

Carleton University
Department of Systems and Computer Engineering
SYSC 3006 (Computer Organization) summer 2020
Lab / Assignment 6

Prerequisites

Review the following concepts from Towards Software Notes or the equivalent lecture slides:

1. Understand the difference between the Bcc and BLcc instructions (slides 227-235).
2. Review the Calling Convention (slides 237-249). Particularly how parameters are passed to subroutines.
3. Understand how the LEA instruction is used to load the address of a label (slides 246-248).

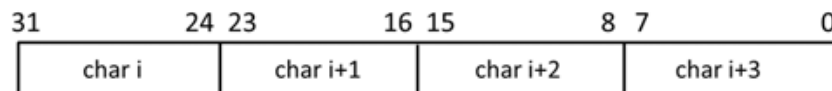
Goal

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

Introduction

Packed Strings

Recall that ASCII characters are 8-bit values. Using 32-bit values to store strings is not efficient, since $\frac{3}{4}$ of the bits in each word are not used. For example, storing a 10-character string in memory using this approach would require 11 (32-bit) words (do not forget that strings have the null-terminator as their final character!). To help save space, strings are often “packed” such that each 32-bit word is used to hold 4 characters of the string. When packed into a 32-bit 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 (including the terminator would only require 3 (32-bit) words of memory. For example, the 10-character packed string “SYSC 3006!” would occupy 3 words of memory:

'S'	'Y'	'S'	'C'
' '	'3'	'0'	'0'
'6'	'!'	0x00	

And the string might be declared as a variable in a program using:

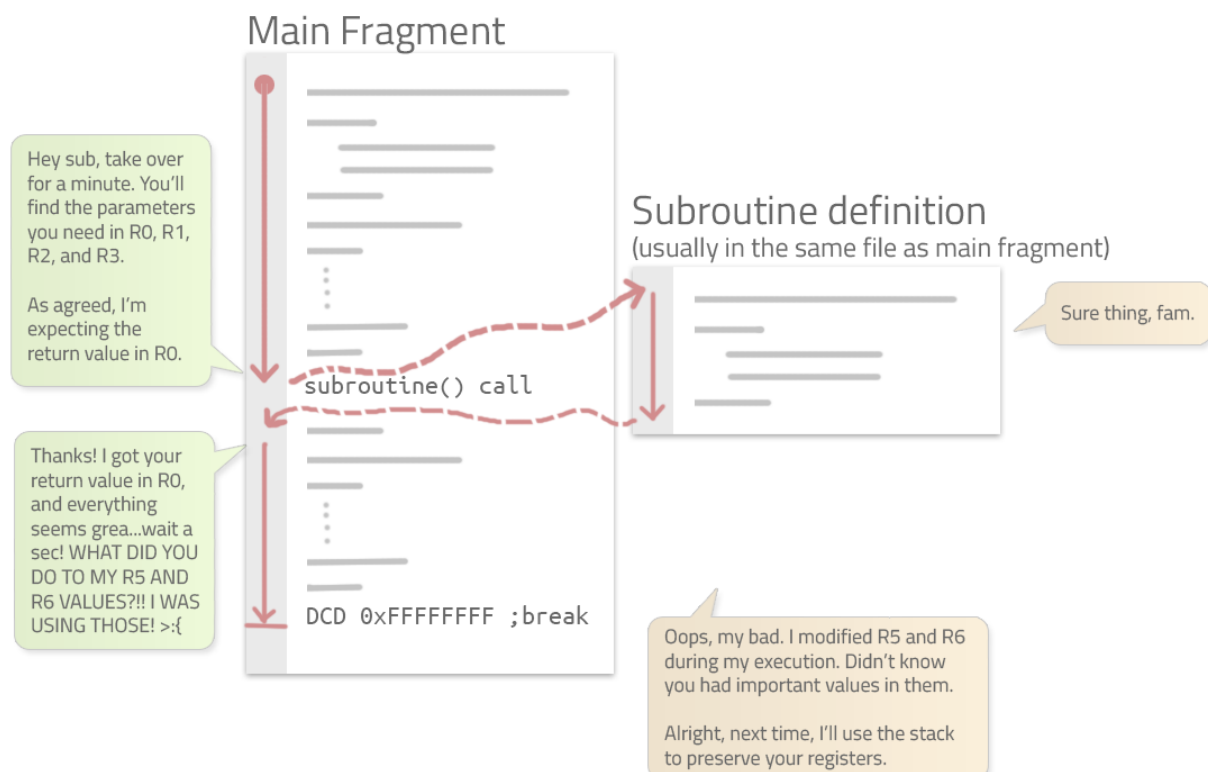
ExampleString:

```
DCD #0x53595343      ; 'S'=0x53 'Y'=0x59 'S'=0x53 'C'=0x43
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]).

Register Preservation on the Stack

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



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 is 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 do not 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 registers we PUSHed earlier.
- b. Replace R14 with R15. This means the PC register (R15) will now point to the return address (i.e. we are done executing the subroutine, let's return and continue the main fragment).

Logisim files setup and components information

The Software Tools needed for this lab are posted on the course website under the tab "Resources". The tools include the Assembler and the Debugger circuit. Do not change any wiring inside the Debugger circuit.

Tutorial Assembling and debugging

To understand how to assemble and debug a given fragment, follow this [video tutorial](#) (5 minutes; no audio), also watch the video posted on cuLearn (just above lab 6) . For more details, you can always refer to the pdf guides accompanying each tool on cuLearn: Debugger User Guide and Assembler User Guide.

As discussed in the Debugger User Manual: Be sure that the clock has been enabled to run freely in the simulation: Simulate → Ticks Enabled. **Note: Ticks are disabled by default when Logisim is loaded, so you must enable ticks before anything will happen.**

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	$Rd \leftarrow Rx * 2^{Ry}$
LSL Rd, Rx, #imm16	$Rd \leftarrow Rx * 2^{imm16}$
LSR Rd, Rx, Ry	$Rd \leftarrow Rx * 2^{-Ry}$
LSR Rd, Rx, #imm16	$Rd \leftarrow Rx * 2^{-imm16}$

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
```

Register Preservation on the Stack

The Debugger and Assembler support the PUSH and POP instructions “almost” as discussed in class.

Exception: the register lists must be written out as a comma separated list.

For example:

```
PUSH { R1, R2, R3 } ; is supported
PUSH { R1 - R3 }   ; is not supported
```

Your Assignment

Use the included editable .docx file (MS-Word) for your answers, then save it as PDF file before submitting it. Here we refer to tables to be filled in, they are not shown here but they do in the assignment .docx file.

Part 1 – Fragment 1 [2-mark/5]

Fragment1 SRC.txt (included in lab 5.zip).

The Fragment1SRC.txt template contains code that loads an indexed character from a packed string into the least significant byte of R0 and makes sure that the rest of R0 is clear (i.e. 0). The Fragment1SRC template does not include pseudocode ... you must 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.

1. [2-mark] Complete the Fragment1SRC code by replacing all occurrences of “***” with the necessary details. Do not add any additional instructions. Then Submit your completed (working) Fragment1SRC code.

Part 2 – Fragment 2 [3-mark/5]

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 ( &(amp; 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.

1. [2.5-mark] Complete the Fragment2SRC code by replacing all occurrences of “***” with the necessary details. Do not add any additional instructions.
2. [0.25-mark] Describe the difference between the Bcc and BLcc instructions, and why the difference is important.
3. [0.25-mark] Briefly describe the programming conventions associated with subroutines that are used in this course.

Must be submitted on cuLearn, locate (Assignment 6 submission) and follow instructions. Submission exact deadline (date and time) is displayed clearly within the Assignment 4 submission on cuLearn.

Note: If you have any question please contact your respective group TA (see TA / group information posted on cuLearn) or use Discord class server.

Good Luck