Advanced Assembly, Branching, and Monitor Utilities

2.1 Objectives:

There are several different ways for an instruction to form effective addresses to acquire data, called addressing modes. One of these addressing modes uses the Tiva’s decision-making capability that is referred to as branching. When you complete this lab, you should be able to:

- Understand how to use several assembler directives.
- Write simple assembly programs to perform multi-byte addition and subtraction.
- Use the status flags with conditional jump instructions.
- Mask and test individual bits using logical AND/OR instructions.
- Input from the keyboard and output to the terminal screen using monitor utilities.

2.2 Related material to read:

- Text, Chapters 3-6 in General
  - Section 3.6, Assembly Directives
  - Section 6.2, Branch Instructions
  - Sections 6.4, Various Looping Constructs
- Text, Chapter 7: See Flowchart examples
- Lab Appendix C (ASCII Table)
- Lab Appendix E (Flow Chart Details)
- Text, pp. v-vi: Assembly Instructions

2.3 Assembler Directives:
There are a few assembler directives we need to discuss before you continue writing programs. These are also described in Section 3.6 of the text. Refer to the sample program in Figure 2.1 for reference while you are going through this section. This program makes use of four of the most common directives you will be using throughout the semester and is designed to copy a table from one location to another location in the internal RAM (0x2000.0400 – 0x2000.041F). There are several other directives that can be found under the μVision help menu. Some of these we will use and discuss later in the semester, so for now, just get to know the EQU, DCB, SPACE, and END directives discussed below.

**EQU**

The EQU directive is used to assign a value to a symbol definition. For example, notice in the Data Section of the program below that there are two symbol definitions: OFFSET and FIRST. The OFFSET symbol is assigned a value of 0x10 and the symbol FIRST is assigned the value 0x2000.0400. At build time, the assembler will “replace” any symbol definition with its corresponding value. For example, the corresponding instruction on line 58, `STRB R0,[R1,#OFFSET]` is the same instruction as `STR R0,[R1,#0x10]`. Using symbol definitions for subroutine names and for values used in multiple places not only makes your code easier to follow, but also allows you to make a single change to a value versus having to make multiple changes everywhere the value is used in the code.

**DCB**

The DCB directive stands for Define Constant Byte, and is used for defining the value of a byte (or bytes) at a given memory location. Let’s look at the same example program. In the data section, you notice the instruction `CTR1 DCB 0x10`. “CTR1” (our name for “counter 1”) is the label, and 0x10 is the byte value. Is this different than the EQU directive? Yes! The EQU directive assigns a value to a label and is only used by the assembler at build time, replacing each instance of the label with its value; however, in this case, the CTR1 label does not equal a value of 0x10; but instead refers to the memory address where the value 0x10 is stored. If you were to make a project that includes code in Figure 2.1 and Startup.s code, download this project to your Tiva, and look at memory location 0x3AC, it would contain the value 0x10. Using the label CTR1 in your program code refers to the memory address 0x3AC, where the value 0x10 is stored. Now look at the following line in the program. You notice the directive `MSG DCB "Copying Table..."`. This directive will store the ASCII byte values of Copying Table... into memory when the program is downloaded before program execution begins. The open quote and closed quote marks are interpreted as the delimiter and are not included as part of the stored ASCII bytes. At what addresses are these ASCII bytes located? The address where the first ASCII byte value (‘C’) is stored is 0x3AD. So the message Copying Table... is located at addresses 0x3AD – 0x3BC. If you were to load this program to your Tiva board and examine memory 0x3AD – 0x3BC, you would see the ASCII byte values of the message Copying Table.... It is also important to note that instead of using multiple DCB directives for a sequence of byte definitions (that don’t need individual labels) you can simply use a comma or space and define multiple bytes using the DCB directive once. For example, the two directives following the directive `MSG DCB "Copying Table..."` could be replaced by the directive `DCB 0x0D,0x04`. 

-2-
SPACE

The SPACE directive is used to reserve a number of bytes of memory, based on the argument. This is similar to DCB except an argument of 4 reserves 4 bytes of memory, instead of storing the value of 4 in memory.

END

Finally, the END directive indicates the logical end of a source program. Any statement following the END directive will be ignored by the assembler. The END directive is not necessary for successful compilation of your code; however it is good practice to use the END directive to show where your source code ends.

As mentioned before, there are many other directives available for the assembler. The four mentioned above will be the ones you use regularly for programming; other directives for different purposes will be introduced later in the semester. For now, make sure you fully understand the directives above and how to use them.

;**************************************************************
; Program_Directives.s
; Copies the table from one location to another memory location.
; Directives and Addressing modes are explained with this program.
;**************************************************************
;**************************************************************
; EQU Directives
; These directives do not allocate memory
;**************************************************************
;LABEL DIRECTIVE VALUE COMMENT
OFFSET EQU 0x10
FIRST EQU 0x2000400
;**************************************************************
; Directives - This Data Section is part of the code
; It is in the read only section so values cannot be changed.
;**************************************************************
;LABEL DIRECTIVE VALUE COMMENT
AREA sdata, DATA, READONLY
THUMB
CTR1 DCB 0x10
MSG DCB "Copying table..."
DCB 0x0D
DCB 0x04
;**************************************************************
; Program section
;**************************************************************
;LABEL DIRECTIVE VALUE COMMENT
AREA main, READONLY, CODE
THUMB
EXTERN OutStr ; Reference external subroutine
EXPORT __main ; Make available
Figure 2.1: Program_Directives.s. It is designed to help you understand various assembler directives. Also it makes efficient use of different addressing modes.

Program_Directives.s

In Program_Directives.s above, ‘loop1’ creates a table in the address range 0x2000.0400 - 0x2000.040F. These address locations contain 0x00 - 0x0F at the end of ‘loop1’. The second loop, ('loop2') copies the table in the address range 0x2000.0400 - 0x2000.040F to the address locations 0x2000.0410 - 0x2000.041F.

2.4 Branching and Relative Addressing:

Relative addressing is used only by branch instructions, and branch instructions use only the relative addressing mode. A branch instruction makes a decision on whether or not to alter program flow based on information in the C, Z, N, and V bits of the Program Status Register (PSR). See Section 6.1 of the text for more information on this.

The branch instruction you will probably use the most is BEQ or branch on equal to zero. In a program you will frequently test a number to see if it equals another number. One way you could do this is by subtracting one number from another and leaving the result in a register. This can update the condition codes. Compare instructions do the same thing, but do not change the value in the register. If the numbers are equal, the Z bit will be set. In other words, if the result of an operation is zero, the Z bit will be 1. The
BEQ instruction can then be used to either jump somewhere else in the program or continue with the next instruction. Refer to the code in Figure 2.2 following.

```
start    LDR R0,=0x20000900
         LDR R1,=0x20000910
         LDR R0,[R0]
         LDR R1,[R1]
         CMP R0,R1
         BEQ done
         ADD R0,R0
         done
         WFI
         B done
```

**Figure 2.2:** Branching code example explaining the use of BEQ instruction.

This code loads a number into register R0 from memory location 0x2000.0900, then compares it to a number in memory location 0x2000.0910, then tests the result. If they are the same, the program jumps to the label 'done'. If not, the next instruction is executed. The **program relative address** jumped to, defined by 'done' is computed when the program is assembled. The instruction B, branch always is an unconditional jump, i.e., you always branch on this instruction.

Many times a program will perform an operation a certain number of times in a loop and then continue. It would be nice to just say 'for i = 1:100 ...', but this is assembly language, so it is not so easy. One way to implement a loop is to initialize a counter; then decrement it each time the loop is executed. The decrement instruction is followed with a BNE, branch on not equal to zero, which causes the program to jump back to the top of the loop. When the counter hits zero, the program executes the next instruction after the loop.

There are several branch instructions in the Tiva instruction set. Refer to Section 6.2 of the text to see what they are and how they are used. Be sure that the flags are being set the way you think they are by the instruction before the branch.

### 2.5 Bit Masking & Packing:

Often the microprocessor makes a decision based only on some of the bits of the data rather than the whole byte. Tiva has instructions that perform bit wise logical operations of AND, OR, and EXCLUSIVE OR that are useful for this. For example, if you want to test bit 2 of a number, you can AND the number with 0x04, which is 00000100 in binary. Bit 2 of the number will stay the same, but every other bit will be set to zero. This is called **bit masking**.

<table>
<thead>
<tr>
<th>Address 0x2000.0314</th>
<th>Mask</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 1 0 1</td>
<td>0 0 0 0 0 1 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
LDR R0, =0x20000314
LDR R1, [R0]
AND R2, R1, #0x04

Figure 2.3: Example of bit masking address 0x2000.0314 with 0x04 to find value of bit 2 of the address. As seen from the code, the result is stored in R2.

The AND operation is useful for checking the value of one bit, or forcing individual bits to zero. What operation would you use to change individual bits to a one?

Sometimes you'll need to test all of the bits in a byte one at a time. The *rotate* instructions are useful for this because there are instructions to shift either the most, or the least significant bits, into the carry or 'C' condition code bit. You can then branch based on the result.

### 2.6 Monitor Utility Subroutines:

The *monitor utility subroutines* are a set of routines that allow communication between the Tiva board and a serial window. Three of them are introduced this week. The **InChar** subroutine inputs the ASCII character from the keyboard of the PC to register R0. For example, typing 'A' would load 0x41 into register R0. The subroutine **Inchar** loops until you press a key. Similarily, the **OutChar** subroutine displays the single ASCII character in register R0.

The program in Figure 2.4 shows how these routines could be used. Type up this program and run it. What happens? We will go into more details of subroutines in the next lab, but for now think of **BL** as an unconditional jump with a way to get back where it came from. The **CMP** instruction in Line 2 compares the character input to 0x20, which is the ASCII code for the SPACEBAR. This program echoes every key you hit using OutChar until you hit SPACEBAR. Refer to Table 2.7 for the ASCII representations.

```
get  BL   InChar
CMP  R0, #0x20
BEQ  done
BL   OutChar
B    get
done B  done
```

Figure 2.4: Utility subroutine example explaining **InChar** and **OutChar** subroutines.

Another useful subroutine is **OutStr**. This subroutine outputs a string of ASCII bytes pointed to by the address in index register R0 until the *end of transmission* character, 0x04, is reached. If you want to display different messages to the terminal screen when certain conditions are met, you can have the string stored in memory with the 0x04 after the last character. Remember that we have already used this subroutine in the program in Figure 2.1. This subroutine prints ‘Copyiing table…’ on the screen. The program stores this message in the memory and the starting address of this message has the label ‘MSG’. On the following line is 0x0D, which is ASCII for a Carriage Return (new line). The end of transmission character, 0x04, follows the actual message. In loop
2, the address of ‘MSG’ is loaded in index register R0 and in the following line, this message is printed on the screen using \texttt{OutStr} subroutine.

### 2.7 Setting up Termite Terminal Emulator

To see the output of the \texttt{OutStr} subroutine, or to send characters to the board using \texttt{InChar}, you will need a terminal window. While there are many terminal emulators out there (HyperTerminal, PuTTY), we have chosen to use Termite from CompuPhase due to its simplicity.

First find the communication port (COM Port) your board is using on your PC.

- Connect your board to the PC using the supplied USB cable.
- Go to \textit{Start}, \textit{Right Click Computer} > \textit{Manage} This will bring up the Computer Management Window. Select \textit{Device Manager} under \textit{System Tools}.
- Expand \textit{Ports (COM & LPT)} and find \textit{Stellaris Virtual Serial Port (COM x)} where x is the number of the communication port. Remember this number.

Next open Termite and change the \textit{Settings} to use the correct COM port.

- Click \textit{Settings} and under \textit{Port}, select the number your board is using.
- Set Transmitted text to append nothing.
- The rest of the settings should remain default at 9600 baud, 8-bit, 1-stop, no parity. (You will learn about these in future lectures)
2.8 Procedure:

Before you come to the lab, do all the calculations in 1. below manually. You should also write program for procedure 1.a. below before coming to the lab. At the start of the lab, show the TA your results, programs, and flowcharts for all parts of the lab. Refer to Appendix E of this manual to understand the basics of how to write the flowcharts. Show the working programs to the TA before you leave the lab and disconnect the board.

1. Manually calculate the results and write down the ‘C’ and ‘Z’ of the condition code register after performing the following additions and subtractions. Write programs to add or subtract the numbers that are stored in the memory and store the sum in memory (i.e. your program should read numbers from the memory, apply mathematical operation to the numbers accordingly and save the result in the memory). Use the Memory Window to initialize the memory locations with values below. Execute the programs and record the sum or subtraction and the ‘C’ and ‘Z’ bits of the condition code register. Verify the results with your manual calculations and the value of the ‘C’ and ‘Z’ bits. Use the following data:

   a. Addition of two single-byte numbers: \( 0xC3 + 0x8D \)

   Helpful steps:

   1) Use LDRB to read a byte of data from memory into a register. How does an instruction refer to memory location? Hint: Look at program from previous lab experiment.

   2) Use ADD to add numbers in registers.

   3) Use STRH or STR in order to save the result from a register to memory—why shouldn’t you use STRB in this case?

   b. Subtraction of two single-byte numbers: \( 0xF5 - 0x34 \)

   c. Three-byte subtraction: \( 0xFE6B34 - 0x58CF21 \)

   d. Multi-byte addition: \( 0x2E68B3F4 + 0x5C2A \)

2. Consider the Program Directives.s in Figure 2.1. Modify that program so that all the following changes are made in your new program:

   a. The table to be copied is of length 0x20.

   b. The original table is created starting at 0x2000.0480

   c. The table starts with 0x00 and successive even numbers are stored in the table.

   d. The duplicate table is created starting at 0x2000.0C00

   e. The duplicate table contains the copy of the original table in the reverse order.

   Show the results to the TA after running this modified program.
2.9 Questions:

1. Describe the function of the assembler directives: *EQU*, *DCB*, *DCW*, and *END*.

2. If the following data is in these registers or memory,
   
   - Register R0 = 0x27
   - Register R1 = 0x2000.0040
   - Memory address 0x2000.0040 = 0x56
   - Memory address 0x2000.0042 = 0x78

   What numbers are in registers R0 and R1 after each of the following instructions is executed?
   
   a. LDRB R0, [R1, #2]
   b. LDRB R0, [R1], #2
   c. LDRB R0, [R1, #2]!

3. Extra Credit: There are 60 students in the class and you want to input their one digit grades after displaying *Grades for the next student =* on the screen. Write a program that uses InChar and OutStr to perform this.

2.10 Lab report:

For the lab write up, include

1. Flowcharts and programs that you wrote before the lab.
2. Manual results that you calculated before the lab.
3. A copy of your working *s files (DO NOT submit *.lst files).
4. A brief discussion of the objectives of the lab and the procedures performed in the lab.
5. Answers to any questions in the discussion, procedure, or question sections of the lab.