Multiple Assignments

Two Outputs Connected Together

```
always @(posedge Clk)
  begin
  if (En1) Q<=D1;
  end
xz
always @(posedge Clk)
  begin
  if (En2) Q<=D2;
  end
```

Multiple Assignment

- *Mutually exclusive* means that Q<=D1 and Q<=D2 could never happen together.
- If they were both in one if-else statement the compiler would know they could never happen together.
- Here both EN1 and EN2 might be true together.

Possible Simulation Results

- The simulator will choose one to do first. No one knows which. The lasting result will be the final one.

Possible Synthesis Results

- The compiler chooses one result.
- The compiler generates two flip-flops and ANDs the result.
- All outputs my be disconnected.

Multiple Assignments

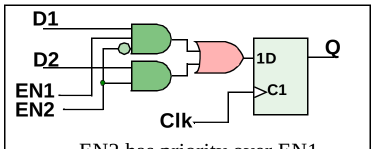
If both statements are in the same procedure, the En2 would replace the En1 result in zero time. In synthesis this would mean the En2 result would take priority over En1.

```
always @(posedge Clk)
    begin
        if (En1) Q=D1;
        if (En2) Q=D2;
    end
```

Blocking was used for flip flops, to ensures Q=D2 is done after Q=D1 and hence replaces Q=D1.

If delays are put on the statements simulation could give a glitch. Synthesis would not. It would generate a circuit which would give EN2 priority.

```
always @(posedge Clk)
    begin
        if (En1) #2 Q<=D1;
        if (En2) #3 Q<=D2;
    end
```



EN2 has priority over EN1

20.• Problem

What happens here?

```
always @(posedge Clk)
    begin
        if (En1) Q=D1;
    end
always @(posedge Clk)
    begin
        if (~En1) Q=D1;
    end
```

Using Case Statements

Demux Inference from case

```
wire [2,0] in;
reg [7:0] Y;
always @(in) begin
   case(in)
     3'd0: Y=8'b00000001;
     3'd1: Y=8'b00000010;
     3'd2: Y=8'b00000100;
     3'd3: Y=8'b00001000;
     3'd4: Y=8'b00010000;
     3'd5: Y=8'b00100000;
     3'd6: Y=8'b01000000;
     3'd7: Y=8'b10000000;
   endcase
```

Full Case

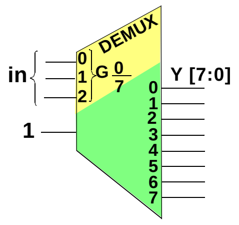
- All cases are covered.
  It avoids latches

***Parallel Case* (Mutually Exclusive)**

- No two cases can be active at once.
  It can be implemented as a mux.

No undefined or parallel cases

- The compiler can statically determine that all possible cases are covered.
- The compiler can statically determine that no two cases are active at once.

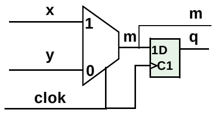


Multiple Assignment Race (cont from previous page)

21.• PROBLEM

What common simulation problem might be caused by this code?

```
wire clok, x, y;
reg m,q;
always @(posedge clok)
    begin: storage
        q <= m;
    end

always @(clok or x or y)
    begin: mux
        if (clok) m = x;
        else m = y;
    end;
```



This is a subtle simulation race. Both the mux and the flip flop respond to clok. The flip flop, as per the rule uses nonblocking assigns. The mux, as per the rule uses blocking assigns.

However on posedge clok the simulator might choose the mux first. Then the mux code would block the flip flop until the mux had switched.

The actual circuit would capture the mux value before the clock switched it.

Using Case

There are several problems that can happen with case statement synthesis.

1. If the case is known to cover all the possibilities the input condition can assume it is said to be a *full case*. Unfortunately the synthesizer will not know this unless case covers all 2^N possibilities for an N bit condition. If the synthesizer does not know it is a full case, it will insert latches.
2. If two different conditions may happen at once, they will activate two different outputs at the same time. This is called a *nonparallel case.*

Latch Inference in *Case*

Not Obvious Full-Case,

Case with a restricted input

```
reg [6:1] Y;

always @(a or b or c)
 begin

// if a,b,c = 1,1,1 make c1=0
   c1 = (a&b&c) ? 0 : c

   case({a,b,c1})
      3'd0: Y=000000;
      3'd1: Y=000001;
      3'd2: Y=000010;
      3'd3: Y=000100;
      3'd4: Y=001000;
      3'd5: Y=010000;
      3'd6: Y=100000;
   endcase
 end
```

Apparent undefined cases

3'd7: Y=000000;

a,b,c = 1,1,1 cannot occur.
Synthesis does not know that.
Thus synthesis will infer 7 latches.

To avoid latches

Put in default

```
  . . . .
  3'd5: Y=010000;
  3'd6: Y=100000;
  default: Y=000000;
```

Better default

A better default is:

default: Y=xxxxxx;

It gives the synthesizer more choice.

Not Obvious Full-Case

If a *case* contains all 2^n cases no latches will be generated.

If it contains less than 2^n cases, latches will be generated unless it is very obvious all cases are covered. Synthesizers do not look back very far to determine if all cases are covered.

Use Default

The default statement does no harm if it is used and not needed.

Always put in a default, whether you need it or not, unless you want the latches.

Place xxxx as a the Default Output

If you know the default will never be selected by the case, then you can put in anything you want.
The logic that is easiest to minimize is x (don't care). Requiring the default to take some particular value, like zero, can greatly increase the size of the circuit.

Nonparallel Case

NOT Mutually Exclusive

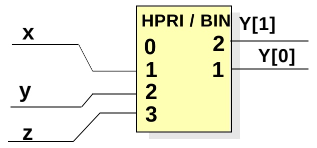
```
always @(x or y or z) begin
  case(1'b1)
    x: Y=2'b01;
    y: Y=2'b10;
    z: Y=2'b11;
    default: Y=2'bxx;
  endcase
```

FORCE Mutually Exclusive (Parallel)

```
always @(x or y or z) begin
  case({x,y,z})
    3'b100:  Y=2'b01;
    3'b010:  Y=2'b10;
    3'b100:  Y=2'b11;
    default: Y=2'bxx;
  endcase
```

Not Mutually Exclusive (nonparallel) Case

- Two cases can be active at once. Priority encoder generated.



Synopsys Compiler Directive

Simulater treats as comments
Tells Synopsys x, y and z can never happen at the same time.

Designer must enforce this!

Force Parallel Case

// synopsys parallel_case
avoids creating a priority encoder

```
case(1'b1) // synopsys parallel_case
  x:   Y=2'b01;
  y:   Y=2'b10;
  z:   Y=2'b11;
  default: Y=2'bxx;
endcase
```

Not Recommended

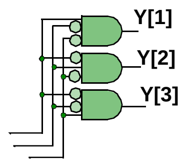
Parallel-Case

Whenever two or more lines of the case statement may be selected at once, the simulation executes the first line encountered in the listing. This is like a priority encoder.

In a *parallel* case, the synthesizer assumes some other circuit keeps two lines from being selected at once.

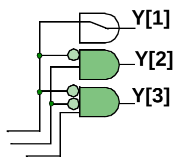
//synopsis directives can be used to tell the synthesizer to force a parallel-case but one can also write the code to explicitly say what is desired. This latter method is synthesizer independent and keeps the simultation and synthesis in agreement. Such code may require the *casex* (or *casez*) command described on the next slide.

Coding for full decoding, priority encoder, or parallel case



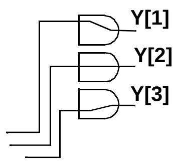
```
always @(x or y or z)
 begin
 case({x,y,z})
  3'b100 : Y=3'b100;
  3'b010 : Y=3'b010;
  3'b001 : Y=3'b001;
  default: Y=3'b000;
 endcase
```

Full decoding:
Assumes more than one of x, y, z can be 1 and removes those cases.



```
always @(x or y or z)
 begin
 casex({x,y,z})
  3'b1xx : Y=3'b001;
  3'b01x : Y=3'b010;
  3'b001 : Y=3'b001;
  default: Y=3'b000;
 endcase
```

Priority decoder
Assumes more than one of x, y, z can be 1 but takes the first one as correct



```
always @(x or y or z)
 begin
 casex({x,y,z})
  3'b1xx : Y=2'b100;
  3'bx1x : Y=2'b010;
  3'bxx1 : Y=2'b001;
  default: Y=2'bxxx;
 endcase
```

Parallel case
Assumes only one of x, y, z can be 1 at a time. Depends on some other circuit to enforce this.

Using casez

Allows Don’t Cares In Case-items

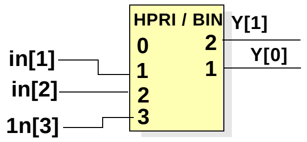
```
    wire [3,1] in;
    reg [1:0] Y;

    always (in) begin
      casez(in)  //or casex (in)
      3’bxx1: Y=2’b01;
      3’bx10: Y=2’b10;
      3’b100: Y=2’b11;
      default Y=2’b00;
      endcase
```

Generates a priority encoder

- xx1 has the highest priority.

Default

- Covers only 3’b000.



Don’t Cares In Right-Hand Side

```
  always @(x or y or z) begin
    case({x,y,z})
      3’b001: Y=2’b01;
      3’b010: Y=2’b10;
      3’b100: Y=2’b11;
      default Y=2’bxx;
    endcase
```

Don’t Cares Can Simplify Logic

- Don’t force the defaults to zero if you don’t care.
  It makes the logic larger.
- Casez not necessary for output don’t cares.
- casez slightly prefered over casex.

Casex/casez

For simulation

Case treats a bit in a variable as having four possible values {0, 1, x, z}, thus x only matches x, not 1 or 0.
Casex treats x, z or ? as a don’t care which can match 0, 1, x or z.
Casez treats z or ? as a don’t care which can match 0, 1, x or z, but x cannot match 0 or 1.

case

case\data	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z,?	0	0	0	1

casex

case\data	0	1	x	z
0	1	0	1	1
1	0	1	1	1
x	1	1	1	1
z,?	1	1	1	1

casez

case\data	0	1	x	z
0	1	0	0	1
1	0	1	0	1
x	0	0	1	1
z,?	1	1	1	1

For example: Given- aa = 3b'1x0;

case (aa)
3’b110: ...// No match because 1 does not match x with a *case* statement.
3’b1x0: ... // Matches

casex (aa)
3’b110: ... // Matches aa because 1 does match x with a *casex* statement.
3’bx10: ...// Matches aa.

casez (aa)
3’bxx0: ... // x does not match 1 with a *casez* statement, although x matches x.
3’bzz0: ...// Matches aa.

For synthesis

No x values ever propagate in synthesis. However x values in the simulation cause an unexpected match with *casex*. Using *casez* will avoid those problems.

Don’t cares in the outputs are fine for *case*, *casez* or *casex*.

Negative Numbers

Confusion Between Reg and Integer

Integers are 2's Complement

Integers declarations default to a 32-bit 2's complement number.
The compiler will eventually decide how many bits are needed.

Reg numbers are nonzero integers

The length of registered numbers is given in the declaration.

```
integer B,C;
reg [4:0] X;
    always @(B,C,X)
     begin
        X = -5'd7;
        B = 10;
        C = B + X;
     end
```

Value of X

X will hold a 5-bit -7
-00111 ⇒ 11000 +1⇒ 11001 {-7 in 2's complement}

Reg numbers are never negative,
Hence 11001 is taken as 25.

B will hold 000. . . 00001010 (32 or less bits)

B + X = 00001010+11001 = 000100011
25 10 35

Did you want 35 or 3?

Negative Numbers

Two's Complement

To change a binary number to its two's complement

Change the exchange the ones and zeros, then add 1, ignore any off-end carries from the add.

-10 ⇒ –001010 ⇒ 110101 +1 ⇒ 110110 {-10 in 2's complement}

Sign Extension

All two's compliment numbers of different lengths must be sign extended when added. Thus:

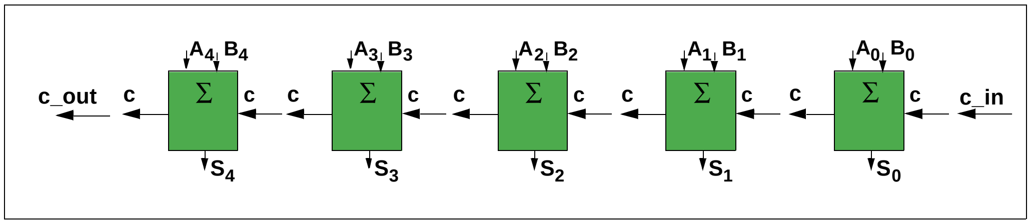
```
reg [4:0] x ; reg [5:0] y, z;
    z = {x[4], x} +y ; // sign extend x to 5 bits.
```

If the bits represent unsigned numbers, then do not sign extend.

Using For Loops For Building Iterative Hardware

Build an 5-bit ripple-carry adder.

```
  reg [4:0] A, B, S;
  reg c, c_out;   wire c_in;            // S is the sum, c  the carry
  always @(c_in or A or B)
    begin
    c = c_in;
    for(i=0; i <=4; i=i+1)
       begin
       {c, S[i]} = A[i] + B[i] + c;     // Concatenate the outputs into a 2-bit vector.
       end
    c_out = c;
    end
```



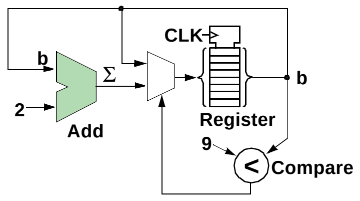
Hardware Loops[1]

Loops give multiple copies of a basic instance.

The code in the loop will be synthesized, a different instance for each iteration.
Output leads from one block, with the same name as an input lead, will connect between iterations.
See the variable “c” in the program.

While loops are partially supported for synthesis. They represent a conditional branch. All while loops must be broken by an @(posedge clock) statement. Thus:-

```
always @(posedge clock)
    begin
        while ( b >9 )
        begin
            @ (posedge clock); // break the zero delay loop
            b <= b+2;
        end
    end
```

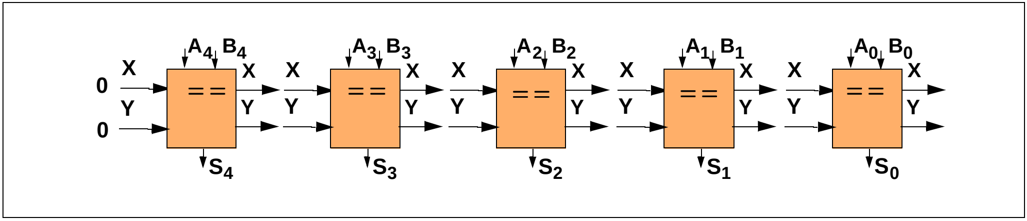


1. Palnitakar, *Verilog*, Prentice Hall, 1998, p..285

An Iterative Comparator Hardware

Build an 5-bit comparator from blocks.

```
reg [4:0] A, B;
reg x, y;      // S is the sum, c  the carry
always @(A or B)
  begin
  x=0; y=0;   // Above the highest order bit, the two are equal
  for(i=0; i <=4; i=i+1)
    begin
    x=(A[i] > B[i]) | x;   // A is larger at this bit or at a higher order bit.
    y=(B[i] > A[i]) | y;   // B is larger at this bit or at a higher order bit.
    end
  end
```



At the output:
x,y = 0,0 means A=B,   x,y = 1, 0 means A > B,   x,y = 0,1 means A < B,

Loops to Generate Iterative Circuits

This is an iterative comparator used as a lab in the Switching Circuits course at Carleton.

It only compares non-negative integers, where the number with the leftmost “1” is the largest.

22.• PROBLEMS

a. Write a **for** loop to calculate the parity of a 6-bit number. It should include-
   if (data[i]) OddPar= ~OddPar;

b. One way to change a binary number to its two’s complement is:
   Start at the right hand side.
   Leave all bits unchanged until after the first “1” is found.
   Invert all bits to the left of the initial “1”.

   Thus: 1001_1000 has complement 0110_1000

   Write a loop to generate such a circuit.

Complier Directives

Tell The Synthesizer What To Do

- Written like comments
  // synopsys . . .
- The simulator will ignore them
- Directs synthesis.
- Simplifies some language problems.
  However it is nearly always possible to avoid them by proper coding.
- Thus simulation will agree with synthesis only if it was coded properly.
- Limits you to one compiler.
- Makes formal verification difficult.
- There are many of these compiler directives.
  Check the Synopsys Manual

Example

Force Asynchronous Reset

```
module latch(Q,D,C,R);
  input D,C,R;
  output Q; reg Q;

// synopsys asynch_set_reset
 "R"
  always @(C or R)
  begin:
    if (R)
     Q = 0;
    else if (C)
     Q = D;
  end
```

Compiler Directives

Other Compiler Directives

// synopsys async_set_reset	
// synopsys sync_set_reset	
//synopsys async_set_reset_local	applies directive to specified signals in a named block
//synopsys one_hot	indicates only one of a list of signals is true at a time. Useful to show set and rest are never both applied at once.

One of the more useful compiler directives is used to force a particular library module for arithmetic operations (next slide).

Formal verification

This is where the logic of a program is compared weith the logic of another program. This is often done after inserting special structures only used for testing, or after had optimizations on a compiled circuit.

The verification programs have trouble with compilier assertions.

Forcing Specific Synopsys Designware

Synopsys uses designware to implement counters, adders, comparators, etc.

Control the type of function used by inserting compiler directives into your code.

Example:

Library DWO1 has two increments, ripple carry "rpl" and carry look-ahead "cla."

Force the named block `bill` to use a carry look-ahead incrementer.

```
always @(count)
  begin : bill   //named procedure
    /* synopsys resource billspecial:
       map_to_module = "DW01_inc",
       implementation = "cla",
       ops = "greasedIncr";
    */
      count = count + 1;    //synopsys label greasedIncr
  end
```

- **Must insert only in a nonclocked, named procedure or function.** **i.e not after @(posedge. . .,**
- `billspecial` **will be the name given this instantiation.**
- "`DW01_inc`" **and** "`cla`" **are from the Synopsys library** DW01
- **The label applies to the most recently parsed function.** `count = count + 1` // **synopsys label greasedIncr**

Mapping to a Specific Library Module

Named Procedures

pecifically map an operation it must be inside a named procedure. named by writing the name after begin.

```
always @(a or ...
     begin: bill
          ...
```

Meanings of the mapping labels

// *synopsys label greasedIncr* labels the + operation with name *greasedIncr*.
This label is bound to the instantiation named *billspecial* by the *ops* =*"greasedInc"*; statement.

The resource is module *DW01_inc*, in the designware library *DW01*

The specific implementation in the library is *cla*.

Libraries are fairly automatic

The simulator will automatically choose an implementation for your criteria.

Experience with Adders

The DW01 library has (1999) had five adders. For a 4 to 7 bit adds in a Viterbi decoder, a Carleton graduate student, Youxing Zhao found:

The conditional sum adder (csa) was the fastest[1].
The ripple carry adder (rpl) was second and significantly slower.
The fast carry look-ahead (clf) was third.
The Brent-Kung (bk) and the carry look-ahead adder (cla) were last and about the same.

1. A. Bellaouar and M Elmasary, *Low-powered Digital VLSI Design Circuits and Systems*, Kluwer 1995, p.424 has a good summary of the csa. Each full adder calculates (S1,C1) and (S0, C0) which are the sum and carry for a carry-in of 1 and 0 respectively. Then muxs are used to select the appropriate answer.

Summary

Guidelies:

- Partition FSMs into next-state calc, outputs and registers.
  Use <= in the register procedure; use = in the others.
- In procedures:
  Feed all right-hand side variables through the trigger list (unless also on the left side.)
  Make all branches evaluate all left-hand side variables.
- If you are using negative numbers, add/sub only registers of equal length, and do sign extensions.
- Do not have the same left-hand side variable stored in two different procedures.
- For case statements:
  Always use a default at the end.
  Use casez if there are don't cares in the control.
  Use x for don't care outputs to minimize logic.
- Flip-flops procedures must start @(...edge clk) or @(...edge clk or ...edge reset)