Bluespec-1: Design Affects Everything

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology
Chip costs are exploding because of design complexity

SoC failures costing time/spins

Source: Aart de Geus, CEO of Synopsys
Based on a survey of 2000 users by Synopsys

Design and verification dominate escalating project costs

Source: IBM/IBS, Inc.

Issues Found on First Spin ICs/ASICs

Source: IBM/IBS, Inc.
Common quotes

“Design is not a problem; design is easy”

“Verification is a problem”

“Timing closure is a problem”

“Physical design is a problem”

Mind set

Almost complete reliance on post-design verification for quality
Through the early 1980s:

The U.S. auto industry

- Sought quality solely through post-build inspection
- Planned for defects and rework

and U.S. quality was...
... less than world class

Adding quality inspectors ("verification engineers") and giving them better tools, was not the solution.

The Japanese auto industry showed the way:
- "Zero defect" manufacturing
New mind set:

Design affects everything!

A good design methodology
- Can keep up with changing specs
- Permits architectural exploration
- Facilitates verification and debugging
- Eases changes for timing closure
- Eases changes for physical design
- Promotes reuse

⇒ It is essential to

Design for Correctness
New semantics for expressing behavior to reduce design complexity

Decentralize complexity: *Rule-based specifications (Guarded Atomic Actions)*
- Let us think about one rule at a time

Formalize composition: *Modules with guarded interfaces*
- Automatically manage and ensure the correctness of connectivity, i.e., correct-by-construction methodology
- Retain resilience to changes in design or layout, e.g. compute latency $\Delta$’s
- Promote regularity of layout at macro level

Bluespec
RTL has poor semantics for composition

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 operation when the FIFO is full. A simultaneous push and pop operation causes the FIFO to be empty, since there is no pop data to prefetch. However, the FIFO is not full.

A pop operation on pop_req_n is asserted (LOW), as long as the FIFO is not empty. 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.

No machine verification of such informal constraints is feasible.

These constraints are spread over many pages of the documentation...
Bluespec promotes composition through guarded interfaces

theModuleA

theFifo.enq(value1);

theFifo.deq();

value2 = theFifo.first();

theModuleB

theFifo.enq(value3);

theFifo.deq();

value4 = theFifo.first();

Self-documenting interfaces;
Automatic generation of logic to eliminate conflicts in use.

FIFO

-- Enqueue arbitration control

-- Dequeue arbitration control

February 24, 2006
In Bluespec SystemVerilog (BSV) ...

- Power to express complex static structures and constraints
  - Checked by the compiler
- "Micro-protocols" are managed by the compiler
  - The compiler generates the necessary hardware (muxing and control)
  - Micro-protocols need less or no verification
- Easier to make changes while preserving correctness

Smaller, simpler, clearer, more correct code
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 $\rightarrow$ action

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

- GCD
- Multiplication
- IP Lookup
Programming with rules: A simple example

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

15 6
9 6 subtract
GCD in BSV

```plaintext
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
```

Assumes x /= 0 and y /= 0
GCD Hardware Module

The module can easily be made polymorphic.

Many different implementations can provide the same interface:

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

module mkGCD (I_GCD)
GCD: Another implementation

```verilog
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

Bluespec SystemVerilog source

Bluespec Compiler

C

Verilog 95 RTL

Bluespec C sim

Verilog sim

RTL synthesis

VCD output

Debussy Visualization

gates

Legend

files
Bluespec tools
3rd party tools

http://csg.csail.mit.edu/6.375/
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
Generated Hardware Module

x
y
en
rdy

start

x_en

y_en

start_en

x

y

start_en

x

y

swap?
subtract?

(!=0)

sub

>
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

Why do we need the state variable?
GCD: Test Bench

```verilog
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; state <= 0; end
        else c1 <= c1+1;
        if (c2 == 63) state <= 2;
    endrule

endmodule
```
GCD: Synthesis results

- **Original (16 bits)**
  - Clock Period: 1.6 ns
  - Area: 4240.10 mm²

- **Unrolled (16 bits)**
  - Clock Period: 1.65 ns
  - Area: 5944.29 mm²

- Unrolled takes 31% fewer cycles on testbench
Multiplier Example

Simple binary multiplication:

\[ \begin{array}{c}
\times & 1001 & // \quad d = 4'd9 \\
0101 & // \quad r = 4'd5 \\
\hline
1001 & // \quad d \ll 0 \text{ (since } r[0] == 1) \\
0000 & // \quad 0 \ll 1 \text{ (since } r[1] == 0) \\
1001 & // \quad d \ll 2 \text{ (since } r[2] == 1) \\
0000 & // \quad 0 \ll 3 \text{ (since } r[3] == 0) \\
0101101 & // \quad \text{product (sum of above)} = 45 \\
\end{array} \]

What does it look like in Bluespec?

One step of multiplication:
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

        endrule

    method Action start

        endmethod

    method Int#(32) result ()

        endmethod
endmodule
```

What is the interface I_mult?
Exploring microarchitectures

IP Lookup Module
A packet is routed based on the “Longest Prefix Match” (LPM) of its IP address with entries in a routing table.

Line rate and the order of arrival must be maintained.
Sparse tree representation

<table>
<thead>
<tr>
<th>IP address</th>
<th>Result</th>
<th>M Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.14.<em>.</em></td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>7.14.7.3</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>10.18.200.*</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>10.18.200.5</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>5.<em>.</em>.*</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

Real-world lookup algorithms are more complex but all make a sequence of dependent memory references.
SW ("C") version of LPM

```c
int lpm (IPA ipa)                         /* 3 memory lookups */
{
  int p;

  p = RAM [ipa[31:16]];       /*  Level 1: 16 bits */
  if (isLeaf(p)) return p;

  p = RAM [p + ipa [15:8]];  /*  Level 2: 8 bits */
  if (isLeaf(p)) return p;

  p = RAM [p + ipa [7:0]];    /*  Level 3: 8 bits */
  return p; /* must be a leaf */
}
```

How to implement LPM in HW?
Not obvious from C code!
Longest Prefix Match for IP lookup: 3 possible implementation architectures

Rigid pipeline
- Inefficient memory usage but simple design

Linear pipeline
- Efficient memory usage through memory port replicator

Circular pipeline
- Efficient memory with most complex control

Designer’s Ranking:
1. Rigid pipeline
2. Linear pipeline
3. Circular pipeline

Which is "best"?
Static Pipeline
rule static (True);

    if (canInsert(c5))
        begin
            c1 <= 0; r1 <= in.first(); in.deq();
        end
    else
        begin
            r1 <= r5; c1 <= c5;
        end

    if (notEmpty(r1)) makeMemReq(r1);
    r2 <= r1; c2 <= c1;
    r3 <= r2; c3 <= c2;
    r4 <= r3; c4 <= c3;
    r5 <= getMemResp(); c5 <= (c4 == n-1) ? 0 : n;
    if (c5 == n) out.enq(r5);

endrule
Circular pipeline

LuReq \rightarrow \text{enter?} \rightarrow \text{getToken} \rightarrow \text{RAM} \rightarrow \text{done?} \rightarrow \text{cbuf} \rightarrow \text{LuResp}

\text{in} \rightarrow \text{active} \rightarrow \text{no}

\text{yes}
Circular Pipeline code

```
rule enter (True);
    t <- cbuf.newToken();
    IP ip = in.first(); ram.req(ip[31:16]);
    active.enq(tuple2(ip[15:0], t)); in.deq();
endrule

rule done (True);
    p <- ram.resp();
    match {.rip, .t} = active.first();
    if (isLeaf(p)) cbuf.complete(t, p);
    else begin
        match {.newreq, .newrip} = remainder(p, rip);
        active.enq(rip << 8, t);
        ram.req(p+signExtend(rip[15:7]));
    end
    active.deq();
endrule
```

February 24, 2006

http://csg.csail.mit.edu/6.375/
## Synthesis results

<table>
<thead>
<tr>
<th>LPM versions</th>
<th>Code size (lines)</th>
<th>Best Area (gates)</th>
<th>Best Speed (ns)</th>
<th>Mem. util. (random workload)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static V</td>
<td>220</td>
<td>2271</td>
<td>3.56</td>
<td>63.5%</td>
</tr>
<tr>
<td>Linear V</td>
<td>410</td>
<td>14759</td>
<td>4.7</td>
<td>99.9%</td>
</tr>
<tr>
<td>Linear BSV</td>
<td>168</td>
<td>15910 (8% larger)</td>
<td>4.7 (same)</td>
<td>99.9%</td>
</tr>
<tr>
<td>Circular V</td>
<td>364</td>
<td>8103</td>
<td>3.62</td>
<td>99.9%</td>
</tr>
<tr>
<td>Circular BSV</td>
<td>257</td>
<td>8170 (1% larger)</td>
<td>3.67 (2% slower)</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

**V** = Verilog

**BSV** = Bluespec System Verilog

Bluespec and Verilog synthesis results are nearly identical

---

Synthesized to TSMC 0.18 µm library

Arvind, Nikhil, Rosenband & Dave ICCAD 2004