Bluespec-1: Design methods to facilitate rapid growth of SoCs

System-on-a-chip

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

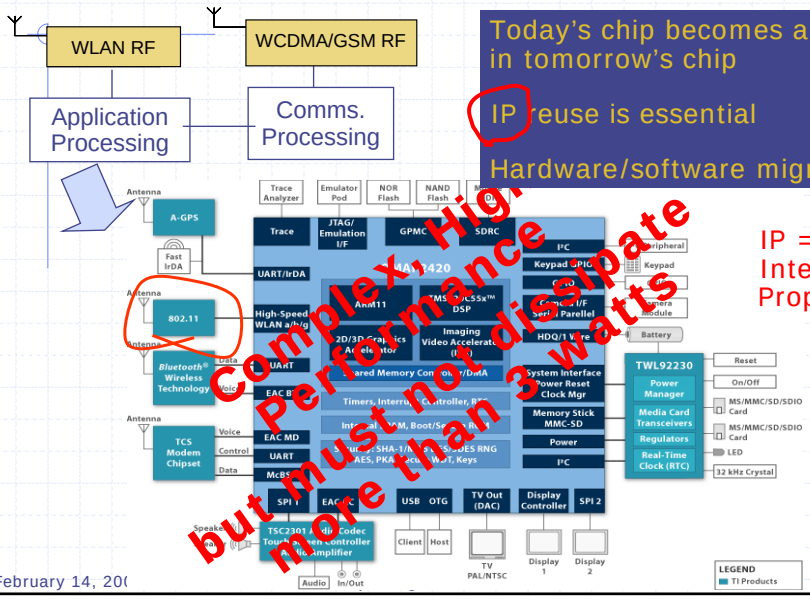
The biggest SoC drivers

- Explosive growth in markets for
  - cell phones
  - game boxes
  - sensors and actuators

Functionality and applications are constrained primarily by:

- cost
- power/energy constrains

Current Cellphone Architecture



An under appreciated fact

- If a functionality (e.g. H.264) is moved from a programmable device to a specialized hardware block, the power/energy savings are 100 to 1000 fold

Power savings ⇒ more specialized hardware

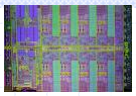
but our mind set

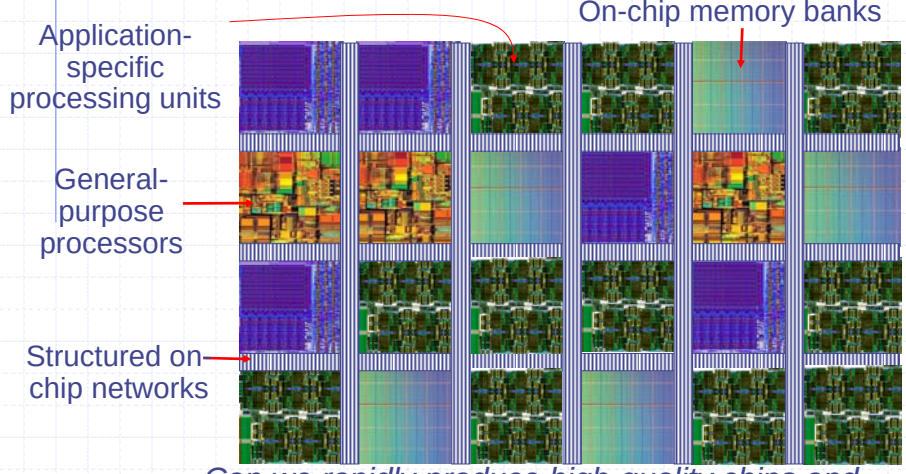
- Software is forgiving
- Hardware design is difficult, inflexible, brittle, error prone, …

SoC Trajectory:

multicores, heterogeneous, regular, …

IBM Cell Processor





Can we rapidly produce high-quality chips and surrounding systems and software?

Things to remember

- Design costs (hardware & software) dominate
- Within these costs verification and validation costs dominate
- IP reuse is essential to prevent design-team sizes from exploding

design cost = number of engineers x time to design

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

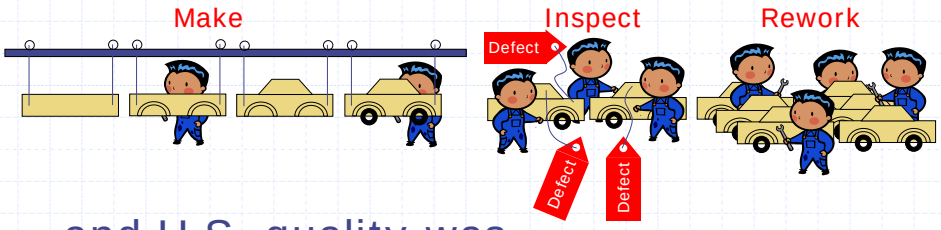
Almost complete reliance on post-design verification for quality

Through the early 1980s:

American Motors CHRYSLER Ford GM

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 ways of expressing behavior to reduce design complexity

- Decentralize complexity: *Rule-based specifications (Guarded Atomic Actions)*
  - Lets you think one *rule* at a time

`Strong flavor of Unity`

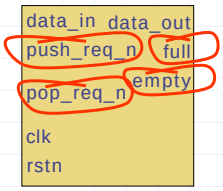
- Formalize composition: *Modules with guarded interfaces*
  - Automatically manage and ensure the correctness of connectivity, i.e., correct-by-construction methodology

Bluespec

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

Reusing IP Blocks

Example: Commercially available FIFO IP block



An error occurs if a push is attempted while the FIF[illegible]l.

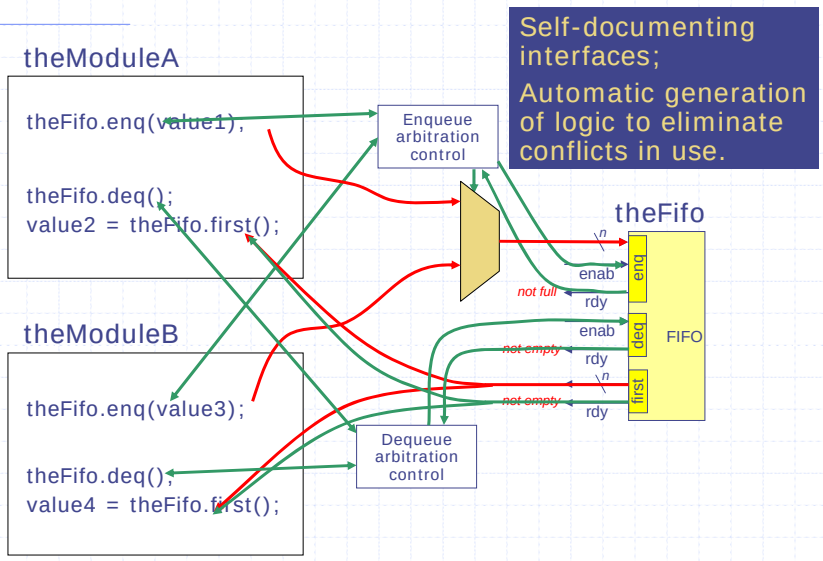
Thus, there is no conflict in a simul[illegible]when the FIFO is full. A simultaneous push and pop [illegible]FIFO is empty, since there is no pop data to prefetch. How[illegible]red in the FIFO.

A pop [illegible] `pop_req_n` is asserted (LOW), as long as the FIFO is not em[illegible] `eq_n` causes the internal read pointer to be incremented on the nex[illegible] of `clk`. Thus, the RAM read data must be captured on the `clk` following the a[illegible]on 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



Bluespec

5-minute break to stretch you legs

- What is it?
- Programming with Rules
  - Example GCD
- Synthesis of circuits
- Another Example: Multiplication

Bluespec is available in two versions:
BSV – Bluespec in System Verilog
ESEPro – Bluespec in SystemC

These lectures will use BSV syntax

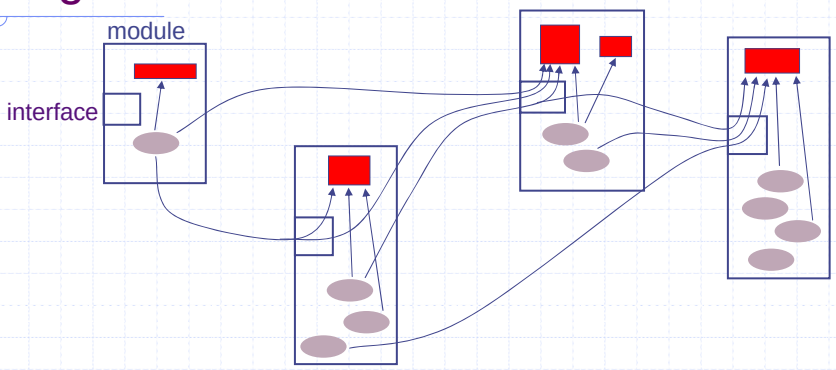
Bluespec SystemVerilog (BSV)

- Power to express complex static structures and constraints
  - Checked by the compiler
- "Micro-protocols" are managed by the compiler
  - The necessary hardware for muxing and control is generated automatically and is correct by construction
- Easier to make changes while preserving correctness

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

➔ *not just simulation, synthesis as well*

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

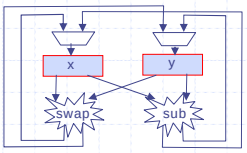
Rules can manipulate state in other modules only *via* their interfaces.

Programming with rules: A simple example

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

15		6	
9		6	*subtract*
3		6	*subtract*
6		3	*swap*
3		3	*subtract*
0	*answer:*	3	*subtract*

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
```

State

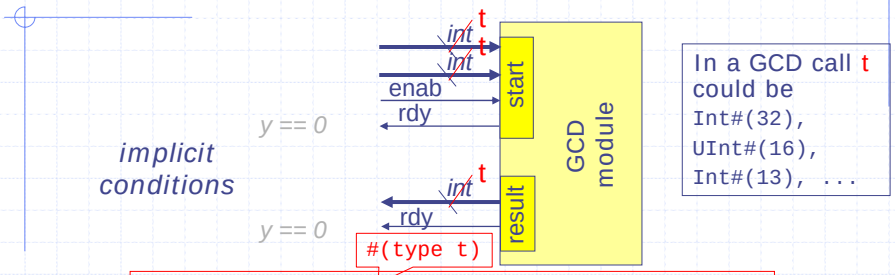
`typedef int Int#(32)`

Internal behavior

External interface

Assumes x /= 0 and y /= 0

GCD Hardware Module



In a GCD call t could be
`Int#(32),`
`UInt#(16),`
`Int#(13), ...`

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

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

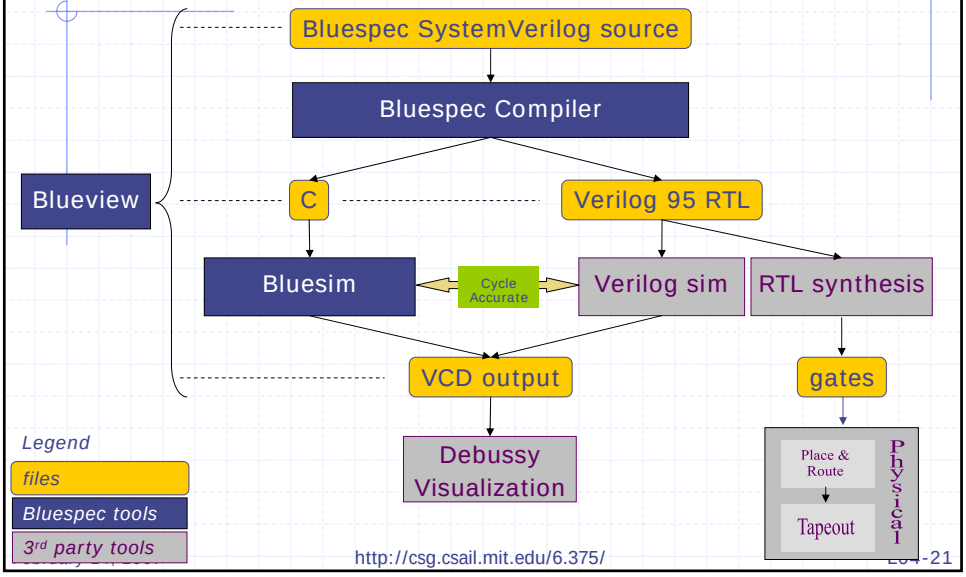
GCD: Another implementation

```
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 ?

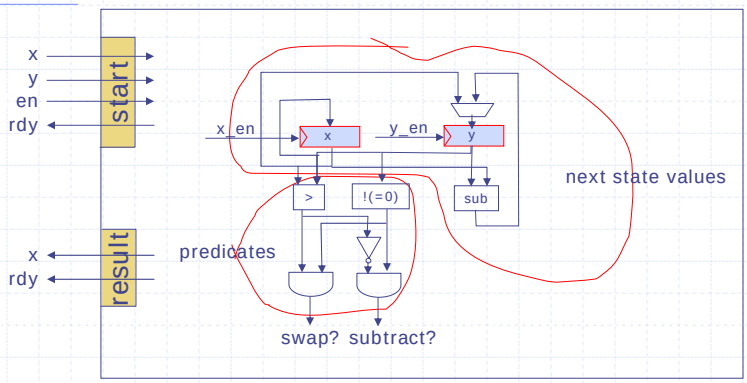
Bluespec Tool flow



Generated Verilog RTL: GCD

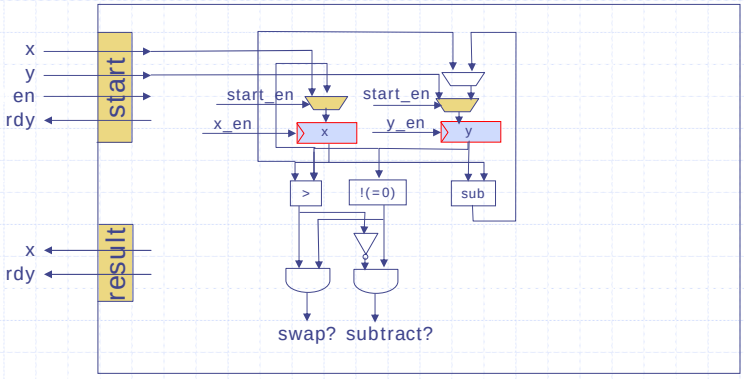
```
module mkGCD(CLK,RST_N,start_a,start_b,EN_start,RDY_start,
             result,RDY_result);
  input  CLK; input  RST_N;
// action method start
  input [31 : 0] start_a; input [31 : 0] start_b; input EN_start;
  output RDY_start;
// value method result
  output [31 : 0] result; output RDY_result;
// 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 = swap?
y_en = swap? OR subtract?

Generated Hardware Module



x_en = swap? OR start_en
y_en = swap? OR subtract? OR start_en

rdy = (y==0)

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

```
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
```

Feeds all pairs (c1,c2)
1 < c1 < 7
1 < c2 < 15
to GCD

GCD: Synthesis results

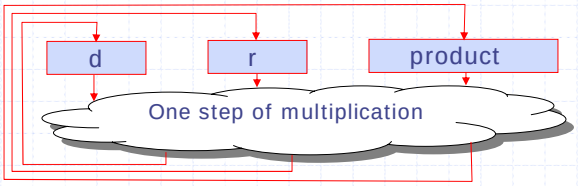
- Original (16 bits)
  - Clock Period: 1.6 ns
  - Area: 4240 μm^2
- Unrolled (16 bits)
  - Clock Period: 1.65ns
  - Area: 5944 μm^2
- Unrolled takes 31% fewer cycles on the testbench

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?



Multiplier in Bluespec

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

Summary

- Market forces are demanding a much greater variety of SoCs
- The design cost for SoCs has to be brought down dramatically by facilitating IP reuse
- High-level synthesis tools are essential for architectural exploration and IP development
- Bluespec is both high-level and synthesizable

Next time: Combinational Circuits and Simple pipelines