inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #16
Representations of Combinatorial Logic Circuits

CPS today!

2005-10-26

There are two handouts today at the front and back of the room!

Lecturer PSOE, new dad Dan Garcia

www.cs.berkeley.edu/~ddgarcia

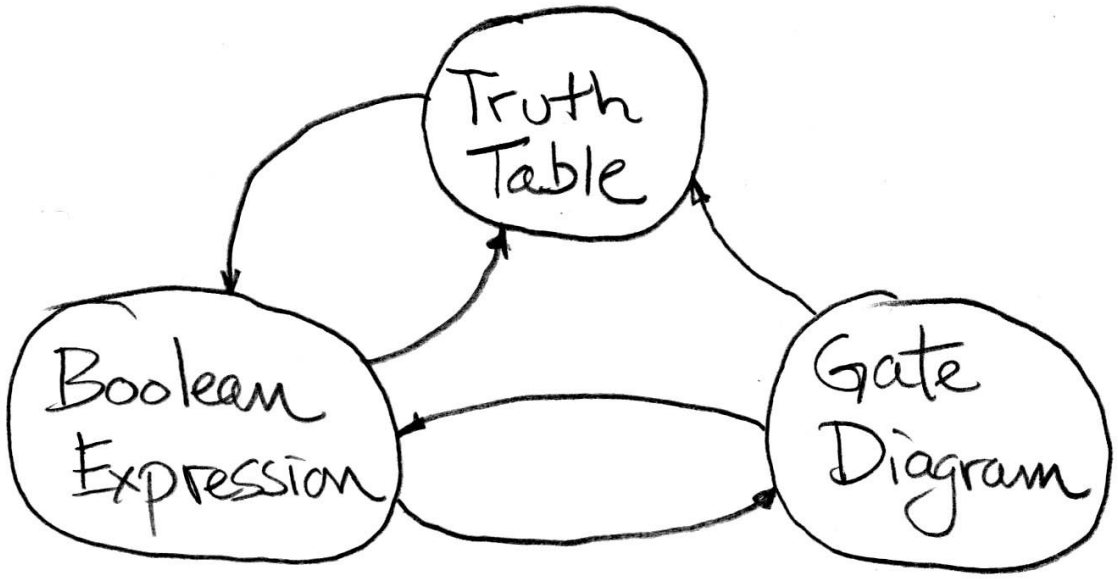
Car makes its own fuel ⇒

An Israeli company has invented a car that can produce its own Hydrogen using common metals like Magnesium and Aluminum. "Exhaust" is harmless metal oxide!

www.isracast.com/tech_news/231005_tech.htm

Review

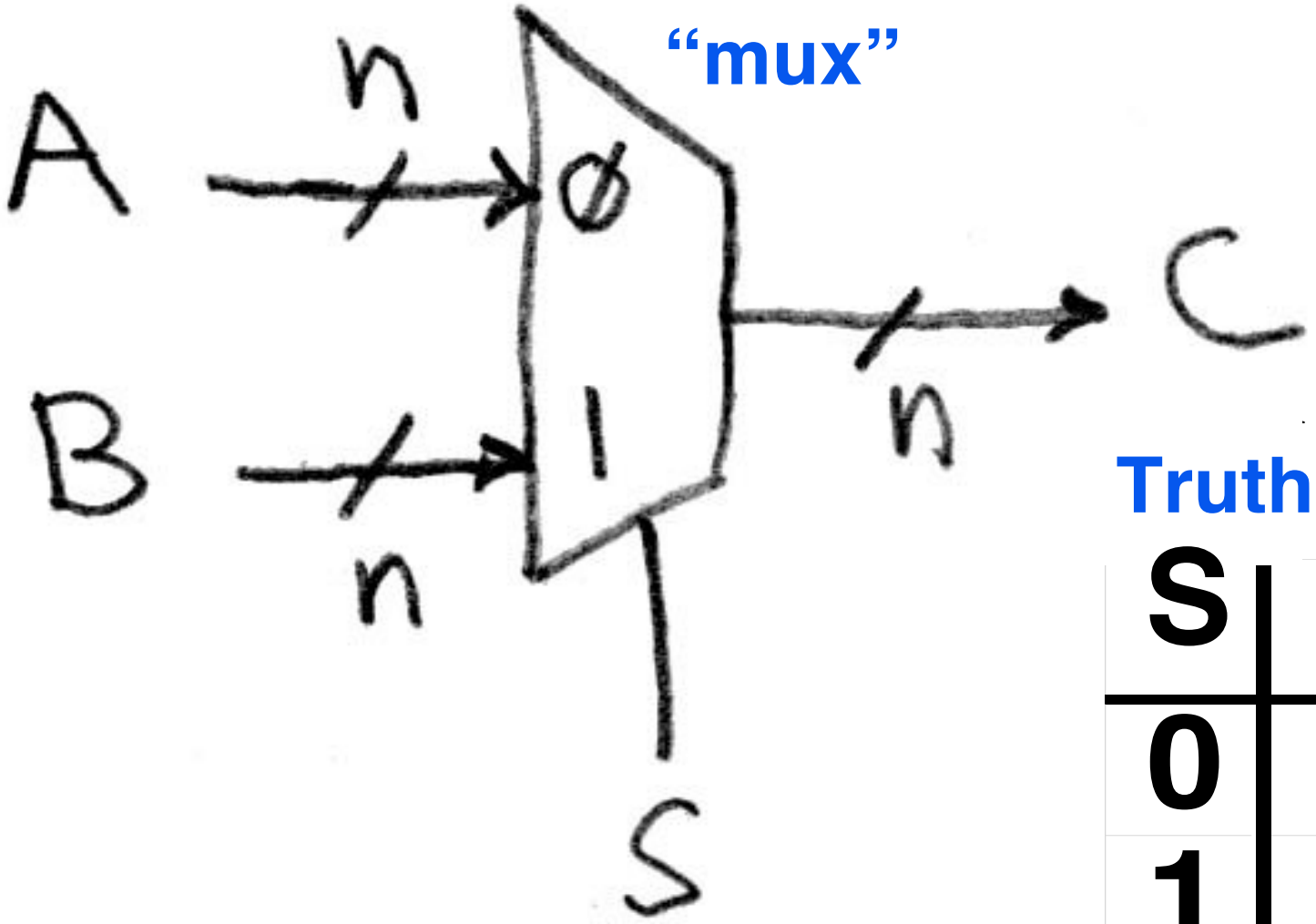
- Pipeline big-delay CL for faster clock
- Finite State Machines extremely useful
  - You'll see them again in 150, 152 & 164
- Use this table and techniques we learned to transform from 1 to another



Today

- Data Multiplexors
- Arithmetic and Logic Unit
- Adder/Subtractor
- Programmable Logic Arrays

Data Multiplexor (here 2-to-1, n-bit-wide)

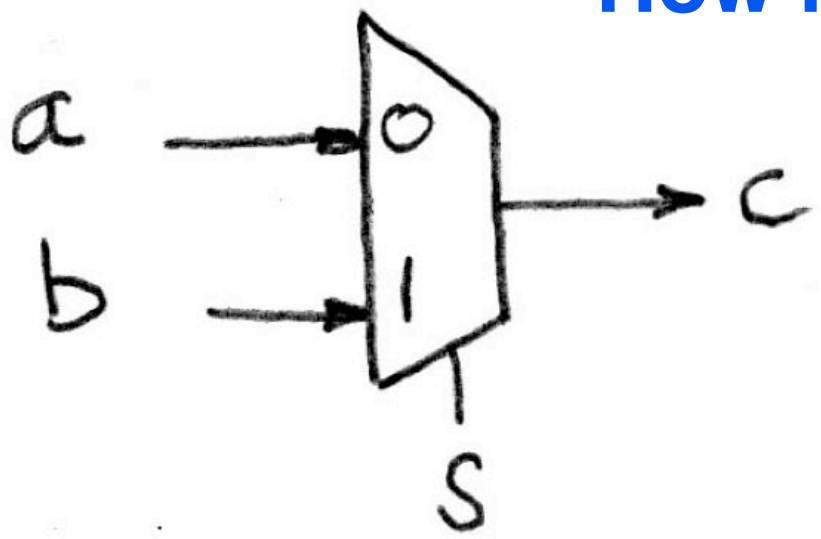


Truth table

S	C
0	A
1	B

N instances of 1-bit-wide mux

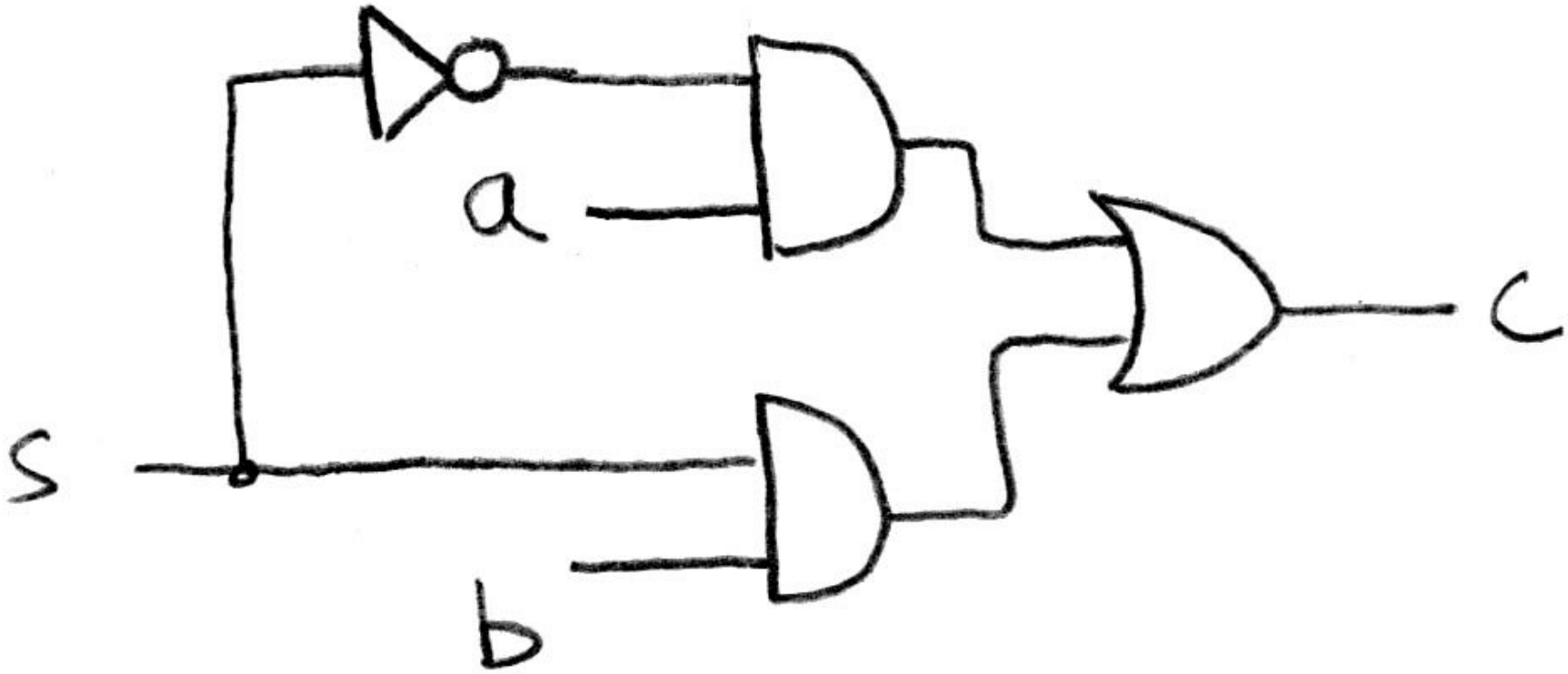
How many rows in TT?



$$
\begin{aligned}
c &= \overline{s}a\overline{b} + \overline{s}ab + s\overline{a}b + sab \\
&= \overline{s}(a\overline{b} + ab) + s(\overline{a}b + ab) \\
&= \overline{s}(a(\overline{b} + b)) + s((\overline{a} + a)b) \\
&= \overline{s}(a(1) + s((1)b) \\
&= \overline{s}a + sb
\end{aligned}
$$

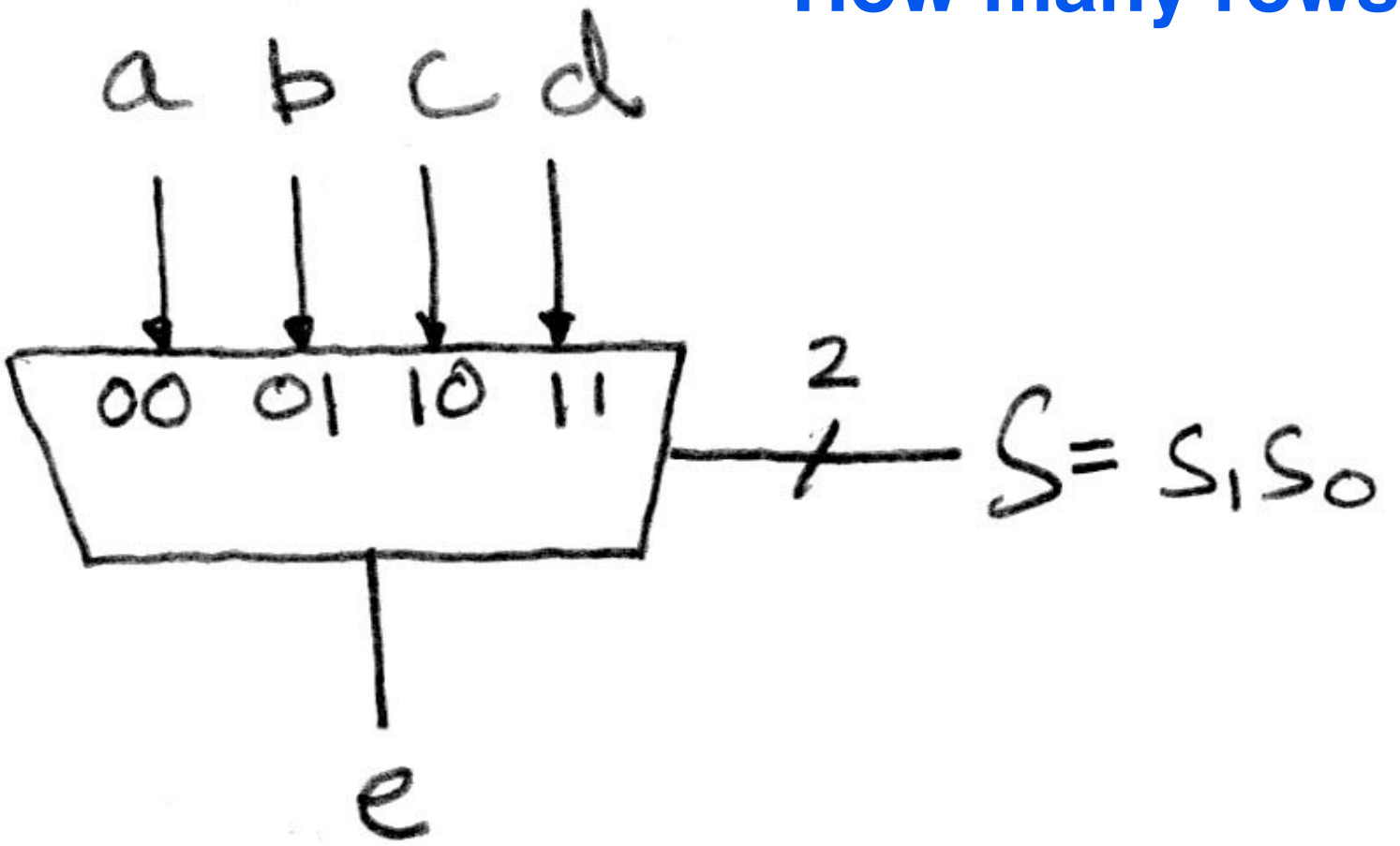
How do we build a 1-bit-wide mux?

$$\overline{s}a + sb$$



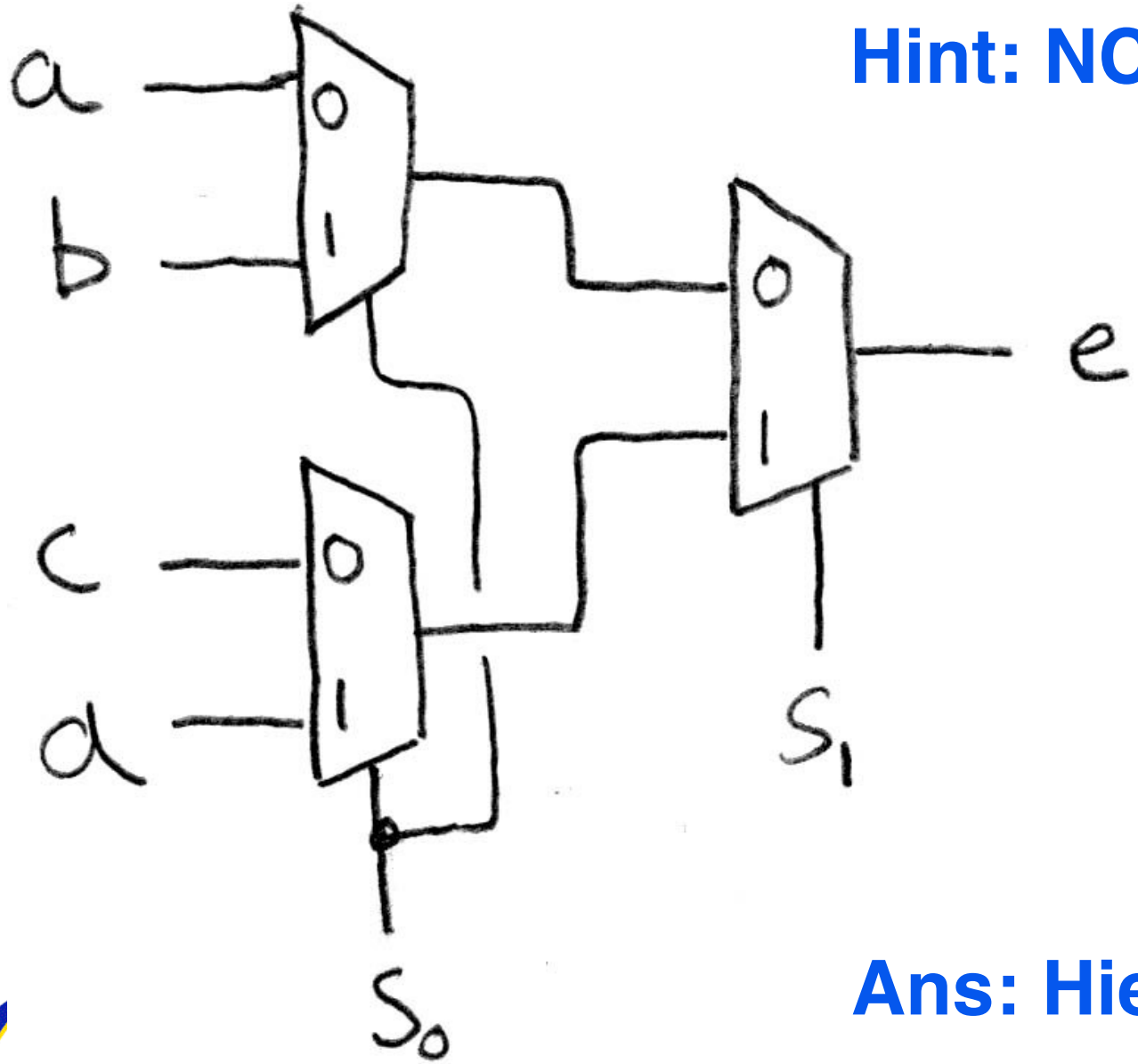
4-to-1 Multiplexor?

How many rows in TT?



$$e = \overline{s_1 s_0} a + \overline{s_1} s_0 b + s_1 \overline{s_0} c + s_1 s_0 d$$

Is there any other way to do it?



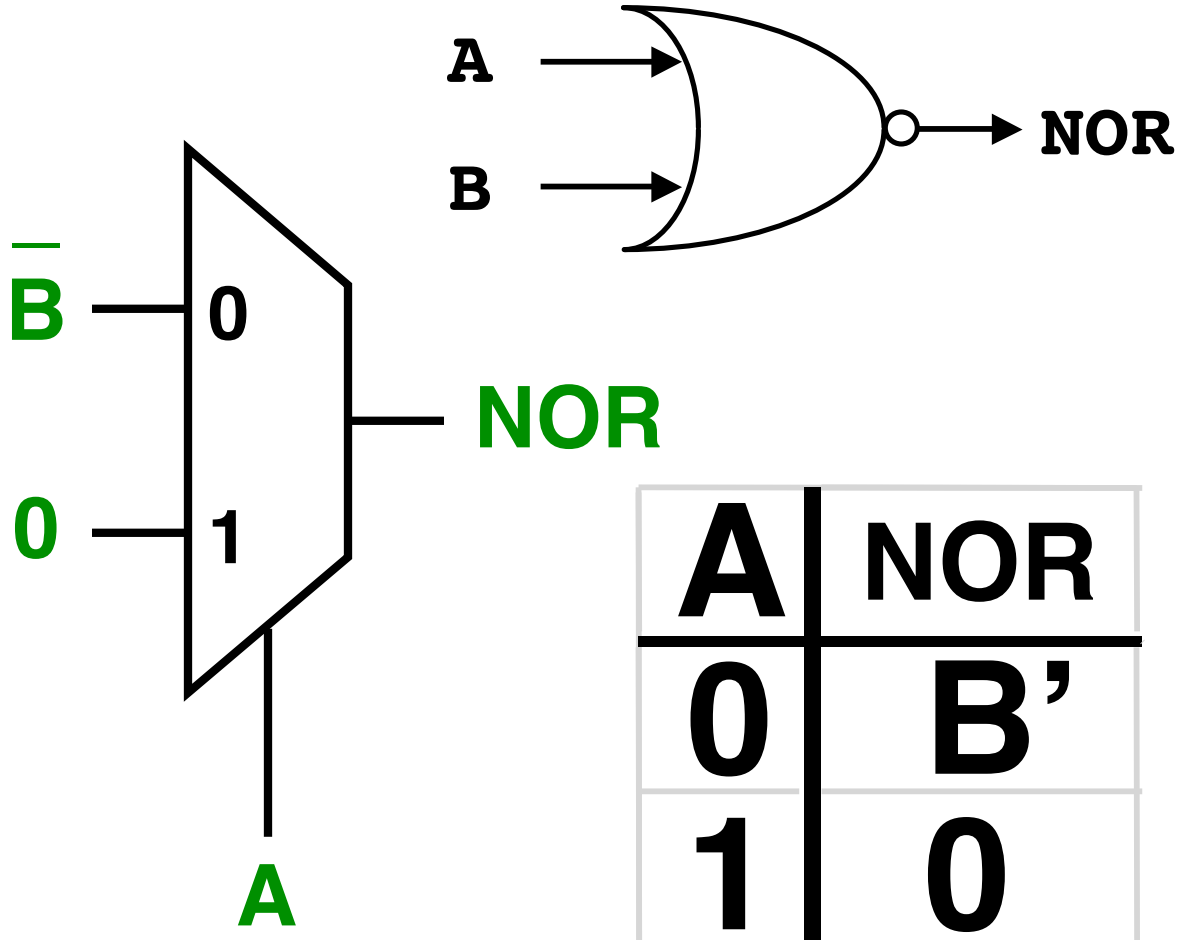
Hint: NCAA tourney!

Ans: Hierarchically!

Do you really understand NORs?

- If one input is 1, what is a NOR?
- If one input is 0, what is a NOR?

A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

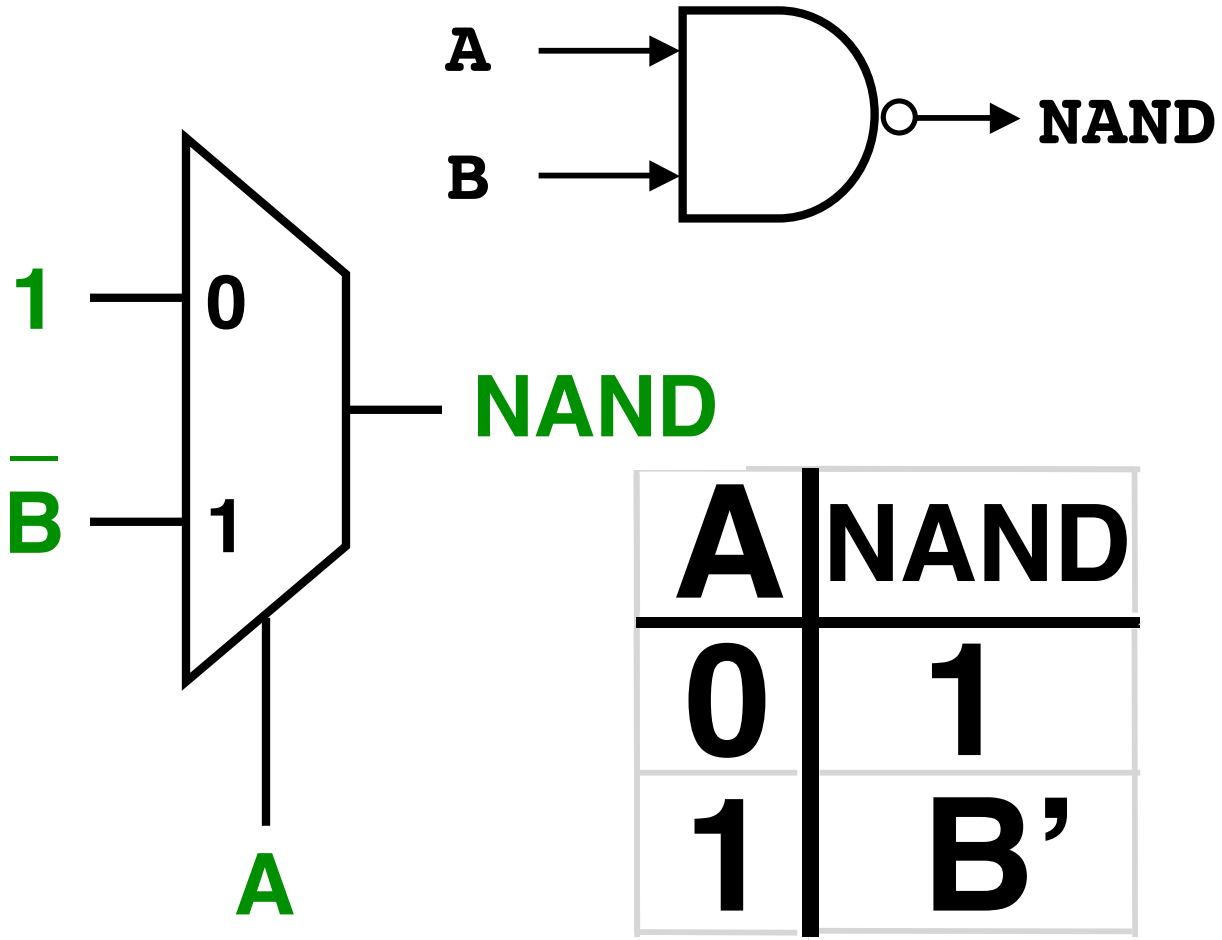


A	NOR
0	B’
1	0

Do you really understand NANDs?

- If one input is 1, what is a NAND?
- If one input is 0, what is a NAND?

A	B	NAND
0	0	1
0	1	1
1	0	1
1	1	0



A	NAND
0	1
1	B’

What does it mean to "clobber" midterm?

- You STILL have to take the final even if you aced the midterm!
- The final will contain midterm-material Qs and new, post-midterm Qs
- They will be graded separately
- If you do "better" on the midterm-material, we will clobber your midterm with the "new" score! If you do worse, midterm unchanged.
- What does "better" mean?
  - Better w.r.t. Standard Deviations around mean
- What does "new" mean?
  - Score based on remapping St. Dev. score on final midterm-material to midterm score St. Dev.

"Clobber the midterm" example

- **Midterm**
  - **Mean: 45**
  - **Standard Deviation: 14**
  - **You got a 31, one σ below. I.e., mean - σ**

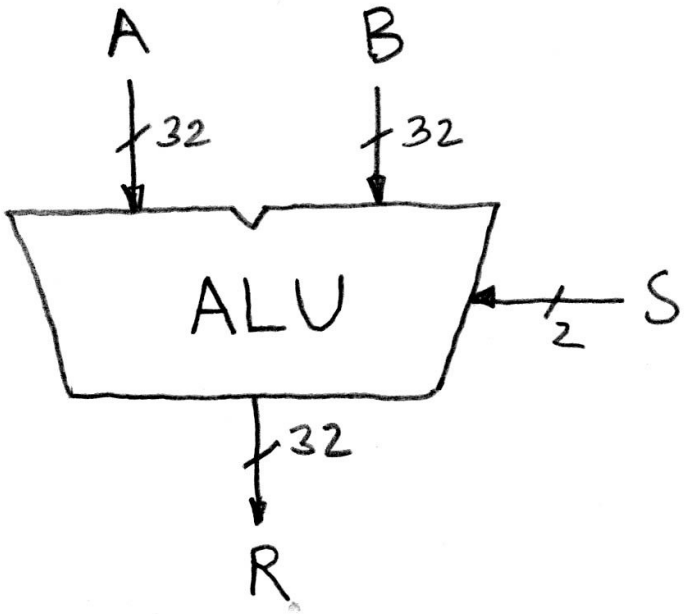
- **Final Midterm-Material Questions**
  - **Mean: 40**
  - **Standard Deviation: 20**
  - **You got a 60, one σ above**
  - **Your new midterm score is now mean + σ = 45 + 14 = 59 (~ double your old score)!**

Administrivia

- Any administrivia?

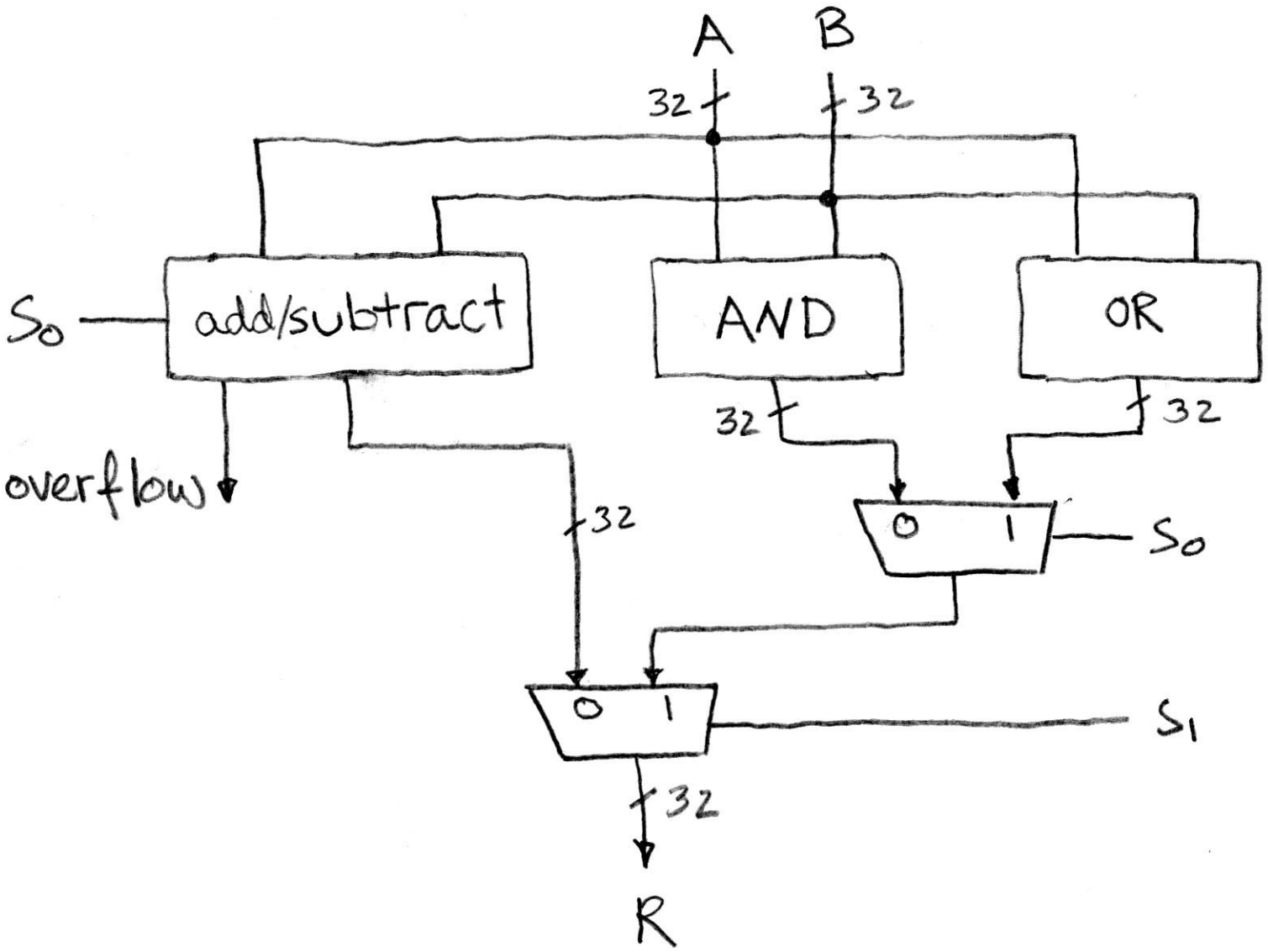
Arithmetic and Logic Unit

- **Most processors contain a special logic block called "Arithmetic and Logic Unit" (ALU)**
- **We'll show you an easy one that does ADD, SUB, bitwise AND, bitwise OR**



when S=00, R=A+B
when S=01, R=A-B
when S=10, R=A AND B
when S=11, R=A OR B

Our simple ALU



Adder/Subtracter Design -- how?

- Truth-table, then determine canonical form, then minimize and implement as we've seen before
- Look at breaking the problem down into smaller pieces that we can cascade or hierarchically layer

Adder/Subtracter – One-bit adder LSB…

$$
\begin{array}{cccc|c}
 & a_3 & a_2 & a_1 & a_0 \\
+ & b_3 & b_2 & b_1 & b_0 \\
\hline
 & s_3 & s_2 & s_1 & s_0
\end{array}
$$

a_0	b_0	s_0	c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$s_0 =$$

$$c_1 =$$

Adder/Subtracter – One-bit adder (1/2)…

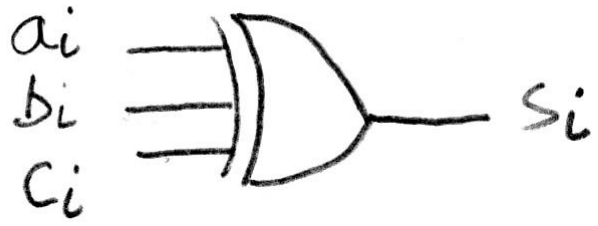
$$
\begin{array}{cccccc}
 & a_3 & a_2 & \boxed{a_1} & a_0 \\
+ & b_3 & b_2 & \boxed{b_1} & b_0 \\
\hline
 & s_3 & s_2 & \boxed{s_1} & s_0
\end{array}
$$

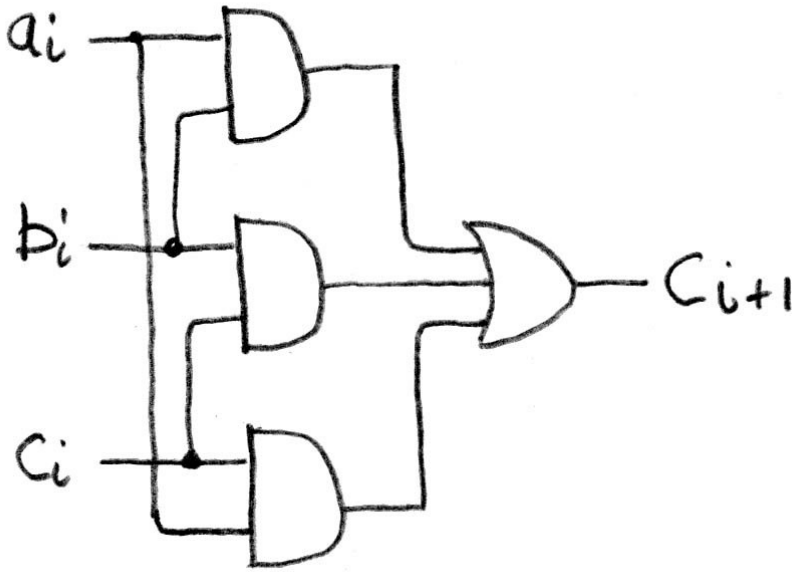
a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i =$$

$$c_{i+1} =$$

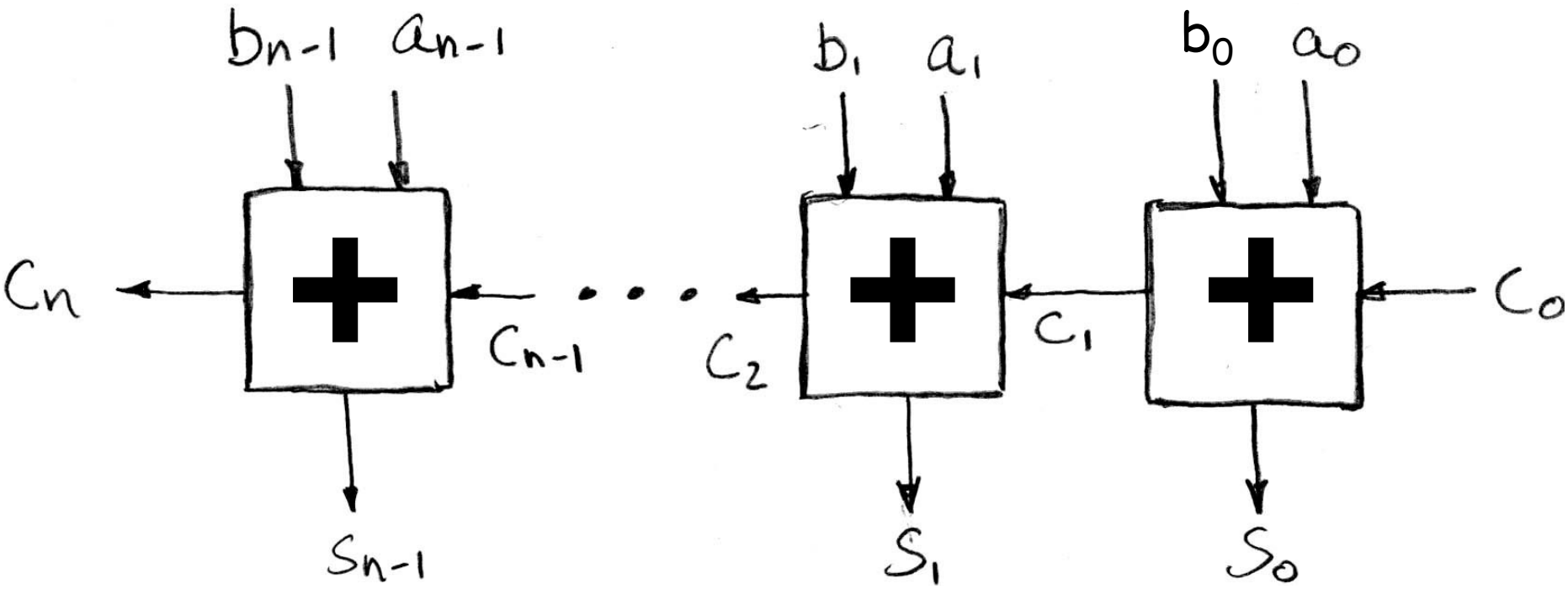
Adder/Subtracter – One-bit adder (2/2)…





$$s_i = \mathrm{XOR}(a_i, b_i, c_i)$$
$$c_{i+1} = \mathrm{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i$$

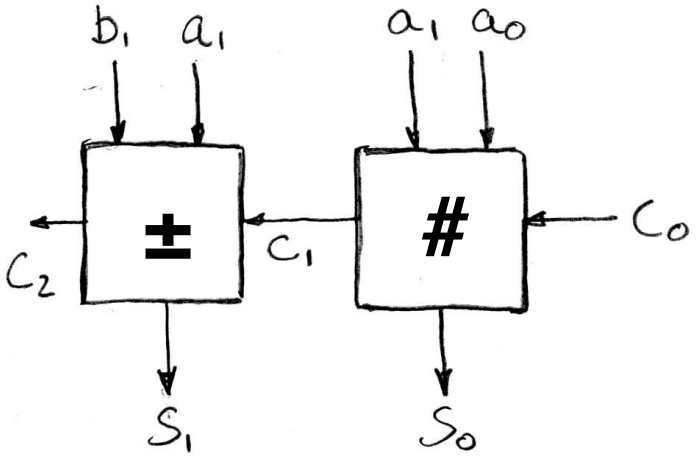
N 1-bit adders ⇒ 1 N-bit adder



What about overflow?
Overflow = c_n?

What about overflow?

- Consider a 2-bit signed # & overflow:
  - 10 = -2 + -2 or -1
  - 11 = -1 + -2 only
  - 00 = 0 NOTHING!
  - 01 = 1 + 1 only



- Highest adder
  - C_1 = Carry-in = C_{in}, C_2 = Carry-out = C_{out}
  - No C_{out} or C_{in} ⇒ NO overflow!
  - C_{in}, and C_{out} ⇒ NO overflow!

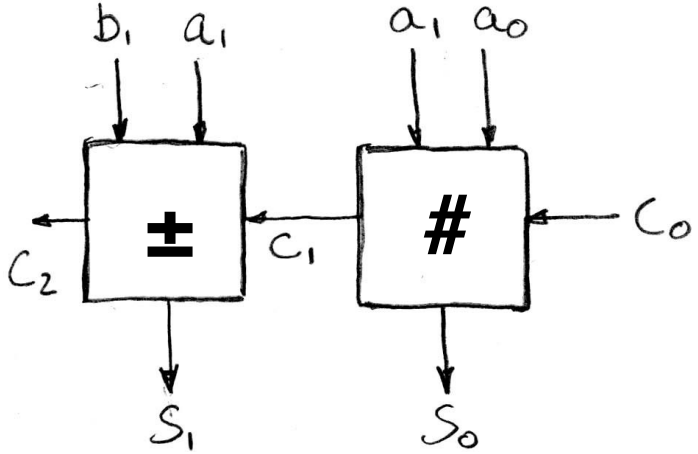
What op?

- C_{in}, but no C_{out} ⇒ A,B both > 0, overflow!
- C_{out}, but no C_{in} ⇒ A,B both < 0, overflow!

What about overflow?

- Consider a 2-bit signed # & overflow:

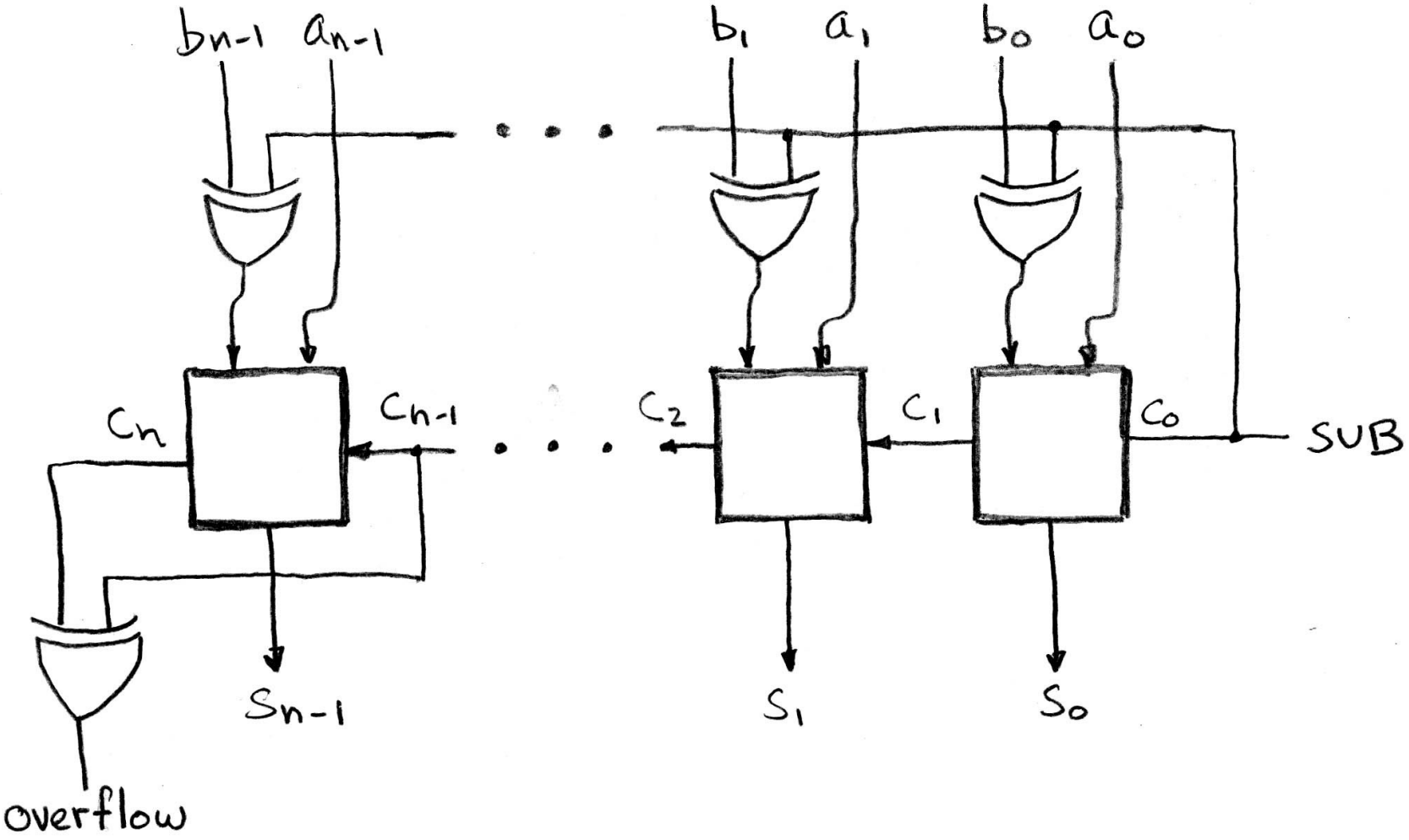
```
10 = -2
11 = -1
00 =  0
01 =  1
```



- Overflows when…
  - C_{in}, but no C_{out} ⇒ A,B both > 0, overflow!
  - C_{out}, but no C_{in} ⇒ A,B both < 0, overflow!

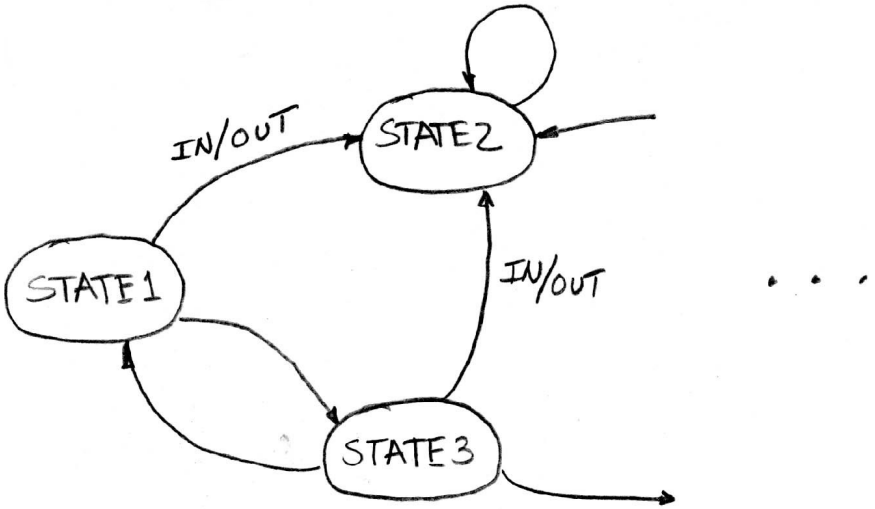
$$\text{overflow} = c_n \text{ XOR } c_{n-1}$$

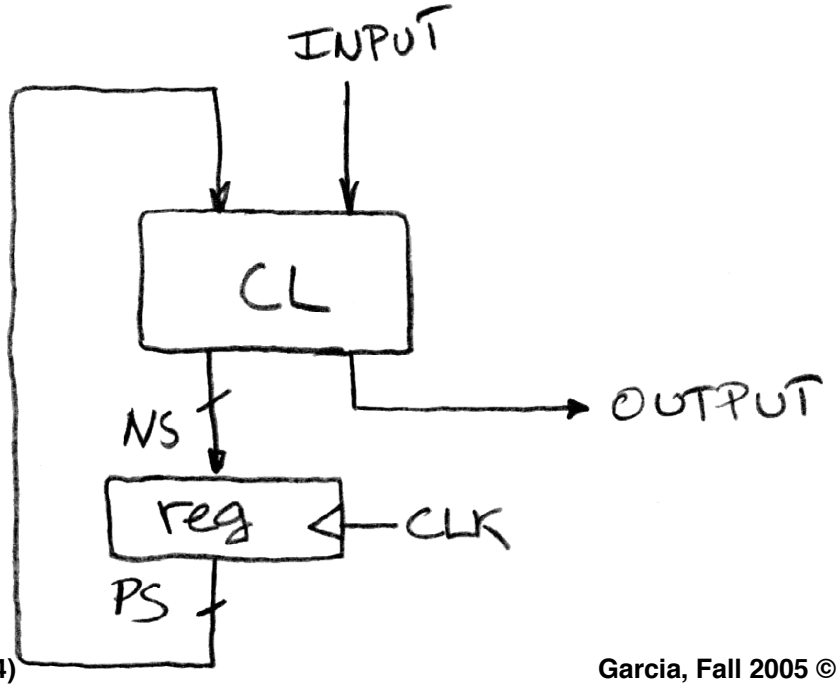
Extremely Clever Subtractor



Review: Finite State Machine (FSM)

- **States** represent possible output values.
- **Transitions** represent changes between states based on inputs.
- **Implement** with CL and clocked register feedback.





Finite State Machines extremely useful!

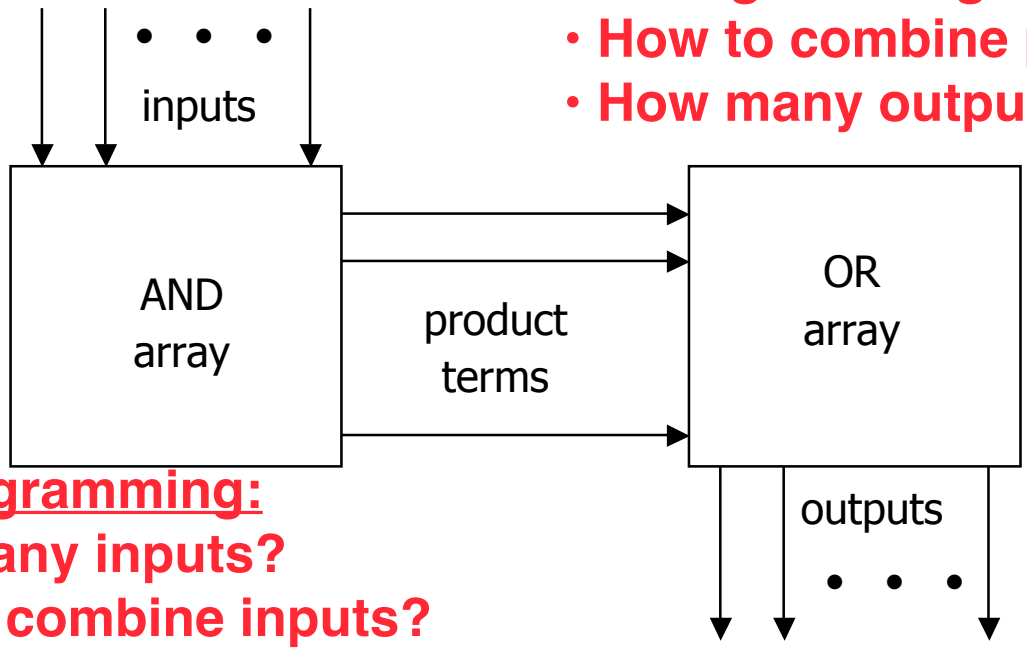
- They define
  - How output signals respond to input signals and previous state.
  - How we change states depending on input signals and previous state
- We could implement very detailed FSMs w/Programmable Logic Arrays

Taking advantage of sum-of-products

- Since sum-of-products is a convenient notation and way to think about design, offer hardware building blocks that match that notation
- One example is Programmable Logic Arrays (PLAs)
- Designed so that can select (program) ands, ors, complements after you get the chip
  - Late in design process, fix errors, figure out what to do later, …

Programmable Logic Arrays

- **Pre-fabricated building block of many AND/OR gates**
  - **"Programmed" or "Personalized" by making or breaking connections among gates**
  - **Programmable array block diagram for sum of products form**



Or Programming:
- **How to combine product terms?**
- **How many outputs?**

And Programming:
- **How many inputs?**
- **How to combine inputs?**
- **How many product terms?**

Enabling Concept

- **Shared product terms among outputs**

example:

F0 = A + B' C'
F1 = A C' + A B
F2 = B' C' + A B
F3 = B' C + A

input side: 3 inputs
1 = uncomplemented in term
0 = complemented in term
– = does not participate

personality matrix

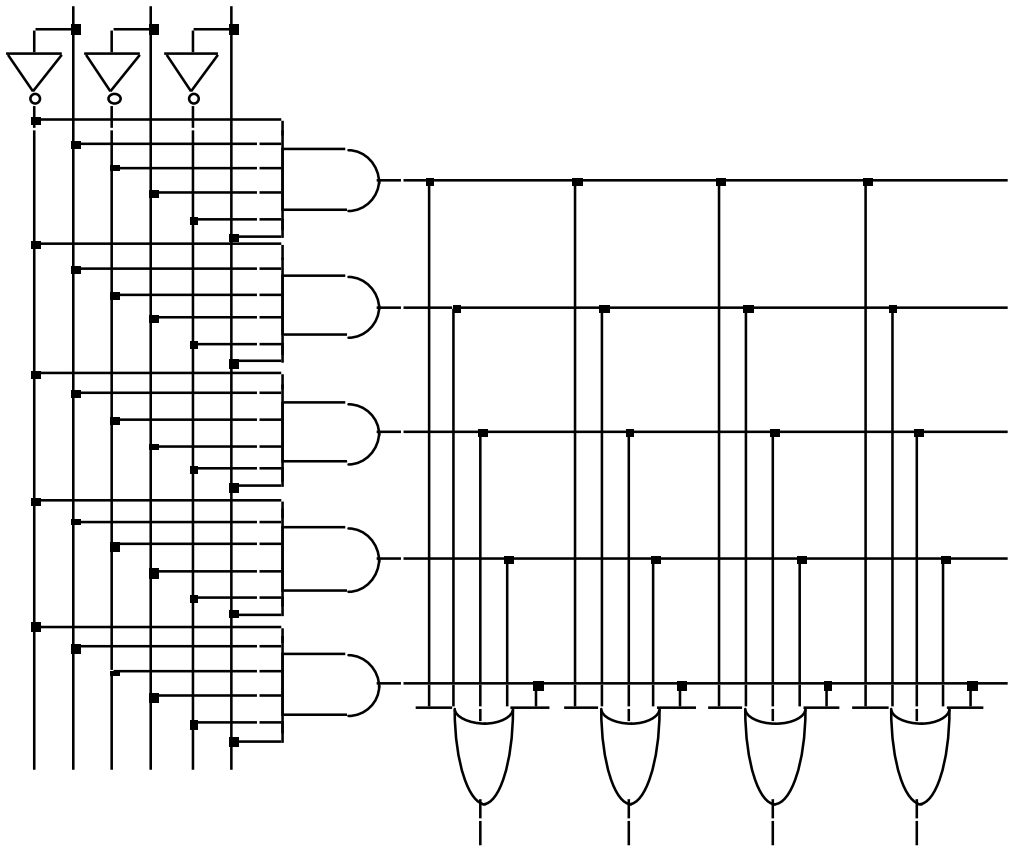
Product term	inputs A	B	C	outputs F0	F1	F2	F3
AB	1	1	–	0	1	1	0
B'C	–	0	1	0	0	0	1
AC'	1	–	0	0	1	0	0
B'C'	–	0	0	1	0	1	0
A	1	–	–	1	0	0	1

output side: 4 outputs
1 = term connected to output
0 = no connection to output

reuse of terms;
5 product terms

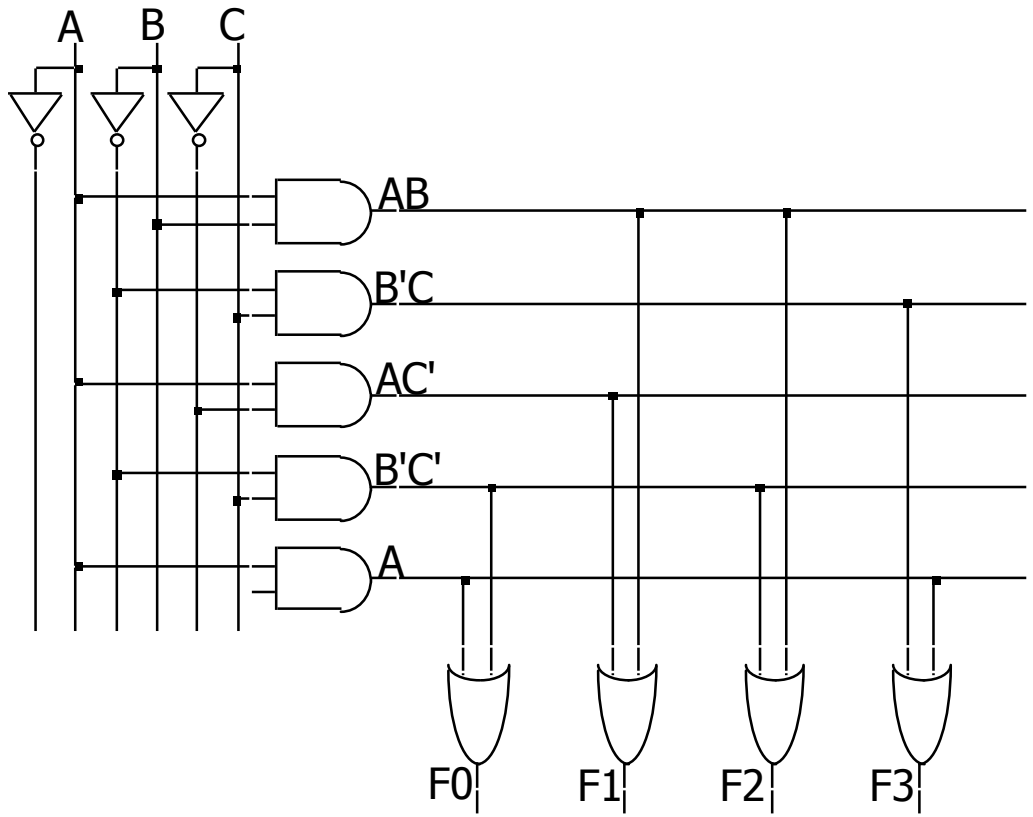
Before Programming

- All possible connections available before "programming"



After Programming

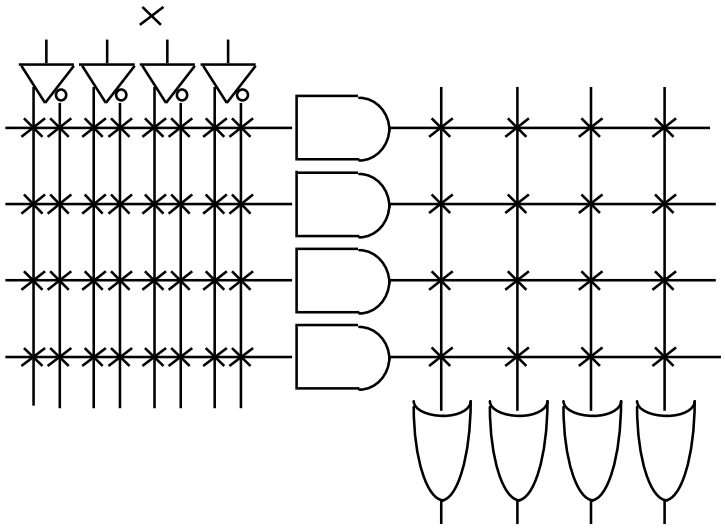
- Unwanted connections are "blown"
  - Fuse (normally connected, break unwanted ones)
  - Anti-fuse (normally disconnected, make wanted connections)

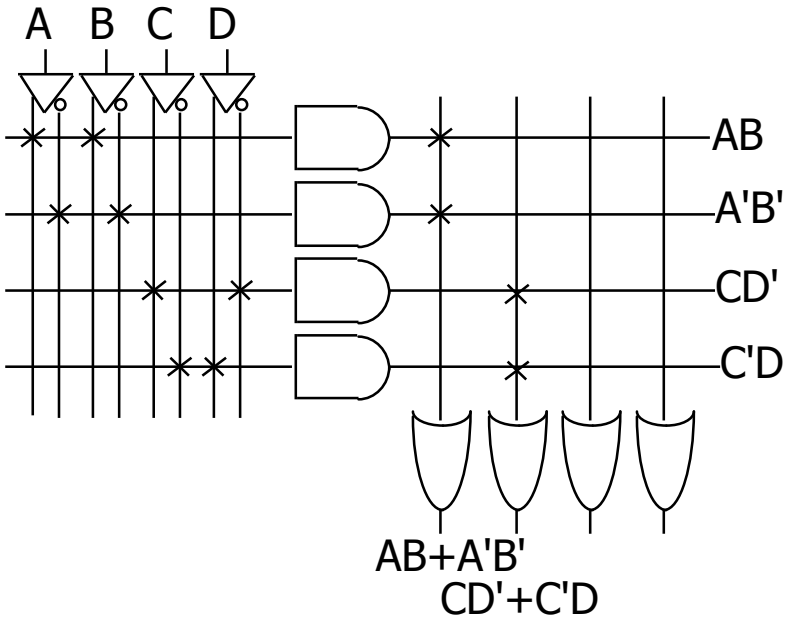


Alternate Representation

- **Short-hand notation--don't have to draw all the wires**
  - **X Signifies a connection is present and perpendicular signal is an input to gate**

notation for implementing
F0 = A B + A' B'
F1 = C D' + C' D





"And In conclusion…"

- Use muxes to select among input
  - S input bits selects 2^S inputs
  - Each input can be n-bits wide, indep of S
- Implement muxes hierarchically
- ALU can be implemented using a mux
  - Coupled with basic block elements
- N-bit adder-subtractor done using N 1-bit adders with XOR gates on input
  - XOR serves as conditional inverter
- Programmable Logic Arrays are often used to implement our CL

Peer Instruction

A. SW can peek at HW (past ISA abstraction boundary) for optimizations

B. SW can depend on particular HW implementation of ISA

C. Timing diagrams serve as a critical debugging tool in the EE toolkit

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

Peer Instruction

A. HW feedback akin to SW recursion

B. We can implement a D-Q flipflop as simple CL (And, Or, Not gates)

C. You can build a FSM to signal when an equal number of 0s and 1s has appeared in the input.

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

Peer Instruction

A. $(a+b) \cdot (\overline{a}+b) = b$

B. N-input gates can be thought of cascaded 2-input gates. I.e.,
$(a \Delta bc \Delta d \Delta e) = a \Delta (bc \Delta (d \Delta e))$
where Δ is one of AND, OR, XOR, NAND

C. You can use NOR(s) with clever wiring to simulate AND, OR, & NOT

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

Peer Instruction

A. Truth table for mux with 4-bits of signals has 2^4 rows

B. We could cascade N 1-bit shifters to make 1 N-bit shifter for sll, srl

C. If 1-bit adder delay is T, the N-bit adder delay would also be T

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT