CSCI-2406 / CSCI-6461

An Introduction to

Computer Architecture

On ARM32/64

John Burns and Sardar Ziyatkhanov

Contents

One

Introduction to ARM32 and RISC

Two

Anatomy of an Executable Program

2.1 ELF Structure for Embedded ARM Systems (Undergraduate Level)

2.1.1 Introduction to ELF on ARM

The **Executable and Linkable Format (ELF)** is a standard file format for executables, object files, shared libraries, and core dumps. It is widely used on Unix-like systems and is the default binary format for development on ARM-based platforms.

On ARM systems, ELF files are generated by compilers and linkers such as `arm-none-eabi-gcc`. These files contain both the machine code and the metadata needed for linking, loading, and debugging. ELF is an *architecture-neutral* format. Its structure accommodates different processor types and instruction sets. For example, in a 32-bit ARM binary, the ELF header field `e_machine` is set to `EM_ARM`.

In *embedded systems*—such as ARM Cortex-M microcontrollers—the ELF file is typically **statically linked** and mapped to fixed memory addresses like flash and SRAM. This differs from *application processors* (e.g., ARM Cortex-A running Linux), where executables are **dynamically linked** and mapped into virtual memory by the operating system loader.

Understanding the ELF format is crucial for ARM developers. It allows inspection of how a program is structured, helps in writing custom linker scripts, and assists with debugging memory layout issues. ELF serves as the foundation for executable software across bare-metal and OS-based ARM environments.

2.1.2 ELF Headers and Section Basics

Each ELF file begins with an **ELF header**, which identifies the file as an ELF binary and provides essential information to interpret the rest of the file. This includes magic numbers (`0x7F`, followed by the ASCII characters `E`, `L`, `F`), the architecture class (32-bit or 64-bit), endianness, and the target machine type (e.g., `EM_ARM`).

The header also contains file offsets to two critical tables:

- **Program Header Table (PHT)** – describes how to map the file into memory at runtime (used by loaders).

- **Section Header Table (SHT)** – describes individual sections of the file (used by linkers and debuggers).

An ELF file essentially presents two distinct views:

- The **section view** is used during linking and debugging. It provides fine-grained components such as code, data, and symbols.
- The **segment view** is used during program loading. It organizes data into larger memory segments.

Common ELF sections found in embedded ARM executables include:

- `.text` – executable machine code
- `.rodata` – read-only constants such as string literals
- `.data` – initialized read/write variables
- `.bss` – uninitialized data, zeroed at runtime
- `.symtab`, `.strtab` – symbol and string tables (used for debugging)

In contrast, the Program Header Table defines memory segments:

- `LOAD` segments define regions that should be loaded into memory (e.g., code into flash, data into RAM).
- Segments may include multiple sections—for example, `.text` and `.rodata` may be grouped into one segment.

Key Distinction:. Sections are intended for the linker and debugger; segments are meant for the loader. Understanding both is crucial for embedded developers who must control exactly what is placed in flash versus RAM.

Three

Assembly Language Programming

Four

Branching and Iteration

4.1 Introduction to Program Control Flow

In previous chapters, we have examined how to load and store values to and from memory. However, real programs do not simply execute instructions in a straight line — they must make decisions and repeat operations. In higher-level languages, this is accomplished with 'if' statements, 'while' loops, 'for' loops, 'switch' cases, and function calls. In ARM Assembly, we must implement these control flow mechanisms using branch instructions.

Control Flow in Assembly

At the heart of control flow in ARM Assembly are instructions that affect the Program Counter (`PC`). The `PC` determines which instruction the CPU executes next. Most instructions implicitly increment `PC` to the next instruction, but branch instructions explicitly update it to a new target.

Control Flow Instructions in ARM Assembly:

- `b label` – Unconditional branch to `label`.
- `bl label` – Branch with link; jumps to a function and stores return address in the Link Register (`lr`).
- `bx lr` – Return from function by branching to the address in `lr`.
- `bne`, `beq`, `bgt`, etc. – Conditional branches that execute based on status flags.

The Role of the Program Counter (PC)

In ARM32, due to the fetch-decode-execute pipeline, reading from the PC actually gives the address of the instruction *two steps ahead*, i.e., `PC + 8`. In ARM64, this is simplified to `PC + 4`. This offset is important when performing PC-relative calculations such as loading data from memory based on the current program position.

Branch Instruction Encoding and Range

The `b` and `bl` instructions use relative addressing with a signed immediate value. In ARM32:

- The offset is 24 bits wide and shifted left by 2, allowing for a range of $\pm 2^{25}$ bytes, or ±32MB.

In ARM64:

- The offset is 26 bits wide, also shifted left by 2, resulting in a branch range of ±128MB.

If a branch target is outside this range, the address must be loaded into a register, and an indirect branch performed:

```
ldr x0, =target_address
br x0          // ARM64 indirect branch
```

Branching and the Pipeline

Modern ARM cores use deep pipelines to maximize throughput. However, branches can disrupt the pipeline:

- If a branch is correctly predicted, the pipeline continues smoothly.
- If mispredicted, the processor must flush instructions and refetch from the correct path, causing a delay.

Core	Branch Misprediction Penalty
Cortex-A53	10 cycles
Cortex-A76	15 cycles
Neoverse V1	20+ cycles

Table 4.1: Misprediction Penalties in Common ARM Cores

Why Control Flow Matters

Control flow constructs — branches, loops, and function calls — enable the creation of algorithms, decision-making structures, and code reuse. Efficient use of branches is critical in performance-sensitive applications, such as signal processing or real-time embedded systems.

In the sections that follow, we will explore how these ideas translate into practical assembly code, including:

- Creating branches based on comparisons.
- Writing loops with different structures (while, do-while, for).
- Minimizing the performance cost of branching using techniques such as conditional execution and loop unrolling.

4.2 ARM Branch Instruction Set

In ARM Assembly, branching is the fundamental method of controlling program flow. Branch instructions modify the Program Counter (`PC`), allowing a jump to another instruction in the program. These jumps may be conditional or unconditional, direct or indirect, and are essential for implementing loops, conditionals, and function calls.

Core Branch Instructions

The most basic branching operations in ARM Assembly are:

- `b label` – Unconditional branch to `label`.
- `bl label` – Branch to `label` and save return address in the Link Register (`lr`).
- `bx lr` – Return from subroutine (branch to address in `lr`).
- `blx reg` – Branch with link to address in `reg`, also supports interworking between ARM and Thumb states.

```
bl my_function   // Call function
...
my_function:
  ; function body
  bx lr          // Return
```

The ‘bl‘ instruction automatically stores the address of the next instruction in `lr`, making it possible to return using `bx lr`.

Branch Delay Slots and Hazards

Although ARM processors do not expose branch delay slots in the way some architectures (e.g., MIPS) do, branches still introduce **pipeline hazards**.

When a branch is executed, the processor may have already fetched subsequent instructions based on speculation. If the branch is taken, and the prediction was incorrect, these prefetched instructions must be flushed from the pipeline, resulting in a stall.

Load-use delay hazard:

```
ldr r0, [r1]       // Load word
add r2, r0, #1     // Uses r0 too soon
```

On some ARM cores, a stall may occur if the result from `ldr` is used in the next cycle. To mitigate this, we can insert an independent instruction:

```
ldr r0, [r1]
add r3, r4, r5      // Independent, fills delay
add r2, r0, #1      // Safe now
```

Conditional Branching and IT Blocks

ARM supports many condition codes to allow for branching based on status flags (N, Z, C, V). These are automatically updated by most arithmetic instructions.

Common conditional branch instructions:

- `beq` – Branch if equal (Z = 1)
- `bne` – Branch if not equal (Z = 0)
- `bgt` – Branch if greater than (Z = 0 and N = V)
- `blt` – Branch if less than (N ≠ V)
- `bge` – Branch if greater than or equal (N = V)

Example:

```
cmp r0, r1
bgt bigger
mov r2, #0          // if r0 <= r1
b end
bigger:
mov r2, #1          // if r0 > r1
end:
```

In Thumb-2, the `IT` (If-Then) instruction allows limited conditional execution of up to 4 instructions without branching:

```
cmp r0, #10
ittt ge              // If-Then-Then-Then block
addge r1, r1, #1
addge r2, r2, #1
addge r3, r3, #1
```

The IT block improves performance by reducing branch frequency and allowing more predictable instruction flow.

Thumb-2 and ARM64 Enhancements

Modern ARM64 introduces new branching capabilities that are both compact and powerful:

- `b.cond label` – Compact conditional branch (e.g., `b.eq`, `b.ne`, `b.ge`).
- `br xN` – Branch to address in register `xN`.
- `ret xN` – Return from subroutine (equivalent to `bx lr`).
- `braa xN, xM` – Branch with return address authentication (used for security).

Example – ARM64 return mechanism:

```
bl some_function
...
some_function:
  ; do something
  ret             // return to caller
```

These enhancements are particularly relevant in performance- and security-critical applications, such as operating system kernels and cryptographic routines.

ARMv8.3 also introduces **Pointer Authentication Codes (PAC)**, allowing the use of ‘braa‘ for secure, authenticated control transfers.

4.3 Conditional Execution

Modern ARM processors support **conditional execution**, a powerful feature that allows certain instructions to execute only if specific conditions (based on the status flags) are met. This eliminates the need for short branches in many cases and helps to reduce pipeline disruptions.

Condition Flags Overview

ARM architecture maintains a special register called the **Application Program Status Register (APSR)**. This register stores condition flags that reflect the result of the most recent arithmetic or logical operation.

- **N (Negative)** – Set if the result is negative (bit 31 of result is 1).
- **Z (Zero)** – Set if the result is zero.
- **C (Carry)** – Set if there was a carry out (for unsigned arithmetic).
- **V (Overflow)** – Set if there was a signed overflow.

Example – Flag update by comparison:

```
cmp r0, r1  // Updates N, Z, C, and V
bge next    // Branch if r0 >= r1 (N == V)
```

These flags are implicitly updated by arithmetic instructions such as ‘add‘, ‘sub‘, ‘cmp‘, and ‘movs‘. Conditional branches and conditional instruction execution depend on the values of these flags.

Using Flags in Practice

ARM supports a full set of conditional suffixes that can be applied to many instructions, allowing them to execute only when a condition is true.

Common condition suffixes:

Suffix	Meaning	Condition
`eq`	Equal	$Z = 1$
`ne`	Not equal	$Z = 0$
`lt`	Less than (signed)	$N \neq V$
`ge`	Greater than or equal (signed)	$N = V$
`cs`	Carry set (unsigned $\geq$)	$C = 1$
`cc`	Carry clear (unsigned $<$)	$C = 0$

Table 4.2: ARM Condition Suffixes

Example – Using conditional arithmetic:

```
cmp r0, r1
addgt r2, r0, r1  // if r0 > r1
sublt r2, r1, r0  // if r0 < r1
```

This technique is particularly useful when minimizing branch penalties. Rather than using ‘bgt‘ or ‘blt‘ to jump around the code, the instructions themselves are conditionally suppressed or executed.

Performance Considerations

Conditional execution can improve performance in tight loops or time-critical code by:

- Reducing the number of branches.
- Preventing pipeline flushes due to mispredictions.
- Increasing code density and reducing memory fetches.

However, not all instructions are equally efficient when executed conditionally. Some instructions, such as multiplication or memory loads, may have longer execution times when conditional execution is involved.

Comparison of instruction timings:

Instruction	Unconditional	Conditional
ADD	1 cycle	1 cycle
MUL	3 cycles	4 cycles
LDR	3 cycles	4 cycles

Table 4.3: Execution Timing: Unconditional vs. Conditional

While conditional execution is a useful tool, it should be applied judiciously. In modern ARM64 cores, the conditional instruction set is more limited than in older ARMv7-A profiles. In such cases, you may be required to use short conditional branches instead.

4.4 Loop Constructs and Patterns

Looping structures are essential for repeating instructions in programs. In C, loops are expressed as `while`, `for`, and `do-while` constructs. ARM Assembly does not have these keywords, so all loops must be built explicitly using branches and comparison instructions.

This section explains how to implement these high-level loop constructs using ARM's conditional branches and registers.

While Loop in Assembly

A `while` loop in C repeatedly checks a condition before executing the loop body.

C code:

```c
int i = 10;
while (i > 0) {
  // loop body
  i--;
}
```

ARM Assembly:

```
mov r1, #10          // i = 10
b test_condition     // jump to condition check

loop_body:
  ; your loop logic here
  subs r1, r1, #1    // i--

test_condition:
  cmp r1, #0
  bgt loop_body      // loop if i > 0
```

Note: 'subs' subtracts and updates flags; 'cmp' checks the loop condition.

For Loop Using SUBS

A `for` loop has explicit initialization, condition, and iteration components.

C code:

```
for (int i = 5; i > 0; i--) {
  // loop body
}
```

ARM Assembly:

```
mov r1, #5            // i = 5

loop:
  ; loop body here

  subs r1, r1, #1     // i--
  bgt loop            // loop if i > 0
```

This form is compact and efficient. It uses ‘subs‘ to both decrement the counter and update the flags for ‘bgt‘.

Do-While Loop

The `do-while` loop guarantees that the body executes at least once.

C code:

```
int i = 5;
do {
  // loop body
  i--;
} while (i > 0);
```

ARM Assembly:

```
mov r1, #5

loop:
  ; loop body here
  subs r1, r1, #1
  bgt loop            // repeat if i > 0
```

Unlike the previous loops, this structure places the condition check *after* the body, ensuring at least one execution.

Nested Loops and Flowcharts

Nested loops involve placing one loop inside another. These are common in array or matrix processing.

C code:

```c
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 2; j++) {
    // inner loop body
  }
}
```

ARM Assembly:

```
mov r0, #0          // i = 0
outer_loop:
  cmp r0, #3
  bge end_outer

  mov r1, #0        // j = 0
inner_loop:
  cmp r1, #2
  bge end_inner

  ; inner loop body here

  add r1, r1, #1
  b inner_loop

end_inner:
  add r0, r0, #1
  b outer_loop

end_outer:
```

Each loop must use separate registers for indexing (‘r0‘, ‘r1‘) and must have clearly marked start and end labels.

*

Visualizing Control Flow

To understand nested loop structure better, we can sketch a flowchart:

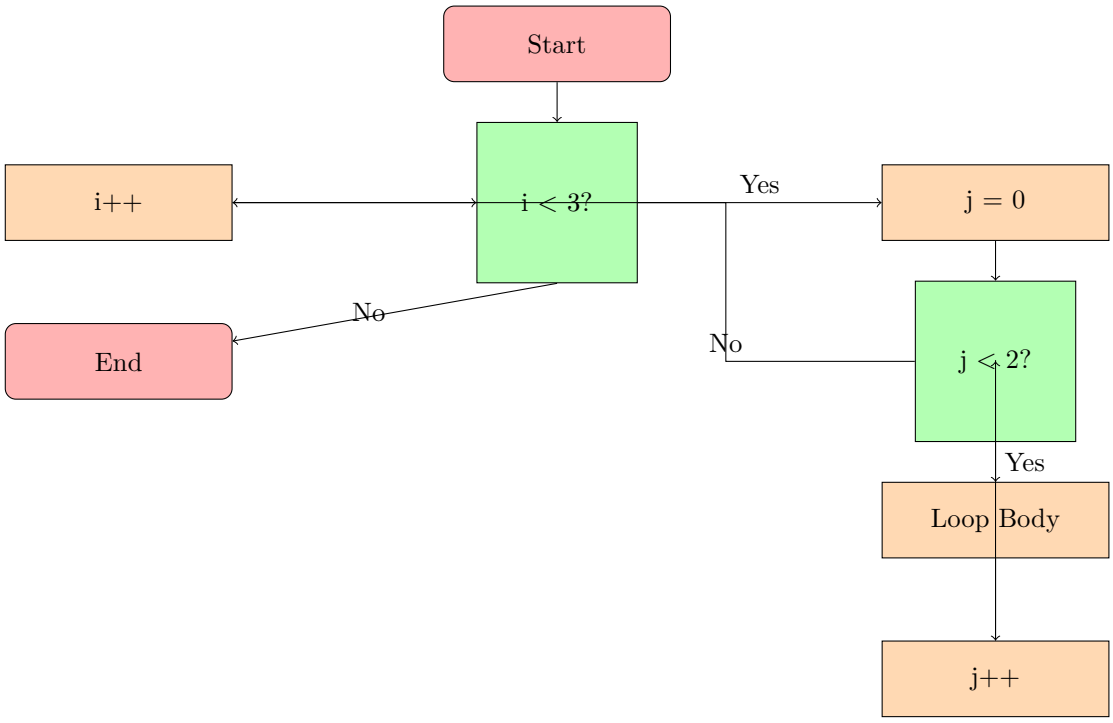


Figure 4.1: Flowchart of Nested Loops

4.5 Optimizing Loops

In performance-critical applications, loops often dominate execution time. Optimizing how loops are written in assembly can yield significant improvements. This section discusses three fundamental loop optimization techniques used in ARM Assembly programming: software pipelining, loop unrolling, and loop tiling.

Software Pipelining

Software pipelining is an instruction scheduling technique that overlaps the execution of operations from different loop iterations. It helps to hide instruction latency and keep the pipeline full.

Consider this loop which loads and processes one value per iteration:

```
loop:
  ldr r0, [r1], #4      // Load next value from array
  add r2, r2, r0        // Accumulate
  subs r3, r3, #1       // Decrement counter
  bne loop              // Repeat if not zero
```

Each instruction depends on the previous one, limiting parallelism.
Software-pipelined version:

```
  ldr r4, [r1], #4     // Load first value (prologue)

loop:
  ldr r0, [r1], #4     // Load next value (next
   iteration)
  add r2, r2, r4       // Use previous value
  mov r4, r0           // Save current for next
   iteration
  subs r3, r3, #1
  bne loop
```

Here, we start by preloading one value and overlapping computation in subsequent iterations. This reduces stalls and improves throughput on superscalar cores.

Loop Unrolling

Loop unrolling involves replicating the loop body multiple times per iteration to reduce the overhead of branching and increase instruction-level parallelism.
C version (before unrolling):

```
for (int i = 0; i < 4; i++) {
  sum += array[i];
}
```

ARM version (unrolled x4):

```
ldr r0, [r1], #4
add r2, r2, r0

ldr r0, [r1], #4
add r2, r2, r0

ldr r0, [r1], #4
add r2, r2, r0

ldr r0, [r1], #4
add r2, r2, r0
```

Benefits of unrolling:

- Fewer branches (reduced control overhead).

- Better instruction scheduling (more ILP).
- Useful when the loop trip count is known and small.

Drawbacks:

- Increases code size.
- May lead to register pressure or cache pressure.

Loop Tiling and Cache Optimization

Loop tiling (also called blocking) improves cache utilization by breaking large loops into smaller tiles that fit into the processor's cache. This is particularly effective for multidimensional arrays and matrix operations.

Tiling Formula: Let:

- L_1 = L1 cache size in bytes
- E = size of each array element in bytes

Then an approximate optimal tile size T is:

$$T = \left\lfloor \sqrt{\frac{L_1}{3E}} \right\rfloor$$

Example: If L1 = 32KB and each element is 4 bytes:

$$T = \left\lfloor \sqrt{\frac{32768}{12}} \right\rfloor = \left\lfloor \sqrt{2730} \right\rfloor = 52$$

Tiled Matrix Multiply:

```
@ Loop over tiles
mov x3, #0           // i
tile_outer:
  cmp x3, N
  bge end_outer

  mov x4, #0         // j
tile_inner:
  cmp x4, N
  bge end_inner

  ; process tile[i][j] here

  add x4, x4, T      // j += tile size
  b tile_inner
end_inner:
```

```
  add x3, x3, T      // i += tile size
  b tile_outer
end_outer:
```

Loop tiling reduces memory latency by ensuring repeated access to data that remains in cache. It is widely used in compilers and libraries for numerical computing.

4.6 Security and Control Flow

Pointer Authentication (PAC)

Pointer authentication adds a cryptographic signature to return addresses, mitigating control-flow hijacking attacks:

```
pacga x0, x1   @ Generate PAC
braa x0, x1    @ Authenticated return
```

Speculation Barriers

Barriers prevent speculation-based attacks by serializing execution:

```
dsb sy         @ Data sync
isb sy         @ Instruction sync
csdb           @ Speculation barrier
```

4.7 Security and Control Flow

In modern ARM architectures, security is a major consideration, especially with the growing concern over control flow hijacking attacks such as return-oriented programming (ROP). ARMv8.3 introduces **Pointer Authentication Codes (PAC)**, which helps protect against these kinds of attacks by signing pointers before they are used in control flow operations. This section explores how PAC works and how control flow in ARM can be secured using various techniques.

Pointer Authentication (PAC)

Pointer Authentication adds a layer of security by ensuring that pointers (such as return addresses) cannot be easily manipulated. ARMv8.3 introduced the concept of PAC, which uses a cryptographic hash to sign pointers. This allows the processor to verify that the pointer has not been tampered with before it is dereferenced.

PAC Calculation: The PAC is computed using the pointer value, the Stack Pointer (SP), and a key, as shown in the following formula:

$$PAC = \text{Pointer} \oplus \text{SP}^{\text{Modifier}} \oplus \text{Key}$$

Here: - `Pointer` is the address being signed (e.g., return address). - `SP` is the stack pointer, which provides the context of the address. - `Modifier` and `Key` are cryptographic values used to generate a unique PAC for each pointer.

PAC Instructions: ARM provides instructions for signing and verifying pointers: - `pacga x0, x1` – Generate a PAC for pointer `x0` using `x1` as the modifier. - `autia x0, x1` – Authenticate the address in `x0` using `x1` as the key. - `braa x0, x1` – Authenticated branch to the address in `x0` using `x1` as the key.

Example – Pointer signing and branching:

```
pacga x0, x1       // Generate PAC for pointer in x0
   using x1
braa x0, x1        // Authenticated branch to address
   in x0
```

This technique prevents attackers from overwriting return addresses and hijacking program flow. By verifying that the PAC matches, ARM processors can detect tampered pointers.

Speculation Barriers

Speculative execution is a feature of modern processors designed to improve performance by guessing the results of instructions before they are fully executed. However, speculative execution can introduce security vulnerabilities, such as **Spectre** and **Meltdown**, which allow attackers to exploit speculative execution and leak sensitive data.

To mitigate these issues, ARMv8.5 introduced new instructions to control speculation, known as **speculation barriers**. These barriers prevent the processor from speculating execution past a certain point, thereby reducing the risk of attacks.

Speculation Barrier Instructions:

- `dsb sy` – Data Synchronization Barrier, ensures that all previous memory operations are completed before continuing.
- `isb sy` – Instruction Synchronization Barrier, flushes the instruction pipeline and ensures that all instructions are synchronized.
- `csdb` – Consumption Speculation Barrier, ensures that no speculative operations can be performed before this instruction.

Example – Inserting a speculation barrier:

```
dsb sy          // Ensure all memory operations are
   completed
isb sy          // Flush the instruction pipeline
```

These barriers are useful in preventing speculative execution from leaking sensitive data and ensuring that all instructions are executed in a secure order.

Return Stack Buffer (RSB) and Control Flow Integrity

To further secure control flow, ARM processors use a **Return Stack Buffer (RSB)**, which is used to predict the return addresses of function calls. This helps the processor quickly return from subroutines without needing to repeatedly fetch and decode the return address from memory.

The `RSB` holds a stack of return addresses, and if the return address is predicted incorrectly, the processor can flush the pipeline and fetch the correct address.

RSB and Control Flow Integrity: ARM's RSB and the use of PAC help to ensure **Control Flow Integrity (CFI)** by preventing unauthorized control transfers. This makes it harder for attackers to overwrite function pointers or return addresses to hijack program control.

Real-World Applications of Control Flow Security

Control flow security is critical in systems that handle sensitive data, such as operating systems, cryptographic libraries, and secure communications. By preventing unauthorized control transfers, these security measures help protect systems from exploits like:

- Return-oriented programming (ROP) attacks
- Function pointer overwrites
- Control flow hijacking

These security features are essential for developing safe and resilient ARM-based applications, especially in environments where security is a top priority, such as mobile devices and embedded systems.

4.8 Exercises

Basic Load and Store Operations

Exercise 4.8.1 *Write an ARM assembly program that does the following:*

1. *Declares an integer variable* `x` *initialized to* `25`.
2. *Loads* `x` *into a register.*
3. *Adds* `10` *to* `x`.

4. *Stores the result back in memory.*

Use `ldr` *and* `str` *instructions to perform memory operations.*

Byte vs Word Access

Exercise 4.8.2 *Given the following memory declaration:*

```
.data
  array: .word 0x12345678
```

Perform the following operations in ARM assembly:

1. *Load the full word into a register and print its value.*
2. *Load only the least significant byte using* `ldrb` *and print its value.*
3. *Store* `0xAB` *into the least significant byte of* `array` *without modifying other bytes.*

Offset Addressing

Exercise 4.8.3 *Given an integer array in memory:*

```
.data
  numbers: .word 5, 10, 15, 20
```

Write an ARM assembly program that:

1. *Loads the second element of the array into a register.*
2. *Adds* `5` *to the second element.*
3. *Stores the modified value back into the array.*

Use offset addressing mode in the `ldr` *and* `str` *instructions.*

Pointer Arithmetic

Exercise 4.8.4 *Convert the following C code into ARM assembly:*

```
unsigned values[5] = { 2, 4, 6, 8, 10 };
unsigned *p = values;
*(p + 2) = *(p + 2) + 5;  // Modify the third element
```

1. *Identify how memory addressing is done in both C and ARM assembly.*
2. *Use a base register and appropriate addressing mode to modify the third element.*

Pointer Traversal in Assembly

Exercise 4.8.5 *Write an ARM assembly program that:*

1. *Defines an array of 6 integers.*
2. *Uses a base register (pointer) to iterate through each element.*
3. *Doubles each value in the array.*

Finding Maximum in an Array

Exercise 4.8.6 *Write a C function that finds the maximum value in an array using pointers:*

```
unsigned find_max(unsigned *arr, int size) {
    unsigned max = *arr;
    for (int i = 1; i < size; i++) {
        if (*(arr + i) > max) {
            max = *(arr + i);
        }
    }
    return max;
}
```

1. *Translate this function into ARM assembly.*
2. *Use a loop with indexed addressing mode to traverse the array.*

Five

Conditional Execution

Six

Addressing Memory

6.1 Load and Store

The ARM32/64 Microarchitecture uses a *load and store* memory access model. This means that before an item from memory can be used in the CPU, the *address* of the item must first be *load*ed into a *base register*. To write (*store*) a value back to memory, we must use the address contained in the base register.

This means obtaining the address of the first element of the variable you wish to access, placing this address in a base register, and computing the access locations of subsequent elements, as an offset of the base register.

Later we will discuss the ARM instructions required for this, but first we will consider a C program as the basis for understanding how memory operations take place.

What to know when addressing memory

We have to plan carefully in order to address memory correctly. Our approach needs to be aware of the following factors:

1. *Data Type*: There are two possible choices here. Addressing 32-bit words or addressing 8-bit bytes

    a) For accessing signed or unsiged integer data we read from memory using `ldr` (LoaD Register) and write to memory using `str` (StoRe Register)

    b) For accessing ASCII character data we read from memory using `ldrb` (LoaD Register Byte) and write to memory using `strb` (StoRe Register Byte)

2. *Addressing modes*:

    a) Offset addressing

    b) Post-index Addressing

    c) Pre-index Addressing

Offset, Post- and Pre- index Addressing will be discussed in Section 6.4, but first we will discuss the general syntax of the memory instructions.

Instruction	Meaning
`ldr ToReg, [FromAddr]`	Load the integer (word) found at address `FromAddr` into the register `ToReg`
`str FromReg, [ToAddr]`	Store the integer found in the register `FromReg` into the value at address `ToAddr`
`ldrb ToReg, [ FromAddr ]`	Load the byte from address `FromAddr` into the register `ToReg`
`strb FromReg, [ ToAddr ]`	Store the byte found in the register `FromReg` into the value at address `ToAddr`

Figure 6.1: Memory Instructions

Instruction Syntax

Let use briefly examine the syntax of the memory access instructions:

Let's consider an example from Table 6.1, we will use the load register (`ldr`)

```
ldr r1, [ r0 ]
```

This should be understood as *read the value from the memory location pointed to by* `r0`*, and place it into* `r1`.

6.2 C Program

We will use the C program shown in Fig. 6.2 to motivate our understanding of how this load and store model works.

1. Place the code into a file called `memory.C`
2. Compile it using: `gcc memory.C -o memory`
3. Run it: `./memory`

C Pointers

In the C code, we declared and initialized a *pointer* as `unsigned* p = array`. This means that `p` points to the address of the first element in `array`. We can now access any element of `array` using the notation `*p`. For example, the code:

```c
#include <stdio.h>

const int sz = 6;
unsigned array[ sz ] = { 9, 2, 3, 4, 8, 10 };
unsigned someValue = 55;

int main (int argc, char** argv) {
  unsigned* p = array;

  // update the 3rd element of the array
  p += 3;
  *p += 10;
  printf("the 3rd element is: %u \n", *p);

  // use p to access / modify the someValue;
  printf("someValue is %u \n", someValue);
  p = &someValue;
  *p = 123;
  printf("now someValue is %u\n", someValue);

  return 0;
}
```

Figure 6.2: Sample C program using pointers

```c
p += 3; // move the pointer to the 3rd index element
```

will move `p` to index 3 in the array of integers, thus it is now pointing to the element that contains the value 4. Notice that `p += 3` is shorthand for `p += (3*w)` where w is the word size of the hardware.

This value can be modified by using the pointer-access notation as follows:

```c
*p += 10; // add 10 to the value at the 3rd index
```

This will change the value to which `p` points, from 4 to 11.

6.3 ARM Assembly “Pointers”

In ARM Assembly, we see that C pointers have a direct counterpart. This is achieved by selecting some register as the *base-register*. The base-register

Address	Value
...	...
...	...
00 FF AB 00	00 00 00 09
00 FF AB 04	00 00 00 02
00 FF AB 08	00 00 00 03
00 FF AB 0C	00 00 00 04
00 FF AB 10	00 00 00 08
00 FF AB 14	00 00 00 0A
...	...
...	...

Figure 6.3: `array` in memory starting at address `00FFAB00`

is initialized to the name of the `.data` object, as we show later. This base register is the direct equivalent of our C program pointer.

To begin, assume that the ARM Assembly `.data` section contains the following code:

```
.data
  array: .word 9, 2, 3, 4, 8, 1
  someValue: .word 55
```

In the example, `array` starts at address `0x00FFAB00`

When this array is placed into memory at program load time, the elements will be stored in *contiguous* memory, with each element occupying 4 bytes (a `.word`) as shown in Fig. 6.3.

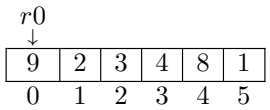
Initializing the base-register

We first load the address of the array using the following syntax (we will give `r0` the role of base register), using the instruction `ldr`. For clarity, we also show the C code equivalent to the assembly instructions:

```
// In C: unsigned* p = array;

// In ARM assembly:
ldr r0, =array
```

Note the use of the special character `=`. This character is used for *initializing* the base-register. This initialization does not need to be repeated (in most cases). As `array` starts at the memory location `00FFAB00`, `r0` now contains the value `00FFAB00`. We say that `r0` is *pointing to* the first element in the array:



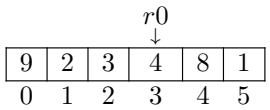
(6.1)

Pointing to the i-th element

Now we consider how to change the base-register so that it points to some arbitrary location (the i-th element).

```
// In C: p += 3;

// In ARM assembly 3 x 4 = 12:
add r0, #12
```



(6.2)

Notice that to change the base register so that it points to the i-th element (in this example $i = 3$), we must add 12 to it. This is because we need to calculate how many *bytes* to move `r0` on by. Because the word size of our platform is 32-bit, and there are 4 bytes in 32 bits, we multiply our new index (in the example it is 3) by 4 to get the next index position address when accessing integer arrays. This is in contrast to C, where we need to add 3.

In C, when the code is compiled, the addition is 12 *too, but this detail handled by the compiler, not the programmer.*

There are other ways to achieve base register pointer changes. For example:

```
// In ARM assembly 3 x 4 = 12:
mov r1, #4
mov r2, #3
mul r3, r1, r2
add r0, r3
```

6.4 Integer Addressing Modes

ARM Assembly has three different modes for accessing memory. You choose the correct mode depending on the situation. It is possible to mix and match them in a program, but this needs to be done with care so that you don't lose track of where the base-register is pointing to. These modes are now summarized in Table 6.4.

Mode	Syntax	Address	Base-register `r0`
Offset	`ldr r1, [r0, V]`	`r0+V`	Unchanged
Pre-Index	`ldr r1, [r0, V]!`	`r0+V`	Changes *before* the location is read
Post-index	`ldr r1, [r0], V`	`r0+V`	Changes *after* the location is read

Figure 6.4: Addressing modes with using `ldr` instruction and the base register `r0`. Note `V` is either (a) a register, (`ldr r1,[r0,r2]`), or (b) an immediate (`ldr r1,[ r0,#4]`)

Offset Addressing

Use this mode when we do not want to change our base-register. For example, in situations where we want to flexibly access any location in the memory object by the addition of some constant.

Suppose our algorithm requires us to set certain elements in our array in a particular order. Let's say the third, first and fourth elements in `array` should be set with some number in `r2`

```
// using offset addressing
ldr r0, =array
mov r2, #0
str r2, [ r0, #8 ]
str r2, [ r0, #0 ]
str r2, [ r0, #12 ]
```

In this sense we can "jump around" our array in any kind of random sequence, only knowing which element we need to access. Because the base-register does not change, there is no need to reset it after each "jump".

On the other hand, a plain iteration over `array` is troublesome using offset addressing. Consider:

```
// using offset addressing to iterate
// to zero out the array is a bit troublesome
ldr r0, =array
mov r1, #0 // incrementer
mov r2, #0 // value to write
mov r3, #6 // array size

loop:
  cmp r1, r3
  strlt r2, [ r0, r1, lsl #2 ]
  addlt r1, r1, #1
  blt loop
```

Here, we use `str r2, [ r0, r1, lsl #2 ]`, which has the effect of shifting left (multiplying by 4), the incrementer in `r1` during each iteration (which generates the sequence, $0, 4, 8, ...$). Obviously, this is extra work for the programmer to manage, so for plain $0, 1, 2, ..., n-1$ iteration tasks, we can use either pre-indexing or post-indexing. Let's consider post-index addressing first.

Post-index Addressing

As mentioned above, offset addressing is not a good choice for a simple iteration over each element of an array (for example, in linear search, or selection sort).

Generating the sequence $0, 4, 8, ..., (n-1) \times 4$ required an iterator and a `lsl` instruction

A much simpler approach, and the default approach used in C, is to increment the address stored in the base-register by some constant amount. In C, the notation `*ptr++` is used to dereference the memory address pointed to be `p` *and then* move `p` to the next element.

```
// in C, dereference the pointer
// and then move it on
unsigned r = *ptr++;
```

Assuming `ptr` is contains `00FFAB00`, then the unsigned integer `r` will be assigned 9, and *immediately after*, `ptr` will be incremented to `00FFAB04`.

Of course, `*ptr++` is just shorthand for:

```
unsigned r = *ptr;
ptr++;
```

This is called *post-index* addressing (or post-fix addressing in C). This should be our default method when iterating over an array in seqential order. In ARM assembly, we implement post-index addressing as follows:

```
// using post-index addressing to iterate sequentially
// over array and zero out each element
ldr r0, =array
mov r2, #0 // value to write
mov r3, #6 // array size

loop:
  cmp r1, r3
  strlt r2, [ r0 ], #4 // much neater syntax
  blt loop
```

Notice the simplification of the loop strucure: we have a much neater syntax for our `str` instuction:

Eliminating the menter (`r1`) and instruction from the (`addlt r1, r1, #1`)

```
// much neater syntax for iteration
strlt r2, [ r0 ], #4
```

versus

```
// more awkward syntax for iteration
strlt r2, [ r0, r1, lsl #2 ]
addlt r1, r1, #1
```

Post-Index Patterns

We now review some of the most common C programming post-index memory access patterns and their equivalent in ARM assembly.

Pattern 6.4.1 *Initialize-Increment*

In C we often see the following code pattern:

```
unsigned r = *ptr++;
```

*You should understand that this code has **two** steps. Step 1., store the value found at the address pointed to by **ptr**, into the variable `r`. Step 2. **then** increment **ptr** to point to the next element in the array.*

In ARM Assembly we have the exact same semantics:

```
ldr r2, [ r0 ], #4
```

*Step 1., load the value found at the address in `r0` into the register `r2`, and 2., **then** add Definition4 to the value in `r0`.*

Pattern 6.4.2 *Assign-Increment*

Another commonly encountered pattern is the assign increment pattern, for example:

```
*ptr++ = r;
```

becomes:

```
str r2, [ r0 ], #4
```

Pattern 6.4.3 *Read-Write-Increment*

The following pattern arises where, in one statement, the value from a pointer is read, modified then written. Following this, the pointer is incremented. As follows:

```
*ptr++ += 3;
```

This pattern must be implemented as ***three*** *instructions:*

```
ldr r2, [ r0 ] // load using offset addressing
add r2, r2, #3 // add our 3 as an immediate
str r2, [ r0 ], #4 // write results, increment pointer
```

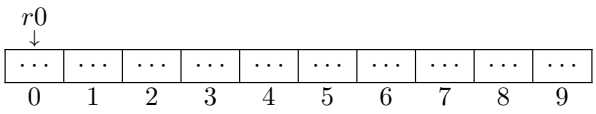
Of course, if you are writing a C program for the ARM platform, the compiler will convert `*ptr++ += 3;` *into the three ARM instructions as shown above.*

Pre-index Addressing

Post-index addressing has the interesting effect of leaving the base-register pointing to the next element in the array. This works well in cases where the next element in the array is *initialized*. However, it does not work well in cases when the next element is not initialized.

Consider a 10-element array of *uninitialized* $(\cdots)$ data:

```
ldr r0, =array
```

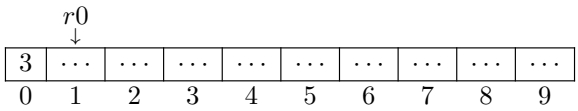


(6.3)

In Fig. 6.3, the base-register is pointing to the first uninitialized element. We can store a value into the first element using post-index addressing:

```
mov r1, #3
str r1, [ r0 ], #4
```

`r0` now points to index 1, but this value is not initialized.



(6.4)

So, what happens here?

```
ldr r1, [ r0 ]
```

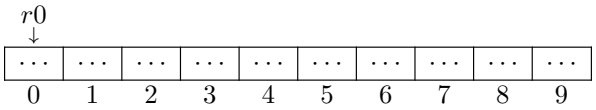
It is obvious that `ldr r1, [ r0 ]` leaves `r1` with garbage. This is not desirable. The base-register is pointing to uninitialized memory. Hence, the post-index addressing into uninitialized produces undesirable results. Of course, we could easily solve this using a work-around:

```
ldr r1, [ r0, #-4 ]
```

But this approach is awkward. We need another approach. Let's reset the base-register `r0`:

```
ldr r0, =array
```

so now things look like this:

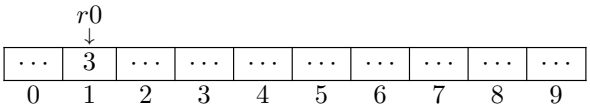


(6.5)

Now use pre-index as follows:

```
mov r1, #3
str r1, [ r0, #4 ]!
```

and the array now looks like:



(6.6)

Now a `ldr` from the base-registger will return a valid value:

```
ldr r1, [ r0 ] //places 3 into r1
```

Because the base-register was moved *before* the write, the subsequent read is guaranteed to be initialized. Of course, the effect here is that the first element is skipped!

But the good news is that `ldr r1, [ r0 ]` produces a properly initialized result every time. In fact, every pair of:

```
str r1, [ r0, #4 ]!
ldr r1, [ r0 ]
```

produces a *guaranteed* initialized value in `r1`, whereas, with

```
str r1, [ r0 ], #4
ldr r1, [ r0 ]
```

no such guarantee exists. In what circumstances might we want to use this pre-index addressing pattern? Use pre-index addressing when we want to move the base-register, write to an element and read that element back.

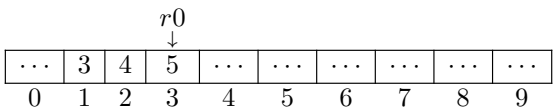
Stacks

Pre-index addressing is used to insert (or *push*) elements onto *stack*-type data structures. We discuss the details of stack-programming in Chapter 7. But we will briefly look ahead now.

Stacks are sometimes called LIFO (Last-In-First-Out) structures. The stack starts out empty and elements are added using pre-index addressing. Elements are removed (or *popped*) by using post-index addressing.

Consider this example:

```
ldr r0, =array
mov r1, #3
str r1, [ r0, #4 ]! // stack "push"
mov r1, #4
str r1, [ r0, #4 ]! // stack "push"
mov r1, #5
str r1, [ r0, #4 ]! // stack "push"
```

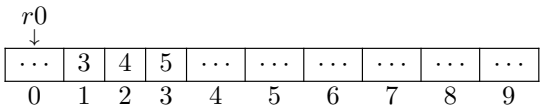


(6.7)

Now the stack has three elements, and the base-register is pointing to the *most recently added* element. We can now remove (*pop*) the elements as follows:

```
ldr r1, [ r0 ], #-4 // stack "pop"
ldr r1, [ r0 ], #-4 // stack "pop"
ldr r1, [ r0 ], #-4 // stack "pop"
```

Notice, this leaves our stack-like structure looking like this:



(6.8)

which is exactly the configuration we had in 6.3.

Notice that when `r0` returns to its initial value (at offset 0), three elements remain in the array at positions 1,2 and 3. Does this matter? No, it does not matter. All that matters is where `r0` is pointing to. Everything following `r0` is considered to be free space.

6.5 Character Addressing Modes

In 6.4 we discussed how integer memory can be accessed. Of course, integer data is not the only type of data we may want to use in our assembly language programs. A *string* type is an array of 1-byte alpha-numeric character elements. Consider the following C code fragment:

```
char *string = "Hello World!";
```

The ARM assembly equivalent declaration is:

```
.data
  string: .asciz "Hello World!"
```

Notice the declaration here: `.asciz`. This type is an array of ASCII characters (ie, a string), which may contain one or more elements. The `z` tells us that the string is terminated with a zero. This zero (the *null* character ASCII code 0) is automatically added to the end of our C and assembly strings during the compilation/assembly phase - we do not need to add it ourselves.

Iterating over strings

A common task is to iterate over a string looking for a particular value, or maybe looking for an upppercase or lowercase character. One problem we face is *how do we know when to stop iterating?*.

The appeareance of the null character signifies we have reached the end of the string in C:

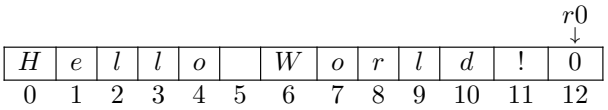
```
char string[] = "Hello World!";
char* cptr = string;
while(*cptr != 0) {
  // process the string
  // increment the pointer
  cptr++;
}
```

The ARM assembly equivalent is very similar - it must also test for the null character (ASCII 0). But because we are loading byte data and not integer data, we use the byte versions load and store instructions shown in Table 6.1:

```
ldrb r0, =string
ldrb r1, [ r0 ], #1 //load the first character
loop:
  cmp r1, #0
  ldrneb r1, [ r0 ], #1 //load the next character
  bne loop

loop_end:
  // the end of the loop
.data
  string: .asciz "Hello World!"
```

The base-register is incremented using post-fix addressing and iteration stops when the null character is encountered:



(6.9)

Let us look carefully at this instruction `ldrneb r1, [ r0 ], #1`. In ARM Assembly, the syntax `ldr<c>b` (where `<c>` is the condition `ne` not equal to) is referred to as a *Pre-UAL* syntax. This syntax has been superceded by UAL syntax, in which case the instructions is `ldrbne` (load register byte if not equal to). However, the pre-UAL syntax is the syntax used by *cpulator*, Hence this is the one we will use in these examples.

We should also note that in load or store byte instructions, the use of the immediate `#1` is correct because we are incrementing the base-register in steps of one byte (`#1`) and not one word (`#4`)

Zero-extending bytes

Looking again at this loop:

```
loop:
  cmp r1, #0
  ldrneb r1, [ r0 ], #1
  bne loop
```

of course, we can see that a single byte of data is loaded into `r1` each time `ldrneb r1, [ r0 ], #1` executes. As we know, all the registers in ARM-32 are 32-bits in size. This means that the byte of data stored in `r1` must be *zero-extended* to fit into the destination register. Consider how the ASCII value of 'A' ($65_{10} = 01000001_2$) would be handled:

$31-24$	$23-16$	$15-8$	$7-0$
00000000	00000000	00000000	01000001
3	2	1	0

(6.10)

As you can see, the 01000001_2 is right aligned into the LSB (byte 0), and the rest of the register is zero-filled to the MSB (bytes 1-3).

6.6 Chapter Summary

This chapter has introduced the process by which ARM assembly programs access the `.data` (static) segment of a program. We have shown the the programmer must carefully choose the correct data access pattern, as well as understand how memory is accessed depending on the particular data type being used. The two data types are integer (declared as `.word`), and alpha-numeric bytes (declared as `.asciz`).

You should develop an understanding of how the C programming pointer arithmetic model is implemented in ARM assembly. Developing a strong understanding of the similarities between C and assembly will make you a more well rounded software engineer, computer engineer, or general IT expert.

In section 6.4 we introduced the topic of *stacks*, and how we can understand these important data structers by studying how to add and remove elements using pre-index and post-index addressing respectively. In the next chapter we will discuss how stacks can be used to help with a variety of programming tasks that would otherwise be impossible without them.

6.7 Exercises

6.7.1 Basic Load and Store Operations

Exercise 6.7.1 *Write an ARM assembly program that does the following:*

1. *Declares an integer variable* `x` *initialized to* `25`.
2. *Loads* `x` *into a register.*
3. *Adds* `10` *to* `x`.
4. *Stores the result back in memory.*

Use `ldr` *and* `str` *instructions to perform memory operations.*

6.7.2 Byte vs Word Access

Exercise 6.7.2 *Given the following memory declaration:*

```
.data
  array: .word 0x12345678
```

Perform the following operations in ARM assembly:

1. *Load the full word into a register and print its value.*
2. *Load only the least significant byte using* `ldrb` *and print its value.*
3. *Store* `0xAB` *into the least significant byte of* `array` *without modifying other bytes.*

6.7.3 Offset Addressing

Exercise 6.7.3 *Given an integer array in memory:*

```
.data
  numbers: .word 5, 10, 15, 20
```

Write an ARM assembly program that:

1. *Loads the second element of the array into a register.*
2. *Adds* `5` *to the second element.*
3. *Stores the modified value back into the array.*

Use offset addressing mode in the `ldr` *and* `str` *instructions.*

6.7.4 Pointer Arithmetic

Exercise 6.7.4 *Convert the following C code into ARM assembly:*

```
unsigned values[5] = { 2, 4, 6, 8, 10 };
unsigned *p = values;
*(p + 2) = *(p + 2) + 5;  // Modify the third element
```

1. *Identify how memory addressing is done in both C and ARM assembly.*
2. *Use a base register and appropriate addressing mode to modify the third element.*

6.7.5 Pointer Traversal in Assembly

Exercise 6.7.5 *Write an ARM assembly program that:*

1. *Defines an array of 6 integers.*
2. *Uses a base register (pointer) to iterate through each element.*
3. *Doubles each value in the array.*

6.7.6 Finding Maximum in an Array

Exercise 6.7.6 *Write a C function that finds the maximum value in an array using pointers:*

```
unsigned find_max(unsigned *arr, int size) {
    unsigned max = *arr;
    for (int i = 1; i < size; i++) {
        if (*(arr + i) > max) {
            max = *(arr + i);
        }
    }
    return max;
}
```

1. *Translate this function into ARM assembly.*
2. *Use a loop with indexed addressing mode to traverse the array.*

Seven

Programming the Stack

7.1 Introduction

In the previous chapter we introduced the concept of a stack. The idea of a stack should be familiar to all those who have studied discrete data structures. The general definition of a stack is a data structure who's elements are added and removed following a last-in-first-out (LIFO) protocol.

To this general definition, we can add that some ARM specifics stack observations:

1. the stack is used to store *short-lived* word data
2. its primarily used to:
   - pass parameters from caller to callee label
   - save and restore general purpose registers
   - store label return address
   - store local variables
3. the programmer does not create the stack, the ARM microarchitecture provides access to the stack via the Stack Pointer register (`SP`) which in turn is managed by the operating system
4. by default, the stack grows from high to low address space (although it can be configured to grow from low to high too)
5. the stack is a finite size, and (in Linux) is limited to a maximum of 8192kb per thread
6. as the thread executes, the stack grows and shinks over time
7. the stack should never *leak* - any words allocated must eventually be deallocated

7.2 Sample C Code

In order to motivate our understanding, we will review a simple C code fragment that shows the key stack features:

```c
void caller() {
  int sum = callee(4,5,6);
}

// callee will return the sum
// of its three parameters
int callee(int a, int b, int c) {
  int tmp = 0;
  tmp = a + b + c;
  return tmp;
}
```

Although this is easy to implement in C, it is quite difficult to implement in ARM assembly. Let us look at the complexity involved.

1. `caller` prepares 3 parameters to share with the callee
2. `caller` invokes `callee`
3. `callee` retrieves the 3 parameters, one after the other
4. `callee` computes the sum of the three parameters and stores the result in `tmp`
5. `callee` prepares `tmp` to make it available to `caller`
6. `callee` returns to `caller`
7. `caller` uses the value returned from `callee`

As you can see, there is a lot going on here. None of the above steps would be possible without the stack. So now we will look at how to "port" this simple C program to ARM assembly by using the Stack Pointer (`sp`) register.

7.3 `caller` prepares parameters

In the example here, `caller` should prepare *three* integers for `callee`. It should do this by using the stack pointer and pre-index addressing to place the values onto the stack, one by one, each time, growing the stack down by one word (`sp, #-4`). *Note the order in which the caller places the aprameters on the stack, it does so from left to right when reading the function signature*

```
caller:
    mov r1, #4
    str r1, [ sp, #-4 ]!
    mov r1, #5
```

```
    str r1, [ sp, #-4 ]!
    mov r1, #6
    str r1, [ sp, #-4 ]!
```

In the above example, let us assume the stack starts at address `00FFFFAA` and below is the empty stack, showing 6 free 32-bit slots

`00 FF FF AA`	·····	← `SP`
`00 FF FF A6`	·····	
`00 FF FF A2`	·····	
`00 FF FF 9E`	·····	
`00 FF FF 9A`	·····	
`00 FF FF 96`	·····	

(7.1)

After `str r1, [ sp, #-4 ]!` has executed 3 times as above, the stack now looks like this:

`00 FF FF AA`	·····	
`00 FF FF A6`	4	
`00 FF FF A2`	5	
`00 FF FF 9E`	6	← `SP`
`00 FF FF 9A`	·····	
`00 FF FF 96`	·····	

(7.2)

As the stack is ready, `caller` invokes `callee` by using the branch-with-link instruction `bl`:

```
caller:
    mov r1, #4
    str r1, [ sp, #-4 ]!
    mov r1, #5
    str r1, [ sp, #-4 ]!
    mov r1, #6
    str r1, [ sp, #-4 ]!
    bl callee

callee:
    // how to access the 3 parameters?
```

7.4 Preserved and Unpreserved Registers

When transferring execution control to a label, the ARM standard defines two register classes you should be aware of. First we have the *Unpreserved* registers.

These registers are `r0-r4` inclusive. Unpreserved registers are not guaranteed to be saved and restored by the callee. Thus, the callee can directly overwrite `r0-r4` without concern for the consequenes for the caller. If the caller has some content in `r0-r4` that must not be lost, then the caller must push these registers on to the stack before transferring control to the callee.

The remaining registers (`r5-r14`) must be saved and restored by the callee. That is, if the callee uses any or all of the preserved set, the callee is responsible for saving and restoring their state by placing them onto the stack right after the link register, and restoring their state before transferring control back to the caller using `bx lr`.

7.5 `callee` reads the parameters

How should `callee` read the parameters from the stack? Of course, there are 3 choices: offset, pre-index, post-index addressing. Which one should be used? First, we can elimate pre-index addressing [1]. We are left with just offset or post-index addressing. Let's loook at using post-index addressing first.

Post-index stack addressing

Let's see how `callee` can use post-index addressing to read back the three parameters:

```
callee:
    ldr r4, [ sp ], #4
    ldr r5, [ sp ], #4
    ldr r6, [ sp ], #4
```

and the stack now looks like this:

`00 FF FF AA`	·····	← `SP`
`00 FF FF A6`	4	
`00 FF FF A2`	5	
`00 FF FF 9E`	6	
`00 FF FF 9A`	·····	
`00 FF FF 96`	·····	

(7.3)

Registers `r4`, `r5` and `r6` are being used to store the parameters to the label and the stack pointer `SP` has returned to the correct starting address, so there is no stack leak, which is good. However, notice the consequences of this approach are two-fold:

1. 3 registers are permanently allocated in the label for the parameters. This is wasteful, because we need to use registers sparingly. If the label had 10 parameters (of course, this is rather unlikely), then we would use 10 registers to store the 10 parameters, which of course is impossible.

[1] why is this?

2. We cannot go back and read the stack variables again, because `SP` has been moved back to the top, the 3 stack variables are now gone - they cannot be safely read again.

For the two reasons enumerated above, the *rarely* use post-index addressing for acessing the variables from the stack. Instead we use offset addressing as follows:

Offset stack addressing

Let's see how `callee` can use offset addressing to read back the three parameters:

```
callee:
    ldr r4, [ sp, #4 ]
    ldr r5, [ sp, #4 ]
    ldr r6, [ sp, #4 ]
```

and the stack now looks like this:

`00 FF FF AA`	·····	
`00 FF FF A6`	4	
`00 FF FF A2`	5	
`00 FF FF 9E`	6	← `SP`
`00 FF FF 9A`	·····	
`00 FF FF 96`	·····	

(7.4)

Registers `r4`, `r5` and `r6` are being used to store the parameters to the label and the stack pointer `SP` has returned to the correct starting address, so there is no stack leak, which is good. However, notice the consequences of this approach are two-fold:

1. 3 registers are permanently allocated in the label for the parameters. This is wasteful, because we need to use registers sparingly. If the label had 10 parameters (of course, this is rather unlikely), then we would use 10 registers to store the 10 parameters, which of course is impossible.

2. We cannot go back and read the stack variables again, because `SP` has been moved back to the top, the 3 stack variables are now gone - they cannot be safely read again.

For the two reasons enumerated above, the *rarely* use post-index addressing for acessing the variables from the stack. Instead we use offset addressing as follows:

```
main:
00FFFF00 mov r5, #5
00FFFF04 mov r6, #10
00FFFF08 b label2
00FFFF0C add r7, r5, r6

label2:
00FFFF10 mov r5, #11
00FFFF14 mov r6, #22
00FFFF18 b label3
00FFFF1C sub r7, r5, r6

label3:
00FFFF20 mov r5, #45
00FFFF24 mov r6, #2
00FFFF28 mul r7, r5, r6
```

Figure 7.1: Sample program with instruction addresses and label branching

7.6 `callee` saves `LR`

In Chapter 4 we discussed how execution flow can by controlled using the branching instruction `B`. We further refined this in Chapter 5 when we discussed conditional branching using instructions such as `ble`, `beq` etc.

When a branch instruction is executed. control jumps to the instruction address represented by the label name. In ARM32, there is no way to return from a label, and to continue execution from the instruction after the branch instruction. That is, there is no `return` keyword.

Orphaned instructions and `return`

Consider the example shown in Sample. 7.1 to motivate our understanding, with instruction memory addresses added on the left hand side.

By the time the instruction at address `00FFFF28` executes, there is no way for `label3` to return to address `00FFFF1C`, so this instruction can never be executed. In fact, the instructions at `00FFFF1C` and `00FFFF0C` are *orphaned*, and will never be executed. Clearly, this is *not* how high-level programming languages work. Therefore, there must be a way for our assembly language to handle the natural requirement of a `return`.

Branch with link `bl` and the link register `lr`

To solve the problem of *where* a label should return to, we replace the `b` instruction with `bl`. The latter means branch *with link*. This causes a special

register known as the link register (`lr`) to be updated with the address of the instruction following the `bl`, as shown:

```
main:
00FFFF00 mov r5, #5
00FFFF04 mov r6, #10
00FFFF08 bl label2
00FFFF0C add r7, r5, r6

label2:
00FFFF10 mov r1, #11
00FFFF14 mov r2, #22
00FFFF18 bl label3
00FFFF1C sub r3, r2, r1
00FFFF20 bx lr

label3:
00FFFF24 mov r1, #45
00FFFF28 mov r2, #2
00FFFF2C mul r3, r2, r1
00FFFF30 bx lr
```

Notice `bl` replaces `b` and `bx lr` is used to branch back to the address stored in `lr`. So we place `bx lr` at the end of our label.

Unfortunately, this does not quite work. Each time `bl` is used, it replaces the `lr` with the address of the instruction following. So although the instruction `bl label2` cause `lr` to be updated correctly, *the subsequent* `bl` instructions will overwrite it. The link register is assigned `00FFFF0C`, then it is assigned `00FFFF1C`. If a label calls a label, which calls a label etc. then *only the most recent* `lr` value is preserved, and all the others are lost. How should we solve this? By using the stack to store `lr` as the first thing a label does. We can rewrite the code as follows:

```
main:
00FFFF00 mov r1, #5
00FFFF04 mov r2, #10
00FFFF08 bl label2
00FFFF0C add r3, r2, r1

label2:
00FFFF10 str lr, [ sp, #-4 ]!
00FFFF14 mov r1, #11
00FFFF18 mov r2, #22
00FFFF1C bl label3
00FFFF20 sub r3, r2, r1
```

```
00FFFF24 ldr lr, [ sp ], #4
00FFFF28 bx lr

label3:
00FFFF2C str lr, [ sp, #-4 ]!
00FFFF30 mov r1, #45
00FFFF34 mov r2, #2
00FFFF38 mul r3, r2, r1
00FFFF3C ldr lr, [ sp ], #4
00FFFF40 bx lr
```

The above solution now properly handles the case of nested labels, and how to return from them using the stack to store temporary `lr` values. Notice the stack pointer is correctly reset at the end, and the callee label always saves the state of the `lr` as the *very first thing it does*

7.7 `callee` saves its working set of registers

When `callee` begins execution, it should, as mentioned in Section 7.6, save the `lr` on the stack as its *first* instruction. It should *then* save the set of working registers it will need in order to complete the label code *without clobbering the callers registers*. As mentioned previously, this is the task of saving the *preserved* registers.

This is a very important responsibility of `callee`. A callee that omits to save its working set of preserved registers will almost certainly clobber the registers of caller.

Consider again the code from Example 7.1:

```
main:
00FFFF00 mov r1, #5
00FFFF04 mov r2, #10
00FFFF08 b label2
00FFFF0C add r3, r2, r1

label2:
00FFFF0C mov r1, #11
00FFFF10 mov r2, #22
00FFFF14 b label3
00FFFF18 sub r3, r2, r1

label3:
00FFFF1C mov r1, #45
00FFFF20 mov r2, #2
00FFFF20 mul r3, r2, r1
```

Notice how, in each label, `callee` is overwriting the values that the `caller` has placed in the registers (`r1` and `r2`). Of course, the `caller` cannot predict how `callee` will use the registers.

the code may be written by different programmers, or the code for `caller` may not be available in source code format to the developer of `caller` s

The ARM best practice guide recommends that the caller should save any unpreserved registers it uses, and the callee is responsible for saving and restoring the preserved registers it will be using in the label code. For example, the code in `label2` overwrites `r1` and `r2` of `caller`, thus clobbering them. We see that the code in `label3` does exactly the same thing for its `caller`.

This kind of problem is easy to solve: `callee` should save its working set of registers on the stack before it begins, and restore the register set when it finishes:

```
main:
00FFFF00 mov r1, #5
00FFFF04 mov r2, #10
00FFFF08 b label2
00FFFF0C add r3, r2, r1

label2:
00FFFF20 str r1, [sp, #-4]
00FFFF24 str r2, [sp, #-4]
00FFFF28 str r3, [sp, #-4]
00FFFF2C mov r1, #11
00FFFF30 mov r2, #22
00FFFF34 b label3
00FFFF38 sub r3, r2, r1
00FFFF3C ldr r3, [sp], #4
00FFFF40 ldr r2, [sp], #4
00FFFF44 ldr r1, [sp], #4

label3:
00FFFF48 str r1, [sp, #-4]
00FFFF4C str r2, [sp, #-4]
00FFFF50 str r3, [sp, #-4]
00FFFF54 mov r1, #45
00FFFF58 mov r2, #2
00FFFF5C mul r3, r2, r1
00FFFF60 ldr r3, [sp], #4
00FFFF64 ldr r2, [sp], #4
00FFFF68 ldr r1, [sp], #4
```

Notice how the save/restore sequence follows the Last-In-First-Out (LIFO) semantics of the stack: save $[r1, r2, r3, ..., rn]$ then restore $[rn, ..., r2, r2, r1]$. Of course, you should only save and restore the set of registers you will be using in your label. There is no need to save and restore all of them.

7.8 Stack Overflow and Underflow

The stack is a critical part of the ARM architecture, and managing its space effectively is vital to prevent runtime errors such as **stack overflow** and **stack underflow**. These errors can disrupt the execution of a program and lead to unpredictable behavior.

7.9 Stack Overflow

A **stack overflow** occurs when more data is pushed onto the stack than the available space can handle. This situation typically arises in recursive functions or when a large amount of memory is allocated to the stack without considering its limitations. In ARM, since the stack grows downwards, exceeding its bounds can cause it to overwrite other important data, leading to crashes or undefined behavior.

Causes of Stack Overflow

The primary causes of stack overflow are:

- **Deep Recursion**: Recursive functions call themselves, and each call pushes its return address and local variables onto the stack. Without a base case or if recursion depth exceeds the stack capacity, the stack will overflow as more data is pushed onto it.
- **Large Local Variable Allocation**: Functions that allocate large arrays or variables on the stack may exceed the available stack space. If this happens, the stack will overflow.

In ARM assembly, the stack grows from high memory to low memory addresses. When the stack grows beyond its allocated size, the `SP` (stack pointer) may overwrite crucial data, leading to undefined behavior and program crashes.

Example of Stack Overflow in Recursive Function

A common scenario that leads to stack overflow is a recursive function that does not have a base case or has excessive recursion depth. Each recursive call pushes more data to the stack, and if the function calls itself indefinitely or too many times, the stack will overflow.

Example: A function that recursively calls itself without a base case:

```
recursive_call:
    recursive_call()   // Each recursive call pushes
   data to the stack.
```

In ARM assembly, this can look like:

```
recursive_call:
    push {lr}            // Save return address (Link
   Register)
    bl recursive_call     // Call the function
   recursively
    pop {lr}             // Restore return address
    bx lr                // Return from the function
```

In this example: 1. Each recursive call adds its return address and any local variables onto the stack using `push lr`. 2. Without a base case, the function continues calling itself, leading to the stack growing larger until it exceeds the allocated size, causing a stack overflow.

Effects of Stack Overflow

When a stack overflow occurs, several issues can arise:

- **Memory Corruption**: The stack is supposed to store critical data such as return addresses, local variables, and saved registers. If the stack overflows, it overwrites this data, leading to corrupted values and unpredictable behavior.
- **Program Crashes or Undefined Behavior**: As the program may attempt to execute corrupted return addresses or access invalid memory locations, it can result in crashes or undefined behavior. This can cause the program to behave in unexpected ways or terminate unexpectedly.
- **Security Vulnerabilities**: Stack overflows are a common vulnerability exploited in buffer overflow attacks. An attacker may intentionally overflow the stack to inject malicious code and gain control over the program.

How to Prevent Stack Overflow

There are several strategies to prevent stack overflow in ARM assembly programming:

1. Limit Recursion Depth

Ensure that recursive functions have a proper base case to terminate the recursion. Avoid infinite recursion, and limit the recursion depth to prevent excessive stack growth.

Example: Recursive function with a base case:

```
recursive_call:
    cmp r0, #0           // Compare argument with 0
    beq end_recursion    // If base case reached,
   return
    push {lr}            // Save return address
```

```
    sub r0, r0, #1      // Decrement counter by 1
    bl recursive_call   // Recursive call
end_recursion:
    pop {lr}            // Restore return address
    bx lr               // Return from the function
```

In this example, the recursion halts once the argument `r0` reaches zero, preventing infinite recursion and stack overflow.

2. Optimize Stack Usage

Minimize stack usage by avoiding large local variables. Instead of allocating large arrays on the stack, allocate them dynamically in the heap.

Example: Using dynamic memory allocation instead of stack allocation:

```
// Avoid allocating large arrays on the stack:
sub sp, sp, #1000  // Allocating 1000 bytes on the
   stack
// Instead, use dynamic memory allocation on the heap
   to avoid stack overflow.
// Example in C: malloc() for large arrays
```

3. Use Tail Recursion

Tail recursion allows a function to reuse its current stack frame for recursive calls, thus avoiding the growth of the stack. A tail-recursive function does not require additional stack space for each recursive call, which helps prevent stack overflow.

Example of Tail Recursion:

```
tail_recursive:
    cmp r0, #0          // Check if base case
    beq end_tail_recursion
    sub r0, r0, #1      // Update argument
    bl tail_recursive   // Tail call (no extra stack
   frame needed)
end_tail_recursion:
    bx lr               // Return from function
```

In tail recursion, the function does not create a new stack frame for each recursive call, thus preventing stack overflow.

4. Use the Stack Wisely

Avoid unnecessary usage of the stack. Only push registers to the stack when they are necessary to preserve, and use registers for temporary variables when possible.

Example: Pushing only necessary registers to the stack:

```
    // Avoid pushing unnecessary registers to the
   stack
    push {r4, r5}        // Only push necessary
   registers
    // Function logic
    pop {r4, r5}         // Restore only the necessary
     registers
```

By limiting the number of pushed registers, the stack usage is minimized, reducing the risk of overflow.

5. Monitor Stack Usage

In embedded systems, monitoring the stack usage through system logs or dedicated tools can help detect potential stack overflows before they happen. A stack guard mechanism can be implemented to check for overflow conditions and raise exceptions if the stack is nearing its limit.

Conclusion on Stack Overflow

Stack overflow is a critical issue in ARM assembly and other low-level programming environments. It is commonly caused by excessive recursion or large allocations on the stack. By limiting recursion depth, using the stack wisely, and employing techniques like tail recursion, you can minimize the risk of stack overflow and ensure that your program executes efficiently and securely.

7.10 The Stack and Function Prologues/Epilogues

Each function typically includes a **prologue** and an **epilogue** that manage the stack, save and restore registers, and adjust the stack pointer for local variable allocation.

7.11 Prologue and Epilogue

The **prologue** and **epilogue** are essential for managing the stack during function calls in ARM assembly. These two sequences of instructions handle saving and restoring registers, allocating and deallocating space for local variables, and ensuring that control is properly returned to the caller.

Prologue: Setting Up the Stack Frame

The **prologue** is the first part of a function, responsible for setting up the stack frame. The stack frame stores the function's local variables, return address, and callee-saved registers. The prologue is crucial for establishing the function's execution environment.

- **Saving the `LR`**: The link register (`LR`) holds the return address, which tells the processor where to resume execution after the function finishes. It must be saved on the stack to ensure the function can return to the correct location.
- **Saving Callee-Saved Registers**: The registers `r4`–`r11` are callee-saved, meaning the callee is responsible for saving and restoring these registers if it uses them. This prevents the callee from inadvertently modifying the caller's state.
- **Adjusting the Stack Pointer (`SP`)**: The `SP` is adjusted to allocate space for local variables. The amount of space depends on how many local variables the function needs to store.

Epilogue: Returning Control to the Caller

The **epilogue** is executed at the end of a function to clean up the stack and return control to the caller. It undoes the actions performed by the prologue and ensures that the function call does not leave the stack in an inconsistent state.

- **Restoring Callee-Saved Registers**: The registers saved in the prologue are restored to their original values. This ensures that the caller's state is preserved across function calls.
- **Deallocating Space for Local Variables**: The `SP` is adjusted to release the space allocated for local variables. This ensures the stack is returned to its previous state.
- **Restoring the `LR` and Returning**: The return address stored in `LR` is restored, and the program jumps back to the caller using the `bx lr` instruction.

Example of Prologue and Epilogue

Here is a simple example of how the prologue and epilogue are implemented in ARM assembly:

```
    // Function Prologue
function_prologue:
    push {lr}            // Save return address (Link
   Register)
    push {r4-r7}         // Save callee-saved
   registers
```

```
    sub sp, sp, #16      // Allocate space for 16
   bytes of local variables

    // Function Body (do some work here)

    // Function Epilogue
function_epilogue:
    add sp, sp, #16      // Deallocate 16 bytes of
   local variable space
    pop {r4-r7}          // Restore callee-saved
   registers
    pop {lr}             // Restore the return address
    (Link Register)
    bx lr                // Return to the caller
```

Why Prologue and Epilogue are Important

The prologue and epilogue are vital for the following reasons:

- **State Preservation**: They ensure that the caller's state is preserved, preventing functions from inadvertently modifying registers or local variables that the caller relies on.
- **Memory Management**: They manage the stack frame, allocating space for local variables and cleaning up afterward.
- **Control Flow**: The epilogue ensures that control is returned to the correct point in the program after a function call, using the return address stored in `LR`.

Summary

The prologue and epilogue work together to ensure that a function call in ARM assembly is executed correctly. The prologue sets up the stack frame, saves necessary registers, and allocates space for local variables. The epilogue restores the registers, deallocates space, and ensures that the function returns control to the correct address. Properly managing the prologue and epilogue helps maintain a consistent execution environment and prevents errors during program execution.

7.12 Register Allocation and Stack Management

Efficient register allocation and stack management are crucial for optimizing memory usage and ensuring that functions work correctly. This is particularly important when dealing with a limited number of registers, as in ARM assembly. Since ARM processors have a smaller number of general-purpose registers

compared to other architectures, effective usage of these registers is vital to maintain high performance and avoid excessive memory access overhead.

Proper management of registers allows for faster execution, since accessing registers is much quicker than accessing memory. Stack management also ensures that functions preserve the state of the program and that memory is used efficiently during function calls.

General-Purpose Registers

ARM provides 16 general-purpose registers, `r0` to `r15`, which are used for various tasks during function calls and computations. Here is a breakdown of each register's role:

- `r0` to `r3`: These registers are primarily used for passing the first four arguments to functions. These registers are caller-saved, meaning that the caller function is responsible for saving their contents if it wants to preserve their values across function calls.
- `r4` to `r11`: These are callee-saved registers. If a callee function uses these registers, it must save their previous values at the start of the function and restore them before returning. This ensures that the caller's data is preserved, and registers are not inadvertently overwritten during function calls.
- `r12` (also known as `ip`, the intra-procedure-call scratch register): This register is used for temporary storage by either the caller or the callee. It is not guaranteed to be preserved across function calls, so it should be used with caution.
- `r13` (`SP`): This is the stack pointer register, which points to the current top of the stack. It is automatically updated as data is pushed or popped from the stack. The stack pointer must be managed carefully, as it dictates the function call return addresses and local variables.
- `r14` (`LR`): The link register holds the return address for function calls. It is updated by the `bl` (branch with link) instruction, and it is used to return control to the caller. However, when functions call other functions, the `LR` may be overwritten, which is why it must be saved when necessary.
- `r15` (`PC`): The program counter holds the address of the next instruction to be executed. It is automatically updated by the processor during program execution.

The general-purpose registers in ARM are a limited resource, so using them efficiently can significantly impact program performance. Minimizing the use of the stack for temporary variables, when possible, helps reduce the overhead associated with stack management.

Callee-Saved and Caller-Saved Registers

ARM has a well-defined convention for how registers are used during function calls. The convention divides registers into two categories: **callee-saved** and **caller-saved** registers. Understanding the roles of these registers and managing them properly is crucial for ensuring efficient program execution.

Callee-Saved Registers

Callee-saved registers (`r4`–`r11`) are those that must be preserved by the called function (callee). The callee is responsible for saving the value of these registers at the beginning of the function and restoring them before returning. This ensures that the caller's data is not inadvertently altered during function execution.

In ARM assembly, the callee-saved registers are used when a function needs to use temporary data but must preserve the state of registers that the caller might need after the function call. For instance, if a function modifies `r4`, it must save its original value before making changes and restore it before returning to the caller.

7.13 Exercises on Programming the Stack

Exercise 1: Stack Management in Recursive Functions

Write a recursive function in ARM assembly to calculate the factorial of a number. Implement the function without a base case and observe how the stack grows, leading to a stack overflow. Then, modify the function to include a base case and explain how adding the base case prevents the overflow.

Instructions:

- Write the recursive function to compute the factorial without a base case.
- Simulate how the stack overflows when the recursion depth exceeds the available space.
- Modify the function to include a base case and explain how it fixes the overflow.

Exercise 2: Understanding Caller-Callee Stack Interaction

Given the C code below:

```c
void caller() {
  int sum = callee(4,5,6);
}
int callee(int a, int b, int c) {
  int tmp = 0;
  tmp = a + b + c;
  return tmp;
}
```

Task: Translate this C code to ARM assembly by using the stack to pass parameters from the caller to the callee, store local variables, and manage the return address.

Instructions:

- Implement the function ‘caller‘ and ‘callee‘ in ARM assembly, passing parameters through the stack.
- Use the ‘SP‘ (Stack Pointer) and ‘LR‘ (Link Register) properly to manage the stack and return address.

Exercise 3: Stack Overflow Prevention

Write a function in ARM assembly that uses a loop to sum integers from 1 to n. Compare the iterative solution with a recursive one and analyze how stack overflow could occur in the recursive version.

Instructions:

- Implement both iterative and recursive versions of the summing function.
- Simulate how the recursive function could lead to a stack overflow if ‘n‘ is too large.
- Modify the recursive version to use tail recursion and explain how it reduces stack usage.

Exercise 4: Prologue and Epilogue Function Design

Write an ARM assembly function that uses a prologue to save registers and allocate space for local variables, and an epilogue to restore registers and deallocate the space.

Instructions:

- Implement a function that saves and restores registers (‘r4‘ to ‘r7‘) as part of the prologue and epilogue.
- Allocate space for local variables in the prologue and deallocate them in the epilogue.
- Ensure the function returns control to the caller correctly using the ‘bx lr‘ instruction.

Exercise 5: Exploring Stack Underflow

Write a program that demonstrates stack underflow in ARM assembly. Underflow occurs when the stack pointer is adjusted incorrectly, and the program attempts to pop more data than was pushed onto the stack.

Instructions:

- Write a function that incorrectly manipulates the stack pointer by popping data without pushing it first.

- Simulate stack underflow by observing incorrect data restoration, or by analyzing how crashes occur when the stack is manipulated incorrectly.
- Explain how correct stack management can prevent stack underflow.

Exercise 6: Stack Size Limitation and Stack Guard

Implement a simple ARM assembly program that simulates exceeding the stack size limit. Show how a guard mechanism (e.g., checking if the stack pointer exceeds a set limit) can prevent stack overflow.

Instructions:

- Write a function that simulates allocating large data on the stack and exceeding the stack's size limit.
- Implement a stack guard that checks if the stack pointer exceeds the set limit and prevents overflow.
- Demonstrate how the guard mechanism can safely terminate the program when a potential overflow is detected.

Eight

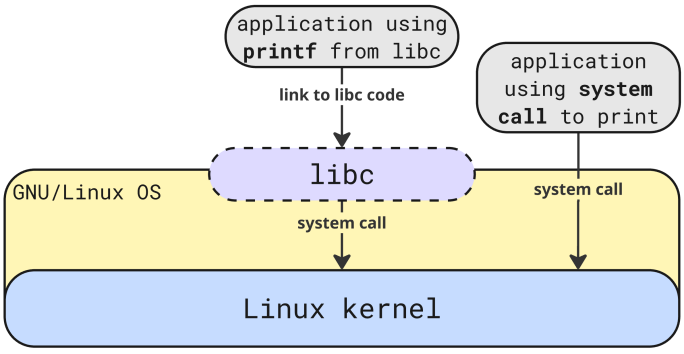
Integrating libc

8.1 Linux Systm Calls

In the previous chapters we learned the basics of ARM32 assembly techniques and guidelines. The code we looked at was limited in terms of its *functionality* - it was limited to logical, mathematical and flow control instructions. The code we looked at did not require any services from the operating system.

These services are known as *system calls*, and a typical application will use many system calls in order to achieve its objectives. Indeed, an application program that does not use system calls can not achieve anything useful.

There are two ways for an assemble program to access the operating system services (1) by using System Calls Directly from the assembly, and (2) linking to a library of functions that provide an abstraction layer above the system call level. Such libraries are very useful because it allows us to reuse high-level abstractions rather having to build the scaffolding code around the service call. The difference is summarized in the figure below.



Directly using system calls from an application process is a difficult task. it requires the developer to execute a software interrept (`svc #0`) instruction, along with populating specific registers correctly. For example, the system call to print `"Hello World\n"` to the screen shown in Fig 8.1:

Using system calls for I/O, process management, memory allocation etc. is perfectly possible, but requires us to reinvent functionality that has already been solved and implemented elsewehere. As showin in Fig. 8.1, we will have

```
print:
    mov r0, #1      // r0 gets 1=stdout
    ldr r1, =string // r1 gets the string address
    mov r2, #12     // r2 is the string length
    mov r7, #4      // r7 gets 4=print system call
    svc #0
.data
  string: .asciz "Hello World\n"
```

Figure 8.1: Printing to the screen using the system call instruction `svc`

```
int main(int argc, char* argv[]) {
  printf("Command line params: num params: %d, \
    program name %s, \
    first param: %s\n", \
    argc, argv[0], argv[1]);
}
```

Figure 8.2: Printing to the screen using `printf` from `libc`

difficulty creating a `printf` type function - one that supports parameter substitution within the string. For example, consider this simple C program:

As we can see clearly from Fig. 8.2, `printf` supprts parameter substitution by way of the `%s, %d, ...` modifiers. How could we achieve this functionality using the `svc` instruction? With great difficulty!

In summary, although all the kernel functions for I/O, memory management, process management etc are accessible via system calls, we tend to avoid using this method because it requires us to build complex scaffolding code that adds to the overall complexity and management of the assembly application. Therefore, we leave it up to you to build your own `printf` function using service calls only.

8.1.1 Linux libc

Most application processes do not invoke system calls directly, but rather invoke the C runtime library `libc`

Nine

Machine Code

Ten

Single Cycle Microarchitecture

Eleven

Multi Cycle Microarchitecture

Twelve

Pipeline Microarchitecture

Thirteen

CPU Caches

Fourteen

Memory and Virtual Memory

Fifteen

Parallel Architectures