

ARM 32-bit Assembly Programming

A practical study guide: registers, the ARM procedure call standard, instructions, the stack, diagrams and fully worked examples



Compiled by HARDIK

LinkedIn: www.linkedin.com/in/hardikhardik

Built from personal study notes and extended with diagrams, explanations, tested code examples.

Contents

1 Getting Started	5
1.1 What is 32-bit ARM?	5
1.2 Your first program	5
1.3 Assembling and running	5
1.4 Anatomy of an instruction	5
2 Registers and the Function Call ABI	7
2.1 The register file	7
2.2 Caller-saved versus callee-saved	7
2.3 Argument passing summary	8
3 System Calls	9
3.1 The syscall ABI	9
3.2 Common Linux syscall numbers	9
3.3 Example: Hello, ARM!	10
4 Sections and Directives	11
4.1 Defining sections	11
4.2 Useful directives	11
4.3 Alignment in the .bss section	11
5 Data Types, Variables and Alignment	13
5.1 Data types	13
5.2 Declaring variables	13
5.3 Alignment	13
6 Loading and Storing Data	14
6.1 Why addresses go through a register	14
6.2 Integers, characters and halfwords	14
6.3 Signed and unsigned loads	15
6.4 Addressing modes	15
6.5 Loading constants	16
7 Condition Codes and Flags	17
7.1 The CPSR flags	17
7.2 Condition code table	17
7.3 How flags get set	18
7.4 Signed versus unsigned	18
8 Operators and Expressions	19

8.1 Arithmetic operators	19
8.2 Increment and decrement	20
8.3 Compound assignment	20
8.4 Relational operators	20
8.5 Logical operators	21
8.6 Bitwise operators	21
8.7 Shift operators	22
8.8 Conditional execution	22
8.9 Creating branches	23
9 Control-Flow Patterns	24
9.1 Reading CMP like C	24
9.2 if / else	24
9.3 while loop	24
9.4 for loop	24
9.5 do-while loop	25
10 Arrays and Scaling	26
10.1 Looping over an array	26
10.2 Scaling: accessing array elements	26
11 Functions and the Stack Frame	29
11.1 A simple function	29
11.2 The stack and PUSH / POP	29
11.3 A function with a stack frame	30
11.4 Local variables on the stack	30
11.5 Nested functions	31
11.6 More than four arguments	31
11.7 Recursion	32
12 Worked Examples	33
12.1 Greatest common divisor	33
12.2 Fibonacci	33
12.3 String length	34
12.4 Counting set bits	34
12.5 Find the maximum of an array	34
13 Quick Reference and Practice	35
13.1 Instruction cheat sheet	35
13.2 Function checklist	35
13.3 Practice exercises	35
13.4 Where to go next	36

Preface

This book started life as a plain text file of study notes on **ARM 32-bit assembly**. It has been reorganised into chapters, illustrated with diagrams, and extended with extra explanations and complete example programs. The original topics and their order are kept, so the book can be used as a revision guide as well as a first introduction.

Four kinds of coloured boxes appear throughout:

NOTE

NOTE marks an important point taken from the original notes.

TIP

TIP marks a habit or shortcut that makes your code cleaner or safer.

WATCH OUT

WATCH OUT marks a common mistake or a common pitfall.

EXTRA

EXTRA marks material added to the original notes: new explanations, patterns and examples.

All code uses **GNU assembler (GAS)** syntax with `@` comments and the `.syntax unified` directive. Every complete program in this book was assembled with `arm-linux-gnueabi-as` and executed under `qemu-arm`, and the results quoted in the text are the ones actually produced. Programs target ARMv7 (32-bit) running Linux. Where a program needs a feature that older cores lack (for example **SDIV**), the text says so.

Use the exit code of a program (`echo $?`) as a quick way to check your results: most examples finish by passing their answer to the `exit` system call.

1 Getting Started

1.1 What is 32-bit ARM?

ARM is a family of **RISC** (Reduced Instruction Set Computer) processors. In 32-bit ARM state every instruction is 32 bits wide and the CPU has sixteen 32-bit core registers, **r0 to r15**. Three ideas shape almost everything you will write:

- **Load/store architecture.** Arithmetic works only on registers. Memory is touched only by load (**LDR**) and store (**STR**) instructions.
- **Conditional execution.** Most instructions can carry a condition such as **EQ** or **GT** and execute only if the flags match (for example **MOVGT**).
- **Barrel shifter.** The second operand of many instructions can be shifted for free, which is the secret behind array indexing like **[r3, r0, LSL #2]**.

The rules that describe how functions pass arguments and who must preserve which registers are called the **ARM Architecture Procedure Call Standard (AAPCS)**. Chapter 2 covers it.

1.2 Your first program

Every program needs an entry point called **_start**, declared global so the linker can find it. This tiny program exits with status 42:

```
.syntax unified
.section .text
.globl _start
_start:
    MOV r0, #42          @ exit status
    MOV r7, #1           @ syscall number 1 = exit
    SWI 0                @ call the kernel
```

1.3 Assembling and running

EXTRA

On an x86 Linux PC install the cross tools with **sudo apt install binutils-arm-linux-gnueabi** **qemu-user**. On a Raspberry Pi running a 32-bit OS, use plain **as** and **ld** instead.

```
arm-linux-gnueabi-as -march=armv7ve first.s -o first.o
arm-linux-gnueabi-ld first.o -o first
qemu-arm ./first
echo $?                # prints 42
```

1.4 Anatomy of an instruction

```
label:  MNEMONIC{cond}{S}  operands      @ comment
loop:   ADDNE              r2, r2, r4        @ add only if Z == 0
        SUBS               r0, r0, #1       @ subtract and update flags
```

Symbol	Meaning	Example
#	Immediate (constant) value	MOV r0, #5

Symbol	Meaning	Example
[]	Memory access through the address inside	LDR r1, [r0]
{ }	A list of registers	PUSH {r4, lr}
!	Write the updated address back to the base register	LDR r1, [r0, #4]!
=	Pseudo-instruction: load a constant or an address	LDR r0, =msg
@	Start of a comment (GAS for ARM)	@ i = 0

WATCH OUT

In GNU assembler for ARM the comment character is @. A semicolon ; is not a comment, and // is not reliable on every version. This book uses @ everywhere so that every listing assembles.

HARDIK

2 Registers and the Function Call ABI

2.1 The register file

ARM 32-bit core registers

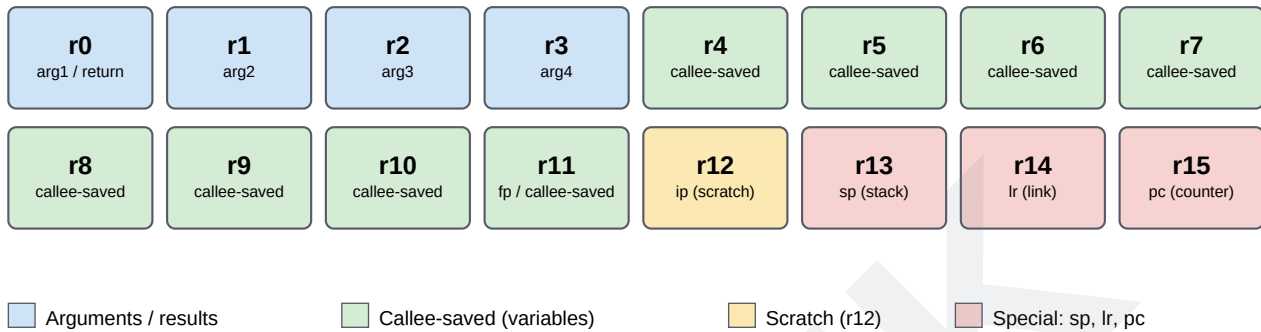


Figure 2.1: The sixteen core registers and the role each plays under the AAPCS.

Register	Alias	Meaning	Who preserves it?
r0 - r3	a1 - a4	Function arguments; scratch registers	Caller (callee may overwrite)
r0	a1	Return value (r0:r1 for a 64-bit result)	Caller
r4 - r11	v1 - v8	Variables, loop counters; callee-saved	Callee must restore
r12	ip	Intra-procedure-call scratch register	Caller
r13	sp	Stack pointer	Callee (must be restored)
r14	lr	Link register: return address	Callee must save if it calls others
r15	pc	Program counter	n/a

2.2 Caller-saved versus callee-saved

This is the single most important idea in the calling convention. A register is either one a function may freely destroy or one it must give back unchanged:

- **r0-r3 and r12 are scratch.** If you are the caller and you still need a value held in one of them after a call, save it yourself before calling.
- **r4-r11 are callee-saved.** If you are writing a function and you want to use r4, you must **PUSH** it on entry and **POP** it before returning, so your caller never notices.
- **lr must survive a nested call.** **BL** overwrites lr, so a function that calls another function must save lr first (Chapter 11).

NOTE

r0-r3 hold the first four arguments, r0 holds the return value, and r4-r11 are for your own variables and loop counters.

2.3 Argument passing summary

Situation	Rule
First four arguments	r0, r1, r2, r3 (left to right)
Fifth and later arguments	Pushed on the stack; the fifth sits at <code>[sp]</code> at the moment of the call
32-bit return value	r0
64-bit return value	r0 (low word) and r1 (high word)
Stack alignment at a call	sp must be a multiple of 8 bytes

3 System Calls

3.1 The syscall ABI

A system call asks the operating system to do something for your program: write text, read a file, exit. On 32-bit ARM Linux you load a number into **r7**, put the arguments in **r0-r6**, and execute a software interrupt. The result comes back in **r0**.

Register	Purpose
r7	System call number
r0 - r6	Arguments
r0	Return value

NOTE

Use **SWI** to make the call. **SWI** (software interrupt) is the older name; modern assemblers also accept the identical **SVC** (supervisor call). Both mean the same instruction.

3.2 Common Linux syscall numbers

Number (r7)	Name	r0	r1	r2
1	exit	status	-	-
3	read	file descriptor	buffer address	byte count
4	write	file descriptor	buffer address	byte count

EXTRA

File descriptor 0 is standard input, 1 is standard output and 2 is standard error. These numbers belong to Linux EABI; other operating systems use different tables.

3.3 Example: Hello, ARM!

```
.syntax unified
.section .data
msg:    .ascii "Hello, ARM!\n"
len = . - msg           @ length computed by the assembler

.section .text
.globl _start
_start:
    MOV r7, #4           @ write
    MOV r0, #1           @ fd 1 = stdout
    LDR r1, =msg          @ buffer address
    MOV r2, #len          @ number of bytes
    SWI 0

    MOV r7, #1           @ exit
    MOV r0, #0           @ status 0
    SWI 0
```

`len = . - msg` subtracts the address of `msg` from the current location (`.`), so the byte count is always correct even if you edit the string. We use `.ascii` here rather than `.asciz` because `.asciz` adds a terminating zero byte that `write` does not need.

4 Sections and Directives

4.1 Defining sections

A program is split into sections, each with a different purpose and different permissions. The linker then places them in memory.

```
.section .text      @ code
.section .data      @ initialised, writable variables
.section .bss       @ zero-initialised variables
.section .rodata    @ read-only constants
.globl _start       @ make the entry label visible to the linker
```

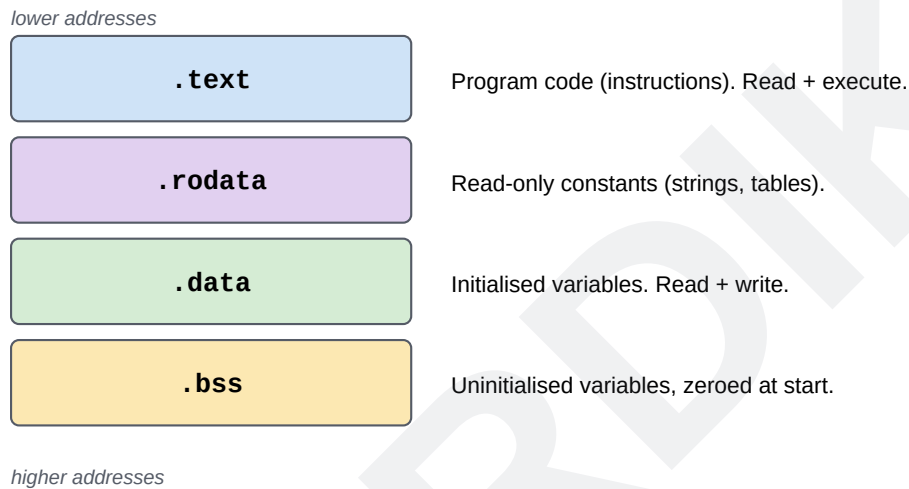


Figure 4.1: The main sections of a program image, from low to high addresses.

4.2 Useful directives

Directive	What it does	Example
.byte	Reserve 1 byte (or a list of bytes)	c: .byte 0x41
.hword	Reserve 2 bytes	s: .hword 0x1234
.word	Reserve 4 bytes	x: .word 0x12345678
.quad	Reserve 8 bytes	q: .quad 1, 2, 3
.ascii	String without terminator	m: .ascii "hi"
.asciz	String with a trailing zero byte	m: .asciz "hi"
.space N	Reserve N zero bytes	buf: .space 64
.equ name, v	Define a constant	.equ SIZE, 5
.align N	Pad to a 2^N-byte boundary	.align 2
.global sym	Export a symbol	.global add
.type f, %function	Mark a label as a function	.type add, %function

4.3 Alignment in the .bss section

NOTE

Always use `.align 4` in the `.bss` section.

```
.section .bss
.align 4          @ 2^4 = 16-byte boundary
buffer: .space 64
```

WATCH OUT

On ARM, GNU `.align N` means a boundary of **2 to the power N** bytes, so `.align 4` is a 16-byte boundary and `.align 2` a 4-byte (word) boundary. Either keeps words correctly aligned; 16 bytes is simply more generous.

HARDIK

5 Data Types, Variables and Alignment

5.1 Data types

C type	Size	ARM directive
char	1 byte	.byte
short	2 bytes	.hword
int	4 bytes	.word
long	4 bytes (ARM 32-bit)	.word
pointer	4 bytes	.word
long long	8 bytes	.quad

EXTRA

A pointer is only 4 bytes on 32-bit ARM, which is why **long** and pointers share **.word**. On 64-bit systems both grow to 8 bytes.

5.2 Declaring variables

```
.section .data
x:  .word 0x12345678    @ int    x = 0x12345678;
c:  .byte 0x41          @ char   c = 'A';
s:  .hword 0x1234       @ short  s = 0x1234;
msg: .asciz "hi"        @ char   msg[] = "hi";
arr: .word 10, 20, 30   @ int    arr[3] = {10, 20, 30};
```

5.3 Alignment

ARM prefers **4-byte alignment for words** and **2-byte alignment for halfwords**. A word stored at an address that is not a multiple of 4 may be slow, may fault on older cores, and is not allowed by some instructions. Place **.align** before data that needs it:

```
.align 2
x: .word 10
```

TIP

If you put a **.byte** before a **.word**, the word may land on an odd address. Add **.align 2** between them to push the word to the next 4-byte boundary.

```
c: .byte 0x41
.align 2           @ pad so the next word is 4-byte aligned
x: .word 10
```

6 Loading and Storing Data

6.1 Why addresses go through a register

In ARM you cannot name a variable directly inside an arithmetic instruction. First put the **address** of the variable in a register, then use that register inside brackets to read or write the value. `LDR r0, =x` is a convenience: the assembler stores the address in a nearby table (the literal pool) and loads it for you.

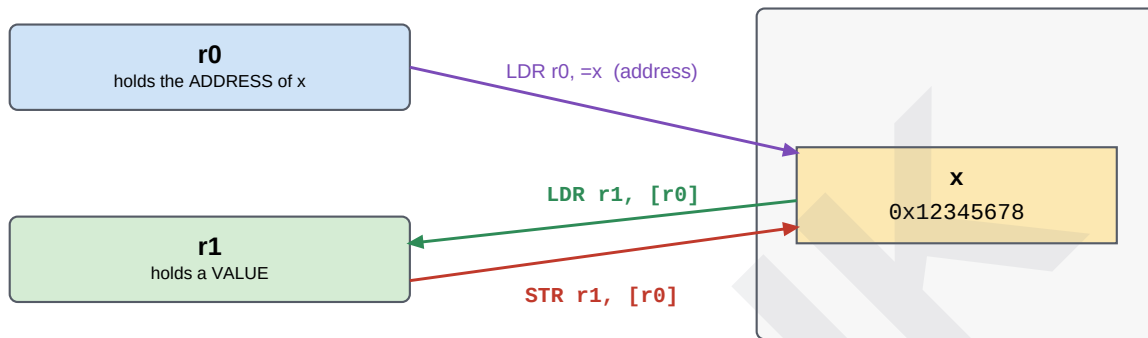


Figure 6.1: `r0` holds the address; `LDR` copies memory into `r1`; `STR` copies `r1` back.

6.2 Integers, characters and halfwords

Integer (word)

```
.section .data
x: .word 0x12345678
.section .text
    LDR r0, =x      @ r0 = address of x
    LDR r1, [r0]    @ r1 = x
    STR r1, [r0]    @ x = r1
```

Character (byte)

```
.section .data
c: .byte 0x41
.section .text
    LDR r0, =c      @ address of c
    LDRB r1, [r0]   @ load byte
    STRB r1, [r0]   @ store byte
```

Short (halfword)

```
.section .data
s: .hword 0x1234
.section .text
    LDR r0, =s      @ address of s
    LDRH r1, [r0]   @ load halfword
    STRH r1, [r0]   @ store halfword
```

Strings

```
msg: .asciz "hi"
    LDR r0, =msg
    LDRB r1, [r0]      @ 'h'
    LDRB r2, [r0, #1]  @ 'i'
```

Constant data

```
.section .rodata
k: .word 5                @ read-only: writing to it will crash
```

6.3 Signed and unsigned loads

Instruction	Meaning	Fills the upper bits with
LDRB	Unsigned byte	zeros (zero-extend)
LDRSB	Signed byte	copies of bit 7 (sign-extend)
LDRH	Unsigned halfword	zeros
LDRSH	Signed halfword	copies of bit 15

```
c: .byte 0xFF
    LDR r3, =c
    LDRB r0, [r3]      @ r0 = 0x000000FF = 255 (zero extend)
    LDRSB r1, [r3]     @ r1 = 0xFFFFFFFF = -1 (sign extend)
```

NOTE

In **LDRSB r0, [addr]** the **addr** stands for a base register such as **[r3]**. **LDRSB** sign-extends; **LDRB** zero-extends.

6.4 Addressing modes

EXTRA

These forms are used constantly in real code. Suppose **r0** is a pointer.

Form	Effect	Name
LDR r1, [r0]	r1 = *r0	Plain
LDR r1, [r0, #4]	r1 = *(r0 + 4); r0 unchanged	Immediate offset
LDR r1, [r0, #4]!	r0 = r0 + 4; then r1 = *r0	Pre-indexed
LDR r1, [r0], #4	r1 = *r0; then r0 = r0 + 4	Post-indexed
LDR r1, [r0, r2]	r1 = *(r0 + r2)	Register offset
LDR r1, [r0, r2, LSL #2]	r1 = *(r0 + r2*4)	Scaled register offset

The post-indexed form is the neat way to walk through an array: each load also advances the pointer.

```
LDR r3, =arr
LDR r4, [r3], #4      @ r4 = arr[0]; r3 now points at arr[1]
LDR r5, [r3], #4      @ r5 = arr[1]; r3 now points at arr[2]
```

6.5 Loading constants

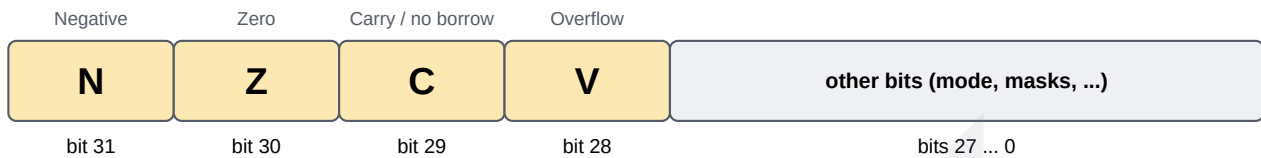
`MOV r0, #imm` can only encode an 8-bit value rotated by an even number of bits. `MOV r0, #255` and `MOV r0, #256` work; `MOV r0, #257` does not. For any 32-bit constant use the `=` pseudo-instruction and let the assembler choose:

```
LDR r0, =0x12345678    @ any 32-bit constant
LDR r1, =257            @ works even though MOV r1, #257 would not
```

7 Condition Codes and Flags

7.1 The CPSR flags

When an instruction updates the flags, four bits in the Current Program Status Register record what happened. Conditional instructions then test those bits.



CPSR = Current Program Status Register. CMP, CMN, TST, TEQ and any instruction with the S suffix update N Z C V.

Figure 7.1: The four condition flags at the top of the CPSR.

Flag	Meaning	Set when
N	Negative	the result has its top bit set
Z	Zero	the result is exactly zero
C	Carry / no borrow	an addition carries out, or a subtraction does not borrow
V	Overflow	the signed result does not fit in 32 bits

7.2 Condition code table

Code	Meaning (after CMP or SUBS)	Flags tested
eq	Equal	Z == 1
ne	Not equal	Z == 0
cs / hs	Unsigned higher or same (carry set)	C == 1
cc / lo	Unsigned lower (carry clear)	C == 0
mi	Negative (minus)	N == 1
pl	Positive or zero (plus)	N == 0
vs	Signed overflow (V set)	V == 1
vc	No signed overflow (V 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
al	Always (the default if omitted)	none tested

NOTE

These codes attach to instructions: **MOVEQ**, **MOVNE**, **BGT**, **BLE**, **BGE**, **BNE** and so on.

7.3 How flags get set

- **CMP** *a*, *b* computes *a* - *b*, throws the result away, and sets the flags.
- **CMN** *a*, *b* does the same for *a* + *b*; **TST** *a*, *b* for *a* AND *b*; **TEQ** *a*, *b* for *a* XOR *b*.
- Any ordinary instruction with an **S** suffix (**ADDS**, **SUBS**, **MOVS**, **ANDS**) also updates the flags.
- Instructions without **S** never touch the flags.

7.4 Signed versus unsigned

EXTRA

The same bits mean different things depending on whether you treat them as signed or unsigned, so ARM gives you two families of conditions: **GT/LT/GE/LE** for signed, **HI/LO/HS/LS** for unsigned.

```
LDR r1, =0xFFFFFFFF @ -1 as signed, 4294967295 as unsigned
MOV r2, #1
CMP r1, r2
BLT signed_less      @ taken: -1 < 1 (signed)
BHI unsigned_higher  @ ALSO true: 4294967295 > 1 (unsigned)
```

TIP

Use the signed family for **int** and the unsigned family for **unsigned**, pointers and addresses. Mixing them up is a classic source of bugs.

8 Operators and Expressions

8.1 Arithmetic operators

Addition and subtraction

```
ADD r2, r0, r1      @ r2 = r0 + r1      (flags unchanged)
ADDS r2, r0, r1     @ r2 = r0 + r1      (updates flags)
SUB r2, r0, r1      @ r2 = r0 - r1
```

WATCH OUT

There is no **ADDX** instruction; the flag-setting add is **ADDS**.

Unary minus and reverse subtract

RSB (reverse subtract) computes the second operand minus the first. That makes negation easy:

```
RSB r1, r0, #0      @ r1 = 0 - r0 = -r0  (unary minus)
RSB r1, r0, #1      @ r1 = 1 - r0
```

WATCH OUT

RSB r1, r0, #1 computes **1 - r0**, not **-r0**. To negate a value, subtract it from **zero**.

Multiplication

```
MUL r2, r0, r1      @ r2 = r0 * r1      (flags unchanged)
MLA r3, r0, r1, r2   @ r3 = r0 * r1 + r2 (multiply-accumulate)
```

Division

Division is not a core ALU operation on ARM, so it has two forms depending on the processor.

```
@ older cores: call the runtime helper (dividend in r0, divisor in r1)
BL __aeabi_idiv      @ signed:  r0 = r0 / r1
BL __aeabi_uidiv     @ unsigned: r0 = r0 / r1

@ newer cores (hardware divide instructions)
SDIV r0, r1, r2      @ signed:  r0 = r1 / r2
UDIV r0, r1, r2      @ unsigned: r0 = r1 / r2

MOV r1, #14          @ dividend
MOV r2, #3            @ divisor
SDIV r0, r1, r2       @ r0 = 4 (the quotient)
                     @ the remainder is NOT produced
```

EXTRA

Get the remainder yourself with multiply-and-subtract: **MLS r3, r0, r2, r1** computes **r3 = r1 - r0*r2**, which is **14 - 4*3 = 2**. Tested: **SDIV** gives 4 and **MLS** gives 2. **SDIV/UDIV** exist on ARMv7-A cores with the divide extension and on ARMv7-R; very old chips such as the original Raspberry Pi fall back to **__aeabi_idiv**.

NOTE

Division is not a core ALU operation. Hardware divide may take many cycles.

8.2 Increment and decrement

There is no `++` or `--` instruction in ARM. Use `ADD` and `SUB`:

```
ADD r0, r0, #1      @ r0++
SUB r0, r0, #1      @ r0--
```

8.3 Compound assignment

```
ADD r0, r0, r1      @ a += b
SUB r0, r0, r1      @ a -= b
```

EXTRA

Because ARM instructions have a separate destination, `a += b` is just "destination = first source". You can also fold a shift in: `ADD r1, r1, r2, LSL #2` means `r1 += r2 * 4`.

8.4 Relational operators

Comparisons such as `<`, `>`, `==`, `!=` do **not** produce a value in ARM. They only set the flags; you then branch (or conditionally execute) on those flags.

```
CMP r0, r1          @ if (r0 == r1)
BEQ equal           @ branch if equal

CMP r0, r1          @ if (r0 != r1)
BNE not_equal       @ branch if not equal

CMP r0, r1          @ if (r0 > r1) signed
BGT greater         @ branch if greater
```

Signed comparisons

Branch	Meaning
BGT	Signed greater than (>)
BLT	Signed less than (<)
BGE	Signed greater than or equal (>=)
BLE	Signed less than or equal (<=)

Unsigned comparisons

Branch	Meaning
BHI	Unsigned higher (>)
BLO	Unsigned lower (<)
BHS	Unsigned higher or same (>=)
BLS	Unsigned lower or same (<=)

EXTRA

If you need a 0 or 1 result rather than a jump, use conditional moves:

```

CMP    r0, r1
MOVGT  r2, #1      @ r2 = (r0 > r1) ? 1 : 0
MOVLE  r2, #0

```

8.5 Logical operators

`&&`, `||` and `!` are built from compares and branches.

Logical AND (&&)

```

@ if (a && b) { ... }      a = r0, b = r1
    CMP r0, #0
    BEQ false_case        @ a is 0, so the whole test fails
    CMP r1, #0
    BEQ false_case        @ b is 0, so the whole test fails
    @ ... body when both are non-zero ...
    B   done
false_case:
    @ ... else part ...
done:

```

Logical OR (||)

```

@ if (a || b) { ... }
    CMP r0, #0
    BNE true_case         @ a is non-zero: stop early
    CMP r1, #0
    BNE true_case
    B   false_case        @ both were zero
true_case:
    @ ...
false_case:

```

Logical NOT (!)

```

@ c = !a                  a = r0, c = r1
    CMP    r0, #0
    MOVEQ  r1, #1         @ a was 0  ->  c = 1
    MOVNE  r1, #0         @ a was non-zero ->  c = 0

```

EXTRA

Notice the **short-circuit** behaviour: as in C, `&&` stops at the first false operand and `||` at the first true one, because each test branches away immediately.

8.6 Bitwise operators

```

AND r2, r0, r1      @ r2 = r0 & r1
ORR r2, r0, r1      @ r2 = r0 | r1
EOR r2, r0, r1      @ r2 = r0 ^ r1
MVN r1, r0          @ r1 = ~r0  (bitwise NOT, "move negation")
BIC r2, r0, r1      @ r2 = r0 & ~r1  (bit clear)

```

TST performs an AND but **does not store the result**; it only sets the flags. The Z flag is set when the tested bits are all zero.

```

TST r0, #1          @ is bit 0 set?
BNE odd             @ Z == 0 -> bit was 1 -> number is odd
BEQ even            @ Z == 1 -> bit was 0 -> number is even

```

WATCH OUT

In C, **!** is logical NOT; the bit-flipping operator is **~**, which is what **MVN** does.

EXTRA

Handy bit recipes, all tested:

```

ORR r1, r1, #(1 << 3) @ set    bit 3
BIC r1, r1, #(1 << 3) @ clear  bit 3
EOR r1, r1, #(1 << 3) @ toggle bit 3

```

8.7 Shift operators

Instruction	C operator	Meaning
LSL r1, r0, #2	<<	Logical shift left (fills with 0)
ASL r1, r0, #2	<<	Arithmetic shift left (same as LSL)
LSR r1, r0, #2	>> (unsigned)	Logical shift right (fills with 0)
ASR r1, r0, #2	>> (signed)	Arithmetic shift right (keeps the sign bit)
ROR r1, r0, #2	rotate	Rotate right (bits wrap around)

WATCH OUT

ASL is just an alias of **LSL**: left shifts are identical for signed and unsigned values. The signed/unsigned distinction only matters for **right** shifts.

```

MOV r0, #-8
ASR r1, r0, #1      @ r1 = -4 (sign preserved)
LSR r2, r0, #1      @ r2 = 0x7FFFFFFC (zero filled, now a huge positive)

```

TIP

Shifting left by n multiplies by 2^n and shifting right divides by 2^n . **LSL #2** is multiply-by-4, which is exactly the scaling used for word arrays.

8.8 Conditional execution

Almost any ARM instruction can be made conditional by adding a condition suffix. This often removes branches entirely:

```

@ r3 = max(r0, r1)
CMP    r0, r1          @ compute r0 - r1
MOVGT  r3, r0          @ if r0 > r1: r3 = r0
MOVLE  r3, r1          @ if r0 <= r1: r3 = r1

```

EXTRA

Absolute value in two instructions: **CMP r0, #0** then **RSBLT r0, r0, #0**.

8.9 Creating branches

Instruction	Meaning
B label	Branch (jump) to label, no condition
BLE label	Branch if the previous comparison was signed less-than-or-equal
BGT label	Branch if the previous comparison was signed greater-than
BL label	Branch with link: save the return address in lr, then jump (function call)
BX lr	Branch to the address in lr (return from a function)

```
loop:
    B loop                @ infinite loop
```

WATCH OUT

LE is not a flag; it is a condition built from the Z, N and V flags ($Z == 1$ or $N != V$).

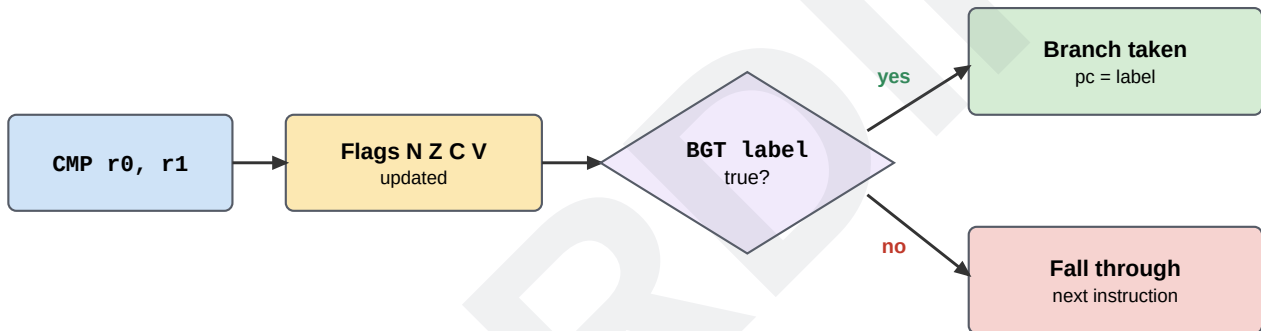


Figure 8.1: A compare sets the flags; a conditional branch tests them and either jumps or falls through.

9 Control-Flow Patterns

9.1 Reading CMP like C

C code	ARM
if (a > b)	CMP a, b + BGT
if (a < b)	CMP a, b + BLT
if (a >= b)	CMP a, b + BGE
if (a <= b)	CMP a, b + BLE
if (a == b)	CMP a, b + BEQ
if (a != b)	CMP a, b + BNE

9.2 if / else

TIP

When translating `if (cond) { A } else { B }`, branch on the **opposite** condition to skip block A. That keeps the true-case code falling straight through.

```
@ if (a > b) x = 1; else x = 2;      a = r0, b = r1, x = r2
    CMP    r0, r1
    BLE    else_part                @ opposite of >
    MOV    r2, #1
    B      end_if
else_part:
    MOV    r2, #2
end_if:
```

9.3 while loop

```
@ while (i < n) { ...; i++; }      i = r0, n = r1
loop:
    CMP    r0, r1
    BGE    done                    @ opposite of <
    @ ... loop body ...
    ADD    r0, r0, #1
    B      loop
done:
```

9.4 for loop

A `for` loop is a `while` loop with the initialiser before and the increment at the bottom. Chapter 10 shows a complete one that sums an array.

9.5 do-while loop

```
@ do { ...; n--; } while (n != 0);
loop:
    @ ... body ...
    SUBS r0, r0, #1      @ decrement AND set flags
    BNE  loop           @ repeat while the result was not zero
```

EXTRA

SUBS then **BNE** is the idiomatic counted loop on ARM because the subtract already sets the flags, so no separate **CMP** is needed.

10 Arrays and Scaling

10.1 Looping over an array

The C loop on the left becomes the assembly loop on the right. Here **r0** is the index **i**, **r1** is the length **n**, **r2** the running **sum**, and **r3** the base address of **arr**.

```
/* C */
sum = 0;
for (i = 0; i < n; i++)
    sum += arr[i];

MOV r0, #0          @ i = 0
MOV r2, #0          @ sum = 0
LDR r3, =arr        @ base address
loop:
CMP r0, r1          @ i < n ?
BGE end
LDR r4, [r3, r0, LSL #2] @ r4 = *(arr + i*4) = arr[i]
ADD r2, r2, r4      @ sum += arr[i]
ADD r0, r0, #1      @ i++
B loop
end:
```

Complete, runnable program

```
.syntax unified
.section .data
arr: .word 10, 20, 30, 40, 50

.section .text
.globl _start
_start:
MOV r0, #0          @ i
MOV r2, #0          @ sum
LDR r3, =arr
MOV r1, #5          @ n
loop:
CMP r0, r1
BGE end
LDR r4, [r3, r0, LSL #2]
ADD r2, r2, r4
ADD r0, r0, #1
B loop
end:
MOV r0, r2          @ exit status = sum
MOV r7, #1
SWI 0              @ echo $? prints 150
```

10.2 Scaling: accessing array elements

ARM does scaling using a **shifted register offset** inside the load or store instruction:

```
LDR Rd, [Rn, Rm, LSL #shift]    @ Rd = *(Rn + (Rm << shift))
```

Part	Meaning
Rn	Base address of the array

Part	Meaning
Rm	Index register
#shift	log2 of the element size, so the index becomes a byte offset

Element size	Directive	Shift	Address formula
1 byte	.byte	0	base + i
2 bytes	.hword	1	base + (i << 1)
4 bytes	.word	2	base + (i << 2)
8 bytes	.quad	3	base + (i << 3)

int nums[4] = {10, 20, 30, 40}; (each element = 4 bytes)



`mov r1, #2 ; ldr r0, [r3, r1, LSL #2] -> address = r3 + (2 << 2) = base + 8 -> r0 = 30`

Figure 10.1: Element 2 of a word array lives $2 \times 4 = 8$ bytes after the base address.

Because **address = base + (i * 4)** and **i * 4 = i << 2**, the shift performs the multiplication for free. With a base of 0x20000000, index 1 gives 0x20000004 and index 2 gives 0x20000008.

Byte array

```
bytes: .byte 10, 20, 30, 40
LDR r3, =bytes
MOV r1, #2
LDRB r0, [r3, r1]           @ r0 = bytes[2] = 30
```

Halfword array

```
hws: .hword 100, 200, 300, 400
LDR r3, =hws
MOV r1, #2
ADD r3, r3, r1, LSL #1      @ r3 = &hws[2]
LDRH r0, [r3]               @ r0 = 300
```

Word array

```
nums: .word 10, 20, 30, 40
LDR r3, =nums
MOV r1, #2
LDR r0, [r3, r1, LSL #2]    @ r0 = nums[2] = 30
```

Doubleword (64-bit) array

```
nums: .quad 1, 2, 3
      LDR r3, =nums
      MOV r1, #2
      ADD r3, r3, r1, LSL #3      @ r3 = &nums[2]
      LDR r0, [r3]               @ r0 = low 32 bits
      LDR r2, [r3, #4]           @ r2 = high 32 bits (little-endian)
```

WATCH OUT

The element directive and the shift must match: **.byte** with no shift, **.hword** with **LSL #1**, **.word** with **LSL #2**. Indexing a word array with a plain byte offset reads a byte inside `nums[0]`, not `nums[2]`.

WATCH OUT

LDRH, **LDRSH**, **LDRSB** and **LDRD** do **not** accept a shifted register index, so **LDRH r0, [r3, r1, LSL #1]** fails to assemble with "instruction does not accept scaled register index". Add the scaled index to the base first, then load. Plain **LDR** and **LDRB** do accept it. **LDRD** can load both halves at once into an even/odd register pair, for example **LDRD r4, r5, [r3]**.

TIP

Another way to walk an array is to advance a pointer with post-indexing: **LDR r4, [r3], #4** loads the element and bumps `r3` to the next one in a single instruction.

11 Functions and the Stack Frame

11.1 A simple function

```
/* C */
int add(int a, int b) { return a + b; }

.global add          @ make the function visible to the linker
.type   add, %function @ tell the assembler this label is a function
add:
    ADD r0, r0, r1    @ r0 = a + b
    BX lr             @ return
```

To call it, put the arguments in r0 and r1, then use **BL**. **BL** copies the address of the next instruction into lr and jumps to the function; **BX lr** later copies lr back into the program counter.

```
/* C */ add(5, 7);

MOV r0, #5
MOV r1, #7
BL add          @ after return: r0 = 12
```

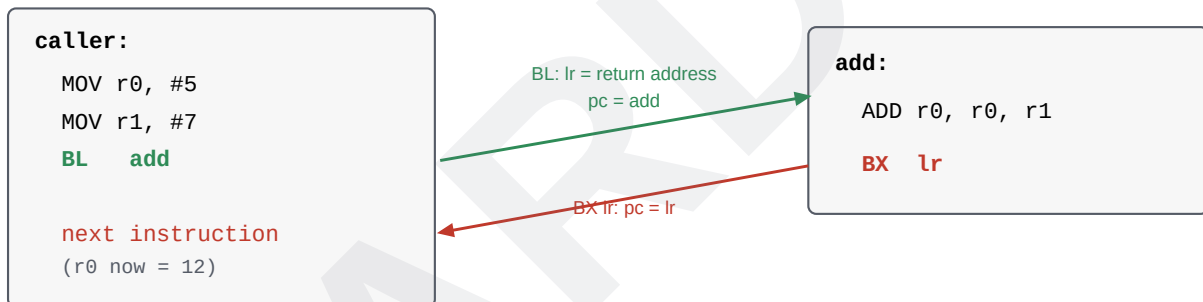


Figure 11.1: BL saves the return address in lr; BX lr jumps back to it.

11.2 The stack and PUSH / POP

The stack is a region of memory used for saved registers, local variables and extra arguments. On ARM it is a **full descending** stack: it grows toward lower addresses and **sp** points at the most recently pushed item.

NOTE

PUSH {r4, lr} saves lr and then r4 on the stack. You can push many registers in one go, for example **PUSH {r4-r11, lr}**.

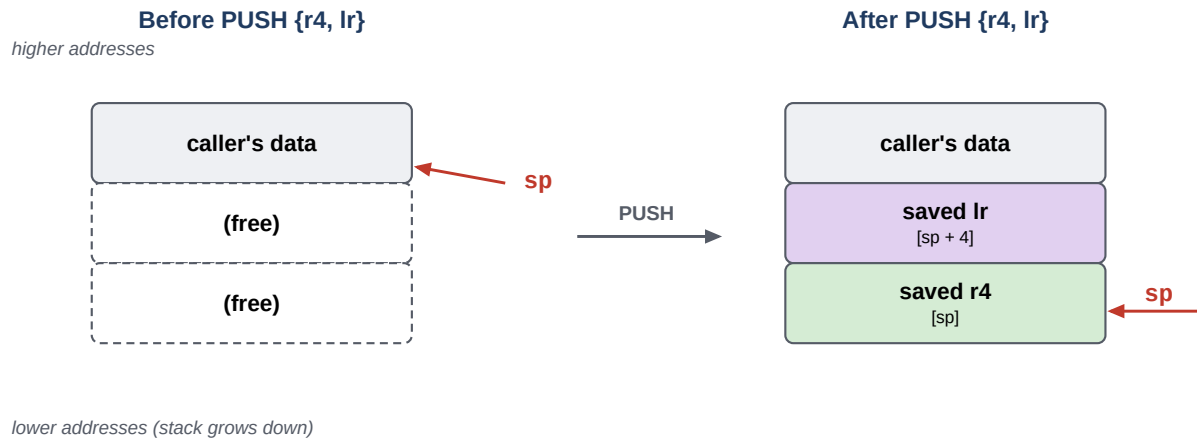


Figure 11.2: `PUSH {r4, lr}` moves `sp` down by 8 bytes; the lowest-numbered register goes to the lowest address.

- `PUSH {r4, lr}` is shorthand for `STMDB sp!, {r4, lr}`.
- `POP {r4, pc}` is shorthand for `LDMIA sp!, {r4, pc}`: it restores `r4` and loads the saved return address directly into the program counter, returning in a single instruction.

WATCH OUT

Never push `sp` or `pc` in a function prologue. Save only what you use: the callee-saved registers you will modify (`r4-r11`) plus `lr` if you call other functions.

11.3 A function with a stack frame

```
/* C */
int inc(int a) { int x; x = a + 1; return x; }

.global inc
inc:
    PUSH {r4, lr}      @ save r4 (we use it) and lr
    ADD  r4, r0, #1     @ x = a + 1
    MOV  r0, r4         @ return value
    POP  {r4, pc}       @ restore r4 and return
```

NOTE

`POP {r4, lr}` followed by `BX lr` also works, but `POP {r4, pc}` does both jobs in one step.

11.4 Local variables on the stack

```
/* C */
int sum(int a, int b) { int temp; temp = a + b; return temp; }

.global sum
sum:
    PUSH {lr}
    SUB  sp, sp, #4     @ allocate 4 bytes for temp
    ADD  r2, r0, r1
    STR  r2, [sp]       @ temp = a + b
    LDR  r0, [sp]       @ return temp
    ADD  sp, sp, #4     @ free the local variable
    POP  {lr}
    BX   lr
```

EXTRA

`sum` is a **leaf function** (it calls nothing), so it does not really need to save `lr` or use the stack at all: `ADD r0, r0, r1` followed by `BX lr` is enough. The longer version is shown to illustrate how locals are allocated. Note that `PUSH {lr}` plus 4 bytes of local space totals 8 bytes, which keeps `sp` 8-byte aligned.

TIP

Always release exactly what you allocated, in reverse order. Whatever you did to `sp` on the way in must be undone on the way out.

11.5 Nested functions

```
/* C */
int f(int a, int b) { return add(a, b) + 1; }

.global f
f:
    PUSH {r4, lr}      @ lr MUST be saved because BL will overwrite it
    BL  add             @ r0 = add(a, b) (a, b are already in r0, r1)
    ADD r0, r0, #1
    POP {r4, pc}
```

`a` and `b` arrive in `r0` and `r1`, which are exactly where `add` expects them, so no moves are needed. Pushing two registers (`r4` and `lr`) also keeps the stack 8-byte aligned, as the AAPCS requires.

11.6 More than four arguments

```
/* C */
int f(int a, int b, int c, int d, int e);
```

The first four arguments go in `r0-r3`. The fifth is passed on the stack, at `[sp]` at the moment of the call.

Caller

```
MOV r0, #1
MOV r1, #2
MOV r2, #3
MOV r3, #4
MOV ip, #5      @ the fifth argument (ip = r12, scratch)
SUB sp, sp, #8   @ reserve space (8 keeps sp aligned)
STR ip, [sp]     @ e goes at [sp]
BL  f
ADD sp, sp, #8   @ remove the argument area
```

Callee

```
f:
    PUSH {r4, lr}      @ sp has now moved down by 8 bytes
    LDR r4, [sp, #8]    @ e: skip over saved r4 and lr
    ADD r0, r0, r1
    ADD r0, r0, r2
    ADD r0, r0, r3
    ADD r0, r0, r4      @ r0 = a + b + c + d + e = 15
    POP {r4, pc}
```

WATCH OUT

Once the callee pushes registers, the fifth argument is no longer at `[sp]`: it sits above the saved registers, here at `[sp, #8]`.

11.7 Recursion

EXTRA

Recursion shows why saving `lr` and `r4` matters. Each call keeps its own `n` safely on the stack.

```
@ int fact(int n) { if (n <= 1) return 1; return n * fact(n - 1); }
fact:
    PUSH {r4, lr}
    CMP r0, #1
    BLE base
    MOV r4, r0      @ remember n
    SUB r0, r0, #1
    BL fact         @ r0 = fact(n - 1)
    MUL r0, r4, r0   @ n * fact(n - 1)
    POP {r4, pc}
base:
    MOV r0, #1
    POP {r4, pc}
```

Called with `r0 = 5` this returns 120, confirmed by running it.

12 Worked Examples

EXTRA

Every program in this chapter was assembled and run. Each one ends by passing its answer to `exit`, so you can read it with `echo $?`.

12.1 Greatest common divisor

```
@ int gcd(int a, int b)    (a, b > 0)    Euclid by subtraction
gcd:
gcd_loop:
    CMP    r0, r1
    BEQ    gcd_done
    SUBGT  r0, r0, r1        @ if a > b:  a -= b
    SUBLT  r1, r1, r0        @ if a < b:  b -= a
    B      gcd_loop
gcd_done:
    BX     lr

_start:
    MOV    r0, #48
    MOV    r1, #18
    BL     gcd                @ r0 = 6
    MOV    r7, #1
    SWI    0                  @ exit status 6
```

Neither `SUBGT` nor `SUBLT` has an `S` suffix, so they do not change the flags. The flags from the single `CMP` therefore remain valid for both lines.

12.2 Fibonacci

```
@ int fib(int n)    fib(0)=0, fib(1)=1
fib:
    MOV    r1, #0        @ a = 0
    MOV    r2, #1        @ b = 1
fib_loop:
    CMP    r0, #0
    BEQ    fib_end
    ADD    r3, r1, r2      @ t = a + b
    MOV    r1, r2          @ a = b
    MOV    r2, r3          @ b = t
    SUB    r0, r0, #1      @ n--
    B      fib_loop
fib_end:
    MOV    r0, r1          @ return a
    BX     lr              @ fib(10) = 55
```

12.3 String length

```
@ int my_strlen(const char *s)
my_strlen:
    MOV    r1, #0                @ count = 0
next:
    LDRB   r2, [r0, r1]          @ load s[count]
    CMP    r2, #0                @ end of string?
    BEQ    done
    ADD    r1, r1, #1
    B      next
done:
    MOV    r0, r1                @ return count
    BX     lr                    @ "assembly" -> 8
```

12.4 Counting set bits

```
@ count the 1 bits in r0, result in r1
    MOV    r1, #0
pc_loop:
    CMP    r0, #0
    BEQ    pc_end
    TST    r0, #1                @ lowest bit set?
    ADDNE  r1, r1, #1            @ yes: count it
    LSR    r0, r0, #1            @ shift the next bit into place
    B      pc_loop
pc_end:                                     @ 0xF0F1 -> 9 ones
```

12.5 Find the maximum of an array

```
@ r0 = pointer to words, r1 = count; returns max in r0
array_max:
    LDR    r2, [r0]              @ best = first element
    MOV    r3, #1                @ i = 1
am_loop:
    CMP    r3, r1
    BGE    am_done
    LDR    r4, [r0, r3, LSL #2]
    CMP    r4, r2
    MOVGT  r2, r4                @ new maximum
    ADD    r3, r3, #1
    B      am_loop
am_done:
    MOV    r0, r2
    BX     lr
```

NOTE

This last example uses `r4`, a callee-saved register, so a polished version would begin with `PUSH {r4}` and end with `POP {r4}` before `BX lr`. Try adding them as an exercise.

13 Quick Reference and Practice

13.1 Instruction cheat sheet

Instruction	Meaning	Example
MOV / MVN	Copy / copy inverted	MOV r0, #5
ADD / ADC	Add / add with carry	ADD r2, r0, r1
SUB / RSB	Subtract / reverse subtract	RSB r1, r0, #0
MUL / MLA / MLS	Multiply / +accumulate / -accumulate	MLA r3, r0, r1, r2
SDIV / UDIV	Hardware divide	SDIV r0, r1, r2
AND / ORR / EOR / BIC	Bitwise and / or / xor / clear	BIC r0, r0, #8
LSL / LSR / ASR / ROR	Shifts and rotate	LSR r0, r0, #1
CMP / CMN / TST / TEQ	Compare or test; set flags only	CMP r0, r1
LDR / STR	Load / store word	LDR r1, [r0]
LDRB / STRB	Load / store byte	LDRB r1, [r0]
LDRH / STRH	Load / store halfword	LDRH r1, [r0]
LDRSB / LDRSH	Load with sign extension	LDRSB r1, [r0]
PUSH / POP	Save / restore registers on the stack	PUSH {r4, lr}
B / BL / BX	Jump / call / return	BX lr
SWI / SVC	Make a system call	SWI 0

13.2 Function checklist

- Arguments in r0-r3 (extra ones on the stack); result in r0.
- If you call another function, push lr and pop it (or pop straight into pc).
- If you use r4-r11, push them on entry and pop them on exit.
- Keep sp 8-byte aligned at every call.
- Return with **BX lr**, or **POP {..., pc}** if you pushed lr.

13.3 Practice exercises

EXTRA

Try these with qemu or on a Raspberry Pi. Check each answer with the exit status.

- **Sum of 1 to n.** Write a loop that adds $1 + 2 + \dots + 10$ and exits with 55.
- **Swap.** Write `swap(int *a, int *b)` that exchanges two words in memory using LDR and STR.
- **Even or odd.** Use **TST** to return 1 for even numbers and 0 for odd ones.
- **Absolute value.** Return $|n|$ using **CMP** and **RSBLT**, with no branches.
- **Power.** Compute 2 to the power n using **LSL**, then try 3 to the power n with **MUL** in a loop.

- **Reverse a string.** Reverse a `.asciz` string in place using two pointers and `LDRB/STRB`.
- **Min and max.** Extend the array-maximum example to also return the minimum.
- **Five arguments.** Write `f(a, b, c, d, e)` that returns `a*b + c*d + e` and call it from `_start`.

13.4 Where to go next

- Compile a small C function with `arm-linux-gnueabi-gcc -S -O1` and read what the compiler produced.
- Look at Thumb and Thumb-2, the compact instruction sets used by Cortex-M microcontrollers.
- Study the ARM64 (AArch64) instruction set, which uses x0-x30 and a cleaner encoding.

Good luck, and happy hacking. — HARDIK

Connect with me on LinkedIn: www.linkedin.com/in/hardikhardik