Module hierarchy: same as in Verilog
- All state is explicit
  - Even registers are modules, and are instantiated as modules
- “Providing” and “Using” an interface
- Interfaces are types; interfaces are transactional
- Interfaces in the module hierarchy
- Synthesis boundaries and inlining
BSV module hierarchy: same as in Verilog

- Modules can instantiate other modules, resulting in a module hierarchy
  - Logically, the hierarchy forms a tree structure

```verilog
module mkM1 (...);
  ...
endmodule

module mkM2 (...);
  mkM1 m1a (...);
  // instantiates mkM1
endmodule

module mkM3 (...);
  mkM1 m1b (...);
  mkM2 m2 (...);
  // instantiates mkM1 mkM2
endmodule
```

Module Interfaces in SystemVerilog

- In Verilog, a module’s interface is its port list
  - The port list is replicated in all modules that provide the same interface (e.g., different implementations of the same FIFO interface)

- In SystemVerilog, it is possible to define an interface once, separately from all modules that provide that interface
  - SV interface ~ VHDL entity
  - SV module ~ VHDL architecture
Module Interfaces in BSV:
Terminology: “provides” and “uses”

- BSV imposes further discipline on interfaces
- Interfaces are structured into *methods*
  - Each method collects input and output ports along with the protocol on those ports, encapsulating a complete “transaction” that can be performed by a module for its external environment

- A module *provides* a particular interface
- Its external environment *uses* the interface

---

Every module uses the interface(s) just below it in the hierarchy.

Every module provides an interface to the module above it in the hierarchy.

- Module m1a provides interface i1a
- Module m2 uses interface i1a
- Module m2 provides interface i2
- Module m1b provides interface i1b
- Module m3 uses interface i1b and uses interface i2
- Module m3 provides interface i3
Module Interface Types

- In SV and BSV, interface definitions define a type
- In BSV, all interfaces are transactional, i.e., all communication is expressed in terms of interface methods
- The example below defines the type BusInterface
  - It contains two methods, sendRequest() and getResponse()
- Don’t worry about the details of the “method” declarations; we’ll discuss them later
- Just observe, for now
  - An interface is reminiscent of a “struct”, where each member is a method
  - A method declaration is reminiscent of a “function prototype”

```
interface BusInterface;
  method Action sendRequest (Request req);
  method ActionValue#(Response) getResponse ()
endinterface: BusInterface
```

Module headers and instantiation

- In the module header, specify the provided interface type in place of the formal port list
- When instantiating:
  - Use the SystemVerilog instantiation syntax:
    - Instantiate an interface
    - Instantiate a module, binding the interface instance in place of the actual port list
  - Or, use the BSV shorthand

```
module mkTop (TopInterface);
  // SystemVerilog form
  BusInterface bific1 ();
  mkM m1 (bific1);
  ...
endmodule: mkTop
```

```
module mkTop (TopInterface);
  // BSV shorthand
  BusInterface bific1 <-> mkM;
  ...
endmodule: mkTop
```
Registers in Verilog and SystemVerilog

- Verilog and SystemVerilog do not have hardware registers.

  - IEEE 1364 standard (Verilog LRM):
    - “A reg need not represent a hardware storage element ...”
    - “NOTE—In previous versions of the Verilog standard, the term register was used to encompass the reg, integer, time, real and realtime types, but that term is no longer used as a Verilog data type.”

  - IEEE P1800 standard (SystemVerilog LRM):
    - “The Verilog keyword reg doesn’t always accurately describe user intent.”
    - “Verilog has already deprecated the use of the term register in favor of variable.”

- Verilog and SystemVerilog are, first, simulation languages, with traditional software variables. With certain stylized usage, synthesis tools will interpret variables as hardware registers.

Registers are modules, in BSV

- In BSV, variables are never interpreted as hardware registers
- In BSV, all registers are modules and are instantiated explicitly
  - Naturally, they’re at the leaves of the module hierarchy
  - Like all modules, registers have interfaces
  - Register interfaces are parameterized types, and registers are strongly-typed

| Reg #(Int#(32)) | interface to a register that contains Int#(32) |
| Reg #(Bit#(16)) | interface to a register containing Bit#(16) values |
| Reg #(State)    | interface to a register containing a State value |
| Reg #(Request)  | interface to a register containing a Request value |
Instantiating registers follows standard module instantiation syntax

- The mkReg() module instantiates a register with a given reset value
  - The initial value must, of course, have the correct type for the type of the register (else type-checking error)
- The mkRegU module instantiates a register with an unspecified reset value

```verilog
Reg #(Int#(32)) r1 <- mkReg (0);  // Synchronously reset to 0
Reg #(Bit#(16)) r2 <- mkRegU;    // unspecified initial value
Reg #(Request) r4 <- mkReg (Request { op: Load; addr: 0 });
```

Writing and reading registers

- The Reg#() interface presents the methods to write and read from a register
  ```verilog
type Reg#(type t);
method t _read;
method Action _write(t a);
endinterface
```
  - Any module register declares the write method as "store the value" and the read method as "return the value"
- For ease of use:
  - a register on the left of an assignment is treated as a write operation
  - A register on the right side of an assignment is treated as a read operation

```verilog
Reg #(Int#(32)) r1 <- mkRegA(0);  // Asynchronously reset to 0

...  
r1 <= 23;  // is equivalent to r1._write(23);
...
let a = r1;  // is equivalent to let a = r1._read;
```
Registers are explicit modules

- “If registers must be specified explicitly, how can BSV be a high-level HDL?”
  - Microarchitecture is the creative (and fun) part of HW design; it distinguishes good designs from bad. The designer should remain involved in this.
    - Designers want precise control over microarchitecture
  - Complex concurrency and control is the hard and tedious part of HW design; it’s where most errors arise. BSV’s Rules dramatically simplify and automate this.

Example: Simple Binary Multiplication

- Consider the simple textbook binary multiplication algorithm illustrated below

```
  1001           // d = 4'd9
×  0101           // r = 4'd5
  1001           // d << 0 (since r[0] == 1)
  0000           // 0 << 1 (since r[1] == 0)
  1001           // d << 2 (since r[2] == 1)
  0000           // 0 << 3 (since r[3] == 0)
  0101101         // product (sum of above) = 45
```
A complete example: multiplier

```verilog
module mkTest (Empty);
    Reg#(int) state <- mkReg(0);
    Mult_ifc m <- mkMult1();
rule go (state == 0);
   m.start (9, 5);
   state <= 1;
endrule
rule finish (state == 1);
   3display("Product = %d",m.result());
   state <= 2;
endrule
endmodule: mkTest

(for now, just focus on interfaces, modules, instantiation and hierarchy; we'll discuss rules and methods later)

interface Mult_ifc;
    method Action start (int x, int y);
    method int result ();
endinterface: Mult_ifc

module mkMult1 (Mult_ifc);
    Reg#(int)  r             <- mkReg(0);
    method int result ();
    method Action start (int x, int y);
    method result () if (r == 0);
endmethod

rule cycle (r != 0);
   return product;
   d <= x; r <= y; product <= 0;
   r <= r >> 1;
   if (r[0] == 1) product <= product + d;
endrule

endmodule: mkMult1
```

Interface Types

- Interfaces are types, and hence can be used in the usual contexts where a type is expected
  - Typedefs
  - Function argument types, result types
  - ...

```verilog
typedef Reg#(Int#(16)) HalfwordReg;
typedef FIFO#(EtherPacket) InPackets;
function Get#(int) intFIFOtoGet (FIFO#(int));
```
Synthesis boundaries and inlining

- BSV encourages modularization: feel free to define modules wherever they make sense, no matter how small.

- The bsc synthesis tool (compiler) will generally inline modules and optimize across those boundaries, so that there is no performance penalty in using small modules.

- To prevent inlining, use a "(* synthesize *)" attribute just preceding the module header. Such modules remain modules in the generated Verilog.
  
  Caveat: see Reference Guide for some restrictions on module headers that can have this attribute.

```
(* synthesize *)
module mkM (BusInterface);
  ...
endmodule: mkM
```

Instance of `mkMult1.v`

- mkTest.v
- mkMult1.v

```getValue
(`* synthesize *)
module mkTest (Empty);
  ...
endmodule: mkTest

module mkMult1 (Mult_ifc);
  ...
endmodule: mkMult1
```
End of Lecture