EE2016: Microprocessers Lab - Assignment 7 Report

Aadarsh Ramachandiran (EE23B001), Aditya Kartik (EE23B003), Rohita Datta (EE23B065)

October 6, 2024

Abstract

This is a detailed report on an assignment based on ARM Assembly Language Programming. The software used to compile and run the program is Keil µVision. The report contains the details of the solution, observation and comments on the programming and debugging experience.

1 Introduction

This assignment involves designing and implementing programs for: (i) understanding a given set of programs and observing the values in registers and error messages displayed (ii) finding the 10th term in a Fibonacci sequence and (iii) dividing a 32-bit number by a 16-bit number. We use ARM Assembly Language programs (.s extension) for all the tasks.

2 Design

2.1 Loading a number into a register

2.1.1 Description

The first task involves understanding a simple program that stores a given value into a register. The code is given in the next subsection. We put a breakpoint at the statement " B Stop" and observe the content of register R0 after executing the program via "Start/Stop Debug Session". The content of R0 register is noted down at the end.

2.1.2 Code

```
    AREA Program, CODE, READONLY
    ENTRY
        MOV r0,#11
Stop
        B Stop
    END
```

- AREA indicates the start of a new area (or segment) of code. Program is the name of the area (arbitrary choice), which can be referenced later. CODE specifies that the area contains executable code. READONLY indicates that the code in this area is not meant to be modified at runtime.
- ENTRY marks the entry point of the program
- The immediate value (11 in this case) is loaded into r0 via the mov command. The # symbol indicates that it is an immediate value rather than a reference to another register.
- Stop and B Stop effectively create an infinite loop as there are no conditions to exit the loop. Stop is a label and B Stop branches the flow of execution to the label Stop. This is to ensure the program keeps running. The significance of that will be realised on implementation.
- END marks the end of the program and tells the assembler that there are no more instructions to process.

2.2 Loading a larger number into a register

2.2.1 Description

We replace #11 with &FFFFFFFF in the program for task 1 and repeat the steps. The code is effectively the same.

2.2.2 Code

```
    AREA Program, CODE, READONLY
    ENTRY
        MOV r0,&FFFFFFFF
Stop
        B Stop
    END
```

- We suspect to receive an error message since MOV command can only load an immediate value of size atmost 8-bits.
- For some reason, we receive no error message. This suggests that the assembler correctly intepretes the larger number and handles it appropriately.
- As a good programming practice, ensure to use MOVW for larger numbers to be moved as immediate value.

2.3 Understanding Arithmetic Instructions

2.3.1 Description

The following code contains a bunch of arithmetic instructions and is run in Keil μVision. The values of r1 are recorded by single stepping through the code.

2.3.2 Code

```
    AREA Reset, CODE, READONLY
    ENTRY
        LDR r0, =7
        MUL r1, r0, r0
        LDR r2, =4
        MUL r1, r2, r1
        LDR r3, =3
        MUL r3, r0, r3
        ADD r1, r1, r3
stop
        B stop
    END
```

- LDR r0, =7 loads the immediate value of 7 into the register r0. The '=' indicates that this is a literal value and not a memory address (as is the case without the '=')
- MUL r1, r0, r0 multiplies the values stored in r0 and r0 (7x7 = 49) and stores it in register r1.
- ADD r1, r1, r3 adds the values stored in r1 and r3 (196+21 = 217) and stores it in r1.

2.4 Retrieving the 10th term of Fibonacci Sequence

2.4.1 Description

We desire to count in the pattern of Fibonacci sequence starting from 0 and 1 till we reach the 10th number. Each subsequent number is the sum of the previous two.

2.4.2 Code

```
; Program to find 10th fibonacci number
    AREA Reset, CODE, READONLY

        ENTRY
            LDR r0, =0 ; Initialized to fib(0). Holds fib(n-2)
            LDR r1, =1 ; Initialized to fib(1). Holds fib(n-1)
            LDR r3, =0 ; Initialized to 0. Holds fib(n)
            LDR r2, =10 ; Counter initialized to 10 since we need 10th
                fib number
LOOP
    ADD r3, r1, r0 ; Fib sequence. fib(n) = fib(n-1) + fib(n-2)
    MOV r0, r1 ; Update fib(n-2)
    MOV r1, r3 ; Update fib(n-1)
    SUB r2, r2, #1 ; Dec counter
    CMP r2, 2 ;  Check if counter has reached 2.
    BNE LOOP ;  Loop until 10th number is reached
STOP
    B STOP
    END
```

- Upon entry, we initialize the first two values as 0 and 1 into two registers r0 and r1. These would also be used to hold the (n-2)th and (n-1)th term every cycle as we count n. Register r3 is initialized to 0. It shall hold the nth term every cycle. The register r2 is initialized to 10 and acts as a counter.
- We iterate through a loop until the counter has decremented to 2 (since the first two terms is already accounted for). In each loop, we add the (n-2)th and (n-1)th term to get the nth term and update the former two with the updated (n-1)th and (n-2)th term. The counter variable is decremented by 1.
- The BNE command is used after the CMP command to compare r2 with 2. This implies that we branch as long as the value in r2 is not equal to 2. This facilitates the counting to get till the 10th number.

2.5 Division of two numbers

2.5.1 Description

We desire to use ARM assembly language to divide a 32-bit number by a 16-bit number and store the quotient as well as the remainder. The algorithm used is a simple repeated subtraction method. We subtract the divisor repeatedly from the dividend and update our quotient counter by 1 each time. Finally, when the dividend becomes smaller than the divisor, we store the dividend's value as our remainder.

2.5.2 Code

```
; Program to divide a 32-bit binary number by a 16-bit binary number
   and store the quotient as well as the reminder.
        AREA Program, CODE, READONLY
               ENTRY
                      LDR R0, Num2 ; Load Divisor
                      LDR R1, Num1 ; Load Dividend
                      MOV R2, #0 ; Initialize quotient to zero

Loop
```

```
    CMP R0, #0 ; Test division by 0
    BEQ Error1 ; Throw error if so
    CMP R1, R0 ; Compare Dividend and Divisor
    BLO Result ; If Dividend is less than Divisor, show result
    ADD R2, R2, #1 ; Else, inc Quotient by 1
    SUB R1, R1, R0 ; Subtract Divisor from Dividend
    B Loop

Error1
    MOV R3, #0xFFFFFF ; Error flag (-1)

Result
    LDR R4, = Remainder ; Get address to store Remainder
    STR R1, [R4] ; Store the Remainder (= The value of remaining
       Dividend)
    LDR R5, = Quotient ; Get address to store Quotient
    STR R2, [R5] ; Store the Quotient
    SWI &11

Num1 DCD &00000008 ; 32-bit Num1 - Dividend
Num2 DCW &0004 ; 16-bit Num2 - Divisor
        ALIGN

        AREA Data2, DATA, READWRITE ;  Write memory
Quotient DCD 0
Remainder DCD 0
```

- The values of Num2 (divisor) and Num1 (dividend) are loaded into the registers R0 and R1.
- R2, which stores the quotient, is initialized to 0. We then execute the loops of repeated subtraction.
- It branches to Error1 label if the divisor is 0, which sets R3 to 0xFFFFFF, which can be interpreted as the error flag.
- If not, it then compares the dividend and divisor. If the dividend is greater than the divisor, the divisor is subtracted from the current value of the dividend which is then stored in R1. The quotient increments by 1.
- If the dividend is lesser than the divisor, code branches to Result where the current dividend is stored as remainder in R4. Similarly, quotient is stored in R5.
  Note: Square brackets are a must in the syntax for STR
- The B Loop statement branches back to Loop to repeat the process until the result is obtained.
- Num1 and Num2 are defined as two data constants. Num1 here is an 8-bit constant, set to 8. Num2 is a 4-bit constant, set to 4.
- Another area called Data2 is created for writable memory where quotient and remainder are initialized to 0.
  Note: One must be very careful while handling the indentation in this program. The labels must have the first level of indentation and no other.

3 Implementation

This section outlines the steps to design and implement the ARM Assembly programs in µVision software.

- Ensure you have Keil µVision installed on your system, or install it if not already done.

- After installation, open the IDE.
- Create a New Project. Click **Project** → **New µVision Project**. Choose a folder to save your project, and give your project a name.
- Select the target device (LPC 2378). Keil has a wide database of microcontrollers from different vendors like STMicroelectronics, NXP, and Texas Instruments. The LPC 2378 is from NXP.
- Configure the Target Device. Once you select the microcontroller, µVision will automatically include the correct startup code and headers for your device. It will also set up your project with the appropriate compiler settings.
- Add Source Files. To add C or assembly code, go to **File** → **New** and create a new file. Write your ARM C or assembly code.
- Save the file and add it to your project by right-clicking on the project in the **Project window** → **Add Files to Group**.
- Compile the Code. Click **Project** → **Build Target** or press **F7**.
- You can now debug or test your code in simulation. Go to **Debug** → **Start/Stop Debug Session**. Use **Step Over** (F10), **Step Into** (F11), and **Run** (F5) to execute the program line by line. Use **View** → **Registers Window** to view the contents of ARM CPU registers like R0-R15, CPSR, etc.

4 Observation

4.1 Loading a number into a register

- After stepping into the line containing the MOV command we observe the contents of register r0 updated to contain 11 (0x0B) in the Registers Window. Initially it contained 0x00.

4.2 Loading a larger number into a register

- We observe the contents of the register r0 to contain FFFFFFFF.
- We do not see any warnings or error messages in the terminal palette.
- It is possible that the assembler implicitly corrects the command to MOVW to facilitate the action.

4.3 Understanding Arithmetic Instructions

- We single step through the code and note the values of r1 register each time from the registers window.
- Initially, it would be 0x00. When we reach the MUL command it is assigned the square of contents of r0. Hence, it is updated to 0x31.
- It is then updated to r2 * r1. Since r2 is 4, it becomes 4 * 49 = 196 (0xC4).
- In the final ADD instructions r1 is updated to r1 + r3. Since, r3 would be 21, r1 is assigned 217 (0xD9).

4.4 Retrieving the 10th term in Fibonacci Sequence

- We observe the contents of register r3, update as 0, 1, 2, 3, 5, 8, 13, 21, 34, 55 in sequence.
- Finally, at the end of the program, the register r3 would contain 55 (0x37) which is the 10th Fibonacci term as desired.

4.5 Division of two numbers

- In our sample program, we set the dividend as 8 and divisor as 4. So, r0 is initialized to 4, r1 to 8 and r2 to 0.
- As we step through the program, first, r1 becomes 4, r2 is updated to 1, and again, r1 is modified to 0, r2 to 2. After this, r1 becomes less than r0, and the program stops. The reminder from r1 and the quotient from r2 is stored into memory.
- We find the correct values stored in the data segment seen through the memory window.

5 Conclusion

The experiment successfully demonstrated programming using ARM assembly language by attempting various tasks from simple register storage, arithmetic programs to finding the 10th term of a Fibonacci sequence and dividing a 32-bit number by a 16-bit number. All the code written for this assignment can also be found here.