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
GCD Hardware Module

The module can easily be made polymorphic

Many different implementations can provide the same interface:

```markdown
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
```

Does it compute faster?
Bluespec Tool flow

Bluespec SystemVerilog source
Bluespec Compiler
Blueview
C
Verilog 95 RTL
Verilog sim
RTL synthesis
Bluesim
Cycle
Accurate
VCD output
Debussy
Visualization
Gates
Legend
files
Bluepec tools
3rd party tools

Bluespec Compiler

Generated Verilog RTL: GCD

```verilog
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

\[
x_{\text{en}} = \text{swap}?
\]
\[
y_{\text{en}} = \text{swap? OR subtract?}
\]
\[
\text{rdy} =
\]

Generated Hardware Module

\[
x_{\text{en}} = \text{swap}?
\]
\[
y_{\text{en}} = \text{swap? OR subtract?}
\]
\[
\text{rdy} =
\]
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 \( \mu \text{m}^2 \)

- Unrolled (16 bits)
  - Clock Period: 1.65 ns
  - Area: 5944 \( \mu \text{m}^2 \)

- 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

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

\( \pi(s) \) is the condition (predicate) of the rule, a.k.a. the “CAN_FIRE” signal of the rule. (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

\[ \pi = \text{enabling condition} \]
\[ \delta = \text{action signals & values} \]

Combining State Updates:

strawman

\[ \pi_1, \pi_n \rightarrow \text{OR} \]
\[ \delta_1, \delta_n \rightarrow \text{OR} \]

next state value
latch enable
Combining State Updates

π's from all the rules

δ's from the rules that update R

Scheduler ensures that at most one $\phi_i$ is true

One-rule-at-a-time Scheduler

1. $\phi_i \Rightarrow \pi_i$

2. $\pi_1 \lor \pi_2 \lor \ldots \lor \pi_n \Rightarrow \phi_1 \lor \phi_2 \lor \ldots \lor \phi_n$

3. One rewrite at a time
   i.e. at most one $\phi_i$ is true
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 ra_rb((z>10)&&(z>20));
 x <= x+1; y <= y+2;
endrule
```

Rule \( a \) and Rule \( b \) are conflict-free if

\[
\forall s . \pi_a(s) \land \pi_b(s) \Rightarrow \\
1. \pi_b(\delta_a(s)) \land \pi_b(\delta_a(s)) \\
2. \delta_a(\delta_b(s)) == \delta_b(\delta_a(s))
\]

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

Rule \( a \) and Rule \( b \) are sequentially composable if

\[
\forall s . \pi_a(s) \land \pi_b(s) \Rightarrow \pi_b(\delta_a(s))
\]

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

1. $\phi_i \Rightarrow \pi_i$
2. $\pi_1 \lor \pi_2 \lor \ldots \lor \pi_n \Rightarrow \phi_1 \lor \phi_2 \lor \ldots \lor \phi_n$
3. Multiple operations such that $\phi_i \land \phi_j \Rightarrow R_i$ and $R_i$ 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

CF rules either do not update the same element or are ME
Scheduling and control logic

Modules (Current state) → Rules

π₁, δ₁

πₙ, δₙ

cond, action

"CAN_FIRE" → "WILL_FIRE"

Scheduler

Muxing

Modules (Next state)

π₁, δ₁

πₙ, δₙ

Extra’s
Sequentially Composable rules ...

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

\[ \text{Composite rules} \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>z &gt; 10</th>
<th>x &lt;= 1</th>
<th>endrule</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rb</td>
<td>z &gt; 20</td>
<td>x &lt;= 2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>z &gt; 10 &amp;&amp; z &gt; 20</th>
<th>x &lt;= 2</th>
<th>endrule</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra_rb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rb_ra</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
\[
\forall s . \pi_a(s) \land \pi_b(s) \Rightarrow
\begin{align*}
1. & \quad \pi_a(\delta_b(s)) \land \pi_b(\delta_a(s)) \\
2. & \quad \delta_a(\delta_b(s)) = \delta_b(\delta_a(s))
\end{align*}
\]

Rule \( a \) and Rule \( b \) are sequentially composable if
\[
\forall s . \pi_a(s) \land \pi_b(s) \Rightarrow \pi_a(\delta_b(s))
\]

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:

\[ \begin{array}{c}
  \times & 1001 & 0101 \\
  & 1001 & 0000 \\
  & 0101 & 10101 \\
\end{array} \]

\( // d = 4'd9 \)
\( // r = 4'd5 \)
\( // d << 0 \) (since \( r[0] == 1 \))
\( // 0 << 1 \) (since \( r[1] == 0 \))
\( // d << 2 \) (since \( r[2] == 1 \))
\( // 0 << 3 \) (since \( r[3] == 0 \))
\( // \text{product (sum of above)} = 45 \)

What does it look like in Bluespec?

Multiplier in Bluespec

\[
\text{module mkMult (I\_mult);} \]
\[
\text{Reg#}(\text{Int}#(32)) \text{ product} \leftarrow \text{mkReg}(0); \\
\text{Reg#}(\text{Int}#(32)) \text{ d} \leftarrow \text{mkReg}(0); \\
\text{Reg#}(\text{Int}#(16)) \text{ r} \leftarrow \text{mkReg}(0); \\
\text{endmodule} \\
\]

\text{rule} \text{ cycle (r != 0);}
\text{ if } (r[0] == 1) \text{ product} \leftarrow \text{product + d;}
\text{ d} \leftarrow \text{d} << 1;
\text{ r} \leftarrow \text{r} >> 1;
\text{endrule} \\

\text{method Action} \text{ start (Int}(#16)x,\text{Int}(#16)y) \text{ if (r == 0);} \\
\text{ d} \leftarrow \text{signExtend}(x); \text{ r} \leftarrow \text{y;}
\text{endmethod} \\

\text{method Int}(32) \text{ result () \text{ if (r == 0);}
\text{ return product;}
\text{endmethod} \\
\text{endmodule} \\
\]

What is the interface \text{I\_mult}?
A2

Fix the code
Arvind, 4/28/2007