Recitation 3 – Introduction to ARM Assembly Language

Overview

This recitation introduces **ARM Assembly Language**, focusing on:

- ARM assembly syntax and structure
- Register usage and load/store architecture
- Writing simple functions in ARM Assembly
- Looping and conditional execution
- Recursion and stack management

ARM Assembly is a **low-level programming language** that maps closely (often 1-to-1) with machine code instructions executed directly by the CPU. Unlike high-level languages such as C/C++, assembly gives you explicit control over registers, memory, and execution flow.

ARM Assembly Basics

General Syntax

```
Label: Opcode Destination, Operand1, Operand2   // Comment
```

Components:

- **Label** (optional): Marks a memory location (used for loops/branches)
- **Opcode**: Instruction to execute (e.g., `ADD`, `MOV`, `SUB`)
- **Destination**: Register where the result is stored
- **Operands**: Source registers or immediate values
- **Comment**: Starts with `//` or `@`

Example

```
add r0, r0, r1   // r0 = r0 + r1
```

Core Concepts

Registers (Workspace)

ARM processors perform arithmetic **only on registers**, not directly on memory.

Register	Purpose

Register	Purpose
R0–R12	General-purpose registers
R13 (SP)	Stack Pointer
R14 (LR)	Link Register (stores return address)
R15 (PC)	Program Counter (next instruction address)

Load/Store Architecture

ARM uses a **load/store architecture**:

- `LDR` – Load from memory → register
- `STR` – Store from register → memory

Example restriction:

Invalid:

```
add [memory_address], #1
```

Correct approach:

```
ldr r0, [address]
add r0, r0, #1
str r0, [address]
```

Assembler Directives

These guide the assembler (not executed by CPU):

- `.global symbol` – Makes function visible to linker
- `.text` – Start of code section

Reference:

- ARM Assembly Directives (GNU AS): https://sourceware.org/binutils/docs/as/ARM-Directives.html

---

Conditional Execution (Suffixes)

ARM allows conditional execution via suffixes:

Instruction	Meaning
`B`	Branch always

Instruction	Meaning
`BNE`	Branch if Not Equal
`BEQ`	Branch if Equal
`ADDNE`	Add if Not Equal

Implemented Functions

---

1. Add Two Numbers

Description

Simple function that:

- Takes two arguments
- Performs addition
- Returns result in `r0`

Calling Convention

- `r0` → first number
- `r1` → second number
- Return value → `r0`

Implementation

```
.global add_asm

// Function: add_asm
// R0 = a, R1 = b
// Returns: R0 = a + b
add_asm:
    add r0, r0, r1
    bx lr
```

---

2. Sum Elements of an Array (Basic Loop)

Description

Iterates through an array of bytes and accumulates their sum.

Parameters

- `r0` → pointer to array

- `r1` → number of elements

Implementation

```
// Function: summation1
// r0: pointer to array
// r1: number of elements (n)

summation1:
    push {r4, lr}
    mov r4, #0          // accumulator = 0

add_loop1:
    ldrb r2, [r0], #1   // load byte, increment pointer
    add r4, r4, r2      // add to accumulator
    subs r1, r1, #1     // decrement counter
    bne add_loop1       // loop if not zero

    mov r0, r4          // return result
    pop {r4, lr}
    bx lr
```

---

3. Sum Elements (Optimized Variation)

Description

Uses **conditional execution (`IT` blocks)** to reduce branching and improve efficiency.

Implementation

```
summation2:
    push {r4, lr}
    mov r4, #0
    add r1, r1, #1          // Adjust offset for loop logic
sum_loop2:
    subs r1, r1, #1
    ldrb r2, [r0], #1

    IT NE                   // If-Then block (execute next if Not Equal)
    addne r4, r4, r2        // Conditional Add

    bne sum_loop2
    mov r0, r4
    pop {r4, lr}
    bx lr
```

Optimization Idea

- Uses conditional execution (`ADDNE`)
- Reduces unnecessary branch instructions
- Can improve pipeline efficiency on some ARM architectures

---

4. Factorial Function (Recursive)

Description

Demonstrates:

- Recursion
- Stack usage (`push` / `pop`)
- Preserving registers
- Proper handling of the Link Register (`lr`)

Parameter

- `r0` → integer `n`

Implementation

```
// Function: factorial
// r0: integer n

factorial:
    cmp r0, #1              // Check base case
    ble base_case           // If n <= 1, go to base_case

    push {r1, lr}           // Save state (Link Register is crucial!)
    mov r1, r0              // Copy n to r1
    sub r0, r0, #1          // n = n - 1
    bl factorial            // Recursive Call

    mul r0, r0, r1          // r0 = factorial(n-1) * n
    pop {r1, lr}            // Restore state
    bx lr

base_case:
    mov r0, #1              // Return 1
    bx lr
```

Key Concepts Demonstrated

- `bl` stores return address in `lr`
- `push {r1, lr}` preserves state across recursive calls
- Stack ensures correctness during nested recursion
- Base case prevents infinite recursion

Key Takeaways

- ARM Assembly gives precise control over CPU operations.
- Arithmetic operations must use registers.
- Memory access requires explicit load/store instructions.
- Conditional execution can optimize performance.
- Stack management is essential for recursion.
- Understanding calling conventions is critical for interoperability with C/C++.