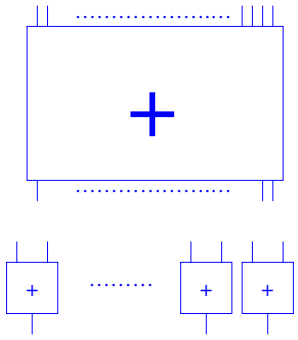
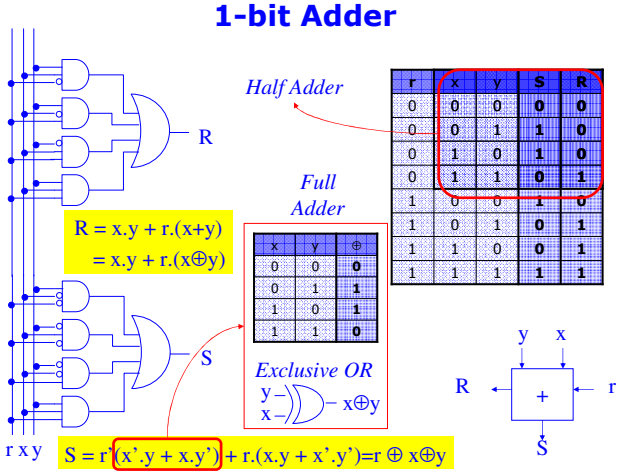
Arithmetic and Logic Unit (ALU)

Designing an Adder

- Traditional circuit design: truth table approach
- Cost/Speed tradeoff:
  - n-bit adder: truth table with 2n inputs, 2^{2n} rows → fast circuit (theoretically), but very costly
  - n 1-bit adders: cheap but slow
  - Tradeoff: n 1-bit adders and additional circuitry to speed up computation



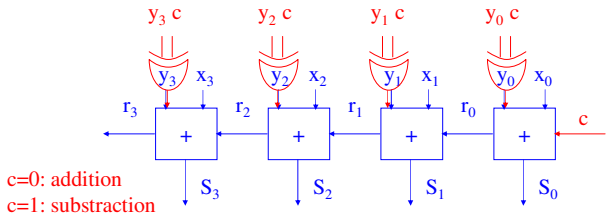
1-bit Adder



r	x	y	S	R
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

x	y	⊕
0	0	0
0	1	1
1	0	1
1	1	0

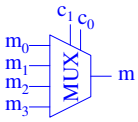
n-bit Addition/Substraction



- Substraction:
  - X-Y = X + (-Y) = X + Y' + 1
  - c = 1 → no additional adder
- Bottleneck: carry propagation

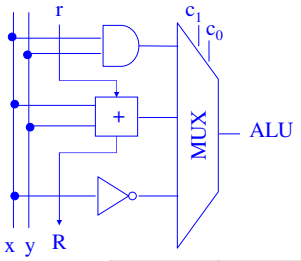
	r_3	r_2	r_1	r_0
X		x_3	x_2	x_1	x_0
Y	+	y_3	y_2	y_1	y_0
Z	**r_3**	**s_3**	**s_2**	**s_1**	**s_0**

A Simple 1-bit ALU



Multiplexor:
select among several inputs

c_1	c_0	m
0	0	m_0
0	1	m_1
1	0	m_2
1	1	m_3



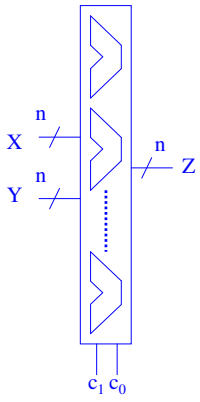
c_1	c_0	ALU
0	0	ADD
0	1	AND
1	0	NOT
1	1	*d*

- Elementary operations:
  - ADD, AND, NOT
- ALU: Arithmetic and Logic Unit

n-bits ALU

- n 1-bit ALUs
- Embryo of instruction set:

c_1	c_0	ALU
0	0	ADD
0	1	AND
1	0	NOT
1	1	*d*



Overflow

		1	0	0	
6		0	1	1	0
5	+	0	1	0	1
-5		**1**	**0**	**1**	**1**

	1	0	0	0	
-4		1	1	0	0
-7	+	1	0	0	1
5	1	**0**	**1**	**0**	**1**

- Detect overflow in twos-complement ?

Overflow

- Overflow signal $S_{overflow}$ (1=overflow)
- Overflow in twos-complement:
  - $x \leq 0$, $y \geq 0 \rightarrow$ no overflow possible
  - $x \geq 0$, $y \geq 0$:
    - Overflow: Z=X+Y (twos-complement)
      Z > 2^{n-1}-1, et $0 \leq X \leq 2^{n-1}-1$, $0 \leq Y \leq 2^{n-1}-1$
      $\Rightarrow 2^{n-1}-1 < Z \leq 2^n-2$
      $\Rightarrow$ In twos-complement, negative numbers coded in $[2^{n-1};2^n-1]$
      $\Rightarrow$ Z is negative
  - $x \leq 0$, $y \leq 0$: same; in case of overflow, Z is positive

  $\rightarrow$ overflow detection criterion

z_{n-1}	x_{n-1}	y_{n-1}	$S_{overflow}$
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**0**
1	1	0	**0**
1	1	1	**0**

$$S_{overflow} = x_{n-1}.y_{n-1}.\overline{z_{n-1}} + \overline{x_{n-1}}.\overline{y_{n-1}}.z_{n-1}$$

Overflow

- Action upon overflow ?
- Several solutions:
  - Stop program (*TRAP*)
    - Example: MIPS R3000;
  - Raise a signaling bit such as $S_{overflow}$
    - Example: Intel x86, Sun SPARC;
  - Do nothing
- Example: using C on a Sun SPARC 32-bits
  -2147483648 (-2^{31}) $\rightarrow$ 2147483647 ($2^{31}-1$)

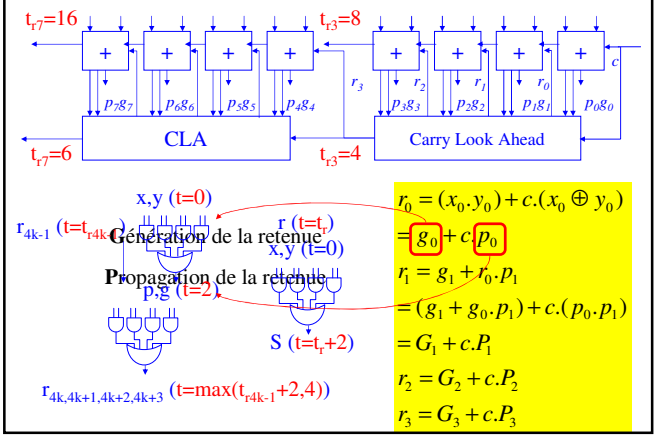
```
int a=2000000000;
int b=2000000000;
int c;

c=a+b;
printf("%d\n",c);
```

⇩

```
-294967296
```

0 1110 1110 0110 1010 1100 1010 0000 0000
+ 0 1110 1110 0110 1010 1100 1010 0000 0000
---
1 1101 1100 1101 0110 0101 0000 0000 0000

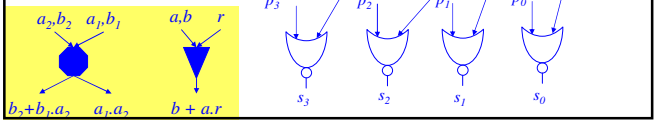
Speeding Up Carry Propagation



$$r_0 = (x_0.y_0) + c.(x_0 \oplus y_0)$$
$$= g_0 + c.p_0$$
$$r_1 = g_1 + r_0.p_1$$
$$= (g_1 + g_0.p_1) + c.(p_0.p_1)$$
$$= G_1 + c.P_1$$
$$r_2 = G_2 + c.P_2$$
$$r_3 = G_3 + c.P_3$$

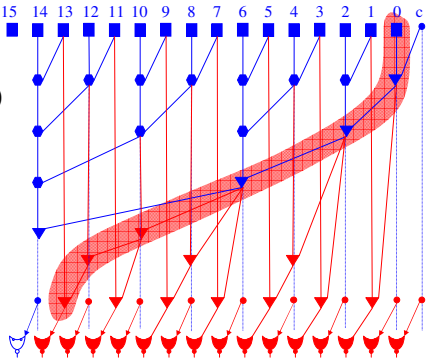
Speeding Up Carry Propagation

- n = number of bits
- CLA: $O(n)$.
- Tree structure: $O(log_2 n)$.
- $p_i = x_i \oplus y_i$
- $g_i = x_i.y_i$
- $s_i = p_i \oplus r_{i-1}$



Speeding Up Carry Propagation

- n=16.
- Brent-Kung.
- CLA: 5 steps (*pg* and 4 CLA) to propagate carry
- Tree: 7 steps (*pg* and 6 *PG* operators)
- Starting with 32 bits, tree (8) faster than CLA (9)



Speeding Up Carry Propagation

- n=16
- Han-Carlson.
- Cost/Performance tradeoff
- Itanium:
  - 64 bits,
  - 0,18μm,
  - 482 ps for one addition