ENGRD 2300
Introduction to Digital Logic Design
Fall 2009

Verilog

Cornell University

Announcements

- **Prelim 1**
  - **Being graded...**
- **Regrade Procedure**
  - **Fill out the regrade request form within one week**
- **HW4 to be posted soon**

Readings

- **Sections 7.1-7.8**
- **Sections 5.1, 5.4, 7.13 (Today)**
  - **Now's the time to worry about the HDL (Verilog) stuff**
  - **I.e., Review Verilog parts of Ch 6**
- **Sections 8.1, 8.4 – 8.5 (Thursday)**

Hardware Description Languages

- **Why use an HDL?**
  - Describe complex designs (millions of gates)
  - Input to synthesis tools (synthesizable subset)
  - Design exploration with simulation
- **Why not use a general purpose language?**
  - Support for structure and instantiation
  - Support for describing bit-level behavior
  - Support for timing
  - Support for concurrency
- **Verilog vs. VHDL (VHSIC HDL)**
  - Verilog is relatively simple and close to C
  - VHDL is close to Ada (DoD-sponsored language)
  - Verilog has 50% of the world digital design market (larger share in US)

Not a Programming Language

- **With every keystroke say, "I'm designing hardware"**

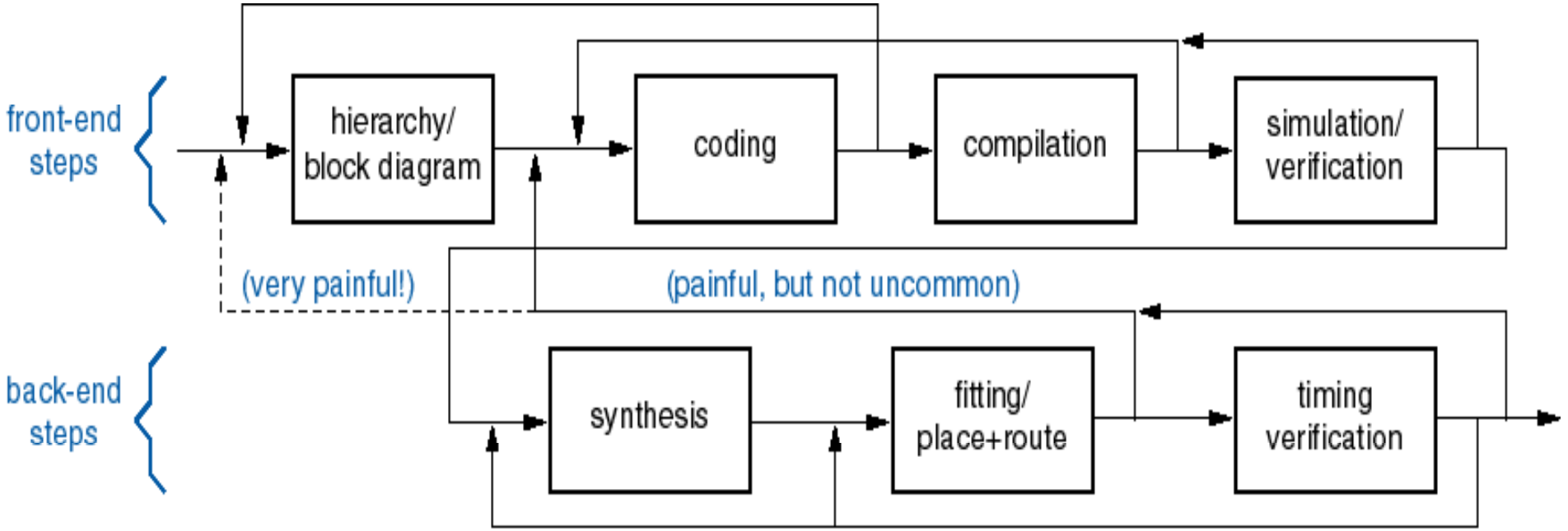
- **Concurrent language**

```
a <= ~b;
c <= d;
```

  - **Which happens first?**

- **Warnings**
  - **It's easy to create tons of hardware**
  - **It's easy to create descriptions that can't be synthesized**

HDL-based design flow



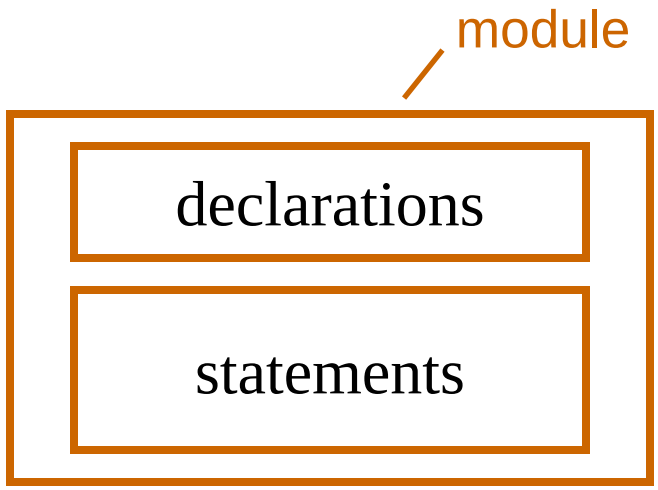
- **Back-end differs by target technology**
  - **May be PLD, CPLD, FPGA, or ASIC**
- **For ASICs, verification and fitting phases are usually much longer (as a fraction of overall project time)**

Verilog

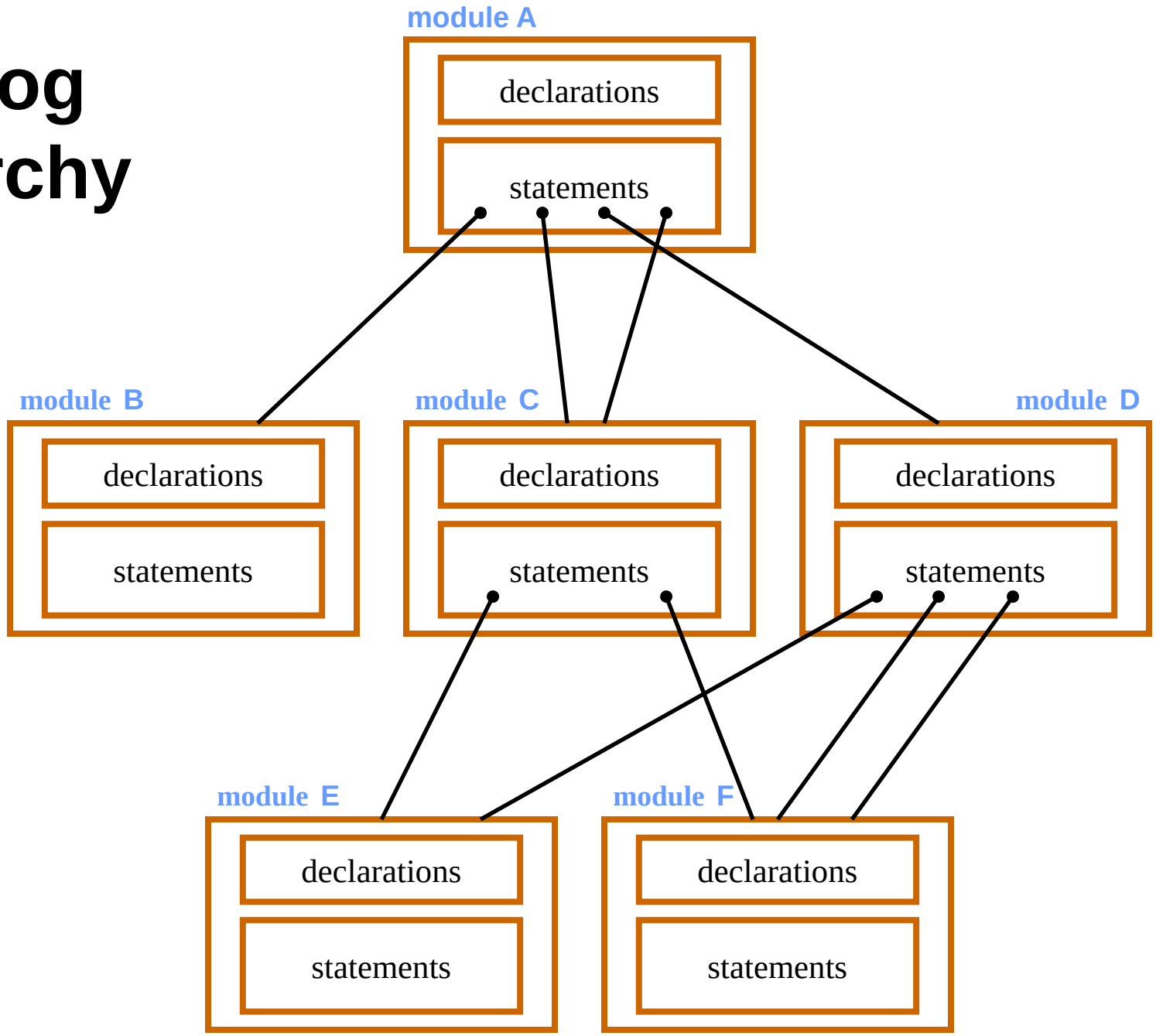
- Developed in the early 1980s by Gateway Design Automation (later bought by Cadence)
  - Syntactically similar to C
  - Shares market 50/50 with VHDL
- Used for design description, simulation, and synthesis
  - Synthesis became practical in the early 90s and use of VHDL and Verilog has taken off since then
- Note that only a subset of the language can be synthesized
  - Easy to code something that can't be synthesized
  - Try to distinguish, which is which!

Verilog Program Structure

- **System is a collection of modules**
  - **Module corresponds to a single piece of hardware**
- **Declarations**
  - **Describe names and types of inputs and outputs**
  - **Describe local signals, variables, constants, etc.**
- **Statements specify what the module does**



Verilog Hierarchy

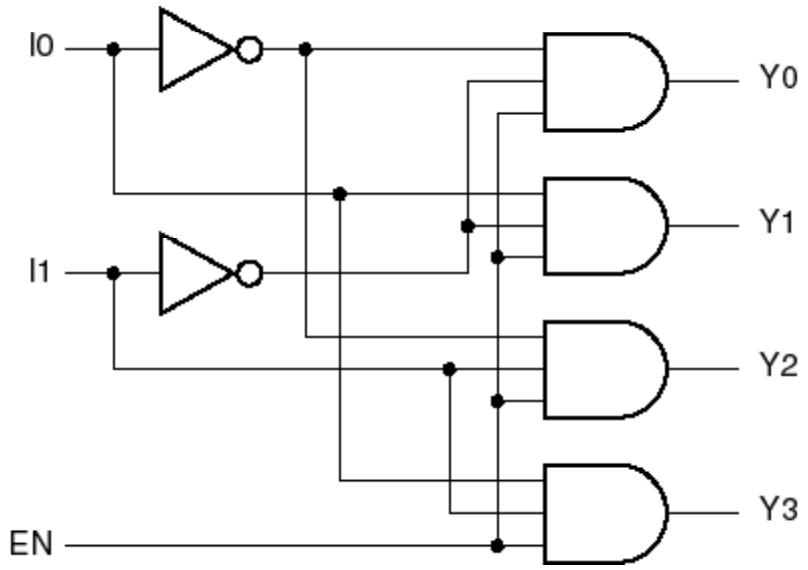


Verilog Program Structure

```
module V2to4dec( i0,i1,en,y0,y1,y2,y3 );
   input i0,i1,en;
   output y0,y1,y2,y3;
   wire noti0,noti1;

   not U1(noti0,i0);
   not U2(noti1,i1);
   and U3(y0,noti0,noti1,en);
   and U4(y1,    i0,noti1,en);
   and U5(y2,noti0,    i1,en);
   and U6(y3,    i0,    i1,en);
endmodule
```

Declarations

Statements



Verilog Signals

- **Verilog signals can have 1 of 4 values**
  - 0 Logical 0, or false
  - 1 Logical 1, or true
  - x Unknown logical value
  - z High impedance
- **Bitwise Boolean Operators**
  - & AND
  - | OR
  - ^ Exclusive OR
  - ^~ Exclusive NOR
  - ~ NOT

A	B	A & B
0	0	0
0	1	0
0	x	0
0	z	0
1	0	0
1	1	1
1	x	x
1	z	x
x	0	0
x	1	x
x	x	x
x	z	x
z	0	0
z	1	x
z	x	x
z	z	x

Logic Operations with Signals

- **Example:**

  0 | x = ? (x)

  0 | z = ? (x)

  1 | x = ? (1)

  1 | z = ? (1)

  x | x = ? (x)

  x | z = ? (x)

  z | z = ? (x)

Verilog Nets

- **Two types of signals**
  - **Nets and Variables**
- **A *net* roughly corresponds to a wire in a circuit**
  - **Provides connectivity between elements in a structural model**
- **Default net type is a *wire***
  - **E.g., `wire noti0, noti1;`**
  - **Any signal not in the input/output list or net declaration is assumed to be a wire**
- **Other net types, used less frequently**
  - **tri, triand, trior, wand, wor, supply0, supply1, …**
  - **Only wire and tri supported by Quartus II**

Verilog Variables

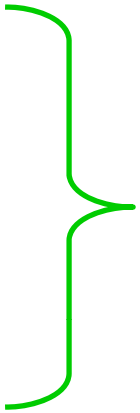
- **Variables store values during program execution, but may not have physical significance**
  - **Common types are `reg` and `integer`**
  - **Variables are only used in *procedural code***
  - **Procedural code can only assign values to variables**
  - **Variables cannot be changed from outside procedural code or from outside a module**
- **A *reg* variable may be a single bit or vector of bits**
- **An *integer* variable is a 32-bit (or larger) vector of bits interpreted as an integer**

Verilog Vectors

- **Grouping of 1-bit signals**
  - **reg [7:0] byte1, byte2, byte3;**
  - **reg [1:16] Zbus;**
  - **wire [0:3] asel;**
- **Bit selection**
  - **byte1[5:2] or Zbus[3:7]**
- **Concatenation**
  - **{byte1,byte2}**
- **Bitwise Boolean operators**
  - **byte1 & byte2**
  - **Zbus | byte3 (pad on left with 0's)**
- **Literals**
  - **7'b0011011**
  - **8'hA7**
- **Vectors are treated as unsigned integers**
  - **Right-most bit is least significant**
  - **Independent of ascending or descending indexing**

Arithmetic and Shift operators

- **Arithmetic**
  - **+ Addition**
  - **- Subtraction**
  - *** Multiplication**
  - **/ Division**
  - **% Modulus**
- **Shift**
  - **<< Shift left**
  - **>> Shift right**



Beware!

These can be expensive, slow, or not synthesizable!

Conditionals

- **Logical Operators**

  && **logical AND**

  || **OR**

  ! **logical NOT**

  == **logical equality**

  != **logical inequality**

  > **greater than**

  >= **greater than or equal**

  < **less than**

  <= **less than or equal**

- **Use these operators *only* in *conditional expressions – not* for Boolean operations**

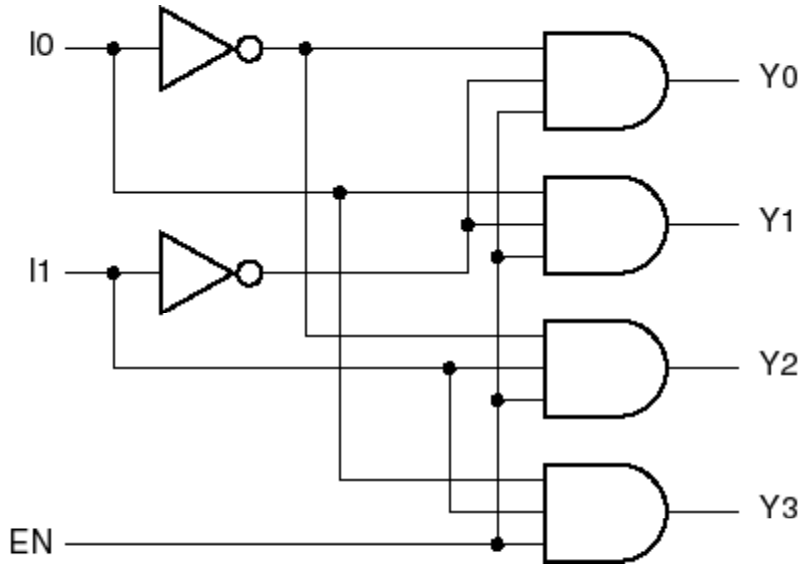
Verilog programming styles

- **Structural**
  - Define explicit components and the connections between them
  - Textual equivalent of drawing a schematic
- **Dataflow**
  - Describe circuit in terms of *flow* of data and *operations* on that data
  - Uses *continuous-assignment* statements
- **Behavioral**
  - Write an algorithm that describes the circuit's output
  - May not be synthesizable or may lead to a very large circuit
  - Always useful for simulation

Structural Style

```
module V2to4dec( i0,i1,en,y0,y1,y2,y3 );
   input i0,i1,en;
   output y0,y1,y2,y3;
   wire noti0,noti1;

   not U1(noti0,i0);
   not U2(noti1,i1);
   and U3(y0,noti0,noti1,en);
   and U4(y1,    i0,noti1,en);
   and U5(y2,noti0,    i1,en);
   and U6(y3,    i0,    i1,en);
endmodule
```

Direct correspondence
between Verilog and schematic



Dataflow Style

```
module V2to4decdf( i0,i1,en,y0,y1,y2,y3 );
  input i0,i1,en;
  output y0,y1,y2,y3;

  assign y0 = en & ~i0 & ~i1;
  assign y1 = en &   i0 & ~i1;
  assign y2 = en & ~i0 &   i1;
  assign y3 = en &   i0 &   i1;
endmodule
```

Note the correspondence between Verilog and Boolean logic

A new value is assigned to the left hand side whenever any value on the right hand side changes

en	i0	i1	y3	y2	Y1	y0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Behavioral Style

```
module V2to4decb( i0,i1,en,y0,y1,y2,y3 );
   input i0,i1,en;
   output y0,y1,y2,y3;
   reg y0,y1,y2,y3;

   always @ (en or i0 or i1)
   begin
     y0 = en & ~i0 & ~i1;
     y1 = en &  i0 & ~i1;
     y2 = en & ~i0 &  i1;
     y3 = en &  i0 &  i1;
   end
endmodule
```

- **Procedural code**
  - **But still very much in dataflow style**
- **Always block is key element of behavioral design**
- **Begin-end block groups sequences of procedural statements**
- **Note change of output to be reg variable – not a wire!**

Always Blocks

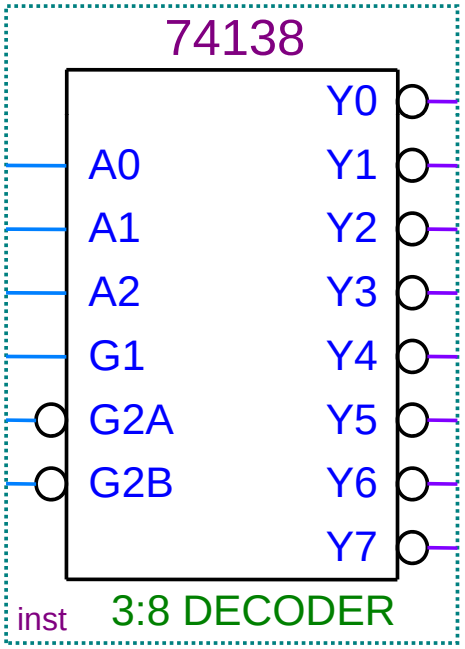
- **Always blocks start execution whenever the value of one of the signals in the *sensitivity list* changes**
  - **Continues executing until there are no more changes in sensitivity list**
- **Always blocks execute *concurrently* with other always blocks, instances, and continuous assignments**
  - **Allows mixing of structural, dataflow and behavioral styles**
- **Procedural statements within an always block execute sequentially**

Procedural Statements

- **Procedural statements act a lot like regular programming language statements**
  - **Blocking assignment**
    - ***variable name = expression*** **;**
  - **Nonblocking assignment**
    - ***variable name <= expression*** **;**
  - **Begin-end blocks**
    - **begin *procedural-statement* … *procedural-statement* end**
  - **If**
    - **if (*condition*) *procedural-statement* else *procedural-statement***
  - **Case**
    - **case (*sel-expr*) *choice* : *procedural-statement* … endcase**
  - **While**
    - **while (*logical-expression*) *procedural-statement***
  - **Repeat**
    - **Repeat (*integer-expression*) *procedural-statement***

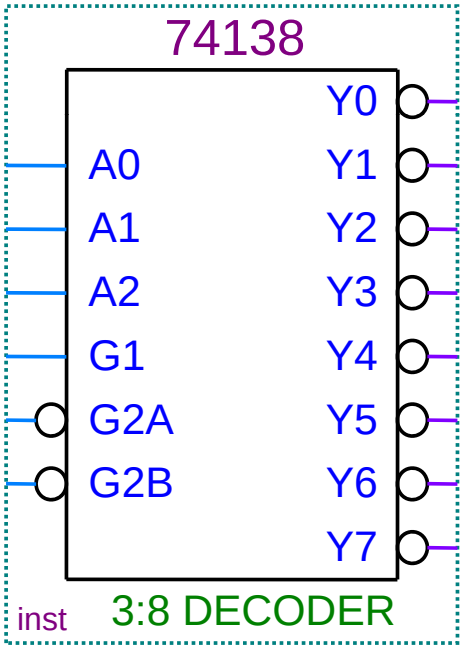
74x138 Decoder

```
module Vr74x138(G1, G2A_L, G2B_L, A, Y_L);
   input G1, G2A_L, G2B_L;
   input [2:0] A;
   output [0:7] Y_L;
   reg G2A,G2B;
   reg [0:7] Y_L, Y;

  <always block>

endmodule
```



74x138 Decoder

```
always @ (G1 or G2A_L or G2B_L or A or Y) begin
    G2A = ~G2A_L;  // convert inputs
    G2B = ~G2B_L;
    if ( G1 & G2A & G2B )
      case (A)
         0: Y = 8'b10000000;
         1: Y = 8'b01000000;
         2: Y = 8'b00100000;
         3: Y = 8'b00010000;
         4: Y = 8'b00001000;
         5: Y = 8'b00000100;
         6: Y = 8'b00000010;
         7: Y = 8'b00000001;
         default: Y = 8'b00000000;
       endcase
    else Y = 8'b00000000;
    Y_L = ~Y;          // convert outputs
  end
```



Blocking vs Nonblocking

- **Blocking statement**
  - **E.g., G2A = ~G2A_L;**
  - **Assignment is *immediate;* i.e., following statements are blocked until after the assignment**
- **Nonblocking statement**
  - **E.g., A <= B & C;**
  - **Assignment is *delayed* until after execution of always block is completed**

```
always
begin
 A = B;
 B = A;
end
```

A = B, B = B

```
always
begin
 A <= B;
 B <= A;
end
```

Swap A, B

Coding Rules*

- Always use blocking assignments in always blocks intended to create combinational logic
- Always use nonblocking assignments in always blocks intended to create sequential logic
- Do not mix blocking and nonblocking assignments in the same always block
- Do not make assignments to the same variable in two different always blocks

*Very strongly suggested recommendations

Inferred Latches

- **You should *always* assign a value to a variable on every path through an always block**
- **If you don't, then**
  - **the simulator infers that you don't want the value of the variable to change,**
  - **and "infers a latch" to store the value of the variable for the next execution of the block**
- **Avoiding inferred latches**
  - **Always assign *default* values to *all* variables at the start of the always block**
  - **Multiple assignments to the same variable in an always block is allowed (*last* assignment used)**

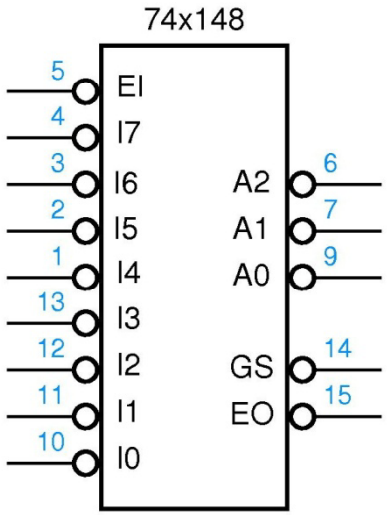
Sensitivity Lists

- It's easy to leave a net or variable out of the always block sensitivity list
- Simulator will simulate statements *as given*
  - Behavior become partially *sequential*, not purely *combinational*
- Synthesizer ignores the error
  - Synthesizes the intended combinational circuit
- You could end up with *different* results from the simulator and the synthesizer!
  - Verilog-2001 allows: `always @ *`

74x148 Priority Encoder

```
module Vr74x148(EI_L, I_L, A_L, EO_L, GS_L);
  input EI_L;
  input [7:0] I_L;
  output [2:0] A_L;
  output EO_L, GS_L;
  reg [7:0] I;
  reg [2:0] A,A_L;
  reg EI, EO_L, EO, GS_L, GS;
  integer j;

  always @ (EI_L or EI or I_L or I or A or EO or GS) begin
    EI = ~EI_L; I = ~I_L;                    // convert inputs
    EO_L = ~EO; GS_L = ~GS; A_L = ~A; // convert outputs
    EO = 1; GS = 0; A = 0;                   // default output values
    begin
      if (EI==0) EO = 0;
      else for (j=0; j<=7; j=j+1)
          if (I[j]==1)
          begin GS = 1; EO = 0; A = j; end
    end
  end
endmodule
```

Inputs									Outputs				
EI_L	I0_L	I1_L	I2_L	I3_L	I4_L	I5_L	I6_L	I7_L	A2_L	A1_L	A0_L	GS_L	EO_L
1	x	x	x	x	x	x	x	x	1	1	1	1	1
0	x	x	x	x	x	x	x	0	0	0	0	0	1
0	x	x	x	x	x	x	0	1	0	0	1	0	1
0	x	x	x	x	x	0	1	1	0	1	0	0	1
0	x	x	x	x	0	1	1	1	0	1	1	0	1
0	x	x	x	0	1	1	1	1	1	0	0	0	1
0	x	x	0	1	1	1	1	1	1	0	1	0	1
0	x	0	1	1	1	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0



Seven Segment Decoder

- Converts 4-bit input to HEX character

```
module seven_seg(B,Y);
  input [3:0] B;
  output [0:6] Y;
  reg [0:6] Y;

  …
endmodule
```

- Hex digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Bits of Y are drivers for segments a,b,c,d,e,f,g
- Segments are driven with a logic LOW

Seven Segment Display Architecture

Behavioral architecture

- Encodings of all 16 HEX digits

```
always @ (B)
  case (B)
     4'b0000 : Y = 7'b0000001;
     4'b0001 : Y = 7'b1001111;
     4'b0010 : Y = 7'b0010010;
     4'b0011 : Y = 7'b0000110;
     4'b0100 : Y = 7'b1001100;
     4'b0101 : Y = 7'b0100100;
     4'b0110 : Y = 7'b0100000;
     4'b0111 : Y = 7'b0001111;
     4'b1000 : Y = 7'b0000000;
     4'b1001 : Y = 7'b0001100;
     4'b1010 : Y = 7'b0001000;
     4'b1011 : Y = 7'b1100000;
     4'b1100 : Y = 7'b0110001;
     4'b1101 : Y = 7'b1000010;
     4'b1110 : Y = 7'b0110000;
     4'b1111 : Y = 7'b0111000;
     default : Y = 7'b1111111;
  endcase
```

Before Next Class

- **Read sections 8.1, 8.4-8.5**

Next time

Registers

Counters

Shift Registers