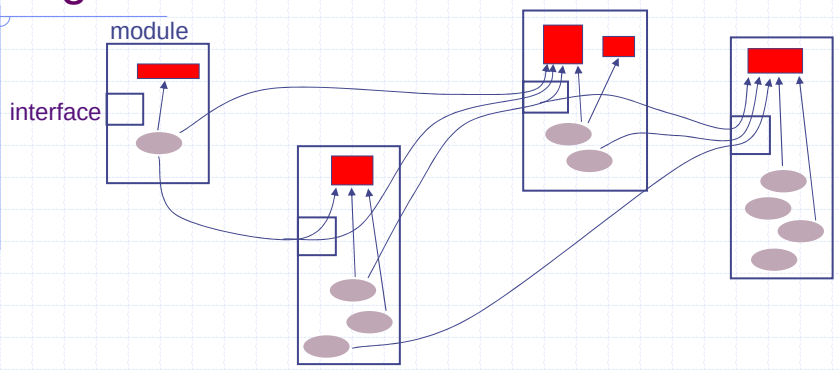
Bluespec-2: Designing with Rules

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.

Rules

- A rule is *declarative* specification of a state transition
  - An action guarded by a Boolean condition

rule *ruleName (<predicate>);*
<action>
endrule

Example 1: simple binary multiplication

```
   1001    // multiplicand (d) = 9
 x 0101    // multiplier (r) = 5
------
   1001    // d << 0 (since r[0] == 1)
  0000     // 0 << 1 (since r[1] == 0)
 1001      // d << 2 (since r[2] == 1)
0000       // 0 << 3 (since r[3] == 0)
-------
0101101    // product (sum of above) = 45
```

(Note: this is just a basic example; there are many sophisticated algorithms for multiplication in the literature)

Example 1: simple binary multiplication

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

module mkMult0 ();
  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—registers (module instantiation)

Behavior

Module Syntax

- Module declaration

module name

```
module mkMult0 ();
  …
endmodule: mkMult0
```

- Module instantiation
  - short form

interface type — *interface type's parameter(s)* — *interface instance* — *module name* — *module's parameter(s)*

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

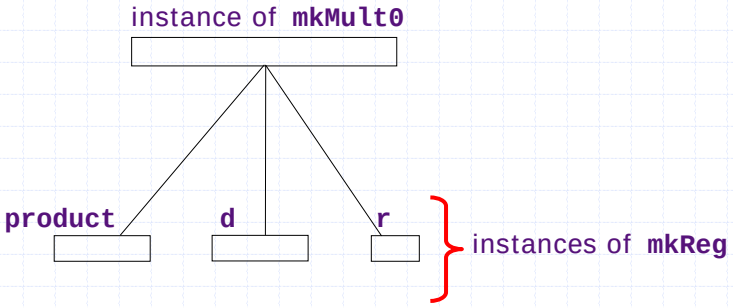
  - long form

```
Reg#(Tout)      product(); // interface
mkReg#(0)       the_product(product);
                           // the instance
```

Variables

- Variables have a type and name values
  `Tin d_init = 9, r_init = 5;`
- Variables never represent *state*
  - I.e., they do not remember values over time
  - They are always like wires, i.e., a variable just represents the value it is assigned
- State is obtained only by module instantiation

The module hierarchy



As in Verilog, module instances can be nested, i.e., the tree can be deeper.

All state elements are at the leaves

Example 1 in Verilog RTL

```
module mkMult0 (CLK, RST_N);
  input CLK;
  input RST_N;

  reg [31:0] product = 0;
  reg [31:0] d       = 9;
  reg [15:0] r       = 5;

  always @ (posedge CLK)
    if (r != 0) begin
      if (r[0] == 1) product <= product + d;
      d <= d << 1;
      r <= r >> 1;
    end
    else
      $display ("Product = %d", product);

endmodule: mkMult0
```

Very similar!

Over-simplified analogy with Verilog process

- In this simple example, a rule is reminiscent of an "always" block:

```
rule rname (<cond>); <action> endrule


always@(posedge CLK)
    if (<cond>) begin: rname
        <action>
    end
```

- But this is not true in general:
  - Rules have interlocks—becomes important when rules share resources, to avoid race conditions
  - Rules can contain *method calls*, invoking actions in other modules

Rule semantics

Given a set of rules and an initial state

while (some rules are applicable*
in the current state)

- choose *one* applicable rule
- apply that rule to the current state to produce the next state of the system**

(*) "applicable" = a rule's condition is true in current state

(**) These rule semantics are "untimed" – the action to change the state can take as long as necessary provided the state change is seen as atomic, i.e., not divisible.

Example 2: Concurrent Updates

- Process 0 increments register x;
  Process 1 transfers a unit from register x to register y;
  Process 2 decrements register y

- This is an abstraction of some real applications:
  - Bank account: 0 = deposit to checking, 1 = transfer from checking to savings, 2 = withdraw from savings
  - Packet processor: 0 = packet arrives, 1 = packet is processed, 2 = packet departs
  - ...

Concurrency in Example 2

- Process *j* (= 0,1,2) only updates under condition *condj*
- Only one process at a time can update a register. Note:
  - Process 0 and 2 can run concurrently if process 1 is not running
  - Both of process 1's updates must happen "indivisibly" (else inconsistent state)
- Suppose we want to prioritize process 2 over process 1 over process 0

Example 2 Using Rules

```
(* descending_urgency = "proc2, proc1, proc0" *)

rule proc0 (cond0);
    x <= x + 1;
endrule

rule proc1 (cond1);
    y <= y + 1;
    x <= x - 1;
endrule

rule proc2 (cond2);
    y <= y - 1;
endrule
```

Functional correctness follows directly from rule semantics

Related actions are grouped naturally with their conditions—easy to change

Interactions between rules are managed by the compiler (scheduling, muxing, control)

Example 2 in Verilog: Explicit concurrency control

```
always @(posedge CLK) // process 0
  if ((!cond1 || cond2) && cond0)
    x <= x + 1;
```

will make it incorrect

```
always @(posedge CLK) // process 1
  if (!cond2 && cond1) begin
    y <= y + 1;
    x <= x – 1;
  end

always @(posedge CLK) // process 2
  if (cond2)
    y <= y – 1;
```

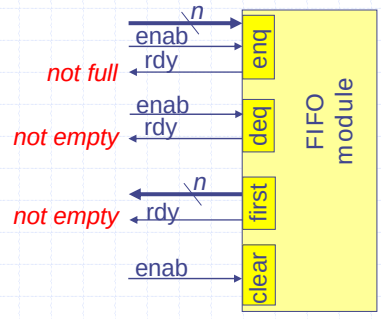
```
always @(posedge CLK) begin
  if (!cond2 && cond1)
    x <= x – 1;
  else if (cond0)
    x <= x + 1;

  if (cond2)
    y <= y – 1;
  else if (cond1)
    y <= y + 1;
end
```

Another solution

Are these solutions correct?
How to verify that they're correct?
What needs to change if the conds change?
What if the processes are in different modules?

A FIFO interface

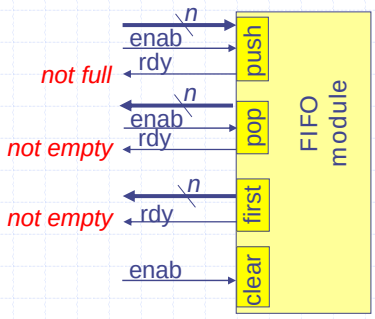
```
interface  FIFO #(type t);
  method Action  enq(t);   // enqueue an item
  method Action  deq();    // remove oldest entry
  method t       first();  // inspect oldest item
  method Action  clear();  // make FIFO empty
endinterface: FIFO
```



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

Actions that return Values: Another FIFO interface

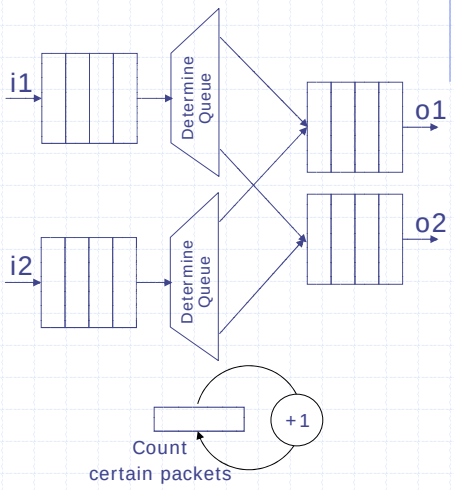
```
interface  FIFO #(type t);
  method Action  push(t);          // enqueue an item
  method ActionValue#(t)  pop();  // remove oldest entry
  method t       first();          // inspect oldest item
  method Action  clear();          // make FIFO empty
endinterface: FIFO
```



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

Example 3: A 2x2 switch, with stats

- Packets arrive on two input FIFOs, and must be switched to two output FIFOs
  - dest(pkt) ∈ {1,2}
- Certain "interesting packets" must be counted
  - interesting(pkt) ∈ {True, False}

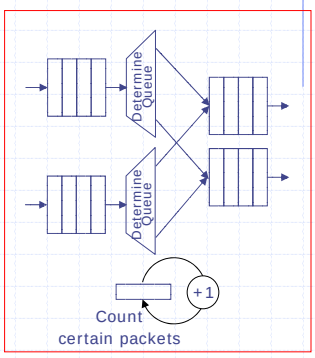


Example 3: Specifications

- Input FIFOs can be empty
- Output FIFOs can be full
- Shared resource collision on an output FIFO:
  - if packets available on both input FIFOs, both have same destination, and destination FIFO is not full
- Shared resource collision on counter:
  - if packets available on both input FIFOs, each has different destination, both output FIFOs are not full, and both packets are "interesting"
- Resolve collisions in favor of packets from the first input FIFO
- Must have maximum throughput: a packet must move if it can, modulo the above rules

Rules for Example 3

```
(* descending_urgency = "r1, r2" *)
// Moving packets from input FIFO i1
rule r1;
  Tin x = i1.first();
  if (dest(x)== 1) o1.enq(x);
  else             o2.enq(x);
  i1.deq();
  if (interesting(x)) c <= c + 1;
endrule

// Moving packets from input FIFO i2
rule r2;
  Tin x = i2.first();
  if (dest(x)== 1) o1.enq(x);
  else             o2.enq(x);
  i2.deq();
  if (interesting(x)) c <= c + 1;
endrule
```



Notice, the rules have no explicit predicates, only actions

Example 3: Commentary

- Muxes and their control for output FIFOs and Counter are generated automatically
- FIFO emptiness and fullness are handled automatically
  - *Rule* and interface *method* semantics make it
    - *Impossible* to read a junk value from an empty FIFO
    - *Impossible* to enqueue into a full FIFO
    - *Impossible* to race for multiple enqueues onto a FIFO
  - No magic -- equally available for user-written module interfaces
- All control for resource sharing handled automatically
  - Rule *atomicity* ensures consistency
  - The "descending_urgency" attribute resolves collisions in favor of rule r1

Example 3: Changing Specs

- Now imagine the following changes to the existing code:
  - Some packets are multicast (go to both FIFOs)
  - Some packets are dropped (go to no FIFO)
  - More complex arbitration
    - FIFO collision: in favor of r1
    - Counter collision: in favor of r2
    - Fair scheduling
  - Several counters for several kinds of interesting packets
  - Non-exclusive counters (e.g., TCP ➔ IP)
  - M input FIFOs, N output FIFOs (parameterized)
- Suppose these changes are required 6 months after original coding
  - Rules based designs provide flexibility, robustness, correctness, ...

Example 4: Shifter

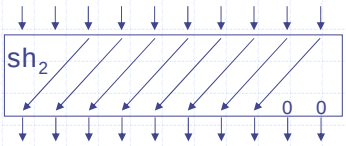
- Goal: implement: y = shift (x,s)

  where y is x shifted by s positions.
  Suppose s is a 3-bit value.

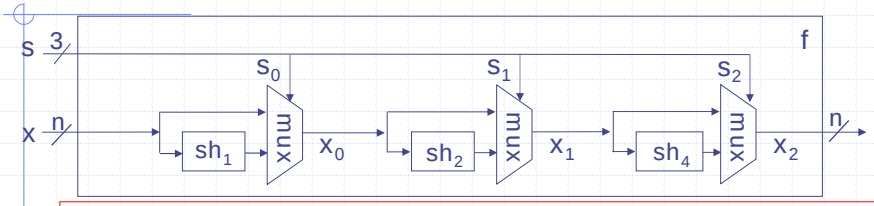
- Strategy:
  - Shift by s =

shift by	4 ($=2^2$)	if s[2] is set,
and by	2 ($=2^1$)	if s[1] is set,
and by	1 ($=2^0$)	if s[0] is set

  - A shift by 2^j is trivial: it's just a "lane change" made purely with wires



Cascaded Combinational Shifter

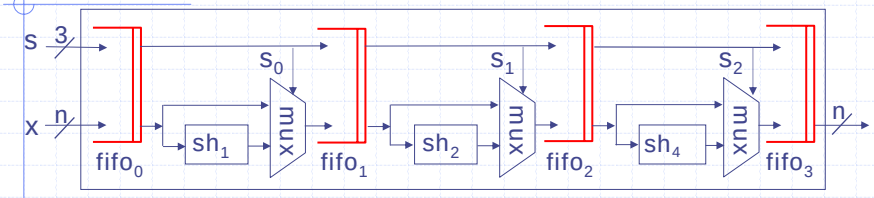


A family of functions

```
function Pair step_j (Pair sx);          where k=2^j
 return ((sx.s[j]==0) ? sx  :
                        Pair{s: sx.s,x:sh_k(sx.x)});
endfunction
function int shifter (int s,int x);
  Pair sx0, sx1, sx2;
  sx0 = step_0(Pair{s:s, x:x});
  sx1 = step_1(sx0);
  sx2 = step_2(sx1);
  return (sx2.x);
endfunction
```

```
typedef struct
        {int x; int s;}
         Pair;
```

Asynchronous pipeline with FIFOs (regs with interlocks)




```
rule stage_1;
  Pair sx0 <- fifo0.pop();  fifo1.push(step_0(sx0));
endrule
```

```
rule stage_2;
  Pair sx1 <- fifo1.pop();  fifo2.push(step_1(sx1));
endrule
```

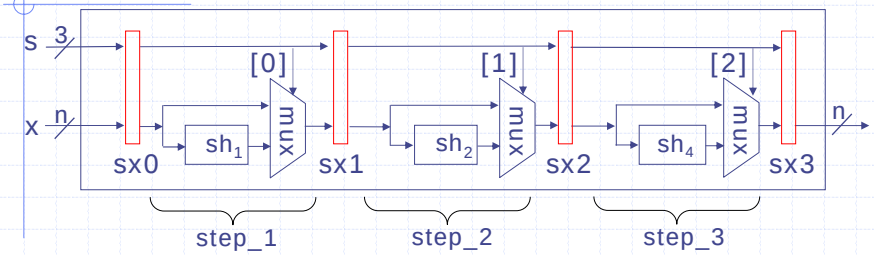
```
rule stage_3;
  Pair sx2 <- fifo2.pop();  fifo3.push(step_2(sx2));
endrule
```


Required simultaneity

If it is *necessary* for several actions to happen together, (i.e., indivisibly, atomically)

Put them in the same rule!

Synchronous pipeline (with registers)




```
rule sync-shifter;
  sx1 <= step_0(sx0);
  sx2 <= step_1(sx1);
  sx3 <= step_2(sx2);
endrule
```

sx1, sx2 and sx3 are registers defined outside of the rules

Discussion

- In the synchronous pipeline, we compose actions in parallel
  - All stages move data simultaneously, in lockstep (atomic!)
- In the asynchronous pipeline, we compose rules in parallel
  - Stages can move independently (each stage can move when its input fifo has data and its output fifo has room)
  - If we had used parallel action composition instead, all stages would have to move in lockstep, and could only move when all stages were able to move
- Your design goals will suggest which kind of composition is appropriate in each situation

Summary: Design using Rules

- Much easier to reason about correctness of a system when you consider just one rule at a time
- No problems with concurrency (e.g., race conditions, mis-timing, inconsistent states)
  - We also say that rules are “interlocked”

➔ *Major impact on design entry time and on verification time*

Types and Syntax notes

Types and type-checking

- BSV is strongly-typed
  - Every variable and expression has a *type*
  - The Bluespec compiler performs strong type checking to guarantee that **values are used only in places that make sense**, according to their type
- This catches a huge class of design errors and typos at compile time, i.e., before simulation!

SV notation for types

- Some types just have a name

```
int, Bool, Action, …
```

- More complex types can have *parameters* which are themselves types

```
FIFO#(Bool)                  // fifo containing Booleans
Tuple2#(int,Bool)            // pair of items: an int and a Boolean
FIFO#(Tuple2#(int,Bool))     // fifo containining pairs of ints
                             // and Booleans
```

Numeric type parameters

- BSV types also allows *numeric* parameters

```
Bit#(16)                 // 16-bit wide bit-vector
Int#(29)                 // 29-bit wide signed integers
Vector#(16,Int#(29))     // vector of 16 Int#(29) elements
```

- These numeric types should not be confused with numeric values, even though they use the same number syntax
  - The distinction is always clear from context, i.e., type expressions and ordinary expressions are always distinct parts of the program text

A synonym for bit-vectors:

- Standard Verilog notation for bit-vectors is just special syntax for the general notation

`bit[15:0]` is the same as `Bit#(16)`

Common scalar types

- **`Bool`**
  - Booleans
- **`Bit#(n)`**
  - Bit vectors, with a width n bits
- **`Int#(n)`**
  - Signed integers of n bits
- **`UInt#(n)`**
  - Unsigned integers of n bits

Types of variables

- Every variable has a *data type*:

```
bit[3:0] vec;    // or    Bit#(4) vec;
vec = 4'b1010;
Bool cond = True;
typedef struct { Bool b; bit[31:0] v; } Val;
Val x = { b: True, v: 17 };
```

- BSV will enforce proper usage of values according to their types
  - You can't apply "+" to a struct
  - You can't assign a boolean value to a variable declared as a struct type

"let" and type-inference

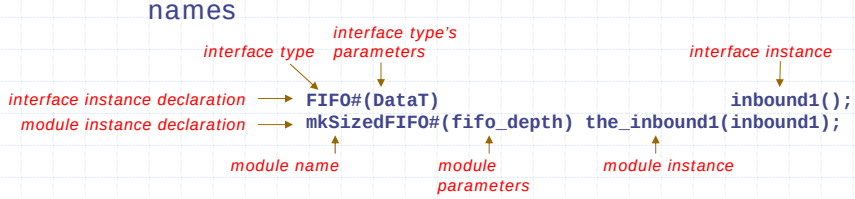
- Normally, every variable is introduced in a declaration (with its type)
- The "let" notation introduces a variable with an assignment, with the compiler inferring its correct type

```
let vec = 4'b1010;     // bit[3:0] vec = …
let cond = True;       // Bool cond = …;
```

- This is typically used only for very "local" temporary values, where the type is obvious from context

Instantiating interfaces and modules

- The SV idiom is:
  - Instantiate an interface
  - Instantiate a module, binding the interface
    - Note: the module instance name is generally not used, except in debuggers and in hierarchical names



- BSV also allows a shorthand:

```
FIFO#(DataT) inbound1 <- mkSizedFIFO(fifo_depth);
```

Rule predicates

- The *rule predicate* can be any Boolean expression
  - Including function calls and method calls
- Cannot have a side-effect
  - This is enforced by the type system
- The predicate must be true for rule execution
  - But in general, this is not enough
  - Sharing resources with other rules may constrain execution

Why not "`reg x;`"?

- *Unambiguity*: In V and SV, "`reg x;`" is a variable declaration which may or may not turn into a HW register
- *Uniformity*: BSV uses SV's module-instantiation mechanism uniformly for primitives and user-defined modules
- *Strong typing*: Using SV's module-instantiation mechanism enables polymorphic, strongly-typed registers