Bluespec-3: Modules & Interfaces

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Based on material prepared by Bluespec Inc, January 2005

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 → action

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

Courtesy of BlueSpec Inc. Used with permission.
**Example 1:**

Simple binary multiplication

```vhdl
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
```

**State**

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

**Behavior**

```
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
```

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

---

**Example 1: Modularized**

```vhdl
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
```

**Interface**

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

**State**

```
Reg#(Tout) product <- mkReg (0);
Reg#(Tout) d       <- mkReg (0);
Reg#(Tin)  r       <- mkReg (0);
```

**Behavior**

```
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
```

---

February 22, 2005
Interfaces

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)

module mkMult1 (Mult_ifc);
  ...
endmodule

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
Module and interface instantiation

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();
mkMult1 mult_inst(m);

Methods are invoked from rules

- 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
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
  \[ x = m.\text{result}() \]
- **Action methods**: Procedures which take 0 or more arguments and perform an action
  \[ m.\text{start}(x) \]
- **Actionvalue methods**: Procedures which take 0 or more arguments, perform an action, and return a result.
  \[ x \leftarrow m.\text{pop}() \]

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)
j == 0

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

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"
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
    - Strobes 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

Syntax note: “<-”

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

```verilog
Queue#(int) q <- mkQueue;

rule r1 (...);
  x <- q.pop();
endrule
```

- These two uses are distinguished by context
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

```verilog
module mkFIFO (...);
    ...
    interface FIFO#(type t);
        Action enq (t n);
    endinterface
    module mkFIFO (...);
        ...
        method enq (x) if (... notFull ...);
    endmethod
    endmodule: mkFIFO
    ...
rule r1 (... cond1 ...);
    ...
    f.enq (... expr1 ...);
endrule
rule r2 (... cond2 ...);
    ...
    f.enq (... expr2 ...);
endrule
endmodule: mkTest
```

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

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
Sharing a method

The compiler inserts logic for sharing a method

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
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.

A FIFO interface

```verilog
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”
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
```

FIFO:
Explicit ready signals

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

```vhdl
interface FIFOF#(type aType);
    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.
Modularizing your design

Consider a speculative, out-of-order microarchitecture.

Suppose we want to focus on the ROB module.

ROB actions

- Insert an instr into ROB
- Get operands for instr
- Writeback results
- Get a ready ALU instr
- Put ALU instr results in ROB
- Get a ready MEM instr
- Put MEM instr results in ROB

Resolve branches

ALU Unit(s)

MEM Unit(s)
Modularizing your design

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

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

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
Alternative Modularization

- Put one module inside the other

Module A

rules

Module B

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

Module A

rules

Module B

Module C

Glue module code ...

```plaintext
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
```
Modular organization: Two Stage Pipeline

[Diagram of a two-stage pipeline with labeled components: iMem, RFile, FIFO, dMem, fetch & decode, execute, and set pc. Arrows indicate the flow: Read method call and Action method call.]

Summary

- An interface type (e.g., Mult_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