

Combinational Circuits: From Boolean Algebra to Gates

Arvind

Computer Science & Artificial Intelligence Lab
M.I.T.

6.004 Outline

- Module 1: RISC-V Assembly language programming (4 lectures) ✓
- Module 2: (Digital) Logic design (8 lectures)
 - Boolean Algebra
 - Combinational circuits
 - (Clocked) Sequential circuits
 - Expressing logic designs in Bluespec
 - Logic synthesis:
 - From Bluespec to Logic circuits
 - From Logic circuits to standard gate libraries
 - Concurrency issues
- Module 3: RISC-V processors (6+1 lectures)
- Module 4: OS, I/O, Virtual memory (3 lectures)
- Module 5: Multicores (2 lectures)

Combinational circuit

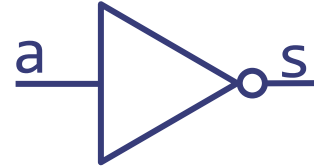


- A combinational circuit has *binary* inputs and outputs
- It represents a *pure* function
$$f: \underbrace{\text{Bool} \times \text{Bool} \times \text{Bool}}_{\text{inputs}} \rightarrow \underbrace{\text{Bool} \times \text{Bool}}_{\text{outputs}}$$
- It has no memory or state, i.e., given the same input it produces the same output

Three simple combinational circuits

- NOT

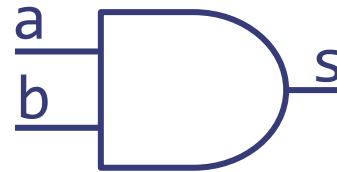
a	s
0	1
1	0



$$s = \sim a$$

- AND

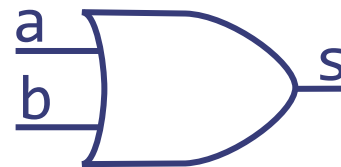
a	b	s
0	0	0
0	1	0
1	0	0
1	1	1



$$s = a \cdot b$$

- OR

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

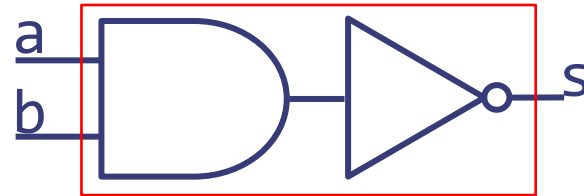


$$s = a + b$$

Some other famous gates

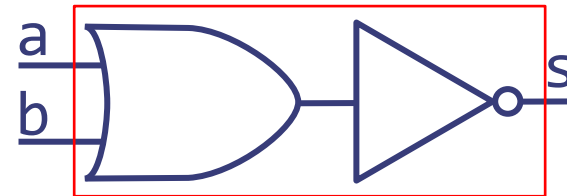
- NAND

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



- NOR

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

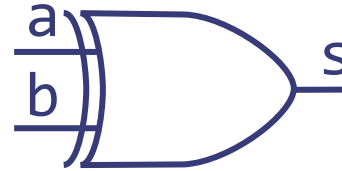


The functionality of these gates can be expressed using NOT, AND, and OR gates

Exclusive OR (XOR): another famous gate

- XOR

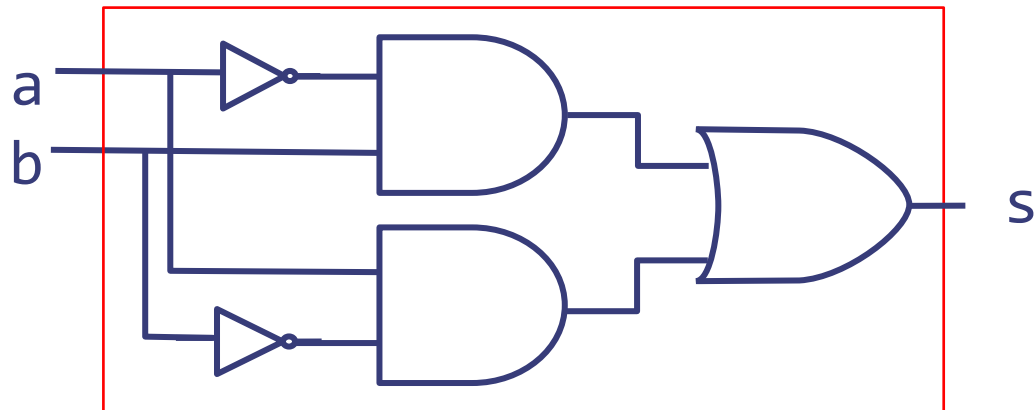
a	b	s
0	0	0
0	1	1
1	0	1
1	1	0



$$s = a \oplus b$$

- From the Truth Table XOR produces a 1 when either (a=0) AND (b=1) or (a=1) AND (b=0). Hence, $a \oplus b = \sim a \cdot b + a \cdot \sim b$

You can express XOR using NOT, AND, and OR gates

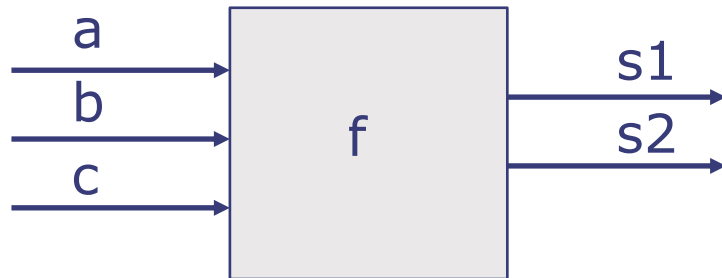


Nomenclature

- We use the words in each of the following categories interchangeably
 - *combinational circuits, Boolean expressions, Boolean circuits*
 - *gate, Boolean operator*
- We use variables to name wires in a combinational circuit

Truth Table

- The functionality of any combinational circuit can be defined using a Truth Table
- It lists all possible combinations of values for inputs and specifies the output for each input



a	b	c	s1	s2
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	0
1	1	0	0	1
1	1	1	0	0

What is the size of the truth table for an n input and m output function?

2^n m -bit rows

Truth Tables are not suitable for describing functions with large number of inputs

From Truth Table to Boolean Expression

- There is a Boolean expression corresponding to every truth table
 - Write a *product term* using only AND and NOT corresponding to each row with a 1
 - The final Boolean expression is the *sum* of all such product terms

$$\begin{aligned}s1 &= (\sim a) \cdot (\sim b) \cdot (\sim c) \\ &\quad + a \cdot (\sim b) \cdot (\sim c) \\ &\quad + a \cdot (\sim b) \cdot c \\ s2 &= (\sim a) \cdot b \cdot c \\ &\quad + a \cdot (\sim b) \cdot (\sim c) \\ &\quad + a \cdot b \cdot (\sim c)\end{aligned}$$

a	b	c	s1	s2
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	0
1	1	0	0	1
1	1	1	0	0

sum-of-products (SOP) representation

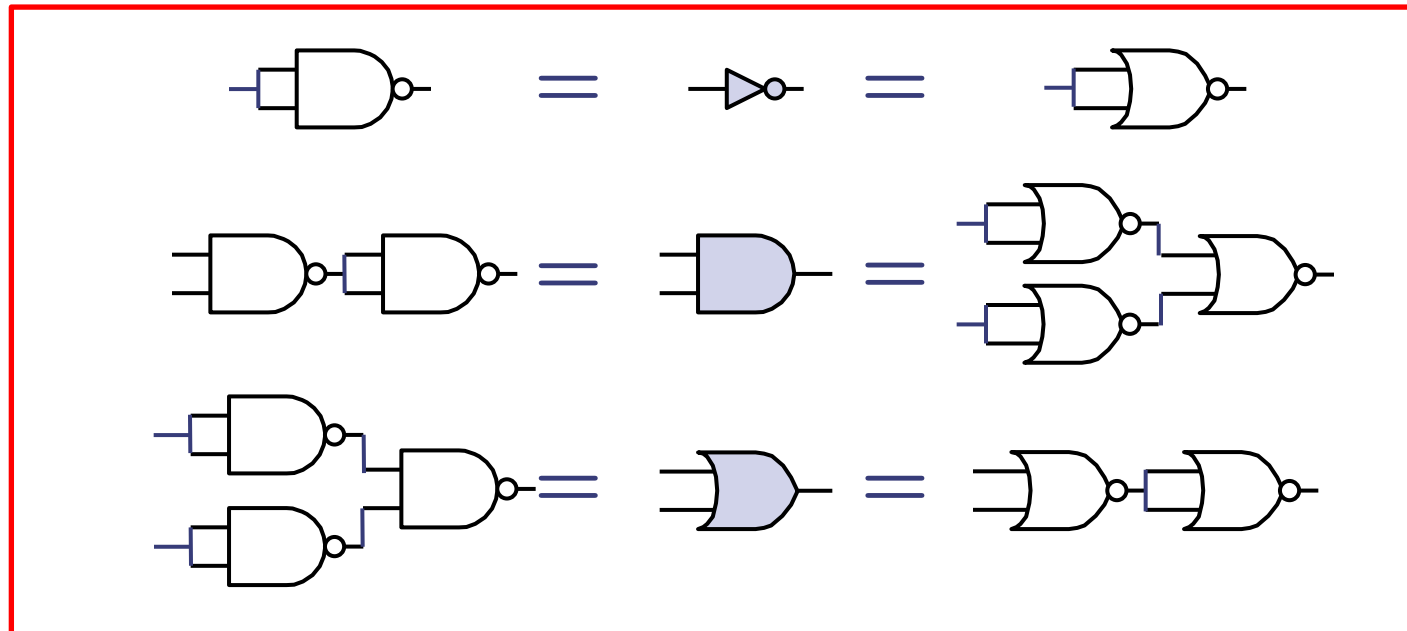
Universal set of gates

- A set of gates is considered *universal* if any Boolean function can be described using only those gates. Examples:

- {AND, OR, NOT}
- {NAND}
- {NOR}

} Pictorial proof

Proof: Any Truth Table can be represented as a Sum of Product



Binary Addition

- Addition in base 2 is performed just like in base 10

Base 10

$$\begin{array}{r} 1 \text{ — carry} \\ 14 \\ + 7 \\ \hline 21 \end{array}$$

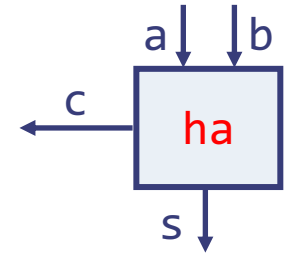
Base 2

$$\begin{array}{r} 1110 \\ 1110 \\ + 111 \\ \hline 10101 \end{array}$$

Let us build a hardware adder

Half Adder

- Half adder (HA): adds two 1-bit numbers and produces a sum and a carry bit



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

Boolean equations

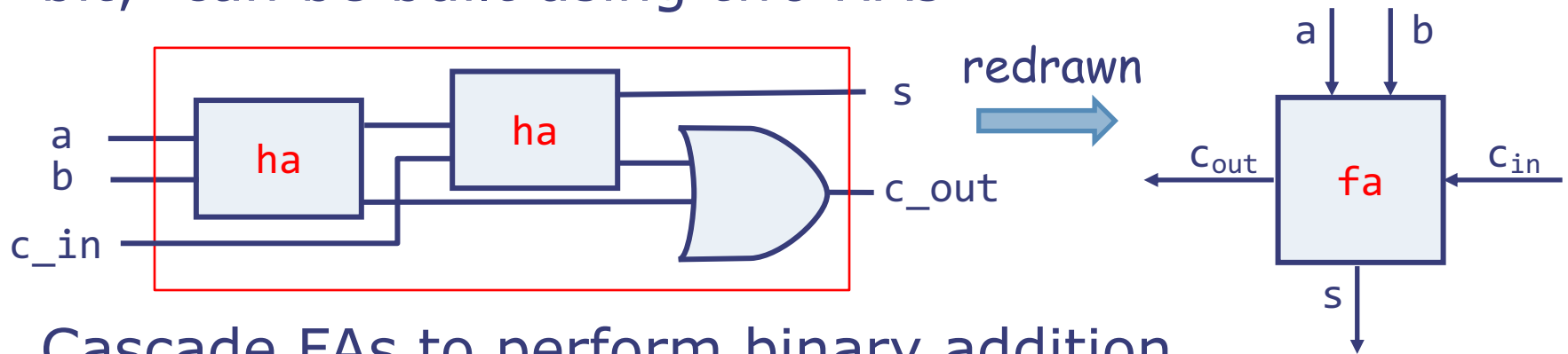
$$s = \sim a \cdot b + a \cdot (\sim b)$$

$$= a \oplus b$$

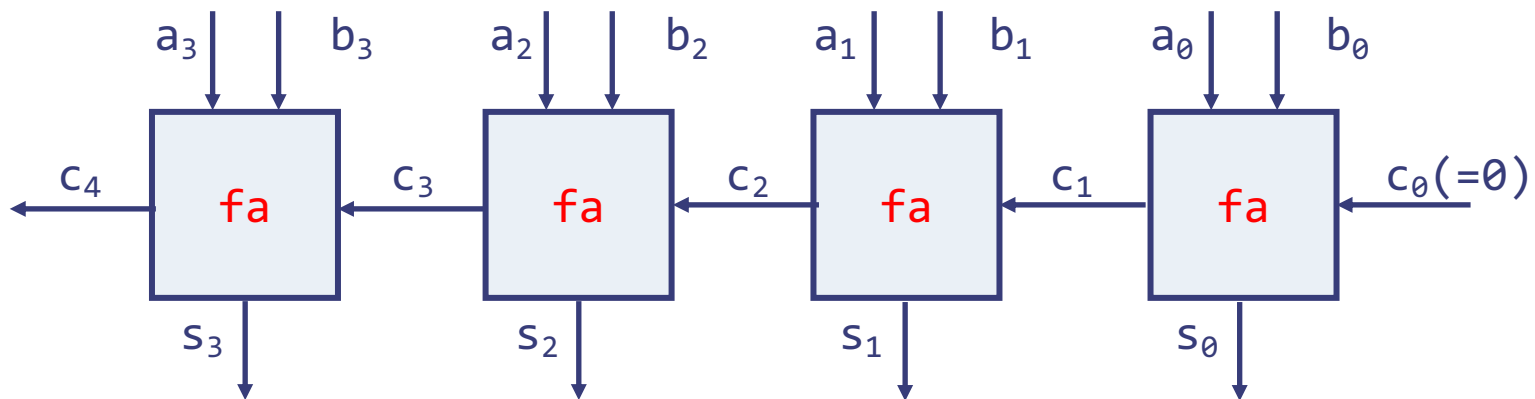
$$c = a \cdot b$$

Combinational Logic for an adder

- Full adder (FA): adds two one-bit numbers and a carry-in bit, and produces a sum bit and a carry-out bit; can be built using two HAs

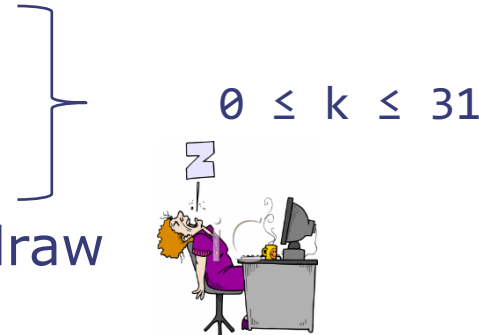


- Cascade FAs to perform binary addition



Describing a 32-bit adder alternatives

- Truth Table with 2^{64} rows and 33 columns (sum(32)+carry(1)) !
- 32 sets of Boolean equations, where each set describes a FA
- Some notation to describe recurrences
 - $t_k = a_k \oplus b_k$
 - $s_k = t_k \oplus c_k$
 - $c_{k+1} = a_k \cdot b_k + c_k \cdot t_k$
- Circuit diagrams - tedious to draw



Such representations are too verbose and not useful when we want computers to simulate the behavior of the circuit, i.e., determine the output given an input

We will use a programming language called Bluespec System Verilog (BSV) to express all circuits

Half Adder in Bluespec

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

```
function ha(a, b);  
  s = a ^ b;  
  c = a & b;  
  return {c,s};  
endfunction
```

XOR

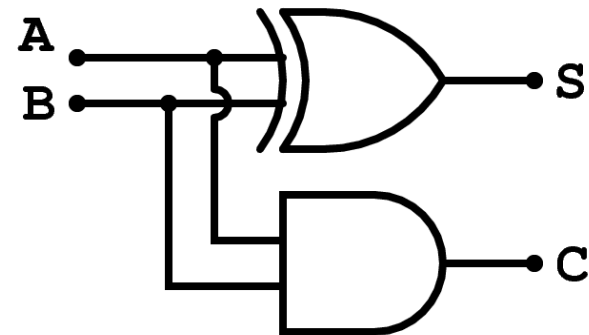
AND

Boolean equations

$$s = \sim a \cdot b + a \cdot (\sim b)$$




$$= a \oplus b$$

$$c = a \cdot b$$



Not quite correct -
needs type
annotations

Half Adder *corrected*

```
function Bit#(2) ha(Bit#(1) a, Bit#(1) b);  
  Bit#(1) s;  a type declaration that says  
  Bit#(1) c;      s is one bit wide  
  Bit#(2) result;  says that result is a two  
  s = a ^ b;      bit vector  
  c = a & b;  
  result[0] = s;  says that the zeroth bit of  
  result[1] = c;  result is the same as s  
  return result;  
endfunction
```


More convenient syntax

```
function Bit#(2) ha(Bit#(1) a, Bit#(1) b);  
  Bit#(1) s = a ^ b;  
  Bit#(1) c = a & b;  
  return {c,s};  
endfunction
```

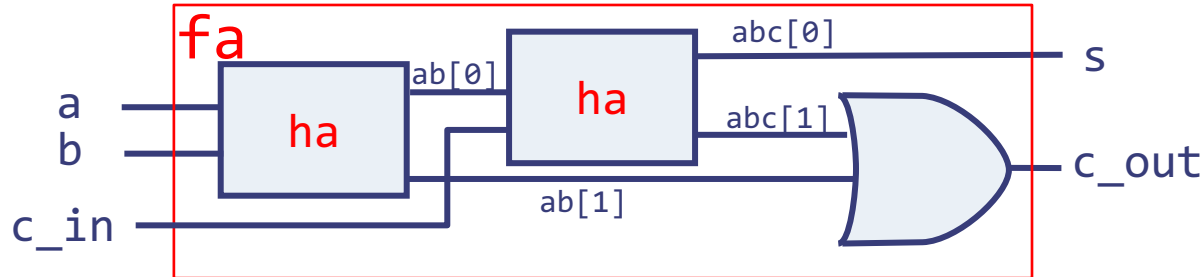
a type of a variable can be declared at same place where we define it
{...} can be used to define a bit vector

- How big is {c,s}?

2 bits

- Using {c,s} notation avoids the need to name intermediate results

Full Adder using HAs



```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
    Bit#(2) ab = ha(a, b);
    Bit#(2) abc = ha(ab[0], c_in);
    Bit#(1) c_out = ab[1] | abc[1];
    return {c_out, abc[0]};
endfunction
```

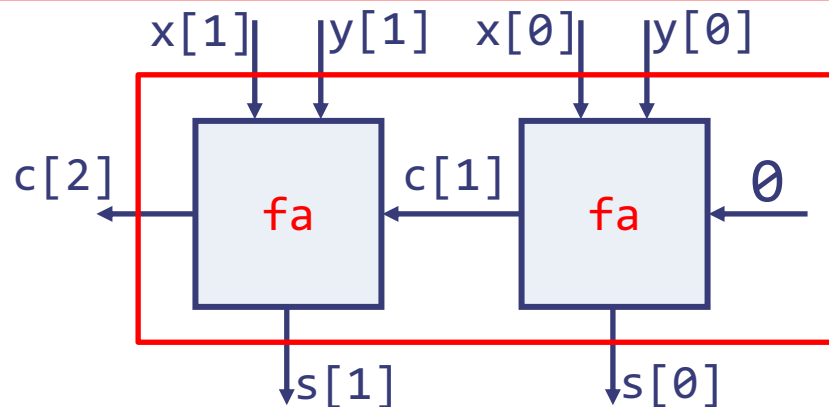
Extracts the sum bit

Extracts the carry bit

`ha` is being used as a black-box;
`fa` code is simply a wiring diagram

2-bit Ripple-Carry Adder

cascading full adders



Use **fa** as a black-box

```
function Bit#(3) add2(Bit#(2) x, Bit#(2) y);  
  Bit#(2) s = 2b'00;      Bit#(3) c = 3b'000;  
  c[0] = 0;  
  Bit#(2) cs0 = fa(x[0], y[0], c[0]);  
  s[0] = cs0[0]; c[1] = cs0[1];  
  Bit#(2) cs1 = fa(x[1], y[1], c[1]);  
  s[1] = cs1[0]; c[2] = cs1[1];  
  return {c[2], s};  
endfunction
```

The same as writing
`{c[2], s[1], s[0]}`;

s has two wires,
initially each s
wire is zero

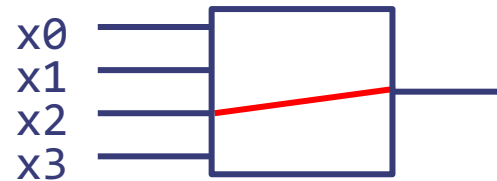
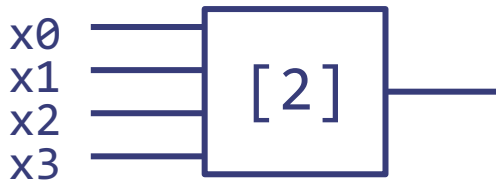
wire s[0] is
updated
wire s[1] is
updated

Selectors and Multiplexers

Selecting a wire: $x[i]$

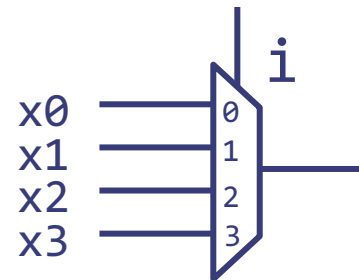
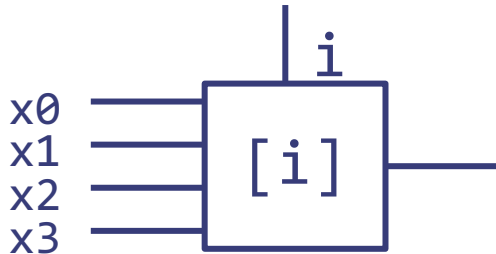
assume x is 4 bits wide

- Constant selector: e.g., $x[2]$



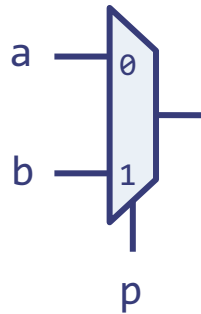
no hardware;
 $x[2]$ is just
the name of
a wire

- Dynamic selector: $x[i]$



4-way mux

A 2-way multiplexer



A mux is a simple conditional expression

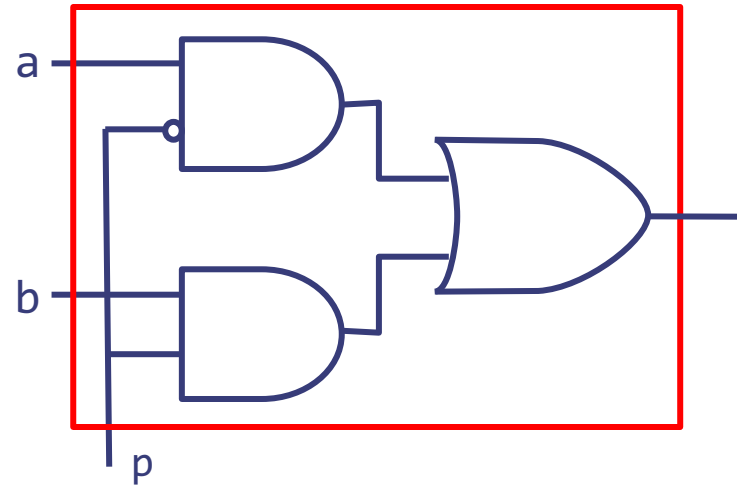
Bluespec

```
(p)? b : a ;
```

Python

```
return b if p else a
```

True is treated as a 1
and False as a 0



Gate-level implementation

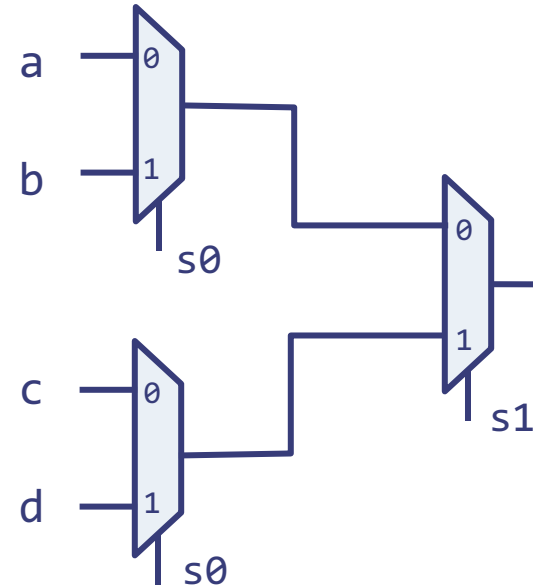
If a and b are n -bit wide then this structure is replicated n times; p is the same input for all the replicated structures

A 4-way multiplexer

syntax:
writing
0,1,2,3
would have
also worked

```
case ({s1,s0})  
  2'b00 : a;  
  2'b01 : b;  
  2'b10 : c;  
  2'b11 : d;  
endcase
```

```
def mux(a, b, c, d, s):  
  if s == 0:  
    return a  
  elif s == 1:  
    return b  
  elif s == 2:  
    return c  
  else:  
    return d
```



$(s1==0) \ \& \ (s0==1)$,
which is the same
writing $\sim s1 \ \& \ s0$

n-way mux can be
implemented using n-1
two-way muxes

Next lecture

- More simple-combinational circuits
- Logic synthesis and circuit optimization

Take Home

1. Provide a truth table for a full adder (fa)
2. Specify a Sum of Products (SOP) representation for each of the full adder outputs.