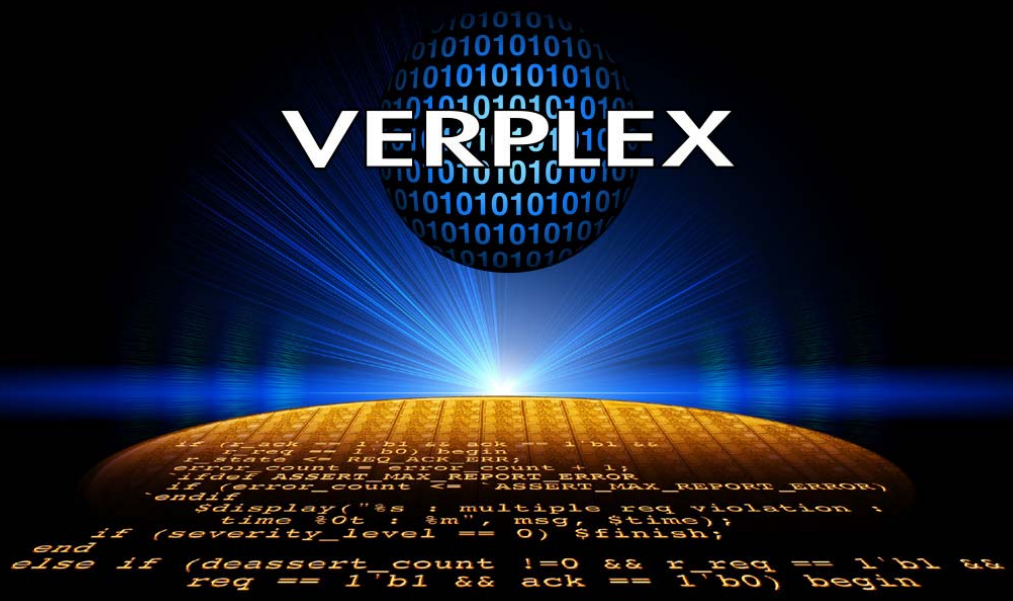
Property Specification:
The key to an assertion-based verification platform

Harry Foster



Electronic Design Processes 2003

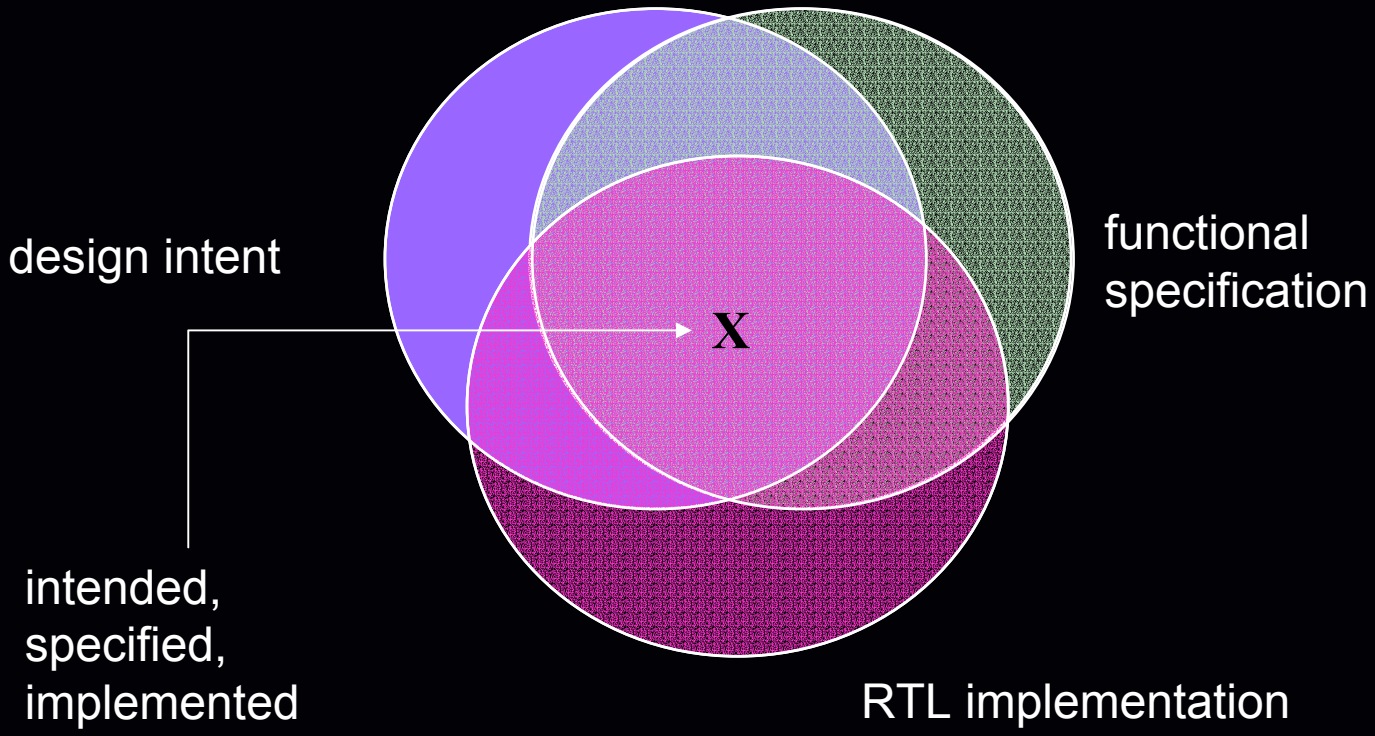
"As the requirements and complexities of electronic design increase, past ad-hoc approaches to design processes are proving inadequate."

- Verification doesn't occur in a vacuum.
- Specification must occur prior to any form of verification
- My talk focuses on the justification and benefits for moving to a more formal process

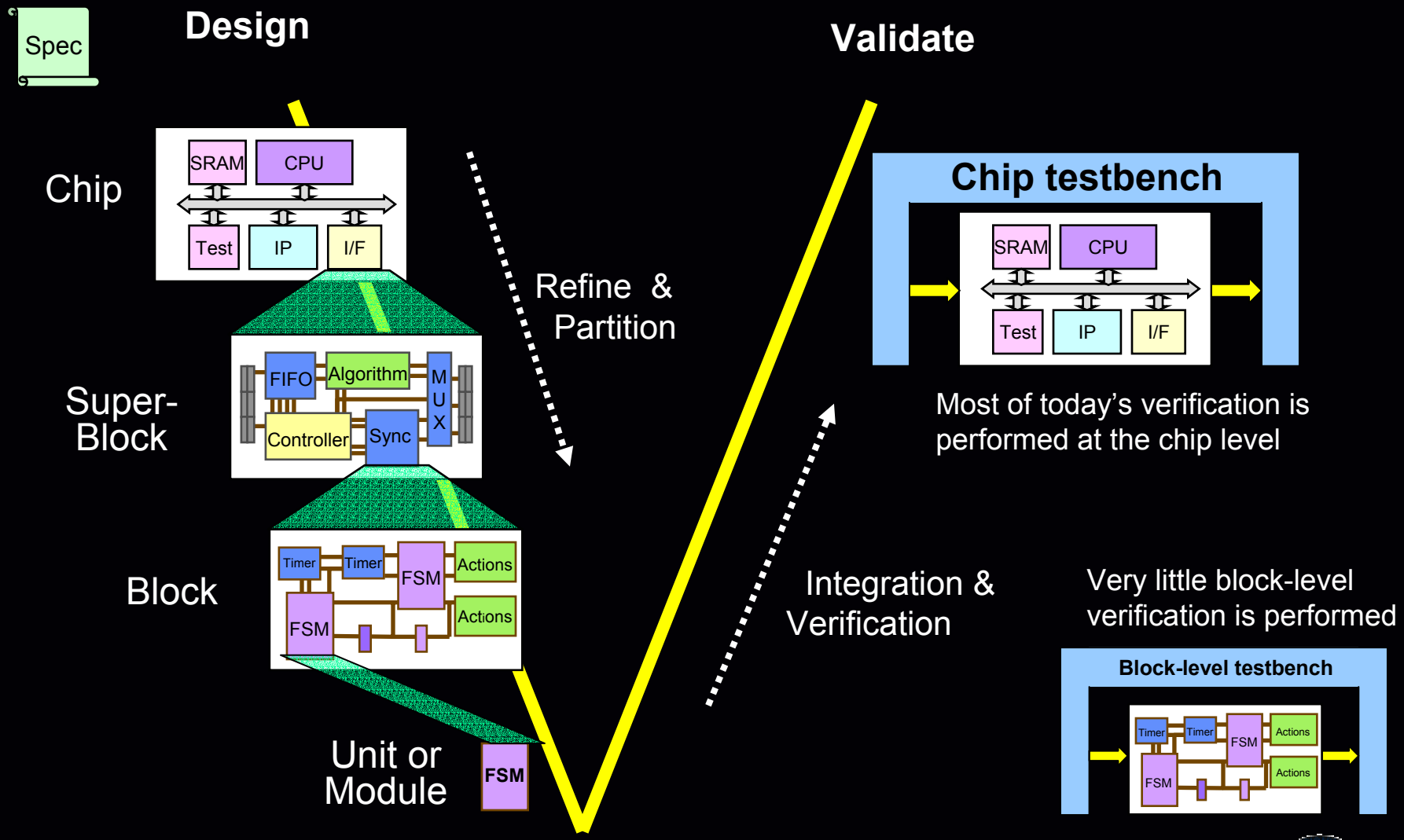
Design Intent Challenge

How can we specify the design intent in a form that can be verified?

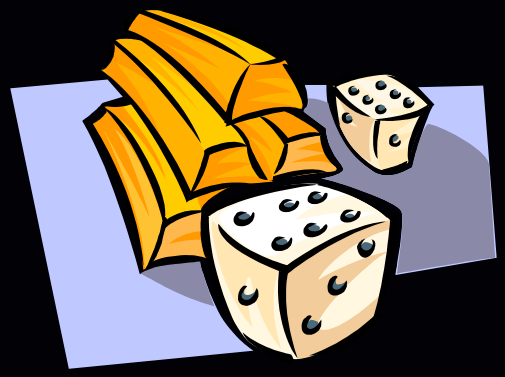
How can how can we know what has been specified has been verified?



Today's Design and Verification Flow



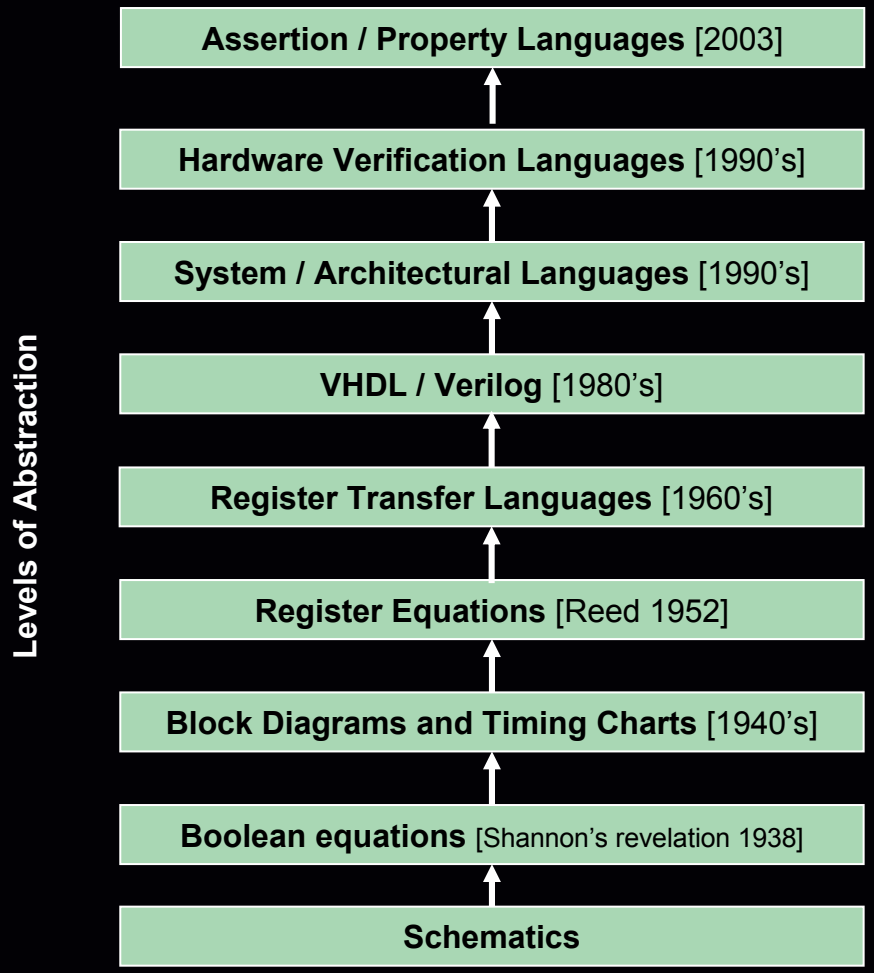
Prediction



- In the future, we will augment our natural language forms of specification with forms that are:
  - mathematically precise
  - verifiable
  - lend themselves to automation
- Design and verification will become property-based
- Multiple tools will operate on these properties
  - Synthesis, testbench generators, simulation assertions, functional coverage
- Formal and dynamic verification will become tightly integrated
  - Each will leverage the strengths of the other
- Property specification is the key ingredient of this revolution, whose end result is improved verification

Specification and Notation Drives Innovation

Standards drive opportunities



- The development of Register Transfer Languages in the mid-1960's lead to the development of synthesis.
- However, it was the standardization of VHDL/Verilog in the early 1990's that opened new markets and helped drive synthesis adoption.

The industry is raising the level of abstraction once again by introducing the assertion and property language standards!

An assertion-based verification platform

Standards will drive opportunities

- *verifiable testplans*
  - for example, executable *functional coverage models*, which help answer the question "*what functionality has not been exercised?*"
- *exhaustive* and *semi-exhaustive* formal property checking
  - for example, model checking and bounded-model checking)
- *dynamic property checking*
  - for example, monitoring assertions in simulation for improved observability reducing debug time
- *testbench generation*
  - leverage property specification to define expected input behavior (constraints) and output behavior (assertions)
- *constraint-driven stimulus generation*
  - based on interface properties (constraints)
- *assertion property synthesis*
  - to address silicon observability challenges during chip bring-up in the lab—as well as HA architectures for runtime analysis.

What is a property?

- **Definition:** ***property***—a collection of *logical* and *temporal relationships* between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behaviors.

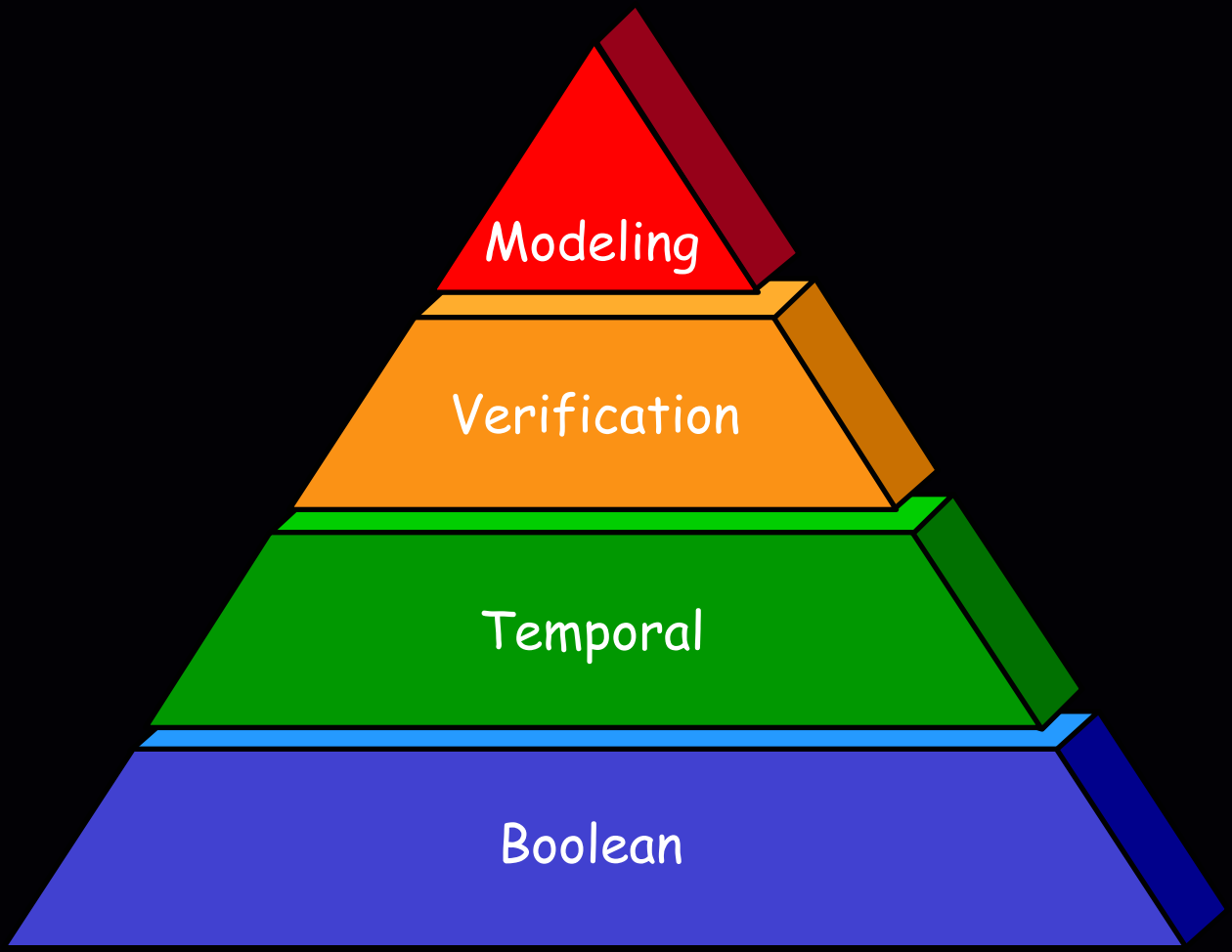
- In general, a *property* describes design intent.
  - Note that properties can either be specified by the designer or automatically extracted, based on structural analysis of the design model.

What is an assertion?

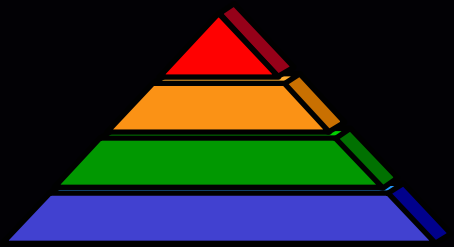
- **Definition:** ***assertion***—a statement that a given property is required to hold within a specific design—and a directive to verification tools to verify that it does hold.

- Its sole purpose is to ensure during verification consistency between the designer's *intention*—and what is *created*.

PSL Assertion Language Structure



PSL Assertion Language Structure



assert always (!(A & B)); *// a and b are always mutually exclusive*

Boolean layer

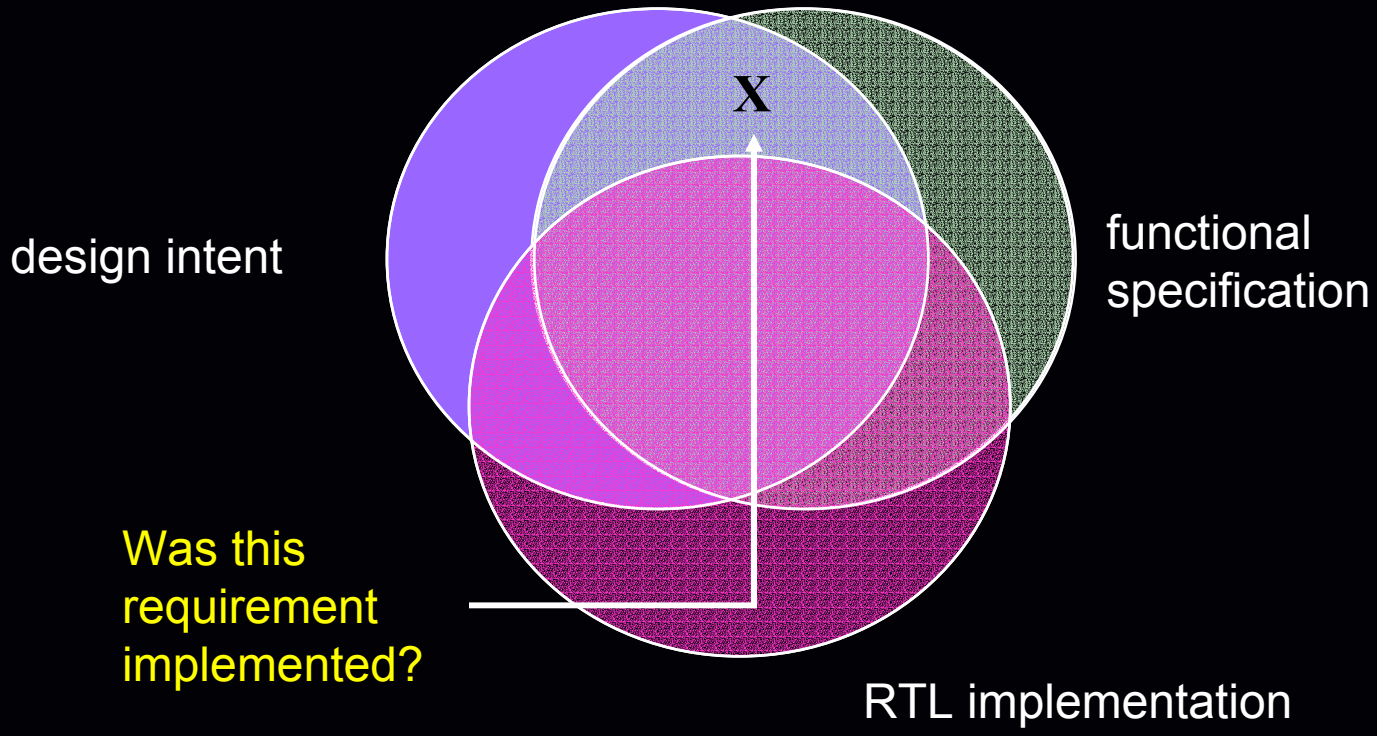
Temporal layer

Verification layer

Design Intent Challenge

How can we specify the design intent in a form that can be verified?

How can how can we know what has been specified has been verified?



Properties and Functional Coverage

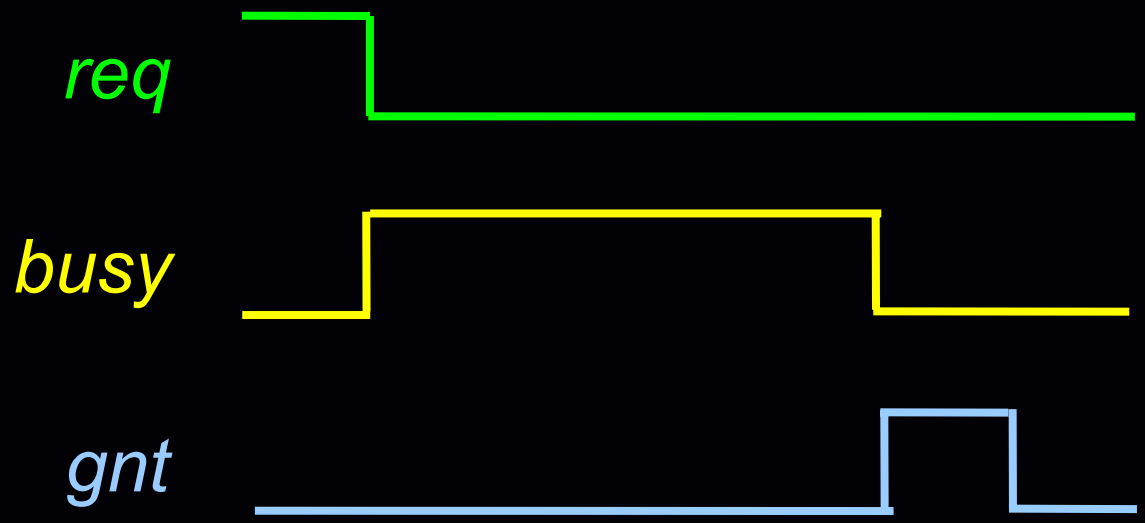
- A property language can specify assertions
  - which monitor and report *undesirable* behavior simulation
  - or can be used as proof targets for a formal engine
- A property language can specify functional coverage
  - which monitors and reports *desirable* behavior that must occur for the verification process to be complete
  - theoretically could be used by a formal engine to determine if an intended behavior is reachable—calculate a witness
- On the sx1000 project at Hewlett-Packard, the coverage model was comprised of over 14,000 functional coverage points. Analysis of the results revealed:
  - several key test generation features believed were enabled were actually disabled
  - helped identify specific functionality not exercised in any verification environment
  - for the first time—we knew exactly what random simulation was checking
  - 90% were hit through standard verification effort—10% required directed test

Functional coverage-transaction specification

A *sequence* describes a waveform

{ *req*; *busy*[*4]; *gnt* }

signal *busy* holds 4 times



AMBA AHB Transaction Example

- Advanced High-Performance Bus (AHB) protocol—supported by the ARM Advanced Microcontroller Bus Architecture (AMBA)
- AHB is a pipelined bus with all transfers taking at least two cycles to complete

< A's address phase > *< B's address phase >*

< A's data phase > *< B's data phase >*

AMBA AHB transaction example

```
sequence SERE_AHB_BURST_MODE_READ = {
  {SERE_AHB_READ_FIRST}; {SERE_AHB_READ_NEXT}[*]
};

cover {SERE_AHB_BURST_MODE_READ};
```

```
sequence SERE_AHB_SLAVE_RESPONSE = {
  `AHB_WAIT[*];      // (!hready && (hresp==`OKAY))
  {
    { `AHB_OKAY}
  | { {!hready;hready} && {`AHB_ERROR [*2]} }
  | { {!hready;hready} && {`AHB_SPLIT [*2]} }
  | { {!hready;hready} && {`AHB_RETRY [*2]} }
 }
};

// slave response to the previous data in parallel with the master's
// asserting the control signals for the next address

sequence SERE_AHB_READ_FIRST = {
  {SERE_AHB_SLAVE_RESPONSE} &&
  {(`AHB_FIRST_TRANS && `AHB_READ_INCR}[*]}
};

sequence SERE_AHB_READ_NEXT = {
  {SERE_AHB_SLAVE_RESPONSE} &&
  {
    {(`AHB_NEXT_TRANS && `AHB_READ_INCR)[*]}
  | {`AHB_MASTER_BUSY[*]}
  }
};

sequence SERE_AHB_BURST_MODE_READ = {
  {SERE_AHB_READ_FIRST}; {SERE_AHB_READ_NEXT}[+]
};

cover {SERE_AHB_BURST_MODE_READ};
```

```
sequence SERE_AHB_SLAVE_RESPONSE = {
  `AHB_WAIT[*];      // (!hready && (hresp==`OKAY))
  {
    { `AHB_OKAY}
  | { {!hready;hready} && {`AHB_ERROR [*2]} }
  | { {!hready;hready} && {`AHB_SPLIT [*2]} }
  | { {!hready;hready} && {`AHB_RETRY [*2]} }
 }
};

// slave response to the previous data in parallel with the master's
// asserting the control signals for the next address

sequence SERE_AHB_READ_FIRST = {
  {SERE_AHB_SLAVE_RESPONSE} &&
  {(`AHB_FIRST_TRANS && `AHB_READ_INCR)[*]}
};

sequence SERE_AHB_READ_NEXT = {
  {SERE_AHB_SLAVE_RESPONSE} &&
  {
    {(`AHB_NEXT_TRANS && `AHB_READ_INCR)[*]}
  | {`AHB_MASTER_BUSY[*]}
  }
};

sequence SERE_AHB_BURST_MODE_READ = {
  {SERE_AHB_READ_FIRST}; {SERE_AHB_READ_NEXT}[+]
};

cover {SERE_AHB_BURST_MODE_READ};
```

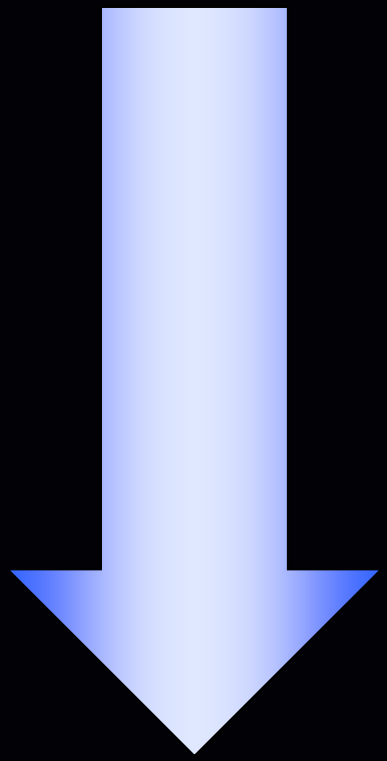
```
sequence SERE_AHB_SLAVE_RESPONSE = {
  `AHB_WAIT[*];      // (!hready && (hresp==`OKAY))
  {
    { `AHB_OKAY}
  | { {!hready;hready} && {`AHB_ERROR [*2]} }
  | { {!hready;hready} && {`AHB_SPLIT [*2]} }
  | { {!hready;hready} && {`AHB_RETRY [*2]} }
 }
};

// slave response to the previous data in parallel with the master's
// asserting the control signals for the next address

sequence SERE_AHB_READ_FIRST = {
  {SERE_AHB_SLAVE_RESPONSE} &&
  {(`AHB_FIRST_TRANS && `AHB_READ_INCR)[*]}
};

sequence SERE_AHB_READ_NEXT = {
  {SERE_AHB_SLAVE_RESPONSE} &&
  {
    {(`AHB_NEXT_TRANS && `AHB_READ_INCR)[*]}
  | {`AHB_MASTER_BUSY[*]}
  }
};

sequence SERE_AHB_BURST_MODE_READ = {
  {SERE_AHB_READ_FIRST}; {SERE_AHB_READ_NEXT}[+]
};

cover {SERE_AHB_BURST_MODE_READ};
```

AMBA AHB Transaction Example

PSL AHB valid transactions following the completion of any previous transaction

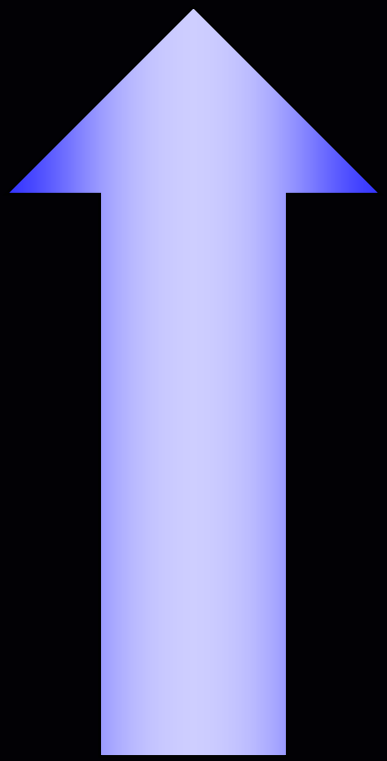
```
assert always {hready} |=> { SERE_AHB_BURST_MODE_READ
                           | SERE_AHB_BURST_MODE_WRITE
                           | SERE_AHB_SINGLE_READ
                           | SERE_AHB_SINGLE_WRITE
                           | SERE_AHB_INACTIVE
                           | SERE_AHB_RESET
                           };
```

Levels of property specification



Top Down Design

Bottom Up Verification



Architecture

Block

RTL Module

System

Block Integration

RTL Design

An RTL implementation level property

```
module fifo (clk,  fifo_clr_n, fifo_reset_n,
      put, // put strobe, active high
      get,   // get strobe, active high
      data_in, data_out
);
  // FIFO parameters
  parameter fifo_width  = `FIFO_WIDTH;
  parameter fifo_depth  = `FIFO_DEPTH;
  parameter fifo_pntr_w = `FIFO_PNTR_W;
  parameter fifo_cntr_w = `FIFO_CNTR_W;
  input clk, fifo_clr_n, fifo_reset_n, put, get;
  input  [fifo_width-1:0] data_in;
  output [fifo_width-1:0] data_out;

  wire [fifo_width-1:0] data_out;
  // FIFO itself
  reg [fifo_width-1:0] fifo[fifo_depth-1:0];
  // FIFO pointers
  reg [fifo_pntr_w-1:0] top; // top
  reg [fifo_pntr_w-1:0] btm; // bottom
  reg [fifo_cntr_w-1:0] cnt; // count
 Integer i;
```

```
always @(posedge clk or negedge fifo_clr_n)
begin
 if (fifo_clr_n == 1'b0) begin
   top    <= {fifo_pntr_w {1'b0}};
   btm <= {fifo_pntr_w {1'b0}};
   cnt  <= { fifo_cntr_w {1'b0}};
   for (i=0; i<fifo_depth; i=i+1)
     fifo[i] <= {fifo_width{1'b0}};
 end
 else if (fifo_reset_n == 1'b0) begin
   top    <= {fifo_pntr_w {1'b0}};
   btm <= {fifo_pntr_w {1'b0}};
   cnt  <= { fifo_cntr_w {1'b0}};
 end
 else begin
  case ({put, get})
    2'b10 : // WRITE
       if (cnt<fifo_depth)  begin
         fifo[top] <= data_in;
         top <= top +  1;
         cnt <= cnt +  1;
       end
```

An RTL implementation level property

```
  2'b01 : //READ
      if(cnt>0) begin
        fifo[btm] <= 0;
        btm <= btm + 1;
        cnt  <= cnt - 1;
      end
    2'b11 : // WRITE & READ
      begin
        fifo[btm] <= 0;
        fifo[top] <= data_in;
        btm <= btm +  1;
        top   <= top +  1;
      end
   endcase
  end
end   // always

assign data_out = fifo[btm];
```

```
// Assert that the FIFO cannot overflow
assert never ({put,get}==2'b10 && cnt==fifo_depth-1) @(posedge clk);

// Assert that the FIFO cannot underflow
assert never (get && cnt==0) @(posedge clk);
```

An RTL implementation level property

```
// Assert that the FIFO cannot overflow
  assert never ({put,get}==2'b10 && cnt==fifo_depth-1) @(posedge clk);

  // Assert that the FIFO cannot underflow
  assert never (get && cnt==0) @(posedge clk);
```

Key features of an assertion

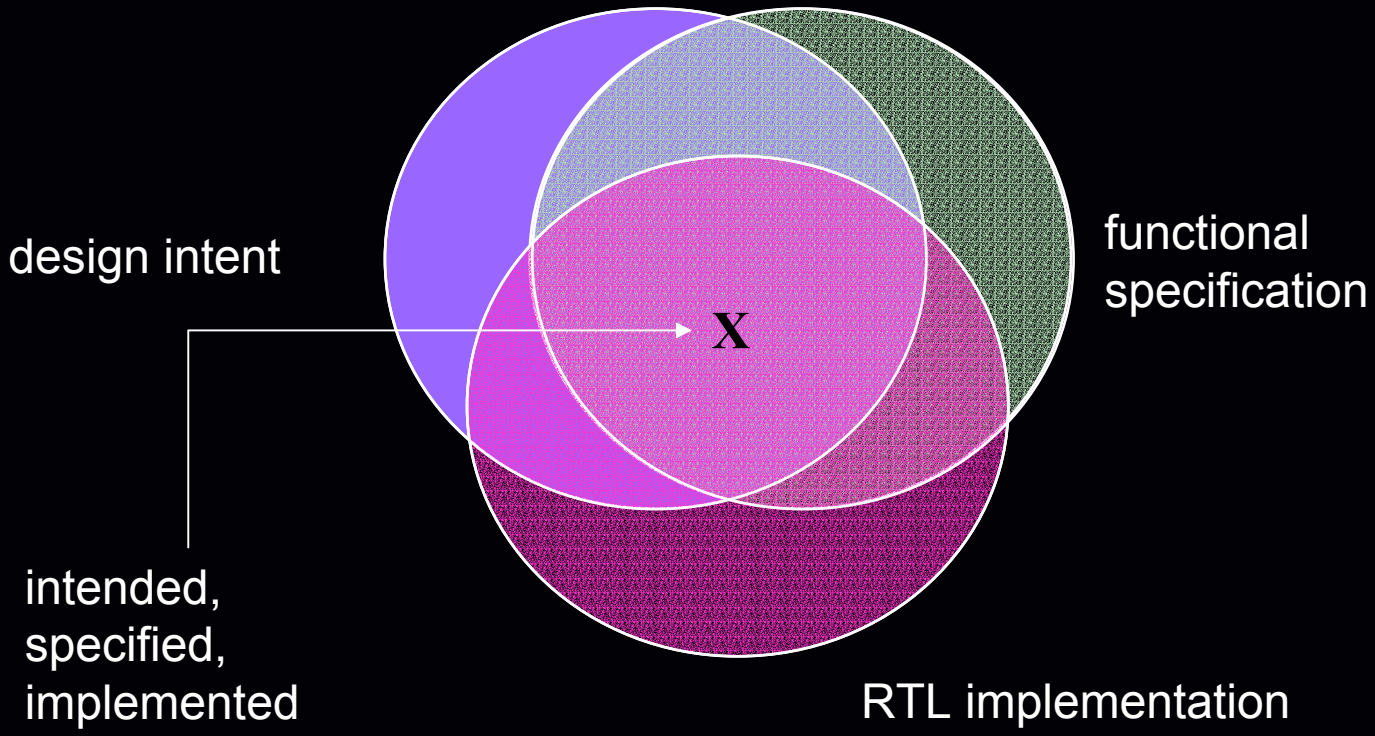
- The previous example demonstrates three key features of assertions:

1. *Error detection,*
2. *Error isolation,* and
3. *Error notification*.

Design intent challenge

How can we specify the design intent in a form that can be verified?

How can we know whether what has been specified has been verified?



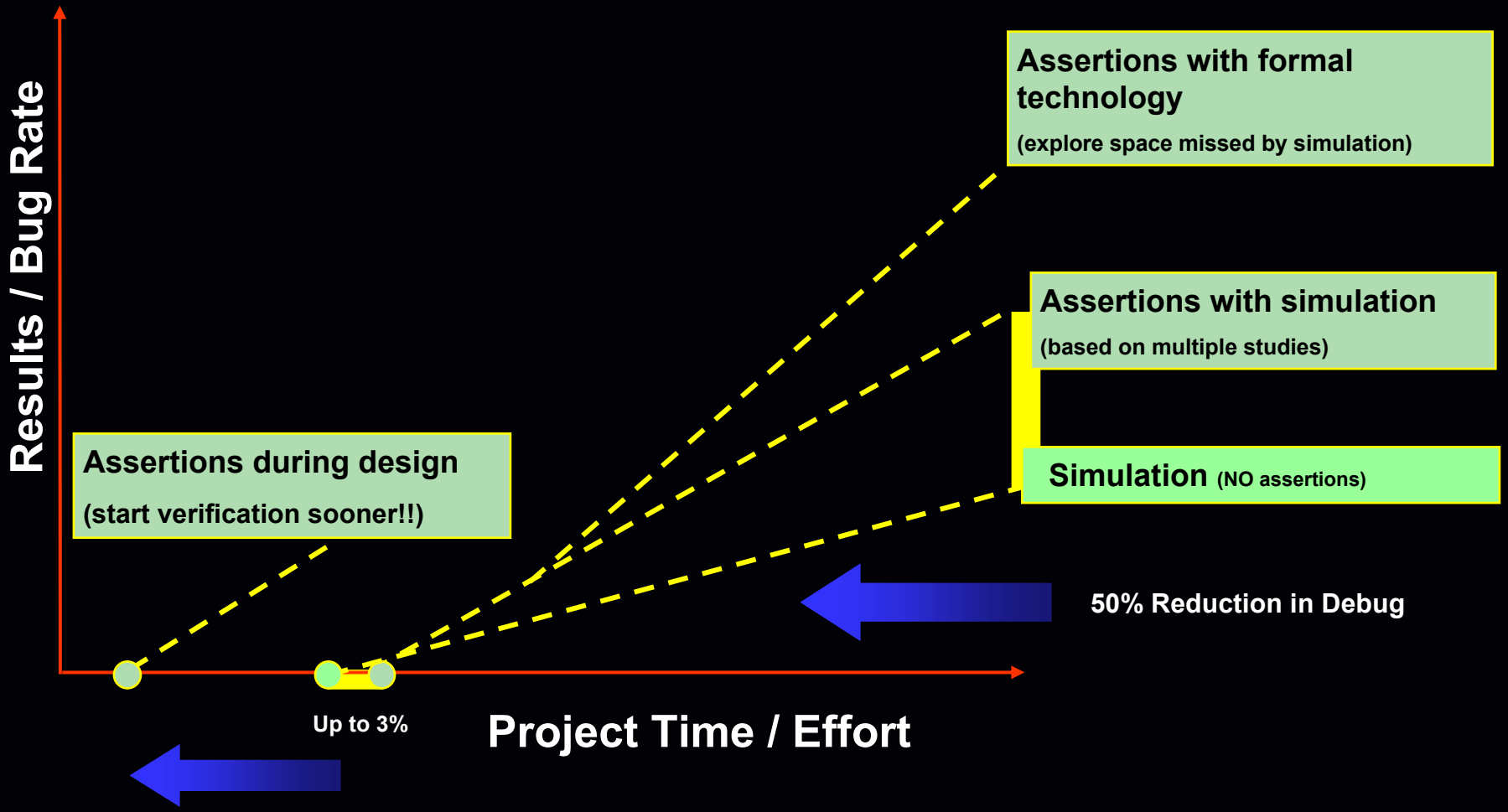
Conclusion

- Design and verification will become *property-based*
- Multiple tools will operate on these properties
  - Synthesis, testbench generators, simulation assertions, functional coverage
- Formal and dynamic verification will become tightly integrated
  - Each will leverage the strengths of the other

Property specification is the key ingredient
required for assertion-based verification platform!

Backup

Assertion Effort Payback Claim



Do Assertions Really Work?

Assertions in real designs:

Assertion Monitors	34%
Cache Coherency Checkers	9%
Register File Trace Compare	8%
Memory State Compare	7%
End-of-Run State Compare	6%
PC Trace Compare	4%
Self-Checking Test	11%
Simulation Output Inspection	7%
Simulation hang	6%
Other	8%

Kantrowitz and Noack *[DAC 1996]*

Assertion Monitors	25%
Register Miscompare	22%
Simulation "No Progress"	15%
PC Miscompare	14%
Memory State Miscompare	8%
Manual Inspection	6%
Self-Checking Test	5%
Cache Coherency Check	3%
SAVES Check	2%

Taylor et at. *[DAC 1998]*

- **34%** of all bugs were found were identified by assertions on **DEC** Alpha 21164 project
  [Kantrowitz and Noack DAC 1996]
- **17%** of all bugs were found were identified by assertions on **Cyrix** M3(p1) project.
  [Kronik '98]
- **25%** of all bugs were found were identified by assertions on **DEC** Alpha 21264 project.
  [Taylor et at.DAC 1998]
- **50%** of all bugs found were identified by assertions on **Cyrix** M3(p2) project
  [Kronik '98]
- **85%** of all bugs were found using over 4000 assertions on **HP**
  [Foster and Coelho HDLCon 2000]