Procedural Blocks

Source material drawn from:

- Roy's ECE 351 and ECE 571 lecture material
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

Review: SystemVerilog procedural blocks

- `initial` block - Used to initialize behavioral statements for simulation
  - Executes once at simulation time 0
- General purpose always block:
  - `always @(<sensitivity list>)`
  - Can model many types of functionality both synthesizable and not synthesizable (ex: clock oscillators, verification code)
  - Downside is that simulators and synthesis tools cannot always determine when the intended usage is
    - Result: synthesis places coding restrictions so it can accurately translate RTL -> netlist
- Specialized always blocks:
  - `always_comb` – for combinational logic
  - `always_latch` – for latch-based (level) sequential logic
  - `always_ff @(<sensitivity list>)` – for edge-triggered sequential logic
  - Also place coding restrictions for synthesis but provide insight into intended usage

Sensitivity lists

- Always blocks tell simulation that functionality being modeled should "always" be evaluated (an infinite loop)
- Simulation and synthesis need more information to:
  - Accurately model hardware behavior
  - Know when to execute the statements in the block
  - For RTL modeling when is:
    - On a clock edge – sequential logic
    - Any of the signals used in the block change value – combinational or latched logic
- The sensitivity list provides the when
  - The list of signals that trigger the execution of the block when they change
  - General purpose `always` and `always_ff` require an explicit sensitivity list
  - `always_comb` and `always_latch` infer an implicit sensitivity list

Review: Explicit Sensitivity lists

- Explicit Sensitivity list
  - Introduced w/ @
  - Contains a list of one or more net or variable names separated by either a comma (,) or the keyword or
  - Can also specify the edge of a scalar (1-bit) signal using the keywords posedge or negedge
  - Required for general purpose always and always_ff
- Examples:

```
• always @(a, b, c) or always @(a or b or c) …
• always @(posedge clk or negedge rstN)…
• always_ff @(posedge clk)
    q <= d;    // sequential flip-flop
• always_ff @(posedge clk or negedge rstN)
   if (!rstN) q<= ‘0;
   else q <= d;
```

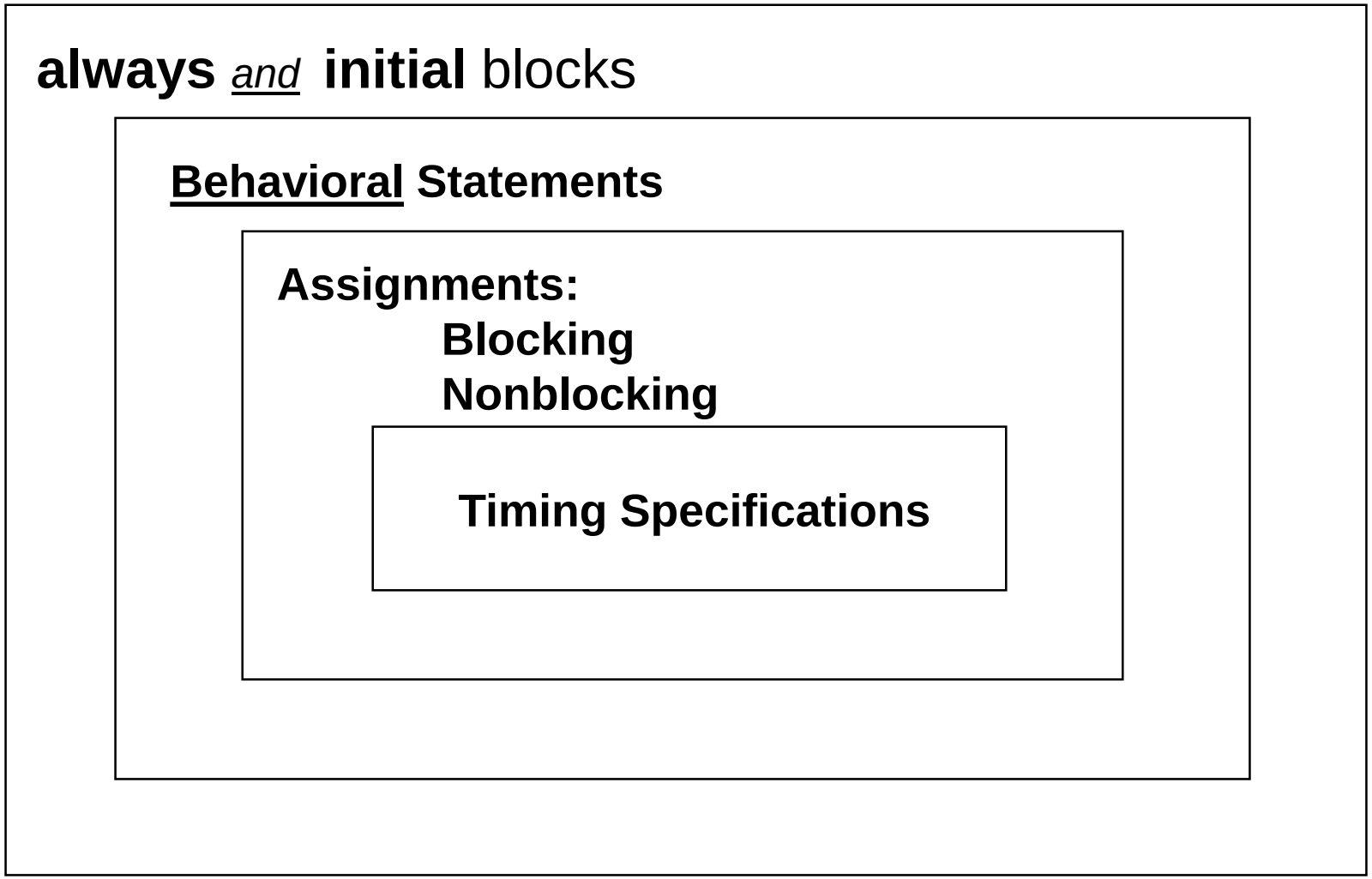
Review: Implicit Sensitivity lists

- Implicit Sensitivity list
  - Automatically inferred for `always_comb` and `always_latch`
  - Includes all RHS variables, expressions, etc.
  - Similar functionality to always @* (Verilog 2001)

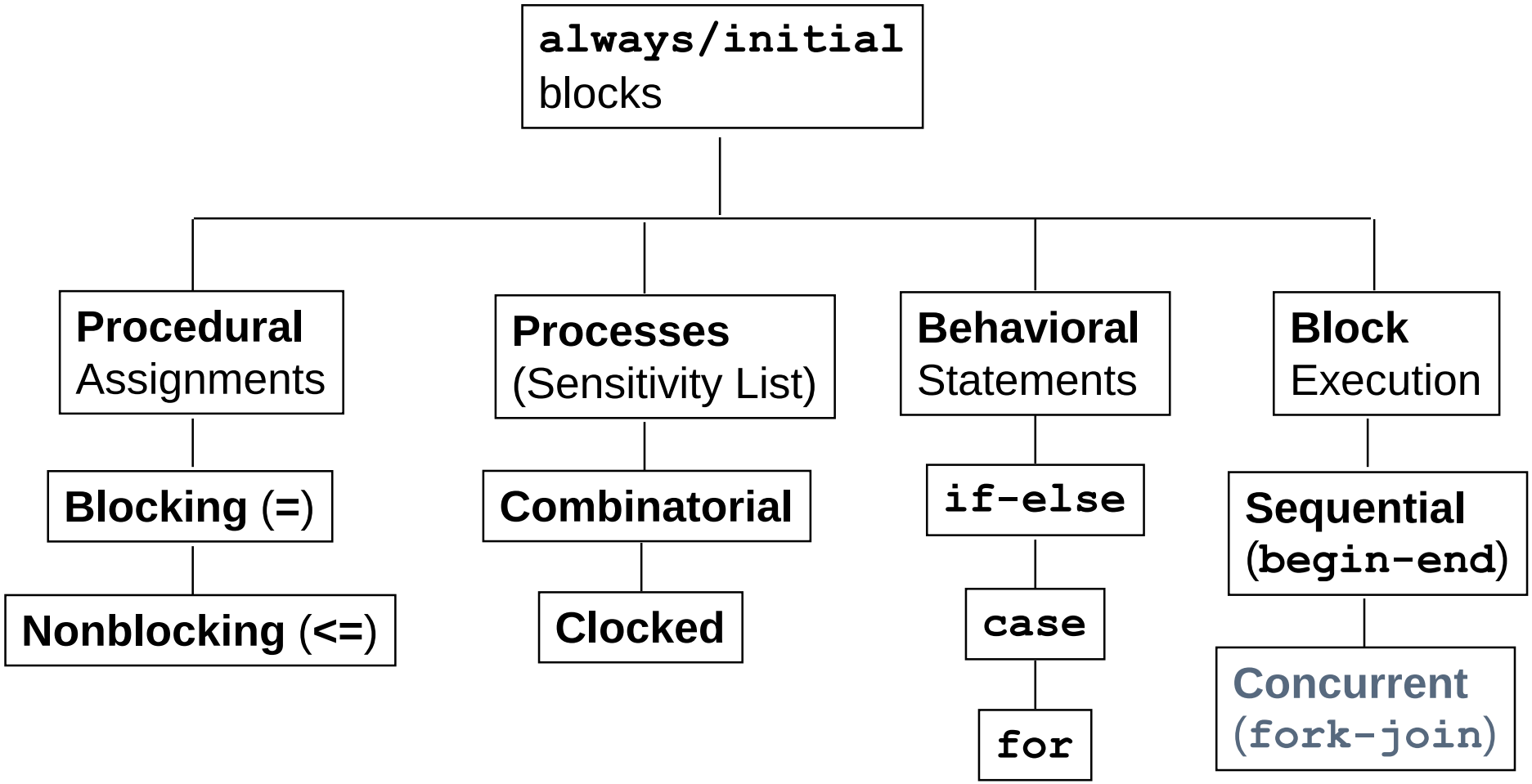
```
• always @(a, b)
    sum = a + b;
• always_comb
    sum = a + b;
• always @*
    sum = a + b;
```

```
• always @(enable, data)
    if (enable) out <= data;
• always_latch
    if (enable) out <= data;
• always @*
    if (enable) out <= data;
```

Elements of a Procedural Block



always and initial blocks



Statement groups

- All procedural block can contain either a single statement or a single group of statements
- Single statements can be nested within another statement

```
always @(posedge clk)
    if (enable)    // single outer statement
        for (int i; i <= 15; i++)    // nested statement
            out[i] = a[i] ^ b[(N - 1) – i];  // another nested stmt
```

- A group of statements is contained between the keywords `begin` and `end`
  - Can include any number of statements…including 1

```
always_comb begin //begin-end is the single group
    sum = a + b;
    dif = a – b;
end
```

Statement groups (cont'd)

- A `begin..end` group can be named using the syntax:

  `begin: <name>…end: <same name>`

- A named statement group can contain local variable declarations which can be used within the statement group but do not exist outside the group

```
always_ff @(posedge clk) begin: two_steps
       logic [7:0] tmp;      // local temporary variable
       tmp <= a + b;
       out <= c - tmp;
end: two_steps
```

Initial Block

- Consists of behavioral statements
- If there are more than one behavioral statement inside an `initial` block, the statements need to be grouped using `begin…end`
- An `initial` block starts at time 0, executes only once during simulation, and then does not execute again
- Assignment statements within an `initial` block execute sequentially - therefore statement order matters
- If there are multiple `initial` blocks, each block executes concurrently at time 0
- Used for initialization, monitoring, waveforms and other processes that must be executed only once during simulation

Initial Block Example

```
module system;
logic a, b, c, d;

// single statement
initial
        a = 1'b0;

/* multiple statements
   need to be grouped */
initial
    begin
            b = 1'b1;
       #5   c = 1'b0;
       #10  d = 1'b0;
    end

initial
    #20  $finish;
endmodule
```

Time	Statement Executed
0	a = 1'b0; b = 1'b1;
5	c = 1'b0;
15	d = 1'b0;
20	$finish

Why?

Always Block

- Used to model a process that is repeated continuously in a digital circuit…but you knew that
- If there are more than one behavioral statement inside an `always` block, the statements need to be grouped using `begin…end`
- An `always` block starts at time 0 and executes the behavioral statements continuously in a looping fashion
- Assignment statements inside an `always` block execute sequentially
  - Therefore, like an `initial` block, statement order matters*

* *sort of…*

Always Block - Example

```
module clock_gen (output logic clk);

parameter period=50, duty_cycle=50;

initial
   clk = 1'b0;

always
   #(duty_cycle*period/100)  clk = ~clk;

initial
   #100  $finish;

endmodule
```

Time	Statement Executed
0	clk = 1'b0
25	clk = ~clk // clk=1
50	clk = ~clk // clk=0
75	clk = ~clk // clk=1
100	$finish

Procedural Assignments

- Update values of variable types including user-defined types based on variables
  - Assigning to a net type (wire, tri, etc.) is illegal inside a procedural block
- The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value
- Only the LHS of a procedural assignment must be a variable. The RHS of assignments can use variables, nets, parameters or literal values

```
wire [15:0] a, b;      // net types
logic [15:0] sum;      // variable types

always_comb begin: adder
    sum = a + b;   // sum must be a variable type
end: adder
```

Two types of procedural assignments

- Blocking Assignment (=) : executed in the order they are specified in a procedural block

- Non-blocking Assignment (<=) : allow scheduling of assignments without blocking execution of the statements that follow in a procedural block

Blocking vs. Nonblocking Assignments (cont'd)

- Blocking ("="): `y = a & b;`
  - Completes the assignment before moving on to the next statement
  - "Blocks" other assignments until the current assignment has completed
  - Results highly dependent on order of assignments

- Nonblocking ("<="): `y <= a & b;`
  - Does not "block" execution of other assignments
  - Evaluates the RHS at the beginning of a simulation "tick"
  - Schedules the LHS update for the end of the "tick"
  - Results less dependent on order of assignments

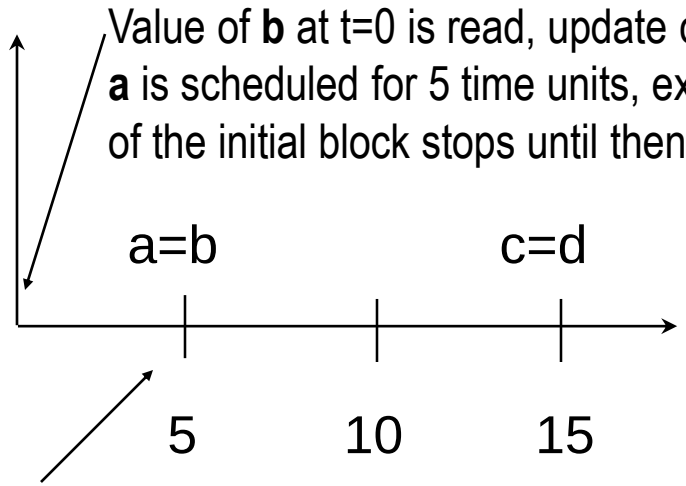
Blocking vs. Nonblocking Assignments (cont'd)

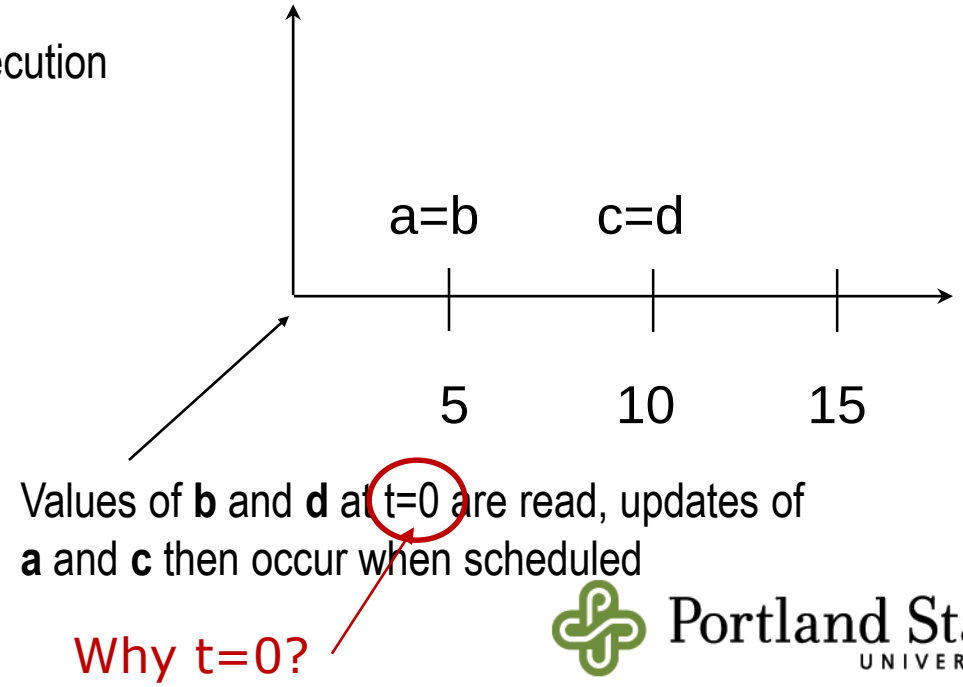
Blocking (=)

```
initial
    begin
       a = #5 b;
       c = #10 d;
    end
```

Nonblocking (<=)

```
initial
    begin
       a <= #5 b;
       c <= #10 d;
    end
```





Non-Blocking Assignments

- All right-hand sides across the entire design are evaluated before any left-hand sides are updated
- The order of right-hand sides evaluated and left-hand sides updated can be arbitrary
- Non-blocking assigns allow us to:
  - Handle concurrent specification in major systems
  - Reduce the complexity of our descriptions
  - Attach lots of concurrent actions to a single event – usually a clock

```
always @(posedge clock) begin
      q0 <= in | q1;
      q1 <= in & q0;
end
```

Blocking vs. Nonblocking Example

```
module nonblocking_example;

logic a = 0, b = 1;

// Two concurrent always blocks with blocking assignments
// always blocks run in any order…inconsistent results
always_ff @(posedge clock)
        a = b;
always_ff @(posedge clock)
        b = a;

// Two concurrent always blocks with nonblocking assignments
// works as expected because <= uses “old” values of b and a
always_ff @(posedge clock)
        a <= b;

always_ff @(posedge clock)
        b <= a;
endmodule: nonblocking_example
```

Which assignment should I use?

- Recommended:
  - Use non-blocking assignments for modeling clocked processes in sequential logic.
  - Use blocking assignments for modeling combinational logic
  - More on this later

Decision Statements

Source material drawn from:

- Roy's ECE 351 and ECE 571 lecture material
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

SystemVerilog `case`

Compares 0, 1, x, z by position

```
case (case_expression)
     case_item1 : case_item_statement1;
     case_item2 : case_item_statement2;
     case_item3 : case_item_statement3;
     case_item4 : case_item_statement4;
     default : case_item_statement5;
endcase
```

```
if (case_expression === case_item1) case_item_statement1;
     else if (case_expression === case_item2)
          case_item_statement2;
     else if (case_expression === case_item3)
          case_item_statement3;
     else if (case_expression === case_item4)
          case_item_statement4;
     else   case_item_statement5;
```

- The typical use of a case statement is to:
  - specify a variable as the case expression, and then list explicit values to be matched as the list of case selection items

SystemVerilog `case` (cont'd)

```
case (case_expression) // case statement header
  case_item_1 : begin
    case_statement_1a;
    case_statement_1b;
  end
  case_item_2 : case_statement_2;
  default     : case_statement_default;
endcase
```

```
case (case_expression)
   case_item1   :  case_item_statement1;
   case_item2   :  begin
                   case_item_statement2a;
                   case_item_statement2b;
                   case_item_statement2c;
                   end
   case_item3   :  case_item_statement3;
   default      :  case_item_statement5;
endcase
```

- Case statement header
  - case, casez, casex keyword
- Case expression
  - constants (e.g. 1'b1), an expression that evaluates to a constant, or a vector
- Case item
  - the expression that is compared against the case expression.
  - C-style break is implied following each case item statement
- Case item statement
  - one or more statements that is executed if the case item matches the current case expression.
  - If more than one statement is required, they must be enclosed with begin…end
- Case default
  - optional, but can include statements to be executed if none of the defined case items match the current case expression

SystemVerilog `case` (cont'd)

The Verilog standard specifically defines that case statements must evaluate the case selection items in the order in which they are listed

- This infers that:
  - there is a priority to the case items
  - treated the same as in a series of if...else...if decisions
  - If there are overlapping case items the first one that matches is selected
- Software tools such as synthesis compilers will typically try to optimize out the additional logic required for priority encoding the selection decisions if the tool can determine that all of the selection items are mutually exclusive
- Simulation and synthesis might interpret case statements differently.
  - Improperly coded Verilog case statements can cause unintended synthesis optimizations or unintended latches which could easily lead to non-functional chips

Unique and Priority case

- SystemVerilog provides the `unique`, `priority`, and `inside` modifiers for case decisions.
- These modifiers are placed before the `case` keyword:

```
unique case
(<case_expression>)
   ... // case items
endcase
```

```
priority case
(<case_expression>)
   ... // case items
endcase
```

- Designed to address coding traps:
  - Give information to synthesis to aid optimization
  - Assertions (simulation error reporting mechanisms)
  - Imply that there is a case item for all the possible legal values that case expression might assume

unique case

- unique is an assertion which implies:
  - All possible values of case expression are in the case items
  - At simulation run time, a match must be found in case items
  - At run time, only one match will be found in case items
- unique guides synthesis:
  - It indicates that no priority logic is necessary
  - This produces parallel decoding which may be smaller/faster
- unique usage
  - Use when each case item is unique and only one match should occur
  - Using a ”default” case item removes the testing for non-existent matches, but the uniqueness test remains
- The unique modifier allows designers to explicitly specify that
  - the order of the case selection items is not significant
  - the selections are permitted to be evaluated in parallel
  - Software tools can optimize out the inferred priority of the selection order

priority case

- A `priority case` statement specifies that:
  - At least one case select expression must match the case expression when it is evaluated
  - If more than one case select expression matches the case expression when it is evaluated, the first matching branch must be taken
- Priority is an assertion which implies:
  - All possible values for case expression are in case items
  - Using a ”default” case item
    - causes the priority requirement to be dropped since all cases are available to be matched
    - indicates that more than one match in case item is OK
- Priority guides synthesis
  - Indicates that all other testable conditions are don’t cares and may be used to simplify logic which can produce logic which is possibly smaller and/or faster

priority case statements (cont'd)

```
module decoder (output logic [3:0] y,
                input logic [1:0] a);
always_comb
    priority case (a)
        0 : y = 1;
        1 : y = 2;
        2 : y = 4;
    endcase
endmodule
```

- **priority**: Full cases
- The remaining cases are covered implicitly by a default item with don't care values.

- Simulation: warning
- Synthesis: OK

priority case statements (cont'd)

```
module decoder (output logic [3:0] y,
                input logic [1:0] a);
always_comb
    priority case (a)
        0 : y = 1;
        1 : y = 2;
        2 : y = 4;
        default: y = 3;
    endcase
endmodule
```

- **priority**: Full cases
- The remaining cases are covered explicitly by the default case

- Simulation: OK, no warning.
- Synthesis: OK

Priority

- Priority is an assertion which implies:
  - All possible values for case expression are in case items
- Priority guides synthesis
  - It indicates that all other testable conditions are don't cares and may be used to simplify logic
  - This produces logic which is possibly smaller/faster
- Priority usage
  - Use to explicitly say that priority is important even though the Verilog case statement is a priority statement.
  - Using a "default" case item will cause priority requirement to be dropped since all cases are available to be matched.
  - Use of a "default" also indicates that more than one match in case item is OK.

case...inside

```
casez (case_expression) inside
 case_item1 : case_statement1;
 case_item2 : case_statement2;
 case_item3 : case_statement3;
 case_item4 : case_statement4;
 default : case_item_statement5;
endcase
```

One way masking:

case expression: ZXZ1X

case item : 0?1??

expression	Z	X	Z	1	X	
item	0	?	1	?	?	
Match (1) /No Match (0) Decision	0	1	0	1	1	0 & 1 & 0 & 1 & 1 = 0 (no match)

- case...inside
  - Only matches wildcards in the case items
  - any X or Z bits in the case expression are not masked
- allows mask bits to be used in the case items

casez

```
casez (case_expression)
 case_item1 : case_statement1;
 case_item2 : case_statement2;
 case_item3 : case_statement3;
 case_item4 : case_statement4;
 default : case_item_statement5;
endcase
```

- z: high-impedance
- x: unknown
- ? : don't care

- casez allows "z" (or "?") to be treated as don't care values in:
  - the case expression
  - the case item when doing case comparison
  - ex: a *case item* 2'b1? in **casez** can match *case expression* of 2'b10, 2'b11, 2'b1x, 2'b1z

Can do the same operation with case inside

casex

- casex
  - allows both “z” and “x” to be treated as don’t care values (?) in either the case expression and/or the case item when doing case comparison
  - any x or z bits in both the case expression and case items are masked out from the comparison.
- Everything for casez also applies for casex, plus “x” is now also a wildcard
- The propagated x’s can easily cause problems when combined with casex statements
- To avoid these problems, the recommendation is:
  - Do NOT use casex at all for synthesizable code.
  - May be useful for testbenches

If - Else Conditional Statements

- Used to make decisions (to execute or not to execute…that is the question) upon certain conditions
- Keywords are `if` and `else`
- Only permitted in initial and always blocks
- Three types of conditional statements
  - Type 1 - if, no else)
    - if (expression) <true statement>
  - Type 2 – if with one else statement
    - if (expression) <true statement>; else <false statement>;
  - Type 3 – Nested if-else-if statement
    - if (expression 1) <true statement 1>;
      else if (expression 2) <true statement 2>;
      else if (expression 3) <true statement 3>;
      else <default statement>;

If - Else Conditional Statements (cont'd)

- Operation
  - <expression> is evaluated
  - If it is true (1 or a non-zero value) the true-statement is executed
  - If it is false (zero) or ambiguous (x) the false-statement is executed
- <expression> can contain any operators
- True-statement and false-statement can each be a single statement or a block of multiple statements
  - Block of statements must be grouped with begin…end
  - A single statement does not need to be grouped but it doesn't hurt to surround it with begin…end

Unique and Priority if...else decisions

- The SystemVerilog `unique` and `priority` decision modifiers also work with if...else decisions.
- These modifiers can also reduce ambiguities with this type of decision, and can trap potential design errors early in the modeling phase of a design.
- The Verilog if...else statement is often nested to create a series of decisions.

unique if...else

```
logic [2:0] sel;
always_comb begin
unique if (sel == 3'b001) mux_out = a;
else if (sel == 3'b010) mux_out = b;
else if (sel == 3'b100) mux_out = c;
end
```

- In simulation, a series of if...else...if decisions will be evaluated in the order in which the decisions are listed
- To maintain the same ordering in hardware implementation priority encoded logic would be required
- The unique modifier indicates that the designer's intent is that the order of the decisions is not important. Software tools:
  - can optimize out the inferred priority of the decision order
  - will check decision sequence to ensure that all decision conditions in a series of if...else...if decisions are mutually exclusive

priority if...else

```
always_comb begin
priority if (irq0) irq = 4'b0001;
else if (irq1) irq = 4'b0010;
else if (irq2) irq = 4'b0100;
else if (irq3) irq = 4'b1000;
end
```

- The `priority` modifier indicates that the designer's intent is that the order of the decisions is important
- Because the model explicitly states that the decision sequence above should be evaluated in order all software tools should maintain the inferred priority encoding
- The `priority` modifier ensures consistent behavior from software tools
  - Simulators, synthesis compilers, equivalence checkers, and formal verification tools can all interpret the decision sequence in the same way
- a `priority if...else` must specify all conditions thus preventing unintentional latched logic

Next Time

- Topics:
  - Looping statements
  - Modeling combinational logic
- You should:
  - Read Sutherland Ch 6
- Homework, projects and quizzes
  - Homework #2 is due to D2L by 10:00 PM on Mon, 04-May
    - No assignments will be accepted after Noon on Tue, 05-May so we can go over solution in class before the exam
  - Midterm e-Exam will be 2:00 PM – 3:45 PM on Thu, 07-May