

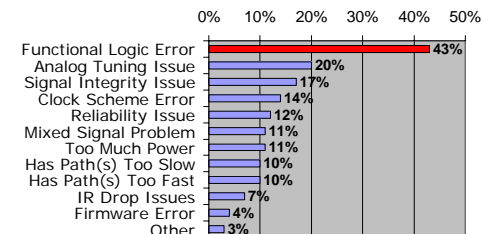
Bluespec-1: Design Affects Everything

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

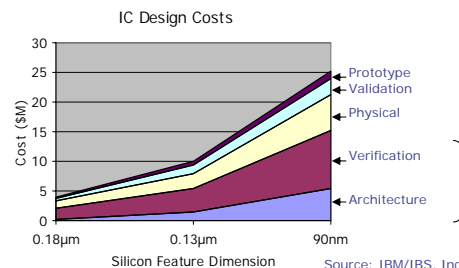
Chip costs are exploding because of design complexity

SoC failures costing time/spins

Issues Found on First Spin ICs/ASICs



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



Design and verification dominate escalating project costs

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

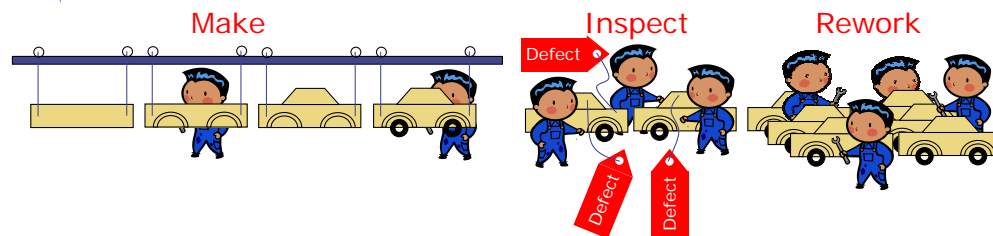
Almost complete reliance on post-design verification for quality

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

February 24, 2006

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

L07-5

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

February 24, 2006

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

L07-6

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 Δ 's
 - Promote regularity of layout at macro level

Bluespec

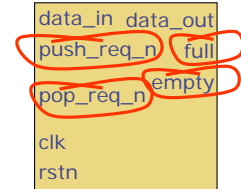
February 24, 2006

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

L07-7

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 simple push and pop operation when the FIFO is full. A simultaneous push and pop operation is possible when the FIFO is empty, since there is no pop data to prefetch. However, a push operation is not allowed when the FIFO is empty.

A pop operation is allowed when pop_req_n is asserted (LOW), as long as the FIFO is not empty. The pop_req_n signal 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.

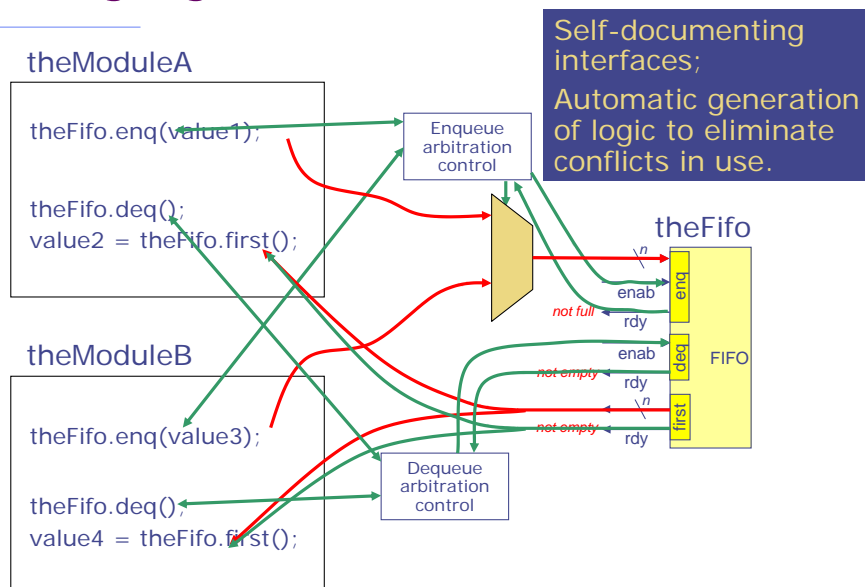
These constraints are spread over many pages of the documentation...

February 24, 2006

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

L07-8

Bluespec promotes composition through guarded interfaces



February 24, 2006

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

L07-9

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

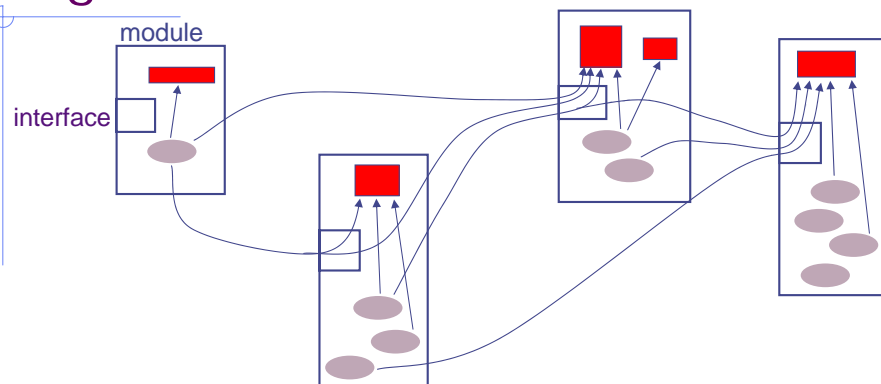
→ *Smaller, simpler, clearer, more correct code*

February 24, 2006

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

L07-10

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 → action

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

February 24, 2006

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

L07-11

Examples

- ◆ GCD
- ◆ Multiplication
- ◆ IP Lookup

February 24, 2006

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

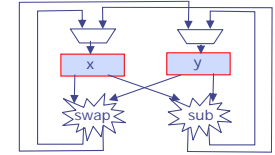
L07-12

Programming with rules: A simple example

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

15	6	
9	6	<i>subtract</i>

GCD in BSV



```

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

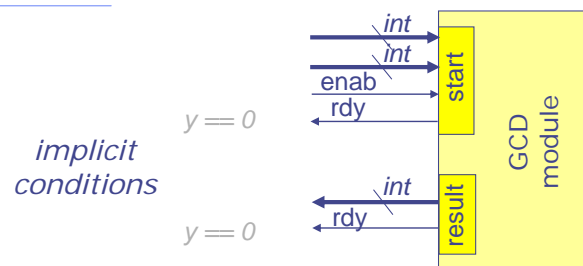
  rule swap ((x > y) && (y != 0));
    x <= y; y <= x;
  endrule
  rule subtract ((x <= y) && (y != 0));
    y <= y - x;
  endrule

  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

```

Assumes x != 0 and y != 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

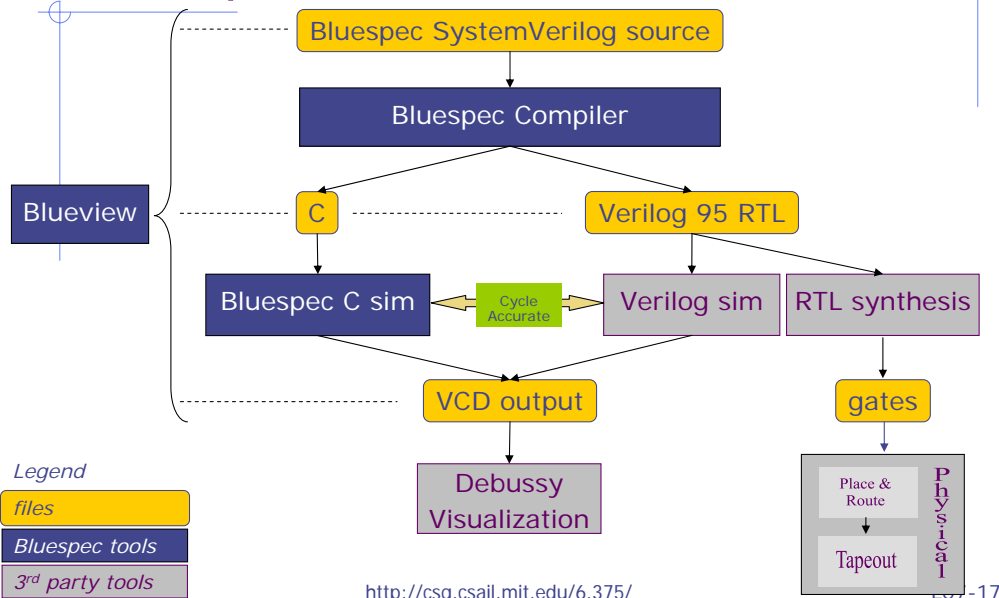
  method Action start(int a, int b) if (y==0);
    x <= a; y <= b;
  endmethod
  method int result() if (y==0);
    return x;
  endmethod
endmodule

```

Combine swap and subtract rule

Does it compute faster ?

Bluespec Tool flow

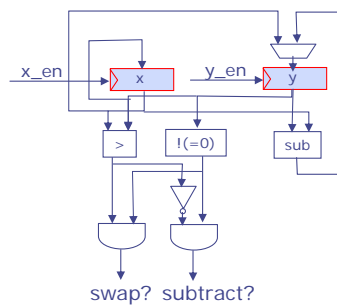


Generated Verilog RTL: gcd

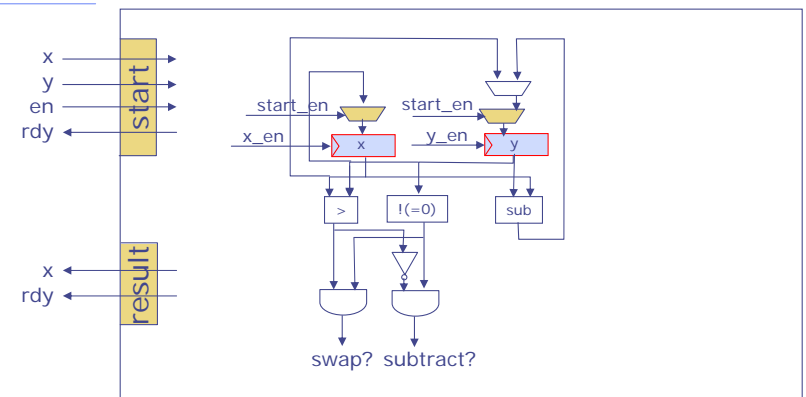
```

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



Generated Hardware Module



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?

February 24, 2006

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

L07-21

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

February 24, 2006

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

L07-22

GCD: Synthesis results

- ◆ Original (16 bits)
 - Clock Period: 1.6 ns
 - Area: 4240.10 mm²
- ◆ Unrolled (16 bits)
 - Clock Period: 1.65ns
 - Area: 5944.29 mm²
- ◆ Unrolled takes 31% fewer cycles on testbench

February 24, 2006

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

L07-23

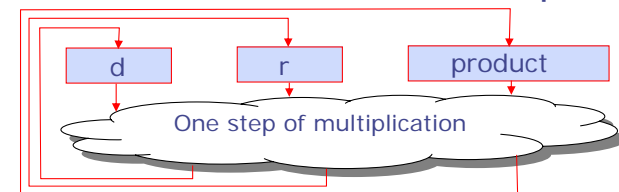
Multiplier Example

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?



February 24, 2006

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

L07-24

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 ()

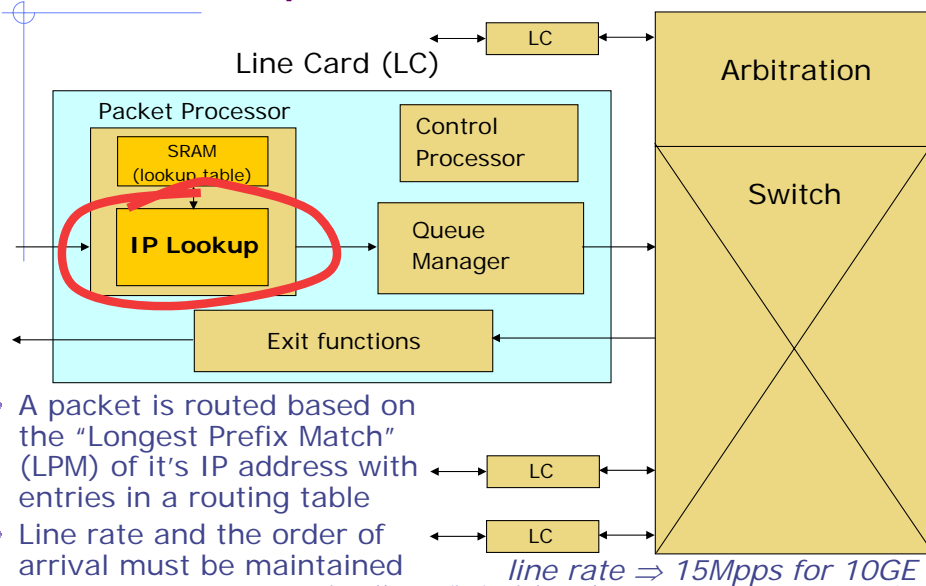
  endmethod
endmodule
  
```

What is the interface I_mult ?

Exploring microarchitectures

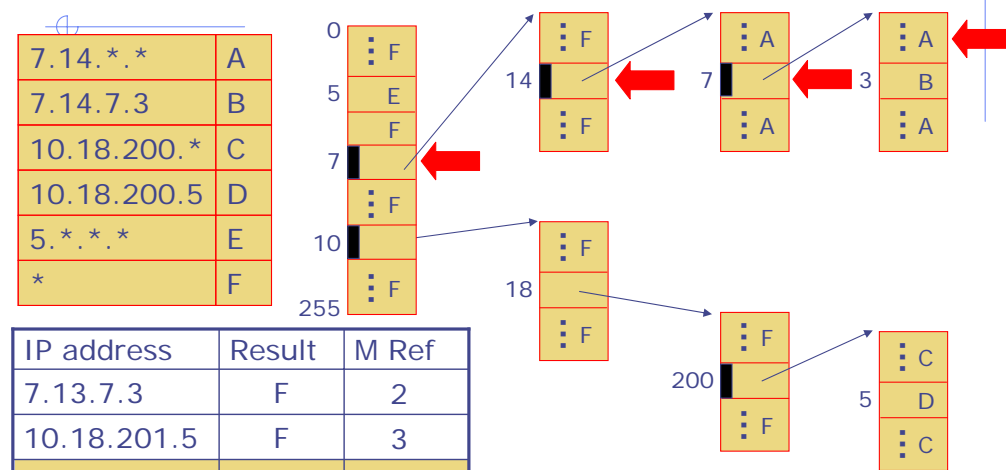
IP Lookup Module

IP Lookup block in a router



- ◆ A packet is routed based on the "Longest Prefix Match" (LPM) of it's IP address with entries in a routing table
- ◆ Line rate and the order of arrival must be maintained

Sparse tree representation



Real-world lookup algorithms are more complex but all make a sequence of dependent memory references.

SW ("C") version of LPM

```

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

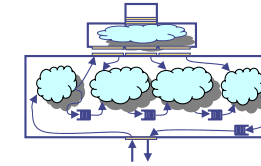


Inefficient memory usage but **simple** design

Designer's Ranking:

1

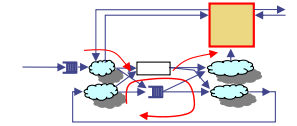
Linear pipeline



Efficient memory usage through memory port replicator

2

Circular pipeline

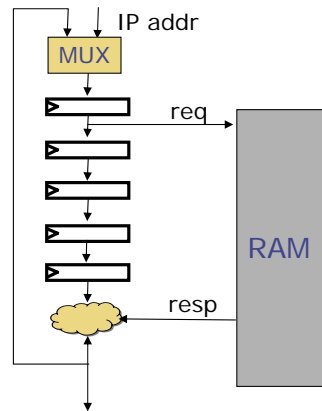


Efficient memory with most **complex** control

3

Which is "best"?

Static Pipeline



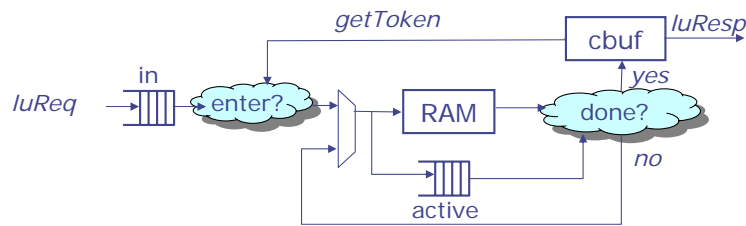
Static code

```

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



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

```

Synthesis results

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

V = Verilog
 BSV = Bluespec System Verilog

Synthesized to TSMC 0.18 μm library

Bluespec and Verilog synthesis results are nearly identical

Next Time

◆ Combinational Circuits and Types