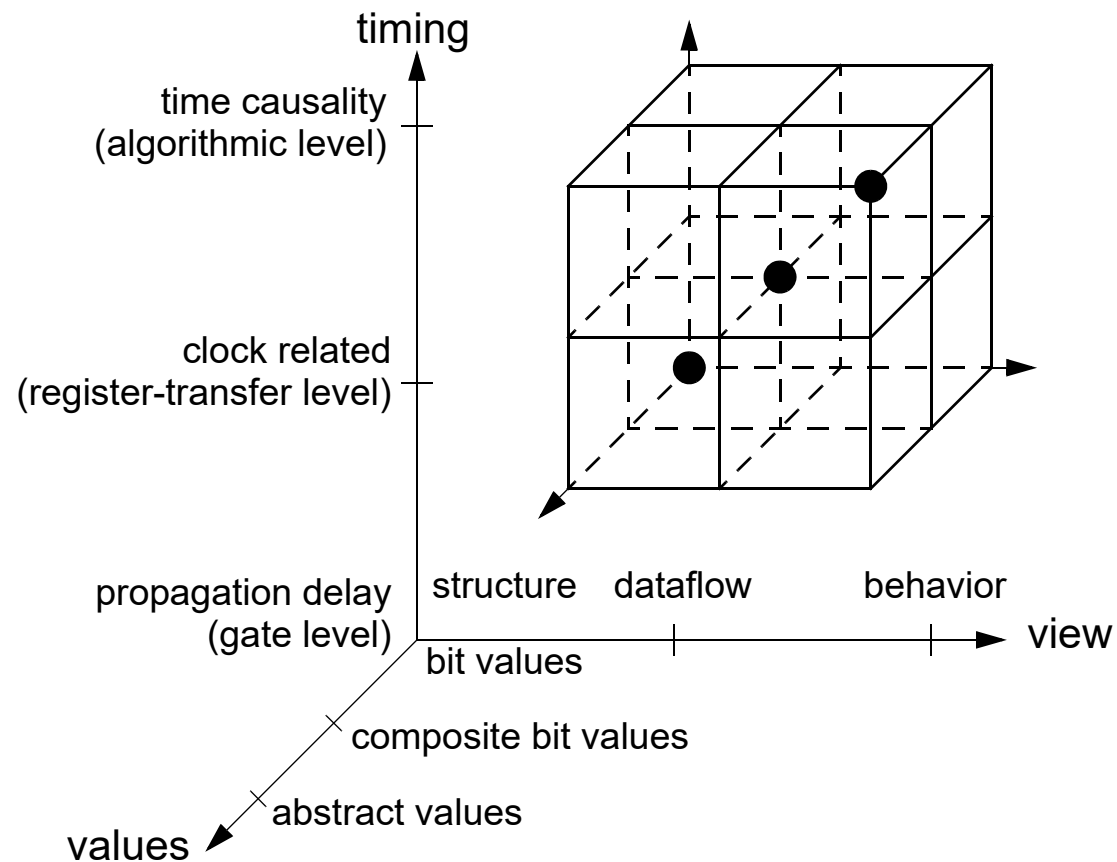


HDL Design Cube



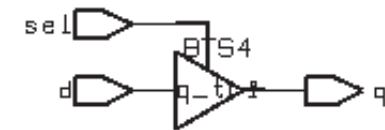


Synthesis and Verilog/SystemVerilog

- **Verilog/SystemVerilog was created as a simulation language**
- **Synthesis restrictions:**
 - the lack of maturity of synthesis tools
 - the state-of-art in synthesis targets RTL synthesis only
 - certain Verilog/SystemVerilog features are simply not synthesizable
- **Potential restrictions to be taken into account**
 - Delay expressions ('#' -clauses) are ignored
 - Certain restrictions on the writing of behavioral statements occur
 - Only a few types are allowed – matching binary logic
 - Description is oriented towards synchronous styles with explicit clocks
 - **!!! Time type is not supported !!!**
 - No explicit nor default initialization
 - Parenthesis in expressions have effect on HW generation
 - Some arithmetic operations are supported partially only

Synthesis rules

- **Guidelines in priority order:**
 - the target signal(s) will be synthesized as flip-flops when there is a signal edge expression, e.g. “ @(posedge CLK) ”, in the behavioral statement
 - only one edge expression is allowed per behavioral statement
 - different statements can have different clocks (tool depending)
 - the target signal will infer three-state buffer(s) when it can be assigned a value 'Z'
 - example: `q = sel == 1 ? d : 'bz;`
 - the target signal will infer a latch (latches) when the target signal is not assigned with a value in every conditional branch, and the edge expression is missing
 - a combinational circuit will be synthesized otherwise
- It is a good practice to isolate flip-flops, latches and three-state buffers inferences to ensure design correctness





Combinational circuit

- **A process is combinational, i.e. does not infer memorization, if:**
 - the behavioral statement has a sensitivity list in the beginning (waiting for changes on all input values); ¹⁾
 - signals are assigned before being read;
 - all signals, which values are read, are part of the sensitivity list; ²⁾ and
 - all output signals are targets of signal assignments independent on the branch of the process, i.e. all signal assignments are covered by all conditional combinations.

¹⁾ waiting on a clock signal, e.g., “ @(posedge clk) ”, implies buffered outputs (FF-s)

²⁾ interpretation may differ from tool to tool

- **SystemVerilog has three new *always* constructs**
 - always_comb – explicit combinational circuit
 - always_latch – explicit latch
 - always_ff – explicit flip-flop



Sensitivity list

- Equivalent statements:

```
always
  @(a or b or c or x or y)
begin
  if (x==1)      s=a;
  else if (y==1) s=b;
  else           s=c;
end
```

==

```
always begin
  @(a or b or c or x or y);
  if (x==1)      s=a;
  else if (y==1) s=b;
  else           s=c;
end
```

- In case of single synchronization process there is no need to “remember” at which synchronization point it was stopped → such behavior does not imply memorization

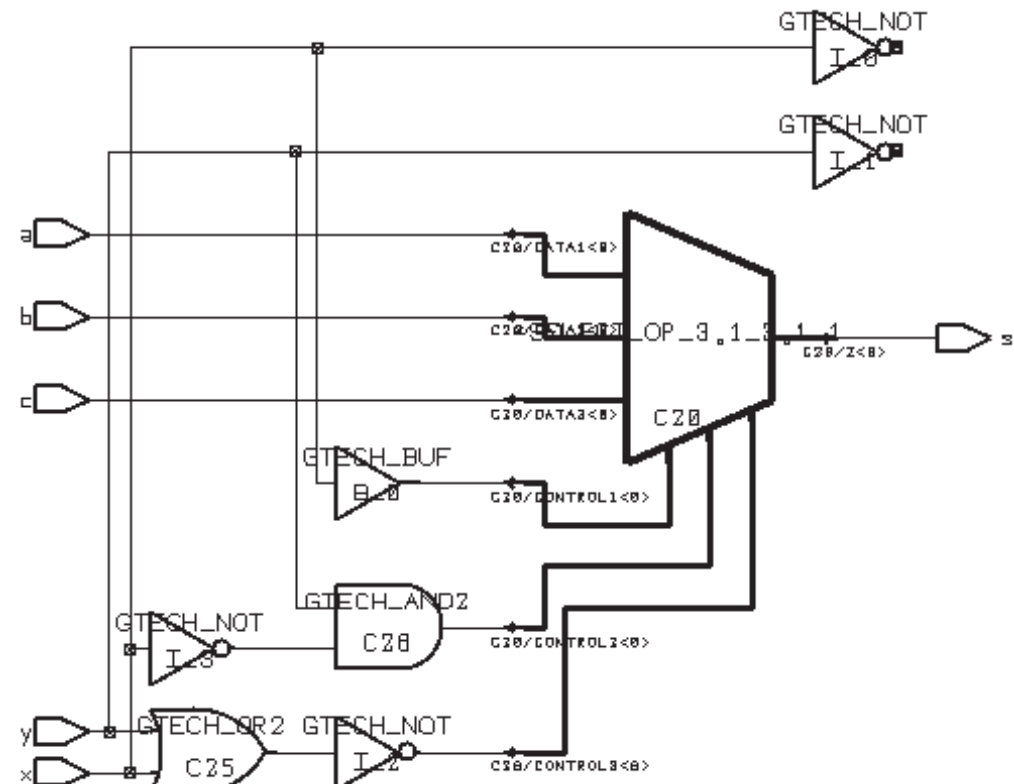
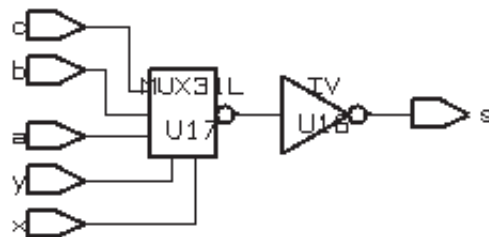
Complex assignments

- No memory

```
assign s = x==1 ? a : y==1 ? b : c;
```

```
always
  @(a or b or c or x or y)
  if (x==1)      s=a;
  else if (y==1) s=b;
  else          s=c;
```

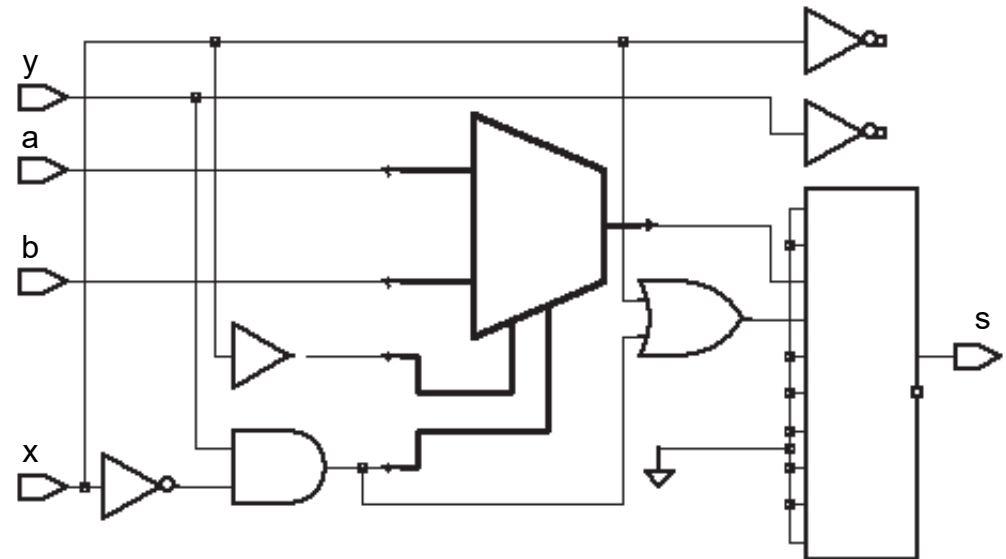
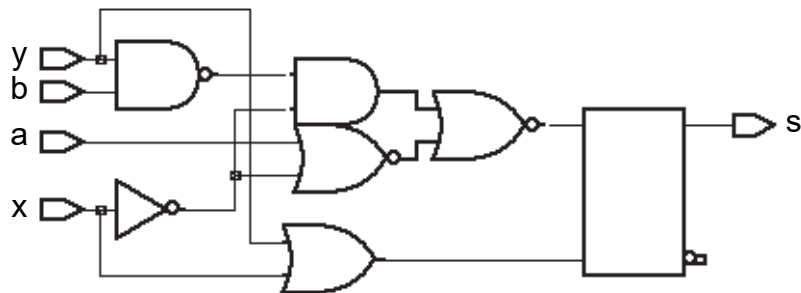
```
always_comb
  if (x==1)      s=a;
  else if (y==1) s=b;
  else          s=c;
```



Complex assignments (#2)

- Memory element generated!

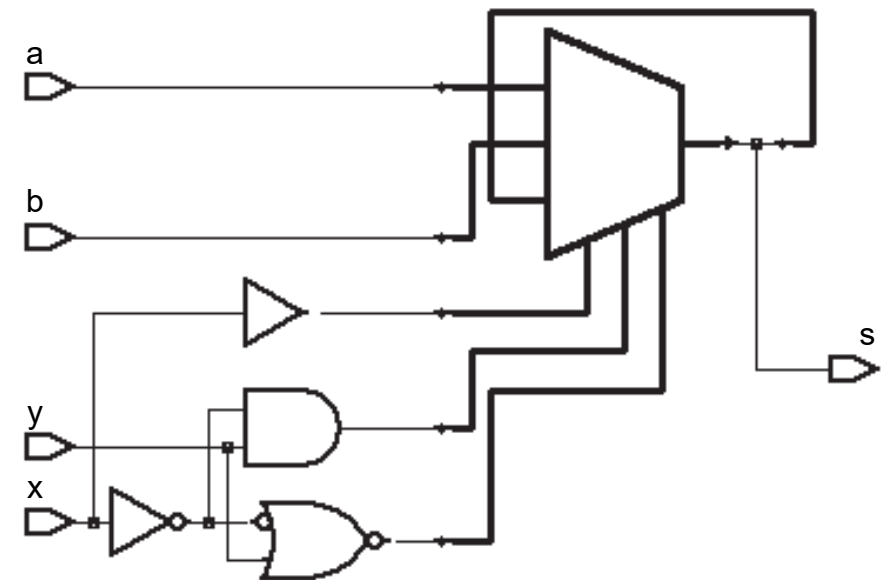
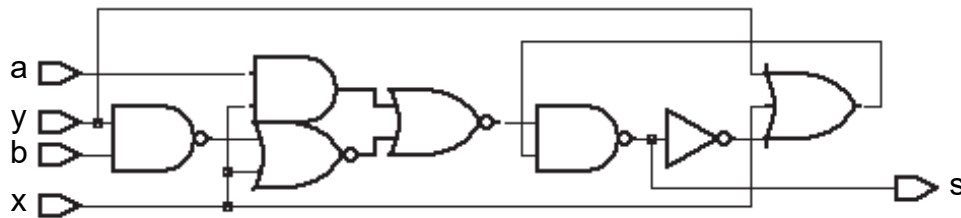
```
always begin
  @(a or b or x or y);
  if (x==1)      s=a;
  else if (y==1) s=b;
end
```



Complex assignments (#3)

- Memory element generated!

```
always begin
  @(a or b or x or y);
  if (x==1)      s=a;
  else if (y==1) s=b;
  else          s=s;
end
```

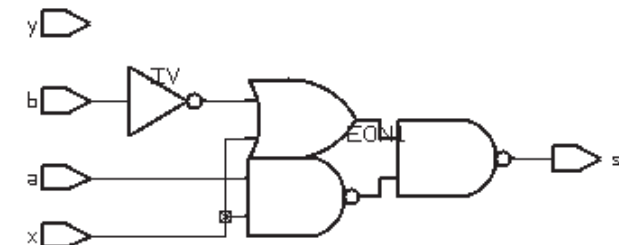
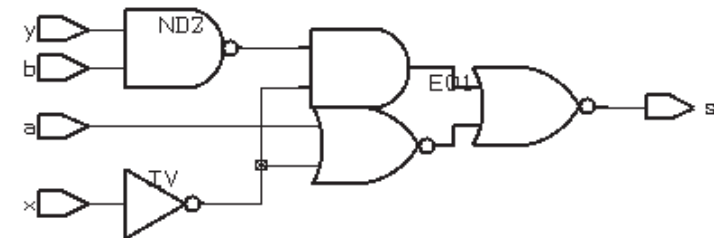


Default values

- The default values inherited from type or subtype definitions
- The explicit initialization that is given when the object is declared
- A value assigned using a statement at the beginning of a process
- Only the last case is supported by synthesis tools!
- Usually, a part of the synthesizable code is devoted to *set/reset* constructions
- Default values can be used to guarantee that the signal always gets a new value

```
always begin
    @(a or b or x or y);
    s=0;
    if (x==1)      s=a;
    else if (y==1) s=b;
end
```

```
always begin
    @(a or b or x or y);
    s='bx;
    if (x==1)      s=a;
    else if (y==1) s=b;
end
```



Latches & flip-flops

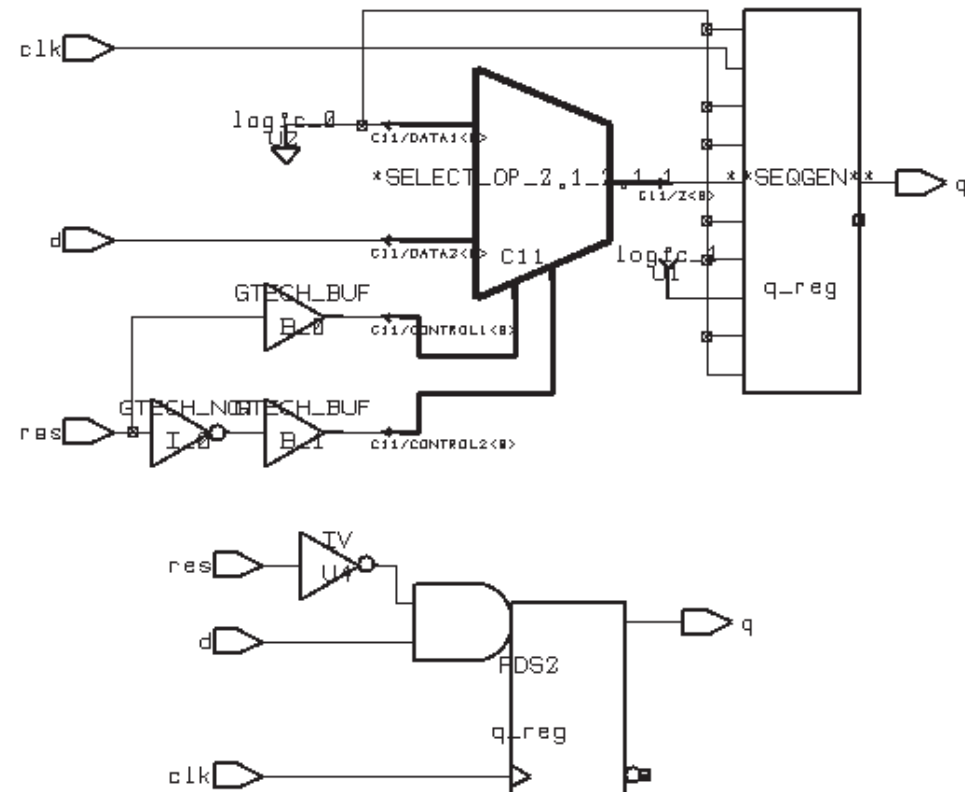
```
always @(clk or d) if (clk) q = d;    /* Latch */
```

```
always @(posedge clk) q = d;          /* Flip-flop */
always @(posedge clk) q <= d;
```

- synchronous reset

```
always @(posedge clk)
  if (res==1) q = 0;
  else      q = d;
```

```
always begin
  @(posedge clk);
  if (res==1) q = 0;
  else      q = d;
end
```

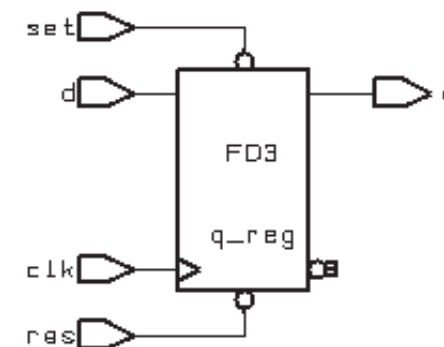


Flip-flops

- asynchronous reset

```
always
  @(posedge res or
    posedge clk)
  if (res==1) q = 0;
  else      q = d;
```

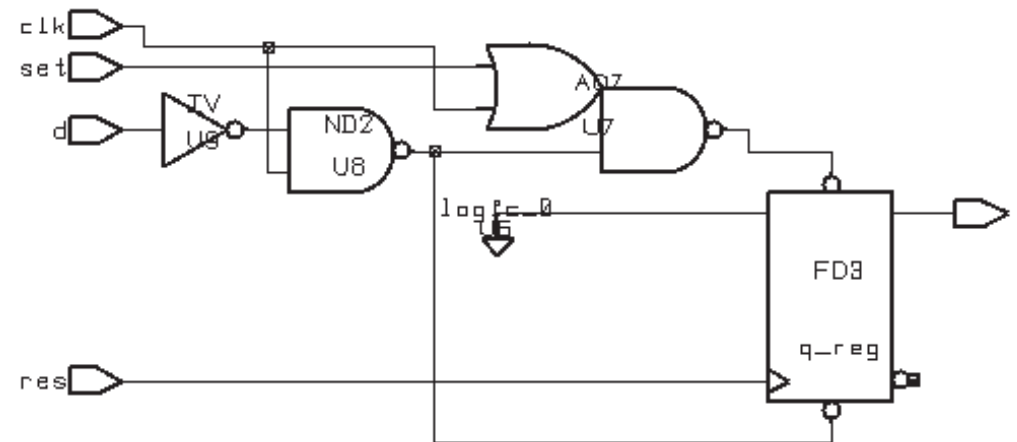
```
always
  @(negedge res or
    negedge set or
    posedge clk)
  if (res==0)      q = 0;
  else if (set==0) q = 1;
  else            q = d;
```



Flip-flops

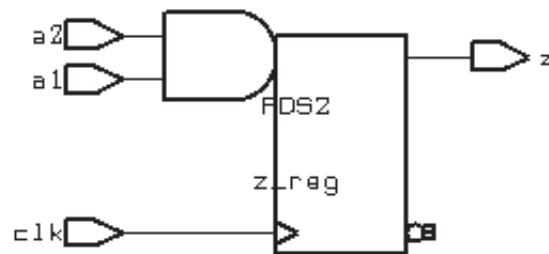
- asynchronous reset - the order of signals!

```
always @(posedge res or posedge set or posedge clk)
  if (clk==1)      q = d;
  else if (set==1) q = 1;
  else             q = 0;
```

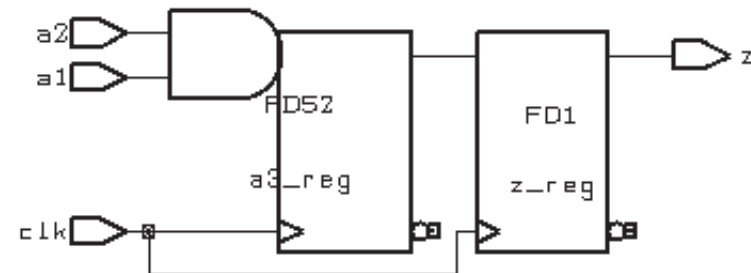


Blocking versus non-blocking

```
module sig_var_b (clk, a1, a2, z);
  input clk, a1, a2;
  output z; reg z; reg a3;
  always @(posedge clk) begin
    a3 = a1 & a2;
    z  <= a3;
  end
endmodule // sig_var_b
```



```
module sig_var_n (clk, a1, a2, z);
  input clk, a1, a2;
  output z; reg z; reg a3;
  always @(posedge clk) begin
    a3 <= a1 & a2;
    z  <= a3;
  end
endmodule // sig_var_n
```

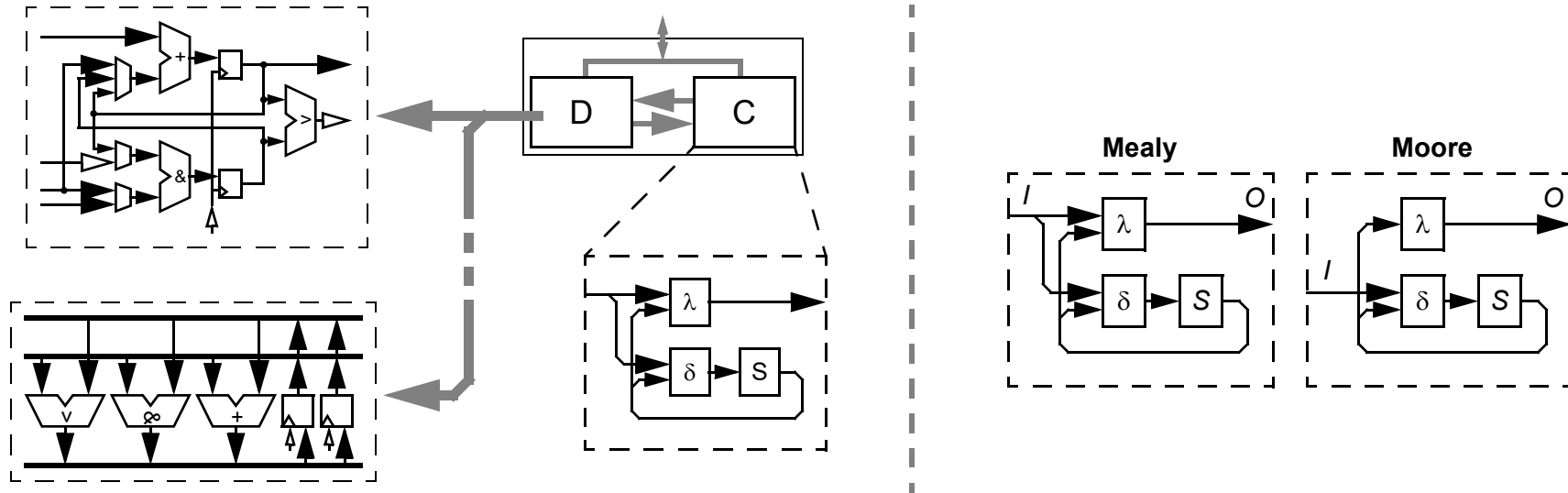


Compare – signal versus variable in VHDL

```
process (CLK)
  variable A3 : BIT;
begin
  if CLK'event and CLK='1' then
    A3 := A1 and A2;
    Z  <= A3;
  end if;
end process;
```

```
signal A3 : BIT;
-- ...
process (CLK) begin
  if CLK'event and CLK='1' then
    A3 <= A1 and A2;
    Z  <= A3;
  end if;
end process;
```

Data-part, control-part & FSM



- **one unit – one process**
 - **functional units – combinational processes** [all inputs in the sensitivity list]
 - **storage units – clocked processes** [activation at clock edge]
- **FSM: $M = (S, I, O, \delta, \lambda)$ – process per block**
- **Three processes – (1) transition function, (2) output function, (3) state register**
- **Two processes – (1) merged transition and output functions, (2) state register** [Mealy]
- **One process – buffered outputs!** [Moore]

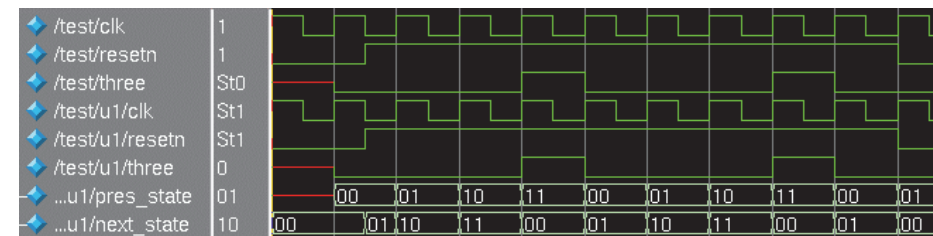
FSM - description styles

Three processes (modulo-4 counter)

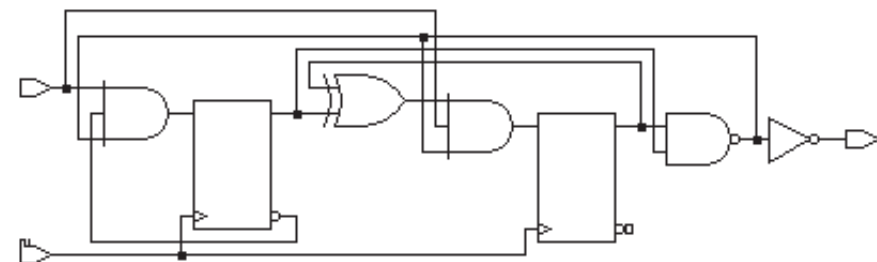
```

module counter03 ( clk, resetn, three );
    input clk, resetn;
    output three;    reg three;
    reg [1:0] pres_state, next_state;
    always @(posedge clk) // State memory
        pres_state <= next_state;
    // Next state function
    always @(resetn or pres_state) begin
        if (resetn==0) next_state = 0;
        else case (pres_state)
            0, 1, 2: next_state = pres_state + 1;
            3:      next_state = 0;
        endcase
    end
    // Output function
    always @(pres_state)
        if (pres_state==3) three = 1;
        else                 three = 0;
endmodule

```



23 gates / 4.36 ns



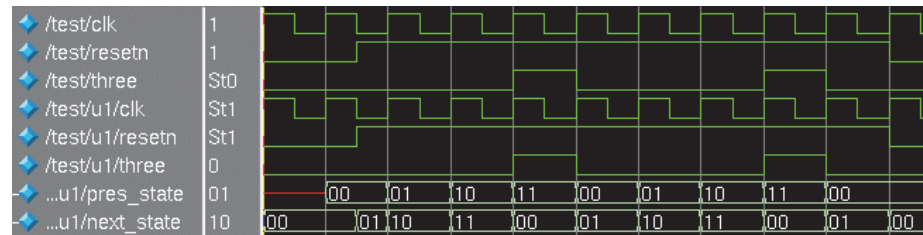
FSM - description styles

Two processes (modulo-4 counter)

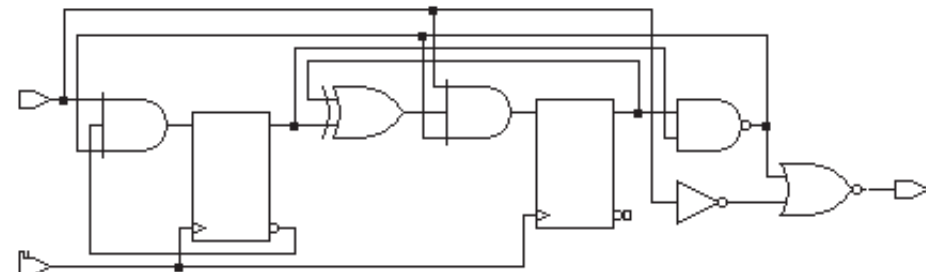
```

module counter03 ( clk, resetn, three );
    input clk, resetn;
    output three;    reg    three;
    reg [1:0] pres_state, next_state;
    always @(posedge clk) // State memory
        pres_state = next_state;
    // Next state & output functions
    always @(resetn or pres_state) begin
        three = 0;
        if (resetn==0) next_state = 0;
        else
            case (pres_state)
                0, 1, 2: next_state = pres_state + 1;
                3: begin next_state = 0;  three = 1;  end
            endcase
    end
endmodule

```



24 gates / 4.36 ns



FSM - description styles

One process (modulo-4 counter)

```

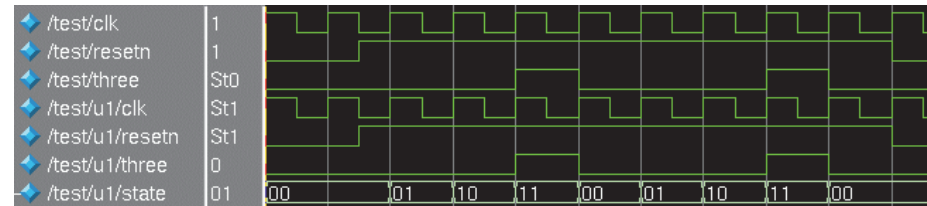
module counter03 ( clk, resetn, three );
  input clk, resetn;
  output three;    reg three;
  reg [1:0] state;
  always @(posedge clk) begin
    three = 0;
    if (resetn==0) state = 0;
    else case (state)
      0, 1:    state = state + 1;
      2: begin state = state + 1; three = 1; end
      3:    state = 0;
    endcase
  end
endmodule

```

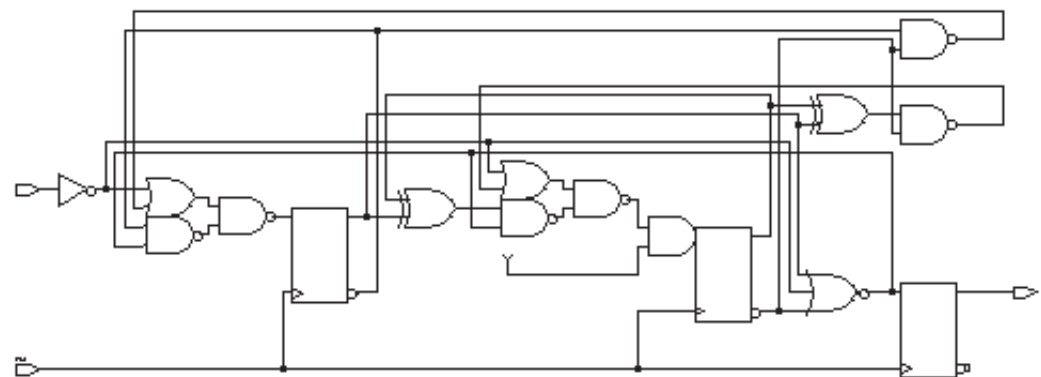
```

// Another version
// to begin the always block
always begin @(posedge clk);
  three = 0; // and so on...

```



38 gates / 5.68 ns

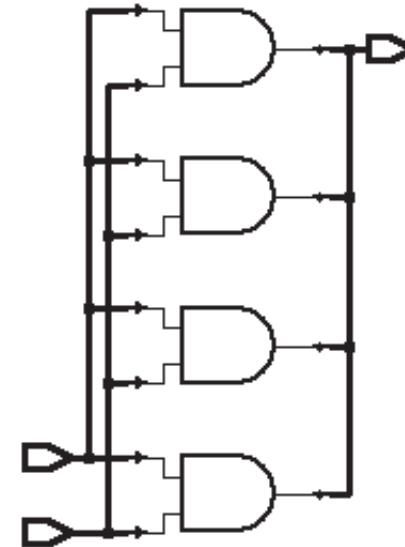


for-loop versus while-loop?

- Is tool dependent!
 - Design Vision (Synopsys) & ISE (Xilinx): *for* - parallel, *while* - parallel
 - No multiple waits!

```
always @(a or b) begin
  for (i=0;i<4;i=i+1)
    x[i] = a[i] & b[i];
end
```

```
always @(a or b) begin
  i = 0;
  while (i<4) begin
    x[i] = a[i] & b[i];
    i = i + 1;
  end
end
```



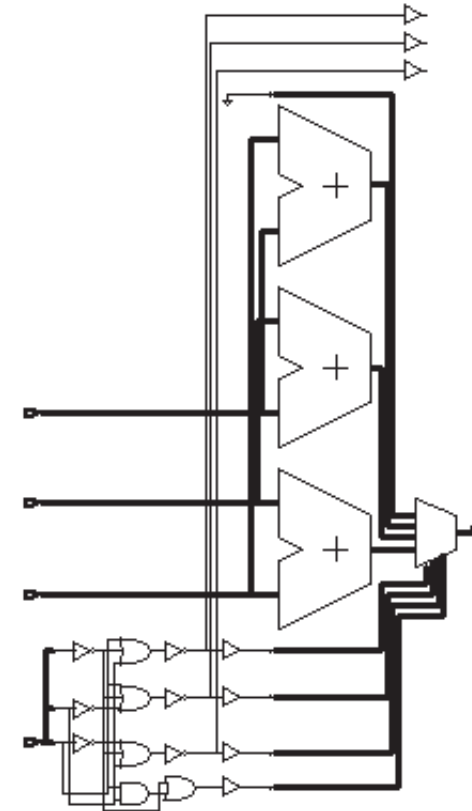
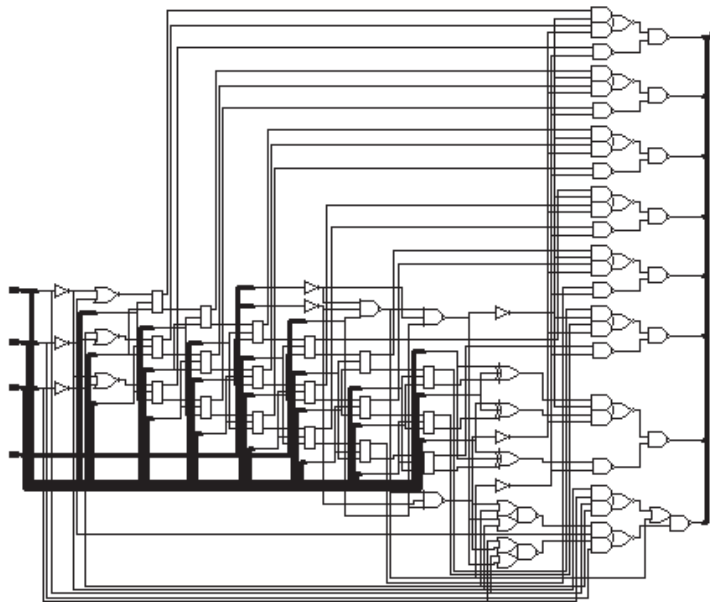
Behavioral RTL vs. “pure” RTL

```

module test (a, b, c, x, o);
  input [7:0] a, b, c;
  input [2:0] x;
  output [7:0] o; reg [7:0] o;
  always @(a or b or c or x)
    if      (x==2)  o <= a+b;
    else if (x==3)  o <= a+c;
    else if (x==6)  o <= b+c;
    else           o <= 0;
endmodule // test

```

Verilog



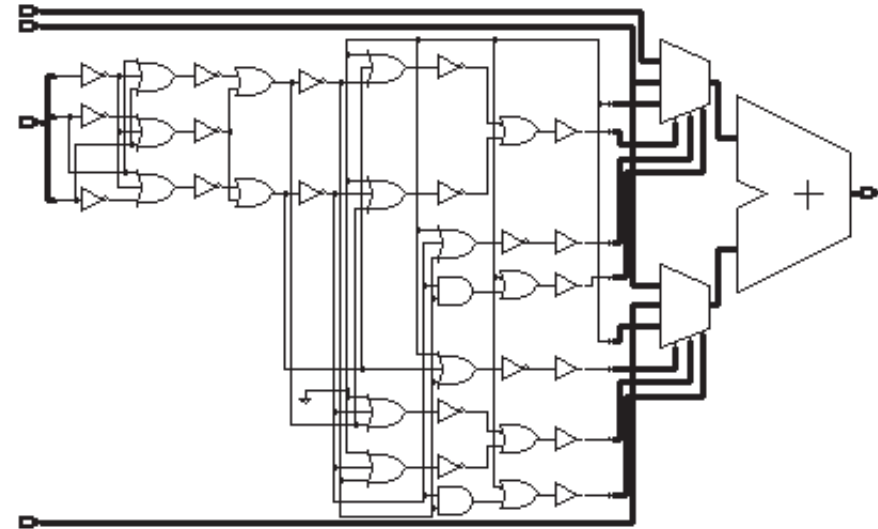
220 gates / 11.57 ns

Behavioral RTL vs. “pure” RTL

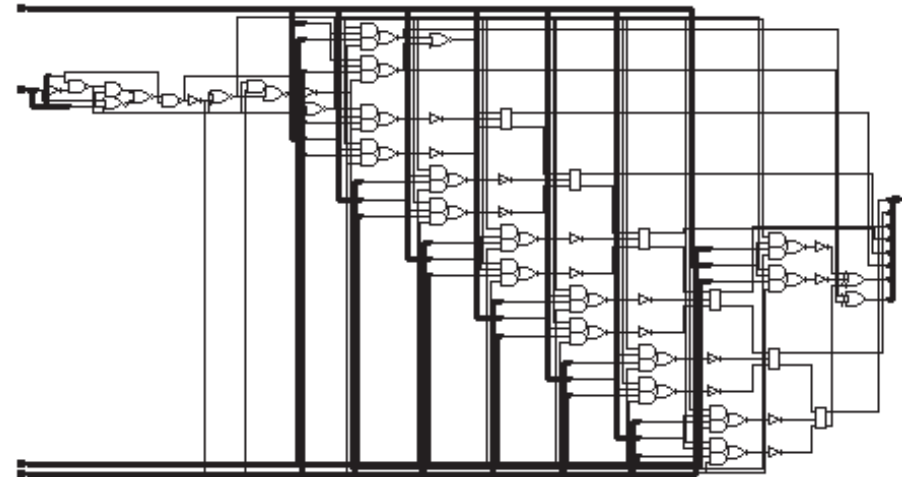
```

module test (a, b, c, x, o);
  input [7:0] a, b, c;
  input [2:0] x;
  output [7:0] o;
  reg [7:0] a1, a2;
  reg [2:0] dc;
  always @(x)
    if (x==2) dc = 1;
    else if (x==3) dc = 2;
    else if (x==6) dc = 3;
    else dc = 0;
  always @(a or b or dc)
    if (dc==1) a1 = a;
    else if (dc==2) a1 = b;
    else if (dc==3) a1 = c;
    else a1 = 0;
  always @(b or c or dc)
    if (dc==1) a2 = b;
    else if (dc==2) a2 = c;
    else if (dc==3) a2 = c;
    else a2 = 0;
  assign o = a1+a2;
endmodule // test

```



117 gates / 19.2 ns



Adder / Subtractor

```
module add_sub (a, b, x, o);
    input [7:0] a, b;
    input x;
    output [7:0] o;
    assign o = x==0 ? a+b : a-b;
endmodule // add_sub
```

Adder & subtracter!!!

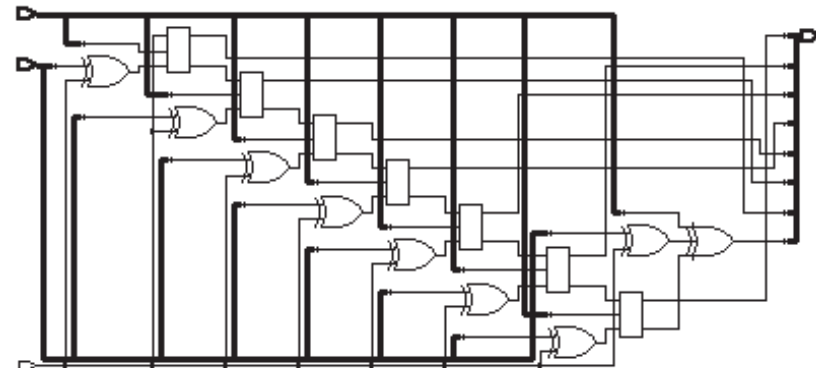
145 gates / 11.64 ns

```
module add_sub (a, b, x, o);
    input [7:0] a, b;
    input x;
    output [7:0] o;
    wire t;

    assign {o,t} = {a,1'b1} +
        ( x==0 ? {b,1'b0} : {~b,1'b1} );

endmodule // add_sub
```

87 gates / 12.45 ns

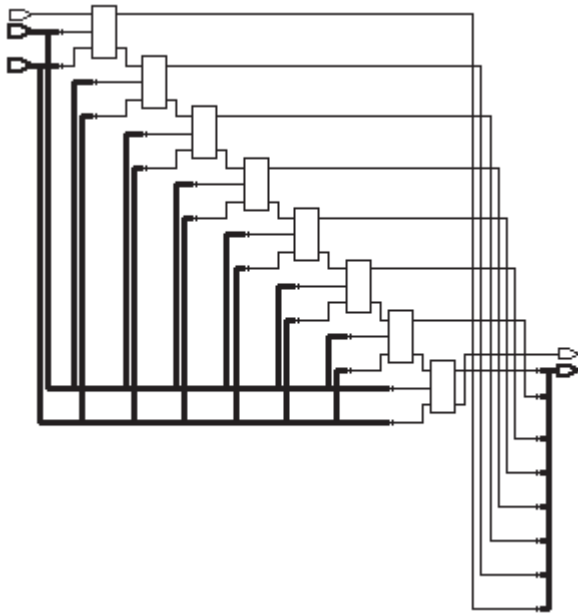


Adders & Subtractors

- Adding “carry-in” & “carry-out”?

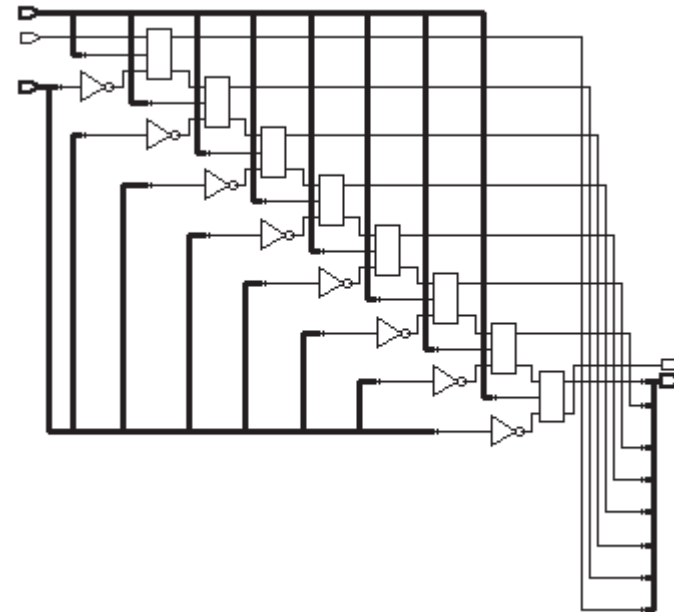
```
assign {co,o,t} = {1'b0,a,1'b1} +
                {1'b0,b,ci};
```

64 g. / 10.66 ns [60 g. / 10.08 ns w/o ci/co]



```
assign {co,o,t} = {1'b0,a,1'b1} +
                {1'b0,~b,ci};
```

72 g. / 10.62 ns [66 g. / 10.35 ns w/o ci/co]



Multiple wait statements

- HDL semantics must be preserved
 - different interpretations possible
- Distributing operations over multiple clock steps

- **Algorithm**

- Inputs: a, b, c, d
- Output: x
- Coefficients: c1, c2
- $x = a + b \cdot c1 + c \cdot c2 + d$
- Timing constraint - 3 clock periods

```
reg [15:0] av, bv, cv, dv;
```

```
always begin
```

```
    av=a; bv=b; cv=c; dv=d;
```

```
    @(posedge clk);
```

```
    @(posedge clk);
```

```
    x = av + bv * c1 + cv * c2 + dv;
```

```
    @(posedge clk);
```

```
end
```

Multiple wait statements

- Behavioral interpretation may lead to an unoptimal solution

```
reg [15:0] av, bv, cv, dv;

always begin
    av=a; bv=b; cv=c; dv=d;
    @(posedge clk);
    @(posedge clk);
    x = av + bv * c1 + cv * c2 + dv;
    @(posedge clk);
end
```

2 multipliers & 3 adders

```
reg [15:0] av, bv, cv, dv, r1, r2;

always begin
    av=a; bv=b; cv=c; dv=d;
    r1 = av + dv;    r2 = bv * c1;
    @(posedge clk);
    r1 = r1 + r2;    r2 = cv * c2;
    @(posedge clk);
    x = r1 + r2;
    @(posedge clk);
end
```

1 multiplier & 1 adder

*Behavioral Synthesis
(High-Level Synthesis)*