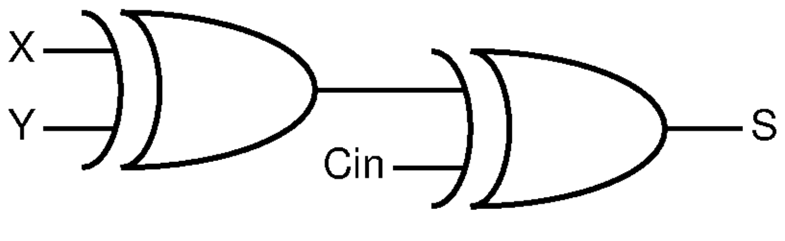
Designing Computer Systems

Arithmetic

$$\begin{array}{cccc} 1 & 1 & & \\ & 1 & 1 & 0 \\ + & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 1 \end{array}$$



Designing Computer Systems

Arithmetic

Arithmetic in digital systems involves a few familiar operations (addition, subtraction, multiplication, and division) on quantitative representations described in Number Systems. Everyone knows what the answer should be. The challenge is making hardware that performs these operations, and detecting when the representation cannot capture the result (overflow). This chapter combines Number Systems and Gate Design to define and implement addition and subtraction.

Addition: Addition is a simple *dyadic* operation in that operates on two operands, the addends, to produce a result, a sum. The rules of addition were first learned in an early grade on decimal values.

```
                             1         1 1        1 1
   6 8 3       6 8 3       6 8 3       6 8 3       6 8 3
 +  3 6 5    +  3 6 5    +  3 6 5    +  3 6 5    +  3 6 5
 --------    --------    --------    --------    --------
                    8         4 8       0 4 8     1 0 4 8
```

When adding these values, one starts at the least significant digit. Adding three and five is easy. The result, eight is expressible within the significance of this digit (the one's place). So this place is done. The next digit, in the ten's place is more complicated. The sum of eight (80) and six (60) is 14 (140). But this cannot be expressed fully in the tens place. So the six (60) is recorded and the ten (100) is carried to the next place, the hundreds place. The sum of six (600), three (300), and the carried in one (100) sums to one thousand. Again, this cannot be captured in the hundred's place. So it is carried to the next digit, the thousand's place. This leads to a habit that humans have, but digital systems cannot support: The presumption of unlimited bit resolution. Humans assume that if there is space in the result line, it can be used to fully, and accurately express the result. Unfortunately digital systems must live within the available bits in the representation. If the representation if three decimal digits, 000 to 999, it cannot represent 1048. So there is an *overflow* error.

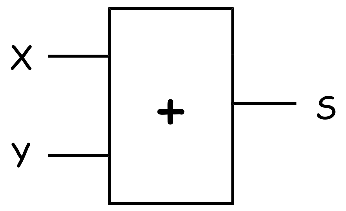
Moving to binary addition is simply a matter of employing a binary notation.

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

Zero added to one results in one. There are fewer characters in the alphabet so this happens less frequently. Note also that places here are not powers of ten (1, 10, 100) but are instead powers of two (1, 2, 4, 8). When one (2) is added to one (2), the result, 10 (4) cannot be represented in the two's place. So zero is recorded and 10 (4) is carried to the next bit position, the 4's place. This 10 (4) is added to the one (4) and zero already in this place to produce a result 10 (8). The zero remains in the 4's place, and the 10 (8) is carried to the 8's place.

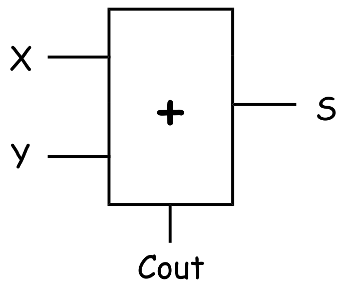
But as before, digital systems may not always have an extra bit position (here in the 8's place) to hold the one carried out of the 4's place. If it's available, 110 (6) added to 011 (3) results in 1001 (9). Otherwise the result is 001 (1), which is incorrect, at least for unsigned integers. Interestingly, in a three bit two's complement representation, 110 (-2) added to 011 (3) *is* 001 (1)! So overflow errors are dependent on the representation. More on this later.

Addition Hardware: The goal is to build hardware to perform arithmetic. It is important to note that the operation of binary addition is invariant to bit position. It's the same operation with respect to given inputs used to compute outputs regardless of whether it is performed in the 1's place, 2's place, etc. So building a one-bit adder is the place to start. Here's an adder for two one-bit binary operands, X and Y. The result is S.



X	Y	S
0	0	0
1	0	1
0	1	1
1	1	10

Again, the rules of addition are applied in binary. 0 + 0 = 0, 1 + 0 = 1, 0 + 1 = 1, 1 + 1 = 10. Only this is a *one-bit* adder. So 10 (2) cannot be represented in this place. An additional output is added to carry this value to the next bit position. This is called Carry Out (Cout). When the sum of X and Y exceeds one, this output signals that the bit position capacity exceeded, and a carry out takes excess output to the next bit position. When X and Y are one, the output is two and carry out transfers this excess. For all other addition cases, the sum can be represented in this bit position so carry out is zero.

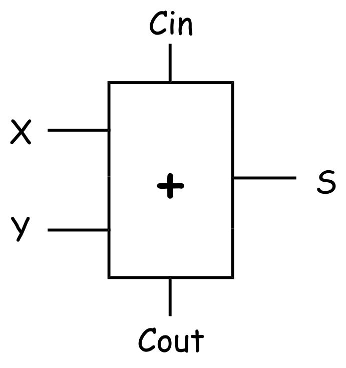


X	Y	Cout	S
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

This output is drawn on the bottom rather than the left side of the adder icon because it is used as an input for another one-bit adder. Since the next bit position must process this carry out signal, the adder also needs another input.

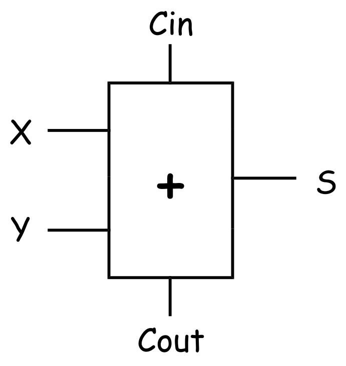
Two for You is One for Me: When a two in one bit position is carry into the next, what is it worth? Moving from least significant to most significant bit positions, each bit is twice the significance of the bit before it. So the two's place is twice the significance of the one's place. The four's place is twice the two's place. The eight's place is twice the four's place. In general, the i+1's place is twice the significance of i's place. So when two is carried out of the i's place, it becomes one in the i+1's place. So a carry out (two) from the lesser significant neighboring bit becomes one as a carry in.

This simplifies the behavior when a *Carry In* (Cin) is added. Now, rather than adding X and Y, the adder is adding X + Y + Cin. All three inputs have the same significance. Here's the behavior assuming the carry in is zero (from before).



X	Y	Cin	Cout	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1		
1	0	1		
0	1	1		
1	1	1		

To handle the new cases where carry in is one, the result includes the sum of X, Y, and Cin.



X	Y	Cin	Cout	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

The first case (X = 0, Y = 0, Cin = 1) produces a sum of one. No carry out required. When (X = 1, Y = 0, Cin = 1), the sum is 10 (2). This results in Sum = 0 and Cout = 1. The same outputs occur when (X = 1, Y = 0, Cin = 1). But when (X = 1, Y = 1, Cin = 1), the sum is 11 (3). Carry out moves 10 (2) to the next bit position. The remaining one becomes the Sum output. So the results are Sum = 1 and Cout = 1.

This one-bit adder, also known as *a Full Adder*, captures the behavior of binary addition. Multi-bit addition can be constructed from cascaded one-bit adders.

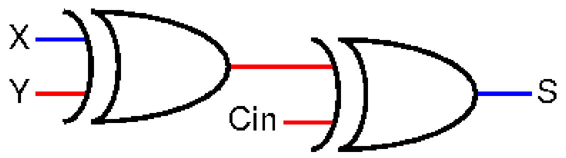
What's Inside a Full Adder?: The behavior of Sum (S) is expressed as odd parity (XOR) which is defined as "true when the number of high inputs is odd". As X, Y, or Cin transitions between zero and one, the number of high inputs changes from even to odd, or odd to even. Independent of the carry out signal, a transition of an input (X, Y, or Cin) results in a transition of the resulting sum, S. Carry out, Cout, has an equally intuitive definition. If there is a one on less than two inputs (X, Y, and Cin), the resulting sum can be handled within the bit position. However if two or more inputs are one, the sum will exceed the bit position's maximum value and carry out must be asserted. This happens in four cases:

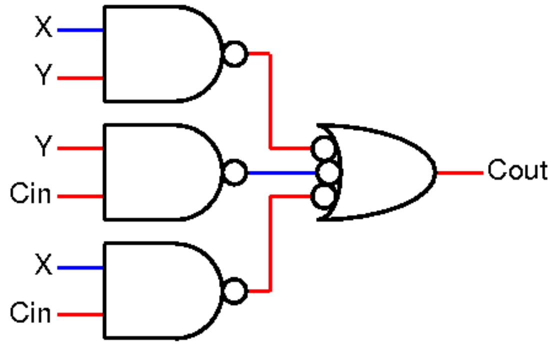
$$X \cdot Y \qquad Y \cdot C_{in} \qquad X \cdot C_{in} \qquad X \cdot Y \cdot C_{in}$$

The expression for carry out can be simplified to a sum of three product terms. The behavior of a full adder can be expressed with these two boolean expressions.

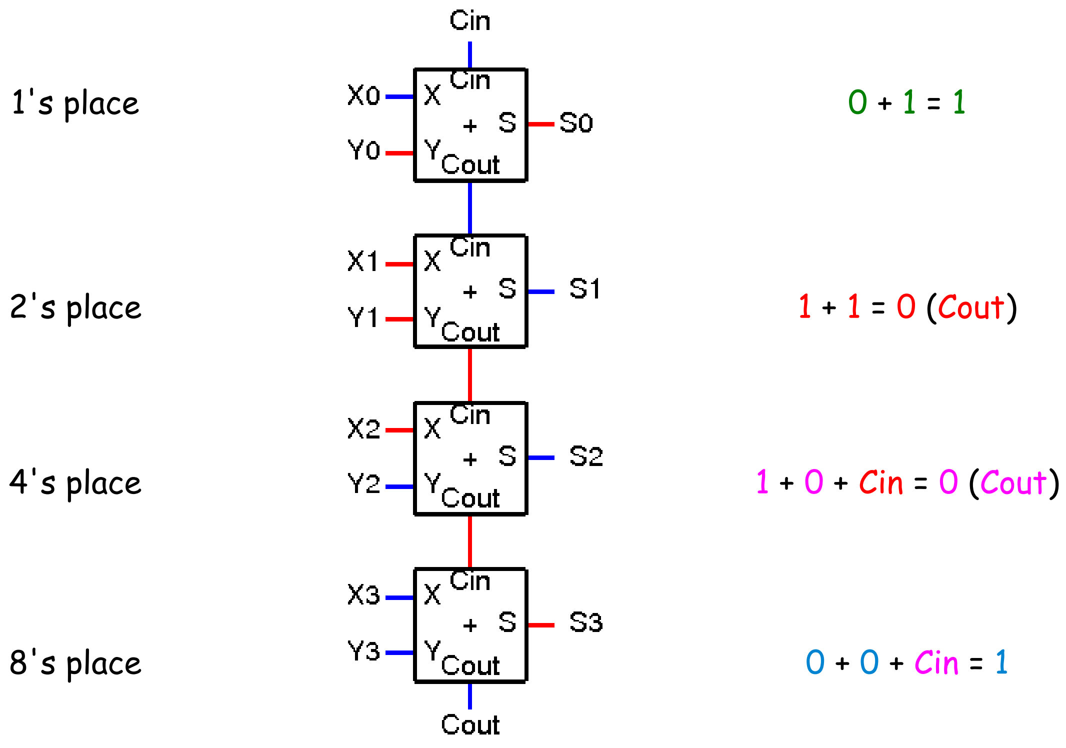
$$S = X \oplus Y \oplus C_{in} \qquad C_{out} = X \cdot Y + X \cdot C_{in} + Y \cdot C_{in}$$

The implementation is straightforward. Odd parity (a checkerboard K-map) is difficult to simplify. The sum of products expression is implemented with NAND gates.





Going Multi-Bit: Since each bit position follows the same operations, a multi-bit adder is created by connecting several replicated one bit adders. The carry in of the least significant bit is set to zero. If it is one, an extra one is added to the sum. *This may come in handy later*. Each bit position is implicit relative to its position. Here's a four bit adder:



In this example X = 0110 (6) and Y = 0011 (3), Each one bit adder handles one bit position. The collection performs the word addition yielding the correct solution 1001 (9) … correct assuming an unsigned integer representation. But what about for other representations?

Fixed Point Arithmetic: A full adder works predictably for unsigned integers. It also supports unsigned fixed point representations, since the bit position relationship does not change. The carry in comes from a bit position that has half the significance. The carry out goes to a bit position that has twice the significance. All that is required is a zero in the carry in to the least significant bit. For all unsigned representations, a carry out of the most significant bit in the representation indicates a result beyond the maximum expressible value. Otherwise the value is correct. If a fixed point is added in the middle of a four bit adder, its operation is unchanged. All that changes is the interpretation of the operands and the result. The operation adds 01.10 (1.5) and 00.11 (0.75) to produce 10.01 (2.25). Here are more examples.

integer		fixed point	
7 + 1 = 8	0111 + 0001 = 1000	01.11 + 00.01 = 10.00	1.75 + 0.25 = 2.0
9 + 6 = 15	1001 + 0110 = 1111	10.01 + 01.10 = 11.11	2.25 + 1.5 = 3.75
4 + 12 = 0 overflow	0100 + 1100 = 0000	01.00 + 11.00 = 00.00	1.0 + 3.0 = 0.0 overflow

In each of these examples, the same operand patterns are applied to the four bit adder, producing the same result. Only the representation differs. Note that for fixed point representations, overflow errors occur (or not) with the same operands, regardless of the position of the point. Fixed point is a scaling of the operands and the corresponding ranges of the representation.

Signed Arithmetic: In Number Systems, there were many advantages to two's complement representations for signed quantities: only one representation of zero, a simple negation, easy differentiation of positive and negative values. Here's one more reason to like two's complement: it employs the same rules for arithmetic. So the examples can be reconsidered as two's complement.

unsigned integer		two's complement integer	
7 + 1 = 8	0111 + 0001 = 1000	0111 + 0001 = 1000	7 + 1 = -8 overflow
9 + 6 = 15	1001 + 0110 = 1111	1001 + 0110 = 1111	-7 + 6 = -1
4 + 12 = 0 overflow	0100 + 1100 = 0000	0100 + 1100 = 0000	4 + -4 = 0.0

In both representations, the same four bit adder performs the same logical operations producing the correct result when the answer is within the representation's range. Different representations give different meanings to the operand sequences. And different ranges produce overflow errors in different places. The two's complement interpretation of the first example 7 + 1 = -8 results from the range of a four-bit two's complement integer: -8 to +7. This is not an error for the unsigned integer representation with a range of 0 to 15. In the third example, the result in the unsigned integer representation overflows its range. The two's complement representation does not.

The four bit adder also performs properly for signed two's complement fixed point. Since this is just scaling of the values (and ranges), this is expected.

unsigned fixed point		two's complement fixed point	
01.11 + 00.01 = 10.00	1.75 + 0.25 = 2.0	01.11 + 00.01 = 10.00	1.75 + 0.25 = -4.0 overflow
10.01 + 01.10 = 11.11	2.25 + 1.5 = 3.75	10.01 + 01.10 = 11.11	-1.75 + 1.5 = -0.25
01.00 + 11.00 = 00.00	1.0 + 3.0 = 0.0 overflow	01.00 + 11.00 = 00.00	1.0 + -1.0 = 0.0

The carry out of the most significant bit indicates overflow with unsigned representations. It tells nothing about overflow in two's complement representations. So how can these errors be detected?

Overflow in Two's Complement: There are several ways to detect two's complement overflows. The most intuitive exploits the easy sign detection of the representation using only the most significant bit. If the MSB is zero, the value is positive. If the MSB is one, it is negative. When an overflow occurs, the inexpressible *wraps around* the range of the representation and ends up in the opposite signed values. In the first example, two positive numbers are added to produce a negative result. Because overflows wrap into the opposite sign, they can be detected using only the most significant bits of the operands and the result. Here are the eight cases when positive and negative values are added.

0010	0010	0110	0010	1010	1010	1001	1101
+ 0011	+ 0110	+ 1100	+ 1100	+ 0110	+ 0101	+ 1010	+ 1011
0101	1000	0010	1110	0000	1111	0011	1000
ok	overflow	ok	ok	ok	ok	overflow	ok

Overflows occur in two cases: when two positive values are added with a negative result, and when two negative values are added with a positive result. Here are the corresponding decimal values:

2	2	6	2	-6	-6	-7	-3
+ 3	+ 6	+ -4	+ -4	+ 6	+ 5	+ -6	+ -5
5	-8	2	-2	0	-1	3	-8
ok	overflow	ok	ok	ok	ok	overflow	ok

Two's complements overflows for addition occur when two positives produce a negative sum, or two negatives produce a positive sum. This can be expressed as a boolean expression, where the "m" subscript indicates the most significant bit:

$$Overflow = X_m \cdot Y_m \cdot \overline{S_m} + \overline{X_m} \cdot \overline{Y_m} \cdot S_m \quad .$$

The same multi-bit adder can be used for unsigned and signed two's complement representations. But different hardware is required to detect overflows. For unsigned addition, the carry out of the most significant bit indicates an overflow. For signed two's complement, hardware that implements this sum of products is needed. More of this hardware will follow an exploration of the next arithmetic operation, *subtraction*.

Giving and Taking Away: Subtraction is the converse of addition. But in addition, the early-learned rules of elementary school are directly applied, in subtraction, a more modern approach to borrowing is employed. Here's an example.

```
                              1        1 1        1 1
   4 0 5 8     4 0 5 8    4 0 5 8    4 0 5 8    4 0 5 8
 - 2 3 7 3   - 2 3 7 3  - 2 3 7 3  - 2 3 7 3  - 2 3 7 3
 _________   _________  _________  _________  _________
                     5        8 5      6 8 5    1 6 8 5
```

Like addition, one starts at the least significant bit. Here, 3 is subtracted from 8 leaving 5. But in the next digit, the ten's place, 7 is subtracted from 5. Like addition, sometimes the operation cannot be completed within the digit. Early on, one learns to search for the next non-zero digit, borrowing one in its place, and regrouping as needed. Here one would borrow from the 4 (4000). But this approach is expensive, since it requires a search though an indeterminate number of digits. A more efficient approach doesn't look for the needed value to borrow; it presumes it exists, passes a borrow signal to the next digit, and completes the operation. This closely resembles the modern approach to borrowing: one places a purchase on a credit card, and decides later whether needed funds are available. While this is an unsound fiscal policy, it works well in computer arithmetic.

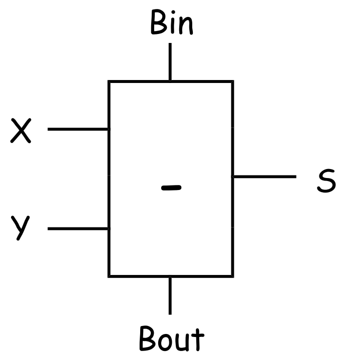
In this example, the process assumes the additional 10 (100) is available to regroup the 5 (50) as 15 (150). Then the 7 (70) is subtracted leaving 8 (80). A borrow out signal is passed to the next digit, the hundred's place. Here, 3 (300) is subtracted from 0. The the borrow in does not add, it subtracts as a 1 (100) in this digit. Again, the total of 4 (400) that is subtracted. So borrow out is asserted to the thousand's place. This provides 10 (1000) that the 4 (400) can be subtracted from. The operation in the thousand's place concludes with 2 (2000) being subtracted along with the borrow value 1 (1000) from the 4 (4000), leaving 1 (1000) remaining.

Binary subtraction follows this decimal example just as binary addition did.

```
                             1          1 1           1 1
  1 0 0 1     1 0 0 1     1 0 0 1     1 0 0 1      1 0 0 1
 - 0 1 1 0   - 0 1 1 0   - 0 1 1 0   - 0 1 1 0    - 0 1 1 0
 ---------   ---------   ---------   ---------    ---------
                     1         1 1       0 1 1      0 0 1 1
```

In this example, barrow out is asserted when the operation cannot be completed with the bit position.

Subtraction Hardware: Subtraction logic resembles a full adder. Only it computes the difference, and it accepts borrow in and generate borrow out. In subtraction, the difference output (D) is computed as X = Y – Bin since both Y and Bin have the same significance. Here's the behavior of a full subtractor.

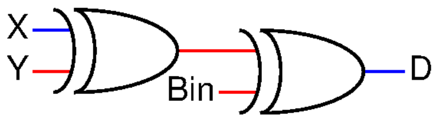


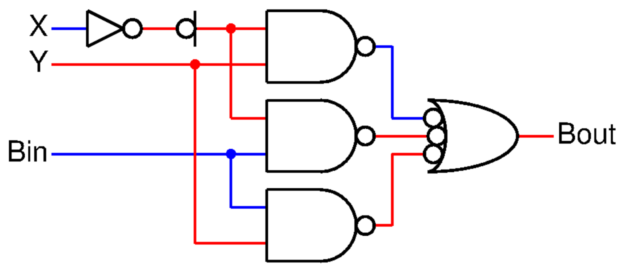
X	Y	Bin	Bout	D
0	0	0	0	0
1	0	0	0	1
0	1	0	1	1
1	1	0	0	0
0	0	1	1	1
1	0	1	0	0
0	1	1	1	0
1	1	1	1	1

It is interesting to note that difference (D) is exactly the same as sum (S) in addition: odd parity. This makes sense when one considers that, from a single bit position's standpoint, adding one has the same effect as subtracting one. The result toggles. The borrow out expression differs from carry out.

$$D = X \oplus Y \oplus B_{in} \qquad B_{out} = \overline{X} \cdot Y + \overline{X} \cdot B_{in} + Y \cdot B_{in}$$

The hardware implementation resembles the full adder, but for a small difference in the borrow out circuit.



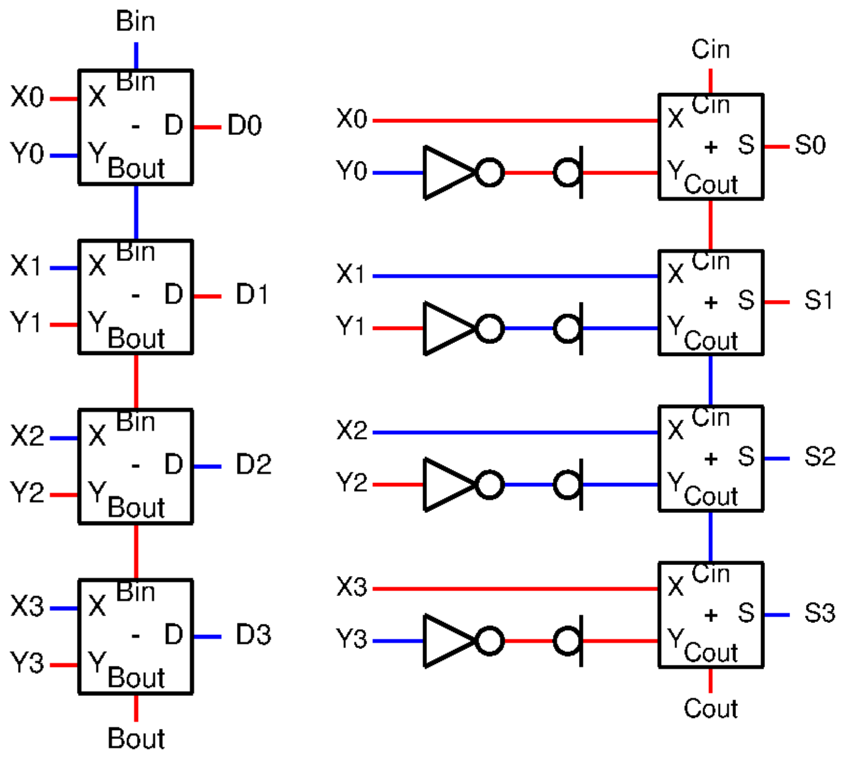


Two for One: Full subtracters can be replicated to form a multiple bit subtractor. But that's not how it's done. The hardware cost for subtraction would match the cost of addition. What if there were a way to get both operations using only one multi-bit arithmetic unit? How can an adder subtract? The answer comes from a little math and the magic of two's complement.

$$Z = X - Y \qquad\qquad Z = X + (-Y)$$

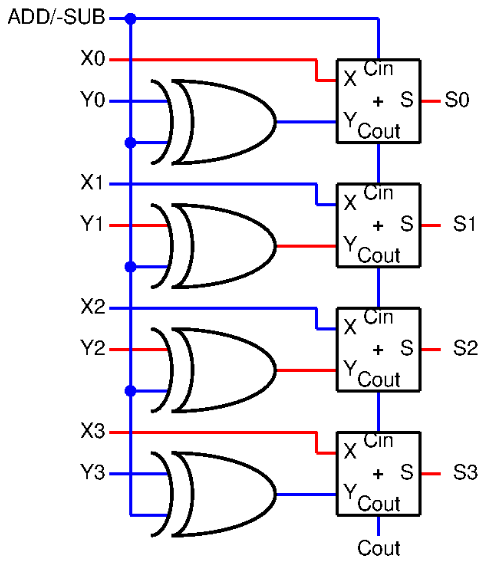
These two expressions compute the same result. So subtraction can be accomplished by adding the negation of the subtracted value. Negation is easy in two's complement. Just invert (complement) each bit and add one. The complement is done with inverters. The added one occurs by setting the carry in of the least significant bit to one. This "feature" was noted in the multi-bit adder.

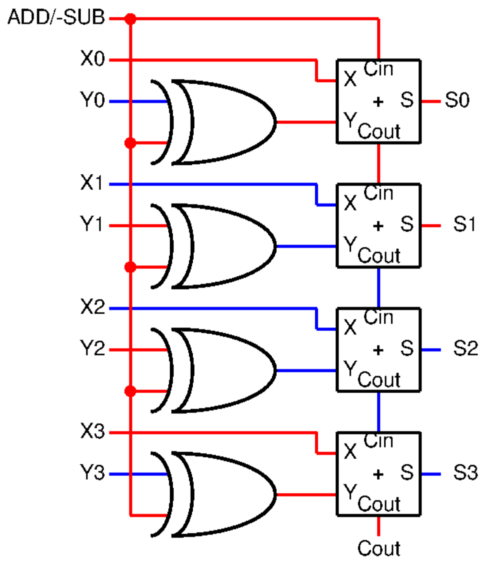
Suppose the desired subtraction is 1001 – 0110. The result, 0011, can be computed using either full subtracters or full adders (with negation logic).



A multi-bit adder/subtracter can employ the adder hardware to perform addition and subtraction. A single control line $\overline{\text{ADD}}$/SUB determines which operation is to be performed. This rather unusual signal label declares that the signal is an active low *add* signal combined with an active high *subtract* signal. Since the signal is either low (zero) or high (one), the circuit is either adding or subtracting. To construct this, the inverters are replaced with selective inverters introduced in

Building Blocks, implemented as XOR gates. The $\overline{\text{ADD}}$/SUB signal also controls the carry in of the least significant bit of the adder. When high, it adds the one needed to negate the second operand being subtracted. The adder/subtractor implementation is shown below in both operation modes.





What About Overflows?: The signed overflow detection logic handles two's complement representations with this design. For unsigned, the carry out indicates addition overflow with a high value (One). Subtraction overflows are represented with a low value (zero) on carry out. The negation of the subtracted value wraps non-overflowing results into the unsigned overflow domain. Ironically, subtracted numbers larger than the first operand become small in negation and do not reach the unsigned overflow domain.

Summary:

- Addition and subtraction follow the rules learned for decimal, but with a few small changes. One does not assume unlimited digits. And subtraction uses borrowing on credit to simplify protracted borrowing.
- Binary addition and subtraction are defined bit-wise, leading to the definition and construction of a full adder and a full subtractor. They are implemented using XOR and NAND gates.
- Overflows occur due to a bound word size. Error detection is straightforward and differs for unsigned and signed representations.
- An adder/subtractor can be constructed using a multi-bit adder and selectable negation circuitry for the second operand.