laalcham@ucsc.edu for CSE 125, UCSC

Multiplexers and FSMs with don't cares

How to avoid latches:
Explicitly describe each case in case statements and use the else block in if/else.

Always add a default case to case statements
if some cases are not described, a latch could use the not described input as an enable

- useful when there is a series of valid cases that all should map to some case
- useful when some case for the inputs is missing so it defaults.

use casez and don't care (?) to for situations where case for inputs has some flexibility

- don't cares must go inside a casez statement

casex also exists but it is not synthesizable. casez is for don't cares.

[Example: casez statement]

```
wire [15:0] in0, in1, in2, in3;
wire [1:0] ctrl0, ctrl1;
always @(*) begin
       casez ( {ctrl0, ctrl1} )
         4'b1?00:   Y = in0;
         4'b0?00:   Y = in1;
         4'b10?1:   Y = in2;
         4'b11?0:   Y = in3;
         default: Y = 16'hxxxx;
       endcase
end
```

Full Case Statement in Verilog

A **full case statement** is a case statement in which all possible case-expression binary patterns can be matched to a case item or to a case default. If a case statement does not include a case default and if it is possible to find an expression that does not match any of the defined case items, the case statement is not full.

- we want **full case statements** so code is **well defined for all inputs**
- not full case statements will generate a latch. If the latch case happens, the FFs will maintain the previous state and wait for defined input.

Case statement report

A case statement report looks something like this:

```
================================================
|           Line           |  full/ parallel   |
================================================
|            9            |    auto/auto      |
================================================
```

in the full/parallel column the report notes the design is auto/auto.
auto == full
no == not full
user == synopsis full_case

[Example: Full Case statement]

```
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
```

auto = full
no = not full
user = //synopsis full_case

BEST SOLUTION! (cover all the cases)

```
Statistics for case statements in always block at line 7 in file
        '.../mux3c.v'
================================================
|            Line            |  full/ parallel  |
================================================
|             9              |     auto/auto     |
================================================
```

[Example: Not Full Case Statement]

infers a latch. synthesis will show no, not full

```
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;
endcase

endmodule
```

Latch inferred

```
Statistics for case statements in always block at line 7 in file
        '.../mux3a.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            9            |     no/auto      |
================================================

Inferred memory devices in process
     in routine mux3a line 7 in file
        '.../mux3a.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       y_reg         |   Latch   |   1   |  -  |  -  | N  | N  |  -  |  -  |  -  |
===============================================================================
```

synopsis pragmas in Verilog force the synthesis tools to not infer a latch. synopsis is the company, and their pragmas generally supported by most tools. No advantage over using the default case, but will be seen somewhere. Litz prefers default case instead.

[Example: Forced to Full Case Statement]

The pragma below tells the program that 11 will never be input, so the case is full

```
module mux3c(output reg    y,
input      [1:0] sel,
input             a, b, c);

always @(*)
case (sel) // synopsys full_case
2'b00:   y = a;
2'b01:   y = b;
2'b10:   y = c;
endcase
endmodule
```

NO latch inferred

```
Warning: You are using the full_case directive with a case statement in which not all cases
are covered.

Statistics for case statements in always block at line 7 in file
        '.../mux3b.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            9            |     user/auto     |
================================================
```

Problem? Pre/Post synthesis simulation mismatch

Parallel Case Statement in Verilog

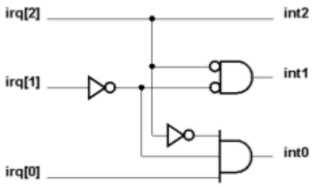
A **parallel case statement** is a case statement in which is it only possible to match a case expression to one and only one case item. If it is possible to find a case expression that matches more than one case item, the matching case items are said to be **overlapping case items** and the case statement is **not parallel.**

[Example: not full, non-parallel case statement]

Input 111 can go multiple places -> not parallel

```
module intctl1a(output reg  int2, int1, int0,
input      [2:0] irq);

always @(*) 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
```




```
Statistics for case statements in always block at line 6 in file
        '.../intctl1a.v'
================================================
|            Line            |  full/ parallel  |
================================================
|             9              |      no/no       |
================================================
```

Problem? Will infer priority encoder

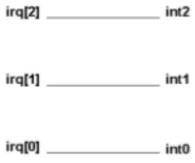
- input like 111 matches all 3 cases: synthesis tool adds a priority encoder, which means additional hardware and less efficiency
- non-full (because no default) and non-parallel (bc matching cases)

[Example: Synopsis pragma to force non-parallel to parallel]

Input 111 only goes to Multiple case stmts

```
module intctl1a(output reg  int2, int1, int0,
input      [2:0] irq);

always @(*) begin
{int2, int1, int0} = 3'b0;
casez (irq) // synopsys parallel case
3'b1??: int2 = 1'b1;
3'b?1?: int1 = 1'b1;
3'b??1: int0 = 1'b1;
endcase
end
endmodule
```




```
Statistics for case statements in always block at line 6 in file
        '.../intctl1b.v'
================================================
|            Line            |  full/ parallel  |
================================================
|             9              |     no/user      |
================================================
```

Problem? - Pre/Post synthesis simulation mismatch

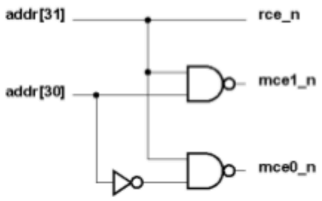
- instead, make clear what you want

[Example: Full and Parallel Case Statement]

Notice that all outputs are defined before the case statement.

```
module addrDecode1d (mce0_n, mce1_n, rce_n, addr);
  (output reg         mce0_n, mce1_n, rce_n,
   input      [31:30] addr                  );

  always @* begin
    {mce1_n, mce0_n, rce_n} = 3'b111;
    casez (addr)
      2'b10: {mce1_n, mce0_n} = 2'b10;
      2'b11: {mce1_n, mce0_n} = 2'b01;
      2'b0?:           rce_n  = 1'b0;
    endcase
  end
endmodule
```



No latches due to default assignment.

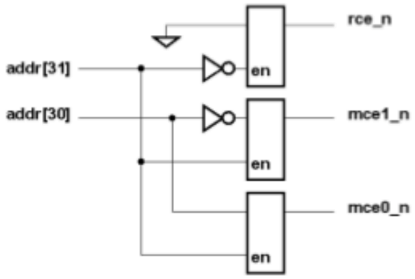
```
Statistics for case statements in always block at line 6 in file
        ...'addrDecode1d.v'
================================================
|            Line            |  full/ parallel  |
================================================
|             9              |    auto/auto     |
================================================
```

[Example: Not Full and Parallel]

On the RHS of the case statement we define how certain outputs are defined. Some outputs are missing for some cases, so outputs might not always be defined, making this a not full case. Every output driven by the comb logic circuit must be defined for all cases.

```
module addrDecode1a
  (output reg         mce0_n, mce1_n, rce_n,
   input      [31:30] addr                  );

  always @*
    casez (addr) // synopsys full_case
      2'b10: {mce1_n, mce0_n} = 2'b10;
      2'b11: {mce1_n, mce0_n} = 2'b01;
      2'b0?:           rce_n  = 1'b0;
    endcase
endmodule
```




```
================================================
|            Line            |  full/ parallel  |
================================================
|             8              |    user/auto     |
================================================

Inferred memory devices in process
     in routine addrDecode1a line 6 in file
          ...'addrDecode1a.v'.
=================================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=================================================================================
|     mce0_n_reg      |   Latch   |   1   |  -  |  -  | N  | N  |  -  |  -  |  -  |
|     mce1_n_reg      |   Latch   |   1   |  -  |  -  | N  | N  |  -  |  -  |  -  |
|      rce_n_reg      |   Latch   |   1   |  -  |  -  | N  | N  |  -  |  -  |  -  |
=================================================================================
```

Still infers latches due to assignment of multiple outputs!

[Example: Not full and Parallel Case Statement]

- the underscore (_) syntax is ignored , it is for readability

This case statement is not full because the MSB is 1 for all listed cases

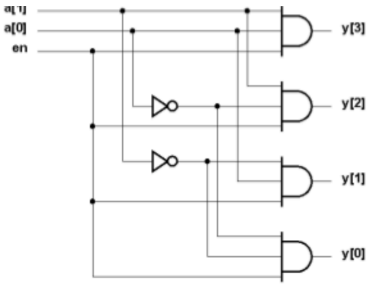
- a latch is not generated because y is assigned above the case statement

The case statement is parallel because no two unique inputs will match multiple cases.

Only 1 bit (depending on a, bit 0, 1, 2, or 3) will be assigned to 1. What should have been done here is assign entire 4-bit y to either 0001, 0010, 0100, or 1000, depending on the value of a. and use a default case for if the enable is low.

```
module code4a
  (output reg [3:0] y,
   input      [1:0] a,
   input            en);

  always @* begin
    y = 4'h0;
    case ({en,a})
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule
```



Matches simulation.

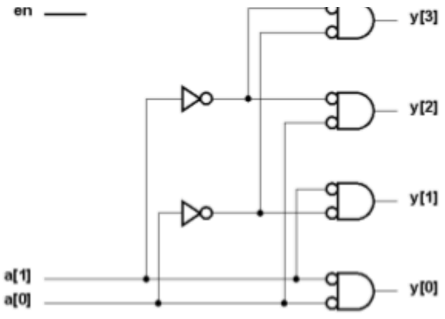
```
Statistics for case statements in always block at line 9 in file
        '.../code4a.v'
================================================
|           Line           |  full/ parallel   |
================================================
|            12            |     no/auto       |
================================================
```

[Example: Full and parallel Case Statement]

Synopsis case that is forced to full.

```
module code4b
  (output reg [3:0] y,
   input      [1:0] a,
   input            en);

  always @* begin
    y = 4'h0;
    case ({en,a}) // synopsys full_case
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule
```



Full case optimized away enable, doesn't match simulation!

```
Statistics for case statements in always block at line 10 in file
        '.../code4b.v'
================================================
|           Line           |  full/ parallel   |
================================================
|            13            |     user/auto     |
================================================
```

Design for Performance

Hardware design is very different from software design. The main technique of HD is **pipelining**

CSE 120: pipeline high level tasks

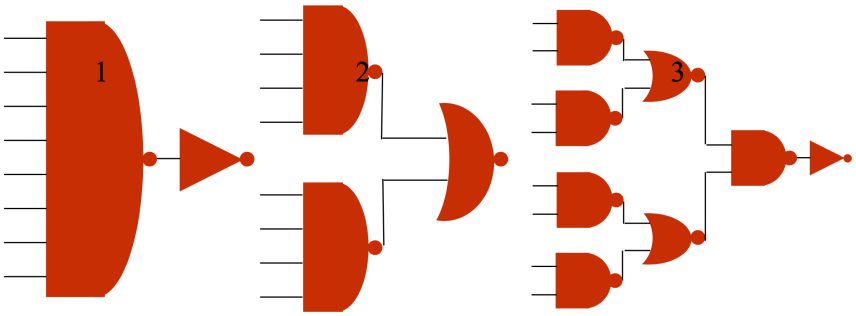
CSE 125: combinatorial logic block: where is the optimal place to insert a pipeline stage?

W**hat's important in HD?**

- performance > features
- performance > modularity
- reimplement same features until timing is met
- get a feeling for optimality

HDLs are dope. **A different mental model is required.**

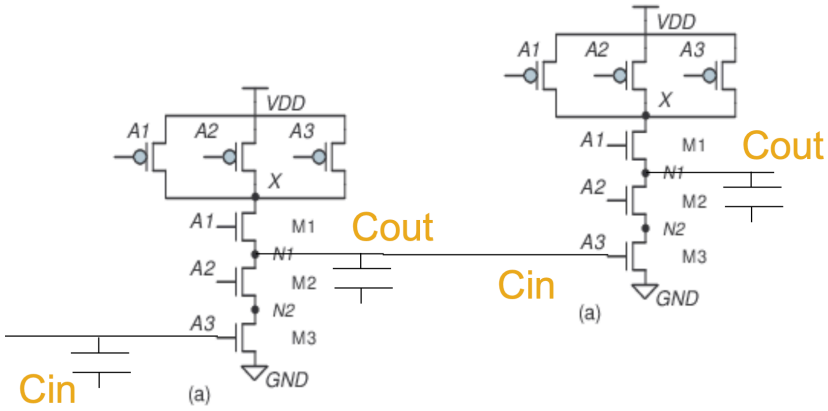
[NAND: Which is better?]



1: 1 level of logic directly produces the result.
2: 2 levels of logic produce the result.
3: 3 levels of logic produce the result.

Libraries provide different implementations of the same design. Which is best is not always clear. Performance depends on transistor implementation, the driving strength that each gate has, and transistor capacitance.

[3 input NAND gate: transistor implementation]
Multiple CMOS circuits are stacked. Propagation speed depends on: capacitors, driving strength of input and Vdd.



From an analog perspective, every wire, gate, transistor has a parasitic capacitance

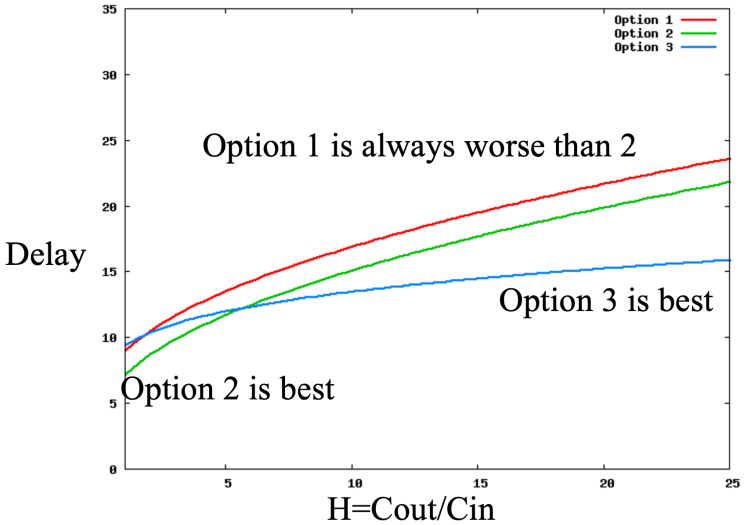
- wire – like a RC circuit with a resistance, capacitance

Cin and Cout
Worst case switching time is used to determine the time needed

- larger Cin capacitance → faster switching time
- larger Cout capacitance → longer switching time to charge/drain capacitance

Output Load Dependence

For a given logic unit
H = Cout / Cin gives the delay for different ratios



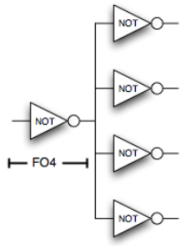
Logic Functions: Delay Guidelines

- think about the underlying hardware needed
- all parts of a design are made with basic logic cells (and, nor, xor, muxes, hal-adders…)
- understand the concept of FO4 when designing

FO4 is a unit that measures propagation delay of a circuit.

F04 is an architecture transparent time unit: **the delay to propagate signal through a NOT gate with a load of 4 NOT gates**

- unlike propagation delay which depends on # gates in sequence and speed of individual gate
- F04 is technology independent



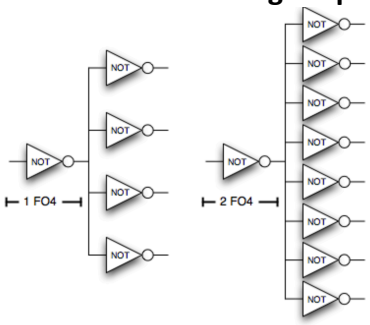
Tcycle = (# FO4) * 0.4 * (technology in nm)
1/(30FO4 * 0.4 * 180) ~ 460MHz
1/(30FO4 * 0.4 * 90) ~ 900MHz
1/(30FO4 * 0.4 * 65) ~ 1.2GHz

Take the number of FO4 delays and multiply by 0.4 and the size of technology in nanometers

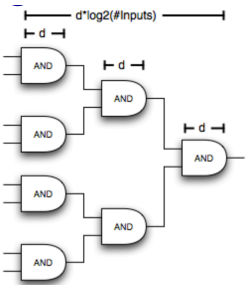
- limit pipeline stage to FO4 delays (very fine grain pipelining)
- FO4 is propagation of the depicted circuit
- technology transparent metric

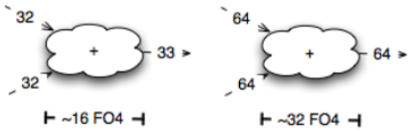
Logic Functions: Delay Rules

1. **Assume that doubling outputs (fan-out/load) doubles the delay.**



2. **Doubling the inputs (fan-in) can increase the delay from exponential to constant, depending on the operation.**





For example, add and subtract with half adders and full adders have a doubled FO4 when input is doubled. A bitwise inverter has constant time increase when inputs are doubled.

- changing # of inputs is harder to know how FO4 delay changes
- sometimes propagation delay does not change at all
- comparator has logarithmic result
- constant propagation delay
- bitwise negation

- what can I exploit?

Logic Functions: Wires

- FO4 measurements only consider logic delay.
- FPGAs have over 50% delay on wires
- ASICs have around 30% delay on wires

How can wire delay be optimized?

- try to place logic clouds close
- floorplan your architecture
- think about location of pipeline stages

For example, even for a bitwise invert, if it's large, wires get longer and there is an effect on performance.

Basic Test Benches in Verilog

```
`timescale 1ns/1ps
module my_test();
mymodule dut(sum,a,b,c);

reg a,b,c; // more about regs later!

initial begin
a = 0;
b = 0;
c = 0;
#10 a = 1;
#10 b = 1;
#10 a = 0;
    c = 1;
end
endmodule
```

All delays (#) in ns
Fractions rounded down to ps
Simulator is unit-less only knows
About integers

Instantiate DUT

Define logic values at time zero.
All undefined values are X

Logic values change at 10ns

Logic values change at 20ns

Logic values change at 30ns

Timescale directive in Verilog

`timescale time_unit/time_precision

time_unit: defines time of a #1 delay
time_precision: how fractions are rounded

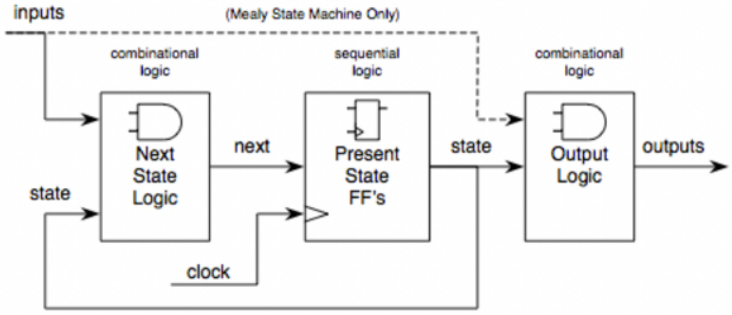
[Examples of using the timescale directive to define time delays]

```
`timescale 10ns/1ns
And #1.55 (z, a, b)          // z is change after 16ns

`timescale 1ns/1ps
And #1.00055 (z, a, b)       // z is changed 1.001 ns after
```

Finite State Machines

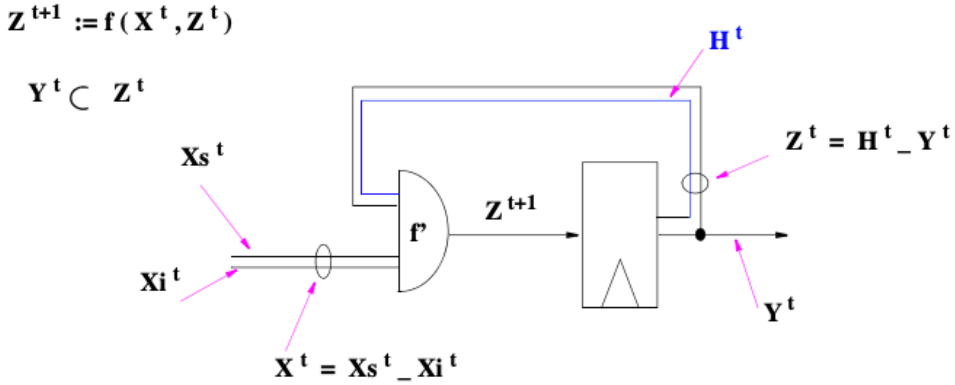
FlipFlops (registers) store the current state
Professor Litz recommends Moore because they are more predictable in terms of timing



SIMPLE MOORE STATE MACHINE

A simple Moore FSM uses the state register as output.

→ removes combinatorial output logic that allows input state to be fed through another logic cloud to generate output.



When designing hardware in collaboration with other engineers, you are prescribed a certain timing closure to meet.

→ Combinatorial logic after a state FF pushes propagation delay into the next pipeline stage!XD

→ A simple Moore FSM means no stealing.

→ utilize registers that hold outputs and move comb logic before the logic. Directly integrate it into the FSM logic.

Write simple Moore FSMs for this class: no combinatorial logic after the state FF. Each state FF is used directly as output.

Advantages of Simple Moore
• merging output with state flop updates gives more opportunities for logic optimization, (higher clocking frequency) • glitch free output b/c it comes directly from a register • fastest clock-to-output • simple modeling, optimization and synthesis

[Example: Simple Moore FSM]

```
module fsm (clk, res_n, i1,i2,i3, o1,o2,o3);
input clk, res_n, i1,12,13;
output reg o1,o2,o3;

parameter IDLE = 4'b0000; // state coding 'table'
parameter START = 4'b0001;
parameter WRITE = 4'b0010;
parameter READ = 4'b0100;

wire o1,o2,o3,h1;
reg [3:0] state, next_state;
assign inp = {i1, i2, i3};
assign {o1, o2, o3, h1} = state;
always @(*) begin //combinational transition block
  casex ({inp, state}) // each transition with one PT
    7'bxx0_0000: next_state <= IDLE;
    7'bxx1_0000: next_state <= START;
    {3'b010, START}: next_state <= WRITE;
    {3'b100, START}: next_state <= READ;
    ...
  endcase
end

always @(posedge clk or negedge res_n) begin //registers
  if (~res_n) begin
    state <= IDLE;
  end else begin
    state <= next_state;
  end
end
endmodule
```

tips

- refer to states using human readable terms. then assign them a particular bit vector