ARM Assembly
Code Style Guide

Aditya Bajaj
Lina Johnson

Table of Contents

Introduction

Summary

This documentation serves the purpose guiding those who are developing in ARM assembly to create clear and consistent code. The reader will be able to reference this documentation to ensure that their code follows the expected guidelines for the book. Although there may be variations on how an individual may approach code style, keep in mind that the suggestions in this guide are composed of best practices found across various organizations, companies, and experts in this field.

Pre-requisites

In order to properly utilize the information being taught the following dependencies are required. For detailed information on where to buy bundles, suggested set up, and more; reference "*BuyingPi.pdf*" in Module 1.

Hardware

- Raspberry Pi
- 16 GB Micro SD card
- Monitor
- USB keyboard
- USB mouse

Software

- PuTTY and a terminal
- X-Windows (if not using terminal)
- Text Editor

Additional Resources

Helpful resources will be referenced throughout each module. General helpful resources are included below.

- ARM Assembly Reference Card
  http://www.cburch.com/cs/230/arm-ref.pdf
- Download page for *Digital Circuit Projects - Second Edition* eBook
  https://cupola.gettysburg.edu/oer/1/
- Download page for *Introduction To MIPS Assembly Language Programming* eBook
  https://cupola.gettysburg.edu/oer/2/
- Download page for *Implementing a One Address CPU in Logisim* eBook
  https://cupola.gettysburg.edu/oer/3/
- Unix Cheat Sheet
  http://www.mathcs.emory.edu/~valerie/courses/fall10/155/resources/unix_cheatsheet.html

File Structure

General Structure

The following defined style must be followed across all ARM programs to ensure readability and proper structure.

File Naming Conventions

The file names should be in camelcase starting with a lowercase letter.

```
# Good Examples
helloWorld.s
toBeOrNotToBe.s
multiplyNumbers.s
addNumbers.s

# Bad Example 1 - Starts with uppercase
HelloWorld.s

# Bad Example 2 - Starts with numerical value
2beOrNot2Be.s

# Bad Example 3 - Snake case and all caps
MULTIPLY_NUMBERS.s

# Bad Example 4 - hyphenated and all lowercase
add-numbers.s
```

Files containing libraries must also include the prefix, “lib”. Doing so makes it clear that the file uses a library without having to open the file up. A library file is any file that does not contain a main function and contains two or more global symbols for functions.

```
# Good Example
libConversion.s

# Bad Example - not clear that it contains libraries
conversion.s
```

Files that contain a main should be suffixed with, “Main” and main should be removed when the program file is generated.

```
# Good Example
calculateHeightMain.s

# Bad Example 3 - not clear where main lives
calculateHeight.s
```

Preamble Section

Every program written must begin with a preamble at the top of the program that states the name of the program, the purpose of the program, the author of the program, functions, and the date when it was written. All functions must be ordered in alphabetical order and include a short description of its purpose. Any inputs and outputs should also be defined in the preamble. It should adhere to the following format:

```
#
# Program Name: helloWorld.s
# Author: John Doe
# Date: 11/11/2020
# Purpose: To print out a hello world message using a
#          system call (svc) from ARM assembly
# Functions: (when applicable)
# Inputs: (when applicable)
# Outputs: (when applicable)
#
```

Comments

All comments in the code should follow the same indentation guideline that the code follows. Comments that pertain to a nested component should follow the same white space as the component it references.

Single lines can be commented after the code using the "//" characters. However only lines that are unclear should be commented. Line comments should be used very sparingly and are not the preferred option.

Data Section

Following the end of the .text section, the .data section must begin. All the variables that were used in the .text section should be declared in the .data section. If there are multiple .text sections, then each section must be followed by a .data section where the respective variables are declared. Proper indentation & comments should be followed for the .data section.

```
.data
    # Prompts the user to enter his/her name
    .namePrompt: .asciz "Enter your name: "

    # Prompts the user to enter his/her age
    .agePrompt: .asciz "Enter your age: "
```

Text Section

All executable assembly language instructions must be defined in the .text section. Proper indentation & comments should be followed for the .text section.

```
.text
.global main

    main:
        # Prompts user for an input
        LDR r0, =userPrompt
        BL printf
```

Case Sensitive Rules

Instructions should be all uppercase. Registers should be all lowercase. Functions should follow the same naming convention rules as variables. All naming conventions should be clear and indicative of the information that is associated with it. Functions should be named in a clear human readable way that is short yet descriptive.

Variables

Global Variables

All global variables need to be declared at the beginning of the file before. Global variables should be in camelcase and ordered in alphabetical and numerical order. The camelcase rule applies for static and static external variables. For constant variables, use all uppercase characters and snakecase (e.g. underscores between words).

```
# Good Constant Variable Example
INCHES_IN_A_FOOT
```

Local Variables

All local variables should be defined in the .data section. The variables should have meaningful names and should be in camelcase. Make sure to order variables in alphabetical and numerical order. Any variables with an ephemeral purpose should use non-preserved registers. If any register other than the non preserved registers are used (e.g. r4-r12), the name of a variable should be associated with them.

```
# Good Examples
tempInFahrenheit
measurementInFeet
input

# Bad Examples - trivial variable names
.global i
LDR r0 =w

# Bad Examples - non-descriptive
.global doAction
LDR r0 =measurement
```

Formatting

Instructions should be all uppercase. Registers should be all lowercase. Functions should also be all lowercase. Single characters should be enclosed with single quotes. Strings should be enclosed with double quotes. The following code example demonstrates proper formatting.

Filename: **`fahrenheitToCelsius.s`**

```
#
# Program Name: fahrenheitToCelcius.s
# Author: Charles Kann
# Date: 11/01/2020
# Purpose: Calculates a temperature from fahrenheit to celsius.
# Inputs:
#   - userInput: Temperature in celsius user wants converted
# Outputs:
#   - convertedMessage: Converted temperature from C to F
#

.text
.global main

main:
    # Save return to OS on stack
    SUB sp, sp, #4
    STR lr, [sp, #0]

    # Prompt user for input with a message
    LDR r0, =temperaturePrompt
    BL  PRINTF

    # Takes user input (SCANF)
    LDR r0, =userInput
    SUB sp, sp, #4
    MOV r1, sp
    BL  SCANF
    LDR r0, [sp, #0]
    ADD sp, sp, #4

    # Convert
    BL fahrenheitToCelcius
    MOV r1, r0

    # Print the converted value as a message to the user
    LDR r0, =convertedMessage
    BL PRINTF

    # Return to the OS
    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOD pc, lr

.data
    # Tells the user what the temperature conversion is
    convertedMessage:  .asciz "\nThe temperature in C is %d\n"
    # Takes the user input for conversion calculation
    userInput: .asciz "%d"
    # Prompts the user to enter in the temperature in fahrenheit
    temperaturePrompt: .asciz "Enter the temperature in F you want in C: \n"
```

Whitespace

Tab characters should not be used at any time. The indentation style that should be used is 4 spaces. Make sure to update your text editors to replace any tab characters with 4 spaces instead.

Newlines

There should be a new line after every instruction. There should also be an additional new line after each segment of the assembly program to delineate the various sections clearly and allow for more easily readable code. There should never be more than a single blank line between logical groupings at any point in the program with the exception of a new page character.

Functions

All functions require clear documentation that specifies the functionality of the function as well as detailing the data manipulation (i.e. register usage and expectations). It should also document any potential errors and dependencies. Alterations such as non-local variable change or print statements must be called out. For functions that are complex, include pseudo code to explain how they work. Functions should include a clear “END OF” comment to clearly indicate the end of the function. Functions should adhere to the following format:

```
# Purpose: To print out a hello world message using a
#          system call (svc) from ARM assembly
# Inputs: (r0, r1, r2, or r3 are allowed)
# Outputs: (only r0 and r1 are allowed)
# Errors: (when applicable, be descriptive)
# Alterations: (when applicable)
# Pseudo Code: (when applicable)

... <function code here> ...

# END OF <function_Name>
```

Loops

Loops should be implemented by specifying a start of loop label and an end of loop label that encapsulates the logic within. A comment is required for the initialization, iteration check, next item, and the end.

Logic Groupings

Logical checks should be grouped and have an empty new line after the group to allow for more readable code. Each logical grouping (e.g. if-elseif-else, etc.) should also include a brief comment above the first executable line to indicate the group. For non-trivial groupings the comments should be more detailed in explaining what is being done in the group.

Memory Allocation

All parameters for a function (r0, r1, r2, and/or r3) that contain saved values that are intended to be used later in the function should be saved to the stack or to a preserved register.

If stack space is allocated and not used to save a register in the push section of the function the purpose of this stack variable should be well documented.

Structured Programming

It is important that no code should ever branch out of a program block. Any branching done must be to a place in the current program block. There are sometimes exceptions (e.g. the branch is at the end of a loop structure), however branches should always be forward in the code.