1 ARM Cortex-M4 Assembly Fundamentals

Learning Objectives

- Describe the Cortex-M4 register set, xPSR flags, and ARMv7-M memory map.
- Write and assemble a minimal ARM program with a vector table and `Reset_Handler`.
- Use core data-processing, shift/rotate, and compare/test instructions.
- Apply conditional execution with condition codes (e.g., `ADDEQ`, `BNE`).
- Debug in **Keil uVision5**: set breakpoints, single-step, inspect registers/memory, and interpret flags.

Experiment Overview

This experiment introduces low-level programming on the Cortex-M4. You will build a minimal startup image, practice common data-processing and control-flow instructions, and use the **Keil uVision5** debugger to observe register and memory effects while stepping through code. By the end, you will be able to implement and test short assembly routines that manipulate data and make decisions based on condition flags.

Contents

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	0x0000_0000 – 0x1FFF_FFFF	Flash / code memory
SRAM	0x2000_0000 – 0x3FFF_FFFF	On-chip static RAM
Peripheral	0x4000_0000 – 0x5FFF_FFFF	Memory-mapped peripheral registers
External RAM	0x6000_0000 – 0x9FFF_FFFF	External RAM (if implemented)
External Device	0xA000_0000 – 0xDFFF_FFFF	External devices / memory (if implemented)
PPB (Private Peripheral Bus)	0xE000_0000 – 0xE00F_FFFF	Cortex-M4 internal peripherals (NVIC, SysTick, MPU, SCB)
System	0xE010_0000 – 0xFFFF_FFFF	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)
             CMP    R0, #10         ; compare R0 with 10 (sets flags)
             BLT    loop_start      ; branch if R0 < 10 (uses flags)
```

Listing 1.1: Instruction format example

1.2.3 Conditional Execution

Most ARM instructions can be made *conditional* by appending a two-letter *condition code* to the mnemonic (e.g., `ADDEQ`, `ADDNE`). The instruction is executed only if the condition evaluates true based on the `xPSR` flags N (negative), Z (zero), C (carry), and V (overflow). When no condition is supplied, the default is `AL` (always).

Table 1.2: Common ARM Condition Codes

Cond.	Meaning	Description
EQ	Equal	Execute if $Z = 1$.
NE	Not equal	Execute if $Z = 0$.
CS/HS	Carry set / Unsigned higher or same	Execute if $C = 1$.
CC/LO	Carry clear / Unsigned lower	Execute if $C = 0$.
MI	Minus (negative)	Execute if $N = 1$.
PL	Plus (non-negative)	Execute if $N = 0$.
VS	Overflow set	Execute if $V = 1$.
VC	Overflow clear	Execute if $V = 0$.
HI	Unsigned higher	Execute if $C = 1$ *and* $Z = 0$.
LS	Unsigned lower or same	Execute if $C = 0$ *or* $Z = 1$.
GE	Greater or equal (signed)	Execute if $N = V$.
LT	Less than (signed)	Execute if $N \neq V$.
GT	Greater than (signed)	Execute if $Z = 0$ *and* $N = V$.
LE	Less or equal (signed)	Execute if $Z = 1$ *or* $N \neq V$.
AL	Always	Always execute (default if no condition).
NV	Never	Reserved / do not use.

Example:

```
CMP     R0, #0          ; set flags from (R0 - 0)
ADDEQ   R1, R1, #1      ; if Z==1: R1++
ADDNE   R2, R2, #1      ; if Z==0: R2++
```

Listing 1.2: Using conditional-suffix mnemonics

1.2.4 Barrel Shifter

The barrel shifter is a hardware feature that allows . . . 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 1.3: Barrel shifter examples

1.3 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.

1.3.1 Arithmetic Instructions

Table 1.3: 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.

Logical and Move Instructions

Table 1.4: 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.

1.3.2 Shift and Rotate Instructions

Table 1.5: 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`.

1.3.3 Compare and Test Instructions

Table 1.6: 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.

1.4 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 1.4: 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

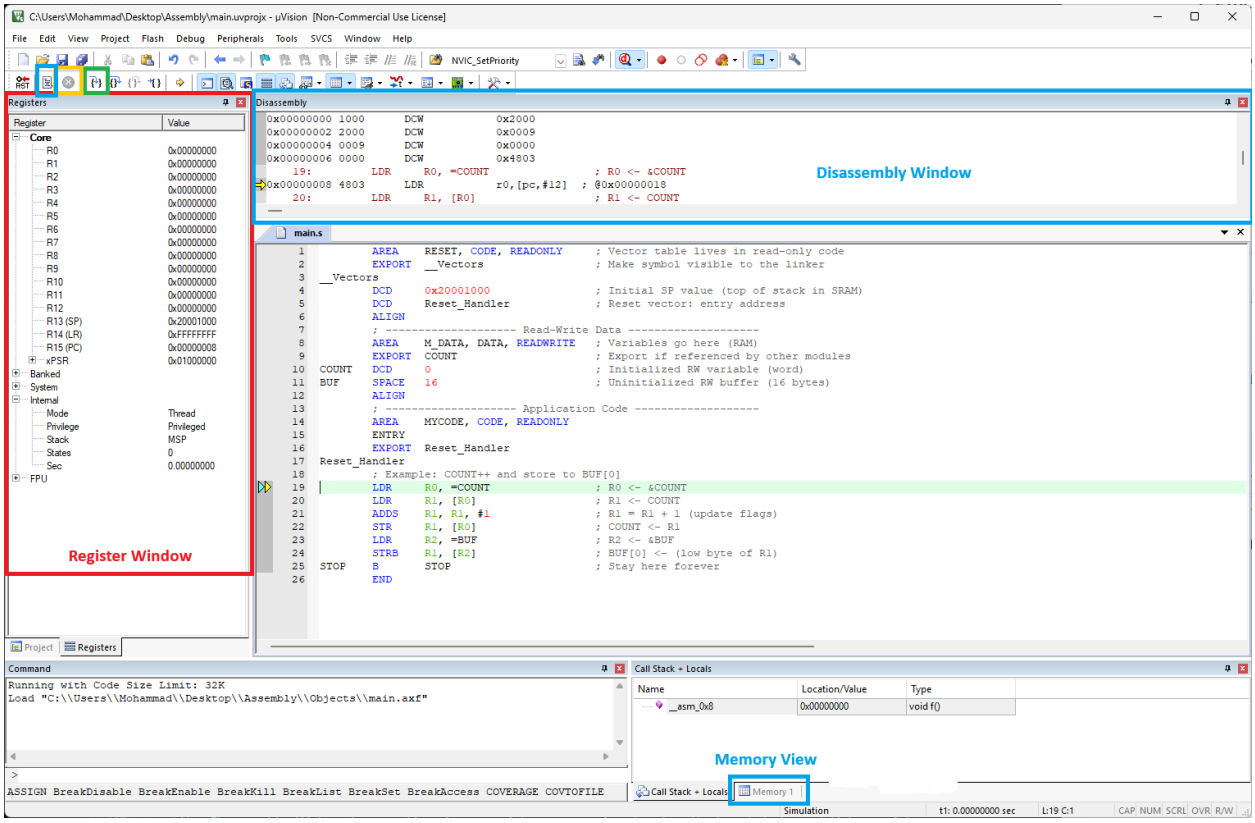

Figure 1.1: Keil uVision5 Debugging Interface

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 — 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, DATA, 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 1.5: Arithmetic and bitwise operations example

2.2.2 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, DATA, 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 1.6: Status flags and logical tests example

2.3 Tasks

2.3.1 Task 1 — Bitwise Operations

Write an ARM assembly program that manipulates the contents of a register using bitwise operations (AND, OR, NOT, XOR) starting with Register R0 containing the value `0x0BADC0DE`.

- Clear bits 7–13 (inclusive)
- Set bits 17–23 (inclusive)
- Flip bits 24–31 (inclusive)

Hint: Use appropriate masks with `BIC`, `ORR`, and `EOR` instructions to manipulate specific bit ranges.

2.3.2 Task 2 — Arithmetic Operations

Write an ARM assembly program that performs the following arithmetic operations assuming R0 = 10 and R1 = 5:

- Add the values in R0 and R1, store the result in R2
- Subtract the value in R1 from R0, store the result in R3
- Multiply the values in R0 and R1, store the result in R4
- Divide the value in R0 by R1, store the result in R5 (use unsigned division)
- Using barrel shifter only, find the multiplication of R0 by 9, store the result in R6