

BIRZEIT UNIVERSITY

Department of Electrical & Computer Engineering
ENCS4110 – Computer Design Laboratory

Experiment 01

Introduction to ARM Cortex-M4 Assembly

Date: November 2025

1 Introduction to ARM Cortex-M4 Assembly

Learning Objectives

After completing this experiment, you will be able to:

- Identify the main components of the ARM Cortex-M4 architecture, including general-purpose registers, stack pointer, link register, program counter, and program status register (xPSR).
- Write, assemble, and debug a minimal ARM assembly program containing a vector table and `Reset_Handler`.
- Use core data-processing, shift/rotate, and compare/test instructions to manipulate register data.
- Apply conditional execution and branching using condition codes (EQ, NE, GT, LT, etc.).
- Debug assembly programs in **Keil uVision5** using breakpoints, single-stepping, and register/memory inspection to analyze instruction effects.

Experiment Overview

This experiment introduces the fundamentals of ARM Cortex-M4 assembly programming and the structure of a minimal embedded program. You will explore the processor's register set, status flags, and memory map, and learn how to write and debug low-level code that manipulates data and controls program flow.

In this experiment, you will:

- Examine the Cortex-M4 register architecture and understand how flags in the xPSR reflect instruction results.
- Build and assemble a minimal startup image with a vector table and `Reset_Handler`.
- Practice data-processing and control-flow instructions, observing how they affect registers and flags.
- Use the **Keil uVision5** debugger to single-step through instructions and inspect register and memory changes.

By the end of this lab, you will understand the core building blocks of ARM assembly programming, from program structure and instruction execution to debugging at the register level.

Contents

1	Theoretical Background	3
1.1	Cortex-M4 Architecture	3
1.1.1	Registers Overview	3
1.1.2	Memory Mapping	3
1.2	Assembly Language Basics	3
1.2.1	Instruction Set Overview	4
1.2.2	General Instruction Format	4
1.3	Basic Program Template (Boilerplate)	5
2	Procedure	6
2.1	Setting Up the Keil uVision5 Environment	6
2.1.1	Creating a New Project	6
2.1.2	Debugging and Running the Program	7
2.2	Examples	8
2.2.1	Example 1: Simple Arithmetic Operations and Flag Manipulation	8

1 Theoretical Background

1.1 Cortex-M4 Architecture

1.1.1 Registers Overview

The Cortex-M4 architecture includes a set of general-purpose registers (R0-R12), a stack pointer (SP), a link register (LR), a program counter (PC), and a program status register (xPSR), all of which are 32-bit registers. The general-purpose registers are used for data manipulation and temporary storage during program execution. The SP is used to manage the call stack, while the LR holds the return address for function calls. The PC points to the next instruction to be executed, and the xPSR contains flags and status information about the processor state.

Program Status Register (xPSR) holds the current state of the processor, including condition flags (Negative, Zero, Carry, Overflow), interrupt status, and execution state. These flags are updated based on the results of arithmetic and logical operations, allowing for conditional branching and decision-making in programs.

1.1.2 Memory Mapping

The Cortex-M4 uses a flat memory model, where all memory locations are accessible through a single address space. This model simplifies programming and allows for efficient access to data and instructions. The memory is divided into several regions, including code memory (for storing instructions), data memory (for storing variables), and peripheral memory (for interfacing with hardware components). The architecture supports both little-endian and big-endian data formats, with little-endian being the default.

Table 1.1: Cortex-M4 Memory Regions (ARMv7-M)

Region	Address Range	Description
Code	0x00000000–0x1FFFFFFF	Flash memory for program code
SRAM	0x20000000–0x3FFFFFFF	On-chip static RAM for data
Peripheral	0x40000000–0x5FFFFFFF	Memory-mapped peripheral registers
External RAM	0x60000000–0x9FFFFFFF	External RAM (if implemented)
External Device	0xA0000000–0xDFFFFFFF	External devices/memory (if implemented)
Private Peripheral Bus	0xE0000000–0xE00FFFFF	Cortex-M4 internal peripherals (NVIC, SysTick, MPU, SCB)
System	0xE0100000–0xFFFFFFFF	System region (reserved/system-level)

1.2 Assembly Language Basics

Assembly language is a low-level programming language that provides a direct correspondence between human-readable instructions and the machine code executed by the processor. Each instruction encodes a specific operation, such as moving data, performing arithmetic or logic, or altering control flow. Because it maps so closely to hardware, assembly allows precise control of system resources and is commonly used in performance-critical routines or when direct access to hardware is required.

Assembly programs are typically composed of three main elements: *instructions*, *directives*, and *labels*.

Instructions Instructions are the executable commands that the CPU carries out. Examples include data movement (`MOV`), arithmetic (`ADD`, `SUB`), logical operations (`AND`, `ORR`), and control flow (`B`, `BL`). Each instruction directly translates to one or more machine code opcodes and determines the actual behavior of the program.

Directives Directives are commands to the assembler that guide how source code is translated into machine code but do not generate instructions themselves. Common examples include `AREA` (define a code or data section), `ALIGN` (align data to memory boundaries), `DCD` (allocate and initialize a word of storage), and `EXPORT` (export a symbol for linking). Directives organize program layout, control memory allocation, and manage symbol visibility.

Labels Labels are symbolic names that mark specific locations in code or data. They act as targets for jumps and branches or as references for data access. Labels improve program readability and maintainability by avoiding hard-coded addresses. For instance, a label like `loop_start` can be used as the destination of a branch instruction, and the assembler automatically computes the correct relative address.

1.2.1 Instruction Set Overview

The ARM Cortex-M4 instruction set is a subset of the ARMv7-M architecture, designed for efficient execution in embedded systems. It includes a variety of instructions for data processing, memory access, and control flow. Key categories of instructions include:

- **Data Processing Instructions:** These include arithmetic operations (e.g., `ADD`, `SUB`), logical operations (e.g., `AND`, `ORR`), and data movement instructions (e.g., `MOV`, `MVN`).
- **Load and Store Instructions:** Instructions like `LDR` (load register) and `STR` (store register) are used to transfer data between registers and memory.
- **Branch Instructions:** Control flow is managed using branch instructions such as `B` (branch), `BL` (branch with link), and conditional branches like `BEQ` (branch if equal).
- **Special Instructions:** These include instructions for system control, such as `NOP` (no operation), `WFI` (wait for interrupt), and instructions for manipulating the stack and handling exceptions.

1.2.2 General Instruction Format

Assembly source lines generally follow this shape:

```
[label] OPCODE{<cond>}{S} operands ; comment
```

where curly braces {} denote optional components, and:

- **label:** optional symbolic name marking the current address.
- **OPCODE:** instruction mnemonic (e.g., `ADD`, `MOV`, `B`).
- **<cond>:** optional condition code (e.g., `EQ`, `NE`, `LT`, `GE`) that predicates execution.
- **S:** optional suffix indicating whether to update the condition flags (e.g., `ADDS`).
- **operands:** registers, immediates, or memory operands (e.g., `R0`, `R1`, `#1` or `[R2]`).
- Anything after a semicolon (;) is a comment ignored by the assembler.

```
loop_start    ADDS    R0, R0, #1      ; R0 = R0 + 1, update flags (N,Z,C,V)
```

Listing 1.1: Instruction format example

1.3 Basic Program Template (Boilerplate)

The minimal skeleton below shows a valid vector table in a READONLY RESET area, a READWRITE data area for variables, and a code area with the `Reset_Handler` entry point.

```
AREA      RESET, CODE, READONLY      ; Vector table lives in read-only code
EXPORT    __Vectors                 ; Make symbol visible to the linker
__Vectors
  DCD      0x20001000               ; Initial SP value (top of stack in SRAM)
  DCD      Reset_Handler           ; Reset vector: entry address
  ALIGN
; ----- Read-Write Data -----
AREA      M_DATA, DATA, READWRITE   ; Variables go here (RAM)
EXPORT    COUNT                     ; Export if referenced by other modules
COUNT    DCD      0                  ; Initialized RW variable (word)
BUF      SPACE     16                ; Uninitialized RW buffer (16 bytes)
  ALIGN
; ----- Application Code -----
AREA      MYCODE, CODE, READONLY
ENTRY
EXPORT    Reset_Handler
Reset_Handler
; Example: COUNT++ and store to BUF[0]
  LDR      R0, =COUNT              ; R0 <- &COUNT
  LDR      R1, [R0]                ; R1 <- COUNT
  ADDS    R1, R1, #1               ; R1 = R1 + 1 (update flags)
  STR     R1, [R0]                ; COUNT <- R1
  LDR      R2, =BUF                ; R2 <- &BUF
  STRB   R1, [R2]                ; BUF[0] <- (low byte of R1)
STOP    B      STOP                ; Stay here forever
  END
```

Listing 1.2: Cortex-M4 boilerplate with READWRITE data

What each directive does

- `AREA <name>, CODE|DATA, READONLY|READWRITE`: defines a section. Put the vector table and program text in `CODE, READONLY`; put variables in `DATA, READWRITE`.
- `EXPORT <symbol>`: makes a label visible to the linker/other modules.
- `DCD, DCW, DCB <values>`: allocate and initialize words, halfwords, or bytes.
- `SPACE <n>`: reserve n bytes of uninitialized storage in RAM.
- `ALIGN`: align to a suitable boundary (commonly 4 bytes for words).
- `ENTRY`: mark the entry point of the image for the toolchain.
- `END`: end of source file.

Notes

- The first two words in `__Vectors` must be the initial stack pointer value and the address of `Reset_Handler`.
- The assembler/linker places sections in appropriate memory regions based on the target device and linker script.
- Labels must start at the beginning of the line (no indentation), while instructions and directives should be indented for proper assembly.
- Variables in `READWRITE` areas are initialized to zero by default. While you can specify initial values using directives like `DCD`, the linker will place these in flash and copy them to RAM during startup, or they may be zeroed out during RAM initialization.

2 Procedure

2.1 Setting Up the Keil uVision5 Environment

Make sure you have the **Keil uVision5** IDE installed on your computer. If not, download and install it from the official Keil website (<https://www.keil.com/demo/eval/arm.htm>).

2.1.1 Creating a New Project

1. Open **Keil uVision5** and create a new project:
 - Go to **Project > New uVision Project...**
 - Choose a directory and name for your project (e.g., `Exp01_ARM_Assembly`).
2. Select the target device:
 - In the "Select Device for Target" dialog, choose ARM Cortex-M4 (ARMCM4) as we will be using it only for simulation.
 - Click "OK" to confirm.
3. Configure project settings:
 - Go to **Project > Options for Target 'Target 1'...**
 - Under the "Debug" tab, select "Use Simulator" as the debug driver.
4. Add a new assembly file to the project:
 - Right-click on "Source Group 1" in the Project window and select **Add New Item to Group 'Source Group 1'...**
 - In the "Add New Item" dialog, select "Assembly File" from the list.
 - Name the file (e.g., `main.s`) and click "Add".
5. Build the project:
 - Click on the "Build" button (or go to **Project > Build Target**), or use the shortcut F7.
 - Check the "Output" window for any errors or warnings. If there are errors, fix them in your assembly code and rebuild.
 - Once the build is successful, you should see a message indicating that the build was completed without errors.
6. Start a debugging session:
7. Click on the "Debug" button (or go to **Debug > Start/Stop Debug Session**), or use the shortcut **Ctrl + F5**.

2.1.2 Debugging and Running the Program

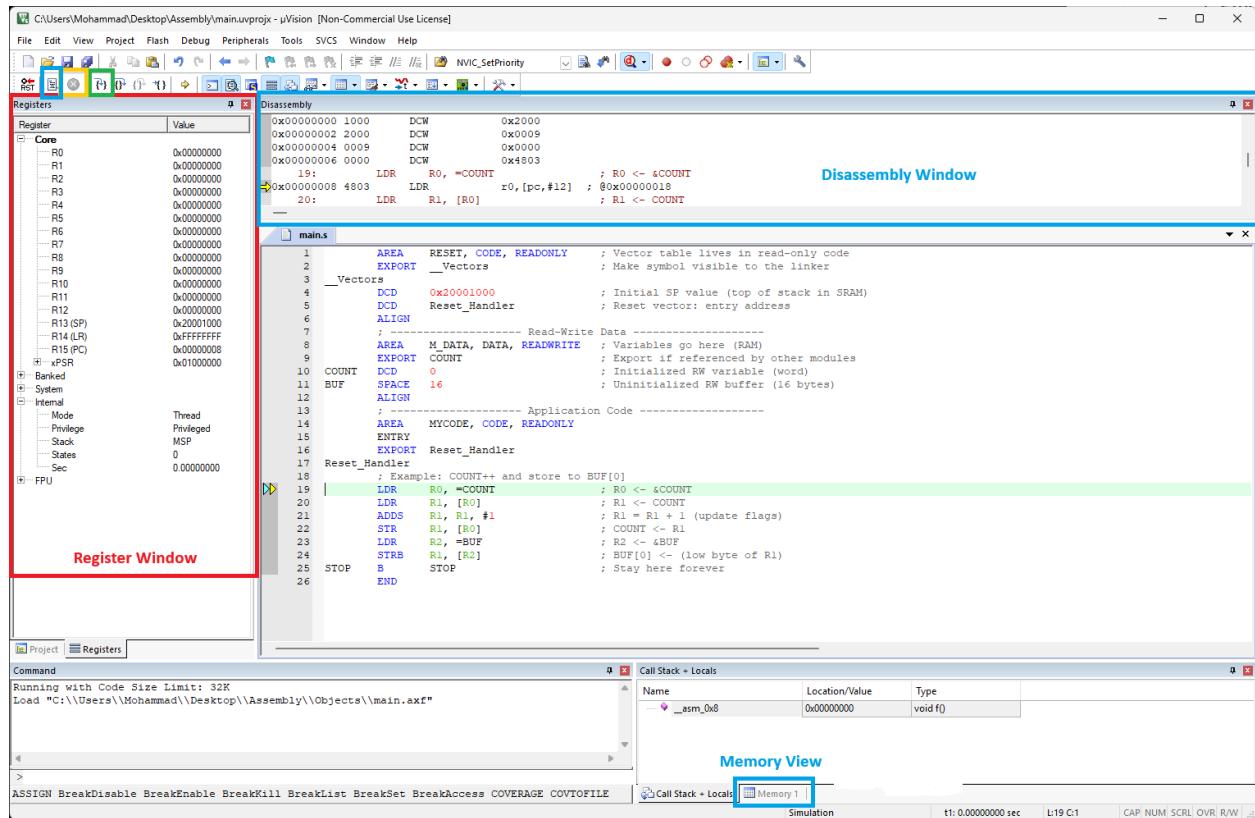


Figure 1.1: Keil uVision5 Debugging Interface

Figure 1.1 shows the Keil uVision5 debugging interface. You can run and debug your assembly program using two main approaches:

1. Step by Step Execution:

- Use the "Step" button (or press F11) marked in green in Figure 1.1 to execute your program one instruction at a time.
- Observe the changes in the registers and memory as you step through each instruction.

2. Run the entire program:

- Use the "Run" button (or press F5) marked in blue in Figure 1.1 to execute your program continuously until it hits a breakpoint or completes execution.
- After running, you must stop the execution using the "Stop" button marked in yellow in Figure 1.1.
- Check the final values in the registers and memory to verify the program's behavior.

2.2 Examples

2.2.1 Example 1: Simple Arithmetic Operations and Flag Manipulation

The following example demonstrates basic arithmetic operations, flag manipulation.

```
AREA      RESET, CODE, READONLY
EXPORT    __Vectors
__Vectors
    DCD      0x20001000          ; Initial SP (example top-of-stack)
    DCD      Reset_Handler      ; Reset vector
    AREA    MYCODE, CODE, READONLY
    ENTRY
    EXPORT  Reset_Handler
Reset_Handler
; ===== Part A: Moving immediates (MOV, MOVW/MOVT, MOV32, LDR =) =====
; 1) Simple 8/12-bit immediate
    MOV      R2, #0x01          ; R2 = ? (after step)
; 2) 16-bit low half into R5
    MOV      R5, #0x3210          ; R5 = ? (low 16 bits set)
; 3) High half into R5: combine with (2)
    MOVT    R5, #0x7654          ; R5 = ? (now 0x76543210)
; 4) 32-bit immediate with MOV32 macro (emits MOVW+MOVT)
    MOV32   R6, #0x87654321      ; R6 = ?
; 5) Literal load of a 32-bit immediate
    LDR      R7, =0x87654321      ; R7 = ?
; ===== Part B: ADD/SUB without and with flag updates =====
    MOV      R2, #0x02          ; R2 = ?
    MOV      R3, #0x03          ; R3 = ?
    ADD      R1, R2, R3          ; R1 = ? (flags unchanged)
; Now set R3 to all ones, then add with flags
    MOV32   R3, #0xFFFFFFFF      ; R3 = ?
    ADDS    R1, R2, R3          ; R1 = ? FLAGS? (N,Z,C,V)
    SUBS    R1, R2, R3          ; R1 = ? FLAGS? (N,Z,C,V)
; Same add but without S (no flag update)
    MOV      R4, #0xFF          ; small value for contrast
    ADD     R1, R2, R4          ; R1 = ? FLAGS? (should be unchanged)
; Now with S, so flags DO update
    ADDS    R1, R2, R4          ; R1 = ? FLAGS? (N,Z,C,V)
; ===== Part C: Overflow / Zero / Negative flag demos =====
; Create an ADD overflow with two large positive numbers
    MOV32   R2, #0x7FFFFFFF      ; R2 = ?
    MOV32   R3, #0x7FFFFFFF      ; R3 = ?
    ADDS    R1, R2, R3          ; R1 = ? Overflow expected? FLAGS?
; Create a ZERO result
    MOV      R2, #1
    SUBS   R1, R2, #1           ; R1 = ? (=0) FLAGS? (Z should be 1)
; Create a NEGATIVE result
    SUBS   R1, R2, #2           ; R1 = ? (negative) FLAGS? (N should be 1)
STOP    B      STOP            ; infinite loop
END
```

Listing 1.3: Example 1: Simple Arithmetic Operations and Flag Manipulation