ARM Assembly Simulator

Maria - Alexia Crișan

3rd of November, 2025

Table of contents:

Project Overview

The **ARM Assembly Simulator** is a Python-based tool designed to emulate a simplified ARM-like architecture. It allows users to write assembly programs, assemble them into machine code, and simulate their execution step-by-step, closely mimicking the behavior of a real ARM processor. The project provides both an assembler and a CPU simulator, giving users full control over the process of encoding, decoding, and executing instructions.

The architecture is modular, with separate components for the assembler, instruction encoders, decoders, memory handling, and the CPU core. This modular design allows easy extension of the instruction set and supports the addition of pseudo-instructions and higher-level abstractions without affecting the base logic.

The simulator currently supports the following features:

- **Core instructions:** Arithmetic and logical operations such as MOV, ADD, SUB, CMP, AND, ORR, EOR, and MVN.

- **Stack operations:** PUSH and POP for saving and restoring register values using the stack pointer.

- **Data transfer instructions:** Memory access operations like LDR (load register) and STR (store register), enabling data movement between registers and memory

- **Branching and control flow**: B, BL, JMS, and RET for implementing loops, function calls, and returns.

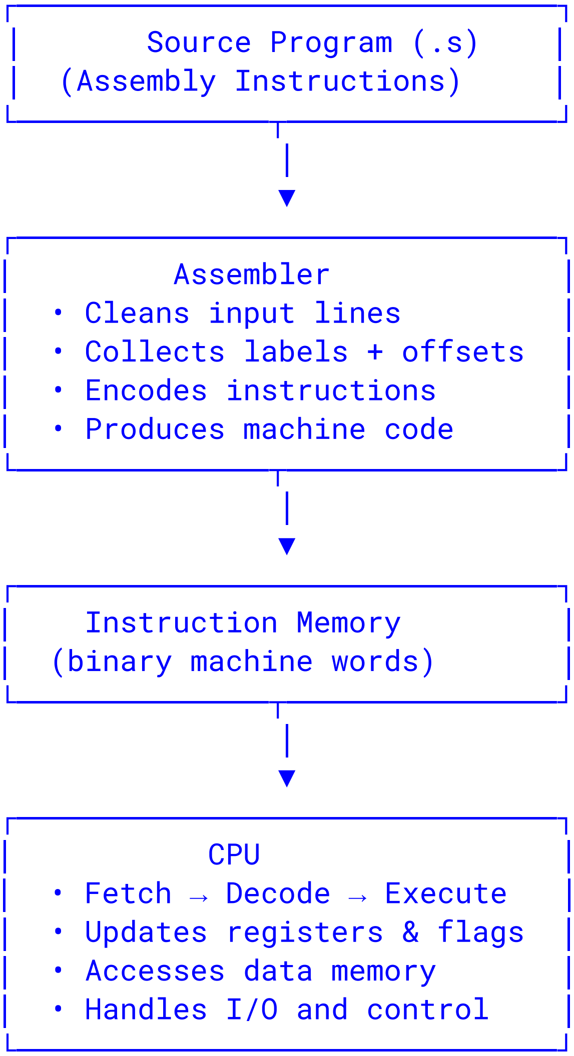
- **System instructions:** HLT (halt), INP (input), and OUT (output) for user interaction and execution control.

- **Pseudo-instructions:** Simplified or derived operations such as INC (increment), DEC (decrement), CLR (clear register), LSL (logical shift left), LSR (logical shift right), and MOD (modulo operation), implemented using combinations of base instructions.

- Visualization and debugging: Real-time visualization of CPU registers, memory state, and flags, enabling users to track execution flow and debug programs more easily.

System Architecture

The simulator follows a clear pipeline model inspired by real CPU design.
Programs are written in assembly, translated into binary machine code, and executed inside a simulated processor.



Source Program (.s)
(Assembly Instructions)

Assembler
- Cleans input lines
- Collects labels + offsets
- Encodes instructions
- Produces machine code

Instruction Memory
(binary machine words)

CPU
- Fetch → Decode → Execute
- Updates registers & flags
- Accesses data memory
- Handles I/O and control

The ARM Assembly Simulator operates through a well-defined pipeline that begins with reading the source assembly file and ends with executing its instructions inside the simulated CPU.

First, the assembler reads the .s source file and removes comments (lines starting with ; or //) and blank lines to ensure only meaningful instructions remain.

Next, the assembler performs label collection by scanning each line and storing every label (a line ending with a colon) along with the corresponding program counter address. This allows branch instructions such as B loop to later resolve their jump targets correctly.

Once the labels are collected, each instruction line is passed to the encoder, which determines the instruction type, whether data-processing, branch, system, or stack, and calls the corresponding encoding function. The encoder then returns a 32-bit integer machine instruction such as MOV R0, #1 -> 0xE3A00001.

After encoding all lines, the assembler writes the output in two formats: a binary file (program_out/program.bin) containing raw bytes and a text file (program_out/program_bits.txt) containing human-readable binary representations. These files represent the program to be executed by the simulator. The binary program is then loaded into instruction memory, where each instruction occupies four bytes and is stored sequentially starting from address zero. When the CPU is initialized with the instruction and data memory, it begins execution by performing a continuous **fetch - decode - execute** cycle until a halt instruction (HLT) is encountered.

During the fetch stage, the CPU reads the next 32-bit instruction from memory using the address in the program counter (register R15).

In the decode stage, the CPU determines the instruction category by checking specific bits and routing it to the correct handler - for example, is_system_instruction() for HLT, INP, or OUT; is_branch_instruction() for B, BL, or RET; or is_data_processing_instruction() for MOV, ADD, and similar operations.

In the execute stage, the instruction is carried out: registers, flags, and memory are updated depending on the instruction semantics. For instance, ADD R1, R0, #1 computes R1 = R0 + 1, MOV R2, #5 sets register 2 to 5, B loop updates the program counter to a label address, and HLT stops execution.

Throughout execution, registers R0 to R15 hold data and control values (with R13 as the stack pointer, R14 as the link register, and R15 as the program counter).

The CPU maintains and updates condition flags (N, Z, C, and V) to represent negative, zero, carry, and overflow states.

The system instructions manage I/O operations directly: INP R0 prompts the user for input, OUT {R0, R1} prints selected register values, and HLT cleanly halts execution.

When the CPU encounters a halt instruction (opcode 0xF0000000), it prints a message confirming the stop and ends the simulation.

The Assembler

The assembler is the first major stage of the ARM Assemby Simulator. Its primary role is to transform human-readable assembly source code into a sequence of binary machine instructions that can be loaded into the simulator's instruction memory. This process mirrors the behavior of a real ARM assembler, where symbolic instructions, labels, and constants are converted into fixed-length binary encodings that the CPU can interpret directly.

The assembler begins by reading the source program from a .s file using the read_source() function. This step loads all lines of text from the file into memory without altering their content. The next stage, handled by clean_lines(), sanitizes the input by removing empty lines, trimming whitespace, and discarding any comments denoted by ; or //. This ensures that only valid instructions and label definitions remain.

Once the source is cleaned, the assembler performs label collection, which is a crucial phase in linking symbolic addresses to actual memory positions. Each label, identified as a line ending with a colon (:), is stored in a dictionary along with its corresponding program counter (PC) value. Since every instruction in this simulator occupies four bytes, the assembler increments the PC by four for each instruction encountered. This label-address mapping allows branch and jump instructions to later calculate their correct destination offsets during encoding.

After labels are resolved, the assembler iterates over each instruction line and calls encode_instruction(), a central function that determines the instruction type (e.g., data-processing, branch, stack, system, or pseudo) and delegates encoding to the appropriate module. The encoder translates each instruction into a 32-bit integer representing the binary machine code equivalent. If a pseudo-instruction expands into multiple actual instructions (for example, LSL or MOD), the assembler correctly handles this by extending the resulting list of binary codes and adjusting the program counter accordingly.

The final output of the assembler is a list of 32-bit integers representing the complete program in machine code form. These codes are later written to output files in binary and human-readable formats and then loaded into the CPU's instruction memory for execution. In essence, the assembler acts as the translator between readable assembly language and the machine-level representation that drives the CPU simulation, serving as the critical link between program design and hardware execution.

The Encoder

The encoder is responsible for translating each assembly instruction into its corresponding 32-bit binary representation, known as the machine instruction. In this simulator, each instruction follows a fixed-size 32-bit encoding format inspired by the ARM instruction set architecture. The encoder determines the instruction type, extracts its operands, and assembles the appropriate bit fields according to predefined encoding rules.

All encoding functions reside inside the encoder/ directory, each handling a different instruction category. The main entry point is encode_instruction(), which receives the raw assembly line, the current program counter (for resolving branch offsets), and the label dictionary. It parses the instruction name, determines which encoder module should process it, and finally returns a 32-bit integer or a list of integers (in the case of pseudo-instructions that expand into multiple real instructions).

Immediate Value Encoding and Handling

In ARM architecture, immediate values (constants prefixed with `#`) are not stored directly in the instruction as plain 32-bit numbers. Instead, ARM uses a **rotated 8-bit immediate encoding scheme**, allowing a wide range of constants to be represented compactly within a 12-bit field called **operand2**.
This simulator implements a simplified version of this same mechanism using the helper function encode_immediate_value() from helpers.py.

Each data-processing instruction has only 12 bits reserved for its second operand.
This space must encode both the **value** and an optional **rotation**, allowing more numbers to fit into a small bit field.

For example, in real ARM processors, the lower 8 bits hold the immediate value, while the upper 4 bits specify how much to rotate it right (in even steps). This means a single 8-bit constant can represent many 32-bit values by applying a rotation.

When the encoder detects an immediate operand, it:

1. Strips the # prefix and converts the value to an integer.
2. Checks if the number fits in 8 bits (0–255).
3. If not, applies rotation logic to pack it efficiently.
4. Returns a 12-bit encoded value that fits into the **operand2** field.

If a number is too large to fit in 8 bits, the encoder attempts to represent it using **bit rotation**. For instance, encoding #1024 (0x400) as a shifted version of 0x40.

This ensures the assembler can handle a wide variety of constants while still maintaining a fixed 32-bit instruction length.

Data Processing Instruction Encoder

The **Data Processing Encoder** is responsible for converting arithmetic and logical instructions such as `ADD`, `SUB`, `MOV`, `CMP`, `AND`, `ORR`, `EOR`, and `MVN` into their 32-bit ARM-like binary representations. These are the most frequently used instruction types in the simulator, handling register manipulation, arithmetic, bitwise operations, and comparisons.

General Format

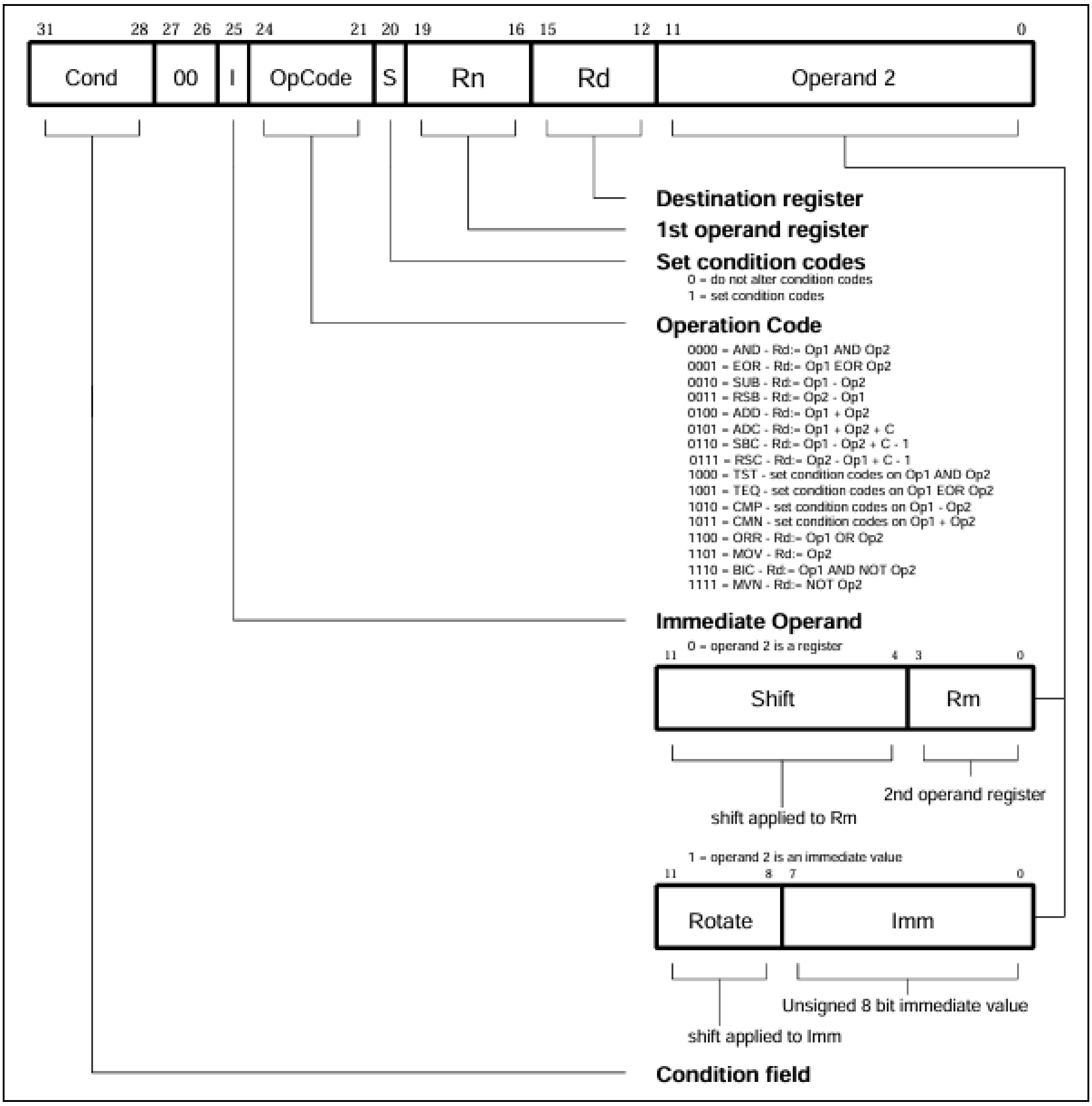
Each data-processing instruction follows the base format:

```
cond | 00 | I | opcode | S | Rn | Rd | operand2
```

- cond — 4 bits: condition code (e.g., 1110 = always execute)
- 00 — identifies this as a data-processing instruction
- I — immediate flag (1 if using an immediate constant, 0 if using a register)
- opcode — 4 bits defining the operation (ADD, SUB, MOV, etc.)

- S — flag update bit (1 if the instruction updates condition flags)
- Rn — first operand register (source)
- Rd — destination register (where result is stored)
- operand2 — second operand (either register or immediate value)

The final 32-bit instruction is constructed by combining these fields using bitwise shifts and OR operations.


31
28
27
26
25
24
21
20
19
16
15
12
11
0
Cond
00
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not alter condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2
Immediate Operand
0 = operand 2 is a register
11
4
3
0
Shift
Rm
shift applied to Rm
2nd operand register
1 = operand 2 is an immediate value
11
8
7
0
Rotate
Imm
Unsigned 8 bit immediate value
shift applied to Imm
Condition field

Supported Instruction Formats

The encoder supports all standard ARM-like data-processing instruction patterns, including both **register** and **immediate** variants.

MOV (Move)

Transfers a value into a register.

```
MOV Rd, #imm          -> Rd = imm
MOV Rd, Rn            -> Rd = Rn
```

Example:
`MOV R0, #5` -> `0xE3A00005`

- `I = 1` if the source is an immediate value (prefixed with `#`)
- `Rn` is set to 0 for MOV since it uses only one operand

ADD (Addition)

Performs register or immediate addition.

```
ADD Rd, Rn, Rm        -> Rd = Rn + Rm
ADD Rd, Rn, #imm      -> Rd = Rn + imm
ADD Rd, Rm/#imm       -> Rd += Rm or Rd += imm
```

The encoder determines if the instruction is two-operand (`ADD R0, R1`) or three-operand (`ADD R0, R1, R2`).
When only two operands are given, the first is treated as both destination and source (e.g., `ADD R0, R1` means `R0 = R0 + R1`).

Example encodings:

- `ADD R1, R0, R2` -> `E0801002`
- `ADD R5, R5, #1` -> `E2855001`

SUB (Subtraction)

Performs subtraction between registers or immediates.

```
SUB Rd, Rn, Rm        -> Rd = Rn - Rm
SUB Rd, Rn, #imm      -> Rd = Rn - imm
SUB Rd, Rm/#imm       -> Rd -= Rm or Rd -= imm
```

Similar operand rules to ADD.

CMP (Compare)

Performs a subtraction without storing the result - only updates flags (Z, N, V).

```
CMP Rn, Rm          -> sets flags based on (Rn - Rm)
CMP Rn, #imm        -> sets flags based on (Rn - imm)
```

For `CMP`, the **S bit** is always set (`S = 1`), enabling flag updates.

Logical Operations

The encoder also supports bitwise operations:

```
AND Rd, Rn, Rm      -> Rd = Rn & Rm
ORR Rd, Rn, Rm      -> Rd = Rn | Rm
EOR Rd, Rn, Rm      -> Rd = Rn ^ Rm
MVN Rd, Rm          -> Rd = NOT Rm
```

These follow the same two or three operand pattern as arithmetic operations. Immediate variants are also supported.

Example: Step-by-Step Encoding

Instruction:
ADD R2, R0, #5

Encoding Steps:

1. Opcode for ADD = 0100
2. I = 1 because it uses an immediate value
3. Rn = 0, Rd = 2
4. Immediate value #5 → encoded to 0x5
5. Combine all fields:

1110 | 00 | 1 | 0100 | 0 | 0000 | 0010 | 000000000101
= 0xE2802005

Result: **E2802005**

Summary

The Data Processing Encoder is the most versatile and frequently used part of the assembler. It supports flexible operand formats, automatically detects immediate vs. register sources, updates flags for comparison operations, and strictly follows ARM's binary layout conventions. This module provides the foundation for arithmetic, logical, and data movement instructions across the entire simulator.

Branch Instruction Encoder

The **Branch Encoder** handles program control instructions - instructions that alter the flow of execution by jumping to another part of the program.
This category includes **B**, **BL**, **JMS**, and **RET**, which correspond to **branch**, **branch with link (subroutine call)**, and **return** operations respectively.

These instructions form the basis of loops, conditional statements, and subroutine calls within the simulated CPU.

General Format

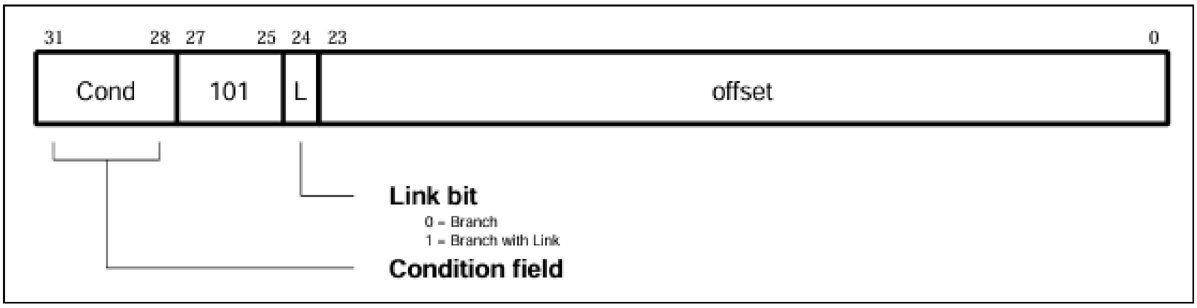
Each branch instruction is encoded according to the following ARM-like structure:

```
cond | 101 | L | offset24
```

- **cond** — 4 bits: condition code
- **101** — identifies the instruction as a branch operation
- **L** — link bit (1 = save return address in LR for subroutines, 0 = simple branch)
- **offset24** — 24-bit signed offset to the branch target (word-aligned)

The **offset** specifies the relative distance (in words) from the current program counter (PC) to the target label. Since instructions are 4 bytes each, the offset is shifted right by 2 bits (>> 2) when encoded, and later shifted back during execution.

The final machine instruction is built by combining all these fields using bitwise shifts and logical OR operations.



Supported Instruction Formats

The **Branch Encoder** supports both unconditional and conditional branch instructions, as well as subroutine calls (`BL`, `JMS`) and returns (`RET`).

Each branch instruction type uses a 24-bit signed offset to jump to a label relative to the current program counter (PC).

B (Unconditional Branch)

Performs an unconditional jump to the specified label.
It updates the **program counter (PC)** to the target address computed as:

```
target = current_PC + 8 + (offset24 << 2)
```

Syntax:

```
B label
```

Example

```
B loop
```

If loop is located 12 bytes ahead:

```
offset = (target - (current + 8)) >> 2 = (12 - 8) >> 2 = 1
```

1110 | 101 | 0 | 000000000000000000000001
-> 0xEA000001

Conditional Branches

Conditional branches execute only if specific CPU flags (**N, Z, V, C**) satisfy the condition associated with the branch.
These are encoded by modifying the **cond** field (bits [31:28]) according to the following table:

Instruction	Condition	cond bits	Meaning
BEQ	EQ	0000	Branch if equal (Z == 1)
BNE	NE	0001	Branch if not equal (Z == 0)
BLT	LT	1011	Branch if less than (N != V)
BGT	GT	1100	Branch if greater than (Z == 0 and N == V)
BGE	GE	1010	Branch if greater or equal (N == V)
BLE	LE	1101	Branch if less or equal (Z == 1 or N != V)

Syntax:

```
BEQ label
BNE label
BLT label
BGT label
BGE label
BLE label
```

Each conditional instruction is encoded exactly like `B label`, except the **cond** bits are replaced using the `BRANCH_COND_MAP` lookup.

BL / JMS (Branch with Link / Jump to Subroutine)

Branches to a label and stores the **return address (PC + 4)** in **LR (R14)**.
Used for subroutine calls or function-like behavior.

Syntax:

```
BL/JMS label
```

Example:

```
BL subroutine
```

At runtime, `LR <- PC + 4` is executed automatically before jumping.

RET (Return from Subroutine)

Returns execution to the address stored in the **link register (LR, R14)**.
This is encoded as a **MOV** instruction from **LR (R14)** to **PC (R15)**, restoring the previous program counter.

Syntax:

```
RET
```

Equivalent ARM instruction:

```
MOV PC, LR
```

Encoding:

This instruction completes the function call cycle initiated by `BL` or `JMS`.

Summary

The Branch Encoder is responsible for all control flow in the simulator - from simple jumps to complex subroutine calls.
It translates symbolic labels into 24-bit signed offsets, manages return-linking through the **L bit**, and handles the RET instruction as a MOV from LR to PC.
By combining these elements, the encoder enables structured control flow, loops, and modular subroutine execution in the ARM Assembly Simulator.

Data Transfer Instruction Encoder

The **Data Transfer Encoder** handles **single-word load and store operations** - specifically `LDR` (Load Register) and `STR` (Store Register).
These instructions move data between the CPU registers and memory and form the backbone of memory access in the ARM-like simulator.

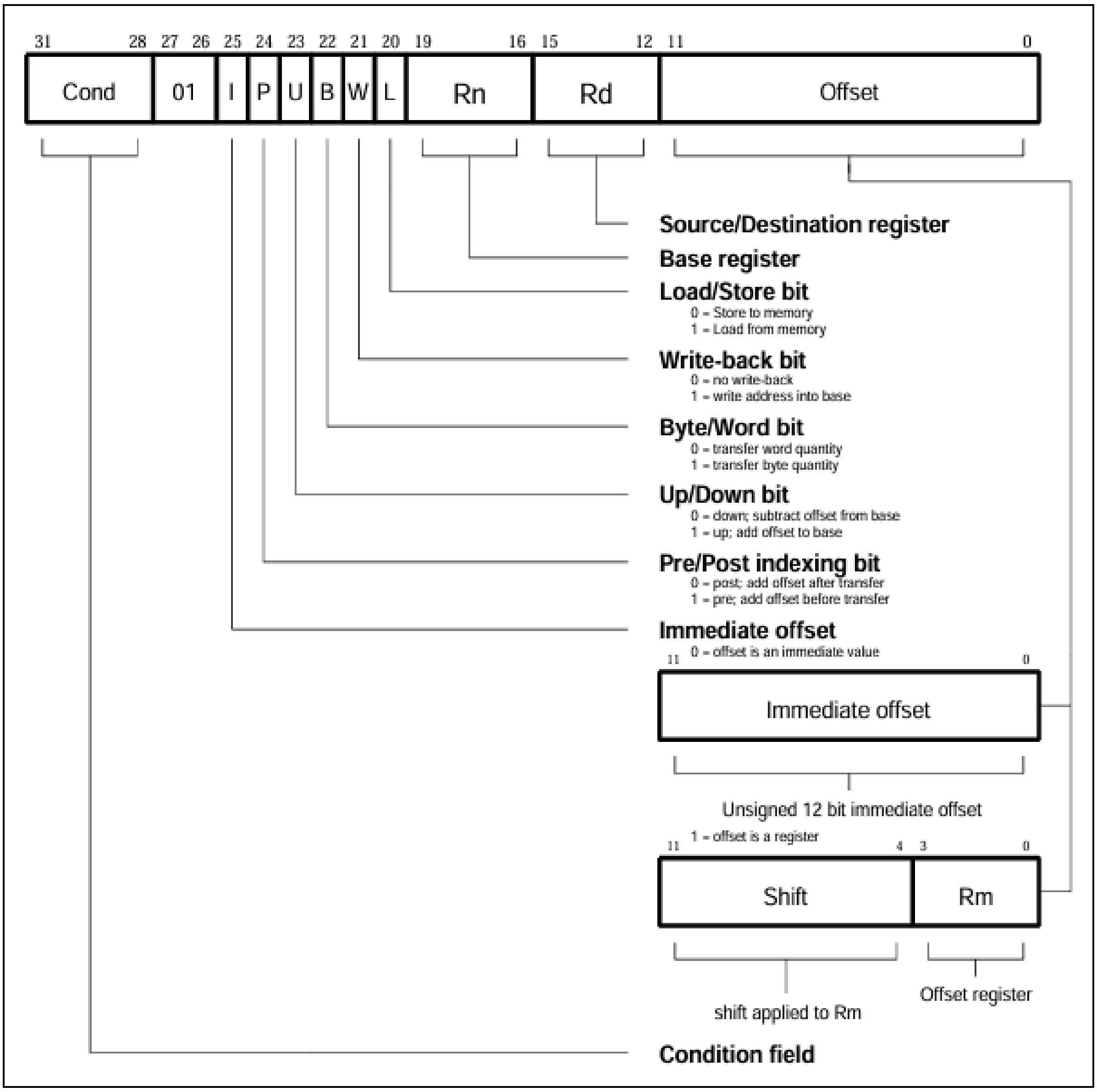
Each instruction is encoded using the **Single Data Transfer** ARM format, providing flexible addressing modes and 12-bit immediate offsets.

General Format

Each LDR or STR instruction is encoded using the following bit layout:

```
cond | 01 | I | P | U | B | W | L | Rn | Rd | offset12
```

- **cond (4 bits)** — condition code
- **01 (bits 27–26)** — identifies the instruction as a single data transfer
- **I** — addressing mode
- **P** — pre/post-indexing bit
- **U** — up/down bit
- **B** — byte/word bit
- **W** — write-back bit
- **L** — load/store bit
- **Rn** — base register (address register)
- **Rd** — destination/source register (for load/store)
- **offset12** — 12-bit immediate offset applied to Rn



Supported Instruction Formats

The encoder supports the most common ARM load and store addressing modes, using either a register or a register plus immediate offset.

LDR - Load Register from Memory

Loads a word from memory into a register.

Syntax:

```
LDR Rd, [Rn]
LDR Rd, [Rn, #imm]
```

Examples:

Assembly	Meaning	Encoding Explanation
`LDR R1, [R2]`	Load word from address in R2 into R1	offset12 = 0
`LDR R3, [R4, #12]`	Load word from address (R4 + 12)	offset12 = 12, U=1

STR - Store Register to Memory

Stores a register value into memory.

Syntax:

```
STR Rd, [Rn]
STR Rd, [Rn, #imm]
```

Examples:

Assembly	Meaning	Encoding Explanation
`STR R1, [R2]`	Store word in R1 to address in R2	offset12 = 0
`STR R3, [R4, #20]`	Store word in R3 to address (R4 + 20)	offset12 = 20, U=1

Summary

Instruction	Type	L bit	U bit	Description
LDR	Load	1	1 (add) / 0 (subtract)	Loads a word from memory into Rd
STR	Store	0	1 (add) / 0 (subtract)	Stores a word from Rd into memory

The Data Transfer Encoder efficiently implements memory access instructions for loading and storing register data.
It supports both direct and offset-based addressing, automatically determines offset direction, and adheres to ARM’s single data transfer encoding format.

Stack Instruction Encoder

The **Stack Instruction Encoder** is responsible for translating high-level stack operations - specifically **PUSH** and **POP** - into their ARM-style machine encodings.
Internally, these map to the ARM **Store Multiple (STM)** and **Load Multiple (LDM)** instructions, using **R13 (the Stack Pointer, SP)** as the base register.

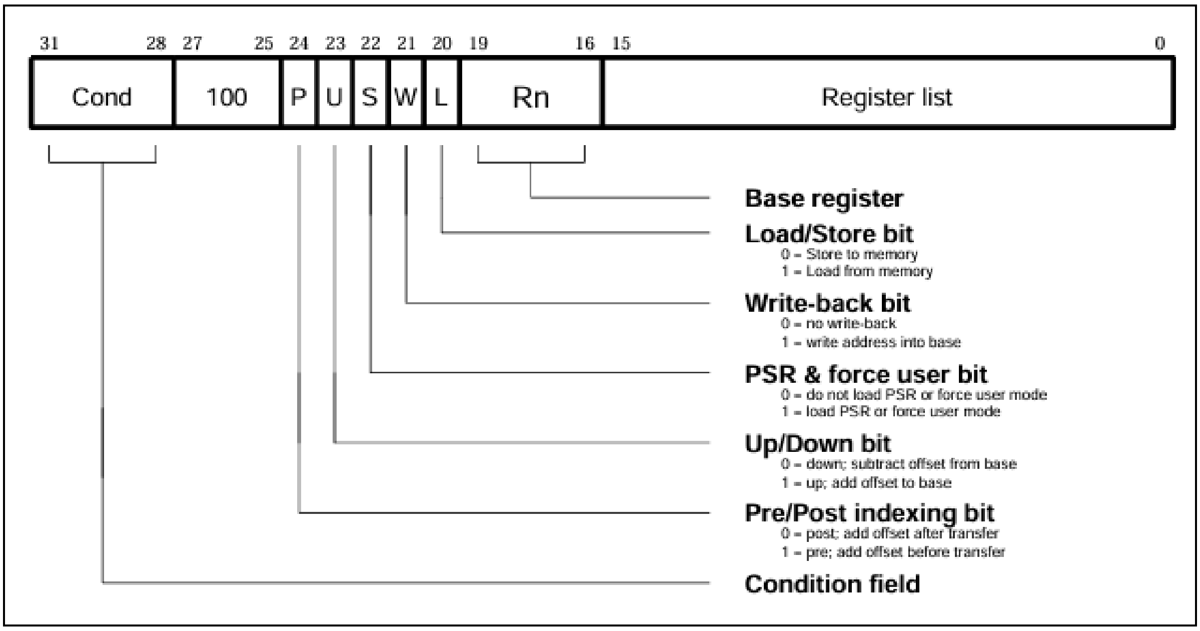
Stack operations are essential for implementing subroutines, preserving register states, and supporting pseudo-instructions like LSL, LSR and MOD that temporarily use a register (e.g., R12).

General Format

Each stack instruction follows the **Block Data Transfer** (LDM/STM) format:

cond | 100 | P | U | S | W | L | Rn | register_list

- cond — 4 bits: condition code (1110 = always)
- 100 — identifies this as a block transfer instruction
- P (Pre/Post) — Pre/Post-indexing bit (set to 1 for pre-indexing)
- U (Up/Down) — Stack direction bit (1 = increment, 0 = decrement)
- S (PSR & force user mode) — Not used here (set to 0)
- W (Write-back) — If set, base register (SP) is updated after transfer
- L (Load/Store) — 1 = Load (POP), 0 = Store (PUSH)
- Rn — Base register (always R13, the Stack Pointer)
- register_list — A 16-bit bitmask specifying which registers to transfer


31
28
27
25
24
23
22
21
20
19
16
15
0
Cond
100
P
U
S
W
L
Rn
Register list
Base register
Load/Store bit
0 = Store to memory
1 = Load from memory
Write-back bit
0 = no write-back
1 = write address into base
PSR & force user bit
0 = do not load PSR or force user mode
1 = load PSR or force user mode
Up/Down bit
0 = down; subtract offset from base
1 = up; add offset to base
Pre/Post indexing bit
0 = post; add offset after transfer
1 = pre; add offset before transfer
Condition field

Supported Instruction Formats

PUSH - Store Registers on the Stack

Pushes one or more registers onto the stack, decreasing the stack pointer after each write. Internally encoded as a **Store Multiple Decrement Full (STMFD)** operation.

Syntax:

```
PUSH {Rlist}
```

Behavior:

```
SP = SP - 4 * len(Rlist)
Store each register to memory starting at new SP
```

Encoding Example:

PUSH {R0, R1, R14}

POP - Load Registers from the Stack

Pops one or more registers from the stack, loading their previous values and incrementing the stack pointer after each read.
Internally encoded as a **Load Multiple Increment Full (LDMFD)** operation.

Syntax:

```
POP {Rlist}
```

Behavior:

```
Load each register from memory starting at SP
SP = SP + 4 * len(Rlist)
```

Summary

Instruction	Operation	W-bit	L-bit	Stack Direction	Description
PUSH	Store registers to stack	1	0	Decrement	Save register values
POP	Load registers from stack	1	1	Increment	Restore register values

The Stack Encoder efficiently implements ARM-style multiple register transfers, enabling stack-based function calls and temporary variable preservation.
It supports any combination of registers, correctly handles SP write-back, and integrates seamlessly with the CPU’s memory model.

Multiply/Divide Instruction Encoder

The **Multiply/Divide Instruction Encoder** is responsible for translating high-level arithmetic operations - specifically **MUL** and **DIV** - into their ARM-style machine encodings. These correspond to the ARM multiply instruction family, which use a compact data-processing format distinct from standard ALU instructions.

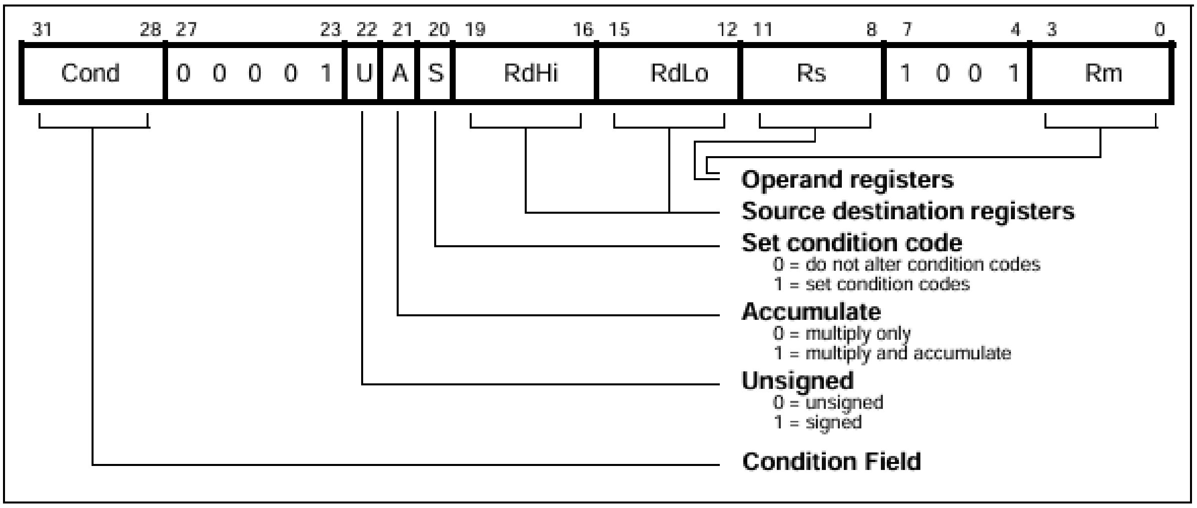
Multiplication and division are essential for implementing arithmetic operations, optimizing numeric routines, and supporting pseudo-instructions that rely on register-based computation.

General Format

Each multiply/divide instruction follows the ARM multiply instruction format:

```
cond | 000000 | A | S | Rd | Rn | Rs | 1001 | Rm
```

- **cond** — 4 bits: condition code
- **000000** — identifies this as a multiply/divide instruction
- **A (Operation)** — 0 = Multiply (MUL), 1 = Divide (DIV)
- **S (Set condition flags)** — 0 = do not update condition flags
- **Rd** — destination register (stores the result)
- **Rn** — unused for simple multiply/divide (set to 0)
- **Rs** — source register (multiplier or divisor)
- **1001** — fixed bit pattern identifying multiply class instructions
- **Rm** — operand register (multiplicand or dividend)



Supported Instruction Formats

MUL - Multiply

Performs an integer multiplication between registers.

Syntax:

```
MUL Rd, Rm, Rs
MUL Rd, Rs
```

Behavior:

```
Rd = Rm * Rs
```

If only two operands are given (`MUL Rd, Rs`), the destination register `Rd` is also used as the first operand (`MUL Rd, Rd, Rs`).

DIV - Divide

Performs an integer division between registers.

Syntax:

```
DIV Rd, Rm, Rs
DIV Rd, Rs
```

Behavior:

```
Rd = Rm / Rs
```

If only two operands are given (`DIV Rd, Rs`), the destination register `Rd` is also used as the first operand (`DIV Rd, Rd, Rs`).

Summary

The Multiply/Divide Encoder implements both **MUL** and **DIV** instructions using the ARM multiply instruction format.

While ARM architecture defines MUL and MLA (Multiply Accumulate) as the primary multiply-class operations, this simulator repurposes the MLA encoding pattern to represent integer division.

In other words, **DIV is encoded using the same bit structure that ARM reserves for MLA**, with the accumulator field (Rn) unused and set to zero.

This design choice ensures consistency in instruction layout and minimizes decoder complexity, allowing the simulator to support both multiplication and division without requiring a dedicated divide instruction format.

Although the real ARM ISA does not define a DIV instruction in this encoding family, the simulator extends it logically for completeness and ease of implementation.

System Instruction Encoder

The **System Instruction Encoder** is responsible for translating system-level operations - such as **HLT**, **INP**, and **OUT** - into their custom 32-bit machine encodings.
These instructions are used for halting execution, performing input/output operations, and interacting with external peripherals or the simulator environment.
Unlike standard ARM instructions, these operations follow a **custom encoding format** unique to this CPU model.

General Format

Each system instruction follows the custom system instruction format:

```
1111 | opcode(3) | operand(25)
```

- **1111** — 4-bit prefix identifying the instruction as a system-level operation
- **opcode (3 bits)** — selects the specific system instruction (`000 = HLT`, `001 = INP`, `010 = OUT`)
- **operand (25 bits)** — used for register data, register lists, or unused depending on instruction type

This compact encoding allows all system operations to be handled through a single decoder path, while leaving room for expansion.

Supported Instruction Formats

HLT - Halt Execution

Stops program execution immediately.
This instruction is typically used to terminate a program or signal that execution has completed.

Syntax:

```
HLT
```

Behavior:

```
Halt the CPU and stop instruction fetch.
```

Encoding Example:

```
1111 | 000 | 0000000000000000000000000
```

INP - Input from Device

Reads a value from an external input source (e.g., keyboard, console, or simulated I/O) and stores it in a destination register.

Syntax:

```
INP Rd
```

Behavior:

```
Rd = input()
```

OUT - Output to Device

Sends the contents of one or more registers to an external output device or stream. This is the primary mechanism for printing or exporting register data in the simulator.

Syntax:

```
OUT {Rlist}
```

Behavior:

```
output(Rlist)
```

Encoding Example:

```
1111 | 010 | register_list_mask
```

Where `register_list_mask` is a 25-bit field, each bit representing a register to output (`bit 0 = R0`, `bit 1 = R1`, etc.).

Summary

Instruction	Opcode	Operand Field	Description
HLT	`000`	Unused	Halts CPU execution
INP	`001`	Destination register	Reads input value into Rd
OUT	`010`	Register list bitmask	Writes values from listed registers to output

Summary

The System Instruction Encoder provides a lightweight, extensible mechanism for handling **non-ALU operations** that control program execution and I/O behavior.
All system instructions share a **common 4-bit 1111 prefix**, which cleanly separates them from regular data-processing and memory-transfer instructions.
While this encoding is not derived from the real ARM ISA, it is purpose-built for this simulator to enable essential system operations such as halting, input/output, and debugging.
Future instructions can be easily added by assigning new opcode values within the same 3-bit opcode field.

Pseudo-Instruction Encoder

The **Pseudo-Instruction Encoder** handles the translation of higher-level, non-native instructions into sequences of valid machine instructions supported by the CPU.
These pseudo-instructions simplify assembly programming by providing shorthand notations for common operations that expand internally into equivalent combinations of **data processing**, **stack**, and **multiply/divide** instructions.
All pseudo-instructions are ultimately resolved during encoding, so the CPU executes only real ARM-style instructions.

General Overview

Pseudo-instructions are not directly part of the CPU's instruction set - instead, they are **synthetic constructs** expanded by the assembler into one or more low-level instructions.
This encoder provides translation for:

```
INC, DEC, CLR, LSL, LSR, MOD, SWAP
```

Each pseudo-instruction reuses existing encoder modules:

- **Data Processing Encoder** - for arithmetic and move operations
- **Multiply/Divide Encoder** - for multiply and divide operations
- **Stack Instruction Encoder** - for saving and restoring temporary registers (R12)

Supported Pseudo-Instructions

INC - Increment Register

Syntax:

```
INC Rd
```

Expansion:

```
ADD Rd, Rd, #1
```

Description:

Increments the contents of register Rd by 1 using an immediate addition.

DEC - Decrement Register

Syntax:

```
DEC Rd
```

Expansion:

```
SUB Rd, Rd, #1
```

Description:
Decrements the contents of register Rd by 1 using an immediate subtraction.

CLR - Clear Register

Syntax:

```
CLR Rd
```

Expansion:

```
MOV Rd, #0
```

Description:
Clears (sets to zero) the specified register.

LSL - Logical Shift Left

Syntax:

```
LSL Rd, Rn, #imm
LSL Rd, #imm
```

Expansion Sequence:

```
PUSH {R12}
```

```
MOV R12, #(2 ** imm)
MUL Rd, Rn, R12
POP {R12}
```

Description:
Shifts Rn left by imm bits using multiplication by a power of two.
Temporarily uses register **R12** as a working register, preserving it with stack operations.

LSR - Logical Shift Right

Syntax:

```
LSR Rd, Rn, #imm
LSR Rd, #imm
```

Expansion Sequence:

```
PUSH {R12}
MOV R12, #(2 ** imm)
DIV Rd, Rn, R12
POP {R12}
```

Description:
Shifts Rn right by imm bits using integer division by a power of two.
Also uses **R12** as a temporary register to perform the division safely.

MOD - Modulus

Syntax:

```
MOD Rd, Rn, Rm
```

Expansion Sequence:

```
PUSH {R12}
DIV R12, Rn, Rm
MUL R12, R12, Rm
SUB Rd, Rn, R12
POP {R12}
```

Description:
Computes Rd = Rn % Rm using integer division and multiplication to find the remainder.
This pseudo-instruction extends the arithmetic set without requiring a dedicated hardware modulus operation.

SWAP (or SWP) - Swap Two Registers

Syntax:

```
SWAP Rn, Rm
```

Expansion Sequence:

```
PUSH {R12}
MOV R12, Rn
MOV Rn, Rm
MOV Rm, R12
POP {R12}
```

Description:
Swaps the values of two registers using **R12** as a temporary storage register.
The original value of **R12** is preserved via stack push/pop.

Summary

Pseudo-Instruction	Expansion	Uses Stack	Temp Register	Description
INC Rd	`ADD Rd, Rd, #1`	No	—	Increment register
DEC Rd	`SUB Rd, Rd, #1`	No	—	Decrement register
CLR Rd	`MOV Rd, #0`	No	—	Clear register
LSL Rd, Rn, #imm	`PUSH/POP R12`, `MOV`, `MUL`	Yes	R12	Logical shift left
LSR Rd, Rn, #imm	`PUSH/POP R12`, `MOV`, `DIV`	Yes	R12	Logical shift right
MOD Rd, Rn, Rm	`PUSH/POP R12`, `DIV`, `MUL`, `SUB`	Yes	R12	Compute remainder
SWAP Rn, Rm	`PUSH/POP R12`, `MOV`	Yes	R12	Exchange register values

Summary

The Pseudo-Instruction Encoder enables a higher-level, user-friendly assembly interface by expanding shorthand operations like `INC`, `DEC`, and `CLR` into real ARM-style instructions. More complex pseudo-operations such as `LSL`, `LSR`, `MOD`, and `SWAP` use **R12** as a scratch register, protected through **stack save and restore** operations to maintain program

correctness.
By combining existing encoders (data-processing, multiply/divide, and stack), this system achieves flexible macro-instruction support without altering the CPU's native instruction set or decoder logic.

The Decoder

The **Decoder** is the core component responsible for interpreting each 32-bit machine instruction fetched from memory and determining what operation it represents.
It translates the raw binary word produced by the assembler into concrete actions performed by the CPU - such as arithmetic, data movement, branching, or system control.

Each instruction type has a dedicated decoder function that extracts operands, interprets control bits, and executes the corresponding behavior on the CPU's registers, memory, and flags.

General Operation

During program execution, the CPU repeatedly performs the **fetch - decode - execute** cycle.
In the **decode** stage, the instruction's bit pattern is analyzed to determine its category.
Based on this classification, the CPU calls the appropriate decode function, which interprets the instruction's fields and carries out its operation.

Each instruction category follows its own encoding pattern, allowing the decoder to distinguish between them using only a few key bits.
This modular design ensures clarity and scalability, so new instruction types can be added without altering the CPU's main control flow.

Decoding Workflow

1. **Instruction Fetch**
   The CPU reads the next 32-bit instruction from instruction memory at the address indicated by the Program Counter (R15).

2. **Pattern Identification**
   The high-order bits of the instruction are compared against known patterns to identify the instruction type:

   - `00` -> Data Processing
   - `01` -> Data Transfer (LDR/STR)
   - `100` -> Stack (PUSH/POP)
   - `101` -> Branch (B, BL, RET)
   - `000000...1001` -> Multiply/Divide (MUL/DIV)
   - `1111` -> System (HLT, INP, OUT)

3. **Delegation to Specific Decoder**

Once the category is determined, the instruction is passed to the corresponding function:

```
execute_load_store(instruction, cpu, memory)
execute_branch(instruction, cpu)
execute_multiply_set(instruction, cpu)
execute_stack_set(instruction, cpu, memory)
execute_system_instruction(instruction, cpu)
execute_data_processing(instruction, cpu)
```

4. **Operand Extraction and Execution**
   Each decoder extracts its operands by shifting and masking the instruction bits.
   It then performs the specified operation, updating registers, flags, or memory as required.

Integration with the CPU

Once a decoder executes its corresponding function, it may:

- Update general-purpose registers (R0–R12)
- Modify special registers (R13–R15 for SP, LR, PC)
- Adjust condition flags (N, Z, C, V)
- Read or write data memory
- Control I/O or halt execution (via system instructions)

This modular approach mirrors real ARM CPU design, where decoding logic is distributed across specialized hardware units.
In this simulator, each decoder is implemented in software but follows the same architectural principles.

Summary

Instruction Type	Bit Pattern	Decoder Function	Primary Actions
Data Processing	`00`	`decode_data_processing_instruction()`	Arithmetic, logical, and move operations
Data Transfer	`01`	`decode_load_store()`	Load and store between registers and memory
Stack Operations	`100`	`decode_stack_instruction()`	PUSH and POP register sets

Branching	`101`	`decode_branch()`	Program flow control and subroutine calls
Multiply/Divide	`000000...1001`	`decode_multiply_set()`	Multiplication and division
System Instructions	`1111`	`decode_system_instruction()`	HLT, INP, OUT, and control operations

Summary

The Decoder serves as the bridge between the binary instruction stream and the logical behavior of the simulated CPU.
By analyzing fixed bit patterns and delegating to specialized decoders, it ensures each instruction is interpreted and executed precisely according to its class.
This modular architecture makes the system extensible, allowing new instruction types or pseudo-instructions to be added with minimal changes to the core logic.
Through clear pattern separation and consistent bit-field decoding, the Decoder faithfully reproduces the control logic of a simplified ARM processor, ensuring accurate and efficient simulation of real CPU behavior.

The Memory

The **Memory** module provides a simplified model of system memory used by both the assembler and the CPU.
It serves as a unified storage space for program instructions and data, enabling read and write operations during execution.
In the ARM Assembly Simulator, the `Memory` class emulates byte-addressable memory with 32-bit word access - closely resembling real processor memory behavior, while remaining lightweight and efficient in Python.

General Overview

Each simulated memory space (instruction and data) is represented by an instance of the `Memory` class.
Internally, it uses a `bytearray` to store raw bytes, supporting flexible access and easy visualization of stored contents.
The default size is **4096 bytes (4 KB)**, though larger or smaller configurations can be specified at initialization.

```
memory = Memory(size=4096)
```

This structure is used for both:

- **Instruction Memory** — stores the machine code generated by the assembler
- **Data Memory** — stores variables, stack data, and runtime values modified by the CPU

Memory Organization

Memory in this simulator is **byte-addressable** but primarily accessed in **32-bit words** (4 bytes).
Each word is aligned on a 4-byte boundary, matching ARM's word-aligned access model.

Address	Content	Description
0x0000	Instruction 1	Start of program code
0x0004	Instruction 2	Next instruction
...	...	...
0x0FF0	Stack	Top of stack (SP) grows downward

All values are stored in **big-endian format**, meaning the most significant byte (MSB) is stored at the lowest address.

Core Methods

`read_word(addr)`

Reads a 32-bit word (4 bytes) from the specified address.

- **Parameters:**
  `addr` - Memory address (must be within valid range)
- **Returns:**
  32-bit unsigned integer
- **Behavior:**
  Combines 4 bytes starting at `addr` into a single integer using big-endian order.

If the address is outside the memory range, a `MemoryError` is raised.

`write_word(addr, value)`

Writes a 32-bit word (4 bytes) into memory.

- **Parameters:**
  `addr` - Destination address (must be within valid range)

value - Integer value to store (automatically masked to 32 bits)

- **Behavior:**
  Converts the integer into 4 bytes (big-endian) and writes them into memory.

Raises a MemoryError if writing beyond the allocated space.

load_bytes(data, start_addr=0)

Loads an arbitrary byte sequence into memory, commonly used to initialize instruction memory.

- **Parameters:**
  data - bytes object containing data to load
  start_addr — Memory address to begin loading (default = 0)

- **Behavior:**
  Writes the byte array into memory sequentially starting at the given address.

Integration with the CPU

Two instances of the Memory class are created at runtime:

1. **Instruction Memory**
   - Holds all machine code words produced by the assembler
   - Accessed by the CPU during the **fetch** stage
   - Read-only during execution
2. **Data Memory**
   - Used for runtime data, stack operations, and variable storage
   - Accessed by the CPU via **LDR**, **STR**, **PUSH**, and **POP**
   - Supports both read and write operations

The **Stack Pointer (R13)** is initialized to the **end of data memory**, and stack operations grow downward as data is pushed.

Memory Safety and Alignment

The simulator enforces strict bounds checking for all memory accesses.
Any read or write outside the valid range raises a MemoryError, preventing segmentation faults or undefined behavior.
This ensures stability and predictable results, even during invalid program execution.

Furthermore, the simulator assumes **word-aligned accesses** for simplicity and correctness - unaligned accesses are not permitted, mirroring ARM's traditional alignment rules.

Summary

The Memory module provides a simple yet faithful abstraction of ARM-style main memory. It allows 32-bit word-level access with byte-level addressing, enforces range checks for safety, and stores data in big-endian order for realism.
Used in tandem with the CPU, it underpins all load/store, stack, and instruction-fetch operations within the simulator.
By keeping the design modular and robust, the Memory class ensures reliable emulation of both instruction and data spaces while maintaining clarity and educational value.

The CPU

The **CPU** (Central Processing Unit) is the core of the ARM Assembly Simulator.
It emulates the execution stage of a simplified ARM-like processor by managing registers, condition flags, instruction flow, and memory interactions.
The CPU coordinates all major stages of the simulation - fetching, decoding, and executing instructions - while maintaining the current program state and ensuring correct data flow between the instruction memory, data memory, and I/O system.

General Overview

The CPU model used in this simulator closely mirrors a classical **Von Neumann** architecture, where a unified memory space holds both instructions and data.
It contains **16 general-purpose registers (R0–R15)**, along with a set of **status flags** (N, Z, C, V) that record arithmetic and logical outcomes.
Each instruction cycle is executed in three main stages:

1. **Fetch:** Read a 32-bit instruction from instruction memory at the address in the Program Counter (PC = R15).

2. **Decode:** Identify the instruction type and extract operands via the Decoder.

3. **Execute:** Perform the operation, update registers and flags, and advance the Program Counter.

This cycle repeats continuously until a HLT instruction or an invalid memory access stops execution.

CPU Registers and Flags

Register	Name	Description
R0–R12	General Purpose	Used for arithmetic, logic, and temporary storage

R13 (SP)	Stack Pointer	Points to the current top of the stack in data memory
R14 (LR)	Link Register	Holds the return address for subroutine calls
R15 (PC)	Program Counter	Holds the address of the next instruction to execute

The **status flags** are stored in a dictionary (`cpu.flags`) and updated automatically by arithmetic or logical operations:

Flag	Meaning	Description
N (Negative)	1 if the result is negative	Indicates sign of the result
Z (Zero)	1 if the result is zero	Indicates equality or empty result
C (Carry)	1 if addition/subtraction generated a carry	Used in unsigned arithmetic
V (Overflow)	1 if signed overflow occurred	Used in signed arithmetic checks

Execution Control

The CPU begins execution with:

- **PC (R15)** initialized to address **0**
- **SP (R13)** initialized to the **end of data memory**
- **All other registers** set to **0**

Execution starts via:

```
cpu.run()
```

The `run()` method repeatedly calls `get_instruction()` until either:

- A **HLT** instruction stops execution
- A **memory access error** occurs
- The **maximum step count** is reached (default: 100 instructions)

When halted, the CPU prints a confirmation message:

```
Execution halted.
```

Summary

The CPU acts as the central execution engine of the simulator.
It orchestrates the entire instruction cycle -fetching from memory, decoding instruction types, executing the corresponding logic, and updating registers, flags, and memory accordingly.
Its modular structure and clean separation between instruction types enable extensibility and maintain fidelity to real ARM processor behavior.
Through clear state visualization and debugging tools, the CPU not only executes assembly programs but also provides an educational view into how real microarchitectures operate internally.

Bibliography

1. **ARM Instruction Set Reference**
   ARM Architecture Reference for Instruction Set and Encoding Formats.
   Indian Institute of Technology Delhi - COL718 Course Reference.
   Retrieved from:
   ARM Instruction Set PDF

2. **OpenAI ChatGPT (GPT-5)**
   Conversational AI model used for technical guidance, documentation drafting, and clarification of ARM instruction set concepts.
   https://chat.openai.com