

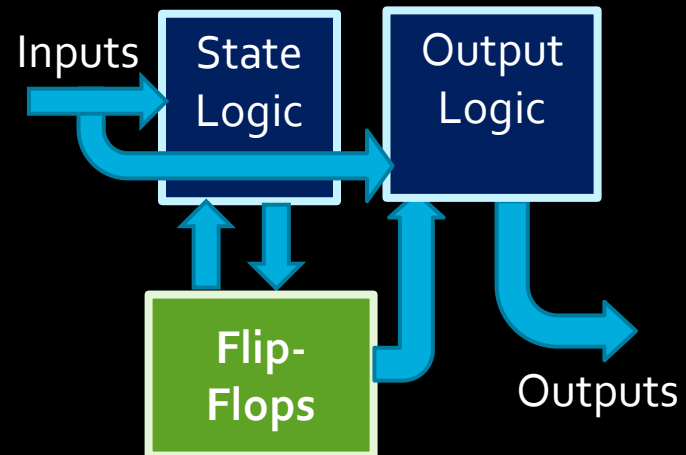


Week 5 Review



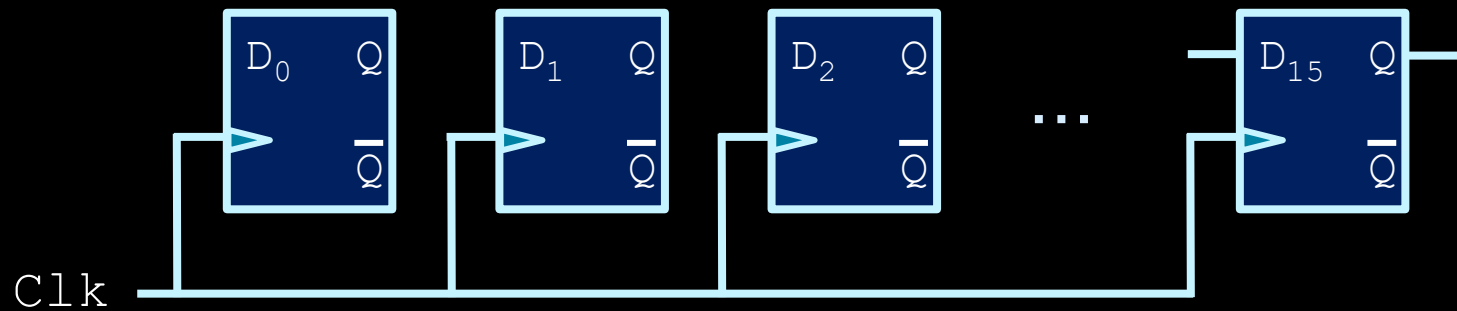
Week 5

- Registers
- Counters
- Basics of state machines
- How to design state machines



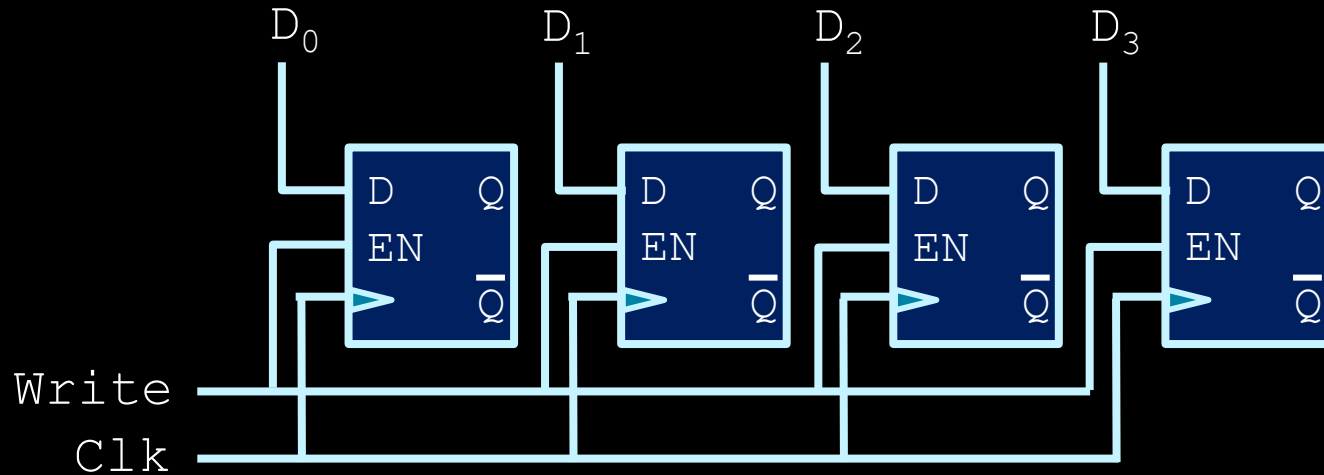
Registers

- An **n-bit register**: a bank of n flip-flops that share a common clock.
- Registers store a multi-bit value.



- All bits written at the same time.
- Key building block of sequential systems and CPUs.

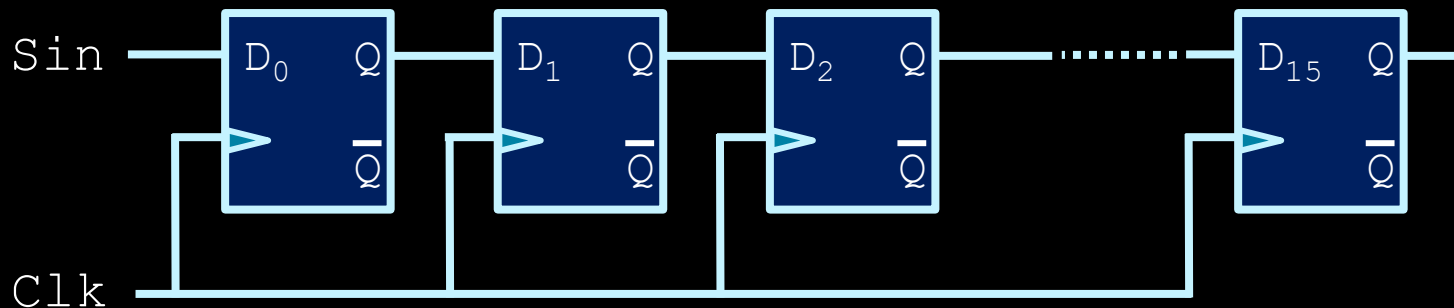
Load registers



- Write value in parallel
- Write-enable (write) signal controls if writing or not.

Shift Registers

- A series of D flip-flops where output of flip-flop i is connected to input of $i+1$



- Load bits **one bit at a time**.
- Or implement **parallel load**.
- Useful to save space or when we **want to shift**.

Counters

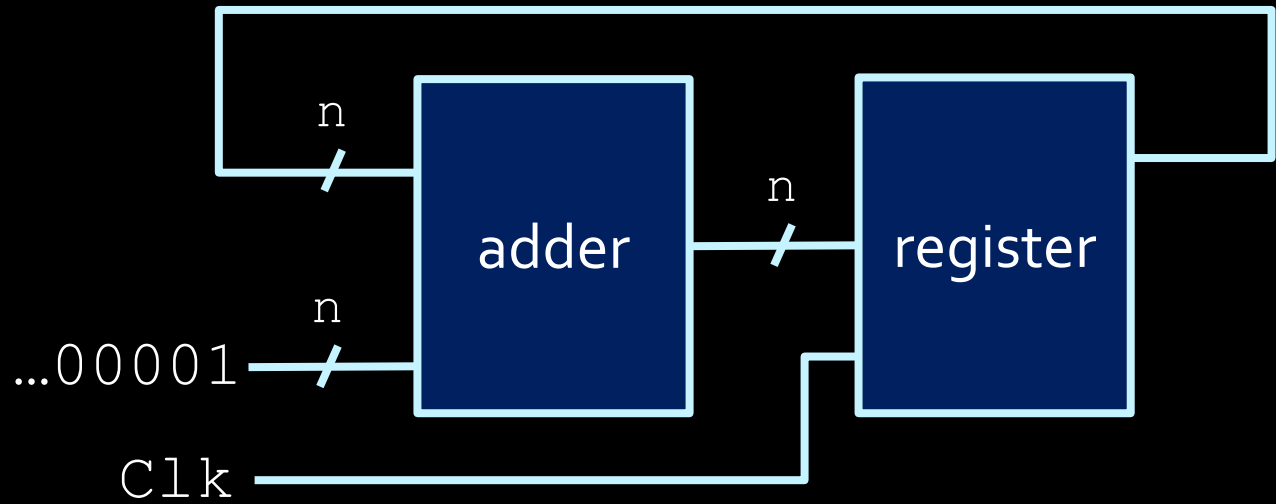
- Basic idea...

one bit...
two bit...
three bits...

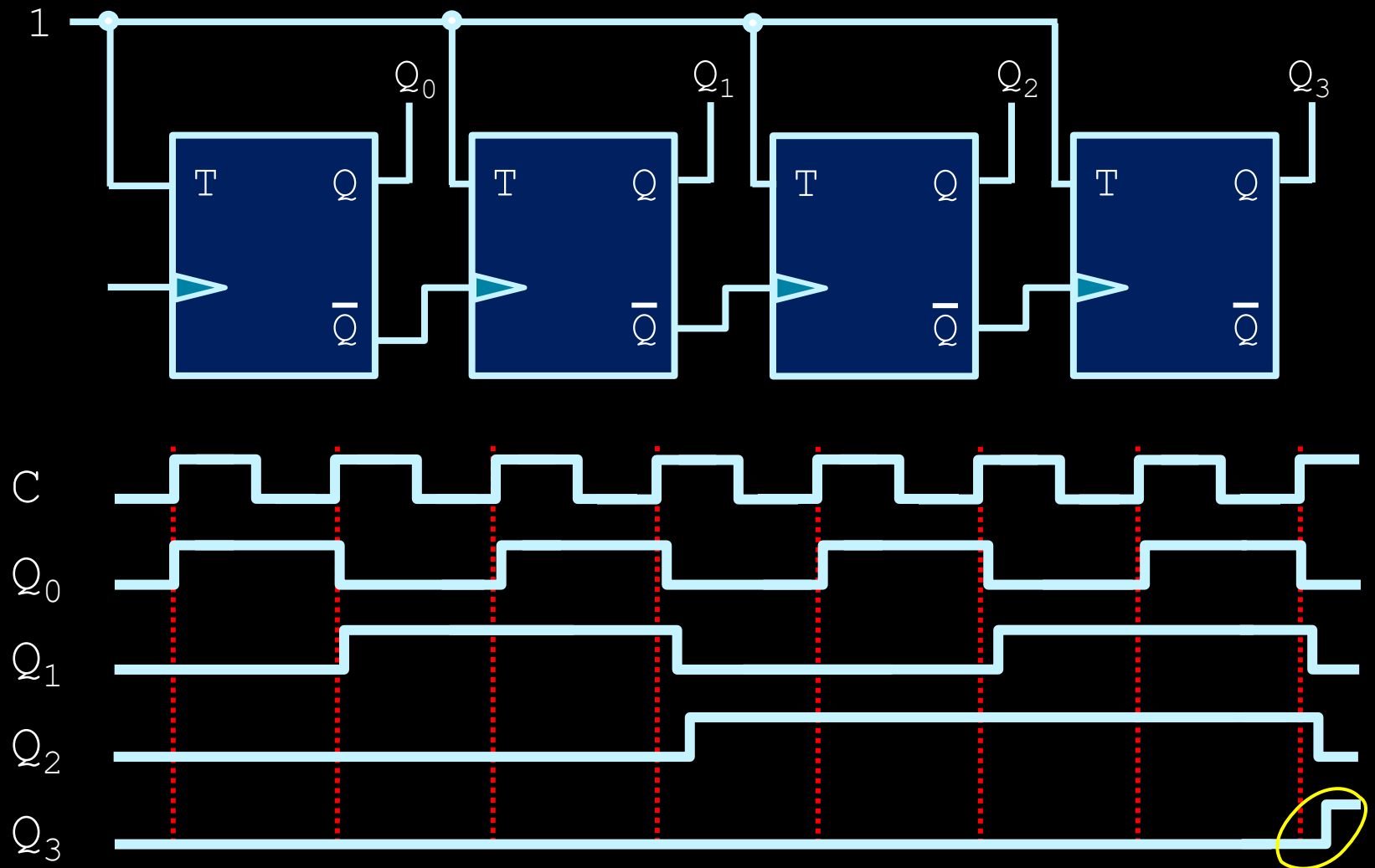


Counters

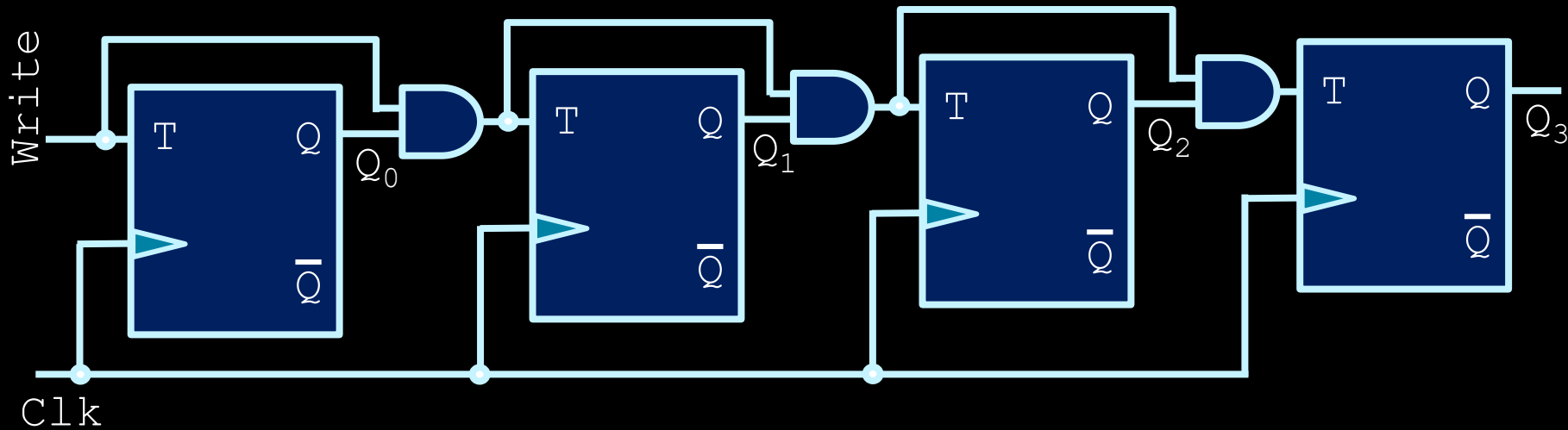
- Basic idea...



Ripple Counter (async!)

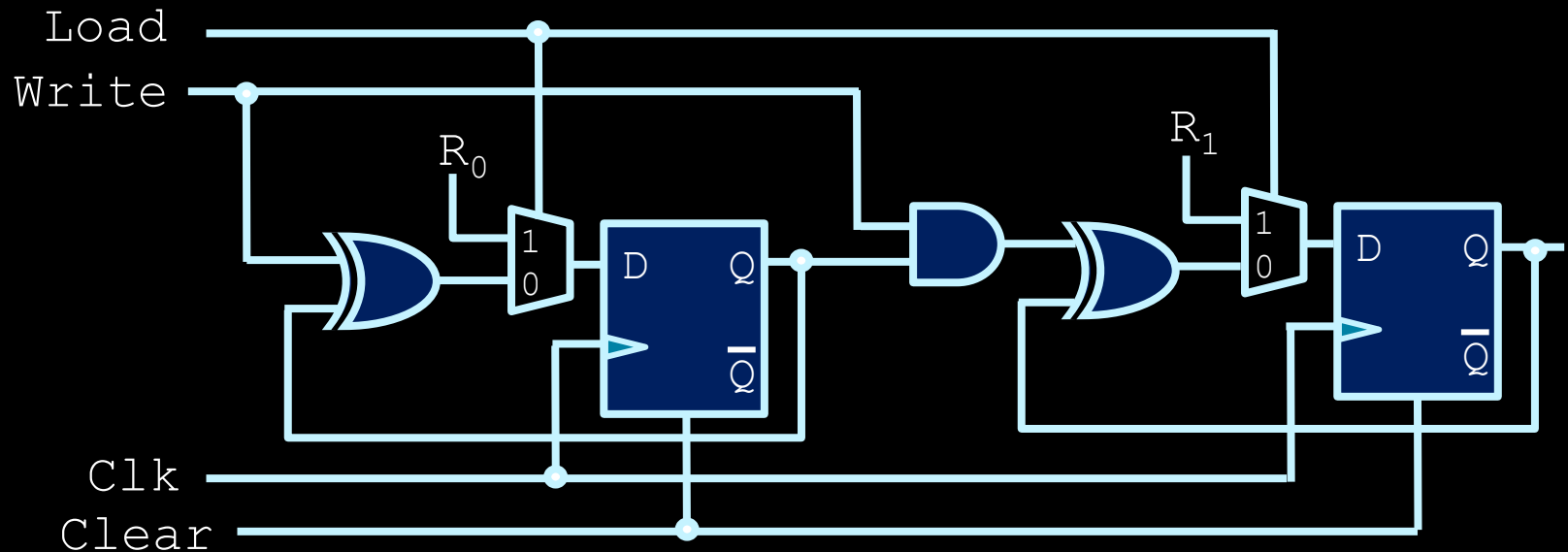


Synchronous Counter



- This is a **synchronous** counter, with a slight delay.

Counters with Load



- Counter with **parallel load, write, and clear** (reset) inputs.
- Useful for countdowns and more.

Question

We want to build a change machine

- We can add either \$0.05 or \$0.10 at a time
- We want to keep track of the current amount in the machine
- We can hold a maximum of \$0.50
- Draw the state diagram
 - For now, ignore the possibility of going over \$0.50

Question

- How many flip-flops would you need to implement the following finite state machine (FSM)?
 - 11 states
 - # flip-flops = $\lceil \log_2 (\# \text{ of states}) \rceil$
 - # flip-flops = 4



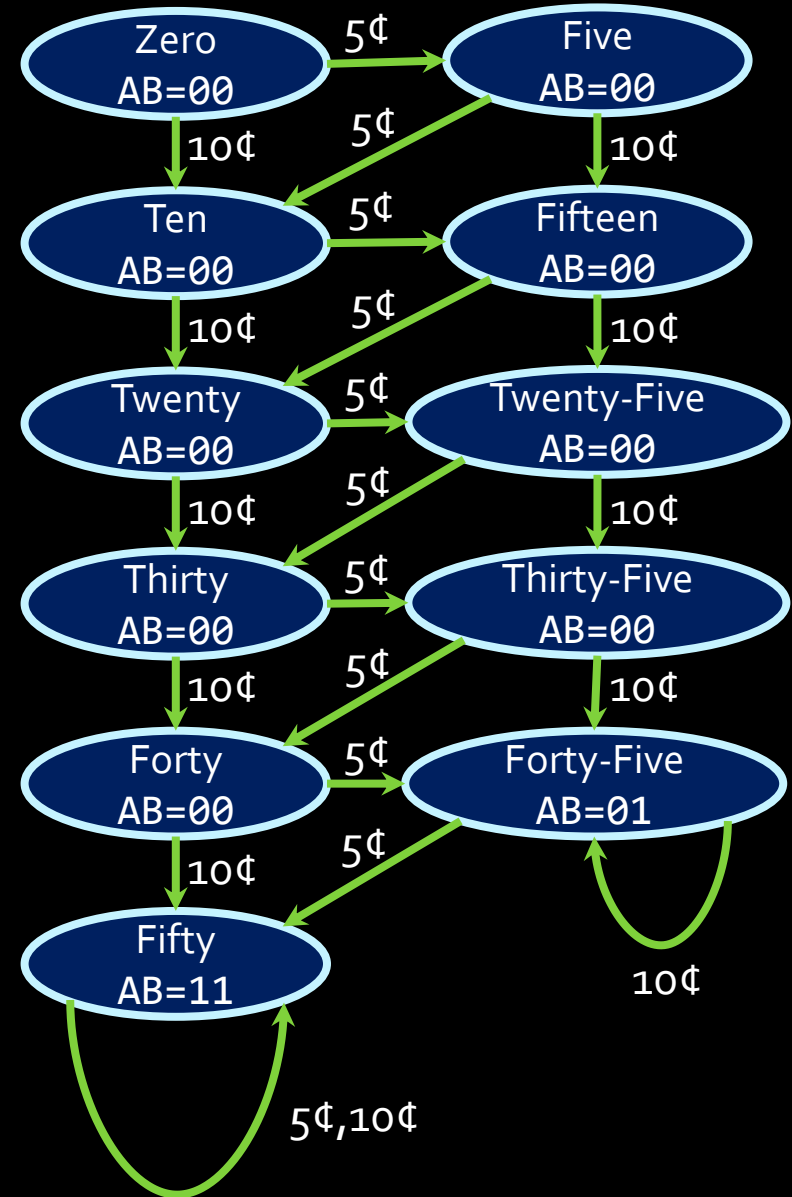
Question

- Dealing with too much money means when we don't accept any more coins.
- New outputs tell machine what to do:
 - $A = 1 \rightarrow$ reject 5c
 - $B = 1 \rightarrow$ reject 10c
- Complete the state machine for this addition



Question

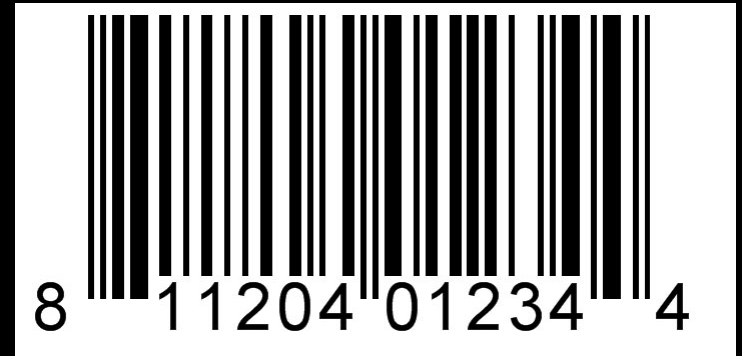
- Dealing with too much money means when we don't accept any more coins.
- New outputs tell machine what to do:
 - $A = 1 \rightarrow$ reject 5c
 - $B = 1 \rightarrow$ reject 10c
- Complete the state machine for this addition



Question: Barcode Reader



- When scanning UPC barcodes, the laser scanner looks for black and white bars that indicate the start of the code.
- If black is read as a 1 and white is read as a 0, the start of the code has a **1010 pattern**.
 - Create a state machine that detects this pattern



Question: Barcode Reader

- Build a machine that detects a **1010 pattern**.



Input 0 0 0 1 0 1 0 0 0 0

Output 0 0 0 1 0 0 0

Input 0 0 0 1 0 1 0 1 0 1 0 0 0 0

Input 1 0 1 1 0 1 0 1 0

Question: Barcode Reader

- Build a machine that detects a **1010 pattern**.
- We know how to build a pattern recognizer:
 - One state for each possible sequence of 4 bits.
 - In other words, one FF per bit:
 F_0 last seen bit, F_1 the bit before that, F_2 for bit seen before that, and F_3 .
- Can we do better than 4 FFs?

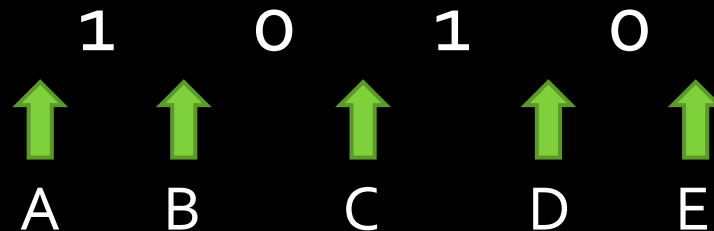


Question: Barcode Reader

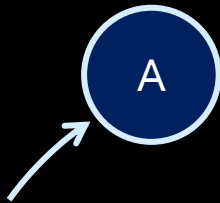
- The previous pattern recognizer:
state for each possible sequence of 4 bits.
 - ▣ It essentially memorized
- Do you have another idea?

Question: Barcode Reader

- The previous pattern recognizer:
state for each possible sequence of 4 bits.
 - It essentially memorized
- Do you have another idea?
 - How about a state that shows us where we are in the pattern?



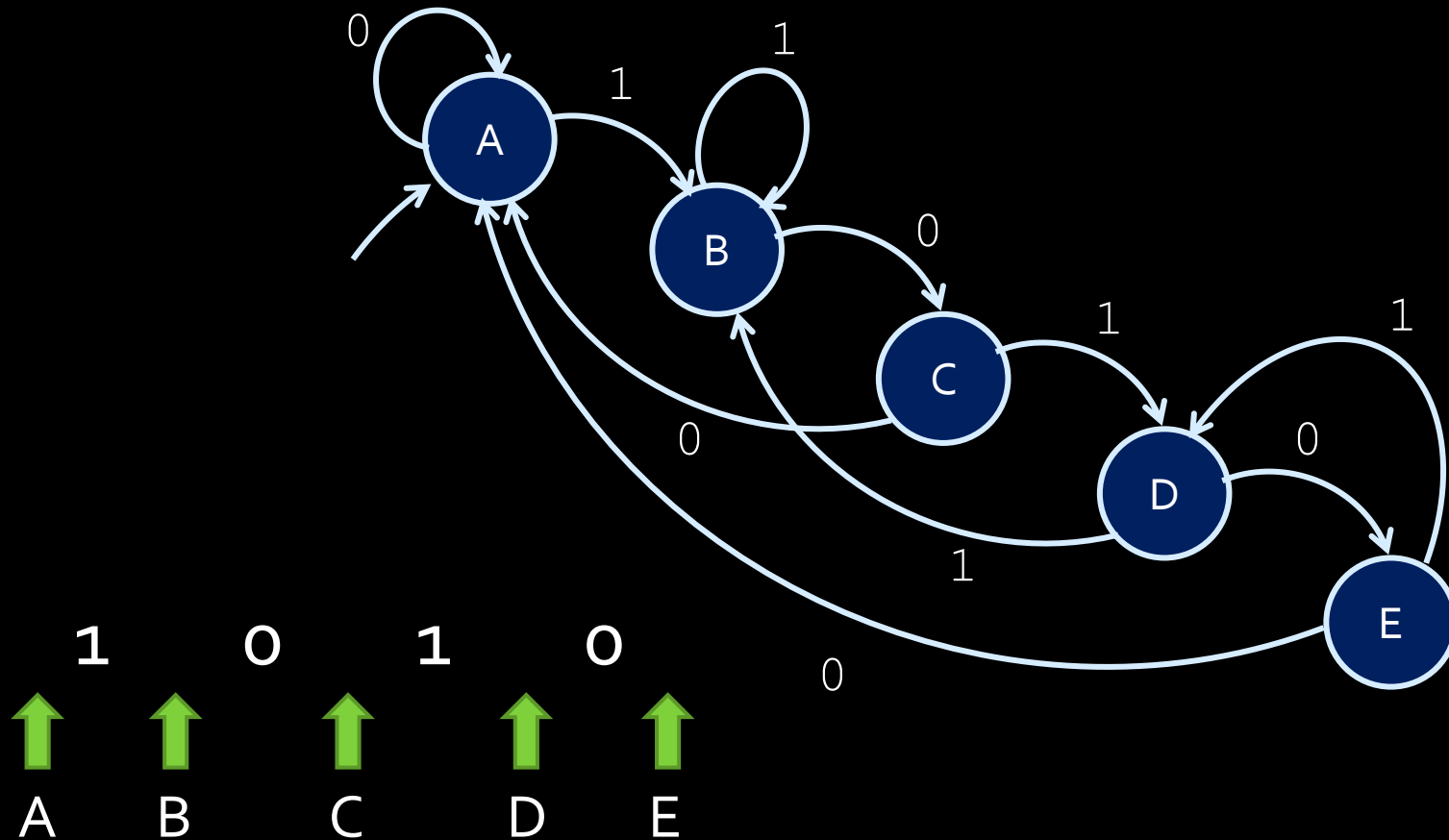
Step #1: Draw state diagram



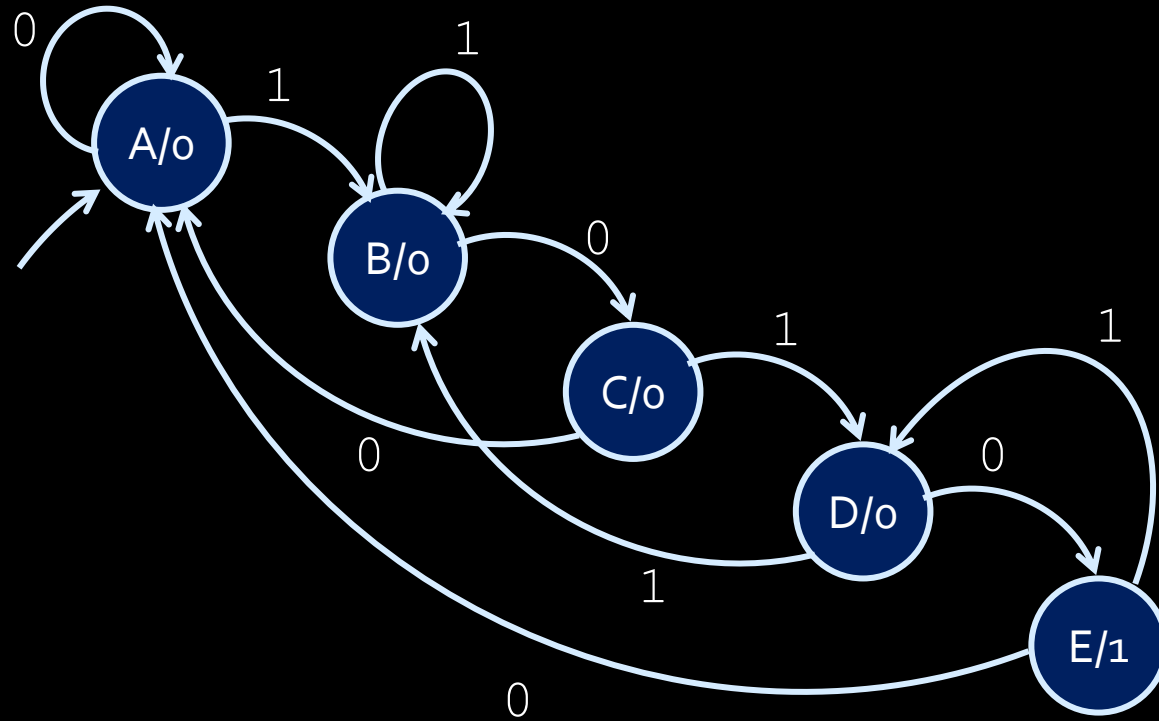
1 0 1 0

↑
A

Step #1: Draw state diagram

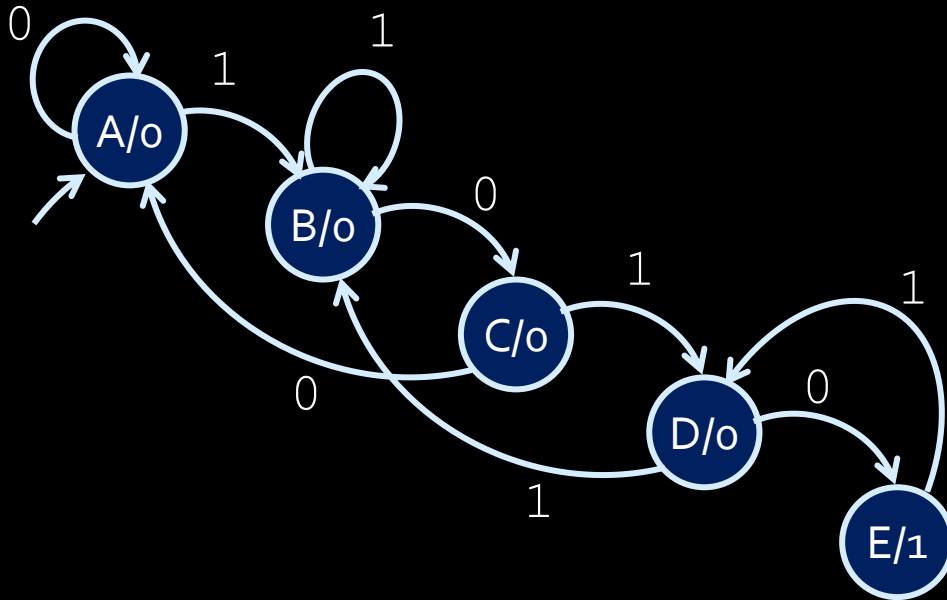


Step #1: ... and outputs



Step #2: State Table

- Write state table with Z

[illegible]

Step #2: State Table

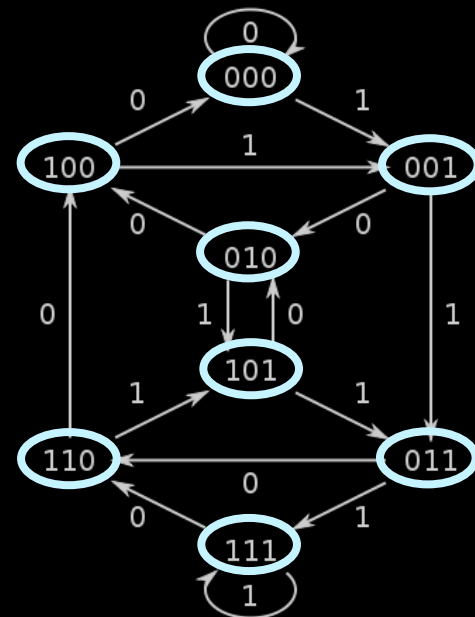
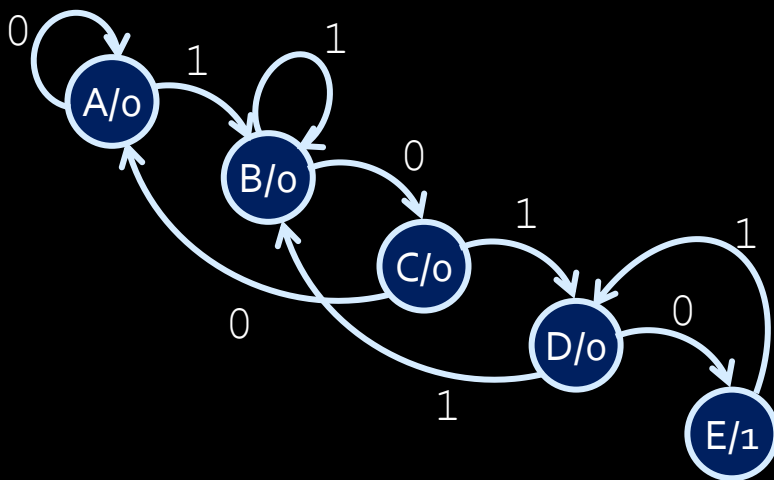
- Write state table with Z
- Output Z is determined by the current state.
 - Denotes Moore machine.
- Next step: allocate flip-flops values to each state.
 - How many flip-flops will we need for 5 states?
 - # flip-flops = $\lceil \log(\# \text{ of states}) \rceil$

Present State	X	Z	Next State
A	0	0	A
A	1	0	B
B	0	0	C
B	1	0	B
C	0	0	A
C	1	0	D
D	0	0	E
D	1	0	B
E	0	1	A
E	1	1	D

→ We need 3 flipflops

Discussion!

- Wait a minute... we built a recognizer before!
- We needed **8 states** to recognize **3-bit** pattern.
- Now we need only **5 states** for a **4-bit** pattern
- Why?

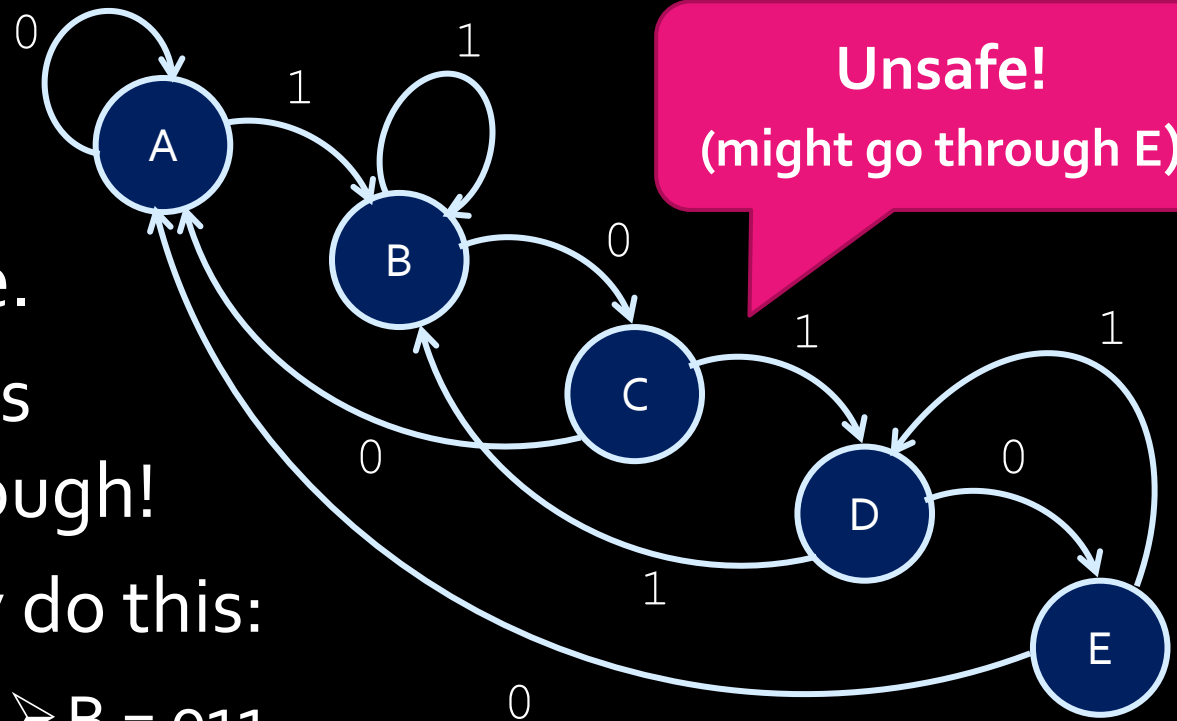


Discussion!

- The 3-bit pattern recognizer is doing more!
 - It is **memorizing** the last 3 bits.
 - To recognize another pattern, simply set the output of the relevant state to one.
 - Even multiple patterns by the same machine.
- The 4-bit pattern recognizer can only recognize **one specific pattern**.
 - Different pattern → different state machine.
 - To recognize n-bit pattern → need $n+1$ states.

Step #3: Flip-Flop Assignment

- 3 flip-flops needed here.
- Assign states carefully though!
- Can't simply do this:
 - A = 100 ➤ B = 011
 - C = 010 ➡ ➤ D = 001
 - E = 000



Why not?

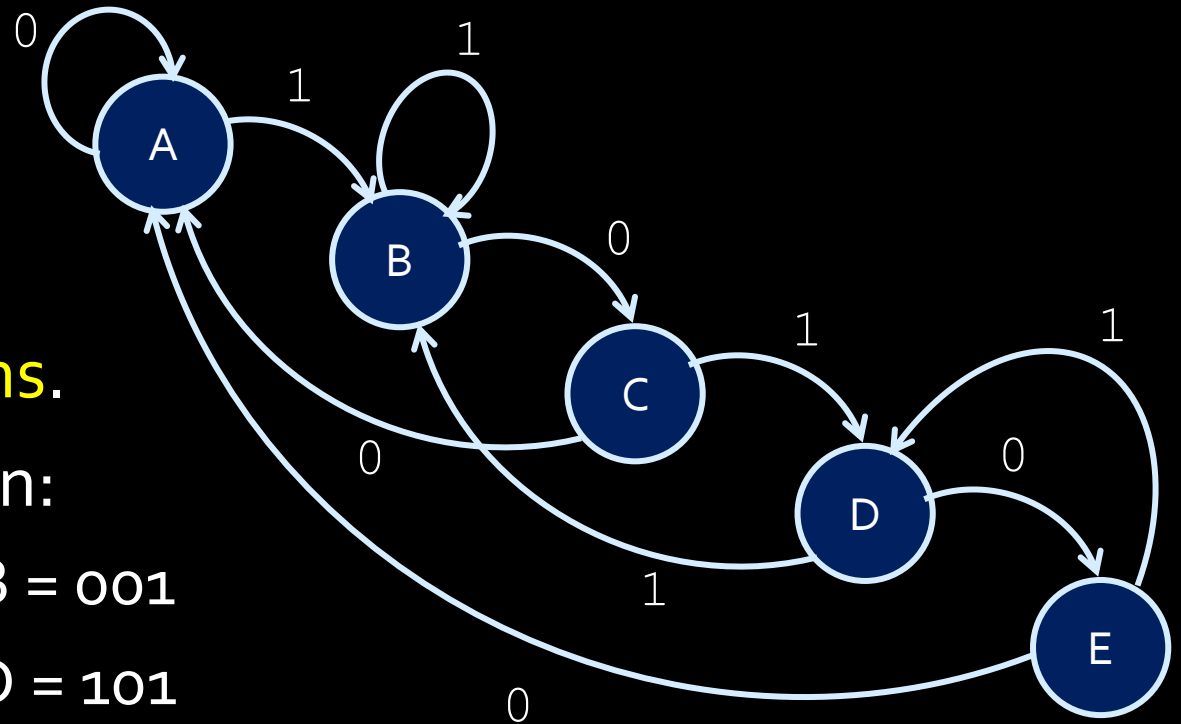
Step #3: Flip-Flop Assignment

- Be careful of **race conditions**.

- Better solution:

➤ A = 000 ➤ B = 001
➤ C = 011 ➤ D = 101
➤ E = 100

- Still has race conditions ($C \rightarrow D$, $C \rightarrow A$), but is safer.
 - “Safer” is defined according to output behaviour.
 - Sometimes, extra flip-flops are used for extra insurance.



Step #4: Redraw State Table

- We can now construct the K-maps for the state logic combinational circuit.
 - ▣ Derive equations for each flip-flop value, given the previous values and the input X .
 - ▣ Three equations total, plus one more for Z (trivial for Moore machines).

F_2	F_1	F_0	X	Z	F_2	F_1	F_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	0	1
0	1	1	0	0	0	0	0
0	1	1	1	0	1	0	1
1	0	1	0	0	1	0	0
1	0	1	1	0	0	0	1
1	0	0	0	1	0	0	0
1	0	0	1	1	1	0	1

Step 5: Circuit design

- Karnaugh map for F_2 :

	$\overline{F_0} \cdot \overline{X}$	$\overline{F_0} \cdot X$	$F_0 \cdot X$	$F_0 \cdot \overline{X}$
$\overline{F_2} \cdot \overline{F_1}$	0	0	0	0
$\overline{F_2} \cdot F_1$	X	X	1	0
$F_2 \cdot F_1$	X	X	X	X
$F_2 \cdot \overline{F_1}$	0	1	0	1

$$F_2 = F_1 X + F_2 \overline{F_0} X + F_2 F_0 \overline{X}$$

Step 5: Circuit design

- Karnaugh map for F_1 :

	$\overline{F_0} \cdot \overline{X}$	$\overline{F_0} \cdot X$	$F_0 \cdot X$	$F_0 \cdot \overline{X}$
$\overline{F_2} \cdot \overline{F_1}$	0	0	0	1
$\overline{F_2} \cdot F_1$	X	X	0	0
$F_2 \cdot F_1$	X	X	X	X
$F_2 \cdot \overline{F_1}$	0	0	0	0

$$F_1 = \overline{F_2} \overline{F_1} F_0 \overline{X}$$

Step 5: Circuit design

- Karnaugh map for F_0 :

	$\overline{F_0} \cdot \overline{X}$	$\overline{F_0} \cdot X$	$F_0 \cdot X$	$F_0 \cdot \overline{X}$
$\overline{F_2} \cdot \overline{F_1}$	0	1	1	1
$\overline{F_2} \cdot F_1$	X	X	1	0
$F_2 \cdot F_1$	X	X	X	X
$F_2 \cdot \overline{F_1}$	0	1	1	0

$$F_0 = X + \overline{F_2} \overline{F_1} F_0$$

Step 5: Circuit design

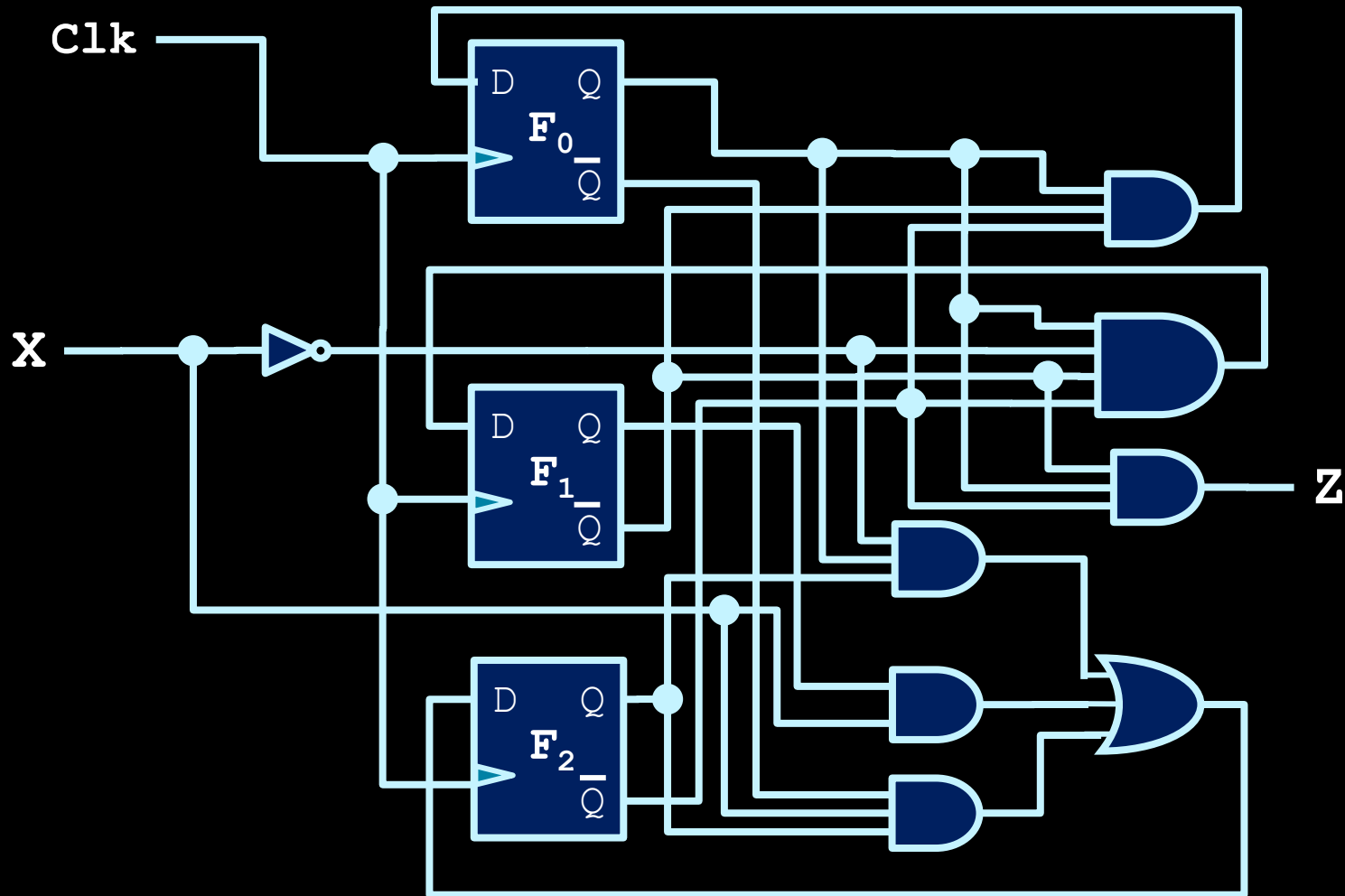
- Output value Z goes high for state E (100), so the output expression is:

$$Z = F_2 \overline{F_1} \overline{F_0}$$

- Note: All of these expressions would be different, given different flip-flop assignments!
 - Practice alternate assignments!

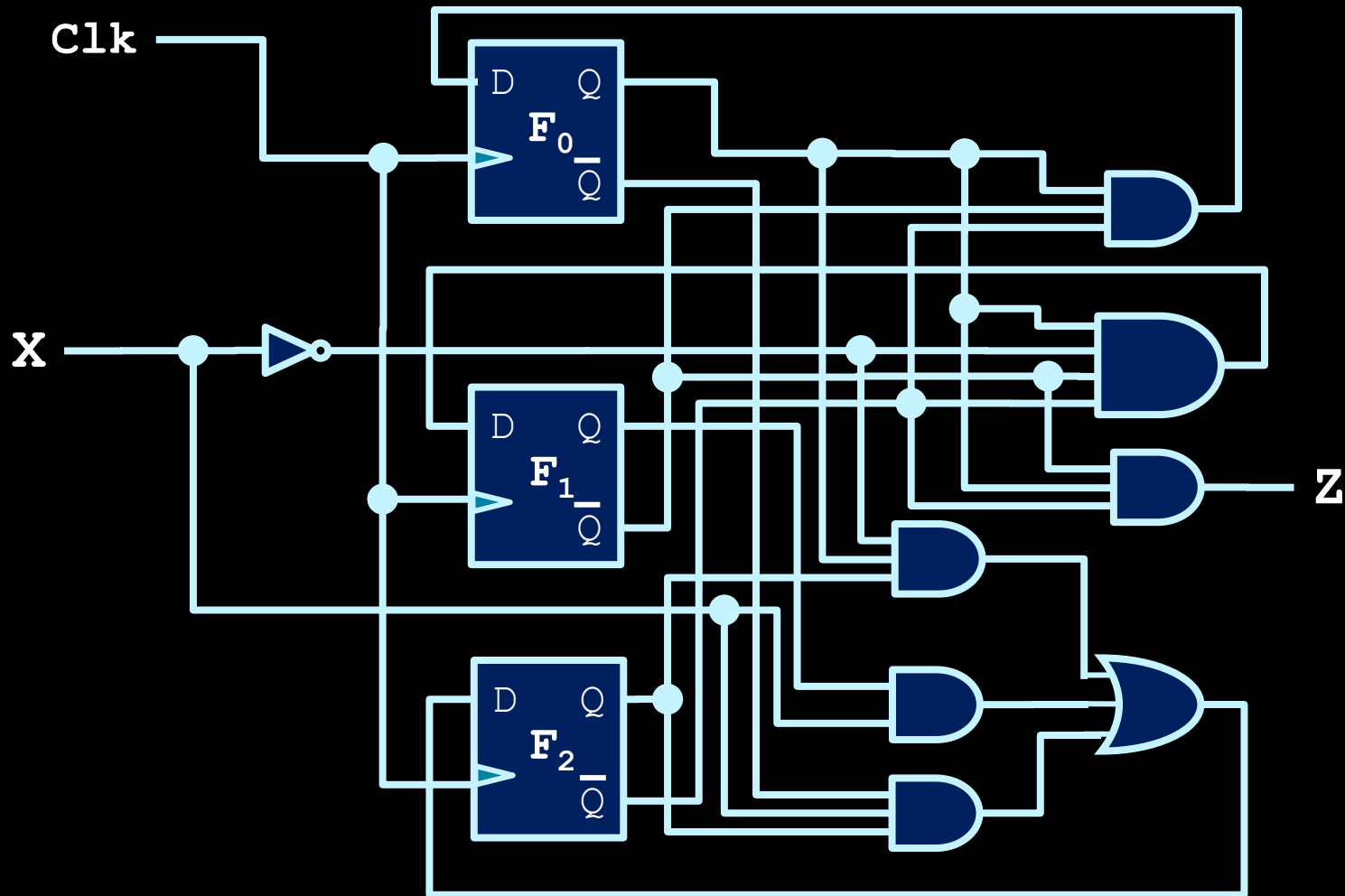
Circuit

Ugh....



Circuit

This is why people use automatic tools!

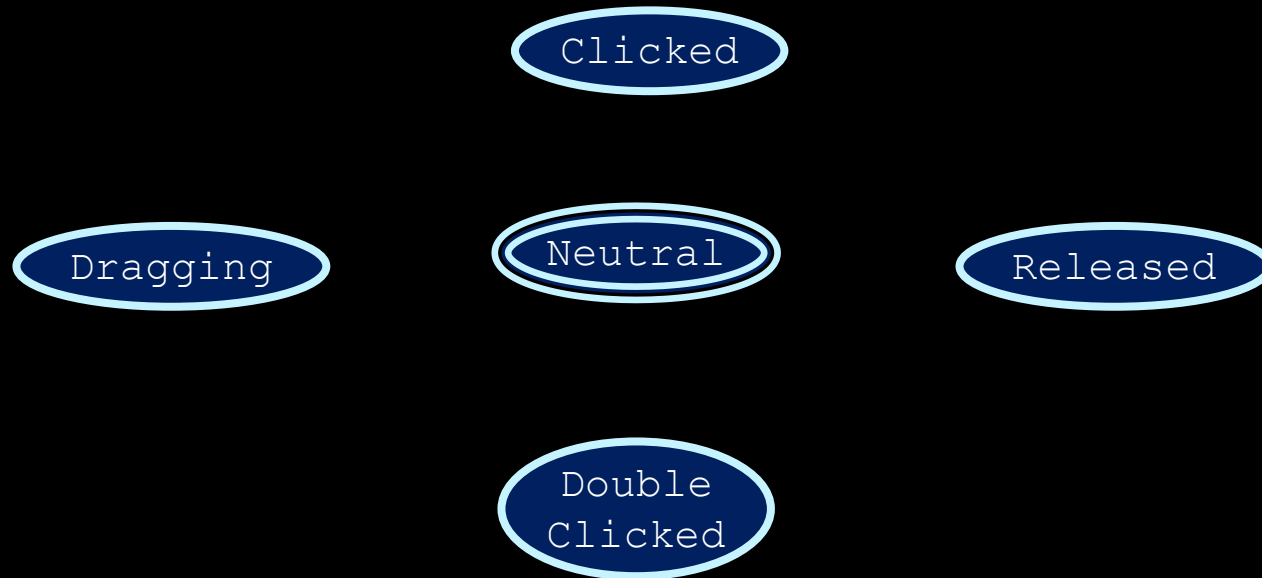


Question: Mouse clicks

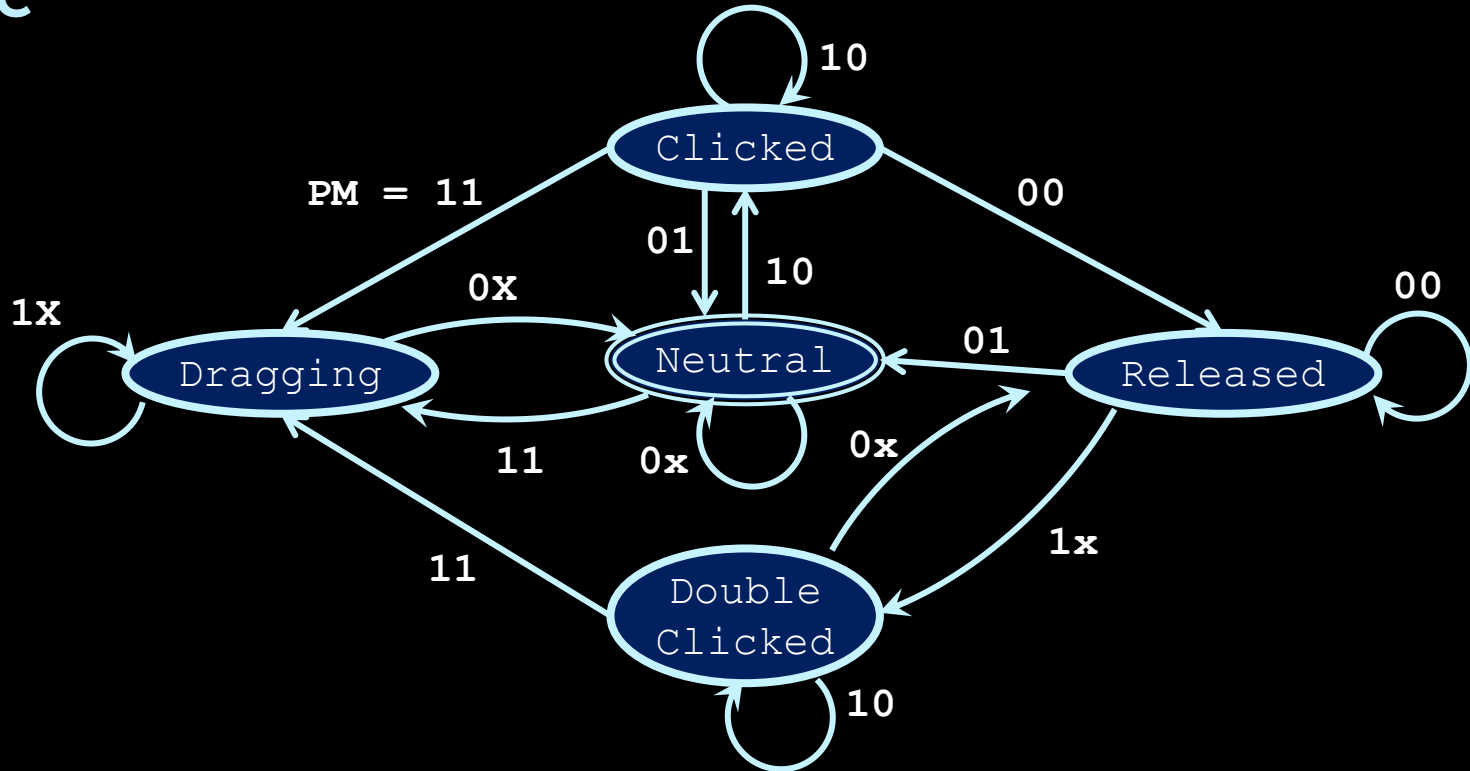
- Design a state machine that takes in two signals:
 - **P** is high if the user is pressing the mouse button.
 - **M** is high if the mouse is being moved.
- Based on the inputs, the state indicates whether the user is **clicking**, **double-clicking**, or **dragging** the mouse on the screen.
 - Hint: you will need more states than that!



Question: Mouse clicks



Question: Mouse clicks



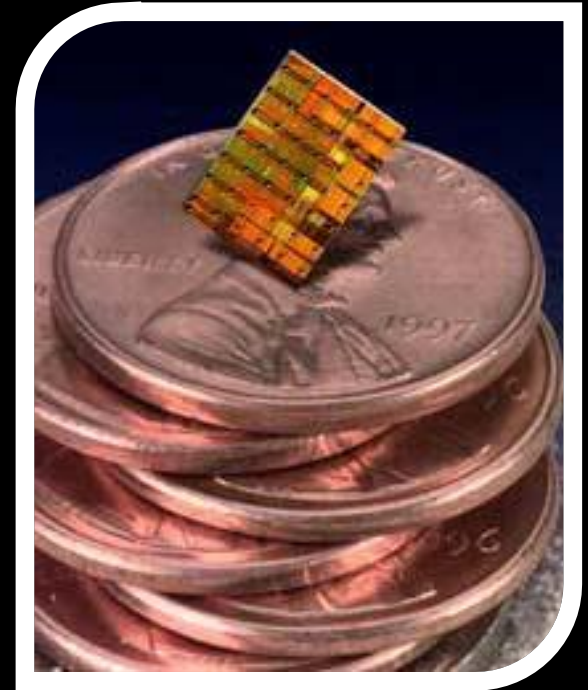
- Transitions indicate the values of P and M.
- X means this transition happens for both 0 and for 1.
- Outputs depend on the state (Moore machine)
- Home exercise: build this FSM.



Week 6: Processor Components

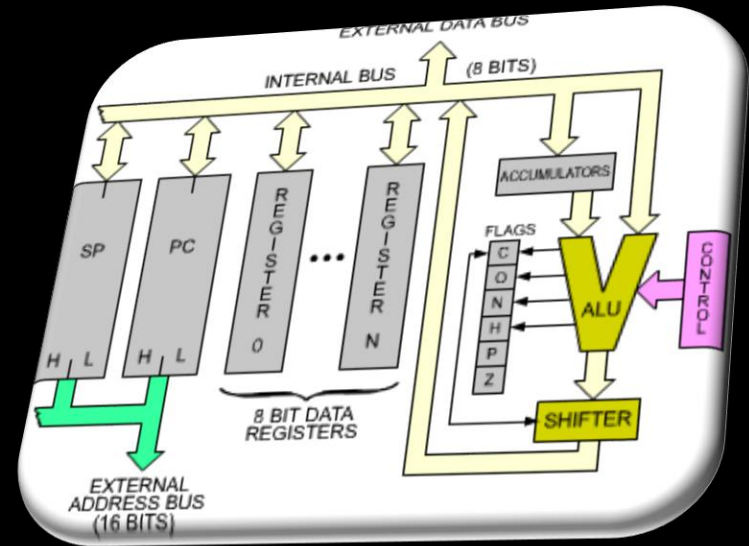
Microprocessors

- So far, we've been making devices, such such as adders, counters and registers.
- The ultimate goal is to make a **microprocessor**, which is a digital device that processes input, can store values and produces output, according to a set of on-board instructions.

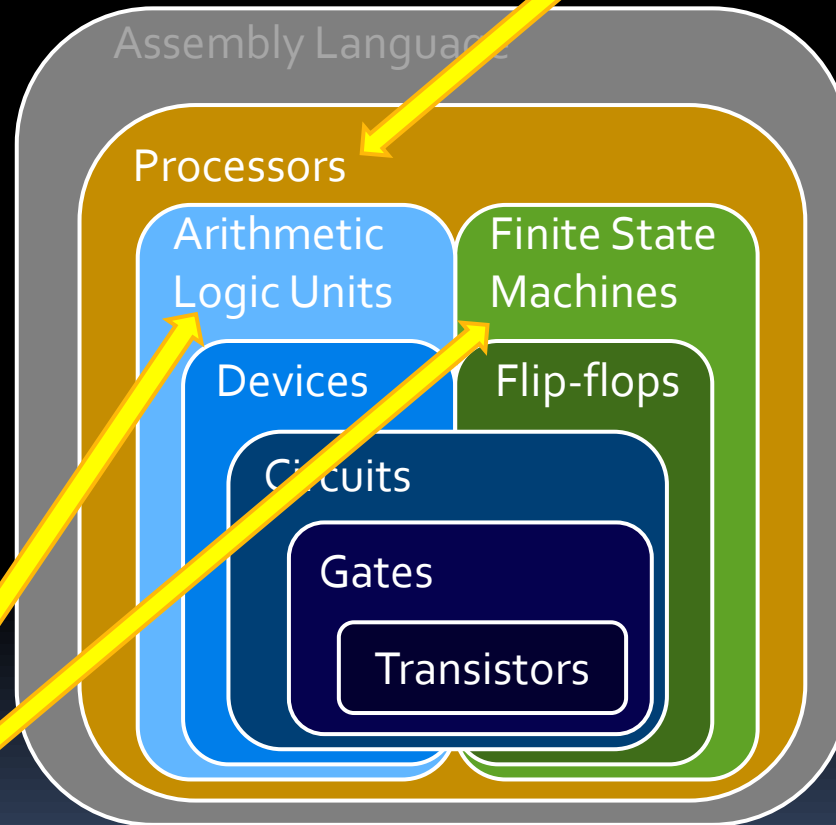


Microprocessors

- Microprocessors are a combination of the units that we've discussed so far:
 - Registers to store values.
 - Adders and shifters to process data.
 - Finite state machines to control the process.
- Microprocessors have been the basis of all computing since the 1970's, and can be found in nearly every sort of electronics.

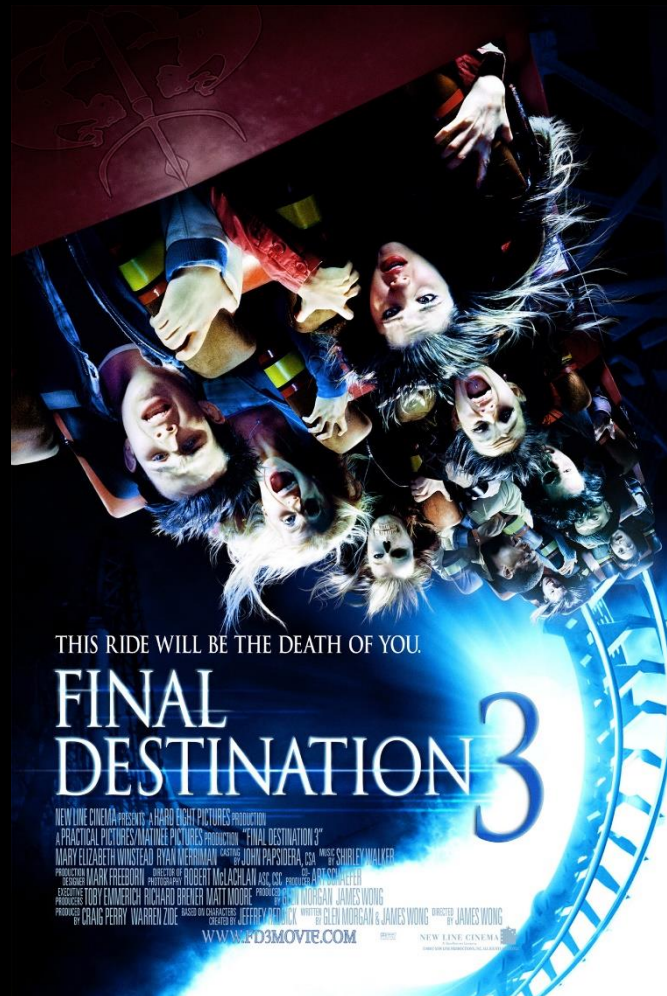


To get to this

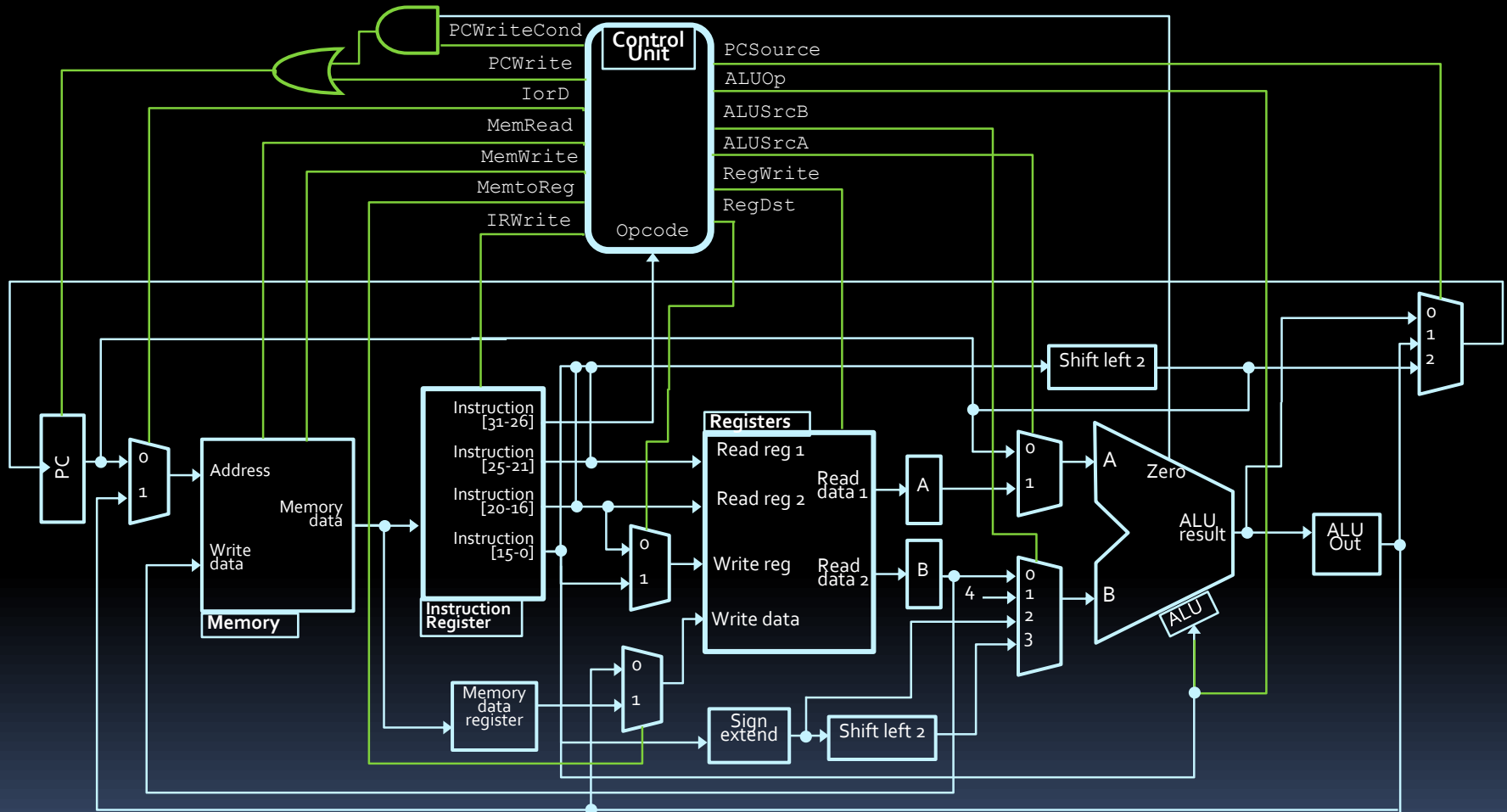


We build these

The Final Destination

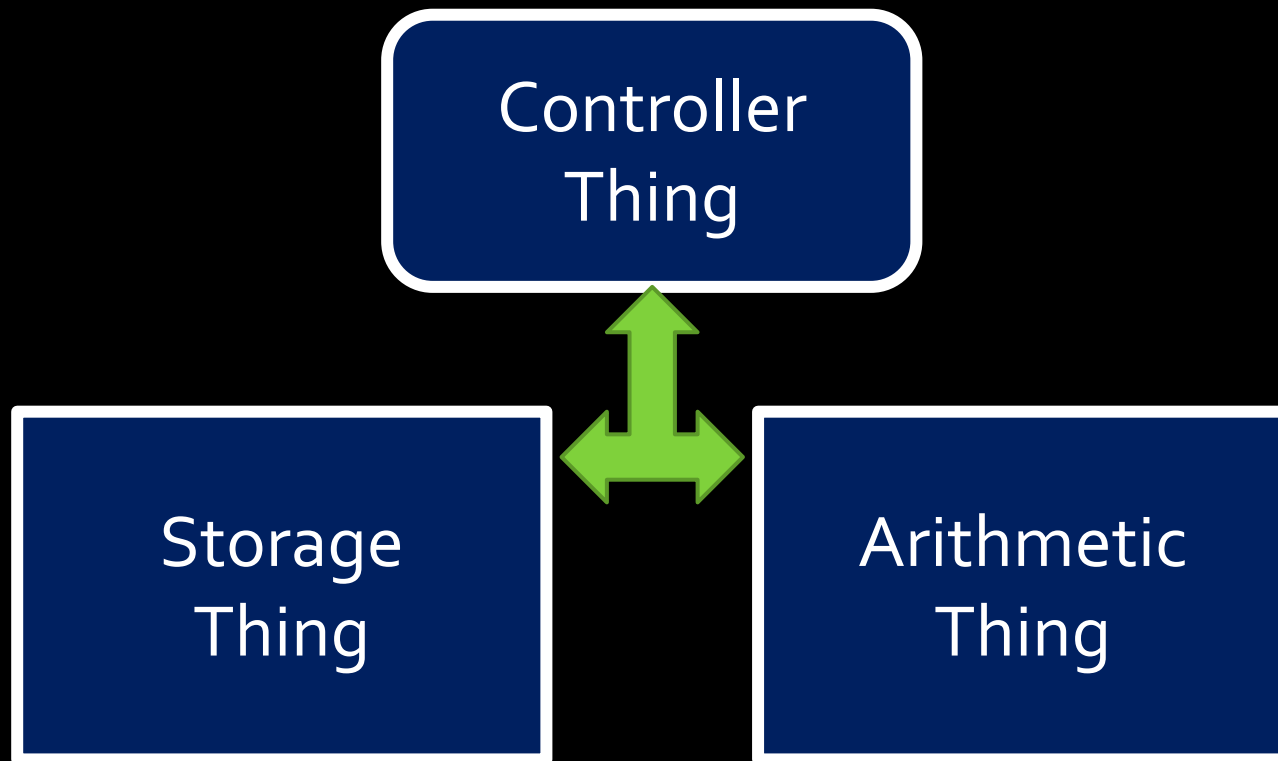


The Final Destination



Deconstructing processors

- Simpler at a high level:



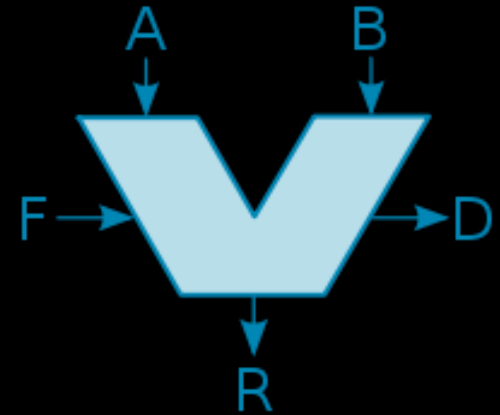
■ The “Arithmetic Thing”

aka: the Arithmetic Logic Unit (ALU)



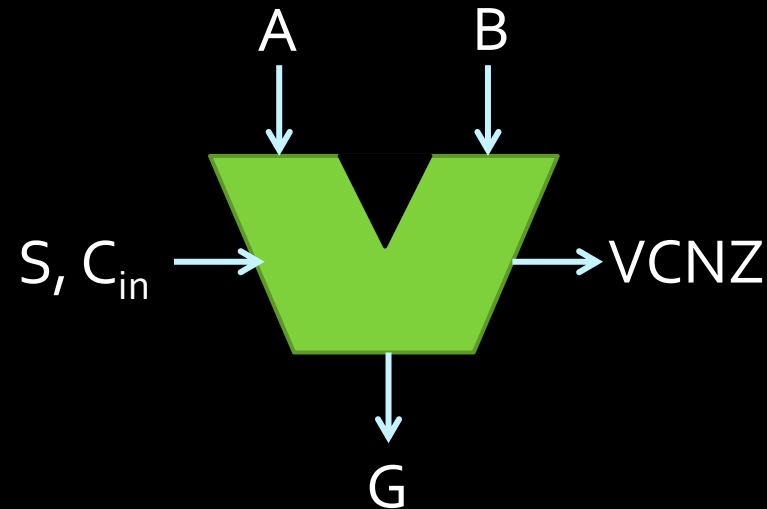
Arithmetic Logic Unit

- The first microprocessor applications were calculators.
 - Remember adders and subtractors?
 - These are part of a larger structure called the **arithmetic logic unit** (ALU).
 - You made a simple one in the labs!
- This larger structure is responsible for the processing of all data values in a basic CPU.



ALU inputs

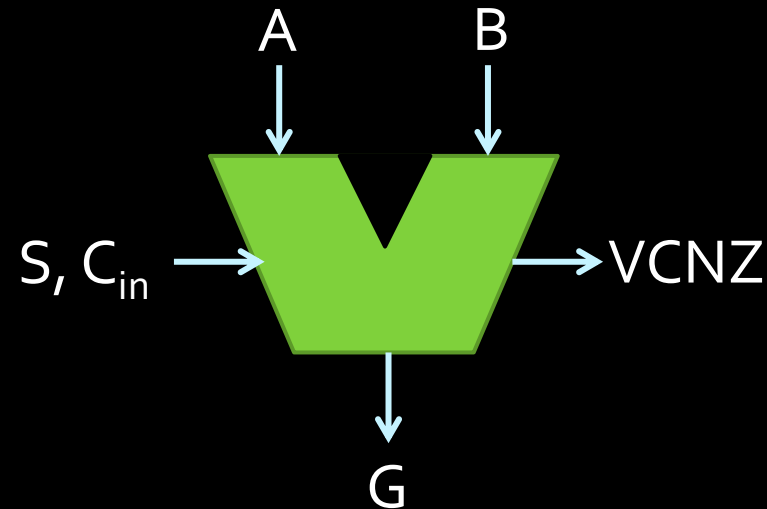
- The ALU performs all of the arithmetic operations covered in this course so far, and logical operations as well (AND, OR, NOT, etc.)
 - **Input S** represents **select bits** (in this case, S_2 S_1 & S_0) that specify which operation to perform.
 - For example: S_2 is a mode select bit, indicating whether the ALU is in arithmetic or logic mode
 - The **carry-in bit** C_{in} is used in operations such as incrementing an input value or the overall result.



ALU outputs

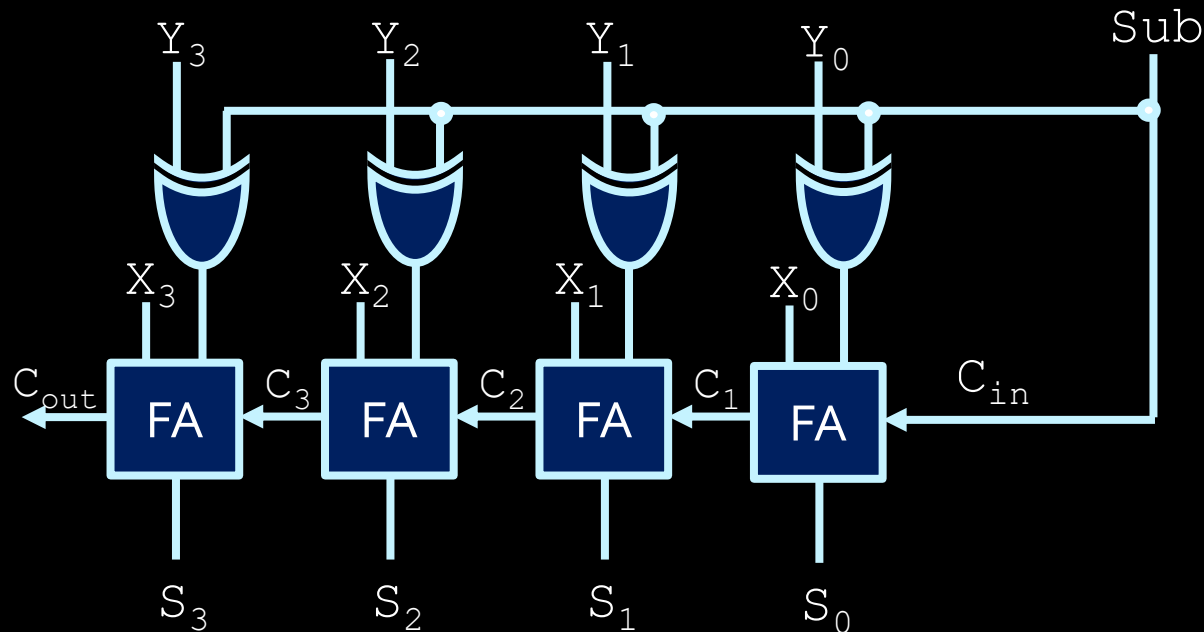
- In addition to the input signals, there are output signals V, C, N & Z which indicate special conditions in the arithmetic result:

- **V**: overflow condition
 - The result of the operation could not be stored in the n bits of G , meaning that the result is incorrect.
- **C**: carry-out bit
- **N**: Negative indicator
- **Z**: Zero-condition indicator (result is zero)

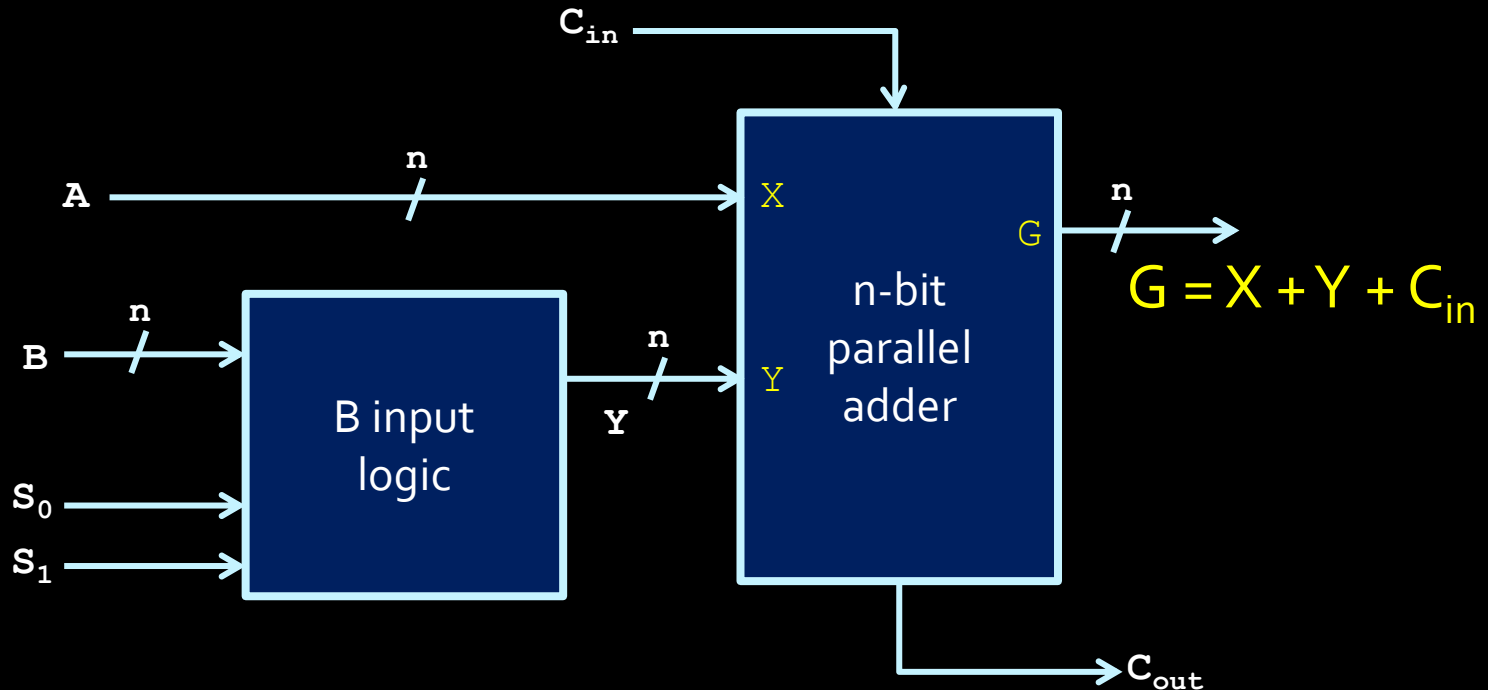


The “A” of ALU

- To understand how the ALU does all of these operations, let's start with the arithmetic side.
- Fundamentally, this side is made of an adder / subtractor unit, which we've seen already:




Arithmetic components



- In addition to addition and subtraction, many more operations can be performed by manipulating what is added to input A , as shown in the diagram above.

Arithmetic operations

$$1+1=3$$


- If the B input logic sends B straight through?
 - $Y = B \rightarrow$ Result of addition operation is $G = A+B$
 - What if B was replaced by all-ones instead?
 - $Y = 111\dots1 \rightarrow$ Result of addition operation: $G = A-1$
 - What if B was replaced by \overline{B} ?
 - $Y = \overline{B} \rightarrow$ Result of addition operation: $G = A-B-1$
 - And what if B was replaced by all zeroes?
 - $Y = 000\dots0 \rightarrow$ Result is: $G = A$.
 - We'll see later: this is useful for moving values between registers, and for loading values into registers.
- Instead of a Sub signal, **the operation you want is signaled using the select bits S_0 & S_1 .**

Operation selection $G = A + Y$

Select bits		Y Input	Result	Operation
s_1	s_0			
0	0	All 0s	$G = A$	Transfer
0	1	B	$G = A+B$	Addition
1	0	\bar{B}	$G = A+\bar{B}$	Subtraction - 1
1	1	All 1s	$G = A-1$	Decrement

- This is a good start! But something is missing...
- Wait, what about the carry-in bit?

Full operation selection

Select		Input	Operation	
S_1	S_0	Y	$C_{in}=0$	$C_{in}=1$
0	0	All 0s	$G = A$ (transfer)	$G = A+1$ (increment)
0	1	B	$G = A+B$ (add)	$G = A+B+1$
1	0	\bar{B}	$G = A+\bar{B}$	$G = A+\bar{B}+1$ (subtract)
1	1	All 1s	$G = A-1$ (decrement)	$G = A$ (transfer)

- Based on the values on the select bits and the carry bit, we can perform any number of basic arithmetic operations by manipulating what value is added to A .

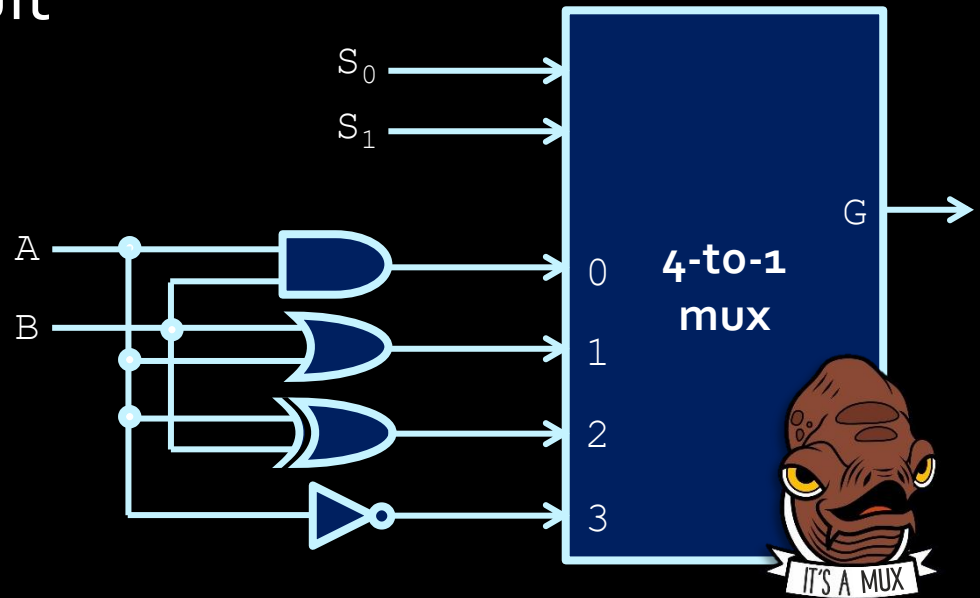
Full operation selection

Select		Input	Operation	
S_1	S_0	Y	$C_{in}=0$	$C_{in}=1$
0	0	All 0s	$G = A$ (transfer)	$G = A+1$ (increment)
0	1	B	$G = A+B$ (add)	$G = A+B+1$
1	0	\bar{B}	$G = A+\bar{B}$	$G = A+\bar{B}+1$ (subtract)
1	1	All 1s	$G = A-1$ (decrement)	$G = A$ (transfer)

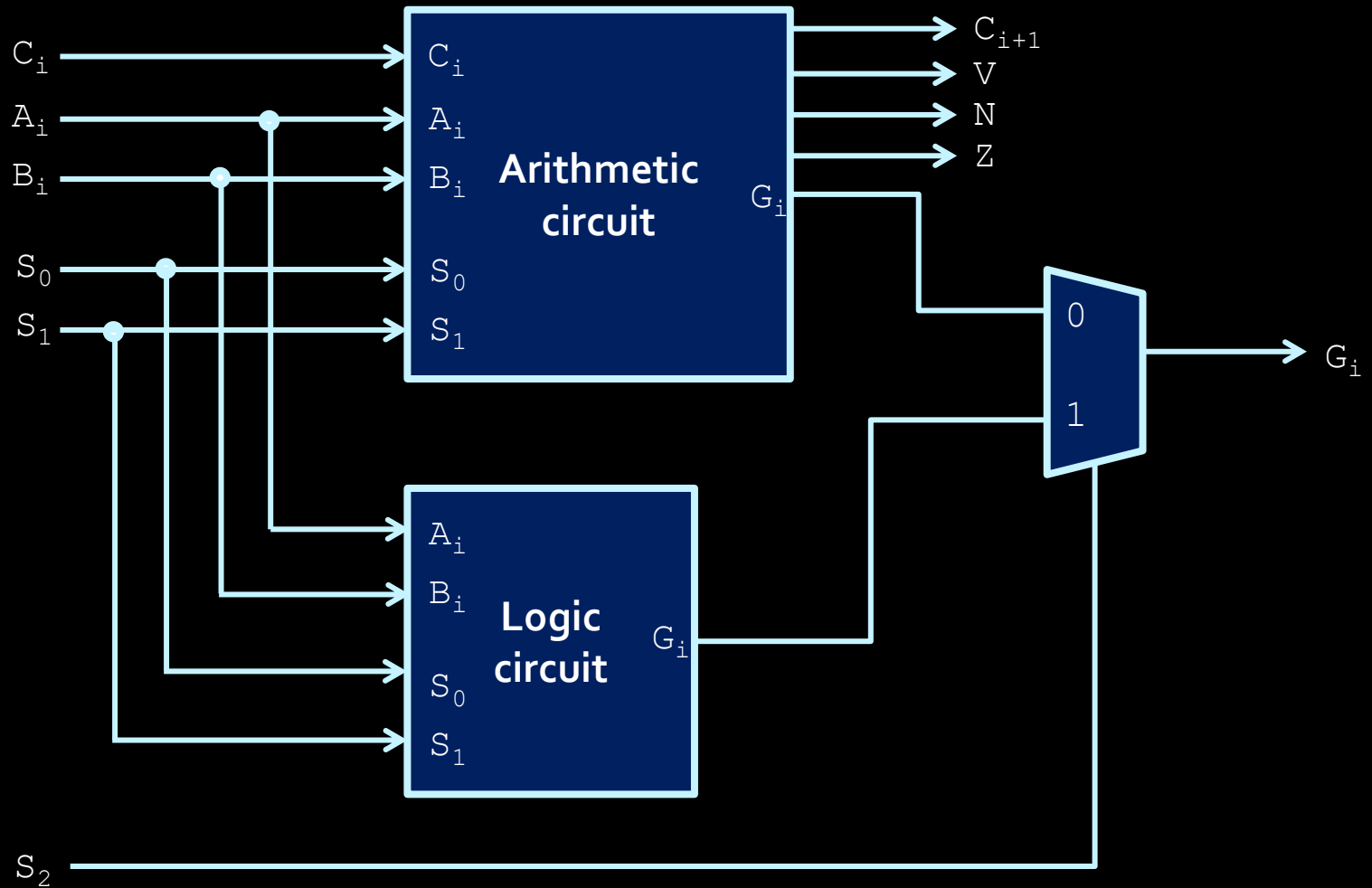
- Based on the values on the select bits and the carry bit, we can perform any number of basic arithmetic operations by manipulating what value is added to A.

The “L” of ALU

- We also want a circuit that can perform logical operations, in addition to arithmetic ones.
- How do we tell which operation to perform?
 - Another select bit!
- If $S_2 = 1$, then logic circuit block is activated.
- Multiplexer is used to determine which block (logical or arithmetic) goes to the output.

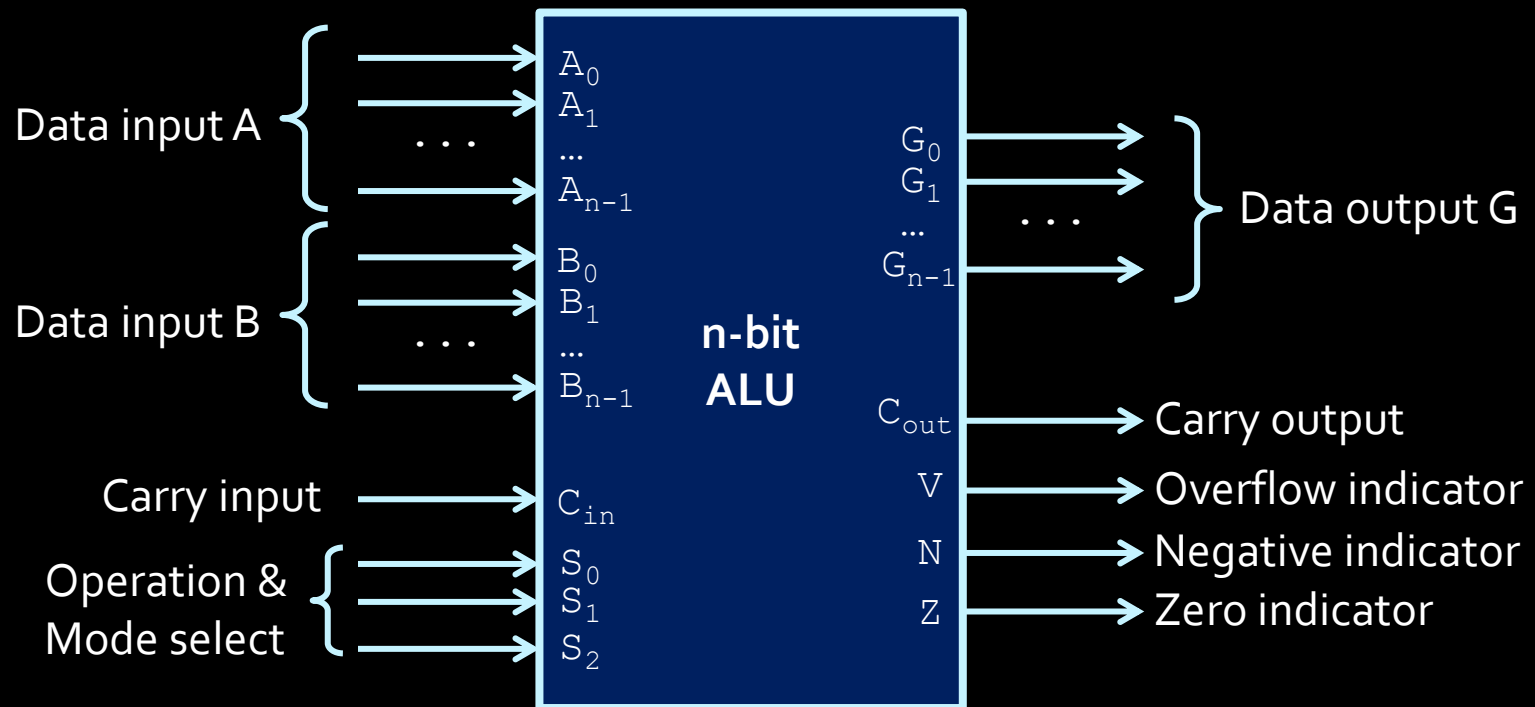


Single ALU Stage



ALU block diagram

- In addition to data inputs and outputs, this circuit also has:
 - ▣ outputs indicating the different conditions,
 - ▣ inputs specifying the operation to perform (similar to S_{ub}).



What about multiplication?

- Multiplication (and division) operations are more complicated than other arithmetic (plus, minus) or logical (AND, OR) operations.
- Three major ways that multiplication can be implemented in circuitry:
 - Layered rows of adder units.
 - An adder/shifter circuit with accumulator.
 - Booth's Algorithm

Multiplication

- Revisiting grade 3 math...

$$\begin{array}{r} 123 \\ \times 456 \\ \hline \end{array}$$

$$\begin{array}{r} 123 \\ \times 456 \\ \hline 1368 \end{array}$$

$$\begin{array}{r} 123 \\ \times 456 \\ \hline 1368 \\ 912 \end{array}$$

$$\begin{array}{r} 123 \\ \times 456 \\ \hline 1368 \\ 912 \\ 456 \end{array}$$

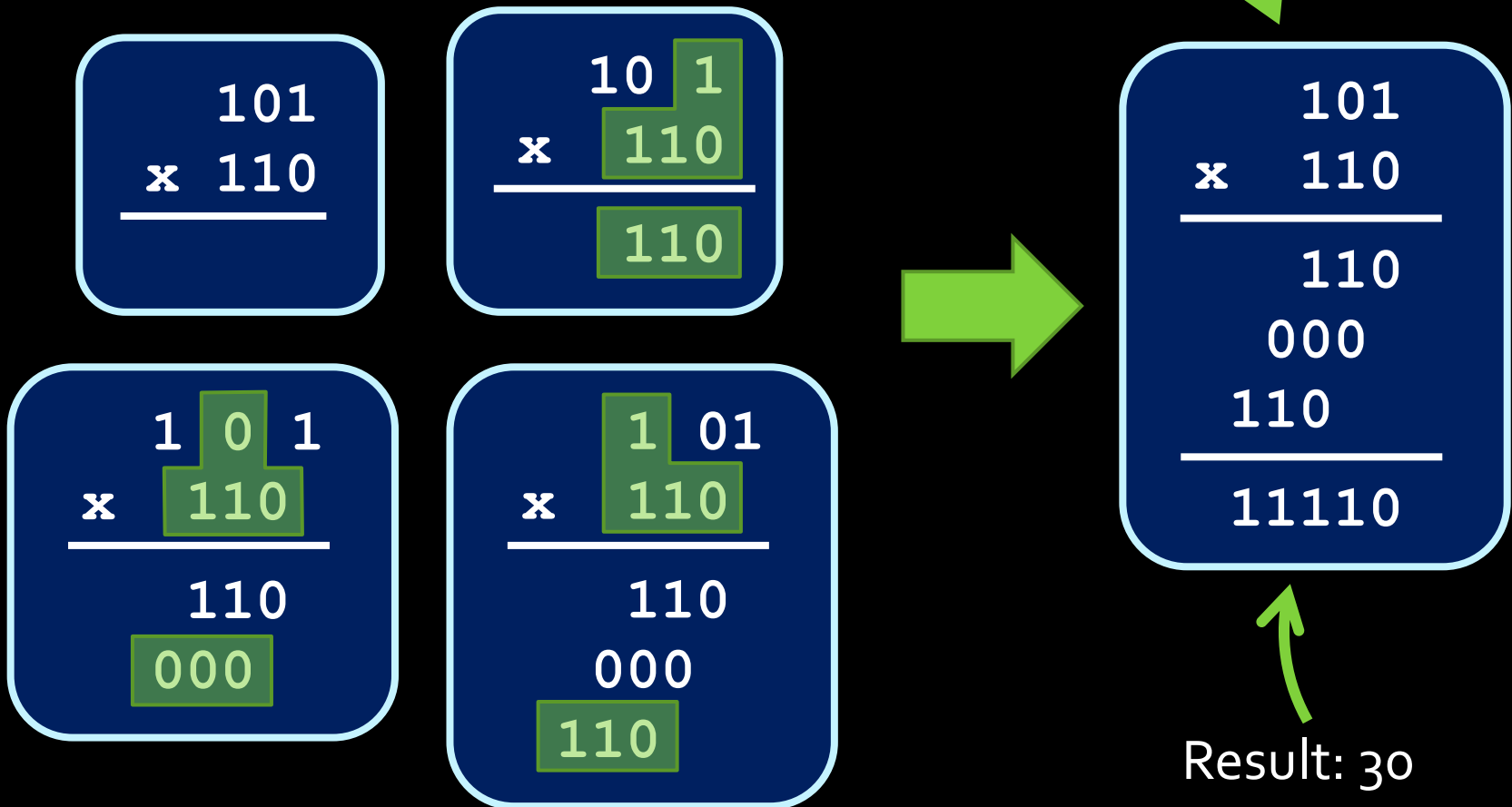


$$\begin{array}{r} 123 \\ \times 456 \\ \hline 1368 \\ 912 \\ 456 \\ \hline 56088 \end{array}$$

Binary Multiplication

- And now, in binary...

5*6 (unsigned)



Binary Multiplication

- Or seen another way....

$$\begin{array}{r} 101 \\ \times 110 \\ \hline \end{array}$$

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 000 \end{array}$$

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 000 \\ 101 \end{array}$$

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 000 \\ 101 \\ 101 \end{array}$$



$$\begin{array}{r} 101 \\ \times 110 \\ \hline 000 \\ 101 \\ 101 \\ \hline 11110 \end{array}$$

Observations

- What is “multiply by 1 bit in binary”?
 - $10101 \times 1 ?$ 10101
 - $10101 \times 0 ?$ 00000
 - It's an **AND**!
- Calculation flow
 - Multiply by 1 bit of multiplier (AND with bit)
 - Shift sum
 - Add partial sums (use adder)
 - Repeat the above

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 110 \\ 000 \\ 110 \\ \hline 11110 \end{array}$$

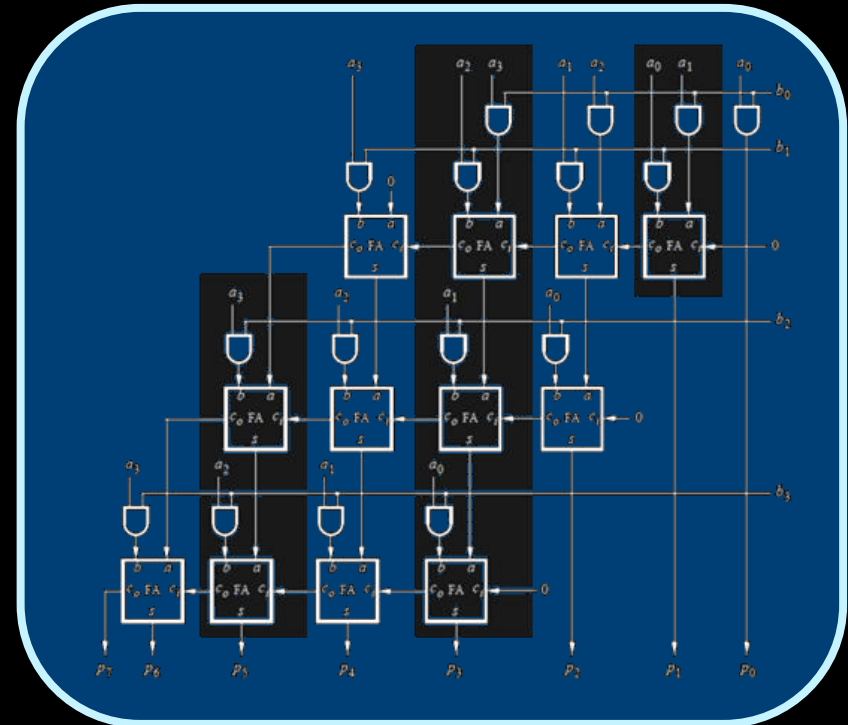
Binary Multiplication

				\times	a_3 b_3	a_2 b_2	a_1 b_1	a_0 b_0	
					a_3b_0	a_2b_0	a_1b_0	a_0b_0	
			a_3b_1	a_2b_1	a_1b_1	a_0b_1			
		a_3b_2	a_2b_2	a_1b_2	a_0b_2				
	a_3b_3	a_2b_3	a_1b_3	a_0b_3					
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0		

Implementation

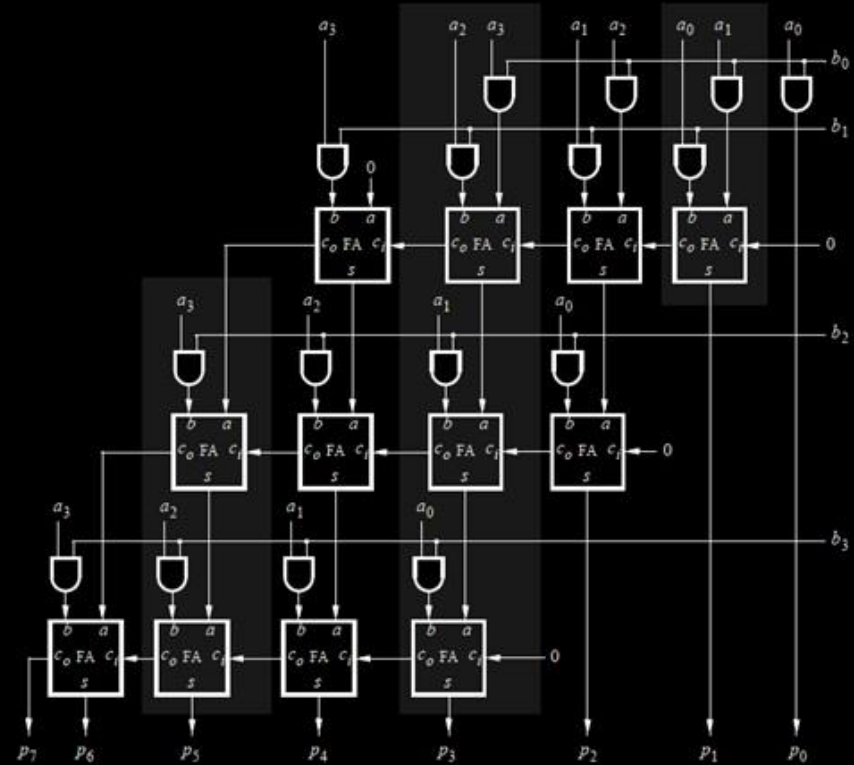
- Implementing this in circuitry involves the summation of several AND terms.
 - AND gates combine input signals.
 - Adders combine the outputs of the AND gates.

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
			\times				
				a_3b_0	a_2b_0	a_1b_0	a_0b_0
		a_3b_1	a_2b_1	a_1b_1	a_0b_1		
	a_3b_2	a_2b_2	a_1b_2	a_0b_2			
	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0



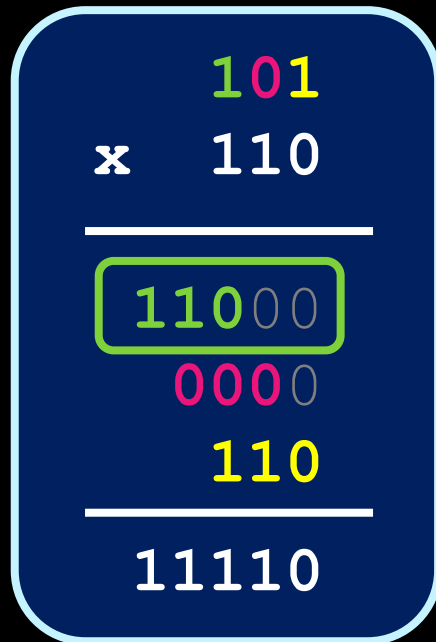
Multiplication

- This implementation results in an array of adder circuits to make the multiplier circuit.
- This can get a little expensive as the size of the operands grows.
 - N-bit numbers $\rightarrow O(1)$ clock cycles, but $O(N^2)$ size.
- Is there an alternative to this circuit?



Accumulator circuits

- What if you could perform each stage of the multiplication operation, one after the other?



$$\begin{array}{r} 101 \\ 110 \times 1 = \\ \hline \end{array}$$

Add into R

shift R to the left

0000000

110

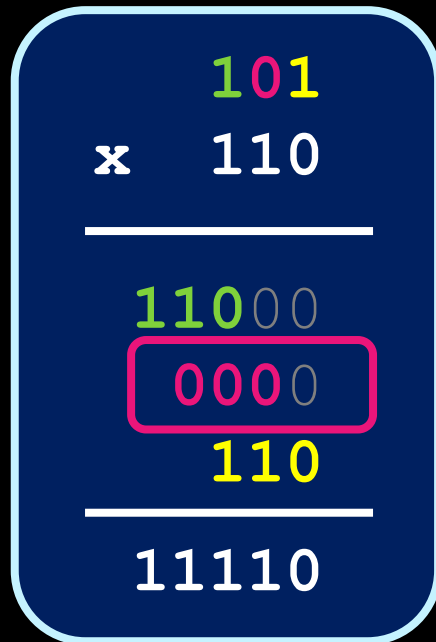
0000110

0001100

R

Accumulator circuits

- What if you could perform each stage of the multiplication operation, one after the other?



$$101 \times 110$$
$$110 \times 0 =$$

Add into R

shift R to the left

0001100

000

0001100

0011000

R

Accumulator circuits

- What if you could perform each stage of the multiplication operation, one after the other?

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 11000 \\ 0000 \\ \hline 11110 \end{array}$$

$$\begin{array}{r} 101 \quad 0011000 \\ 110 \times 1 = 110 \\ \quad 0011110 \end{array}$$

Done! Result
is ready

Accumulator circuits

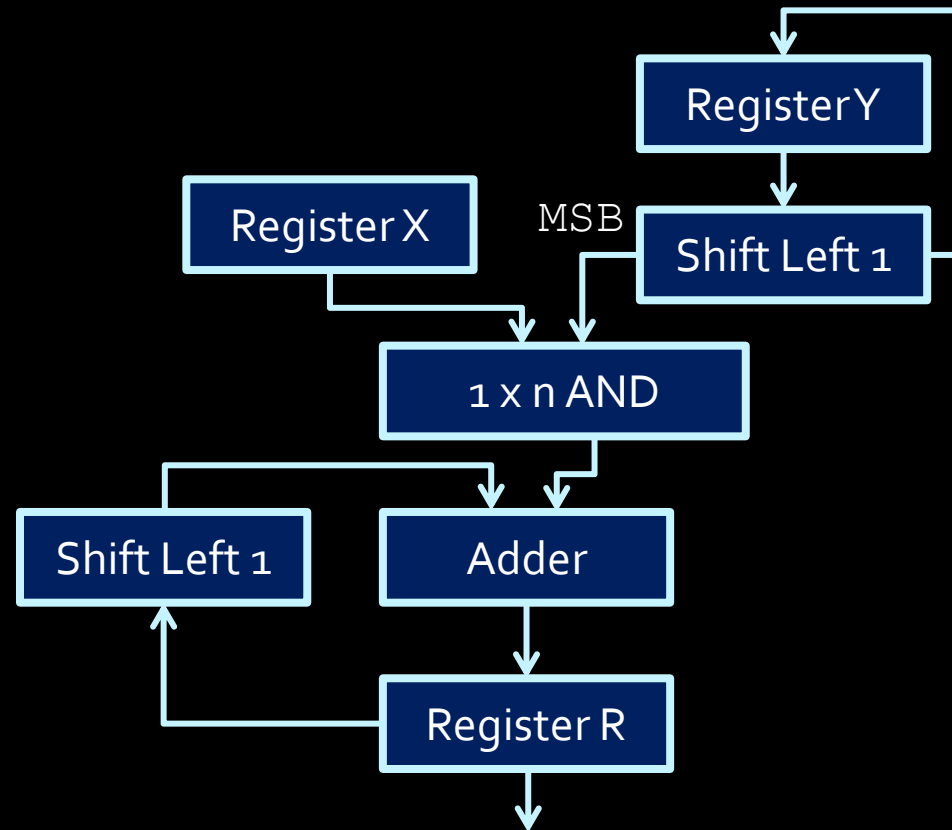
- We can implement the same algorithm in hardware:
 - Use shift register and check the MSB to get bits of **101**
 - Use shift register with parallel load to store and shift **R**
 - Or use load register with shifter circuit
 - Use adder to compute:
 $R = R + 110 \times 1$

A hand-drawn style binary multiplication diagram on a dark blue background with rounded corners. It shows the multiplication of 101 by 110. The multiplier 101 is at the top, with the first bit 1 in green, 0 in pink, and 1 in yellow. Below it is the multiplicand 110 in white. A horizontal line separates the multiplier from the partial products. There are three partial products: 11000 (110 in green, 00 in grey), 0000 (000 in pink, 0 in grey), and 110 (110 in yellow). Another horizontal line separates the partial products from the final result 11110 at the bottom.

$$\begin{array}{r} \text{101} \\ \times \text{110} \\ \hline \text{11000} \\ \text{0000} \\ \text{110} \\ \hline \text{11110} \end{array}$$

Accumulator circuits

- This circuit needs only a single row of adders and a couple of shift registers.
- How wide does register R have to be?
 - When multiplying **n-bit** number by **k-bit** number, result may need up to **n+k bits**.



Accumulator circuits

- This circuit requires only $O(N)$ hardware, and requires N clock cycles.
- This circuit performs one addition for every bit.
- Can we use fewer additions?
- Booth's algorithm exploits sequences of ones and zeros in real data.
- Move to next part.

