# A Proposal for a More Generic, More Accountable, Verilog

Cherif Salama Walid Taha

DCC 2010

# Technology Gap

Verilog/VHDL progress
vs.
manufacturing technology progress

# Technology Gap

Verilog/VHDL progress
vs.
manufacturing technology progress

# There is Still Hope

- Functional HDLs:
  - HML, Hawk, Lava, Chalk
- System Modeling and High Level Synthesis HDLs:
  - Bluespec, SystemC
- Programming Languages Advances
  - Static analysis and abstract interpretation
  - Generative programming
  - Indexed types

# Our Approach

- Start with Verilog
- Identify its limitations
- Eliminate/Alleviate them one by one
  - Retain backward compatibility

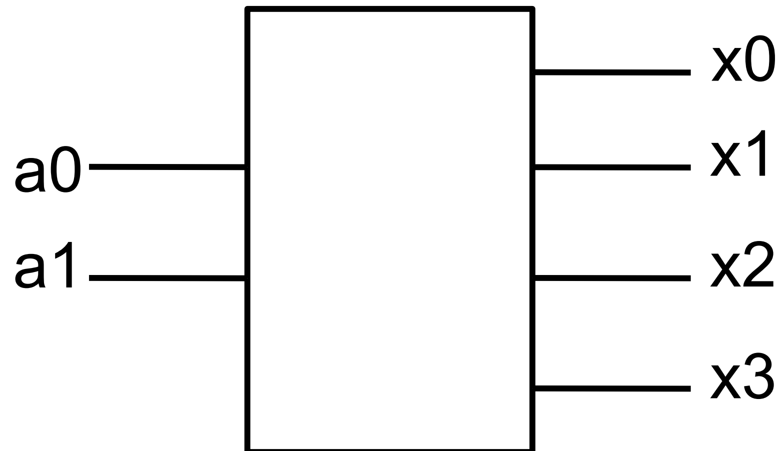
# Our Goal

- Create a better Verilog
  - **Generic:** Reusable Parameterized Libraries (circuit families)
  - **Accountable:** Statically guaranteed to have desirable properties
    - Synthesizable
    - Resource bounded
    - Free from as much classes of errors as possible

# This Talk

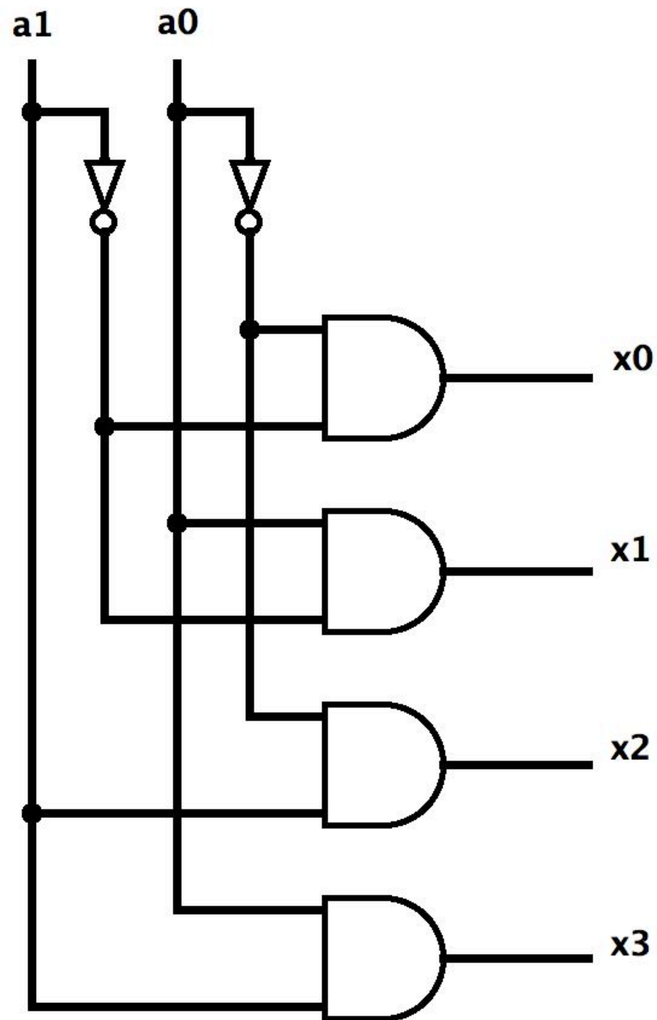
- Existing Verilog
- Previous Work: More Accountable
- Proposed Extensions
  - Motivation
  - Features
  - Still More Accountable?

# Decoder Example



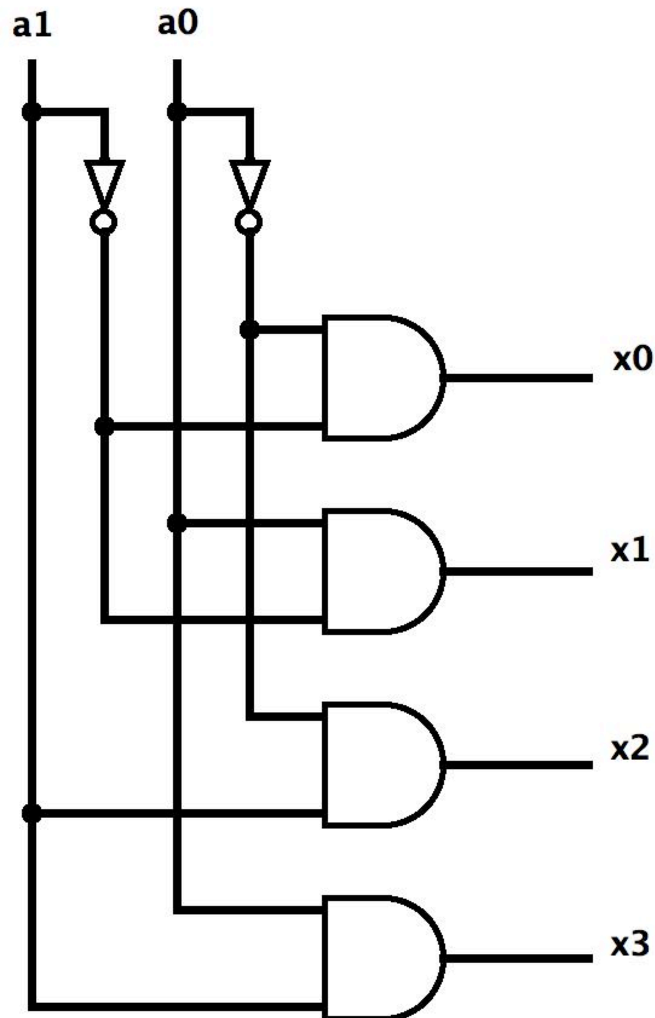

| a1 | a0 | x3 | x2 | x1 | x0 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

# Decoder Example



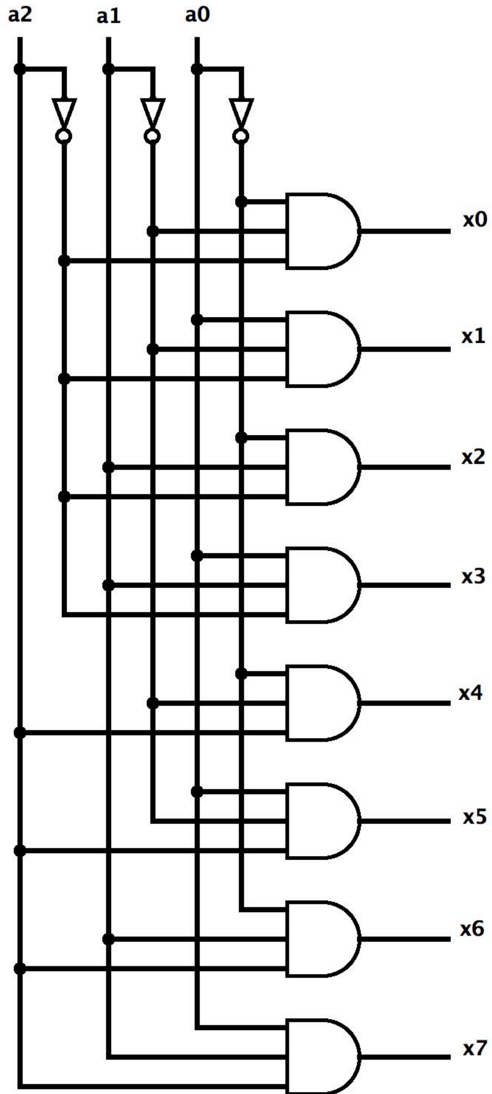

| a1 | a0 | x3 | x2 | x1 | x0 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

# Decoder Example



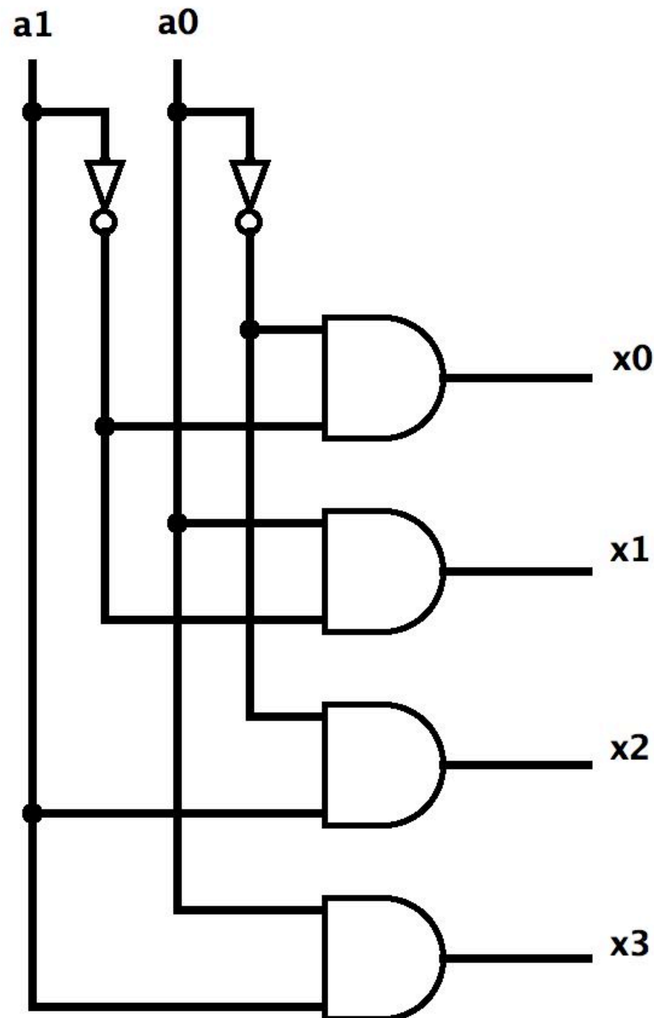

```
module decoder(x,a);
   input  [1 : 0] a;
   output [3 : 0] x;
   wire   [1 : 0] na;

   not (na[0], a[0]);
   not (na[1], a[1]);
   and (x[0], na[0], na[1]);
   and (x[1],  a[0], na[1]);
   and (x[2], na[0],  a[1]);
   and (x[3],  a[0],  a[1]);
endmodule
```

# Decoder Example




```
module decoder(x,a);
   input  [2 : 0] a;
   output [7 : 0] x;
   wire   [2 : 0] na;

   not (na[0], a[0]);
   not (na[1], a[1]);
   not (na[2], a[2]);
   and (x[0], na[0], na[1], na[2]);
   and (x[1],  a[0], na[1], na[2]);
   and (x[2], na[0],  a[1], na[2]);
   and (x[3],  a[0],  a[1], na[2]);
   and (x[4], na[0], na[1],  a[2]);
   and (x[5],  a[0], na[1],  a[2]);
   and (x[6], na[0],  a[1],  a[2]);
   and (x[7],  a[0],  a[1],  a[2]);
endmodule
```

# Generative Decoder




```
module decoder(x,a);
   parameter            N=4;
   input [N-1:0]        a;
   output [2**N-1:0] x;
   wire [N-1:0]         na;
   wire [N-1:0]         t [2**N-1:0];
   genvar               i,j;

   for(i=0;i<N;i=i+1)
     not (na[i],a[i]);

   for(i=0;i<2**N;i=i+1) begin
      for(j=0;j<N;j=j+1) begin
         if ((i>>j) % 2==0)
           assign t[i][j] = na[j];
         else
           assign t[i][j] = a[j];
      end
      assign x[i] = &t[i];
   end
endmodule
```

# Decoder Elaboration

```
module decoder(x,a);
   parameter          N=4;
   input [N-1:0]      a;
   output [2**N-1:0]  x;
   wire [N-1:0]       na;
   wire [N-1:0]       t [2**N-1:0];
   genvar             i,j;

   for(i=0;i<N;i=i+1)
     not (na[i],a[i]);

   for(i=0;i<2**N;i=i+1) begin
      for(j=0;j<N;j=j+1) begin
          if ((i>>j) % 2==0)
            assign t[i][j] = na[j];
          else
            assign t[i][j] = a[j];
      end
      assign x[i] = &t[i];
   end
endmodule
```

Elaboration
N=2

# Decoder Elaboration

```
module decoder(x,a);
   parameter          N=4;
   input [N-1:0]      a;
   output [2**N-1:0] x;
   wire [N-1:0]       na;
   wire [N-1:0]       t [2**N-1:0];
   genvar             i,j;

   for(i=0;i<N;i=i+1)
     not (na[i],a[i]);

   for(i=0;i<2**N;i=i+1) begin
      for(j=0;j<N;j=j+1) begin
          if ((i>>j) % 2==0)
            assign t[i][j] = na[j];
          else
            assign t[i][j] = a[j];
      end
      assign x[i] = &t[i];
   end
endmodule
```

Elaboration
N=2

```
module  decoder_2(x,a);
   input  [1 : 0] a;
   output [3 : 0] x;
   wire   [1 : 0] na;
   wire   [1 : 0] t[3 : 0];

   not (na[0], a[0]);
   not (na[1], a[1]);

   assign t[0][0] = na[0];
   assign t[0][1] = na[1];
   assign x[0] = &t[0];
   assign t[1][0] = a[0];
   assign t[1][1] = na[1];
   assign x[1] = &t[1];
   assign t[2][0] = na[0];
   assign t[2][1] = a[1];
   assign x[2] = &t[2];
   assign t[3][0] = a[0];
   assign t[3][1] = a[1];
   assign x[3] = &t[3];
endmodule
```

# Generative Constructs

- Pros:
  - Full control over resulting circuit structure
  - Very suitable for describing circuit families
  - Works both for simulation and synthesis
- Cons:
  - No static guarantees about the properties of the generated circuit
  - Limited set of constructs

# This Talk

- Existing Verilog
- Previous Work: More Accountable
- Proposed Extensions
  - Motivation
  - Features
  - Still More Accountable?

# More Accountable

- Static Checks
  - Static = pre-elaboration
  - Why? Because static checks:
    - Save time
    - Are valid for all possible parameter values
    - Help with debugging and maintenance
    - Pinpoint errors in the (more concise) code that the designer had written

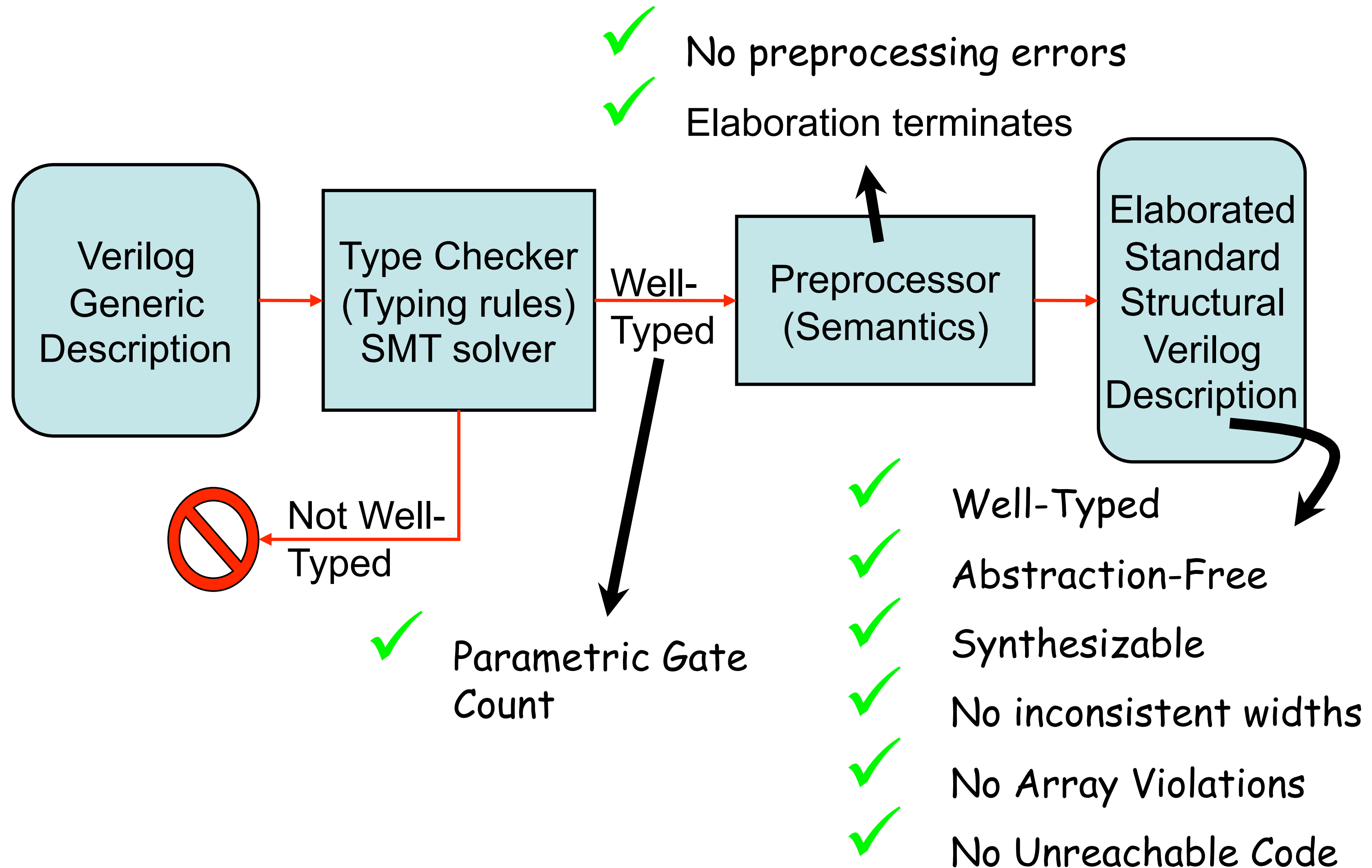
# Approach

- We use statically typed two-level languages (STTL)
  - Preprocessing is level 0 computation
  - The result after preprocessing is level 1 computation
- To do this we formally define:
  - Core syntax: Structural + Generative that we call Featherweight Verilog (FV)
  - Preprocessing semantics: formalize elaboration
  - Type system

# Implementation: VPP

- Preprocessor with built-in type checker
- Implemented in OCaml
- Uses Verilog syntax (not FV)
- Generates purely structural standard Verilog code
- Available for download from
  – http://www.cs.rice.edu/~cra1/Home/vpp.tar.gz

# VPP Static Guarantees



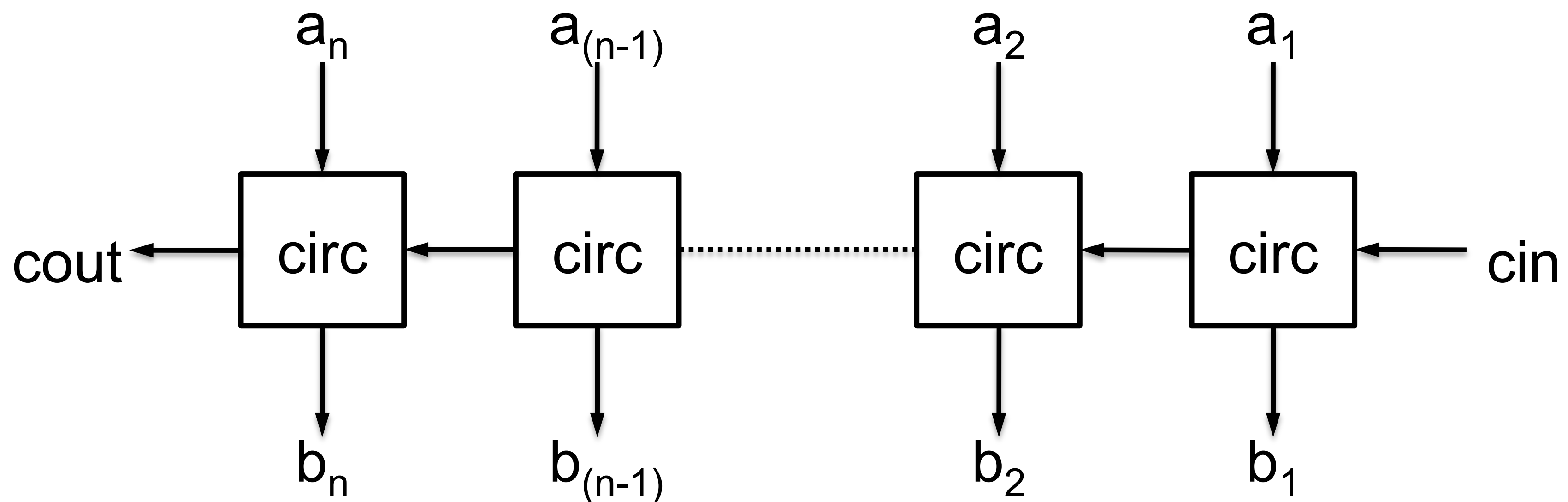
# This Talk

- Existing Verilog
- Previous Work: More Accountable
- Proposed Extensions
  - – Motivation
  - – Features
  - – Still More Accountable?

# Motivation: Limited Expressivity

- Existing generative constructs limited to
  - Parameterized modules (only integers)
  - Loops
  - Conditionals
- Do not work well to describe
  - Tree shaped circuits
  - Butterfly circuits
  - Connection patterns

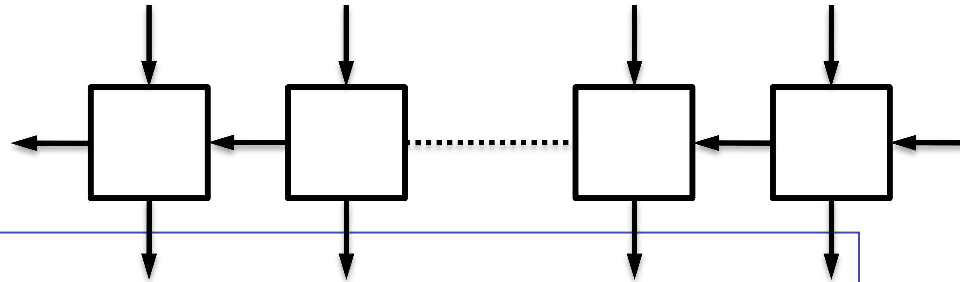
# Example




Lava:

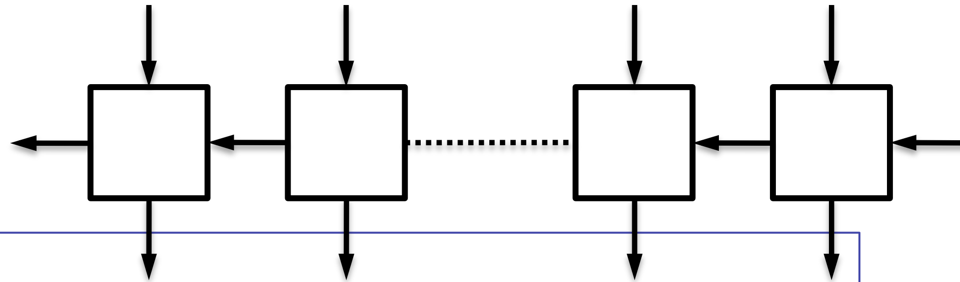
```
ripple circ (cin, []) = ([], cin)

ripple circ (cin, a:as) = (b:bs, cout)
  where
    (b, carry) = circ (cin, a)
    (bs, cout) = ripple circ (carry, as)
```

# Proposed Verilog Features



```
module ripple
 (output 't1 [N:1] b, output 't3 cout,
  input  't2 [N:1] a, input  't3 cin);

  parameter N = 4;
  parameter circ (output ’t1 b, output ’t3 co,
                  input  ’t2 a, input  ’t3 ci);

  't3 carry;

  if (N<=0)
    assign cout = cin;
  else begin
    circ c (b[1],carry,a[1],cin);
    ripple ##(circ) #(N-1) r
          (b[N:2],cout,a[N:2],carry);
  end
endmodule
```

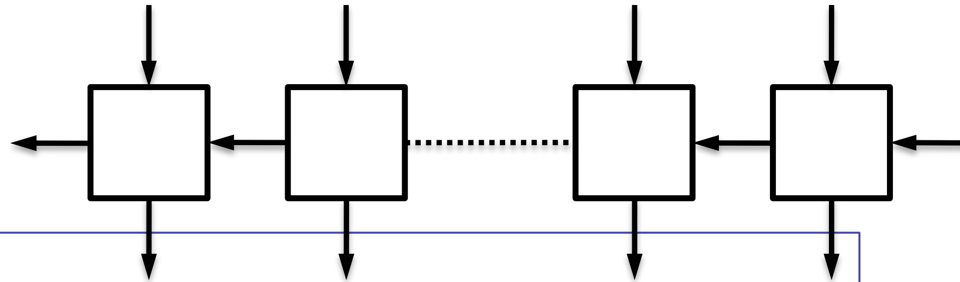
# Proposed Verilog Features



```
module ripple
 (output 't1 [N:1] b, output 't3 cout,
  input  't2 [N:1] a, input  't3 cin);

  parameter N = 4;
  parameter circ (output ’t1 b, output ’t3 co,
                  input  ’t2 a, input  ’t3 ci);

  't3 carry;

  if (N<=0)
    assign cout = cin;
  else begin
    circ c (b[1],carry,a[1],cin);
    ripple ##(circ) #(N-1) r
           (b[N:2],cout,a[N:2],carry);
  end
endmodule
```

- Recursive Modules

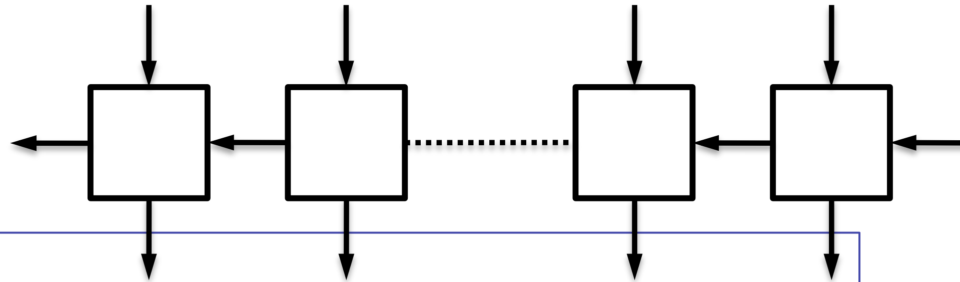
# Proposed Verilog Features



```
module ripple
 (output 't1 [N:1] b, output 't3 cout,
  input  't2 [N:1] a, input  't3 cin);

  parameter N = 4;
  parameter circ (output ’t1 b, output ’t3 co,
                  input  ’t2 a, input  ’t3 ci);

  't3 carry;

  if (N<=0)
    assign cout = cin;
  else begin
    circ c (b[1],carry,a[1],cin);
    ripple ##(circ) #(N-1) r
           (b[N:2],cout,a[N:2],carry);
  end
endmodule
```

- Recursive Modules
- Higher Order Modules

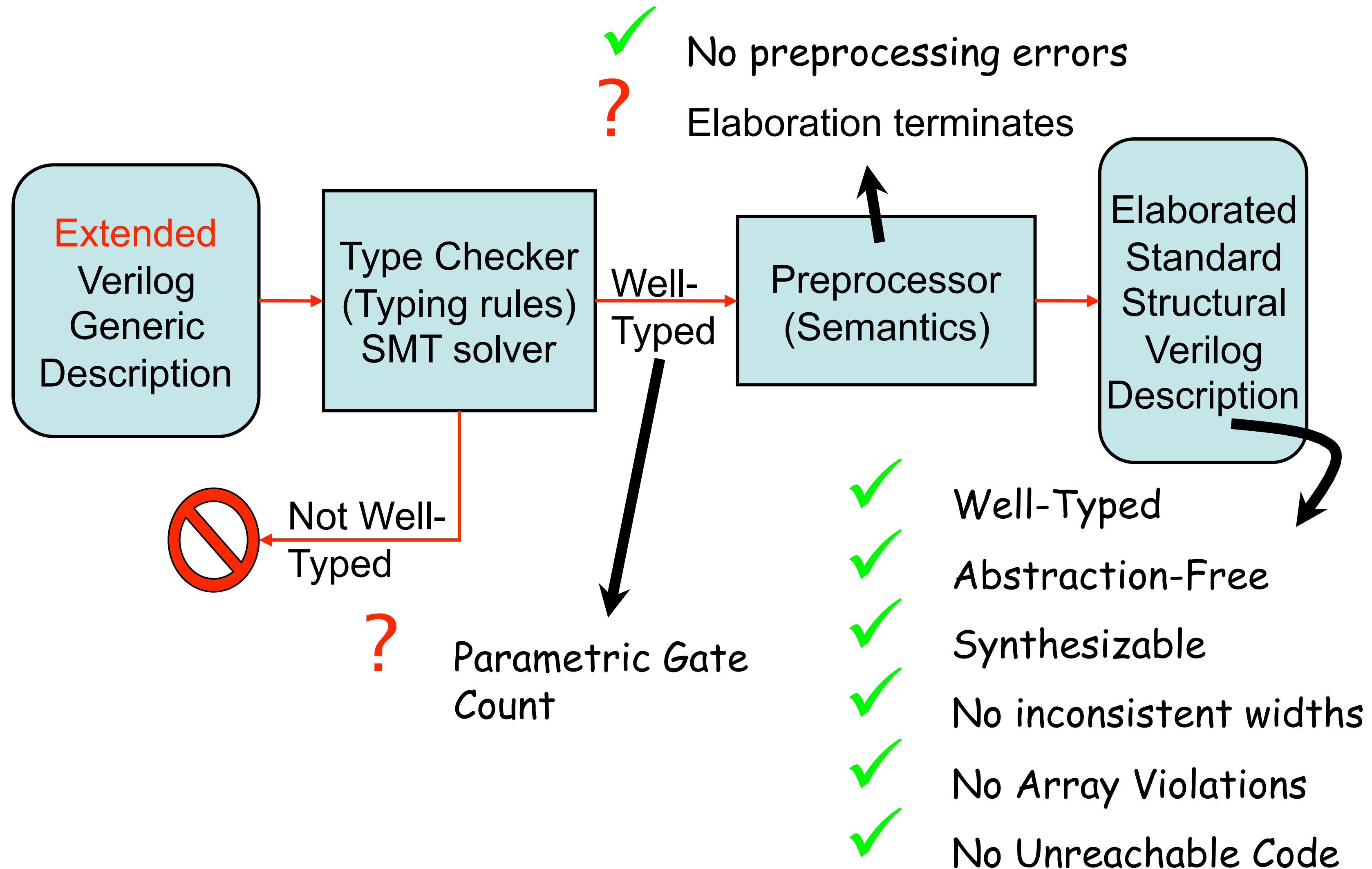
# Proposed Verilog Features



```
module ripple
  (output 't1 [N:1] b, output 't3 cout,
   input  't2 [N:1] a, input  't3 cin);

   parameter N = 4;
   parameter circ (output ’t1 b, output ’t3 co,
                   input  ’t2 a, input  ’t3 ci);

   't3 carry;

   if (N<=0)
     assign cout = cin;
   else begin
     circ c (b[1],carry,a[1],cin);
     ripple ##(circ) #(N-1) r
            (b[N:2],cout,a[N:2],carry);
   end
endmodule
```

- Recursive Modules
- Higher Order Modules
- Polymorphic types

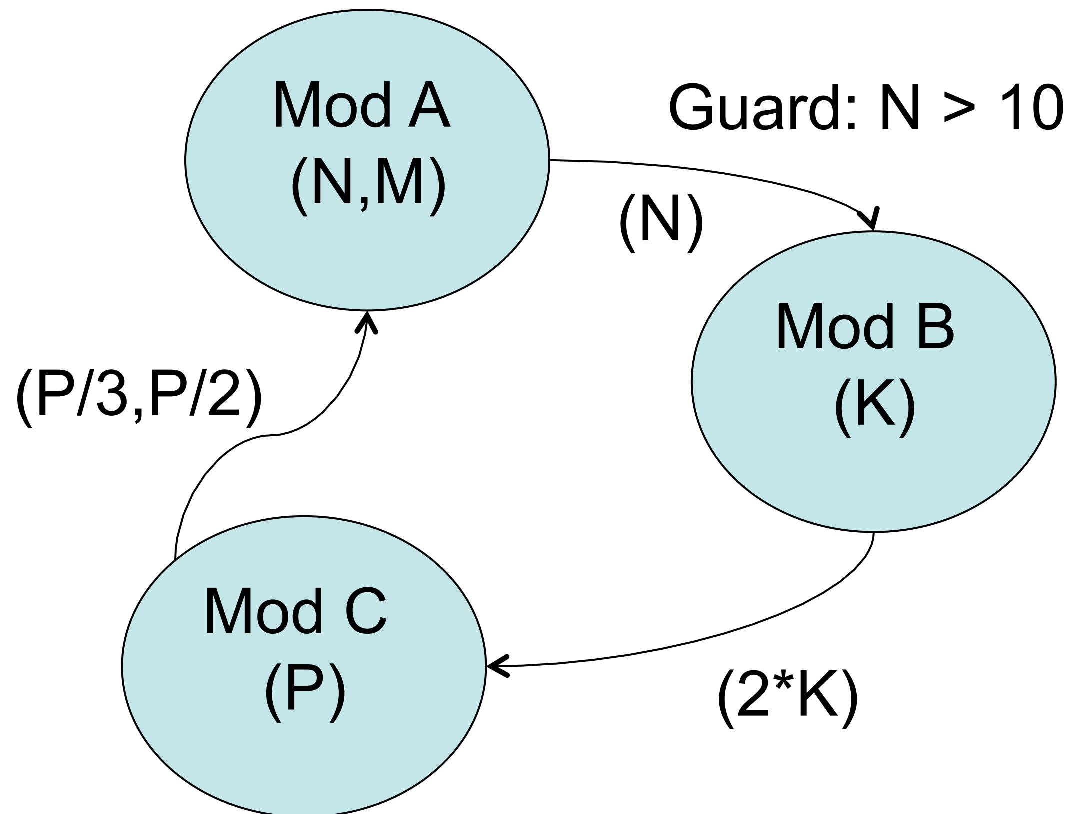
# VPP Static Guarantees



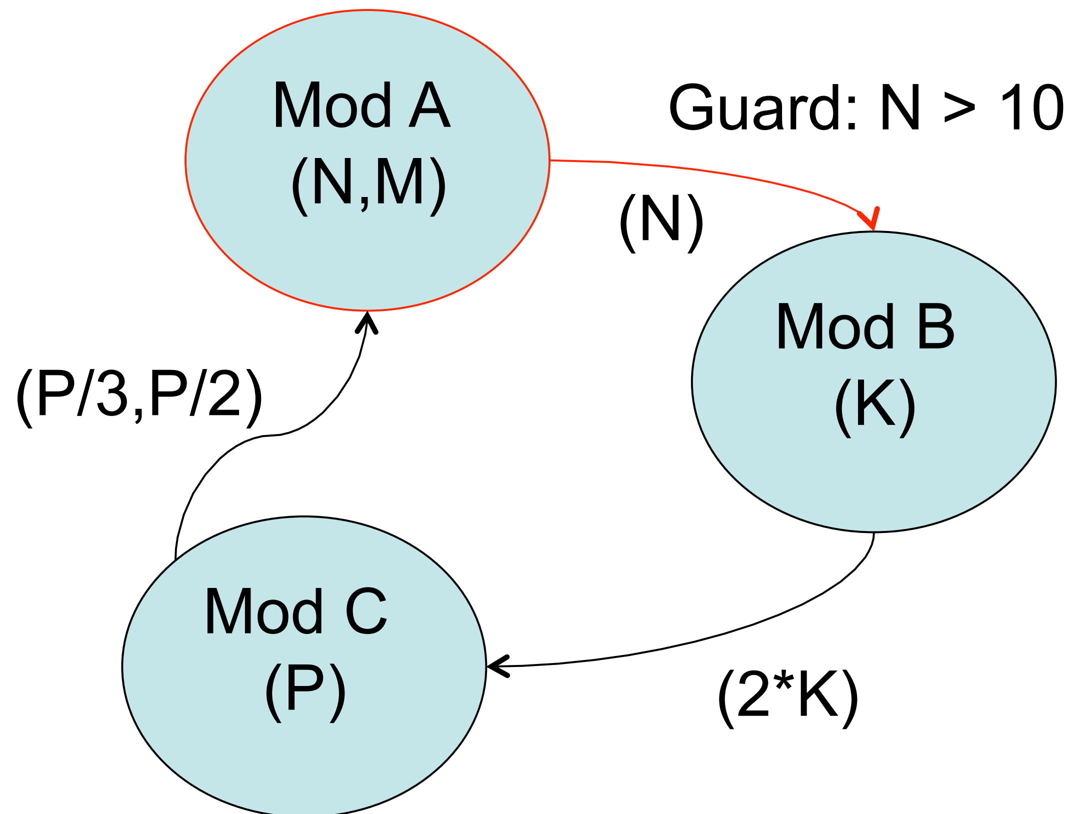
# Termination Analysis

- Potential sources of non-termination:
  - Loops
    - Restricted in form
    - proven using the SMT solver
  - Recursive Module Instantiations
    - Requires cycle detection in the directed graph of module dependencies
    - Each vertex is a module labeled with parameter names
    - Each Edge is an instantiation labeled with parameter values and instantiation guard

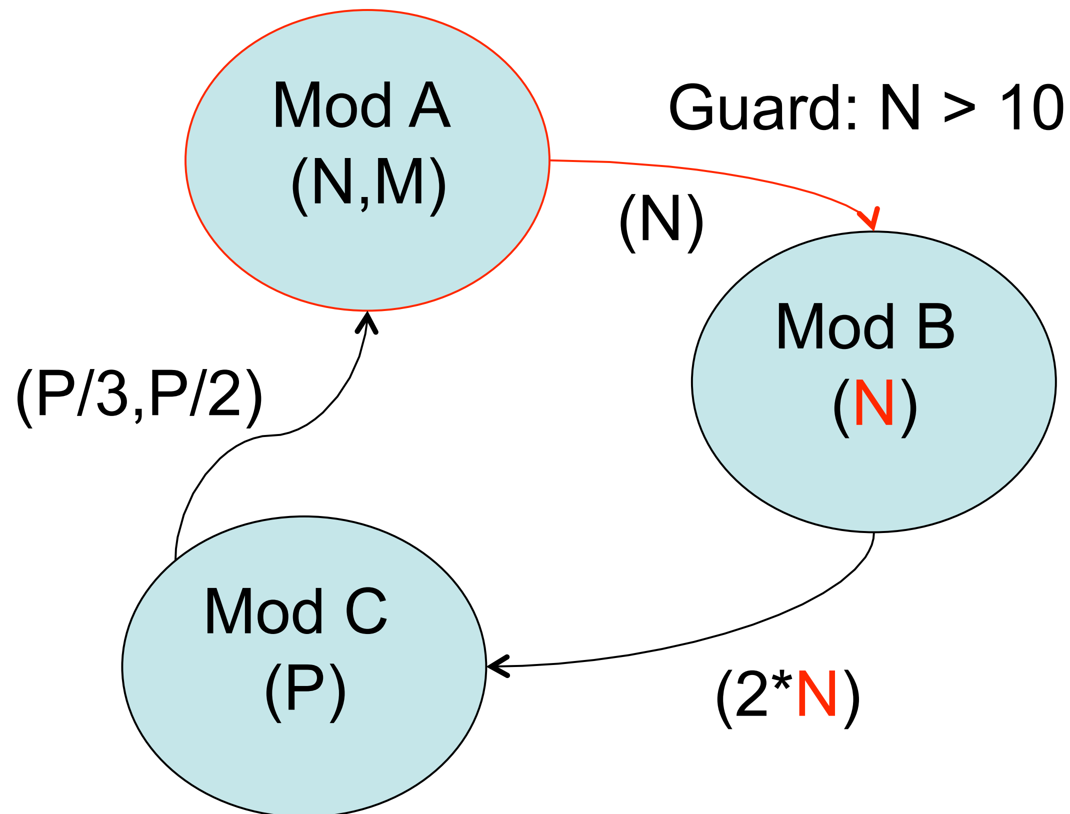
# Example



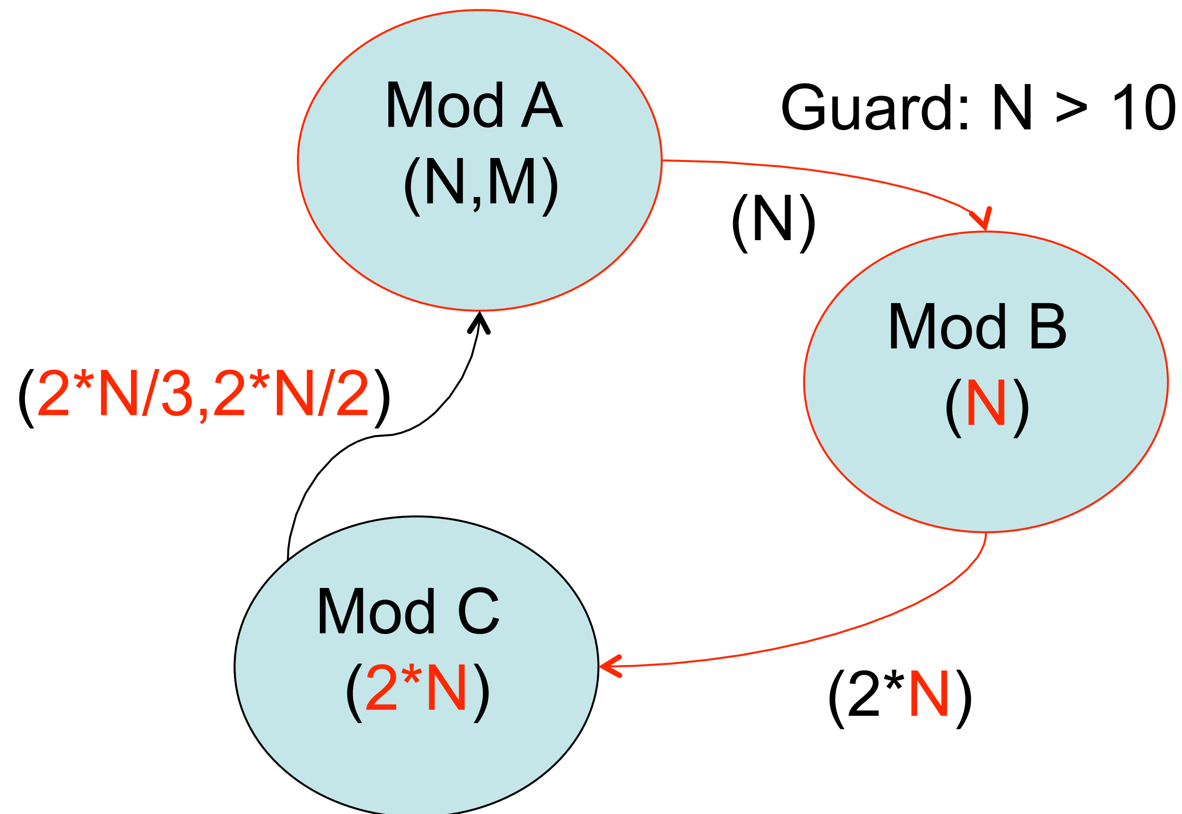
# Example



# Example



# Example

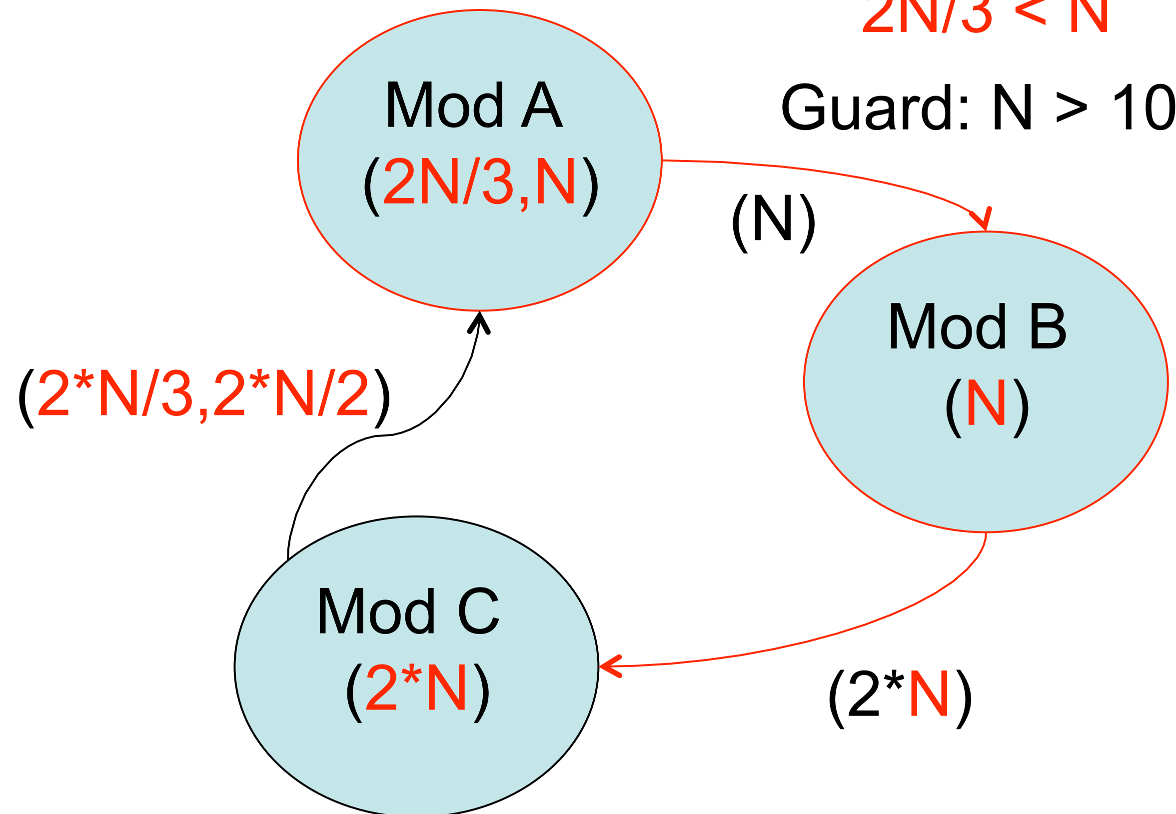


# Example

We only need to prove:

2N/3 < N

Guard: N > 10




An SMT solver can easily prove this

# Parametric Gate Count

- Gate count expressions can now include recurrences
- Currently VPP return them (unsolved)
- Plan is to use a recurrence relation solver (e.g. PURRS)
- Can the recurrence relation solver be used for termination analysis as well?

# Additional Extensions

- More Generic
  - Constraints on parameter values
  - Parameter currying
  - Algebraic data types
  - What else?
- More Accountable
  - Static delay estimation
  - Checking for short circuits
  - Adding and verifying fan-out constraints
  - What else?

# Conclusions

- Verilog is naturally
  - A two level language (elaboration)
  - Dependently typed (no extra annotations needed)
- Preprocessing allows for extending Verilog
  - Safely
  - In a backward compatible way
- Generative constructs (control + expressivity) + static guarantees (light-weight correctness check + resource estimation) should help in design space exploration and productivity

# Thank You

Questions

Feedback

Suggestions