3 ARM Cortex-M4 Flow Control, Procedures, and Stack

Learning Objectives

After completing this experiment, you will be able to:

- Implement conditional and unconditional branching using ARM branch instructions.
- Design and implement loop constructs (`for`, `while`) using compare and branch instructions.
- Create and call procedures with proper parameter passing and return mechanisms.
- Manage the stack for local variables, parameter passing, and nested procedure calls.
- Apply the ARM Procedure Call Standard (AAPCS) to ensure correct register usage and calling conventions.

Experiment Overview

This experiment introduces program control flow, procedure implementation, and stack management on the ARM Cortex-M4 processor. You will learn how to control program execution using branches and loops, create modular and reusable code using procedures, and manage memory efficiently through stack operations.

In this experiment, you will:

- Implement conditional branches and loop structures at the assembly level.
- Write and call procedures that follow standard calling conventions.
- Handle nested calls and parameter passing using the stack.
- Observe and analyze how the stack changes during procedure entry and return.

By the end of this lab, you will understand how to use branching and looping for program control, write well-structured procedures that follow AAPCS rules, and manage the stack during function calls and nested procedures.

Contents

1 Theoretical Background

1.1 Flow Control

Flow control instructions alter the sequential execution of instructions by changing the program counter (PC). These instructions enable the implementation of conditional statements, loops, and procedure calls that are fundamental to structured programming.

1.1.1 Condition Evaluation

Before implementing control flow, we must understand how to evaluate conditions and set processor flags. ARM provides dedicated instructions for comparing values and testing bit patterns that update the condition flags without storing results.

Table 3.1: ARM Cortex-M4 Compare and Test Instructions

Instr.	Syntax	Description / Usage
CMP	`CMP Rn, Operand2`	Compare `Rn` with `Operand2` (`Rn - Operand2`); updates flags (Z, N, C, V).
CMN	`CMN Rn, Operand2`	Compare negative (`Rn + Operand2`); used for checking against negative values.
TST	`TST Rn, Operand2`	Logical AND test (`Rn AND Operand2`); sets Z if all tested bits are 0.
TEQ	`TEQ Rn, Operand2`	Logical XOR test (`Rn EOR Operand2`); sets Z if operands are equal.

Usage Notes:

- These instructions only affect the condition flags (N, Z, C, V) — they do not store a result.
- `CMP`/`CMN` are arithmetic comparisons; `TST`/`TEQ` are logical bitwise comparisons.
- Many data-processing instructions can update flags by appending `S` (e.g., `ADDS`, `SUBS`).
- Common use: immediately followed by conditional branches such as `BEQ`, `BNE`, `BGT`, etc.

Examples:

```
        CMP     R0, #10          ; Compare R0 with 10
        BLT     LessThan10       ; Branch if R0 < 10
        BGE     GreaterOrEqual   ; Otherwise, R0 >= 10
```

Listing 3.1: Arithmetic comparison using CMP

```
        MOV     R1, #0x12        ; R1 = 0001 0010b
        TST     R1, #0x10        ; Test if bit 4 is set
        BEQ     BitClear         ; Branch if bit 4 = 0 (Z=1)
        BNE     BitSet           ; Branch if bit 4 = 1 (Z=0)
```

Listing 3.2: Bit test using TST

```
        MOV     R2, #0x55
        MOV     R3, #0x55
        TEQ     R2, R3           ; XOR -> result 0, sets Z=1
        BEQ     ValuesEqual      ; Branch if equal
```

Listing 3.3: Equality check using TEQ

1.1.2 Conditional Branching

Branch instructions are the primary mechanism for implementing flow control in ARM assembly. They modify the program counter to jump to different parts of the code based on conditions or unconditionally.

Table 3.2: ARM Cortex-M4 Branch Instructions

Instr.	**Syntax**	**Description / Usage**
B	`B label`	Unconditional branch to `label` (always jumps)
B<cond>	`B<cond> label`	Conditional branch based on flags
BL	`BL label`	Branch with link: calls a subroutine, storing return address in `LR`.
BX	`BX Rm`	Branch to address in register, often `BX LR` to return from a subroutine.
CBZ	`CBZ Rn, label`	Branch if `Rn == 0`. Example: `CBZ R0, Done`.
CBNZ	`CBNZ Rn, label`	Branch if `Rn != 0`. Example: loop until counter reaches zero.

`CBZ/CBNZ` instructions have specific constraints:
- **Register**: operand must be a low register `R0-R7`.
- **Range**: branch is *forward-only*; the destination must be within 0–126 bytes after the instruction.
- **Flags**: does not update condition flags (`N, Z, C, V`).

For backward or longer jumps, use `CMP/TST` with conditional branches (`BEQ`, `BNE`, `BGT`, ...).

1.1.3 Conditional Execution

ARM assembly supports conditional execution, where most instructions can be conditionally executed based on the current state of the condition flags. This feature allows for efficient implementation of conditional statements without explicit branching.

Conditional Instruction Format: Most ARM instructions can be made conditional by appending a condition code suffix:

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

Examples:
- `ADDEQ R0, R1, R2` — Add only if equal (Z=1)
- `MOVNE R3, #10` — Move only if not equal (Z=0)
- `SUBGT R4, R4, #1` — Subtract only if greater than (signed)

Advantages of Conditional Execution:
- **Performance**: Eliminates branch instructions for simple conditional operations
- **Code density**: Reduces the number of instructions needed
- **Pipeline efficiency**: Avoids branch prediction penalties for simple conditions
- **Atomic operations**: Multiple related conditional operations can be grouped

1.1.4 How Branch Instructions Work

Branch instructions change the flow of execution by modifying the Program Counter (`PC`). When a branch is executed, the instruction encodes an *offset* which is added to the current value of the `PC`.

Offset calculation: The branch instruction contains a signed immediate value (positive or negative). The processor adds this offset (aligned to halfword boundaries) to the current `PC`.

- A *positive offset* causes a **forward branch** (jump to a higher memory address, later in the program).
- A *negative offset* causes a **backward branch** (jump to a lower memory address, earlier in the program).

Example: Suppose a branch instruction is located at address `0x100`, and the assembler encodes an immediate offset of `-0x08`. The effective target address will be:

$$0x100 + 4 + (-0x08) = 0xFC$$

This means the processor jumps **backward** to an earlier instruction. Such negative offsets are typically used to implement loops (e.g., repeat until zero).

1.1.5 Condition Codes

Conditional branches use condition codes that test the processor status flags (N, Z, C, V) set by previous instructions. These conditions enable implementation of high-level constructs like if-statements and loops.

Table 3.3: 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.

1.2 Loop Patterns

Loops are fundamental control structures that repeat a block of code based on conditions. ARM assembly implements loops using combinations of compare instructions, conditional branches, and counters.

1.2.1 For Loop Structure

A typical for loop has the structure: initialization, condition check, body execution, and increment/decrement. This type of loop executes a known number of times.

```
        AREA M_DATA, DATA, READONLY
array   DCD 10, 20, 30, 40, 50    ; array of 5 integers
length  EQU 5                     ; number of elements (; just a constant, no memory)
```

Listing 3.4: Declaring Array and Length

```
        ; Initialization
        MOV     R0, #0                  ; i = 0
        MOV     R1, #0                  ; sum = 0
        LDR     R3, =array              ; load base address of array into R3

for_start
        ; Condition check
        CMP     R0, #length             ; compare i with length
        BGE     for_end                 ; if i >= length, exit loop

        ; Loop body
        LDR     R2, [R3, R0, LSL #2]    ; load array[i]; EA: R3 + (R0 * 4)
        ADD     R1, R1, R2              ; sum += array[i]

        ; Increment
        ADD     R0, R0, #1              ; i++
        B       for_start               ; repeat

for_end
```

Listing 3.5: For loop implementation pattern

1.2.2 While Loop Structure

While loops check the condition before executing the loop body, potentially executing zero times if the initial condition is false. This type of loop is useful when the number of iterations is not known in advance and depends on dynamic conditions.

```
        AREA M_DATA, DATA, READONLY
mystring DCB "Hello World!", 0     ; null-terminated string
```

Listing 3.6: Declaring Null-Terminated String

Note: `0` and `'0'` are two different values, as the former is actually zero, while the latter is the ASCII code for the character '0' (which is 48 in decimal).

```
    ; Intialization
    LDR     R0, =mystring   ; pointer to string
    MOV     R1, #0          ; character count = 0

while_start
                            ; Condition check
    LDRB    R2, [R0], #1    ; load current character and post-increment pointer
    CMP     R2, #0          ; check for null terminator
    BEQ     while_end       ; if zero, exit loop

                            ; Loop body - do something with R2


    B       while_start     ; repeat
while_end
```

Listing 3.7: While loop with string processing example

1.3 Procedures and Stack

1.3.1 ARM Architecture Procedure Call Standard (AAPCS)

Procedures are reusable blocks of code that encapsulate a specific task. They promote modular design, code reuse, and clearer program structure. In ARM assembly, procedures are implemented using branch-and-link instructions along with register usage conventions defined by the ARM Architecture Procedure Call Standard (AAPCS).

The AAPCS is the set of rules that define how functions exchange data and how registers must be preserved during a procedure call:

- **R0–R3**: Hold the first four parameters. `R0` also holds the return value. Caller-saved.
- **Stack**: Any additional parameters beyond the first four are passed on the stack.
- **R4–R11**: Must be preserved by the callee. If a procedure uses them, it must save and restore them.
- **SP (R13)**: Stack pointer, always points to the current top of the stack.
- **LR (R14)**: Link register holds the return address. Caller-saved.

Note: Callees are the procedures being called, while callers are the ones calling the procedure.

1.3.2 Procedure Templates

A procedure is entered with a `BL` (branch-with-link) instruction, which stores the return address in the link register `LR`. The callee returns by branching to `LR` (e.g., `BX LR`). By the AAPCS, the first four arguments are passed in `R0-R3` and the primary return value is placed in `R0`.

Basic Procedure Template

Example: simple procedure that expects two integers in `R0` and `R1` and returns their sum in `R0`.

```
AddTwo  PROC
        ADD R0, R0, R1    ; return R0+R1 in R0
        BX  LR
        ENDP
```

Listing 3.8: Basic procedure structure

Note: The `PROC` and `ENDP` directives help define the start and end of a procedure and they could be safely omitted.

Procedure with Preserved Registers When a procedure uses callee-saved registers (R4-R11), it must preserve their original values by saving them on the stack and restoring them before returning:

```
ProcessArray PROC
        PUSH    {R4-R6, LR}       ; Save used registers and LR
        MOV     R4, R0            ; Save array pointer
        MOV     R5, R1            ; Save array length
        MOV     R6, #0            ; Initialize counter

        ; Process array using R4, R5, R6...

        MOV     R0, R6            ; Return counter value
        POP     {R4-R6, PC}       ; Restore registers and return
        ENDP
```

Listing 3.9: Procedure using preserved registers

Nested Procedure Calls When one procedure calls another, the link register (`LR`) must be preserved, otherwise the return address would be lost. This is done by pushing `LR` onto the stack before making another call.

```
OuterProc
        PUSH    {LR}              ; Save return address
        BL      InnerProc         ; Call inner procedure
        MOV     R1, R0            ; Use return value
        POP     {PC}              ; Return to caller

InnerProc
        MOV     R0, #42           ; Return value
        BX      LR                ; Return
```

Listing 3.10: Nested procedure example

1.3.3 Stack Operations

The stack is a memory region used to hold return addresses, local variables, and saved registers. On the Cortex-M4, the stack is implemented as a *full descending stack*: it grows from high memory addresses to low addresses, and the stack pointer (`SP`) always points to the last stored value.

1.3.4 Stack Model (Full, Descending)

There are four common stack models based on two characteristics: whether the stack is full or empty, and whether it grows up (ascending) or down (descending).

- **Full Ascending**: SP points to the last used location; stack grows toward higher addresses.
- **Empty Ascending**: SP points to the next free location; stack grows toward higher addresses.
- **Empty Descending**: SP points to the next free location; stack grows toward lower addresses.
- **Full Descending**: SP points to the last used location; stack grows toward lower addresses.

The ARM Cortex-M4 uses a **full descending stack**, meaning:

- The stack pointer (`SP`) points to the last used location (full).
- The stack grows toward lower memory addresses (descending).

1.3.5 Stack Operations

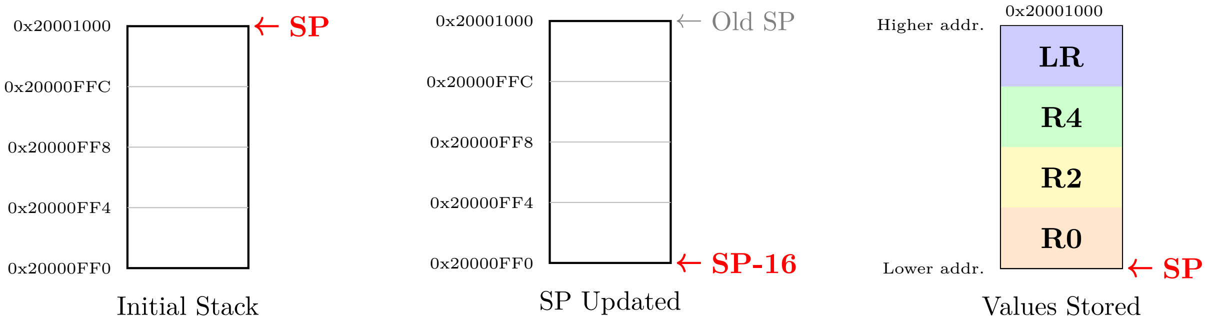
The Cortex-M4 provides `PUSH` and `POP` instructions that automatically update the stack pointer (`SP`) and allow saving or restoring multiple registers in a single instruction. These operations are essential for implementing procedure calls, local variables, and context switching.

PUSH The `PUSH` instruction saves one or more registers onto the stack in a single operation. When executed, the stack pointer (`SP`) is decremented to reserve space, and the specified registers are written to consecutive memory locations starting from the new `SP` value. **Rule:** `PUSH` stores registers on the stack, with the **lowest-numbered register** placed at the **lowest memory address** and the **highest-numbered register** placed at the **highest memory address**. This guarantees a consistent and predictable memory layout for saved contexts.

Example:

```
PUSH {R4, R0, R2, LR}
```

Even if the list appears unordered, values are laid out by register number: R0 (lowest address), then R2, R4, and LR (highest address). Each register occupies 4 bytes, so `SP` decreases by 16 bytes in total. The program counter (`PC`) cannot be pushed by this instruction.



(a) Initial empty stack (b) Stack pointer decremented (c) Registers stored in order

Figure 3.1: Stack operation sequence for `PUSH {R4, R0, R2, LR}`

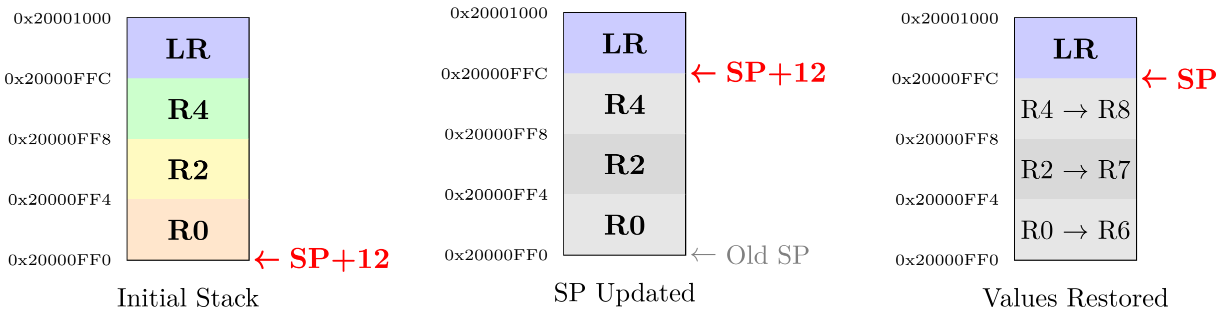
Explanation: Figure 3.1**(a)** shows the empty stack with `SP` at the top (highest address). When `PUSH` executes, the processor **decrements `SP` by 16 bytes** to reserve space for four registers (Figure 3.1**(b)**). Finally, in Figure 3.1**(c)**, values are stored according to the rule: the lowest-numbered register (R0) at the lowest address, then R2 and R4, and the highest-numbered (LR) at the highest address. Because the stack grows downward, `SP` points to the last written word after the operation.

POP The `POP` instruction restores one or more registers from the stack. Conceptually, the processor reads each value from the addresses currently covered by `SP` and then releases that stack space. **Rule:** `POP` loads registers from the stack such that the **lowest-numbered register** is restored from the **lowest memory address**, and the **highest-numbered register** from the **highest memory address**. After all specified registers are restored, `SP` has increased by 4 bytes per register.

Example:

```
        POP {R6-R8}
```

This restores R6, then R7, then R8 from successively higher addresses; the stack pointer increases by 12 bytes overall. If `PC` is included in the list, the loaded value becomes the new program counter, returning from the current procedure.



(a) Initial stack after Figure 3.1 (b) Stack pointer incremented (c) Registers restored in order

Figure 3.2: Stack operation sequence for `POP {R6-R8}`

Explanation: In Figure 3.2**(a)**, the stack contains values saved earlier. Executing `POP {R6-R8}` **reclaims 12 bytes of stack space** (three registers) by moving `SP` upward in memory (Figure 3.2**(b)**). Then, as shown in Figure 3.2**(c)**, registers are restored following the rule: the lowest-numbered register (R6) comes from the lowest address, followed by R7 and R8 from higher addresses. After the last load, `SP` points to the top of the reclaimed block, completing the reversal of the earlier push.

2 Procedure

2.1 Examples

2.1.1 Example 1: Array Example — Find Maximum Element

This example demonstrates how to find the maximum element in an array using a standard for loop structure.

```
        AREA    RESET, CODE, READONLY
        EXPORT  __Vectors
__Vectors
        DCD     0x20001000          ; Initial SP
        DCD     Reset_Handler       ; Reset vector
        ALIGN

        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler

; Find maximum element in ARR[0..LEN-1]
; Result (max) is written to MAXRES.
Reset_Handler
        LDR     R0, =ARR            ; R0 = &ARR[0]
        MOV     R1, #0              ; R1 = i (index)
        LDR     R2, [R0]            ; R2 = max = ARR[0]

for_start
        CMP     R1, #LEN            ; i >= LEN ?
        BGE     for_end             ; yes -> done

        LDR     R3, [R0, R1, LSL #2]; R3 = ARR[i]
        CMP     R3, R2              ; if ARR[i] > max
        MOVGT   R2, R3              ;    max = ARR[i]
        ADD     R1, R1, #1          ; i++
        B       for_start

for_end
        LDR     R4, =MAXRES         ; store result for easy checking
        STR     R2, [R4]

STOP    B       STOP

        AREA    CONSTANTS, DATA, READONLY
ARR     DCD     10, 20, 30, -5, 11, 0
LEN     EQU     6

        AREA    MYDATA, DATA, READWRITE
MAXRES  DCD     0                   ; expect 30

        END
```

Listing 3.11: Find maximum element in an array

Check: Verify that the maximum element is correctly identified and stored in `MAXRES`.

2.1.2 Example 2: String Example — Count Uppercase Letters

This example demonstrates how to process a null-terminated string and count the number of uppercase letters (A-Z) using a while-loop structure.

```
        AREA    RESET, CODE, READONLY
        EXPORT  __Vectors
__Vectors
        DCD     0x20001000          ; Initial SP
        DCD     Reset_Handler       ; Reset vector
        ALIGN

        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler

; Count uppercase ASCII letters in the null-terminated string MYSTR.
; Result (count) is written to UPPERCOUNT.
Reset_Handler
        LDR     R0, =MYSTR          ; R0 = ptr to string
        MOV     R1, #0              ; R1 = count

while_next
        LDRB    R2, [R0], #1        ; R2 = *p++; post-increment pointer
        CBZ     R2, while_end       ; if '\0' -> exit

        CMP     R2, #'A'            ; below 'A'?
        BLT     while_next
        CMP     R2, #'Z'            ; above 'Z'?
        BGT     while_next

        ADD     R1, R1, #1          ; count++

        B       while_next

while_end
        LDR     R3, =UPPERCOUNT
        STR     R1, [R3]

STOP    B       STOP

        AREA    CONSTANTS, DATA, READONLY
MYSTR   DCB     "Hello ARM World!", 0

        AREA    MYDATA, DATA, READWRITE
UPPERCOUNT DCD  0                   ; expect 5 ('H','A','R','M','W')

        END
```

Listing 3.12: Count uppercase letters in a string

Check: Verify that the program correctly counts the uppercase letters and stores the result in UPPERCOUNT.

2.1.3 Example 3: Stack Example — Nested Uppercase Counter

This example demonstrates a nested call: `CountUpperNested(ptr)` scans a null-terminated string and calls `IsUpper(ch)` for each character. It shows saving/restoring `LR` and using a callee-saved register (`R4`) for the running count.

```
        AREA    RESET, CODE, READONLY
        EXPORT  __Vectors
__Vectors
        DCD     0x20001000          ; Initial SP
        DCD     Reset_Handler       ; Reset vector
        ALIGN
        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler
; IsUpper(R0 = ch) -> R0 = 1 if 'A'..'Z', else 0
IsUpper
        CMP     R0, #'A'
        BLT     not_upper
        CMP     R0, #'Z'
        BGT     not_upper
        MOV     R0, #1
        BX      LR
not_upper
        MOV     R0, #0
        BX      LR
; CountUpperNested(R0 = ptr) -> R0 = count of uppercase letters
CountUpperNested
        PUSH    {R4, LR}            ; save callee-saved + return address
        MOV     R1, R0              ; R1 = ptr (keep pointer here)
        MOV     R4, #0              ; R4 = count

cu_next
        LDRB    R0, [R1], #1        ; R0 = *ptr++; post-increment pointer in R1
        CBZ     R0, cu_done         ; if null terminator, finish
        BL      IsUpper             ; R0 = 0/1 based on 'A'..'Z'
        ADD     R4, R4, R0          ; count += result
        B       cu_next
cu_done
        MOV     R0, R4              ; return count in R0
        POP     {R4, PC}
Reset_Handler
        LDR     R0, =mystring
        BL      CountUpperNested
        LDR     R2, =UPPERCOUNT
        STR     R0, [R2]
STOP    B       STOP
        AREA    CONSTANTS, DATA, READONLY
mystring DCB    "Hello ARM World!", 0  ; Uppercase: H, A, R, M, W -> 5
        AREA    MYDATA, DATA, READWRITE
UPPERCOUNT  DCD 0                      ; should be 5
        END
```

Listing 3.13: Nested procedure call to count uppercase letters

Check: Verify that `UPPERCOUNT` contains 5 for the test string.

2.2 Tasks

2.2.1 Task 1: Count Vowels in a String

Implement procedures to process strings with the following requirements:

- Create a procedure `CountVowels` that takes a string pointer in R0 and returns the number of vowels (a, e, i, o, u) in R0.
- Use nested procedure calls where `CountVowels` calls a helper procedure `IsVowel`.
- Follow AAPCS conventions for parameter passing and register usage.

2.2.2 Task 2: Factorial Calculation (Iterative)

Implement a procedure to calculate the factorial of a non-negative integer:

- Create a procedure `Factorial` that takes a non-negative integer in R0 and returns its factorial in R0.
- Use an iterative approach with a loop to compute the factorial.
- Ensure proper handling of edge cases, such as $0! = 1$.
- Follow AAPCS conventions for parameter passing and register usage.

2.2.3 Task 3: Factorial Calculation (Recursive)

Implement a recursive version of the factorial calculation:

- Create a procedure `FactorialRec` that takes a non-negative integer in R0 and returns its factorial in R0.
- Use recursion to compute the factorial, ensuring proper base case handling.
- Manage the stack appropriately to save and restore registers as needed.
- Follow AAPCS conventions for parameter passing and register usage.
- Test the procedure with various inputs to verify correctness.