The Semantic Challenge of Verilog HDL

HDL = Hardware Description Language

Overview of talk:

- HDL-based design
- Verilog and VHDL
- Tutorial introduction to Verilog
- Semantic challenges
The Semantic Challenge of Verilog HDL

Modern HDLs

- Wide spectrum
  - behaviour
  - structure
  - test harnesses

- Event simulation semantics
  - changes propagated via event scheduling

- Language wars
  - VHDL (based on Ada) versus
  - Verilog (based on C)

"VHDL is one of the biggest mistakes the Electronics Design Automation industry has ever made"

[Attributed to Joe Costello (CEO of Cadence) in John Cooley’s conference report on IVC ’95]
Uses of HDLs

- Behavioural prototyping
  - initial proof of concept
  - behavioural models later refined

- Specification checking
  - double entry comparative simulation
  - compare specification with implementation

- Verification (by simulation)
  - apply test data
Verilog HDL

- Widely used
  - Sun, Apply, Hewlett-Packard …
  - 25,000 Verilog designers today
  - 5,000 new ones each year
  - twice market share of VHDL (93 estimate)

- Taught to second year CS undergraduates at Cambridge University

- Designed by industry (Gateway/Cadence)
  - VHDL designed by Government committee

- Supported by fast simulators
  - many component models available

- Undergoing IEEE standardization
Overview of Verilog (1)

- **Modules** are main units of behaviour
- Module behaviour can be specified:
  - **behaviourally**: \( o = \neg (i_1 \land i_2) \)

```verilog
module NAND (i1, i2, o);
    input i1, i2;
    output o;
    assign o = ~(i1 & i2);
endmodule
```

- **structurally**:

```
module AND (i1, i2, o);
    input i1, i2;
    output o;
    wire w;
    NAND NAND1(i1, i2, w);
    NAND NAND2(w, w, o);
endmodule
```

Mike Gordon  
http://www.cl.cam.ac.uk/users/mjcg/  
LICS'95, San Diego
Overview of Verilog (2)

- Each module has:
  - name
  - port list
  - declarations
  - body

- The body consists of one or more items:
  - `<module instance>`
  - `<continuous assignment>`
  - initial `<statement>`
  - always `<statement>`

- Each item generates a separate thread
Events and Scheduling

- Events are changes to wires or registers (also abstract named events – ignored here)

- Statements can schedule events to:
  - occur at particular times
  - be triggered by other events
    * in the current time slot
    * at a later simulation time

- Several execution threads may be active
  - they are enabled, delayed or guarded

- Simulation time advances when all enabled threads have run
Zero-delay Assignments

- **Continuous** assignments: \( \text{assign } w = e \)
  - whenever value of \( e \) changes
  - value on wire \( w \) scheduled for updating:
    * with new value of \( e \)
    * in the current time slot

- **Procedural** assignment: \( r = e \)
  - when reached in sequence
  - register \( r \) scheduled for updating
    * with value of \( e \)
    * as next event in current thread

- **Non-blocking** assignment: \( r <= e \)
  - when reached in sequence
  - register \( r \) scheduled for updating
    * with new value of \( e \)
    * end of current slot as a separate thread
Example Assignments

- **Continuous assignment:**
  
  ```verilog
  assign z = x + y;
  ```

  - whenever \( x + y \) changes
  - \( z \) scheduled for updating

- **Blocking procedural assignment**
  
  ```verilog
  x = 1; y = 2; x = y; y = x;
  ```

  - result: \( x = 2 \) and \( y = 2 \)

- **Non-blocking procedural assignment**
  
  ```verilog
  x = 1; y = 2; x <= y; y <= x;
  ```

  - result: \( x = 2 \) and \( y = 1 \)
Timing Control

- **Delay control**
  
  \[ \#50 \text{<statement>} \]
  
  delays \text{<statement>} for 50 units of simulation time

- **Event control**
  
  \[ @(x \text{ or } y) \text{<statement>} \]
  
  delays \text{<statement>} until \(x\) or \(y\) changes

- **Edge-sensitive event control**
  
  \[ @(\text{posedge } clk) \text{<statement>} \]
  
  delays \text{<statement>} until \(clk\) changes to 1

- **Level-sensitive event control**
  
  \[ \text{wait}(c) \text{<statement>} \]
  
  delays \text{<statement>} until \(c\) becomes true
Timing Controlled Assignments:
Continuous Assignments

- assign #50 w = e
- whenever value of e changes
- value on wire w is scheduled for updating:
  - with new value of e
  - after 50 units of simulation time
- inertial delay
  - changes persisting for < 50 are ignored
Timing Controlled Assignments: Procedural Assignments

- **Blocking** assignment: \( r = \#50 \, e \)
  - when reached in sequence
  - register \( r \) scheduled for updating with current value of \( e \) after delay of 50
  - Sequential thread delayed for 50

- **Non-blocking** assignment: \( r <= \#50 \, e \)
  - when reached in sequence
  - separate thread created
    * update \( r \) with current value of \( e \) after delay of 50
  - old thread not delayed
    (sequential flow not blocked)
Inertial versus Transport Delay

- Compare:
  1. `assign #50 z = x + y`
  2. `always @(x or y) z <= #50 x + y`

- Differences:
  1. - `z` must be a wire
     - specifies inertial delay
     - changes scheduled immediately
  2. - `z` must be a register
     - specifies transport delay
     - `@` scheduled changes to end of slot
Inertial Assignment (in VHDL but not Verilog)

- Inertial assignment: \( r <- \#50 \ e \)
  - when reached in sequence
  - separate thread created
    * update \( r \) with current value of \( e \)
  - old thread not delayed
    * i.e. non-blocking
  - thread killed if \( r \) changes within 50
    * i.e. inertial
More on Event Control

- Recall: \( @ (x \text{ or } y) \ <statement> \)
- Delays \(<statement>\) until \(x\) or \(y\) changes
- Actually delays until \(\text{end}\) of time slot in which change occurs
- New construct: \( \Delta (x \text{ or } y) \ <statement> \)
- delays \(<statement>\) until \(x\) or \(y\) changes
  - \(<statement>\) enabled
    - \(x\) or \(y\) change
- \(\Delta\) not in Verilog
  - added for semantics of continuous assignment
- \( @ (x \text{ or } y) = \Delta (x \text{ or } y) \ #0 \)
Semantics of Continuous Assignment

- Associate register $\hat{w}$ with each wire $w$
- Suppose $e$ contains variables $x_1, \ldots, x_n$
- Define
  
  \[
  \text{assign } #n \ w = e
  \]

  to stand for

  \[
  \text{always } \Delta(x_1 \text{ or } \cdots \text{ or } x_n) \hat{w} \leftarrow #n \ e
  \]
Verilog’s Data Types

- **Four basic values:**
  - **0:** logic zero, or false
  - **1:** logic one, or true
  - **x:** unknown logic value
  - **z:** high impedance state

- **Vectors** represent words and busses
  
  ```
  reg[3:0] v
  ```
  - declares v to be a 4-bit register
  - components are: v[3], v[2], v[1], v[0]

- **Also:**
  - memories, integers, reals
  - events
Complete Example: a Multiplier

Mike Gordon
http://www.cl.cam.ac.uk/users/mjcg/
LICS’95, San Diego
The Multiplier in Verilog

- **Specification:**
  
  ```verilog
  reg CARRY;
  
  PROD = #(2*n) A * B;
  ```

- **Implementation**

  ```verilog
  begin
  P = 0; A = X; B = Y;
  repeat (n)
  begin #1 {CARRY,P} = P + (n{A[0]}) & B;
    #1 {P,A} = {CARRY,P,A} >> 1;
  end
  PROD = {P,A};
  end
  ```
module MULT(X, Y, PROD);
parameter n = 4;
input [n-1:0] X, Y; output [2*n-1:0] PROD;
reg CARRY;
reg [n-1:0] P, A, B; reg [2*n-1:0] PROD;

always @(X or Y)
begin
    P = 0; A = X; B = Y;
    Display "Start: %Time, %A, %B"
    repeat (n)
        begin #1 {CARRY,P} = P + ({n{A[0]}} & B);
            #1 {P,A} = {CARRY,P,A} >> 1;
        end
    PROD = {P,A};
    Display "End: %Time, %PROD"
end
endmodule
module MULT_TEST_DATA (x,y);
parameter n = 4;
output x,y; reg [n-1:0] x,y;

initial
begin x = 0;
    forever
    begin
        y = 0;
        while (y <= x) #10 y = y + 1;
        x = x + 1;
    end
end

endmodule
module MULT_TEST ();
    parameter n = 4;
    wire [n-1:0] x,y;
    wire [2*n-1:0] p;
    MULT_TEST_DATA M1(x,y);
    MULT M2(x,y,p);
    initial #1675 $finish;
endmodule
Output from Simulator

Start: Time = 0, A = 0 (0000), B = 0 (0000)
End:   Time = 8, PROD = 0 (00000000)

Start: Time = 10, A = 1 (0001), B = 0 (0000)
End:   Time = 18, PROD = 0 (00000000)

Start: Time = 1660, A = 15 (1111), B = 14 (1110)
End:   Time = 1668, PROD = 210 (11010010)

Start: Time = 1670, A = 15 (1111), B = 15 (1111)
End:   Time = 1678, PROD = 225 (11100001)
Semantic Challenges

- Formal semantics of Verilog
- Validity of simplified semantics
- A minimal simulation calculus
- Equivalence between modules
- Correctness of synthesizers
Formal Semantics of Verilog

- Devise a formal semantics
  - accurate to the spirit of the language
  - c.f. IEEE Simulation Semantics

- Diverse approaches for VHDL:
  - stream processing (Fuchs & Mendler)
  - functional programming (Breuer et al)
  - labelled transition systems (Van Tassel)
  - evolving algebras (Börger et al.)
  - Petri nets (Olcoz)
  - automata (Döhmen & Herrmann)
  - flow graphs (Reetz & Kropf)
  - denotational semantics (Davis)
  - state-delta temporal logic (Filippenko)
Simplified Semantics

- **Simulation semantics:**
  - easy to formalise (maybe)
  - hard to work with

- **Need simpler semantics**
  - maybe just for ‘well-behaved’ subsets
  - needs to be related to simulation semantics

- **Tractable semantics are level oriented**
  - simulation semantics is edge-oriented
  - how can these be related?

Mike Gordon
http://www.cl.cam.ac.uk/users/mjcg/

LICS’95, San Diego
A Minimal Simulation Calculus

- Verilog is large and complicated
- Need to distill essence into a simple setting
  - nice to also handle VHDL cycle
- Consider:
  
<table>
<thead>
<tr>
<th>ML and Haskell</th>
<th>λ-calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occam</td>
<td>CSP</td>
</tr>
<tr>
<td>Lotos</td>
<td>CCS</td>
</tr>
<tr>
<td>VHDL, Verilog</td>
<td>???</td>
</tr>
</tbody>
</table>

- Need theory of equivalence and refinement
- Relation to existing:
  - process calculi
  - programming logics

Mike Gordon
http://www.cl.cam.ac.uk/users/mjcg/

LICS'95, San Diego
Equivalence Between Modules

- Proof-of-concept behavioural prototypes are refined to implementations
- Costly if prototype and implementation differ
- Example:
  
  **prove**

  \[
  \text{PROD} = \#(2n) \ A \times B;
  \]

  **equivalent to:**

  ```
  begin
  \text{P} = 0; \ \text{A} = \text{X}; \ \text{B} = \text{Y};
  repeat (n)
  begin #1 \{\text{CARRY},\text{P}\} = \text{P} + (\{n\{\text{A}[0]\}\} \ & \ \text{B});
  \ #1 \{\text{P},\text{A}\} = \{\text{CARRY},\text{P},\text{A}\} \gg 1;
  end
  \text{PROD} = \{\text{P},\text{A}\};
  end
  ```
Correctness of Synthesisers

- Synthesis algorithms should be correct
  - nice to formally prove correctness
  - w.r.t. simulation semantics
- Existing work uses simplified semantics
  - need to verify validity
- Example on next two slides ⋮
Example Synthesisable Module

module EXAM(clk, p, q, r, s, out);

    input clk, p, q, r, s;
    output [1:0] out;
    reg [1:0] out;

    always @(posedge clk)
    begin
        if (p)
            if (q & ~r)
                out <= { ~s, s };
            else
                out <= ~out;
        end
    endmodule

Mike Gordon
http://www.cl.cam.ac.uk/users/mjcg/
LICS'95, San Diego
Example of Design Synthesis

module EXAM(clk, p, q, r, s, out);
    input clk, p, q, r, s; output [1:0] out;
    reg [1:0] out;
    always @(posedge clk)
        begin
            if (p) if (q & ~r) out <= { ~s, s }; else out <= ~out;
        end
endmodule

module EXAM(clk, p, q, r, s, out);
    wire u10016, u10015, u10014, u10013, u10012, u10011, u10010;
    wire u10009, u10008, u10007, u10006, u10005, u10004, u10003;
    wire [1:0] u10002; output [1:0] out; input s, r, q, p, clk;
    INV u10014(u10014, out[0]);
    CVMUX2 u10015(u10015, u10010, s, out[0]);
    CVMUX2 u10016(u10016, u10008, u10014, u10015);
    BUF u10017(u10002[0], u10016); INV u10005(u10005, r);
    AND2 u10006(u10006, q, u10005); INV u10007(u10007, u10006);
    AND2 u10008(u10008, p, u10007); INV u10009(u10009, out[1]);
    AND2 u10010(u10010, p, u10006); INV u10011(u10011, s);
    CVMUX2 u10012(u10012, u10010, u10011, out[1]);
    CVMUX2 u10013(u10013, u10008, u10009, u10012);
    BUF u10018(u10002[1], u10013);
    DFF u10004(u10004, u10002[0], clk, 1, 0, 0);
    BUF u10019(out[0], u10004);
    DFF u10003(u10003, u10002[1], clk, 1, 0, 0);
    BUF u10020(out[1], u10003);
endmodule
Conclusions

- Verilog & VHDL are real-world languages
- Need more theoretical support
- Pose interesting challenges
  - semantic
  - logical
- Formal methods for electronic design automation (EDA) are starting to be commercially significant

My name is Henry Cox. I am in the process of preparing a report discussing the potential commercialization of formal verification for a large EDA vendor. (I signed a NDA, so I’m afraid that I can’t tell you who it is.)

[recent email message]