Assignment 1 - SOLUTION

Assignment 1 - Objectives:

- Conversion
- Unsigned and signed arithmetic operations and overflow
- C programming, endian and bit-level manipulation

Show your work (as illustrated in lectures).

If you write your answers by hand (as opposed to using a computer application to write them), when uploading your answer for each question, please, do not take photos (no .jpg) of your answers even if Crowdmark says so below! Scan them instead! Why? Because photos are often difficult to read. Scanning produces pdf documents of better quality, hence easier to read, hence easier to mark! :)

Marking scheme:

- This assignment will be marked as follows:
  - Questions 1 and 2 will be marked for correctness.
  - The program for Question 3 will be tested for correctness, robustness and whether all the requirements are satisfied.
- The amount of marks for each question is indicated as part of the question.
- A solution will be posted on Monday after the due date.

Due: Friday January 20 at 23:59:59 on Crowdmark (for Q. 1, Q. 2 and Q. 3) and CourSys (for Q. 3)

Late assignments will receive a grade of 0, but they will be marked (if they are submitted before the solutions are posted on Monday) in order to provide feedback to the student.

Enjoy!

Q1 a.

/ 2

Conversion

Convert each of the unsigned decimal values below into its corresponding binary value (w = 8), then convert the binary value into its corresponding hexadecimal value.

I. 157_{10}

II. 248_{10}

i. 157_{10} → using division

$157 \div 2 = 78$ R 1
78 = 39 R 0
39 = 19 R 1
19 = 9 R 1
9 = 4 R 1
4 = 2 R 0
2 = 1 R 0
1 = 0// R 1 MSbit

→ using subtraction

$157 - 128\ (2^7) = 29$
$29 - 16\ (2^4) = 13$
$13 - 8\ (2^3) = 5$
$5 - 4\ (2^2) = 1$
$1 - 1\ (2^0) = 0$//

$157_{10} = \underline{1001}\,\underline{1101}_2 = 9D_{16}$ (0x9D)

//

ii. 248_{10} → using division

$248 \div 2 = 124$ R 0
124 = 62 R 0
62 = 31 R 0
31 = 15 R 1
15 = 7 R 1
7 = 3 R 1
3 = 1 R 1
1 = 0// R 1 MSbit

→ using subtraction

$248 - 128\ (2^7) = 120$
$120 - 64\ (2^6) = 56$
$56 - 32\ (2^5) = 24$
$24 - 16\ (2^4) = 8$
$8 - 8\ (2^3) = 0$//

$248_{10} = \underline{1111}\,\underline{1000}_2 = F8_{16}$ (0xF8)

//

Q1 b. **(2 points)**

Conversion

Convert each of the signed decimal values below into its corresponding two's complement binary value (w = 8), then convert the binary value into its corresponding hexadecimal value.

I. 123_{10}

II. -74_{10}

i. 123_{10} → using division

$123 \div 2 = 61$ R1
$61 = 30$ R1
$30 = 15$ R0
$15 = 7$ R1
$7 = 3$ R1
$3 = 1$ R1
$1 = 0$ R1

→ using subtraction

$123 - 64\ (2^6) = 59$
$59 - 32\ (2^5) = 27$
$27 - 16\ (2^4) = 11$
$11 - 8\ (2^3) = 3$
$3 - 2\ (2^1) = 1$
$1 - 1\ (2^0) = 0$

padding

$123_{10} = 01111011_2 = 7B_{16}$ (0x7B)

+ve

ii. $-74_{10} \rightarrow U2B(-74_{10} + 2^{w}) = U2B(-74_{10} + 256_{10})$ ($\rightarrow w = 8$)

$= U2B(182_{10})$

$\Rightarrow 182 \div 2 = 91 \text{ R}0$

91	= 45	R1
45	= 22	R1
22	= 11	R0
11	= 5	R1
5	= 2	R1
2	= 1	R0
1	= 0 //	R1

$= \underline{1011}\,\underline{0110}_2 // = B6_{16}$

OR $-74_{10} \rightarrow (\sim(U2B(|-74_{10}|))) + 1$

$\Rightarrow |-74_{10}| = 74_{10}$

$\Rightarrow U2B(|-74_{10}|) = 74 \div 2 = 37 \text{ R}0$

37	= 18	R1
18	= 9	R0
9	= 4	R1
4	= 2	R0
2	= 1	R0
1	= 0 //	R1

$= \underset{\text{padding}}{0}1001010_2$

$\Rightarrow \sim(U2B(|-74_{10}|)) = \sim(01001010_2)$

$= 10110101_2$

$\Rightarrow (\sim(U2B(|-74_{10}|))) + 1 = 10110101_2$

$$\begin{array}{r} 10110101_2 \\ +\quad 1 \\ \hline 10110110_2 // \end{array}$$

Q1 c. (2 points)

Conversion

Interpret each of the binary values below (w = 8) first as an unsigned decimal value, then as a signed decimal value (using the two's complement encoding scheme).

I. 11101001_2

II. 10010110_2

i. $11101001_2 \rightarrow$ as a signed "decimal number" (two's complement)

$B2T(11101001_2) \Rightarrow -1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^3 + 1\cdot 2^0$

$= -128 + 64 + 32 + 8 + 1 = -23_{10}$

Note: $-23_{10} + 2^w$ ($w=8$)

$= -23_{10} + 256$

$= 233_{10}$

$\rightarrow$ as an unsigned "decimal number"

$B2U(11101001_2) \Rightarrow 1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^3 + 1\cdot 2^0$

$= 128 + 64 + 32 + 8 + 1 = 233_{10}$

---

ii. $10010110_2 \rightarrow$ as a signed "decimal number" (two's complement)

$B2T(10010110_2) \Rightarrow -1\cdot 2^7 + 1\cdot 2^4 + 1\cdot 2^2 + 1\cdot 2^1 -$

(can drop "1" $\rightarrow -2^7 + 2^4 + 2^2 + 2^1$)

$= -128 + 16 + 4 + 2 = -106_{10}$

Note: $-106_{10} + 2^w$

$= -106_{10} + 256$

$= 150_{10}$

$\rightarrow$ as an unsigned "decimal number"

$B2U(10010110_2) \Rightarrow 2^7 + 2^4 + 2^2 + 2^1$

$= 128 + 16 + 4 + 2 = 150_{10}$

Q1 d. (2 points)

Conversion

Convert 247_{10} (unsigned value) into a signed value directly, without converting it first to its corresponding binary value (w = 8).

Answer: $U2T(247_{10}) = 247_{10} - 2^{W} = 247_{10} - 2^{8} = 247_{10} - 256 = -9_{10}$

Q1 e. (2 points)

Conversion

Convert -112_{10} (signed value) into an unsigned value directly, without converting it first to a binary number (w = 8).

Answer: $T2U(-112_{10}) = -112_{10} + 2^{W} = -112_{10} + 2^{8} = -112_{10} + 256 = 144_{10}$

Q2 a. (3 points)

Unsigned and signed arithmetic operations and overflow

Convert each of the operands (unsigned decimal values), in I. and in II. below, into its corresponding binary value (w = 8), then perform both the decimal addition and the binary addition, indicating the **true sum** and the **actual sum** and whether these two sums are the same or not.

Unsigned addition:

I. $74_{10} + 63_{10}$

II. $123_{10} + 157_{10}$

For the binary addition, clearly label all **carry in** bits (by using the label "carry in") and the **carry out** bit (by using the label "carry out").

Finally, indicate whether or not an **overflow** occurred. If an overflow occurred, explain how addition overflow can be detected when adding decimal operands in a C program where you do not have access to the carry out bit.

I. $74_{10} + 63_{10}$

$74_{10} \rightarrow$ using subtraction

$74 - 64\ (2^6) = 10$

$10 - 8\ (2^3) = 2$

$2 - 2\ (2^1) = 0$

$\therefore 74_{10} = 01001010_2$

$63_{10} \rightarrow$ using subtraction

$63 - 32\ (2^5) = 31$

$31 - 16\ (2^4) = 15$

$15 - 8\ (2^3) = 7$

$7 - 4\ (2^2) = 3$

$3 - 2\ (2^1) = 1$

$1 - 1\ (2^0) = 0$

$\therefore 63_{10} = 00111111_2$

using operands		@ bit-level	
		1 1 1 1 1 1 1	← carry in bits
i. 74_{10}	→	01001010_2	
$+ 63_{10}$	→	$+ 00111111_2$	
137_{10} ← true sum		$10001001_2 = 137_{10}$	
unsigned		carry out bit is 0	Actual sum

"True sum" is what we obtain when we add two operands when we have infinite amount of space (paper) to record the sum.

"Actual sum" is what we obtain when we (the computer) add two operands when we (the computer memory) have finite amount of space to record the sum.

So, **True sum** = **Actual sum** -> no overflow occurred, i.e., the **True sum** has not overflowed the range of unsigned values that can be represented with 8 bits (w = 8) -> [0..255_{10}]

(extra) For unsigned addition, we can **detect** addition overflow when adding decimal operands in a C program, where we do not have access to the carry out bit, (in other words, whether the **True sum** overflows the range [0 .. 255_{10}] or not) by looking at whether or not the actual sum >= one of the operand. Here, because 137_{10} >= 74_{10} (or 137_{10} >= 63_{10}), we know that the **True sum** does not overflow the range (no overflow occurred).

From Practice Problem 2.7 in our textbook:

```
/* Determine whether arguments can be added without
overflow. */
int uadd_ok(unsigned x, unsigned y) {
  unsigned sum = x + y;
  return sum >= x;    // or return sum >= y;
}
```

II. 123_{10} + 157_{10}

123_{10} → using subtraction

$123 - 64\ (2^6) = 59$
$59 - 32\ (2^5) = 27$
$27 - 16\ (2^4) = 11$
$11 - 8\ (2^3) = 3$
$3 - 2\ (2^1) = 1$
$1 - 1\ (2^0) = 0$

$\therefore 123_{10} = 01111011_2$

157_{10} → using subtraction

$157 - 128\ (2^7) = 29$
$29 - 16\ (2^4) = 13$
$13 - 8\ (2^3) = 5$
$5 - 4\ (2^2) = 1$
$1 - 1\ (2^0) = 0$

$\therefore 157_{10} = 10011101_2$

ii. 123_{10} →
$+ 157_{10}$ →
280_{10} ← true sum

1 1 1 1 1 1 1 ← carry in bits
$+ 01111011_2$
10011101_2
$\cancel{1}00011000_2 = 24_{10}$

↖ carry out bit ∴ discarded

↑ (actual sum

True sum ≠ actual sum

So, **True sum** != **Actual sum** -> **overflow occurred**, i.e., the **True sum** has overflowed the range of unsigned values that can be represented with 8 bits (w = 8) -> [$0..255_{10}$]

For unsigned addition, we can **detect** addition overflow when adding decimal operands in a C program, where we do not have access to the carry out bit, (in other words, whether the **True sum** overflows the range [$0 .. 255_{10}$] or not) by looking at whether or not the actual sum >= one of the operand. Here, because it is not the case that 24_{10} >= 123_{10} (or it is not the case that 24_{10} >= 157_{10}), we know that the **True sum** has overflowed the range (overflow occurred).

From Practice Problem 2.7 in our textbook:

```
/* Determine whether arguments can be added without
overflow. */
int uadd_ok(unsigned x, unsigned y) {
  unsigned sum = x + y;
  return sum >= x;    // or return sum >= y;
}
```

Q2 b. (6 points)

Unsigned and signed arithmetic operations and overflow

Convert each of the operands (signed decimal values), in I. to IV. below, into its corresponding **two's complement** binary value (w = 8), then perform both the decimal addition and the binary addition, indicating the **true sum** and the **actual sum** and whether these two sums are the same or not.

Signed (two's complement) addition:

I. 28_{10} + -74_{10}

II. -117_{10} + 126_{10}

III. 74_{10} + 63_{10}

IV. -119_{10} + -105_{10}

For the binary addition, clearly label all **carry in** bits (by using the label "carry in") and the **carry out** bit (by using the label "carry out").

Finally, indicate whether or not a **positive** or a **negative overflow** occurred. If an overflow occurred, explain how addition overflow can be detected when adding decimal operands in a C program where you do not have access to the carry out bit.

I. $28_{10} + -74_{10}$

28_{10} → using subtraction

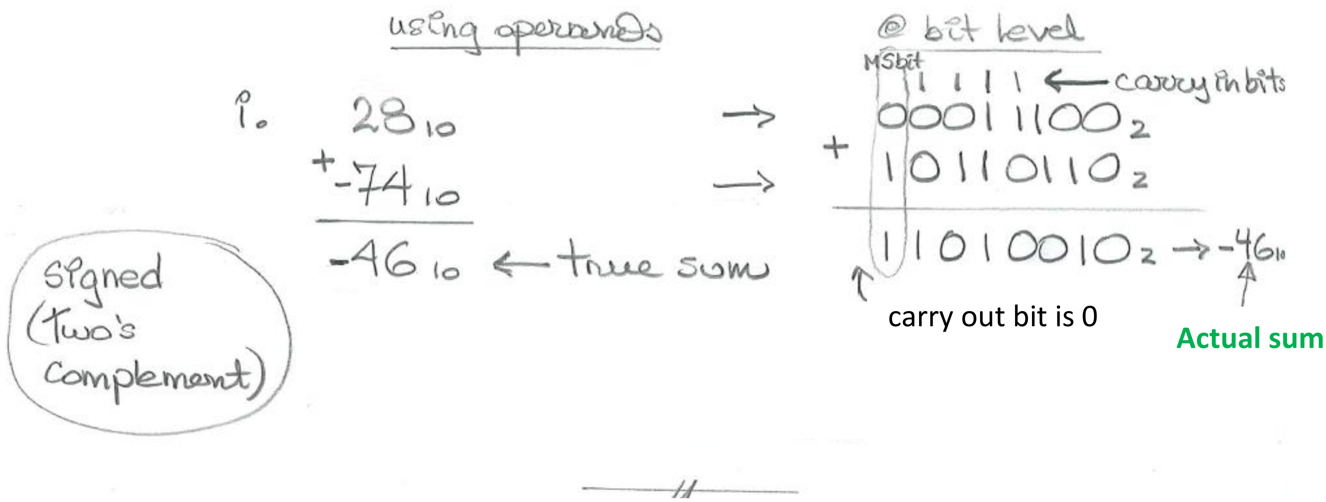
$28 - 16\ (2^4) = 12$

$12 - 8\ (2^3) = 4$

$4 - 4\ (2^2) = 0$

$\therefore 28_{10} = 00011100_2$

-74_{10} → see Question 1 b. ii.



So, **True sum** = **Actual sum** -> no (positive/negative) overflow occurred, i.e., the **True sum** has not overflowed the range of signed values that can be represented with 8 bits (w = 8) -> [-128_{10} .. 127_{10}].

(extra) For signed (two's complement) addition, we can **detect** positive or negative addition overflow when adding decimal operands in a C program, where we do not have access to the carry out bit, (in other words, whether the **True sum** overflows the positive or negative range [-128_{10} .. 127_{10}] or not) by looking at whether or not

x >= 0 && y >= 0 && actual sum < 0 -> if true, then positive overflow

or

x < 0 && y < 0 && actual sum >= 0 -> if true, then negative overflow

Here, because it is not the case (it is false) that

28_{10} >= 0 && -74_{10} >= 0 && -46_{10} < 0 -> true && false && true = false

or

28_{10} < 0 && -74_{10} < 0 && -46_{10} >= 0 -> false && true && false = false

then we know that the **True sum** does not overflow the range (no overflow occurred).

II. **-**117_{10} + 126_{10}

$-117_{10} \rightarrow$ using subtraction

$$-117 - (-128)\ \boxed{(-2^7)} = 11$$
$$11 - 8\ \boxed{(2^3)} = 3$$
$$3 - 2\ \boxed{(2^1)} = 1$$
$$1 - 1\ \boxed{(2^0)} = 0$$

$\therefore -117_{10} = 10001011_2$

↑ negative weight

$126_{10} \rightarrow$ using subtraction

$$126 - 64\ (2^6) = 62$$
$$62 - 32\ (2^5) = 30$$
$$30 - 16\ (2^4) = 14$$
$$14 - 8\ (2^3) = 6$$
$$6 - 4\ (2^2) = 2$$
$$2 - 2\ (2^1) = 0$$

$\therefore 126_{10} = 01111110_2$

ii.

$$\begin{array}{r} -117_{10} \\ +126_{10} \\ \hline 9_{10} \end{array} \leftarrow \text{true sum}$$

$\longrightarrow$

$$\begin{array}{rl} & \text{MSbit} \\ & 111111 \leftarrow \text{carry in bits} \\ & 10001011_2 \\ + & 01111110_2 \\ \hline & \cancel{1}00001001_2 \rightarrow 9_{10} \end{array}$$

carry out bit ∴ discarded

since true sum = actual sum,

So, **True sum** = **Actual sum** -> no (positive/negative) overflow occurred, i.e., the **True sum** has not overflowed the range of signed values that can be represented with 8 bits (w = 8) -> [-128_{10} .. 127_{10}].

(extra) For signed (two's complement) addition, we can **detect** positive or negative addition overflow when adding decimal operands in a C program, where we do not have access to the carry out bit, (in other words, whether the **True sum** overflows the positive or negative range [-128_{10} .. 127_{10}] or not) by looking at whether or not

x >= 0 && y >= 0 && actual sum < 0 -> if true, then positive overflow

or

x < 0 && y < 0 && actual sum >= 0 -> if true, then negative overflow

Here, because it is not the case (it is false) that

-117_{10} >= 0 && 126_{10} >= 0 && 9_{10} < 0 -> false && true && false = false

or

-117_{10} < 0 && 126_{10} < 0 && 9_{10} >= 0 -> true && false && ture = false

then we know that the **True sum** does not overflow the range (no overflow occurred).

III. $74_{10} + 63_{10}$

For T2B(74_{10}) and T2B(63_{10}), see Question 2 a. i.

iii. $74_{10} + 63_{10} = 137_{10}$ ← true sum

$\rightarrow$

MSbit

```
   111111      ← carry in bits
   01001010₂
 + 00111111₂
 -----------
   10001001  → -119₁₀
```

carry out bit is 0

Actual sum

signed (two's complement)

True sum ≠ actual sum

So, **True sum** != **Actual sum** -> positive overflow occurred, i.e., the **True sum** overflows the positive side of the range of signed values that can be represented by 8 bits (w = 8). Indeed, 137_{10} is beyond the positive side of the range [-128_{10} .. 127_{10}].

For signed (two's complement) addition, we can **detect** positive or negative addition overflow when adding decimal operands in a C program, where we do not have access to the carry out bit, (in other words, whether the **True sum** overflows the positive or negative range [-128_{10} .. 127_{10}] or not) by looking at whether or not

x >= 0 && y >= 0 && actual sum < 0 -> if true, then positive overflow

or

x < 0 && y < 0 && actual sum >= 0 -> if true, then negative overflow

Here, because it is the case (it is true) that

74_{10} >= 0 && 63_{10} >= 0 && -119_{10} < 0 -> true && true && true = true

then we know that a positive overflow occurred (the **True sum** does overflow the positive range).

IV. $-119_{10} + -105_{10}$

$-119_{10} \rightarrow$ using subtraction

$-119 - (-128) \; (-2^7) = 9$
$9 - 8 \; (2^3) = 1$
$1 - 1 \; (2^0) = 0$

$\therefore -119_{10} = 10001001_2$

negative weight

$-105_{10} \rightarrow$ using subtraction

$-105 - (-128) \; (-2^7) = 23$
$23 - 16 \; (2^4) = 7$
$7 - 4 \; (2^2) = 3$
$3 - 2 \; (2^1) = 1$
$1 - 1 \; (2^0) = 0$

$\therefore -105_{10} = 10010111_2$

negative weight

iv.

$$\begin{array}{rcr} & & 1\;1\;1\;1\;1 \leftarrow \text{carry in bits} \\ -119_{10} & \rightarrow & 10001001_2 \\ + -105_{10} & \rightarrow & +\;10010111_2 \\ \hline -224_{10} \leftarrow \text{true sum} & & \cancel{1}00100000_2 \rightarrow 32_{10} \end{array}$$

$\nwarrow$ carry out bit

Actual sum

True sum $\neq$ actual sum

So, **True sum** != **Actual sum** -> negative overflow occurred, i.e., the **True sum** overflows the negative side of the range of signed values that can be represented by 8 bits (w = 8). Indeed, -224_{10} is beyond the negative side of the range [-128..127].

For signed (two's complement) addition, we can **detect** positive or negative addition overflow when adding decimal operands in a C program, where we do not have access to the carry out bit, (in other words, whether the **True sum** overflows the positive or negative range [-128_{10} .. 127_{10}] or not) by looking at whether or not

x >= 0 && y >= 0 && actual sum < 0 -> if true, then positive overflow

or

x < 0 && y < 0 && actual sum >= 0 -> if true, then negative overflow

Here, because it is the case (it is true) that

$-119_{10} < 0$ && $-105_{10} < 0$ && $32_{10} >= 0$ -> true && true && true = true

then we know that a negative overflow occurred (the **True sum** does overflow the negative range).

Q3 (13 points)

C Code, endian and bit-level manipulation

Download `Assn1-files.zip` from our course web site then extract and open `Assn1_Q3.c`, `Assn1_main.c` and `makefile` in a text editor. Read and understand their content.

Using the `makefile`, compile and execute the program.

Make sure you do all this on our ***target machine***. Why? Because, when marking your assignment, the TA will be compiling and testing your code on the target machine. Hence, you want to make sure the code you submit does compile and execute as expected on the target machine.

Requirements:

- While answering this question, you must not change the prototype of the functions given. The reason is that these functions will be tested using a test driver built based on these function prototypes.
- Your code must be readable and easy to understand. Therefore:
  - Comment your code and write your program such that its statements are well spaced.
  - No goto statements, please!

a. [2 marks] Modify the `printf` statement of the `show_bytes(…)` function such that it first prints the memory address of each byte then the content of the byte itself on its own line. Here is an example:

`0x7ffe5fb887cc 0x80`

where `0x7ffe5fb887cc` is the memory address of a byte which contains the value `0x80`.

Compile and test your program.

```
// Q3 a.
void show_bytes(byte_pointer start, size_t len) {
  size_t i;
  for (i = 0; i < len; i++)
    // printf(" %.2x", start[i]); // Original statement
    printf("%p 0x%.2x\n", &start[i], start[i]);
  printf("\n");
  return;
}
```

b. [2 marks] Looking at the output of this program, would you say that our target machine is a **little endian** or a **big endian** computer? Justify your answer by including some of the output of your program in your answer and pointing out the endianness.

Put your answer to this question in your `Assn1_Q3.c` file, just below the `show_bytes(…)` function and transform your answer into a comment such that your program still compiles.

```
/*Q3 b.
For number 12345 (0000 0000 0000 0000 0011 0000 0011 1001) I get:
      0x7ffe09e4097c  0x39
      0x7ffe09e4097d  0x30
      0x7ffe09e4097e  0x00
      0x7ffe09e4097f  0x00

where the least significant byte (LSB) is stored at memory
address 0x7ffe09e4097c and most significant byte (MSB) is stored
at memory address 0x7ffe09e4097f
since 0x7ffe09e4097c is a smaller memory address value than
0x7ffe09e4097f, this indicates that our target machine is a
little endian computer

For the negative number -12345 (1111 1111 1111 1111 1100 1111
1100 0111) I get:
      0x7ffd3e027fbc  0xc7
      0x7ffd3e027fbd  0xcf
      0x7ffd3e027fbe  0xff
      0x7ffd3e027fbf  0xff

Again little endian since the LSB is stored at 0x7ffd3e027fbc
and the MSB is stored at 0x7ffd3e027fbf.
*/
```

c. [2 marks] Modify the `printf` statement of the `show_bytes_2(…)` function such that, instead of using array notation to access each element of the array `start`, it uses pointer notation to access each of these elements. The output of your function should be the same as the funciton you modified in a. above, with each pair of values – memory address and memory content – printed on its own line:

`0x7ffe5fb887cc 0x80`

Bottom line: `show_bytes(…)` and `show_bytes_2(…)` should print the same thing given the same data. They just do it differently.

```
// Q3 c.
void show_bytes_2(byte_pointer start, size_t len) {
```

```
  size_t i;
  // WHY hx in 0x%.2hx"? Why not: 0x%.2x?
  for (i = 0; i < len; i++) {
     printf(" %p 0x%.2hx\n", start + i, *(start + i)); //

     // OR :for a more general version
     // printf("%p 0x%.2hx\n", (start + i * sizeof(*start)),
     //                         *(start + i * sizeof(*start)));
     // However, in the context of Q3 c. sizeof(*start) is
     // optional since start is a byte_pointer.
     // Can you see why?
  }
  printf("\n");
  return;
}
```

d. [4 marks] Write a function called `show_bits( … )`. This function must have the following prototype:

```
void show_bits(int);
```

This function must print the bit pattern of the parameter of type `int`.

Compile and test your program.

Here are two test cases (data and expected results) to illustrate the behaviour of this function:

Test Case 1:

If the parameter (`int`) is **12345**, then `show_bits( … )` prints:

```
00000000000000000011000000111001
```

Test Case 2:

If the parameter (`int`) is **-12345**, then `show_bits( … )` prints:

```
11111111111111111100111111000111
```

Check out Lab 2 for a possible solution!

e. [3 marks] Write a function called `mask_LSbits( … )`. This function must have the following prototype:

```
int mask_LSbits( int n );
```

This function creates (returns) a mask with the **n** least significant bits set (to 1). For example, if **n** is 2, the function returns 3 (i.e., 0x00000003) and if **n** is 15, the function returns 32767 (i.e., 0x00007fff).

What happens when **n >= w** or when **n <= 0**?
When n >= w, your function must return a mask of all 1's.
When n <= 0, your function must return a mask of all 0's, i.e., 0.

Requirements (for e.):

- When creating the mask, while implementing this function, you **must not** use
  - division, modulus and/or multiplication,
  - iterative statements (such as loops),
  - call other function(s),
  - however, you can use `sizeof(…)`.
- Exceptions:
  - You can call the function `sizeof(…)` as many times as you wish. If you call `sizeof(…)` (note that you do not have to), you can then use multiplication with it. For example: `sizeof(…) * 8`.
  - You can use conditional statements in your function only when you validate the parameter.

Testing: For part e., make sure you test your code with test cases that use valid and invalid test data.
Here is an example of a test case that uses **valid test data**: calling `mask_LSbits(4)`
Here is an example of a test case that uses **invalid test data**: calling `mask_LSbits(0)`

To test your program, make sure you add your test cases to the test provided test driver `Assn1_main.c`.

For this Question 3, you need to make two submissions:

1. Please, submit your **`Assn1_Q3.c`** on CourSys.
2. Please, scan your **`Assn1_Q3.c`** as a pdf document and upload it below.

```
//Q3 e.

int  mask_LSbits(int shift_amount){

   // Parameter validation:

  // if shift_amount <= 0, return 0 -> a mask of all 0s
  if( shift_amount <= 0 ) return 0;

  //if shift_amount >= 32 return -1 -> a mask of all 1s
  if( shift_amount >= 32) return -1;
```

```
    // Way 1:
    // "-1" is 32 1s
   // shift these 1s to the left "shift_amount" times and
    // then reverse (flip) the bits i.e 0 to 1 and 1 to 0
   int new_value = ~(-1 << shift_amount);
   return new_value;

    // Way 2:
    // "1" is 31 1s and 1 in LSbit position
   // shift these bits to the left "shift_amount" times and
    // then add "-1" (32 1s)
    return (1 << shift_amount) - 1;

}
```