INF2270 — Spring 2011

Lecture 4: Signed Binaries and Arithmetic

UNIVERSITETET I OSLO

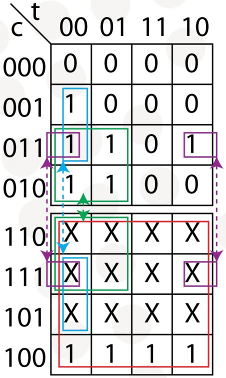
content

content

UNIVERSITETET I OSLO

Karnaugh maps with X's

With 3 variables along one edge a Karnaugh map needs to be folded into 3-dimensions and one has to look for cubes instead of rectangels.
X's can arbitrarily be assigned a '0' or a '1' and can thus be used to extend cubes.



c \ t	00	01	11	10
000	0	0	0	0
001	1	0	0	0
011	1	1	0	1
010	1	1	0	0
110	X	X	X	X
111	X	X	X	X
101	X	X	X	X
100	1	1	1	1

$$
\begin{aligned}
h &= (c_2) \\
&\vee (\bar{t}_1 \wedge c_1) \\
&\vee (\bar{t}_0 \wedge \bar{t}_1 \wedge c_0) \\
&\vee (\bar{t}_0 \wedge c_0 \wedge c_1)
\end{aligned}
$$

content

Half Adder

Example:

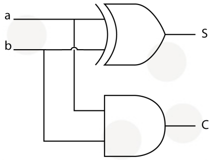
```
     0001
+    0011
---------
=    0100
```

Truth table for a 1-bit half adder:

a	b	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

S is the result and C is the carry bit, i.e. a bit indicating if there is an overflow and an additional bit is necessary to represent the result.

Schematics:



Full Adder (1/2)

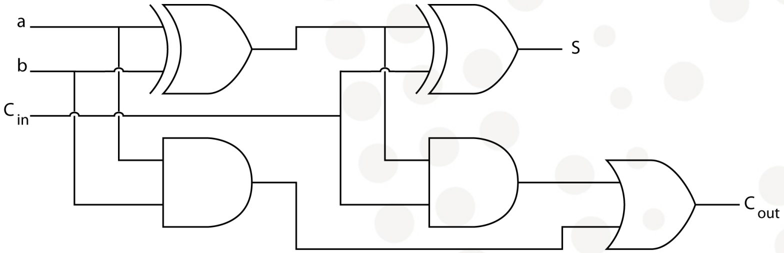
A half adder cannot be cascaded to a binary addition of an arbitrary bit-length since there is no carry input. An extension of the circuit is needed.

Full Adder truth table:

C_{in}	a	b	S	C_{out}
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 (2/2)

Schematics:



content

Sign and Magnitude representation

Maybe the most obvious way of representing positive and negative binary numbers is to add a sign-bit. Example for 8 bit numbers (7-bit magnitude and 1 sign-bit, -127 to +127):

$$\begin{aligned} 87 &= 01010111 \\ -87 &= 11010111 \end{aligned}$$

A problem here is that there is also a 'signed zero', i.e. +0 and -0, which does not really make sense.

Two’s complement Representation

The two’s complement (used in most digital circuits today) is a signed binary number representation that does not have this problem and comes with a number of other convenient properties. In 8-bit two’s complement the unsigned numbers 0 to 127 represent themselves, whereas the unsigned numbers 128 to 255 represent the numbers -128 to -1 (=the unsigned number -256). Thus, also in this representation all numbers with the first bit equal to ’1’ are negative numbers.

87	=	01010111	
-41	=	11010111	(= 215-256)
-87	=	10101001	(= 169-256)

Inverting in Two's Complement

Finding the inverse of a number in two's complement is simple. The same operation is performed for both, positive to negative and negative to positive:

1. invert each bit
2. add 1

Example:

1.	87=01010111	→	10101000
2.	10101000+1	=	10101001=-87
1.	-87=10101001	→	01010110
2.	01010110+1	=	01010111=87

content

Binary Subtraction

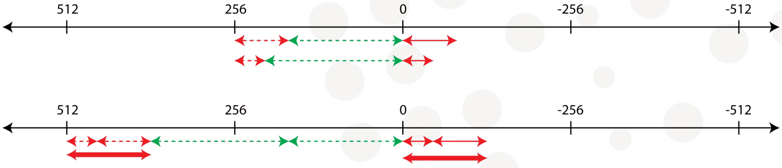
A really cool property of the two’s complement representation is that one can simply add two positive or two negative or a negative and a positive number with a full-adder of the same length, ignore an eventual overflowing carry, and the result *will be correct* (provided the result is not bigger than the positive maximum (e.g. 127 with 8 bit numbers) and no smaller than the negative minimum(e.g. -128 with 8 bit numbers)). Note that ’ignoring an overflowing carry’ in effect is a modulo operation of the result. For example adding two unsigned 8 bit numbers and ignoring an eventual overflow carry (which would mean that the unsigned result is bigger than 255) is performing a modulo 256 operation on the result.

Two’s Complement Addition/Subtraction 8-bit Examples

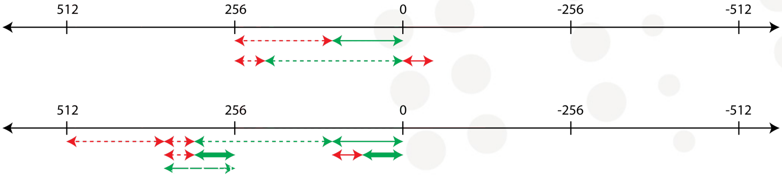
signed op	equiv. unsigned op	mod 256	signed res
-41-87	215+169 = 384	128	-128
87-41	87+215 = 302	46	46

Two's Complement Addition/Subtraction 8-bit Graphically

Sum of two negative numbers



Sum of a negative and positive number



Two’s Complement Subtraction

Thus, if negative and positive numbers can simply be added, a subtraction can be performed by inverting the number that is to be subtracted and adding them.
Thus, to compute a-b:

1. invert b by inverting every single bit
2. add the two numbers and set the carry in signal for the adder to 1 (in order to complete the inversion of b)

Arithmetic Right-Shift

An *arithmetic right-shift* is a shift operation that performs a division by two correctly in the two's complement representation. Obviously a so called *logic shift* that shifts in a '0' from the left would turn a negative number into a positive one, which cannot be correct. Instead, the bit that is shifted in from the left needs to be the former most significant bit (MSB). Note that the result is rounded towards $-\infty$ and not towards zero.

Examples:

decimal	binary	shifted	decimal
-3	1101	1110	-2
-88	10101000	11010100	-44

Extending the bit-length in two’s complement

A last note that will come in handy for the mandatory exercise: To extend the number of bits with which to represent a signed integer, the additional bits on the left need to be filled in with the formar MSB.

Examples:

decimal	4 bit		8 bit
-2	1110	→	11111110
-5	1011	→	11111011
5	0101	→	00000101