BSV Training

Lec_Interfaces_TLM

Transaction-Level Modeling (TLM)
Get/Put, Client/Server, interface transformers, mkConnection, Connectable typeclass

www.bluespec.com

Introduction

- BSV's 'method' construct in interfaces allows you to define arbitrary methods for a module, specifying arguments and their types, and results and their types.

- However, as a matter of style, readability, documentation, succinctness and maintainability, we often reuse certain standard interfaces from the BSV library, such as Get/Put and Client/Server, along with standard definitions from the BSV library for connecting modules with such interfaces

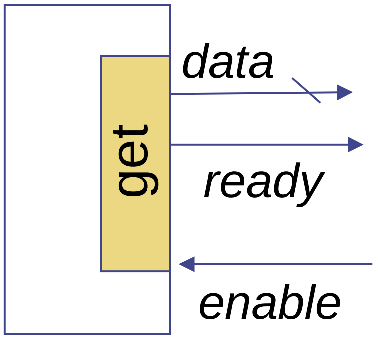
- In the SystemC world, use of such stylized interfaces is often called TLM (for "Transaction Level Modeling"). It's the same idea here (BSV has had this capability since 2000!), and it's more than just for modeling—we use them routinely in production synthesized code.

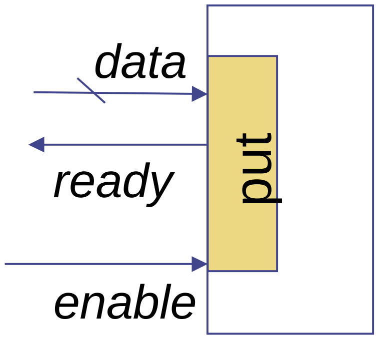
Get/Put interfaces

Instead of defining *ad hoc* methods in interfaces, one often uses interfaces already defined in the BSV library that capture certain common design patterns. Example:

```
// For getting a value out of a module
interface Get#(type t);
  method ActionValue#(t) get();
endinterface: Get

// For putting a value into a module
interface Put#(type t);
  method Action put(t x);
endinterface: Put
```

By themselves, these do not imply anything more than the interface wires and protocol; what's in the module depends on how you to define these methods.

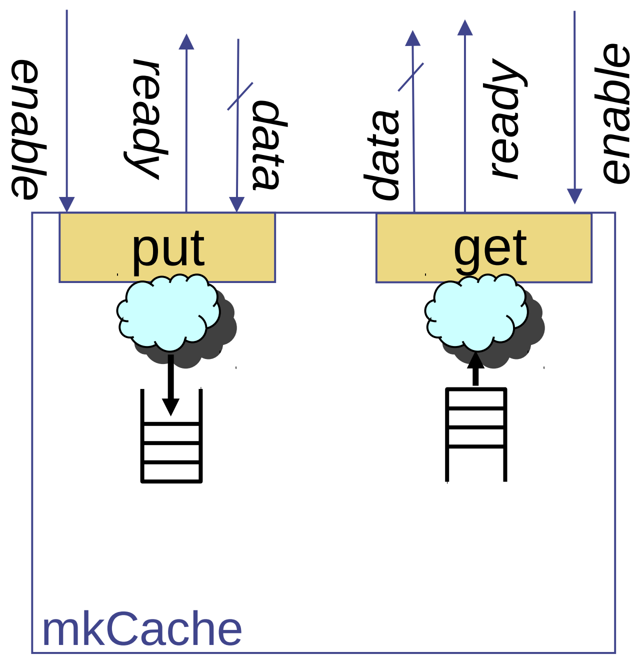




Get/Put example

An interface provided by a cache towards a processor

```
interface CacheIfc;
  interface Put#(Req_t)   p2c_request;
  interface Get#(Resp_t) c2p_ response;
   …
endinterface
```




```
module mkCache (CacheIfc);
   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
```

Standard interface transformers

A FIFO 'enq' operation can be seen as a 'put' operation.
FIFO 'first' and 'deq' operations can be seen as a 'get' operation.
These ideas can be captured as functions that transform interfaces.
(These two examples already exist in the BSV library)

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

```
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
```

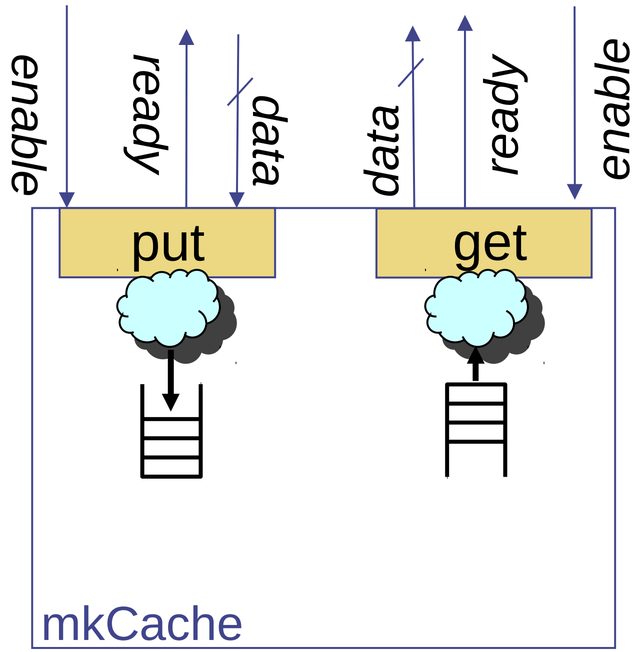
```
module mkCache (CacheIfc);
   FIFO#(Req_t)  p2c <- mkFIFO;
   FIFO#(Resp_t) c2p <- mkFIFO;

   … 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
```

Using standard interface transfomers

This simplifies the interface definition in the module.
There is no HW cost to this (it statically elaborates to the same HW).

```
interface CacheIfc;
  interface Put#(Req_t)    p2c_request;
  interface Get#(Resp_t) c2p_ response;
   …
endinterface
```

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

   … rules expressing cache logic …

  interface p2c_request = toPut (p2c);

  interface c2p_response = toGet (c2p);

endmodule
```

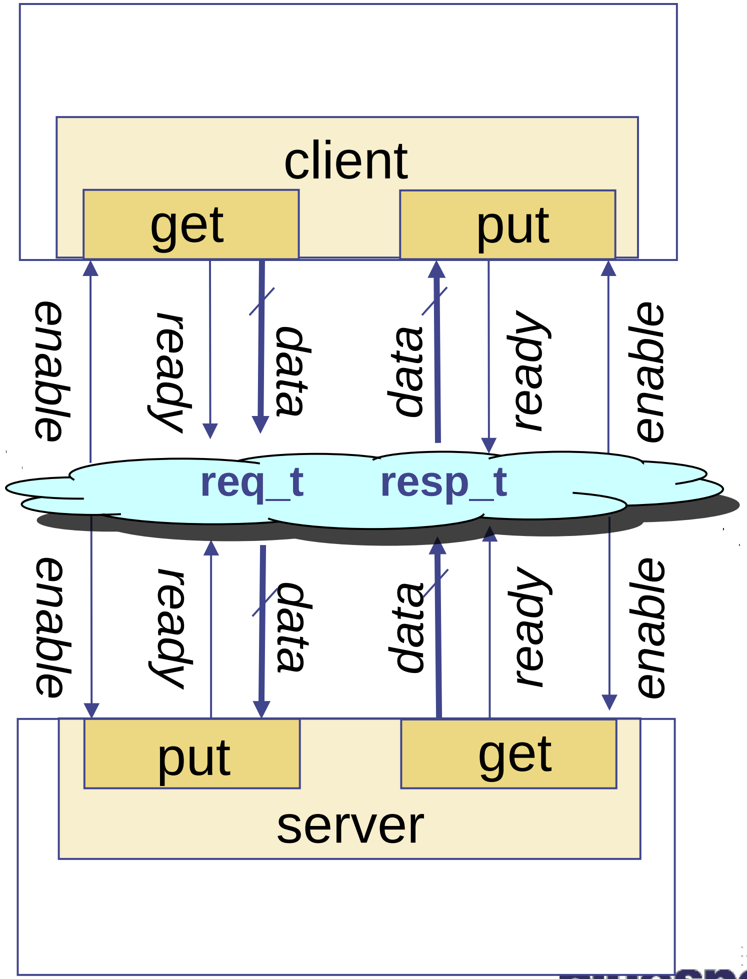


Note: toGet and toPut are actually overloaded functions from ToPut and ToGet typeclasses. The BSV library supplies instances for many interfaces, including FIFOs.

Client/Server interfaces

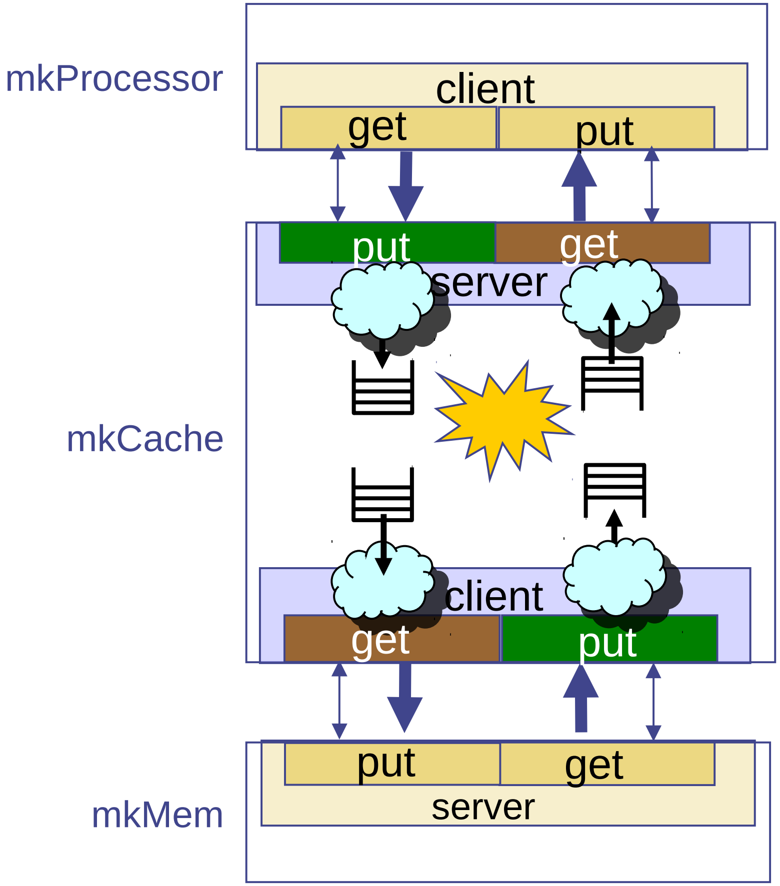
Interfaces can be nested.
I.e., inside an interface declaration, instead of declaring methods, you can use an already-defined interface. Here is another example from the BSV library.

```
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
```



Example: using Client/Server for the cache

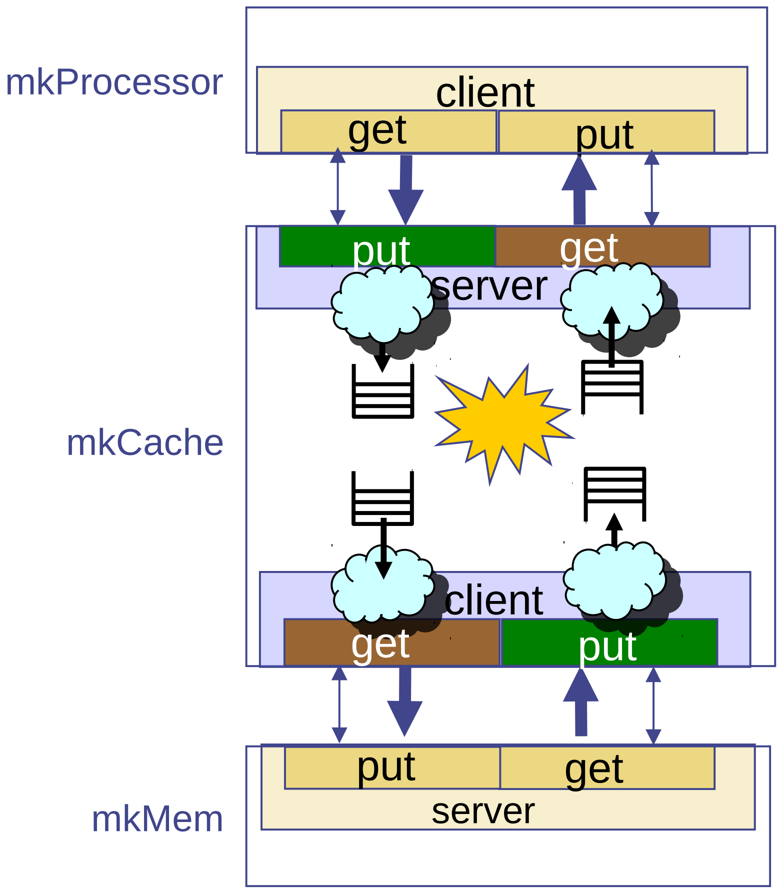
```
interface CacheIfc;
   interface Server#(Req_t, Resp_t)  ipc;
   interface Client#(Req_t, Resp_t)
icm;
endinterface
```




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

   FIFO#(Req_t)   c2m <- mkFIFO;
   FIFO#(Resp_t)  m2c <- mkFIFO;

   … rules expressing cache logic …

   interface Server ipc;
      interface Put request = toPut (p2c);
      interface Get response = toGet (c2p);
   endinterface

   interface Client icm;
      interface Get request = toGet (c2m);
      interface Put response = toPut (m2c);
   endinterface
endmodule
```

Example: using interface transformers

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



This can be further simplified with another common interface transformer

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

  FIFO#(Req_t)   c2m <- mkFIFO;
  FIFO#(Resp_t)  m2c <- mkFIFO;

  … rules expressing cache logic …

  interface Server ipc =
    toGPServer (p2c, c2p);

  interface Client icm =
    toGPClient (c2m, m2c);
endmodule
```


bluespec

In general, interface transformers are modules

In the examples so far, toGet, toPut, toGPClient and toGPServer were simple functions.

Functions in BSV are "pure"—they cannot contain any internal state, and therefore can represent only "instantaneous" (combinational) computation.

In general, an interface transformer may need state and temporal computation

- E.g., a transformer that "serializes" from wide data to narrow data

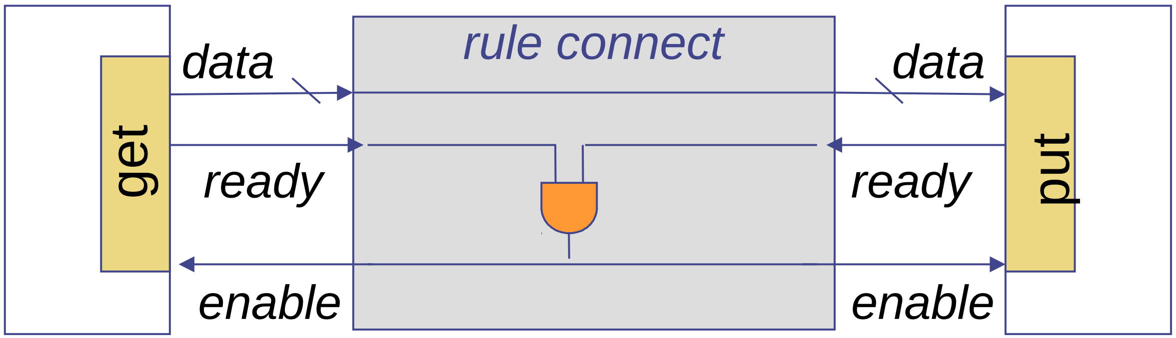
Such transformers will have to be modules, not just functions.

They're often written using the "connections" methodology, discussed next.

Connecting Get and Put

A Get and a Put interface (carrying the same type of data) can be connected with an explicit rule.

```
module mkTop (…)
   Get#(int) m1 <- mkM1;
   Put#(int) m2 <- mkM2;

   rule connect;
      let x <- m1.get();  m2.put (x);          // note implicit conditions
   endrule
endmodule
```



But, as we will see in the next few slides, even this design pattern can be captured with an abstraction.

Capturing the design pattern

We can define a parameterized module that captures the design pattern.

```
module mkConnectionGetPut #(Get#(t) g, Put#(t) p) (Empty);
   rule connect;
      let x <- g.get();  p.put (x);
   endrule
endmodule
```

```
module mkTop (…)
   Get#(int) m1 <- mkM1;
   Put#(int) m2 <- mkM2;

   mkConnectionGetPut (m1, m2);
endmodule
```

```
// Technically:   Empty e <- mkConnection (m1, m2);
//  Replaces:
//     rule connect;
//        let x <- m1.get();  m2.put (x);
//     endrule
```


bluespec

Further generalization of the connection pattern

Similarly, we could create abstractions for other types of connections:

mkConnectionPutGet(p,g)
mkConnectionClientServer (c,s), mkConnectionServerClient (s,c)
mkConnectionAXIMasterAXISlave (am, as)
mkConnectionTLMMasterTLMSlave (tm,ts)
....

Instead of inventing new names for each such connection between pairs of related interface types, we can use BSV's "*overloading*" mechanism to use a common name, "mkConnection", for all of them.

Using *overloading resolution*, the compiler will figure out the correct module to be used for the connection, based on the interface argument types.

The concepts related to overloading in BSV are:

- "typeclass"
- "instance"
- "deriving" (automatic creation of certain instances)

(Typeclasses and overloading are discussed in more detail in another lecture)

The “Connectable” typeclass

```
typeclass Connectable #(type t1, type t2);
   module mkConnection #(t1 m1, t2 m2) (Empty);
endtypeclass
```

This declares a “type class”, which is a set of types on which certain “overloaded” identifiers can be declared. (This declaration is already in the BSV library.)

This can be read as: “two types t1 and t2 are in the Connectable typeclass when an overloaded identifier mkConnection has been defined for them, with the module type shown”.

We populate a typeclass explicitly using “instance” declarations:

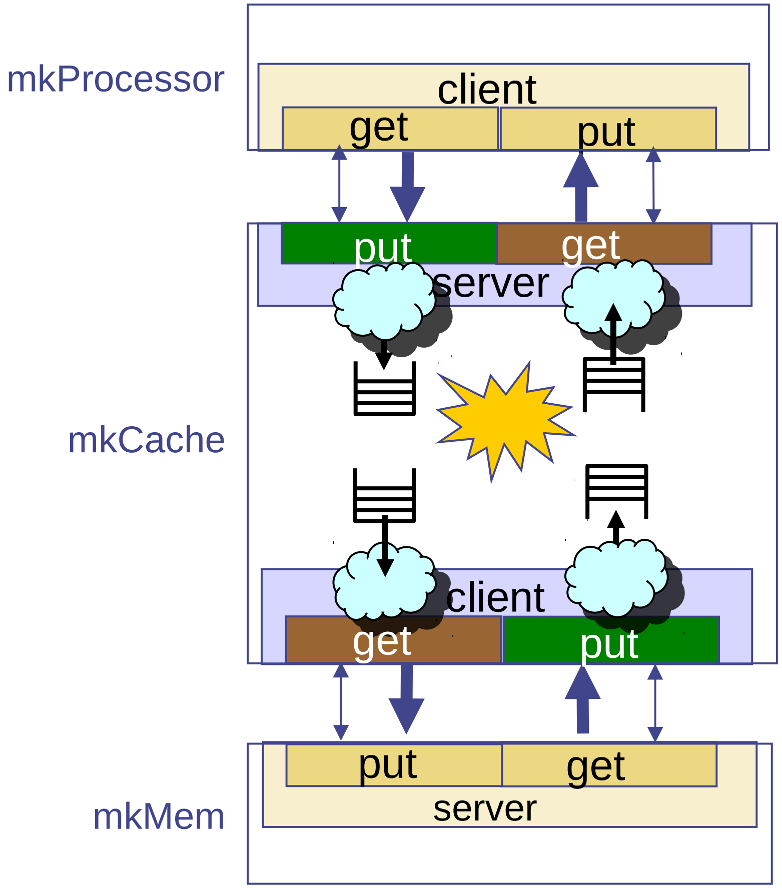
```
instance Connectable #(Get #(t), Put#(t));
   module mkConnection #(Get#(t) m1, Put#(t) m2) (Empty);
      rule r;
         let x <- m1.get; m2.put (x);
      endrule
   endmodule
endinstance
```

The BSV library provides instances for Get/Put, Client/Server, and many other types

[C++ gurus: Connectable is like a “virtual class”, with “virtual member” mkConnection.
The Get/Put pair of types “inherits” from this virtual class by providing a definition for mkConnection.]

Example: using mkConnection

The top-level of our processor-cache-memory system (mkTopLevel) reduces to 5 lines of code:




```
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
```

End

Questions?

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