10. ALU - Arithmetic and Logical Unit

With the present knowledge we will construct an ALU that can perform a limited number of functions.

10.1 Binary addition

$$a_k 2^k + b_k 2^k = (a_k + b_k) 2^k$$

a_k	b_k	S	
0	0	0	
0	1	1	
1	0	1	
1	1	1	+ 1 carry ($2 \cdot 2^k = 2^{k+1}$)

The "1 carry" means that there must be a 1 from a neighbouring addition, i.e. from $a_{k-1}2^{k-1} + b_{k-1}2^{k-1}$:

c_{k-1}	a_k	b_k	S_k	C_k
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

a_k

S_k:

0	1	0	1	
1	0	1	0	c_{k-1}

b_k

$$S_k = a_k \oplus b_k \oplus c_{k-1}$$

a_k

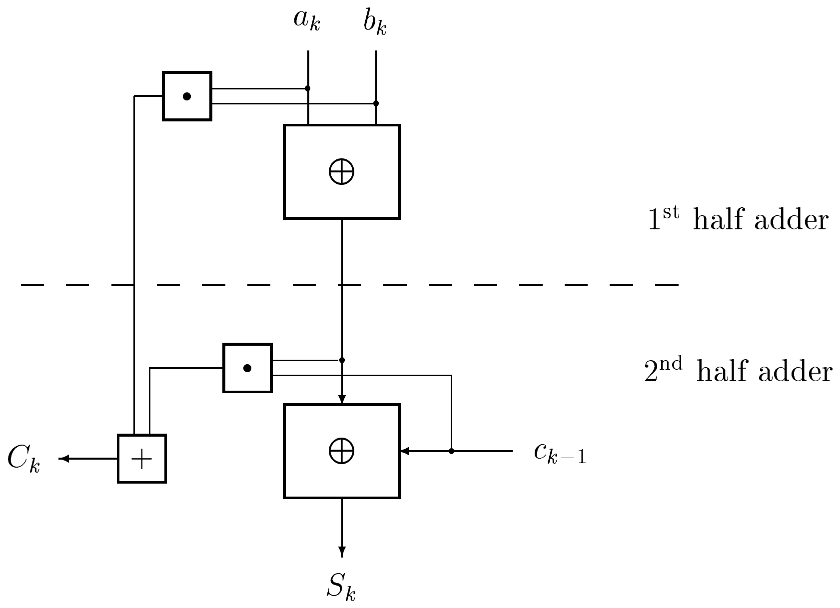
C_k:

0	0	1	0	
0	1	1	1	c_{k-1}

b_k

$$C_k = a_k b_k + a_k c_{k-1} + b_k c_{k-1}$$

From this, we could directly construct a circuit, using these Karnaugh maps, but usually an adder is constructed by connecting two half adders:



The function table can be divided into two parts:

a_k	b_k	S_k	C_k
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

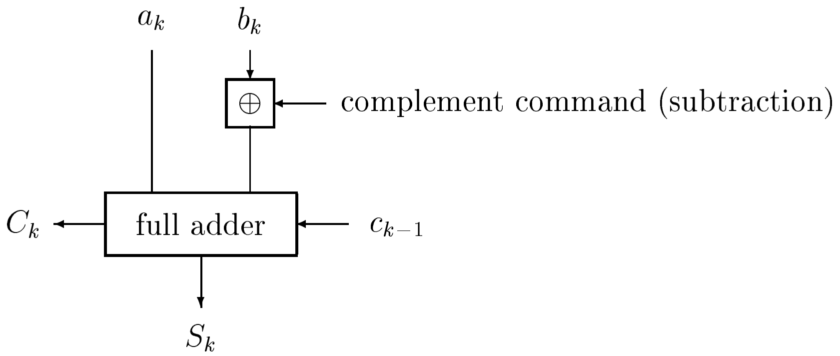
$S_k = a_k \oplus b_k$ $\quad = $ sub-sum
$C_k = a_k \cdot b_k$

S_k	c_{k-1}	S_k	C_k
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$S_k = S_k \oplus c_{k-1}$ $\quad = $ output sum
$C_k = S_k \cdot c_{k-1}$

Both circuits are identical; they are called “half adder”.
Together with an OR gate they form a “full adder”.

- If we want to compute the <u>difference</u> between a_k and b_k (subtraction), we need to take the complement of b_k. A simple way is to use an extra exclusive-OR:



The adder needs a 1 on the least significant bit because the "2-complement code" equals the "1-complement code + 1."

10.2 High-speed adders

In chaining single-bit full adders together in order to sum words of n bits, the addition of the corresponding bits is fast but a carry generated at the least significant bits may cause a change up to the most significant bits. This is called *ripple-through* or *carry propagation.* To make the addition faster, we can compute the input carry at bit k directly from all inputs of preceeding stages $k-1$, $k-2$, ... Such adders are called carry-lookahead adders.

To this purpose we compute the carry C_k at stage k by using two other functions: g_k = carry generate signal and p_k = carry propagate signal.

So for stage k these are defined as $g_k = a_k b_k$ and $p_k = a_k + b_k$.
The carry signal to be propagated to the next stage $k+1$ is defined by $C_k = g_k + p_k c_{k-1}$.
In other words: C_k is a combination of the carry generated at stage k and that propagated from stage $k-1$.
Substituting g_k and p_k in C_k, we obtain $C_k = a_k b_k + (a_k + b_k)c_{k-1} = a_k b_k + a_k c_{k-1} + b_k c_{k-1}$, which equals the expression obtained from the second Karnaugh diagram above.

Similarly, c_{k-1} can be expressed in terms of g_{k-1}, p_{k-1} and c_{k-2}:

$$c_{k-1} = g_{k-1} + p_{k-1}c_{k-2}$$

Substituting this in the previous equation, we get

$$C_k = g_k + p_k g_{k-1} + p_k p_{k-1} C_{k-2}$$

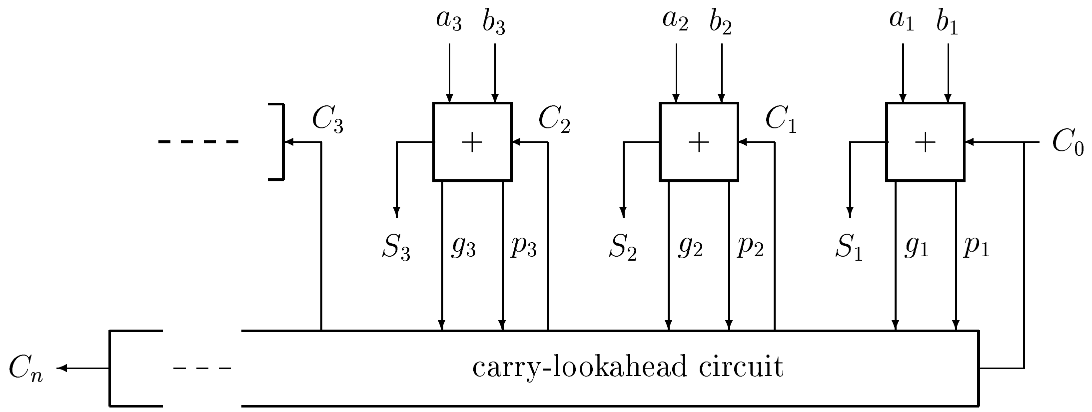
This can be repeated, so

$$C_1 = g_1 + p_1 C_0$$

$$C_2 = g_2 + p_2 g_1 + p_2 p_1 C_0$$

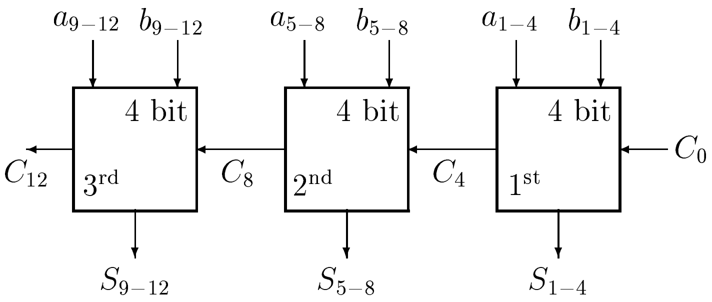
$$C_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 C_0$$

$$C_4 = g_4 + p_4 g_3 + p_4 p_3 g_2 + p_4 p_3 p_2 g_1 + p_4 p_3 p_2 p_1 C_0$$



The lookahead circuit is a two-level circuit consisting of ANDs and ORs. If the delay of a gate is d, the total delay of the carry lookahead is $\boxed{3d}$ (d for g and p plus AND plus OR).

The number of required gate inputs increases with n, the word length, so for practical reasons the word length is limited. For larger word lengths a number of limited carry-lookahead adders is then used:



4-bit
carry-lookahead
adders

For this 12-bit adder with group carry-lookahead and ripple carry between groups the delay is

$$\left.\begin{array}{rcl} 1^{\text{st}} & : & 3d \\ 2^{\text{nd}} & : & 2d \ \ (p'\text{s and } g'\text{s already available, no delay}) \\ 3^{\text{rd}} & : & 2d \end{array}\right\} \boxed{7d}$$

For a ripple carry adder of 12 bits (12 full adders) this is in the worst case $12 \times 2 = \boxed{24d,}$ and 0 when no carry's occur. Hence, on the average the delay will be $12d$.

$$\left.\begin{array}{ll} \text{Full carry} & 3d \\ \text{Group carry} & 7d \\ \text{Ripple carry} & 24d \end{array}\right\} \begin{array}{c}\text{worst}\\ \text{case}\end{array}$$

10.3 Overflow

For a 2-complement adder we need an overflow detection. How can this be achieved? Suppose

$$\begin{array}{rcl} S_A & = & \text{sign of A} \\ S_B & = & \text{sign of B} \\ S_R & = & \text{sign of result} \\ C & = & \text{carry out} \end{array}$$

All bits (variables) are at the following positions:

$$\begin{array}{c|c|ccc} & S_A & \times & \times & \dots \\ & S_B & \times & \times & \dots \\ \hline C & S_R & \times & \times & \dots \end{array}$$

in which the $\times$ bits can be 0 or 1.

- First adding two positive numbers:

  If we look at one specific combination

  $S_A = S_B = 0$

$$\begin{array}{c|c|c} & 0 & 1 \\ & 0 & 1 \\ \hline 0 & 1 & 0 \end{array}$$

  we see that the most significant bits have caused a change of the sign bit. This is impossible because the sum of two positive numbers must always be positive. Hence, OV=1!

S_A	S_B	S_R	C	OV		
0	0	0	0	0	$\longleftarrow$	new sign=0, no overflow
0	0	1	0	1	$\longleftarrow$	$S_R = 1$, so there must be an overflow because $S_A + S_B = 0$ is required

  The other combinations with $S_A = S_B = 0$ cannot occur and C can never be 1, so <u>0001 and 0011 are don't cares</u>.

- Now A positive and B negative:

S_A	S_B	S_R	C	OV				
0	1	0	1	0	$\longleftarrow$	$A \geq \lvert B \rvert$	$\rightarrow$	S_R becomes 0, but this can only be if a one is added and then we get $S_B + 1 = C$ (because $S_B = 1$)
0	1	1	0	0	$\longleftarrow$	$A > B$	$\rightarrow$	we get a negative number, sign remains 1, no carry

  0100 cannot occur (for $S_R = 0$, adding a 1 is required so $C = 1$).
  0111 cannot occur (S_B would have been added a 2; this is impossible because the carry from the adders to the right can only be ≤ 1).

S_A	S_B	S_R	C	OV	
1	0	1	0	0	$\lvert A \rvert > \lvert B \rvert$
1	0	0	1	0	$\lvert A \rvert \leq \lvert B \rvert$

  How about don't cares here?

- For two negative numbers:

S_A	S_B	S_R	C	OV		
1	1	1	1	0	$\longleftarrow$	the sum is negative and delivers a carry because $S_A + S_B$ result in a carry
1	1	0	1	1	$\longleftarrow$	now there is no carry from the adders to the right, but this implies that there is an overflow: sum=positive

  The don't cares are 1100 and 1110, but the latter is impossible because $S_A = S_B = 1$ gave already a carry.

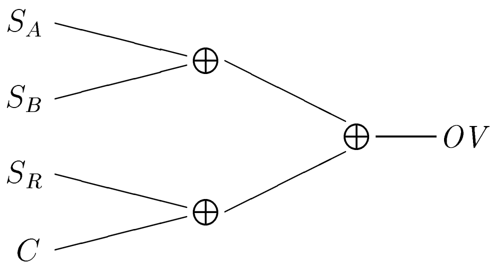
- Table for addition (without don't cares) and two solutions:

S_A	S_B	S_R	C	OV
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	0
1	0	1	0	0
1	0	0	1	0
1	1	1	1	0
1	1	0	1	1

	00	01	11	10
00	0	×	×	×
01	×	0	1	0
11	×	×	0	×
10	1	0	×	0

S_B, C, S_R, S_A

$OV = S_A S_B \bar{S}_R + \bar{S}_A \bar{S}_B S_R$



	1		1
1		1	
	1		1
1		1	

$OV = S_A \oplus S_B \oplus S_R \oplus C$

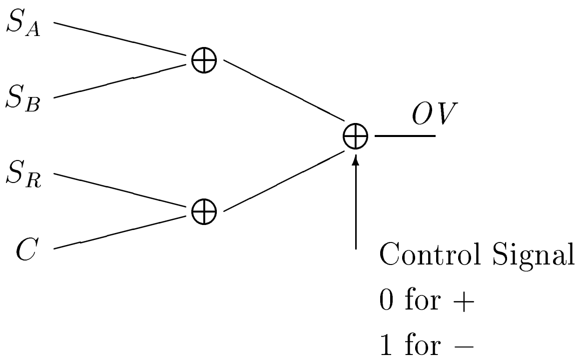
Subtraction, i.e. $A - B$, is obtained by inverting B and adding a 1 to the least significant bit. We obtain a similar table (*try this!*) where the S_B terms must be inverted.

S_A	S_B	S_R	C	OV	
0	0	0	1	0	$A \geq B$ } (B = positive)
0	0	1	0	0	$A < B$
0	1	0	0	0	$A + B < 2^{n-1}$
0	1	1	0	1	$A + B > 2^{n-1}$
1	0	1	1	0	
1	0	0	1	1	
1	1	1	0	0	
1	1	0	1	0	

<u>However</u>, this is the table for addition, but now with $\bar{S}_B$ instead of S_B, so

$$OV = S_A \oplus \bar{S}_B \oplus S_R \oplus C = S_A \oplus S_B \oplus S_R \oplus C \oplus 1$$

The circuit becomes:



(which must also generate an <u>input carry</u> for the least significant bit, 2-complement!)

10.4 Parts of a simple μprocessor ALU with + and −