2 ARM Cortex-M4 Instructions and Addressing Modes

Learning Objectives

- Recognize the three main categories of ARM Cortex-M4 instructions: data processing, load/store, and branch.
- Perform arithmetic, logical, shift/rotate, and compare/test operations on registers.
- Access memory using load and store instructions with different addressing modes (immediate, register offset, pre-/post-indexed).
- Apply conditional execution using condition codes in the `xPSR`.
- Observe and interpret the effects of instructions on registers, memory, and status flags using the **Keil uVision5** debugger.

Experiment Overview

This experiment introduces the core instruction set of the ARM Cortex-M4 processor. You will learn how to manipulate data in registers using arithmetic and logical instructions, how to transfer data between memory and registers using load/store instructions, and how to control program execution using condition codes and branches.

In the process, you will:

- Step through assembly instructions in the debugger and watch how registers and memory change.
- Observe how flags (Z, N, C, V) are updated and used for conditional execution.
- Practice pointer-style memory access through different addressing modes.

By the end of this lab, you will have gained practical experience in writing, executing, and debugging ARM Cortex-M4 assembly programs. These foundational skills in instruction-level programming, register manipulation, and memory access will serve as essential building blocks for subsequent experiments on flow control, interrupt handling, and peripheral interfacing.

Contents

1 Theoretical Background

As mentioned in Experiment 1, assembly instruction are split into three main categories: data processing, load/store, and branch instructions. This experiment focuses on data processing instructions, load/store instructions and their addressing modes. While branch instructions would be covered in more detail in Experiment 3 (Flow Control), a brief overview is provided here for completeness.

1.1 Data Processing Instructions

Data processing instructions perform arithmetic and logical operations on data stored in registers. They can also manipulate the condition flags in the xPSR based on the results of the operations. Common data processing instructions take the following form:

```
OPCODE{<cond>}{S} Rd, Rn, Operand2
```

where:

- `OPCODE`: the operation to be performed (e.g., `ADD`, `SUB`, `AND`, `ORR`).
- `<cond>`: optional condition code that predicates execution.
- `S`: optional suffix indicating whether to update the condition flags.
- `Rd`: destination register where the result is stored.
- `Rn`: first operand register.
- `Operand2`: second operand, which can be an immediate value limited to 8 bits, a register, or a barrel shifter operation.

1.1.1 Arithmetic Instructions

Arithmetic instructions perform basic mathematical operations. Some common arithmetic instructions include addition, subtraction, multiplication, and their variants. The following table summarizes some of the most commonly used arithmetic instructions in the ARM Cortex-M4 architecture.

Table 2.1: Common ARM Cortex-M4 Arithmetic Instructions

Instr.	**Syntax**	**Operation**	**Description**
ADD	`ADD{S} Rd, Rn, Operand2`	$Rd \leftarrow Rn + Operand2$	*Operand2* may be a register, an immediate, or a shifted register.
ADC	`ADC{S} Rd, Rn, Operand2`	$Rd \leftarrow Rn + Operand2 + C$	Adds carry-in C.
SUB	`SUB{S} Rd, Rn, Operand2`	$Rd \leftarrow Rn - Operand2$	Standard subtraction.
SBC	`SBC{S} Rd, Rn, Operand2`	$Rd \leftarrow Rn - Operand2 - (1 - C)$	Subtract with borrow (borrow encoded via carry flag C).
RSB	`RSB{S} Rd, Rn, Operand2`	$Rd \leftarrow Operand2 - Rn$	Reverse subtract.
MUL	`MUL{S} Rd, Rn, Rm`	$Rd \leftarrow (Rn \times Rm)_{[31:0]}$	32×32 → low 32 bits.
MLA	`MLA Rd, Rn, Rm, Ra`	$Rd \leftarrow (Rn \times Rm) + Ra$	Multiply-accumulate.
MLS	`MLS Rd, Rn, Rm, Ra`	$Rd \leftarrow Ra - (Rn \times Rm)$	Multiply-subtract.
UMULL	`UMULL RdLo, RdHi, Rn, Rm`	$\{RdHi, RdLo\} \leftarrow Rn \times Rm$	Unsigned 32×32 → 64-bit product.
SMULL	`SMULL RdLo, RdHi, Rn, Rm`	$\{RdHi, RdLo\} \leftarrow Rn \times Rm$	Signed 32×32 → 64-bit product.

Note: C denotes the carry flag in `xPSR`. *Operand2* may be an immediate or a shifted register depending on the encoding.

1.1.2 Logical and Move Instructions

Logical instructions perform bitwise operations on data, while move instructions transfer data between registers or load immediate values. The following table summarizes some of the most commonly used logical and move instructions in the ARM Cortex-M4 architecture.

Table 2.2: Logical and Move Instructions

Instr.	Syntax	Operation	Description
AND	`AND Rd, Rn, Operand2`	$Rd \leftarrow Rn \,\&\, Operand2$	Bitwise AND.
ORR	`ORR Rd, Rn, Operand2`	$Rd \leftarrow Rn \mid Operand2$	Bitwise OR.
EOR	`EOR Rd, Rn, Operand2`	$Rd \leftarrow Rn \oplus Operand2$	Bitwise XOR.
BIC	`BIC Rd, Rn, Operand2`	$Rd \leftarrow Rn \,\&\, \neg Operand2$	Bit clear.
MVN	`MVN Rd, Operand2`	$Rd \leftarrow \neg Operand2$	Bitwise NOT of operand.
MOV	`MOV Rd, Operand2`	$Rd \leftarrow Operand2$	Register or immediate move.
MOVW	`MOVW Rd, #imm16`	$Rd[15{:}0] \leftarrow imm16$	Write low halfword.
MOVT	`MOVT Rd, #imm16`	$Rd[31{:}16] \leftarrow imm16$	Write high halfword (low preserved).

Note: C denotes the carry flag in `xPSR`. *Operand2* may be an immediate or a shifted register depending on the encoding.

We will be using bitwise logical instructions extensively in this experiment to manipulate specific bits in registers from setting, clearing, and flipping certain bits, or checking if certain bits are set or cleared.

Setting and Clearing Bits

To set, clear, or toggle specific bits in a register, you can use the following logical instructions:

- `ORR Rd, Rn, #mask`: Sets bits in `Rd` where the corresponding bits in `mask` are 1.
- `BIC Rd, Rn, #mask`: Clears bits in `Rd` where the corresponding bits in `mask` are 1.
- `EOR Rd, Rn, #mask`: Toggles bits in `Rd` where the corresponding bits in `mask` are 1.

to check if certain bits are set or cleared, you can use the `TST` instruction:

- `TST Rn, #mask`: Performs a bitwise AND between `Rn` and `mask`, updating the condition flags based on the result. If the result is zero, the zero flag (Z) is set, indicating that none of the bits in `mask` are set in `Rn`.

1.1.3 Shift and Rotate Instructions

Table 2.3: Shift and Rotate Instructions

Instr.	Syntax	Operation	Description
LSL	`LSL Rd, Rm, #sh|Rs`	$Rd \leftarrow Rm \ll sh$	Logical left shift by immediate or by register.
LSR	`LSR Rd, Rm, #sh|Rs`	$Rd \leftarrow Rm \gg sh$	Logical right shift (zero fill).
ASR	`ASR Rd, Rm, #sh|Rs`	$Rd \leftarrow Rm \gg sh$	Arithmetic right shift (sign fill).
ROR	`ROR Rd, Rm, #sh|Rs`	$Rd \leftarrow \mathrm{ROR}(Rm, sh)$	Rotate right by immediate or by register.
RRX	`RRX Rd, Rm`	$Rd \leftarrow \mathrm{ROR}_C(Rm, 1)$	Rotate right 1 bit through carry (uses C as incoming bit 31, outgoing bit $0 \rightarrow C$).

Note: Shift amount can be an immediate `#sh` (0–31) or a register `Rs` (low 8 bits used). For immediates: `LSL #0` = no shift; `LSR #0` is treated as shift by 32; `ASR #0` is treated as shift by 32; `ROR #0` means `RRX`.

Not all shift/rotate instructions are explicitly present in the ARMv7-M ISA. For example, there is no ROL (rotate left) or ASL (arithmetic shift left) instruction, as these operations can be achieved using existing shift instructions: ROL can be implemented using ROR with a complementary shift amount, and ASL is equivalent to LSL.

1.1.4 Compare and Test Instructions

Compare and test instructions are used to compare values and set condition flags without producing a direct result in a register. These instructions are useful for conditional execution and branching based on the results of comparisons. The following table summarizes the compare and test instructions in the ARM Cortex-M4 architecture.

Table 2.4: Compare and Test Instructions

Instr.	**Syntax**	**Operation**	**Description**
CMP	`CMP Rn, Operand2`	Flags from $(Rn - Operand2)$	Comparison; no register result.
CMN	`CMN Rn, Operand2`	Flags from $(Rn + Operand2)$	"Compare negative" (add then set flags).
TST	`TST Rn, Operand2`	Flags from $(Rn \,\&\, Operand2)$	Bitwise test; no register result.
TEQ	`TEQ Rn, Operand2`	Flags from $(Rn \oplus Operand2)$	XOR test; no register result.

1.2 Load and Store Instructions

Since the ARM Cortex-M4 architecture follows the RISC design philosophy, it uses a load/store architecture. This means that data processing instructions can only operate on data in registers, and any data in memory must first be loaded into a register before it can be processed. Similarly, results from data processing operations must be stored back to memory if they need to be preserved. The following table summarizes the load and store instructions in the ARM Cortex-M4 architecture.

Table 2.5: Load and Store Instructions (Summary)

Instr.	**Syntax Example**	**Description**
LDR / STR	`LDR/STR Rt, [Rn, #off]`	Load/store a 32-bit word.
LDRB / STRB	`LDRB/STRB Rt, [Rn, #off]`	Load/store an 8-bit byte.
LDRH / STRH	`LDRH/STRH Rt, [Rn, #off]`	Load/store a 16-bit halfword.
LDRSB / LDRSH	`LDRSB/LDRSH Rt, [Rn, #off]`	Load signed byte/halfword and sign-extend to 32 bits.
LDRD / STRD	`LDRD/STRD Rt, Rt2, [Rn, #off]`	Load/store a 64-bit doubleword (two registers).

- `LDR Rt, =label` is a *pseudo-instruction*. The assembler replaces it with code to load the **address** of `label` into `Rt`.
- `LDR Rt, label` (without `=`) loads the **contents stored at** `label`.

```
        AREA    MYDATA, DATA, READONLY
XVAL    DCD     0x12345678          ; word in memory
YPTR    DCD     YVAL                ; contains the address of YVAL

        AREA    MYDATA2, DATA, READWRITE
YVAL    DCD     0

        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler

Reset_Handler
        LDR     R0, =XVAL           ; R0 = &XVAL (address of XVAL)
        LDR     R1, [R0]            ; R1 = 0x12345678 (load word from memory)

        LDRB    R2, [R0]            ; R2 = 0x78 (lowest byte of XVAL)
        LDRH    R3, [R0]            ; R3 = 0x5678 (lowest halfword of XVAL)

        MOV     R4, #0xFF
        LDR     R0, YPTR            ; R0 = contents of YPTR = &YVAL
        STRB    R4, [R0]            ; store 0xFF into YVAL (low byte only)

STOP    B       STOP
        END
```

Listing 2.1: Examples of Load and Store Instructions

Note:

- The line `YPTR DCD YVAL` allocates a 32-bit word at the label `YPTR` and initializes it with the address of `YVAL`.
- Executing `LDR R0, YPTR` loads the contents of `YPTR` (the address of `YVAL`) into `R0`, making `R0` a pointer to `YVAL`.
- By contrast, `LDR R0, =YVAL` directly loads the address of `YVAL` into `R0` without going through memory.

Address	Label	Contents
0x2000	XVAL	0x12345678
0x2004	YPTR	0x2008 (address of YVAL)
0x2008	YVAL	0x00000000

1.3 Branch Instructions and Condition Codes

Branch instructions change the flow of execution by jumping to a different part of the program. They can be unconditional or conditional based on the status flags in the xPSR. The following table summarizes the branch instructions in the ARM Cortex-M4 architecture.

Table 2.6: Branch Instructions

Instr.	Syntax Example	Description
B	`B label`	Unconditional branch to `label`.
B<cond>	`B<cond> label`	Conditional branch based on condition flags.
BL	`BL label`	Branch with link (calls subroutine, saves return address in LR).
BX	`BX Rm`	Branch to address in register `Rm` (LR is usually used here).

Table 2.7: Condition Codes

Cond.	Meaning	Description
EQ	Equal	Z set (zero result)
NE	Not equal	Z clear (non-zero result)
CS/HS	Carry set / Unsigned higher or same	C set
CC/LO	Carry clear / Unsigned lower	C clear
MI	Minus / Negative	N set (negative result)
PL	Plus / Positive or zero	N clear
VS	Overflow set	V set
VC	Overflow clear	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	Z clear and N == V
LE	Signed less than or equal	Z set or N != V
AL	Always	(unconditional)
NV	Never	(not used)

1.4 Addressing Modes

Addressing modes define how the effective address or operand value is obtained by an instruction. The ARM Cortex-M4 supports several common addressing modes, summarized below:

Table 2.8: General Addressing Modes in ARM Cortex-M4

Mode	Syntax Example	Description
Immediate	`MOV R0, #10`	Operand is a constant value encoded in the instruction.
Register Direct	`MOV R0, R1`	Operand is taken directly from a register.
Register Indirect	`LDR R0, [R1]`	Register holds the address of the operand in memory.
Register Offset	`LDR R0, [R1, R2]`	Effective address = base register + offset register.
Immediate Offset	`LDR R0, [R1, #4]`	Effective address = base register + constant offset.
Pre-indexed	`LDR R0, [R1, #4]!`	Base updated first, then memory access.
Post-indexed	`LDR R0, [R1], #4`	Memory access first, then base register updated.

```
; Immediate Offset
    LDR     R0, [R1, #4]      ; R0 = word at memory[R1 + 4]
; Register Offset
    LDR     R0, [R1, R2]     ; R0 = word at memory[R1 + R2]
; Pre-indexed
    LDR     R0, [R1, #4]!     ; R1 = R1 + 4, then load R0 = [R1]
; Post-indexed
    LDR     R0, [R1], #4      ; load R0 = [R1], then R1 = R1 + 4
```

Listing 2.2: Examples of Offset, Pre-indexed, and Post-indexed Addressing Modes

2 Procedure

2.1 Examples

2.1.1 Example 1 — Arithmetic and Bitwise Operations

This example demonstrates basic arithmetic and bitwise operations in ARM assembly, showing how to set, clear, and flip bits.

```
    AREA  RESET, CODE, READONLY
    EXPORT __Vectors
__Vectors
    DCD 0x20001000
    DCD Reset_Handler
    ALIGN

; Data section
NUM1    DCD 50              ; First integer
NUM2    DCD 12              ; Second integer
RP      DCD RESULT          ; Pointer to RESULT variable

    AREA MYDATA, DATA, READWRITE
RESULT  DCD 0               ; Will hold the final computed value

    AREA MYCODE, CODE, READONLY
    ENTRY
    EXPORT Reset_Handler

Reset_Handler
    ; Load values from memory into registers
    LDR R1, NUM1            ; R1 = 50
    LDR R2, NUM2            ; R2 = 12

    ; Perform arithmetic
    ADD R3, R1, R2          ; R3 = 50 + 12 = 62
    SUB R3, R3, #4          ; R3 = 62 - 4 = 58
    MUL R4, R3, R2          ; R4 = 58 * 12 = 696

    ; Logical operations
    AND R5, R4, #0xFF       ; R5 = 696 & 0xFF = 0xB8 (184)
    ORR R5, R5, #0x01       ; R5 = 0xB8 | 0x01 = 0xB9 (185)
    BIC R5, R5, #0x08       ; R5 = 0xB9 & ~0x08 = 0xB1 (177)
    EOR R5, R5, #0x02       ; R5 = 0xB1 ^ 0x02 = 0xB3 (179)

    ; Store result in memory using a pointer
    LDR R6, RP              ; R6 = address of RESULT
    STR R5, [R6]            ; RESULT = R5

    ; Read back for verification
    LDR R7, [R6]            ; R7 = RESULT
STOP    B STOP

    END
```

Listing 2.3: Arithmetic and bitwise operations example

2.1.2 Example 2 — Status Flags and Logical Tests

This example demonstrates the use of status flags and logical tests in ARM assembly, including conditional execution based on comparison results.

```
        AREA  RESET, CODE, READONLY
        EXPORT __Vectors
    __Vectors
        DCD 0x20001000
        DCD Reset_Handler
        ALIGN

        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler

    Reset_Handler
        ; Set up registers
        MOVS R0, #10          ; R0 = 10, updates flags (N=0, Z=0)
        MOVS R1, #10          ; R1 = 10, updates flags

        ; Compare R0 and R1 using SUBS (R0 - R1)
        SUBS R2, R0, R1       ; R2 = 10 - 10 = 0
        ; Flags after SUBS:
        ; Z=1 (result zero), N=0, C=1 (no borrow), V=0

        ; Compare with immediate using TST (bitwise AND, updates flags)
        MOV R3, #0x0F         ; R3 = 0x0F (binary 00001111)
        TST R3, #0x08         ; Test bit 3
        ; Flags:
        ; Z=0 (bit 3 is set), N=0

        TST R3, #0x10         ; Test bit 4
        ; Flags:
        ; Z=1 (bit 4 not set), N=0

        ; Test equivalence using TEQ (bitwise XOR, updates flags)
        MOV R4, #0x55         ; 0x55 = 01010101b
        MOV R5, #0x55         ; same value
        TEQ R4, R5            ; R4 XOR R5 = 0
        ; Flags:
        ; Z=1 (equal), N=0

        MOV R6, #0x33         ; 0x33 = 00110011b
        TEQ R4, R6            ; 0x55 XOR 0x33 != 0
        ; Flags:
        ; Z=0, N=0

        ; Negative result example with ADDS
        MOVS R7, #5           ; R7 = 5
        SUBS R7, R7, #10      ; R7 = 5 - 10 = -5 (two's complement)
        ; Flags:
        ; N=1 (negative), Z=0, C=0, V=0
    STOP    B STOP
        END
```

Listing 2.4: Status flags and logical tests example

2.1.3 Example 3 — Load/Store with Different Addressing Modes

This example demonstrates load and store instructions using various addressing modes.

```
        AREA    RESET, CODE, READONLY
        EXPORT  __Vectors
__Vectors
        DCD     0x20001000          ; Initial SP (example)
        DCD     Reset_Handler       ; Reset vector
        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler
Reset_Handler
        ; Base address of the array
        LDR     R0, =ARRAY          ; R0 = &ARRAY
        ; -------------------------------------------------------
        ; 1) Immediate Offset: EA = R0 + #8  (third element)
        ; -------------------------------------------------------
        LDR     R1, [R0, #8]        ; R1 = ARRAY[2] = ? (expect 30)
        LDR     R7, =OUT
        STR     R1, [R7, #0]        ; OUT[0] = R1
        ; -------------------------------------------------------
        ; 2) Pre-indexed: R2 = [R2 + #4], then R2 = R2 + #4
        ;    Use a scratch pointer so R0 remains the base.
        ; -------------------------------------------------------
        MOV     R2, R0              ; R2 = &ARRAY
        LDR     R3, [R2, #4]!       ; R2 -> &ARRAY[1], R3 = ARRAY[1] = ? (expect
    20)
        STR     R3, [R7, #4]        ; OUT[1] = R3
        ; After this, R2 now points at ARRAY[1].
        ; -------------------------------------------------------
        ; 3) Post-indexed: R4 = [R4], then R4 = R4 + #12
        ;    Load first element, then advance pointer to the 4th.
        ; -------------------------------------------------------
        MOV     R4, R0              ; R4 = &ARRAY
        LDR     R5, [R4], #12       ; R5 = ARRAY[0] = ? (expect 10), R4 ->
    &ARRAY[3]
        STR     R5, [R7, #8]        ; OUT[2] = R5
        ; -------------------------------------------------------
        ; 4) Register Offset: EA = R0 + R6
        ;    Offset register holds byte offset (multiple of 4 for words).
        ; -------------------------------------------------------
        MOV     R6, #12             ; byte offset to ARRAY[3]
        LDR     R8, [R0, R6]        ; R8 = ARRAY[3] = ? (expect 40)
        STR     R8, [R7, #12]       ; OUT[3] = R8
        ; read second element via register offset
        MOV     R6, #4              ; byte offset to ARRAY[1]
        LDR     R9, [R0, R6]        ; R9 = ARRAY[1] = ? (expect 20)
STOP    B       STOP
        AREA    CONSTS, DATA, READONLY
ARRAY   DCD     10, 20, 30, 40      ; four words at consecutive addresses
        AREA    MYDATA, DATA, READWRITE
OUT     DCD     0, 0, 0, 0          ; capture buffer for observed loads. Could by
    replaced by space 16
        END
```

Listing 2.5: Load/store with different addressing modes example

2.2 Tasks

2.2.1 Task 1 — Bitwise Register Manipulation

Start with `R0 = 0x12345678`. Perform the following operations and observe the results in the debugger:

- Clear bits 4–7 (second hex nibble).
- Set bits 8–11 (force nibble to `F`).
- Toggle bits 28–31 (highest nibble).

Hint: Use `BIC`, `ORR`, and `EOR` with appropriate masks.

2.2.2 Task 2 — Arithmetic and Status Flags

Assume initial values `R0 = 25` and `R1 = 10`.

- Add `R0 + R1`, store the result in `R2`.
- Subtract `R1` from `R0`, store the result in `R3`.
- Multiply `R0` and `R1`, store the result in `R4`.
- Compare `R0` and `R1` using `CMP`. If greater, set `R5 = 1`; otherwise, set `R5 = 0`.

Observe how the `Z` (zero) and `N` (negative) flags in the xPSR are affected.

2.2.3 Task 3 — Addressing Modes with an Array

Given the following array:

```
ARRAY   DCD  0x11, 0x22, 0x33, 0x44
```

Perform the following loads using different addressing modes:

- Load the first element (`0x11`) using *immediate offset*.
- Load the second element (`0x22`) using *pre-indexed* addressing.
- Load the third element (`0x33`) using *post-indexed* addressing.
- Load the fourth element (`0x44`) using a *register offset* (offset in another register).

Store each result into an output buffer `OUT` and verify the results in memory.