

CSC 209 Review 9 Solution

September 2, 2020

1. a) 0

Notes

- a) is 0 because $(i \gg 1 + j \gg 1 = i \gg 10 \gg 1 = 0)$
- **Bitwise Shift Operators**
 - has lower precedence than arithmetic operators

Example:

$i \ll 2 + 1$ means $i \ll (2+1)$ and not $(i \ll 2) + 1$

- \ll : Left Shift
- \gg : Right Shift
- *Tip*: Always shift only on unsigned numbers for portability

Example

```
unsigned short i, j;

i = 13;          /* i is now 13 (binary 0000000000001101) */
j = i << 2;      /* j is now 52 (binary 0000000000110100) */
j = i >> 2;      /* j is now 3 (binary 0000000000000011) */
```

As these examples show, neither operator modifies its operands. To modify a variable by shifting its bits, we'd use the compound assignment operators $\ll=$ and $\gg=$:

```
i = 13;          /* i is now 13 (binary 0000000000001101) */
i <<= 2;         /* i is now 52 (binary 0000000000110100) */
i >>= 2;         /* i is now 13 (binary 0000000000001101) */
```

Shifts to left

Shifts to right

- $\gg=$ / $\ll=$: Are bitwise shift equivalent of $+=$

b) 0

Notes

- `i` is 1111111111111111
- `i` is 0000000000000000
- so `i & i = 0`
- `~`: Bitwise complement (NOT)

a	$\sim a$
0	1
1	0

Example:

```

1      0   1   1   1   //<- this is 7
2      -----
3      1   0   0   0   //<- this is 8
4
5      so, ~ 7 = 8

```

- `&`: Bitwise *and*

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```

1      0   1   1   1   //<- this is 7
2      0   1   0   0   //<- this is 4
3      -----
4      0   1   0   0   //<- this is 4
5
6      so, 7 & 4 = 4

```

- `^`: Bitwise *exclusive or*
- `|`: Bitwise *inclusive or*

c) 1

Notes

- `i` is 1111111111111110
- `j` is 0000000000000000
- `i & j` is 0000000000000000 or 1
- `i & j ^ k` is 1

- \wedge : Bitwise XOR

a	b	$a \wedge b$
0	0	0
0	1	1
1	0	1
1	1	0

Example:

```

1      0   1   1   1   //<- this is 7
2      0   1   0   0   //<- this is 4
3      -----
4      0   0   1   1   //<- this is 3
5
6      so,  $7 \wedge 4 = 3$ 
7

```

d) 0

Example

- i is 0000000000000111
- j is 0000000000001000
- $i \wedge j$ is 0000000000000000 or 0
- k is 0000000000001001
- $i \wedge j \& k$ is 0000000000000000 or 0

Correct Solution

15

Notes

- There is a precedence to the order of operations

Highest:	\sim
	$\&$
	\wedge
Lowest:	$ $

2. • toggling from 0 to 1

```
i = 0x0000;  
i |= 0x0001;
```

or

```
i |= 1 << 0; where i = 0x0000;
```

- toggling from 1 to 0

```
i = 0x0001;  
i &= ~0x0001;
```

or

```
i &= ~(1 << 0); where i = 0x0001;
```

Correct Solution

- toggling from 0 to 1 of 4th bit

```
i = 0x0010;  
i ^= 0x0000;
```

or

```
i ^= 1 << 4; where i = 0x0000;
```

- toggling from 1 to 0 of 4th bit

```
i = 0x0010;  
i ^= 0x0010;
```

or

```
i ^= (1 << 4); where i = 0x0010;
```

Notes

- Toggling can be done using bitwise XOR
- **Setting a bit**
 - Is done using `|` or bitwise OR

```
i = 0x0000;          /* i is now 0000000000000000 */
i |= 0x0010;         /* i is now 00000000000010000 */
```

– The idiom of above is $i \mid= 1 \ll j$

- **Clearing a bit**

– Is done using $|$ or bitwise AND

```
i = 0x00ff;          /* i is now 0000000011111111 */
i &= ~0x0010;        /* i is now 0000000011101111 */
```

– The idiom of above is $i \&= \sim(i \ll j)$

3. It swaps the elements between x and y .

Notes

- Preprocessor performs operations of statements in order from left to right

1	2	3
<pre>#define M(x,y) ((x)^(y), (y)^(x), (x)^(y))</pre>		
New value of x	New value of y, using x from 1	New value of x, using y from 2, x from 1

4. `#define MK_COLOR(r,g,b) (long) ((b | (g << 8)) | (b | (r << 16)))`

Rough Work

1. store b in bit 0

b

2. store g in bit 8

$b \mid g \ll 8$

3. store r in bit 16

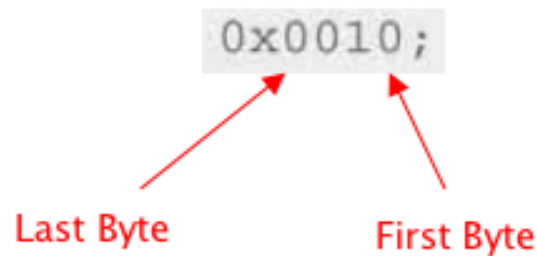
$b \mid r \ll 16$

Correct Solution

```
#define MK_COLOR(r,g,b) (long) ((r | (g << 8)) | (r | (b << 16)))
```

Notes

- First Byte is furthest from 0x and first byte is closest to 0x



5. • GET_RED

```
#define GET_RED(c) (long) (c & 0x007)
```

- GET_GREEN

```
#define GET_GREEN(c) (long) ((c >> 8) & 0x007)
```

- GET_BLUE

```
#define GET_BLUE(c) (long) ((c >> 16) & 0x007)
```

Notes

- 0x0007 in binary is 0x0000000000001111
- $c \gg 4$ shifts c to right by 4 bits and return overlapping value between $c \gg 4$ and 0x0000000000001111 (0x007)
- Test code is below

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MK_COLOR(r,g,b) (long) ( (r | (g << 8)) | (r | (b << 16)) )
5  #define GET_RED(c) (long) (c & 0x007)
6  #define GET_GREEN(c) (long) ((c >> 8) & 0x007)
7  #define GET_BLUE(c) (long) ((c >> 16) & 0x007)
```

```

8
9  int main() {
10     long i, r = 4, g = 5, b = 6, r2, g2, b2;
11
12     i = MK_COLOR(r,g,b);
13
14     r2 = GET_RED(i);
15     g2 = GET_GREEN(i);
16     b2 = GET_BLUE(i);
17
18     printf("%ld\n", i);
19     printf("%ld\n", r2);
20     printf("%ld\n", g2);
21     printf("%ld\n", b2);
22
23     return 0;
24 }

```

6. a)

```

2  unsigned short swap_bytes(unsigned short i) {
3      unsigned short j, k;
4
5      j = i & 0x007; // extract first byte
6
7      i = i >> 4;
8      k = i & 0x007; // extract second byte
9
10     i = i >> 4; // shift down layer two bytes
11
12     i |= j << 8; // add first byte to position of fourth byte
13     i |= k << 12; // add second byte to position of third byte
14 }

```

b)

```

2  unsigned short swap_bytes(unsigned short i) {
3
4      i = i >> 8 | i << 8;
5
6      return i;
7  }

```

Rough Works

1. Extract first two bytes

```

j = i & 0x0007
i = i >> 4
k = i & 0x0007

```

2. Shift later two bytes down

```
i = i >> 4
```

3. Add first two bytes to last two bytes

```
i |= j << 8;
i |= k << 12;
```

```
71 unsigned int rotate_left(unsigned int i, int n) {
72     return i >> 28 | i << n;
73 }
74
75 unsigned int rotate_right(unsigned int i, int n) {
76     return i << 28 | i >> n;
77 }
```

8. a) Is a binary with n many 1s from the first bit
 b) Extracts last n bits in i

Correct Solution

- a) Is a binary with n many 1s from the first bit
 b) Extracts last n bits **starting from position m** in i

```
9. a) unsigned int count_ones(unsigned char *ch)
72 {
73     unsigned char *p;
74     unsigned int count;
75
76     for (p = ch; *p != '\0'; p++){
77         if (*p == '1') {
78             count++;
79         }
80     }
81
82     return count;
83 }
```

```
b) unsigned int count_ones(unsigned char ch)
2 {
3     int sum = 0;
4
5     sum += (i >> 0) & 1;
6     sum += (i >> 1) & 1;
```



```

7         sum += (i >> 2) & 1;
8         sum += (i >> 3) & 1;
9         sum += (i >> 4) & 1;
10        sum += (i >> 5) & 1;
11        sum += (i >> 6) & 1;
12        sum += (i >> 7) & 1;
13        return count;
14    }

```

Notes

- Unsigned char goes from 0 (00000000) to 255 (11111111)
- I am having trouble how to convert from loop to without loop :'. I need help
- Example

100010101 - Here there are 4 1s.

```

101 unsigned int reverse_bits (unsigned int n)
102 {
103     int m = 15, p = 0;
104     unsigned int byte_left_end, byte_right_end, res = 0;
105
106     while (m > p) {
107         byte_left_end = n >> m & 1;
108         byte_right_end = n >> p & 1;
109
110         res |= byte_left_end << (15 - m) | byte_right_end << (15 - p
111     );
112
113         m--;
114         p++;
115     }
116     return res;
117 }

```

Rough Work

- Start at $n = 15, m = 0$
- Swap bit at n with m
- Repeat until $m > n$

Notes

- unsigned int has 4 bytes or (0x0000) or (0000000000000000, 16 bits)

11. The precedence of `&`, `^`, and `—` is lower than the precedence of the relational and equality operators.

As a result, the expression will first evaluate

```
(SHIFT_BIT | CTRL_BIT | ALT_BIT) == 0
```

first.

The work around is to put parenthesis around `key_code & (SHIFT_BIT | CTRL_BIT | ALT_BIT)`.

12. The function tries to insert `low_byte` to `high_byte` after shifting `high_byte` by 8 bits to left.

It doesn't work because `+` sign takes precedence over `<<`.

To fix this issue, a parenthesis is required around `high_byte << 8`.

13. It reduces the value of `n` by 1, then uses bitwise operator (`&`) to extract the common bits between `n` and `n-1`.

Correct Solution

```
14 struct float {
15     unsigned int sign: 1;
16     unsigned int exponent: 8;
17     unsigned int : 0;
18     unsigned int fraction: 23;
19 }
```

Correct Solution

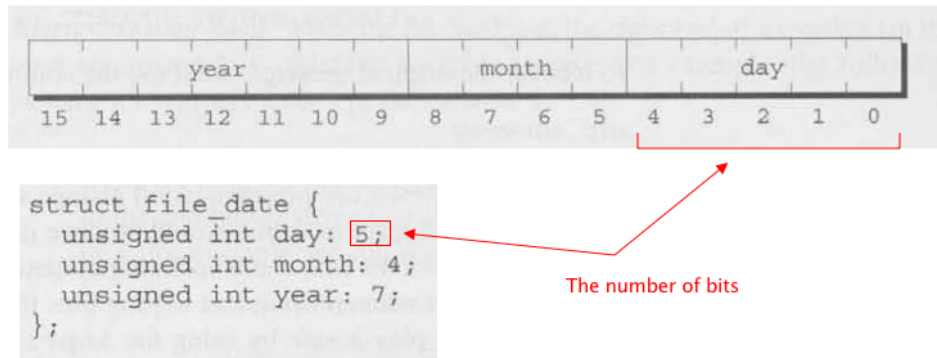
```
1 struct float {
2     unsigned int fraction: 23;
3     unsigned int exponent: 8;
4     unsigned int sign: 1;
5 }
```

Notes

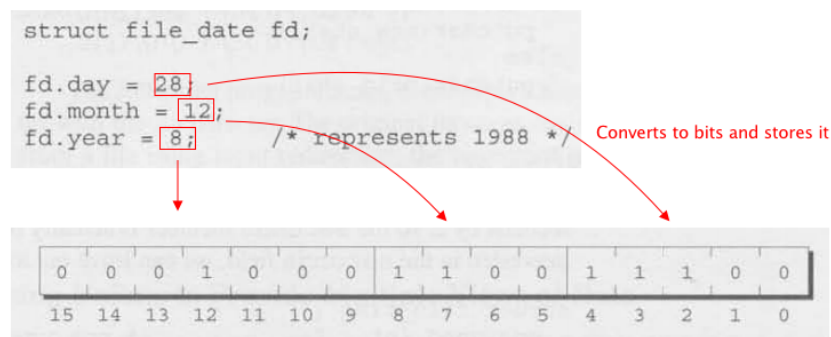
- Bit-Fields in Structures

- Bit-fields are tricky and potentially confusing

Example



- Type of bit-field must be either `int`, `unsigned int`, `signed int`
 - * `int` is ambiguous
 - * Author suggests to declare all bit-fields to be either `unsigned int` or `signed int`
- Assignment of bit-field in structure

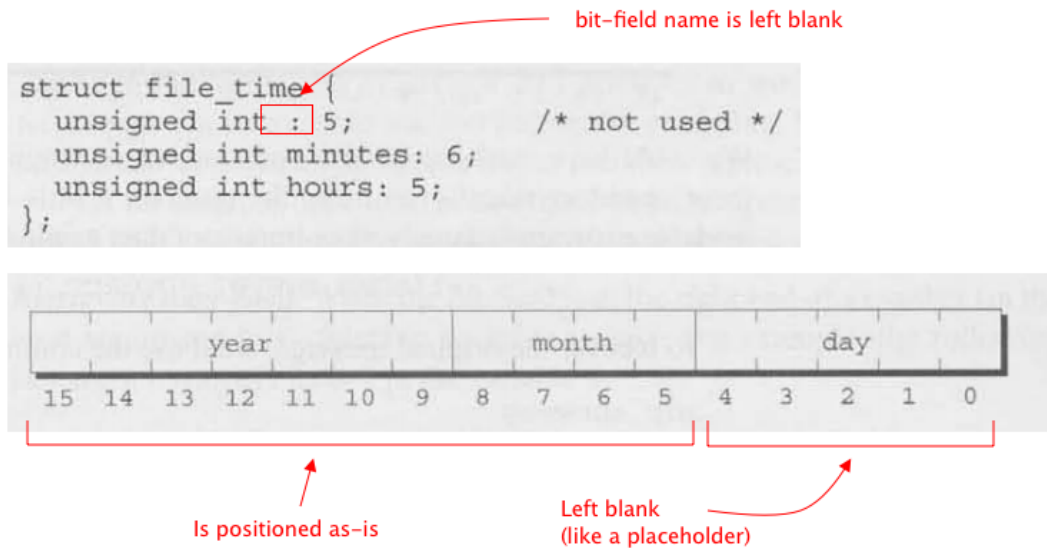


WARNING bit-fields don't have addresses

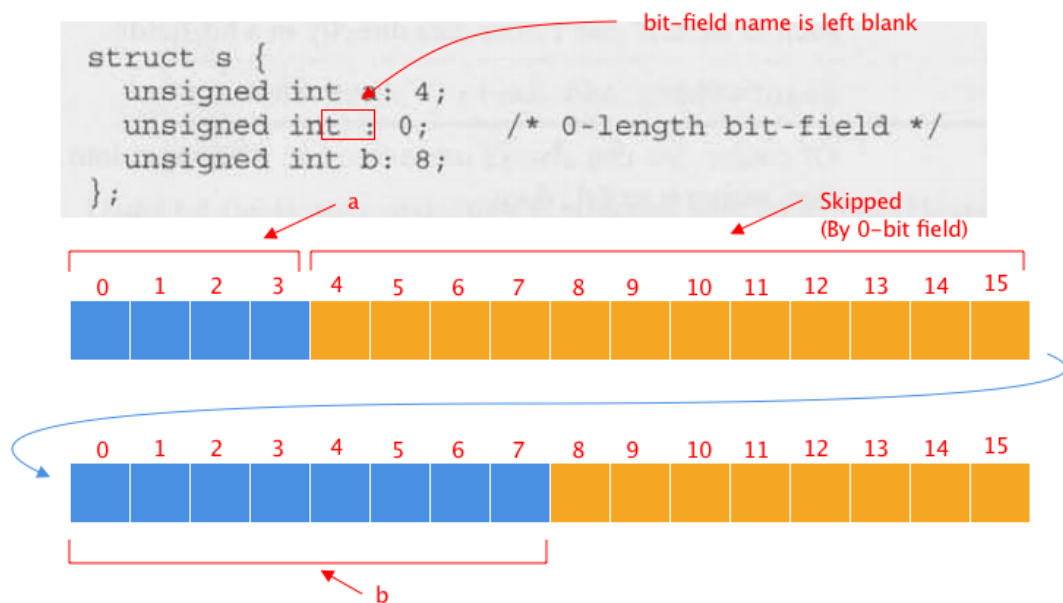
- * C **doesn't** allow address operator `&` to a bit field

• How Bit-Fields Are Stored

- C allows to omit the name of bit-field.
- Bit-field without name acts as a padding



- 0 bit-field padding cause the next field to be aligned on the next container boundary
- 0 bit-field must be unnamed



15. a) Some compilers print -1 instead of 1 because it's value of sign is set at different value.
 b) To avoid the problem, use **unsigned int** instead of **int** in **struct**.

Correct Solution

- a) Some compilers print -1 instead of 1 because it's value of sign is set at different value.

- b) To avoid the problem, use `unsigned int` instead of `int` in `struct`. This will allow flag to have value 0 and 1.

Another way is by adding an extra member `sign` of size 1 to `struct`. This will allow flag to have value -1, 0 and 1.

```

1 struct float {
2     int sign: 1;
3     int flag: 1;
4 }

```

16. Notes

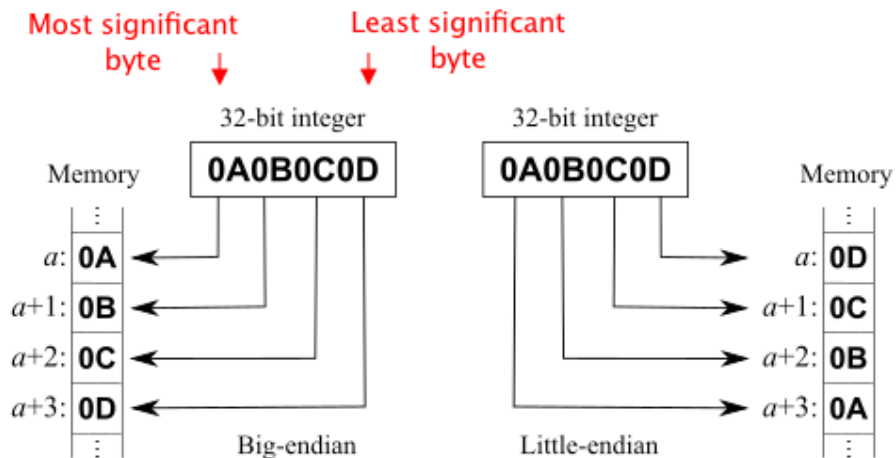
• Other Low Level Technique

– Using Unions to Provide Multiple Views of Data

- * Unions are used in C for an entirely different purpose: viewing a block of memory in two or more different ways
 - This is useful for **registers**

– Little Endian / Big Endian

- * **Little Endian** stores most significant byte (closest to 0x) of a word at the smallest memory address (address closest to 0)
- * **Big Endian** stores the least-significant byte at the smallest address (address closest to 0)



– AX / BX / CX / DX

- * Are called **Data registers**
 - Stores data for processing without access to memory

- * Are used to speed up processor operations ^[1]
 - Brute force method reads data from and storing data into memory
 - Brute force method sends data to control bus & memory storage unit
 - Brute force method slows down the processor

AX (Primary Accumulator):

- Is used in input/output and most arithmetic instructions

BX (Base Register):

- Used in indexed addressing

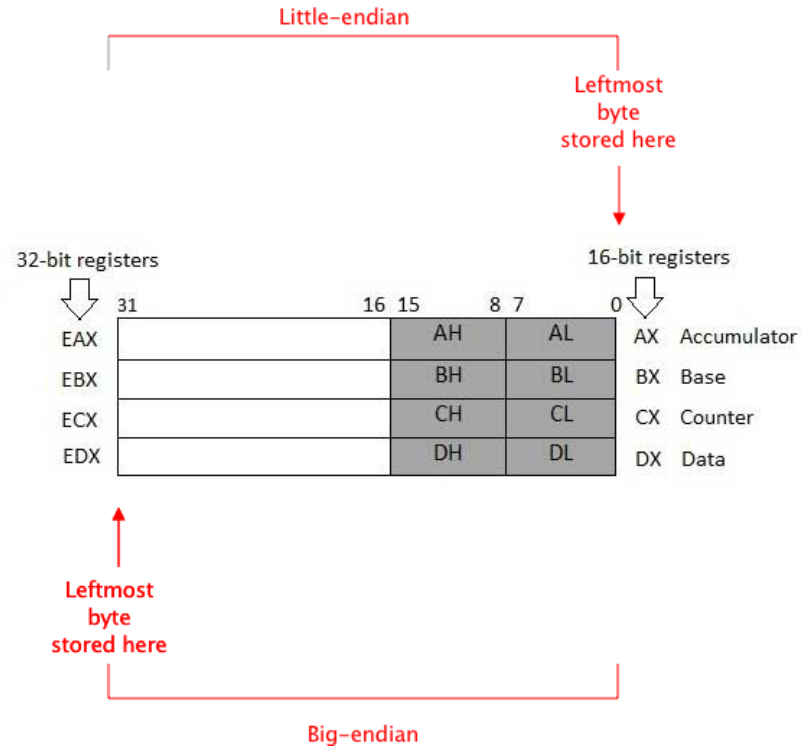
CX (Count Register):

- Stores loop count in iterative operations

DX (Data Register):

- Is also used for Input/Output operations.
- Is used with AX register for multiply and divide operations involving large values

```
union {  
    struct {  
        WORD ax, bx, cx, dx;  
    } word;  
    struct {  
        BYTE al, ah, bl, bh, cl, ch, dl, dh;  
    } byte;  
} regs;
```



Example

References

- 1) TutorialsPoint, Assembly - Registers, [link](#)