Advanced Verilog Design Techniques

Agenda

- Writing Synthesizable Verilog
- Inferring Common Logic Functions
- Coding State Machines
- Improving Logic Utlization & Performance
- Introduction to Testbenches
- Writing Parameterized Code

Advanced Verilog Design Techniques

Writing Synthesizable Verilog

Simulation vs. Synthesis

- Simulation
  - Code executed in the exact way it is written
  - User has flexibility in writing behavioral model
  - Initialization of logic supported
- Synthesis
  - Code is interpreted & hardware created
    - Knowledge of PLD architecture is important
  - Synthesis tools require certain coding to generate correct logic
    - Subset of Verilog language supported
    - Coding style is important for fast & efficient logic
  - Initialization controlled by device
    - Logic implementation can be adjusted to support initialization
- Pre- & post-synthesis logic should operate the same

Writing Synthesizable Verilog

- Synthesizable Verilog constructs
- Sensitivity lists
- Blocking vs. non-blocking
- Latches vs. registers
- **if-else** structures
- **case** statements
- Using functions and tasks
- Combinatorial loops
- Gated clocks

Some Synthesizable Verilog Constructs

- Module
- Ports
- Parameter
- Net data types
  - wire, tri
  - tri0, tri1
  - wor, wand, trior, triand
  - supply0, supply1
- Variable (register) data types
  - reg, integer
- Assign (continuous)
- Always & initial blocks
- Gates
  - and, nand, nor, or xor, xnor
  - buf, not, bufif0, bufif1, notif0, notif1
- UDPs
- Blocking (=) & non-blocking (<=)
- Sequential blocks (begin/end)
- Tasks & functions
- Bit & part select
- Behavioral constructs
  - If-else statements
  - Case statements
  - Loop statements (with restrictions)
- Disable
- Module instantiation
- Readmemb & readmemh system tasks
- Compiler directives
  - `define, `undef, `ifdef, `else, `endif
  - `include
  - `unconnected_drive, `nounconnected_drive
  - `resetall

– *Synthesis tools may place certain restrictions on supported constructs*

– *See the online help in Quartus II (or your target synthesis tool) for a complete list*

  –*The Quartus II software supports many constructs not supported by other synthesis tools*

Some Non-Synthesizable Verilog Constructs

- Net data type
  - trireg
- Variable data types
  - Real, time
- Gates
  - rtran,tran,tranif0, tranif1 rtranif0,rtranif1
- Switches
  - cmos, nmos, rcmos, rnmos, pmos, rpmos
- Pullup & pulldown
- Strengths*
- Assign (procedural continuous) & deassign
- Force & release
- Delay controls*
- Event (intra-procedural)
- Named event
- Fork & join
- Specify blocks*
- `timescale*
- Most system tasks
  - Display systems tasks*

- *These are some of the constructs not supported by Quartus II synthesis*
  - ** = Ignored for synthesis*
- *See the online help in Quartus II (or your target synthesis tool) for a complete list*

See:

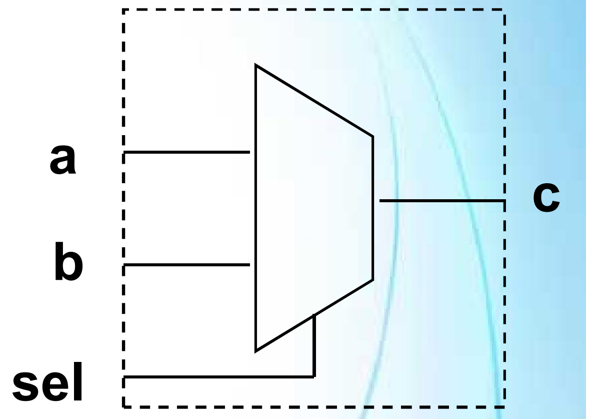
- http://eesun.free.fr/DOC/VERILOG/verilog_manual1.html#14.1
- http://www.asic-world.com/verilog/synthesis2.html
- Quartus II HELP

Two Types of RTL Processes

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

```
always @ (a, b, sel)
always @ *
```

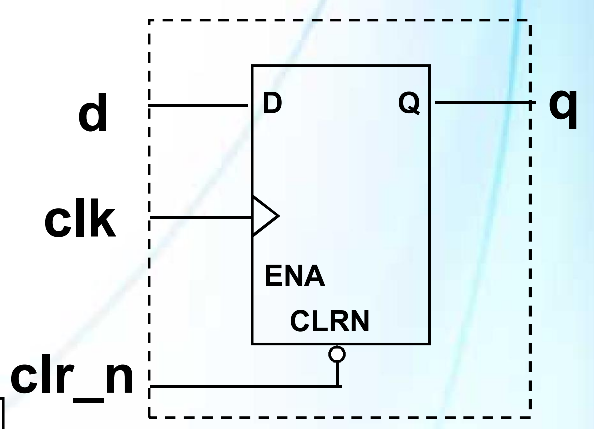
Sensitivity list includes all inputs used In the combinatorial logic



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

```
always @(posedge clk, negedge clr_n)
```

Sensitivity list does not include the **d** *input, only the clock and control signals*



Sensitivity Lists

- Incomplete sensitivity list in an **always** block may result in differences between RTL and gate-level simulations
  - Synthesis assumes complete sensitivity list
  - Should include all inputs to the procedural block (or use @* shortcut)

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

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

```
always @ *
    y = a & b & c;
```

Correct way for the intended AND logic !

Blocking vs. Non-Blocking Recommendations

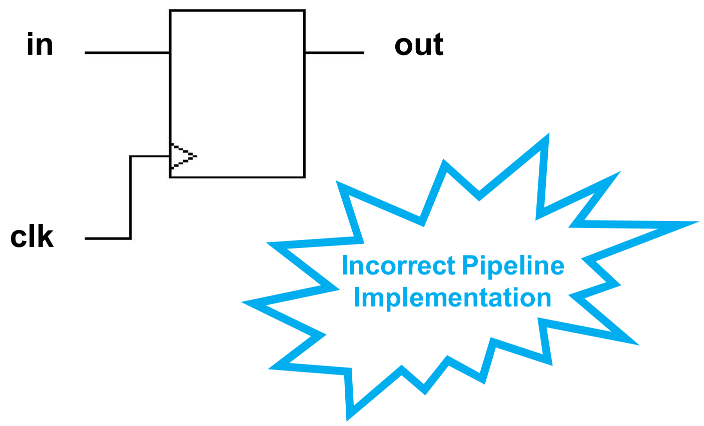
- Blocking
  - Use blocking assignments in combinatorial procedural blocks
  - Blocking assignments easier to read
  - Blocking assignments uses less memory & simulates faster
    - No scheduling of updated signals
- Non-blocking
  - Use non-blocking assignments for clocked procedural blocks

Clocked Process Example

Blocking (=)

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

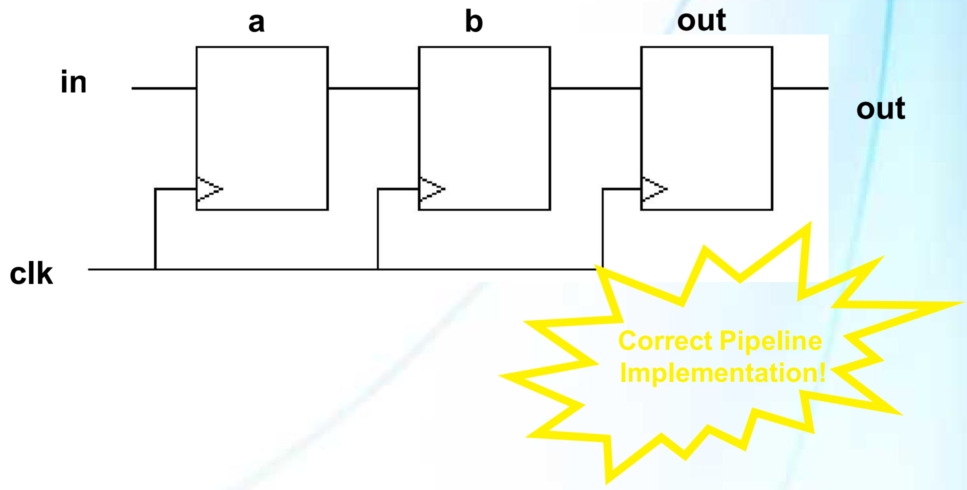
Synthesized Result:



Nonblocking (<=)

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

Synthesized Result:



Latches vs. Registers

- Altera devices have registers in logic elements, not latches
- Latches are implemented using combinatorial logic & can make timing analysis more complicated
  - Look-up table (LUT) devices use LUTs in combinatorial loops
  - Product-term devices use more product-terms
- 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
    - Lead to simulation/synthesis mismatches

if-else Structure

- **if-else** structure implies prioritization and dependency
  - Nth clause implies all N-1 previous clauses not true
    - Beware of needlessly “ballooning” logic

Logical Equation

$$(\langle cond1\rangle \bullet A) + (\langle \underline{cond1}\rangle' \bullet \langle cond2\rangle \bullet B) + (\langle \underline{cond1}\rangle' \bullet \langle \underline{cond2}\rangle' \bullet cond3 \bullet C) + \ldots$$

  - Consider restructuring **if** statements
    - May flatten the multiplexer and reduce logic

```
IF <cond1> THEN
     IF <cond2> THEN
```

$\Rightarrow$

```
IF <cond1> AND <cond2> THEN
```

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

When Writing if-else Structures

- Cover all cases
  - Uncovered cases in combinatorial processes result in latches
- Assign values before reading data type object
  - Reading unassigned data type object results in latches
- For efficiency, consider
  - Using don't cares (X) for final **else** clause (avoiding unnecessary default conditions)
    - Synthesis tool has freedom to encode don't cares for maximum optimization
  - Assigning initial values and explicitly covering only those results different from initial values

Prioritization & Dependency (if-else)

- Nested **else if** statements imply prioritization 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, 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)

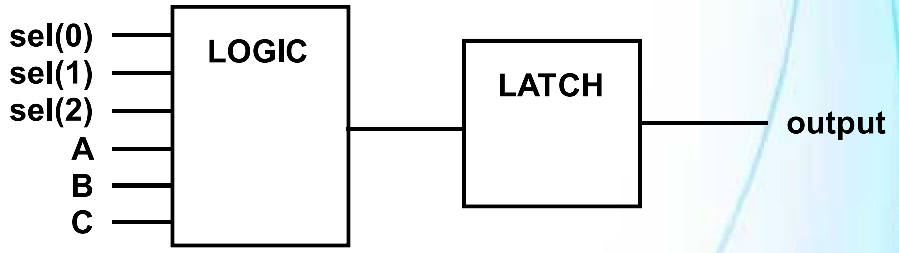
- Individual **if** statements where logic is exclusive *may* be more efficient

```
reg [4:0] state;
parameter s0 = 5'h0, s1 = 5'h11, s2 = 5'h12, s3 = 5'h14, s4 = 5'h18;
always @ (posedge clk, 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

- Combinatorial processes that do not cover all possible input conditions generate latches

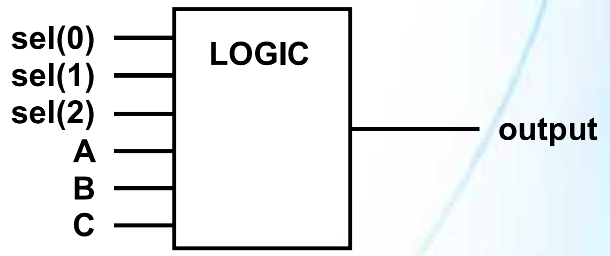
```
always @ *
begin
        if (sel == 3'b001)
                out = a;
        else if (sel == 3'b010)
                out = b;
        else if (sel == 3'b100)
                out = c;
end
```



Unwanted Latches Removed

- Close all **if-else** structures
  - If possible, assign “don’t care’s” to else clause for improved logic optimization

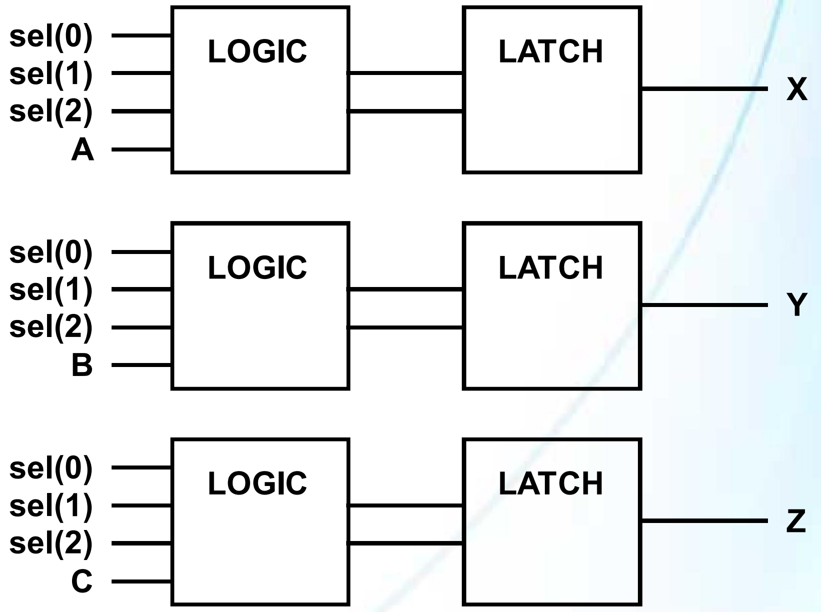
```
always @ *
begin
        if (sel == 3’b001)
                output = a;
        else if (sel == 3’b010)
                output = b;
        else if (sel == 3’b100)
                output = c;
        else
                output = 3’bx;
end
```



Mutually Exclusive if-else Latches

- Beware of building unnecessary dependencies
  - e.g. Outputs x, y, z are mutually exclusive, **if-else** causes all outputs to be dependant on all tests & creates latches

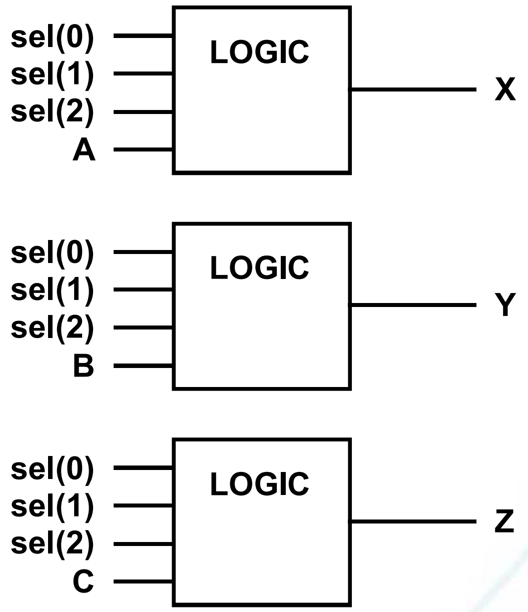
```
always @ (sel,a,b,c)
begin
        if (sel == 3'b010)
                x = a;
        else if (sel == 3'b100)
                y = b;
        else if (sel == 3'b001)
                z = c;
        else
                x = 0;
                y = 0;
                z = 0;
end
```



Mutually Exclusive Latches Removed

- Use separate **if** statements and close each with initialization or **else** clause

```
always @ (sel,a,b,c)
begin
        x = 0;
        y = 0;
        z = 0;
        if (sel == 3’b010)
                x = a;
        if (sel == 3’b100)
                y = b;
        if (sel == 3’b001)
                z = c;
end
```

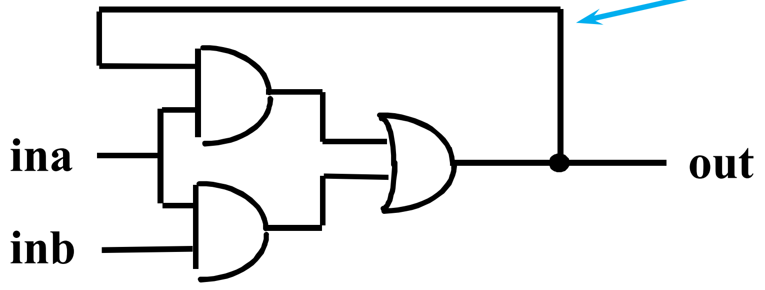


Unwanted Latches – Nested if Statements

- Use nested **if** statements with care
  - e.g. These nested **if** statements do not cover all possible conditions (open **if** statements) & latch is created

```
always @ (ina, 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	?

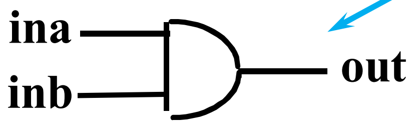


- Uncovered cases infer latches
  - No default value for data type objects
- Equation
  - A1L3 = LCELL(ina & (A1L3 # inb));

Unwanted Latches Removed – Nested if Statements

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

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

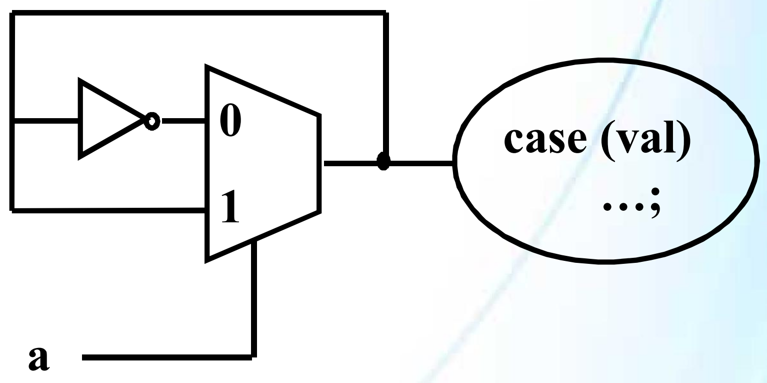


- Using initialization to cover all cases; no latch inferred
- Equation
  - A1L3 = inb & ina;

Variable Not Initialized

- Reading a variable data type object in a combinatorial process before it has been assigned a value infers a latch

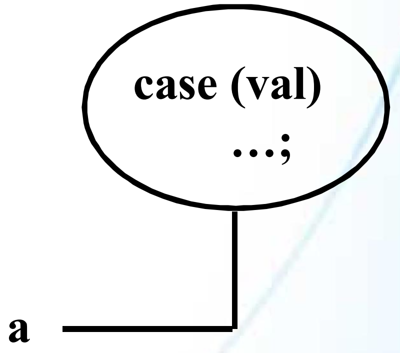
```
always @ *
        begin
                if (a == 1) val = val;
                else val = val + 1’b1;
                case (val)
                                1’b0 : q = i[0];
                                1’b1 : q = i[1];
                endcase
        end
```



Assigning Initial Value to Variable

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

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



Case Statements

- Case statements usually synthesize more efficiently when mutual exclusivity exists
- When writing case statements
  - Make sure all case items are unique/parallel
    - Verilog language does not require this
    - Non-parallel cases infer priority encoders (i.e. less efficient logic)
  - Cover all cases
    - Like **if-else**, uncovered cases infer latches
    - Caused by
      - Incomplete case statement
      - Outputs not defined for one case item

Case Statement Recommendations

- Initialize all case outputs <u>or</u> ensure outputs assigned in each case
- Use **default** clause to close undefined cases (if any remain)
- Assign initialized or default values to don't cares (X) for further optimization, if logic allows

Case Statement Example

```
always @ *
begin
//   in1 = 32'bx;
     case (state)
              4'b0000:   in1 = data_a;
              4'b1001:   in1 = data_b;
              4'b1010:   in1 = data_c;
              4'b1100:   in1 = data_d;
     endcase
end
```

Incomplete case

```
always @ *
begin
     case (state)
              4'b0000:   in1 = data_a;
              4'b1001:   in1 = data_b;
              4'b1010:   in1 = data_c;
              4'b1100:   in1 = data_d;
              default:   in1 = 32'bx;
     endcase
end
```

Completed case

“Full” Case

- Definition: 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
  - Used to reduce logic count as synthesis accounts only for defined possibilities
  - May cause pre-synthesis and post-synthesis simulation mismatch

Mismatched Code (BAD)

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

Full Statement (GOOD)

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

“Parallel” Case

- Definition: Only possible to match a case expression to one and only one case item
- Non-parallel case statements infer priority encoders
  - Code all intentional priority encoders with if-else statements for clarity
- Adding “parallel_case” synthesis directive to non-parallel case statement
  - Used to reduce the logic size as synthesis tool removes priority
  - May cause pre-synthesis and post-synthesis simulation mismatch

Mismatched Code (BAD)

```
always @ (a) begin
        casez (a) //synthesis parallel_case
                3’b1??: out = 2’b01;
                3’b?1?: out = 2’b10;
                3’b??1: out = 2’b11;
        endcase
 end
```

Non-Prioritized Code (GOOD)

```
always @ (a) begin
        case (a)
                3’b100: out = 2’b01;
                3’b010: out = 2’b10;
                3’b001: out = 2’b11;
                default: out = 2’bxx;
        endcase
 end
```

Using Functions & Tasks

- Functions and tasks can be used in synthesized modules to make code more readable/reusable

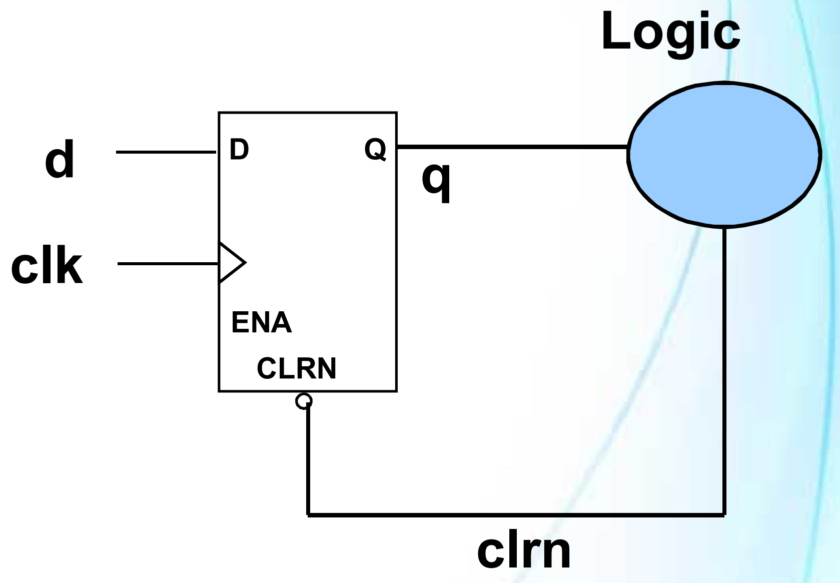
- Functions
  - Synthesize to combinational logic
  - Variables in functions must be local
- Tasks
  - Synthesize to combinational or sequential logic
  - Cannot contain time delay

- Each call generates a separate block of logic
  - No logic sharing
  - Implement resource sharing, if possible (discussed later)

Combinational Loops

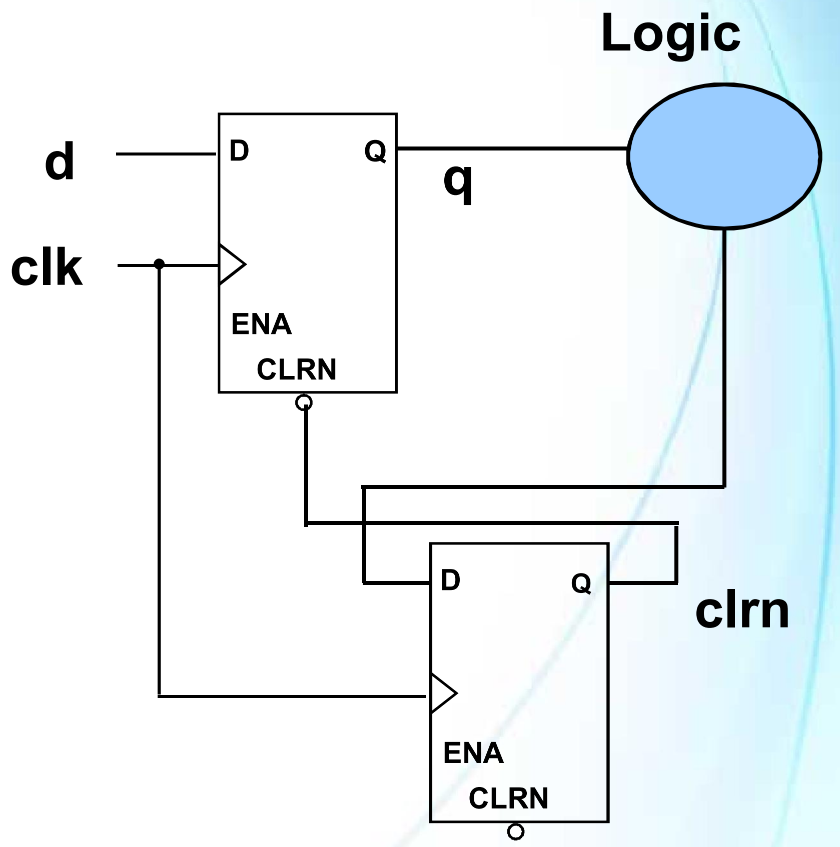
- Common cause of instability
- Behavior of loop depends on the relative propagation delays through logic
  - Propagation delays can change
- Simulation tools may not match hardware behavior

```
always @ (posedge clk, negedge clrn)
 begin
        if (!clrn)
                q <= 0;
        else
                q <= d;
end

assign clrn = (ctrl1 ^ ctrl2) & q;
```



Combinational Loops

- All feedback loops should include registers



```
always @ (posedge clk, negedge clrn)
 begin
      if (!clrn)
            q <= 0;
      else
            q <= d;
end

always @ (posedge clk)
      clrn <= (ctrl1 ^ ctrl2) & q;
```

Gated Clocks

- Can lead to both functional and timing problems
  - Clock behavior subject to both synthesis and placement & routing
  - Can be a source of additional clock skew
  - Glitches on clock path possible
- Recommendations:
  - Use clock enables for clock gating functionality
  - Use dedicated device resources (e.g. clock control blocks) to gate clocks synchronously and reduce power
  - If you must build your own gating logic
    - Use a synchronous gating structure
    - Ensure global clock routing is used for clock signal
    - Gate the clock at the source

Gated Clock Examples

```
assign g_clk = gate & clk;

always @ (posedge g_clk, negedge clrn)
 begin
        if (!clrn)
                q <= 0;
        else
                q <= d;
end
```

Poor clock gating – Active clock edges occurring near gate signal changes may result in glitches

```
always @ (negedge clk)
        sgate = gate

assign g_clk = sgate & clk;

always @ (posedge g_clk, negedge clrn)
 begin
        if (!clrn)
                q <= 0;
        else
                q <= d;
end
```

Better clock gating – Gate signal clocked by falling edge clk, so gate may only change on inactive clock edge (Use OR gate when falling edge is the active clock edge)

Advanced Verilog Design Techniques

Inferring Common Logic Functions

Inferring Logic Functions

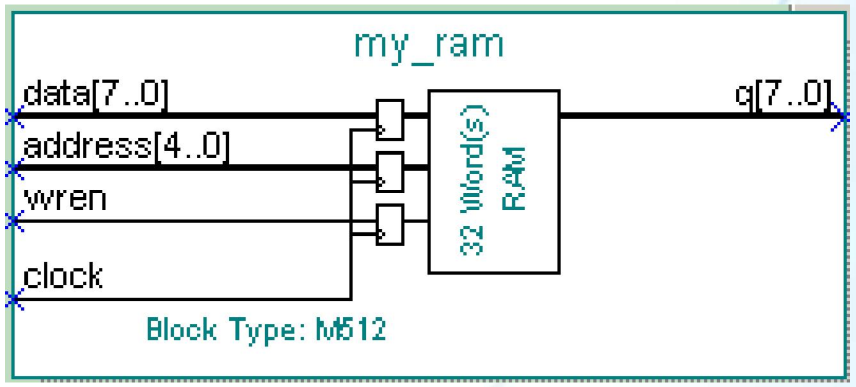
- Using behavioral modeling to describe logic blocks
- Synthesis tools recognize description & insert equivalent logic functions (e.g. megafunctions)
  - Functions typically pre-optimized for utilization or performance over general purpose functionally equivalent logic
  - Use synthesis tool's templates (if available) as starting point
  - Use synthesis tool's graphic display to verify logic recognition
- Makes code vendor-independent

Logic Inference Example

```
always @ (posedge clock)
    begin
        if (wren)
            mem[address] <= data;
        q <= mem[address_out];
    end
```

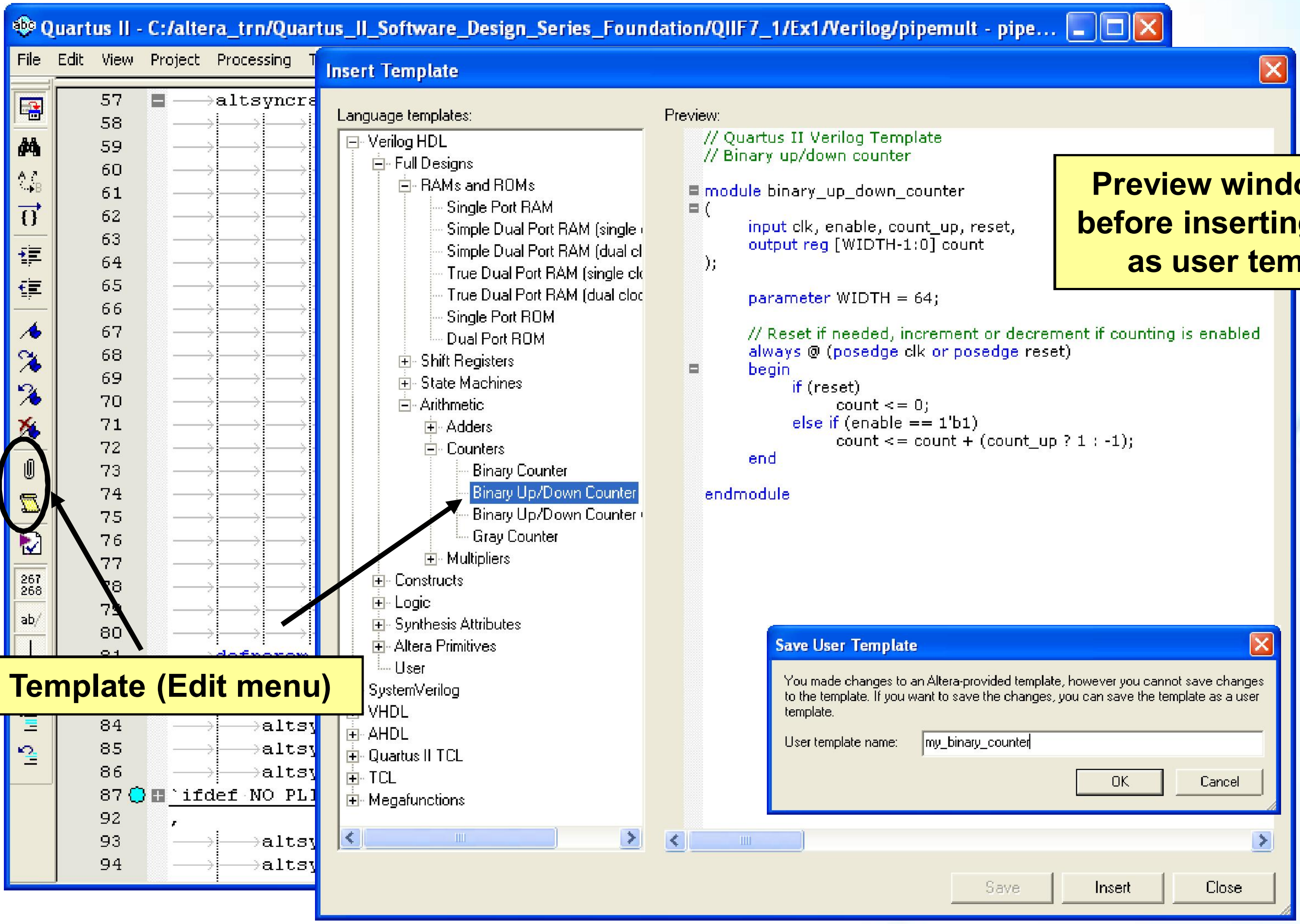
Synthesis tool sees

Replaces with



Altera megafunction and/or library cells

Verilog Templates



```
// Quartus II Verilog Template
// Binary up/down counter

module binary_up_down_counter
(
        input clk, enable, count_up, reset,
        output reg [WIDTH-1:0] count
);

        parameter WIDTH = 64;

        // Reset if needed, increment or decrement if counting is enabled
        always @ (posedge clk or posedge reset)
        begin
                if (reset)
                        count <= 0;
                else if (enable == 1'b1)
                        count <= count + (count_up ? 1 : -1);
        end

endmodule
```

You made changes to an Altera-provided template, however you cannot save changes to the template. If you want to save the changes, you can save the template as a user template.

User template name: my_binary_counter

Preview window: edit before inserting & save as user template

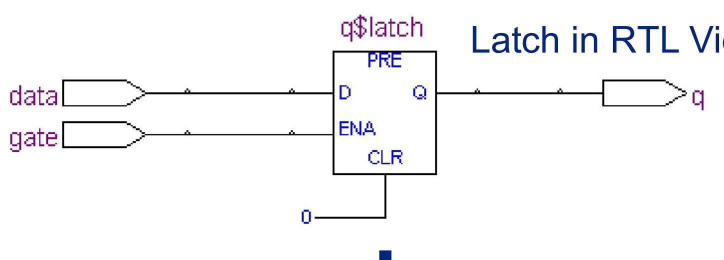
Insert Template (Edit menu)

Inferring Common Functions

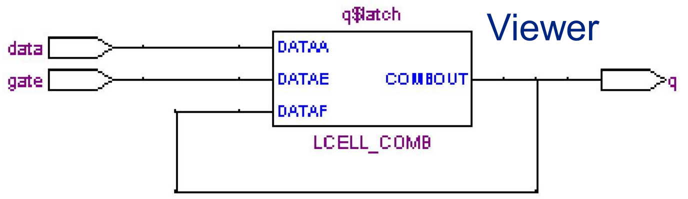
- Latches
- Registers
- Counters
- Tri-states
- Memory

“Wanted” Latches

```
module latches (
     input d, gate;
     output reg q
);

wire d, gate;

always @(d, gate)

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



Latch in RTL Viewer



Latch in Technology Viewer

sensitivity list includes both inputs

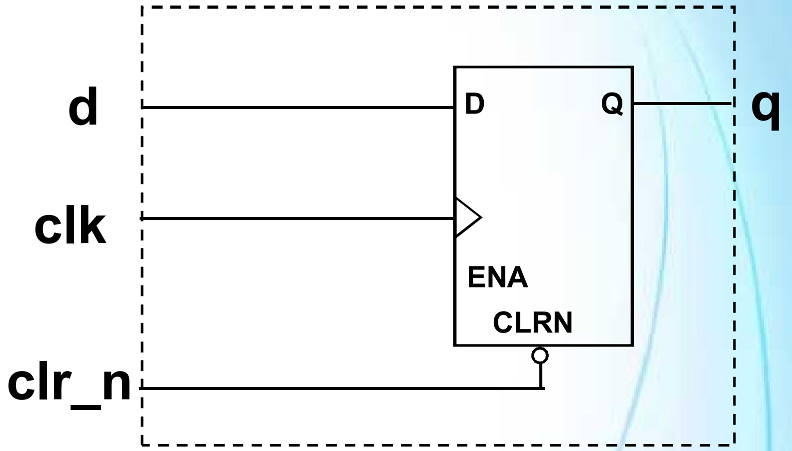
level sensitive…not edge

What happens if gate = ‘0’?

⇨ ***Implicit Memory & Feedback***

Inferring Flip-Flops

```
module basic_dff (
    input clk, d, clr_n;
    output reg q;
);
always @(posedge clk, negedge clr_n)
    if (!clr_n)
        q <= 0;
    else
        q <= d;
endmodule
```



- *Simple register logic with reset*
- *<u>Recommendation</u>: Always use reset (asynchronous or synchronous) to get system into known or initial state*

Secondary Control Signals

- Register control signals vary between FPGA & CPLD families
  - Clear, preset, load, clock enable, etc.
- Avoid using signals not available in architecture
  - Functionality of design supported by creating extra logic cells
  - Less efficient, possibly slower results

DFF with Secondary Control Signals

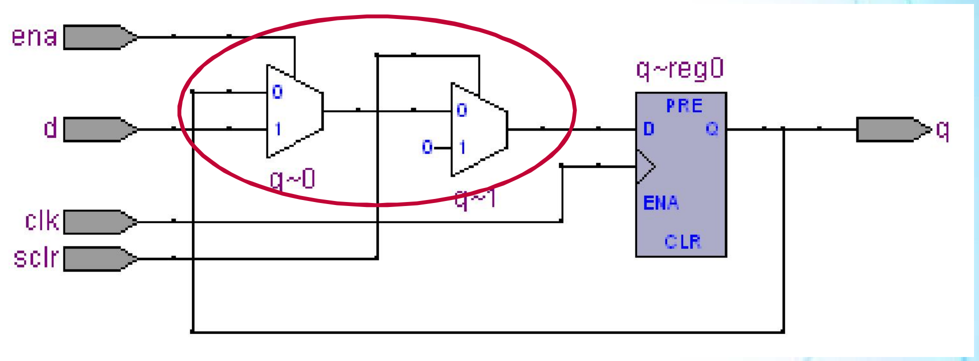
```
module dff_full (
        input clk, ena, d,
        input clr_n, sclr, pre_n,
        input aload, sload, adata, sdata,
        output reg q
);

always @(posedge clk, negedge clr_n,
                negedge pre_n, posedge aload)
        if (!clr_n)
                q <= 1'b0;
        else if (!pre_n)
                q <= 1'b1;
        else if (aload)
                q <= adata;
        else if (ena)
                if (sclr)
                        q <= 1'b0;
                else if (sload)
                        q <= sdata;
                else
                        q <= d;
endmodule
```

- *This is how to implement all asynchronous and synchronous control signals for the Altera PLD registers*
    - *Conditions in the sensitivity list are asynchronous*
    - *Conditions not in the sensitivity list are synchronous*
- *Remove signals not required by your logic*
    - *Most PLD architectures would require additional logic to support all at once*

- *Why is this asynchronous signal **adata** left out of the sensitivity list?*
    - *Verilog limitation*
    - *Inferred correctly (simulation mismatch)*

Incorrect Control Signal Priority

```
module dff_sclr_ena (
      input clk, d, ena, sclr,
      output reg q
);

always @ (posedge clk)
      if (sclr)
            q <= 1'b0;
      else if (ena)
            q <= d;
endmodule
```



- 2 control signals
- Considerations
  - Do the registers in the hardware have both ports available?
  - How does hardware behave? Does clear or enable have priority?
- *Sync clear has priority enable over in code*
- *Enable has priority over sync clear in silicon*
- *Additional logic needed to force code priority*

Control Signal Priority

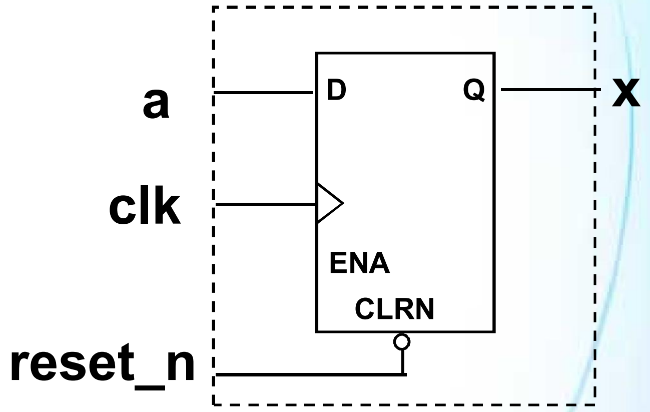
1. Asynchronous clear (aclr)
2. Asynchronous preset (pre)
3. Asynchronous load (aload)
4. Enable (ena)
5. Synchronous clear (sclr)
6. Synchronous load (sload)

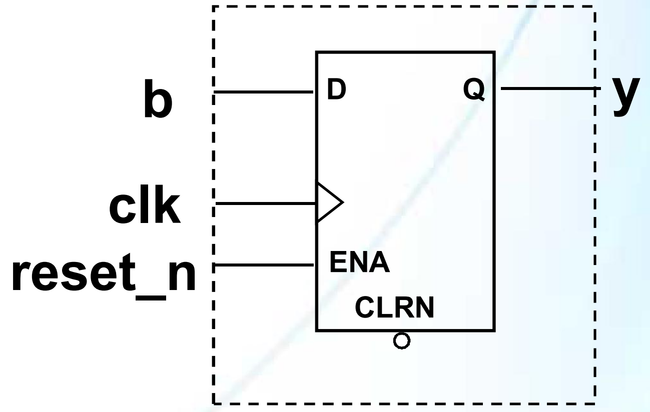
- Same for all Altera FPGA families
  - All signals not supported by all families
- Re-ordering may generate extra logic

Incorrect Control Logic

```
module dff_inc (
        input clk, reset_n, a, b,
        output reg x, y
);
always @(posedge clk, negedge reset_n)
        if (reset_n == 0)
                x <= 0;
        else begin
                x <= a;
                y <= b;
        end
endmodule
```

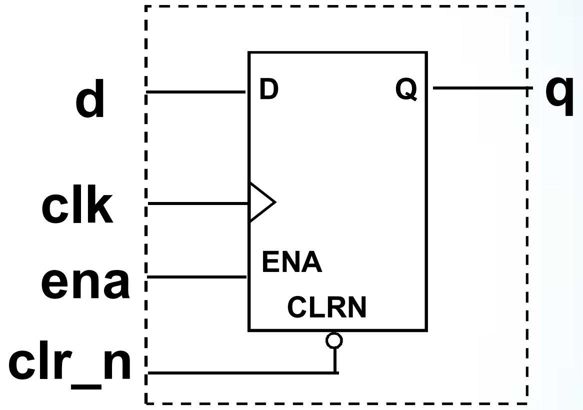
- ***y*** *Is Not Included in Reset Condition*
- ***reset*** *Clears* ***x*** *but Acts More Like an Enable for* ***y***





DFF with Clock Enable

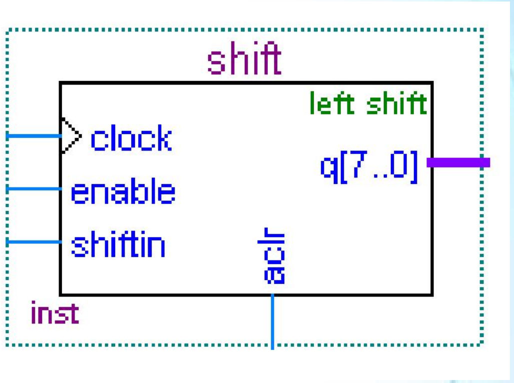
```
module dff_ena (
        input clk, clr_n, d,
        input ena_a, ena_b, ena_c,
        output reg q
);

        always @ (posedge clk, negedge clr_n)
                if (clr_n == 1'b0)
                        q <= 1'b0;
                else if (ena == 1'b1)
                        q <= d;

        assign ena <=
                        (ena_a | ena_b) ^ ena_c;

endmodule
```



- *To ensure that this is synthesised using DFFE primitives (DFF with enable)*
  - *–Place the enable statement directly after the rising edge statement*
  - *–Place enable expressions in separate process or assignment*
- *If the synthesis tool does not recognize this as an enable it will be implemented using extra LUTs*

Shift Registers

```
module shift (
    input aclr_n, enable, shiftin, clock,
    output reg [7:0] q
);
always @(posedge clock, negedge aclr_n)
    if (!aclr_n)
        q[7:0] <= 0;
    else if (enable)
        q[7:0] <= {q[6:0], shiftin};
endmodule
```

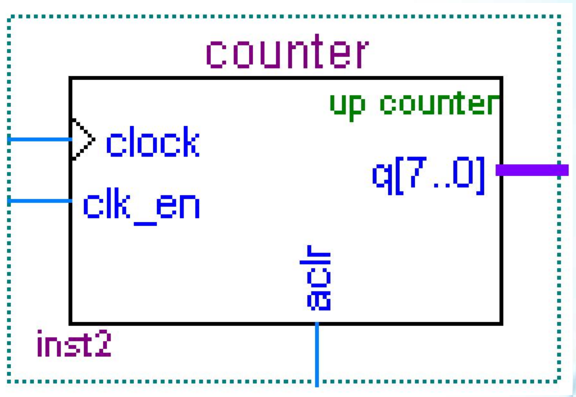


Shift function
•*Use { , } for concatenation*

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

Basic Counter w/Async Clear & Clock Enable

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

always @(posedge clk, negedge aclr_n)
        if (!aclr_n)
                q[7:0] <= 0;
        else if (ena)
                q <= q + 1;

endmodule
```



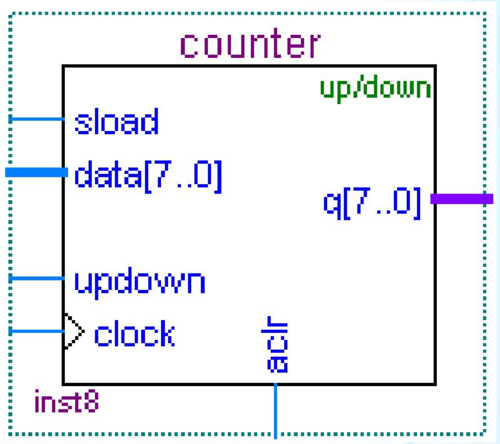
Count function

- *Binary up counter with asynchronous clear and clock enable*
- *Add or remove secondary controls similar to DFF*

Up/Down Counter w/Sync Load

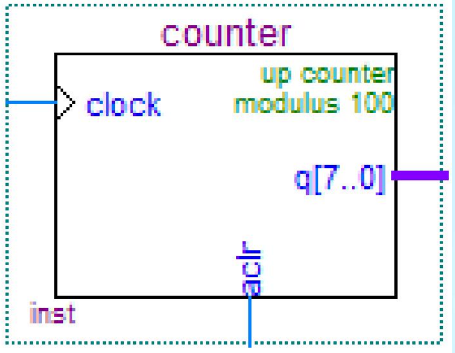
```
module count (
        input clk, aclr_n, updown, sload,
        input [7:0] data,
        output reg [7:0] q = 0
);
always @ (posedge clk, negedge aclr_n)
begin
        if (aclr_n == 0)
                q[7:0] <= 0;
        else if (sload == 1)
                q <= data;
        else begin
     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

Up/down behavioral description



Modulus-100 Up Counter

```
module count
        #(parameter modulus = 100)
        (
        input clk, aclr_n,
        output reg [7:0] q = 0
);

always @(posedge clk, negedge aclr_n)
begin
        if (aclr_n == 0)
                q[7:0] <= 0;
        else begin
                if (q == modulus – 1)
                        q <= 8’d0;
                else
                        q = q + 1;
        end
end
endmodule
```



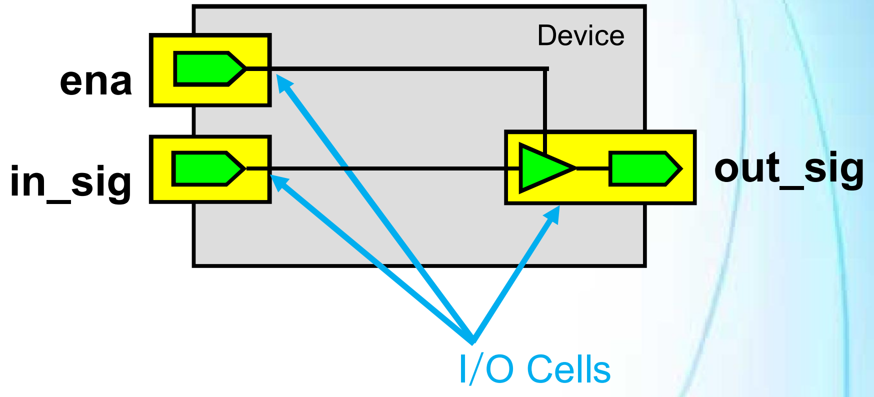
Tri-States

- IEEE-defined ‘Z’ or ‘z’ value
  - Simulation: Behaves like high-impedance state
  - Synthesis: Converted to tri-state buffers
- Altera devices have tri-state buffers only in I/O cells
  - Benefits:
    - Eliminates possible bus contention
    - Location of internal logic is a non-issue
    - Cost savings
      - Don’t pay for unused tri-state buffers
      - Less testing required of devices
  - Internal tri-states must be converted to combinatorial logic
  - Complex output enable may cause errors or inefficient logic
- The **wire** and **tri** keywords have identical syntax and function
  - Use **wire** to indicate a net with a single driver
  - Use **tri** to indicate a net that can have multiple drivers

Inferring Tri-States Correctly

Conditional Signal Assignment

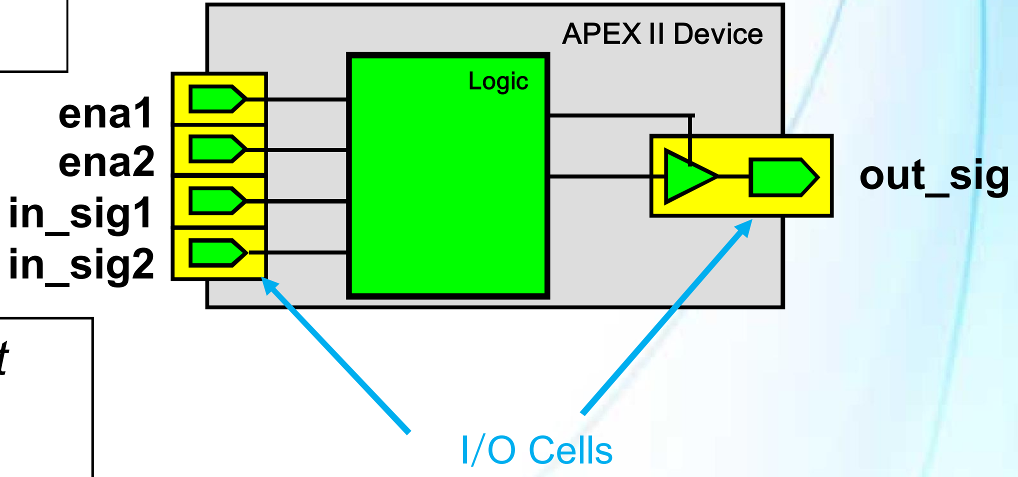
```
module dff_1(in_sig, ena, out_sig);

input in_sig, ena;
output tri out_sig;

assign out_sig = ena ? in_sig : 1'bZ;

endmodule
```



- *Only 1 Assignment to Output Variable*
- *Uses Tri-State Buffer in I/O Cell*
  - ***out_sig** should connect directly to top-level I/O pin (through hierarchy is ok)*

Inferring Tri-states Incorrectly

```
module dff_1(in_sig, ena, out_sig);

input in_sig, ena;
output tri out_sig;

assign out_sig = ena1 ? in_sig1 : 1'bZ;
assign out_sig = ena2 ? in_sig2 : 1'bZ;

endmodule
```



- *2 Assignments to Same Signal Not Allowed in Synthesis Unless 'Z" Is Used*
- *Output Enable Logic Emulated in LEs*
- *Simulation & Synthesis Do Not Match*

Bidirectional Pins

```
module module_name (
        ...,
         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*
- *For registered bidirectional I/O, use separate process to infer registers*

bidi as an input

bidi as an tri-stated output

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 (e.g. Quartus II Handbook)
- Tools may have limitations in architecture implementation
  - Newer Altera devices support synchronous inputs only
  - Limitations in clocking schemes
  - Memory size limitations
  - Read-during-write support
- Must create an array variable to hold memory values

Inferred Single-Port Memory (1)

```
module sp_ram_async_read (
     output [7:0] q,
     input [7:0] d,
     input [6:0] addr,
     input we, clk
);

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

always @(posedge clk)
     if (we)
          mem[addr] <= d;

assign q = mem[addr];

endmodule
```

Memory array

- *Code describes a **128 x 8 RAM** with <u>synchronous write</u> & <u>asynchronous read</u>*
- ***Cannot be implemented in Altera embedded RAM due to asynchronous read***
  - *Uses general logic and registers*

Inferred Single-Port Memory (2)

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

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

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

endmodule
```

- *Code describes a **128 x 8 RAM** with <u>synchronous write</u> & <u>synchronous read</u>*
- *<u>Old data</u> read-during-write behaviour*
  - *Memory read in same process/cycle as memory write using **non-blocking** statements*
  - *Check target architecture for support as unsupported features built using LUTs/registers*

Recommendation: Read Quartus II Handbook, Volume 1, Chapter 6 for more information on inferring memories and read during write behavior

Inferred Simple Dual-Port Memory

```
module sdp_sc_ram (
	output reg [7:0] q,
	input [7:0] d,
	input [6:0] wr_addr, rd_addr,
	input we, clk
);

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

always @(posedge clk) begin
	if (we)
		mem[wr_addr] <= d;
	q <= mem[rd_addr];
end

endmodule
```

- *Code describes a **simple dual-port** (separate read & write addresses) 64 x 8 RAM with single clock*
- *Code implies old data read-during-write behaviour*
  - *New data support in simple dual-port requires additional RAM bypass logic*

Inferred Dual-Port Memory

```
module dp_dc_ram (
        output reg [7:0] q_a, q_b,
        input [7:0] data_a, data_b,
        input [6:0] addr_a, addr_b,
        input clk_a, clk_b, we_a, we_b
);

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

always @(posedge clk_a)
begin
        if (we_a)
                mem[addr_a] <= data_a;
        q_a <= mem[addr_a];
end

always @(posedge clk_b)
begin
        if (we_b)
                mem[addr_b] <= data_b;
        q_b <= mem[addr_b];
end
endmodule
```

- *Code describes a **true dual-port** (two individual addresses) 64 x 8 RAM with dual clocks*
- *May not be supported in all synthesis tools*
- *Old data same-port read-during-write behaviour shown*
  - *Supported only in Stratix III, Stratix IV and Cyclone III devices*
  - *Other device families support new data (blocking assignments)*
- *Mixed port behaviour undefined with multiple clocks*

Initializing Memory Contents

```
module ram_init (
        output reg [7:0] q,
        input [7:0] d,
        input [6:0] wr_addr, rd_addr,
        input we, clk
);

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

initial
        $readmemh(“ram.dat”, mem);

always @(posedge clk) begin
        if (we)
                mem[wr_addr] <= d;
        q <= mem[rd_addr];
end

endmodule
```

- *Use $readmemb or $readmemh system tasks to assign initial contents to inferred memory*
- *Initialization data stored in .dat file converted to **.MIF** (Altera memory initialization file)*
- *Contents of .MIF downloaded into FPGA during configuration*
- *Alternate: Use an initial block and loop to assign values to array address locations*

Using System Tasks to Initialize Memory

- Each number in file given separate address
  - White spaces and comments separate numbers
- Examples
  - $readmemb ("<file_name>", <memory_name>);
  - $readmemb ("<file_name>", <memory_name>, <mem_start_addr>);
  - $readmemb ("<file_name>", <memory_name>, <mem_start_addr>, <mem_finish_addr>);
- <mem_start_addr> and <mem_finish_addr> are optional
  - Indicate beginning and ending memory addresses in which data will be loaded
    - Addresses can be ascending or descending
  - Defaults are the start index of the memory array and the end of the data file or memory array
  - Uninitialized addresses will be don't care's (X's) for synthesis and will be loaded with 0's by Quartus II software

ram.dat

```
0000_0000
0000_0101
0000_1010
0000_1111
0001_0100
0001_1001
0001_1110
0010_0011
0010_1000
// Repeat segment at
//  address 20 hex
@20
0000_0000
0000_0101
0000_1010
0000_1111
0001_0100
0001_1001
0001_1110
0010_0011
0010_1000
```

Unsupported Control Signals

- e.g. Clearing RAM contents with reset

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

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

always @(posedge clk, negedge aclr_n)
begin
        if (!aclr_n) begin
                mem[addr] <= 0;
                q <= 0;
        end
        else if (we) begin
                mem[addr] <= d;
                q <= mem[addr];
        end
end

endmodule
```

- *Memory content cannot be cleared with reset*
- *Synthesizes to logic*
- *<u>Recommendations</u>*
  1. *Avoid reset checking in RAM read or write processes*
  2. *Be wary of other control signals (i.e. clock enable) until validated with target architecture*

Inferred ROM (Case)

```
reg [6:0] q;

always @ (posedge clk)
begin
        case (addr)
                6’b000000:  q <= 8’b0111111;
                6’b000001:  q <= 8’b0011000;
                6’b000010:  q <= 8’b1101101;
                6’b000011:  q <= 8’b1111100;
                6’b000100:  q <= 8’b1011010;
                6’b000101:  q <= 8’b1110110;
                                 •••
                6’b111101:  q <= 8’b1110111;
                6’1111110:  q <= 8’b0011100;
                6’b111111:  q <= 8’b1111111;
        end case
end
```

- *Automatically converted to ROM*
  - *Tools generate ROM using embedded RAM & initialization file*
- *Requires constant explicitly defined for each choice in CASE statement*
- *May use **romstyle** synthesis attribute to control implementation*
- *Like RAMs, address or output must be registered to implement in Altera embedded RAM*

Inferred ROM (Memory File)

```
module dp_rom (
        output reg [7:0] q_a, q_b,
        input [6:0] addr_a, addr_b,
        input we, clk
);

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

initial
        $readmemh(“ram.dat”, mem);

always @ (posedge clk)
begin
        q_a <= mem[addr_a];
        q_b <= mem[addr_b];
end

endmodule
```

- *Using $readmemb or $readmemh to initialize ram contents*
- *No write control*
- *Example shows dual-port access*
- *Automatically converted to ROM*
- *Tools generate ROM using embedded RAM & initialization file*

Advanced Verilog Design Techniques

Coding State Machines

State Machine Coding

- Parameters or local 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, but not recommended

- Registers are used to store state

```
reg [2:0] current_state, next_state;
```

- Do not bit-slice current_state or next_state (e.g. state[1:0])
- Separate sequential process from combinational process
- Sequential process should always include synchronous or asynchronous reset
- Use **case** statement to do the next-state logic, instead of **if-else** statement
  - Some synthesis tools do not recognize **if-else** statements for implementing state machines

** Discussed later*

State Machine Viewer

- Use to verify correct coding of state machine

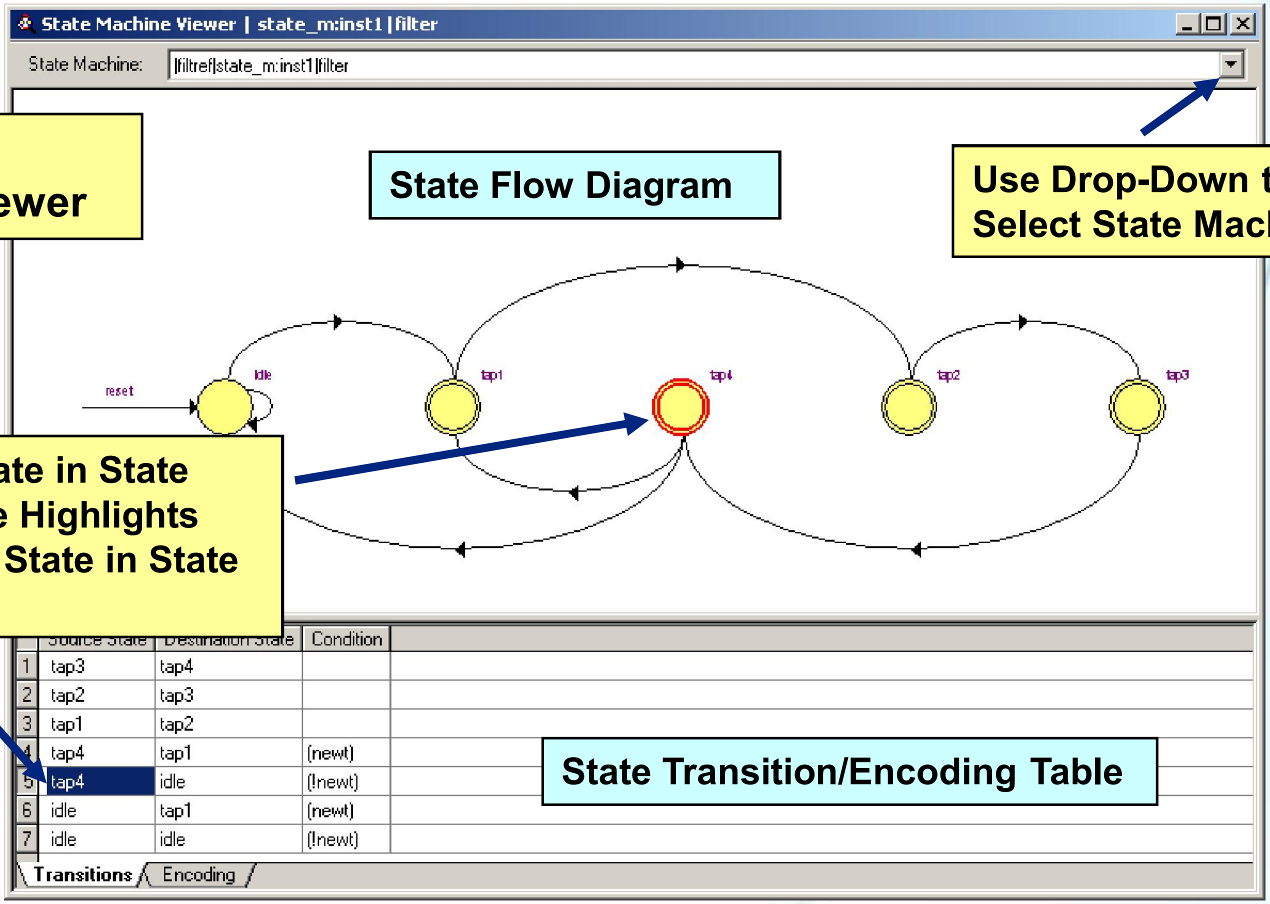
Tools Menu ⇒ State Machine Viewer

State Flow Diagram

Use Drop-Down to Select State Machine

Highlighting State in State Transition Table Highlights Corresponding State in State Flow Diagram

State Transition/Encoding Table



State Machine: |filtref|state_m:inst1|filter

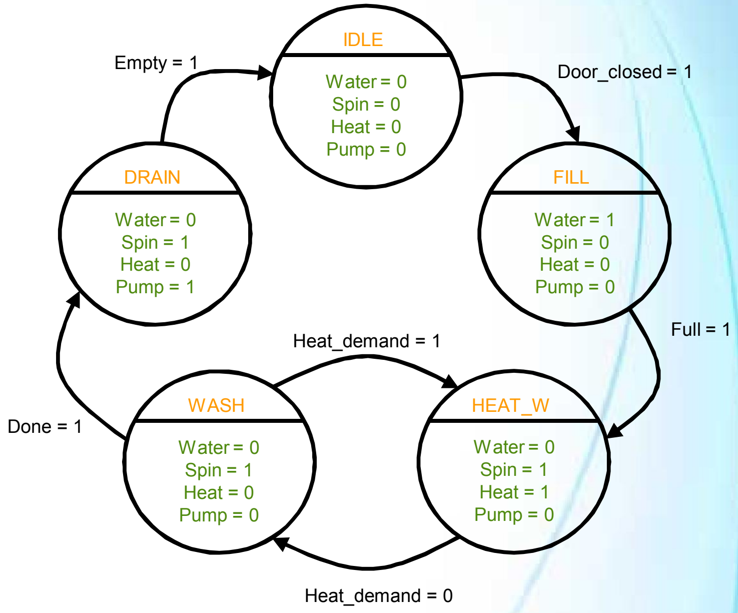
	Source State	Destination State	Condition
1	tap3	tap4	
2	tap2	tap3	
3	tap1	tap2	
4	tap4	tap1	(newt)
5	tap4	idle	(!newt)
6	idle	tap1	(newt)
7	idle	idle	(!newt)

State Declaration

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

reg [2:0] current_state, next_state;

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

States

State Registers

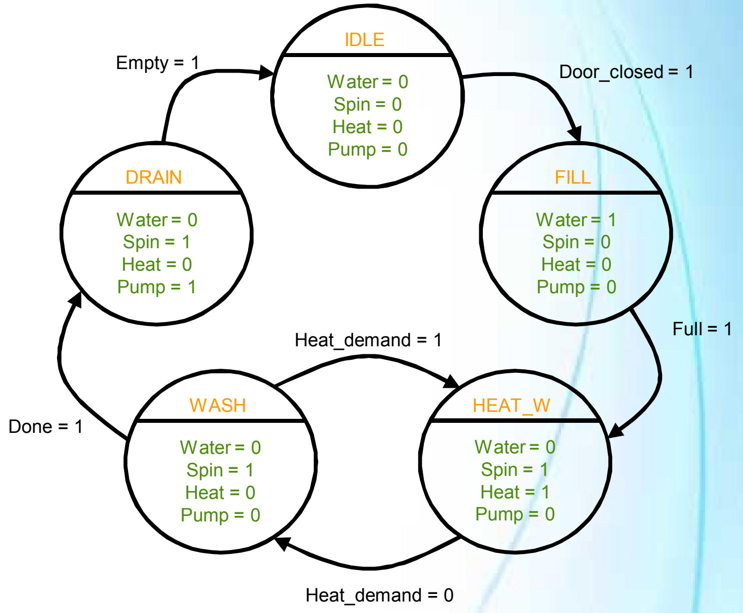


Next-State Logic

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

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

State transitions

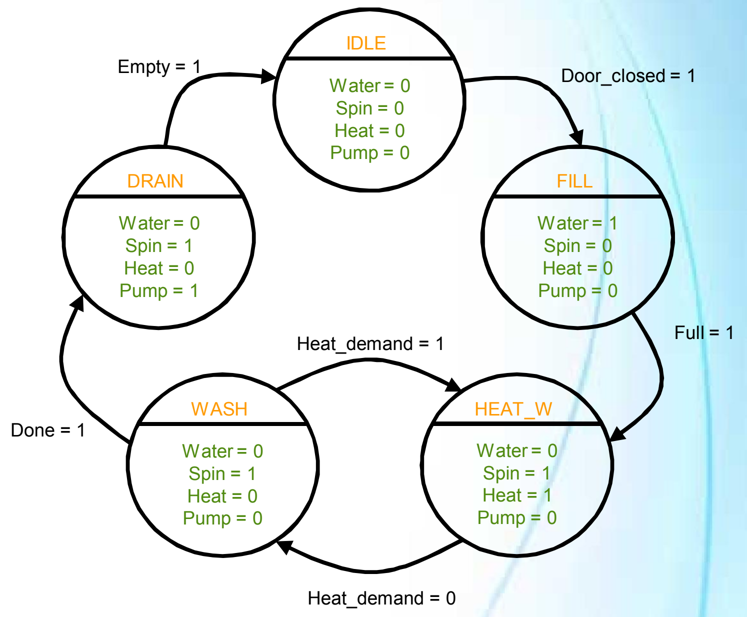
Next state logic



Moore Combinatorial Outputs

```
//output logic
always @*
begin
    water = 0;
    spin = 0;
    heat = 0;
    pump = 0;
    case (current_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
```

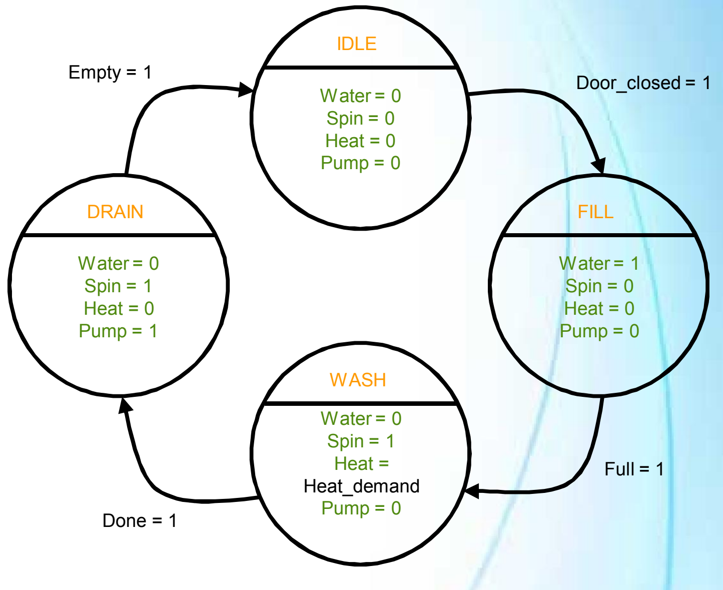
Default output conditions



– *Output logic function of state only*

Mealy Combinatorial Outputs

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



– *Output logic function of state only and input(s)*

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	?

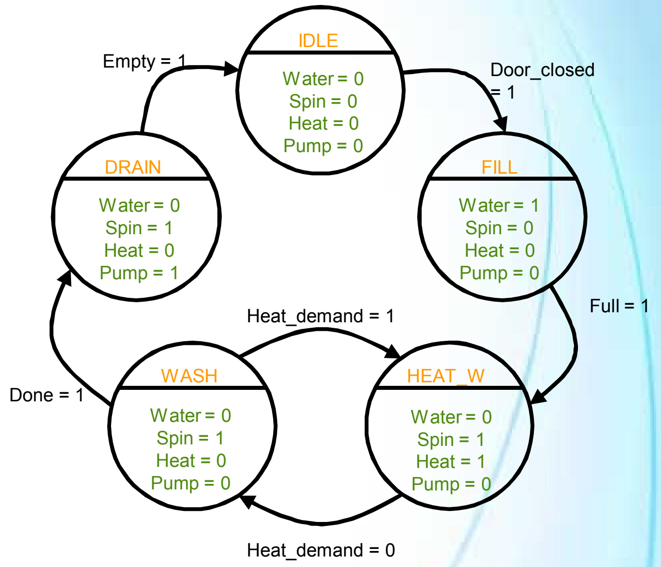
- Quartus II default encoding styles for Altera devices
  - One-hot encoding for look-up table (LUT) devices
    - Architecture features lesser fan-in per cell and an abundance of registers
  - Binary (minimal bit) or grey-code encoding for product-term devices
    - Architecture features fewer registers and greater fan-in

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
- To account for undefined states
  - Explicitly code for them (manual)
  - Use “safe” synthesis constraint (automatic)

'Safe' State Machine ??

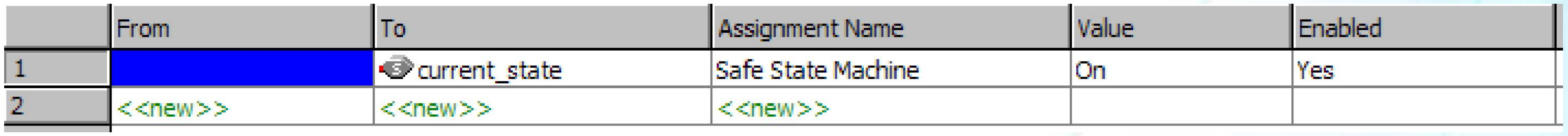
```
//State transitions
always @(posedge clk)
    current_state <= next_state;

//Next State logic
always @ *
begin
    next_state = current_state;   //default condition
    case (current_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*

Creating "Safe" State Machines

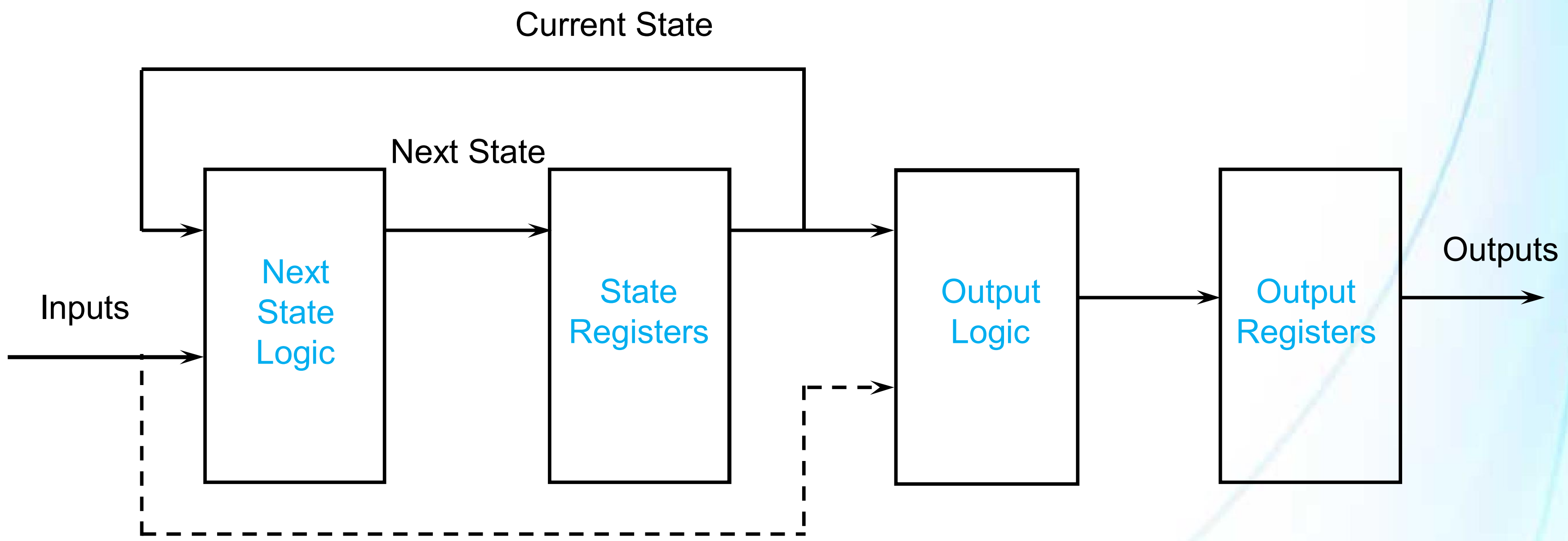
- Default clause does not make state machines "safe"
  - Once state machine is recognized, synthesis tool only accounts for explicitly defined states
  - Exception: Number of states equals power of 2 **AND** binary/grey encoding enabled
- Safe state machines created using synthesis constraints
  - Quartus II software uses
    - SAFE STATE MACHINE assignment applied project-wide and to individual FSMs
    - Verilog synthesis attribute
- May increase logic usage



	From	To	Assignment Name	Value	Enabled
1		current_state	Safe State Machine	On	Yes
2	<<new>>	<<new>>	<<new>>		

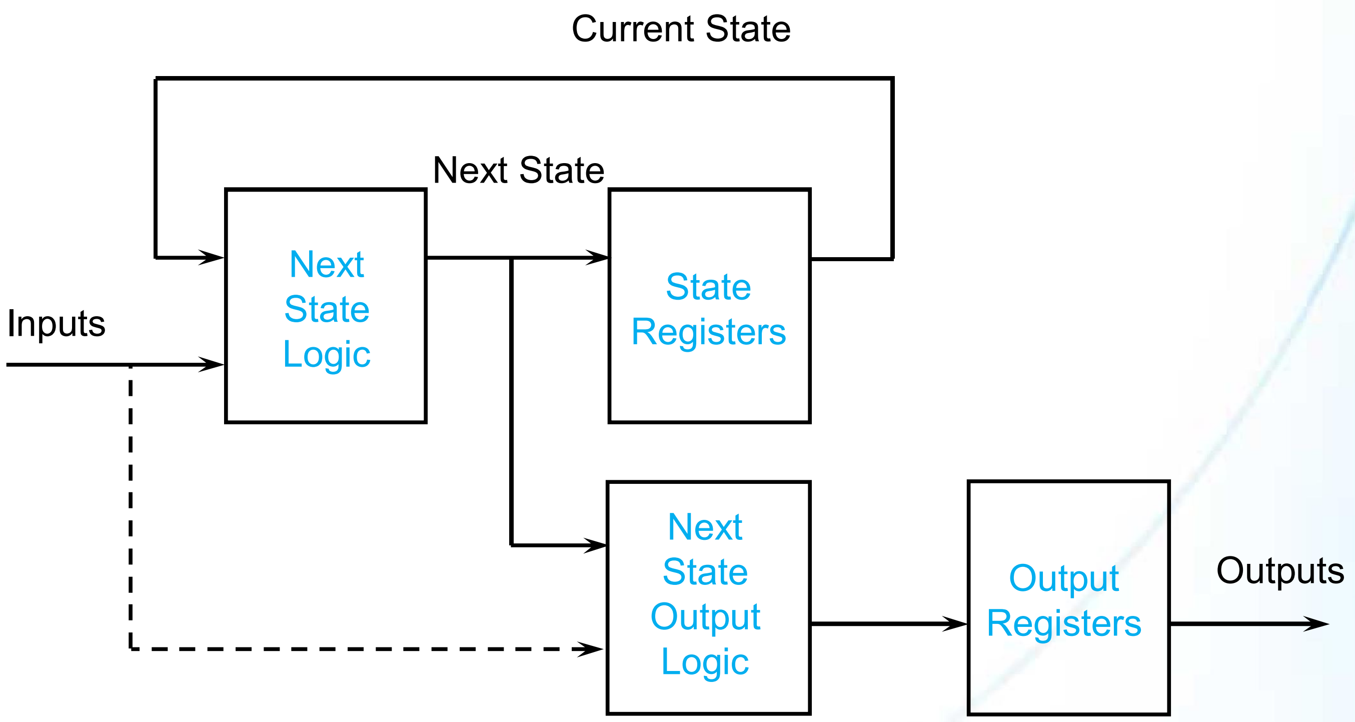
Registered Outputs

- Remove glitches by adding output registers
  - Adds a stage of latency



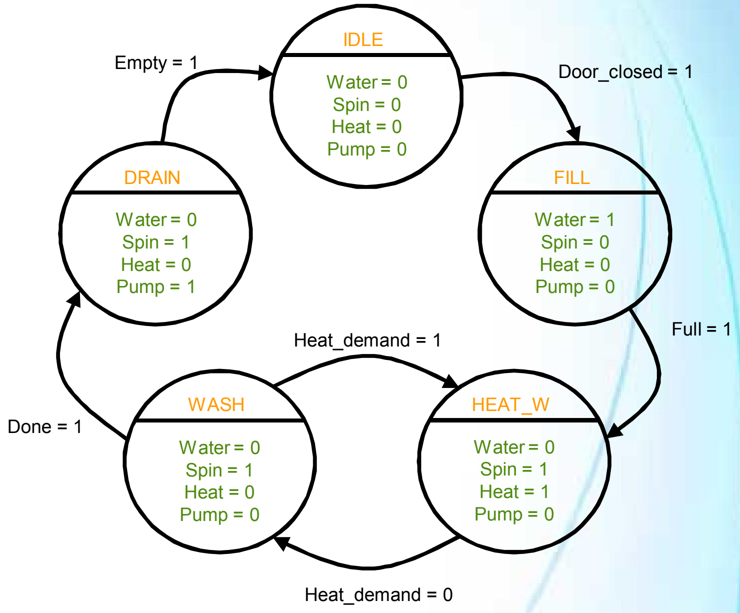
Registered Outputs w/o Latency

- Base outputs on next state vs. current state
  - Output logic uses next state to determine what the next outputs will be
  - On next rising edge, outputs change along with state registers



Registered Outputs w/o Latency

```
//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
```



- *Base output logic case statement on next state variable (instead of current state variable)*
- *Wrap output logic with a clocked process*

Using Custom Encoding Styles

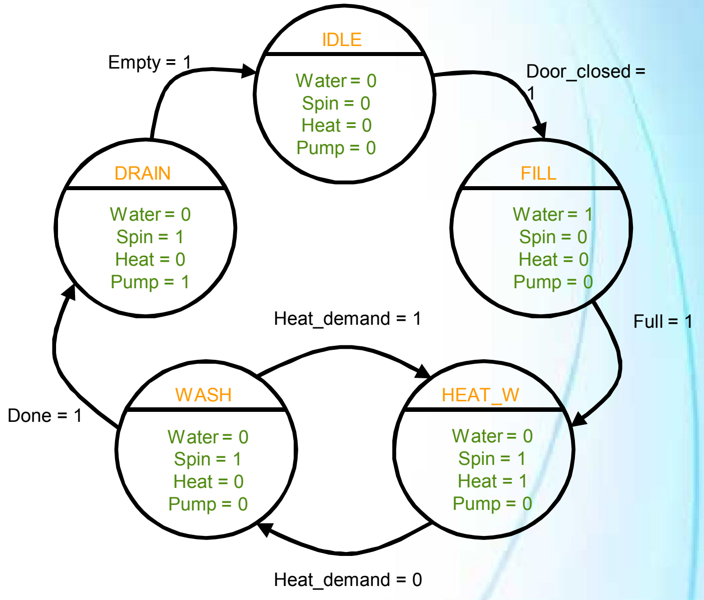
- Remove glitches without output registers
- Eliminate combinatorial output logic
- Outputs mimic state bits
  - Use additional state bits for states that do have exclusive outputs

State	Outputs Water Spin Heat Pump	Custom Encoding
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] current_state, next_state;

parameter idle = 'b0000, fill = 'b1000,
        heat_w = 'b0110, wash = 'b0100,
        drain = 'b0101;
```

Custom Encoding



- *Must also set **State Machine Processing** assignment to **"User Encoded"***
- *<u>Output assignments are coded per previous examples</u> (slides 83 or 84)*
  - *Synthesis automatically handles reduction of output logic*
- *Some tools use synthesis attributes like **enum_encoding** OR **syn_enum_encoding** to perform custom state encoding*

Writing Efficient State Machines

- Remove counting, timing, arithmetic functions from state machine & implement externally
- Reduces overall logic & improves performance

Advanced Verilog Design Techniques

Improving Logic Utillization & Performance

Improving Logic Utilization & Performance

- Balancing Operators
- Resource Sharing
- Logic Duplication
- Pipelining

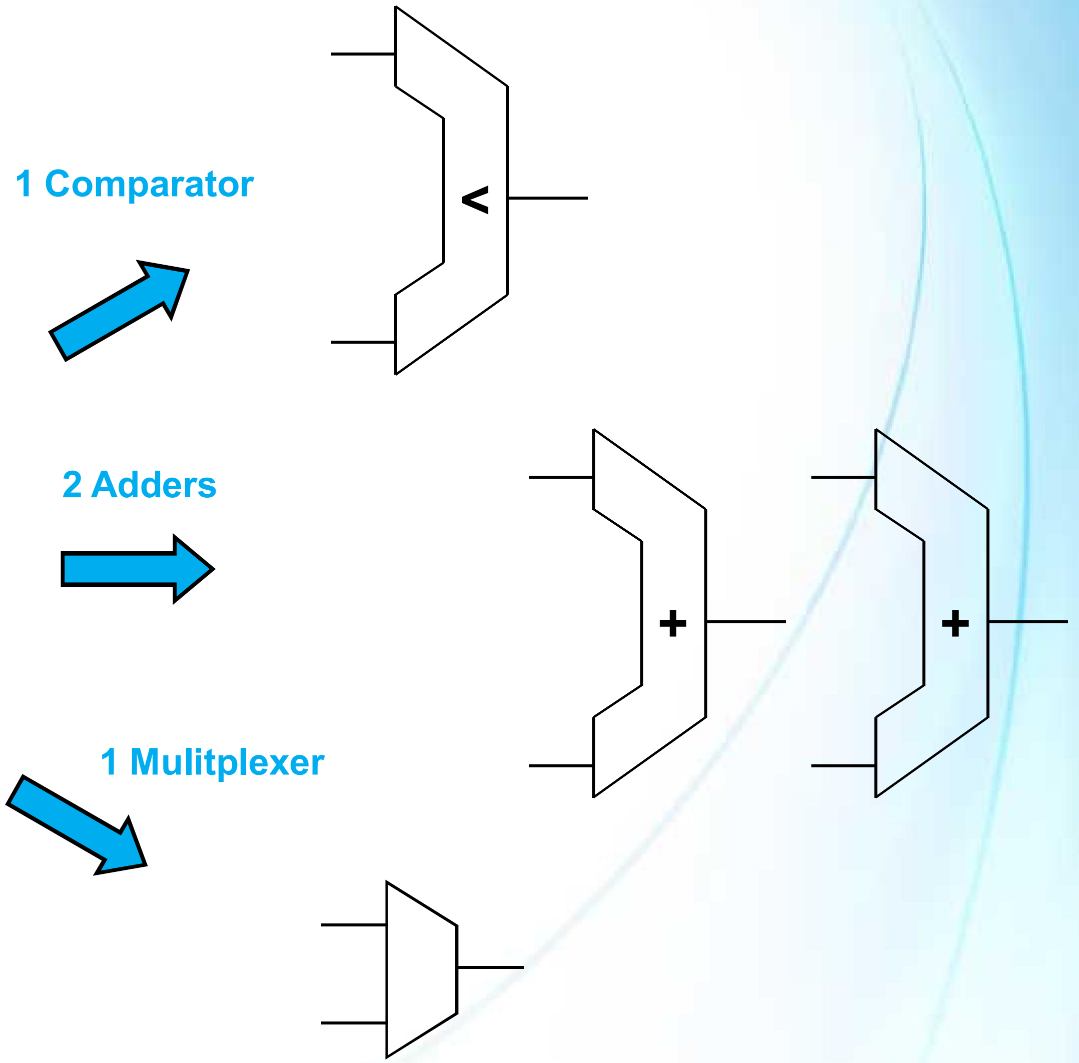
Operators

- Synthesis tools replace operators with pre-defined (pre-optimized) blocks of logic
- Designer should control when & how many operators
  - Ex. Dividers
    - Dividers are large blocks of logic
    - Every '/' and '%' inserts a divider block and leaves it up to synthesis tool to optimize
    - Better resource optimization usually involves cleverly using multipliers or shift operations to do divide

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;
```

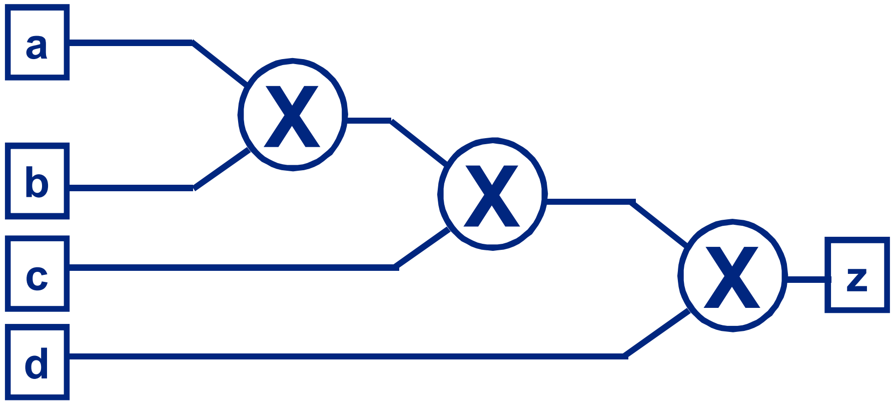


Balancing Operators

- Use parenthesis to define logic groupings
  - Increases performance & utilization
  - Balances delay from all inputs to output
  - Circuit functionality unchanged

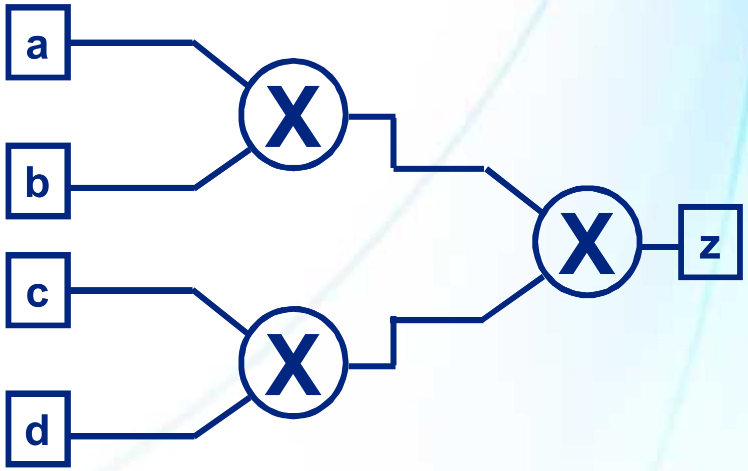
Unbalanced

out = a * b * c * d;



Balanced

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

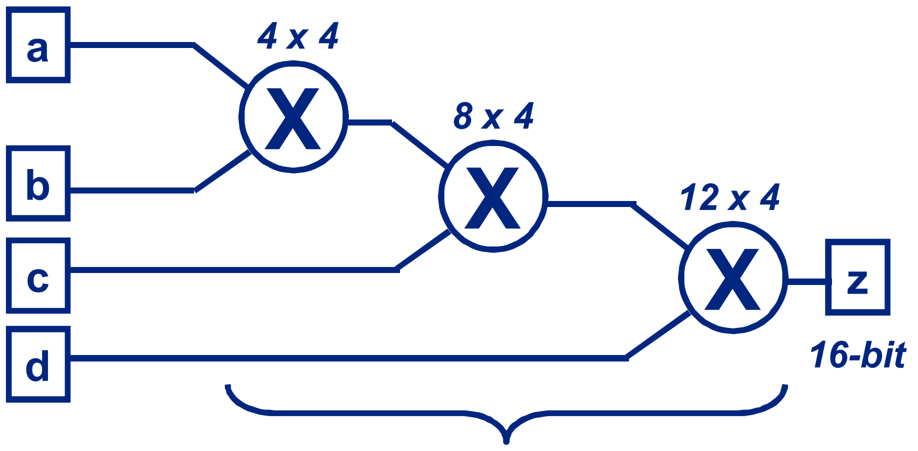


Balancing Operators: Example

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

Unbalanced

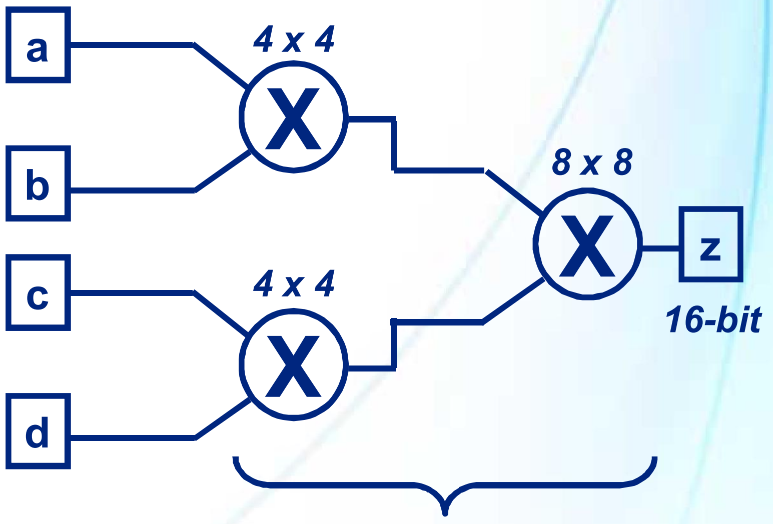
out = a * b * c * d



Delay through 3 Stages of Multiply

Balanced

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



Delay through 2 Stages of Multiply

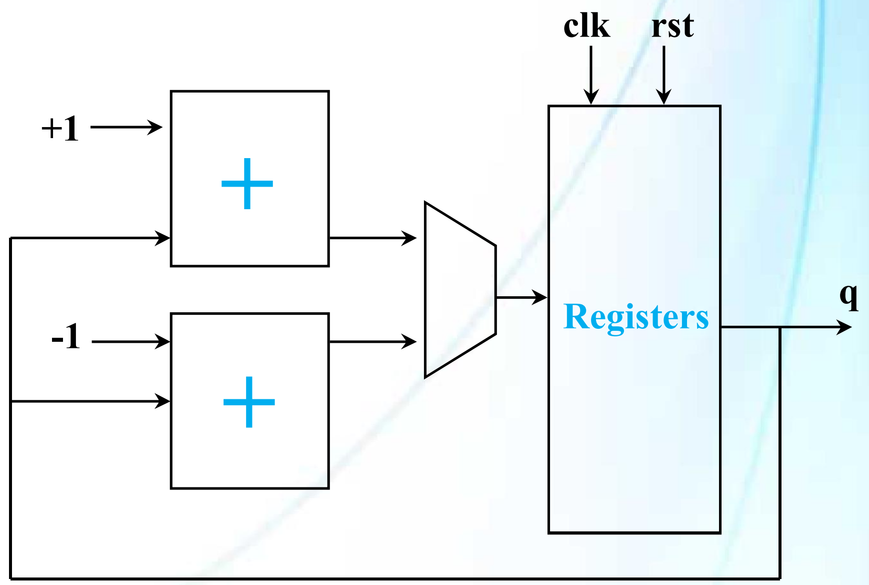
Resource Sharing

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

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

- *Up/down counter*
- *2 adders are mutually exclusive & can be shared (typically if-else with same operator in both choices)*

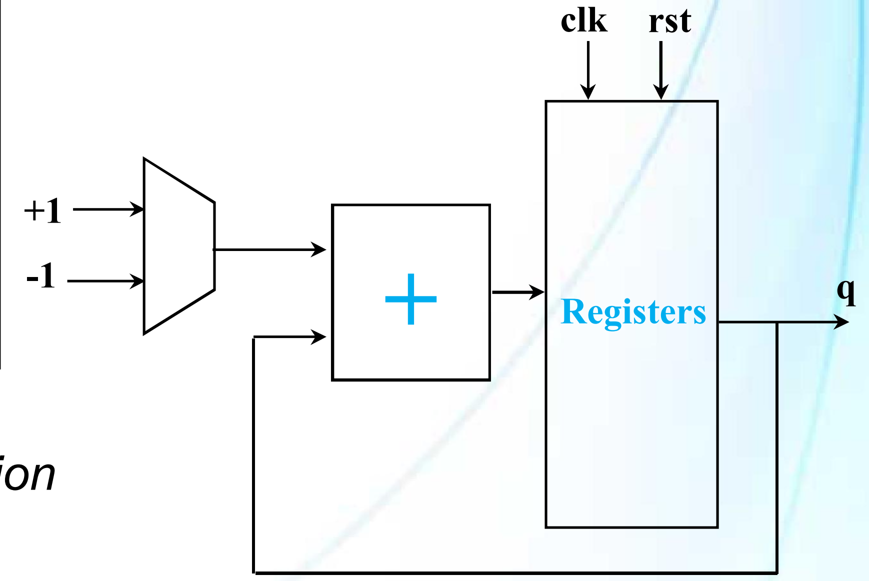


Sharing Mutually Exclusive Operators

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

always @ (posedge clk, 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



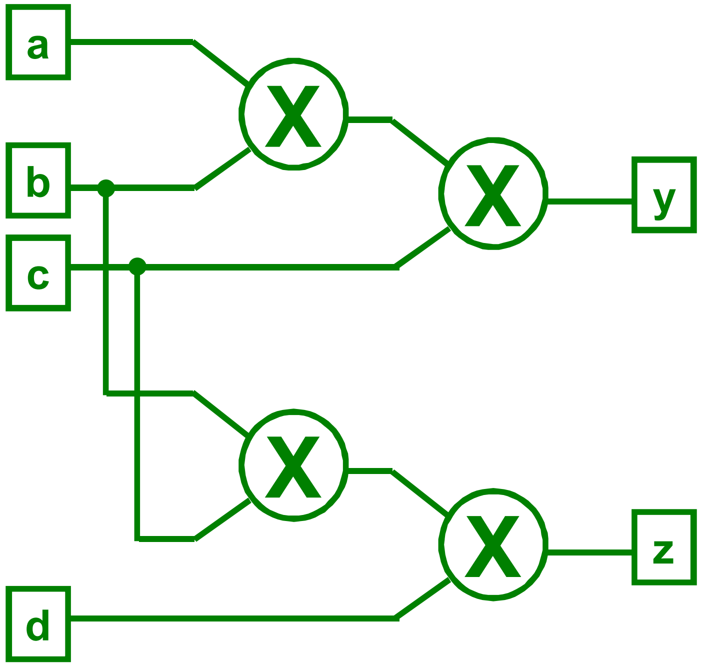
How Many Multipliers?

$$y = a * b * c$$
$$z = b * c * d$$

How Many Multipliers? (Answer)

y = a * b * c

z = b * c * d



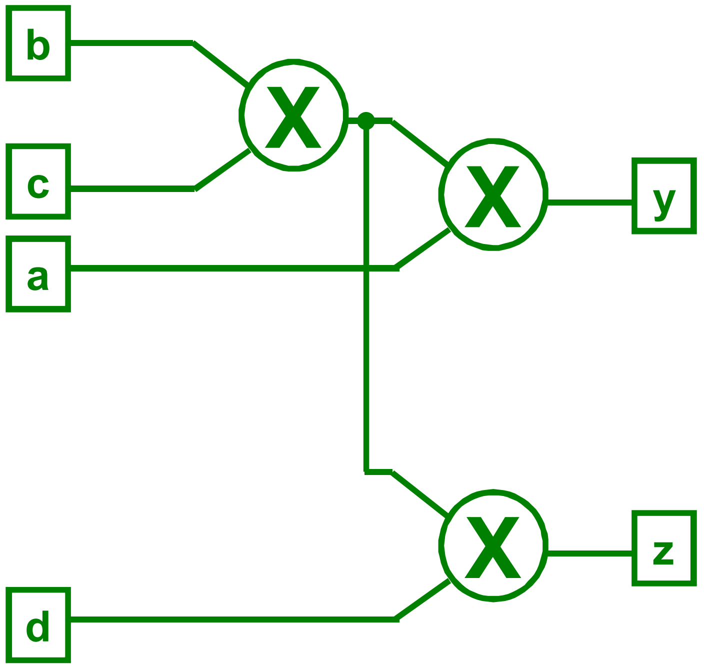
4 Multipliers!

How Many Multipliers Again?

$$y = a * (b * c)$$
$$z = (b * c) * d$$

How Many Multipliers Again? (Answer)

y = a * (b * c)
z = (b * c) * d



3 Multipliers!

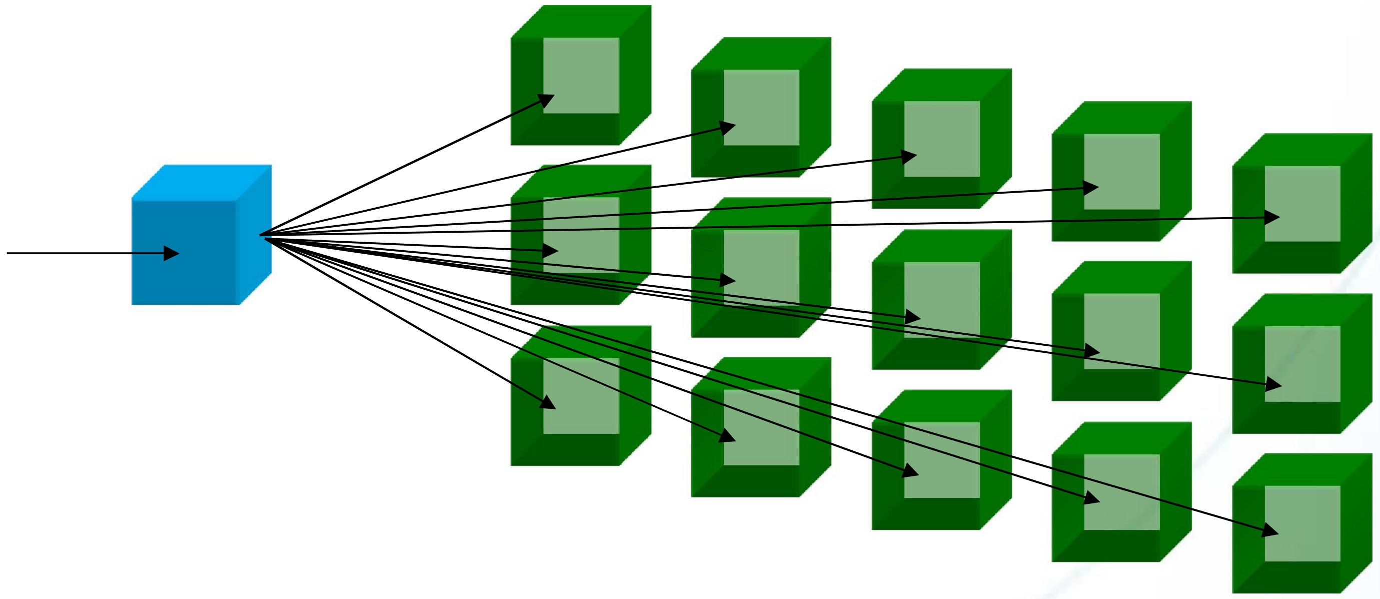
- *This is called <u>sharing common subexpressions</u>*
- *Some synthesis tools do this automatically, but some don't!*
- *Parentheses guide synthesis tools*
- *If (b*c) is used repeatedly, assign to temporary signal*

Logic Duplication

- Intentional duplication of logic to reduce fan-out
- Synthesis tools can perform automatically
  - User sets maximum fan-out of a node

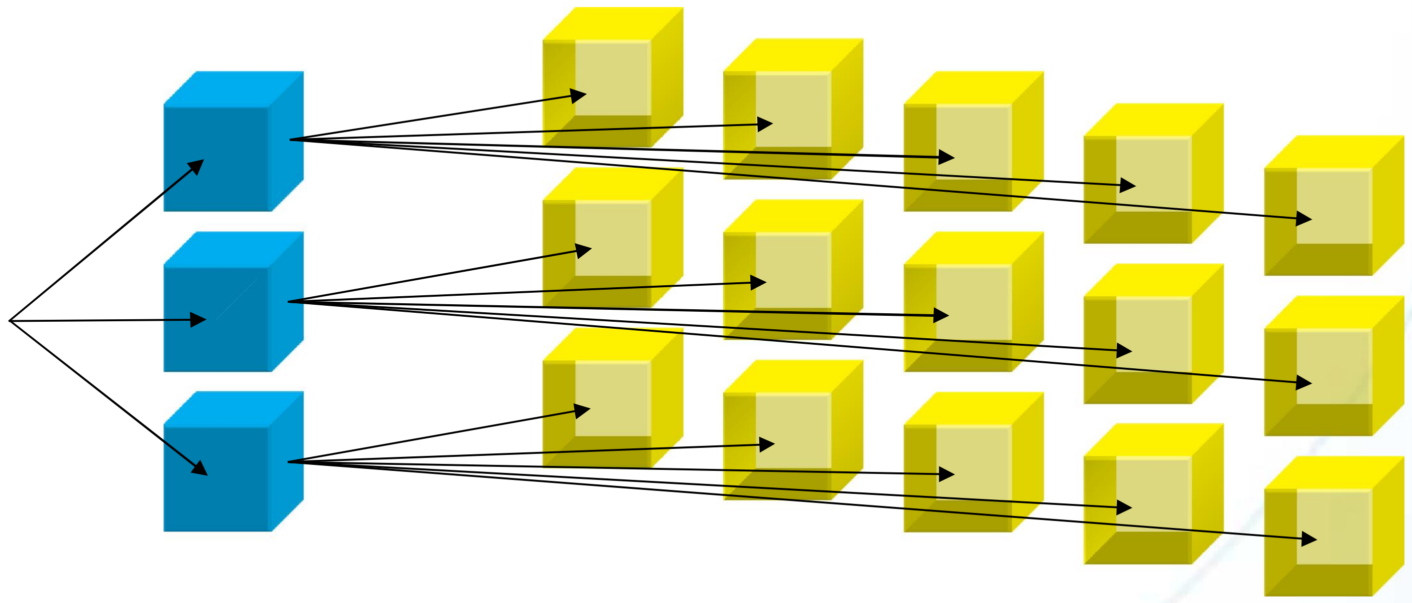
Fan-Out Problems

- High fan-out increases placement difficulty
  - High fan-out node cannot be placed close to all destinations
  - Ex: Fan-out of 1 & 15



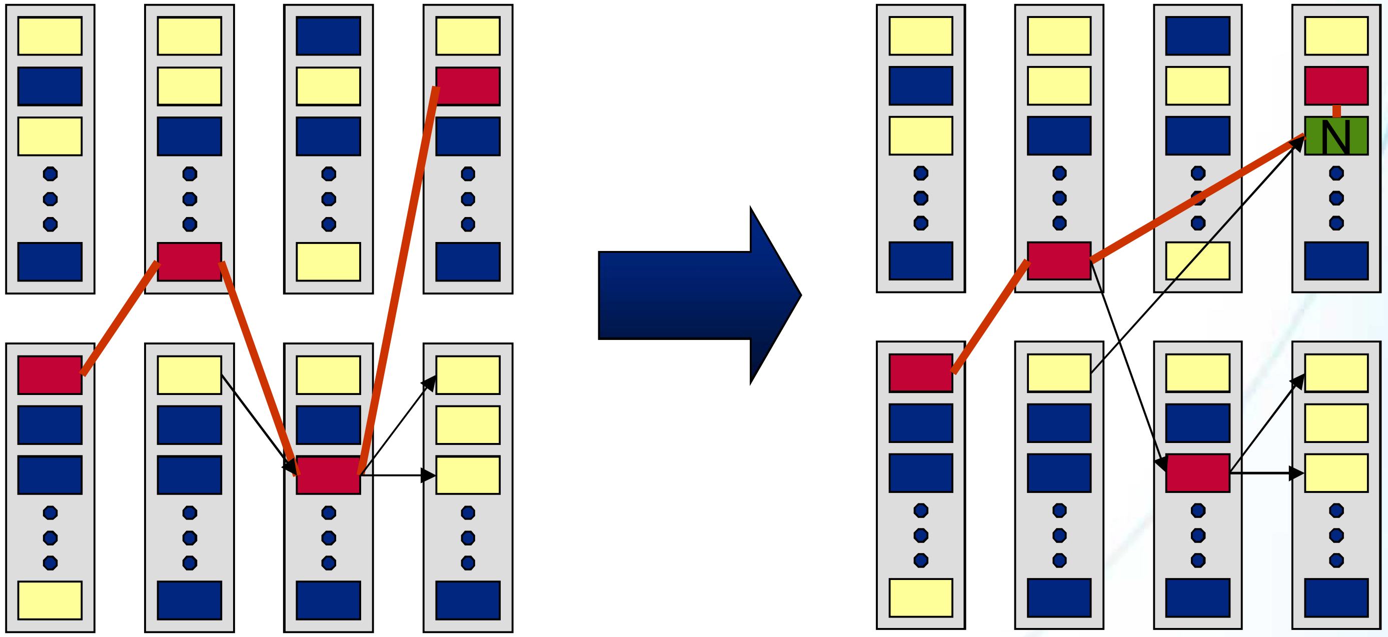
Controlling Fan-Out

- By replicating logic fan-out can be reduced
  - Worst case path now contains fan-out 3 and 5



Logic Duplication Example

- High fan-out node duplicated & placed to reduce delay

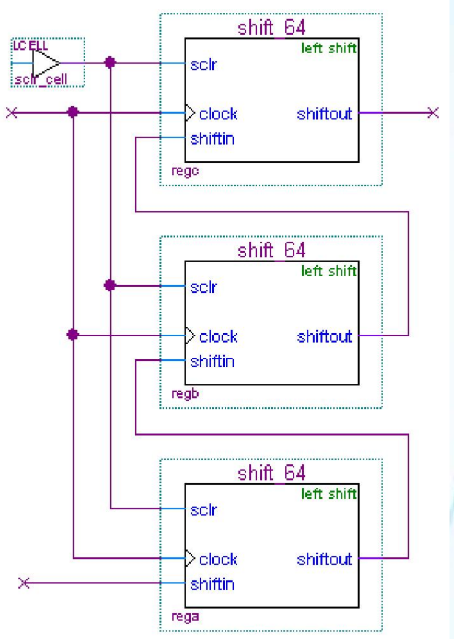


Automatic Fan-Out Control

- Most synthesis tools feature options which limit fan-out
- Advantage: Easy experimentation
- Disadvantage: Less control over results
  - Knowing which nodes have high fan-out & their destination helps floor-planning

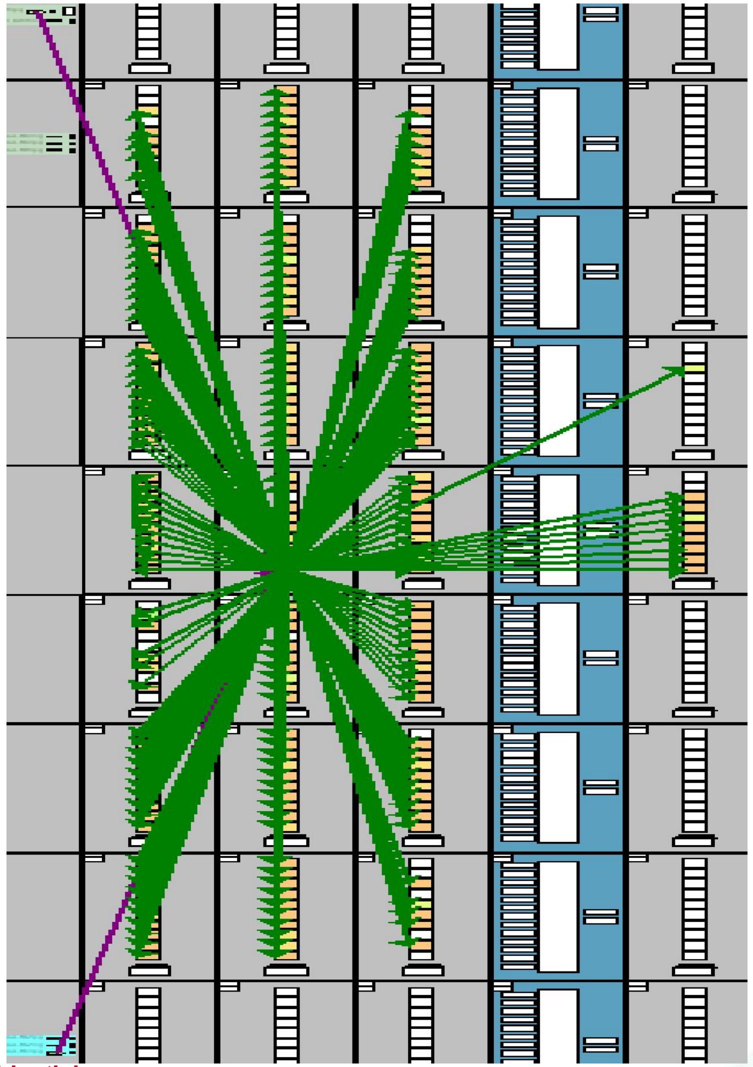
Shift Register example

```
module test (
    input clk, sclr_in, shiftin,
    output shiftout
);
reg [63:0] rega, regb, regc;
reg sclr;
always @(posedge clk)
    sclr <= sclr_in;
always @(posedge clk) begin
    if (sclr)
        regc <= 0;
    else
        regc <= {regc[62:0], regb[63]};
    if (sclr)
        regb <= 0;
    else
        regb <= {regb[62:0], rega[63]};
    if (sclr)
        rega <= 0;
    else
        rega <= {rega[62:0], shiftin};
end
assign shiftout = regc[63];
endmodule
```



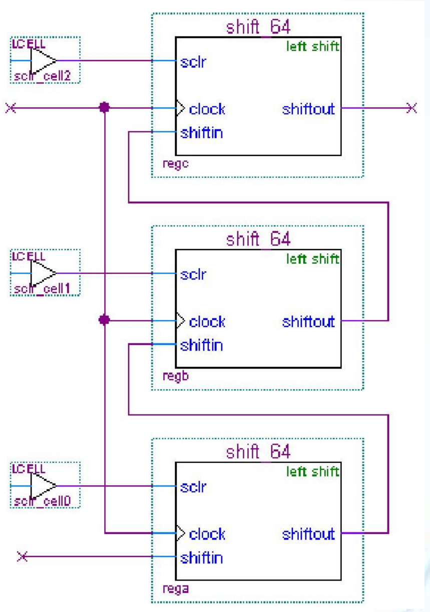
- *sclr fans out to each DFF within 3 64 bit shift registers*
- *The shift registers are cascaded to produce one 192 bit shift register*
- *sclr provides a synchronous clear function*

Fan-Out to 192 Registers



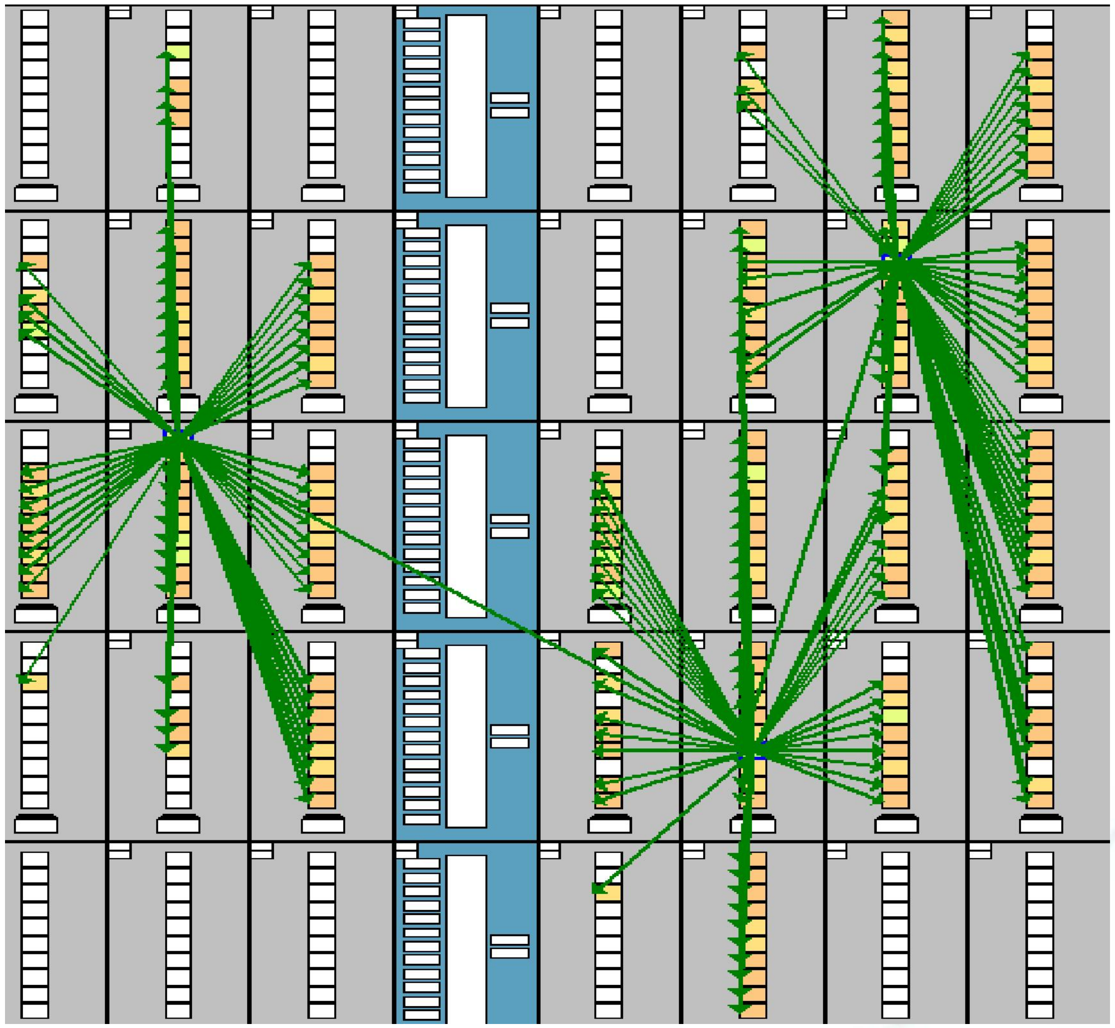
Shift Register with Reduced Fan-Out

```
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*
- *Fan-out from the previous cell has gone from 1 to 3 but this is insignificant*

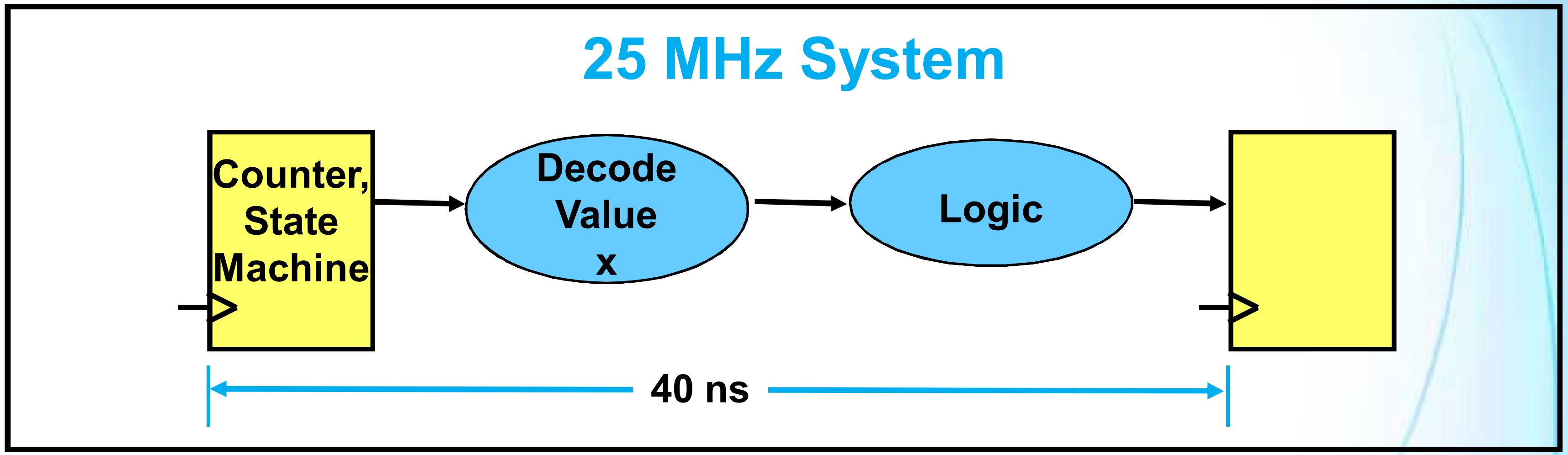
Fan-Out to 64 Registers

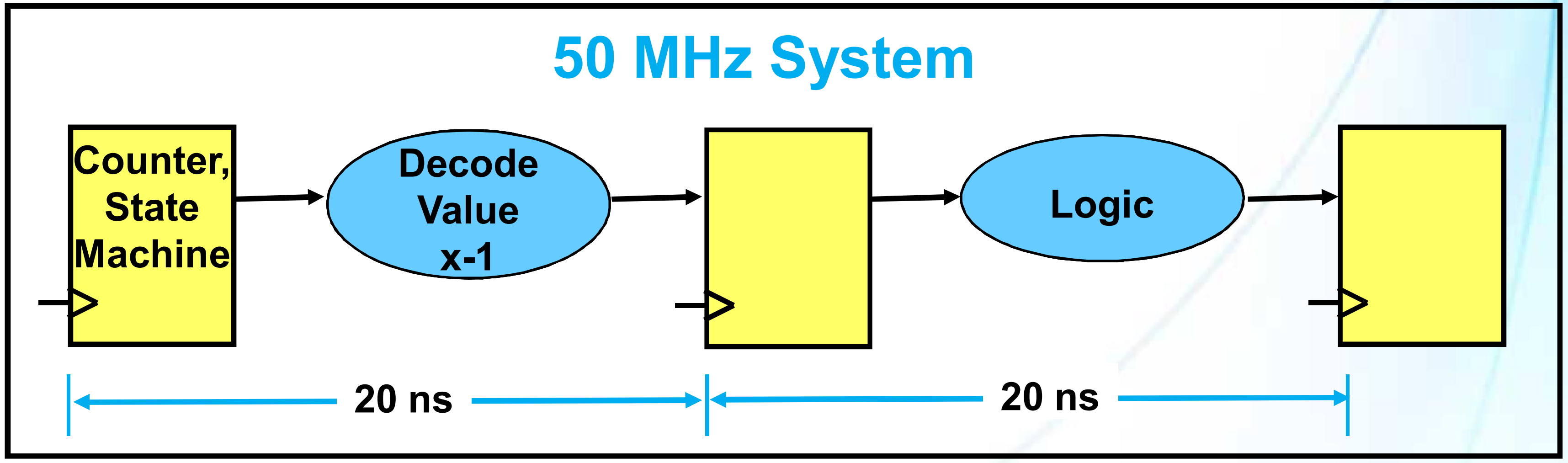


Pipelining

- 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





Adding Single Pipeline Stage in Verilog

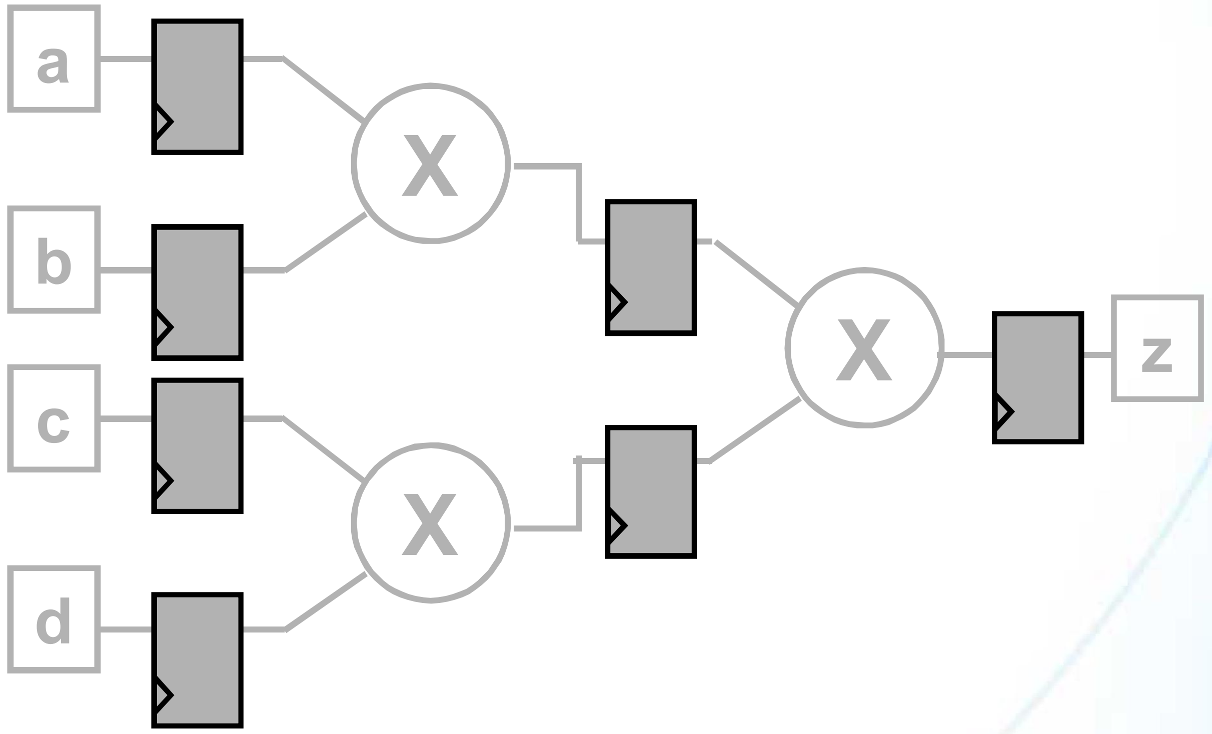
Non-Pipelined

```
module test (
      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, 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 (
      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, 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



Advanced Verilog Design Techniques

Introduction to Testbenches

Purpose of Testbench

- Generate stimulus to test design for normal transactions, corner cases and error conditions
  - Direct tests
  - Random tests
- Automatically verify design to spec and log all errors
  - Regression tests
- Log transactions in a readable format for easy debugging

Three Classes of Traditional Testbenches

I. Test bench applies stimulus to target code and outputs are manually reviewed

II. Test bench applies stimulus to target code and verifies outputs functionally
   - Requires static timing analysis

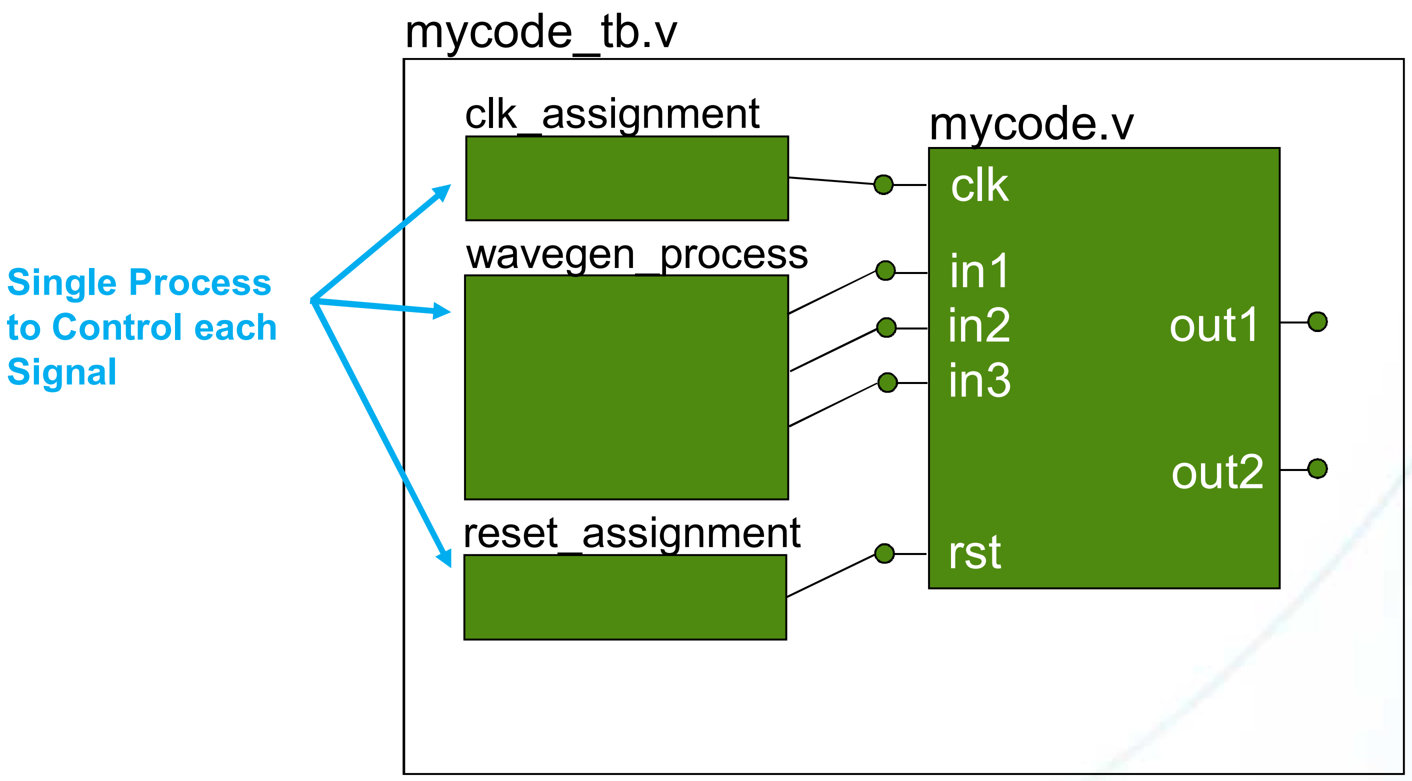
III. Test bench applies stimulus to target code and verifies outputs with timing
   - Does not require full static timing analysis
   - Code and test bench data more complex
   - Not covered

Advantages/Disadvantages

Testbench Type	Advantages	Disadvantages	Recommendation
Class I	• Simple to write	• Requires manual verification • Takes longer for others (not original designer) to verify • Easy for others to miss errors	• Great for verifying simple code • Not intended for re-use
Class II	• Easy to perform verification once complete • “Set and forget it”	• Takes longer to write • More difficult to debug initially	• Better for more complicated designs, designs with complicated stimulus/outputs and higher-level designs • Promotes re-usability
Class III	• Most in-depth • “Guarantees” design operation, if successful (subject to model accuracy)	• Takes longest to write • Most difficult to debug • Physical changes (i.e. target device, process) requires changing testbench	• Might be overkill for many FPGA designs • Required for non-Altera ASIC designs

General Class I Methods

- Create "test harness" code to instantiate the device under test (DUT) or target code
- Create stimulus signals to connect to DUT

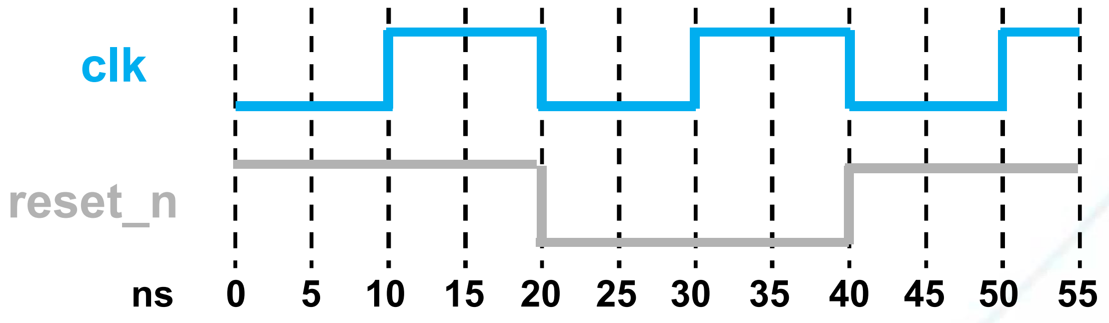


Test Vector Generation

- Develop sequence of fixed input values
- Test vector development from bottom up
  - Write basic tasks
  - Write more complex tasks based on basic tasks
  - Perform tests
- Example – memory testing
  - Basic tasks: readmem, writemem
  - 2nd level tasks: initmem, copymem, comparemem
  - Generation of tests based on tasks

Concurrent Statements

- Signals with regular or limited transitions can be created with separate ***initial*** blocks (concurrent statements)
- These statements can begin a testbench and reside outside any other processes



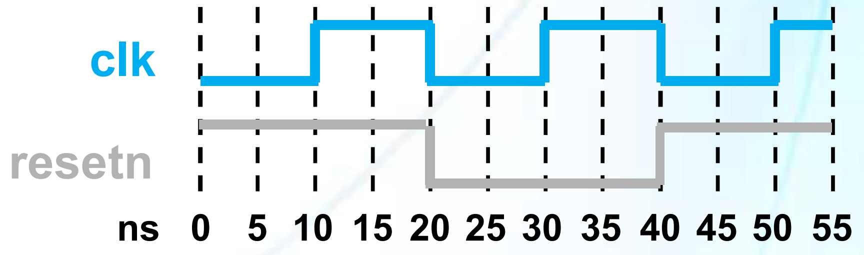
Example Initial Blocks

```
`timescale 1ns / 1ns
`define CLKPERIOD 20
module tb();

reg clk;
reg resetn;

initial // Clock procedural block
begin
   clk = 0;
   forever clk = #(`CLKPERIOD/2) ~clk;
end

initial  // Reset procedural block
begin
   resetn = 1;
   #20 resetn = 0;
   #20 resetn = 1;
end

initial #2500 $stop;

endmodule
```

- *Use **initial** blocks to define*
- *Initial block tells the simulator to run the code at time zero*
- *Code continues running until all statements finish*

Clock Process

Reset Process



Creating Periodic Signals

```
`timescale 1ns / 1ns
`define CLKPERIOD 30
module count_gen_tb();

reg clk;
reg [7:0] bus, count;

initial clk=0;

always #(`CLKPERIOD/2) clk=(clk !== 1?1:0);

initial
begin: count_gen
        count = 0;
        bus = 0;
        forever    begin
                repeat (2) @(posedge clk) ;
                bus = count;
                count = count + 1;
        end
end

initial #2500 disable count_gen;

endmodule
```

- Use separate initial or always blocks to create more periodic stimulus
  - Initial blocks more common

- ***Initial*** *and* ***always*** *blocks to define free-running clock*
- *Second* ***initial*** *block (count_gen) with forever loop to input counting pattern (every other clock edge)*
- *Third* ***initial*** *block uses* ***disable*** *statement to turn off counting pattern after 2500 ns*

Working with Delay

```
`timescale 1ns / 1ns
`define CLKPERIOD 30
module tb();

reg clk;
reg reset;

initial
begin
   clk = 0;
   forever clk = #(`CLKPERIOD/2) ~clk;
end

initial
begin
   reset = 1;
   #20 reset = 0;
   #20 reset = 1;
end

initial #2500 $stop;

endmodule
```

Define the time scale & precision

Define a clock period

- Defining a timescale
  - Choose the largest precision that can accurately model the system
    - Too small of a precision can needlessly increase memory usage and simulation time
  - Timescale passed to all modules without defined timescales that are subsequently elaborated
    - Simulator warning
- Defining clock period(s)
  - Place in top level test bench module or in a “definitions” header file that is included from top
- Use blocking assignments and regular assignment delay for specifying stimulus
  - Simulate faster and use less memory than non-blocking
  - Non-blocking assignments with intra-assignment delay can be used to model delay lines

Simple Verilog Class I Testbench

```
`timescale 1 ns/1 ns
module addtest();

parameter CLKPERIOD = 20;
parameter PERIOD = 60;
reg [3:0] a,b;
wire [3:0] sum;
reg clk;

adder add1(.clk(clk), .a(a), .b(b), .sum(sum));

initial clk = 1'b0;

always #(CLKPERIOD/2) clk=(clk !== 1?1:0);

initial
begin: adder_stim
        @ (negedge clk) ;
        a = 3'b0;       b = 3'b0;
        #PERIOD ;
        forever begin
                @ (negedge clk) ;
                a = a + 3'd2;
                b = b + 3'd3;
                #PERIOD ;
        end
end

initial #1000 $stop;

endmodule
```

Top-level entity has no ports

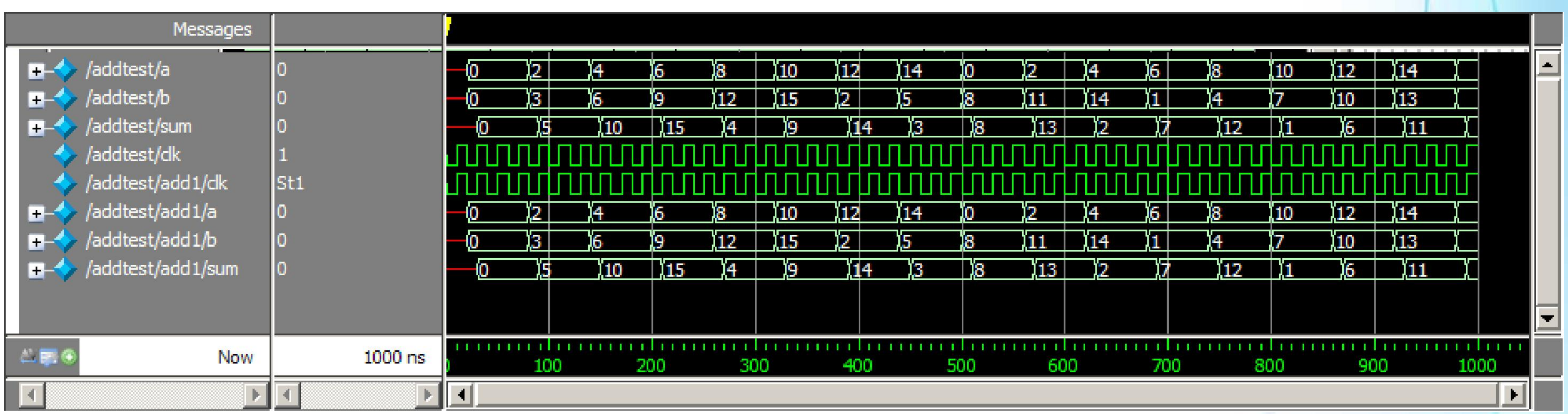
Signals to assign values & observe

Instantiate lower-level entity

Create clock to synchronize actions

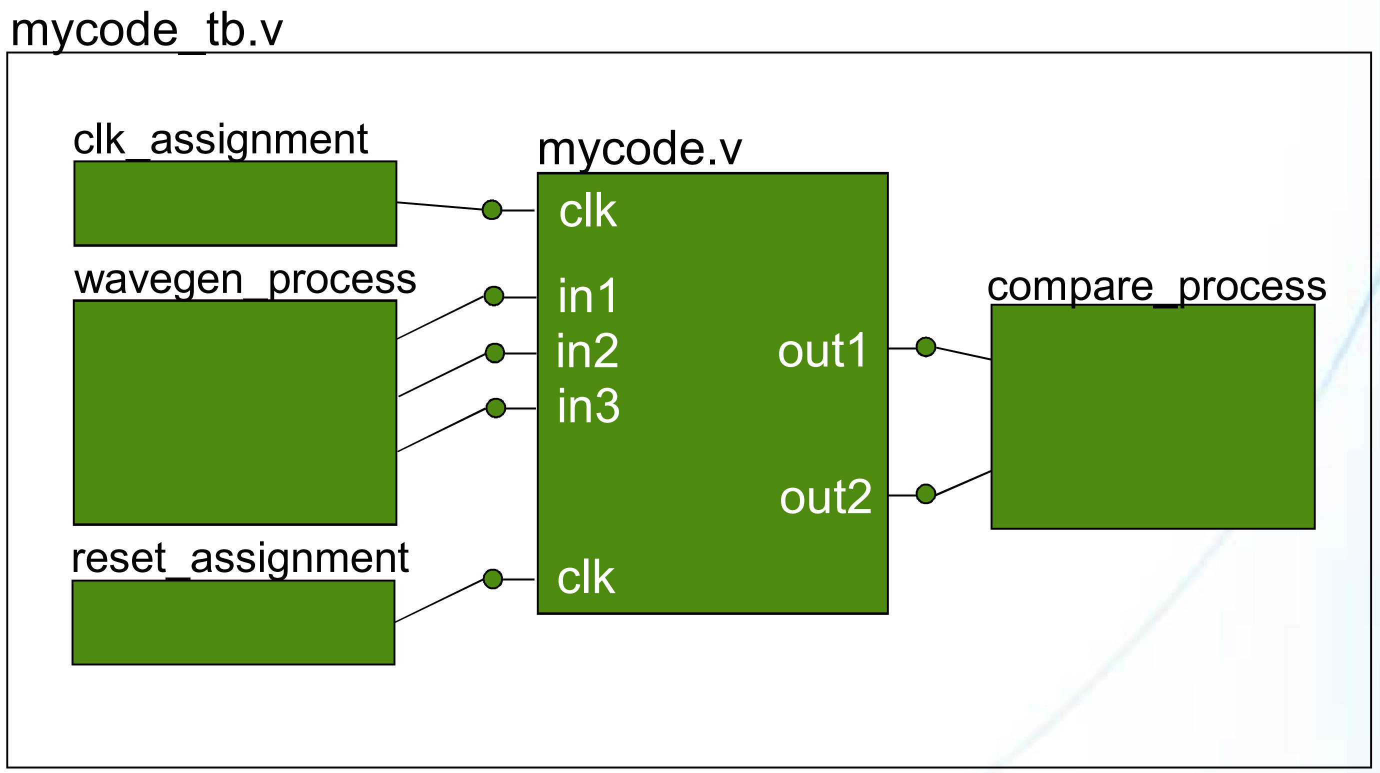
Apply stimulus; Note input data changing only on inactive clock edge

Example Results


```
Messages
/addtest/a           0
/addtest/b           0
/addtest/sum         0
/addtest/clk         1
/addtest/add1/clk    St1
/addtest/add1/a      0
/addtest/add1/b      0
/addtest/add1/sum    0
Now                  1000 ns
```

Class II (& III) Methods

- Add a compare process to an existing design so that outputs can be monitored



Self Verification Methods

- May use a “compare process” to check received results against expected results
- Single simulation can use one or multiple testbench files
  - Single testbench file containing all stimulus and all expected results
  - Multiple testbench files based on stimulus, expected results or functionality (e.g. data generator, control stimulus)
- Many times signaling is too complicated to model without using vectors saved in “time-slices”

Simple Self Verifying Test Benches

```
adder add1(.clk(clk), .a(a), .b(b), .sum(sum));

initial begin
    @ (negedge clk) ;
    a = 0; b = 0;
    #40 if (sum !== 0) begin
        $display("Sum is wrong");
        $display("Expected 0, but received %d",
                                 sum);
        $finish;
    end

    @ (negedge clk) ;
    a = a + 3'd2; b = b + 3'd3;
    #40 if (sum !== 5) begin
        $display("Sum is wrong");
        $display("Expected 5, but received %d",
                                 sum);
        $finish;
    end

    // Repeat above varying values of a and b

    end
end
```

- **Code repeated for each test case**
- **Result checked**

- *Simple self verifying test bench*
- *Each block checks for correct answer*
  - *Minimizes human error*
- *Code not very efficient*
  - *Each test case requires a lot of repeated code*
- *Improve this code by introducing a task*

Simplify Test Bench with Task

```
adder add1(.clk(clk), .a(a), .b(b), .sum(sum));
initial begin
    test(0, 0, 0);
    test(2, 3, 5);
    test(4, 6, 10);
    test(6, 9, 15);
    test(8, 12, 4);
    test(10, 15, 9);
    $finish ;
end
task test;
    input [3:0] in_a, in_b, exp_result;
    begin
        @(negedge clk) ;
        a = in_a;
        b = in_b;
        #40 if( sum !== exp_result) begin
            $display(“Result is wrong");
            $display(“Expected %d, but received %d”,
                exp_result, sum);
            $finish;
        end
    end
endtask
```

- Task used to simplify test bench

- *Task improves efficiency and readability of testbench*
- *Advantage: Easy to write*
- *Disadvantages*
  - *Each task execution (like last example) assigns values to a, b then waits to compare sum to its predetermined result*
  - *Very difficult to do for complicated signaling*

Storing Stimulus/Results in “Time Slices”

- Write stimulus/results into text files
  - e.g. Store one time slice per line (all inputs or expected outputs separated by spaces)
- Read stimulus and expected results into Verilog memory arrays inside test bench
  - Use the *$readmemh* or *$readmemb* system tasks to read external files into a Verilog arrays (discussed earlier)

Example Test Bench Using Text Files & Memories

```
module addtest_mem_tb;
reg [3:0] input_mem [0:13];  // Memory to hold input stimulus
reg [3:0] exp_mem [0:6]; // Memory to hold expected results
reg [3:0] a, b, exp_result;
wire [3:0] sum;
reg clk;
integer i;

initial begin  //Initialization block
    clk = 1'b0;
    $readmemb("init.dat", input_mem); // Read input stimulus into memory
    $readmemb("exp.dat", exp_mem);  // Read expected output results into memory
end

always #10 clk=(clk !== 1?1:0);  //Define system clock

//Instantiate the device under test (dut)
adder add1 (.clk(clk), .a(a), .b(b), .sum(sum));

initial begin
    for (i=0; i< 7; i = i+1) begin // Loop through memory values
        @(negedge clk) ;  // Drive inputs on inactive clock edge

        a = input_mem[2*i];   // Read 2 values from input stimulus memory to
        b = input_mem[(2*i)+1];   //   be driven into dut

        exp_result = exp_mem[i];  // Read 1 value from expected outputs memory

        @(negedge clk) ; //perform checking on next falling edge clock
        if (exp_result !== sum)    // Compare dut result against expected result
            $display("%0d : Calculated %0d, Expected %0d", //   & print to display
                        $time, sum, exp_result);
  end
end
endmodule
```

- This example uses 1 data file/memory for input stimulus & one data file/memory for expected results
- For loop used to loop through all stimulus and expected results memories
  - Be careful of stimulus files being shorter than memories
- Note that more complex operation would require more complicated stimulus (e.g. output qualifier signal, input stimulus and output checking in separate procedural blocks, separate memories per data input)

Example Input Memory Files

init.dat

```
// a_in        b_in
@000
   0000        0000
@004
   0010        0011
   0100        0110
   0110        1001
   1000        1100
   1010        1111
```

exp.dat

```
0000
XXXX
0101
1010
1111
0100
1001
```

– *Each value separated by white space read into separate memory address*
– *Both $readmemb and $readmemh treat all types of white spaces (e.g. spaces, new line characters, tabs) identically*

Example Test Plan

- Develop high-level behavioral (i.e. non-synthesizable) model of design
- Create stimulus/test vectors to simulate model
- Generate expected results from behavioral model simulation
- Replace behavioral blocks with RTL model blocks
  - Simulate each RTL block with other behavioral blocks to ensure functionality is the same

Advanced Verilog Design Techniques

Parameterized Code

Parameterized Code

- Writing code to enable reuse and flexibility
  - Logic blocks that are made scalable
  - Different configurations of same model
- 2 Verilog constructs support parameterization
  - Parameters
  - Constant function
  - Generate constructs
    - If
    - Case
    - Loop

Parameters (Review)

- Module constants used to pass information to an instance
  - Scalable code
  - Changing module behavior
- Must resolve to a constant at compile time

```
module  reg_bank

                 // Parameter defaults to 1, but can be overridden
        (

        );
        •
        •
        •
endmodule
```

Parameterized Counter

```
module counter is

        (


        );
        always @ (posedge clk or posedge clr)
                if (clr)
                        q <= 0;
                else if (sload)
                        q <= data;
                else if (cnt_en)
                        q <= q + 1;
endmodule
```

Parameter WIDTH used to scale counter

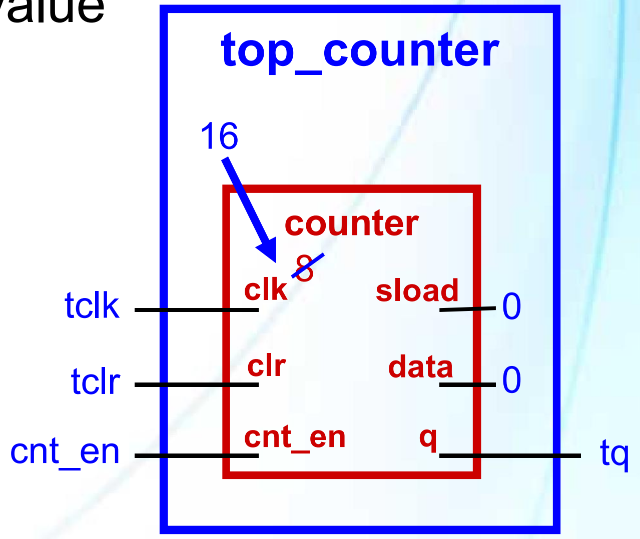
– *Parameter set to default value to make optional*

Instantiating a Parameterized Module

- Must map to parameters & ports in instantiation
- Generic & port resolution done at compile time
- Default parameters overridden if explicitly defined
  - Specified 16 overrides default 8
  - No need to specify a parameter in instantiation if using its default
- Unconnected outputs leave empty <.q ()> or omit completely
- Set unused inputs to correct active/inactive value

```
counter # (.WIDTH(16))
        u1 (.clk(tclk), .clr(tclr),
                .cnt_en(tcnt_en),
                .sload(1'b0),
                .data(16'b0),
                .q(tq));
```

Inactive value for sload



Complete Code

```
module top_counter
        (
        input tclk, tclr, tcnt_en,
        output reg [15:0] tq
        );

        counter #(.WIDTH(16))
                u1 (.clk(tclk), .clr(tclr),
                        .cnt_en(tcnt_en),
                        .sload(1'b0),
                        .data(16'b0),
                        .q(tq));
endmodule
```

– *This instantiation method for parameters introduced in Verilog-2001 and is the most explicit and easy to read*
  - *–Other methods supported*
  - *–May require explicitly defining all parameters even if defaults are used*
  - *–Using defparams not recommended*

Local Parameters

- Verilog-2001 enhancement
- Parameter that cannot be redefined during instantiation
  - Base local parameters on parameters
  - Protects parameters from accidental redefinition
- Cannot be used in module definition

```
module memory
        #(parameter D_WIDTH = 8,
                A_WIDTH = 4)
        (
        input clk, we,
        input [A_WIDTH-1:0] addr,
        input [D_WIDTH-1:0] data,
        output reg [D_WIDTH-1:0] q
        );
        localparam M_DEPTH = 2**A_WIDTH;
        reg [D_WIDTH-1:0] mem [0:M_DEPTH-1];
        always @ (posedge clk)
        begin
                if (we)
                        mem[addr] <= data;
                q <= mem[addr];
        end
endmodule
```

Constant Functions

```
module ram_const_func
        #(parameter D_WIDTH = 8, RAM_DEPTH = 2048)
        (
        output reg [D_WIDTH-1:0] q,
        input [D_WIDTH-1:0] d,
        input [clogb2(RAM_DEPTH)-1:0] addr,
        input we, clk
);

function integer clogb2 (
        input integer mem_depth
);
        begin
                for (clogb2 = 0; mem_depth > 0;
                                        clogb2 = clogb2 + 1)
                        mem_depth = mem_depth >> 1;
        end
endfunction

reg [7:0] mem [0:RAM_DEPTH];

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

endmodule
```

Constant Function call used to determine address width

Constant function ***clogb2*** *definition*

- Functions whose result evaluates to a constant at elaboration time
- Adds more complexity & readability to calculations used to determine vector widths and array sizes
- Arguments must be constants at elaboration time (e.g. parameters)
- Added in Verilog-2001
  - Previously only constant expressions based on arithmetic operators supported for object definitions

Generate Constructs

- Used to create structural blocks
  - Each module generated creates a level of hierarchy
    - Similar to performing separate module/primitive instantiations
    - Reduces amount of code
  - Resolved at elaboration (compile) time
    - Must resolve to a constant number of blocks
  - Use **generate-endgenerate** (generate regions) to explicitly define generate region[1]
    - Cannot nest generate regions

[1] *Generate regions are consdered optional by the Verilog specification, but are required by the Quartus II software*

Generate Constructs (cont.)

- Generate blocks can contain
  - Module declarations
  - Module/primitive instantiations
  - Parameter overrides
  - Assign statements
  - Procedural blocks
- Generate blocks cannot contain
  - Port declarations
  - Parameter declarations

Generate Constructs Types

- if-generate
  - Conditionally selects <u>whether</u> structure is made or not
- case-generate
  - Conditionally selects <u>which</u> structure from multiple choices is made
- Loop-generate
  - Creates zero or a set <u>number of duplicates</u> of a structure
  - No need to individual instantiate each duplicate

- Can be nested for further complexity

if-generate Construct

- Syntax

```
generate
    if <expression>
        // generated structure
    else
        // alternate generate structure
endgenerate
```

- Condition controls whether structure is created
  - Same syntax as procedural block **if-else**
  - Each structure may contain only one item
    - Surround multiple statements with begin/end

if-generate Example 1

```
module counter
        #(parameter WIDTH = 8, RISE_OR_FALL = 1)
        (
        input clk, clr, sload, cnt_en,
        input [WIDTH-1:0] data,
        output reg [WIDTH-1:0] q
        );

        wire clk_buf;
        generate
        //  Rising or falling  clock generation logic
                if (RISE_OR_FALL)
                        assign clk_buf = clk;
                else
                        assign clk_buf = ~clk;
        endgenerate

always @ (posedge clk_buf or negedge clr)
        begin
                if (!clr)
                        q <= 0;
                else if (sload)
                        q <= data;
                else if (cnt_en)
                        q <= q + 1'b1;
        end
endmodule
```

rising or falling edge clock

- *Simple code slice to implement both rising & falling edge counter*
  - *Generate block with ASSIGN statements*
- *Different than using if-else*
  - *No mux (or gated) clock created*

if-generate Example 2

```
generate
    if (PIPELINE = “YES”)
    begin: pipelined_mult
        mult_clk #(.AWIDTH(D_WIDTHA), .BWIDTH(D_WIDTHB))
            u1 (.dataa(dataa), .datab(datab), .aclr(aclr),
            .clken(clken), .clock(clock), .result(result));
    end

    if (PIPELINE = “NO)
    begin: combinational_mult
        comb_mult #(.AWIDTH(D_WIDTHA), .BWIDTH(D_WIDTHB))
            u1 (.dataa(dataa), .datab(datab), result(result));
    end
endgenerate
```

- *This code uses a text string (PIPELINE) to call a pipelined or non-pipelined implementation of a multipler block*
  - *Generate block with module instantiations*
- *One or the other (or neither) gets instantiated in design*
- *Begin/end unnecessary since one statement in each, but needed to name generate block*
  - *Block name not required*

case-generate Construct

- Syntax

```
generate
    case <expression>
        <condition1> : // generated structure
        <condition2> : // generated structure
        <conditionN> : // generated structure
        default : // generated structures
endgenerate
```

- Condition controls which structure is created
  - Same syntax as procedural block **case**
  - Each structure may contain only one item
    - Surround multiple statements with begin/end

case-generate example

```
module data_conversion
        #(parameter DATA_PATH = 8)
        (
        input clk, reset, clk_en,
        input [DATA_PATH-1:0] data,
        output reg [DATA_PATH-1:0] result
        );

        generate  // Data conversion of bytes received serially
                case (DATA_PATH)
                        8 : byte_convert u1 (.datain(data), .dataout(result),
                                        .clk(clk), .clken(clken), .clr(reset));
                        16 : hword_convert u1 (.datain(data), .dataout(result),
                                        .clk(clk), .clken(clken), .clr(reset));
                        32 : word_convert u1 (.datain(data), .dataout(result),
                                        .clk(clk), .clken(clken), .clr(reset));
                endcase
        endgenerate
endmodule
```

– *Case statement performs test to see which of the byte conversion blocks get instantiated in the design*
– *Notice all 3 possible instances named "u1"*
  –*Legal as only one will exist in compiled design*

Loop-generate (for-generate)

- Syntax

```
genvar i; // Required
generate
    for (i=<init_val>; <exp using i>; <inc using i>)
    begin : <generate_block_name>  // Required
        // generate structure
    end
endgenerate
```

- Sets the number of structures created
  - Similar to procedural block for loop
  - Can only use concurrent statements
- Generate block must be named for loop-generate
- Requires **genvar** declaration
  - Index used as integer count during loops
    - Does not exist once code is elaborated
  - One **genvar** cannot be shared by multiple loop constructs
    - Use multiple **genvar** variables for multiple loops

For Generate Example

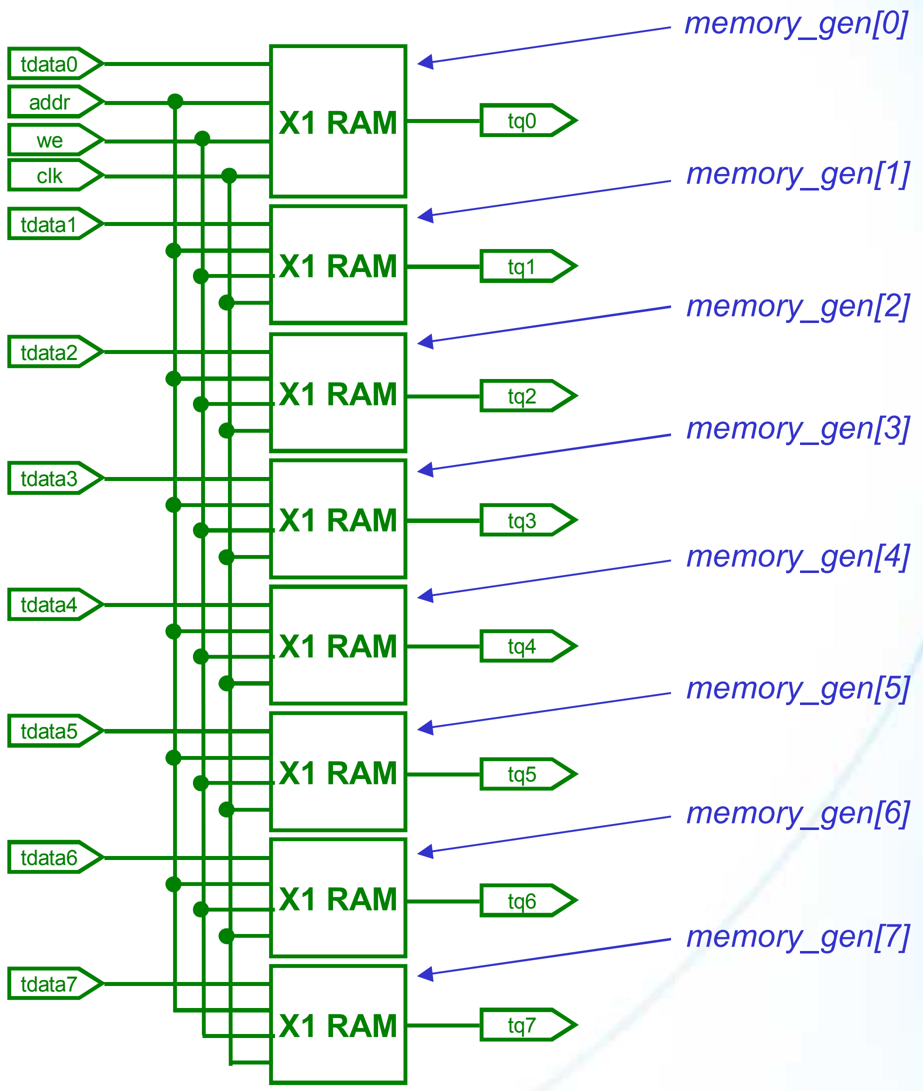
```
module ram16x8
     #(parameter DWIDTH = 8, AWIDTH = 4)
     (
     input tclk, twe,
     input [DWIDTH-1:0] tdata,
     input [AWIDTH-1:0] taddress,
     output [DWIDTH-1:0] tq
     );

     genvar j;

     generate
          for (j=0; j<DWIDTH; j=j+1)
          begin : memory_gen // Generate block name
               ram #(.WIDTH_AD(AWIDTH))
                    u1 (.clk(tclk), .we(twe), .data(tdata[j]),
                         .address(taddress), .q(tq[j]));
          end
     endgenerate

endmodule
```

```
module ram
     #(parameter WIDTH_AD = 2)
     (
      input clk, we, data,
     input [WIDTH_AD-1:0] address,
     output reg q
     );
```

Signal Bit RAM

8 single-bit RAMs generated

8 - 16x1 RAM Functions



Nested Generate Statements

```
module dffn (
        input clk, d,
        output reg q
        );
        always @ (posedge clk)
                q <= d;
endmodule
```

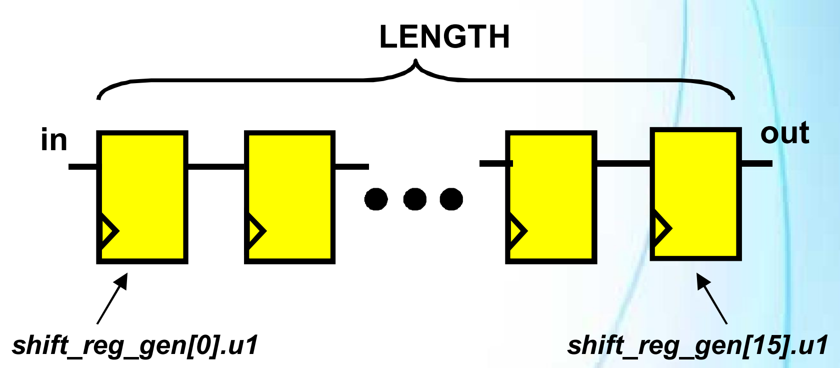
```
module shift_reg
        #(parameter LENGTH = 16)
        (
        input in, clk,
        output out
        );

        wire [LENGTH-1:1] sig;
        genvar n;

        generate
                for (n=0; n<LENGTH; n=n+1)
                begin:  shift_reg_gen
                        case (n)
                                0 : dffn u1 (.q(sig[n+1]), .clk(clk), .d(in));
                                LENGTH-1 : dffn u1 (.q(out), .clk(clk), .d(sigr[n]));
                                default : dffn u1 (.q(sig[n+1]), .clk(clk), .d(sig[n]));
                        endcase
                end
        endgenerate
endmodule
```

generates first register

generates last register

generates intermediate register(s)

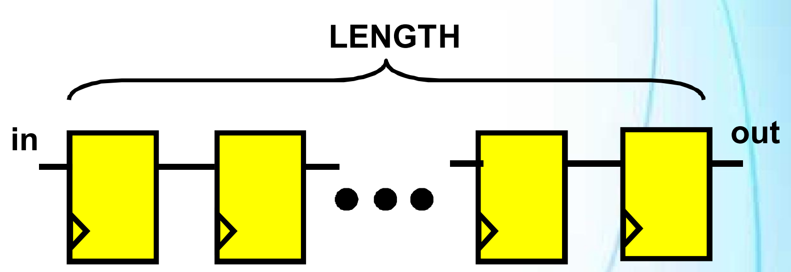
Code generates a shift register



Nested Generate Statements (Alt.)

```
module shift_reg
	#(parameter LENGTH = 16)
	(
	input in, clk,
	output out
	);

	reg [LENGTH-1:1] sig;
	genvar n;

	generate
		for (n=0; n<LENGTH; n=n+1)
		begin:  shift_reg_gen
			case (n)
				0 : always @ (posedge clk)
				    sig[n+1] <= in;
				LENGTH-1 : always @ (posedge clk)
				    out <= sig[n];
				default : always @ (posedge clk)
				    sig[n+1] <= sig[n];
			endcase
		end
	endgenerate
endmodule
```

Alternate code generates a shift register



- *This example illustrates how procedural (always) blocks can also be used in generate statements*
- *Same as writing individual procedural blocks*
  - *–Code expanded at elaboration*

Summary

- More advanced Verilog language concepts than versus introductory class
- Efficient coding styles for good synthesis
- Testbench creation for simulation
- Building re-usable code