UNIVERSITY ELECTRICAL ENGINEERING • EXERCISE SHEET

8.2 Embedded C and assembly

Name: _____ Class: _____ Date: _____ **Total: 21 marks**

KEY WORDS volatile • read-modify-write - -

Objective

Read and write register-level embedded C, trace C constructs through their assembly, and verify every fragment by its exact state changes and bit-width limits.

1 Key knowledge and worked examples

■ volatile, masks, and the register contract

Registers need `volatile` (the value changes outside the program's control, and the compiler must not cache or delete the access). Bit operations are the vocabulary: set (`|= mask`), clear (`&= ~mask`), toggle (`^= mask`), test (`& mask`).

■ Worked example: one register, four operations

```
volatile uint32_t *const PORT = (uint32_t *)0x2000_0100;
*PORT |=  (1u << 3);   // set bit 3:      0x00000000 -> 0x00000008
*PORT &= ~(1u << 5);   // clear bit 5:    0x00000028 -> 0x00000008
*PORT ^=  (1u << 3);   // toggle bit 3:   0x00000008 -> 0x00000000
if (*PORT & (1u << 5)) // test bit 5:     0x00000000 -> false
```

The state trace is the verification: each line's before/after is computable by hand, and the final test reads the register AFTER the operations – 0x00, bit 5 clear. **Read-modify-write** - - is the pattern that makes set and clear safe on the OTHER bits (a bare $*PORT = 0x08$ would clobber them).

■ The C-to-assembly map

$count = count + 1$ is load–add–store; `while(1)` is an unconditional branch; an `if` is a conditional branch. A 32-bit counter at 84 MHz wraps in $(2^{32})/84 \times 10^6 = 51.1s$ – the trace and the clock arithmetic must agree on the same program.

2 Practice

2.1 Write one line to set bits 4 and 5 of `*REG` without touching other bits; one line to clear bit 1. [2]

2.2 A 16-bit register field occupies bits [10:8]. Extract it into a variable `mode` (value 0–7). [3]

2.3 Why does removing `volatile` break a polling loop even though the code is "the same"? [2]

3 Exam-style questions

3.1 A GPIO port: DIR at 0x2000_0100 (bit n = 1 means output), OUT at 0x2000_0104, IN at 0x2000_0108 – all 32-bit, all volatile.

Write C that (a) configures pin 2 as output and pins 0–1 as inputs (DIR bits 0 and 1 cleared, bit 2 set – other bits untouched), (b) reads the two input pins into a variable, (c) writes the output pin HIGH if BOTH inputs are 1. Trace DIR and OUT through the sequence for the case both inputs read 1, starting from DIR = 0xFFFFFFFF, OUT = 0x00000000. [7]

3.2 Translate this C loop into its instruction-by-instruction behaviour (state the registers and the state trace for three iterations), then compute the total cycles for 100 iterations at 4 cycles per iteration, and the elapsed time at 84 MHz:

```
uint32_t delay = 3;
while (delay) { delay = delay - 1; }
```

State what the compiler may legally do to this loop at optimisation −O2 and what a hardware-based delay would do differently. [7]

Solutions

2.1

- `*REG |= (3u << 4);` (bits 4 and 5 = mask 0x30).
- `*REG &= ~(1u << 1);`.

2.2

- `uint32_t mode = (*REG >> 8) & 0x7;` (shift into place, then mask 3 bits).

2.3

- Without `volatile`, the compiler may hoist the register load out of the loop (the value "cannot change" as far as it knows) – the loop then reads a cached copy forever and never sees the hardware update.

3.1

```
volatile uint32_t *const DIR = (uint32_t *)0x20000100;
volatile uint32_t *const OUT = (uint32_t *)0x20000104;
volatile uint32_t *const IN  = (uint32_t *)0x20000108;

*DIR &= ~((1u << 0) | (1u << 1));  // pins 0,1 input
*DIR |=  (1u << 2);                // pin 2 output
uint32_t inputs = *IN & 0x3;
if (inputs == 0x3) *OUT |= (1u << 2); else *OUT &= ~(1u << 2);
```

- Trace: DIR starts 0xFFFFFFFF. After (a): clear bits 0,1 → 0xFFFFFFFC; set bit 2 (already 1) → 0xFFFFFFFC.
- Both inputs 1: $inputs = 0x3$, so `*OUT |= 4`: OUT 0x00000000 → 0x00000004.

3.2

- Instructions per iteration: load `delay` (register r0), compare/branch if zero, subtract 1, store, branch back.
- Trace: r0 = 3 → branch not taken → r0 = 2 (stored) → loop → r0 = 1 → r0 = 0 → loop exits on the next compare. Three iterations of the body.
- Cycles: 3 iterations × 4 = 12 cycles (plus the final failing check ≈ 2) ≈ 14 cycles.
- Time at 84 MHz: $14/(84 \times 10^6\,\mathrm{s}^{-1}) \approx 0.17\ \mu s$.
- At −O2 the compiler may DELETE the loop entirely (the result $delay = 0$ is computable at compile time – the loop has no observable effect on volatile state). A hardware delay (timer countdown) cannot be optimised away because the counter is a volatile register the compiler must respect.