Raspberry Pi ARM Calculator

Names

Roman Millem, Easton Culver, Muhammad Mahmood

CECS 525

Abstract

The objective is to bring familiarity with the use of ARM on the Raspberry Pi, along with utilizing the interface between c code and assembly. Essentially the functionality of functions were created to be later utilized in c for certain commands. The apparent goal was to create a four function calculator using only a single c file and a single assembly file. The output should have an interface screen of sorts to choose which function to utilize and the division operation had an additional criteria of returning a remainder along with a single output. Initially create the functionality of the operations in assembly, while subtraction and addition are existing functions in ARM, simply take the two inputs and subtract or add them. Further, there is a mul function or add as many times as the multiple, but it gets a little more complicated with negatives and other cases. Lastly, there was the case of division where a function for this does not exist. Regards to the division function one may keep subtracting and checking for certain flags depending on the case, and have exception cases for dividing by 0 for instance. Additionally, there were implementing remainders which required an equation of sorts where the quotient is multiplied by the divisor then subtract from the dividend. When it comes to outputting these values into c, send the desired values into register 0 then use the command of “move pc, lr”, then call the value in c as a function. In the division case send the quotient and remainder as two separate outputs or put them in an array and return the array. Lastly, set up switch case statements in the c code and create print statements to show the user options and run certain functions of the calculator, including an exit command to terminate the program.

Body

To achieve the goal of creating a calculator in the ARM format on a Raspberry Pi the basic operating system used for a Raspberry Pi was used. Raspbian is a version of Debian GNU/Linux OS specifically used for Raspberry Pi’s. This made it a perfect environment to perform all the actions that were needed to complete the initial project with no problems at all. Going from that, there were a handful of code snippets given that provided a strong base of understanding to finish the project.

The first given snippet of C code was used to generate an assembly file, and with this a simple understanding of assembly in ARM was able to be obtained. The C code was fairly simple with only a few variables, and three separate types of loops. This translated into a much longer assembly code that performed the exact same function. This showed a way to perform simple mathematical functions, as well as the different types of branches that can be used in the assembly language. It does this by first creating the initial variables as labels, and giving them the corresponding data. Var1 is created as a signed character variable with a value of 1, Var2 is created as an unsigned character variable with a value of 2, Var3 is created as a signed integer variable with a value of 3, and finally Var4 is created as an unsigned integer variable with a value of 4. Then it stores the fifth variable, with the value 5, into r3 and starts the for loop. This calls down to L2, which makes sure that the parameters are correct before looping into L3, which performs simple mathematical calculations. With the smulbb being used to multiply the variables, uxtb used to add the variables, sxtb used to subtract the variables, and finally a simple subtraction method to divide the variables. After this it then jumps to the do-while loop. Where it loads the current value of the Var4 into r3, subtracts 1 from it, and then returns it back to the Var4. Then checks if it passes the parameter to loop again, if not it goes into the while loop. This puts the value of Var2 into a register, and then sets it to 2, and checks if it is equal to 3. If it isn’t it branches to the end of the program.

The second given snippet of C code was also used to generate an assembly file. This one had a more straightforward approach in that it’s only goal was to swap the value of two different variables. Once this was converted to an assembly file it, again, was much longer than the corresponding C file. It used a simple call from the main function to the swap function. This utilized pointers to the values of the variables, as well as a temporary variable to swap the values, and then finally return it back to the initial main function. This starts off with the main function being used to store the values into the appropriate variables, with the first one being equal to 10, and the second then being equal to 20. These it stores in accessible locations for the swap function, before finally calling it. Moving into the swap function it uses sp as a temporary location to store the values while it swaps the values from the variables and branches back to the main function, which clears out the variables before ending the program.

In part 3A, the given c code was used to generate an assembly file. This assembly listing utilizes the stack pointer, the frame pointer, and link register to accomplish the task of incrementing an input character and returning the new character to print it out. The program reserves 12 bytes of space in the stack after initializing the frame pointer to stack address -4. The ARM (A32) calling convention allocates the only function parameter of next\_char() into register 0 (r0). This value is copied into register 3 (r3), which is also used for subroutine argument passing. The input character is first stored, then retrieved from the next position in the stack with the instructions ‘strb r3, [fp,#-5]’ and ‘ldrb r3, [fp,#-5]’. The ‘b’ on the end of the instructions ‘strb’ and ‘ldrb’ is necessary since unsigned characters are 8 bit data types. R3 is then incremented, which adds one to the ASCII value of ‘A’, returning ‘B’ into r3. With the ‘utxb’ instruction, r3 is extended to 32 bits, backfilling the value with 0’s. The value of r3 is transferred to r0 which acts as the return register for the function. The stack is then cleaned, deleting the created frame and restoring the frame pointer to the calling address. The instruction ‘bx lr’ returns from the function by branching to the link register. The main function pushes the frame pointer and link register onto the stack and specifies the calling address to 4. After the call to ‘next\_char’, the return value is moved into r1 from r0. The printf sentence is stored in r0 so that when the printf function is called, the output is formatted correctly, printing the data in r0 first followed by the data in r1..

Part 3B builds on part 3A by implementing the ‘next\_char’ function in a separate assembly source file and calling the function from the main function in a c source file. The two source files were compiled into a single executable using the command ‘gcc P1-3B.c P1-3B.s’. An assembly file was generated from the c source file to examine the difference between the listings generated in part A and part B. The newly generated listing from part B only contains the main function, supporting data labels and directives. The main functions of both listings are identical. The ‘next\_char’ function is implemented in a separate file and is simplified significantly compared to the generated function implementation in the previous part. The entire function is implemented using two instructions, ‘add r0, #1’, and ‘mov pc,lr’. This implementation capitalizes on the ARM (A32) calling convention, bypassing the need for manual stack and frame pointer manipulation, instead using r0 as the input and output passing register. The ‘mov pc,lr’ instruction returns from the function, continuing from the call in main.

Source Code (Software)

Listed below is C code provided:

//group 1

//Project 1 Part 1

//Compile using 'gcc P1-1.c'

//Global Data Types - initial explanation

signed char var1 = 1; //creates a signed character variable

unsigned char var2 = 2; //creates an unsigned character variable

signed int var3 = 3; //creates a signed integer variable

unsigned int var4 = 4; //creates an unsigned integer variable

const int num = -10; //creates a constant integer variable

char wave[10]="goodbye!!!"; //creates a character array of length 10

void main()

{

//Local data type

int var5 = 5; //creates a local integer variable

//various loop types

for (var5; var5>0; var5--) //for loop

{

var1\*=var1; //multiplies var1

var1/=var1; //divides var1

var1+=var1; //adds var1

var1-=var1; //subtracts var1

}

do //do-while loop

{

var4-=1; //subtracts 1 from var4

}while(var4>0); //until var is equal to or lower than 0

while(var3==3) //while loop

{

var2 = 2; //sets var2 equal to 2

Break; //ends the while loop

}

}

This C code was put into a compiler to compile it in assembly language, and the following code is the result:

.arch armv6

.eabi\_attribute 28, 1 //assigns the tag the corresponding value

.eabi\_attribute 20, 1

.eabi\_attribute 21, 1

.eabi\_attribute 23, 3

.eabi\_attribute 24, 1

.eabi\_attribute 25, 1

.eabi\_attribute 26, 2

.eabi\_attribute 30, 6

.eabi\_attribute 34, 1

.eabi\_attribute 18, 4

.file "P1-1.c"

.text

.global var1 //creates the global variable

.data

.type var1, %object

.size var1, 1

Var1: //creates the global signed char variable, and defines it

.byte 1

.global var2

.type var2, %object

.size var2, 1

Var2: //creates the global unsigned char variable, and defines it

.byte 2

.global var3

.align 2

.type var3, %object

.size var3, 4

Var3: //creates the global signed int variable, and defines it

.word 3

.global var4

.align 2

.type var4, %object

.size var4, 4

Var4: //creates the global unsigned int variable, and defines it

.word 4

.global num

.section .rodata

.align 2

.type num, %object

.size num, 4

Num: //creates the num variable

.word -10

.global wave

.data

.align 2

.type wave, %object

.size wave, 10

Wave: //creates the array variable that says goodbye

.ascii "goodbye!!!"

.text

.align 2

.global main

.arch armv6

.syntax unified

.arm

.fpu vfp

.type main, %function

Main: //starts the main function

@ args = 0, pretend = 0, frame = 8

@ frame\_needed = 1, uses\_anonymous\_args = 0

@ link register save eliminated.

str fp, [sp, #-4]!

add fp, sp, #0

sub sp, sp, #12

mov r3, #5

str r3, [fp, #-8]

b .L2

.L3: //performs the first loops addition, sub, mult, and division

ldr r3, .L9

ldrsb r3, [r3]

uxtb r2, r3

ldr r3, .L9

ldrsb r3, [r3]

uxtb r3, r3

smulbb r3, r2, r3 //multiplies r2, and r3

uxtb r3, r3 //adds r3 to r3

sxtb r2, r3 //subtracts r3 from r2

ldr r3, .L9 //loads .L9 into r3

strb r2, [r3] //stores [r3] into r2

ldr r3, .L9 //loads .L9 into r3

mov r2, #1 //puts the value of #1 into r2

strb r2, [r3] //stores [r3] into r2

ldr r3, .L9

ldrsb r3, [r3] //loads the signed byte

uxtb r3, r3 //adds r3 to r3

lsl r3, r3, #1

uxtb r3, r3 //adds r3 to r3

sxtb r2, r3 //subtracts r2 from r2

ldr r3, .L9

strb r2, [r3]

ldr r3, .L9

mov r2, #0 //mves the values of #0 into r2

strb r2, [r3]

ldr r3, [fp, #-8]

sub r3, r3, #1

str r3, [fp, #-8]

.L2: //checks to see if L3 breaks from the for-loop parameters

ldr r3, [fp, #-8]

cmp r3, #0 //compares r3 to #0

bgt .L3

.L4: //starts the do-while loop function

ldr r3, .L9+4

ldr r3, [r3]

sub r3, r3, #1 //subtracts #1 from r3

ldr r2, .L9+4

str r3, [r2]

ldr r3, .L9+4

ldr r3, [r3]

cmp r3, #0

bne .L4 //branches if not equal

b .L8

.L7: //while loop parameter check

ldr r3, .L9+8

mov r2, #2

strb r2, [r3]

b .L6 //branches to .L6

.L8: //starts the while loop

ldr r3, .L9+12

ldr r3, [r3]

cmp r3, #3

beq .L7 //branches if equal

.L6: //break from the while loop

nop

add sp, fp, #0

@ sp needed

ldr fp, [sp], #4

bx lr //branches and changes the instruction set

.L10: //used to set the alignment

.align 2

.L9: //used to end the program

.word var1

.word var4

.word var2

.word var3

.size main, .-main

.ident "GCC: (Raspbian 8.3.0-6+rpi1) 8.3.0"

.section .note.GNU-stack,"",%progbits

The following code snippet was provided in C:

//Group 1

//Project 1 Part 2

//volatile modifier for variables that change due to hardware interrupts

volatile int var;

//functions with pointers

void swap(int \*x, int \*y) //used to swap the values of two integers using a temporary int

{

int temp; //creates the temp variable to store the value

temp = \*x;

\*x = \*y;

\*y = temp;

}

//A stack frame is used here for the subroutine call

//also, the stack is used when switching between os and main

int main() //main function that assigns variables numbers, then swaps them

{

//local variables

int a, b; //creates and then assigns the variables their functions

a = 10;

b = 20;

swap(&a, &b); //swaps using pointers

return 0;

}

When compiled as an assembly file the following code was produced:

.arch armv6

.eabi\_attribute 28, 1 //used to assign values to the different attributes

.eabi\_attribute 20, 1

.eabi\_attribute 21, 1

.eabi\_attribute 23, 3

.eabi\_attribute 24, 1

.eabi\_attribute 25, 1

.eabi\_attribute 26, 2

.eabi\_attribute 30, 6

.eabi\_attribute 34, 1

.eabi\_attribute 18, 4

.file "P1-2.c" //name of the file

.text

.comm var,4,4

.align 2 //sets the alignment

.global swap

.arch armv6

.syntax unified

.arm

.fpu vfp

.type swap, %function

Swap: //starts the swap function

@ args = 0, pretend = 0, frame = 16

@ frame\_needed = 1, uses\_anonymous\_args = 0

@ link register save eliminated.

str fp, [sp, #-4]! //stores the value with an offset

add fp, sp, #0 //adds sp and #0 together

sub sp, sp, #20 //subtracts #20 from sp

str r0, [fp, #-16] //goes through and stores values to swap them

str r1, [fp, #-20]

ldr r3, [fp, #-16]

ldr r3, [r3]

str r3, [fp, #-8]

ldr r3, [fp, #-20]

ldr r2, [r3]

ldr r3, [fp, #-16]

str r2, [r3]

ldr r3, [fp, #-20]

ldr r2, [fp, #-8]

str r2, [r3]

nop

add sp, fp, #0 //adds fp and #0 together

@ sp needed

ldr fp, [sp], #4

bx lr //branches and changes the instruction set

.size swap, .-swap

.align 2

.global main

.syntax unified

.arm

.fpu vfp

.type main, %function

Main: //starts the main function

@ args = 0, pretend = 0, frame = 8

@ frame\_needed = 1, uses\_anonymous\_args = 0

push {fp, lr} //assigns the variables their values

add fp, sp, #4 //adds #4 and sp together

sub sp, sp, #8 //subtracts #8 from sp

mov r3, #10 //stores #10 into r3

str r3, [fp, #-8] //loads the value into r3

mov r3, #20 //moves the value #20 into r3

str r3, [fp, #-12]

sub r2, fp, #12

sub r3, fp, #8

mov r1, r2

mov r0, r3

bl swap //branches to the swap function

mov r3, #0

mov r0, r3

sub sp, fp, #4

@ sp needed

pop {fp, pc}

.size main, .-main

.ident "GCC: (Raspbian 8.3.0-6+rpi1) 8.3.0"

.section .note.GNU-stack,"",%progbits

The following is another code snippet provided:

//Group 1

//Project 1 Part 3A

#include <stdio.h> //includes the library

unsigned char next\_char(char in) //is a function to return the next letter in the alphabet

{

return in + 1;

}

void main()

{

printf("Next Character= %c\n", next\_char('A')); //prints to console the a short phrase, and then the return of the next\_char function

}

When converted to Assembly the following code was produced:

.arch armv6

.eabi\_attribute 28, 1

.eabi\_attribute 20, 1

.eabi\_attribute 21, 1

.eabi\_attribute 23, 3

.eabi\_attribute 24, 1

.eabi\_attribute 25, 1

.eabi\_attribute 26, 2

.eabi\_attribute 30, 6

.eabi\_attribute 34, 1

.eabi\_attribute 18, 4

.file "P1-3A.c"

.text

.align 2

.global next\_char

.arch armv6

.syntax unified

.arm

.fpu vfp

.type next\_char, %function

Next\_char: //creates the next char function

@ args = 0, pretend = 0, frame = 8

@ frame\_needed = 1, uses\_anonymous\_args = 0

@ link register save eliminated.

str fp, [sp, #-4]! //creates stack frame at address -4

add fp, sp, #0 //frame pointer local reference to function input parameter

sub sp, sp, #12 //reserves 12B of space in stack

mov r3, r0 //copy to r3

strb r3, [fp, #-5] //store char in stack frame

ldrb r3, [fp, #-5] //get stored value

add r3, r3, #1 //increment char to next char

uxtb r3, r3 //extending 8bit char to 32 bits

mov r0, r3 //copy new char into return register r0

add sp, fp, #0 //delete created frame from stack

@ sp needed

ldr fp, [sp], #4 //restore frame pointer to calling address

bx lr //return from function

.size next\_char, .-next\_char

.section .rodata

.align 2

.LC0: //prints out to the console

.ascii "Next Character= %c\012\000"

.text

.align 2

.global main

.syntax unified

.arm

.fpu vfp

.type main, %function

Main: //starts the main function

@ args = 0, pretend = 0, frame = 0

@ frame\_needed = 1, uses\_anonymous\_args = 0

push {fp, lr} //pushes return address onto stack

add fp, sp, #4 //frame pointer calls to stack address 4

mov r0, #65 //ASCII ‘A’ stored in r0

bl next\_char //calls the next\_char function

mov r3, r0 //copies ‘A’ into r3

mov r1, r3 //copies ‘A’ into r3

ldr r0, .L4 //prepares transfer register memory for printing

bl printf //call to printf

nop //effective atomic pause

pop {fp, pc} //return address moves to program counter

.L5:

.align 2

.L4:

.word .LC0

.size main, .-main

.ident "GCC: (Raspbian 8.3.0-6+rpi1) 8.3.0"

.section .note.GNU-stack,"",%progbits

The final code snippet provided had both a C and assembly part, shown below:

//Group 1

//Project 1 Part 3B

#include <stdio.h> //includes the library

unsigned char next\_char(char in); //creates the next character function

void main() //starts the main function

{

printf("Next Character= %c\n",next\_char('A')); //prints out the result of next\_char function

}

With the paired assembly code being:

//assembly subroutine

.section ".text"

.global next\_char //declares the next\_char function to be global

next\_char: //defines the next\_char function

ADD r0,#1

MOV pc,lr

.end

When compiled it created the following assembly code:

.arch armv6

.eabi\_attribute 28, 1

.eabi\_attribute 20, 1

.eabi\_attribute 21, 1

.eabi\_attribute 23, 3

.eabi\_attribute 24, 1

.eabi\_attribute 25, 1

.eabi\_attribute 26, 2

.eabi\_attribute 30, 6

.eabi\_attribute 34, 1

.eabi\_attribute 18, 4

.file "P1-3B.c" //includes the c file

.text

.section .rodata

.align 2

.LC0: //performs the next character printout

.ascii "Next Character= %c\012\000"

.text

.align 2

.global main

.arch armv6

.syntax unified

.arm

.fpu vfp

.type main, %function

Main: //starts the main function

@ args = 0, pretend = 0, frame = 0

@ frame\_needed = 1, uses\_anonymous\_args = 0

push {fp, lr} //pushes return address onto stack

add fp, sp, #4 //frame pointer calls to stack address 4

mov r0, #65 //ASCII ‘A’ stored in r0

bl next\_char //calls the next\_char function

mov r3, r0 //copies ‘A’ into r3

mov r1, r3 //copies ‘A’ into r3

ldr r0, .L2 //prepares transfer register memory for printing

bl printf //call to printf

nop //effective atomic pause

pop {fp, pc} //return address moves to program counter

.L3: //ends the program

.align 2

.L2:

.word .LC0

.size main, .-main

.ident "GCC: (Raspbian 8.3.0-6+rpi1) 8.3.0"

.section .note.GNU-stack,"",%progbits

Schematics (Hardware):

None

Analysis

From the C code that was provided in the beginning a lot about the assembly language when used in the ARM environment. Drawing upon previous experience within the assembly language, as well as the much more readable C code provided it was clear to see how to properly create programs in the ARM format. The learning process was fairly simple with being given a C code that could easily be understood what each step did, compiled it within the given OS, and then going on to convert it into assembly to see how it was written in assembly. This provided a solid basis on the assembly language that was able to extend into the final project of trying to create a calculator within the same environment. While not given the exact individual tools, combining what was learned with the code examples and the previously known programming knowledge the final program became a lot easier. The final result was a simple calculator, but the real world applications are almost endless since it is the start of understanding the assembly language in the ARM format. Looking at it just from a language perspective almost any simple program can be created. After adding in the combination of C code and assembly it becomes even easier to create more advanced programs that deal with embedded systems or microcomputers.

Conclusion

Creating a 4 function calculator and getting a better understanding of ARM Assembly interface with c code along with utilizing a raspberry pi was the basis of the project. The calculator can be created by initializing and creating functions in assembly then call them in c code to display the functions. From the results it can be concluded that the simplest way of creating the calculator with the restriction of one c file and one assembly file was to define different functions in assembly, namely addition; subtraction; multiplication; and division. Use preset functions and the fact that multiple registers are available to replicate the four necessary mathematical operations. Lastly, to set the output to r0 then move the value to the c function. A simple method which is displayed to some degree in the provided demo code. Where converting the code into assembly files and commenting through it showed the use of certain commands in ARM that may assist in creating the calculator. Using “Move pc, lr” can be seen in the demo code, and assisted in producing the idea for the solution.

References

1. ARM Information Center: <http://infocenter.arm.com/help/index.jsp>
2. ARM Keil: <http://www.keil.com/support/man/docs/uv4/uv4_dg_adsas.htm>