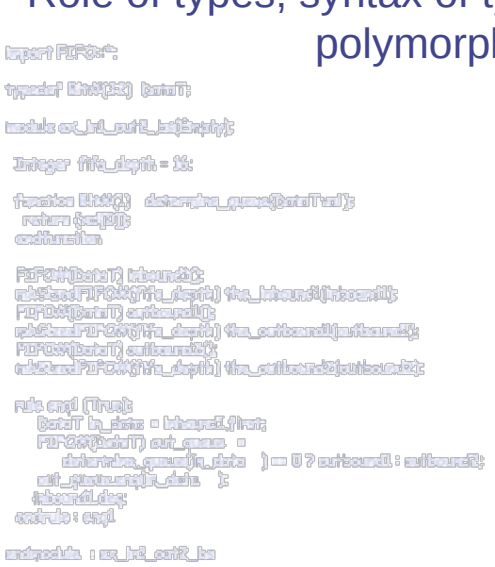
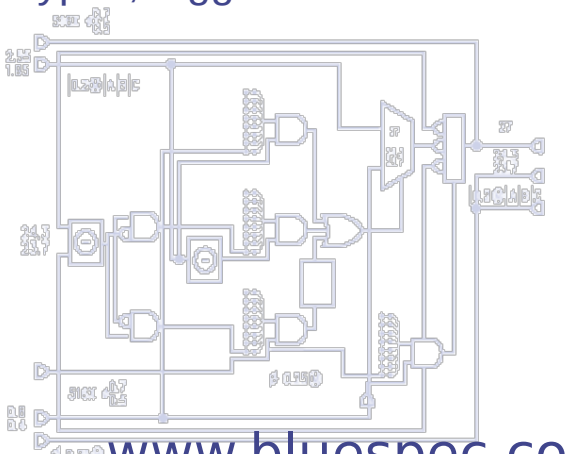
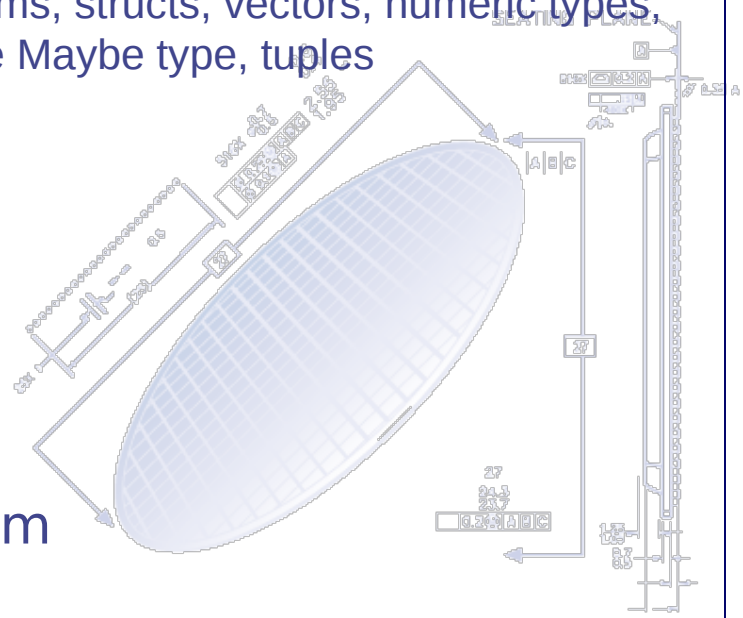
BSV Training

Lec_Types

Role of types; syntax of types and type expressions; enums, structs, vectors, numeric types; polymorphic types, tagged unions and the Maybe type, tuples







www.bluespec.com

The role of types in modern programming languages

Most modern programming languages make strong use of *data types* to raise the level of abstraction and for correctness

- Types are an abstraction: ultimately, all computation, whether in SW or HW, is done on bits, but it is preferable to think in terms of integers, fixed point numbers, floating point numbers, booleans, symbolic state names, ethernet packets, IP addresses, employee records, vectors, and so on
- *Strong type-checking* is used to eliminate unintentional "mis-interpretation" of bits, such as taking the square of symbolic state, subtracting two IP addresses, indexing into an employee record, and so on

Note: historically, and even in some modern programming texts, types are explained in terms of bit representations. The more modern (and mathematical view), is to view each type as a set of abstract values along with the operations that can be performed on those values (i.e., an algebra). This view totally divorces a type from its representation; indeed the same type can have many possible representations. This is the view embraced in BSV.

Types in BSV

- BSV has basic scalar types just like Verilog
- BSV has SystemVerilog type mechanisms like typedefs, enums, structs, tagged unions, arrays and vectors, interface types, type parameterization, polymorphic types
- BSV also has types for static entities like functions, modules, interfaces, rules, and actions
  - (so you can write static-elaboration functions that compute with such entities)
- BSV has very powerful *systematic user-defined overloading*—typeclasses and instances (more powerful than C++)
  - This is used heavily by advanced users
- Type-checking in BSV is *very* strict
  - Even registers are strongly-typed
  - No silent extensions and truncations
  - Typical anecdote (observed by BSV and Haskell programmers, which have the same type system):

    "if it gets through the type-checker, it just works"

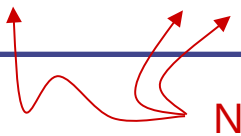
Syntax of types: Type Expressions

BSV uses SystemVerilog's notation for parameterized types

```
Type ::= TypeConstructor #(Type1, ..., TypeN)
       | TypeConstructor            // special case when N=0)
```

i.e., a type expression is a type constructor applied to zero or more other types. In the special case where it is applied to zero other types, the #() part can be omitted. Examples:

Type	*Comments*
Integer	Unbounded signed integers (static elaboration only)
Int#(18)	18-bit signed integers Note: 'int' is a synonym for Int#(32)
UInt#(42)	42-bit unsigned integers
Bit#(23)	23-bit bit vectors Note: 'bit[15:0]' is a synonym for Bit#(16)
Bool	Booleans, with constants True and False
Reg#(UInt#(42))	Interface of register that contains 42-bit unsigned integers
Mem#(A,D)	Interface of memory with address type A and data type D
Server#(Rq,Rsp)	Interface of server module with request type Rq and response type Rsp)



Note uppercase first letter in type names

Typedefs, enums, and structs

```
typedef  Bit #(32) Word;
typedef  Bit #(32) Addr;
typedef  Bit #(32) Data;

typedef  Bit#(4) RegName;
```

Simple typedefs (left) are just synonyms for readability; all these types are equivalent.

Enum and struct typedefs (below) define new types, not equivalent with any other type. (So, type-checking prevents misuse.)

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

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

```
typedef struct {
   Opcode    op;
   RegName   dest;
   Bit #(32) v1;
   Bit #(32) v2;
} DecodedInstr
  deriving (Bits);
```

”deriving (Eq)” tells bsc to pick a “natural” equality comparison operator for this type.

“deriving (Bits)” tells bsc to pick a “natural” bit-representation for this type.

(detailed treatment in a later section)

bluespec

structs

Enum and struct types are “first class” types. They can be stored in state elements. They can be passed as method and function arguments and results, etc. (unlike C, but like C++).

```
Reg #(Opcode) rg_op <- mkReg (Noop);
FIFOF #(DecodedInstr) buf <- mkFIFOF;
```

Strongly typed: they can never contain values of any other type, even if they are represented in the same number of bits.

Like C/C++, you can declare a variable with a struct type, and incrementally assign its members (fields). However, we often directly build entire struct values using struct expressions:

```
rule fetch (buf.notStall (instr));
   let  di = DecodedInstr { op:   instr.op,
                            dest: instr.dest,
                            v1:   rf.sel1 (instr.src1),
                            v2:   rf.sel2 (instr.src2) };
   buf.enq (di);
   pc <= pc + 1;
endrule: fetch
```

Vectors

We commonly use Vectors to express repeated structures.
In any package where we use them, we must first import the Vector package:

```
import Vector :: *;
```

Vectors are just type constructors, like any other. In particular, they can contain any types (including other Vectors):

```
interface EHR #(numeric type n, type t);
   interface Vector #(n, Reg #(t)) ports;
endinterface

typedef  Vector #(10, Vector #(5, Int #(16)))  Matrix;
```

Vectors are indexed with the usual square bracket "[ ]" notation:

```
Int #(5) new_val = extend (ctr.ports [p]) + extend (delta);
if (new_val > 7) ctr.ports [p] <= 7;
else if (new_val < -8) ctr.ports [p] <= -8;
else ctr.ports [p] <= truncate (new_val);
```

Numeric types

- Some type constructors take *numeric types* in certain type-parameter positions. Examples:
  - 18-bit signed integers
  - Vector of sixteen 42-bit unsigned integers

```
Int #(18)
```

```
Vector #(16, UInt #(42))
```

- In a position where a numeric type is expected, you can provide:
  - Literal numeric values: 18, 16, 42, ...
  - Numeric type expressions:

```
TAdd #(18,16), TMul #(2,32), TLog #(19)
```

- Although these have superficial similarity to numeric values and numeric value expressions:
  18 16 42 18+16 2*32 log2(19), ...
  they are not the same!
- Specifically, numeric types and type expressions are much weaker than full-blown numeric values and arithmetic expressions because:
  - Type-checking is performed in a separate, earlier phase of the compiler before any numeric value expression evaluation
  - Type-checking needs to be resolved statically

Numeric types vs. numeric values

Numeric types are used in certain positions in type expressions and in type classes. Examples:

```
Bit #(23)
Int #(Tadd #(n, 16))
Vector #(8, sometype )
MyFifo #(4, sometype )
Bit #(SizeOf ( sometype ))
```

Numeric *types* are completely distinct from numeric *values*
(even though we use the same literals, and we do similar ops like Add, Log, …).

The central reason for this is that *bsc* does type-checking early, and must resolve types completely during compile-time. The numeric types sub-language is carefully designed to allow this. It would be undecidable if it could express arbitrary arithmetic, and thus we cannot use ordinary numeric values in types.

However, there is no such danger in going in the opposite direction, i.e., to use a numeric type where we need an ordinary numeric value. For this, BSV provides a pseudo-function (see Reference Guide Sec. B.3.3) suggested by this prototype (which is of course not legal syntax because BSV functions take values as arguments, not types):

```
function Integer valueOf ( a numeric type );
```

Polymorphic types

- Any type-parameter in a type expression can be a *type variable* (identifier beginning with a lower-case letter). Examples:
  - *n*-bit signed integers `Int #(n)`
  - Vector of *m* elements, each of of type *t* `Vector #(m, UInt #(t))`
- This allows for writing highly parameterized designs
- [C++ users: BSV type variables are like *template types*]

The “Maybe” type

BSV (and SystemVerilog) have kind of type called “tagged unions”.
One tagged union type frequently used in BSV is the “Maybe” type.
(Reference Manual section B.2.10)

The type declaration

```
typedef union tagged {
    void  Invalid;
    t     Valid;
} Maybe #(type t)
  deriving (Eq, Bits);
```

A “Maybe#(t)” value is
- *either “invalid” (with no associated value, i.e., void)*
- *or “valid” with an associated value of type “t”*

i.e., a “valid” bit along with a value

Creating values of this type

```
tagged Invalid
```

creates 0 (invalid) along with a don’t care value

```
tagged Valid expression
```

creates 1 (valid) along with the value of the expression

Using values of this type, with “pattern matching”

```
if (value matches tagged Valid .x)
    … here you can use x,  the valid associated value …
else
    … here you handle the “invalid” case …
```

Tagged unions are similar to “unions” in C/C++, except that tagged unions are type-safe, whereas unions are not. There is no way to examine the value when the valid bit is “invalid”.

"Tuple" types

A 2-tuple is just a pair of values; a 3-tuple is a triple, … and so on

The 2-tuple type:

```
Tuple2 #(t1, t2)
```

A pair of values, the first one of type t1, and the second of type t2

Creating values of this type

```
tuple2 (expression1, expression2)
```

creates a value with two components, the value of expression1 and the value of expression2

Using values of this type: functions to extract components:

```
tpl_1 (expression), tpl_2 (e), …
```

extract the j'th component of tuple that is value of expression

Using values of this type, with "pattern matching"

```
match { .x, .y } = expression;
```

declares new variables x and y, and binds them to the components of the 2-tuple value of expression

2-tuples are just structs with 2 fields (in general, an *n*-tuple is a struct with *n* fields).

But tuples are so useful and common that they're pre-defined in BSV.

Exporting abstract types

File Foo.bsv

```
package Foo;

import Bar :: *;
   …
```

Code here can declare identifiers of type Request, can declare registers holding values of type Request, etc. It can retrieve values of type Request from method m1, send arguments to method m2, etc.

But it can never examine the fields (members) of such values, since the member names are not available here. Thus, within Foo, Request is an opaque (abstract) type.

```
   …
endpackage: Foo
```

Bar exports mkBar and the type Request, but not the fieldnames (members) addr and data.

File Bar.bsv

```
package Bar;

export Request, mkBar;

   …

typedef struct {
   Bit #(16) addr;
   Bit #(32) data;
   } Request;

   …

module mkBar (…);
   …
   method Request m1 (…)…

   method Action m2 (Request r)…

   …

endpackage
```


bluespec

Hands-on

- BSV-by-Example book: Examples in Chapter 3 and 10

End

Questions?

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