

# Tutorial 4: Arithmetic and Logic Operations

## Problem 1: Data Representation

Consider the number  $A = 0xEEEEDDDD$ . What is the value of this number in decimal? What is its value when represented in 8-bit unsigned and signed numbers? What is its value when logical logically shifted to the right by two bits and represented in 8-bit unsigned and signed numbers? What is its value when arithmetically shifted to the right by two bits and represented in 8-bit unsigned and signed numbers?

We can represent number  $A = 0xEEEEDDDD$  in two ways; signed and unsigned integer representations:

$$\begin{aligned} \text{Unsigned: } & (+1)^1 \times 2^{31} + 1 \times 2^{30} + 1 \times 2^{29} + \dots + 1 \times 2^0 = 4008631773 \\ \text{Signed: } & (-1)^1 \times 2^{31} + 1 \times 2^{30} + 1 \times 2^{29} + \dots + 1 \times 2^0 = -286335523 \end{aligned}$$

Representation of number  $A = 0xEEEEDDDD$  in 8 bits only retains its lower 8 bits  $0xDD$ . Its signed and unsigned integer representations are:

$$\begin{aligned} \text{Unsigned: } & (+1)^1 \times 2^7 + 1 \times 2^6 + 1 \times 2^4 + \dots + 1 \times 2^0 = 221 \\ \text{Signed: } & (-1)^1 \times 2^7 + 1 \times 2^6 + 1 \times 2^4 + \dots + 1 \times 2^0 = -35 \end{aligned}$$

Logical shifting of number  $A = 0xEEEEDDDD$  to the right can be illustrated in binary as:

$(A = 0xEEEEDDDD) \gg \text{logical } 2 = (A = 0b1110, 1110, 1110, 1110, 1101, 1101, 1101, 1101) \gg \text{logical } 2 = (0b0011, 1011, 1011, 1011, 1011, 0111, 0111, 0111) = 0x3BBBB777$ . Note that the emptied 2 bits from the left are filled with 0 and two bits from the right have disappeared.

Representation of number  $A = 0x3BBBB777$  in 8 bits only retains its lower 8 bits  $0x77$ . Its signed and unsigned integer representations are:

$$\begin{aligned} \text{Unsigned: } & (+0)^1 \times 2^7 + 1 \times 2^6 + 1 \times 2^4 + \dots + 1 \times 2^0 = 119 \\ \text{Signed: } & (-0)^1 \times 2^7 + 1 \times 2^6 + 1 \times 2^4 + \dots + 1 \times 2^0 = 119 \end{aligned}$$

Arithmetic shifting of number  $A = 0xEEEEDDDD$  to the right can be illustrated in binary as:

$(A = 0xEEEEDDDD) \gg \text{arithmetic } 2 = (A = 0b1110, 1110, 1110, 1110, 1101, 1101, 1101, 1101) \gg \text{arithmetic } 2 = (0b1111, 1011, 1011, 1011, 1011, 0111, 0111, 0111) = 0xFBBBB777$ . Note that the emptied 2 bits from the left are filled with sign of the number (-) and two bits from the right have disappeared.

Representation of number  $A = 0xFBBBB777$  in 8 bits only retains its lower 8 bits  $0x77$ . Its signed and unsigned integer representations are:

$$\begin{aligned} \text{Unsigned: } & (+0)^1 \times 2^7 + 1 \times 2^6 + 1 \times 2^4 + \dots + 1 \times 2^0 = 119 \\ \text{Signed: } & (-0)^1 \times 2^7 + 1 \times 2^6 + 1 \times 2^4 + \dots + 1 \times 2^0 = 119 \end{aligned}$$

## Problem 2: Shift Operations in C

Consider the C code in Figure 1. Answer the following questions.

What are the outputs of the `printf` statements?

What are the outputs of the `printf` statements if  $(a = 0x7722)$ ?

```

#include <stdio.h>

int main (void)
{
    short a = 0xDEE5;
    char b;

    b = a;
    printf("short = \"%d\\\"\\n\\n", a >> 1);
    printf("char = \"%d\\\"\\n\\n", b >> 1);

    return 0;
}

```

**Figure 1: Program on Shift Operations**

Number is declared as short (a 16-bit signed number). Therefore (a = 0xDEE5) is represented as signed number as:

$$\text{Signed: } (-1)^1 \times 2^{15} + 1 \times 2^{14} + 1 \times 2^{12} + \dots + 1 \times 2^0 = -8475$$

Converting the number to a char (an 8-bit signed number), retains its lower 8 bits (0xE5). Representation of (0xE5) in b bits gives

$$\text{Signed: } (-1)^1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + \dots + 1 \times 2^0 = -27$$

Right Shift operations (a >>) and (b >> 1) are taken as arithmetic shifting as both short and char types are signed representations.

Shifting number (a = 0xDEE5) to the right by 1 bit can be illustrated in binary as:

(a = 0xDEE5) >> logical 1 = (a = 0b1101, 1110, 1110, 0101) >> arithmetic 1 = (a = 0b1110, 1111, 0111, 0010) = 0xEF72. Note that the emptied 1 bit from the left is filled with sign of the number (-) and one bit from the right has disappeared. Number = 0xEF72 = -4238. Note that -4238 is half -8475 when rounded towards negative infinity.

(b = 0xE5) >> logical 1 = (a = 0b1110, 0101) >> arithmetic 1 = (a = 0b1111, 0010) = 0xF2. Note that the emptied 1 bit from the left is filled with sign of the number (-) and one bit from the right has disappeared. Number = 0xF2 = -14. Note that -14 is half -27 when rounded towards negative infinity.

```

short = "-4238"

char = "-14"

```

**Figure 2: The Outputs of printf Statements**

Number is declared as short (a 16-bit signed number). Therefore (a = 0x7722) is represented as signed number as:

$$\text{Signed: } (0) \times 2^{15} + 1 \times 2^{14} + 1 \times 2^{12} + \dots + 1 \times 2^1 = 30498$$

Converting the number to a char (an 8-bit signed number), retains its lower 8 bits (0x22). Representation of (0x22) in b bits gives

$$\text{Signed: } (0) \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + \dots + 0 \times 2^0 = 34$$

Right Shift operations (a >>) and (b >> 1) are taken as arithmetic shifting as both short and char types are signed representations.

Shifting number (a = 0x7722) to the right by 1 bit can be illustrated in binary as:

(a = 0x7722) >> logical 1 = (a = 0b0111, 0111, 0010, 0010) >> arithmetic 1 =

(a = 0b0011, 1011, 1001, 0001) = 0x3B91. Note that the emptied 1 bit from the left is filled with sign of the number (+) and one bit from the right has disappeared. Number = 0x3B91 = 15249. Note that 15249 is half 30498.

(b = 0x22) >> logical 1 = (a = 0b0010, 0010) >> arithmetic 1 = (a = 0b0001,0001) = 0x11. Note that the emptied 1 bit from the left is filled with sign of the number (+) and one bit from the right has disappeared. Number = 0x11 = 17. Note that 17 is half 34.

```
short = "15249"
char = "17"
```

**Figure 3: The Outputs of printf Statements**

### Problem 3: Rotate Operation

Consider the ARM Assembly code in Figure 4 that does rotation of bits in a register. Write an equivalent C version of this program. If (V1 = 0xEEBAE213), what would register A1 contain after the execution this instruction?

```
mov a1, v1, ror #8 ;a1 ← v1 >> 8 bits , a1[31:24] ← v1[7:0]
```

**Figure 4: Assembly Program on Rotation**

The instruction (mov a1, v1, ror #8) rotates content of register V1 to the right (logically) places rotated version into register A1. The bits that are shifted out from the right enter in from the left. The C statement in Figure 5 is functionally equivalent to the assembly language instruction in Figure 4.

```
b = (a >> 8) | (a << 24);
```

**Figure 5: Program on Rotation**

The C statement (b = (a >> 8) | (a << 24);) first logically shifts (a) to the right by 8 bits and empties 8 bits on the left through the operation (a >> 8). Next it logically shifts (a) to the left by 24 bits and empties 24 bits on the right through the operation (a << 24). Combining of these two operations through (|) operation achieves the rotation. We can test this through the C program in Figure 6. The output of this program is provided in Figure 7.

```
#include <stdio.h>

int main (void)
{
    unsigned int a=0xEEBAE213;
    int b;

    b = (a >> 8) | (a << 24);
    printf("integer = \"%x\"\n\n", a);
    printf("rotated = \"%x\"\n\n", b);

    return 0;
}
```

**Figure 6: Assembly Program on Rotation**

```
integer = "EEBAE213"
rotated = "13EEBAE2"
```

**Figure 7: The Outputs of printf Statements**

Note that in the C program in Figure 6 variable (a) is declared as (unsigned int). This will ensure that the shift operation (a >> 8) is logical and empties the 8 bits on the left. If (a) is declared as (int), then the shift operation (a >> 8) is interpreted as arithmetic shift right and will leave 1 in the emptied 8 bits on the left. That will cause C statement (b = (a >> 8)|(a << 24);) to produce (b = 0xFFEEBAE2).

Also note that in the C program in Figure 6 if variable (a) is declared as (int), then its equivalent assembly language program will require more than one instruction corresponding to the C statement (b = (a >> 8)|(a << 24);). The equivalent assembly instructions are provided in Figure 8.

```
mov a1, v1, asr #8 ;a1 ← (v1 >> aritmatcally shifted right by 8 bits)
orr a1, v1, lsl #24 ;a1 ← a1 | ((v1 << aritmatcally shifted left by 24 bits)
```

**Figure 8: Assembly Program on Rotation for int**

However, if the variable (a) is guaranteed to only take negative values then a single assembly instruction given in Figure 9 is sufficient. That is because (asr) instruction fills the emptied bits with (1) and subsequent ORing with (1) through the (orr) instruction becomes redundant, as ORing anything with (1) produces (1).

```
mov a1, v1, asr #8 ;a1 ← (v1 >> aritmatcally shifted right by 8 bits)
```

**Figure 9: Assembly Program on Rotation for int (If only negative values are possible)**