BSV point of view

- In the history of programming languages, *types* have played a central role in abstraction mechanisms
  - Types and type-checking play a central role in ensuring correctness
- BSV is an HDL that uses many of these ideas, wherever they make sense for HW
- Basic types in BSV are simple, familiar from Verilog
- SystemVerilog extensions include typedefs, enums, structs, tagged unions, interface types, type parameterization/polymorphism
- BSV further extends this to systematic *overloading*
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

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
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 = “Hello world\n”;</td>
</tr>
<tr>
<td>...</td>
<td>... see manual for more ...</td>
<td>...</td>
</tr>
</tbody>
</table>

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)

Type synonyms with “typedef”

- “typedef” is often used to define a new, more readable or convenient synonym for an existing type
  - The new type is just a synonym; variables and expressions of either type can be mixed/assigned freely

```
typedef existingType NewType;
typedef int Addr;
typedef bit[63:0] Data;
typedef bit[15:0] Halfword;
typedef Bool RoundRobinFlag;
```

Reminder: type names begin with uppercase letter!
Defining a new enum type

- An enum type defines a new type (not a synonym!) with a set of scalar values with symbolic names.

  - Because it’s a new type, it’s more robust (type-safe) to define an enum, compared to using ints or bit-vectors to represent the set of values (type-checking ensures that you cannot accidentally use an unrelated integer/bit value).

```plaintext
typedef enum { Identifier, … , Identifier } NewType deriving (Bits, Eq);
typedef enum { Green, Yellow, Red } TrafficLight deriving (Bits, Eq);
typedef enum { Reset, Count, Decision } State deriving (Bits, Eq);
```

Syntax notes for enum typedefs

- Enum label identifiers must begin with uppercase letter.
- Enum labels can be repeated in different enum definitions.
- The default encoding of labels (because of “deriving (Bits)”) is 0, 1, 2, …, using just enough bits to encode the full set.
  - (Other encodings may be specified: see LRM on enums and later discussion on overloaded pack/unpack functions.)
Using an enum type

- A defined enum type is used just like any other type: to declare variables, function/module parameters, etc.
- The enum labels are used as constant values of that type

```verilog
typedef enum { Reset, Count, Decision } State deriving (Bits, Eq);
State defaultState = Reset;

function State nextState (State s);
case (s)
  Reset:  s = Count;
  default: s = Decision;
endcase
return s;
endfunction
```

Using an enum type

- Each defined enum type is a new type distinct from all others

```verilog
typedef enum { Green, Yellow, Red } TrafficLight deriving (Bits, Eq);
typedef enum { Reset, Count, Decision } State deriving (Bits, Eq);

bit [1:0] x;
State s1 = x; // type-checking error
State s2 = Yellow; // type-checking error
```
Defining a new struct type

- A struct (sometimes called a “record”) is a composite type. A struct value is a collection of values, each of which has a particular type and is identified by a member or field name.

```plaintext
typedef struct { type identifier; ...; type identifier; } NewType
deriving (Bits, Eq);

typedef enum { Load, Store, LoadLock, StoreCond } Command
deriving (Bits, Eq);

typedef struct {
    Command command;
    bit [31:0] addr;
    bit [63:0] data;
} BusRequest
deriving (Bits, Eq);
```

Syntax notes for struct typedef

- New type name, begins with uppercase
- Member names begin with lowercase
- Member names can be repeated in other structs

Explained later, with “Overloading”
Struct representations

- The default representation of a struct (because of “deriving (Bits)”) is simply a concatenation of the member representations, from MSB to LSB
  - (See later discussion on Overloading/pack/unpack for the general-purpose way of specifying arbitrary encodings)

```plaintext
typedef enum { Load, Store, LoadLock, StoreCond } Command deriving (Bits, Eq);

typedef struct {
    Command command;
    bit [31:0] addr;
    bit [63:0] data;
} BusRequest deriving (Bits, Eq);
```

Using a struct type

- A defined struct type is used just like any other type: to declare variables, function/module parameters, etc.
- The member names are used to access members of that type

```plaintext
BusRequest req;
req.command = Load;
req.addr = baseAddr + 32'h16;
req.data = 64'h9BEEF;
if (req.command == LoadLock)
    req.command = StoreCond;
```
Entire struct values

- A struct type defines a set of struct values, independent of entities that may carry such values (wires, registers, ...)
  - A struct value is not, per se, associated with any storage

```verilog
BusRequest req = BusRequest {
    command: Load,  // Command to load data
    addr: baseAddr + 32'h16,  // Address to load data
    data: 64'h9BEEF };  // Data to load
```

Function for incrementing address

```verilog
function BusRequest incrAddr (BusRequest req);
    req.addr = req.addr + 4;  // Increment address
    return req;
endfunction
```

Using “let” for struct initialization

- It is often convenient to use “let” for declaring and initializing a struct value

```verilog
let req = BusRequest {
    command: Load,  // Command to load data
    addr: baseAddr + 32'h16,  // Address to load data
    data: 64'h9BEEF };  // Data to load

// Compiler deduces ‘req’ to have type BusRequest
```

- lets are not allowed in global scope
Each struct type is distinct

- Each defined struct type is a new type distinct from all others (even though they may happen to have members with the same types)

```plaintext
typedef struct { int a; Bool b; } Foo deriving (Bits, Eq);
typedef struct { int c; Bool d; } Baz deriving (Bits, Eq);
typedef Bar Baz; // type alias or synonym

Foo x;
Baz y;
Bar z;

x = y;      // type-checking error
x.a = y.c;  // ok
x.b = y.d;  // ok
y = z;      // ok
```

More struct examples

// Example 1
// Define structure frame for transmission
typedef struct {
  Bit#(8) header;
  Vector#(2,Bit#(8)) payload;
  Bit#(8) trailer;
} Frame;

// Create the frame content
Frame frame;
frame.header = 8'h01;
frame.payload[0] = 8'hAF;
frame.payload[1] = 8'h12;
frame.trailer = 8'h20;

// Example 2
// Define structure pixel
typedef struct {
  Vector#(3,UInt#(8)) color;
  UInt#(4) brightness;
  Bool blinking;
} Pixel;

// Define vector screen line
Vector#(256, Pixel) screenLine;
screenLine[0].color[1] = 8'hFF;
screenLine[0].brightness = 4'd2;
screenLine.blinking = FALSE;
Parameterized types

- Many types have some other types associated with them in some orthogonal (independent) way. E.g.,
  - With each array type, we associate the type of each item contained in the array
  - With each memory type, we associate the type of addresses and the type of data
  - With each register file type, we associate the type of register names and register data

- System Verilog introduces a notation for this:

\[
\text{Type } \# \left( \text{Type, \ldots, Type} \right)
\]

### Example

- `Mem #( Addr, Data)`
- `RegFile #( RegName, RegData)`
- `Client #(Request, Response)`
- `Server #(Request, Response)`

\[
\text{yields Requests, accepts Responses} \quad \text{accepts Requests, yields Responses}
\]

Numeric (Size) Types

- In BSV, some type parameters are numeric and indicate something about the size of each value of that type

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit#(n)</td>
<td>Bit-vector of width n</td>
<td>Bit#(132) vect1 = 132’d30; // = bit[131:0]</td>
</tr>
<tr>
<td></td>
<td><code>NB:</code> bit [n-1:0] = Bit#(n)</td>
<td></td>
</tr>
<tr>
<td>UInt#(n)</td>
<td>Unsigned integers of width n</td>
<td>UInt#(4) vect2 = 4’b1;</td>
</tr>
<tr>
<td>Int#(n)</td>
<td>Signed integers of width n</td>
<td>Int#(16) vect3 = 16’hFF00;</td>
</tr>
<tr>
<td></td>
<td><code>NB:</code> int = Int#(32)</td>
<td></td>
</tr>
<tr>
<td>Vector#(n,f)</td>
<td>Vector of n elements, each of type f</td>
<td>Vector#(3, Bool) vect4;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vector#(14,Int#(32)) vect5;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vector#(16,Tuple2#(Bool, Bit#(8))) vect6;</td>
</tr>
</tbody>
</table>
Numeric (Size) Types

- In numeric type positions, you can only supply the constants 0, 1, 2, 3, ...
  - (But see later, type variables for polymorphic types)
  - (Also, later, see how to express arithmetic constraints on numeric types)
- You cannot mix ordinary numeric expressions with numeric types. There is no ambiguity:
  - Numeric types only occur in places where a type expression is expected (e.g., a type parameter to another type)
  - Ordinary numeric expressions only occur in value contexts

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

```verbatim
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
```
A useful type: Maybe#(t)

- In HW design one frequently has the following situation:
  - A value of some type t
  - An accompanying "valid bit" which says whether the value is meaningful or not
- The BSV library provides the type Maybe#(t) for this purpose

```plaintext
Maybe#(int) m1 = tagged Valid 23;  // valid bit True, value 23
Maybe#(int) m2 = tagged Invalid;   // valid bit False, value unspecified
m2 = m1;                           // This sets m2 Valid and 23

// Some functions
Bool b  = isValid (m2);             // b == valid bit of m2
int d   = fromMaybe (34, m2);       // d = value of m2 if valid, else 34
```

Tagged Unions

- Type which contains tags identifying one of a set of types
- Each tag could take different types
- Maybe type is a particular case of tagged union: Valid/Invalid

```plaintext
// Define tagged union BusTraffic
typedef union tagged {
  struct {
    Bit#(8)  header;
    Bit#(8)  payload;
    Bit#(8)  trailer;
  } LongFrame;
  struct {
    Bit#(10) payload;
    Bit#(8)  trailer;
  } ShortFrame;
  Bit#(10) Symbol;
} BusTraffic;

// Write a tagged union member
BusTraffic busElem1, busElem2;
busElem1 = tagged Symbol 10’d6;
busElem2 = tagged ShortFrame(payoad:10’d2, trailer:8’d1);
```

```plaintext
2 bits 8 bits 8 bits 8 bits
00 Header  payload  trailer
01 payload  trailer
10 payload  payload
```

BusTraffic
LongFrame
Short Frame
Symbol
A first brush with Overloading

- Overloading: the ability to use a common function name or operator on some repertoire of types. Example:
  - “+” is meaningful on bits, integers, floating point, perhaps colors (add RGB values?), etc.
  - “<” is meaningful on bits, numeric types, vectors, perhaps Ethernet packets (less priority?), ...

- In most languages, overloading is *ad hoc*
  - Usually, only on a fixed set of operators
  - Usually, not extensible by the designer/programmer
  - Minor exception: SystemVerilog allows extensibility over a fixed set of operators
  - Major exception: C++ has systematic extensibility

A first brush with Overloading

- BSV has a systematic way to extend overloading to any operator and function, and to any type
  - The terminology includes *typeclasses*, *typeclass instances*, *provisos*, and *deriving*
  - But we’ll visit these in a later lecture
  - For now, we’ll just focus on the *deriving* construct
A first brush with Overloading: deriving (Bits)

- Rather than use ad hoc rules about how to represent a particular type \( t \) in bits, BSV takes the following systematic route:
  - There are two overloaded functions:
    - function Bit#(n) pack (t x);
    - function t unpack (Bit#(n) y);
  - The pack() function encapsulates how to convert a value \( x \) of type \( t \) into a bit-vector of width \( n \)
  - The unpack() function does the reverse

- For any user-defined type, the user can define these functions, and therefore fully control bit representations (future topic)

- With “deriving (Bits)” in the type definition, the user directs the compiler to define pack() and unpack() using “default packing”

- Note: without pack() and unpack(), the compiler assumes no bit representation! This is why, whenever you have a type that will be present in your HW, you must either say “deriving(Bits)”, or define pack() and unpack() explicitly!

A first brush with Overloading: deriving (Eq)

- Rather than use ad hoc rules about how to test equality between two values of type \( t \), BSV takes the following systematic route:
  - There are two overloaded operators for equality and inequality:
    - function Bool \( \equiv \) (t x, t y); // using Verilog notation “\( \equiv \)”
    - function Bool \( \neq \) (t x, t y); // for “escaped identifiers”

- For any user-defined type, the user can define these operators, and therefore fully control the meaning of equality/inequality (future topic)

- By saying “deriving (Eq)” in the type definition, the user directs the compiler to define \( \equiv \) and \( \neq \) using “default rules”

- Note: without \( \equiv \) and \( \neq \), the compiler assumes no way to test for equality! This is why, whenever you expect to do such tests, you must either say “deriving(Eq)”, or define \( \equiv \) and \( \neq \) explicitly!
Future topics on Types

- Polymorphic types (generic types, "template classes"): when types contain type variables
- Tagged Union types, and pattern-matching
- Constraints on numeric types
  - “Size of reg = log() of size of address space”
- More on overloading
  - Interpreting integer literals
  - Bitwise (ops: &, |, ^, ~^, ^~, invert, <<, >>)
  - Ord (ops: <, <=, >, >=)
  - Arith (ops: +, -, negate, *)
  - Bounded (consts: minBound, maxBound)
  - Typeclasses, provisos, typeclass instances

End of Lecture