt, Nathan Zabloudil

CSE 525

Abstract

The purpose of this lab was to gain exposure to integral commands and directives that will allow implementation of necessary functionalities in this lab and future labs. This project was also intended to convey the relationship between C and its corresponding assembly code. In this project, the ARM Cortex A53 processor was utilized via a Raspberry Pi. Much of this lab involved conducting research and examining the ArmV6 Architecture Reference Manual. We were successful in all of our endeavors, as our demo calculator worked successfully, and we were able to learn every directive.

Body

The platform used for this project was a Raspberry Pi, which resulted in quite a few issues on our end. Namely, the fact that our model ran incredibly slow, and this would frequently result in various crashes, making progress quite slow. The solution thought up to solve these problems was to emulate a Raspberry Pi via a Virtual Machine and manipulate and test the code that way. This was quite successful and mitigated the issues of using our model. This did result in issues with our demo, but one of our group members had their own Raspberry Pi we used for demonstration.

To begin, the group inserted the initially created assembly code into our text editor of choice (Geanie, the option preloaded onto our Raspberry Pi). Team 4 then ran and exported c files out of the created assembly files, and began to do research on the various directives as we went. E provided students a helpful user manual for specific calls across assembly, which was used to research each call and find their purpose.

The demo was a challenge, as there is no built in function in this particular set of ARM instructions to divide with remainder. So new functions, one to get the quotient (without decimals) and one to get the remainder, were created from pre-existing assembly functions to recreate division. The addition, subtraction, and multiplication functions were all very easy to implement, as all that needs to be done is make sure the assembly functions for ADD, SUB, and MUL can be called by C by creating a “shell” subroutine for them, and that the results are stored in r0 so they can return to the C function that called them.

To create a division function, the divisor was subtracted (or added depending on the combination of negative and positive numbers) repeatedly from the dividend until the divisor could no longer subtract/add fully from it. Each time a subtraction or addition was successful, a counter (which begins at 0) increased. This counter value is what was stored in r0 for return for the division function. For the modulo/remainder function, it operates the exact same except it returns the last result recorded before the divisor could no longer “fit” in the dividend. For example we’ll use 5/2: 5-2=3, 3-2=**1**, 1-2=-1. Because 1-2 was unsuccessful (it’s negative), the last successful result (1) must be the remainder.

Source Code (Software)

P1-1.s

```
.arch armv6                     //sets architecture to armv6
        .eabi_attribute 28, 1   //sets EABI object attributes
        .eabi_attribute 20, 1   //EABI is when the processor boots to load an application with no
                                  intermediate kernel (from Google)
        .eabi_attribute 21, 1   //Assigns 1 to EABI attribute 21
        .eabi_attribute 23, 3   //Assigns 3 to EABI attribute 23
        .eabi_attribute 24, 1   //Assigns 1 to EABI attribute 24
        .eabi_attribute 25, 1   //Assigns 1 to EABI attribute 25
        .eabi_attribute 26, 2   //Assigns 2 to EABI attribute 26
        .eabi_attribute 30, 6   //Assigns 6 to EABI attribute 30
        .eabi_attribute 34, 1   //Assigns 1 to EABI attribute 34
        .eabi_attribute 18, 4   //Assigns 4 to EABI attribute 18
        .file   "P1-1.c"        //sets the tile to be translated to P1-1.c
        .text                   //signifies the beginning of code
        .global var1             //makes var1 visible to linker
        .data                   //signifies the beginning or read/write data
        .type   var1, %object //sets var to type object
        .size   var1, 1         //sets size of var1 to 1
var1:
        .byte   1               //declares byte with 1
        .global var2             //makes var2 visible to linker
        .type   var2, %object //sets var to type object
        .size   var2, 1         //sets size of var2 to 1
var2:
        .byte   2               //declares byte with value 2
        .global var3            //makes var3 visible to linker
        .align  2               //ensures the address of the next line is a multiple of 2
        .type   var3, %object //sets var to type object
        .size   var3, 4         //sets size of var3 to 4
var3:
        .word   3               //declares word  with value 3
        .global var4            //makes var4 visible to linker
        .align  2               //ensures the address of the next line is a multiple of 2
        .type   var4, %object //sets var to type object
        .size   var4, 4         //sets size of var4 to 4
var4:
        .word   4               //declares word with value 3
        .global num             //makes num visible to linker
        .section  .rodata       //Sets current section to .rodata
        .align  2               //ensures the address of the next line is a multiple of 2
        .type   num, %object  //sets num to type object
        .size   num, 4          //sets size of num to 4
```

```
num:
        .word   -10             //declares word with value -10
        .global wave            //makes wave visible to linker
        .data                   //signifies the beginning or read/write data
        .align  2               //ensures the address of the next line is a multiple of 2
        .type   wave, %object //sets num to type object
        .size   wave, 10        //sets size of wabe to 10
wave:
        .ascii  "goodbye!!!" //declares an ascii string
        .text                   //signifies the beginning of code
        .align  2               //ensures the address of the next line is a multiple of 2
        .global main            //makes main visible to the linker
        .arch armv6             //sets architecture to armv6
        .syntax unified         //Using the unified ARM assembly syntax
        .arm                    //Generates ARM instructions
        .fpu vfp                //Identifies vfp as the floating point to assemble for
        .type   main, %function         //sets main as type function
main:
        @ args = 0, pretend = 0, frame = 8              //Setting up the main function
        @ frame_needed = 1, uses_anonymous_args = 0     //Setting up the main function
        @ link register save eliminated.                //Setting up the swap function
        str     fp, [sp, #-4]!    //store register, uses fp as source, sp as base, and #-4 as the offset
        add     fp, sp, #0        //add(immediate) adds fp and sp into fp, #0 being the immediate
                                  value to add to sp
        sub     sp, sp, #12       //sub(immediate) subtracts sp from sp and stores in sp where #12
                                   is the value subtracted from the value obtained by sp (second)
        mov     r3, #5            //moves register r3 into register 5 register
        str     r3, [fp, #-8]     //calculates an address from base register(fp) and offset register
                                  value(#-8) to store the memory location in r3
        bl      .L2               //branches to target address (L2)
.L3:
        ldr     r3, .L9          //loads register r3 with contents of .L9
        ldrsb   r3, [r3]         //loads register with a signed byte from r3 into r3
        uxtb    r2, r3           //unsigned extend byte extracts 8-bit integer from register r2,
                                   extends it to 32 bits, and writes the result to r3
        ldr     r3, .L9          //loads register r3 with contents of .L9
        ldrsb   r3, [r3]         //loads register with a signed byte from r3 into r3
        uxtb    r3, r3           //unsigned extend byte extracts 8-bit integer from register r3,
                                   extends it to 32 bits, and writes the result to r3
        smulbbr3, r2, r3
        uxtb    r3, r3          //unsigned extend byte extracts 8-bit integer from register r3,
                                  extends it to 32 bits, and writes the result to r3
        sxtb    r2, r3          //Signed Extend Byte extracts an 8-bit value from a register(r3),
                                  sign extends it to 32 bits, and writes to r2
        ldr     r3, .L9         //loads register r3 with contents of .L9
        strb    r2, [r3]        //stores register byte to memory , r2 is source register, r3 is the
                                base register
        ldr     r3, .L9         //loads register r3 with contents of .L9
        mov     r2, #1          //move r2 with an offset of 1
        strb    r2, [r3]        //stores register byte to memory , r2 is source register, r3 is the
base
                                register
        ldr     r3, .L9         //loads register r3 with contents of .L9
        ldrsb   r3, [r3]        //loads register with a signed byte from r3 into r3
        uxtb    r3, r3          //unsigned extend byte extracts 8-bit integer from register r3,
                                extends it to 32 bits, and writes the result to r3
        lsl     r3, r3, #1      //logical shift left of r3 using the #1 value
        uxtb    r3, r3          //unsigned extend byte extracts 8-bit integer from register r3,
                                extends it to 32 bits, and writes the result to r3
        sxtb    r2, r3           //Signed Extend Byte extracts an 8-bit value from a register(r3),
                                sign extends it to 32 bits, and writes to r2
        ldr     r3, .L9         //loads register r3 with contents of .L9
            strb    r2, [r3]        //stores register byte to memory , r2 is source register, r3 is the
                                      base register
        ldr     r3, .L9         //loads register r3 with contents of .L9
        mov     r2, #0          //move r2 with an offset of 0
        strb    r2, [r3]        //stores register byte to memory , r2 is source register, r3 is the
base
                                register
        ldr     r3, [fp, #-8]   //load register r3 with fp (offset of -8)
        sub     r3, r3, #1      //subtract r3 from r3 with an offset of 1
        str     r3, [fp, #-8]   //store fp into r3 with an offset of -8
.L2:
        ldr     r3, [fp, #-8]   //load register r3 with fp (offset of -8)
        cmp     r3, #0          //subtracts 0 from r3, updates flags, and discards result
        bgt     .L3             //branch if greater than L3
.L4:
        ldr     r3, .L9+4       //load r3 with .L9 + 4
        ldr     r3, [r3]        //load r3 with r3
        sub     r3, r3, #1      //subtract r3 with an offset of 1 from r3 and store in r3
        ldr     r2, .L9+4       //load r2 with .L9 + 4
        str     r3, [r2]        //store r2 in r3
        ldr     r3, .L9+4       //load r3 with .L9+4
        ldr     r3, [r3]        //load r3 with r3
        cmp     r3, #0          //subtracts 0 from r3, updates flags, and discards result
        bne     .L4             //branch if not equal to .L4
        bl      .L8             //break to .L8
.L7:
        ldr     r3, .L9+8       //loads register r3 with .L9 +8
       ldrb       r2, [r3] @ zero_extendqisi2 //load register byte calculates an address from a base
                                              register (r3) value and an immediate offset, loads a
                                              byte from memory zero-extends it to form a 32-bit
                                              word, and writes it to r2
        ldr     r3, .L9+8             //loads r3 with .L9+8
        strb    r2, [r3]              //stores register byte to memory , r2 is source register, r3 is
                                      the base register
        bl      .L6                   //branch to .L6
.L8:
        ldr     r3, .L9+12             //load r3 with .L9+12
        ldr     r3, [r3]              //Load Register (register) calculates an address from a base
                                       register (r3) value and an offset register value, loads a word
                                       from memory, and writes it to r3
        cmp     r3, #3                //subtracts 3 from r3, updates flags, and discards result
        beq     .L7                    //if equal break to .L7
        nop                           //no operation
.L6:
        nop                           //no operation
        add     sp, fp,  #0           //add sp and fp with 0 offset and store in sp
        @ sp needed
        ldr     fp, [sp], #4          //Load Register (register) calculates an address from a
                                        base register (sp) value and an offset register value (4),
                                        loads a word from memory, and writes it to fp
        bx      lr                    //branch and exchange to lr
.L10:
        .align  2                      //ensures the address of the next line is a multiple of 2
.L9:
        .word   var1                  //assigns 4 bytes to var1
        .word   var4                  //assigns 4 bytes to var2
        .word   var2                  //assigns 4 bytes to var3
        .word   var3                  //assigns 4 bytes to var4
        .size   main, .-main          //allocates the correct amount of space for main
        .ident  "GCC: (Raspbian 10.2.1-6+rpi1) 10.2.1 20210110" //places tag in this file
        .section  .note.GNU-stack,"",%progbits  //sets .note.GNU-stack w/ attr progbits
```

P1-2.s

```
.arch armv6
        .eabi_attribute 28, 1         //Assigns 1 to EABI attribute 28
        .eabi_attribute 20, 1         //Assigns 1 to EABI attribute 20
        .eabi_attribute 21, 1         //Assigns 1 to EABI attribute 21
        .eabi_attribute 23, 3         //Assigns 3 to EABI attribute 23
        .eabi_attribute 24, 1         //Assigns 1 to EABI attribute 24
        .eabi_attribute 25, 1         //Assigns 1 to EABI attribute 25
        .eabi_attribute 26, 2         //Assigns 2 to EABI attribute 26
        .eabi_attribute 30, 6         //Assigns 6 to EABI attribute 30
        .eabi_attribute 34, 1         //Assigns 1 to EABI attribute 34
        .eabi_attribute 18, 4         //Assigns 4 to EABI attribute 18
        .file     "P1-2.c"            //Begins a new lew logical file called P1-3A.c
        .text                         //indicates beginning of code
        .global var                   //makes var visible to linker
        .bss
        .align  2
        .type   var, %object
        .size   var, 4
var:
        .space  4                   //Reserves a zeroed block of memory that is 4 bytes
        .text                       //Switch to text segment
        .align  2                   //Aligns on a 2 byte boundary
        .global swap                //Assigns swap as a global symbol
        .arch armv6                 //Changing the assembly
        .syntax unified             //Using the unified ARM assembly syntax
        .arm                        //Set to standard Arm instruction encoding
        .fpu vfp                    //Setting up floating-point computation
        .type   swap, %function     //Declaring the swap function
swap:
        @ args = 0, pretend = 0, frame = 16             //Setting up the swap function
        @ frame_needed = 1, uses_anonymous_args = 0     //Setting up the swap function
        @ link register save eliminated.        //Setting up the swap function
         str    fp, [sp, #-4]!          //Store the value of fp onto the local stack frame at address
                                          4
        add     fp, sp, #0          //Sets fp to equal sp
        sub     sp, sp, #20         //Allocates 0x20 bytes of stack space for local variables
        str     r0, [fp, #-16]      //Stores the value of the register 0 in fr-0x16
        str     r1, [fp, #-20]      //Stores the value of register 1 in fr-0x20
        ldr     r3, [fp, #-16]      //Loads fp-0x16 into r3
        ldr     r3, [r3]            //Loads the address in r3 into r3
        str     r3, [fp, #-8]       //Stores the value of r3 into fr-0x08
        ldr     r3, [fp, #-20]      //Loads r3 from fp-0x20
        ldr     r2, [r3]            //Push the value of r3 into r2
        ldr     r3, [fp, #-16]      //Loads the value of fp-0x16 into r3
        str     r2, [r3]            //Stores the value of r2 into r3
        ldr     r3, [fp, #-20]      //Load the value of r3 into fp-0x20
        ldr     r2, [fp, #-8]       // Load the value of r2 into fp-0x08
        str     r2, [r3]            //Store the value of r2 into r3
        nop                         //No opperration
        add     sp, fp, #0          //Sets sp to the value of fp
        @ sp needed
        ldr     fp, [sp], #4        //loads word from  sp + 4 and writes it to fp
        bx      lr                  //Assembler branches to link register
        .size   swap, .-swap        //sets size of swap to the inverse of swap
        .align  2                   //ensures the address of the next line is a multiple of 2

        .global main                //makes main visible to linker
        .syntax unified             //Using the unified ARM assembly syntax
        .arm                        //causes ARM instruction set to be generated
        .fpu vfp                    //Identifies vfp as the floating point to assemble for
        .type   main, %function     //sets main to type function
main:
```

```
@ args = 0, pretend = 0, frame = 8     //Setting up the main function
@ frame_needed = 1, uses_anonymous_args = 0      //Setting up the main function
push    {fp, lr}                    //stores fp and lr in the stack
add     fp, sp, #4                  //fp = sp + 4
sub     sp, sp, #8                  //sp = sp - 8
mov     r3, #10                     //Set r3 to 10
str     r3, [fp, #-8]               //Store the value of r3 into fp-0x08
mov     r3, #20                     //Set r3 to 20
str     r3, [fp, #-12]              //Store the value of r3 into fp-0x12
sub     r2, fp, #12                 //r2 = fp - 12
sub     r3, fp, #8                  //r3 = fp - 8
mov     r1, r2                      //Store r2 into r1
mov     r0, r3                      //Store r3 into r0
bl      swap                        //Call the swap function
mov     r3, #0                      //Store 0 in r3
mov     r0, r3                      //Store r3 into r0
sub     sp, fp, #4                  // sp = fp - 4
@ sp needed
pop     {fp, pc}                    //Pop registers fp and pc off the descending stack
.size   main, .-main                //sets size of main to inverse of main
.ident  "GCC: (Raspbian 10.2.1-6+rpi1) 10.2.1 20210110"  //places tag in this file
.section  .note.GNU-stack,"",%progbits         //sets .note.GNU-stack w/ attr progbits
```

The assembler begins by invoking the necessary directives. The main subroutine of the assembly portion begins by pushing the fp and lr to the stack and allocating space for the local variables by offsetting the frame and stack pointer. In the main function of the C code, variable a is 10 while b is 20, so r0 is set to 10 while r1 is set to 20 via mov, str, and sub instructions. After r0 and r1 are loaded with their values, the assembler branches to the swap subroutine. The swap subroutine executes instructions that swap the values located in the pertinent memory addresses: Initially, [fp-0x20] contains 20 while [fp-0x08] contains 10. Once the instructions in the swap subroutine prior to nop have been executed, [fp-0x20] now contains 10 whereas [fp-0x08] contains 20, so the values have been swapped. After the nop statement, the sp is set to fp, and a word in sp + 4 is written to the the fp before branching back to main. R0 is then set to 0 since the main function in C returns 0.

P1-3A.s

```
.arch armv6                       //Sets architecture to armv6
.eabi_attribute 28, 1             //Assigns 1 to EABI attribute 28
.eabi_attribute 21, 1             //Assigns 1 to EABI attribute 21
.eabi_attribute 23, 3             //Assigns 3 to EABI attribute 23
.eabi_attribute 24, 1             //Assigns 1 to EABI attribute 24
.eabi_attribute 25, 1             //Assigns 1 to EABI attribute 25
.eabi_attribute 26, 2             //Assigns 2 to EABI attribute 26
.eabi_attribute 30, 6             //Assigns 6 to EABI attribute 30
```

```
        .eabi_attribute 34, 1                 //Assigns 1 to EABI attribute 34
        .eabi_attribute 18, 4                 //Assigns 4 to EABI attribute 18
        .file     "P1-3A.c"                   //Begins a new lew logical file called
                                              P1-3A.c
        .text                                 //indicates beginning of code
        .align   2                            //ensures the address of the next line is a
                                              multiple of 2
        .global next_char                     //next_char is now visible to linker
        .arch armv6                           //Sets architecture to armv6
        .syntax unified                       //indicates adherence to unified ARM/Thumb
                                              assembly syntax
        .arm                                  //causes ARM instruction set to be generated
        .fpu vfp                              //Identifies vfp as the floating point to assemble for
        .type    next_char, %function         //Sets next_char to type function
next_char:
        @ args = 0, pretend = 0, frame = 8    //Setting up the next_char function
        @ frame_needed = 1, uses_anonymous_args = 0      //Setting up the  next_char function
        @ link register save eliminated.      //Setting up the  next_char function
        str       fp, [sp, #-4]!              //-4 is added to sp, and the contents of fp are then
                                              stored at the new address in sp
        add       fp, sp, #0                  //Sets fp to equal sp
        sub       sp, sp, #12                //sp=sp-12
        mov      r3, r0                        //Copies value from register 0 to register 3 (r3 = 65)
        strb      r3, [fp, #-5]               //Stores byte in R3 at address fp-5
        ldrb      r3, [fp, #-5]               //gets value in address fp-5, zero-extends it to form
                                                32-bit word and writes it to r3
        add       r3, r3, #1                  //R3=R3+1
        uxtb      r3, r3                      //Extends the value stored in R3 from an 8 bit value
                                                to a 32 bit value and stores it in R3
        mov      r0, r3                       //Copies value from register 3 to register 0
        add       sp, fp, #0                 //sets sp to equap fp
        @ sp needed
        ldr        fp, [sp], #4               //loads word from  sp + 4 and writes it to fp
        bx         lr                         //Assembler branches to link register, which in this
                                                 instance holds the address to return back to main
        .size     next_char, .-next_char      //sets size of next_char to the inverse of next_char
        .section            .rodata           //Sets current section to .rodata
        .align   2                            //ensures address of the next line is a multiple of 2
.LC0:
        .ascii    "Next Character=%c\012\000"             //declares ascii string “next
                                                          character=%c\new line
        .text                                 //indicates beginning of code
        .align   2                            //ensures address of the next line is a multiple of 2
        .global main                          //sets main to global
        .syntax unified                       //Using the unified ARM assembly syntax
        .arm                                  //causes ARM instruction set to be generated
        .fpu vfp                              //Identifies vfp as the floating point to assemble for
        .type   main, %function             //sets main to type function
main:
        @ args = 0, pretend = 0, frame = 0  //Setting up the main function
        @ frame_needed = 1, uses_anonymous_args = 0     //Setting up the main function
        push    {fp, lr}                    //Stores fp and lr in the stack
        add     fp, sp, #4                  //fp=sp + 4
        mov     r0, #65                     //Writes 65 to R0
        bl      next_char                   //Branches to next_char and copies address of next
                                              instruction into LR
        mov     r3, r0                      //Copies value in register 0 to register 3
        mov     r1, r3                      //Copies value in register 3 to register 1
        ldr     r0, .L4                     //loads r0 with the value returned by .L4
        bl      printf                      //Branches to printf and copies address of next
                                            instruction into LR
        nop                                 //No operation
        pop     {fp, pc}                    //Pop registers fp and pc off the descending stack

.L5:
        .align  2                           //ensures address of the next line is a multiple of 2
.L4:
        .word   .LC0                        //declares word with value returned by .LC0
        .size   main, .-main                //sets size of main to invers of main
        .ident  "GCC: (Raspbian 10.2.1-6+rpi1) 10.2.1 20210110"         //places tag in this file
        .section  .note.GNU-stack,"",%progbits  //sets .note.GNU-stack w/ attr progbits
```

The assembler begins by invoking the necessary directives. The main subroutine in assembly starts off by pushing the fp and lr to the stack. Before branching to the next_char subroutine which corresponds to the next_char C function, the assembler increments the value of the fp by 4 and writes the value 65 to register 0. After branching to the next_char subroutine, the value of fp is stored at an offset of the sp address. The fp is then set to the value of the sp, and the sp is decreased by 12. The r0 and r3 then undergo many instructions; strb and ldrb are used to load r3 with the input char, and increment r3 by 1, similar to the "return in+1;" statement in the C code. The uxtb instruction then extends the char value in R3 to a 32 bit value, and that value is copied to r0. The sp is then set to the value of fp, and the fp is loaded with the word from sp + 4. The assembler then returns to the main subroutine, where the value in r0 is vopied to r1 via 2 MOV instructions, and r0 is loaded with the necessary characters for the printf line in C. The assembler returns back to the C code.

P1-3B.s

```
        .arch armv6                             //Sets architecture to armv6
        .eabi_attribute 28, 1                   //Assigns 1 to EABI attribute 28
        .eabi_attribute 20, 1                   //Assigns 1 to EABI attribute 20
        .eabi_attribute 21, 1                       //Assigns 1 to EABI attribute 21
        .eabi_attribute 23, 3                       //Assigns 3 to EABI attribute 23
        .eabi_attribute 24, 1                       //Assigns 1 to EABI attribute 24
        .eabi_attribute 25, 1                       //Assigns 1 to EABI attribute 25
        .eabi_attribute 26, 2                       //Assigns 2 to EABI attribute 26
        .eabi_attribute 30, 6                       //Assigns 6 to EABI attribute 30
        .eabi_attribute 34, 1                       //Assigns 1 to EABI attribute 34
        .eabi_attribute 18, 4                       //Assigns 4 to EABI attribute 18
        .file    "P1-3B.c"                          //Begins a new lew logical file called
                                                      P1-3B.c
        .text                                       //signifies the beginning of code
        .section        .rodata                     //sets section to .rodata
        .align   2                                  //align to boundary 2
.LC0:
        .ascii   "Next Character=%c\012\000"        //declares ascii string “next
                                                     character=%c\new line
        .text                                       //signifies the beginning of code
        .align   2                    //ensures the address of the next line is a multiple of 2
        .global main                                //sets main global
        .arch armv6                                 //sets architecture to ARMv6
        .syntax unified                             //Using the unified ARM assembly syntax
        .arm                                        //Set to standard Arm instruction encoding
        .fpu vfp                                    //Setting up floating-point computation
        .type    main, %function                    //sets main to type function
main:
        @ args = 0, pretend = 0, frame = 0          //Setting up the main function
        @ frame_needed = 1, uses_anonymous_args = 0      //Setting up the main function
        push     {fp, lr}                           //Stores fp and lr in the stack
        add      fp, sp, #4                         //fp = sp + 4
        mov      r0, #65                            //Copies 65 to r0
        bl       next_char                          //branches to next_char subroutine
        mov      r3, r0                             //copies value in r0 to to r3
        mov      r1, r3                             //copies value in r3 to r1
        ldr      r0, .L2                            //loads r0 with the value returned by .L2
        bl       printf                             //branches to printf
        nop                                         //no operation
        pop      {fp, pc}                           //loads fp and pc from stack
.L3:
        .align   2                    //ensures the address of the next line is a multiple of 2
.L2:
        .word    .LC0                               //declares word with value returned by .LC0
        .size    main, .-main                       //sets size of main to the inverse of main
        .ident   "GCC: (Raspbian 10.2.1-6+rpi1) 10.2.1 20210110"  //places tag in this file
        .section  .note.GNU-stack,"",%progbits      //sets .note.GNU-stack w/ attr progbits
```

The c code for P1-3B.s is identical to that of P1-3A.s apart from the removal of the "return in + 1" statement. Thus, it's corresponding assembly code is much shorter, and the next_char subroutine has only 2 instructions: r0 increments by 1 and the lr is copied to the pc.

P1-3BASM.s

```
//Assembly Subroutine
        .section ".text"                        //Sets current section to text
        .global next_char                       //Makes next_char a global variable

next_char:
        ADD r0,#1                                //Adds 1 to r0
        MOV pc,lr                               //Copies lr to pc
        .end                                    //Ends the file
```

demo.c

```
#include <stdio.h>

int addASM(int num1, int num2); //gets assembly addASM
int subASM(int num1, int num2); //gets assembly subASM
int divASM(int num1, int num2); //gets assembly divASM
int mulASM(int num1, int num2); //gets assembly mulASM
int getRemainder(int num1, int num2); //gets assembly getRemainder

int main(){
        int number1; //initializes number1
        char operator; //initializes operator
        int number2; //initializes number3

        printf("--------------------------------------------------------------------------------\n");

        printf("-Hello! Welcome to this simple calculator program.\n-This program has four
operations: +, -, *, /\n-To use the calculator, you will be asked to input a number, then an
operator, then another number all in the same line (example: '1+1')\n-Please note that arithmetic
requiring numbers larger than 2,147,382,647 is not possible in this calculator\n-To exit, enter '%'
when asked for an operator or press ctrl+c at any time\n"); //instructions

        printf("--------------------------------------------------------------------------------\n\n");

        while(1){ //A loop so multiple equations can be input before the user exits the program
        printf("-Please input an equation (e.g. '5/5', '10-2', etc. No spaces allowed):\n");
//asks for first number
        scanf("%d", &number1); //scans first number

        scanf("%c", &operator); //space before the %c so that the previous \n isn't
accepted by scanf instead of user input

        if(operator == '%'){ //checks if it's the 'quit' operator
            break; //exits loop (and therefore ends program) if so.
        }

        scanf("%d", &number2);        //scans second number

        if(number2 == 0 && operator == '/'){ //checks to make sure the user is not
dividing by 0 before calling any function
            printf("Cannot divide by 0\n"); //tells the user they cannot divide by 0
        }
        else{ //if not dividing by zero, does math
            if(operator == '+'){
                printf("\n%d %c %d = %d\n\n", number1, operator, number2,
addASM(number1, number2)); //prints out results for addition if '+'
            }
            else if(operator == '-'){
                printf("\n%d %c %d = %d\n\n", number1, operator, number2,
subASM(number1, number2)); //prints out results for subtraction if '-'
            }
            else if(operator == '*'){
                printf("\n%d %c %d = %d\n\n", number1, operator, number2,
mulASM(number1, number2)); //multiplication if '*'
            }
            else if(operator == '/'){
                printf("\n%d %c %d = %dr%d\n\n", number1, operator, number2,
divASM(number1, number2), getRemainder(number1, number2)); //prints out division (with
remainder) if '/'
            }
            else{printf("Something went wrong, please try again.\n\n",operator);}
//Just in case an operator is not recognized, tells the user as such
        }
    }

    return 0; //returns 0 to main
}
```

demoASM.s

```
    .section ".text"            @defines our section
    .global addASM              @creates global function addASM
```

```
        .global subASM              @creates global function subASM
        .global mulASM              @creates global function mulASM
        .global divASM              @creates global function divASM
        .global getRemainder        @creates global function getRemainder

addASM:                             @addASM routine
        ADDS r0, r1                 @uses built in function to add the two registers (which is
where the input params are)
        MOV  pc, lr                 @returns back to where we were before calling

subASM:                             @subASM routine
        SUBS r0, r1                 @uses built in function to subtract the two registers (which
is where the input params are)
        MOV  pc, lr                 @returns back to where we were before calling

mulASM:                             @mulASM routine
        MULS r0, r1                 @uses built in function to multiply the two registers (which
is where the input params are)
        MOV  pc, lr                 @returns back to where we were before calling



divASM:                             @Check which numbers are negative, then adds/subtracts
appropriately and adds to the result with each operation
        PUSH {lr}                   @pushes lr so we can ensure we can return to where we
were originally
        MOV r2, #0                  @fills r2 with 0
        ADDS r0, #0                 @adds r0 with 0 to see if neg flag is set
        BMI num_one_neg             @if it is, go to num_one_neg
        B       num_one_pos         @if not, go to num_one_pos

num_one_pos:                        @Executes fully (including loop1 and loop1_2) if both
nums positive
        ADDS r1, #0                 @checks if second number sets any negative flags
        BMI num_two_neg1            @if so, branch
loop1:                              @The main subtraction loop where num2 is subtracted
from num1 until a negative is reached
        SUBS r0, r1                 @subtracts repeatedly til negative result
        BMI loop1_2                 @checks for negative, branches if so
        ADD  r2, #1                 @if not negative then adds 1 to the counter (which counts
how many times num2 fits into num1)
        B   loop1                   @Loops back
loop1_2:
        MOV  r0, r2                 @Copies r2 to r0 so the actual division result can be sent
back
        B         done              @Goes to routine where we finish
```

```
num_one_neg:                            @Executes fully if num1 neg but num2 pos
        ADDS r1, #0                     @Checks if num2 is neg
        BMI num_two_neg2                @If so branches
loop2:                                  @main loop where we add r1 to r0 until it's positive
        ADDS r0, r1                     @adding
        BMI loop2_2                     @branches if negative to another segment that will loop
back to here
        CMP r0, #0                      @check if r0 is 0. Subtracts one more from the counter (r2)
if so.
        BNE loop2_3                     @Branches if not equal to 0 to finish loop
        SUB r2, #1                      @otherwise adds (or rather subtracts) one more to the
counter THEN finishes the loop
loop2_3:
        MOV  r0, r2                     @Copies result to result register
        B    done                       @done
loop2_2:
        SUB  r2, #1                     @Subtracts from counter (essentially counts up, as the
"counter" is negative)
        B    loop2                      @Loops back

num_two_neg1:                           @for when only the second number is negative
loop3:
        ADDS r0, r1                     @Adds the two numbers together until negative result
        BMI loop3_2                     @If negative, finishes loop
        SUB  r2, #1                     @If not negative, "adds" to the counter
        B    loop3                      @loops back til negative
loop3_2:
        MOV  r0, r2                     @copies result to result register
        B          done                 @done

num_two_neg2:                           @for when both numbers are negative
loop4:
        SUBS r0, r1                     @subtracts num2 from num1 til positive (since you're
subtracting a negative, r0 raises)
        BMI loop4_2                     @branches if negative to a loop that will loop back here
        CMP r0, #0                      @Check if r0 is 0. Subtracts from the counter (r2) if so.
        BNE loop4_3                     @If not, branches to done.
        ADD  r2, #1                     @If so, adds to the counter one last time
loop4_3:
        MOV  r0, r2                     @Copies r2 to result register
        B    done                       @done
loop4_2:
        ADD  r2, #1                     @Adds to the counter
        B    loop4                      @loops back

done:
        POP  {lr}                       @Pops that pushed lr
        MOV  pc, lr                     @returns



getRemainder:                           @This is near identical to the divASM function (so we
can return a remainder instead of the quotient this time), so I will mostly just comment out
the parts that are different

        PUSH {lr}
        MOV r2, #0
        ADDS r0, #0
        BMI num_one_negR
        B        num_one_posR

num_one_posR:                           @Executes fully (including loop1 and loop1_2) if both
nums positive
        ADDS r1, #0
        BMI num_two_neg1R
loop1R:
        SUBS r0, r1
        BMI loop1_2R
        ADD  r2, #1
        B    loop1R
loop1_2R:
        ADD  r0, r1                     @Once r0 is negative, we simply add r1 back to get the
remainder
        B         done

num_one_negR:                           @Executes fully if num1 neg but num2 pos
        ADDS r1, #0
        BMI num_two_neg2R
loop2R:
        ADDS r0, r1
        BMI loop2_2R
        CMP r0, #0
        BNE loop2_3R
        ADD r0, r1                      @If r0 = 0, adds back r1 to "counter" the subtraction that
will happen in loop2_3r
        SUB r2, #1
loop2_3R:
        SUB r0, r1                      @Subtracts r1 from r0 to get the remainder as a negative
        MOV  r3, r0                     @Copies this remainder value to r3
        SUB  r0, r3                     @Subtracts the negative remainder from itself twice
        SUB  r0, r3                     @To get the absolute value of the remainder
        B    done
loop2_2R:
        SUB  r2, #1
        B    loop2R

num_two_neg1R:                          @for when only the second number is negative
loop3R:
        ADDS r0, r1
        BMI loop3_2R
        SUB  r2, #1
        B    loop3R
loop3_2R:
        SUB  r0, r1             @subtracts r1 from r0 to get the remainder (is already positive so
no need for absolute value stuff)
        B       done

num_two_neg2R:                  @for when both numbers are negative
loop4R:
        SUBS r0, r1
        BMI loop4_2R
        CMP r0, #0              @Subtracts 1 to check if r0 is 0. Subtracts one more from the
counter (r2) if so.
        BNE loop4_3R
        SUB  r0, r1             @If r0 is zero, we subtract r1 from r0 an additional time to
counteract the addition that happens in loop4_3R
        ADD  r2, #1
loop4_3R:
        ADD  r0, r1             @Adds r0 and r1 to get the final remainder as a negative
        MOV  r3, r0             @copies r0 to r3
        SUB  r0, r3             @Subtracts a a negative from itself twice so that
        SUB  r0, r3             @we can get a positive remainder result
        B    done
loop4_2R:
        ADD  r2, #1
        B    loop4R
doneR:
        POP  {lr}
        MOV  pc, lr

        .end                    @end of assembly
```

Schematics (Hardware) - None

Analysis

Important directives and instructions were researched by examining reference manual and other resources available online. Pertinent directives include:

.align - This directive aligns the current location to a specified boundary, by padding with zeros.
.arch - this directive is used to control the target architecture of the program. Upon selection, previous architecture extensions will be reset.
.arm - this directive is identical to .code 32 in that it generates the instruction set for the ARM architecture.
.ascii - this directive generates and places strings into consecutive addresses. This directive can be followed by 0 or multiple string literals. Each string literal must be separated by a comma.
.bss - this directive instructs the assembler to append the following statements to the end of the bss section.
.byte - this directive takes zero or more expressions, each separated by a comma. Each expression is assembled into the next byte
.data - This directive tells the assembler that the following information is program data.
.eabi_attribute - This directive sets a build attribute in the output file
.end - This directive informs the Assembler they've reached the end of the Source File.
.file - This directive signifies the creation of a new logical file. The string that comes after this directive will be the name of the new file.
.fpu - This directive allows you to specify the floating-point unit to assemble for. Valid values for the -mfpu command-line option also work as arguments for this directive.
.global - This directive makes the listed variable global.
.ident - this directive is sometimes used to place tags in object files. Its behavior will depend on the target; for example, if using a.out object file format, the assembler will accept the directive for source-file compatibility with existing assemblers.
.section - This directive makes the listed section the current section, as well as adds attributes.
.size - This directive declares the symbol's size to be an expression.
.space - This directive reserves a zeroed block of memory.
.syntax - This directive is used to choose between divided or unified syntax, with divided being the default option. Some main features include:

- It is not necessary for # to precede immediate operands.
- New instructions for V6T2 and later architectures are available.
- All instructions set flags only if they have an S affix.

.text - This directive tells the assembler to define the current section as text
.type - This directive declares the type of a symbol and its visibility.
.word - This directive creates a space of memory (similar to an array) where each element holds 4 bytes.

The assembly strictly adheres to the standard ARM calling convention in all parts of the lab; the first four registers (r0-r3) are exclusively used in the assembly portion of the code and the data manipulated by these registers are transient, since they are briefly stored in these registers before they are moved to other registered are often overwritten by new data within the span of the program. These first four registers typically deal with values that will be temporary, and will thus be subject to change by other functions. The subsequent registers r4-r9 generally contain static values, and will not be altered, even by functions. Registers r10-r15 are considered special purpose registers, which may not always be used. These registers have specialized roles. The stack is also used to store values in registers as needed. The likelihood of using the stack increases as the required parameters for functions increase.

Conclusion

The purpose of this project was to introduce and provide experience using the ARM Instruction Set Architecture as well as the ARM Cortex A53 processor used in the Raspberry PI. This project was also intended to introduce using common directives and instructions. Additionally, this project emphasized following ARM calling conventions. It is good practice to follow standard ARM calling conventions because doing so allows the efficient transfer of parameters to and from functions.

References

Cite all sources you researched and/or used to perform this lab If no references were used then type 'none

Pseudo Ops (Using as). (n.d.). Sourceware.org. Retrieved September 11, 2022, from https://sourceware.org/binutils/docs/as/Pseudo-Ops.html

Documentation – Arm Developer. (n.d.). Developer.arm.com. Retrieved September 11, 2022, from https://developer.arm.com/documentation/100076/0100/A32-T32-Instruction-Set-Reference

Assembler Directives - x86 Assembly Language Reference Manual. (2012, October 1). Retrieved September 11, 2022, from https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html