Computer Arithmetic

INTEGER NUMBERS

UNSIGNED INTEGER NUMBERS

- $n-bit$ number: $b_{n-1}b_{n-2}\dots b_0$.
- Here, we represent 2^n integer positive numbers from 0 to 2^n-1.

SIGNED INTEGER NUMBERS

- n-bit number $b_{n-1}b_{n-2}\dots b_1b_0$.
- Here, we represent integer positive and negative numbers. There exist three common representations: sign-and-magnitude, 1's complement, and 2's complement. In these 3 cases, the MSB always specifies whether the number is positive (MSB=0) or negative (MSB=1).
- It is common to refer to signed numbers as numbers represented in 2's complement arithmetic.

SIGN-AND-MAGNITUDE (SM):

- Here, the sign and the magnitude are represented separately.
- The MSB only represents the sign and the remaining $n-1$ bits the magnitude. With n bits, we can represent 2^n-1 numbers.
- **Example** (n=4): `0110 = +6`   `1110 = -6`

1'S COMPLEMENT (1C) and 2'S COMPLEMENT (2C):

- If MSB=0 $\rightarrow$ the number is positive and the remaining $n-1$ bits represent the magnitude.
- If MSB=1 $\rightarrow$ the number is negative and the remaining $n-1$ bits do not represent the magnitude.
- When using the 1C or the 2C representations, it is mandatory to specify the number of bits being used. If not, assume the minimum possible number of bits.

	1'S COMPLEMENT	2'S COMPLEMENT
Range of values	$-2^{n-1}+1\ to\ 2^{n-1}-1$	$-2^{n-1}\ to\ 2^{n-1}-1$
Numbers represented	2^n-1	2^n
Inverting sign of a number	Apply 1C *operation*: invert all bits	Apply 2C *operation*: invert all bits and add 1
Examples	✓ `+6=0110` → `-6=1001` ✓ `+5=0101` → `-5=1010` ✓ `+7=0111` → `-7=1000`	✓ `+6=0110` → `-6=1010` ✓ `+5=0101` → `-5=1011` ✓ `+7=0111` → `-7=1001`
	✓ If `-6=1001`, we get +6 by applying the 1C *operation* to `1001` → `+6 = 0110`.	✓ If `-6=1010`, we get +6 by applying the 2C *operation* to `1010` → `+6 = 0110`.
	✓ Represent -4 in 1C: We know that `+4=0100`. To get -4, we apply the 1C operation to `0100`. Thus, `-4 = 1011`.	✓ Represent -4 in 2C: We know that `+4=0100`. To get -4, we apply the 2C operation to `0100`. Thus `-4 = 1100`.
	✓ Represent 8 in 1C: This is a positive number → MSB=0. The remaining $n-1$ bits represent the magnitude. Magnitude (unsigned number) with a min. of 4 bits: $8=1000_2$. Thus, with a minimum of 5 bits, $8=01000_2$ (1C). ✓ What is the decimal value of `1100`? We first apply the 1C *operation* (or take the 1's complement) to `1100`, which results in `0011(+3)`. Thus `1100=-3`.	✓ Represent 12 in 2C: This is a positive number → MSB=0. The remaining $n-1$ bits represent the magnitude. Magnitude (unsigned number) with a min. of 4 bits: $12=1100_2$. Thus, with a minimum of 5 bits, $12=01100_2$ (2C). ✓ What is the decimal value of `1101`? We first apply the 2C *operation* (or take the 2's complement) to `1101`, which results in `0011(+3)`. Thus `1101=-3`.

Getting the decimal value of a number in 2C representation:

- If the number B is positive, then MSB=0: $b_{n-1}=0$.

$$\rightarrow B=\sum_{i=0}^{n-1}b_i2^i=b_{n-1}2^{n-1}+\sum_{i=0}^{n-2}b_i2^i=\sum_{i=0}^{n-2}b_i2^i \quad (a)$$

- If the number B is negative, $b_{n-1}=1$ (MSB=1). If we take the 2's complement of B, we get K (which is a positive number). In 2's complement representation, K *represents* $-B$. Using $K=2^n-B$ (K and B are treated as unsigned numbers):

$$\sum_{i=0}^{n-1} k_i 2^i = 2^n - \sum_{i=0}^{n-1} b_i 2^i$$

- We want to express $-K$ in terms of b_i, since the integer value $-K$ is the actual integer value of B.

$$-K = -\sum_{i=0}^{n-1} k_i 2^i = \sum_{i=0}^{n-1} b_i 2^i - 2^n = b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i - 2^n = 2^{n-1}(b_{n-1} - 2) + \sum_{i=0}^{n-2} b_i 2^i$$

$$B = -K = 2^{n-1}(1-2) + \sum_{i=0}^{n-2} b_i 2^i = -2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \ (b)$$

- Using (a) and (b), the formula for the decimal value of B (either positive or negative) is:

$$B = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

- **Examples**: $10110_2 = -2^4 + 2^2 + 2^1 = -10$ $\quad 11000_2 = -2^4 + 2^3 = -8$

SUMMARY

- The following table summarizes the signed representations for a 4-bit number:

n=4: $b_3b_2b_1b_0$	SIGNED REPRESENTATION: Sign-and-magnitude	1's complement	2's complement
0 0 0 0	0	0	0
0 0 0 1	1	1	1
0 0 1 0	2	2	2
0 0 1 1	3	3	3
0 1 0 0	4	4	4
0 1 0 1	5	5	5
0 1 1 0	6	6	6
0 1 1 1	7	7	7
1 0 0 0	0	-7	-8
1 0 0 1	-1	-6	-7
1 0 1 0	-2	-5	-6
1 0 1 1	-3	-4	-5
1 1 0 0	-4	-3	-4
1 1 0 1	-5	-2	-3
1 1 1 0	-6	-1	-2
1 1 1 1	-7	0	-1
Range for n bits:	$[-(2^{n-1}-1), 2^{n-1}-1]$	$[-(2^{n-1}-1), 2^{n-1}-1]$	$[-2^{n-1}, 2^{n-1}-1]$

- Keep in mind that 1C (or 2C) representation and the 1C (or 2C) operation are very different concepts.
- Note that the sign-and-magnitude and the 1C representations have a redundant representation for zero. This is not the case in 2C, which can represent an extra number.
- **Special case in 2C**: If -2^{n-1} is represented with n bits, the number 2^{n-1}requires $n+1$ bits. For example, the number -8 can be represented with 4 bits: `-8=1000`. To obtain +8, we apply the 2C operation to `1000`, which results in `1000`. But `1000` cannot be a positive number. This means that we require 5 bits to represent `+8=01000`.
- **Representation of Integer Numbers with n bits**: $b_{n-1}b_{n-2} \dots b_0$.

	UNSIGNED	SIGNED (2C)
Decimal Value	$D = \sum_{i=0}^{n-1} b_i 2^i$	$D = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} b_i 2^i$
Range of values	$[0, 2^n - 1]$	$[-2^{n-1}, 2^{n-1}-1]$

SIGN EXTENSION

- **UNSIGNED NUMBERS**: Here, if we want to use more bits, we just append zeros to the left.
  Example: $12 = 1100_2$ with 4 bits. If we want to use 6 bits, then $12 = 001100_2$.

- **SIGNED NUMBERS:**
  - ✓ **Sign-and-magnitude**: The MSB only represents the sign. If we want to use more bits, we append zeros to the left, with the MSB (leftmost bit) always being the sign.
    Example: $-12 = 11100_2$ with 5 bits. If we want to use 7 bits, then $-12 = 1001100_2$.

  - ✓ **2's complement** (also applies to 1C): In many circumstances, we might want to represent numbers in 2's complement with a certain number of bits. For example, the following two numbers require a minimum of 5 bits:
    $10111_2 = -2^4 + 2^2 + 2^1 + 2^0 = -9$ $\quad 01111_2 = 2^3 + 2^2 + 2^1 + 2^0 = +15$

What if we want to use 8 bits to represent them? In 2C, we sign-extend: If the number is positive, we append 0's to the left. If the number is negative, we attach 1's to the left. In the examples, we copied the MSB three times to the left:
$11110111_2 = -2^4 + 2^2 + 2^1 + 2^0 = -9$ $\quad$ $00001111_2 = 2^3 + 2^2 + 2^1 + 2^0 = +15$

Demonstration of sign-extension in 2C arithmetic:

- To increase the number of bits for representing a number, we append the MSB to the left as many times as needed:
$$b_{n-1}b_{n-2} \dots b_0 \equiv b_{n-1} \dots b_{n-1}b_{n-1}b_{n-2} \dots b_0$$
Examples: $00101_2 = 0000101_2 = 2^2 + 2^0 = 5$
$10101_2 = 1110101_2 = -2^4 + 2^2 + 2^0 = -2^6 + 2^5 + 2^4 + 2^2 + 2^0 = -11$

We can think of the sign-extended number as an m-bit number, where $m > n$:
$$b_{n-1} \dots b_{n-1}b_{n-1}b_{n-2} \dots b_0 = b_{m-1} \dots b_n b_{n-1}b_{n-2} \dots b_0, \text{ where: } b_i = b_{n-1}, i = n, n+1, \dots, m-1$$

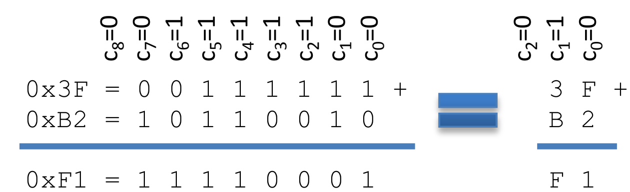
- We need to demonstrate that $b_{n-1}b_{n-2} \dots b_0$ represents the same decimal number as $b_{n-1} \dots b_{n-1}b_{n-1}b_{n-2} \dots b_0$, i.e., that the sign-extension is correct for any $m > n$.
We need that: $b_{m-1} \dots b_n b_{n-1}b_{n-2} \dots b_0 = b_{n-1} \dots b_{n-1}b_{n-1}b_{n-2} \dots b_0 = b_{n-1}b_{n-2} \dots b_0$

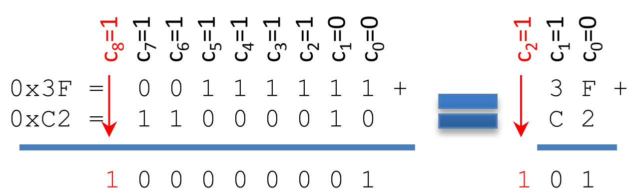
Using the formula for 2's complement numbers:
$$-2^{m-1}b_{m-1} + \sum_{i=0}^{m-2} 2^i b_i = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$
$$-2^{m-1}b_{m-1} + \sum_{i=n-1}^{m-2} 2^i b_i + \sum_{i=0}^{n-2} 2^i b_i = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i \Rightarrow -2^{m-1}b_{m-1} + \sum_{i=n-1}^{m-2} 2^i b_i = -2^{n-1}b_{n-1}$$
$$-2^{m-1}b_{n-1} + b_{n-1}\sum_{i=n-1}^{m-2} 2^i = -2^{n-1}b_{n-1},$$
$$Recall: \sum_{i=k}^{l} r^i = \frac{r^k - r^{l+1}}{1-r}, r \neq 1 \rightarrow \sum_{i=k}^{l} 2^i = \frac{2^k - 2^{l+1}}{1-2} = 2^{l+1} - 2^k$$
Then:
$$-2^{m-1}b_{n-1} + b_{n-1}(2^{m-1} - 2^{n-1}) = -2^{n-1}b_{n-1}$$
$$-2^{m-1}b_{n-1} + 2^{m-1}b_{n-1} - 2^{n-1}b_{n-1} = -2^{n-1}b_{n-1} \therefore -2^{n-1}b_{n-1} = -2^{n-1}b_{n-1}$$

ADDITION/SUBTRACTION

UNSIGNED NUMBERS

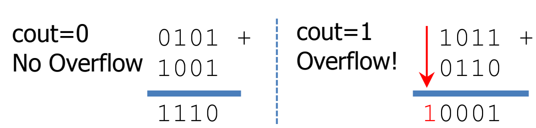
- The example depicts addition of two 8-bit numbers using binary and hexadecimal representations. Note that every summation of two digits (binary or hexadecimal) generates a carry when the summation requires more than one digit. Also, note that c_0 is the *carry in* of the summation (usually, c_0 is zero).
- The last carry (c_8 when n=8) is the *carry out* of the summation. If it is '0', it means that the summation can be represented with 8 bits. If it is '1', it means that the summation requires more than 8 bits (in fact 9 bits); this is called an overflow. In the example, we add two numbers and overflow occurs: an extra bit (in red) is required to correctly represent the summation. This *carry out* can also be used for *multi-precision addition*.



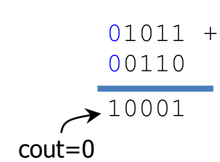


Arithmetic Overflow:

- Suppose we have only 4 bits to represent binary numbers. Overflow occurs when an arithmetic operation requires more bits than the bits we are using to represent our numbers. For 4 bits, the range is 0 to 15. If the summation is greater than 15, then there is overflow.



- For n bits, overflow occurs when the sum is greater than $2^n - 1$. Also: $overflow = c_n = c_{out}$.
Overflow is commonly avoided by sign-extending the two operators. For unsigned numbers, sign-extension amounts to zero-extension. For example, if the summands are 4-bits wide, then we append a 0 to both summands, using 5 bits to represent the summands (see figure on the right).



- For two n-bits summands, the result will have at most $n+1$ bits ($2^n - 1 + 2^n - 1 = 2^{n+1} - 2$).

Subtraction:

- In the example, we subtract two 8-bit numbers using the binary and hexadecimal (this is a short-hand notation) representations. A subtraction of two digits (binary or hexadecimal) generates a borrow when the difference is negative. So, we borrow 1 from the next digit so that the difference is positive. Recall that a borrow in a subtraction of two digits is an extra 1 that we need to subtract. Also, note that b_0 is the *borrow in* of the summation. This is usually zero.
- The last borrow (b_8 when n=8) is the *borrow out* of the subtraction. If it is zero, it means that the difference is positive and can be represented with 8 bits. If it is one, it means that the difference is negative and we need to borrow 1 from the next digit. In the example, we subtract two 8-bit numbers, the result we have borrows 1 from the next digit.

		$b_8=0$	$b_7=0$	$b_6=0$	$b_5=0$	$b_4=1$	$b_3=1$	$b_2=1$	$b_1=1$	$b_0=0$	
0x3A	=		0	0	1	1	1	0	1	0	-
0x2F	=		0	0	1	0	1	1	1	1	
0x0B	=		0	0	0	0	1	0	1	1	

$b_2=0$	$b_1=1$	$b_0=0$	
	3	A	-
	2	F	
	0	B	

		$b_8=1$	$b_7=1$	$b_6=0$	$b_5=0$	$b_4=0$	$b_3=1$	$b_2=0$	$b_1=1$	$b_0=0$	
0x3A	=		0	0	1	1	1	0	1	0	-
0x75	=		0	1	1	1	0	1	0	1	
0xC5	=		1	1	0	0	0	1	0	1	

$b_2=1$	$b_1=0$	$b_0=0$	
	3	A	-
	7	5	
	C	5	

- Subtraction using unsigned numbers only makes sense if the result is positive (or when doing <u>multi-precision subtraction</u>). In general, we prefer to use signed representation (2C) for subtraction.

SIGNED NUMBERS (2C REPRESENTATION)

- The advantage of the 2C representation is that the summation can be carried out using the same circuitry as that of the unsigned summation. Here the operands can be either positive or negative.
- The following are addition examples of two 4-bit signed numbers. Note that the *carry out* bit DOES NOT necessarily indicate overflow. In some cases, the carry out must be ignored, otherwise the result is incorrect.

```
+5 = 0101 +      -5 = 1011 +      +5 = 0101 +      -5 = 1011 +
+2 = 0010        +2 = 0010        -2 = 1110        -2 = 1110
+7 = 0111        -3 = 1101        +3 = X0011       -7 = X1001
cout=0           cout=0           cout=1           cout=1
```

- Now, we show addition examples of two 8-bit signed numbers. The *carry out* c_8 is not enough to determine overflow. Here, if $c_8 \neq c_7$ there is overflow. If $c_8=c_7$, no overflow and we can ignore c_8. Thus, the overflow bit is equal to c_8 XOR c_7.

- **Overflow**: It occurs when the summation falls outside the 2's complement range for 8 bits: $[-2^7, 2^7-1]$. If there is no overflow, the carry out bit must not be part of the result.

		$c_8=0$	$c_7=1$	$c_6=0$	$c_5=1$	$c_4=1$	$c_3=1$	$c_2=0$	$c_1=0$	$c_0=0$	
+92	=		0	1	0	1	1	1	0	0	+
+78	=		0	1	0	0	1	1	1	0	
+170	=	0	1	0	1	0	1	0	1	0	

overflow = $c_8 \oplus c_7=1$ -> overflow!

$+170 \notin [-2^7, 2^7-1]$ -> overflow!

		$c_8=1$	$c_7=0$	$c_6=1$	$c_5=0$	$c_4=0$	$c_3=0$	$c_2=0$	$c_1=0$	$c_0=0$	
-92	=		1	0	1	0	0	1	0	0	+
-78	=		1	0	1	1	0	0	1	0	
-170	=	1	0	1	0	1	0	1	1	0	

overflow = $c_8 \oplus c_7=1$ -> overflow!

$-170 \notin [-2^7, 2^7-1]$ -> overflow!

		$c_8=1$	$c_7=1$	$c_6=1$	$c_5=1$	$c_4=0$	$c_3=0$	$c_2=0$	$c_1=0$	$c_0=0$	
+92	=		0	1	0	1	1	1	0	0	+
-78	=		1	0	1	1	0	0	1	0	
+14	=	~~1~~	0	0	0	0	1	1	1	0	

overflow = $c_8 \oplus c_7=0$ -> no overflow

$+14 \in [-2^7, 2^7-1]$ -> no overflow

		$c_8=0$	$c_7=0$	$c_6=0$	$c_5=0$	$c_4=1$	$c_3=1$	$c_2=0$	$c_1=0$	$c_0=0$	
-92	=		1	0	1	0	0	1	0	0	+
+78	=		0	1	0	0	1	1	1	0	
-14	=	~~0~~	1	1	1	1	0	0	1	0	

overflow = $c_8 \oplus c_7=0$ -> no overflow

$-14 \in [-2^7, 2^7-1]$ -> no overflow

- To avoid overflow, a common technique is to sign-extend the two summands. For example, for two 4-bits summands, we add an extra bit; thereby using 5 bits to represent the operators.

		$c_5=0$	$c_4=0$	$c_3=1$	$c_2=1$	$c_1=0$	$c_0=0$	
+7	=		0	0	1	1	1	+
+2	=		0	0	0	1	0	
+9	=		0	1	0	0	1	

		$c_5=1$	$c_4=1$	$c_3=0$	$c_2=0$	$c_1=0$	$c_0=0$	
-7	=		1	1	0	0	1	+
-2	=		1	1	1	1	0	
-9	=		1	0	1	1	1	

Subtraction

- Note that $A - B = A + 2C(B)$. To subtract two signed (2C) numbers, we first apply the 2's complement operation to B (the subtrahend), and then add the numbers. So, in 2's complement arithmetic, subtraction ends up being an addition of two numbers.

7 - 3 = 7 + (-3):

```
+3=0011 → -3=1101

               c4=1 c3=1 c2=1 c1=1 c0=0
cout = 1       +7 =  0    1    1    1   +
overflow = 0   -3 =  1    1    0    1
               -------------------------
               +4 =  0    1    0    0
```

- For an n-bit number, overflow occurs when the summation/addition result is outside the range $[-2^{n-1}, 2^{n-1} - 1]$. The overflow bit can quickly be computed as $overflow = c_n \oplus c_{n-1}$. $c_n = c_{out}$.

- The largest value (in magnitude) of addition of two n-bits operators is $-2^{n-1} + (-2^{n-1}) = -2^n$. In the case of subtraction, the largest value (in magnitude) is $-2^{n-1} - (2^{n-1} - 1) = -2^n + 1$. Thus, the addition/subtraction of two n-bit operators needs at most $n + 1$ bits. $c_n = c_{out}$ is used in *multi-precision addition/subtraction*.

SUMMARY

- Addition/Subtraction of two n-bit numbers:

	UNSIGNED	**SIGNED (2C)**
Overflow bit	c_n	$c_n \oplus c_{n-1}$
Overflow occurs when:	$A + B \notin [0, 2^n - 1]$, $c_n = 1$	$(A \pm B) \notin [-2^{n-1}, 2^{n-1} - 1]$, $c_n \oplus c_{n-1} = 1$
Result range:	$[0, 2^{n+1} - 1]$	$A + B \in [-2^n, 2^n - 2]$, $A - B \in [-2^n + 1, 2^n - 2]$
Result requires at most:	$n + 1$ $bits$	

- In general, if one operand has n bits and the other has m bits, the result will have at most $max(n, m) + 1$. When adding both numbers, we first force (via sign-extension) the two operators to have the same number of bits: $max(n, m)$.

Multiplication of Integer Numbers

UNSIGNED NUMBERS

- Simple operation: first, generate the products, then add up all the columns (consider the carries).

```
                        a3    a2    a1    a0   x
                        b3    b2    b1    b0
                      -----------------------
                      a3b0  a2b0  a1b0  a0b0
                a3b1  a2b1  a1b1  a0b1
          a3b2  a2b2  a1b2  a0b2
    a3b3  a2b3  a1b3  a0b3
  -----------------------------------------------
  p7    p6    p5    p4    p3    p2    p1    p0
```

```
11 x  ➡          1 0 1 1 x
13               1 1 0 1
---      1 1 1 1 0 0 0 0
143              1 0 1 1
               0 0 0 0
             1 0 1 1
           1 0 1 1
         ---------------
         1 0 0 0 1 1 1 1
```

```
13 x  ➡              1 1 0 1 x
15                   1 1 1 1
---                  1 0 0 0
195      1 10 10 10  1 1 0 1
                   1 1 0 1
                 1 1 0 1
               1 1 0 1
             ---------------
             1 1 0 0 0 0 1 1
```

- If the two operators are n-bits wide, the maximum result is $(2^n - 1) \times (2^n - 1) = 2^{2n} - 2^{n+1} + 1$. Thus, in the worst case, the multiplication requires $2n$ bits.
- If one operator in n-bits wide and the other is m-bits wide, the maximum result is: $(2^n - 1) \times (2^m - 1) = 2^{n+m} - 2^n - 2^m + 1$. Thus, in the worst case, the multiplication requires $n + m$ bits.

SIGNED NUMBERS (2C)

- A straightforward implementation consists of checking the sign of the multiplicand and multiplier. If one or both are negative, we change the sign by applying the 2's complement operation. This way, we are left with unsigned multiplication.
- As for the final output: if only one of the inputs was negative, then we modify the sign of the output. Otherwise, the result of the unsigned multiplication is the final output.

```
1 0 1 x ➡     0 1 1 x       0 1 0 x ➡     0 1 0 x       1 1 1 x ➡     0 0 1 x            0 1 1 x
0 1 0         0 1 0         1 1 0         0 1 0         1 1 0         0 1 0              0 1 0
              -----                       -----                       -----              -----
              0 0 0                       0 0 0                       0 0 0              0 0 0
            0 1 1                       0 1 0                       0 0 1              0 1 1
          0 0 0                       0 0 0                       0 0 0              0 0 0
        -----------                 -----------                 -----------        -----------
        0 0 0 1 1 0                 0 0 0 1 0 0                 0 0 0 0 1 0        0 0 0 1 1 0
             ⬇                           ⬇
        1 1 1 0 1 0                 1 1 1 1 0 0
```

- **Note**: If one of the inputs is -2^{n-1}, then when we change the sign we get 2^{n-1}, which requires $n + 1$ bits. Here, we are allowed to use only n bits; in other words, we do not have to change its sign. This will not affect the final result since if we were to use $n + 1$ bits for 2^{n-1}, the MSB=0, which implies that the last row is full of zeros.

```
1 0 0 x  ➡   1 0 0 x        0 1 1 x  ➡   0 1 1 x        1 0 0 x  ➡   1 0 0 x
0 1 1        0 1 1          1 0 0        1 0 0          1 0 0        1 0 0
          --------                    --------                    --------
             1 0 0                       0 0 0                       0 0 0
           1 0 0                       0 0 0                       0 0 0
         0 0 0                       0 1 1                       1 0 0
       -----------                 -----------                 -----------
       0 0 1 1 0 0                 0 0 1 1 0 0                 0 1 0 0 0 0
           ⬇                           ⬇
       1 1 0 1 0 0                 1 1 0 1 0 0
```

- **Note**: If one input is negative and the other is positive, we can use the negative number as the multiplicand and the positive number as the multiplier. Then, we can operate as if it were unsigned multiplication, with the caveat that we need to sign extend each partial sum to $2n$ bits (if both operators are n-bits wide), or to $n + m$ (if one operator is n-bits wide and the other is m-bits wide).

```
1 0 0 1 x  ➡        1 0 0 1 x
0 1 1 0             0 1 1 0
                  ---------
            0 0 0 0 0 0 0 0
            1 1 1 1 0 0 1
            1 1 1 0 0 1
            0 0 0 0 0
           ----------------
            1 1 0 1 0 1 1 0
```

- For two n-bit operators, the final output requires $2n$ bits. Note that it is only because of the multiplication $-2^{n-1} \times -2^{n-1} = 2^{2n-2}$ that we require those $2n$ bits (in 2C representation).
- For an n-bit and an m-bit operator, the final output requires $n + m$ bits. Note that it is only because of the multiplication $-2^{n-1} \times -2^{m-1} = 2^{n+m-2}$ that we require those $n + m$ bits (in 2C representation).

DIVISION OF INTEGER NUMBERS

UNSIGNED NUMBERS

- The division of two unsigned integer numbers A/B (where A is the dividend and B the divisor), results in a quotient Q and a remainder R, where $A = B \times Q + R$. Most divider architectures provide Q and R as outputs.

```
        15  ⬅ Q                    00001111  ⬅ Q
B ➡ 9 ) 140 ⬅ A        B ➡ 1001 ) 10001100  ⬅ A
         90                         1001
        ---                        -----
         50                         10001
         45                          1001
        ---                         -----
          5 ⬅ R                      10000
                                      1001
                                     -----
                                       1110
                                       1001
                                      -----
                                        101 ⬅ R
```

ALGORITHM

```
R = 0
for i = n-1 downto 0
    left shift R (input = a_i)
    if R ≥ B
       q_i = 1, R ← R-B
    else
       q_i = 0
    end
end
```

- For n-bits dividend (A) and m-bits divisor (B):
  - ✓ The largest value for Q is $2^n - 1$ (by using $B = 1$). The smallest value for Q is 0. So, we use n bits for Q.
  - ✓ The remainder R is a value between 0 and $B - 1$. Thus, at most we use m bits for R.
  - ✓ If $A = 0$, $B \neq 0$, then $Q = R = 0$.
  - ✓ If $B = 0$, we have a division by zero. The result is undetermined.

- In computer arithmetic, integer division usually means getting $Q = \lfloor A/B \rfloor$.

- Examples:

```
157/10:          00001111
Q = 15    1010 ) 10011101
R = 7             1010
                 -----
                  10011
                   1010
                  -----
                   10010
                    1010
                   -----
                    10001
                     1010
                    -----
                      111
```

```
161/21:           00000111
Q = 7     10101 ) 10100001
R = 14             10101
                  ------
                   100110
                    10101
                   ------
                    100011
                     10101
                    ------
                      1110
```

```
337/46:             000000111
Q = 7     101110 ) 101010001
R = 15              101110
                   -------
                    1001100
                     101110
                    -------
                      111101
                      101110
                     -------
                        1111
```

```
418/20:            000010100
Q = 20    10100 ) 110100010
R = 18             10100
                  ------
                    11000
                    10100
                   ------
                     10010
```

SIGNED NUMBERS

- The division of two signed numbers $A/_B$ should result in Q and R such that $A = B \times Q + R$. As in signed multiplication, we first perform the unsigned division $|A|/_{|B|}$ and get Q' and R' such that: $|A| = |B| \times Q' + R'$. Then, to get Q and R, we apply:

	Quotient Q	Residue R	
$A \times B < 0$	$-Q'$	$-R'$	$A < 0, B > 0$
		R'	$A > 0, B < 0$
$A \times B \geq 0, B \neq 0$	Q'	R'	$A \geq 0, B > 0$
		$-R'$	$A < 0, B < 0$

- Important: To apply $Q = -Q' = 2C(Q')$, Q' must be in 2C representation. The same applies to $R = -R' = 2C(R')$. So, if $Q' = 1101 = 13$, we first turn this unsigned number into a signed number $\rightarrow Q' = 01101$. Then $Q = 2C(01101) = 10011 = -13$.

- Example: $\frac{011011}{0101} = \frac{27}{5}$
  - ✓ Convert both numerator and denominator into unsigned numbers: $\frac{11011}{101}$
  - ✓ $\frac{|A|}{|B|} \Rightarrow Q' = 101$, $R' = 10$. Note that these are unsigned numbers.
  - ✓ Get Q and R: $A \leq 0, B > 0 \rightarrow Q = Q' = 0101 = 5$, $R = R' = 010 = 2$.
    Note that Q and R are signed numbers.
  - ✓ Verification: $27 = 5 \times 5 + 2$.

```
       00101
101 ) 11011
      101
      ---
        111
        101
        ---
         10
```

- Example: $\frac{0101110}{1011} = \frac{46}{-5}$
  - ✓ Turn the denominator into a positive number $\rightarrow \frac{0101110}{0101}$
  - ✓ Convert both numerator and denominator into unsigned numbers: $\frac{101110}{101} = \frac{|A|}{|B|}$
  - ✓ $\frac{|A|}{|B|} \Rightarrow Q' = 1001$, $R' = 001$. Note that these are unsigned numbers.
  - ✓ Get Q and R: $A > 0, B < 0 \rightarrow Q = 2C(Q') = 2C(01001) = 10111 = -9$, $R = R' = 001 = 1$.
  - ✓ Verification: $46 = -5 \times -9 + 1$.

```
       001001
101 ) 101110
      101
      ---
         0110
          101
          ---
            1
```

- Example: $\frac{10110110}{01101} = \frac{-74}{13}$
  - ✓ Turn the numerator into a positive number $\rightarrow \frac{01001010}{01101}$
  - ✓ Convert both numerator and denominator into unsigned numbers: $\frac{1001010}{1101}$
  - ✓ $\frac{|A|}{|B|} \Rightarrow Q' = 101$, $R' = 1001$. Note that these are unsigned numbers.
  - ✓ Get Q and R: $A < 0, B > 0 \rightarrow Q = 2C(0101) = 1011 = -5$, $R = 2C(R') = 2C(01001) = 10111 = -9$.
  - ✓ Verification: $-74 = 13 \times -5 + (-9)$.

```
        0000101
1101 ) 1001010
        1101
        ----
         10110
          1101
          ----
          1001
```

- Example: $\frac{10011011}{1001} = \frac{-101}{-7}$
  - ✓ Turn the numerator and denominator into positive numbers $\rightarrow \frac{01100101}{0111}$
  - ✓ Convert both numerator and denominator into unsigned numbers: $\frac{1100101}{111}$
  - ✓ $\frac{|A|}{|B|} \Rightarrow Q' = 1110$, $R' = 11$. These are unsigned numbers.
  - ✓ Get Q and R: $A < 0, B < 0 \rightarrow Q = Q' = 01110 = 14$, $R = 2C(R') = 2C(011) = 101 = -3$.
  - ✓ Verification: $-101 = -7 \times 14 + (-3)$.

```
      0001110
111 ) 1100101
      111
      ---
      1011
       111
       ---
       1000
        111
        ---
         11
```

Basic Arithmetic Units for Integer Numbers

- Boolean Algebra is a very powerful tool for the implementation of digital circuits. Here, we map Boolean Algebra expressions into binary arithmetic expressions for the implementation of binary arithmetic units. Note the operators '+', '.' in Boolean Algebra are not the same as addition/subtraction, and multiplication in binary arithmetic.

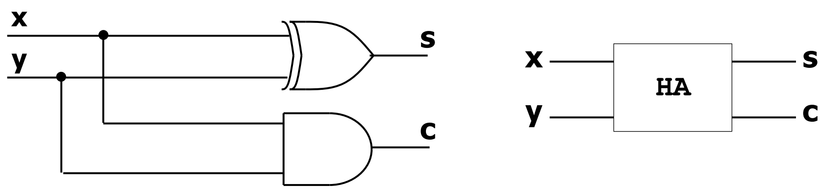
Addition/Subtraction

UNSIGNED NUMBERS

- **1-bit Addition**:
  - ✓ Addition of a bit with carry in: The circuit that performs this operation is called Half Adder (HA).

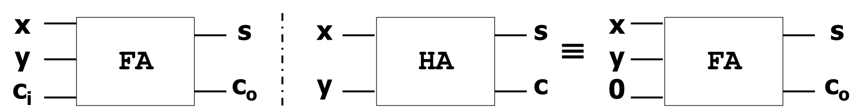
		0 +	0 +	1 +	1 +
	x +	0	1	0	1
	y				
carry out ←	c s → sum	0 0	0 1	0 1	1 0

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



  - ✓ Addition of a bit with carry in: The circuit that performs this operation is called Full Adder (FA).

c_i
x +
carry out y sum
c_o s



- ***n*-bit Addition**:
  The figure on the right shows a 5-bit addition. Using the truth table method, we would need 11 inputs and 6 outputs. This is not practical! Instead, it is better to build a cascade of Full Adders.

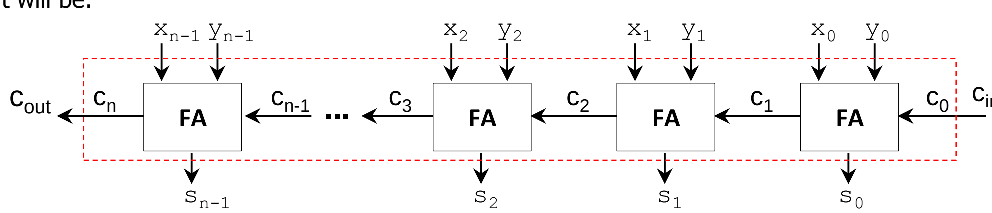
  $c_5=0$ $c_4=1$ $c_3=1$ $c_2=1$ $c_1=0$ $c_0=0$

  15: 0 1 1 1 1 +
  10: 0 1 0 1 0
  25: 1 1 0 0 1

  C_{out} C_{in}
  $x_4x_3x_2x_1x_0$ +
  $y_4y_3y_2y_1y_0$
  $s_4s_3s_2s_1s_0$

  For an n-bit addition, the circuit will be:

  C_{out} C_{in}
  $x_{n-1}x_{n-2}...x_1x_0$ +
  $y_{n-1}y_{n-2}...y_1y_0$
  $s_{n-1}s_{n-2}...s_1s_0$



Full Adder Design

x_i	y_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

c_i \ x_iy_i	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$s_i = \overline{x_i}y_i\overline{c_i} + x_i\overline{y_i}\overline{c_i} + \overline{x_i}\overline{y_i}c_i + x_iy_ic_i$$

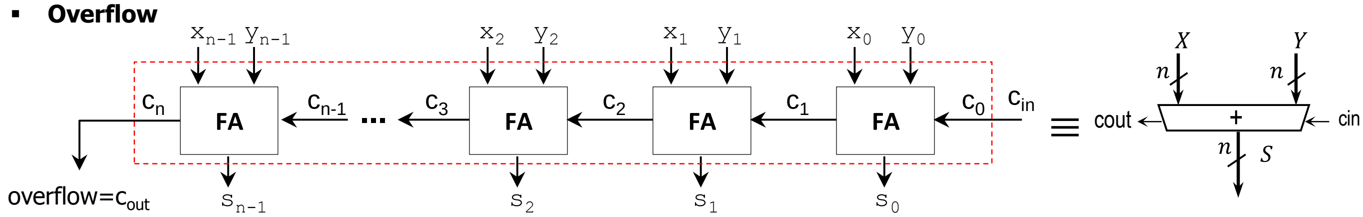
$$s_i = (x_i \oplus y_i)\overline{c_i} + \overline{(x_i \oplus y_i)}c_i$$

$$s_i = x_i \oplus y_i \oplus c_i$$

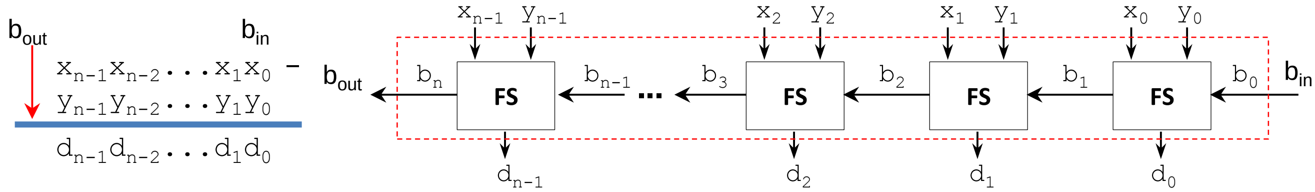
c_i \ x_iy_i	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

- **Overflow**

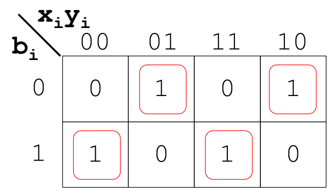


- **n-bit Subtractor**:
We can build an n-bit subtractor for unsigned numbers using Full Subtractor circuits. In practice, subtraction is better performed in the 2's complement representation (this accounts for signed numbers).



Full Subtractor Design

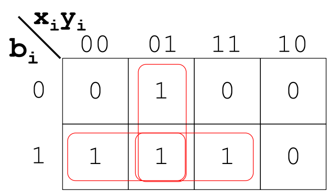
x_i	y_i	b_i	b_{i+1}	d_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



$$d_i = \overline{x_i}\,y_i\,\overline{b_i} + x_i\,\overline{y_i}\,\overline{b_i} + \overline{x_i}\,\overline{y_i}\,b_i + x_i y_i b_i$$

$$d_i = (x_i \oplus y_i)\,\overline{b_i} + \overline{(x_i \oplus y_i)}\,b_i$$

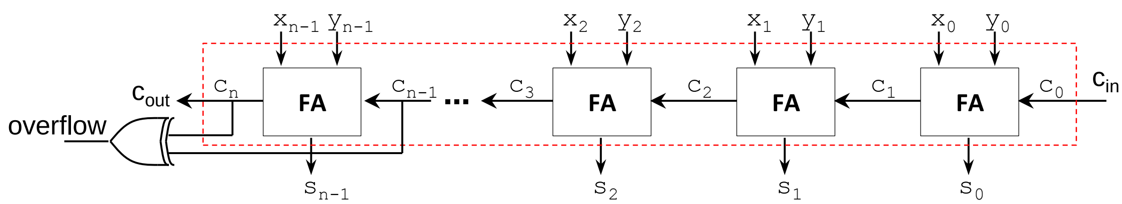
$$d_i = x_i \oplus y_i \oplus b_i$$



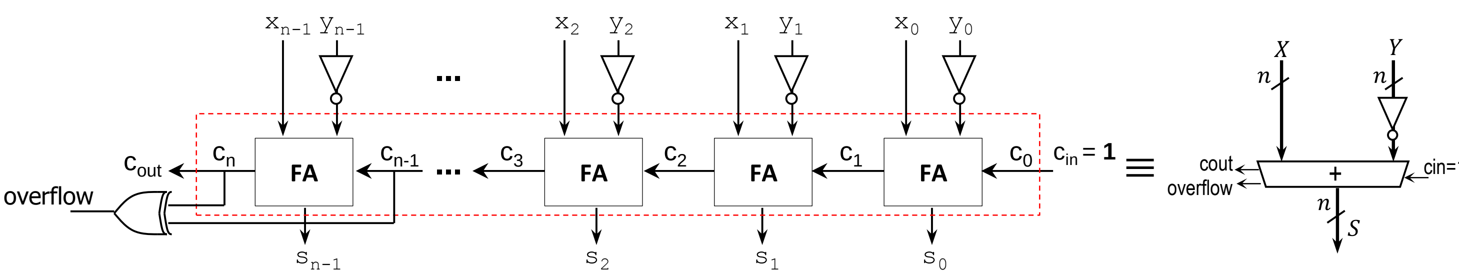
$$b_{i+1} = \overline{x_i}\,y_i + \overline{x_i}\,b_i + y_i b_i$$

SIGNED NUMBERS

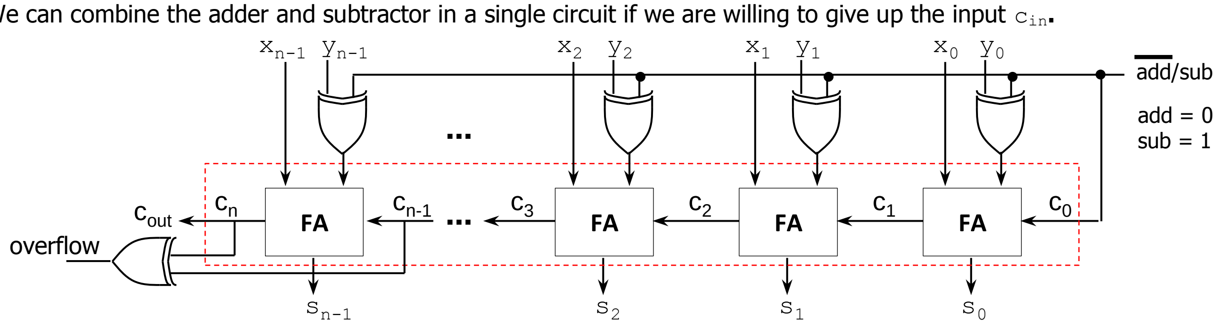
- The figure depicts an n-bit adder for 2's complement numbers:



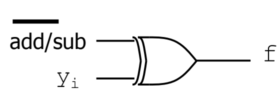
- **Subtraction**: $A - B = A + 2C(B)$. In 2C arithmetic, subtraction is actually an addition of two numbers.
The digital circuit for 2C subtraction is based on the adder. We account for the 2's complement operation for the subtrahend by inverting every bit in the subtrahend and by making the c_{in} bit equal to 1.



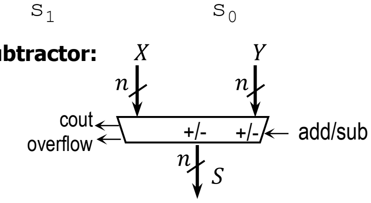
- **Adder/Subtractor Unit for 2's complement numbers:**
We can combine the adder and subtractor in a single circuit if we are willing to give up the input c_{in}.



$\overline{\text{add}}/\text{sub}$	y_i	f
0	0	0
0	1	1
1	0	1
1	1	0



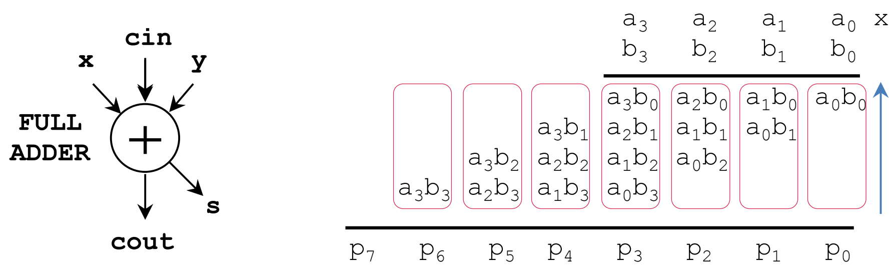
Adder/Subtractor:

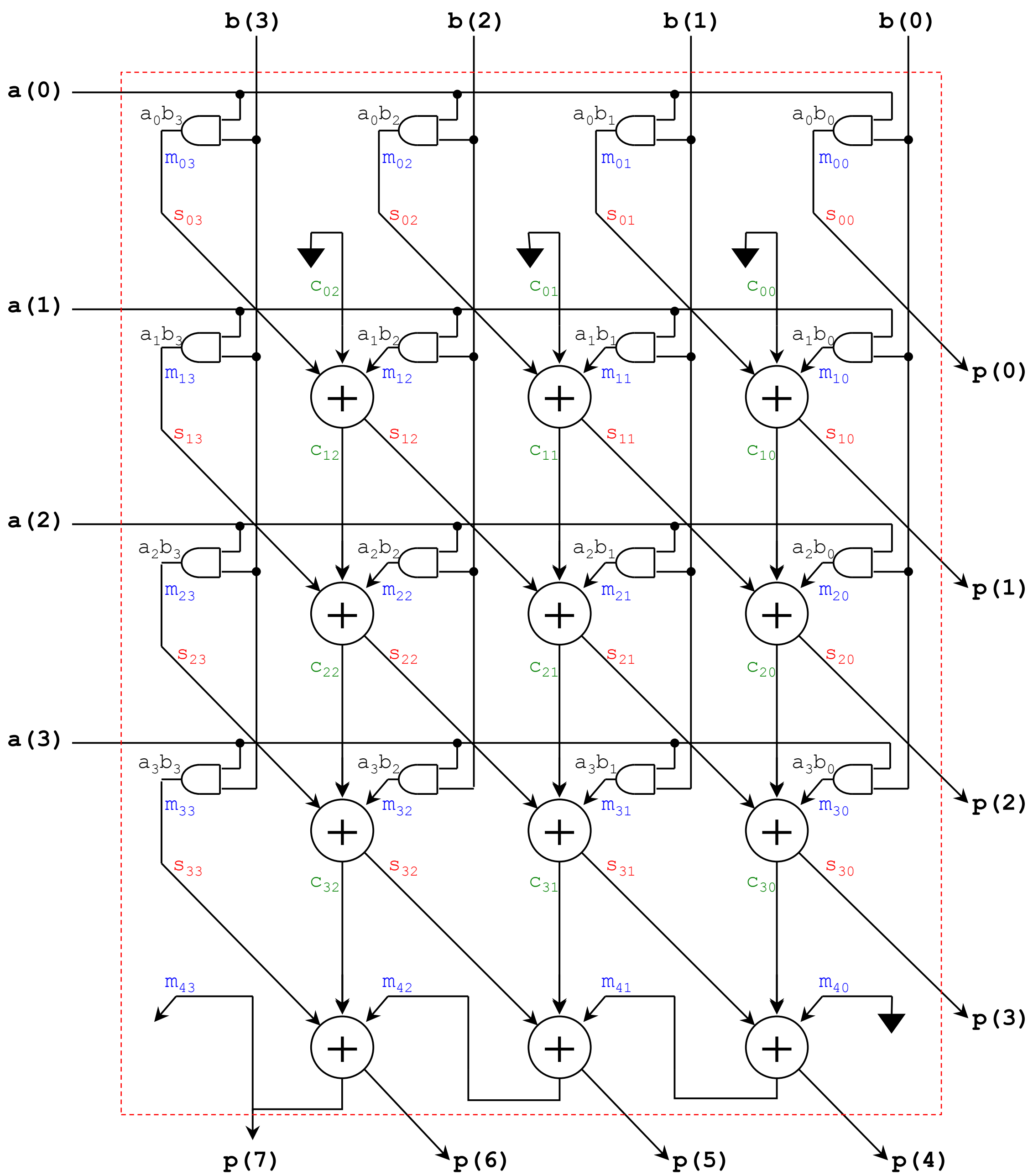


MULTIPLICATION

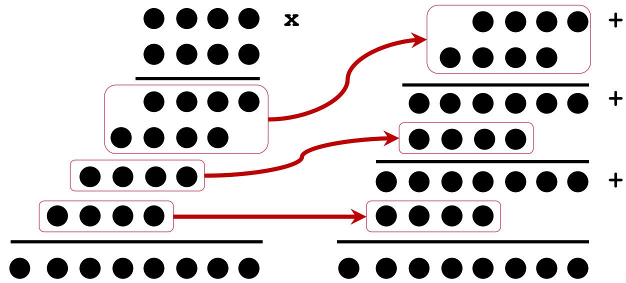
UNSIGNED NUMBERS

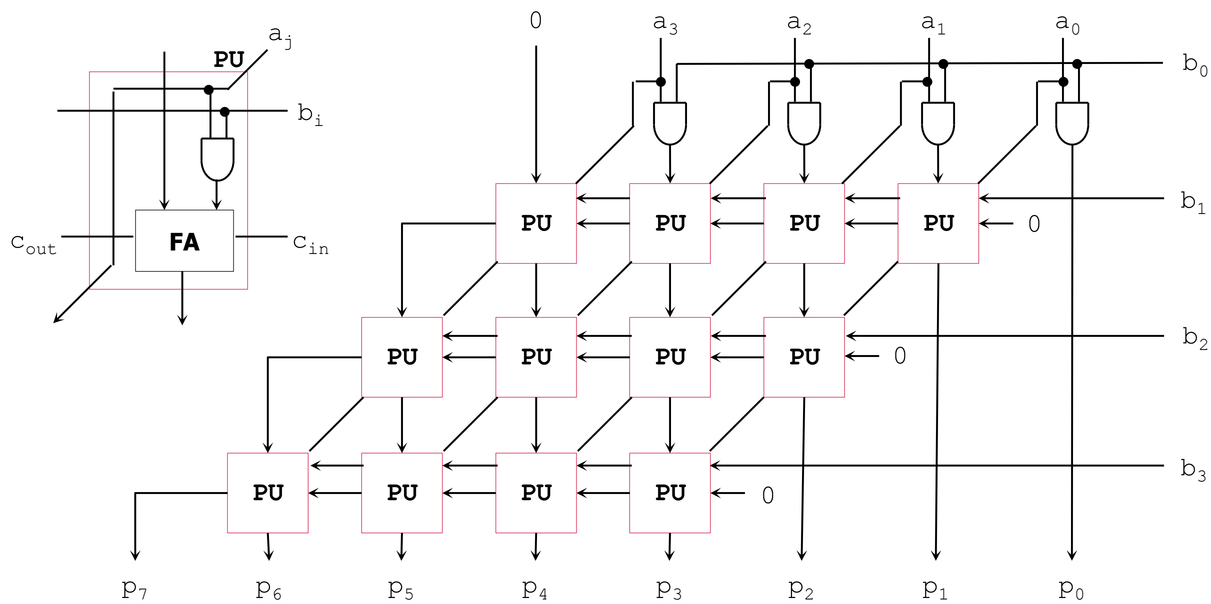
- The figure shows the process for multiplying two unsigned numbers of 4 bits.
- A straightforward implementation of the multiplication operation is also depicted in the figure below: at every diagonal of the circuit, we add up all terms in a column of the multiplication.





- An alternative implementation of the multiplication operation is depicted below for 4-bit unsigned numbers. It is much simpler to see how only two rows are added up at each stage.

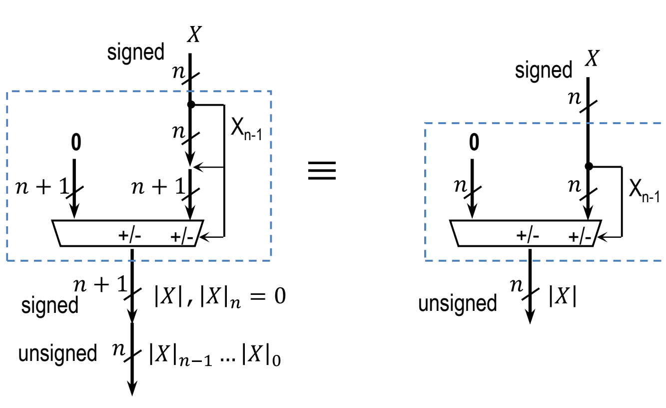


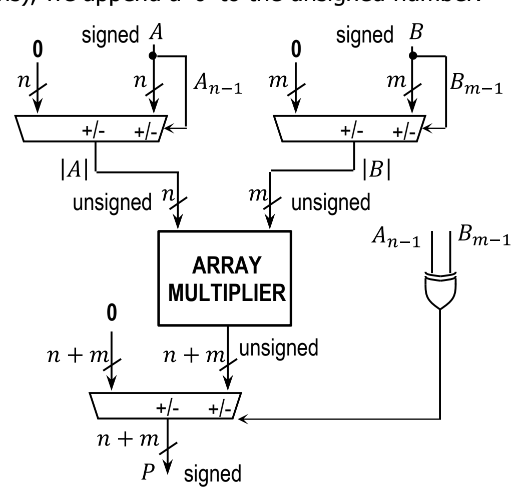


SIGNED NUMBERS (2C)

- This signed multiplier uses an unsigned multiplier, three adder subtractors (with one constant input), and a logic gate.
  - ✓ The initial adder/subtractor units provide the absolute values of A and B.
  - ✓ The largest unsigned product is given by 2^{n+m-2} ($n+m-1$ bits suffice to represent this number), so the $(n+m)$-bit unsigned product has its MSB=0. Thus, we can use this $(n+m)$-bit unsigned number as a positive signed number. The final adder/subtractor might change the sign of the positive product based on the signs of A and B.
- **Absolute Value**: For an n-bit signed number X, the absolute value is defined as: $|X| = \begin{cases} 0+X, X \geq 0 \\ 0-X, X < 0 \end{cases}$
  - ✓ Thus, the absolute value $|X|$ can have at most $n+1$ bits. To avoid overflow, we sign-extend the inputs to $n+1$ bits. The result $|X|$ has $n+1$ bits. Since $|X|$ is an absolute value, then $|X|_N = 0$. Thus, we can get $|X|$ as an unsigned number by discarding the MSB, i.e., using only n bits: $|X|_{n-1}$ downto $|X|_0$.
  - ✓ Alternatively, we can omit the sign-extension (since we are discarding $|X|_n$ anyway), and we will get $|X|$ as an unsigned number. If we need $|X|$ as a signed number (for further computations), we append a '0' to the unsigned number.

Absolute Value Circuit

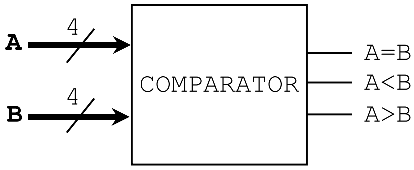


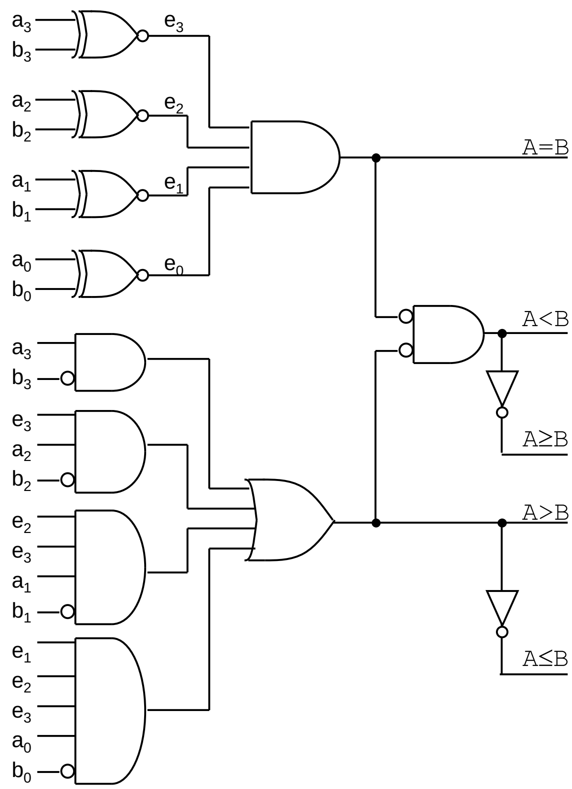


COMPARATORS

UNSIGNED NUMBERS

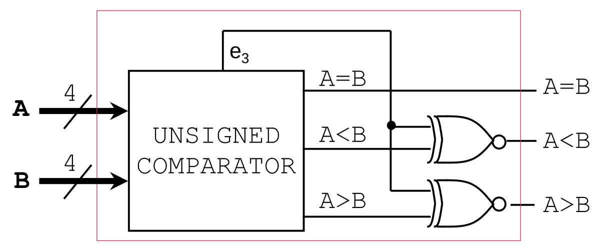
- For $A = a_3a_2a_1a_0$, $B = b_3b_2b_1b_0$
  - ✓ $A > B$ when:
    $a_3 = 1, b_3 = 0$
    Or: $a_3 = b_3$ and $a_2 = 1, b_2 = 0$
    Or: $a_3 = b_3, a_2 = b_2$ and $a_1 = 1, b_1 = 0$
    Or: $a_3 = b_3, a_2 = b_2, a_1 = b_1$ and $a_0 = 1, b_0 = 0$





SIGNED NUMBERS

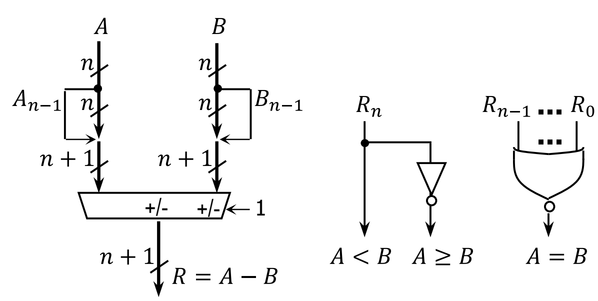
- **First Approach:**
  - ✓ If $A \geq 0$ and $B \geq 0$, we can use the unsigned comparator.
  - ✓ If $A < 0$ and $B < 0$, we can also use the unsigned comparator. Example: $1000_2 < 1001_2$ (-8 < -7). The closer the number is to zero, the larger the unsigned value is.
  - ✓ If one number is positive and the other negative: Example: $1000_2 < 0100_2$ (-8 < 4). If we were to use the unsigned comparator, we would get $1000_2 > 0100_2$. So, in this case, we need to invert both the A>B and the A<B bit.



  - ✓ Example: For a 4-bit number in 2's complement:
    - If $a_3 = b_3$, A and B have the same sign. Then, we do not need to invert any bit.
    - If $a_3 \neq b_3$, A and B have a different sign. Then, we need to invert the A>B and A<B bits of the unsigned comparator.

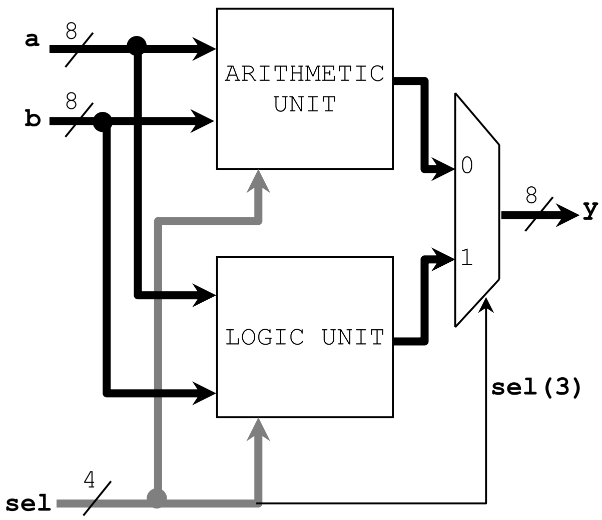
    $e_3 = 1$ when $a_3 = b_3$. $e_3 = 0$ when $a_3 \neq b_3$.
    Then it follows that:
    $$(A < B)_{signed} = \bar{e_3} \oplus (A < B)_{unsigned} = \overline{e_3 \oplus (A < B)_{unsigned}}$$
    $$(A > B)_{signed} = \overline{e_3 \oplus (A > B)_{unsigned}}$$

- **Second Approach:**
  - ✓ Here, we use an adder/subtractor in 2C arithmetic. We need to sign-extend the inputs to consider the worst-case scenario and then subtract them.
  - ✓ We can determine whether A is greater than B, based on:
    $$R_n = \begin{cases} 1 \to A - B < 0 \\ 0 \to A - B \geq 0 \end{cases}$$
  - ✓ To determine whether $A = B$, we compare the $n+1$ bits of R to 0 ($R = A - B$). However, note that $(A - B) \in [-2^n + 1, 2^n - 2]$. So, the case $R = -2^n = 10\ldots0$ will not occur. Thus, we only need to compare the bits R_{n-1} to R_0 to 0.



ARITHMETIC LOGIC UNIT (ALU)

- Two types of operation: Arithmetic and Logic (bit-wise). The `sel(3..0)` input selects the operation. `sel(2..0)` selects the operation type within a specific unit. The arithmetic unit consist of adders and subtractors, while the Logic Unit consist of 8-input logic gates.

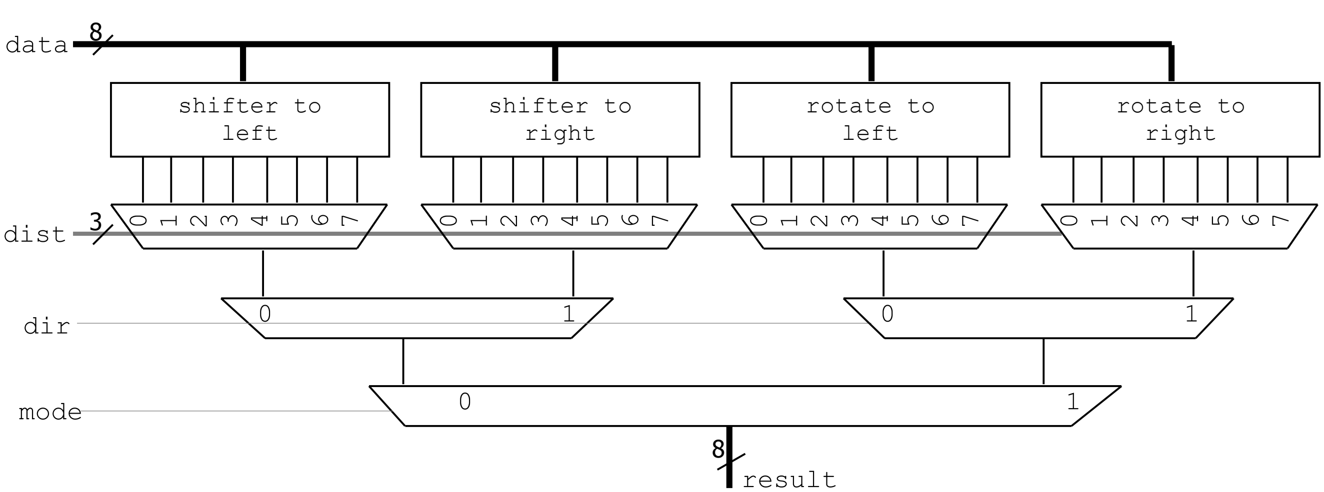


sel	Operation	Function	Unit
0 0 0 0	y <= a	Transfer 'a'	ARITHMETIC
0 0 0 1	y <= a + 1	Increment 'a'	
0 0 1 0	y <= a - 1	Decrement 'a'	
0 0 1 1	y <= b	Transfer 'b'	
0 1 0 0	y <= b + 1	Increment 'b'	
0 1 0 1	y <= b - 1	Decrement 'b'	
0 1 1 0	y <= a + b	Add 'a' and 'b'	
0 1 1 1	y <= a - b	Subtract 'b' from 'a'	
1 0 0 0	y <= NOT a	Complement 'a'	LOGIC
1 0 0 1	y <= NOT b	Complement 'b'	
1 0 1 0	y <= a AND b	AND	
1 0 1 1	y <= a OR b	OR	
1 1 0 0	y <= a NAND b	NAND	
1 1 0 1	y <= a NOR b	NOR	
1 1 1 0	y <= a XOR b	XOR	
1 1 1 1	y <= a XNOR b	XNOR	

BARREL SHIFTER

- Two types of operation: Arithmetic (mode=0, it implements 2^i) and Rotation (mode=1)
- Truth table for a 8-bit Barrel Shifter:
  `result[7..0]` (output): It is shifted version of the input `data[7..0]`. `sel[2..0]`: number of bits to shift. `dir`: It controls the shifting direction (`dir=1`: to the right, `dir=0`: to the left). When shifting to the right in the Arithmetic Mode, we use sign extension so as properly account for both unsigned and signed input numbers.

mode = 0. ARITHMETIC MODE				mode = 1. ROTATION MODE			
dir	**dist[2..0]**	**data[7..0]**	**result[7..0]**	**dir**	**dist[2..0]**	**data[7..0]**	**result[7..0]**
X	0 0 0	abcdefgh	abcdefgh	X	0 0 0	abcdefgh	abcdefgh
0	0 0 1	abcdefgh	bcdefgh0	0	0 0 1	abcdefgh	bcdefgha
0	0 1 0	abcdefgh	cdefgh00	0	0 1 0	abcdefgh	cdefghab
0	0 1 1	abcdefgh	defgh000	0	0 1 1	abcdefgh	defghabc
0	1 0 0	abcdefgh	efgh0000	0	1 0 0	abcdefgh	efghabcd
0	1 0 1	abcdefgh	fgh00000	0	1 0 1	abcdefgh	fghabcde
0	1 1 0	abcdefgh	gh000000	0	1 1 0	abcdefgh	ghabcdef
0	1 1 1	abcdefgh	h0000000	0	1 1 1	abcdefgh	habcdefg
1	0 0 1	abcdefgh	aabcdefg	1	0 0 1	abcdefgh	habcdefg
1	0 1 0	abcdefgh	aaabcdef	1	0 1 0	abcdefgh	ghabcdef
1	0 1 1	abcdefgh	aaaabcde	1	0 1 1	abcdefgh	fghabcde
1	1 0 0	abcdefgh	aaaaabcd	1	1 0 0	abcdefgh	efghabcd
1	1 0 1	abcdefgh	aaaaaabc	1	1 0 1	abcdefgh	defghabc
1	1 1 0	abcdefgh	aaaaaaab	1	1 1 0	abcdefgh	cdefghab
1	1 1 1	abcdefgh	aaaaaaaa	1	1 1 1	abcdefgh	bcdefgha



FIXED-POINT (FX) ARITHMETIC

INTRODUCTION

FX FOR UNSIGNED NUMBERS

- We know how to represent positive integer numbers. But what if we wanted to represent numbers with fractional parts?
- Fixed-point arithmetic: Binary representation of positive decimal numbers with fractional parts.

  FX number (in binary representation): $(b_{n-1}b_{n-2}\dots b_1b_0.\boldsymbol{b_{-1}b_{-2}\dots b_{-k}})_2$

- **Conversion from binary to decimal:**

$$D=\sum_{i=-k}^{n-1} b_i\times 2^i = b_{n-1}\times 2^{n-1}+b_{n-2}\times 2^{n-2}+\dots+b_1\times 2^1+b_0\times 2^0+\boldsymbol{b_{-1}\times 2^{-1}}+\boldsymbol{b_{-2}\times 2^{-2}}+\cdots\boldsymbol{b_{-k}\times 2^{-k}}$$

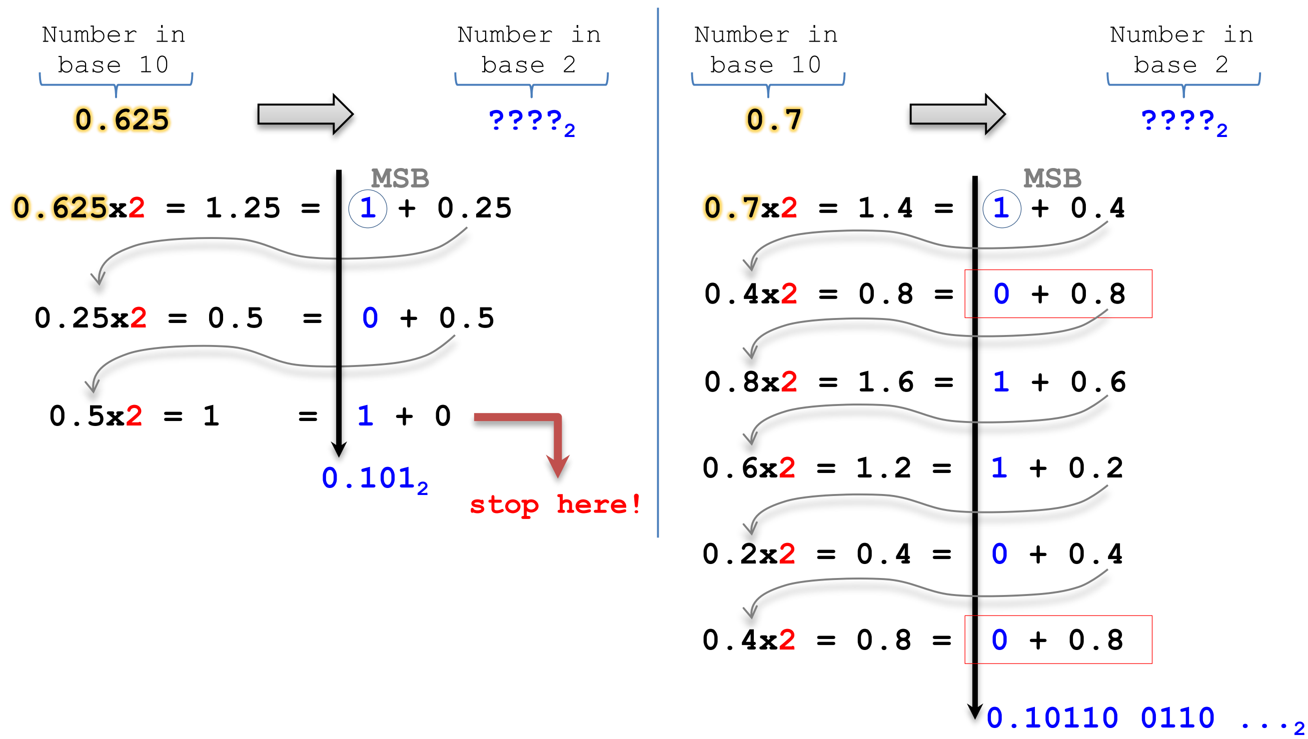
  Example: $1011.101_2 = 1\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0+1\times\boldsymbol{2^{-1}}+\boldsymbol{0}\times\boldsymbol{2^{-2}}+\boldsymbol{1}\times\boldsymbol{2^{-3}}=11.625$

  To convert from binary to hexadecimal:

  Binary: 10101.10101_2 → $\underbrace{0001}_{1}\ \underbrace{0101}_{5}.\underbrace{1010}_{A}\ \underbrace{1000}_{8}$

  hexadecimal: 1 5 . A 8

- **Conversion from decimal to binary:** We divide the number into its integer and fractional parts. We get the binary representation of the integer part using the successive divisions by 2. For the fractional part, we apply successive multiplications by 2 (see example below). We then combine the integer and fractional binary results.
  - ✓ **Example:** Convert 31.625 to FX (in binary): We know $31 = 11111_2$. In the figure below, we have that $0.625 = 0.101_2$. Thus: $31.625 = 11111.101_2$.



FX FOR SIGNED NUMBERS

- Method: Get the FX representation of $+379.21875$, and then apply the 2's complement operation to that result.
- **Example:** Convert -379.21875 to the 2's complement representation.
  - ✓ 379 = 101111011_2. 0.21875 = 0.00111_2. Then: +379.21875 (2C) = 0101111011.00111_2.
  - ✓ We get -379.2185 by applying the 2C operation to +379.21875 ⇒ -379.21875 = 1010000100.11001_2 = 0xE84.C8. To convert to hexadecimal, we append zeros to the LSB and sign-extend the MSB. Note that the 2C operation involves inverting the bits and add 1; the addition by '1' applies to the LSB, not to the rightmost integer.

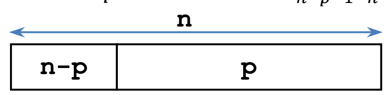
INTEGER REPRESENTATION

- $n-bit$ number: $b_{n-1}b_{n-2}\dots b_0$

	UNSIGNED	SIGNED
Decimal Value	$D=\sum_{i=0}^{n-1} b_i 2^i$	$D=-2^{n-1}b_{n-1}+\sum_{i=0}^{n-2} b_i 2^i$
Range of values	$[0, 2^n-1]$	$[-2^{n-1}, 2^{n-1}-1]$

FIXED POINT REPRESENTATION

- Typical representation $[n\ p]$: $n-bit$ number with p fractional bits: $b_{n-p-1}b_{n-p-2}\dots b_0.b_{-1}b_{-2}\dots b_{-p}$



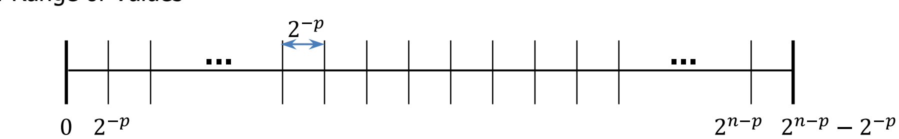
	UNSIGNED	SIGNED
Decimal Value	$D=\sum_{i=-p}^{n-p-1} b_i 2^i$	$D=-2^{n-p-1}b_{n-p-1}+\sum_{i=-p}^{n-p-2} b_i 2^i$
Range of values	$\left[\frac{0}{2^p}, \frac{2^n-1}{2^p}\right]=[0, 2^{n-p}-2^{-p}]$	$\left[\frac{-2^{n-1}}{2^p}, \frac{2^{n-1}-1}{2^p}\right]=[-2^{n-p-1}, 2^{n-p-1}-2^{-p}]$
Dynamic Range	$\frac{\lvert 2^{n-p}-2^{-p}\rvert}{\lvert 2^{-p}\rvert}=2^n-1$ $(dB)=20\times\log_{10}(2^n-1)$	$\frac{\lvert -2^{n-p-1}\rvert}{\lvert 2^{-p}\rvert}=2^{n-1}$ $(dB)=20\times\log_{10}(2^{n-1})$
Resolution (1 LSB)	2^{-p}	2^{-p}

- Dynamic Range:

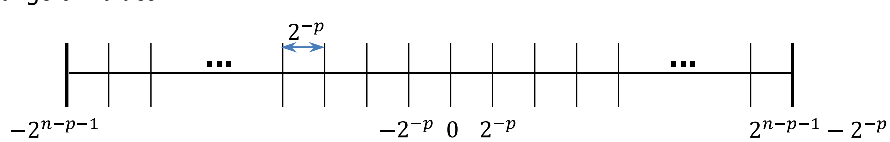
$$Dynamic\ Range=\frac{largest\ abs.value}{smallest\ nonzero\ abs.value}$$

$$Dynamic\ Range(dB)=20\times\log_{10}(Dynamic\ Range)$$

- Unsigned numbers: Range of Values



- Signed numbers: Range of Values

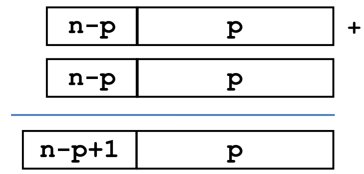


- Examples:

	FX Format	Range	Dynamic Range (dB)	Resolution
UNSIGNED	[8 7]	[0, 1.9922]	48.13	0.0078
	[12 8]	[0, 15.9961]	72.24	0.0039
	[16 10]	[0, 63.9990]	96.33	0.0010
SIGNED	[8 7]	[-1, 0.9921875]	42.14	0.0078
	[12 8]	[-8, 7.99609375]	66.23	0.0039
	[16 10]	[-64, 63.9990234375]	90.31	0.0010

FIXED-POINT ADDITION/SUBTRACTION

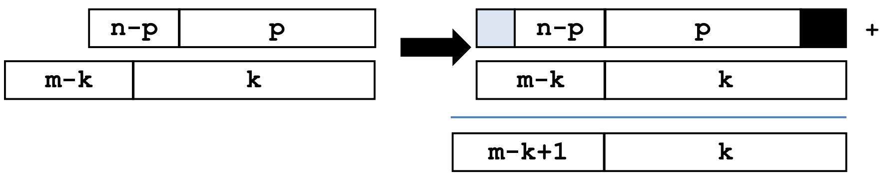
- Addition of two numbers represented in the format $[n\ \ p]$:



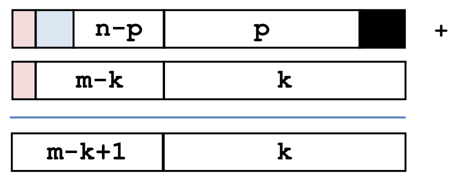
$A \times 2^{-p} \pm B \times 2^{-p} = (A \pm B) \times 2^{-p}$

We perform integer addition/subtraction of A and B. We just need to interpret the result differently by placing the fractional point where it belongs. Notice that the hardware is the same as that of integer addition/subtraction.

When adding/subtracting numbers with different formats $[n\ \ p]$ and $[m\ \ k]$, we first need to align the fractional point so that we use a format for both numbers: it could be $[n\ \ p]$, $[m\ \ k]$, $[n-p+k\ \ k]$, $[m-k+p\ \ p]$. This is done by zero-padding and sign-extending where necessary. In the figure below, the format selected for both numbers is $[m\ \ k]$, while the result is in the format $[m+1\ \ k]$.



Important: The result of the addition/subtraction requires an extra bit in the worst-case scenario. In order to correctly compute it in fixed-point arithmetic, we need to sign-extend (by one bit) the operators prior to addition/subtraction.



Multi-operand Addition: N numbers of format $[n\ \ p]$: The total number of bits is given by : $n + \lceil \log_2 N \rceil$ (this can be demonstrated by an adder tree). Notice that the number of fractional bits does not change (it remains p), only the integer bits increase by $\lceil \log_2 N \rceil$, i.e., the number of integer bits become $n - p + \lceil \log_2 N \rceil$.

- **Examples**: Calculate the result of the additions and subtractions for the following fixed-point numbers.

UNSIGNED		SIGNED	
0.101010 + 1.0110101	1.00101 - 0.0000111	10.001 + 1.001101	0.0101 - 1.0101101
10.1101 + 1.1001	100.1 + 0.1000101	1000.0101 - 111.01001	101.0001 + 1.0111101

Unsigned:

```
c8=1 c7=1 c6=1 c5=1 c4=0 c3=1 c2=0 c1=0 c0=0
  0.1 0 1 0 1 0 0 +
  1.0 1 1 0 1 0 1
-----------------
1 0.0 0 0 1 0 0 1
```

```
b7=0 b6=0 b5=0 b4=1 b3=1 b2=1 b1=1 b0=0
1.0 0 1 0 1 0 0 -
0.0 0 0 0 1 1 1
---------------
1.0 0 0 1 1 0 1
```

```
c6=1 c5=1 c4=1 c3=0 c2=0 c1=1 c0=0
  1 0.1 1 0 1 +
    1.1 0 0 1
-------------
1 0 0.0 1 1 0
```

```
c10=0 c9=0 c8=0 c7=1 c6=0 c5=0 c4=0 c3=0 c2=0 c1=0 c0=0
1 0 0.1 0 0 0 0 0 0 +
    0.1 0 0 0 1 0 1
-------------------
1 0 1.0 0 0 0 1 0 1
```

Signed:

```
c9=1 c8=1 c7=0 c6=0 c5=0 c4=1 c3=0 c2=0 c1=0 c0=0
1 1 0.0 0 1 0 0 0 +
1 1 1.0 0 1 1 0 1
-----------------
1 0 1.0 1 0 1 0 1
```

```
0.0 1 0 1 0 0 0 -
1.0 1 0 1 1 0 1
```

```
c8=0 c7=0 c6=0 c5=0 c4=0 c3=0 c2=0 c1=0 c0=0
0.0 1 0 1 0 0 0 +
0.1 0 1 0 0 1 1
---------------
0.1 1 1 1 0 1 1
```

```
1 0 0 0.0 1 0 1 0 -
1 1 1 1.0 1 0 0 1
```

```
c9=0 c8=0 c7=0 c6=0 c5=1 c4=1 c3=1 c2=1 c1=0 c0=0
1 0 0 0.0 1 0 1 0 +
0 0 0 0.1 0 1 1 1
-----------------
1 0 0 1.0 0 0 0 1
```

```
c10=1 c9=1 c8=1 c7=0 c6=1 c5=1 c4=1 c3=0 c2=0 c1=0 c0=0
1 0 1.0 0 0 1 0 0 0 +
1 1 1.0 1 1 1 1 0 1
-------------------
1 0 0.1 0 0 0 1 0 1
```

FIXED-POINT MULTIPLICATION

- **Unsigned multiplication**

  Multiplication of two signed numbers represented with different formats $[n\ \ p], [m\ k]$:

```
          | n-p |    p    |  x
      | m-k |      k      |
 -----------------------------
 | n+m-p-k |       p+k      |
```

$(A \times 2^{-p}) \times (B \times 2^{-k}) = (A \times B) \times 2^{-p-k}$. We can perform integer multiplication of A and B and then place the fractional point where it belongs. The format of the multiplication result is given by $[n+m\ \ p+k]$. There is no need to align the fractional point of the input quantities.

Special case: $m = n, k = p$

$(A \times 2^{-p}) \times (B \times 2^{-p}) = (A \times B) \times 2^{-2p}$. Here, the format of the multiplication result is given by $[2n\ \ 2p]$.

✓ Multiplication procedure for unsigned integer numbers:

```
                      a3    a2    a1    a0   x
                      b3    b2    b1    b0
                    -------------------------
                    a3b0  a2b0  a1b0  a0b0
              a3b1  a2b1  a1b1  a0b1
        a3b2  a2b2  a1b2  a0b2
  a3b3  a2b3  a1b3  a0b3
------------------------------------------------
  p7    p6    p5    p4    p3    p2    p1    p0
```

Example: when multiplying, we treat the numbers as integers. Only when we get the result, we place the fractional point where it belongs.

```
2.75 = 10.11 x          1 0 1 1 x
 6.5 = 110.1            1 1 0 1
                        -------
                        1 0 1 1
                      0 0 0 0
                    1 0 1 1
                  1 0 1 1
                ---------------
                1 0 0 0 1 1 1 1

       17.875 = 1 0 0 0 1.1 1 1
```

- **Signed Multiplication:** We first take the absolute value of the operands. Then, if at least one of the operands was negative, we need to change the sign of the result. We then place the fractional point where it belongs.

Examples:

```
01.001 x       01.001 x          1 1 0 1 1 1 x
1.001001       0.110111              1 0 0 1
                                 -----------
                                 1 1 0 1 1 1
                               0 0 0 0 0 0
                             0 0 0 0 0 0
                           1 1 0 1 1 1
                         -------------------
                           1 1 1 1 0 1 1 1 1

                         0.1 1 1 1 0 1 1 1 1

                         1.0 0 0 0 1 0 0 0 1
```

```
10.0001 x      01.1111 x          1 0 1 0 0 1 x
01.01001       01.01001             1 1 1 1 1
                                  -----------
                                  1 0 1 0 0 1
                                1 0 1 0 0 1
                              1 0 1 0 0 1
                            1 0 1 0 0 1
                          1 0 1 0 0 1
                        -----------------------
                        1 0 0 1 1 1 1 0 1 1 1

                        0 1 0.0 1 1 1 1 0 1 1 1

                        1 0 1.1 0 0 0 0 1 0 0 1
```

```
1000.000 x     01000.000 x         1 1 0 1 0 1 1 x
10.010101      01.101011           1 0 0 0 0 0 0
                                   -------------
                                   0 0 0 0 0 0 0
                               1 1 0 1 0 1 1
                     ---------------------------
                     1 1 0 1 0 1 1 0 0 0 0 0 0 0

                 0 1 1 0 1.0 1 1 0 0 0 0 0 0 0
```

```
0.1101010 x     0.110101 x          1 1 0 1 0 1 x
11.1111011      0.0000101                 1 0 1
                                    -----------
                                    1 1 0 1 0 1
                                  0 0 0 0 0 0
                                1 1 0 1 0 1
                                -------------------
                                1 0 0 0 0 1 0 0 1

                0.0 0 0 0 1 0 0 0 0 1 0 0 1

                1.1 1 1 1 0 1 1 1 1 0 1 1 1
```

FIXED-POINT DIVISION

- **Unsigned Division**: A_f/B_f

  We first need to align the numbers so they have the same number of fractional bits, then divide them treating them as integers. The quotient will be integer, while the remainder will have the same number of fractional bits as A_f.

  A_f is in the format $[na\ a]$. B_f is in the format $[nb\ b]$

  Step 1: For $a \geq b$, we align the fractional points and then get the integer numbers A and B, which result from:

  $$A = A_f \times 2^a \quad B = B_f \times 2^a$$

  Step 2: Integer division: $\frac{A}{B} = \frac{A_f}{B_f}$

  The numbers A and B are related by the formula: $A = B \times Q + R$, where Q and R are the quotient and remainder of the integer division of A and B. Note that Q is also the quotient of $\frac{A_f}{B_f}$.

  Step 3: To get the correct remainder of $\frac{A_f}{B_f}$, we re-write the previous equation:

  $$A_f \times 2^a = (B_f \times 2^a) \times Q + R \rightarrow A_f = B_f \times Q + (R \times 2^{-a})$$

  Then: $Q_f = Q,\ R_f = R \times 2^{-a}$

  Example:

  $$\frac{1010.011}{11.1}$$

  Step 1: Alignment, $a = 3$

  $$\frac{1010.011}{11.1} = \frac{1010.011}{11.100} = \frac{1010011}{11100}$$

  Step 2: Integer Division

  $$\frac{1010011}{11100} \Rightarrow 1010011 = 11100(10) + 11011 \rightarrow\ Q = 10, R = 11011$$

  Step 3: Get actual remainder: $R \times 2^{-a}$

  $$R_f = 11.011$$

  Verification: $1010.011 = 11.1(10) + 11{,}011,\ Q_f = 10, R_f = 11011$

  ✓ **Adding precision bits to Q_f (quotient of A_f/B_f):**

  The previous procedure only gets Q as an integer. What if we want to get the division result with x number of fractional bits? To do so, after alignment, we append x zeros to $A_f \times 2^a$ and perform integer division.

  $$A = A_f \times 2^a \times 2^x \qquad B = B_f \times 2^a$$

  $$A_f \times 2^{a+x} = (B_f \times 2^a) \times Q + R \rightarrow A_f = B_f \times (Q \times 2^{-x}) + (R \times 2^{-a-x})$$

  Then: $Q_f = Q \times 2^{-x},\ R_f = R \times 2^{-a-x}$

  Example: $\frac{1010{,}011}{11{,}1}$ with $x = 2$ bits of precision

  Step 1: Alignment, $a = 3$

  $$\frac{1010.011}{11.1} = \frac{1010.011}{11.100} = \frac{1010011}{11100}$$

  Step 2: Append $x = 2$ zeros

  $$\frac{1010011}{11100} = \frac{10100011\mathbf{00}}{11100}$$

  Step 3: Integer Division

  $$\frac{10100011\mathbf{00}}{11100} \Rightarrow 10100011\mathbf{00} = 11100(1011) + 11000$$

  $$Q = 1011, R = 11000$$

  Step 4: Get actual remainder and quotient (or result): $Q_f = Q \times 2^{-x}, R_f = R \times 2^{-a-x}$

  $$Q_f = 10.11, R_f = 0.11000$$

  Verification: $1010.01100 = 11.1(10.11) + 0{,}11000.$

- **Signed division**: In this case (just as in the multiplication), we first take the absolute value of the operators A and B. If only one of the operators is negative, the result of abs(A)/abs(B) requires a sign change.
  What about the remainder? You can also correct the sign of R_f (using the procedure specified in the case of signed integer numbers). However, once the quotient is obtained with fractional bits, getting R_f with the correct sign is not very useful.

- Example: We get the division result (with $x = 4$ fractional bits) for the following signed fixed-point numbers:

  ✓ $\frac{101.1001}{1.011}$: To positive (numerator and denominator), alignment, and then to unsigned: $a = 4$: $\frac{101.1001}{1.011} = \frac{010.0111}{0.1010} \equiv \frac{100111}{1010}$

```
        0000111110
1010 ) 1001110000
        1010
        10011
         1010
         10010
          1010
          10000
           1010
            1100
            1010
             100
```

Append $x = 4$ zeros: $\frac{100111\mathbf{0000}}{1010}$
Unsigned integer Division:

$Q = 111110, R = 100$
$\rightarrow Qf = 11.1110\ (x = 4)$

Final result (2C): $\frac{101.1001}{1.011} = 011.111$ (this is represented as a signed number)

  ✓ $\frac{11.011}{1.01011}$: To positive (numerator and denominator), alignment, and then to unsigned, $a = 5$: $\frac{00.101}{0.10101} = \frac{0.10100}{0.10101} \equiv \frac{10100}{10101}$

```
         000001111
10101 ) 101000000
         10101
         100110
          10101
          100010
           10101
            11010
            10101
              101
```

Append $x = 4$ zeros: $\frac{10100\mathbf{0000}}{10101}$
Unsigned integer Division:

$Q = 1111, R = 101$
$\rightarrow Qf = 0.1111(x = 4)$

Final result (2C): $\frac{11.011}{1.01011} = 0.1111$ (this is represented as a signed number)

  ✓ $\frac{10.0110}{01.01}$: To positive (numerator), alignment, and then to unsigned, $a = 4$: $\frac{01.1010}{01.01} = \frac{01.1010}{01.0100} \equiv \frac{11010}{10100}$

```
         000010100
10100 ) 110100000
        10100
          11000
          10100
           10000
```

Append $x = 4$ zeros: $\frac{11010\mathbf{0000}}{10100}$
Unsigned integer Division:

$Q = 10100, R = 10000$
$\rightarrow Qf = 1.0100(x = 4)$ ✶ Qf here is represented as an unsigned number

Final result (2C): $\frac{10.0110}{01.01} = 2C(01.01) = 10.11$

  ✓ $\frac{0.101010}{110.1001}$: To positive (denominator), alignment, and then to unsigned, $a = 5$: $\frac{0.10101}{001.0111} = \frac{0.10101}{001.01110} \equiv \frac{10101}{101110}$

```
          000000111
101110 ) 101010000
          101110
          1001100
           101110
            111100
            101110
              1110
```

Append $x = 4$ zeros: $\frac{10101\mathbf{0000}}{101110}$
Unsigned integer Division:

$Q = 111, R = 1110$
$\rightarrow Qf = 0.0111(x = 4)$

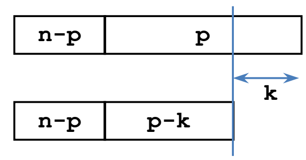
Final result (2C): $\frac{0.101010}{110.1001} = 2C(0.0111) = 1.1001$

ARITHMETIC FX UNITS. TRUNCATION/ROUNDING/SATURATION

ARITHMETIC FX UNITS

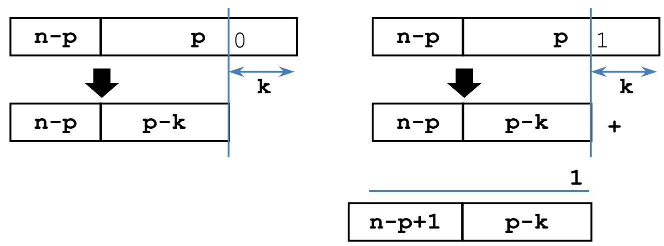
- They are the same as those that operate on integer numbers. The main difference is that we need to know where to place the fractional point. The design must keep track of the FX format at every point in the architecture.
- One benefit of FX representation is that we can perform truncation, rounding and saturation on the output results and the input values. These operations might require the use of some hardware resources.

TRUNCATION



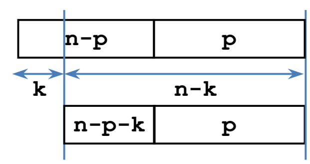
- This is a useful operation when less hardware is required in subsequent operations. However this comes at the expense of less accuracy.
- To assess the effect of truncation, use PSNR (dB) or MSE with respect to a double floating point result or with respect to the original $[n\ \ p]$ format.
- Truncation is usually meant to be truncation of the fractional part. However, we can also truncate the integer part (chop off k MSBs). This is not recommended as it might render the number unusable.

ROUNDING



- This operation allows for hardware savings in subsequent operations at the expense of reduced accuracy. But it is more accurate than simple truncation. However, it requires extra hardware to deal with the rounding operation.
- For the number $b_{n-p-1}b_{n-p-2}\dots b_0.b_{-1}b_{-2}\dots b_{-p}$, if we want to chop k bits (LSB portion), we use the b_{k-p-1} bit to determine whether to round. If the $b_{k-p-1}=0$, we just truncate. If $b_{k-p-1}=1$, we need to add '1' to the LSB of the truncated result.

SATURATION



- This is helpful when we need to restrict the number of integer bits. Here, we are asked to reduce the number of integer bits by k. Simple truncation chops off the integer part by k bits; this might completely modify the number and render it totally unusable. Instead, in saturation, we apply the following rules:

  ✓ If all the $k+1$ MSBs of the initial number are identical, that means that chopping by k bits does not change the number at all, so we just discard the k MSBs.

  ✓ If the $k+1$ MSBs are not identical, chopping by k bits does change the number. Thus, here, if the MSB of the initial number is 1, the resulting $(n-k)$-bit number will be $-2^{n-k-p-1}=10\dots0$ (largest negative number). If the MSB is 0, the resulting $(n-k)$-bit number will be $2^{n-k-p-1}-2^{-p}=011\dots1$ (largest positive number).

Examples: Represent the following signed FX numbers in the signed fixed-point format: $[8\ 7]$. You can use rounding or truncation for the fractional part. For the integer part, use saturation.

- $1{,}01101111$:
  To represent this number in the format $[8\ 7]$, we keep the integer bit, and we can only truncate or round the last LSB:
  After truncation: $1{,}0110111$
  After rounding: $1{,}0110111+1=1{,}0111000$

- $11{,}111010011$:
  Here, we need to get rid of on MSB and two LSBs. Let's use rounding (to the next bit).
  Saturation in this case amounts to truncation of the MSB, as the number won't change if we remove the MSB.
  After rounding: $11{,}1110100+1=11{,}1110101$
  After saturation: $1{,}1110101$

- $101{,}111010011$:
  Here, we need to get rid of two MSB and two LSBs.
  Saturation: Since the three MSBs are not the same and the MSB=1 we need to replace the number by the largest negative number (in absolute terms) in the format $[8\ 7]$: $1{,}0000000$

- $011{,}1111011011$:
  Here, we need to get rid of two MSB and three LSBs.
  Saturation: Since the three MSBs are not identical and the MSB=0, we need to replace the number by the largest positive number in the format $[8\ 7]$: $0{,}1111111$

FLOATING POINT REPRESENTATION

- There are many ways to represent floating numbers. A common way is:

$$X = \pm significand \times 2^e$$

±	e	significand
	E	m

- Exponent e: Signed integer. It is common to encode this field using a bias: $e + bias$. This facilitates zero detection ($e + bias = 0$). Note that the exponent of the actual number is always e regardless of the bias (the bias is just for encoding).

$$e \in [-2^{E-1}, 2^{E-1} - 1\,]$$

- Significand: Unsigned fixed point number. Usually normalized to a particular range, e.g.: $[0,1), [1,2)$.

m	
k	p

Format (unsigned): $[m\ \ p]$. Range: $\left[0, \frac{2^m-1}{2^p}\right] = [0, 2^{m-p} - 2^{-p}]$, $k = m - p$
If $k = 0 \rightarrow$ Significand $\in [0, 1 - 2^{-p}] = [0,1)$
If $k = m \rightarrow$ Significand $\in [0, 2^m - 1]$. Integer significand.

Another common representation of the significand is using $k = 1$ and setting that bit (the MSB) to 1. Here, the range of the significand would be $[0, 2^1 - 2^{-p}]$, but since the integer bit is 1, the values start from 1, which result in the following significand range: $[1, 2^1 - 2^{-p}]$. This is a popular normalization, as it allows us to drop the MSB in the encoding.

IEEE-754 STANDARD

- The representation is as follows:

$$X = \pm 1.f \times 2^e$$

sign bit	biased exponent	significand
±	e+bias	f
	E	p

- **Significand**: Unsigned FX integer. The representation is normalized to $s = 1.f$, where f is the mantissa. There is always an integer bit 1 (called hidden 1) in the representation of the significand, so we do not need to indicate in the encoding. Thus, we only use f the mantissa in the significant field.
Significand range: $[1, 2 - 2^{-p}] = [1,2)$

- **Biased exponent**: Unsigned integer with E bits. $bias = 2^{E-1} - 1$. Thus, $exp = e + bias \rightarrow e = exp - bias$. We just subtract the $bias$ from the exponent field in order to get the exponent value e.
  - ✓ $exp = e + bias \in [0, 2^E - 1\,]$. exp is represented as un unsigned integer number with E bits. The bias makes sure that $exp \geq 0$. Also note that $e \in [-2^{E-1} + 1, 2^{E-1}\,]$.
  - ✓ The IEEE-754 standard reserves the following cases: i) $exp = 2^E - 1$ ($e = 2^{E-1}$) to represent special numbers (NaN and $\pm\infty$), and ii) $exp = 0$ to represent the zero and the denormalized numbers. The remaining cases are called ordinary numbers.

- **Ordinary numbers**:

	biased exponent	significand
±	e+bias∈[1,2^E-2]	f
	E	p

Range of e: $[-2^{E-1} + 2, 2^{E-1} - 1]$.
Max number: $largest\ significand \times 2^{largest\ exponent}$

$$max = 1.11\ldots1 \times 2^{2^{E-1}-1} = (2 - 2^{-p}) \times 2^{2^{E-1}-1}$$

Min. number: $smallest\ significand \times 2^{smallest\ exponent}$

$$min = 1.00\ldots0 \times 2^{-2^{E-1}+2} = 2^{-2^{E-1}+2}$$

$$Dynamic\ Range = \frac{max}{min} = \frac{(2 - 2^{-p}) \times 2^{2^{E-1}-1}}{2^{-2^{E-1}+2}} = (2 - 2^{-p}) \times 2^{2^E - 3}$$

$$Dynamic\ Range\ (dB) = 20 \times \log_{10}\{(2 - 2^{-p}) \times 2^{2^E-3}\}$$

- **Plus/minus Infinite:** $\pm\infty$

	biased exponent	significand
±	e+bias = 2^E-1	f=0
	E	p

The exp field is a string of 1's. This is a special case where $exp = 2^E - 1$. ($e = 2^{E-1}$)

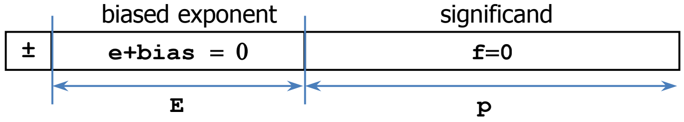
$$\pm\infty = \pm 2^{2^{E-1}}$$

- **Not a Number:** NaN

	biased exponent	significand
±	e+bias = 2^E-1	f≠0
	E	p

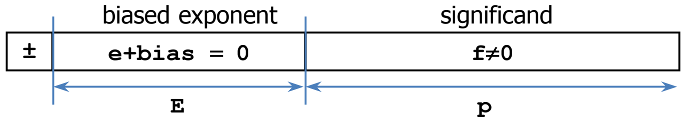
The exp field is a strings of 1's. $exp = 2^E - 1$. This is a special case where $exp = 2^E - 1$ ($e = 2^{E-1}$). The only difference with $\pm\infty$ is that f is a nonzero number.

- **Zero**:



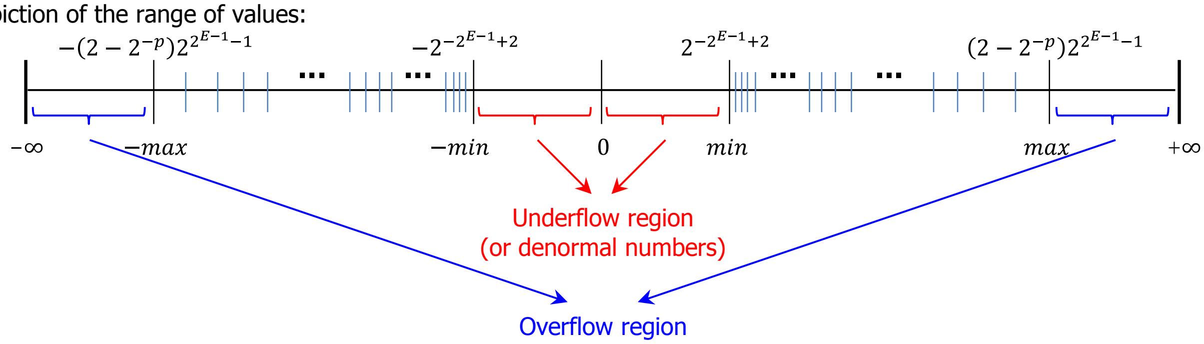
Zero cannot be represented with a normalized significand $s = 1.00\ldots0$ since $X = \pm1.f \times 2^e$ cannot be zero. Thus, a special code must be assigned to it, where $s = 0.00\ldots0$ and $exp = 0$. Every single bit (except for the sign) is zero. There are two representations for zero.
The number zero is a special case of the denormalized numbers, where $s = 0.f$ (see below).

- **Denormalized numbers**: The implementation of these numbers is optional in the standard (except for the zero). Certain small values that are not representable as normalized numbers (and are rounded to zero), can be represented more precisely with denormals. This is a "graceful underflow" provision, which leads to hardware overhead.



These numbers have the exp field equal to zero. The tricky part is that e is set to $-2^{E-1}+2$ (not $-2^{E-1}+1$, as the $e + bias$ formula states). The significand is represented as $s = 0.f$. Thus, the floating point number is $X = \pm0.f \times 2^{-2^{E-1}+2}$. These numbers can represent numbers lower (in absolute value) than min (the number zero is a special case).
Why is e not $-2^{E-1}+1$? Note that the smallest ordinary number is $2^{-2^{E-1}+2}$.
The largest denormalized number with $e = -2^{E-1}+1$ is: $0.11\ldots1 \times 2^{2^{E-1}-1} = (1-2^{-p}) \times 2^{-2^{E-1}+1}$.
The largest denormalized number with $e = -2^{E-1}+2$ is: $0.11\ldots1 \times 2^{2^{E-1}-2} = (1-2^{-p}) \times 2^{-2^{E-1}+2}$.
By picking $e = -2^{E-1}+2$, the gap between the largest denormalized number and the smallest normalized is smaller. Though this specification makes the formula $e + bias = 0$ inconsistent, it helps in accuracy.

- Depiction of the range of values:



- The IEEE-754-2008 (revision of IEEE-754-1985) standard defines several representations: half (16 bits, E=5, p=10), single (32 bits, E = 8, p = 23) and double (64 bits, E = 11, p = 52). There is also quadruple precision (128 bits) and octuple precision (256 bits). You can define your own representation by selecting a particular number of bits for the exponent and significand. The table lists various parameters for half, single and double FP arithmetic (ordinary numbers):

	Ordinary numbers		Exponent bits (E)	Range of e	Bias	Dynamic Range (dB)	Significand range	Significand bits (p)
	Min	Max						
Half	2^{-14}	$(2-2^{-10})2^{+15}$	5	$[-14,15]$	15	180.61 dB	$[1,2-2^{-10}]$	10
Single	2^{-126}	$(2-2^{-23})2^{+127}$	8	$[-126,127]$	127	1529 dB	$[1,2-2^{-23}]$	23
Double	2^{-1022}	$(2-2^{-52})2^{+1023}$	11	$[-1022,1023]$	1023	12318 dB	$[1,2-2^{-52}]$	52

- Rules for arithmetic operations:
  - ✓ $Ordinary\ number \div (+\infty) = \pm0$
  - ✓ $Ordinary\ number \div (0) = \pm\infty$
  - ✓ $(+\infty) \times Ordinary\ number = \pm\infty$
  - ✓ $NaN + Ordinary\ number = NaN$
  - ✓ $(0) \div (0) = NaN$ $(\pm\infty) \div (\pm\infty) = NaN$
  - ✓ $(0) \times (\pm\infty) = NaN$ $(\infty) + (-\infty) = NaN$

Examples:

- F43DE962 (single): 1111 0100 0011 1101 1110 1001 0110 0010
  $e + bias = 1110\,1000 = 232 \rightarrow e = 232 - 127 = 105$
  $Mantissa = 1.011\,1101\,1110\,1001\,0110\,0010 = 1.4837$
  $X = -1.4837 \times 2^{105} = -6.1085 \times 10^{31}$

- 007FADE5 (single): 0000 0000 0111 1111 1010 1101 1110 0101
  $e + bias = 0000\,0000 = 0 \rightarrow Denormal\ number \rightarrow e = -126$
  $Mantissa = 0.111\,1111\,1010\,1101\,1110\,0101 = 0.9975$
  $X = 0.9975 \times 2^{-126} = 1.1725 \times 10^{-38}$

ADDITION/SUBTRACTION

$b_1 = \pm s_1 2^{e_1}$, $s_1 = 1.f_1$ $\qquad\qquad$ $b_2 = \pm s_2 2^{e_2}$, $s_1 = 1.f_2$

$\rightarrow b_1 + b_2 = \pm s_1 2^{e_1} \pm s_2 2^{e_2}$

If $e_1 \geq e_2$, we simply shift s_2 to the right by $e_1 - e_2$ bits. This step is referred to as alignment shift.

$s_2 2^{e_2} = \frac{s_2}{2^{e_1-e_2}} 2^{e_1}$

$$\rightarrow b_1 + b_2 = \pm s_1 2^{e_1} \pm \frac{s_2}{2^{e_1-e_2}} 2^{e_1} = \left(\pm s_1 \pm \frac{s_2}{2^{e_1-e_2}}\right) \times 2^{e_1} = s \times 2^e$$

$$\rightarrow b_1 - b_2 = \pm s_1 2^{e_1} \mp \frac{s_2}{2^{e_1-e_2}} 2^{e_1} = \left(\pm s_1 \mp \frac{s_2}{2^{e_1-e_2}}\right) \times 2^{e_1} = s \times 2^e$$

- **Normalization**: Once the operators are aligned, we can add. The result might not be in the format $1.f$, so we need to discard the leading 0's of the result and stop when a leading 1 is found. Then, we must adjust e_1 properly, this results in e.
  - ✓ For example, for addition, when the two operands have similar signs, the resulting significand is in the range $[1,4)$, thus a single bit right shift is needed on the significant to compensate. Then, we adjust e_1 by adding 1 to it (or by left shifting everything by 1 bit). When the two operands have different signs, the resulting significand might be lower than 1 (e.g.: 0.000001) and we need to first discard the leading zeros and then right shift until we get $1.f$. We then adjust e_1 by adding the same number as the number of shifts to the right on the significand.

  Note that overflow/underflow can occur during the addition step as well as due to normalization.

  Example: $s_3 = \left(\pm s_1 \pm \frac{s_2}{2^{e_1-e_2}}\right) = 00011.1010$

  First, discard the leading zeros: $\quad s_3 = 11.1010$
  Normalization: right shift 1 bit: $\quad s = s_3 \times 2^{-1} = 1.11010$
  Now that we have the normalized significand s, we need to adjust the exponent e_1 by adding 1 to it: $e = e_1 + 1$:

  $$(s_3 \times 2^{-1}) \times 2^{e_1+1} = s \times 2^e = 1.1101 \times 2^{e_1+1}$$

  Example: $b_1 = 1.0101 \times 2^5$, $b_2 = -1.1110 \times 2^3$

  $$b = b_1 + b_2 = 1.0101 \times 2^5 - \frac{1.1110}{2^2} \times 2^5 = (1.0101 - 0.011110) \times 2^5$$

  $1.0101 - 0.011110 = 0.11011$. To get this result, we convert the operands to the 2C representation (you can also do unsigned subtraction if the result is positive). Here, the result is positive. Finally, we perform normalization:

  $$\rightarrow b = b_1 + b_2 = (0.11011) \times 2^5 = (0.11011 \times 2^1) \times 2^5 \times 2^{-1} = 1.1011 \times 2^4$$

- **Subtraction**: This operation is very similar.

  Example: $b_1 = 1.0101 \times 2^5$, $b_2 = 1.111 \times 2^5$

  $$b = b_1 - b_2 = 1.0101 \times 2^5 - 1.111 \times 2^5 = (1.0101 - 1.111) \times 2^5$$

  To subtract, we convert to 2C representation: $R = 01.0101 - 01.1110 = 01.0101 + 10.0010 = 11.0111$. Here, the result is negative. So, we get the absolute value ($|R| = 2C(1.0111) = 0.1001$) and place the negative sign on the final result:

  $$\rightarrow b = b_1 - b_2 = -(0.1001) \times 2^5$$

Example:

- ✓ X = `50DAD000 - D0FAD000`:

  `50DAD000`: `0101 0000 1101 1010 1101 0000 0000 0000`

  $e + bias = 10100001 = 161 \rightarrow e = 161 - 127 = 34$ $\qquad$ $Mantissa = 1.10110101101$

  `50DAD000` $= 1.10110101101 \times 2^{34}$

  `D0FAD000`: `1101 0000 1111 1010 1101 0000 0000 0000`

  $e + bias = 10100001 = 161 \rightarrow e = 161 - 127 = 34$ $\qquad$ $Mantissa = 1.11110101101$

  `D0FAD000` $= -1.11110101101 \times 2^{34}$

  $X = 1.10110101101 \times 2^{34} + 1.11110101101 \times 2^{34}$ (unsigned addition)

  $X = 11.1010101101 \times 2^{34} = 1.11010101101 \times 2^{35}$

  $e + bias = 35 + 127 = 162 = 10100010$

  X = `0101 0001 0110 1010 1101 0000 0000 0000` = `516AD000`

```
c12=1 c11=1 c10=1 c9=1 c8=1 c7=0 c6=1 c5=0 c4=1 c3=1 c2=0 c1=1 c0=0

   1.1 0 1 1 0 1 0 1 1 0 1 +
   1.1 1 1 1 0 1 0 1 1 0 1
  -------------------------
 1 1.1 0 1 0 1 0 1 1 0 1 0
```

Example:

- ✓ X = `60A10000 + C2F97000`:

  `60A10000`: `0110 0000 1010 0001 0000 0000 0000 0000`

  $e + bias = 11000001 = 193 \rightarrow e = 193 - 127 = 66$ $\quad Mantissa = 1.0100001$

  `60A10000` = 1.0100001×2^{66}

  `C2F97000`: `1100 0010 1111 1001 0111 0000 0000 0000`

  $e + bias = 10000101 = 133 \rightarrow e = 133 - 127 = 6$ $\quad Mantissa = 1.11110010111$

  `C2F97000` = $-1.11110010111 \times 2^{6}$

  $X = 1.0100001 \times 2^{66} - 1.11110010111 \times 2^{6}$

  $X = 1.0100001 \times 2^{66} - \frac{1.11110010111}{2^{60}} \times 2^{66}$

  Representing the division by 2^{60} requires more than $p + 1 = 24$ bits. Thus, we can approximate the 2nd operand with 0.

  $X = 1.0100001 \times 2^{66}$

  X = `0110 0000 1010 0001 0000 0000 0000 0000` = `60A10000`

MULTIPLICATION

$b_1 = \pm s_1 2^{e_1}$, $b_2 = \pm s_2 2^{e_2}$

$\rightarrow b_1 \times b_2 = (\pm s_1 2^{e_1}) \times (\pm s_2 2^{e_2}) = \pm(s_1 \times s_2) 2^{e_1 + e_2}$

Note that $s = (s_1 \times s_2) \in [1,4)$.

Example:

$b_1 = 1.100 \times 2^2$, $b_2 = -1.011 \times 2^4$,

$b = b_1 \times b_2 = -(1.100 \times 1.011) \times 2^6 = -(10{,}0001) \times 2^6$,

Normalization of the result:
$b = -(10{,}0001 \times 2^{-1}) \times 2^7 = -(1{,}00001) \times 2^7$.

Note that if the multiplication requires more bits than allowed by the representation (32, 64 bits), we have to do truncation or rounding. It is also possible that overflow/underflow might occur due to large/small exponents and/or multiplication of large/small numbers.

Example:

- ✓ X = `7A09D300 × 0BEEF000`:

  `7A09D300`: `0111 1010 0000 1001 1101 0011 0000 0000`

  $e + bias = 11110100 = 244 \rightarrow e = 244 - 127 = 117$ $\quad Mantissa = 1.00010011101001100000000$

  `7A09D300` = $1.000100111010011 \times 2^{117}$

  `0BEEF000`: `0000 1011 1110 1110 1111 0000 0000 0000`

  $e + bias = 00010111 = 23 \rightarrow e = 23 - 127 = -104$ $\quad Mantissa = 1.11011101111000000000000$

  `0BEEF000` = $1.11011101111 \times 2^{-104}$

  $X = 1.000100111010011 \times 2^{117} \times 1.11011101111 \times 2^{-104}$

  $X = 10.0000000101000110101111111101 \times 2^{13} = 1.00000000101000110101111111101 \times 2^{14} = 1.6466 \times 10^4$

  $e + bias = 14 + 127 = 141 = 10001101$

  X = `0100 0110 1000 0000 1010 0011 0101 1111` = `4680A35F` (four bits were truncated)

DIVISION

$b_1 = \pm s_1 2^{e_1}$, $b_2 = \pm s_2 2^{e_2}$

$$\rightarrow \frac{b_1}{b_2} = \frac{\pm s_1 2^{e_1}}{\pm s_2 2^{e_2}} = \pm \frac{s_1}{s_2} 2^{e_1 - e_2}$$

Note that $s = \left(\frac{s_1}{s_2}\right) \in (1/2, 2)$
Here, the result might require normalization.

Example:
$b_1 = 1.100 \times 2^2$, $b_2 = -1.011 \times 2^4$

$$\rightarrow \frac{b_1}{b_2} = \frac{1.100 \times 2^2}{-1.011 \times 2^4} = -\frac{1.100}{1.011} 2^{-2}$$

$\frac{1.100}{1.011}$: unsigned division, here we can include as many fractional bits as we want.

With $x = 4$ (and $a = 0$) we have:

$$\frac{1100\mathbf{0000}}{1011} \Rightarrow 1100\mathbf{0000} = 10101(1011) + 11$$

$$Q_f = 1{,}0101, R_f = 00{,}0011$$

If the result is not normalized, we need to normalized it. In this example, we do not need to do this.

$$\rightarrow \frac{b_1}{b_2} = \frac{1.100 \times 2^2}{-1.011 \times 2^4} = -1.0101 \times 2^{-2}$$

Example

✓ X = 49742000 ÷ 40490000:

49742000: 0100 1001 0111 0100 0010 0000 0000 0000

$e + bias = 10010010 = 146 \rightarrow e = 146 - 127 = 19$ $Mantissa = 1.11101000010000000000000$

497420000 = $1.1110100001 \times 2^{19}$

40490000: 0100 0000 0100 1001 0000 0000 0000 0000

$e + bias = 10000000 = 128 \rightarrow e = 128 - 127 = 1$ $Mantissa = 1.10010010000000000000000$

0BEEF000 = 1.1001001×2^1

$$X = \frac{1.1110100001 \times 2^{19}}{1.1001001 \times 2^1}$$

```
              0000000000100110110
11001001000 ) 1111010000100000000
              11001001000
                101011001000
                 11001001000
                  100100000000
                   11001001000
                     101011100000
                      11001001000
                       100100110000
                        11001001000
                          10111010000
```

Alignment:

$$\frac{1.1110100001}{1.1001001} = \frac{1.1110100001}{1.1001001000} = \frac{11110100001}{11001001000}$$

Append x = 8 zeros: $\frac{11110100001\mathbf{00000000}}{11001001000}$

Integer division
$Q = 100110110, R = 1011101000 \rightarrow Qf = 1.00110110$

Thus: $X = \frac{1.1110100001 \times 2^{19}}{1.1001001 \times 2^1} = 1.0011011 \times 2^{18} = 1.2109375 \times 2^{18} = 317440$
$e + bias = 18 + 127 = 145 = 10010001$

X = 0100 1000 1001 1011 0000 0000 0000 0000 = 489B0000