Hardware Description Languages

- Basic idea:
  - Language constructs describe circuits with two basic forms:
  - Structural descriptions: connections of components (gates & flip-flops). Nearly one-to-one correspondence with schematic diagram (circuit structure).
  - Behavioral descriptions: use statements (assignments and tests of logical conditions) to describe the relationships between inputs and outputs (circuit function).

"Structural" example:

```verilog
Decoder(output x0, x1, x2, x3; inputs a, b) {
  wire a_L, b_L;
  inv(b_L, b);
  inv(a_L, a);
  and(x0, a_L, b_L);
  and(x1, a_L, b);
  and(x2, a, b_L);
  and(x3, a, b);
}
```

"Behavioral" example:

```verilog
Decoder(output x0, x1, x2, x3; inputs a, b) {
  case [a b]
    00: [x0 x1 x2 x3] = 0x1;
    01: [x0 x1 x2 x3] = 0x2;
    10: [x0 x1 x2 x3] = 0x4;
    11: [x0 x1 x2 x3] = 0x8;
  endcase;
}
```
Verilog Concurrent Statements

- Concurrent statements specify digital logic operation, from which a realization is synthesized; 3 common types:

  1. Instance statement
     - Instantiates a module, used in *structural* descriptions
     - Similar to a constructor call in OO languages (C++, Java, ...)

  2. Continuous assignment statement
     - For *behavioral* descriptions of *combinational* circuits

  3. *always* blocks (non-continuous assignments)
     - For *behavioral* descriptions of synchronous *sequential* circuits

- Concurrent statements “execute” *simultaneously* and *continuously*
  - Modeling the continuous operation of hardware where connected elements affect each other continuously, not just at particular, ordered time steps

Verilog Built-in Gates

- Built-in gate names are reserved words
  - *and*, *nand*, *or*, *nor* ~ any number of inputs per gate
  - *xor*, *xnor*
  - *buf* = 1-input noninverting buffer
  - *not* = inverter
  - *bufif0*, *bufif1* = 1-input buffer w/ tri-state out
  - *notif0*, *notif1* = inverter w/ tri-state outputs

- Other predefined components include
  - AND-OR-INVERT (sum-of-products) gates
  - flip-flops, decoders, multiplexers, ...
Verilog Instance Statement

- Two formats of instance statement:
  \[
  \text{component-name instance-identifier ( expr, expr, \ldots, expr );}
  \]
  \[
  \text{component-name instance-identifier ( .port-name(expr),
  .port-name(expr),
  \ldots
  .port-name(expr) );}
  \]
  - Multiple instances of the same component/workunit distinguished by unique names ("instance-identifier")

- The 1st format depends on the order in which port names appear in the original component/workunit definition
  - Expressions listed in the same order as ports to which they connect
  - For built-in gates, the defined port order is \((\text{output, input, input, \ldots})\)
    - The order among the multiple inputs doesn’t matter
    - For built-in three-state buffers and inverters, the defined order is \((\text{output, data-input, enable-input})\)

- The 2nd format explicitly names the ports
  - Recommended because it helps avoid mistakes in coding

Structural Model - XOR example

- Notes:
  - The instantiated gates are not "executed". They are \textit{always active}.
  - XOR gate already exists as a built-in (only an example – no need to define it)
  - Undeclared variables are assumed to be \\textit{wires}. Don’t let this happen to you!
### Structural Example: 2-to-1 Mux

**C/C++ style comments**
- Built-ins don't need instance names
- Multiple instances can share the same "master" name

/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
  input in0, in1, select;
  output out;
  wire s0, w0, w1;
  not (s0, select);
  and (w0, s0, in0),
      (w1, select, in1);
  or (out, w0, w1);
endmodule // mux2

/* 2-input multiplexor in gates */
module mux4 (in0, in1, in2, in3, select, out);
  input in0, in1, in2, in3;
  input [1:0] select;
  output out;
  wire w0, w1;
  module mux2
    .select(select[0]),
    .in0(in0),
    .in1(in1),
    .out(w0)),
  m_hi
    .select(select[0]),
    .in0(in2),
    .in1(in3),
    .out(w1)),
  m_fin
    .select(select[1]),
    .in0(w0),
    .in1(w1),
    .out(out);
endmodule // mux4

### Instantiation, Signal Array, Named Ports

**C/C++ style comments**
- Signal array: Declares select[1], select[0]
- Named ports. Highly recommended.

module mux4 (in0, in1, in2, in3, select, out);
  input in0, in1, in2, in3;
  input [1:0] select;
  output out;
  wire w0, w1;
  module mux2
    .select(select[0]),
    .in0(in0),
    .in1(in1),
    .out(w0)),
  m_hi
    .select(select[0]),
    .in0(in2),
    .in1(in3),
    .out(w1)),
  m_fin
    .select(select[1]),
    .in0(w0),
    .in1(w1),
    .out(out);
endmodule // mux4
Parameterized Module

- Parameterize structural modules to handle inputs and outputs of any width
- Example: 3-input majority function
  - Outputs “1” if at least two inputs are “1”
    \[ \text{OUT} = I0 \cdot I1 + I1 \cdot I2 + I2 \cdot I0 \]

```verilog
module Maj(out, i0, i1, i2);
  parameter WID = 1;
  input [WID-1:0] i0, i1, i2;
  output [WID-1:0] out;
  assign out = i0 & i1 | i1 & i2 | i2 & i0;
endmodule
```
- When `Maj` module instantiated using regular syntax, the parameter `WID` takes on default value 1
- Instance statement allows parameter substitution using #
  - Example: `X`, `Y`, `Z` are 8-bit input vectors, the 8-bit majority function:
    ```verilog
    Maj #(8) U1 (.OUT(W), .I0(X), .I1(Y), .I2(Z));
    ```

Simple Behavioral Model

- Shorthand for explicit instantiation of AND gate (in this case).
- The assignment happens *continuously* (modeling the continuous operation of hardware);
  therefore, any change on the right-hand-side (RHS) signals is reflected immediately on the output port (except for the small delay associated with the implementation of the “&” operation).
- Different from assignment in C that takes place when the program counter reaches that place in the program.
Example: Ripple Adder

_module FullAdder(a, b, ci, r, co);
  input a, b, ci;
  output r, co;
  assign r = a ^ b ^ ci;
  assign co = a&ci | a&b | b&cin;
_endmodule

_module Adder(A, B, R);
  input [3:0] A;
  input [3:0] B;
  output [4:0] R;
  wire c1, c2, c3;
  FullAdder
    add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1), .r(R[0]) ),
    add1(.a(A[1]), .b(B[1]), .ci(c1), .co(c2), .r(R[1]) ),
    add2(.a(A[2]), .b(B[2]), .ci(c2), .co(c3), .r(R[2]) ),
    add3(.a(A[3]), .b(B[3]), .ci(c3), .co(R[4]), .r(R[3]) );
_endmodule

Continuous Assignment Statements

- **assign** net-name = expression;
- **assign** net-name[bit-indx] = expression;
- **assign** net-name[msb:lsb] = expression;
- **assign** net-concatenation = expression;

Continuous-assignment statements are evaluated **continuously** (because hardware elements affect each other continuously, not just at particular, ordered time steps)

- The **order** of continuous assignment statements in a module doesn’t matter
- Continuous-assignment statement is unconditional, but different values can be assigned using the **conditional operator** (? : )
Continuous Assignment Examples

- assign R = X | (Y & ~Z);
- assign r = &X;
- assign R = (a == 1'b0) ? X : Y;
- assign P = 8'hff;
- assign P = X * Y;
- assign P[7:0] = {4{X[3]}, X[3:0]};
- assign {cout, R} = X + Y + cin;
- assign Y = A << 2;
- assign Y = {A[1], A[0], 1'b0, 1'b0};

Non-continuous Assignments

always blocks and procedural code

- Syntax of Verilog always blocks
  - always @ (signal-name or ... or signal-name) procedural-statement
  - always procedural-statement

- Procedural statements in an always block execute sequentially, as in software program
  - However, always blocks execute concurrently with other concurrent statements in the module
- Note: assign statements must be used outside always statements; both are evaluated concurrently
Procedural Sensitivity Lists

- The execution of a statements within a procedure can be controlled using an event-control sensitivity list
  - An always procedure must re-evaluate the outputs whenever an "input" changes value
    - An "input" is any signal used to determine the value of assignments
- Procedures automatically become active at time zero
- Execution of statements is delayed until a change occurs on a signal in the "sensitivity list"

```
always @ ( <edge> <signal> or <edge> <signal> )
```

- `<edge>` may be posedge (positive) or negedge (negative)
  - If no edge is specified, then any transition is used
- Sensitivity to multiple signals is specified using an “or” separated list

always Block Example (1)

- Sensitivity list signals @(...) determine when the always block executes
  - The block is initially suspended and starts executing when any signal in the sensitivity list changes its value
  - This continues until the block executes without any sensitivity-list signal changing its value

```module and_or_gate (out, in1, in2, in3);
  input in1, in2, in3;
  output out;
  reg out;

  always @ (in1 or in3)
  begin
    out = (in1 & in2) | in3;
  end
endmodule```

“begin-end block” brackets multiple procedural statements (not necessary in this example)
A Combinational Logic Sensitivity

- **Verilog features a “wildcard” token to indicate a combinational logic sensitivity list**
  - The `@*` token is a time control which indicates that the control is automatically sensitive to any change on any “input” to the statement or group-of-statements that follows
    - An “input” is any signal whose value is read by the statement or statement group
- **SystemVerilog introduced `always_comb` for modeling combinational logic**
  - The simulator infers the sensitivity list to be all variables from the contained statements

```verilog
always @(sel or a or b or c or d)
case (sel)
  2'b00: y = a;
  2'b01: y = b;
  2'b10: y = c;
  2'b11: y = d;
endcase
```

**Note:** `case` statement is defined later.

Verilog Procedural Statements

- **Blocking assignment:** `variable-name = expression;`
  - Evaluates the expression immediately and assigns to variable
- **Nonblocking assignment:** `variable-name <= expression;`
  - Evaluates the expression immediately but does NOT assign to variable until an infinitesimal delay after the `always` block has completed execution
- **begin-end block**
  - Encloses a list of procedural statements that execute sequentially
- **if statement**
  - A `condition` (logical expression) is tested; if true the enclosed statement is executed
- **case statement**
  - A “selection expression” followed by a list of “choices” and corresponding procedural statements
- **Looping statements:** `for, while, repeat`
  - Execute the enclosed procedural statements for a given number of iterations
Blocking vs. Nonblocking Statements

- **Blocking assignment**: 
  \[ \text{variable-name = expression;} \]
  - “immediate assignment” or within a specifiable delay
  - Evaluates the expression immediately and assigns to variable
  - Use **blocking** assignments to create **combinational** logic

- **Nonblocking assignment**: 
  \[ \text{variable-name <<= expression;} \]
  - “nonblocking and slightly deferred assignment” or “late assignment”
  - Evaluates the expression immediately but does NOT assign to variable until an infinitesimal delay after the \texttt{always} block has completed execution
  - Use **nonblocking** assignments to create **sequential** logic

---

**always Block Example (2)**

- **Sensitivity list** signals \@(...) determine when the \texttt{always} block executes
  - For example, the flip-flop includes only \texttt{clk} in the sensitivity list
  - Flip-flop remembers its old value of \texttt{q} until the next rising edge of the \texttt{clk}, even if \texttt{d} changes in the interim
  - In contrast, continuous assignment statements (\texttt{assign}) are reevaluated anytime any of the inputs on the right hand side changes
    therefore, such code necessarily describes combinational logic

```systemverilog
module register ( //a vector of flip-flops
    input logic clk,
    input logic [3:0] d;
    output logic [3:0] q );

always_ff @ (posedge clk)
    q <= d;
endmodule
```

- **SystemVerilog introduced** \texttt{always_ff, always_latch, and always_comb} (seen above) to imply flip-flops, latches, or combinational logic
- This reduces the risk of common errors
### if statement

- **A condition** (logical expression) is tested; if true the enclosed procedural statement is executed
- **Nested if-else example:**

  ```verilog
  module mux4 (in0, in1, in2, in3, select, out);
  input in0, in1, in2, in3;
  input [1:0] select;
  output out;
  reg out;

  always @ (in0 in1 in2 in3 select)
    if (select == 2'b00) out=in0;
    else if (select == 2'b01) out=in1;
    else if (select == 2'b10) out=in2;
    else out=in3;

  endmodule // mux4
  ```

  - Nested if structure leads to “priority logic” structure, with different delays for different inputs (in3 to out delay > than in0 to out delay).

### case statement

- **Evaluates the “selection expression,” finds the first “choice” that matches the expression’s value and executes the corresponding procedural statement**
- **case statement example:**

  ```verilog
  module mux4 (in0, in1, in2, in3, select, out);
  input in0, in1, in2, in3;
  input [1:0] select;
  output out;
  reg out;

  always @ (in0 in1 in2 in3 select)
    case (select)
      2'b00: out=in0;
      2'b01: out=in1;
      2'b10: out=in2;
      2'b11: out=in3;
    endcase

  endmodule // mux4
  ```

- Recall that we could use a “wildcard” token `*` to indicate a combinational logic sensitivity list or `always_comb` in SystemVerilog.
Incomplete case statement

- Listed choices may not be “all inclusive”—some possible values of the selection expression may be missing
- Incomplete case statement example:

```verilog
module mux3 (in0, in1, in2, select, out);
  input in0, in1, in2;
  input [1:0] select;
  output reg out;
  
  always @(in0 in1 in2 select)
  case (select)
    2'b00: out = in0;
    2'b01: out = in1;
    2'b10: out = in2;
  endcase
endmodule // mux3
```

If sel = 2'b11 = 3, mux will output the previous value! —inferred an unwanted latch

Inferring an unwanted latch can be prevented with a default statement:

```verilog
default: out = 1'bx;
```

for looping statement

- Syntax of a Verilog for statement:

```verilog
for (loop-index = first-expr; logical expression; loop-idx = next-expr )
  procedural-statement
```

- for statement example — prime-number detector:

```verilog
module Vprimebv (input [15:0] N, output reg F);
  reg prime;
  integer i;

  always @ (N) begin
    prime = 1; // initial value
    if ((N=1) || (N==2)) prime = 1; // special cases
    else if ((N % 2) == 0) prime = 0; // even, not prime
    else for (i = 3; i <= 255; i = i+2) 
      if ( (N % i) == 0 ) && (N != i) prime = 0; // set to 0 if N is divisible by any i
    if (prime==i) F = 1; else F = 0;
  end
endmodule // Vprimebv
```
Behavioral vs. Structural

- Rule of thumb:
  - Behavioral doesn’t have sub-components
  - Structural has sub-components:
    - Instantiated Modules
    - Instantiated Gates
    - Instantiated Primitives
- Most levels are mixed

Behavioral Example

- Behavioral only
- No instantiations
Behavioral and Structural

- Behavioral:
  - Adder function
  - Register function

- Structural:
  - Top module
  - Two instantiations

Structural Low-level Details
Design Strategy

- Generally, complex systems are designed *hierarchically*
- The overall system is described *structurally* by instantiating its major components ("subsystems")
- Each subsystem is described structurally from its building blocks …
- Continued recursively until pieces are simple enough to describe *behaviorally*
- Recommended to avoid (at least minimize) mixing structural and behavioral descriptions within a single module