

ARM Mazidi Solutions — All Sections

ARM Mazidi Solutions — All Sections

Chapter 1

Section 1.1

Section 1.2

Chapter 2

Section 2.1

Section 2.2

Section 2.3

Section 2.4

Section 2.5

Section 2.7

Section 2.8

Section 2.9

Section 2.10

Chapter 3

Section 3.1

Section 3.2

Section 3.3

Section 3.5

Chapter 4

Section 4.1

Section 4.2

Section 4.3

Section 4.4

Chapter 5

Section 5.1

Section 5.2

Section 5.3

Chapter 6

Section 6.1

Section 6.2

Section 6.3

Section 6.4

Section 6.5

Chapter 7

Section 7.1

Section 7.2

ARM Mazidi Solutions — All Sections

Chapter 1

Section 1.1

Problems are paraphrased to respect copyright. See *Mazidi*, **Chapter 1 §1.1** (“Introduction to Microcontrollers”), PDF ~pp. 15–17.

1) True or False. A general-purpose microprocessor has on-chip ROM.

Answer: False.

Why: General-purpose microprocessors (e.g., x86) provide only the CPU and **require external ROM/RAM and I/O**. (*See §1.1, PDF p. ~15.*)

2) True or False. Generally, a microcontroller has on-chip ROM.

Answer: True.

Why: A microcontroller integrates **CPU + program ROM/Flash + RAM + I/O** on a **single chip**. (*§1.1, p. ~15–16.*)

3) True or False. A microcontroller has on-chip I/O ports.

Answer: True.

Why: On-chip **GPIO and peripheral interfaces** (timers/serial, etc.) are part of the MCU integration. (*§1.1, p. ~16.*)

4) True or False. A microcontroller has a fixed amount of RAM on the chip.

Answer: True.

Why: The MCU’s **on-chip RAM size is fixed** for a given device family/part number. (*§1.1, p. ~15–16.*)

5) What components are usually put together with the microcontroller onto a single chip?

Answer: CPU, program ROM/Flash, data RAM, I/O ports, and typically **timers/counters** and **serial peripherals** (UART/SPI/I²C); many devices also integrate **ADC/PWM/interrupt controller**. (*§1.1, p. ~16.*)

6) Intel’s Pentium chips used in Windows PCs need external _____ and _____ chips to store data and code.

Answer: RAM and ROM (BIOS/Flash).

Why: The Pentium is a **microprocessor**—it relies on **external memory** for both data and program storage. (*§1.1, p. ~15.*)

7) List three embedded products attached to a PC.

Example answers: Keyboard, mouse, printer.

(Other valid examples: scanner, webcam, external modem, game controller.) (*General §1.1 examples.*)

8) Why would someone want to use an x86 as an embedded processor?

Answer (concise): To leverage **PC compatibility and ecosystem**—abundant **development tools**, existing **software/OS support**, and **familiarity/performance** for certain embedded applications. (*§1.1 context.*)

9) Give the name and the manufacturer of some widely used 8-bit microcontrollers.

Answer (any three):

- **8051 family** — originally **Intel**; produced by many vendors (e.g., **NXP**, **Silicon Labs**, **Atmel/Microchip**).
 - **PIC** — **Microchip Technology**.
 - **AVR** — originally **Atmel** (now **Microchip**).
- (Also acceptable: **Zilog Z8**, **Motorola/Freescale 68HC05/08**.) (*Historical overview in §1.1.*)

10) In Question 9, which one has the most manufacture sources?

Answer: 8051 family.

Why: It has been **second-sourced by many manufacturers** for decades. (§1.1.)

11) In a battery-based embedded product, what is the most important factor in choosing a microcontroller?

Answer: Power consumption (low-power operation).

Why: Directly impacts **battery life** (sleep/active current, clocking options). (§1.1 design considerations.)

12) In an embedded controller with on-chip ROM, why does the size of the ROM matter?

Answer: It limits the maximum program size (firmware features, tables, libraries) and **affects cost/part selection**. (§1.1.)

13) In choosing a microcontroller, how important is it to have multiple sources for that chip?

Answer: Important.

Why: Multiple sources reduce **supply-risk**, improve **price/lead-time**, and ensure **long-term availability** and drop-in replacements (as with many **8051** parts). (§1.1.)

14) What does the term “third-party support” mean?

Answer: Availability of **tools and resources from companies other than the MCU vendor**—e.g., **compilers, assemblers, debuggers, IDEs, RTOS, programmers, evaluation boards, libraries**. Strong third-party support shortens development time. (§1.1.)

Section 1.2

Problems are paraphrased to respect copyright. See *Mazidi*, **Chapter 1 §1.2** (“The ARM Family History”), PDF ~pp. **17–20**.

Quick scan (True/False)

15 — n/a • 16 **False** • 17 **True** (with context) • 18 **True** • 19 **False** • 20 **True** •
21 **False** • 22 **False** • 23 **True** • 24 **False** • 25 **False** • 26 **False**

15) What does ARM stand for?

Answer: Originally **Acorn RISC Machine**; later **Advanced RISC Machines**. Today the company is **Arm Ltd.**

Why: The project began at Acorn Computers and evolved into a standalone IP company. (§1.2)

16) True or False. In ARM, architectures have the same names as families.

Answer: **False.**

Why: **Architecture versions** are named **ARMv4/v5/v6/v7/v8**, while **product families** are **ARM7/ARM9, Cortex-A/R/M**, etc. Names are related but **not the same**. (§1.2)

17) True or False. In 1990s, ARM was widely used in the microprocessor world.

Answer: **True** (in embedded/mobile).

Why: ARM became the **dominant 32-bit RISC** in handhelds/embedded devices (PDAs, phones), even though desktop PCs remained x86. (§1.2)

18) True or False. ARM is widely used in Apple products, like iPhone and iPod.

Answer: **True.**

Why: Apple’s mobile devices use ARM-based SoCs. (§1.2)

19) True or False. Currently the Microsoft Windows does not support ARM products.

Answer: **False.**

Why: Microsoft has supported ARM in **Windows CE/Embedded, Windows Phone, and modern Windows on ARM**. (§1.2 context)

20) True or False. All ARM chips have standard instructions.

Answer: **True** (core ISA is standardized per architecture version).

Why: The **instruction set architecture (ARM/Thumb/Thumb-2, plus optional extensions like VFP/NEON)** is defined by Arm; implementations conform to the relevant spec. (§1.2)

21) True or False. All ARM chips have standard peripherals.

Answer: **False.**

Why: **Peripherals are vendor-specific** (UART, timers, GPIO mapping, ADC, etc.), so registers and drivers differ across manufacturers/families. (§1.2)

22) True or False. The ARM corporation also manufactures the ARM chip.

Answer: **False.**

Why: Arm is an **IP licensor**; **licensees** (e.g., ST, NXP, TI, Samsung, Microchip) manufacture the chips. (§1.2)

23) True or False. The ARM IP must be licensed from ARM corp.

Answer: True.

Why: Companies **license CPU cores/architectures** (soft/hard IP) from Arm to build their SoCs/MCUs. (§1.2)

24) True or False. A serial-communication program written for a TI ARM chip should run without any modification on a Freescale ARM chip.

Answer: False.

Why: While the **core ISA** is compatible, **peripheral registers/clock trees/interrupts** are different; **UART drivers** and init code are **not portable** without adaptation. (§1.2)

25) True or False. An Assembly program written for one family of ARM Cortex chip can execute on any other Cortex ARM chip.

Answer: False (not universally).

Why: Portability depends on **ISA mode** and **architecture level** (e.g., **Cortex-M** is **Thumb-2-only**, while **Cortex-A** may use **ARM/AArch32/AArch64**). Also, **system control** and memory maps differ. Pure, ISA-only code might port, but **in general modifications are required**. (§1.2)

26) True or False. At the present time, ARM has just one manufacturer.

Answer: False.

Why: There are **many** Arm licensees (ST, NXP, TI, Microchip/Atmel, Renesas, Samsung, etc.). (§1.2)

27) What is the difference between the ARM products of different manufacturers?

Answer (concise):

- **Same core architecture**, but **different peripherals** (timers, UART, I²C/SPI, ADC, PWM), **memory sizes**, **clock trees**, **packages**, **power modes**, and **toolchain support**.
 - Result: **software drivers**, **BSPs**, and **startup code** are **vendor/family specific** even when the CPU core is the same. (§1.2)
-

28) Name some 32-bit microcontrollers.

Examples (any correct subset):

- **STM32** (STMicroelectronics, Cortex-M)
 - **NXP LPC / i.MX RT** (Cortex-M)
 - **TI Tiva-C / MSP432** (Cortex-M)
 - **Microchip SAM** (ex-Atmel, Cortex-M)
 - **Renesas RA** (Cortex-M)
 - **PIC32** (Microchip, MIPS-based — still 32-bit MCU)
- (§1.2 examples/history)
-

29) What is Intel's challenge in decreasing the power consumption of the x86?

Answer: Maintaining **backward compatibility** with the complex legacy **x86 ISA** (variable-length decode and large micro-architectural support) imposes **power/complexity overheads**. Reducing power while keeping performance and compatibility is the key challenge compared to lean RISC designs like ARM. (§1.2 discussion)

Chapter 2

Section 2.1

Problems are paraphrased to respect copyright. For theory and examples, see **Mazidi, Ch. 2 §2.1**.

1) ARM is a(n) _____-bit microprocessor.

Answer: 32-bit.

Why: Classic ARM (AArch32) uses a 32-bit programming model and word size here.

2) The general-purpose registers are _____ bits wide.

Answer: 32 bits.

Why: Registers **R0–R15** are each 32-bit wide in the AArch32 model.

3) The value in `MOV R2, #value` is _____ bits wide.

Answer: 8 bits (immediate field as taught in this section).

Why: The introductory encoding uses an **8-bit literal** (later chapters explain rotations/literal pools for larger constants).

4) The largest number that an ARM GPR can have is _____ in hex.

Answer: 0xFFFFFFFF

Why: Unsigned maximum for a 32-bit register.

5) What is the result of the following code and where is it kept?

```
MOV    R2, #0x15
MOV    R1, #0x13
ADD    R2, R1, R2
```

Answer: R2 = 0x28 (40 decimal), kept in **R2**.

Why: `ADD Rd, Rn, Op2` \rightarrow `R2 = R1 + R2 = 0x13 + 0x15 = 0x28`.

6) Which of the following is/are illegal?

(a) `MOV R2, #0x50000` (b) `MOV R2, #0x50` (c) `MOV R1, #0x00`
(d) `MOV R1, 255` (e) `MOV R17, #25` (f) `MOV R23, #0xF5` (g) `MOV 123, 0x50`

Answer: (a), (d), (e), (f), (g) are illegal; (b) and (c) are legal.

Why (brief):

- (a) exceeds the simple 8-bit immediate taught here.
- (b) legal (8-bit immediate).
- (c) legal (zero is allowed).
- (d) missing # for immediate.
- (e) **R17** does not exist (valid GPRs are **R0–R15**).
- (f) **R23** does not exist.
- (g) destination must be a **register**, not an immediate.

7) Which of the following is/are illegal?

(a) `ADD R2, #20, R1` (b) `ADD R1, R1, R2` (c) `ADD R5, R16, R3`

Answer: (a) and (c) are illegal; (b) is legal.

Why:

- (a) Format is `ADD Rd,Rn,Operand2`; the **immediate** can only be `Operand2`, not `Rn`.
 - (b) Valid three-operand form `Rd=R1, Rn=R1, Rm=R2`.
 - (c) **R16** is outside the GPR range (only **R0–R15**).
-

8) What is the result of the following code and where is it kept?

```
MOV    R9,#0x25
ADD    R8,R9,#0x1F
```

Answer: R8 = 0x44 (68 decimal), kept in **R8**.

Why: $0x25 + 0x1F = 0x44$.

9) What is the result of the following code and where is it kept?

```
MOV    R1,#0x15
ADD    R6,R1,#0xEA
```

Answer: R6 = 0xFF (255 decimal), kept in **R6**.

Why: $0x15 + 0xEA = 0xFF$.

10) True or False. We have 32 general-purpose registers in the ARM.

Answer: False.

Why: The classic programmer's model exposes **16 architected registers** (R0–R15); some are special-purpose (`SP=R13`, `LR=R14`, `PC=R15`). Some modes bank a subset, but there are **not 32 GPRs**.

Notes for learners

- Remember the **three-operand** form: `ADD Rd,Rn,Operand2`.
- **GPR range** is **R0–R15**; higher numbers like `R16`, `R23` are invalid.
- The immediate in `MOV` is introduced as **8-bit** here; later you'll learn techniques for forming larger constants.

Section 2.2

Problems are paraphrased to respect copyright. For theory and examples, see **Mazidi, Ch. 2 §2.2**.

11) True or False. R13 and R14 are special function registers.

Answer: True.

Why: In the programmer's model, R13 = SP (stack pointer) and R14 = LR (link register) — both are **special-purpose** registers.

12) True or False. The peripheral registers are mapped to memory space.

Answer: True.

Why: ARM microcontrollers use **memory-mapped I/O**; peripheral registers occupy regions in the **address space**.

13) True or False. The on-chip Flash is the same size in all members of ARM.

Answer: False.

Why: Flash size **varies by device/family** (e.g., 8KB ... MBs).

14) True or False. The on-chip data SRAM is the same size in all members of ARM.

Answer: False.

Why: SRAM size is **device-dependent**.

15) What is the difference between the EEPROM and data SRAM space in the ARM?

Answer: EEPROM is non-volatile (retains contents with power off, slower writes, limited endurance); **SRAM is volatile** (contents lost on power-off, fast read/write).

16) Can we have an ARM chip with no EEPROM?

Answer: Yes.

Why: Many ARM MCUs have **no true EEPROM**; Flash (or emulated EEPROM) is used instead.

17) Can we have an ARM chip with no data RAM?

Answer: No.

Why: Practical execution requires **RAM** for stack/variables.

18) What is the maximum number of bytes that the ARM can access?

Answer: $4\text{GB} = 2^{32} \text{ bytes} = 0x00000000\text{--}0xFFFFFFFF$ (architectural 32-bit address space; specific MCUs may implement less).

19) Find the address of the last location of on-chip Flash for each case (first location = 0).

- (a) 32KB → last = 0x7FFF
- (b) 8KB → last = 0x1FFF
- (c) 64KB → last = 0xFFFF
- (d) 16KB → last = 0x3FFF
- (e) 128KB → last = 0x1FFFF
- (f) 256KB → last = 0x3FFFF

Reasoning: last address = size – 1 (bytes).

20) Show the lowest and highest values (in hex) that the ARM program counter can take.

Answer: Lowest = 0x00000000, Highest = 0xFFFFFFFF.

Note: Alignment/state bits may constrain actual fetch addresses, but the architectural range is as above.

21) A given ARM has 0x7FFF as the last location of its on-chip ROM. What is the size?

Answer: 0x8000 bytes = 32KB.

Why: Size = last - first + 1 = 0x7FFF - 0x0000 + 1.

22) Repeat Question 21 for 0x3FFF.

Answer: 0x4000 bytes = 16KB.

23) Find the on-chip program memory size (in KB) for these address ranges (inclusive):

- (a) 0x0000–0x1FFF → size = 0x2000 = 8KB
 - (b) 0x0000–0x3FFF → size = 0x4000 = 16KB
 - (c) 0x0000–0x7FFF → size = 0x8000 = 32KB
 - (d) 0x0000–0xFFFF → size = 0x10000 = 64KB
 - (e) 0x0000–0x1FFFF → size = 0x20000 = 128KB
 - (f) 0x0000–0x3FFFF → size = 0x40000 = 256KB
-

24) Find the on-chip program memory size (in KB) for these address ranges (inclusive):

- (a) 0x000000–0xFFFFFFFF → size = 0x1000000 = 16MB = 16384KB
 - (b) 0x000000–0x7FFFFF → size = 0x800000 = 8MB = 8192KB
 - (c) 0x000000–0x3FFFFF → size = 0x400000 = 4MB = 4096KB
 - (d) 0x000000–0x1FFFFF → size = 0x200000 = 2MB = 2048KB
 - (e) 0x000000–0x0FFFFF → size = 0x100000 = 1MB = 1024KB
 - (f) 0x000000–0x07FFFF → size = 0x80000 = 81920 bytes = 80KB
-

Notes for learners

- **Inclusive ranges:** If the first address is 0, the last address is (size - 1).
- Use powers of two: 1KB = 1024 bytes, 1MB = 1024KB.
- Program counter range shown is the architectural 32-bit span; concrete devices map only a subset.

Section 2.3

Problems are paraphrased to respect copyright. For theory and examples, see **Mazidi, Ch. 2 §2.3**.

25) Show a simple code to store values 0x30 and 0x97 into locations 0x20000015 and 0x20000016, respectively.

Approach: Use **byte stores (STRB)** because the addresses are **unaligned**. Load the base address with `LDR r0,=imm`.

```

        AREA    |.text|, CODE, READONLY
        EXPORT  _start
        THUMB

_start:
        LDR     r0, =0x20000015    ; first byte address
        MOVS    r1, #0x30
        STRB    r1, [r0]           ; [0x20000015] = 0x30

        ADDS    r0, r0, #1         ; next byte address 0x20000016
        MOVS    r1, #0x97
        STRB    r1, [r0]           ; [0x20000016] = 0x97

        B       .
        END

```

Explanation: STRB writes a single byte; no alignment issues.

26) Show a simple code to load the value 0x55 into locations 0x20000030–0x20000038.

Approach: Fill a **range of 9 bytes** starting at 0x20000030 with 0x55 via a loop.

```

        AREA    |.text|, CODE, READONLY
        EXPORT  _start
        THUMB

_start:
        LDR     r0, =0x20000030    ; start address
        MOVS    r1, #0x55          ; byte to store
        MOVS    r2, #9             ; count: 0x30..0x38 inclusive

fill155:
        STRB    r1, [r0]           ; *r0 = 0x55
        ADDS    r0, r0, #1         ; next byte
        SUBS    r2, r2, #1
        BNE     fill155

        B       .
        END

```

Explanation: Because the range is byte-wise and inclusive, $\text{count} = \text{last} - \text{first} + 1 = 0x38 - 0x30 + 1 = 9$.

27) True or False. We cannot load immediate values into the data SRAM directly.

Answer: True (in the strict instruction sense).

Why: There is **no store-immediate** form. You first load the immediate into a **register** (e.g., `MOVS r1, #imm`) and then **store** it with `STR/STRB` to SRAM.

28) Show a simple code to load the value 0x11 into locations 0x20000010–0x20000015.

Approach: Fill a **range of 6 bytes** with 0x11 using `STRB` and a counter.

```

        AREA    |.text|, CODE, READONLY
        EXPORT  _start
        THUMB

_start:
        LDR     r0, =0x20000010    ; start address
        MOVS    r1, #0x11          ; byte to store
        MOVS    r2, #6             ; 0x10..0x15 inclusive

fill11:
        STRB    r1, [r0]
        ADDS    r0, r0, #1
        SUBS    r2, r2, #1
        BNE     fill11

        B       .
        END

```

Explanation: $\text{count} = 0x15 - 0x10 + 1 = 6$.

29) Repeat Problem 28, except load the value into locations 0x20000034–0x2000003C.

Approach: Same loop; **9 bytes** from 0x34 to 0x3C inclusive.

```

        AREA    |.text|, CODE, READONLY
        EXPORT  _start
        THUMB

_start:
        LDR     r0, =0x20000034
        MOVS    r1, #0x11          ; requested value
        MOVS    r2, #9             ; 0x34..0x3C inclusive

fill_more:
        STRB    r1, [r0]
        ADDS    r0, r0, #1
        SUBS    r2, r2, #1
        BNE     fill_more

        B       .
        END

```

Explanation: $\text{count} = 0x3C - 0x34 + 1 = 9$.

Notes for learners

- Use **STRB** for byte writes; **STR** for word (aligned) writes.
- The pseudo-instruction **LDR Rd,=imm** loads 32-bit addresses/constants via a literal pool—handy for SRAM addresses like **0x2000_XXXX**.
- For inclusive ranges: **count = last – first + 1**.

Section 2.4

Problems are paraphrased to respect copyright. For theory and examples, see **Mazidi, Ch. 2 §2.4**.

30) The status register is a(n) _____-bit register.

Answer: 32-bit.

Why: The CPSR (Current Program Status Register) is 32 bits; the top four bits hold N Z C V.

31) Which bits of the status register are used for the C and Z flag bits, respectively?

Answer: C = bit 29, Z = bit 30.

Why: The high nibble of CPSR is N(31) Z(30) C(29) V(28).

32) Which bits of the status register are used for the V and N flag bits, respectively?

Answer: V = bit 28, N = bit 31.

33) In the ADD instruction, when is C raised?

Answer: When there is an **unsigned carry out of bit 31** (i.e., the 32-bit addition exceeds 0xFFFFFFFF).

34) In the ADD instruction, when is Z raised?

Answer: When the **result is zero** (all 32 bits are 0).

35) What is the status of the C and Z flags after the following code?

```
LDR  R0, =0xFFFFFFFF
LDR  R1, =0xFFFFFFFF1
ADDS R1, R0, R1
```

Answer: C = 1, Z = 0.

Why: $0xFFFFFFFF + 0xFFFFFFFF1 = 0x1FFFFFFFF0 \rightarrow$ result (low 32 bits) $0xFFFFFFFF0$ (**non-zero**) with a **carry out** $\rightarrow C=1, Z=0$.

36) Find the C flag value after each of the following codes.

(a)

```
LDR  R0, =0xFFFFF54
LDR  R5, =0xFFFFF4C
ADDS R2, R5, R0
```

Answer: C = 1.

Why: $\text{Sum } 0xFFFFF54 + 0xFFFFF4C = 0x1FFFFFF18 \rightarrow$ carry out.

(b)

```
MOVS R3, #0
LDR  R6, =0xFFFFFFFF
ADDS R3, R3, R6
```

Answer: C = 0.

Why: $\text{Sum } 0 + 0xFFFFFFFF = 0xFFFFFFFF \rightarrow$ no carry.

(c)

```

LDR    R3, =0xFFFFFFFF? ? ; (value shown near 0xFFFFFFFFxx in the problem)
LDR    R8, =0xFFFFFFFF05
ADDS   R2, R3, R8

```

Answer: C = 1 (for the given near-0xFFFFFFFFxx values).

Why: Adding two values both close to 0xFFFFFFFF exceeds 32 bits, producing a carry out (**unsigned overflow**).

37) Write a simple program in which the value 0x55 is added 5 times.

Approach: Accumulate in a loop using ADDS so flags update; keep result in R0.

```

        AREA    |.text|, CODE, READONLY
        EXPORT  _start
        THUMB

_start:
        MOVS    r0, #0x00          ; accumulator
        MOVS    r1, #0x55          ; value to add
        MOVS    r2, #5             ; loop count

add_loop:
        ADDS    r0, r0, r1          ; r0 += 0x55
        SUBS    r2, r2, #1
        BNE     add_loop

        ; Result: r0 = 5 * 0x55 = 0x1A9 (low 32 bits), C set if a carry occurred
        B       .
        END

```

Explanation: The loop adds 0x55 five times (0x1A9 total). Using ADDS updates flags each step; the final flags depend on intermediate carries/zero results (none here).

Notes for learners

- Remember the CPSR high bits order: **N(31) Z(30) C(29) V(28)**.
- Use the **s suffix** (e.g., ADDS, SUBS) to **update flags**.
- **Unsigned vs. signed:** **C** indicates **unsigned carry/borrow**; **V** indicates **signed overflow**.

Section 2.5

Problems are paraphrased to respect copyright. For theory and examples, see **Mazidi, Ch. 2 §2.5**.

38) State the hex value for each of the following EQU constants

Name	Definition	Value (hex)
MYDAT_1	EQU 55	0x37
MYDAT_2	EQU 98	0x62
MYDAT_3	EQU 'G'	0x47
MYDAT_4	EQU 0x50	0x50
MYDAT_5	EQU 200	0xC8
MYDAT_6	EQU 'A'	0x41
MYDAT_7	EQU 0xAA	0xAA
MYDAT_8	EQU 255	0xFF
MYDAT_9	EQU 2_10010000	0x90
MYDAT_10	EQU 2_01111110	0x7E
MYDAT_11	EQU 10	0x0A
MYDAT_12	EQU 15	0x0F

Notes: 2_ denotes **binary**; character constants (e.g., 'G') use ASCII.

39) State the hex value for each of the following EQU constants

Name	Definition	Value (hex)
DAT_1	EQU 22	0x16
DAT_2	EQU 0x56	0x56
DAT_3	EQU 2_10011001	0x99
DAT_4	EQU 32	0x20
DAT_5	EQU 0xF6	0xF6
DAT_6	EQU 2_11111011	0xFB

40) Show a simple code to load the value 0x10102265 into locations 0x40000030–0x4000003F.

Approach: That range is 16 bytes, i.e., **four words** at 0x30, 0x34, 0x38, 0x3C. Use STR (word store) with a 4-iteration loop.

```

        AREA    |.text|, CODE, READONLY
        EXPORT  _start
        THUMB

_start:
        LDR     r0, =0x40000030      ; start (word-aligned)
        LDR     r1, =0x10102265      ; value to store
        MOVS    r2, #4               ; four words = 16 bytes

store_loop40:
        STR     r1, [r0]              ; *(uint32_t*)r0 = 0x10102265
        ADDS    r0, r0, #4           ; next word address
        SUBS    r2, r2, #1
        BNE     store_loop40

        B       .
        END

```

Explanation: Using `STR` avoids byte-by-byte stores and respects word alignment.

41) (a) Load the value `0x23456789` into locations `0x40000060–0x4000006F`, and (b) add them together, placing the result in `R9` as values are added. Use `EQU` to name the locations `TEMP0–TEMP3`.

Approach: Define the four word addresses with `EQU`. Store the word at each address and accumulate the sum in `R9`.

```

        AREA    |.text|, CODE, READONLY
        EXPORT  _start
        THUMB

TEMP0    EQU    0x40000060
TEMP1    EQU    0x40000064
TEMP2    EQU    0x40000068
TEMP3    EQU    0x4000006C

_start:
        LDR     r1, =0x23456789        ; word to replicate
        MOVS    r9, #0                 ; accumulator = 0

        ; -- store to TEMP0..TEMP3 and accumulate --
        LDR     r0, =TEMP0
        STR     r1, [r0]
        ADDS    r9, r9, r1

        LDR     r0, =TEMP1
        STR     r1, [r0]
        ADDS    r9, r9, r1

        LDR     r0, =TEMP2
        STR     r1, [r0]
        ADDS    r9, r9, r1

        LDR     r0, =TEMP3
        STR     r1, [r0]
        ADDS    r9, r9, r1

        ; Now r9 = 4 * 0x23456789 = 0x8D159E24 (mod 2^32)
        B       .
        END

```

Explanation: The range `0x60–0x6F` covers **four words** (16 bytes). Each store is word-aligned. The final sum is `0x8D159E24` (no wrap in 32-bit math).

Notes for learners

- `EQU` defines a **symbolic constant**; it does **not** allocate memory.
- Bases: `0x...` = hex, `2...` = binary, decimal is default.
- For aligned word ranges, prefer `STR` with a **4-byte stride**; for byte ranges use `STRB`.

Section 2.7

Problems are paraphrased to respect copyright. For workflow/background, see **Mazidi, Ch. 2 §2.7**.

42) Assembly language is a _____ (low, high)-level language while C is a _____ (low, high)-level language.

Answer: low, high.

Why: Assembly maps closely to machine instructions; C abstracts the hardware.

43) Of C and Assembly, which is more efficient in terms of code generation (program memory used)?

Answer: Assembly (typically smaller/tighter when hand-optimized).

Why: It gives direct control over instructions. (*Modern compilers may be close, but the textbook expectation is Assembly → smaller code.*)

44) Which program produces the obj (object) file?

Answer: The **assembler** (for .s / .asm sources).

Note: For C sources the **compiler** also emits object files.

45) True or False. The source file has the extension “asm”.

Answer: True (commonly accepted; many toolchains also use .s/.S).

46) True or False. The source code file can be a non-ASCII file.

Answer: False.

Why: Source files are **text (ASCII/UTF-8)**; non-text/binary is invalid as source.

47) True or False. Every source file must have an EQU directive.

Answer: False.

Why: EQU defines constants; it's **optional**.

48) Do the EQU and END directives produce opcodes?

Answer: No.

Why: They are **assembler directives (pseudo-ops)**, not CPU instructions.

49) Why are directives also called pseudocode/pseudo-ops?

Answer: Because they give **instructions to the assembler/linker**, not to the CPU; they **do not generate machine opcodes**.

50) The file with the _____ extension is downloaded into ARM Flash ROM.

Answer: **.hex** (Intel HEX) (*sometimes .bin is also used*).

51) Give three file extensions produced by ARM Keil.

Answer (any three): **.obj**, **.hex**, **.lst** (*also common: .axf, .map, .o*).

Notes for learners

- Typical build: **source (.s/.asm/.c)** → **object (.obj/.o)** → **executable (.axf)** → **image (.hex/.bin)**.
- **Directives** (e.g., `AREA`, `EQU`, `END`) shape assembly/placement but don't execute on the CPU.
- Keil/MDK often uses **.axf** (ELF/DWARF) for debug and **.hex** for programming the MCU.

Section 2.8

Problems are paraphrased to respect copyright. For background, see **Mazidi, Ch. 2 §2.8**.

52) Every ARM family member wakes up at address _____ when it is powered up.

Answer: 0x00000000 (reset vector base).

Why: On reset, execution starts from the **vector table at address 0x00000000** (PC is loaded from that table). Some MCUs can remap, but the architectural default is **0x00000000**.

53) A programmer puts the first opcode at address 0x100. What happens when the microcontroller is powered up?

Answer: It does **not** start at 0x100. The CPU **fetches from the reset vector at 0x00000000**. To run code at 0x100, the reset vector (or early code) must **branch/jump** there; otherwise the CPU executes whatever is at/pointed to by address 0x0.

54) ARM instructions are _____ bytes.

Answer: 4 bytes (32-bit) in ARM state.

Note: Thumb instructions are **16-bit** (some 32-bit encodings), but this section refers to **ARM** instructions.

55) Program: add each digit of your 5-digit ID and store the sum at 0x4000100

Approach: Define the digits with EQU (replace D1..D5 with your own digits), sum them, and store the result as a **word** in SRAM.

```

        AREA    |.text|, CODE, READONLY
        EXPORT  _start
        THUMB

; === Replace these with your actual ID digits (0-9) ===
D1      EQU     1
D2      EQU     2
D3      EQU     3
D4      EQU     4
D5      EQU     5

DST      EQU     0x4000100

_start:
    MOVS      r0, #0                ; accumulator
    MOVS      r1, #D1
    ADDS      r0, r0, r1
    MOVS      r1, #D2
    ADDS      r0, r0, r1
    MOVS      r1, #D3
    ADDS      r0, r0, r1
    MOVS      r1, #D4
    ADDS      r0, r0, r1
    MOVS      r1, #D5
    ADDS      r0, r0, r1

    LDR       r2, =DST
    STR       r0, [r2]              ; store sum as 32-bit word

    B         .
    END

```

Explanation: Five immediate adds accumulate the digit sum; STR writes the 32-bit result to 0x4000100.

56) Show the placement of data for:

```

LDR R1, =0x22334455
LDR R2, =0x20000000
STR R1, [R2]

```

- **Little-endian (ARM MCUs default):**

0x20000000: 55, 0x20000001: 44, 0x20000002: 33, 0x20000003: 22

- **Big-endian:**

0x20000000: 22, 0x20000001: 33, 0x20000002: 44, 0x20000003: 55

Why: Little-endian stores the **least-significant byte** at the **lowest address**.

57) Show the placement of data for:

```
LDR R1, =0xFFEEDDCC
LDR R2, =0x2000002C
STR R1, [R2]
```

- **Little-endian:**

0x2000002C: CC, 0x2000002D: DD, 0x2000002E: EE, 0x2000002F: FF

- **Big-endian:**

0x2000002C: FF, 0x2000002D: EE, 0x2000002E: DD, 0x2000002F: CC

58) How wide is the memory in the ARM chip?

Answer: 8 bits (byte-addressable).

Why: Memory is organized in **bytes**; loads/stores can access **byte/halfword/word**, but the fundamental addressable unit is **8-bit**.

59) How wide is the data bus between the CPU and the program memory in the ARM7 chip?

Answer: 32 bits.

Why: ARM7 implements a **32-bit data path** for instruction and data accesses (device-specific memories may vary, but the core bus is 32-bit).

60) In ADD Rd,Rn,operand2, how many bits are allocated for Rd and how does that cover all GPRs?

Answer: 4 bits for Rd → 16 possible values → R0–R15.

Why: A 4-bit field in the encoding (bits [15:12] in ARM state) selects any of the **16 architected registers**.

Notes for learners

- **Reset/PC:** On Cortex-M, the **initial SP** is at 0x00000000 and the **reset handler address** is at 0x00000004 inside the vector table; execution still **originates from the table at 0x0**.
- **Endianness:** Most microcontrollers ship **little-endian**; big-endian layouts simply reverse byte order at consecutive addresses.
- **Instruction sizes:** ARM (A32) = 32-bit instructions; Thumb (T32) = mostly 16-bit with some 32-bit encodings.

Section 2.9

Problems are paraphrased to respect copyright. For background on addressing forms, see **Mazidi, Ch. 2 §2.9**.

61) Give the addressing mode for each

- (a) `MOV R5, R3` → **Register (register-to-register move)** — Operand is a **register**.
- (b) `MOV R0, #56` → **Immediate** — Operand is an **immediate literal**.
- (c) `LDR R5, [R3]` → **Register indirect** (single data transfer, `[Rn]` = base with **offset 0**, pre-indexed, no write-back).
- (d) `ADD R9, R1, R2` → **Register** (three-register data processing; `Operand2` is a **register**).
- (e) `LDR R7, [R2]` → **Register indirect** (as in (c)).
- (f) `LDRB R1, [R4]` → **Register indirect (byte)** — loads a **byte** from address in **R4**.

62) Show the contents of the memory locations after execution (assume little-endian, the common MCU default).

(a)

```
LDR R2, =0x129F
LDR R1, =0x1450
LDR R2, [R1]
```

- `0x1450 = 0x9F`
- `0x1451 = 0x12`

Why: The constant `0x129F` is laid out in memory as bytes **9F 12 00 00** (little-endian). Loading from `[R1]` reads the word; the memory bytes remain as shown.

(b)

```
LDR R4, =0x8C63
LDR R1, =0x2400
LDRH R4, [R1]
```

- `0x2400 = 0x63`
- `0x2401 = 0x8C`

Why: Halfword `0x8C63` is stored as bytes **63 8C** in little-endian order.

Notes for learners

- **Register indirect** means the **effective address** comes from a register (e.g., `[R3]`). Adding an offset uses forms like `[R3, #imm]` or `[R3, Rm{, shift}]`.
- **Immediate form** uses a literal (`#imm`) as the operand; for memory, there is **no “store-immediate”**—load the immediate into a register, then store.
- Endianness controls **byte order** in memory; ARM MCUs are typically **little-endian**.

Section 2.10

Problems are paraphrased to respect copyright. For background, see **Mazidi, Ch. 2 §2.10**.

63) What do RISC and CISC stand for?

Answer: RISC = Reduced Instruction Set Computer; CISC = Complex Instruction Set Computer.

64) In _____ (RISC, CISC) architecture we can have 1-, 2-, 3-, or 4-byte instructions.

Answer: CISC.

Why: CISC ISAs typically use **variable-length encodings** (e.g., 1–15 bytes in x86).

65) In _____ (RISC, CISC) architecture instructions are fixed in size.

Answer: RISC.

Why: RISC ISAs prefer **fixed-length instructions** (e.g., ARM/A32 uses 32-bit fixed; Thumb uses fixed 16-bit with some 32-bit encodings within that mode).

66) In _____ (RISC, CISC) architecture instructions are mostly executed in one or two cycles.

Answer: RISC.

Why: RISC designs emphasize **simple, pipelined, single-cycle** operations and load/store memory access.

67) In _____ (RISC, CISC) architecture we can have an instruction to ADD a register to external memory.

Answer: CISC.

Why: CISC allows **ALU operations directly on memory operands** (e.g., `ADD [mem], reg`), whereas RISC requires **load → operate → store**.

68) True or False. Most instructions in CISC are executed in one or two cycles.

Answer: False.

Why: CISC instructions often have **variable cycle counts** depending on addressing mode/memory access; they are not predominantly 1–2 cycles in the classic model.

Notes for learners

- **ARM** is a **RISC** architecture: fixed-size instruction encodings per mode, **load/store** design, simple addressing in ALU ops.
- **CISC** favors **rich addressing modes** and **variable-length encodings**, enabling memory operands in ALU instructions.

Chapter 3

Section 3.1

Problems are paraphrased to respect copyright. This section focuses on **ADDS/ADC** and the **C (carry) / Z (zero)** flags.

1) Find C and Z for each case. Also give the result and where it is saved.

(a)

```
MOV    R1, #0x3F
MOV    R2, #0x45
ADDS   R3, R1, R2
```

- **Computation:** $0x3F + 0x45 = 0x84$
- **Result:** $R3 = 0x00000084$
- **Flags:** $C=0, Z=0$ (no carry out; result not zero).

(b)

```
LDR    R0, =0x95999999
LDR    R1, =0x94FFFF58
ADDS   R1, R1, R0
```

- **Computation:** $0x95999999 + 0x94FFFF58 = 0x12A9998F1 \rightarrow$ low 32 bits $0x2A9998F1$
- **Result:** $R1 = 0x2A9998F1$
- **Flags:** $C=1$ (carry out), $Z=0$.

(c)

```
LDR    R0, =0xFFFFFFFF
ADDS   R0, R0, #1
```

- **Computation:** $0xFFFFFFFF + 1 = 0x1_0000_0000 \rightarrow$ low 32 bits $0x00000000$
- **Result:** $R0 = 0x00000000$
- **Flags:** $C=1, Z=1$.

(d)

```
LDR    R2, =0x00000001
LDR    R1, =0xFFFFFFFF
ADD    R0, R1, R2      ; does NOT set flags
ADCS   R0, R0, #0      ; adds carry-in and sets flags
```

- After **ADD**: $R0 = 0x00000000$ (flags **unchanged**).
- **ADCS** uses the **previous C** (not set by the **ADD**). Assuming prior $C=0$ (typical unless set earlier):
 - **Result:** $R0 = 0x00000000$
 - **Flags set by ADCS:** $C=0, Z=1$.
(If prior $C=1$, then $R0=0x00000001$ and $Z=0$.)

(e)

```
LDR    R0, =0xFFFFFFFF
ADDS   R0, R0, #2
ADC    R1, R0, #0      ; uses carry from ADDS; does not set flags
```

- **Computation:** $0xFFFFFFFF + 2 = 0x1_0000_0000 \rightarrow R0 = 0x00000000$
- **Flags after ADDS:** $C=1, Z=1$
- **Then ADC:** $R1 = R0 + 0 + C = 0 + 0 + 1 = 0x00000001$ (flags unchanged).

2) State the three steps in a subtraction (SUB) and apply them.

Three steps (A – B):

1. **One's complement** of B $\rightarrow \sim B$.
2. **Add 1** to form **two's complement** of B.
3. **Add** to A: $A + (\sim B + 1)$. In ARM, the **C flag after subtraction** means: **C=1** \rightarrow **no borrow**, **C=0** \rightarrow **borrow**.

Apply to 8-bit examples (showing intermediate two's complement):

- (a) $0x23 - 0x12$
 - $\sim 0x12 = 0xED$, $+1 \rightarrow 0xEE$; $0x23 + 0xEE = 0x111 \rightarrow$ result $0x11$, **C=1** (no borrow).
 - (b) $0x43 - 0x51$
 - $\sim 0x51 = 0xAE$, $+1 \rightarrow 0xAF$; $0x43 + 0xAF = 0xF2 \rightarrow$ result $0xF2$ (i.e., $-0x0E$ in 8-bit), **C=0** (borrow occurred).
 - (c) $0x99 - 0x39$
 - $\sim 0x39 = 0xC6$, $+1 \rightarrow 0xC7$; $0x99 + 0xC7 = 0x160 \rightarrow$ result $0x60$, **C=1** (no borrow).
-

Notes for learners

- **ADD** vs **ADDS**: only forms with **S** update flags.
- **ADC/ADCS** add the **carry-in**; **ADCS** also **updates** flags.
- In ARM subtraction, remember: **C = NOT borrow**.

Section 3.2

Problems are paraphrased to respect copyright. Assume **independent** sub-questions unless noted. Initial registers for Q3: **R0 = 0xF000, R1 = 0x3456, R2 = 0xE390.**

3) Perform each operation; give the result and the destination register.

- (a) `AND R3,R2,R0` \rightarrow `R3 = 0xE390 AND 0xF000 = 0xE000`
- (b) `ORR R3,R2,R1` \rightarrow `R3 = 0xE390 OR 0x3456 = 0xF7D6`
- (c) `EOR R0,R0,#0x76` \rightarrow `R0 = 0xF000 XOR 0x0076 = 0xF076`
- (d) `AND R3,R2,R2` \rightarrow `R3 = 0xE390 AND 0xE390 = 0xE390`
- (e) `EOR R0,R0,R0` \rightarrow `R0 = 0x00000000` (XOR with itself clears)
- (f) `ORR R3,R0,R2` \rightarrow `R3 = 0xF000 OR 0xE390 = 0xF390`
- (g) `AND R3,R0,#0xFF` \rightarrow `R3 = 0xF000 AND 0x00FF = 0x00`
- (h) `ORR R3,R0,#0x99` \rightarrow `R3 = 0xF000 OR 0x0099 = 0xF099`
- (i) `EOR R3,R1,R0` \rightarrow `R3 = 0x3456 XOR 0xF000 = 0xC456`
- (j) `EOR R3,R1,R1` \rightarrow `R3 = 0x00000000`

4) Value in R2 after executing:

```
MOV    R0,#0xF0
MOV    R1,#0x55
BIC    R2,R1,R0
```

Answer: `R2 = R1 AND (~R0) = 0x55 AND 0x0F = 0x05 \rightarrow R2 = 0x00000005.`

5) Value in R2 after executing:

```
LDR    R1,=0x55555555
MVN    R0,#0
EOR    R2,R1,R0
```

Answer: `R0 = ~0 = 0xFFFFFFFF; R2 = R1 XOR R0 = ~R1 = 0xAAAAAAAA \rightarrow R2 = 0xAAAAAAAA.`

Notes for learners

- **BIC Rd,Rn,Op2:** `Rd = Rn AND NOT Op2.`
- **MVN Rd,Op2:** `Rd = NOT Op2.`
- **EOR with itself clears a register; AND with itself preserves it; ORR with anything sets the union of bits.**

Section 3.3

Problems are paraphrased to respect copyright. Results are shown in **32-bit hex** unless stated. When an instruction has the **S** suffix (e.g., **MOVS**), the **N/Z/C** flags are updated from the **shifter**: for **ROR #n** the carry-out is the original bit (**n-1**); for **LSR #n** it is the original bit (**n-1**); for **LSL #n** it is the original bit (**32-n**); for **RRX** the carry-out is the original bit **0** and bit31 is filled with the old **C**.

6) Assuming **c=0**, what is the value of **R1** after:

```
MOV    R1, #0x25
MOVS   R1, R1, ROR #4
```

Answer: **R1** = 0x50000002.

Why: 0x00000025 ROR 4 = 0x50000002. (Carry-out would be old bit3 = 0.)

7) Assuming **c=0**, what are the values of **R0** and **C** after:

```
LDR    R0, =0x3FA2
MOV     R2, #8
MOVS   R0, R0, ROR R2
```

Answer: **R0** = 0xA200003F, **C** = 1.

Why: Rotate right by 8: bytes 00 00 3F A2 → A2 00 00 3F; carry-out is original bit7 = 1.

8) Assuming **c=0**, what are the values of **R2** and **C** after:

```
MOV     R2, #0x55
MOVS   R2, R2, RRX
```

Answer: **R2** = 0x0000002A, **C** = 1.

Why: **RRX** shifts right by one, **bit31 ← old C(0)**, and carry-out gets the original **bit0 (1)**.

9) Assuming **c=0**, what is the value of **R1** after:

```
MOV     R1, #0xFF
MOV     R3, #5
MOVS   R1, R1, ROR R3
```

Answer: **R1** = 0xF8000007.

Why: 0x000000FF ROR 5 = 0xF8000007 (carry-out would be original bit4 = 1).

10) Give the destination register value after the instruction executes

(Assembler shorthand: **MOV Rd, #imm, ROR #n** rotates the 8-bit immediate by **n** and writes the result to **Rd**.)

- (a) **MOV R1, #0x88, ROR #4** → **R1** = 0x80000008
- (b) **MOV R0, #0x22, ROR #22** → **R0** = 0x00008800
- (c) **MOV R2, #0x77, ROR #8** → **R2** = 0x77000000
- (d) **MOV R4, #0x5F, ROR #28** → **R4** = 0x000005F0
- (e) **MOV R6, #0x88, ROR #22** → **R6** = 0x00022000
- (f) **MOV R5, #0x8F, ROR #16** → **R5** = 0x008F0000
- (g) **MOV R7, #0xF0, ROR #20** → **R7** = 0x000F0000
- (h) **MOV R1, #0x33, ROR #28** → **R1** = 0x00003300

11) Give the destination register value for each **MVN** (bitwise NOT of the rotated immediate)

- (a) **MVN R2, #0x01** → **R2** = 0xFFFFF000

- **(b)** MVN R2, #0xAA, ROR #20 \rightarrow R2 = 0xFFFF55FFF
- **(c)** MVN R1, #0x55, ROR #4 \rightarrow R1 = 0xAFFFFFFFA
- **(d)** MVN R0, #0x66, ROR #28 \rightarrow R0 = 0xFFFFF99F
- **(e)** MVN R2, #0x80, ROR #24 \rightarrow R2 = 0xFFFF7FFF
- **(f)** MVN R6, #0x10, ROR #20 \rightarrow R6 = 0xFFFFFFF
- **(g)** MVN R7, #0xF0, ROR #24 \rightarrow R7 = 0xFFFF0FFF
- **(h)** MVN R4, #0x99, ROR #4 \rightarrow R4 = 0x6FFFFFF6

12) Compute the results (and the C flag) for mixed immediate/register shifts

(a)

```
MOV    R0, #0x04
MOVS   R1, R0, LSR #2
MOVS   R3, R0, LSR R1      ; R1 holds the shift amount (=1)
```

Answer: R1 = 0x00000001, R3 = 0x00000002, C = 0.

Why: 0x04 >>2 = 0x01 (carry-out old bit1=0); then 0x04 >>1 = 0x02 (carry-out old bit0=0).

(b)

```
LDR    R1, =0x0000A0F2
MOV    R2, #0x3
MOVS   R3, R1, LSL R2
```

Answer: R3 = 0x00050790, C = 0.

Why: A0F2 <<3 = 0x50790; bits shifted out of the top were zero \rightarrow C=0.

(c)

```
LDR    R1, =0x0000B085
MOV    R2, #3
MOVS   R4, R1, LSR R2
```

Answer: R4 = 0x00001610, C = 1.

Why: B085 >>3 = 0x1610; carry-out is original bit2 = 1.

13) Give the register values and final C after each sequence

(a)

```
SUBS   R2, R2, R2      ; R2 = 0, sets Z=1, C=1 (no borrow)
MOV    R0, #0xAA
MOVS   R1, R0, ROR #4
```

Answer: R2 = 0x00000000, R0 = 0x000000AA, R1 = 0xA000000A, **C = 1**.

Why: Rotate right 4 \rightarrow carry-out = original bit3 of R0 = 1.

(b)

```
MOV    R2, #0xAA, ROR #4
MOV    R0, #1
MOVS   R1, R2, ROR R0
```

Answer: R2 = 0xA000000A, R1 = 0x50000005, **C = 0**.

Why: ROR by 1 \rightarrow carry-out = original bit0 of R2 = 0.

(c)

```
LDR    R1, =0x00001234
MOV    R2, #0x10, ROR #2      ; R2 low byte = 0x04  $\rightarrow$  shift amount = 4
MOVS   R1, R1, ROR R2
```

Answer: R2 = 0x40000004, R1 = 0x40000123, **C = 0**.

Why: $0x1234 \text{ ROR } 4 = 0x40000123$; carry-out = original bit3 = 0.

(d)

```
MOV    R0, #0xAA
; assume entry C=0 unless set earlier
MOVS   R1, R0, RRX
```

Answer: $R1 = 0x00000055$, **C = 0**.

Why: $RRX: \text{bit31-old } C(0), \text{result } 0xAA \gg 1 = 0x55$, carry-out = original bit0 (0).

14) Find the C flag after each of the following

(a)

```
MOV    R0, #0x20
MOVS   R1, R0, LSR #2
```

Answer: **C = 0** (original bit1 of R0 is 0).

(b)

```
LDR    R8, =0x00000006
MOVS   R1, R8, LSR #2
```

Answer: **C = 1** (original bit1 of R8 is 1).

(c)

```
LDR    R6, =0x0000001F
MOVS   R1, R6, LSL #3
```

Answer: **C = 0** (for LSL #3, carry-out is original bit29 which is 0 here).

Notes for learners

- **RRX is a rotate with carry:** $C_{out} \leftarrow \text{old bit0}, \text{bit31} \leftarrow \text{old } C$.
- For register-specified shifts ($\dots \text{ROR } R_m, \dots \text{LSR } R_m$, etc.), only the **low byte** of R_m is used; the effective shift is modulo 32.
- When using the **S** suffix, remember: the flags come from the **shifter**, not the destination ALU stage.

Section 3.5

Problems are paraphrased to respect copyright. Results are shown in **8-bit hex** where appropriate.

15) Convert 0x76 (packed BCD) to ASCII digits; place ASCII codes in R1 and R2

Approach

- Extract **tens** = high nibble (value >> 4) and **ones** = low nibble (value & 0xF).
- Convert each digit to ASCII by **adding 0x30**.

Solution

```

        AREA    |.text|, CODE, READONLY
        EXPORT  _start
        THUMB

_start:
        MOVS    r0, #0x76          ; packed BCD = 0x76 (digits 7 and 6)

        LSRS    r1, r0, #4          ; r1 = 0x07 (tens)
        ANDS    r2, r0, #0x0F       ; r2 = 0x06 (ones)

        ADDS    r1, r1, #0x30        ; ASCII '7' = 0x37
        ADDS    r2, r2, #0x30        ; ASCII '6' = 0x36

        B       .
        END

```

Result: R1 = 0x37 (ASCII '7'), R2 = 0x36 (ASCII '6').

16) Keyboard provides ASCII 0x33 ('3') and 0x32 ('2'). Convert them to packed BCD and store in R2

Approach

- Convert ASCII to numeric: **subtract 0x30** from each.
- Pack: (tens << 4) | ones.

Solution

```

        AREA    |.text|, CODE, READONLY
        EXPORT  _start
        THUMB

_start:
        MOVS    r0, #0x33          ; ASCII '3'
        MOVS    r1, #0x32          ; ASCII '2'

        SUBS    r0, r0, #0x30        ; r0 = 3
        SUBS    r1, r1, #0x30        ; r1 = 2

        LSLS    r0, r0, #4          ; r0 = 0x30 (tens in high nibble)
        ORR     r2, r0, r1          ; r2 = 0x32 (packed BCD)

        B       .
        END

```

Result: R2 = 0x32 (packed BCD “3 2”).

Notes for learners

- **ASCII digit ↔ numeric digit:** digit_ascii = digit + 0x30 and digit = digit_ascii - 0x30.
- **Packed BCD** stores two 4-bit digits per byte; useful when printing/reading decimal without full division.

Chapter 4

Section 4.1

Problems are paraphrased to respect copyright. When helpful, results are shown in **decimal** and **hex**.

1) In ARM, looping action using a single register is limited to _____ iterations.

Answer: 4,294,967,296 iterations (2^{32}).

Why: A loop counter can be held in one 32-bit register and decremented with `SUBS ..., #1` and `BNE` until zero.

2) If a conditional branch is not taken, what instruction executes next?

Answer: The next sequential (fall-through) instruction (i.e., the one at `PC+4` in ARM state).

3) In calculating the branch target, a displacement is added to register _____.

Answer: `PC (R15)` — branches are **PC-relative**.

4) The mnemonic `BNE` stands for _____.

Answer: **Branch if Not Equal** (i.e., `Z == 0`).

5) What is the advantage of using `BX` over `B`?

Answer: `BX` branches to an address in a register and can switch instruction set state (ARM ↔ Thumb based on bit0), whereas `B` is PC-relative and does not change state.

6) True or False. The target of a `BNE` can be anywhere in the 4 GB address space.

Answer: **False.** The PC-relative range of ARM `B{cond}` is limited (see Q8).

7) True or False. All ARM branch instructions can branch to anywhere in the 4 GB byte space.

Answer: **False.** Branch ranges are **finite** (PC-relative immediates).

8) Dissect the `B` instruction: how many bits are for the operand vs. the opcode, and how far can it branch?

Answer: In ARM state, `B{cond}` uses **24 bits** for the **signed immediate operand** (`imm24`), and **8 bits** for the **opcode/condition** (`cond[31:28] + 101 + L`). The target is `PC + sign_extend(imm24 << 2)`, so the range is approximately **±32MB** ($\pm 2^{25}$ bytes).

9) True or False. All conditional branches are 2-byte instructions.

Answer: **False.** In ARM (A32) they are **4 bytes**; only **Thumb** has 16-bit conditional branches.

10) Show code for a nested loop that performs an action 10,000,000,000 times.

```

; Outer = 10,000 (0x2710), Inner = 1,000,000 (0x0F4240)
AREA |.text|, CODE, READONLY
EXPORT _start
THUMB

_start:
LDR    r2, =0x00002710      ; outer count = 10,000
Outer:
LDR    r1, =0x000F4240      ; inner count = 1,000,000
Inner:
; ---- ACTION HERE (one time per inner iteration) ----
NOP                      ; replace with your code
; -----
SUBS   r1, r1, #1
BNE    Inner              ; run inner exactly 1,000,000 times
SUBS   r2, r2, #1
BNE    Outer              ; repeat outer 10,000 times
B      .
END

```

Total iterations: $10,000 \times 1,000,000 = 10,000,000,000$.

11) Show code for a nested loop that performs an action 200,000,000,000 times.

```

; Outer = 20,000 (0x4E20), Inner = 10,000,000 (0x00989680)
AREA |.text|, CODE, READONLY
EXPORT _start
THUMB

_start:
LDR    r2, =0x00004E20      ; outer = 20,000
Outer2:
LDR    r1, =0x00989680      ; inner = 10,000,000
Inner2:
; ---- ACTION HERE ----
NOP
; -----
SUBS   r1, r1, #1
BNE    Inner2
SUBS   r2, r2, #1
BNE    Outer2
B      .
END

```

Total iterations: $20,000 \times 10,000,000 = 200,000,000,000$.

12) How many times is the loop body executed?

```

MOV    R0, #0x55
MOV    R2, #40
L1:    LDR    R1, =10000000    ; ten million per outer pass
L2:    EOR    R0, R0, #0xFF    ; loop body (the "action")
        SUB    R1, R1, #1
        BNE    L2
        SUB    R2, R2, #1
        BNE    L1

```

Answer: 400,000,000 times ($40 \times 10,000,000$).

13) Status of Z and C after CMP

Recall: CMP Rn, Op2 computes $R_n - Op2$.

- **Z = 1** if equal.
- **C = 1** if no borrow (i.e., $R_n \geq Op2$ as unsigned).
- **(a)** $R0=0x32, R1=0x28 \rightarrow 0x32 - 0x28 \rightarrow Z=0, C=1$.
- **(b)** $R1=0xFF, R2=0x6F \rightarrow Z=0, C=1$.

- (c) $R2=0x34, R3=0x88 \rightarrow Z=0, C=0$.
- (d) $R1=0, R2=0 \rightarrow Z=1, C=1$.
- (e) $R2=0, R3=0xFF \rightarrow Z=0, C=0$.
- (f) $R0=0, R1=0 \rightarrow Z=1, C=1$.
- (g) $R4=0x78, R2=0x40 \rightarrow Z=0, C=1$.
- (h) $R0=0xAA \ \& \ 0x55 = 0x00$, compare with #0 $\rightarrow Z=1, C=1$.

14) Rewrite “Program 4-1” to find the lowest grade

Assume an array of **N unsigned bytes** at **GRADES**, result in **R2**.

```

        AREA    |.text|, CODE, READONLY
        EXPORT  find_min
        THUMB

GRADES  EQU     0x20000000
N       EQU     40

find_min:
    LDR        r0, =GRADES        ; r0 = base
    LDR        r1, =N              ; r1 = count
    LDRB       r2, [r0], #1        ; r2 = current minimum (first element)
    SUBS       r1, r1, #1          ; remaining

loop_min:
    CBZ        r1, done
    LDRB       r3, [r0], #1
    CMP        r3, r2              ; if r3 < r2 update min
    BHS        skip                ; BHS: r3 >= r2 (unsigned) → keep old min
    MOV        r2, r3              ; new min
skip:    SUBS   r1, r1, #1
    BNE        loop_min
done:    BX     lr
        END

```

15) The target of a BNE is backward if the relative offset is _____.

Answer: **negative** (sign-extended $\text{imm24} \ll 2$ is < 0).

16) The target of a BNE is forward if the relative offset is _____.

Answer: **positive**.

Notes for learners

- $B\{\{cond\}\}$ targets are **PC-relative**; the assembler converts labels to signed offsets.
- For **very long jumps**, use an absolute branch via a register: `LDR rX,=dest ; BX rX`.
- Flag meanings for CMP: think **unsigned** for **C** (borrow/no-borrow) and **equality** for **Z**.

Section 4.2

Problems are paraphrased to respect copyright. Short teaching notes follow each answer.

17) BL is a(n) ___-byte instruction.

Answer: 4 bytes.

Why: In ARM state the encoding is a single 32-bit word. (*In Thumb, BL is encoded as two 16-bit halfwords—still 32 bits total.*)

18) In ARM, which register is the link register?

Answer: R14 (LR).

Why: BL writes the return address into LR.

19) True or False. The BL target address can be anywhere in the 4-GB byte address space.

Answer: False.

Why: BL is PC-relative with a signed 24-bit immediate ($\ll 2$); the branch range is about $\pm 32\text{MB}$ from the call site.

20) Describe how we can return from a subroutine in ARM.

Answer: Restore PC from LR, typically with **BX LR** (preferred, preserves state) or **MOV PC, LR**. If LR was saved on the stack, use **POP {PC}** (or load LR then **BX LR**).

21) In ARM, which address is saved when the BL instruction executes?

Answer: The return address—i.e., the address of the instruction following BL—is saved in LR (R14). (*Architecturally, it is the next sequential instruction address.*)

Notes for learners

- BLX calls via a register or immediate and can switch state (ARM \leftrightarrow Thumb).
- Typical subroutine prologue/epilogue on MCUs: **PUSH {LR}** ... **POP {PC}** (or **BX LR**) if the routine calls other functions.

Section 4.3

Problems are paraphrased to respect copyright. We show the cycle math and the final wall-clock delay using $F = 1/T$.

Timing assumptions used

- One **machine cycle** = one CPU clock period $T = 1/F$.
- Instruction costs used in this section:
NOP = 1, SUBS (reg, #imm) = 1, LDR (literal) = 3, BNE taken = 3, BNE not-taken = 1.
- For a nested delay with outer count **M**, inner count **N**, and **k** NOPs in the inner body, the total cycles are:

$$\text{TotalCycles} = M \times (k+4) \times N + 5M - 1$$

(derivation: inner loop costs = $(k+4)N - 2$; per outer iteration add LDR=3 and the outer SUBS/BNE; include first MOV once).

22) Oscillator frequency if the machine cycle = 1.25 ns

Answer: 800 MHz (i.e., 800MHz).

23) Machine cycle if $F = 200 \text{ MHz}$

Answer: 5.00 ns (i.e., 5ns).

24) Machine cycle if $F = 100 \text{ MHz}$

Answer: 10.00 ns (i.e., 10ns).

25) Machine cycle if $F = 160 \text{ MHz}$

Answer: 6.25 ns (i.e., 6.25ns).

26) Delay of the subroutine ($M=200$, $N=4,000,000,000$, inner has $k=1$ NOP) at 80MHz

- **Total cycles:** 4,000,000,000,999
- **Delay:** 50000.000012488 s \approx 13 h 53 min 20.000012 s

27) Delay of the subroutine ($M=100$, $N=50,000,000$, inner has $k=2$ NOPs) at 50MHz

- **Total cycles:** 30,000,000,499
- **Delay:** 600.000009980 s \approx 10 min 0.000010 s

28) Delay of the subroutine ($M=200$, $N=20,000,000$, inner has $k=3$ NOPs) at 40MHz

- **Total cycles:** 28,000,000,999
- **Delay:** 700.000024975 s \approx 11 min 40.000025 s

29) Delay of the subroutine ($M=500$, $N=20,000$, inner has $k=3$ NOPs) at 100MHz

- **Total cycles:** 70,002,499
- **Delay:** 0.700024990 s \approx 0.700025 s

30) “ARM chip does not have the NOP instruction” — what is used instead?

Answer: Assemblers accept the mnemonic **NOP**, which they assemble to a no-effect data-processing instruction such as **MOV r0, r0** (on classic ARM/ARM7). Newer architectures add a real NOP encoding, but on older parts it's this **MOV** pseudo-op.

Cross-checks

- If your core/flash adds wait states, timing will be **longer** than the idealized values above.
- If your toolchain lists different cycle counts (e.g., **LDR** latency), redo the math with those numbers; the structure stays the same.

Section 4.4

Problems are paraphrased to respect copyright. Answers assume ARM (A32) where most instructions include a **condition field**.

31) Which bits of the ARM instruction are set aside for condition execution?

Answer: bits [31:28] — the 4-bit **cond** field (e.g., `EQ/NE/CS/CC/.../AL`).

32) True or False. Only ADD and MOV have the conditional execution feature.

Answer: False. In ARM state, **nearly all instructions** (data processing, loads/stores, branches, etc.) have the **cond** field; the default is **AL** (always).

33) True or False. In ARM, the conditional execution is default.

Answer: True. Every instruction encodes a condition; when no suffix is written, it means **AL** (execute **always**).

34) Which flag is examined before the instruction MOVEQ executes?

Answer: The **Z (zero)** flag — `EQ` means **Z = 1**.

35) Difference between ADDEQ and ADDNE.

Answer: Both add, but **ADDEQ** executes only if **Z = 1** (equal), whereas **ADDNE** executes only if **Z = 0** (not equal).

36) Difference between BAL and B.

Answer: No functional difference in ARM state. `BAL` is just `B` with the explicit **AL** condition (branch **always**).

37) Difference between SUBCC and SUBCS.

Answer: The operation is the same (`SUB`), but the condition differs:

- **CC = C = 0** (carry clear → **borrow occurred**, unsigned <).
- **CS = C = 1** (carry set → **no borrow**, unsigned ≥).

38) Difference between ANDEQ and ANDNE.

Answer: **ANDEQ** executes if **Z = 1**; **ANDNE** executes if **Z = 0**.

39) True or False. The decision to execute SUBCC is based on the Z flag.

Answer: False. `CC` is based on the **C (carry)** flag being 0.

40) True or False. The decision to execute ADDEQ is based on the Z flag.

Answer: True. `EQ` tests **Z = 1**.

Notes for learners

Common condition suffixes (test on CPSR **N,Z,C,V**):

- EQ $Z=1$, NE $Z=0$
- CS/HS $C=1$, CC/LO $C=0$
- MI $N=1$, PL $N=0$
- VS $V=1$, VC $V=0$
- HI $C=1$ & $Z=0$, LS $C=0$ or $Z=1$
- GE $N=V$, LT $N!=V$, GT $Z=0$ & $N=V$, LE $Z=1$ or $N!=V$
- AL always

Chapter 5

Section 5.1

Problems are paraphrased to respect copyright. Answers show the 32-bit **two's-complement** representation (hex).

How to convert (quick refresher)

- **Positive** values: write the hex value and **zero-extend to 8 hex digits** (32 bits).
- **Negative** $-N$: write N in hex (32-bit), then **invert** (bitwise NOT) and **add 1**.

1) 32-bit representations

item	value	32-bit two's-complement
(a)	-23	0xFFFFFFE9
(b)	$+12$	0x0000000C
(c)	$-0x28$	0xFFFFFDD8
(d)	$+0x6F$	0x0000006F
(e)	-128	0xFFFFFFF8
(f)	$+127$	0x0000007F
(g)	$+365$	0x0000016D
(h)	$-32,767$	0xFFFF8001

Checks (sketch):

- (a) $23 = 0x00000017$; $\sim 17 = 0xFFFFF8E8$; $+1 \rightarrow 0xFFFFF8E9$.
- (h) $32767 = 0x00007FFF$; $\sim = 0xFFFF8000$; $+1 \rightarrow 0xFFFF8001$.

2) 32-bit representations

item	value	32-bit two's-complement
(a)	-230	0xFFFFF1A
(b)	$+1200$	0x00004B0
(c)	$-0x28F$	0xFFFFD71
(d)	$+0x6FF$	0x00006FF

Checks (sketch):

- (a) $230 = 0x000000E6$; $\sim = 0xFFFFF19$; $+1 \rightarrow 0xFFFFF1A$.
- (c) $0x28F$; $\sim = 0xFFFFD70$; $+1 \rightarrow 0xFFFFD71$.

Notes for learners

- The **sign bit** is bit31 (1 = negative).
- Adding a positive number to its two's-complement negative gives **0** modulo 2^{32} .
- To verify: in most programmer's calculators, set **word size = 32**, **two's complement**, and toggle **DEC/HEX**.

Section 5.2

Problems are paraphrased to respect copyright. Byte arithmetic means **8-bit two's-complement** (range $-128\dots+127$).

3) Find the overflow flag (V) for each; do byte-sized calculations

Rule of thumb (ADD): **same sign in, different sign out** $\Rightarrow V=1$; otherwise $V=0$.

item	operation (8-bit)	numeric sum	8-bit result	V
(a)	$(+15) + (-12)$	+3 0x03		0
(b)	$(-123) + (-127)$	-250 0x06 (wrap)		1
(c)	$(+0x25) + (+34)$	+71 0x47		0
(d)	$(-127) + (+127)$	0 0x00		0
(e)	$(+100) + (-100)$	0 0x00		0

Notes: In (b) both operands are **negative** yet the 8-bit result has a **positive sign bit (0)** \rightarrow overflow.

4) Sign-extend the following to 32 bits and show a tiny program to verify

Assumptions on source widths: decimal values within $|N| \leq 128$ are treated as **8-bit**; $0x999$ is treated as **12-bit**; -129 is shown for **16-bit** (also give optional 9-bit note).

item	source width	original value	32-bit sign-extended
(a) -122	8-bit	0x86	0xFFFFFFFF86
(b) -0x999	12-bit	0x999	0xFFFFFFFF999
(c) +0x17	8-bit	0x17	0x00000017
(d) +127	8-bit	0x7F	0x0000007F
(e) -129	16-bit	0xFF7F	0xFFFFFFFF7F

If you instead assume a 9-bit source for (e), $0x017F \rightarrow 0xFFFFFE7F$, still representing -129 .

Verification snippets (Thumb):

- 8-bit to 32-bit (use `SXTB`), example for **-122**:

```

THUMB
MOVS    r0, #0x86      ; 8-bit pattern for -122
SXTB    r1, r0          ; r1 = 0xFFFFFFFF86

```

- 12-bit to 32-bit (generic `LSL/ASR`), example for **0x999**:

```

LDR      r0, =0x00000999 ; treat as 12-bit signed
LSL      r0, r0, #20      ; move sign bit to bit31
ASR      r0, r0, #20      ; arithmetic right shift back  $\Rightarrow$  0xFFFFF999

```

- 16-bit to 32-bit (use `SXTH`), example for **-129**:

```

LDR      r0, =0xFF7F
SXTH     r1, r0           ; r1 = 0xFFFFFFFF7F (-129)

```

5) Modify Program 5-2 to find the highest temperature (signed bytes)

Assume an array of **N signed bytes** at `TEMPS` (e.g., $-40\dots+125^\circ\text{C}$). We scan with signed loads and keep the **maximum**.

```
        AREA    |.text|, CODE, READONLY
        EXPORT  find_max_temp
        THUMB

TEMPS   EQU     0x20000000      ; array base
N       EQU     64              ; number of samples

find_max_temp:
    LDR        r0, =TEMPS
    LDR        r1, =N
    LDRSB      r2, [r0], #1      ; r2 = current max (first element), sign-extended
    SUBS       r1, r1, #1

.loop:
    CBZ        r1, .done
    LDRSB      r3, [r0], #1      ; signed load
    CMP        r3, r2            ; signed compare (works because both are 32-bit signed)
    BLE        .skip            ; if r3 <= r2 keep old max
    MOV        r2, r3            ; else update max
.skip: SUBS    r1, r1, #1
    BNE        .loop
.done:  BX      lr              ; max in r2
    END
```

Why this works: `LDRSB` performs **sign extension** from byte to 32-bit; `CMP` and the conditional `BLE` use signed interpretation when comparing general registers, so we correctly track the **highest** signed temperature.

Notes for learners

- On ARM, `V` reflects **signed overflow**, while `C` reflects **unsigned carry/no-borrow**.
- Sign-extend with: `SXTB` (8→32), `SXTH` (16→32), or the **LSL+ASR trick** for arbitrary widths.

Sketch: $12.9375 = 1100.1111_2 = 1.1001111 \times 2^3$ and $98.8125 = 1100010.1101_2 = 1.1000101101 \times 2^6 \rightarrow$ add the bias 1023 and fill the fraction.

Notes for learners

- The **hidden 1** is present for all **normalized** numbers (not for subnormals).
- Rounding mode by default is **round to nearest, ties to even**; that's why some decimal fractions (e.g., 0.00075) get long fraction fields and rounding.
- For quick checks: interpret the hex in a programmer's calculator; confirm S, E, and F by splitting the bit fields.

Chapter 6

Section 6.1

Problems are paraphrased to respect copyright. Where useful, I show short teaching notes and the arithmetic.

1) What is the bus bandwidth unit?

Answer: bytes per second (often expressed as **MB/s**).

2) Give the variables that affect bus bandwidth.

Answer:

- **Data-bus width** (bytes per transfer).
- **Bus clock frequency** (transfers per second).
- **Cycles per transfer** (wait states, turnaround, arbitration).
- Optional efficiency factors: **burst length**, **cache/hit ratio**, etc.

Rule of thumb: $\text{Bandwidth} = (\text{bus_width_bytes} \times \text{bus_clock}) / (\text{cycles_per_transfer})$.

3) True/False — One way to increase bus bandwidth is to widen the data bus.

Answer: True. Wider bus \Rightarrow more bytes moved per cycle.

4) True/False — Increasing the number of address-bus pins raises bus bandwidth.

Answer: False. A wider address bus increases the **addressable space**, not the transfer rate.

5) Calculate memory-bus bandwidth

Assume a **32-bit data bus (4 bytes)** and that $\text{cycles per transfer} = 1 + \text{wait_states}$.

- **(a) 100MHz, 0 WS** \rightarrow transfers/sec = $100\text{M} / 1 = 100\text{M}$.
Bandwidth = $4 \times 100\text{M} = 400\text{MB/s}$.
- **(b) 80MHz, 1 WS** \rightarrow transfers/sec = $80\text{M} / 2 = 40\text{M}$.
Bandwidth = $4 \times 40\text{M} = 160\text{MB/s}$.

6) Indicate which addresses are word aligned (address % 4 = 0)

address	aligned?	reason
(a) 0x1200004A	<input type="checkbox"/>	ends A (not 0/4/8/C)
(b) 0x52000068	<input type="checkbox"/>	ends 8
(c) 0x66000082	<input type="checkbox"/>	ends 2
(d) 0x23FFFF86	<input type="checkbox"/>	ends 6
(e) 0x23FFFFFF0	<input type="checkbox"/>	ends 0
(f) 0x4200004F	<input type="checkbox"/>	ends F
(g) 0x18000014	<input type="checkbox"/>	ends 4
(h) 0x43FFFFFF3	<input type="checkbox"/>	ends 3
(i) 0x44FFFF05	<input type="checkbox"/>	ends 5

7) Show how data is placed (little- vs big-endian)

```
LDR    R2, =0xFA98E322
LDR    R1, =0x20000100
STR    [R1], R2
```

- **Little-endian** (LSB at lowest address):
0x20000100: 0x22, 0x20000101: 0xE3, 0x20000102: 0x98, 0x20000103: 0xFA.
- **Big-endian** (MSB at lowest address):
0x20000100: 0xFA, 0x20000101: 0x98, 0x20000102: 0xE3, 0x20000103: 0x22.

8) True/False — In ARM, instructions are always word aligned.

Answer: False (as a general statement). In **ARM state** they are word-aligned, but in **Thumb state** instructions are **halfword-aligned**.

9) True/False — In a word-aligned address the lower hex digit is 0, 4, 8, or C.

Answer: True. (Those correspond to the low two bits 00.)

10)–14) Memory cycles required (32-bit bus)

Assumption: A 64-bit `LDRD` on a 32-bit bus reads **one 32-bit word per cycle**.

- If the starting address is **word-aligned**: **2 cycles**.
- Otherwise the 8-byte window crosses **three** 32-bit words: **3 cycles**.
- Halfword (`LDRH`): **1 cycle** when halfword-aligned (address % 2 = 0).
- Byte (`LDRB`): **1 cycle** at any alignment.

10)

```
LDR    R1, =0x20000004
LDRD   [R1], R2
```

Start **aligned** \Rightarrow **2 cycles**.

11)

```
LDR    R1, =0x20000102
LDRD   [R1], R2
```

Start **unaligned (...02)**; 8 bytes span three words \Rightarrow **3 cycles**.

12)

```
LDR    R1, =0x20000103
LDRD   [R1], R2
```

Start **unaligned (...03)** \Rightarrow **3 cycles**.

13)

```
LDR    R1, =0x20000006
LDRH   [R1], R2
```

Halfword **aligned (...06)** \Rightarrow **1 cycle**.

14)

```
LDR    R1, =0x20000C10
LDRB   [R1], R2
```

Byte access (any alignment) \Rightarrow **1 cycle**.

Notes for learners

- **Alignment faults** may occur on some cores for unaligned word/halfword accesses; others handle them in hardware but need extra cycles.
- Effective bandwidth is lowered by **wait states** and **unaligned** accesses—align your data when you can.

Section 6.2

Problems are paraphrased to respect copyright. Short, teachable answers below.

15) True or false. In ARM the R13 is designated as stack pointer.

Answer: True. R13 is the architectural SP (banked per mode on classic ARM).

16) When BL is executed, how many stack locations are used?

Answer: Zero. BL stores the return address in LR (R14); the stack is used **only if** your code saves LR (e.g., PUSH {LR}).

17) When B is executed, how many stack locations are used?

Answer: Zero. B is a plain branch and does **not** touch the stack.

18) In ARM, stack pointer is _____ register.

Answer: R13 (SP).

19) Describe how the return operation is performed in ARM.

Answer: Restore PC from LR. Common sequences:

```
BX    LR           ; preferred (keeps ARM/Thumb state)
; or
MOV   PC, LR       ; simple return
; if LR was saved on stack:
POP   {PC}         ; load PC from stack (also returns)
```

Prologue/epilogue pattern for subroutines that call others:

```
PUSH {LR}         ; save caller's return
...               ; body, may BL further routines
POP  {PC}         ; restore and return
```

20) Give the size of the stack in ARM.

Answer: Not fixed by the ISA. Stack size is configured by your **linker/RTOS** and limited by **RAM**. Each pushed register occupies **4 bytes** (32-bit words); the stack **grows downward** on classic ARM (full-descending).

21) In ARM, which address is saved when BL is executed?

Answer: The address of the instruction following BL (the return address) is saved into **LR (R14)**.

Notes for learners

- Many exception/privileged modes have **banked SP/LR**, so **R13/R14** can differ per mode (e.g., IRQ vs Thread).
- On Cortex-M, PUSH/POP mnemonics expand to STMDB/ LDMIA with SP and handle multiple registers in one go.

Section 6.3

Problems are paraphrased to respect copyright. I use the ARM bit-band **alias formulas** throughout.

22) Which memory regions of ARM are bit-addressable?

Answer: Two 1-MB regions:

- **SRAM bit-band region:** $0x2000_0000-0x200F_FFFF \rightarrow$ **alias** at $0x2200_0000-0x23FF_FFFF$.
- **Peripheral bit-band region:** $0x4000_0000-0x400F_FFFF \rightarrow$ **alias** at $0x4200_0000-0x43FF_FFFF$.

23) Bit-addressable SRAM alias region (generic ARM)

Answer: $0x2200_0000-0x23FF_FFFF$.

24–30) Bit addresses for a given byte address

Use $\text{bit_word} = \text{alias_base} + (\text{byte_offset} \times 32) + (\text{bit} \times 4)$ where $\text{byte_offset} = \text{byte_addr} - \text{base}$ and $\text{base} = 0x2000_0000$ (SRAM) or $0x4000_0000$ (Peripheral). The table lists **bit0...bit7** for the byte.

- **24) Byte** $0x2000_0004 \rightarrow$ **alias base** $0x2200_0000$, **offset** $0x4 \times 32 = 0x80$
bit0..7: $0x22000080, 0x22000084, 0x22000088, 0x2200008C, 0x22000090, 0x22000094, 0x22000098, 0x2200009C$.
- **25) Byte** $0x2000_0100 \rightarrow$ **offset** $0x100 \times 32 = 0x2000$
bit0..7: $0x22002000-0x2200201C$ (step 4).
- **26) Byte** $0x200F_FFFF \rightarrow$ **offset** $0x0FFFF \times 32 = 0x1FFFFE0$
bit0..7: $0x23FFFFE0-0x23FFFFFC$ (step 4).
- **27) Byte** $0x2000_0020 \rightarrow$ **offset** $0x20 \times 32 = 0x400$
bit0..7: $0x22000400-0x2200041C$ (step 4).
- **28) Byte** $0x4000_0008 \rightarrow$ **alias base** $0x4200_0000$, **offset** $0x8 \times 32 = 0x100$
bit0..7: $0x42000100-0x4200011C$ (step 4).
- **29) Byte** $0x4000_000C \rightarrow$ **offset** $0xC \times 32 = 0x180$
bit0..7: $0x42000180-0x4200019C$ (step 4).
- **30) Byte** $0x4000_0020 \rightarrow$ **offset** $0x20 \times 32 = 0x400$
bit0..7: $0x42000400-0x4200041C$ (step 4).

31) The following are bit addresses. Indicate where each one belongs (region, byte address, and bit number).

item	bit-address (alias)	Region	byte address	bit
(a)	$0x2200004C$	SRAM	$0x20000002$	3
(b)	$0x22000068$	SRAM	$0x20000003$	2
(c)	$0x22000080$	SRAM	$0x20000004$	0
(d)	$0x23FFFF80$	SRAM	$0x200FFFFC$	0
(e)	$0x23FFFF00$	SRAM	$0x200FFFF8$	0
(f)	$0x4200004C$	Peripheral	$0x40000001$	3
(g)	$0x42000014$	Peripheral	$0x40000000$	5
(h)	$0x43FFFFFF0$	Peripheral	$0x400FFFFF$	4
(i)	$0x43FFFF00$	Peripheral	$0x400FFFF8$	0

(Computed by inverting the alias formula.)

32) Of the 4GB address space, how many bytes are also assigned a bit address? Which bytes?**Answer:** 2MB of byte locations are bit-addressable:

- **SRAM bytes:** 0x2000_0000–0x200F_FFFF (1MB).
- **Peripheral bytes:** 0x4000_0000–0x400F_FFFF (1MB).

33) True/False — The bit-addressable region cannot be accessed in byte.**Answer: False.** You can access those addresses normally (byte/halfword/word) via the **original** (non-alias) addresses.**34) True/False — The bit-addressable region cannot be accessed in word.****Answer: False.** Word/halfword/byte accesses work at the original addresses; the alias is **extra** for per-bit read/write.**35) Program — Test D7 of RAM[0x2000_0020]; if high, write 1 to D1 of RAM[0x2000_0000]**

```

THUMB
; Alias addresses
LDR    r0, =0x2200041C    ; bit-band for 0x20000020 bit7
LDR    r1, =0x22000004    ; bit-band for 0x20000000 bit1

LDR    r2, [r0]           ; r2 = 0 or 1 (state of D7)
CBZ    r2, done
MOVS   r3, #1
STR    r3, [r1]           ; set D1 = 1
done:  BX    lr

```

36) Program — Test D7 of I/O 0x4000_0000; if low, write 0 to D0 of 0x400F_FFFF

```

THUMB
LDR    r0, =0x4200001C    ; 0x40000000 bit7
LDR    r1, =0x43FFFFFFE0  ; 0x400FFFFFFF bit0

LDR    r2, [r0]           ; 0 or 1
CMP    r2, #0
BNE    done
MOVS   r3, #0
STR    r3, [r1]           ; clear D0
done:  BX    lr

```

37) Set all bits high at RAM[0x2000_0000]**(a) Using byte address**

```

MOVS   r2, #0xFF
LDR    r1, =0x20000000
STRB   r2, [r1]           ; write 0xFF (D7..D0 = 1)

```

(b) Using bit addresses

```

LDR    r1, =0x22000000    ; bit0
MOVS   r2, #1
STR    r2, [r1]           ; D0 = 1
STR    r2, [r1, #4]       ; D1 = 1
STR    r2, [r1, #8]       ; D2 = 1
STR    r2, [r1, #12]      ; D3 = 1
STR    r2, [r1, #16]      ; D4 = 1
STR    r2, [r1, #20]      ; D5 = 1
STR    r2, [r1, #24]      ; D6 = 1
STR    r2, [r1, #28]      ; D7 = 1

```

38) Program — Test whether SRAM[0x2000_0000] is divisible by 8

(Unsigned 32-bit word assumed. Divisible by 8 \Leftrightarrow low three bits are zero.)

```

LDR    r0, =0x20000000
LDR    r1, [r0]
TST    r1, #7                ; mask 0b111
BEQ    is_div_by_8          ; yes if zero
; else not divisible

```

39) Explain LDM (Load Multiple)

Loads a list of registers from **consecutive memory** starting at a base address. Addressing options (IA/IB/DA/DB) control the order; optional **write-back** (!) updates the base by $4 \times (\text{\#registers})$.

40) Explain STM (Store Multiple)

Stores a list of registers to consecutive memory from a base address, with the same addressing modes and **write-back** option as LDM.

41) Difference between LDM and LDR

LDR moves **one** register; LDM transfers **many** registers in a single instruction (block transfer).

42) Difference between STM and STR

STR stores **one** register; STM stores **many** registers (block transfer).

43) LDMIA and its impact on SP

Increment-After: reads starting at [SP], then increments after each word. With write-back (LDMIA SP!, Ellipsis) it **pops** registers from a **full-descending** stack and **increases SP** by $4 \times n$.

44) LDMIB and its impact on SP

Increment-Before: first adds 4 to SP, reads from the next address. With LDMIB SP! SP still **increases by $4 \times n$** , but the first load is from SP+4. Not used for full-descending stacks.

45) STMIA and its impact on SP

Increment-After store. STMIA SP!, Ellipsis writes at [SP] upward and **increases SP** by $4 \times n \rightarrow$ corresponds to an **empty-ascending** stack (not the common ARM full-descending push).

46) STMIB and its impact on SP

Increment-Before store. STMIB SP!, Ellipsis first adds 4 to SP then stores, repeating upward; SP **increases by $4 \times n$** . (For standard ARM PUSH, prefer STMFD/STMDB SP!, Ellipsis which **decrements SP**.)

Notes for learners

- PUSH/POP aliases: PUSH {regs} \leftrightarrow STMDB SP!, {regs}; POP {regs} \leftrightarrow LDMIA SP!, {regs}.
- Bit-band alias words read back 0/1; writing any non-zero value acts as 1.

Section 6.4

Problems are paraphrased to respect copyright. Explanations are kept short and practical.

47) True or False — Write-back is by default enabled in pre-indexed addressing mode.

Answer: False. In pre-indexed form write-back happens **only** when you add ! (e.g., $[Rn, Rm]!$). Without ! it's an **offset** access (no write-back).

48) Indicate the addressing mode

- (a) `LDR R1, [R5], R2, LSL #2` → **Post-indexed**, register offset with shift (address = $[R5]$, then $R5 += (R2 \ll 2)$).
- (b) `STR R2, [R1, R0]` → **Offset / pre-indexed without write-back** (effective addr = $R1 + R0$, $R1$ unchanged).
- (c) `STR R2, [R1, R0, LSL #2]!` → **Pre-indexed with write-back** (addr = $R1 + (R0 \ll 2)$, then $R1$ updated).
- (d) `STR R9, [R1], R0` → **Post-indexed** with register offset (store at $[R1]$, then $R1 += R0$).

49) What is an ascending stack?

A stack that **grows toward higher addresses** as items are pushed; **SP increases** on push.

50) Difference between an empty and a full stack

- **Full stack:** **SP points to the last occupied location.** A push **writes before** moving away (with DB or IB depending on direction).
- **Empty stack:** **SP points to the next free location.** A push **writes at SP** then moves (with IA or DA depending on direction).

51) Store R0 in a full descending stack

```
PUSH    {{R0}}                ; alias for STMDB SP!, {R0}  (FD: pre-decrement, store)
; equivalently:
STMDB   SP!, {{R0}}
```

52) Load R9 from an empty descending stack

```
LDMIB   SP!, {{R9}}           ; ED: increment-before on pop, SP increases
; (For comparison: POP {R9} == LDMIA SP!, {R9} is for a full-descending stack.)
```

Notes for learners

- Mapping between stack names and addressing modes (store/push first):
FD ↔ **STMDB / LDMIA** ↔ **DA** ↔ **STMIB / LDMDA** ↔ **ED** ↔ **STMDA / LDMIB** ↔ **EA** ↔ **STMIA / LDMDB**

Section 6.5

Problems are paraphrased to respect copyright. Short derivations shown for each.

53) If `LDR R2, [PC, #8]` is located at address `0x300`, what memory address is accessed?

ARM state rule: $PC_effective = current_address + 8$.

Here: $PC_effective = 0x300 + 0x8 = 0x308$.

$EA = PC_effective + 0x8 = 0x308 + 0x8 = **0x310**$.

Answer: `0x00000310`.

54) Using PC-relative addressing, write an `LDR` that accesses a location `0x20` bytes ahead of itself.

We want $EA = current_address + 0x20 = (current_address + 0x8) + imm$.

Therefore $imm = 0x20 - 0x8 = 0x18$.

```
LDR    R2, [PC, #0x18]    ; accesses (this instruction address + 0x20)
```

(In Thumb state, use `#0x1C` because $PC = addr + 4$.)

Notes for learners

- `ADR Rd, label` emits a PC-relative add; `LDR Rd, =imm` is often assembled into a **literal load** via a PC-relative address.

Chapter 7

Section 7.1

Problems are paraphrased to respect copyright. Short, teachable answers below.

1) The ARM7 uses a pipeline of _____ stages.

Answer: 3 stages.

2) Give the names of the pipeline stages in the ARM7.

Answer: Fetch → Decode → Execute.

3) The ARM9 uses a pipeline of _____ stages.

Answer: 5 stages.

4) Give the names of the pipeline stages in the ARM9.

Answer: Fetch → Decode → Execute → Memory → Write-back.

Notes for learners

- ARM7's 3-stage design keeps things simple but limits clock speed.
- ARM9 separates the **memory** access and **write-back** phases, enabling better overlap and higher frequencies (with hazards handled by the core).

Section 7.2

Problems are paraphrased to respect copyright. Short, teachable answers follow.

5) The number of pipeline stages in a superpipeline system is _____ (less, more) than in a superscalar system.

Answer: more. Superscalar splits work into **more, shorter stages** to raise the clock; superscalar focuses on **parallel units** rather than stage count.

6) Which has one or more execution units, superpipeline or superscalar?

Answer: Superscalar. It issues to **multiple functional units** per cycle (ALUs, load/store, etc.).

7) Which part of on-chip cache in ARM is write-protected, data or code?

Answer: Code (instruction) cache. The CPU doesn't write instructions directly; fills/evictions happen via the memory system, while data cache is writable by stores.

8) What is instruction pairing, and when can it happen?

Answer: Pairing means **issuing/executing two instructions in the same cycle**. It occurs on **superscalar** cores when the two instructions are **independent**, target **different execution units/ports**, and satisfy alignment/resource rules (no hazards).

9) What is data dependency, and how is it avoided?

Answer: A situation where one instruction **needs the result** of another (RAW) or conflicts on destinations (WAW) or sources (WAR). Avoided by **reordering, register renaming, forwarding/bypassing**, or inserting **stalls** when necessary.

10) True/False — Instructions are fetched according to the order in which they were written.

Answer: True. Fetch is in **program order** (subject to branch prediction).

11) True/False — Instructions are executed according to the order in which they were written.

Answer: False. Modern CPUs may **execute out of order** to hide latencies.

12) True/False — Instructions are retired according to the order in which they were written.

Answer: True. **In-order retirement (commit)** preserves precise exceptions and architectural state.

13) The visible registers R0, R1, ... are updated by which unit of the CPU?

Answer: The **write-back/retire stage** (register file write-back).

14) True/False — Among the instructions, STR (store) operations are never executed out of order.

Answer: False. Stores may be **issued/out-of-order** and buffered; however, they are typically **made visible (committed) in order** to maintain memory consistency.

Notes for learners

- Two orthogonal levers for speed: **deeper pipelines** (superpipeline) and **wider issue** (superscalar).

- Out-of-order execution + in-order retirement is the common combination in ARM performance cores.