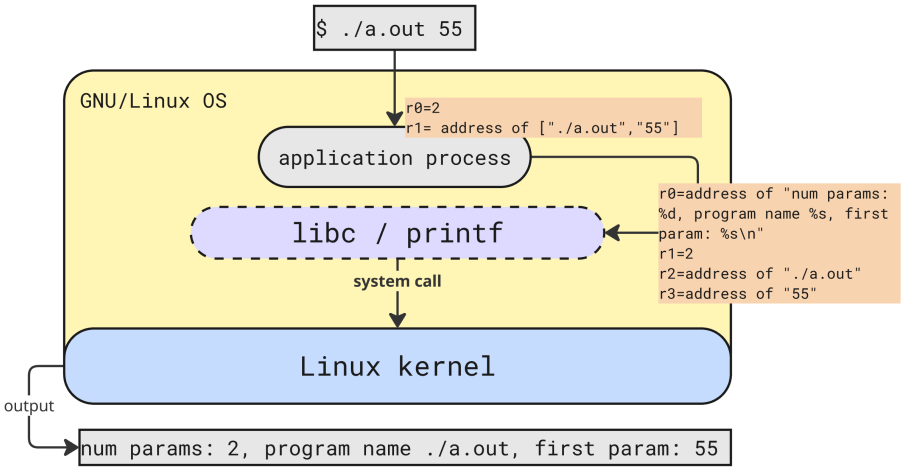
An Introduction to

Computer Architecture



RISC ARM32/64

Suitable for both Graduate and Undergraduate Courses

John Burns and Sardar Ziyatkhanov

Contents

Contents

One

Introduction

1.1 Historical Context

The evolution of CPU architectures over the years since the 1970s has taken two distinct pathways. The first pathway, and the one taken by Intel with their x86, [1] was to squeeze more and more [circuitry] onto the silicon fabric of the CPU. Starting from around 29000 transistors in 1978, the Arrow Lake variant (launched in 2024) has over 4 billion transistors today [Com25].

The astonishing growth of transistor density is summarized in Fig. 1.1. As you can see from this log-linear graph, growth in transistor density followed the Moore' Law doubling model quite closely, only beginning to slow then plateau in from 2008 onwards.

In addition to the growth in transistor density, the instruction set of the x86 in 1978 had about 100 instructions, whereas today, this number is closer to 1600.

[1] first introduced in 1978

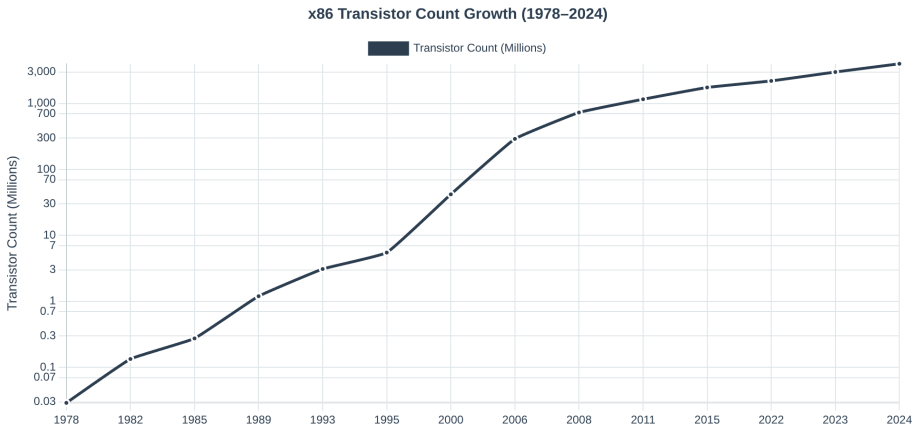


Figure 1.1: The growth of the number of transistors in the x86 architecture from 1978 to the present time. This is a log scale with the y-axis showing a measurement in millions.

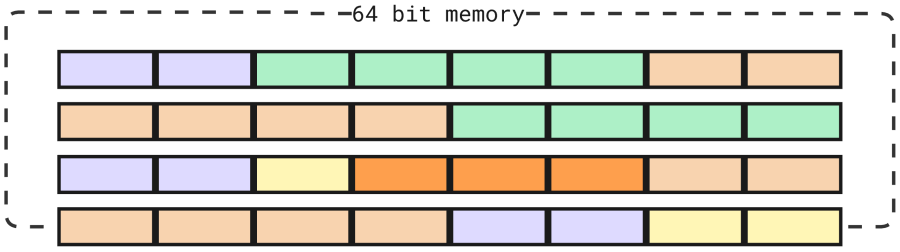


Figure 1.2: Variable length instructions in 64-bit x86 memory. Illustrating 2-byte (purple), 4-byte instructions (green), 6-byte in orange and 1-byte (yellow).

In addition, clock speeds have increased by a factor of 600 - from 5 MHz to 6 GHz today. Finally pipeline depth: has gone from 4 to about 30 stages in 2024. Pipeline microarchitectures will be discussed in more detail in Chapter 12.

As you can see, the Intel x86 trend has been to continuously pack more and more complex circuitry onto the CPU silicon.

1.2 Complex Instruction Set Computers

The Intel x86 as mentioned in the previous section is an example of a *CISC* (Complex Instruction Set Computing) architecture This architecture uses a large set of *complex* instructions, each of which is capable of performing multiple operations in a single instruction. In other words, a CISC instruction is often the *encapsulation* of a set of lower-level operations.

CISC instructions typically offer the programmer a higher level of *abstraction* than, as we shall see, the RISC (Reduced Instruction Set Architecture) discussed next. The Intel x86 Instruction Set Architecture (ISA) supports *variable length* instructions. This means that instructions can occupy less than the full word size of the CPU. Variable length instructions are a space saving solution - reducing the overall footprint of the executable and thereby reducing the amount of memory required to store the process cide. X86 instruction lengths can vary in the range 1 to 15 bytes. This means that the number of bytes used to encode a single instruction can differ, unlike some other architectures that use fixed-length instructions (eg, RISC ISAs)

So instead of padding all instructions out to the same length, variable length instructions allow for more efficient use of main memory.

We present an illustrative model of variable length memory in Fig .1.2. in 64-bit x86 memory. Here, 2-byte instructions in purple, 4-byte instructions in green, 6-byte in orange and one byte in yellow.

1.3 Reduced Instruction Set Computers

CISC Instruction Set Architectures (ISAs) were, until relatively recently, the dominant consumer CPU platform. Excluding desktop Apple Mac systems

(which used the PowerPC), Windows desktop computers used Intel x86 exclusively. For most end-users, the CISC capabilities of the ISA were an ever-increasing advantage. It made the standard desktop PC more and more powerful and capable, and allowed software engineers to design and build ever more CPU demanding applications (CAD/CAM, animation, modelling and simulation, numerical analysis and so on).

However, as the x86 became the dominant desktop solution, there were plenty of alternative CPU architectures available. These architectures (for example, Apple PowerPC, IBM RS/6000) were notable for one major difference to CISC: they were designed with a *reduced* instruction set (RISC) architecture. For example, the Apple Mac Powerbook G5 used a RISC CPU (from IBM) had only around 500 instructions in the totality of the ISA.

Why might a computer architecture be designed with a deliberately smaller instruction set size than a comparable Intel CPU? There are several reasons for this approach, but there are 3 keys ones:

1. **Energy:** RISC CPUs consume less energy because the architecture has much less circuitry

2. **Speed:** RISC instructions *generally* take fewer CPU cycles to execute than CISC instructions because each instruction has less overhead as is considerably simpler in its circuitry implementation.

3. **Complexity:** RISC instructions are always the same size (the word size of the CPU - 32/bit or 64/bit). Because there are no variable length instructions in CISC, there is no need for the complex circuitry to deal with this.

The fixed length instruction size model is shown in Fig. 1.3. In this example all instructions (purple) are exactly 64-bits in length, with any unused space padded (gray) out. The same logic applies to RISC 32-bit ISAs (although to a lesser extent).

Not only is the ISA reduced in terms of the number of instructions, but we may say that each of these instructions is simpler (this being a relative term of course) to design and implement. Indeed, early versions of the ARM RISC did not contain native CPU support for floating point numbers. It was not until 2001 that on-CPU support for floating point numbers were added to the ARMv5TE architecture.

The Mobile Revolution

Of course, we are all familiar with the phenomenal growth in adoptation of mobile devices. From the early 1990s models (typically running custom OS code), to the complex and capable Android and Apple devices of today. There are now only *two* mobile OS standards - Android and iOS. In parallel to the rise of mobile devices, a new RISC mobile architecture emerged from a company called Advanced RISC Machines (ARM).

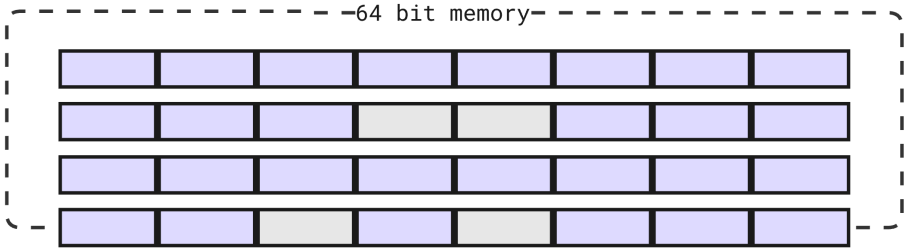


Figure 1.3: RISC fixed length instructions in 64-bit x86 memory. All instructions (purple) are exactly 64-bits in length, with any unused space padded (gray). The same logic applies to RISC 32-bit ISAs (although to a lesser extent)

Although ARM was around even in the 1980s, the release of ARM7TDMI 32-bit RISC architecture was an enabling factor for OEMs to launch complex mobile architectures. ARM7TDMI had only 200 unique instructions. This feature alone gave ARM architectures an unrivalled competitive advantage over Intel in the mobile market.

ARM CPUs consumed far less energy than CISC cpus, requiring very little cooling hardware within the device, and enabling battery powered devices to run complex operating systems such as Android (with the linux kernel). Today, ARMv9-A still has only 500–600 instructions, a substantial difference to Intel, and one that ensures that ARM continues to dominate both the mobile market, and the productivity laptop market seqment (ARM powers Mac hardware).

1.4 This book

Approach

This book introduces the reader to Computer Architecture. Any book that covers this subject matter needs to decide if it will focus on CISC or RISC ISAs. We have decided to use RISC in the chapters dedicated to assembly programming (Chapters 5 through to Chapter 8), and in the chapters dedicated to microarchitectures (Chapters 10-11).

There are several reasons for this decision, viz:-

1. ARM32 assembly, in particular, is a straightforward language and one that is easy to understand.

2. Both ARM32 and ARM64 are *load and store* architectures (more on that later). So the instructions are WYSIWYG [2], unlike intel x86, where a single instruction can mask a complex implementation.

[2] WYSIWYG - *what you see is what you get*

3. ARM is making ever greater inroads into the data centre. For example, AWS reported in December 2024 that more than 50% of its new CPU capacity added in the previous two years was ARM based. Percentages form the other cloud providers is likely to be similar.

As energy consumption continues to grow in significance for data centre operators, the role of RISC based systems, and ARM64 in particular, is becoming more and more significant.

In the assembly language chapters that follow, we will assume the reader is using either `https://cpulator.01xz.net/?sys=arm` (for an in-browser, ARM32 emulator with limited functionality), or they have installed `qemu-sysem` emulator for a complete and accurate emulation of ARM32/ARM64 CPUs. Of course, we do not assume the reader has an Apple Mac (Air or Powerbook). But if you do use Apple, then you do not need an emulator, all you need is a text editor and compiler (`gcc`) will do fine.

Code Repository

All the code samples used in this book, along with scripts, READMEs and documentation, can be freely cloned from the following resource:
`https://github.com/jzburns/csci-comp-arch.git`

CPUlator

We recommend two strategies for those starting to learn assembly language programming for the first time. The first resource is called *CPUlator*. CPUlator is a standard web brower *simulator* for the M7/ARM32 ISA. It is an excellent introduction to assembly programming because it has a decent UI that allows you to step through code, add breakpoints, step into and out of label code etc. CPUlator also allows the programmer to view and modify registers, the flag register, stack pointer, and program counter registers, as well as viewing memory during and after program execution.

There are many useful resources to help the new programmer get started with CPUlator. One youtube channel we recommend is called *LaurieWired*. Laurie has an excellent ARM32 video playlist we recommend you work through these carefully in order to better understand CPUlator.

You will find the link to *LaurieWired* along with some other useful web resources in the `README.md` file in the `csci-comp-arch` repository mentioned already.

Please note, although CPUlator provides an excellent emulation learning environment, by its nature, it cannot fully emulate system calls (and does not support any references to `libc`). Therefore, once you develop your competence in ARM assembly programming, you will need to compile and run actual binaries of your own. This is not possible in CPUlator. For this, we need either a true ARM platform (such as Raspberry Pi 3/4/5, Apple Macbook or Air), or a complete system emulator that allows you to boot into full hardware emulation. For this we recommend using `qemu-system`, which we discuss next.

`qemu-system`

Finally, let us conside a situation where we want to develop software for an architecture different to the one we are working at. For example, consider a situation where a developer is building software for a ARM64/ARM7TDMI Raspberry Pi, but works on a Linux x86 laptop. How can this be achieved? In this situation, virtual machines are not useful as they cannot execute software that is compiled for a different CPU architecture.

In this situation, there are only two solutions:

- Provide every developer with their own Raspberry Pi hardware. But what do we do when we are building OS-level software for expensive mobile platforms? We cannot give every developer their own expensive mobile device for testing.
- Instead, we need to use a software layer that acts as an *emulator* which fully reproduces the platform we wish to develop on. This includes kernel, file system, binary tool chains and so on. This is the purpose of `qemu`.

As can be seen in Fig. 1.4, we use `qemu-system-aarch64` to run a Debian Linux (kernel 5.12) Emulating ARM7DTI/2 core on top of the host running Arch Linux 6.14 (kernel) on a 4-core CPU x86 ISA. This combination of different kernels, file systems and ISAs is not possible in containerization or virtualization.

`qemu-system-aarch64` example

We conclude this section with a concrete example of how to use `qemu-system-aarch64`. Firstly, note that `qemu-system-*` is an full system emulator for a specific ISA. In this case, `aarch64` which is the ARM64 architecture. There are many useful `qemu` resources for you to study - but start at `https://www.qemu.org/` for an excellent overview of the tools and tool chains available.

Consider the following `qemu-sysem-aarch64` command:

```
qemu-system-aarch64
  -machine raspi3b
  -cpu cortex-a72
  -smp 4 -m 1G
  -kernel kernel.img
  -dtb treeblob.dtb
  -drive "file=bullseye.img,format=raw,index=0,if=sd"
  -append "rw earlyprintk loglevel=8 console=ttyAMA0
   ,115200 dwc_otg.lpm_enable=0 root=/dev/mmcblk0p2
   rootdelay=1"
  -device usb-net,netdev=net0 -netdev user,id=net0,
   hostfwd=tcp::5555-:22
  -nographic
```

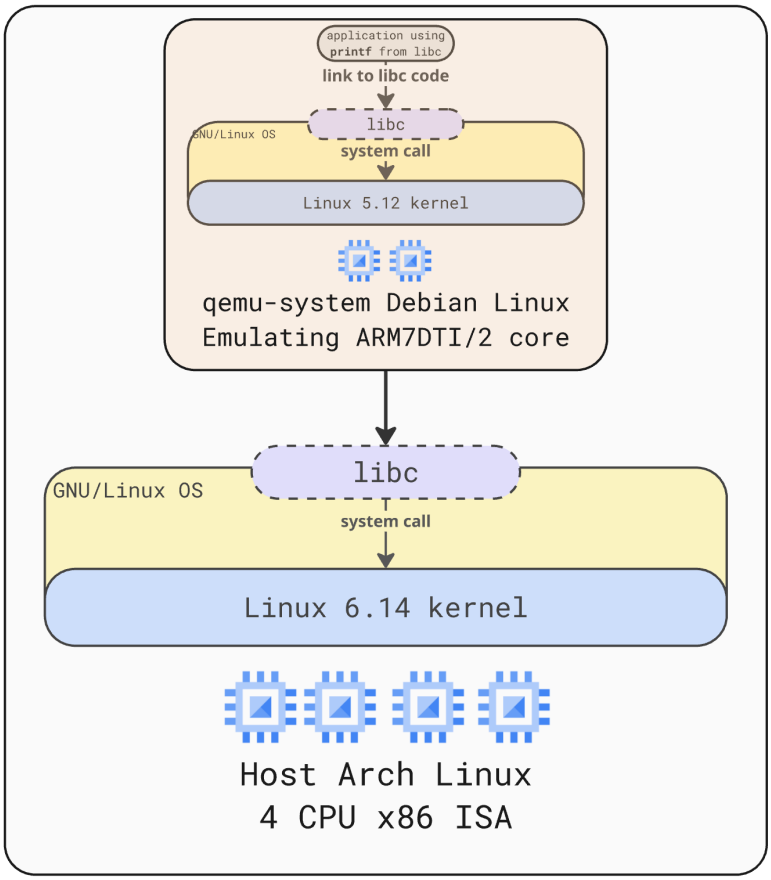


Figure 1.4: Using `qemu-system-aarch64` to run a Debian Linux (kernel 5.12) Emulating ARM7DTI/2 core on top of the host running Arch Linux 6.14 (kernel) on a 4-core CPU x86 ISA

When we execute this `qemu-system-aarch64` instruction on our host (and assuming `kernel.img`, `treeblob.dtb` and [Debian] `bullseye.img` are available locally), then we will shortly start a Raspberry Pi 3B ARM32 cortex-a72 with the kernel and filesystem of our choice. You can find this script in the book github repository (located in `qemu/start.sh`).

We can then use `ssh` to connect to `localhost` on port 5555 and our instance is ready to begin using for our project developement!

1.5 Chapter Summary

In this chapter we have introduced the reader to CISC versus RISC ISAs. We discussed the design principles that motivate the developement of CISC ISAs:

that ISA complexity and circuitry complexity outweigh energy consumption performance. Conversely, we noted that RISC ISAs are relatively less complex and significantly less energy demanding.

We do not need to consider CISC ISAs any further as the remainder of this book will look exclusively at either ARM32 or ARM64 systems in general.

You are advised to develop a strong understanding of these two architectures and the advantages and disadvantages of each. In this chapter we also introduced you to the source code resources for you to use, along with some suitable emulation platforms. In the early chapters of this book, `CPUlator` is ideal because it gives us a nice developement and debugging experience inside a standard web browser. But as we develop our program complexity, we will start using `qemu-sysem` in order to use this full set of `libc` tools present in the Linux OS of our choice.

In the following chapter we will discuss the internal structure of an executable program. It is important to understand this structure well, because when we begin assembly language programming the structures of our executable will be explicitly named and referenced.

1.6 Chapter Exercises

Exercise 1.6.1 *Both ARM7DTI and ARM64 are* ***load and store*** *ISAs. Write a brief note comparing load and store with the "traditional" CISC memory access ISA instructions. Give and example to illustrate your answer.*

Exercise 1.6.2 *The ARM specification discusses emulation. It refers to a limited emulation mode known as* ***ARM Semihosting****. Write a brief explanation on this ane give an example to support your answer. How does CPUlator support* ***ARM Semihosting****?*

Exercise 1.6.3 *CPUlator is an ARM* ***simulator*** *whereas* `qemu-system-aarch64` *is an ARM* ***emulator****. Write a brief note explaining the difference between these two concepts and give some examples.*

Exercise 1.6.4 *Here is a full list of the* `qemu-system` *emulators on our system:*

```
qemu-system-aarch64        qemu-system-ppc
qemu-system-alpha          qemu-system-ppc64
qemu-system-arm            qemu-system-riscv32
qemu-system-avr            qemu-system-riscv64
qemu-system-hppa           qemu-system-rx
qemu-system-i386           qemu-system-s390x
qemu-system-loongarch64    qemu-system-sh4
qemu-system-m68k           qemu-system-sh4eb
qemu-system-microblaze     qemu-system-sparc
qemu-system-microblazeel   qemu-system-sparc64
qemu-system-mips           qemu-system-tricore
```

```
qemu-system-mips64          qemu-system-x86_64
qemu-system-mips64el        qemu-system-xtensa
qemu-system-mipsel          qemu-system-xtensaeb
qemu-system-or1k
```

Taking any two of these platforms (except `qemu-system-i386` *and* `qemu-system-x86_64`*) compare and contrast them under the following headings:*

1. *Clock speeds*
2. *ISA size (number of instructions)*
3. *Energy consumption*
4. *Operating System / kernel*
5. *Number of units still running*
6. *typical workload*

Two

ELF Structure for Embedded ARM Systems

UNDERGRADUATE
Chapters 2.1–2.7

2.1 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 not only the machine code but also metadata that assists the loader and debugger. ELF is an *architecture-neutral* format—its structure supports multiple instruction sets and processor types. For instance, in a 32-bit ARM executable, the ELF header's `e_machine` field is set to `EM_ARM`.

In *embedded systems*—such as ARM Cortex-M microcontrollers—the ELF file is typically **statically linked** and mapped to specific memory addresses (e.g., flash and SRAM). This is in contrast to *application processors*—like ARM Cortex-A running Linux—where executables may be **dynamically linked** and loaded into virtual memory by an operating system.

Understanding the ELF format is essential for developers working with ARM-based systems. It enables one to inspect and manipulate program layout, debug memory issues, and understand how code and data are placed into memory. ELF serves as the backbone for everything from bare-metal firmware to complex user-space applications.

2.2 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.

2.3 Program Headers and Segments

While the Section Header Table (SHT) is primarily used by linkers and debugging tools, the **Program Header Table (PHT)** describes how an ELF file is mapped into memory for execution. This is the loader's perspective on the binary layout and is crucial for both operating systems and embedded tools that load the ELF.

Each entry in the Program Header Table defines a *segment*, which is a contiguous memory region that will be created in memory. Segments may include one or more sections, grouped by purpose or access rights. For example, the `.text` and `.rodata` sections are often placed in the same read-only and executable segment.

Common Segment Types:.

- `PT_LOAD` – A loadable segment (most common)
- `PT_INTERP`, `PT_DYNAMIC`, `PT_NOTE` – Used in dynamic linking and runtime metadata

Each segment descriptor contains:

- `p_offset` – File offset to the segment
- `p_vaddr`, `p_paddr` – Virtual and physical memory addresses
- `p_filesz`, `p_memsz` – Size of segment in file vs in memory
- `p_flags` – Segment permissions: Read, Write, Execute

In Embedded Systems:. In bare-metal ARM environments, there is no operating system loader. Instead, programming tools (like flashers or debuggers) use the Program Header Table to determine what parts of the file should be loaded into memory. Typical layout includes:

- A flash segment containing `.text` and `.rodata`, marked as readable and executable
- A RAM segment for `.data` and `.bss`, marked as readable and writable

The flash segment includes initial values for `.data`, while the RAM segment typically includes memory reserved for uninitialized data (BSS). The difference between `p_memsz` and `p_filesz` in a RAM segment often indicates the presence of `.bss`, which must be zeroed at startup.

Understanding segments is essential when writing linker scripts or using tools to program embedded devices from an ELF file.

2.4 Tools to Inspect ELF

Even on embedded ARM systems, standard Unix tools are extremely useful for examining the structure of ELF files. These tools allow developers to understand how code and data are organized, validate memory layouts, and debug issues with linking or flashing.

readelf. The `readelf` tool provides a detailed, low-level view of an ELF file's structure. Examples include:

- `readelf -h firmware.elf` – Displays the ELF header, including architecture (e.g., ELF32 for ARM), entry point, and ABI version.
- `readelf -S firmware.elf` – Lists all sections and their addresses.
- `readelf -l firmware.elf` – Displays program headers (segments), showing how the file will be loaded into memory.

objdump. The `objdump` tool allows disassembly of ELF binaries:

- `arm-none-eabi-objdump -d firmware.elf` – Disassembles the `.text` section, showing ARM or Thumb instructions.
- `objdump -h firmware.elf` – Prints section headers with sizes, addresses, and attributes.

nm. The `nm` utility lists symbols defined in the ELF:

- `nm firmware.elf` – Shows function and variable symbols, useful for verifying address placement and linkage.

Practical Use. These tools help verify that:

- `.text` is mapped to flash
- `.data` and `.bss` are placed in RAM
- The entry point (`e_entry`) corresponds to the startup/reset handler

Being familiar with these tools is essential for low-level debugging, analyzing linker script behavior, and ensuring correctness of the final ELF output, particularly in embedded contexts where visibility is limited.

2.5 Memory Layout in Embedded ARM ELF

In embedded ARM systems, particularly with microcontrollers like ARM Cortex-M, memory layout is tightly coupled with the physical hardware. Unlike general-purpose operating systems that use virtual memory, embedded systems operate directly on physical memory addresses. The ELF file must therefore be structured to reflect the actual memory map of the device.

Typical Memory Map. Most ARM microcontrollers feature:

- **Flash Memory** — for storing the program code and read-only data. Typically located at `0x00000000` or `0x08000000`.
- **SRAM (RAM)** — used for variables, stack, heap, and runtime data. Usually starts at `0x20000000`.

The ELF file generated by the linker includes this memory mapping in its program headers, which describe how segments like code (`.text`) and data (`.data`) should be loaded into flash and RAM respectively.

How the ELF Reflects Memory Layout. The linker script assigns memory addresses to each section:

- `.text`, `.rodata`, `.isr_vector` → Flash (read-only)
- `.data`, `.bss` → RAM (read-write)

Flash Layout:

- `.isr_vector` — interrupt vector table at the very beginning
- `.text` — program instructions
- `.rodata` — constants like strings
- Initial values of `.data`

RAM Layout at Runtime:

- `.data` — initialized variables (copied from flash)
- `.bss` — zero-initialized variables
- Stack and heap (managed during execution)

Program Structure at Runtime. When loaded into memory, the ELF segments not only place the code and variables into Flash and RAM, but the runtime environment also sets up additional regions such as the **stack** (usually growing downward from the top of RAM) and the **heap** (growing upward from the end of `.bss`). These are not explicitly represented in the ELF file, but are essential for program execution.

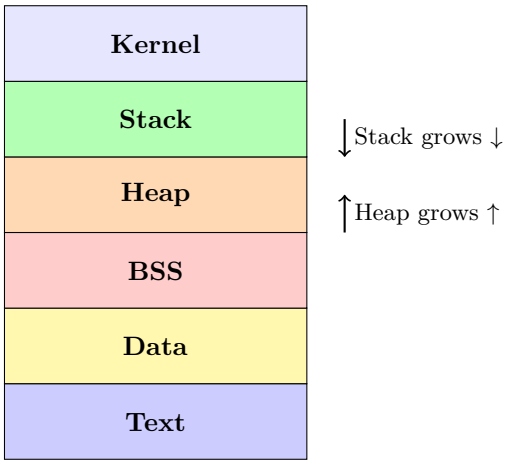


Figure 2.1: Runtime memory layout of an embedded ELF image showing code, data, stack, and heap

Load vs Runtime Addresses. The ELF format allows distinguishing between where a section is stored in the file (Load Memory Address — LMA) and where it should be placed in memory during execution (Virtual Memory Address — VMA).

For example:

```
.data : { *(.data*) } > RAM AT > FLASH
```

2.6 Linking Process Basics

Linking is the final stage of compilation that produces the complete ELF binary from multiple object files. On embedded ARM systems, this process is typically static, meaning that all code and data are fully resolved and placed at fixed memory addresses.

Role of the Linker. The linker takes multiple object files (`.o`) and combines them into a single ELF file. Its main tasks include:

- Merging sections such as `.text`, `.data`, `.bss` from different input files
- Resolving symbol references (e.g., function calls or global variables)
- Assigning memory addresses according to a linker script
- Generating ELF metadata including headers and symbol tables

Linker Script. Embedded projects often use a custom linker script to specify memory layout. A typical script defines memory regions and assigns sections accordingly:

```
MEMORY {
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 256K
  RAM   (rwx): ORIGIN = 0x20000000, LENGTH = 64K
}

SECTIONS {
  .text : { *(.text*) } > FLASH
  .rodata : { *(.rodata*) } > FLASH
  .data : { *(.data*) } > RAM AT > FLASH
  .bss  : { *(.bss*) } > RAM
}
```

Listing 2.1: Example linker script snippet

This layout places code and read-only data in flash, while placing writable data and uninitialized memory in RAM. The `AT > FLASH` directive means the initial values for `.data` are stored in flash, but loaded into RAM at runtime.

Symbol Resolution. The linker also resolves symbols such as function names or variables declared as `extern`. For example, if one file defines a variable and another references it, the linker matches the reference to the definition and fills in the correct address.

Relocation Fixups. During the static linking process, all relocations are resolved. This includes adjusting instruction addresses (e.g., branches, function calls) and data references to their final addresses in flash or RAM. No further relocation is needed at runtime, making the resulting ELF fully self-contained.

Pedagogical Example. Consider two C files:

- `file1.c` defines `int x = 5;` and `void foo() {...}`
- `file2.c` contains `extern int x;` and calls `foo()`

The compiler produces separate object files, and the linker merges them, placing `x` in `.data` and patching the call to `foo()` with its final address in `.text`.

Outcome. After linking, the ELF file:

- Contains all code and data needed to run on the target
- Is mapped exactly to flash and RAM locations
- Has all addresses resolved — ready to be flashed or loaded into the device

This process is central to embedded firmware development, where precise control of memory and startup behavior is required.

GRADUATE
Chapters 2.8–2.11

2.7 Symbol Resolution and Linking Semantics

In complex systems (especially operating systems or large embedded applications) the way symbols are defined and resolved becomes critical. The linker must coordinate across multiple object files and static libraries to correctly associate references with definitions.

Strong vs Weak Symbols. ELF defines two types of symbols:

- **Strong symbols** — definitive declarations such as global functions or initialized variables.
- **Weak symbols** — fallback or optional declarations, typically used for default handlers or overrideable functionality.

The linker follows these rules:

1. Only one strong symbol of a given name is allowed; multiple strong definitions result in an error.
2. A strong symbol overrides a weak symbol of the same name.
3. If multiple weak definitions exist and no strong one is found, any of the weak definitions may be used.

This system is commonly used in embedded startup code, where weak default handlers (e.g., interrupt service routines) can be overridden by strong user-defined implementations.

Static Libraries and Symbol Pulling. When linking against static libraries (`.a` files), the linker includes only the object files that resolve an undefined symbol. Notably:

- Weak undefined references **do not** cause an archive member to be pulled in.
- Only strong references trigger inclusion of matching object files from the archive.

Symbol Table and Resolution Flow. The linker creates a global symbol table by combining:

- Defined symbols from object files and libraries
- Undefined references that need to be resolved

It then matches each undefined symbol to a defined one. If none is found and the reference is strong, the linker reports an error. If it's weak, the symbol can be resolved to zero (or left unresolved, depending on context).

Example Scenario. Suppose:

- `foo.c` defines `int x = 5;`
- `bar.c` has `extern int x;` and calls `foo();`
- `libutil.a` defines a weak version of `foo()`

If the strong `foo()` is present in `foo.c`, it will override the weak one in `libutil.a`. If it is not, the linker may use the weak version.

Understanding these rules is essential when building modular systems, reusing libraries, and writing linker scripts that handle default fallbacks gracefully.

2.8 ELF Relocation Mechanism

During compilation, object files are typically relocatable: function calls, data references, and constant addresses are not yet resolved. These unresolved parts are patched later by the linker using a mechanism called **relocation**.

Relocation Entries. Each relocatable ELF object (`.o` file) contains a relocation section (e.g., `.rel.text` or `.rela.data`) listing instructions or data that require adjustment. Each relocation entry specifies:

- The offset in the section to be patched
- The symbol to which it refers
- The relocation type (e.g., absolute address, PC-relative, etc.)

Common ARM Relocation Types. For 32-bit ARM (ARMv7), typical relocation types include:

- `R_ARM_ABS32` — absolute 32-bit address
- `R_ARM_THM_CALL` — Thumb BL (branch with link) instruction
- `R_ARM_REL32` — relative offset from current instruction
- `R_ARM_MOVW_ABS_NC`, `R_ARM_MOVT_ABS` — used in 32-bit constant generation

For 64-bit ARM (AArch64), equivalent relocations include:

- `R_AARCH64_ADRP_REL`, `R_AARCH64_ADD_ABS_LO12_NC`
- `R_AARCH64_JUMP_SLOT`, `R_AARCH64_RELATIVE`

Static Linking. In statically linked executables, all relocation entries are applied at link time:

- Function calls are resolved to fixed addresses.
- Variable references are patched with absolute or relative values.
- The relocation sections are often stripped from the final ELF.

This ensures the binary is ready to run directly from memory without further adjustments.

Example:. A call to a function `foo()` in Thumb mode might generate a relocation of type `R_ARM_THM_CALL`, which instructs the linker to patch a placeholder with the address of `foo` using a 16-bit BL instruction encoding. The linker must ensure that the call is within range or insert veneers if necessary.

Relocation Records. Each relocation entry (in ELF) refers to:

- A symbol index (into the symbol table)
- The target location to patch
- The relocation type

Important Note:. Relocation is entirely resolved in static ELF files for embedded systems. In contrast, dynamically linked ELF binaries keep some relocations for the dynamic linker to apply at runtime.

Understanding relocation types is critical for advanced debugging, reverse engineering, and creating custom linker scripts that properly define address ranges and symbol resolution.

2.9 Dynamic Linking, GOT, and PLT

On systems running an operating system (e.g., Linux on ARM Cortex-A), executables often make use of **dynamic linking** to share common libraries at runtime. This allows for:

- Smaller binary sizes
- Easier updates to shared libraries
- Memory efficiency through shared code pages

Key Dynamic ELF Sections. Several sections are included in ELF files to support dynamic linking:

- `.interp` — holds the path to the dynamic linker (e.g., `/lib/ld-linux.so.3`)
- `.dynamic` — a table containing dynamic linking metadata (e.g., `DT_NEEDED`, `DT_SYMTAB`)
- `.dynsym` and `.dynstr` — symbol table and associated strings for dynamic references
- `.rel.plt` and `.rel.dyn` (or `.rela.*`) — relocation entries the dynamic linker must process
- `.got`, `.got.plt` — Global Offset Table sections
- `.plt` — Procedure Linkage Table stubs

Global Offset Table (GOT). The GOT is a table of pointers to global variables and functions that are dynamically resolved. Initially, the GOT entries may point to resolver routines. After the dynamic linker finishes its work, GOT entries are updated to point to the actual memory addresses.

Split GOT Usage:

- `.got` — general-purpose entries (e.g., global variables)
- `.got.plt` — specific to PLT entries for function calls

In ARM shared object files, early GOT entries often serve special roles:

- GOT[0] — might hold address of the dynamic section
- GOT[1] — module ID
- GOT[2] — resolver trampoline

Procedure Linkage Table (PLT). The PLT contains a sequence of stubs, one per dynamically linked function. These stubs perform an indirect jump via the GOT entry for the function. If the GOT entry hasn't been resolved yet, it triggers a call to the dynamic linker, passing the symbol index.

Lazy Binding Flow (ARMv7 Example):

1. First call to `foo()` jumps to `foo@plt`.
2. The PLT stub jumps to `GOT[foo]`, which initially points to a dynamic linker resolver.
3. The resolver looks up `foo` in the loaded shared libraries.
4. GOT[foo] is patched with the resolved address.
5. Future calls go directly through GOT[foo] without invoking the resolver again.

This technique is known as **lazy binding** and is the default on most Linux systems. It reduces startup time, especially when many functions are imported but not all are used.

Dynamic Relocation Types. Relocations in dynamic linking are processed at runtime by the dynamic loader. Common ARM types include:

- `R_ARM_JUMP_SLOT` — used in `.rel.plt`; updates GOT entry with function address
- `R_ARM_ABS32` — absolute relocation for data addresses
- `R_ARM_REL32` — relative offset adjustments

On AArch64 systems, analogous relocation types include:

- `R_AARCH64_JUMP_SLOT`, `R_AARCH64_RELATIVE`, `R_AARCH64_GLOB_DAT`

ELF Metadata for the Loader. Dynamic linking metadata is stored in the `.dynamic` section, a table of tags (prefixed with `DT_`):

- `DT_NEEDED` — names of required shared libraries
- `DT_PLTGOT` — location of the GOT
- `DT_JMPREL`, `DT_PLTRELSZ` — location and size of PLT relocations

The loader reads these entries at runtime and performs the necessary symbol resolution using the dynamic symbol table and relocations.

Security and Performance Implications. Modern systems may disable lazy binding (`LD_BIND_NOW=1`) for better startup consistency. Some also use techniques like *RELRO* (Read-Only Relocations) and *PIE* (Position Independent Executables) to harden the dynamic linking process.

Summary. The GOT and PLT work together to enable dynamic linking by allowing:

- Functions to be indirectly called through GOT entries
- The dynamic loader to patch unresolved symbols at runtime
- Shared libraries to remain modular and position-independent

Understanding this mechanism is essential for advanced development, debugging symbol resolution issues, and analyzing runtime behavior of complex ELF-based systems.

2.10 Relocation Types in Dynamic Linking

In dynamically linked ELF files, symbol resolution is deferred to runtime. This requires the dynamic linker (e.g., `ld-linux.so`) to process relocation entries and patch memory with correct addresses. The relocation mechanism ensures that function calls, data accesses, and global variables are linked to their final locations after loading shared libraries.

Dynamic Relocation Sections. Dynamic relocations are typically found in the following sections:

- `.rel.dyn` or `.rela.dyn` — general-purpose runtime relocations
- `.rel.plt` or `.rela.plt` — relocations for function calls via the Procedure Linkage Table

The difference between `rel` and `rela`:

- `REL` entries take the addend from the memory being relocated.
- `RELA` entries include the addend explicitly in the relocation entry itself.

ARM 32-bit typically uses `REL`; AArch64 uses `RELA`.

Common ARM Relocation Types (ARMv7).

- `R_ARM_JUMP_SLOT` — used for PLT function calls; updates a GOT entry
- `R_ARM_GLOB_DAT` — absolute address of a global symbol
- `R_ARM_ABS32` — general absolute data reference
- `R_ARM_RELATIVE` — adjusts memory by adding the program's load base (used for position-independent code)

Common AArch64 Relocation Types.

- `R_AARCH64_JUMP_SLOT` — for PLT function calls
- `R_AARCH64_GLOB_DAT` — global data address resolution
- `R_AARCH64_RELATIVE` — adds base address to a memory location
- `R_AARCH64_ABS64` — absolute 64-bit address

Thread-Local Storage Relocations. Thread-local variables introduce additional relocation types (e.g., `R_ARM_TLS_*`, `R_AARCH64_TLS_*`). These are resolved differently depending on TLS model (e.g., local-exec, initial-exec, global-dynamic).

Copy Relocations. Some global variables (typically defined in shared libraries) are copied into the main executable at runtime using copy relocations:

- The symbol is declared in both the executable and the shared library.
- The dynamic linker copies the value from the shared object into the executable's memory during relocation.
- This is typically represented by `R_ARM_COPY` or `R_AARCH64_COPY`.

Relocation Application. At runtime, the loader:

1. Iterates over relocation entries in `.rel.dyn` and `.rel.plt`.
2. Applies each relocation based on its type and symbol reference.
3. Patches the appropriate memory address with the resolved value.

Example (ARM 32-bit):. A relocation entry of type `R_ARM_JUMP_SLOT` for a function `foo` tells the loader:

- "Patch this GOT entry with the runtime address of `foo()`."
- Initially, the entry might point to the dynamic resolver.
- After the first call, it is replaced with the actual function pointer.

Why It Matters. Understanding dynamic relocation types is essential for:

- Writing and debugging shared libraries
- Reverse engineering ELF binaries
- Implementing loaders, debuggers, and binary instrumentation tools

These relocation entries form the foundation of position-independent execution and dynamic linking on modern ARM systems.

2.11 Linker Script: Deep Dive

The linker script is a critical tool for embedded developers working with ELF files. It controls how input sections from object files are arranged in memory, and directly influences the contents and structure of the final ELF file.

Overview. A typical linker script has two main parts:

- The `MEMORY` block — defines memory regions (flash, RAM) available on the target
- The `SECTIONS` block — maps input sections to those memory regions and sets their placement order

Memory Region Declaration. Each memory region is given a name, origin address, and length. For example:

```
MEMORY {
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 512K
  RAM   (rwx): ORIGIN = 0x20000000, LENGTH = 128K
}
```

Listing 2.2: Example MEMORY declaration

Here, FLASH is marked as readable and executable, and RAM as readable, writable, and executable (typically needed for stack/heap).

Section Mapping. The `SECTIONS` block tells the linker how to assign and order output sections. Each section declaration can:

- Gather matching input sections (e.g., all `.text.*`)
- Specify the output address region
- Set runtime and load addresses (using `AT()` or `>`)

```
SECTIONS {
  .text : { *(.isr_vector) *(.text*) } > FLASH
  .rodata : { *(.rodata*) } > FLASH
  .data : { *(.data*) } > RAM AT> FLASH
  .bss : { *(.bss*) } > RAM
}
```

Listing 2.3: Typical SECTIONS block

AT vs > Placement.

- `> RAM` — places section in RAM at runtime (virtual memory address)
- `AT > FLASH` — specifies that initial values are loaded from FLASH (load memory address)

This distinction is important for sections like `.data`, which must be loaded from flash but reside in RAM at runtime.

Symbols and Labels. Linker scripts can define symbols that the startup code relies on to copy or zero sections:

```
__data_load_start = LOADADDR(.data);
__data_start = ADDR(.data);
__data_end = .;
```

Listing 2.4: Symbol usage

These can be used in C code as `extern` declarations to implement startup initialization.

Scatter Loading and Multi-region Layouts. More complex embedded systems may have multiple flash banks, external RAM, or special-purpose memory. These require:

- Multiple `MEMORY` entries
- Carefully assigned section placements
- Multiple `PT_LOAD` segments

ARM proprietary linkers (e.g., `armlink`) use concepts like *load regions* and *execution regions* to implement scatter loading, which maps closely to ELF's segment/section architecture.

Program Header Generation. The linker generates ELF program headers (`PT_LOAD`, etc.) based on:

- Section addresses and load locations
- Permissions (based on section flags)

Tools like `readelf -l` can be used to inspect resulting segments. For example:

- A flash segment with `R-X` permissions
- A RAM segment with `RW-` permissions, and `p_filesz < p_memsz` indicating zero-initialized `.bss`

Best Practices.

- Group all code and constants together in flash
- Place all modifiable data in RAM
- Avoid gaps or overlaps in memory placement
- Use clear symbol labels to aid in boot-time initialization

Debugging Tip. If startup code fails or variables hold garbage values, it's often due to:

- Incorrect `AT()` vs `>` placement
- Missing or incorrect symbol labels
- Misaligned memory ranges in the linker script

Conclusion. Understanding the linker script at a low level is essential for embedded ARM development. It gives fine-grained control over how memory is used, ensures correct runtime behavior, and shapes the structure of the ELF file that is ultimately loaded onto the device.

Exercises with Answers

Exercise 2.11.1 ***Introduction to ELF on ARM:*** *Explain the main difference between an ELF file and a raw binary file. Why is the ELF format especially useful for the development of embedded systems?*

Answer: *An ELF file includes metadata such as section headers, program headers, and symbol tables that describe how to load and execute the program. A raw binary is simply a flat memory dump with no structural information. ELF files are useful in embedded systems because they provide the loader with information on how to map code and data to physical memory.*

Exercise 2.11.2 ***ELF Headers and Section Basics:*** *Given an output from* `readelf -h`*, identify the target architecture, endianness, entry point, and object file type (relocatable, executable, etc.). What does the Section Header Table contain, and why is it important?*

Answer: *The ELF header provides:*

- *Target architecture (e.g., ARM)*
- *Endianness (little or big endian)*
- *Entry point (e.g., address of* `main()` *or reset handler)*
- *Object type (e.g., executable or relocatable)*

The Section Header Table lists sections like `.text`*,* `.data`*, and* `.bss`*, and includes their sizes, addresses, and file offsets. It is mainly used by the linker and debugging tools.*

Exercise 2.11.3 ***Program Headers and Segments:*** *Describe the role of the Program Header Table in loading an ELF file. List three common segment types found in ELF files and explain their purpose.*

Answer: *The Program Header Table (PHT) describes how the ELF file is loaded into memory. Each entry defines a segment with file offset, virtual address, memory size, and access permissions. Common segment types:*

- `PT_LOAD` *— Loadable segment (e.g., code or data)*
- `PT_INTERP` *— Interpreter path for dynamic linking*
- `PT_DYNAMIC` *— Dynamic linking information*

Exercise 2.11.4 ***In Embedded Systems (Segment Mapping):*** *Why do embedded systems typically use two* `PT_LOAD` *segments in an ELF file? What is the significance of having* `p_filesz < p_memsz` *in one of them?*

Answer: *One* `PT_LOAD` *segment maps flash (code and read-only data), and the other maps RAM (writable data and BSS). If* `p_filesz < p_memsz`*, it indicates that part of the segment (typically* `.bss`*) is not stored in the file but is allocated and zeroed in memory at runtime.*

Exercise 2.11.5 ***Tools to Inspect ELF:*** *Run the following commands on a sample ELF file and explain the output:*

- `readelf -S firmware.elf`
- `objdump -d firmware.elf`
- `nm firmware.elf`

What insights do these tools provide into section layout, function addresses, and symbol resolution?

Answer: `readelf -S` *shows section headers, addresses, sizes, and flags.* `objdump -d` *displays disassembled machine code, allowing inspection of instruction-level layout.* `nm` *lists symbols (functions, variables), their addresses, and linkage type (global, local, etc.). These tools help validate code placement, linkage, and runtime behavior.*

Exercise 2.11.6 ***Memory Layout in Embedded ARM ELF:*** *Draw a memory map of a typical Cortex-M ELF image, labeling flash and RAM regions. Mark where the following sections are located at runtime:*

- `.isr_vector`
- `.text`
- `.data`
- `.bss`

Answer:

- *Flash (*`0x08000000`*):* `.isr_vector`*,* `.text`*,* `.rodata`*, initial* `.data`
- *RAM (*`0x20000000`*):* `.data` *(after copy),* `.bss` *(zero-initialized), stack and heap*

Exercise 2.11.7 ***Linking Process Basics:*** *Describe the role of the linker in embedded development. What is the purpose of a linker script, and how does it influence the final memory layout of the ELF?*

Answer: *The linker merges object files and resolves symbols. A linker script defines memory regions and assigns sections to them. It controls where code and data go in flash and RAM, affecting runtime behavior and ELF segment layout.*

Exercise 2.11.8 ***Symbol Resolution:*** *Suppose one file defines a function* `foo()` *and another file calls it. What steps does the linker perform to match the call to the definition? What happens if two strong definitions of* `foo()` *exist?*

Answer: *The linker scans symbol tables. It sees* `foo()` *as undefined in one file and defined in another, so it resolves the reference. If two strong definitions exist, the linker throws a multiple definition error.*

Exercise 2.11.9 ***Relocation Fixups:*** *Explain what a relocation entry is. What information does it contain? Why are relocations important when linking multiple object files?*

Answer: *A relocation entry indicates that a piece of code or data needs to be updated based on symbol resolution. It includes the offset to patch, the symbol index, and the type (absolute, relative, etc.). Relocations allow code from separate files to be integrated correctly at final link time.*

Exercise 2.11.10 ***Pedagogical Example – Static Linking:*** *Given two files,* `main.c` *and* `utils.c`*, where* `main` *calls a function from* `utils`*, describe the process from compilation to final ELF generation. Use specific terms such as object file, relocation, symbol table, section merging, and entry point.*

Answer: *Each file is compiled into an object file with its own symbol table.* `main.o` *references a symbol defined in* `utils.o`*. The linker merges sections, resolves symbols, applies relocations, and writes an ELF file with a defined entry point. The final ELF contains no unresolved references and is ready to be loaded onto hardware.*

Exercise 2.11.11 ***Outcome:*** *After linking is complete, what does the ELF file contain? How is it different from an object file? What parts of the ELF are used by the loader, the debugger, and the runtime environment?*

Answer: *The ELF file contains fully linked sections, program headers, symbol tables (for debugging), and entry point info. Unlike object files, it has no unresolved symbols or relocations. The loader uses program headers, the debugger uses symbol tables, and the runtime uses the actual memory image.*

Exercise 2.11.12 ***Symbol Resolution:*** *What is the difference between strong and weak symbols in ELF? How does the linker handle them when both types are present for the same name?*

Answer: *Strong symbols represent definitive definitions (e.g., initialized variables or functions), whereas weak symbols serve as fallback definitions. If both exist, the strong one overrides the weak one. If only weak definitions exist, the linker may choose any of them.*

Exercise 2.11.13 ***Static Libraries and Linking:*** *Why do weak undefined symbols not cause archive members in static libraries to be pulled during linking?*

Answer: *Because weak undefined symbols are considered optional, the linker does not treat their absence as an error and does not pull in archive members unless a strong reference is present.*

Exercise 2.11.14 ***Relocation Mechanism:*** *What are relocation entries in ELF, and what do they typically contain?*

Answer: *Relocation entries mark where addresses must be patched. Each entry includes:*

- *The offset within the section to modify,*
- *The symbol being referenced,*
- *The relocation type (e.g., absolute, relative).*

Exercise 2.11.15 ***ARM Relocation Types:*** *During static linking on ARM, what does a relocation type like* `R_ARM_ABS32` *instruct the linker to do?*

Answer: *It instructs the linker to replace the 32-bit placeholder with the absolute address of the referenced symbol.*

Exercise 2.11.16 ***Procedure Linkage Table (PLT):*** *Describe the purpose of the Procedure Linkage Table (PLT) in dynamic linking.*

Answer: *The PLT contains stubs for each dynamically linked function. These stubs jump via the GOT. If the function hasn't been resolved yet, the stub triggers a resolver which looks up and patches the actual function address in the GOT.*

Exercise 2.11.17 ***Lazy Binding:*** *What is lazy binding, and why is it useful in systems with many shared libraries?*

Answer: *Lazy binding defers the resolution of function addresses until they are first called, reducing initial startup time and resolving only the symbols that are actually needed during execution.*

Exercise 2.11.18 ***Dynamic Relocations:*** *Compare the roles of* `.rel.dyn` *and* `.rel.plt` *sections in a dynamically linked ELF file.*

Answer: `.rel.dyn` *holds relocations for variables and data, while* `.rel.plt` *contains relocations for function calls managed by the PLT and resolved through the GOT.*

Exercise 2.11.19 ***REL vs RELA:*** *What is the functional difference between REL and RELA relocation formats in ELF?*

Answer: *REL entries assume the addend is stored in the section being relocated. RELA entries explicitly store the addend in the relocation record itself.*

Exercise 2.11.20 ***Linker Script Semantics:*** *In a linker script, what does* `> RAM AT > FLASH` *mean?*

Answer: *It means the section is placed in RAM at runtime but its contents are loaded from flash. This layout is used for initialized data that needs to be copied during startup.*

Exercise 2.11.21 ***Linker-Defined Symbols:*** *Why is defining symbols like* `__data_start` *and* `__data_end` *useful in embedded systems?*

Answer: *These symbols define boundaries used by startup code to initialize RAM sections (e.g., copying data, zeroing* `.bss`*). They also help in memory usage tracking and debugging.*

Three

Assembly Language Programming

We begin our study of assembly by focusing on ARM7DTI which is a ARM32-bit platform. The ARM64 platform contains many additions and updates, and in most real-world scenarios you will be building and deploying software on this ISA. However, the 32-bit variant is interesting from the pedagogical point of view because it is a smaller and significantly simpler ISA.

As we proceed through the following chapters we will compare C programming and assembly to understand how tasks in high level programming languages are implemented in low level assembly.

3.1 Basic Instructions: Adding and Subtracting

Let us begin by doing a very simple task: adding two numbers together and storing the result in a third number. Let's choose 5 and 10 as our two numbers. Mathematically this is simply $c = 5 + 10$, and in C:

```
int a = 5;
int b = 10;
int c = a + b;
```

As you can see, we used three named integer variables, (`a`, `b` and `c`). We also used assignments (`=`) and the addition function (`+`) How do we implement this in assembly? We need to use the assembly assignment instruction (`mov`) and the addition mathematical function (`add`) instruction:

```
mov r1, #5     // move 5 into r1
mov r2, #10    // move 10 into r2
add r3, r1, r2 // add r1 and r2 place result in r3
```

Discussion

Notice some interesting features of the assembly code. Let's deal with each of these in turn:

1. We use *registers* to store data. Registers are word-size storage areas on-board the CPU itself. Because they are on the CPU, register updates happen in one CPU clock cycle (in other words, zero latency).

In ARM7DTI there are 15 *general purpose* registers. These registers are named `r0-r14`. Register `r15` is a special purpose register and should not be used by the programmer.

As there are only 15 general purpose registers, we must be careul not to use them unneccessarily, otherwise we may run out of registers.

2. Two new instructions have been introduced: `mov` and `add`. Notice also that when we want to use an integer value, we use the `#` sign in front of it. For example, `#10`. This is a language requirement. Numbers like this are called *immediates*. Later we will explain the derivation of the word, but for now, let us accept this term as-is.

3. In ARM7DTI all instructions are either right-to-left or left-to-right ordered [1]. The `mov` instruction is right-to-left ordered. It means take the *value* from the right hand side and place it into the *location* on the left hand side. Is it possible to have a `mov` that uses *two* locations (eg, `mov r1, r2`)? Yes. Is it possible to have a move with two values `mov #3, #4`)? Of course, no.

   Notice that `mov` is a two operand instruction

4. The `add` instruction is also a right-to-left instruction. Notice that `add` is a *three* operand instruction. So the code `add r3, r1, r2` is understood as *"add the value in r2 to the value in r1 and place the result in r3"*. Does this instruction change the value of either `r1` or `r2`? No. Does it change the value of `r3`? Yes.

   Some variants of `add` that achieve the same result:

   ```
   mov r1, #5
   add r3, r1, #10
   ```

   and

   ```
   mov r1, #5
   add r3, r1
   ```

   Notice here that `r3` does not contain 15 but rather it contains 10.

5. Finally, notice the following illegal instructions:

   ```
   // illegal
   add r3, #10, r1 // cannot add a reg to a constant
   // causes this compiler error
   Error: constant expression expected -- ‘add r3
       ,#10,r1’
   ```

[1] most are right-to-left ordered

and

```
// illegal
add r3, #10, #5
// causes this compiler error
Error: bad expression -- `add r3,#10,#5'
```

We cannot add a register to a constant, nor can we add a constant to a constant. The reason for these contraints is not exactly as you might suppose. We will discuss these types of constraints in Chapter 9 (Machine Code), but suffice it to say, the constraints are due to the nature of ARM7DTI RISC ISA, and in particular, the constraint of fixed word-size instruction length.

Negative Numbers

In C, we use the type `unsigned` to specify that the variable cannot take on a value of less than zero. However, due to the implicit type conversion rules in C, it is legal to perform the following:

For example:

```
unsigned i = 100; // a non-negative number
i = -1; // implicit type conversion
```

If we deal only with unsigned integers, then all the bits of the ISA word size are allocated in representing the number. To make the study of this a litle easier, let us deal with a 6-bit computer (the principles generalize to 32− and 64−bit systems in exactly the same way).

The largest non-negative number expressible in a computer of word size N bits, is $2^N - 1$. This means in a six bit CPU, the largest non-negative number is $2^6 - 1 = 63$. How then should we represent a negative number? Here we introduce the idea of *two's complement* to store a negative number.

A two's complement negative number is computed as follows:

1. Take the bit pattern of the positive number and invert all the bits ($0 \rightarrow 1$ and $1 \rightarrow 0$)
2. Add one (bit) to the resulting number.

Let us look at an example: -5 on a 6−bit CPU.

1. $5 = 000101 \rightarrow 111010$
2. Now, add 1 to $111010 = 111011$

Notice that 111011 can be thought of in two different ways. As a non-negative number it is

$1 * (2^5) + 1 * (2^4) + 1 * (2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) = 59$

However, because the most-significant bit (MSB) is 1, this number is *also* a negative number. The negative number is computed as follows:

$-1 * (2^5) + 1 * (2^4) + 1 * (2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) = -5$

Let us test to see if *addition* under two's complement works correctly. We know that $27 + -5 = 22$. Does it work? Let's see:

$$\begin{array}{r} 011011 \\ +111011 \\ \hline\hline 1 \overset{\text{carry}}{\longleftarrow} 010110 \end{array}$$

As $010110 = 22$ we can see that addition and subtraction of posive and negative two's complement works correctly.

Note, in the example above, we had a *carry-out* of the MSB. This arises because *there was no more space* left in the 6-bit computer for the full answer. (in other words, the result needed a 7-bit computer, which we were not using)

Registers can hold positve or negative numbers without the need to use words such as `signed`, or `unsgined`:

```
mov r1, #10 // move 10 into r1
mov r2, #-5 // move negative 5 into r2
mov r3, r2  // move (r2) negative 5 into r3
```

Finally, let us look briefly at `sub` instructions.

```
mov r1, #10 // move 10 into r1
mov r2, #-5 // move negative 5 into r2
mov r3, #5 // move 5 into r3
add r4, r1, r2 // result is 5, stored in r4
sub r4, r1, r3 // result is 5, stored in r4
sub r5, r2, r1 // result is -15, stored in r4
// etc
```

Mathematical Instructions

Table 3.1 shows some of the more commonly used ARM7DTI mathematical operations. It is by no means complete. Please refer to [Ins25] for a comprehensive treatment of ARM7DTI instructions.

Table 3.1: Top 10 ARM32 Mathematical Instructions

Instruction	Type	Description
`add{cond}{S}`	Arithmetic	Adds two registers or a register and an immediate value, storing the result in a destination register.
`sub{cond}{S}`	Arithmetic	Subtracts one register or immediate value from another, storing the result in a destination register.
`mul{cond}{S}`	Multiplication	Multiplies two registers and stores the 32-bit result in a destination register.
`mla{cond}{S}`	Multiplication	Multiplies two registers, adds a third register to the product, and stores the result in a destination register.
`umull{cond}{S}`	Multiplication	Unsigned multiply long, multiplies two 32-bit registers to produce a 64-bit result, stored in two registers.
`smull{cond}{S}`	Multiplication	Signed multiply long, similar to `UMULL` but for signed integers.
`adc{cond}{S}`	Arithmetic	Adds two registers or a register and an immediate value with carry, storing the result in a destination register.
`sbc{cond}{S}`	Arithmetic	Subtracts one register or immediate value from another with borrow, storing the result in a destination register.

A note on `{cond}{S}`

You may notice in Table 3.1 and Table 3.2 the text `{cond}{S}` appended to the end of the mathematical instruction (for example `sub{cond}{S}`). We should first note that there are two different thiings going on here, both of which are optional:

1. `instr{cond}` indicates the instruction should execute only if `{cond}` is *true*. This is called a conditional instruction. For example `mullt` consists of `mul` and `lt`. Here `lt` means *less than*, and signifies that the `mul` operation should only execute *if the condition flags permit*.

2. `instr{s}` indicates the instruction outcome of the instruction (less-than, zero, greater-than etc) should be sent to the *conditional registers* so that other instructions may access the information.

We will discuss both `instr{cond}` and `instr{s}` in the next chapter. For now, to keep things simple, we omit these optional appendages and focus on the mathematical operation itself.

Multiplication using the `mul` family

We now turn to multiplication operations. There are two main multiplication instructions in ARM assembly ISA: `mul` and `mula`. We will briefly discuss

each. We use `mul` when we want to store the result of the operation into some register as follows:

```
1 mov r1, #10
2 mov r2, #5
3 mul r4, r1, r2
```

We can understand this line 3. as *"multiply the value in `r1` by the value in `r2` and store the result in `r3`"*
Here, `r4` now contains 50. This is equivalent to the C code:

```
1 int a, b, c;
2 a = 10;
3 b = 5;
4 c = a * b;
```

In C we often want a *multiplication-accumulation* function as:

```
1 int a, b, c;
2 a = 10;
3 b = 5;
4 c = a * b;
5 d += c;
```

Because multiplication-accumulation is such a common mathematical operation, ARM assembly has a special *four* operand instruction, (`mula`) to do this:

```
1 mov r1, #10
2 mov r2, #5
3 mla r4, r1, r2, r5
```

Note here that `r5` is the accumulation register, and is an accumulation of the product of `a` and `b` plus whatever was in `r5` previously.

3.1.1 Logical Operators

Finally, let us briefly mention the logical Operators in ARM7DTI ISA:

3.2 Program Flow and the Program Counter

Sequential Execution

Assembly programs have a wide number of structural characteristics in common with high level languages (such as C). These structural similarities are not total. There are plenty of points of difference. For the programmer writing assembly

Table 3.2: ARM32 Logical Operator Instructions

Instruction	Description
`and{cond}{S} Rd, Rn, <Operand2>`	Bitwise AND between `Rn` and `Operand2`, result in `Rd`
`orr{cond}{S} Rd, Rn, <Operand2>`	Bitwise OR between `Rn` and `Operand2`, result in `Rd`
`eor{cond}{S} Rd, Rn, <Operand2>`	Bitwise XOR between `Rn` and `Operand2`, result in `Rd`
`bic{cond}{S} Rd, Rn, <Operand2>`	Bitwise AND of `Rn` with NOT of `Operand2`, result in `Rd`
`orn{cond}{S} Rd, Rn, <Operand2>`	Bitwise OR of `Rn` with NOT of `Operand2`, result in `Rd`

code for the first time, one of the most obvious differences is in program flow. In assembly, there is no *scope-control* equivalent.

This means that your assembly code will start at the entry point and continue sequentially until your program either exits or branches. We will discuss branching in Chapter **??**.

In the example below, the program begins executing at line 1. and stops at line 6.

```
1 mov r1, #10
2 mov r2, #-5
3 mov r3, #5
4 add r4, r1, r2
5 sub r4, r1, r3
6 sub r5, r2, r1
```

The Program Counter (PC)

In ARM7DTI and ARM64 there is a dedicated register known as the *Program Counter* register (`pc` register), the job of which is to point to the address in memory of the currently executing instruction [2]. Suppose in the above example, the instructions are placed into memory in the following sequential hexadecimal addresses:

```
//Address  Instruction
00FFFF00   mov r1, #10
00FFFF04   mov r2, #-5
00FFFF08   mov r3, #5
00FFFF0C   add r4, r1, r2
00FFFF10   sub r4, r1, r3
```

[2] this is a slight simplification which we refine in Chapter **??**

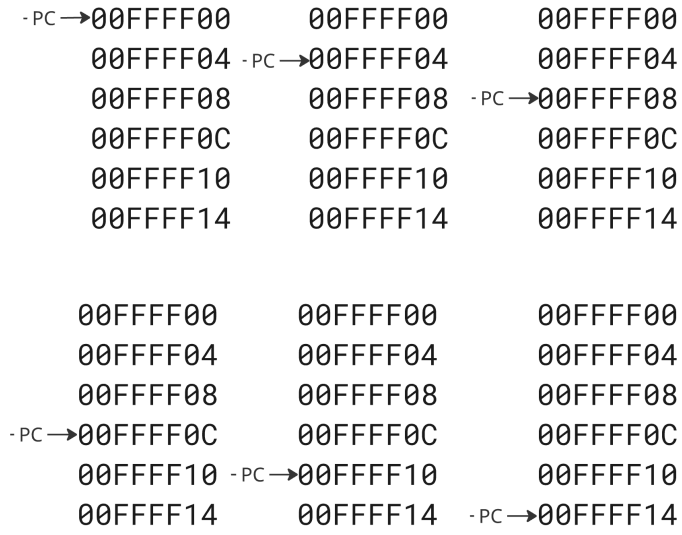


Figure 3.1: The progress of the program counter `pc` values over six instructions

```
00FFFF14   sub r5, r2, r1
```

Then the program counter will change over time as shown in Fig. 3.1:

The `pc` is modifiable by the programmer. But we need to be extremely carefull in doing so, because changing the programmer counter to address A will cause the instruction at address A to be read and processes by the CPU. A should always contain a valid instruction - if it does not - the program will crash.

Program Exit

A C program can exit and return control to the operating system in one of two ways, both shown in Fig 3.2.

Note in Fig. 3.2 - by convention - a program that returns 0 to the operating system (eg, `return 0`) indicates it exited normally. A non-zero value (eg, `exit(3)`) indicates the program encountered an error or some abnormal ending.

If the function call stack is deep and unwinding it via return values is inconvenient, a function can use `exit()` to terminate the program in the function. For example see Fig 3.3:

In assembly there is a direct equivalent to the C `return` and `exit()`. We will briefly show the options in assembly, but the instructions we will explain in detail in future chapters.

```
int main(int argc, char* argv[] {
  // method 1 - use "return"
  return 0;

  // method 2 - use "exit"
  exit(0);
}
```

Figure 3.2: We can use `return 0` or `exit(0)` to return from `main`. The best practice is to use `return`

```
int main(int argc, char* argv[] {
    A();
}
void A() {
    B();
}
void B() {
    C();
}
void C(){
    if (fatal_error) {
        // just exit here
        exit(2);
    }
}
```

Figure 3.3: Due to the deep nesting of function calls, it is easier here to `exit(2)` rather than unwinding the function call stack

3.3 Chapter Summary

In this chaper we introduced the basic concepts of ARM7DTI assembly programming. We reviewed the main mathematical and logical operations, the role of the Program Counter (`pc`). We also discussed how to terminate an assembly language program (`return` versus `exit()`) and the tradeoffs of each approach. We introduced some important topics that will be covered later in this book, in particular:

1. Stack manipulation using `push` and `pop`.
2. Branch with link using `bl`
3. Setting the `pc`.

```
main:
  // the "return" pattern
  push {lr} // save lr on the stack
  // implement your application
  mov r0, #0 // set the return value in r0
  pop {pc} // set the pc to the lr

  // the "exit" pattern
  // implement your application
  mov r0, #0 // set the return value
  bl exit // call exit from libc
```

Figure 3.4: In assembly we can `return 0` by using using the first pattern and call `exit(0) using the second pattern.`

4. Conditional execution and/or conditions register updates using `instr{cond}{S}`

We will return to these interesting and importand topics in future chapters.

3.4 Chapter Exercises

Exercise 3.4.1 *Implement the following C program in assembly:*

```
int main(int argc, char* argv[] {
  // we encountered an error
  return 16;
}
```

Exercise 3.4.2 *Convert this assembly program to C and compile it:*

```
mov r0, #5
mov r1, #50
mov r2, #0
mla r3, r2, r1, r4
```

Use `printf` *to test the result is correct.*

Exercise 3.4.3 *Modify the previous program to "return" to the operating system with a* 0*.*

Exercise 3.4.4 *Write the assembly code to use the logical* `and` *operator to test if a number is odd or even. If the number is odd place* 0 *into* `r0` *and if it's even, place* 1 *in* `r0`*.*

Exercise 3.4.5 *This code does not properly "return" to the opeating system. Why not? What does this code actually do?*

```
push {lr}
mov r0, #0
pop {lr}
```

Four

Conditional Execution

4.1 Branching

All programming languages allow us to control the flow of execution based on some logical test. Of course, we are all familiar with this typical program form:

```
if (condition_is_true) {
    // execute instructions a,b,c,d
}
else {
    // execute instructions e,f,g,h
}

// execute instructions i,j,k,l
```

In C we use `if` and `else`, along with scope delimiters `{ }`. Scope delimiters are used to specify how many instructions are to be included in the `if` part, and how many in the `else` part. In C, when there are no curly branches present following the `if` and/or `else`, then the scope is limited to the end of the next line delimited by `;`

More formally, we can say that code containing `if` and `else` has a *branch* structure. You may have seen *flowcharts*, which are useful in understanding the logical branches in a program. Let's look at the above C program in the form of a flowchart as shown in Fig 4.1

Although in a C program we use `if/else` keywords to create a branch in the instruction flow, in ARM7DTI assembly we use a combination of instructions to effect this. But before we look at the appropriate ARM instructions, let us think a little more deeply about `if` tests.

4.1.1 `if` test as subtraction

You may not have realised it, but all `if` tests can be thought of as subtraction operations. For example, suppose we have $a = 5$ and $b = 10$, then

1. `if a > b` is true if $a - b > 0$ which means the result is not zero and not negative
2. `if a >= b` is true if $a - b > 0$ or $a - b = 0$ which means the result is either zero or not negative

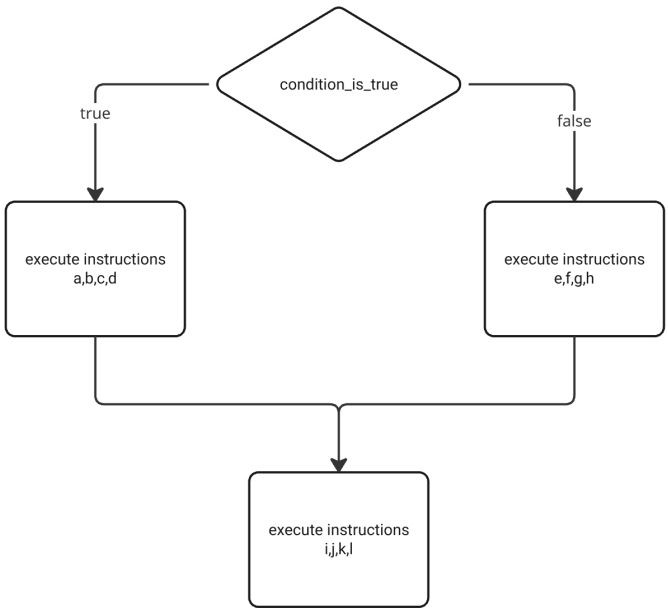


Figure 4.1: Flow chart for the C program showing two *branches*

3. `if a == b` is true if $a - b = 0$ which means the result is zero
4. `if a < b` is true if $a - b < 0$ which means the result is negative
5. `if a <= b` is true if $a - b < 0$ or $a - b = 0$ which means the result is negative or zero

Notice in the above that we use only two *indicators:* the zero indicator and the negative indicator. In C we don't have to test for these indicators, the compiler generates the code behind the scenes to do the subtraction. But in ARM7DTI assembly we must understand this process, because there is no direct `if/else` equivalent, there is only *subsection* of the operands and the *test* of the outcome. So next we look at these *indicators*. Where are they stored and how do we access them.

4.2 The Current Program Status Register

4.2.1 `NZCV` flags

The ARM7DTI processor has a dedicated register known as the Current Program Status Register `CPSR`, which is updated with the outcome status of mathematical operations. As mentioned above, this outcome status is (a) result was a negative, (b) result was a zero. We also add two new outcome status indicators (c) the result includes a *carry-out*, and (d) the result includes a *overflow*. Although the `CPSR` is a 32-bit register, only the last 4 least significant

bits (LSBs) are of interest to us. Collectively, these 4-bits are often referred to as the `NZCV` bits of the `CPSR` register.

The CPSR register cannot be directly read or written to by the programmer. (only the ALU can update the CPSR).

4.2.2 Carry-out

Let us briefly review how a carry-out result arises. For this we need only study the MSBs of the register, for example, in a 6-bit computer where we have $25 + -5$

$$\begin{array}{r} 011011 \\ +111011 \\ -\,-\,--\\ 1 \overset{\text{carry out}}{\longleftarrow} 010110 \end{array}$$

Note, the can say a carry-out occurs whenever we require one bit *beyond* the MSB to store the result. So in this case, the `NZCV` register is `0010`

4.2.3 Overflow

Although an overflow may seem to be similar to a carry-out, it is completely different. We get an overflow whenever we add two numbers of the same sign, and the result is a number of a different sign. For example, adding $-32 + -1$ in a 5-bit computer:

$$\begin{array}{r} 100000 \\ +111111 \\ -\,-\,--\\ 1 \overset{\text{carry out and overflow}}{\longleftarrow} 011111 \end{array}$$

Notice here that the sign changed - we added two negative numbers and the result was a positive number (+31). This is both an overflow and a carry-out. So in this case, the `NZCV` register is `0011`.

Overflow does not *always* result in a carry out. For example

$$\begin{array}{r} 010000 \\ +010000 \\ -\,-\,--\\ 100000 \end{array}$$

In the above example, we have $16+16 = 32$ so there is no carry-out (because 32 can be represented in a 6-bit word), yet the MSB bit has changed from positive to negative. It is important to note that, in this case, an overflow

does not necessarily mean an error. It is only an error if your calculations are *signed*. If your calculations are *unsigned* then the result is correct and there is no problem.

In conclusion, overflow arises when two operands of the *same* sign are added, and the result is a number of a *different* sign ($+n + +m = -s$ or $-n + -m = +s$)

If you are not interested in the operand signs, then the overflow event does not matter but carry-out certainly do matter.

4.3 Back to `if`

Now that we understand how the `NZCV` gets updated, and what it can be used for (ref. Table **??**), we now look at the ARM7DTI approach for implementing the humble `if`, by using the `cmp` instruction.

4.3.1 Compare and test

To implement `if` we use the `cmp` instruction followed by a test appended to the instruction that follows the `cmp` (see Table 4.1). `cmp` has two operands, the left and the right. It subtracts the right operand from the left operand and updates `NZCV` with the outcome of the operation. The actual numerical result is not stored anywhere.

In `cmp`, both the left and right operands can be registers (this is usually the way), or the left operand can be a register and the right operand an immediate. For example:

```
mov r1, #10
mov r2, #5
mov r3, #20
cmp r3, #20 // NZCV = 0110
cmp r2, r1  // NZCV = 1000
cmp r1, r2  // NZCV = 0010
```

Note that each of the above 3 `cmp` Instructions will update the `NZCV` flags. This means that the result will be overwritten and lost. Therefore, if we use `cmp`, we never follow it immediately with another `cmp`. We first must *examine* the `NZCV` and make a decision. Lets look at an example in C:

```
int i = 10;
int r = 5;

if (i > r) {
  r = i;
}
```

We can say that the `r = i` is a conditional operation - it should only happen if `i > r`. Let's do this in assembly:

```
mov r1, #10
mov r2, #5
cmp r1, r2
movgt r2, r1
```

Note the new version of `mov` here: `movgt` which means *move if greater than*. If what is greater than what? if `r1` is greater than `r2`. How is this determined? by subtracting `r2` from `r1` and checking the `NZCV = 0000` (not negative and not zero).

The full list of conditionals is shown in Table 4.1

Table 4.1: Full set of ARM7DTI conditional codes

Condition	Meaning	Flags Tested
`eq`	Equal (zero)	Z = 1
`ne`	Not equal (non-zero)	Z = 0
`cs/hs`	Carry set / Unsigned higher or same	C = 1
`cc/lo`	Carry clear / Unsigned lower	C = 0
`mi`	Negative (minus)	N = 1
`pl`	Positive or zero (plus)	N = 0
`vs`	Overflow set	V = 1
`vc`	Overflow clear	V = 0
`hi`	Unsigned higher	C = 1 and Z = 0
`ls`	Unsigned lower or same	C = 0 or Z = 1
`ge`	Signed greater than or equal	N = V
`lt`	Signed less than	N ≠ V
`gt`	Signed greater than	Z = 0 and N = V
`le`	Signed less than or equal	Z = 1 or N ≠ V

Almost all ARM7DTI instructions, with he exception of `cmp`, can optionally have *one* conditional code appended. The the conditional code is omitted then the instruction *always* executes.

For example:

```
addlt r2, r4 // add r4 to r2 NVCV = 1000
subeq r2, r4 // add r4 to r2 NVCV = 0110
mulhi r2, r3, r4 // multiply r3 by r4 if NVCV = 0010
```

4.3.2 Signed and Unsigned Conditions

To make the following examples easier to understand we will use a simple 3-bit computer. The ideas here apply equally to a computer of word size of any integer.

Let us think through the example of a signed *greater than* test. When we say *signed*, what this means is that *if* the number has the `MSB=1` it must be

treated as a negative number. We will later compare this to an unsigned test using the same bit pattern.

Let's look at the first example, using `gt` (for example, `movgt`). As you can see, `gt` is a signed greater than test, and for it to be true the `NZCV` flags must be: `Z=0` and `N=V`. The `Z=0` requirement is obvious. But less obvious is `N=V`. Let's look at a 3-bit example: `r0=-1` and `r1=2`.

When the instruction `cmp r0, r1` runs, we have $-1 + -2 = -3$.

$$\begin{array}{r} 111 \\ +110 \\ \hline 101 \end{array}$$

The result is correct, $101 = -3$. But do the flags agree with requirement for `gt`?

- `Z=0` true
- `N=V` false (negative but no overflow)

No - there is a disagreement, because the result is negative without an overflow. Therefore, $r0 < r1$ and the condition is false.

Therefore, we would expect the instruction `movgt r4, #5` to *not* execute.

How about reversing the test? `cmp r1, r0`. When it runs we have $2 - -1 = 3$.

$$\begin{array}{r} 001 \\ +010 \\ \hline 011 \end{array}$$

The result is correct, $011 = 3$. But do the flags agree with requirement for `gt`?

- `Z=0` true
- `N=V` true (no negative, no overflow)

Yes - there is agreement, as the result is not negative and not overflow. Therefore, $r1 > r0$ and the condition is true. Therefore, we would expect the instruction `movgt r4, #5` to execute.

Just for clarity, let us think again about the previous example from an *unsigned* point of view. Again, `r0=7` and `r1=2`, lets see what happens when we compare them and test if `r0` is `gt` `r1` (it is). This becomes $7 - 2 = 5$, or

$$\begin{array}{r} 111 \\ +110 \\ \hline --- \\ 1 \overset{\text{carry out}}{\longleftarrow} 101 \end{array}$$

Does this agree with the definition for unsgined greater than (`hi`)?

- `Z=0` true
- `Z=0` true (no overflow)

Yes, $7 > 2$. Rememeber that although for us, we think of these two numbers as unsigned, when `cmp r0, r1` runs, it multiplies the second operand by -1 and then adds it to the first operand. In unsigned arithmetic this always produces a negative second operand.

You can see the contradiction here: when `cmp r0, r1` is followed by `gt`, the result waa false (-1 is not greater than 2), but when we use `hi` we find the result is true! ($7 > 2$).

If our algorithm uses only unsigned arithmetic, then we should use `hi` and `ls` for our greater than and less then or equal to. Conversely, if we are using signed arithmetic, then use `le` and `gt` respectively.

4.3.3 Updating `NZCV`

`cmp` *always* updates the `cpsr` (`NZCV` bits) with the outcome of the instruction. But there are other instructions that can produce Zero, Negative, Carry and Overflow outcomes. For example, a calculation such $m - n > 0$ always results in a carry (`C=1`). How do we update `NZCV` ourselves? It can only be done *indirectly* - by appending `s` to the instruction in question. For example:

```
mov r0, #5
mov r1, #3
subs r0, r1  // results in C=1
sub r0, r1   // does not change NZCV
```

Notice in the example above, `subs r0, r1` should be understood as a substraction instruction that updates `NZCV` with the outcome of `r0` minus `r1` (from the ALU). Whereas `sub r0, r1` performs no such update. Therefore, when you want to make a decision about the outcome of an operation, append `s` to the instruction in question.

4.3.4 Combining condition codes and `NZCV` updates

Is it possible to combine both condition codes and `NZCV` updates in one instruction? Yes. For example: `addlts r0, r1` will update the `NZCV` with the outcome of the `addlt` so long as the *previous* instruction resulted in a signed `lt` outcome (`Z = 0` and `N = V`).

4.3.5 Branching

Now that we have seen how `if` is implemented in assembly as the pair `cmp` + *instr*`cond`, we should return to scope control. Let's look at an example: where 5 instructions should execute if the `lt` condition is true:

```
mov r0, #-1
mov r1, #2
mov r3, #0
cmp r0, r1
// the next 5 instructions
// only execute if lt is true
movlt r4, #5
movlt r5, #10
mullt r6, r5, r4
sublt r3, r6
movlt r5, r6
```

The above structure does not resemble how we write `if` statements in C. What we would like to do is to "jump" to the a block of code if the condition is true, and if it's false, then it should continue executing from the condition. This is the important role that *branching* plays. We will discuss branching in Chapter **??**.

4.4 Chapter Summary

In this chapter we have introduced the ARM7DTI equivalent of the C program `if` keyword. We noted that `if` in ARM7DTI is always at least*two* instructions: the `cmp` instruction followed by one or more instructions with the condition code appended (see Table 4.1).

In this chapter we discussed several examples of how signed and unsigned conditions are handled. We noted that in some cases, signed and unsigned tests of the same condition type (eg, `gt` and `hi`) produce different results. Rememeber that as although the bit patterns in memory are the same, the results in the both `gt` and `hi` are tested against different parts of the `NZCV`. We also looked at how the programmer can indirectly update `NZCV` (by appending `s` to the instruction), and we also briefly noted that condition codes and `NZCV` updates can be combined into one instruction such as `addlts r0, r1`.

4.5 Exercises Summary

Exercise 4.5.1 *Using three few different examples test to see if a carry-out always arises whenever we have a positive and a negative addition where the result is positive. Does it hold true?*

Exercise 4.5.2 *Derive an ARM7DTI assembly example where* `NZCV` $= 0011$ *(Positive result, with carry-out and overflow). Test it in CPUlator.*

Exercise 4.5.3 *Derive an ARM7DTI assembly example where* `NZCV` $= 1010$ *(Negative result, with carry-out, no overflow). Test it in CPUlator.*

Exercise 4.5.4 *Give informal examplea from the real world, where (a) an overflow would be a concern, and (b) where an overflow would* not *be a concern.*

Exercise 4.5.5 *Write and ARM7DTI program to demonstrate how* `lt` *and* `\lo` *conditional codes produce different results for the same bit pattern.*

Five

Branching and Iteration

5.1 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, 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 5.1: 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.

5.2 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.

5.3 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 5.2: 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 5.3: 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.

5.4 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:

```
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:

```
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:

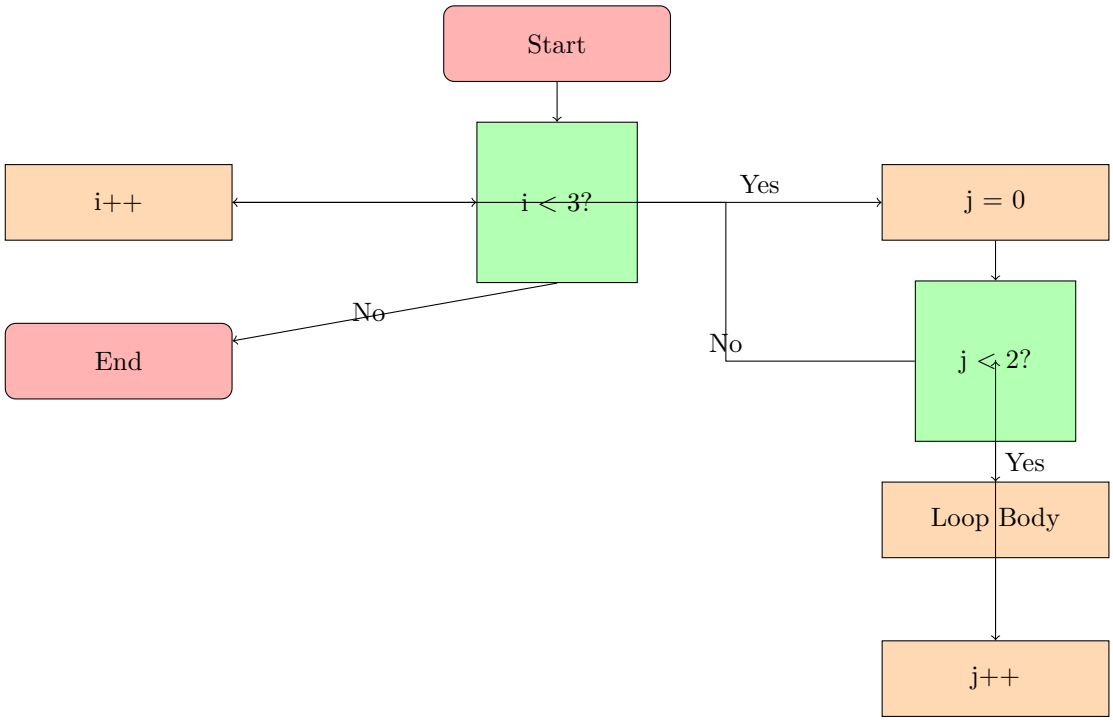


Figure 5.1: Flowchart of Nested Loops

5.5 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.

5.6 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
```

5.7 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.

5.8 Exercises

Basic Load and Store Operations

Exercise 5.8.1 *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 5.8.2 *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 5.8.3 *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 5.8.4 *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 5.8.5 *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 5.8.6 *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.*

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:

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

```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

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 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
```

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`

```
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.

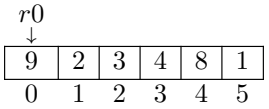
Initializing the base-register

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:



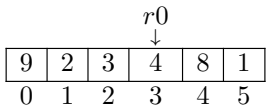
(6.1)

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.

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`

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]`)

```
// 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
```

Eliminating the incrementer (`r1`) and one instruction from the loop (`addlt r1, r1, #1`)

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

```
// 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
```

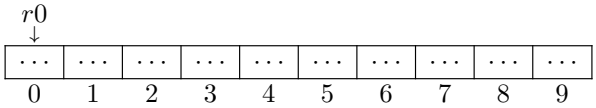
Of course, if you are writing a C program for the ARM platform, the compiler will convert `*ptr++ += 3;` *into the three ARM instructions as shown above.*

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
```

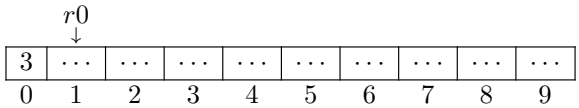


(6.3)

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
```

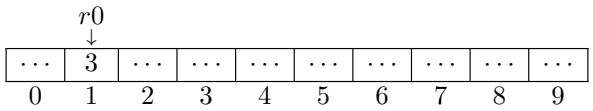
so now things look like this:

(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.

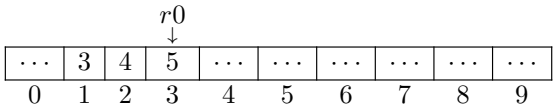
Stacks

Pre-index addressing is used to insert (or *push*) elements onto *stack*-type data structures. We discuss the details of stack-programming in Chapter **??**. 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"
```

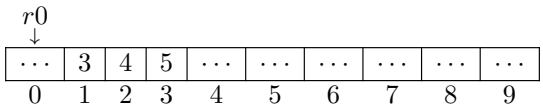


(6.7)

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++;
}
```

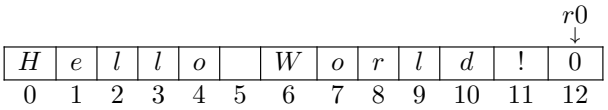
The ARM assembly equivalent is very similar - it must also test for the null character (ASCII 0). But because we are loading byte data and not integer data, we use the byte versions load and store instructions shown in Table 6.1:

```
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 alphanumeric 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.

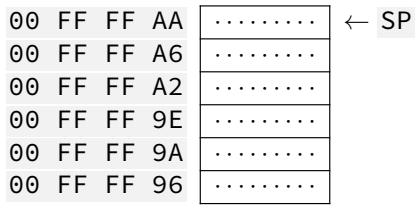
7.3 `caller` prepares parameters

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 ]!
```

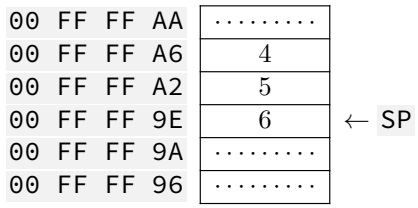
```
    mov r1, #6
    str r1, [ sp, #-4 ]!
```

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



(7.1)

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



(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.
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.

[1] why is this?

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:

7.6 `callee` SAVES `LR`

In Chapter 5 we discussed how execution flow can by controlled using the branching instruction `B`. We further refined this in Chapter **??** 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

```
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

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 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:

```
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 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 Fig. 8.1.

Directly using system calls from an application process is a difficult task. It requires the developer to execute a software interrept (`svc #0`) instruction,

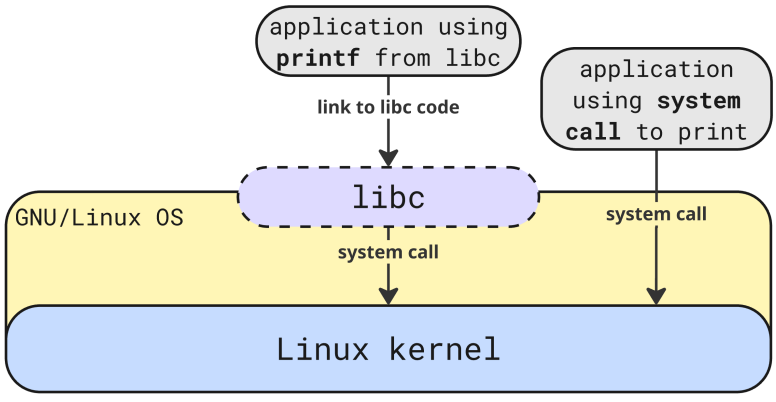


Figure 8.1: Application code can use either `libc` or system calls (`svc` to print to the screen)

```
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 8.2: Assembly code for printing to the screen using the system call instruction `svc`

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

Figure 8.3: C code for printing to the screen using `printf` from `libc`

along with populating specific registers correctly. For example, the system call to print `"Hello World\n"` to the screen shown in Fig 8.2, and as you can see, the implementation in `print` is rather complex for such as simple taks.

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. 8.2, we will have 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. 8.3, `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 do not invoke system calls directly, instead they call library code from `libc` (in Unix) in order print to the screen (for example). From

Fig. 8.1, our code is *linked* to the binary code contained in `libc` so that we can the implementations can be reused without the need to write our own implementation.

GNU/Linux `libc` contains over 400 functions [1] functions that can be called from user-layer applications. A complete discussion on `libc` is beyond the scope of this book. Readers interested in looking at `libc` in detail should refer to the reference `https://www.gnu.org/software/libc/`

However, we will discuss one `libc` example next. We will use the `printf` function - which has already been shown in Fig. 8.3. `printf` can be used to print a string along with any number of *arguments* to the string. See `https://man7.org/linux/man-pages/man3/printf.3.html`.

8.1.2 Printing to the screen using `printf`

To make the example more interesting, we will user `printf` to print the following information to the screen:

- the number of command line parameters to the program
- the name of the program
- the first parameter to the program

Assuming that the program is called `a.out`, the running the following command:

```
$ ./a.out 55
num params: 2, program name ./a.out, first param: 55
```

This example demonstrates some interesting features. First, notice that we need to obtain command line parameters the OS passes into our application. In C, we use the standard main function signature `int argv, char *argc[]` to obtain the command line parameters passed in. But how do we get the equivalent of `int argv, char* argc[]` in ARM assembly? The convention is that:

- `r0` contains the `argv`
- `r1` is a base-register which points to an array of `char*` pointers. This is the definition of `char* argv[]`
- The stack pointer `sp` is *not* used to pass items from the command line into `main`

Based on the code in Fig 8.4 we see how to integrate `printf` from `libc`. We will briefly discuss this solution next.

[1] excluding the _ functions. If we include these functions, this number rises to over 3000

```
main:
  // save the return address from main
  push {lr}

  // in ARM32 r1 is the base-pointer
  // to an array of pointers to strings
  // zero-ith parameter - program name
  ldr r2, [ r1 ]
  // first parameter - program parameters
  ldr r3, [ r1, #4 ]

  // if have n parameters they are
  // accessed as r1 + (n * 4)
  // we are finished accessing r1
  // so we can prepare it for printf
  mov r1, r0
  ldr r0, =output_string

  // use puts or printf from libc
  bl printf

  // set up mains return parameters
  mov r0, #0

  // now return from main
  pop {pc}

.data
  output_string:
    .asciz "num params: %d, program name %s, first
    param: %s\n"
```

Figure 8.4: Assembly code for printing to the screen using `printf` from `libc`

Discussion of Fig 8.4

Let's focus on these two instructions with line numbers:

```
1. ldr r2, [ r1 ]
2. ldr r3, [ r1, #4 ]
```

line 1: `r1` is dereferenced (`[r1]`) we get `argv[0]` (which is *always* the name of the program).

line 2: Using offset addressing to point to the next pointer and dereferencing (`[ r1, #4 ]`) brings us to `argv[1]`.

Next, we look at how to prepare parameters for passing information `libc` functions. To better understand how to prepare parameters into `libc` you must carefully review the function prototype from the man pages:

`int printf(const char *restrict format, ...);`

Notice that the first parameter to `printf` is the formatting string. In our case, this string is:

`num params:  %d, program name %s, first param:  %sn`

```
3. mov r1, r0
4. ldr r0, =output_string
5. bl printf
```

ARM32 has some specific rules around passing parameters into functions. For the first *four* parameters, registers `r0-r3` should be used. This means the stack is not involved. If there are more than four parameters, then all subsequent parameters should be pushed on to the stack from right-left ordering. However, here we have only 4 parameters, so `r0-r3` will be used, as follows:

1. line 3: place the value of `argv` into `r1`. (
2. line 4: set `r0` as the address of the formatting string
3. line 5: branch with link to `printf`

Graphically this is summarized in Fig 8.5, where the application (`a.out`) is invoked and the OS passes in command line parameters (in `r0` and `r1`). The application then process the parameters and sets up the registers correctly (`r0-r3`), then invokes `printf`

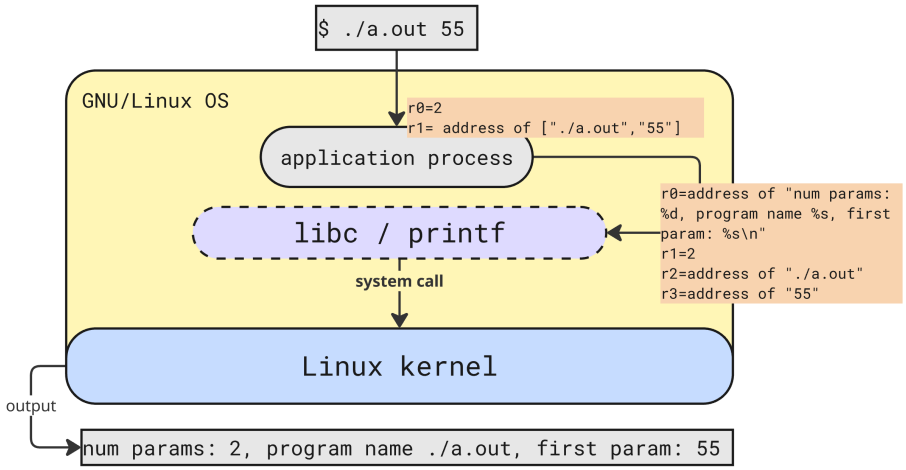


Figure 8.5: The application (`a.out`) is invoked and the OS passes in command line parameters (in `r0` and `r1`). The application then process the parameters and sets up the registers correctly (`r0-r3`), then invokes `printf`

Finally

Finally, note this code fragment:

```
push {lr}
// .. call to printf
mov r0, #0
pop {pc}
```

as we know, all lables (and this includes `main`), should save the link register (`push {lr}`) as the first instruction. Notice when the `printf` routine returs we set `mov r0, #0`. This is used by the OS to understand if our program completed successfully or not. [2]). Finally, the link register value is popped and placed in the program counter (`pop {pc}`). This is effectively how a return takes place. Of course, when `main` returns the program is finished executing.

How is the value in `r0` used? We can use the linux parameter `$?` to check:

```
$ ./a.out 55
num params: 2, program name ./a.out, first param: 55
$ echo \$?
0
```

When we run `echo $?` and get 0, we are receiving the 0 placed into `r0` prior to the return from `main`

[2]where a non-zero value is used to indicate an error

8.2 Chapter Exercises using `libc functions`

1. Write an assembly program to iterate over `argv` printing out each parameter (using `printf`) on a new line. For example:

```
$ ./a.out 55 66 77 88 99
param 1: 55
param 2: 66
param 3: 77
param 4: 88
param 5: 99
```

2. In `libc`, upper and lower case string manipulation is handled by `int toupper(int c)` and `int tolower(int c)` functions respectively. Using , write an assemply program to convert its use input from upper or mixed case, to all lower case. For example:

```
$ ./a.out MY NamE iS TOLOwer
lower_case: my name is tolower
```

3. The `libc` function `int atoi (const char *string)` takes a string pointer and converts it into an integer. For example, calling

```
int r = atoi("55")
```

would convert "55" to 55 in a C program. Write an assembly program that takes two parameters and adds them together, printing the result output:

```
$ ./a.out 55 65
Total is: 120
```

For this to work correctly you will need to use both `atoi` and `printf` from `libc`

4. Write an assembly language program to read from a the users name from the command line, and print a friendly greeting to the user. For example:

```
$ ./a.out
Hi! - what is your name? [ user enters Bob ]
Hi Bob, nice to meet you!
```

You will need to use `printf` and `gets` from `libs`

5. Write and assembly program to give the user 3 random numbers r_1, r_2, r_3 where $0 \leq r_k \leq 100$. You can store the random values in `r4-r6`. Print the results to the user. For example, here is some sample output:

```
$ ./a.out
Your 3 numbers are: 55, 22, 10
```

   You will need to use `int rand (void)` and of course, `printf`

Nine

Machine Code

So far in this book we have been focused on the *higher-level* challenges of writing selection, iteration, branching, accessing memory, reading and writing the stack, and so on. Of course, even assembly language program consists of textual instructions that a CPU simply does not understand. To the CPU, the only understandable input is a word-size binary (or hexadecimal) number.

So we begin this chapter by noting that **each and every line of code we write in ARM7 *must* fit into one single word (32-bit) field.**

So, for example, we can conclude that each and every line of code from this fragment (taken from Chapter 6) must fit into one 32-bit word in ARM7DTI:

```
ldr r0, =array
mov r1, #0
mov r2, #0
mov r3, #6

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

It is the objective of this chapter to first explain the process by which instructions (like those shown) above, can be encoded in a 32-bit field. As you will see, there is nothing mysterious about encoding instructions into *machine code*, if we follow some basic structural patterns. These patterns are based on the *class* of instruction: (a) data processing, (b) memory, and (c) branching.

We will discuss each of these instruction classes in turn, starting with encoding data processing instructions.

9.1 Recap - Compiling/Assembling

As you may remember from Chapter 2, the process by which our C code goes from text file to ELF executable is conceptually straightforward:

1. C text file is preprocssed to handle directives like `#include`

2. Next, the file is converted into an intermediate representation (IR by the *compiler*
3. The IR is the converted into machine code
4. Finally, external code or paths are linked into the ELF output, and the file is ready.

When transforming an assembly program to executable ELF format, there are fewer steps. There is no IR phase, and the assembly program is processed thus:

1. The assembly is the converted into machine code by the *assembler*
2. External code (or paths in the case of shared libraries) are linked into the ELF output, and the executable file is ready.

In the above example we had eight instructions. At the end of the process there will be 8 32-bit instructions in our ELF file too, although there will also be a lot of set-up and configuration instructions added by the assembler (/compiler) and the linker. However, *conceptually*, our 8 instructions in a text file will result in 8 executable 32-bit fields in our ELF file (again, assuming we are on the ARM7DTI platform).

So our 8 instructions in ELF format will look like:

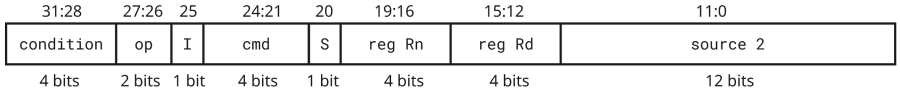
```
e59f0018
e3a01000
e3a02000
e3a03006
e1510003
b7802101
b2811001
bafffffb
```

or in binary format:

```
11100101100111110000000000011000
11100011101000000001000000000000
11100011101000000010000000000000
11100011101000000011000000000110
11100001010100010000000000000011
10110111100000000010000100000001
10110010100000010001000000000001
10111010111111111111111111111011
```

Now that we know *what* is happening, we will focus on *how* it happens: how a text instruction (eg `ldr r0, =array`) is converted into a binary representation (in this example, `11100101100111110000000000011000`)

(a) General DP format:



(b) `mul{cond}{s}` format:

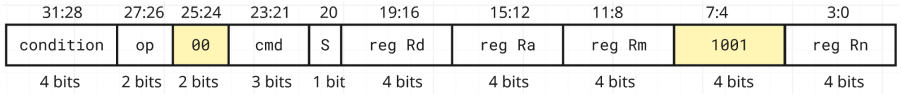


Figure 9.1: The structure of a 32-bit data processing instruction. (a) is the general form, (for all DP instructions excluding `mul{cond}{s}`), 12 bits of space are allocated for the `source 2` field discussed later. (b) is the structure for a `mul{cond}{s}` and its associated variants (eg, `mula`). Once again, notice that fields in yellow are hard-coded by the assembler and cannot be modified by the programmer.

9.2 Data Processing Instructions

The class of *data processing* instructions are those whose job it is to move and operate on data. For example, these are the all the `mov{cond}{S}` variants, along with all mathematicl and logical instructions (the more common of which are shown in Table 3.1).

In some ways, data processing instructions are the most complex in terms of their 32-bit binary encoding. The encoding must allow for a great deal of variation in instruction format, for example:

```
mla r0, r1, r2, r3
addlts r4, r5, r6
and r5, r6, #1
sub r7, #4
```

Notice variation here: `sub r7, #4` and `mla r0, r1, r2, r3` for example. Both of these instructions must be encoded using the data processing encodinig layout that we will discuss next. How this happens is very interesting.

9.2.1 Instruction Layout

We first begin by reviewing the structure of a 32-bit data processing instruction, the layout of which is found in Fig. 9.1.

There are *two* classes of Data Processing instructions: the non-multiply functions and the multiply functions. These are structured according part (a) or part (b) in Fig. 9.1. Note that in the case of the multiply instructions, there are two hard-coded parts (shown in yellow) that the assembler determines.

The non-multiply instructions

There are six different parts to the non-multiply instruction. We discus each of them in turn, from the most significant bit (31) to the least significant bit (0). These are discussed in Table 9.1.

Bit from:to	**Name**	**Meaning**
31 : 28	`cond`	the *condition* code for the instruction. The full list is shown in Table 9.2
27 : 26	`Op`	the *operation* code for the instruction. As follows: 1. `00` Data Processing 2. `01` Memory 3. `10` Branching
25	`I`	the *immediate* bit. Where $I = 1$ the instruction contains an immediate value (for example `add r0, r1, #5`) and where $I = 0$ there is no immediate (for example, `add r0, r1, r2`, `add r0, r1, r2, lsl #2`)
24 : 21	`cmd`	the operation to perform. In the case of a data processing instruction, with 4 bits for `cmd` there are only 16 possible variants as shown in Table 9.2
20	`S`	the *update-flags* bit, when $S = 1$ the output form the operation (for example, a negative number) is updated to the `NZCV` condition register (eg, `adds r0, r1, #-10`). Where $S = 0$ the register is not updated (eg, `add r0, r1, #-10`)
19 : 16	Register `Rn`	the *first source register*. For example, in the instruction `add r0, r1, r2`, `Rn` is `r1`
15 : 12	Register `Rd`	the *destination register*. For example, in the instruction `add r0, r1, r2`, `Rd` is `r0`
11 : 0	`src2`	the second *source* parameters. There are 3 possible structures for `src2`, depending on whether the instruction uses (a) an immediate, for example `add r0, r1, #2`, (b), a immediate shifted register, for example `add r0, r1, r3, lsl #2`, or (c) a register shifted register, for example `add r0, r1, r3, lsl r4` See Fig. 9.2 for a more details

Table 9.1: Data Processing structure for the *non-multiply* instructions

The nomultiply instructions

In contrast to the non-multiply instructions, ARM7DTI multiply instructions have 10 different parts, two of which are hard-coded by the assembler and

Code	Meaning
0000	Equal to (eq)
0001	Not equal (ne)
0010	Carry set (cs/hs)
0011	Carry clear (cc/lo)
0100	Negative (minus) (mi)
0101	Positive or zero (plus) (pl)
0110	Overflow set (vs)
0111	Overflow clear (vc)
1000	Unsigned higher (hi)
1001	Unsigned less than or same (ls)
1010	Signed greater than or equal to (ge)
1011	Signed less than (lt)
1100	Signed greater than (gt)
1101	Signed less than or equal to (le)
1110	Unconditional - always execute

Code	cmd
0000	and
0001	eor
0010	sub
0011	rsb
0100	add
0101	adc
0110	sbc
0111	rsc
1000	tst
1001	teq
1010	cmp
1011	cmn
1100	orr
1101	mov
1110	bic
1111	mvn

Table 9.2: On the left, is the condition codes for 31 : 8 (there is no 1111 pattern). On the right, the structure for `cmd` (24 : 21)

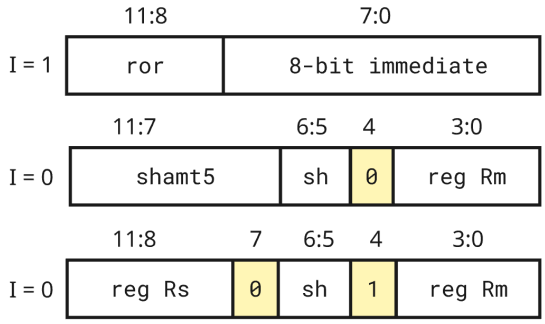


Figure 9.2: The structure of the 12-bit `src2` field in a data processing instruction. There are the 3 possible variants discussed in the text. Note, fields coloured with a yellow background are *hard-coded* values and cannot be changed by the programmer.

cannot be modified by the programmer. Again, we discus each of them in turn, from the most significant bit (31) to the least significant bit (0). These are discussed in Table 9.3. Notice that these instructions do not have any `src2` sections.

Bit from:to	Name	Meaning
31 : 28	`cond`	the *condition* code for the instruction. The full list is shown in Table **??**
27 : 26	`Op`	the *operation* code for the instruction. As follows: 1. `00` Data Processing 2. `01` Memory 3. `10` Branching
25	`I`	the *immediate* bit. Where $I = 1$ the instruction contains an immediate value (for example `add r0, r1, #5`) and where $I = 0$ there is no immediate (for example, `add r0, r1, r2`, `add r0, r1, r2, lsl #2`)
24 : 21	`cmd`	the operation to perform. In the case of a data processing instruction, with 4 bits for `cmd` there are only 16 possible variants as shown in Table **??**
20	`S`	the *update-flags* bit, when $S = 1$ the output form the operation (for example, a negative number) is updated to the `NZCV` condition register (eg, `adds r0, r1, #-10`). Where $S = 0$ the register is not updated (eg, `add r0, r1, #-10`)
19 : 16	Register `Rn`	the *first source register*. For example, in the instruction `add r0, r1, r2`, `Rn` is `r1`
15 : 12	Register `Rd`	the *destination register*. For example, in the instruction `add r0, r1, r2`, `Rd` is `r0`
11 : 0	`src2`	the second *source* parameters. There are 3 possible structures for `src2`, depending on whether the instruction uses (a) an immediate, for example `add r0, r1, #2`, (b), a immediate shifted register, for example `add r0, r1, r3, lsl #2`, or (c) a register shifted register, for example `add r0, r1, r3, lsl r4` See Fig. 9.2 for a more details

Table 9.3: Data Processing structure for the *non-multiply* instructions

9.2.2 Worked Examples

Need to discuss this move example because it uses the `and` bit pattern 0000
`mul r1, r0, r3` `e0010390` `1110 00 0 0000 0 0001 0000 0011 1 00 1 0000`

Ten

Single Cycle Microarchitecture

Eleven

Multi Cycle Microarchitecture

Twelve

Pipeline Microarchitecture

Thirteen

CPU Caches

Fourteen

Virtual Memory

Fifteen

Parallel Architectures