EE E6930
Advanced Digital Integrated Circuits

Spring, 2002
Lecture 18. Introduction to Verilog-A/Verilog-AMS
References

• cdsdoc documentation on Verilog-A
• On-line course reader paper.
Introduction

• Verilog and VHDL dominate the market for digital modelling (with about equal market share). The language are functionally “equivalent” and most simulators support both (and allow you to mix them!). Both are standardized by the IEEE.

• There has been a desire to extend these languages into the analog space by supporting the solution of continuous-time differential-algebraic equations. Verilog-A (later Verilog-AMS) and VHDL-AMS were such extensions. Both are making their way through the slow standards process although Verilog-AMS has the (significant) weight of Cadence behind it.
Two domains

- Verilog-AMS captures two different simulation domains
  - Digital simulation domain
    - Discrete event simulation kernel
  - Analog simulation domain (Verilog A)
    - Differential equation solver. Solutions change continuously between individual solutions. Time-step selection depends on tolerances.
- Verilog-AMS provides mapping modules to provide translation of values and times from one domain to the other to create a true mixed-signal simulation environment.
Use of HDL

• In the digital world, HDLs are used for both simulation and synthesis.

• In the analog world, HDLs will be used primarily for simulation and perhaps synthesis is the very limited context of parameterized library elements (fixed circuit topologies).
Digital domain

- In our digital classes, we have learned VHDL. Verilog is “identical” functionality (the syntax is just different).
- Here is an AOI gate:

// Verilog code for AND-OR-INVERT gate

module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;
  assign F = ~((A&B)|(C&D));
end module;
Digital domain

• We also have the equivalent of a VHDL process statement in Verilog

```verbatim
reg F;
always @(A or B or C or D)
begin
    F = ~((A&B)|(C&D));
end
```

Just like in VHDL, to synthesize combinational logic using an always block, all variables must be assigned under all conditions and all inputs must be in the sensitivity list. (Otherwise, you will make a latch!)
Digital domain

- One can combine continuous assignments, (structural) hierarchy, and an always statement.

```
// continuous assignments
Assign selb = ~sel;
Assign fb = ~((a&sel)||(b&selb));
Assign f = ~fb;

// structural
INV G1 (sel, selb);
AOI G2 (selb, a, sel, b, fb);
INV G3 (.a(fb), .f(f));

// always block
Always @(sel or a or b)
Begin
  if (sel == 1)
    f = a;
  else
    f = b;
end
```
Creating latches

```verbatim
// Positive edge-triggered D flip-flop with reset
module d_flop(clock, reset, d, q, q_n);
    input clock, reset, d;
    output q, q_n;
    reg q;

    assign q_n = ~q;
    always @(posedge clock or posedge reset)
        if (reset == 1) q <= 0;
        else q <= d;
endmodule;
```
Verilog-A

- **Contributional operator “<+”** assigns an expression to a signal. Expression can be linear, non-linear, algebraic, or differential/integral functions of inputs signals.

```verilog
// Resistor
module res(p, n);
    inout p, n;
    electrical p, n;
    parameter real r=0 from (0:inf];

    analog
    V(p, n) <+ r*I(p, n);

    Endmodule;

parameter real res_value = 25;
res #(r(res_value)) mod(a1, a2);
```
Analog operators

- Time derivative operator

```verilog
// capacitor
module cap(p, n)
  inout p, n;
  electrical p, n;
  parameter real c=0 from [0:inf);
  analog
  I(p, n) <+ c*ddt(V(p, n));
endmodule;
```
Analog operators

- Time integral operators

```plaintext
// PID controller model with differential input and single output
module pid(in1, in2, out);
    input in1, in2;
    output out;
    electrical in1, in2, out;

    parameter real kp=1, ki=1, kd=1;

    real vin;

    analog
    begin
        vin = v(in1, in2)/2;
        v(out) <+ kp*vin + ki*idt(vin, 0) + dc*ddt(vin);
    end
endmodule
```
Analog operators

- Transition operator

module ideal_dac(in, out);
  input [0:dac_size-1] in;
  output out;
  electrical in, out;
  parameter real dac_size = 2 from (1:inf);
  parameter vth = 2.5;
  parameter real trise = 0 from [0:inf);
  parameter real tfall = 0 from [0:inf);

  real code;
  integer pow2 [0:dac_size];

  analog
  begin
    @(initial_step)
      for (i=0; i<= dac_size;i=i+1) pow2[i] = pow(2,i);
      code = 0;
      for (i = 0; i < dac_size; i=i+1)
        code = code + (V(in[i]) > vth) ? Pow2[i] : 0;
      V(out) <+ transition(code/pow2[dac_size], 0, trise, tfall);
  end
endmodule
Analog operators

- Slew operator

\[
\text{slew}(\text{expr}, \text{msr}, \text{msr})
\]

\[
\frac{\Delta y}{\Delta t} \leq \text{rate}_{\text{msr}}
\]

\[
\text{expr}(t)
\]

\[
\text{output}(t)
\]

\[
t_0
\]
Analog operator

- Delay operator

```
module delay_op(out, in);
    inout out, in;
    electrical out, in;

    analog
        V(out) <+ delay(V(in), 50n);

endmodule
```
Analog operators

• Laplace transform operators
  – laplace_zp(expr, numer, denom). Zeros and poles of the filter are specified as pairs of real numbers, specifying the real and imaginary components of each zero or pole.
  – laplace_nd(expr, numer, denom). Zeros and poles of the filter are specified as polynomial coefficients from lower order term to highest
  – laplace_zd(expr, numer, denom). Zeros are specified as pairs of real numbers. Poles are specified as polynomial coefficients.
  – laplace_np(expr, numer, denom). Zeros are specified as polynomial coefficients. Poles are specified as pairs of real numbers.
Analog operators

- Laplace transform operators

```
module laplace_op(out, in);
inout out, in;
electrical out, in;
analog
V(out) <+ laplace_np(V(in),
{ 1 }, {-0.81, 0.59, -0.81, -0.59, -0.31,
0.95, -0.31, -0.95, -1.0, 0.0});
endmodule
```
Analog operators

- **Z-transform operators**
  - `zi_zp(expr, numer, denom, T, trf, t0)`
  - `zi_nd`
  - `zi_zd`
  - `zi_np`

- **T** is the period of the filter, **trf** specifies the transition time, **t0** specifies the time of the first transition.
Analog operators

- Z transform operators

Module discrete_op(out, in);
  inout out, in;
  electrical out, in;

  analog
  V(out) <+ zi_nd(V(in), {1.0},
  {1.0}, 10u);
endmodule
Access to simulation environment

- $\text{realtime}()$
- $\text{bound\_step}()$. Sets an upper bound on the time step in simulation.

```verilog
module v_sine_generator(out);
  output out;
  voltage out;

  parameter real freq = 1K from (0:inf),
     ampl = 1,
     offset = 0;

  analog
  begin
    V(cout) <+ ampl * sin(`M_TWO_PI * freq * $\text{realtime}$) + offset;
    bound_step(0.05/freq);
  end
endmodule
```
Analog events

- Cross event

```verilog
module sample_hold (in, out, clk);
  input in, clk;
  output out;
  electrical in, out, clk;

  parameter real slewrate = 1.0e-9 from (0:inf);
  parameter real clk_vth = 2.5;

  real v;

  analog
  begin
    if (analysis("static") || (V(clk) > clk_vth))
      v = V(in); // passing phase
    @(cross(V(clk) – clk_vth, -1))
      v = V(in); // sampling phase
    V(out) <= slew(v, slewrate);
  end
endmodule
```
Analog events

- Timer event

```
timer(start, period)
```

start

period
Examples

- $\textit{discontinuity}$ function tells simulator about discontinuities, but doesn’t mean the simulator will handle them correctly!

```verilog
module v_deadband_amp(inp, inm, out);
  input inp, inm;
  output out;
  electrical inp, inm, out;

  parameter real vin_low = -1m,
          vin_high = 1m from (vin_low:inf),
          vout_dead = 0,
          gain_left = 1,
          gain_right = 1;

  real vout, vin;

  analog begin
    vin = V(inp, inm);
    @(cross(vin-vin_high)) $\textit{discontinuity}(1);
    @(cross(vin-vin_low)) $\textit{discontinuity}(1);

    if (vin >= vin_high)
      vout = gain_right*(vin-vin_high) + vout_dead;
    else
      if (vin <= vin_low)
        vout = gain_left*(vin-vin_high) + vout_dead;
      else
        vout = vout_dead;
  V(out) <+ vout;
  end
endmodule
```

```verilog
module trisource (vout);
  output vout;
  electrical vout;
  parameter read wavelength = 10.0, amplitude = 1.0;
  integer slope;
  real wstart;

  analog begin
    @(timer(0, wavelength)) begin
      slope = +1;
      wstart = $\textit{realtime};
      $\textit{discontinuity}(1);
    end
    @(timer(wavelength/2, wavelength)) begin
      slope = -1;
      wstart = $\textit{realtime};
      $\textit{discontinuity}(1);
    end
    v(vout) <+ amplitude * slope * (4 * ($\textit{realtime} – wstart) / wavelength – 1);
  end
endmodule
```
Examples

- $\text{strobe}$ lets you print during a simulation

```vhdl
module period_measure(in);
  input in;
  voltage in;

  parameter real vth = 0;

  integer crossings;
  real latest, previous;

  analog
  begin
    @(initial_step)
    begin
      crossings=0; previous = 0; latest = 0;
    end

    @(cross(V(in)-vth,+1))
    begin
      crossings = crossings+1;
      previous= latest;
      latest = $\text{realtime}$;
    end

    @(final_step)
    begin
      if (crossings < 2)
        $\text{strobe}$("Could not measure period.\n");
      else
        $\text{strobe}$("period = %g, crossings = %d\n",
          latest-previous, crossings);
      end
    end
  endmodule
```
Examples

- Indirect contributional statements

\[
\frac{d^2 x}{dt^2} = \omega_o^2 x(t)
\]

\[
\left. \frac{dx}{dt} \right|_{t=0} = \omega_o
\]

analog begin
    if (analysis("dc"))
        V(dx) <+ w0;
    else
        V(dx) <+ ddt(V(x));
    V(x) : ddt(V(dx)) == -w0*w0*V(x);
end
disciplines and natures

- These are the standard electrical discipline declarations (contained in the standard definitions file):

  ```plaintext
  // current in amperes
  nature current
  units = "A";
  access = I;
  abstol = 1e-12;
  endnature

  // potential in volts
  nature voltage
  units = "V";
  access = V;
  abstol = 1e-6;
  endnature

  discipline electrical
  potential voltage;
  flow current;
  enddiscipline;

  electrical n1, n2;
  V(n1, n2)
  I(n1, n2)
  ```