



BSV Training

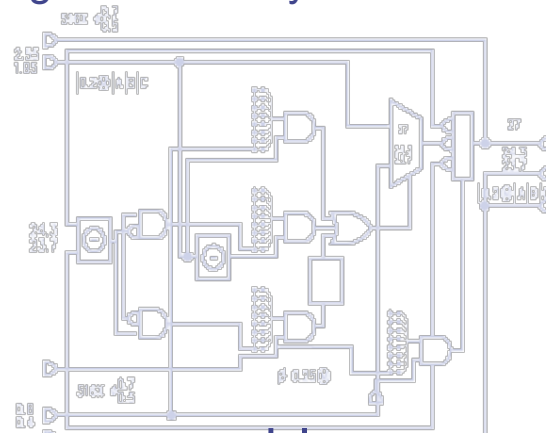
Lec_Interop_C

Importing C code into BSV (for simulation only) using “import BDPI”.
Exporting a BSV subsystem into a SystemC program

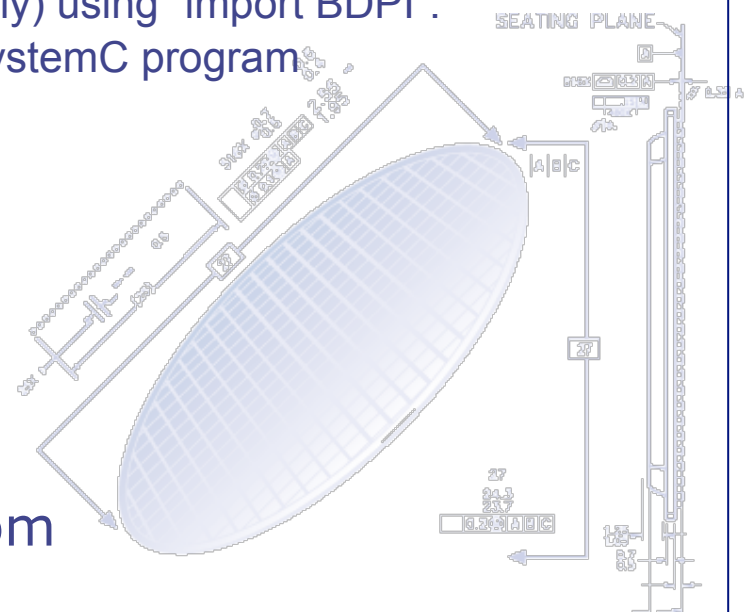
```

import PkF004;
typedef Bit2(24) (out0);
module ex_hdl_csr2_hdl{empty};
Integer nfa_depth = 16;
function Bit2(24) distnfa_pump(out0Tnd);
return (out0);
endfunction
PkF004(out0T) in0out0;
in0out0PkF004(nfa_depth) in0_in0out0(in0out0);
PkF004(out0T) out0out0;
in0out0PkF004(nfa_depth) in0_out0out0(out0out0);
PkF004(out0T) out0out0;
in0out0PkF004(nfa_depth) in0_out0out0(out0out0);
rule exp0 (True);
out0T in0_in0 = in0out0(in0);
PkF004(out0T) out0_out0 =
distnfa_pump(in0_in0) = 0 ? out0out0 : out0out0;
in0_out0out0 =
in0_out0out0;
endrule : exp0
endmodule : ex_hdl_csr2_hdl

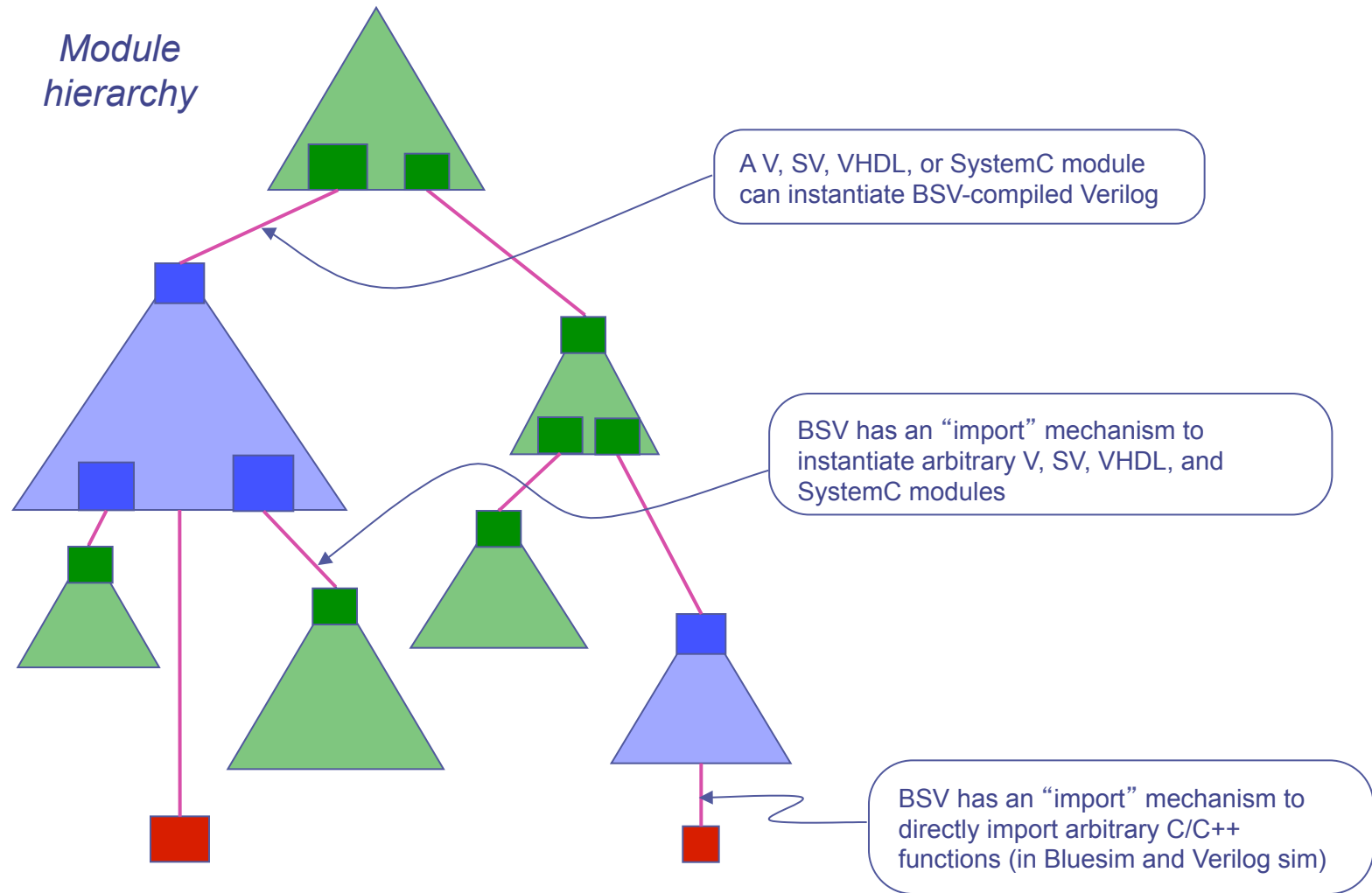
```



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++

Plugging C/C++ into BSV

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

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 ActionValue #(Bit#(32)) bsv_rand32();  
  
module test(Empty);  
  FIFO#(Bit#(32)) myFIFO <- mkSizedFIFO(9);  
  
  rule fill;  
    let x <- bsv_rand32;  
    myFIFO.enq(x);  
  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();  
}
```

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 ($\leq 64b$), the corresponding C argument or result is the nearest C scalar integer type adequate to contain it (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

Recommendation: arguments and results

- Although the ‘import C’ arguments and results can be of arbitrary type, it’s usually too much effort, and too error-prone, and too non-portable to exploit this capability
 - The representation of C types varies across C compilers and across target architectures (insertion of padding for word alignment, little- and big-endianness, etc.
- **Recommendation:**
 - a) Stick to just a few standard, highly portable types: 32b and 64b scalars, and arrays/vectors thereof.
 - In the C code, use standard types declared in `<stdint.h>`: `uint32_t`, `int32_t`, `uint64_t`, `int64_t`, and arrays of those types
 - In the BSV code, correspondingly, use `UInt#(32)`, `Int#(32)`, `UInt#(64)`, `Int#(64)`, and `Vector#(n,...)` of those types
 - b) In both the C code and the BSV code, write functions to convert from “native” types (enums, structs, unions) to standard types (a) and back.
 - Being purely in C and BSV, such conversion functions are easy to write and very predictable and reliable
 - E.g., convert a struct to and array of `uint32_t` or `uint64_t`
 - E.g., convert a C pointer to a `uint64_t` (pointers may not fit in `uint32_t`)
 - c) In both the C code and the BSV code, use these functions on arguments and results so that only these standard types (a) are passed between C and BSV

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.)

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);
    }
}
```

Plugging BSV into SystemC

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

...
```

bsc -systemc

BSV module
mkFoo

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.

SystemC program

BSV “plug-in”

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

Plugging BSV into SystemC

Note: Only the .h file is intended for human readability.

Although the .cxx file is C++ source code, it is a complex translation of the cycle-level behavior of the source BSV module and does not try to mirror the BSV source structure in any recognizable way.

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

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.

bsc -systemc

BSV module
mkFoo

SystemC program

BSV "plug-in"



End

[illegible]

Questions?

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

