Carleton University

Department of Systems and Computer Engineering

SYSC 3006   Fall 2016

Computer Organization

Lab #8

Prelab: The **Pre-Lab** involves the completion of Fragment 1 and a prep review for Fragment 2. It is best to come to the lab with (at least) most of the details needed to make Fragment1 work.

In this lab you will:

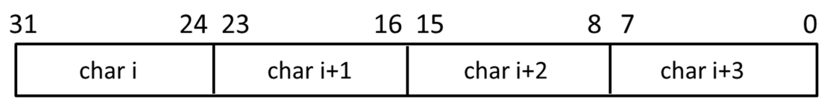
- Program a given computer system.
- Implement and test using the Assembler and Debugger (Logisim) circuit.

Read this document carefully before deciding what to do.

This lab explores the design and use of assembly subroutines. As a reminder, the Software Tools (Assembler and Debugger) needed for this lab are posted in the Resources section of the course website. Review the Fragment Source > Assembly > Debugging workflow video , if necessary.

Packed Strings

Recall that ASCII characters are 8-bit values. Using 32-bit values to store characters is not efficient, since ¾ of the bits in each memory word are not used. For example, storing a 10 character string in memory using this approach would require 11 (32-bit) words (don't forget that strings have the null-terminator 0x0 as their final character!). To help save space, strings are often "packed" such that each 32-bit memory word is used to hold 4 characters of the string. When packed into a 32-bit memory word, bits 24 to 31 of the word hold character i of the string, bits 16 to 23 of the word hold character i+1 of the string, bits 8 to 15 of the word hold character i+2 of the string, and bits 0 to 7 of the word hold character i+3 of the string. A 32-bit word in a packed string is shown below:



Using this packing approach, a 10 character string would only require 3 (32-bit) words of memory. For example, the 10 character packed string “SYSC 3006!” would occupy 3 words of memory as follows:

31 24	23 16	15 8	7 0
‘S’	‘Y’	‘S’	‘C’
‘ ‘	‘3’	‘0’	‘0’
‘6’	‘!’	0x00	0-padding

A string can be thought of as an array of characters, while a packed string can be thought of as an array of 32-bit packed words. Recall how we initialized an array of integers, called Arr, in Lab 7 (by simply labeling the head or the first element of the array with the label Arr). We will use the same concept to declare and initialize packed strings. For instance, the string "SYSC 3006!" might be declared as a variable in a program using:

```
ExampleString
        DCD   #0x53595343  ; ‘S’=0x53   ‘Y’=0x59   ‘S’=0x53   ‘C’=0x43. These are ASCII hex codes.
        DCD   #0x20333030
        DCD   #0x36210000  ; the 1st 00 is the null-terminator, the 2nd 00 is just padding to fill the word
```

The characters in a packed string can be indexed using a C-like approach. In the example above, the first ‘S’ is the 0th character (i.e. ExampleString[0]) and the ‘3’ is the 5th character (i.e. ExampleString[5]).

Fragment1

The Fragment1SRC.txt template contains code that loads an indexed character from a packed string into the least significant nybble of R0 and makes sure that the rest of R0 is clear (i.e. 0).

The Fragment1SRC template does not include pseudocode … you have to figure out what the code needs to accomplish and then fill in the necessary details to accomplish that objective. You will need to think about manipulating the binary encoding of the packed string to accomplish the objective.

The Fragment1SRC template uses the ExampleString (above) and looks for the 5th character, so when it terminates, R0 should contain 0x00000033 (i.e. ExampleString[5], which is the character ‘3’). After getting it to work for the 5th character, change the index to be sure it will work for other characters in the packed string.

Pre-lab: see the notes about LSR and LSL instructions at the end of this document, then complete the Fragment1SRC code by replacing all occurrences of “***” with the necessary details. Do not add any additional instructions.

Fragment 2

Fragment1 has been carefully designed to allow it to be easily converted into a subroutine. The Fragment2SRC.txt template is designed to incorporate your Fragment1 solution into a subroutine called **CharAt**, and then use that subroutine to count the number of characters in a packed string.

The CharAt subroutine header is:

```
char  CharAt ( &( PackedString[] ), uint charIndex )
```

In the pseudocode: &(…) is used to indicate a parameter passed by reference, and uint is an unsigned integer type.

The Fragment2SRC.txt template uses the CharAt subroutine to count the number of characters in a packed string. The pseudocode is:

```
R2 = 0    ; initial count = 0
for ( R0 = CharAt( &(ExampleString), R2 ); R0 != null; R0 = CharAt( &(ExampleString), ++R2 ) ) {
        ; empty loop body
        ; after loop terminates, R2 = number of characters in ExampleString
}
```

Recall that ++R2 increments the value of R2 before using the resulting value.

Pre-lab: *before* attempting this fragment, review the following concepts from TowardsSoftware (or the equivalent slides in Lecture 17):

- Understand the difference between the Bcc and BLcc instructions (slides 222-230).
- Review the Calling Convention (slides 232-234). Particularly how parameters are passed to subroutines.
- Understand how the LEA instruction is used to load the address of a label (slides 246-248).
- See the notes at the end of this document on the Register Preservation Convention. (equivalent to slides 249-252).

In the lab: complete the Fragment2SRC code by replacing all occurrences of "***" with the necessary details. Do not add any additional instructions.

Fragment 3 [OPTIONAL]

Convert your Fragment2SRC solution into Fragment3SRC.txt. The code in Fragment3SRC must use nested subroutine calls where the LengthOf subroutine calls the CharAt subroutine. LengthOf returns the length of a packed string.

The Fragment3SRC pseudocode is:

```
char  CharAt( &( PackedString[] ), uint charIndex )      ; as in Fragment2SRC, details omitted here

uint   LengthOf( &( PackedString[] ) )    ; very similar to code in Fragment2SRC ☺ (see note below)
{       R2 = 0     ; initial count = 0
        for ( R0 = CharAt( &(PackedString[]), R2 ); R0 != null; R0 = CharAt( &( PackedString[]), ++R2 ) ) {
             }
        return  R2
}

void  Main ();
{       R0 = LengthOf( &(ExampleString ) )
}
```

In the pseudocode: &(…) is used to indicate a parameter passed by reference, and uint is an unsigned integer type.

Note: LengthOf is intended to be a general subroutine that would work for any packed string, not just ExampleString. Therefore, code for LengthOf must not make explicit references to ExampleString. (Warning: the part of the Fragment2SRC code that you will convert into LengthOf makes explicit references to ExampleString … you must change this!)

Show your completed Fragment3SRC code to the TA after testing (i.e. Assembly > Debugging) to confirm its correctness.

Good Luck! ☺

Notes - Part I: Logical Shift Instructions

The Debugger and Assembler support the following instructions that have not been discussed in class:

LSR, **LSL**: Logical Shift Right (LSR) and Logical Shift Left (LSL)

The LSR and LSL instructions are used to shift the value in a register in the specified direction (Left or Right) by a specified number of bits. The result is always a 32 bit value, and any bits shifted out of the 32-bit value range are ignored (with the exception of the Carry Flag). The number of bits a value is shifted is called the shift *distance*. The "Logical" part of the names means that 0s are shifted into the register as the data moves. The shift distance is limited to 32 bits maximum (since a register only has 32 bits to hold the result). Note that shifting left one bit is the same as multiplying by 2, while shifting right is the same as (unsigned) dividing by 2.

FLAGS: LSR and LSL are data manipulation instructions. They set the flags in the following ways:

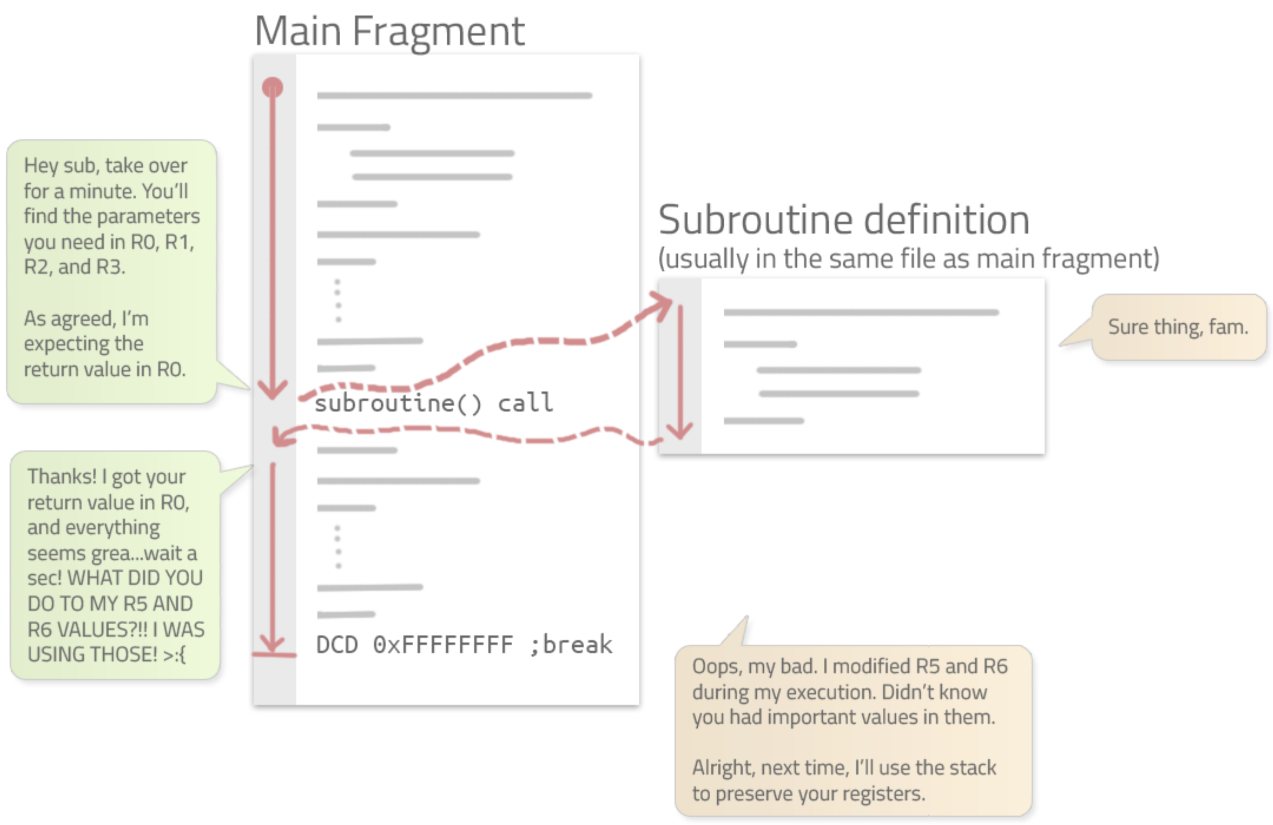
Z == 1 iff the result is 0
C == the value of the last bit shifted out of the 32-bit result
N == 1 iff the msb of the result = 1
V == 0 never gives signed overflow

Syntax and RTL:	LSL Rd, Rx, Ry	Rd ← Rx * 2^{Ry}
	LSL Rd, Rx, #imm16	Rd ← Rx * 2^{imm16}
	LSR Rd, Rx, Ry	Rd ← Rx ÷ 2^{Ry}
	LSR Rd, Rx, #imm16	Rd ← Rx ÷ 2^{imm16}

```
Examples:    LSL   R4, R4, R3     ; R4 is shifted left by the number of bits specified in R3
             LSR   R5, R2, #4     ; The value from R2 is shifted right 4 bits and then
                                  ; stored in R5
```

Notes - Part II: Register Presevation on the Stack

The following will illustrate how and why PUSH/POP instructions are used when calling a subroutine. Consider this scenario:


Main Fragment
Hey sub, take over for a minute. You'll find the parameters you need in R0, R1, R2, and R3. As agreed, I'm expecting the return value in R0.
subroutine() call
Thanks! I got your return value in R0, and everything seems grea...wait a sec! WHAT DID YOU DO TO MY R5 AND R6 VALUES?!! I WAS USING THOSE! >:{
DCD 0xFFFFFFFF ;break
Subroutine definition
(usually in the same file as main fragment)
Sure thing, fam.
Oops, my bad. I modified R5 and R6 during my execution. Didn't know you had important values in them. Alright, next time, I'll use the stack to preserve your registers.

Preserving Registers on the Stack: to resolve the problem illustrated above, we shall agree (i.e. make it a convention) that a subroutine is responsible to preserve any registers it knows it will modify. The Stack is a space available near the end of Main Memory (address 0x800), which can be used to temporarily store (PUSH) and retrieve (POP) values to/from. Hence, subroutines can use the stack to preserve the caller's registers (the caller in the example above is the Main Fragment).

Here's how to ensure a given subroutine properly preserves the caller's registers onto the stack:

1. At the beginning of the subroutine, we call PUSH { /* Comma-separated list of Registers to preserve */}. This list contains:
   a. All the registers *modified* within the body of the subroutine.
   b. Additionally we push register R14 (Link Register) because it contains our return address.
   c. If the subroutine is returning a value (into R0), then we don't need to push and preserve the caller's R0 value (and the caller already knows this, thanks to Parameter Passing Convention).
2. Subroutine execution continues..
3. At the end of the subroutine, we call POP { /* Comma-separated list of Registers to be retrieved*/ }. This list contains:
   a. All the registered we PUSHed earlier.
   b. Replace R14 with R15. This means the PC register (R15) will now point to the return address (i.e. we're done executing the subroutine, let's return and continue the main fragment).