Hardware description languages

- Describe hardware at varying levels of abstraction
- Structural description
  - textual replacement for schematic
  - hierarchical composition of modules from primitives
- Behavioral/functional description
  - describe what module does, not how
  - synthesis generates circuit for module
- Simulation semantics

HDLs

- Abel (circa 1983) - developed by Data-I/O
  - targeted to programmable logic devices
  - not good for much more than state machines
- ISP (circa 1977) - research project at CMU
  - simulation, but no synthesis
- Verilog (circa 1985) - developed by Gateway (absorbed by Cadence)
  - similar to Pascal and C
  - delays is only interaction with simulator
  - fairly efficient and easy to write
  - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
  - similar to Ada (emphasis on re-use and maintainability)
  - simulation semantics visible
  - very general but verbose
  - IEEE standard

Verilog

- Supports structural and behavioral descriptions
- Structural
  - explicit structure of the circuit
  - e.g., each logic gate instantiated and connected to others
- Behavioral
  - program describes input/output behavior of circuit
  - many structural implementations could have same behavior
  - e.g., different implementation of one Boolean function
- We'll only be using behavioral Verilog in DesignWorks
  - rely on schematic when we want structural descriptions

Structural model

```
module xor_gate (out, a, b);
  input     a, b;
  output    out;
  wire      abar, bbar, t1, t2;

  inverter invA (abar, a);
  inverter invB (bbar, b);
  and_gate and1 (t1, a, bbar);
  and_gate and2 (t2, b, abar);
  or_gate  or1 (out, t1, t2);

endmodule
```

Simple behavioral model

- Continuous assignment

```
module xor_gate (out, a, b);
  input         a, b;
  output        out;
  reg           out;

  assign #6 out = a ^ b;

endmodule
```

simulation register - keeps track of value of signal

delay from input change to output change

Simple behavioral model

- always block

```
module xor_gate (out, a, b);
  input         a, b;
  output        out;
  reg           out;

  always @(a or b) begin
     #6 out = a ^ b;
  end

endmodule
```

specifies when block is executed ie. triggered by which signals

Driving a simulation

```
module stimulus (x, y);
  output       x, y;
  reg [1:0]    cnt;

  initial begin
    cnt = 0;
    repeat (4) begin
      #10 cnt = cnt + 1;
      $display ("@ time=%d, x=%b, y=%b, cnt=%b",
        $time, x, y, cnt); end
    #10 $finish;
  end

  assign x = cnt[1];
  assign y = cnt[0];
endmodule
```

2-bit vector

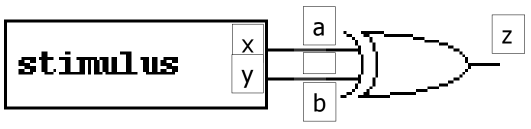
initial block executed only once at start of simulation

print to a console

directive to stop simulation

Complete Simulation

- Instantiate stimulus component and device to test in a schematic



Comparator Example

```
module Compare1 (A, B, Equal, Alarger, Blarger);
  input     A, B;
  output    Equal, Alarger, Blarger;

  assign #5 Equal = (A & B) | (~A & ~B);
  assign #3 Alarger = (A & ~B);
  assign #3 Blarger = (~A & B);
endmodule
```

More Complex Behavioral Model

```
module life (n0, n1, n2, n3, n4, n5, n6, n7, self, out);
  input     n0, n1, n2, n3, n4, n5, n6, n7, self;
  output    out;
  reg       out;
  reg [7:0] neighbors;
  reg [3:0] count;
  reg [3:0] i;

  assign neighbors = {n7, n6, n5, n4, n3, n2, n1, n0};

  always @(neighbors or self) begin
    count = 0;
    for (i = 0; i < 8; i = i+1) count = count + neighbors[i];
    out = (count == 3);
    out = out | ((self == 1) & (count == 2));
  end

endmodule
```

Hardware Description Languages vs. Programming Languages

- Program structure
  - instantiation of multiple components of the same type
  - specify interconnections between modules via schematic
  - hierarchy of modules (only leaves can be HDL in DesignWorks)
- Assignment
  - continuous assignment (logic always computes)
  - propagation delay (computation takes time)
  - timing of signals is important (when does computation have its effect)
- Data structures
  - size explicitly spelled out - no dynamic structures
  - no pointers
- Parallelism
  - hardware is naturally parallel (must support multiple threads)
  - assignments can occur in parallel (not just sequentially)

Hardware Description Languages and Combinational Logic

- Modules - specification of inputs, outputs, bidirectional, and internal signals
- Continuous assignment - a gate's output is a function of its inputs at all times (doesn't need to wait to be "called")
- Propagation delay- concept of time and delay in input affecting gate output
- Composition - connecting modules together with wires
- Hierarchy - modules encapsulate functional blocks
- Specification of don't care conditions (accomplished by setting output to “x”)

Hardware Description Languages and Sequential Logic

- Flip-flops
  - representation of clocks - timing of state changes
  - asynchronous vs. synchronous
- FSMs
  - structural view (FFs separate from combinational logic)
  - behavioral view (synthesis of sequencers)
- Data-paths = ALUs + registers
  - use of arithmetic/logical operators
  - control of storage elements
- Parallelism
  - multiple state machines running in parallel
- Sequential don't cares

Flip-flop in Verilog

- Use always block's sensitivity list to wait for clock edge

```
module dff (clk, d, q);

   input  clk, d;
   output q;
   reg    q;

   always @(posedge clk)
      q = d;

endmodule
```

More Flip-flops

- Synchronous/asynchronous reset/set
  - single thread that waits for the clock
  - three parallel threads – only one of which waits for the clock

```
module dff (clk, s, r, d, q);
    input  clk, s, r, d;
    output q;
    reg    q;

    always @(posedge clk)
        if (reset)    q = 1'b0;
        else if (set) q = 1'b1;
        else          q = d;

endmodule
```

```
module dff (clk, s, r, d, q);
    input  clk, s, r, d;
    output q;
    reg    q;

    always @(posedge reset)
        q = 1'b0;
    always @(posedge set)
        q = 1'b1;
    always @(posedge clk)
        q = d;

endmodule
```

Structural View of an FSM

- Traffic light controller: two always blocks - flip-flops separate from logic

```
module FSM (HL, FL, ST, clk, C, TS, TL);
    output [2:0] HL, FL;     reg    [2:0] HL, FL;
    output ST;               reg    ST;
    input  clk;
    input  C, TS, TL;
    reg    [1:0] present_state;
    reg    [1:0] next_state;

    initial begin HL = 3'b001; FL = 3'b100; present_state = 2'b00; end

    always @(posedge clk) // registers
        present_state = next_state;

    always @(present_state or C or TS or TL)
        // compute next-state and output logic whenever state or inputs change
        // put equations here for next_state[1:0], HL[2:0], FL[2:0], and ST
        // as functions of C, TS, TL, and present_state[1:0]
endmodule
```

Behavioral View of an FSM

- Specification of inputs, outputs, and state elements

```
module FSM(HR, HY, HG, FR, FY, FG, ST, TS, TL, C, reset, Clk);
  output     HR;
  output     HY;
  output     HG;
  output     FR;
  output     FY;
  output     FG;
  output     ST;
  input      TS;
  input      TL;
  input      C;
  input      reset;
  input      Clk;

  reg [6:1]  state;
  reg        ST;
```

```
'define highwaygreen   6'b001100
'define highwayyellow  6'b010100
'define farmroadgreen  6'b100001
'define farmroadyellow 6'b100010


assign HR = state[6];
assign HY = state[5];
assign HG = state[4];
assign FR = state[3];
assign FY = state[2];
assign FG = state[1];
```

specify state bits and codes
for each state as well as
connections to outputs

Behavioral View of an FSM (cont'd)

```
   initial begin state = 'highwaygreen; ST = 0; end

   always @(posedge Clk)
     begin
       if (reset)
         begin state = 'highwaygreen; ST = 1; end
       else
         begin
           ST = 0;
           case (state)
             'highwaygreen:
               if (TL & C) begin state = 'highwayyellow; ST = 1; end
             'highwayyellow:
               if (TS) begin state = 'farmroadgreen; ST = 1; end
             'farmroadgreen:
               if (TL | !C) begin state = 'farmroadyellow; ST = 1; end
             'farmroadyellow:
               if (TS) begin state = 'highwaygreen; ST = 1; end
           endcase
         end
     end
 endmodule
```

case statement
triggerred by
clock edge

Timer for Traffic Light Controller

- Another FSM

```
module Timer(TS, TL, ST, Clk);
  output TS;
  output TL;
  input     ST;
  input     Clk;
  integer   value;

  assign TS = (value >=  4); //  5 cycles after reset
  assign TL = (value >= 14); // 15 cycles after reset

  always @(posedge ST) value = 0; // async reset

  always @(posedge Clk) value = value + 1;

endmodule
```

Complete Traffic Light Controller

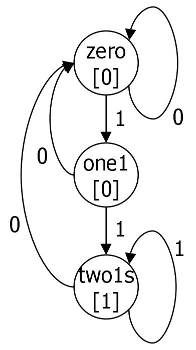
- Tying it all together (FSM + timer)

```
module main(HR, HY, HG, FR, FY, FG, reset, C, Clk);
 output HR, HY, HG, FR, FY, FG;
 input  reset, C, Clk;

  Timer part1(TS, TL, ST, Clk);
  FSM   part2(HR, HY, HG, FR, FY, FG, ST, TS, TL, C, reset, Clk);
endmodule
```

Verilog FSM - Reduce 1s example

- Moore machine

```
‘define zero  0
‘define one1  1        ← state assignment
‘define two1s 2

module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg [2:1] state;       // state variables
  reg [2:1] next_state;

  always @(posedge clk)
    if (reset) state = ‘zero;
    else       state = next_state;
```



Moore Verilog FSM (cont’d)

```
always @(in or state)

  case (state)
    ‘zero:
  // last input was a zero
   begin
     if (in) next_state = ‘one1;
     else    next_state = ‘zero;
   end
    ‘one1:
  // we’ve seen one 1
   begin
     if (in) next_state = ‘two1s;
     else    next_state = ‘zero;
   end
    ‘two1s:
  // we’ve seen at least 2 ones
   begin
     if (in) next_state = ‘two1s;
     else    next_state = ‘zero;
   end
  endcase
```

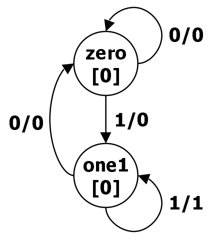
← crucial to include all signals that are input to state and output equations

note that output only depends on state ↓

```
always @(state)
  case (state)
    ‘zero: out = 0;
    ‘one1: out = 0;
   ‘two1s: out = 1;
  endcase

endmodule
```

Mealy Verilog FSM

```
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  ‘register state;      // state variables
  reg next_state;

  always @(posedge clk)
    if (reset) state = ‘zero;
    else       state = next_state;

  always @(in or state)
    case (state)
      ‘zero:            // last input was a zero
     begin
       out = 0;
       if (in) next_state = ‘one;
       else    next_state = ‘zero;
     end
      ‘one:             // we’ve seen one 1
     if (in) begin
        next_state = ‘one; out = 1;
     end else begin
        next_state = ‘zero; out = 0;
     end
    endcase
endmodule
```



Synchronous Mealy Machine

```
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg state; // state variables

  always @(posedge clk)
    if (reset) state = ‘zero;
    else
     case (state)
      ‘zero:     // last input was a zero
     begin
       out = 0;
       if (in) state = ‘one;
       else    state = ‘zero;
     end
      ‘one:      // we’ve seen one 1
     if (in) begin
        state = ‘one; out = 1;
     end else begin
        state = ‘zero; out = 0;
     end
    endcase
endmodule
```