Chapter 6 · Section 6.3 — Exercises (Mazidi)

2025-09-01

Chapter 6 · Section 6.3 — Exercises (Mazidi)

> 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` → **alias** at `0x2200_0000–0x23FF_FFFF`.
- **Peripheral bit-band region:** `0x4000_0000–0x400F_FFFF` → **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 `bit_word = alias_base + (byte_offset × 32) + (bit × 4)` where `byte_offset = byte_addr − base` and `base =` 0x2000_0000 (SRAM) or 0x4000_0000 (Peripheral). The table lists **bit0…bit7** for the byte.

- **24)** Byte `0x2000_0004` → alias base `0x2200_0000`, offset `0x4×32=0x80`
  bit0..7: `0x22000080, 0x22000084, 0x22000088, 0x2200008C, 0x22000090, 0x22000094, 0x22000098, 0x2200009C`.

- **25)** Byte `0x2000_0100` → offset `0x100×32=0x2000`
  bit0..7: `0x22002000–0x2200201C` (step 4).

- **26)** Byte `0x200F_FFFF` → offset `0x0FFFFF×32=0x1FFFFE0`
  bit0..7: `0x23FFFFE0–0x23FFFFFC` (step 4).

- **27)** Byte `0x2000_0020` → offset `0x20×32=0x400`
  bit0..7: `0x22000400–0x2200041C` (step 4).

- **28)** Byte `0x4000_0008` → alias base `0x4200_0000`, offset `0x8×32=0x100`
  bit0..7: `0x42000100–0x4200011C` (step 4).

- **29)** Byte `0x4000_000C` → offset `0xC×32=0x180`
  bit0..7: `0x42000180–0x4200019C` (step 4).

- **30)** Byte `0x4000_0020` → offset `0x20×32=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)	`0x43FFFFF0`	Peripheral	`0x400FFFFF`	4
(i)	`0x43FFFF00`	Peripheral	`0x400FFFF8`	0

(Computed by inverting the alias formula.)

---

32) Of the 4 GB address space, how many bytes are also assigned a bit address? Which bytes?

Answer: 2 MB of byte locations are bit-addressable:

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

---

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, =0x43FFFFE0     ; 0x400FFFFF 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 ⇔ **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 × (#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×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×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×n` → 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×n**. (For standard ARM PUSH, prefer `STMFD/STMDB SP!, Ellipsis` which **decrements SP**.)

Notes for learners

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