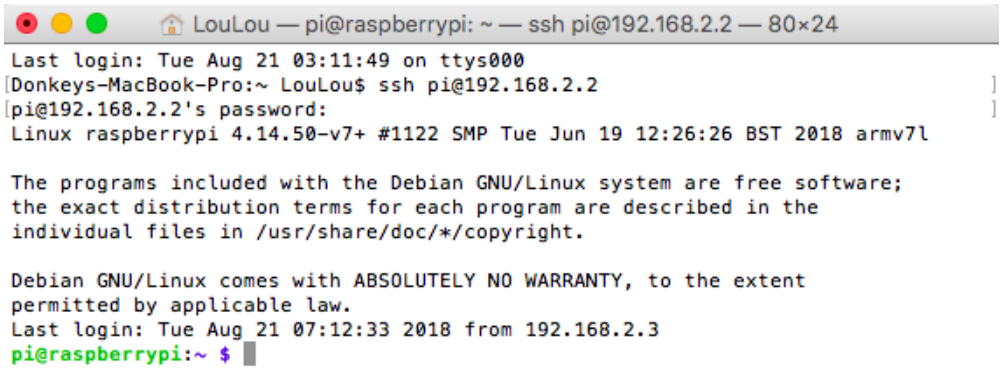
CSE 525 MICROCOMPUTER SYSTEMS DESIGN
PROJECT 1
Assembly and C in Microcomputer Design

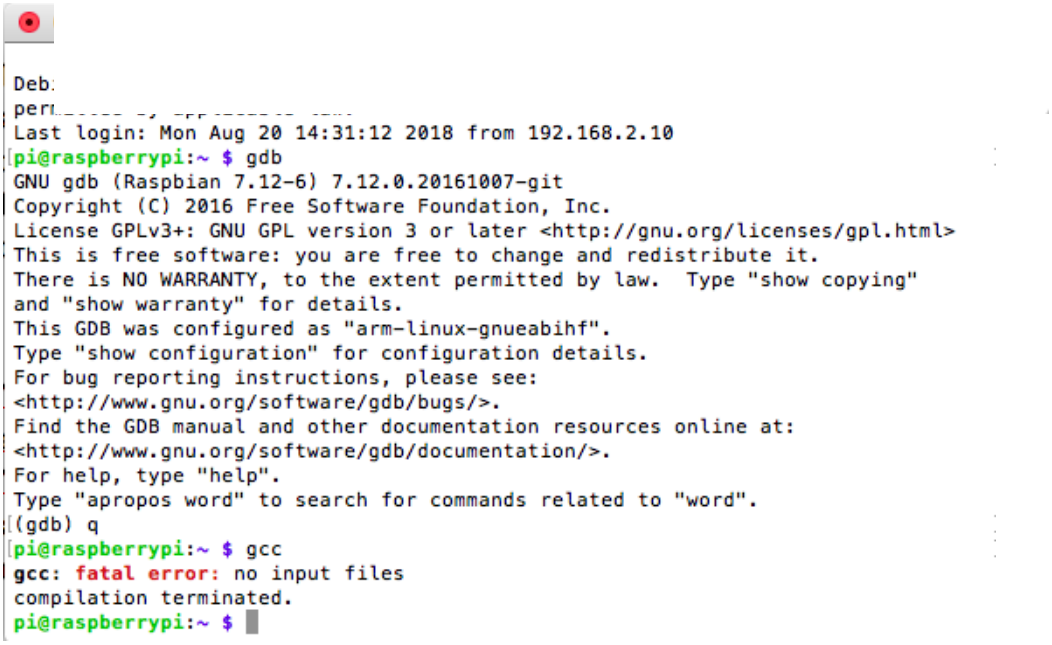
Yellow highlight points out lab related action required from the student.
Green highlight points out report related action required from the student.
Blue highlight emphasizes certain terms and information.

Project 1 is designed to introduce the student to Assembly and C programming as important components in microcomputer and embedded design. One important goal to achieve from project 1 is to understand the relationship between C and its equivalent assembly code. For example, examining an assembly code is a great way to debug its equivalent C code for logical errors and with respect to the target hardware. Professional C–compilers allow for the mixing of assembly source code with C source code using inline code and/or external calling conventions. A processor's assembly language/instruction set and addressing modes must be mastered in order to help one understand the processor's internal architecture and behavior – a behavior that extends out to the rest of the system during run-time. Project 1 is also designed to demonstrate the programming and proper use of the ARM Cortex A53 processor used in the Raspberry PI 3 Model B computer. The student is required to research areas that are unfamiliar as well as ask questions.

Part 1: Using a Raspberry PI lab computer, explore GCC, GAS, GDB, SSH, and the ARM programming model. When exploring SSH, you can optionally install, if necessary, SmarTTY onto your laptop to be able to SSH to a Raspberry PI computer in the lab (need to be on Belknap campus WIFI). Note: The latest Windows 10 won't need a third-party app such as SmarTTY in order to SSH, simply bring up a command-line window (run 'CMD') and use the SSH command at the command line. Apple OSX and Linux computers also have the SSH command built-in, simply launch a Terminal window and use. After connected to a RPI computer, you will find that GCC, GAS, and GDB are already installed (native) by default in the Raspbian OS. See these excellent references:

http://www.microdigitaled.com/ARM/ASM_ARM/Software/ARM_Assembly_Programming_Using_Raspberry_Pi_GUI.pdf
https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf
https://www.gnu.org/software/gdb/documentation/
https://sourceware.org/binutils/docs/as/





And of course, programming on the Raspberry PI computer can be accomplished directly using its own keyboard and mouse instead of remotely via WIFI and SSH.

Lab Tip:

One helpful way of learning a processor's architecture and operational behavior is by first learning its programming model including its Assembly instruction set and addressing modes and how these things are implemented according to the manufacturer's specifications or guidelines. A good starting point is the C compiler because a C compiler written for a particular processor is designed based on the manufacturers specifications or guidelines. Therefore, when a C program is written and compiled and the Assembly listing is examined, the manufacturer's specifications or guidelines are revealed. Note: You can also find or determine the specifications/guidelines in the manufacturer's technical data sheets. Understanding how the compiler generates the correct Assembly code will help you to do the same. If you "think" like the C compiler when you are writing Assembly source code then you

will be following the manufacturer's specifications or guidelines and your Assembly code will be 100% compatible however you are using it. So, learn to think like the C compiler by always examining the Assembly it generates. Sooner or later, you won't need to examine the Assembly unless for a quick reference or debugging logical errors. And as a C and Assembly systems programmer, you will benefit greatly by making this lab tip a habit.

Refer to the ARM Info Center for information about ARM Assembly Instructions, and Addressing Modes, and Calling Conventions, basically everything about the ARM technology can be found there.

Note: We are using the ARM Cortex A53 64-bit processor but the current Raspbian is a 32-bit OS that comes with the ARMv6 GCC toolchain for 32-bit builds. The A53 processor is designed to be backwards compatible with 32-bit software but with some decrease in performance. No 64-bit ARMv8 Raspbian exists at the time of this writing. However, there is a 64-bit SUSE Linux OS that runs on the Raspberry PI and supports ARMv8 64-bit builds.

Enter the following code in your favorite editor and save as P1-1.c, I will use nano. The Raspbian Text Editor and Leafpad are good built-in editors too. You may also choose to install and use the Code::Blocks editor on the Raspberry PI by entering 'sudo apt-get install codeblocks'. Compile by entering 'gcc P1-1.c'

```
//Eugene Rockey, Copyright 2018, All rights reserved.
//Project 1 Part 1
//Compile on Raspberry PI using the command 'gcc P1-1.c' to make sure there are no errors.
//Compile on Raspberry PI using the command 'gcc -S P1-1.c' to generate Assembly listing.
//Open the generated Assembly file named P1-1.s and fulfill part 1 requirements.

//Global Data Types
signed char var1 = 1;
unsigned char var2 = 2;
signed int var3 = 3;
unsigned int var4 = 4;
const int num = -10;
char wave[10]="goodbye!!!";
void main()
{
        //Local Data Type
        int var5 = 5;
        //Various Loop Types
        for (var5;var5>0;var5--)
        {
```

```
                var1*=var1;
                var1/=var1;
                var1+=var1;
                var1-=var1;
        }
        do
        {
                var4-=1;
        }while(var4>0);
        while(var3 == 3)
        {
                var2 = var2;
                break;
        }
}
```

After compiling on the Raspberry PI, generate an Assembly file listing (P1-1.s) by entering 'gcc -S P1-1.c'. In your report, discuss every unique Assembler directive in that listing, discuss every equivalent assembly section corresponding to each line of C code, comment all the Assembly lines of code describing the operation and addressing modes of each instruction.

Part 2: Use a Raspberry PI lab computer to continue to explore the ARM programming model. Enter and compile the following code with the name P!-2.c. After compiling on the Raspberry PI, generate an Assembly file listing (P1-1.s). In your report, discuss every new and unique Assembler directive in that listing, discuss every equivalent assembly section corresponding to each line of C code, comment all the Assembly lines of code describing the operation and addressing modes of each new and unique instruction.

```
//Eugene Rockey, Copyright 2018, All rights reserved.
//Project 1 Part 2
//Compile on Raspberry PI using the command 'gcc P1-2.c' to make sure there are no errors.
//Compile on Raspberry PI using the command 'gcc -S P1-2.c' to generate Assembly listing.
//Open the generated Assembly file P1-2.s and fulfill part 2 requirements.

//volatile modifier for variables that chage due to hardware interrupts, RTC, etc...
volatile int var;

//Function with Pointers
void swap(int *x, int *y)
{
```

```
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
//A stack frame (fp) is used here for the subroutine(function) call.
//Also, the stack is used when switching between OS and main() too.
int main()
{
        //local variables
        int a, b;
        a = 10;
        b = 20;
        swap(&a, &b);
        return 0;
}
```

Part 3:

A) Use a Raspberry PI computer to explore the ARM Calling Conventions(an industry standard). Calling Conventions are about how the C compiler utilizes the processor's registers and memory(stack) to pass input and output parameters between main() and subroutines or functions and between subroutines or functions. The processor's manufacturer specifies guidelines on how this should be done so that all other manufacturer's who implement said processor are on the same page making all their products compatible with each other. Note: If you do not want your product to be compatible then do not follow the industry standard Calling Conventions, use the registers and memory in some other way. However, this means you will also have to write your own custom C compiler in accordance with your custom register and memory usage or just program in 100% Assembly. We use Rasbian, which implements the industry standard ARM Calling Conventions; therefore, we must also take into account the ARM Calling Conventions in this Lab.

Enter and compile the following code with the name P1-3A.c. After compiling on the Raspberry PI, generate an Assembly file listing (P1-3A.s). In your report, discuss every new and unique Assembler directive in that listing, discuss every equivalent assembly section corresponding to each line of C code, comment all the Assembly lines of code describing the operation and addressing modes of each new and unique instruction. Comment and discuss the ARM Calling Conventions detected in the Assembly listing, specifying register and or memory(stack) used to pass parameters between main() and the subroutine or function.

```
//Eugene Rockey, Copyright 2018, All rights reserved.
//Project 1 Part 3A
//Compile on Raspberry PI using the command 'gcc P1-3A.c' to make sure no errors exist.
```

```
//Run and test the program by entering './a.out'
//Generate the Assembly listing by entering ‘gcc -S P1-3A.c’
//Open the generated Assembly file ‘P1-3A.s’ and fulfill part 3A requirements.

#include <stdio.h>
unsigned char next_char(char in)
{
        return in + 1;
}

void main()
{
   printf("Next Character= %c\n",next_char('A'));
}
```

B) Enter and compile the following source files with the names P1-3B.c and P1-3BASM.s. After compiling on the Raspberry PI, generate an Assembly file listing (P1-3B.s). In your report, discuss every new and unique Assembler directive in that listing, discuss every equivalent assembly section corresponding to each line of C code, comment all the Assembly lines of code describing the operation and addressing modes of each new and unique instruction. Comment and discuss the ARM Calling Conventions detected in the Assembly listing, specifying register and or memory(stack) used to pass parameters between main() and the subroutine or function.

Put the following C source code into its own P1-3B.c source file...

```
//Eugene Rockey, Copyright 2018, All rights reserved.
//Project 1 Part 3B
//Compile the program by entering ‘gcc P1-3B.c P1-3BASM.s’ and check for errors.
//Run the program by entering './a.out' at the command line.
//Generate Assembly listing by entering 'gcc -S P1-3B.c P1-3BASM.s'
//Open the generated equivalent Assembly file ‘P1-3B.s’ and fulfill part 3B requirements.

#include <stdio.h>
unsigned char next_char(char in);
void main()
{
   printf("Next Character= %c\n",next_char('A'));
}
```

Put the following Assembly source code into its own P1-3BASM.s file...

```
//Assembly Subroutine
//Eugene Rockey, Copyright 2018, All Rights Reserved
        .section ".text"
        .global next_char
next_char:
        ADD r0,#1     //How is the ARM calling convention obeyed here?
        MOV pc,lr
        .end
```

This document is subject to change.