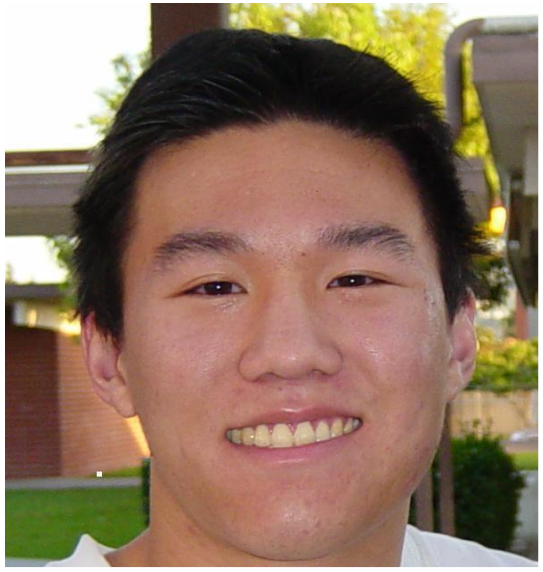
inst.eecs.berkeley.edu/~cs61c

CS61CL : Machine Structures

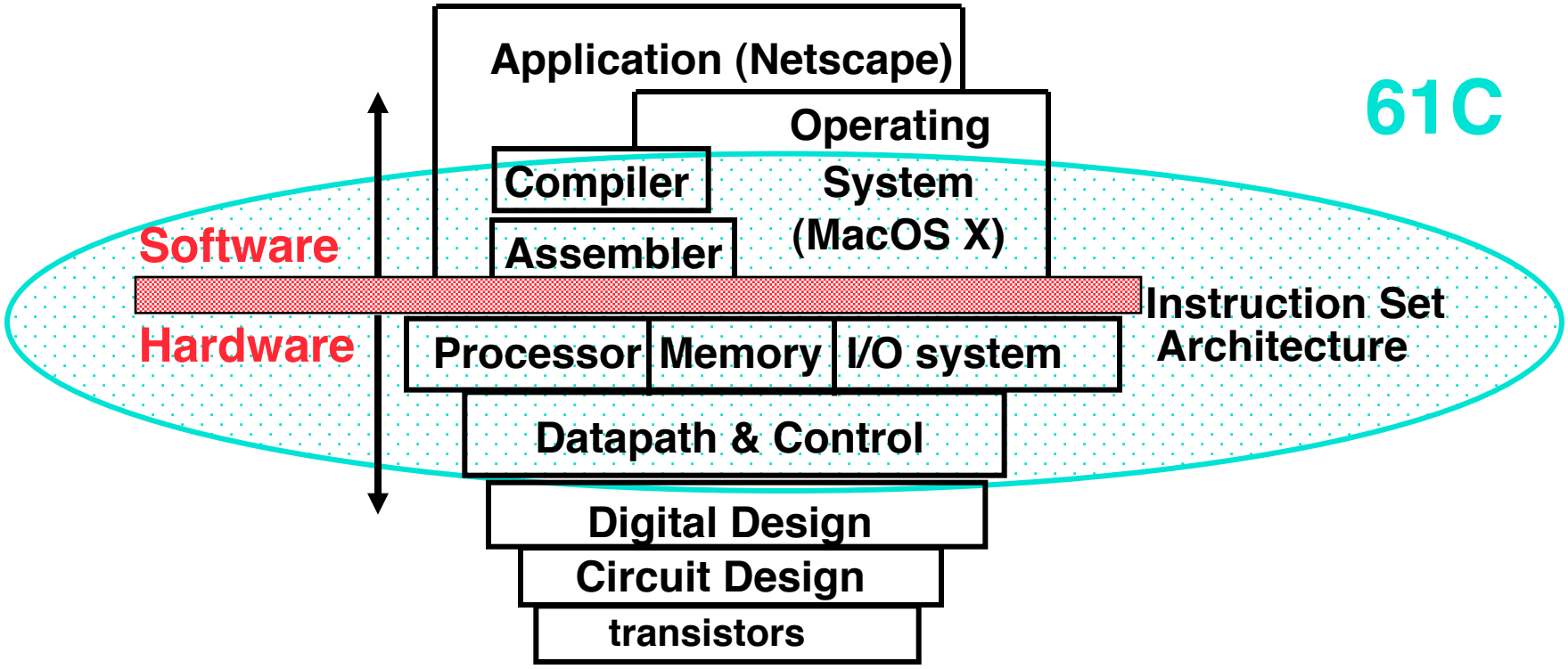
Lecture #8 – State Elements, Combinational Logic

2009-07-15



James Tu, TA

What are "Machine Structures"?



Coordination of many *levels of abstraction*

ISA is an important abstraction level: contract between HW & SW

61C Levels of Representation

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g.,MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (Logic, Logisim, etc.)

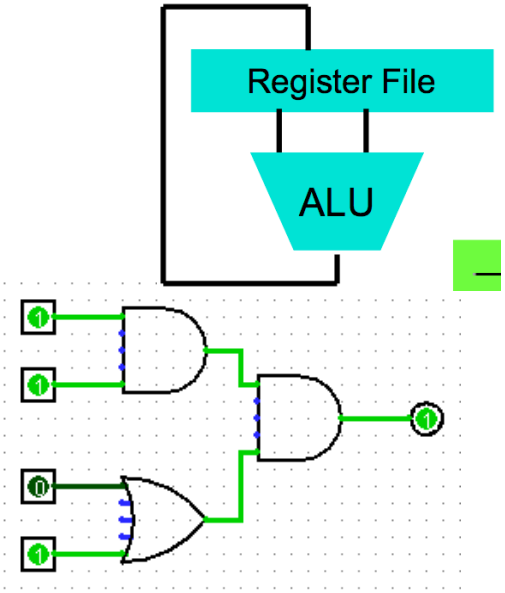
Architecture Implementation

Logic Circuit Description (Logisim, etc.)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



Synchronous Digital Systems

The hardware of a processor, such as the MIPS, is an example of a Synchronous Digital System

Synchronous:

- **Means all operations are coordinated by a central clock.**
  - **It keeps the "heartbeat" of the system!**

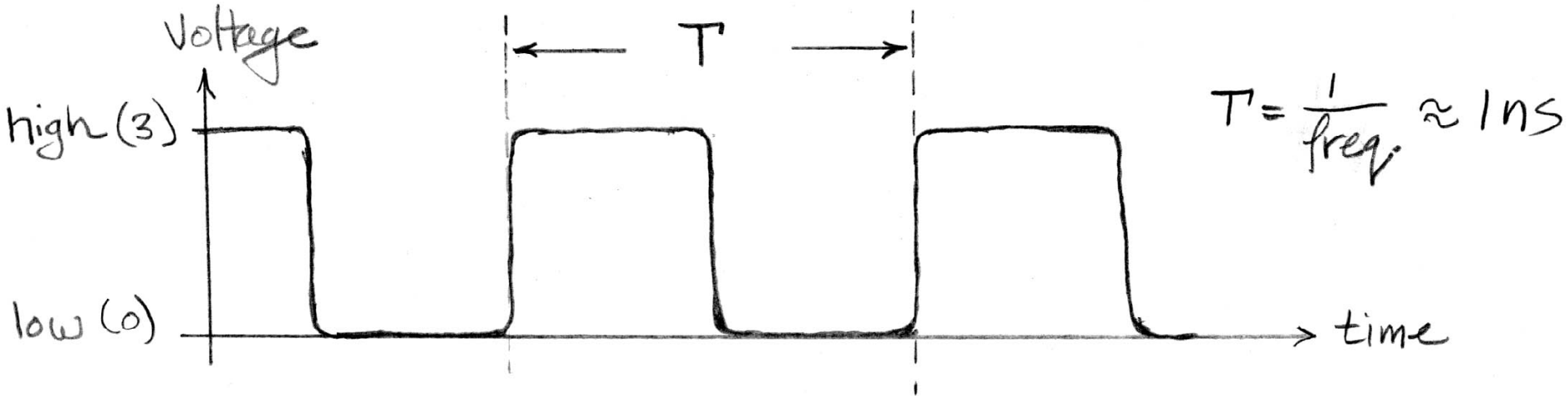
Digital:

- **Mean all values are represented by discrete values**
- **Electrical signals are treated as 1's and 0's and grouped together to form words.**

Logic Design

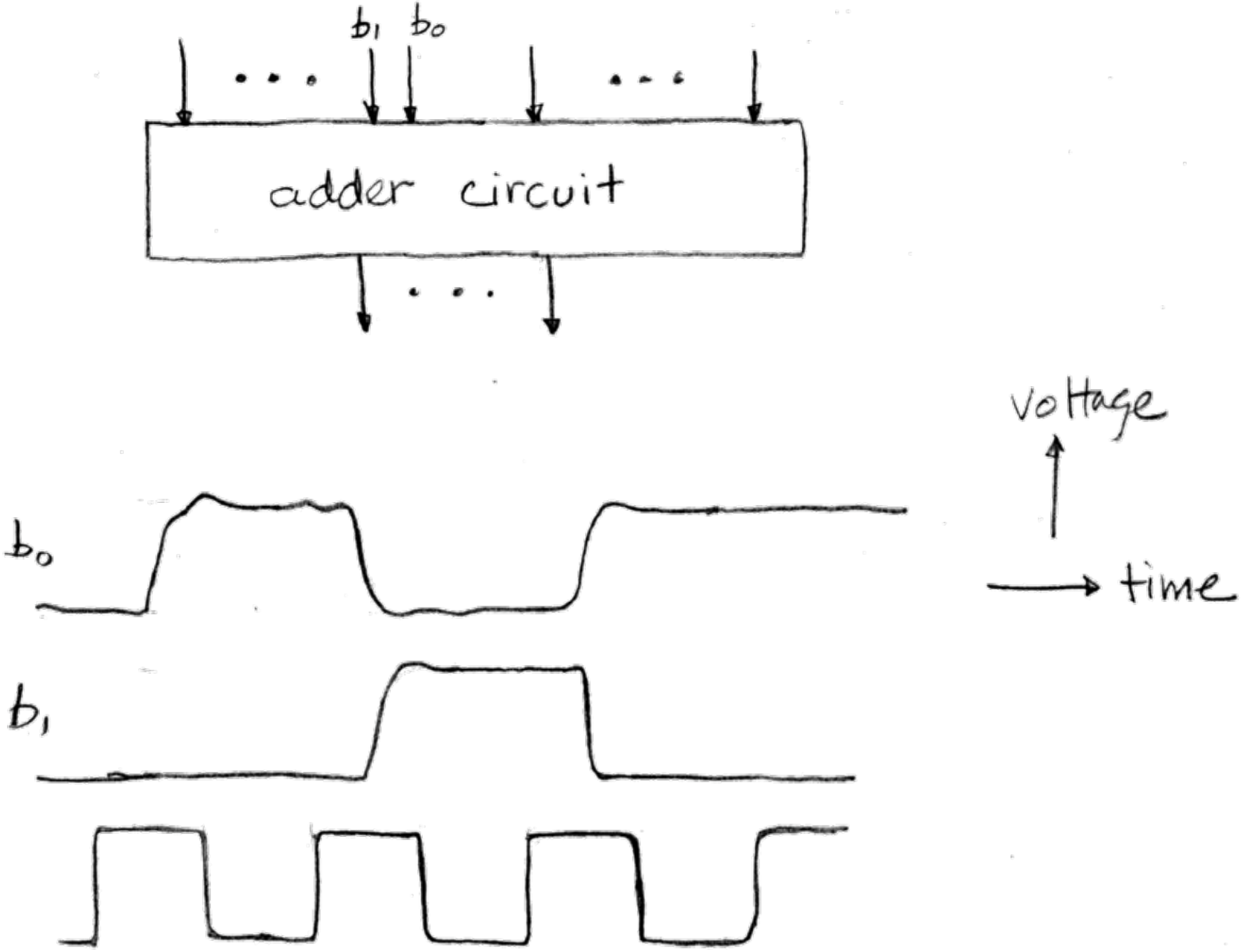
- Next 2 weeks: we'll study how a modern processor is built; starting with basic elements as building blocks.
- Why study hardware design?
  - Understand capabilities and limitations of hardware in general and processors in particular.
  - What processors can do fast and what they can't do fast (avoid slow things if you want your code to run fast!)
  - Background for more detailed hardware courses (CS 150, CS 152, EE 192)
  - There is just so much you can do with processors. At some point you may need to design your own custom hardware.

Signals and Waveforms: Clocks

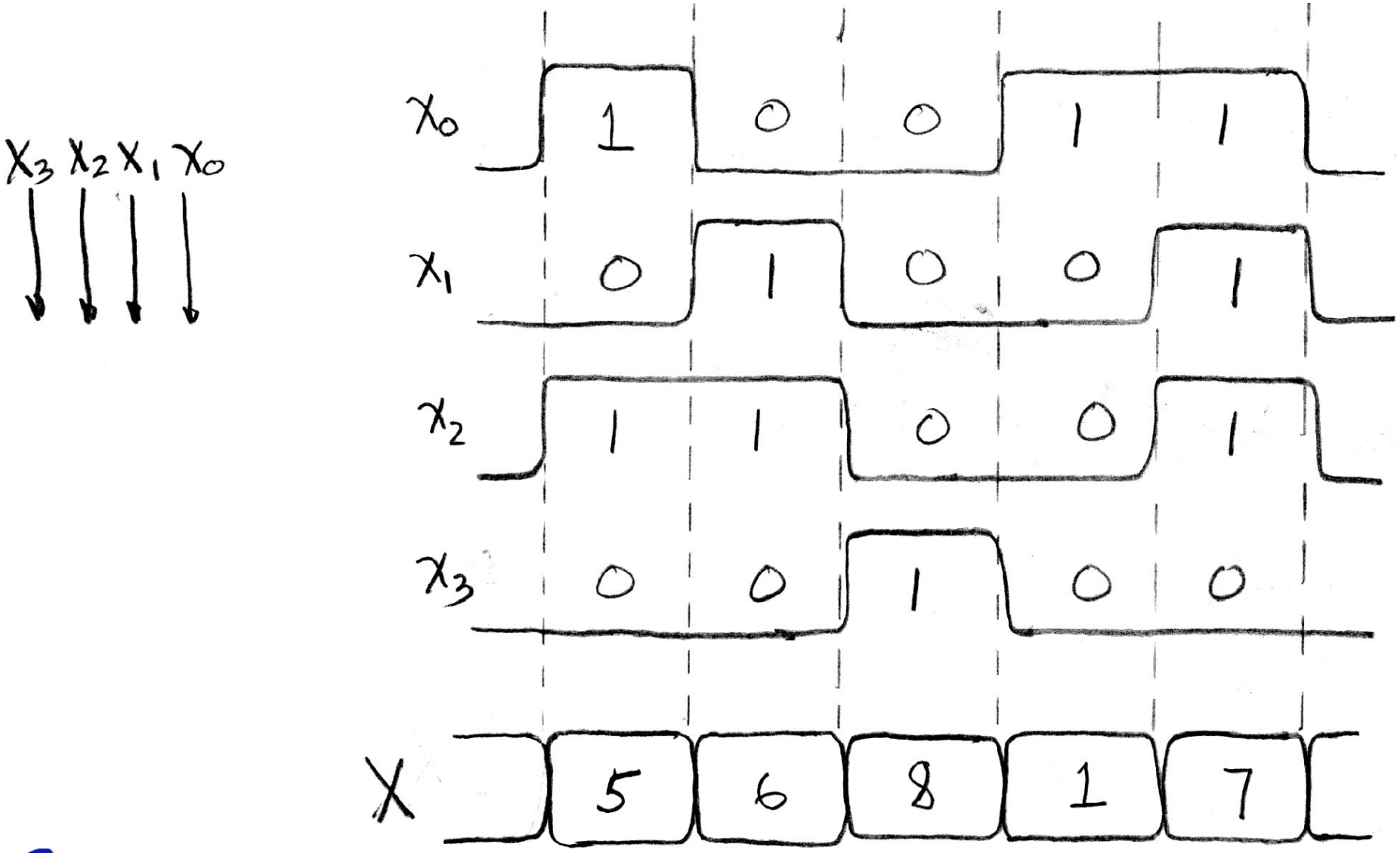


- **Signals**
  - **When digital is only treated as 1 or 0**
  - **Is transmitted over wires continuously**
  - **Transmission is effectively instant**
    - **Implies that any wire only contains 1 value at a time**

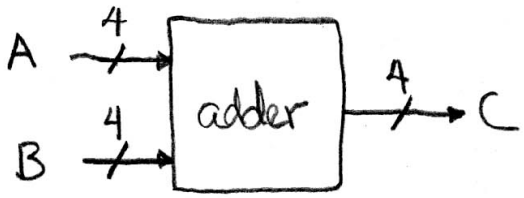
Signals and Waveforms



Signals and Waveforms: Grouping

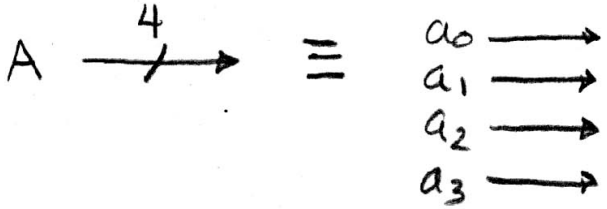


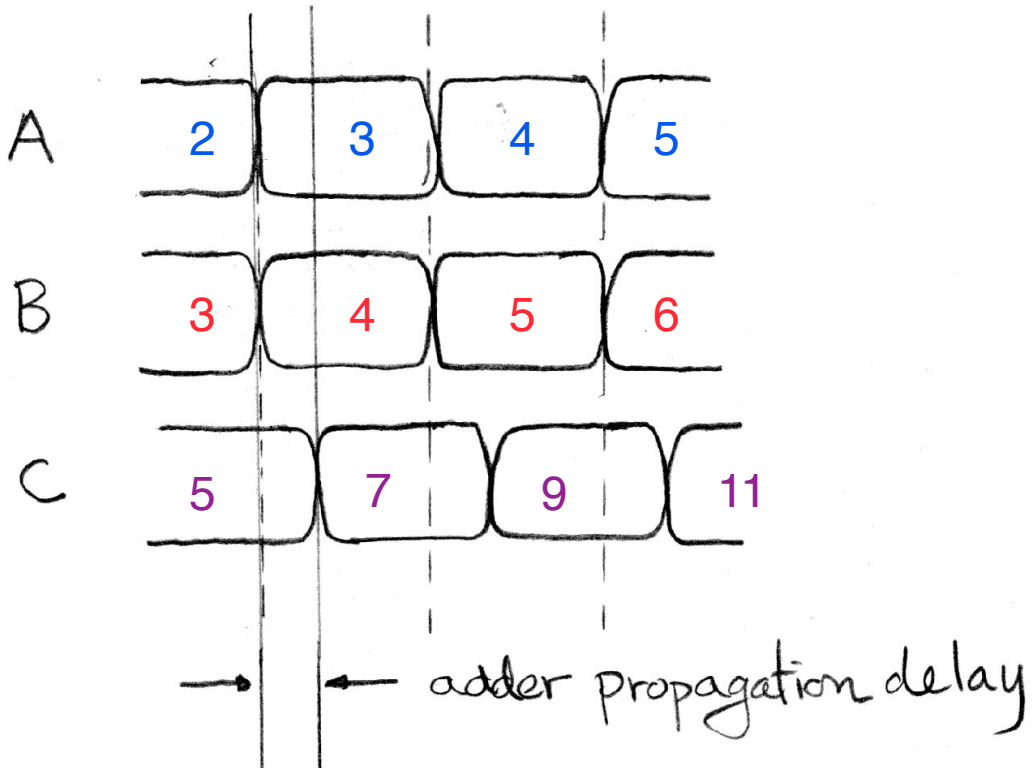
Signals and Waveforms: Circuit Delay



$A = [a_3, a_2, a_1, a_0]$

$B = [b_3, b_2, b_1, b_0]$





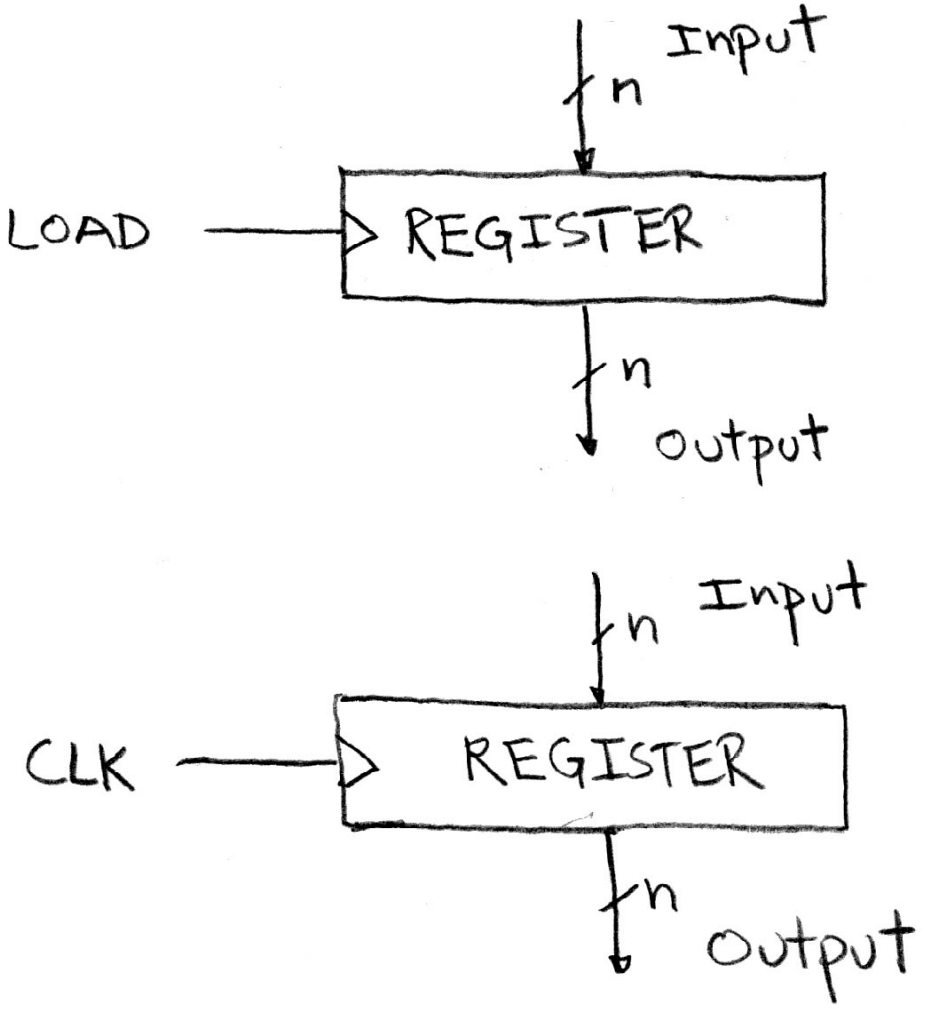
Type of Circuits

- Synchronous Digital Systems are made up of two basic types of circuits:
- Combinational Logic (CL) circuits
  - Our previous adder circuit is an example.
  - Output is a function of the inputs only.
  - Similar to a pure function in mathematics, y = f(x). (No way to store information from one invocation to the next. No side effects)
- State Elements: circuits that store information.

Uses for State Elements

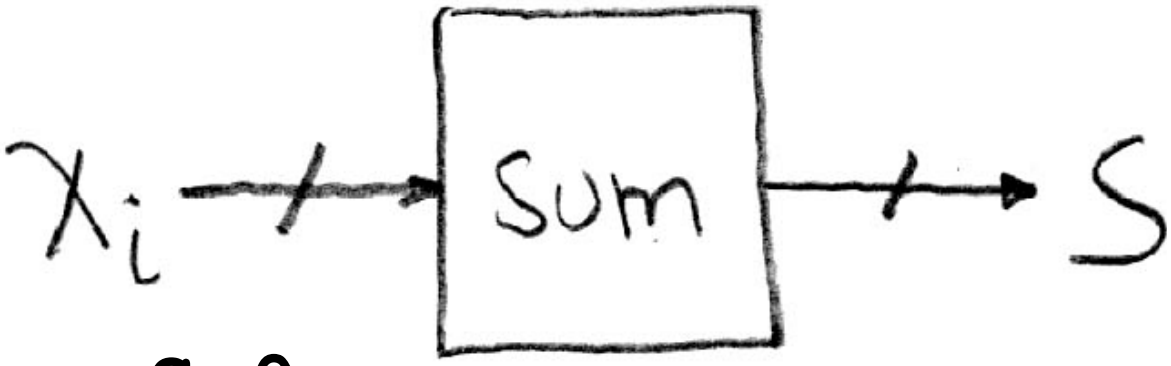
1. As a place to store values for some indeterminate amount of time:
   - Register files (like $1-$31 on the MIPS)
   - Memory (caches, and main memory)
2. Help control the flow of information between combinational logic blocks.
   - State elements are used to hold up the movement of information at the inputs to combinational logic blocks and allow for orderly passage.

Circuits with STATE (e.g., register)



Accumulator Example

Why do we need to control the flow of information?



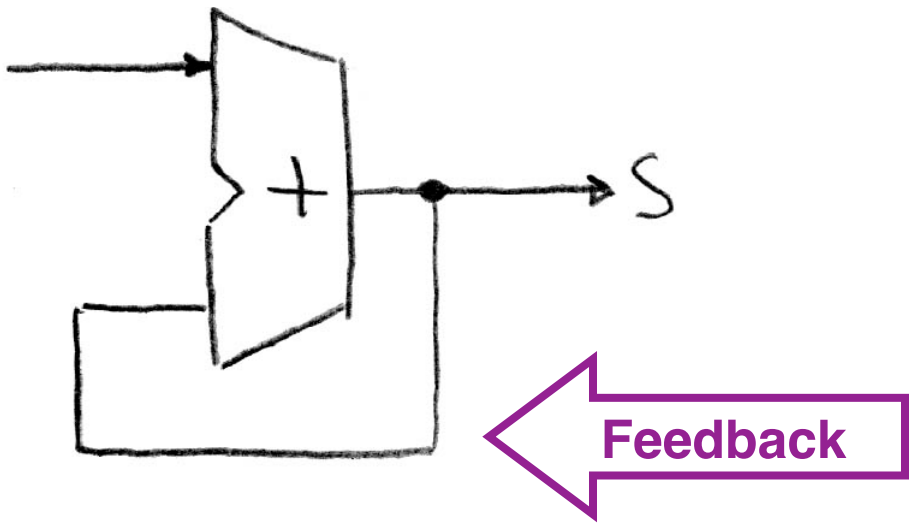
Want:

```
S=0;
for (i=0;i<n;i++)
  S = S + Xi
```

Assume:

- **Each X value is applied in succession, one per cycle.**
- **After n cycles the sum is present on S.**

First try…Does this work?

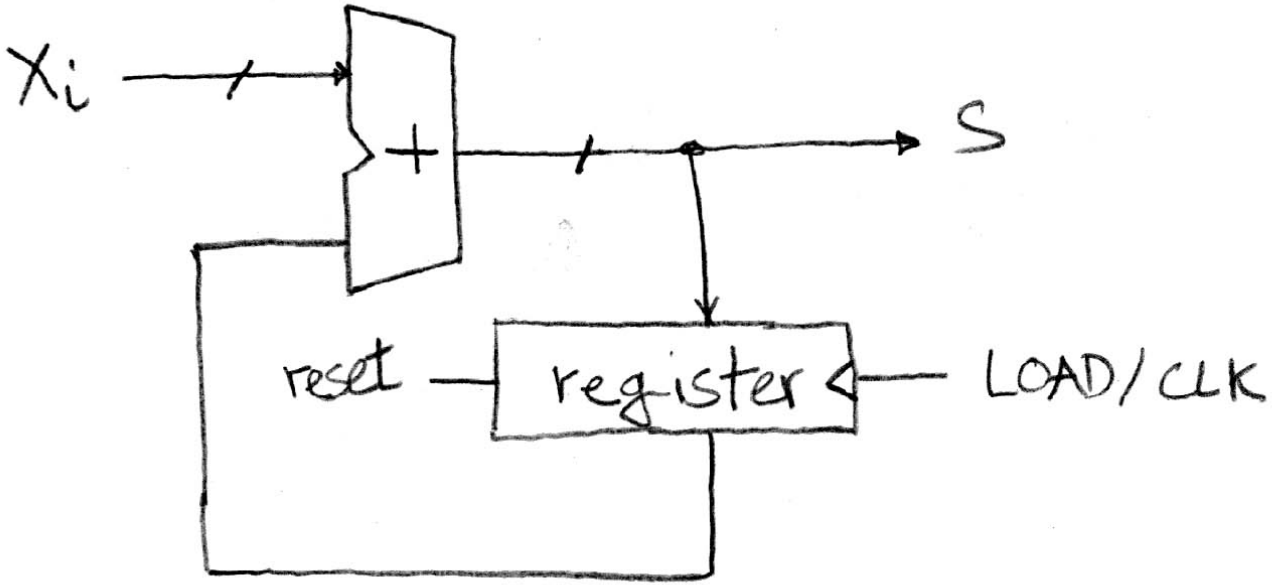


Nope!

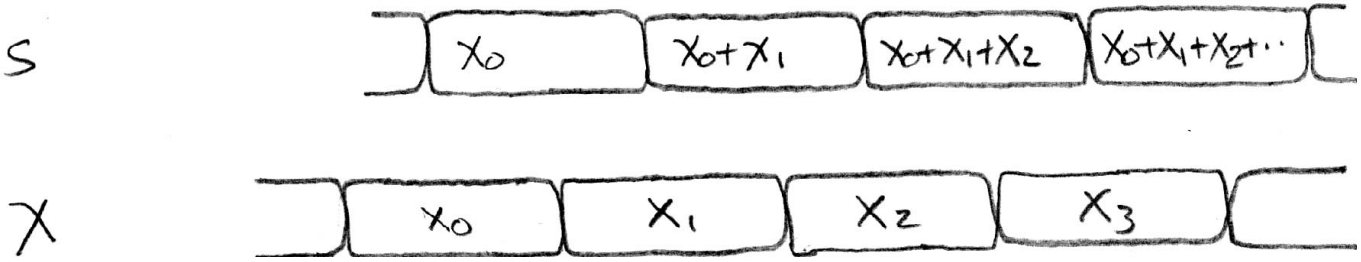
Reason #1… What is there to control the next iteration of the ‘`for`’ loop?

Reason #2… How do we say: ‘`S=0`’?

Second try…How about this?

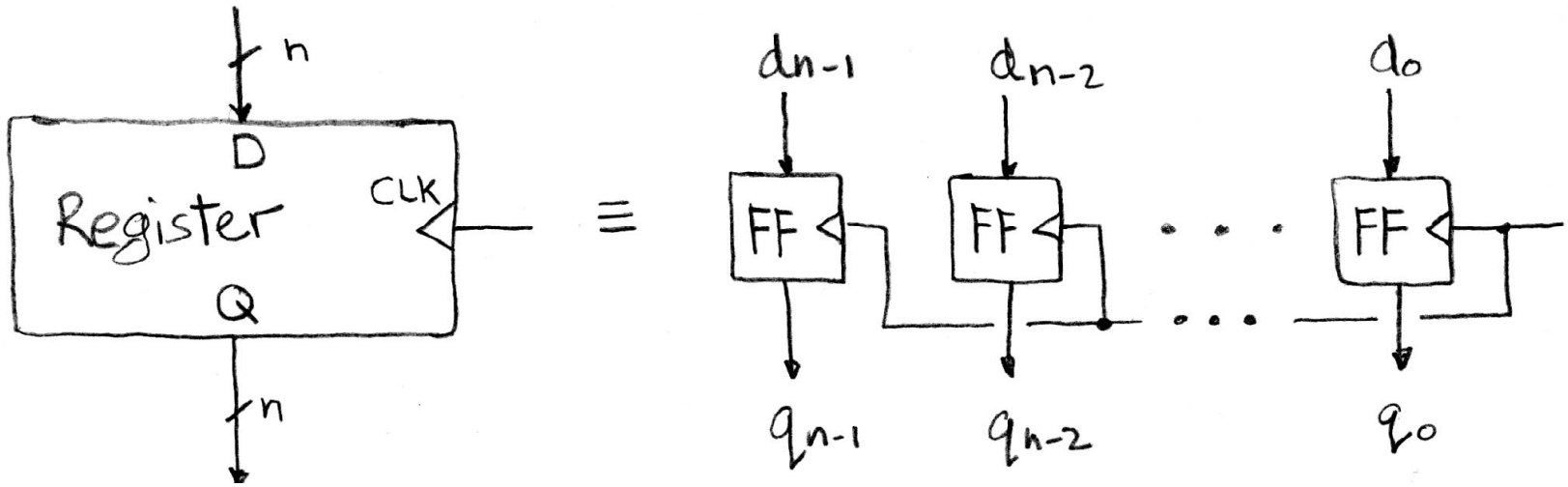


Rough timing…



Register is used to hold up the transfer of data to adder.

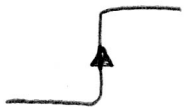
Register Details…What's inside?



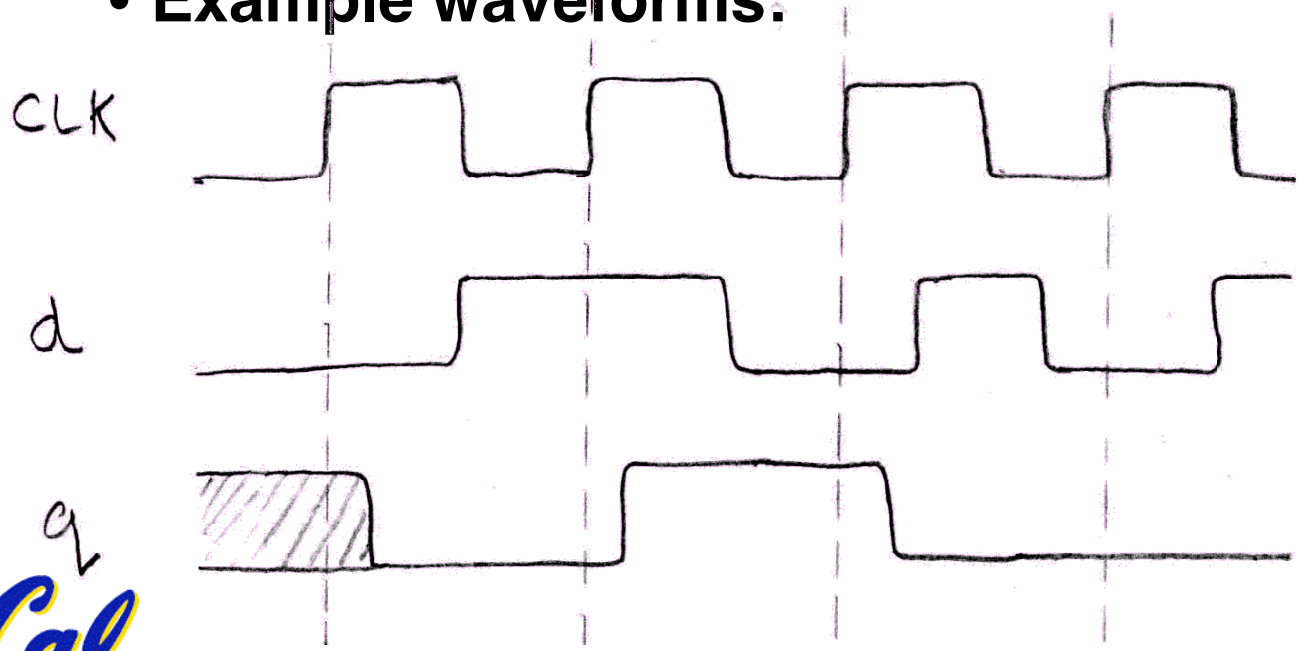
- n instances of a "Flip-Flop"
- Flip-flop name because the output flips and flops between and 0,1
- D is "data", Q is "output"
- Also called "d-type Flip-Flop"

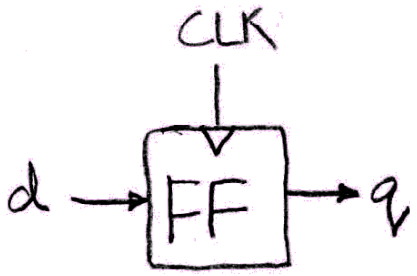
What's the timing of a Flip-flop? (1/2)

- **Edge-triggered d-type flip-flop**
  - **This one is "positive edge-triggered"**



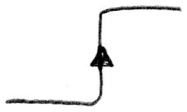
- **"On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored."**
- **Example waveforms:**



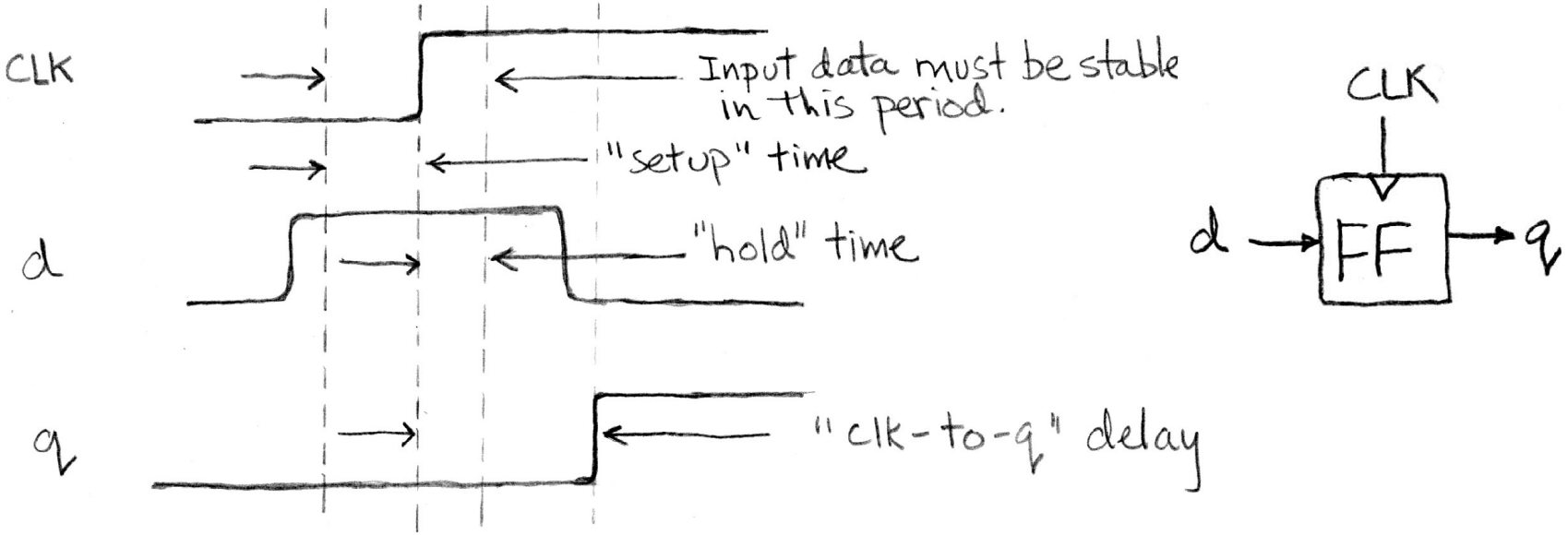


What's the timing of a Flip-flop? (2/2)

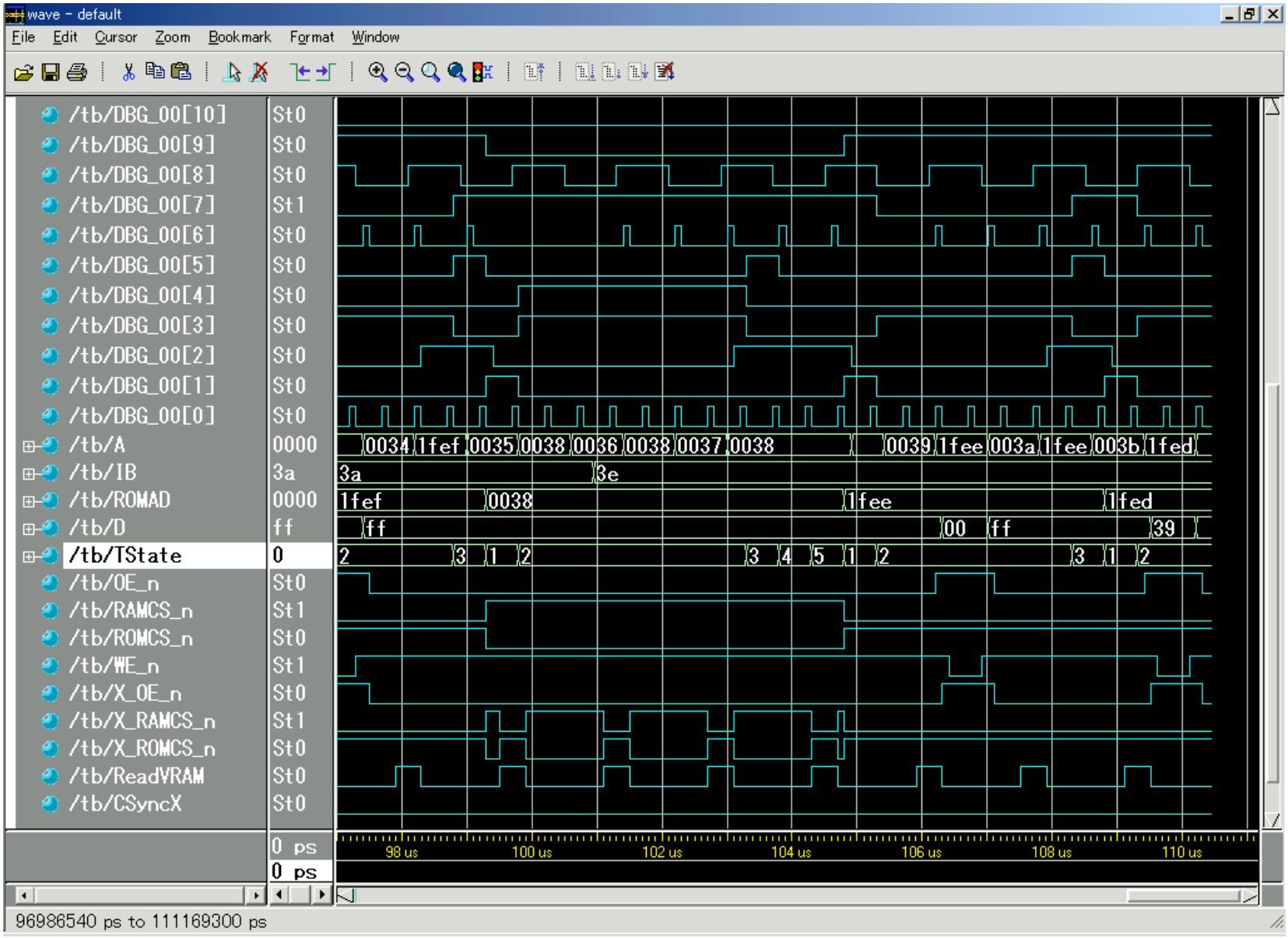
- **Edge-triggered d-type flip-flop**
  - **This one is "positive edge-triggered"**



- **"On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored."**
- **Example waveforms (more detail):**



Sample Debugging Waveform

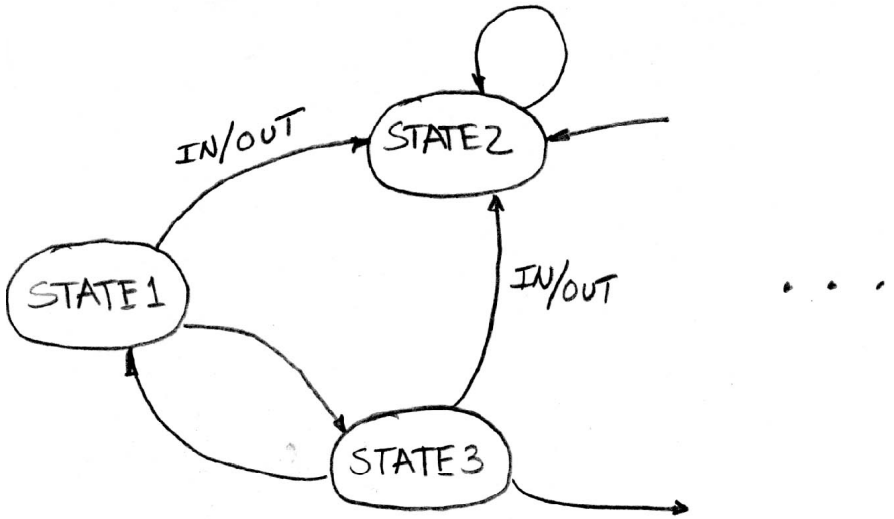


Recap of Timing Terms

- **Clock (CLK)** - steady square wave that synchronizes system
- **Setup Time** - when the input must be stable before the rising edge of the CLK
- **Hold Time** - when the input must be stable after the rising edge of the CLK
- **"CLK-to-Q" Delay** - how long it takes the output to change, measured from the rising edge
- **Flip-flop** - one bit of state that samples every rising edge of the CLK
- **Register** - several bits of state that samples on rising edge of CLK or on LOAD

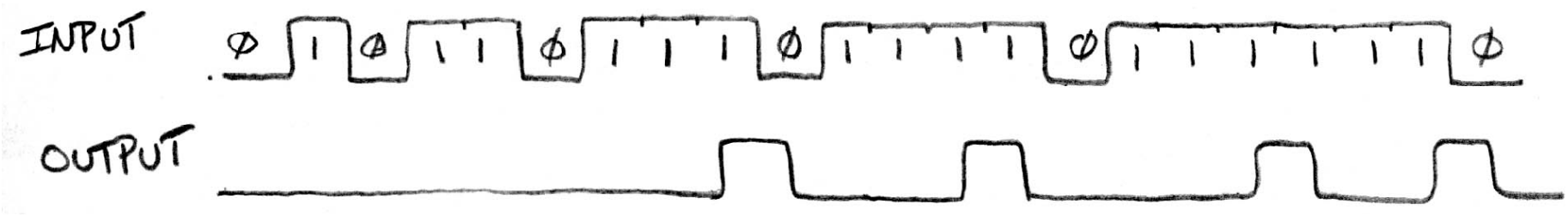
Finite State Machines (FSM) Introduction

- You have seen FSMs in other classes.
- Same basic idea.
- The function can be represented with a "state transition diagram".
- With combinational logic and registers, any FSM can be implemented in hardware.

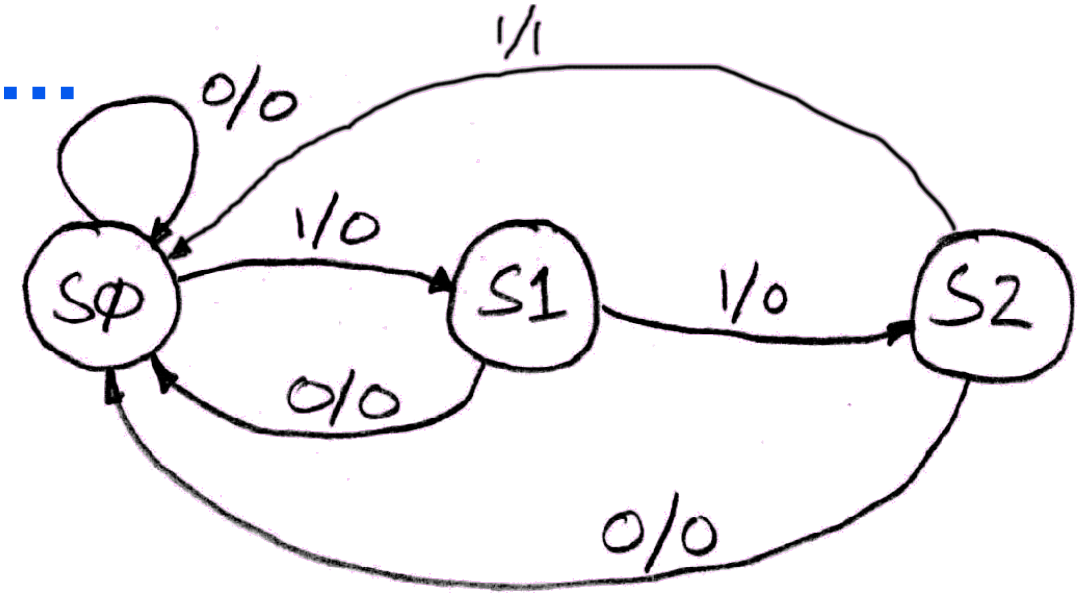


Finite State Machine Example: 3 ones…

FSM to detect the occurrence of 3 consecutive 1's in the input.



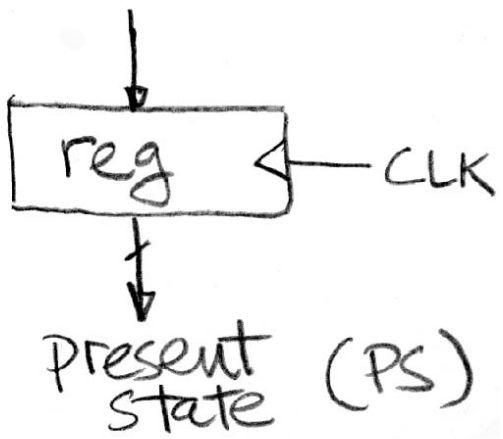
Draw the FSM…

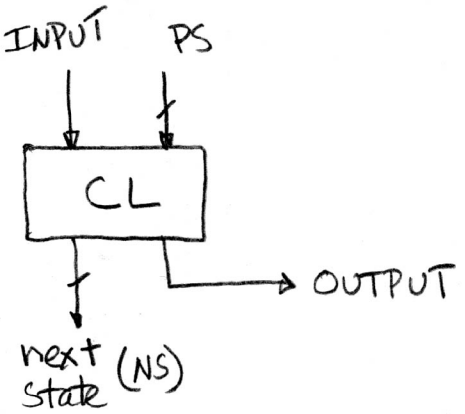


Assume state transitions are controlled by the clock:
on each clock cycle the machine checks the inputs and moves to a new state and produces a new output…

Hardware Implementation of FSM

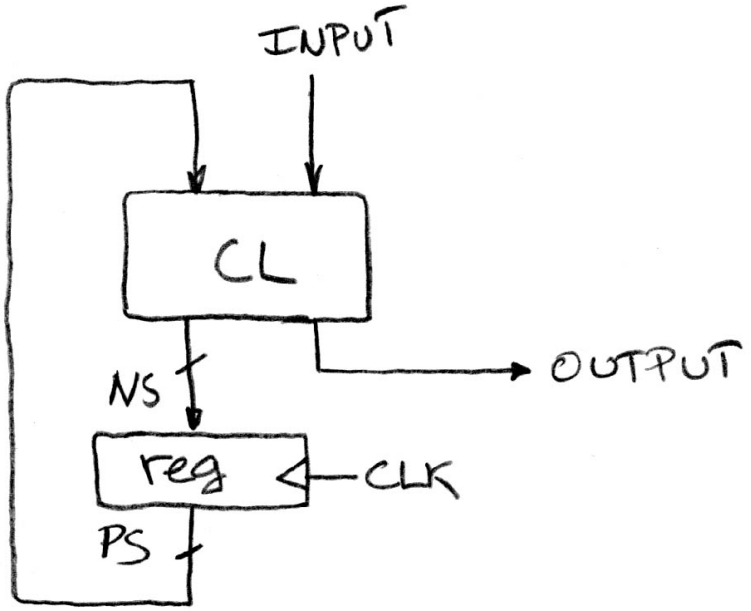
… Therefore a register is needed to hold the a representation of which state the machine is in. Use a unique bit pattern for each state.





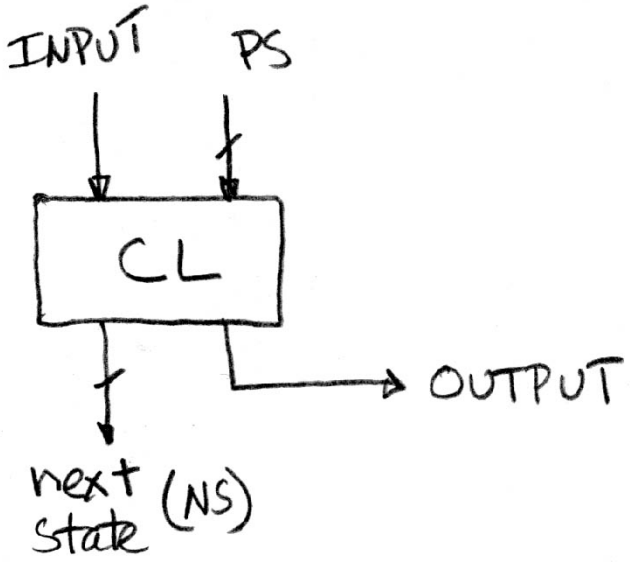
Combinational logic circuit is used to implement a function maps from *present state and input* to *next state and output.*

=



Hardware for FSM: Combinational Logic

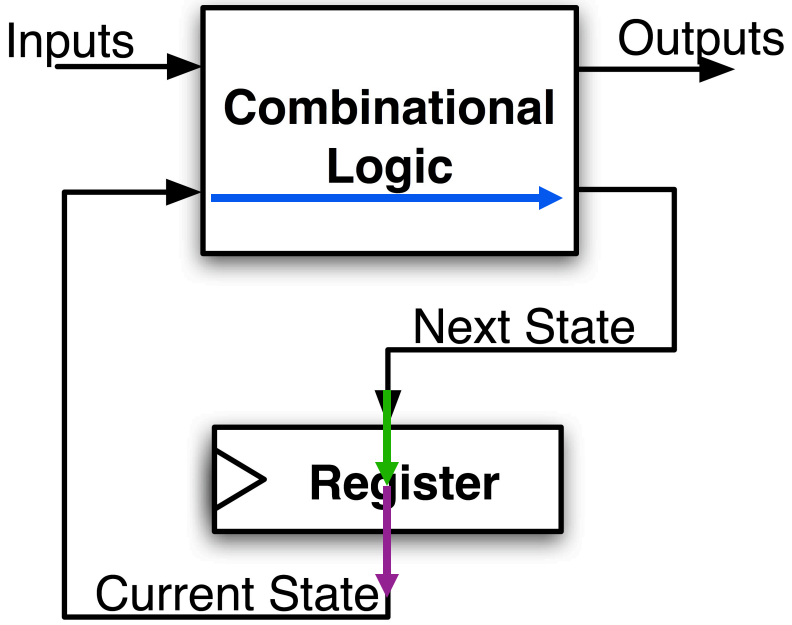
Later in today's lecture, we will discuss the detailed implementation, but for now can look at its functional specification, truth table form.



Truth table…

PS	Input	NS	Output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

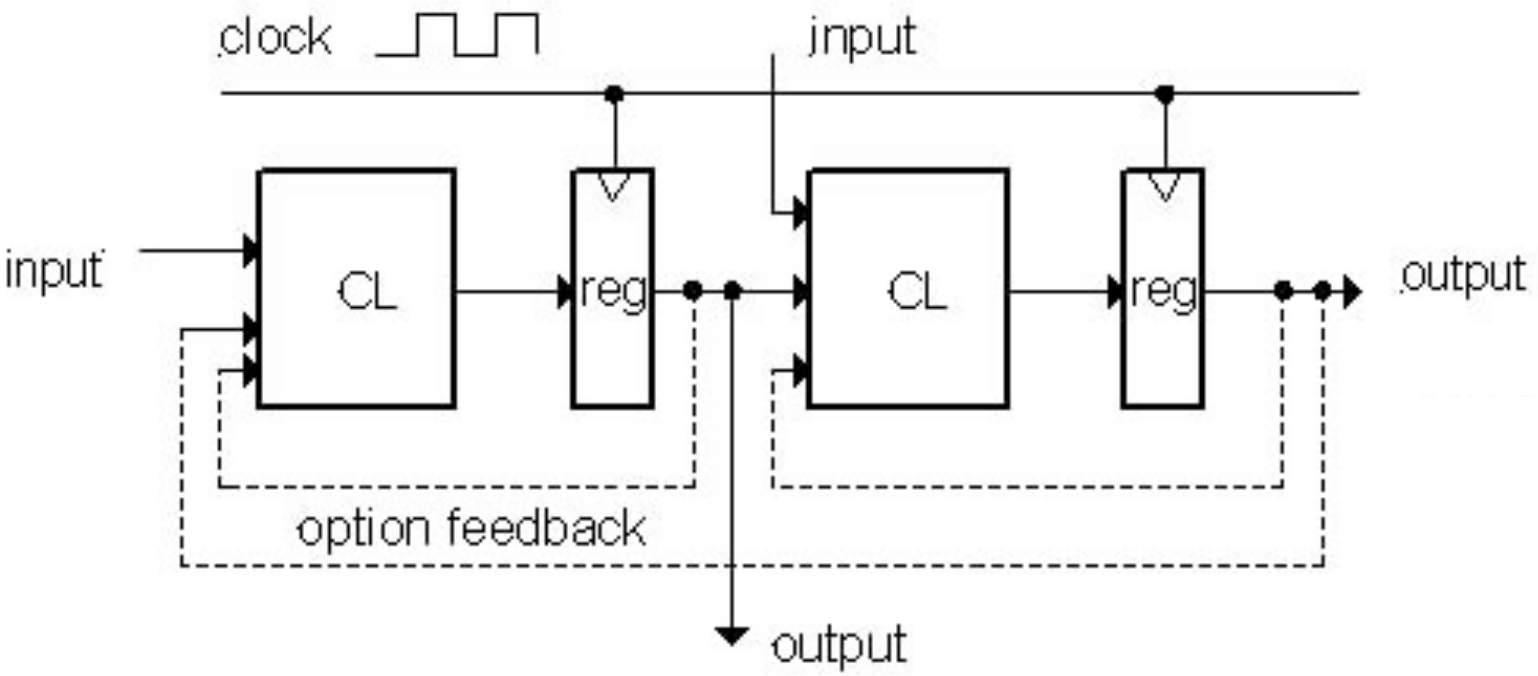
Maximum Clock Frequency



- What is the maximum frequency of this circuit?

Max Delay = Setup Time + CLK-to-Q Delay + CL Delay

General Model for Synchronous Systems



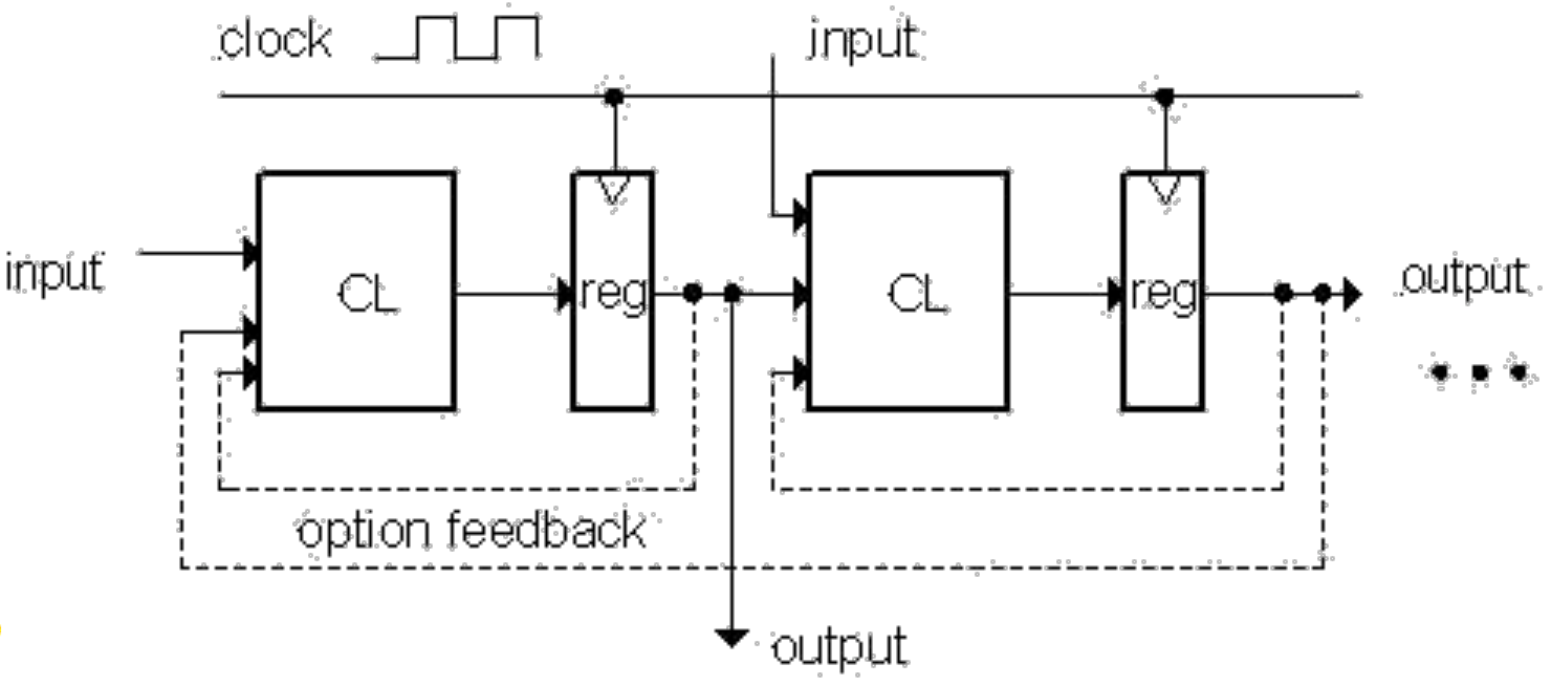
- Collection of CL blocks separated by registers.
- Registers may be back-to-back and CL blocks may be back-to-back.
- Feedback is optional.
- Clock signal(s) connects only to clock input of registers. (NEVER put it through a gate)

Administrivia

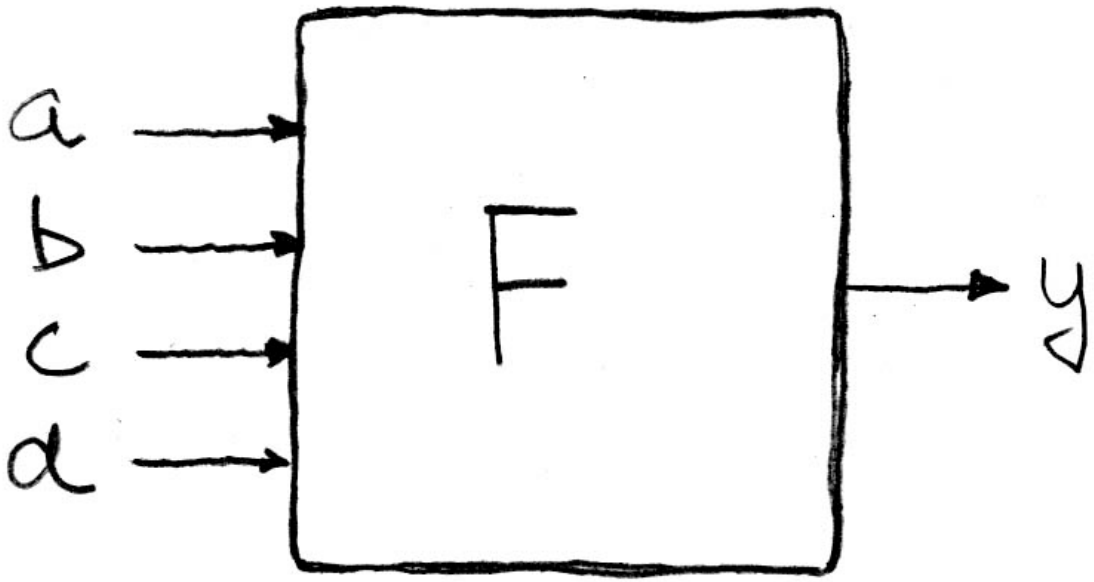
- Project 2 due Friday @ 11:59 PM
- Midterm 7/20 (Monday) in class

Combinational Logic

- FSMs had states and transitions
- How to we get from one state to the next?
- Answer: Combinational Logic



Truth Tables



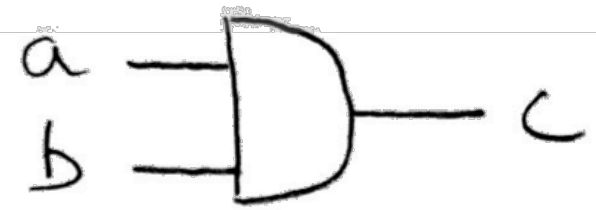
a	b	c	d	y
0	0	0	0	F(0,0,0,0)
0	0	0	1	F(0,0,0,1)
0	0	1	0	F(0,0,1,0)
0	0	1	1	F(0,0,1,1)
0	1	0	0	F(0,1,0,0)
0	1	0	1	F(0,1,0,1)
0	1	1	0	F(0,1,1,0)
0	1	1	1	F(0,1,1,1)
1	0	0	0	F(1,0,0,0)
1	0	0	1	F(1,0,0,1)
1	0	1	0	F(1,0,1,0)
1	0	1	1	F(1,0,1,1)
1	1	0	0	F(1,1,0,0)
1	1	0	1	F(1,1,0,1)
1	1	1	0	F(1,1,1,0)
1	1	1	1	F(1,1,1,1)

TT Example #1: 1 iff one (not both) a,b=1

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

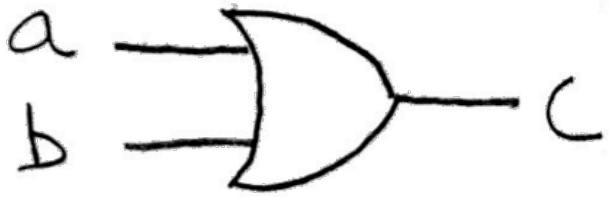
Logic Gates (1/2)

AND



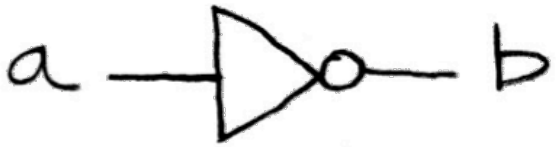
ab	c
00	0
01	0
10	0
11	1

OR



ab	c
00	0
01	1
10	1
11	1

NOT

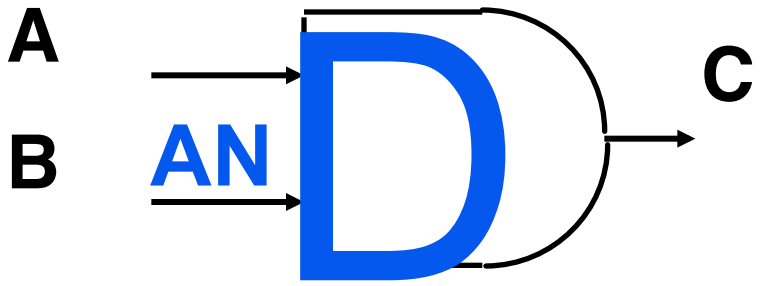


a	b
0	1
1	0

And vs. Or review – Dan's mnemonic

AND Gate

Symbol

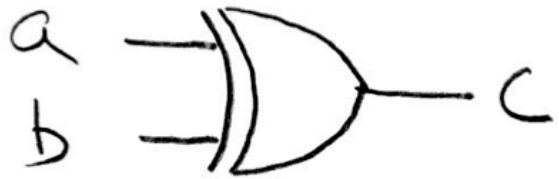


Definition

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

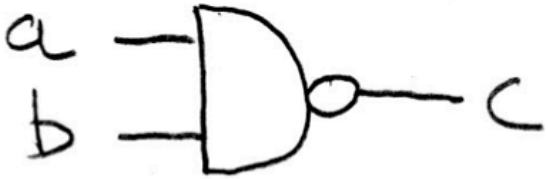
Logic Gates (2/2)

XOR



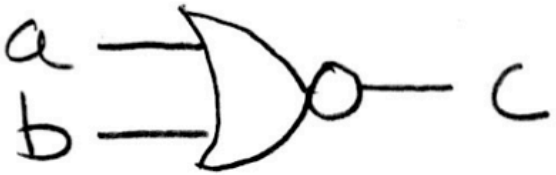
ab	c
00	0
01	1
10	1
11	0

NAND



ab	c
00	1
01	1
10	1
11	0

NOR



ab	c
00	1
01	0
10	0
11	0

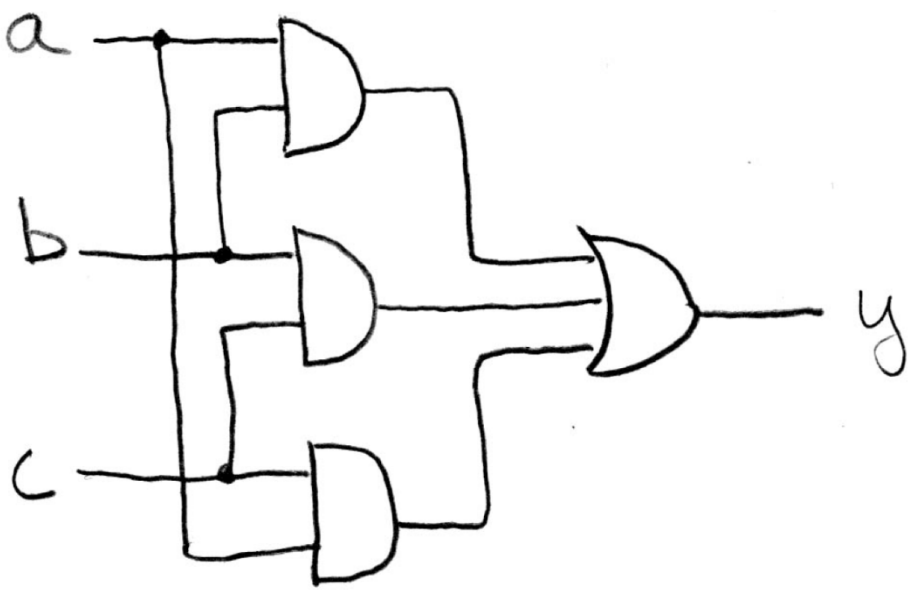
2-input gates extend to n-inputs

- **N-input XOR is the only one which isn't so obvious**
- **It's simple: XOR is a 1 iff the # of 1s at its input is odd⇒**

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Truth Table ⇒ Gates (e.g., majority circ.)

a	b	c	y
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



Boolean Algebra

- George Boole, 19th Century mathematician
- Developed a mathematical system (algebra) involving logic
  - later known as "Boolean Algebra"

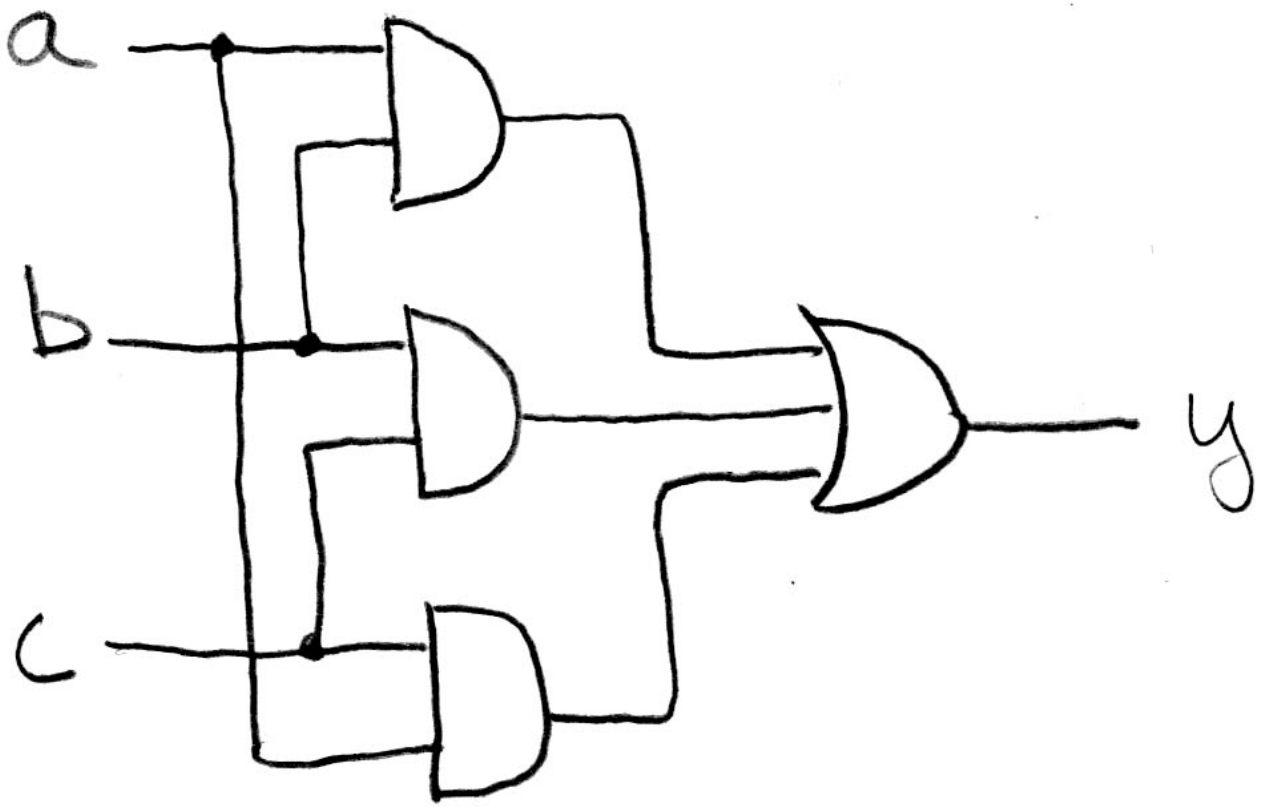
- Primitive functions: AND, OR and NOT
- The power of BA is there's a one-to-one correspondence between circuits made up of AND, OR and NOT gates and equations in BA

+ means OR, · means AND, $\bar{x}$ means NOT

Laws of Boolean Algebra

$x \cdot \overline{x} = 0$	$x + \overline{x} = 1$	complementarity
$x \cdot 0 = 0$	$x + 1 = 1$	laws of 0's and 1's
$x \cdot 1 = x$	$x + 0 = x$	identities
$x \cdot x = x$	$x + x = x$	idempotent law
$x \cdot y = y \cdot x$	$x + y = y + x$	commutativity
$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$	associativity
$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$	distribution
$xy + x = x$	$(x + y)x = x$	uniting theorem
$\overline{x \cdot y} = \overline{x} + \overline{y}$	$\overline{(x + y)} = \overline{x} \cdot \overline{y}$	DeMorgan's Law

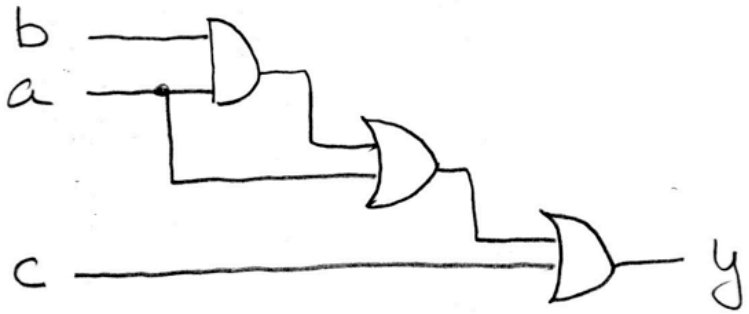
Boolean Algebra (e.g., for majority fun.)



$$y = a \cdot b + a \cdot c + b \cdot c$$

$$y = ab + ac + bc$$

BA: Circuit & Algebraic Simplification



original circuit

$$\downarrow$$

$$y = ((ab) + a) + c$$

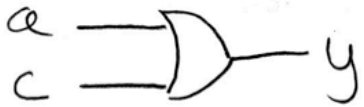
equation derived from original circuit

$$\downarrow$$

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

algebraic simplification

$$\downarrow$$



simplified circuit

BA also great for circuit verification
Circ X = Circ Y?
use BA to prove!

Boolean Algebraic Simplification Example

$$
\begin{aligned}
y &= ab + a + c \\
&= a(b + 1) + c && \textit{distribution, identity} \\
&= a(1) + c && \textit{law of 1's} \\
&= a + c && \textit{identity}
\end{aligned}
$$

Canonical form (1/2)

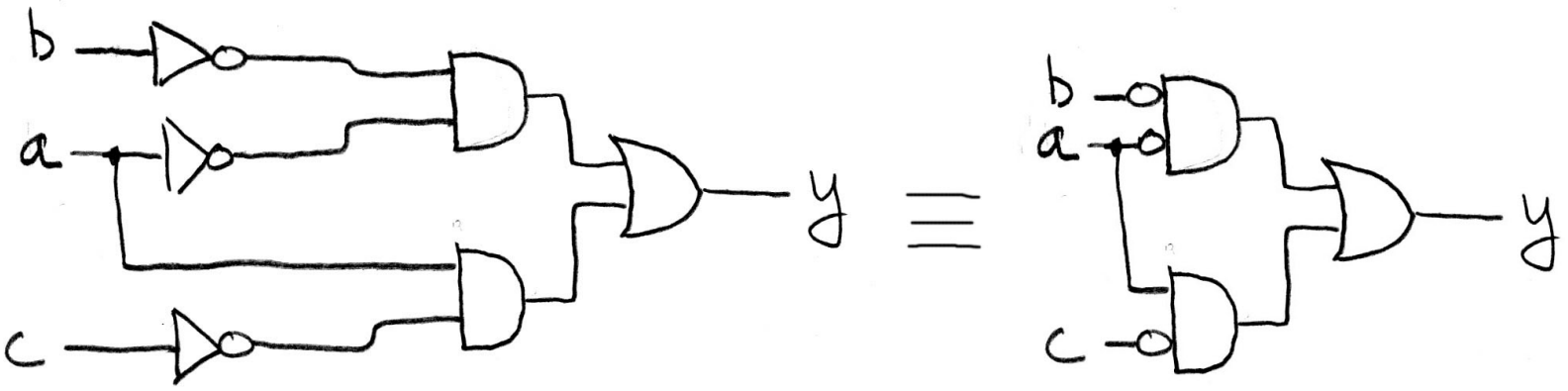
	abc	y
$\overline{a} \cdot \overline{b} \cdot \overline{c}$	000	1
$\overline{a} \cdot \overline{b} \cdot c$	001	1
	010	0
	011	0
$a \cdot \overline{b} \cdot \overline{c}$	100	1
	101	0
$a \cdot b \cdot \overline{c}$	110	1
	111	0

Sum-of-products (ORs of ANDs)

$$= \overline{a}\overline{b}\overline{c} + \overline{a}\overline{b}c + a\overline{b}\overline{c} + ab\overline{c}$$

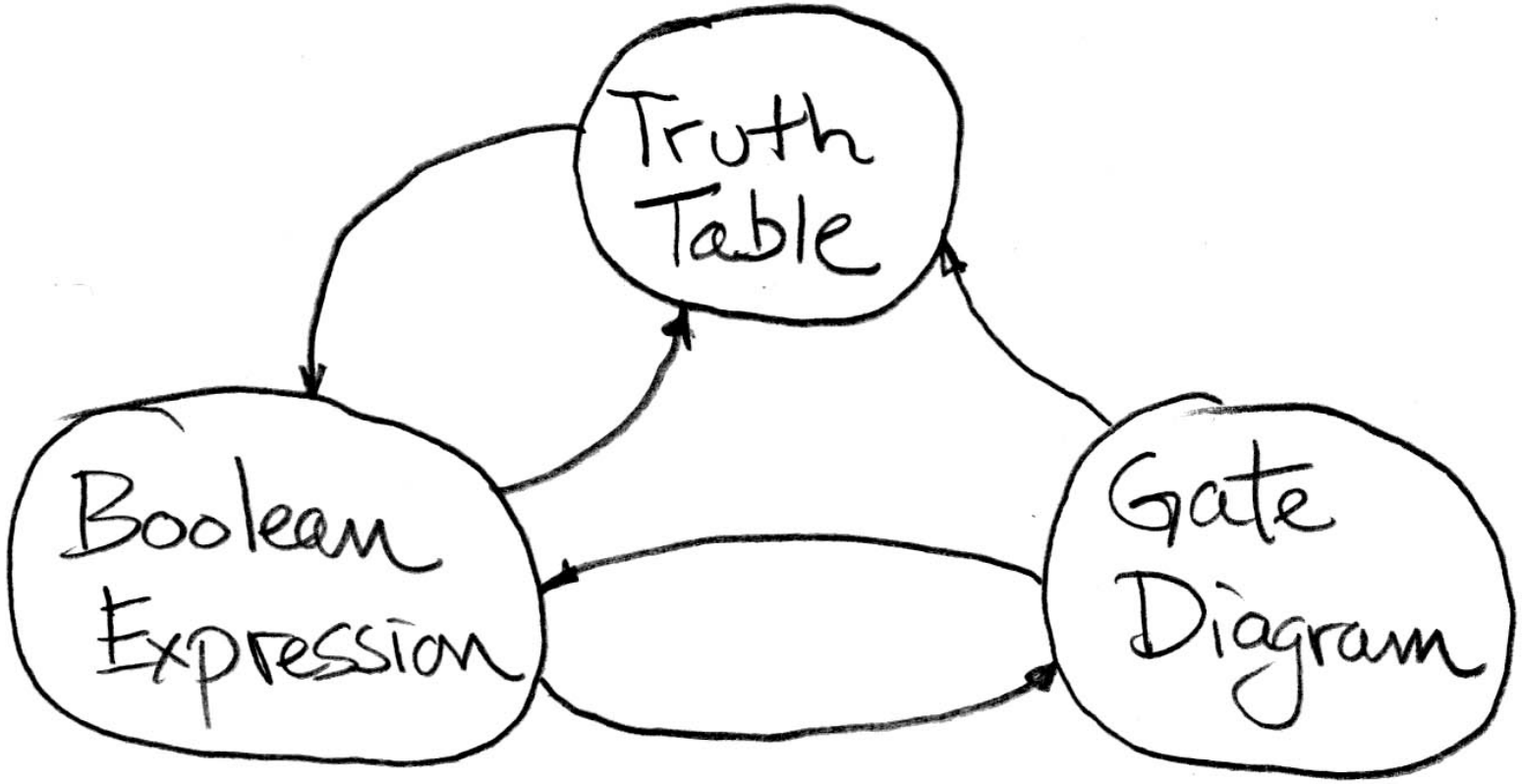
Canonical forms (2/2)

$$
\begin{aligned}
y &= \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c} \\
&= \bar{a}\bar{b}(\bar{c} + c) + a\bar{c}(\bar{b} + b) && \textit{distribution} \\
&= \bar{a}\bar{b}(1) + a\bar{c}(1) && \textit{complementarity} \\
&= \bar{a}\bar{b} + a\bar{c} && \textit{identity}
\end{aligned}
$$

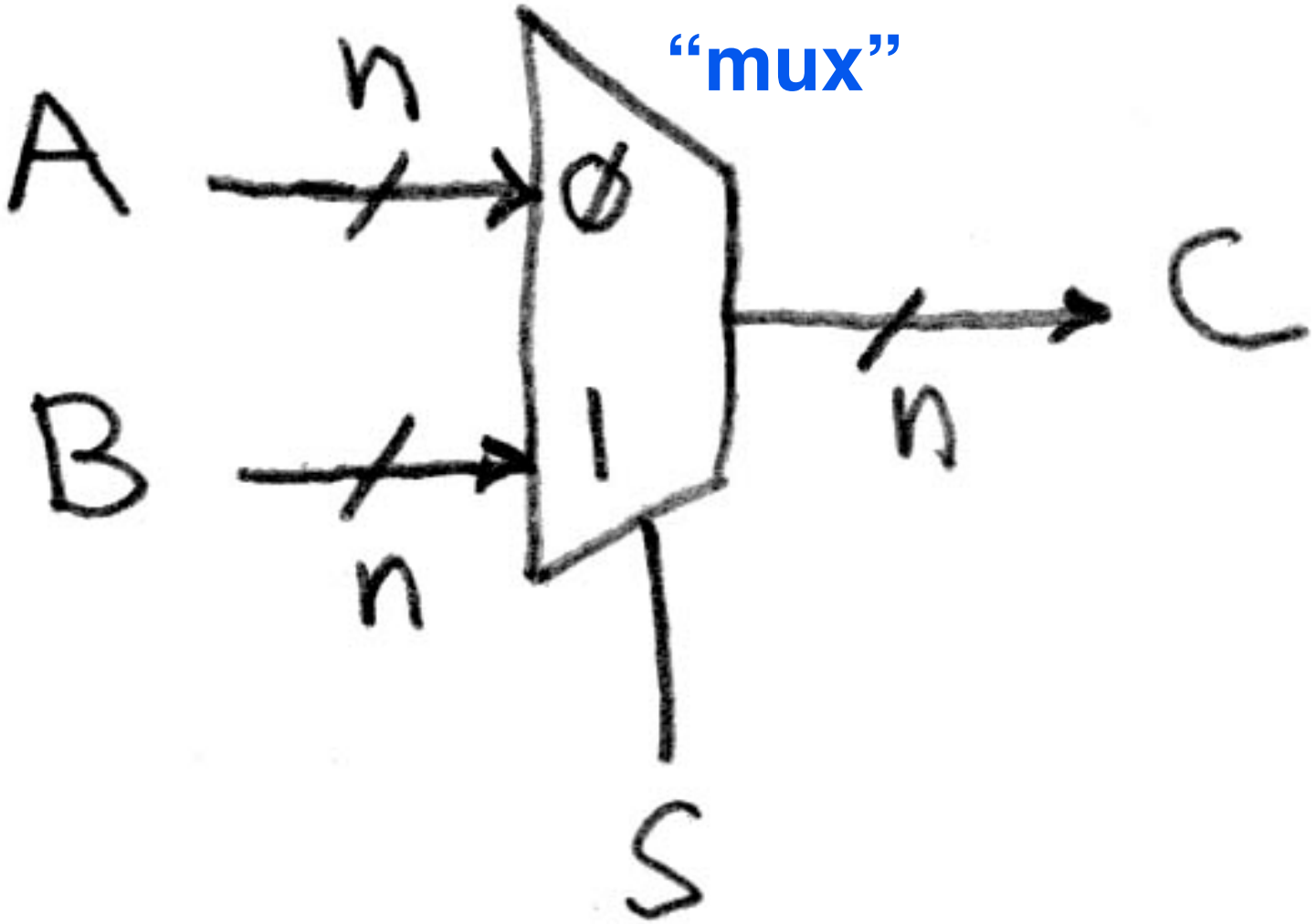


Review

- **Use this table and techniques we learned to transform from 1 to another**

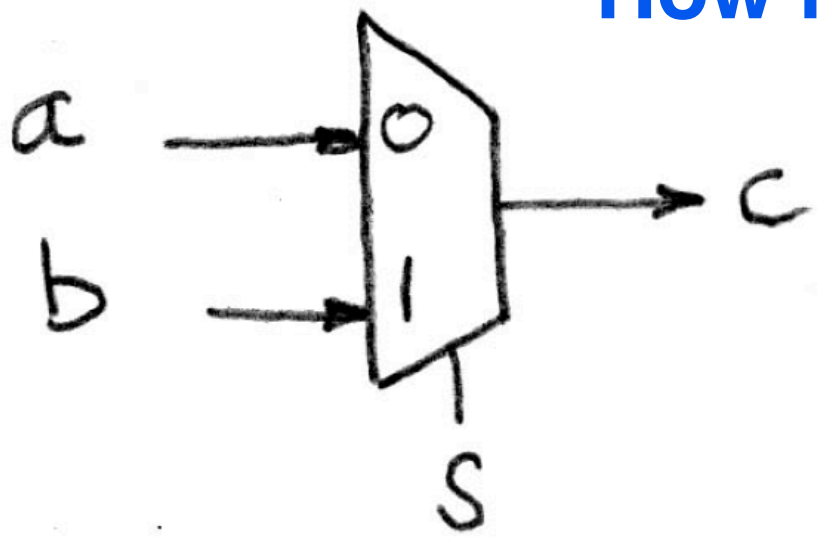


Data Multiplexor (here 2-to-1, n-bit-wide)



N instances of 1-bit-wide mux

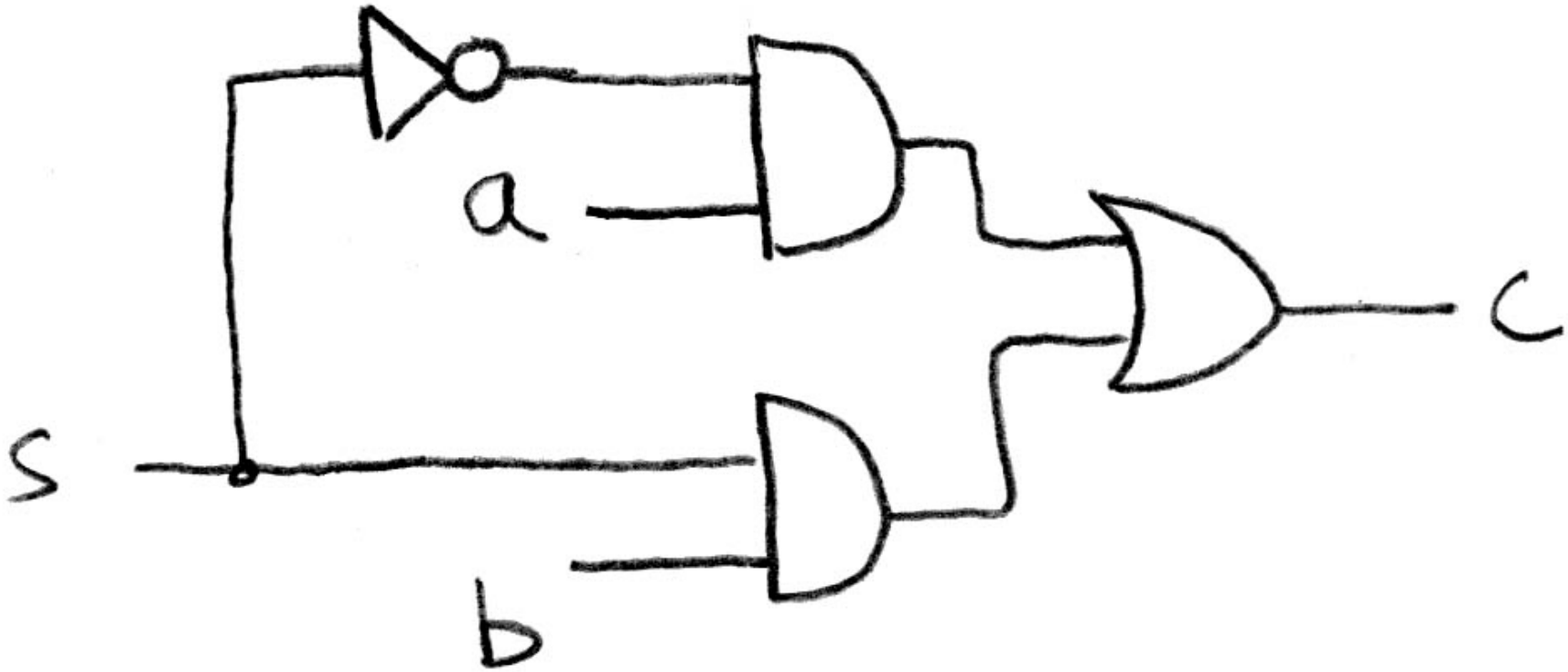
How many rows in TT?



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

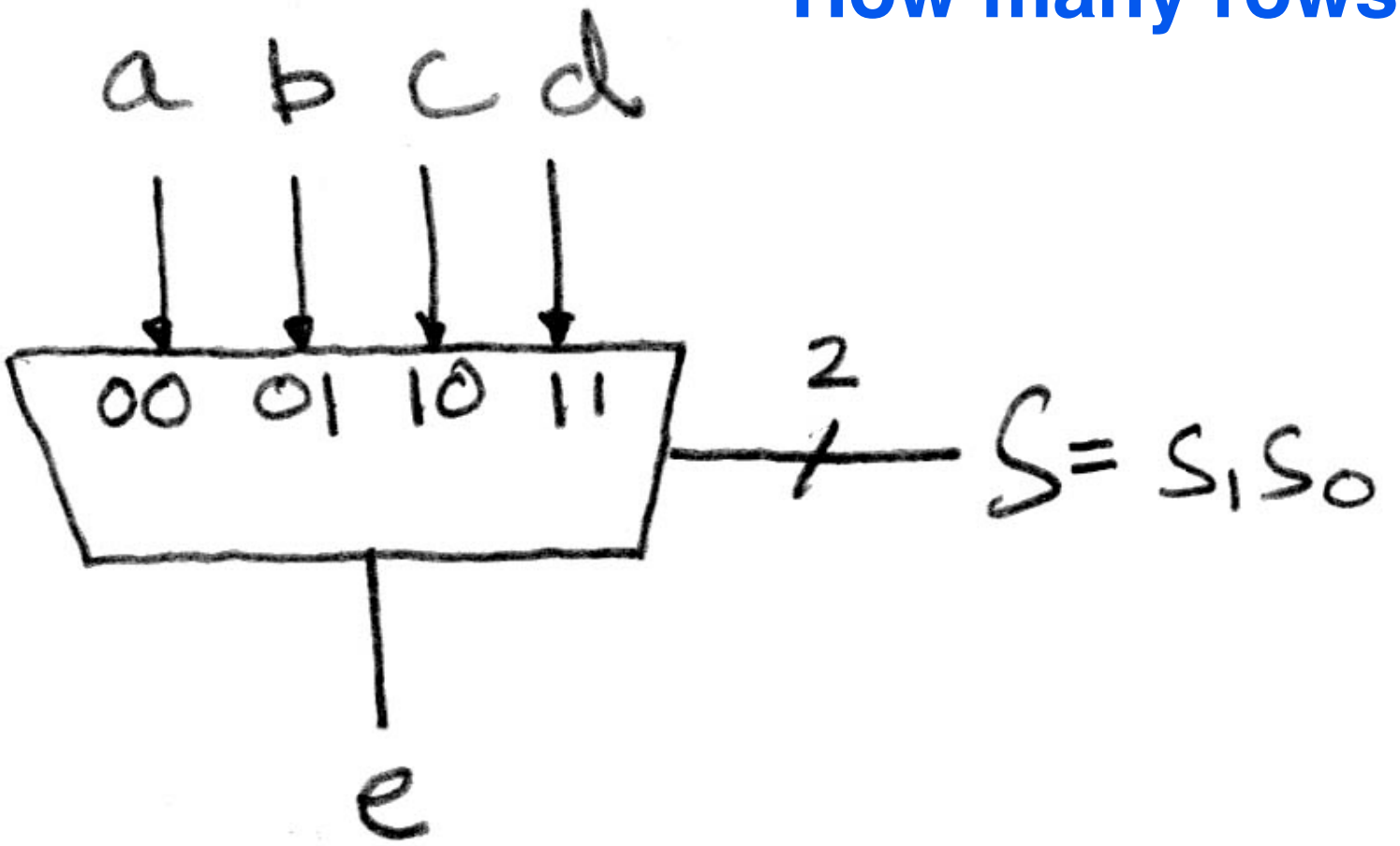
How do we build a 1-bit-wide mux?

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



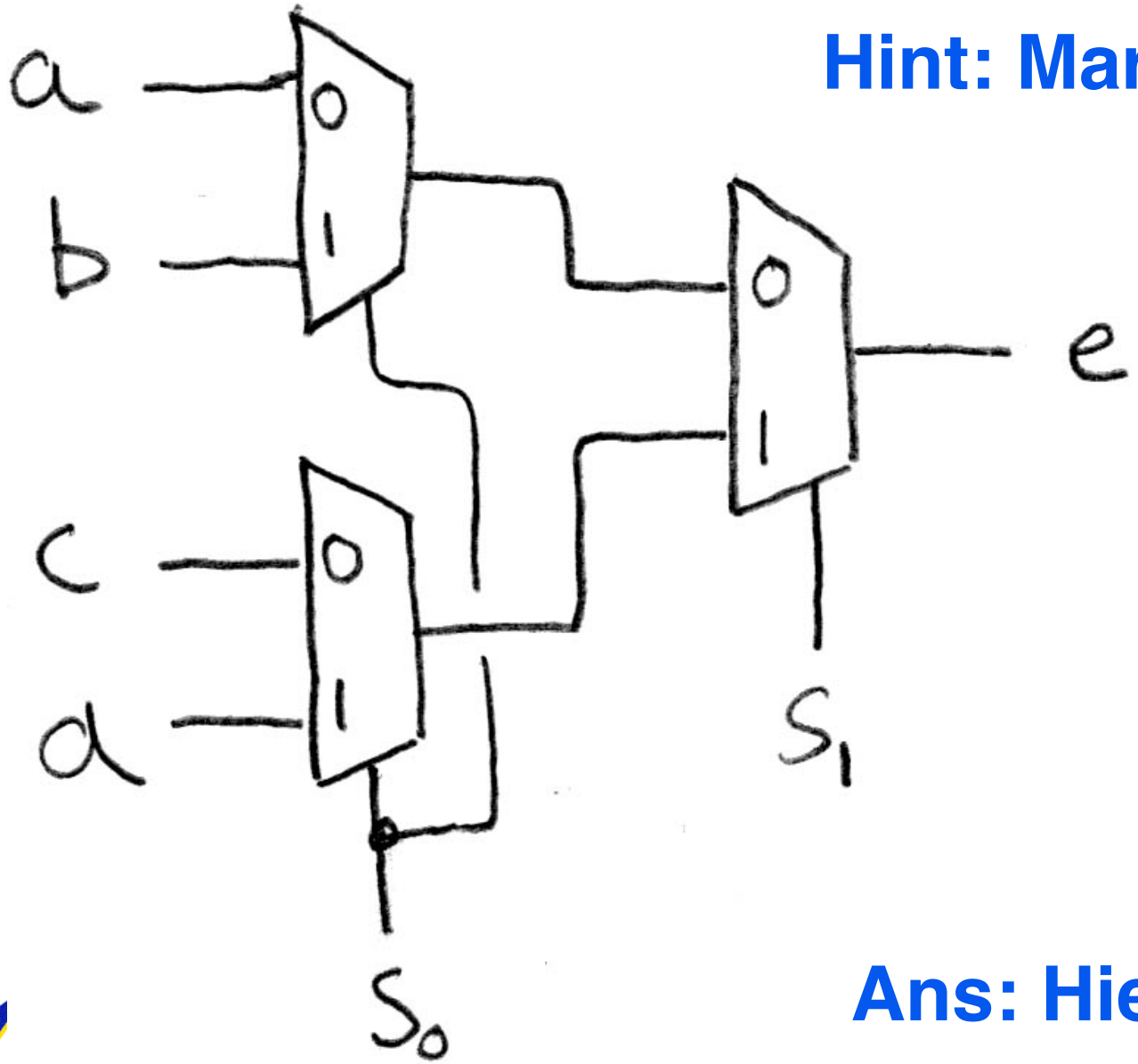
4-to-1 Multiplexor?

How many rows in TT?



$$e = \overline{s_1}\,\overline{s_0}a + \overline{s_1}s_0b + s_1\overline{s_0}c + s_1s_0d$$

Is there any other way to do it?



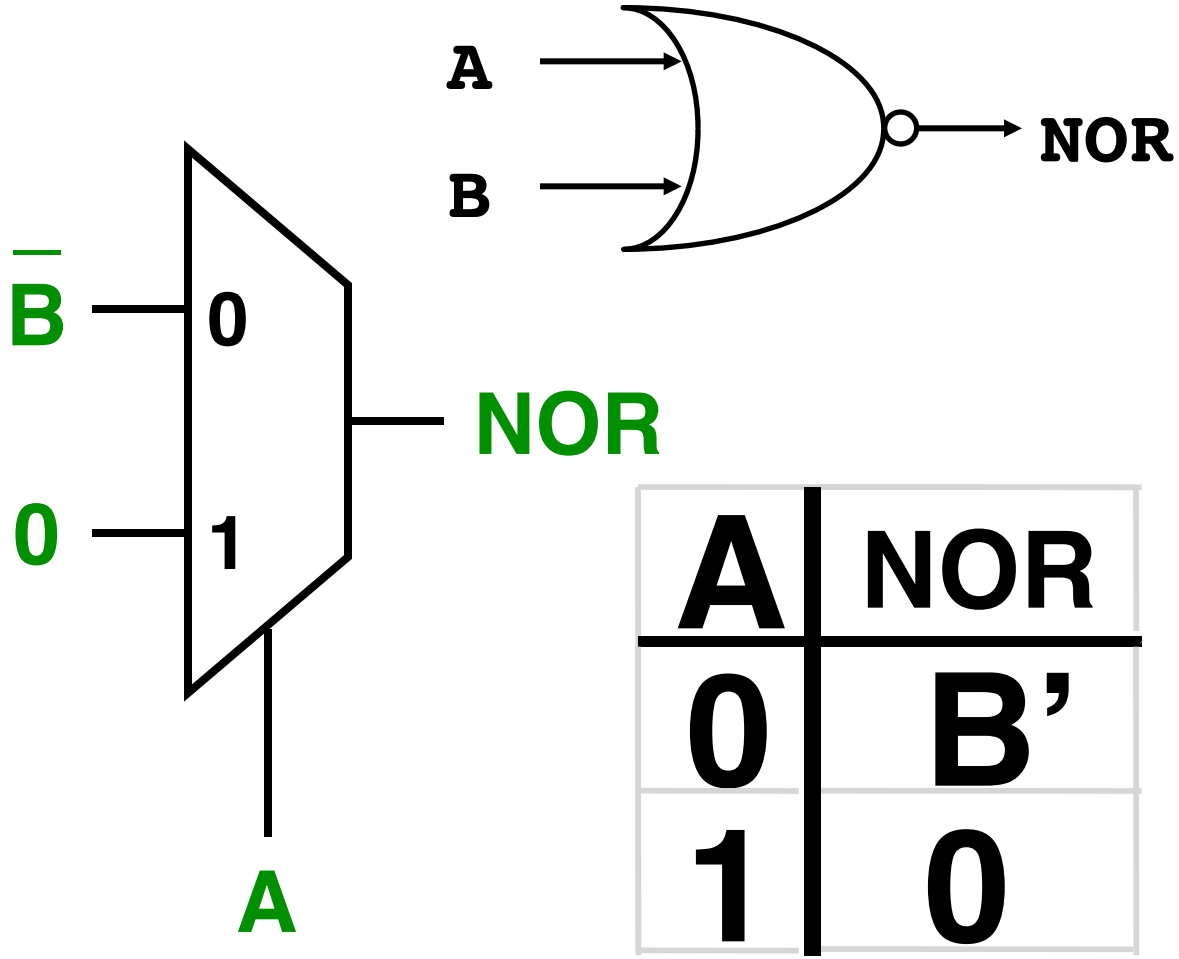
Hint: March Madness

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

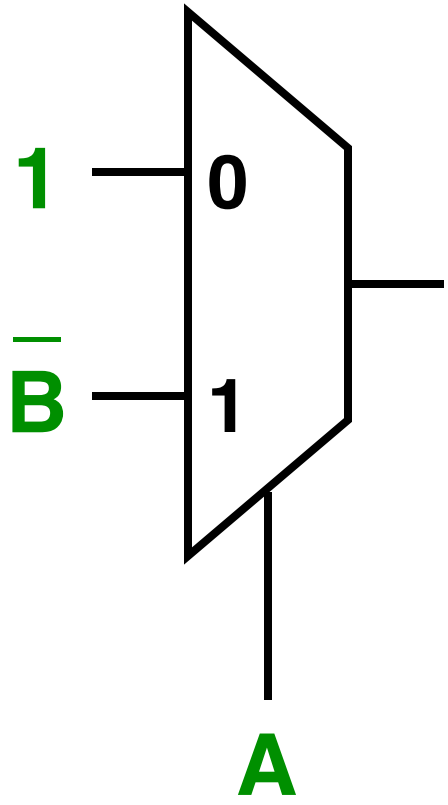


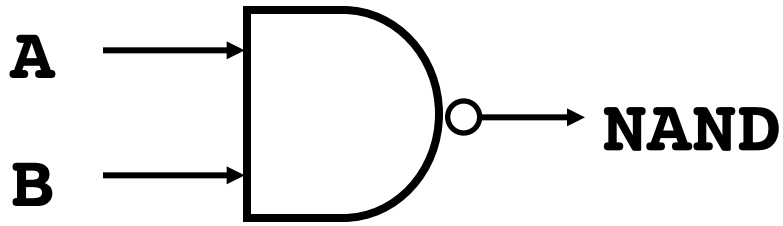
A	NOR
0	B’
1	0

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

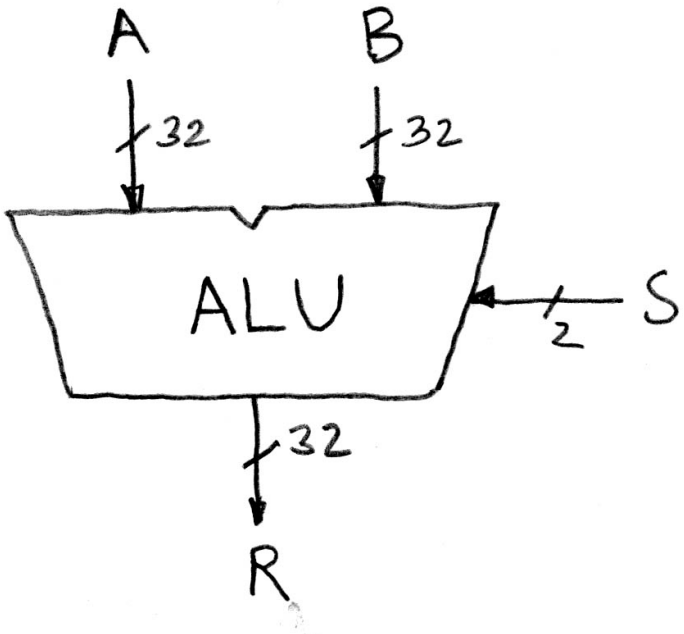




A	NAND
0	1
1	B’

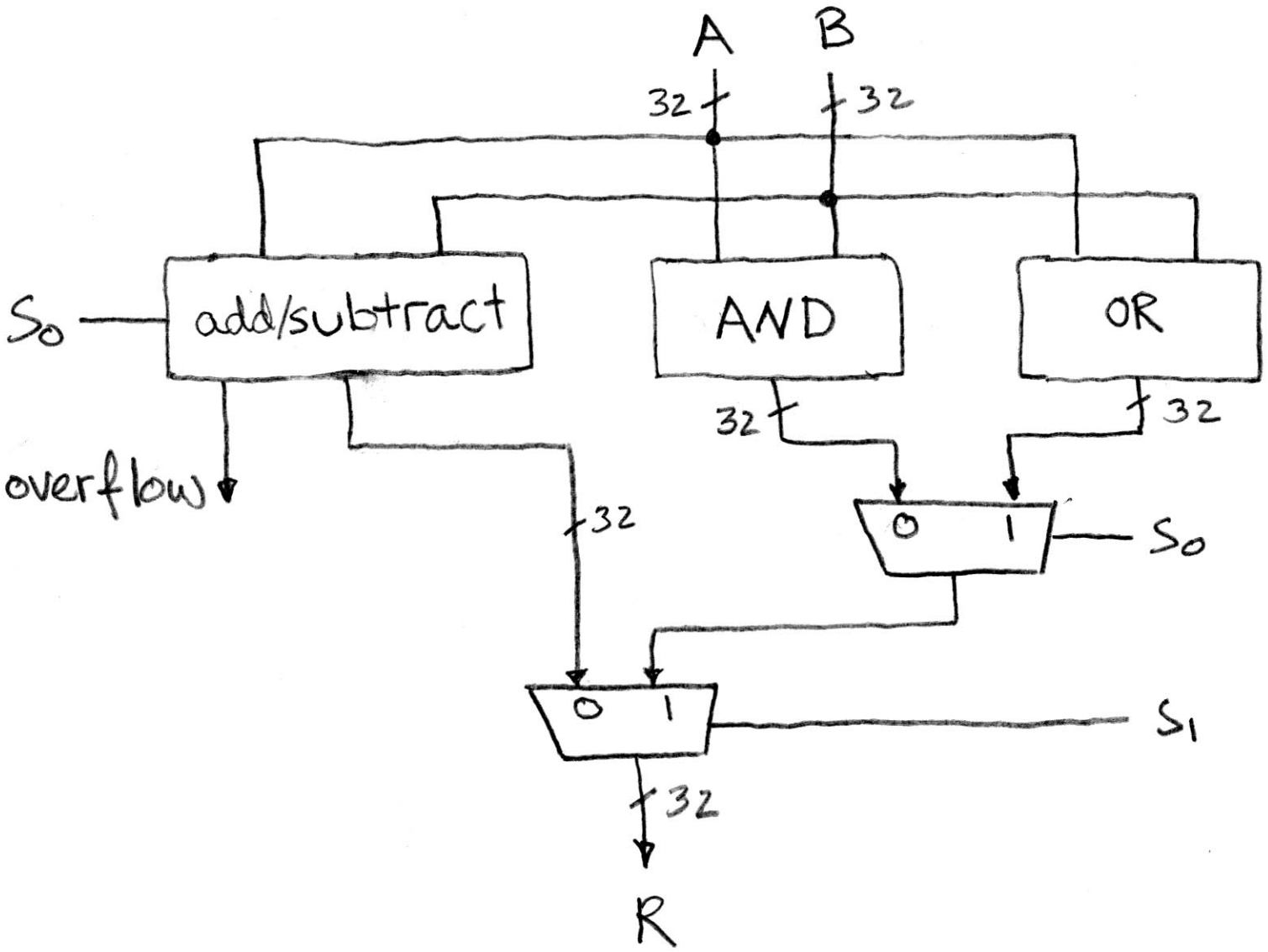
Arithmetic and Logic Unit

- **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



Conclusion

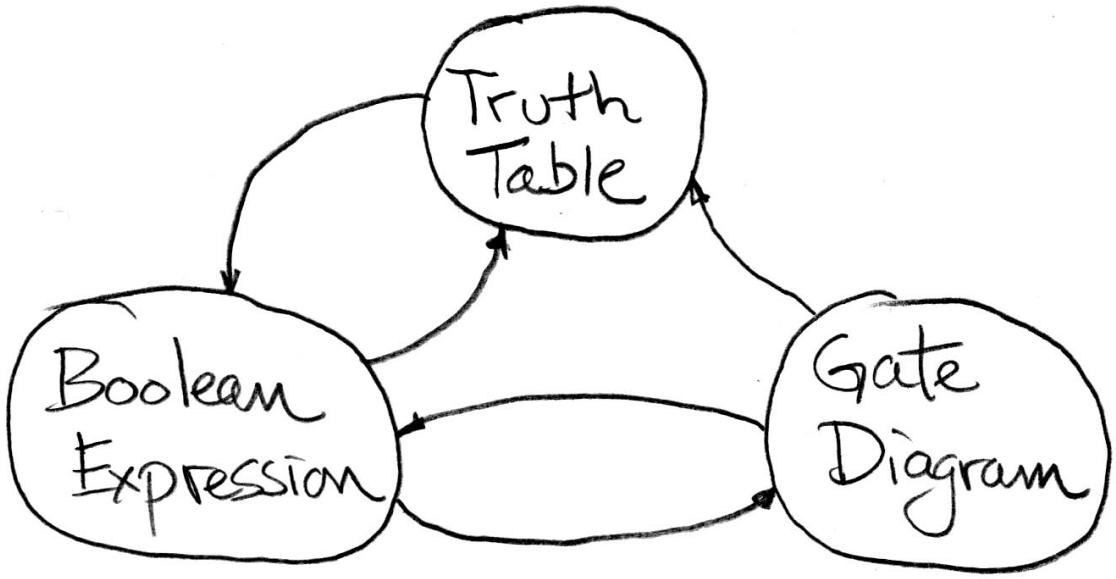
- ISA is very important abstraction layer
  - Contract between HW and SW
- Clocks control pulse of our circuits
- Voltages are analog, quantized to 0/1
- Circuit delays are fact of life
- Two types of circuits:
  - Stateless Combinational Logic (&,|,~)
  - State circuits (e.g., registers)

Conclusion

- State elements are used to:
  - Build memories
  - Control the flow of information between other state elements and combinational logic
- D-flip-flops used to build registers
- Clocks tell us when D-flip-flops change
  - Setup and Hold times important
- Finite State Machines extremely useful

Conclusion

- 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



Conclusion

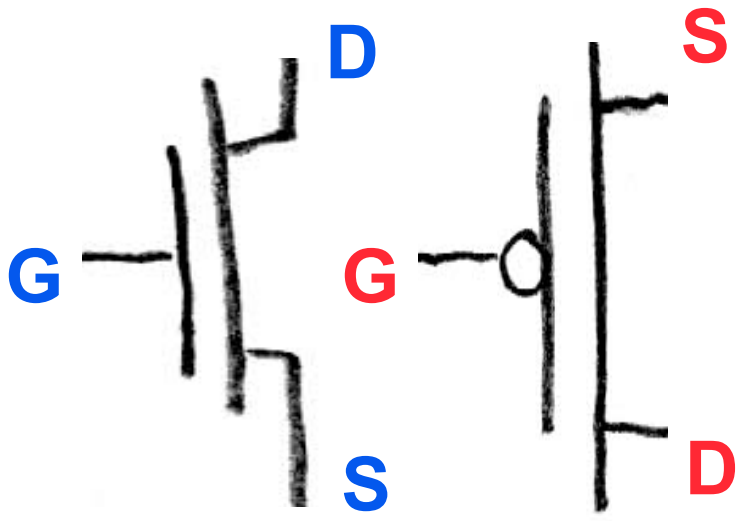
- Use muxes to select among input
  - S input bits selects 2^S inputs
  - Each input can be n-bits wide, indep of S
- Can implement muxes hierarchically
- ALU can be implemented using a mux
  - Coupled with basic block elements

Bonus slides

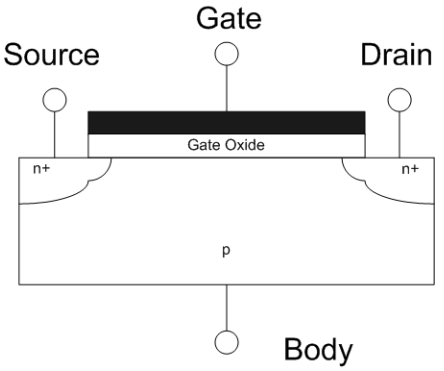
- These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Transistors 101

- **MOSFET**
  - **Metal-Oxide-Semiconductor Field-Effect Transistor**
  - **Come in two types:**
    - **n-type NMOSFET**
    - **p-type PMOSFET**



- **For n-type (p-type opposite)**
  - **If voltage not enough between G & S, transistor turns "off" (cut-off) and Drain-Source NOT connected**
  - **If the G & S voltage is high enough, transistor turns "on" (saturation) and Drain-Source ARE connected**

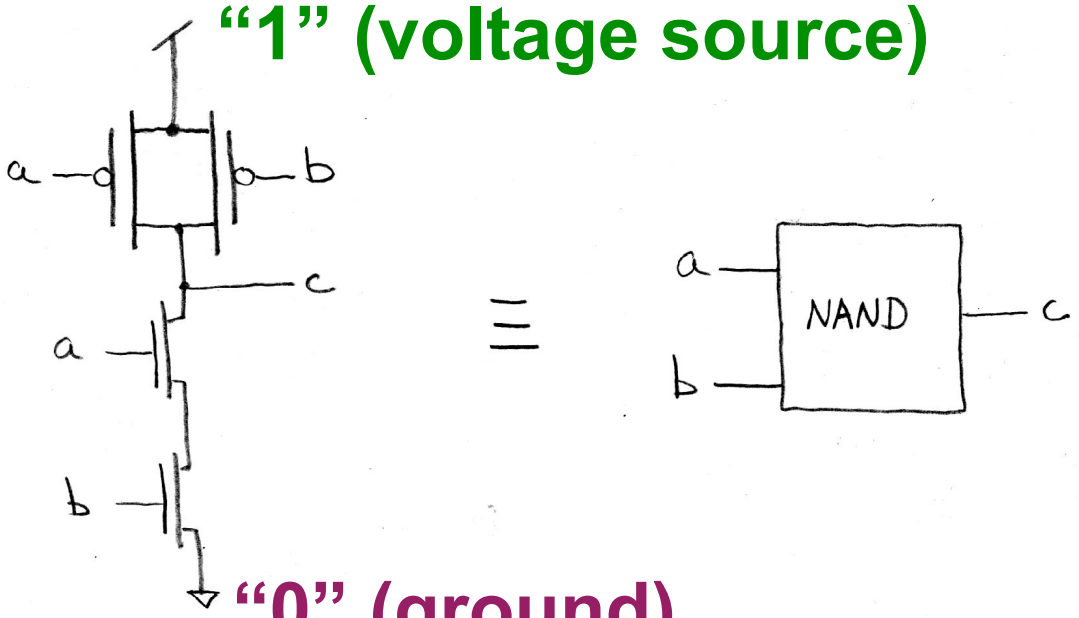


Side view

www.wikipedia.org/wiki/Mosfet

Transistor Circuit Rep. vs. Block diagram

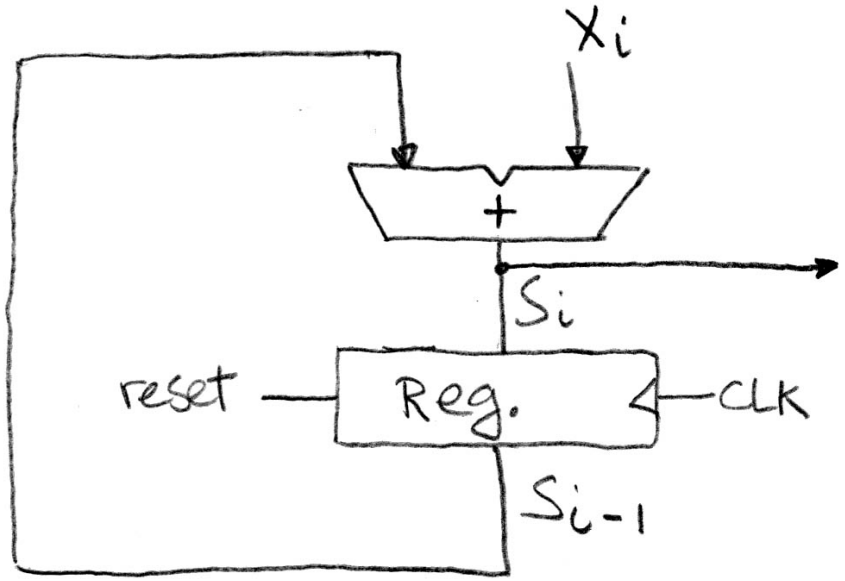
- Chips is composed of nothing but transistors and wires.
- Small groups of transistors form useful building blocks.



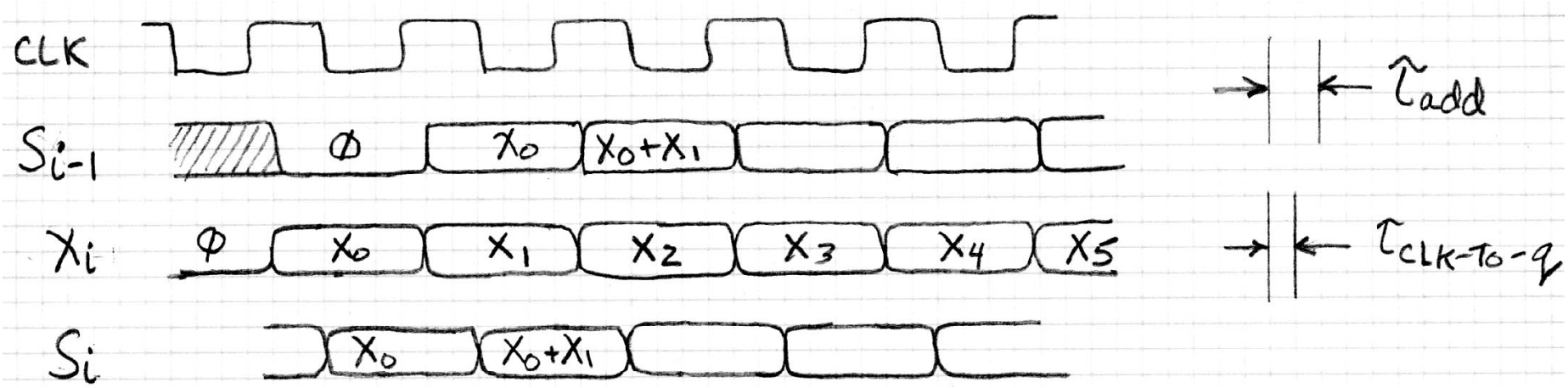
a	b	c
0	0	1
0	1	1
1	0	1
1	1	0

- Block are organized in a hierarchy to build higher-level blocks: ex: adders.

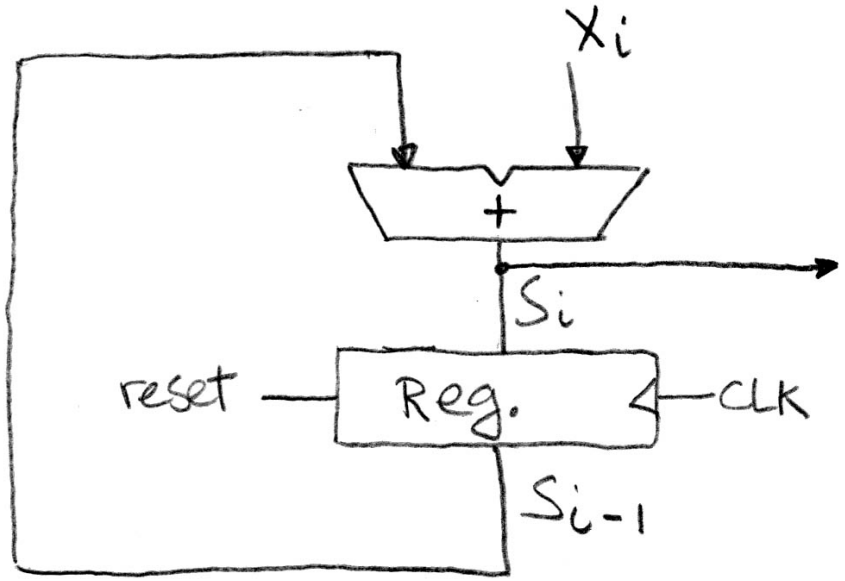
Accumulator Revisited (proper timing 1/2)



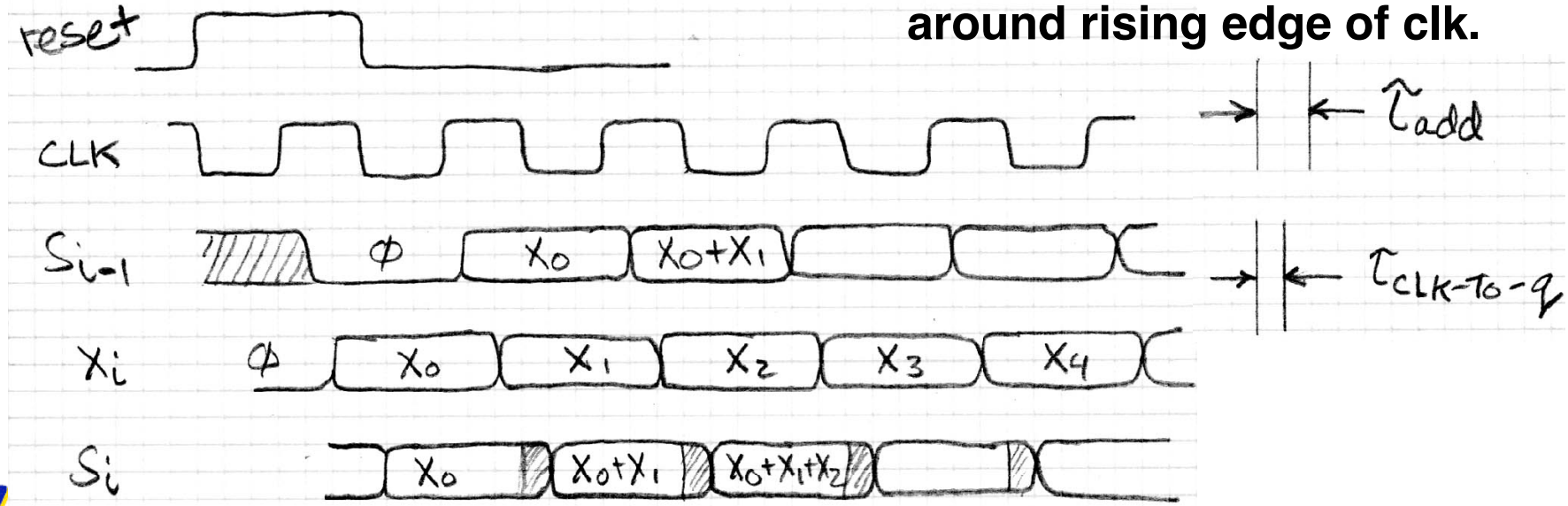
- **Reset input to register is used to force it to all zeros (takes priority over D input).**
- **S_{i-1} holds the result of the i^{th}-1 iteration.**
- **Analyze circuit timing starting at the output of the register.**



Accumulator Revisited (proper timing 2/2)

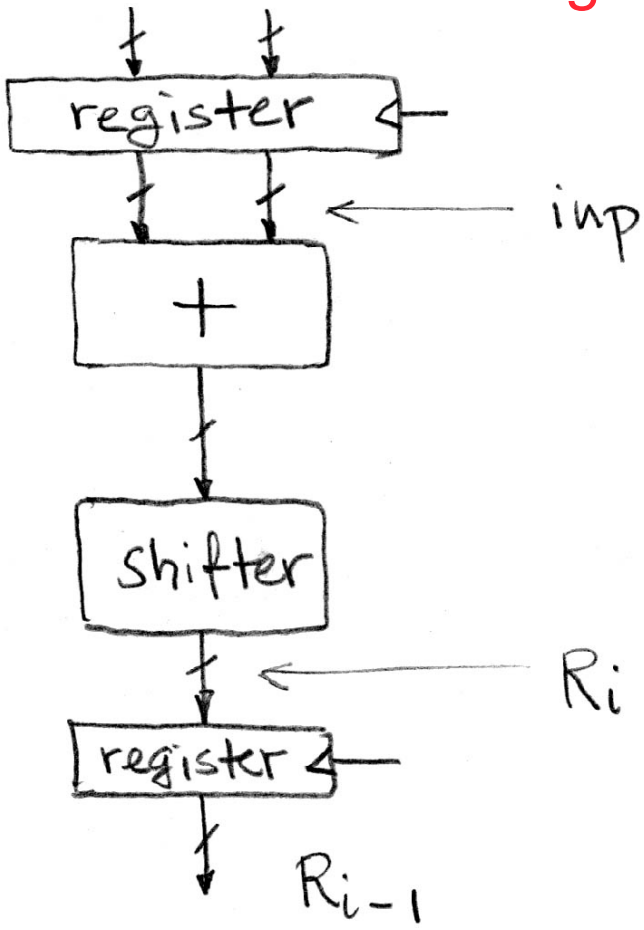


- reset signal shown.
- Also, in practice X might not arrive to the adder at the same time as S_{i-1}
- S_i temporarily is wrong, but register always captures correct value.
- In good circuits, instability never happens around rising edge of clk.

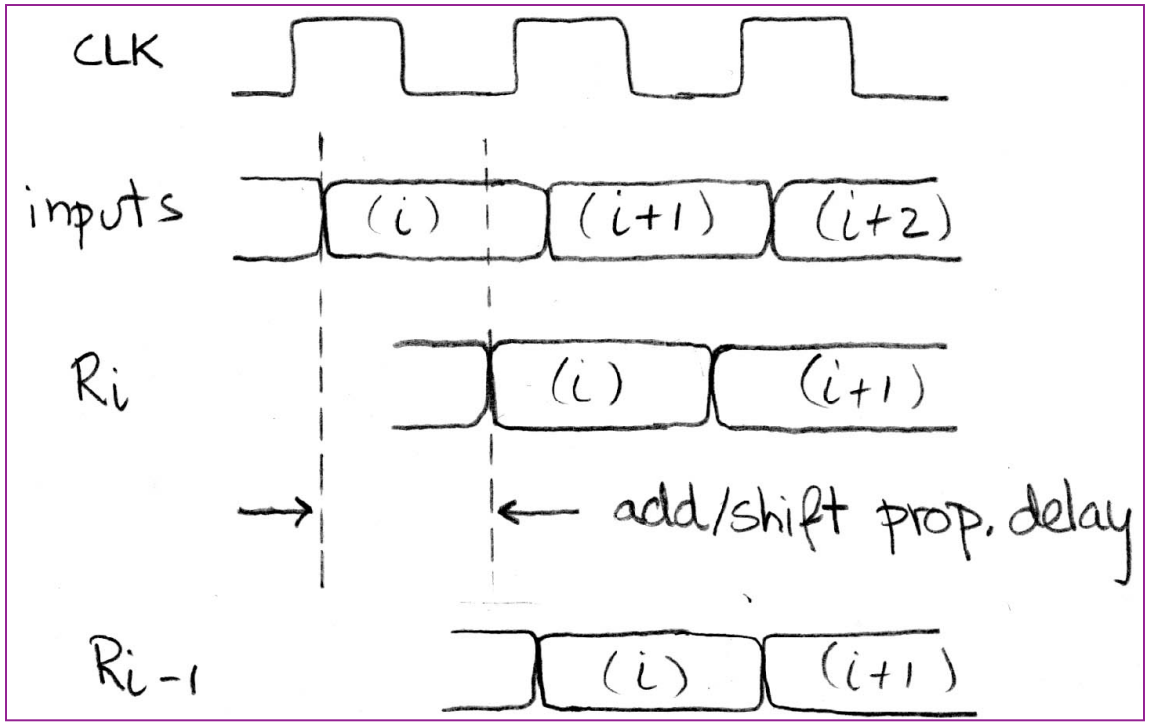


Pipelining to improve performance (1/2)

Extra Register are often added to help speed up the clock rate.

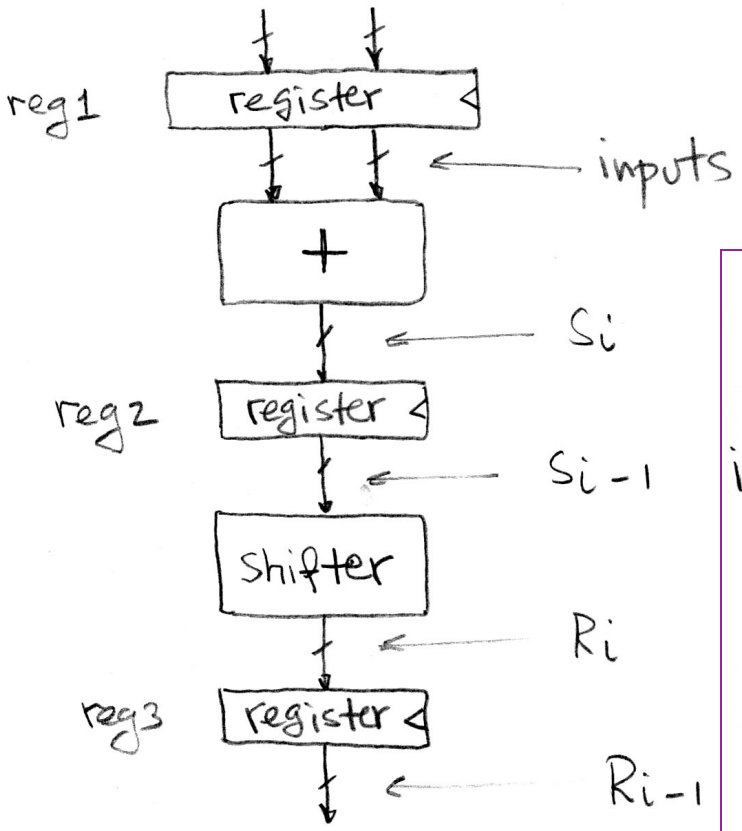


Timing…



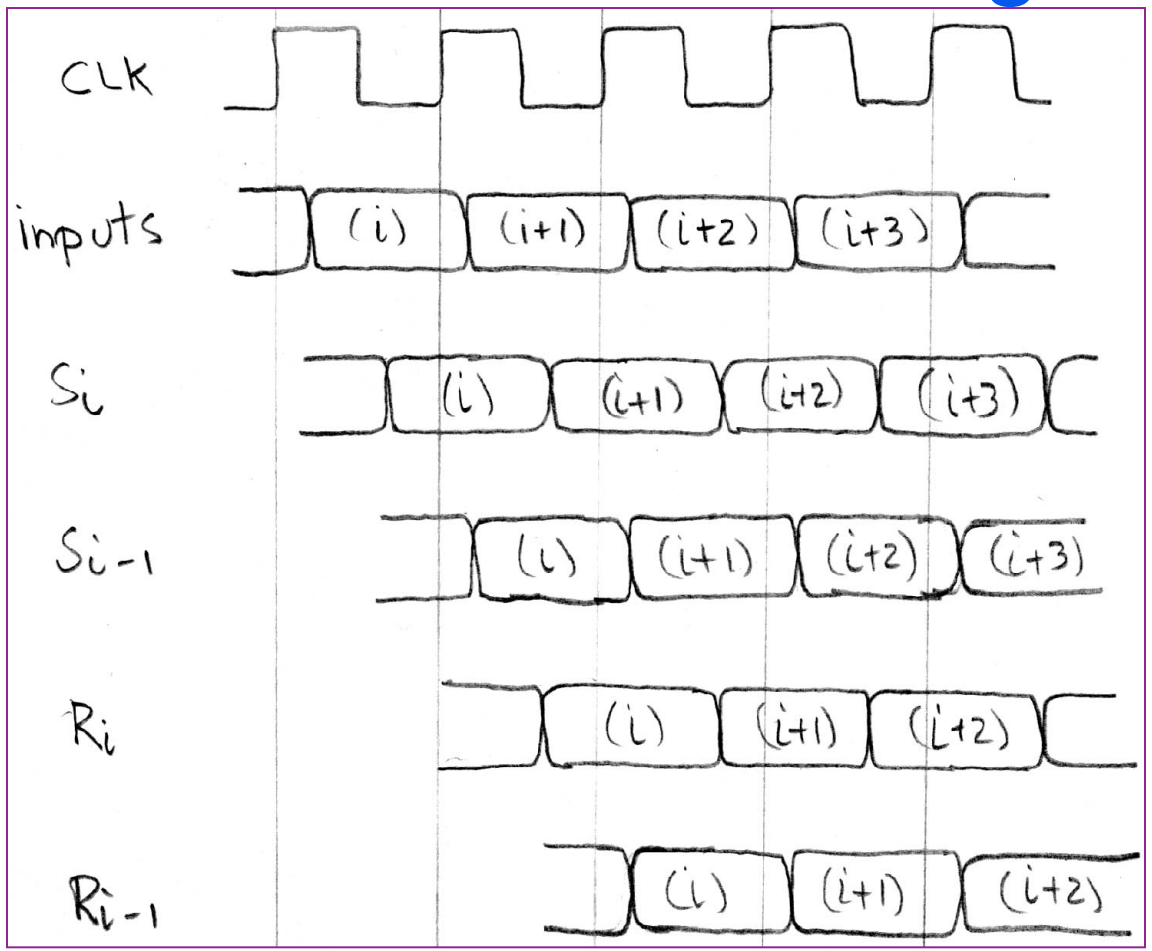
Note: delay of 1 clock cycle from input to output.
Clock period limited by propagation delay of adder/shifter.

Pipelining to improve performance (2/2)

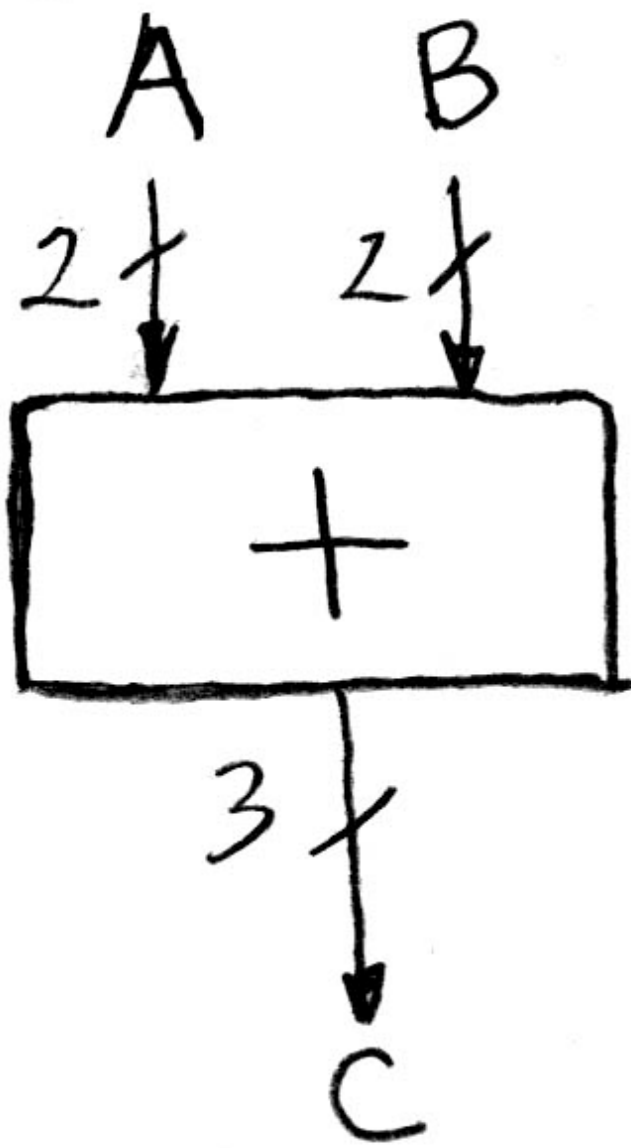


- Insertion of register allows higher clock frequency.
- More outputs per second.

Timing…



TT Example #2: 2-bit adder



A a_1a_0	B b_1b_0	C $c_2c_1c_0$
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

How Many Rows?

TT Example #3: 32-bit unsigned adder

A	B	C
000 ... 0	000 ... 0	000 ... 00
000 ... 0	000 ... 1	000 ... 01
.	.	.
.	.	.
.	.	.
111 ... 1	111 ... 1	111 ... 10

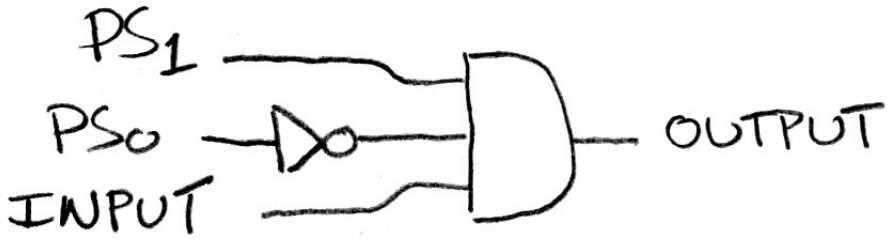
How Many Rows?

TT Example #3: 3-input majority circuit

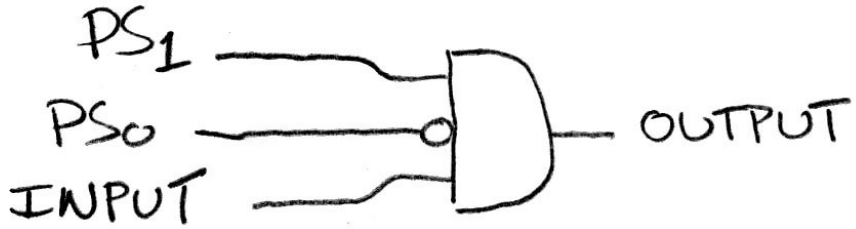
a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Truth Table ⇒ Gates (e.g., FSM circ.)

PS	Input	NS	Output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

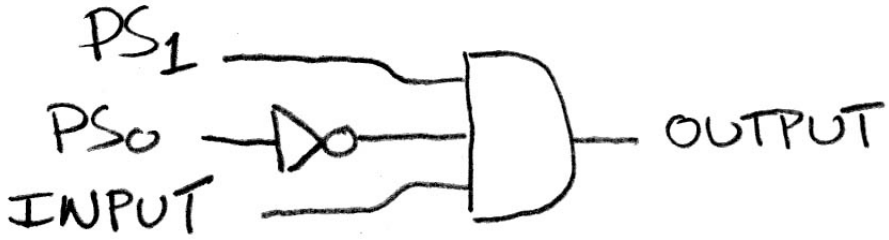


or equivalently…

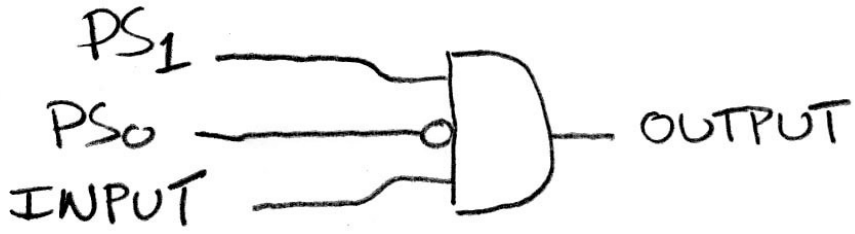


Boolean Algebr (e.g., for FSM)

PS	Input	NS	Output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1



or equivalently…



$$y = PS_1 \cdot \overline{PS_0} \cdot INPUT$$

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)…

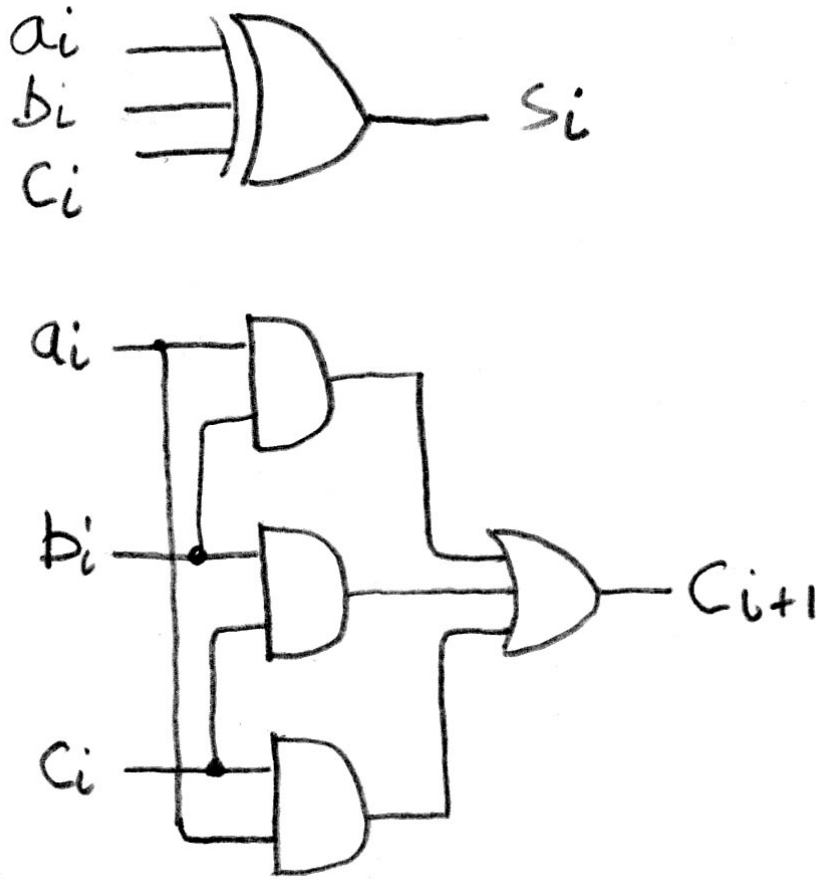
$$
\begin{array}{cccccc}
 & a_3 & a_2 & \boxed{a_1} & a_0 \\
+ & b_3 & b_2 & \boxed{b_1} & b_0 \\
\hline
 & s_3 & s_2 & \boxed{s_1} & s_0
\end{array}
$$

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

$$s_i =$$

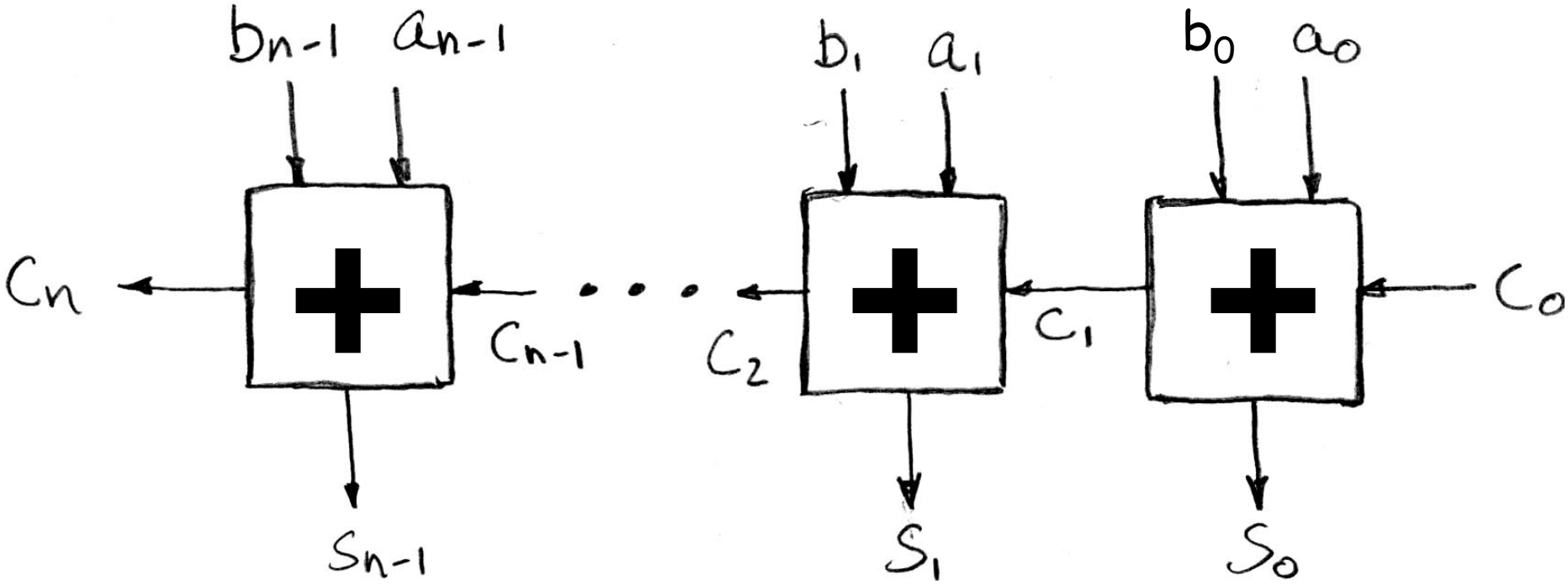
$$c_{i+1} =$$

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

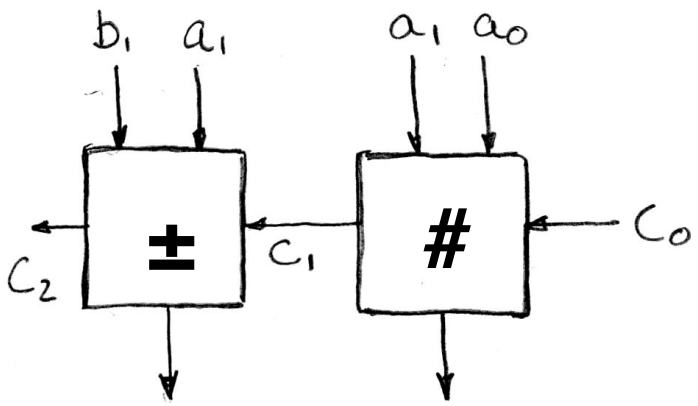


What about overflow?

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 = [illegible] C_{out}
  - No C_{out} or C_{in} ⇒ NO overflow!
  - C_{in}, and C_{out} ⇒ NO overflow!

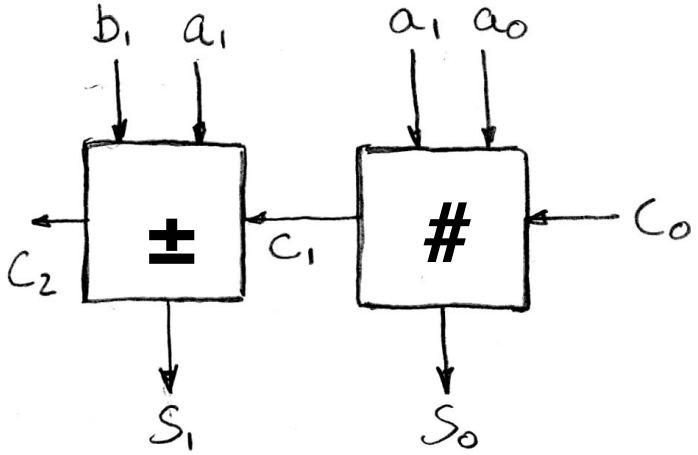
What op?

- 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  ← -2 or -1
11 = -1  ← -2 only
00 =  0  NOTHING!
01 =  1  ← 1 only
```



- 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