Experiment 4

Assembly Language Instructions

Lab Objective

In this lab, we will learn some basic ARM assembly language instructions and write a simple programs in assembly language.

ARM Assembly Instructions

ARM assembly instructions can be divided in three different sets.

Data processing instructions manipulate the data within the registers. These can be arithmetic (sum, subtraction, multiplication), logical (boolean operations), relational (comparison of two values) or move instructions.

Memory access instructions move data to and from the main memory. Since all other operations only work with immediate constant values (encoded in the instruction itself) or with values from the registers, the load/store instructions are necessary to deal with all but the smallest data sets.

Branch instructions instructions change the control flow, by modifying the value of the Program Counter (R15). They are needed to implement conditional statements, loops, and function calls.

Data processing Instructions

ARM data processing instructions enable the programmer to perform arithmetic and logical operations on data values in registers. All other instructions just move data around and control the sequence of program execution, so the data processing instructions are the only instructions which modify data values. Most data processing instructions can process one of their operands using the barrel shifter (discussed in class).

If you use the 'S' suffix on a data processing instruction, it updates the flags in the CPSR. Move and logical operations update the carry flag C, negative flag N, and zero flag Z. The carry flag (C) is set as a result of the barrel shift when the last bit is shifted out. The negative flag (N) is set to bit 31 of the result. The zero flag (Z) is set if the result is zero.

Move Instruction (MOV)

Inside the processor, the data resides in the registers. MOV is the basic instruction that moves the constant data in the register or move that data from one register to another. In the Thumb instruction set MOVT, instruction moves 16-bit immediate value to top halfword (bits 16 to 31) and the bottom halfword remains unaltered.

Example 4.1

```
        MOV     R0, #0xFF       ; Move 8-bit hex number to R0
        MOV     R1, #-0x281     ; Move 12-bit signed hex number to R1
        MOV     R4, #0xD364     ; Move 16-bit hex number to the lower
                                ; halfword of R4
        MOVT    R4, #0xFFB1     ; Move 16-bit hex number to the upper
                                ; halfword of R4
        MOV     R5, R4          ; Move the contents of R4 to R5
```

Shift and Rotate instructions

Shift and Rotate instructions are used to change the position of bit values in a register. Different variants of these instructions are discussed in this manual.

Logical Shift Right (LSR)

This instruction is similar to the unsigned divide by 2^n where n specifies the number of shifts. Execute the following instructions in your compiler and observe the results.

Example 4.2

```
        MOV     R0, #0x06       ; Move 0x06 to R0
        MOV     R3, #0x200      ; Move 0x200 to R5
        MOV     R5, #0x9B1D     ; Move 0x9B1D to R5

STOP
        LSR     R4, R3, #6      ; Logical shift right by 6 bits
        LSR     R2, R3, R0      ; Logical shift right by the value
                                ; in lower byte of R0
        LSR     R6, R5, #5      ; Logical shift right by 5 bits without
                                ; flags update
        LSRS    R7, R5, #5      ; Logical shift right by 5 bits with
                                ; flags update
```

```
        B       STOP
        END                     ; End of the program
```

Arithmetic Shift Right (ASR)

This instruction is similar to the signed divide by 2^n where n specifies the number of shifts. Execute the following instructions in your compiler and observe the result.

Example 4.3

```
        MOV     R0, #0x07       ; Move 0x07 to lower byte of R0
        MOV     R3, #-0x669     ; Move the 12-bit signed hex number to R3
        MOV     R5, #0xD634     ; Move the 16-bit signed hex number to R5
        MOVT    R5, #0xFFB1     ; Move the 16-bit signed hex number to
                                ; upper halfword of R5
STOP
        ASR     R4, R3, #5      ; Arithmetic shift right by 5 bits
        ASR     R2, R3, R0      ; Arithmetic shift right by the lower
                                ; byte value of R0
        ASR     R6, R5, #5      ; Arithmetic shift right by 5 bits
                                ; without flags update
        ASRS    R7, R5, #5      ; Arithmetic shift right by 5 bits
                                ; with flags update
        B       STOP
        END                   ; End of the program
```

Logical Shift Left (LSL)

Logical shift left instruction works fine for both signed and unsigned numbers. This instruction is synonymous to multiply by 2^n where n specifies the number of shifts.

Example 4.4

```
        MOV     R7, #0xC        ; Move 0x0C to lower byte of R7
        MOV     R0, #-0x281     ; Move the 12-bit signed hex number to R0
        MOV     R2, #0x25C3     ; Move the 16-bit signed hex value
                                ; to the lower halfword of R2
        MOV     R4, #0x593C     ; Move the 16-bit hex value
                                ; to the lower halfword of R4
        MOVT    R4, #0xA377     ; Move the 16-bit hex value
                                ; to the upper halfword of R4
```

```
STOP
        LSL     R1, R0, #3      ; Logical shift left by 3 bits
        LSL     R3, R2, R7      ; Logical shift left by the value
                                ; in the lower byte of R7
        LSL     R5, R4, #10     ; Logical shift left by 10 bits
                                ; without flags update
        LSLS    R6, R4, #10     ; Logical shift left by 10 bits
                                ; with flags update
        B       STOP

        END                 ; End of the program
```

Rotate Right (ROR)

Bit-rotates do not discard any bits from the register. Instead, the bit values are removed from one end of the register and inserted into the other end.

Example 4.5

```
        MOV     R0, #0x9        ; Move 0x09 to lower byte of R0
        MOV     R1, #0x25C3     ; Move the 16-bit signed hex value
                                ; to the lower halfword of R2
        MOV     R3, #0x73A2     ; Move the 16-bit hex value
                                ; to the lower halword of R3
        MOVT    R3, #0x5D81     ; Move the 16-bit hex value
                                ; to the upper halfword of R3

STOP
        ROR     R2, R1, #24     ; Rotate right left by 24 bits
        ROR     R7, R3, R0      ; Rotate right by the value
                                ; in the lower byte of R0
        ROR     R4, R3, #10     ; Rotate right by 10 bits
                                ; without flags update
        RORS    R5, R3, #10     ; Rotate right by 10 bits
                                ; with flags update
        B       STOP
        END                 ; End of the program
```

Rotate Right Extended (RRX)

RRX is a ROR operation with a crucial difference. It rotates the number to the right by one place but the original bit 31 is filled by the value of the Carry flag and the original bit 0 is moved into the Carry flag. This allows a 33-bit rotation by using both the register and the

carry flag.

Example 4.6

```
        MOV    R3, #0xD129     ; Move the 16-bit hex value
                               ; to the lower halfword of R3
        MOVT   R3, #0xF29A     ; Move the 16-bit hex value
                               ; to the upper halfword of R3

STOP
        RRX    R4, R3          ; Rotate right extended without flags update
        RRXS   R6, R3          ; Rotate right extended with flags update

        B      STOP
        END                ; End of the program
```

Memory Access Instructions

ARM is a load/store architecture. It does not support memory to memory data processing operations. All data processing operations are executed in the registers .i.e., data values needed for an operation must be moved into registers before using them. We need instructions that interact with memory to move data from memory to registers and vice versa. ARM has three sets of instructions which interact with main memory.

1. Single register data transfer (LDR/STR)
2. Block data transfer (LDM/STM)
3. Single Data Swap (SWP)

In this lab manual, we will discuss single register data transfer instructions and different addressing modes that can be used to move the contents from memory to registers and vice versa.

Single Register Data Transfer

The basic data transfer instructions supported by ARM assembler can load and store different size of data from and to the memory respectively.

Example 4.7

```
            THUMB           ; Marks the THUMB mode of operation
```

```
;****** Data Variables are declared in DATA AREA ******;

        AREA    MyData, DATA, READWRITE

X       SPACE    2      ; 2 bytes for variable X are reserved
                        ; in memory at 0x20000000
data    SPACE    2      ; 2 bytes for variable data is declared in memory

;***** The user code (program) is placed in CODE AREA *****;

        AREA    |.text|, CODE, READONLY, ALIGN=2

        ENTRY           ; ENTRY marks the starting point
                        ; of the code execution
        EXPORT __main

__main
; ***** User code starts from the next line *****;

      MOV R0, #0x2D
      MOV R1, #0x40

      LDR R2, =data     ; Load the address of data variable in R2
      ADD R0, R0, R1
      STR R0, [R2]      ; Store the contents of R0 at address loaded in R2
                        ; Observe memory window at address specified in R2
STOP
      B   STOP          ; Infinite loop to STOP
      ALIGN
      END               ; End of the program, matched with ENTRY keyword
```

LDR instruction can also be used to load any constant in the registers. Any 32-bit numeric constant can be constructed in a single instruction. It is used to generate constants that are out of range of the MOV and MVN instructions.

Example 4.8

```
        THUMB           ; Marks the THUMB mode of operation

;****** Data Variables are declared in DATA AREA ******

        AREA    MyData, DATA, READWRITE

Result  SPACE 4         ; 4 bytes reserved for variable Result
```

```
;***** The user code (program) is placed in CODE AREA *****;

        AREA    |.text|, CODE, READONLY, ALIGN=2

        ENTRY           ; ENTRY marks the starting point of
                        ; the code execution
        EXPORT __main

__main
; ***** User code starts from the next line *****;

        LDR  R1, =0x458C3   ; Loads the value 0x458C3 in R1
                            ; which is out of range of MOV instruction
        LDR  R2, =0xA8261
        LDR  R4, =Result    ; Load the address of variable
                            ; result in R4
        MOV  R0, #0
        MUL  R3, R1, R2
        LSRS R3, #9         ; Logical shift right with flags update
        STR  R3, [R4]       ; Store the contents of R3 to memory
                            ; address already loaded in R4
STOP
        B   STOP
        END
```

These instructions provide the most flexible way to transfer single data items between an ARM register and memory. The data item may be a byte, a 16-bit half-word, a 32-bit word or a 64-bit double word. An optional modifier should be added to access the appropriate data type. Following table shows the data types available and their ranges.

type	Data Type	Range
	32-bit word	0 to $2^{32}-1$ or -2^{31} to $2^{31}-1$
B	unsigned Byte	0 to $2^{8}-1$
SB	Signed Byte	-2^{7} to $2^{7}-1$
H	unsigned Halfword	0 to $2^{16}-1$
SH	Signed Halfword	-2^{15} to $2^{15}-1$
D	Double word	0 to $2^{64}-1$ (two registers used)

In the following example Byte data type is used

Example 4.9

```
        THUMB           ; Marks the THUMB mode of operation

;****** Data Variables are declared in DATA AREA ******;
```

```
        AREA    MyData, DATA, READWRITE

High    DCB     0       ; 1 byte reserved for variable High
Low     DCB     0       ; 1 byte reserved for variable Low
Result  DCB     0       ; 1 byte reserved for variable Result
Result1 DCD     0       ; 4 bytes reserved for variable Result1

;***** The user code (program) is placed in CODE AREA *****;

        AREA    |.text|, CODE, READONLY, ALIGN=2

        ENTRY           ; ENTRY marks the starting point of
                        ; the code execution
        EXPORT __main

__main
; ***** User code starts from the next line *****;

      LDR  R0, =High      ; R0 =
      LDR  R1, =Low       ; R1 =
      LDR  R2, =Result    ; R2 =
      LDR  R5, =Result1   ; R5 =

      MOV  R3, #0x5
      MOV  R4, #0x8

      LSL  R3, #4         ; Shift the lower nibble to upper nibble of
                          ; the least significant byte

      STRB R3, [R0]       ; Observe memory window at the address
                          ; stored in R0

      STRB R4, [R1]       ; Observe memory window at the address
                          ; stored in R1

      ORR  R3, R3, R4     ; Set lower four bits of R3 equal to
                          ; lower four bits of R4

      STRB R3, [R2]       ; Observe memory window at the address
                          ; stored in R2

      STR  R3, [R5]       ; Takes 32 bits to store the result

      ALIGN
STOP
        B    STOP
      END
```

Addressing Modes

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: preindex with writeback, preindex, and postindex.

Example 4.10

```
        THUMB

;****** Data Variables are declared in DATA AREA ******;

        AREA    MyData, DATA, READWRITE

data    SPACE   4       ; 4 bytes reserved for variable data

        AREA    |.text|, CODE, READONLY, ALIGN=2

        ENTRY           ; ENTRY marks the starting point of
                        ; the code execution
        EXPORT __main

__main
; ***** User code starts from the next line *****;

        MOV R0, #0x4A8
        MOV R1, #0x761
        MOV R4, #0x8D
        MOV R3, #0x0C

        SUB R1, R1, R0

; ********** Offset Addressing **********

        LDR R2, =data       ; Load the address of data in R2

        STR R1, [R2]        ; Store the 32-bit contents of R1 register
                            ; at the address loaded in R2. The contents
                            ; of R2  does not change.

        STR R1, [R2, #4]    ; A word in register R1 is stored at
                            ; memory address calculated by adding the
                            ; offset to R2. Contents of R2 does not change.

        STR R1, [R2, R3]    ; Store the contents of R1 at memory address
                            ; calculated by adding value in the base
                            ; register R2 to the value in the register R3.
                            ; Both R2 and R3 remain unchanged.
```

```
        STR R1, [R2, R3, LSL #2]
        ; Store the 32-bit value in R1 to the memory address calculated by
        ; adding the address in base register R2 to the value obtained
        ; by shifting the contents of R3 towards left by 2 bits. R2
        ; and R3 remain unchanged.

; ********** Post-index Addressing **********

        LDR R2, =data       ; Load the address of data in R2

        STR R1, [R2], #12
        ; Contents of R1 will be stored in memory at the address loaded in
        ; R2. R1 is updated with new address after adding the constant,
        ; specified, to it.

        STR R4, [R2], #24

; ********** Pre-index Addressing **********

        LDR R2, =data       ; Load the address of data in R2

        STR R1, [R2, #32]!
        ; 32-bit word in R1 will be stored in memory at an address
        ; calculated by adding 32 to the address loaded in R2.
        ; The contents of R2 also get updated with this new address

STOP
        B   STOP
        END
```

Exercises

1. R1 = 0x37AF. Apply LSR instruction on this value and store the result in R2. Consider shift amounts as 1,3,12.
2. Assume R3 = 0x395A62. Apply ASR instruction and store the result in R4. Consider shift amounts as 1,5,15.
3. R3 register contains a negative value -321. Apply the LSR and ASR instruction and observe the result. Which instruction gives the correct result and why. Write your result in both the cases and explain.
4. Suppose R0 register contains an unsigned hex value 0xA642. Multiply this hex value by 8 and write the result in R2.
5. Register R1 has signed value -458 and register R2 has unsingned value 458. Apply the LSL operation on both the registers and observe the result. (Hint: Hexadecimal representation of 458 is 0x1CA)

6. Compare the results of question 5 and explain why ASL is not needed.
7. What is the net result if a bit pattern (32 bits) is logical left shifted 2 positions and then logical right shifted 2 positions?
8. Assume that R0 register contain 0xC5AF2. Using shift operation move the contents of R0 to R4.
9. Now consider the value of R0 in previous as a signed value(use sign extension) and multiply it by 8 and observe the result
10. Move hexadecimal value 0xA964 to the register R1 and observe the result of each of LSR, ASR, LSL, ROR and RRX instructions. Assume the shift amount equal to 1. State your findings.
11. Move the hexadecimal value 0xD29D1C8B to register R5 and apply the operations LSR, ASR, LSL, ROR and RRX with flags update. Write your results.
12. Rotate the contents of R3 register to left by 12 bits. Assume R3 = 0x568A1FC3.
13. Consider R5 = 0x65FF. Move this value to the higher 16 bits using only one instruction.
14. Consider R6 = 0x76543210. Swap the top and bottom halves of R6.
15. Perform a RRX operation on R6 register and observe the result.