Lab11–Assembly Code
CPSC 2311- Spring 2024

CLEMSON
School of COMPUTING

Introduction

The goal of this lab is to introduce assembly code; the low-level languages used to interact with computer hardware. We will be looking at and manipulating x86, and also creating a small program using ARM. You will be answering a few questions in the provided **Lab11Questions.txt** file as we progress through the lab. You will submit the Lab11Questions.txt file along with all of the other files you create during this lab.

IMPORTANT:
Everyone is required to complete this lab. If you choose not to complete this lab you will receive a 0 and this lab will not be dropped.

Due: Monday, April 22, 2024, midnight

Lab Instructions

You should **ssh to the SoC server cerf15** to complete this lab.

Part 1 – x86 intro

Our lab machines run on an x86-64 instruction set architecture (ISA). An ISA is essentially a description of what can run on a computer's hardware, and is the boundary between hardware and software. Compilers turn code (C, C++, Java, etc) into something the machine can actually understand using the ISA. We can use **gcc** to generate x86-64 code so we can view it directly (you've done this in lecture).

Create a directory called Lab 11. In your Lab11 directory, create a new file **driver.c** and write this simple loop:

```
#include <stdio.h>
int main(){

      int min = 0;
      int max = 10;
      for(int i = 0; i <= max; i++){
            printf("%d ", i);
      }

      printf("\n");
      return 0;
}
```

Compile and run the above program normally to make sure it behaves as expected:
gcc -Wall driver.c
You can ignore the warning of the unused variable.

Now, compile again using this command:
gcc -S -fverbose-asm driver.c

The -S flag generates an x86 file, and the -fverbose-asm flag adds some comments which could be helpful. Now take a quick glance at the **driver.s** file that was generated to see how 1 small for loop program was translated into something with roughly 4 times as many lines. This is why we abstract coding into higher level language like C. We can compile the **driver.s** file using **gcc** exactly as we did **driver.c.** Do that now, and run the executable to see the results are the same.

Now, we'll take a closer look at **driver.s**
You do not need to know all the specifics the x86-64 instruction set for this lab, and we won't cover it in-depth, but we will get to some basic understanding of what is happening.

Some basics:
The commands you see at the start of each line (push, mov, sub, etc.) are called mnemonics. They're usually followed by a letter (q, l, w, b). At a base level, the mnemonic is the description for what we're doing to some variable, the letter represents the size of the variable. The **$** signals literal values, while **%** signals a register. If you see the following lines:

```
movl  $0, -8(%rbp)    #, min
movl  $10, -4(%rbp)   #, max
```

The **mov** mnemonic places a value into a memory address. Those 2 lines are where we assign the value **0** to min, and the value **10** to max. This is where that -fverbose-asm flag comes in handy by commenting the variable names to the right of the line. Otherwise, we would've had to work back to understand the memory locations being referred to by **-8(%rbp)** and **-4(%rbp)**. As it stands, it's sufficient to know that **%rbp** is a pointer to the base of the stack, and **%rsp** points to the top of the stack. The first few lines in **main:** set up the stack to hold enough space for the ints we use, and we can tell that max, min, and i are respectively stored at memory locations 8, 4, and 12 bytes offset from the base of the stack.

Part 2- x86 manipulation
Now let's mess with the driver.s file a little and see what happens. Go ahead and try changing the values from the 2 lines above to other numbers and see what happens. Use your results to fill in the table for question 2

Change the values back to 0 and 10, and notice how the end of **main:** has the **jmp** command. This sends the program counter to the label specified. In this case, .**L2**
It's similar to a function call. When you call a function, the code there will be executed next.

See if you can follow along with these brief descriptions of **.L2** and **.L3**

.L2
Places **i**'s value in a temp register, and the compares that value to **max**
jle – **J**ump if **L**ess than or **E**qual to
Any time a comparison is done, the result is stored in a register which is looked at whenever a conditional jump command is used.
The rest in this section can be ignored

.L3
Puts **i**'s value in a temp register
Uses the temp register to connect to the start of a stream operation **%esi**
Uses the LCO section at the top of the file to call printf with the specified string.
Increments **i**

You should now be able to answer questions 1 – 5

Part 3 – ARM intro
We're going to dive right into another assembly language called ARM. If you look at the provided **main.c** file, you'll see a basic main function which creates several arrays and then prints out the result when those arrays are passed into a function called absVal() which is not defined yet. This function returns the absolute value of the all integers in an array added together. Your job is to write the absVal function in ARM (with some help, of course).

This is the standard idea of the function in C:

```
int absVal(int* arr, int length){
     int total = 0;
     for(int i = 0; i < length; i++){
          if(arr[i] < 0){
               total -= arr[i];
          }else{
               Total += arr[i];
          }
     }
     return total;
}
```

Open the **absVal.s** file. The basic outline for the function is already in place. We can now use this outline to cover some of the ARM fundamentals. Specifically, how functions work in ARM. When you call a function, the parameters are stored in registers, which are designated by the letter R and then a number, starting at R0. Since we have two parameters, those will be passed in R0 and R1. Meanwhile, when our function ends and we end up back in main, the computer will look for the return value in R0. This means the last step of the absVal function will be to move the absolute value to the R0 register. We don't do that at the beginning because R0 is where the array address is stored, and we don't want to overwrite it.

A key concept in ARM is keeping track of which registers are storing which information. That's why it's good to define the registers you're using in a comment at the beginning of a file such has been done for you in absVal.s

I've already set up the basic structure of your **absVal.s** file.

Labels: any word followed by a colon is a label. It defines a section of code, and is not in itself a command. That means you can have as many labels as you want, and they don't interfere with the code at all. In the following example, the computer will execute instruction 1 and then instruction 2

```
Main:
Boo:
     instruction 1
lessthan:
loop:
go:
label:
     instruction 2
```

The point of labels is to tell the program where to go whenever branches are used. Branches in ARM are the equivalent of jumping in x86. Branches/jumps are how we deal with conditionals in assembly languages. If you look back to the C example above, you'll see we have 3 conditionals: 1 for the for loop, the if, and the else. This means we have different sections of code that may or may not be executed; we'll divide those sections into different labels with ARM

I've already set up labels in **absVal.c**
The **loop** label is where we set up the basic loop execution
We'll branch to **add** if the current int (arr[i]) is positive, and to **subtract** if it's negative
increment will be used to increment 'i' and then branch immediately back to **loop**
done sets the return value

Things that were completed for you

Before we get into what you need to do, let's take a look at what has already been completed.

loop

In loop, r2 and r1 are compared, and then we bge (**b**ranch if **g**reater than or **e**qual to) to **done**
This accomplishes the `i < length` conditional in the for loop above.

If i is less than the length (ie, not greater than or equal to), then we continue on in the **loop** code. The description for the next two lines is in the comments next to them. You need to look up the **lsl** and **ldr** commands online and answer question 6 to describe what those commands mean.

Increment

This section has been completed for you to demonstrate how mnemonics and opcodes work in ARM

```
add     r2, r2, #1
```
This line is the equivalent of r2 = r2 + 1
Basic math operations in ARM follow this same format (hint to how **lsl** works)
Remember how in x86 the **$** showed we were using the value of the following number (an immediate)? In ARM, the # represents the same thing.

```
bal     loop
```

this is a branch command, because it starts with **b**, but **al** means always. So at the end of the increment section, the code will then always go back to **loop**

done

done moves (**mov**) the total from where we've been storing it on r5 to r0
pops some stuff off the stack. If you notice, at the beginning of the function, we pushed several values onto the stack: r3, r4, r5, and lr
We stored r3, r4 and r5 on the stack because it's highly possible that main() was using those registers to store something, but we need those registers for our functions. When the program returns to main, we want to preserve whatever values were already in those registers, so that main can function correctly. So we push them on the stack, then pop them back in the registers when we're done.

Look up the meaning of lr and pc to give your best shot at answering question 7.

Part 4 – ARM coding

Step 1: initializing variables
You should initialize all the registers you'll be using to zero (besides the parameters)
One has been done for you; use it as an example for the others.

Step 2: if/else
I've already written the comparison opcode for the if/else in the c function above. Use branching to go to the appropriate label. Here's a list of possible branching suffixes:

```
al      always (just 'b' is also branch always)
eq      equal (zero)
ne      nonequal (nonzero)
cs      carry set (same as HS)
cc      carry clear (same as LO)
mi      minus
pl      positive or zero
vs      overflow set
vc      overflow clear
hs      unsigned higher or same
lo      unsigned lower
hi      unsigned higher
ls      unsigned lower or same
ge      signed greater than or equal
lt      signed less than
gt      signed greater than
le      signed less than or equal
```

Step 3: Add and subtract

Fill in what needs to happen in the body of the if/else statement, and then branch to the next step in the code. Look up commands in ARM if you're not sure. Use the other examples in the given code.

Compiling ARM
To compile your code, use the following command:
Use cerf15 on the SoC servers when running the below command.

```
arm-linux-gnueabi-gcc-9 absVal.s main.c --static
```

Submission Instructions

zip all files for this lab and submit them on canvas. Your submission should be named <lastname>.zip

Files to include: completed **Lab11Questions.txt**, **driver.c, driver.s, main.c, absVal.s**

Put your name, username, and lab section in a comment at the top of all files.