Describing Combinational circuits in BSV

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.
Boolean Algebra and Arithmetic

- Numbers can be represented in binary (base 2) notation and arithmetic can be performed on binary numbers.
Boolean Algebra and Arithmetic

- Numbers can be represented in binary (base 2) notation and arithmetic can be performed on binary numbers.
- Binary arithmetic has one-to-one correspondence with Boolean algebra, i.e., operations on Boolean expressions.
### Boolean Algebra and Arithmetic

- Numbers can be represented in binary (base 2) notation and arithmetic can be performed on binary numbers
- Binary arithmetic has one-to-one correspondence with Boolean algebra, i.e., operations on Boolean expressions
- Thus, all arithmetic operations can be expressed as Boolean or combinational circuits!
Encoding Positive Integers

- Bit $i$ in a binary representation (in the right-to-left order) is assigned the weight $2^i$

\[
\begin{array}{cccccccccc}
2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0
\end{array}
\]
Encoding Positive Integers

- Bit $i$ in a binary representation (in the right-to-left order) is assigned the weight $2^i$

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{11}$</td>
<td>$2^{10}$</td>
<td>$2^9$</td>
<td>$2^8$</td>
<td>$2^7$</td>
<td>$2^6$</td>
<td>$2^5$</td>
<td>$2^4$</td>
<td>$2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Value of an N-bit number is given by the formula
Encoding Positive Integers

- Bit $i$ in a binary representation (in the right-to-left order) is assigned the weight $2^i$

```
  2^11 2^10 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
0  1  1  1  1  1  1  0  1  0  0  0  0
```

- Value of an $N$-bit number is given by the formula

$$v = \sum_{i=0}^{N-1} 2^i b_i$$
Encoding Positive Integers

- Bit $i$ in a binary representation (in the right-to-left order) is assigned the weight $2^i$

- Value of an $N$-bit number is given by the formula

\[ v = \sum_{i=0}^{N-1} 2^i b_i \]

- What value does 011111010000 encode?
Encoding Positive Integers

- Bit $i$ in a binary representation (in the right-to-left order) is assigned the weight $2^i$

\[
\begin{array}{cccccccccccc}
2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
\]

- Value of an N-bit number is given by the formula

\[v = \sum_{i=0}^{N-1} 2^i b_i\]

- What value does 011111010000 encode?

\[V = 0 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + \ldots\]
Encoding Positive Integers

- Bit $i$ in a binary representation (in the right-to-left order) is assigned the weight $2^i$

<table>
<thead>
<tr>
<th>$2^{11}$</th>
<th>$2^{10}$</th>
<th>$2^9$</th>
<th>$2^8$</th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Value of an N-bit number is given by the formula

$$v = \sum_{i=0}^{N-1} 2^i b_i$$

- What value does 011111010000 encode?

$$V = 0 \cdot 2^{11} + 1 \cdot 2^{10} + 1 \cdot 2^9 + \ldots$$

$$= 1024 + 512 + 256 + 128 + 64 + 16 = 2000$$
Encoding Positive Integers

- Bit \( i \) in a binary representation (in the right-to-left order) is assigned the weight \( 2^i \)

\[
\begin{array}{cccccccccccc}
2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0
\end{array}
\]

- Value of an \( N \)-bit number is given by the formula

\[
v = \sum_{i=0}^{N-1} 2^i b_i
\]

- What value does 011111010000 encode?

\[
V = 0 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + \ldots
\]

\[
= 1024 + 512 + 256 + 128 + 64 + 16 = 2000
\]

Smallest number? Largest number?
Encoding Positive Integers

- Bit $i$ in a binary representation (in the right-to-left order) is assigned the weight $2^i$

<table>
<thead>
<tr>
<th>2^{11}</th>
<th>2^{10}</th>
<th>2^9</th>
<th>2^8</th>
<th>2^7</th>
<th>2^6</th>
<th>2^5</th>
<th>2^4</th>
<th>2^3</th>
<th>2^2</th>
<th>2^1</th>
<th>2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Value of an N-bit number is given by the formula $v = \sum_{i=0}^{N-1} 2^i b_i$

- What value does 011111010000 encode?

\[
V = 0 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + \ldots \\
= 1024 + 512 + 256 + 128 + 64 + 16 = 2000
\]

Smallest number? 0  Largest number?
Encoding Positive Integers

- Bit $i$ in a binary representation (in the right-to-left order) is assigned the weight $2^i$

<table>
<thead>
<tr>
<th>$2^{11}$</th>
<th>$2^{10}$</th>
<th>$2^9$</th>
<th>$2^8$</th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Value of an $N$-bit number is given by the formula

  $$V = 0 \cdot 2^{11} + 1 \cdot 2^{10} + 1 \cdot 2^9 + \ldots$$
  $$= 1024 + 512 + 256 + 128 + 64 + 16 = 2000$$

- **What value does 011111010000 encode?**

  $$v = \sum_{i=0}^{N-1} 2^i b_i$$

- Smallest number? 0  
  - Largest number? $2^{N-1}$
Binary Addition

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

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

Base 10

\[
\begin{array}{c}
14 \\
+ 7 \\
\hline
\end{array}
\]
Binary Addition

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

Base 10

\[
\begin{array}{c}
1 \\
14 \\
+ 7 \\
\hline
1 \\
\end{array}
\]
Binary Addition

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

Base 10

\[
\begin{array}{c}
1 \\
14 \\
+ \quad 7 \\
\hline
1
\end{array}
\]

\textit{carry}
Binary Addition

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

Base 10

```
14
+  7
---
21  (carry 1)
```
Binary Addition

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

```
<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 + 7</td>
<td>1110 + 111</td>
</tr>
<tr>
<td>21</td>
<td>1111</td>
</tr>
</tbody>
</table>
```
Binary Addition

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

Base 10

\[
\begin{align*}
14 + 7 & = 21 \\
\text{carry} 1 & = 0
\end{align*}
\]

Base 2

\[
\begin{align*}
1110 + 111 & = 10001
\end{align*}
\]
### Binary Addition

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

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>01</td>
</tr>
</tbody>
</table>

*1 carry*
Binary Addition

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

Base 10

14 + 7 = 21

110 + 111 = 101

Base 2
Binary Addition

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

Base 10

14
+ 7
---
21

Base 2

1110
+ 111
---
10101
Binary Addition

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

Base 10  |  Base 2
---|---
14  |  1110
+ 7  |  + 111
---|---
21 | 10101

1 carry
Binary Addition

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

Base 10  |  Base 2
---|---
14  | 1110
+ 7  | 1110
---|---
21  | 10101

Let us build a hardware adder
Combinational Logic for an adder
Combinational Logic for an adder

1. Half adder (HA): adds two 1-bit numbers and produces a sum and a carry bit
Combinational Logic for an adder

1. Half adder (HA): adds two 1-bit numbers and produces a sum and a carry bit
2. Full adder (FA): adds two one-bit numbers and a carry, and produces a sum bit and a carry bit
Combinational Logic for an adder

1. Half adder (HA): adds two 1-bit numbers and produces a sum and a carry bit
2. Full adder (FA): adds two one-bit numbers and a carry, and produces a sum bit and a carry bit
   - Can be built using two HAs
Combinational Logic for an adder

1. Half adder (HA): adds two 1-bit numbers and produces a sum and a carry bit
2. Full adder (FA): adds two one-bit numbers and a carry, and produces a sum bit and a carry bit
   Can be built using two HAs
3. Cascade FAs to perform binary addition
Combinational Logic for an adder

1. Half adder (HA): adds two 1-bit numbers and produces a sum and a carry bit
2. Full adder (FA): adds two one-bit numbers and a carry, and produces a sum bit and a carry bit
   Can be built using two HAs
3. Cascade FAs to perform binary addition
Describing a 32-bit adder
alternatives

- Truth Table with $2^{32}$ rows and two columns (sum and carry)!
Describing a 32-bit adder alternatives

- Truth Table with $2^{32}$ rows and two columns (sum and carry)!
- 32 sets of Boolean equations, where each set describes a FA
Describing a 32-bit adder alternatives

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

\[
\begin{align*}
0 \leq k \leq 31
\end{align*}
\]
Describing a 32-bit adder
alternatives

- Truth Table with $2^{32}$ rows and two columns (sum and carry)!
- 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
Describing a 32-bit adder alternatives

- Truth Table with $2^{32}$ rows and two columns (sum and carry)!
- 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.
Describing a 32-bit adder
alternatives

- Truth Table with $2^{32}$ rows and two columns (sum and carry)!
- 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 BSV

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean equations

\[ s = \sim a \cdot b + a.(\sim b) = a \oplus b \]
\[ c = a \cdot b \]
Half Adder in BSV

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean equations

\[ s = \sim a \cdot b + a.(\sim b) \]
\[ = a \oplus b \]
\[ c = a \cdot b \]
## Half Adder in BSV

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean equations:

\[
s = \sim a \cdot b + a.(\sim b) = a \oplus b \\
c = a \cdot b
\]

```python
function ha(a, b);
    s = a ^ b;
    c = a & b;
    return {c,s};
endfunction
```
Half Adder in BSV

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean equations

\[
\begin{align*}
  s &= \neg a \cdot b + a \cdot (\neg b) \\
  c &= a \cdot b 
\end{align*}
\]

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

XOR
Half Adder in BSV

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean equations

\[ s = \neg a \cdot b + a \cdot (\neg b) \]
\[ = a \oplus b \]
\[ c = a \cdot b \]

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

XOR

AND

September 13, 2018
# Half Adder in BSV

## Boolean equations

\[ s = \sim a \cdot b + a.(\sim b) \]
\[ c = a \oplus b \]

## Function

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

- **XOR**
- **AND**

---

Not quite correct - needs type annotations
Half Adder *corrected*

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

- "Bit#(1) a" type declaration says that a is one bit wide
```
Half Adder corrected

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

- "Bit #(1) a" type declaration says that a is one bit wide
- {c,s} represents bit concatenation
Half Adder corrected

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

- “Bit#(1) a” type declaration says that a is one bit wide
- {c,s} represents bit concatenation
- How big is {c,s}?
Half Adder *corrected*

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

- "Bit#(1) a" type declaration says that a is one bit wide
- \( \{c,s\} \) represents bit concatenation
- How big is \( \{c,s\} \)?
  
  2 bits
function Bit#(2) ha(Bit#(1) a, Bit#(1) b);
   Bit#(1) s = a ^ b;
   Bit#(1) c = a & b;
   return {c, s};
endfunction
function Bit#(2) ha(Bit#(1) a, Bit#(1) b);
    Bit#(1) s = a ^ b;
    Bit#(1) c = a & b;
    return {c, s};
endfunction

ha can be used as a black-box as long as we understand its type signature
Suppose we write \( t = h(a, b) \) then \( t \) is a two bit quantity representing \( s \) and \( c \) values.

\[
\begin{align*}
\text{function } & \quad \text{Bit\#(2) } h(a, b); \\
& \quad \text{Bit\#(1) } s = a \text{^ b}; \\
& \quad \text{Bit\#(1) } c = a \text{& b}; \\
& \quad \text{return } \{c, s\}; \\
\end{align*}
\]

\( h(a) \) can be used as a black-box as long as we understand its type signature.
Suppose we write $t = ha(a,b)$ then $t$ is a two bit quantity representing $s$ and $c$ values.

- We can recover $s$ and $c$ values from $t$ by writing $t[0]$ and $t[1]$, respectively.
Full Adder using HAs

\[
\begin{align*}
fa & \quad \text{ha} \quad \text{ha} \\
\text{a} & \quad \text{b} \\
\text{c\_in} & \quad \text{c\_out} \\
\end{align*}
\]
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
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
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
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

ha is being used as a black-box; fa code is simply a wiring diagram
The "let" syntax
The “let” syntax

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

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

No need to write the type if the compiler can deduce it
```
2-bit Ripple-Carry Adder
cascading full adders

function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
2-bit Ripple-Carry Adder
cascading full adders

Use \texttt{fa} as a black-box

\begin{verbatim}
function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
\end{verbatim}
2-bit Ripple-Carry Adder
cascading full adders

Use `fa` as a black-box

```haskell
function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
    Bit#(2) s = 0;
    Bit#(3) c = 0;
    c[0] = 0;
```

- `y[1]` → `c[1]` → `s[0]` from first `fa`
- `x[0]` → `c[0]` to second `fa`
- `y[0]` → `c[0]` to second `fa`
2-bit Ripple-Carry Adder
cascading full adders

function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
Bit#(2) s = 0;
Bit#(3) c = 0;
c[0] = 0;

Use fa as a black-box

s has two wires, initially each s wire is zero
2-bit Ripple-Carry Adder
cascading full adders

function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
Bit#(2) s = 0; Bit#(3) c = 0;
c[0] = 0;
let cs0 = fa(x[0], y[0], c[0]);
s[0] = cs0[0]; c[1] = cs0[1];

Use fa as a black-box
function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
    Bit#(2) s = 0;
    Bit#(3) c = 0;
    c[0] = 0;
    let cs0 = fa(x[0], y[0], c[0]);
    s[0] = cs0[0];
    c[1] = cs0[1];

Use fa as a black-box
2-bit Ripple-Carry Adder

cascading full adders

function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
    Bit#(2) s = 0;
    Bit#(3) c = 0;
    c[0] = 0;
    let cs0 = fa(x[0], y[0], c[0]);
    s[0] = cs0[0];
    c[1] = cs0[1];
    let cs1 = fa(x[1], y[1], c[1]);
    s[1] = cs1[0];
    c[2] = cs1[1];

Use fa as a black-box
2-bit Ripple-Carry Adder

cascading full adders

function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
    Bit#(2) s = 0; Bit#(3) c = 0;
    c[0] = 0;
    let cs0 = fa(x[0], y[0], c[0]);
    s[0] = cs0[0]; c[1] = cs0[1];
    let cs1 = fa(x[1], y[1], c[1]);
    s[1] = cs1[0]; c[2] = cs1[1];

Use fa as a black-box

wire s[1] is updated
2-bit Ripple-Carry Adder
cascading full adders

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

Use `fa` as a black-box
Assigning to Multi-Bit Words

- Means $c$ is three bits wide and each element is set to zero

```plaintext
Bit#(3) c = 0;
```
Assigning to Multi-Bit Words

- Means $c$ is three bits wide and each element is set to zero

```haskell
Bit#(3) c = 0;
c[0] = c0;
```
Assigning to Multi-Bit Words

- Means $c$ is three bits wide and each element is set to zero

```plaintext
Bit#(3) c = 0;
c[0] = c0;
```

- $c0$ is assigned to element 0 of $c$, but the values of the rest of the elements are not affected
Assigning to Multi-Bit Words

Bit#(3) c = 0;

- Means c is three bits wide and each element is set to zero

- c0 is assigned to element 0 of c, but the values of the rest of the elements are not affected

- Each bit in a multi-bit word must have an initial value
Assigning to Multi-Bit Words

*Bit#(3) c = 0;*

- Means `c` is three bits wide and each element is set to zero

  `c[0] = c0;`

- `c0` is assigned to element 0 of `c`, but the values of the rest of the elements are not affected

- Each bit in a multi-bit word must have an initial value
  - An attempt to use uninitialized bits will raise a compiler warning and result in unexpected behavior
Selectors and Multiplexers
Selecting a wire: $x[i]$

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

  - We can also select multiple bits: e.g.,
    $x[2:1]$ means $\{x[2], x[1]\}$

  - Assume $x$ is 4 bits wide.
Selecting a wire: \( x[i] \)

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

- We can also select multiple bits: e.g., \( x[2:1] \) means \( \{x[2], x[1]\} \)

Assume \( x \) is 4 bits wide.
Selecting a wire: \( x[i] \)

- Constant selector: e.g., \( x[2] \)
- We can also select multiple bits: e.g., \( x[2:1] \) means \( \{x[2], x[1]\} \)

Assume \( x \) is 4 bits wide.
Selecting a wire: $x[i]$

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

  ![Constant selector diagram]

  - Assume $x$ is 4 bits wide
  - $x[2]$ is just the name of a wire

- **Dynamic selector:** $x[i]$

  ![Dynamic selector diagram]

  - We can also select multiple bits: e.g., $x[2:1]$ means \{x[2], x[1]\}
Selecting a wire: \( x[i] \)

- **Constant selector**: e.g., \( x[2] \)

  - We can also select multiple bits: e.g., \( x[2:1] \) means \( \{x[2], x[1]\} \)

- **Dynamic selector**: \( x[i] \)

  - Assume \( x \) is 4 bits wide

  - No hardware; \( x[2] \) is just the name of a wire
Selecting a wire: \( x[i] \)

- **Constant selector:** e.g., \( x[2] \)
  
  Assume \( x \) is 4 bits wide

- **We can also select multiple bits:** e.g., \( x[2:1] \) means \( \{x[2], x[1]\} \)

- **Dynamic selector:** \( x[i] \)
  
  4-way mux
A 2-way multiplexer

A mux is a simple conditional expression
A 2-way multiplexer

A mux is a simple conditional expression

BSV: \((s==0)\)? a : b ;
A 2-way multiplexer

A mux is a simple conditional expression

BSV  \[(s==0)\)? a : b ;\]

Python  a if s == 0 else b
A 2-way multiplexer

A mux is a simple conditional expression

BSV: (s==0)? a : b ;

Python: a if s == 0 else b

Gate-level implementation
A 2-way multiplexer

A mux is a simple conditional expression

BSV: \((s==0)\)? a : b ;
Python: a if s == 0 else b

Gate-level implementation

If \(a\) and \(b\) are \(n\)-bit wide then this structure is replicated \(n\) times; \(s\) is the same input for all the replicated structures
A 4-way multiplexer

case \{s_1, s_0\}
  0: a;
  1: b;
  2: c;
  3: d;
endcase
A 4-way multiplexer

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

```plaintext
case ({s1,s0})
    0:  a;
    1:  b;
    2:  c;
    3:  d;
endcase
```
A 4-way multiplexer

case ({s1,s0})
  0:  a;
  1:  b;
  2:  c;
  3:  d;
endcase

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

n-way mux can be implemented using n-1 two-way muxes
Shift operators
Logical right shift by 2

- Fixed size shift operation is cheap in hardware – just wire the circuit appropriately.
Logical right shift by 2

- Fixed size shift operation is cheap in hardware – just wire the circuit appropriately
- Rotates and arithmetic shifts are similar
Logical right shift by 2

- Fixed size shift operation is cheap in hardware – just wire the circuit appropriately
- Rotates and arithmetic shifts are similar
Logical right shift by 2

- Fixed size shift operation is cheap in hardware – just wire the circuit appropriately
- Rotates and arithmetic shifts are similar
Logical right shift by 2

- Fixed size shift operation is cheap in hardware – just wire the circuit appropriately
- Rotates and arithmetic shifts are similar

Rotate

Arithmetic

useful for multiplication and division by $2^n$
Logical right shift by $n$

- Suppose we want to build a shifter which shifts a value $x$ by $n$ where $n$ is between 0 and 31.
Logical right shift by $n$

- Suppose we want to build a shifter which shifts a value $x$ by $n$ where $n$ is between 0 and 31.
- One way to do this is by connecting 31 different shifters via a mux.
Suppose we want to build a shifter which shifts a value $x$ by $n$ where $n$ is between 0 and 31. One way to do this is by connecting 31 different shifters via a mux.
Logical right shift by \( n \)

- Suppose we want to build a shifter which shifts a value \( x \) by \( n \) where \( n \) is between 0 and 31
- One way to do this is by connecting 31 different shifters via a mux

How many 2-way one-bit muxes are needed to implement this structure?
Logical right shift by $n$

- Suppose we want to build a shifter which shifts a value $x$ by $n$ where $n$ is between 0 and 31
- One way to do this is by connecting 31 different shifters via a mux

How many 2-way one-bit muxes are needed to implement this structure?

$$n \times (n - 1)$$
Logical right shift by $n$

- Suppose we want to build a shifter which shifts a value $x$ by $n$ where $n$ is between 0 and 31.
- One way to do this is by connecting 31 different shifters via a mux.

How many 2-way one-bit muxes are needed to implement this structure?

$$n \times (n - 1)$$

Can we do better?
Logical right shift by $n$

- Shift $n$ can be broken down into $\log n$ steps of fixed-length shifts of size 1, 2, 4, ...
Logical right shift by $n$

- Shift $n$ can be broken down into $\log n$ steps of fixed-length shifts of size 1, 2, 4, ...
  - For example, we can perform Shift 3 ($=2+1$) by doing shifts of size 2 and 1
Logical right shift by $n$

- Shift $n$ can be broken down into $\log n$ steps of fixed-length shifts of size 1, 2, 4, ...
  - For example, we can perform Shift 3 ($=2+1$) by doing shifts of size 2 and 1
  - Shift 5 ($=4+1$) by doing shifts of sizes
Logical right shift by $n$

- Shift $n$ can be broken down into $\log n$ steps of fixed-length shifts of size $1, 2, 4, \ldots$
  - For example, we can perform Shift 3 ($= 2 + 1$) by doing shifts of size 2 and 1
  - Shift 5 ($= 4 + 1$) by doing shifts of sizes 4 and 1
Logical right shift by $n$

- Shift $n$ can be broken down into $\log n$ steps of fixed-length shifts of size 1, 2, 4, ...
  - For example, we can perform Shift 3 ($=2+1$) by doing shifts of size 2 and 1
  - Shift 5 ($=4+1$) by doing shifts of sizes 4 and 1
  - Shift 21 ($=16+4+1$) by doing shifts of sizes 16, 4, and 1
Logical right shift by $n$

- Shift $n$ can be broken down into log $n$ steps of fixed-length shifts of size 1, 2, 4, ...
  - For example, we can perform Shift 3 ($=2+1$) by doing shifts of size 2 and 1
  - Shift 5 ($=4+1$) by doing shifts of sizes 4 and 1
  - Shift 21 ($=16+4+1$) by doing shifts of sizes 16, 4 and 1
Logical right shift by $n$

- Shift $n$ can be broken down into log $n$ steps of fixed-length shifts of size 1, 2, 4, ...
  - For example, we can perform Shift 3 (=2+1) by doing shifts of size 2 and 1
  - Shift 5 (=4+1) by doing shifts of sizes 4 and 1
  - Shift 21 (=16+4+1) by doing shifts of sizes 16, 4 and 1
- For a 32-bit number, a 5-bit $n$ can specify all the needed shifts
  - $3_{10} = 00011_2$, $5_{10} = 00101_2$, $21_{10} = 10101_2$
Logical right shift by $n$

- Shift $n$ can be broken down into log $n$ steps of fixed-length shifts of size 1, 2, 4, ...
  - For example, we can perform Shift 3 ($=2+1$) by doing shifts of size 2 and 1
  - Shift 5 ($=4+1$) by doing shifts of sizes 4 and 1
  - Shift 21 ($=16+4+1$) by doing shifts of sizes 16, 4 and 1
- For a 32-bit number, a 5-bit $n$ can specify all the needed shifts
  - $3_{10} = 00011_2$, $5_{10} = 00101_2$, $21_{10} = 10101_2$
- The bit encoding of $n$ tells us which shifters are needed; if the value of the $i^{th}$ (least significant) bit is 1 then we need to shift by $2^i$ bits
Conditional operation: shift versus no-shift

- We need a mux to select the appropriate wires: if $s$ is one the mux will select the wires on the left otherwise it would select wires on the right.
Conditional operation: shift versus no-shift

- We need a mux to select the appropriate wires: if \( s \) is one the mux will select the wires on the left otherwise it would select wires on the right

```vhdl
(s==0)?{a,b,c,d}:{2'b0,a,b};
```
Logical right shift circuit

- Define \( \log n \) shifters of sizes 1, 2, 4, ...
Logical right shift circuit

- Define log $n$ shifters of sizes 1, 2, 4, ...
- Define log $n$ muxes to perform a particular size shift
Logical right shift circuit

- Define \( \log n \) shifters of sizes 1, 2, 4, ...
- Define \( \log n \) muxes to perform a particular size shift
- Suppose \( s = \{s_1, s_0\} \) is a two bit number. Shift circuit to shift a number by \( s \) can be expressed as two nested conditionals expressions
Lecture L04: Binary Arithmetic
Lecture L05: Complex combinational circuits in Bluespec