Avoiding Unwanted Latches: Rule 2

All inputs used in the procedure must appear in the trigger list

Any input change must recalculate the outputs.
If no recalculation is done, the old values must be remembered.
The synthesizer will insert latches to do this.

Things to Include

Right-hand side variables:

Except variables both calculated and used in the procedure.

```
always @(a or b or c or x or y)
   begin
     x=a; y=b; z=c;
     w=x+y;
   end
```

Branch controlling variables

The controlling variable for every `if` and `case`.

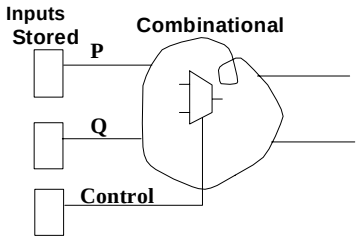
```
always @(r or s)
       begin
         if (r)          begin   x=2; y=0; z=0;  end
             elseif (s)  begin   x=0; y=3; z=0;  end
                  else   begin   x=0; y=0; z=4;  end
        end
```


Writing Procedural Code Without Latches II

Eliminating Latches

Let the inputs to a combinational logic block be held by latches, flip flops, or by input switches. Then the outputs only change if an input(s) change.

Moreover variables thought of as control variables are just as much inputs as those thought of as data.



Re-evaluation must be done if any input changes

The trigger list (event list) controls when the procedure is evaluated. This must contain all input variables.

Inputs

Data Inputs:

All inputs which appear on the right hand side in any operation.

However if they appear on both the right and left sides of expression, they are not included because the variable changing inside the loop would retrigger the loop. This could cause infinite zero-delay loops. It is hard to think of a legitimate synthesizable concept using a procedure that retriggers itself.

Control inputs

Any variable checked by the control of an *if* or *case* statement.

Other procedural operators do not cause branches or are not synthesizable.

11.• PROBLEM What latches, if any will be generated?

```
always@ (z or x)
   if (z==1) w=x; else w=~v;
```

More problems on next page.

Synthesis of Flip Flops and Registers

always @ Generates Flip-flops, Latches, or Combinational Logic

Flip-Flops

Positive-Edge Flip-Flops

```
always
   @(posedge Clk)
```

- This statement that tells the synthesizer to generate flip flops.
- There is a `negedge` also

Both-Edge Trigger

```
always
  @(C or D)
```

- This will give combinational logic.
  If all outputs are re-evaluated when any input changes.
- Otherwise it will generate a latch(es)

Rising-edge triggered flip-flop

```
wir D, Clk;
reg Q;
   always @(posedge Clk)
     begin
       Q <= D;
     end
// On positive edge Clk, D transfers to Q
// Otherwise Q holds its previous value,
// even if D changes.
```


Synthesis of Flip-flops

always @(posedge clk)

The synthesizer interprets this to mean flip flop(s)

This command, and only this command (or `always@(negedge clk)`) gives a flip-flop or a register of flip-flops.

12.• MORE PROBLEMS ON GENERATING UNWANTED LATCHES

Are any signals latched in the following code? Which ones?

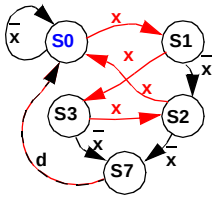
a)
```
always@(aziz or bob or chu)
     case (aziz)
        2'b00 : z=bob;
        2'b01 : z=chu;
        2'b11 : y=bob & chu;
        2'b10 : y=bob | chu;
     endcase
```

b)
```
always @(a or b or c) begin
     if (c) begin x=a; y=b; end  else y=b+x;
        x=3'd6; //  Never mind that this makes x=a redundant, the point is do we generate latches.
     end
```

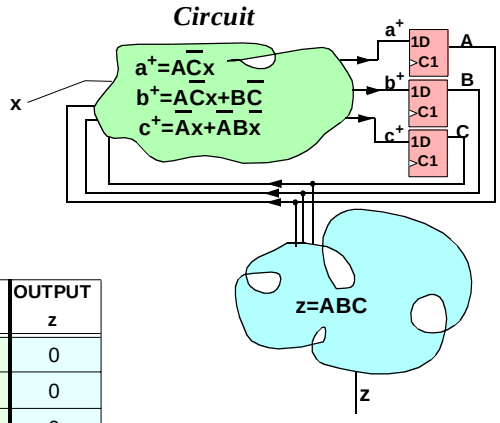
Finite State Machines

A State Machine Is Defined By Its *Next-state Table.*

State Graph and Next-State Table



State Graph



State Table

State ABC	Next State x=0	$a^+b^+c^+$ x=1	OUTPUT z
S0=000	S0=0	S1=1	0
S1=001	S2=2	S3=3	0
S3=011	S7=7	S2=2	0
S2=010	S7=7	S0=0	0
S7=111	S0=0	S0=0	1
Default	S0=0	S0=0	0

Tell long story about fail-safe here.

Finite-State Machines (FSMs)

This is a model for many circuits. For example counters are FSMs with no inputs.

State

The state is the collective contents of all the flip-flops (latches).

A state machine is described by

a. Its states and a description of their physical meaning.

b. The way the machine makes transitions to the next state.
These must be based on the present state and the present inputs only.

c. The outputs from each state.

Outputs

a. Moore Outputs: These may depend only on the state flip-flops.
Moore machines are easier to design and can give glitch free outputs.

b. Mealy outputs depend on the flip-flops and/or on the inputs directly.
Mealy machines usually have fewer states and thus are often smaller.

Standard Form for a Verilog FSM

```
// state flip-flops
reg [2:0] state, nxt_st;
// state definitions
parameter reset=0, S1=1, S2=2, S3=3, ..
```

```
// State Machine description
//   NEXT STATE CALCULATIONS
always @(state or . . . )
 begin

   next_state = ...

 end
```

```
// Separate the registers from
// the next state calculations.
// REGISTER DEFINITION
always @(posedge clk)
 begin
    state <= next_state
 end
```

special equals coming up soon

```
//  OUTPUT CALCULATIONS

output= f(state, inputs)
```


Standard Form for FSMs

Break FSMs into four blocks

State Definitions

The states must always be of type **reg.**

The states are normally given meaningful names rather than numbers. There are two common methods:

1. Use parameters as shown on the slide.
2. Use macros (**`define**) to do textual substitution when compiling. Synopsys suggests you use **`define** for global names and **parameters** for local names.

```
`define reset 0  // Use one `define per line and no semicolon.
`define S1 1
`define S2 2
  . . .
 if (x) next_state =`S1; else next_state = `S0; //Use a back quote whenever a macro is used.
```

Next State Calculations

Registers

Outputs

Do It My Way

All Verilog programmers expect finite state machines to be constructed this way. If you mix up these four parts, not only will you have much more trouble debugging your code, but any programmer reading your code will wonder who taught you!

Code Your FSMs This Way

```
module FSMzy(clk, x, z)
input clk, x;    output z;

// state flip-flops
reg [2:0] state, nxt_st;

// state definition
parameter S0=0, S1=1, S2=2, S3=3, S7=7;

// State Machine description using case
//   NEXT STATE CALCULATIONS
always @(state or x)
 begin
  case (state)
  S0: if (x) nxt_st = S1;
        else nxt_st = S0;
  S1: if (x) nxt_st = S3;
        else nxt_st = S2;
  S2: if (x) nxt_st = S0;
        else nxt_st = S7;
  S3: if (x) nxt_st = S2;
        else nxt_st = S7;
  S7:  nxt_st = S0;
  default:    nxt_st = S0;
  endcase
 end
```

```
// Separate the flip flops from
// the next state calculations.
// REGISTER DEFINITION
always @(posedge clk) begin
     state <= nxt_st;
   end
```

```
//  OUTPUT CALCULATIONS
assign z=(state = = S7);
```

```
endmodule
```


Generic Code for FSMs

Never never mix the next state calculations in with the flip flop definitions.

Next State Calculations

It is very common to use a combination of `case` and `if` for this block.
Note the `default` case handles all the state values not specifically mentioned in the `case`. If this were not put in then cases for state = 4, 5 and 6 would have to be explicitly mentioned or latches would be generated.

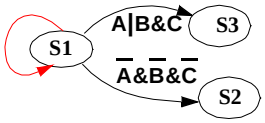
Output Calculations

Here they were so simple one `assign` statement could easily be used. No need to write an always block.

Avoiding Common Verilog Errors in FSMs

- **Cover every possible branch of every `if` or `case` to avoid latches.**
  Put default values at the start to use if nothing overwrites them. or put an `else` with every `if`.

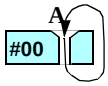
```
always @( state or A or B or C.......);
       begin
         next_state = S1;      // Use this as a default value;
         if (A | B&C)  next_state = S3;
         if ((~A)&(~B)&C)  next_state=S2;
       . . .
```



- **`State` must be a trigger variable in the procedure to update `next_state`.**

```
always @( state or A or B or .......);
  begin
      case (state)
```

- **A variable on the left side of the equal sign must not be in the trigger list or the machine may go into an infinite loop.**




```
always @( state or A or B or C.......);
  begin
    A = B+C;   // A will immediately retrigger the always procedure.
```


Avoiding Common Errors in FSMs

Cover Every Possible Branch

The alternative to writing over a default value is making the defaults appear in every else.

```
always @( state or A or B or C.......);
        begin
           if (A | B&C)       next_state = S3; else next_state = S1
           if ((~A)&(~B)&C)   next_state = S1; else next_state = S1;
           . . .
```

State must be a trigger variable

Also any variables in the conditions for any `if` must be in the trigger list.

```
always @( state or A or B or C.......);
        begin
           if (A | B&C) next_state = S3;
           if ((state == 3'b011) & C) next_state ...
```

13.• PROBLEM

```
parameter s1=1, s2=2, s0=0, s3=3;
always @(state or x or y)
   case(state)
   s1:  if (x&y) nxtstate=s2; elseif (x|y) nxtstate=s0;
   . . .
```

What is wrong with the above.

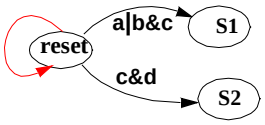
Avoiding Common Verilog Errors in FSMs

- End all case statements with the `default` case whether you need it or not.

```
case (state)
  . . .
  default:  next_state = reset;
endcase
```

- Do not forget the self loops in your state graph

```
case (state)
  reset: if (a|b&c) next_state=S1;
    else  if (c&d) next_state=S2;
    else  next_state = reset;    // and don't forget it!
```



- Always partition the FSM into:
  - a state definition part
  - a next-state calculation procedure
  - a register procedure. // This is the only part that should have a clock.
  - an output calculation part

- Using "=" instead of "<=" in the register procedure

```
always @ (posedge clk)
  Q <= D;
```


Avoiding Common Errors in FSMs

End all `case` statements with the `default` case

Do not confuse the "default case," which is part of the Verilog language, with the "default values" which are often used at the start of an always procedure. See Slide 28

The default case takes care of any cases you forgot and removes a major source of erroneous latches. Your code may be such that some cases never occur. Verilog is unlikely to be able to figure that out and will put in latches anyway unless you use the default.

Always partition the FSM

Even if you are so smart you can mix up the parts and make it work, the person who tries to maintain your code will be thoroughly confused.

Using "=" instead of "<=" in the register procedure

The "<=" is the "nonblocking" transfer symbol. (See Slide 41) Not using it will lead to obscure errors when:

1) The same variable appears on both the right and left sides in a procedure.

```
Q <= Q <<1   // As in this shift register
```

2) There are several `always @ (posedge clk)` registers in the design.
In this case not using the "<=" symbol can lead to races, which are discussed later.

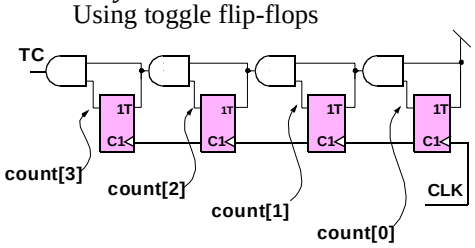
Simpler FSMs

Counters

Counters are a simple FSM machine.

Separation of the flip-flop generation code and the next-state code is not worth the effort.

Binary Counter
Using toggle flip-flops




```
reg [3:0] count; wire TC;
// TC=Terminal count (Carry out)

    always @(posedge clk or posedge rset)
      begin
          if (rset) count <= 0;
          else count <= count+1;
      end
assign TC = & count; // Reduction operator
```

In any @(posedge clk) procedure, use the nonblocking "<=" assignment operator. (see next slide)

Very Simple FSMs

When the Next State Calculation is One Simple Line

For very simple next state calculations we break the rule about separating the next state calculations from the registers.

14.• PROBLEM

a. Alternate counter code. Will this synthesize?[1]

```
always @(posedge clk or posedge rset)
   begin
      count <= count+1;
      if (rset) count <= 0;
   end
```

b. What does this code segment do?

```
reg [7:0] numb;
always @(posedge clk or posedge rset)
   begin
      if (du) numb = numb + 8'hff;
          else numb = numb + 1;
      if (rset) numb <= 0;
   end
```

1. a) From what you know now, it would. Check out Slide 38 to see the problem. b)Check out 2's compliment numbers.

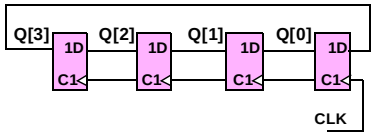
Very Simple FSMs

Shift Registers

The operator "<< N" shifts left N bits.
The operator ">>N" shifts right N bits.

Zeros are shifted in at the ends.

Shift register




```
reg [3:0] Q;
    always @(posedge clk or posedge rset)
        begin
            if (rset) Q <= 0;
            else begin
                Q   <= Q << 1; // Left shift 1
                Q[0] <= Q[3]; // Nonblocking:
// The old Q[3] is sent to Q[0].
//  Not the revised Q[3] from the previous line.
            end
        end
```

The Nonblocking assignment operator "<="

For variables on the right of "<=" (Q here) are grabbed as the procedure is triggered.
These old values are used for calculating the lefthand side.

Simple FSMs (cont)

We break the separation rule again.

15.• PROBLEM

In a right shift with two's complement numbers, the most significant bit replicates itself during the shift. Thus 10101 shifted right once becomes 11010. This is called sign extension.

Revise the right shift code below to give the correct answer for two's complement numbers.

```
reg [7:0] Q
always @(posedge clk or posedge rset)
   begin
      if (rset) Q <= 0;
      Q <= Q >> 1;
   end
```

Procedural Synthesis

Logic Inference

Deciding what logic to synthesize from code is called *inference*.

always @ can infer:
flip-flops,
latches, and/or
combinational logic.

Flop-Flops

```
always @(posedge Clk)
```

- **This is the statement that tells the logic compiler to generate flip flops.**

Latches and Combinational

```
always @(C or D)
```

- **This may generate a latch.**
  It may give just combinational logic.

Combinational

- **If any input change causes a recalculation of all outputs.**

Latches

- **For any output which is not recalculated for all possible input changes.**

Logic Inference

Generating Logic From Procedures

A major concern is that synthesized logic and simulation both yield the same result. This is certainly not always true. Three examples that sometimes do not match are:

1. One has an incomplete trigger list but does not generate a latch. See Comment on Slide 33.
2. The use of functions which never infer latches.See Slide 35.
3. Assigning the same variable in two different procedures. See Slide 45.

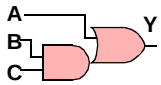
Combinational Inference

- Used to allow procedural code (if, for, case, . .) in combinational logic.
- Should include all inputs in trigger list.

Result of Synthesizing Poor Code

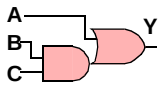
Bad Combinational Gate

```
reg Y;
    always @(A or B)
        Y = A | (B&C);
. . .
```



Good Combinational Gate

```
. . .
    always @(A or B or C)
        Y = A | (B&C);
. . .
```



For the Bad gate

- **Simulator:**
  C changes are only noted when triggered by a change in A or B.
- **Compiler:**
  Generated an AND-OR gate
- **SNAWS**
  Simulation may Not Agree With Synthesis

for the Good Gate

- Including all variables in the trigger list removed all problems.

Incomplete Trigger Lists

What happens

The *Design Compiler* from Synopsys™ generated an AND-OR gate here as specified. My theory is that because the gate was so explicitly expressed it decided to generate it without the latch.

Other synthesizers might put in latches.

One synthesis engine, no longer available, made a logic block which returned a constant Y=0.

Moral

Put all the input variable in the trigger list unless you are trying to build latches.
Input variables are:
all variables on the right-hand side of the equal sign,
all control variables for **if** and **case**.

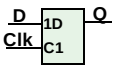
Latch Inference

Inserting Latches With or Without Your Asking

Latches From IFs

Latch Inference from if

```
reg Q;
    always @(Clk or D)
    begin
      if (Clk) Q <= D;
    end
```



There is no else

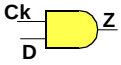
HDLs infer no `else` as "hold the old value."

Here a latch was wanted.

Often the else is ignored through ignorance and generates a latch.

No Latch Inference from if

```
reg Z;
    always @(Ck or D)
    begin
      Z<=1'b0; //Initialize
      if (Ck) Z <= D;
    end
```



There is an effective else

Either use an `else`. or initialize before the `if`.

Then no latch is assumed.

```
always @(Ck or D)
    begin
      if (Ck==1) Z <= D;
      else Z <= 0; //Used else
    end
```


Latch Insertion in Combinational Ifs

- Latches are inserted if the **else** branch is not explicitly stated.
  This is a very common error.
- The easy way to make sure all else cases are covered is to assign a default value to all outputs at the start of the procedure. Then use the `if` statements to overwrite it.

Latch Inference (Cont)

Latches Inferred From Non-full Case

Latch Inference from case

```
// decimal-decoder
  wir [3,0] in;
  reg [10:1] Y;
    always @(in)
      case(in)
      4'h1: Y=0000000001;
      4'h2: Y=0000000010;
      4'h3: Y=0000000100;
      4'h4: Y=0000001000;
      4'h5: Y=0000010000;
      4'h6: Y=0000100000;
      4'h7: Y=0001000000;
      4'h8: Y=0010000000;
      4'h9: Y=0100000000;
      4'ha: Y=1000000000;
      endcase
```

These are undefined cases

```
      4'h0: Z=0000000000;
      4'hb: Z=0000000000;
      4'hc: Z=0000000000;
      4'hd: Z=0000000000;
      4'he: Z=0000000000;
      4'hf: Z=0000000000;
```

If one occurs Z will stay at its previous values.
Thus synthesis will infer 10 latches.

To avoid latches

Either include all cases,

or equivalently use

```
    . . . .
      4'h9: Y=0100000000;
      4'ha: Y=1000000000;
  default: Y=0000000000;
```


Latches Generated When Case is Not Full

Full Case

A full case is when an output is specified for all 2^N cases. Where N is the number of bits in the case control.

If the case is not full, latches will be generated, even the unspecified cases can never happen.

16.• PROBLEM

Suggest another method of avoiding latches in a non-full case without using `default`.
Hint, put in a value that will be overwritten.

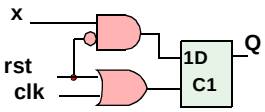
Synthesis of Latches With Synopsys

Asynchronous Reset for Latches

Theoretical Latch

```
always @(clk or reset or x)
if(rst) Q <= 0;
  else if (clk) Q <= x;
```

Synthesizes into

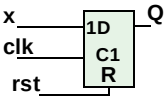


Logically correct but not what is desired

Add a //Synopsys command

```
// Synopsys async_set_reset "rst"
always @(clk or rst or x)
if(rst) Q <= 0;
  else if (clk) Q <= x;
```

Synthesizes into



With the Ambit synthesizer

Use

```
// ambit synthesis set-reset asynchronous  signal = "rst"
```


Synthesis of Latches

Synopsys has trouble telling what an asynchronous reset is for latches.

The Verilog code given does not seem to be enough to tell it to look for a standard cell with an asynchronous reset input.

```
// Synopsys async_set_reset "rst"//
```

is needed to tell it to find the right standard cell.

Functions

A handy way to write repetitive code.

Rotate W if A =1

```
module rot_rt_3(Z,W,A);
  input [4:0] W; input A;
  output [4:0] Z;
  reg [4:0] X,Y,Z;
    always @(W,A)
    begin
      X=rot_f(W);
      Y=rot_f(X);
      Z=rot_f(Y);
    end
  function [4:0] rot_f;
    input [4:0] w;
    reg [4:0] z; //local varb
      z={w([0],w[3:1]};
      rot_f=(A)?z:w;
      end
  endfunction
endmodule
```

Functions

- Simpler to write than submodules.
- Functions forget everything between calls. Hence latches are not generated.
- Functions contain no timing control, delay or event checking statements.
- Functions generate only combinational logic.

Inputs: Outputs

- No arguments; inputs defined by input statement.
- Other variables (like A) are known inside function.
- One output via function call. For more outputs, concatenate variables.

Scope

- Functions must be contained within the module that uses them.
- Their argument passing is too weak to go outside.
- Functions can call functions.

Functions

Use functions for clarity and to save writing

A function can be used to define a block of code which will be repeated.

1. It is clearer and shorter than typing multiple detailed copies.
2. It is a little simpler than using submodules:
   a. The call is slightly easier to read.

      `X=rot_f(W);` seems a little clearer to me than `rot_m rot1(X,W);`

   b. `A` reader, seeing a function, knows immediatelly it is only combinational logic. This is not known for a submodule without checking.

   c. Variable definitions are simpler:

- Out put is defined in function definition line.
  `function [3:0] rot_f(W);`
- Variables not passed by arguments declared outside the function, are known by the function. See "A" below, used in **`rot_f`** on the slide.
  `rot_f = (A)? w: z;` // A defined outside the function.
- Reg type variables, declared outside the function, are known inside the function and can be changed inside the function. See "B" below.
  `reg B, function fun1(W); input w; begin B=~w; ... endfunction.`
- Local variables can be defined inside the function when needed. See "z" below, used in **`rot_f`** on the slide.
  `reg [3:0] z;` // defined in the function and not known outside the function..
- The order of arguments in the function input statement must be the same order as in the invocation.
  `X=fun1(A,W,B); ... function fun; input a; input [3:0] w; input b; ...`

Functions (cont)

Functions Never Infer Latches

No Latch Inference from if

```
module nolatch(Z,D,C);
  input D,C;
  output Z; reg Z;
    always @(C or D)
    begin
      Z=and_func(C,D)
    end

  function and_func;
    input c,d;
    reg z;
      begin
        if (c) z=d;
        // No else
        and_func=z;
      end
  endfunction
endmodule
```

Functions

- One or more inputs.
- Functions contain no timing control, delay or event checking statements.
- Arguments are passed by order of call via the order of the input statement.

Functions generate only combinational logic.

- Functions forget everything between calls. Hence latches are not generated.

But no clear principle says what takes the place of inferred latches.

- Avoid latch-inferring inferences.
- Functions must be contained within a module. Their argument passing is too weak to go outside.

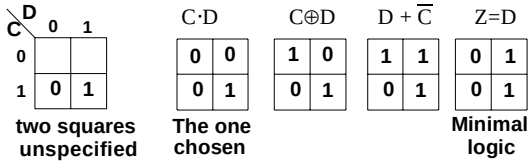
Functions Never Infer Latches

An incomplete specification

The lack of an else leaves two squares in the Karnaugh map undefined.

The possible combinatorial alternatives are:-

$Z=C\cdot D$, $Z=C\oplus D$, $Z=D+\overline{C}$, or $Z=D$.



The **and_function** shown was sent through the Design Compiler, and it generated an AND gate.

It did not generate the simplest logic treating the unspecified cases as don't cares. That would give Z=D. It did treat the unspecified cases as zero which gives $Z=C\cdot D$.

Functions subject to SNAWS[1]

Until a definite principle is generally known, assume anything may come out of the unspecified case.

Tasks

Task are like functions but can contain multiple outputs and timing information, but not @(posedge clk). Tasks are not treated in these notes or by most synthesizers.

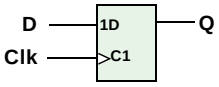
1. Simulation does Not Agree With Synthesis

Flip-Flop Inference

Flip-Flops

Positive-Edge Flip-Flops

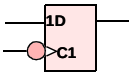
```
wir D, Clk;
reg Q;
  always @(posedge Clk)
    Q <= D;
```



Negative-edge trigger

`always @(negedge C)`

If a negative edge ff in library fine. If not check what happens,



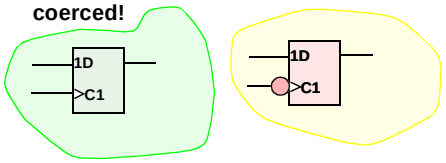
Both in one design

Are you sure you want to do this?

Group in separate levels of hierarchy to keep timing analysis simple.

Will your test methods cover having both at once?

Automatically inserted scan has to be coerced!



Flip-Flop Inference

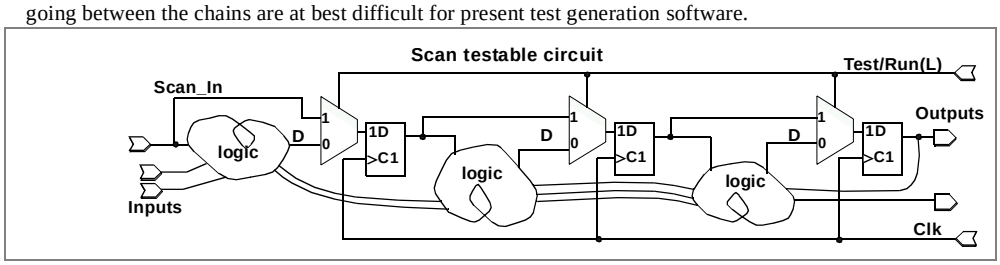
Negative Edge Trigger

Many libraries do not have falling-edge triggered flip-flop. The synthesizer may then insert an inverter in the clock line. Unless the circuit is all falling edge triggered, this will cause clock skew, and the synthesizer will then insert inverters in the data lines to fix hold time violations.

Scan Testing

Scan testing is a very common test system. The flip flops all have muxs connected to their inputs. When Test/Run = 0 the circuit runs normally. When Test/Run = 1, the flip-flops are all connected as a shift register. The shift register makes it very easy to set the flip-flops to any values. Then Test/Run is set to 0 to inject these values into the logic and perform a test on the logic.

If some of the flip-flops are falling edge triggered, then one needs two scan chains (shift registers) and the tests going between the chains are at best difficult for present test generation software.



Why Mix Clock Edges

A few circuits, like RAMbus, use both clock edges. Conversion from these inputs could use one or two opposite edge flip-flops although it is simpler to use latches. Other uses, such as eliminating hold time violations in shift registers, usually have better fixes.

Flip-Flops With Asynchronous Reset

Synthesis is Fussy

Flip-Flop With Asynchronous Reset

```
wir D, Clk, Rst;
reg Q;
always @(posedge Clk
              or posedge Rst)
  begin
    if (Rst) Q<=0;
    else Q <= D;
  end
```

If the flip flop resets when high

- Use negedge
- Use (! Rst_n) in the if condition.

```
always @(posedge Clk
              or negedge Rst-
n)
    begin
      if (!Rst_n) Q<=0;
      else Q <= D;
```

This Format Must Be Followed

- **Reset and Clk are single-bit variables**
- **always @(---edge Clk) or ---edge Rst)**
  if (Rst) . . .
  else
  //what happens on active clk edge
  endif
- **A second reset must go in an else if**
- **if immediately follows:**
  always @(. . .) begin
- **Else automatically assumes it will only be done on a positive(neg) Clk edge.**
- **Condition is restricted to:**
  if (Rst)
  if (~Rst_n)
  if (!Rst_n)
  if (Rst = = 1'b1)
  if (Rst_n = = 1'b0)
  ~~if (R[1])~~ is not allowed.
  ~~if (R > (2-1))~~ is not allowed.

Flip-Flops With Asynchronous Reset

Synthesis assumes that anything with @(posedge . . .) is a flip flop and does not leave any freedom for creative coding.

17.• PROBLEM

```
always @(posedge clk or posedge rst or posedge set)
   if (rst&(~set)) state<=start;
      elseif (set) state<=4'hf;
      else begin
         state<=nxtstate;
      end
```

What is wrong with the above for synthesis?

Flip-Flops With Synchronous Reset

Easier than asynchronous

Typical Synch Reset Code

```
reg Q;
always @(posedge Clk)
  begin
    if (Rst) Q<=0;
    else Q <= D;
  end
```

Much Less Rigid Format

- Leave off the `or posedge Rst`.
- The `if - else` could be replaced by:

  `Q<= (!R)&D;`

Synchronous resets are easy, but-

They are not as good as asynchronous resets.

1. They have setup and hold times
2. Be careful using them for "power up" reset.
   a. The reset signal may violate the setup time.
   b. Machine will end up half in the reset state, and half in state one.
3. If clock dividers are reset, the flip flops they feed may never see the reset.

Flip-Flops with Synchronous Reset

These have much more freedom for coding but are less useful.

Awkward properties of resets

When applying a synchronous reset, one must be sure that all the flip flops to be rest are supplied with at least one active clock edge during reset. If some flip flops are supplied from a clock divider, and the flip flops in the divider are reset, then divided clock will never appear during reset. This is a disadvantage of synchronous reset.

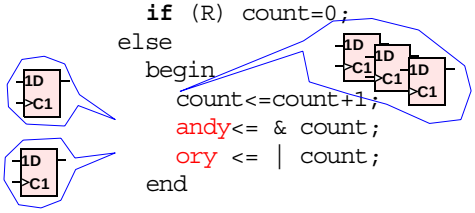
Asynchronous reset is like any other asynchronous signal. When it changes on the clock edge, some flip flops will get the old value and some the new. This means some of the flip flops will stay in reset another cycle, and the others will come out of reset now. To avoid this one must synchronize the reset signal, that is pass it through a D flip flop before applying it to the other flip flops.

In general the best reset is applied asynchronously and removed synchronously. However the synchronization is done to the reset signal. The individual flip flops still use an asynchronous reset.

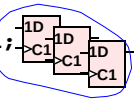
Minimizing Flip-Flops

Do Not Declare Extra Registers Inside An @(posedge ---) Procedure

```
reg [2:0] count
reg andy, ory; ;
 always @(posedge Clk or posedge R)
   begin
       if (R) count=0;
     else
       begin
          count<=count+1;
          andy<= & count;
          ory <= | count;
       end
```




```
wire andy, ory
]always @(posedge Clk or posedge R)
     if (R) count=0;
     else count<=count+1;
  assign andy= & count;
  assign ory = | count;
```



Too much inside @(posedge ---)

- andy and ory do not need storage. Storing count is enough.
- Move them outside into combinational logic.
- The upper square will generate 5 ff, the lower square 3 ff.

Minimizing Flip-Flops

The concept is very simple. Partition your finite state machines and do not include outputs in the block which is only supposed to contain registers.

18.• PROBLEM

What would you have done differently if you had written the code below.[1]

```
always @(posedge clk or posedge rst)
   if (rst) state<=start;
      else begin
         state<=nxtstate;
         nxtstate= state & (x^y);
      end
```

1. Mixing "=" and "<=" won't synthesize. nxtstate will be stored when it should not be.

Blocking or Nonblocking

Using "=" or "<="

Blocking "="

Blocking assignment is like C code.

The next assignment waits (is blocked) until the present one is finished.

```
x = A + B;
z = x + D;   // z will use the new x value
```

Nonblocking "<="

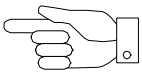
Nonblocking is like flip flops.

The inputs are grabbed and held at the time the procedure is triggered.
All the assignments use these held values.

```
always @(posedge clk)
x <= A + B;
z = x + D; // z will use the old value x had as the clk changed
```

Rule For Using "=" and "<="

Use "=" for combinational logic.
Use "<=" for registers, latches and flip flops.
Don't mix them in one procedure.



Blocking and Nonblocking

Nonblocking

In synthesis nonblocking will act as though all right-hand variables were sampled on procedure entry

Even for variables calculated within the procedure.

Example

```
always @(a or b or c)
      b<=a;
      if (b)      // will be the old b
```

Blocking

Blocking means calculations for the next statement are blocked until the present statement is completed.

```
initial begin
   #1 e=2;
   #1 b=1;   // completed at t=2
   #1 b<=0 // completed at t=3. A previous blocking statement delays both blocking and nonblocking.
         e<=b; // completed at t=3; the preceding statement is nonblocking so this grabbed the old b=1.
         f=e;   // completed at t=3, using the old e=2. It did not wait for e<=b to complete.
```

Rule for Synthesizable Procedures

Use blocking "=" for all combinational logic.
Allows assignments to depend on previous assignments like C code does.

Use nonblocking "<=" for flip-flops and registers.
Blocking behaves like D-flip-flops, transferring all data simultaneously.

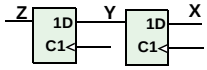
Use nonblocking after always@(posedge clk) in synchronous test benches.

Blocking and Nonblocking (Example)

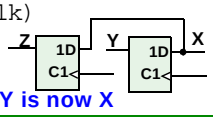
Shift Registers

```
wir Clk, X;
reg Z,Y;
```

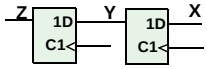
```
always @(posedge Clk)
  begin
    Z=Y; Y=X;
  end
```



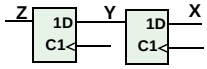

```
always @(posedge Clk)
  begin
    Y=X; Z=Y;
  end
```




```
always @(posedge Clk)
  begin
    Z<=Y; Y<=X;
  end
```




```
always @(posedge Clk)
  begin
    Y<=X; Z<=Y;
  end
```



The Nonblocking Assignment "<="

- This is like a real flip-flop.
  On the clock edge,
  the old outputs are grabbed and used as the right-hand side inputs.
- The outputs are all revised based on these grabbed inputs.

The Blocking Assignment "="

- Like a C++ program.
- Statements at the top can change inputs beneath them.

You Can't Use Both

- Verilog for simulation lets you use a reasonable mix of "=" and "<=."
- Synthesizers will not allow both in one block.

Blocking and Nonblocking

Top Figure

Normal shift register

Z=Y; // Z get old Y

Y=X; //Y gets input X

Next Figure Down

Y=X; // Y gets input

Z=Y; // In C this would also make Z=X. So does it here.

Bottom Two Figures

Here the values of Y and X are saved when the procedure is entered.
Changing Y on the left (output) side does not change it on the input side.