Lab 3
Subroutines and ASCII Output

As programs get larger and more complicated, it is extremely important that they are divided into components, and how the components are connected is critical. In Lab 3, we will begin this process of dividing up the application. The application is first divided between program and data. The program itself is then divided into multiple components called subroutines. Depending upon the programming language and application, subroutines are also know as functions, modules, procedures, and methods. In Lab 3, we will develop subroutine v_asc, the first of many subroutines that we will eventually save in a library which becomes a toolbox of utilities.

Prototype

Almost every CPU ever designed has an instruction that "calls" the operating system for services. We've already used the SVC (service call) instruction to terminate application programs. Now we will use it to write to the display monitor. In following labs, we will use it to read from the keyboard, and it can also be used to read and write disk files and Input/Output data lines as well.

```
        .global   _start        @ Provide program starting address
                                  to linker
```

```
_start:  ldr     R1,=msgtxt      @ Set R1 pointing to message to be
                                 displayed
         mov     R2,#10          @ Number of bytes in message
         mov     R0,#1           @ Code for stdout (standard output,
                                 i.e., monitor)
         mov     R7,#4           @ Linux service command code to
                                 write string.
         svc     0               @ Issue command to display string on
                                 stdout
         mov     R0,#0           @ Exit Status code 0 for “normal
                                 completion"
         mov     R7,#1           @ Service command 1 will terminate
                                 this program
         svc     0               @ Issue Linux command to terminate
                                 program
         .data
msgtxt:  .ascii  "Hey there\n"   @ 10 character message (blank and \n
                                 each count as 1 char.)
         .end
```

Listing 3.0: Output a string of ASCII characters

In Listing 3.0, a second Linux service request is being introduced to write a string of bytes. With R7 set to 4, the “svc 0” service call will write to the device specified by register R0. A value of R0=1 indicates that the string should be written to the standard output device “stdout,” which is usually the computer’s display monitor. The memory location of the string is loaded into R1 and its length into R2. We will now stop using the exit status code service request to display answers. The second “svc 0” which has R7=1 will terminate the program with a status code of 0 (from R0) implying a successful performance.

I’ve also introduced two more assembler directives: .data and .ascii. The .ascii directive tells the assembler to place a string of ASCII characters into memory. Special control characters can be indicated by a sequence beginning with a back slash (\n represents “line feed,” hexadecimal value 0A). See Appendix D for some background on the

ASCII character set if you like. The .data directive enables the assembler (with the help of the linker) to separate the areas of the computer memory dedicated to instructions from that of data values. In this example, it is not necessary, but it is setting a pattern for later programming examples.

Go ahead and run this sample program. You can use the same command sequences (Listing 2.0), but you will not need the echo command (unless, of course, you want to see the exit status of zero). Try different text messages, but be sure to change the value in register R2 to match the length of the message.

Introductions

In Lab 3, I will introduce the ARM instructions and assembly language directives that are typically used to divide a program into components. We will also begin using more appropriate techniques for displaying data through the Linux "write" service routine.

	List of ARM instructions introduced in Lab 3		
3.0.2.	ldr	R1,=msgtxt	@ Set R1 pointing to message to be displayed
3.1.4.	bl	v_asc	@ Call subroutine (Branch and Link) to display text
3.1.22.	bx	LR	@ Return (Branch eXchange) to the calling program

	List of assembler directives introduced in Lab 3
.ascii	Put string of ASCII characters into memory
.data	Inform linker to group following code with other data in memory
.text	Inform linker to group following code with other instructions in memory

	List of Linux service calls introduced in Lab 3
4	Write array of bytes to device

	List of gdb debugging commands introduced in Lab 3
c	Continue
x	Dump memory

Principles

One of the principal hallmarks of the industrial revolution was the use of interchangeable parts in the manufacturing process. In a similar manner, subroutines and operating system services are building blocks for developing large sophisticated software applications. Both subroutines and services are predefined program segments that can be used over and over again by calling them from the main application program. In most cases, the exact actions performed by these building blocks can be modified slightly based on a set of input parameters referred to as “arguments” and are usually contained in registers. After performing their assigned tasks, subroutines and services return program control to the instruction following the point from which they were called.

Linux Services

We’ve already been using one Linux service to quit our programs. This service (having [R7]=1) is unique in that it does not return control to the instruction after the service call. The new service that we will be using ([R7]=4) will perform the task of writing to the display monitor and then returning control to the instruction following the service call.

One of the main responsibilities of an operating system, such as Linux, is to provide services for application programs. A large portion of these services involves reading and writing peripheral devices (display monitor, keyboard, mouse, network, etc.) and disk files (real spinning disks as well as solid-state memory devices). The calling program must provide Linux with the details of what is to be performed:

1. What is to be done
2. Which device is to written or read
3. Where the data (buffer) is in the program's memory
4. How much data is to be written or read

In the case of Linux, as well as most operating systems, this information is provided in the CPU's registers. For some devices such as a disk or external memory, another parameter providing the location on the disk is many times required as well.

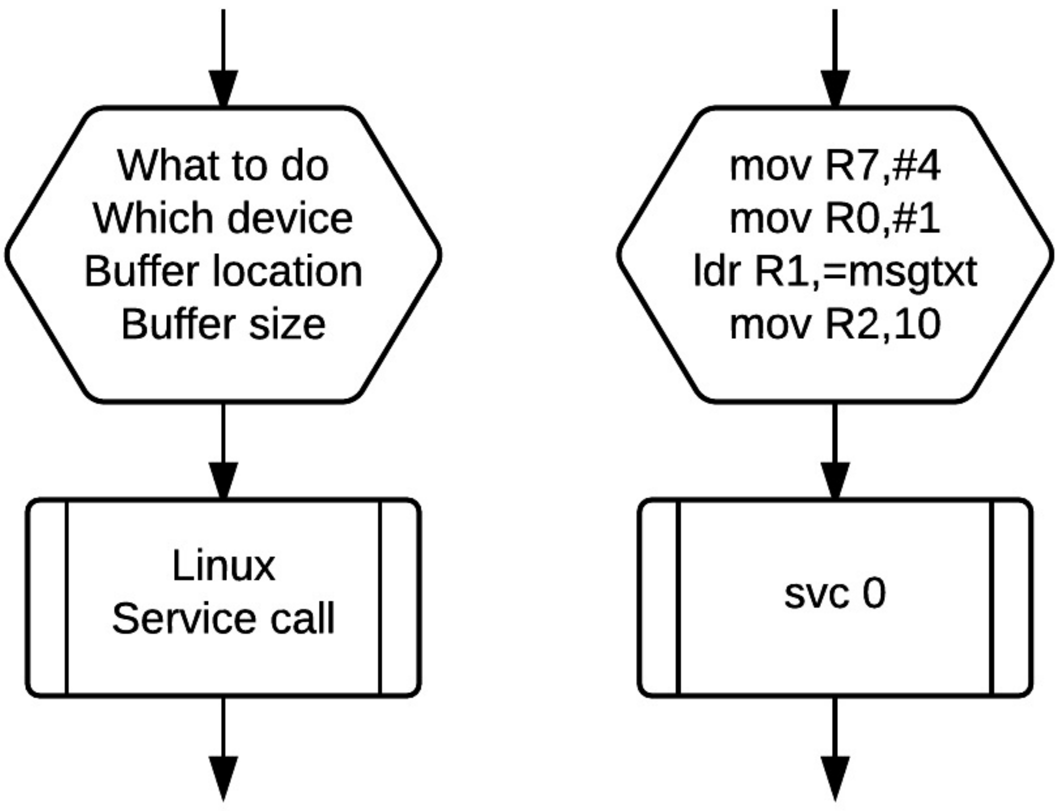

Figure 3.0: Linux service call to write to display monitor

A second responsibility of an operating system is to coordinate the sharing of a computer's resources among multiple tasks and users. This includes making sure each user receives its fair share of CPU time, memory, I/O device access, and other resources without interfering with other users. That's the main reason why in a Linux environment an assembly language program cannot read and write directly to devices.

Subroutines

A user application can and should be broken up into multiple building blocks. These subroutine building blocks are called in a manner similar to calling operating system services. In the ARM architecture the BL (branch and link) instruction calls subroutines in a manner similar to how the SVC (service call) instruction calls operating system services

A subroutine is a section of code that is "called" to perform a specific job. Examples of jobs a subroutine can perform:

- Display a number to the user
- Get keyboard input from the user
- Get input from a specific device such as a temperature sensor
- Change the speed of a motor
- Perform a particular type of analysis such as a least-squares fit of data

The advantages of using subroutines are many:

- Subroutines help organize the construction of the program.
- The code only takes up memory space once.
- It's less work to modify or correct one area of code rather than many areas.
- "Information hiding" occurs because one part of the program is unable to access data in another part and cannot accidentally change it.
- Division of programming assignments among different programmers is easier.

The disadvantages of subroutines are few.

- There is a slight performance degradation compared to "in-line code" due to the overhead of the call and return.
- It can lead to too much of a good thing: Too many tiny subroutines can lead to confusion.

Our first subroutine will be a very simple one. It will later be included in a library of subroutines which output various forms of data to the display monitor. Subroutine “v_asc” will display an ASCII message on the display monitor, a common occurrence in most computer programs. It’s so simple, you might ask, “Why bother at all, just use the Linux SVC directly and reduce the additional overhead of a subroutine essentially calling the same type of subroutine.” In a very small program with only a few calls to display, I would agree. However, for larger programs, a dedicated display subroutine provides a lot of flexibility from the maintenance perspective. Just for example, let’s say we have developed a program with thousands of calls to display, and now the “marketplace” requires that we send our display messages to a different device (one that the simple Linux call cannot perform). Wouldn’t it be more convenient to accommodate that change in one place in the subroutine’s code rather than hunt it down in the large program and try to successfully change it thousands of times?

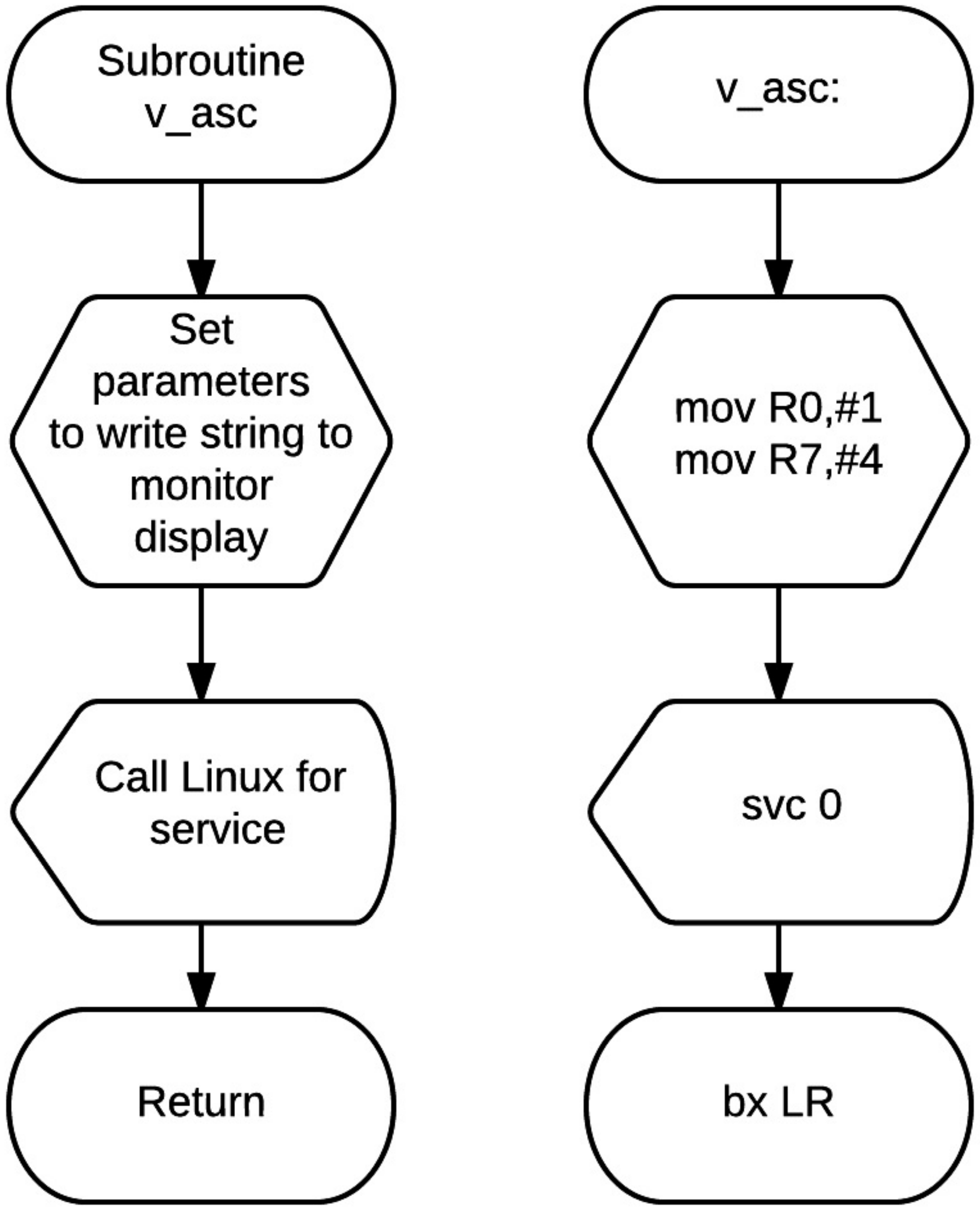

Figure 3.1: Subroutine v_asc displays messages

Figure 3.1 shows the structure of the v_asc subroutine that is called by the "main" program illustrated in Figure 3.2. These two components will replace the prototype program we just ran (Listing 3.0). When a program starts, Linux gives control to the main program identified by having the _start label. In general, a main program may call as many subroutines as it requires, and each of these subroutine can even call more subroutines, etc. There is even a special "recursive" type of subroutine that can call itself.

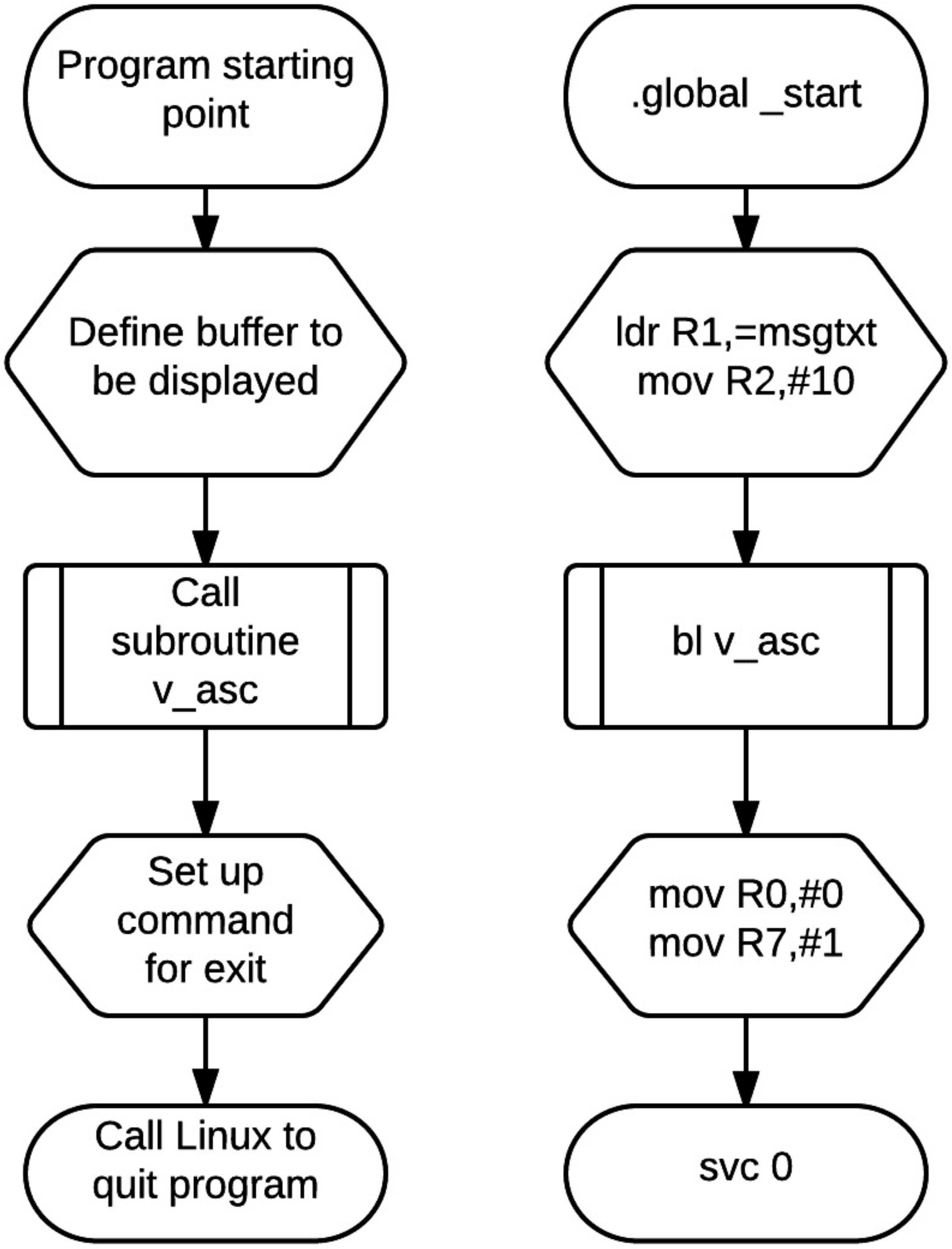

Figure 3.2: Main program calling subroutine v_asc

Functions, Methods, Procedures

The concept of a subroutine is also known by several other names. A "function" is a subroutine that not only performs a specified task, but also returns a value (usually in one of the registers). Some higher level languages such as C use the name "function," while the term "procedure" or "method" is the name used in other languages. No matter what the name, the concept is the same: a block of code that performs a predefined task (usually customized by data in registers), and it may return a value.

Interrupts

A third type of subroutine commonly used in embedded systems applications is the interrupt service routine. There are basically two types of interrupts on most CPUs:

- Software: The SVC instruction we've been using to call Linux kernel services in every program in this book. The SVC (service call) instruction is also known as the SWI (software interrupt).
- Hardware: A change in state of an I/O device essentially "calls" a device driver (subroutine) using an "instruction" that behaves almost identical to the SVC instruction.

Although interrupt handling is very important to embedded systems development, I will not be covering it in this book because its use is closely guarded by Linux and every other operating system supporting multiple users running on the same computer. However, once you become proficient in writing subroutines, you will be well on your way to writing good interrupt handlers.

Link Register (LR)

In the ARM architecture, a subroutine is typically called using the branch and link (BL) instruction which loads the address of the instruction following the BL into the LR link register. The idea is to provide the subroutine with a return address. Over the decades, this has been done in many different CPU architectures with instructions named BALR (Branch and Link Register), LMJ (Load Modifier and Jump), SLJ (Store Location and Jump), RJ (Return Jump), and of course CALL (not an acronym, just plain "call"). BALR and LMJ provided the return address in a register, while SLJ and RJ put it into the first word of the subroutine's memory, and CALL saves the return address onto the "stack," a concept to be introduced in Lab 4.

A subroutine must be careful to not accidentally lose its return address. If one subroutine calls another, the contents of the LR has to be saved beforehand otherwise the second BL will overwrite it. Also, the LR is a second name for register R14 which can be used as a general purpose accumulator.

Coding and Debugging

As Listing 3.1 illustrates, the operation of every subroutine should be locally documented in comments:

1. Name of the subroutine and what it does
2. Resources it needs (R1 points to the string to be displayed and R2 contains the length of the string).
3. How the subroutine is going to return to the program that called it (LR register contains the return address)
4. What is in the registers when the subroutine is complete and returns to the calling program

```
            .global    _start        @ Provide program starting
```

```
                                        address to linker
_start:   ldr      R1,=msgtxt           @ Set R1 pointing to message to be
                                        displayed
          mov      R2,#10               @ Number of bytes in message
          bl       v_asc                @ Call subroutine to view text
                                        string in ASCII.
          mov      R0,#0                @ Exit Status code of 0 for “normal
                                        completion"


          mov      R7,#1                @ Service command code 1
                                        terminates program
          svc      0                    @ Issue Linux command to
                                        terminate program

          .data
msgtxt:   .ascii   "Hey                 @ 10 character message (blank and
                   there\n"             \n each count as 1 char.)
          .text

@         Subroutine v_asc will display a string of characters
@                  R1: Points to beginning of ASCII string
@                  R2: Contains length of string in bytes
@                  LR: Contains the return address
@                  Registers R0 and R7 will be used by v_asc and
                   not saved

v_asc:    mov      R0,#1                @ Code for stdout (standard
                                        output, i.e., monitor)
          mov      R7,#4                @ Linux service command code to
                                        write string.
          svc      0                    @ Issue command to display string
                                        on stdout
          bx       LR                   @ Return to the calling program
          .end
```

Listing 3.1: Setting up a subroutine to output ASCII string

Program Line Numbers

In Listing 3.1, I've included the line number for each line of the text file containing the program and data. These numbers are not actually in the text file itself, but usually provided by the editor, assembler, and debugger. Notice how I started from one rather than zero. I did this to match the values provided by the editor, assembler, and debugger.

Do I really start counting from zero? No. Of course not. So why did I do it in identifying the labs, listings, and figures in this book? Simply to get you accustomed to the idea that in computer software and hardware, there are many situations in which zero is the first value and not one.

Two additional instructions are introduced in Listing 3.1:

- Branch and link (BL): Calls a subroutine by saving the current program location (PC) in the link register (LR) and then branching (also known as jumping in many CPU architectures) to a another location in the program.
- Branch exchange (BX): Returns from a subroutine by branching to the program location specified in a register.

In labs 1 and 2, we used the exit status code to display the results of our calculations. From now on, we'll display our results using more appropriate techniques. We will also use the GNU gdb debugger to see intermediate values leading up to producing the final values to be displayed.

Since this lab is introducing the mechanics of subroutines and an ASCII display in particular, the two values we'll be examining are the following:

1. The contents of the link register (before and after the BL instruction)
2. The ASCII contents of the text message

Start the debugger like you did in Lab 2 (~$ gdb model), and then enter the l(ist) command to display the first part of the program.

```
(gdb) l
1               .global   _start          @ Provide program starting address to linker
2     _start:   ldr       R1,=msgtxt      @ Set R1 pointing to message to be displayed
3               mov       R2,#10          @ Number of bytes in message
4               bl        v_asc           @ Call subroutine to view text in ASCII.
5               mov       R0,#0           @ Exit Status code: “normal completion"
6               mov       R7,#1           @ Command code 1 terminates program
7               svc       0               @ Issue Linux command to terminate program
8
9               .data
10    msgtxt:   .ascii    "Hey there\n"   @ 10 character message.
(gdb)l
11              .text
12
13    @         Subroutine v_asc will display a string of characters
14    @         R1: Points to beginning of ASCII string
15    @         R2: Contains length of string in bytes
16    @         LR: Contains the return address
17    @         Registers R0 and R7 will be used by v_asc and not saved
18
19    v_asc:    mov       R0,#1           @ Code for stdout (standard output)
20              mov       R7,#4           @ Linux service command to write string.
```

Listing 3.2: List source code line numbers using debugger

Next, set two breakpoints in the program using the "b" command to suspend program execution at two locations:

1. At line 4 so we can examine the registers before subroutine v_asc is called
2. At line line 19 so that we can examine the return address in the LR link register

Then start the program execution with the "run" command as illustrated in Listing 3.3. It will stop at the first breakpoint it encounters. At that point, enter the "i r" (info registers) command to the debugger.

```
(gdb) b 19
Breakpoint 1 at 0x8090: file model.s, line 19.
(gdb) b 4
Breakpoint 2 at 0x807c: file model.s, line 4.
(gdb)
(gdb) run
Starting program: /home/pi/model
Breakpoint 2, _start () at model.s:4
4       bl          v_asc        @ Call subroutine to view text string
                                 in ASCII.
(gdb) i r
r0      0x0         0
r1      0x100a0     65696
r2      0xa         10
r3      0x0         0
r4      0x0         0
r5      0x0         0
r6      0x0         0
r7      0x0         0
r8      0x0         0
r9      0x0         0
r10     0x0         0
r11     0x0         0
r12     0x0         0
```

```
sp      0x7efff860    0x7efff860
lr      0x0           0
pc      0x807c        0x807c <_start+8>
cpsr    0x10          16
(gdb)
```

Listing 3.3: Register contents before calling v_asc

Notice the following register contents shown in Listing 3.3:

1. The contents of the PC (program counter) is hex 807C before the call. This is the memory location of the BL instruction which is calling v_asc.
2. The contents of the LR link register is 0 before the call and is hex 8080 after the call. The 8080 address is (807C plus 4 equals 8080) the address of the instruction immediately after the BL subroutine call. We add 4 because in the ARM architecture each instruction is 4 bytes long. Note: The ARM CPU can be switched into a special 2-byte instruction format mode known as Thumb code (described in Lab 17 and Appendix N).

Continue the program execution by entering the “c” command as shown in Listing 3.4. The program will now be suspended at the next breakpoint which is at line 20 just after entering the subroutine.

```
(gdb) c
Continuing.

Breakpoint 1, v_asc () at model.s:20

20      mov           R7,#4         @ Linux service command code to
                                    write string.

(gdb) i r
r0      0x1           1
r1      0x100a0       65696
r2      0xa           10
```

```
r3      0x0           0
r4      0x0           0
r5      0x0           0
r6      0x0           0
r7      0x0           0
r8      0x0           0
r9      0x0           0
r10     0x0           0
r11     0x0           0
r12     0x0           0

sp      0x7efff860    0x7efff860
lr      0x8080        32896
pc      0x8090        0x8090 <v_asc+4>
cpsr    0x10          16
(gdb)
```

Listing 3.4: Register contents inside subroutine v_asc

While we're temporarily suspended inside the subroutine v_asc, let's take a look at the data to be displayed. Note that register R1 indicates the data to be displayed begins at address 0x100a0. The "x" command allows "dumping" data stored in memory. Use the "gdb help" command or see Appendix J for more options. The following are just a few of the options possible with the "x" command:

- x/10cb 0x100A0 — Display memory at hex address 100a0 as characters 10 bytes long
- x/10xb 0x100a0 — Display memory at hex address 100a0 in hexadecimal 10 bytes long
- x/1cs 0x100a0 — Display memory at hex address 100a0 as characters 1 string long (ends with byte of zero)

```
(gdb) x/10cb 0x100a0
0x100a0
<msgtxt>:     72 'H' 101 'e' 121 'y' 32 ' ' 116 't' 104 'h' 101 'e'114 'r'
```

```
0x100a8           101 'e' 10 '\n'
<msgtxt+8>:
(gdb) x/10xb 0x100a0
0x100a0
<msgtxt>:         0x48 0x65 0x79 0x20 0x74 0x68 0x65 0x72
0x100a8
<msgtxt+8>:       0x65 0x0a
(gdb) x/1cs 0x100a0
0x100a0
<msgtxt>:         "Hey there\nA\025"
```

Listing 3.5: Examples using debug “x” command

After getting the above memory dumps using the gdb “x” commands, enter “c” to the (gdb) prompt to continue executing the program instructions. It will then display the message, subroutine v_asc will return to where it was called, and the program will “quit.” You will still be in the gdb debugger, so you could restart the whole program again by entering “run” or you could exit to the Linux prompt by entering the “q” command.

Maintenance

Review Questions

1. What are two important functions provided by an operating system such as Linux?
2. Almost every CPU ever designed has a way to call subroutines which involves saving the current PC (Program Counter) in either memory or a register so that the subroutine knows where to return control. What is the disadvantage of saving the return address in a specific memory location, and what is

the disadvantage of saving the return address in a register?

3. Using the BL instruction on the ARM CPU, what is the problem of one subroutine calling another?
4. A recursive subroutine is one that calls itself. A popular example of using a recursive subroutine is calculating a factorial where n! = n(n-1)(n-2)..1. What must be done with the return address in the link register to make recursive subroutines work?
5. * Which register(s) will be changed after executing each of the following instructions, and what are their new values? Assume that each instruction begins with the following contents: R0 = 0, R1 = 1, R2 = 2, LR = 36000, and PC = 32892.
    a. ORR R0, R3
    b. ASR R3, #1
    c. BL 34000
    d. BX LR
    e. ADD PC, #4

Programming Exercises

1. Write subroutine "v_byte" to output exactly one ASCII character that is contained in the low byte (bits 7-0) in register R0. Register R2 will not need to be specified because it is always going to be 1.
2. Write a subroutine similar to v_asc, but embed the output in brackets. For example if R1 contains the address of the word Computer, then [Computer] will be displayed. You may want your new subroutine to call v_asc and v_byte. Be careful to save the LR link register contents.

programs in this book as well as the vast majority of all current computer applications, the programmer "tells" the program how to work. Between these two extremes, there is also a variety of non-procedural "techniques" such as database and data flow manipulation languages.

1.5 When updating a line of source code, should the comment on the line be updated as well?

Yes, usually. However, sometimes the comment is right, but the code was wrong. For example, the comment said why the line of code was present, but the code did not work. The worst case is when someone changed what the code was supposed to be doing, but left the old comment which is now irrelevant and much worse than no comment at all.

2.8 Which register will be changed after executing each of the following instructions, and what is its new value? Assume that each instruction begins with the following contents: R0 = 0, R1 = 1, R2 = 2, R3 = 3, R4 = 4, and R5 = 5.

a. AND R2,R3 changes nothing (R2 is "changed" to 2, but it already was 2)
b. ADD R4, R5 @ changes contents of R4 to 5
c. LSL R1, #4 @ changes value of R1 to 16
d. EOR R3, #1 @ changes value of R3 to 2
e. MUL R4, R5 @ changes value of R4 to 20
f. SVC 0 @ changes nothing or anything (depends on the service call)

3.5 Which register(s) (including Z-flag in CPSR) will be changed after executing each of the following instructions, and what are their new values? Assume that each instruction begins with the following contents: R0 = 0, R1 = 1, R2 = 2, SP = 36010, PC = 32000, and condition code Z=1.

a. ORR R0, R3 @ changes [R0] = 3 and [PC] = 32896
b. ASR R3, #1 @ changes [R3] = 1 and [PC] = 32896
c. BL 34000 @ changes [LR] = 32896 and [PC] = 34000
d. BX LR @ changes [PC] = 36000
e. ADD PC, #4 @ changes [PC] = 32904 Note: If you thought the [PC] should be 32896, then you didn't include 8 bytes for "pipe lining."

4.2. The CMP instruction sets the NZCV status bits. Why do you think its mnemonic isn't cmps and would cmps work also?

The CMP instruction is like a SUBS instruction, except the difference of two operands is not placed into a result register. In other words, the only thing the CMP instruction does is set the NZCV flags (and update the PC, of course). The "S" suffix is always assumed, and therefore not necessary. The assembler even flags an error if CMPS is given. Could the assembler have accepted CMPS? Sure, but it doesn't. The Thumb arithmetic instructions are the same way. All set the NZCV status bits, so ADDS, SUBS, etc. are not allowed.

5.4 The instruction mov R1,#5120 should not be possible (due to the limitations noted in the previous question). What machine code instruction does the assembler generate to make it work? Clue: 5120 = 4096+1024.

MOV is one of the arithmetic/logic instructions that has its immediate value represented by an 8-bit base M rotated to the right by a 4-bit shift count S. The decimal value 4096 is 1 2^{12}, while1024 is 1 2^{10}. Adding the two in binary is 0b1010000000000, which can be represented as 0b101 shifted left 10 bit position or rotated right 22 bit positions. Therefore the lower 12 bits of the MOV instruction which contains the shift count 11 and base 0b101 is 0xB05. Other possibilities that would work are 0xC14 and 0xD50.

6.2. Compare the merits of a byte-addressable computer architecture to one that is word addressable.

Word-addressable:

- Don't have to be concerned about big and little endian.
- Don't have to worry about alignment issues. Many CPUs will either degrade performance or not load/store "word" instructions on addresses that are not multiples of four bytes.
- For CISC architectures that have the memory address inside the instruction, much more memory can be addressed. For example, the Univac 1108 had an address space of 18 bits, which is 256K words. This equated to 1.5M of 6-bit Fieldata characters (bytes). Some instructions even had a 16-bit limitation which is 65K words. Although these sizes seem minuscule today, in the 1960s, they were impressive.

Byte-addressable:

- No complicated sub-word instruction options Many word-addressable CPUs had a special field within their instruction format to select a particular sixth or quarter word. Some didn't even have that, so bytes had to be masked and shifted into places.
- A variety of integer sizes were available: bytes, words, half-words, and double words.
- Packed-decimal arithmetic was generally available.

7.4. In Listing 7.3, the CMP R3,#0 instruction on line 23 can be eliminated if line 16 is modified. What modification is this and why is it probably a bad idea even though it would function perfectly?

The MOVS instruction could set the NZCV status bits. However, there is a potential maintenance problem with having too many intervening instructions between the setting of the status bits and their ultimate use. Consider the possibility of a future maintenance programmer not noticing where the status bits are set and used, and then inserts a new instruction between them that changes the status bits. I do, however, like the fact a "move" instruction can optionally set the status bits. Most CPUs do not have this capability.

8.4. The LDR and STR instructions allow a negative offset. Is this negative direction set with two's complement or sign and magnitude format?

Technically, it's in the sign and magnitude format, but the sign and magnitude are not adjacent to each other. The "sign" is actually the U-bit in the LDR/STR instruction format, where U=1 means positive and U=0 means negative.

9.2. How is a macro different from a subroutine?

- A macro is called while the assembler is running, and a subroutine is called when the application program (being written) is running.
- A macro generates text lines that will later be "assembled," while a subroutine works with numbers and text of the running application.
- Each macro call makes the program physically larger and take up more memory, while subroutines generally reduce memory requirements by eliminating duplicate code.

9.3. Give an example of a useful macro that generates neither any instructions nor any data.

Just a few examples are listed below:

- .align statement to indicate a word or double word boundary will be used next.
- .set or .equ statements to assign values to constants used at assembly time.

- .text or .data to indicate where following instructions and data are to be placed.
- .arm or .thumb to indicate the instruction format for following code.

9.5. An emulator doesn't have to be programmed in assembly language. What would be the advantage of writing one in C or another higher level language?

> The C programming language was developed to provide portability to system software. So, an emulator for the ARM could then be run on almost any computer. If the emulator was written in a hardware description language, such as VHDL or Verilog, then it would essentially become a fairly fast substitute for a real ARM CPU or be part of a reconfigurable computing device.

10.1. What are the advantages of fixed point over floating point?

- Fixed point is exact. No error is present.
- Fixed point arithmetic is very fast.
- It is very easy and fast to convert between character representation and fixed point.

11.1 "By hand, without a computer," convert the following decimal fractions into binary and provide the answers in hexadecimal.

a. 0.5_{10} = ½ = $0.1000_2 = 0.8_{16}$
b. $0.625_{10} = 0.1010_2 = 0.A_{16}$
c. $0.25_{10} = 0.0100_2 = 0.4_{16}$
d. $0.03125_{10} = 0.00001_2 = 0.08_{16}$
e. $0.0078125_{10} = 0.0000001_2 = 0.02_{16}$

11.2 "By hand, without a computer," convert the following binary fractions from hexadecimal back into real numbers in base 10.

a. $.C0000000_{16} = 0.11_2 = ½ + ¼ = ¾ = 0.75_{10}$
b. $.E0000000_{16} = 0.111_2 = 0.875_{10}$
c. $.10000000_{16} = 0.0001_2 = 0.0625_{10}$
d. $.50000000_{16} = 0.0101_2 = 0.3125_{10}$

11.3 Using a calculator for division by powers of 2, convert the following binary fractions from hexadecimal into the real number that each is "approaching" in base 10.

a. $.33333333_{16} = 0.00110011001100110011001100110011_2$ => 0.2_{10}
b. $.66666666_{16}$ => 0.4_{10}
c. $.CCCCCCCC_{16}$ => 0.8_{10}
d. $.E6666666_{16}$ => 0.9_{10}

12.1 Convert the following real numbers into single precision IEEE 754 floating point and provide the answers in hexadecimal.

a. 128.0 is 43000000 in floating point
b. 9.25 is 41140000 in floating point
c. -9.25 is C1140000 in floating point
d. 0.03125 is 3D000000 in floating point
e. 128.03125 is 43000800 in floating point
f. 0.0 is 00000000 in floating point
g. -0.0 is 80000000 in floating point

12.2 Convert the following IEEE 754 floating point numbers back into real numbers in base 10.

a. 42a80000 is 84.0
b. C1A80000 is -21.0
c. 424C8000 is 51.125
d. BF100000 is -0.5625
e. 3DCCCCCD is 0.1

13.1 Why does the v_flt program display 0.5 when it should display 0 for a floating point number consisting of all 32 zero bits?

> Hint: Is the floating point zero normalized or not? Does subroutine v_flt handle special cases of IEEE 754?

13.2 By examining Figure 13.1, what is the smallest absolute value non-zero normalized number?

> Hint: Convert $1 \times 2^{1-127}$ to decimal.

14.3. Is getting that extra 1-bit of precision in the significant more important to the single precision, double precision, or half precision format numbers?

> The half-precision floating point format is a 16-bit package containing a sign bit, five bits for the biased exponent, and ten bits for the significant. Note: Half-precision is only used for storage and is not supported for computation within IEEE 754 devices. That extra bit obtained by not taking up a bit-position for the most significant bit improves the resolution of the Half precision format the most.

- Half precision (significant is 10 bits): 1 in 2^{10}
- Single precision (significant is 23 bits): 1 in 2^{23}
- Double precision (significant is 52 bits): 1 in 2^{52}

15.3. If the lane size for integer addition within NEON could be one bit wide, it would be exactly the same as which logical operation?

> The exclusive OR operation. Think back to your digital electronics days. How do you build a “half adder”?

16.3. What is a principal danger in using “pass by reference"?

> One of the hallmarks of object oriented programming is “information hiding.” If a part of a program does not need access to a part of the data, don’t give it access. “Pass by

reference" provides the location of the data to the subroutine, and if the subroutine makes a mistake, it can write over the original source of the data. In "pass by value," only a copy of the original data is sent as an argument to a subroutine. Of course, if all programs and subroutines worked perfectly, none of this would be a concern.

17.4. We switch between Thumb mode and ARM mode with the BX instruction. Although we could set the T-bit in the CPSR directly, why would this lead to problems if we did?

The quick and easy answer is that the ARM hardware reference manuals say don't do it, and a special way has been set up to perform the switch using the BX instruction. But also remember that the ARM has a "pipelined" architecture where by the time an instruction is actually executed, the following two instructions have already been fetched and are being prepared to execute. The 16-bit/32-bit instruction mode would be switched while instructions are in an intermediate state. The BX, like all branch instructions that execute, will clear the pipeline.

17.5. What simple change can be made to the c_int subroutine so that it works regardless of whether it is called with the T-bit already set or not?

Define a second entry point for a call from a routine already in Thumb mode. Then make sure that bit 0 of the LR is set so that the BX return instruction will leave the CPSR in Thumb mode.

18.1 The loop which multiplies the significant by either 10.0 or 0.1 to accommodate the base ten exponent results in some loss of precision in the conversion of ASCII character format to floating point. What two relatively simple modifications to that technique can greatly improve the resulting precision?

1. Do the multiplication using double precision.
2. Use a table of pre-generated precise powers of ten.

18.2 Why will multiplying by 0.1 always result in a loss of precision in binary computers?

> Base ten is not a multiple of base 2, like base 8 and base 16 are multiples. Some numbers like 0.1 cannot exactly be represented as a base two fraction for the same reason 1/3 is $0.333333..._{10}$.

19.2 Name four operations available in NEON that are not available in VFPv3.

> Logical, integer, shift, and count leading zeroes