BIP Selection Written Exam

C Programming

SOLUTIONS

Full name		**Group**	
Programme / year		**Signature**	

Time allowed: 90 minutes. Total: 100 points. Answer directly on this sheet, in the marked spaces; if you run out of space, continue on the back, stating the problem and the item. No materials are allowed.

Conventions: `unsigned char` is 8 bits, `int` and `unsigned int` are 32 bits. Bits are numbered from 0, bit 0 being the least significant (rightmost) bit. Hexadecimal values are written with the `0x` prefix and at least two digits (for example `0x0B`); binary values are written on 8 bits, with leading zeros (for example `0000 1011`). Code must be standard C, without library functions (other than those already present in the programs).

For the examiner only	**P1 (35 p)**	**P2 (35 p)**	**P3 (30 p)**	**Total (100 p)**
Points				

Problem 1: Number bases and bitwise operators (35 p)

Consider the declaration `unsigned char x = 0xB4;`. In items a) and b), each expression is evaluated separately, with `x = 0xB4`.

a) (4 p) Write the value of x in binary (8 bits, with leading zeros) and in decimal.

Binary (8 bits)	**Decimal**
1011 0100	**180**

b) (10 p) Write the value of each expression in hexadecimal, according to the rules of the C language.

No.	**Expression**	**Result (hex)**
1	`x & 0x0F`	**0x04**
2	`x | 0x01`	**0xB5**
3	`x ^ 0xFF`	**0x4B**
4	`x >> 4`	**0x0B**
5	`(unsigned char)(x << 1)`	**0x68**

c) (12 p) Complete the function bodies without `if` statements and without loops. It is guaranteed that `0 <= n <= 31`. `set_bit` and `clear_bit` return the value of v with only bit n changed, all other bits unchanged; `test_bit` returns exactly 1 or exactly 0. Then fill in the returned values, in the base indicated.

```
/* sets bit n to 1 */
unsigned int set_bit(unsigned int v, int n) {
    return v | (1u << n);
}
/* clears bit n to 0 */
unsigned int clear_bit(unsigned int v, int n) {
    return v & ~(1u << n);
}
/* returns 1 if bit n is 1, otherwise 0 */
int test_bit(unsigned int v, int n) {
    return (v >> n) & 1;
}
```

Call	**Returned value**
`set_bit(0xB4, 0)`, hex	**0xB5**
`clear_bit(0xB4, 2)`, hex	**0xB0**
`test_bit(0xB4, 7)`, decimal	**1**

d) (9 p) Complete the function that returns the number of bits equal to 1 in the 32-bit representation of v. You may use loops and bitwise operators; the operators %, / and * are not allowed. Then write the returned value, in decimal.

```
int count_ones(unsigned int v) {
    int c = 0;
    while (v != 0) {
        c += v & 1;
        v >>= 1;
    }
    return c;
}
```

Call	Returned value
count_ones(0xB4)	**4**

Problem 2: A 4-byte message (35 p)

A program receives messages as an array `unsigned char m[4]`. The bits of a byte are numbered from 7 (most significant) to 0 (least significant). Message layout:

Byte	Content
`m[0]`	bits 7...4: command code (0...15); bits 3...0: sequence number (0...15)
`m[1]`, `m[2]`	an unsigned 16-bit number: `m[1]` holds bits 15...8 and `m[2]` holds bits 7...0
`m[3]`	check byte; the message is correct if `m[3]` equals `m[0] ^ m[1] ^ m[2]`

Example message: `0x23 0x01 0x2C 0x0E`.

a) (8 p) Write one C expression, in terms of `m[0]`, whose value is the command code, and one whose value is the sequence number. Then write their values, in decimal, for the example message.

Field	C expression	Value (example)
Command code	**`m[0] >> 4`**	**2**
Sequence number	**`m[0] & 0x0F`**	**3**

b) (9 p) Complete the function that returns the 16-bit number defined above, formed by `m[1]` and `m[2]`. Then write the returned value for the example message.

```
unsigned int get_value(const unsigned char m[]) {
    return (m[1] << 8) | m[2];   /* or: m[1] * 256 + m[2] */
}
```

Value for the example message (decimal)	**300 (0x012C)**

c) (10 p) Complete the function that returns exactly 1 if the message is correct and exactly 0 otherwise. Then complete the calculation for the example message, step by step, in hexadecimal.

```
int is_valid(const unsigned char m[]) {
    unsigned char c = m[0] ^ m[1] ^ m[2];
    return c == m[3];
}
```

Step	Operation	Result (hex)
1	`m[0] ^ m[1]` =	**0x22**
2	result of step 1 `^ m[2]` =	**0x0E**
3	Is the message valid? (YES / NO)	**YES (`0x0E == m[3]`)**

d) (8 p) Complete the function that returns the byte `m[0]` for command code `cmd` and sequence number `nr`. It is guaranteed that `0 <= cmd <= 15` and `0 <= nr <= 15`. Then write the value returned by the given call, in hexadecimal.

```
unsigned char make_header(
        unsigned char cmd, unsigned char nr) {
    return (cmd << 4) | (nr & 0x0F);
}
```

Call	Returned value (hex)
`make_header(5, 9)`	**0x59**

Problem 3: The Sieve of Eratosthenes with bits (30 p)

The Sieve of Eratosthenes finds the primes less than `n` by marking every number that is not prime (0, 1 and the composite numbers). Each number `k >= 0` is stored as a single bit of the array `unsigned char c[]`: bit `k % 8` of byte `c[k / 8]`. A 1 bit means the number is marked (not prime), a 0 bit means unmarked. Before use, all bytes of `c` are 0 and the array has enough bytes for the numbers used.

a) (6 p) Write one C expression, in terms of `k`, equal to the byte index (`k / 8`), and one equal to the bit position (`k % 8`), without the operators `/`, `%` and `*`. Then write their values, in decimal, for `k = 13` and `k = 37`.

Task	Byte index	Bit position
C expression	**k >> 3**	**k & 7**
`k = 13`	**1**	**5**
`k = 37`	**4**	**5**

b) (6 p) Complete the functions without the operators `/` and `%`. `mark` sets the bit of `k` to 1 without changing any other bit of the array. `is_marked` returns exactly 1 if `k` is marked and exactly 0 otherwise.

```
void mark(unsigned char c[], int k) {
    c[k >> 3] |= 1 << (k & 7);
}
int is_marked(const unsigned char c[], int k) {
    return (c[k >> 3] >> (k & 7)) & 1;
}
```

c) (8 p) Fill in each blank (1), (2), (3) with a single C expression so that, for any `n`, after the call the marked numbers are exactly the non-primes less than `n`. Condition (1) must be true only for the values of `i` that are prime. For (2), choose the largest starting value of `j` for which the result is still correct. Then write, in hexadecimal, `c[0]` and `c[1]` after the call `sieve(c, 16)` on a 2-byte array initially 0.

```
void sieve(unsigned char c[], int n) {
    int i, j;
    mark(c, 0);
    mark(c, 1);
    for (i = 2; i * i < n; i++)
        if ( ____(1)____ )
            for (j = ____(2)____; j < n; j += ____(3)____)
                mark(c, j);
}
```

Blank	Your answer
(1)	**!is_marked(c, i)**
(2)	**i * i**
(3)	**i**
`c[0]` after `sieve(c, 16)` (hex)	**0x53**
`c[1]` after `sieve(c, 16)` (hex)	**0xD7**

d) (10 p) The program below should print how many primes there are in the interval [2, N - 1] (for `N = 100` the correct result is 25). It contains exactly 5 errors, each on a different line. For each one, give the line number, briefly explain the error and write the fix, which changes only that line.

```
1  #include <stdio.h>
2  #define N 100
3  int main(void) {
4      unsigned char c[N / 8] = {0};
5      int i, j, cnt;
6      for (i = 2; i * i < N; i++)
7          if (c[i >> 3] & (1 << (i & 7)) == 0)
8              for (j = i * i; j < N; j += i)
9                  c[j >> 3] = 1 << (j & 7);
10     for (i = 2; i < N; i++)
11         if (c[i >> 3] & (1 << (i & 7)))
12             cnt++;
13     printf("%d\n", cnt);
14     return 0;
15 }
```

No.	Line	Error (short explanation)	Fix
1	**4**	**the array has 12 bytes, but number 99 needs byte 12, so 13 bytes**	`unsigned char c[(N + 7) / 8] = {0};`
2	**5**	**`cnt` is not initialised**	`int i, j, cnt = 0;`
3	**7**	**`==` has higher precedence than `&`**	`if ((c[i >> 3] & (1 << (i & 7))) == 0)`
4	**9**	**`=` overwrites the byte and erases earlier marks**	`c[j >> 3] |= 1 << (j & 7);`
5	**11**	**the condition counts the marked (composite) numbers, not the primes**	`if (!(c[i >> 3] & (1 << (i & 7))))`