CS3350B Computer Organization
Chapter 2: Synchronous Circuits
Part 2: Stateless Circuits

Iqra Batool

Department of Computer Science
University of Western Ontario, Canada

Wednesday January 31, 2024

Outline

Stateless Circuits are Combinational Circuits

- **Stateless** ⇒ No memory.
- **Combinational** ⇒ Output is a combination of the inputs alone.

Combinational circuits are formed from a combination of logic gates and other combinational cirtcuits.

- ↳ Modular Design,
- ↳ Reuse,
- ↳ Simple to add new components.

Sometimes, these are called **functional blocks**, they implement *functions*.

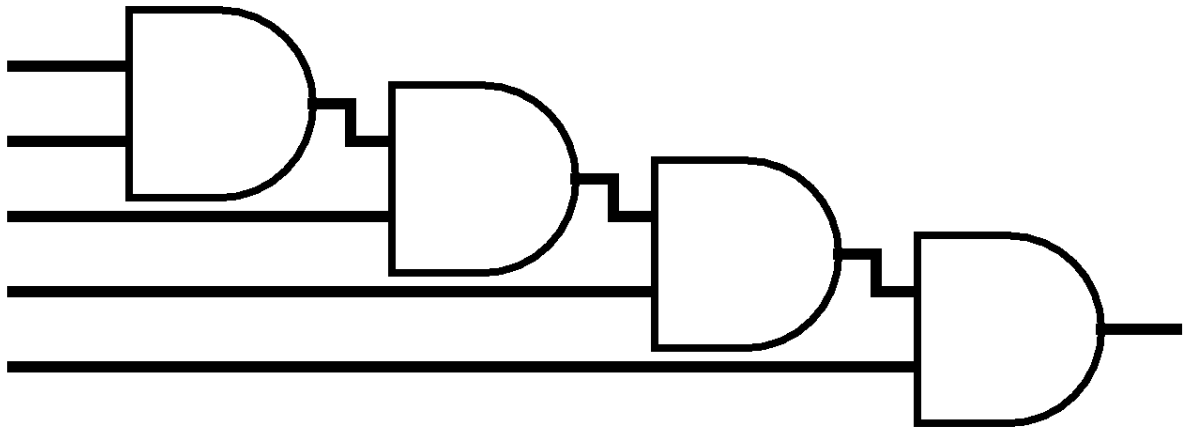
Increasing Arity

Arity: the number of inputs to a gate, function, etc.

How can we build an n-way add from simple 2-input and gates?

↳ Simply chain together $n-1$ 2-way gates.

Example: 5-way AND

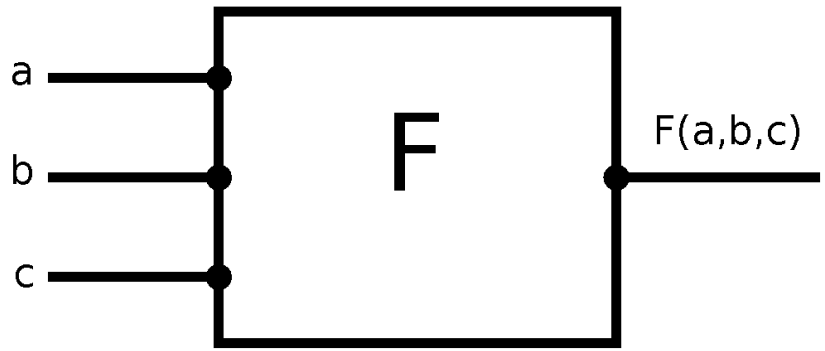


Works for AND, OR, XOR. *Doesn't* work for NAND, NOR.

Block Diagrams

A **block diagram** or **schematic diagram** can use used to express the high-level specification of a circuit.

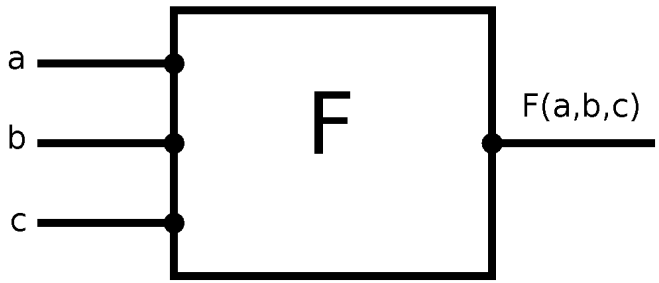
- How many inputs, how many bits for each input?
- How many outputs, how many bits for each output?
- What does the circuit do? Formula or truth table.



$$F \equiv \overline{abc} + abc$$

a	b	c	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

From Blocks to Gates (1/2)



a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

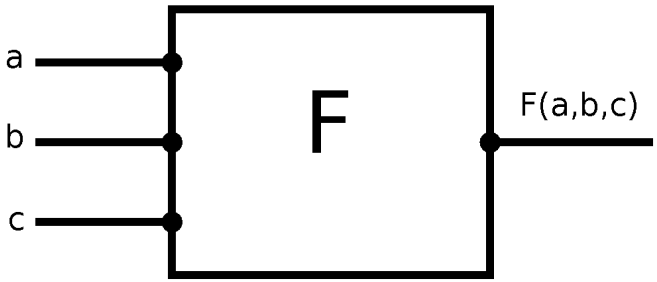
1. Generate truth table.
2. Get canonical form:

$$F \equiv \overline{a}bc + a\overline{b}c + ab\overline{c} + abc$$

3. Simplify if possible:

$$\begin{aligned}
&\overline{a}bc + a\overline{b}c + ab\overline{c} + abc \\
&\equiv \overline{a}bc + a\overline{b}c + ab\overline{c} + abc + abc + abc \\
&\equiv \overline{a}bc + abc + a\overline{b}c + abc + ab\overline{c} + abc \\
&\equiv bc + ac + ab
\end{aligned}$$

From Blocks to Gates (2/2)

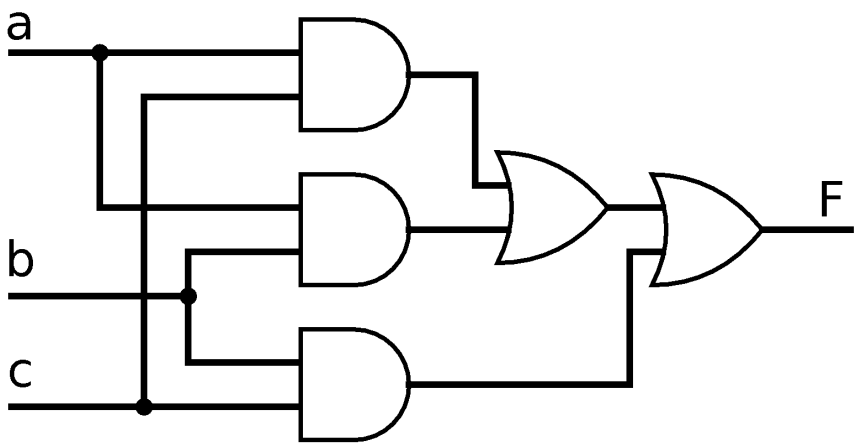


a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

3. Simplify if possible:

$$F \equiv bc + ac + ab$$

4. Draw your circuit from simplified formula.

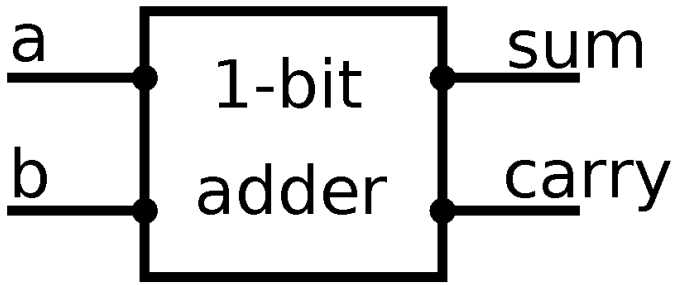


This is called a *majority* circuit. Output is true iff majority of inputs are true.

Outline

1 Bit Adder

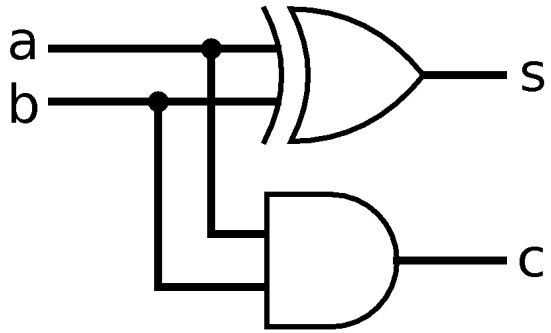
Adder interprets bits as a binary number and does addition.



a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$s = add(a, b)$

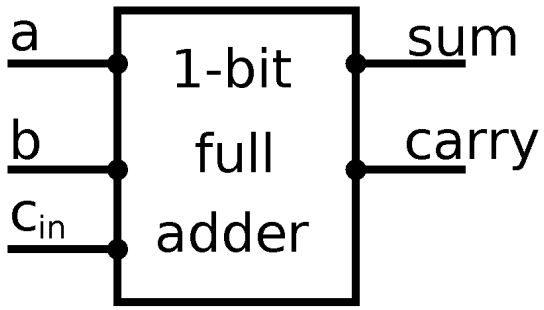
c = carry (overflow) bit

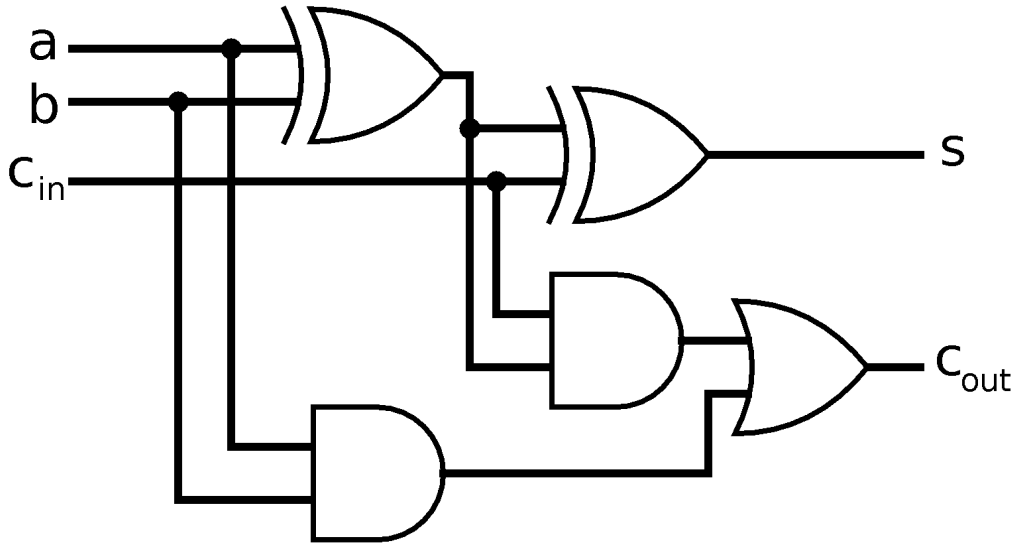


1 Bit Full Adder

Full Adder does addition of 3 inputs: a, b, and carry$_{in}$.

↳ Previous adder was a *half* adder.





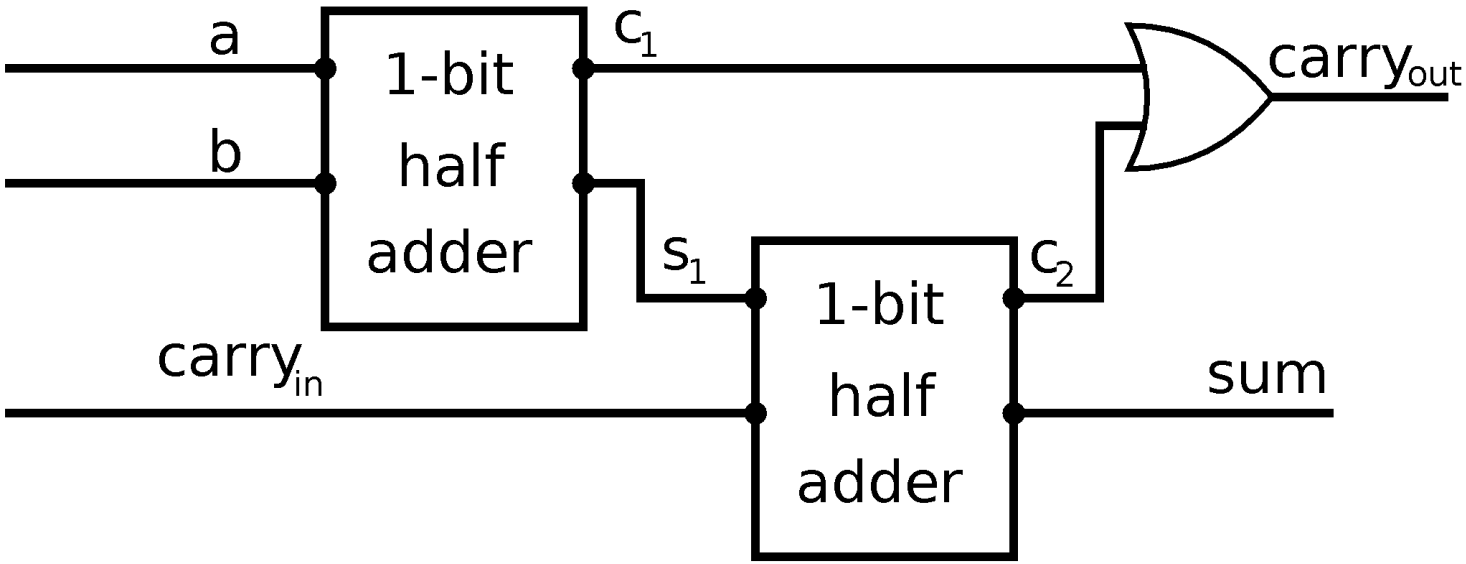
$$s = (a\ XOR\ b)\ XOR\ c_{in}$$

$$c = ab + (XOR(a, b) \cdot c_{in})$$

1 Bit Full Adder using Half Adders

A full adder can be built from half adders.

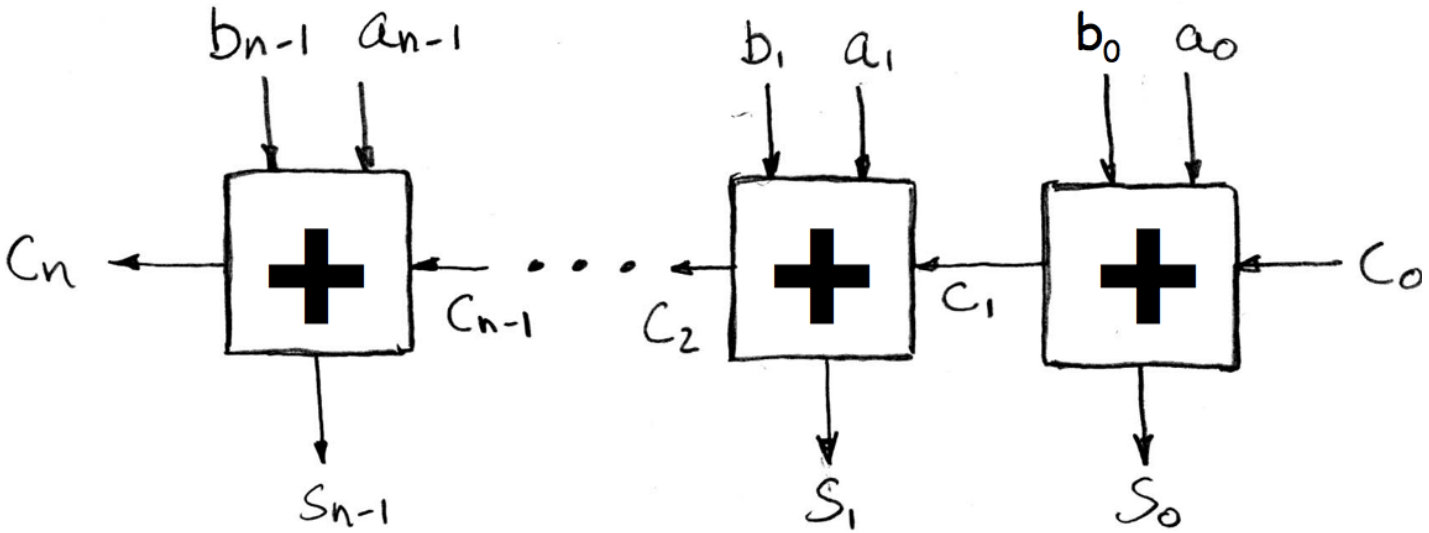
↳ Modular design, reuse, simplified view.



n-Bit Full Adder

n-bit adder: Add n bits with carry.

- Just like long addition done by hand.
- Combine n full adders, adding bit by bit, carrying the carry from lowest-ordered bit to highest-ordered bit.
- Final carry bit is c_n.



Addition Overflow (1/2)

Overflow occurs when arithmetic results in a number strictly larger than can fit in the predetermined number of bits.

- For *unsigned* integers, overflow is detected by c_n being 1.
- For *signed* (i.e. twos-compliment) integers, overflow more interesting.

Example: Addition in 4 bits.

$$\begin{array}{rl} 1000 & \text{(carry bits)} \\ 1101 & \\ +\ 0110 & \\ \hline 10011 & \Rightarrow \quad c_n \text{ is 1. Overflow?} \end{array}$$

Addition Overflow (1/2)

Overflow occurs when arithmetic results in a number strictly larger than can fit in the predetermined number of bits.

- For *unsigned* integers, overflow is detected by c_n being 1.
- For *signed* (i.e. twos-compliment) integers, overflow more interesting.

Example: Addition in 4 bits.

$$\begin{array}{rl} 1000 & \text{(carry bits)} \\ 1101 & \\ +\ 0110 & \\ \hline 10011 & \Rightarrow \quad c_n \text{ is 1. Overflow?} \end{array}$$

$$\begin{array}{rl} -3 & \\ +\ 6 & \\ \hline 3 & \Rightarrow \quad \text{No overflow} \\ & \text{Discard last carry bit} \end{array}$$

Addition Overflow (2/2)

In twos-compliment when is there overflow?

- Most significant bit encodes a negative number in two's compliment.
- If both operands are positive *and* $c_{n-1} \equiv 1$ then we have overflow.
- If one positive and one negative, overflow impossible.
  - ↳ Their sum is always closer to zero than either of the operands.
- If both operands are negative *and* $c_n \equiv 1$ then we have overflow.

$$\begin{array}{r} 1000 \\ 0101 \\ +0110 \\ \hline 1011 \end{array} \Rightarrow \text{Overflow}$$

$$\begin{array}{r} 1000 \\ +1000 \\ \hline 10000 \end{array} \Rightarrow \text{Overflow}$$

$$\begin{array}{r} 1000 \\ 1101 \\ +\ 0110 \\ \hline 10011 \end{array} \Rightarrow \text{No overflow}$$

Overflow in two's compliment: c_n XOR c_{n-1}.

n-bit Subtractor (1/2)

n-bit subtractor: Subtract two n-bit numbers.

- We want $s = a - b$.
- Start with an n-bit adder.
- XOR b with a **control signal** for subtraction.
  - ↳ signal is 1 for subtraction, 0 for addition.

XOR works as conditional inverter.

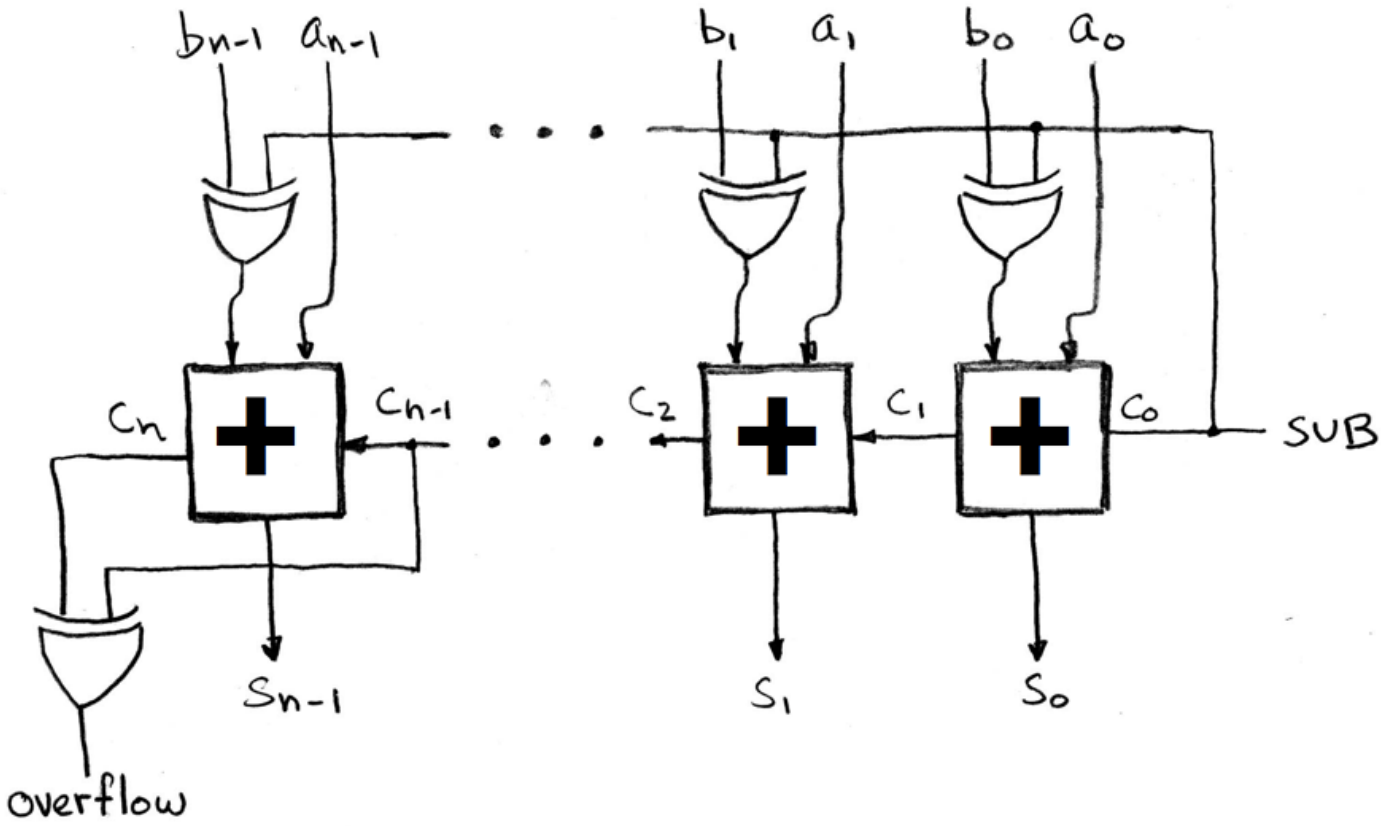
↳ A XOR B ≡ C $\Longrightarrow$ if (B) then $\overline{\text{A}}$ ≡ C else A ≡ C.

A	B	A ⊕ B ≡ C
0	0	0
0	1	1
1	0	1
1	1	0

n-bit Subtractor (2/2)

Control signal SUB acts as c_0.

- Recall: *signed negation*. Invert and add one.
- XOR does invert.



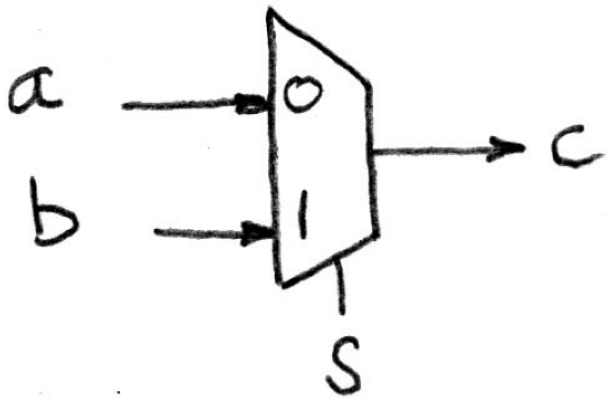
Outline

Multiplexer

A **multiplexer** "mux" conditionally chooses among its inputs to set value of output.

- Uses **control signal** to control which input is chosen.
- Still no state, output depending only on inputs: input bits and control signal.

2-way multiplexer



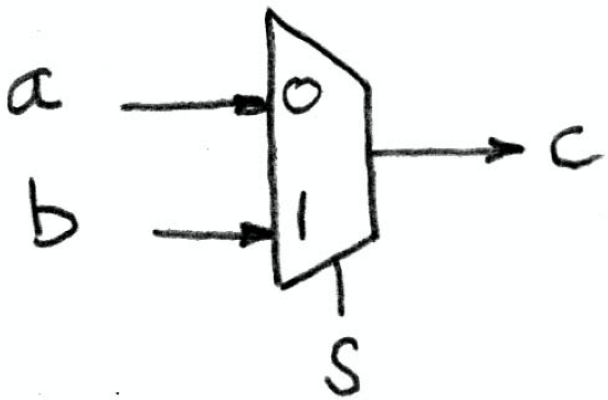
If $s \equiv 0$ then $c \equiv a$.
If $s \equiv 1$ then $c \equiv b$.

Notice actual value of a and b have no effect on decision.

↳ 0 and 1 in multiplexer is not the *value* of a or b but the "index".

2-way Multiplexer

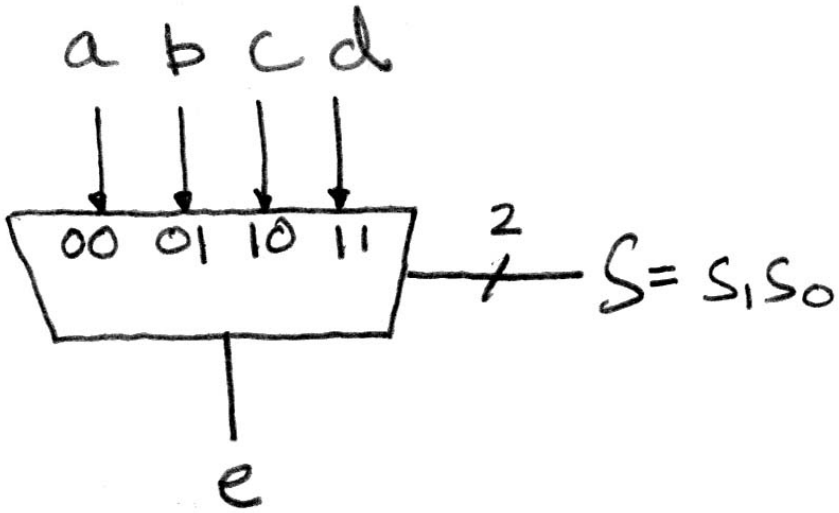
How to encode this "if-then" behaviour without actual conditionals?



$$
\begin{aligned}
c &\equiv MUX(a, b, s) \\
&\equiv \overline{s}a(b + \overline{b}) + sb(a + \overline{a}) \\
&\equiv \overline{s}a + sb
\end{aligned}
$$

Note: $X \cdot (Y + \overline{Y})$ encodes "X independent of what the value of Y is".

4-way Multiplexer

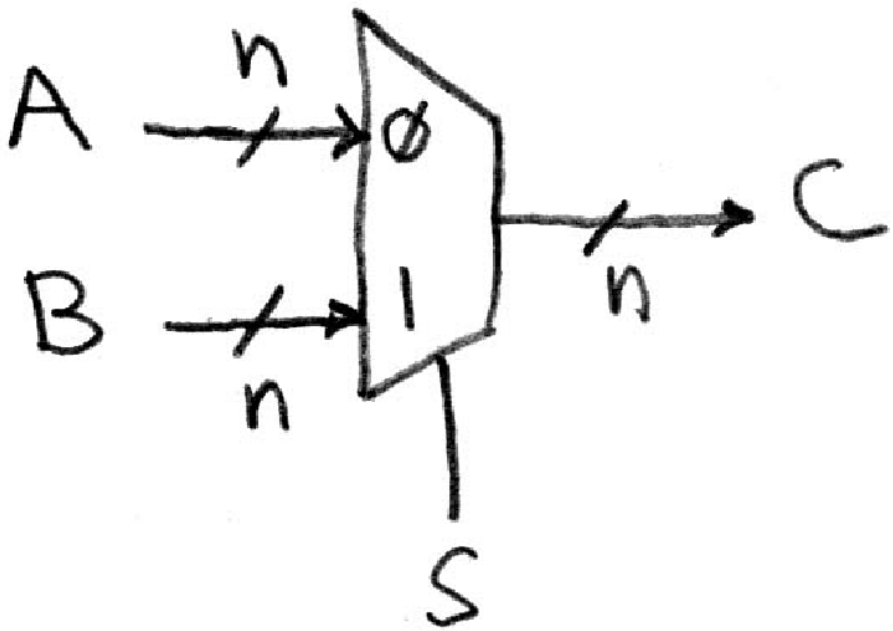


$$e \equiv MUX(a, b, c, d, S)$$
$$\equiv \overline{s_1 s_0} a + \overline{s_1} s_0 b + s_1 \overline{s_0} c + s_1 s_0 d$$

The index of each input is now 0 through 3.

- Need 2 bits to choose among 4 inputs.
- Control signal's **bit-width** is now 2.

Big Data Multiplexer

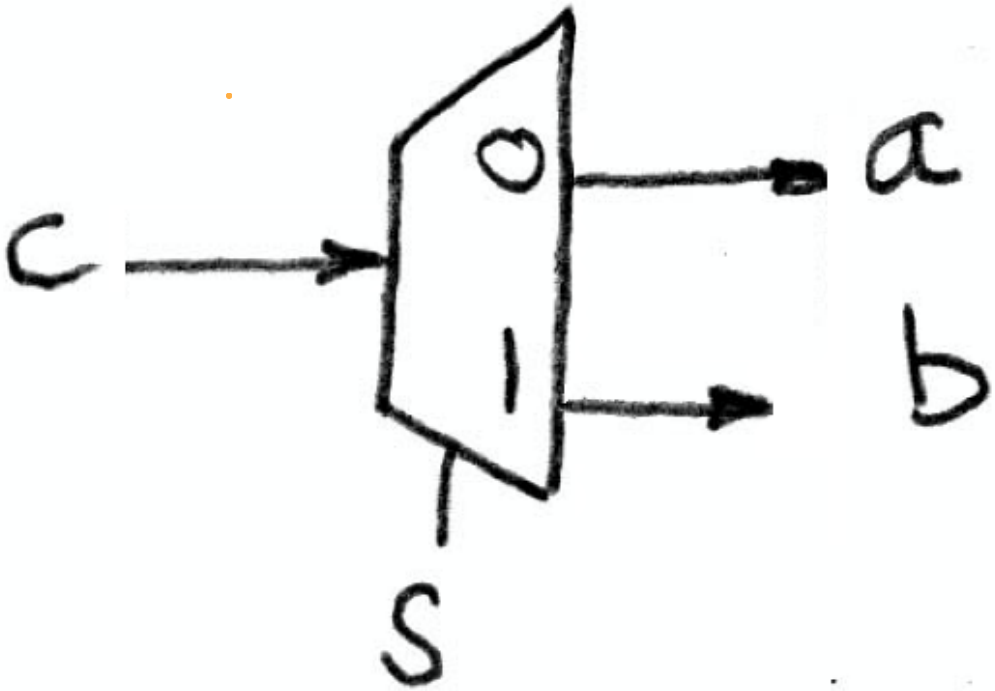


Bit-width of input and output must match, but bit-width of *control signal* only determined by number of inputs to choose from.

Demultiplexer

Demultiplexer "demux" conditionally chooses among its outputs.

- Opposite of MUX.
- Un-selected outputs are set to 0.

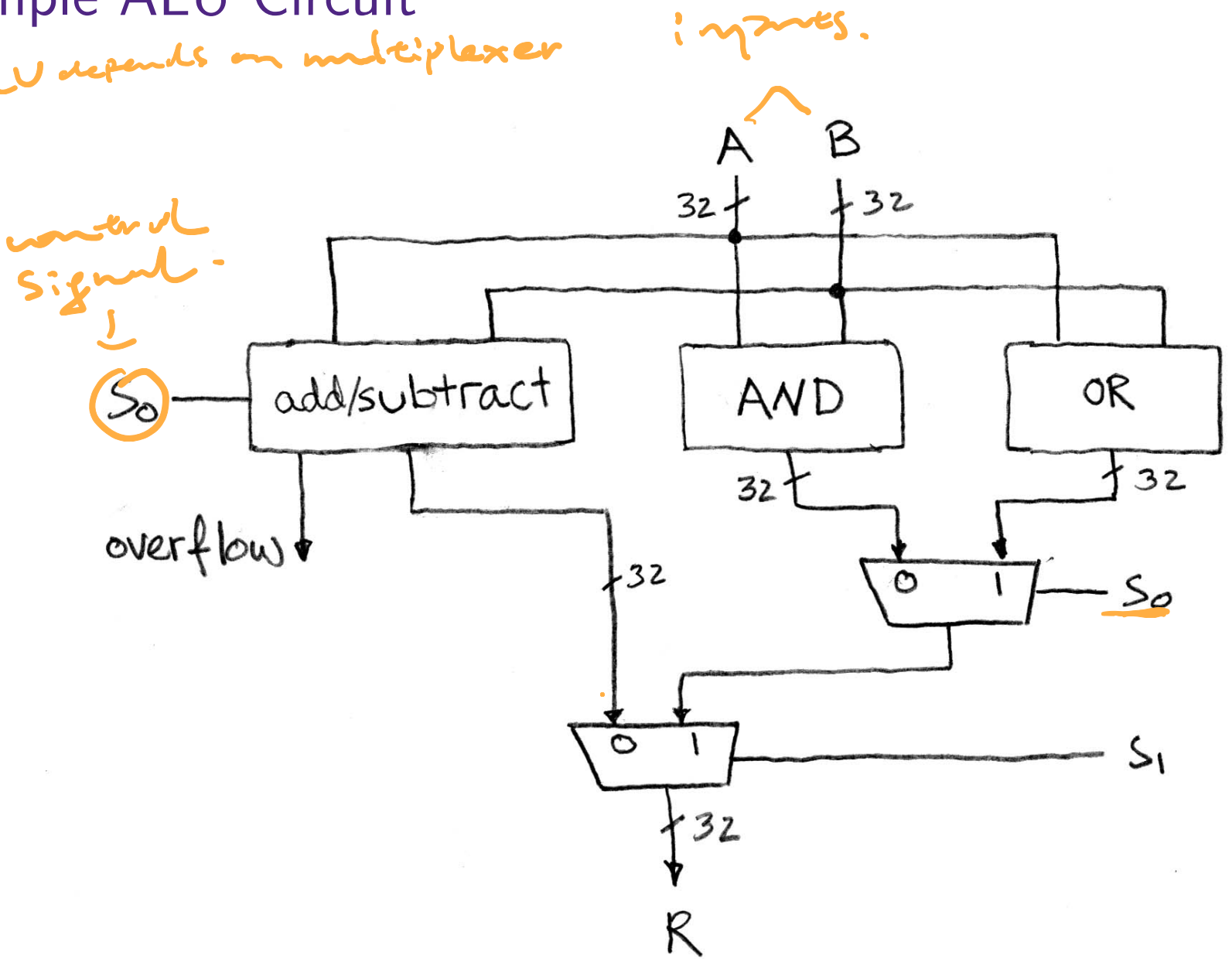


Outline

Arithmetic Logic Unit

- An **ALU** is a black-box type circuit which can do many *different* arithmetic or logic operations on its inputs.
  - ↳ Not many at one time, but selectively acts as many.
- Depending on the implementation can do addition, subtraction, multiplication, division, logical AND, logical OR, shifting, etc.
- Use a control signal to choose which operation to perform.
- Essentially a big collection of MUX and combinational blocks for each operation.

Simple ALU Circuit



Optimizing ALU

Remember, every additional gate increases delay and space. Instead, optimize via the normal four step process:

1. Generate a truth table,
2. Get canonical form from truth table,
3. Simplify expression,
4. Make circuit.

Another option: programmable logic array.

Programmable Logic Array

A **programmable logic array** (PLA) directly implements a truth table/canonical disjunctive normal form.

- Each AND row is a truth table row.
- Each OR column is one output bit.
- Each $\oplus$ is a *programmable* (i.e. optional) AND gate (on the AND plane; OR gate on the OR plane) joining the input to the circuit.