BIRZEIT UNIVERSITY

Department of Electrical & Computer Engineering
ENCS4110 – Computer Design Laboratory

Experiment 02

ARM Cortex-M4 Instructions and Addressing Modes

Date: October 2025

2 ARM Cortex-M4 Instructions and Addressing Modes

Learning Objectives

After completing this experiment, you will be able to:

- Perform arithmetic, logical, and shift/rotate operations using data-processing instructions (including `Operand2` with the barrel shifter).
- Move data between registers and memory using load/store instructions with immediate, register-offset, and pre-/post-indexed addressing modes.
- Declare and initialize data objects (arrays, strings, buffers) with assembler directives, and use pointers to access and modify them.
- Trace how instructions affect registers and xPSR flags using the **Keil uVision5** debugger (breakpoints, single-step, register/memory views).

Experiment Overview

This experiment develops fluency with the ARM Cortex-M4 instruction set for data manipulation and memory access. You will practice using arithmetic, logical, and shift/rotate instructions, and learn how data moves between registers and memory through load/store instructions and their addressing modes. You will also define data with assembler directives and use pointers to read and update memory.

In this experiment, you will:

- Write and run short assembly routines that use data-processing instructions to transform register values.
- Apply immediate, register-offset, and pre-/post-indexed addressing modes to load from and store to memory.
- Define arrays, strings, and buffers with directives, and use pointers to traverse and modify them.
- Observe instruction effects on registers and flags with the **Keil uVision5** debugger to verify correctness.

By the end of this lab, you will be able to implement, assemble, and debug Cortex-M4 programs that perform register-level computation and structured memory access—providing the foundation for flow control and procedure calls in later experiments.

Contents

1 Theoretical Background 4
1.1 Data Processing Instructions 4
1.1.1 Arithmetic Instructions 4
1.1.2 Logical and Move Instructions 5
1.1.3 Shift and Rotate Instructions 6
1.1.4 Barrel Shifter 6
1.2 Load and Store Instructions 6
1.2.1 Declaring Data in Memory 7
1.2.2 Understanding Pointer Declarations 8
1.2.3 Loading Addresses and Values: `LDR`, `LDR =`, and `ADR` 8
1.3 Addressing Modes 8
2 Procedure 10
2.1 Examples 10
2.1.1 Example 1 — Data Processing Instructions 10
2.1.2 Example 2 — Load/Store with Different Addressing Modes 11
2.2 Tasks 12
2.2.1 Task 1 — Bitwise Register Manipulation 12
2.2.2 Task 2 — Addressing Modes with an Array 12

1 Theoretical Background

As mentioned in Experiment 1, assembly instructions 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. branch instructions and flow control will be covered in the next experiment.

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:

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

where:

- `LABEL`: optional label for branching.
- `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 (see Section 1.1.4).

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`	$Rd \leftarrow Rn - Operand2 - \overline{C}$	Subtract with carry. If carry flag is clear, result is reduced by one. Used for multiword arithmetic.
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`	$Rd \leftarrow Rn \,\vert\, Operand2$	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.

In this experiment, you will work with bitwise logical instructions to manipulate individual bits within registers. Such operations are fundamental in microcontroller programming, where control and status registers often contain multiple configuration fields packed into a single 32-bit word. Understanding how to set, clear, toggle, or test specific bits without altering the rest of the register is essential for safely modifying hardware configurations and controlling peripherals.

Set and Clear 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.

BIC is essentially an AND operation with the negated mask, i.e., `BIC Rd, Rn, #mask` is equivalent to `AND Rd, Rn, #~mask`.

Check Bits To check whether certain bits are set or cleared, you can use the `AND` instruction followed by a comparison:

- `AND Rd, Rn, #mask`: Isolates bits in `Rn` where the corresponding bits in `mask` are 1. The result is stored in `Rd`.
- You can then use `CMP Rd, #0` to determine if the result is zero (all masked bits cleared) or non-zero (at least one bit set).
- Alternatively, you can use `TST Rn, #mask`, which performs the AND operation and updates the condition flags without storing the result.

The `TST` instruction and its interaction with the condition flags will be explored in more detail in the next experiment, where you will learn how conditional execution and branching depend on flag states.

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 Barrel Shifter

The barrel shifter is a hardware feature that allows for efficient shifting and rotating of register values as part of data processing instructions. It can perform operations such as logical shifts (left or right), arithmetic shifts, and rotations on the second operand (`Operand2`) before it is used in the instruction without wasting extra instructions or cycles.
Examples of barrel shifter usage:

```
ADD    R0, R2, R1, LSL #2    ; R0 = R2 + (R1 << 2) using barrel shifter
SUB    R3, R4, R5, LSR #1    ; R3 = R4 - (R5 >> 1) using barrel shifter
ORR    R6, R7, R8, ROR #3    ; R6 = R7 | (R8 rotated right by 3)
```

Listing 2.1: Barrel shifter examples

1.2 Load and Store Instructions

Since the ARM Cortex-M4 follows the RISC design philosophy, it uses a load/store architecture. This means that arithmetic and logical instructions operate only on registers. Any data in memory must first be loaded into a register before processing, and results must be stored back to memory if they need to be preserved.

Table 2.4: 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).

```
Reset_Handler
        LDR     R0, =XVAL           ; R0 = &XVAL (address of XVAL)
        LDR     R1, [R0]            ; R1 = 0x12345678 (load word from memory)
        ADR     R5, XVAL            ; R5 = PC-relative address of XVAL (if in range)

        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

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

        AREA    MYDATA, DATA, READWRITE
YVAL    DCD     0

        END
```

Listing 2.2: Examples of Load and Store Instructions

1.2.1 Declaring Data in Memory

Data in assembly is defined using *assembler directives* that reserve and optionally initialize memory. Common directives include `DCD`, `DCW`, and `DCB`, which define words, halfwords, and bytes, respectively. These are typically placed within a `DATA` area to create arrays, lookup tables, buffers, or strings.

- `DCD` — Define Constant Word (32 bits per element)
- `DCW` — Define Constant Halfword (16 bits per element)
- `DCB` — Define Constant Byte (8 bits per element)
- `SPACE` — Reserve uninitialized memory (in bytes)
- `FILL` — Fill memory with a specified value for a given length

```
            AREA    CONSTANTS, DATA, READONLY
NUMBERS     DCD     10, 20, 30, 40          ; array of 32-bit integers
BYTES       DCB     1, 2, 3, 4              ; array of bytes
TEXT        DCB     "HELLO",0               ; null-terminated ASCII string
            AREA    MYDATA, DATA, READWRITE
BUFFER      SPACE   64                      ; reserve 64 bytes (uninitialized)
PATTERN     FILL    0xFF, 64                ; fill 64 bytes with 0xFF
```

Listing 2.3: Declaring arrays and strings in memory

Each label (e.g., `NUMBERS`, `TEXT`) marks the starting address of a data object in memory. You can load these addresses into registers using `LDR R0, =NUMBERS` or `ADR R0, TEXT`, then access individual elements through the appropriate addressing modes.

Note: Strings are stored as consecutive ASCII characters in memory. A terminating zero (0x00) is typically appended to indicate the end of the string, similar to C-style strings.

1.2.2 Understanding Pointer Declarations

The directive `YPTR DCD YVAL` reserves a 32-bit word at the label `YPTR` and initializes it with the address of `YVAL`. In other words, `YPTR` acts as a *pointer variable* that holds the address of another variable (`YVAL`). Executing `LDR Rn, YPTR` loads the 32-bit contents stored at `YPTR`—that is, the address of `YVAL`—into `Rn`, making `Rn` a pointer to `YVAL`.

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

1.2.3 Loading Addresses and Values: `LDR`, `LDR =`, and `ADR`

In ARM assembly, it is important to distinguish between loading a *value* from memory and loading the *address* of a label. Although these instructions look similar, their behavior and purpose differ depending on how the assembler interprets them.

- **`LDR Rn, label`** Loads the 32-bit *value* stored at the memory address identified by `label` into register `Rn`. The CPU performs a direct memory read:

  $$\texttt{Rn} \leftarrow [\texttt{label}]$$

  Example: `LDR R0, XVAL` loads the contents of `XVAL` (e.g., `0x12345678`) into `R0`.
- **`LDR Rn, =label`** Loads the *address* of `label` into `Rn`, rather than the data stored at that address. The assembler generates this by either constructing the address using `MOVW/MOVT` instructions or placing it in a nearby *literal pool* for PC-relative loading.[1]

  $$\texttt{Rn} \leftarrow \texttt{\&label}$$

  Example: `LDR R0, =XVAL` places the address of `XVAL` in `R0`.
- **`ADR Rn, label`** Loads the *address* of `label` into `Rn` by computing it relative to the current program counter. This method requires no literal pool or memory access but only works for nearby addresses (about ±4 KB in Thumb mode):

  $$\texttt{Rn} \leftarrow \texttt{PC} + \texttt{offset(label)}$$

1.3 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 2.5: 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.

[1] For more details, see ARM Developer Documentation, "Literal pools and `LDR =const`". https://developer.arm.com/documentation/dui0473/m/dom1359731147760

```
; 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 2.4: Examples of Offset, Pre-indexed, and Post-indexed Addressing Modes

2 Procedure

2.1 Examples

2.1.1 Example 1 — Data Processing Instructions

This example demonstrates various arithmetic and bitwise operations on registers.

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

        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
        ; Arithmetic operations
        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)
        ; Shift and Barrel Shifter
        LSL     R8, R4, #4          ; logical left shift by 4 bits
        LSR     R9, R4, #8          ; logical right shift by 8 bits
        ADD     R5, R5, R5, LSL #1  ; R5 = R5 + (R5 << 1) = 3 * R5
        ; 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

        AREA CONSTANTS, DATA, READONLY
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
    END
```

Listing 2.5: Arithmetic and bitwise operations example

2.1.2 Example 2 — 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
        LDR     R0, =ARRAY          ; R0 = &ARRAY (base address)
        ; 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.
        END
```

Listing 2.6: 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 (verify in hex and binary):

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

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

2.2.2 Task 2 — Addressing Modes with an Array

Given:

```
ARRAY   DCD  0x11, 0x22, 0x33, 0x44
OUT     SPACE 16
```

Load each element using a different addressing mode, then store to `OUT`:

- 0x11 via *immediate offset*
- 0x22 via *pre-indexed*
- 0x33 via *post-indexed*
- 0x44 via *register offset*

Hint: Put `ARRAY`'s base in `R1` (e.g., `LDR R1, =ARRAY`). Verify `OUT` in memory after execution.