Full and Parallel Case

CMPE 415 Programmable Logic Devices

Prof. Ryan Robucci

UMBC

DEPARTMENT OF COMPUTER SCIENCE AND ELECTRICAL ENGINEERING

References

- (Cummings 1999) Some examples and discussion are cited from Clifford E. Cummings' "full_case parallel_case", the Evil Twins of Verilog Synthesis http://www.sunburst-design.com/papers/CummingsSNUG1999Boston_FullParallelCase.pdf

Hardware Synthesis vs. Simulation

- Simulators don't interpret HDL code the same as synthesis tools
- Simulated behavior of unsynthesized, behavioral code is sometimes grossly mismatched to the hardware synthesized
- Many such mismatches can be avoided by following the prescribed coding guidelines
- Sometimes simulating post-synthesis HDL code is good idea. post-synthesis HDL code is the code output from the process of mapping all HDL code to structural code using the modules representing the building blocks of the target hardware technology (e.g. FPGA, standard-cell ASIC). The modules for those building blocks have modules provided which have already been coded specifically to stimulate closely the behavior of the final physical hardware
- Lets take a quick look at a couple examples where simulation and hardware synthesis lead to a different interpretation of HDL code

Ambiguous Parallel Blocks

```
// next two are not guaranteed an order
always @(posedge clk) begin
  y1 = a;
end

always @(posedge clk) begin
  y1 = b;
end

// next two are not guaranteed an order
always @(posedge clk) begin
  y2 = b;
end

always @(posedge clk) begin
  y2 = a;
end
```

Synthesis Report

```
    Register <y2> equivalent to <y1> has been removed
    Register <y2> equivalent to <y1> has been removed
    Found 1-bit register for signal <y1>.
    Summary:
     inferred   2 D-type flip-flop(s).
Unit <ambigous_parallel> synthesized.
```

Essentially, identifies four registers, two of them are the same, and the remaining two drive the same output

```
=========================================================================
HDL Synthesis Report

Macro Statistics
# Registers                                            : 2
 1-bit register                                        : 2

Advanced HDL Synthesis Report

Macro Statistics
# Registers                                            : 2
 Flip-Flops                                            : 2

=========================================================================
*                       Low Level Synthesis                             *
=========================================================================

ERROR:Xst:528 - Multi-source in Unit <ambigous_parallel> on signal <y2>;
                        this signal is connected to multiple drivers.
Drivers are:
   Output signal of FD instance <y1>
   Output signal of FD instance <y1_ren>
```

Non-conflicting parallel blocks

next two may be OK for **simulation**, but why?

```
always @(posedge clk) begin
  if (a == 1) y3= 1;
end

always @(posedge clk) begin
  if (a == 0) y3= 0;
end
```

Though this may simulate, the compiler will likely not be able to synthesize it, though again this is synthesizer dependent

Why not? Because a common approach is initially synthesize each procedural block independently and then infer connections.

Remember the rule: Avoid assigning to a variable from more than one always block

The Verilog Case Statement

- The structure of a case statements might suggest a parallel and independent evaluation of cases

```
case (case_select)
  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
```

- However, case statements are equivalent to if-else decision trees
  - Logic is implied by a priority that is expressed by the order or the statements

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

(Cummings 1999)

Notes on Case Statements

Order of case_items sets priority (first match wins), therefore the order encodes meaning/logic

- Be very attentive to the order of the cases in a case statement.

Example: Assume you are standing outside your door considering an afternoon in the park or a return inside

Decision Algorithm: If **see lightning** go inside
Else if **see rain** walk to park with umbrella
Else run to park

Taking in isolation the latter part of the second line, we could conclude an intent to

"walk to the park if we **see rain**"

The algorithm does not intend this. It does tells us to

"walk to the park if we **see rain** **AND** we **do not see lighting**"

- Conditions presented as case items in case statements must be interpreted in the same manor, with attention to the order and precedence

Critical Design Considerations for always blocks

- Consider **every** output individually for **every** traversal path of a the decision tree.

  Stated otherwise: Consider every output for every input-and-state combination.

- Note if the signal is being used externally to the block it is assigned. This can determine how the signal is interpreted.
  - Our coding rule is that if we are coding an clocked always block, then any signal assigned using a blocking assign should not be used outside the block so that the registered version of the signal is trimmed and only the combinatorial version remains. We enforce this by using named blocks and local variables for the output of combinatorial circuits
- What happens when there are conditions within states in which outputs or variables are not assigned?...

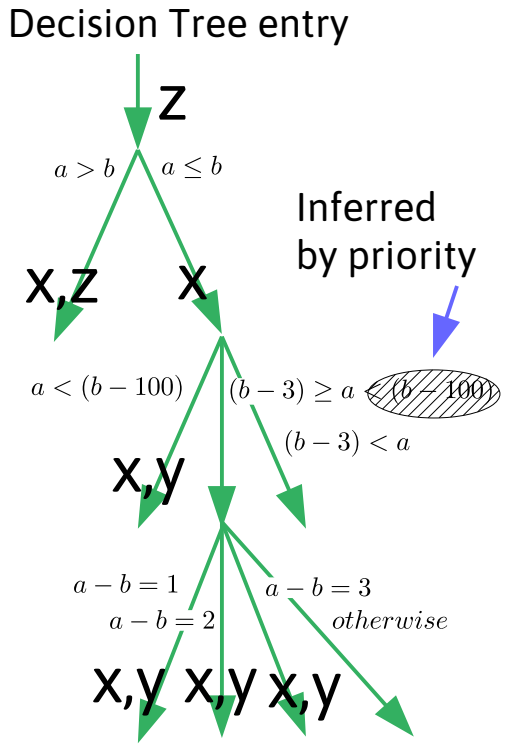
Complete Assignment for Combinatorial Functions

The definition of a combinatorial function must have an assignment to every variable for all possible cases. Does your code cover all the cases? This must be assessed for each and every output variable.

```
z = 1;
if (a > b) begin
   x = 1;
   z = 4;
end else begin
   x=0;
   if (a < (b-100)) begin
      x = 2; y=1;
   else if (a>=(b-3)) begin
      case(a-b)
         1: begin x = 3; y=1; end
         2: begin x = 3; y=2; end
         3: begin x = 3; y=3; end
      endcase
   end
end
```

Draw a decision tree, label conditions for branching carefully including branches for default decisions.

Mark where variables are assigned. If a tree can be traversed to a leaf without assigning every output variable then it is not complete.



Tip: flattening your decision hierarchy can avoid mistakes

Unintentional and Intentional Latches

- In a combinatorial block any input condition in which any output is not assigned may result in an inferred latch.

Latches

```
always @(x,en)
  if (en==1) y = x;

always @(a)
  if (a>1)   y=a;
```

Combinatorial

```
always @(x,en)
  if (en==1) y = x;
  else       y = 0;

always @(a)
   if (a>1)  y=a;
   else      y=0;
```

- Avoid the use of latches without consulting the user good for recommended practices, coding style, and precautions
- Assess any report of an inferred latch and investigate if it was truly intentional

Register Enables

- Inside a sequential-trigger block, not specifying an output may generate an enable on that output register unless the finite state machine extractor and synthesizer figures out how to avoid it by repeating logic for you in certain states. Functionally, not specifying an output in a condition in a state means the previous value should be held.

Examples which potentially generate a register w/ Enable

```
always @(posedge clk)
   if (en==1)
        y<=x;

always @(posedge clk)
   if (a<10)
        y<=a;
   else if (a>20)
        y<=20;
```

No Enable

```
always @(posedge clk)
   if (a<10)
        y<=a;
   else if (a>20)
        y<=20;
   else
        y<=10;
```

Don't Care using 'x'

- What if output not provided for every input condition?
  - If Edge Triggered, an enable may be added to registers
  - If Combinatorially triggered, Latches created
- Assign outputs to be 'x' to avoid extra enables or latches respectively, this encodes a don't care condition for an output.

Potentially Generates a Register with Enable

```
always
  @(posedge clk)
   if (a<10)
        y<=a;
   else if (a>20)
        y<=20;
```

No Enable

```
always
 @(posedge clk)
   if (a<10)
        y<=a;
   else if (a>20)
        y<=20;
   else
        y<=8'bx;
```

Latches

```
always @(x,en)
  if (en==1) y = x;

always @(a)
  if (a>1)    y=a;
```

Combinatorial Blocks

```
always @(x,en)
  if (en==1) y = x;
  else       y = 8'bx;

always @(a)
   if (a>1)   y=a;
   else       y=8'bx;
```

Registered Logic

- Implementing combinatorial logic signals in a edge triggered block may create "registered logic" if the signal is used outside the block (if not used outside the block, Xilinx will report "trimming" of the register)

•Combining combinatorial logic in an edge-triggered block limits you to only implement registered logic outputs. You can't have a combinatorial output from an edge-triggered block, though you can have internal combinatorial logic

•In the example below, if c is used outside the block a register will be created for it, though it's combinatorial version survives internally to immediately propagated through the remaining calculations. If c is not used elsewhere, the register is trimmed. Some do not recommend this practice.

```
always @(posedge clk)
   c=a+b;
   d=c+3;
   q<=d;
```

•We'll discuss this more later, but for now our coding guidelines dictate that we not create registers using blocking statements. If it is trimmed, we are still following this rule.
So, **signals <u>assigned in sequentially triggered block using a blocking statement</u> should not be used outside the block they are assigned**

Simple case statement

```
wire lighting; //see lightning
wire rain;     // feel rain
wire sun;      // sun shining
reg go;        // flag decision goto park
reg run;       //flag decision run
reg umbrella;  //flag decision to use umbrella
.
.
.

case ({lighting,rain,sun})
   {3'b110} : go =0; run = 0; umbrella=0;
   {3'b010} : go =1; run = 0; umbrella=1;
   {3'b001} : go =1; run = 1; umbrella=0;
endcase
```

As we will see, this is oversimplified

Inferred Latch

```
case ({lighting,rain,sun})
   {3'b110} : go =0; run = 0; umbrella=0;
   {3'b010} : go =1; run = 0; umbrella=1;
   {3'b001} : go =1; run = 1; umbrella=0;
endcase
```

- Many “physical” cases of {lighting,rain,sun} are not covered above and thus a physical latch may be inferred for synthesis.
- Even if all physical cases are covered, in simulation lighting, rain and sun can take on values x and z

  Meaning a latch is inferred for simulation

  i.e. if **lighting**, **rain**, or **sun** is equal to x or z

  do nothing (hold the previous value)

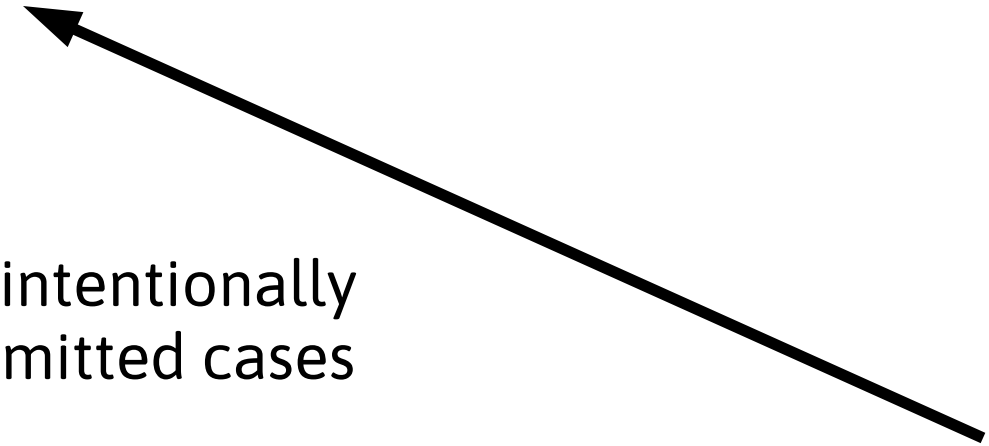
Adding Default Don't Care Case

Setting values to x is treated as Don't Care for synthesis

```
module mux3c (y, a, b, c, sel);
  output        y;
  input  [1:0] sel;
  input         a, b, c;
  reg           y;
  always @(a, b, c, sel)
  case (sel)
    2'b00:    y = a;
    2'b01:    y = b;
    2'b10:    y = c;
    default: y = 1'bx;
  endcase
endmodule
```

covers sel=="11"
allowing logic optimization in synthesis
For sim, it also covers any condition where sel has x or z
in it and says output should be set to 'x'

```
case ({lighting,rain,sun})
  {3'b110} : go =1'b0; run = 1'b0; umbrella=1'b0;
  {3'b010} : go =1'b1; run = 1'b0; umbrella=1'b1;
  {3'b001} : go =1'b1; run = 1'b1; umbrella=1'b0;
  default : go = 1'bx; run = 1'bx; umbrella = 1'bx;
endcase
```



default case item covers intentionally and accidentally omitted cases

safe?

What if I see the sun and there is lighting?
Use default cases with caution.

```
case ({lighting,rain,sun})
  {3'b110} : go =1'b0; run = 1'b0; umbrella=1'b0;
  {3'b010} : go =1'b1; run = 1'b0; umbrella=1'b1;
  {3'b001} : go =1'b1; run = 1'b1; umbrella=1'b0;
  default : go = 1'b0; run = 1'bx; umbrella = 1'bx;
endcase
```

- Maybe better to default to 0 when you miss a case?
- Sometimes a coding style include a safe catchall is recommended.
- Note, if you do this, you'll miss out on having go=x in simulation when the input is a neglected case or x or z.
- An x result allows you to clearly identify neglected cases.
- One option is to add a print statement to the case, (which is of course not synthesized)

```
case ({lighting,rain,sun})
  {3'b110} : go =1'b0; run = 1'b0; umbrella=1'b0;
  {3'b010} : go =1'b1; run = 1'b0; umbrella=1'b1;
  {3'b001} : go =1'b1; run = 1'b1; umbrella=1'b0;
  default : go = 1'b0; run = 1'bx; umbrella = 1'bx;
endcase
```

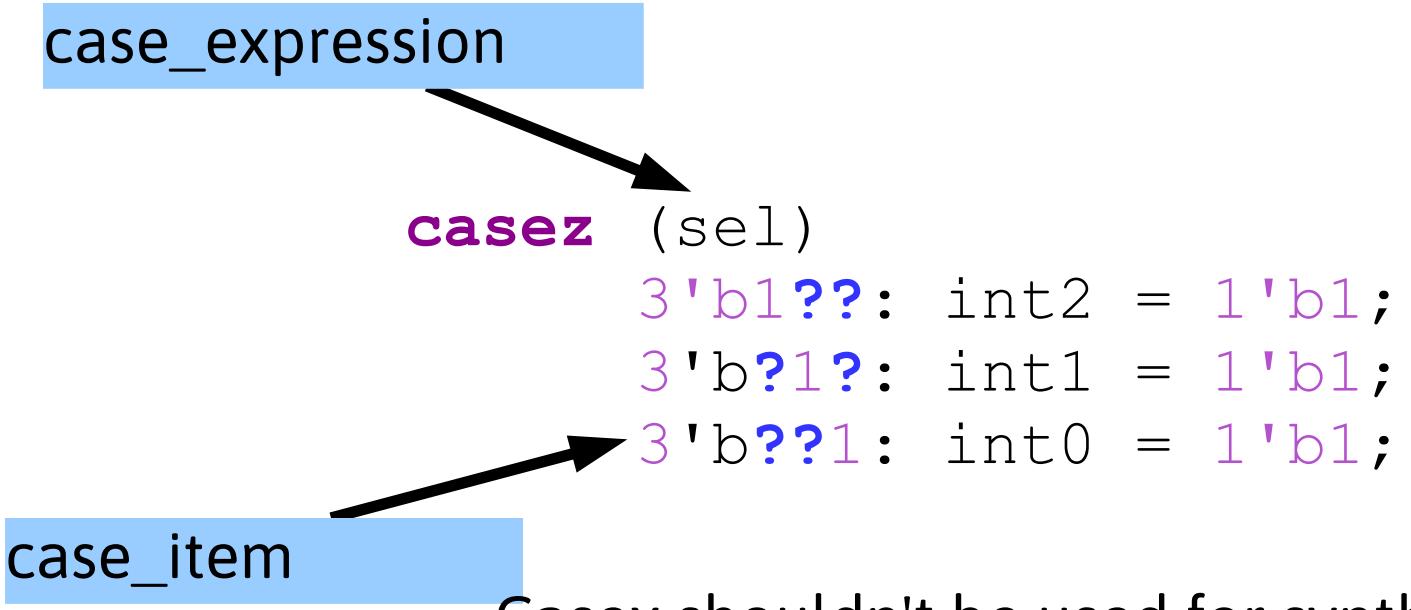
Alternate to using default, not preferred:

```
go = 1'b0; run = 1'bx; umbrella = 1'bx;
case ({lighting,rain,sun})
  {3'b110} : go =1'b0; run = 1'b0; umbrella=1'b0;
  {3'b010} : go =1'b1; run = 1'b0; umbrella=1'b1;
  {3'b001} : go =1'b1; run = 1'b1; umbrella=1'b0;
endcase
```

Implementing Don't care inputs

casex treats x and z as dc
casez treats z as dc
Note: in Verilog, **?** is an alias for **z** in numerical literals

So, when coding a case statement with "don't cares," use a casez statement and use "?" characters instead of "z" characters in the case items to indicate "don't care" bits.



Casex shouldn't be used for synth. only sim.

Full Case and Parallel Case

Full case: Does your logic cover all the cases for each and variable achieving complete assignment.
Avoids inferred latches.

Parallel Case: Are all the cases mutually exclusive?
Overlapping cases can mean complex logic is involved to determine correct action.

Control logic for Full, Parallel case statements can clearly be implemented as sum of products and easily optimized.

Concepts applies to if-else constructs as well.

Non-parallel case

```
module intctl1b (int2, int1, int0, irq);
  output       int2, int1, int0;
  input  [2:0] irq;
  reg          int2, int1, int0;
  always @(irq) begin
    {int2, int1, int0} = 3'b0;
    casez (irq)
      3'b1??: int2 = 1'b1;
      3'b?1?: int1 = 1'b1;
      3'b??1: int0 = 1'b1;
    endcase
  end
endmodule
```

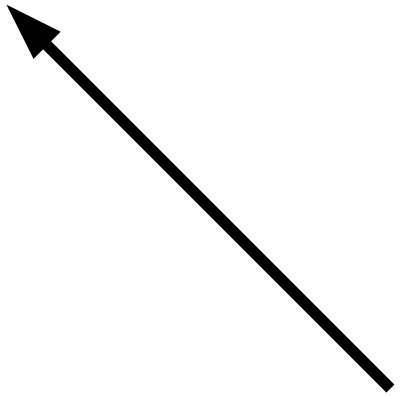
(Cummings 1999)

- This is a non-parallel case, order matters here
- Priority is given to the first case and so on
- First match blocks others

- Based on order, this implements a priority encoder
- Encoding logic in “the order” is not best style

Non-parallel case

```
casez ({lighting,rain,sun})
  3'b1?? : go =1'b0; run = 1'b0; bring_umbrella=1'b0;
  3'b?1? : go =1'b1; run = 1'b0; bring_umbrella=1'b1;
  3'b??1 : go =1'b1; run = 1'b1; bring_umbrella=1'b0;
  default : go = 1'b0; run = 1'bx; bring_umbrella = 1'bx;
endcase
```



Note the second case is only run if rain==1
And lighting != 1
This logic happens to be intended

Note: This default covers clear night “000” and any 'x' inputs in simulation
Allows synthesizer to optimize logic for run and bring_umbrella

Non-parallel case

```
casez ({lighting,rain,sun})
  3'b1?? : go =1'b0; run = 1'b0; bring_umbrella=1'b0;
  3'b?1? : go =1'b1; run = 1'b0; bring_umbrella=1'b1;
  3'b??1 : go =1'b1; run = 1'b1; bring_umbrella=1'b0;
  3'b000 : go =1'b0; run = 1'bx; bring_umbrella=1'bx;
  default : go = 1'bx; run = 1'bx; bring_umbrella = 1'bx;
endcase
```

Non-parallel case

```
module decoder (sel, a, b, c);
  input [2:0] sel;
  output a, b, c;
  reg a, b, c;
  always @(sel) begin
    {a, b, c} = 3'b0;
    casez (sel)
      3'b1??: a = 1'b1;
      3'b?1?: b = 1'b1;
      3'b??1: c = 1'b1;
    endcase
  end
endmodule
```

-
-
-

Parallel Case

```
module intctl2a (int2, int1, int0, irq);
  output       int2, int1, int0;
  input  [2:0] irq;
  reg          int2, int1, int0;
  always @(irq) begin
    {int2, int1, int0} = 3'b0;
    casez (irq)
      3'b1??: int2 = 1'b1;
      3'b01?: int1 = 1'b1;
      3'b001: int0 = 1'b1;
    endcase
  end
endmodule
```

(Cummings 1999)

parallel case
This code Implements a priority encoder regardless of case order

Statement preceding **casez** provides default case, so this block is also "full-case".

Parallel Case

```
casez ({lighting,rain,sun})
  3'b1?? : go =1'b0; run = 1'b0; bring_umbrella=1'b0;
  3'b01? : go =1'b1; run = 1'b0; bring_umbrella=1'b1;
  3'b001 : go =1'b1; run = 1'b1; bring_umbrella=1'b0;
  3'b000 : go =1'b0; run = 1'bx; bring_umbrella=1'bx;
  default: go =1'bx; run = 1'bx; bring_umbrella = 1'bx;
endcase
```

the zero in the leftmost position makes it clear that this case does not apply when there is lighting

It also creates a case_item with matches that are mutually exclusive to the first.

- When all **case_items** and any included **default** have mutually exclusive match sets, we have a parallel cases
- If any **input** to the **case_expression** can match more than one **case_item** or a **case_item** and **default**, the we do not have a parallel cases

Parallel and Non-Parallel Examples

As an academic exercise, consider effect of moving first case.

Non-Parallel Case, order does matter

```
casez ({lighting,rain,sun})
  3'b?1? : go =1'b1; run = 1'b0; bring_umbrella=1'b1;
  3'b??1 : go =1'b1; run = 1'b1; bring_umbrella=1'b0;
  3'b1?? : go =1'b0; run = 1'b0; bring_umbrella=1'b0;
  3'b000 : go =1'b0; run = 1'bx; bring_umbrella=1'bx;
  default: go =1'bx; run = 1'bx; bring_umbrella = 1'bx;
endcase
```

Note: Walking in park with umbrella during a lighting Storm can be unhealthy

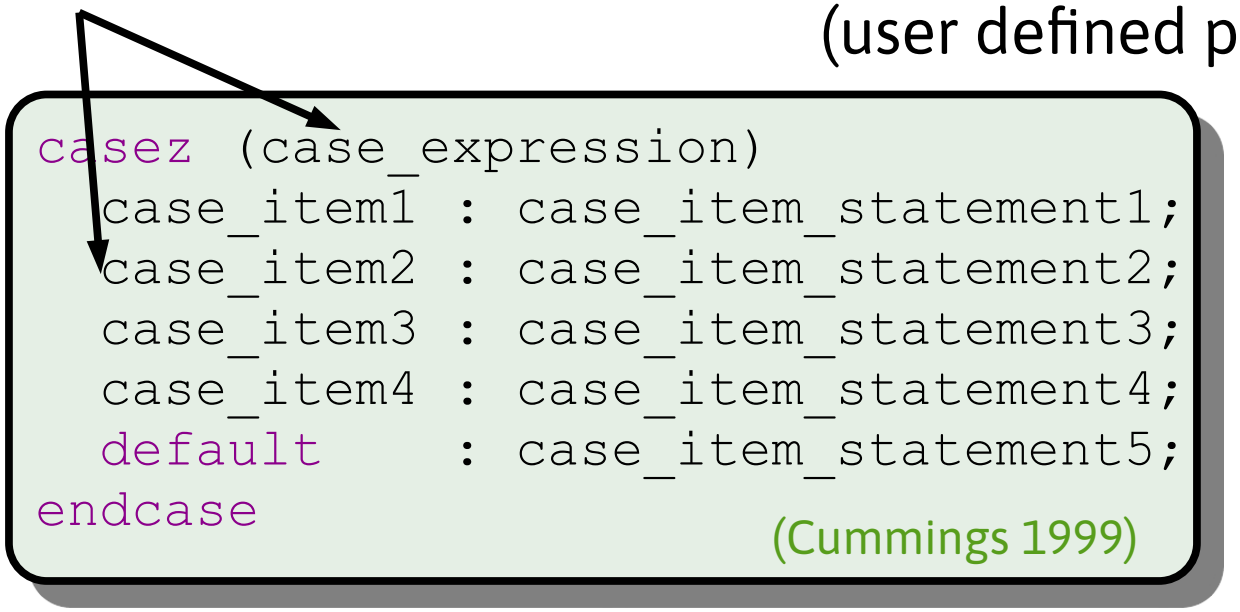
Parallel Case, order doesn't matter

```
casez ({lighting,rain,sun})
  3'b01? : go =1'b1; run = 1'b0; bring_umbrella=1'b1;
  3'b001 : go =1'b1; run = 1'b1; bring_umbrella=1'b0;
  3'b1?? : go =1'b0; run = 1'b0; bring_umbrella=1'b0;
  3'b000 : go =1'b0; run = 1'bx; bring_umbrella=1'bx;
  default: go =1'bx; run = 1'bx; bring_umbrella = 1'bx;
endcase
```

casex,casez

In literal numbers ? is an alternative for z
It is not a shorthand for 0,1,x,z as with UDPs

(user defined primitives)



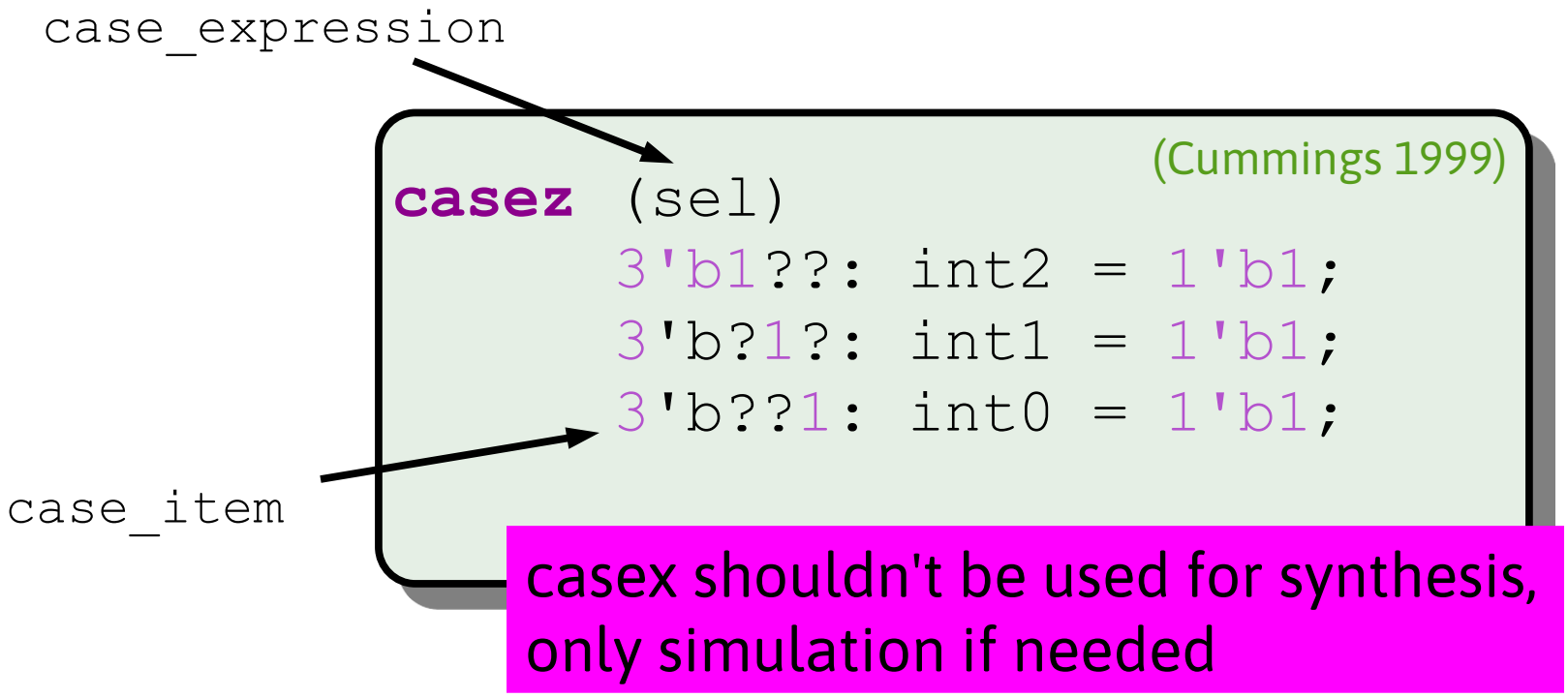
```
casez (case_expression)
  case_item1 : case_item_statement1;
  case_item2 : case_item_statement2;
  case_item3 : case_item_statement3;
  case_item4 : case_item_statement4;
  default    : case_item_statement5;
endcase
```

(Cummings 1999)

Implementing Don't care inputs

```
casex treats x and z as don't care
casez treats z      as don't care
```

When coding a case statement with "don't cares," use a casez statement and use "?" characters instead of "z" characters in the case items to indicate "don't care" bits.



Which Case?

sel = 2'bx1

```
case (sel)
  2'b00    : case_item_statement1;
  2'b10    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casez (sel)
  2'b00    : case_item_statement1;
  2'b10    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casex (sel)
  2'b00    : case_item_statement1;
  2'b10    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase
```

```
sel = 2'bz1

case (sel)
  2'b00    : case_item_statement1;
  2'b10    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casez (sel)
  2'b00    : case_item_statement1;
  2'b10    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casex (sel)
  2'b00    : case_item_statement1;
  2'b10    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase
```

sel = 2'b11

```
case (sel)
  2'b00    : case_item_statement1;
  2'b1x    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casez (sel)
  2'b00    : case_item_statement1;
  2'b1x    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casex (sel)
  2'b00    : case_item_statement1;
  2'b1x    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase
```

```
sel = 2'b1x

case (sel)
  2'b00    : case_item_statement1;
  2'b1x    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casez (sel)
  2'b00    : case_item_statement1;
  2'b1x    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase
```

Not useful for synth

```
casex (sel)
  2'b00    : case_item_statement1;
  2'b1x    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase
```

sel = 2'b1z

```
case (sel)
  2'b00    : case_item_statement1;
  2'b1x    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casez (sel)
  2'b00    : case_item_statement1;
  2'b1x    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casex (sel)
  2'b00    : case_item_statement1;
  2'b1x    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase
```

sel = 2'b1x

```
case (sel)
  2'b00    : case_item_statement1;
  2'b1z    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casez (sel)
  2'b00    : case_item_statement1;
  2'b1z    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase

casex (sel)
  2'b00    : case_item_statement1;
  2'b1z    : case_item_statement2;
  2'b01    : case_item_statement3;
  default  : case_item_statement4;
endcase
```

Adding Default Don't Care Case

```
module mux3c (y, a, b, c, sel);
  output         y;
  input  [1:0] sel;
  input          a, b, c;
  reg            y;
  always @(a , b , c , sel)
    y=1'bx;
    case (sel)
      2'b00:    y = a;
      2'b01:    y = b;
      2'b10:    y = c;
      default: y = 1'bx;
    endcase
endmodule
```

(Cummings 1999)

- Two options
- The second one is preferred for readability

Allowed Use of x,z,? for Synthesis

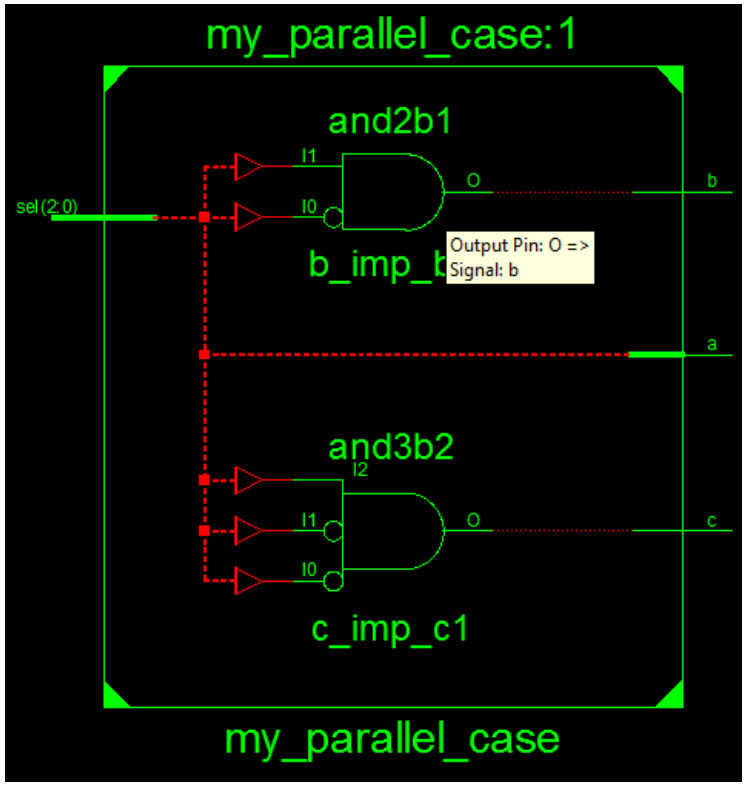
```
casex/casez (case_expression)
   case_item1 : case_item_statement1;
   case_item2 : case_item_statement2;
   case_item3 : case_item_statement3;
   case_item4 : case_item_statement4;
   default    : case_item_statement5;
endcase
```
(Cummings 1999)

- casex allows x in the literal **case_expression** and in **case-item** literal expression
- Synthesis only considers x and z/? in **case-item** expression

- casez allows z in the **case _expression** and in the literal **case-item** expression
- synthesis only considers z/? in **case-item** expression

While this may be dependent on the synthesis tool, a reasonable practice is to not use x or z/? in the case_expression for code intended for synthesis

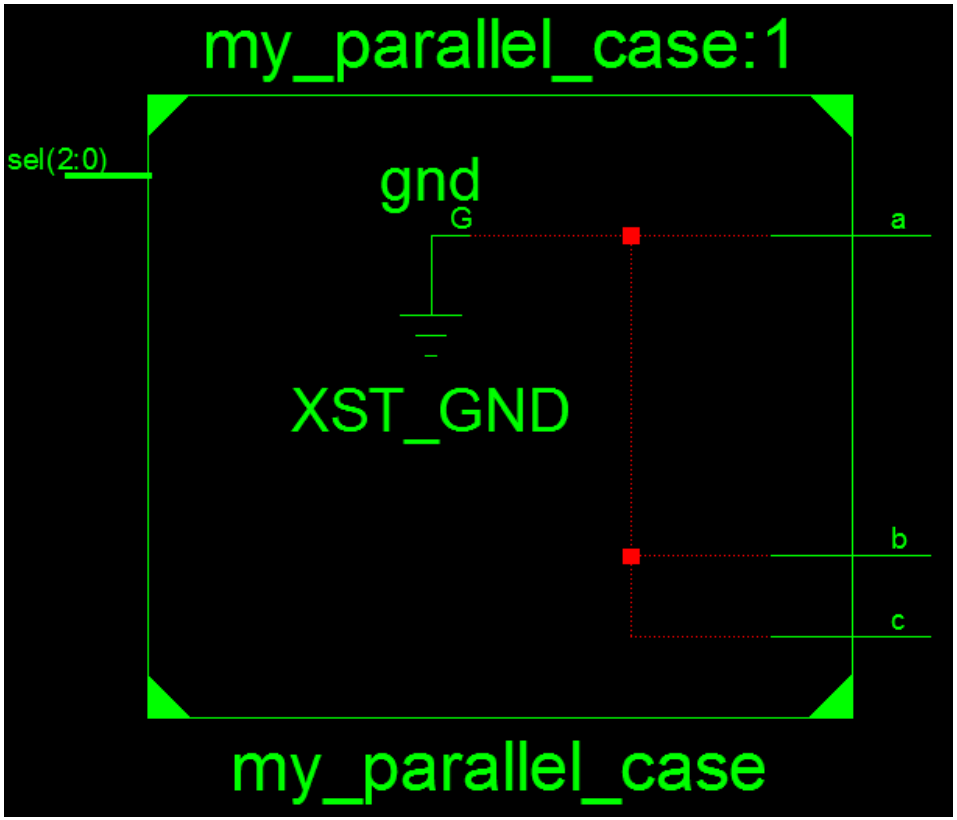
```
module my_parallel_case (sel, a, b, c);
   input [2:0] sel;
   output a, b, c;
   reg a, b, c;
   always @(sel) begin
      {a, b, c} = 3'b0;
      casez (sel)
         3'b1??: a = 1'b1;
         3'b?1?: b = 1'b1;
         3'b??1: c = 1'b1;
      endcase
   end
endmodule
```



Same as

```
3'b1??: a = 1'b1;
3'b01?: b = 1'b1;
3'b001: c = 1'b1;
```

```
module my_parallel_case (sel, a, b, c);
   input [2:0] sel;
   output a, b, c;
   reg a, b, c;
   always @(sel) begin
      {a, b, c} = 3'b0;
      case (sel)
        3'b1??: a = 1'b1;
        3'b?1?: b = 1'b1;
        3'b??1: c = 1'b1;
      endcase
   end
endmodule
```



Remember,? Is same as z in literal expressions

```
module mux3c (output reg y,input [1:0] sel,
input a, b, c);
  always @*
    case (sel)
      2'b00: y = a;
      2'b01: y = b;
      2'b10: y = c;
      default: y = 1'bx;
    endcase
endmodule
```

- **In simulation see x output**
- **In synthesis, x is treated as don't care and nothing extra is created**

(Cummings 1999)

```
module mux3a (output reg y, input  [1:0] sel, input a,
b, c);
always @*
  case (sel)
    2'b00: y = a;
    2'b01: y = b;
    2'b10: y = c;
  endcase
endmodule
```

Not full case
'b11 produces latch behavior in synthesis and simulation

(Cummings 1999)

Use of Compiler Directives

- Some synthesis tools have special directives that can be provided in comments
  One such directive is to provide full-case not codded explicitly in the standard HDL

- Full Case Directive Example:

```
module mux3b (y, a, b, c, sel);
  output reg y;
  input [1:0] sel;
  input a, b, c);
  always @*
    case (sel) // synthesis full_case
      2'b00: y = a;
      2'b01: y = b;
      2'b10: y = c;
    endcase
endmodule
```

(Cummings 1999)

- Full case directive is redundant if default case is given
- Without the directive, all unmatched cases become latches for SYNTHESIS, and cause holding of values in SIM.
- With the directive, the output defaults to don't care when not covered by the coded cases for the purpose of synthesis avoiding latches.
- This causes a discrepancy between simulation and synthesis unless post-synthesis code is used for simulation
- Using full_case directive out of habit when missing a case accidentally in design is poor practice

- Parallel Case Directive Example:

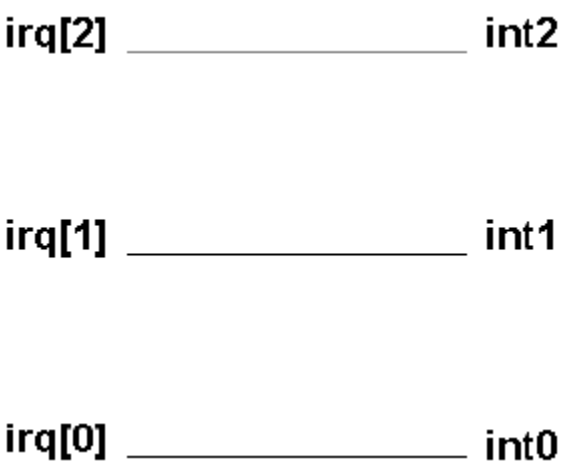
```
module intctl1b
(output reg int2, int1, int0,
input [2:0] irq );
  always @* begin
    {int2, int1, int0} = 3'b0;
    casez (irq) // synthesis parallel_case
      3'b1??: int2 = 1'b1;
      3'b?1?: int1 = 1'b1;
      3'b??1: int0 = 1'b1;
    endcase
  end
endmodule
```

(Cummings 1999)

This is basically a priority encoder
i.e. case_items 1??,01?,001 for sim

But in synthesis case 2 is handled
irrespective of case1 and generates the following

Synthesis result



Coding with a constant select

It is possible to use a constant select and use variable case items:

```
reg [2:0] encode ;
 case (1)
   encode[2] : $display(“Select Line 2”) ;
   encode[1] : $display(“Select Line 1”) ;
   encode[0] : $display(“Select Line 0”) ;
  default $display(“Error: One of the bits expected ON”);
endcase
```