* About Verilog
  + Developed in 1983 by Phil Moorby and Prabhu Goel
  + Loosely typed, case sensitive
  + Comprehensive, i.e., no user extensions
  + Net type determines how it's resolved if it has multiple drivers
  + Several transistor and gate level primitives
  + Construct for user defined primitives
  + (Programmable Logic Array) PLA  modelling mechanism for FPGAs
  + Propagation delay for nets and primitives
  + Drive strength for continuous assignments and primitives
  + Path delays, pulse filtering and timing checks for modules

* Full Adder example (behavioural):  
  module fullAdder (S, Cy, A, B, C);  
    input A, B, C;  
    output S, Cy;  
    assign S = A ^ B ^ C;  
    assign Cy = (A & B) | (B & C) | (C & A);  
  endmodule

* In terms of truth table:  
  primitive Carry (Cy, A, B, C);  
    input A, B, C;  
    ouput Cy;  
    table  
    //   A     B     C          Cy  
          1      1     ?    :       1;  
          1      ?     1    :       1;  
          ?      1     1    :       1;  
          0      0     ?    :       1;  
          0      ?     0    :       1;  
          ?      0     0    :       1;  
    endtable  
  endprimitive

* 4 bit carry lookahead adder

//structural representation of sum and carry (with gates)  
module sum (sum, a, b, cy\_in);  
  input a, b, cy\_in;  
  output sum;  
  wire t;  
  xor x1 (t, a, b);  
  xor x2 (sum, t, cy\_in);  
endmodule  
  
module carry (cy\_out, a, b, cy\_in);  
  input a, b, cy\_in;  
  output cy\_out;  
  wire t1, t2, t3;  
  and g1 (t1, a, b);  
  and g2 (t2, a, cy\_in);  
  and g3 (t3, b, cy\_in);  
  or g4 (cy\_out, t1, t2, t3);  
endmodule

//behavioural

module add (cy\_out, sum, a, b, cy\_in);  
   input a, b, cy\_in;  
   output sum, cy\_out,;  
   sum s1 (sum, a, b, cy\_in);  
   carry c1 (cy\_out, a, b, cy\_in);  
endmodule

module add4 (s, cy4, cy\_in, x, y);  
   input [3:0] x, y;  
   input cy\_in;  
   output [3:0] s;  
   output cy4;  
   wire [2:0] cy\_out;  
   add B0 (cy\_out[0], s[0], x[0], y[0], ci);  
   add B1 (cy\_out[1], s[1], x[1], y[1], cy\_out[0]);  
   add B2 (cy\_out[2], s[2], x[2], y[2], cy\_out[1]);  
   add B3 (cy\_out[3], s[3], x[3], y[3], cy\_out[2]);  
endmodule

* Partial physical desciption for 4 bit adder:  
  module add4;  
    input x[3:0], y[3:0], cy\_in;  
    output s[3:0], cy4;  
    boundary [0, 0, 130, 150];  
    port x[0] aluminum width = 1 origin = [0,35];  
    port y[0] aluminum width = 1 origin = [0,85];  
    port cy\_in polysilicon width = 2 origin = [70, 0];  
    port s[0] aluminum width = 1 origin = [120, 65];  
      
    add a0 origin = [0, 0];  
    add a1 origin = [0, 120];  
  endmodule
* Desciption of HW as set of modules
* Can *instantiate* one module in another (creates a copy)
* 2 ways of specifying module:
  + Behavioral representation: behavior, program-like (starting point). For modelling sequential and parallel behaviour
  + Structural representation : internal logic structure. HW component hierarchy and SW subroutine hierarchy
* Modules interconnected by nets (like wires)
* What we can do with Verilog code
  + Simulate system and verify the operation
    - Use *test benches* to provide input and verify output
  + Use synthesis tool to map to HW
    - Converts it into netlist of low-level primitives
    - HW could be ASIC or FPGA
    - Don’t need testbenches, can directly test using real signals
    - ASIC: high performance, high packing density, used in large numbers, expensive  
      FPGA: fast turnaround time, flexible, performance trade-off
* **initial** : block executed only once
* $**monitor** : like printer (prints whenever any of the variables in brackets change)
* %b : bit
* **$dumpfile:** dump testbench changes to a vcd (value change dump) file
* **$dumpvars (0, testbenchModule):** All variables to be dumped.
* Test Bench
  + the variables in the initial block that appear on LHS (ie, whose values we are changing) need to be declared as reg
  + Other variables to be declared as wire
* Icarus verilog
  + Iverilog -o <output\_file> source.v testbench.v  
    vvp <output\_file>  //runs the simulation as prints results
* Module ~ Function
  + Basic unit of HW
  + Can't contain definition of other modules
  + Other modules can be instantiated inside a module
    - A copy of the instantiated module is embedded in the main(instantiating) module for each instatntiation
    - Allows creation of hierarchy
* module module\_name (list\_of\_ports);  
    input/output declarations  
    local net declarations  
    assign statements  
    parallel statements // concurrent (as it's HW)  
  endmodule
* Eg:
  + module simpleand (f,x,y);  
      input x,y;  
      output f;  
      assign f = x & y;  
    endmodule  
      
    //Final gate netlist depends on synthesis tool, may implement above as an AND gate, or NAND + NOT
  + // 2-level combinational circuit  
    module two\_level (a,b,c,d,f);  
      input a,b,c,d;  
      output f;  
      wire t1,t2; //intermediate nets  
      assign t1 = a & b;  
      assign t2 = ~(c | d);  
      assign f = ~(t1 & t2);  
    endmodule
* Read SystemVerilog page

tri - can be used to emphasize multiple drivers are present

* Verilog supports 4 value levels
  + 0
  + 1
  + x : unknown (all registers initialized to x)
  + z : high impedance (all unconnected nets)

* Signal strengths (in descending order)

|  |  |
| --- | --- |
| **Strength** | **Type** |
| supply | Driving |
| strong | Driving |
| pull | Driving |
| large | Storage |
| weak | Driving |
| medium | Storage |
| small | Storage |
| highz | High Impedance |

* + If 2 signals get driven on a wire, stronger signal prevails
  + useful for MOS level circuits eg: dynamic MOS
* **Data types in Verilog**
  + **Net**
    - Must be continuously driven
    - Can be driven anywhere outside procedures
    - Can't store a value
    - Represents connection b/w HW elements
    - Recipient of the driver's value
    - During synthesis always maps to a wire
    - 1 bit value by default
    - Default value is 'z' (high impedance state)
      * Not driven, neither 1 nor 0
    - Typically used to model connections b/w continuous assignments and instantiations
    - Different types:
      * wire, wor, wand, tri, supply0, supply1, etc.
      * *wire* and *tri* are equivalent
        + Wire can only be driven using assign statements
      * *wor* and *wand* insert OR and AND respectively at the connection
      * *supply0* and *supply1* model power supply connections
    - Eg
      * module use\_wire (A,B,C,D,f);  
          input A,B,C,D;  
          output f;  
          wire f;  
          assign f = A & B;  
          assign f = C | D;  
          //f will be indeterminate  
        endmodule
      * module use\_wand (A,B,C,D,f);  
          input A,B,C,D;  
          output f;  
          wand f;  //implied AND, like connecting output of 2 CMOS gates  
          assign f = A & B;  
          assign f = C | D;  
          // now f is (A & B) & (C | D)  
        endmodule
      * module using\_supply\_wire (A,B,C,f);  
          input A,B,C;  
          output f;  
          supply0 gnd;  
          supply1 vdd;  
          nand G1 (t1,vdd,A,B);  
          xor  G2 (t2,C,gnd);  
          and G3 (f,t1,t2);  
        endmodule
  + **Variables**
    - Can only be driven in a procedures
    - 4 Variable types:
      * Register
      * Integer
      * Time
      * Real

* **Register**
  + Retains/holds last value assigned
  + Typically used to represent storage elements, sometimes can translate to combinational circuits also
  + X = A + B // no 'assign'
  + During synthesis, maps either to 'wire' or a 'storage cell' depending on the context of the assignment
  + Not to be confused with a hardware register, ie flip-flop
  + Register types in Verilog

|  |  |
| --- | --- |
| reg | most common |
| integer | used for counting |
| real | floating point numbers |
| time | simulation time (not used in synthesis) |

* + reg
    - default value = x
    - can be assigned in synchronism with a clock or otherwise
    - Treated as unsigned number
    - Used to model actual sequential HW components like counters, shift registers, etc.
    - Can only be driven in procedural blocks like always and initial
    - eg:
      * reg x,y; // 1 bit
      * reg [15:0] bus; //vector
      * A = B + C; //no 'assign', for such assignment LHS must be declared as reg
      * A <= B + C;
    - Eg:  
      module simple\_counter (clk,rst,count);  
        input clk,rst;  
        output [31:0] count;  
        reg[31:0] count;  
        
        always @(posedge clk or posedge rst) //asynchronous reset  
        begin  
          if (rst)  
            count = 32'b0;  
          else  
            count = count + 1;  
        end  
      endmodule
  + integer
    - signed
    - default size is 32 bits (can't specify size)
    - Synthesis tool determines the size using data flow analysis
    - Eg  
      wire[15:0] X,Y;  
      integer C;  
      C = X + Y; //size of C can be 17 bits
    - Also read systemverilog page
  + real
    - floating point numbers
    - if assigned to an integer, rounded off
    - can't hold high value Z or X, initialized to 0.
    - Eg:  
      real e, pi;  
      initial  
      begin  
          e = 2.718;  
          pi = 314.159e-2  
      end  
      integer x;  
      initial  
        x = pi; //x gets value 3  
        …
  + time
    - store simulation time
    - *$time* gives current simulation time
    - Eg:  
      time curr\_time;  
      initial  
        .…  
        curr\_time = $time;

* Some features are simulation only. Ignored by synthesis tool
  + eg: gate delays
* **Vectors**

wire x,y,z; // single bit

wire [7;0] sum; //MSB is sum[7], LSB is sum[0];  
  
Eg: 32 bit instruction register, 6-bit opcode,3 5-bit operands  
reg[31:0] IR;  
reg[5:0] opcode;

reg[4:0] reg1, reg2, reg3;

reg[10:0] offset;

opcode = IR[31:26]

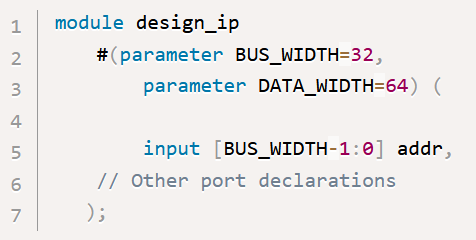
reg1 =  IR[25:21];

reg2 = [20:16];

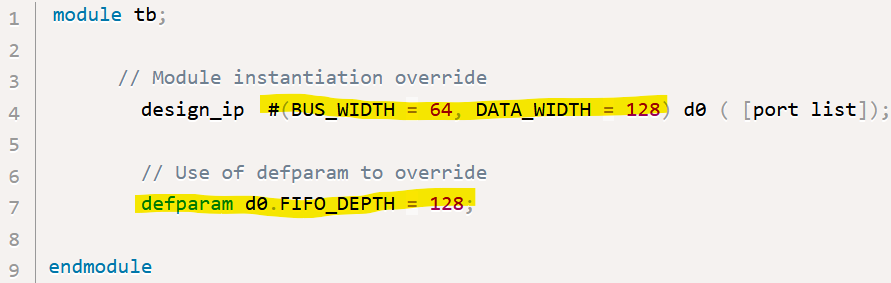
reg3 = IR[15:11];

offset = IR[10:0];

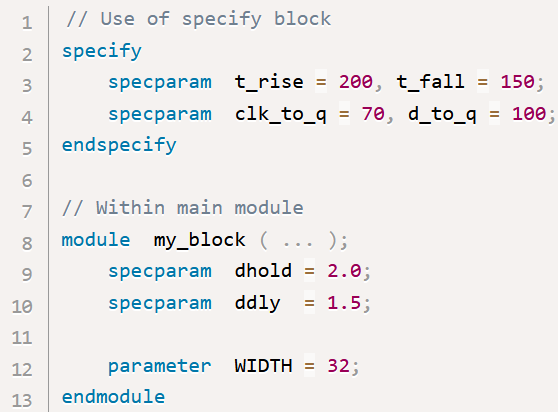
* Multidimenstional arrays:  
    reg[31:0] register\_bank[15:0]; //16 32 bit numbers  
    integer maxtix[7:0][15:0];
* Memory modeled as arrays:  
    reg mem\_bit[0:2047]; // 2K 1-bit words  
    reg [15:0] mem\_word[0:1023] // 1K 16-bit words
* **Specifying  constants**
  + <size>'<base><number>
  + Eg:  
    1'b0101  
    1'b0  
    12'hB3C  //12 bit number: 1011 0011 1100  
    12'h8xF  //12-bit number:1000 xxxx 1111  
    25       //signed number, in 32 bits
  + Integer and real typically expressed in unsized form.
* **parameter**
  + constant with a given name
  + Illegal to modify value at runtime
  + if size and type not specified, defaults to size and type of final assigned value
  + similar to #define in C
  + Eg:  
    parameter HI = 25, LO = 5;  
    parameter up = 2'b00, down = 2'b01,teady = 2'b10;  
    parameter RED = 3'b100, YELLOW = 3'b010, GREEN = 3'b001;
  + Eg: parameterized design: N-bit counter  
    module counter (clear,clock,count);  
      parameter N = 7;  
      input clear,clock;  
      output[N:0] count;  
      reg[N:0] count;  
      always @(negedge clock)  
      begin  
        if(clear)  
           count <= 0;  
        else   
          count <= count + 1;  
      end  
    endmodule
  + 2 types
    1. Module Parameters



Can override during instantiation or using *defparam* :

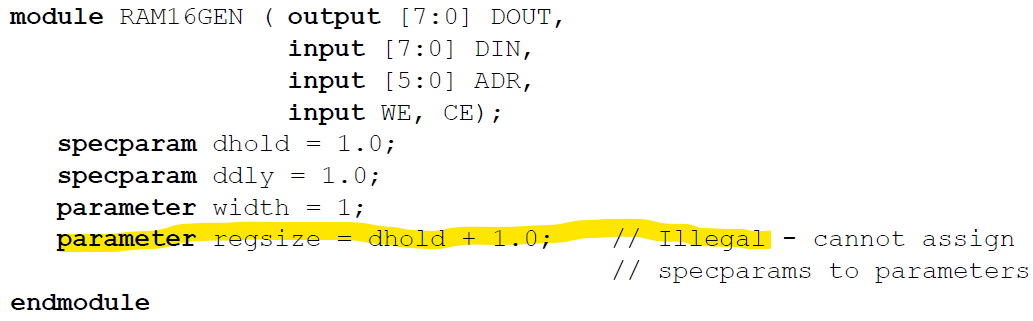


1. Specify Parameters
   * Used for specifying timing/delay values (keyword: *specparam*)
   * Can be inside the specify block or the main module body:



* Spec params without a range will get the range of the final assigned value
* Spec params with a range specification cannot have the range overridden

|  |  |
| --- | --- |
| **Specify parameter** | **Module parameter** |
| Declared by specparam | Declared by parameter |
| Can be declared inside specify block or within main module | Can only be declared within the main module, outside specify blocks |
| May only be used inside a module or specify block | May not be used inside specify blocks |
| May be assigned specparams and parameters | May not be assigned specparams |
| SDF can be used to override values | Instance declaration parameter values or defparam can be used to override |



* Defparam
  + Access component of a module hierarchically (like XMR) and change a parameter
  + Now obsolete (but still in LRM): Alternate: set param through configuration
  + Adds lot of complexity
* Also see SystemVerilog notes (Parameter section)
* **Const Constant**
  + Can be set during simulation (such as in an automatic task). Local params set during elaboration
  + Hierarchical names are allowed  
    const logic option = a.b.c ;
  + can be set to any expression that would be legal without the const keyword
  + An instance of a class (an object handle) can also be declared with the const keyword:

const class\_name object = new(5,3);

* Arguments of new shall be constant expressions
* Non const members of the class can be written
* **output**
  + wire by default unless explicitly declared as a register
* **Pre-defined Logic Gates (Primitive Gates)**
  + and G (out, in1, in2, … , inN)
  + nand G (out, in1, in2, … , inN)
  + or G (out, in1, in2, … , inN)
  + nor G (out, in1, in2, … , inN)
  + xor G (out, in1, in2, … , inN)
  + xnor G (out, in1, in2, … , inN)
  + not G (out, in)
  + buf G (out, in) //buffer
  + bufif0 G (out, in, ctrl);
    - if ctrl = 0, out = in
    - else out = z
  + bufif1 G (out, in, ctrl);
    - if ctrl = 1, out = in
    - else out = z
  + notif0 G (out, in, ctrl);
  + notif1 G (out, in, ctrl);
  + Output port must be a net
  + Input ports may be nets or registers
  + Can specify optional delay (only for simulation):  
    and #5 G1 (f,A,B);
  + Example: xor implementation using nand, and, nor:  
    `timescale 10ns/1ns  
    module exclusive\_or (f,a,b);  
      input a,b;  
      output f;  
      wire t1,t2,t3;  
      nand #5 m1(t1,a,b);  
      and #5 m2(t2,a,t1);  
      and #5 m3(t3,t1,b);  
      nor #5 m4(f,t2,t3);  
    endmodule
* **`timescale**
  + `timescale <reference\_time\_unit>/<
  + precision used for rounding off
  + valid values: 1,10,100
  + valid units: s, ms, us, ns, ps, fs
  + only for simulation (ignored by synthesis tool)
  + compiler directive, not a statement (so no semicolon needed)
* Specifying connectivity during instantiation
  + Positional association:
    - parameters of the instantiated module listed in same order as the original module description
    - Eg:  
      module testbench;  
        reg X1,X2,X3,X4,X5,X6;  
        wire OUT;  
        example DUT(X1,X2,X3,X4,X5,X6,OUT);  
        
        intial  
           begin  
              $monitor ($time, "X1=%b, "X2=%b, "X3=%b, "X4=%b, "X5=%b, "X6=%b, OUT=%b, X1,X2,X3,X4,X5,X6,OUT);  
              #5 X1=1; X2=0; X3=0;X4=1; X5=0; X6=0;  
              #5 X1=0; X3=1;  
              #5 X1=1; X3=0;  
              #5 X6=1;  
              #5 $finish;  
          end  
      endmodule  
        
      module example (A,B,C,D,E,F,Y);  
        wire t1,t2,t3,Y;  
        nand #1 G1(t1,A,B);  
        and #1 G2(t2,C,~B,D);  
        nor #1 G3(t3,E,F);  
        nand #1 G4(Y,t1,t2,t3);  
      endmodule
  + Explicit association:
    - parameters of instantiated module listed in arbitrary order
    - Less chance of error
    - Eg:  
      module testbench;  
        reg X1,X2,X3,X4,X5,X6;  
        wire OUT;  
        example DUT(.OUT(Y),.X1(A),.X2(B),.X3(  
        
        intial  
           begin  
              $monitor ($time, "X1=%b, "X2=%b, "X3=%b, "X4=%b, "X5=%b, "X6=%b, OUT=%b, X1,X2,X3,X4,X5,X6,OUT);  
              #5 X1=1; X2=0; X3=0;X4=1; X5=0; X6=0;  
              #5 X1=0; X3=1;  
              #5 X1=1; X3=0;  
              #5 X6=1;  
              #5 $finish;  
          end  
      endmodule
* Register mapping to wire:  
  module reg\_maps\_to\_wire (A<B,C,F1,f2);  
    input A,B,C;  
    output f2,f2;  
    wire A,B,C;  
    reg f1,f2;  
    always @(A or B or C) //when either A or B or C changes  
    begin  
      f1 = ~(A&B);  
      f2 = f1^C;  
    end  
  endmodule
* **Register mapping to storage cell (latch)**  
  module a\_problem\_case (A<B,C,F1,f2);  
    input A,B,C;  
    output f2,f2;  
    wire A,B,C;  
    reg f1,f2;  
    always @(A or B or C) //when either A or B or C changes  
    begin  
      f2 = f1 ^ f2; //f2 appearing on LHS and RHS => storage cell generated  
      f1 = ~(A&B);    
    end  
  endmodule
* **Latch**  
  module simple\_latch (data, load, d\_out);  
    input data,load;  
    output d\_out;  
    wire t;  
    always @(load or data)  
    begin  
      if(!load)  
        t = data; //no else, so a latch generated for t (if load is 1, value of t should not change, so storage cell needed)  
       d\_out = !t;  
    end  
  endmodule
* **Arithmetic Operators**

|  |  |
| --- | --- |
| + | unary plus |
|  | unary minus |
| + | add |
|  | subtract |
| \* | multiply |
| / | divide |
| % | modulus |
| \*\* | exponentiation |

* If any of the operands are x or z, the result is x
* **Logical Operators**  
    0 = false, nonzero = true
  + reduce each operand to single bit, then perform bitwise operation:  
    0 -> if all bits 0  
    1 -> if any bit 1  
    X -> if any bit Z or X and no bit 1
  + same as bit-wise operators if operands are 1 bit expressions

|  |  |
| --- | --- |
| ! | logical negation |
| && | logical AND |
| || | logical OR |

* **Relational Operators**

|  |  |
| --- | --- |
| != | not equal |
| == | equal |
| >= | greater or equal |
| <= | less or equal |
| > | greater |
| < | less |

If 1 operand unsigned, operation is unsigned

if both signed, smaller operand is sign extended to match bigger

if 1 is real, other operand converted to real

if any operand contains Z or X, result is X

* **Bitwise Operators**

|  |  |
| --- | --- |
| ~ | bitwise NOT |
| & | bitwise AND |
| | | bitwise OR |
| ^ | bitwise XOR |
| ~^ or ^~ | bitwise XNOR |

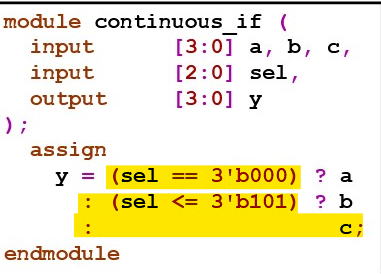
* **Reduction Operators**
  + perform bit-wise operation on all bits of a single operand, resulting in a single bit
  + reduce set of items into a single item
  + Accept single word operand (unary) and produces single bit as output

|  |  |
| --- | --- |
| ~ | bitwise NOT |
| & | bitwise AND |
| | | bitwise OR |
| ^ | bitwise XOR |
| ~^ or ~^ | bitwise XNOR |

* eg:   
  wire[3:0] x;  
  wire y;  
  assign y = &x; // computes and of all bits of x  
  wire[3:0] a,b,c;  
  wire f1,f2,f3;  
  assign a =  4'b0111;  
  assign b =  4'b1100;  
  assign c =  4'b0100;  
  assign f1 = ^a  // f1 gets value 1  
  assign f2 = &(a ^b) // f2 gets value 0  
  assign f3 = ^a & ~^b;  // f3 gets value 1

* **Shift Operators**

|  |  |  |
| --- | --- | --- |
| >> | shift right | assign target = data >> 3; |
| << | shift left |  |
| >>> | arithmetic shift right (the sign bit is inserted,  ie, 1 for -ve, 0 for +ve) | assign target = data >>> 2 |

* **Conditional Operator**  
  cond\_expr ? true\_expr : false\_expr;
* 

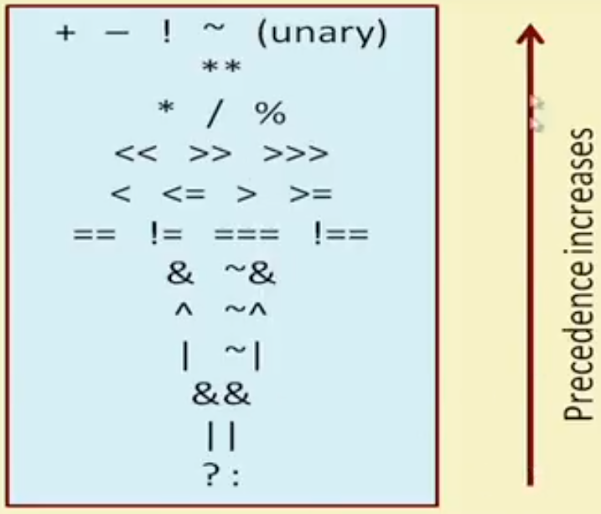
* **Concatenation Operator**  
  {.…, .…, .…} *Joins bits from two or more comma separated expressions*

if lhs/rhs size doesn't match behaviour is undefined  
  
eg:  
assign f = {a,b};  
assign f = {a,3'b101,b};  
assign f = {x[2],y[0],a};

* **Replication Operator**  
  {n{m}}  
  *Joins together n copies of an expression m, where n is a constant*  
    
  eg:  
  assign f = {2'b10, 3{2'b01}, x}; // f gets 10010101x

* **Example: 8 bit adder**  
  module parallel\_adder(sum,cout,in1,  
    input[7:0] in1,in2;  
    input cin;  
    output[7:0] sum;  
    output cout;  
   assign #20 {cout,sum} = in1 + in2 + cin;  
  endmodule

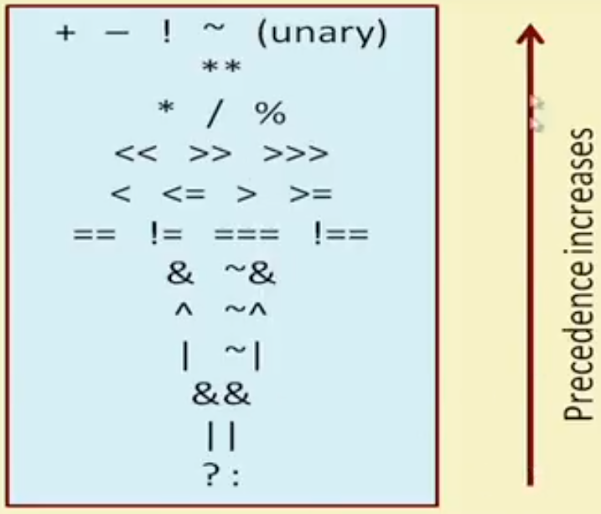
* **=== / !==**   
  Case equality operator. *Exact matching (including x), aka identity operator*== matches x and 0 , x and 1, === only matches x and x
* **Operator Precedence**



* **Description Styles**
  1. **Data Flow**Uses continuous assignment using *assign* statements

**Continuous Assignment**

* Typically used for combinational circuits
* Models behavioral design style, data flow description style
* Continuous assignment : LHS gets updated whenever RHS changes (LHS is continuously driven)
* LHS must be a *net* type, typically a *wire*
* RHS may contain both *register* and *net* types
* a separate process, so cannot be used as a sequential statement in a procedural block
* multiple continuous assignments are "wired together" (depending on wire, want,wor. uwire doesn't support multiple drivers)
* Static binding b/w LHS (net) and RHS (net/register)
* Typically placed in beginning, after port declarations
* A module can have any number of assign statements
* If RHS has a non-constant index, a MUX is generated by synthesis tool, else, wire
* Conditional operator generates a MUX:  
    
  module generate\_set\_of\_mux(a,b,fsel);  
    input[0:3] a,b;  
    input sel;  
    output [0:3] f;  
    assign f = sel ? a : b; //generates MUX (four 2x1 Muxes (input and output are 4 bit wide)  
  endmodule
* Eg: assign f = (a==0) ? (c+d) : (c-d)



There are actually 16 2x1 Mux's (1 for each bit)

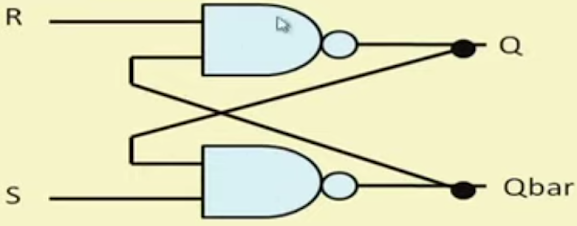
* Eg:   
  module generate\_decoder (out,in,select);  
    input in;  
    input[0:1] select;  
    output[0:3] out;  
    assign out[select] = in;  
  endmodule  
    
  Non constant index in expression on LHS generates a decoder/demultiplexer

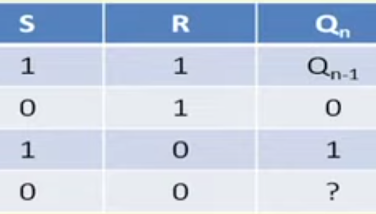


* Constant index on LHS generates simple wire
* **Example of using assign statement to describe sequential circuit:**module level\_sensitive\_latch(D,Q,En);  
    input D,En;  
    output Q;  
    assign Q = En ? D : Q; //if En is 0, Q remembers its old value  
  endmodule

|  |  |  |
| --- | --- | --- |
| **En** | **D** | **Q** |
| 0 | X | Qn-1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

* Modeling a Simple S-R Latch





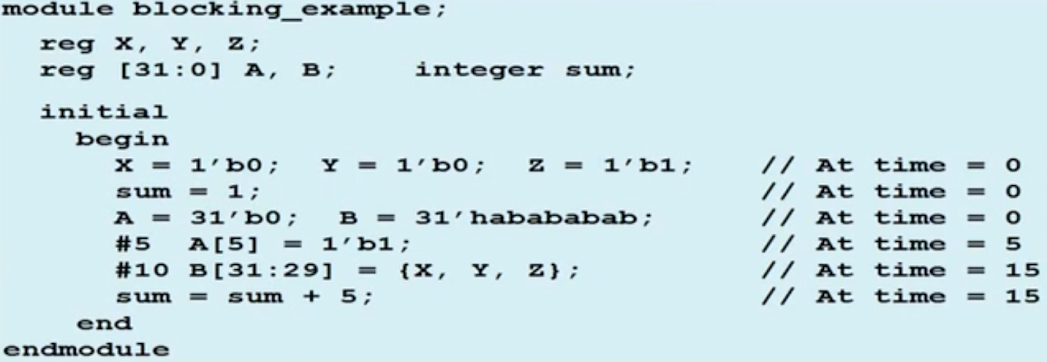
S =  0, R = 0 is not a valid state (race condition - which gate is faster) : simulator will hang, output never converges  
  
module sr\_latch(Q,Qbar,S,R);  
  input S,R;  
  output Q,Qbar;  
  
  assign Q = ~(R & Qbar);  
  assign Qbar = ~(S & Q);  
endmodule

1. **Behavioral**Uses procedural assignments
   1. Blocking
   2. Non-blocking

Similar to high level language

For modelling sequential and parallel behaviour

* Procedural Blocks
  + Region of code containing *sequential* statements
  + Statements execute in order they are written
  + Multiple procedural blocks in a design interact concurrently (no ordering between them)
  + **Procedural Assignments**
    - Used for assignment operations inside a procedural block (ie initial or always)
    - Value assigned remains unchanged (storage) until reassigned (unlike *assign*)
    - LHS can be:
      1. Register type variable (reg, integer, real or time)
      2. Bit select of these variables (e.g. sum[15])
      3. Part select of these variables (e.g. IR[31:26])
      4. Concatenation of any of the above (e.g. {a[5],b[5:2],c} = X + Y;)
    - RHS can be
      1. net type variable
      2. register type variable
    - Not its own process, so can run in a procedural block (unlike continuous assignment)
    - Suitable for sequential logic
    - 2 Types (see ring counter example)
      1. Blocking (=)
         * Syntax:  
           variable\_name = [delay\_or\_event\_control] expression;
         * Statements in same block executed sequentially
         * Statement in one procedural block doesn't block one in another block
         * Recommended for combinational logic (can also generate sequential elements though)
         * Eg:  
           integer a,b,c;  
           initial  
             begin  
                a = 10; b = 20; c = 15;  
                a = b + c; //a becomes 35  
                b = a + 5; // b becomes 40  
                c = a-b; //c becomes -5  
             end



1. Non-blocking (<=)
   * Syntax:  
     variable\_name <= [delay\_or\_event\_control] expression;
   * Allow scheduling of assignments w/o blocking execution of statements that follow within the same procedural block
   * Allows concurrent procedural assignment, suitable for sequential logic.
     + several *reg* type variables can be assigned synchronously under the control of a common clock
   * good option for variables that represent design storage, eg registers
   * Eg:
     + integer a,b,c;  
       initial  
          begin  
             a = 10; b = 20; c = 15;  
          end  
       initial  
          begin  
             a <= #5 b+c; //a becomes 35 at t = 5  
             b <= #5 a+5; //b becomes 15 at t = 5  
             c <= #5 a-b; //c becomes -10 at t = 5  
          end

* Swapping 2 values
  1. always @(posedge clk)  
        a = b;    
     always @(posedge clk)  
        b=a;  //*race condition*, both registers get same val, either 'a' or 'b'  
              //depending on which assignment scheduled first by simulation tool
  2. always @(posedge clk)  
       a <= b;  
     always @(posedge clk)   
       b <= a; // no race condition, variables correctly swapped (RHS is calculated, LHS update is \*scheduled\*)
  3. always @(posedge clk)  
       begin  
         ta = a;  
         a = b;  
         b = ta; //swapped correctly using blocking assignments, but need extra variables  
       end

* **Some rules to be followed**
  + Recommended not to mix blocking and non-blocking assignments in the same "always" block
  + Delays are only for simulation, ignored by synthesizer
    - may lead to functional mismatch b/w design model and synthesized netlist
  + A variable cannot be the target of both a blocking and a non-blocking assignment
    - x = x + 5;  
      x <= y;  // not permissible, warning/error

* Types
  1. initial Block
     + Non-synthesizable or testbench construct
     + Executed once at the beginning of the simulation (at t = 0)
     + Grouped between "begin … end"
     + Used only in test benches. Not synthesizable.
     + If multiple initial blocks, all run concurrently
     + Specifies stimulus to be applied to DUT
     + Specifies how DUT outputs are to displayed/handled
     + Specifies where waveform information is to be dumped
     + Eg:  
       module testbench\_example;  
         reg a,b,cin,sum,cout;  
         
         initial  
           cin = 1'b0;  
         
         initial  
           begin  
              #5 a = 1'b1; b = 1'b1;  
              #5 b = 1'b0;  
           end  
         
         initial  
           #25 $finish  //ends the simulation  
       endmoudle
  2. always Block
     + Syntax:   
       always @(<event\_expression>)  
       begin  
         <sequential\_statement\_1>;  
         <sequential\_statement\_2>;  
         …  
         <sequential\_statement\_n>;  
       end
     + sensitivity list can also be \*:  
       always @\* //gets triggered whenever any input changes
     + Synthesizable construct
     + Continuous loop that never terminates
     + Starts at t = 0
     + Grouped between "begin … end"
     + Models HW circuits in a more natural way (runs when power is on)
     + Used to model a block of activity repeated indefinitely in a digital circuit, eg: continuous clock signal
     + For simulation purposes, we can specify delays
     + Eg:  
       module generating\_clock;  
         output reg clk;  
         initial  
           clk = 1'b0;  //initialized to 0 at t=0  
         always  
           #5 clk = ~clk //toggle every 5 time units  
         
         initial  
           #500 $finish  
       endmodule
     + initial and always executed concurrently, order is not defined
     + can have multiple always blocks, all run concurrently

* Restrictions
  + Only reg type variables can be assigned values inside initial or always blocks
    - When condition is not true (in case of always), the LHS variables should remember the last value
    - Can be mapped to HW registers (may also be at times still be mapped to a combinational circuit)
  + Any type of variable can be used in the event expression
  + There can be multiple assignments inside a begin…end block, which may execute sequentially or concurrently, depending on kind of assignment:
    - Blocking assignment: a = b + c;
    - Non-blocking assignment: a <= b + c;
* User defined primitives  
  primitive mux (output out, input sel,a,b);

table

               // sel   a     b     out

0    1     ?  :  1;

0    0     ?  :  0;

1    ?     0  :  0;

1    ?     1  :  1;

x     0    0  :  0;

x    1     1  :  1;

endtable

endprimitive

module tb;

reg sel,a,b;

reg[2:0] dly;

wire out;

integer i;

// instantiate UDP

mux u\_mux(out,sel,a,b);

initial begin

a <= 0;

b <= 0;

for (i = 0; i < 10; i = i+1) begin

dly = $random

#(dly) a <= $random;

dly = $random

#(dly) b <= $random;

dly = $random

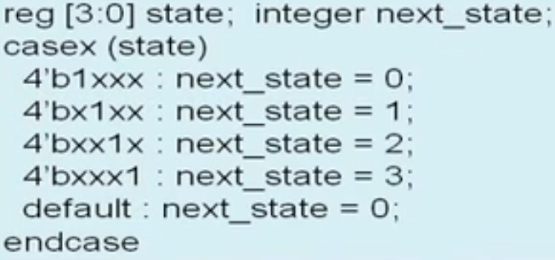
#(dly) sel <= $random;

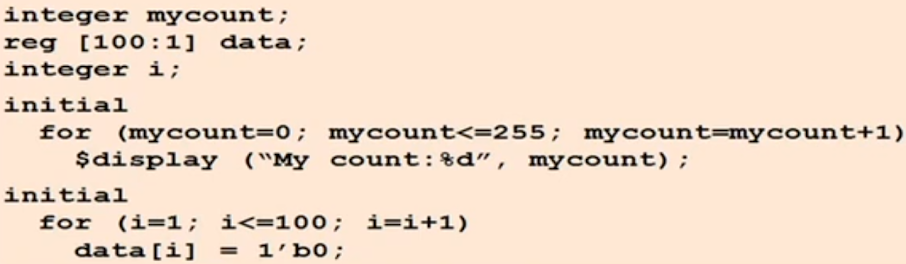
end

end

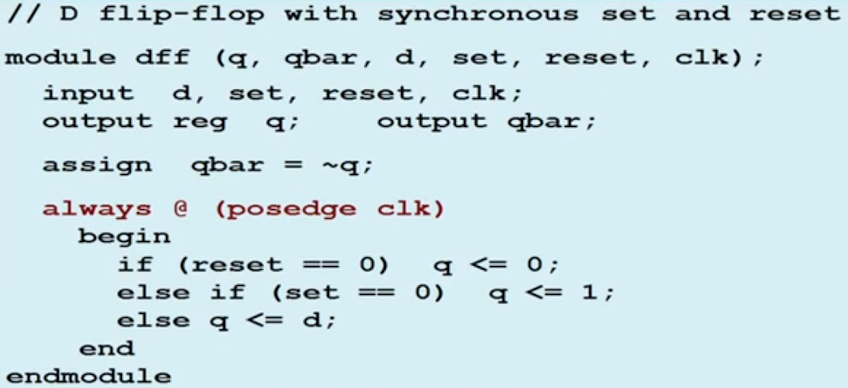
endmodule

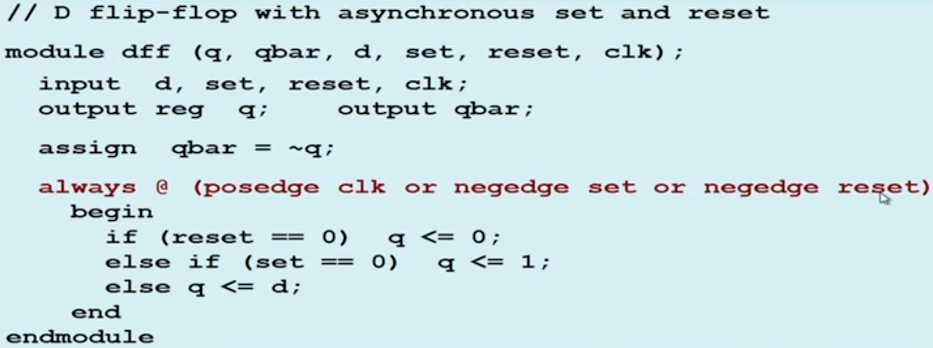
* **Procedural Sequential Statements**
  + begin … end
    - combines multiple sequential statement
    - not required for single statement
  + if … else
    - if (<expression>)  
        sequential\_statement;
    - if (<expression>)  
        sequential\_statement;  
      else  
        sequential\_statement;
    - if (<expression1>)  
        sequential\_statement;  
      else if (<expression1>)  
        sequential\_statement;  
      else if (<expression3>)  
        sequential\_statement;  
      else default\_statement;
    - Each of the above sequential\_statement can be a single statement or group with begin … end
  + case
    - case <expression>  
         expr1: sequential\_statement1;  
         expr2: sequential\_statement2;  
         expr3: sequential\_statement3;  
        …  
         exprN: sequential\_statementN;  
         default: default\_statement;  
      endcase
    - order of expr's is order of comparison
    - 2 variation:
      * casez: treats all "z" values in case alternatives or case expressions as don’t cares
      * casex: treats all "z" and "x" values in case alternatives or case expressions as don’t cares



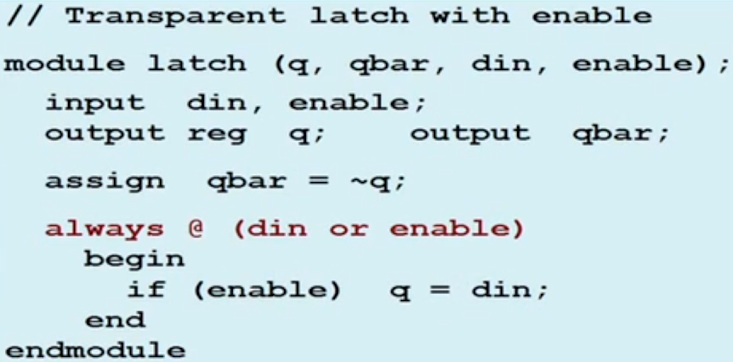
* while loop  
  while (<expression>)  
    sequential\_statement; //single statement or group within begin/end
* for loop  
  for (initial\_condition; terminating\_condition; update\_control\_var)  
    sequential\_statement;   //single statement or group within begin/end
  + Can be used to initialize an array of memory
  + 
* repeat loop  
  repeat (<expression>)  
     sequential\_statement;
  + executes the loop fixed number of times
  + expression can be a constant, variable or signal value
    - if expression is variable or signal, the value used is the one at the beginning of the loop
  + expression cant be a general logical expression
  + Eg:  
    reg clock;  
    initial  
      begin  
        clock = 1'b0;  
        repeat (100)  
          #5 clock = ~ clock  // exactly 100 clock pulses  
      end

* forever loop  
  forever  
    sequential\_statement; //single statement or group within begin/end
  + executes forever until $finish is encountered in the test bench
    - equivalent to while where expression is always true
  + typically used with timing specifier, otherwise it would run indefinitely without advancing time. Rest of the design will never be executed.
  + Eg:  
    reg clk;  
    initial  
      begin  
        clk = 1'b0;  
        forever #5 clk = ~clk; //clock period of 10 units  
      end
* #(time\_value)
  + Makes a block suspend for time\_value units of time
  + Time unite specified using `timescale command
* @(event\_expression)
  + Makes a block suspend until "event\_expression" triggers
  + event\_expression can be
    - Change of a signal value
    - Positive or negative edge occurring on signal (posedge or negedge)
    - List of above-mentioned events, separated by "or" or comma
  + Posedge = transition from {0,x,z} to 1, or 0 to {x,z}
  + Negedge = {1,x,z} to 0 or 1 to {x,z}
  + Eg:
    - @(in) // "in" changes
    - @(a or b or c)  // any of "a", "b" , "c" changes
    - @(a,b,c)  // any of "a", "b" , "c" changes
    - @(posedge clk)  // positive edge of "clk"
    - @(posedge clk or negedge reset)
    - @(\*)  //any variable changes
  + Flip Flop example (edge triggered)





* Latch (level triggered)



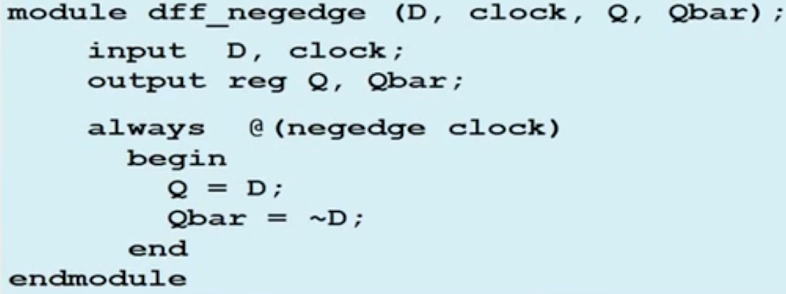
* **Examples**
  + ***Combinational logic, 2x1 mux***

module mux21 (in1,in0,s,f);  
  input in1,in0,s;  
  output reg f;  
    
  always @(in1 or in0 or s) // can use ',' instead of 'or', could also use "always @(\*)  
     if (s)

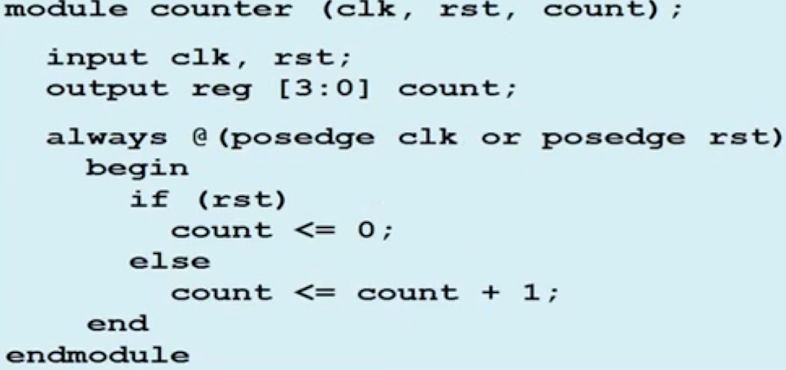
        f = in1;  
     else

        f = in0;  
endmodule

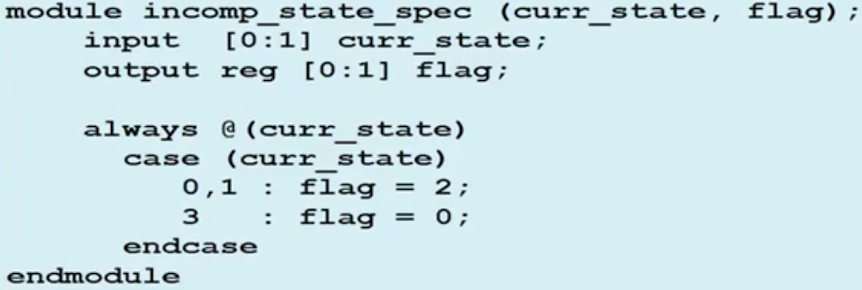
* **Sequential logic, Explicit Synchronous Negative Edge D flip-flop**



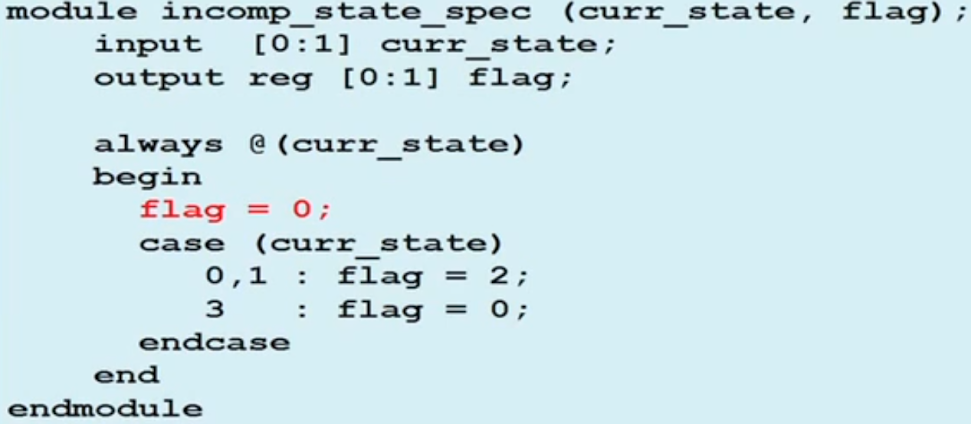
* **4-bit Counter with Asynchronous Reset**



* **Another Sequential Circuit**



Above looks like a combinational circuit, but it's sequential, because:  
Case is incompletely specified, ie, if current\_state is 2, flag remains unchanged => is mapped to a storage element (latch)  
  
By adding 1 line, we can change it to pure combinational => no latches generated:

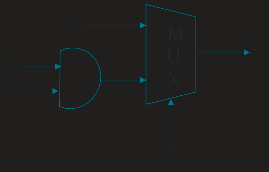


* **Sequential**

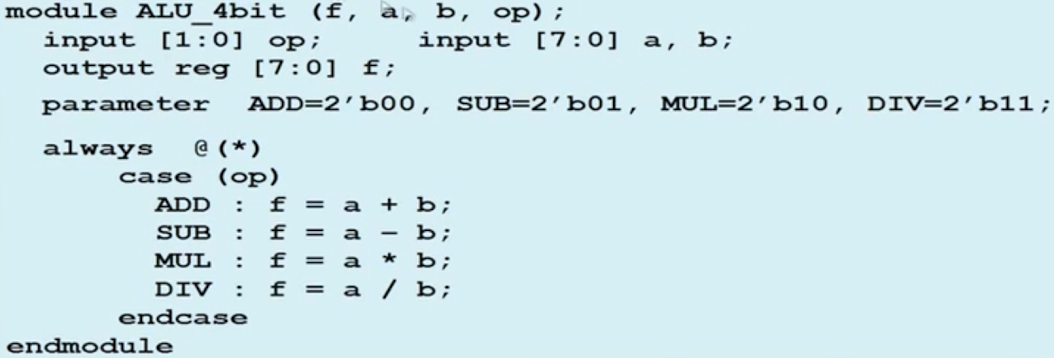
module xyz (input a,b,c, output reg f)  
  always @(\*)  
    if (a==1) f = b & c; //for a = 0, value of f unspecified => latch  
endmodule



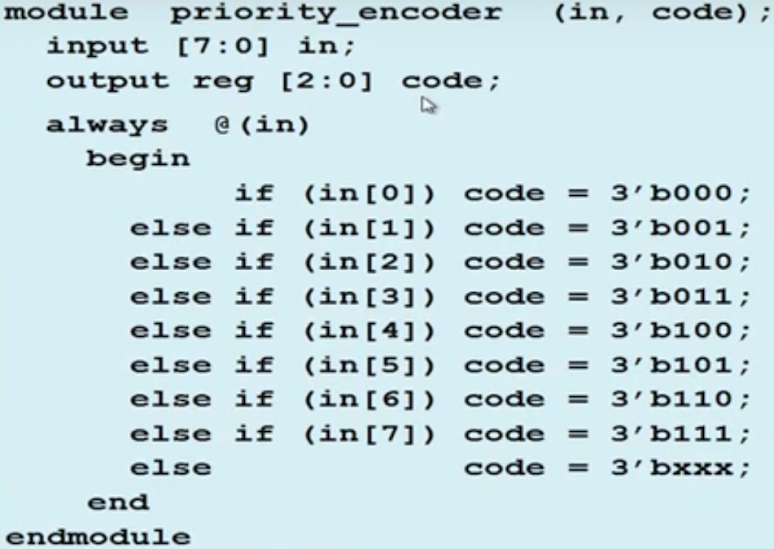
**Slight Modification, Combinational:**  
module xyz (input a,b,c, output reg f)  
  always @(\*)  
    f = c;  
    if (a==1) f = b & c;   
endmodule



* **4-Function ALU**

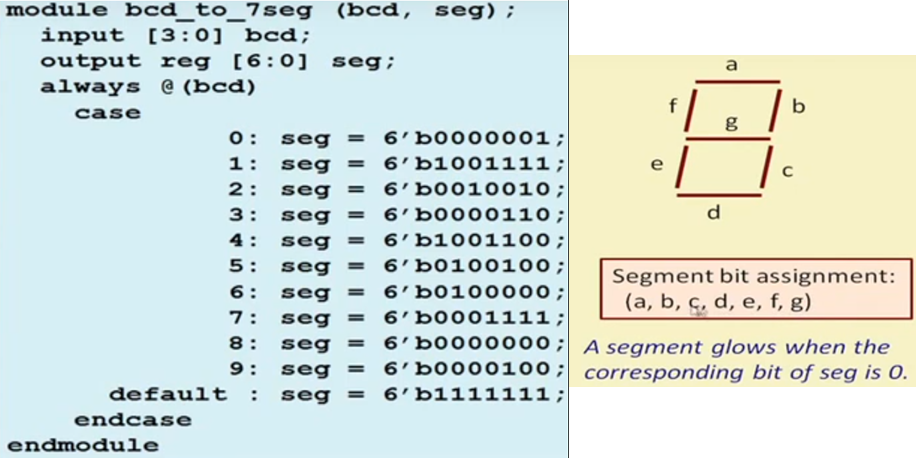


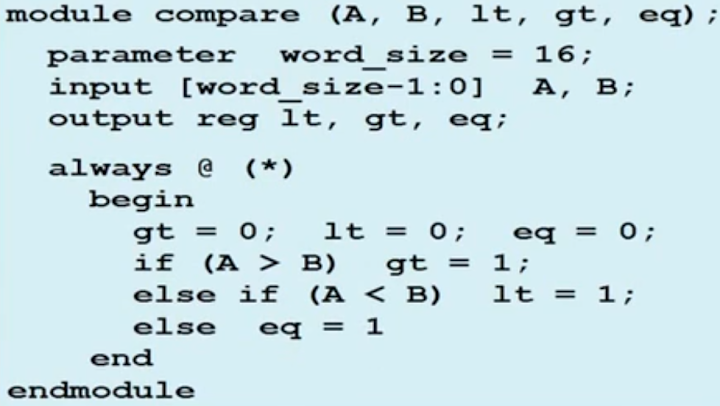
* **Priority Encoder**



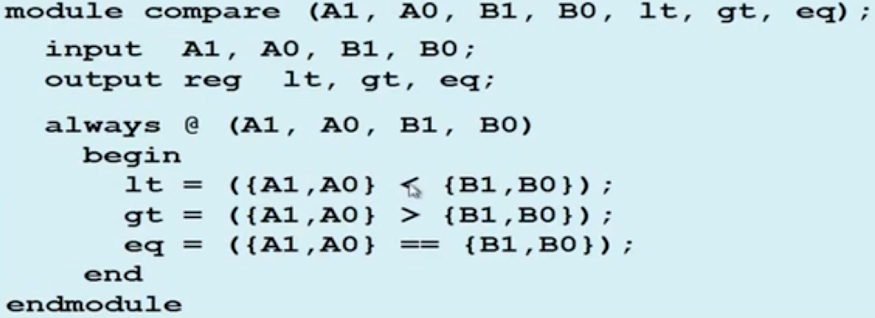
0 has highest priority

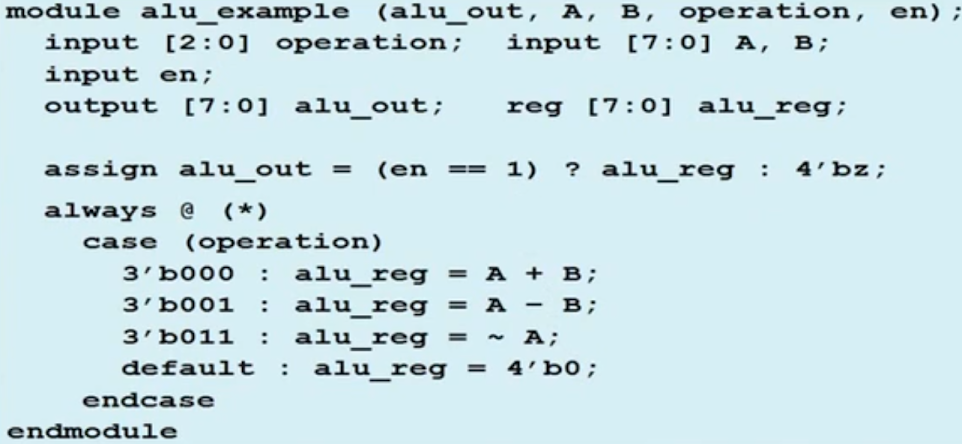
* **BCD to 7 Segment Display**



* **n-bit Comparator (pure behavioral)**
* 

* **2-Bit Comparator**



* **ALU**
* 

* **Shortcuts in Declarations**
  + output and reg can be combined  
    output reg[7:0] data;
  + Variable can be initialised when it's declared  
    reg clock =  0; // instead of reg clock; initial clock = 0;

* Can specify inputs and outputs in module line:  
  module xyz (input a,b,c, ouput reg f);
* Cross Module Reference (XMR)  
  [http://only-vlsi.blogspot.com/](http://only-vlsi.blogspot.com/2017/09/xmr-cross-module-reference.html)

* Generate Block
  + Multiply module instances (Generate For)
  + Perform conditional instantiation of any module (Generate If)
  + Ability for design to be based on Verilog Parameters (Generate Case)
  + Cannot contain port, parameter, specparam declarations, or specify blocks
  + Between generate …. Endgenerate
  + Generate For Loop
    - // Design for a half-adder  
      module ha ( input   a, b,  
                  output  sum, cout);

  assign sum  = a ^ b;  
  assign cout = a & b;  
endmodule

// A top level design that contains N instances of half adder  
module my\_design  
        #(parameter N=4)  
                (        input [N-1:0] a, b,  
                        output [N-1:0] sum, cout);

// Declare a temporary loop variable to be used during  
        // generation and won't be available during simulation  
        genvar i; //genvar tells the tool that this variable should be used during elaboration of  
                //generate block

// Generate for loop to instantiate N times  
        generate  
                for (i = 0; i < N; i = i + 1) begin  
               ha u0 (a[i], b[i], sum[i], cout[i]);  
                end  
        endgenerate  
Endmodule  
  
//Example of instantiation of my\_design in the testbench:  
my\_design #(.N(N)) md( .a(a), .b(b), .sum(sum), .cout(cout));

* Generate If
  + //Design #1: Multiplexer design uses an "assign" statement to assign  
    // out signal  
    module mux\_assign ( input a, b, sel,  
                       output out);  
      assign out = sel ? a : b;

  // The initial display statement is used so that  
  // we know which design got instantiated from simulation logs  
  initial  
          $display ("mux\_assign is instantiated");  
endmodule

// Design #2: Multiplexer design uses a "case" statement to drive out signal  
module mux\_case (input a, b, sel,  
                 output reg out);  
  always @ (a or b or sel) begin  
          case (sel)  
              0 : out = a;  
             1 : out = b;  
          endcase  
  end

  // The initial display statement is used so that  
  // we know which design got instantiated from simulation logs  
  initial  
    $display ("mux\_case is instantiated");  
endmodule

// Top Level Design: Use a parameter to choose either one  
module my\_design (        input a, b, sel,  
                                 output out);  
  parameter USE\_CASE = 0;

  // Use a "generate" block to instantiate either mux\_case  
  // or mux\_assign using an if else construct with generate  
  generate  
          if (USE\_CASE)  
        mux\_case mc (.a(a), .b(b), .sel(sel), .out(out));  
      else  
        mux\_assign ma (.a(a), .b(b), .sel(sel), .out(out));  
  endgenerate

endmodule

* Generate Case
  + // Design #1: Half adder  
    module ha (input a, b,  
               output reg sum, cout);  
      always @ (a or b)  
      {cout, sum} = a + b;

initial  
    $display ("Half adder instantiation");  
endmodule

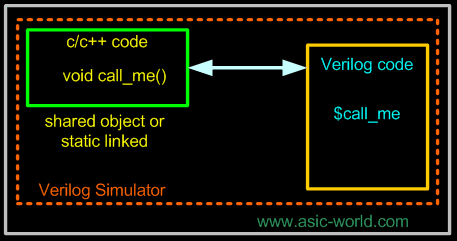
// Design #2: Full adder  
module fa (input a, b, cin,  
           output reg sum, cout);  
  always @ (a or b or cin)  
    {cout, sum} = a + b + cin;

initial  
      $display ("Full adder instantiation");  
endmodule

// Top level design: Choose between half adder and full adder  
module my\_adder (input a, b, cin,  
                 output sum, cout);  
  parameter ADDER\_TYPE = 1;

generate  
    case(ADDER\_TYPE)  
      0 : ha u0 (.a(a), .b(b), .sum(sum), .cout(cout));  
      1 : fa u1 (.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));  
    endcase  
  endgenerate  
endmodule

* **Programming Language Interface (PLI)**
  + Mechanism to invoke C or C++ functions from Verilog code
  + Function invoked = "*System call*"
  + Built-in system calls: $random, $display, $stop, $finish
  + Allows user to create custom system calls for:
    - Power analysis.
    - Code coverage tools.
    - Modifying the Verilog simulation data structure - more accurate delays.
    - Custom output displays.
    - Co-simulation.
    - Designing debug utilities.
    - Simulation analysis.
    - C-model interface to accelerate simulation.
    - Testbench modeling.
  + Steps
    - Write functions in C/C++
    - Compile to generate shared libraries (.dll for Windows, .so for Linux)
    - Use the functions in verilog testbench
    - Pass C/C++ function details (libraries) to simulator during compilation of Verilog code
    - Run simulator



* Whenever simulator encounters '$', control is passed to the PLI routine
* Details (7 part tutorial on CLI) [http://www.asic-world.com/](http://www.asic-world.com/verilog/pli1.html)

Escaped Identifiers

* start with \, end with space

* **Simulation Cycle**
  + Process: verilog code to evaluate, eg: procedural blocks and continuous assignments
  + Evaluation Event: evaluation of a process
  + Update Event: change in value of net/variable
  + Event Queue: queue of events ordered by simulation time
  + Scheduling an Event: putting an event onto the event queue
  + Simulation Cycle: processing all active events in the event queue
  + When there's a blocking assignment in a procedural block, simulator checks if there are any other (always) blocks affected by this update, then executes them (back to active region)
  + Then in the NBA phase the non blocking assignments are executed, and may trigger further always blocks (back to active region)

Functions

* declared only within a module
* by default static : only 1 copy of the function variables exist. generally not recursive
* can access variables from outside using hierarchical reference
* Cannot have inout/output ports
* can declare variables, not nets
* cannot communicate with the scheduler
  + assignments are always blocking
  + executes completely and instantly
* can assign to module variable, but not nets (can lead to side effects)
  + better to return a wide enough vector for any required values, then deconstruct after returning from function
* automatic functions (verilog 2001):
  + creates a new temporary set of variables for each instance
  + can be recursive

* can be used in RHS of an assign statement, or in a procedural block
* Constant Functions
  + Function value must be caluable at elaboration
    - usually inputs are constant
  + cannot be placed in any generate scope
  + cannot contain hierarchical references or system function calls (except $display - in simulator, not elab)
  + cannot make constant function calls  in a context requiring constant expressions
  + can access only functions and module parameters and nothing else declared outside function definition
  + defparam use may produce undefined results

**Tasks**

* by default static
* automatic tasks (like functions)
  + needed if multiple procedural blocks are calling the task (they'll be concurrent and need their own set of task variables)
* task assignments can be blocking or nonblocking
* can declare variables, not nets
* can use the scheduler with delay and timing controls
* can re-write any task not having time as a function
* can be disabled (forced to exit):  
  module mod;  
     …  
     always @ (posedge interrupt)  
     begin  
         disable task\_name;  
      end  
  .…  
  + can disable a named block in the same way
* Function and task arguments are passed by value, but can hierarchically reference the variables in the defining scope

* Design -> gate level netlist
  + gates, cells from specified target technology library
* Try to meet design constraints for area and performance
* Produce reports showing how successful it was
* Optimize circuits to meet constraints
* Generate small/fast/compromise design
* Can try different architectures to see which meets the constraints

SYNTHESIS

1. HDL --> boolean network of sequential storage elements and logic "cone" expressions that feed them
2. logic restructuring: prune unneeded logic, guess operator implementation eg, use carry lood-ahead adder or ripple carry adder based on operand width
3. map design to gates available in the target library, perform optimizations based on given design constraints and parameters

* Steps
  1. RTL
  2. Define and apply constraints
  3. Synthesis Strategy
     1. Synthesize entire design or in smaller parts?

Top down strategy: propagate constraints down  
Bottom up: develops higher level constraints based on estimated timing of previously synthesized sub-design

1. Verify that post synthesis structural design  is functionally equivalent to pre-synthesis RTL design (can do through formal equivalence checking). Can also do :
   1. self-checking test bench
   2. play back captured RTL vectors
   3. compare waveform dumps

* Example of design constraints:
  + required clock spec
  + input/output timing requirements
  + area goals
  + environmental conditions
* For synthesis, must not mix blocking and non-blocking assignments to the same variable

* If for a combination of inputs, a variable is not updated, it retains its previous value. Synthesis tool infers a latch to implement this behaviour
* To avoid latch inference, provide a value for every combination of inputs (eg. give else clause for every if, or, give default assignments)
* Continuous assignments
  + drive nets
  + represent combinational logic

For sequential logic, output is not all times a combinational function solely of inputs

* Some storage may be involved
* Since storage is already present, incomplete assignment does not infer a latch
* So default assignments not required

**Blocking vs Nonblocking Assignments**

* Better to use nonblocking assignments for storage elements
* For synthesis, cannot mix the two for the same variable
* For nonblocking, order of assignments not relevant
  + simulation schedules the assignments for the NBA region of the stratified event queue
* For above example, for case 3 synthesis infers 2 registers (for c and b)  
  For case 4: on register (b is a temp variable)

**Temporary Variables in Sequential procedures**

* Written first and then read in the same procedure
* Alias for an expression
  + no registered inferred
* Typical use model: assign using blocking assignment, then use in RHS of nonblocking assignments

**Persistent Variables in Sequential procedures**

* Read first and then written in the same procedure
* Synthesis infers a register to hold the value for the next read

To infer a latch:

* Make assignments to latch logic as you do for sequential logic (make nonblocking assignment to latch/storage variables
* Latch  based design can have higher performance than register based, but can be difficult to design and debug
* Latch inference is almost always a mistake (should only be done deliberately for special cases)

3 state logic is infered if a variable is conditionally assigned high impedance value