Learning Objectives

1. Introduction to Raspberry Pi
2. Basics of ARM architecture
3. ARM Assembly Basics
4. GDB: Debugging your program
5. Using C Libraries in Assembly Code
6. Arithmetic in ARM Assembly (Assignment)

1 Raspberry Pi

Raspberry Pi is a small single board computer. In our lab we will be using Raspberry PI 3 Model B. It has a 1.2GHz (maximum frequency) Quad Core processor and 1GB RAM. It also has the following peripherals.

- Wireless LAN and Bluetooth
- 100 Base Ethernet
- 40-pin extended GPIO
- 4 USB 2 ports
- 4 Pole stereo output and composite video port
- Full size HDMI
- CSI camera port for connecting a Raspberry Pi camera
- DSI display port for connecting a Raspberry Pi touchscreen display
- Micro SD port for loading your operating system and storing data
- Upgraded switched Micro USB power source up to 2.5A

We will be using some of them in this course.

2 Starting Raspberry Pi

1. We have already installed the Raspbian Operating System on microSD card. microSD card slot is at the bottom of the board as shown in the Figure 1.
2. Connect a USB keyboard, mouse and hdmi cable to the Raspberry Pi peripherals as shown in the Figure 2. Connect the other end of the hdmi cable to the monitor. **Show your setup to a TA before you power it up.**

**Make sure your Raspberry Pi board is on an insulated surface.** All connections on the board are exposed. The board does not have any insulated supports. Placing it on a conducting surface can result in a short circuit putting you in danger and destroying the board. In particular, top of lab PC’s and your laptop are not safe places.

3. Connect the micro USB cable to the board. This cable will power the board. Connect the USB end of the cable to power a lab PC USB port, your laptop USB port or a mobile charger. This should turn on Raspberry Pi and show you a GUI as shown in the Figure 3.

4. We will use GCC tool chain for compiling and debugging our programs.

5. We have installed Emacs, Vim and VS code (May or may not work) for you to edit your programs.

### 3 Lab Setup

1. Use illinoisNet_Guest to connect Raspberry Pi to the internet.

2. Clone your repository CS233 repository.

3. Run (all in one line) to add our repository as a remote

   ```
   git remote add honors_release \ 
   https://github-dev.cs.illinois.edu/cs233-fa20/_honors_release.git
   ```

Run the following (separate commands) to get the files for this lab

```
git fetch honors_release
git merge --allow-unrelated-histories honors_release/RaspberryPiLab0
```
Figure 2. Peripheral Connections

Figure 3. Raspbian GUI
You only need to do this step once. Once you make a commit with merged lab, it will be available in your repository and simply cloning it would be enough.

4. Commit and push your work frequently. If it gives you a missing git configuration error then configure git without the \texttt{--global} option. This is important because the hardware will be shared. We don’t want you to modify any global settings.

5. Once you complete your lab, commit and push your final version. Then delete the files from Raspberry Pi. This is important because the hardware will be shared.

6. You have the option of either working in pairs or individually but we \textbf{strongly encourage pairs}. If you work in pairs, please make sure to fill the partners.txt file.

7. You will have to do this lab during TA Office Hours but please read the full lab manual before coming to lab.
4 ARM Basics

Raspberry Pi has an ARM based processor. ARM is a Reduced Instruction Set Computer (RISC) architecture. It has a small number of simple instruction for writing programs. ARM based processors are found in all kinds of devices including cellphones and smart devices. In this course, we will be looking at ARM version 7. In this lab, we will have a basic overview of the registers available in ARM and we will be looking at arithmetic instructions.

ARMv7 processors have 16 registers named r0–r15. Figure 4 show the registers. The registers also have synonyms and we can use both names in a program to refer to these register. For this lab, we will be using r0 to r11 and r14 which is also known as LR (link register). The synonyms of the registers indicate some special functions of these registers which we will study in later labs. All the registers are 32-bit wide.

<table>
<thead>
<tr>
<th>Register</th>
<th>Synonym</th>
<th>Special</th>
<th>Role in the procedure call standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>r15</td>
<td></td>
<td>PC</td>
<td>The Program Counter.</td>
</tr>
<tr>
<td>r14</td>
<td></td>
<td>LR</td>
<td>The Link Register.</td>
</tr>
<tr>
<td>r13</td>
<td></td>
<td>SP</td>
<td>The Stack Pointer.</td>
</tr>
<tr>
<td>r12</td>
<td></td>
<td>IP</td>
<td>The Intra-Procedure-call scratch register.</td>
</tr>
<tr>
<td>r11</td>
<td>v8</td>
<td></td>
<td>Variable-register 8.</td>
</tr>
<tr>
<td>r10</td>
<td>v7</td>
<td></td>
<td>Variable-register 7.</td>
</tr>
<tr>
<td>r9</td>
<td>v6</td>
<td>v6</td>
<td>Platform register. The meaning of this register is defined by the platform standard.</td>
</tr>
<tr>
<td></td>
<td>SB</td>
<td>SB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TR</td>
<td>TR</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td>v5</td>
<td></td>
<td>Variable-register 5.</td>
</tr>
<tr>
<td>r7</td>
<td>v4</td>
<td></td>
<td>Variable register 4.</td>
</tr>
<tr>
<td>r6</td>
<td>v3</td>
<td></td>
<td>Variable register 3.</td>
</tr>
<tr>
<td>r5</td>
<td>v2</td>
<td></td>
<td>Variable register 2.</td>
</tr>
<tr>
<td>r4</td>
<td>v1</td>
<td></td>
<td>Variable register 1.</td>
</tr>
<tr>
<td>r3</td>
<td>a4</td>
<td></td>
<td>Argument / scratch register 4.</td>
</tr>
<tr>
<td>r2</td>
<td>a3</td>
<td></td>
<td>Argument / scratch register 3.</td>
</tr>
<tr>
<td>r1</td>
<td>a2</td>
<td></td>
<td>Argument / result / scratch register 2.</td>
</tr>
<tr>
<td>r0</td>
<td>a1</td>
<td></td>
<td>Argument / result / scratch register 1.</td>
</tr>
</tbody>
</table>
5 First Program

Our first ARM assembly language program is shown below. This program is also available in the lab files as `first.s`. We will use this program to learn how to compile and run our assembly programs.

```assembly
/* This is a block comment
   It can span multiple lines */

// This is a single line comment
.text // Beginning of the code section
.global main // Telling the compiler that main will be
              // globally available. Operating system
              // will jump to this point to run our program

main: // This is the main label. Entry point of
       // the program is always main

    mov r0, #20 // r0 = 20
    mov r1, #30 // r1 = 30
    add r2, r0, r1 // r2 = r0 + r1
    add r2, r2, #40 // r2 = r2 + 40
    mov r0, r2 // copy r2 to return value in r0. r0 = r2
    bx lr // return back to Operating system
```

The comments in the program explain it. We are just adding three numbers in this program. First the numbers are stored in register `r0` and `r1`. Then they are added and the result is stored in `r2`. Next we add a constant using the same add instruction. Finally we move the result to `r0`. We use `r0` to return the sum because `r0` has a special role of taking results of computations back to the calling function. Our program starts execution when the operating system calls the main function so in this case the value in `r0` will go back to the operating system. The operating system does not know that we are returning a sum here. For operating system the value returned by the program is the “exit code” designed to inform user if the program executed successfully (exit code = 0) or ended abnormally (exit code != 0). For now we will use the exit code to return sum. We will see a better method in the next section.

6 Compiling and Running the Programs

As mentioned earlier we will be using GCC tool chain to compile our programs. Execute following command.

```bash
gcc -o first first.s
```

This will produce the executable file `first`. Execute the the executable first as follows:

```bash
./first
```

There will be no output on the screen. Operating system will store the exit code it receives in the environment variable `?`. We can check the environment variable by executing the command.
echo $?

TODO: Play with first.s. Change the register values and recompile to see the changes in the results. Change one of the add instruction in first to accept two constants (example: add r2, #60, #70). What error message do you get on compiling this code?

7 Debugging the Program

We will use GDB for debugging our programs. GDB is part of the GCC tool chain. GDB is an extensive tool. You are encouraged to explore it as much as you can. We are going to cover a few basic commands that we will need for this lab. A summary is given in Table 1. All the commands in the table also have short forms. You use `gdb help` to look up the short forms.

<table>
<thead>
<tr>
<th>GDB Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>List available commands</td>
</tr>
<tr>
<td>break [address]</td>
<td>Set breakpoint at the specified address. The address can be given in many forms. Examples: break first.s:13 // sets a breakpoint at line 13 of file first.s break first.s:main // sets a breakpoint at label main in file first.s</td>
</tr>
<tr>
<td>run</td>
<td>Runs the program to be debugged. It will stop at the breakpoint.</td>
</tr>
<tr>
<td>s</td>
<td>Single step. When the step in the source file is a function call, gdb will step into the function and stop.</td>
</tr>
<tr>
<td>n</td>
<td>Single step. When the step in the source file is a function call, gdb will execute the whole function and stop at the next line in source file.</td>
</tr>
<tr>
<td>c</td>
<td>Continue execution until the next break point.</td>
</tr>
<tr>
<td>info break</td>
<td>Shows a numbered list of breakpoints in the program</td>
</tr>
<tr>
<td>delete break [number]</td>
<td>Delete breakpoint. The number is the number of the breakpoint in the list of breakpoints shown by the info break command.</td>
</tr>
<tr>
<td>info registers</td>
<td>Displays register values.</td>
</tr>
</tbody>
</table>

Table 1. GDB Reference

Steps to debug first.s:

1. `gcc -g -o first first.s`
   Compiles first.s with debugging information.

2. `gdb first`
   This will start GDB and change the prompt to `(gdb)`.

3. `break first.s:main`
   Set a breakpoint in first.s at label main. Note: this will not work is compilation is done without -g flag.

4. `run`
   This will run the program at stop at the start of main.

5. `info registers`
   Display values of all the registers.
6. Run till next line in first.s.

7. Repeat the last 2 steps and notice how the register values change as the program executes. Keep repeating the steps till you reach the end of the program.

**TODO:** Follow the steps listed above for first.s

### 8 Using C Library Functions in Assembly Code

In our first program, our program inputs were fixed and we returned the result of the program as an error code. In this section, we will improve that aspect of our program. We would like our program to take inputs from user and print results to screen before exiting. In C, this is achieved through the use of function `printf` and `scanf`. The following C code prompts the user for input and prints is back to screen.

```c
#include <stdio.h>

int num;
int main()
{
    printf("Enter a Number : ");
    scanf("%d", &num);
    printf("The Number is %d\n", num);
    return 0;
}
```

The assembly equivalent of the same program is provided in `inOut.s` in the lab files. It is also listed here for reference.

```
.data
    // Data segment of the program
promptstr: .ascii "Enter a Number :\0"
input: .ascii "%d\0"
outstr: .ascii "The Number is %d\n\0"

.align 4
    // address of an integer should be multiple of 4
num:

.text
    // Code segment
numptr: .word num
promptstrptr: .word promptstr
inputptr: .word input
outstrptr: .word outstr

.global main
main:
    push {lr}  // lr register must be saved
    // if the code has any function calls
    ldr r0, promptstrptr
    // printf argument is passed in r0
    bl printf
    // call to printf
```
The program starts with a data segment. Data segment is the memory area where a program can store its global data. We define all the string that we will need in this program in this segment. We also reserve space for an integer number that will hold the value of the number that user will input.

The text segment contains the main code for the whole process. The first line of code push lr saves the value of the register lr on stack. This register contain return address i.e. address of the location where our program should return when it finishes. We need to save this register because calling other functions will change the value of this register. We will see more details about this later in the semester.

In ARM, registers r0, r1, r2 and r3 are used to pass arguments to functions. The first argument is placed in r0, the second is placed in r1 and so in. If a function needs more than 4 inputs than other techniques are used. Both printf and scanf are capable of handling a variable number of arguments (can be > 4) but we will restrict ourselves to a maximum of 4 to avoid complications. For both printf and scanf the first argument is the format string. Strings are passed as pointers which is implemented in the code. For the remaining arguments, printf takes the arguments directly in the registers but scanf takes pointers to the locations where it will store the items it reads. Since we can only use 4 arguments, we will limit ourselves to reading and writing 3 numbers at a time. The function is called using the branch and link (bl) instruction. We will learn more about how this instruction works with lr register later in the course. The program listed above is divided so that mapping between C statements and the assembly program are highlighted. At the end of the program we pop instruction to restore lr register. At this point lr contains the correct return address. We use branch and exchange bx instruction to return control to the operating system.

printf and scanf are part of libc and GCC includes these by default in the compilation process. That’s why we do not need to change anything in the compilation command for this program. The full list of functions available in libc can be found at https://www.gnu.org/software/libc/manual/html_node/Function-Index.html If we want to use a function that is not part of libc then we will have to explicitly include the library in the compilation process. This may be required for your final project.

TODO: Make sure that you completely understand the assembly program using printf and scanf.
9 Final Lab Assignment

Write an assembly program that takes three numbers as input. Let the number be \( a, b \) and \( c \). Calculate the value of the following function:

\[
f(a, b, c) = \frac{ab + bc}{bc - ab} + \frac{ac + bc}{ab - ac}
\]

You should write this program in the file `lab0.s`. Complete output of your program should look like this:

Enter 3 integers separated by spaces: 4 5 6
\( f(a, b, c) = -8 \)

Here 4 5 6 is entered by the user. Your program should use a single `scanf` at the beginning and a single `printf` at the end after completing the computation. It is important to make a single call to print the result because of the way function calls modify the registers. We will learn about that in a future lab. You will also have to look up ARM instructions for arithmetic operations. A quick reference is ARM Quick Reference Sheet (http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf). This quick reference sheet will be your go to guide for ARM instructions for all labs.

Some test cases:
- \( f(4,5,6) = -8 \)
- \( f(10,20,30) = -7 \)
- \( f(-7,8,-9) = 8 \)
- \( f(-7,8,-9000) = 1 \)
- \( f(-7,8,9000) = 0 \)
- \( f(-7,800,-9) = 9 \)
- \( f(-7000,8,-9) = -1 \)

**IMPORTANT:** You must show your code and the result to a TA to get credit for the lab.
10 Additional Resources

1. GNU ARM Assembler Quick Reference  

2. The gnu Assembler  
   (https://web.eecs.umich.edu/ prabal/teaching/resources/eecs373/Assembler.pdf)

3. Debugging Assembly Code with gdb  
   (http://web.cecs.pdx.edu/ apt/cs491/gdb.pdf)

4. Introduction to ARM Assembly Basics  
   (https://azeria-labs.com/writing-arm-assembly-part-1/)

5. ARM Quick Reference Sheet  
   (http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf)

6. Procedure Call Standard for the ARM® Architecture  

   (https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf).