Emulating ARM assembly programming on Keil MicroVision

EE2016 Microprocessors

Sanjeev Subrahmaniyan S B

EE23B102

Abstract

ARM processors are the buzzword in the microprocessor industry now, with most devices having processor built on the ARM RISC architecture. They are particulary common in embedded and handheld devices like microcontrollers and smartphones. ARM assembly language programming offers a mode to program the processors at a very low level, exercising control over how the registers and the processor works, thereby providing extremely high performance speeds. In this experiment, basic to advanced tasks are emulated on a ARM Cortex M7 processor using the assembly language instructions. The tasks including checking evenness of a number, processing 8 bit numbers on a 32 bit processor and computing the factorial of a number.

1 Introduction

Microcontrollers are indispensable in the electronic industry, being the base for all prototyping and testing. In this experiment, the Cortex M7 ARM processor is emulated on KEIL Microvision software. This gives one the opportunity to gain familiarity with the ARM assembly language and write elementary programs and simulate them before working with the hardware implmentation. It also enables one to learn to use the Keil MicroVision software for emulation purposes - including viewing registers and so on.

2 Objectives

The tasks emulated are:

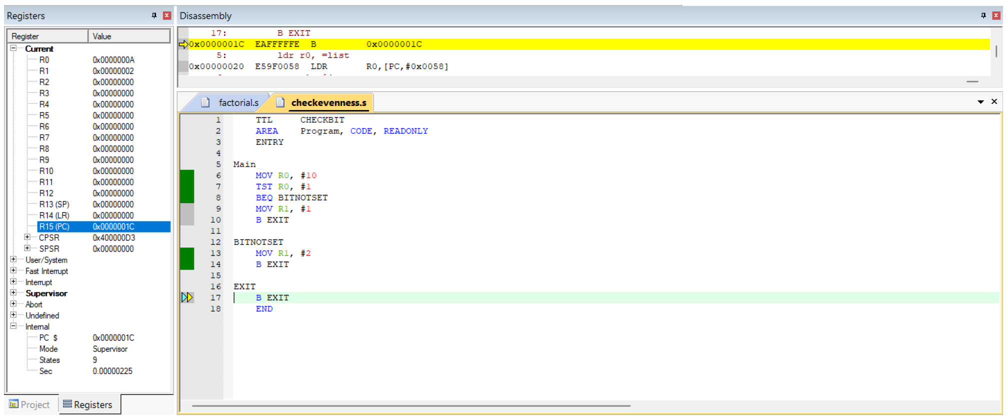
1. Checking if a 32-bit number is even or odd without dividing or bitshifting.

2. Joining the first nibbles and second nibbles of 8 bit numbers together for a given list of 4 8 bit numbers.

3. Evaluating the factorial of a number.

3 Assembly language programs and interpretation

3.1 Checking for even-ness of a number



Register	Value
Current	
R0	0x0000000A
R1	0x00000002
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x0000001C
CPSR	0x400000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	
PC $	0x0000001C
Mode	Supervisor
States	9
Sec	0.00000225

```
    17:          B EXIT
0x0000001C  EAFFFFFE  B         0x0000001C
     5:          ldr r0, =list
0x00000020  E59F0058  LDR       R0,[PC,#0x0058]
```

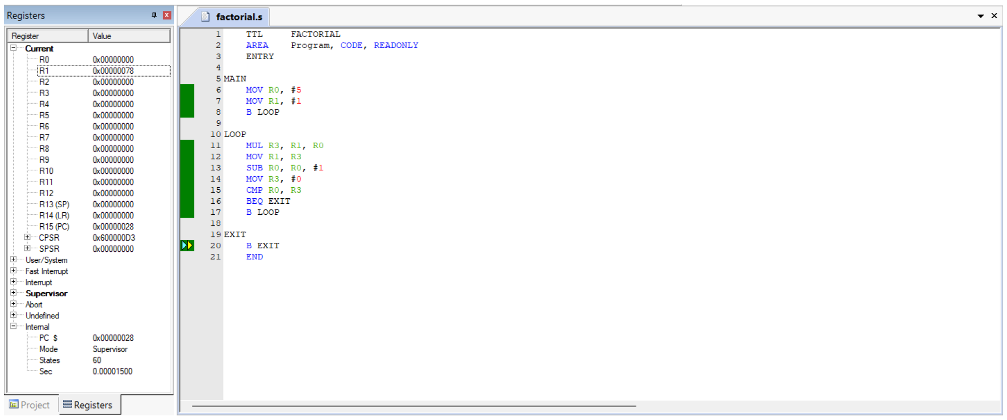
```
    TTL     CHECKBIT
    AREA    Program, CODE, READONLY
    ENTRY

Main
    MOV R0, #10
    TST R0, #1
    BEQ BITNOTSET
    MOV R1, #1
    B EXIT

BITNOTSET
    MOV R1, #2
    B EXIT

EXIT
    B EXIT
    END
```

Figure 1: Checking if a given number is even or odd

1. In the ARM program given above, the number is checked if it is even or odd by testing the least significant bit of the number. If this bit is set, the number is odd and even otherwise.

2. This logic is implemented and accordingly 1 or 2 is moved into a register to indicate the parity of the number under consideration.

3.2 Evaluating the factorial of a number and storing it in memory



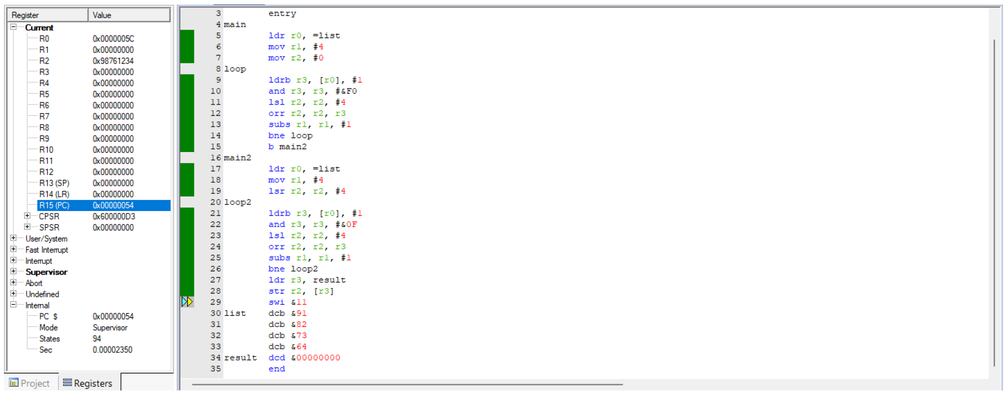
```
Registers
Register        Value
Current
  R0            0x00000000
  R1            0x00000078
  R2            0x00000000
  R3            0x00000000
  R4            0x00000000
  R5            0x00000000
  R6            0x00000000
  R7            0x00000000
  R8            0x00000000
  R9            0x00000000
  R10           0x00000000
  R11           0x00000000
  R12           0x00000000
  R13 (SP)      0x00000000
  R14 (LR)      0x00000000
  R15 (PC)      0x00000028
  CPSR          0x600000D3
  SPSR          0x00000000
User/System
Fast Interrupt
Interrupt
Supervisor
Abort
Undefined
Internal
  PC $          0x00000028
  Mode          Supervisor
  States        60
  Sec           0.00001500
```

```
    TTL     FACTORIAL
    AREA    Program, CODE, READONLY
    ENTRY

MAIN
    MOV R0, #5
    MOV R1, #1
    B LOOP

LOOP
    MUL R3, R1, R0
    MOV R1, R3
    SUB R0, R0, #1
    MOV R3, #0
    CMP R0, R3
    BEQ EXIT
    B LOOP

EXIT
    B EXIT
    END
```

Figure 2: Factorial computation of a number

1. To evaluate the factorial of a number, the method undertaken is to start with the number and 1 in two registers.
2. Then, the number in a register(1 initially) is multiplied by the value in the other register(the number whose factorial is to be computed).
3. The factorial number is then decremented and this process is continued until the value in the factorial identity register is 1. In the code snippet in the screenshot given above, the factorial of 5 is evaluated and the output is stored in the R1 register.
4. Note that
$$5! = 5 * 4 * 3 * 2 * 1 = 120 = 0x78$$
which is the result that we expect to see.

3.3 Compiling the first and last nibbles of 8 bit numbers



```
Register        Value
Current
  R0            0x0000005C
  R1            0x00000000
  R2            0x98761234
  R3            0x00000000
  R4            0x00000000
  R5            0x00000000
  R6            0x00000000
  R7            0x00000000
  R8            0x00000000
  R9            0x00000000
  R10           0x00000000
  R11           0x00000000
  R12           0x00000000
  R13 (SP)      0x00000000
  R14 (LR)      0x00000000
  R15 (PC)      0x00000054
  CPSR          0x600000D3
  SPSR          0x00000000
User/System
Fast Interrupt
Interrupt
Supervisor
Abort
Undefined
Internal
  PC $          0x00000054
  Mode          Supervisor
  States        94
  Sec           0.00002350
```

```
        entry
main
        ldr r0, =list
        mov r1, #4
        mov r2, #0
loop
        ldrb r3, [r0], #1
        and r3, r3, #&F0
        lsl r2, r2, #4
        orr r2, r2, r3
        subs r1, r1, #1
        bne loop
        b main2
main2
        ldr r0, =list
        mov r1, #4
        lsr r2, r2, #4
loop2
        ldrb r3, [r0], #1
        and r3, r3, #&0F
        lsl r2, r2, #4
        orr r2, r2, r3
        subs r1, r1, #1
        bne loop2
        ldr r3, result
        str r2, [r3]
        swi &11
list    dcb &91
        dcb &82
        dcb &73
        dcb &64
result  dcd &00000000
        end
```

Figure 3: Compiling the first and last 4 bits of 8 bit numbers in memory

1. This problem required me to take 4 8 bit numbers, split them into 2 parts of 4 bits each and compile the first nibbles and second nibbles of all the numbers.
2. The method that I have used is to iterate over all the numbers stored in the memory and first extract the first 4 bits of each of those numbers and put them together into the target register.
3. Then the cycle is reset and the same process is repeated for the second 4 bits of each of those numbers.
4. As can be seen in the screenshot, for the input hexadecimal numbers 0x91, 0x82, 0x73, 0x64, the output must be 0x98761234, which is the number stored in R2 on the register panel(top left).

4 Conclusion

From this experiment, one could gain familiarity with the ARM instruction set and write simple programs. Given the wide applicability and prominence of RISC based ARM processors in the market, the understanding of assembly programming is crucial. This experiment provided good insights to the objective and highlighted the key differences between AVR and ARM. It is also noted that RISC instruction sets vary among different devices.