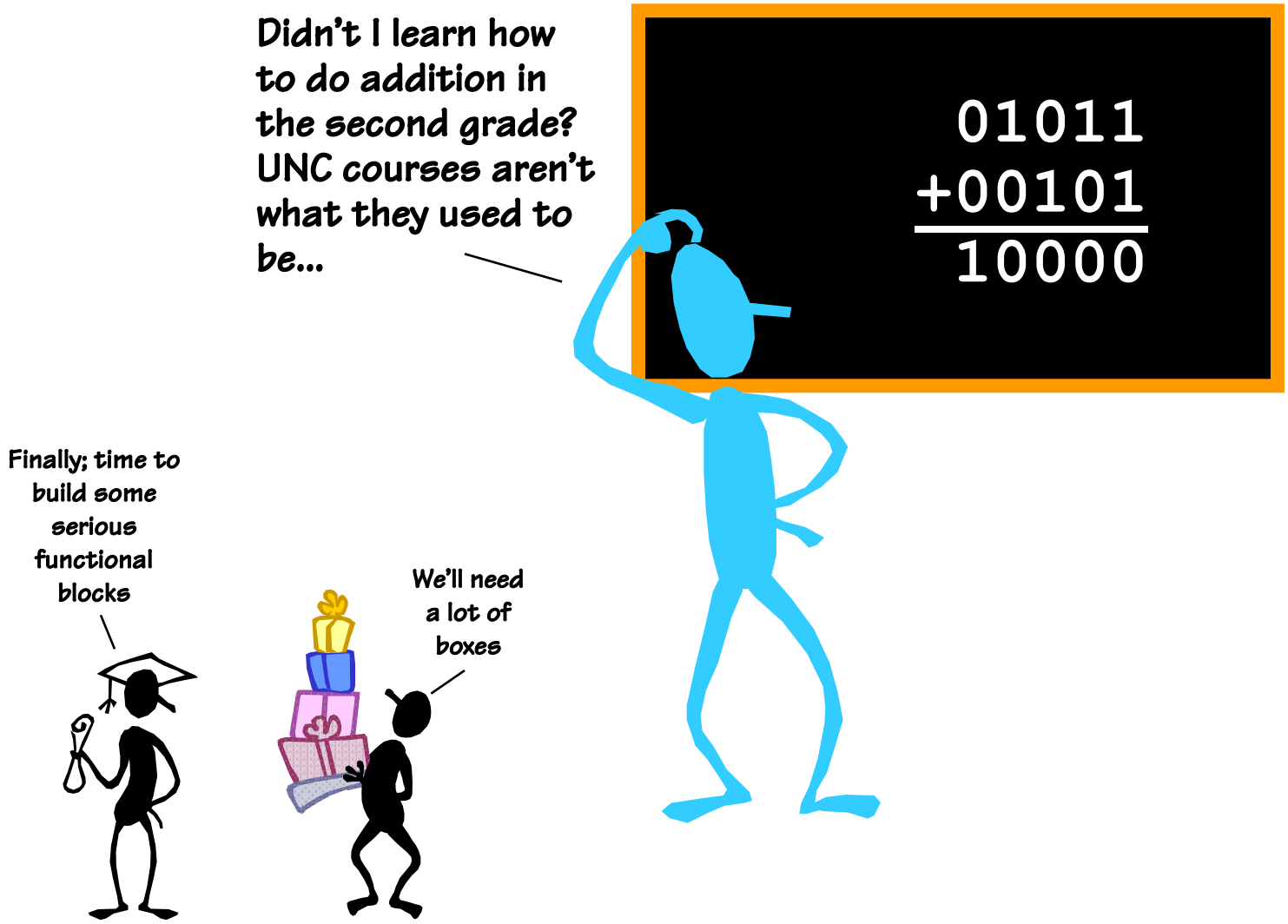
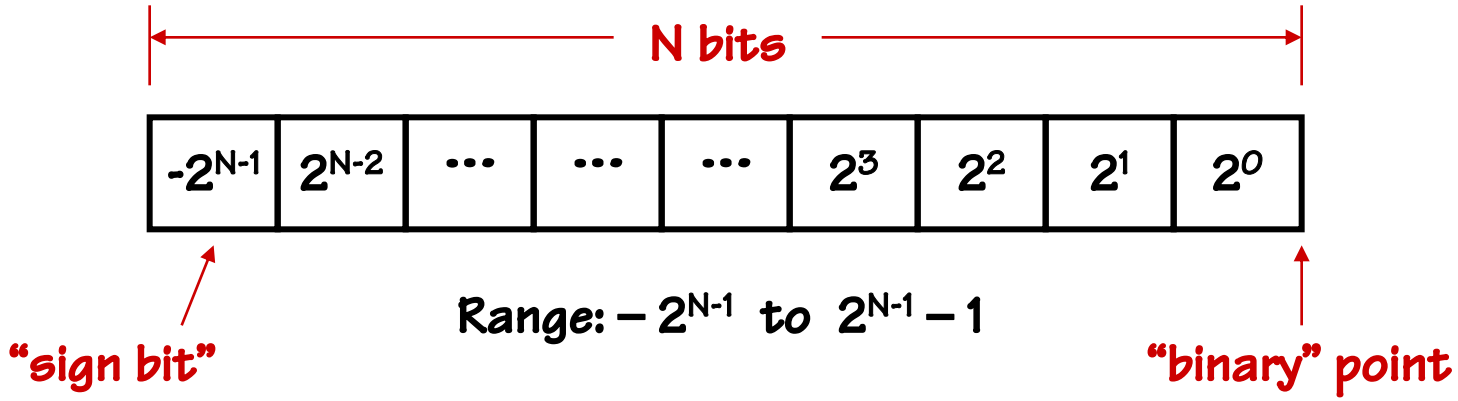
Arithmetic Circuits



Review: 2's Complement



8-bit 2's complement example:

$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

If we use a two's-complement representation for signed integers, the same binary addition procedure will work for adding both signed and unsigned numbers.

By moving the implicit "binary" point, we can represent fractions too:

$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625$

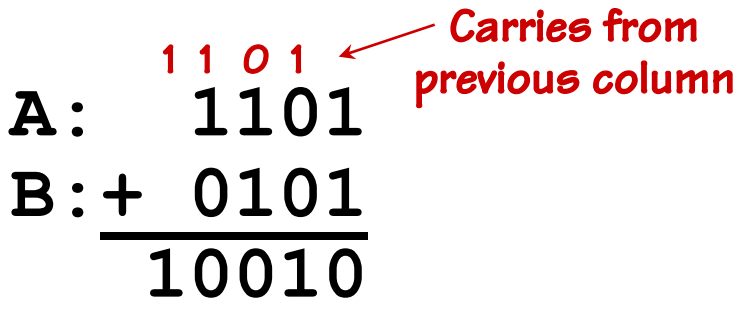
Binary Addition

Here's an example of binary addition as one might do it by "hand":

```
A:  1101
B:+ 0101
   10010
```

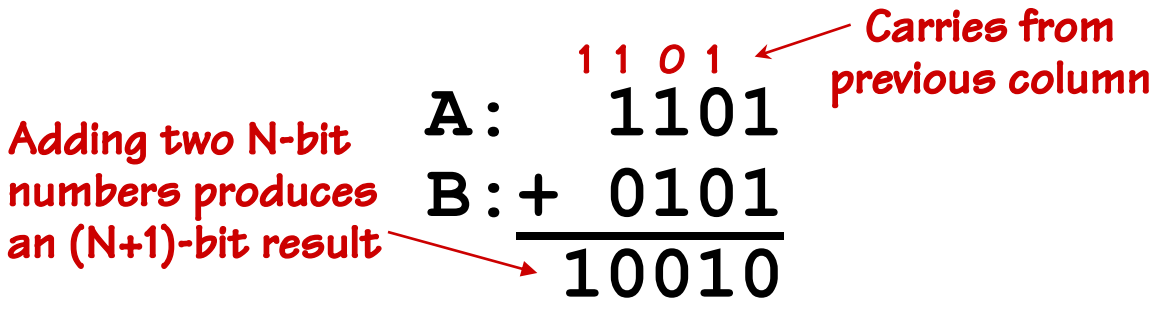
Binary Addition

Here's an example of binary addition as one might do it by "hand":



Binary Addition

Here's an example of binary addition as one might do it by "hand":



Binary Addition

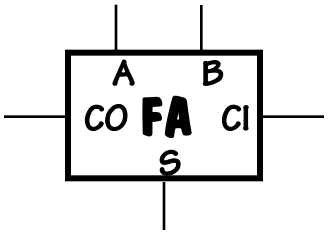
Here's an example of binary addition as one might do it by "hand":

```
      1 1 0 1
A:    1101
B:+   0101
     -------
     10010
```

Carries from previous column

Adding two N-bit numbers produces an (N+1)-bit result

Let's start by building a block that adds one column:



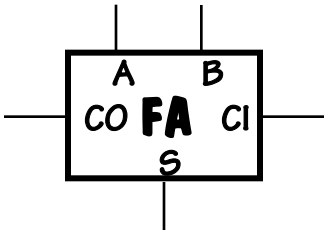
Binary Addition

Here's an example of binary addition as one might do it by "hand":

```
          1 1 0 1
A:      1101
B:+     0101
       10010
```

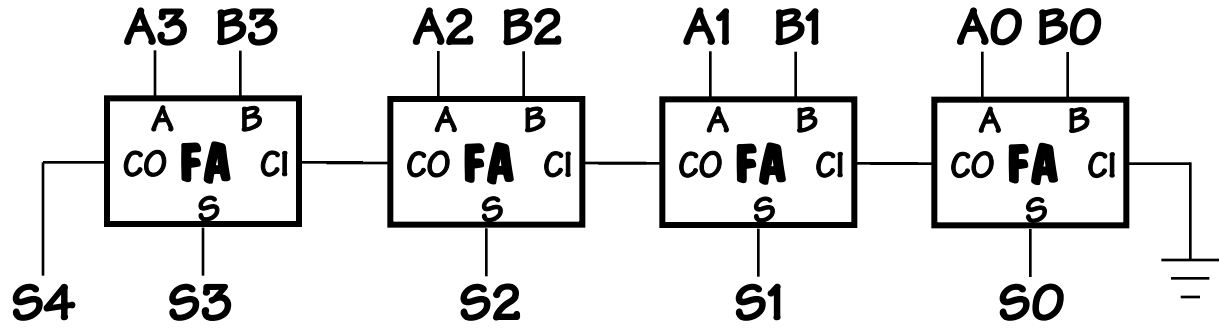
Carries from previous column

Adding two N-bit numbers produces an (N+1)-bit result



Let's start by building a block that adds one column:

Then we can cascade them to add two numbers of any size…



Designing a "Full Adder": From Last Time

1) Start with a truth table:

C_i	A	B	C_o	S
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

2) Write down eqns for the "1" outputs

$$C_o = \overline{C}_i A B + C_i \overline{A} B + C_i A \overline{B} + C_i A B$$
$$S = \overline{C}_i \overline{A} B + \overline{C}_i A \overline{B} + C_i \overline{A} \overline{B} + C_i A B$$

3) Simplifing a bit

$$C_o = C_i(A + B) + AB$$
$$S = C_i \oplus A \oplus B$$

Designing a "Full Adder": From Last Time

1) Start with a truth table:

C_i	A	B	C_o	S
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

2) Write down eqns for the "1" outputs

$$C_o = \overline{C}_i A B + C_i \overline{A} B + C_i A \overline{B} + C_i A B$$
$$S = \overline{C}_i \overline{A} B + \overline{C}_i A \overline{B} + C_i \overline{A} \overline{B} + C_i A B$$

3) Simplifing a bit

$$C_o = C_i(A + B) + AB$$
$$S = C_i \oplus A \oplus B$$

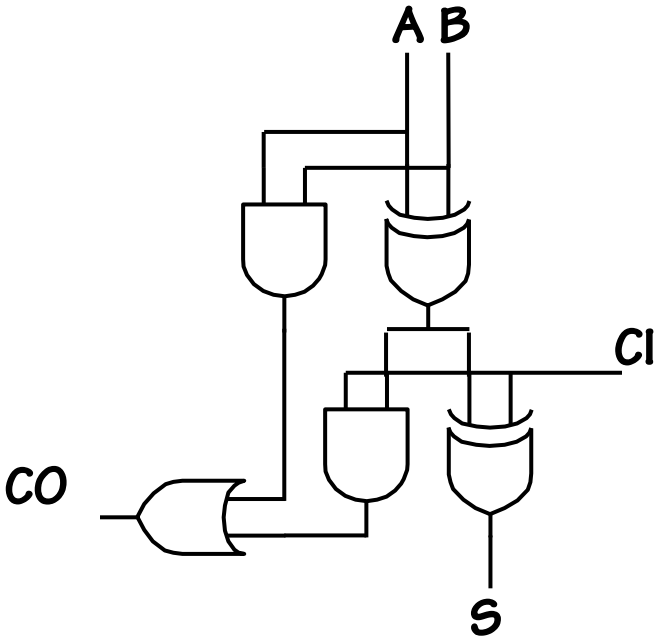
$$C_o = C_i(A \oplus B) + AB$$
$$S = C_i \oplus (A \oplus B)$$

For Those Who Prefer Logic Diagrams …

$C_o = C_i(A \oplus B) + AB$

$S = C_i \oplus (A \oplus B)$

- A little tricky, but only 5 gates/bit

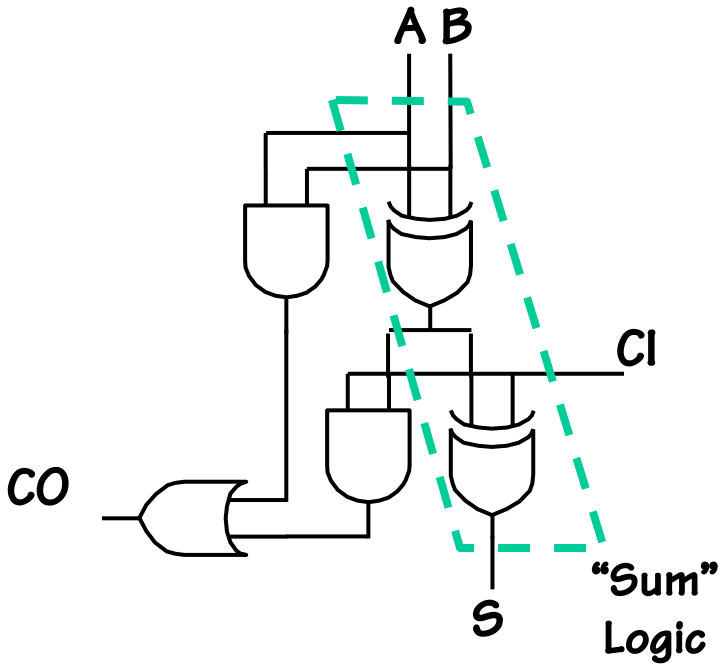


For Those Who Prefer Logic Diagrams …

$C_o = C_i(A \oplus B) + AB$

$S = C_i \oplus (A \oplus B)$

- A little tricky, but only 5 gates/bit

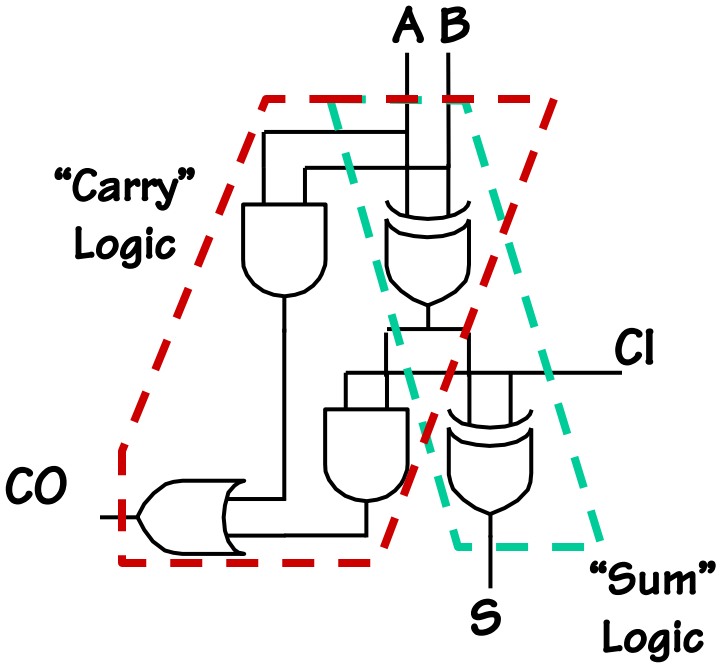


For Those Who Prefer Logic Diagrams …

$C_o = C_i(A \oplus B) + AB$

$S = C_i \oplus (A \oplus B)$

- A little tricky, but only 5 gates/bit



Subtraction: A-B = A + (-B)

Using 2's complement representation: –B = ~B + 1

Subtraction: A-B = A + (-B)

Using 2's complement representation: $-B = \sim B + 1$

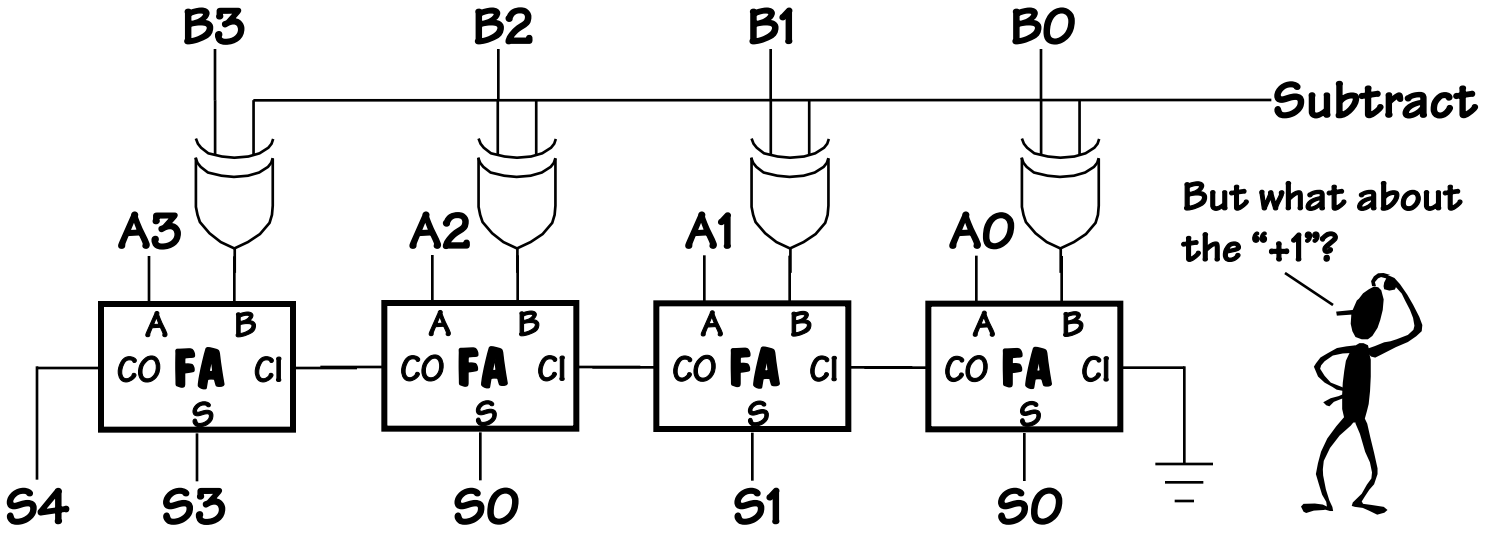
~ = bit-wise complement

Subtraction: A-B = A + (-B)

Using 2's complement representation: –B = ~B + 1

~ = bit-wise complement

So let's build an arithmetic unit that does both addition and subtraction.
Operation selected by *control input*:

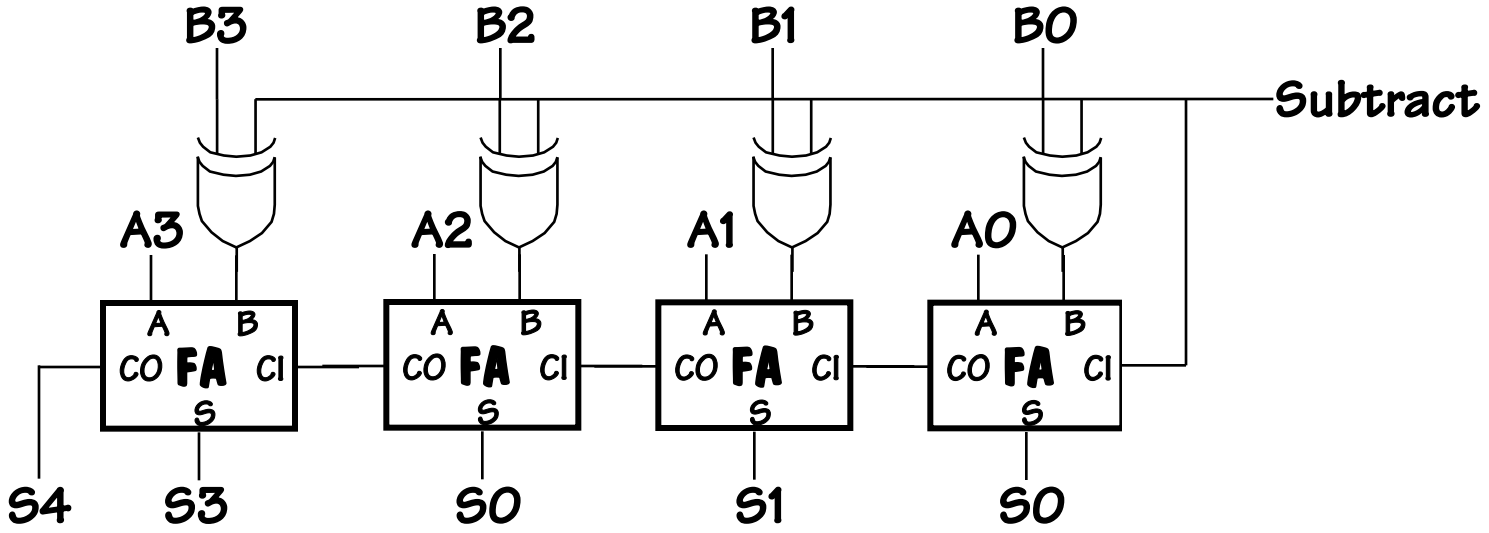


Subtraction: A-B = A + (-B)

Using 2's complement representation: $-B = \sim B + 1$

~ = bit-wise complement

So let's build an arithmetic unit that does both addition and subtraction.
Operation selected by *control input*:



Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 *big NOR gate*

Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 — *big NOR gate*

N (negative): result is < 0 — S_{N-1}

Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 *big NOR gate*

N (negative): result is < 0 S_{N-1}

C (carry): indicates that add in the most significant position produced a carry, e.g., "1 + (-1)" *from last FA*

Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 *big NOR gate*

N (negative): result is < 0 S_{N-1}

C (carry): indicates that add in the most significant position produced a carry, e.g., "1 + (-1)" *from last FA*

V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., "$(2^{i-1} - 1) + (2^{i-1} - 1)$"

Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 *big NOR gate*

N (negative): result is < 0 S_{N-1}

C (carry): indicates that add in the most significant position produced a carry, e.g., "1 + (-1)" *from last FA*

V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., "$(2^{i-1} - 1) + (2^{i-1} - 1)$"

$$V = A_{i-1}\, B_{i-1}\, \overline{N} + \overline{A}_{i-1}\, \overline{B}_{i-1}\, N$$

Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 *big NOR gate*

N (negative): result is < 0 S_{N-1}

C (carry): indicates that add in the most significant position produced a carry, e.g., "1 + (-1)" *from last FA*

V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., "$(2^{i-1} - 1) + (2^{i-1} - 1)$"

$$V = A_{i-1} B_{i-1} \overline{N} + \overline{A}_{i-1} \overline{B}_{i-1} N$$

-or-

$$V = CO_{i-1} \oplus CI_{i-1}$$

Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 *big NOR gate*

N (negative): result is < 0 S_{N-1}

C (carry): indicates that add in the most significant position produced a carry, e.g., "1 + (-1)" *from last FA*

V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., "$(2^{i-1} - 1) + (2^{i-1} - 1)$"

$$V = A_{i-1} B_{i-1} \overline{N} + \overline{A}_{i-1} \overline{B}_{i-1} N$$

-or-

$$V = CO_{i-1} \oplus CI_{i-1}$$

To compare A and B, perform A–B and use condition codes:

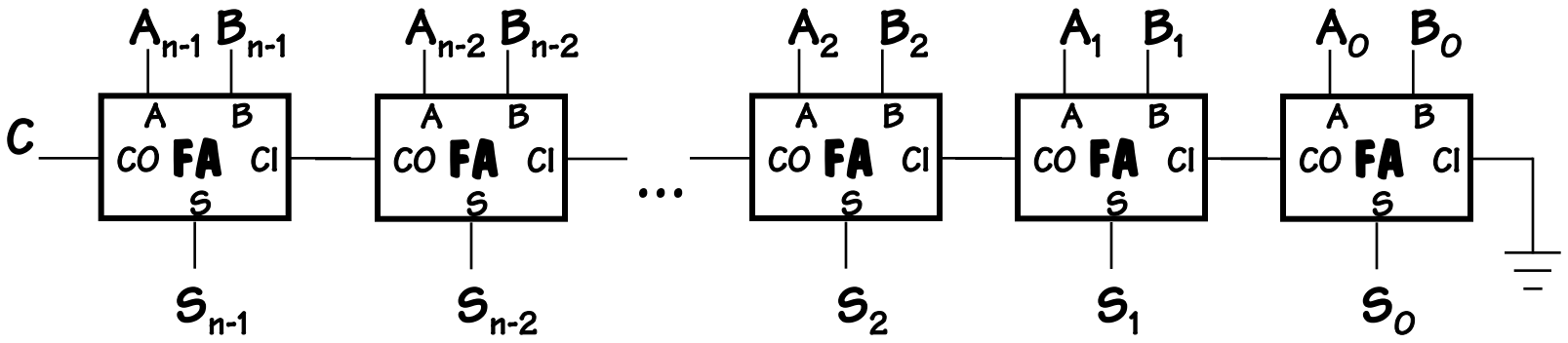
Signed comparison:

LT	N⊕V
LE	Z+ (N⊕V)
EQ	Z
NE	~Z
GE	~ (N⊕V)
GT	~ (Z+ (N⊕V))

Unsigned comparison:

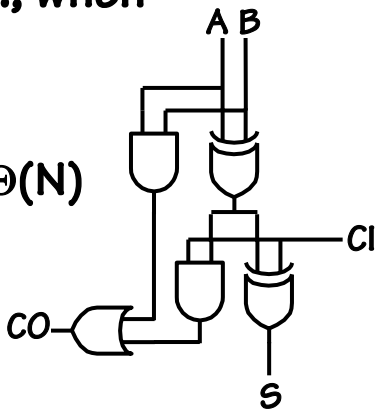
LTU	C
LEU	C+Z
GEU	~C
GTU	~ (C+Z)

T_{PD} of Ripple-Carry Adder



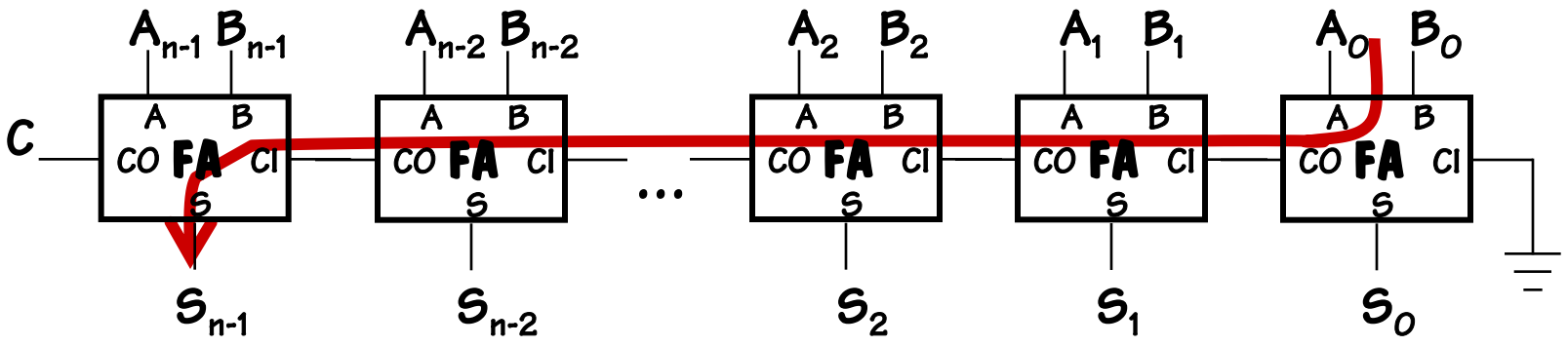
Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$$t_{PD} = \underbrace{(t_{PD,XOR} + t_{PD,AND} + t_{PD,OR})}_{\text{A,B to CO}} + \underbrace{(N-2)*(t_{PD,OR} + t_{PD,AND})}_{\text{CI to CO}} + \underbrace{t_{PD,XOR}}_{CI_{N-1} \text{ to } S_{N-1}} \approx \Theta(N)$$



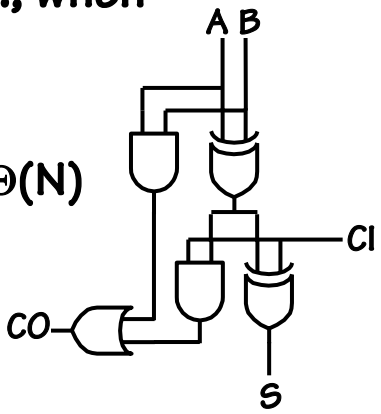
Θ(N) is read "order N" and tells us that the latency of our adder grows in proportion to the number of bits in the operands.

T_{PD} of Ripple-Carry Adder



Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$$t_{PD} = \underbrace{(t_{PD,XOR} + t_{PD,AND} + t_{PD,OR})}_{\text{A,B to CO}} + \underbrace{(N-2)*(t_{PD,OR} + t_{PD,AND})}_{\text{CI to CO}} + \underbrace{t_{PD,XOR}}_{CI_{N-1} \text{ to } S_{N-1}} \approx \Theta(N)$$



Θ(N) is read "order N" and tells us that the latency of our adder grows in proportion to the number of bits in the operands.

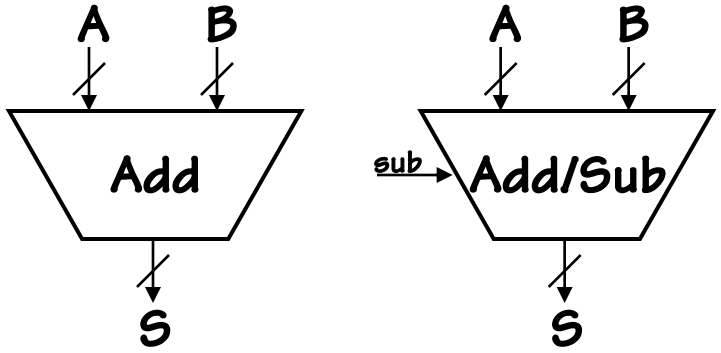
Faster Carry Logic

- Carry-Lookahead Adders (CLA)
- Carry-Skip Adders
- Carry-Select Adders

Adder Summary

Adding is not only common, but it is also tends to be one of the most time-critical of operations. As a result, a wide range of adder architectures have been developed that allow a designer to tradeoff complexity (in terms of the number of gates) for performance.

At this point we'll define a high-level functional unit for an adder, and specify the details of the implementation as necessary.



Shifting Logic

Shifting is a common operation that is applied to groups of bits. Shifting can be used for alignment, as well as for arithmetic operations.

X << 1 is approx the same as 2*X

X >> 1 can be the same as X/2

For example:

$$X = 20_{10} = 00010100_2$$

Left Shift:

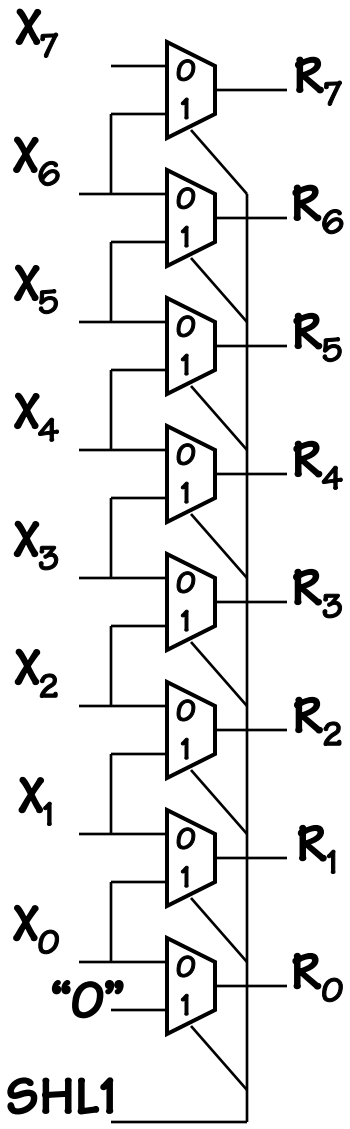
$(X << 1) = 00101000_2 = 40_{10}$

Right Shift:

$(X >> 1) = 00001010_2 = 10_{10}$

Signed or "Arithmetic" Right Shift:

$(-X >> 1) = (11101100_2 >> 1) = 11110110_2 = -10_{10}$



Boolean Operations

It will also be useful to perform logical operations on groups of bits.

Which ones?

Boolean Operations

It will also be useful to perform logical operations on groups of bits.

Which ones?

ANDing is useful for "masking" off groups of bits.

ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)

Boolean Operations

It will also be useful to perform logical operations on groups of bits.

Which ones?

ANDing is useful for "masking" off groups of bits.

ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)

ANDing is also useful for "clearing" groups of bits.

ex. 10101110 & 00001111 = 00001110 (0's clear first 4 bits)

Boolean Operations

It will also be useful to perform logical operations on groups of bits.

Which ones?

ANDing is useful for "masking" off groups of bits.

ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)

ANDing is also useful for "clearing" groups of bits.

ex. 10101110 & 00001111 = 00001110 (0's clear first 4 bits)

ORing is useful for "setting" groups of bits.

ex. 10101110 | 00001111 = 10101111 (1's set last 4 bits)

Boolean Operations

It will also be useful to perform logical operations on groups of bits.

Which ones?

ANDing is useful for "masking" off groups of bits.

ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)

ANDing is also useful for "clearing" groups of bits.

ex. 10101110 & 00001111 = 00001110 (0's clear first 4 bits)

ORing is useful for "setting" groups of bits.

ex. 10101110 | 00001111 = 10101111 (1's set last 4 bits)

XORing is useful for "complementing" groups of bits.

ex. 10101110 ^ 00001111 = 10100001 (1's complement last 4 bits)

Boolean Operations

It will also be useful to perform logical operations on groups of bits.
Which ones?

ANDing is useful for "masking" off groups of bits.
ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)

ANDing is also useful for "clearing" groups of bits.
ex. 10101110 & 00001111 = 00001110 (0's clear first 4 bits)

ORing is useful for "setting" groups of bits.
ex. 10101110 | 00001111 = 10101111 (1's set last 4 bits)

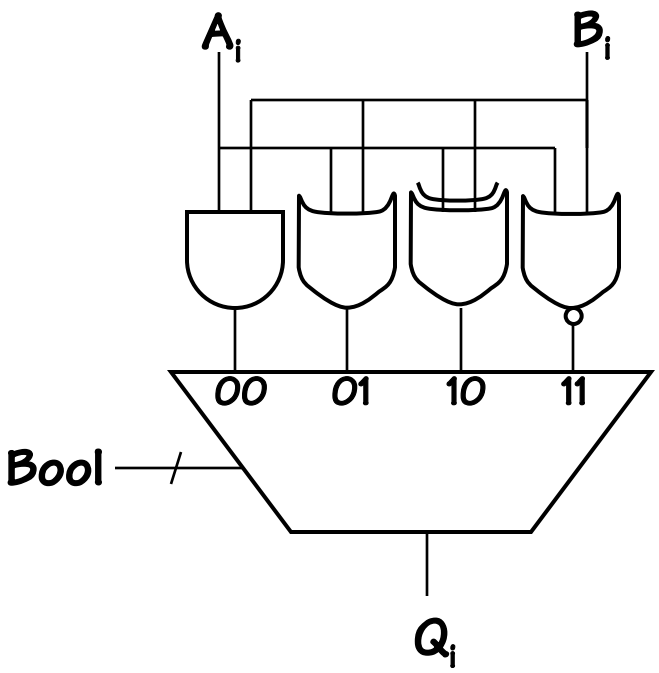
XORing is useful for "complementing" groups of bits.
ex. 10101110 ^ 00001111 = 10100001 (1's complement last 4 bits)

NORing is useful.. Uhm, because John Hennessy says it is!
ex. 10101110 # 00001111 = 01010000 (0's complement, 1's clear)

Boolean Unit

It is simple to build up a Boolean unit using primitive gates and a mux to select the function.

Since there is no interconnection between bits, this unit can be simply replicated at each position. The cost is about 7 gates per bit. One for each primitive function, and approx 3 for the 4-input mux.

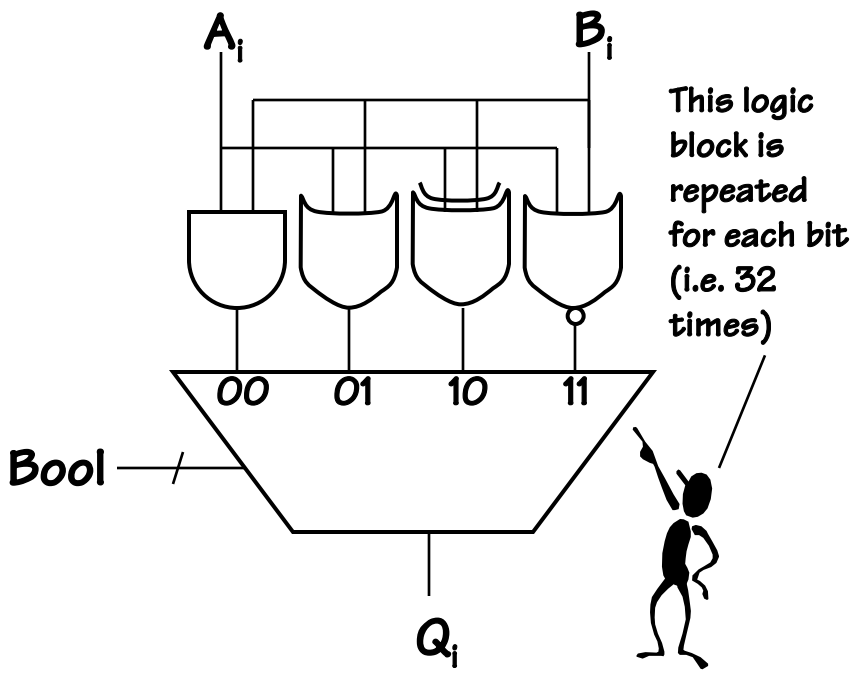


This is a straightforward, but not too elegant of a design.

Boolean Unit

It is simple to build up a Boolean unit using primitive gates and a mux to select the function.

Since there is no interconnection between bits, this unit can be simply replicated at each position. The cost is about 7 gates per bit. One for each primitive function, and approx 3 for the 4-input mux.



This is a straightforward, but not too elegant of a design.

An ALU, at Last

Now we're ready for a big one! An Arithmetic Logic Unit.