Signed/Unsigned Integer Arithmetic in C

Vineel Kovvuri
http://vineelkovvuri.com

Contents

1 Introduction

This article is about understanding how integer conversions happen in C language. C standard defines the integer conversion rules not specific to any architecture. It makes things more complicated for the programmers to try and understand them.

Why do we need integer conversions at all? The answer is simple we need to have single type for the expression. Let's say we have an expression <expr1><op><expr2>when expr1 and expr2 are different types, we want the resulting expression from this to have one type.

The C99 specification defines below set of integer types called 'standard integer types'. Interestingly, the sizes of these integer types are not defined at all. It only defines the minimum supported size. For example, int need not be 8 bytes long on x64 platforms, the only definition for int is it should have at least 16 bits and similarly long should have at least 32 bits it need not be 8 bytes long. Depending on the platform and the processor architecture the ABI (Application Binary Interface) and the 64bit programming model will define the size of these basic types. Windows x64 follows LLP64(meaning only 'long long' and pointer size is 64 bit wide), So below are the sizes of the standard types that we are sticking to in this article.

Type	Size
signed char	1 bytes
unsigned char	1 bytes
signed short	2 bytes
unsigned short	2 bytes
signed int	4 bytes
unsigned int	4 bytes
signed long	4 bytes
unsigned long	4 bytes
signed long long	8 bytes
unsigned long long	8 bytes

Also, The specification leaves other aspects of C language definition undefined and this made room for optimization for the compilers. For example, the result of signed arithmetic leading to overflow/underflow is not defined because the specification predates to the machine architectures when 2's complement representation for -ve numbers is not universal, Even though virtually every architecture now uses 2's complement representation for -ve numbers. Hence the result of unsigned overflow/underflow is well defined but not signed overflow/underflow!

2 How signed-ness is represented in the hardware?

Processors do have the concept of signed/unsigned, but unlike in C language, where this information is baked into the variable type definition, processor registers do not hold the type/signed/unsigned information. After all, registers are just placeholders of data. But the instructions themselves do have the notion of signed/unsigned, That is why we have signed 'imul', unsigned 'mul', signed (JL/JG) vs unsigned(JB/JA) jump instructions. This is an important distinction between programming languages and machine code. It helps in understanding how high-level code gets translated to underlying machine code. Instructions which produce/consume signed data understand that -ve numbers should be in 2's complement form. So if a register (8bit) has 11110110 it's up to the instruction to treat that data as either -10 (signed) or 246 (unsigned). If an operation results in a -ve result say -5 then destination register will be written 11111011(2's complement representation of -5). 'add' and 'sub' instructions themselves are not affected by signed and unsigned numbers because of modulo arithmetic.

3 How signed-ness is interpreted in assembly?

In one of the above paragraphs, we said, processor registers does not hold any signed/unsigned type information with them and it's up to the instructions to interpret the data as either signed or unsigned.

But the interesting thing about x86/x64 processors is, they provide a flag register(EFLAGS) where some of the bits are called Status Flags. These bits represent the status of what happened with the previous instruction.

The most important flags of these for our discussion are

1. Carry flag - Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. **This flag indicates an overflow condition for unsigned-integer arithmetic**.
2. Sign flag - Set equal to the most significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
3. Overflow flag - Set if the integer result is **too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand** cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic

Of the above three status flags, Carry flag is obvious but other two flags need some explanation. Sign flag is set irrespective of whether you are treating the numbers as positive or negative, All it cares is if the result's MSB is set or not.

For example in below code, 'sub' can treat 0x81 as -127 or +129. So the result can also be either interpreted as -128 or +128, But the processor after executing 'sub' indicates this possibility in Sign Flag. So operations(JA/JB) who want to treat the result an unsigned will ignore this sign flag whereas operations (like JG/JL) who treat it as signed will take this sign flag into account.

```
mov al, 0x81
sub al, 0x1     // This triggers sign flag
```

Overflow flag is a little different. First of all, the Overflow flag is not set when an operation results in an overflow of the number that can be represented in a register. Like for example, if we try to add 1 to 0xff does not set this bit.

```
mov al, 0xFF // 0b11111111
add al, 0x1  // This will not trigger Overflow flag
```

Overflow flag is mainly meant for indicating if there is a chance of result if interpreted as signed will cross beyond what can be represented in the register. for example if we take a byte(al) it can contain value from 0b00000000(0) to 0b11111111(255) as unsigned or 0b10000000(-128) to 0b01111111(127) as signed number. Now overflow flag is set when a result goes beyond 0b01111111 i.e., too large a positive number and less than 0b10000000 i.e., too small a negative number

```
mov al, 0x7e // This is always positive
add al, 0x2  // This will trigger overflow flag
```

Let us see how overflow flag is used to determine the below conditional when var1 and var2 are signed numbers.

```
if (var1 < var2) {
    printf("var1 < var2");
}
```

The above code is translated to cmp followed by jl. cmp does var1-var2. And jl says execute printf if SF != OF. Now lets see the combinations why SF needs to be not equal to OF

```
var1 = -125;
var2 = 10;
```

`var1-var2 => -135 =>` **which cannot be represented in a byte meaning the sign bit of the byte will not be set but the overflow bit gets set. So SF=0 and OF=1**

```
var1 = -120;
var2 = -100;
```

`var1-var2 => -20 =>` **this can be represented in a byte and its -ve so SF = 1 but because it is still under the range of singed numbers representable in a byte OF = 0**

That is why jl is defined as SF != OF, When this is true it means var1 <var2 even if var1 and var2 are signed(meaning -ve numbers)

4 Signed vs Unsigned integers in C

Signed numbers are the way -ve numbers are represented and unsigned numbers are the way in which +ve numbers are represented. If let's say a data type has 1 byte of storage then the possible bit representation will be from 0b00000000(0) - 0b11111111(255). Now, these 255 valid slots can be interpreted as all positive numbers or divide the range into two halves and call one half of the slot as positive and other as negative numbers. There are other alternate representations for -ve number but almost all systems use 2's complement notation to represent -ve numbers. The take away here is if we include -ve number in the 255 slots then we will represent from -128 to 127. According to 2's complement notation -128 is represented as 0b10000000(0x80) whereas 127 is represented as 0b01111111 (0x7F).

5 Integer Arithmetic Conversions in C

C language defines below set of rules to convert the arguments in an expression.

5.1 Rank

Every integer type has an integer conversion rank defined as follows:

1. No two signed integer types shall have the same rank, even if they have the same representation.
2. The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.
3. The rank of long long int shall be greater than the rank of long int, which shall be greater than the rank of int, which shall be greater than the rank of short int, which shall be greater than the rank of signed char.
4. The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.
5. The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.
6. The rank of char shall equal the rank of signed char and unsigned char.
7. The rank of _Bool shall be less than the rank of all other standard integer types.
8. The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).
9. The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.
10. For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 has greater rank than T3.

At a high level the gist of above rules is as follows

```
Rank(signed char) ==  Rank(unsigned char) <  Rank(signed short) ==  Rank(unsigned
↪  short) <  Rank(signed int) ==  Rank(unsigned int) <  Rank(signed long) ==
↪  Rank(unsigned long) <  Rank(signed long long) ==  Rank(unsigned long long)
```

5.2 Integer Promotions

The following may be used in an expression wherever an int or unsigned int may be used:

- An object or expression with an integer type whose integer conversion rank is less than or equal to the rank of int and unsigned int.
- A bit-field of type _Bool, int, signed int, or unsigned int.

- 1a) If an int can represent all values of the original type, the value is converted to an int;
- 1b) otherwise, it is converted to an unsigned int. These are called the integer promotions. All other types are unchanged by the integer promotions.

5.3 Usual arithmetic conversions

- 2a) If both operands have the same type, then no further conversion is needed.
- 2b) Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
- 2c) Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
- 2d) Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with a signed integer type.
- 2e) Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

5.3.1 Rule 2c interpretation

The gist of 2c is to make sense of expressions like z = a+b where a is singed number and b is unsigned number. Now a 4 byte signed int is added to 8 byte unsigned long long, according to this rule 4 byte signed int will become 8 byte unsigned long long. The important point to remember here is conversion of signed int to unsigned long long, The way the compiler does this is by sign extend 4 byte signed number so that its absolute value will not change i.e.., -10 remain -10 whether it is a 4 byte or 8 byte. But according to these rules the expression will have the type unsigned long long so statements like below will result in unexpected behavior. Since the var1+var2 is an unsigned expression the compiler will enforce it by using a unsigned jump instruction called jae instead of jge instruction.

Whenever a signed data type has to be converted to unsigned data type the absolute value remains unchanged because of sign extension will be done using movsx instruction. So if a variable is type signed short with value -10, when this variable is sign extended to unsigned int the value of even after the conversion remains -10 when interpreted as singed int because the binary represent for -10 for 2 byte storage and 4 byte storage yields the same.

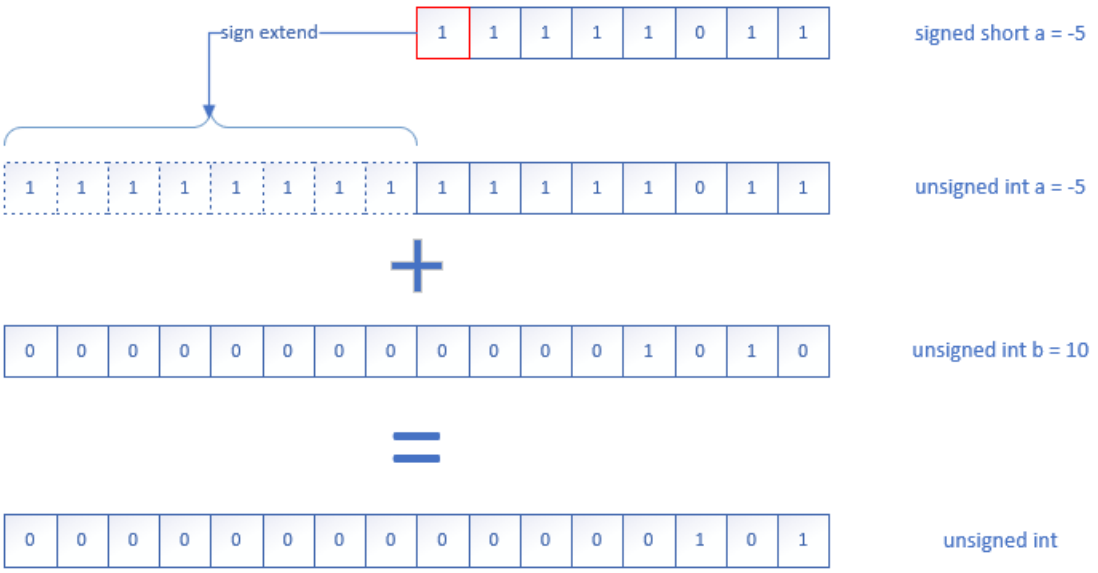

Figure 1: signed number converted to unsigned numbers

godbolt's compiler explorer is very helpful in understanding how the compiler translates expressions and apply the above rules. Clang AST view especially helps in graphically looking at how these promotions/conversion happen

```
signed int var1 = -100;
unsigned long long var2 = 10;
if (var1 + var2 < 0) { //resulting unsigned long long expression will never be
↪  lessthan 0
    printf("This will never get printed");
}
```

```
$LN4:
        sub     rsp, 56                                 ; 00000038H
        mov     DWORD PTR var1$[rsp], -100                      ; ffffffffffffff9cH
        mov     QWORD PTR var2$[rsp], 10
        movsxd  rax, DWORD PTR var1$[rsp]
        add     rax, QWORD PTR var2$[rsp]
        test    rax, rax
        jae     SHORT $LN2@main      //This is an unsigned jump
        lea     rcx, OFFSET FLAT:$SG4416
        call    printf
$LN2@main:
        xor     eax, eax
        add     rsp, 56                                 ; 00000038H
        ret     0
main    ENDP
```

```
Fragment of clang's AST for above program

      |-BinaryOperator <line:7:9, col:23> 'bool' '<'
      | |-BinaryOperator <col:9, col:16> 'unsigned long long' '+'
      | | |-ImplicitCastExpr <col:9> 'unsigned long long' <IntegralCast>
      | | | `-ImplicitCastExpr <col:9> 'int' <LValueToRValue>
      | | |   `-DeclRefExpr <col:9> 'int' lvalue Var 0x55ea65e030e0 'var1' 'int'
```

```
|   | `-ImplicitCastExpr <col:16> 'unsigned long long' <LValueToRValue>
|   |   `-DeclRefExpr <col:16> 'unsigned long long' lvalue Var 0x55ea65e031b0
↪  'var2' 'unsigned long long'
|   `-ImplicitCastExpr <col:23> 'unsigned long long' <IntegralCast>
|     `-IntegerLiteral <col:23> 'int' 0
```

These rules can only dictate what need to be done for each expression in a statement one at a time. It cannot determine the type of the complete statement at once because of this we might end up with unexpected results if we are not careful enough about how the type is propagating. for example,in var = exp1 + exp2*exp3 the rules can only tell how exp2*exp3 interact and how that result interacts with exp1. exp2*exp3's new type may not gel well with exp1 and might result in an unexpected result.

5.3.2 Rule 2d interpretation

Rule 2d tries to make sense when you have two variables with small the unsigned number and signed number are present in an expression and the singed storage can hold the unsigned number entirely, then by this rule, the unsigned number gets converted to signed number.

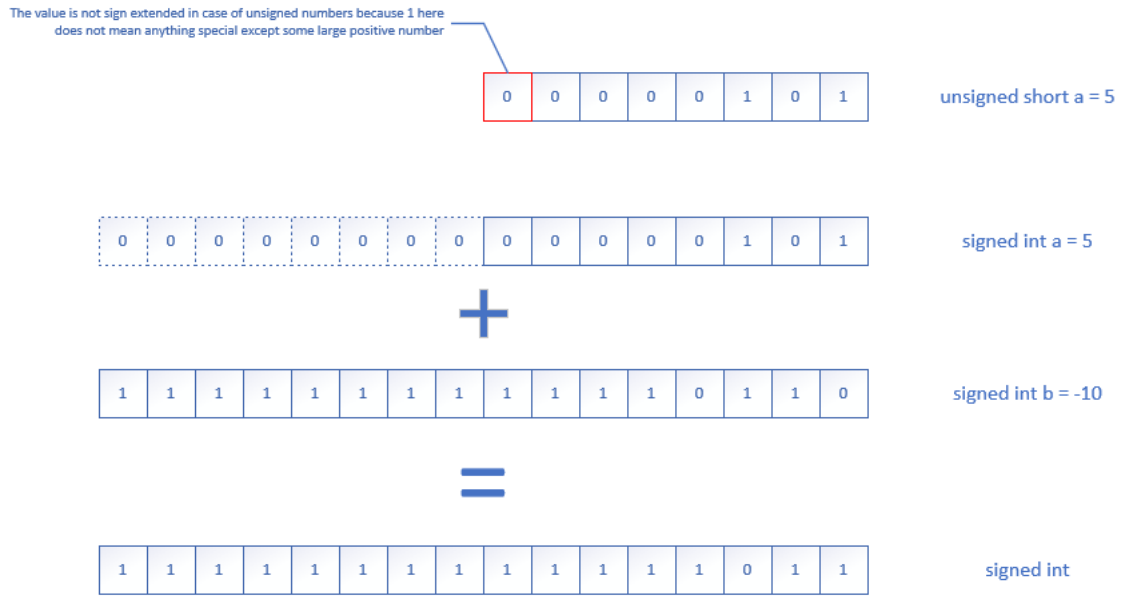

Figure 2: unsigned number converted to signed numbers

For example:

```
unsigned int var1 = 100;
signed long long var2 = -10;
if (var1 + var2 < 0) { //resulting in signed long long expression
    printf("This gets printed if var1 + var2 is < 0");
}
```

```
        sub     rsp, 56                         ; 00000038H
        mov     DWORD PTR var1$[rsp], 100   ; 00000064H
        mov     QWORD PTR var2$[rsp], -10
        mov     eax, DWORD PTR var1$[rsp]
        add     rax, QWORD PTR var2$[rsp]
        test    rax, rax
```

```
        jge     SHORT $LN2@main                 // This is a signed jump
        lea     rcx, OFFSET FLAT:$SG4869
        call    printf
$LN2@main:
        xor     eax, eax
        add     rsp, 56                                 ; 00000038H
        ret     0
main    ENDP
```

```
Fragment of clang's AST for above program

      |-BinaryOperator <line:7:9, col:23> 'bool' '<'
      | |-BinaryOperator <col:9, col:16> 'long long' '+'
      | | |-ImplicitCastExpr <col:9> 'long long' <IntegralCast>
      | | | `-ImplicitCastExpr <col:9> 'unsigned int' <LValueToRValue>
      | | |   `-DeclRefExpr <col:9> 'unsigned int' lvalue Var 0x56305e1200e0
      ↪  'var1' 'unsigned int'
      | | `-ImplicitCastExpr <col:16> 'long long' <LValueToRValue>
      | |   `-DeclRefExpr <col:16> 'long long' lvalue Var 0x56305e1201a8 'var2'
      ↪  'long long'
      | `-ImplicitCastExpr <col:23> 'long long' <IntegralCast>
      |   `-IntegerLiteral <col:23> 'int' 0
```

5.3.3 Rule 2e interpretation

And finally, Rule 2e is applied only between unsigned int and signed long because neither 2c,2d fits these type on LLP64 model. One interesting thing we can observe if read between the lines is, the resultant type has unsignedness but the type is of signed variable and also both operand types are converted unlike other rules. For example:

```
unsigned int var1 = 100; // this gets converted to unsigned long
signed long var2 = -10;  // this will also converted to unsigned long
if (var1 + var2 < 0) { //resulting type is unsigned long
    printf("This will not be printed");
}
```

```
        sub     rsp, 56                                 ; 00000038H
        mov     DWORD PTR var1$[rsp], 100       ; 00000064H
        mov     DWORD PTR var2$[rsp], -10
        mov     eax, DWORD PTR var2$[rsp]
        mov     ecx, DWORD PTR var1$[rsp]
        add     ecx, eax
        mov     eax, ecx
        test    eax, eax
        jae     SHORT $LN2@main                 // This is an unsigned jump
        lea     rcx, OFFSET FLAT:$SG4869
        call    printf
$LN2@main:
        xor     eax, eax
        add     rsp, 56                                 ; 00000038H
        ret     0
main    ENDP
```

```
Fragment of clang's AST for above program
```

```
|-BinaryOperator <line:7:9, col:23> 'bool' '<'
| |-BinaryOperator <col:9, col:16> 'long' '+'
| | |-ImplicitCastExpr <col:9> 'long' <IntegralCast>
| | | `-ImplicitCastExpr <col:9> 'unsigned int' <LValueToRValue>
| | |   `-DeclRefExpr <col:9> 'unsigned int' lvalue Var 0x564cedbd0100
↪  'var1' 'unsigned int'
| | `-ImplicitCastExpr <col:16> 'long' <LValueToRValue>
| |   `-DeclRefExpr <col:16> 'long' lvalue Var 0x564cedbd01c8 'var2' 'long'
| `-ImplicitCastExpr <col:23> 'long' <IntegralCast>
|   `-IntegerLiteral <col:23> 'int' 0
```

5.4 Integer conversion matrix on LLP64 Programming Model

Below table summaries the rules applied for each expression combination

LLP64 Programming Model		Size(MS VC x64)	1	1	2	2	4	4	4	4	8	8
		Rank	1	1	2	2	3	3	4	4	5	5
Size(MS VC x64)	Rank	<type1> oper <type2>	signed char	unsigned char	signed short	unsigned short	signed int	unsigned int	signed long	unsigned long	signed long long	unsigned long long
1	1	signed char	1a	1a	1a	1a	1a	1b	2b	2c	2b	2c
1	1	unsigned char		1a	1a	1a	1a	1b	2d	2b	2d	2b
2	2	signed short			1a	1a	1a	1b	2b	2c	2b	2c
2	2	unsigned short				1a	1a	1b	2d	2b	2d	2b
4	3	signed int					2a	1b	2b	2c	2b	2c
4	3	unsigned int						2a	2e	2b	2d	2b
4	4	signed long							2a	2c	2b	2c
4	4	unsigned long								2a	2d	2b
8	5	signed long long									2a	2c
8	5	unsigned long long										2a

Figure 3: Rules applied for each combination of expression

Below table summaries the resulting data type of the expression

LLP64 Programming Model		Size(MS VC x64)	1	1	2	2	4	4	4	4	8	8
		Rank	1	1	2	2	3	3	4	4	5	5
Size(MS VC x64)	Rank	<type1> oper <type2>	signed char	unsigned char	signed short	unsigned short	signed int	unsigned int	signed long	unsigned long	signed long long	unsigned long long
1	1	signed char	signed int	signed int	signed int	signed int	signed int	unsigned int	signed long	unsigned long	signed long long	unsigned long long
1	1	unsigned char		signed int	signed int	signed int	signed int	unsigned int	signed long	unsigned long	signed long long	unsigned long long
2	2	signed short			signed int	signed int	signed int	unsigned int	signed long	unsigned long	signed long long	unsigned long long
2	2	unsigned short				signed int	signed int	unsigned int	signed long	unsigned long	signed long long	unsigned long long
4	3	signed int					signed int	unsigned int	signed long	unsigned long	signed long long	unsigned long long
4	3	unsigned int						unsigned int	unsigned long	unsigned long	signed long long	unsigned long long
4	4	signed long							signed long	unsigned long	signed long long	unsigned long long
4	4	unsigned long								unsigned long	signed long long	unsigned long long
8	5	signed long long									signed long long	unsigned long long
8	5	unsigned long long										unsigned long long

Figure 4: Resulting type of the expression

6 References

1. INT02-C. Understand integer conversion rules
2. C99 Standard

3. Should I use Signed or Unsigned Ints In C? (Part 1)
4. Should I use Signed or Unsigned Ints In C? (Part 2)
5. X86 Opcode and Instruction Reference
6. Why is imul used for multiplying unsigned numbers?
7. System V Application Binary Interface AMD64 Architecture Processor Supplement
8. Why did the Win64 team choose the LLP64 model?
9. Abstract Data Models

7 Binary representation

7.1 unsigned numbers

0|0x0 |00000000	54|0x36|00110110	108|0x6c|01101100	162|0xa2|10100010
1|0x1 |00000001	55|0x37|00110111	109|0x6d|01101101	163|0xa3|10100011
2|0x2 |00000010	56|0x38|00111000	110|0x6e|01101110	164|0xa4|10100100
3|0x3 |00000011	57|0x39|00111001	111|0x6f|01101111	165|0xa5|10100101
4|0x4 |00000100	58|0x3a|00111010	112|0x70|01110000	166|0xa6|10100110
5|0x5 |00000101	59|0x3b|00111011	113|0x71|01110001	167|0xa7|10100111
6|0x6 |00000110	60|0x3c|00111100	114|0x72|01110010	168|0xa8|10101000
7|0x7 |00000111	61|0x3d|00111101	115|0x73|01110011	169|0xa9|10101001
8|0x8 |00001000	62|0x3e|00111110	116|0x74|01110100	170|0xaa|10101010
9|0x9 |00001001	63|0x3f|00111111	117|0x75|01110101	171|0xab|10101011
10|0xa |00001010	64|0x40|01000000	118|0x76|01110110	172|0xac|10101100
11|0xb |00001011	65|0x41|01000001	119|0x77|01110111	173|0xad|10101101
12|0xc |00001100	66|0x42|01000010	120|0x78|01111000	174|0xae|10101110
13|0xd |00001101	67|0x43|01000011	121|0x79|01111001	175|0xaf|10101111
14|0xe |00001110	68|0x44|01000100	122|0x7a|01111010	176|0xb0|10110000
15|0xf |00001111	69|0x45|01000101	123|0x7b|01111011	177|0xb1|10110001
16|0x10|00010000	70|0x46|01000110	124|0x7c|01111100	178|0xb2|10110010
17|0x11|00010001	71|0x47|01000111	125|0x7d|01111101	179|0xb3|10110011
18|0x12|00010010	72|0x48|01001000	126|0x7e|01111110	180|0xb4|10110100
19|0x13|00010011	73|0x49|01001001	127|0x7f|01111111	181|0xb5|10110101
20|0x14|00010100	74|0x4a|01001010	128|0x80|10000000	182|0xb6|10110110
21|0x15|00010101	75|0x4b|01001011	129|0x81|10000001	183|0xb7|10110111
22|0x16|00010110	76|0x4c|01001100	130|0x82|10000010	184|0xb8|10111000
23|0x17|00010111	77|0x4d|01001101	131|0x83|10000011	185|0xb9|10111001
24|0x18|00011000	78|0x4e|01001110	132|0x84|10000100	186|0xba|10111010
25|0x19|00011001	79|0x4f|01001111	133|0x85|10000101	187|0xbb|10111011
26|0x1a|00011010	80|0x50|01010000	134|0x86|10000110	188|0xbc|10111100
27|0x1b|00011011	81|0x51|01010001	135|0x87|10000111	189|0xbd|10111101
28|0x1c|00011100	82|0x52|01010010	136|0x88|10001000	190|0xbe|10111110
29|0x1d|00011101	83|0x53|01010011	137|0x89|10001001	191|0xbf|10111111
30|0x1e|00011110	84|0x54|01010100	138|0x8a|10001010	192|0xc0|11000000
31|0x1f|00011111	85|0x55|01010101	139|0x8b|10001011	193|0xc1|11000001
32|0x20|00100000	86|0x56|01010110	140|0x8c|10001100	194|0xc2|11000010
33|0x21|00100001	87|0x57|01010111	141|0x8d|10001101	195|0xc3|11000011
34|0x22|00100010	88|0x58|01011000	142|0x8e|10001110	196|0xc4|11000100
35|0x23|00100011	89|0x59|01011001	143|0x8f|10001111	197|0xc5|11000101
36|0x24|00100100	90|0x5a|01011010	144|0x90|10010000	198|0xc6|11000110
37|0x25|00100101	91|0x5b|01011011	145|0x91|10010001	199|0xc7|11000111
38|0x26|00100110	92|0x5c|01011100	146|0x92|10010010	200|0xc8|11001000
39|0x27|00100111	93|0x5d|01011101	147|0x93|10010011	201|0xc9|11001001
40|0x28|00101000	94|0x5e|01011110	148|0x94|10010100	202|0xca|11001010
41|0x29|00101001	95|0x5f|01011111	149|0x95|10010101	203|0xcb|11001011
42|0x2a|00101010	96|0x60|01100000	150|0x96|10010110	204|0xcc|11001100
43|0x2b|00101011	97|0x61|01100001	151|0x97|10010111	205|0xcd|11001101
44|0x2c|00101100	98|0x62|01100010	152|0x98|10011000	206|0xce|11001110
45|0x2d|00101101	99|0x63|01100011	153|0x99|10011001	207|0xcf|11001111
46|0x2e|00101110	100|0x64|01100100	154|0x9a|10011010	208|0xd0|11010000
47|0x2f|00101111	101|0x65|01100101	155|0x9b|10011011	209|0xd1|11010001
48|0x30|00110000	102|0x66|01100110	156|0x9c|10011100	210|0xd2|11010010
49|0x31|00110001	103|0x67|01100111	157|0x9d|10011101	211|0xd3|11010011
50|0x32|00110010	104|0x68|01101000	158|0x9e|10011110	212|0xd4|11010100
51|0x33|00110011	105|0x69|01101001	159|0x9f|10011111	213|0xd5|11010101
52|0x34|00110100	106|0x6a|01101010	160|0xa0|10100000	214|0xd6|11010110
53|0x35|00110101	107|0x6b|01101011	161|0xa1|10100001	215|0xd7|11010111

216|0xd8|11011000	226|0xe2|11100010	236|0xec|11101100	246|0xf6|11110110
217|0xd9|11011001	227|0xe3|11100011	237|0xed|11101101	247|0xf7|11110111
218|0xda|11011010	228|0xe4|11100100	238|0xee|11101110	248|0xf8|11111000
219|0xdb|11011011	229|0xe5|11100101	239|0xef|11101111	249|0xf9|11111001
220|0xdc|11011100	230|0xe6|11100110	240|0xf0|11110000	250|0xfa|11111010
221|0xdd|11011101	231|0xe7|11100111	241|0xf1|11110001	251|0xfb|11111011
222|0xde|11011110	232|0xe8|11101000	242|0xf2|11110010	252|0xfc|11111100
223|0xdf|11011111	233|0xe9|11101001	243|0xf3|11110011	253|0xfd|11111101
224|0xe0|11100000	234|0xea|11101010	244|0xf4|11110100	254|0xfe|11111110
225|0xe1|11100001	235|0xeb|11101011	245|0xf5|11110101	255|0xff|11111111

7.2 signed numbers

-128|0x80|10000000	-72|0xb8|10111000	-16|0xf0|11110000	40|0x28|00101000
-127|0x81|10000001	-71|0xb9|10111001	-15|0xf1|11110001	41|0x29|00101001
-126|0x82|10000010	-70|0xba|10111010	-14|0xf2|11110010	42|0x2a|00101010
-125|0x83|10000011	-69|0xbb|10111011	-13|0xf3|11110011	43|0x2b|00101011
-124|0x84|10000100	-68|0xbc|10111100	-12|0xf4|11110100	44|0x2c|00101100
-123|0x85|10000101	-67|0xbd|10111101	-11|0xf5|11110101	45|0x2d|00101101
-122|0x86|10000110	-66|0xbe|10111110	-10|0xf6|11110110	46|0x2e|00101110
-121|0x87|10000111	-65|0xbf|10111111	-9|0xf7|11110111	47|0x2f|00101111
-120|0x88|10001000	-64|0xc0|11000000	-8|0xf8|11111000	48|0x30|00110000
-119|0x89|10001001	-63|0xc1|11000001	-7|0xf9|11111001	49|0x31|00110001
-118|0x8a|10001010	-62|0xc2|11000010	-6|0xfa|11111010	50|0x32|00110010
-117|0x8b|10001011	-61|0xc3|11000011	-5|0xfb|11111011	51|0x33|00110011
-116|0x8c|10001100	-60|0xc4|11000100	-4|0xfc|11111100	52|0x34|00110100
-115|0x8d|10001101	-59|0xc5|11000101	-3|0xfd|11111101	53|0x35|00110101
-114|0x8e|10001110	-58|0xc6|11000110	-2|0xfe|11111110	54|0x36|00110110
-113|0x8f|10001111	-57|0xc7|11000111	-1|0xff|11111111	55|0x37|00110111
-112|0x90|10010000	-56|0xc8|11001000	0|0x0 |00000000	56|0x38|00111000
-111|0x91|10010001	-55|0xc9|11001001	1|0x1 |00000001	57|0x39|00111001
-110|0x92|10010010	-54|0xca|11001010	2|0x2 |00000010	58|0x3a|00111010
-109|0x93|10010011	-53|0xcb|11001011	3|0x3 |00000011	59|0x3b|00111011
-108|0x94|10010100	-52|0xcc|11001100	4|0x4 |00000100	60|0x3c|00111100
-107|0x95|10010101	-51|0xcd|11001101	5|0x5 |00000101	61|0x3d|00111101
-106|0x96|10010110	-50|0xce|11001110	6|0x6 |00000110	62|0x3e|00111110
-105|0x97|10010111	-49|0xcf|11001111	7|0x7 |00000111	63|0x3f|00111111
-104|0x98|10011000	-48|0xd0|11010000	8|0x8 |00001000	64|0x40|01000000
-103|0x99|10011001	-47|0xd1|11010001	9|0x9 |00001001	65|0x41|01000001
-102|0x9a|10011010	-46|0xd2|11010010	10|0xa |00001010	66|0x42|01000010
-101|0x9b|10011011	-45|0xd3|11010011	11|0xb |00001011	67|0x43|01000011
-100|0x9c|10011100	-44|0xd4|11010100	12|0xc |00001100	68|0x44|01000100
-99|0x9d|10011101	-43|0xd5|11010101	13|0xd |00001101	69|0x45|01000101
-98|0x9e|10011110	-42|0xd6|11010110	14|0xe |00001110	70|0x46|01000110
-97|0x9f|10011111	-41|0xd7|11010111	15|0xf |00001111	71|0x47|01000111
-96|0xa0|10100000	-40|0xd8|11011000	16|0x10|00010000	72|0x48|01001000
-95|0xa1|10100001	-39|0xd9|11011001	17|0x11|00010001	73|0x49|01001001
-94|0xa2|10100010	-38|0xda|11011010	18|0x12|00010010	74|0x4a|01001010
-93|0xa3|10100011	-37|0xdb|11011011	19|0x13|00010011	75|0x4b|01001011
-92|0xa4|10100100	-36|0xdc|11011100	20|0x14|00010100	76|0x4c|01001100
-91|0xa5|10100101	-35|0xdd|11011101	21|0x15|00010101	77|0x4d|01001101
-90|0xa6|10100110	-34|0xde|11011110	22|0x16|00010110	78|0x4e|01001110
-89|0xa7|10100111	-33|0xdf|11011111	23|0x17|00010111	79|0x4f|01001111
-88|0xa8|10101000	-32|0xe0|11100000	24|0x18|00011000	80|0x50|01010000
-87|0xa9|10101001	-31|0xe1|11100001	25|0x19|00011001	81|0x51|01010001
-86|0xaa|10101010	-30|0xe2|11100010	26|0x1a|00011010	82|0x52|01010010
-85|0xab|10101011	-29|0xe3|11100011	27|0x1b|00011011	83|0x53|01010011
-84|0xac|10101100	-28|0xe4|11100100	28|0x1c|00011100	84|0x54|01010100
-83|0xad|10101101	-27|0xe5|11100101	29|0x1d|00011101	85|0x55|01010101
-82|0xae|10101110	-26|0xe6|11100110	30|0x1e|00011110	86|0x56|01010110
-81|0xaf|10101111	-25|0xe7|11100111	31|0x1f|00011111	87|0x57|01010111
-80|0xb0|10110000	-24|0xe8|11101000	32|0x20|00100000	88|0x58|01011000
-79|0xb1|10110001	-23|0xe9|11101001	33|0x21|00100001	89|0x59|01011001
-78|0xb2|10110010	-22|0xea|11101010	34|0x22|00100010	90|0x5a|01011010
-77|0xb3|10110011	-21|0xeb|11101011	35|0x23|00100011	91|0x5b|01011011
-76|0xb4|10110100	-20|0xec|11101100	36|0x24|00100100	92|0x5c|01011100
-75|0xb5|10110101	-19|0xed|11101101	37|0x25|00100101	93|0x5d|01011101
-74|0xb6|10110110	-18|0xee|11101110	38|0x26|00100110	94|0x5e|01011110
-73|0xb7|10110111	-17|0xef|11101111	39|0x27|00100111	95|0x5f|01011111

96|0x60|01100000	104|0x68|01101000	112|0x70|01110000	120|0x78|01111000
97|0x61|01100001	105|0x69|01101001	113|0x71|01110001	121|0x79|01111001
98|0x62|01100010	106|0x6a|01101010	114|0x72|01110010	122|0x7a|01111010
99|0x63|01100011	107|0x6b|01101011	115|0x73|01110011	123|0x7b|01111011
100|0x64|01100100	108|0x6c|01101100	116|0x74|01110100	124|0x7c|01111100
101|0x65|01100101	109|0x6d|01101101	117|0x75|01110101	125|0x7d|01111101
102|0x66|01100110	110|0x6e|01101110	118|0x76|01110110	126|0x7e|01111110
103|0x67|01100111	111|0x6f|01101111	119|0x77|01110111	127|0x7f|01111111