Combinational Logic and Introduction to Minispec

Combinational Logic and Introduction to Minispec

Reminders:

Quiz 1 review tonight 7:30-9:30pm

Quiz 1 on Thursday 7:30-9:30pm

Lecture Goals

- Learn how to design large combinational circuits through three useful examples:
  - Adder
  - Multiplexers
  - Shifter

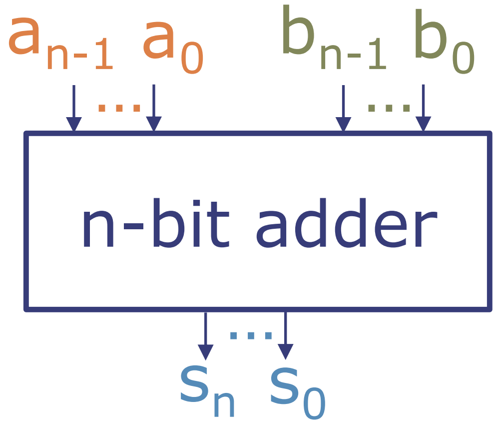
Lecture Goals

- Learn how to design large combinational circuits through three useful examples:
  - Adder
  - Multiplexers
  - Shifter

- Learn how to implement combinational circuits in the Minispec hardware description language (HDL)
  - Design each combinational circuit as a function, which can be simulated or synthesized into gates

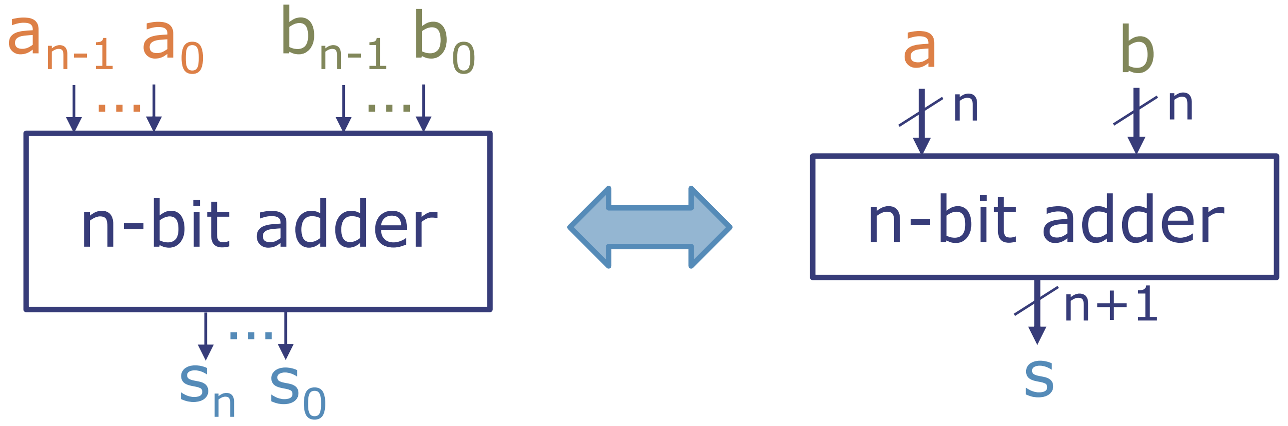
Building a Combinational Adder

- Goal: Build a circuit that takes two n-bit inputs a and b and produces (n+1)-bit output s=a+b



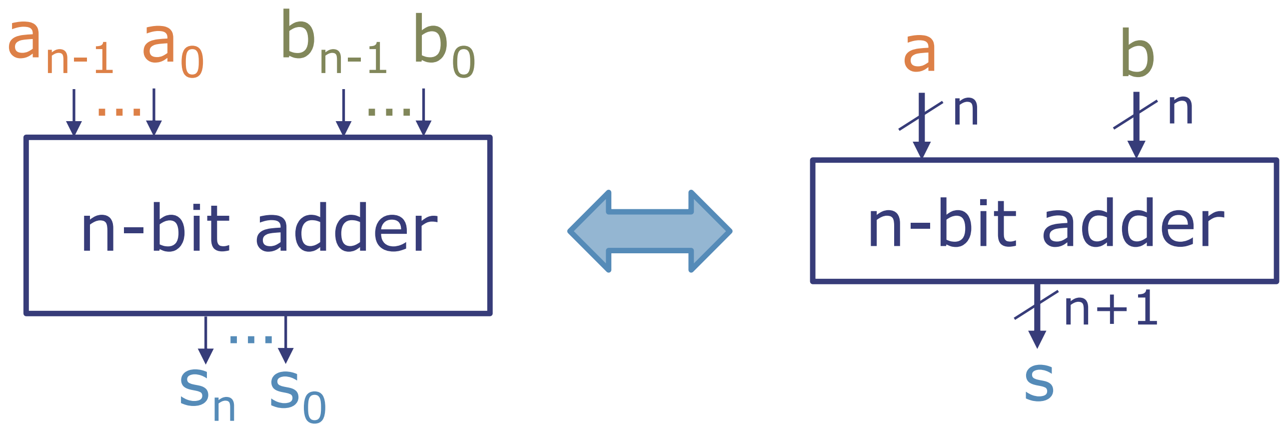
Building a Combinational Adder

- Goal: Build a circuit that takes two n-bit inputs a and b and produces (n+1)-bit output s=a+b



Building a Combinational Adder

- Goal: Build a circuit that takes two n-bit inputs a and b and produces (n+1)-bit output s=a+b



- Approach: Implement the binary addition algorithm we have seen (called the standard algorithm)

```
 1110   carry
  1110
+ 0111
 10101
```

Formalizing the Standard Algorithm

carry

```
   1110
    1110
+   0111
  ------
  10101
```

$$
\begin{array}{r}
c_4c_3c_2c_1 0 \\
a_3a_2a_1a_0 \\
+ \quad b_3b_2b_1b_0 \\
\hline
s_4s_3s_2s_1s_0
\end{array}
$$

Formalizing the Standard Algorithm

carry	1110	$\Rightarrow$	$c_4c_3c_2c_10$
	1110		$a_3a_2a_1a_0$
+	0111		$+ \quad b_3b_2b_1b_0$
	10101		$s_4s_3s_2s_1s_0$

- The i^{th} step of each addition

Formalizing the Standard Algorithm

```
carry    1110                 c4c3c2c1 0
          1110                    a3a2a1a0
     +    0111           +        b3b2b1b0
     ---------           -----------------
         10101                s4s3s2s1s0
```

- The i^th step of each addition
  - Takes three 1-bit inputs: a_i, b_i, c_i (carry-in)

Formalizing the Standard Algorithm

$$
\begin{array}{lr}
\text{carry} & 1110 \\
 & 1110 \\
+ & 0111 \\
\hline
 & 10101
\end{array}
\qquad \Rightarrow \qquad
\begin{array}{lr}
 & c_4c_3c_2c_1 0 \\
 & a_3a_2a_1a_0 \\
+ & b_3b_2b_1b_0 \\
\hline
 & s_4s_3s_2s_1s_0
\end{array}
$$

- The i^{th} step of each addition
  - Takes three 1-bit inputs: a_i, b_i, c_i (carry-in)
  - Produces two 1-bit outputs: s_i, c_{i+1} (carry-out)
  - The 2-bit output $c_{i+1}s_i$ is the binary sum of the three inputs

Formalizing the Standard Algorithm

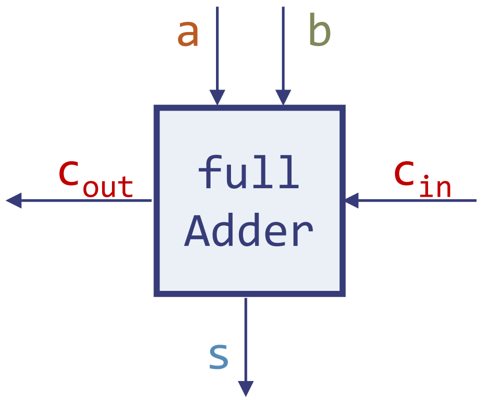
$$
\begin{array}{rr}
\text{carry} & 1110 \\
 & 1110 \\
+ & 0111 \\
\hline
 & 10101
\end{array}
\quad\Rightarrow\quad
\begin{array}{rr}
 & c_4 c_3 c_2 c_1 0 \\
 & a_3 a_2 a_1 a_0 \\
+ & b_3 b_2 b_1 b_0 \\
\hline
 & s_4 s_3 s_2 s_1 s_0
\end{array}
$$

- The i^{th} step of each addition
  - Takes three 1-bit inputs: a_i, b_i, c_i (carry-in)
  - Produces two 1-bit outputs: s_i, c_{i+1} (carry-out)
  - The 2-bit output $c_{i+1}s_i$ is the binary sum of the three inputs

Can you build a circuit that performs a single step with what you've learned so far?

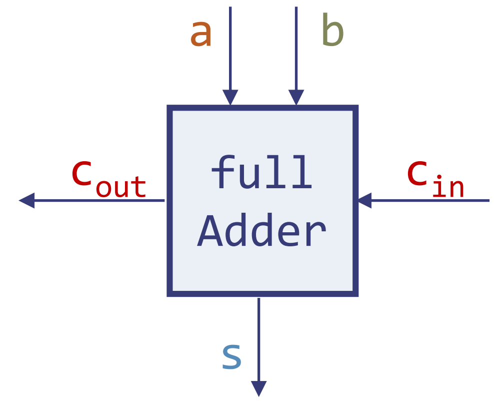
Combinational Logic for an Adder

- First, build a full adder (FA), which
  - Adds three one-bit numbers: a, b, and carry-in
  - Produces a sum bit and a carry-out bit

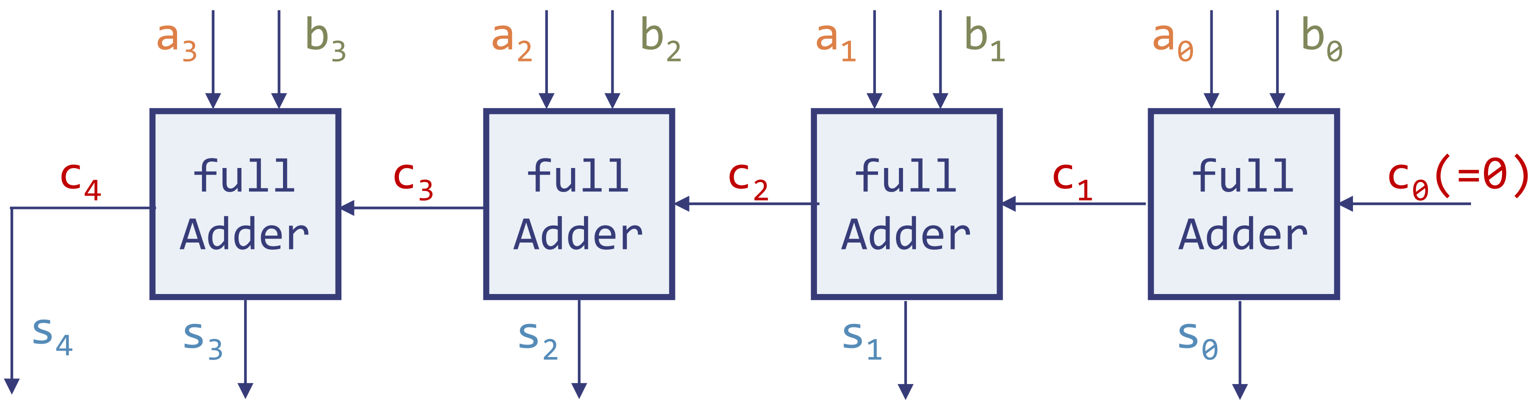


Combinational Logic for an Adder

- First, build a full adder (FA), which
  - Adds three one-bit numbers: a, b, and carry-in
  - Produces a sum bit and a carry-out bit

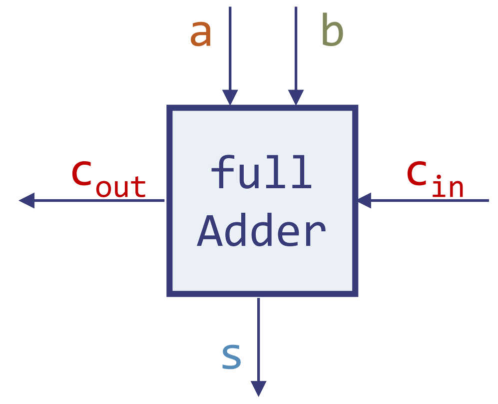


- Then, cascade FAs to perform binary addition

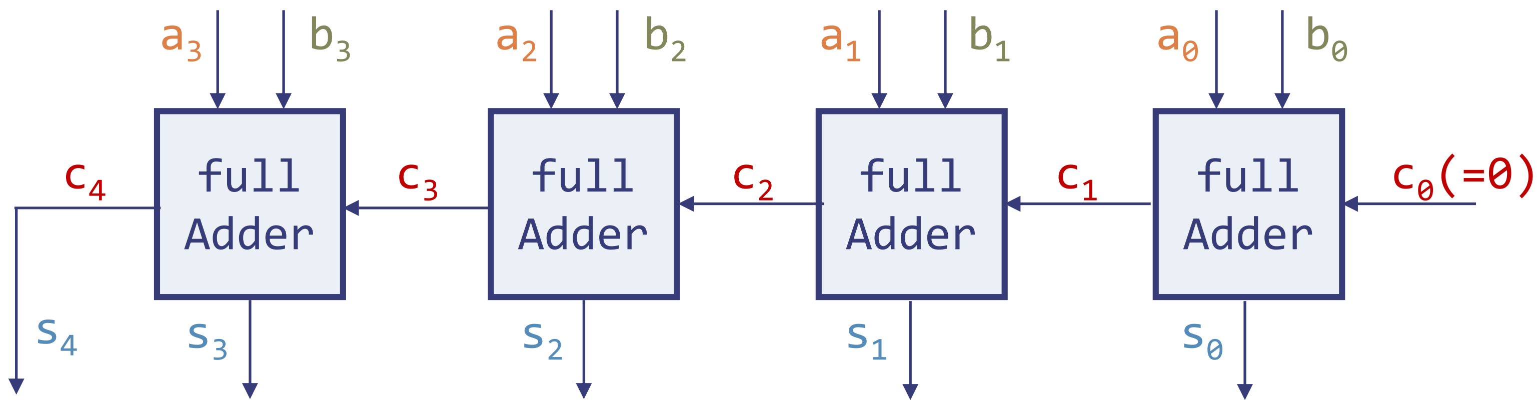


Combinational Logic for an Adder

- First, build a full adder (FA), which
  - Adds three one-bit numbers: a, b, and carry-in
  - Produces a sum bit and a carry-out bit



- Then, cascade FAs to perform binary addition

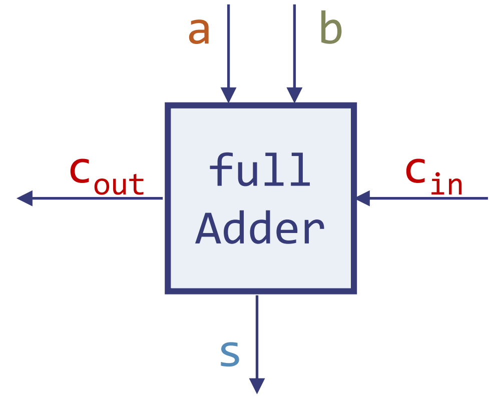


- Result: A ripple-carry adder (simple but slow)

Deriving the Full Adder

Truth table

a	b	c_{in}	c_{out}	s
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		



Boolean expressions

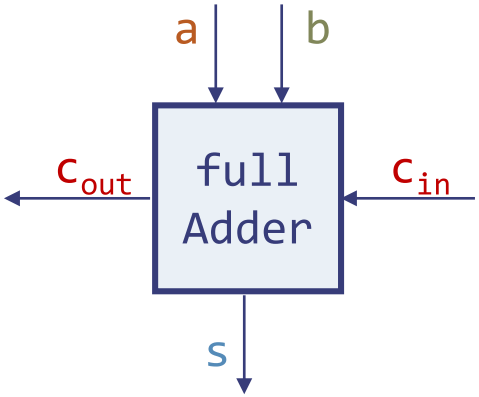
$s =$

$c_{out} =$

Deriving the Full Adder

Truth table

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		



Boolean expressions

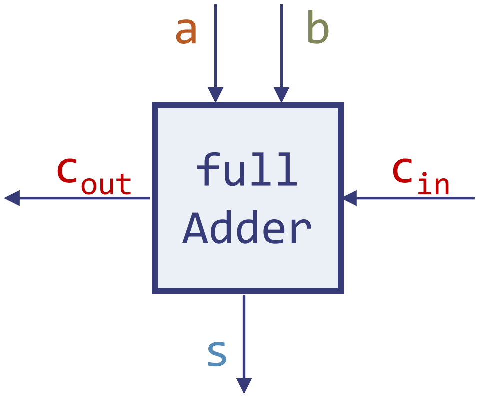
$s =$

$c_{out} =$

Deriving the Full Adder

Truth table

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1		
1	0	0	0	1
1	0	1		
1	1	0		
1	1	1		



Boolean expressions

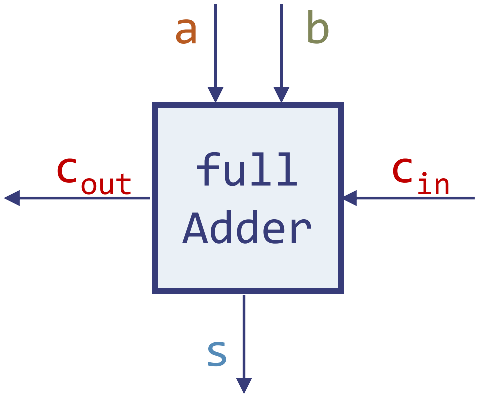
s =

c_{out} =

Deriving the Full Adder

Truth table

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1		



Boolean expressions

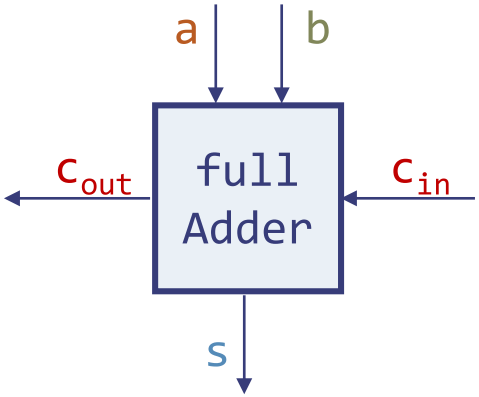
$s =$

$c_{out} =$

Deriving the Full Adder

Truth table

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Boolean expressions

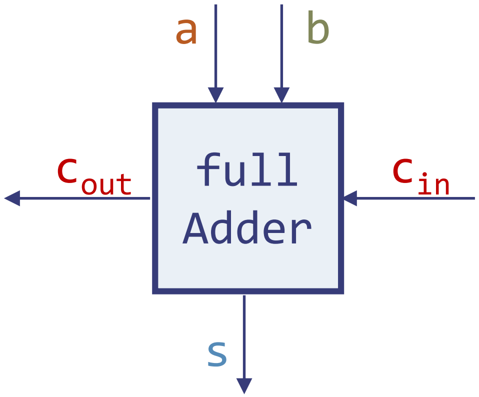
s =

c_{out} =

Deriving the Full Adder

Truth table

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Boolean expressions

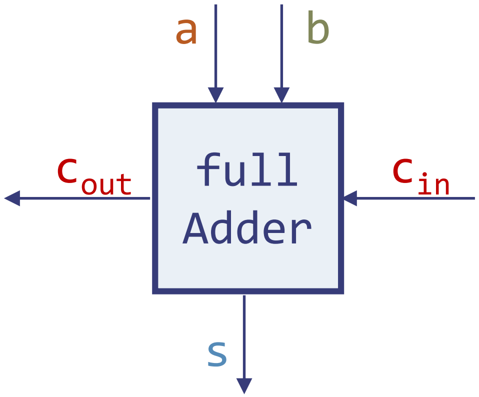
$$s = a \oplus b \oplus c_{in}$$

$$c_{out} =$$

Deriving the Full Adder

Truth table

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Boolean expressions

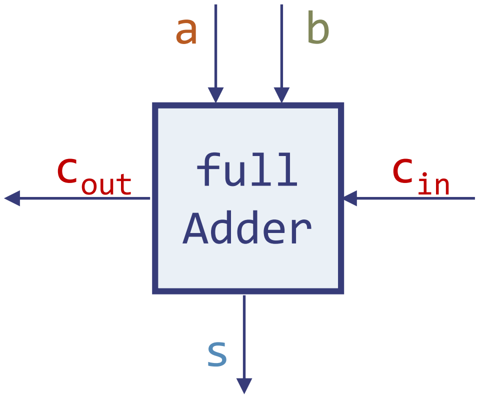
$$s = a \oplus b \oplus c_{in}$$

$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

Deriving the Full Adder

Truth table

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Boolean expressions

$$s = a \oplus b \oplus c_{in}$$

$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

Describing a 32-bit Adder
alternatives

- Truth table with 2^{64} rows and 33 columns

Describing a 32-bit Adder
alternatives

- Truth table with 2^{64} rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA

Describing a 32-bit Adder
alternatives

- Truth table with 2^{64} rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences
  - $s_k = a_k \oplus b_k \oplus c_k$
  - $c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k$

$$\left.\begin{array}{l} s_k = a_k \oplus b_k \oplus c_k \\ c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \end{array}\right\} 0 \le k \le 31$$

Describing a 32-bit Adder
alternatives

- Truth table with 2^{64} rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences
  - $s_k = a_k \oplus b_k \oplus c_k$
  - $c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k$

  $$\left.\begin{array}{l} s_k = a_k \oplus b_k \oplus c_k \\ c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \end{array}\right\} \quad 0 \le k \le 31$$
- Circuit diagrams: tedious to draw, error-prone

Describing a 32-bit Adder
alternatives

- Truth table with 2^{64} rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences
  - $s_k = a_k \oplus b_k \oplus c_k$
  - $c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k$

  $$\left.\begin{array}{l} s_k = a_k \oplus b_k \oplus c_k \\ c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \end{array}\right\} \quad 0 \le k \le 31$$
- Circuit diagrams: tedious to draw, error-prone

Describing a 32-bit Adder

alternatives

- Truth table with 2^{64} rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences
  - $$\left.\begin{array}{l} s_k = a_k \oplus b_k \oplus c_k \\ c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \end{array}\right\} \quad 0 \le k \le 31$$
- Circuit diagrams: tedious to draw, error-prone

- A hardware description language (HDL), *i.e.*, a programming language specialized to describe hardware

Describing a 32-bit Adder
alternatives

- Truth table with 2^{64} rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences

$$\left.\begin{aligned} s_k &= a_k \oplus b_k \oplus c_k \\ c_{k+1} &= a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \end{aligned}\right\} \quad 0 \le k \le 31$$

- Circuit diagrams: tedious to draw, error-prone

- A hardware description language (HDL), *i.e.*, a programming language specialized to describe hardware
  - Precisely specify the structure and behavior of digital circuits

Describing a 32-bit Adder
alternatives

- Truth table with 2^{64} rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences

$$\left.\begin{array}{l} \bullet\ s_k = a_k \oplus b_k \oplus c_k \\ \bullet\ c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \end{array}\right\} \quad 0 \le k \le 31$$

- Circuit diagrams: tedious to draw, error-prone

- A hardware description language (HDL), *i.e.*, a programming language specialized to describe hardware
  - Precisely specify the structure and behavior of digital circuits
  - Designs can be automatically simulated or synthesized to hardware

Describing a 32-bit Adder
alternatives

- Truth table with 2^{64} rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences
  - $s_k = a_k \oplus b_k \oplus c_k$
  - $c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k$

  $\left.\begin{array}{l} s_k = a_k \oplus b_k \oplus c_k \\ c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \end{array}\right\} 0 \le k \le 31$
- Circuit diagrams: tedious to draw, error-prone

- A hardware description language (HDL), *i.e.*, a programming language specialized to describe hardware
  - Precisely specify the structure and behavior of digital circuits
  - Designs can be automatically simulated or synthesized to hardware
  - Enables building hardware with same principles used to build software (write and compose simple, reusable building blocks)

Describing a 32-bit Adder
alternatives

- Truth table with 2^{64} rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences

$$\left.\begin{array}{l} s_k = a_k \oplus b_k \oplus c_k \\ c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \end{array}\right\} \quad 0 \le k \le 31$$

- Circuit diagrams: tedious to draw, error-prone

- A hardware description language (HDL), *i.e.*, a programming language specialized to describe hardware
  - Precisely specify the structure and behavior of digital circuits
  - Designs can be automatically simulated or synthesized to hardware
  - Enables building hardware with same principles used to build software (write and compose simple, reusable building blocks)
  - Uses a familiar syntax (functions, variables, control-flow statements, etc.)

Describing a 32-bit Adder
alternatives

- Truth table with 2^{64} rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences
  - $s_k = a_k \oplus b_k \oplus c_k$
  - $c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k$

  $$\left.\begin{array}{l} s_k = a_k \oplus b_k \oplus c_k \\ c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \end{array}\right\} \; 0 \le k \le 31$$
- Circuit diagrams: tedious to draw, error-prone

- A hardware description language (HDL), *i.e.*, a programming language specialized to describe hardware
  - Precisely specify the structure and behavior of digital circuits
  - Designs can be automatically simulated or synthesized to hardware
  - Enables building hardware with same principles used to build software (write and compose simple, reusable building blocks)
  - Uses a familiar syntax (functions, variables, control-flow statements, etc.)

But be aware of the differences!

Introduction to Minispec

A simple HDL based on Bluespec

Combinational Logic as Functions

- In Minispec, combinational circuits are described using functions

```
function Bool inv(Bool x);
    Bool result = !x;
    return result;
endfunction
```

Combinational Logic as Functions

- In Minispec, combinational circuits are described using functions

Function name

```
function Bool inv(Bool x);
    Bool result = !x;
    return result;
endfunction
```

Combinational Logic as Functions

- In Minispec, combinational circuits are described using functions

Return type

Function name

```
function Bool inv(Bool x);
    Bool result = !x;
    return result;
endfunction
```

Combinational Logic as Functions

- In Minispec, combinational circuits are described using functions

Return type

Function name

Input arguments

```
function Bool inv(Bool x);
    Bool result = !x;
    return result;
endfunction
```

Combinational Logic as Functions

- In Minispec, combinational circuits are described using functions

Return type

Function name

Input arguments

```
function Bool inv(Bool x);
    Bool result = !x;
    return result;
endfunction
```

Statement(s), including a return statement

Combinational Logic as Functions

- In Minispec, combinational circuits are described using functions

Return type

Function name

Input arguments

```
function Bool inv(Bool x);
    Bool result = !x;
    return result;
endfunction
```

Statement(s), including a return statement

- All values have a fixed type, which is known statically (e.g., result is of type Bool)

Combinational Logic as Functions

- In Minispec, combinational circuits are described using functions

Return type

Function name

Input arguments

```
function Bool inv(Bool x);
    Bool result = !x;
    return result;
endfunction
```

Statement(s), including a return statement

- All values have a fixed type, which is known statically (e.g., result is of type Bool)
- Note: Types Start With An Uppercase Letter, variable and function names are lowercase

Bool Type and Operations

- Values of type Bool can be True or False
- Bool supports Boolean and comparison operations:

```
Bool a = True;
Bool b = False;

Bool x = !a;      // False since a == True
Bool y = a && b; // False since b == False
Bool z = a || b; // True since a == True

Bool n = a != b; // True; equivalent to XOR
Bool e = a == b; // False; equivalent to XNOR
```

Bool Type and Operations

- Values of type Bool can be True or False
- Bool supports Boolean and comparison operations:

```
Bool a = True;
Bool b = False;

Bool x = !a;      // False since a == True
Bool y = a && b; // False since b == False
Bool z = a || b; // True since a == True

Bool n = a != b; // True; equivalent to XOR
Bool e = a == b; // False; equivalent to XNOR
```

- Bool is the simplest type, but working with many single-bit values is tedious
  - Need a type that represents multi-bit values!

Bit#(n) Type and Operations

- Bit#(n) represents an n-bit value
- Bit#(n) supports the following basic operations:
  - Bitwise logical: ~ (negation), & (AND), | (OR), ^ (XOR)

```
Bit#(4) a = 4'b0011;  // 4-bit binary 3
Bit#(4) b = 4'b0101;  // 4-bit binary 5
Bit#(4) x = ~a;       // 4'b1100
Bit#(4) y = a & b;    // 4'b0001
Bit#(4) z = a ^ b;    // 4'b0110
```

Bit#(n) Type and Operations

- Bit#(n) represents an n-bit value
- Bit#(n) supports the following basic operations:
  - Bitwise logical: ~ (negation), & (AND), | (OR), ^ (XOR)

```
Bit#(4) a = 4'b0011;  // 4-bit binary 3
Bit#(4) b = 4'b0101;  // 4-bit binary 5
Bit#(4) x = ~a;       // 4'b1100
Bit#(4) y = a & b;    // 4'b0001
Bit#(4) z = a ^ b;    // 4'b0110
```

  - Bit selection

```
Bit#(1) l = a[0];   // 1'b1 (least significant)
Bit#(3) m = a[3:1]; // 3'b001
```

Bit#(n) Type and Operations

- Bit#(n) represents an n-bit value
- Bit#(n) supports the following basic operations:
  - Bitwise logical: ~ (negation), & (AND), | (OR), ^ (XOR)

```
Bit#(4) a = 4'b0011;  // 4-bit binary 3
Bit#(4) b = 4'b0101;  // 4-bit binary 5
Bit#(4) x = ~a;       // 4'b1100
Bit#(4) y = a & b;    // 4'b0001
Bit#(4) z = a ^ b;    // 4'b0110
```

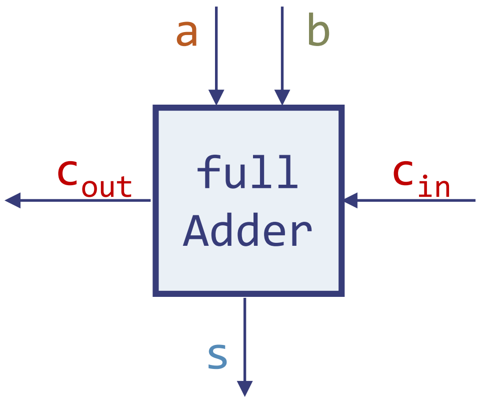
  - Bit selection

```
Bit#(1) l = a[0];   // 1'b1 (least significant)
Bit#(3) m = a[3:1]; // 3'b001
```

  - Concatenation

```
Bit#(8) c = {a, b}; // 8'b00110101
```

Full Adder in Minispec

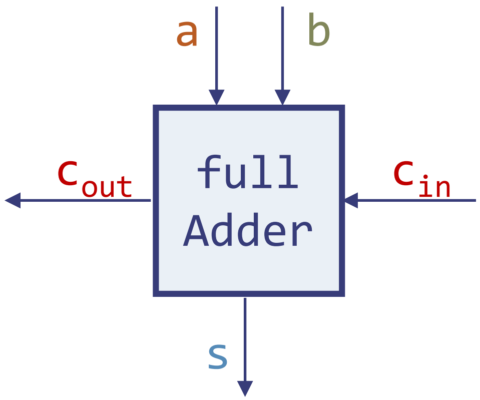


$$s = a \oplus b \oplus c_{in}$$

$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

```
function Bit#(2) fullAdder(Bit#(1) a, Bit#(1) b, Bit#(1) cin);


endfunction
```

Full Adder in Minispec

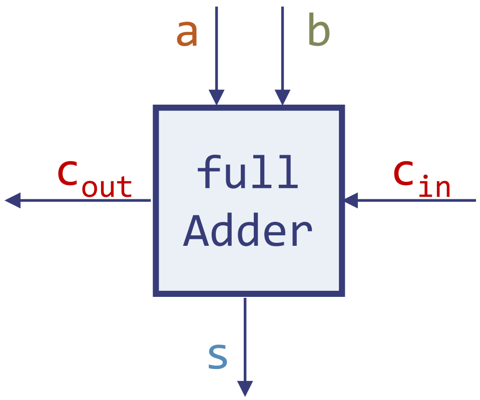


$$s = a \oplus b \oplus c_{in}$$

$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

```
function Bit#(2) fullAdder(Bit#(1) a, Bit#(1) b, Bit#(1) cin);
    Bit#(1) s = a ^ b ^ cin;


endfunction
```

Full Adder in Minispec

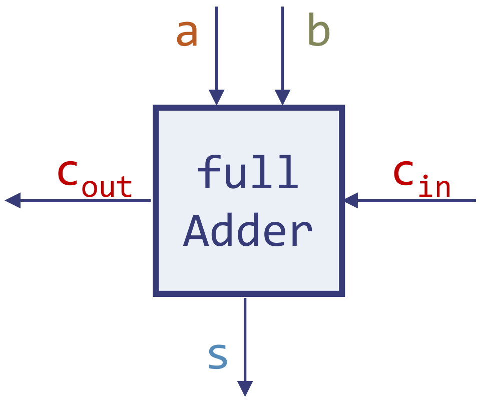


$$s = a \oplus b \oplus c_{in}$$

$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

```
function Bit#(2) fullAdder(Bit#(1) a, Bit#(1) b, Bit#(1) cin);
    Bit#(1) s = a ^ b ^ cin;
    Bit#(1) cout = (a & b) | (a & cin) | (b & cin);

endfunction
```

Full Adder in Minispec

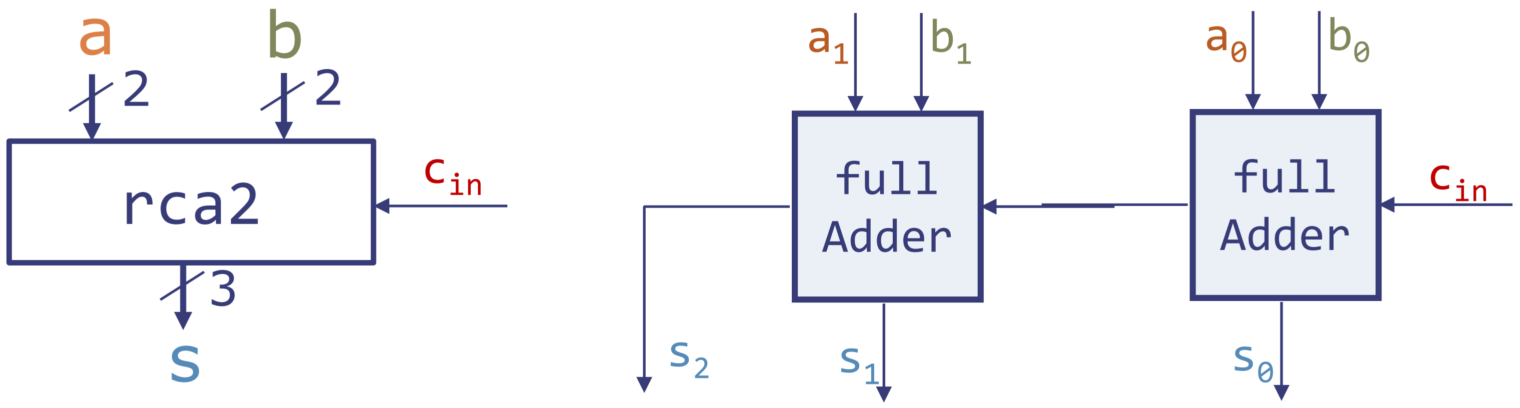


$$s = a \oplus b \oplus c_{in}$$

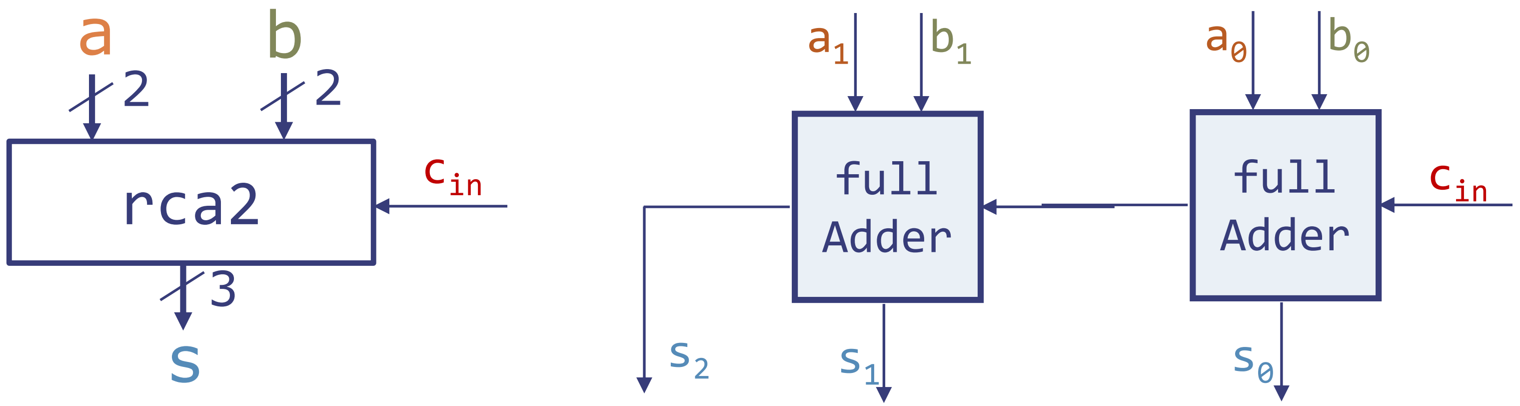
$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

```
function Bit#(2) fullAdder(Bit#(1) a, Bit#(1) b, Bit#(1) cin);
    Bit#(1) s = a ^ b ^ cin;
    Bit#(1) cout = (a & b) | (a & cin) | (b & cin);
    return {cout, s};
endfunction
```

2-bit Ripple-Carry Adder

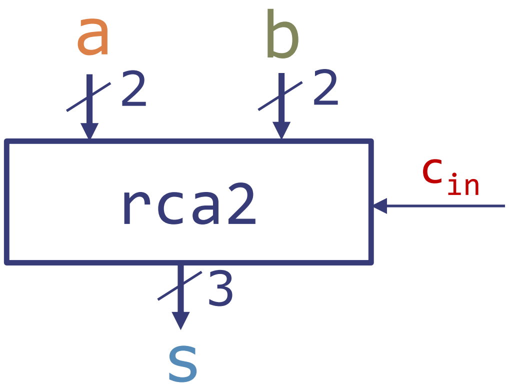


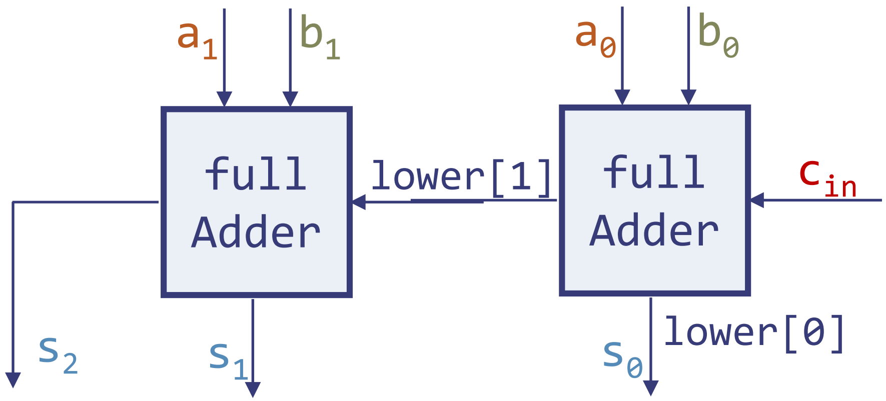
2-bit Ripple-Carry Adder




```
function Bit#(3) rca2(Bit#(2) a, Bit#(2) b, Bit#(1) cin);


endfunction
```

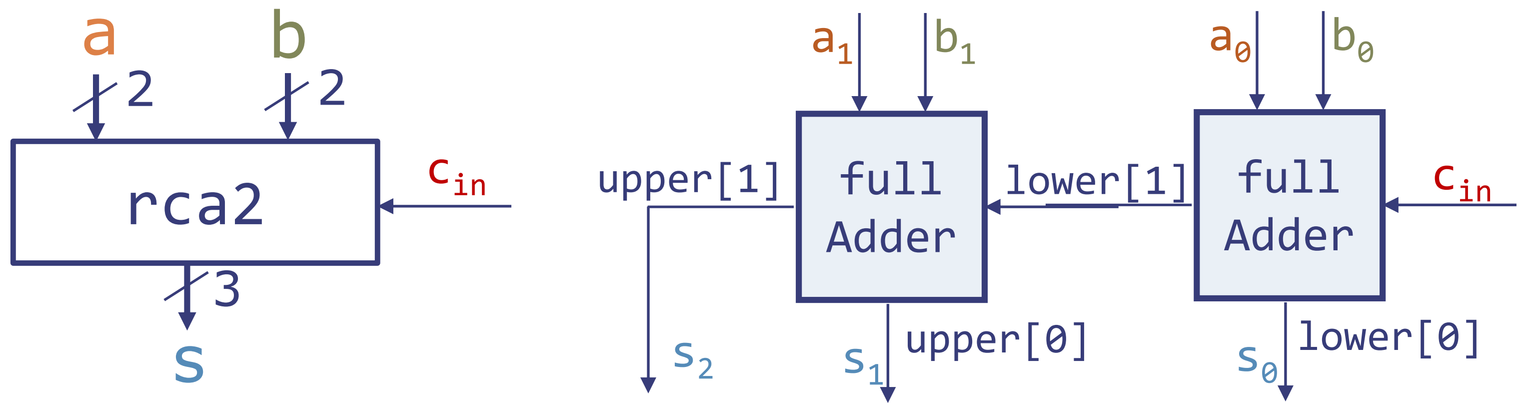
2-bit Ripple-Carry Adder





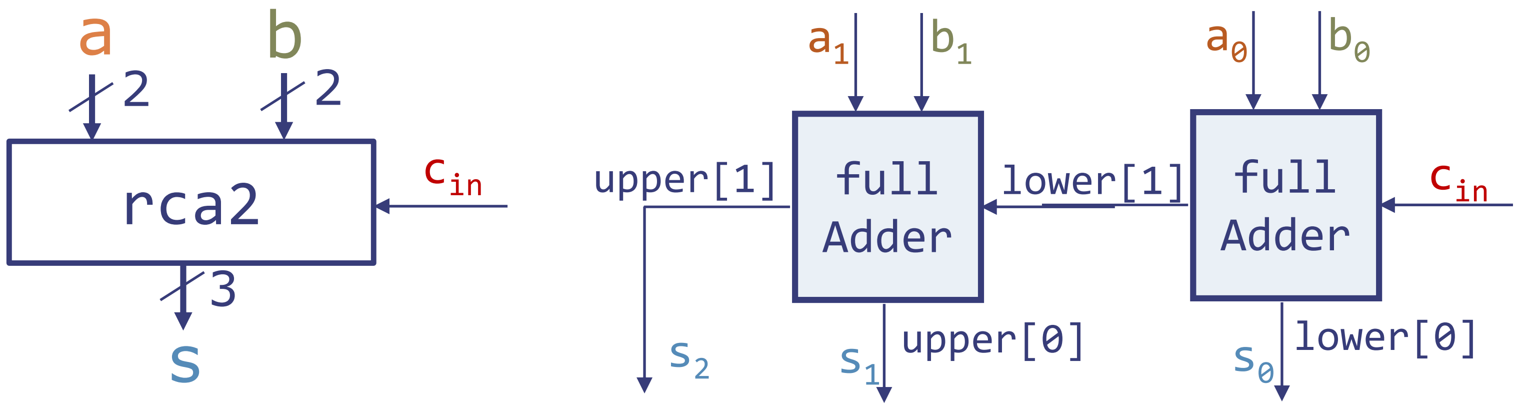

```
function Bit#(3) rca2(Bit#(2) a, Bit#(2) b, Bit#(1) cin);
    Bit#(2) lower = fullAdder(a[0], b[0], cin);


endfunction
```

2-bit Ripple-Carry Adder



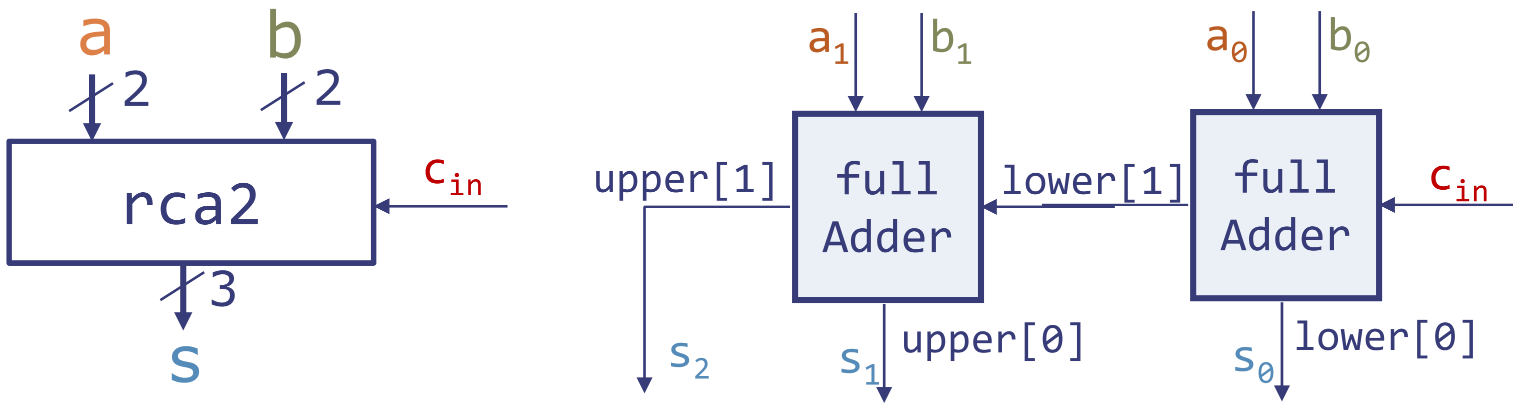

```
function Bit#(3) rca2(Bit#(2) a, Bit#(2) b, Bit#(1) cin);
    Bit#(2) lower = fullAdder(a[0], b[0], cin);
    Bit#(2) upper = fullAdder(a[1], b[1], lower[1]);

endfunction
```

2-bit Ripple-Carry Adder




```
function Bit#(3) rca2(Bit#(2) a, Bit#(2) b, Bit#(1) cin);
    Bit#(2) lower = fullAdder(a[0], b[0], cin);
    Bit#(2) upper = fullAdder(a[1], b[1], lower[1]);
    return {upper, lower[0]};
endfunction
```

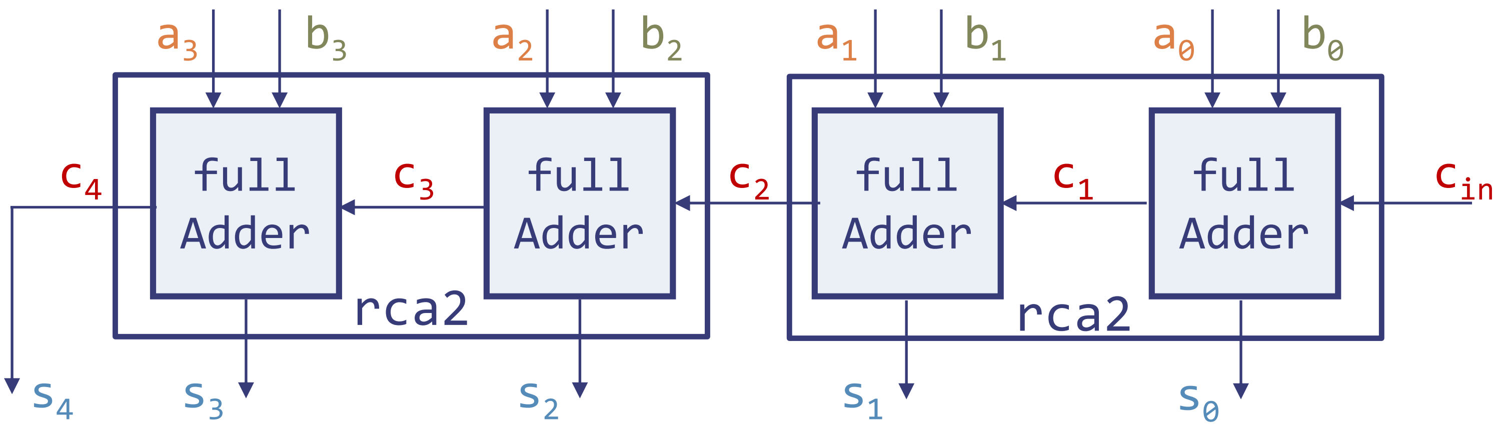
2-bit Ripple-Carry Adder




```
function Bit#(3) rca2(Bit#(2) a, Bit#(2) b, Bit#(1) cin);
    Bit#(2) lower = fullAdder(a[0], b[0], cin);
    Bit#(2) upper = fullAdder(a[1], b[1], lower[1]);
    return {upper, lower[0]};
endfunction
```

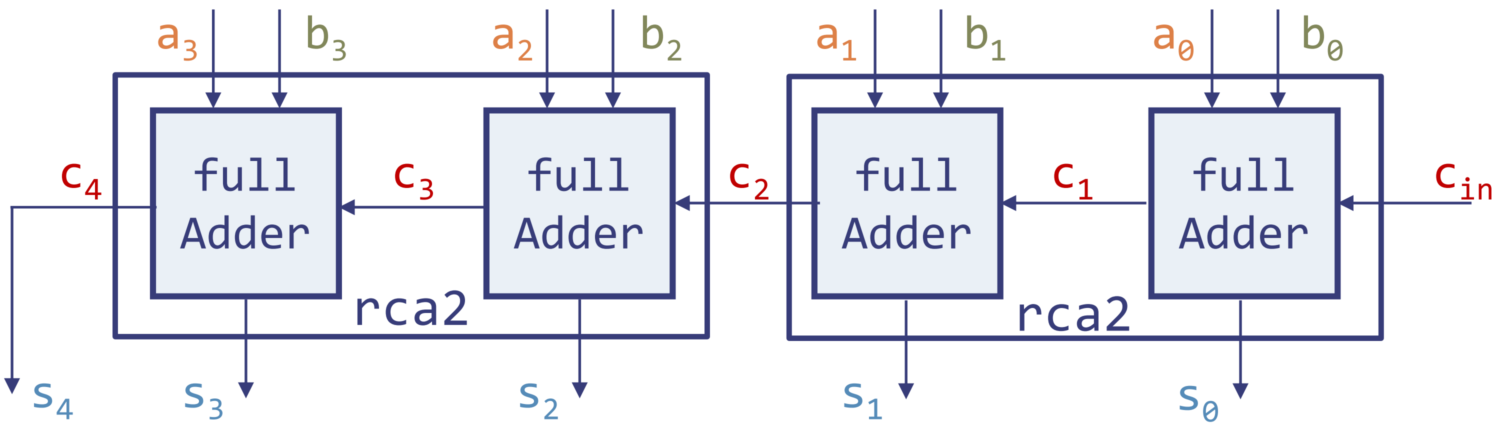
- Functions are inlined: Each function call creates a new instance (copy) of the called circuit
  - Allows composing simple circuits to build larger ones

4-bit Ripple-Carry Adder




```
function Bit#(5) rca4(Bit#(4) a, Bit#(4) b, Bit#(1) cin);
    Bit#(3) lower = rca2(a[1:0], b[1:0], cin);
    Bit#(3) upper = rca2(a[3:2], b[3:2], lower[2]);
    return {upper, lower[1:0]};
endfunction
```

4-bit Ripple-Carry Adder



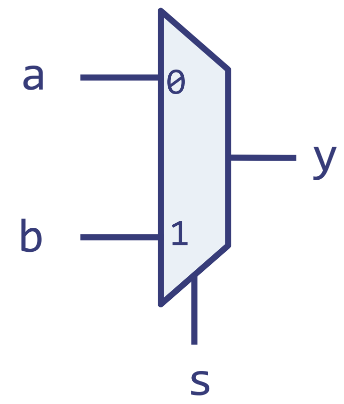

```
function Bit#(5) rca4(Bit#(4) a, Bit#(4) b, Bit#(1) cin);
    Bit#(3) lower = rca2(a[1:0], b[1:0], cin);
    Bit#(3) upper = rca2(a[3:2], b[3:2], lower[2]);
    return {upper, lower[1:0]};
endfunction
```

- Composing functions lets us build larger circuits, but writing very large circuits this way is tedious
  - Next lecture: Writing an n-bit adder in a single function

Multiplexers

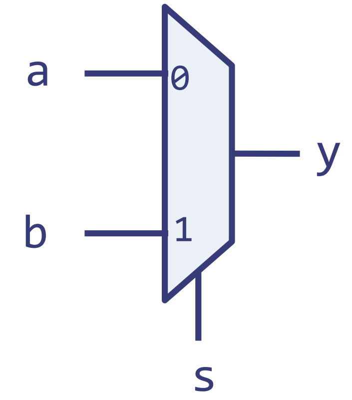
2-way Multiplexer

- A 2-way multiplexer or mux selects between two inputs a and b based on a single-bit input s (select input)



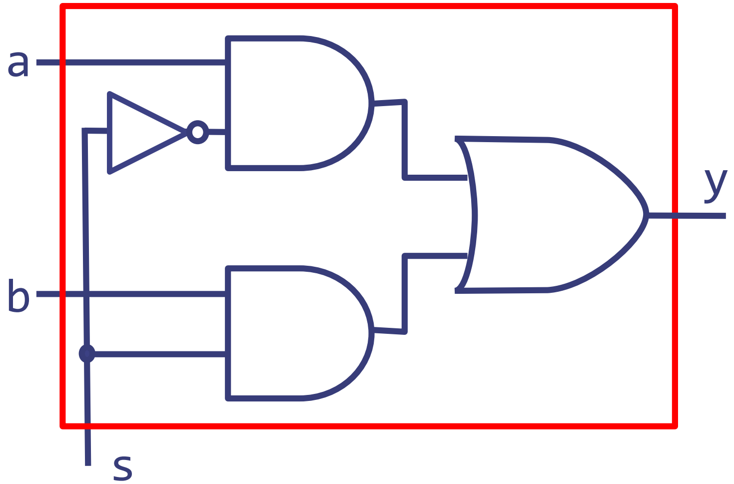
2-way Multiplexer

- A 2-way multiplexer or mux selects between two inputs a and b based on a single-bit input s (select input)



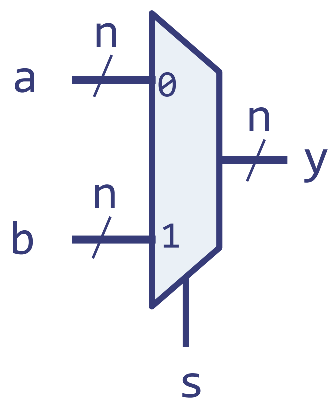
$$y = a \cdot \overline{s} + b \cdot s$$

- Gate-level implementation:



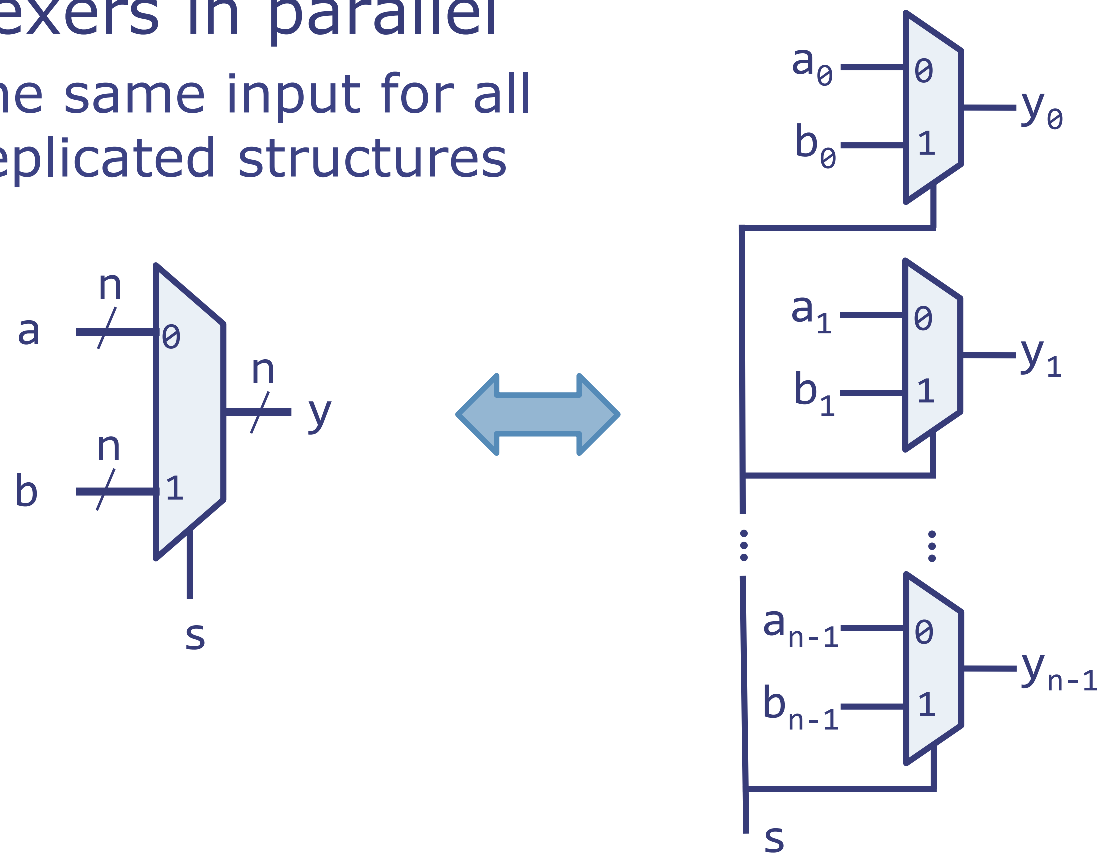
2-way Multiplexer with *n*-bit inputs

- If *a* and *b* are *n*-bit wide, the 2-way multiplexer can be implemented with *n* one-bit 2-way multiplexers in parallel
  - s is the same input for all the replicated structures



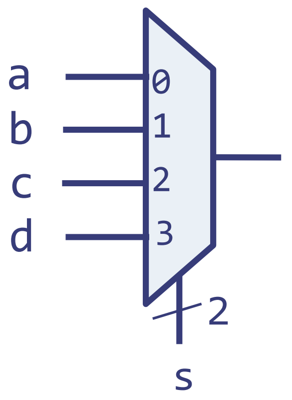
2-way Multiplexer with *n*-bit inputs

- If *a* and *b* are *n*-bit wide, the 2-way multiplexer can be implemented with *n* one-bit 2-way multiplexers in parallel
  - s is the same input for all the replicated structures



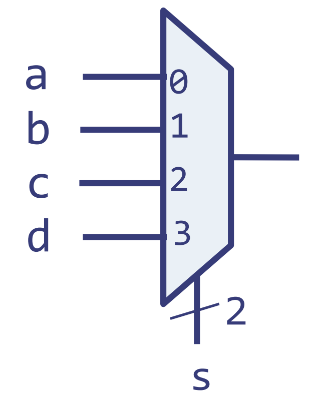
4-way Multiplexer

- A 4-way multiplexer selects between four inputs based on the value of a 2-bit input s

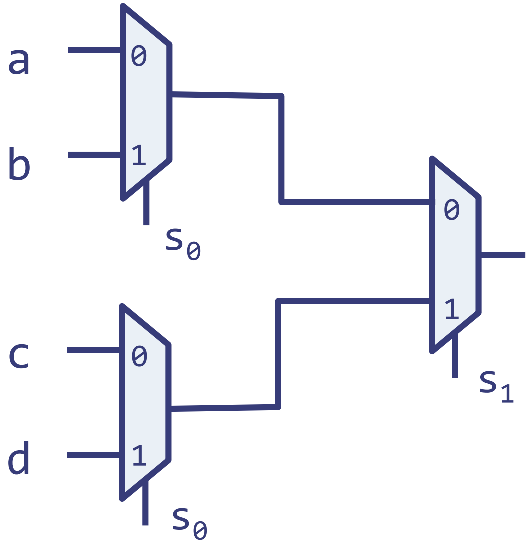


4-way Multiplexer

- A 4-way multiplexer selects between four inputs based on the value of a 2-bit input s

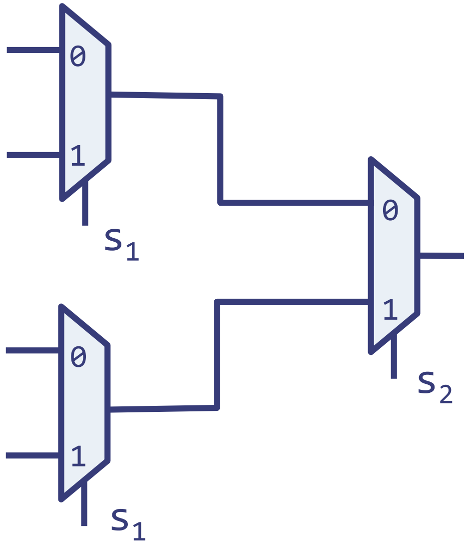


- Typically implemented using 2-way multiplexers



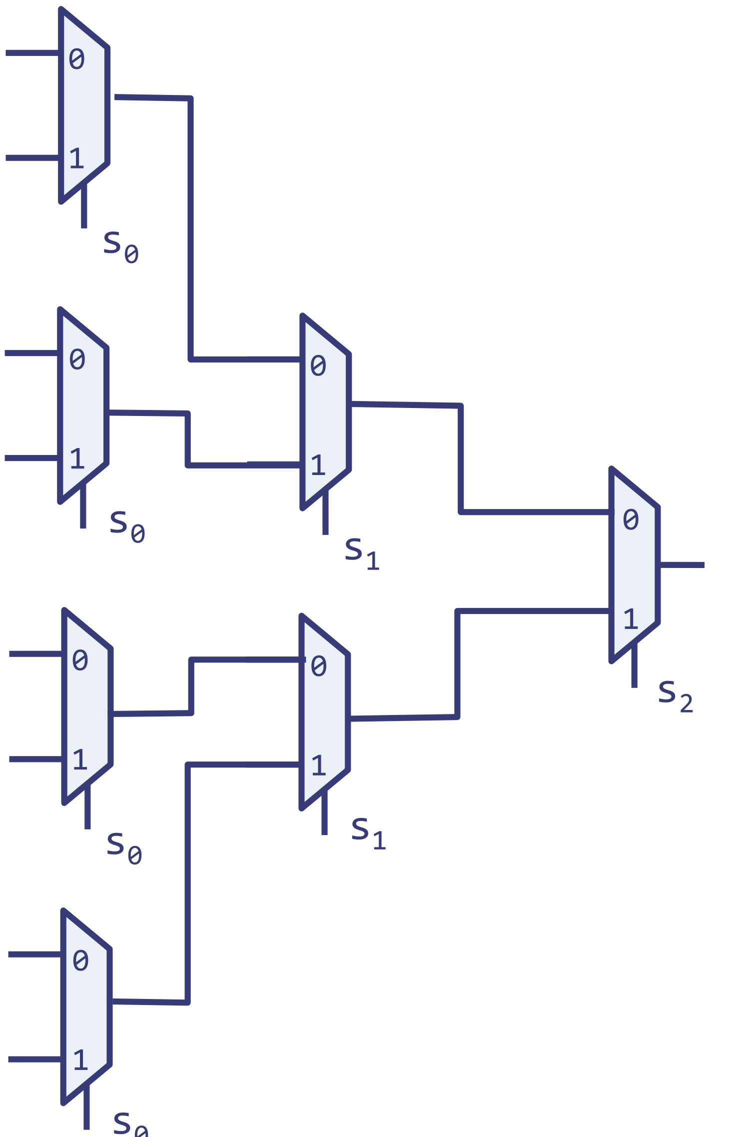
k-way Multiplexer

- A k-way multiplexer can be implemented with a tree of k-1 2-way multiplexers



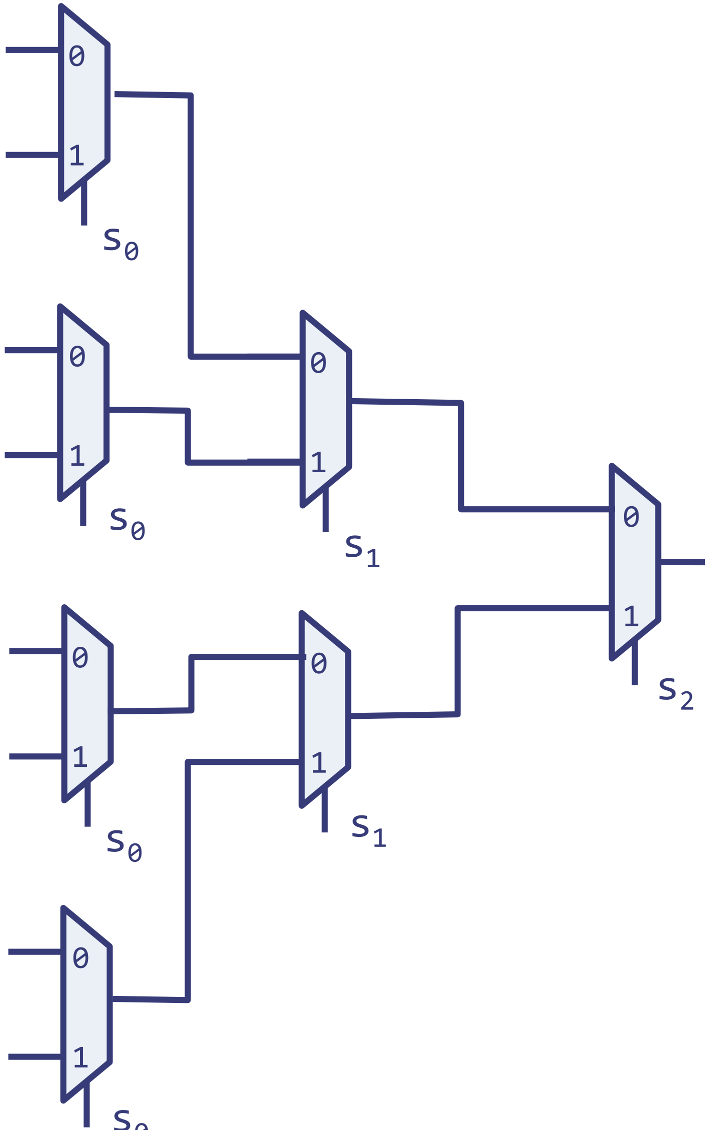
k-way Multiplexer

- A k-way multiplexer can be implemented with a tree of k-1 2-way multiplexers
  - Example: 8-way multiplexer



k-way Multiplexer

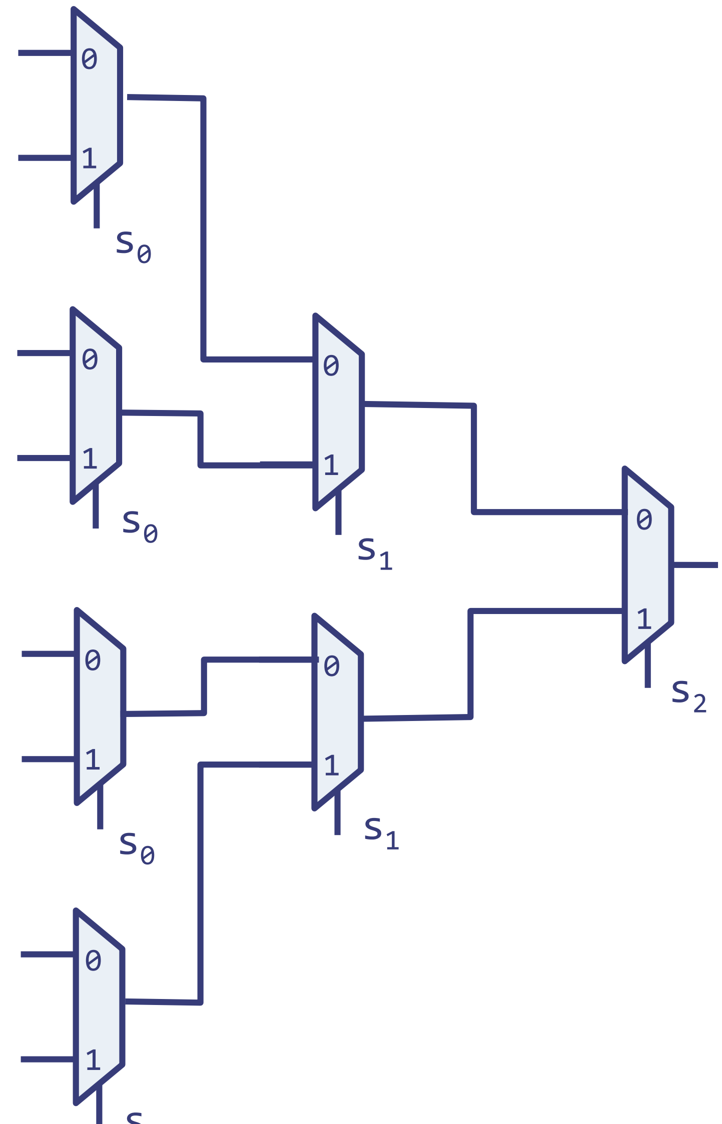
- A k-way multiplexer can be implemented with a tree of k-1 2-way multiplexers
  - Example: 8-way multiplexer



How many 2-way one-bit muxes needed to implement a k-way n-bit mux?

k-way Multiplexer

- A k-way multiplexer can be implemented with a tree of k-1 2-way multiplexers
  - Example: 8-way multiplexer

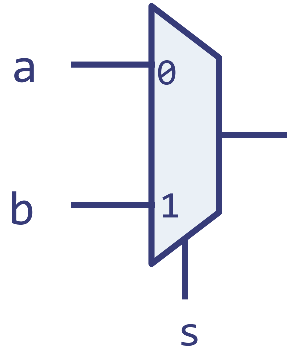


How many 2-way one-bit muxes needed to implement a k-way n-bit mux?

(k-1)*n

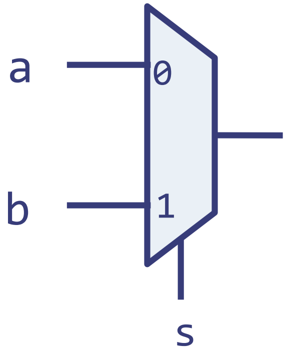
Multiplexers in Minispec

- 2-way mux → Conditional operator



Multiplexers in Minispec

- 2-way mux → Conditional operator

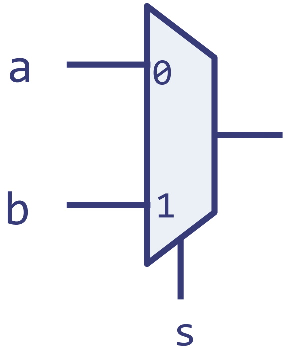


Minispec

$\boxed{\texttt{s? b : a}}$

Multiplexers in Minispec

- 2-way mux → Conditional operator

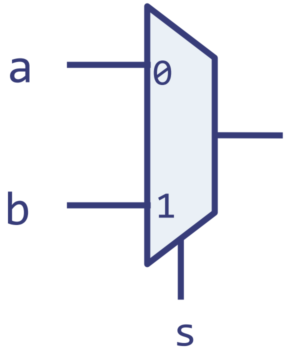


Minispec `s? b : a`

s has type Bool; True is treated as 1 and False as 0

Multiplexers in Minispec

- 2-way mux → Conditional operator



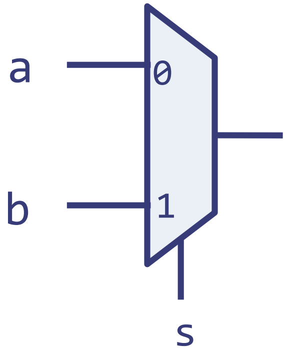
Minispec `s? b : a`

Python `b if s else a`

s has type Bool; True is treated as 1 and False as 0

Multiplexers in Minispec

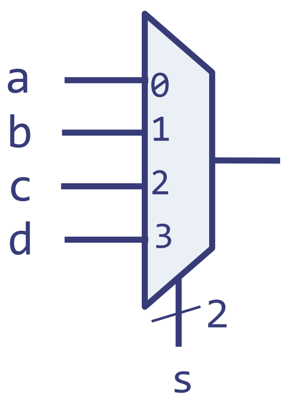
- 2-way mux → Conditional operator



Minispec	`s? b : a`
Python	`b if s else a`

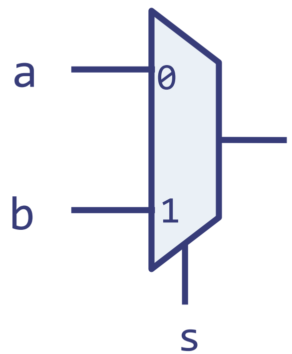
s has type Bool; True is treated as 1 and False as 0

- *k*-way mux → Case expression



Multiplexers in Minispec

- 2-way mux → Conditional operator

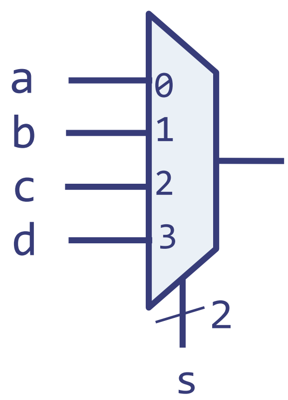


Minispec `s? b : a`

Python `b if s else a`

s has type Bool; True is treated as 1 and False as 0

- *k*-way mux → Case expression

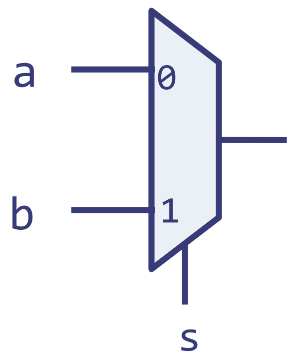


Minispec

```
case (s)
    0 :  a;
    1 :  b;
    2 :  c;
    3 :  d;
endcase
```

Multiplexers in Minispec

- 2-way mux → Conditional operator



Minispec

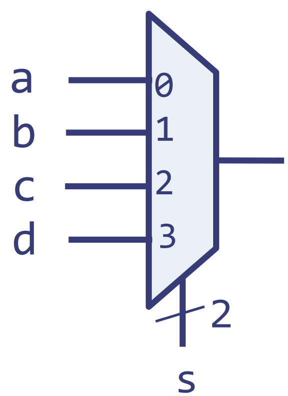
```
s? b : a
```

Python

```
b if s else a
```

s has type Bool; True is treated as 1 and False as 0

- *k*-way mux → Case expression



Minispec

```
case (s)
    0 :  a;
    1 :  b;
    2 :  c;
    3 :  d;
endcase
```

s has type Bit#(2)

Aside: No Conditional Execution!

- Given this conditional statement…

```
s? foo(x) : bar(y)
```

Aside: No Conditional Execution!

- Given this conditional statement…

```
s? foo(x) : bar(y)
```

- In software, the program would first evaluate s, then run either `foo(x)` or `bar(y)`

Aside: No Conditional Execution!

- Given this conditional statement…

```
s? foo(x) : bar(y)
```

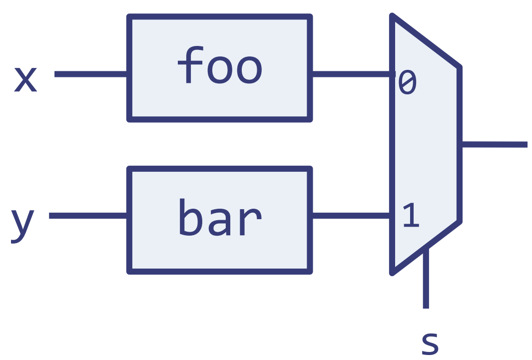
- In software, the program would first evaluate s, then run either `foo(x)` or `bar(y)`
- But in hardware, this statement instantiates and evaluates both `foo(x)` and `bar(y)`, in parallel!

Aside: No Conditional Execution!

- Given this conditional statement…

```
s? foo(x) : bar(y)
```

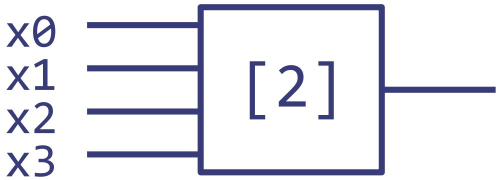
- In software, the program would first evaluate s, then run either `foo(x)` or `bar(y)`
- But in hardware, this statement instantiates and evaluates both `foo(x)` and `bar(y)`, in parallel!



Selecting a Wire: x[i]

assume x is 4 bits wide

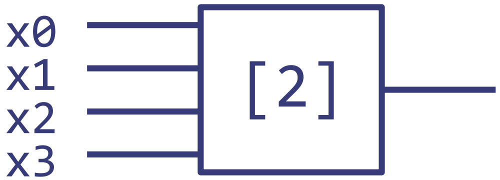
- Constant selector: e.g., x[2]

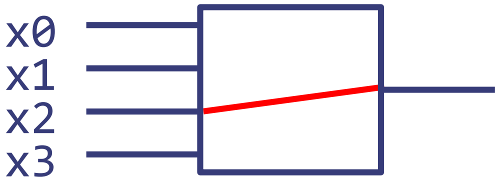


Selecting a Wire: x[i]

assume x is 4 bits wide

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

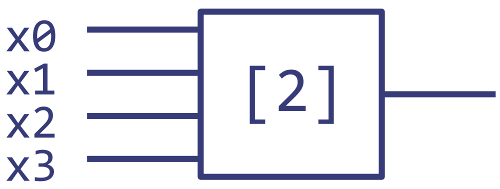


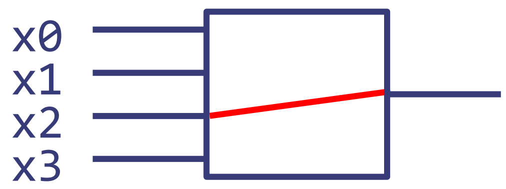


Selecting a Wire: x[i]

assume x is 4 bits wide

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



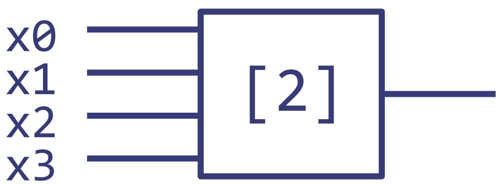


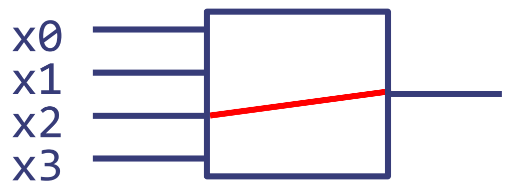
no hardware; x[2] is just the name of a wire

Selecting a Wire: x[i]

assume x is 4 bits wide

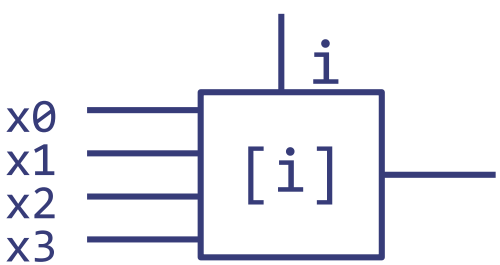
- Constant selector: e.g., x[2]





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

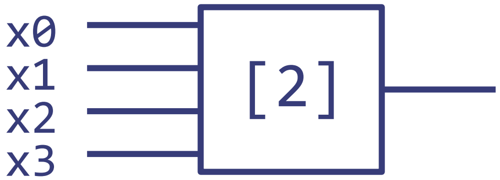
- Dynamic selector: x[i]

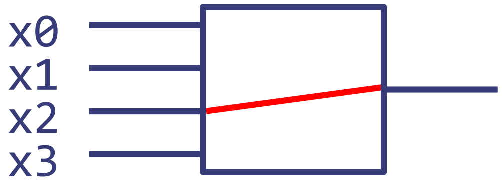


Selecting a Wire: x[i]

assume x is 4 bits wide

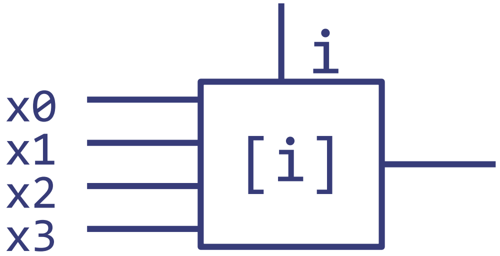
- Constant selector: e.g., x[2]

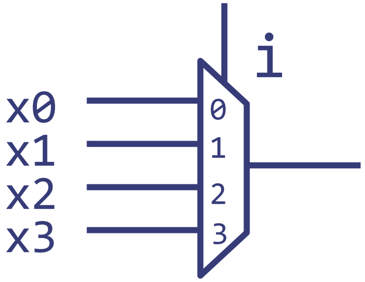




no hardware; x[2] is just the name of a wire

- Dynamic selector: x[i]

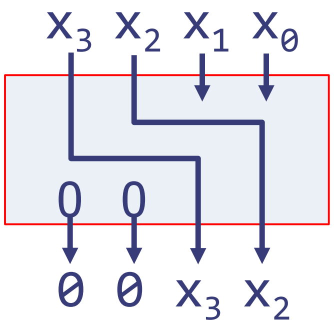




4-way mux

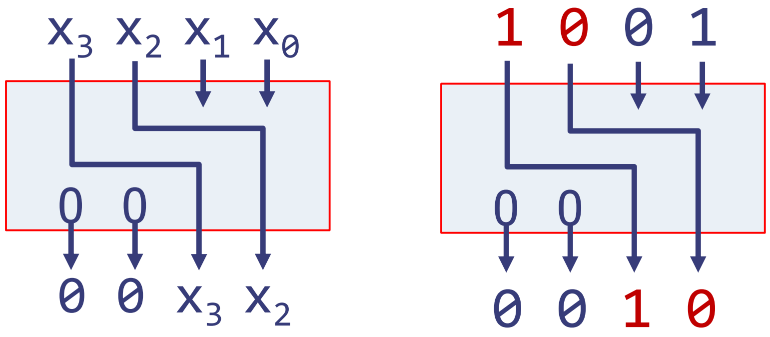
Shift operators

Fixed-Size Shifts



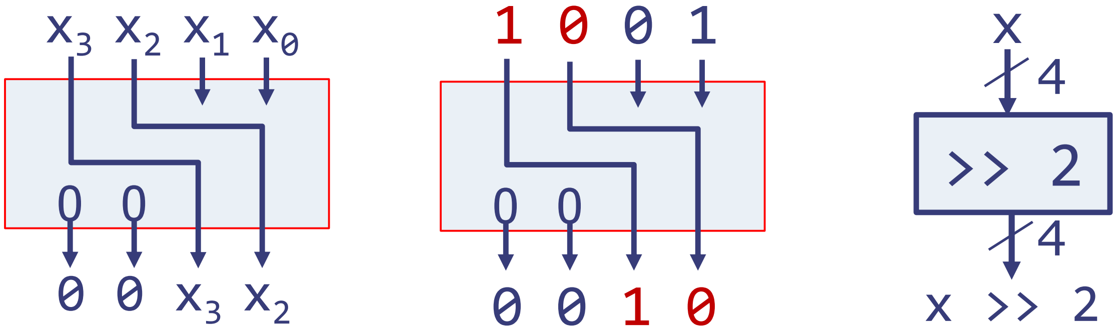
- Fixed-size shift operation is cheap in hardware
  - Just wire the circuit appropriately

Fixed-Size Shifts



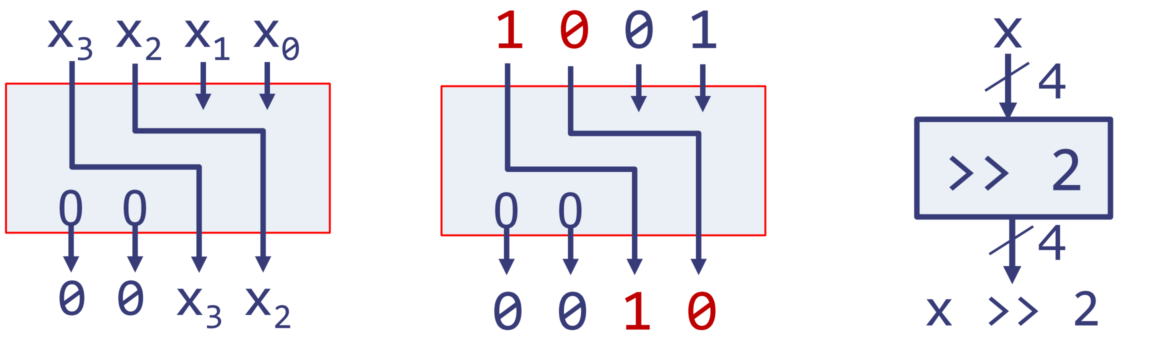
- Fixed-size shift operation is cheap in hardware
  - Just wire the circuit appropriately

Fixed-Size Shifts

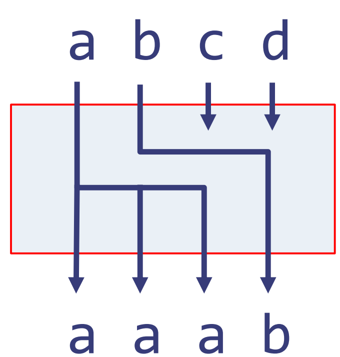


- Fixed-size shift operation is cheap in hardware
  - Just wire the circuit appropriately

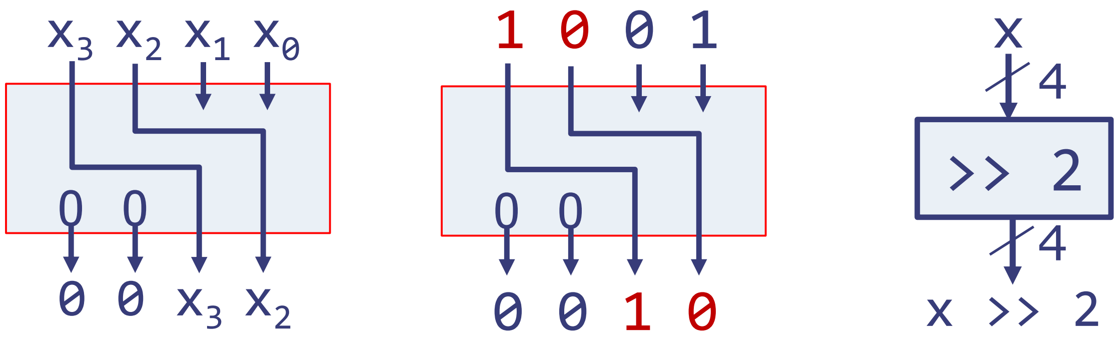
Fixed-Size Shifts



- Fixed-size shift operation is cheap in hardware
  - Just wire the circuit appropriately
- Arithmetic shifts are similar

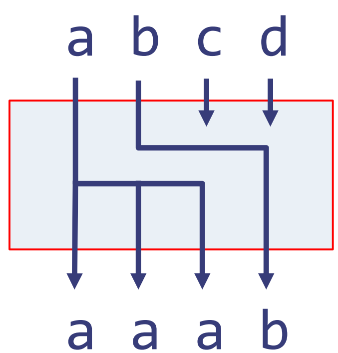


Fixed-Size Shifts

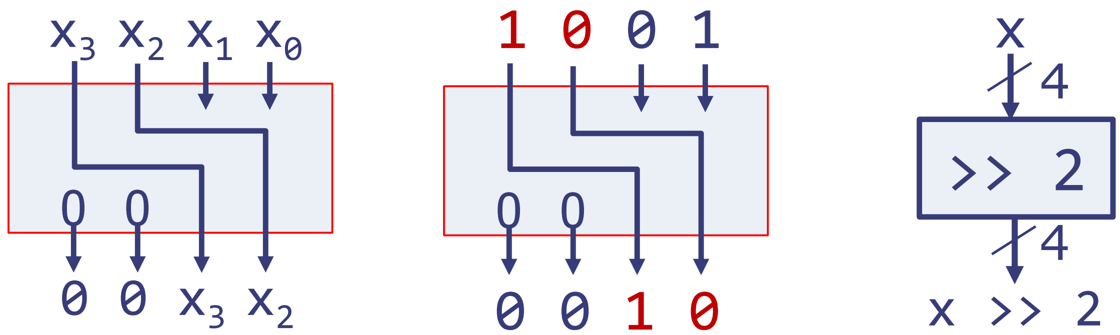


- Fixed-size shift operation is cheap in hardware
  - Just wire the circuit appropriately
- Arithmetic shifts are similar

Arithmetic right shift by n divides integer in two's complement representation by 2^n

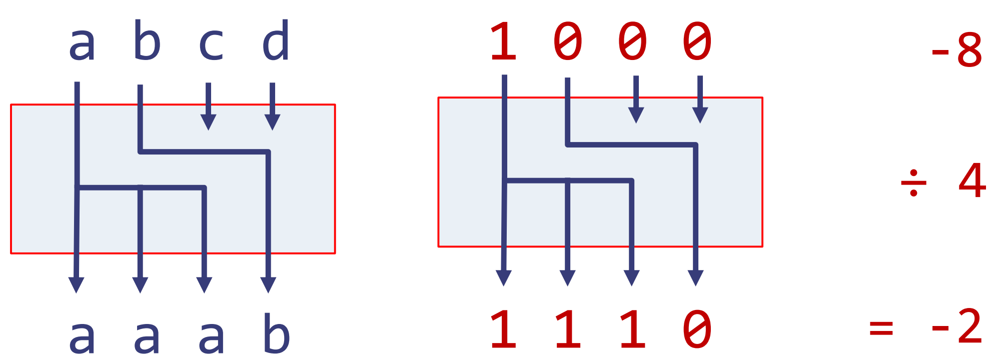


Fixed-Size Shifts



- Fixed-size shift operation is cheap in hardware
  - Just wire the circuit appropriately
- Arithmetic shifts are similar

Arithmetic right shift by n divides integer in two's complement representation by 2^n



Logical Right Shift by *s*

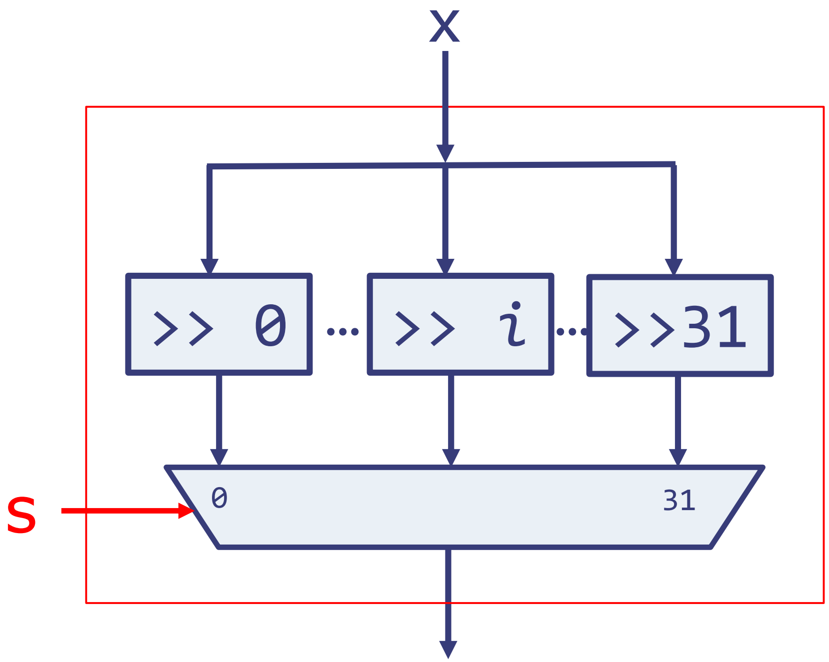
- Suppose we want a shifter that right-shifts an N-bit input x by *s*, where $N=32$ and $0 \le s \le 31$

Logical Right Shift by *s*

- Suppose we want a shifter that right-shifts an N-bit input x by *s*, where N=32 and $0 \leq s \leq 31$
- Naïve approach: Create 32 different fixed-size shifters and select using a mux

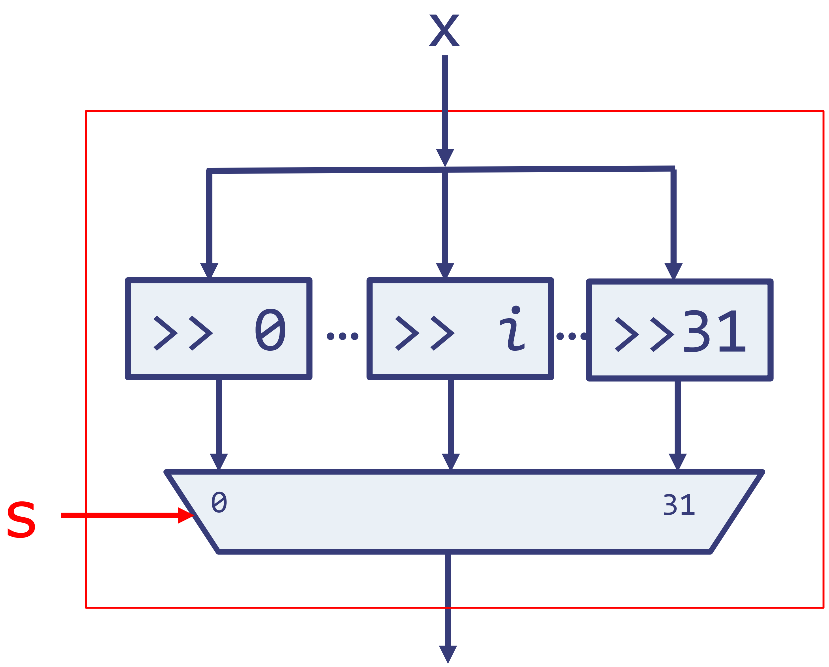
Logical Right Shift by *s*

- Suppose we want a shifter that right-shifts an N-bit input x by *s*, where N=32 and 0≤s≤31
- Naïve approach: Create 32 different fixed-size shifters and select using a mux



Logical Right Shift by *s*

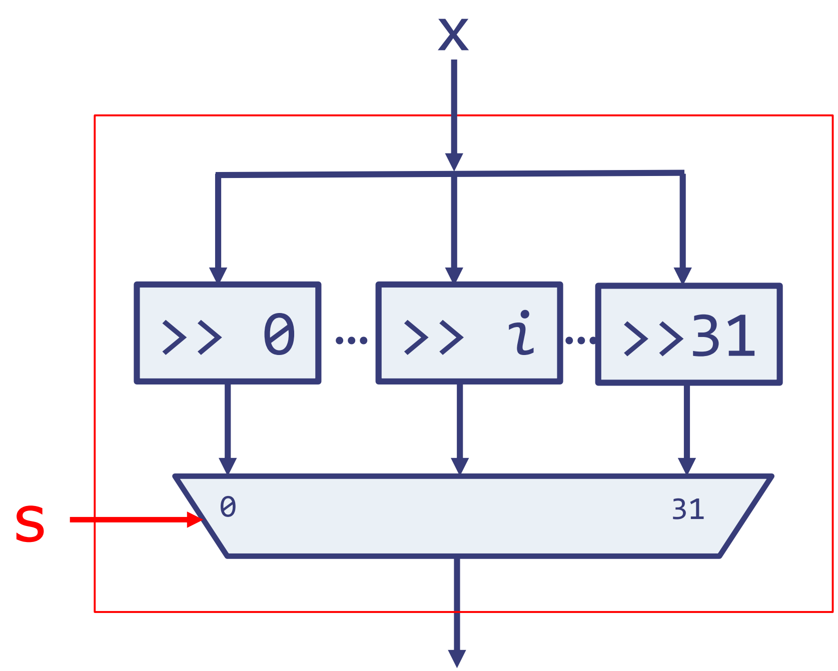
- Suppose we want a shifter that right-shifts an N-bit input x by *s*, where N=32 and 0≤s≤31
- Naïve approach: Create 32 different fixed-size shifters and select using a mux



How many 2-way 1-bit muxes are needed to implement this 32-way 32-bit mux?

Logical Right Shift by *s*

- Suppose we want a shifter that right-shifts an N-bit input x by *s*, where N=32 and 0≤s≤31
- Naïve approach: Create 32 different fixed-size shifters and select using a mux

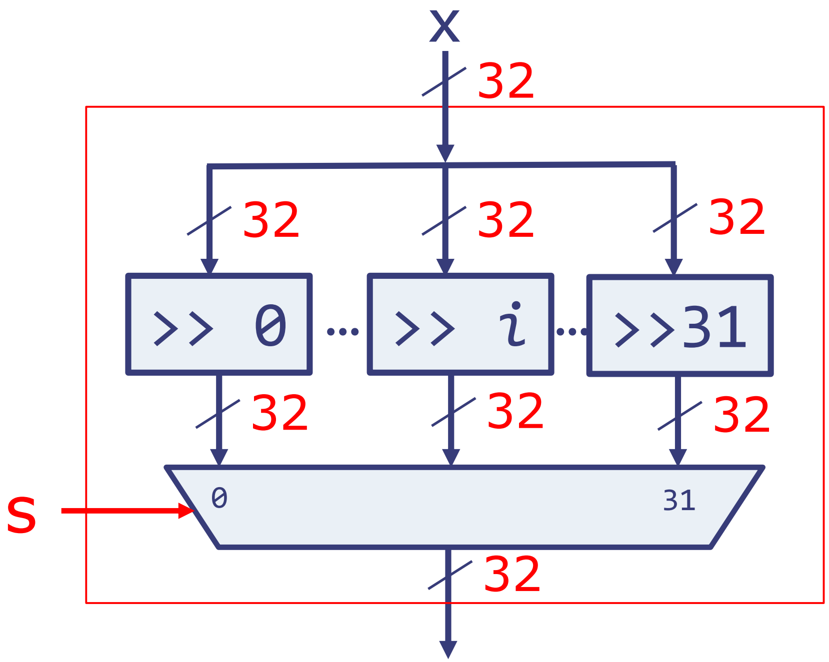


How many 2-way 1-bit muxes are needed to implement this 32-way 32-bit mux?

(32-1)

Logical Right Shift by *s*

- Suppose we want a shifter that right-shifts an N-bit input x by *s*, where N=32 and 0≤s≤31
- Naïve approach: Create 32 different fixed-size shifters and select using a mux

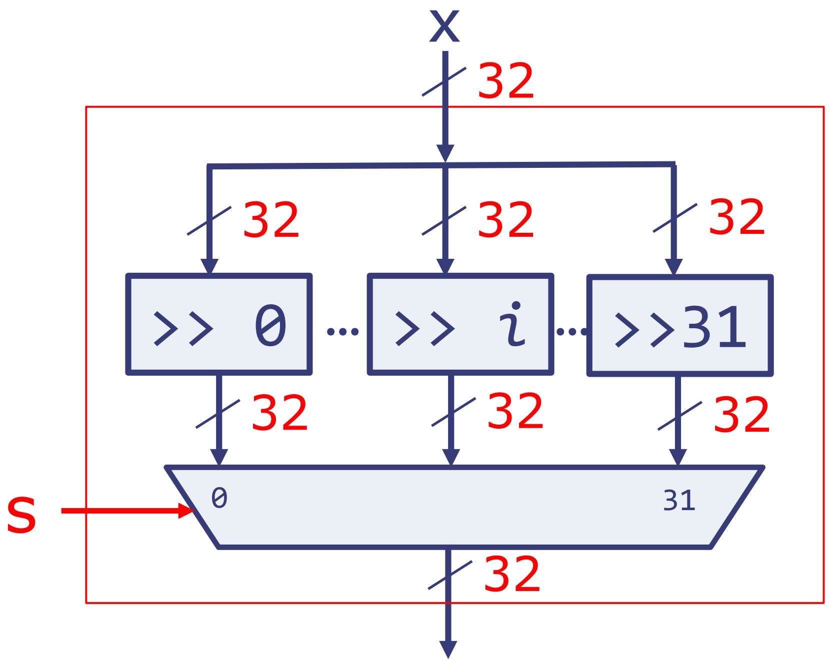


How many 2-way 1-bit muxes are needed to implement this 32-way 32-bit mux?

(32-1)*32 = 992

Logical Right Shift by *s*

- Suppose we want a shifter that right-shifts an N-bit input x by *s*, where N=32 and 0≤s≤31
- Naïve approach: Create 32 different fixed-size shifters and select using a mux



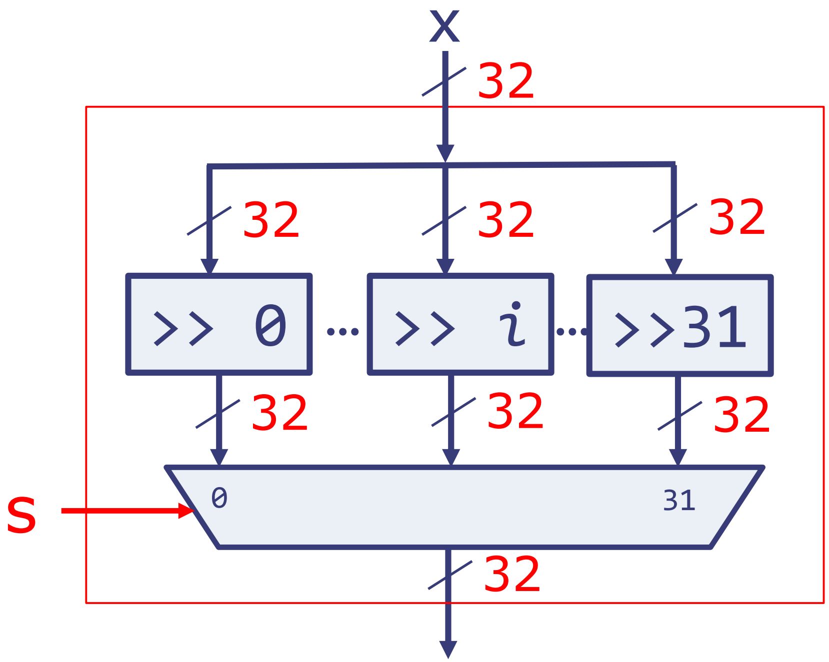
How many 2-way 1-bit muxes are needed to implement this 32-way 32-bit mux?

(32-1)*32 = 992

= ~4k gates

Logical Right Shift by *s*

- Suppose we want a shifter that right-shifts an N-bit input x by *s*, where N=32 and 0≤s≤31
- Naïve approach: Create 32 different fixed-size shifters and select using a mux



How many 2-way 1-bit muxes are needed to implement this 32-way 32-bit mux?

(32-1)*32 = 992

= ~4k gates

We can do better!

Barrel Shifter

An efficient circuit to perform variable-size shifts

- A barrel shifter performs shift by *s* using a series of fixed-size power-of-2 shifts

Barrel Shifter
An efficient circuit to perform variable-size shifts

- A barrel shifter performs shift by *s* using a series of fixed-size power-of-2 shifts
  - For example, shift by 5 (=4+1) can be done with shifts of sizes 4 and 1

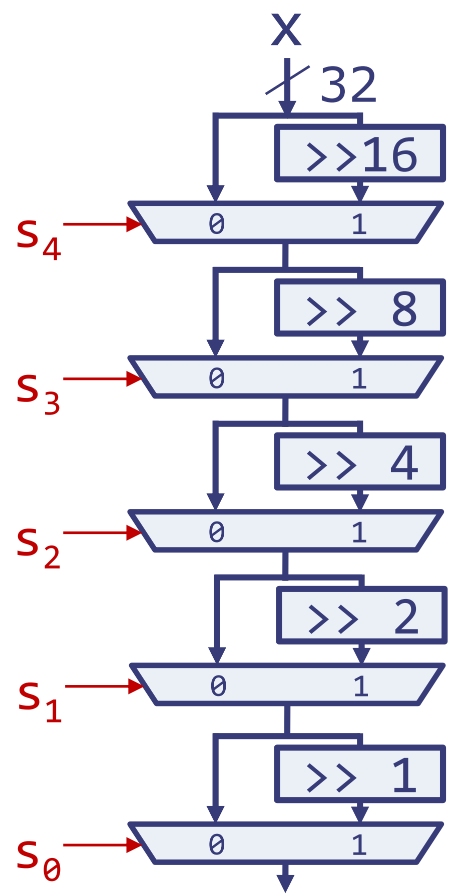
Barrel Shifter
An efficient circuit to perform variable-size shifts

- A barrel shifter performs shift by *s* using a series of fixed-size power-of-2 shifts
  - For example, shift by 5 (=4+1) can be done with shifts of sizes 4 and 1
  - The bit encoding of *s* tells us which shifts are needed: if the i^{th} bit of *n* is 1, then we need to shift by 2^i
    - Ex: 5 = 0b00101

Barrel Shifter

An efficient circuit to perform variable-size shifts

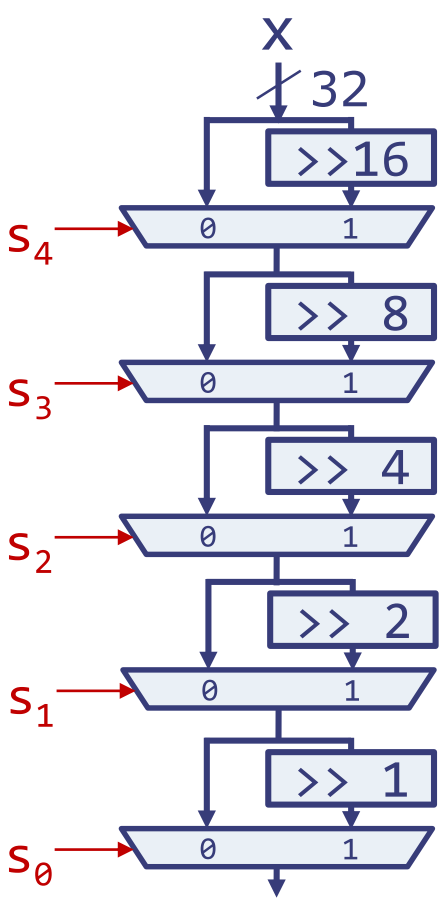
- A barrel shifter performs shift by *s* using a series of fixed-size power-of-2 shifts
  - For example, shift by 5 (=4+1) can be done with shifts of sizes 4 and 1
  - The bit encoding of *s* tells us which shifts are needed: if the i^{th} bit of *n* is 1, then we need to shift by 2^i
    - Ex: 5 = 0b00101
  - Implementation: A cascade of $log_2 N$ muxes that choose between shifting by 2^i and not shifting



Barrel Shifter
An efficient circuit to perform variable-size shifts

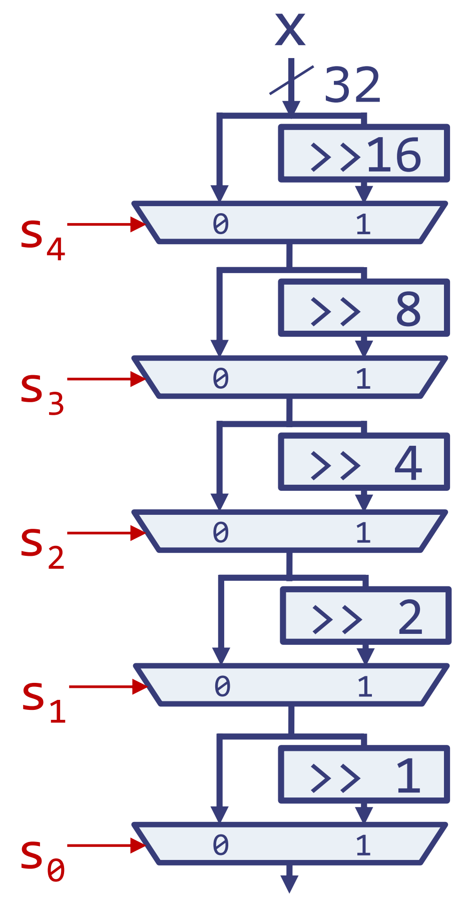
- A barrel shifter performs shift by *s* using a series of fixed-size power-of-2 shifts
  - For example, shift by 5 (=4+1) can be done with shifts of sizes 4 and 1
  - The bit encoding of *s* tells us which shifts are needed: if the i^{th} bit of *n* is 1, then we need to shift by 2^i
    - Ex: 5 = 0b00101
  - Implementation: A cascade of $log_2 N$ muxes that choose between shifting by 2^i and not shifting

How many 2-way 1-bit muxes?



Barrel Shifter
An efficient circuit to perform variable-size shifts

- A barrel shifter performs shift by *s* using a series of fixed-size power-of-2 shifts
  - For example, shift by 5 (=4+1) can be done with shifts of sizes 4 and 1
  - The bit encoding of *s* tells us which shifts are needed: if the i^{th} bit of *n* is 1, then we need to shift by 2^i
    - Ex: 5 = 0b00101
  - Implementation: A cascade of $log_2 N$ muxes that choose between shifting by 2^i and not shifting

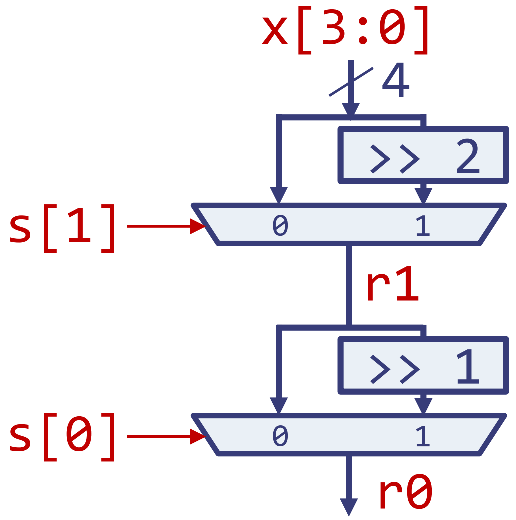


How many 2-way 1-bit muxes?

$N*log_2N = 32*5 = 160$

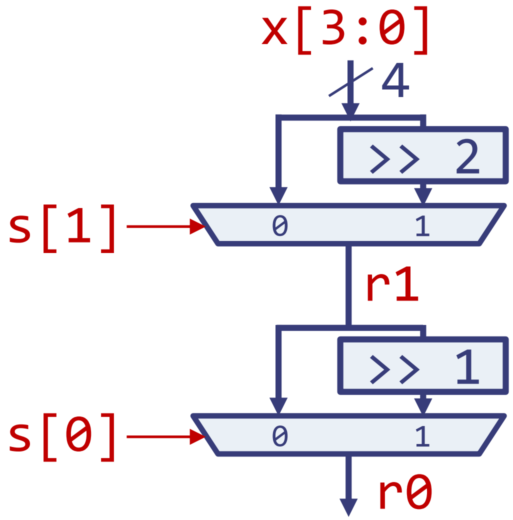
Barrel Shifter Implementation

- Example in Minispec for N=4
  - Only need 2 bits for s, why?



Barrel Shifter Implementation

- Example in Minispec for N=4
  - Only need 2 bits for s, why?

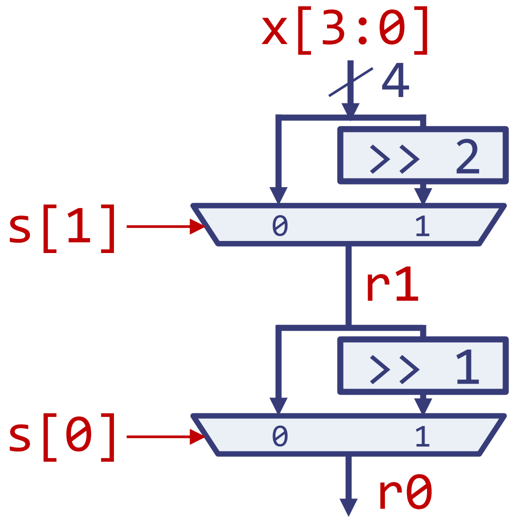



```
function Bit#(4) barrelShifter(Bit#(4) x, Bit#(2) s);


endfunction
```

Barrel Shifter Implementation

- Example in Minispec for N=4
  - Only need 2 bits for s, why?

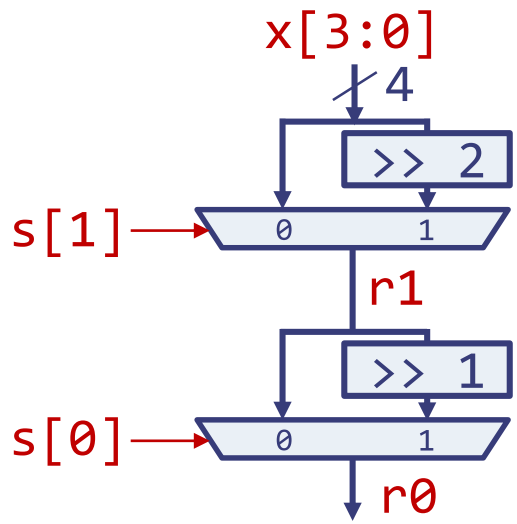
- Use conditional operator for 2-way muxes




```
function Bit#(4) barrelShifter(Bit#(4) x, Bit#(2) s);



endfunction
```

Barrel Shifter Implementation

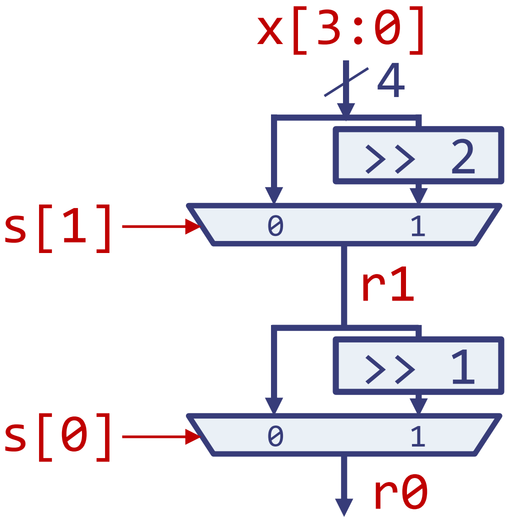
- Example in Minispec for N=4
  - Only need 2 bits for s, why?
- Use conditional operator for 2-way muxes
- Use concatenation and bit selection for fixed shifts




```
function Bit#(4) barrelShifter(Bit#(4) x, Bit#(2) s);



endfunction
```

Barrel Shifter Implementation

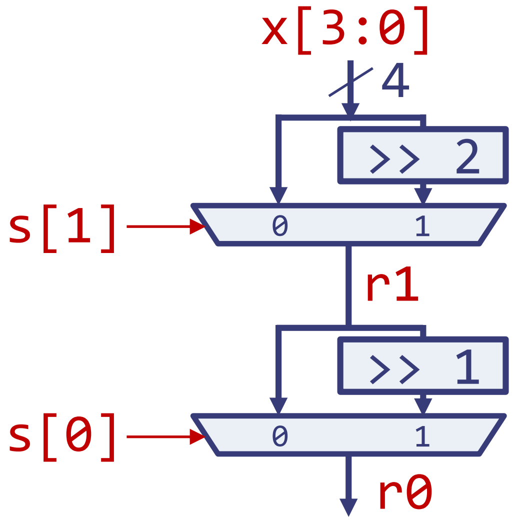
- Example in Minispec for N=4
  - Only need 2 bits for s, why?
- Use conditional operator for 2-way muxes
- Use concatenation and bit selection for fixed shifts




```
function Bit#(4) barrelShifter(Bit#(4) x, Bit#(2) s);
    Bit#(4) r1 = (s[1] == 0) ?  x : {2'b00, x[3:2]};


endfunction
```

Barrel Shifter Implementation

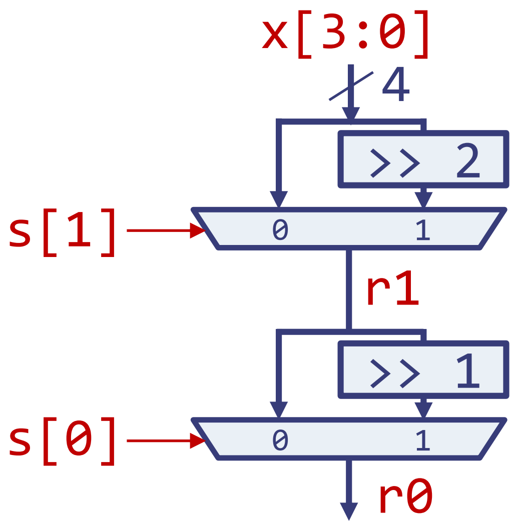
- Example in Minispec for N=4
  - Only need 2 bits for s, why?
- Use conditional operator for 2-way muxes
- Use concatenation and bit selection for fixed shifts




```
function Bit#(4) barrelShifter(Bit#(4) x, Bit#(2) s);
    Bit#(4) r1 = (s[1] == 0) ?  x : {2'b00, x[3:2]};
    Bit#(4) r0 = (s[0] == 0) ? r1 : {1'b0, r1[3:1]};

endfunction
```

Barrel Shifter Implementation

- Example in Minispec for N=4
  - Only need 2 bits for s, why?
- Use conditional operator for 2-way muxes
- Use concatenation and bit selection for fixed shifts




```
function Bit#(4) barrelShifter(Bit#(4) x, Bit#(2) s);
    Bit#(4) r1 = (s[1] == 0) ?  x : {2'b00, x[3:2]};
    Bit#(4) r0 = (s[0] == 0) ? r1 : {1'b0, r1[3:1]};
    return r0;
endfunction
```

Thank you!

Next lecture:
Complex combinational circuits
and advanced Minispec