ECE 2510

Lab 2

Objectives

- Understand the relationship between C programs and assembly language.
- Understand binary arithmetic, carry and overflow flags
- Using C programs to interface with peripherals
- Familiarize students with Keil uVision's debugging features.
- Learn how to start an assembly program
- Try out some assembly instructions and watch the registers changes

Introduction

In this lab, we will continue to use C programs to interact with the Nucleo evaluation board. We can write programs that use the different peripherals available on the board. Additionally, this lab will show you how we can use the debugger to step through the code and display the value of our variables. We will also learn how to create an ARM assembly project and write some assembly instructions.

Registers

Registers are the smallest unit of storage in a CPU that is used by assembly instructions. Figure 1 shows the different registers in the ARM architecture (including 12 general purpose registers, a stack pointer, link register, program counter, and special purpose registers).

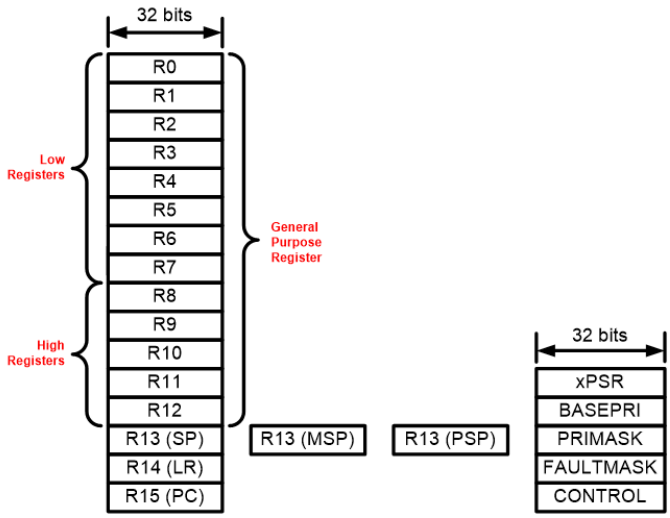


Figure 1- ARM's registers (courtesy of Dr. Zhu from UMaine)

Making a Keil uVision ASM project

When making an assembly file in Keil uVision, instead of selecting a .c file like we did in lab 1, add a .s file to the Source group. Once the file is made, take the assembly shell given in Elearning and copy it into the file. This will be the basic code outline used in most labs.

Task 1

In the C programming language, we can create variables or an array of variables that have the same type. Each variable we create is allocated at some location in memory. We can use the debugger to view the data in memory. In this task, you will learn about different datatypes in C and learn how to display variables in the Debug window.

Procedure:

1. Create a global char named var1.
2. Create a global int name var2.
3. Create a global char array with 5 elements named arrayA.
4. Create a global int array with 5 elements named arrayB.
5. Write a program that sets var1 = 5, var2 = 1000, arrayA equal to 10, 20, 30, 40, 50, and arrayB equal to 100, 200, 300, 400, 500
6. Simulate the program and step through the code line-by-line. Use the debugger to view the variables in memory. Follow the example below to see how to use the debugger.

After the program is completed, take a screenshot of the final values in var1, var2, arrayA and arrayB.

NOTES: do not download the program to the Nucleo board. Instead, run just the simulation.

To see the contents of the memory debugger, click on the “memory tab” in the bottom right corner of the uVision window. It will then prompt you for a memory address. If you are storing an array at memory location 0x1000 (as an example) you would type in 0x1000 and expect to see the array values populating those memory locations. Var1 should be stored at 0x2000007C, Var2 should be stored at 0x20000080, arrayA should be stored at 0x20000060, and arrayB should be stored at 0x20000068.

Task 2

Next, we will investigate arithmetic using 32-bit integers. For the following examples, you will perform hand calculations to find the result of each arithmetic operation and determine if there is an overflow and/or a carry out.

For the following binary numbers, perform the following tasks:

1. Add/subtract the two binary numbers together by hand; Indicate whether there is overflow or carry.
2. Find the decimal values of the binary numbers, perform the arithmetic by hand and find the values of the NZCV flags.
3. Convert the decimal results from (4) to binary and verify you your answers from (1)
4. Next, write a small C program that adds/subtracts the numbers together.

Simulate the program. Look at the assembly code of the C program you wrote.

After each arithmetic operation, take a screenshot of the debug window showing the result.
5. Do your simulation results match your hand calculations?

32 Bit arithmetic

The values are given in 8 bit format, but sign extend them to 32 bits when coding them

00001010 + 00011101 =

11111111 + 00000001 =

11101011 + 00011111 =

10010010 + 10100110 =

01100100 - 00010111 =

00011001 - 00011010 =

10001100 - 11000100 =

01101110 - 10100110 =

Write an ARM assembly code for adding/subtracting the below numbers. After each arithmetic operation, take a screenshot of the debug window showing the result, overflow flag, and carry flag. For this task, you need to create a new project the same way you learned before, but choose assembly instead of C. Then copy the shell file from elearning into main.s.

0x00000001 + 0x7FFFFFFF =

0xFFFFFFFF + 0x00000005 =

0xDEADBEEF- 0xDEADBEEF =

0xFFFFFFFF - 0x00000001 =

Simulate the results and screenshot the NZCV flags. Do they make sense? Why or why not?

Task 3

- Create a new ASM project by following the steps above.
- Write an assembly code the adds the binary numbers in Task 2 into the assembly shell after the “Main” label.

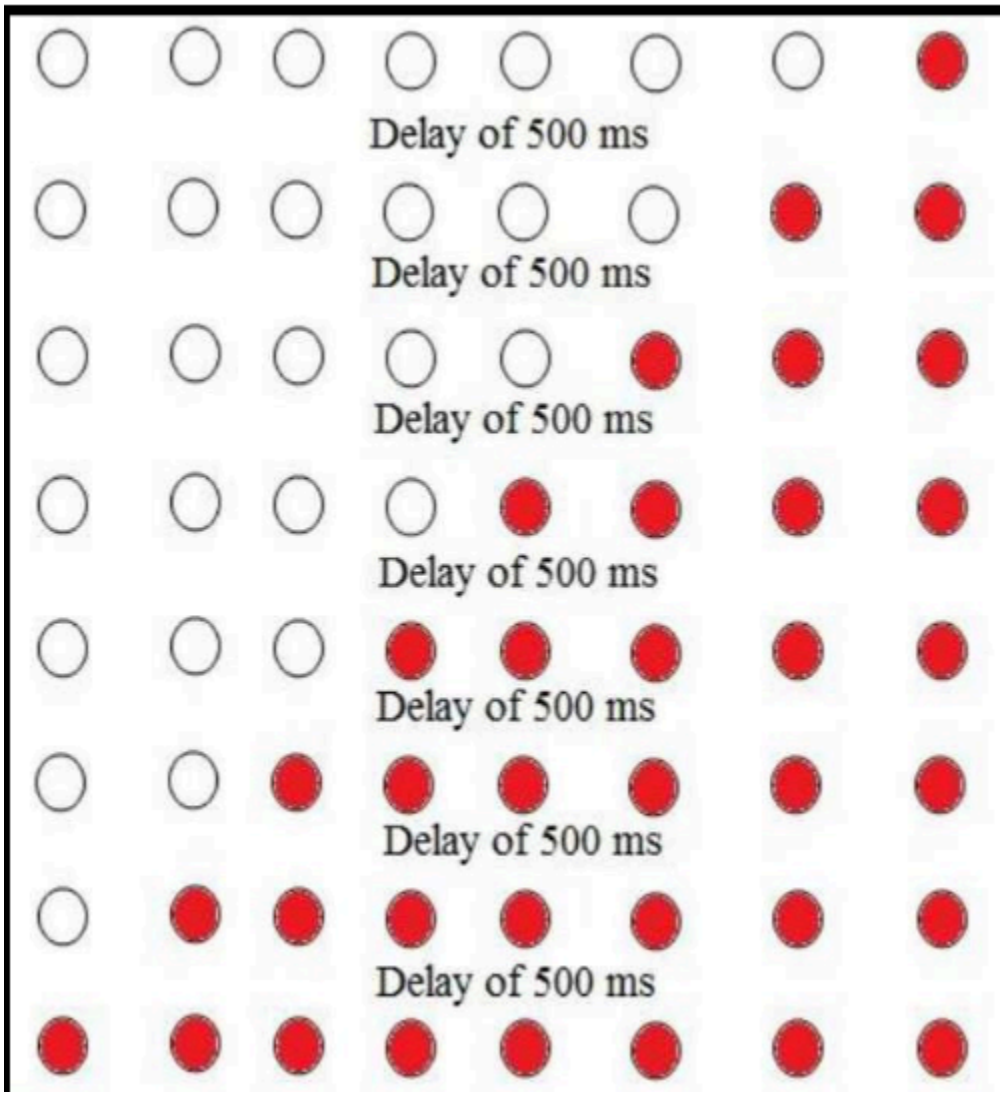
- Do the results match your hand calculations and the C program from Task 2?
- Watch and record the CCR flag after each operation. Do the CCR flags match your manual calculations from Task 2.

Task 4

As we learned in Lab 1, we can control each LED by writing 0's and 1's to PORT?. For this task, you are going to write a C program that will display the patterns in Figure 2 on the LEDs. Use the delay function from Lab1 so that the pattern changes every 500ms.

• Modify the code in Figure 3 to display the patterns below (with a 500ms delay between each pattern)

• Run the program on the Nucleo board and show your lab TA.



Task 5

- Create a new ASM project by following the steps above.
- Type the following code into the assembly shell after the “Main:” label.

```
MOV R0, #0x0F
ADDS R0, R0, #0x27

ADDS R0, R0, #0x45

ADDS R0, R0, #4

MOV R1, #2_01011111

MOV R1, #0

ADDS R1, R1, #0x6A
```

Next to each instruction, write the Register Transfer Language (RTL) as a comment
• Calculate the result of each instruction by hand
• Simulate the program step-by-step, take a screenshot of the destination register and verify the result matches hand calculations

Reminder: in ARM, #0x represents hexadecimal immediate and #2_ represents binary immediate

Lab Report:

Task 1:

• C code

• Screenshot of memory window showing final state of all variables

Task 2:

• Hand calculations: result of arithmetic operation, carry flag, and overflow flag (show your work!)

• C code

• screenshots of each arithmetic operation, showing the result

Task 3:

• Assembly code

• screenshots of each arithmetic operation, showing the result, carry flag, and overflow flag

Task 5:

Assembley code

• screenshots of each arithmetic operation, showing
the result, carry flag, and overflow flag

• RTL code for each instruction

Conclusion:

• Discuss any challenges you had with CodeWarrior.