

Bluespec-3: Modules & Interfaces

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

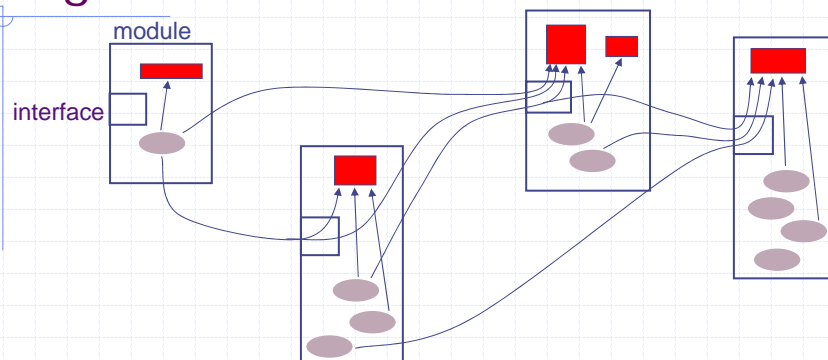
Based on material prepared by Bluespec Inc,
January 2005

February 28, 2005

<http://csg.csail.mit.edu/6.884/>

L09-1

Bluespec: State and Rules organized into *modules*



All *state* (e.g., Registers, FIFOs, RAMs, ...) is explicit.
Behavior is expressed in terms of atomic actions on the state:

Rule: condition \rightarrow action

Rules can manipulate state in other modules only *via* their interfaces.

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-2

Example 1: simple binary multiplication

```

typedef bit[15:0] Tin;
typedef bit[31:0] Tout;

module mkMult0 (Empty);
    Tin d_init = 9, r_init = 5;    // compile-time constants

    Reg#(Tout) product <- mkReg (0);
    Reg#(Tout) d        <- mkReg ({16'h0000, d_init});
    Reg#(Tin)  r         <- mkReg (r_init);

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

    rule done (r == 0);
        $display ("Product = %d", product);
    endrule: done

endmodule: mkMult0

```

Replace it by a "start" method

State

Behavior

Replace it by a "result" method

This module has no interface methods; it only multiplies 9 by 5!

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-3

Example 1: Modularized

```

interface Mult_ifc;
    method Action start (Tin, x, Tin y);
    method Tout  result ();
endinterface

module mkMult1 (Mult_ifc);
    Reg#(Tout) product <- mkReg (0);
    Reg#(Tout) d        <- mkReg (0);
    Reg#(Tin)  r         <- mkReg (0);

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

    method Action start (d_init, r_init) if (r == 0);
        d <= d_init; r <= r_init;
    endmethod

    method result () if (r == 0);
        return product;
    endmethod

endmodule: mkMult1

```

State

Behavior

Interface

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-4

Interfaces

```
interface Mult_ifc;
  method Action start (Tin x, Tin y);
  method Tout    result();
endinterface

module mkMult1 (Mult_ifc);
...
endmodule
```

- An interface *declaration* defines an interface *type*
 - Corresponds, roughly, to the port list of an RTL module
 - Contains prototypes of *methods*, which are “transactions” that can be invoked on the module
- A module declaration specifies the interface that it *implements* (a.k.a. *provides*)

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-5

A Test bench for Example 1

```
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);
    $display ("Product = %d",
              m.result());
    state <= 2;
  endrule
endmodule: mkTest
```

Instantiating the
mkMult module

Invoking mkMult's
methods

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-6

Module and interface instantiation

```

module mkTest (Empty) ;
    Reg#(int) state <- mkReg(0);
    Mult_ifc m      <- mkMult1();
    ...
endmodule

module mkMult1 (Mult_ifc);
    ...
    ...
    ...
endmodule

```

- ◆ Modules instantiate other modules
 - Just like instantiating primitive state elements like registers

- ◆ Standard module-instantiation shorthand:

- This:

```
Mult_ifc m <- mkMult1();
```

- is shorthand for:

```

Mult_ifc m();           (interface instantiation)
mkMult1 mult_inst(m);  (module instantiation)

```

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-7

Methods are invoked from rules

```

module mkTest (Empty) ;
    ...
    Mult_ifc m <- mkMult1();

    rule go (state==0);
        m.start(9,5);
        state <= 1;
    endrule
    ...
endmodule

module mkMult1 (Mult_ifc);
    ...
    ...
    ...
    method Action start (x, y)
        if (r == 0);
        d <= x; r <= y;
    endmethod
    ...
endmodule

```

- ◆ Rule condition: `state==0 && r==0`
 - Conjunction of explicit (`state==0`) and implicit (`r==0`) conditions
- ◆ Rule actions: `state<=1, d<=9` and `r<=5`
 - Thus, a part of the rule's action is in a different module, behind a method invocation

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-8

Three Method Forms

BSV method calls look like function and procedure calls:

- ◆ *Value methods*: Functions which take 0 or more arguments and return a value
`x = m.result()`
- ◆ *Action methods*: Procedures which take 0 or more arguments and perform an action
`m.start(x)`
- ◆ *Actionvalue methods*: Procedures which take 0 or more arguments, perform an action, and return a result.
`x <- m.pop()`

Value methods can be called from any expression but action or actionvalue methods can be called only from a rule or a method body (not from a rule or method predicate)

Methods as ports

- ◆ Interface method types can be interpreted directly as I/O wires of a module:
 - Arguments are input signals
 - Return values are output signals
 - An implicit condition is an output "ready" signal
 - An Action type (side-effect) indicates an incoming "enable" signal

Methods as ports: Mult_ifc

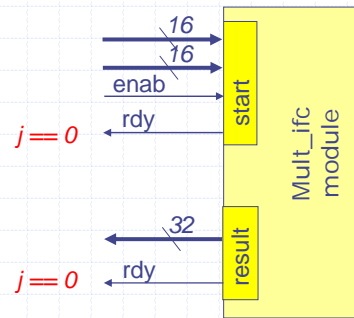
```
interface Mult_ifc;
    method Action start (Tin x, Tin y);
    method Tout    result ();
endinterface
```

start:

- 16-bit arguments
- has side effect (action)

result:

- no argument
- 32-bit result
- no side effect



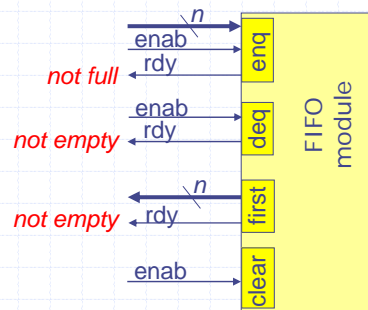
February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-11

Methods as ports: FIFO interface

```
interface FIFO #(type t);
    method Action enq(t); // enqueue an item
    method Action deq(); // remove oldest entry
    method t      first(); // inspect oldest item
    method Action clear(); // make FIFO empty
endinterface: FIFO
```



$n = \#$ of bits needed
to represent the
values of type "t"

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-12

Methods as ports: summary

- ◆ Methods can be viewed as a higher-level description of ports:
 - A method groups related ports together
 - ◆ e.g., data_in, RDY and ENABLE
 - Enforces the “micro-protocol”
 - ◆ Called only when ready
 - ◆ Strokes data at the right time
 - ◆ ... and more ...
- ◆ It is easy to relate the generated Verilog to the BSV source:
 - Transparent translation from methods to ports

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-13

Syntax note: “<-”

- ◆ “<-” is used in two ways:
 - Module instantiation shorthand
 - Invoking an ActionValue method

```
Queue#(int) q <- mkQueue;  
...  
rule r1 (...);  
  x <- q.pop();  
  ...  
endrule
```

- ◆ These two uses are distinguished by context

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-14

Two uses of "<-"

- ◆ In both uses, the operator
 - Has a side-effect
 - ◆ "instantiate a module"
 - ◆ "discard an element from the FIFO"
 - And returns a value
 - ◆ "return the interface"
 - ◆ "return the discarded FIFO element"
- ◆ In one case these happen during static elaboration
- ◆ In the other case these happen dynamically (during HW execution)

Sharing methods

A method can be invoked from more than one rule

```
module mkTest (...);
...
FIFO#(int) f <- mkFIFO();
...
rule r1 (... cond1 ...);
...
f.enq (... expr1 ...);
...
endrule

rule r2 (... cond2 ...);
...
f.enq (... expr2 ...);
...
endrule
endmodule: mkTest
```

```
interface FIFO#(type t);
  Action enq (t n);
  ...
endinterface

module mkFIFO (...);
...
method enq (x) if (... notFull ...);
...
endmethod
...
endmodule: mkFIFO
```

(In general the two invoking rules could be in different modules)

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-17

Sharing methods

- ◆ In software, to call a function/procedure from two processes just means:
 - Create two instances (usually on two stacks)
- ◆ A BSV method represents real hardware
 - There is only one instance (per instantiated module)
 - It is a *shared* resource
 - Parallel accesses must be *scheduled (controlled)*
 - Data inputs and outputs must be *muxed/ distributed*
- ◆ The BSV compiler inserts logic to accomplish this sharing
 - This logic is not an artifact of using BSV—it is logic that the designer would otherwise have to design manually

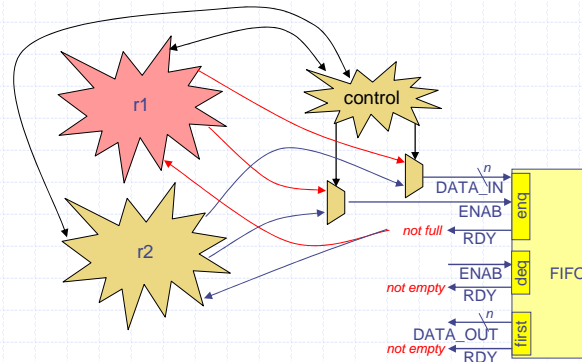
February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-18

Sharing a method

The compiler inserts logic for sharing a method



February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-19

Important special cases

- Value methods without arguments need no muxing or control, since they have no inputs into the module
 - Examples:
 - ♦ `r._read` for a register
 - ♦ `f.first` for a FIFO
 - Note: these methods are combinational functions, but they depend on the module's internal state
- (Advanced topic) BSV primitives can specify a *replication* factor for certain methods, so two calls to the "same" method actually get connected (automatically) to different replicas of the method
 - E.g., a read method of a multi-ported register file

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-20

Interface variations

- ◆ It is the designer's choice how to expose the functionality of a module using interface methods
- ◆ E.g., a FIFO can have several interfaces

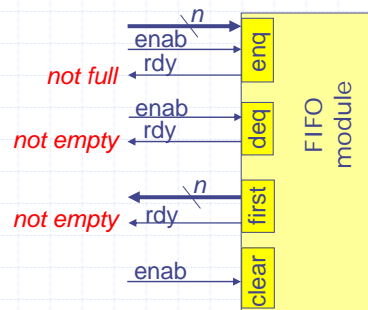
February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-21

A FIFO interface

```
interface FIFO #(type t);  
  method Action enq(t); // enqueue an item  
  method Action deq(); // remove oldest entry  
  method t      first(); // inspect oldest item  
  method Action clear(); // make FIFO empty  
endinterface: FIFO
```



n = # of bits needed to represent the values of type "t"

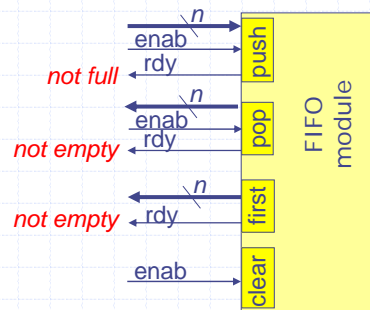
February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-22

Another FIFO interface: Combine first & deq

```
interface FIFO #(type t);
  method Action  push(t);           // enqueue an item
  method ActionValue#(t) pop();    // remove oldest entry
  method t       first();          // inspect oldest item
  method Action  clear();          // make FIFO empty
endinterface: FIFO
```



may or may not provide
"first" method

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-23

FIFO: Explicit ready signals

- ◆ The designer might want to expose the implicit ready signals—the `notFull` and `notEmpty` signals:

```
interface FIFOF#(type aType);
  ... enq ... first ... deq ... clear
  method Bool notFull();
  method Bool notEmpty();
endinterface
```

- ◆ The original `enq/deq/first` methods may or may not be protected by implicit conditions, depending on the module implementation

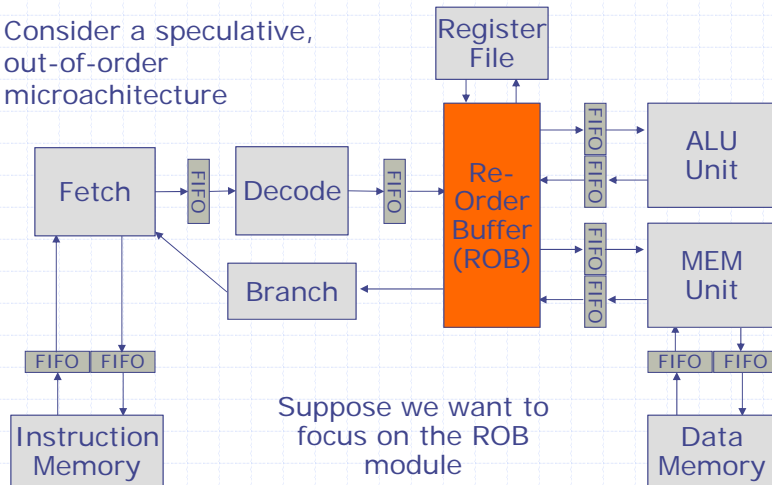
February 22, 2005

<http://csg.csail.mit.edu/6.884/>

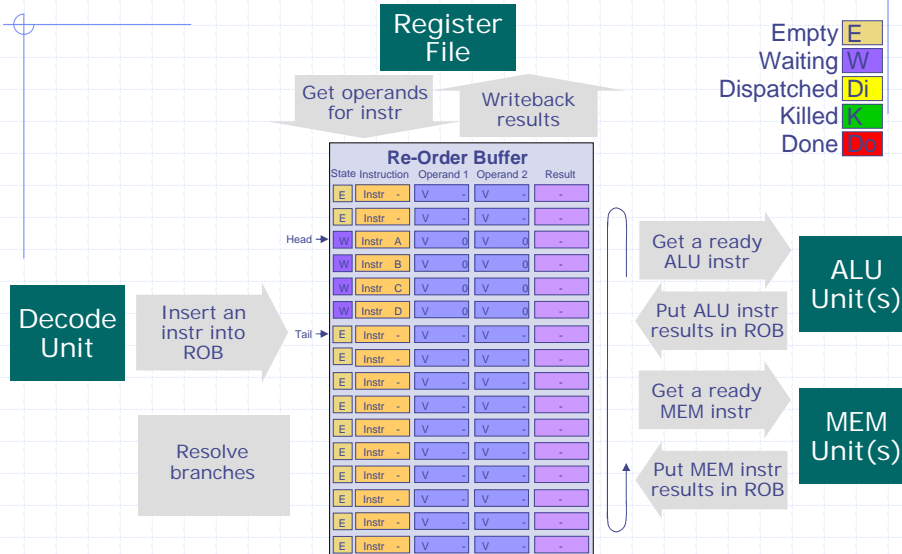
L07-24

Modularizing your design

- Consider a speculative, out-of-order microarchitecture

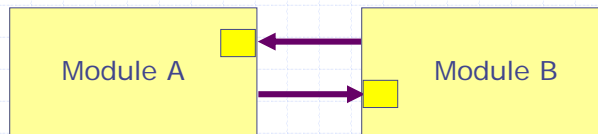


ROB actions



Modularizing your design

- ◆ A natural organization for two modules may be “recursive”.



but unfortunately BSV does not handle recursive module calls ...

February 22, 2005

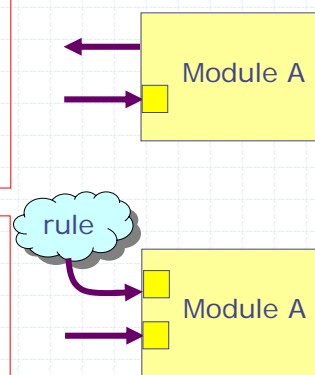
<http://csg.csail.mit.edu/6.884/>

L07-27

A rule that calls a method can be turned into a method

```
module moduleA (InterfaceA);  
  rule foo(True);  
    MsgTypeB msg <-  
    modB.getMessage();  
    <use msg>  
  endrule  
endmodule: moduleA
```

```
module ModuleA (InterfaceA);  
  method foo(MsgTypeB msg);  
    <use msg>  
  endmethod  
endmodule: moduleA
```



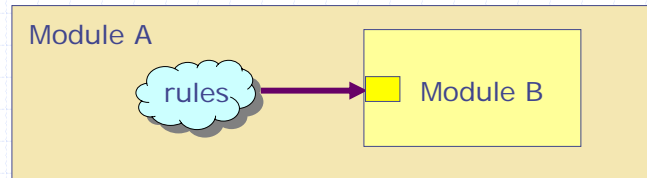
February 22, 2005

<http://csg.csail.mit.edu/6.884/>

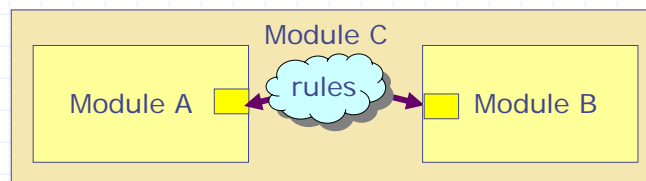
L07-28

Alternative Modularization

- ◆ Put one module inside the other



- ◆ Create a new module and put both modules inside it. Provide rules to pass values inbetween



February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-29

Glue module code ...

```
module mkTest (Empty);

  InterfaceA modA <- mkModuleA();
  InterfaceB modB <- mkModuleB();

  rule messagefromAtoB (True);
    MsgTypeA msg <- modA.getMessageToB();
    modB.handleMessageFromA(msg);
  endrule

  rule messagefromBtoA (True);
    MsgTypeB msg <- modB.getMessageToA();
    modA.handleMessageFromB(msg);
  endrule

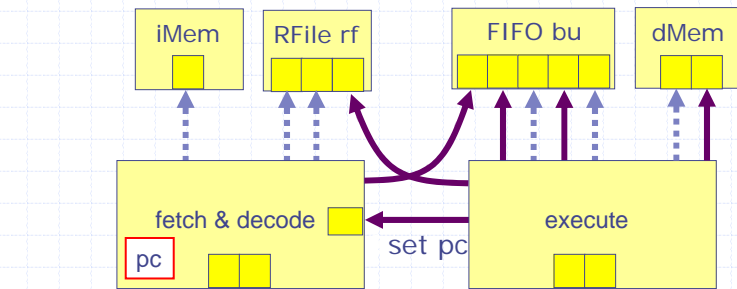
endmodule: mkTest
```

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-30

Modular organization: Two Stage Pipeline



-> Read method call
- > Action method call

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-31

Summary

- ◆ An *interface type* (e.g., `Multi_ifc`) specifies the prototypes of the *methods* in such an interface
 - The same interface can be provided by many modules
- ◆ Module definition:
 - A *module header* specifies the interface type *provided* (or *implemented*) by the module
 - Inside a module, each method is defined
 - ◆ Can contain implicit conditions, actions and returned values
 - Many module definitions can provide the same interface
- ◆ Module use:
 - An interface and a module are instantiated
 - Interface methods can be invoked from rules and other methods
 - ◆ Method implicit conditions contribute to rule conditions
 - ◆ Method actions contribute to rule actions

⇒ rule semantics extends smoothly across module boundaries

February 22, 2005

<http://csg.csail.mit.edu/6.884/>

L07-32