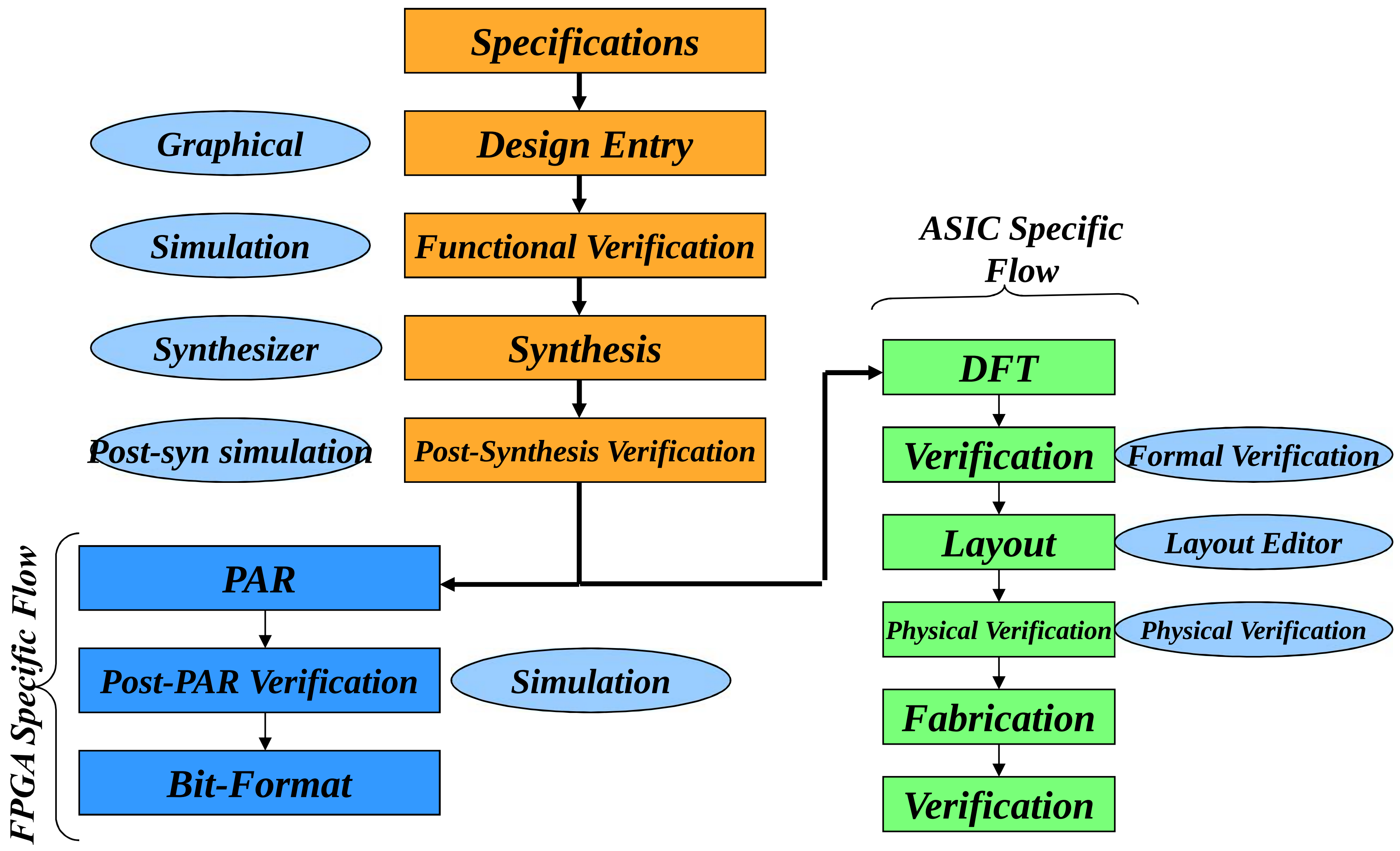
DIGITAL SYSTEM DESIGN USING VERILOG

CONTENTS

DESIGN FLOW IN ASIC / FPGA



NEED FOR HARDWARE DESCRIPTION LANGUAGE (HDL)

Model, Represent, And Simulate Digital Hardware

- Hardware Concurrency
- Parallel Activity Flow
- Semantics for Signal Value And Time

Special Constructs And Semantics

- Edge Transitions
- Propagation Delays
- Timing Checks

VERILOG HDL

Basic Unit – A module

Module

- Describes the functionality of the design
- States the input and output ports

Example: A Computer

- Functionality: Perform user defined computations
- I/O Ports: Keyboard, Mouse, Monitor, Printer

MODULE

General definition

```
module module_name ( port_list );
        port declarations;
        …
        variable declaration;
        …
        description of behavior
endmodule
```

❑ Example

```
module HalfAdder (A, B, Sum Carry);
input A, B;
output Sum, Carry;
assign Sum = A ^ B;
//^ denotes XOR
assign Carry = A & B;
// & denotes AND
endmodule
```

CONVENTIONS

Comments

```
// Single line comment
/* Another single line comment */
/* Begins multi-line (block) comment
    All text within is ignored
   Line below ends multi-line comment
*/
```

Number

decimal, hex, octal, binary

unsized decimal form

size base form

include underlines, +,-

String

" Enclose between quotes on a single line"

CONVENTIONS (CONT.)

Identifier

- A … Z
- a … z
- 0 … 9
- Underscore

Strings are limited to 1024 chars

First char of identifier must not be a digit

Keywords

Operators

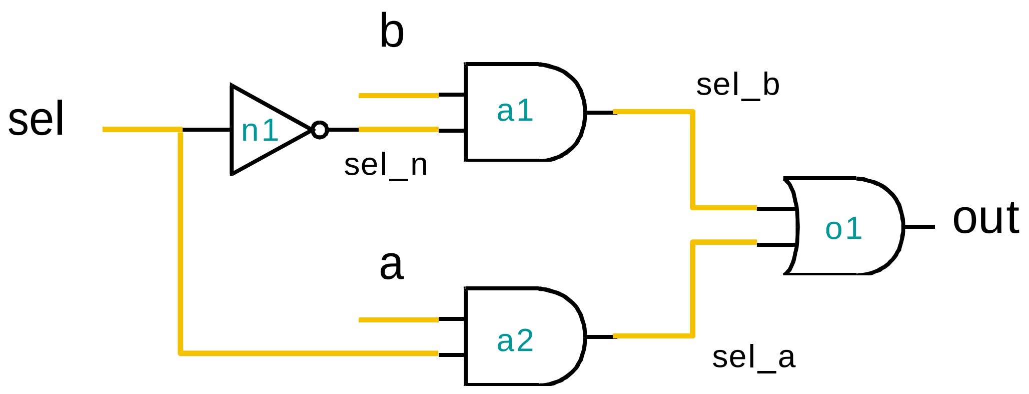
Verilog is case sensitive

DESCRIPTION STYLES

Structural: Logic is described in terms of Verilog gate primitives

Example:

```
not  n1(sel_n, sel);
and a1(sel_b, b, sel_b);
and a2(sel_a, a, sel);
or   o1(out, sel_b, sel_a);
```

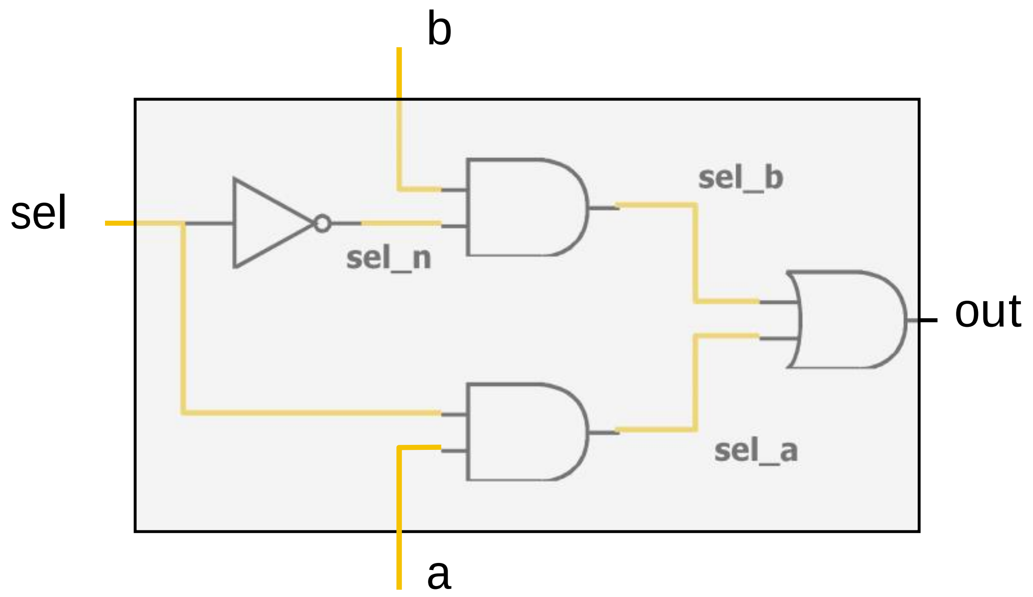


DESCRIPTION STYLES

Dataflow: Specify output signals in terms of input signals

Example:

```
assign out = (sel & a) | (~sel & b);
```

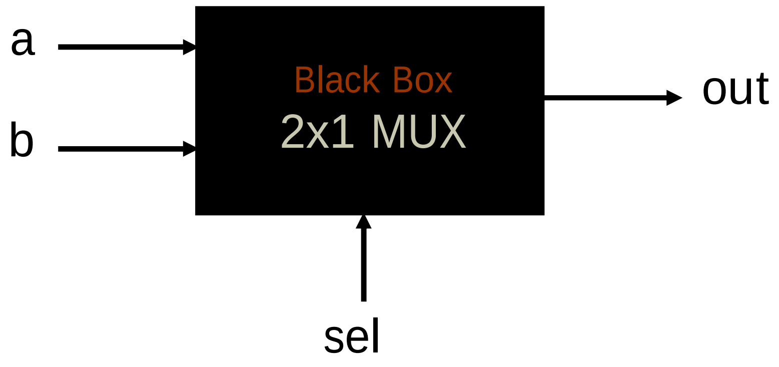


DESCRIPTION STYLES

Behavioral: Algorithmically specify the behavior of the design

Example:

```
if (select == 0) begin
        out = b;
end
else if (select == 1) begin
        out = a;
end
```



STRUCTURAL MODELING

Execution: Concurrent

Format (Primitive Gates):

```
and G2(Carry, A, B);
```

First parameter (Carry) – Output

Other Inputs (A, B) - Inputs

DATAFLOW MODELING

Uses continuous assignment statement

- Format: assign [delay] net = expression;
- Example: assign sum = a ^ b;

Delay: Time duration between assignment from RHS to LHS

All continuous assignment statements execute concurrently

Order of the statement does not impact the design

DATAFLOW MODELING

Delay can be introduced

- Example: assign #2 sum = a ^ b;
- "#2" indicates 2 time-units
- No delay specified : 0 (default)

Associate time-unit with physical time

- `timescale time-unit/time-precision
- Example: `timescale 1ns/100 ps

Timescale

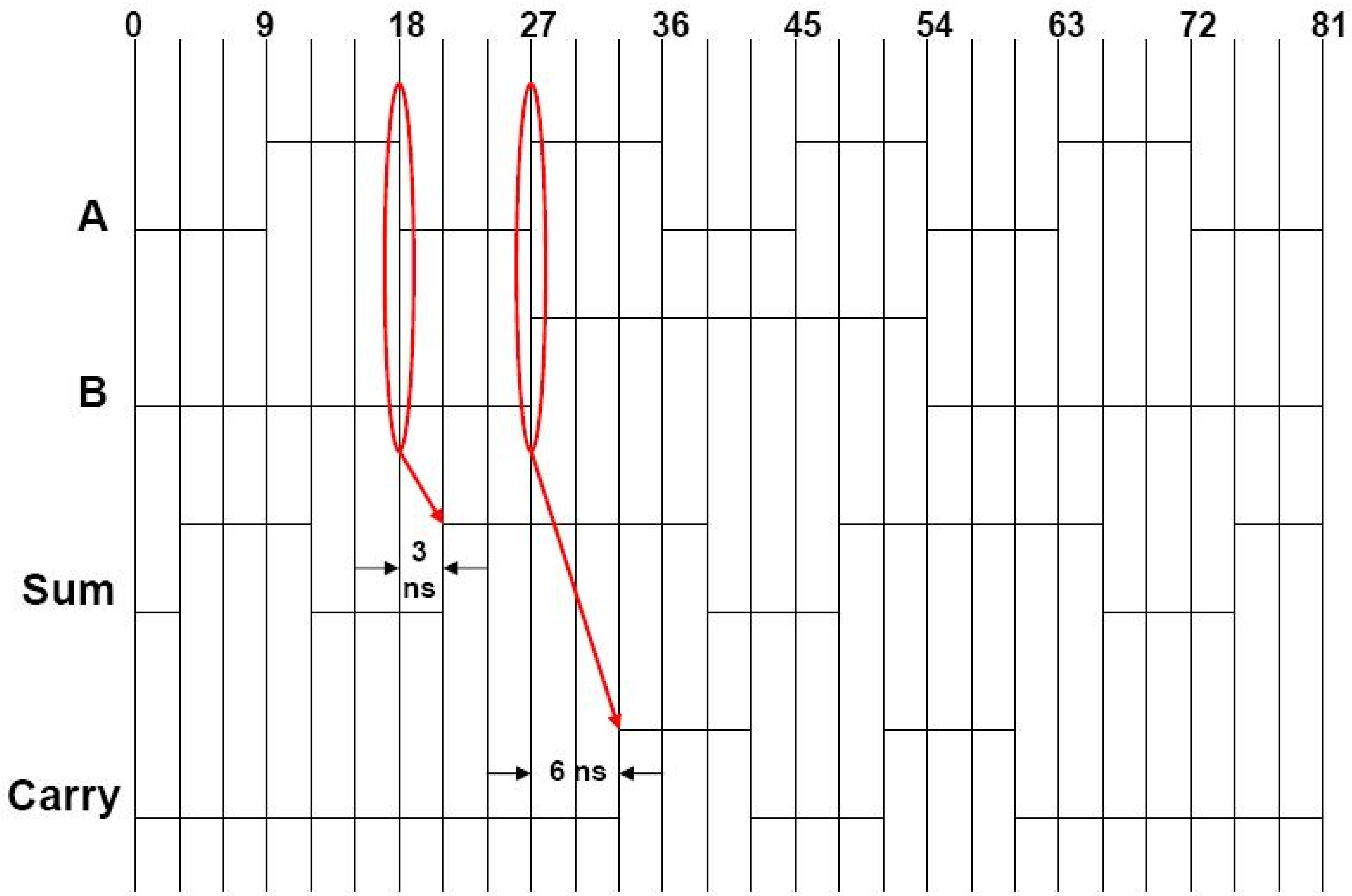
`timescale 1ns/100ps

- 1 Time unit = 1 ns
- Time precision is 100ps (0.1 ns)
- 10.512ns is interpreted as 10.5ns

DATAFLOW MODELING

Example:

```
`timescale 1ns/100ps
module HalfAdder (A, B, Sum, Carry);
   input A, B;
   output Sum, Carry;
   assign #3 Sum = A ^ B;
   assign #6 Carry = A & B;
endmodule
```


0
9
18
27
36
45
54
63
72
81
A
B
Sum
Carry
3 ns
6 ns

Example:

```
module mux_2x1(a, b, sel, out);
   input a, a, sel;
   output out;
   always @(a or b or sel)
   begin
         if (sel == 1)
              out = a;
      else out = b;
   end
endmodule
```

Sensitivity List

BEHAVIORAL MODELING (CONT.)

always statement : Sequential Block

Sequential Block: All statements within the block are executed sequentially

When is it executed?

- Occurrence of an event in the sensitivity list
- Event: Change in the logical value

Statements with a Sequential Block: Procedural Assignments

Delay in Procedural Assignments

- Inter-Statement Delay
- Intra-Statement Delay

BEHAVIORAL MODELING (CONT.)

Inter-Assignment Delay

- Example:

```
Sum = A ^ B;
#2 Carry = A & B;
```

- Delayed execution

Intra-Assignment Delay

- Example:

```
Sum = A ^ B;
Carry = #2 A & B;
```

- Delayed assignment

PROCEDURAL CONSTRUCTS

Two Procedural Constructs

- initial Statement
- always Statement

initial Statement : Executes only once

always Statement : Executes in a loop

Example:

```
...
initial begin
  Sum = 0;
  Carry = 0;
end
...
```

```
...
always @(A or B) begin
  Sum = A ^ B;
  Carry = A & B;
end
...
```

EVENT CONTROL

Event Control

- Edge Triggered Event Control
- Level Triggered Event Control

Edge Triggered Event Control

```
@ (posedge CLK) //Positive Edge of CLK
  Curr_State = Next_state;
```

@ negedge	@ posedge
$1 \to x$	$0 \to x$
$1 \to z$	$0 \to z$
$1 \to 0$	$0 \to 1$
$x \to 0$	$x \to 1$
$z \to 0$	$z \to 1$

Level Triggered Event Control

```
@ (A or B) //change in values of A or B
  Out = A & B;
```

LOOP STATEMENTS

Loop Statements

- Repeat
- While
- For

Repeat Loop

- Example:

```
repeat (Count)
  sum = sum + 5;
```

- If condition is a x or z it is treated as 0

LOOP STATEMENTS

While Loop

- Example:

```
while (Count < 10) begin
  sum = sum + 5;
  Count = Count +1;
end
```

- If condition is a x or z it is treated as 0

For Loop

- Example:

```
for (Count = 0; Count < 10; Count = Count + 1) begin
  sum = sum + 5;
end
```

CONDITIONAL STATEMENTS

if Statement

Format:

```
if (condition)
   procedural_statement
else if (condition)
   procedural_statement
else  procedural_statement
```

Example:

```
if (Clk)
   Q = 0;
else
   Q = D;
```

CONDITIONAL STATEMENTS

Case Statement

Example 1:

```
case (X)
  2'b00: Y = A + B;
  2'b01: Y = A – B;
  2'b10: Y = A / B;
endcase
```

Example 2:

```
 case (3'b101 << 2)
   3'b100: A = B + C;
   4'b0100: A = B – C;
   5'b10100: A = B / C; //This statement is executed
endcase
```

CONDITIONAL STATEMENTS (CONT.)

Variants of case Statements:

- casex and casez

casez – z is considered as a don’t care

casex – both x and z are considered as don’t cares

Example:

```
casez (X)
  2’b1z: A = B + C;
  2’b11: A = B / C;
endcase
```

DATA TYPES

Net Types: Physical Connection between structural elements

Register Type: Represents an abstract storage element.

Default Values

- Net Types : z
- Register Type : x

Net Types: wire, tri, wor, trior, wand, triand, supply0, supply1

Register Types : reg, integer, time, real, realtime

DATA TYPES

Net Type: Wire

wire [msb : lsb] wire1, wire2, …

- Example

  wire Reset; // A 1-bit wire

  wire [6:0] Clear; // A 7-bit wire

Register Type: Reg

reg [msb : lsb] reg1, reg2, …

- Example

reg [3: 0] cla; // A 4-bit register

reg cla; // A 1-bit register

RESTRICTIONS ON DATA TYPES

Data Flow and Structural Modeling

- Can use only ***wire*** data type
- Cannot use ***reg*** data type

Behavioral Modeling

- Can use only ***reg*** data type (within initial and always constructs)
- Cannot use ***wire*** data type

MODELING OF COMBINATIONAL LOGIC CIRCUITS

```
module ha(a,b,sum,carry);
    input a,b;
    output sum,carry;
 assign sum = a ^ b;
 assign carry = (a & b);
endmodule
```

MODELING OF SEQUENTIAL LOGIC CIRCUITS

```
module counter(
    input rst,clk,dir,
    output reg  [3:0] q
    );
always@(posedge (clk))
begin
if (rst)
q<=4’b0000;
else if (dir)
q<=q+4’b0001;
else
q<=q-4’b0001;
end
endmodule
```

TEST BENCH AND FUNCTIONAL SIMULATION

```
module counter_test  ;
  wire  [3:0]  q   ;
  reg    rst   ;
  reg    clk   ;
  reg    dir   ;
  counter
   DUT  (
       .q (q ) ,
      .rst (rst ) ,
      .clk (clk ) ,
      .dir (dir ) );
```

```
// "Clock Pattern" : dutyCycle = 50
// Start Time = 0 ns, End Time = 1 us,
Period = 20 ns
  initial
  begin
              clk  = 1'b0  ;
             # 10 ;
// 10 ns, single loop till start period.
   repeat(49)
   begin
               clk  = 1'b1  ;
              #10  clk  = 1'b0  ;
              #10 ;
// 990 ns, repeat pattern in loop.
   end
              clk  = 1'b1  ;
             # 10 ;
// dumped values till 1 us
  end
```

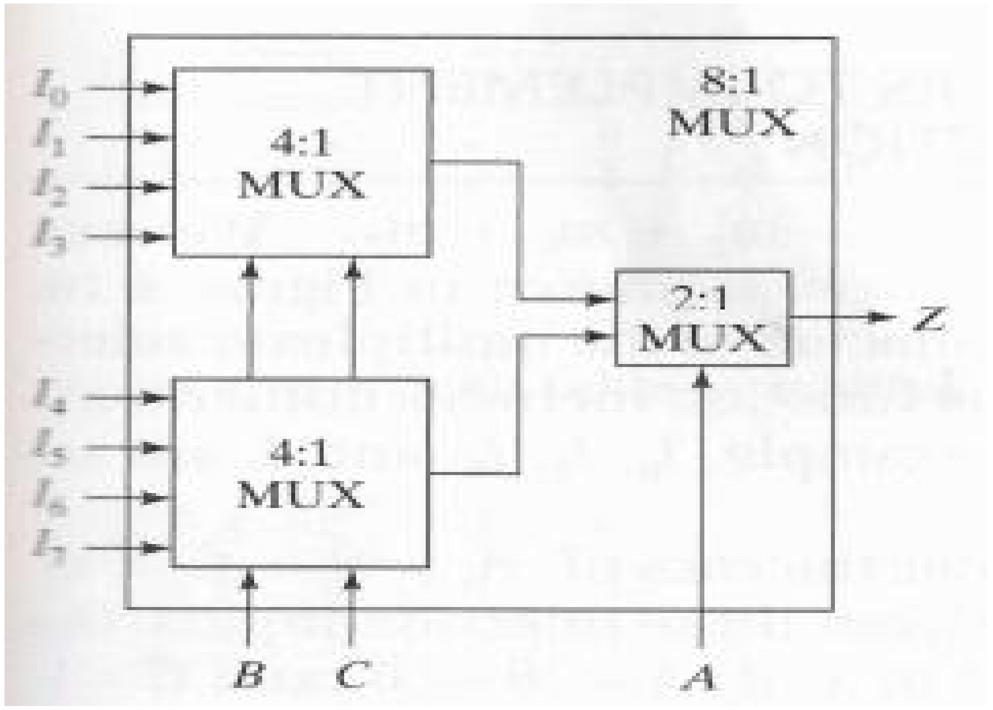
TEST BENCH AND FUNCTIONAL SIMULATION

```
// "Constant Pattern"
// Start Time = 0 ns, End Time = 1 us, Period = 0 ns
  initial
  begin
             rst  = 1’b0  ;
            # 0        rst  = 1’b1  ;
            # 20       rst  = 1’b0  ;
            # 980 ;
// dumped values till 1 us
  end
```

```
// "Constant Pattern"
// Start Time = 0 ns, End Time = 1 us, Period = 0 ns
  initial
  begin
             dir  = 1’b0  ;
            # 0        dir  = 1’b1  ;
            # 500      dir  = 1’b0  ;
            # 500 ;
// dumped values till 1 us
  end

  initial
           #2000 $stop;
endmodule
```

Exercise

Design a Mux 8:1 using Verilog according to the following structure and verify the functionality using suitable test bench



EXERCISE

Write an RTL and Test bench to verify

- D-flip flop with asynchronous active low reset
- D-flip flop with synchronous active high reset

MEMORIES

An array of registers

```
reg [ msb : lsb ] memory1 [ upper : lower ];
```

Example

```
reg [ 0 : 3 ] mem [ 0 : 63 ];
// An array of 64 4-bit registers
reg mem [ 0 : 4 ];
// An array of 5 1-bit registers
```

COMPILER DIRECTIVES

‘define – (Similar to #define in C) used to define global parameter

Example:

```
‘define BUS_WIDTH 16
 reg [ ‘BUS_WIDTH - 1 : 0 ] System_Bus;
```

‘undef – Removes the previously defined directive

Example:

```
‘define BUS_WIDTH 16
   …
   reg [ ‘BUS_WIDTH - 1 : 0 ] System_Bus;
   …
‘undef BUS_WIDTH
```

COMPILER DIRECTIVES (CONT.)

‘include – used to include another file

Example

```
‘include “./fulladder.v”
```

SYSTEM TASKS

Display tasks

- $display : Displays the entire list at the time when statement is encountered
- $monitor : Whenever there is a change in any argument, displays the entire list at end of time step

Simulation Control Task

- $finish : makes the simulator to exit
- $stop : suspends the simulation

Time

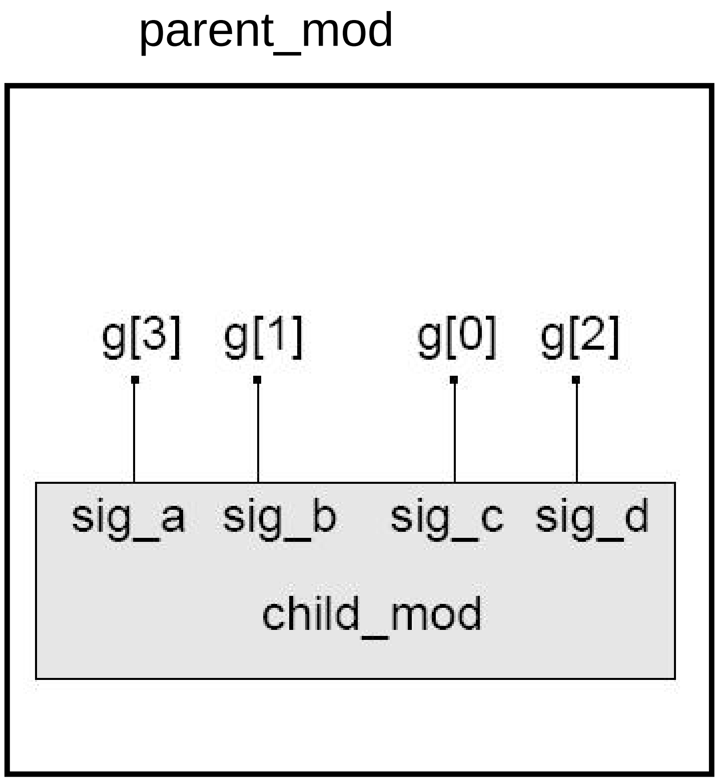
- $time: gives the simulation

TYPE OF PORT CONNECTIONS

C

```
module child_mod (sig_a, sig_b,
sig_c, sig_d);
 input      sig_a, sig_b;
 output     sig_c, sig_d;

 // module description goes here.

endmodule
```

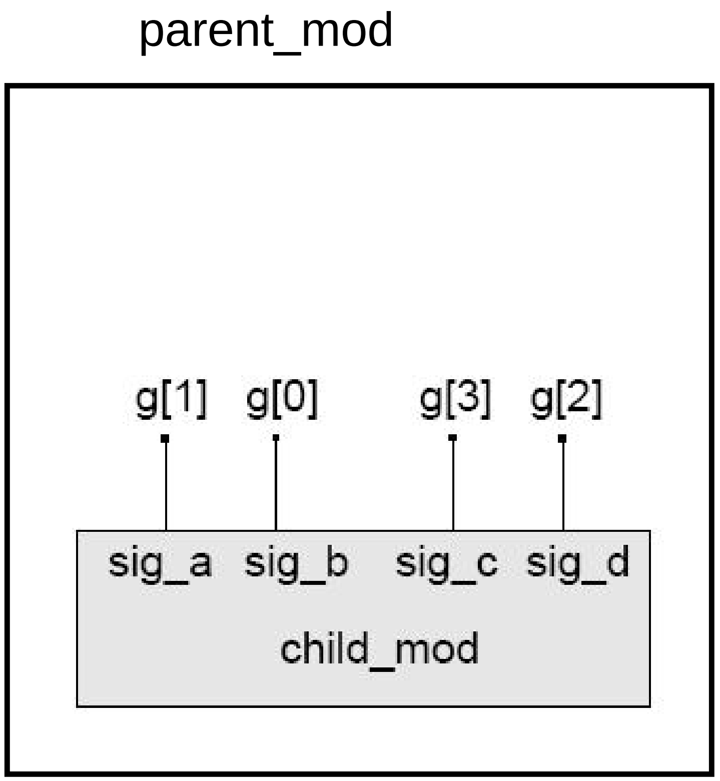
```
module parent_mod;
  wire [4:0] g;

  child_mod G1 (g[3], g[1], g[0], g[2]);

  // Listed order is significant
endmodule
```



TYPE OF PORT CONNECTIONS (CONT.)

```
module child_mod (sig_a, sig_b,
sig_c, sig_d);
  input      sig_a, sig_b;
  output     sig_c, sig_d;

  // module description goes here.

endmodule
```

```
module parent_mod;
  wire [3:0] g;
    // Listed order not significant

   child_mod G1 (   .sig_c(g[3]),
                    .sig_d(g[2]),
                    .sig_b(g[0]),
                    .sig_a(g[1]));
endmodule
```



EMPTY PORT CONNECTIONS

If an input port of an instantiated module is empty, the port is set to a value of z (high impedance).

```
module child_mod(In1, In2, Out1, Out2)
  input In1;
  input In2;
  output Out1;
  output Out2;
 //behavior relating In1 and In2 to Out1
endmodule
```

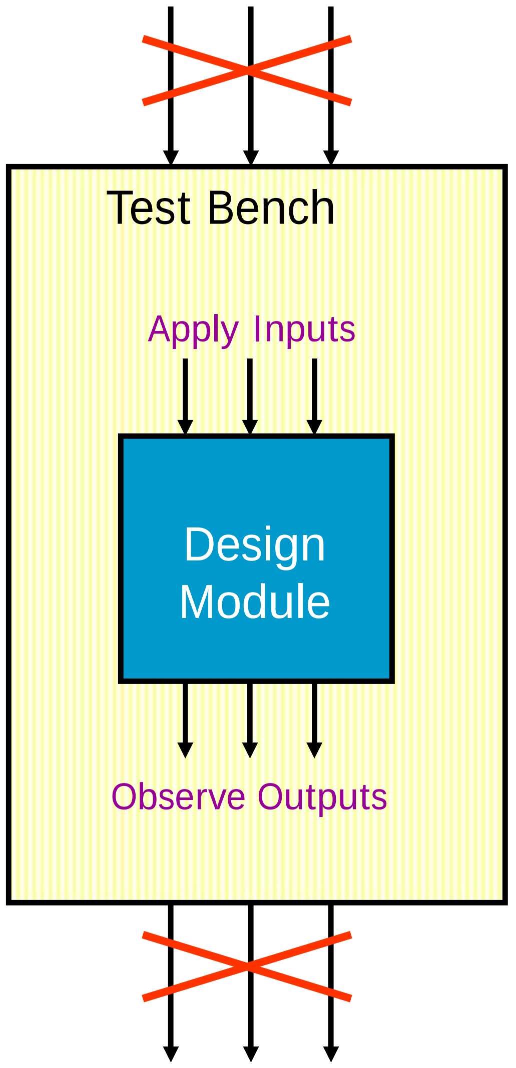
```
module parent_mod(…….)

        child_mod mod(A, ,Y1, Y2);
        //Empty Input
    endmodule
```

If an output port of an instantiated module is left empty, the port is considered to be unused.

```
module parent_mod(…….)
        child_mod mod(A, B, Y1,  ); //Empty Output
       endmodule
```

TEST BENCH

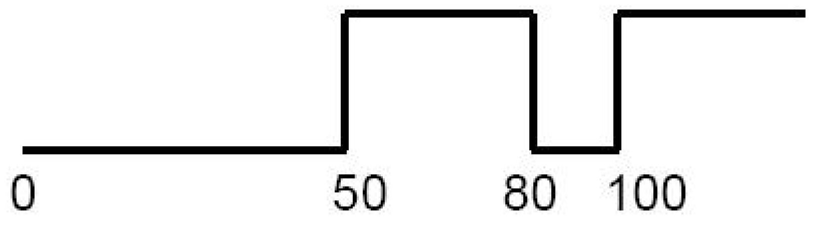
```
‘timescale 1ns/100ps
module Top;
        reg PA, PB;
        wire PSum, PCarry;

        HalfAdder G1(PA, PB, PSum, PCarry);

        initial begin: LABEL
         reg [2:0] i;
         for (i=0; i<4; i=i+1) begin
          {PA, PB} = i;
          #5 $display (“PA=%b PB=%b
PSum=%b     PCarry=%b”, PA, PB, PSum,
PCarry);
         end // for
        end // initial
endmodule
```



TEST BENCH - GENERATING STIMULUS

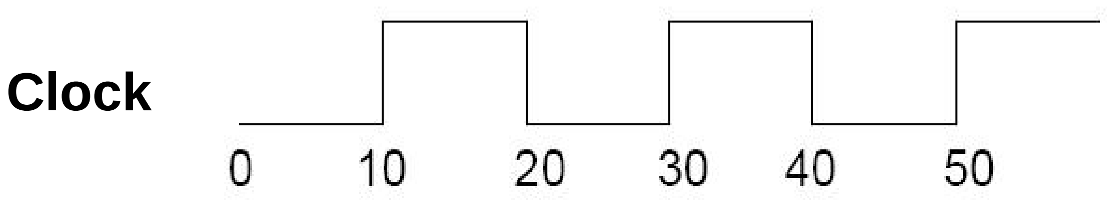
Example: A sequence of values

```
initial begin
  Clock = 0;
  #50 Clock = 1;
  #30 Clock = 0;
  #20 Clock = 1;
end
```



TEST BENCH - GENERATING CLOCK

Repetitive Signals (clock)



A Simple Solution:

```
wire Clock;
assign #10 Clock = ~ Clock
```

Caution:

- Initial value of Clock (***wire*** data type) = z
- ~z = x and ~x = x

TEST BENCH - GENERATING CLOCK (CONT.)

Initialize the Clock signal

```
initial begin
    Clock = 0;
end
```

Caution: Clock is of data type *wire*, cannot be used in an *initial* statement

Solution:

```
reg Clock;
…
initial begin
  Clock = 0;
end
…
always begin
#10 Clock = ~ Clock;
end
```

forever loop can also be used to generate clock

DESIGN ENTRY METHODS

Depending on the design complexity & functionality a digital logic block can

be represented by using one of the following approaches:

1.RTL Coding using HDL(i.e VHDL/Verilog HDL etc.)

2.Truth Table representation

3.State Diagram

4.Flow chart

5.Block diagram

FUNCTIONAL SIMULATION

Functional emulation of a circuit design through software programs is Simulation.

It is the process of applying stimuli to a model over time and producing corresponding responses from the model

Simulation is used for design verification:

- Validate assumptions
- Verify logic
- Verify performance (timing)

STEPS IN SIMULATION

Compilation :

- **Checks VHDL source code for syntax and semantic rules of VHDL.**
- **If a syntax or semantic error occurs then the compiler gives an error message.**

Elaboration :

- **Ports are created for each instance of component**
- **Memory storage is allocated for each required signals**
- **Interconnections among the port signals are specified**
- **Executes VHDL processes/ Verilog constructs in proper sequence mechanism**

STEPS IN SIMULATION

Initialization :

Here initial values present in the declaration statement are assigned to either signals or variables.

Execution :

Here simulator accepts simulation commands like (run, assign, watch) ,which controls the simulation of the system and specify desired output.

Every process is executed until it suspends and signal values are updated after the process suspends.

Interconnections among the port signals are specified

Simulation ends when signals are updated with new values

TYPES OF SIMULATION

Functional Simulation :

- Ignores timing aspects.
- Verifies only the functionality of the design

Behavioral Simulation :

- Given functionality is modeled using HDLs.
- Timing aspects are considered.
  Example : f<=a and b after 5ns.

Gate level Simulation :

- Its used to check the Timing performance of a design.
- Delay parameters of Logic cells are used to verify timings
- Gate delays are vendor specific.

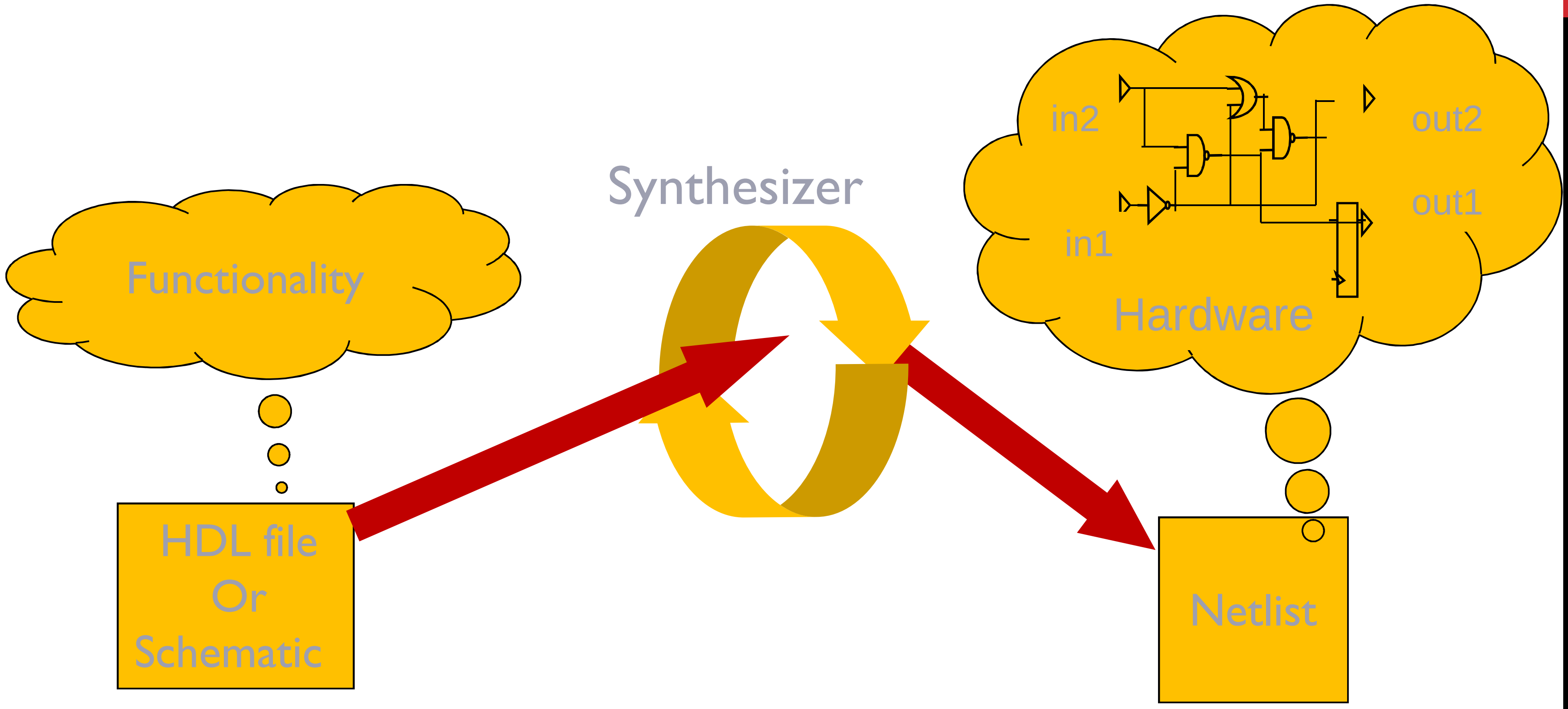
SYNTHESIS

“ Synthesis process converts user’s Hardware description into structural logic description.

Synthesis provides a means to convert schematics or HDL into real-world hardware.

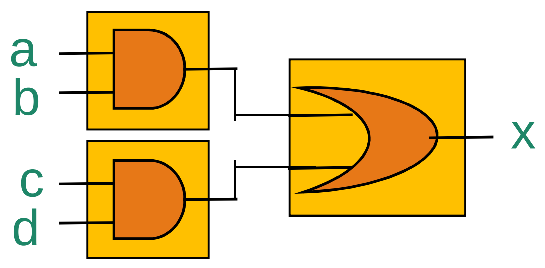
Synthesis tools convert the described hardware into a net list that a vendor may use to create the chip or board.

Logic Synthesis

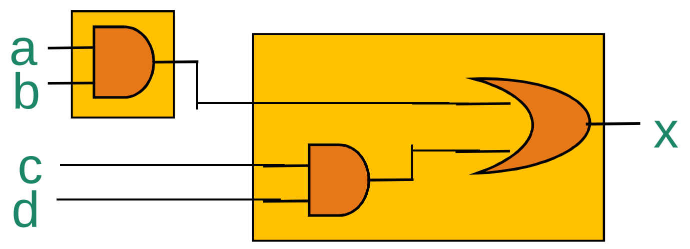


ASIC & FPGA Synthesis

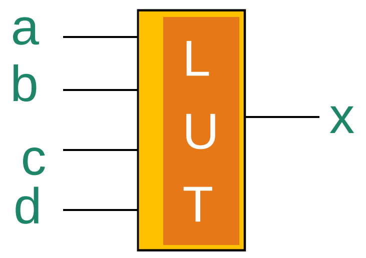
X = (a and b) or (c and d) ;



Generic



ASIC



FPGA

SYNTHESIS – DESIGN FLOW

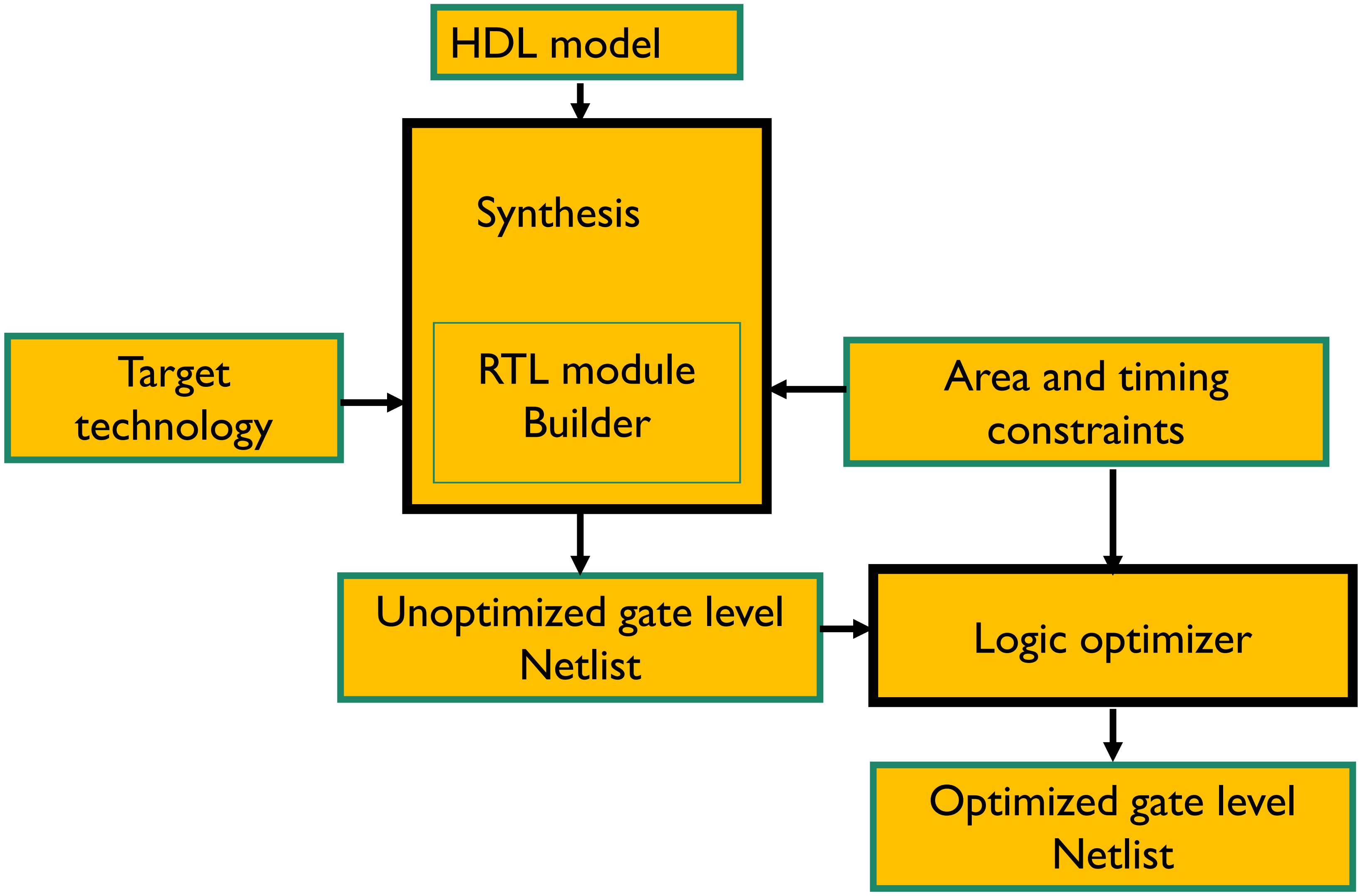
Synthesis and Optimization processes generate a gate level net list for the target technology.

This net list can be optimized under various constraints, such as area or speed.

Synthesis process is as follows:

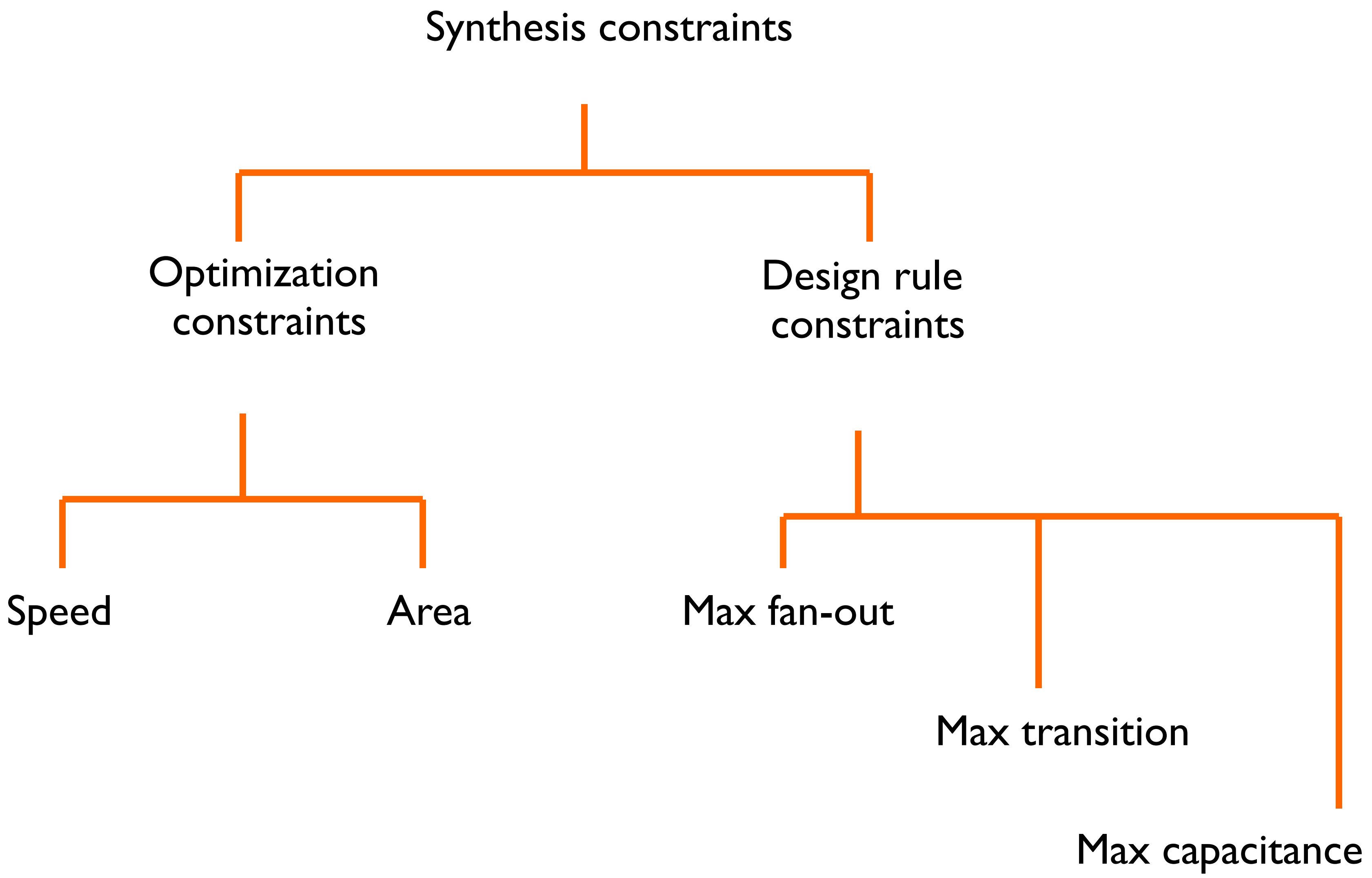
- Input data is transformed into a description based on Boolean Equations.
- A Technology independent logic level optimization based upon the Boolean Equations is performed by applying rules of Boolean Algebra.
    - **This eliminates redundant logic and reduces the area requirement or delay of the circuit.**
    - **Logic level optimization is split into phases:**
        - **Flattening and Structuring.**

SYNTHESIS PROCESS



HDL model
Synthesis
Target technology
RTL module Builder
Area and timing constraints
Unoptimized gate level Netlist
Logic optimizer
Optimized gate level Netlist

SYNTHESIS CONSTRAINTS



SYNTHESIS ADVANTAGES

Synthesis is constraint driven :

-It meets design criteria of Timing , Size , and Power.

- ❑ **A synthesized design can be re synthesized into new technologies.**
- ❑ **Forces higher levels of Abstraction.**
- ❑ **Leads to ease in debugging and code Portability.**
- ❑ **Enables designer to focus on larger design goals, as it is easier to relate RTL to hardware.**

SYNTHESIS TOOLS - FEATURES

Synthesis tools should provide features such as

- Control over Critical Path Synthesis.
- Remove all false paths.
- Resource Sharing of Adders, Incrementers, Decrementers, and multipliers.
- Automatic RAM Inference.
- Replicate logic to meet Fan out limitations.
- Remove unused logic , duplicate flip-flops etc.
- Provide optimization across Design Hierarchy.

CODING FOR SYNTHESIS

Good coding style means that the Synthesis tool can identify constructs within your code that it can easily map to technology features.

All programmable Devices may have its own unique architectural resources. For ex: Xilinx Virtex series has inbuilt RAM.

- Readymade codes(Macros) are provided by vendors which are optimized to the target technology

  Ex. LogiBlox in Xilinx

Note: For more information on coding guidelines for synthesizable HDL refer to:

HDL Coding guidelines

Reference: **http://standards.ieee.org/getieee/1800/download/1800-2012.pdf**

MODELING FOR SYNTHESIS

Design methodology uses synthesis tool to convert RTL code to library cells

- Code is interpreted & hardware created
    - Knowledge of FPGA architecture is helpful
- Synthesis tools require certain coding to generate correct logic
    - Subset of Verilog language supported
    - The goals of coding style are efficiency and predictability
- Initialization controlled by device

Pre- & post-synthesis logic should operate the same

WRITING SYNTHESIZABLE VERILOG

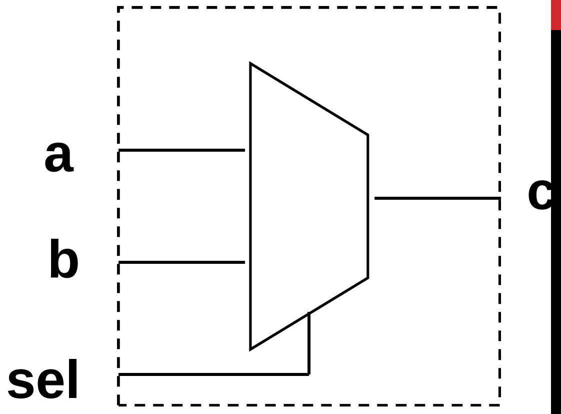
- **Sensitivity lists**
- **Blocking vs. non-blocking**
- **Latches vs. registers**
- **if-else structures**
- **case statements**
- **Combinatorial loops**
- **Internally generated clocks**
- **Using functions and tasks**
- **Inferring logic functions**

TWO TYPES OF PROCESSES

- **Combinatorial Process**
  - Sensitive to all inputs used in the combinatorial logic

always @(a **or** b **or** sel**)**

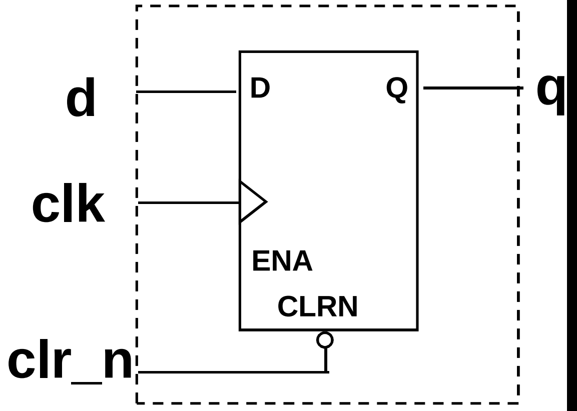
sensitivity list includes all inputs used in the combinatorial logic



- **Clocked Process**
  - Sensitive to a clock or/and control signals

always @(posedge clk **or negedge** clr_n**)**

sensitivity list does not include the **d** *input, only the clock or/and control signals*



SENSITIVITY LIST

Incomplete sensitivity list in an always block may result in differences between RTL and gate-level simulations

```
always @ (a or b)
        y = a & b & c;
```

Incorrect Way – the simulated behavior is not that of the synthesized 3-input AND gate

```
always @ (a or b or c)
        y = a & b & c;
```

Correct way for the intended AND logic !

BLOCKING VS. NON-BLOCKING

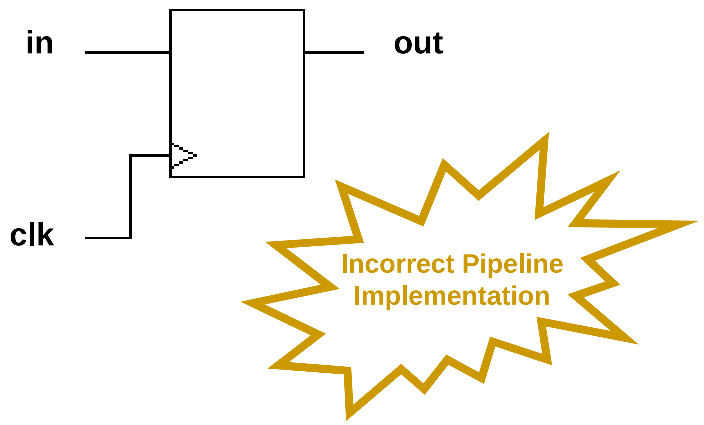
Recommendation: Use non-blocking assignments for clocked processes when writing synthesizable code

BLOCKING VS. NON-BLOCKING EXAMPLE

Blocking (=)

```
Always @(posedge clk)
    begin
            a = in;
            b = a;
            out = b;
    end
```

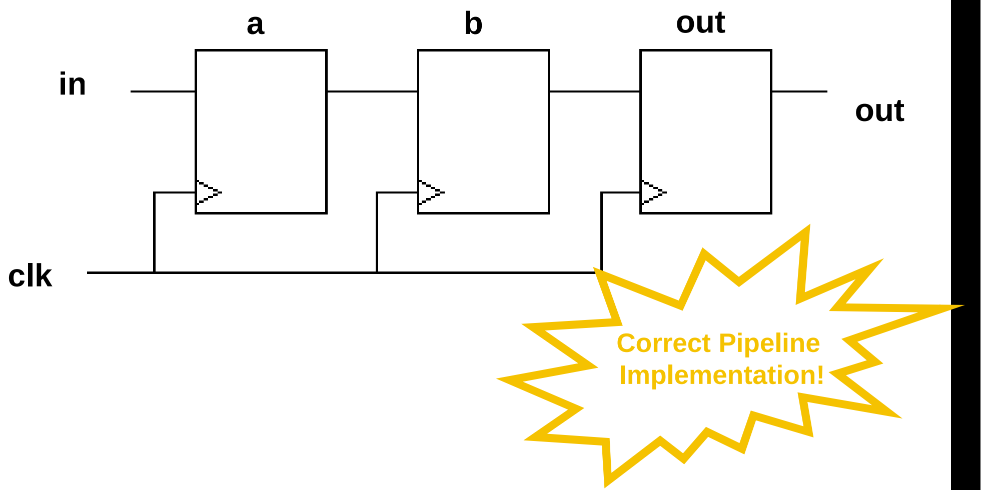
__Synthesized Result__:



Nonblocking (<=)

```
Always @ (posedge clk)
    begin
            a <= in;
            b <= a;
            out <= b;
    end
```

__Synthesized Result__:



LATCHES VS. REGISTERS

Latches are implemented using combinatorial logic & can make timing analysis more complicated

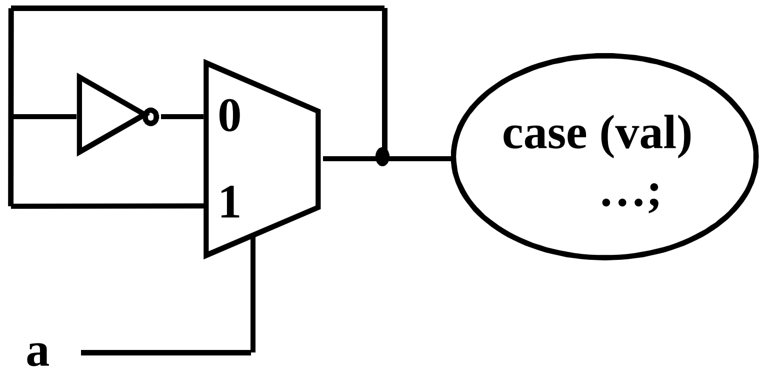
Recommendations

- Design with registers (RTL)
- Watch out for inferred latches
    - Latches inferred on combinatorial outputs when not results not specified for set of input conditions

VARIABLE NOT INITIALIZED

Reading a data type object before it has been assigned a value infers a latch

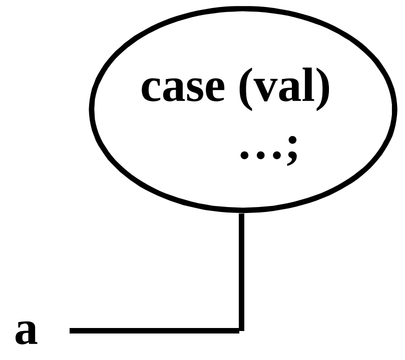
```
always @ (i or a)
      begin
            if (a == 1) val = val;
            else val = val + 1;
            case (val)
                        1'b0 : q = i[0];
                        1'b1 : q = i[1];
            endcase
      end
```



ASSIGNING INITIAL VALUE TO VARIABLE

Assigning a value to a data type object prior to reading it prevents a latch

```
always @ (i or a)
        begin
                val = 1'b0;
                if (a == 1) val = val;
                else val = val + 1;
                case (val)
                                1’b0 : q = i[0];
                                1’b1 : q = i[1];
                endcase
        end
```



IF-ELSE STRUCTURE

if-else structure implies prioritization and dependency

If sequential statements are mutually exclusive, individual if structures may be more efficient

When writing if and if-else structures

- Cover all cases
    - Uncovered cases result in latches
- Assign values before reading data type object
    - Reading unassigned data type object results in latches
- For efficiency, consider assigning initial values and explicitly covering only those results different from initial values

PRIORITIZATION & DEPENDENCY (IF-ELSE)

Nested else if statements imply prioritization of and dependency in logic

```
reg [4:0] state;
parameter  s0 = 5‘h0, s1 = 5‘h11, s2 = 5‘h12, s3 = 5‘h14, s4 = 5‘h18;
always @ (posedge clk or negedge reset_n) begin
        if (reset_n == 0) state <= s0;
        else begin
                if (state == s0) begin
                        if (in == 1) state <= s1;  else state <= s0;
                end
                else if (state == s1) state <= s2;
                else if (state == s2) state <= s3;
                else if (state == s3) state <= s4;
                else if (state == s4) begin
                        if (in == 1) state <= s1;  else state <= s0;
                end
        end
end
```

PRIORITIZATION/DEPENDENCY REMOVED (IF-ELSE)

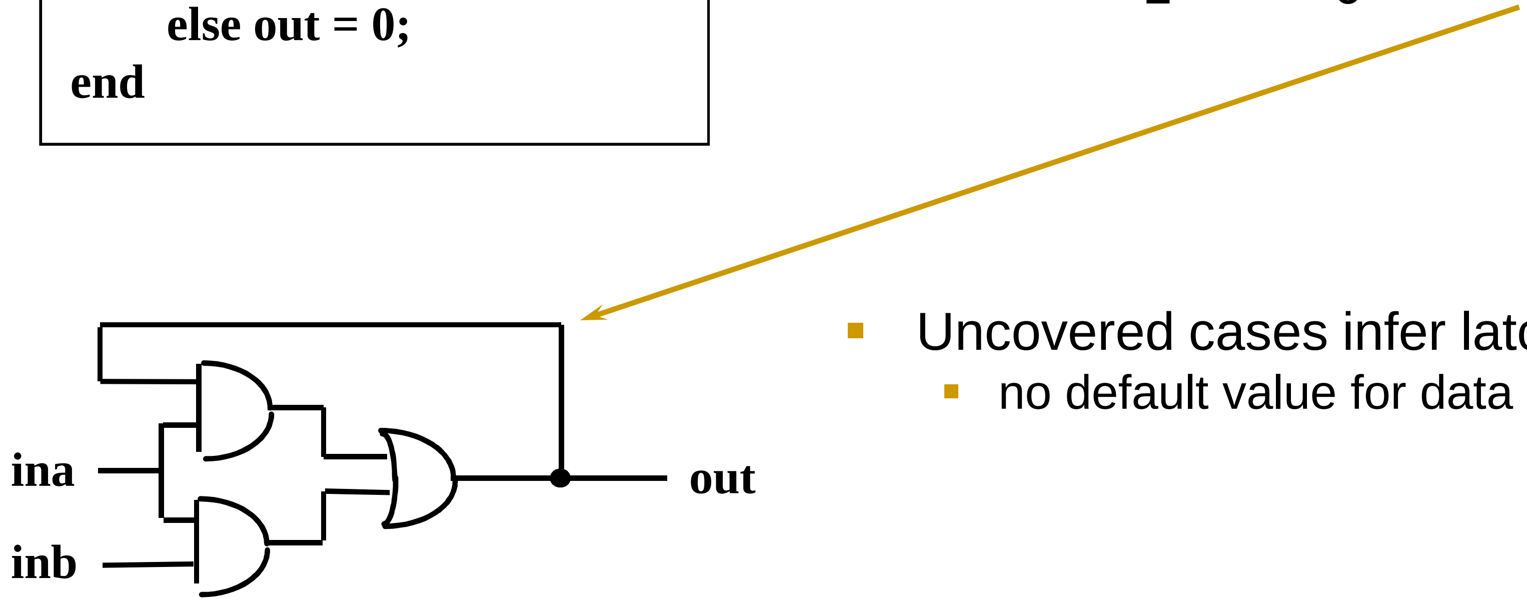
Individual if statements where logic is exclusive *may* be more efficient

```
reg [4:0] state;
parameters0 = 5’h0, s1 = 5‘h11, s2 = 5‘h12, s3 = 5’h14, s4 = 5‘h18;
always @ (posedge clk or negedge reset_n) begin
        if (reset_n == 0) state <= s0;
        else begin
                if (state == s0) begin
                        if (in == 1) state <= s1;  else state <= s0;
                end
                if (state == s1) state <= s2;
                if (state == s2) state <= s3;
                if (state == s3) state <= s4;
                if (state == s4) begin
                        if (in == 1) state <= s1;  else state <= s0;
                end
        end
end
```

UNWANTED LATCHES – NESTED IF STATEMENTS

```
always @ (ina or inb)
begin
        if (ina == 1) begin
                if (inb == 1) out = 1;
                end
        else out = 0;
end
```

ina	inb	out
1	1	1
0	0	0
0	1	0
1	0	?

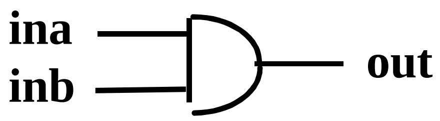


- Uncovered cases infer latches
  - no default value for data type objects

UNWANTED LATCHES REMOVED – NESTED IF STATEMENTS

```
always @ (ina or inb)
begin
        if (ina == 1)
                begin
                if (inb == 1) out = 1;
                else out = 0;
                end
        else out = 0;
end
```

ina	inb	out
1	1	1
0	0	0
0	1	0
1	0	0



- All cases are covered; no latch inferred

INEFFICIENT CASE STATEMENT

```
reg [4:0] state;
parameter   s0 = 5'b00000,   s1 = 5'b10001,
            s2 = 5'b10010,   s3 = 5'b10100,
            s4 = 5'b11000;
always @ (posedge clk or negedge reset_n) begin
    if (reset_n == 0) state <= s0;
    else
        case (state)
            s0:        if (in == 1) state <= s1;
                       else state <= s0;
            s1:        state <= s2;
            s2:        state <= s3;
            s3:        state <= s4;
            s4:        if (in == 1) state <= s1;
                       else state <= s0;
endcase
end
```

EFFICIENT CASE STATEMENT

```
reg [4:0] state;
parameter     s0 = 5'b00000,          s1 = 5'b10001,
              s2 = 5'b10010,          s3 = 5'b10100,
              s4 = 5'b11000;
always @ (posedge clk or negedge reset_n) begin
  if (reset_n == 0) state <= s0;
  else
    case (state)
      s0:         if (in == 1) state <= s1;
                  else state <= s0;
      s1:         state <= s2;
      s2:         state <= s3;
      s3:         state <= s4;
      s4:         if (in == 1) state <= s1;
                  else state <= s0;
      default:    state <= s0;
    endcase
end
```

All 32 cases covered

A case structure where all cases are covered will typically synthesize most efficiently

“FULL” CASE

All possible case-expression binary patterns can be matched to a case item or a case default

Non-full case statements infer latches

Adding “full_case” synthesis directive to non-full case statements may cause pre-synthesis and post-synthesis simulation mismatch

Mismatched Code

```
always @ (ena or a) begin
        out = 2’b00;
        case ({ena,a}) //synthesis full_case
                2’b1_0:  out[a] = 1’b1;
                2’b1_1:  out[a] = 1’b1;
        endcase
        end
```

Full Statement

```
always @ (ena or a) begin
        out = 2’b00;
        case ({ena,a})
                2’b1_0:  out[a] = 1’b1;
                2’b1_1:  out[a] = 1’b1;
                default:  out = 2’b00;
        endcase
        end
```

“PARALLEL” CASE

Only possible to match a case expression to one and only one case item

Non-parallel case statements infer priority encoders

Adding “parallel_case” synthesis directive to non-parallel case statement may cause mismatch between Verilog functional model and synthesized logic

“FULL CASE” & “PARALLEL CASE” DIRECTIVES

Don’t use!

- Code synthesizable case statements to be “full” and “parallel” when designing

Great paper discusses the perils of CASE synthesis directives

- http://www.sunburst-design.com/papers/CummingsSNUG1999Boston_FullParallelCase.pdf

COMBINATIONAL LOOPS

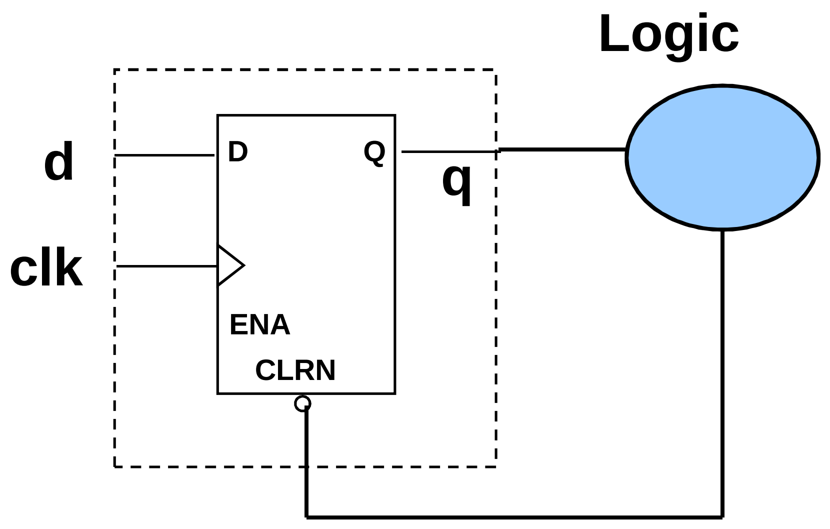
Common cause of instability

Behavior of loop depends on the relative propagation delays through logic

- Propagation delays can change

Simulation tools may not match hardware behavior

All feedback loops should include registers



INTERNALLY GENERATED CLOCKS

Can lead to both functional and timing problems

If you must have internally generated clocks, proper design considerations required

- Keep the clock generation circuitry as a separate module at top level of the design
- Always register the output of combinational logic before using it as a clock

<u>Recommendation:</u> Use FPGA PLL megafunction for gated clocks

USING FUNCTIONS & TASKS

Functions and tasks can be used in synthesized modules

- Variables in functions must be local

Make generic if possible for reusability

```
// function for ...

function [‘BUS_WIDTH-1:0] out;
input a;
integer a;
  begin
  // ... Code for function goes here
  end
endmodule
```

INFERRING LOGIC FUNCTIONS

Using behavioral modeling to describe logic blocks

- e.g. Latches, registers, counters, tri-states, memory, arithmetic

Synthesis tools recognize description & insert logic functions

- Functions typically pre-optimized for utilization or performance

Makes code vendor-independent

BIDIRECTIONAL PINS

```
module module_name (..., bidi, ...);

inout bidi;
wire incoming_signal;

assign incoming_signal = bidi;

logic operating on incoming_signal
logic generating outgoing_signal

assign bidi = oe ? outgoing_signal : 1'bZ;
```

- *Declare Pin As Direction* ***inout***
- *Use* ***inout*** *As Both Input & Tri-State Output*

bidir as an input

bidir as an tri-stated output

INFERRING LATCHES

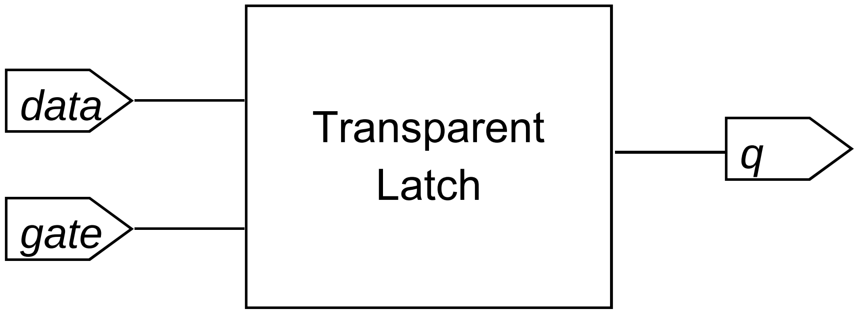
```
module latches (d, gate,q);

input d, gate ;
output reg q ;

wire d, gate ;

always @(d or gate)

       if (gate)
     q <= d ;
endmodule
```

sensitivity list includes both inputs

level sensitive…not edge

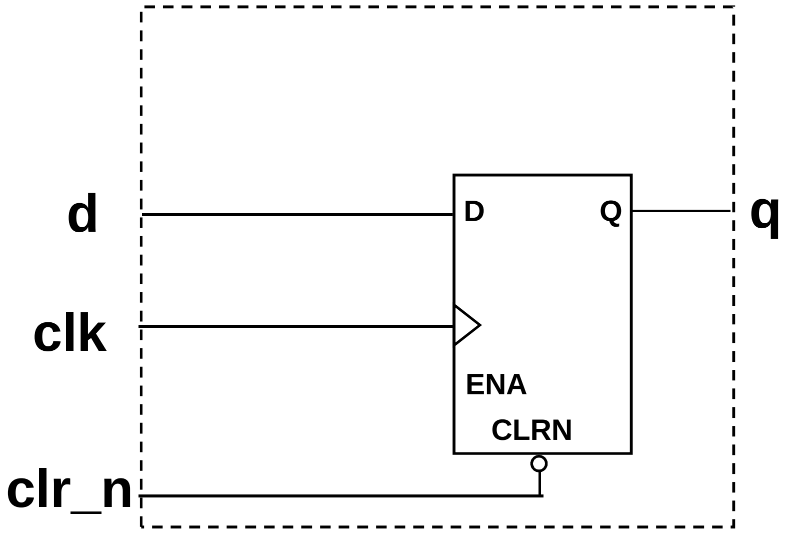
What happens if gate = ‘0’?

⇨ ***Implicit Memory & Feedback***



INFERRING FLIP-FLOPS

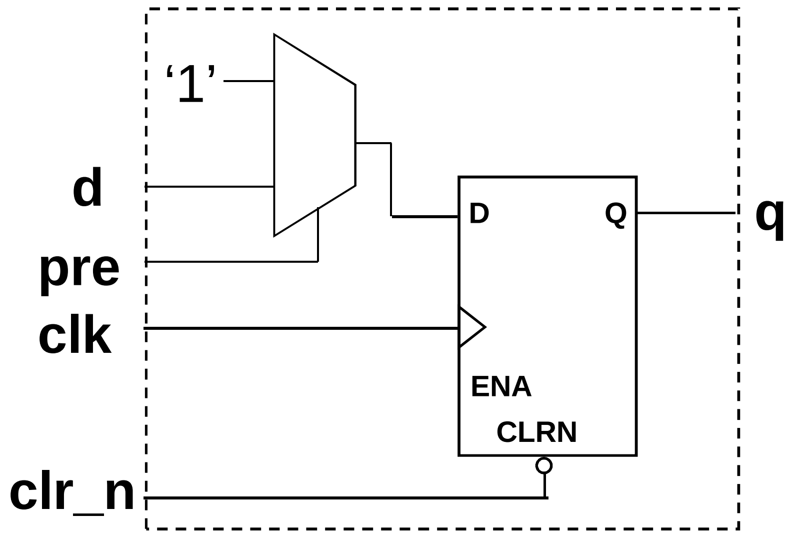
```
module dff_1(clk, d, clr_n, pre, q);
input clk, d, clr_n, pre;
output reg q;

always @(posedge clk or negedge clr_n)
    if (clr_n == 0)
        q <= 0;
    else
        q <= d;
endmodule
```



– *Simple register logic*

INFERRING FLIP-FLOPS

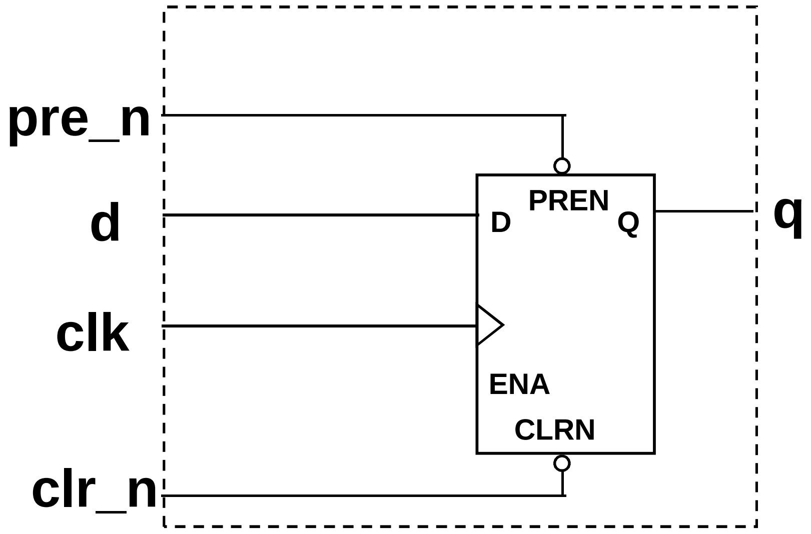
```
module dff_1(clk, d, clr_n, pre, q);
input clk, d, clr_n, pre;
output reg q;

always @(posedge clk or negedge clr_n)
     if (clr_n == 0)
          q <= 0;
     else
          if (pre == 1)
               q <= 1;
           else
               q <= d;
endmodule
```



- *This implements asynchronous clear and synchronous preset control signals for the register*
- *Use Non-blocking Assignment*
- *<u>Synchronous controls are not included in sensitivity list</u>*

DFF WITH ASYNC CLEAR & PRESET

```
module dff_1(clk, d, clr_n, pre_n, q);
input clk, d, clr_n, pre_n;
output reg q;

always @(posedge clk or negedge clr_n
or negedge pre_n)
    if (clr_n == 0)
        q <= 0;
    else if (pre_n == 0)
        q <= 1;
    else
        q <= d;
endmodule
```



- *2 Asynchronous Controls*
- *Do the Registers in the Hardware Have Both Ports?*
- *How Does Hardware Behave? Does Preset or Clear Have Priority If Both Asserted?*

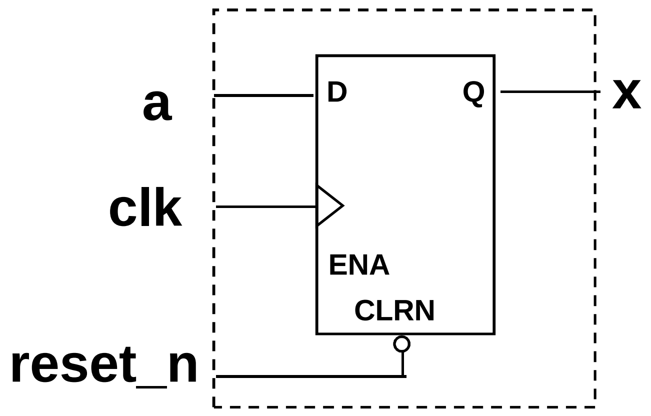
CONTROL SIGNAL PRIORITY

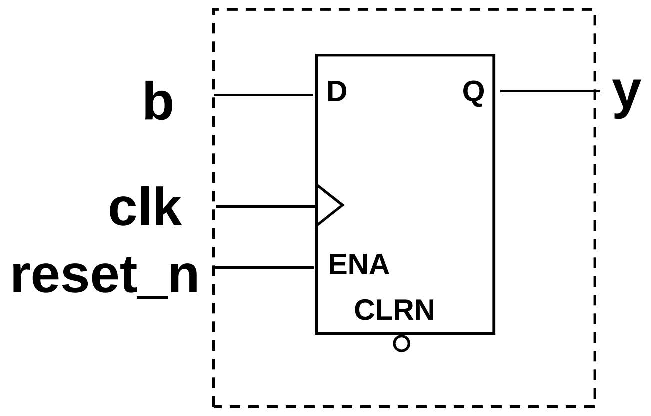
1. **Asynchronous clear (aclr)**
2. **Asynchronous preset (pre)**
3. **Asynchronous load (aload)**
4. **Enable (ena)**
5. **Synchronous clear (sclr)**
6. **Synchronous load (sload)**

INCORRECT CONTROL LOGIC

```
module dff_1(clk, reset_n, a, b, x, y);
input clk, reset_n, a, b;
output reg x, y;

always @(posedge clk or negedge reset_n)
        if (reset_n == 0)
                x <= 0;
        else begin
                x <= a;
                y <= b;
        end
endmodule
```

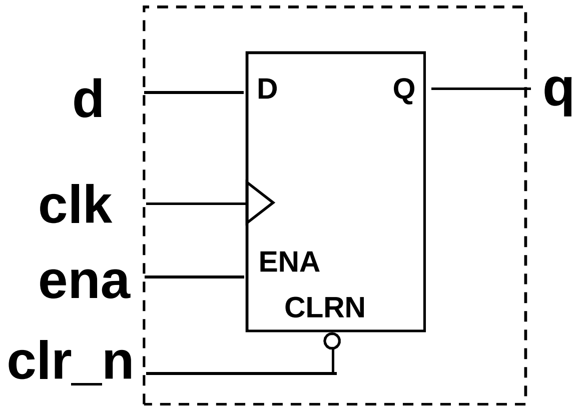
- ***y*** *Is Not Included in Reset Condition*
- ***reset*** *Clears* ***x*** *but Acts More Like an Enable for* ***y***





DFF WITH CLOCK ENABLE

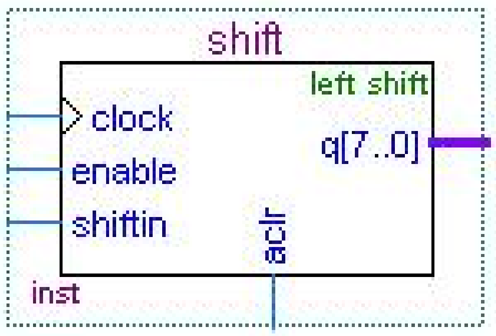
```
module dff_1(clk, d, q, ena, clr_n);
input clk, d, ena, clr_n;
output reg q;

always @(posedge clk or negedge clr_n)
    if (clr_n == 0)
        q <= 0;
    else if (ena == 1)
        q <= d;
endmodule
```



- *If the synthesis tool does not recognize this as an enable it will be implemented using extra LUTs*
- *Any other synchronous controls (e.g. sclr, sload) are nested underneath the enable test (i.e. use “if,” not “else if”)*

SHIFT REGISTERS

```
module shift(aclr_n, enable, shiftin, clock, q);
input aclr_n, enable, shiftin, clock;
output reg [7:0] q;

always @(posedge clock or negedge aclr_n)
    if (aclr_n == 0)
        q[7:0] <= 0;
    else if (enable)
        q[7:0] <= {q[6:0], shiftin};
endmodule
```

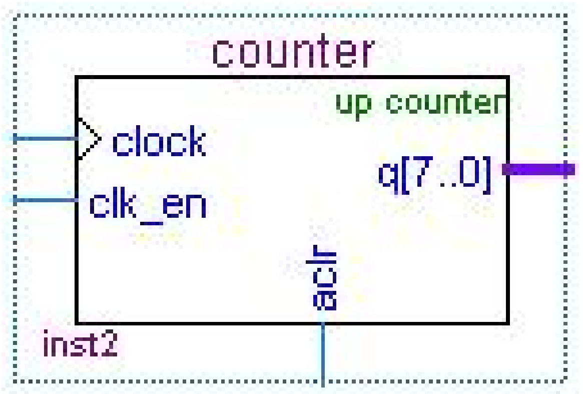


Shift function
•*Use { , } for concatenation*

- *Shift register with parallel output, serial input, asynchronous clear and enable which shifts left*
- *Add or remove synchronous controls in a similar manor to previous slides*

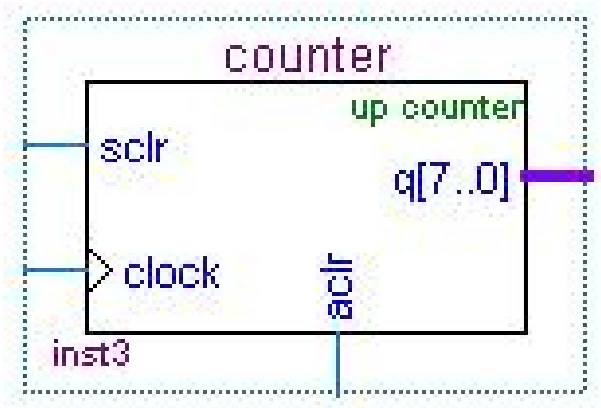
COUNTER W/CLOCK ENABLE

```
module count(clk, ena, aclr_n, q);
input clk, aclr_n, ena;
output reg [7:0] q = 0;
always @(posedge clk or negedge aclr_n)
        if (aclr_n == 0)
                q[7:0] <= 0;
        else if (ena)
                q <= q + 1;
endmodule
```



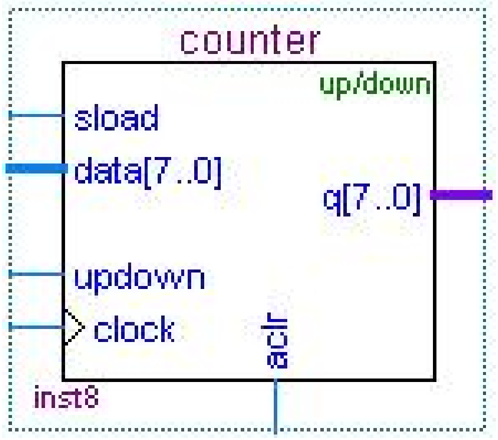
COUNTER W/SYNCH AND ASYNCH CLEAR

```
module count(clk, sclr_n, aclr, q);
input clk, aclr, sclr_n;
output reg [7:0] q = 0;
always @(posedge clk or negedge aclr_n)
      if (aclr_n == 0)
            q[7:0] <= 0;
      else if (sclr_n)
               q[7:0] <= 0 ;
      else
               q <= q + 1;
endmodule
```



UP/DOWN COUNTER W/SYNC LOAD

```
module count(clk, updown, aclr_n, sload, data, q);
input clk, aclr_n, updown, sload;
input [7:0] data;
output reg [7:0] q = 0;

always @(posedge clk or negedge aclr_n) begin
        if (aclr_n == 0)
                q[7:0] <= 0;
        else if (sload == 1)
                q <= data;
        else begin
                q <= q + (updown ? 1 : -1);
           end
end
endmodule
```



MEMORY

Synthesis tools have different capabilities for recognizing memories

Synthesis tools are sensitive to certain coding styles in order to recognize memories

- Usually described in the tool documentation

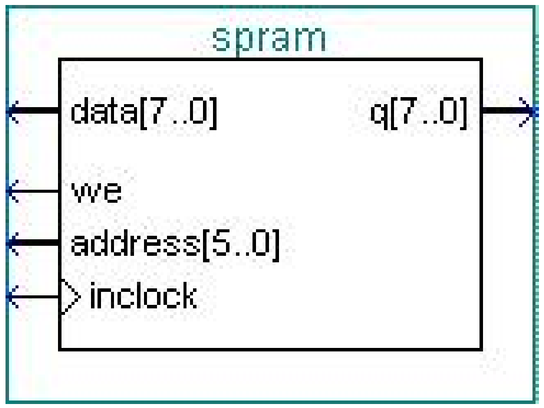
Tools may have limitations in architecture implementation

- Newer FPGA devices support synchronous writes only
- Limitations in clocking schemes

Must declare an array data type to hold memory values

INFERRED SINGLE-PORT MEMORY (1)

```
module ram(q, a, d, we, clk);

output reg [7:0] q;
input [7:0] d;
input [6:0] a;
input we, clk;

reg [6:0] read_add;
reg [7:0] mem [127:0];

always @(posedge clk) begin
  if (we)
    mem[a] <= d;
  read_add <= a;
end
assign q = mem[read_add];
endmodule
```



– *Code describes a **128 x 8 RAM** with <u>synchronous write</u> & <u>asynchronous read</u>*

INFERRED SINGLE-PORT MEMORY (2)

```
module ram(q, a, d, we, clk);
output reg [7:0] q;
input [7:0] d;
input [6:0] a;
input we, clk;

reg [6:0] read_add;
reg [7:0] mem [127:0];

always @(posedge clk) begin
  if (we)
    mem[a] <= d;
    read_add <= a;
    q <= mem[read_add];

end

endmodule
```

– *Code describes a* ***128 x 8 RAM*** *with* *<u>synchronous write</u>* *&* *<u>synchronous read</u>*

INSTANTIATING LOGIC FUNCTIONS

Using structural modeling to build logic

Synthesis tools replace structure with pre-built logic functions

Makes code insensitive to coding style

MOORE STATE MACHINES

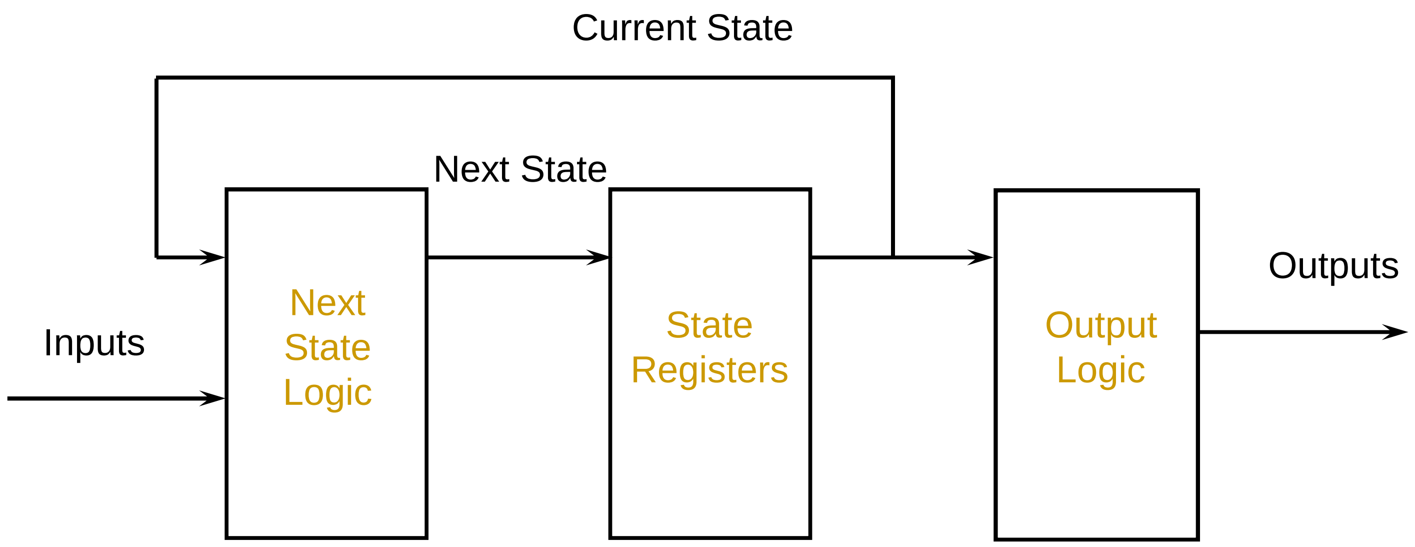
Outputs Are a Function of the States Only

Settle to their Final Values a Few Gate Delays After the Active Clock Edge

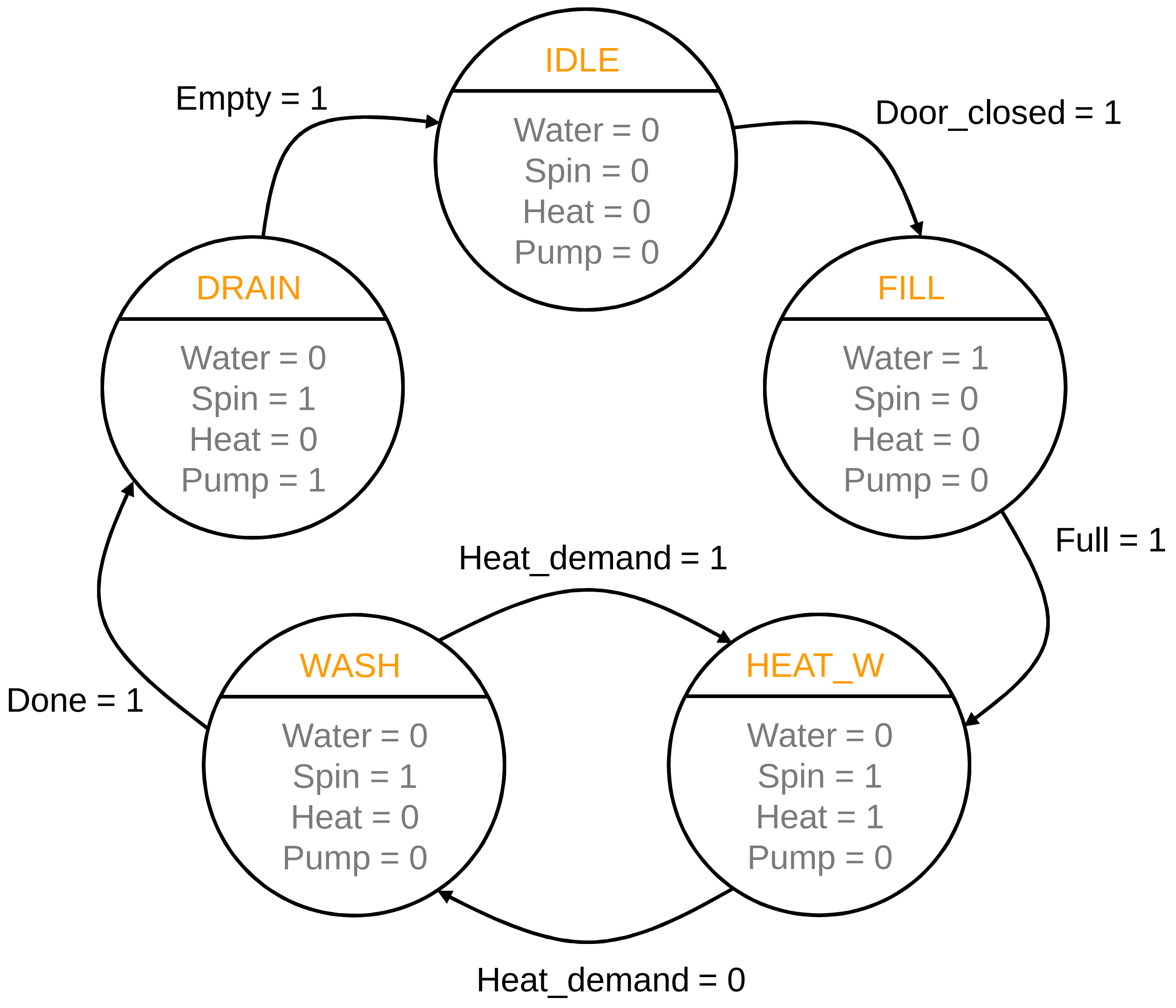
Outputs will be Constant for the Remainder of the Clock Period, Even if Inputs Happen to Change During the Clock Period

Isolates the Outputs from the Inputs

MOORE – BLOCK DIAGRAM



MOORE – STATE DIAGRAM



<u>INPUTS</u>

Door_closed

Full

Heat_demand

Done

Empty

<u>OUTPUTS</u>

Water

Spin

Heat

Pump

MEALY STATE MACHINES

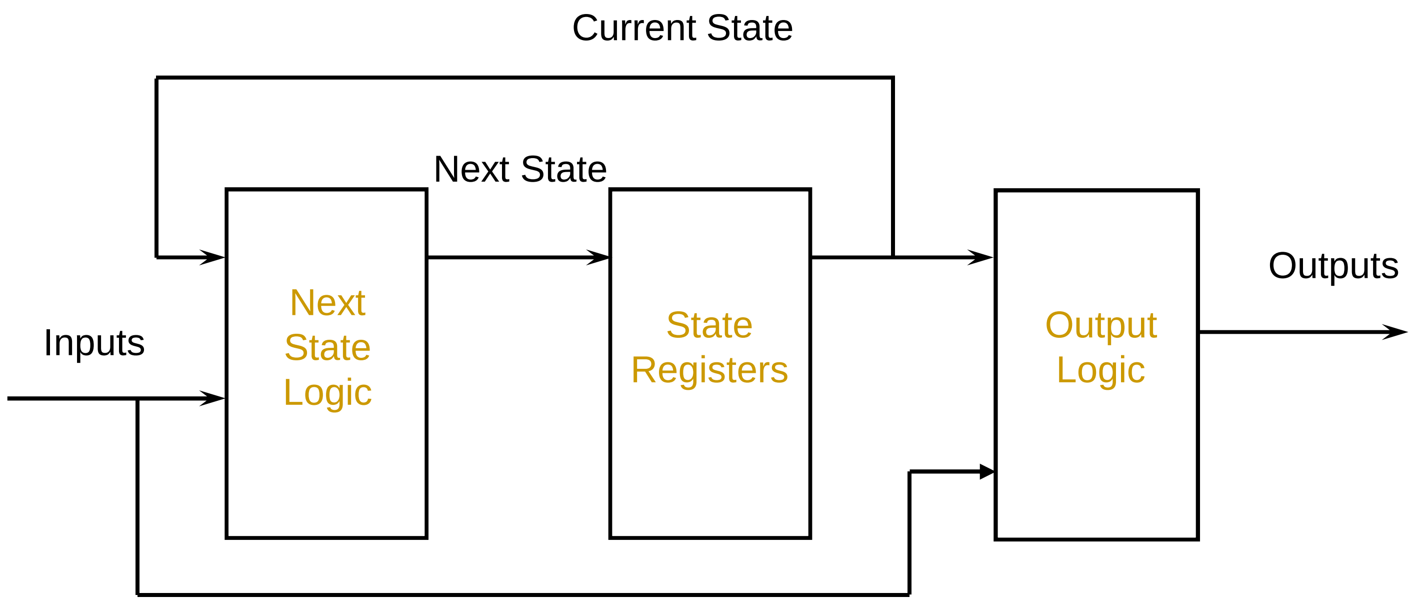
Outputs are a Function of the Inputs as well as the States

An Output May Change in the Middle of a Clock Period if an Input Changes

Output May Not be Consistent Throughout a Clock Cycle

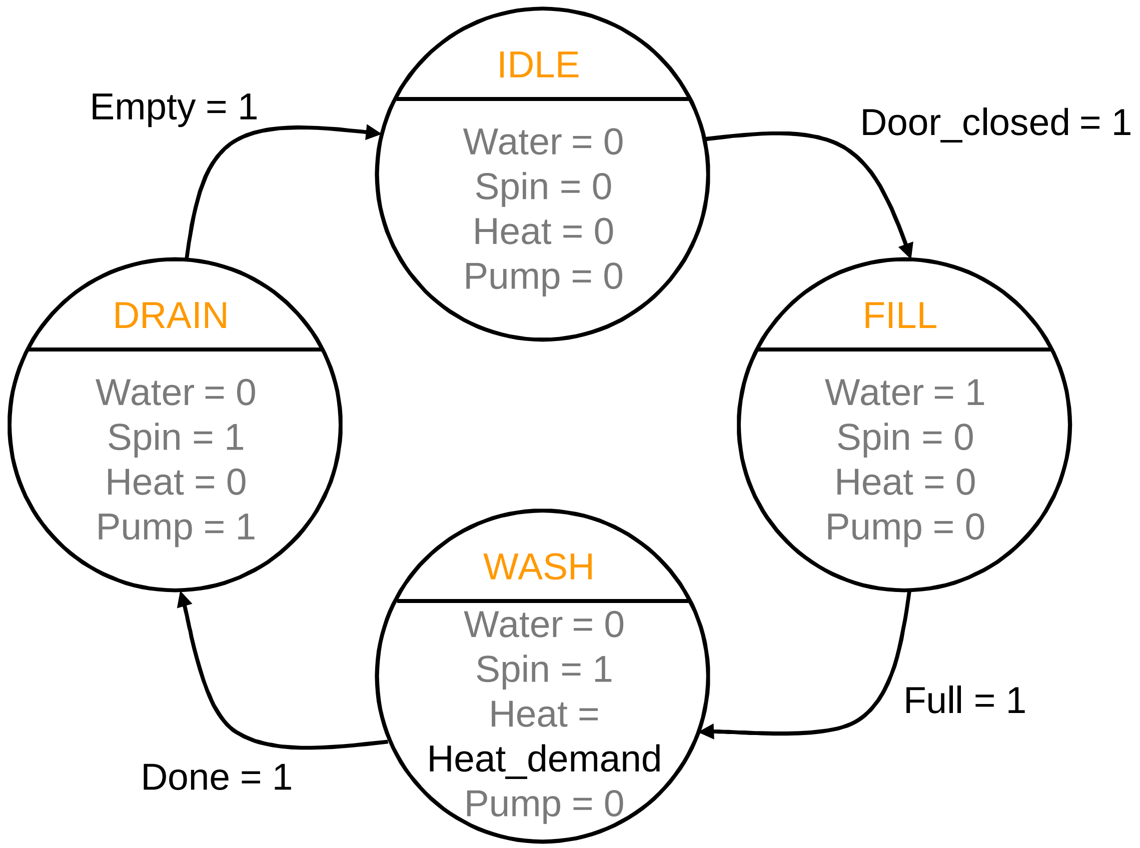
However, Allows the Outputs to Follow Spurious Input Changes

MEALY – BLOCK DIAGRAM



MEALY – STATE DIAGRAM

Mealy Often Leads to Fewer States



INPUTS
- Door_closed
- Full
- Heat_demand
- Done
- Empty

OUTPUTS
- Water
- Spin
- Heat
- Pump

STATE MACHINE CODING

Parameters used to define states

parameter idle=0, fill=1, heat_w=2, wash=3, drain=4;

- Parameter “values” are replaced with state encoding values as chosen by synthesis tool
  - Use options/constraints in synthesis tool to control encoding style (e.g. binary, one-hot, safe, etc.)
- ‘define statements also supported

Registers are used to store state

reg [2:0] state, next_state;

Use CASE statement to do the next-state logic, instead of IF-THEN statement

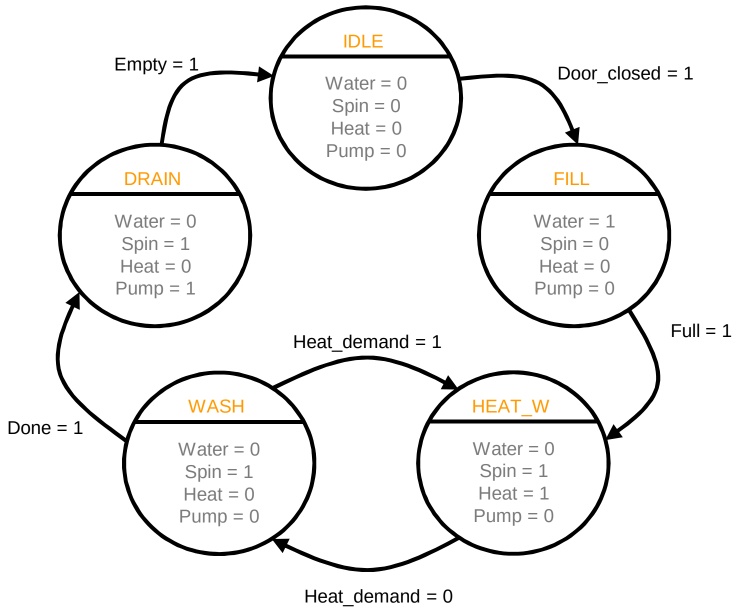
- Some synthesis tools recognize CASE statements for implementing state machines

STATE DECLARATION

```
module state_machine (clk, reset, door_closed,
        full, heat_demand,  done, empty,
        water, spin, heat, pump);
input clk, reset, door_closed, full, heat_demand,
        done, empty;
output reg water, spin, heat, pump;

reg [2:0] state, next_state;

parameter idle=0, fill=1, heat_w=2, wash=3,
        drain=4;
```

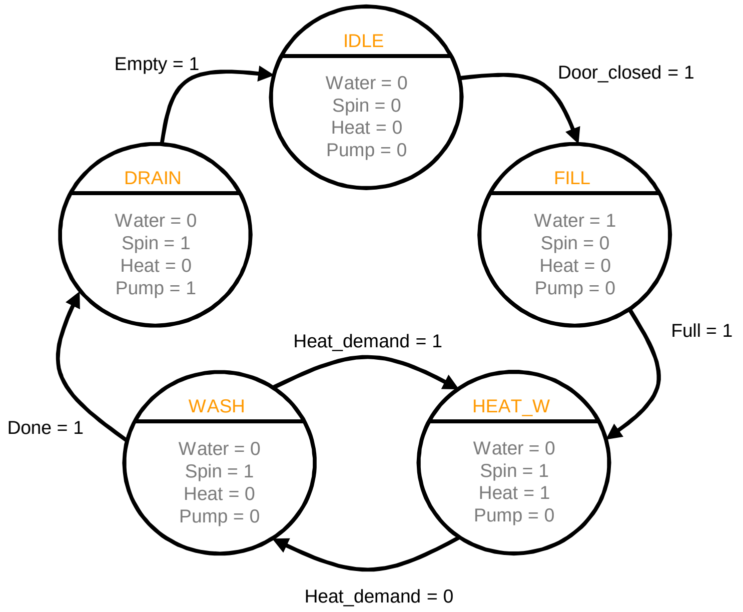
State Registers

States



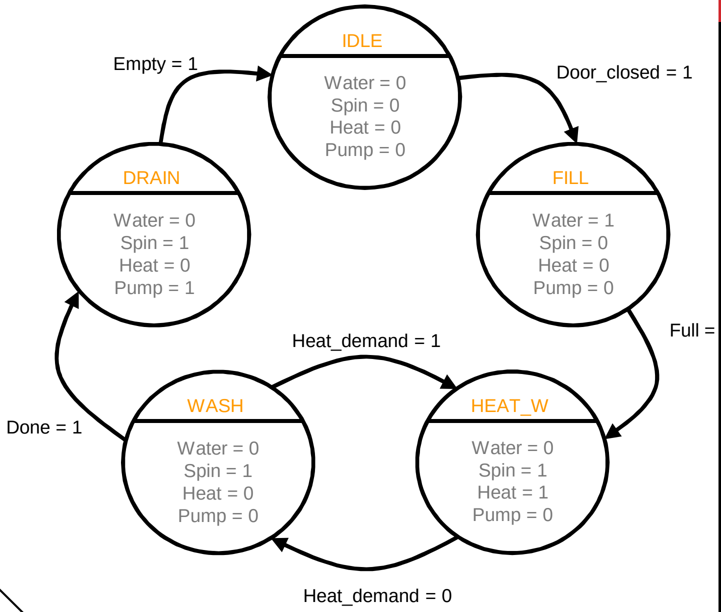
NEXT STATE LOGIC

```
//Next State logic
always @(state or reset or door_closed or full or
    heat_demand or done or empty)
begin
    next_state = state;   //default condition
    case (state)
        idle: if (door_closed) next_state = fill;
        fill: if (full) next_state = heat_w;
        heat_w: if (heat_demand == 0) next_state = wash;
        wash: begin
            if (heat_demand) next_state = heat_w;
            if (done) next_state = drain;
            end
        drain: if (empty) next_state = idle;
        default: next_state = idle;
    endcase
end
```



USING TWO PROCESSES

```
//State transitions
always @(posedge clk)
    state <= next_state;

//Next State logic
always @(state or reset or door_closed or full or
    heat_demand or done or empty)
begin
    next_state = state;   //default condition
    case (state)
        idle: if (door_closed) next_state = fill;
        fill: if (full) next_state = heat_w;
        heat_w: if (heat_demand==0) next_state = wash;
        wash: begin
            if (heat_demand) next_state = heat_w;
            if (done) next_state = drain;
            end
        drain: if (empty) next_state = idle;
        default: next_state = idle;
    endcase
end
```



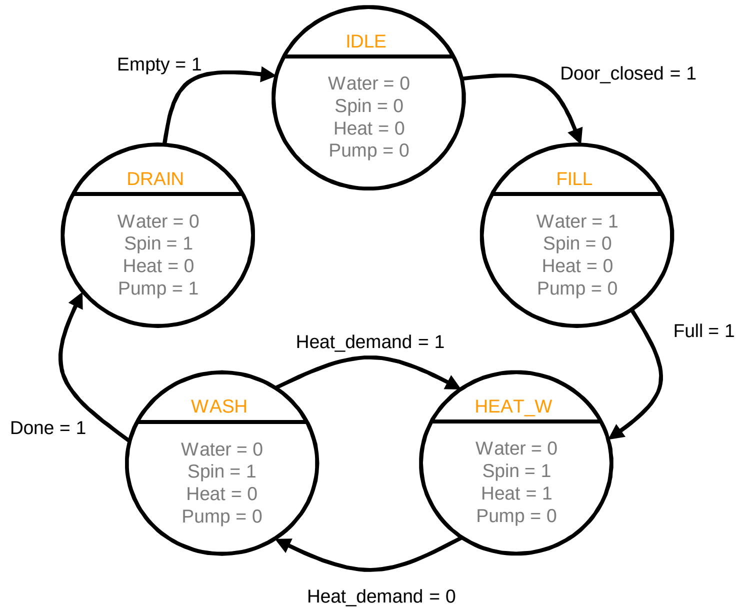
State transitions

Next state logic

SYNCHRONOUS RESET

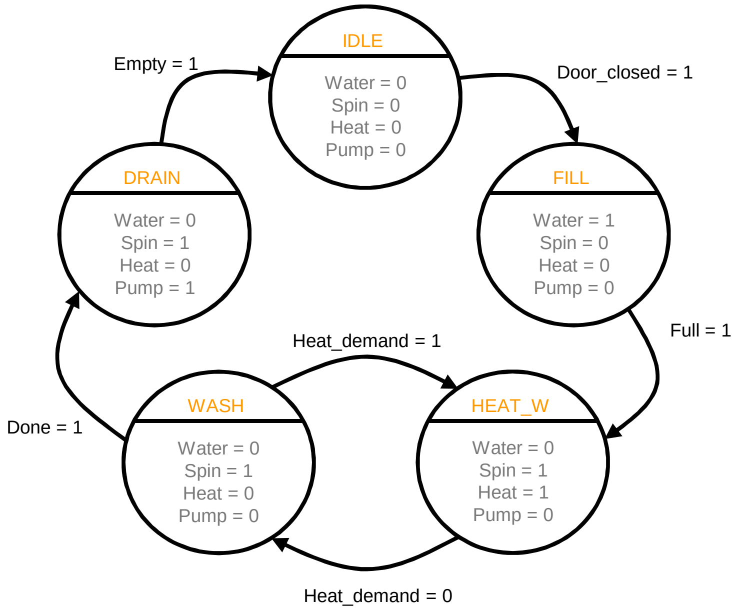
Synchronous reset

```
//State transitions
always @(posedge clk)
    if (reset)
        state <= idle;
    else
        state <= next_state;

//Next State logic
always @(state or reset or door_closed or full or
    heat_demand or done or empty)
begin
    next_state = state;   //default condition
    case (state)
        idle: if (door_closed) next_state = fill;
        fill: if (full) next_state = heat_w;
        heat_w: if (heat_demand==0) next_state = wash;
        wash: begin
            if (heat_demand) next_state = heat_w;
            if (done) next_state = drain;
            end
        drain: if (empty) next_state = idle;
        default: next_state = idle;
    endcase
end
```



COMBINATORIAL OUTPUTS

```
//output logic
always @(state)
begin
    water = 0;
    spin = 0;
    heat = 0;
    pump = 0;
    case (state)
        idle: ;
        fill: water=1;
        heat_w: begin  spin =1; heat=1; end
        wash: spin =1;
        drain: begin spin =1; pump=1; end
    endcase
end
```

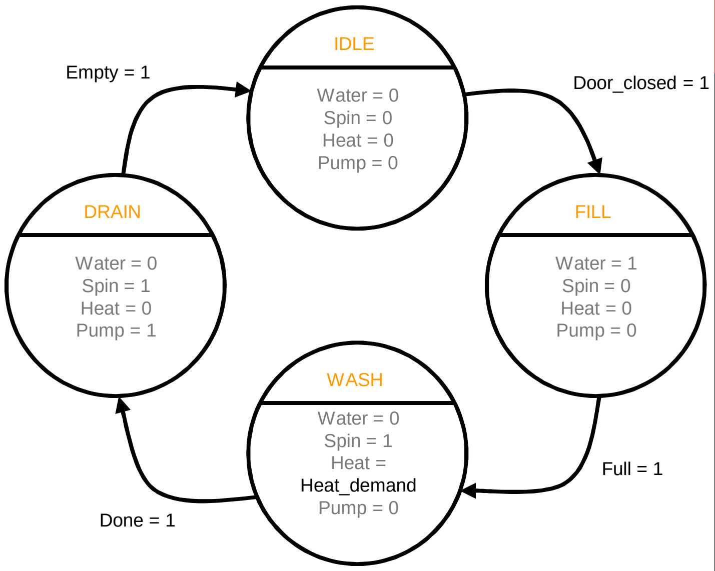


Default output conditions

MEALY COMBINATORIAL OUTPUTS

```
//output logic
always @(state)
begin
    water=0;
    spin=0;
    heat=0;
    pump=0;
    case (state)
        idle: ;
        fill: water=1;
        wash: begin  spin =1; heat = heat_demand; end
        drain: begin spin =1; pump=1; end
    endcase
end
```

Mealy Output

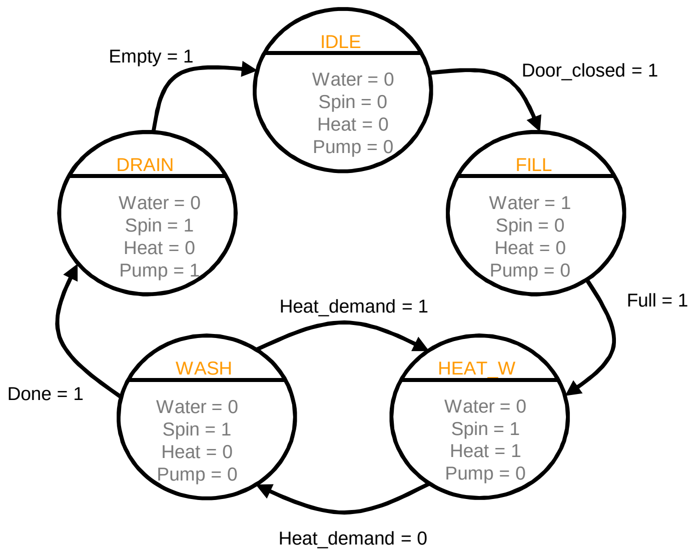


STATE MACHINE ENCODING STYLES

State	Binary Encoding	Grey-Code Encoding	One-Hot Encoding	Custom Encoding
Idle	000	000	00001	?
Fill	001	001	00010	?
Heat_w	010	011	00100	?
Wash	011	010	01000	?
Drain	100	110	10000	?

MANUAL STATE ENCODING

```
module state_machine (clk, reset, door_closed,
        full, heat_demand,  done, empty, water, spin, heat,
        pump);
input clk, reset, door_closed, full, heat_demand,
        done, empty;
output reg water, spin, heat, pump;

reg [2:0] state, next_state;
parameter idle = 3'b001, fill = 3'b011, heat_w = 3'b010,
        wash =3'b110, drain =3'b100;
```



Grey-Code Encoding

<u>Note</u>: Also requires setting STATE MACHINE ENCODING assignment to User-Encoded

UNDEFINED STATES

Noise and spurious events in hardware can cause state machines to enter undefined states

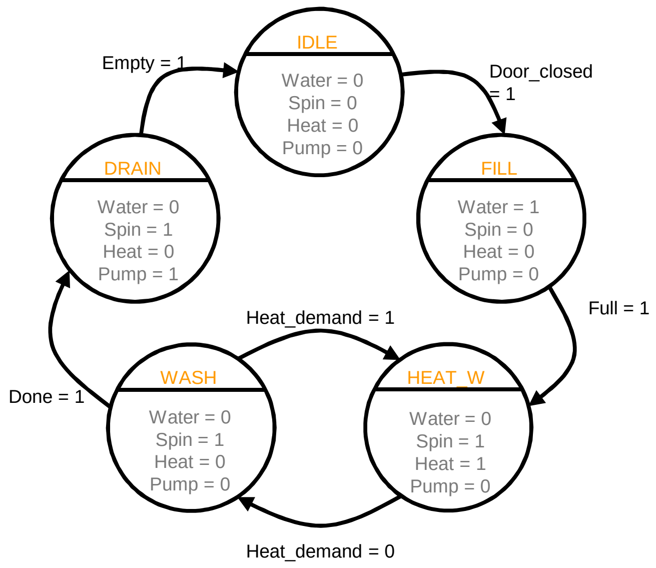
If state machines do not consider undefined states, it can cause mysterious “lock-ups” in hardware

Good engineering practice is to consider these states

Synthesis tools may require explicit coding to account for undefined states

‘SAFE’ STATE MACHINE ??

```
//State transitions
always @(posedge clk)
    state <= next_state;

//Next State logic
always @(state or reset or door_closed or full or
        heat_demand or done or empty)
begin
    next_state = state;   //default condition
    case (state)
        idle: if (door_closed) next_state = fill;
        fill: if (full) next_state = heat_w;
        heat_w: if (heat_demand==0) next_state= wash;
        wash: begin
            if (heat_demand) next_state=heat_w;
            if (done) next_state=drain;
            end
        drain: if (empty) next_state=idle;
        default: next_state = idle;
    endcase
end
```

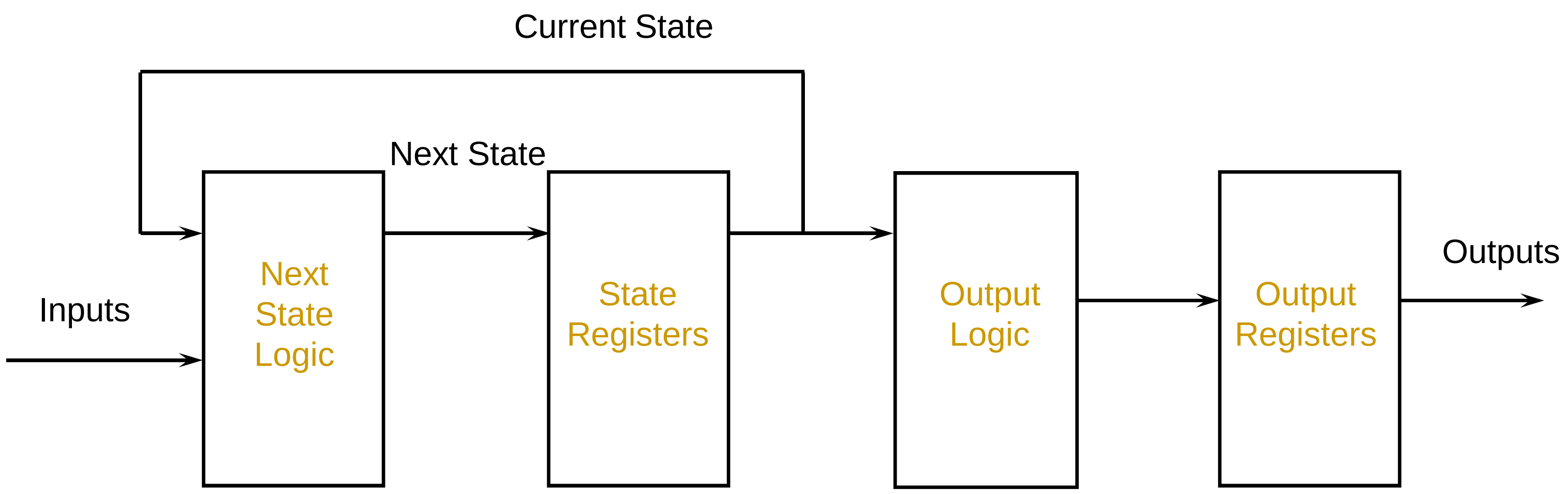


– *Using default clause to cover all undeclared states*

REGISTERED OUTPUTS

Remove glitches by adding output registers

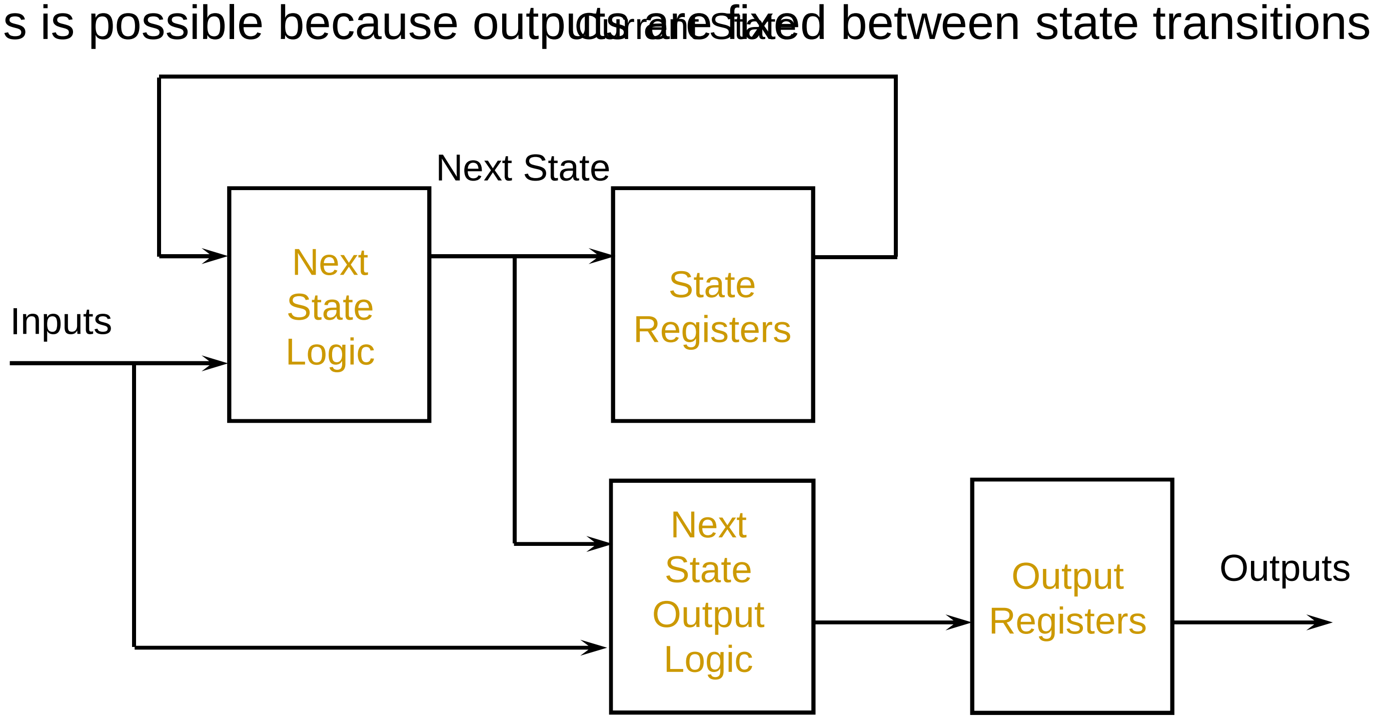
- Adds a stage of latency



REGISTERED OUTPUTS W/O LATENCY

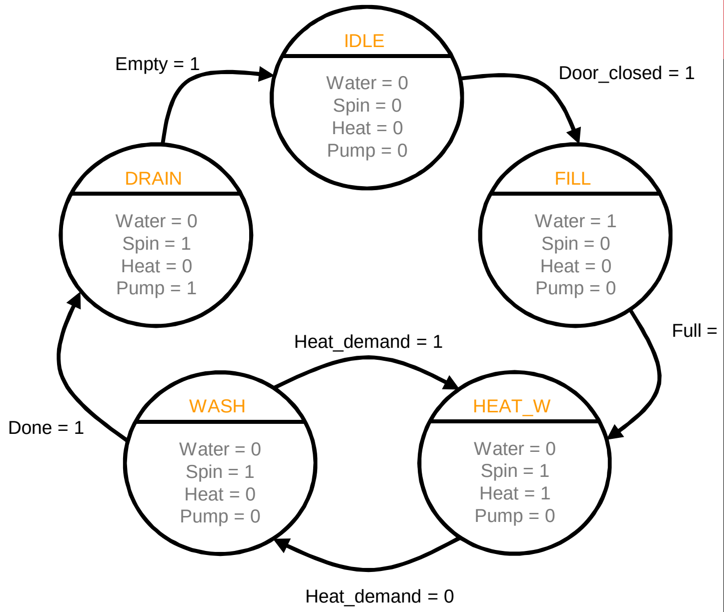
With a Mealy machine next state outputs are a function of current state and inputs

- This is possible because outputs are fixed between state transitions



REGISTERED OUTPUTS

```
//output logic
always @(posedge clk)
begin
    water = 0;
    spin = 0;
    heat = 0;
    pump = 0;
    case (next_state)
        idle: ;
        fill: water <= 1;
        heat_w: begin  spin <= 1; heat <= 1; end
        wash: spin <= 1;
        drain: begin spin <= 1; pump <= 1; end
    endcase
end
```



USING CUSTOM ENCODING STYLES

Remove glitches without output registers

Eliminate combinatorial output logic

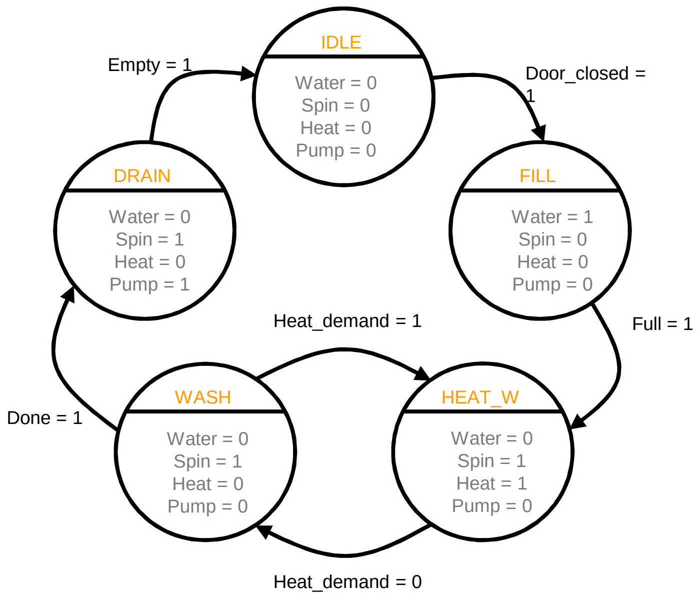
Outputs mimic state bits

- Use additional state bits for states that do have exclusive outputs

State	Outputs				Custom Encoding
	Water	Spin	Heat	Pump	
Idle	0	0	0	0	0000
Fill	1	0	0	0	1000
Heat_w	0	1	1	0	0110
Wash	0	1	0	0	0100
Drain	0	1	0	1	0101

CUSTOM STATE ENCODING

```
reg [3:0] state, next_state;
parameter idle = ’b0000, fill = ’b1000,
        heat_w = ’b0110, wash = ’b0100,
        drain = ’b0101;



//output logic
always @(state)
begin
        water <= state[3];
        spin <= state[2];
        heat <= state[1];
        pump <= state[0];
end
```



Custom Encoding

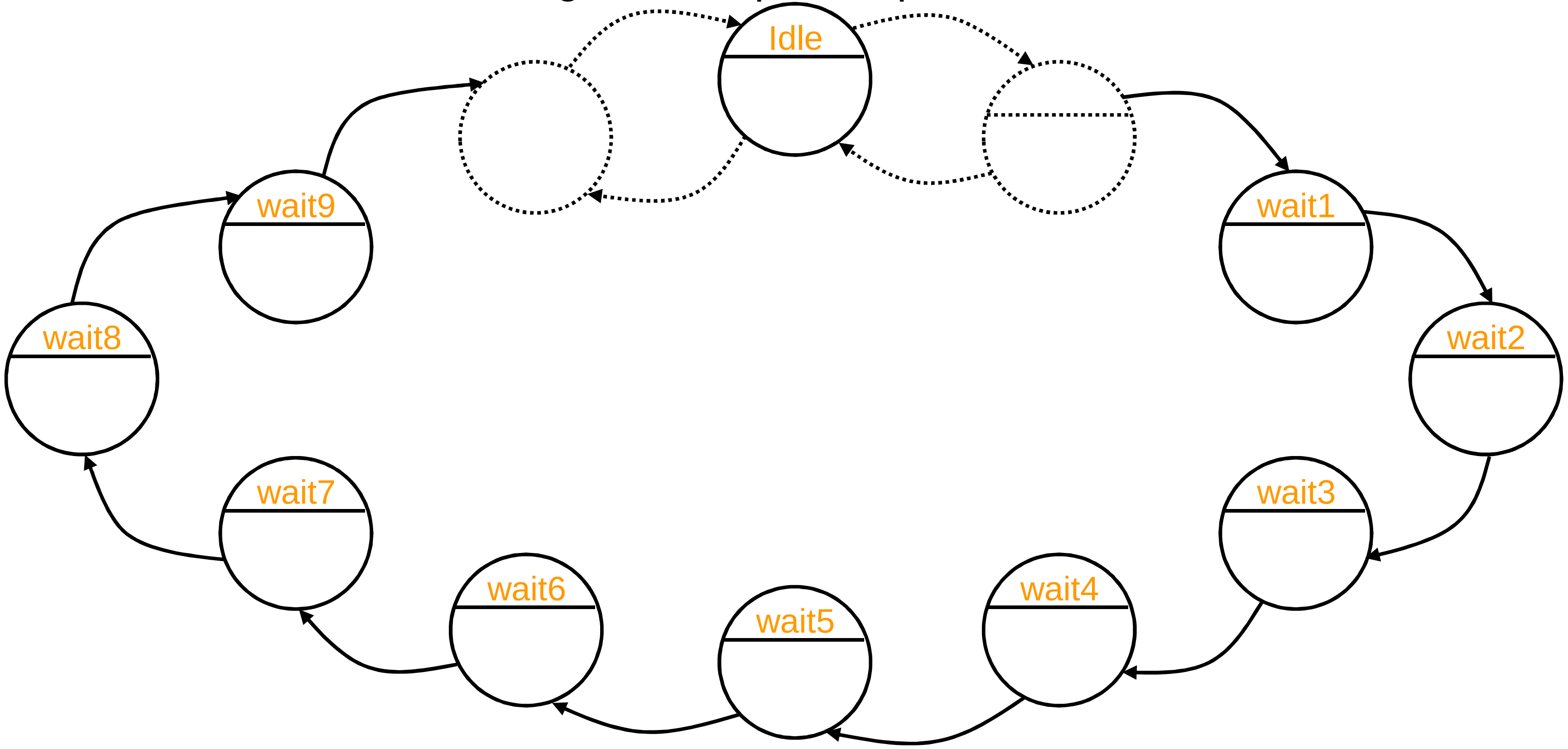
Assign outputs to state bits

<u>Note</u>: Also requires setting STATE MACHINE ENCODING assignment to User-Encoded

EFFICIENT STATE MACHINES

Remove Counting, timing functions from the state machine and implement externally

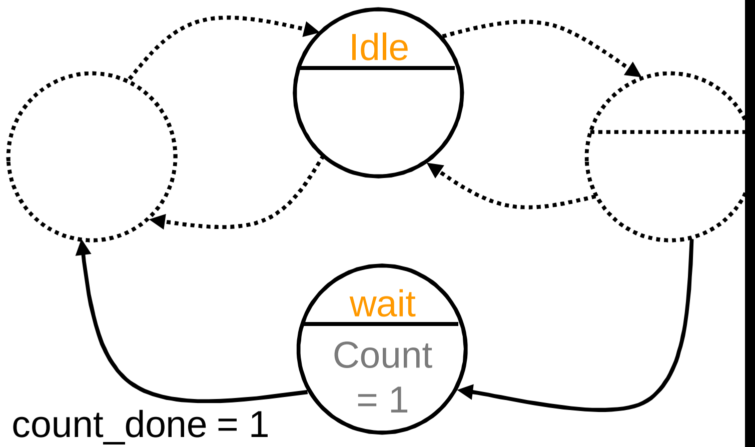
- This reduces overall logic and improves performance

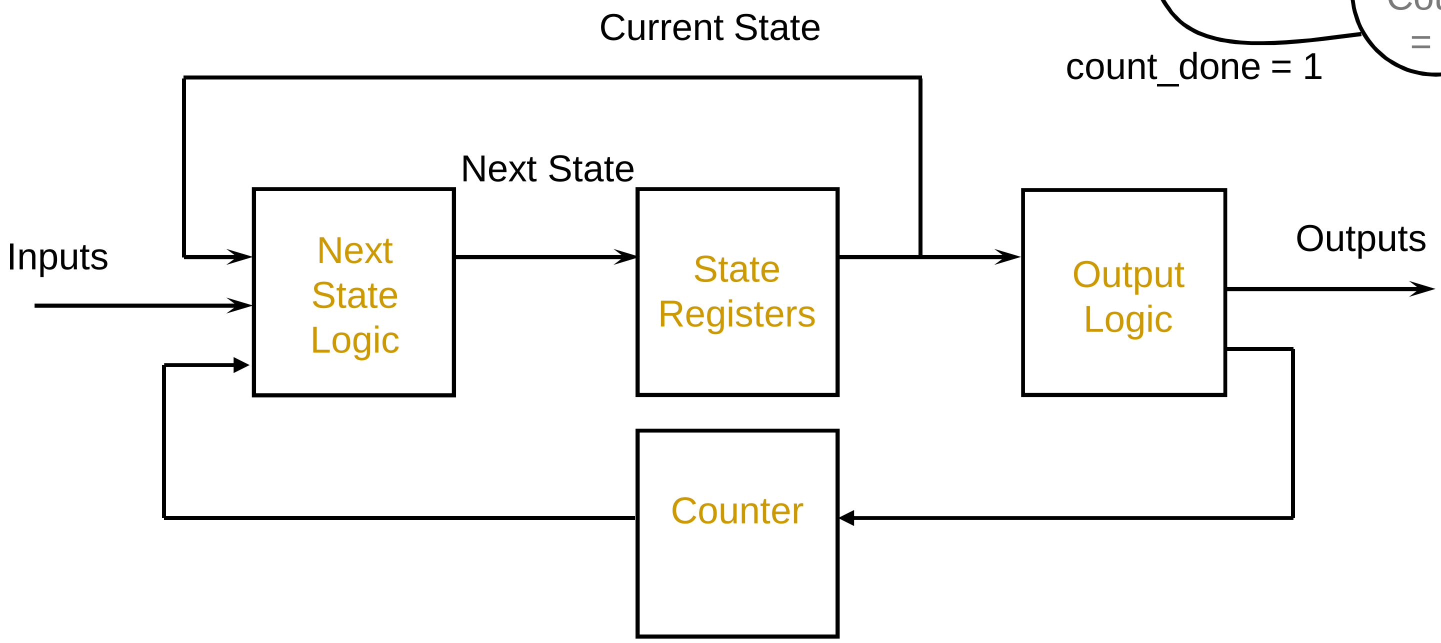


EXTERNAL COUNTER

Add outputs to control counter

- Counter overflow becomes input to state machine





GENERAL DESIGN CONSIDERATIONS

Write code target device in mind

Use synchronous design methods

Avoid combinatorial loops

Avoid gated clocks

LOGIC OPTIMIZATION FOR FPGA

Balancing Operators

Resource Sharing

Logic Duplication

Pipelining

OPERATORS

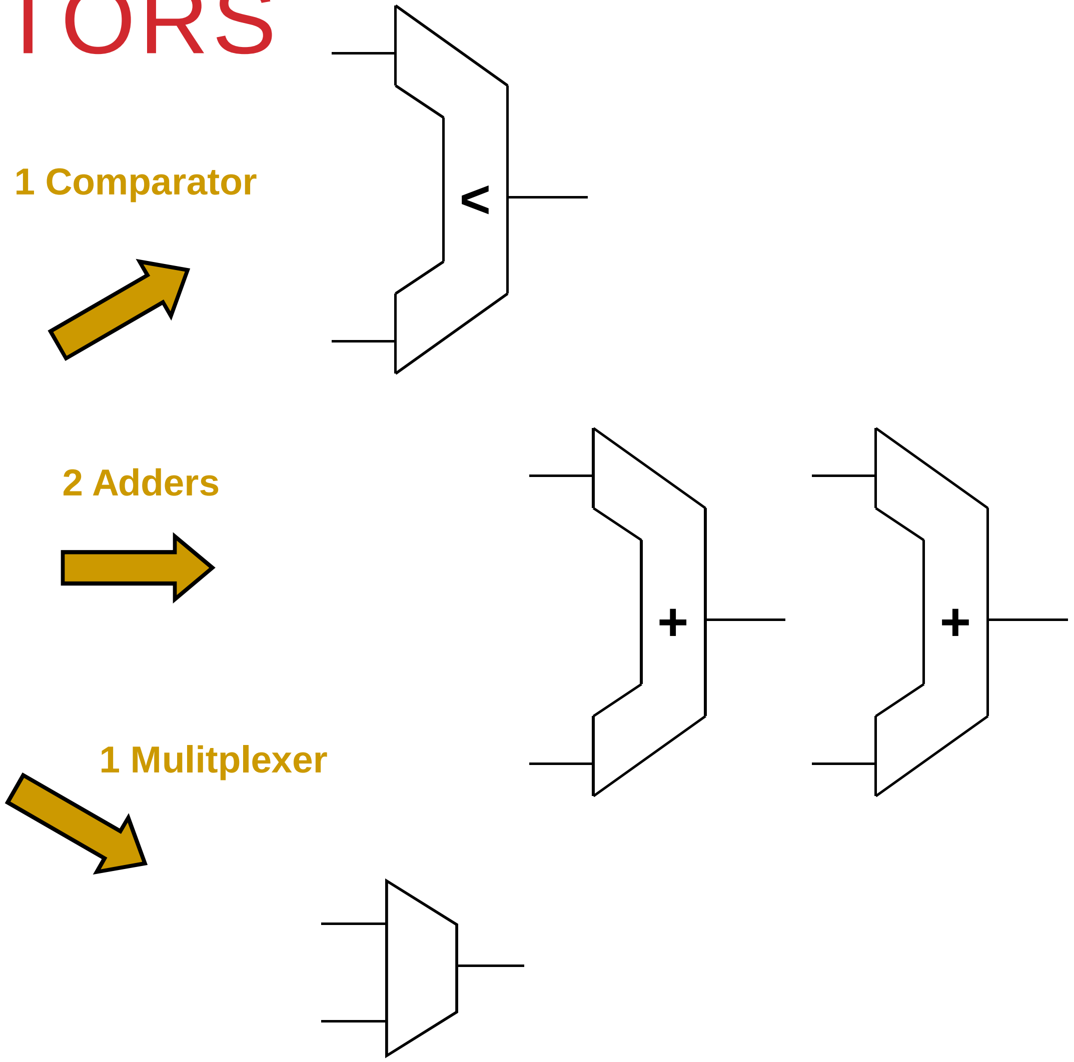
Synthesis tools replace operators with pre-defined (pre-optimized) blocks of logic

Designer should control when & how many operators

GENERATING LOGIC FROM OPERATORS

- *Synthesis tools break down code into logic blocks*
- *They then assemble, optimize & map to hardware*

```
if (sel < 10)
        y = a + b;
else
        y = a + 10;
```



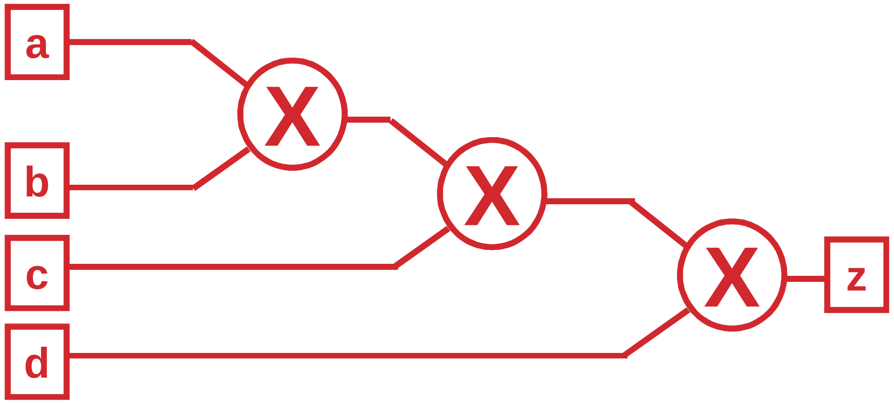
BALANCING OPERATORS

Use parenthesis to define logic groupings

- Increases performance & utilization
- Balances delay from all inputs to output
- Circuit functionality unchanged

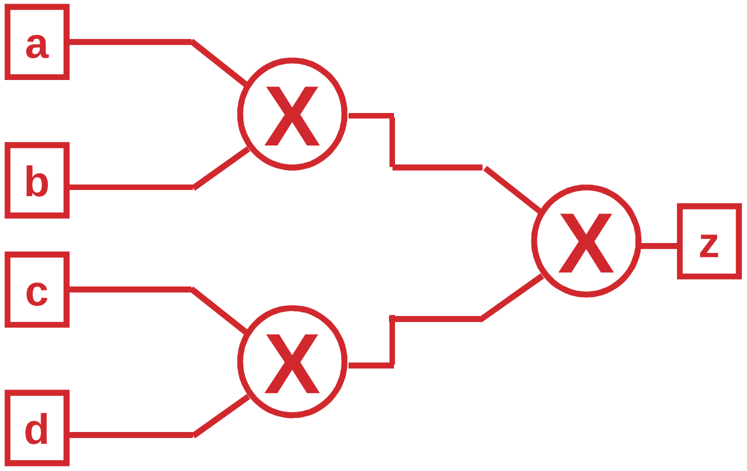
Unbalanced

out = a * b * c * d;



Balanced

out = (a * b) * (c * d);

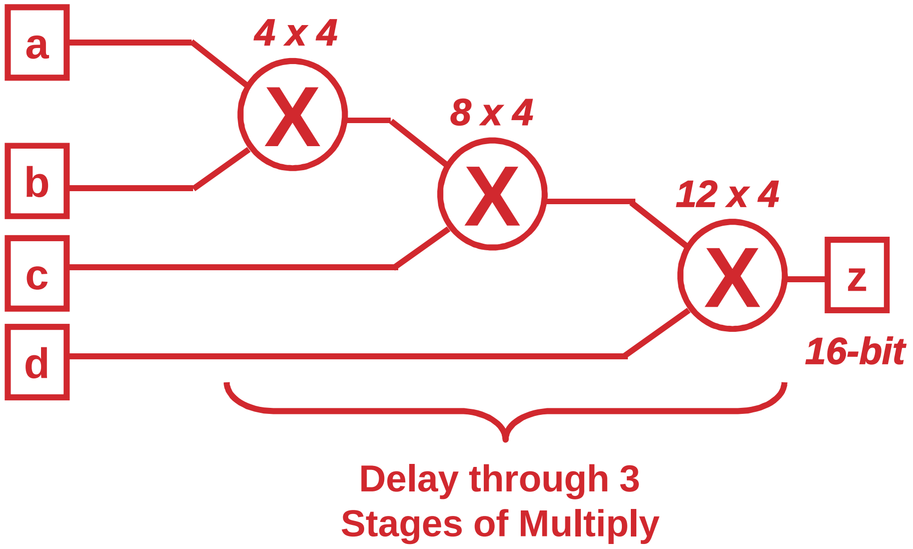


BALANCING OPERATORS: EXAMPLE

a, b, c, d: 4-bit vectors

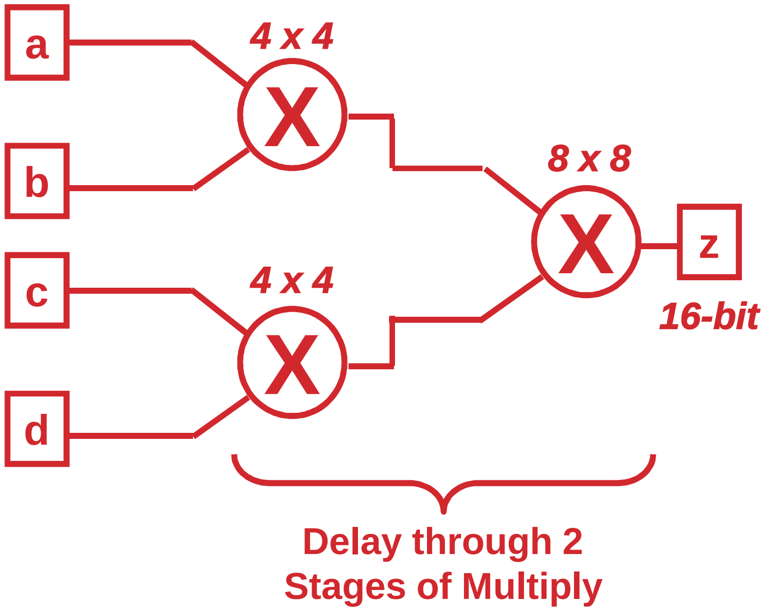
Unbalanced

out = a * b * c * d



Balanced

out = (a * b) * (c * d)



RESOURCE SHARING

Reduces number of operators needed

- Reduces area

Two types

- Sharing operators among mutually exclusive functions
- Sharing common sub-expressions

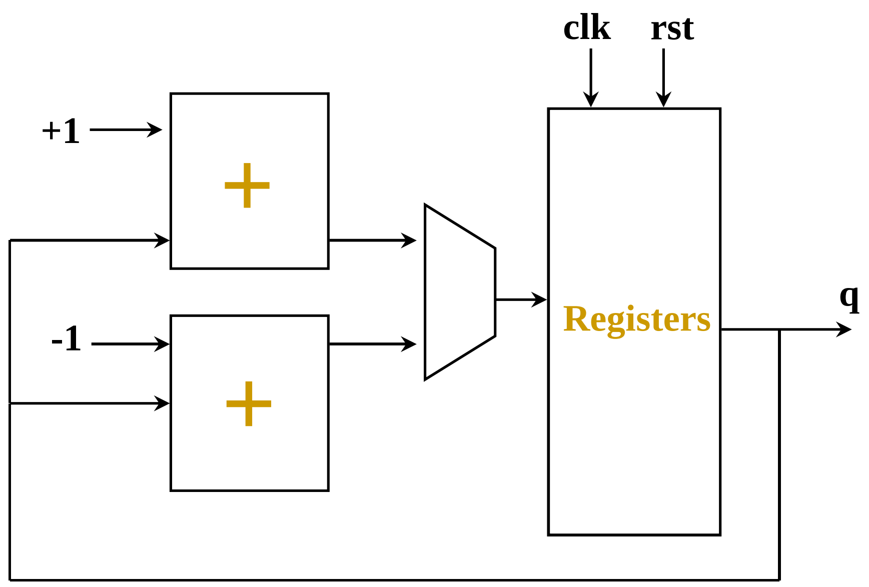
Synthesis tools can perform automatic resource sharing

- Feature can be enabled or disabled

MUTUALLY EXCLUSIVE OPERATORS

```
module test(rst, clk, updn, q);
input rst, clk, updn;
output [7:0] q;
reg [7:0] q;

always@(posedge clk or negedge rst) begin
  if (!rst)
    q<=0;
  else if (updn)
        q <= q + 1;
      else
        q <= q - 1;
end
endmodule
```

- *Up/down counter*
- *EDA tool recognizes this code as a counter and will implement it efficiently, but in general it is not true!*

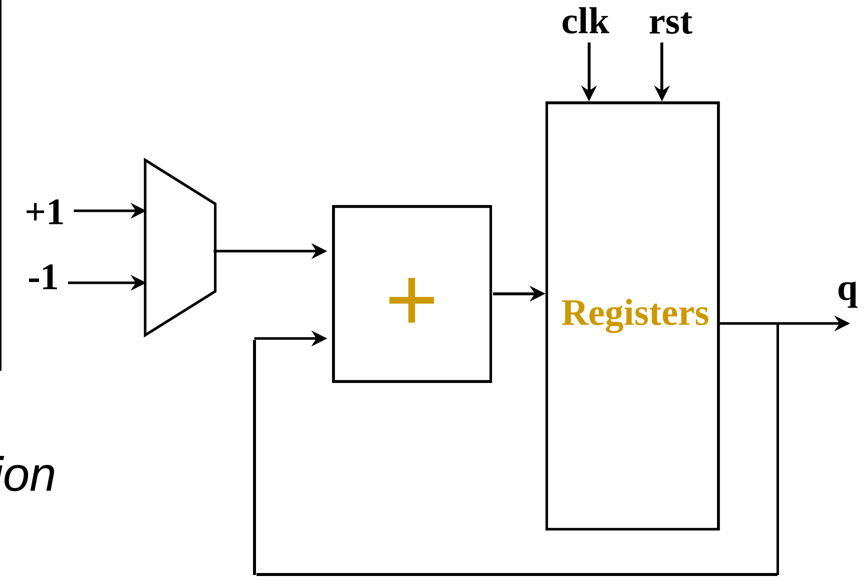


SHARING MUTUALLY EXCLUSIVE OPERATORS

```
module test(rst, clk, updn, q);
input rst, clk, updn;
output [7:0] q;
reg [7:0] q;

always@(posedge clk or negedge rst)
begin
    if (!rst)
        q <= 0;
    else
        q <= q + ( updn ? 1 : -1);
end
endmodule
```

- *Up/down counter*
- *Only one adder required*

Single add operation



LOGIC DUPLICATION

Intentional duplication of logic to reduce fan-out

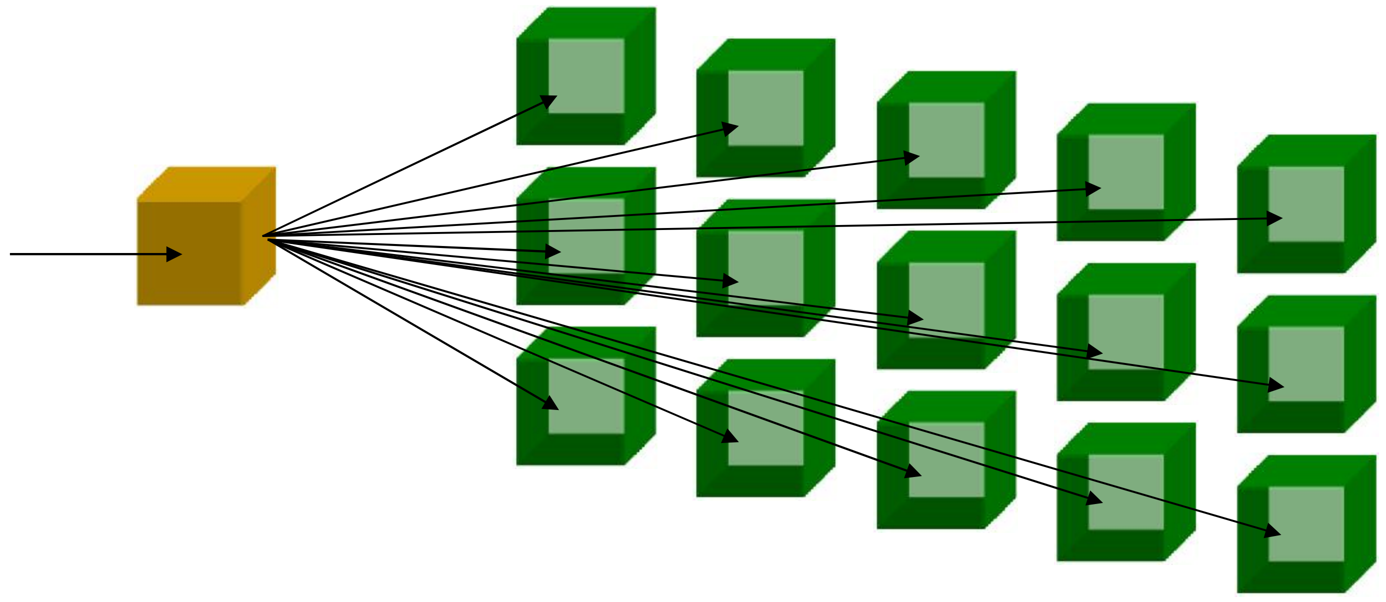
Synthesis tools can perform automatically

- User sets maximum fan-out of a node

FAN-OUT PROBLEMS

High fan-out puts extra loading on to signals slowing performance

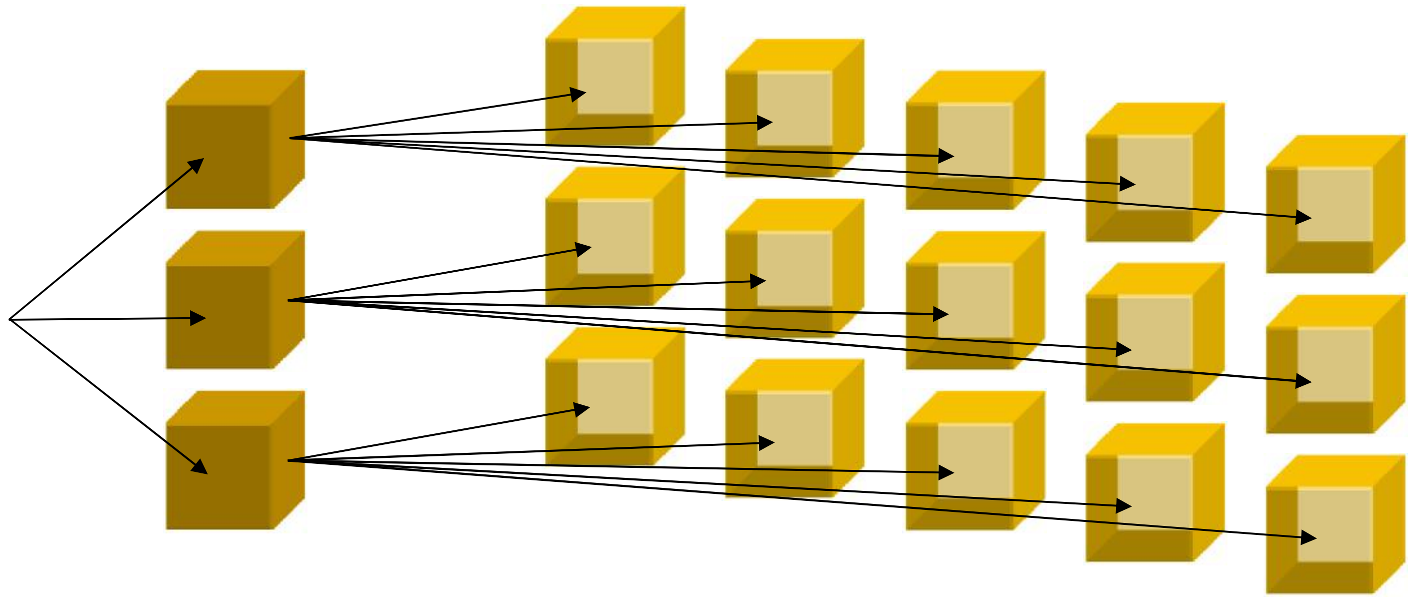
- In this case fan-out is 1 and 15



CONTROLLING FAN-OUT

By replicating logic fan-out can be reduced

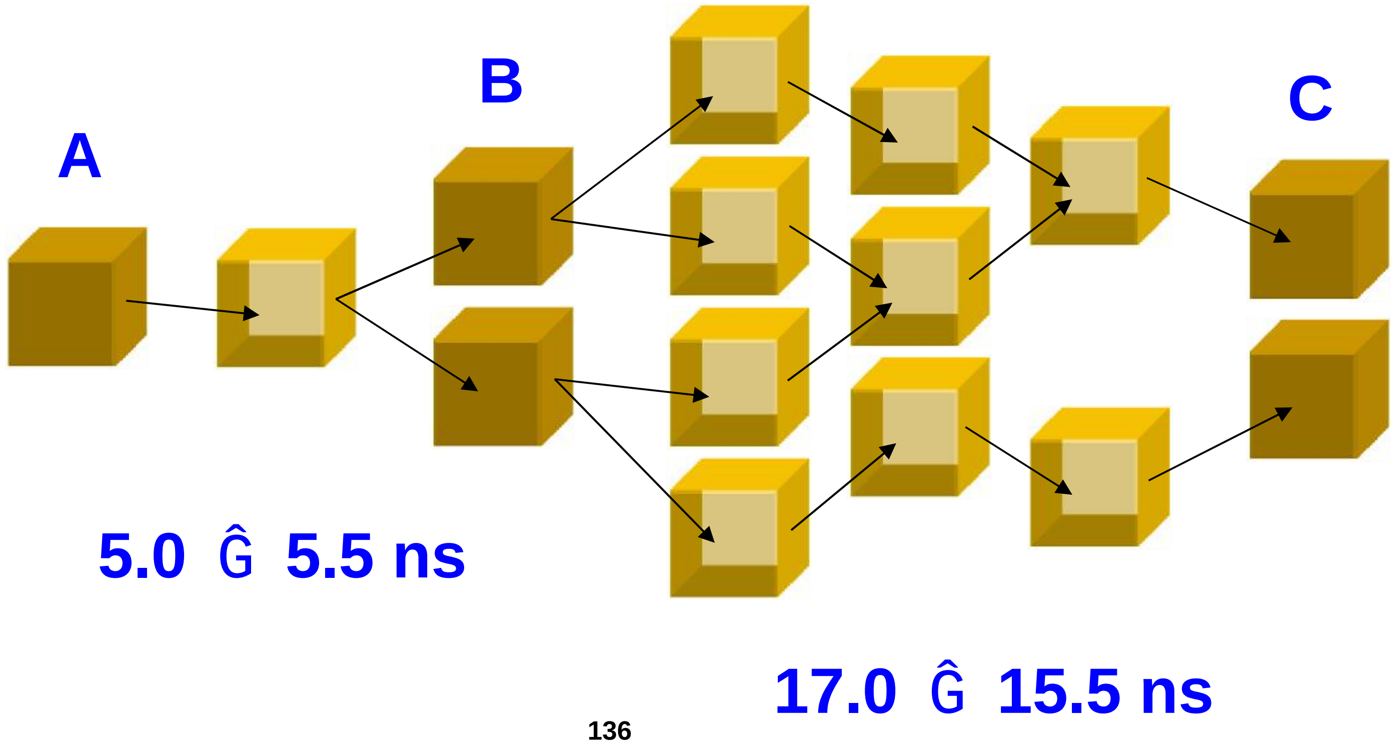
- Worst case path now contains fan-out 3 and 5



REPLICATING REGISTERS

The register at point B is replicated to reduce fan-out from this point

- Path A-B becomes slower, path B-C becomes faster
- As B-C is critical path result is improved performance



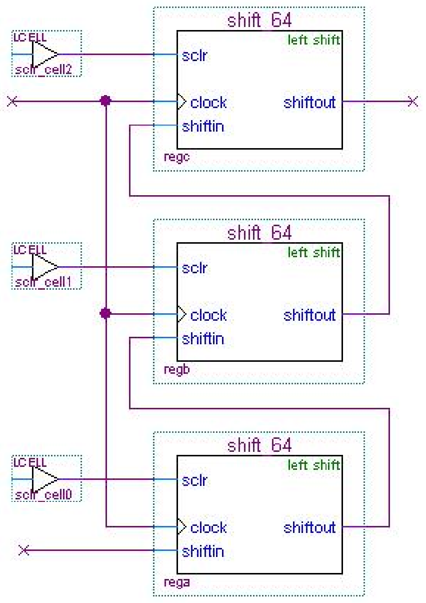
AUTOMATIC FAN-OUT CONTROL

Most Synthesis tools feature options which can limit fan-out

- Advantage: Allows easy experimentation
- Disadvantage: less control over results
    - Knowing which nodes fan-out where helps with floor-planning

SHIFT REGISTER WITH REDUCED FAN-OUT

```
always @(posedge clk) begin
   if (sclr2)
      regc <= 0;
   else
      regc <= {regc[62:0], regb[63]};
   if (sclr1)
      regb <= 0;
   else
      regb <= {regb[62:0], rega[63]};
   if (sclr0)
      rega <= 0;
   else
      rega <= {rega[62:0], shiftin};
end
assign shiftout = regc[63];
```



- *sclr is replicated so that it appears 3 times*
- *This reduces the loading by a factor of three*
- *Fan-out from the previous cell has gone from 1 to 3 but this is insignificant*

PIPELINING

Purposefully inserting register(s) into middle of combinatorial data (critical) path

Increases clocking speed

Adds levels of latency

- More clock cycles needed to obtain output

Some tools perform automatic pipelining

- Same advantages/disadvantages as automatic fan-out

ADDING SINGLE PIPELINE STAGE IN VERILOG

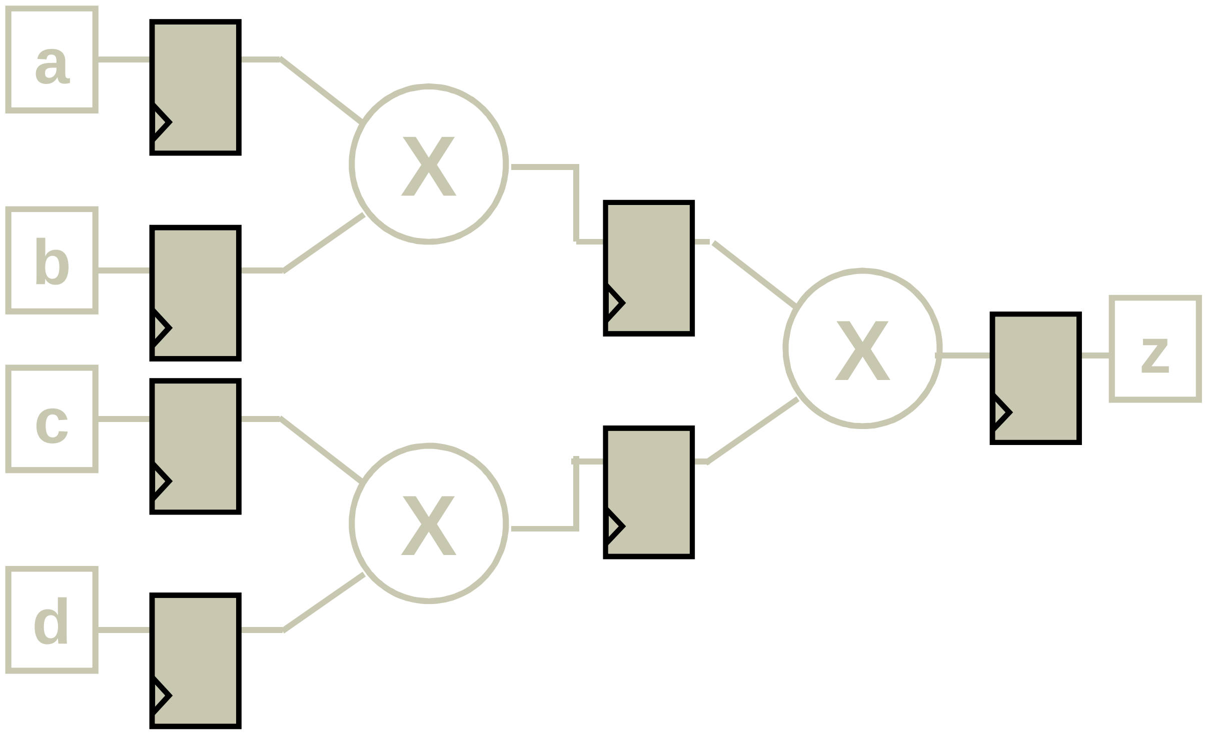
Non-Pipelined

```
module test (clk, clr_n, a, b, c, d, result);
input clk, clr_n;
input [7:0] a, b, c, d;
output reg [31:0] result;
reg [7:0] atemp, btemp, ctemp, dtemp;

always @(posedge clk or negedge clr_n) begin
    if (!clr_n) begin
        atemp <= 0;
        btemp <= 0;
        ctemp <= 0;
        dtemp <= 0;
        result <= 0;
    end else begin
        atemp <= a;
        btemp <= b;
        ctemp <= c;
        dtemp <= d;
        result <= (atemp * btemp) *(ctemp *dtemp);
    end
end
endmodule
```

Pipelined

```
module test (clk, clr_n, a, b, c, d, result);
input clk, clr_n;
input [7:0] a, b, c, d;
output reg [31:0] result;
reg [7:0] atemp, btemp, ctemp, dtemp;
reg [15:0] int1, int2;

always @(posedge clk or negedge clr_n) begin
    if (!clr_n) begin
        atemp <= 0;
        btemp <= 0;
        ctemp <= 0;
        dtemp <= 0;
        int1 <= 0;
        int2 <= 0;
        result <= 0;
    end else begin
        atemp <= a;
        btemp <= b;
        ctemp <= c;
        dtemp <= d;
        int1 <= atemp * btemp;
        int2 <= ctemp * dtemp;
        result <= int1*int2;
    end
end
endmodule
```

PIPELINED 4-INPUT MULTIPLIER



MORE SYNTHESIS GUIDELINES

Proper use of parentheses guide the synthesis tools in eliminating common sub expressions.

We can duplicate the logic which generate the signal for minimizing fan-out.

Trade –off:- loading effect of signal is reduced thereby speeding propagation at the cost of logic and interconnect complexity.

Use of mode "buffer" is not recommended for Synthesis.

- Gate level VHDL Simulation models from ASIC vendors never use the mode buffer.
- There is potential for a port mode type mismatch when using mode "buffer". This may result into problems during integration of different blocks.

MORE SYNTHESIS GUIDELINES

Style of HDL coding often has a direct impact on the result the synthesis tool delivers.

- Partitioning of designs play a very crucial role in achieving good synthesis results.
- “Synthesis tools provides best results when the critical path lies in one hierarchical block as opposed to traversing multiple hierarchical blocks.”

Avoid OVER-constraining the design

- Design performance suffers.
    - Critical timing paths get the best placement and fastest routing options.
    - As the number of critical path increases, the ability to obtain the design performance objective decreases.
    - Run times increase.

MORE SYNTHESIS GUIDELINES

Constraining designs:-

- Constraints are means of communicating our requirements to the synthesis and backend tools.

Categories of constraints are

- Timing constraints
    - Maximum frequency
    - Duty cycle
    - Input delays
    - Output delays
    - False paths

Run a trial without constraints to get an idea of what is possible and what is not.

Leave some allowance for future expansions also.

MORE SYNTHESIS GUIDELINES

For best result:-

- Use technology primitives(macros) from the target technology libraries whenever possible.
- Try small designs on the target technology to find its limitations and strengths.
- Partition and design correctly.
    - Eliminate glue logic at the top level.
    - Partition block size based on the logic function, central processing unit(CPU)resources and memories.
    - Separate random logic from structure logic –data path logic.
    - It is recommended that timing-sensitive modules are separated from the area sensitive modules.
        - This allows designers to apply different optimization strategies depending on the design goals.

MORE SYNTHESIS GUIDELINES

Simulate your design before synthesis

- Logic errors that are not caught are passed on during synthesis and the synthesized results will contain the same logic errors.

Avoid combinational loops in the processes.

If a port is declared to be an integer data type, the range should be specified, else the tool will infer a 32 bit port.

Avoid mixed clock edges

- If a larger number of both positive and negative edge flip flop are required then they should be placed in different modules.

MORE SYNTHESIS GUIDELINES

Coding for performance

- Common mistake is to ignore hardware and start as if programming. To achieve best performance, the designer must think about hardware.

- Improve performance by:-
    - Avoiding unnecessary priority structures in logic.
    - Optimizing logic for late arriving signals.
    - Structuring arithmetic for performance.
    - Avoiding area inefficient code.
    - Buffering high fan-out signals.
    - Pipelining for high performance.

REFERENCES

Samir Palnitkar, *"Verilog HDL: A Guide to Digital Design and Synthesis*" Second edition, Prentice Hall, 2003.

J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, 2nd ed. Springer, 2003.

J Bhaskar, "*A Verilog HDL Primer", Third* Edition, Star Galaxy Publishing, 2005

Clifford E. Cummings*, – "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!"* Available:http://www.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=synth-verilog-cummins.pdf

Clifford E. Cummings*, – "Verilog Nonblocking Assignments with Delays, Myths & Mysteries"*

Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.149.3862&rep=rep1& type=pdf

Altera - Recommended HDL Coding Styles – Altera, Aug 18 2014, https://www.altera.co.jp/ja_JP/pdfs/literature/hb/qts/qts_qii51007.pdf

Altera - Quartus II Handbook Volume 1: Design and Synthesis, May 4 2015, https://www.**altera**.com/literature/hb/qts/qts_qii5v1.pdf

Xilinx Synthesis and Simulation Design Guide, Sep21 2004 , www.**xilinx**.com/itp/**xilinx**10/books/docs/sim/sim.pdf

Philippe Garrault and Brian Philofsky- "HDL Coding Practices to Accelerate Design Performance" Jan 6 2006, https://www.**xilinx**.com/support/**documentation**/white_papers/wp231.pdf

"SystemVerilog – UNIFIED HARDWARE DESIGN, SPECIFICATION, AND VERIFICATION LANGUAGE" – The IEEE 1800-2012 LRM, Web link: http://standards.ieee.org/getieee/1800/download/1800-2012.pdf