Embedded Systems
Tufts University, Spring 2024

Prof. Chang Prof. Bell Notes by Daniel Siegel*

May 7, 2025

* siegel.bio

Course Overview 1

These notes were taken by Daniel Siegel using TeXpress, Overleaf, and an LLM-based note input tool. They document the material covered in the Embedded Systems class at Tufts University, focusing on key concepts, examples, and problem-solving techniques for reference and study.

Course Overview

This course provides a comprehensive understanding of embedded systems, including:

- Hardware concepts: Microprocessor architectures, bus structures, memory systems.
- Embedded software: Memory management, efficient hardware-aware algorithms, real-time systems.
- Practical applications: Interfacing, mixing assembly and high-level programming, interrupt systems, and I/O.

The course includes laboratory work for hands-on experience in software and hardware design. **Prerequisites:** ES4 and COMP11 or instructor permission.

Course Information

- **Instructor:** Prof. C. Hwa Chang (hchang@ece.tufts.edu), Office: Halligan 132, Phone: (617) 627-5178.
- **Instructor:** Prof. Steven Bell (Steven.Bell@tufts.edu), Office: Halligan 112, Phone: (617) 627-3217.
- **Teaching Assistants:** Led by Sam Sugarman with a team of assistants for labs and office hours (details below).
- **Class Schedule:** Tuesdays and Thursdays, 3:00 PM – 4:15 PM (JCC 170).
- **Textbook:** *Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C*, 4th Ed., Yifeng Zhu.
- **Course Website:** Assignments, handouts, and announcements available on Canvas: login.canvas.tufts.edu.

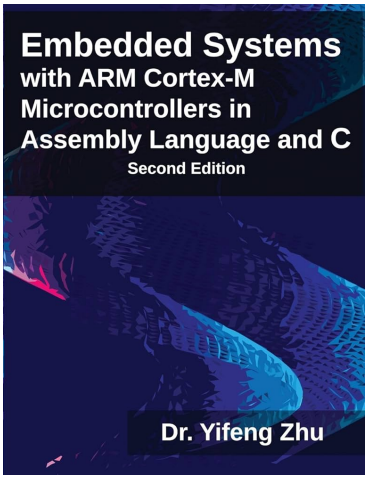


Figure 1.1: Class Textbook.

Topics and Schedule

The course includes the following topics:

1. Microprocessor architectures, bus structures, and memory maps.
2. Assembly programming and structured programming.
3. I/O systems, subroutines, and stacks.
4. Mixing assembly and C, interrupts, and PWM.
5. Final topics: Step motor control and embedded system applications.

Quizzes: Weekly in-class quizzes on Thursdays; the top six scores count towards your grade. **Homework:** Weekly assignments due Tuesdays; solutions posted on Canvas. Late submissions carry penalties.

Lab and Final Project

- Four labs and one final project are required.
- Labs start the week of January 21, held in Halligan 109.
- Work in pairs or independently, with no partner changes without permission.
- A report is required from each student, and all teams must prepare slides to present their final projects.

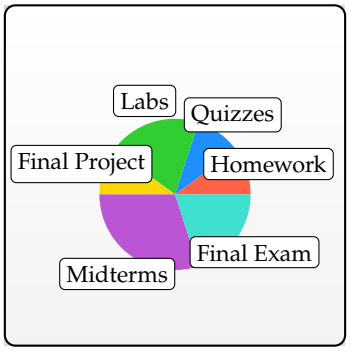
Exams and Grading

Exams: Two midterms and one comprehensive final exam.

- Midterm 1: March 3rd, 11:50 AM – 1:20 PM.
- Midterm 2: March 31st, 11:50 AM – 1:20 PM.
- Final Exam: May 7th, 3:30 PM – 5:30 PM.

Grading Breakdown:

- Homework: 10%
- Quizzes: 10%
- Labs: 20%
- Final Project: 10%
- Midterms: 30% (15% each)
- Final Exam: 20%



Policies and Support

Academic Integrity: Adhere to Tufts' Academic Integrity policies. Work may be reviewed via TurnItIn to ensure originality. For more information, see Tufts Academic Integrity Handbook. **Health and Safety:** Mask-wearing is optional but encouraged. Updates will follow university guidelines: Healthy@Tufts. **Accommodations:** Contact the StAAR Center (StaarCenter@tufts.edu) for academic accommodations.

Additional Notes

- Tutoring may be available through HKNtufts@gmail.com.
- Study groups are highly recommended.

Introduction 2

Topics Covered in ES4 (prerequisite class)

1. Number Systems

Understanding different base systems:

- Base 2: Binary
- Base 8: Octal
- Base 16: Hexadecimal

2. Combinational Logic Circuits

Logic Gates:

- AND, OR, NOT
- NAND, NOR

Logic Devices:

- Decoder
- Encoder
- Multiplexer

3. Sequential Logic Circuits

Registers and Applications:

- Storage elements for data retention.
- Used in state machines and counters.

4. Digital Design Tools

VHDL (VHSIC Hardware Description Language):

- VHSIC: Very High Speed Integrated Circuits
- Enables the modeling and simulation of digital systems.

Types of Computers

1. **Supercomputers:** Used for high-performance computing applications.
2. **Servers:** Centralized computing resources for networks.
3. **Desktop/Laptops:** General-purpose computing devices.
4. **Embedded Systems:** Specialized computers integrated into devices.
5. **Quantum Computers:** Emerging technology, especially significant for AI applications.

Fact: There are over 2.4 billion tablets and smartphones in use globally, not including other embedded device applications.

Quantum Computers and Traditional Systems

Integration: Some quantum computers can be integrated with traditional computing systems, enhancing their versatility and computational power.

Embedded Systems Overview

Embedded systems combine hardware and software to perform a specific function as part of a larger system. The components of embedded systems include:

Components of Embedded Systems

1. **Embedded Hardware Design:** The physical components such as microprocessors, sensors, and actuators that enable functionality.
2. **Embedded Software Design:** The programming and logic implemented to control hardware and execute tasks efficiently.
3. **Real-Time Embedded Systems:** Systems designed to process data and respond within strict time constraints, critical for time-sensitive applications.
4. **Embedded Operating Systems:** Lightweight operating systems specifically optimized for embedded environments, managing hardware and software resources efficiently.

Examples of Embedded Systems in Larger Systems

Embedded systems often function as integral parts of larger systems. A common example is the automobile, where embedded systems handle tasks such as:

- Navigation systems
- Airbag control
- Traction control
- Climate control
- Audio/Video control

These subsystems improve functionality, safety, and user experience.

Embedded Real-Time Operating Systems (RTOS)

An important feature of some embedded systems is the inclusion of an Embedded Real-Time Operating System (RTOS). These systems ensure precise and predictable task execution, making them vital for applications requiring immediate responses.

Examples of Embedded Systems in Action

1. **Automobiles:** Embedded systems in cars handle tasks such as airbag deployment, navigation systems, traction control, and climate control.
2. **VR/AR Glasses:** These rely on embedded systems to process real-time data, track user movements, and provide immersive virtual or augmented reality experiences.
3. **Drones:** Embedded systems enable navigation, flight control, obstacle avoidance, and real-time video transmission.
4. **Medical Devices:** Examples include pacemakers, insulin pumps, and imaging systems, which rely on embedded systems for precise and reliable operation.
5. **Amazon Warehouse Systems:** Warehouses such as those using the Kiva Picking system employ robots with embedded systems to transport and organize packages efficiently. **Amazon Robotics:** Originally developed by Kiva Systems, this technology was acquired by Amazon to optimize warehouse operations globally.

Basic Computer Systems

A basic computer system consists of five main parts:

1. **Arithmetic & Logic Unit (ALU) plus Register File:** These form the **Data Path**, responsible for performing arithmetic and logical operations and temporarily storing data. [1]
2. **Control Unit (CU):** The **Control Path** manages the sequencing of operations and controls data flow within the system. [2]
3. **Memory System:** Stores programs (codes) and data necessary for execution. [3]
4. **Input System:** Accepts data and instructions from the outside world for processing. [4]
5. **Output System:** Displays or transmits processed data to the outside world. [5]

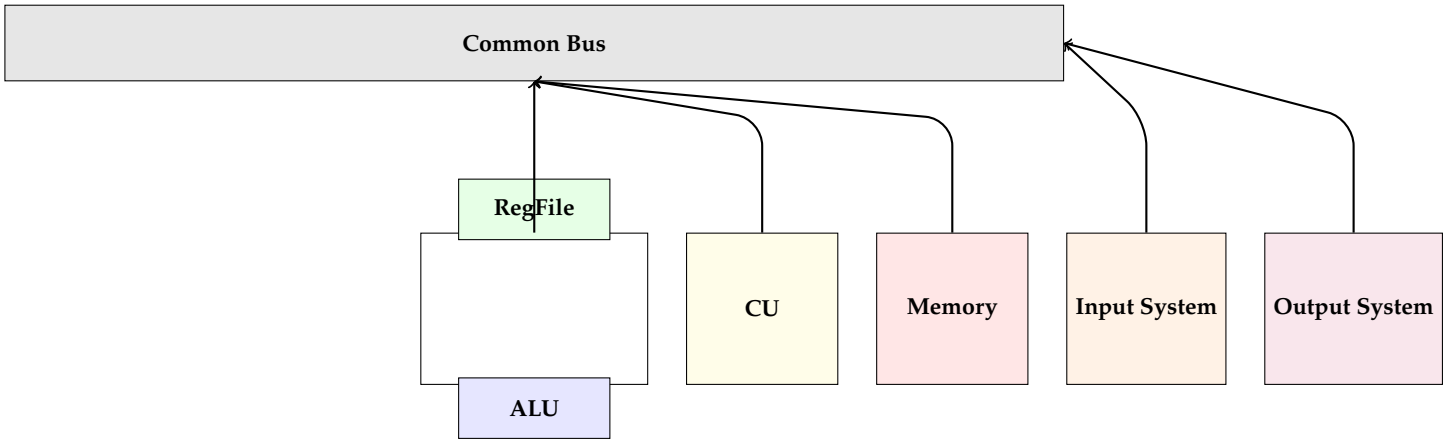
1: Processor (CPU)

2: Also Processor

3: Programs (Codes) and Data

4: I/O System
I/O Integration and Devices

5: Also I/O System



Building a Key in ES4

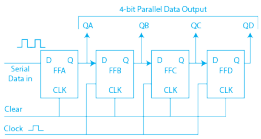


Figure 2.1: Registers Diagram.

Key Concepts:

- **Sequential:** Has memory; all previous inputs count.
- **Conditional:** No memory; only present inputs count.
- **Registers:** Features include Reset, Load, Shift, and Rotate.

Types of Registers in a CPU

1. General Purpose Registers: These registers are used for various operations and typically act as operands in arithmetic and logical instructions.

2. Special Purpose Registers: Specialized registers designed for specific tasks within the CPU. For example:

- **PC (Program Counter):** Points to the next instruction to be executed.

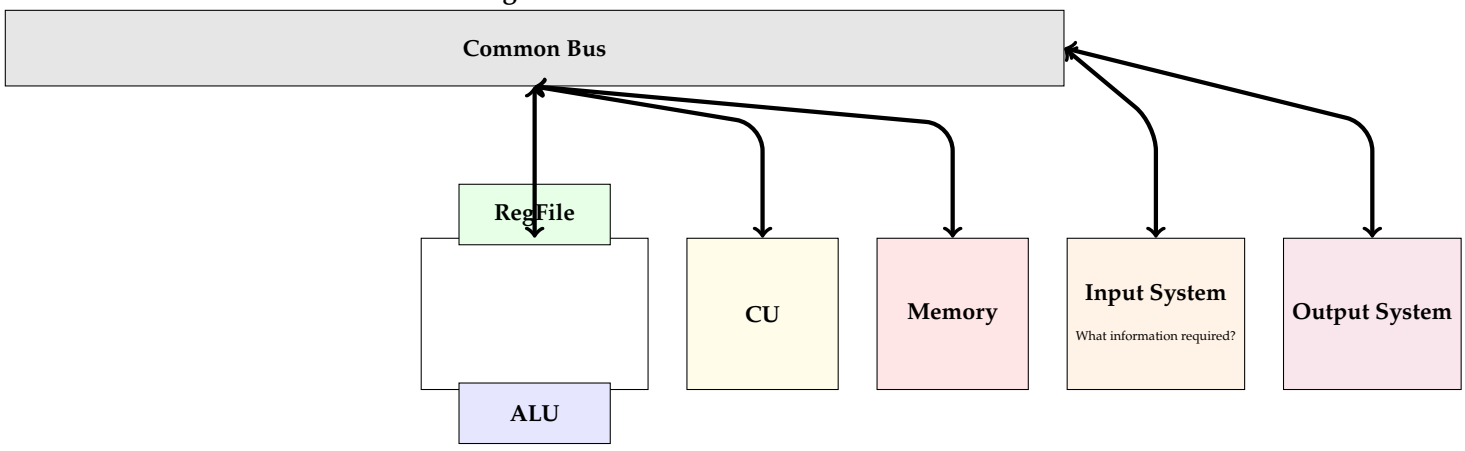
Next Instruction Location: The next instruction is stored in memory and is fetched by the CPU as indicated by the Program Counter (PC).

Control Path

Control Path:

- Contains the hardware instruction logic.
- Decodes and manages execution of instructions fetched.
- Acts as an arbiter for all five parts, including itself.
- **To generate control signals which coordinate all 5 parts, fetches from memory.**

Back to the Common Bus Diagram:



Signal Types

Signal Types

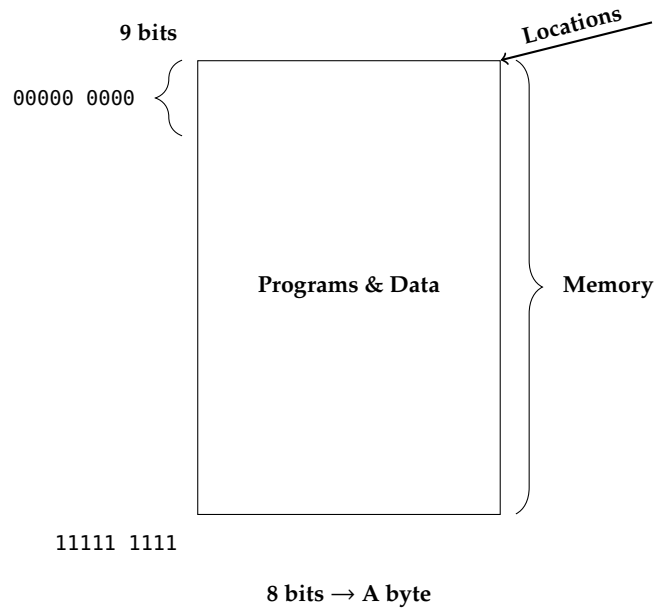
1. **Address Signal:** Sent out through the input to identify the device for both input and output operations.
2. **Data Signals:** Carry the actual data being transferred between the CPU, memory, and I/O devices.
3. **Control Signals (Both Ways):** Control signals from the CPU direct device operations, while I/O devices send signals back to registers in the CPU.
4. **I/O Device Status Signal:** Indicates the status of an I/O device, such as whether it is ready, busy, or requires attention. These signals are sent back to the CPU, not outward.

Memory Systems (From the Perspective of System Software)

Von Neumann Architecture: Program and data share a single memory **space**.

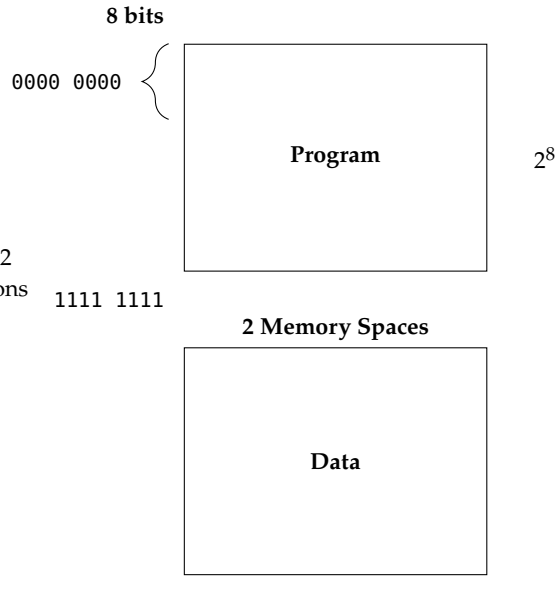
Harvard Architecture: Program and data are stored in separate memory **spaces**.

Von Neumann



$= 2^9 = 512$
# of Locations

Harvard



2^8

Memory Systems (From the Perspective of Hardware)

1. Volatile Memory

Memory that cannot retain information without power. Examples include:

- **RAM (Random Access Memory):** Temporary storage for active processes.
- **DRAM (Dynamic RAM):** Typically uses capacitors to store bits of data.
  - Capacitors discharge over time, causing data loss.
  - Requires periodic refreshing to maintain stored data.
- **SRAM (Static RAM):** Made of **FFS (Flip-Flop Structures)** instead of capacitors.
  - Does not require refreshing.
  - Faster and more reliable than DRAM but more expensive.

2. Non-Volatile Memory

Memory that retains information even when power is removed. Examples include:

- Flash memory (e.g., USB drives, SSDs).
- ROM (Read-Only Memory) and its variations (PROM, EPROM, EEPROM).

Examples of Non-Volatile Memory

- **ROM (Read-Only Memory):** A type of memory pre-programmed with data that cannot be modified during normal operation.
- **PROM (Programmable ROM):** Programmable once by the user and cannot be reprogrammed or erased.[6]
- **EPROM (Erasable PROM):** Can be erased using UV light and reprogrammed.[7]
- **EEPROM (Electrically Erasable PROM):**
  - Does not require physical removal for programming (in-circuit programming).
  - Commonly used in memory sticks and other flash-based devices.
- **Flash Memory:** A type of EEPROM that programs data block by block (typically from 64 bytes to 512 KB), unlike EEPROM, which is programmed byte by byte.[8]

6: Obsolete: Out-of-Circuit Programming

7: Also Obsolete

8: In-Circuit Programmable

ARM and Microprocessor Basics | 3

ROM Basics

In this section, we explore the fundamentals of a 4-bit decoder and its application in Read-Only Memory (ROM).

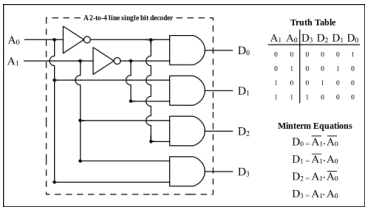


Figure 3.1: 4-Bit decoder.

4-Bit Decoder

A $2^n \times n$ decoder assigns each node a state of 0 or 1. For example, with 4 nodes and $n = 2$ (using two logic gates A_0 and A_1), we can represent $2^2 = 4$ states. If extended further to a $2^n \times 2$ configuration, this gives $2^2 \times 2 = 8$ total states across the nodes.

PROM (Programmable Read-Only Memory)

A PROM matrix, illustrated to the right, consists of programmable states at the intersections of memory lines.

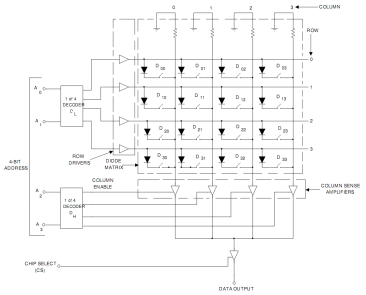


Figure 3.2: PROM Matrix.

Memory Layout

- **Address Lines**: A_1 and A_0, used to identify memory locations.
- **Output Lines**: D_2, D_1, and D_0, represented by three OR gates at the bottom.

The ROM consists of 4 memory locations (referred to as "words"), which can store the following contents:

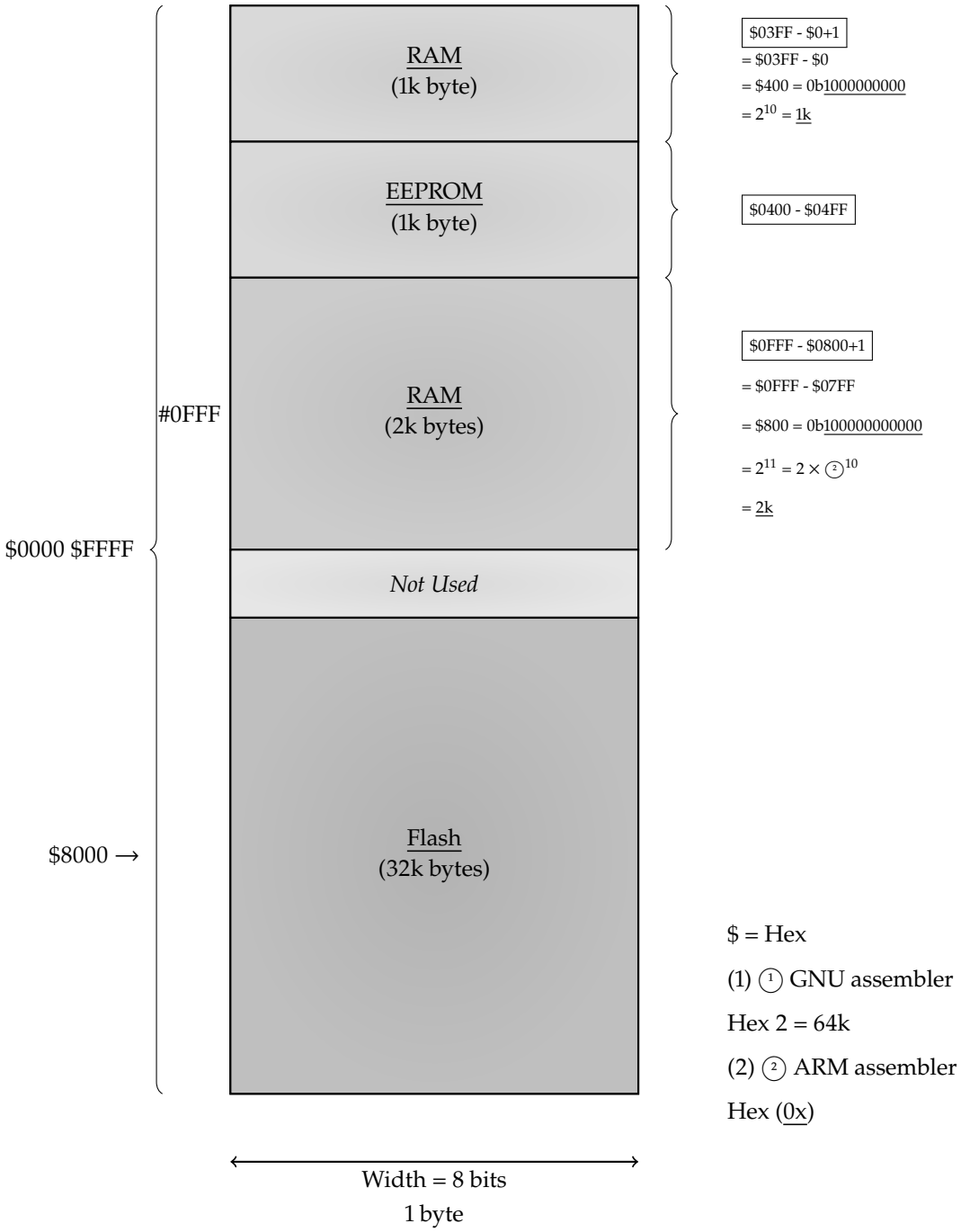
Word Address	D_2	D_1	D_0
0	1	0	1
1	0	1	1
2	1	0	0
3	1	1	1

- **Locations**: 2^n locations; for $n = 2$, we have $2^2 = 4$ locations.
- **Address Lines**: n lines correspond to 2^n addressable locations.
- **Width**: The width of the memory (number of D lines) is 3 (D_2, D_1, D_0).
- **Length**: The memory length refers to the number of memory locations (e.g., 0, 1, 2, 3).

To decode the memory addresses, an $n \times 2^n$ decoder is required. For $n = 2$, this translates to $2^2 = 4$ gates.

The circuit shown is a **Combinational Circuit**.

Why? All outputs are determined solely by the current inputs, with no dependency on past states.



- **Memory Space**: $64k \times 8$ bits
- **Address Bus**: 16 address lines ($A_{15} - A_0$)
  - $2^{16} = 64k$ addressable locations
- **Data Bus**: 8 data lines
- **Control Bus**: Includes lines for:
  - Write
  - Read
  - Additional control signals (e.g., chip select)
- **Decoder Requirements**: 16×2^{16} decoder needed

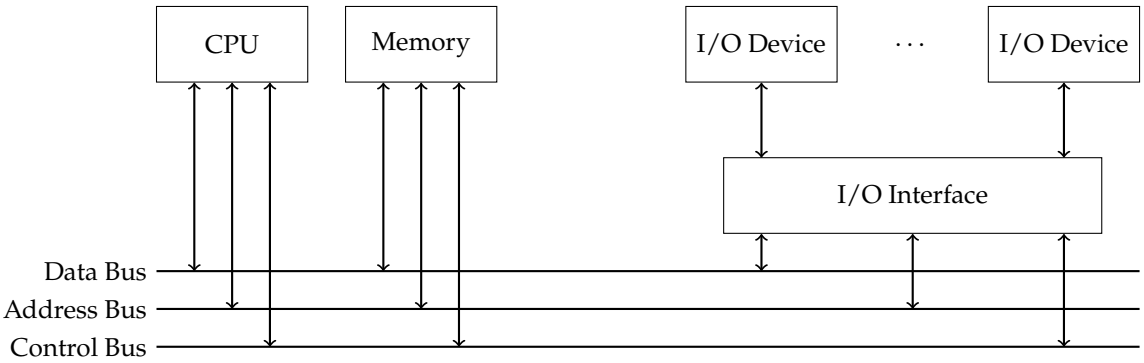
Memory Devices

- **RAM**: Used for temporary data storage, volatile.
- **ROM**: Read-only memory, non-volatile, stores fixed instructions.
- **Flash Memory**: Non-volatile, erasable, used for firmware storage.

Total memory devices typically depend on system architecture, but these three are commonly found in embedded systems.

Common Bus

- **Address Bus**: Identifies a piece of data in the register file, memory, or I/O device.
- **Data Bus**: Contains instructions or data (operands).
- **Control Bus**: Contains signals which coordinate the system's operations.



(I) Directions of lines
(II) How an instruction gets executed?

Instruction Execution Workflow

① **Program Counter (PC):** The PC holds the current address. The CPU sends the address (content of PC) to the memory.
② **Memory Handshake:** The memory acknowledges the request from the CPU and exchanges data.
③ **Instruction Execution:** The CPU increments the PC to point to the next instruction, executes the current instruction, and, on certain special instructions, stores the result in the register F'_f.
④ **Special Instructions:** Two special instructions:

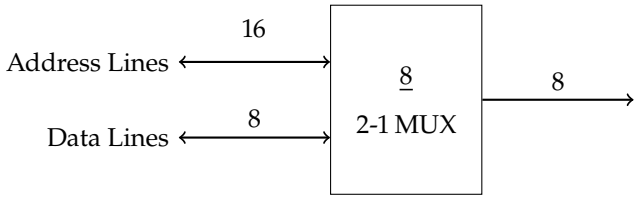
- **STORE**: Saves a value to memory.
- **LOAD**: Retrieves a value from memory.

Bus Lines

A bus consists of several lines, which can be used for different purposes.

- Example:
  - **8-bit data lines**
  - **16-bit address lines**
- The physical lines can be **multiplexed** (shared) to reduce the total number of required lines.

- For instance, **24 bus lines** can be implemented using only **16 physical lines**.



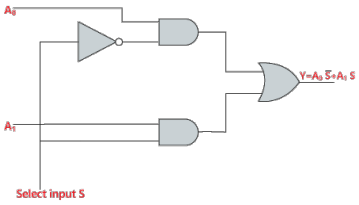


Figure 3.3: Multiplexer Diagram.

ARM: Acron RISC Machine

- **1981**: Acron RISC Machine (ARM) introduced.
- **1990**: Advanced RISC rebranded.

① RISC: Reduced Instruction Set Computer

- **Load/Store Architecture**: Operations occur only between registers.
- **Simple Instruction Set**: Optimized for fast execution and reduced complexity.
- Instructions executed by F'_f (register operations).

② CISC: Complex Instruction Set Computer

- **Powerful Instructions**: Capable of executing powerful operations in fewer instructions.
- **Multiple Memory Access Methods**: Offers flexibility for accessing and manipulating memory.

Why ARM Processors?

- **2005**: 98% of the more than one billion mobile phones sold used ARM processors.
- **2009**: ARM processors accounted for approximately 90% of all embedded 32-bit RISC processors.
- **2010**: **6.1 billion** ARM-based processors sold, representing:
  - 95% of smartphones
  - 35% of digital televisions and set-top boxes
  - 10% of mobile computers
- **2014**: Over 50 billion ARM processors produced.
- **2017**: 100 billion ARM processors produced.
- **2021**: 200 billion ARM processors produced.
- **2024**: 300 billion ARM processors produced.
- **2025**: 100 billion AI-ready ARM processors expected.

Figure 3.4: Example of an ARM chip.

- **ADC**: Analog-to-Digital Converter
- **DAC**: Digital-to-Analog Converter
- **USART**: Universal Synchronous/Asynchronous Receive/Transmit
- **2023**: NVIDIA attempted to purchase ARM.

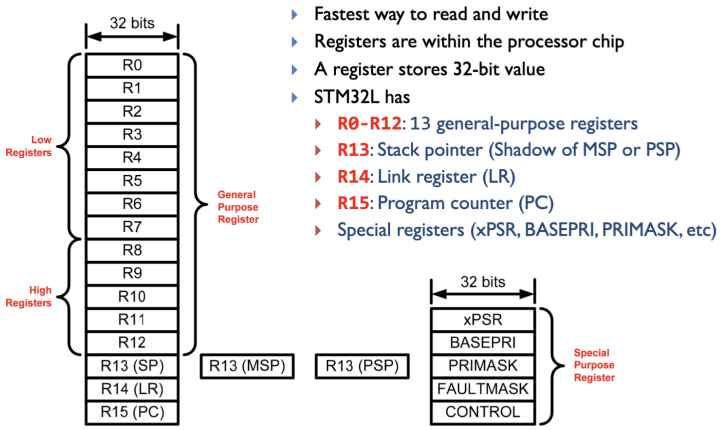
- **RISC-V**: An open-source platform started in 2010.

- **2025**: Projected to reach a 1.41B dollar market.
- **ARM Business Model**: ARM sells intellectual property (IPs) and does not manufacture IC chips.

A: Applications (performance intensive)

R: Real Time

M: Microcontroller (wide range of embedded systems)



32-bit Architecture Overview

- **32-bit Architecture:**
  - Includes 16 general-purpose registers used as operands.
  - Each register is addressed using 4 bits (16 $\rightarrow$ 4 bits address).
- **Register Usage:**
  - Typically, the low registers are used as operands.
  - This means **3 bits** are used to select the operand.
- **Data Flow:**
  - **Read:** Data is read from registers.
  - **Write:** Data is written to registers.
  - Physically, all operations occur on the **same register**.

Memory Architecture | 4

Memory Systems

1. Volatile:
DRAM, SRAM

2. Non-Volatile:

$$\left\{\begin{array}{l}\text{RAM}\\ \text{PROM}\\ \text{EPROM}\end{array}\right\} \textit{Life: 1,000,000 writes}$$

$$\left\{\begin{array}{l}\text{EEPROM}\\ \text{Flash}\end{array}\right\} \textit{Life: 100,000 writes}$$

Memory

Memory is arranged as a series of "locations":

- Each location has a unique **"address"**.
- Each location holds a byte (**byte-addressable**).
  - *e.g.* The memory location at address `0x080000180` contains the byte value `0x70`, i.e., 112.
- The number of locations in memory is limited.
  - *e.g.* 4GB of RAM.
  - 1 Gigabyte (GB) = 2^{30} bytes.
  - 2^{32} locations $\rightarrow$ 4,294,967,296 locations!
- Values stored at each location can represent either **program data** or **program instructions**.
  - *e.g.* The value `0x70` might be the code used to tell the processor to add two values together.
- 1kilo$(KB) = 2^{10} \approx 10^3$
- 1mega$(MB) = 2^{20} \approx 10^6$
- 1giga$(GB) = 2^{30} \approx 10^9$
- 1tera$(TB) = 2^{40} \approx 10^{12}$
- 1peta$(Peta) = 2^{50} \approx 10^{15}$
- 1exa$(Exa) = 2^{60} \approx 10^{18}$
- 1zetta$(Zetta) = 2^{70} \approx 10^{21}$

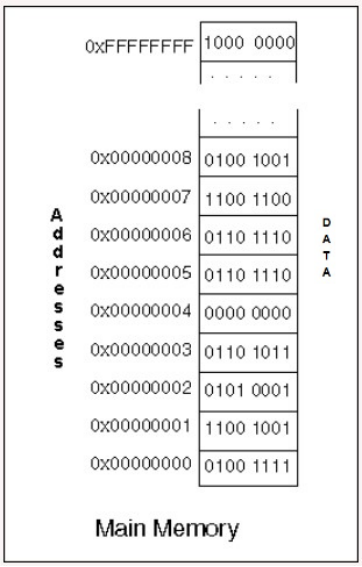


Figure 4.1: Memory Diagram.

Memory Notes

I. 3 Things in a Memory System

- ① Memory Address:
  e.g. 32 bits
- ② Memory Location:
  e.g. 8 bits
- ③ Memory Contents:
  e.g. $70

Memory can be represented as:

$$\text{Mem[addr]} \quad \textit{e.g.} \quad \text{Mem[0x08000180]}$$

Updating memory contents:

$$\text{Mem[addr]} \leftarrow \text{0xA2}$$

RTL (*Register Transfer Language*)

$$\text{R1} \leftarrow \text{Mem[addr]}$$

Location of Content **depends on the content.**

3. Data Units:

- Byte — *1 byte*
- Half Word (Hw) — *2 bytes*
- Word — *4 bytes*
- Long — *8 bytes*

Word: `Mem[0x080001AC]` ← (the lowest address of the data unit)

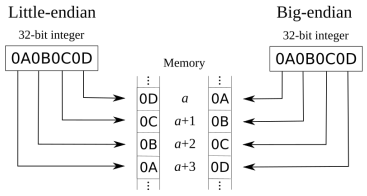


Figure 4.2: Big vs. Little Endian.

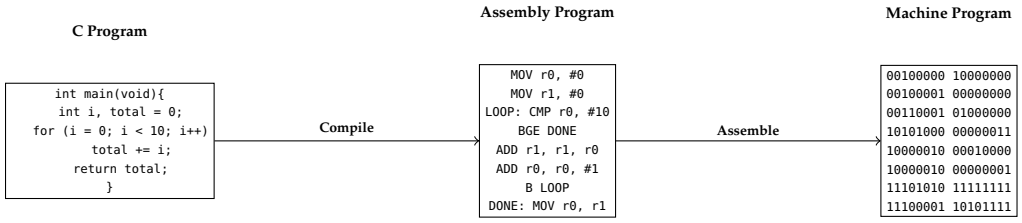
IV. Two Ways to Store Data in Memory

- ① **Big Endian:** The least significant byte (LSB) is stored at the **highest** address.
- ② **Little Endian (Small Endian):** The least significant byte (LSB) is stored at the **lowest** address.

Example Representation:

$$\text{Mem[0x080001AC]} = \begin{cases} \text{Big Endian: } \texttt{0xA00118BC} \\ \text{Little Endian: } \texttt{0xBC1801A0} \end{cases}$$

Levels of Program Code



High-Level Language:
* Level of abstraction closer to problem domain
* Provides for productivity and portability

Assembly Language:
* Textual representation of instructions
* **ARM Assembler**
* **GNU Assembler**

Hardware Representation:
* Binary digits (bits)
* Encoded instructions and data

Opcode Details

(E) `10111...000`
NOP: No Operations

NOP (*No Operation*) is inserted by the assembler as a placeholder to fill unused instruction slots.

`r1` ← `0` **Opcode:** `0010` (specific operation, context-dependent)

The opcode specifies the operation the CPU will perform. For example, `0010` might represent a move operation.

Registers: `r0` ~ `r7` (3 bits):

Registers store temporary data used in CPU instructions.

Register Details

`000`	Register 0
`001`	Register 1
`111`	Register 7

Represents register `r0` in the CPU's register file.

Represents register `r1`, used to hold intermediate values.

Represents register `r7`, often used for special purposes or flags.

Types of Instructions

1. (1) **Continuous:** PC ← PC + 4
2. (2) **Branch:** PC ← (PC) + 4 + 2 × offset

Refer to Appendix C & G.

Continuous instructions are sequentially executed, where the program counter (PC) increments by 4 for each instruction.

Opcode: `11100`

`B #imm11 « 1`

`0∗01 « 1 = 0...010`

Branch instructions allow the program to jump to a different memory location. The offset value determines the target address relative to the current PC.

This opcode represents a branch instruction (`B`) where the target address is calculated using an 11-bit immediate value shifted left by 1.

What is the offset?

The '#imm11' is an 11-bit immediate value that is shifted left by 1 to calculate the branch offset.

The immediate value '0*01' is shifted left by 1, resulting in '0...010'. This determines the branch offset.

Destination: ? **Check:** `0xA = 10`

The destination PC is calculated by adding '4 + 2 * offset' to the current PC.

`10 = (PC) + 4 + 2 ∗ offset`
10 is the new PC.

Conditional Branch Details

RLT Loop: (This refers to a conditional branch loop where the condition checks if the result is less than zero.)

Appendix G tells us:

- **Opcode:** `1101`
- **Condition (cond):** `1011` (**LT**, Less Than)
- **Immediate (imm):** `imm`
  (offset is part of this immediate value.)

This opcode represents a conditional branch instruction.
The condition code checks if the result is less than zero before branching.

Branch Destination: (PC) + 4 + 2 × offset

Loop Calculation:

$$0\text{xC} + 4 + 2 \times \text{offset}$$

Therefore:

$$\text{offset} = 12 + 4 + 2 \times \text{offset} \quad \Rightarrow \quad \text{offset} = -5$$

The offset of −5 makes the branch loop back to the desired memory location to continue execution.

Two's Complement Details

Example: `imm = -5 = -00000101`
`= 11111011 (Two's Complement)`

Two Algorithms to Find Two's Complement:

1. **1's Complement + 1:**
   `00110100` (Original)
   `11111010 + 1`
   `= 11111011`
2. **Direct Calculation:**
   `11001100` (Two's Complement Result)

The two's complement representation flips all bits (1's complement) and adds 1 to obtain the final binary value.

Self B Self (Assembly Code)

Appendix G tells us: **Opcode:** `11100`, **Immediate (imm11):** `imm11`

The new program counter (PC) is calculated as:

$$\text{New(PC)} = \text{PC} + 4 + 2 \times \text{offset}, \quad \text{New(PC)} = 0x10$$

Substituting:

$$0x10 = \text{PC} + 4 + 2 \times \text{offset} \quad \Rightarrow \quad \text{offset} = -2$$

Therefore: `imm11 = -00000000 010 = 111111111 10` (Two's complement)

Memory, I/O, and Instruction Excecution

5

Quiz 1 Review - Number Systems and Digital Logic

- Memory consists of capacitors that leak, requiring periodic refreshing.
- EEPROM is a type of memory.
- Volatile memory loses information when power is off.
- Non-volatile memory retains information even without power.
- Harvard architecture stores instructions and data in separate memory spaces.
- Von Neumann architecture stores instructions and data in the same memory space.
- A memory address with six hexadecimal digits corresponds to 24 bits.
- The memory range extends from 001000 to 0017FF.
- A half adder performs basic addition of two binary numbers.

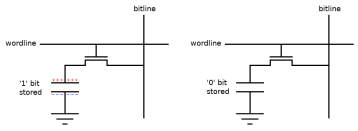


Figure 5.1: DRAM using capacitors

Memory Systems and Architectures

- A memory system includes an address decoder and additional logic.
- Memory space consists of a continuous set of memory locations.
- Memory devices can be either volatile or non-volatile.
- Memory architectures include Harvard and Von Neumann.
- A memory map is a diagram that shows used and unused memory addresses.

Components of Memory Space

- Each memory space consists of:
  - An address.
  - A location.
  - Content.

Memory-Mapped and Isolated I/O

- Memory-mapped I/O shares the same address space for memory and I/O.
- Isolated I/O uses separate address spaces for memory and I/O.

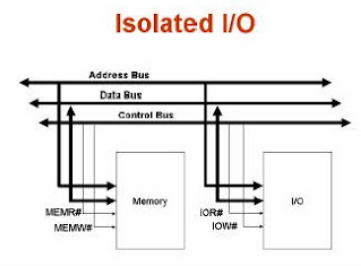


Figure 5.2: Memory-Mapped and Isolated I/O

Machine Code and Assembly

- Example instruction: `ADD r1, r1, r0`.
- Machine code representation:
  - (1) `DN = 1` for high register, `DN = 0` for low register.
  - (2) Binary representation: `00011 00 000 001 001`.

Figure 5.3: Machine Code from Assembly

Target vs. Host Machines

- Target machine: ARM Cortex-M.
- Host machine: Development environment.
- Tools used:
  - Editor.
  - Assembler.
  - Loader.
  - Linker.
  - Debugger.
  - Simulator.

ARM and GNU Assemblers

- ARM assembler is covered in textbooks and lectures.
- GNU assembler is used in labs.
- Comment syntax:
  - `;` for ARM assembler.
  - `@` for GNU assembler.
- Compiler assigns:
  - Registers `r1, r2, r3` for variables `a, b, c`.
  - Register `r0` for return value.

Number Formats and Registers

- Immediate data notation:
  - `#0x00` for ARM hexadecimal.
  - `#&00` for GNU.
- Link Register (`LR`) corresponds to `r14`.
- Memory access:
  - Regardless of endianness, addresses always point to the lowest byte.
  - `M(A)` refers to either location or content, depending on context.
  - `M(A)` $\leftarrow$ `(r1)` stores a value.
  - `r2` $\leftarrow$ `M(A)` loads a value.

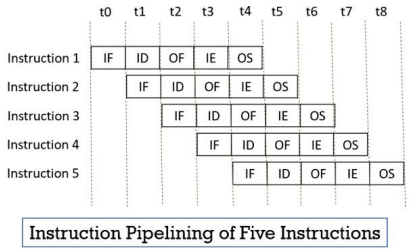


Figure 5.4: Instruction Pipelining

Performance Metrics

- MIPS (Million Instructions Per Second).
- Instruction latency: 3 clock cycles per instruction.
- Throughput:
  - Number of instructions per clock cycle: $\frac{3}{5} = 0.6$.
  - For longer sequences, throughput approaches 1.
  - Exact formula: $\frac{n \text{ instructions}}{n+\text{stages}-1}$.
  - Example: $\frac{4}{6}$ for a 32-bit instruction pipeline.

Example: Sum of an Array

- Variables used:
  - `A[10]`: Array with 10 elements.
  - `total`: Sum of elements.
  - `i`: Loop index stored in a register.
- Harvard architecture is used to store instructions and data in separate memory spaces.
- Two memory spaces are allocated: one for instructions and one for data.
- Example instruction:
  - `MOVS r1, #0x00`
  - Opcode: `00100 r1 = 001, imm = 0`
  - Memory representation:
    * `M[0x8000000] = 0x00`
    * `M[0x8000001] = 0x21`

Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00			
90000001	00	sum	int (4 bytes)	000000FF(255_{10})
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1_{10})
90000005	FF			
90000006	1F			
90000007	FF			
90000008	FF			
90000009	FF	averge	double (8 bytes)	1FFFFFFFFFFFFFF ($4.45015E-308_{10}$)
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90			
9000000F	00	ptrSum	int* (4 bytes)	90000000
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

Figure 5.5: Memory Diagram

Instruction Behavior and Memory Layout

6

Memory Representation

Little Endian Storage

In a little-endian system, the least significant byte (LSB) of a multi-byte value is stored at the lowest memory address. This affects how arrays and constants are stored in memory.

Given an array `a[]` starting at address `0x20000000`, memory is allocated as follows:

Offset	Value	Contents
0	1	`a[0]` (0x0001)
2	0	(blank)
4	2	`a[1]` (0x0002)
6	0	(blank)
8	3	`a[2]` (0x0003)

Each variable is stored in memory at 4-byte aligned addresses.

PC-Relative Loading Instructions

Pseudo and Machine Instructions

The instruction:

```
LDR r2, =total_Add
```

is a **pseudo-instruction**, meaning it is interpreted by the assembler rather than directly executed by the hardware.

The actual machine instruction generated by the assembler is:

```
LDR r2, [PC, #offset]
```

This instruction loads a value from memory, using the program counter (`PC`) as a base. The effective address is computed as:

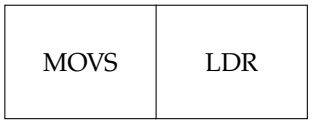
$$r2 \leftarrow m[PC + 4 + 4 \times \text{offset}]$$

Explanation:

- The `PC` value is adjusted by 4 due to pipelining.
- The offset is multiplied by 4 to maintain word alignment.

Instruction Encoding Structure

32 bits



Directives for Constant Definitions

underlineDCW

- **ARM:** `DCW` defines a 2-byte constant.
- **GNU:** `.short` defines the same type of 2-byte value.
- **DCD:** Used for defining a 4-byte constant (word).

Types of Instructions

Instructions can be classified into three categories:

- **Copy Instruction:** Translates directly into the same machine instruction.
- **Pseudo Instruction:** Depends on the assembler, which translates it into a machine instruction.
- **Directive:** Assembler command (e.g., `DCW, DCD`) that does not translate into machine instructions.

Memory Addressing

The first address in this memory system starts at:

$$\texttt{0x20000000}$$

Example access:

$$m[\texttt{0x20000000} + 0]$$

Instruction example:

$$\texttt{r1} \leftarrow \texttt{r1} + \texttt{r0}$$

Another load instruction using a shift operation:

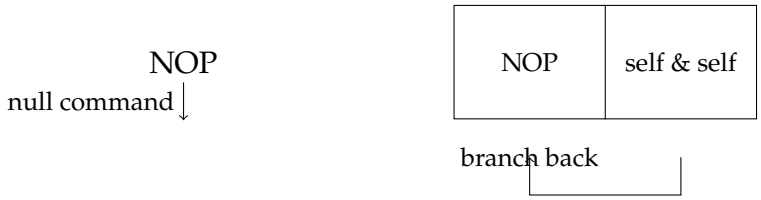
$$\texttt{LDR [\_ \_ \_ LSR \#2]}$$

The variable `total` is stored at:

```
0x20000024
```

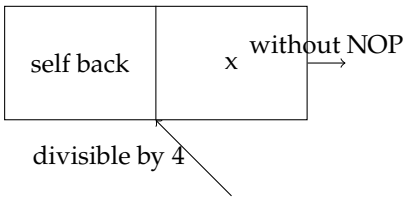
Branching and Execution Flow

NOP and Branching Behavior



Branching Without NOP

If `NOP` is removed, execution alignment may be affected.



Assembly Instruction Fields

In an assembly language instruction, there are four key fields:

- **Label (Symbol)** – This represents an identifier, such as a memory location or loop, e.g., `bop`.
- **Mnemonic** – The operation to be performed, such as `LDR`.
- **Operands** – The values or registers involved in the instruction, e.g., `r1 = addr`.
- **Comments** – Notes for human readability, e.g., `j`.

Symbol Table and First Pass Processing

The assembler constructs a **symbol table** during its first pass through the code. The table stores:

- **Symbols** – Identifiers (labels) found in the program.
- **Values** – Assigned memory addresses corresponding to those symbols.

The symbol table is built following these steps:

1. Initially, the assembler sets the **Location Counter (LC)** to the beginning address of the program:

$$LC = 0x00000000$$

2. The assembler scans each line of the program:
   - If a **symbol** (label) is found, it is assigned the current value of LC.
3. The LC is then incremented by the number of bytes occupied by the instruction (determined from Appendix G).
4. The process continues until an `END` directive (for ARM assemblers) or the equivalent termination condition in GNU assemblers.

Symbol Table

The symbol table for the given program might look as follows:

Symbol	Value
loop	`0x0800000`
check	`0x080000C`
self	`0x0800023`
total_addr	`0x0800024`
∅_addr	`0x0800026`

Machine Code Generation

To generate machine code, the assembler references:

- Appendix C, G, and H for instruction encoding.
- Built-in assembler logic to compute immediate values.

Immediate Value (`imm8`) Calculation in LDR

When processing an instruction like:

```
LDR r2, =total_addr
```

The assembler must determine the appropriate `imm8` (immediate value) in the instruction encoding.

Steps to calculate `imm8`:

1. The assembler checks the address of `total_addr` in the symbol table (e.g., `0x0800024`).
2. The base register is typically the Program Counter (PC), which points to the current instruction + 4 (due to pipelining).

3. The offset is computed as:

$$\texttt{imm8} = \frac{(\text{target address} - (\text{PC} + 4))}{4}$$

4. This value is encoded as an 8-bit immediate in the instruction.

This process allows the assembler to generate a valid instruction encoding for the `LDR` operation.

Memory (continued) 7

Definition Context

Appendix G

0010 001 0000 0101

Fields: Opcode, Rdn, imm8

Interpretation

This 16-bit instruction consists of:

- **Opcode:** The first few bits indicate the operation being performed.
- **Rdn:** The register destination or source, depending on the instruction type.
- **imm8:** An 8-bit immediate value, typically used for operations like loading constants or offsets.

Appendix G of Last Instruction

11100 1111 1111 110

Fields: Opcode, imm11

Interpretation

This instruction is a 16-bit immediate branch:

- **Opcode:** Identifies it as a branch instruction.
- **imm11:** An 11-bit signed immediate value, used to determine the branch offset.

The instruction updates the program counter (`PC`) based on the immediate value:

$$\texttt{PC} \leftarrow \texttt{PC} + 8 + 2 \times \texttt{imm11}$$

Explanation:

- The `PC` is incremented by 8 due to pipeline effects (fetching two instructions ahead).
- The offset is computed as $2 \times$ `imm11` because ARM Thumb instructions are halfword-aligned (16-bit increments).
- The final value of `PC` determines the next instruction location.

Memory Usage Diagram

The following diagram illustrates the memory layout, indicating which address ranges are used for volatile (RAM) and non-volatile storage (EEPROM, Flash).

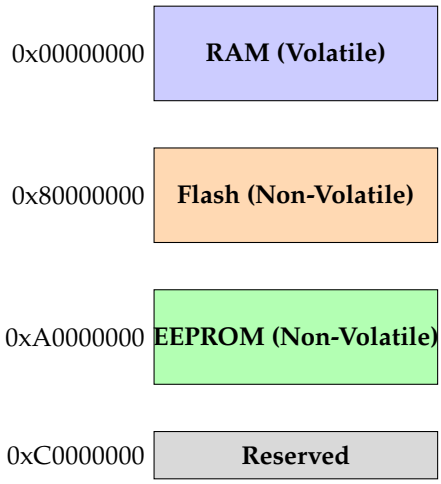
Address Range	Usage
`0x00000000 - 0x1FFFFFFF`	RAM (Volatile Memory)
`0x20000000 - 0x3FFFFFFF`	Reserved / Peripheral Registers
`0x40000000 - 0x5FFFFFFF`	I/O Mapped Registers
`0x60000000 - 0x7FFFFFFF`	External Memory (if available)
`0x80000000 - 0x9FFFFFFF`	Flash Memory (Non-Volatile)
`0xA0000000 - 0xBFFFFFFF`	EEPROM (Non-Volatile)
`0xC0000000 - 0xFFFFFFFF`	Reserved

Memory Classification

- **Data:** Stored in RAM (volatile memory), meaning it is lost when power is removed. - **Instructions:** Stored in non-volatile memory, such as Flash or EEPROM, ensuring that program code remains intact after power cycling.

Visual Representation

To provide a better understanding, the following block diagram illustrates how memory is structured:



Summary

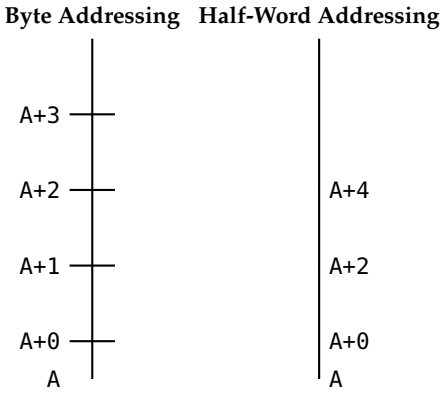
- Data is dynamically stored in RAM, which allows fast read/write access but does not persist after shutdown. - Instructions, including firmware and boot code, reside in non-volatile memory such as Flash or EEPROM. - Reserved memory regions may be used for hardware registers or specific peripheral mappings.

8 Data Representation

Bit, Byte, Half-Word, Word, Double-Word

In computing, memory is structured hierarchically, with data stored in units of increasing size. The smallest addressable unit in most architectures is the byte (8 bits), while larger data structures such as half-words (16 bits), words (32 bits), and double-words (64 bits) are aligned based on memory addressing rules.

The diagram below illustrates how memory addresses are incremented based on alignment constraints. On the left ladder, each rung represents a sequential byte address, whereas on the right ladder, addresses increase in steps of two (for half-word alignment).



Byte Addressing (Left Ladder) Each memory address corresponds to a single byte, meaning data stored in this scheme is accessed sequentially, increasing by 1 for each subsequent byte. **Half-Word Addressing (Right Ladder)** Half-words (16-bit units) require alignment to even addresses. The memory addresses increase in steps of two, ensuring that each half-word starts at a properly aligned boundary.

Unsigned n-bit Integers

In an unsigned n-bit integer representation, numbers are represented in the range:

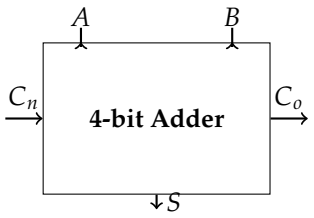
$$A_{n-1}, A_{n-2}, \ldots, A_1, A_0$$

where each A_i represents a binary digit (0 or 1), and the highest value is given by:

$$2^n - 1$$

4-bit Full Adder

A full adder (FA) takes in two binary numbers and computes their sum while propagating carry information. The diagram below represents a 4-bit adder with carry-in (C_n), sum output (S), and carry-out (C_o).



Carry/Borrow Flag Bit for Unsigned Numbers

- When adding two unsigned numbers in an n-bit system, a **carry** occurs if the result exceeds the maximum unsigned integer that can be represented, i.e., $2^n - 1$.
- When subtracting two unsigned numbers, a **borrow** occurs if the result is negative, meaning it is smaller than the smallest unsigned integer that can be represented, i.e., 0.
- On **ARM Cortex-M4** processors, the **carry flag** and the **borrow flag** are physically the same flag bit in the status register (**PSR**, Program Status Register).

Signed n-bit Integers

	Sign-and-Magnitude	**One's Complement**	**Two's Complement**
Range	$[-(2^{n-1}-1), 2^{n-1}-1]$	$[-(2^{n-1}-1), 2^{n-1}-1]$	$[-2^{n-1}, 2^{n-1}-1]$
Zero	Two zeroes (± 0)	Two zeroes (± 0)	One zero
# of Numbers	$2^n - 1$	$2^n - 1$	2^n

All modern computer systems use **Two's Complement (TC)** representation.

Hardware for $A + B$ and $A - B$

Two's complement arithmetic allows for a unified hardware implementation:

- Addition ($A + B$) follows standard binary addition rules.
- Subtraction ($A - B$) is performed using two's complement: $A - B = A +$ (Two's Complement of B).
- The same ALU (Arithmetic Logic Unit) can handle both operations efficiently.

Full Adders (FAs)

A full adder computes the sum and carry-out:

$$\text{Carry} = C_n \text{ and Overflow} = C_n$$

For unsigned addition:

$$A + B \geq 2^n$$

Two's Complement Representation

Two's complement allows subtraction by adding the complement of a number:

$$A - B = A + (\overline{B} + 1)$$

Adder/Subtractor Table

The table below summarizes the behavior of the adder/subtractor unit.

Add/Sub	**P(U)**	x_s
0	0	0
0	1	1
1	0	1
1	1	0

Example: $n = 4$

For a 4-bit system, we examine subtraction using two's complement.

Example 1: $2 - 1$

$$\begin{array}{ll} \mathbf{0010} & (2) \\ \mathbf{-0001} & (1) \end{array}$$

Convert -1 **to Two's Complement:**

$$\begin{array}{ll} \mathbf{0001} & \text{(Original)} \\ \mathbf{1111} & \text{(Two's complement of 0001)} \end{array}$$

Perform Addition:

$$\begin{array}{ll} \mathbf{0010} & (2) \\ +\,\mathbf{1111} & \text{(Two's complement of 1)} \\ \hline \mathbf{0001} & \end{array}$$

Since there is no carry-out from the most significant bit, no overflow occurs.

Example 2: $2 - 3$

$$\begin{array}{ll} \mathbf{0010} & (2) \\ \mathbf{-0011} & (3) \end{array}$$

Convert -3 to Two's Complement:

0011	(Original)
1101	(Two's complement of 0011)

Perform Addition:

$$\begin{array}{ll} \mathbf{0010} & (2) \\ +\,\mathbf{1101} & (\text{Two's complement of } 3) \\ \hline \mathbf{1111} & \end{array}$$

Overflow Analysis

$C_o = 0$, but carry into the MSB $= 1 \Rightarrow$ **Overflow Occurs**

Why Does Overflow Occur?

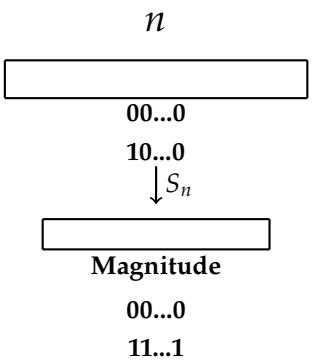
Overflow happens when the result of an addition exceeds the representable range of the n-bit system. In two's complement:

- **Overflow occurs if the sign bit changes unexpectedly**.
- If both operands have the same sign and the result has a different sign, overflow is present.

For $2 - 3$, the expected result is -1, which should be 1111 in two's complement. However, since there was an incorrect carry propagation, **overflow occurs**.

Summary of Overflow Rules

- **Addition:** Overflow occurs when adding two positive numbers results in a negative number, or when adding two negative numbers results in a positive number.
- **Subtraction:** Overflow occurs if the sign bit flips unexpectedly due to an incorrect carry propagation.



Signed Integer Computation

In a 4-bit signed system, the most significant bit (MSB) determines the sign:

- $0 \rightarrow$ Positive number
- $1 \rightarrow$ Negative number (using two's complement)

Example 1: $2 - 3$

$$\begin{array}{rl} 1101 & \text{(Negative)} \\ 0010 & (2) \\ -0011 & (3) \end{array}$$

Convert -3 to Two's Complement:

$$\begin{array}{ll} 0011 & \text{(Original)} \\ 1101 & \text{(Two's complement of 3)} \end{array}$$

Perform Addition:

$$\begin{array}{rl} 0010 & (2) \\ +1101 & \text{(Two's complement of 3)} \\ \hline 1111 & (-1) \end{array}$$

Since the MSB remains 1, the result is correctly interpreted as -1, so no overflow occurs.

Multiplication in Signed Systems

For an m-bit system, multiplying two n-bit signed numbers produces a $2n$**-bit signed product**:

$$\text{n-bits} \times \text{n-bits} \rightarrow \text{(2n)-bit signed product}$$

Example 2: Signed Multiplication -1×2

1. Represent the numbers:
   - -1 in 4-bit: 1111
   - 2 in 4-bit: 0010
2. **Multiply**:
   $$(-1) \times 2 = -2$$
   The product should be '-2', which in an **8-bit signed system is:**

1111 1110

This demonstrates how **sign extension** is crucial in interpreting the multiplication result.

—

Overflow Flag for Signed Numbers

Overflow in a signed system occurs **when the result does not fit within the representable range**.

Detecting Overflow in Addition/Subtraction

Consider two n-bit signed numbers:

$$A = A_{n-1}A_{n-2}\ldots A_1A_0 \text{ and } B = B_{n-1}B_{n-2}\ldots B_1B_0$$

When we subtract:

$$A - B = S_{n-1}S_{n-2}\ldots S_1S_0$$

Overflow occurs when:

$$\text{Overflow Flag} = C_{\text{in}} \oplus C_{\text{out}}$$

Where: - C_{in} is the carry **into** the most significant bit (MSB). - C_{out} is the carry **out** from the MSB.

Overflow Conditions

- **Addition Overflow:** Occurs when adding two positives results in a negative, or when adding two negatives results in a positive.
- **Subtraction Overflow:** Occurs when the sign bit flips unexpectedly due to an incorrect carry propagation.

Final Overflow Rule:

$$\text{Overflow (Signed System)} = C_{\text{in}} \oplus C_n$$

Where: - C_{in} is the carry into the MSB. - C_n is the carry out from the MSB.

This equation is used in hardware to efficiently determine whether an arithmetic operation has resulted in overflow.

Program Status Register 9

Pseudo Instructions (for Assembler)

Pseudo-instructions are assembler-level commands that are not actual machine instructions but are translated by the assembler into valid machine instructions. They simplify programming by providing shorthand representations.

Directives (for Assembler)

Assembler directives are commands that instruct the assembler how to process the assembly code but do not generate machine code themselves. They help in organizing and managing memory, defining constants, and controlling the assembly process.

Example: Subtracting $(-9) - 8$ in a 5-bit System

Step 1: Determine the Binary Representation

In a 5-bit signed integer representation (Two's Complement format):

- The range is -16 to $+15$.
- -9 in 5-bit two's complement: 10111.
- -8 in 5-bit two's complement: 11000.

Step 2: Perform the Subtraction

Subtracting -8 from -9 is equivalent to adding -9 and the two's complement of -8.

1. Compute the two's complement of -8:

$$\text{Invert } 11000 \text{ to } 00111, \text{ then add } 1:$$

$$00111 + 1 = 01000 \Rightarrow \text{(Two's complement of -8)}$$

2. Add -9 (10111) and $+8$ (01000):

$$\begin{array}{r} 10111 \\ +01000 \\ \hline 11111 \end{array}$$

Step 3: Analyze Overflow and Condition Codes

Flag	Name	Condition
V	Overflow Flag	Set if the carry into the most significant bit (MSB) differs from the carry out. Here, carry-in = 0, carry-out = 1, so $V = 1$.
N	Negative Flag	Set if the result's MSB is 1, indicating a negative number. Since there is an overflow, the result is incorrect. $N = 0$.
Z	Zero Flag	Set if the result is zero. The result is 11111, which is not zero, so $Z = 0$.
C	Carry Flag	Set if there is a carry-out from the MSB. The carry-out from MSB is 1, so $C = 1$.

Thus, the final condition codes are:

$$N = 0, \quad Z = 0, \quad C = 1, \quad V = 1$$

Condition Codes in the Processor Status Register (PSR)

The **Program Status Register (PSR)** contains condition codes that store the results of arithmetic operations. These are:

Bit	Name	Meaning (After Addition or Subtraction)
31	**N** (Negative)	Set if the result is negative (MSB = 1).
30	**Z** (Zero)	Set if the result is zero.
29	**C** (Carry)	Set if there is an unsigned overflow (carry-out from MSB).
28	**V** (Overflow)	Set if there is signed overflow (carry-in $\oplus$ carry-out).

CPU Does Not Know Signed or Unsigned Numbers

The CPU itself does not inherently distinguish between signed and unsigned numbers; it simply performs binary arithmetic. It is up to the programmer to interpret the result correctly.

- Signed numbers use two's complement representation.
- Unsigned numbers treat all bits as part of a positive magnitude.
- The condition codes help determine the correct interpretation.

Using Condition Codes for Decision Making

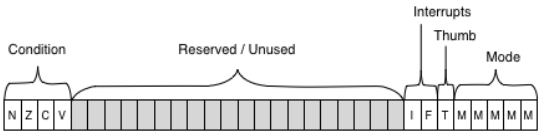
The processor provides conditional execution based on condition flags, allowing efficient branching and comparisons. In many architectures (e.g., ARM), condition codes are used to determine control flow:

- `BEQ` (Branch if Equal) $\rightarrow$ Uses Zero Flag (Z).
- `BMI` (Branch if Minus) $\rightarrow$ Uses Negative Flag (N).

- `BCS` (Branch if Carry Set) → Uses Carry Flag (C).
- `BVS` (Branch if Overflow Set) → Uses Overflow Flag (V).

Illustration of PSR

The PSR layout is as follows:



The key fields in the PSR store the condition flags and are updated after each arithmetic operation.

Pointer Reference and Dereference

10

Find Out String Length

Strings in C are terminated with a null character (`NUL`, ASCII value `0x00`). You can determine the length of a string using either pointer dereferencing or the array subscript operator.

```c
// Using pointer dereference
int length(const char *str) {
    const char *ptr = str;
    while (*ptr != '\0') {
        ptr++;
    }
    return ptr - str;
}
```

```c
// Using array subscript
int length(const char str[]) {
    int i = 0;
    while (str[i] != '\0') {
        i++;
    }
    return i;
}
```

A Brief Review of Pointers

Pointer vs. Array

A `char *ptr` can be used to mimic an array of characters, but there are crucial differences in how memory is managed and how it behaves compared to a statically declared `char array[]`.

Accessing Elements

In both cases, elements can be accessed using the array indexing syntax:

Pointer and Array Access

$$\texttt{ptr[i]} \equiv \texttt{*(ptr + i)}$$

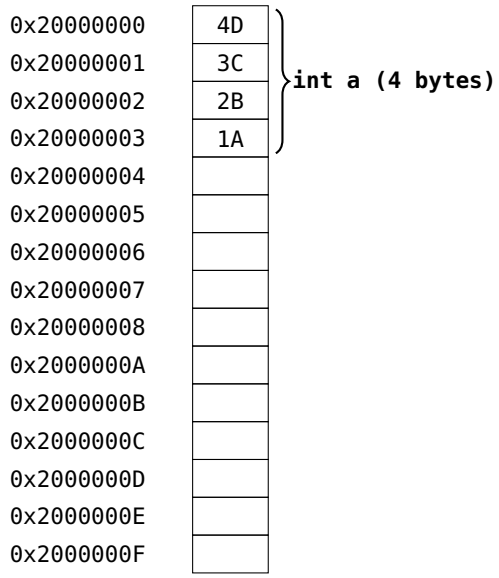
Reference and Dereference

- **`&a` (Reference)**: Taking the address of an existing variable `a` to assign it to a pointer variable.
- **`*p` (Dereference)**: Accessing the value stored at the memory location pointed to by `p`.

Integer Data Type

Example: Integer Memory Representation

`int a = 0x1A2B3C4D;`
Assume `a` is stored at memory location `0x20000000`.



Pointer Example: Address and Dereferencing

1. Declaring a Pointer

```
int *p = &a;
```

This creates a pointer variable `p` that stores the memory address of `a`. That means `p` does not hold an integer value, but rather the location in memory where `a` is stored.

2. Undefined Dereference

If `p` is not initialized properly (i.e., not assigned a valid address), then dereferencing it leads to undefined behavior:

```
(*p is not defined)
```

3. Dereferencing the Pointer

```
int c = *p;
```

Here, we retrieve the value stored at the address `p` is pointing to and assign it to `c`. If `p = &a`, then `*p == a`, so this is equivalent to:

$$c = a$$

4. Pointer Arithmetic and Memory Access

```
int d = *(p + 1);
```

Since `p` is an `int*`, `p + 1` moves **4 bytes ahead** (assuming `sizeof(int) = 4` bytes). Thus, if `p = &a` and `a` is stored at address `0x20000000`, then:

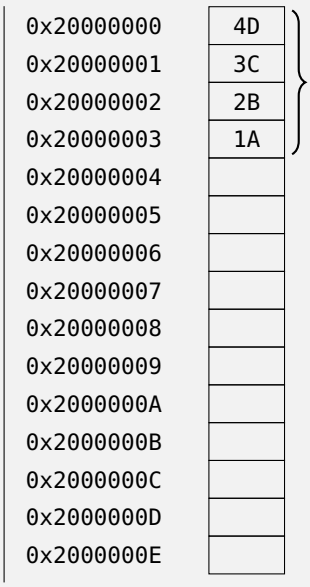
$$p + 1 \longrightarrow 0x20000004$$

This means `*(p + 1)` retrieves the integer stored at `0x20000004` and assigns it to `d`.

5. Visual Representation

Memory Layout (Little Endian)

```
int a = 0x1A2B3C4D;
int *p = &a;
int d = *(p + 1);
```



Key Takeaways:

- `*p` retrieves the value stored at the address `p` points to.
- `*(p + 1)` accesses the next integer in memory (4 bytes ahead for `int`).
- Accessing `*(p + 1)` is only valid if `p` is pointing to an array or properly allocated memory.
- If `p` is uninitialized, dereferencing it (`*p`) leads to undefined behavior.

Pointer Operations Breakdown

For Loop Structure with Pointers

A typical loop involving pointers follows these steps:

1. **Initialization**: Setting the pointer to the start of a string or memory block.
2. **Check the Condition**: Ensuring the pointer does not exceed a boundary or reach a terminating character.
3. **Action**: Performing operations on the data the pointer references.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53	54	55	56	57	58	59	5A

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	71	72	73	74	75	76	77	78	79	7A

```
'a' - 'A' = 0x61 - 0x41 = 0x20 = 32
```

Pointer dereference operator `*`

```
void toUpper(char *pStr){
  for(char *p = pStr; *p; ++p){
    if(*p >= 'a' && *p <= 'z')
      *p -= 'a' - 'A';
      //or: *p -= 32;
  }
}
```

Array subscript operator `[]`

```
void toUpper(char *pStr){
  char c = pStr[0];
  for(int i = 0; c; i++, c = pStr[i];) {
    if(c >= 'a' && c <= 'z')
      pStr[i] -= 'a' - 'A';
      // or: pStr[i] -= 32;
  }
}
```

Figure 10.1: Turning lowercase into uppercase using using these concepts

Key Concepts

- **Declaration:** `char *p;` defines a pointer to a character.
- **Initialization:** `p = pStr;` assigns it to a string.
- **Dereferencing:** `*p` accesses the value stored at the pointer's location.
- **Pointer Arithmetic:** `p++;` advances to the next memory location.

ARM Instruction Set Architecture 11

ISA (Instruction Set Architecture)

An Instruction Set Architecture (ISA) defines the supported instructions for a specific processor. In ARM ISA, instructions fall into different categories:

- **Instructions for the target machine**: These are the actual machine-level operations executed by the processor.
- **Pseudo-instructions**: These are assembler-level conveniences that simplify coding but are converted into real machine instructions.
- **Directives**: These are special commands for the assembler to assist in organizing and defining data.

Assembler Directives

Assembler directives are used to define and allocate memory or assist in code organization. Some common directives include:

- `DCW` – Define Constant Word (stores a 16-bit value)
- `DCD` – Define Constant Doubleword (stores a 32-bit value)
- `EQU` – Define a constant value
- `AREA` – Define a named section in memory

ARM-1: 32-bit data
26-bit address (2 trailing 0's assumed at the end)

$$\text{Memory space} = 2^{26} \times 4 = 2^{28} = 2^8 \times 2^{20} = 256 \times 1M = 256M$$

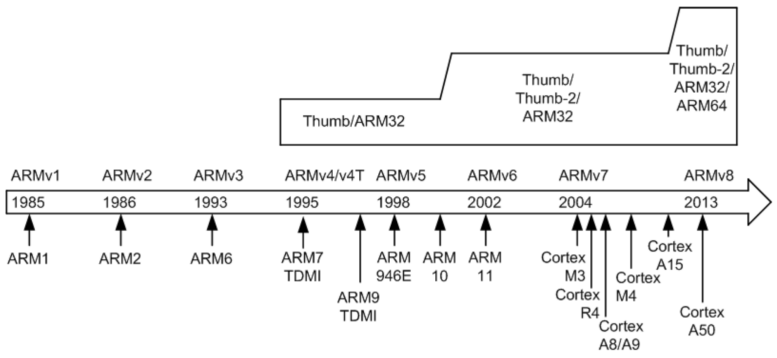


Figure 11.1: History of ARM Processor

Assembly Directives | 12

Assembly Directives and IEEE754 Single Precision Format

Directives are not instructions; they provide key information for the assembler. Below are details on the IEEE754 representation and common assembly directives.

IEEE754 Single Precision Format

The IEEE754 format encodes a floating-point number as follows:

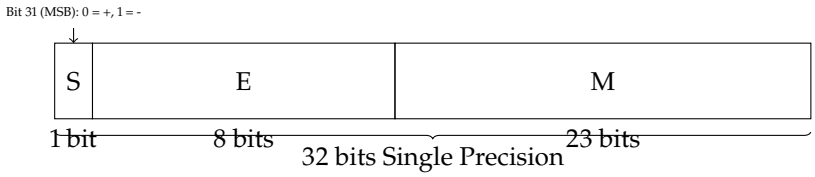
$$(-1)^S \times 1.\text{Mantissa} \times 2^{(\text{Exponent})-127}$$

For example, consider the binary representation:

$$\underbrace{0}_{\text{Sign}} \quad \underbrace{10000010}_{\text{Exponent}} \quad \underbrace{1101000\ldots0}_{\text{Mantissa}}$$

- **Sign:** A 0 means a positive number.
- **Exponent:** The field 10000010_2 equals 130_{10}. Subtracting the bias (127) gives an actual exponent of 3.
- **Mantissa:** With an implicit leading 1, the normalized mantissa is $1.1101000\ldots_2$. Multiplying by 2^3 shifts the binary point three places, yielding approximately

$$1.1101000\ldots_2 \times 2^3 \approx 1110.1000_2 \approx 14.5.$$



Assembly Directives

AREA Make a new block of data or code.
ENTRY Declare an entry point where the program execution starts.
ALIGN Align data or code to a particular memory boundary.
DCB Allocate one or more bytes (8 bits) of data.
DCW Allocate one or more half-words (16 bits) of data.
DCD Allocate one or more words (32 bits) of data.
SPACE Allocate a zeroed block of memory with a particular size.
FILL Allocate a block of memory and fill it with a given value.
EQU Give a symbol name to a numeric constant.
RN Give a symbol name to a register.
EXPORT Declare a symbol and make it referable by other source files.
IMPORT Provide a symbol defined outside the source file.

INCLUDE/GET Include a separate source file within the source file.
PROC Declare the start of a procedure.
ENDP Designate the end of a procedure.
END Designate the end of a source file.

Example Assembly Code:

```
AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5   ; Define an array with five integers

AREA myCode, CODE, READONLY ; Define a code section
EXPORT __main             ; Make __main visible to the linker
ENTRY                     ; Mark the entrance to the entire program
__main PROC             ; Start of the main procedure
  ...                   ; Assembly program starts here.
ENDP                    ; End of the main procedure
END                     ; End of the source file
```

Notes:

- The `AREA` directive signals the start of a new data or code section. Each area is a basic, independent unit processed by the linker and must have a unique name.
- An assembly program must have at least one code area. By default, code areas are `READONLY` and data areas are `READWRITE`.

Mantissa Conversion Example

Given the floating-point representation:

$$\underbrace{0}_{\text{Sign}} \quad \underbrace{10000010}_{\text{Exponent}} \quad \underbrace{1101000\ldots0}_{\text{Mantissa}}$$

- **Sign:** 0 indicates a positive value.
- **Exponent:** $10000010_2 = 130_{10}$, so the actual exponent is $130 - 127 = 3$.
- **Mantissa:** The normalized mantissa is $1.1101000\ldots_2$. Multiplying by 2^3 shifts the binary point:

$$1.1101000\ldots_2 \times 2^3 \approx 1110.1000_2 \approx 14.5.$$

Step-by-Step Fractional Conversion:

1. Multiply the fractional part (e.g., 0.473) by 2:

   $$0.473 \times 2 \approx 0.946.$$

   Since $0.946 < 1$, the first bit is $\underline{0}$.
2. Multiply the new fractional part (0.946) by 2:

   $$0.946 \times 2 \approx 1.892.$$

   The integer part is 1; record a bit $\underline{1}$ and subtract 1 to get 0.892.

3. Multiply 0.892 by 2:

$$0.892 \times 2 \approx 1.784.$$

   Record a bit $\underline{1}$ and subtract 1 to obtain 0.784.
4. Continue this iterative process until the desired precision is reached.

This method constructs the binary fraction used in the mantissa field of the IEEE754 format.

Directive: ENTRY

The `ENTRY` directive marks the very first instruction to be executed within an application program. It is essential because it defines the program's entry point. Notably, an application must have exactly one `ENTRY` directive, regardless of the number of source files it contains.

Example: ENTRY Directive in Assembly

```
AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5   ; Define an array with five integers

AREA myCode, CODE, READONLY ; Define a code section
EXPORT __main             ; Make __main visible to the linker
ENTRY                     ; Mark the entrance to the entire program
__main PROC             ; PROC marks the beginning of a subroutine
  ...                   ; Assembly program starts here.
ENDP                    ; Mark the end of a subroutine
END                     ; Mark the end of a source file
```

Key Points:

- The `ENTRY` directive indicates where program execution begins.
- It ensures there is one clear starting point for the application.

Directive: END

The `END` directive signals the end of a source file. Every assembly source file must conclude with this directive to inform the assembler that there is no more code or data to process.

Example: END Directive in Assembly

```
AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5   ; Define an array with five integers

AREA myCode, CODE, READONLY ; Define a code section
EXPORT __main             ; Make __main visible to the linker
ENTRY                     ; Mark the entrance to the entire program
__main PROC             ; PROC marks the beginning of a subroutine
```

```
  ...                   ; Assembly program starts here.
ENDP                    ; Mark the end of a subroutine
END                     ; Mark the end of the source file
```

Key Points:

- The `END` directive indicates the termination of the source file.
- Including this directive is mandatory in every assembly program.

Directive: PROC and ENDP

The `PROC` and `ENDP` directives mark the start and end of a subroutine (also called a procedure or function) in an assembly source file. They clearly delineate the boundaries of a subroutine and enable the assembler to correctly manage multiple functions within the same file.

Example: PROC and ENDP in Assembly

```
AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5   ; Define an array with five integers

AREA myCode, CODE, READONLY ; Define a code section
EXPORT __main               ; Make __main visible to the linker
ENTRY                       ; Mark the entrance to the entire program
__main PROC                 ; PROC marks the beginning of a subroutine
  ...                       ; Assembly program starts here.
ENDP                        ; Mark the end of the subroutine
END                         ; Mark the end of the source file
```

Key Points:

- `PROC` indicates the start of a subroutine.
- `ENDP` indicates the end of that subroutine.
- A single source file can contain multiple subroutines, each defined by its own `PROC` and `ENDP` pair.
- **Important:** `PROC` and `ENDP` cannot be nested; a function cannot be defined within another function.

Directive: EXPORT and IMPORT

The `EXPORT` directive declares a symbol and makes it visible to the linker, allowing other modules to reference it. Conversely, the `IMPORT` directive informs the assembler that a symbol, not defined in the current source file, is defined externally in another file. The functionality of `IMPORT` is similar to the `extern` keyword in C.

Example: EXPORT and IMPORT in Assembly

```
AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5   ; Define an array with five integers

AREA myCode, CODE, READONLY ; Define a code section
EXPORT __main             ; Make __main visible to the linker
ENTRY                     ; Mark the entrance to the entire program
__main PROC             ; PROC marks the beginning of a subroutine
  ...                   ; Assembly program starts here.
ENDP                    ; Mark the end of the subroutine
END                     ; Mark the end of the source file
```

Key Points:

- `EXPORT` declares a symbol to be accessible by the linker across multiple source files.
- `IMPORT` is used to reference symbols defined in other assembly files.
- The `IMPORT` directive works similarly to the `extern` keyword in C.

Directive: Data Allocation

Data allocation directives reserve memory space for constants and variables in an assembly program. They specify both the type of data and the size of the memory block to be allocated.

Directives and Their Descriptions:

`DCB`	Define Constant Byte: Reserve 8-bit values.
`DCW`	Define Constant Half-word: Reserve 16-bit values.
`DCD`	Define Constant Word: Reserve 32-bit values.
`DCQ`	Define Constant Quad-word: Reserve 64-bit values.
`DCFS`	Define single-precision floating-point numbers: Reserve 32-bit values.
`DCFD`	Define double-precision floating-point numbers: Reserve 64-bit values.
`SPACE`	Defined Zeroed Bytes: Reserve a number of zeroed bytes.
`FILL`	Defined Initialized Bytes: Reserve memory and fill each byte with a specified value.

Key Points:

- Each directive is used to allocate a specific block of memory with a defined size.
- These directives help in organizing data storage in the program by reserving the exact amount of memory required for each data type.

Directive: Data Allocation – Floating Point Numbers

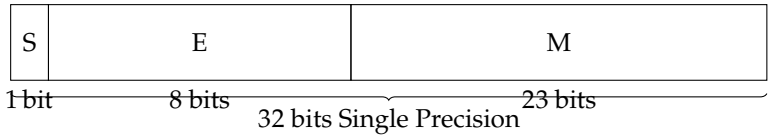
Floating point numbers are allocated using directives that reserve space for numbers in the IEEE 754 format. Two primary formats exist:

- **Single Precision:** Represented as a single long rectangle.
- **Double Precision:** Represented as two long rectangles side by side (conceptually dividing the 64-bit value into two 32-bit sections).

IEEE 754 Single Precision Format

The single precision format is divided into three fields:

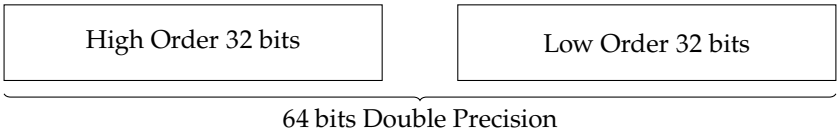
- **Sign (S):** 1 bit
- **Exponent (E):** 8 bits
- **Mantissa (M):** 23 bits (with an implicit leading 1)



IEEE 754 Double Precision Format

Double precision numbers are 64 bits in size. One common way to depict them is as two adjacent 32-bit rectangles, illustrating the higher-order and lower-order words:

- **High Order 32 bits:** Contains the sign, exponent, and part of the mantissa.
- **Low Order 32 bits:** Contains the remaining bits of the mantissa.



Single Precision Examples

Example 1:

Binary Representation:
0 1000 0010 10110...0

Calculation:

Value $= (-1)^0 \times 1.1011\ldots0 \times 2^{(130-127)}$

$= 1.1011\ldots_2 \times 2^3 \approx 1101.1_2$

≈ 13.5 (decimal)

Example 2:

Binary Representation:
1 1000 0001 01100...0

Calculation:

Value $= (-1)^1 \times 1.01100\ldots0 \times 2^{(129-127)}$

$= -\left(1.01100\ldots_2 \times 2^2\right)$

Supplemental Information

- The **bias** for single precision is 127; hence the actual exponent is the encoded exponent minus 127.
- An implicit leading 1 is assumed in the normalized mantissa.
- In Example 1, the exponent field $1000\,0010_2$ equals 130 (decimal), so the multiplier is $2^{130-127} = 2^3$.
- In Example 2, the exponent field $1000\,0001_2$ equals 129 (decimal), so the multiplier is $2^{129-127} = 2^2$, and the sign bit of 1 indicates a negative value.

Directive: Data Allocation – Floating Point Numbers (Example 3)

Example 3: Find the IEEE 754 Single Precision Floating Point Representation of 3.14

Step 1: Convert to Binary

- The integer part 3 converts to 11_2.
- The fractional part 0.14 converts approximately to $0.0010001111010111\ldots_2$.
- Thus, $3.14 \approx 11.0010001111010111\ldots_2$.

Step 2: Normalize the Binary Number

- Express the binary number in normalized form:

$$11.0010001111010111\ldots_2 = 1.10010001111010111\ldots_2 \times 2^1$$

Step 3: Determine the IEEE 754 Fields

- **Sign (S):** 0 (since 3.14 is positive).
- **Exponent (E):** With a bias of 127, the stored exponent is $1+127 = 128$. In binary, $128_{10} = 10000000_2$.
- **Mantissa (M):** The normalized fraction is $10010001111010111\ldots_2$. This is rounded/truncated to 23 bits:

$$10010001111010111000011 \quad \text{(approximate)}$$

Step 4: Assemble the Final Representation

The final IEEE 754 single precision representation is:

$$\underbrace{0}_{\text{Sign}} \quad \underbrace{10000000}_{\text{Exponent}} \quad \underbrace{10010001111010111000011}_{\text{Mantissa}}$$

This corresponds to the hexadecimal value `0x4048F5C3`.

Summary

- **Sign:** 0 (positive).
- **Exponent:** 10000000 (binary for 128, i.e., 1 + 127).
- **Mantissa:** 10010001111010111000011 (23 bits after rounding).

Directive: Data Allocation

The following assembly code example demonstrates various data allocation directives used to reserve memory space for strings, integers, and floating-point numbers. Each directive specifies the type and size of data to be stored.

```
AREA myData, DATA, READWRITE
hello   DCB "Hello World!",0   ; Allocate a null-terminated string
dollar  DCB 2,10,0,200          ; Allocate integers ranging from -128 to 255
scores  DCD 2,3,8,4             ; Allocate 4 words containing decimal values
miles   DCW 100,200,50,0        ; Allocate integers between -32768 and 65535
Pi      DCFS 3.14              ; Allocate a single-precision floating-point number
Pi      DCFD 3.14              ; Allocate a double-precision floating-point number
p       SPACE 255              ; Allocate 255 bytes of zeroed memory space
f       FILL 20,0xFF,1         ; Allocate 20 bytes and fill each byte with 0xFF
binary  DCB 2_01010101         ; Allocate a byte using a binary value
octal   DCB 8_73               ; Allocate a byte using an octal value
char    DCB 'A'                ; Allocate a byte initialized to the ASCII value of 'A'
```

Key Points:

- `DCB` reserves a constant byte (8 bits) and can be used for strings, small integers, or to specify binary/octal/character values.
- `DCW` reserves a constant half-word (16 bits), suitable for larger integer values.
- `DCD` reserves a constant word (32 bits), often used for storing decimal values.
- `DCFS` reserves space for a single-precision floating-point number (32 bits).
- `DCFD` reserves space for a double-precision floating-point number (64 bits).
- `SPACE` allocates a block of zeroed bytes.
- `FILL` allocates memory and initializes each byte with a specified value.

Directive: EQU and RN

The `EQU` directive associates a symbolic name to a numeric constant—much like the `#define` in C—allowing you to use meaningful names in your assembly code instead of hard-coded numbers. In contrast, the `RN` directive assigns a symbolic name to a specific register, making the code more readable and maintainable.

Example Code

```
; Interrupt Number Definition (IRQn)
BusFault_IRQn EQU -11   ; Cortex-M3 Bus Fault Interrupt
SVCall_IRQn   EQU -5    ; Cortex-M3 SV Call Interrupt
PendSV_IRQn   EQU -2    ; Cortex-M3 Pend SV Interrupt
SysTick_IRQn  EQU -1    ; Cortex-M3 System Tick Interrupt

Dividend      RN 6     ; Defines 'Dividend' as register 6
Divisor       RN 5     ; Defines 'Divisor' as register 5
```

Notes:

- The `EQU` directive is used to define constant values, which improves code clarity and maintainability.
- The `RN` directive is used to give a register a symbolic name. (In some assemblers, this is defined using `[EQU/ARM]` or `.set/GNU` syntax.)

Additionally, a `FILL` directive (e.g., `FILL 20, 0xFF, 2`) may be used to initialize blocks of memory with specific patterns. In our context, it can illustrate how registers or memory segments are preloaded with values such as 0xFF, 0x00, 0xFF, 0x00, etc.

Summary:

- `EQU` assigns constant numeric values to symbolic names.
- `RN` assigns symbolic names to registers, improving code clarity.
- The register file diagram visualizes registers R0–R21, each being 32 bytes in size and potentially used as 16 double precision registers.

Directive: ALIGN

The `ALIGN` directive forces the location counter (LC) to be adjusted to a specified alignment boundary. This is crucial for ensuring that code or data starts at addresses that meet hardware or performance requirements.

Example: ALIGN in Assembly

```
AREA example, CODE, ALIGN = 3 ; Memory address begins at a multiple of 8 (2^3)
ADD r0, r1, r2             ; Instructions start at a multiple of 8

AREA myData, DATA, ALIGN = 2 ; Data address starts at a multiple of 4 (2^2)
a    DCB 0xFF              ; The first byte of a 4-byte word
ALIGN 4, 3                ; Align to a boundary of 4 bytes with an offset of 3
b    DCB 0x33              ; Set the fourth byte of a 4-byte word
c    DCB 0x44              ; Add a byte making the next data misaligned
ALIGN                     ; Force the next data to be aligned
d    DCD 12345            ; Skip three bytes and store the word
```

Explanation

- In the `example` code section, `ALIGN = 3` causes the LC to start at a multiple of $2^3 = 8$. This means that even if the current LC is not a multiple of 8, it is adjusted upward to the nearest such address.
- In the `myData` section, `ALIGN = 2` ensures the data begins at an address that is a multiple of $2^2 = 4$.
- The directive `ALIGN 4, 3` forces the LC to align to a 4-byte boundary, then adds an offset of 3. For example, if the current LC is `0x08000002`, aligning to 4 bytes with an offset of 3 would require finding the smallest N such that:

  $$N(4) + 3 \geq 0x08000002$$

  and typically, the assembler will adjust LC to the nearest address that satisfies the alignment constraint (e.g., if aligning to 8 bytes, LC becomes `0x08000008`).
- The purpose is to minimize the value of N while meeting the alignment requirement, thereby ensuring data is placed at the proper memory boundaries.

Summary

The `ALIGN` directive:

- Ensures that code and data are stored at addresses that are multiples of a given power of two.
- Adjusts the location counter (LC) to the nearest required boundary, optionally with an offset.
- Helps to satisfy hardware constraints and improve performance in memory access.

For instance, if the LC is initially `0x08000002` and an alignment of $2^3 = 8$ is required, the assembler will adjust the LC to `0x08000008`.

Directive: INCLUDE or GET

The `INCLUDE` or `GET` directive is used to incorporate one assembly source file within another. This is particularly useful for including constant definitions (via `EQU`) stored in a separate file.

```
INCLUDE constants.s  ; Load constant definitions

AREA main, CODE, READONLY
EXPORT __main
ENTRY
__main PROC
  ...
ENDP
END
```

Data Addressing Modes

Data addressing in assembly can be performed using several modes. The most common modes include:

- **Immediate:** The data is directly contained within the instruction.
- **Pre-index:** The address is computed by adding an offset to a base register before accessing memory.
- **Post-index:** The memory is accessed first and then an offset is added to the base register.
- **Pre-index with Update:** Similar to pre-index addressing, but the base register is updated with the computed address.

Immediate Addressing

In immediate addressing, the data is embedded in the instruction itself, making it immediately available when the instruction is decoded.

```
MOV R0, #100  ; R0 is loaded with the immediate value 100
```

Register-Indexed Addressing

In register-indexed addressing, a register holds the memory address (i.e., it points to data in memory). This mode is useful for accessing elements in an array or structure.

```
LDR R0, [R1]  ; R0 is loaded with the value pointed to by R1
```

Pre-index Addressing

Pre-index addressing computes the effective address by adding an immediate offset to the base register before the memory access.

```
LDR R1, [R0, #4]  ; Compute address: R0 + 4, then load the value into R1
```

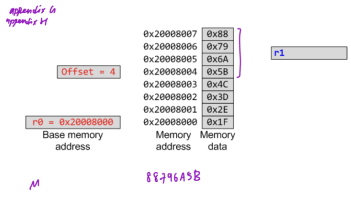


Figure 12.1: Pre-Index with Annotations.

Post-index Addressing

In post-index addressing, the memory access is performed first using the address in the base register, and then the offset is added to update the base register.

```
LDR R1, [R0], #4  ; Load value from address in R0 into R1, then update R0 to R0+4
```

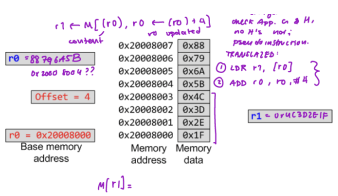


Figure 12.2: Post-Index with Annotations.

Pre-index with Update Addressing

Pre-index with update addressing is similar to pre-index addressing, but in this mode, the base register is updated with the computed address after the offset is added.

```
LDR R1, [R0, #4]!  ; Compute address (R0 + 4), load value into R1, and update R0 to (R0+4)
```

Figure 12.3: Pre-Index with Updates.

Advanced Operations 13

SIMD: Single Instruction, Multiple Data

Performing addition and subtraction simultaneously is a core advantage of SIMD (Single Instruction, Multiple Data) processing. This concept is covered in more depth in **EE125** and **EE155**.

64-bit Subtraction Example

The following assembly example demonstrates a 64-bit subtraction operation where $C = A - B$.

```
; 64-bit Subtraction
; C = A - B
; A = 00000002FFFFFFFF
; B = 0000000400000001
; Result stored in r5 (upper) and r4 (lower)

    LDR r0, =0xFFFFFFFF  ; Load lower part of A
    LDR r1, =0x00000002  ; Load upper part of A
    LDR r2, =0x00000001  ; Load lower part of B
    LDR r3, =0x00000004  ; Load upper part of B

    SUBS r4, r0, r2      ; Subtract lower part with carry
    SBC  r5, r1, r3      ; Subtract upper part with borrow

    ; Result stored in (r5, r4) as C
```

Two's and Sixteen's Complement

r0 - r2: $0xFFFFFFFF - 0x00000001$ (Two's complement), $+0xFFFFFFFF$

Two's complement inversion results in adding one to the negated value.

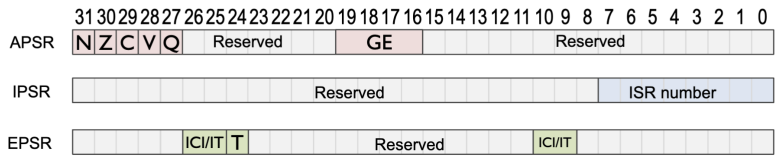


Figure 13.1: Program Status Register

Sixteen's Complement

Sixteen's complement is derived by subtracting each digit from F until the last nonzero digit.

Example:

Given: $0x28800000$

1. Subtract each digit from F:

$$F - 2 = D, \quad F - 8 = 7, \quad F - 8 = 7, \quad F - 0 = F$$

2. The sixteen's complement of $0x28800000$ is:

$$0xD77FFFFF$$

8-bit subtraction using a 4-bit adder/subtractor:

8-bit subtraction (normally)
$11000111 - 00011000$
$= 11000111 + (\neg 00011000 + 1)$
$= 11000111 + 11101000$
$= 10101111$

$$N = 1, \quad Z = 0, \quad C = 1, \quad V = 0$$

Using a 4-bit adder/subtractor:

First 4-bit subtraction (SUBS)
$0111 - 1000$
$= 0111 + (\neg 1000 + 1)$
$= 0111 + 1000$
$= 1111$

Second 4-bit subtraction (SBC with borrow)
$1100 - 1110$
$= 1100 + (\neg 1110 + 1)$
$= 1100 + 0010$
$= 1110$

Final result: 11101111

The subtraction is split into two 4-bit operations. The first subtraction results in 1111. The second subtraction accounts for the borrow and produces 1110, combining to form the full 8-bit result: 11101111.

ARM Arithmetic and Logic Instructions

14

Key Concepts: ARM Instructions

- Instructions often follow a consistent syntax pattern with optional condition codes and flag updates.
- Most arithmetic and logic instructions can update the status register (PSR) flags (N, Z, C, V) using the 'S' suffix.
- The Program Status Register (PSR) contains crucial flags like N (Negative), Z (Zero), C (Carry), and V (Overflow) that reflect the outcome of operations.
- Many instructions feature a flexible second operand (`Operand2`), which can be an immediate value or a register, potentially modified by the barrel shifter.

The 'S' suffix is key for conditional execution and multi-word math.

`Operand2` flexibility is a hallmark of ARM, allowing complex operations in one instruction.

Instruction Overview

The ARM instruction set provides a comprehensive suite of operations categorized as follows:

- **Shift Operations**: Logic Shift Left (`LSL`), Logic Shift Right (`LSR`), Arithmetic Shift Right (`ASR`), Rotate Right (`ROR`), Rotate Right with Extend (`RRX`).
- **Logic Operations**: Bitwise AND (`AND`), OR (`ORR`), Exclusive OR (`EOR`), OR NOT (`ORN`), Move NOT (`MVN`).
- **Bit Manipulation**: Bit Field Clear (`BFC`), Bit Field Insert (`BFI`), Bit Clear (`BIC`), Count Leading Zeroes (`CLZ`).
- **Bit/Byte Reordering**: Reverse Bit (`RBIT`), Reverse Byte (`REV`, `REV16`, `REVSH`).
- **Addition**: Add (`ADD`), Add with Carry (`ADC`).
- **Subtraction**: Subtract (`SUB`), Reverse Subtract (`RSB`), Subtract with Carry (`SBC`).
- **Multiplication**: Multiply (`MUL`), Multiply-Accumulate (`MLA`), Multiply-Subtract (`MLS`), Signed/Unsigned Multiply Long (`SMULL`, `UMULL`), Signed/Unsigned Multiply-Accumulate Long (`SMLAL`, `UMLAL`).
- **Division**: Signed Divide (`SDIV`), Unsigned Divide (`UDIV`).
- **Saturation**: Signed Saturate (`SSAT`), Unsigned Saturate (`USAT`).
- **Sign Extension**: Signed Extend Byte/Halfword (`SXTB`, `SXTH`), Unsigned Extend Byte/Halfword (`UXTB`, `UXTH`).
- **Bit Field Extract**: Signed/Unsigned Bit Field Extract (`SBFX`, `UBFX`).

`REV` instructions are useful for endianness conversions.

Hardware division is available on Cortex-M3 and later.

Saturation prevents wrap-around, common in DSP.

Common Instruction Pitfalls

- Forgetting the 'S' suffix when flags are needed for conditional execution or multi-word arithmetic.
- Incorrectly handling the Carry (C) flag in multi-word addition (`ADC`) or subtraction (`SBC`).
- Choosing the wrong condition codes (`<cond>`) for conditional branching (e.g., using signed vs. unsigned comparisons incorrectly).
- Using logical shifts (`LSR`) instead of arithmetic shifts (`ASR`) for signed numbers, or vice-versa.

The C flag acts as borrow in subtraction.

`ASR` preserves the sign bit.

General Instruction Syntax

Most ARM data processing instructions adhere to a common syntax:

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

Where:

- `<Operation>` is the instruction mnemonic (e.g., `ADD`, `SUB`, `MOV`).
- `{<cond>}` is an optional condition code suffix (e.g., `EQ`, `NE`, `CS`) that allows conditional execution based on PSR flags.
- `{S}` is an optional suffix that causes the instruction to update the N, Z, C, and V flags in the APSR.
- `Rd` is the destination register.
- `Rn` is the register holding the first source operand.
- `Operand2` is the second source operand. This is flexible and can be an immediate value (a constant prefixed with #), a register (`Rm`), or a register value shifted using the barrel shifter (`Rm, <shift> #imm`).

E.g., `ADDEQ` adds only if the Z flag is set.

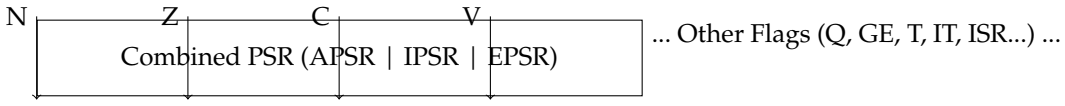
Program Status Register (PSR)

The PSR holds processor status and control information. It's a composite of APSR, IPSR, and EPSR. The Application PSR (APSR) contains the condition flags updated by instructions with the `S` suffix:

- **N** (Negative): Set if the result is negative (MSB is 1).
- **Z** (Zero): Set if the result is zero.
- **C** (Carry): Meaning depends on the operation. For addition, set if there's an unsigned overflow. For subtraction, set if no borrow occurred. Also affected by shifts.
- **V** (Overflow): Set if a signed overflow occurred (result is outside the representable signed range).

MSB: Most Significant Bit.

Signed overflow is different from unsigned carry.



APSR Condition Flags shown

UAL vs. Thumb Syntax

Modern ARM assembly uses the Unified Assembler Language (UAL), which provides a more consistent syntax than the older Thumb syntax.

UAL is generally preferred for clarity and compatibility.

```
; UAL Syntax (typically 3 operands)
ADD r1, r2, r3      ; r1 = r2 + r3
ADD r1, r2, #4      ; r1 = r2 + 4

; Traditional Thumb Syntax (often 2 operands, dest=src1)
ADD r1, r3          ; r1 = r1 + r3
ADD r1, #15         ; r1 = r1 + 15
```

Arithmetic Instructions

Addition (ADD, ADC)

Understanding Multi-Word Arithmetic

Operations on data larger than the native register size (e.g., 64-bit math on a 32-bit CPU) require breaking the operation into stages.

- Split the operation into 32-bit chunks (e.g., lower and upper halves).
- Process the least significant chunk first using an instruction that sets the Carry flag (`ADDS`, `SUBS`).
- Process subsequent chunks using instructions that incorporate the Carry flag (`ADC`, `SBC`).
- Requires careful management of register pairs to hold the multi-word values.

Example: 64-bit addition uses `ADDS` for the lower 32 bits and `ADC` for the upper 32 bits.

The 'S' is vital here!

`ADC` adds the carry, `SBC` subtracts the borrow.

The `ADD` instruction performs addition. `ADC` adds the operands **plus** the value of the Carry flag.

```
; Basic addition examples
ADD r1, r2, r3      ; r1 = r2 + r3
ADD r1, r2, #4      ; r1 = r2 + 4

; Example: Update flags (Z=1, C=1)
LDR r0, =0xFFFFFFFF ; r0 = -1
LDR r1, =0x00000001 ; r1 = 1
ADDS r3, r0, r1     ; r3 = r0+r1 = 0. Sets Z=1, C=1

; 64-bit addition: C = A + B
; A in r1:r0, B in r3:r2, Result C in r5:r4
LDR r0, =0xFFFFFFFF ; Lower 32 bits of A
LDR r1, =0x00000002 ; Upper 32 bits of A
LDR r2, =0x00000001 ; Lower 32 bits of B
LDR r3, =0x00000004 ; Upper 32 bits of B
```

```
ADDS r4, r0, r2     ; r4 = A[31:0]+B[31:0]. Updates Carry
ADC r5, r1, r3      ; r5 = A[63:32]+B[63:32]+Carry
```

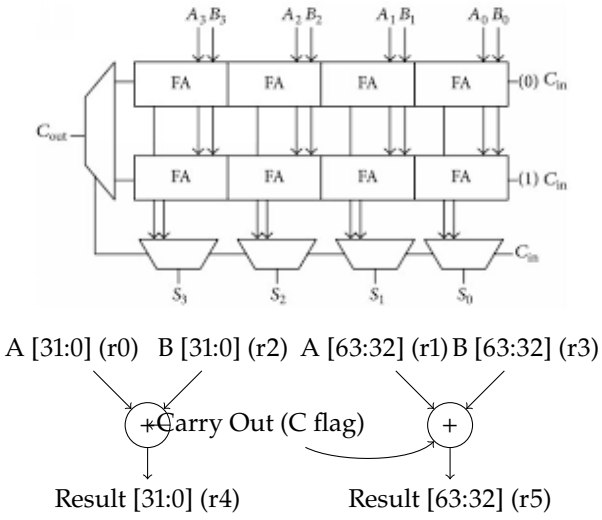


Figure 14.1: Conceptual Flow of 64-bit Addition using ADDS and ADC.

Subtraction (SUB, SBC, RSB)

`SUB` performs subtraction. `SBC` subtracts the operands *and* subtracts the borrow indicated by the Carry flag (calculates `Rd = Rn - Op2 - (1-C)`). `RSB` (Reverse Subtract) calculates `Rd = Op2 - Rn`.

C=1 means *no* borrow, C=0 means borrow occurred.

`RSB` is useful for negating numbers: `RSB r0, r0, #0`.

```
; Basic subtraction
SUB r1, r2, r3      ; r1 = r2 - r3
SUB r1, r2, #4      ; r1 = r2 - 4
RSB r1, r2, #0      ; r1 = 0 - r2 (negation)

; 64-bit subtraction: C = A - B
; A in r1:r0, B in r3:r2, Result C in r5:r4
LDR r0, =0xFFFFFFFF ; Lower 32 bits of A
LDR r1, =0x00000002 ; Upper 32 bits of A
LDR r2, =0x00000001 ; Lower 32 bits of B
LDR r3, =0x00000004 ; Upper 32 bits of B
SUBS r4, r0, r2     ; r4 = A[31:0]-B[31:0]. Updates Carry (Borrow)
SBC r5, r1, r3      ; r5 = A[63:32]-B[63:32]-Borrow (1-C)
```

Multiplication (MUL, MLA, MLS, UMULL, SMULL, UMLAL, SMLAL)

Multiplication Optimization Tips

- Use shifts (`LSL`) for multiplication by powers of 2 (e.g., `x LSL # 3` is x * 8).
- Combine shifts and adds/subs for small constants (e.g., `ADD r1, r0, r0, LSL # 3` calculates r1 = r0 * 9).
- Use `MLA` for fused multiply-add operations (`a*b + c`) common in loops and filters.
- Use long multiplication (`UMULL`, `SMULL`) only when the full 64-bit result is necessary.

Example: x * 10 can be calculated as '(x « 3) + (x « 1)'.

Replaces a slower `MUL` instruction.

Often single-cycle.

ARM provides instructions for both 32-bit and 64-bit results.

- **MUL**: 32x32 -> lower 32 bits. `MUL r6, r4, r2` ; r6 = (r4 * r2)[31:0].
- **MLA**: Multiply and Accumulate. `MLA r6, r4, r1, r0` ; r6 = r0 + (r4 * r1)[31:0].
- **MLS**: Multiply and Subtract. `MLS r6, r4, r1, r0` ; r6 = r0 - (r4 * r1)[31:0].
- **UMULL**: Unsigned Multiply Long (32x32 -> 64). `UMULL RdLo, RdHi, Rn, Rm`.
- **SMULL**: Signed Multiply Long (32x32 -> 64). `SMULL RdLo, RdHi, Rn, Rm`.
- **UMLAL**: Unsigned Multiply Accumulate Long. `UMLAL RdLo, RdHi, Rn, Rm` adds the 64-bit product to the existing 64-bit value in RdHi:RdLo.
- **SMLAL**: Signed Multiply Accumulate Long. `SMLAL RdLo, RdHi, Rn, Rm`.

Result is the same for signed/unsigned if only lower 32 bits needed.

Computes `Ra + (Rn * Rm)`.

Computes `Ra - (Rn * Rm)`.

Result in two registers: `RdHi:RdLo`.

Handles signed operands correctly for 64-bit result.

```
; Long multiplication example (Unsigned)
UMULL r3, r4, r0, r1  ; r4:r3 = r0 * r1 (r4=MSB, r3=LSB)

; Signed Multiply Accumulate Long example
SMLAL r3, r4, r0, r1  ; r4:r3 = r4:r3 + (signed)(r0 * r1)
```

Division (SDIV, UDIV)

Cortex-M processors (like M3/M4/M7) include hardware division instructions.

- **SDIV**: Signed Division. `SDIV Rd, Rn, Rm` ; Rd = Rn / Rm.
- **UDIV**: Unsigned Division. `UDIV Rd, Rn, Rm` ; Rd = Rn / Rm.

Division often takes more cycles than multiplication.

Other Arithmetic (Saturation, Sign Extension)

- **Saturation (SSAT, USAT)**: Clamp the result of an operation (often involving shifts) to a specified signed or unsigned range. Useful in DSP to prevent unexpected wrap-around on overflow.

- **Sign/Zero Extension (SXTB, SXTH, UXTB, UXTH)**: Extend a byte or halfword value in a register to a full 32-bit word, either preserving the sign (SX) or padding with zeros (UX).

Essential when loading smaller types into 32-bit registers for arithmetic.

Bitwise Logic Instructions

These instructions operate on individual bits of the operands.

- **AND**: Bitwise AND. `AND Rd, Rn, Op2` ; Rd = Rn & Op2. Result bit is 1 only if both input bits are 1.
- **ORR**: Bitwise OR (Inclusive OR). `ORR Rd, Rn, Op2` ; Rd = Rn | Op2. Result bit is 1 if either input bit is 1.
- **EOR**: Bitwise Exclusive OR (XOR). `EOR Rd, Rn, Op2` ; Rd = Rn Ôp2. Result bit is 1 if input bits are different.
- **ORN**: Bitwise OR NOT. `ORN Rd, Rn, Op2` ; Rd = Rn | (Op2).
- **BIC**: Bit Clear. `BIC Rd, Rn, Op2` ; Rd = Rn & (Op2). Clears bits in Rn where Op2 has a 1.
- **MVN**: Move NOT (Bitwise Negation). `MVN Rd, Op2` ; Rd = Op2.

Often used for toggling bits.

Very useful for clearing status flags or masking.

```
; Logic Examples
AND r2, r0, r1      ; r2 = r0 & r1
ORR r2, r0, r1      ; r2 = r0 | r1
EOR r2, r0, r0      ; r2 = r0 ^ r0 = 0 (clear register)
MVN r1, r0          ; r1 = ~r0 (bitwise not)
BIC r0, r0, #0x0F   ; Clear the lower 4 bits of r0
```

Bit Field Instructions (BFC, BFI, SBFX, UBFX)

These instructions manipulate contiguous sequences of bits (bit fields).

Useful for packed data or register manipulation.

- **BFC**: Bit Field Clear. `BFC Rd, #lsb, #width` ; Clears 'width' bits in Rd starting at 'lsb'.
- **BFI**: Bit Field Insert. `BFI Rd, Rn, #lsb, #width` ; Inserts the bottom 'width' bits of Rn into Rd starting at 'lsb'.
- **SBFX/UBFX**: Signed/Unsigned Bit Field Extract. `SBFX Rd, Rn, #lsb, #width` ; Extracts 'width' bits starting at 'lsb' from Rn into the LSBs of Rd, sign-extending (SBFX) or zero-extending (UBFX).

lsb: least significant bit position of the field.

width: number of bits in the field.

```
; Bit Field Examples
BFC R4, #8, #12     ; Clear bits 19:8 of R4
BFI R9, R2, #8, #12 ; Copy bits 11:0 of R2 into bits 19:8 of R9
UBFX R0, R1, #4, #8 ; Extract bits 11:4 of R1 into R0 (zero-padded)
```

Bitwise vs. Boolean Operators (in C)

It's crucial to distinguish C's bitwise operators from its logical Boolean operators.

Using the wrong operator is a common C bug!

- **Bitwise**: & (AND), | (OR), ^ (XOR), ~ (NOT). Operate independently on each bit across the integer's width.

- **Boolean**: && (Logical AND), || (Logical OR), ! (Logical NOT). Treat non-zero values as true, zero as false, and perform short-circuit evaluation.

```
// Examples in C
int a = 0x10; // Binary 0001 0000
int b = 0x01; // Binary 0000 0001

int bitwise_and = a & b; // Result: 0x00 (0)
int logical_and = a && b; // Result: 1 (true, because both a and b are non-zero)

int bitwise_not_b = ~b;   // Result: 0xFFFFFFFE (assuming 32-bit int)
int logical_not_b = !b;   // Result: 0 (false, because b is non-zero/true)
```

Common C Bit Manipulation Idioms

Let `a` be a variable and `k` be the bit position (0-indexed).

- **Check bit k**: `if (a & (1 << k))` ...
- **Set bit k**: `a |= (1 << k);`
- **Clear bit k**: `a &= ~(1 << k);`
- **Toggle bit k**: `a ^= (1 << k);`

`(1 << k)` creates a mask with only bit `k` set.

Forces bit `k` to 1.

`~(1 << k)` creates a mask with only bit `k` clear.

Flips the value of bit `k`.

Memory Access

Memory Access Best Practices

- Respect data alignment: Accessing words (`LDR`/`STR`) on word boundaries (address divisible by 4) and halfwords (`LDRH`/`STRH`) on halfword boundaries (address divisible by 2) is crucial for performance and sometimes correctness on ARM. Unaligned access might cause faults or be significantly slower (emulated in software).
- Use Load/Store Multiple (`LDM`/`STM`) instructions for transferring several registers efficiently, especially for stack operations or block copies.
- Choose appropriate addressing modes (offset, pre-indexed, post-indexed) to match the access pattern (e.g., array access, stack push/pop).
- Be mindful of memory barriers (`DMB`, `DSB`, `ISB`) when dealing with memory shared between different execution units (e.g., CPU and DMA) or requiring strict ordering for peripherals.

Bytes (`LDRB`/`STRB`) can be accessed at any address.

Ensures memory operations complete in the expected order.

Addressing Modes

`LDR` and `STR` support several ways to calculate the memory address:

- **Offset Addressing**: `[Rn, #offset]` or `[Rn, Rm]`. The address is base register (`Rn`) + offset. The base register `Rn` is *not* changed.

Simple access relative to a base pointer.

- **Pre-indexed Addressing**: `[Rn, #offset]!` or `[Rn, Rm]!`. The address is calculated as base + offset, the transfer occurs, and then the base register `Rn` is updated with the calculated address.
- **Post-indexed Addressing**: `[Rn], #offset` or `[Rn], Rm`. The transfer occurs using the address currently in the base register `Rn`. Then, the base register `Rn` is updated by adding the offset.

Update pointer *before* access. Useful for sequential access.

Update pointer *after* access. Also good for sequential access.

```
; Examples: Assume r1 = 0x1000 initially
LDR r0, [r1, #4]    ; Offset: r0=Mem[0x1004]. r1 still 0x1000
LDR r0, [r1, #4]!   ; Pre-index: r0=Mem[0x1004]. r1 becomes 0x1004
LDR r0, [r1], #4    ; Post-index: r0=Mem[0x1000]. r1 becomes 0x1004

STRB r0, [r1]       ; Store lower byte of r0 to Mem[0x1000]
LDRSH r2, [r1]      ; Load signed halfword from Mem[0x1000] into r2
```

Load/Store Multiple Registers (LDM, STM)

These instructions allow transferring multiple registers between the core and memory in a single instruction, significantly improving efficiency for tasks like context switching, function prologues/epilogues, and block memory copies.

Much faster than repeated LDR/STR for >2 registers.

Syntax and Addressing Modes

The general syntax is `LDM<mode> Rn!, reglist` and `STM<mode> Rn!, reglist`.

- `Rn` is the base register containing the starting memory address (often the stack pointer, SP).
- `{!}` is optional writeback. If present, the base register `Rn` is updated after the *entire* transfer.
- `{reglist}` is a list of registers enclosed in braces, e.g., `{r0-r3, r5, lr}`.
- `<mode>` specifies how the address changes during the transfer, combined with whether the pointer updates before/after each individual register transfer:
  - **IA** (Increment After): Address increases after each transfer. Base addr -> Base addr + 4*N.
  - **IB** (Increment Before): Address increases before each transfer. Base addr + 4 -> Base addr + 4*N.
  - **DA** (Decrement After): Address decreases after each transfer. Base addr -> Base addr - 4*N.
  - **DB** (Decrement Before): Address decreases before each transfer. Base addr - 4 -> Base addr - 4*N.

N = number of registers in list.

Memory Operation Optimization

Performance Tips:

- Use LDM/STM for transferring 3 or more registers; it's usually faster than multiple LDR/STRs.
- Ensure data structures accessed frequently are aligned appropriately (e.g., 4-byte alignment for words) to avoid performance penalties or faults. Cache line alignment (if applicable) can further improve performance for larger structures.
- Minimize unnecessary memory accesses; keep frequently used values in registers.
- Use appropriate addressing modes (pre/post-indexed) for efficient pointer updates in loops (e.g., array processing).

Data Manipulation and Memory Access 15

Barrel Shifter

The barrel shifter is a hardware component that allows efficient bitwise shifting and rotation of values within registers. It is commonly used in arithmetic operations, data alignment, and fast multiplication.

Shift and Rotate Operations

- **Logical Shift Left (LSL)** – Moves all bits left by a specified number of places, inserting zeros on the right. Used for fast multiplication by powers of two.
- **Logical Shift Right (LSR)** – Moves all bits right by a specified number of places, inserting zeros on the left. Often used for division by powers of two.
- **Arithmetic Shift Right (ASR)** – Similar to LSR but preserves the sign bit (MSB) when shifting right, useful for signed division.
- **Rotate Right (ROR)** – Moves all bits right in a circular fashion, with bits wrapping around from the right end to the left.
- **Rotate Right Extended (RRX)** – Similar to ROR, but also includes the carry flag as an additional bit in the rotation.

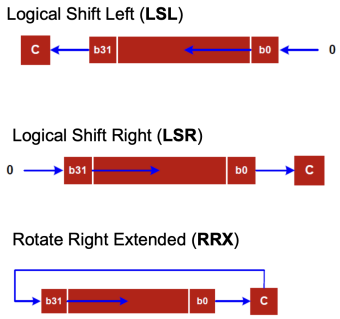


Figure 15.1: Barrel Shifter: Logical and RRX Shifts.

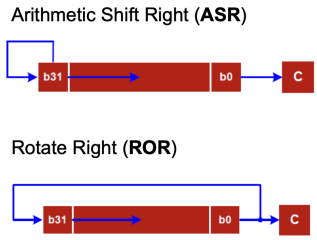


Figure 15.2: Barrel Shifter: Arithmetic and ROR Rotations.

Why No Rotate Left?

There is no explicit rotate left instruction because it can be achieved using `ROR` with a different offset. Specifically, rotating left by n bits is equivalent to rotating right by $(32 - n)$ bits in a 32-bit register.

Implementation of Barrel Shifter

A barrel shifter is typically implemented using a cascade of parallel 2-to-1 multiplexers, allowing efficient bitwise shifts and rotations. The following example illustrates a four-bit barrel shifter performing a rotate right operation.

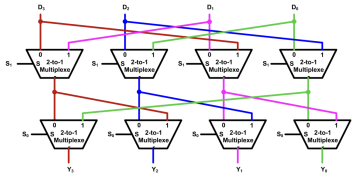


Figure 15.3: Cascade of 2-to-1 Multiplexers.

Example: Four-bit Barrel Shifter (Rotate Right)

The table to the right shows how the barrel shifter rotates a four-bit value based on control signals S_1 and S_0:

Each bit is shifted in parallel, wrapping around cyclically to implement the rotate right operation.

S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	D_3	D_2	D_1	D_0
0	1	D_0	D_3	D_2	D_1
1	0	D_1	D_0	D_3	D_2
1	1	D_2	D_1	D_0	D_3

Figure 15.4: Four-bit Barrel Shifter Structure.

Barrel Shifter Examples

The barrel shifter is commonly used in ARM assembly instructions to perform efficient arithmetic and logical operations. Below are some examples demonstrating its usage:

```
; Example 1: Left Shift (Multiply by 8)
ADD r1, r0, r0, LSL #3
; r1 = r0 + (r0 << 3) = r0 + 8 * r0

; Example 2: Logical Right Shift (Unsigned Division by 8)
ADD r1, r0, r0, LSR #3
; r1 = r0 + (r0 >> 3) = r0 + (r0 / 8) (unsigned)

; Example 3: Arithmetic Right Shift (Signed Division by 8)
ADD r1, r0, r0, ASR #3
; r1 = r0 + (r0 >> 3) = r0 + (r0 / 8) (signed)
```

Using Barrel Shifter for Optimization

The barrel shifter can be used to optimize multiplication operations, reducing instruction count and improving execution speed. The following example shows how a multiplication by 9 can be rewritten using the barrel shifter:

```
; Optimized multiplication using Barrel Shifter
ADD r1, r0, r0, LSL #3
; Equivalent to r1 = r0 * 9

; Equivalent multiplication using a separate register
MOV r2, #9  ; Load constant 9 into r2
MUL r1, r0, r2  ; Multiply r0 by 9
```

Endianness

Endianness refers to the byte order used when storing multi-byte values in memory. It determines how the least significant and most significant bytes are arranged.

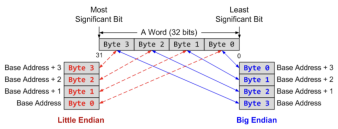


Figure 15.5: Little vs. Big Endian Representation.

Little Endian

- The least significant byte (LSB) is stored at the lowest memory address.
- Used by ARM (default), x86, and most modern processors.

Big Endian

- The most significant byte (MSB) is stored at the lowest memory address.
- Used by some legacy architectures (e.g., Motorola 68k, PowerPC in certain configurations).

Addressing Convention

Regardless of the endianness, the memory address of a word is defined as the lowest address of all the bytes it occupies.

Endianness in ARM

- ARM is **Little Endian** by default.
- It can be configured to operate in **Big Endian** mode if required.

Load-Modify-Store

$$x = x + 1;$$

The Load-Modify-Store sequence is a common approach used in assembly programming to read a value from memory, modify it, and store it back. This operation is essential for updating variables in memory.

```
; Assume the memory address of x is stored in r1
LDR r0, [r1]   ; Load value of x from memory
ADD r0, r0, #1 ; x = x + 1
STR r0, [r1]   ; Store updated x back into memory
```

Single Register Data Transfer

The following table summarizes the different load and store instructions for word, byte, and halfword transfers in ARM assembly:

Instruction	Description
LDR	Load Word
LDRB	Load Byte
LDRH	Load Halfword
LDRSB	Load Signed Byte
LDRSH	Load Signed Halfword
STR	Store Word
STRB	Store Lower Byte
STRH	Store Lower Halfword

Load/Store Multiple Registers

The ARM architecture allows transferring multiple registers in a single instruction using the `STM` (Store Multiple) and `LDM` (Load Multiple) instructions. These are useful for saving and restoring register states efficiently, such as during function calls or context switching.

Synonyms for STM and LDM

The following instructions are functionally equivalent:

- **STM** (Store Multiple) = **STMIA** (Increment After) = **STMEA** (Empty Ascending)
- **LDM** (Load Multiple) = **LDMIA** (Increment After) = **LDMFD** (Full Descending)

Register Order and Memory Addressing

For `STM/LDM` operations: - The order in which registers are listed in the instruction does not affect execution. - The lowest-numbered register is stored/loaded at the lowest memory address.

Example: Stack Growth and Memory Addressing

Consider a memory size of 64 KB, starting at address `0x20080000`. Depending on the stack configuration, the Stack Pointer (`SP`) determines whether the stack grows upwards (ascending) or downwards (descending):

- SP1 = `0x20080000`, Stack Grows Upwards (Ascending) → Full Stack - SP2 = `0x20080004`, Stack Grows Downwards (Descending) → Empty Stack

A full descending stack (`FD`) stores data before decrementing the stack pointer, while an empty ascending stack (`EA`) stores data after incrementing it.

Stack Pointer Values for Different Stack Types

The stack pointer (`SP`) values for different stack configurations:

- **Full Ascending (FA):** `SP = 0x2007FFFC`
- **Empty Ascending (EA):** `SP = 0x20080000`
- **Full Descending (FD):** `SP = 0x20090000`
- **Empty Descending (ED):** `SP = 0x2008FFFC`

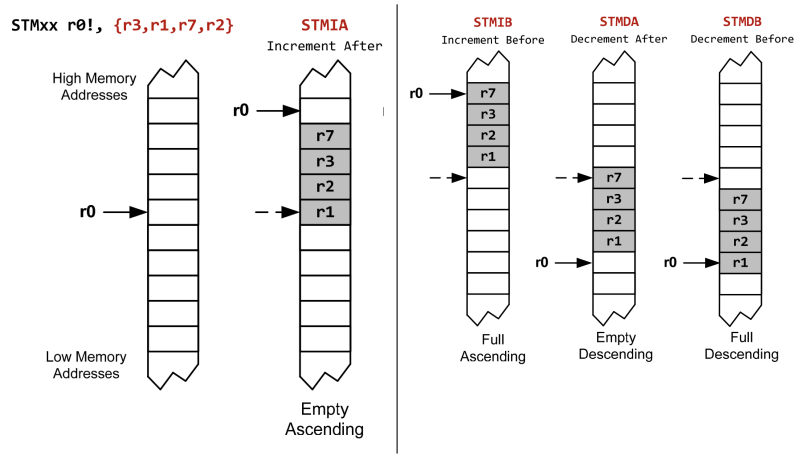
Understanding Stack Behavior

- A **full stack** means the stack pointer references the last used address. - An **empty stack** means the stack pointer references the next available address. - The stack grows either **ascending** (higher addresses) or **descending** (lower addresses). - ARM defaults to a **full descending stack (FD)**, but it can be configured differently.

This section provides a conceptual understanding of stack behavior in ARM, crucial for efficient memory management and function call handling.

Store Multiple Registers

The `STM` (Store Multiple) instruction allows saving multiple registers to memory in a single operation, which is useful for preserving context during function calls or interrupt handling.



Assembly Control Flow: Loops and Conditionals

16

This chapter provides a detailed examination of control flow mechanisms in ARM assembly, specifically focusing on the implementation of C-style `for` loops, conditional execution based on processor flags, specialized branching instructions, and the `IT` (If-Then) instruction block.

For Loop Implementations

Translating C loops to assembly.

The standard C `for` loop, used for iteration, can be translated into assembly in multiple ways. We consider the example: `int sum=0; for(i=0; i<10; i++)  sum += i; .`

Implementation 1: Check at Bottom

This approach executes the loop body at least once (if the initial condition allows entry via the first branch) and checks the condition at the end.

```
; C Code: for(i=0; i<10; i++) { sum += i; }
; r0 = i, r1 = sum
  MOV r0, #0       ; i = 0
  MOV r1, #0       ; sum = 0
  B check          ; Branch to initial condition check

loop:              ; Start of loop body
  ADD r1, r1, r0   ; sum += i
  ADD r0, r0, #1   ; i++

check:             ; Condition check point
  CMP r0, #10      ; Compare i with 10
  BLT loop         ; Branch to loop if i < 10 (Signed Less Than)

endloop:           ; Loop exit
  ; Final sum is in r1
```

The flowchart shows initialization, an immediate jump to the check, and then looping between the body and the check until `i < 10` is false.

Implementation 2: Check at Top

More common 'for' loop translation.

This implementation checks the condition at the beginning of each iteration. This is often considered a more standard translation of a C `for` loop.

```
; C Code: for(i=0; i<10; i++) { sum += i; }
; r0 = i, r1 = sum
  MOV r0, #0       ; i = 0
  MOV r1, #0       ; sum = 0
```

```
loop:               ; Start of loop iteration (includes check)
  CMP r0, #10       ; Compare i with 10
  BGE endloop       ; Branch if i >= 10 (Signed Greater or Equal)

  ; Loop body
  ADD r1, r1, r0    ; sum += i
  ADD r0, r0, #1    ; i++
  B loop            ; Branch back to loop check

endloop:            ; Loop exit
  ; Final sum is in r1
```

The flowchart shows initialization followed by the loop block which starts with the condition check (`i<10`). If true, the body executes, and it loops back; if false, it exits.

Conditional Execution

Using processor flags (N,Z,C,V).

ARM instructions can be executed conditionally based on the Application Program Status Register (APSR) flags (N, Z, C, V), typically set by a preceding `CMP` or data processing instruction with the 'S' suffix.

Condition Codes and Flags Tested

A condition suffix can be added to many instructions. The following table details these conditions:

Instruction Suffix	Condition Description	Flags Tested
`EQ`	Equal	Z = 1
`NE`	Not Equal	Z = 0
`HS / CS`	Unsigned Higher or Same / Carry Set	C = 1
`LO / CC`	Unsigned Lower / Carry Clear	C = 0
`MI`	Minus / Negative	N = 1
`PL`	Plus / Positive or Zero	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

Conditional Execution Examples

Simple If-Else Statement

Implementing `if (a <= 0) y = -1; else y = 1;` where `a` is in R0 and `y` is in R1:

```
CMP r0, #0       ; Compare a with 0, setting flags
MOVLE r1, #-1    ; If Less than or Equal (Z=1 or N!=V), r1 = -1
MOVGT r1, #1     ; If Greater Than (Z=0 and N==V), r1 = 1
```

This uses the `LE` and `GT` conditions to execute one of the two `MOV` instructions based on the result of the `CMP`.

If-Else with Multiple OR Conditions

Implementing `if (a == 1 || a == 7 || a == 11) y = 1; else y = -1;` where `a` is in R0 and `y` is in R1:

```
CMP r0, #1       ; Compare a with 1
CMPNE r0, #7     ; If Not Equal, compare a with 7
CMPNE r0, #11    ; If Still Not Equal, compare a with 11
; At this point, Z=1 if any of the comparisons resulted in Equal
MOVEQ r1, #1     ; If Equal (Z=1), set y = 1
MOVNE r1, #-1    ; If Not Equal (Z=0 after last CMP), set y = -1
```

This relies on the fact that `CMPNE` does not set the Z flag if its condition (NE) is met, only clearing it if the comparison finds equality. Therefore, the Z flag remains set from a previous `CMP` if an equality was found.

Relies on flags not being set by intermediate CMPNE if condition met.

Compound Boolean Expression

Implementing `if(x <= 20 || x >= 25)  a = 1;` where `x` is signed in R0, `a` in R1:

```
; r0 = x, r1 = a
CMP r0, #20      ; Compare x and 20
MOVLE r1, #1     ; a = 1 if Less or Equal (x <= 20)
; Need to ensure 'a' isn't reset if the first condition was true
; but the second isn't. The provided example uses:
CMP r0, #25      ; Compare x and 25
MOVGE r1, #1     ; a = 1 if Greater or Equal (x >= 25)
```

Note: As mentioned before, this specific sequence might overwrite `r1` incorrectly depending on the value of `x`. A conditional branch approach might be safer for strict C equivalence..

Sequential MOVs might have side effects.

Compare and Branch Instructions

Optimized compare-with-zero and branch.

These instructions provide a compact way to compare a register with zero and branch conditionally, without altering the condition flags.

- `CBZ Rn, label`: Compare and Branch if Zero. Branches if `Rn` is 0. Equivalent to: `CMP Rn, #0`; `BEQ label` (but flags are unaffected).
- `CBNZ Rn, label`: Compare and Branch if Non Zero. Branches if `Rn` is not 0. Equivalent to: `CMP Rn, #0`; `BNE label` (but flags are unaffected).

Loop Control: Break and Continue

Illustrative C code shows the behavior of `break` and `continue`.

- **Break Example**: `for(int i=0; i<5; i++) if (i==2) break; printf("%d, ", i);` -> Output: `0, 1, .` Exits loop early.
- **Continue Example**: `for(int i=0; i<5; i++) if (i==2) continue; printf("%d, ", i);` -> Output: `0, 1, 3, 4, .` Skips current iteration.

Assembly Example: String Length using Break Logic

Finding the length of a null-terminated string demonstrates using `CBNZ` to exit a loop (similar to `break`) when a null character is found.

```
; Input: r0 = string memory address
; Output: r1 = string length
  MOV r1, #0       ; len = 0
loop:
  LDRB r2, [r0]    ; Load byte (*str) into r2
  ; Check if the byte is the null terminator ('\0')
  CBNZ r2, notZero ; If r2 is Not Zero, branch to notZero
  ; If r2 is Zero (null terminator found), fall through
  B endloop        ; Branch to endloop (acts like break)

notZero:
  ADD r0, r0, #1   ; Increment string pointer (str++)
  ADD r1, r1, #1   ; Increment length counter (len++)
  B loop           ; Go back to the start of the loop

endloop:
  ; String length result is in r1
```

This code iterates, loading each character. `CBNZ` checks if it's non-zero. If it's zero, the loop terminates via `B endloop`. Uses CBNZ for null check.

IT (If-Then) Instruction Block

Thumb-2 conditional execution block.

The `IT` instruction allows for conditional execution of one to four subsequent instructions based on a specified condition. This is particularly relevant in Thumb-2 instruction set architecture.

Syntax and Semantics

The syntax is `IT{x{y{z}}} {cond}`.

- `{cond}`: The condition applied to the first instruction following `IT`.
- `x, y, z`: Optional specifiers for the 2nd, 3rd, and 4th instructions. Each can be:
  - `T` (Then): The instruction executes if `{cond}` is true.
  - `E` (Else): The instruction executes if `{cond}` is false.

The `IT` instruction itself encodes the conditions for the following instructions.

Examples

ITTE Example

`ITTE NE` specifies an If-Then-Then-Else block based on the Not Equal condition.

```
ITTE NE        ; If NE, T, T, E. (Assembler might generate this)
  ANDNE r0,r0,r1 ; Executes if NE (First T) - 16-bit AND
  ADDNE r2,r2,#1 ; Executes if NE (Second T) - 32-bit ADD(S)
  MOVEQ r2,r3    ; Executes if EQ (E - Else condition) - 16-bit MOV
```

ITT Example

`ITT EQ` specifies an If-Then-Then block for the Equal condition.

```
ITT EQ
  MOVEQ r0,r1    ; Executes if EQ (First T)
  ADDEQ r0,r0,#1 ; Executes if EQ (Second T)
```

A branch (`BEQ dloop`) is permitted at the end of an IT block.

ITT AL Example (Implicit)

The `AL` (Always) condition can sometimes be used. An example shows 16-bit non-flag-setting instructions within the implied block (`ADDAL`, `SUBAL`), followed by a 32-bit instruction (`ADD`) outside the block.

Assembler Generation

Programmers typically do **not** need to write `IT` instructions manually. The assembler automatically generates the necessary `IT` instruction(s) when it encounters conditional instructions that require them.

Assembler handles IT generation.

Translation Example

A sequence of instructions intended to be conditional:

```
; Intended logic: if EQ T, T, E, T
  ADD r0, r0, r0  ; Inst 1 (Condition: EQ, Pattern: T)
  ADD r1, r0, r0  ; Inst 2 (Condition: EQ, Pattern: T)
  ADD r2, r0, r0  ; Inst 3 (Condition: NE, Pattern: E)
  ADD r3, r0, r0  ; Inst 4 (Condition: EQ, Pattern: T)
```

This sequence, if intended for an IT block, would require an `ITTET EQ` instruction and would translate by the assembler to:

```
ITTET EQ          ; Assembler generates this
  ADDEQ r0, r0, r0
  ADDEQ r1, r0, r0
  ADDNE r2, r0, r0
  ADDEQ r3, r0, r0
```

Subroutines, Stacks, and Calling Conventions 17

This chapter introduces subroutines (functions/procedures) in the context of ARM Cortex-M microcontrollers, covering their basic operation, the critical role of the stack, and the standard conventions for passing parameters and managing registers.

Subroutine Fundamentals

- **Definition**: A reusable block of code with a single entry point and a single exit point, designed to perform a specific task and return control to the caller. Also known as functions or procedures.
- **Calling Mechanism (RISC - ARM)**:
  - **Call**: The `BL <subroutine_label>` (Branch with Link) instruction is used. It performs two actions:
    1. Saves the return address (the address of the instruction immediately following the `BL`) into the Link Register (LR, which is R14).
    2. Sets the Program Counter (PC, R15) to the address of the `<subroutine_label>`.
  - **Return**: The `BX LR` (Branch and Exchange) instruction is typically used. It loads the value of LR into the PC, causing execution to resume at the instruction after the original `BL` call. The 'Exchange' part relates to switching instruction sets (ARM/Thumb), but Cortex-M only uses Thumb.
- **Environment Preservation**: Calls must manage parameters and preserve the caller's environment (registers the caller expects to remain unchanged).

Single entry/exit block.

LR holds return address.

BX LR to return.

Stack Implementation in Cortex-M

Memory for temporary storage.

The stack is a region of memory used for temporary storage, crucial for subroutine calls, interrupt handling, and local variable allocation.

Stack Model: Full Descending (FD)

Cortex-M typically uses a Full Descending stack.

Default: SP points to last item, grows down.

- **Full**: The Stack Pointer (SP, R13) points to the last valid data item pushed onto the stack (Top of Stack).
- **Descending**: The stack grows towards lower memory addresses. Pushing data causes SP to decrease.
- **Initialization**: The SP must be initialized to the end (highest address, the "stack base") of the allocated stack memory region before use. This is typically handled by startup code (`startup.s`). For the STM32-Discovery board mentioned, SP might start at `0x20000200`.

Stack Operations and Instructions

PUSH

- Action: Stores one or more registers onto the stack.
- SP Behavior: SP is decremented *before* each register is stored (`SP = SP - 4`), fitting the "descending" model. For multiple registers, SP is adjusted by 4 times the number of registers. The store happens at the new SP address, fitting the "full" model.
- Instruction: `PUSH {reglist}`.
- Equivalence: Functionally equivalent to `STMDB SP!, {reglist}` (Store Multiple Decrement Before, with writeback ‘¡). PUSH STMDB SP!, ...
- Register Order: The order of registers in `{reglist}` does *not* matter. The hardware always stores the registers based on their number, with the **lowest-numbered register stored at the lowest memory address** (i.e., pushed last onto a descending stack). Lowest reg no. at lowest address.

POP

- Action: Loads one or more registers from the stack.
- SP Behavior: Data is loaded from the address pointed to by SP ($Rd = (*SP)$, fitting the "full" model). Then SP is incremented *after* each register is loaded (`SP = SP + 4`, fitting the "descending" stack shrinking). For multiple registers, SP is adjusted by 4 times the number of registers.
- Instruction: `POP {reglist}`.
- Equivalence: Functionally equivalent to `LDMIA SP!, {reglist}` (Load Multiple Increment After, with writeback ‘¡). POP LDMIA SP!, ...
- Register Order: The order of registers in `{reglist}` does *not* matter. The hardware always loads the registers based on their number, with the **lowest-numbered register loaded from the lowest memory address** (i.e., popped first from a descending stack). Lowest reg no. from lowest address.

FD Stack Example Trace

Consider `PUSH {r3, r1, r7, r2}` assuming initial SP = 16, r1=0, r2=4, r3=8, r7=12.

1. SP becomes 16 - (4*4) = 0.
2. Memory writes (lowest address first, pushed last): r1 at address 0 (New SP), r2 at address 4, r3 at address 8, r7 at address 12. Final SP points to r1 at address 0.

Now, `POP {r3, r1, r7, r2}`.

1. Load r1 from address 0 (SP). SP becomes 4.
2. Load r2 from address 4 (SP). SP becomes 8.
3. Load r3 from address 8 (SP). SP becomes 12.
4. Load r7 from address 12 (SP). SP becomes 16.

The original values are restored to the registers, and SP returns to its initial value.

Alternative Stack Models

While FD is common for Cortex-M, ARM supports other models (ED, FA, EA) with different combinations of SP pointing to full/empty slots and growth direction (Ascending/Descending), along with corresponding LDM/STM addressing modes (`IA, IB, DA, DB`). The table shows the equivalent LDM/STM instructions for each stack type, such as `STMDB` for FD push and `LDMIA` for FD pop.

ARM supports 4 stack models.

ARM Procedure Call Standard (AAPCS)

Rules for function calls.

The AAPCS provides a standardized convention for subroutine calls, ensuring interoperability between code possibly generated by different compilers or written by different programmers.

Register Roles

The AAPCS defines specific roles for registers R0-R15:

- **R0-R3: Argument and Result Registers**
  - Pass first four 32-bit arguments (R0=Arg1, R1=Arg2, R2=Arg3, R3=Arg4).
  - Return results (up to 128 bits using R0-R3). R0 holds the primary (or sole) 32-bit return value. 64-bit results use R1:R0.
  - **Caller-Saved** (Scratch Registers / Not Preserved): The subroutine (callee) can modify these registers without restriction. If the caller needs the values in R0-R3 after the call, the caller must save them before the `BL` instruction.

R0-R3: Args/Return, Caller Saves.

- **R4-R11: Callee-Saved Variable Registers**
  - Used to hold local variables within a subroutine.
  - **Callee-Saved** (Must be Preserved): The subroutine *must* preserve the original values of these registers if it modifies them. This is typically done by pushing them onto the stack upon entry and popping them off before returning. The caller can assume these registers retain their values across the subroutine call. (R9 can sometimes be platform-specific/V6, caller-saved according to the table, though sometimes considered callee-saved depending on platform conventions). The table explicitly marks R4-R8, R10, R11 as 'Yes' for Subroutine Preserved.

R4-R11: Locals, Callee Saves.

- **R12 (IP): Intra-Procedure-call Scratch Register**
  - Holds intermediate values between procedures or during complex calls.
  - **Caller-Saved** (Not Preserved).

R12: Scratch, Caller Saves.

- **R13 (SP): Stack Pointer**
  - Points to the current top of the stack.

- **Callee-Saved** (Must be Preserved): Must have the same value on exit as on entry, adjusted for any arguments pushed/popped by the caller if applicable under specific ABI variants, but generally preserved by balancing PUSH/POP within the callee.

SP: Callee Saves.

- **R14 (LR): Link Register**
  - Stores the return address on a `BL` instruction.
  - **Caller-Saved** (Not Preserved): The callee modifies LR when making nested calls and must save/restore the original LR if it needs to return correctly to its own caller.

LR: Link Reg, Caller Saves (effectively).

- **R15 (PC): Program Counter**
  - Holds the address of the instruction being executed.
  - Not typically modified directly except via branching instructions (`B`, `BL`, `BX`, `POP {..., pc}`). N/A for preservation.

Register Preservation Summary

A key distinction is whether the Caller or Callee is responsible for preserving a register's value across a subroutine call:

- **Caller Saves** (R0-R3, R12, R14(LR) if nested call occurs): Caller must save if needed. Callee can overwrite freely.
- **Callee Saves** (R4-R11, R13(SP)): Callee must save/restore if used. Caller assumes they are preserved.

Subroutine Examples and Recursion

18

This chapter continues the discussion of subroutines by detailing parameter passing for different data sizes, providing practical examples including nested calls and register swapping, and exploring the concept and implementation of recursive functions.

Passing Arguments and Return Values

AAPCS argument passing rules.

The AAPCS specifies how arguments larger than 32 bits and additional arguments beyond the first four are handled.

Argument Passing Rules

- **32-bit Arguments**: R0, R1, R2, R3 for Arg1 to Arg4 respectively.
- **64-bit Arguments**: Use register pairs. Arg1 in R1(MSB32):R0(LSB32), Arg2 in R3(MSB32):R2(LSB32).
- **128-bit Arguments**: Use R3(MSB32):R2:R1:R0(LSB32).
- **Extra Arguments**: Arguments beyond those fitting in R0-R3 are pushed onto the stack by the caller *before* the `BL` instruction. The caller is responsible for removing these arguments from the stack (*stack cleanup*) after the subroutine returns.

More than 4 args use the stack.

Return Value Rules

- **32-bit Return Value**: Returned in R0.
- **64-bit Return Value**: Returned in R1(MSB32):R0(LSB32).
- **128-bit Return Value**: Returned in R3(MSB32):R2:R1:R0(LSB32).

Subroutine Examples

Example 1: Sum of Squares

Calculate $R0 = R0^2 + R1^2$ (Note: result returned in R0 per AAPCS).

```
; --- Caller ---
  MOV R0, #3      ; Pass argument x=3 in R0
  MOV R1, #4      ; Pass argument y=4 in R1
  BL SSQ          ; Call the subroutine SSQ
  MOV R2, R0      ; Retrieve the result from R0 into R2 (Example uses R2)
  B ENDL          ; Branch to end
  ; ...

; --- Callee (Subroutine) ---
SSQ PROC          ; Start of subroutine SSQ
  MUL R2, R0, R0  ; R2 = R0*R0 (Uses scratch R2)
  MUL R3, R1, R1  ; R3 = R1*R1 (Uses scratch R3)
```

```
  ADD R2, R2, R3  ; R2 = (R0*R0) + (R1*R1)
  MOV R0, R2      ; Place the final result into R0 for return
  BX LR           ; Return to caller
ENDP              ; End of subroutine
```

This example follows AAPCS by using r0, r1 for arguments, r0 for the return value, and scratch registers r2, r3 without saving/restoring them. A detailed trace shows how PC and LR change during the call (`BL SSQ` at `0x08000130` sets LR to `0x08000134`) and return (`BX LR` sets PC to `0x08000134`). Note on PC/LR LSB: Page 20 claims LR LSB is 0, page 21 claims it's 1 for Thumb mode. Cortex-M only supports Thumb, so `BL` sets LR LSB to 1, and `BX LR` expects LSB=1 to maintain Thumb state. Instructions are fetched 4 bytes at a time (two 16-bit or one 32-bit).

LR LSB=1 for Thumb state.

Example 2: Swapping Registers using Stack

Attempting to swap R1 and R2. Initial values R1=`0x11111111`, R2=`0x22222222`, SP=`0x20000200`.

- **Incorrect Attempt**: `PUSH {R1}`, `PUSH {R2}`, `POP {R1}`, `POP {R2}` does not swap the registers as intended because the individual POPs retrieve values in the reverse order they were pushed individually.
- **Correct Attempt**: `PUSH {R1, R2}`, `POP {R2, R1}` *does* perform the swap correctly.

PUSH r1,r2, POP r2,r1 swaps.

  - `PUSH {R1, R2}`: Stores R1 at lower address (`0x200001F8`), R2 at higher address (`0x200001FC`). SP points to R1's location (`0x200001F8`). Order stored: R1, then R2 based on register number.
  - `POP {R2, R1}`: Loads lowest register number (R1) from lowest address (`0x200001F8`) first. Then loads R2 from next address (`0x200001FC`). This achieves the swap.

Example 3: Nested Subroutine Call

Calculate $R0 = R0^4$ by calling a squaring function (`SQ`) twice from an intermediate function (`QUAD`).

Functions calling functions.

```
; --- Caller (MAIN) ---
MAIN PROC
  MOV R0, #2      ; Input value = 2
  BL QUAD         ; Call QUAD function (calculates R0^4)
ENDL B ENDL       ; Infinite loop after call
ENDP

; --- Intermediate Function (QUAD) ---
QUAD PROC
  PUSH {LR}       ; *** Save LR (return address to MAIN) ***
  BL SQ           ; Call SQ (R0 = R0^2)
  ; R0 now holds R0^2
  BL SQ           ; Call SQ again (R0 = (R0^2)^2 = R0^4)
  ; R0 now holds R0^4
  POP {LR}        ; *** Restore LR (return address to MAIN) ***
```

```
  BX LR           ; Return to MAIN
ENDP

; --- Squaring Function (SQ) ---
SQ PROC
  MUL R0, R0, R0  ; R0 = R0 * R0
  BX LR           ; Return (to QUAD)
ENDP
```

The critical step here is that `QUAD` must save the LR it received from `MAIN` before calling `SQ`, because the `BL SQ` instruction will overwrite LR with the return address back to `QUAD`. `PUSH {LR}` and `POP {LR}` achieve this preservation. A detailed trace shows SP, LR, PC, and register values changing through the nested calls, illustrating the stack usage for LR preservation. For example, when `MAIN` calls `QUAD` (`BL QUAD` at `0x0800013C`), LR becomes `0x08000140` (address of `B ENDL`). `QUAD` pushes this LR value. When `QUAD` calls `SQ` (`BL SQ` at `0x08000150`), LR becomes `0x08000154` (address of second `BL SQ`). `SQ` returns using this LR. After the second `BL SQ`, `QUAD` pops the original LR (`0x08000140`) and uses it to return to `MAIN`.

Must save LR for nested calls.

Recursion

A recursive function is one that calls itself to solve a problem.

Concept

- **Self-Call**: The function invokes itself, typically with a modified input that moves closer to a base case.
- **Base Case**: A condition under which the function does not call itself, stopping the recursion.
- **Divide and Conquer**: Often used for problems that can be broken down into smaller instances of the same problem (sub-problems of the same type).

Needed to stop recursion.

Classic Example: Factorial

Calculating $n! = n \times (n-1) \times \cdots \times 1$.

- **Recursive Definition**:
  - `factorial(0) = 1` (Base Case)
  - `factorial(1) = 1` (Base Case)
  - `factorial(n) = n * factorial(n-1)` for $n > 1$ (Recursive Step)
- **C Implementation**:

```
int factorial(int n) {
    int f;
    // Base case often uses n==1 or n<=1
    if (n == 1) {
        return 1;
```

```
    } else {
        // Recursive step
        f = n * factorial(n - 1);
        return f;
    }
}
```

A call trace for 'factorial(5)' shows the nested calls and return values: 5fact(4) -> 4fact(3) -> 3fact(2) -> 2fact(1) -> returns 1 -> returns 2 -> returns 6 -> returns 24 -> returns 120.

Recursion vs. Iteration

- Any recursive algorithm can be implemented iteratively (using loops).
- **Pros of Recursion**: Can lead to code that more naturally reflects the problem structure, potentially making it easier to write, program, and debug.
- **Cons of Recursion**: Generally slower due to function call overhead (saving/restoring registers, managing stack frames) and consumes more memory because each recursive call uses stack space.

Often simpler code structure.

Uses more stack memory.

Assembly Implementation of Recursion

Requires careful management of the stack to save necessary context for each recursive call.

Stack management is key.

- **Context Saving**: Before making a recursive call (`BL`), the function must push LR (its own return address) and any callee-saved registers (R4-R11) it modifies onto the stack. It also needs to preserve the current value of arguments if they are needed *after* the recursive call returns (e.g., the 'n' in 'n * factorial(n-1)').
- **Context Restoring**: After the recursive call returns, the function must pop the saved registers and LR (often popped directly into PC) to calculate its own result and return correctly.

Factorial Assembly Code

Assembly implementation for calculating factorial(n), example call with n=3.

```
; Input: R0 = n
; Output: R0 = n!
factorial PROC
  PUSH {r4, lr}   ; Save R4 (to store current n) and LR
  MOV r4, r0      ; Move n into R4 (callee-saved)

  CMP r4, #1      ; Compare n with 1 (Base case check)
  BNE NZ          ; If n != 1, branch to recursive step
  ; Base Case (n=1)
  MOVS r0, #1     ; Result is 1
  ; Use B loop_end or similar name to skip MUL and go to POP
```

```
  B loop_end      ; Branch to end to pop and return

NZ:               ; Recursive Step (n > 1)
  SUBS r0, r4, #1 ; Set argument for recursive call: r0 = n - 1
  BL factorial    ; Recursive call: factorial(n-1). Result returns in R0.
  ; After returning, R0 = (n-1)!, R4 still holds n
  MUL r0, r4, r0  ; Calculate final result: n * (n-1)!

loop_end:         ; Common exit point (label used in BNE path was 'loop')
                  ; Renaming for clarity. In doc, target of BNE is NZ,
                  ; Base case MOVS r0, #1 is followed by B loop,
                                    ; where loop is the label for POP {r4, pc}.
  POP {r4, pc}    ; Restore R4 and pop saved LR into PC to return
ENDP
```

This code shows saving R4 (to preserve 'n' across the recursive call) and LR. It checks the base case (`n == 1`). If not the base case, it calls itself with `n-1`, then multiplies the result by the saved 'n' (`r4`). The `POP {r4, pc}` restores the register and performs the return in one instruction. A memory diagram trace shows how the stack (SP), LR, and R4 change during the execution for 'factorial(3)'. The stack grows downwards as `PUSH` occurs for each call level, storing R4 and LR, and shrinks upwards as `POP` restores them on return.

POP r4, pc restores and returns.

Interrupts 19

Motivations for Interrupts

Why use interrupts?

Interrupts let embedded systems react to external events *immediately*—without burning CPU cycles in tight polling loops—and form the backbone of pre-emptive multitasking.

- **Timely event notification** – no wasted polling.
- **Efficiency** – the CPU or MCU can sleep until needed, saving power.
- **Multitasking support** – an RTOS tick, DMA completion, sensor-ready GPIO, . . .

Related concepts

- **Multitasking**: One core, many tasks by rapid switching.
- **Multithreading**: Independent execution branches *inside* a task.
- **Multi-core**: True parallelism on separate CPU cores.

Interrupts vs. Exceptions

External vs. internal

- **Interrupt**: Event *outside* the core (timer, UART, GPIO . . .).
- **Exception**: Event *inside* the core (faults, `SVC`, . . .).

ARM Cortex-M documentation places both under the umbrella term "exception".

Functions of an Interrupt System

- Detect asynchronous events.
- Finish the current instruction, then branch to the correct ISR.
- Automatically save context, run the ISR, then restore context.
- Allow per-source enable/disable and global masking.
- Resolve simultaneous requests by a priority scheme.

Basic Interrupt Flow

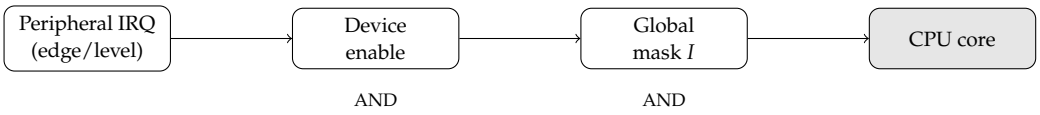
Lifecycle

1. A peripheral asserts an `IRQ` line.
2. CPU completes the current instruction → checks masks/priorities.
3. Vector fetch → program counter jumps to the matching ISR.
4. ISR handles the event.
5. `BX LR` (with `EXC_RETURN`) restores state and resumes.

General Interrupt Hardware Logic & Pending State

Enabling / masking

Interrupt delivery involves three gates: device-level enable, NVIC/global mask, and priority logic. If any gate blocks the request it becomes *pending* (`ISPR/ICPR` remember it).



Interrupt Priorities

Multiple IRQs

- **Intrinsic**: Position in the Interrupt Vector Table (IVT).
- **Configurable**: Software writes NVIC `IPR` fields.
- Higher-priority ISRs may pre-empt lower ones (*nesting*).

Interrupt Vector Table (IVT)

Map of handlers

The IVT normally starts at `0x0000 0000`:

Address	Priority	Type	Acronym	Description
0x0000 0000	–	–	–	Initial MSP (Main Stack Pointer)
0x0000 0004	−3	fixed	`Reset_Handler`	Reset entry
0x0000 0008	−2	fixed	`NMI_Handler`	Non-maskable interrupt
0x0000 000C	−1	fixed	`HardFault_Handler`	All fault escalation
0x0000 0010	cfg	settable	`MemManage_Handler`	Memory-management fault
0x0000 0014	cfg	settable	`BusFault_Handler`	Bus fault (prefetch/data)
0x0000 0018	cfg	settable	`UsageFault_Handler`	Usage fault (undef. instr., div 0, . . .)
0x0000 002C	cfg	settable	`SVC_Handler`	Supervisor call (`SVC`)
0x0000 0030	cfg	settable	`DebugMon_Handler`	Debug monitor
0x0000 0038	cfg	settable	`PendSV_Handler`	Pendable service request
0x0000 003C	cfg	settable	`SysTick_Handler`	System-tick timer

IRQ #	Address	Handler name (CMSIS)
0	0x0000 0040	`WWDG_IRQHandler`
1	0x0000 0044	`PVD_PVM_IRQHandler`
6	0x0000 0058	`EXTI0_IRQHandler`
7	0x0000 005C	`EXTI1_IRQHandler`
11	0x0000 006C	`DMA1_Channel1_IRQHandler`

Table 19.1: Example peripheral-interrupt vectors (Cortex-M)

While an ISR is running, PSR[8:0] holds ExceptionNumber = 16+IRQn.

Processor Modes & Stack Pointers

MSP vs. PSP

- **Thread Mode**: normal code; uses MSP unless CONTROL[1]=1.
- **Handler Mode**: ISR context; always privileged; always MSP.
- Separating OS (MSP) and user (PSP) stacks prevents corruption.

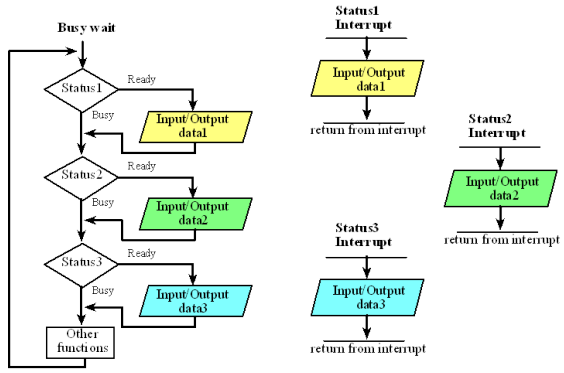
Automatic Stacking / Unstacking

On exception entry the core pushes

$$\underbrace{\text{R0 R1 R2 R3 R12 LR PC PSR}}_{\text{32 bytes}}$$

on the current stack (full-descending). A `BX LR` using an `EXC_RETURN` value reverses the process.

Interrupt Flow Visualisation



Interrupt Numbering Scheme

System vs. peripheral

Cortex-M supports up to 256 interrupt sources:

- **System exceptions** (–16 to –1)
- **Peripheral IRQs** (0 to 239)

CMSIS `IRQn` vs. PSR exception number

`PSR_Exception` = 16 + `IRQn` (e.g. `SysTick_IRQn = -1` → `PSR = 15`).

NVIC Interrupt-Control Registers

Per-IRQ control

Action	Register(s)	What to do
Enable IRQ	`ISER[n]`	Write 1 to bit n mod 32
Disable IRQ	`ICER[n]`	Write 1 to bit n mod 32
Set Pending	`ISPR[n]`	Software-pend an IRQ
Clear Pending	`ICPR[n]`	De-pend an IRQ
Read Active	`IABR[n]`	Bit 1 $\rightarrow$ ISR currently running
Trigger (SW)	`STIR`	Write IRQ number to fire

Debugger Snapshot (SVC example)

Mode: Handler (privileged)
SP: MSP=0x2000 05C8
LR: 0x`FFFFFFF9` (`EXC_RETURN`, "return to MSP")
xPSR: 0x`0100 000B` $\rightarrow$ ExceptionNumber = 11 (`SVC`)

```
SVC_Handler:
        CPSID   I               ; mask further IRQs
        PUSH    {r4-r8, lr}     ; manual save
        ; ... handler work ...
        POP     {r4-r8, lr}
        CPSIE   I
        BX      lr              ; auto-restore and resume
```

CMSIS `IRQn_Type` extracts (STM32L476)

```
/* ----- Core exceptions ------------------------------------------- */
NonMaskableInt_IRQn   = -14, /* 2  Non-maskable interrupt */
HardFault_IRQn        = -13, /* 3  Hard-fault interrupt   */
MemoryManagement_IRQn = -12, /* 4  MPU fault              */
BusFault_IRQn         = -11, /* 5  Bus fault              */
UsageFault_IRQn       = -10, /* 6  Usage fault            */
SVCall_IRQn           =  -5, /* 11 Supervisor call        */
PendSV_IRQn           =  -2, /* 14 Pendable service       */
SysTick_IRQn          =  -1, /* 15 System tick            */

/* ----- Peripheral interrupts ------------------------------------- */
WWDG_IRQn             =   0, /* Window watchdog           */
PVD_PVM_IRQn          =   1, /* PVD / PVM monitor         */
EXTI0_IRQn            =   6, /* External line 0           */
DMA1_Channel1_IRQn    =  11, /* DMA1 channel 1            */
```

20 Enabling Peripheral Interrupts

ISER / ICER

Most Cortex-M MCUs expose three 32-bit `ISER` registers for *set-enable* and matching `ICER` registers for *clear-enable*. The n-th interrupt (`IRQn`) is mapped to bit $n \bmod 32$ in word $n/32$. Writing a 1 enables (or disables) that source; writes of 0 are ignored.

- **High-level API** – `NVIC_EnableIRQ(IRQn)` / `NVIC_DisableIRQ(IRQn)` do the math for you.
- **Bare-metal** – `NVIC->ISER[IRQn»5]=1«(IRQn&0x1F);` (and analogously for `ICER`).

IRQ Register Blocks

ISPR / IABR

Besides enable bits the NVIC supplies `ISPR`/`ICPR` (software-pend/de-pend) and `IABR` (live *active* flags) so firmware or a debugger can inspect or force delivery.

Access Macros

Listing 20.1: Word/bit helpers

```
#define NVIC_REG(base,irq)  (*((volatile uint32_t*)(base)+((irq)
    >>5)))
#define NVIC_BIT(irq)       (1U << ((irq) & 0x1F))

/* Enable Timer 7 (IRQ 44) */
NVIC_REG(NVIC_ISER_BASE,44) = NVIC_BIT(44);
```

Interrupt Priority Model

Lower = stronger

Reset, NMI and HardFault have fixed priorities ($-3 \ldots -1$). All other exceptions share an eight-bit priority field split into *pre-empt* and *sub-priority* parts. Fewer pre-empt bits means finer in-group ordering, and vice-versa.

Table 20.1: Priority grouping with `NVIC_SetPriorityGrouping()`

n	Pre-empt bits	Sub bits
0	0	4
1	1	3
2 (default)	2	2
3	3	1
4	4	0

Example: with grouping 2 the call `NVIC_SetPriority(EXTI0_IRQn, 0xF)` writes `0xF0` into `IP[6]` (4 MSBs used).

Masking Critical Sections

PRIMASK / BASEPRI

- **PRIMASK=1** – blocks every IRQ except NMI.
- **FAULTMASK=1** – blocks *also* HardFault.

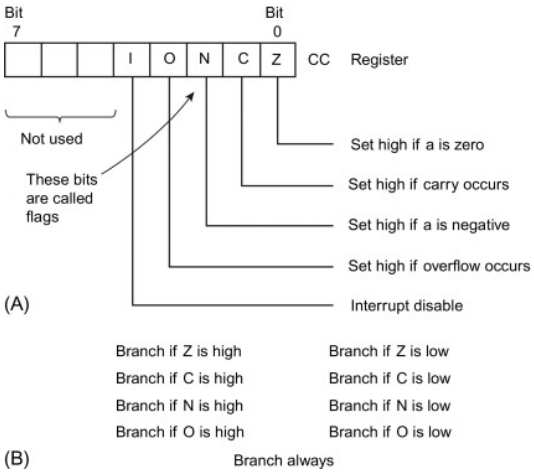
- **BASEPRI=X** – blocks all priorities numerically $\leq X$.(Remember: 0 = unmasked.)

```
MOV   R0,#(5<<4)   ; mask prio 0.5
MSR   BASEPRI,R0
; ---- timing-critical code ----
MOV   R0,#0
MSR   BASEPRI,R0
```

SysTick System Timer

24-bit down-counter

SysTick produces periodic interrupts for OS ticks, delays or general time-keeping. A 24-bit down-counter reloads the user value in `LOAD` on zero, sets `COUNTFLAG` and (if `TICKINT`=1) issues an exception.



Key registers

SysTick_CTRL	CLKSOURCE, TICKINT, ENABLE flags
SysTick_LOAD	24-bit reload value (0 → off)
SysTick_VAL	Read: current count; Write: clear+reload
SysTick_CALIB	TENMS (10 ms reference) RO

SysTick Initialisation Example

```
void SysTick_Init(uint32_t ticks) {
  SysTick->CTRL = 0;               // off
  SysTick->LOAD = ticks - 1;       // period
  NVIC_SetPriority(SysTick_IRQn,
      (1 << __NVIC_PRIO_BITS) - 1);// lowest prio
  SysTick->VAL  = 0;               // clear count
  SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
                  SysTick_CTRL_TICKINT_Msk   |
                  SysTick_CTRL_ENABLE_Msk;   // go
}
```

A companion `SysTick_Handler` usually decrements a global `volatile TimeDelay` for blocking delay loops.

Reload-Value Calculation

80 MHz → 10 ms

With an 80 MHz core clock a 10 ms tick demands

$$\text{Reload} = 80\ \text{MHz} \times 10\ \text{ms} - 1 = 800{,}000 - 1 = 799\,999.$$

Debugger Snapshot (Timer 7)

Mode: Handler
LR: 0x`FFFFFFF9`
xPSR: ExceptionNumber = 60 (`TIM7_IRQn`)

Quick-Reference Table

API	Purpose	Notes
`NVIC_EnableIRQ`	Turn on IRQ	Safe wrapper
`NVIC_SetPriority`	Set prio	4 MSBs significant
`__set_BASEPRI(X)`	Mask prio $\leq X$	0 = unmask
`SysTick_LOAD`	Period	24 b max

Takeaways

- Three NVIC layers—enable, pending, active—give full visibility over each source.
- Priority grouping splits 8 bits into pre-empt versus tiebreak sub-priority.
- PRIMASK, FAULTMASK and BASEPRI implement coarse to fine critical-section control.
- SysTick is a self-contained 24-bit timer ideal for RTOS heartbeats and busy-wait delays.

GPIO Overview

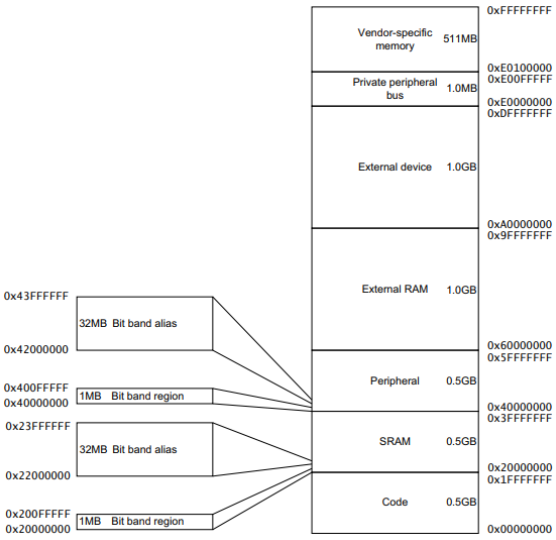
Zero-wait pin access

With memory-mapped I/O every peripheral register lives in the same 32-bit address space as code and data, so ordinary `LDR/STR` instructions toggle pins in a single cycle — no special `IN/OUT` op-codes.

- **Uniform API** – pointers, structs, and bit-fields work.
- **Speed** – reads/writes go over the fast AHB/APB fabric.
- **Simplicity** – C code looks like plain memory access.

Cortex-M4 Address Map

Top-level regions



Flash starts at `0x00000000`, SRAM at `0x20000000`, peripherals at `0x40000000`, and NVIC/SysTick at `0xE0000000`.

STM32L4 GPIO Windows

Port blocks

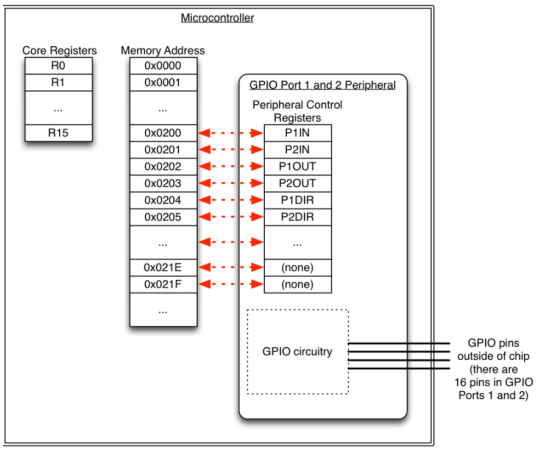
Table 20.2: Base addresses (1 KB each)

Port	Base
A	0x48000000
B	0x48000400
C	0x48000800
D	0x48000C00
E	0x48001000
F–H	...

Each block exposes 12 registers (48 bytes).

GPIO Register Layout

Core set



`MODER`, `OTYPER`, `OSPEEDR`, `PUPDR`, `IDR`, `ODR`, `BSRR`, `LCKR`, `AFR[2]`, `BRR`, `ASCR`.

ODR – Output Data Register

Write to drive pins

Address `0x48000014`. Each bit reflects the logic level on the matching GPIO pin when the port is configured as output.

```
*((uint32_t *)0x48000014) |= 1UL << 14; /* set PB14 high */
```

Listing 20.2: Set PB14 high directly

Typed Access in C

Listing 20.3: Minimal GPIO typedef

```
typedef struct {
  volatile uint32_t MODER;
  volatile uint32_t OTYPER;
  volatile uint32_t OSPEEDR;
  volatile uint32_t PUPDR;
  volatile uint32_t IDR;
  volatile uint32_t ODR;
  volatile uint32_t BSRR;
  volatile uint32_t LCKR;
  volatile uint32_t AFR[2];
  volatile uint32_t BRR;
  volatile uint32_t ASCR;
} GPIO_TypeDef;

#define GPIOA ((GPIO_TypeDef *)0x48000000)

GPIOA->ODR |= 1UL << 14; /* toggle PA14 */
```

Clock Enable

RCC_AHB2ENR

Before any pin manipulation you must gate the port clock:

```
RCC->AHB2ENR |= (1U << 1); /* GPIOB clock on */
```

Bit 1 is `GPIOBEN`.

Pin Direction Setup

MODER

Listing 20.4: Configure Pin 2 output on Port B

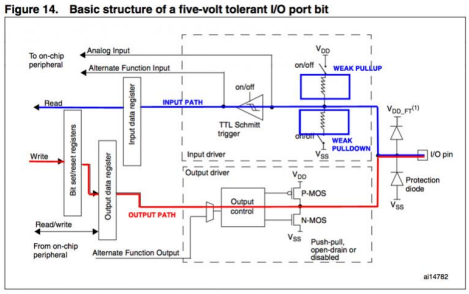
```
/* Pin 2 output on port B */
GPIOB->MODER &= ~(3UL << 4); /* clear bits 5:4 */
GPIOB->MODER |= (1UL << 4);  /* set 01 = output */
```

Each pin consumes two bits: 00 input, 01 output, 10 alternate, 11 analog.

I/O Pad Anatomy

Push-pull vs open-drain

Figure 14. Basic structure of a five-volt tolerant I/O port bit



1. V_{DD_FT} is a potential specific to five-volt tolerant I/Os and different from V_{DD}.

Key controls: push-pull/open-drain (`OTYPER`), drive strength (`OSPEEDR`), pull-ups/downs (`PUPDR`).

Example – Red LED on PB2

Discovery board

1. Enable GPIOB clock.
2. Set PB2 to output push-pull.
3. Write 1 to bit 2 of `ODR`. High = LED on, Low = off.

`GPIOB->ODR ^= 1UL << 2;` toggles the LED.

Key Takeaways

- GPIO ports are ordinary memory; bit-twiddling uses standard C.
- Clock gating via `RCC_AHB2ENR` is mandatory.
- `MODER`, `OTYPER`, and `PUPDR` fully define pin behaviour.
- `BSRR` sets/resets outputs atomically — handy in ISRs.

GPIO Output Type Register (`OTYPER`)

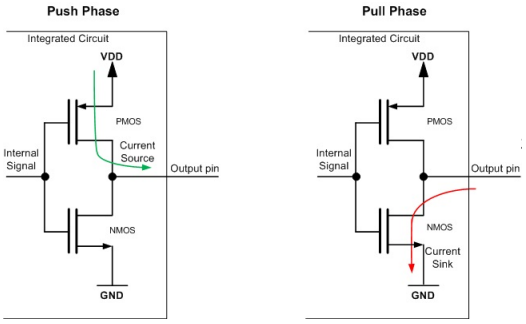
Push-pull or open-drain

`OTYPER` has one bit per pin (bits 0-15); 0 selects push-pull, 1 selects open-drain.

Listing 20.5: Set PB2 to push-pull

```
1 /* PB2 push-pull */
2 GPIOB->OTYPER &= ~(1UL << 2);
```

Push-pull vs. Open-drain



- **Push-pull** actively drives high or low.
- **Open-drain** only sinks; logic 1 = Hi-Z.

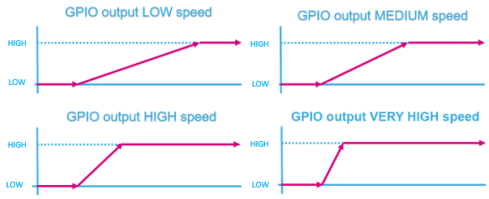
Bit value	Push-pull	Open-drain
1	High	Hi-Z
0	Low	Low

Table 20.3: Output truth table

GPIO Output Speed (`OSPEEDR`)

Rise/fall slew

Four settings: low, medium, fast, very high. Choose the lowest speed that meets timing.



Slew Rate Formula

$$\text{SlewRate}_{\max} = \frac{\Delta V}{\Delta t}$$

GPIO Output Data Register (`ODR`)

Address offset 0x14; one bit per pin.

```
GPIOB->ODR |= 1UL << 2;   /* set PB2 high */
```

Listing 20.6: Set PB2 high

Red LED Example (PB2)

1. Enable GPIOB clock via `RCC->AHB2ENR`.
2. Configure PB2 as output push-pull.
3. Write 1 to bit 2 of `ODR` to light the LED.

`GPIOB->ODR ^= 1UL << 2;` toggles the LED.

GPIO Initialisation Steps

- Enable peripheral clock.
- Configure `MODER`, `OTYPER`, `OSPEEDR`, `PUPDR` as needed.
- Use `BSRR` for atomic set/reset inside ISRs.

Minimal GPIO TypeDef

```c
typedef struct {
  volatile uint32_t MODER;
  volatile uint32_t OTYPER;
  volatile uint32_t OSPEEDR;
  volatile uint32_t PUPDR;
  volatile uint32_t IDR;
  volatile uint32_t ODR;
  volatile uint32_t BSRR;
  volatile uint32_t LCKR;
  volatile uint32_t AFR[2];
} GPIO_TypeDef;

#define GPIOB ((GPIO_TypeDef *)0x48000400)
```

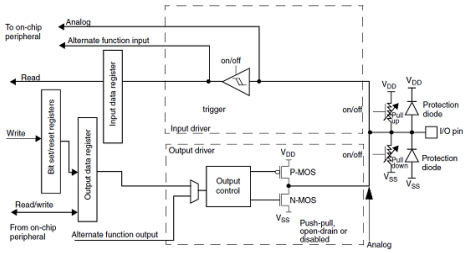
Listing 20.7: Minimal GPIO Typedef for GPIOB

Assembly Pin Toggle

```asm
GPIOA_BASE EQU 0x48000000
GPIO_ODR EQU 0x14

LDR r7, =GPIOA_BASE
LDR r1, [r7, #GPIO_ODR]
ORR r1, r1, #(1 << 14)
STR r1, [r7, #GPIO_ODR]
```

GPIO Input Path



Set pin to input (`MODER = 00`) and choose pull-up (01) or pull-down (10) in `PUPDR`.

Pull-up / Pull-down Logic

Floating inputs read undefined; pulls give a defined idle level.

Joystick Example

Discovery-board joystick on PA0-PA4 uses input mode with pull-ups;
pressing shorts the line to ground.

Key Points

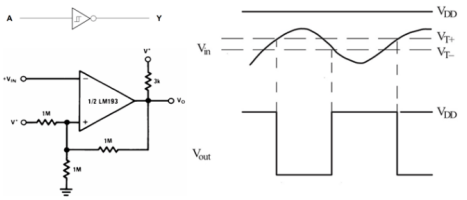
- `OTYPER` selects push-pull or open-drain per pin.
- Output speed (slew) trades EMI for timing head-room.
- Use `ODR` for simple writes, `BSRR` for atomic writes.
- Pull-ups / pull-downs stabilize high-impedance inputs.

GPIO Examples 21

Schmitt-Trigger Inputs

Noise immunity

Digital inputs on STM32L4 include an on-die Schmitt trigger: slow, noisy analog edges are squared up before the logic block. Two distinct thresholds (T_H, T_L) provide hysteresis.



Clock Enable

RCC_AHB2ENR

`RCC->AHB2ENR` = RCC_AHB2ENR_GPIOAEN;|

GPIO Input Recipe

1. Clear the two `MODER` bits: input mode (00).
2. Choose pulls in `PUPDR` if needed (00 none, 01 PU, 10 PD).
3. Read pin state in `IDR`.

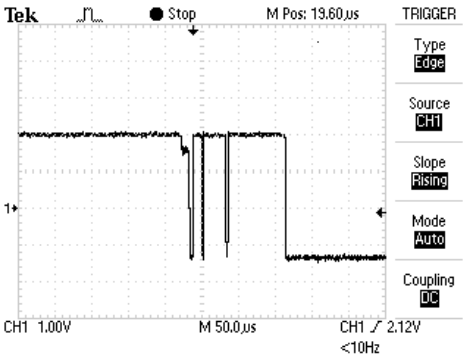
Listing 21.1: Read PA0 joystick press

```
uint32_t pressed;
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN; /* clock */
GPIOA->MODER &= ~3UL;                /* PA0 input */
GPIOA->PUPDR &= ~3UL;                /* no pulls */
pressed = (GPIOA->IDR & 1UL) != 0;
```

Button Debouncing

Glitches

A mechanical push-button bounces for ~5-20 ms. Hardware RC or firmware state-machine filters spikes.



Motor Types

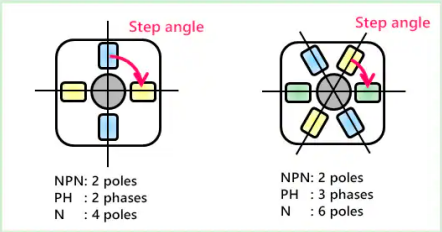
- **Servo** – closed loop, high torque, feedback.
- **Stepper** – open loop, holds discrete steps, cheap.

Stepper Motor Basics

Step angle

$$\theta_{\text{step}} = \frac{360^\circ}{\text{steps per rev}} = \frac{360^\circ}{P \times T}$$

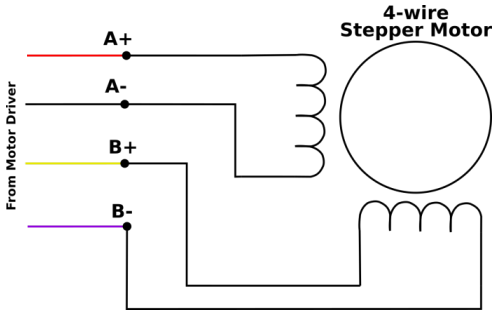
P = stator phases, T = rotor pole pairs.



Stepping Modes

Wave energize one coil at a time – lowest torque.
Full energize two coils – full torque, 200 steps/rev typical.
Half alternate 1/2 coils – doubles step count, smoother.

Wave-Step Sequence (4-wire Bipolar)



Full-Step Control Code

Listing 21.2: Clockwise full steps

```
static const uint8_t FS[4] = {0x48, 0x88, 0x84, 0x44};

void FullStepR(int n){
 for(int rev=0; rev<n; ++rev){
   for(int i=0; i<4; ++i){
     GPIOB->ODR &= ~(0x00CC); /* clear */
     GPIOB->ODR |=  FS[i];    /* set   */
     delay_us(1800);
}}}
```

Half-Step Control Code

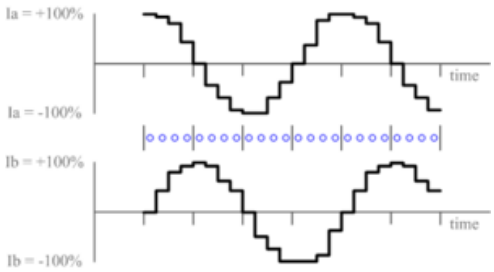
```
static const uint8_t HS[8] =
 {0x84,0x04,0x44,0x40,0x48,0x08,0x88,0x80};

void HalfStepL(int n){
 for(int rev=0; rev<n; ++rev){
   for(int i=7; i>=0; --i){
     GPIOB->ODR &= ~(0x00CC);
     GPIOB->ODR |=  HS[i];
     delay_us(900);
}}}
```

Listing 21.3: Counter-clockwise half steps

Micro-Stepping Idea

Apply sine current to phase A, cosine to phase B: smoother torque and finer positioning (up to 256 μsteps).



Sine-cosine

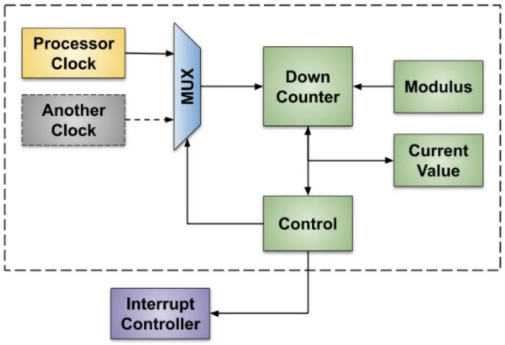
Key Points

- Enable GPIO clock before configuring ports.
- Use input mode plus pulls for buttons / keypads.
- Debounce either in hardware or with a timer.
- Steppers: choose wave, full, half, or micro-stepping per need.
- Micro-stepping drives coils with sine-cosine DAC/PWM for quiet motion.

Timer Overview

- Free-running 16-bit or 32-bit counter, clocked independently of the CPU.
- Core functions: input-capture, output-compare, PWM generation, one-pulse mode.

Clock Chain



Prescaler (PSC) divides the bus clock to f_{CNT}. Counter counts up/down between 0 and ARR. When the counter matches $\mathtt{CCR}_x$ the compare circuitry sets OCREF, producing a waveform or interrupt.

Key Registers

PSC	Prescaler (16-bit) – reloads on update event
ARR	Auto-reload value – timer period
CCR1-4	Compare/Capture register per channel
RCR	Repetition counter, adds extra cycles before update

Output-Compare Modes

1. 000 Frozen (no change)
2. 001 Set high when CNT == CCR
3. 010 Set low when CNT == CCR
4. 011 Toggle on match
5. 100 Force low, 101 Force high.

Counting Modes

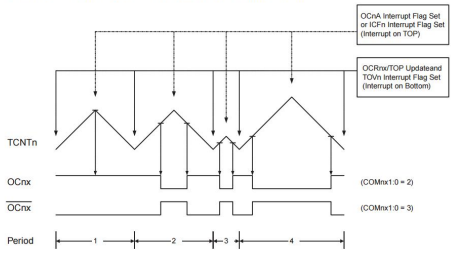
Edge-aligned Up-count or down-count; update on overflow/underflow.
Center-aligned Up then down; period = $2 \times \mathtt{ARR} \times T_{\mathrm{CLK}}$.

PWM Basics

$$\text{Period} = (\mathtt{ARR}+1)\,T_{\mathrm{CLK}}, \qquad \text{Duty} = \frac{\mathtt{CCR}}{\mathtt{ARR}+1}$$

- **Mode 1** (low-true): output high while CNT < CCR.
- **Mode 2** (high-true): output low while CNT < CCR.

Figure 16-9. Phase and Frequency Correct PWM Mode, Timing Diagram



Left / Right / Center Alignment

- Left-aligned: rising edges align at CNT = 0.
- Right-aligned: falling edges align at CNT = ARR.
- Center-aligned: symmetric about mid-point; reduced harmonics.

ARR Preload and RCR

Setting ARPE=1 stores writes in a shadow register, then transfers to ARR on the next update event for glitch-free period changes. RCR inserts (`RCR` + 1) repetitions before each update, useful for burst PWM.

Output Polarity & Enables

- CCxE / CCxNE bits enable main or complementary output.
- CCxP selects active-high or active-low polarity.
- MOE, OSSI, OSSR control global output enable and safe states.

Input-Capture

- Detects edge time-stamps on a pin; choose rising, falling, or both.
- Optional digital filter removes spurious glitches.

Pulse-Width ISR Skeleton

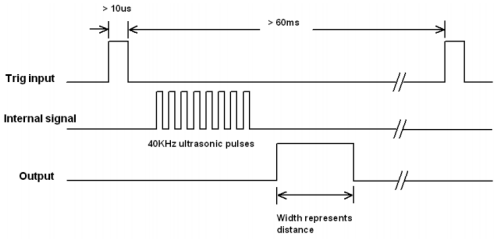
```
volatile uint32_t pulse, last;
volatile uint8_t  level;

void TIM4_IRQHandler(void){
  uint32_t now;
  if (TIM4->SR & TIM_SR_CC1IF){      /* capture */
    now     = TIM4->CCR1;
    level = 1 - level;               /* toggle */
    if (level == 0)                  /* falling edge */
        pulse = now - last;          /* width */
    last = now;
  }
  if (TIM4->SR & TIM_SR_UIF)         /* overflow */
      TIM4->SR &= ~TIM_SR_UIF;
}
```

Ultrasonic Distance Example

Round-trip distance:

$$d = \frac{tv}{2} \quad \text{with } v \approx 343\,\text{m/s}$$



Practical conversion: $d\,[\text{cm}] = t\,[\mu\text{s}]/58.82$.

Key Takeaways

- PSC and ARR set base frequency; CCR sets duty or compare point.
- Edge-aligned PWM aligns rising or falling edges; center-aligned halves EMI.
- ARPE prevents glitches when updating ARR on-the-fly.
- Input-capture plus overflow logic measures pulse width and frequency.

Metrics and Acknowledgments

- **Pages** - 104
- **Words** - 12805
- **Average words/page** - 123
- **Figures / Tables** - 46/41
- **LaTeX source lines** - 5262

Acknowledgments

- Prof. Chang — course instruction.
- Prof. Bell — lab organization and support.
- Overleaf — online LaTeX platform used for note-taking and LLM-assisted shorthand expansion (TikZ diagrams, etc.).
- Various online sources for embedded-systems imagery; diagrams otherwise authored by the note-taker.