EE2016: Microprocessors Lab
Experiment # 5:ARM Assembly - Computations in ARM

Amizhthni PRK, EE21B015
Nachiket Dighe, EE21B093

October 23, 2022

Contents

List of Figures

1 Equipments used

- KEIL 5 IDE for ARM
- Flashmagic software for programming flash memory
- This experiment used just emulation, needed no hardware components such as ARM7 hardware kit, USB to serial converter and Serial cross cable.
- We thoroughly went through the first 6 chapters of Welsh ARM textbook.

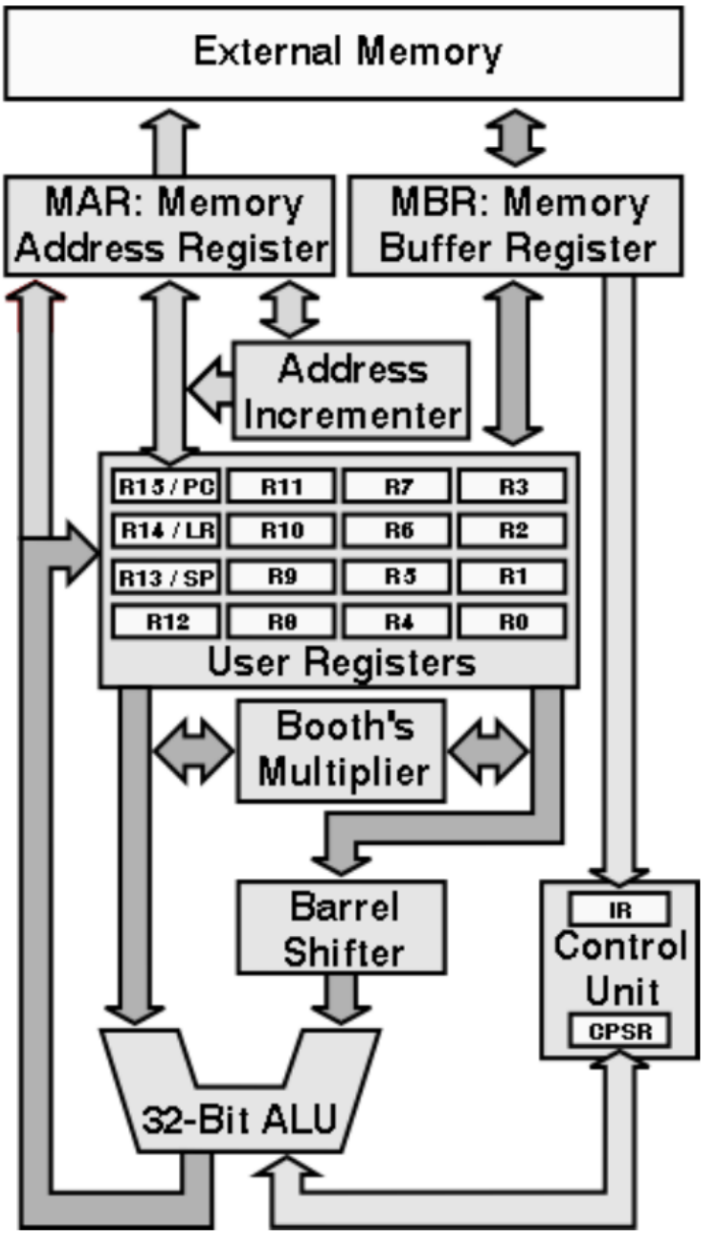

Figure 1: The internal structure of ARM processor

Operation Mnemonic	Meaning	Operation Mnemonic	Meaning
ADC	Add with Carry	ORR	Logical OR
ADD	Add	RSB	Reverse Subtract
AND	Logical AND	RSC	Reverse Subtract with Carry
B	Unconditional Branch	SBC	Subtract with Carry
Bcc	Branch on Condition	SMLAL	Mult Accum Signed Long
BIC	Bit Clear	SMULL	Multiply Signed Long
BL	Branch and Link	STM	Store Multiple
CMP	Compare	STR	Store Register (Word)
EOR	Exclusive OR	STRB	Store Register (Byte)
LDM	Load Multiple	SUB	Subtract
LDR	Load Register (Word)	SWI	Software Interrupt
LDRB	Load Register (Byte)	SWP	Swap Word Value
MLA	Multiply Accumulate	SWPB	Swap Byte Value
MOV	Move	TEQ	Test Equivalence
MRS	Load SPSR or CPSR	TST	Test
MSR	Store to SPSR or CPSR	UMLAL	Mult Accum Unsigned Long
MUL	Multiply	UMULL	Multiply Unsigned Long
MVN	Logical NOT		

Figure 2: The exhaustive set of commands used

Mnemonic	Condition	Mnemonic	Condition
CS	Carry S et	CC	Carry Clear
EQ	Equal (Zero Set)	NE	Not Equal (Zero Clear)
VS	Overflow Set	VC	Overflow Clear
GT	Greater Than	LT	Less Than
GE	Greater Than or Equal	LE	Less Than or Equal
PL	Plus (Positive)	MI	Minus (Negative)
HI	Higher Than	LO	Lower Than (aka CC)
HS	Higher or Same (aka CS)	LS	Lower or Same

Figure 3: Condition code mnemonics

2 Factorial of a Number

2.1 Aim in this part 1 of experiment:

Compute the factorial of a given number using ARM processor through assembly programming.

2.2 Algorithm, Approach, Code Explanation

- We start with a code block named FACTORIAL
- We store the given input value whose factorial is to be calculated in register R0.
- In this case, we store 6 in R0.
- The BL instruction copies the address of the next instruction into r14 (lr, the link register), and causes a branch to label.
- Therefore, only once the register 3 stores the value 1. Thus this is a subroutine.
- CMP – Compare: subtracts a register or an immediate value from a register value and updates condition codes.
- CMP sets Z flag if R0 is 1
- R3 is initially 1. It is multiplied by the value in R0.
- R0 is initially set at 6. The next value of R3 is thus 6.
- R0 is reduced by 1, thus it becomes 5. This register also acts as the count.
- Branch if greater than command, i.e if R0$\geq$1, then loop branches and it continues.
- In the next loop, R3 becomes 6 $\times$ 5 and so on.
- After 6 such loops, the BGT condition fails and MOV instruction is used.
- Register R14 is also known as the Link Register or LR. It is used to hold the return address for a subroutine. When a subroutine call is performed via a BL instruction, R14 is set to the address of the next instruction. To return from a subroutine you need to copy the Link Register into the Program Counter.
- Clearly, R3 contains 0X2D0 which is 720 in hexadecimal.
- R14 and R15 have the same value.
- Register R15 holds the Program Counter known as the PC. It is used to identify which instruction is to be preformed next. As the PC holds the address of the next instruction it is often referred to as an instruction pointer.
- When an instruction writes to R15 the normal result is that the value written is treated as an instruction address and the system starts to execute the instruction at that address3.
- Current Processor Status Registers: CPSR Rather surprisingly the current processor status register (CPSR) contains the current status of the processor. This includes various condition code flags, interrupt status, processor mode and other status and control information.

- The exception modes also have a saved processor status register (SPSR), that is used to preserve the value of the CPSR when the associated exception occurs. Because the User and System modes are not exception modes, there is no SPSR available.

2.3 Code

The code for finding the factorial of a given number is given in listing 1. An extra method using subroutines is mentioned under 2. We have not used the code in our experiment yet.

```
    AREA factorial,CODE,READONLY
                                ; Name this block of code factorial

ENTRY                           ; Mark first instruction to execute
        MOV R0,#6                   ; Set up parameters
                                    ; R0 takes in value of input 6
        BL FACT
B1  B B1                        ; Subroutine call is performed via BL
 ↪  instruction,

FACT MOV R3,#1                  ; Set up parameters
                                ; R3 takes up value 1
loop CMP R0,#1                  ; Set up parameters
        MULGT R3,R0,R3              ; R3 is multiplied by the next numbers
        SUBGT R0,R0,#1              ; R0 has the count, R0 is reduced by 1
                                    ; every time a number is multiplied
        BGT loop                    ; When R0 is non zero
                                    ; the line control shifts to the block
 ↪  named loop
        MOV PC,R14                  ; Link Register or LR is used to
                                    ; hold the return address for a
 ↪  subroutine
                                ; R14 is set to the address of the next
 ↪  instruction.
                                ; To return from a subroutine you need to
 ↪  copy the
                                ; Link Register into the Program Counter.
        END                         ; End of program
```

Listing 1: Code for factorial finding

```
*           a subroutine to find the factorial of a number

             TTL Ch10Ex6
             AREA Program, CODE, READONLY
             ENTRY

Main
             LDR R0, Number ;get number
             BL Factor ;branch/link
             STR R0, FNum ;store the factorial
```

```
11
12                SWI &11 ;all done
13
14  *             ========================
15  *             Factor subroutine
16  *             ========================
17
18  *            Purpose
19  *            Recursively find the factorial of a number
20  *
21  *            Initial Condition
22  *            R0 contains the number to factorial
23  *
24  *            Final Condition
25  *            R0 = factorial of number
26  *
27  *            Registers changed
28  *            R0 and R1 only
29  *
30  *             Sample case
31  *             Initial condition
32  *             Number = 5
33  *
34  *             Final condition
35  *             FNum = 120 = 0x78
36
37  Factor
38                STR R0, [R12], #4 ;push to stack
39                STR R14, [R12], #4 ;push the return address
40                SUBS R0, R0, #1 ;subtract 1 from number
41                BNE F_Cont ;not finished
42
43                MOV R0, #1 ;Factorial == 1
44                SUB R12, R12, #4 ;adjust stack pointer
45                B Return ;done
46
47  F_Cont
48                BL Factor ;if not done, call again
49  Return
50                LDR R14, [R12], #-4 ;return address
51                LDR R1, [R12], #-4 ;load to R1 (can't do MUL R0, R0, xxx)
52                MUL R0, R1, R0 ;multiply the result
53                MOV PC, LR ;and return
54
55                AREA Data1, DATA
56  Number DCD 5        ;number
57  FNum DCD 0 ;factorial
58                END
```

Listing 2: Code for factorial finding using subroutines

2.4 Outputs

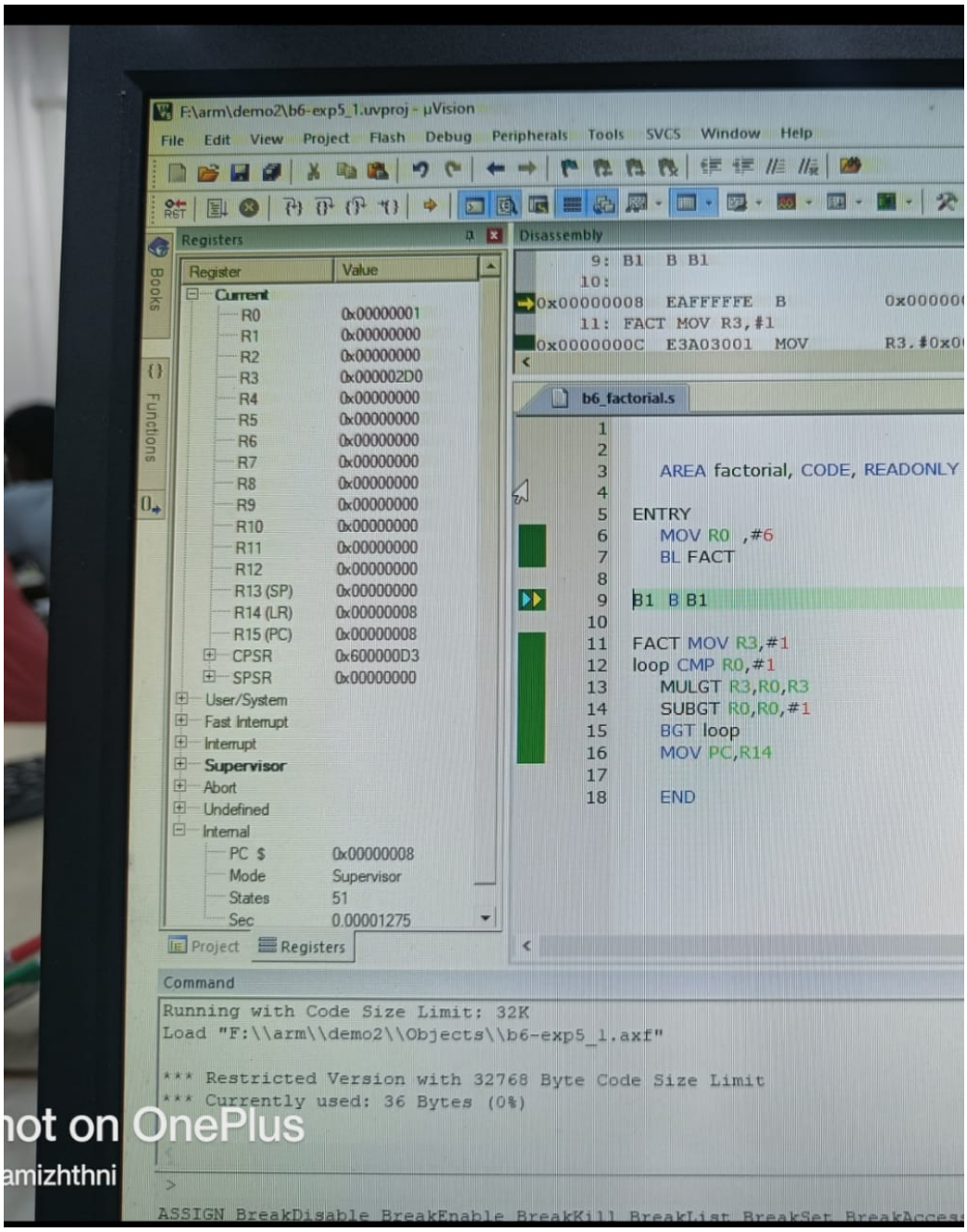

Figure 4: Output(R3 = 0x2D0) for input of 6

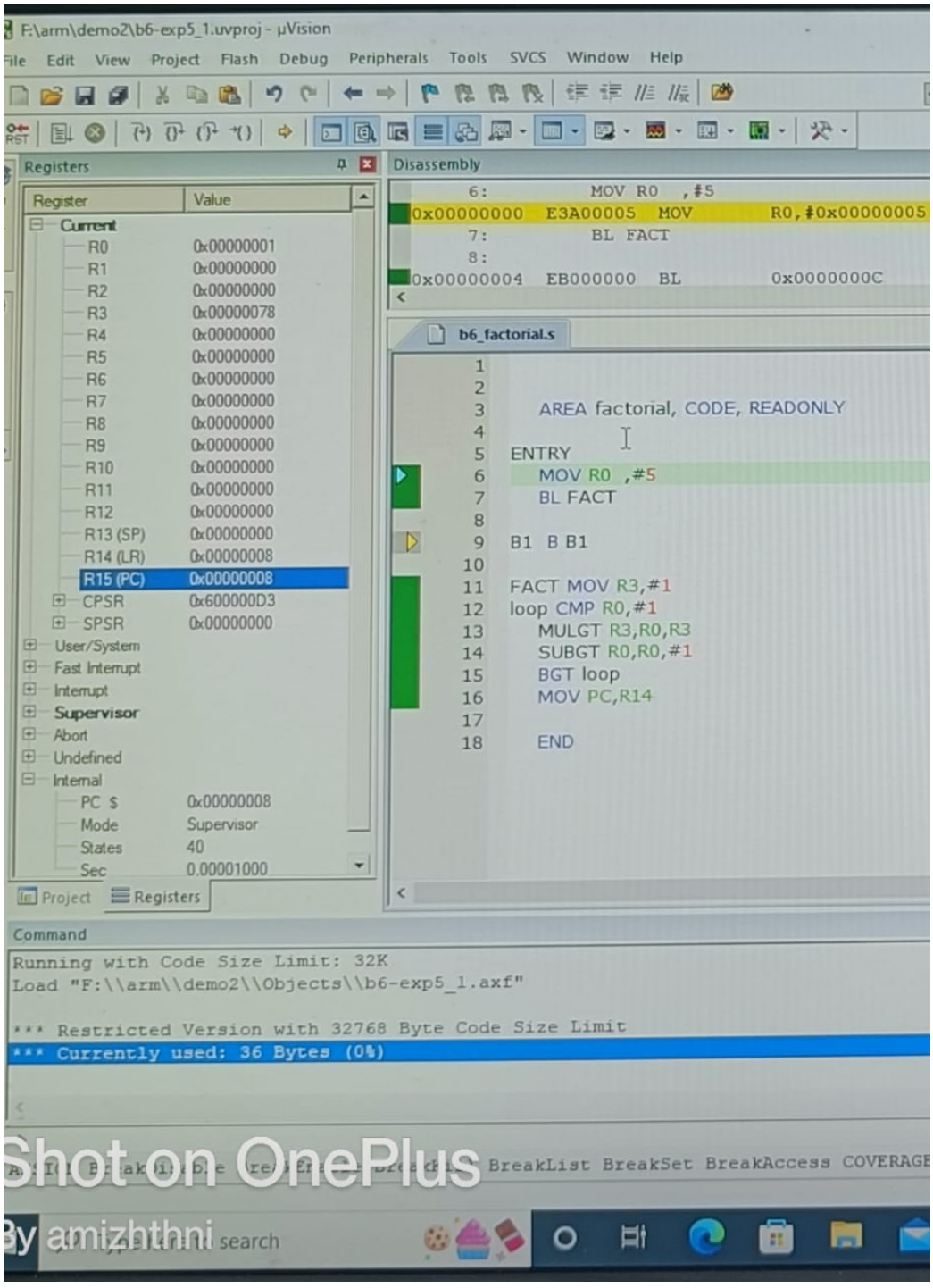

Figure 5: Output(R3 = 0x078) for input of 5

3 Combination of lower 4-bits of each Byte of a given 4-byte LIST

3.1 Aim in this part 2 of experiment:

- Combine the low four bits of each of the four consecutive bytes beginning at LIST into one 16-bit half word. The value at LIST goes into the most significant nibble of the result. Store the result in the 32-bit variable RESULT.

3.2 Algorithm

- START
- Store the contents of the given 4-byte LIST in a Register R0
- Initialize R3 to 12 for using it to rotate the contents of Register R0
- Logically AND R0 with 0x0F so that the Least significant nibble is stored in R4
- To combine the given result we Logically OR register R4 with 0
- Since this should be our most Significant Nibble, rotate it Left by 12 bits
- Change the counter R3 by 4(for making it 2nd Most significant Nibble)
- Do this Process until R3 becomes 0
- Store the given Result in R1
- END the program

3.3 Code

The code used for combining Elements of an array is given below in listing 3.

```
1  AREA program,CODE,READONLY
2           ENTRY
3  Main
4          LDR R0, LIST ;Store contents of array LIST in register R0
5          LDR R1,=0 ;Initialize R1 to 0
6          LDR R3,= 12 ;Initialize R3 to 12 for rotation of bits
7
8  loop
9          AND R4, R0, #&0F ;To get the last 4 bits and store in R4
10         MOV R4, R4, LSL R3 ;Shift the result by 3 nibbles
11         ORR R1, R4
12         SUB R3, R3, #4
13          MOV R0, R0, LSR #8 ;To get the last 4 bits of the 2nd element of
   ↪  LIST
14         CMP R0, #0
15         BNE loop
16
17         LDR R2,=Result ;
18         STR R1, [R2]
19     LSL R1,#16 ;To store the result in first 16-bits of 32-bits
```

```
20
21          SWI &11
22
23  LIST DCB &61, &22, &13, &84
24
25   AREA DataRAM, DATA, READWRITE
26  Result DCD 0
27
28          END
```

Listing 3: Combining lower 4-bits of each Byte of LIST

3.4 Explanation

- When the list of 4 bytes are 01,02,03 and 04.
- The lower four bits are 1,2,3 and 4. They are stored in R1 as a combined number.

3.5 Output figures

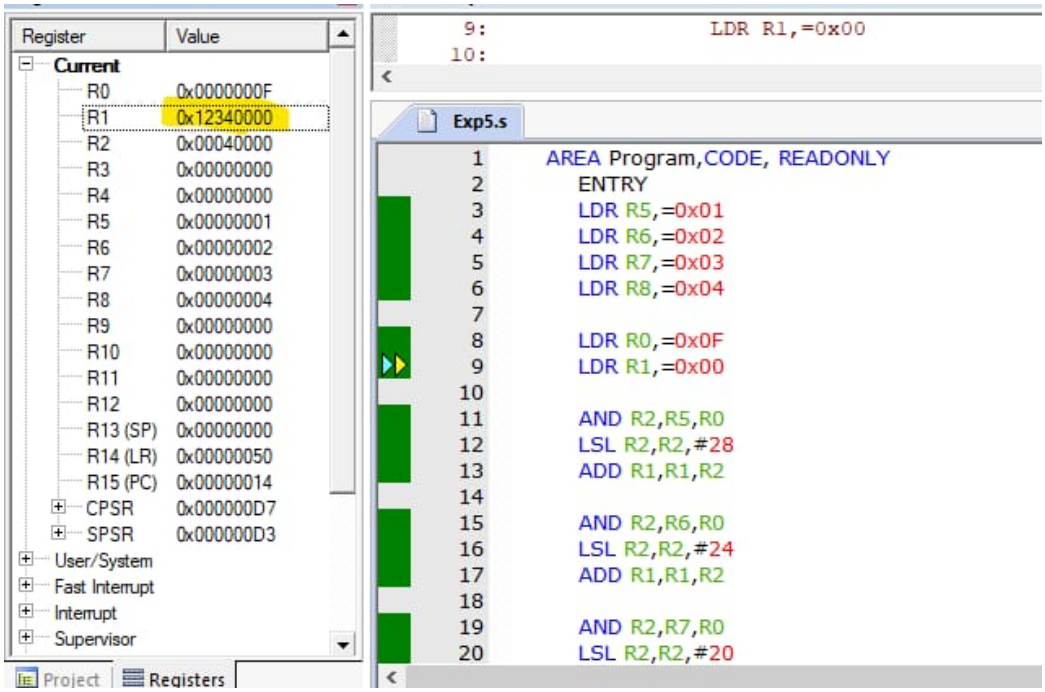

Figure 6: Output in Register R1 when we take input as 0x01, 0x02, 0x03, 0x04

4 To check Odd/Even number

4.1 Aim in this part 3 of experiment:

- Given a 32 bit number, identify whether it is an even or odd. (Your implementation should not involve division).

4.2 Algorithm, Approach, Code Explanation

- We name the code block PROGRAM.
- R0 is the register to be loaded.
- If the value of expression is within range of a MOV or MVN instruction, the assembler generates the appropriate instruction.
- If the value of expression is not within range of a MOV or MVN instruction, the assembler places the constant in a literal pool and generates a program-relative LDR instruction that reads the constant from the literal pool.
- Here the pseudo instruction is used to generate literal constants when an immediate value cannot be moved into a register because it is out of range of the MOV and MVN instructions.
- R0 stores the input number
- R1 has value 1. When R0 is used in an AND logical operation with 1 (It is a 32 bit logical bitwise AND operation), the LSB is stored in R2.
- STR instructions store a register value into memory.
- **Why this works**: When given number is odd, the last bit(LSB) is 1. 1 AND 1 is 1.
- If R2 is 1, the number is ODD, This is indeed the case when input is 0xFF.
- **Why this works**: When given number is even, the last bit(LSB) is 0. 0 AND 1 is 0.
- If R2 is 0, the number is EVEN, This is indeed the case when input is 0x0A.

4.3 Code

The code used for finding whether the given number is odd/even **??**.

```
1  AREA program,CODE,READONLY
2
3        ENTRY
4        LDR R0,=0x0A ;input
5            LDR R1,=0x01 ;Register to logically AND with input
6            AND R2,R0,R1 ;logical AND with R1 to find the LSB of the input
   ↪  and store in R2
7            STR R3,[R2];If R0 is 1 then the number is odd, if R0 is 0 number
   ↪  is even
8
9        END
10
```

Listing 4: 32 bit odd/even

4.4 Output figures

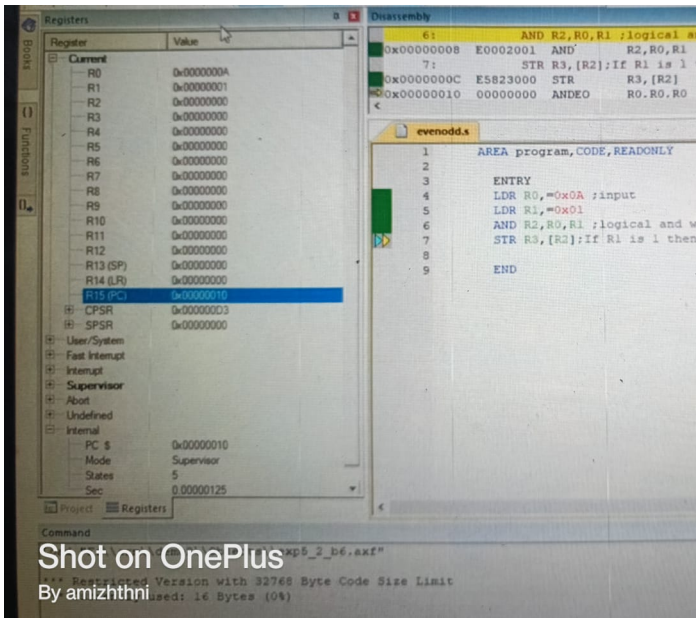

Figure 7: When input is 0x0A, result in Register R2 = 0x00

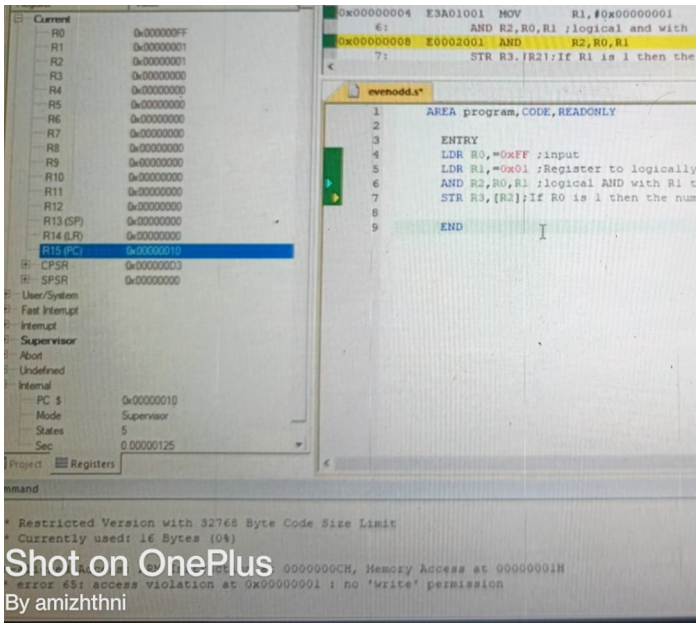

Figure 8: When input is 0xFF,result in register R2 = 0x01