Introduction to Bluespec: A new methodology for designing Hardware

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

What is needed to make hardware design easier

- Extreme IP reuse
  - 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
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.

Bluespec promotes composition through guarded interfaces

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

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

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 \(\rightarrow\) action

Rules can manipulate state in other modules only via their interfaces.
GCD: A simple example to explain hardware generation from Bluespec

Programming with rules: A simple example

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

15 6
GCD in BSV

```haskell
module mkGCD (I_GCD);
  Reg#(Int#(32)) x <- mkRegU;
  Reg#(Int#(32)) 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#(32) a, Int#(32) b)
    if (y==0);
      x <= a; y <= b;
  endmethod

  method Int#(32) result()
    if (y==0);
      return x;
  endmethod

endmodule
```

GCD Hardware Module

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

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

BlueSpec Tool flow

Bluespec System Verilog source
Bluespec Compiler
C

Verilog 95 RTL
Bluesim

Verilog sim
RTL synthesis

VCD output
Debussy Visualization

Power estimation tool

FPGA
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 ;
...```

February 8, 2010 L02-13

Generated Hardware

```
x
y
en
rdy
start

x_en
y_en

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

sub

swap? subtract?
x

result

x_en =
y_en =

February 8, 2010 L02-14
```
### Generated Hardware Module

- **Inputs:**
  - `x`, `y`, `x_en`, `y_en`
  - `result`, `sub` (output)
  - `start`, `rdy`

- **Behavior:**
  - `x_en = swap?`
  - `y_en = swap? OR subtract?`
  - `rdy =` (further details not provided)

---

### GCD: A Simple Test Bench

```verilog
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 8, 2010 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

GCD: Synthesis results

- Original (16 bits)
  - Clock Period: 1.6 ns
  - Area: 4240 µm²
- Unrolled (16 bits)
  - Clock Period: 1.65ns
  - Area: 5944 µm²
- Unrolled takes 31% fewer cycles on the testbench
Need for a rule scheduler

Example 1

```plaintext
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

\begin{verbatim}
rule ra (z > 10);
  x <= y + 1;
endrule

rule rb (z > 20);
  y <= x + 1;
endrule
\end{verbatim}

Can these rules be enabled together?

Can they be executed concurrently?

Example 3

\begin{verbatim}
rule ra (z > 10);
  x <= y + z;
endrule

rule rb (z > 20);
  y <= y + z;
endrule
\end{verbatim}

Can these rules be enabled together?

Can they be executed concurrently?
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

$$s_{\text{next}} = \begin{cases} \delta(s) & \text{if } \pi(s) \\ s & \text{else} \end{cases}$$

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

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

Combining State Updates:

**strawman**

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

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

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
The plan

- Express combinational circuits in Bluespec
- Express synchronous pipelines
  - single-rule systems; no scheduling issues
- Multiple rule systems and concurrency issues
  - Eliminating dead cycles
- Asynchronous pipelines and processors

Each idea would be illustrated via examples

No discussion of Bluespec syntax in the lectures; you are suppose to learn that by yourself and in tutorials