Adders, ALUs, etc.

- In a digital circuit, there is often a need to perform arithmetic computations: addition, subtraction, multiplication, etc.
- Depending on the available functional units in the module library being used, the designer may have one or more alternatives:
  - Dedicated functional units:
    - adder
    - subtractor
    - multiplier
  - Multi-function functional units:
    - adder / subtractor
  - More general functional units = Arithmetic Logic Units (ALUs):
    - ALU (addition, subtraction, multiplication)
    - ALU (addition, subtraction, multiplication, division, comparison)

Half Adder

- Consider what happens when you add two binary digits:

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

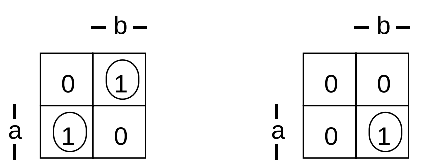
(In "0) 1": the left digit is the carry out, the right digit is the sum.)

- We can construct a truth table for a *half adder* (HA) — a device that adds two binary digits *a* and *b*, producing a *sum* and a *carry out*

a	b	sum	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

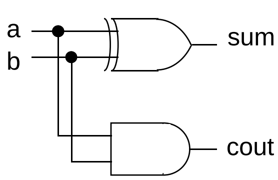
Implementing a Half Adder

- A half adder can be implemented directly in 2-level SOP form:



sum = ab' + a'b   cout = ab

- A half adder can also be implemented using an **xor** gate:



- Which implementation is better? Why?

Which Implementation of the Half Adder is Better?

- Given this number of transistors and amount of propagation delay (in ns)

	2-input		3-input		4-input	
and	6	2.4	8	2.8	10	3.2
or	6	2.4	8	2.8	10	3.2
xor	14	4.2				
nand	4	1.4	8	1.8	10	2.2
nor	4	1.4	8	1.8	10.	2.2

inverter (1-input) 2 1.0

- In SOP form, using and and or gates:
  - sum = ___ transistors, cout = ___ transistors, total ___
  - Max delay until outputs are ready = ___ns
- In SOP form, using nand gates:
- In xor form (using a single complex gate instead of three simple gates):

Full Adder

- A half adder is fine for the *least significant bit* (LSB) of a multi-bit number, but not the other bits

		carry in ↓ 1	1
	1	1	1
00	01	01	11
+01	+01	+11	+11
0) 01	0) 10	1) 00	1) 10

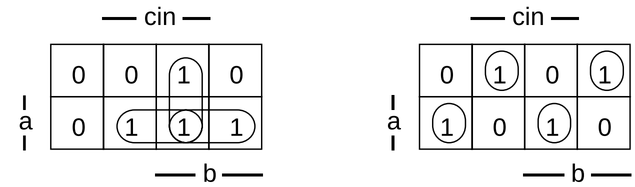
(in "1) 00": carry out ↑ 1, sum ↑ 00)

- A *full adder* (FA) adds two binary digits *a* and *b*, plus a *carry in* (from the previous digit), producing a *sum* and a *carry out*

a	b	cin	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder (cont.)

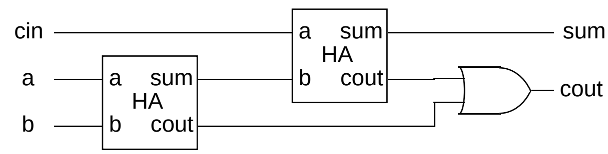
- A full adder can be implemented directly in 2-level SOP form:



cout = a•cin + b•cin + ab

sum = ab'cin' + a'b'cin + abcin + a'bcin'

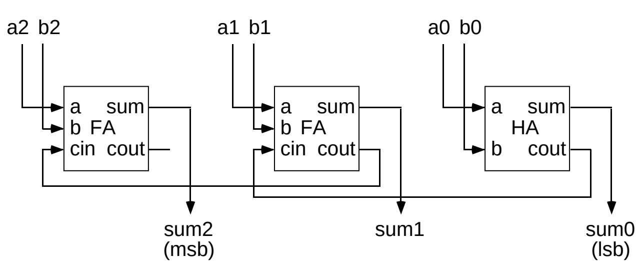
- A full adder can also be implemented by two half adders:



- Again, which implementation is better, and why? Various alternatives left as an exercise for anyone interested…

n-Bit Adder

- An *n*-bit adder can now be constructed out of *n*-1 full adders, and 1 half adder



- This kind of *n*-bit adder is called a *ripple adder*
  - Why?
  - What are its limitations?

Half Subtractor

- Consider what happens when you subtract two binary digits:

0	0	1	1
–0	–1	–0	–1
)	)	)	)

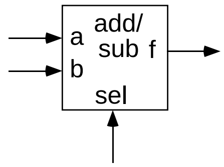
(borrow ↗ before ")", difference ↖ after ")")

- Using the same techniques that we used to construct a half adder, we can construct a *half subtractor* — a device that subtracts binary digit *b* from binary digit *a*, producing a *difference* and a *borrow*

ALU

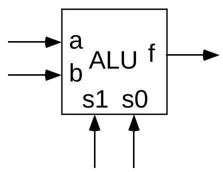
- Just as we can build an adder or a subtractor, we could build an adder / subtractor…

sel	f
0	a+b
1	a–b



- …or a more general Arithmetic Logic Unit (ALU) capable of performing a number of arithmetic functions

s1	s0	f
0	0	a+b
0	1	a–b
1	0	a•b
1	1	a&b



- And we can generalize this idea to construct an *n*-bit ALU capable of performing whatever functions we want