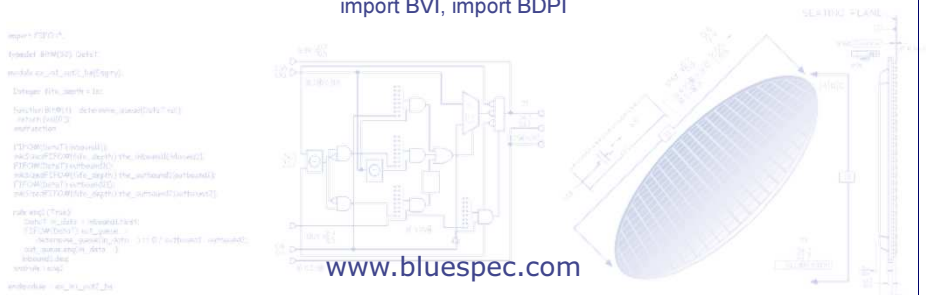
bluespec

BSV Training

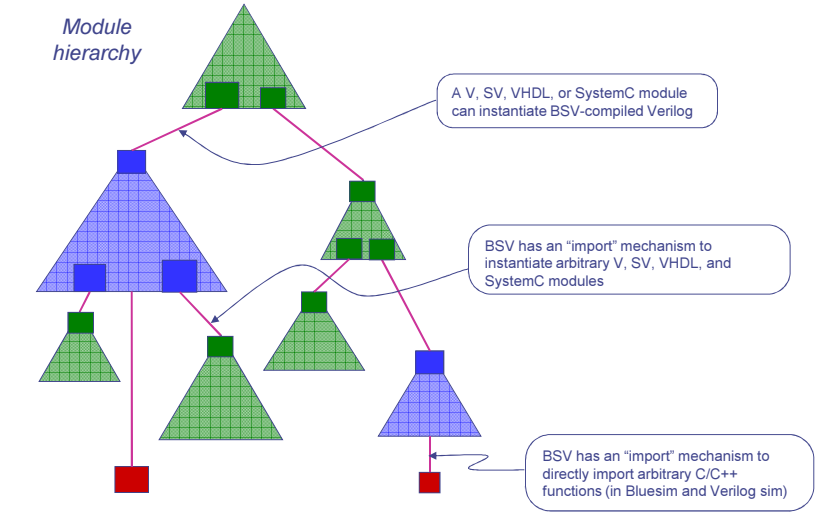
Section: Importing Verilog and C into BSV

import BVI, import BDPI



www.bluespec.com

BSV interoperates with V, SV, VHDL, SystemC, and C/C++



Legend

- V (Verilog), SV (SystemVerilog), VHDL, or SystemC (event-driven)
- BSV
- C, C++

2

bluespec

Plugging RTL (or RTL-like SystemC) into BSV

bluespec

When does BSV interface to RTL (V, SV, VHDL)?

Plugging RTL into BSV:

- Many BSV designs are used in projects where there is already some existing verified RTL IP, which we simply wish to re-use
- Users may wish to add a new "primitive" to BSV that is important for an application domain
  - In fact, all BSV "primitives" are imported this way—knowledge about primitives is not built into the *bsc* compiler

This is done using BSV's "import BVI" capability

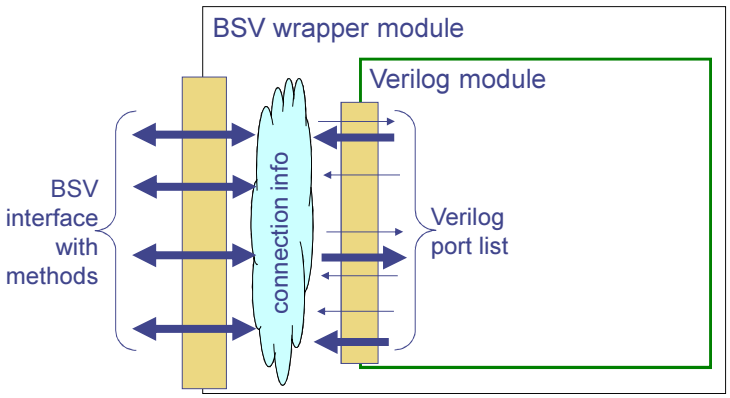
Plugging BSV into RTL:

- A BSV design may fit into a larger RTL design
- A BSV design may be verified in an existing verification environment that is based on SystemVerilog (e.g., VMM), 'e', etc.

This is done using by scheduling and naming control on the top-level interface, to be compatible with the RTL environment into which it fits

bluespec

Basic structure of importing a Verilog component



- Define the BSV interface type that the module should provide to its environment
- Use the "import BVI" mechanism to define a wrapper module, which will:
  - Describe how the interface method arguments, results, enables, clocks, resets, etc. connect to the Verilog module ports
  - Describe the BSV *scheduling constraints* on the methods

bluespec

Example: importing a MAC (multiply-accumulate) module

The Verilog module we wish to import:

```
module mymac(EN, a, b, clear_value, clear, out, clk, rst_b);

input a, b, EN, clear, clear_value, clk, rst_b;
output out;

reg  [15:0] out;
wire [15:0] a, b, clear_value;

always@(posedge clk or negedge rst_b)
if (!rst_b)
   out <= 0;
else
   out <= clear ? clear_value : (EN ? out+a*b: out);
endmodule
```

- when EN == 1, out <= out + a * b
- When clear == 1, out <= clear_value

bluespec

Define the desired BSV interface and the wrapper

```
interface Mac_IFC ;
   method Action acc (Int#(16) a, Int#(16) b);
   method Action reset_acc (Int#(16) value);
   method Int#(16) read_y;
endinterface
```

```
import "BVI" mymac =
module mkMac (Mac_IFC);

  default_clock clk (clk);
  default_reset reset (rst_b);

  method  acc(a, b)       enable(EN);
  method  reset_acc(clear_value)      enable(clear);
  method  out read_y();

  schedule (read_y) SB (reset_acc, acc);
  schedule (acc) C (reset_acc);

endmodule
```

Clock and reset

Methods

Scheduling constraints

bluespec

Notes on the import Verilog mechanism

The previous slides showed only a simple example. The Reference Guide Sec. 15 goes into a lot more detail:

- Connecting clocks and resets
- Connecting method ENABLE and RDY signals
- The method-to-RTL connections need not be 1-to-1, and can involve other logic
- The available scheduling annotations
- Caveat: the schedule is just an assertion by you taken at face value by the compiler when using this module in your BSV program; the compiler makes no attempt to 'verify' that the scheduling assertions are sensible

The actual Verilog code is not touched by the *bsc* compiler. The *bsc* compiler simply generates a Verilog instantiation of the module, and your RTL simulator finds and instantiates the Verilog module

These import mechanisms can also be used for importing VHDL and SystemVerilog, and RTL-style SystemC

- Most RTL simulators allow free intermixing of Verilog, VHDL, SystemVerilog and SystemC

bluespec

What about Bluesim, when you import RTL?

Bluesim does not currently support "co-simulation" of Bluesim with a Verilog simulator

- If you just import a Verilog module, your only simulation option is Verilog simulation, i.e.,
  - Use *bsc* to generate Verilog from the BSV code
  - Use a Verilog simulator to simulate all the Verilog code (BSV-generated, and imported)

How do the primitives in the BSV library work in Bluesim?

- Every primitive in the BSV library is implemented both in Verilog and in C
  - The C code is imported using the "import BDPI" mechanism (described later), into a module with exactly the same BSV interface
- For every primitive, we have a wrapper module that instantiates one or the other, like this:

```
IfcType m <- if (genVerilog) mkVerilogWrapper;
             else mkCWrapper;
```

- "genVerilog" is a built-in boolean which is true when compiling for Verilog and False when compiling for Bluesim (Reference Guide Sec. B.6). Thus, at compile time, *bsc* chooses the appropriate instantiation (i.e., this is a "static elaboration" choice).

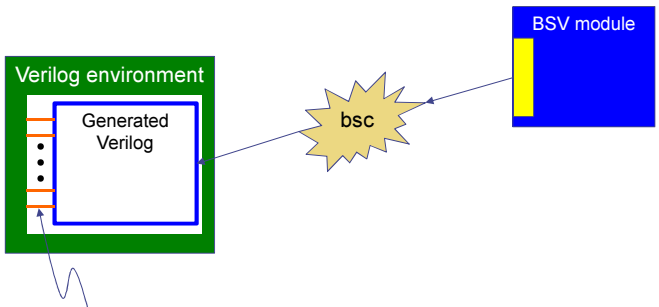
bluespec

Plugging BSV into RTL

bluespec

Plugging BSV into RTL: key issues

When plugging BSV into RTL (more accurately: BSV-generated RTL into RTL), the key issues are:



We need:

- Exactly the right set of input and output signal ports, with exactly the right port names, as expected by the RTL environment
- Exactly the right signaling protocols on these ports, as expected by the RTL environment (may be different from standard BSV method protocol of RDY and EN signal)

Example: the "Verilog environment" may be an AMBA AXI bus port

bluespec

Exposing method conditions

- A method's condition can always be exposed as an explicit Bool method
- The examples below are from the BSV library
  - Note: in the lib FIFOF, enq, first and deq are still guarded by their implicit conditions; the notFull and notEmpty methods are just additional methods

```
interface FIFO#(type t);
  // enq has "notFull" condition
  method Action          enq(t x);
  // first/deq have "notEmpty" condition
  method t               first;
  method Action          deq;
  method Action          clear;
endinterface: FIFO
```

```
interface FIFOF#(type t);
  // enq has "notFull" condition
  method Action   enq(t x);
  // first/deq have "notEmpty" condition
  method t        first;
  method Action   deq;
  method Action   clear;
  method Bool     notEmpty;
  method Bool     notFull;
endinterface: FIFOF
```


bluespec

Exposing method conditions

- Implementing exposed conditions is trivial: just replicate the method condition
  - E.g., below, enq()'s condition canEnque is returned explicitly as a Boolean in notFull()
  - Don't worry: the generated code will share a single instance of the condition logic
    - Or, share it explicity by writing let x = canEnque and using x twice

```
module mkFIFOF (FIFOF#(type t));
  …
  method Action enq(t x)        if (canEnque); …  endmethod
  method t          first       if (canDeque); …  endmethod
  method Action  deq            if (canDeque); …  endmethod

  method Bool    notFull;     return canEnque;  endmethod
  method Bool    notEmpty;  return canDeque; endmethod
endmodule: mkFIFOF
```


bluespec

Removing RDY signals

- The "always_ready" attribute can be applied to a method to indicate
  - that it is always ready (the compiler will check this!)
  - that the RDY signal must be removed from the generated Verilog

```
(* always_ready = "add, result" *)
module mkAdder (Adder);
  Reg#(int) acc <- mkRegA(0);

  method Action add(a, b);
    acc <= a + b;
  endmethod

  method int result;
    return acc;
  endmethod
endmodule
```

```
interface Adder;
  method Action  add(int a, int b);
  method int         result;
endinterface
```

The compiler will check that add and result are always ready to be invoked, i.e. their implicit and explicit conditions are always True

The compiler will not generate any RDY signal for add and result

bluespec

Removing ENABLE signals

- The “always_enabled” attribute can be attached to a method (Action or ActionValue) to indicate
  - that it is assumed True, i.e., the data input signals (args) are driven by the environment on *every* cycle
  - Of course, this implies “always ready”, i.e., the method’s condition must be always True (otherwise it would be an error to drive the EN signal)
  - that the EN signal must be removed from the generated Verilog

```
(* always_enabled = “add” *)
module mkAdder (Adder);
  Reg#(int) acc <- mkRegA(0);

  method Action add(a, b);
    acc <= a + b;
  endmethod

  method int result;
    return acc;
  endmethod
endmodule
```

```
interface Adder;
  method Action  add(int a, int b);
  method int     result;
endinterface
```

The compiler will check that add can be always invoked, i.e. its method condition is always True

The compiler will not generate any EN signal for add

bluespec

Unguarded methods

- Suppose a module must read datum on EVERY CYCLE

```
(* always_enabled = “accept” *)
module mkFoo (IfcFoo);
  FIFO#(int) fifo <- mkFIFO;
  …
  method Action accept (int);
    fifo.enq(bus);
  endmethod
endmodule
```

The bsc compiler will signal an error, since the enq() method may not be ready (the fifo may be full).

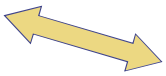
It cannot verify that accept() is always ready, so it cannot allow the always_enabled spec

bluespec

Unguarded methods

- Answer: use "unguarded" methods, i.e., methods with no conditions, relying on you to guard them explicitly
  - The BSV library provides FIFOs with unguarded methods
  - Warning: these are dangerous! Use with care, only in these "impedance matching" situations

```
(* always_enabled = "accept" *)
module mkFoo (IfcFoo);
  FIFO#(int) fifo <- mkUGFIFOF;
  …
  method Action accept (int);
    fifo.enq(bus);
  endmethod
endmodule
```



This will work.

Warning: you have to use some other means to avoid buffer overflow!

bluespec

Summary: plugging BSV into RTL

- A BSV Action method with one argument that is always ready and always enabled becomes, exactly, a Verilog input port. E.g.,

```
(* always_ready, always_enabled *)
method  Action  accept (int);
```

- A BSV value method that is always ready becomes, exactly, a Verilog output port. E.g.,

```
(* always_ready *)
method  int  yield_data;
```

- Sec. 13.2.1 shows attributes by which you can control the exact names of these generated Verilog ports

With these capabilities, and unguarded methods, you can produce a Verilog interface with any desired ports, with any desired names, and with any desired behavior (protocol)
- *The BSV AXI library has examples of interfaces to the AMBA AXI bus (masters, slaves, etc.)*

bluespec

Plugging C/C++ into BSV

bluespec

Motivations for importing C/C++

- Many applications begin life as C/C++ models:
  - When you are creating a HW accelerator for an existing SW program
  - When you are trying to choose the best algorithm before going to HW implementation

  As you incrementally develop your BSV implementation, you may wish to reuse C/C++ components temporarily (or even permanently, for the testbench)

- When you add a new "primitive" to BSV by importing Verilog, you may also want to import a corresponding C model for running in Bluesim
  - In fact, all BSV "primitives" are imported this way—the knowledge is not built into the compiler

bluespec

Importing C

- You declare a BSV function prototype (i.e., just the types) which is then implemented in C
- Example:

```
// BSV code

import "BDPI" rand32 =
   function Bit#(32) rand32();

module test(Empty);
 FIFO#(Bit#(32)) myFIFO <- mkSizedFIFO(9);

 rule fill;
    myFIFO.enq(rand32);
 endrule

 rule empty;
    myFIFO.deq;
    $display("Number %d", myFIFO.first);
 endrule
endmodule
```

```
// C code
#include <stdio.h>
#include <stdlib.h>

unsigned int rand32()
{
   return (unsigned int) rand();
}
```


bluespec

Importing C: arguments and results

- There is a 1-to-1 correspondence between the arguments and result in the BSV prototype and in the C function.
- Allowed arguments types
  - any type that is in the Bits#() typeclass
  - String
  - polymorphic types
- Allowed result types:
  - any type that is in the Bits#() typeclass
  - Action
  - ActionValue#(t) where t is in the Bits#() typeclass
  - polymorphic types (and polymorphic ActionValue #(t) types)
- For "small" types (< 64b), the corresponding C argument or result is the nearest larger C scalar integer type (char, short, long, long long)
- For "larger" types and polymorphic types, the C function receives a "void *" pointer to the storage for that value
  - This storage contains the "raw bits" of the BSV representation
  - For return types that are passed using "void *" like this, the "void *" pointer is passed as the first argument of the C function, not returned as the C function value
- Details and examples in the Reference Guide, Sec. 16

bluespec

Linking in the imported C function(s)

- Compilation of an "import BDPI" produces intermediate ".ba" files

```
# bsc -u -sim DUT.bsv
checking package dependencies
compiling DUT.bsv
Foreign import file created: compute_vector.ba
code generation for mkDUT starts
Elaborated Bluesim module file created: mkDUT.ba
code generation for mkTB starts
Elaborated Bluesim module file created: mkTB.ba
```

- In the link stage (for Bluesim or Verilog sim, you supply these .ba files and the C file

```
# bsc -sim -e mkTB -o bsim mkTB.ba mkDUT.ba compute_vector.ba vectors.c
Bluesim object created: mkTB.{h,o}
Bluesim object created: mkDUT.{h,o}
Bluesim object created: schedule.{h,o}
User object created: vectors.o
Bluesim binary file created: bsim
```

When you compile for Verilog sim, bsc generates the appropriate "VPI"-like linkage files that allows your Verilog simulator to import the C code (for most popular simulators like VCS, NCSim, Modelsim, iVerilog, CVC, etc.)

bluespec

Importing C++ (as opposed to C)

The BSV "import BDPI" mechanism can only directly invoke C functions (it assumes function linkage conventions for C, and C++ typically has different linkage conventions).

C++ has an 'extern "C"' construct that tells the C++ compiler to compile a function with C linkage instead of C++ linkage. That function, in turn, can freely call C++ functions.

Thus, the following example illustrates the idiom for calling C++ from BSV:
- We using "import BDPI" to invoke a function myMainC_function() which has C linkage
- This, in turn, calls the desired C++ function myMainCPP_function():

```
// File foo.cpp    (C++)

//  This is the C++ function we would like to invoke from BSV
int myMainCPP_function(int a, int b) {
   … C++ code …
}

extern "C" {   //  This is the function imported into BSV
   void myMainC_function (int a, int b)
   {
        return myMainCPP_function (a, b);
   }
}
```


bluespec

Plugging BSV into SystemC

bluespec

Plugging BSV into SystemC

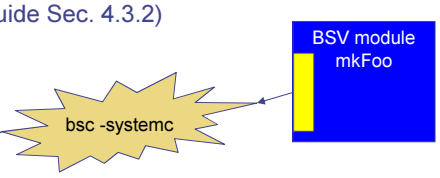
The *bsc* compiler directly supports, via a command-line flag, the creation of a "plug-in" into a SystemC program (User Guide Sec. 4.3.2)

```
// File mkFoo_systemc.h (SystemC)

SC_MODULE (mkFoo)
{
  public:
   sc_in<…> …;
   sc_out<…> …;
   …
  public:
   SC_CTOR (mkFoo) …
   ~mkFoo() …
}
```

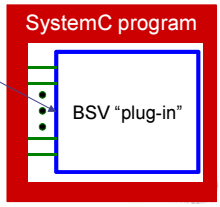
```
// File mkFoo_systemc.cxx
// C++ code (not SystemC)
// implementing the declarations
// in the .h file

…
```



The .h file is standard SystemC. It declares mkFoo as a SystemC SC_MODULE. The sc_in and sc_out ports are SystemC signal declarations, corresponding exactly to what you would have got if you had compiled the BSV code to Verilog.
You should "#include" this .h file in your SystemC program, which can then invoke the constructor mkFoo to instantiate the module, and use the sc_in and sc_out ports to communicate with it, in the usual SystemC way.

The .cxx file is compiled with your C++ compiler as usual, and linked in with the rest of your SystemC program's object files, to create the SystemC executable.



(Bluespec tests this facility with the standard OSCI SystemC simulator, but expects it to work with any SystemC simulator)

bluespec

End

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