Part 2:

Introduction to Binary Numbers & Arithmetic

Decimal vs. binary representation

The *decimal* number 805 means

$8 \text{x} 10^2 + 0 \text{x} 10^1 + 5 \text{x} 10^0$.

The *place values*, from right to left, are 1,10,100,1000,..., or $10^0,10^1,10^2,10^3,\ldots$.

The *base* or *radix* is 10.

All digits must be less than the base, i.e. from 0 to 9.

The *binary* number 1011_2 means

$1 \text{x} 2^3 + 0 \text{x} 2^2 + 1 \text{x} 2^1 + 1 \text{x} 2^0$.

The *place values*, from right to left, are 1,2,4,8,..., or $2^0,2^1,2^2,2^3,\ldots$.

The *base* or *radix* is 2 (in decimal) or 10_2 (in binary).

All digits must be less than the base, i.e. either 0 or 1.

Advantages of binary representation

Binary notation is convenient for electronic processing because:

(1) Only two voltage levels are needed to represent all digits;

(2) Arithmetic tables are simple, and can be implemented using logic gates.

Addition table:

$0+0 = 00$ $0+1 = 01$

$1+0 = 01$ $1+1 = 10$

Multiplication table:

$0 \times 0 = 0$ $0 \times 1 = 0$

$1 \times 0 = 0$ $1 \times 1 = 1$

Each table has 4 entries. In decimal, each table would have 100_{10} entries! (Notice that $4 = 100_2$.)

Converting from base x

While computers work in base 2, people prefer base 10. So conversions to and from binary are needed. We can illustrate the methods using a 4-digit integer in an arbitrary base. The number $abcd_x$ (base x) means

$$ax^3 + bx^2 + cx + d.$$

This polynomial can be used directly to convert the number to decimal. We can reduce the number of arithmetical operations in the conversion by writing the polynomial in *nested form*:

$$((ax + b)\, x + c)\, x + d.$$

Converting from base x - Examples

Taking the example from slide 2,

$$1011_2 = 1 .\ 2^3 + 0 .\ 2^2 + 1 .\ 2^1 + 1 .\ 2^0 = 8 + 0 + 2 + 1 = 11$$

or, in nested form,

$$1011_2 = ((1 .\ 2 + 0) .\ 2 + 1) .\ 2 + 1 = 11.$$

Here's an example in base 7:

$$41035_7 = (((4 .\ 7 + 1) .\ 7 + 0) .\ 7 + 3) .\ 7 + 5 = 9973.$$

In base 16 (known as *hexadecimal* or "hex"), we use the letters A to F to represent the "digits" 10 to 15 :

$$FFFF_{hex} = ((15 .\ 16 + 15) .\ 16 + 15) .\ 16 + 15 = 65535.$$

Converting to base *x*

The nested form

$$((ax + b)\,x + c)\,x + d$$

can be written

$$abcd_x = Cx + d$$

where $C = Bx + c$

where $B = ax + b$

where $a = 0x + a$.

Now all the above are whole numbers, and a,b,c,d are all less than x, being the digits of a base-x number. So the equations at left imply that d,c,b,a are the remainders when $abcd_x$ is divided repeatedly by x.

To convert a whole number to base x, divide it repeatedly by x until the quotient is zero, and write the remainders in reverse.

Converting to base x - Examples

To convert 11_{10} to binary, we repeatedly divide by 2. Let us write the quotients on the left and remainders on the right:

11	
5	1
2	1
1	0
0	1

Reading the remainders up the column gives 1011_2.

To convert 9973 to base 7:

9973	
1424	5
203	3
29	0
4	1
0	4

The result is 41035_7.

Converting fractions from base x

The number *abcd*. *pqrs*$_x$ (base *x*) means

$$ax^3 + bx^2 + cx + d + px^{-1} + qx^{-2} + rx^{-3} + sx^{-4}.$$

The part to the left of the *radix point* (.) is the integer part and the part to the right is the fractional part. Again, the above expression may be used to convert the number to decimal. (Exercise: Can you devise a *nested form* to speed up the calculation?)

To convert from decimal to base *x*, the integer part is processed by the repeated-division method, while the fractional part requires separate treatment [next slide].

Converting fractions to base x

The fractional part (red) of *abcd.pqrs*$_x$ is

$$px^{-1} + qx^{-2} + rx^{-3} + sx^{-4}\,.$$

Multiply by x :

$$p + qx^{-1} + rx^{-2} + sx^{-3}\,.$$

Take the fractional part and multiply by x again:

$$q + rx^{-1} + sx^{-2}\,.$$

Repeat the separate-and-multiply step until there is no fractional part left*:

$$r + sx^{-1}$$

$$s + 0.$$

Now read off the resulting integer parts (red) in *forward order*, and you get the base-x digits *pqrs*.

* If the process doesn't terminate itself, stop when you've got enough digits.

Example: Convert 11.8125 to base 8

For the integer part, we repeatedly divide by 8 (remainders are in blue):

11	
1	3
0	1

Integer part is 13_8 .

For the fractional part, we repeatedly multiply by 8 [next column].

Each line in the table is 8 times the fractional part of the previous line:

.8125

6.5

4.0

Fractional part is $.64_8$.

Answer:

$$11.8125_{10} = 13.64_8 .$$

Example: Convert 6.4_{10} to base 7

The integer part is trivial:

6
0 6

Integer part is 6_7 .

The fractional part is 4/10. Since 10 is not divisible by 7, repeated multiplication by 7 will never cancel the denominator and never convert the fractional part to an integer. So the process will not terminate:

.4
2.8
5.6
4.2
1.4

...and this pattern repeats.

Answer: 6.25412541...

Notes on base conversions

In explaining how to convert *to* base *x* [slides 6 & 9], we have written the given numbers in powers of *x* to show why the methods work. In the worked examples, of course, we have written the given numbers in decimal.

Because we humans prefer base 10, we use repeated division to convert to other bases, and power-series expressions to convert to base 10.

But a computer prefers base 2. So it works the other way around, using repeated division to express its results in base 10, and power-series expressions to convert human input to base 2.

Octal (base 8)

It is especially easy to convert between octal and binary:

$abcdefgh_2$

$$= a.2^7 + b.2^6 + c.2^5 + d.2^4 + e.2^3 + f.2^2 + g.2 + h$$

$$= (a.2 + b).2^6 + (c.2^2 + d.2 + e).2^3 + (f.2^2 + g.2 + h)$$

$$= (ab_2).8^2 + (cde_2).8 + (fgh_2)$$

The expressions in parentheses, being less than 8, are the octal digits. The process can also be reversed. Note that one octal digit corresponds to three binary digits because $8 = 2^3$. The binary digits ("bits") are grouped from right to left, i.e. away from the radix point.

Binary-octal conversion - Examples

(1) Convert $10111110101100011010001000_2$ to octal :

$$010\ 111\ 110\ 101\ 100\ 011\ 010\ 001\ 000\ *$$

$$=\ 2\quad 7\quad 6\quad 5\quad 4\quad 3\quad 2\quad 1\quad 0_8$$

$$=\ 276543210_8\ .$$

(* The leading 0 is optional. Each conversion of three binary digits to one octal digit is done by inspection.)

(2) Fractional parts are grouped from left to right and padded with zeros (the proof is left as an exercise).

Example: Convert 11111111.10001_2 to octal:

$$011\ 111\ 111\ .\ 100\ 010_2 = 377.42_8\ .$$

Hexadecimal or "hex" (base 16)

We have seen that one octal digit corresponds to 3 bits because $8 = 2^3$. Similarly, one hex digit corresponds to 4 bits because $16 = 2^4$. The following generalized example includes a fractional part, which must be padded with a 0:

$$
\begin{aligned}
&abcdef.ghijklm_2 \\
&= a.2^5 + b.2^4 + c.2^3 + d.2^2 + e.2 + f \\
&\quad + g.2^{-1} + h.2^{-2} + i.2^{-3} + j.2^{-4} + k.2^{-5} + l.2^{-6} + m.2^{-7} \\
&= (a.2 + b).2^4 + (c.2^3 + d.2^2 + e.2 + f) \\
&\quad + (g.2^3 + h.2^2 + i.2 + j).2^{-4} + (k.2^3 + l.2^2 + m.2^1 + 0).2^{-8} \\
&= (ab_2).16 + (cdef_2) + (ghij_2).16^{-1} + (klm0_2).16^{-2}.
\end{aligned}
$$

Hex-binary conversion - Examples

(1) Convert 789ABCDEF$_{\text{hex}}$ to binary:

0111 1000 1001 1010 1011 1100 1101 1110 1111$_2$.

(The leading 0 can be omitted. Conversion of individual hex digits can be done by inspection; recall that the letters A to F represent the “digits” 10 to 15.)

(2) Convert 1011100.101101$_2$ to hex:

0101 1100 . 1011 0100$_2$ = 5C.B4$_{\text{hex}}$.

(The digits are counted off away from the radix point. The trailing zeros on the fractional part are needed to complete the group of four. The leading 0 is optional.)

Conversion to binary via octal

The direct conversion of 2001_{10} to binary looks like this ...

2001	
1000	1
500	0
250	0
125	0
62	1
31	0
15	1
7	1
3	1
1	1
0	1

... and gives 11111010001.

It may be quicker to convert to octal first ...

2001	
250	1
31	2
3	7
0	3

... yielding 3721_8, which can be instantly converted to $11\ 111\ 010\ 001_2$.

Negative numbers & subtraction

Mathematicians define subtraction as addition of the additive inverse:

$$a - b = a + (-b).$$

In theory, this trick reduces subtraction to addition. In practice, we still need subtraction because we use a magnitude-sign notation for negative numbers.

That is, if b is positive, we write its additive inverse as

$$-b = -|b|$$

and we evaluate $a + (-b)$ as $a - b$.

To eliminate subtraction in base-10 integer arithmetic, we can represent $-b$ by the nines complement or tens complement of b.

Nines-complements

If we confine the discussion to *4-digit decimal arithmetic*, the nines complement of b is defined as

$$b' = 10^4 - 1 - b = 9999 - b.$$

Evaluation of the nines complement does not require the full subtraction algorithm, because there is no borrowing. Each digit is simply subtracted from 9.

In nines-complement arithmetic, we represent $-b$ by b'. The numbers 0 to 4999 are represented literally, while -0 to -4999 are represented by 9999 down to 5000. Zero can be represented as 0000 or 9999.

Subtraction by nines-complements

Suppose a and b are in the range 0 to 4999. Then

$$a+b' = a + 9999 - b = 9999 + a - b = 9999 - (b-a).$$

If $a=b$, then $a+b' = 9999$, which means 0, which is $a-b$. If $a>b$, then $a+b'$ is at least 10^4, and we must subtract 9999 to obtain $a-b$; this can be done by adding the carry from the leftmost column ("*end-around carry*"). If $a<b$, we see from the green expression that $a+b' = (b-a)'$, which represents $a-b$ (and has no end-carry). Also,

$$a'+b' = 9999 - a + 9999 - b = 9999 + (a+b)'.$$

The end-around carry leaves $(a+b)'$, which means $-a-b$.

Nines-complement examples

(1) 2708 - 1984:

2708

\+ 8015 (= 1984’)

= 10723

└→ 1 (end-around carry)

= 0724.

(2) 1984 - 2708:

1984

\+ 7291 (= 2708’)

= 9275 (= 0724’).

End-around carry is zero.

Result means -724.

(3) -2708 - 1984:

7291 (= 2708’)

\+ 8015 (= 1984’)

= 15306

└→ 1 (end-around carry)

= 5307 (= 4692’).

Result means -4692.

(4) 2708 + 1984:

This is trivial, as no conversions are required. The result is 4692.

Tens-complements

We can eliminate the double representation of zero and the end-around-carry by using the tens complement, which is found by adding 1 to the nines complement. Hence, in *4-digit decimal arithmetic*, the tens complement of b is defined as

$$b^* = 10^4 - b.$$

In tens-complement arithmetic, 0 to 4999 are represented literally, while -1 to -5000 are represented by 9999 down to 5000, which are the tens complements of 1 to 5000. Zero is always 0000. Note that +5000 is not represented.

Subtraction by tens-complements

Again, suppose a,b are in the range 0 to 4999. Then

$$a+b^* = a + 10^4 - b = 10^4 + a - b = 10^4 - (b-a).$$

If $a=b$ or $a>b$, then $a+b^*$ is at least 10^4 and reduces to $a-b$ if we throw away the carry. If $a<b$, we see from the green expression that $a+b^* = (b-a)^*$, which is less than 10^4 (leaving no carry) and represents $a-b$. Also,

$$a^*+b^* = 10^4 - a + 10^4 - b = 10^4 + (a+b)^*.$$

Discarding the carry leaves $(a+b)^*$, which means $-a-b$.

In all cases, discarding the carry (if any) gives the tens-complement representation of the expected result.

Tens-complement examples

(1) 2708 - 1984:

2708
\+ 8016 (= 1984*)
= 10724
or 0724 (discarding carry).

(2) 1984 - 2708:

1984
\+ 7292 (= 2708*)
= 9276 (= 0724*).
No carry to discard.
Result means -724.

(3) -2708 - 1984:

7292 (= 2708*)
\+ 8016 (= 1984*)
= 15308
or 5308 (discarding carry).
Result is 4692*
and means - 4692.

(4) 2708 + 1984:

This is trivial. The result is 4692.

[Slide 21 does the same examples in nines-comp.]

Overflow in tens-complement

Suppose a,b are in the range 0 to 4999. Then

$$a+b^* = 10^4 + a - b = 10^4 - (b - a).$$

The result is in the range 5001 to 14999. After the carry (if any) is dropped, this represents -4999 to +4999, which is the correct range for a-b.

But if $a+b > 4999$, then $a+b$ represents a negative number; this is positive overflow.

Recall that a^*+b^* becomes $(a+b)^*$ when the carry is dropped. If $a+b > 5000$, then $(a+b)^* < 5000$ and stands for a positive number, not -a-b; this is negative overflow.

Negative numbers in binary

The nines complement in decimal corresponds to the ones complement in binary. In both notations, the carry from the most significant digit is added to the least significant digit (“end-around carry”).

The tens complement in decimal corresponds to the twos complement in binary. In both notations, the carry from the most significant digit is dropped.

[The next 10 slides concern ones- and twos complements.]

Ones-complements

In *n-digit binary arithmetic*, the ones complement of b is

$$b' = 2^n - 1 - b.$$

In binary, $2^n - 1$ is a row of n ones. So to find b', we subtract each digit from 1, or invert each digit; this is called a bitwise inversion.

In ones-complement arithmetic, we represent $-b$ by b'. The numbers 0 to $2^{n-1} - 1$ are represented literally [for n=4, these numbers are 0000 to 0111], while -0 to $-(2^{n-1} - 1)$ are represented by $2^n - 1$ down to 2^{n-1} [1111 down to 1000]. Zero can be represented as 0 or $2^n - 1$ [0000 or 1111].

Subtraction by ones-complements

Suppose a,b are in the range 0 to $2^{n-1} - 1$. Then

$$a+b' = a + 2^n - 1 - b = 2^n - 1 + a - b = 2^n - 1 - (b-a).$$

If $a=b$, then $a+b' = 2^n - 1$, which means 0, which is $a-b$. If $a>b$, then $a+b'$ is at least 2^n, and we must subtract $2^n - 1$ to obtain $a-b$; this subtraction can be accomplished by the *end-around carry*. If $a<b$, then $a+b' = (b-a)'$, which represents $a-b$ (and has no end-carry). Also,

$$a'+b' = 2^n - 1 - a + 2^n - 1 - b = 2^n - 1 + (a+b)'.$$

The end-around carry leaves $(a+b)'$, which means $-a-b$. So $a-b$ and $-a-b$ evaluate correctly in ones-complement.

4-bit ones-complement examples

(1) 0101 - 0010 (5 - 2):

0101

\+ 1101 (= 0010’)

= 10010

↳ 1 (end-around carry)

= 0011 (= 3).

(2) 0010 - 0101 (2 - 5):

0010

\+ 1010 (= 0101’)

= 1100 (= 0011’).

End-around carry is zero.

Result means -3.

(3) -0101 - 0010 (-5 - 2):

1010 (= 0101’)

\+ 1101 (= 0010’)

= 10111

↳ 1 (end-around carry)

= 1000 (= 0111’).

Result means -7.

(4) 0101 + 0010 (5 + 2):

This is trivial, as no conversions are required. The result is 0111 (= 7).

Twos-complements

In *n-digit binary arithmetic*, the twos complement of *b* is

$$b^* = b' + 1 = 2^n - b\ .$$

In twos-complement arithmetic, the values 0 to $2^{n-1} - 1$ [0000 to 0111 for $n = 4$] are represented literally, while the values -1 to -2^{n-1} [-0001 to -1000] are represented by $2^n - 1$ down to 2^{n-1} [1111 down to 1000], which are the twos complements of 1 to 2^{n-1}. Note that 2^{n-1} [1000] represents the value -2^{n-1} [-1000] while the value $+2^{n-1}$ [+1000] is not represented. N.B.: Negative numbers are marked by a 1 in the leftmost bit or Most Significant Bit (MSB). So the MSB is also the sign bit.

Subtraction by twos-complements

Again, suppose a,b are in the range 0 to $2^{n-1} - 1$. Then

$$a+b^* = a + 2^n - b = 2^n + a - b = 2^n - (b-a).$$

If $a=b$ or $a>b$, then $a+b^*$ is at least 2^n and reduces to $a-b$ if we throw away the end-carry (subtracting 2^n). If $a<b$, then $a+b^* = (b-a)^*$, which represents $a-b$ (and there is no end-carry to throw away). Also,

$$a^*+b^* = 2^n - a + 2^n - b = 2^n + (a+b)^*.$$

Dropping the carry leaves $(a+b)^*$, which means $-a-b$.

In all cases, discarding the carry (if any) gives the twos-complement representation of the expected result.

4-bit twos-complement examples

(1) 0101 - 0010 (5 - 2):

0101
+ 1110 (= 0010*)
= 10011
or 0011 (discarding carry).

(2) 0010 - 0101 (2 - 5):

0010
+ 1011 (= 0101*)
= 1101 (= 0011*).
No carry to discard.
Result means -3.

(3) -0101 - 0010 (-5 - 2):

1011 (= 0101*)
+ 1110 (= 0010*)
= 11001
or 1001 (discarding carry).
Result is 0111*
and means -7.

(4) 0101 + 0010 (5 + 2):

This is trivial, as no conversions are required. The result is 0111 (= 7).

[Slide 29 does the same examples in ones-comp.]

Overflow in twos-complement

Suppose a,b are in the range 0 to $2^{n-1} - 1$. Then

$$a+b^* = 2^n + a - b = 2^n - (b - a).$$

The result is in the range $2^{n-1} + 1$ to $2^n + 2^{n-1} - 1$. After any carry is dropped, this represents $-(2^{n-1} - 1)$ to $2^{n-1} - 1$, which is the correct range for a-b.

But if $a+b > 2^{n-1} - 1$, then $a+b$ represents a negative number; this is positive overflow.

Recall that a^*+b^* becomes $(a+b)^*$ when the carry is dropped. If $a+b > 2^{n-1}$, then $(a+b)^* < 2^{n-1}$ and represents a positive number, not $-a$-b; this is negative overflow.

Positive overflow detection

Addition of 4-bit positive numbers *without overflow* looks like this:

```
   0xxx
+  0xxx
=  0xxx .
```

The carry in to the MSB must have been 0, and the carry out is 0. (We can repeat the illustration for any number of bits.)

Positive overflow looks like this:

```
   0xxx
+  0xxx
=  1xxx .
```

The carry in to the MSB must have been 0, but the carry out is 1.

Overflow occurs when

carry in $\neq$ carry out.

Negative overflow detection

Addition of negative twos-complement numbers *without overflow*:

```
      1xxx
  +   1xxx
  =  11xxx .
```

The carry in to the MSB must have been 1 (otherwise the sum bit would be 0), and the carry out is 1.

Negative overflow:

```
      1xxx
  +   1xxx
  =  10xxx .
```

The carry in to the MSB must have been 0, but the carry out is 1.

So negative overflow, like positive, occurs when

carry in ≠ carry out.

Hardware overflow signal

We have seen that the condition for twos-complement* overflow is

carry in $\neq$ carry out

in the MSB. An "XOR" gate has an output of 1 when the inputs are unequal. So if we define the overflow flag as

overflow = (carry in) xor (carry out),

it will be 1 (or "set" or "true") when an overflow occurs.

* The same condition works for ones-complement. To prove this, we have to consider which of the cases on the preceding two slides can involve 1111, which is *not* negative and breaks the rule that a 1 in the MSB signals a negative number. The proof is left as an exercise.

Half adder

When two binary numbers are added [cf. slides 29 and 32], the right-hand bits are added using this addition table [cf. slide 3]:

0+0 = 00   0+1 = 01

1+0 = 01   1+1 = 10

In the two-bit sum, the right bit is the sum bit and the left bit the carry bit.

Let the bits to be added be *A* and *B*, the sum bit *S* and the carry bit *C*. Then the addition table may be expressed as a truth table:

A	*B*	*S*	*C*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

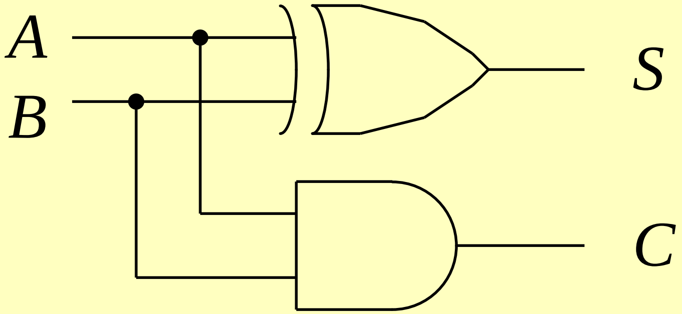
Half adder (continued)

The truth table is fully described by the Boolean relations

$$S = A \text{ xor } B$$

$$C = AB$$

which lead directly to the gate implementation:



In hand calculations, the sum bit is written under the column, while the carry bit is added to the next column to the left; that column has *three* bits to be added. An adder accepting three 1-bit inputs is called a full adder [next slide]; one accepting only two inputs is called a half adder [left].

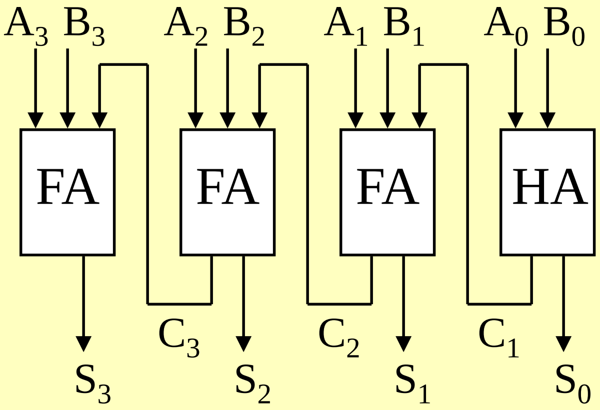
Full adder - used in 4-bit adder

A full adder adds three numbers each of which can take the values 0 and 1. The result lies in the range 00_2 to 11_2 and can be represented by a sum bit and carry bit, as in a half adder.

Suppose we want to add two 4-bit numbers $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$.

Let the sum be $S_3S_2S_1S_0$, and let C_1 be the carry into the column of A_1 and B_1, etc. Let FA denote a full adder and HA a half adder.

The circuit is:



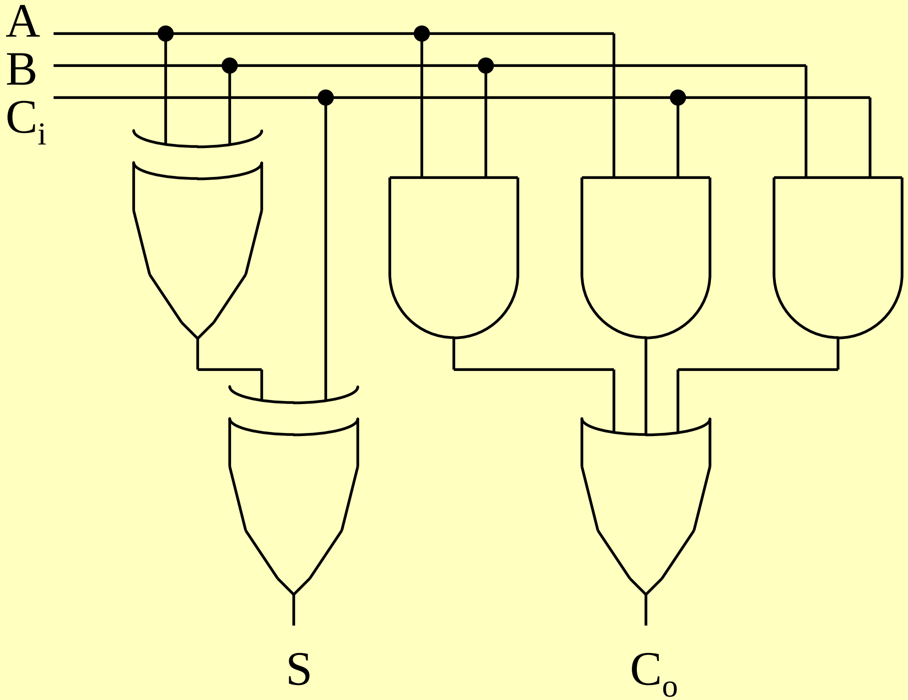
Full adder - implementation (1)

Let C_i be the carry-in, S the sum, and C_o the carry-out.

S is 1 if the inputs A,B,C_i include an odd number of 1's. C_o is 1 if any two (or more) inputs are 1's.

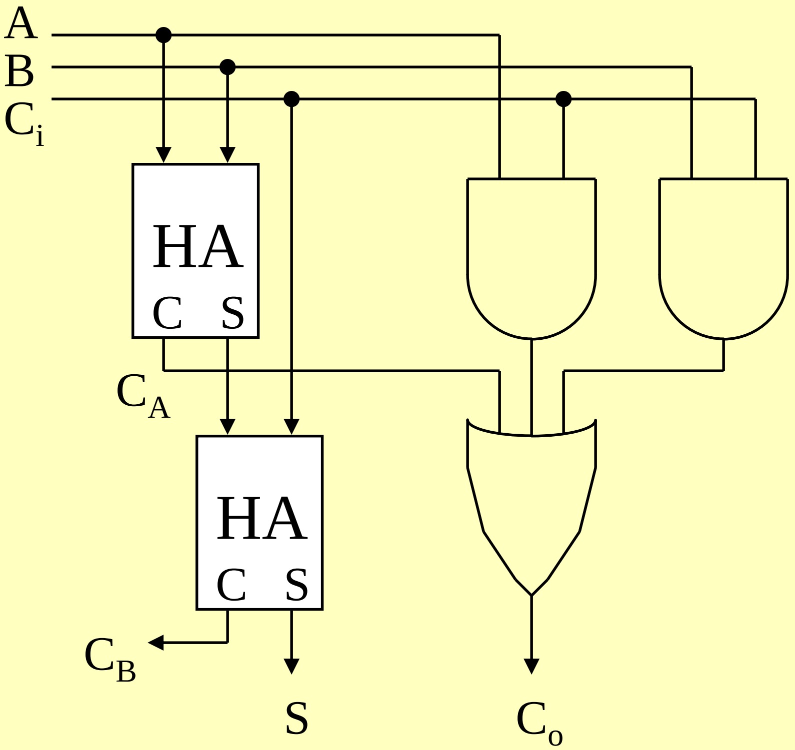
TRUTH TABLE

A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Full adder - implementation (2)

In the previous slide, the XOR and AND gates at the top left comprise a half adder, and the lower XOR gate is part of a half adder. So the circuit may be redrawn as shown here. The labels ‘C’ and ‘S’ distinguish between the outputs of each HA. The carry-out variables from the two HAs are called C_A and C_B ; C_B is unused.



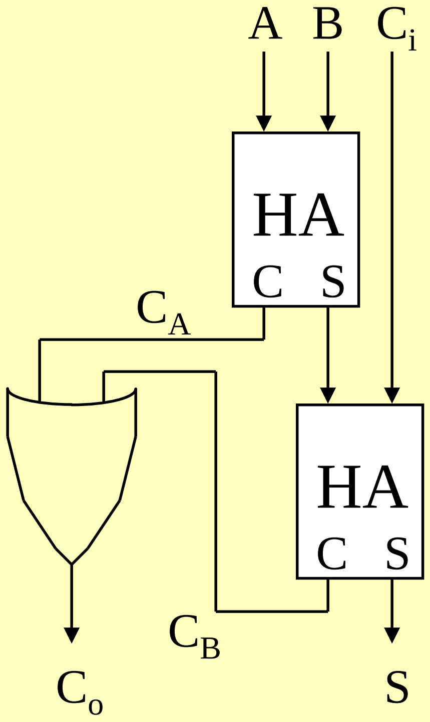
Full adder - implementation (3)

Now let’s see if we can use C_B in the calculation of C_o .

TRUTH TABLE

A	B	C_i	C_A	C_B	C_o
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	0	1

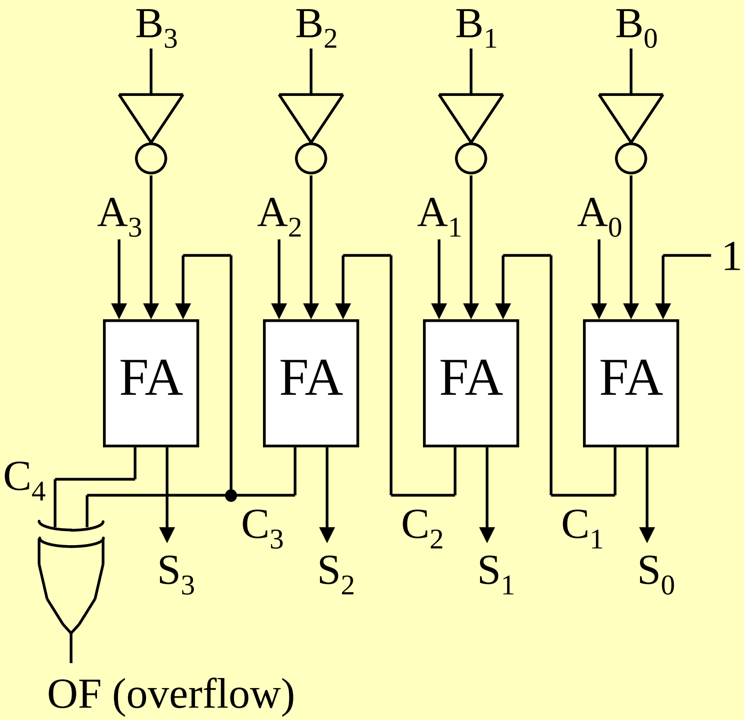
From the truth table we see that $C_o = C_A$ or C_B . So the FA circuit simplifies to:



Full adder - used in 4-bit subtractor

To subtract $B_3B_2B_1B_0$ from $A_3A_2A_1A_0$ using twos-complement arithmetic, we add $A_3A_2A_1A_0$ to the twos complement of $B_3B_2B_1B_0$. To find the complement, we invert the bits and add one. The "add one" step can be done by replacing the right-hand half adder with a full adder.

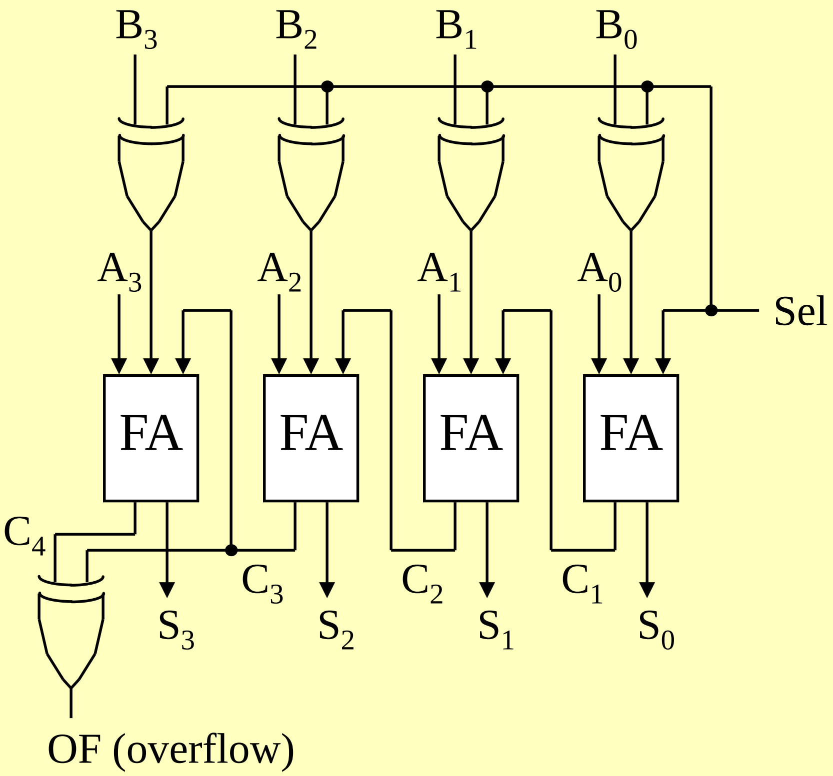
Subtractor (with overflow):



4-bit adder/subtractor

An XOR gate is a controlled inverter. If one input is 0, the output is the other input; if one input is 1, the output is the inverse of the other input. We can use XOR gates to select between $B_3B_2B_1B_0$ and its ones complement. The Sel(ect) input also controls the "add one" step.

Adder/subtractor:



The "ripple-carry" effect.

In the 4-bit adders shown on the last few slides, the carry-out from each FA is connected to the carry-in of the next. This system is called ripple-carry.

Logic gates do not react instantaneously to changes in their inputs. There is a *delay* in the calculation of C_1. When C_1 reaches its correct value, there is a *further delay* in the calculation of C_2 (which depends on C_1), and so on. So the carry "ripples" through the full adders. The sum bits also depend on the incoming carry bits, causing a *cumulative delay* in the calculation of the sum.

Carry acceleration (1)

Methods for reducing carry delay include the carry-select adder (CSA) and carry look-ahead (CLA). These are useful when we connect several 4-bit adders in cascade to make a larger adder.

In a 4-bit CSA, we have *two complete 4-bit adders*, one with a carry-in of 0 and the other with a carry-in of 1. The "real" carry-in is used to *select between the outputs* of the two adders. The two sums and two carry-outs can be computed while waiting for the carry-in from the previous adder in the chain.

Carry acceleration (2)

In an adder with carry look-ahead (CLA), each FA has *two* carry outputs, called generate carry (G) and propagate carry (P). G means “$C_o = 1$” (regardless of C_i), while P means “$C_o = C_i$”. Using G and P as intermediate values, all the carry-ins to a 4-bit adder can be computed from the inputs and C_0 (C_0 is the least significant or rightmost C_i).

We can also produce G and P outputs for the whole 4-bit adder; P means “$C_4 = C_0$”. When several 4-bit adders are cascaded, the G and P outputs of each adder can be combined like those of a single FA; the combining circuit is called a carry look-ahead generator (CLAG).

Arithmetic Logic Units (ALUs)

The 4-bit adder/subtractor [slide 44] is the simplest example of a multifunction arithmetic unit; the “Sel” input selects the desired function from the available options.

A more realistic multi-function unit would have more functions, controlled by several “select” bits.

One select bit might determine whether the function is arithmetical (add, divide-by-2) or logical (bitwise XOR, shift-right).

A multifunction circuit with arithmetical and logical functions is called an arithmetic logic unit (ALU).