

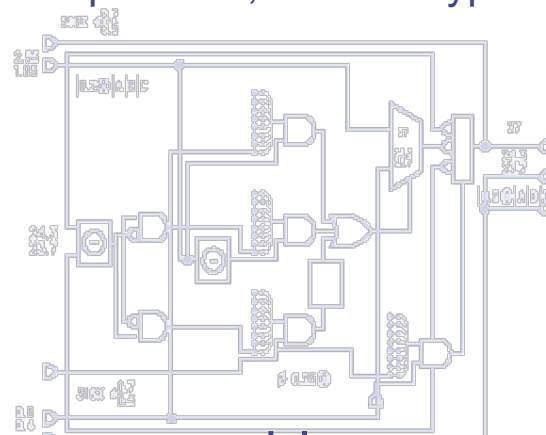


BSV Training

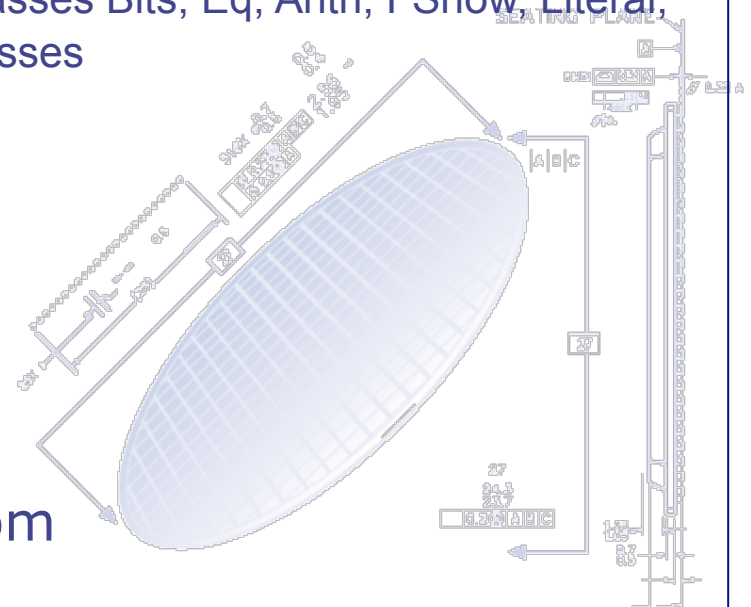
Lec_Typeclasses

Overloading: Typeclasses and instances. Built-in typeclasses Bits, Eq, Arith, FShow, Literal, provisos, numeric typeclasses

```
import PUF004;
typeclass Bits(B) (Bit0T);
module ex_let_outR_in[Empty];
Integer nfa_depth = 16;
function Bits(B) distrupta_pump(Bit0T);
return (outR);
endfunction
PUF004(Bit0T) in_boud;
in_boud = PUF004(nfa_depth) in_boud(in_boud);
PUF004(Bit0T) out_boud;
out_boud = PUF004(nfa_depth) in_boud(out_boud);
PUF004(Bit0T) out_boud;
out_boud = PUF004(nfa_depth) in_boud(out_boud);
rule exp1 (True);
Bit0T in_data = in_boud(in_data);
PUF004(Bit0T) out_data =
  distrupta_pump(in_data) == 0 ? out_boud : out_boud;
endrule;
endmodule;
endmodule;
endmodule;
```



www.bluespec.com



Type Parameterization: Polymorphism and Overloading

- BSV, like C++ and many advanced programming languages, has two kinds of parameterization using types: Polymorphism and Overloading (Typeclasses)
- *Overloading*: The use of the same name (identifier) for *multiple* functions and operators that have different argument and/or result types. In an actual function call, the compiler uses the types of the arguments/results to determine which of the actual multiple functions is intended; this compiler activity is called *overloading resolution*.
 - Example: “+” on integers, floating point numbers, complex numbers, vectors, ...
 - The programmer explicitly declares each possibility
- FYI: overloading in other languages:
 - C, Verilog, SystemVerilog: limited overloading (built-in only, not user-definable), for common operators like +, -, *, /, ==, !=, ...
 - C++: Fully extensible. User can extend overloading of built-in operators like +, -, ... and can also overload user-defined functions and methods
- BSV: overloading that is as powerful as C++
 - BSV’s overloading is type-checked even during separate compilation (not so in C++)
 - Overloading is used extensively within *synthesizable* designs
 - BSV’s overloading system is taken from the Haskell functional programming language

Overloading

Overloading in BSV involves the following concepts:

- *Typeclasses*: defining a set of overloaded names (identifiers) and their types
- *Instances*: defining actual functions for overloaded names
- *Provisos*: constraints specifying that a particular function or module parameter must belong to a certain typeclass, i.e., must be overloaded in a certain way
- “deriving” clauses on type definitions: automatic creation of certain instances

In this lecture we focus on the most common uses of overloading:

- Attaching “deriving” to type definitions, most often “deriving (Bits, Eq)”.
- Attaching “provisos” to functions and modules
- Defining new instances of existing typeclasses

Separating the logical view of types from representations

Consider the following type declarations (from a CPU example):

```
typedef Bit#(4) RegName;

typedef enum { Noop, Add, Bz, Ld, St } Opcode deriving (Bits, Eq);

typedef struct {
    Opcode    op;
    RegName   dest;
    RegName   src1;
    RegName   src2;
} Instr
    deriving (Bits);
```

There are many choices in how we implement these in bits:

- How many bits for Opcode (3 are enough, but should we use more, e.g., for future expansion)?
- What encodings of bits for Noop, Add, Bz, Ld and St?
- How many bits for Instr, and what layout for the fields op, dest, src1 and src2?

Further, the choices may change (as we add more opcodes, re-architect the CPU, ...).

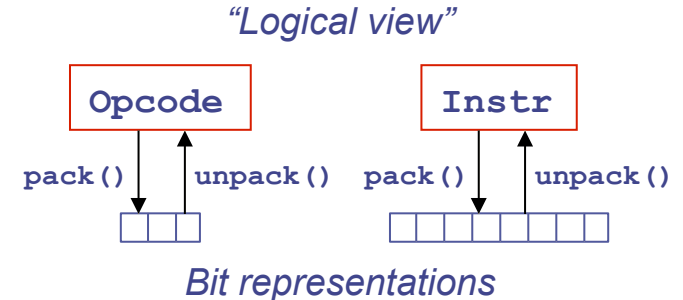
In BSV, using overloading, we cleanly separate out the logical view (in the red box above) from implementation choices, so that:

- we only have to change two overloaded functions to change our implementation choices
- the rest of our code (typically > 99%) only uses the logical view, and never has to be changed

Separating the logical view of types from representations

We use two overloaded functions, `pack()` and `unpack()` to “convert” between the logical view and bits:

Why use overloaded functions? Because we’d like to use the same function names, “pack” and “unpack” for many types (like “Opcode” and “Instr”) and not have to invent new names for each case.



Note: this is a common “experienced programmer” trick in C++. Instead of exposing the representation members of a class as “public” members, the programmer labels them “private”, and only accesses them from outside via public “setter” and “getter” methods. This allows future changes in representation without having to change all the code that uses objects of that class.

When you use a BSV primitive like `Reg` to hold a type like `Instr`:

```
Reg #(Instr) rg_instr <- mkReg( Instr{op:Noop,
                                   dest:0,
                                   src1:0,
                                   src2:0});

...
rule rl_exec ( rg_instr.op == Add );
...
endrule
```

When you initialize the reg, or later use `_write`, we only use the logical view; the module internally uses “`pack()`” to convert to bits for storing in a hardware register.

When you use `_read`, we only see the logical view; the module internally uses “`unpack()`” to convert the bits in the hardware register into the logical view.

A typeclass is a group of related overloaded identifiers

pack() and unpack() are related:

- we typically define them together for each new logical type
- we typically expect them to be inverses
 - $\text{pack}(\text{unpack}(x)) == x$ $\text{unpack}(\text{pack}(y)) == y$

Similarly, when we define arithmetic operations on various types (int, float, vector, ...)

- we typically define the operations together: $+$, $-$, $*$, $/$, ...
- we typically expect them to have an “algebra”:
 - $x + (y + z) == (x + y) + z$ $x * (y + z) = x * y + x * z$... and so on

In BSV, this idea of a related group of overloaded identifiers is formalized into the concept of a “typeclass”. A typeclass declares which overloaded identifiers belong to the group, and what are their most general types. This is independent of any actual definitions of these overloaded identifiers (which is called an “instance” of the typeclass).

For example, the BSV library pre-defines the following typeclasses:

```
typeclass Bits #(type t, numeric type n);  
  function Bit #(n) pack    (t x);  
  function t          unpack (Bit #(n) bs);  
endtypeclass
```

```
typeclass Arith #(type t);  
  function t \+ (t x, t y);  
  function t \- (t x, t y);  
  function t \* (t x, t y);  
  function t \/ (t x, t y);  
  ...  
endtypeclass
```

Note: using a standard Verilog trick to use “\” for non-alphanumeric identifiers

The actual overloaded functions form an “instance”

To specify a custom representation for a new type, we use an “instance” declaration:

```
instance Bits #(Opcode, 4);
  function Bit #(4) pack (Opcode op);
    case (op)
      Noop : return 4'b0010;
      Add  : return 4'b0100;
      Bz   : return 4'b0001;
      ... etc. ...
    endcase
  endfunction

  function Opcode unpack (Bit #(4) b4);
    case (b4)
      4'b0010: return Noop;
      4'b0100: return Add;
      ... etc. ...
    endcase
  endfunction
endinstance
```

Here, we have chosen a 4-bit representation

This is arbitrary user-written code, so we can choose any representation that is appropriate for our application

What about cases of 4-bit values that do not represent any of the five opcodes? As usual, it's your design choice about what should happen. E.g.,

- Leave it unspecified. In fact, if you don't use 'pack' and 'unpack' explicitly (only used inside BSV primitives), BSV's strong type-checking guarantees that b4 cannot be an illegal value.
- Map illegal values to Noop.
- Test for illegal values and raise an error, before calling pack only on legal values
- etc.

The “deriving” shortcut (instead of “instance”)

In practice, the “obvious” representation is almost always perfectly fine. In these cases, we use the “deriving (Bits)” shortcut instead of declaring an instance.

```
typedef Bit#(4) RegName;  
  
typedef enum { Noop, Add, Bz, Ld, St } Opcode deriving (Bits, Eq);  
  
typedef struct {  
    Opcode    op;  
    RegName   dest;  
    RegName   src1;  
    RegName   src2;  
} Instr  
deriving (Bits);
```

bsc will implicitly declare an instance for Opcode, using 3 bits, and encoding the labels as 001, 010, 011, 100 and 101, respectively

bsc will implicitly declare an instance for Instr, using 15 bits, and representing the struct by the bit-concatenation:

```
{ pack(op), pack(dest), pack(src1), pack(src2) }
```

Note that these are recursive uses of pack() on the types Opcode and Bit#(4)

For every type you use in the hardware, you must either using “deriving (Bits)” or provide an explicit “instance” declaration. (You don’t need to do this for types only used during static elaboration.)

Other typeclasses pre-defined in the BSV library

There are many pre-defined typeclasses, described in detail in Reference Manual, sec B.1. You can use the “deriving” shortcut for Bits, Eq, and Bounded.

Typeclass “Eq” declares the overloaded operators: `==` `!=`

Typeclass “Bounded” declares: `minBound` `maxBound`
(the smallest and largest values of a type)

Typeclass “Arith” declares: `+` `-` `*` `/`
(and many more arithmetic operators and functions)

Typeclass “Ord” declares: `<` `<=` `>` `>=` `compare` `min` `max`

Typeclass “Bitwise” declares: `&` `|` `^` `~^` `^~` `invert` `<<` `>>` `msb` `lsb`

Typeclass “BitExtend” declares: `extend` `zeroExtend` `signExtend` `truncate`

For any user-defined type, you can define your own instance of these typeclasses. E.g., if you define `Arith#(Matrix)` then you can use `+`, `-`, `*`, ... on Matrix values

Users can also define their own new typeclasses (and populate them with instances).

Motivating “provisos”

Suppose we define a function to sort vectors:

```
function Vector #(n, Int#(32)) sort (Vector #(n, Int#(32)) v);  
  ...  
  if (v [j] < v [k]) ...  
  ...  
endfunction
```

Inside the function, we are comparing two of the vector elements with the “<” operator, which is an overloaded operator in the “Ord” typeclass. This is ok, since the elements have type “Int #(32)”, which has been pre-declared in BSV to be in the “Ord” typeclass, and so it is clear what “<” means here.

But what if we want “sort” to be polymorphic, i.e., to sort vectors of elements of any type *t*, not just “int”, provided “<” is meaningful on *t*?

“Provisos” are assertions on polymorphic types

The *polymorphic* function to sort vectors adds a “provisos” assertion:

```
function Vector #(n, t) sort (Vector #(n, t) v)
  provisos (Ord #(t));
...
  if (v [j] < v [k]) ...
...
endfunction
```

The “provisos” clause asserts that any “t” on which this function is used must be in the “Ord” typeclass. This helps *bsc*’s typechecker in two ways:

- *bsc* knows that “<” here is a meaningful operation
 - If you omitted the “provisos” clause, the type-checker will complain
- *bsc* can check, at each place where “sort” is called, that the actual argument is ok, namely a vector of elements of a type which is in the “Ord” typeclass

Note for C++ programmers

You can write a similar “sort” function in C++, where “t” is a C++ template type. So, why does C++ not use/need provisos?

It’s because in C++, templates are explained (and implemented) in terms of in-line expansion, so type-checking of such a function is postponed until each actual call. Eventually, after enough in-line expansion, the code is no longer polymorphic, and so it can be type-checked. This is not only inefficient (repeated type-checking of the function body at each call), but leads to some of C++’s famously complex error messages.

“Provisos”: more examples

```
module mkFoo #(... parameters ...) (InterfaceType#(t))  
  provisos (Bits #(t,tsize), Eq #(t), Arith #(t));  
  
  ...  
endmodule
```

- The “Bits” proviso is an assertion that “t” has the “pack” and “unpack” functions to convert to bits and back.
 - This proviso is typically necessary because inside the module you are probably storing values of type “t” in registers, or other state elements, where they are, of course, ultimately stored as bits.
- The “Eq” proviso is an assertion that “t” has the “==” and “!=” operators defined.
 - This proviso is typically necessary because somewhere inside the module you are probably comparing values of type “t” for equality or inequality.
- The “Arith” proviso is an assertion that “t” has the arithmetic operators “+”, “-”, “*” etc. defined
 - This proviso is typically necessary because somewhere inside the module you are probably performing arithmetic operations on values of type “t”

If you perform overloaded operations on a polymorphic type inside the module, and you forgot to add a corresponding “provisos” clause in the module header, the type-checker will complain about it.

The “Literal” typeclass for integer conversion

Suppose we have an assignment statement like this:

```
rg_x <= 2012;
```

How should we interpret the literal “2012”? How many bits should it have? Is it signed or unsigned? If `rg_x` contains a user-defined type, is this assignment even meaningful? Can we extend the language so that it is meaningful for user-defined types?

Most language have *ad hoc* answers to these questions.
In BSV, there is a systematic answer, based on overloading.

BSV has a built-in type called “Integer” which stands for mathematical (unbounded) integers.* All literals, like “2012” have this type.

BSV has a pre-defined typeclass containing an overloaded function “fromInteger” to convert from Integer to some destination type t:

```
typeclass Literal #(type t);  
    function t fromInteger (Integer t);  
endtypeclass
```

Thus, the above assignment is interpreted as:

```
rg_x <= fromInteger ( the Integer with value 2012 );
```

This gives a precise semantics for integer literals, and is extensible to user-defined types (by just defining new instances for the Literal typeclass):

* Of course, in reality an “unbounded” Integer will be limited by the size of your computer’s memory, but that is a pretty good practical abstraction of infinity!

Motivating the Fmt type and the FShow typeclass

When debugging code, it is useful to “pretty-print” user-defined types.

```
typedef enum { Noop, Add, Bz, Ld, St } Opcode deriving (Bits, Eq);

typedef struct {
    Opcode    op;
    RegName   dest;
    RegName   src1;
    RegName   src2;
} Instr
    deriving (Bits);
```

E.g., when printing an Opcode value with \$display:

```
$display ("The current opcode is: ", rg_instr.op);
```

we should see an output with the symbolic value, not the bit representation value:

```
The current opcode is: Add
```

E.g., we would like to print a full Instr value with \$display (not just its scalar fields):

```
$display ("The current instr is: ", rg_instr);
```

should produce an output like this:

```
The current instr is: Instr { op: Add, dest:3, src1:5, src2: 6 };
```

The Fmt type

BSV provides a pseudo-function, `$format`, whose arguments are just like the arguments to `$display`. However, instead of printing anything, it returns an object of type `Fmt`. E.g.,

```
function Fmt showOpcode (Opcode op) ;  
  case (op)  
    Noop: return $format ("Noop") ;  
    Add:  return $format ("Add") ;  
    ... and so on ...  
  endcase  
endfunction
```

In BSV, `$display` also accepts `Fmt` arguments, in addition to the usual types of arguments.

Now, we can do what we wished for in the last slide:

```
$display ("The current opcode is: ", showOpcode (rg_instr.op)) ;
```

and we see an output with the symbolic value, not the bit representation value:

```
The current opcode is: Add
```

Similarly, we could define a `showInstr ()` for `Instr` values, producing a `Fmt` value.

The FShow typeclass

Instead of inventing new and different names for the “show” functions for each user-defined type, BSV pre-defines a typeclass FShow with the overloaded function name “fshow”

```
typeclass FShow #(t);  
  function Fmt fshow (t);  
endtypeclass
```

Instead of defining “showOpCode” (prev. slide), we define “fshow” for type “Opcode”:

```
instance FShow #(Opcode);  
  function Fmt fshow(Opcode op);  
    case (op)  
      Noop: return $format ("Noop");  
      Add:  return $format ("Add");  
      ... and so on ...  
    endcase  
  endfunction  
endinstance
```

and we use “fshow” in \$displays:

```
$display ("The current opcode is: ", fshow (rg_instr.op));
```


The “deriving” shortcut for “FShow”

For most user-defined types—enums, structs, tagged unions and vectors—there is a “natural” textual representation for printing them

- like the examples in the the previous slides for the enum Opcode and the struct Instr

So, instead of having to write an instance of FShow explicitly for each user-defined type, we can let *bsc* do it automatically using the “deriving” shortcut:

```
typedef enum { Noop, Add, Bz, Ld, St } Opcode deriving (Bits, Eq, FShow);

typedef struct {
    Opcode    op;
    RegName   dest;
    RegName   src1;
    RegName   src2;
} Instr
    deriving (Bits, FShow);
```

(Note: this is a new feature in October 2012. If your installation predates this, you may not have it yet. You will have it when you next upgrade your installation.)

Numeric typeclasses and provisos

In many codes, the sizes of various entities have some defined numeric relationship. BSV provides a pre-defined (and not user-extensible) set of numeric typeclasses. These can be used in provisos to assert numeric relationships.


E.g., suppose we are defining a new FIFO type that is parameterized by n , its capacity. Inside the FIFO, we will have a vector of depth n to hold the data. But we also need registers of width $\log(n)$ bits to keep track of the head and tail of the FIFO.

```
interface MyFifo #(n, t);  
  ... methods ...  
endinterface  
  
module mkMyFifo (MyFifo #(n, t)  
  provisos (Log #(n, m));  
  
  Vector #(n, Reg #(t)) vr_data <- replicateM (mkRegU);  
  Reg #(Bit #(m)) rg_head <- mkReg (0);  
  Reg #(Bit #(m)) rg_tail <- mkReg (0);  
  
  ... rest of module ...  
endmodule
```

This proviso asserts that $\log(n) = m$. Since n is known, it constrains m to this value. (BSV numeric provisos can be read as true equations, and constraints can flow in both directions). Subsequently, we use m to declare the width of our FIFO head and tail registers.

Numeric typeclasses and provisos

BSV provides a collection of numeric typeclasses, described in Sec. B.3.1 in the BSV Reference Guide:

Add # (n1, n2, n3)	corresponding assertions 	$n1 + n2 = n3$
Mul # (n1, n2, n3)		$n1 * n2 = n3$
Div # (n1, n2, n3)		$\text{ceiling } (n1/n2) = n3$
Max # (n1, n2, n3)		$\text{max } (n1, n2) = n3$
Min # (n1, n2, n3)		$\text{min } (n1, n2) = n3$
Log # (n1, n2)		$\text{ceiling } (\log_2 (n1)) = n2$

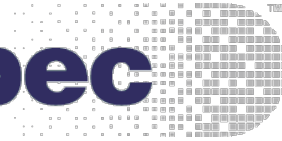
Summary on typeclasses

BSV typeclasses are very powerful (the concept is borrowed with almost no change from the advanced functional programming language Haskell, which has had it since 1991 and with which there is widespread experience).

Typeclasses clean up certain things that are typically done in an ad hoc (and therefore not scalable) manner in many existing SW programming languages and HW design languages:

- Encapsulated, orthogonal, changeable bit representations for user-defined types
- Systematic overloading, with separate compilation
- Conversion of integer literals to built-in and user-defined types
- Composable, recursive pretty-printing of user-defined types
- Numeric type relationships of sizes of hardware components

bluespec



End

```
import PCell*;
typedef Bit[24] StateT;
module ex_hdl_csrR_hdl {
  Integer nfa_depth;
  function Bit[24] determine_pump(StateT s);
  return (s[0]);
  endfunction
  PCellT(bstate) inboud;
  addressPCellT(nfa_depth) the_inboud(inboud);
  PCellT(bstate) outboud;
  addressPCellT(nfa_depth) the_outboud(outboud);
  PCellT(bstate) outboud;
  addressPCellT(nfa_depth) the_outboud(outboud);
  rule exp1 (True);
  bstate in_data = inboud;
  PCellT(bstate) out_data =
    determine_pump(in_data) == 0 ? outboud : outboud;
  out_data;
  in_data;
  out_data;
  endrule; exp1
endmodule; ex_hdl_csrR_hdl
```

Questions?

Join online forums at www.bluespec.com, and ask your question,
or send an e-mail to support@bluespec.com

