Lecture 09: More on Module Interfaces

- Packages and Libraries

- Interfaces: raising the level of abstraction
  - The Get and Put interfaces
  - Nested interfaces. E.g., Client/Server
  - The Connectable typeclass

- Interfaces: lowering the level of abstraction
  - Motivation: matching an external port list/protocol spec
  - Making implicit conditions explicit
  - Removing READY and ENABLE signals
  - “Unguarded” methods
  - Driving input and output buses to/from a module’s rules with zero latency
Packages

- SystemVerilog introduces the concept of a “package” to organize related definitions into a name space
  - BSV libraries are organized in packages
- Need to import library packages explicitly

```verilog
package Foo;
typedef ... T1;
typedef ... T2;
T1 x = ...;
interface I1 ...
interface I2 ...
module mkM1 ...
module mkM2 ...
endpackage: Foo

package Baz;
  import Foo :: *;
  import ...
  ...
  can use T1, T2, x, I1, I2, mkM1, mkM2 ...
  ...
endpackage: Baz
```

BSV general library conventions

- Module names typically begin with “mk”, pronounced “make” e.g.,
  - mkReg, mkFIFO, mkRegFile
- Types (and Interfaces, because they are types) begin with an uppercase letter- e.g.,
  - Action, ActionValue#(t), Vector#(Bool), Put#(Tin)
- Functions and other identifiers begin with a lowercase letter, e.g.,
  - min, max, map, foldl, fifoToPut
The Standard Prelude Package

- The only package that is automatically imported
  - (no need for explicit "import package :: *")

- A “built in” library of primitive instances, types, and typeclasses
  - Typeclasses: Bits, Eq, Ord, Bitwise, Arith, Bounded, Literal
    - Defining all the built-in operators (==, +, &&, |, >, ...)
  - Types: Action, Bit, Bool, int, Maybe, Empty, ...
  - Registers (initialized, uninitialized, asynch reset, ...)

- Documented in the Reference Guide

Bluespec Libraries

- Bluespec provides an extensive library with parameterized modules, interfaces, functions, types, etc.
  - Register Files, FIFOs and variations, RWire and variations, RAM models
  - Complex, fixedpoint, one-hot, Wallace adder, population count, random number generator
  - FSM generators, rigid pipelines
  - Get/Put and variations, Client/Server, Connectables
  - Localbus (Control/Status regs)
  - Clock synchronizers, Tri-state models

- The Reference Guide (in the SW release) contains documentation

- Most of the libs are written in BSV itself; very few are “built-in”
  - Users can create their own libraries
Interfaces: raising the level of abstraction

- Recall: Interfaces contain methods

- Interfaces can also contain other interfaces
  - We use this to build a hierarchy of interfaces
  - Get/Put ➔ Client/Server ➔ ...
  - These capture common interface design patterns
  - There is no HW overhead to such abstraction

- Connections between standard interfaces can be packaged (and used, and reused)
  - The Connectable typeclass

Get and Put Interfaces

- Provides simple handshaking mechanism for getting data from a module or putting data into it
- Easy to connect together

```plaintext
interface Get#(type t);   // polymorphic
  method ActionValue#(t) get();
endinterface: Get

interface Put#(type t);
  method Action put(t x);
endinterface: Put
```

Diagram showing the get and put interfaces with signals for data, ready, and enable.
Get and Put Interfaces

- Example: interface provided by a cache (c) to a processor (p)

```
interface CacheFc:
  interface Put#(Req_t) p2c_request;
  interface Get#(Resp_t) c2p_response;
  endinterface

module mkCache (CacheFc);
  FIFO#(Req_t) p2c <- mkFIFO;
  FIFO#(Resp_t) c2p <- mkFIFO;
  ... rules expressing cache logic ...
  interface p2c_request;
  method Action put (Req_t req);
    p2c.enq (req);
    endmethod
  endinterface
  interface c2p_response;
  method ActionValue#(Resp_t) get ();
    let resp = c2p.first; c2p.deq;
    return resp;
    endmethod
  endinterface
endmodule
```

Get and Put Interfaces

```
function Put#(Req_t) toPut(FIFO#(Req_t) fifo);
  return (interface Put;
    method Action put (a);
      fifo.enq (a);
    endmethod
  endinterface);
endfunction
```

Example: interface provided by a cache (c) to a processor (p)
Get and Put Interfaces

- Example: interface provided by a cache (c) to a processor (p)

```sv
function Get#(Resp_t) toGet(FIFO#(Resp_t) fifo);
return (interface Get;
    method ActionValue#(Resp_t) get ();
    let a = fifo.first;
    fifo.deq;
    return a;
endmethod;
endinterface);
endfunction
```

```sv
module mkCache (CacheIfc);
    interface CacheIfc;
        // interface provided by a cache (c) to a processor (p)
        fifo.deq;
        let a = fifo.first;
        get put
    endinterface
endmodule
```

```sv
module mkCache (CacheIfc);
    interface c2p_response;
        rules expressing cache logic …
    endinterface
    FIFO#(Req_t) p2c_request;
    method Action put (Req_t req);
        p2c.enq (req);
        enable
        ready
        data
    endmethod;
endinterface
method Action put (Req_t req);
    p2c.enq (req);
    enable
    ready
    data
endmethod;
```

Converting FIFOs to Get/Put

- The BSV library provides functions to convert FIFO interfaces to Get/Put.

```sv
module mkCache (CacheIfc);
    interface CacheIfc;
        interface Put#(Req_t) p2c_request;
        interface Get#(Resp_t) c2p_response;
        …
    endinterface
endmodule
```

```sv
module mkCache (CacheIfc);
    interface p2c_request =
        toPut (p2c);
    interface c2p_response =
        toGet (c2p);
endmodule
```

Absolutely no difference in the HW!

Note: toGet and toPut are overloaded functions from ToPut and ToGet typeclasses. Instances are supplied for many FIFO-like interfaces.
Client/Server interfaces

- Get/Put pairs are very common, and duals of each other, so the library defines Client/Server interface types for this purpose.

```
interface Client #(req_t, resp_t);
  interface Get#(req_t) request;
  interface Put#(resp_t) response;
endinterface

interface Server #(req_t, resp_t);
  interface Put#(req_t) request;
  interface Get#(resp_t) response;
endinterface
```

```text
client

get

put

server
```

```
module mkCache (CacheIfc);
  FIFO#(Req_t) p2c <- mkFIFO;
  FIFO#(Resp_t) c2p <- mkFIFO;
  FIFO#(Req_t) c2m <- mkFIFO;
  FIFO#(Resp_t) m2c <- mkFIFO;
endmodule

rules expressing cache logic ...
```

```
module mkCache (CacheIfc);
  FIFO#(Req_t) p2c <- mkFIFO;
  FIFO#(Resp_t) c2p <- mkFIFO;
  FIFO#(Req_t) c2m <- mkFIFO;
  FIFO#(Resp_t) m2c <- mkFIFO;
endmodule

... rules expressing cache logic ...

interface Server ipc
  interface Put put(a);
    method Action put(a);
    p2c.enq(a);
  endmethod
endinterface

interface Get get;
  method ActionValue#(Resp_t) get;
  c2p.deq;
  return c2p.first;
endmethod
endinterface
```

```
// Server
```
interface CacheIfc;
  interface Server#(Req_t, Resp_t)  ipc;
  interface Client#(Req_t, Resp_t) icm;
endinterface

module mkCache (CacheIfc);
  // from / to processor
  FIFO#(Req_t)   p2c <- mkFIFO; FIFO#(Resp_t) c2p <- mkFIFO;
  // to / from memory
  FIFO#(Req_t)   c2m <- mkFIFO; FIFO#(Resp_t) m2c <- mkFIFO;
  ... rules expressing cache logic ...
  interface ipc = fifosToServer (p2c, c2p);
  interface icm = fifosToClient (c2m, m2c);
endmodule

...
Connecting Get and Put

- A module m1 providing a Get interface can be connected to a module m2 providing a Put interface with a simple rule

```verbatim
module mkTop (...)
    Get#(int) m1 <- mkM1;
    Put#(int) m2 <- mkM2;
rule connect;
    let x <- m1.get();  m2.put (x);  // note implicit conditions
endrule
endmodule
```

Connectables

- There are many pairs of types that are duals of each other
  - Get/Put, Client/Server, YourTypeT1/YourTypeT2, ...

- The BSV library defines an overloaded module `mkConnection` which encapsulates the connections between such duals
  - The BSV library predefines the implementation of `mkConnection` for Get/Put, Client/Server, etc.

- Because overloading in BSV is extensible, you can overload `mkConnection` to work on your own interface types T1 and T2

- The BSV library also recursively defines `mkConnection` for `tuples` and `vectors` of interfaces that are already individually connectable (including your own interface types)
mkConnection

- Using these interface facilities, assembling systems becomes very easy

```systemverilog
interface CacheIfc;
  interface Server#(Req_t, Resp_t)  ipc;
  interface Client#(Req_t, Resp_t)  icm;
endinterface

module mkTopLevel (...)
  // instantiate subsystems
  Client #(Req_t, Resp_t)         p  <- mkProcessor;
  Cache_Ifc #(Req_t, Resp_t)  c  <- mkCache;
  Server #(Req_t, Resp_t)         m <- mkMem;
  // instantiate connects
  mkConnection (p, c.ipc);
  mkConnection (c.icm, m);
endmodule
```

Interfaces: lowering the level of abstraction

- Why?
  - When a BSV module \( m \) must connect to a non-BSV environment with a pre-specified port list and protocol, \( m \)'s interface needs to be *matched* exactly to the environment
  - Example: connecting to AMBA AHB
  - Example: in an existing IP subsystem, replacing a block with a new one written in BSV

- This usually means that we can’t use BSV’s Action, ActionValue and value method protocols and signaling
Interfaces: exposing method conditions

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

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

Implementing exposed conditions is trivial: just replicate the implicit condition

- E.g., below, enq()’s implicit condition canEnqueue 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 explicitly by writing let x = canEnqueue and using x twice

```plaintext
module mkFIFO (FIFO#(type t));
  ...
  method Action enq(t x) if (canEnqueue); ...
endmethod
  method t first if (canDequeue); ...
endmethod
  method Action deq if (canDequeue); ...
endmethod
  method Bool notFull; return canEnqueue; endmethod
  method Bool notEmpty; return canDequeue; endmethod
endmodule: mkFIFO
```
Interfaces: removing READY signals

- The "always_ready" attribute can be applied to a method to indicate
  - that it is always ready (the compiler will flag an error if it is not true)
  - that the READY signal must be removed from the generated Verilog

```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
```

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 READY signal for `add` and `result`.

Interfaces: removing ENABLE signals

- The "always_enabled" attribute can be applied to a method (Action, ActionValue) to indicate
  - that it is assumed True, i.e., the data input signals (args) are driven by the environment on every cycle
    This implies the method’s implicit condition must be always True
  - that the ENABLE signal must be removed from the generated Verilog

```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
```

The compiler will check that `add` can be always invoked, i.e. its implicit and explicit conditions are always True.

The compiler will not generate any ENABLE signal for `add`.

---

Bluespec SystemVerilog™ Training 12  Lecture 09: Interfaces II
**Interfaces: “unguarded” methods**

- Suppose a module must read a 34-bit decoded instruction bus on EVERY CYCLE, of which the lower 3 bits are an opcode, with opcode == 0 indicating IDLE, and we want to capture the (non-IDLE) instructions in an input FIFO

```plaintext
typedef enum {Idle, Bz, Add, Sub} OpCode deriving(Bits, Eq);

typedef struct { Bit#(32) operands; OpCode opCode; } Cmd deriving(Bits, Eq);
```

```plaintext
interface IfcFoo#(type t);
  method Action instr(t bus);
endinterface
```

(* always_enabled = “instr” *)

```plaintext
module mkFoo (IfcFoo#(Cmd));

(* always_enabled = "instr" *)
module mkFoo (IfcFoo#(Cmd));
  FIFO#(Cmd) cmds <- mkFIFO;
  ...
  method Action instr(bus);
  if (bus.opCode != Idle)
    cmds.enq(bus);
  endmethod
endmodule
```

The compiler will signal an error, since the enq() method may not be ready (the fifo may be full).
It cannot verify that instr is always ready, so it cannot allow the always_enabled spec.

**Interfaces: “unguarded” methods**

- What if we use FIFOF and test for the fifo being notFull?

```plaintext
typedef enum {Idle, Bz, Add, Sub} OpCode deriving(Bits, Eq);

typedef struct { Bit#(32) operands; OpCode opCode; } Cmd deriving(Bits, Eq);
```

```plaintext
interface IfcFoo#(type t);
  method Action instr(t bus);
endinterface
```

(* always_enabled = “instr” *)

```plaintext
module mkFoo (IfcFoo#(Cmd));

(* always_enabled = "instr" *)
module mkFoo (IfcFoo#(Cmd));
  FIFO#(Cmd) cmds <- mkFIFO;
  ...
  method Action instr(bus);
  if (bus.opCode != Idle && cmds.notFull)
    cmds.enq(bus);
  endmethod
endmodule
```

This will still not work, because The compiler doesn’t know that (notFull == True) implies enq is Ready.
Interfaces: “unguarded” methods

- Answer: use “unguarded” methods, i.e., methods with no implicit 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

```vhdl
typedef enum {Idle, Bz, Add, Sub} OpCode deriving(Bits, Eq);
typedef struct { Bit#(32) operands; OpCode opCode; } Cmd deriving(Bits, Eq);

(* always_enabled = "instr" *)
module mkFoo (IfcFoo#(Cmd));
  FIFO#(Cmd) cmds <- mkUGFIFO;
  method Action instr(bus);
    if (bus.opCode != Idle && cmds.notFull)
      cmds.enq(bus);
  endmethod
endmodule
```

This will work. It is possible always to enq in a mkUGFIFO

Warning: effect of using enq without checking notFull is unspecified!

Interfaces: summary on matching an external port spec

- Any external port spec can be matched exactly, using one or more of the following techniques

- Expose implicit conditions into explicit Bool methods

- Use “always_ready” attributes to remove output READY signals from any value, Action or ActionValue method
  - Compiler will check that this is legal

- Use “always_enabled” attributes to remove input ENABLE signals from any Action of ActionValue method
  - Implies “always_ready”, and the compiler will check this, i.e., method bodies cannot use other methods that have implicit conditions
  - Use components with unguarded methods, if necessary
    - Tip: when defining a component with unguarded methods, remove guards only where necessary. E.g., a fifo with unguarded enq but guarded first/deq

- What remains are pure input and output buses (Verilog ports)
Driving input buses into rules with zero latency

- If an input bus is registered, the method body just assigns the value to a register or fifo. 1 cycle later, it’s available to all rules in the module.

```verilog
method Action m1 (int bus1, int bus2);
    reg_a <= bus1;
    fifo_b.enq (bus2);
endmethod
```

- Sometimes, one or more input buses must be made available directly (zero latency) to a rule for combining/ arbitration/... In this case, use an RWire
  - The Prelude also contains a variation on RWire called Wire
  - Instead of the wset() and wget() RWire methods, respectively, it has
    - a _write() method (same as a Reg, therefore can use “<=" assignment)
    - a _read() method (same as a Reg, therefore can read it implicitly)
  - Instead of a separate Valid bit, it’s encoded in the implicit cond of _read
Driving output buses from rules with zero latency

- If an output bus is registered, the method body just reads the value from a register or fifo. 1 cycle earlier, it was put there by some rule or method.

  ```verilog
  rule r1 (condA);
  reg_a <= 3;
  endrule

  rule r2 (condB)
  fifo_b.enq(9);
  endrule

  method int m1;
  return reg_a;
  endmethod

  method int m2;
  return fifo_b.first;
  endmethod
  ```

- Sometimes, an output bus must be driven directly (zero latency) from a rule. In this case, use an RWire or a Wire.
Summary

- BSV programs are organized into packages.
  - The standard BSV libraries provide many useful packages (documented in the Reference Guide)

- Interfaces: raising the level of abstraction
  - We can very quickly, succinctly and correctly define very complex interfaces, and connect them, using interface abstraction, and BSV library elements Get/Put, Client/Server, Connectables, etc.

- Interfaces: lowering the level of abstraction
  - We can precisely interface to any external port list/protocol spec, by exposing implicit conditions, using always_ready and always_enabled attributes, using “Unguarded” methods, and using RWires/Wires for zero-latency (unregistered) communication

End of Lecture