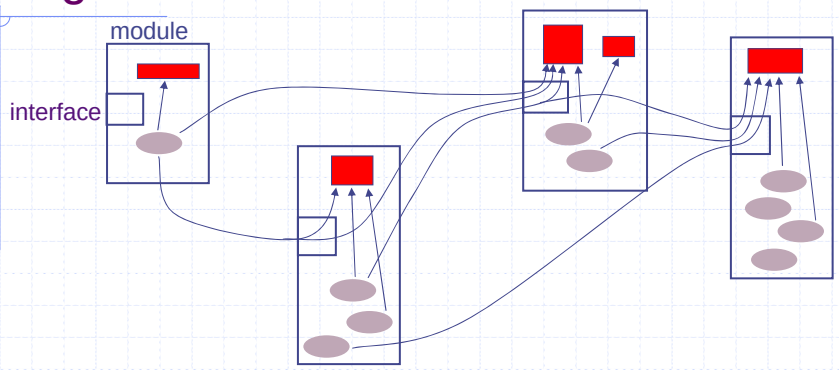
GCD: A simple example to introduce Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

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.

Programming with rules: A simple example

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

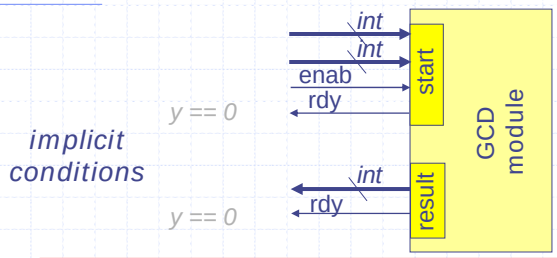
15 6

GCD in BSV

```
module mkGCD (I_GCD);
    Reg#(int) x <- mkRegU;
    Reg#(int) y <- mkReg(0);

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

    method Action start(int a, int b) if (y==0);
      x <= a;  y <= b;
    endmethod
    method int result() if (y==0);
        return x;
    endmethod
endmodule
```

Assume a/=0

GCD Hardware Module




```
interface I_GCD;
    method Action start (int a, int b);
    method int result();
endinterface
```

- The module can easily be made polymorphic
- Many different implementations can provide the same interface: `module mkGCD (I_GCD)`

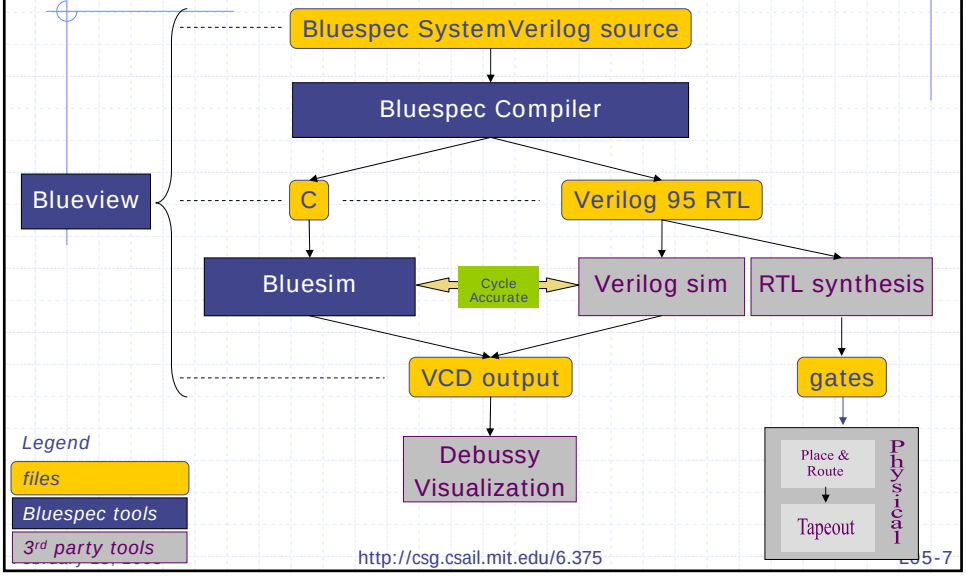
GCD: Another implementation

```
module mkGCD (I_GCD);
    Reg#(int) x <- mkRegU;
    Reg#(int) 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 a, int b) if (y==0);
      x <= a;  y <= b;
    endmethod
    method int result() if (y==0);
        return x;
    endmethod
endmodule
```

Combine swap and subtract rule

Does it compute faster ?

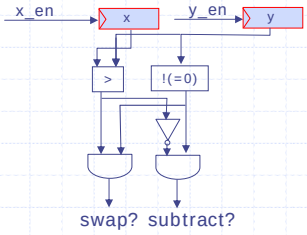
Bluespec Tool flow



Generated Verilog RTL: GCD

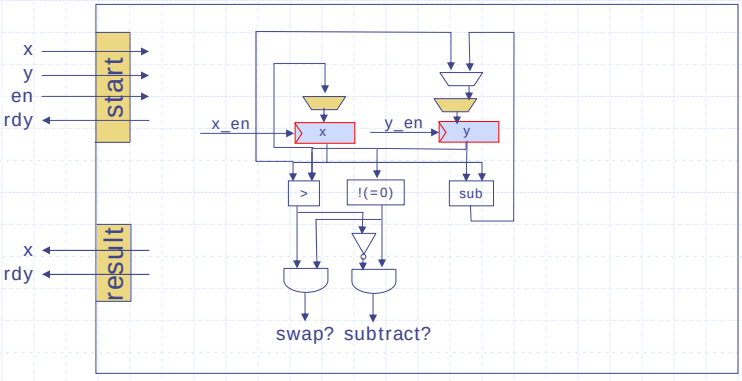
```
module mkGCD(CLK,RST_N,start_a,start_b,EN_start,RDY_start,
             result,RDY_result);
  input  CLK; input  RST_N;
// action method start
  input [31 : 0] start_a; input [31 : 0] start_b; input EN_start;
  output RDY_start;
// value method result
  output [31 : 0] result; output RDY_result;
// 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 ;
...
```

Generated Hardware



x_en =
y_en =

Generated Hardware Module



x_en = swap?
y_en = swap? OR subtract?
rdy =

GCD: A Simple Test Bench

```
module mkTest ();
  Reg#(int) 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
```


GCD: Test Bench

```
module mkTest ();
  Reg#(int)  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

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

Rule scheduling and the synthesis of a scheduler

GAA Execution model

Repeatedly:

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

Rule: As a State Transformer

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

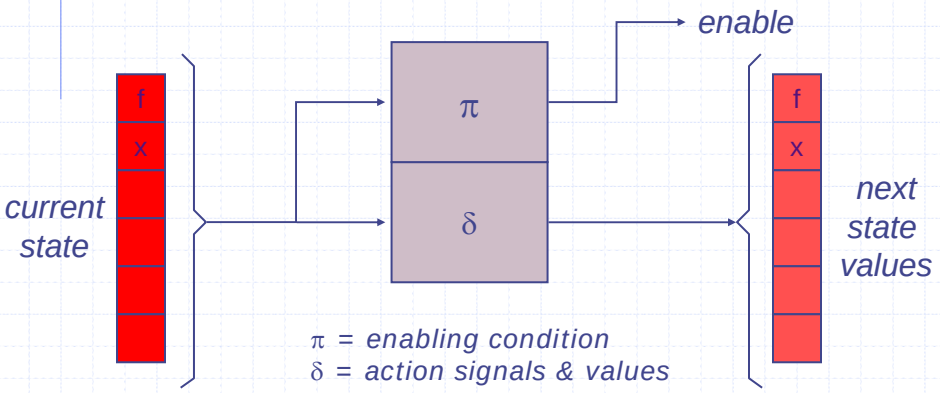
$$s_{next} = \textit{if } \pi(s) \textit{ then } \delta(s) \textit{ else } s$$

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

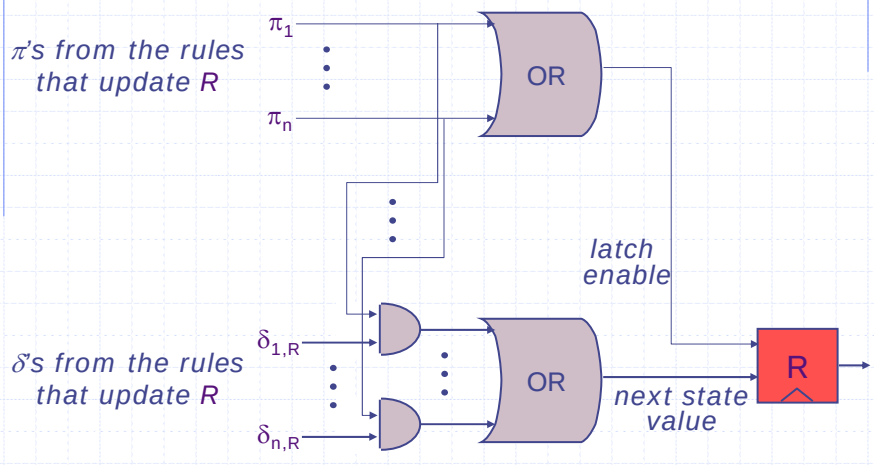
$\delta(s)$ is the "state transformation" function, i.e., computes the next-state value in terms of the current state values.

Compiling a Rule

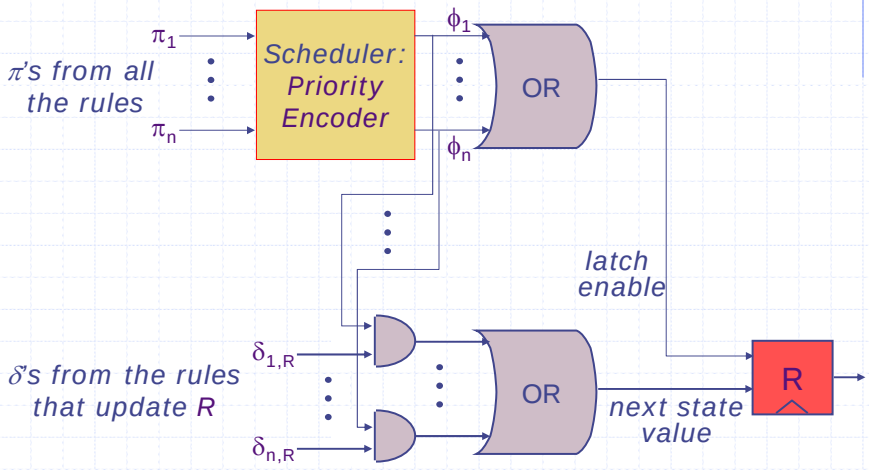
```
rule r (f.first() > 0) ;
        x <= x + 1 ;    f.deq ();
endrule
```



Combining State Updates: *strawman*

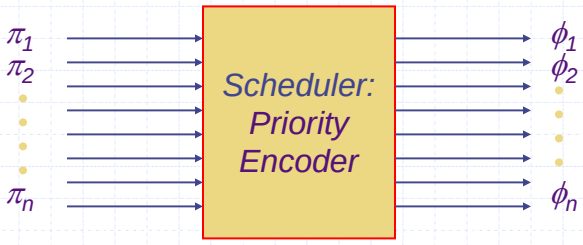


Combining State Updates



Scheduler ensures that at most one ϕ_i is true

One-rule-at-a-time Scheduler



1. $\phi_i \Rightarrow \pi_i$
2. $\pi_1 \vee \pi_2 \vee .... \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee .... \vee \phi_n$
3. One rewrite at a time
   i.e. at most one ϕ_i is true

Very conservative way of guaranteeing correctness

Executing Multiple Rules Per Cycle:
Conflict-free rules

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

rule rb (z > 20);
   y <= y + 2;
endrule
```

Parallel execution behaves like ra < rb = rb < ra

$Rule_a$ and $Rule_b$ are conflict-free if

$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow$ 1. $\pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s))$

2. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$

Parallel Execution can also be understood in terms of a composite rule

```
rule ra_rb((z>10)&&(z>20));
   x <= x+1; y <= y+2;
endrule
```


Executing Multiple Rules Per Cycle:
Sequentially Composable rules

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

rule rb (z > 20);
   y <= y + 2;
endrule
```

Parallel execution behaves like ra < rb

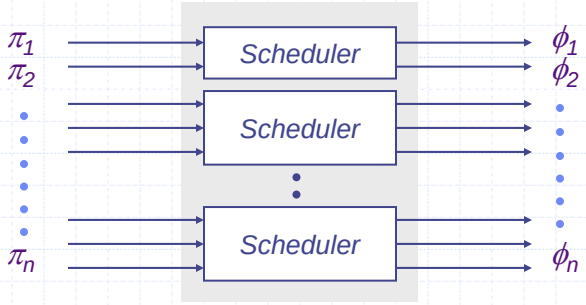
$Rule_a$ and $Rule_b$ are sequentially composable if

$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow \pi_b(\delta_a(s))$

Parallel Execution can also be understood in terms of a composite rule

```
rule ra_rb((z>10)&&(z>20));
   x <= y+1; y <= y+2;
endrule
```

Multiple-Rules-per-Cycle Scheduler



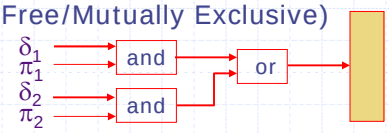
Divide the rules into smallest conflicting groups; provide a scheduler for each group

1. $\phi_i \Rightarrow \pi_i$
2. $\pi_1 \vee \pi_2 \vee .... \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee .... \vee \phi_n$
3. *Multiple operations such that*
   $\phi_i \wedge \phi_j \Rightarrow$ *R_i and R_j are conflict-free or sequentially composable*

Muxing structure

- Muxing logic requires determining for each register (action method) the rules that update it and under what conditions

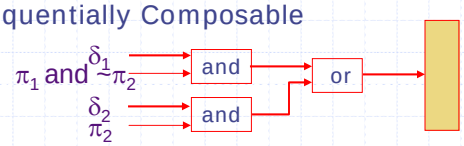
Conflict Free/Mutually Exclusive)



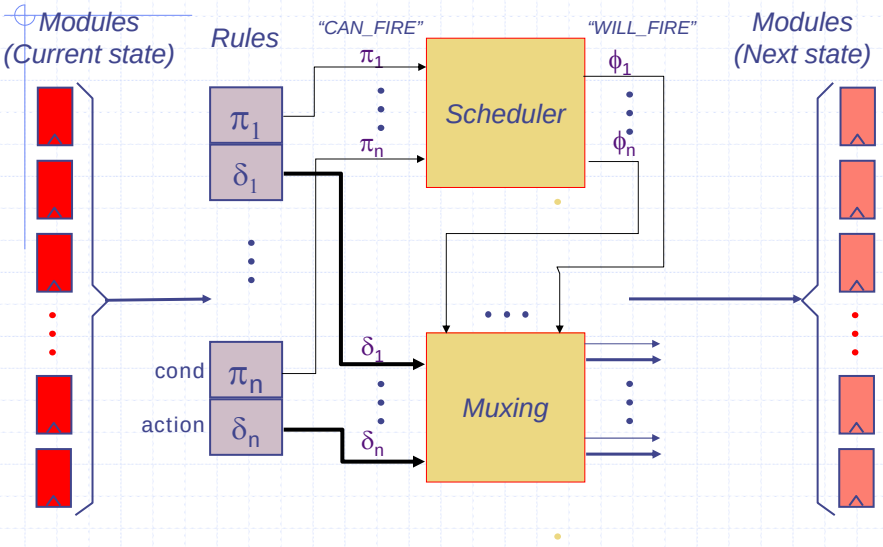
CF rules either do not update the same element or are ME

$\pi_1 \rightarrow \sim\pi_2$

Sequentially Composable



Scheduling and control logic



Extra's

Sequentially Composable rules ...

```
rule ra (z > 10);
   x <= 1;
endrule

rule rb (z > 20);
   x <= 2;
endrule
```

Parallel execution can behave either like ra < rb or rb < ra but the two behaviors are not the same

Composite rules

Behavior ra < rb

```
rule ra_rb(z>10 && z>20);
   x <= 2;
endrule
```

Behavior rb < ra

```
rule rb_ra(z>10 && z>20);
   x <= 1;
endrule
```


Mutually Exclusive Rules

- $Rule_a$ and $Rule_b$ are mutually exclusive if they can never be enabled simultaneously

$$\forall s \; . \; \pi_a(s) \Rightarrow \sim \pi_b(s)$$

Mutually-exclusive rules are Conflict-free even if they write the same state

Mutual-exclusive analysis brings down the cost of conflict-free analysis

Compiler determines if two rules can be executed in parallel

$Rule_a$ and $Rule_b$ are conflict-free if

$\forall s \; . \; \pi_a(s) \wedge \pi_b(s) \Rightarrow$

1. $\pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s))$
2. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$

$D(Ra) \cap R(Rb) = \phi$
$D(Rb) \cap R(Ra) = \phi$
$R(Ra) \cap R(Rb) = \phi$

$Rule_a$ and $Rule_b$ are sequentially composable if

$\forall s \; . \; \pi_a(s) \wedge \pi_b(s) \Rightarrow \pi_b(\delta_a(s))$

$D(\pi_b) \cap R(Ra) = \phi$

These properties can be determined by examining the domains and ranges of the rules in a pairwise manner.

These conditions are sufficient but not necessary.
Parallel execution of CF and SC rules does not increase the critical path delay

Homework problem

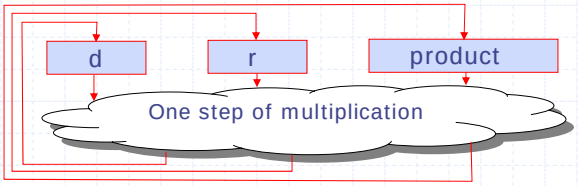
Binary Multiplication

Exercise: Binary Multiplier

Simple binary multiplication:

```
   1001   // d = 4'd9
 x 0101   // r  = 4'd5
 -------
   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
```

What does it look like in Bluespec?



A2

Multiplier in Bluespec

```
module mkMult (I_mult);
  Reg#(Int#(32)) product <- mkReg(0);
  Reg#(Int#(32)) d       <- mkReg(0);
  Reg#(Int#(16)) r       <- mkReg(0);

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

  method Action start (Int#(16)x,Int#(16)y) if (r == 0);
     d <= signExtend(x); r <= y;
  endmethod

  method Int#(32) result () if (r == 0);
     return product;
  endmethod
endmodule
```

What is the interface I_mult ?

A2 Fix the code

Arvind, 4/28/2007