Introduction to ARM Assembly - solutions

Anas Shrinah

December 5, 2022

Solutions.

We have learnt how a computer is built from the bottom up, how the logic can be implemented in hardware. We also tested simple assembly codes with our 4-bit CPU. Now is the time to move on to a more advanced CPU. For the next step of the unit, we will write assembly programs using the ARM instruction set. To test these codes, we need an ARM processor. If you have a computer with an ARM-based processor, then you can start coding in RAM assembly straightaway. However, if you do not have access to a machine with an ARM processor, you can use an ARM emulator.

Credits: The content of this worksheet is based on lab materials originally prepared and developed by previous lecturers of COMSM1302 at the University of Bristol.

Goals of this lab

In this lab:

- You learn how to use the GNU tool chain to execute and debug assembly programs.
- You will learn the main directives in ARM assembly codes.
- You will get the chance to practice what you have learnt in the lectures about the ARM instruction set.
- You will get the opportunity to look at your code with the eye of a verification engineer.
- In summary, this lab will equip you with the essential practical skills to write ARM assembly programs.

3 Programs

Now we have gotten the hang of how the execution works and can run a program. Next, we are going to write assembly codes to solve the following problems.

3.1 The determinant of a 2 * 2 matrix

Write a program to compute the determinant of a 2×2 matrix.

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = a * d - b * c$$

1. Load the registers r0,r1,r2, and r3 with the values of a,b,c, and d respectively.
2. Calculate $a * d$
3. Calculate $b * c$
4. Calculate $a * d - b * c$
5. Add an infinite loop at the end of your code.

Suggest some 2 * 2 matrices to test your code. It is usually a good idea to test your code with border cases.

To determine the border cases of your code answer the following questions:

1. Are we using signed or unsigned representation?
    - We can use singed representation if we are guaranteed that the variables are positive and $a*d \geq b*c$. Otherwise we need to use signed representation.
2. What is the range of the values of the variables a,b,c, and d?
    - As we are using signed representation, the range is from -2,147,483,648 (0x8000000) to 2,147,483,647 (0x7FFFFFFF).
3. What is the range of the determinate that we can calculate?
    - Considering we are using 32 bits registers, with signed representation, the range is from -2,147,483,648 (0x8000000) to 2,147,483,647 (0x7FFFFFFF).
4. What is the range of the register that will hold the result of the intermediate calculation $a * d$?
    - As we are using signed representation, the range is from -2,147,483,648 (0x8000000) to 2,147,483,647 (0x7FFFFFFF).
5. Can we guarantee the result of $a * d$ or $b * c$ can fit in 32-bit registers? Why?
    - No, because multiplying two 32-bit numbers can result in 64 bits.
6. Give an example of two inputs of a and b that would cause $a * d$ to not fit in a 32-bit register.
    - $0x80000000 \times 0x2 = 0x800000000$ (the result needs 33 bits)

In this example, we assume it is the user responsibility to understand and use our code properly. In the extra task, you will be asked to automate the process of checking the correctness of the result of your code.

The code:

```
@ a b
@ c d

@ det = a*d - b*c
@ load the values a, b, c, and d to the registers r0, r1, r3, and r4, respectively.
@ Note I have used ldr rather than mov because the instruction move cannot load big constants.
ldr r0,=0x1 @ a
ldr r1,=0x2 @ d
ldr r2, =0x3 @ b
ldr r3,=0x4 @ c


mul r4,r0,r3 @ Multiply a by d and store the result in r4.
mul r5,r1,r2 @ Multiply b by c and store the result in r5.

sub r9,r4,r5 @ Subtract b*c (r5) from a*d (r4) and store the result in r9.
```

3.2 The 3n+1 problem

The 3n+1 problem is a mathematical open problem. Given a positive integer n, if you repeatedly do the following:

1. If n is even, replace it by n/2.
2. If n is odd, replace it by 3n+1.

For every positive integer tried so far, this algorithm eventually reaches n = 1, at which point it cycles 1 - 4 - 2 - 1 forever. The open problem is whether this property holds for all positive integers. We will not be able to answer this question, but your assignment is the following:

Write an ARM assembly program to runs the 3n+1 procedure above. Your program should halt if the value 1 is reached. You can assume that we will be working with small enough values that there will be no integer overflow.

Please note that you do not need to use division or multiplication instructions for this code.

Solution 1

```
.section        .text
.align          2
.global         _start

_start:

        mov     r1,#7 @ this is the input number

_loop:
        cmp r1, #1 @ check if r1 (n) is equal to 1
        beq _end @ if the r1 == 1, jump to the infinite loop to halt the program execution.

        tst r1, #1 @ TST with 1, it does a logical AND with 1.
        @ This instruction will zero out all bits but the rightmost one (LSB)
        @ 0000 0111  &  0000 0001 = 0000 0001 for odd numbers -> z flag will be cleared
        @ 0000 0100 &   0000 0001 = 0000 0000 for even numbers Z flag will be set
        bne _odd @ bne gets executed if the zero flag is not set.
        @ In our case the zero flag is not set (i.e. cleared ).
        @ if r1 was odd number when we executed tst r1, #1.


        @ If the execution reaches the following instruction, then r1 is even.
        @ If r1 was odd, the previous bne _odd would have been executed
        @ and the execution would have jumped to the label _odd.
        mov r1,r1, asr #1 @ Replace r1 by r1/2. Remember asr #1 divides r1 by 2. if r1 = 0100,
        @ shifting r1 by 1 to the right causes r1 to be come r1 = 0010.

        b    _loop @ The execution reaches this instruction if r1 was even and r1 was
        @ replaced by r1/2 or if r1 was odd and we did @ r1 = 3 r1 + 1. check the code below.

        @ b _loop causes the execution to go to back to testing if r1 has odd or even value

_odd:
        @if r1 is odd: r1 = 3 r1 + 1
        @ r1 = 3 r1 + 1
        @ the execution reaches this instruction if r1 has an odd number.
        add r1, r1, r1, lsl #1 @ remember lsl #1 multiplies r1 by 2.
        @lsl #1 shifts r1 by one to the left. if r1 = 0101,
        @ lsl #1 of r1 will cause r1 to becomes r1 = 1010.

        add r1, r1, #1@ add one to r1

        b       _loop @ go back to testing the value of r1 if it is odd or even.

_end:   b       _end @ infinite loop
```

Solution 2

```
.section        .text
.align          2
.global         _start

_start:

        mov     r1,#7 @ this is the input number

_loop:
        cmp r1, #1 @ check if r1 (n) is equal to 1
        beq _end @ if the r1 is equal to 1, jump to the infinite loop to
        @ halt the program execution.

        tst r1, #1

        moveq r1 ,r1, r1 asr #1 @ if tst with one caused the z flag to be set,
        @ then r1 has an even number, check the description in the code above.
        @ Hence, we have to divide r1 by 2. Note moveq only gets executed if z was set.
        addne r1, r1, r1 lsl #1 @ if tst with one caused the z flag to be cleared,
        @ then r1 has an odd number, check the description in the code above.
        @ Hence, we have tmultiply r1 by three and then add one to the result.
        @ Note addne only gets executed if z was cleared.
        addne r1, r1, #1
        b _loop

_end:  b _end
```

3.3 Great Common Divisor - Euclidian Algorithm

The greatest common divisor of two positive integers a and b is the largest divisor common to a and b. For example, GSD(3,7) = 1, GSD(2,60) = 2, and GSD(60,75) = 15.

A nice and algorithmic way of calculating GCD is the Euclidian Algorithm. For more information about this algorithm check this video https://www.youtube.com/watch?v=JUzYl1TYMcU.

Your task is to research this method and write an assembly code to calculate the GCD for two positive numbers. First, move the value of the first number to r0 and the value of the second number to r1. The code should store the GCD of r0 and r1 in r2 before it goes to an infinite loop (halt).

```
.section .text
.align    2
.global   _start

_start:     mov r0, #10 @ first number
            mov r1, #45 @ second number
            bl _calc_GCD @call the great common divisor calculating subroutine r3 is the output.

            b _end @ go to the infinite loop




_calc_GCD:
```

```
            cmp r0,r1 @ check what number is greater.
            @ Store the grater number in r2 and the smaller number in r3
            movgt r2,r0 @ if r0 is greater than r1, move it to r2.
            movgt r3,r1 @ if r0 is greater than r1, move r1 to r3.
            movlt r2,r1 @ if r0 is smaller than r1 move r1 to r2.
            movlt r3,r0 @ if r0 is smaller than r1, move it to r3.



_loop:

            udiv r4,r2,r3 @ divide the grater number by the smaller number (integer division)
            @ find the remainder of the division remainder =  dividend - (quotient * divisor).
            mul  r5,r4,r3 @ This instruction calculates the (quotient (r4) * divisor (r3))
            @  and store the result in r5.
            sub  r6,r2,r5 @ find the remainder of the division
            @"dividend (r2) - (quotient * divisor) (r5)" and store the result in r6.

            cmp r6,#0 @ if the remainder equals zero, terminate. r3 is the GCD.
            moveq pc,lr @ return to the main code.

            mov r2,r3 if the remainder is not equal to zero, repeat the loop.
            @ Use the divisor as the dividend a
            mov r3,r6 @ use the remainder as the divisor

b _loop


_end:                   b _end @ an infinite loop
```

4 Extra task

Please only attempt these tasks when you have done all other sections.

4.1 Improve the determinant of a 2 * 2 matrix code to make it sound

In Section 3.1, we assumed it is the responsibility of the end-user to use our program wisely. But the truth is, as programmers, it is our responsibility to specify the behaviour of our code. We have two options here.

- List down all the ranges and combinations of the values for which we guarantee our code to produce correct results, or
- give a warning to the user when we are not sure that our program will return the correct answer for the given inputs.

Think about the ranges and limits where the calculation $a * d$, $b * c$ and $a * d - b * c$ can go wrong because of using 32-bit registers.

Update your code from Section 3.1 to give a warning to the user when the inputs might cause the result to be invalid. Use r8 as a warning flag. Put 1 in r8 if the result of the calculation is correct. Put -1 in r8 if the result is wrong.

This code does not aim to provide a complete solution, but it rather motivates you to think about the corner cases that might arise from such problem.

```
.section .text
.align    2
.global   _start

_start:



@ This example shows r0 and r3 are of different signs
@ldr r0,=0x80000001
@ldr r1,=0xfffffffe
@ldr r2, =0x7fffffff
@ldr r3,=0x2

@ This example shows r0 and r3 are both negative
@ldr r0,=0x80000001 @ a
@ldr r1,=0xfffffff9 @ d
@ldr r2, =0x5 @ b
@ldr r3,=0xfffffffe@ c


@ This example shows r0 and r3 are both positive
ldr r0,=0x7fffffff
ldr r1,=0xfffffffe
ldr r2, =0x7fffffff
ldr r3,=0x2




mul r4,r0,r3 @ a*d
mul r5,r1,r2 @ b*c

sub r12,r4,r5 @ a*d - b*c


@ The following code checks if the result of a * d can fit in 32 bits
@ using signed representation.
@
@ The are three cases: a and d are both positive, both negative, or they have different signs.
@ If a and d are positive, we have to divide the the largest possible number (0x7fffffff)
@ by the smallest number between a and d. The result must be less than the largest number
@ between a and d. This is required to insure the multiplication of a and d is less than
@ the largest positive number.
@ If both a and d are negative, convert them to positive and
@ process them as if they were positive.
@ If they have different signs, divide the lowest possible number 0x80000000
@ by the lowest number between a and d.
@ if the largest number between a and d is larger than or equal to the result of dividing
@ the lowest negative number by the lowest number between a and d,
```

```
@ then we can guarantee multiplying a and d will result in a number that
@ is less than the lowest possible number. Hence the result might be wrong.


@ if a or d equal to the lowest negative number, then check the other number if it is not 1,
@ then the result of a * d will be wrong. This is a special case, not coded in this program.

@ you have also to check the the flags after doing the subtraction a*d - b*c
@ to make sure the result is correct.


@ I have not implemented the code to check b and c.
@ Same concept can be used. Implement a routine call to reuse the code.

ldr r4,=0x80000000 @ the lowest negative  32-bits number
ldr r5,=0x7fffffff @ the largest positive 32-bits number



@find the largest number between r0 and r3.
@Assign the largest number to r7 and the smallest to r6.
cmp r0,r3
ble _r0_less_equal_r3
@ if the execution reached here, then _not_r0_less_equal_r3 i.e r0 (a) is the grater than r3 (d)
mov r6,r3 @ r6 is the lowest number = r3 (d)
mov r7,r0 @ r7 is the largest number = r0 (a)
b _check_if_both_poitive_negative_or_different_sgins


_r0_less_equal_r3:
mov r6,r0 @ r6 is the lowest number = r0 (a)
mov r7,r3 @ r7 is the largest number = r3 (d)



_check_if_both_poitive_negative_or_different_sgins:
@ check if r0 and r3 are both positive or both negative OR they have different signs.
and r9,r0,r4 @ r4 =0x80000000 (lowest negative number):
@r9 = 0x80000000 if a (r0) is negative or r9 = 0 if r0 is positive

and r10,r3,r4 @ r4 =0x80000000 (lowest negative number):
@ r10 = 0x80000000 if d (r3) is negative or r10 = 0 if r3 is positive

cmp r9,r10 @check if a and d both are positive or negative.
beq _both_postive_negative @both positive or negative
@ if the execution reached here, then a and d have different signs


sdiv r11,r4,r6 @ divide the lowest negative number by the lowest number
@ (here the lowest number between a and d must be negative
@ because here a and d have different sign)
cmp r7,r11 @ if the largest number between a and d is larger than or equal to
```

```
@ the result of dividing the lowest negative number by the lowest number (a or d),
@ then we can guarantee multiplying a and d will result in a number that is less than the
@ lowest possible number. Hence the result might be wrong.
bgt _wrong_result @ because the multiplication of two number will not fit in 23 bits.




_both_postive_negative:

tst r0,r4
beq _both_postive
@ here both negative
mov r8,#-1
mul r9,r6,r8 @ convert a which is negative to positive a
mul r10,r7,r8 @ convert d which is negative to positive d
mov r6,r9
mov r7,r10


_both_postive:
udiv r6,r5,r6
cmp r7,r6
bgt _wrong_result @ because the multiplication of two number will not fit in 23 bits.



mov r8,#0
b _end

_wrong_result:
mov r8,#1
b _end




_end:                   b _end @ an infinite loop
```