ssignment 1 - SOLUTION

Objectives:

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

---

1. [6 marks] Conversion - Marked by Pranav

   a. 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}

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}

ii. 248_{10} → using division

$248 \div 2 = 124$ R0
124 = 62 R0
62 = 31 R0
31 = 15 R1
15 = 7 R1
7 = 3 R1
3 = 1 R1
1 = 0 R1 MSbit

→ using subtraction

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

$248_{10} = 11111000_2 = F8_{16}$ (0xF8)

b. 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}

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}

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

$= U2B(182_{10})$

$\Rightarrow 182 \div 2 = 91$ R0
91 $= 45$ R1
45 $= 22$ R1
22 $= 11$ R0
11 $= 5$ R1
5 $= 2$ R1
2 $= 1$ R0
1 $= 0$ R1

$= 10110110_2 = B6_{16}$

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

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

$\Rightarrow U2B(|-74_{10}|) = 74 \div 2 = 37$ R0
37 $= 18$ R1
18 $= 9$ R0
9 $= 4$ R1
4 $= 2$ R0
2 $= 1$ R0
1 $= 0$ R1

padding
$= 01001010_2$

$\Rightarrow \sim(U2B(|-74_{10}|)) = \sim(01001010_2)$
$= 10110101_2$

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

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

c. 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

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

$B2T(11101001_2) \Rightarrow -1\cdot2^7 + 1\cdot2^6 + 1\cdot2^5 + 1\cdot2^3 + 1\cdot2^0$

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

Note: $-23_{10} + 2^{w}$ (w=8)
$= -23_{10} + 256$
$= 233_{10}$

$\rightarrow$ as an unsigned "decimal number"

$B2U(11101001_2) \Rightarrow 1\cdot2^7 + 1\cdot2^6 + 1\cdot2^5 + 1\cdot2^3 + 1\cdot2^0$

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

II. 10010110_2

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

$B2T(10010110_2) \Rightarrow -1\cdot2^7 + 1\cdot2^4 + 1\cdot2^2 + 1\cdot2^1 -$

(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"

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

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

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

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

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

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

2. [6 marks] Unsigned and signed arithmetic operations and overflow - Marked by Pranav

For **a.** below, convert each of the operands (**unsigned** decimal values) into its corresponding binary value (w = 8).

For **b.** below, convert each of the operands (**signed** decimal values) into its corresponding **two's complement** binary value (w = 8).

For **a.** and **b.** below, perform both the decimal addition and the binary addition and indicate the **true sum** and the **actual sum** and whether they are the same or not.

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 (for **signed** values, specify whether the overflow is positive or negative).
- If an overflow occurred, explain how addition overflow can be detected 1) at the bit level, and 2) using the decimal operands (Note: you cannot use "carry out bit" in your explanation since the carry out bit is "inaccessible" when you are adding decimal operands).

a. Unsigned addition:

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

i. 74_{10} $\rightarrow$

$+ 63_{10}$ $\rightarrow$

137_{10} ← true sum

unsigned

@ bit-level

1111111 ← carry in bits

01001010_2

$+ 00111111_2$

$10001001_2 = 137_{10}$

no carry out bit

Or 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}]

For unsigned addition, we can **detect** whether the **True sum** overflows the range [0 .. 255_{10}] or not

1. at the bit level -> by looking at the carry out bit. In this problem, there is no carry out bit (or another way of saying this is that the carry out bit is 0) indicating that the **True sum** does not overflow the range, we can express the **True sum** using 8 bits (no overflow occurred).
2. using the decimal operands -> 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).

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

1111111 ← carry in bits
$+\ 01111011_2$
10011101_2
$\cancel{1}00011000_2 = 24_{10}$

carry out bit ∴ discarded

actual sum

True sum $\neq$ 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** whether the **True sum** overflows the range [0 .. 255_{10}] or not

1. at the bit level -> by looking at the carry out bit. In this problem, the carry out bit is 1 indicating that the **True sum** overflows the range (overflow occurred) as we need a 9^{th} bit to express the **True sum** (such 9^{th} bit does not exist when w = 8).
2. using the decimal operands -> 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).

---

b. Signed (two's complement) addition:

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

$28_{10} \rightarrow$ using subtraction

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

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

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

$-74_{10} \rightarrow$ see Question 1 b. ii.

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

Signed (Two's Complement)

using operands

i. 28_{10}
$+ -74_{10}$
$-46_{10} \leftarrow$ true sum

MS carry in bit is 0

@ bit level

MSbit

$$\begin{array}{rl} & 1111 \leftarrow \text{carry in bits} \\ & 00011100_2 \\ + & 10110110_2 \\ \hline & 11010010_2 \rightarrow -46_{10} \end{array}$$

carry out bit is 0 (there is no carry out bit)

Actual sum

—//—

So, **True sum** = **Actual sum** -> no 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}].

For signed (two's complement) addition, we can **detect** whether the **True sum** overflows the range [-128_{10} .. 127_{10}] or not

1. at the bit level -> by looking at the carry out bit as well as the most significant (MS) carry in bit. In this problem, the carry out bit is 0 and the most significant (MS) carry in bit is also 0 indicating that the **True sum** does not overflow the range, i.e., that we can express the **True sum** using 8 bits (no overflow occurred).
2. using the decimal operands -> by looking at whether or not

   x < 0 && y < 0 && actual sum >= 0 (indicates an negative overflow)

   or

   x >= 0 && y >= 0 && actual sum < 0 (indicates positive overflow)

   Here, because 28_{10} >= 0 but -74_{10} < 0, we know that the **True sum** does not overflow the range (no overflow occurred).

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

-117_{10} -> using subtraction

$-117 - (-128)\ (-2^7) = 11$

$11 - 8\ (2^3) = 3$

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

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

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

negative weight

126_{10} -> 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. $-117_{10} + 126_{10} = 9_{10}$ ← true sum

MSbit

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

carry out bit ∴ discarded

since true sum = actual sum,

So, **True sum** = **Actual sum** -> no 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}].

For signed (two's complement) addition, we can **detect** whether the **True sum** overflows the range [-128_{10} .. 127_{10}] or not

1. at the bit level -> by looking at the carry out bit as well as the most significant (MS) carry in bit. In this problem, the carry out bit is 1 and the most significant (MS) carry in bit is also 1 indicating that the **True sum** does not overflow the range, i.e., that we can express the **True sum** using 8 bits (no overflow occurred).
2. using the decimal operands -> by looking at whether or not

   x < 0 && y < 0 && actual sum >= 0 (indicates an negative overflow)

   or

   x >= 0 && y >= 0 && actual sum < 0 (indicates positive overflow)

   Here, because $-117_{10} < 0$ but $126_{10} >= 0$, we know that the **True sum** does not overflow the range (no overflow occurred).

Note that, contrarily to the "unsigned" situation, where a carry out bit indicated an overflow, in the "signed" situation, a carry out bit does not necessarily indicate an overflow.

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 $\longrightarrow$

MSbit

```
 11111    ← carry in bits
 01001010₂
+00111111₂
----------
010001001 → -119₁₀
```

carry out bit is 0 (there is no carry out bit)

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** whether the **True sum** overflows the range [-128_{10} .. 127_{10}] or not

1. at the bit level -> by looking at the carry out bit as well as the most significant (MS) carry in bit. In this problem, the carry out bit is 0 and the most significant (MS) carry in bit is 1 indicating that the **True sum** overflows the range, i.e., that we cannot express the **True sum** using only 8 bits, that we need a 9th bit to do so (overflow occurred).
2. using the decimal operands -> by looking at whether or not

   x < 0 && y < 0 && actual sum >= 0 (indicates an negative overflow)

   or

   x >= 0 && y >= 0 && actual sum < 0 (indicates positive overflow)

   Here, because 74_{10} >= 0 && 63_{10} >= 0 && actual sum (-119_{10}) < 0, we know that a positive overflow occurs

---

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

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

$$\begin{array}{lll} -119 - (-128) & \boxed{(-2^7)} & = 9 \\ 9 - \quad 8 & \boxed{(2^3)} & = 1 \\ 1 - \quad 1 & \boxed{(2^0)} & = 0 \end{array}$$

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

negative weight

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

$$\begin{array}{lll} -105 - (-128) & (-2^7) & = 23 \\ 23 - \quad 16 & (2^4) & = 7 \\ 7 - \quad 4 & (2^2) & = 3 \\ 3 - \quad 2 & (2^1) & = 1 \\ 1 - \quad 1 & (2^0) & = 0 \end{array}$$

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

negative weight

iv.

MS carry in bit is 0   1 1 1 1 1 ← carry in bits

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

↖ 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** whether the **True sum** overflows the range [-128_{10} .. 127_{10}] or not

1. at the bit level -> by looking at the carry out bit as well as the most significant (MS) carry in bit. In this problem, the carry out bit is 1 and the most significant (MS) carry in bit is 0 indicating that the **True sum** overflows the range, i.e., that we cannot express the **True sum** using only 8 bits, that we need a 9th bit to do so (overflow occurred).
1. using the decimal operands -> by looking at whether or not

x < 0 && y < 0 && actual sum >= 0 (indicates an negative overflow)

or

x >= 0 && y >= 0 && actual sum < 0 (indicates positive overflow)

Here, because $-119_{10} < 0$ && $-105_{10} < 0$ && actual sum (32_{10}) >= 0, we know that a negative overflow occurs.

---

3. [8 marks] C Code, endian and bit-level manipulation - Marked by Sedi

Possible solution to Q3 - Student #1

```
#include <stdio.h>
#include <stdlib.h>

typedef unsigned char *byte_pointer;

//QUESTION 3.D//
void show_bits(int x) {
    int binarystorage[32];
    int temp = x;
    int i = 0;
    int z = 31;

    for (i=0; i <=31; i++) { //figure out the signed bit pattern using
modulus
        binarystorage[i] = abs(temp % 2); //storing the values of modulus in
array
        temp = temp / 2;
    }
    if (x < 0) { //if the original input value is negative,
        // find position of the first 1 from the right most side
        for (z = 0;z <= 31; z++) {
            if (binarystorage[z] == 1) {
                break;
            }
        }

        //complementing all the values after the first 1 that appeared
        for (int k = z + 1; k <= 32; k++) {

            if (binarystorage[k] == 1) {
                binarystorage[k] = 0;
            }
            else if (binarystorage[k] == 0) {
                binarystorage[k] = 1;
            }
        }
    }

    //printing out the resulting bit pattern
    for (z = 31;z >= 0; z--) {
        printf("%d", binarystorage[z]);
    }
    printf("\n");
}

//QUESTION 3.E//
int mask_LSbits(int n) {
    int mask;
    if (n <= 0) {
        mask = 0;
```

```
    }

    // Assuming sizeof(int)*8 = 32 is a safe assumption
    // considering that the target machine is CSIL Linux workstation
    // Although this would be considered better code:
    // else if (n >= sizeof(int)*8)

        else if (n >= 32) {
        mask = - 1;
    }

    // The '1' will be shifted n times to the left and then the result will
    // have 1 subtracted from it to yield the mask
    else {
        mask = (1 << n) - 1;
    }
    // DEBUGGING //
    // printf("%d\n", mask); //prints the mask in decimal form
    // printf("0x%x\n", mask); //prints the mask in hexadecimal form

    return mask;
}
```

Possible solution to Q3 - Student #2

```
#include <stdio.h>
#include <stdlib.h>

typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, size_t len) {
    size_t i;

    for (i = 0; i < len; i++){

        //the original printf code (given code)
        //printf(" %.2x", start[i]);

//Question 3)part A: show_bytes(…)
// printing the memory address of each byte and its content
        printf(" %p  0x%.2x", &start[i],start[i]);

/*Question 3)part B:
   for number 12345 I got 0x7ffe09e4097c  0x39  0011 1001
        0x7ffe09e4097d  0x30  0011 0000
        0x7ffe09e4097e  0x00  0000 0000
        0x7ffe09e4097f  0x00  0000 0000

the most significate bit is stored at 0x7ffe09e4097f and least significate
bit is at 0x7ffe09e4097c which indicated that our computer is little endian

for number 14 I got 0x7ffcb1e0b9ec  0x0e  0000 1110
        0x7ffcb1e0b9ed  0x00  0000 0000
        0x7ffcb1e0b9ee  0x00  0000 0000
        0x7ffcb1e0b9ef  0x00  0000 0000

again little endian since MSB is at 0x7ffcb1e0b9ef and LSB is at
0x7ffcb1e0b9ec

for a negative number -12345 I got 0x7ffd3e027fbc  0x00  0000 0000
        0x7ffd3e027fbd  0xe4  1110 0100
        0x7ffd3e027fbe  0x40  0100 0000
        0x7ffd3e027fbf  0xc6  1100 0110

again little endian since MSB is at 0x7ffd3e027fbf and LSB is at
0x7ffd3e027fbc
*/

//Question 3)part C: show_bytes_2(…)
// Changing the function such that instead of using array notation
        // to access each element of the array it uses pointer notation

        printf(" %p  0x%.2x", start + i,*(start + i));
        printf ("\n");
    }
    printf("\n");
}

...
```

```
//Question 3) part E: creating a mask

int  mask_LSbits(int shift_number){


        // if shift_number <= 0, returning a mask of all 0â€™s
        if( shift_number <= 0 ){
                return 0;
        }

        //if shift_number >= 32 returning a mask of all 1â€™s
        if( shift_number >= 32){
                return -1;
        }

        // number with 32 1s
        int one_bits = -1;

        // shifting the 1s to the left, shift_number times and then reversing
the
        //bits i.e 0 to 1 , 1 to 0
        int new_value = ~(one_bits << shift_number);

        return new_value;
}
```