

Introduction to Bluespec: A new methodology for designing Hardware

Arvind

Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-1

What is needed to make hardware design easier

- ◆ Extreme IP reuse “Intellectual Property”
 - Multiple instantiations of a block for different performance and application requirements
 - Packaging of IP so that the blocks can be assembled easily to build a large system (black box model)
- ◆ Ability to do modular refinement
- ◆ Whole system simulation to enable concurrent hardware-software development

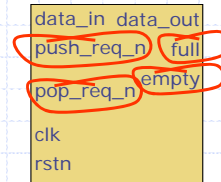
February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-2

IP Reuse sounds wonderful until you try it ...

Example: Commercially available FIFO IP block



An error occurs if a push is attempted while the FIFO is full.

Thus, there is no conflict in a simultaneous push and pop when the FIFO is full. A simultaneous push and pop cannot occur when the FIFO is empty, since there is no pop data to prefetch. However, push data is captured in the FIFO.

A pop operation occurs when pop_req_n is asserted (LOW), as long as the FIFO is not empty. Asserting pop_req_n causes the internal read pointer to be incremented on the next rising edge of clk. Thus, the RAM read data must be captured on the clk following the assertion of pop_req_n.

These constraints are spread over many pages of the documentation...

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-3

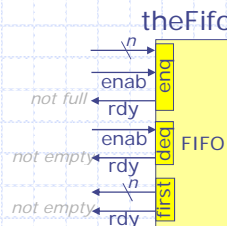
Bluespec promotes composition through guarded interfaces

theModuleA

```
theFifo.enq(value1);  
  
theFifo.deq();  
value2 = theFifo.first();
```

theModuleB

```
theFifo.enq(value3);  
  
theFifo.deq();  
value4 = theFifo.first();
```



February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-4

Bluespec: A new way of expressing behavior using Guarded Atomic Actions

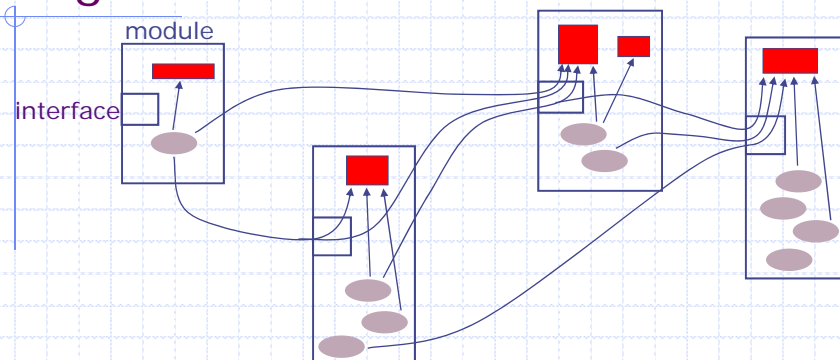
- ◆ Formalizes composition
 - Modules with guarded interfaces
 - Compiler manages connectivity (muxing and associated control)
- ◆ Powerful static elaboration facility
 - Permits parameterization of designs at all levels
- ◆ Transaction level modeling
 - Allows C and Verilog codes to be encapsulated in Bluespec modules

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-5

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: guard → action

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

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-6

GCD: A simple example to explain hardware generation from Bluespec

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-7

Programming with rules: A simple example

Euclid's algorithm for computing the Greatest Common Divisor (GCD):

15

6

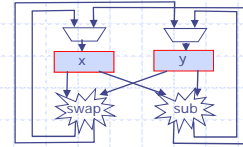
answer:

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-8

GCD in BSV



```
module mkGCD (I_GCD);
```

```
  Reg#(Int#(32)) x <- mkRegU;
  Reg#(Int#(32)) y <- mkReg(0);
```

State

```
  rule swap ((x > y) && (y != 0));
    x <= y; y <= x;
  endrule
  rule subtract ((x <= y) && (y != 0));
    y <= y - x;
  endrule
```

Internal behavior

```
  method Action start(Int#(32) a, Int#(32) b)
    if (y==0);
      x <= a; y <= b;
  endmethod
  method Int#(32) result() if (y==0);
    return x;
  endmethod
```

External Interface

```
endmodule
```

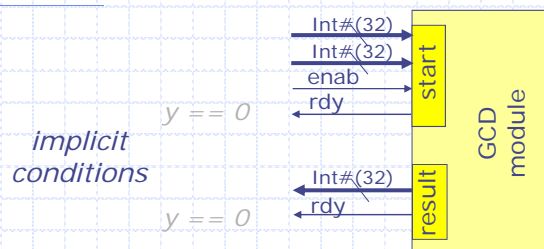
Assume a/=0

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-9

GCD Hardware Module



```
interface I_GCD;
  method Action start (Int#(32) a, Int#(32) b);
  method Int#(32) result();
endinterface
```

- ◆ The module can easily be made polymorphic
- ◆ Many different implementations can provide the same interface:


```
module mkGCD (I_GCD)
```

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-10

GCD: Another implementation

```

module mkGCD (I_GCD);
  Reg#(Int#(32)) x <- mkRegU;
  Reg#(Int#(32)) y <- mkReg(0);

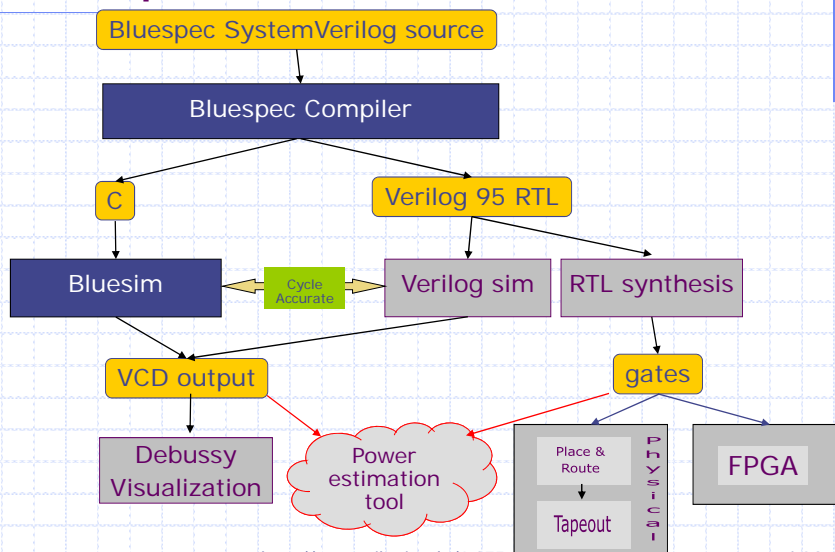
  rule swapANDsub ((x > y) && (y != 0));
    x <= y; y <= x - y;
  endrule
  rule subtract ((x<=y) && (y!=0));
    y <= y - x;
  endrule

  method Action start(Int#(32) a, Int#(32) b)
    if (y==0);
      x <= a; y <= b;
  endmethod
  method Int#(32) result() if (y==0);
    return x;
  endmethod
endmodule

```

Combine swap and subtract rule

Bluespec Tool flow



Generated Verilog RTL: GCD

```

module mkGCD(CLK,RST_N,start_a,start_b,EN_start,RDY_start,
             result,RDY_result);
    input  CLK; input  RST_N;
    // action method start
    input [31 : 0] start_a; input [31 : 0] start_b; input EN_start;
    output RDY_start;
    // value method result
    output [31 : 0] result; output RDY_result;
    // register x and y
    reg [31 : 0] x;
    wire [31 : 0] x$D_IN; wire x$EN;
    reg [31 : 0] y;
    wire [31 : 0] y$D_IN; wire y$EN;
    ...
    // rule RL_subtract
    assign WILL_FIRE_RL_subtract = x_SLE_y__d3 && !y_EQ_0__d10 ;
    // rule RL_swap
    assign WILL_FIRE_RL_swap = !x_SLE_y__d3 && !y_EQ_0__d10 ;
    ...

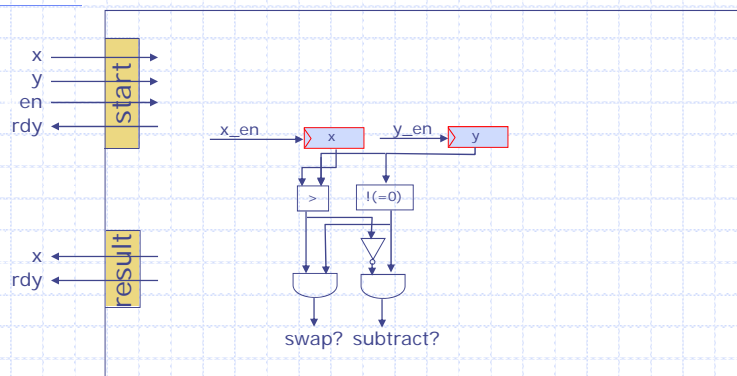
```

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-13

Generated Hardware



```

rule swap ((x>y)&&(y!=0));
    x <= y; y <= x; endrule
rule subtract ((x<=y)&&(y!=0));
    y <= y - x; endrule

```

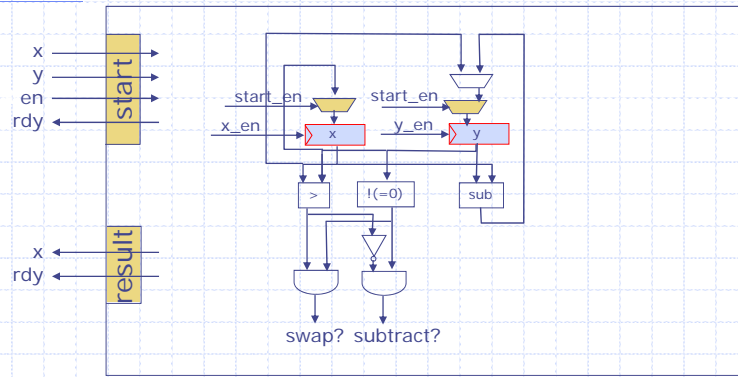
x_en =
y_en =

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-14

Generated Hardware Module



$x_en = \text{swap?}$
 $y_en = \text{swap? OR subtract?}$
 $rdy =$

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-15

GCD: A Simple Test Bench

```

module mkTest ();
    Reg#(Int#(32)) state <- mkReg(0);
    I_GCD gcd <- mkGCD();

    rule go (state == 0);
        gcd.start (423, 142);
        state <= 1;
    endrule

    rule finish (state == 1);
        $display ("GCD of 423 & 142 =%d",gcd.result());
        state <= 2;
    endrule
endmodule
    
```

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-16

GCD: Test Bench

```
module mkTest ();
  Reg#(Int#(32)) state <- mkReg(0);
  Reg#(Int#(4))   c1 <- mkReg(1);
  Reg#(Int#(7))   c2 <- mkReg(1);
  I_GCD           gcd <- mkGCD();

  rule req (state==0);
    gcd.start(signExtend(c1), signExtend(c2));
    state <= 1;
  endrule

  rule resp (state==1);
    $display ("GCD of %d & %d =%d", c1, c2, gcd.result());
    if (c1==7) begin c1 <= 1; c2 <= c2+1; end
    else c1 <= c1+1;
    if (c1==7 && c2==63) state <= 2 else state <= 0;
  endrule
endmodule
```

Feeds all pairs (c1,c2)
1 < c1 < 7
1 < c2 < 63
to GCD

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-17

GCD: Synthesis results

- ◆ Original (16 bits)
 - Clock Period: 1.6 ns
 - Area: 4240 μm^2
- ◆ Unrolled (16 bits)
 - Clock Period: 1.65ns
 - Area: 5944 μm^2
- ◆ Unrolled takes 31% fewer cycles on the testbench

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-18

Need for a rule scheduler

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-19

GAA Execution model

Repeatedly:

- ◆ Select a rule to execute
- ◆ Compute the state updates
- ◆ Make the state updates

Highly non-deterministic

User annotations can help in rule selection

Implementation concern: Schedule multiple rules concurrently without violating one-rule-at-a-time semantics

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-20

Example 1

```
rule ra (z > 10);  
  x <= x + 1;  
endrule  
  
rule rb (z > 20);  
  y <= y + 1;  
endrule
```

- ◆ Can these rules be enabled together?
- ◆ Can they be executed concurrently?

Example 2

```
rule ra (z > 10);  
  x <= y + 1;  
endrule  
  
rule rb (z > 20);  
  y <= x + 1;  
endrule
```

- ◆ Can these rules be enabled together?
- ◆ Can they be executed concurrently?

Example 3

```
rule ra (z > 10);  
  x <= y + z;  
endrule  
  
rule rb (z > 20);  
  y <= y + z;  
endrule
```

- ◆ Can these rules be enabled together?
- ◆ Can they be executed concurrently?

Rule: As a State Transformer

A rule may be decomposed into two parts $\pi(s)$ and $\delta(s)$ such that

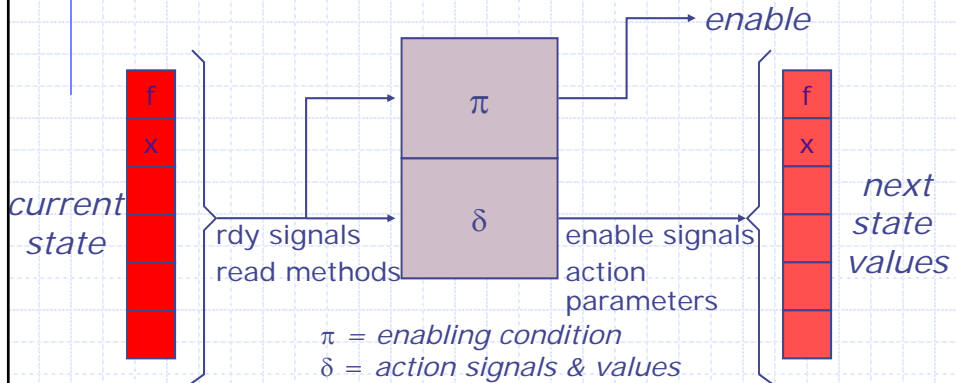
$$s_{next} = \text{if } \pi(s) \text{ then } \delta(s) \text{ else } s$$

$\pi(s)$ is the condition (predicate) of the rule, a.k.a. the "CAN_FIRE" signal of the rule. π is a conjunction of explicit and implicit conditions

$\delta(s)$ is the "state transformation" function, i.e., computes the next-state values from the current state values

Compiling a Rule

```
rule r (f.first() > 0) ;
    x <= x + 1 ; f.deq () ;
endrule
```



February 7, 2011

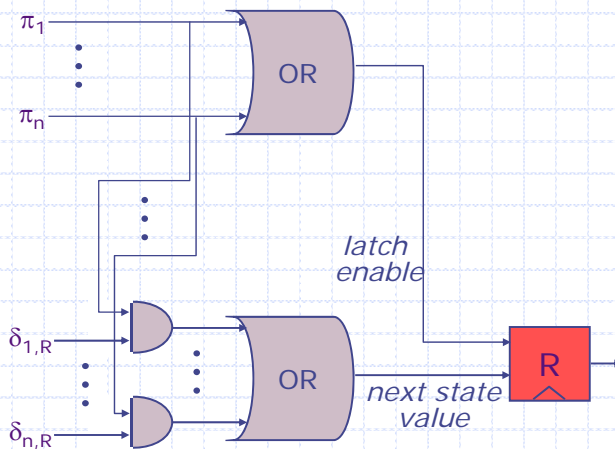
<http://csg.csail.mit.edu/6.375>

L02-25

Combining State Updates: strawman

π 's from the rules that update R

δ 's from the rules that update R

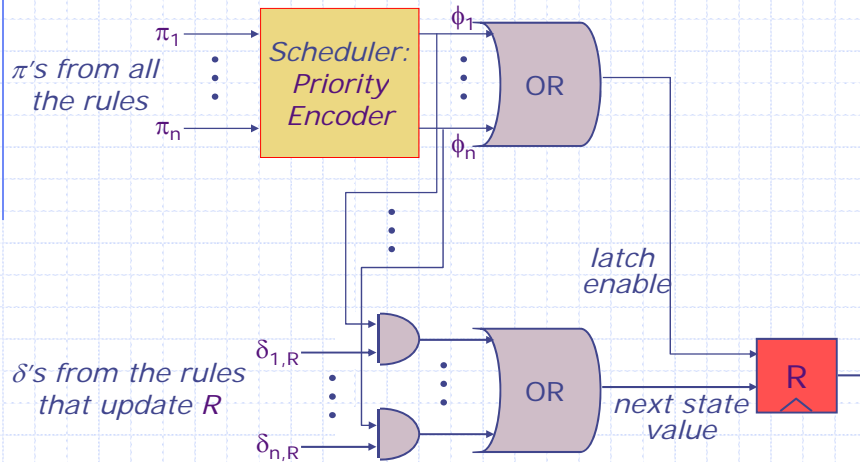


February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-26

Combining State Updates



Scheduler ensures that at most one ϕ_i is true

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-27

A compiler can determine if two rules can be executed in parallel without violating the one-rule-at-a-time semantics

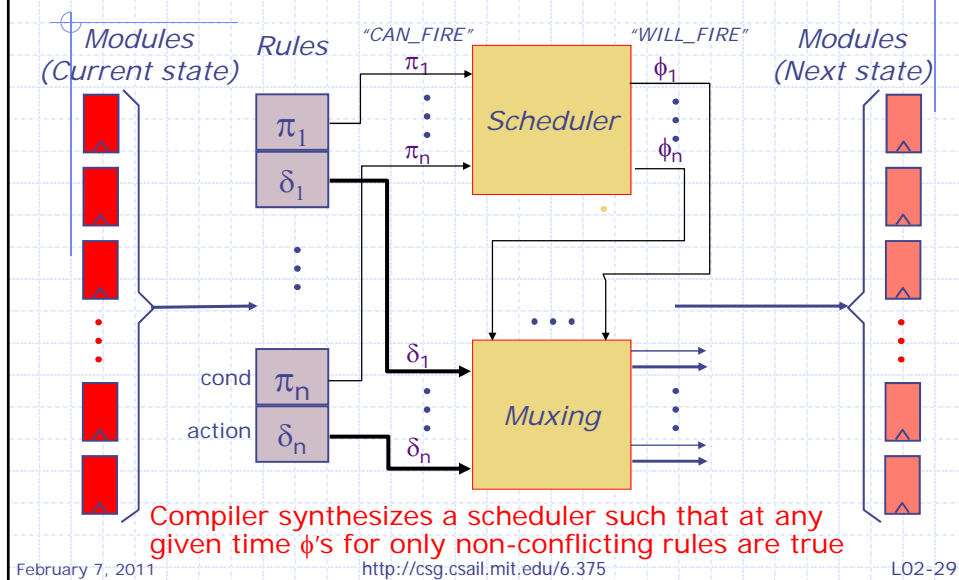
James Hoe, Ph.D., 2000

February 7, 2011

<http://csg.csail.mit.edu/6.375>

L02-28

Scheduling and control logic



The plan

- ◆ Express combinational circuits in Bluespec
- ◆ Express Inelastic pipelines
 - single-rule systems; no scheduling issues
- ◆ Multiple rule systems and concurrency issues
 - Eliminating dead cycles
- ◆ Elastic pipelines and processors
- ◆ FSM Library
- ◆ Hardware-Software codesign

Each idea would be illustrated via examples

Minimal discussion of Bluespec syntax in the lectures; you are suppose to learn that by yourself and in the lab sessions