

ECE/CS 250

Computer Architecture

Summer 2019

Basics of Logic Design: Boolean Algebra, Logic Gates, and the ALU (Combinational Logic)

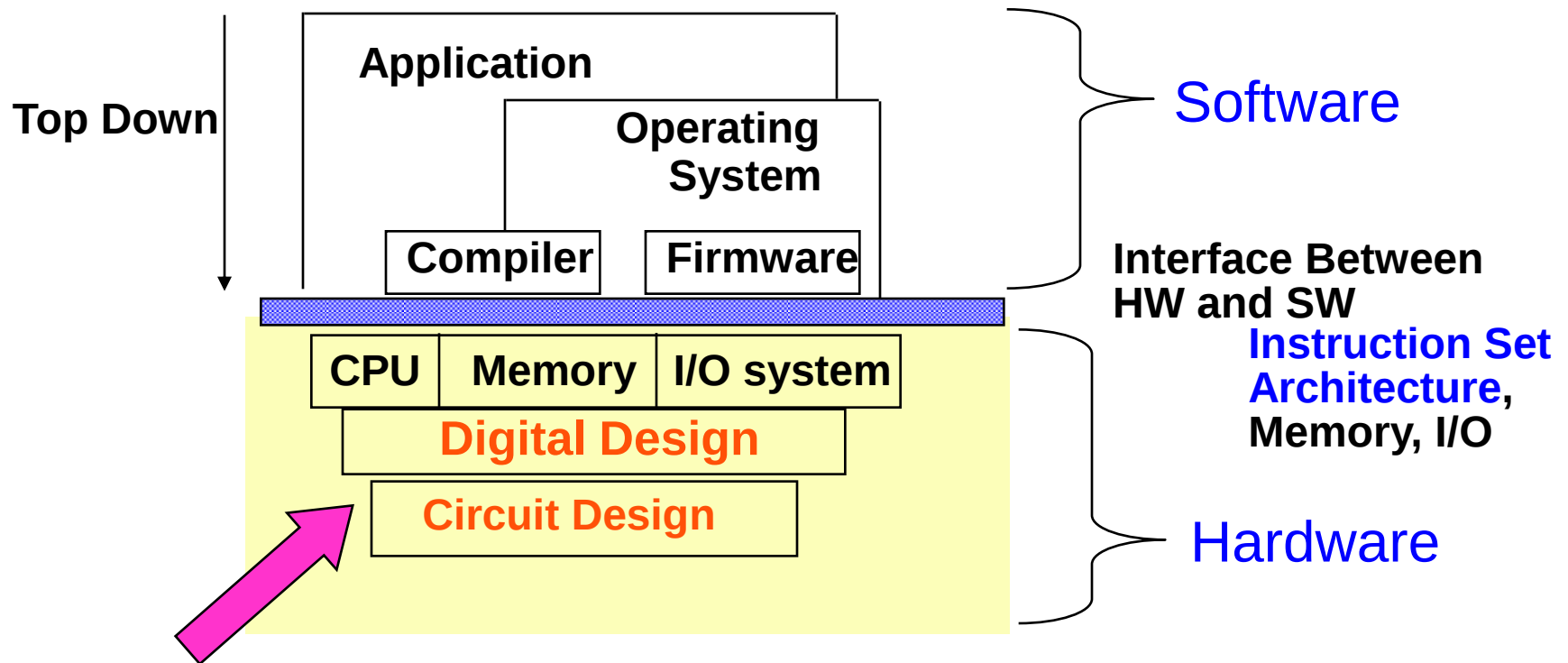
Tyler Bletsch
Duke University

Slides are derived from work by
Daniel J. Sorin (Duke), Alvy Lebeck (Duke), and Drew Hilton (Duke)

Reading

- Appendix B (parts 1,2,3,5,6,7,8,9,10)
- This material is covered in MUCH greater depth in ECE/CS 350 – please take ECE/CS 350 if you want to learn enough digital design to build your own processor

What We've Done, Where We're Going



(Almost) Bottom UP to CPU

Computer = Machine That Manipulates Bits

- Everything is in binary (bunches of 0s and 1s)
 - Instructions, numbers, memory locations, etc.
- Computer is a machine that operates on bits
 - Executing instructions → operating on bits
- Computers physically made of transistors
 - Electrically controlled switches
- We can use transistors to build logic
 - E.g., if this bit is a 0 and that bit is a 1, then set some other bit to be a 1
 - E.g., if the first 5 bits of the instruction are 10010 then set this other bit to 1 (to tell the adder to subtract instead of add)

How Many Transistors Are We Talking About?

Pentium III

- Processor Core 9.5 Million Transistors
- Total: 28 Million Transistors

Pentium 4

- Total: 42 Million Transistors

Core2 Duo (two processor cores)

- Total: 290 Million Transistors

Core2 Duo Extreme (4 processor cores, 8MB cache)

- Total: 590 Million Transistors

Core i7 with 6-cores

- Total: 2.27 Billion Transistors

How do they design such a thing? Carefully!

Abstraction!

- Use of **abstraction** (key to design of any large system)
 - Put a few (2-8) transistors into a **logic gate** (or, and, xor, ...)
 - **Combine gates into logical functions** (add, select,...)
 - Combine adders, shifters, etc., together into modules
 - Units with well-defined interfaces for large tasks: e.g., decode
 - Combine a dozen of those into a core...
 - Stick 4 cores on a chip...

Boolean Algebra

- First step to logic: Boolean Algebra
 - Manipulation of True / False (1/0)
 - After all: everything is just 1s and 0s
- Given inputs (variables): A, B, C, P, Q...
 - Compute outputs using logical operators, such as:
- NOT: $\neg A$ ($= \sim A = \bar{A}$)
- AND: $A \& B$ ($= A \cdot B = A * B = AB = A \wedge B$) = `A&&B` in C/C++
- OR: $A \mid B$ ($= A + B = A \vee B$) = `A || B` in C/C++
- XOR: $A \wedge B$ ($= A \oplus B$)
- NAND, NOR, XNOR, Etc.

Truth Tables

- Can represent as **truth table**: shows outputs for all inputs

a	NOT (a)
0	1
1	0

a	b	AND (a , b)
0	0	0
0	1	0
1	0	0
1	1	1

a	b	OR (a , b)
0	0	0
0	1	1
1	0	1
1	1	1

a	b	XOR (a , b)
0	0	0
0	1	1
1	0	1
1	1	0

a	b	XNOR (a , b)
0	0	1
0	1	0
1	0	0
1	1	1

a	b	NOR (a , b)
0	0	1
0	1	0
1	0	0
1	1	0

Any Inputs, Any Outputs

- Can have any # of inputs, any # of outputs
- Can have arbitrary functions:

a	b	c	f_1	f_2
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	1	0
1	1	0	0	1
1	1	1	1	1

Let's Write a Truth Table for a Function...

- Example:
(A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

A	B	C	Output

Let's write a Truth Table for a function...

- Example:
(A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

Fill in Inputs

Counting in Binary

A	B	C	Output
0	0	0	

Let's write a Truth Table for a function...

- Example:
(A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

Fill in Inputs

Counting in Binary

A	B	C	Output
0	0	0	
0	0	1	

Let's write a Truth Table for a function...

- Example:
(A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

Fill in Inputs

Counting in Binary

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

Let's write a Truth Table for a function...

- Example:
(A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

Fill in Inputs

Counting in Binary

Compute Output

$$(0 \& 0) \mid !0 = 0 \mid 1 = 1$$

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

Let's write a Truth Table for a function...

- Example:
(A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

Fill in Inputs

Counting in Binary

Compute Output

$(0 \& 0) \mid !1 = 0 \mid 0 = 0$

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

Let's write a Truth Table for a function...

- Example:
(A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

Fill in Inputs

Counting in Binary

Compute Output

$(0 \& 1) \mid !0 = 0 \mid 1 = 1$

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

Let's write a Truth Table for a function...

- Example:
(A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

Fill in Inputs

Counting in Binary

Compute Output

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



Logisim example

basic_logic.circ : example1

Suppose I turn it around...

- Given a Truth Table, find the formula?

Hmmm..

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

Suppose I turn it around...

- Given a Truth Table, find the formula?

Hmmm ...

Could write down every "true" case

Then OR together:

$(\neg A \ \& \ \neg B \ \& \ \neg C) \mid$

$(\neg A \ \& \ \neg B \ \& \ C) \mid$

$(\neg A \ \& \ B \ \& \ \neg C) \mid$

$(A \ \& \ B \ \& \ \neg C) \mid$

$(A \ \& \ B \ \& \ C)$

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

Suppose I turn it around...

- Given a Truth Table, find the formula?

Hmmm..

Could write down every "true" case

Then OR together:

$(\neg A \ \& \ \neg B \ \& \ \neg C) \mid$

$(\neg A \ \& \ \neg B \ \& \ C) \mid$

$(\neg A \ \& \ B \ \& \ \neg C) \mid$

$(A \ \& \ B \ \& \ \neg C) \mid$

$(A \ \& \ B \ \& \ C)$

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

Suppose I turn it around...

- Given a Truth Table, find the formula?

Hmmm..

Could write down every "true" case

Then OR together:

$(\neg A \ \& \ \neg B \ \& \ \neg C) \mid$

$(\neg A \ \& \ \neg B \ \& \ C) \mid$

$(\neg A \ \& \ B \ \& \ \neg C) \mid$

$(A \ \& \ B \ \& \ \neg C) \mid$

$(A \ \& \ B \ \& \ C)$

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

Suppose I turn it around...

- This approach: “sum of products”
 - Works every time
 - Result is right...
 - But really ugly

$(\neg A \ \& \ \neg B \ \& \ \neg C) \mid$
 $(\neg A \ \& \ \neg B \ \& \ C) \mid$
 $(\neg A \ \& \ B \ \& \ \neg C) \mid$
 $(A \ \& \ B \ \& \ \neg C) \mid$
 $(A \ \& \ B \ \& \ C)$

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

Suppose I turn it around...

- This approach: “sum of products”
 - Works every time
 - Result is right...
 - But really ugly

(!A & !B & !C) |

(!A & !B & C) |

(!A & B & !C) |

(A & B & !C) |

(A & B & C)

Could just be (A & B) here ?

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

Suppose I turn it around...

- This approach: “sum of products”
 - Works every time
 - Result is right...
 - But really ugly

(!A & !B & !C) |
(!A & !B & C) |
(!A & B & !C) |
(A&B)

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

Suppose I turn it around...

- This approach: “sum of products”
 - Works every time
 - Result is right...
 - But really ugly

(!A & !B & !C) |

(!A & !B & C) |

(!A & B & !C) |

(A&B)

Could just be (!A & !B) here

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

Suppose I turn it around...

- This approach: “sum of products”
 - Works every time
 - Result is right...
 - But really ugly

$(\neg A \ \& \ \neg B) \mid$
 $(\neg A \ \& \ B \ \& \ \neg C) \mid$
 $(A \ \& \ B)$

Could just be $(\neg A \ \& \ \neg B)$ here

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

Suppose I turn it around...

- This approach: “sum of products”
 - Works every time
 - Result is right...
 - But really ugly

$(\neg A \ \& \ \neg B) \mid$
 $(\neg A \ \& \ B \ \& \ \neg C) \mid$
 $(A \ \& \ B)$

Looks nicer...

Can we do better?

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

Suppose I turn it around...

- This approach: “sum of products”
 - Works every time
 - Result is right...
 - But really ugly

$(\neg A \ \& \ \neg B) \mid$
 $(\neg A \ \& \ B \ \& \ \neg C)$
 $(A \ \& \ B)$



This has a lot in common:
 $\neg A \ \& \ (\text{something})$

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

Just did some of these by intuition.. but

- Somewhat intuitive approach to simplifying
- This is **math**, so there are formal rules
 - Just like “regular” algebra

Boolean Function Simplification

- Boolean expressions can be simplified by using the following rules (bitwise logical):

- $A \& A = A$

- $A \& 0 = 0$

- $A \& 1 = A$

- $A \& !A = 0$

$$A \mid A = A$$

$$A \mid 0 = A$$

$$A \mid 1 = 1$$

$$A \mid !A = 1$$

- $!!A = A$

- $\&$ and $\mid$ are both commutative and associative

- $\&$ and $\mid$ can be distributed: $A \& (B \mid C) = (A \& B) \mid (A \& C)$

- $\&$ and $\mid$ can be subsumed: $A \mid (A \& B) = A$

DeMorgan's Laws

- Two (less obvious) Laws of Boolean Algebra:
 - Let's push negations inside, flipping & and |

$$\neg (A \ \& \ B) = (\neg A) \ | \ (\neg B)$$

$$\neg (A \ | \ B) = (\neg A) \ \& \ (\neg B)$$

- You should try this at home – build truth tables for both the left and right sides and see that they're the same

Simplification Example:

$\neg (\neg A \mid \neg (A \ \& \ (B \mid C)))$

DeMorgan's

$\neg \neg A \ \& \ \neg \neg (A \ \& \ (B \mid C))$

Double Negation Elimination

$A \ \& \ (A \ \& \ (B \mid C))$

Associativity of $\&$

$(A \ \& \ A) \ \& \ (B \mid C)$

$A \ \& \ A = A$

$A \ \& \ (B \mid C)$

You try this:

Come up with a formula for this Truth Table

Simplify as much as possible

Sum of Products:

$(\neg A \ \& \ \neg B \ \& \ \neg C) \mid$

$(\neg A \ \& \ B \ \& \ \neg C) \mid$

$(A \ \& \ \neg B \ \& \ C) \mid$

$(A \ \& \ B \ \& \ C)$

Simplify this part

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

You try this:

Simplify:

$$(\neg A \ \& \ \neg B \ \& \ \neg C) \mid (\neg A \ \& \ B \ \& \ \neg C)$$

Regroup (associative/commutative):

$$((\neg A \ \& \ \neg C) \ \& \ \neg B) \mid ((\neg A \ \& \ \neg C) \ \& \ B)$$

Un-distribute (factor):

$$(\neg A \ \& \ \neg C) \ \& \ (\neg B \mid B)$$

OR identities:

$$(\neg A \ \& \ \neg C) \ \& \ \text{true} \quad = \quad (\neg A \ \& \ \neg C)$$

You try this:

Come up with a formula for this Truth Table

Simplify as much as possible

Sum of Products: Result of simplifying

$(\neg A \ \& \ \neg C)$ |

$(A \ \& \ \neg B \ \& \ C)$ |

$(A \ \& \ B \ \& \ C)$

You can simplify this part in the same way...

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

You try this:

Come up with a formula for this Truth Table

Simplify as much as possible

Sum of Products:

$(\neg A \ \& \ \neg C) \mid$
 $(A \ \& \ C)$

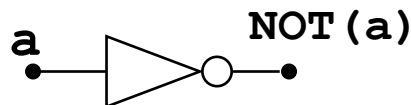
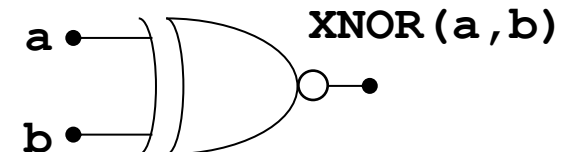
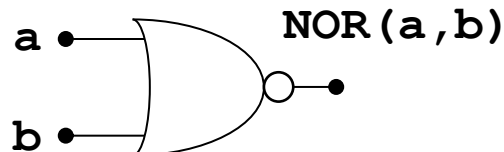
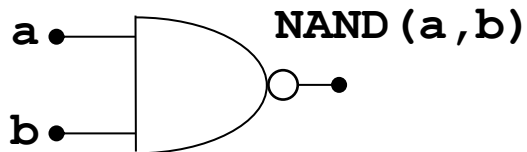
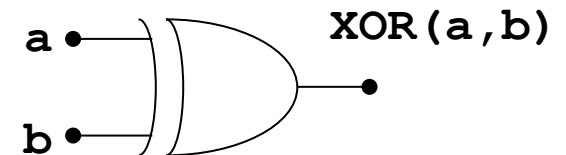
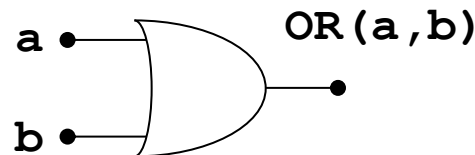
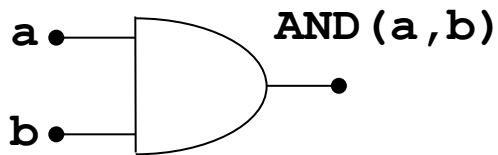
A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Applying the Theory

- Lots of good theory
- Can reason about complex Boolean expressions
- But why is this useful?

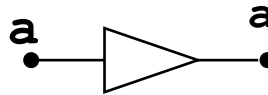
Boolean Gates

- **Gates** are electronic devices that implement simple Boolean functions (building blocks of hardware)

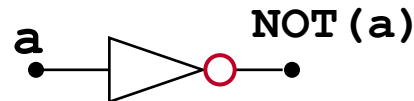


Guide to Remembering your Gates

- This one looks like it just points its input where to go
 - It just produces its input as its output
 - Called a buffer



- A circle always means negate (invert)

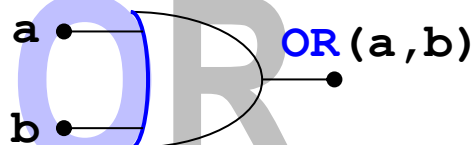


Circle = NOT

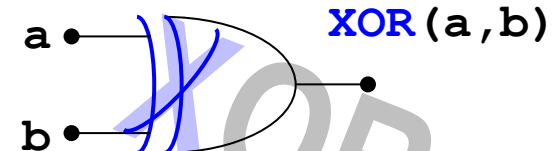
Guide to Remembering your Gates



Straight like an A

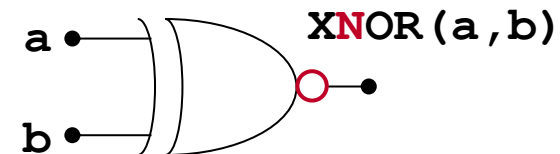
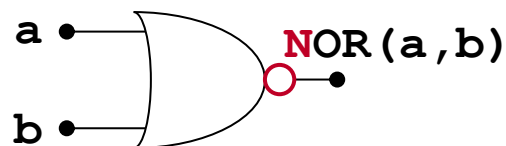
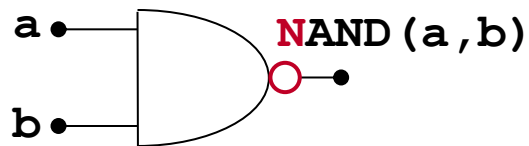


Curved, like an O

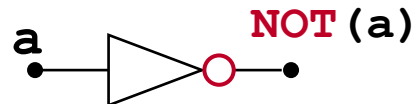


XOR looks like OR (curved line),
but has two lines (like an X does)

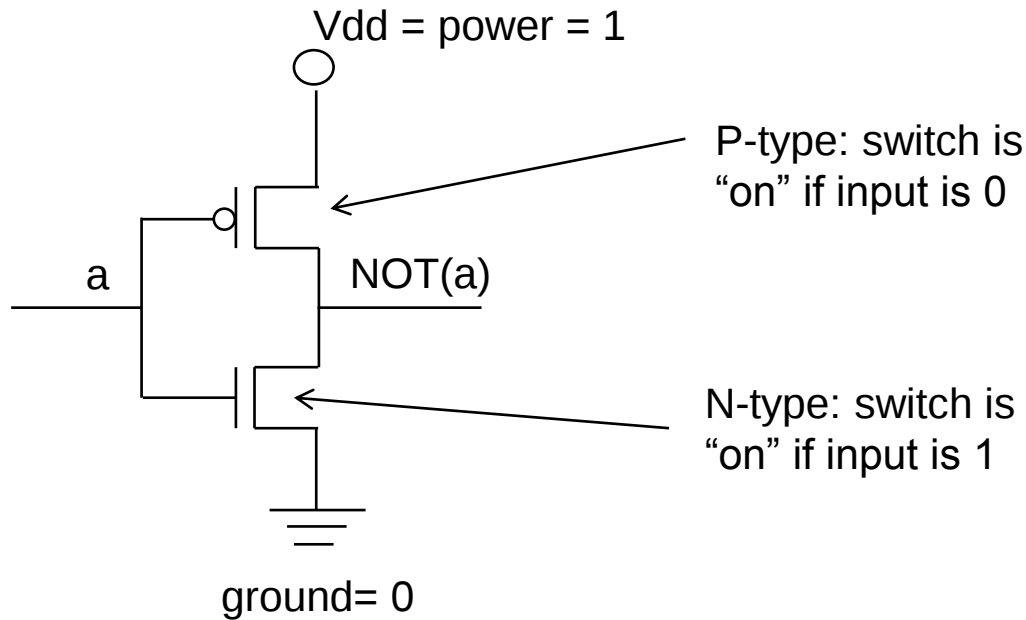
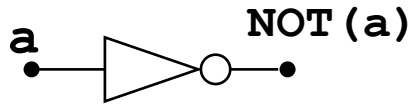
Circle means NOT



(XNOR is 1-bit "equals" by the way)



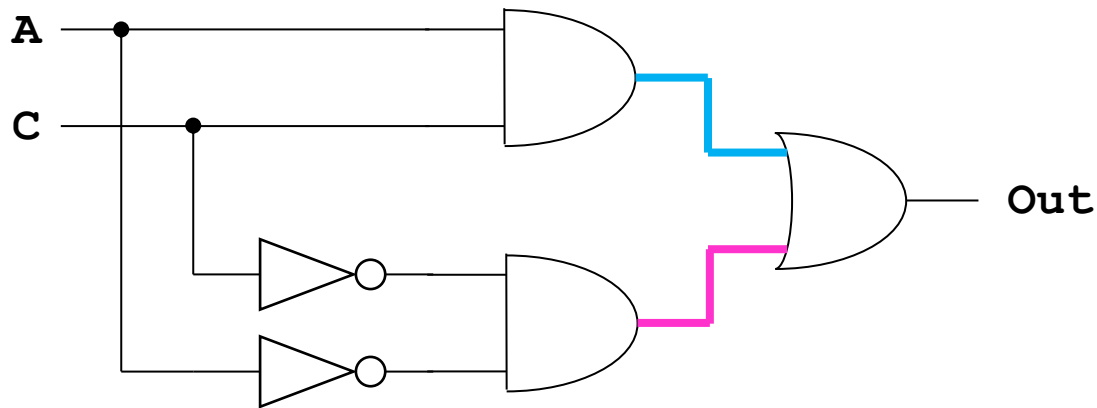
Brief Interlude: Building An Inverter



Boolean Functions, Gates and Circuits

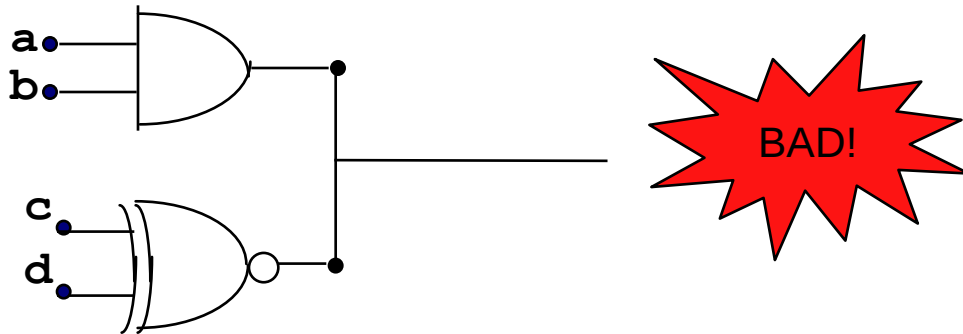
- Circuits are made from a network of gates.

$$(!A \ \& \ !C) \mid (A \ \& \ C)$$



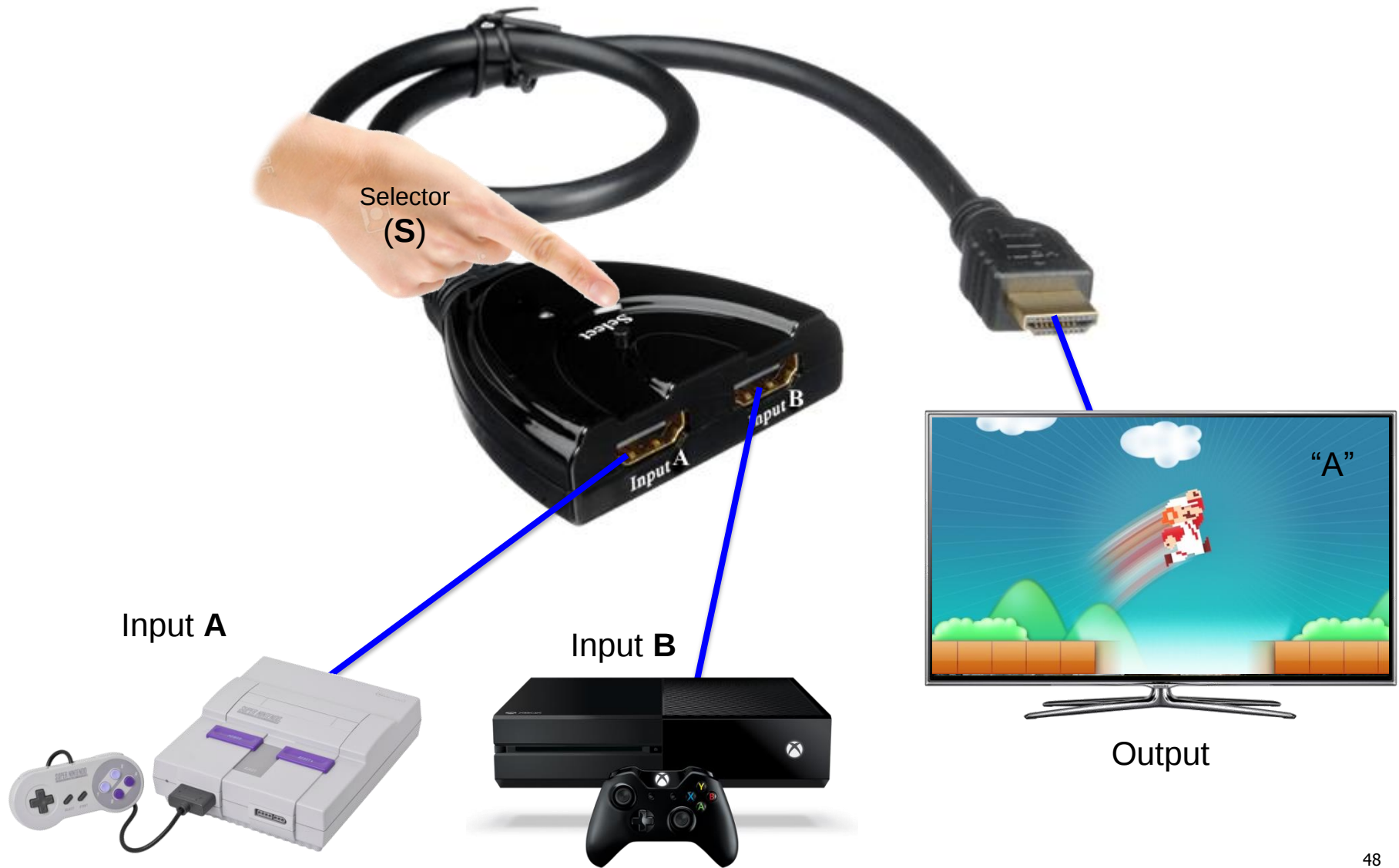
A few more words about gates

- Gates have inputs and outputs
 - If you try to hook up two outputs, bad things happen (your processor catches fire)

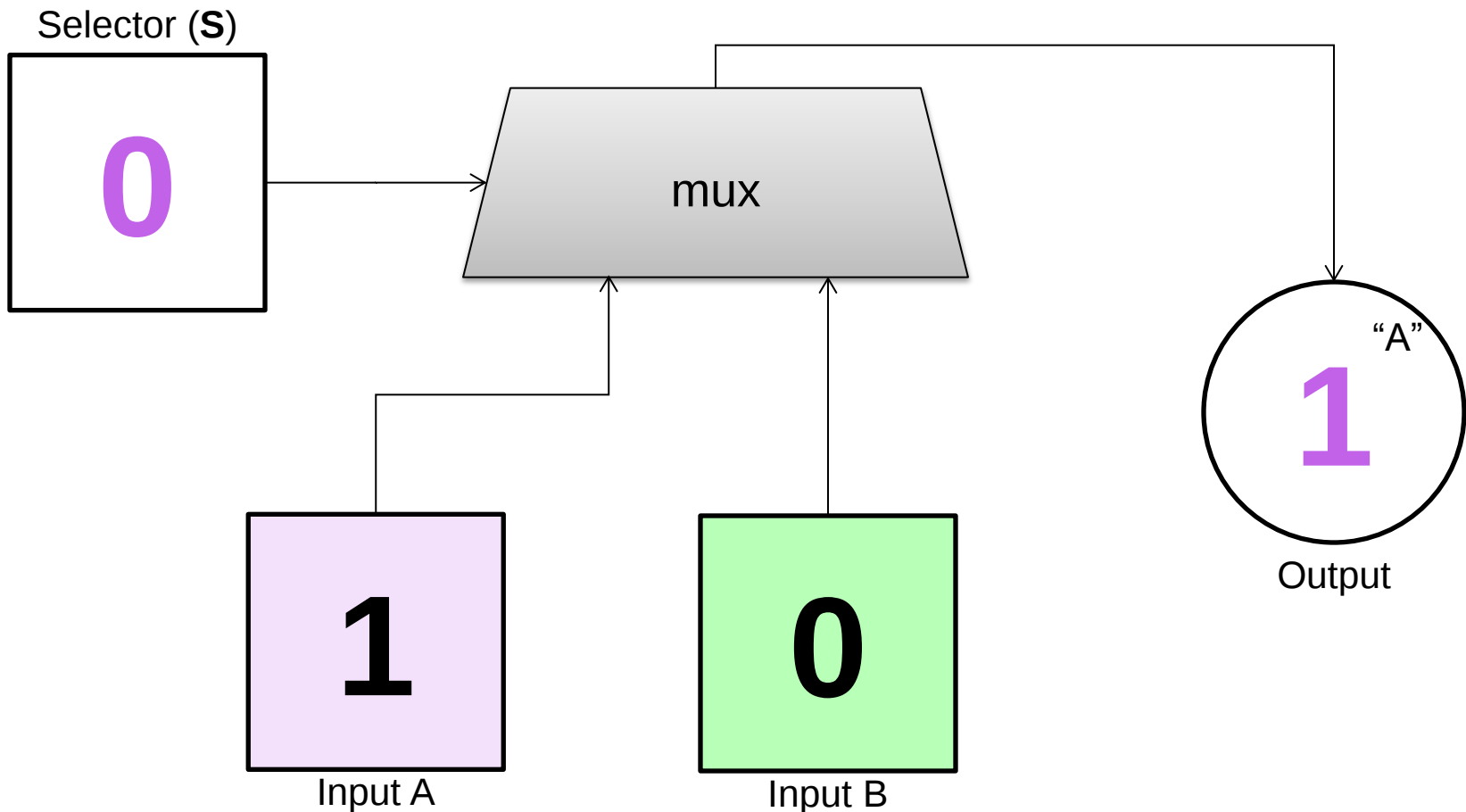


- If you don't hook up an input, it behaves kind of randomly (also not good, but not set-your-chip-on-fire bad)

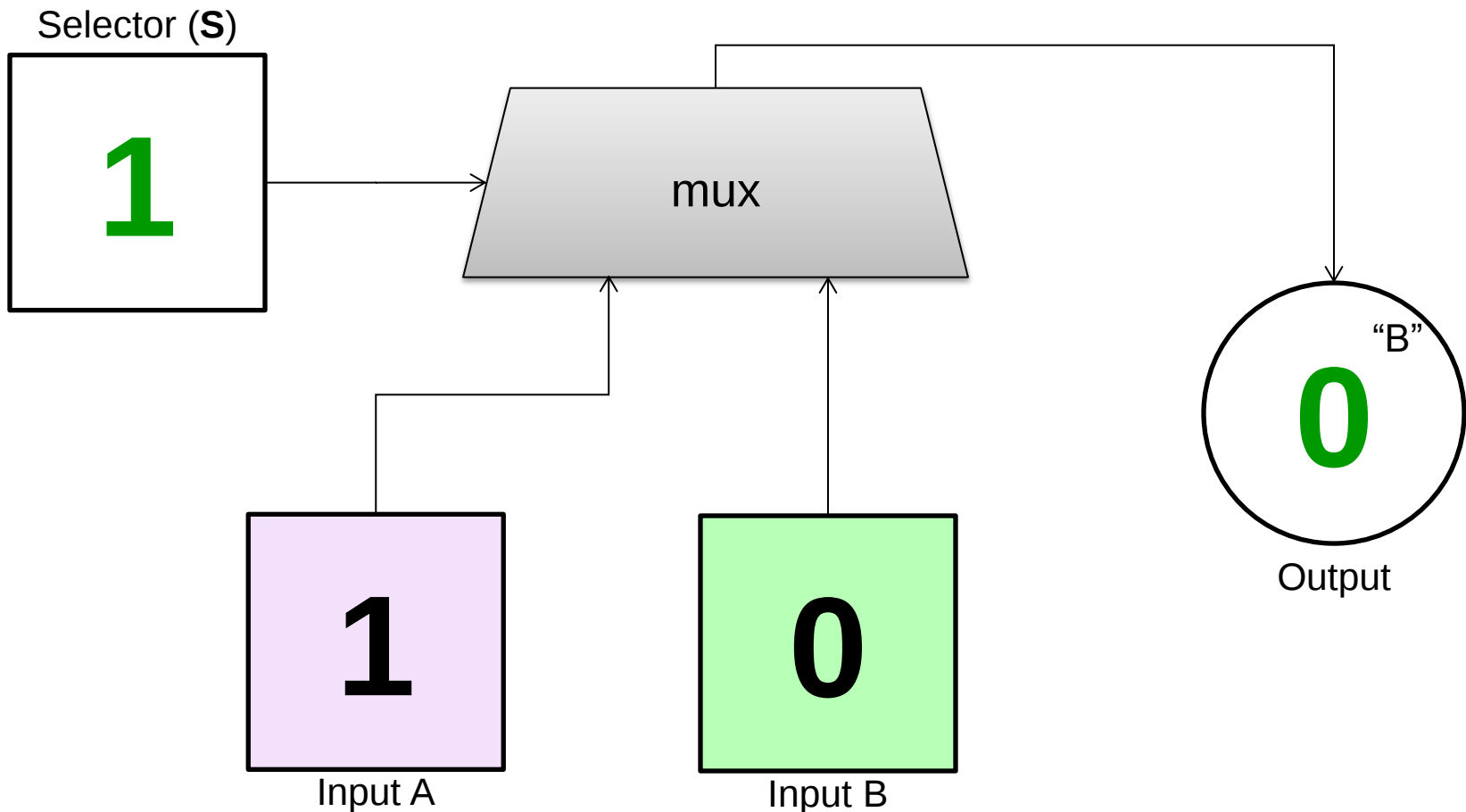
Introducing the Multiplexer (“mux”)



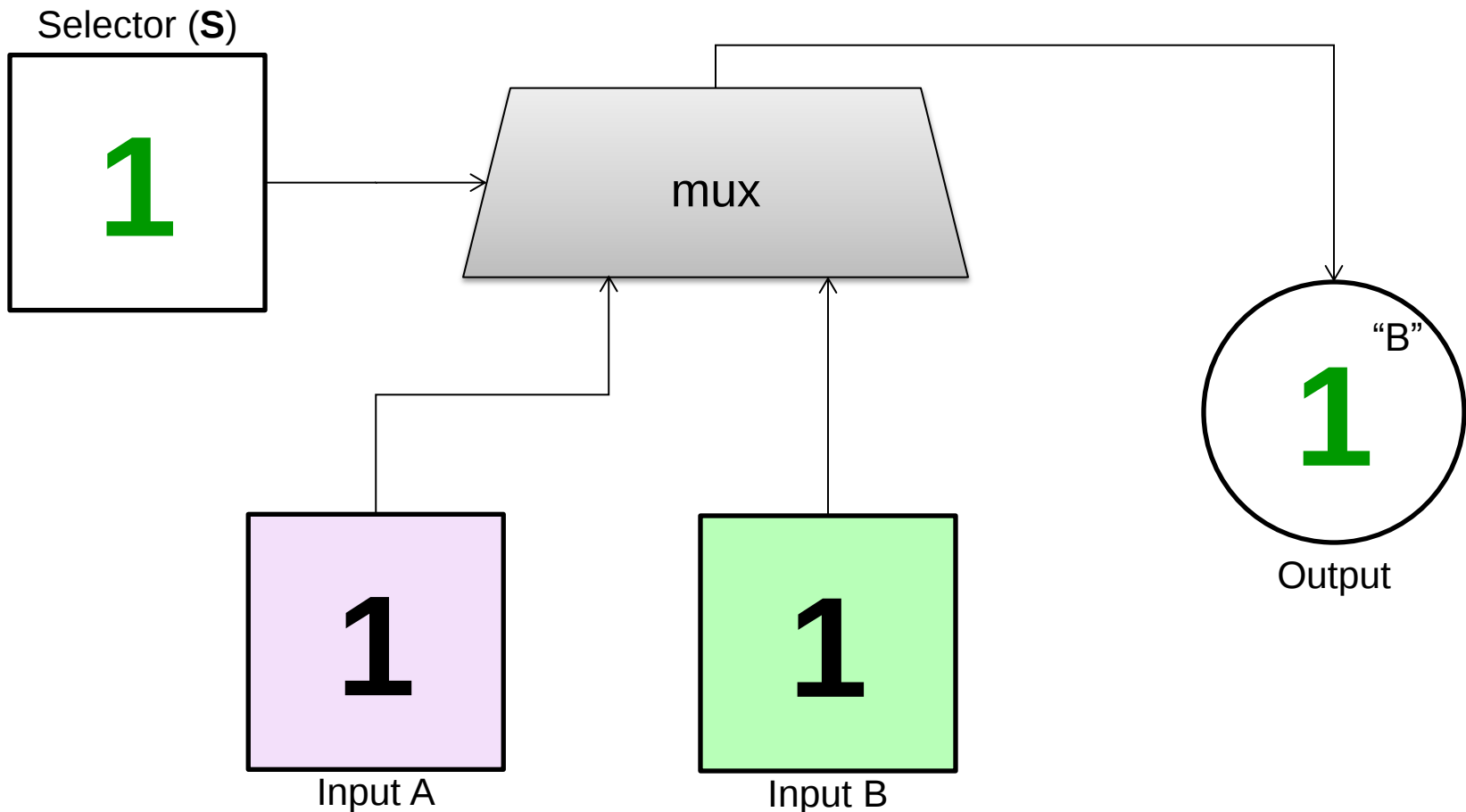
Introducing the Multiplexer (“mux”)



Introducing the Multiplexer (“mux”)



Introducing the Multiplexer (“mux”)



Let's Make a Useful Circuit

- Pick between 2 inputs (called 2-to-1 MUX)
 - Short for multiplexor

- What might we do first?

- Make a truth table?
 - S is selector:
 - S=0, pick A
 - S=1, pick B

- Next: sum-of-products

$(\neg A \ \& \ B \ \& \ S) \mid$
 $(A \ \& \ \neg B \ \& \ \neg S) \mid$
 $(A \ \& \ B \ \& \ \neg S) \mid$
 $(A \ \& \ B \ \& \ S)$

- Simplify

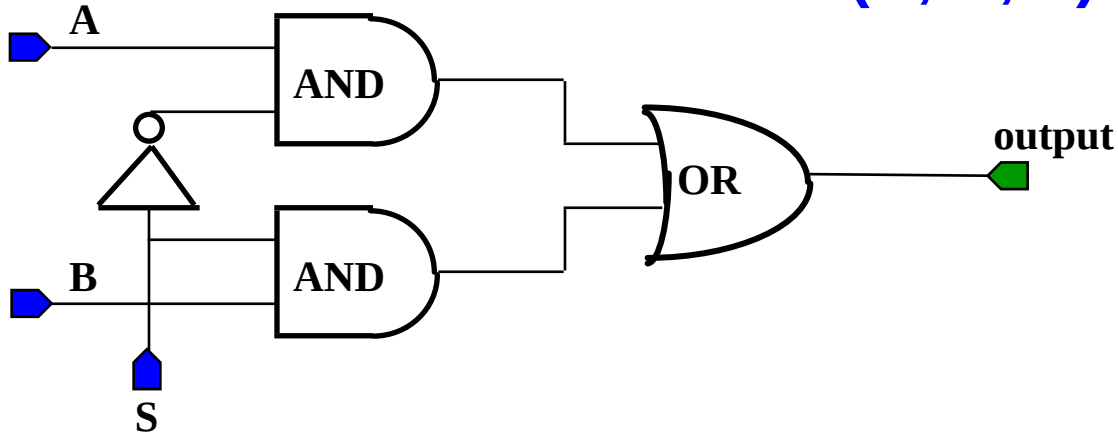
$(A \ \& \ \neg S) \mid (B \ \& \ S)$

A	B	S	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

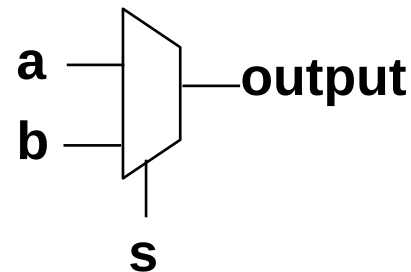
Circuit Example: 2x1 MUX

Draw it in gates:

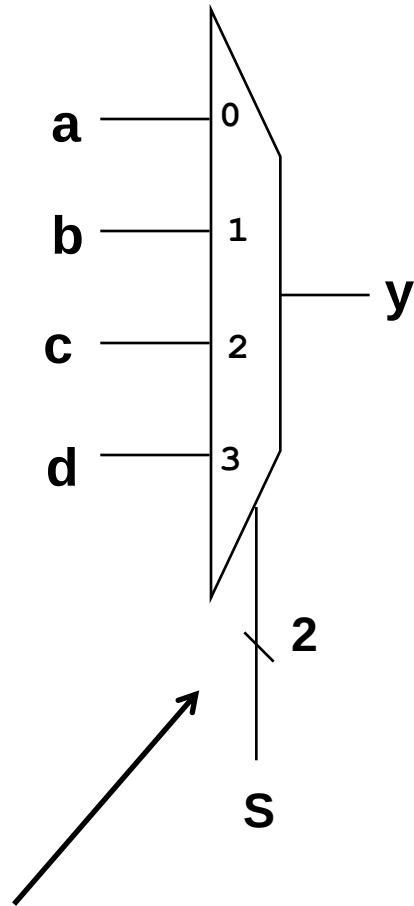
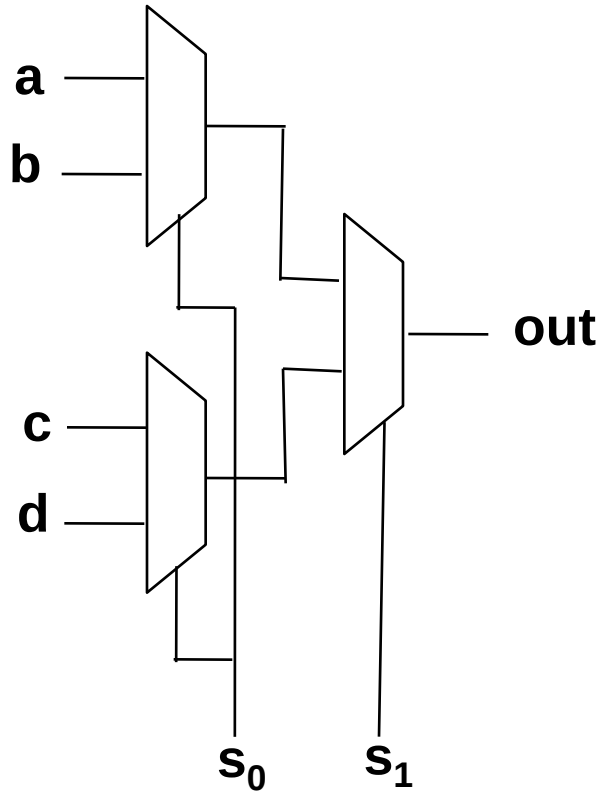
$$\text{MUX}(A, B, S) = (A \& !S) \mid (B \& S)$$



So common, we give it
its own symbol:



Example 4x1 MUX



The / 2 on the wire means “2 bits”

Arithmetic and Logical Operations in ISA

- What operations are there?
- How do we implement them?
 - Consider a 1-bit Adder

Designing a 1-bit adder

- What boolean function describes the low bit?
 - XOR
- What boolean function describes the high bit?
 - AND

$$0 + 0 = 00$$

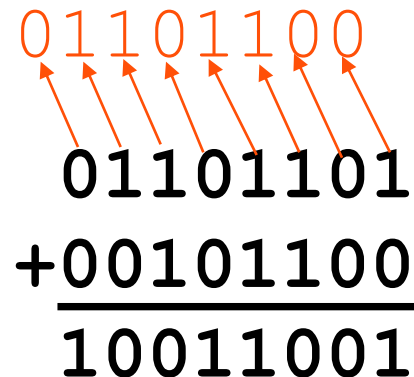
$$0 + 1 = 01$$

$$1 + 0 = 01$$

$$1 + 1 = 10$$

Designing a 1-bit adder

- Remember how we did binary addition:
 - Add the **two bits**
 - Do we have a **carry-in** for this bit?
 - Do we have to **carry-out** to the next bit?

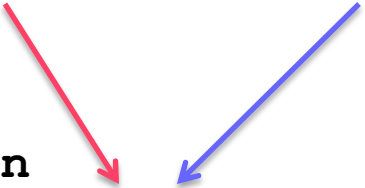


A binary addition diagram illustrating carry propagation. The first number, 01101101, is written in black. The second number, +00101100, is written in black below it, with a horizontal line underneath. The result, 10011001, is written in black below the line. Above the first number, the digits 01101100 are written in orange. Orange arrows point from each of these orange digits down to the corresponding digit of the first number (01101101), showing a carry-in of 1 for each bit position from the second number.

$$\begin{array}{r} 01101101 \\ +00101100 \\ \hline 10011001 \end{array}$$

Designing a 1-bit adder

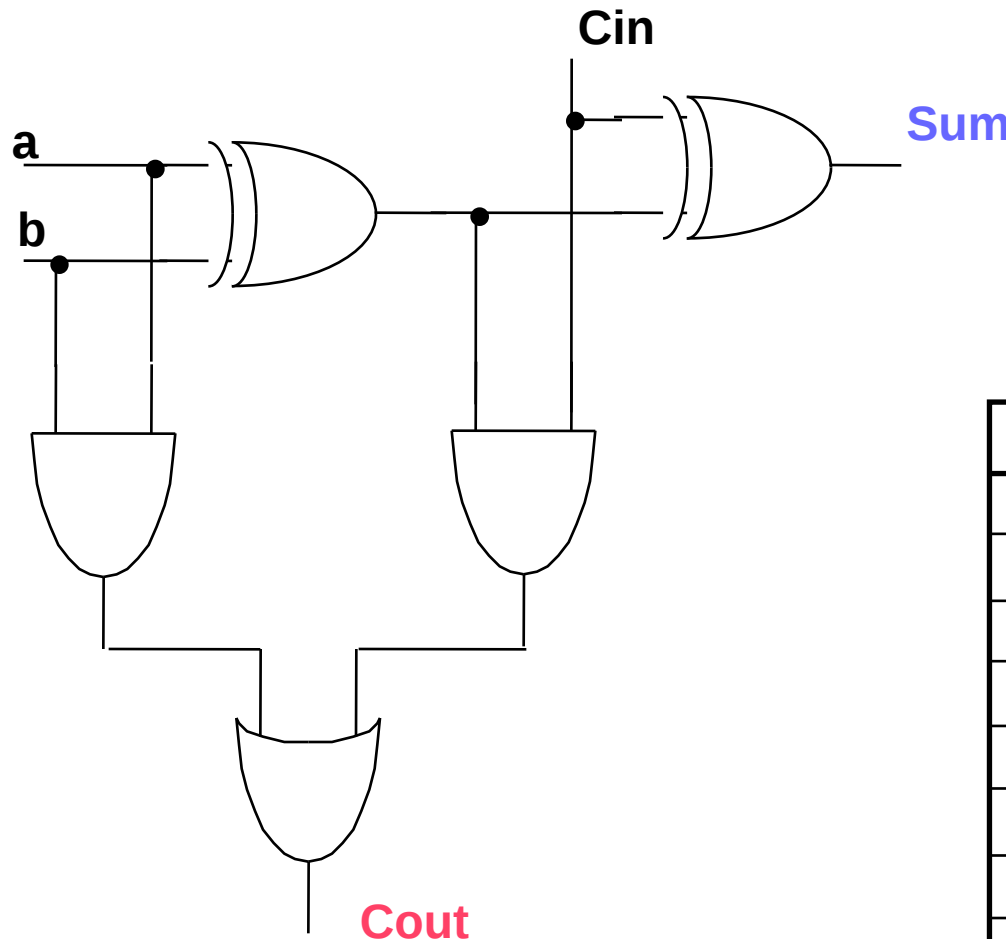
- So we'll need to add three bits (including carry-in)
- Two-bit output is the **carry-out** and the **sum**



a		b		C _{in}	=	
0	+	0	+	0	=	00
0	+	0	+	1	=	01
0	+	1	+	0	=	01
0	+	1	+	1	=	10
1	+	0	+	0	=	01
1	+	0	+	1	=	10
1	+	1	+	0	=	10
1	+	1	+	1	=	11

Turn into expression,
simplify,
circuit-ify,
yadda yadda yadda...

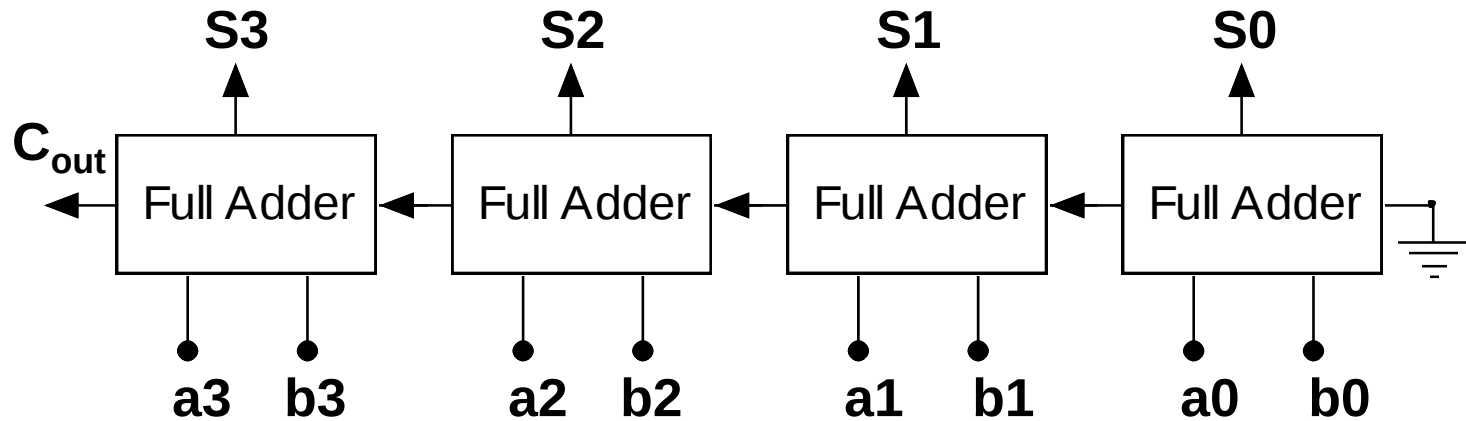
A 1-bit Full Adder



$$\begin{array}{r} 01101100 \\ 01101101 \\ +00101100 \\ \hline 10011001 \end{array}$$

a	b	C _{in}	Sum	C _{out}
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

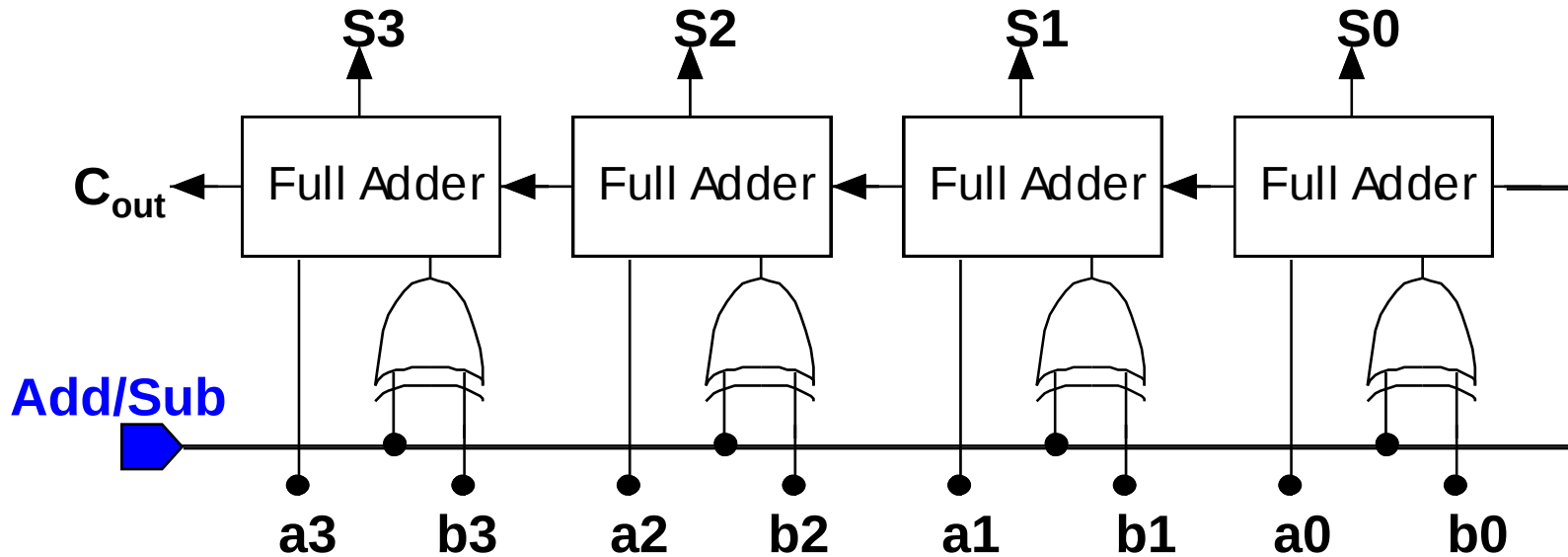
Example: 4-bit adder



Subtraction

- How do we perform integer subtraction?
- What is the hardware?
 - Recall: hardware was why 2's complement was good idea
- Remember: Subtraction is just addition
$$X - Y =$$
$$X + (-Y) =$$
$$X + (\sim Y + 1)$$

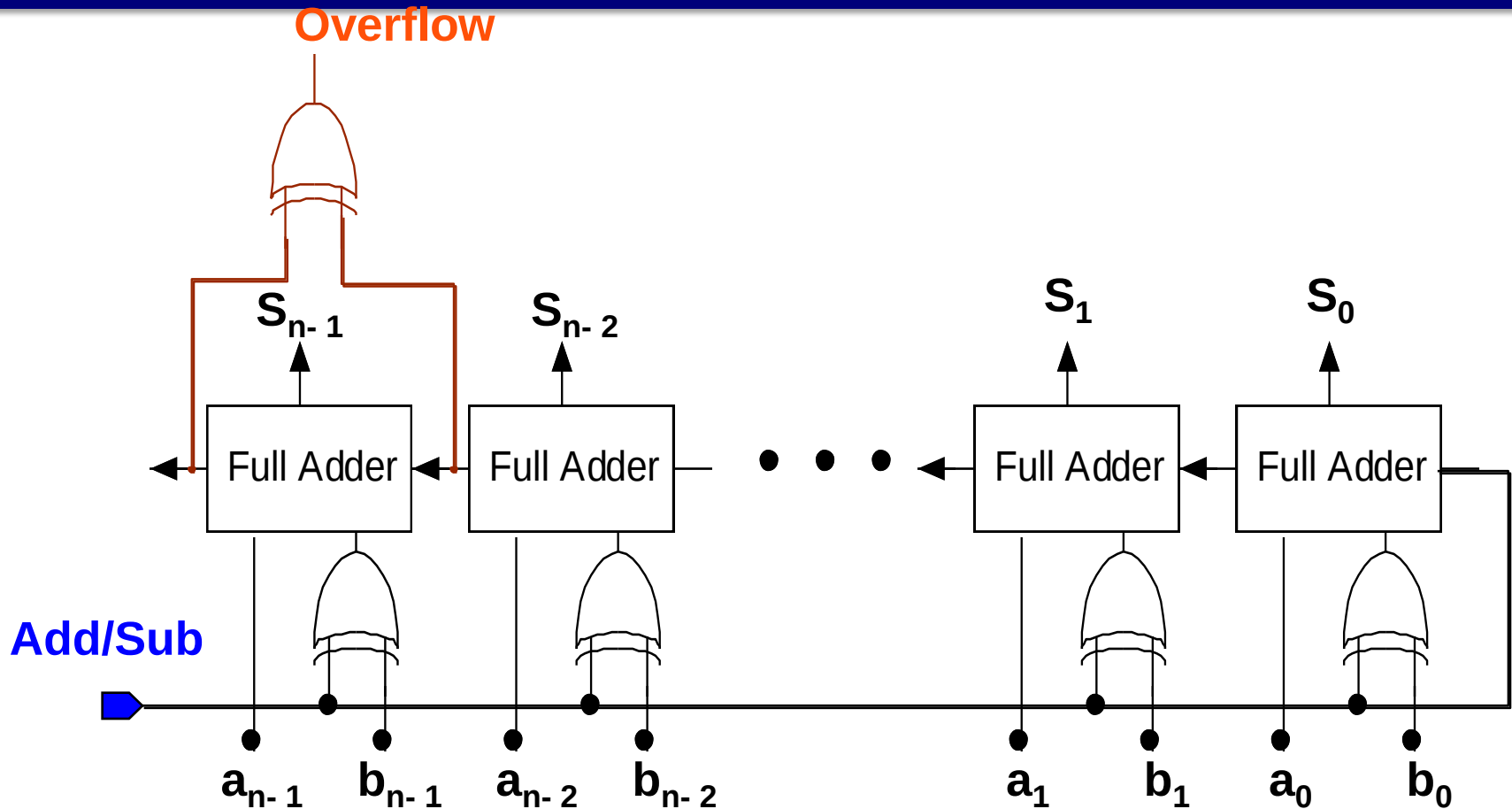
Example: Adder/Subtractor



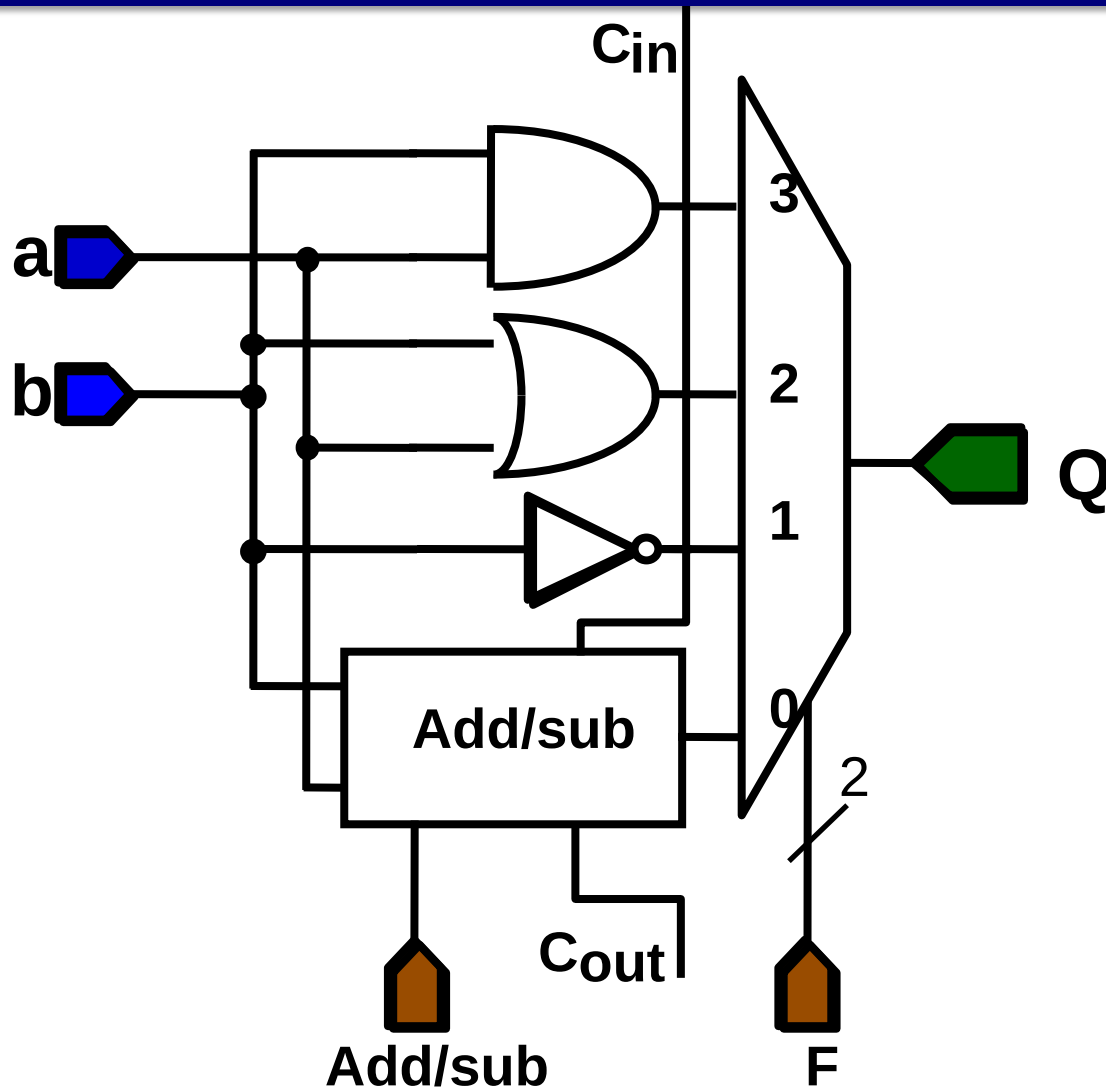
Overflow

- We can detect unsigned overflow by looking at CO
- How would we detect signed overflow?
 - If adding positive numbers and result "is" negative
 - If adding negative numbers and result "is" positive
 - At most significant bit of adder, check if $CI \neq CO$
 - Can check with XOR gate

Add/Subtract With Overflow Detection

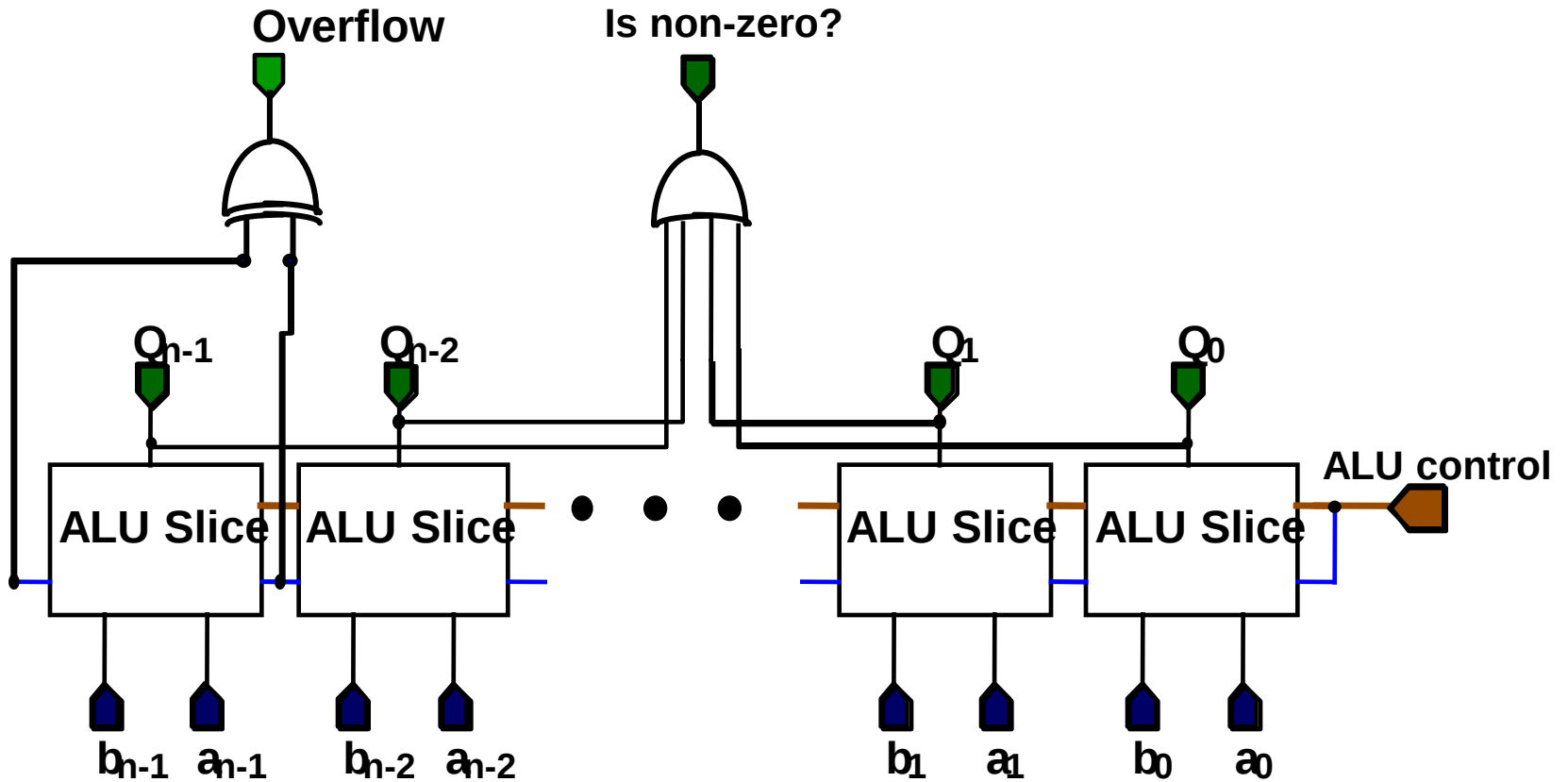


ALU Slice



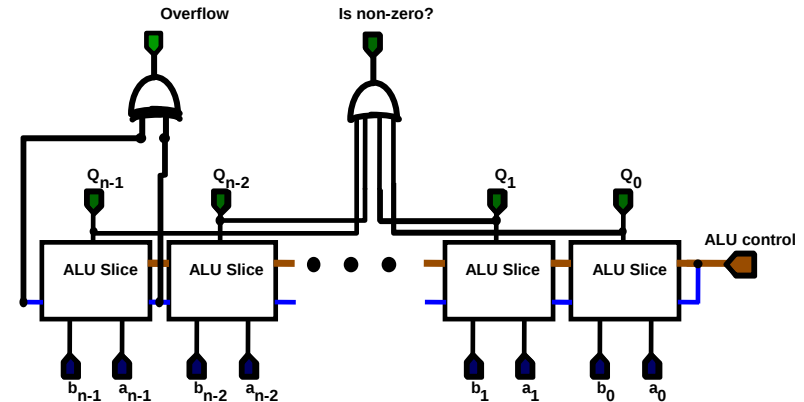
A	F	Q
0	0	$a + b$
1	0	$a - b$
-	1	NOT b
-	2	$a \text{ OR } b$
-	3	$a \text{ AND } b$

The ALU

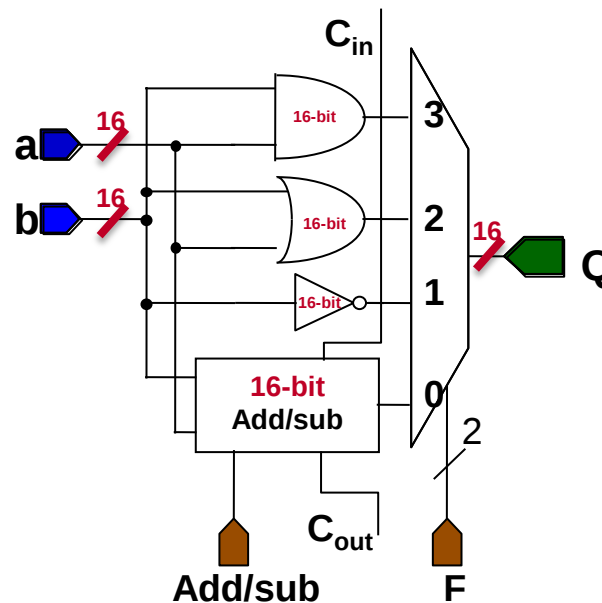


Alternate ALU design

- Previous design did ALU stuff for each bit, then chained them.

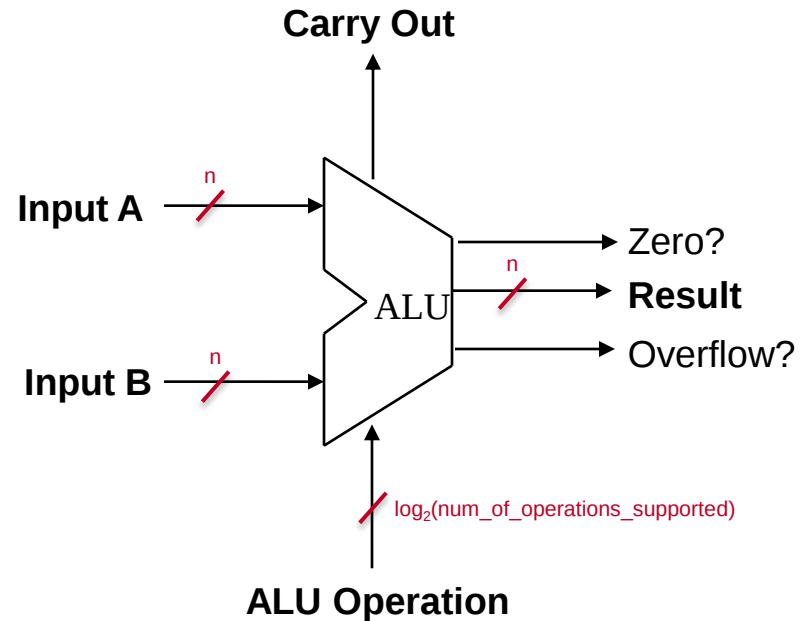


- Can also do each word-size operation and mux the resulting words.



Abstraction: The ALU

- General structure
- Two operand inputs
- Control inputs
- We can build circuits for
 - Multiplication
 - Division
 - They are more complex



Another Operations We Might Want: Shift

- Remember the << and >> operations?
 - Shift left/shift right?
 - How would we implement these?
- Suppose you have an 8-bit number
 $b_7b_6b_5b_4b_3b_2b_1b_0$
- And you can shift it left by a 3-bit number

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$
$$S_2 S_1 S_0$$

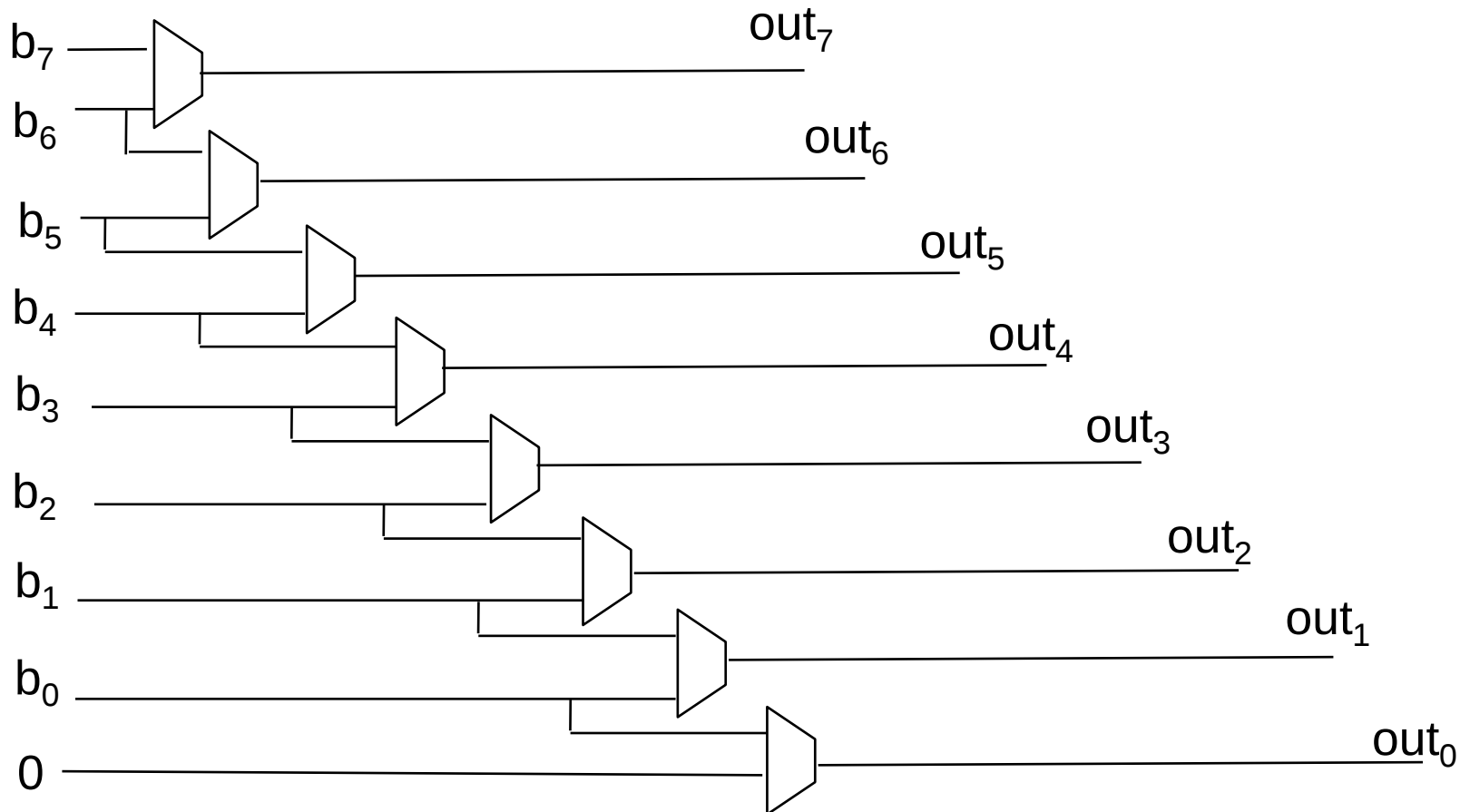
- Option 1: Truth Table?
 - $2^{11} = 2048$ rows? Yuck.

...but you can do it. Truth table gives this expression for output bit 0:

[illegible]

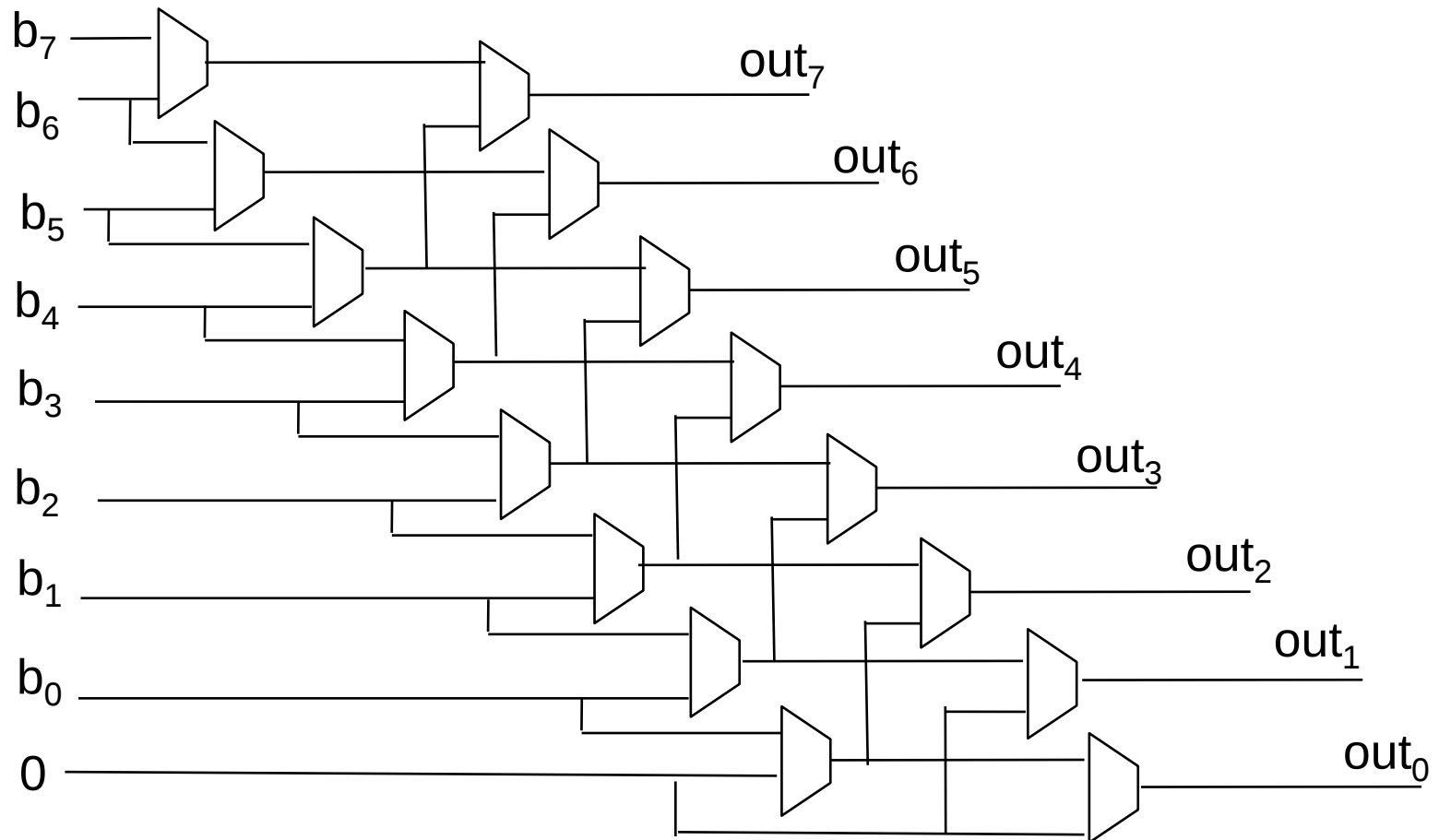
Let's simplify

- Simpler problem: 8-bit number shifted by 1 bit number (shift amount selects each mux)



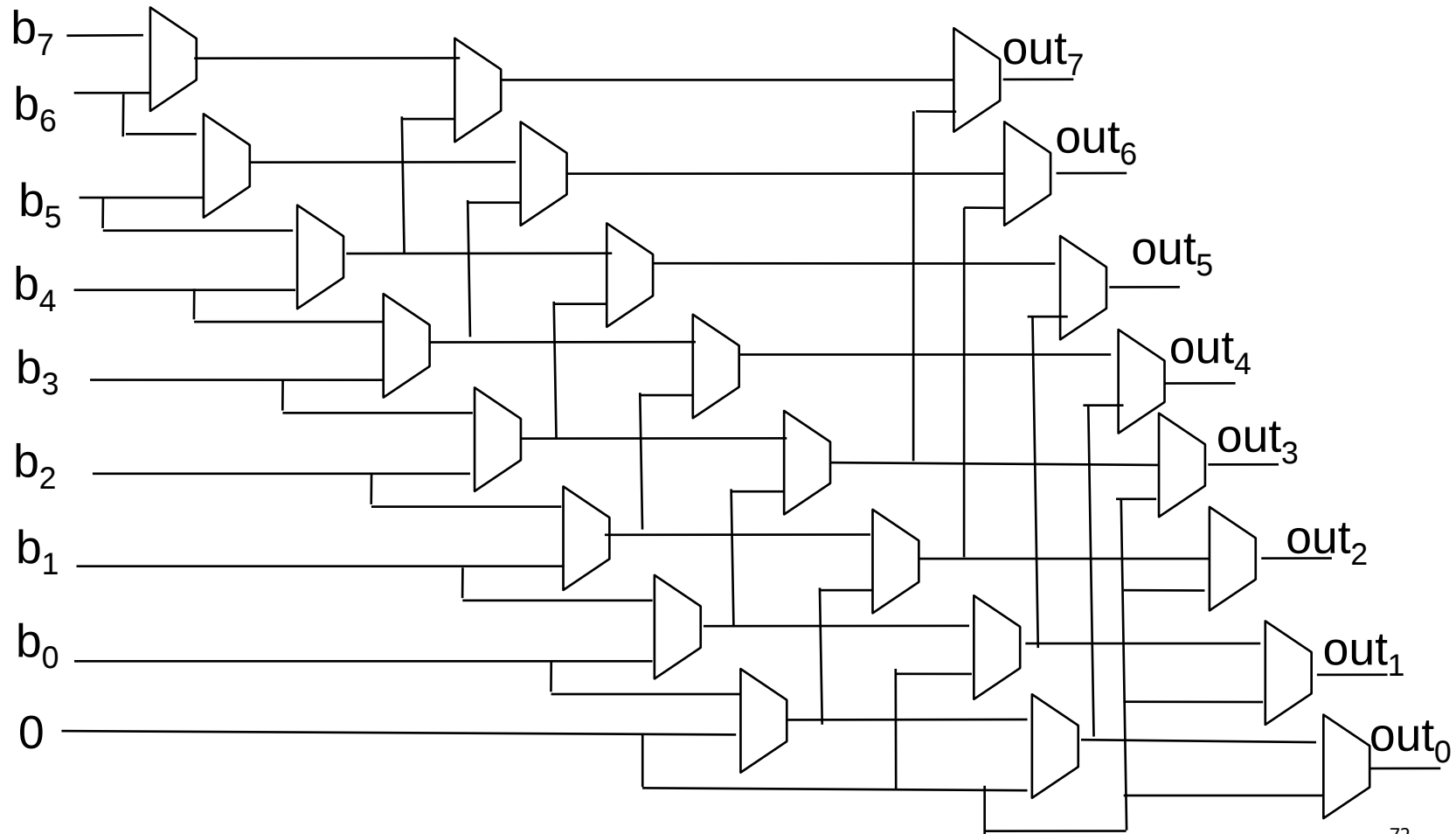
Let's simplify

- Simpler problem: 8-bit number shifted by 2 bit number



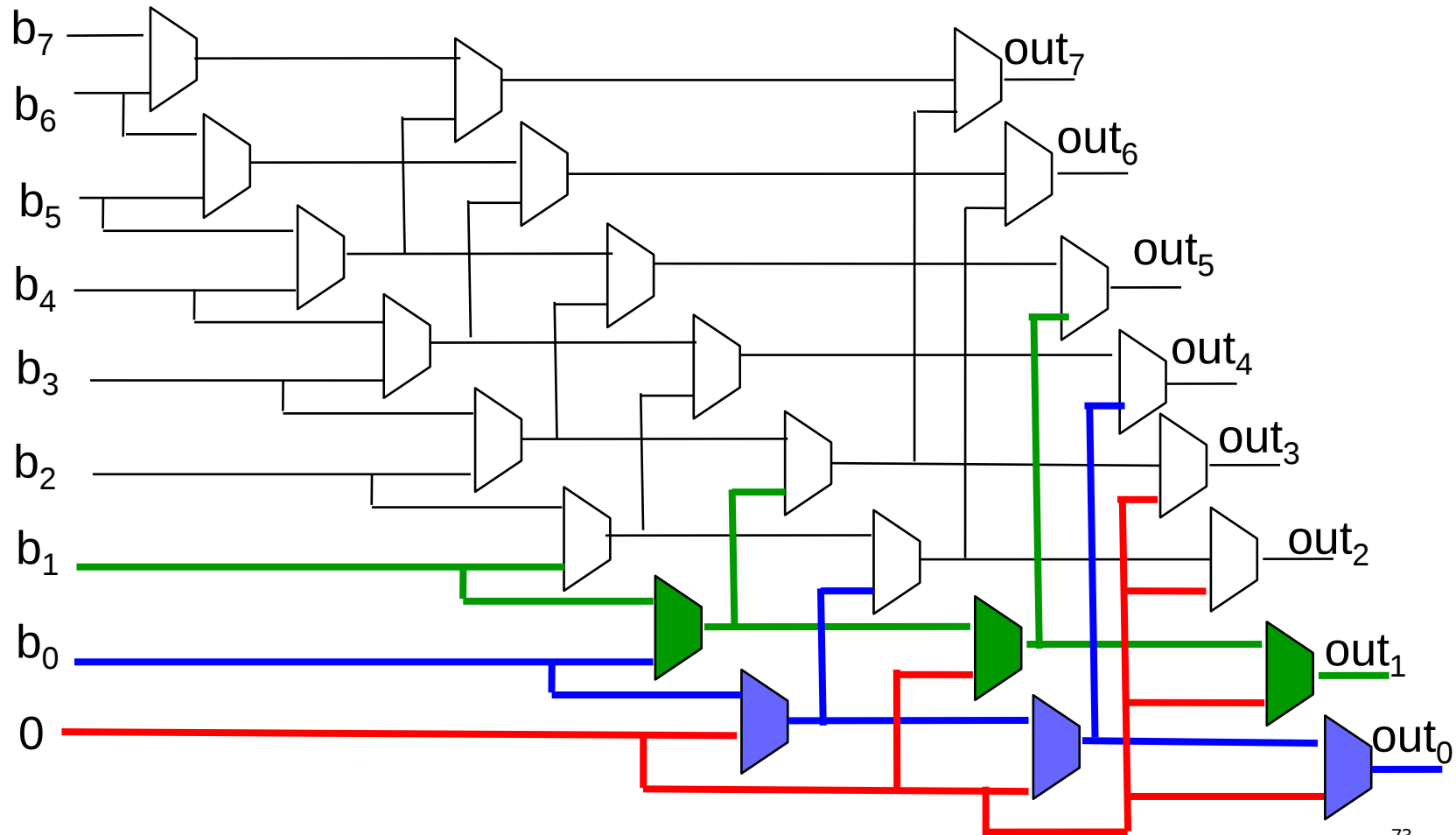
Now shifted by 3-bit number

- Full problem: 8-bit number shifted by 3 bit number



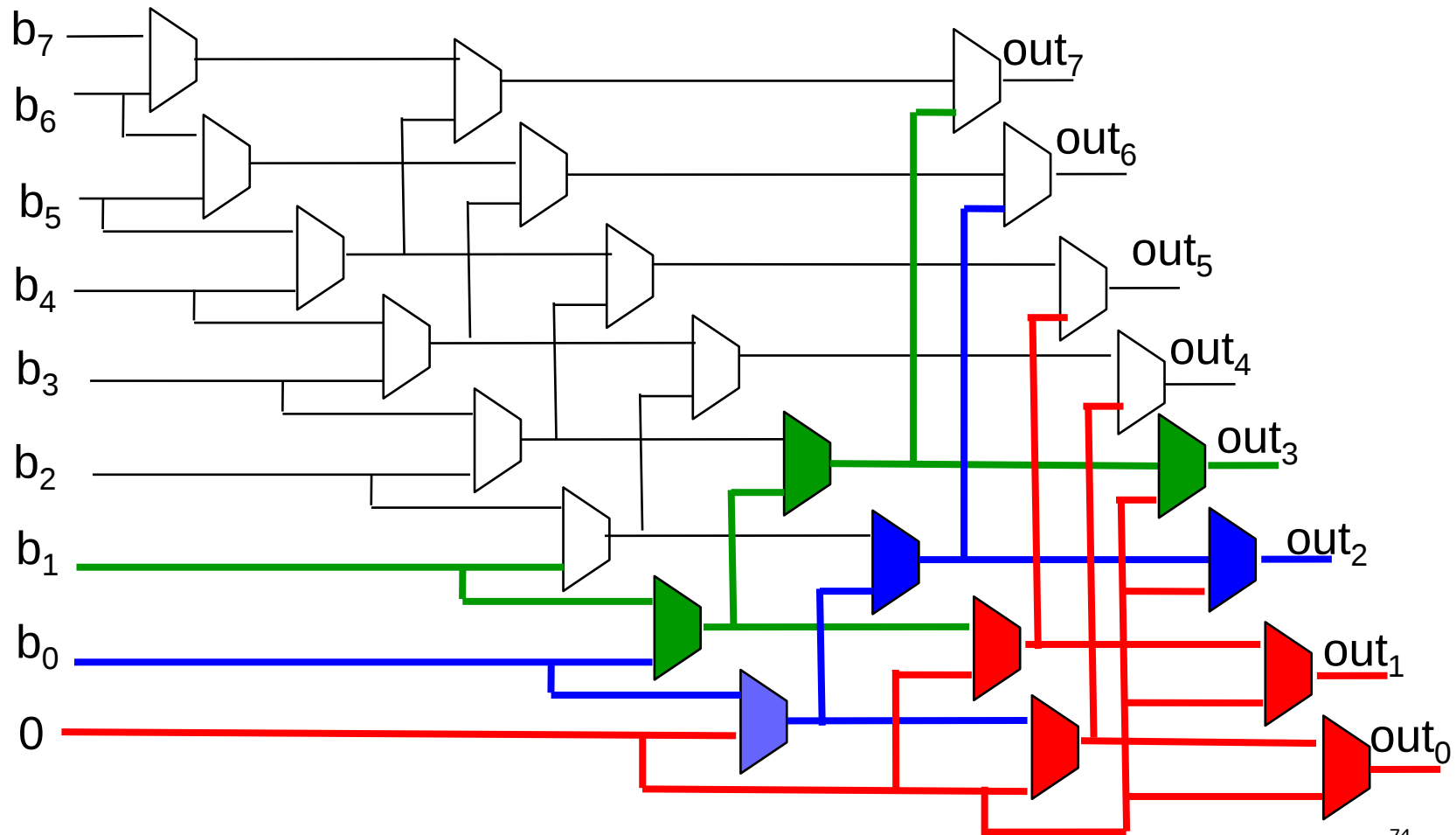
Now shifted by 3-bit number

- Shifter in action: shift by 000 (all muxes have S=0)



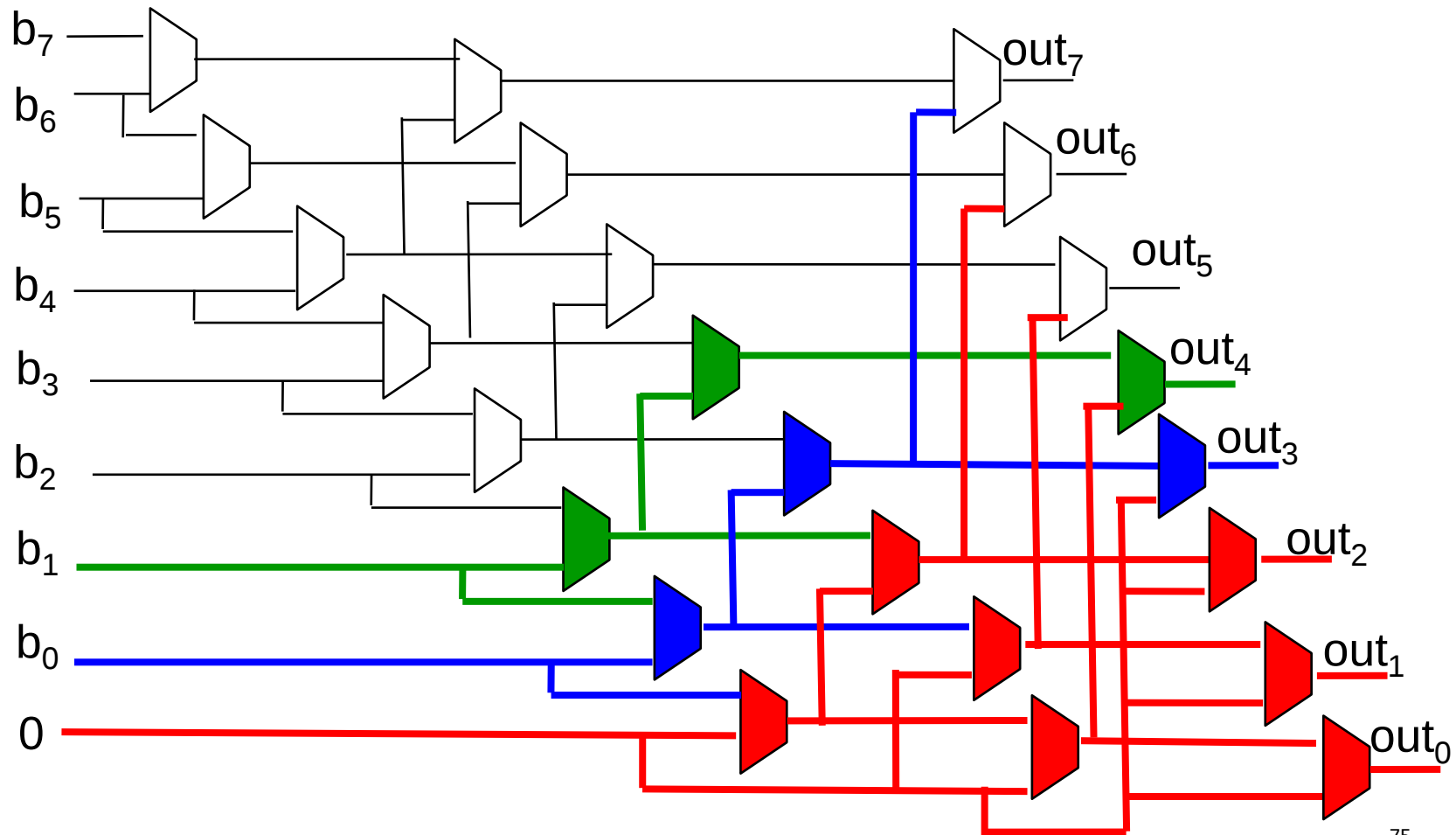
Now shifted by 3-bit number

- Shifter in action: shift by 010
 - From L to R: $S = 0, 1, 0$



Now shifted by 3-bit number

- Shifter in action: shift by 011
 - From L to R: $S = 1, 1, 0$ (reverse of shift amount)



Summary

- Boolean Algebra & functions
- Logic gates (AND, OR, NOT, etc)
- Multiplexors
- Adder
- Arithmetic Logic Unit (ALU)
- Bit shifting