Behavioural Models

- Continuous Assignment Statements
- Arithmetic Operators
- Decoders

Continuous Assignment Statement

```
module AOI_5(output y_out, input a, b, c, d, e);
  assign y_out = ~((a & b) | (c & d & e));
endmodule
```

assign declares a continuous assignment. Whenever any of the variables on the RHS change (an event), the expression is re-evaluated to update the LHS. The assignment is ‘sensitive’ to the variables on the RHS.

```
module AOI_5(output y_out, input a, b, c, d, e, enable);
  assign y_out = enable ? ~((a & b) | (c & d & e)) : 1’bz;
endmodule
```

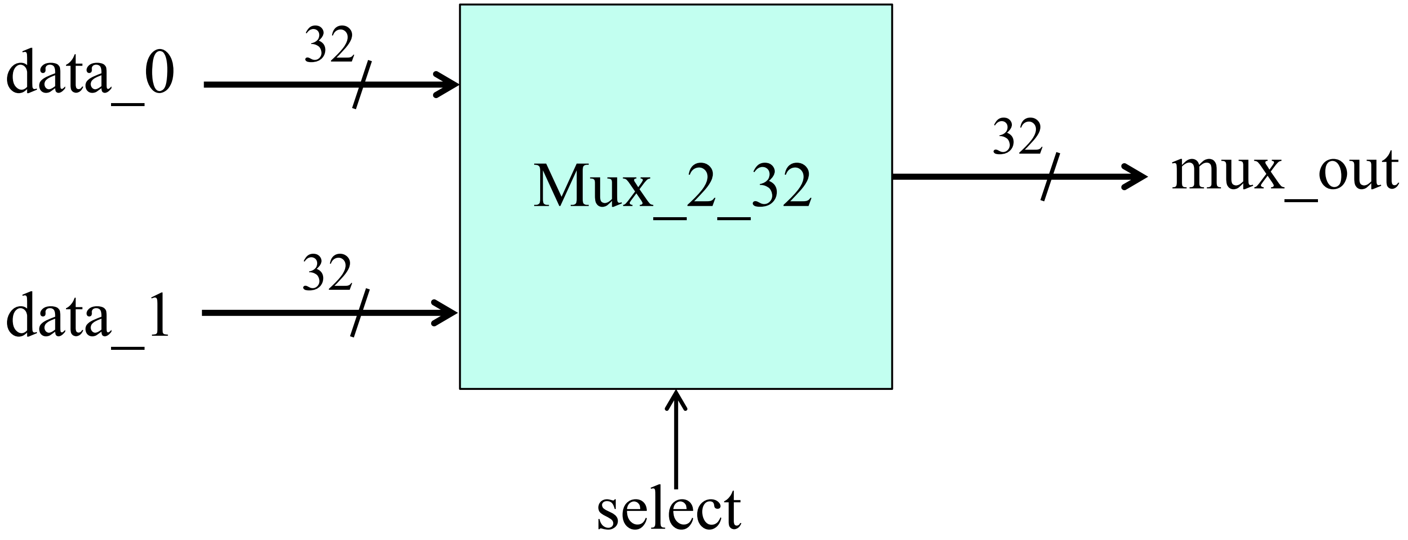
The conditional operator (? :) can be used to add an enable.
`?` lets you switch between the two expressions either side of **`:`**
If enable is asserted (true) the expression to the left of the colon is evaluated, otherwise the expression to the right of the : is evaluated.

Concurrency

- A module may contain multiple continuous assignments
- Assignments are active concurrently with all other continuous assignments, primitives, instantiated modules, behavioral statements
- Continuous assignments can also be written without the **assign** keyword as part of a wire declaration
- Propagation (inertial) delay can be included in a continuous assignment

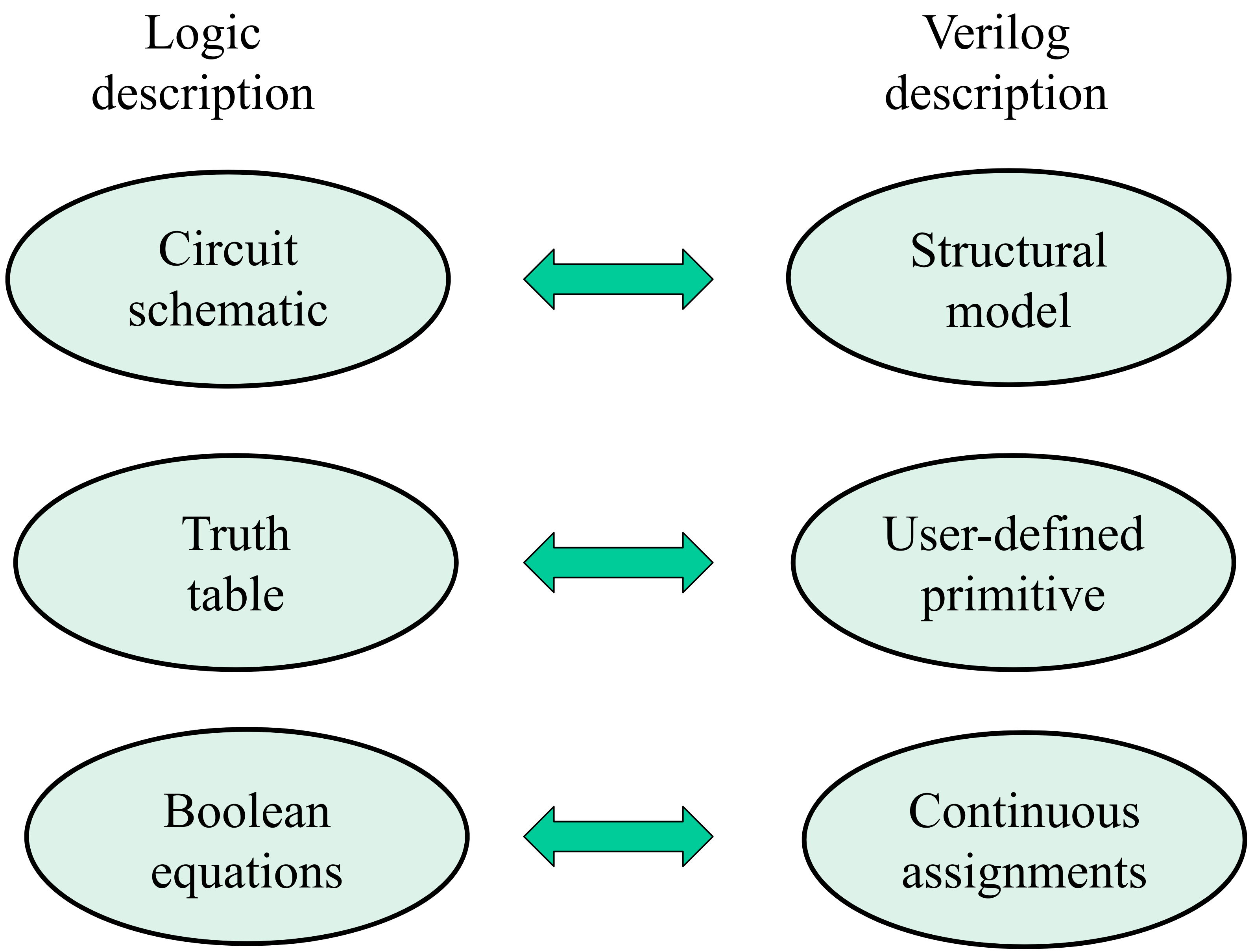
```
module AOI_5(output y_out, input a, b, c, d);
  wire #1 y1 = a & b;
  wire #1 y2 = c & d;
  wire #1 y_out = ~(y1 | y2);
endmodule
```

Multiplexer with Continuous Assignment



```
module Mux_2_32 #(parameter  word_size  = 32) (
  output  [word_size - 1:0]     mux_out;
  input    [word_size - 1:0]     data_0, data_1;
  input                                        select
);
  assign mux_out = select ? data_1 : data_0
endmodule
```

a parameter is a constant value defined within the module


Logic description
Verilog description
Circuit schematic
Structural model
Truth table
User-defined primitive
Boolean equations
Continuous assignments

4-Bit Adder

- 4-bit adder adds two 4-bit binary inputs A and B, sets 4-bit output S
- Could describe structurally
  - Carry-ripple: 4 full-adders
- Behaviorally
  - Simply: S = A + B
  - "always" procedure sensitive to A and B
    - Adder is combinational – must include all inputs in sensitivity list
    - Note: procedure resumes if *any* bit in either vector changes

```
module Add4(A, B, S);

  input [3:0] A, B;
  output [3:0] S;
  reg [3:0] S;

  always @(A, B) begin
    S = A + B;
  end
endmodule
```

Built-in arithmetic operators include:

+ : addition	- : subtraction
* : multiplication	/ : division
% : modulus	** : power (“a ** b” is a raised to the power of b)

4-Bit Adder with Carry-In and Carry-Out

- Adders have carry-in and carry-out bits
- S = A + B + Ci
  - Yields correct sum
    - "+" operator handles different bit-widths – extends Ci to 4 bits, padded on left with 0s
  - But carry-out?
    - S is only 4 bits; Co is a fifth bit
- Solution – Do 5-bit add, separate fifth bit (carry-out) from lower four

```
module Add4wCarry(A, B, Ci, S, Co);
  input [3:0] A, B;
  input Ci;
  output [3:0] S;
  reg [3:0] S;
  output Co;
  reg Co;
  reg [4:0] A5, B5, S5;

  always @(A, B, Ci) begin
    A5 = {1'b0, A};
    B5 = {1'b0, B};
    S5 = A5 + B5 + Ci;
    S = S5[3:0];
    Co = S5[4];
  end
endmodule
```

- Concatenation Operator "{ }" Joins bits from two or more expressions
- Expressions separated by commas within { }

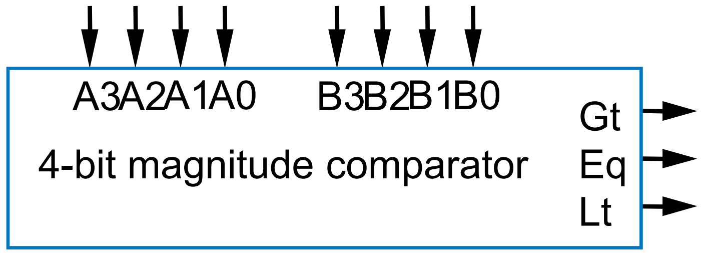
alternative description :

```
always @(A, B, Ci) begin
   {Co, S} = A + B + Ci;
end
endmodule
```

- Use concatenation on the left side of assignment
  - {Co, S} = A + B + Ci
  - Left side thus 5 bits wide
- Rule
  - For the + operator, all operands extended to width of widest operand, including left side
  - Left side is 5 bits → A, B, and Ci all extended to 5 bits, left padded with 0s
  - e.g., A: 0011, B: 0001, Ci: 0 → 00011+00001+00000 yields 00100
    - Co gets first 0, S gets 0100
- Though longer, previous description synthesizes to same circuit
  - reg [4:0] A5, B5, S5; – Synthesize into wires

4-bit Unsigned/Signed Magnitude Comparator

- Previously
  - Dealt only with unsigned numbers
    - input, output, reg declarations are unsigned unless otherwise specified
- Now consider a simple magnitude comparator that compares a 4-bit unsigned number A with a 4-bit signed number B, with outputs for greater than, less than, and equal
  - A can be 0 to 15 (0000 to 1111)
  - B can be -8 to 7 (1000 to 0111)
  - Need to represent both unsigned and signed numbers



Declare A input as before, but declare B input with signed keyword.
When comparing A and B using "<", first convert unsigned A to signed value using $signed system function.

```
module Comp4(A, B, Gt, Eq, Lt);

  input [3:0] A;
  input signed [3:0] B;
  output Gt, Eq, Lt;
  reg Gt, Eq, Lt;

  always @(A, B) begin
    if ($signed({1'b0,A}) < B) begin
      Gt = 0; Eq = 0; Lt = 1;
    end
    else if ($signed({1'b0,A}) > B) begin
      Gt = 1; Eq = 0; Lt = 0;
    end
    else begin
      Gt = 0; Eq = 1; Lt = 0;
    end
  end
endmodule
```

- "$signed(A)" would not work – changes positive number to negative
  - e.g., 1000 would change from meaning 8 to meaning -8
- Instead, first extend A to five bits
  - {1'b0,A} – e.g., 1000 becomes 01000
- Then convert to signed
  - $signed({1'b0,A}) – e.g., 01000 as 5-bit signed number is still 8 (due to 0 in highest-order bit)
- Operands of "<" automatically sign-extended to widest operand's width
  - So B extended to 5-bits with sign bit preserved
- Comparison is thus correct

```
module Testbench();

   reg [3:0] A_s;
   reg signed [3:0] B_s;
   wire Gt_s, Eq_s, Lt_s;

   Comp4 CompToTest(A_s, B_s, Gt_s, Eq_s, Lt_s);

   initial begin
      A_s = 4'b0011; B_s = 4'b0001;
      #10 A_s = 4'b1111; B_s = 4'b0111;
      #10 A_s = 4'b0111; B_s = 4'b1011;
      #10 A_s = 4'b0001; B_s = 4'b0010;
      #10 A_s = 4'b0001; B_s = 4'b0001;
      #10 A_s = 4'b0000; B_s = 4'b1111;
      #10 A_s = 4'd1; B_s = -4'd1;
      #10 A_s = 4'd1; B_s = -4'd8;
endmodule
```

- Testbench should test multiple values for inputs A and B
  - Should perform comparisons for both positive and negative values of B
  - Should have at least one test case in which A is greater than, less than, and equal to B
- Note that reg variable B_s, used to connect with B, defined as signed
- Vectors illustrate use of binary constants as well as decimal constants
  - Negative binary constant achieved using 1 in high-order bit (two-'s complement form)
  - Negative decimal constant requires negative sign "-" in front of constant

- Simulation
  - First two vectors compare positive values for both inputs
    - 0011 > 0001 → Gt_s = 1
    - 1111 > 0111 → Gt_s = 1
  - Third test compares A with negative B
    - 0111 > 1011 → Gt_s = 1
      - 7 > -5
  - Fourth and fifth test should result in the Lt_s and Eq_s output asserted, respectively
    - 0001 < 0010 → Lt_s = 1
    - 0001 = 0001 → Eq_s = 1
  - Next test compares 0 to -1
    - 0000 > 1111 → Gt_s = 1
  - Next test compare 1 to -1
  - Last test compares 1 to -8

```
A_s = 4'b0011; B_s = 4'b0001;
#10 A_s = 4'b1111; B_s = 4'b0111;
#10 A_s = 4'b0111; B_s = 4'b1011;
#10 A_s = 4'b0001; B_s = 4'b0010;
#10 A_s = 4'b0001; B_s = 4'b0001;
#10 A_s = 4'b0000; B_s = 4'b1111;
#10 A_s = 4'd1; B_s = -4'd1;
#10 A_s = 4'd1; B_s = -4'd8;
```

- Unintentional use of one of many of Verilog's automatic conversions
  - B_s <= -4'd15
  - -4d'15
    - 4-bit decimal 15 would be 1111
    - Negative of 1111 (15) is 10001 (-15) – Automatically converted to 5 bits
    - Assignment to B_s drops the high-order bit, making B_s=0001
  - Many similar types of automatic conversions in Verilog
  - Use great caution

Decoders

3-to-8 decoder
assign

```
module decode3to8 (output [7:0] y, input [2:0]a);
assign y[0] = ~a[2] & ~a[1] & ~a[0];
assign y[1] = ~a[2] & ~a[1] &  a[0];
assign y[2] = ~a[2] &  a[1] & ~a[0];
assign y[3] = ~a[2] &  a[1] &  a[0];
assign y[4] =  a[2] & ~a[1] & ~a[0];
assign y[5] =  a[2] & ~a[1] &  a[0];
assign y[6] =  a[2] &  a[1] & ~a[0];
assign y[7] =  a[2] &  a[1] &  a[0];
endmodule
```

3-to-8 decoder
for loop

```
module decode3to8 (output reg [7:0] y, input [2:0]a);
integer i;
always @ (*)
  for (i = 0; i <= 7; i = i + 1)
    if  (a == i)
      y[i] = 1;
    else
      y[i] = 0;
endmodule
```