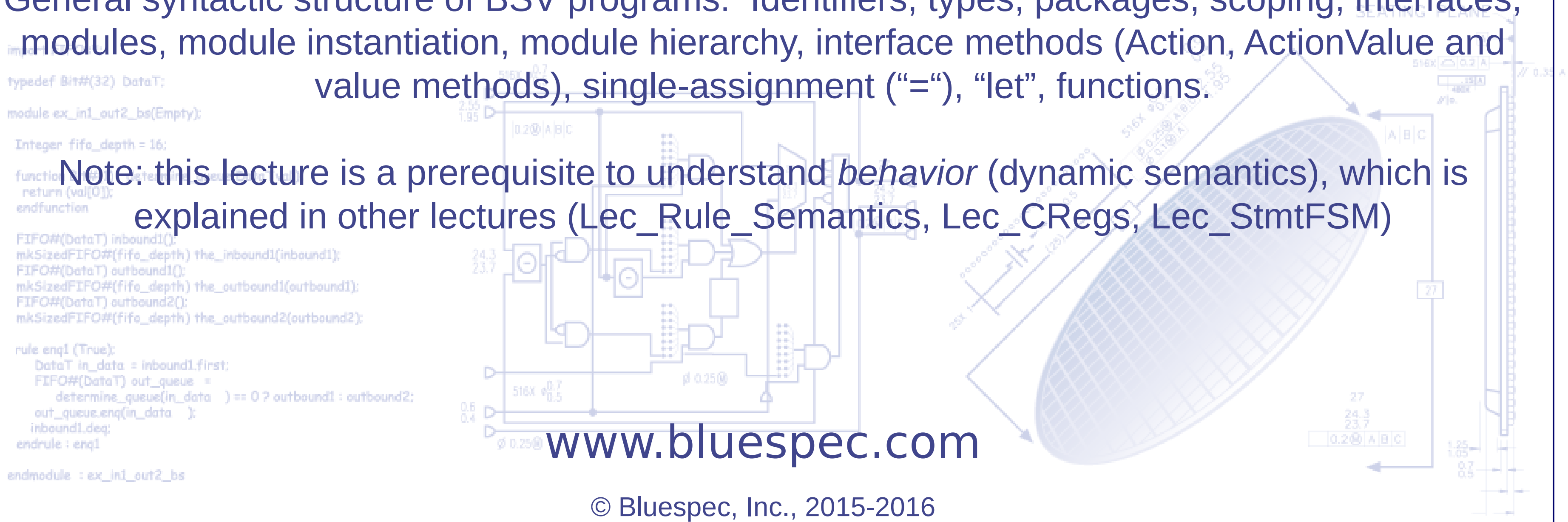
BSV Training

Lec_Basic_Syntax

General syntactic structure of BSV programs: Identifiers, types, packages, scoping, interfaces, modules, module instantiation, module hierarchy, interface methods (Action, ActionValue and value methods), single-assignment (“=“), “let”, functions.

Note: this lecture is a prerequisite to understand *behavior* (dynamic semantics), which is explained in other lectures (Lec_Rule_Semantics, Lec_CRegs, Lec_StmtFSM)



www.bluespec.com

Basic syntax elements, and identifiers

Basic syntax elements (identifiers, comments, whitespace, strings, integer constants, infix operators, etc.) all follow standard Verilog and SystemVerilog syntax.

Standard “static scoping” rules for identifier visibility and shadowing in nested scopes.

BSV identifiers are case sensitive.

The case of the first letter in an identifier is significant:

- Constants and Types begin with an uppercase letter: Int, UInt, Bool, True, False, ...
- Variables and type variables begin with a lowercase letter (type1, x, y, w, mkMult, ...)

Two exceptions, for legacy sake: the types ‘int’ and ‘bit’

- *These are familiar types from Verilog*
- *We recommend the equivalent standard BSV syntax Int#(32) and Bit#(1) instead*

Module naming convention: in all our examples we use names like
“mkTestbench” and “mkMult”

- The “mk” prefix is pronounced “make” and reinforces the idea that, as in Verilog, a module declaration is a generator, i.e., the module can be *instantiated* multiple times, each time providing a fresh instance.

- This is just a convention, for style and readability (not a syntax rule)

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:

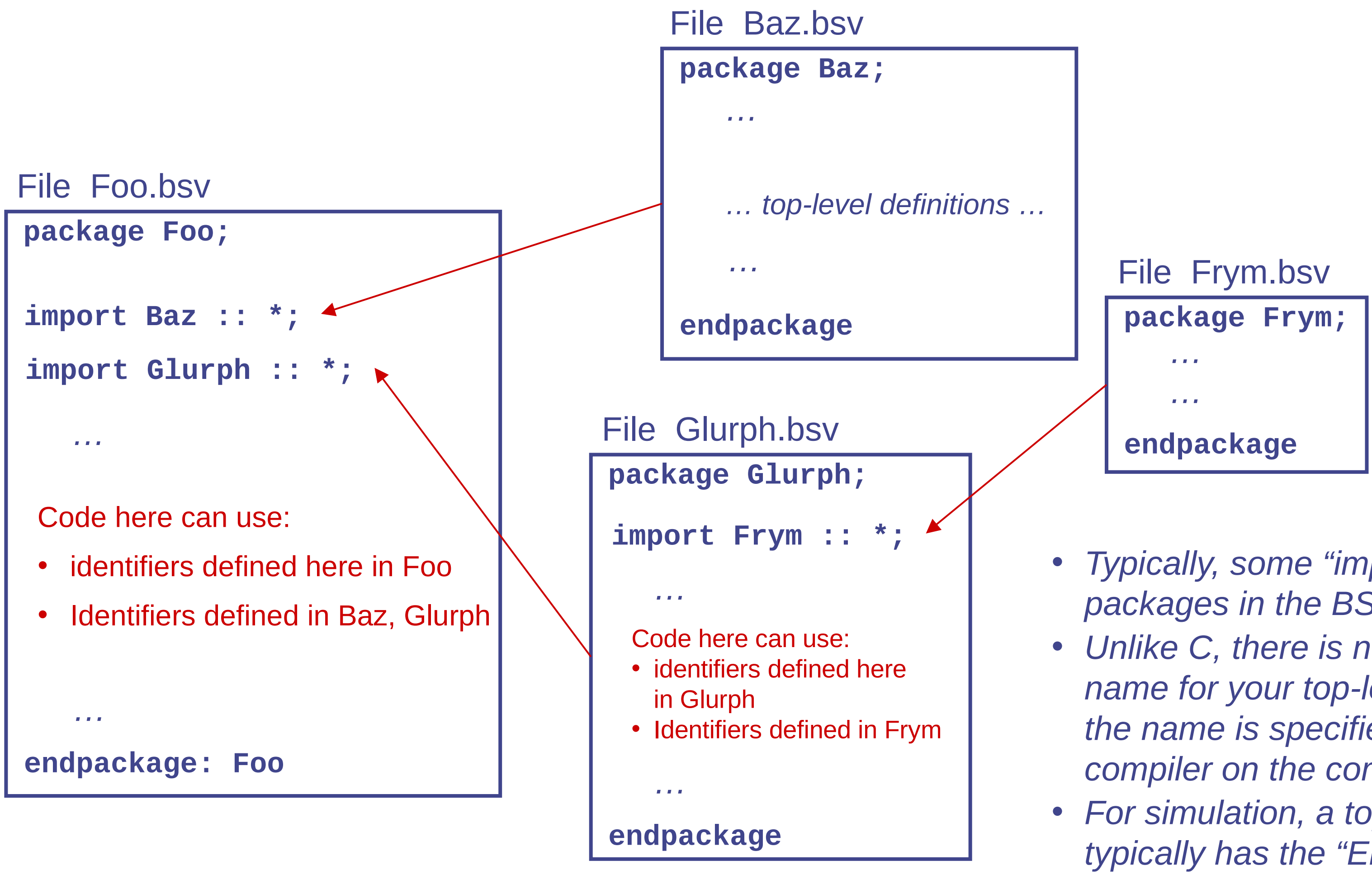
Type	*Comments*
Integer	Unbounded signed integers (static elaboration only)
Int#(18)	18-bit signed integers Note: 'int' is a synonym for Int#(32)
UInt#(42)	42-bit unsigned integers
Bit#(23)	23-bit bit vectors Note: 'bit[15:0]' is a synonym for Bit#(16)
Bool	Booleans, with constants True and False
Reg#(UInt#(42))	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

Overall syntactic structure of a BSV program

A complete BSV program is a collection of files (each file is a BSV “package”).

A package Foo may “import” another package Baz, making the top-level identifiers defined in Baz visible (and usable) in Foo.



File Baz.bsv

```
package Baz;
   …


   … top-level definitions …

   …
endpackage
```

File Foo.bsv

```
package Foo;

import Baz :: *;
import Glurph :: *;

   …
```

Code here can use:

- identifiers defined here in Foo
- Identifiers defined in Baz, Glurph

```
   …
endpackage: Foo
```

File Frym.bsv

```
package Frym;
   …
   …
endpackage
```

File Glurph.bsv

```
package Glurph;

import Frym :: *;

   …
```

Code here can use:

- identifiers defined here in Glurph
- Identifiers defined in Frym

```
   …
endpackage
```

- *Typically, some “imports” refer to packages in the BSV library*
- *Unlike C, there is no special “main” name for your top-level module; the name is specified to the **bsc** compiler on the command line*
- *For simulation, a top-level module typically has the “Empty” interface*

What’s in a package?

File name and package name must match; *bsc* will complain if they do not.

The package/endpackage lines are optional; *bsc* will use the filename if they are absent.

File Foo.bsv

```
package Foo;

import Baz :: *;
export a, b;

typedef struct {…} S;

interface Foo_IFC;
  …
endInterface

UInt #(16) a = 23;

function int f (int x);
  …
endfunction

module mkFoo (…);
  …
endmodule

endpackage: Foo
```

import/export statements

type declarations

interface declarations

value (constant) declarations

function declarations

module declarations

Trailing “: Foo” is optional, and must match the name at the top

Namespace control with imports and exports

File Foo.bsv

```
package Foo;

import Baz :: *;

import Glurph :: *;

   …
```

Code here can use:

- identifiers defined here in Foo
- B_ifc, mkB and Baz::x from Baz
- Glurph::x from Glurph
- identifiers defined Frym

```
   …
endpackage: Foo
```

Baz only exports the three identifiers B_ifc, mkB and x; all other identifiers in Baz are *private* to Baz.

File Baz.bsv

```
package Baz;

export B_ifc, mkB, x;
   …
interface B_ifc;
   …
endinterface
   …
module mkB (…);
   …
endmodule
   …
UInt #(16) x = 42;
   …
endpackage
```

Glurph exports its own identifier x, and also *re-exports* everything it imports from Frym.

File Glurph.bsv

```
package Glurph;

import Frym :: *;

export Frym :: *, x;
   …
Bool x = False;
   …
endpackage
```

With no export statement, *all* of Frym's identifiers are exported.

File Frym.bsv

```
package Frym;
   …
   …
endpackage
```

Separate compilation

The *bsc* tool processes each file separately:

- Parsing, typechecking, name resolution, and certain other things
- It generates code separately (for Bluesim or Verilog generation) only for modules marked with the “(* synthesize *)” attribute (or, equivalently, named with the “-g” flag on the command line)

We recommend using the “(* synthesize *)” attribute liberally, i.e., on as many modules as possible:

- It can speed up compilation since modules are analyzed and compiled separately, and because it enables incremental compilation (bsc does not have to recompile already compiled modules whose source files have not changed)
- The greater retention of structure into the compiled code provides more clarity when debugging and viewing waveforms

What’s in an interface declaration?

interface name

type parameters

```
interface Foo_IFC #(numeric type n, type t);

   method Action m1 (int x, Bool y);

   method ActionValue #(int) m2 (… args …);

   method int m3 (… args …);

   interface Put #(int) i4;
   …
endInterface
```

Action method declarations

(methods can have arguments)

ActionValue method declaration

Value method declarations (return type is not Action or ActionValue)

sub-interface declarations

What’s in a module declaration?

module name

module parameters

module interface type

```
module Foo_IFC #(int n) (Foo_IFC);

   UInt #(16) a = 23;

   Reg #(int) x <- mkReg (10);
   Switch #(4,4) switch <- mkSwitch;

   function int f2 (…);
      …
   endfunction

   rule rl_r1 (…);
      …
   endrule

   method Action m1 (int x, Bool y);
      …
   endmethod

   interface i4;
      …
   endinterface
endmodule
```

Value (constant) declarations and definitions

Module instantiations

Function declarations

Rules

Method definitions

Sub-interface definitions

bluespec

What’s in a rule?

rule name

rule condition (“explicit condition”)

```
  rule rl_req_A (rg_running && (rg_nA < rg_n) && (rg_nA == rg_nB));

    Req_I req = Req {command:READ, addr:rg_addr_A};

    f_memReqs.enq (req);
    rg_nA <= rg_nA + 1;
    rg_addr_A  <= rg_addr_A + 4;
endrule
```

Value definitions

(“single-assignments”)

An Action
(here, an Action method invocation)

Two Actions
(here, register assignments, which are also just Action method invocations with special syntax)

Note: rule and method bodies can also contain conditionals and generative loops.

- *A rule condition is always an expression of type Bool*
  - *(Therefore, by BSV strong type-checking rules, it cannot have a side-effect, i.e., it cannot contain an Action!)*
- *The rule body is an expression of type Action*
  - *The overall Action of a rule body may, in turn, be consist of smaller Actions, which, in turn, may contain smaller Actions, and so on (Action is a recursively defined type)*

What’s in a method definition?

method name

method condition (“implicit condition”)

```
method ActionValue#(t) get () if (xs[0]==1 && (rg_inj == n));
   let new_xs = shiftInAtN (readVReg (xs), tagged Invalid);
   writeVReg (xs, new_xs);
   if (xs[1] == 1)
      rg_inj <= 0;
   return x0;
endmethod
```

Value definitions
(“single-assignments”)

Actions
(only in Action and ActionValue methods, not in Value methods)

return statements (only in Value and ActionValue methods; not in Action methods)

Note: rule and method bodies can contain conditionals and generative loops.

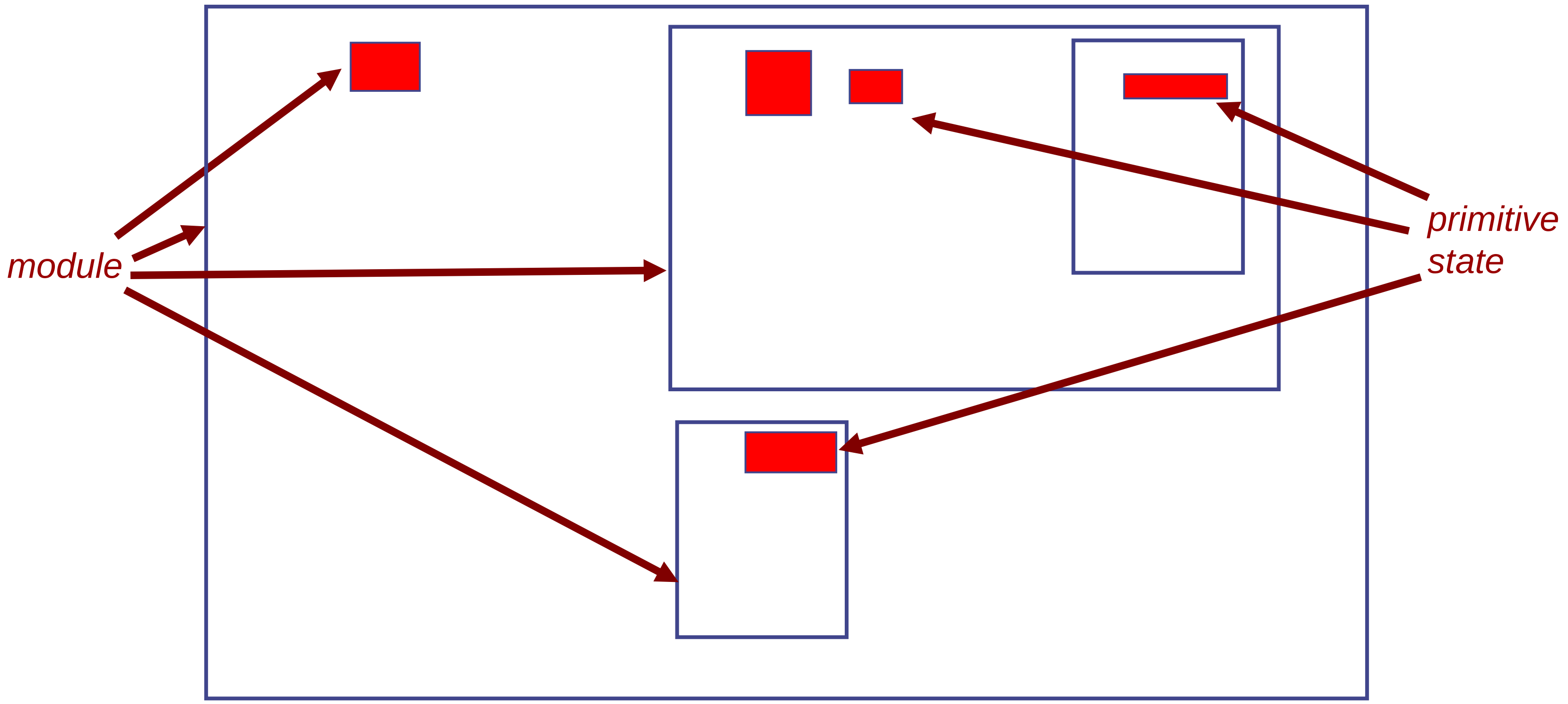
- *A method condition is always an expression of type Bool*
  - *(Therefore, by BSV strong type-checking rules, it cannot have a side-effect, i.e., it cannot contain an Action!)*
- *For ActionValue methods, the method body contains Actions (like a rule body), and a ‘return’ statement for the return value*
- *For Action methods, the method body contains Actions (like a rule body)*
- *For value methods, the method body is a begin-end block with a ‘return’ statement*

Circuit structure: module hierarchy and state

A BSV design consists of a *module hierarchy* (just like in Verilog, SystemVerilog and SystemC)

The leaves of the hierarchy are “primitive” state elements, including registers, FIFOs, etc.

Even registers are (semantically) modules (unlike in Verilog, SystemVerilog, ...).



All “primitives” in BSV are in fact implemented in Verilog and “imported” using BSV’s standard import mechanism. Hence, you can easily create new primitives or import existing Verilog IP.

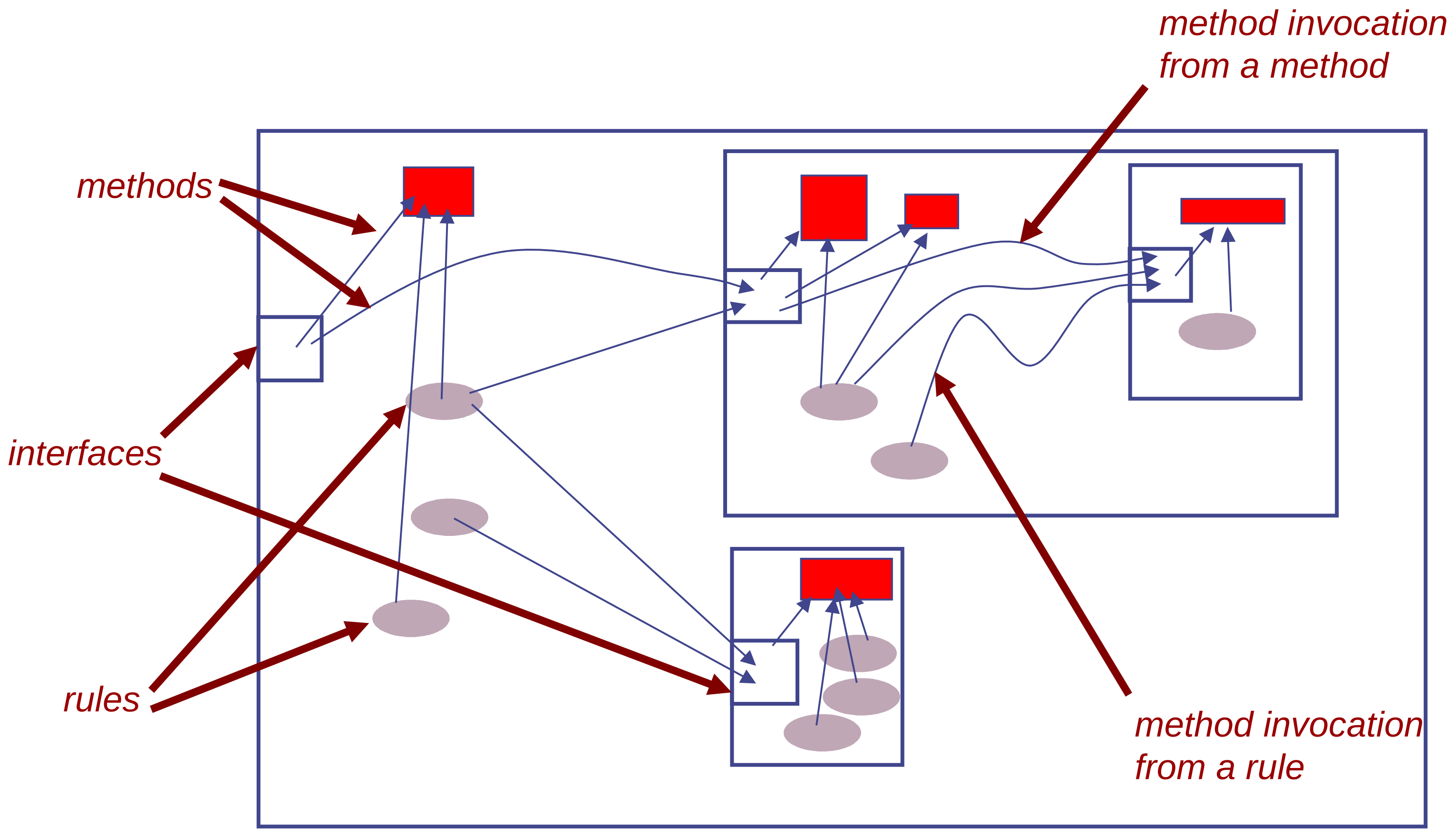
Rules and interface methods

Modules provide interfaces, which contain *interface methods.*

Modules contain rules, which use methods in other modules.

All inter-module communication is via methods (object-oriented)

A method can itself use methods of other modules.



method invocation from a method
methods
interfaces
rules
method invocation from a rule

Even registers are modules …

Registers are just modules with the following interface:

```
interface Reg #(t);
   method Action _write (t v);
   method t      _read  ();
endinterface: Mult_ifc
```

Where "t" is "int" or "Bool" or some other type

Following standard BSV syntax, a register update would look like this:

```
x._write (x._read () << 1);
```

But, for convenience, the BSV compiler allows you to omit "._read()" from register reads, and will insert it for you. So, our update becomes:

```
x._write (x << 1);
```

Further, for convenience, BSV provides special syntax using the conventional "<=" for register assignment for "._write()". So, our update becomes:

```
x <= x << 1;
```

Module instantiation

Module instantiation has the following syntax:

interface_type *instance_name* **<-** *module_name* **(** *module_parameters* **);**

"(*module_parameters*)" can be omitted if a module has no parameters.

Examples:

```
    Mult_ifc m <- mkMult;
```

```
    Reg#(int)  w     <- mkRegU;
    Reg#(Bool) got_x <- mkReg (False);
```

`mkRegU` is a module with no parameters; the register's initial (reset) value is unspecified.

`mkReg` is a module with a parameter for the register's initial (reset) value.

Interfaces: introduction

- All inter-module communication in BSV is expressed using *interface methods*
  - BSV is completely “transactional”, or “object-oriented”
  - (Traditional “in” and “out” signal ports are just a special case!)

- Every module *provides* an interface, i.e., it has an interface whose methods are invoked by other modules that need to communicate with it
  - This is always specified as an *interface type*

- Methods can be viewed just as rule fragments—they have the same features as rules, with the same semantics:
  - Conditions and bodies (which can be Actions)
  - Thus, interfaces and methods are a way to organize a large design into smaller, manageable, reusable modules

Modules and interfaces: example

```
interface FIFO #(type  any_t);
    method Action   enq (any_t  x);
    method any_t    first;
    method Action   deq;
    method Action   clear;
endinterface: FIFO
```

Interface declaration

(FIFO#(t) is a standard interface in the BSV library)

Module that *provides* a FIFO interface

```
module mkFIFO (FIFO#(some_t));
   …
endmodule
```

Instantiating modules with FIFO interfaces

Using FIFO interface methods

```
module mkBaz (…);
  FIFO#(int) f1 <- mkFIFO;
  FIFO#(int) f2 <- mkFIFO;
  ...
  ...
  rule r1 (f1.first() > 8);
      f2.enq (f1.first() + 2);
      f1.deq;
  endrule
endmodule
```

Three kinds of methods, depending on return type

```
interface FIFOwPop #(type t);
    method Action              enq (t x);
    method t                   first;
    method ActionValue#(t)  pop;    // combines first and deq
endinterface: FIFOwPop
```

- *Value* methods: Takes 0 or more arguments and has a return type other than Action or ActionValue. E.g., ‘first’.
  - BSV’s type discipline guarantees that it cannot have a side-effect
  - It is always purely combinational

- *Action* methods: Takes 0 or more arguments and has Action type. Represents a pure side-effect inside the module. E.g., ‘enq’

- *ActionValue* methods: Takes 0 or more arguments and has ActionValue type. Represents a side-effect inside the module, and also returns a value. E.g., ‘pop’

Rule and method conditions cannot have side-effects (they’re “pure”).

Thus, Action and ActionValue methods can only be used in rule bodies, and bodies of other Action and ActionValue methods. Value methods can be used anywhere, including in rule and method conditions.

Using ActionValue methods

```
module mkFIFOwPop (FIFOwPop#(int));
   …
endmodule
```

```
module mkFoo (…);
  FIFOwPop#(int) f1 <- mkFIFOwPop;
  FIFOwPop#(int) f2 <- mkFIFOwPop;
  …
  rule r1 (…);
      int x <- f1.pop;
      f2.enq (x + 2);
  endrule
endmodule
```

Note this assignment symbol

In both cases, the right-hand side of the assignment has a side-effect and returns a value:

- The first one is a side-effect *during static elaboration*: it creates a module and returns its interface
- The second one is a side-effect *during dynamic execution*: it pops an element from the FIFO and returns it

Defining methods in modules

```
module modName [#(type arg,…)] ( IfcType);
  …
  …
  method [ type ] methodName1 (arg, …, arg) [ if ( cond ) ];
    … method body …
  endmethod
  …
  method [ type ] methodNameN (arg, …, arg) [ if ( cond ) ];
    … method body …
    return expr          // if it is a value method
  endmethod
endmodule
```

- All the method definitions of a module are written at the end of the module (after the sub-module instantiations, rules, etc.)
- The ‘if’ conditions become the implicit conditions of the methods

Interfaces are a means to *modularize* rules

- Action and ActionValue methods become a part of any rule from which they are invoked
  - The method condition (implicit condition) becomes a part of the rule condition
  - The method's actions become a part of the rule's actions

- Value methods also become a part of any rule from which they are invoked
  - The method condition (implicit condition) becomes a part of the rule condition

Thus,

- Modules, interfaces and methods can be viewed as a way to organize a large piece of behavior into smaller, localized chunks

- Interfaces do not have any separate semantics—they seamlessly integrate into standard rule semantics

Variables and “single-assignment” in BSV

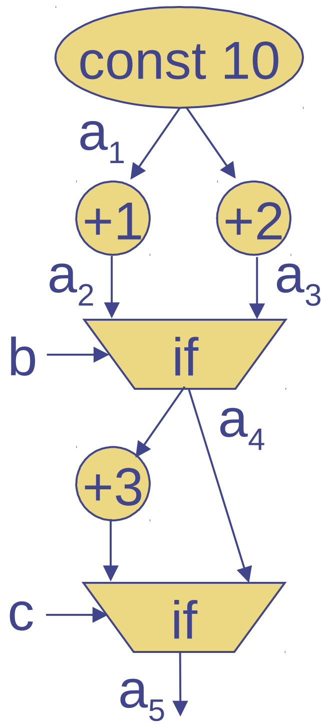
- Variables in BSV represent immutable values (as in mathematics)
  - Note: in C, C++, Verilog, etc., a variable represents a storage location that can be updated dynamically and repeatedly by assignment. In BSV, dynamic updates are *only* expressed with Action (including register assignment “<=”).

- Repeated syntactic assignment in BSV with “=” is just a notational device for incrementally describing more complex expressions
  - It’s like a new variable (e.g. with a new subscript) from that point on
  - I.e., it represents a new value over the “rest of the program text”, and is not associated with “time”
  - This concept is well-known as “static single-assignment” or (SSA) in functional programming languages (and inside most modern compilers)

```
int a = 10;

if (b) a = a + 1;
else  a = a + 2;

if (c) a = a + 3;
```

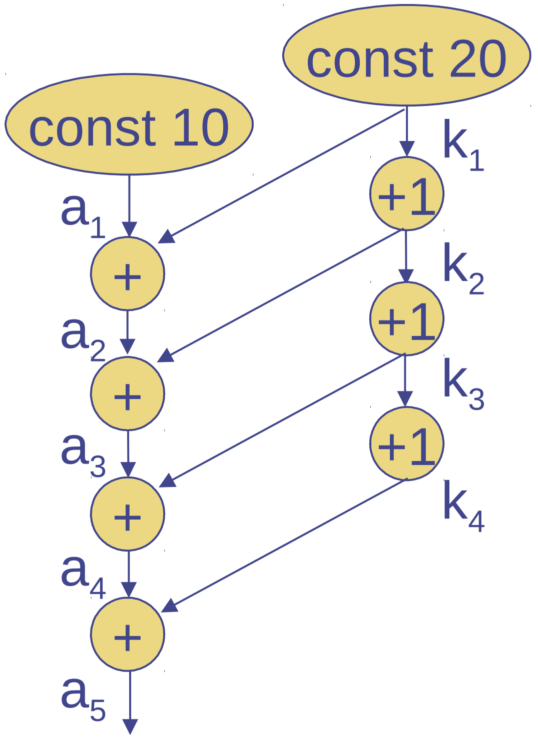


Ordinary variable assignment: =

Another example, involving a statically elaborated loop

- *[Compiler gurus: this is just “SSA form” (static single assignment)]*

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



bluespec

Assignment symbols: =, <- and <=

‘=‘ is used just for variable assignment. It is purely static. Example:

```
Int#(124) b = myMemory.lookup(addr);
```

‘<=‘ is used as a syntactic shorthand for _write method invocation. Example:

```
rule doIt;
   myRegA <= 32;
endrule
```

Note: “<=“ is also used inside expressions for the “less than or equal to” operator. This is always unambiguous due to context.

‘<-‘ is used at the top-level of modules for module instantiation:

```
Reg#(Bit#(32)) myRegA <- mkRegA(0);
```

‘<-‘ is used in rules and methods to invoke an ActionValue and assign its returned value:

```
rule doIt;
   let a <- myFIFO.pop;
   ...
endrule
```

“let”

- BSV has a “let” statement by which you can declare a variable and initialize it, with the compiler deducing the type of the variable based on the initial value expression
- It can reduce clutter, especially when the type is “obvious” from the init value

```
type var;
var = init;
```

Equivalent to declaring and defining a variable, except that the type is automatically inferred by the compiler from the “init” expression;

```
let var = init;        // syntax

let x = 24’h9BEEF;

let y = x + 3;

let z = MyStruct {memberA: exprA, memberB: exprB};
```

Compiler infers type: **`Bit#(24)`**

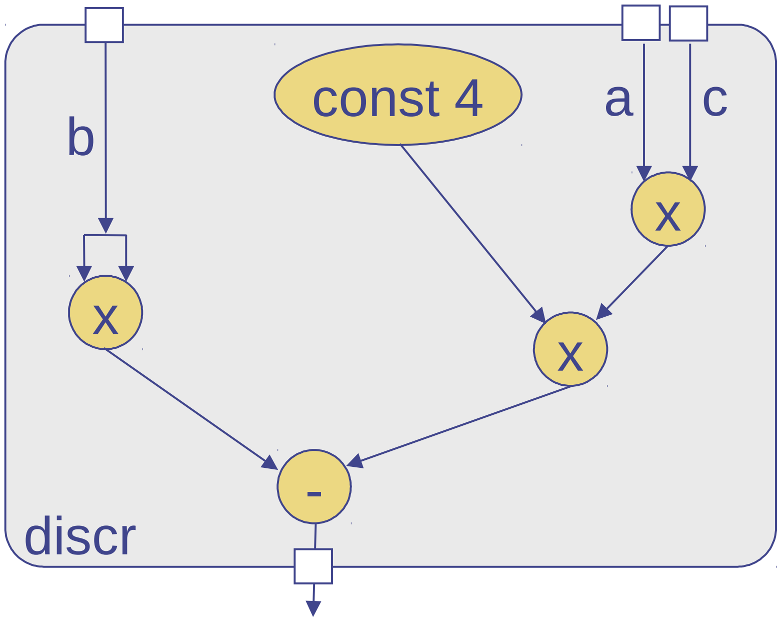
Compiler infers type: **`Bit#(24)`**

Compiler infers type: **`MyStruct`**

Functions

- Functions in BSV have no run-time semantics. They are purely a static syntactic convenience, and are expanded in-line during static elaboration
  - No “stack frame”, no “call and return”
  - Think of them as circuits plugged into place wherever used

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



bluespec

End

Questions?

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