Lecture 12: Conclusion

- Importing Verilog IP into BSV
- Embedding C code in BSV
- More on Overloading: Provisos
- Generating FSMs using the StmtFSM package
- Organization of the topics we have covered
- A listing of topics not covered in this basic training course
- A recap of the main contributions of BSV and how they improve your productivity
Importing a Verilog module into BSV

- Example motivations:
  - Adding a new primitive to the BSV repertoire
    - (BSV registers, fifos, ... are implemented like this)
  - Connecting to SRAM RTL produced by a 3rd party memory block generator tool
  - Connecting to an existing piece of IP written in Verilog
Importing a Verilog module into BSV

- Define the BSV interface that the Verilog module should provide
  - i.e., the “BSV view” of the Verilog module

- Use the “import BVI” mechanism to implement a BSV wrapper module that will:
  - Connect the wires of the BSV interface methods to the appropriate ports of the Verilog module
  - Specify the scheduling relationships between the methods of the interface
    - When you write a module in BSV, the bsc synthesis tool automatically figures out these scheduling relationships, so you never see them explicitly
    - When you import a Verilog module, you have to specify them

MAC Example

- A Verilog MAC (multiply-accumulate) unit:
  - When EN=1, out <= out+a*b
  - When clear = 1, out <= clear_value

```vhdl
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
```
MAC example

- Define the BSV interface that the imported module should have:

```hs
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
```

MAC example

- Define the BSV wrapper

```hs
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
```
These mechanisms work for VHDL, too

- Most well-known commercial simulators allow free intermixing of Verilog and VHDL
  - Verilog modules can use Verilog syntax to instantiate VHDL modules
  - VHDL modules can use VHDL syntax to instantiate Verilog modules

- Many of Bluespec’s customers utilized this
  - To import VHDL IP into Bluespec using ‘import BVI’
  - To plug Bluespec-generated Verilog into VHDL environments

- See also “attributes” section in Reference Guide about precise control over interface signal naming, which facilitates this kind of interoperability

Embedding C code in BSV
Embedding C code in BSV

- Using the "import BDPI" mechanism the user can specify that the implementation of a BSV function is provided as a C function.

- The function’s returned type can be value, Action or ActionValue.

- There is a direct relationship between the C argument types and the BSV types, for the details please check "$BLUESPECDIR/../doc/reference-guide.pdf"
More on Overloading: Provisos

[Review, from earlier lecture]
A first brush with Overloading

• Overloading: the ability to use a common function name or operator on some repertoire of types.
  Example:
  • “+” is meaningful on bits, integers, floating point, perhaps colors (add RGB values?), etc.
  • “<=” is meaningful on bits, numeric types, vectors, perhaps Ethernet packets (less priority?), ...

• In most languages, overloading is ad hoc
  • Usually, only on a fixed set of operators
  • Usually, not extensible by the designer/programmer
  • Minor exception: SystemVerilog allows extensibility over a fixed set of operators
  • Major exception: C++ has systematic extensibility
[Review, from earlier lecture]  
A first brush with Overloading

- BSV has a systematic way to extend overloading to any operator and function, and to any type
  - The terminology includes *typeclasses, typeclass instances, provisos, and deriving*

More on Overloading: Provisos

- Motivation: Suppose
  - we write a polymorphic module, or function
    - i.e., its type includes some *type variable “t”*
    - In general, this means that when you use such a module or function, you can use any type for t
  - However, inside the module or function, we wish to use some operator (such as + or ==) on entities of type t
  - Remember: not all types have + or == defined
  - Therefore, we wish to *constrain* the type t to just those types that have + or == defined
    - “t is any type, provided arithmetic/equality operators are defined on t”

- A “provisos” clause can be attached to module and function headers to express such constraints
More on Overloading: Provisos

- “provisos (Eq#(t))” is a constraint that says:
  - “t must belong to the typeclass Eq”, which means
    - == and != are defined for t

- “provisos (Arith#(t))” is a constraint that says:
  - “t must belong to the typeclass Arith”, which means
    - +, -, negate and * are defined for t

- “provisos (Bits#(t, n))” is a constraint that says:
  - “t and n must belong to the typeclass Bits”, which means
    - pack() and unpack() are defined for t, converting t ⇔ Bit#(n)

### Example 1, polymorphic function to add three values:

```haskell
function calc3(t val1, t val2, t val3) provisos(Arith#(t))
  return val1 + val2 + val3;
endfunction
```

### Example 2, polymorphic function to check identity of three values

```haskell
function eq3(s x, s y, s z) provisos(Eq#(s))
  return (x == y && y == z);
endfunction
```

Note!
- ‘calc3’ uses ‘+’
- ‘eq3’ uses ‘==’
More on Overloading: Provisos

- In the last two examples:
  // Calling calculate and equal functions with types Int#(8)
  Int#(8) result, num1 = 8'd2, num2 = 8'd3, num3 = 8'd4;
  Bool check;
  result = calc3 (num1, num2, num3); // ok, since '+' is defined on Int#(8)
  check = eq3 (num1, num2, num3); // ok, since '==' is defined on Int#(8)

- What if we try to apply our functions to complex numbers?
  // Complex number definition
  typedef struct {
    Int#(8) real_part;
    Int#(8) imaginary
  } Complex;
  Complex c1, c2, c3, result; // Four Complex numbers
  // Calling calculate and equal functions with types Complex
  result = calc3 (c1, c2, c3);
  check = eq3 (c1, c2, c3);

Does this work?

More on Overloading: Provisos

- Review: one way to define ‘==’ and ‘!=’ on a type is to attach a ‘deriving (Eq)’ clause to the type definition

  // Complex number definition with deriving
  typedef struct {
    Int#(8) real_part;
    Int#(8) imaginary
  } Complex deriving(Bits, Eq);

Will this work?

A: Solves ‘eq3’ problem but not ‘calc3’
More on Overloading: Instance

- The “instance” construct is the general mechanism for giving new definitions of overloaded operators and functions.

```haskell
instance typeclass #( type, type, ...);
    function/operator definition ...
    function/operator definition ...
    ... for all the functions/operators required by the typeclass ...
endinstance
```

- Note: the “deriving” construct is just a shorthand for a few important special cases.

```haskell
instance Arith#(Complex);

function Complex \( + \) (Complex a, Complex b);
    Complex sum;
    sum.real_part = a.real_part + b.real_part; // Calculate real part
    sum.imaginary = b.imaginary + b.imaginary; // Calculate imaginary part
    return sum; // return the result
endfunction

... and similarly for \(-\), \(*\) and negate, the remaining operators/functions
... required for the Arith typeclass
endinstance
```

Solves ‘calc3’ problem
Generating FSMs using the StmtFSM package

State Machine Generation

- BSV has powerful notation for construction of state machines
  - Useful for orchestration of complex events
  - Useful in designs and in testbenches

- Integrated cleanly into the language (Rule semantics)
  - Clean, well-defined semantics for resources shared between multiple state machines (e.g., shared counters, queues)

- Fully synthesizable

- Automatic generation of necessary state variables and registers
State Machine Concepts

- A state machine is an orchestration of basic actions
  - “Orchestration” => sequencing, looping, conditionals, jumps, parallel execution, ...

- In BSV, the basic action is exactly the same as a rule body, i.e., an entity of type Action, which can of course be a composite of primitive sub-actions (such as a register assignment)

- FSMs can be composed—FSM pieces can be combined into larger FSMs

- The compositional principle is:
  - Every FSM has a start state and a done state
  - When constructing a larger FSM from FSM pieces, the start and done state of the larger FSM is derived systematically from the pieces

State Machine Composition

- Sequencing
  - Syntax: “seq a1; a2; ...; aN endseq”
  - Where each aJ is either an Action, or itself a sub-FSM
State Machine Composition

- **Conditionals**
  - Syntax: “if (c) fsm$_1$”
  - Syntax: “if (c) fsm$_1$ else fsm$_2$”

- **Iteration**
  - Syntax: “for (... loop control ...) fsm$_1$”
  - Syntax: “while (e) fsm$_1$”
  - Syntax: “repeat (n) fsm$_1$”
  - Loop bodies can contain “break” and “continue”
State Machine Composition

- Parallel composition
  - Syntax: “par fsm_1; fsm_2; ...; fsm_N endpar”
  - The sub-fsm start at the same time, and proceed independently; the whole fsm is done when all of them are done

```
interface FSM;
  method Action start ();
  method Bool done ();
  method Action waitTillDone ();
endinterface: FSM

module mkFSM #( Stmt s ) ( FSM );
```

StmtFSM

- The StmtFSM package (in the library) provides a way of defining FSMs using traditional programming notation

```
implementation FSM;
```

- The mkFSM module takes a parameter of type Stmt. It is a “spec” of a state machine. The module constructs/generates the FSM.
StmtFSM - example

Stmt specfsm =
  seq
    write( 15, 51 );
    read( 15 );
    ack ;
    ack ;
    write( 16, 61 );
    write( 17, 71 );
  // a memory operation and an
  // acknowledge can occur at
  // simultaneously
  action
    read( 16 );
    ack ;
  endaction
  action
    read( 17 );
    ack ;
  endaction
  action
    endseq;

FSM testfsm <- mkFSM (specfsm);

• seq / endseq define a
  sequence (multi-cycle)

• action / endaction define
  a simultaneous block (in
  one cycle)

• par / endpar defines a
  parallel block (multi-cycle)

// Specify an FSM generating a test sequence
Stmt test_seq =
  seq
    for (i <= 0; i < NI; i <= i + 1)                      // each source
      for (j <= 0; j < NJ; j <= j + 1) begin            // each destination
        let pkt <- gen_packet ();
        send_packet (i, j, pkt);                        // test i-j path in isolation
      end
    // then, test arbitration by sending packets simultaneously to same dest
    action
      send_packet (0, 1, pkt0);                        // to dest 1
      send_packet (1, 1, pkt1);                        // to dest 1 (so, collision)
    endaction
  endseq

mkAutoFSM (fsm); // Generate the FSM and code to run it automatically
State Machine Generation

- Easy to specify precise orchestration of stimulus
  - sequencing, parallel, conditional, iteration
- Same Rule semantics
  - Automatically flow-controlled
    - E.g., a par may try to send packets into two ports simultaneously, to verify arbitration; what if one port is temporarily blocked?
    - The par will automatically block
    - Packets can still be sent into the other port
  - Robust to latency variations
    - E.g., two FSMs sharing a resource (e.g., a counter, a port) will automatically be arbitrated

The topics we have covered can be organized into categories:

<table>
<thead>
<tr>
<th>HW Structure:</th>
<th>Module Hierarchy, providing and using interfaces</th>
<th>Combinational Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior:</td>
<td>Rules, Interface methods and abstraction</td>
<td>Rule Scheduling, FSMs</td>
</tr>
<tr>
<td>Types:</td>
<td>Simple Types, Enums, Structs, Parameterized Types, Maybe#(t), Overloading/deriving</td>
<td></td>
</tr>
<tr>
<td>Clocks:</td>
<td>Multiple Clock Domains and Gated Clocks</td>
<td></td>
</tr>
<tr>
<td>Pragmatics:</td>
<td>Performance Tuning, Matching BSV interfaces to external and internal Verilog, Using the tools, debugging</td>
<td></td>
</tr>
</tbody>
</table>
Elevating Design above RTL

Bluespec SystemVerilog

- Behavioral
- Fully synthesizable
- Structural

Correct: Concurrency and Communications
Atomic Transactions using Rules and interface methods for complex control and concurrency:
- Across multiple shared resources
- Across module boundaries

Correct: Construction and Configurability
- High-level types closer to spec
- Much more powerful “generate”
- Powerful parameterization
- Powerful static checking & formal semantics
- Advanced clock management

VHDL/Verilog/SystemVerilog/SystemC

Summary

- Bluespec SystemVerilog can have a revolutionary improvement in your productivity
  - 2x even with this basic training, more as you master the techniques
- BSV is aimed at design, but it improves the entire flow from spec to verified netlist
  - And, change and evolution of designs
- The fundamental ideas are:
  - Rules and Rule-based Interface Methods are a breakthrough in describing HW behavior
  - Aggressive use of types and abstraction mechanisms from modern programming languages
End of Lecture