TEC de Monterrey®
DEL SISTEMA TECNOLÓGICO DE MONTERREY

VERILOG DESIGN TECHNIQUES

LUIS F. GONZÁLEZ PÉREZ
ITESM - GDA

OBJECTIVES

- Write synthesizable RTL code
- Develop testbenches to perform RTL debugging
- Synthesize and place & route designs
- Simulate design funcionality in the ModelSim®-Altera software

WRITING SYNTHESIZABLE VERILOG

SIMULATION VS SYNTHESIS

- Simulation
  - Code executed exactly the way it is written
- Sythesis
  - Code is interpreted & hardware created
    - PLD knowledge is important
  - Synthesis tools require certain coding style to generate correct logic
    - Subset of Verilog language supported

- **Pre- & post-synthesis logic should function the same**

OPERATORS

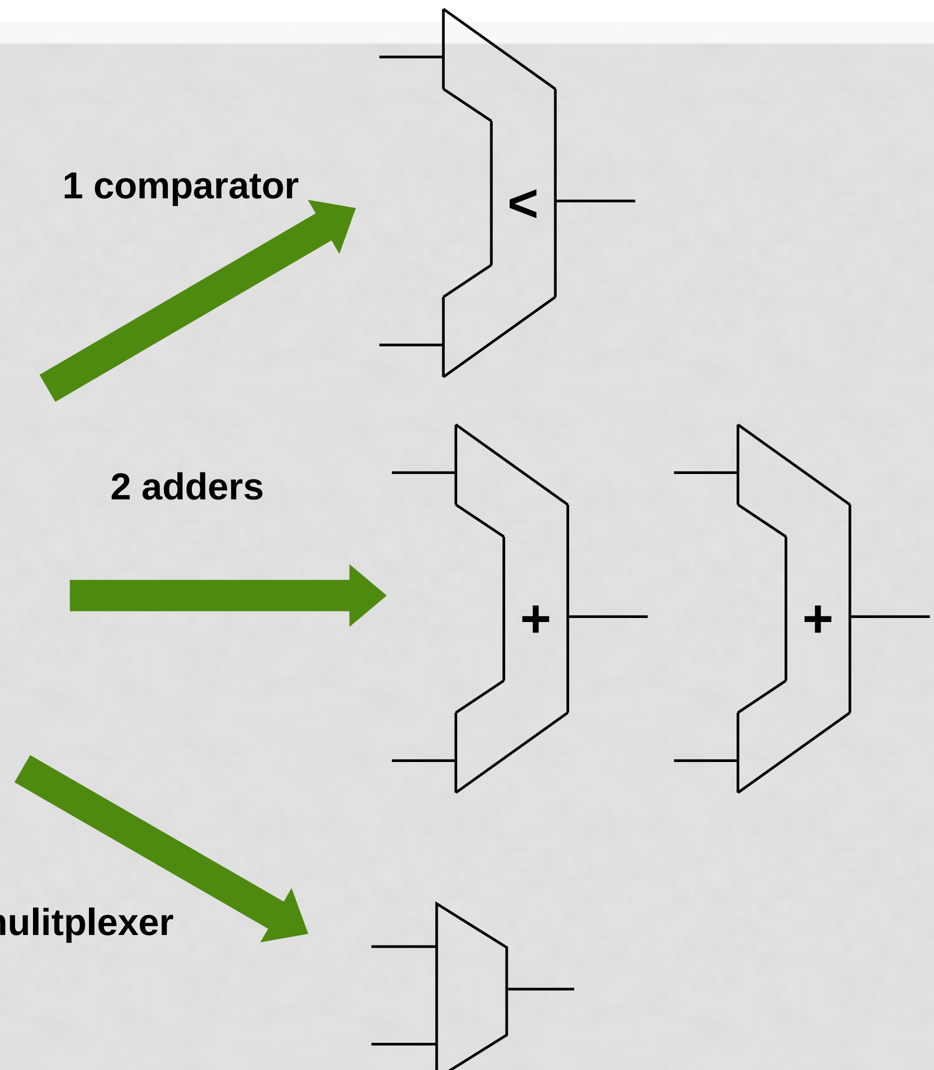
- Synthesis tools replace operators with pre-defined (pre-optimized) blocks of logic

- Designer should be aware of operator use and control when & how many

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;
```

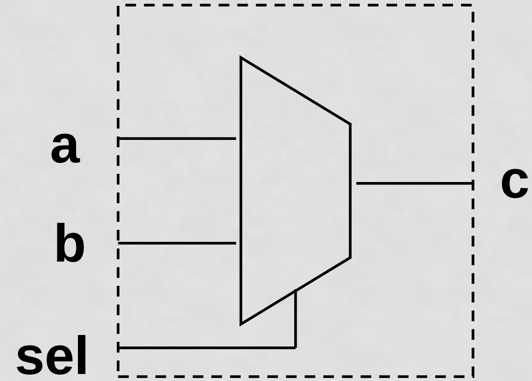


TWO TYPES OF RTL PROCESSES

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

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

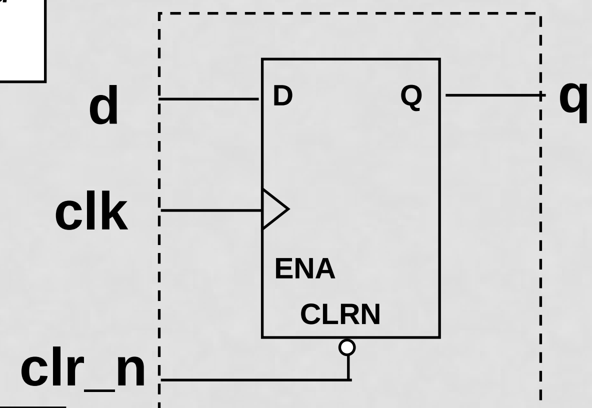
Sensitivity list includes all inputs used in the combinatorial logic



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

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

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



SENSITIVITY LISTS

- A sensitivity list is a list of signals in a Verilog **always** statement that a simulator monitors for changes
- If the always statement infers only flip-flops with associated combinatorial logic on their input or output there is no need to include all input signals in the sensitivity list. Only the clock signal and any asynchronous reset is needed
- If only combinatorial logic is being modeled then **all** input signals to the always statement **must** be included in the sensitivity list
- A signal inadvertently omitted from the sensitivity list will not affect the synthesized circuit but yields unexpected and misleading simulation results

SENSITIVITY LISTS

- Incomplete sensitivity lists in an always block may result in differences between RTL and gate-level simulation
  - Synthesis assumes complete sensitivity list
  - Should include all inputs to the procedural block

```
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 assignments (=)
  - Use in combinatorial procedural blocks
  - Easier to read
  - Use less memory & simulate faster
    - No scheduling of updated signals

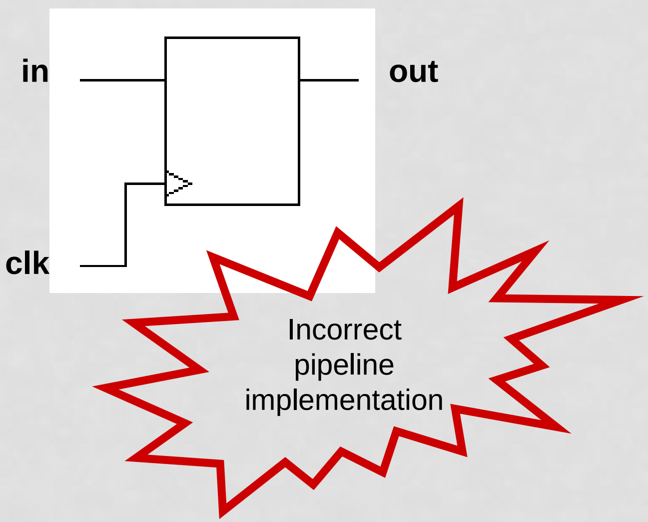
- Non-blocking assignments (<=)
  - Use in clocked procedural blocks

CLOCKED PROCESS EXAMPLE

Blocking (=)

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

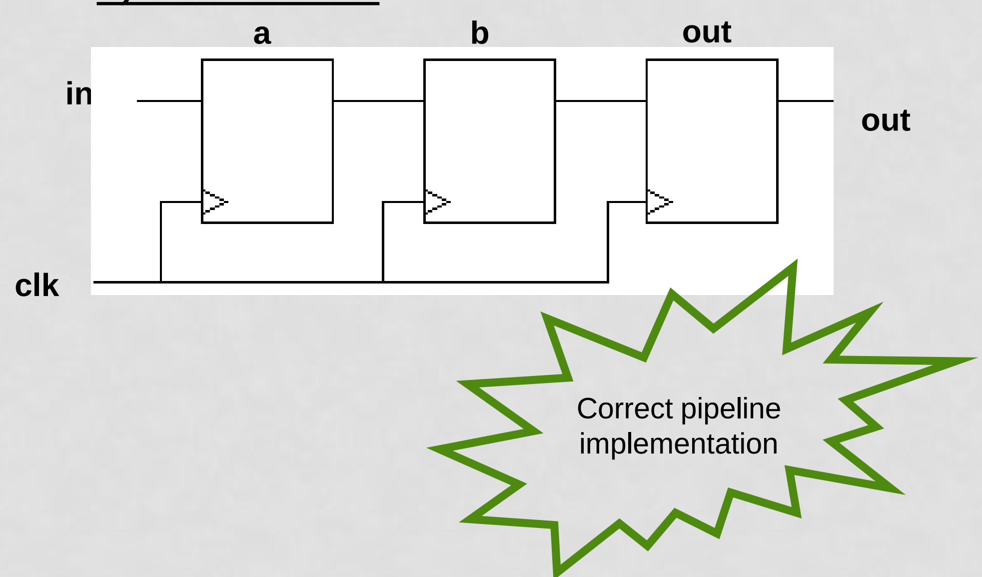
Synthesized result:



Nonblocking (<=)

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

Synthesized result:



LATCHES VS REGISTERS

- Most PLDs have registers in logic elements, not latches
- Latches implemented using combinatorial logic
  - Makes timing analysis more complicated
  - This consumes LUTs in FPGA devices
  - Product-term devices (CPLDS) use more product-terms
- Recommendations
  - Design with registers
  - Take care of inferred latches
    - Inferred on combinatorial outputs when results not specified for set of input conditions
    - Lead to simulation/synthesis mismatches

if-else STRUCTURE

- Implies proritization and dependency
  - N-th clause implies all n-1 previous clauses not true
    - Beware of needlessly logic

Logical equation

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

  - Restructure ***if*** statements
    - May flatten the multiplexer and reduce logic

```
if <cond1> begin
   if <cond2> …
```

```
if <cond1> and <cond2>
```

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

RECOMMENDATIONS

- Cover all cases
  - Uncovered cases in combinatorial processes result in **latches**
- Assign values before reading data type objects
  - Reading unassigned data type objects results in **latches**
- For efficiency
  - Use don't cares (X) for final else clause since synthesis tool has freedom to encode don't cares for maximum optimizaction
  - Assigning intial values and explicitly covering only those results different from initial values

PRIORITIZATION & DEPENDENCY (if-else)

- Nested else if statements imply priotitization 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

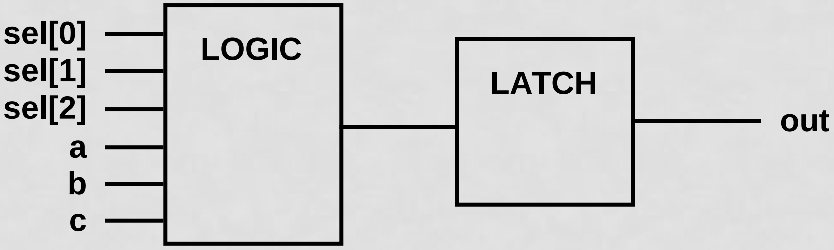
- Individual if statements where logic is exclusive may be more efficient

```
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

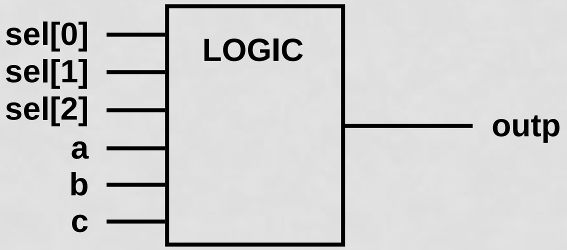
```
always @ *
begin
    if (sel == 3'b001)
        out = a;
    else if (sel == 3'b010)
        out = b;
    else if (sel == 3'b100)
        out = c;
end
```



UNWANTED LATCHES REMOVED

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

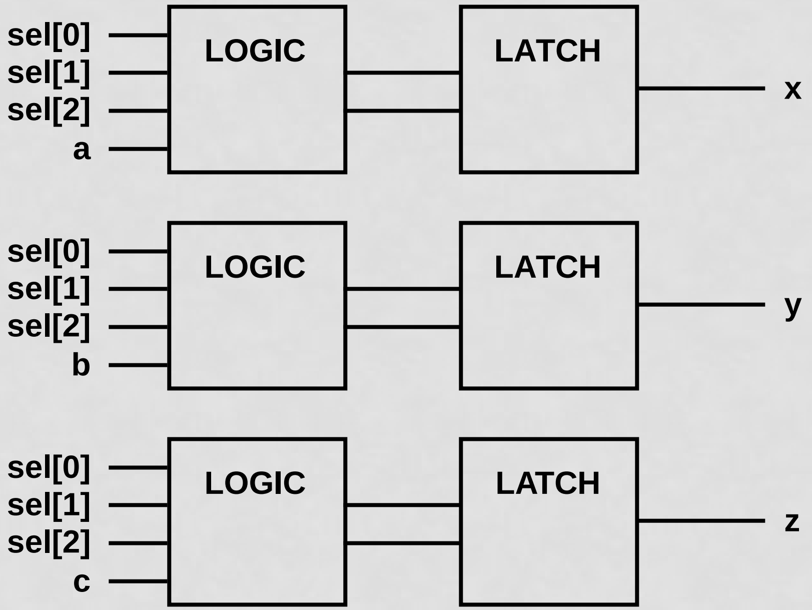
```
always @ *
begin
    if (sel == 3’b001)
        outp = a;
    else if (sel == 3’b010)
        outp = b;
    else if (sel == 3’b100)
        outp = c;
    else
        outp = 3’bx;
end
```



MUTUALLY EXCLUSIVE if-else LATCHES

- Avoid building unnecessary dependencies
  - Outputs x, y, x are mutually exclusive
  - If-else causes all outputs to be dependant on all tests & creates latches

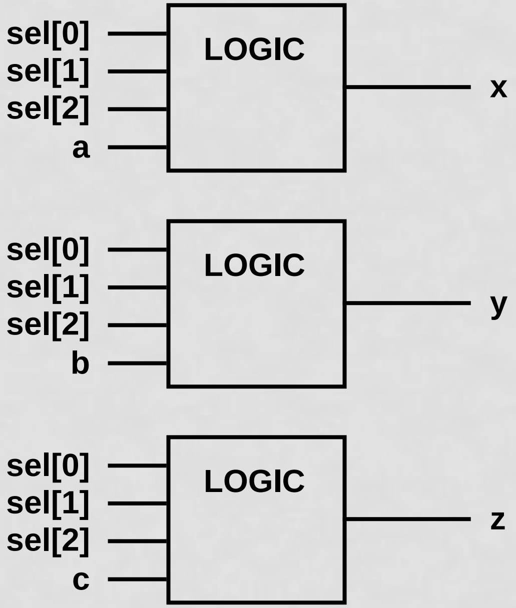
```
always @ (sel,a,b,c)
begin
    if (sel == 3'b010)
        x = a;
    else if (sel == 3'b100)
        y = b;
    else if (sel == 3'b001)
        z = c;
    else
        x = 0;
        y = 0;
        z = 0;
end
```



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
```

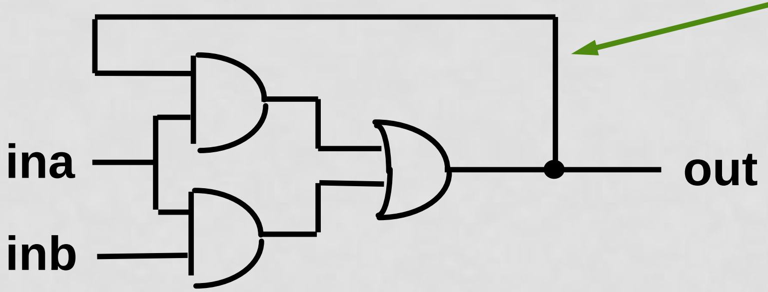


UNWANTED LATCHES IN NESTED *if* STATEMENTS

- Use nested if statements with care
  - Nested if statements may not cover all possible conditions & 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	?

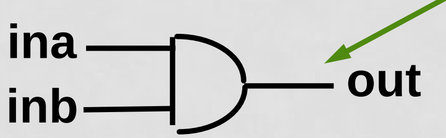


- Uncovered cases infer latches
  - No default value for data type objects
- Logic equation
  - A1L3 = LCELL(**ina** & (A1L3 # **inb**));

UNWANTED LATCHES REMOVED

```
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

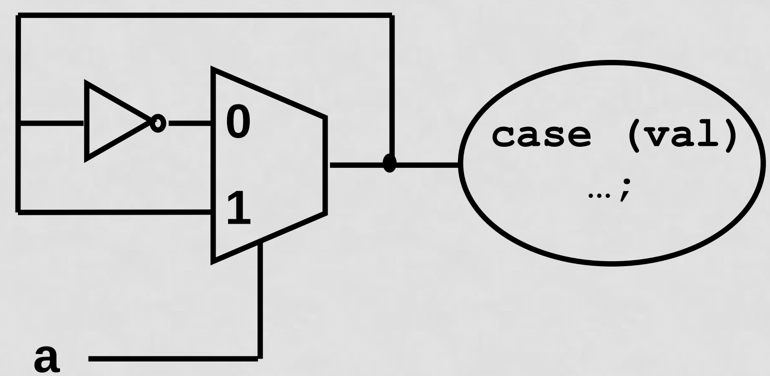


- Using initialization to cover all cases; no latch inferred
- Logic equation
  - A1L3 = **inb** & **ina**;

VARIABLE NOT INITIALIZED

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

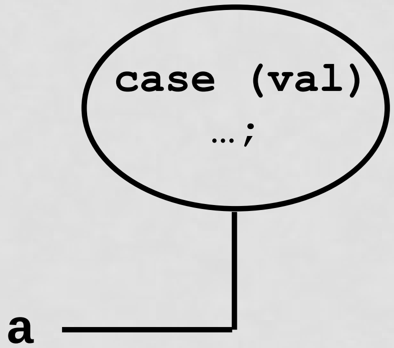
```
always @ *
   begin
      if (a == 1) val = val;
      else val = val + 1'b1;
      case (val)
            1'b0 : q = i[0];
            1'b1 : q = i[1];
      endcase
   end
```



ASSIGNING INITIAL VALUE TO A VARIABLE

- Assign value to a data type object prior to reading it to prevent latch inference

```
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
- Recommendations
  - Make sure all case items are unique/parallel
    - Non-parallel cases infer priority encoders (i.e. less efficient logic)
  - Cover all cases
    - Uncovered cases infer latches
    - Caused by
      - incomplete **case** statements
      - Outputs not defined for one **case** item
  - Initialize all case outputs or ensure outputs assigned in each case
  - Use default clause to close undefined cases (if any remain)
  - Try to initialize to don't cares for further optimization

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

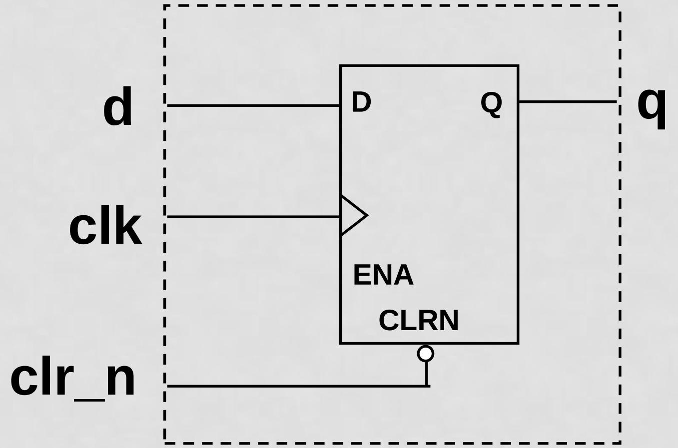
INFERRING COMMON LOGIC FUNCTIONS

INFERRING LOGIC FUNCTIONS

- Use behavioral modeling to describe logic blocks
- Synthesis tools recognize description & insert equivalent logic functions
  - 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

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*

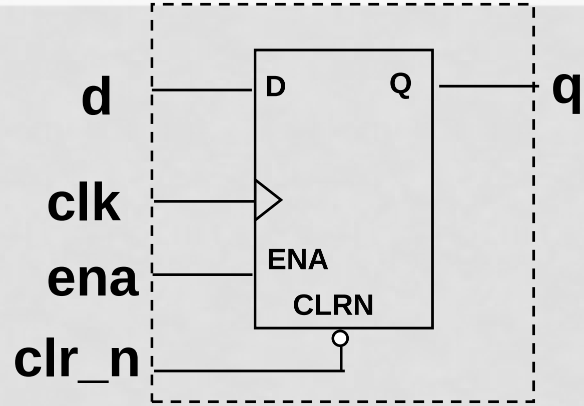
DFF WITH SECONDARY CONTROL SIGNALS

```
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
```

- *Template shows how to implement all asynchronous and synchronous control signals for 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 require additional logic to support all at once*

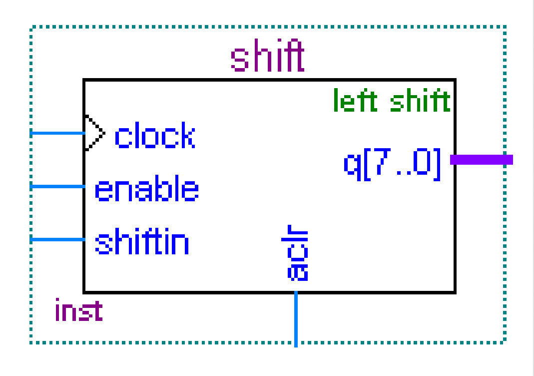
DFF WITH CLOCK ENABLE

```
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 guarantee that this is synthesized using DFFE primitives (DFF with enable)
  - *Place the enable check directly after the rising edge statement*
  - *Place enable expressions in separate process or assignment*
- *May still be recognized correctly with other coding styles*
- *Implemented using extra LUTs if not recognized by synthesis tool*

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
```

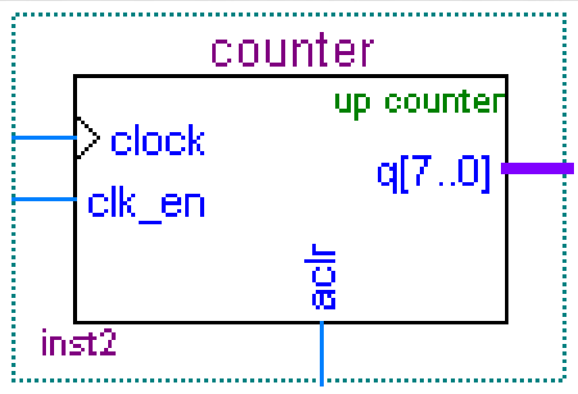
- *Shift register with parallel output, serial input, asynchronous clear, and enable which shifts left*
- *Add or remove synchronous controls in a manner similar to DFF*

Shift function

- *Use { , } for concatenation*

BASIC COUNTER WITH ASYNC CLEAR & CLOCK ENABLE

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

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

endmodule
```

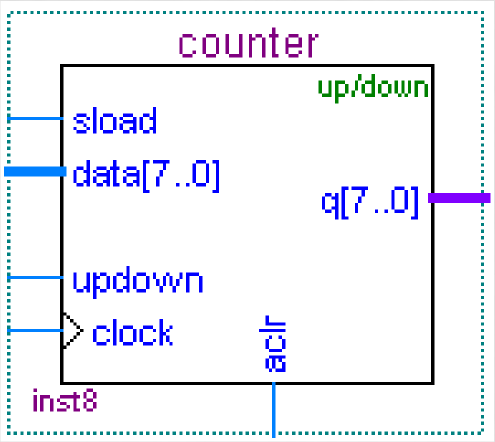


Count function

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

UP/DOWN COUNTER WITH SYNC LOAD

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

MEMORY

- Synthesis tools have different capabilities for recognizing memories
- Synthesis tools are sensitive to certain coding styles in order to recognize memories
- Tools may have limitations in architecture implementation
  - Synchronous inputs
  - Memoty 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 [0:127];

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>*
– ***Old data*** *read-during-write behavior*
  – *Memory read in same process/cycle as memory write using* ***non-blocking*** *assignments*
  – *Check target architecture for support as unsupported features get built using LUTs/registers*

INFERRED SINGLE-PORT MEMORY (3)

```
module sp_ram_sync_rdwn (
   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; // Blocking
   q = mem[addr];  // Blocking
end

endmodule
```

- *Same memory with **new data** read-during-write behavior*
  - *Read performed in same process/cycle using **blocking** assignments*
- *Check target architecture for support*
- *Use* `ramstyle` *synthesis attribute set to* `no_rw_check` *to prevent extra logic generation that compiler would need to add to follow HDL code read-during-write behavior*
  - *Output when reading and writing same address is don't care*

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)* ***128 x 8 RAM*** *with single clock*
- *Code implies* ***old data*** *read-during-write behavior*
  - *New data support in simple dual-port (using blocking assignments) 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) **128 x 8 RAM** with dual clocks*
- *May not be supported in all synthesis tools*
- ***Old data same-port** read-during-write behavior shown*
  - *New data (blocking assignments) only supported in pre-90 nm devices*
- *Mixed port behavior (read and write on different ports for same address) 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*

INFERRED ROM (case STATEMENT)

```
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*

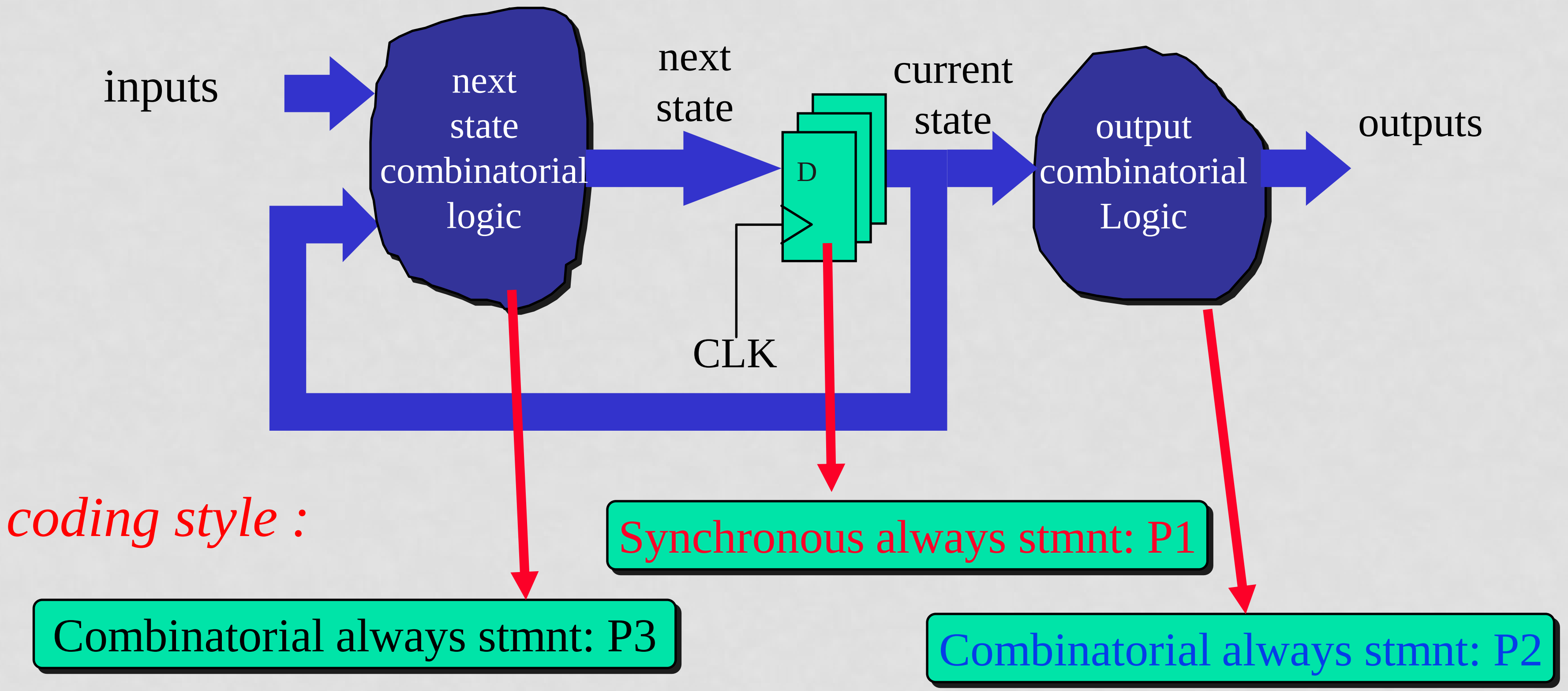
INFERRED ROM (MEMORY FILE)

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

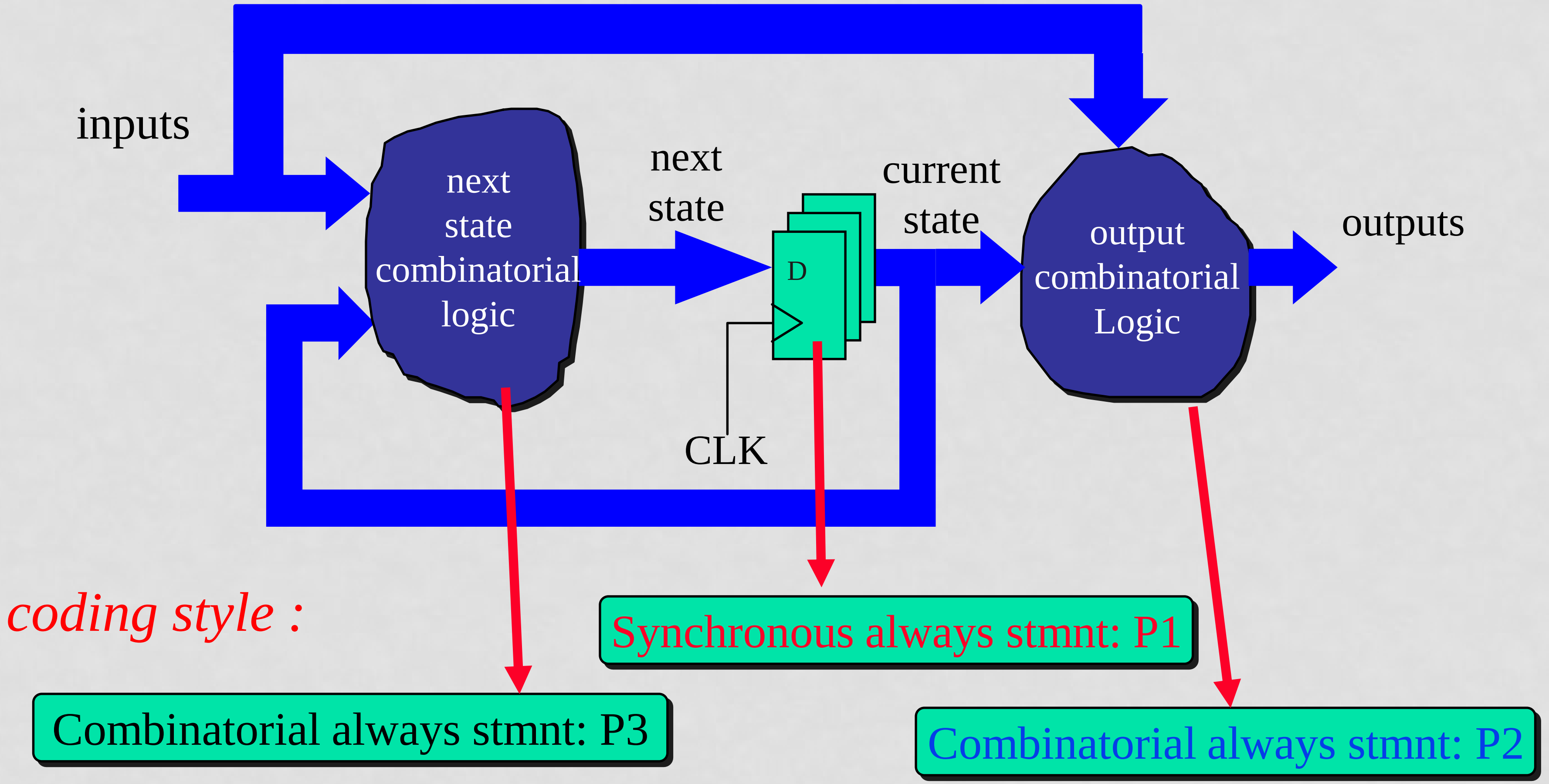
- *Using* `$readmemb` *or* `$readmemh` *to initialize ram contents*
- *No write control*
- *Example shows* ***dual-port*** *access*
- *Automatically converted to ROM*
- *Tools generate ROM using embedded RAM & initialization file*

STATE MACHINES

MOORE MACHINE



MEALY MACHINE



STATE MACHINE CODING

- Parameters or local parameters used to define states

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

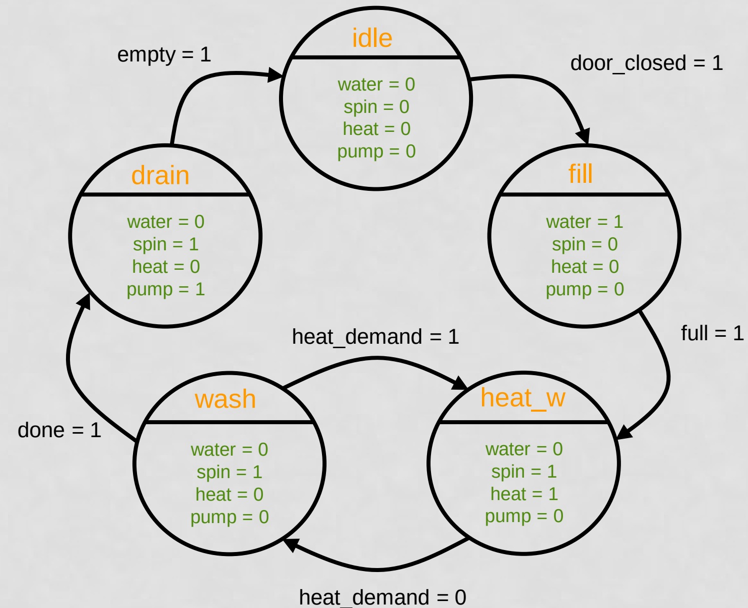
  - Parameter values are replaced with state encoding values as chosen by synthesis tool
    - Use options/constraints in synthesis tool to control encoding style (e.g. binary, one-hot, safe, etc.)
- Registers are used to store state

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

- Separate sequential process from combinatorial process
- Sequential process should always include synchronous or asynchronous reset
- Use case to do the next-state logic, instead of if-else

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;
```



State registers

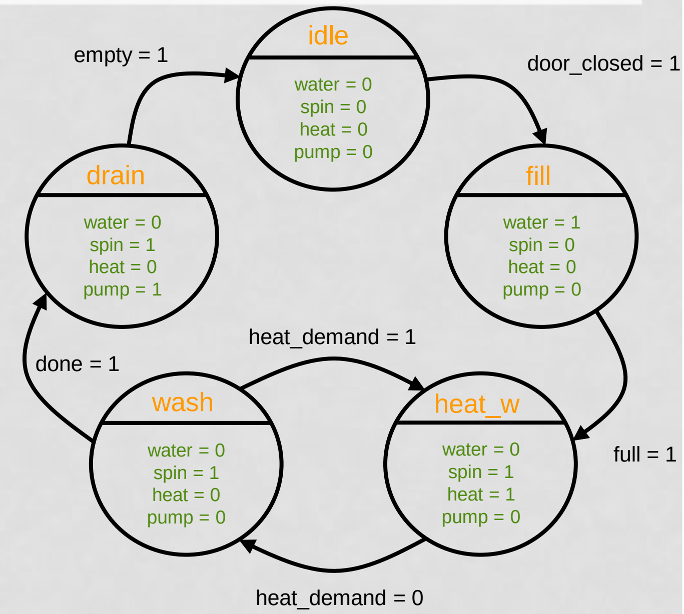
States

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

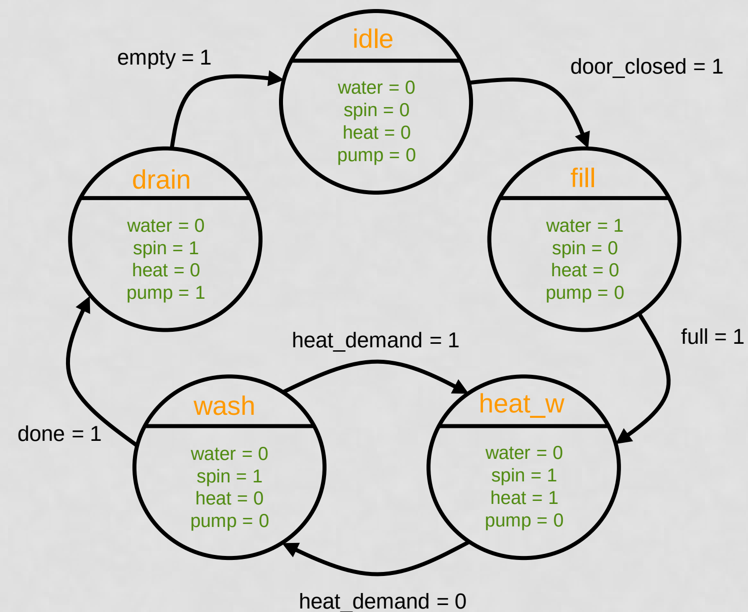
Next state logic



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*

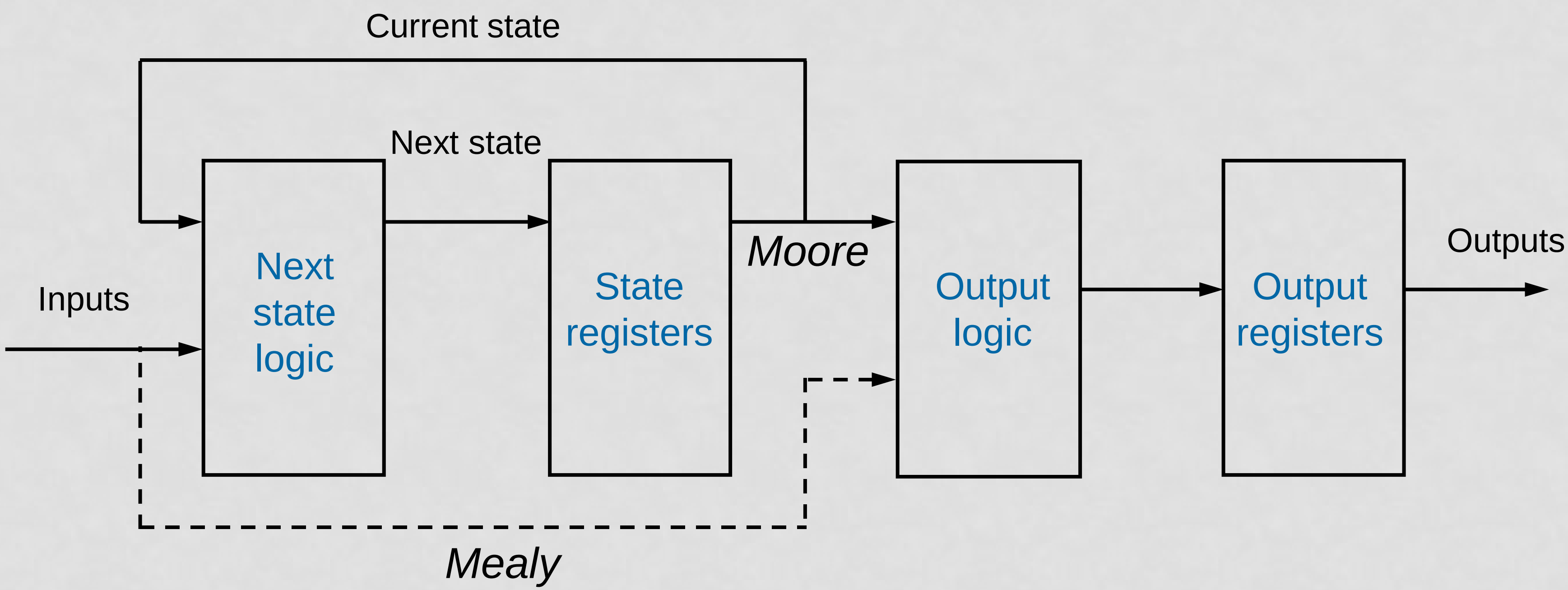
STATE MACHINE ENCODING STYLES

State	Binary	Grey-code	One-hot	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 uses default encoding styles for Altera devices
  - One-hot encoding for LUT devices
    - Architecture features lesser fan-in per cell and lots of registers
  - Binary or grey-code encoding for product-term devices
    - Architecture features fewer registers and greater fan-in

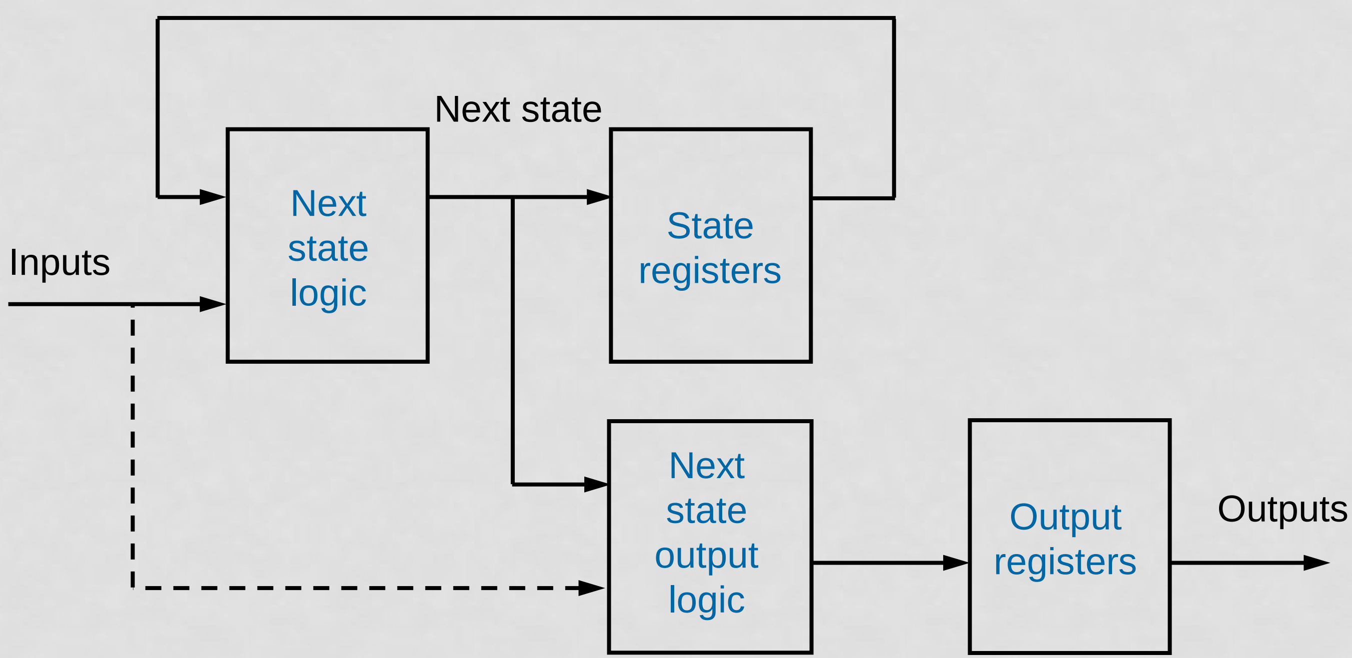
REGISTERED OUTPUTS

- Remove glitches by adding output registers
  - Caution: adds a stage of latency



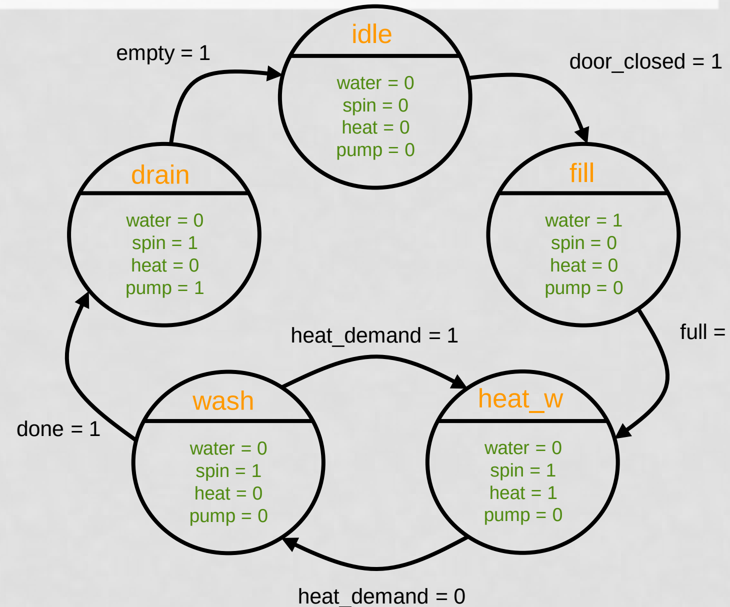
REGISTERED OUTPUTS WITHOUT LATENCY

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



REGISTERED OUTPUTS WITHOUT LATENCY

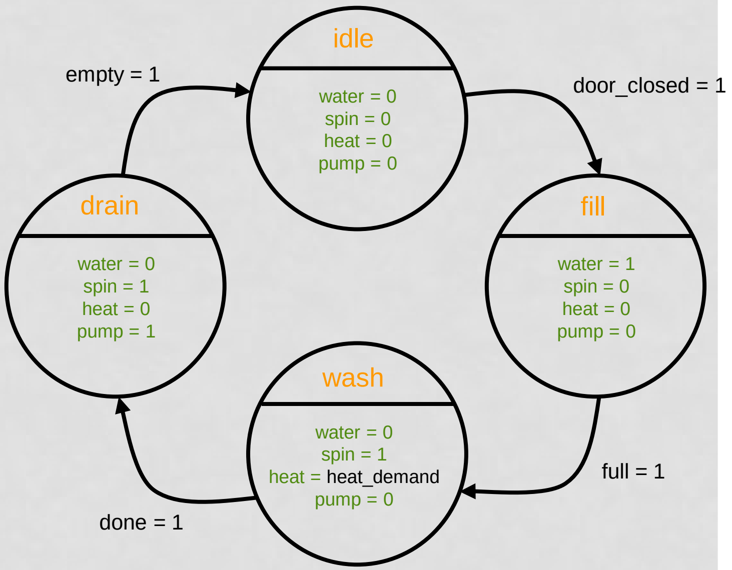
```
//output logic
always @ (posedge clk)
begin
  water = 0;
  spin = 0;
  heat = 0;
  pump = 0;
  case (next_state)
    idle:    ;
    fill:    water <= 1;
    heat_w:  begin spin <= 1; heat <= 1; end
    wash:    spin <= 1;
    drain:   begin spin <= 1; pump <= 1; end
  endcase
end
```



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

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 = heat_demand;
             end
    drain:   begin spin = 1; pump = 1; end
  endcase
end
```



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

TESTBENCHES

TESTBENCH

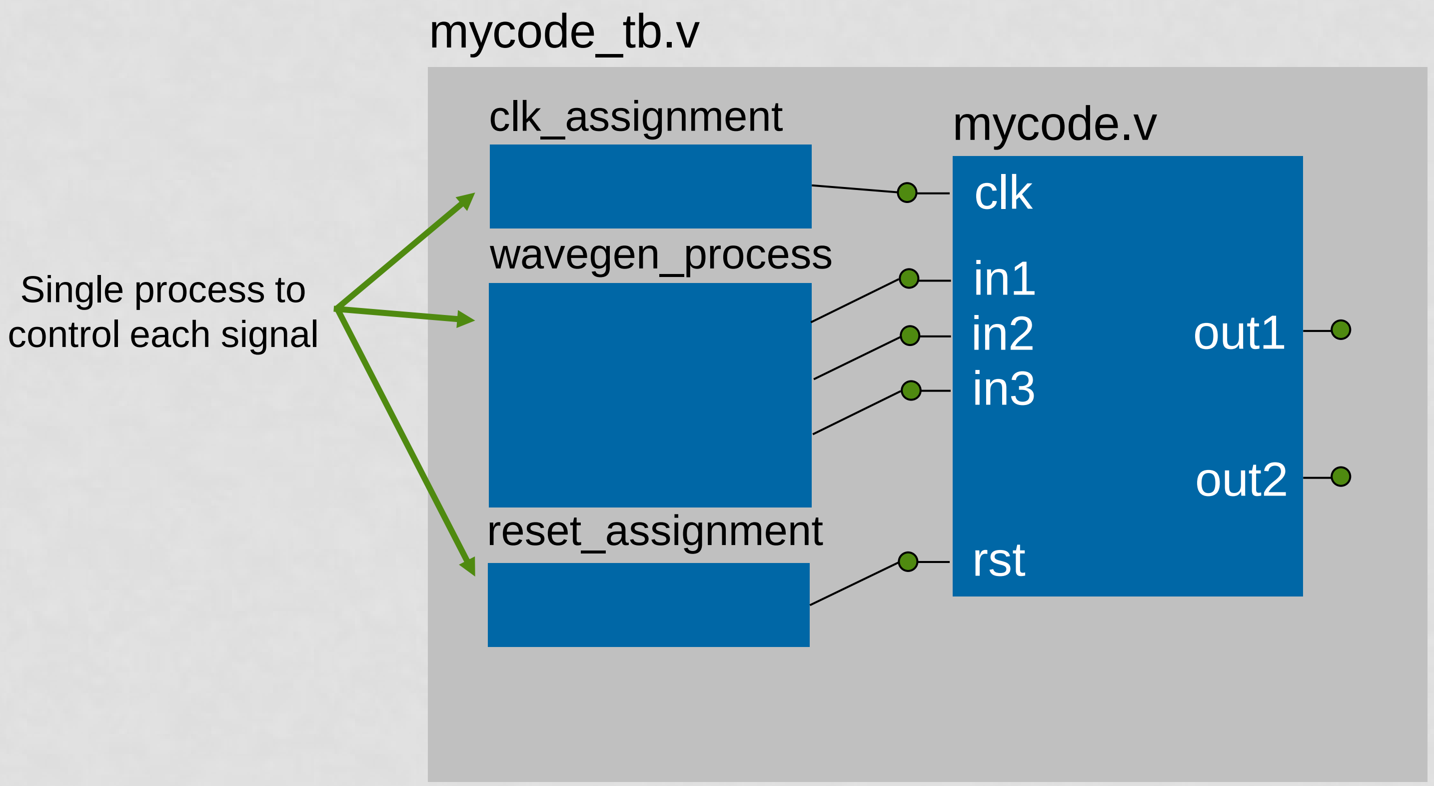
- Generates stimulus to test design for normal transactions, corner cases and error conditions
  - Direct test
  - Random test
- Automatically verify design to spec and log all errors
  - Regression tests
- Log transactions in a readable format for easy debugging

THREE CLASES

I. Test bench applies stimulus to target code and outputs are manually reviewed
   - Requires static timing analysis
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

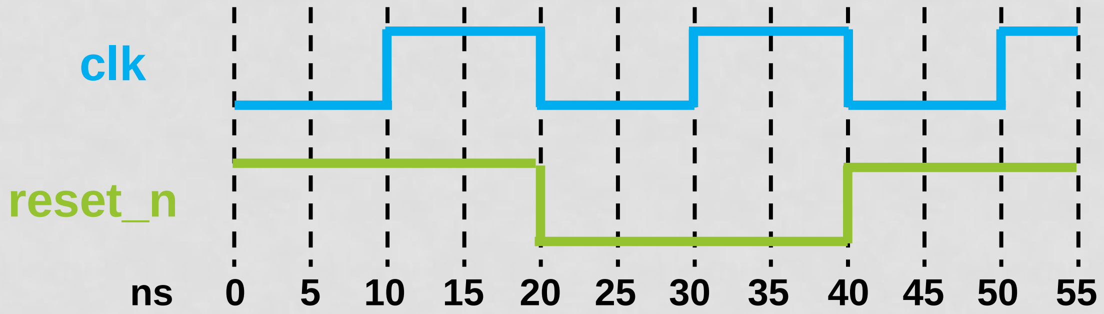
CLASS I METHODS

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



CONCURRENT STATEMENTS

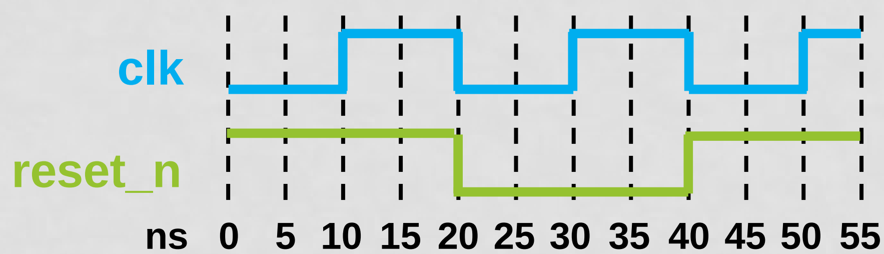
- Signals with regular or limited transitions can be created with separate `initial` blocks (concurrent statements)
- Can be used to begin a testbench and reside outside any other processes



EXAMPLE INITIAL BLOCKS

```
`timescale 1ns / 1ns
`define CLKPERIOD 20
module tb();  // No ports

reg clk;
reg reset_n;

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

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

initial #2500 $stop;

endmodule
```

- *Use* `initial` *blocks*
- *Tells the simulator to run the code at time zero*
- *Code continues running until all statements finish*



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` *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*

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

- Define a timescale
  - Choose largest precision that can accurately model the system
    - Too small needlessly increases memory usage and simulation time
  - Timescale passed to all modules without defined timescales that are subsequently elaborated
    - Simulator warning
- Define clock period(s)
  - Place in top-level test bench module or in a “definitions” header file 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

CLASS I TESTBENCH EXAMPLE

```
`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

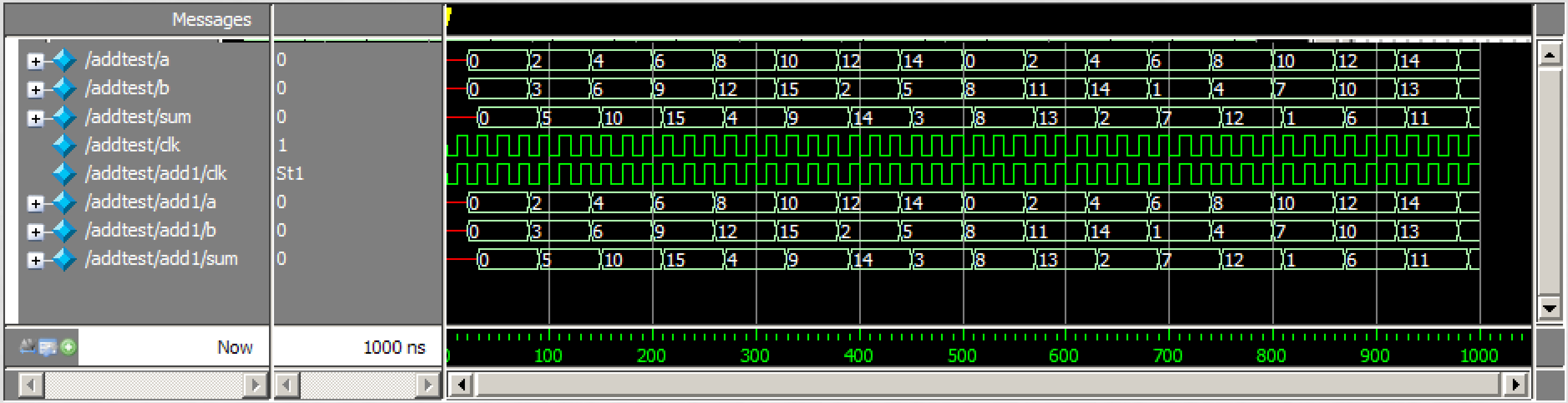
Signals to assign values & observe

Instantiate lower-level entity

Create clock to synchronize actions

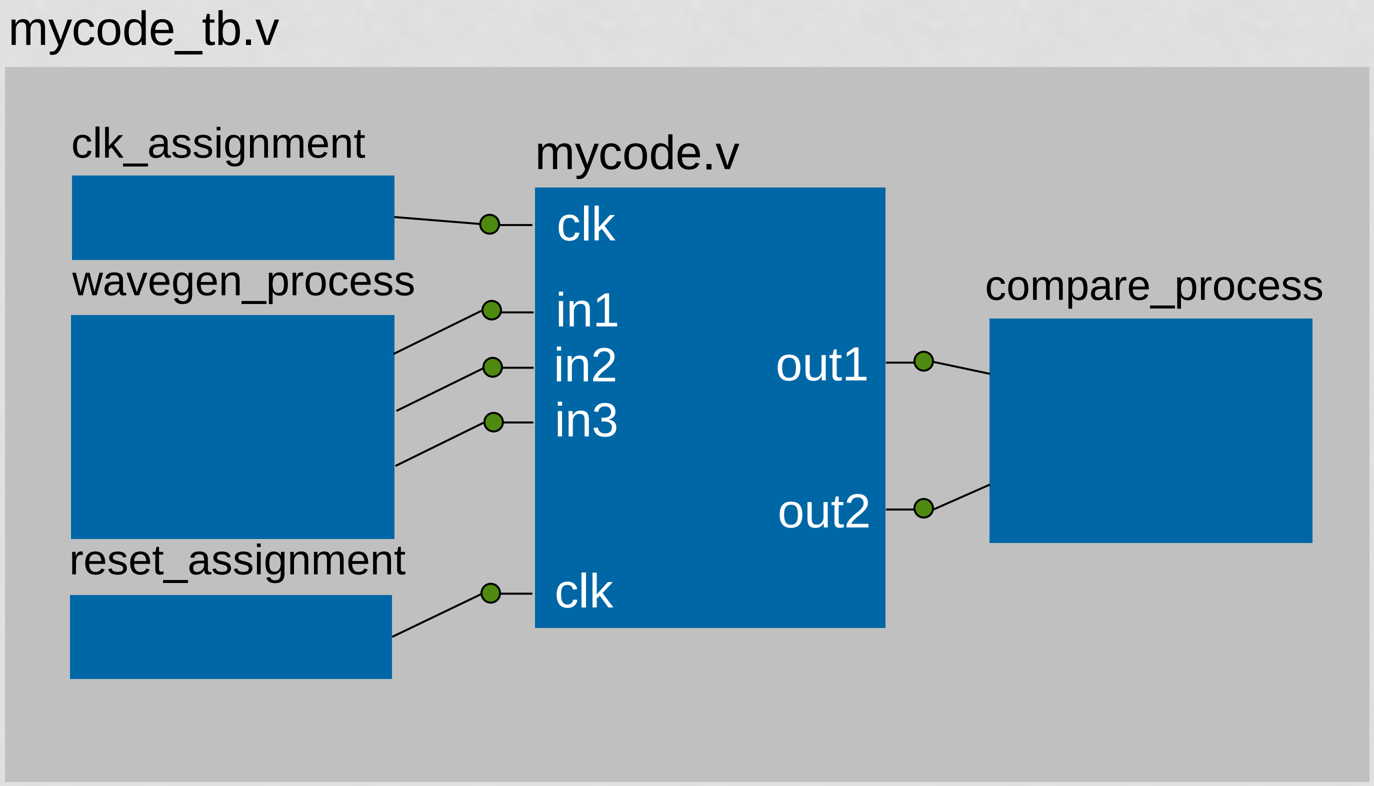
Apply stimulus; note input data changing on inactive clock edge

EXAMPLE RESULTS



CLASS II 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"

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
```

- *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 `$readmemh` or `$readmemb` system tasks to read external files into a Verilog array *(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 design 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
```

- One 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!
- 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, etc.)

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*

VERILOG FILE I/O SYSTEM TASKS

- Use to move values between files and variables
- Syntax similar to ANSI-C
- Examples
  - Read values into non-memory vectors in testbench
  - Capture simulation results in an external file
    - Keeps a record of simulation (versus simulator display)
    - May be easier to analyze
  - Manipulate data files of different formats (beyond binary & hex)
  - Determine the number of lines in a stimulus file
  - Use software program to generate (and format) vectors and read them into testbench

FILE OPENING & CLOSING COMMANDS

- $fopen
  - Opens file for accessing & returns file handle
  - Syntax:

```
fd = $fopen("<filename>", <type>);
```

```
integer my_fd;
initial
   my_fd = $fopen("invecs.txt", "r");
```

- Fd: 32-bit variable used as file descriptor (fd)
- Type: character string indicating how file is opened
  - "r" or "rb": Text or binary file open for reading
  - "w" or "wb": Text or binary file open for writing (file data deleted)
  - "a" or "ab": Text or binary file open for appending
  - "r+" or "rb+": Text of binary file open for reading and writing

FILE OPENING & CLOSING COMMANDS

- $fclose
  - Closes a file
  - Syntax:

```
$fclose(fd);
```

```
initial
   #2000 $fclose(my_fd);
```

READING DATA FILE COMMANDS

- **`$fgetc`**
  - Reads and returns a single byte (character) from file
    - Returns integer EOF (-1) if end of file has been reached
    - Use variable wider than 8 bits to differentiate between EOF and 0xFF
  - Syntax: **`c = $fgetc(fd);`**

```
integer char;
char = $fgetc (my_fd);
```

- **`$ungetc`**
  - Inserts character into file stream so that it will be the next character read by `$fgetc`
    - Returns integer 0 if successful; EOF if unsuccessful
  - Syntax: **`c = $ungetc(<char>, fd);`**

```
integer c;
c = $ungetc (k, my_fd);
```

- **`$fgets`**
  - Reads string characters from file into `reg` variable
    - Ends when
      - Variable array fills
      - Newline character reached
      - End of file
    - Returns integer number of characters read if successful; 0 if unsuccessful
  - Syntax: **`c = $fgets(str, fd);`**

```
integer c, no_chars;
reg [8*no_chars-1:0] str;
c = $fgets (str, my_fd);
```

READING DATA FILE COMMANDS (2)

- **`$fscanf`**
  - Reads formatted text from file
    - Returns number of successfully assigned items
    - Uses C-style formatting (e.g. `%d`, `%h`, `%b`, `%c`, `%s`)
    - Terminated if EOF reached
  - Syntax: `c = $fscanf(fd, <format>, <args>);`
    - **format**
      - Whitespace treated as single character matching zero or more whitespace characters
      - Standard characters (non-conversion codes) must match next character in file
      - `*` character skips an input field (e,g, "`%d %d %* %d`")
    - **args**
      - Variables into which file values are read
      - Must have enough arguments to match formatting

```
integer c, a, b;
reg [15:0] data;
c = $fscanf(my_fd, "%d %d %b\n",
          a, b, data);
```

*Two decimal numbers stored in variables **a** and **b** and one binary number stored in **data***

- **`$fread`**
  - Reads binary data from file into `reg` variable or memory
    - Returns number of items read; 0 if error
    - 'X' and 'Z' not supported
    - Data always loaded from lowest to highest address
  - Syntax: `c = $fread(<reg_or_mem>, fd,<start_addr>, <end_addr>);`
    - **<start_addr>** & **<end_addr>**
      - Optional
      - Ignored if reading to `reg` variable (not memory)

```
integer c;
reg [15:0] r;
reg [15:0] mem;
c = $fread(r, my_fd);
c = $fread(mem, my_fd, 20);
```

ADDITIONAL FILE COMMANDS

- **`$ftell`**
  - Returns byte position (offset from beginning of file) of next character to be read from or written to file
  - Syntax: `c = $ftell(fd);`
- **`$fseek`**
  - Sets the position of the next read or write operation to the file
  - Syntax: `c = $fseek(fd, <offset>, <operation>);`
    - **<offset>**: signed distance in bytes from **<operation>** point
    - **<operation>**
      - **0**: Beginning of file
      - **1**: Current byte location
      - **2**: End of file
- **`$rewind`**
  - Sets the position to the beginning of the file
  - Syntax: `c = $frewind(fd);`
- **`$ferror`**
  - Writes the error reason, if any, for the most recent file operation
    - Returns error code if error; 0 if no error
    - String should be at least 640 bits wide
    - Use variable wider than 8 bits to differentiate between EOF and 0xFF
  - Syntax: `c = $ferror(fd, <string>);`
- **`$feof`**
  - Returns non-zero if EOF reached during previous file read; 0 otherwise
  - Syntax: `c = $feof(fd);`

FILE OUTPUT COMMANDS

- **`$fdisplay | $fdisplayb | $fdisplayh | $fdisplayo`**
- **`$fwrite | $fwriteb | $fwriteh | $fwriteo`**
- **`$fmonitor | $fmonitorb | $fmonitorh | $fmonitoro`**
- **`$fstrobe | $fstrobeb | $fstrobeh | $fstrobeo`**

- Each works just like corresponding display system tasks but to files
  - First argument is file handle

STRING SYSTEM TASKS

- Use to read and write formatted strings
  - Read values from files to strings before manipulating
  - Write formatted values to strings before writing to files

- **`$sscanf`**
  - Reads from string
    - Similar to `$fscanf` replacing `fd` with `reg` variable (i.e. string)
- **`$swrite | $swriteb | $swriteh | $swriteo`**
  - Writes to string (similar to `$write` or `$fwrite` except to `reg` variable/string)

EXAMPLE

```
module addtest_file_tb;
reg [3:0] a, b, exp_result;
wire [3:0] sum;
reg clk;
integer i;
integer input_fd, exp_fd, display_fd; // File descriptors (handles) for file operations
integer input_ch, exp_ch;             // Integers to store outputs of $fscanf

initial begin
   clk = 1'b0;
   input_fd = $fopen("init2.dat", "r"); // Open init2.dat for reading
   exp_fd = $fopen("exp.dat", "r");     // Open exp.dat for reading
   display_fd = $fopen("result.out", "w"); // Open result.out for writing
end

always #10 clk = (clk !== 1 ? 1:0);

adder add1 (.clk(clk), .a(a), .b(b), .sum(sum));  // DUT instantiation

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

      // Scan 2 binary values from input stimulus file into dut inputs
      input_ch = $fscanf (input_fd,"%b %b", a, b);

      // Read 1 binary value from expected outputs file
      exp_ch = $fscanf (exp_fd, "%b", exp_result);

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

– *Same as* `$readmemb` *example from earlier replacing all memory operations with file operations*

EXAMPLE FILE I/O FILES

init2.dat

```
0000    0000
xxxx    xxxx
0010    0011
0100    0110
0110    1001
1000    1100
1010    1111
```

exp.dat

```
0000
xxxx
0101
1010
1111
0100
1001
```

result.out (error example)

```
#120 : Calculated 250, Expected 255
```

- *Note: No native commenting or addressing support for `$fscanf` as with `$readmemb`/`$readmemh`*
- *To support commenting/addressing, must use file I/O operations to manually parse through stimulus files*

SOME FINAL DESIGN AND MODELING RECOMMENDATIONS

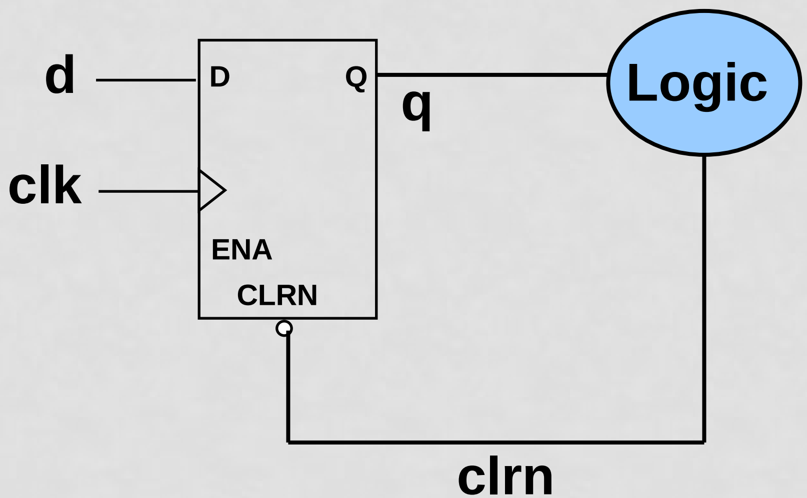
DESIGN AND MODELING RECOMMENDATIONS

- Before attempting to code a model at the RTL level, determine a sound architecture and partition accordingly
- Keep in mind the hardware intent
- Keep in ming the synthesis modeling style and its associated restrictions
- Use global clock and rest signals
- Avoid asynchronous feedback
  - Common cuase of instability
  - Behavior of loop depends on relative propagation delay through logic (propagation delays may change)
  - Simulation tools may not match hardware behavior
- All feedback loops should include registers synchronized to a clock

DESIGN AND MODELING RECOMMENDATIONS

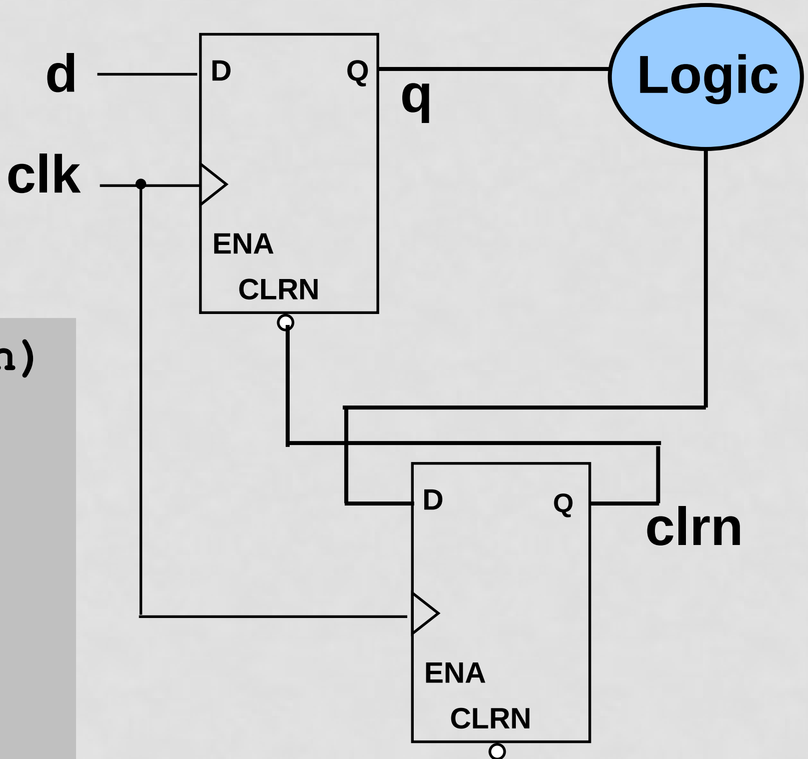
- Asynchronous feedback example

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

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



DESIGN AND MODELING RECOMMENDATIONS

- Avoiding asynchronous feedback




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

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

DESIGN AND MODELING RECOMMENDATIONS

- Remove any race conditions
- Split large counters (large propagation delay)
- Use spare pins to aid controllability and observability of internal circuit nodes
- Make the circuit easy to initialize to a known state
- Ensure the sensitivity list of always statements are complete
- Make models generic as far as possible for model reuse
- Do not repeat identical sections of code in different branches of the same conditional statement; instead, move them out of the conditional expression

DESIGN AND MODELING RECOMMENDATIONS

- Loop invariant signals should not be contained in a loop
- Do not model many small always statements. It takes time to activate and deactivate them. If there are many registers being clocked form the same clock source it is better to put them in one process rather than in separate ones
- Use meaningful signal names
- Use comments !!!!

THANKS!