Name: Minh Binh Nguyen

PantherID: 002-46-4288

**Task 4 Report**

**Part 1:**

I started with typing the code as instructed (Figure 1). I expected an error when running the program (Figure 2).

A screenshot of a cell phone

Description automatically generated

Figure 1

A screenshot of a cell phone

Description automatically generated

Figure 2

As the prompt shows, I figured it must be a typo of the keyword. Then I looked it up and found out that the correct keyword for a 16-bit integer is “.hword”, so I changed my code (Figure 3).

A screenshot of a cell phone

Description automatically generated

Figure 3

After assembled and linked it again, the program works as it should, then I can debug it using gdb (Figure 4).

A screenshot of a cell phone

Description automatically generated

Figure 4

By looking at the result of “info registers”, we can tell that the registers hold the values as they are expected to (Figure 5).

A screenshot of a cell phone

Description automatically generated

Figure 5

Notice that r1 holds a value of 0xFFFFFFFF, this is a negative number since the most significant bit is 1. If you two’s compliment this number, you’ll get the actual value of this negative number, which is -1.

**Part 2:**

I started with connect my Pi to my computer, created a program called “arithmetic3.s”, then typed in my code (Figure 6).

A screenshot of text

Description automatically generated

Figure 6

When declare the variables, I used “.byte” because these are 8-bit integers.

I used “ldrb” to load a byte value when loading val1 and val2, which are unsigned integers. With val3 (a signed integer), I used “ldrsb” (Figure 7).

A close up of text on a black background

Description automatically generated

Figure 7

While debugging the program, I checked the content of r1, which is the address of val2, then I checked the memory at that address. The memory holds the value 0Bh, which is 11 as expected (Figure 8).

A screenshot of a cell phone

Description automatically generated

Figure 8

The ldrb works like the movzx in x86, r1 now holds Bh in hex (11 in decimal) (Figure 9).

A screenshot of a computer

Description automatically generated

Figure 9

Checking the memory at val3 gives the value of 10h in hex (which is 16 in decimal as expected) (Figure 10).

A screenshot of a cell phone

Description automatically generated

Figure 10

Checking the memory at val1 gives the value of C4h in hex (this is -60 in decimal after two’s compliment) (Figure 11).

A screenshot of a cell phone

Description automatically generated

Figure 11

This is the result. Register r1 now holds an unexpected value FFFFFF5Ah. We can see that 5Ah is 90 in decimal (which is expected), but the F’s that come before it is not something that we are looking for. This is because when we use ldrb to load the value of -60 (which is required to be an unsigned integer), ldrb acts like movzx in x86. This makes the content of r2 hold C4h only (we expected it to be FFFFFFC4h to get the correct answer). When the sub is executed, it two’s-compliments this value and adds it to r1 (Figure 12).

A screenshot of a cell phone

Description automatically generated

Figure 12

To get the correct result, we need to load val1 as a signed integer using ldrsb instead. I made the changes to the code (Figure 13).

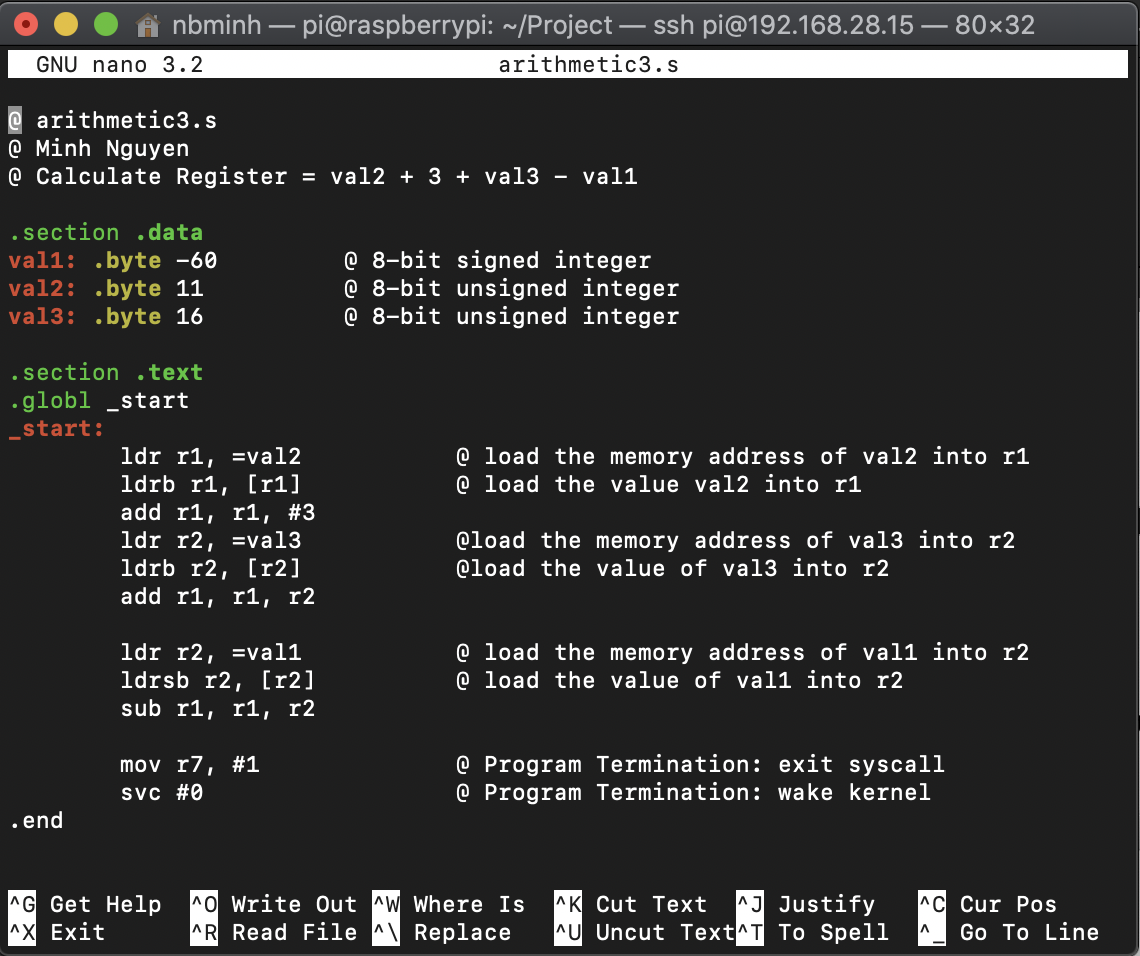


Figure 13

The final result of the expression is in r1 after running the program. The result is 90 in decimal, 5Ah in hex. As we can see, the -60 was loaded correctly into r2, therefore it produces an expected result (Figure 14).

A screenshot of a cell phone

Description automatically generated

Figure 14

By using the command “p/t $cpsr”, I can see that the only flag was set is the interrupt flag (Figure 14).

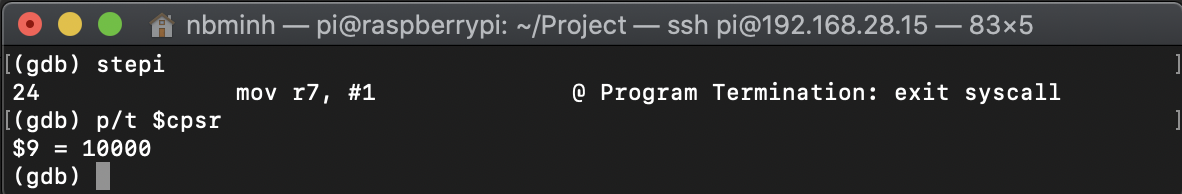


Figure 14

This table explains what the content of cpsr means (Table 1).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | thumb | fast | interrupt | overflow | carry | zero | negative |
| cpsr | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

Table 1