Expressions are combinational circuits
  - constants, variables, operators and function applications

Basic Types
  - Strong typing
  - Types describe values (independent of wires and storage elements)

Variables: declaration, assignment, control structures
  - The “non-procedural” or “combinational” view of variable assignment
  - Think HW, don’t think simulation!

Functions are simply parameterized combinational circuits
  - Function application simply connects a parameterized combinational circuit to actual inputs

Variable scoping
Expressions are combinational circuits

- Expressions are built from constants, variables, operators and function applications
  - Variables are just names for wires in a combo ckt
  - In BSV we never think of variables as storage elements or containers
- Expressions just describe the “data flow” in a comb. circuit

\[
\begin{align*}
b &= 10; \\
bsq &= b \times b; \\
discr &= bsq - 4a \times c;
\end{align*}
\]

Syntax details

- Comments: // line comment, /*... */ block comment
- Whitespace: space, tab, newline, form feed, return
- Identifiers: a combination of letters, digits, $ or _
- Case sensitive: Var, vaR or var are not the same
- Standard static scoping
- Examples of integer literals (compiler figures out required width and type):
  - 125, 56, 2
  - 'b1001, 'h23, 'd90, 'o675, 'B1001, 'H23, 'D90, 'O675
  - '0, '1 all zeros all ones
- Examples of bit-vector literals (exact width):
  - 4'b1001, 8'h2f, 16'd90, 2'o2

(same as in Verilog, SystemVerilog)
Basic Operators
(from Verilog/SystemVerilog)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+ - ! ~</td>
<td>Plus, minus, not, bit wise invert</td>
<td>* / %</td>
<td>Multiplication, division, modulus</td>
</tr>
<tr>
<td>&amp;</td>
<td>And reduction</td>
<td>+ -</td>
<td>Addition, subtraction</td>
</tr>
<tr>
<td>¬&amp;</td>
<td>Nand reduction</td>
<td>=&lt; &gt;&lt;&gt;</td>
<td>Logical shift left, right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;= &gt;= &lt; &gt;</td>
<td>Comparison operators</td>
</tr>
<tr>
<td>¬</td>
<td></td>
<td>&gt;= ! =</td>
<td>Equality, inequality</td>
</tr>
<tr>
<td>^</td>
<td>Xor reduction</td>
<td>&amp;</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>¬^ or ^~</td>
<td>Xnor reduction</td>
<td>^</td>
<td>Bitwise xor</td>
</tr>
<tr>
<td>¬^ or ^~</td>
<td>Bitwise xnor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise or</td>
<td></td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>Logical and</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical or</td>
</tr>
</tbody>
</table>

Basic Types

- Types play a central role in BSV
- Types are described with Type Expressions
  - Simple type expressions are just identifiers
  - (Later, we'll see more complex type expressions)
- In general, type identifiers begin with an uppercase letter
  - Exceptions: ‘int’ and ‘bit’, for compatibility with Verilog

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Unbounded signed integers. Static elaboration only.</td>
<td>Integer num1 = 3;</td>
</tr>
<tr>
<td>int</td>
<td>32-bit wide signed integers</td>
<td>int num2 = 'h1;</td>
</tr>
<tr>
<td>bit[15:0]</td>
<td>16-bit wide bit vector</td>
<td>bit [15:0] x = 23;</td>
</tr>
<tr>
<td>Bool</td>
<td>Possible values True/False</td>
<td>Bool condition = False;</td>
</tr>
<tr>
<td>String</td>
<td>As in Verilog, VHDL or C</td>
<td>String msg = &quot;Hello world\n&quot;;</td>
</tr>
<tr>
<td>...</td>
<td>... see manual for more ...</td>
<td></td>
</tr>
</tbody>
</table>
Syntax details

- BSV’s identifier convention:
  - Type identifiers begin with an uppercase letter
    - (with the previously mentioned exception of ‘int’ and ‘bit’ for compatibility with Verilog)
  - Value identifiers begin with a lowercase letter

- Later, when we talk about polymorphic types we will introduce type variables
  - Type variables also begin with a lowercase letter

- All of this is usually obvious from context

Strong Typing

- Every variable and expression has a type
- The Bluespec compiler checks that constructs in the language are applied correctly according to types:
  - Operator’s/function’s arguments are of the correct type
  - Assignment is to the correct type
  - Module’s parameters are of the correct type
  - Module’s interface is of the correct type
- In case of mismatch, issues an error message
- More stringent than Verilog/SystemVerilog
  - Even registers are strongly typed
  - No automatic sign- or zero-extension; no automatic truncation
    - But you don’t have to tediously calculate the amount of extension or truncation; the compiler will do it for you
**Strong Typing: extension/truncation**

- More stringent than Verilog/SystemVerilog: no automatic sign- or zero-extension; no automatic truncation
  - But you don’t have to tediously calculate the amount of extension or truncation; the compiler will do it for you

```plaintext
bit[31:0] x;

x = signExtend(25'h9BEEF);
x = zeroExtend(25'h9BEEF);
x = {0, 25'h9BEEF}; // same as zeroExtend
x = zeroExtend(39'h9BEEF); // error: input too wide
x = truncate(37'h9BEEF);
x = truncate(25'h9BEEF); // error: input too narrow
```

**Types describe sets of values**

- A type describes a set of values

- Types are independent of entities that may carry values (such as wires, registers, …)
  - No inherent connection with storage, or updating

- This is true even of complex types (to be described later in the training)
  - E.g., struct { int …, Bool …}
  - This just represents a set of pairs of values, where the first member of each pair is an int value, and the second member of each pair is a Bool value
An important strong property of BSV types

• Any expression
  • is guaranteed by BSV’s type-checking rules to represent a pure (combinational) value:
    • It cannot allocate any state
    • It cannot update any state
  • except if its type contains either of the following two special types (to be described later)
    • Action
    • ActionValue

• Hence, any such expression can be freely shared or replicated without changing behavior
  • The BSV compiler exploits this to perform aggressive “common subexpression elimination” optimization (CSE)

Variable declaration, initialization and assignment

• Every variable has a type
• We use standard Verilog notation for declaring a variable with its type, optional initialization, and assignment

```plaintext
type var [= init], var [= init], ...;

int x, y = 23, z;
Bool b;

z = y + 2;
x = z * z;
b = (x >= 23);
```
Variable assignment

- BSV does not use Verilog’s “process” or “procedural” notation (to express behavior, BSV uses Rules)
  - A variable is not an "updatable container"
  - An assignment is not a “procedural” statement that updates a container
  - (Registers and state elements are modules, discussed later)
- A variable is just a name for an expression
- Repeated assignment is just a notation for incrementally building up expressions
  - Think of it as a new variable from that point on
- Think hardware, not simulation or software!

```
int a1 = 23;
a2 = a1 * a1;
a3 = a2 - 5;
```

Variable assignment

- Repeated assignment is just a notation for incrementally building up expressions
  - Think of it as a new variable from that point on
  - It’s just static elaboration!
- Think hardware, not simulation or software!

```
int a = 10;
if (b) a = a + 1;
else a = a + 2;
if (c) a = a + 3;
```
Variable assignment

- Repeated assignment is just a notation for incrementally building up expressions
  - Think of it as a new variable from that point on
  - It’s just static elaboration!
- Think hardware, not simulation or software!

```
int a = 10;
for (int k = 20; k < 24; k = k+1)
    a = a + k;
```

Expressions are combinational circuits

- In summary: expressions and assignments just describe the “data flow” in a combinational circuit
  - Variables are not storage locations
- Think static elaboration, for variables and assignments, not processes or procedures
- Think hardware, not simulation or software
  - No “sensitivity lists”
  - No “always_comb” (SystemVerilog)
  - No “execute an assignment to drive this wire”
Variable declaration and initialization using “let”

- BSV has a “let” statement in which a variable can be declared and initialized, with the compiler deducing the type of the variable based on the type of the right-hand side.

  - Use it judiciously! The implicit type can make it less readable (and therefore less maintainable).

```plaintext
let var = init;  // syntax
let x = 24'h9BEEF;  // compiler deduces type bit[23:0] for x
let y = x + 3;   // compiler deduces type bit[23:0] for y
```

Functions are parameterized expressions

- A function is just an abstraction of a combinational expression.
- Arguments are inputs to the circuit.
- The result is the output of the circuit.
  - (Note: the output could be a struct, carrying multiple values)

```plaintext
function int discr (int a, int b, int c);
  return b*b - 4*a*c;
endfunction
```
Function application

- Function application is just a way to compose (connect) a combinational expression into some context

```plaintext
function int discr (int a, int b, int c);
return b*b - 4*a*c;
endfunction
```

```plaintext
d = discr (10, p, q);
```

---

Function application

- Function application is just a way to compose (connect) a combinational expression to something else

- Think static elaboration:
  - Function application just instantiates some combinational hardware

- Think hardware, not simulation or software
  - No “call the function, execute the body, return the result”
  - No “allocate a stack frame”
Functions: advanced topics

- (Not covered in this lecture)

- Functions can be used both for static elaboration and to describe combinational circuits

- When used for static elaboration, function arguments and results can have any type, not just restricted types as in Verilog or SystemVerilog
  - Rules, interfaces, modules, even functions
  - This is a very powerful abstraction mechanism
  - This leads to very powerful “generate” capabilities, where one can programatically describe complex hardware
    - Loops/linear recursion for linear repetitive circuit structures
    - Binary recursion for tree-structured circuits (e.g., “s” reduction)

Static scoping

- BSV follows traditional static scoping rules
  - Each use of a variable refers to the declaration in the nearest textually enclosing scope
Static scoping

- Recommendation for readability: don't re-assign a variable in an inner scope
  - Semantically meaningful (advanced topic), but can be confusing to read

```plaintext
int x = 3;

function int f (int b, int y);
  ...
  if (x > b) begin
    ...
    x = x + 1; // re-assigning x from surrounding scope
  end
  ...
endfunction
```

End of Lecture