Verilog HDL

Gordon J. Pace

March 2000

Comparing HDLs

- Usability
  - Expressive programming language
    (Fast design)
  - Abstraction mechanisms
    (Reusable design)
- Description style
  - Structural
  - Register Transfer Level
  - Behavioural
  - Mathematical

What should I be able to do with a HDL specification?

- Refinement
- Simulation
- Synthesis
- Verification

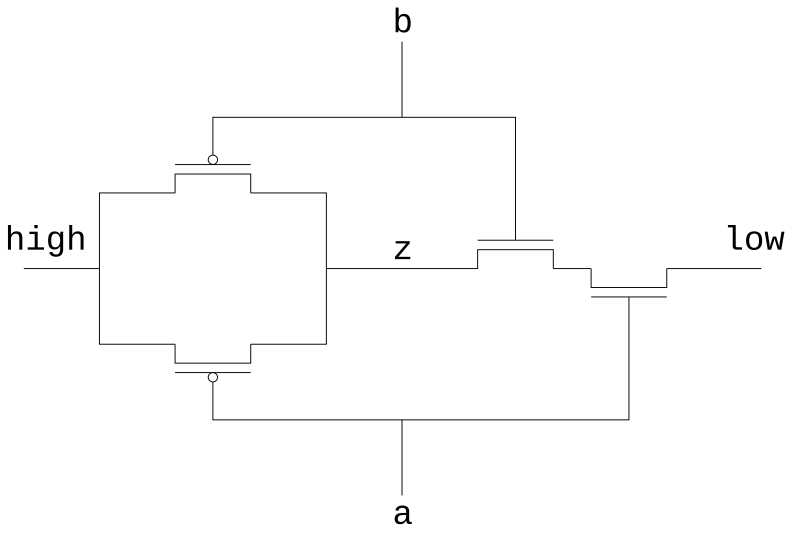
History of VHDL

- Based on Ada
- American DoD documentation language
- 1981: VHSIC committee formed
- 1985: IEEE standard started
- 1987: Standard finished
- 1993: Standard updated

History of Verilog

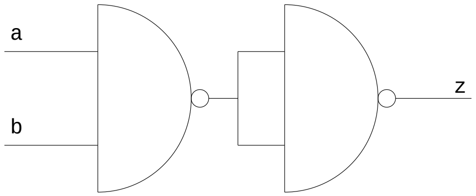
- Based on C
- 1983: Gateway Design Corporation propriety language
- 1987: Bought by Cadence
- 1990: OVI formed and language made public domain
- 1992: IEEE standard started
- 1995: Standard approved

Structural Verilog (I)



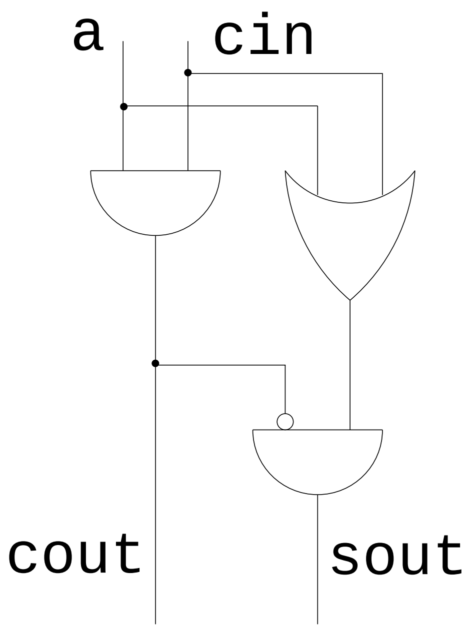

```
module Nand(a,b,z);
  assign high=1;
  assign low=0;
  transif0 T1 (a,high,z);
  transif0 T2 (b,high,z);
  transif1 T3 (a,low ,tmp);
  transif1 T4 (b,tmp ,z);
endmodule
```

Structural Verilog (II)




```
module And(a,b,z);
   Nand NandLeft (a,b,tmp);
   Nand NandRight (tmp,tmp,z);
endmodule
```

RTL Verilog




```
module HalfAdderRTL(a,cin,sout,cout);
   assign sout = (a | cin) & ~cout
   assign cout = a & cin;
endmodule
```

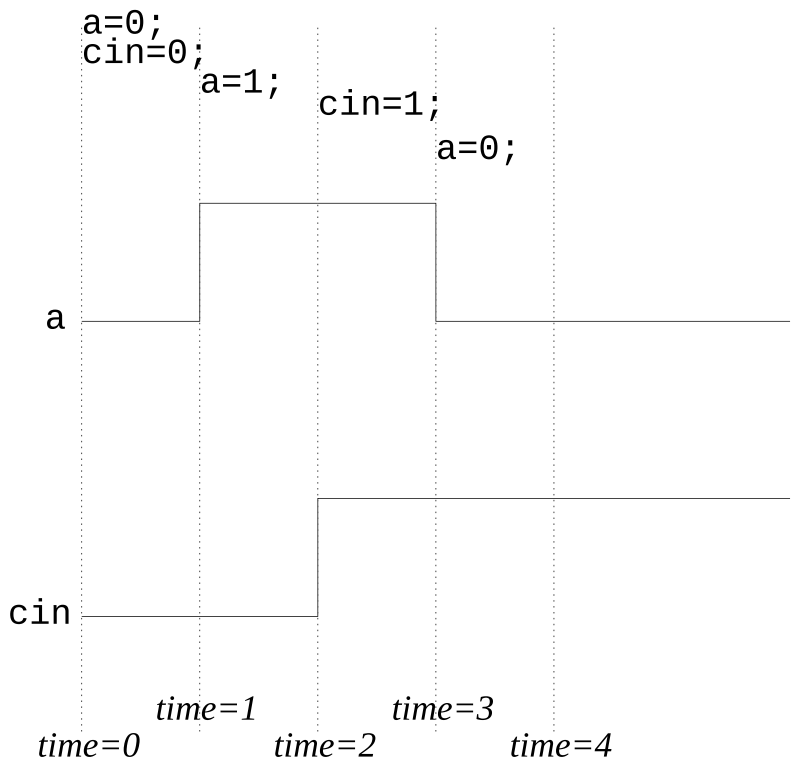
Behavioural Verilog

```
module HalfAdderBeh(a,cin,sout,cout);
   always
      begin
         @(a or cin)
         cout = a & cin;
         sout = (a | cin) & ~cout;
      end
endmodule
```

Behavioural Verilog

```
module TestBench;
   HalfAdderRTL(a,cin,sout_rtl,cout_rtl);
   HalfAdderBeh(a,cin,sout_beh,cout_beh);

   initial
      begin
            a   = 0;
            cin = 0;
         #1 a   = 1;
         #1 cin = 1;
         #1 a   = 0;
      end
endmodule
```



A Short Digression

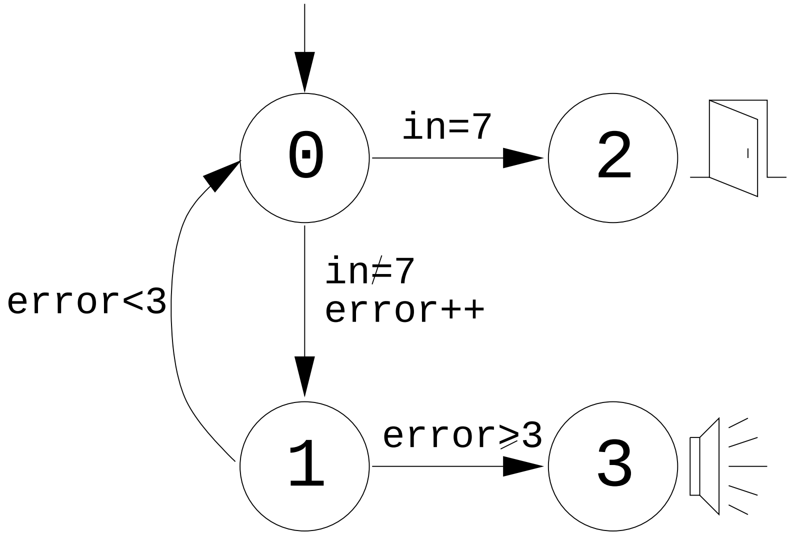
- Variable types
- Variable initialization
- Zero delay:

```
initial x=0; #1; x=0;
initial x=1; #1; #0 x=1;
```

Add/Sub Register

```
initial
    begin
       total = 0;
       while (1)
          begin
             @(posedge clk)
             {overflow,total} = total+in;
             @(posedge clk)
             {overflow,total} = total-in;
          end
    end
```

State Machine Encoding




```
initial
   state=0; error=0; open=0; siren=0;
   forever @(posedge clk)
           case (state) of
           0:  if (input==7) state=2;
                   else begin
                           state=1;
                           error=error+1;
                        end;
           1:  if (error>=3) state=3;
                   else state=0;
           2:  open=1;
           3:  siren=1;
           endcase
```

Signals and Variables: VHDL Style

```
initial
   begin
      w <= 1;
      v =  w;
   end
```

```
initial
   begin
      w <= v;
      v <= w;
   end
```

```
initial
   begin
      w <= 1;
      #0 v = w;
   end
```

The Simulation Cycle

1. Transform into one flat module
2. Create one parallel thread per module item
3. Initialise variables
4. Run along threads which are enabled
5. Enable all threads delayed by 0 (if any go to 4)
6. Perform one non-blocking assignment (then go to 4)

Flattening the modules

```
module halfadder (a,cin,s,c);
   always
      @(a or cin)
      c = a & cin;

   always
      @(cin or a)
      #0
      s <= ~c & (cin | a);
endmodule

module fulladder (cin,a,b,s,c);
   initial
       @(c1 or c2)
       s = c1 | c2;

   halfadder HA1 (a,cin,s1,c1);
   halfadder HA2 (b,s1,s,c2);
endmodule
```

```
initial
      @(c1 or c2)
      s = c1 | c2;
initial
   while (1)
      @(a or cin)
      c1 = a & cin;
initial
   while (1)
      @(cin or a)
      #0
      s1 <= ~c1 & (cin | a);
initial
   while (1)
   @(b or s1)
    c2 = a & s1;
initial
   while(1)
      @(s1 or b)
      #0
      s <= ~c2 & (s1 | b);
```

Simulation: State

- The values of the variables
- Position in the threads
- Threads can be:
  - `enabled`
  - `delayed by n`
  - `guarded by @g`
  - `finished`
- A queue of non-blocking queue of assignments to each variable

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = x
cin = x
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = empty
c1  = empty
```

States of threads:

```
Thread 1: enabled
Thread 2: enabled
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = x
cin = x
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = empty
c1  = empty
```

States of threads:

```
Thread 1: @(cin or a)
Thread 2: enabled
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = x
cin = 1
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = empty
c1  = empty
```

States of threads:

```
Thread 1: enabled
Thread 2: enabled
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = 1
cin = 1
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = empty
c1  = empty
```

States of threads:

```
Thread 1: enabled
Thread 2: finished
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = 1
cin = 1
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = [0]
c1  = empty
```

States of threads:

```
Thread 1: enabled
Thread 2: finished
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = 1
cin = 1
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = [0]
c1  = empty
```

States of threads:

```
Thread 1: delayed by 0
Thread 2: finished
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = 1
cin = 1
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = [0]
c1  = empty
```

States of threads:

```
Thread 1: enabled
Thread 2: finished
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = 1
cin = 1
s1  = x
c1  = 1
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = [0]
c1  = empty
```

States of threads:

```
Thread 1: finished
Thread 2: finished
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = 1
cin = 1
s1  = 0
c1  = 1
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = empty
c1  = empty
```

States of threads:

```
Thread 1: finished
Thread 2: finished
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = x
cin = x
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = empty
c1  = empty
```

States of threads:

```
Thread 1: enabled
Thread 2: enabled
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = x
cin = 1
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = empty
c1  = empty
```

States of threads:

```
Thread 1: enabled
Thread 2: enabled
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = 1
cin = 1
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = empty
c1  = empty
```

States of threads:

```
Thread 1: enabled
Thread 2: finished
```

```
initial
      @(cin or a)
      s1 <= ~(cin & a) & (cin | a);
      #0
      c1 = cin & a

initial
   cin = 1;
   a   = 1;
```

Variable values:

```
a   = 1
cin = 1
s1  = x
c1  = x
```

Variable assignment queues:

```
a   = empty
cin = empty
s1  = empty
c1  = empty
```

States of threads:

```
Thread 1: @(cin or a)
Thread 2: finished
```

Comparing VHDL's and Verilog's simulation cycle

- Verilog has global state
- Verilog does not distinguish between signals and variables
- Verilog has extra layer: `#0` and `<=`
- Verilog can be non-deterministic

Which is better: VHDL or Verilog?

	VHDL	Verilog
pros	strongly typed	simpler language
	easier refinement	more expressive language
	parametrised moduled	initialisation
	good design reusability	fast simulation
cons	too strongly typed	more complex simulation cycle
	too verbose	non-determinism
	slower design	data-types too simple

$\boxed{\text{Neither!}}$

Which should I learn: VHDL or Verilog?

- Different companies use different HDLs
- Tools availability
- Mixed language design

Both!

EE Times 6th March 2000: Verilog, VHDL groups merge

... Open Verilog International and VHDL International ... are joining forces to form a united standards body ...

... more designers are using a mix of the two languages ...

... predicted that by 2001, the HDL simulation market will largely be dominated by mixed-language simulators ...