Describing Combinational circuits in BSV

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
Three simple combinational circuits

\[ a \rightarrow s = \sim a \]

<table>
<thead>
<tr>
<th>a</th>
<th>( s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Three simple combinational circuits

- **NOT**
  - Table:
    | a | s |
    |---|---|
    | 0 | 1 |
    | 1 | 0 |
  - Equation: \( s = \sim a \)

- **AND**
Three simple combinational circuits

**NOT**

<table>
<thead>
<tr>
<th>a</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s = \sim a \]

**AND**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ s = a \cdot b \]
Three simple combinational circuits

- **NOT**
  - a \[\begin{array}{c|c}
  a & s \\
  \hline
  0 & 1 \\
  1 & 0 \\
  \end{array}
  \]
  - \(s = \sim a\)

- **AND**
  - a \[\begin{array}{c|c|c}
    a & b & s \\
    \hline
    0 & 0 & 0 \\
    0 & 1 & 0 \\
    1 & 0 & 0 \\
    1 & 1 & 1 \\
  \end{array}
  \]
  - \(s = a \cdot b\)

- **OR**

February 13, 2018


L03-2
Three simple combinational circuits

- **NOT**
  
<table>
<thead>
<tr>
<th>a</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **AND**
  
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **OR**
  
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ s = \sim a \]
\[ s = a \cdot b \]
\[ s = a + b \]
Three simple combinational circuits

**NOT**

<table>
<thead>
<tr>
<th>a</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**AND**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**OR**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Any combinational circuit can be built using these three gates.
Some other famous gates

**NAND**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s = \neg(a \cdot b) \]

**NOR**
Some other famous gates

**NAND**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s = \neg(a \cdot b) \]

**NOR**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s = \neg(a + b) \]
Some other famous gates

**NAND**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s = \sim(a \cdot b) \]

**NOR**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s = \sim(a + b) \]

Can you express these gates using NOT, AND, and OR gates?
Some other famous gates

**NAND**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s = \sim(a \cdot b) \]

**NOR**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s = \sim(a + b) \]

Can you express these gates using NOT, AND, and OR gates?
Some other famous gates

**NAND**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$s = \neg(a \cdot b)$

**NOR**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$s = \neg(a + b)$

Can you express these gates using NOT, AND, and OR gates?
Exclusive OR (XOR): another famous gate

\[ s = a \oplus b \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Exclusive OR (XOR): another famous gate

Can you express XOR using NOT, AND, and OR gates?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s = a \oplus b \]
**Exclusive OR (XOR): another famous gate**

From the Truth Table XOR produces a 1 when either \((a=0) \text{ AND } (b=1)\) or \((a=1) \text{ AND } (b=0)\). Hence, \(a \oplus b = \neg a \cdot b + a \cdot \neg b\)

Can you express XOR using NOT, AND, and OR gates?
Exclusive OR (XOR): another famous gate

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

Can you express XOR using NOT, AND, and OR gates?

\[
\begin{array}{c|c|c}
   a & b & s \\
   
   0 & 0 & 0 \\
   0 & 1 & 1 \\
   1 & 0 & 1 \\
   1 & 1 & 0 \\
\end{array}
\]
Pictorial conventions for drawing inverters
Pictorial conventions for drawing inverters

[Diagram of two inverters]
Pictorial conventions for drawing inverters
Pictorial conventions for drawing inverters

All these represent the same circuit
We use the words in each of the following categories interchangeably:

- combinational circuits, Boolean expressions, Boolean circuits
- gate, Boolean operator
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
Describing Complex Combinational circuits

A combinational circuit with \( n \) input variables and \( m \) outputs has \( 2^n \) rows and \( m \) columns in its Truth Table representation.
Describing Complex Combinational circuits

A combinational circuit with \( n \) input variables and \( m \) outputs has \( 2^n \) rows and \( m \) columns in its Truth Table representation.

- Truth Tables are not a practical representation for circuits with large number of inputs.
Describing Complex Combinational circuits

A combinational circuit with \( n \) input variables and \( m \) outputs has \( 2^n \) rows and \( m \) columns in its Truth Table representation.

- Truth Tables are not a practical representation for circuits with large number of inputs.
- Circuit diagrams are even more tedious to draw.
Describing Complex Combinational circuits

- A combinational circuit with \( n \) input variables and \( m \) outputs has \( 2^n \) rows and \( m \) columns in its Truth Table representation.
  - Truth Tables are not a practical representation for circuits with large number of inputs.
  - Circuit diagrams are even more tedious to draw.
  - Both representations are useless when we want computers to simulate the behavior of a circuit, i.e., determine the output given an input.
A combinational circuit with \( n \) input variables and \( m \) outputs has \( 2^n \) rows and \( m \) columns in its Truth Table representation.

- Truth Tables are not a practical representation for circuits with large number of inputs.
- Circuit diagrams are even more tedious to draw.
- Both representations are useless when we want computers to simulate the behavior of a 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

<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 = a \oplus b
\[
c = a \cdot b
\]
# Half Adder

<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>

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

Boolean equations

\[ s = a \oplus b \]
\[ c = a \cdot b \]
Half Adder

<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>

Function: \( ha(a, b); \)

\[
\begin{align*}
    s &= a \oplus b; \\
    c &= a \& b; \\
    \text{return} \ & \{c, s\}; \\
\end{align*}
\]

Boolean equations:

\[
\begin{align*}
    s &= a \oplus b \\
    c &= a \cdot b \\
\end{align*}
\]

February 13, 2018

Half Adder

<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

\[
\begin{align*}
  s &= a \oplus b \\
  c &= a \cdot b
\end{align*}
\]

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

February 13, 2018
Half Adder

<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

\[
s = a \oplus b \\
c = a \cdot b
\]

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

Not quite correct – needs type annotations
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
Half Adder \textit{corrected}

\begin{verbatim}
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
\end{verbatim}
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 \textit{corrected}

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

“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
BSV notes

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 \( t = ha(a,b) \) then \( t \) is a two bit quantity representing \( c \) and \( s \) values.

```verilog
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 \( t = ha(a, b) \) then \( t \) is a two bit quantity representing \( c \) and \( s \) values. We can recover \( c \) and \( s \) values from \( t \) by writing \( t[1] \) and \( t[0] \), respectively.

```verilog
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.
Full Adder
1-bit adder with a carry-in input
Full Adder
1-bit adder with a carry-in input

```
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
```
Full Adder
1-bit adder with a carry-in input

```
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
Full Adder
1-bit adder with a carry-in input

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
Full Adder
1-bit adder with a carry-in input

```
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.

Extracts the sum bit
Extracts the carry bit
The “let” syntax
The "let" syntax

```haskell
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

```verilog
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.
Types

A type is a grouping of values:
- Integer: 1, 2, 3, ... Bool: True, False  Bit: 0,1

More complex types can be defined in terms of simpler types
- Tuple2#(Integer, Integer) represents a pair of Integers
- function Integer fname (Integer arg) represents a function from Integers to Integers and is named fname
Types

A type is a grouping of values:

- Integer: 1, 2, 3, ... Bool: True, False  Bit: 0, 1

More complex types can be defined in terms of simpler types:

- Tuple2#(Integer, Integer) represents a pair of Integers
- function Integer fname (Integer arg) represents a function from Integers to Integers and is named fname

Every expression in a BSV program has a type; sometimes it is specified explicitly and sometimes it is deduced by the compiler.
Types

A type is a grouping of values:

- **Integer:** 1, 2, 3, … **Bool:** True, False  **Bit:** 0,1

More complex types can be defined in terms of simpler types

- **Tuple2#(Integer, Integer)** represents a pair of Integers

**function** Integer **fname** (Integer **arg**) represents a function from Integers to Integers and is named **fname**

Every expression in a BSV program has a type; sometimes it is specified explicitly and sometimes it is deduced by the compiler

Thus, we say an expression has a type or belongs to a type
Types

A type is a grouping of values:
- Integer: 1, 2, 3, …
- Bool: True, False
- Bit: 0, 1

More complex types can be defined in terms of simpler types
- Tuple2#(Integer, Integer) represents a pair of Integers
- function Integer \( \text{fname} \) (Integer arg) represents a function from Integers to Integers and is named \( \text{fname} \)

Every expression in a BSV program has a type; sometimes it is specified explicitly and sometimes it is deduced by the compiler.

Thus, we say an expression has a type or belongs to a type.

An expression has exactly one type
Parameterized types: #

A type declaration itself can be parameterized by other types.

Parameters are indicated by using the syntax `#`

- For example `Bit#(n)` represents n bits and can be instantiated by specifying a value of n. `Bit#(1), Bit#(32), Bit#(8), ...`
Type synonyms

typedef Bit#(8) Byte;
Type synonyms

typedef Bit#(8) Byte;
typedef Bit#(32) Word;
Type synonyms

typedef Bit#(8) Byte;
typedef Bit#(32) Word;
typedef Bit#(32) Data;
Type synonyms

```plaintext
typedef Bit#(8) Byte;
typedef Bit#(32) Word;
typedef Bit#(32) Data;
typedef Tuple2#(a,a) Pair#(type a);
```
Type synonyms

typedef Bit#(8) Byte;
typedef Bit#(32) Word;
typedef Bit#(32) Data;
typedef Tuple2#(a,a) Pair#(type a);
typedef Int#(n) MyInt#(numeric type n);
Type declaration versus deduction

- The programmer writes down types of some expressions in a program and the compiler infers the types of the rest of expressions.
Type declaration versus deduction

- The programmer writes down types of some expressions in a program and the compiler infers the types of the rest of expressions.
- If the type inference cannot be performed or the type declarations are inconsistent then the compiler complains.
Type declaration versus deduction

- The programmer writes down types of some expressions in a program and the compiler infers the types of the rest of expressions.
- If the type inference cannot be performed or the type declarations are inconsistent then the compiler complains.

```plaintext
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#(2) c_out = ab[1] | abc[1];
    return {c_out, abc[0]};
endfunction
```
Type declaration versus deduction

- The programmer writes down types of some expressions in a program and the compiler infers the types of the rest of expressions.
- If the type inference cannot be performed or the type declarations are inconsistent then the compiler complains.

```plaintext
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#(2) c_out = ab[1] | abc[1];
    return {c_out, abc[0]};
endfunction
```

February 13, 2018
L03-16
Type declaration versus deduction

- The programmer writes down types of some expressions in a program and the compiler infers the types of the rest of expressions.
- If the type inference cannot be performed or the type declarations are inconsistent then the compiler complains.

```plaintext
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#(2) c_out = ab[1] | abc[1];
    return {c_out, abc[0]};
endfunction
```

Type checking prevents lots of silly mistakes.
Selectors and Multiplexers
Selecting a wire: $x[i]$

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

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

Constant Selector: e.g., $x[2]$

Assume $x$ is 4 bits wide
Selecting a wire: $x[i]$

Constant Selector: e.g., $x[2]$

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., $x[2]$  
  
- **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., \( x[2] \)

  - \( x_0 \)
  - \( x_1 \)
  - \( x_2 \)
  - \( x_3 \)

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

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

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

- **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 simple conditional expression
A 2-way multiplexer

A mux is simple conditional expression

Gate-level implementation
A 2-way multiplexer

A mux is simple conditional expression

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

Gate-level implementation
A 2-way multiplexer

A mux is simple conditional expression

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

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

A mux is 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 will be replicated `n` times
A 4-way multiplexer

\[
\text{case } \{s1, s0\} \text{ matches}
\]
\[
\begin{align*}
0 & : a; \\
1 & : b; \\
2 & : c; \\
3 & : d;
\end{align*}
\]
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
```
A 4-way multiplexer

case \( \{s_1, s_0\} \) matches
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

Other types of shifts are similar
Logical right shift by 2

Fixed size shift operation is cheap in hardware – just wire the circuit appropriately.

Other types of shifts are similar.
Logical right shift by 2

- Fixed size shift operation is cheap in hardware – just wire the circuit appropriately
- Other types of shifts are similar

Rotate

Arithmetic
Logical right shift by 2

Fixed size shift operation is cheap in hardware – just wire the circuit appropriately

Other types of 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.
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.
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 size
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 size 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 size 4 and 1
- Shift 21 (=16+4+1) by doings shifts of size...
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 size 4 and 1
- Shift 21 ($=16+4+1$) by doing shifts of size 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 size 4 and 1
- Shift 21 (=16+4+1) by doing shifts of size 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 size 4 and 1
- Shift 21 ($=16+4+1$) by doing shifts of size 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.

$\begin{align*}
(s==0)\?\{a,b,c,d\}:@\{0,0,a,b\};
\end{align*}$
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
- Shift circuit can be expressed as $\log n$ nested conditional expressions where $s_0, s_1$ ..
  Represent the bits of $n$
Logical right shift circuit

- Define \( \log n \) shifters of sizes 1, 2, 4, ...
- Define \( \log n \) muxes to perform a particular size shift
- Shift circuit can be expressed as \( \log n \) nested conditional expressions where \( s_0, s_1 \) ..

Represent the bits of \( n \)

We will explore such a design in the next recitation