

The ARM Instruction Set and Cross-Compilation

Computer Architecture Exploitation and Security

Table of Contents

[Objectives 2](#_Toc37293837)

[Background Reading 2](#_Toc37293838)

[Important Information 2](#_Toc37293839)

[Introduction 3](#_Toc37293840)

[Problem 1 3](#_Toc37293841)

[Problem 2 5](#_Toc37293842)

L*abs must be submitted by the due date for full credit. After due date late submissions will be accepted for a period of one week (seven days) and the grade will be reduced by ten percent (10%) per day after due day.* ***Assignments that are submitted more than seven days late will receive a grade of zero (0).***

I certify that the work submitted in this assignment is my own and that it has not been taken in whole or in part from any other source. I understand that the penalty for plagiarism will include a grade of zero (0) for this assignment plus disciplinary action in accordance with SAIT policies.

**EVALUATION**:

|  |  |  |
| --- | --- | --- |
| ARM Instruction Set | 23 |  |
| Cross-Compilation | 10 |  |
| TOTAL MARK | 33 |  |

Computer Architecture Exploitation and Security

The ARM Instruction Set and Cross-Compilation

Objectives

This lab focuses on the following objectives:

* Describe registers.
* Explain ALU functionality.
* Describe processor control registers.
* Cross-compilation

Background Reading

* ARM Instruction Reference
* <https://developer.arm.com/documentation/dui0068/b/arm-instruction-reference>

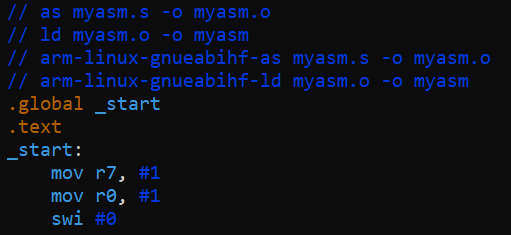
.

# Problem 1 \_\_23

Use the following arm instructions to perform the respective calculations and based on the results complete the tables. Reference for Conditional Execution- CPSR <https://developer.arm.com/documentation/den0024/a/The-A64-instruction-set/Data-processing-instructions/Conditional-instructions>

**Take a moment to consider what would be the easiest approach to solve the questions below:**

**Hint:** Assembly code template:



**adds r0, r0, r1**

|  |  |  |
| --- | --- | --- |
| **Register or Memory Address** | **Before** | **After** |
| r0 | 0x22145678 |  |
| r1 | 0x02545678 |  |
| cpsr bits N,Z,C,V | 0,0,0,0 |  |

**subs r0, r0, r1**

|  |  |  |
| --- | --- | --- |
| **Register or Memory Address** | **Before** | **After** |
| r0 | 0x23456788 |  |
| r1 | 0x23456789 |  |
| cpsr bits N,Z,C,V | 0,0,0,0 |  |

**ands r0, r0, r1**

|  |  |  |
| --- | --- | --- |
| **Register or Memory Address** | **Before** | **After** |
| r0 | 0xa1b2c3d4 |  |
| r1 | 0x21436587 |  |
| cpsr bits N,Z,C,V | 0,0,0,0 |  |

**orrs r0, r0, r1**

|  |  |  |
| --- | --- | --- |
| **Register or Memory Address** | **Before** | **After** |
| r0 | 0xa1b2c3d4 |  |
| r1 | 0x21436587 |  |
| cpsr bits N,Z,C,V | 0,0,0,0 |  |

**eors r0, r0, r1**

|  |  |  |
| --- | --- | --- |
| **Register or Memory Address** | **Before** | **After** |
| r0 | 0xa1b2c3d4 |  |
| r1 | 0x21436587 |  |
| cpsr bits N,Z,C,V | 0,0,0,0 |  |

**bic r0,r1,r2**

|  |  |  |
| --- | --- | --- |
| **Register or Memory Address** | **Before** | **After** |
| r0 | 0xa1b2c3d4 |  |
| r1 | 0x21436587 |  |
| r2 | 0x20000420 |  |
| cpsr bits N,Z,C,V | 0,0,1,0 |  |

**adc r0,r1,r2,lsl #3**

|  |  |  |
| --- | --- | --- |
| **Register or Memory Address** | **Before** | **After** |
| r0 | 0xa1b2c3d4 |  |
| r1 | 0x21436587 |  |
| r2 | 0x02000240 |  |
| cpsr bits N,Z,C,V | 0,0,1,0 |  |

# Problem 2 Cross-Compilation \_\_10

Because of the limited size of memory provided on embedded systems, developers will normally create a cross-compilation environment. The term cross-compilation simply means that the environment where the tools (gcc, objdump), are being used will not be the environment to allow that executable to run natively.

**The idea behind cross-compilation is that you can:**

1. Have the toolchain (development tools to create analyze and modify binary files) on a system with sufficient resources (CPU, RAM, HD) to allow for rapid prototyping.
2. Then you can transfer only the binary files to the embedded system to test.

For this problem you will follow a simple workflow to create a binary on your local **Ubuntu** computer and then transfer that file to the **Raspberry pi** after compiling your executable.

If your VM is **x86\_64** and the OS on the Raspberry Pi is the (aarch64), **confirm** **with command lscpu,** perform the following steps:

* 1. Install the cross-compilation toolchain for ARM-64 on your Linux x86\_64 VM.

NOTE: Some of the tools may already be installed

**sudo apt install gcc make** **gcc-aarch64-linux-gnu binutils-aarch64-linux-gnu**

**Raspberry Pi (with 32bit - Ubuntu, Raspberry PiOS or Kali installed)**

If you installed Raspberry Pi OS, Ubuntu or Kali 32-bit perform the following steps:

1. Install the cross-compilation toolchain for ARM-32 on your **Linux VM**

**sudo apt install gcc make gcc-arm-linux-gnueabi binutils-arm-linux-gnueabi**

1. Use gcc to compile one of the C code you created before
2. Use executable file created by gcc and answer the following questions
   * 1. What is the file type?
     2. What the architecture?
     3. What is LSB?
3. Now build the the same program with the arm cross-toolchain you compiled before as follows: **aarch64-linux-gnu-gcc program-name.c -o programarrch64 -static**
4. Use the newly created file **programaarch64** to verify the file type and the architecture.
5. Can you execute **programaarch64** on your Linux x86\_64 VM? Why?
6. Attach the screen that demo results after using file command with programaarch64 and after execution
7. Use the templates below to generate a program that:
   * 1. Asks the user to enter a string no longer than 30 characters.
     2. Take the string and print it back to the screen using assembly.
     3. Use **aarch64-linux-gnu-as** and **aarch64-linux-gnu-ld commands** to create the executable
     4. ARM uses X0, X1, Xn for there registers which is different from RDI etc that Intel uses.
8. Attach a screen captures the demo of the results. Use file command to demo the file type and architecture of the executable.
9. Your executables should run on ARM architecture (raspberry pi)
10. Transfer the executable file to your Raspberry Pi and show a screen capture of it running on the Raspberry Pi.

**; Intel Code written for NASM Assembler**

extern printf

extern exit

global main

hello: db "Hello World",0xa,0x0

section .text

main:

lea rdi, [hello]

xor rax, rax

call printf

mov rdi, 0

xor rax, rax

call exit

**# Intel Code written for GAS Assembler**

.intel\_syntax noprefix

.extern printf

.extern exit

.globl main

hello: .ascii "Hello World\n\0"

.text

main:

lea rdi, [hello]

xor rax, rax

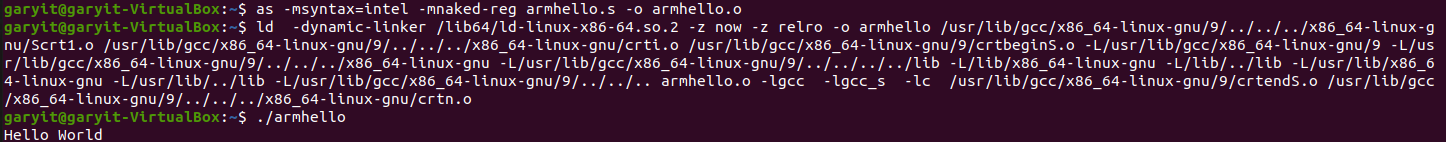
call printf

mov rdi, 0

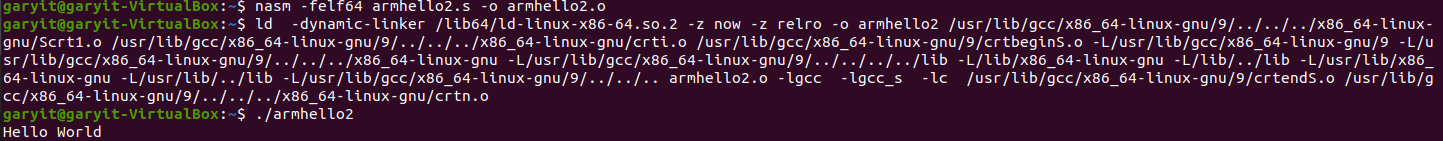
xor rax, rax

call exit

Commandline used to generate executable **armhello** starting with the GAS Assembler: **as**



Commandline used to generate executable **armhello2** starting with the GAS Assembler: n**asm**



Linker command line used for linking the object files.

ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z now -z relro -o armhello2 /usr/lib/gcc/x86\_64-linux-gnu/9/../../../x86\_64-linux-gnu/Scrt1.o /usr/lib/gcc/x86\_64-linux-gnu/9/../../../x86\_64-linux-gnu/crti.o /usr/lib/gcc/x86\_64-linux-gnu/9/crtbeginS.o -L/usr/lib/gcc/x86\_64-linux-gnu/9 -L/usr/lib/gcc/x86\_64-linux-gnu/9/../../../x86\_64-linux-gnu -L/usr/lib/gcc/x86\_64-linux-gnu/9/../../../../lib -L/lib/x86\_64-linux-gnu -L/lib/../lib -L/usr/lib/x86\_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86\_64-linux-gnu/9/../../.. armhello2.o -lgcc -lgcc\_s -lc /usr/lib/gcc/x86\_64-linux-gnu/9/crtendS.o /usr/lib/gcc/x86\_64-linux-gnu/9/../../../x86\_64-linux-gnu/crtn.o

**# ARM Code written for GAS Assembler**

.extern printf

.extern exit

.globl main

hello: .asciz "Hello World\n"

.align 4

.text

main:

ldr r0, =[hello]

bl printf

mov r0, 0

b exit

