Modern Assembly Language Programming with the ARM Processor Suggested Laboratory Exercises DRAFT

Larry D. Pyeatt

October 31, 2016

Contents

Part I

Assembly as a Language

Lab 1

Setting Up Your System/Logging In

This laboratory manual and the accompanying textbook are meant to be used with an ARM-based system running Linux. Several options are available. Your instructor will select the system that they prefer, and they will guide you through the process of setting up and logging in to your Linux-based ARM system.

1.1 Assignment

The purpose of this assignment is to set up your software development environment and learn to use it. There are three major pieces of a software development environment:

1. a text editor,
2. an assembler and/or compiler(s),
3. a linker,
4. a debugger,
5. a build (make) system.

Often, the linker and assembler are automatically invoked by the compiler when necessary. The make system is typically not used for very simple programs with only one or two source code files. However, as the number of source files grows, the make system becomes indispensable. These components are often managed by other software to form an Integrated Development Environment (IDE), but there is value to learning to use each component separately. In some situations, an IDE may not be available, or may not be a good choice. One drawback to most IDEs is that they only support a Graphical User Interface (GUI).

In the real world of networked computer systems, it is often necessary to edit files on a remote computer. This is often impossible or impractical for a GUI-based editor. However, some editors have the ability to work in either GUI or non-GUI mode. The ability to use one of these editors is a valueable skill. It only takes a few minutes to learn the basics. On Unix/Linux systems, there are two popular stand-alone text editors which can be run remotely without a GUI.

1.1.1 Emacs

Emacs is an editor written by programmers, for programmers. Emacs has a built-in Lisp interpreter which allows it to be extended without limits. Hundreds of Lisp extensions are available to support programming in almost any language. On most systems, these extensions are installed by default. Emacs has a GUI mode, but can also be run on a basic terminal without a mouse.

Commands are entered by holding down the Control key or the Meta key. For example, holding the Control key while pressing the 'f' key is written as C-f and causes the cursor to move forward one character. M-x indicates holding the Meta key while pressing 'x'. On keyboards without a Meta key, the Escape key is used instead (but the excape key is pressed and released before the next key is pressed). All of the basic cursor movement can be accomplished without the arrow (or any other special) keys.

Table 1.1: Short list of Emacs commands.

C-g	cancel an operation (go back to normal mode)
C-/	undo
C-h	*help*
C-h C-h	*help* on *help*: tell me how to use help
C-h a	*help apropos*: find help sections containing a string
C-x C-c	exit emacs
C-x C-f	find file (open file in new buffer)
C-x C-s	save current buffer (save file)
C-x C-w	write current buffer (save file as)
C-x k	kill current buffer (close file)
C-x b	switch to other buffer
C-x i	insert-file
C-b	go *back* one char
C-f	go *forward* one char
C-n	go to *next* line
C-p	go to *previous* line
C-v	go to next page: PageDown (v points down)
M-v	go to previous page: PageUp
C-s	*search*
C-r	*reverse* search: search backwards
C-d	*delete* char
C-k	*kill* from point to end of line: add it to the yank buffer
C-␣	set mark
C-w	*wipe*: kill text from mark to point and put it in the yank buffer
C-y	*yank*: insert contents of the yank buffer
M-y	*yank* previous yank buffer (repeat to go further back in history)
C-x 2	split window vertically
C-x 3	split window horizontally
C-x 1	close other windows
C-x o	other-window (if you've split the screen)

Like most modern editors, Emacs allows the user to select and manipulate regions of text. However, Emacs provides a way to do it without a mouse. The location of the cursor is known as the "point". A "mark" can be set at the current point. Once that is done, the point can be moved. All of the text between the point and the mark is selected as if with a mouse.

Table 1.1 shows some of the most useful Emacs commands. It is possible to edit a file using only the following six commands: C-x C-s, C-x C-c, C-f, C-b, C-p, and C-n. However, it is much more efficient to learn a few more commands. There are commands for killing and yanking rectangular regions of text, changing and controlling the behavior of the editor, spell-checking, and many others. Emacs is arguably the most powerful and extensible editor ever developed.

1.1.2 vi or vim

Vi is an acronym for the "visual interactive" editor, which was developed by Bill Joy as part of the early BSD Unix system. Vim (Vi IMproved) is a clone of vi with additional features. Like Emacs, vim supports both a character-based user interface and a graphical user interface. Vim can be customized using an internal scripting language known as vimscript, and also supports scripts written in Lua, Perl, Python, Racket, Ruby, and Tcl. Most modern Unix/Linux systems provide vim rather than vi.

Vim is a "modal" editor, meaning that it has specific modes of operation which are controlled by the user. The user changes the mode by using specific keys. Figure 1.1 shows the three modes for vim, and the keys that cause mode changes. Text can only be entered when vim is in input mode. In this mode, the cursor keys cannot be used to move the cursor. At first, this can be disconcerting for users who are used to free-movement editors, but most users become comfortable with it very quickly. Table 1.2 shows the most important commands for running vim. These commands are rougly equivalent to the Emacs commands given in Table 1.1

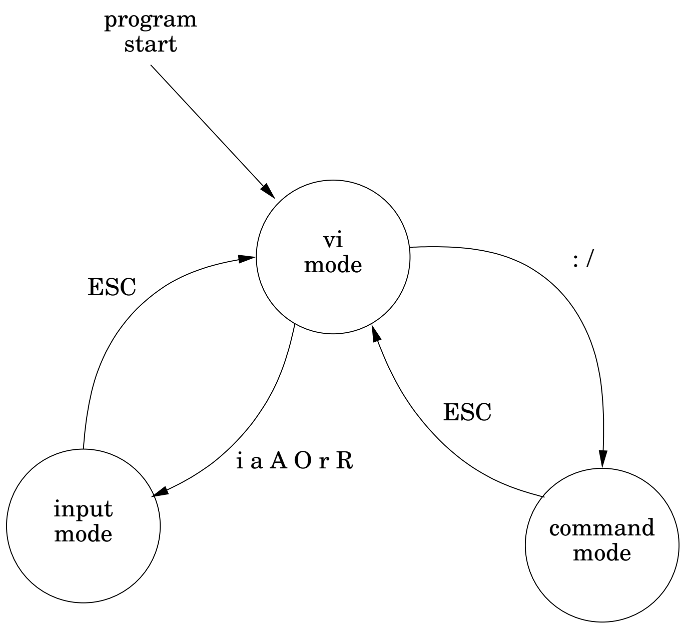


Figure 1.1: Modes for vi and vim.

1.2 Instructions

Your task for this laboratory assignment is to set up and/or login to your ARM system, and use a non-GUI editor. The following sections outline the two most powerful and useful editors on Unix/Linux systems. It is very helpful to know the basic commands for both of them, but for now, just select one and start use it to type in the text provided by your instructor.

Table 1.2: Short list of vi/vim commands.

vi mode	input mode	command mode
h j k l - cursor movement	any character - input text	:q - quit
CTL-f - forward screen	ESC - exit input mode	:q! - quit without saving
CTL-b - backward screen		:w - write
G - end of file		:wq - write and quit
x - delete character		:num goto line num
dw - delete word		:r file - import file
dd - delete line		:set all - vi settings
yy - copy line in buffer		
D - delete to EOL		/str - find str
p - paste/put buffer		
u - undo last command		
CTL-r - redo last undo (linux/vim)		
. - repeat last editing command		
n - find next occurrence of string		
cw - change word		
i - insert (go to input mode)		
a - append (go to input mode)		
A - append at EOL (go to input mode)		
O - open line (go to input mode)		
r - replace character (go to input mode)		
R - overwrite (go to input mode)		
: - command (go to command mode)		
/ - search (go to command mode)		

Lab 2

Hello World

This lab is just to get you started with the tools for writing assembly.

2.1 Assignment

You are to type and run three versions of the classic “Hello World” program. At this point, you don’t have to understand everything about the code. The point of this exercise is to practice typing assembly code in the editor, and to learn to compile/assemble and run your programs from the command line..

2.2 Instructions

1. Log in to your Raspberry Pi and start a text editor.
2. Type in the programs shown below.
3. Follow the instructions in the program headers to ompile or assemble and run them.

Listing 2.1: C Version

```
/**************************************************************************
 * helloc.c
 *
 * "Hello World" in C
 *
 * Scott K Logan
 *
 * Tue Jan 15, 2012
 *
 * This is a simple Hello World program written in C and using
 * printf() from the C standard library.
 *
 * It should be compiled and linked as follows:
 *   gcc -o helloc helloc.c
 *
 * gcc will call the compiler, assembler, and linker, telling the linker
 * to include the C standard library in the executable program.
 **************************************************************************/

#include <stdio.h>
```

```

int main()
{
  printf("Hello World\n");
  return 0;
}
```

Listing 2.2: Assembly Version 2

```
/*************************************************************************
 * hellogcc.S
 *
 * "Hello World" in ARM Assembly
 *
 * Scott K Logan
 *
 * Tue Jan 15, 2012
 *
 * This is a simple Hello World program written in ARM assembly and using
 * printf() from the C standard library.
 *
 * It should be compiled and linked as follows:
 *   gcc -o hellogcc hellogcc.S
 *
 * gcc will call the assembler and linker, telling the linker to include
 * the C standard library in the executable program.
 *************************************************************************/

        .data
msg:
        .ascii "Hello World!\n"
len     = . - msg

        .text
.globl  main
        @@ main is called by the _start function which is in the C
        @@ standard library.
main:
        stmfd   sp!,{lr}
        @ printf(msg)
        ldr     r0, =msg    @ buf -> msg
        bl  printf
        @ return from main()
        mov     r0, #0      @ status -> 0
        ldmfd   sp!,{lr}
        mov     pc,lr       @ return to _start function. It will call exit
```

Listing 2.3: Assembly Version 1

```
/*************************************************************************
 * helloas.S
 *
 * "Hello World" in ARM Assembly
 *
 * Scott K Logan
 *
 * Tue Jan 15, 2012
 *
 * This is a simple Hello, World program written in ARM assembly using
 * Linux system calls.
 * This version will run without the C standard library, and provides its
 * own _start function.
 *
 * It should be assembled and linked as follows:
 *   as -o helloas.o helloas.S
 *   ld -o helloas helloas.o
 ***************************************************************************/

        .data
msg:
        .ascii "Hello World!\n"
len     = . - msg

        .text
        .globl _start
_start:
        @ write(int fd, const void *buf, size_t count)
        mov r0, #1      @ fd -> stdout
        ldr r1, =msg    @ buf -> msg
        ldr r2, =len    @ count -> len(msg)
        mov r7, #4      @ write is syscall #4
        swi #0          @ invoke syscall

        @ exit(int status)
        mov r0, #0      @ status -> 0
        mov r7, #1      @ exit is syscall #1
        swi #0          @ invoke syscall
```

Lab 3

`printf` **and** `scanf`

One advantage to learning assembly on a Linux system is that you have access to the entire C standard library. Any function that can be called from a C program can also be called from assembly. For this lab, you must use the `printf` and `scanf` functions from the C standard library.

The built-in Unix/Linux manual pages provide documentation for all of the functions in the C standard library. The `man` program can be used to view the documentation from the command line. For example, to read documentation on the `scanf()` function, simply type `man scanf` at the command line. The built-in manual also has documentation on itself, which can be read by typing `man man` at the command line as shown below:

```
lpyeatt@rp08 $ man man
man(1)                      General Commands Manual                    man(1)



NAME
       man - format and display the on-line manual pages

SYNOPSIS
       man  [-acdfFhkKtwW]  [--path]  [-m system] [-p string] [-C config_file]
       [-M pathlist] [-P pager] [-B browser] [-H htmlpager] [-S  section_list]
       [section] name ...


DESCRIPTION
       man formats and displays the on-line manual pages.  If you specify sec-
       tion, man only looks in that section of the manual.  name  is  normally
       the  name of the manual page, which is typically the name of a command,
       function, or file.  However, if name contains  a  slash  (/)  then  man
       interprets  it  as a file specification, so that you can do man ./foo.5
       or even man /cd/foo/bar.1.gz.

       See below for a description of where man  looks  for  the  manual  page
       files.


MANUAL SECTIONS
       The standard sections of the manual include:

       1      User Commands

       2      System Calls

       3      C Library Functions

       4      Devices and Special Files

       5      File Formats and Conventions

       6      Games et. Al.

       7      Miscellanea

       8      System Administration tools and Deamons

       Distributions  customize  the  manual section to their specifics, which
       often include additional sections.
       .
       .
       .
```

3.1 Assignment

Write an ARM assembly program to do the following:

1. Prompt the user to enter an integer.
2. Read an integer from the keyboard into memory.
3. Prompt the user to enter another integer.
4. Read an integer from the keyboard into memory.
5. Load the two integers into CPU registers.
6. Add them together.
7. Print the result.

3.2 Instructions

Type in the program shown below and verify that it works as expected, then

1. add another variable, `m`, using the declaration of `n` as a template, (requires one assembly directive)
2. add another call to scanf, so that your program reads two numbers, `m` and `n`, from the user, (requires three instructions)
3. before calling printf:
   (a) load `m` into register 2, (requires two instructions)
   (b) load `n` into register 1, (already written; no change necessary)
   (c) add register 2 to register 1 and store the results in register 1 (requires one instruction),
4. change `str2` to “The sum is:”.

You will need to use the add instruction. The syntax is: `add Rd, Rm, Rn` where `Rd` is the register where you want the result stored, `Rm` is the register containing the augend, and `Rn` is the register containing the addend. For example, the instruction `add r0, r2, r3` would add the contents of `r2` to the contents of `r3` and store the result in `r0`.

3.3 C library Input and Output

The following program demonstrates how to call the `printf` and `scanf` functions. Note that `scanf` will store the value read in *memory* and not in a *register*. After `scanf` is called, the data must be loaded into a register before it can be used.

Listing 3.1: Program to echo a number to standard output.

```
        .data
str1:   .asciz  "%d"            @ Format string for reading an int with scanf
        .align  2
str2:   .asciz  "You entered %d\n" @ Format string for printf
        .align  2
n:      .word   0               @ A place to store an integer

        .text
        .globl main             @ This is a comment
main:   stmfd   sp!, {lr}       /* push lr onto stack                 */

        @ scanf("%d\0",&n)
        ldr     r0, =str1       /* load address of format string      */
        ldr     r1, =n          /* load address of int variable       */
        bl      scanf           /* call scanf("%d",&n)              */

        @  printf("You entered %d\n",n)
        ldr     r0, =str2       /* load address of format string      */
        ldr     r1, =n          /* load address of int variable       */
        ldr     r1, [r1]        /* load int variable                  */
        bl      printf          /* call printf("You entered %d\n",n) */

        ldmfd   sp!, {lr}       /* pop lr from stack                  */
        mov     r0, #0          /* load return value                  */
        mov     pc, lr          /* return from main                   */
        .end
```

Lab 4

Arrays, Loops, and Functions

4.1 Assignment

For this lab, you must use the `printf` and `getchar` functions from the C standard library to read an array of characters, print the characters on the terminal, and print the checksum of the array of characters. “A checksum or hash sum is a small-size datum computed from an arbitrary block of digital data for the purpose of detecting errors that may have been introduced during its transmission or storage. The integrity of the data can be checked at any later time by recomputing the checksum and comparing it with the stored one. If the checksums match, the data was likely not accidentally altered.”[1] You are to compute a 32-bit checksum by summing the ASCII values of the characters in the array.

You must do this using variables with static duration, then modify your code to use variables with automatic duration. i.e. use assembler directives to allocate variables, then change the code so that the variables are allocated on the stack at run-time.

4.2 Instructions

4.2.1 Part 1

Write an ARM assembly program to do the eqivalent of the following C code:

Listing 4.1: Program to calculate checksum with a bug.

```
/* This program uses global variables with static duration
   and global scope */

char buffer[4096];
int i=0;
int sum=0;

int checksum(char buffer[])
{
  for(i=0; buffer[i] != 0; sum += buffer[i++]);
  return sum;
}

int main()
{
```

[1]Wikipedia

```
  printf("Enter text (ctrl-D to end): ");
  do
    buffer[i]=getchar();/* getchar returns -1 on EOF */
  while((buffer[i] != -1)&&(++i < 4095));
  buffer[i] = 0;
  printf("%s\n",buffer);

  printf("\nThe checksum is %08X\n",checksum(buffer));
  printf("\nThe checksum is %08X\n",checksum(buffer));
  return 0;
}
```

If your assembly code is correctly implemented, the second checksum will be incorrect. WHY?

4.2.2 Part 2

Write an ARM assembly program to do the eqivalent of the following C code:

Listing 4.2: Program to calculate checksum.

```
/* This program uses automatic variables */
int checksum(char buffer[])
{
  int i;
  int sum=0;
  for(i=0; buffer[i] != 0; sum += buffer[i++]);
  return sum;
}
int main()
{
  char buffer[4096];
  int i=0;
  printf("Enter text (ctrl-D to end): ");
  do
    buffer[i]=getchar(); /* getchar returns -1 on EOF */
  while((buffer[i] != -1)&&(++i < 4095));
  buffer[i] = 0;
  printf("%s\n",buffer);

  printf("\nThe checksum is %08X\n",checksum(buffer));
  printf("\nThe checksum is %08X\n",checksum(buffer));
  return 0;
}
```

If your assembly code is correctly implemented, the second checksum will be correct. WHY?

Lab 5

Structured Data

5.1 Assignment

In digital systems, an image is usually stored as a 2-dimensional array of Picture Elements (pixels). Each pixel is stored as a data structure which specifies the color for one small region in the image image. The most common way to specify the color of each pixel is to provide values for the red, green, and blue components of the color. Eight bits are usually adequate for each component, so a pixel could be specified in C as follows:

```
typedef struct{
  unsigned char red, green, blue;
}rgbpixel;
```

Given that definition, color image data can be stored as a two-dimensional ($height \times width$) array, but it is often more efficient to structure the image as an array of rows, where each row is a pointer to a one-dimensional array of pixels. Also, it is useful to create a data structure containing a pointer to the array of rows, along with other information about the image, such as the width and height. This arrangement is shown in Figure 5.1, and the following listing shows the corresponding C data structure:

```
typedef struct{
  rgbpixel **rows; /* pointer to array of pointers */
  int height;      /* number of rows (length of array of pointers) */
  int width;       /* number of columns (length of each row) */
}rgbimage;
```

For a grayscale image, each pixel is an `unsigned char`, rather than an `rgbpixel`.

There are many file formats for storing digital images. Netpbm is a very simple and widely used format which is particularly useful as an intermediate format when converting between other

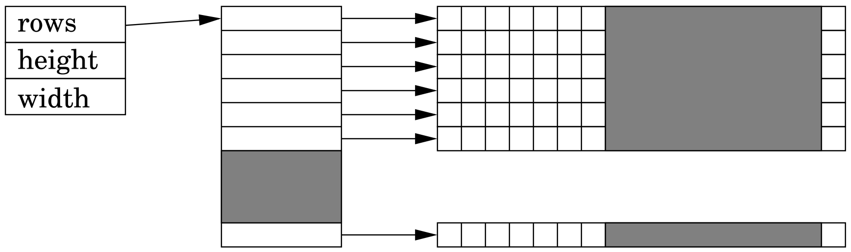


Figure 5.1: Data structure for storing an image in memory.

formats. Netpbm supports color, grayscale, or bitmap images stored in either ASCII or binary. This results in three formats for the data structure in memory, and six possible file formats.

For this laboratory assignment, you will be given a program, written in C, which converts a color Netpbm file to grayscale. The goal is to write an assembly language version of the function which does the actual conversion of the image. For each pixel, the color must be converted to a single number giving a grayscale value. The *luminosity* method performs the conversion using a weighted average. Human eyes are more sensitive to green than other colors, so green is weighted most heavily. Humans are much less sensitive to colors in the blue spectrum, so that component is weighted least heavily. The formula for luminosity is $gray = 0.21r + 0.72g + 0.07b$ where $gray$ is the grayscale value, and r, g, and b are the color components of the pixel. The trick is to perform the calculation using only integer mathematics. Applying a little algebra:

$$gray = 0.21r + 0.72g + 0.07b = \frac{54r + 184g + 18b}{256}.$$

Since a division by 256 is the same as shifting right by 8 bits, the entire calculation can be performed with three multiplications, two additions, and a shift.

5.2 Instructions

1. Download `structured.tgz` from the site provided by your instructor and extract the files:

```
lpyeatt@rp01 $ tar xfz structured.tgz
```

2. Edit `colortogray.S` and finish the function `color_to_gray`. Refer to the file named `netpbm.h` and the C source files for additional information.

Lab 6

A Queue ADT

6.1 Assignment

For this lab, you will implement some C++ functions for a queue ADC using an array. The main difficulty in linking assembly language with C++ is that C++ compilers "mangle" the names of class member functions. In order to write them in assembly, we must discover what name the C++ compiler creates. The second issue is that the C++ compiler always passes a pointer to the object, as the first parameter. So a C++ class member function has a "hidden" parameter, which is a pointer to the object being modified.

For example, given the following C++ class,

```
class example_class{
  private:
  .
  .
  .
  public:
  .
  .
  .
  int do_something(int x, int j);
  .
  .
  .
  }
```

the `do_something` member function would be mangled so that its C prototype looks like this:

```
int _ZN13example_class12do_somethingEii(example_class *this, int x, int j);
```

Additionally, a C++ objects are stored in memory in the same way as C structs. The only real difference is that C++ objects can have virtual functions. Virtual functions are stored in the "struct" as pointers to the actual functions.

6.2 Instructions

1. Download `queue.tgz` from the site provided by your instructor and extract the files:

   ```
   lpyeatt@rp01 $ tar xfz queue.tgz
   ```

2. Go into the `queue` directory and run `make`.
3. Use `objdump` to get a list of all of the symbols in the ADT.

```
lpyeatt@rp01 $ objdump --syms queue.o
```

4. Find the “mangled” names for the `enque`, `deque`, `isfull`, and `isempty` member functions.
5. Create a file named `queue_asm.S` and write the `enque` and `deque` functions in assembly, using the mangled names.
6. Comment out the enqueue function in the `queue.cc` file.
7. Modify the makefile to assemble and link `queue_asm.S` along with the C++ source files.

Part II

Performance Mathematics

Lab 7

Integer Mathematics

7.1 Division

7.1.1 Assignment

A prime number n is a natural number (i.e. a non-negative integer) which is greater than 1 and is not divisible by any other natural number other than 1 and n. In other words, n has no non-trivial factors other than 1 and n. So for example, 2 is the smallest prime number. To test whether a natural number n is a prime number or not, you can set up a loop to test every natural number between 1 and n (i.e. natural numbers from 2 to $n-1$) one by one to see whether n is divisible by any of them and count the number of non-trivial factors for n you have seen in the whole process. If n is not divisible by any of them, the number of non-trivial factors for n is 0 and n is a prime number; otherwise n is not a prime number.

7.1.2 Instructions

You are to find all prime numbers between 1 and n, where n is entered by the user. Your program should:

- prompt the user to input a natural number n,
- determine and display all the prime numbers between 1 and n in increasing order, and
- print the number of primes found between 1 and n.

Your program must have at least two functions:

`isprime` should accept one parameter, x and return 0 or 1, indicating whether or not x is prime.

`divide` should accept two parameters, x and y and should return the quotient and remainder (as two separate integers) found by dividing x by y.

7.2 Optimizing Multiplication and Division

7.2.1 Assignment

If you are like most people, you collect a fair amount of change in your pocket. A painless savings plan is to dump all the loose change into a jar at the end of the week. However, we'd like some idea of the amount of money we'll have saved in a year so we can start thinking about what we want to buy.

7.2.2 Instructions

In the this exercise you will write an assembly program to estimate the yearly savings based on the amount of change saved at the end of four weeks. The amount of change saved at the end of each week will be entered as four numbers: the number of pennies, nickels, dimes, and quarters. The program should read in four sets of weekly data. The program must then use exactly four calls to `printf` to outpute the following results:

1. the total count for each type of coin,
2. the total in dollars and cents,
3. the weekly average, and
4. the estimated yearly savings.

Your code must:

1. implement division by a constant as multiplication by its reciprocal,
2. implement multiplication by a constant as a sequence of shift and add operations.

Example Output

```
Enter the number of pennies, nickels, dimes, and quarters for week 1: 1 2 3 4
Enter the number of pennies, nickels, dimes, and quarters for week 2: 5 6 7 8
Enter the number of pennies, nickels, dimes, and quarters for week 3: 1 2 3 4
Enter the number of pennies, nickels, dimes, and quarters for week 4: 3 2 1 0

Over four weeks you have collected 10 pennies, 12 nickles, 14 dimes,
and 16 quarters.

This comes to $6.10
Your weekly average is $1.52
Your estimated yearly savings is $73.20
```

Lab 8

Fixed Point Math

For this lab, you will work with fixed point numbers. You will link your assembly code with C functions that help with input and output of fixed-point numbers.

8.1 Assignment

Below, you will find two C functions, `strtofixed` and `print_U` along with a helper function `base10double`. Note that you will have to modify them slightly to deal with signed numbers. You may type those functions as shown and use them in your program. You are to create a table for the function $f(x)$, where

$$f(x) = \frac{x^3 - x^2 - 2x}{\frac{(x-2)^4}{11} + 3}$$

Your fixed-point numbers should have a minimum of 16 bits in the fractional part during all phases of calculation.

8.2 Instructions

1. Output a brief descriptive message telling the user that the program will create a table for $f(x) = \frac{x^3-x^2-2x}{\frac{(x-2)^4}{11}+3}$.
2. Prompt the user and read a value for the lower limit of x. This will be the value of x for the first row in the table.
3. Prompt the user and read a value for the upper limit of x. This will be the value of x for the last row in the table.
4. Prompt the user and read a value for the number of rows they want in the table.
5. Generate a table with two columns. Show x in the first column, and $f(x)$ in the second column.
6. BONUS: Modify `print_U` so that it as a third parameter to specify how many digits to show after the decimal point. Print the table with 4 digits to the right of the decimal point on every number.
7. BONUS: Allow the user to specify how many bits of fractional precision should be used in the calculations.

Fixed Point I/O in C

```
#include <stdio.h>
#include <string.h>

#define MAX_DECIMAL_DIGITS 8
/* Multiply an unpacked BCD number by 2. Return 1 if there is a
   carry out of the most significant digit. 0 otherwise.  The
   resulting number is returned in n1.
*/
int base10double(char n1[MAX_DECIMAL_DIGITS])
{
  int i, tmp, carry=0;
  for(i=0;i<MAX_DECIMAL_DIGITS;i++)
    {
      n1[i] += (n1[i] + carry);
      if(n1[i] > 9)
        {
          n1[i] -= 10;
          carry = 1;
        }
      else
        carry=0;
    }
  return carry;
}

/* Convert a string into a signed fixed-point binary
   representation with up to 32 bits of fractional part.
*/
int strtoSfixed(char *s, int frac_bits)
{
  char *point = s;
  unsigned int value;
  int i,negative=0;
  char digits[MAX_DECIMAL_DIGITS];
  /* get the integer portion*/
  if(*s=='-')
    {
      negative=1;
      s++;
    }
  value = atoi(s);
  /* find the decimal point */
  while((*point != '.')&&(*point != 0))
    point++;
  /* if there is nothing after the decimal point, or there is
     not a decimal point, then shift and return what we already
     have */
  if(( *point == 0 ) || ( *(point+1) == 0 ))
    {
      if(negative)
        value = -value;
      return value << frac_bits;
    }
  ++point;
  /* convert the remaining part into an unpacked BCD number. */
  for(i=(MAX_DECIMAL_DIGITS-1);i>=0;i--)
```

```
    {
      if(*point == 0)
        digits[i] = 0;
      else
        {
          digits[i] = *point - '0';
          ++point;
        }
    }
  /* convert the unpacked BCD number into binary */
  while(frac_bits > 0)
    {
      value <<= 1;
      if(base10double(digits))
        value |= 1;
      frac_bits--;
    }
  /* negate if there was a leading '-' */
  if(negative)
    value = -value;
  return value;
}

/* Print an unsigned fixed point number with the given number of
   bits in the fractional part.  NOTE: frac_bits must be between
   0 and 28 for this function to work properly.
*/
void printS( int num, int frac_bits )
{
  unsigned int mask = (1 << frac_bits) - 1;
  unsigned int fracpart;
  if(num < 0)
    {
      printf("-");
      num = -num;
    }
  /* Print the integer part (with the sign, if it is negative) */
  printf("%d.",num>>frac_bits);
  /* Remove the integer part and keep the fraction part */
  fracpart = num & mask;
  /* Print all of the digits in the fraction part . The post -
     test loop ensures that the first digit is printed , even if
     it is zero. */
  do {
    /* Remember that multiplying by the constant ten can be done
       using a shift followed by an add with operand2 shifted ,
       or the other way around ... two instructions on the ARM
       processor. That is much faster than a mul instruction. */
    fracpart *= 10;
    printf ("%u", fracpart >> frac_bits);
    fracpart &= mask ;
  } while (fracpart != 0);
}
```

Lab 9

Floating Point Math

9.1 Assignment

If floating point hardware is available, it can be used to greatly accelerate non-integral mathematical operations. For this lab, you are to re-write the program from Lab 8, but this time you must use floating point instructions to perform your calculations.

9.2 Instructions

Re-write the program from Lab 8, but this time, use ARM VFP instructions to perform floating point mathematics. You can use `printf` and `scanf` for output and input of floating point values. The following example shows how to print floating point values.

```

fmt: asciz "%f\n"

     .
     .
     .

     @@ printf expects all floating point parameters to be passed
     @@ as IEEE double precision. If the second parameter to printf
     @@ is a float, then printf expects the 64-bit value to be
     @@ passed in two registers.  The choice of registers depends
     @@ on the stack pointer.
     @@ printf acts as if r0-r3 are part of the stack and requires
     @@ the 64-bit value to be aligned on an 8-byte boundary.  The
     @@ following lines print out the number in d0 by moving it to
     @@ the appropriate registers and calling printf.
     tst     sp,#4           @ check to see if stack is aligned
     vmovne  r1, r2, d0      @ move to r1,r2 if not aligned
     vmoveq  r2, r3, d0      @ move to r2,r3 if aligned
     ldr     r0,=fmt         @ load pointer to format string
     bl      printf
```

Lab 10

NEON

10.1 Assignment

For this lab, you will reimplement a previous lab, but this time you must use NEON to perform your calculations. If you are not careful, your output will contain roundoff errors.

10.2 Instructions

Re-implement the `color_to_gray` function from Lab 5, using NEON instructions. Note that you can load multiple pixels and process them in parallel.

Part III

Accessing Devices

Lab 11

General Purpose Input/Output

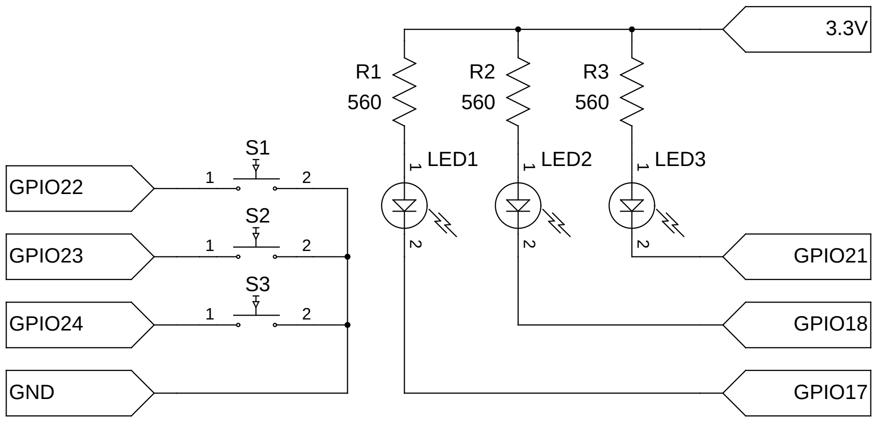
11.1 Assignment

The purpose of this lab is to become familiar with reading from and writing to general purpose I/O (GPIO) ports. The lab is divided into two distinct parts as defined below.

11.2 Instructions

11.2.1 Part 1

You will wire some buttons and LEDs to your single board computer (SBC). The following schematic shows one possible arrangement for a Raspberry Pi. The precise implementation may vary depending on which SBC you have. For further instructions, consult your instructor.



Next, you will download `GPIO.tgz` from the site provided by your instructor and extract the files:

```
Set the direction bits for the pins.
Enable the internal pull-up resistors on button pins.
while TRUE do
  Read the state of the buttons from the input ports
  Store to LED output ports
end while
```

Algorithm 1: Algorithm for controlling LEDs.

```
lpyeatt@rp01 $ tar xfz GPIO.tgz
```

This code provides a program to read the button states and illuminate the LEDs. The objective for part 1 is to display the current state of the buttons on the LEDs in positions corresponding to the buttons. Algorithm 2 gives an outline for this program.

11.2.2 Part 2

Part 2 consists of modifying the code given in Part 1 so that it maintains a count of button presses. The current count will be displayed on your terminal using `printf` and the least significant three bits of the count will be displayed on the LEDs. The “leftmost” switch will increment the count, the “center” switch will reset the count, and the “rightmost” switch will decrement the count. Algorithm 2 provides one possible approach to writing the program.

Some notes

The buttons and LEDs are active low. For the LEDs, this means that writing a low to the pin connected to the LED will turn the LED on. For the buttons, depressing the button will read low on the pin while the default button state will be high.

The *Raspberry Pi (BCM2835 ARM) Peripherals* manual (an be found on the course website) explains the functionality of the registers you will use mentioned above.

```
Map the GPIO device to a known location
Set direction bits
Enable internal pull-up resistors on button pins
Count ← 0
Print Count on terminal
Write least significant 3 bits of Count to LED ports
while TRUE do
  ButtonState ← GetDebouncedButtons
  if button 1 is pressed then
    Count ← Count +1
  end if
  if button 2 is pressed then
    Count ← 0
  end if
  if button 3 is pressed then
    Count ← Count −1
  end if
  Print Count on terminal
  Write least significant 3 bits of Count to LED ports
  repeat
    Read the state of the buttons
  until Neither button is pressed
end while

function GetDebouncedButtons
  DebounceCount ← 0
  repeat
    repeat
      Read the state of the buttons
      if Neither button is pressed then
        DebounceCount ← 0
      end if
    until One of the buttons is pressed
    DebounceCount ← Count +1
  until DebounceCount > DebounceThreshold
  return Buttonstate
end function
```

Algorithm 2: Algorithm for counting button presses.

Lab 12

Pulse Width Modulation

12.1 Assignment

The purpose of this lab is to become familiar with using the Pulse Width Modulator device.

12.2 Instructions

12.2.1 Raspberry Pi

Using the wiring from our previous lab, one of the LEDs is connected to GPIO 18. That pin can be reconfigured so that it is connected to the PWM device. You will modify the code from previous lab so that, instead of displaying the count on 3 LEDs, it uses the count to control the duty cycle of the PMW device, driving the LED on GPIO 18.

Thus, pushing the "leftmost" button will increase the brightness of the LED, pushing the "rightmost" button will decrease the brightness, and pushing the center button will turn the LED off.

The PWM Device

The documentation on the PWM device is poorly written, and contains errors. The steps for configuring and using the PWM for this project are:

1. Configure GPIO 18 for Alternate Function 5.
2. Configure the PWM Clock. ORDER IS IMPORTANT!
   (a) Write `0x5A000020` to the PWM Clock Control Register.
   (b) Repeatedly read the PWM Clock Control Register until bit 7 becomes '0.'
   (c) Write the divisor you want to the PWM Clock Divisor Register.
   (d) Write `0x5A000011` to the PWM Clock Control Register to select the oscillator and enable the clock.
3. Initialize `PWM_RNG1` to your desired cycle time (try 1023)
4. Initialize `PWM_DAT1` to zero
5. Configure and enable PWM Channel 1:
   - `MSEN1 = 0`

   - `USEF1 = 0`
   - `MODE1 = 0`
   - `PWEN1 = 1`

6. Enter main loop, which writes a number between 0 and 1023 to `PWM_DAT1` to change the brightness of the LED.

The description of the PWM Clock Manager is completely missing. It is, in fact, one of the General Purpose GPIO Clocks described in section 6.3 on page 105. The offset to the PWM Clock Control Register is `0xA0` and the offset for the PWM Clock Divisor Register is `0xA4`. Those offsets are relative to the base of the **GPIO clock manager device**, and not the GPIO device itself. The base address of the clock manager device is stored in the global variable `clkbase` by the `IO_map` function. So the addresses for the PWM clock registers are `clkbase + 0xA0` and `clkbase + 0xA4`. When selecting the clock source, I have only been able to select source 1 (oscillator) or source 6 (PLLD).

Some notes

The buttons and LEDs are active low. For the LEDs, this means that writing a low to the pin connected to the LED will turn the LED on. For the buttons, depressing the button will read low on the pin while the default button state will be high.

The *Raspberry Pi (BCM2835 ARM) Peripherals* manual (an be found on the course website) explains the functionality of the registers you will use mentioned above.

12.2.2 pcDuino

Using the wiring from our previous lab, one of the LEDs is connected to GPIO5. That pin can be reconfigured so that it is connected to the PWM device. You will modify the code from previous lab so that, instead of displaying the count on 3 LEDs, it uses the count to control the duty cycle of the PMW device, driving the LED on GPIO5.

Thus, pushing the "leftmost" button will increase the brightness of the LED, pushing the "rightmost" button will decrease the brightness, and pushing the center button will turn the LED off.

The PWM Device

The documentation on the PWM device is poorly written, and contains errors. The steps for configuring and using the PWM for this project are:

1. Configure GPIO Port B, pin 2 for function 010_2..
2. Configure the PWM device. to have a 100Hz period.
   (a) Write the correct value to the PWM_CH0_PERIOD register so that the PWM device has a period of 100 clock cycles and the duty cycle is zero.
   (b) Read the PWMCTL register and set the appropriate bits so that PWM0 is enabled in PWM mode and the prescaler is set to divide by 2400. becomes '0.'
3. Enter main loop, which writes a number between 0 and 100 to the lower 16 bits of PWM_CH0_PERIOD to change the brightness of the LED.

Lab 13

UART

13.1 Assignment

For this lab, you are to write a very simple UART driver.

13.2 Instructions

1. Create a file called `UART.S` and add functions to read and write bytes using the UART. At a minimum, you must define the following functions: `UART_put_byte` and `UART_get_byte`. You may also want to write functions to set the baud rate, data bits, and other parameters.

2. Edit `printf.S` so that you can write strings and integers. The `printf` function should use the functions provided in your `UART.S` file. (Note: If you wish, you can replace `printf.S` with `printf.c` and write the `printf` function in C rather than assembly.)

3. Modify your `main` so that it reads data and echoes it back to the user before entering the main loop, and once in the main loop, it prints the count through the UART every time the user presses a button.

Lab 14

Running Without an Operating System

14.1 Assignment

The purpose of this lab is to learn about "bare metal" programming. You will use code written for the previous two labs, modify and re-compile it to run without an operating system, then extend it. You will be provided with a startup function and a linker script.

14.2 Instructions

1. Get some bare metal code to run.
   (a) Download `baremetal.tgz` from the site provided by your instructor and extract the files:
   ```
   lpyeatt@rp01 $ tar xfz baremetal.tgz
   ```
   (b) Copy your code from the previous lab into the new directory.
   (c) Edit `Makefile` in the new directory, as necessary.
   (d) Run `make` to compile your code. You may need to edit some of the source code and/or the makefile to get it to compile. When it compiles properly, it will create a kernel image file.
   (e) Copy your original kernel image to a safe place, so that you can replace the original Linux kernel after testing your bare-metal code. If you lose the original `kernel.img` file, then you will be unable to boot Linux until you get a new copy.
   (f) Copy the bare metal kernel image to the appropriate location on your SBC.
   (g) Reboot your SBC and verify that your code works.
   (h) To get Linux running again, copy the original kernel image back to its original location.
2. LOTS of EXTRA CREDIT: Modify the UART driver and button debouncer so that they use interrupts.