

Decision Statements in SV

Decision Statements

- The primary constructs for RTL modeling are the decision statements
 - if...else and case (with its wildcard variations).
- These decision constructs are
 - the **heart** of RTL modeling,
 - model combinational logic, latches, and flip-flops.
- Care must be taken to ensure that **if...else** and **case** generate the intended hardware.
- Failing to follow proper coding guidelines can cause simulation versus synthesis **mismatches**.
- SV language enhancements have been added to Verilog to help reduce or eliminate these mismatches.

Verilog case

```
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 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*.
- *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*

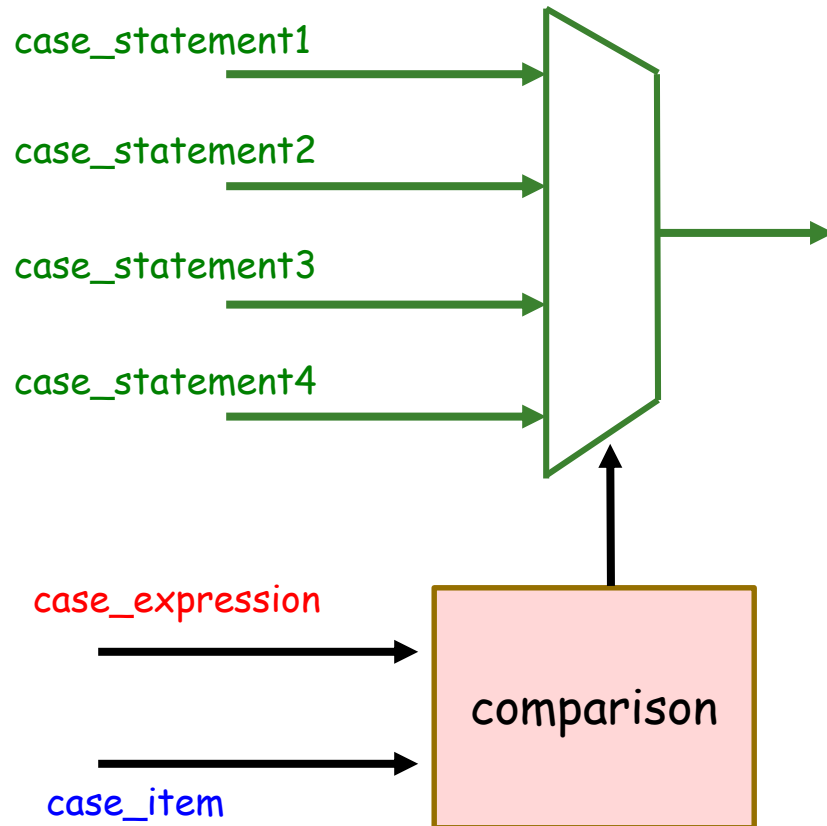
Verilog case

```
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**.

Hardware Model for case



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

Verilog case Semantics

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

- The Verilog 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,
 - the same as in a series of *if...else...if* decisions.
- *It allows overlapping case items.*
- *Case is already a priority structure.*

Mismatches between Simulation and Synthesis

- Simulation and synthesis might interpret case statements differently.
- Improperly coded Verilog **case** statements can frequently cause unintended synthesis optimizations or unintended latches.
- These problems, if not caught in pre-silicon simulations or gate level simulations, can easily lead to a non-functional chip.

Verilog Wildcard 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 modeling
 - "z" is treated as don't care values "?"
 - in both the **case expression** and the **case item**.
 - allows the use of the question mark (?) in place of z:
 - masked out !

- For example, a case item 2'b1z in **casez** can match case expression of 2'b10, 2'b11, 2'b1x, 2'b1z.

Matching Rules with casez

	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1



	0	1	x	z=?
0	1	0	0	1
1	0	1	0	1
x	0	0	1	1
z=?	1	1	1	1

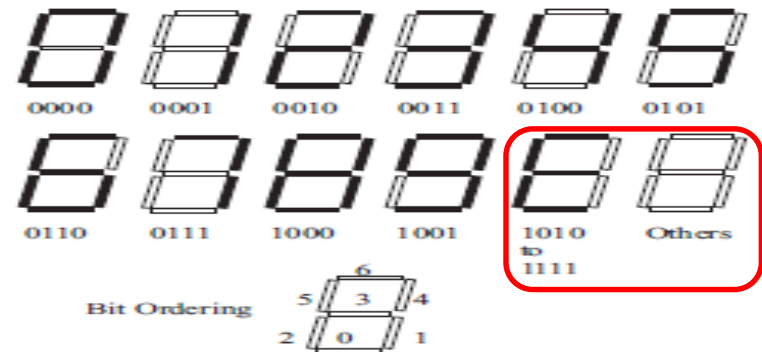
```

module sevenseg(output logic [6:0] data,
input logic [3:0] address);
always_comb
  casez (address)
    4'b0000 :      data = 7'b1110111;
    4'b0001 :      data = 7'b0010010;
    4'b0010 :      data = 7'b1011101;
    4'b0011 :      data = 7'b1011011;
    4'b0100 :      data = 7'b0111010;
    4'b0101 :      data = 7'b1101011;
    4'b0110 :      data = 7'b1101111;
    4'b0111 :      data = 7'b1010010;
    4'b1000 :      data = 7'b1111111;
    4'b1001 :      data = 7'b1111011;
    4'b101?, 4'b11?? : data = 7'b1101101;
    default :      data = 7'b0000000;
  endcase
endmodule

```

Seven-Segment Decoder with casez

Shorten the case items



casez: Two-Way Marking

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

- case expression: ZXZ10
- case item : 0?1ZX
- Two way masking:
- case expression: ?X?10
- case item : 0?1?X

expression	?	x	?	1	0	
item	0	?	1	?	x	
Decision	1	1	1	1	0	$1*1*1*1*0=0$

casez: Two-Way Marking

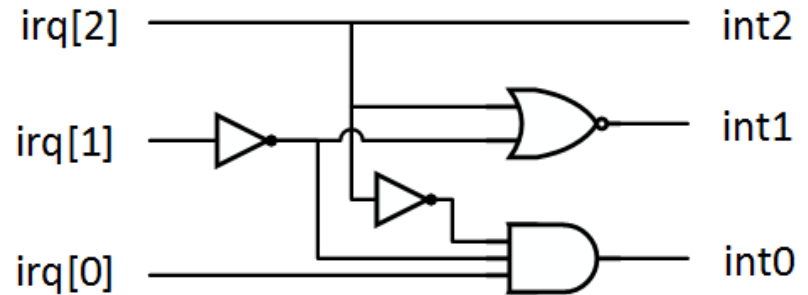
```
casez (case_expression)
  case_item1 : case_statement1;
  case_item2 : case_statement2;
  case_item3 : case_statement3;
  case_item4 : case_statement4;
  default : case_item_statement5;
endcase
```

- case expression: ZXZ1X
- case item : 0?1Z?
- Two way masking:
- case expression: ?X?1X
- case item : 0?1??

expression	?	x	?	1	X	
item	0	?	1	?	?	
Decision	1	1	1	1	1	1*1*1*1*1=1

Verilog casez with Overlaps

```
always @(irq) begin
  {int2, int1, int0} = 3'b000;
  casez (irq)
    3'b1?? : int2 = 1'b1;
    3'b?1? : int1 = 1'b1;
    3'b??1 : int0 = 1'b1;
    default: {int2, int1, int0} = 3'b000;
  endcase
end
```



- Verilog "wildcard" casez statements can have **overlapping case items**.
- If more than one case item can match a case expression,
 - **the first** matching case item has priority.
- Priority logic can be inferred from a case statement.

Casez May Mask Bugs

```
always @(irq) begin
  {int2, int1, int0} = 3'b000;
  casez (irq)
    3'b1?? : int2 = 1'b1;
    3'b?1? : int1 = 1'b1;
    3'b??1 : int0 = 1'b1;
    default: {int2, int1, int0} = 3'b000;
  endcase
end
```

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

- While wildcard case comparison can be useful, it also has its dangers.
- If the LSB of irq in the code snippet is unconnected
 - case expression = 3'b00Z,
- the third case item will still match and int0 will be set to 1,
 - potentially masking a bug!

SV: casez...inside

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



```
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
```

- **casez...inside**
 - ❑ uses a **one-way**, asymmetric masking for the comparison,
 - ❑ any X or Z bits in the **case expression** are not masked,
 - ❑ allows **mask bits in the case items**.
- Add more control, with less flexibility.

Casez...inside: Reduce Two-way To One Way

```
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
```

- case expression: ZXZ1X
- case item : 0?1Z?
- **One way** masking:
- case expression: ZXZ1X
- case item : 0?1??

expression	Z	x	Z	1	X	
item	0	?	1	?	?	
Decision	0	1	0	1	1	$0*1*0*1*1=0$

Even Wilder: casex in Verilog

- **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 a wildcard.
- The propagated x's can cause problems when combined with **casex**.
- Recommendation is
 - not to use **casex** at all for synthesizable code.
 - Can be useful for testbenches

Matching Rules with casex

	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

	0	1	x	z=?
0	1	0	0	1
1	0	1	0	1
x	0	0	1	1
z=?	1	1	1	1

	0	1	x=?	z=?
0	1	0	1	1
1	0	1	1	1
x=?	1	1	1	1
z=?	1	1	1	1

Example 1: Regular Verilog case

```
module decoder (output logic [3:0] y,  
                input logic [1:0] a);  
  
  always_comb  
  case (a)  
    0 : y = 1;  
    1 : y = 2;  
    2 : y = 4;  
    3 : y = 8;  
    default : y = 'x;  
  endcase  
endmodule
```

```
module decoder (output logic [3:0] y,  
                input logic [1:0] a);  
  
  always_comb  
  case (a)  
    0 : y = 1;  
    1 : y = 2;  
    2 : y = 4;  
    3 : y = 8;  
  endcase  
endmodule
```

- Simulation: ?
- Synthesis: ?

Example 1: Regular Verilog case

```
module decoder (output logic [3:0] y,  
                input logic [1:0] a);  
  
  always_comb  
  case (a)  
    0 : y = 1;  
    1 : y = 2;  
    2 : y = 4;  
    3 : y = 8;  
    default : y = 'x;  
  endcase  
endmodule
```

```
module decoder (output logic [3:0] y,  
                input logic [1:0] a);  
  
  always_comb  
  case (a)  
    0 : y = 1;  
    1 : y = 2;  
    2 : y = 4;  
    3 : y = 8;  
  endcase  
endmodule
```

- Simulation: different
- Synthesis: same

Example 2: Incomplete Verilog case

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

- Missing cases
- Latch introduced.

- Simulation: ?
- Synthesis: ?

Example 2: Incomplete Verilog case

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

- Missing cases
- Latch introduced.

- Simulation: **No warning.**
- Synthesis: latch introduced.

SV Enhancements

- SV provides special unique and priority modifiers to case, casex, and casez decisions.
- These modifiers are placed before the case, casex, or casez keywords:

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

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

- To address these coding traps.
 - Both give information to synthesis to aid optimization.
 - Both are assertions (simulation error reporting mechanisms)
 - Both imply that there is a case item for all the possible legal values that case expression might assume.

SV: unique case

- A unique case statement requires that:
 - **Completeness: Full**
 - All possible values of case expression are in the case items
 - Simulation: a match must be found in case items
 - Synthesis: a "default" case item removes the testing for non-existent matches
 - **Uniqueness: Exclusiveness, Disjoint**
 - Simulation: only one match will be found in case items
 - Synthesis: no priority, faster parallel decoding

Example 3: unique case

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

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

- Simulation: ?
- Synthesis: ?

Example 3 unique case

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

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

- Simulation: **warning** !
- Synthesis: OK

Example 3: unique case with Default

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

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

- Simulation: ?
- Synthesis: ?

Example 3: unique case with Default

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

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

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

Detecting Incomplete Case Items

- A unique case must specify all case items
- When a case, casex, or casez statement is specified as unique,
 - software tools will issue a run-time warning if
 - the value of the case expression does not match any of the case selection items, and
 - there is no default case.

Detecting Incomplete Case Items

```
logic [2:0] opcode; // 3-bit wide vector
always_comb
  unique case (opcode)
    3'b000: y = a + b;
    3'b001: y = a - b;
    3'b010: y = a * b;
    3'b100: y = a / b;
  endcase
```

- The example will result in a run-time warning if, during simulation, opcode has a value of 3, 5, 6 or 7.

SV: 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 item matches the case expression when it is evaluated,
 - the first matching branch must be taken

SV: priority case

- 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

SV: priority case

- 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.

SV: priority case

- Priority is a misleading name.
 - Verilog case statement itself is already a priority structure.
- Here **priority** keyword simply means: **full case !**

Example 4: priority case

```
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: ?
- Synthesis: ?

Example 4: priority case

```
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

Example 4: priority case with Default

```
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 implicitly by a default item with don't care values.

- Simulation: ?
- Synthesis: ?

Example 4: priority case with Default

```
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 implicitly by a default item with don't care values.

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

Observations

- By adding **unique/priority**,
 - we see more warning messages **early at simulation phase !**

Non-Overlap or Overlap (Disjoint or Non-Disjoint)

Complete and Disjoint: unique case

- The unique modifier specifies that the case selection items are **complete (or full)**.
- The unique modifier allows designers to explicitly specify that
 - the order of the case selection items is not significant, and
 - the selections are permitted to be evaluated in parallel.
- Software tools can **optimize out** the inferred priority of the selection order
- Any case expression value that occurs should match **one and only one** case select item.

Complete and Disjoint: unique case

```
always_comb  
unique case (opcode)  
    2'b00: y = a + b;  
    2'b01: y = a - b;  
    2'b10: y = a * b;  
    2'b11: y = a / b;  
endcase
```

- The example illustrates
 - that the case items are both mutually exclusive and
 - that all possible case select values are specified.
- The unique keyword documents and verifies that these conditions are true.

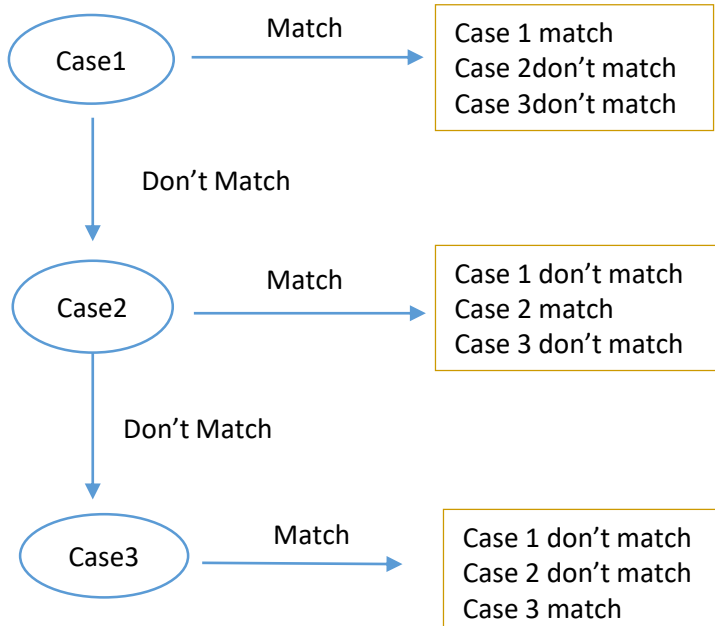
Checking for Disjoint Cases

- A unique case cannot have overlapping conditions.
- When a case, casex, or casez statement is specified as unique,
 - software tools must perform additional semantic checks to verify that each of the case selection items is mutually exclusive.
- If a case expression value occurs during run time that matches more than one case selection item,
 - the tool must generate a run-time warning message.

Observations on Non-disjoint Items

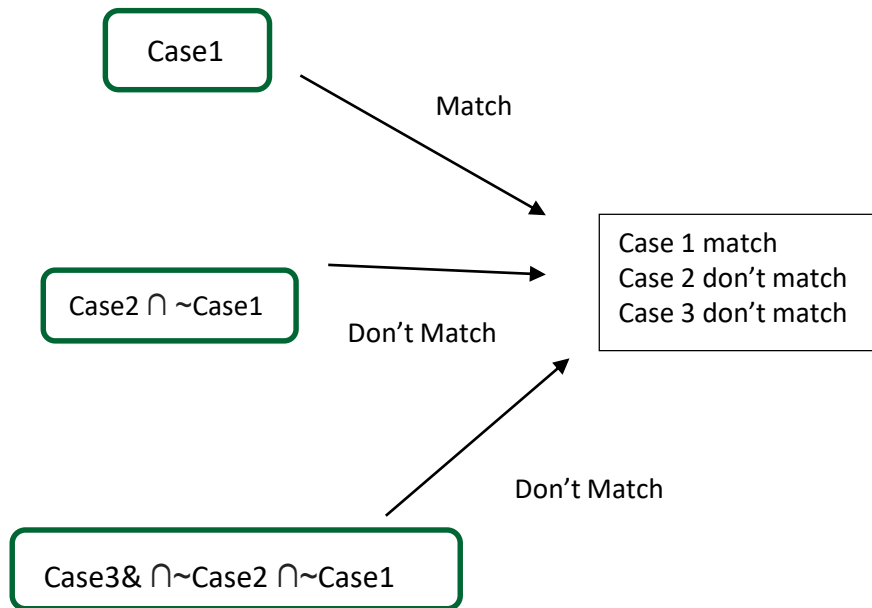
- In semantics, Verilog case statement allows case items to overlap.
 - Perform evaluation with priority,
 - when the condition is evaluated first, turn out match,
 - it will not continue evaluate even there are other possible cases match the condition.
- So the case statement can be divided into two groups.
 - overlap and no-overlap (independent).
 - In the synthesis step, the two groups can be realized differently.

Synthesis on Overlaps



- **Sequential implementation:**
 - As the case statement is overlapping, so the computation has to follow the user written order.
 - certain logic flow to follow.
 - Each circle is a combination decision logic.

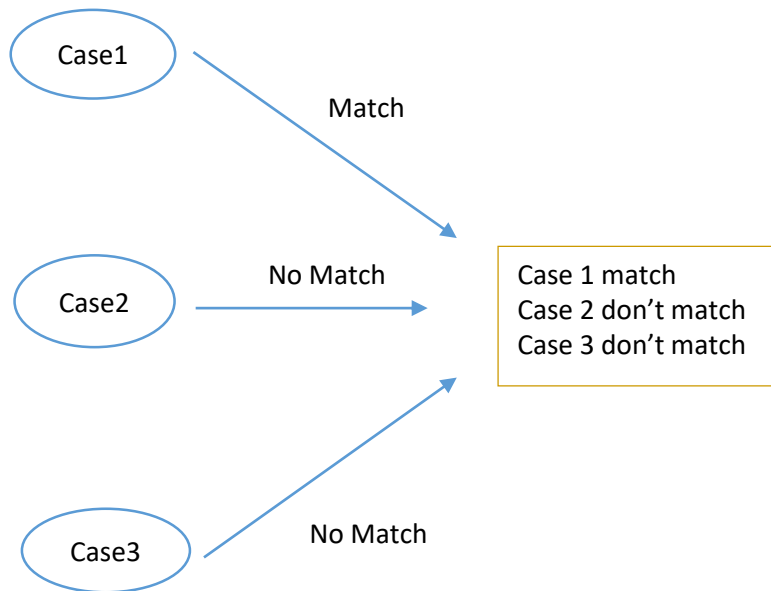
Synthesis on Overlaps



■ Parallel implementation:

- Since it is not efficient to wait for each decision block, so there is way to implement the same logic in parallel.
- Condition is fed in the all case block at the same time, by modify he case following the evaluation order, this logic block will generate the same result as last one, and guarantee there will be only 1 case match each time.

Observations



- When the unique case is provided, if the condition is not really exclusive, it may generate mismatch and synthesize incorrectly.

Verilog casez with Overlaps

- A casez statement is used to allow specific bits of the selection items to be excluded from the comparison with the case expression.
- When specifying don't care bits, it is easy to inadvertently specify multiple case selection items that could be true at the same time.

Verilog casez with Overlaps

```
logic [2:0] request;
always_comb
casez (request) // design should
    // only generate one
    // grant at a time
    3'b1??: slave1_grant = 1;
    3'b?1?: slave2_grant = 1;
    3'b???1: slave3_grant = 1;
endcase
```

- A casez statement is used to decode which of three bus request signals is active.
- The **designer's expectation** is that the design can **only issue one request at a time**.
- The casez selection allows comparing to one specific request bit, and masking out the other bits, which could reduce the gate-level logic needed.
- Since **only one request should occur at a time**, the order in which the 3 bits are examined should not matter, and there should **never be two case items true at the same time**.

casez with Overlaps

```
logic [2:0] request;
always_comb
casez (request) // design should
    // only generate one
    // grant at a time
    3'b1??: slave1_grant = 1;
    3'b?1?: slave2_grant = 1;
    3'b??1: slave3_grant = 1;
endcase
```

- The casez statement will compile for simulation without an error.
- If a case expression value could match more than one case selection item (**two requests occurred at the same time, for example**), then only the first matching branch is executed.
- Though the code is legal, **lint check programs and synthesis compilers** will generally warn that there is a potential overlap in the case items.
- However, these tools have no way to determine if the designer intended to have an overlap in the case select expressions.

SV: unique casez

```
logic [2:0] request;
always_comb
unique casez (request) // design should
    // only generate one
    // grant at a time
    3'b1??: slave1_grant = 1;
    3'b?1?: slave2_grant = 1;
    3'b??1: slave3_grant = 1;
endcase
```

- The unique modifier documents that
 - the designer **did not intend**, or expect, that **two case select items could be true at the same time**.
- When the unique modifier is added,
 - **all software tools**, including simulators, will generate a **warning** any time the case statement is executed and the case expression **matches multiple case items**.

Example A: casez with No Overlap

```
always_comb
begin
  casez(a1)
    4'b1??? : y1 = 2'b11;
    4'b01?? : y1 = 2'b10;
    4'b001? : y1 = 2'b01;
    4'b0001 : y1 = 2'b00;
    default : y1 = 2'b00;
  endcase
end
```

- All the cases are disjoint.
- No matter which order to be used, the result is the same.

- Simulation: No warning

Example B: casez with Overlap

```
always_comb
begin
  casez (a3)
    4'b1??? : y3 = 2'b11;
    4'b?1?? : y3 = 2'b10;
    4'b??1? : y3 = 2'b01;
    4'b???1 : y3 = 2'b00;
    default: y1 = 2'b00;
  endcase
end
```

- The cases are overlapping.
- They are allowed in Verilog.
- The operational semantics:
 - Executed in the written priority order.

- Simulation: No warning.

Example C: unique casez with Overlap

```
always_comb
begin
  unique casez(a2)
    4'b1??? : y2 = 2'b11;
    4'b?1?? : y2 = 2'b10;
    4'b???1? : y2 = 2'b01;
    4'b???1 : y2 = 2'b00;
    default : y1 = 2'b00;
  endcase
end
```

- The keyword unique means:
 - The cases are disjoint (parallel)
 - All cases are listed. (full)
- Reality:
 - The cases are overlapping.

- Simulation:
 - Warning: More than one conditions match in 'unique case' statement.

Example D: priority casez with Overlap

```
always_comb
begin
  priority casez(a4)
    4'b1??? : y4 = 2'b11;
    4'b?1?? : y4 = 2'b10;
    4'b???1? : y4 = 2'b01;
    4'b???1 : y4 = 2'b00;
    default: y1 = 2'b00;
  endcase
end
```

- The keyword **priority** means:
 - All cases are listed. (full)
- **Priority is a misleading name:**
 - No special priority meaning, since the implicit execution order is based on the written priority order anyway.

- Simulation: No warning.

Synthesis on priority case

- Because the model explicitly states that case selection items should be evaluated in order,
 - all software tools must maintain the inferred priority encoding, should it be possible for multiple case selection items to match.
- Some synthesis compilers might automatically optimize priority case statements to parallel evaluation if the compiler sees that the case selection items are mutually exclusive.

Four Models for Priority Decoder

always_comb	always_comb	<table><tr><td>0000</td><td>00</td></tr><tr><td>0001</td><td>00</td></tr><tr><td>0010</td><td>01</td></tr><tr><td>0011</td><td>01</td></tr><tr><td>0100</td><td>10</td></tr><tr><td>0101</td><td>10</td></tr><tr><td>0110</td><td>10</td></tr><tr><td>0111</td><td>10</td></tr><tr><td>1000</td><td>11</td></tr><tr><td>1001</td><td>11</td></tr><tr><td>1010</td><td>11</td></tr><tr><td>1011</td><td>11</td></tr><tr><td>1100</td><td>11</td></tr><tr><td>1101</td><td>11</td></tr><tr><td>1110</td><td>11</td></tr><tr><td>1111</td><td>11</td></tr></table>	0000	00	0001	00	0010	01	0011	01	0100	10	0101	10	0110	10	0111	10	1000	11	1001	11	1010	11	1011	11	1100	11	1101	11	1110	11	1111	11	always_comb
0000	00																																		
0001	00																																		
0010	01																																		
0011	01																																		
0100	10																																		
0101	10																																		
0110	10																																		
0111	10																																		
1000	11																																		
1001	11																																		
1010	11																																		
1011	11																																		
1100	11																																		
1101	11																																		
1110	11																																		
1111	11																																		
begin	begin		begin																																
casez(a1)	casez(a3)		priority casez(a4)																																
4'b1??? : y1 = 2'b11;	4'b1??? : y3 = 2'b11;		4'b1??? : y4 = 2'b11;																																
4'b01??? : y1 = 2'b10;	4'b?1??? : y3 = 2'b10;		4'b?1??? : y4 = 2'b10;																																
4'b001? : y1 = 2'b01;	4'b???1? : y3 = 2'b01;		4'b???1? : y4 = 2'b01;																																
4'b0001 : y1 = 2'b00;	4'b????1 : y3 = 2'b00;		4'b????1 : y4 = 2'b00;																																
default : y1 = 2'b00;	default : y1 = 2'b00;		default : y1 = 2'b00;																																
endcase	endcase		endcase																																
end	end		end																																

- All the four models give the same simulation results as a priority decoder.

Synthesis of Example A: casez with No Overlap

```

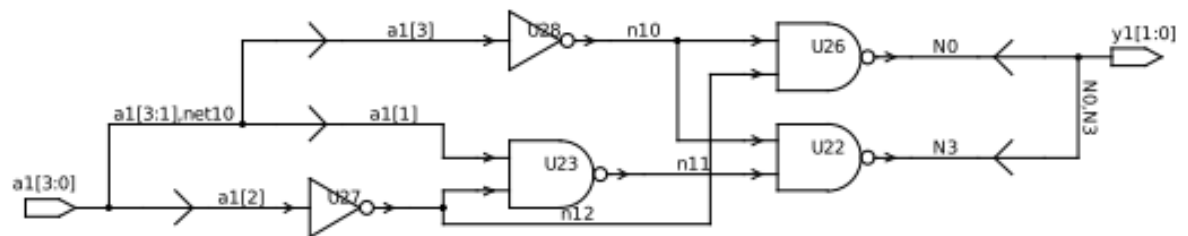
always_comb
begin
  casez(a1)
    4'b1??? : y1 = 2'b11;
    4'b01?? : y1 = 2'b10;
    4'b001? : y1 = 2'b01;
    4'b0001 : y1 = 2'b00;
    default : y1 = 2'b00;
  endcase
end

```

- All the cases are **disjoint**.
- $Y[0] = a[1]a[2]' + a[3]$
- $Y[1] = a[2] + a[3]$

0000	00
0001	00
0010	01
0011	01
0100	10
0101	10
0110	10
0111	10
1000	11
1001	11
1010	11
1011	11
1100	11
1101	11
1110	11
1111	11

a3a2a1a0	y1y0
000x	00
001x	01
010x	10
011x	10
100x	11
101x	11
110x	11
111x	11



Synthesis of Example B: casez with Overlap

0000	00
0001	00
0010	01
0011	01
0100	10
0101	10
0110	10
0111	10
1000	11
1001	11
1010	11
1011	11
1100	11
1101	11
1110	11
1111	11

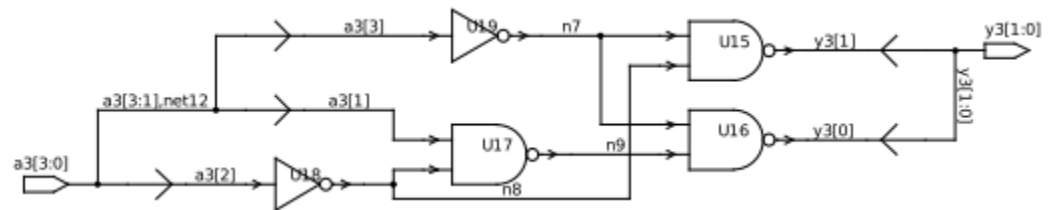
```

always_comb
begin
  casez(a3)
    4'b1??? : y3 = 2'b11;
    4'b?1?? : y3 = 2'b10;
    4'b??1? : y3 = 2'b01;
    4'b???1 : y3 = 2'b00;
    default : y1 = 2'b00;
  endcase
end

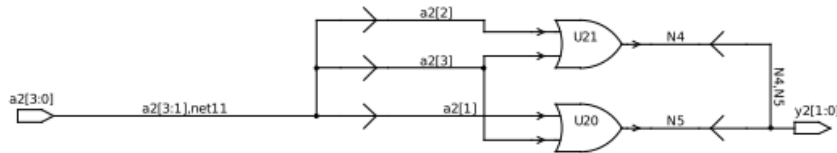
```

a3a2a1a0	y1y0
000x	00
001x	01
010x	10
011x	10
100x	11
101x	11
110x	11
111x	11

- The cases are **overlapping**.
- $Y[0] = a[2]'a[3]'a[1] + a[3] = a[1]a[2]' + a[3]$
- $Y[1] = a[2]a[3]' + a[3] = a[2] + a[3]$



Synthesis of Example C: unique casez with Overlap



```
always_comb
```

```
begin
```

```
unique casez(a2)
```

```
4'b1??? : y2 = 2'b11;
```

```
4'b?1??? : y2 = 2'b10;
```

```
4'b???1? : y2 = 2'b01;
```

```
4'b????1 : y2 = 2'b00;
```

```
default: y2 = 2'b00;
```

```
endcase
```

```
end
```

- Extracted logic:
- $Y[0] = a[1] + a[3]$
- $Y[1] = a[2] + a[3]$

```
always_comb
```

```
begin
```

```
casez(a4) // synopsys full_case parallel_case
```

```
4'b1??? : y4 = 2'b11;
```

```
4'b?1??? : y4 = 2'b10;
```

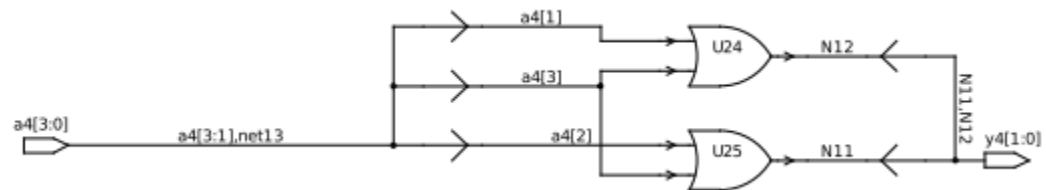
```
4'b???1? : y4 = 2'b01;
```

```
4'b????1 : y4 = 2'b00;
```

```
default: y4 = 2'b00;
```

```
endcase
```

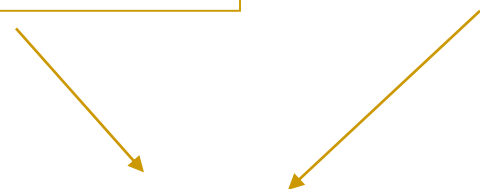
```
end
```



Synthesis of Example C: unique casez with Overlap

- Extracted logic:
- $Y[0] = a[1] + a[3]$
- $Y[1] = a[2] + a[3]$

- The cases are overlapping.
- $Y[0] = a[2]'a[3]'a[1] + a[3] = a[1]a[2]' + a[3]$
- $Y[1] = a[2]a[3]' + a[3] = a[2] + a[3]$



$a_3a_2a_1a_0$	y_1y_0
000x	00
001x	01
010x	10
011x	11
100x	11
101x	11
110x	11
111x	11

$a_3a_2a_1a_0$	y_1y_0
000x	00
001x	01
010x	10
011x	10
100x	11
101x	11
110x	11
111x	11

Example E: Overlap with/without unique

```
module casetest3 ( input [2:0] a1, output reg [2:0] y1,  
                  input [2:0] a2, output reg [2:0] y2,  
                  input [2:0] a3, output reg [2:0] y3,  
                  input [2:0] a4, output reg [2:0] y4);
```

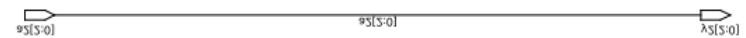
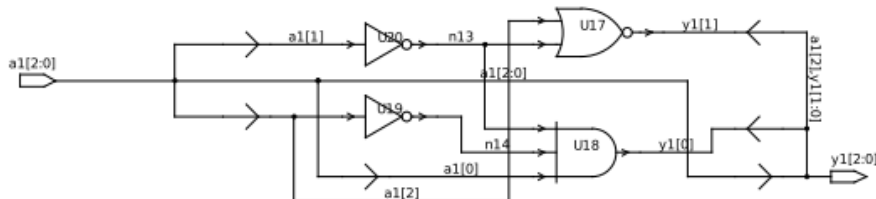
```
//cases overlap without unique  
always_comb  
begin  
    casez (a1)  
        3'b1?? : y1 = 3'b100;  
        3'b?1? : y1 = 3'b010;  
        3'b??1 : y1 = 3'b001;  
        default: y1 = 3'b000;  
    endcase  
end
```

```
//cases overlap with unique  
always_comb  
begin  
    unique casez (a2)  
        3'b1?? : y2 = 3'b100;  
        3'b?1? : y2 = 3'b010;  
        3'b??1 : y2 = 3'b001;  
        default: y2 = 3'b000;  
    endcase  
end
```

Example E: Overlap with/without unique

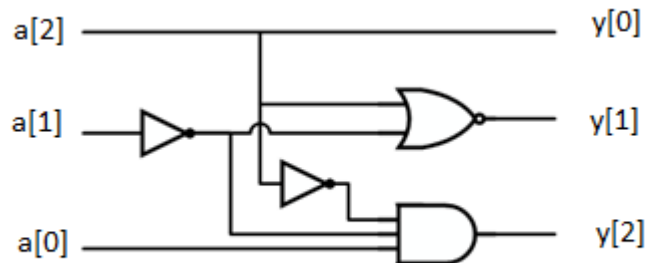
```
//cases overlap without unique
always_comb
begin
    casez (a1)
        3'b1?? : y1 = 3'b100;
        3'b?1? : y1 = 3'b010;
        3'b??1 : y1 = 3'b001;
        default: y1 = 3'b000;
    endcase
end
```

```
//cases overlap with unique
always_comb
begin
    unique casez (a2)
        3'b1?? : y2 = 3'b100;
        3'b?1? : y2 = 3'b010;
        3'b??1 : y2 = 3'b001;
        default: y2 = 3'b000;
    endcase
end
```

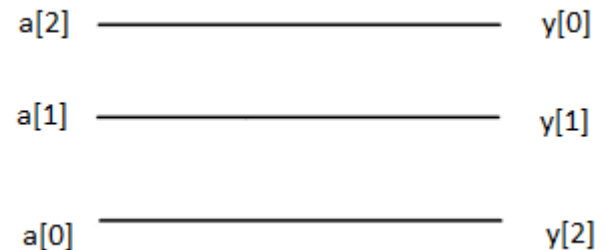


Example E: Overlap with/without unique

```
//cases overlap without unique
always_comb
begin
    casez (a1)
        3'b1?? : y1 = 3'b100;
        3'b?1? : y1 = 3'b010;
        3'b??1 : y1 = 3'b001;
        default: y1 = 3'b000;
    endcase
end
```



```
//cases overlap with unique
always_comb
begin
    unique casez (a2)
        3'b1?? : y2 = 3'b100;
        3'b?1? : y2 = 3'b010;
        3'b??1 : y2 = 3'b001;
        default: y2 = 3'b000;
    endcase
end
```

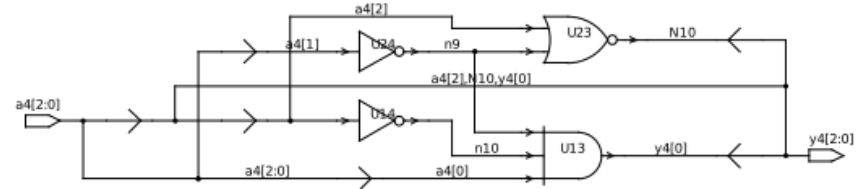
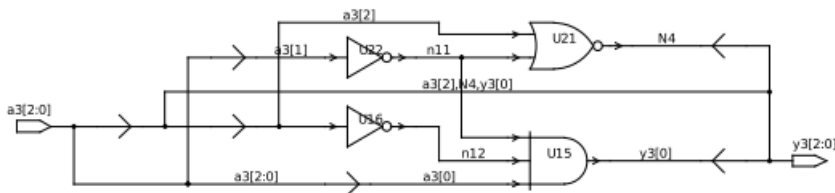


Example F: Non-Overlap with/without unique

```
//cases disjoint without unique
always_comb
begin
    casez (a3)
        3'b1?? : y3 = 3'b100;
        3'b01? : y3 = 3'b010;
        3'b001 : y3 = 3'b001;
        default: y3 = 3'b000;
    endcase
end
```

=

```
//cases disjoint with unique
always_comb
begin
    unique casez (a4)
        3'b1?? : y4 = 3'b100;
        3'b01? : y4 = 3'b010;
        3'b001 : y4 = 3'b001;
        default: y4 = 3'b000;
    endcase
end
```



Verilog Solutions versus SV Solutions

Verilog Pragmas for Synthesis

- A pragma is a special property **only used** by synthesis tool to guide synthesis process by modifying the behavior of synthesis compilers.
- Two different forms:
 - Comments:
 - Synthesis compilers allow pragmas to be **hidden within Verilog comments**.
 - Attributes:
 - The IEEE 1364.1 pragmas are specified using the Verilog **attribute** construct.

Comments: Synthesis Directives

- Comments can be used to state synthesis directives:
- `/* <comment> */`
- `// <comment>`
- In the comment, precede the synthesis directive with the synthesis keyword.

- You can also precede the synthesis attribute or directive with one of the following keywords:
 - Synopsys
 - pragma
 - exemplar
- Example:
 - `reg my_reg /* synthesis preserve */;`
 - `case /* synthesis full_case*/;`

Attributes: Synthesis Directives

- Attributes specify special properties of a Verilog object or statement, for use by specific software tools, such as synthesis.
- Attributes were added in Verilog-2001.

- Format
- (* begins an attribute, terminated by a *)
- Example:
- (* full_case, parallel_case *) case (state)
- ...
- endcase

parallel_case and full_case

- **Comments** for Two key directives:
 - `// synthesis full_case`
 - `// synthesis parallel_case`
- Only for the synthesis tool
- Not for simulators

- **Attributes** to express directives.
 - `(* full_case=1 *)`
 - `(* parallel_case=1 *)`
 - `case (a)`
 - `<rest_of_case_statement>`
- The attributes only apply to synthesis tools.
- **The intention** was to have simulators know them as well.
- However, it may not be the case so far.

Pragmas Versus Unique/Priority

- Initially, there was no way in Verilog to provide special synthesis requirement to synthesis tool,
 - so the EDA companies were using the comments with special keyword to pass on the synthesis information.
- Later, Verilog added attributes to passing on the information.
- Then they realized that some of the synthesis information is also important to the simulation (to narrow the semantic gap).
- **Equivalent language constructs** were added into SV such as **unique** and **priority**.

Synthesis parallel_case Pragma

```
always_comb
(* synthesis, parallel_case *)
case (opcode)
    2'b00: y = a + b;
    2'b01: y = a - b;
    2'b10: y = a * b;
    2'b11: y = a / b;
endcase
```

- One of the pragmas specified in the Verilog synthesis standard is parallel_case.
- This instructs synthesis compilers to **remove priority encoding**, and evaluate all case selection items in parallel.

Verilog Attributes: Parallel Case

- A parallel case:
 - each combination of inputs is covered exactly once by one case item.
- It is legal to write a case statement such that
 - an input pattern can match to two or more case items.
- Because the items are matched in the order they are written, this implies the existence of priority logic.
- Adding the parallel_case attribute forces the synthesis tool to treat the case statement as if it really is parallel.
- Inevitably, this will result in synthesized hardware that behaves differently to what was simulated at RTL.

Synthesis full_case Pragma

```
always_comb
(* synthesis, full_case *)
case (State)
    3'b001: NextState = 3'b010;
    3'b010: NextState = 3'b100;
    3'b100: NextState = 3'b001;
endcase
```

- Another pragma is full_case.
- This pragma instructs the synthesis compiler that, for all unspecified case expression values, the outputs assigned within the case statement are unused, and can be **optimized out by the synthesis compiler**.

Verilog: A Priority Encoder with Incomplete Cases

```
always @(a)
  casez (a)
    4'b1??? : y = 2'b11;
    4'b01?? : y = 2'b10;
    4'b001? : y = 2'b01;
    4'b0001 : y = 2'b00;
    default : y = 2'b00;
  endcase
```



```
always @(a)
  casez (a)
    4'b1??? : y = 2'b11;
    4'b01?? : y = 2'b10;
    4'b001? : y = 2'b01;
    4'b0001 : y = 2'b00;
    default : y = 2'b00;
  endcase
```

- If the default item were omitted, the pattern
 - ❑ 4b'0000
 - ❑ or 4b'000z
 - ❑ or any pattern that included an x
 - ❑ would not be matched.
- the case statement would not be full.

Verilog Attributes: Full Case

```
always @(a)
  casez (a)
    4'b1??? : y = 2'b11;
    4'b01?? : y = 2'b10;
    4'b001? : y = 2'b01;
    4'b0001 : y = 2'b00;
  endcase
```



```
always @(a)
  casez (a)
    4'b1??? : y = 2'b11;
    4'b01?? : y = 2'b10;
    4'b001? : y = 2'b01;
    4'b0001 : y = 2'b00;
    default : y = 2'b??;
  endcase
```

```
(* full_case *)
case (a)
  <rest_of_case_statement>
```

- By including the `full_case` attribute, the synthesis tool treats any unspecified combinations of inputs as don't care conditions
 - Namely, assume that a default item exists.
- Thus, the simulated and synthesized interpretations of the code would be different.
- If the default item is present, the `full_case` attribute is redundant!

full_case & parallel_case

```
// full_case applied to one-hot state machine
logic [3:0] state, next_state;
always_comb begin // next state logic decode
    next_state = '0; // latch prevention
    case (1'b1) // synopsys full_case parallel_case
        state[0]: next_state[1] = 1'b1;
        state[1]: next_state[2] = 1'b1;
        state[2]: next_state[3] = 1'b1;
        state[3]: next_state[0] = 1'b1;
    endcase
end
```

- This is a case statement in which the case expression is a constant, and the case items contain variables
- The most common application for an inverse case statement is decoding one hot states.

unique and priority Modifiers

- The relation between the new SV decision modifiers and the old synthesis directives:

full_case parallel_case version	SystemVerilog version
<code>case (...)</code> ... <code>endcase</code>	<code>case (...)</code> ... <code>endcase</code>
<code>case (...) // full_case</code> ... <code>endcase</code>	<code>priority case (...)</code> ... <code>endcase</code>
<code>case (...) // parallel_case</code> ... <code>endcase</code>	<code>unique case (...)</code> ... <code>default: ...</code> <code>endcase</code>
<code>case (...) // full_case parallel_case</code> ... <code>endcase</code>	<code>unique case (...)</code> ... <code>endcase</code>

- However, the SV modifiers affect both synthesis and simulation.

Unique and Priority Do More Than Synthesis Pragmas

- For synthesis, a unique case is equivalent to enabling both the `full_case` and `parallel_case` pragmas.
- A priority case is equivalent to enabling the `full_case` pragma.
- However, the SystemVerilog unique and priority decision modifiers do more than the `parallel_case` and `full_case` pragmas.
- These modifiers reduce the risk of mismatches between software tools, and provide additional semantic checks that can catch potential design problems much earlier in the design cycle.

Mismatches

- Synthesis pragmas
 - modify how synthesis interprets the Verilog case statements,
 - but they do not affect simulation semantics and might not affect the behavior of other software tools.
- This can lead to mismatches in how different tools interpret the same case statement.
- The unique and priority modifiers are part of the language, instead of being an informational synthesis pragma.
- As part of the language, simulation, synthesis compilers, formal verification tools, lint checkers and other software tools can apply the same semantic rules, ensuring consistency across various tools.

Prevent Mismatches

- The **unique** and **priority** modifiers are part of the language, instead of being an informational synthesis pragma.
- As part of the language, simulation, synthesis compilers, formal verification tools, lint checkers and other software tools can
 - apply the **same semantic rules**,
 - ensuring consistency across various tools.

Full or Incomplete Cases: Introducing Latches/FFs or Not?

Combinational Logic: No Latch/FF!

- **unique** = full and parallel cases
- **priority** = full cases
- By adding one of these two keywords, full case is guaranteed.
 - assigning the open case with don't care.
- *By using unique and priority, the latch will not be introduced.*
- Always true?

4 to 2 Priority Encoder

Inputs				Outputs		
A3	A2	A1	A0	Y1	Y0	Valid
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

```
module encoder (output logic [1:0] y, logic valid,
               input logic [3:0] a);
    always_comb
    unique casez (a)
        4b'1??? : {y,valid} = 3'b111;
        4b'01?? : {y,valid} = 3'b101;
        4b'001? : {y,valid} = 3'b011;
        4b'0001 : {y,valid} = 3'b001;
        default : {y,valid} = 3'b000;
    endcase
endmodule
```

- If the design were synthesized, however, there would be an ambiguity and the synthesis tool might attempt to impose its own priority.
- To avoid any ambiguity, it is good practice to qualify a **casez** statement with the **unique** modifier.
- If there is **an overlap**, an error would be flagged during compilation.
- We can reproduce the structure of the truth table, by making one assignment to y and valid simultaneously.
- Curly braces { } are used to concatenate variables, to express two separate variables as one vector.

Full Cases May Introduce FF

```
module Fsm (output logic OutA, OutB,  
            input Clock, Reset, InA, InB);  
enum{S0, S1, S2} PresentState, NextState;  
always_ff @(posedge Clock or posedge Reset)  
    if (Reset)  
        PresentState <= S0;  
    else  
        PresentState <= NextState;  
always_comb  
case (PresentState)  
    S0: begin  
        OutA = 1'b1;  
        if (InA)  
            NextState = S1;  
        else  
            NextState = S0;  
        end  
    S1: begin  
        OutA = InB;  
        OutB = 1'b1;  
        if (InA)  
            NextState = S2;  
        else  
            NextState = S1;  
        end  
    S2: begin  
        OutB = InA;  
        NextState = S0;  
        end  
endcase  
endmodule
```

```
S1: begin  
    OutA = InB;  
    OutB = 1'b1;  
    if (InA)  
        NextState = S2;  
    else  
        NextState = S1;  
    end  
S2: begin  
    OutB = InA;  
    NextState = S0;  
    end  
endcase  
endmodule
```

- This will, again, simulate as a state machine giving apparently correct behavior.
- When synthesized, however, **OutA and OutB will be registered through asynchronous latches**, because in state S0 no value is assigned to OutB and hence OutB holds onto its value.
- In S2, no value is assigned to OutA.
 - This should generate warnings.

Solutions

```
always_comb
begin
    OutA = 1'b0;
    OutB = 1'b0;
    case (PresentState)
    S0: begin
        OutA = 1b'1;
        if (InA)
            NextState = S1;
        else
            NextState = S0;
        end
    end
```

```
S1: begin
    OutA = InB;
    OutB = 1b'1;
    if (InA)
        NextState = S2;
    else
        NextState = S1;
    end
S2: begin
    OutB = InA;
    NextState = S0;
    end
endcase
end
```

- This error can be resolved by explicitly including an assignment to both OutA and OutB in every branch of the case statement.

- Alternatively, both signals can be given default values at the start of the procedure.
- This procedure now synthesizes to purely combinational logic, while the other procedure synthesizes to edge-triggered sequential logic.

Sequential Logic: Yes Latch/FF!

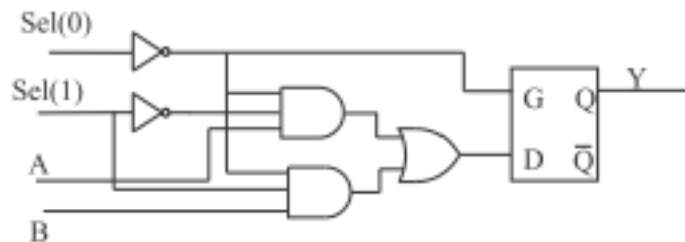
S	R	Q	\bar{Q}
0	0	1	1
0	1	0	1
1	0	1	0
1	1	Q	\bar{Q}

```
module rslatch2 (output logic q, qbar,  
                input logic r, s);  
always @(r, s)  
unique case ({r, s})  
    2'b00: {q, qbar} <= 2'b11;  
    2'b01: {q, qbar} <= 2'b10;  
    2'b10: {q, qbar} <= 2'b01;  
    default;  
endcase  
endmodule
```

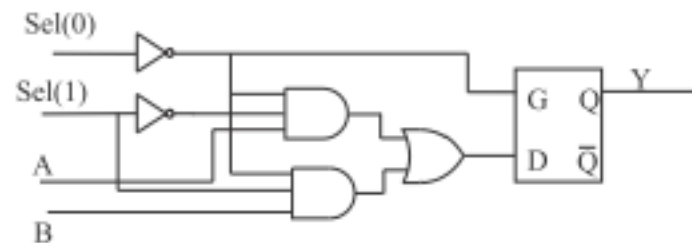
- In the first 3 branches of the **case**, values are assigned to q and $qbar$ depending on the combination of inputs.
- Nothing is assigned in the fourth, **default** branch,
 - so the q and $qbar$ values are retained.
 - the values are latched.
- If the module is synthesized, **a latch will be inferred**.
- The case statement is non-overlapping (**unique**) and that there is a default (in which nothing happens).

Sequential Logic: Yes Latch/FF!

```
always @(Sel, A, B)
  case (Sel)
    2'b00 : Y <= A;
    2'b10 : Y <= B;
    default;
  endcase;
```



```
always @(Sel, A, B)
  case (Sel)
    2'b00 : Y <= A;
    2'b10 : Y <= B;
  endcase;
```



Summary: unique/priority

- 1) Fundamental Issue: unique/priority
 - In combinational logic modeling, unique/priority can help reduce the chance of introducing the latches
- 2) Efficiency Issue: unique
 - Overlap (Dependent) versus Non-Overlap (Parallel)

SV: unique/priority

- **unique**
 - 1) overlapping cases:
 - **Mismatch** in synthesis, dependency is dropped, it is **forced** to be independent.
 - 2) disjoint cases:
 - with or without unique.
 - Synthesis decides.
- Verilog incomplete case:
 - No warning on simulation.
 - If a case is missing,
 - the previous value is kept in simulation
 - A latch is introduced in synthesis
 - **unique/priority**: the missing case is treated as don't care in synthesis.

Unique Case: Property 1

- First, unique case specifies that
 - all case selection items can be evaluated in parallel, without priority encoding.
- Software tools such as synthesis compilers can optimize the decoding logic of the case selection items to create smaller, more efficient implementations.
- This aspect of unique case is the same as synthesis `parallel_case` pragma.

Unique Case: Property 2

- Second, unique case specifies that
 - there should be no overlap in the case selection items.
- During the run-time execution of tools such as simulation, if the value of the case expression satisfies two or more case selection items, a run-time warning will occur.
- This semantic check can help trap design **errors early** in the design process.
- The synthesis `parallel_case` pragma **does not** provide this important semantic check.

Unique Case: Property 3

- Third, unique case specifies that
 - all values of the case expression that occur during simulation must be covered by the case selection items.
- With unique case, if a case expression value occurs that does not cause a branch of the case statement to be executed, a run-time warning will occur.
- This semantic check can also help trap design **errors much earlier** in the design cycle.
- This is similar to the `full_case` pragma for synthesis, but the synthesis pragma does not require that other tools perform any checking.

Decision Statements:

if...else

Enhanced if...else Decisions

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

- 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.

Simulation And Synthesis Might Interpret If...Else Differently

- 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.
- Often, however, the specific order is not essential in the desired logic.
- The order of the decisions is merely the way the engineer happened to list them in the source code.

A Unique If...Else Can Be Evaluated In Parallel.

```
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
```

- 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.

A Unique If...Else Cannot Have Overlapping Condition

- Software tools will perform checking on a **unique if** decision sequence to ensure that all decision conditions in a series of if...else...if decisions are **mutually exclusive**.
- This allows the decision series to be executed in parallel, without priority encoding.
- A software tool will generate a run-time warning if it determines that more than one condition is true.
- This warning message can occur at either compile time or run-time.

An Overlap In The Decision Conditions

```
logic [2:0] sel;  
always_comb begin  
    unique if (sel[0]) mux_out = a;  
    else if (sel[1]) mux_out = b;  
    else if (sel[2]) mux_out = c;  
end
```

- This additional checking can help detect modeling errors early in the verification of the model.
- Any or all of the conditions for the first, second and third decisions could be true at the same time.
 - This means that the decisions must be evaluated in the order listed, rather than in parallel.
- Because the unique modifier was specified, software tools can generate a warning that the decision conditions are not mutually exclusive.

A Unique If...Else Warns Of Unspecified Conditions

```
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
```

- When the **unique** modifier is specified with an if decision, software tools are required to generate a run-time warning if the if statement is evaluated and **no branch is executed**.
- The example would generate a run-time warning if the unique if...else...if sequence is entered and sel has any value other than 1, 2 or 4.

Enhanced if...else decisions

```
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
```

- This run-time semantic check guarantees that all conditions in the decision sequence that actually occur during run time have been fully specified.
- When the decision sequence is evaluated, one branch will be executed.
- This helps ensure that the logic represented by the decisions can be implemented as combinational logic, without the need for latches.

A Priority If...Else Must Evaluate In Order

```
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.

priority if...else must specify all conditions: Preventing unintentional latched logic

- As with the unique modifier, when the priority modifier is specified with an if decision, software tools will perform run-time checks that a branch is executed each time an if...else...if sequence is evaluated.
- A run-time warning will be generated if no branch of a priority if...else...if decision sequence is executed.
- This helps ensure that all conditions in the decision sequence that actually occur during run time have been fully specified, and that when the decision sequences are evaluated, a branch will be executed.
- The logic represented by the decision sequence can be implemented as priority-encoded combinational logic, without latches.

Synthesis Guidelines

- An if...else...if decision sequence that is qualified with unique or priority is synthesizable.
- A primary goal of SystemVerilog is to enable modeling large, complex designs more concisely than was possible with Verilog.

References

- "Synthesizing SystemVerilog" by S. Sutherland, D. Mills. 2013
- Sutherland, "SystemVerilog Saves the Day—the Evil Twins are Defeated! unique and priority" are the new Heroes", Synopsys Users Group (SNUG) conference, San Jose, California, March 2005.
- Mills and Sutherland, "SystemVerilog Assertions Are For Design Engineers Too!", Synopsys Users Group (SNUG) conference, San Jose, California, March 2006.
- Mills, "Being assertive with your X (SystemVerilog assertions for dummies)", Synopsys Users Group Conference (SNUG) San Jose, California, 2004.
- Mills, "Yet Another Latch and Gotchas Paper", Synopsys Users Group (SNUG) conference, San Jose, California, March 2012.
- Sutherland, "I'm Still in Love with My X!", Design and Verification Conference (DVCon), San Jose, California, February 2013.
- Greene, Salz and Booth, "X-Optimism Elimination during RTL Verification", Synopsys Users Group Conference (SNUG) San Jose, 2012.