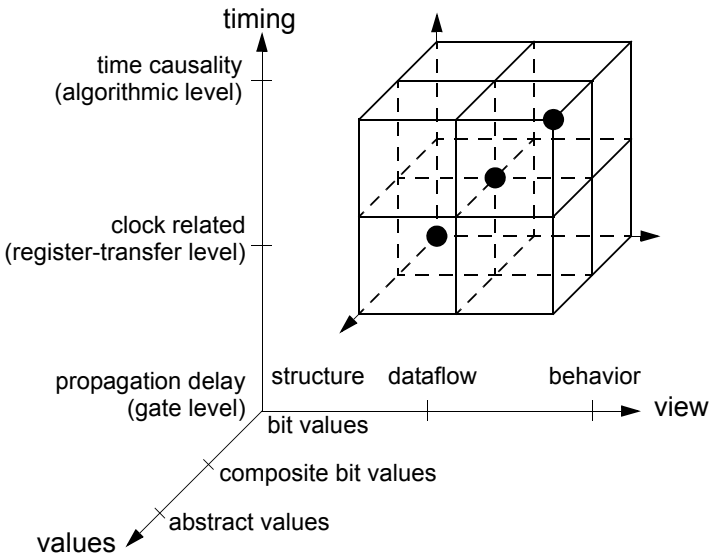
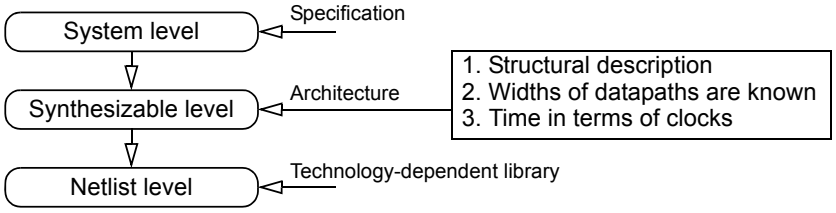
HDL Design Cube



Synthesis and VHDL

- **VHDL LRM defines only simulation semantics of the language.**
- **LRM – Language Reference Manual**



- **Synthesis restrictions:**
  - **the lack of maturity of synthesis tools**
  - **the state-of-art in synthesis targets RTL synthesis only**
  - **certain VHDL features are simply not synthesizable**
- **The same applies to Verilog...**

Synthesis style

- Delay expressions (*after* clauses, *wait for* statements are ignored)
- Certain restrictions on the writing of process statement occur
- Only a few types are allowed
  - integer, enumerated, e.g., bit, bit_vector, signed
- Type conversion and resolution functions are not interpreted
- Description is oriented towards synchronous styles with explicit clocks
- Types: enumeration, integer, one-dimensional array, record

```
type WORD is array (31 downto 0)   of BIT;
type RAM  is array (1023 downto 0) of WORD;
```

- In record, an item address becomes hardware coded
- !!! Time type is not supported !!!
- No explicit or default initialization
- Parenthesis in expressions have effect on HW generation
- Some arithmetic operations are supported partially only

Sensitivity list

- Equivalent processes:

```
process (A, B, C)
  ...
begin
  ...
end process;
```

==

```
process
  ...
begin
wait on A, B, C;
  ...
end process;
```

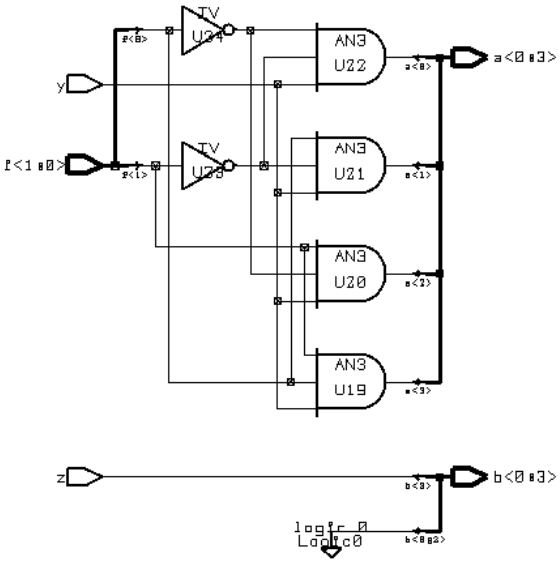
~~

```
process
  ...
begin
  ...
wait on A, B, C;
end process;
```

- Some synthesizers support only sensitivity list for combinational logic
- 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
- Process with multiple synchronization points, i.e. several wait states, infer memorization – FSM

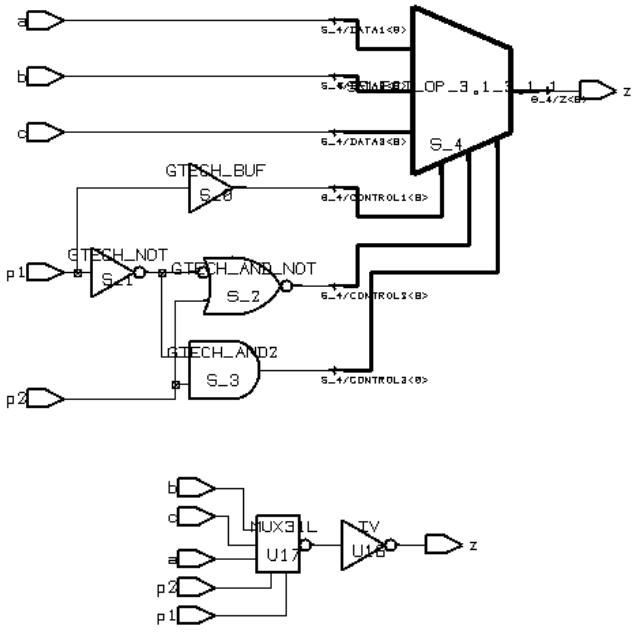
Assignment statement synthesis

```
  signal A,B: BIT_VECTOR(0 to 3);
  signal I: INTEGER range 0 to 3;
  signal Y,Z: BIT;
  -- . . .
process ( I, Y, Z ) begin
  A<="0000";
  B<="0000";
  A(I)<=Y;  -- Computable index
  B(3)<=Z;  -- Constant index
end process;
```

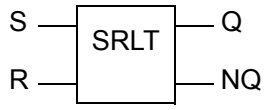


if statement synthesis

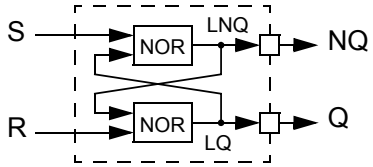
```
  signal A,B,C,P1,P2,Z: BIT;
  -- . . .
process (P1,P2,A,B,C) begin
  if (P1 = '1') then
      Z <= A;
  elsif (P2 = '0') then
      Z <= B;
  else
      Z <= C;
  end if;
end process;
```



SR latch




```
use WORK.CHECK_PKG.all;
entity SRLT is
  port ( S, R:  in bit;
         Q, NQ: out bit );
begin
  NOT_AT_THE_SAME_TIME(S,R);
end SRLT;
```




```
architecture A1 of SRLT is
  signal LQ:  bit := '1';
  signal LNQ:  bit := '0';
begin
  LNQ <= S nor LQ;
  LQ  <= R nor LNQ;
  Q   <= LQ;
  NQ  <= LNQ;
end A1;
```

- **NB! Asynchronous feed-back is temporarily cut by synthesizers...**

Combinational circuit

- **A process is combinational, i.e. does not infer memorization, if:**
  - **the process has an explicit sensitivity list or contains a single synchronization point (waiting for changes on all input values); [1)]**
  - **no local variable declarations, or variables 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.**

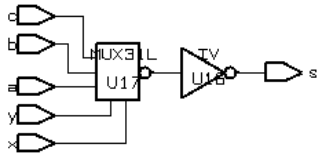
[1)] waiting on a clock signal, e.g., " wait on clk until clk='1'; ", implies buffered outputs (FF-s)

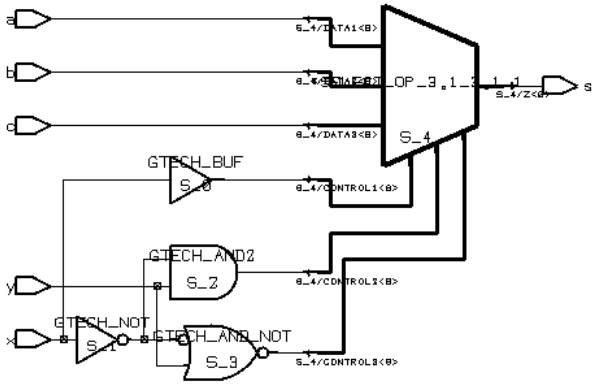
[2)] interpretation may differ from tool to tool

Complex assignments

- **No memory:**

```
S <= A when X='1' else B when Y='1' else C;

process (A, B, C, X, Y) begin
  if    X='1' then   S <= A;
  elsif Y='1' then   S <= B;
  else               S <= C;
  end if;
end process;
```

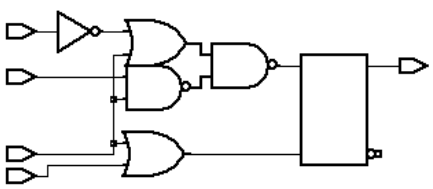


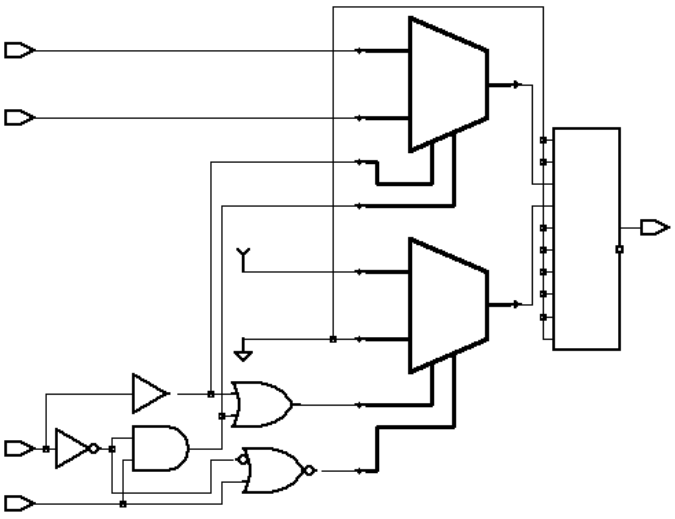


Complex assignments

- **Memory element generated:**

```
process (A, B, X, Y) begin
  if    X='1' then  S <= A;
  elsif Y='1' then  S <= B;
  end if;
end process;
```

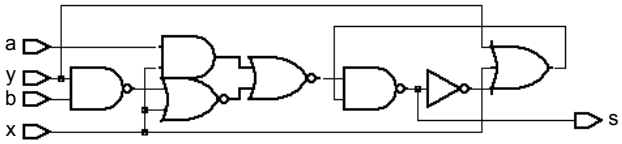


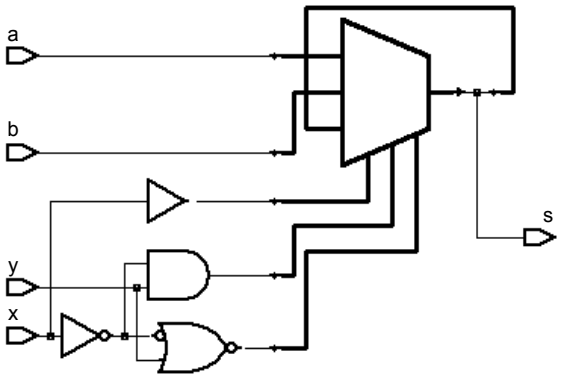


Complex assignments

- **Memory element generated (#2):**

```
S <= A when X='1' else B when Y='1' else S;

process (A, B, X, Y) begin
  if    X='1' then   S <= A;
  elsif Y='1' then   S <= B;
  else               S <= S;
  end if;
end process;
```

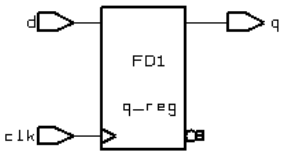




Flip-flops

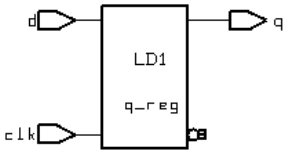
- **Process**

```
P1_FF: process (CLK)
begin
    if CLK='1' and CLK'event then
        Q<=D;
    end if;
end process P1_FF;

P2_FF: process
begin
    wait on CLK until CLK='1';
    Q<=D;
end process P2_FF;
```



Latch vs. Flip-flop?

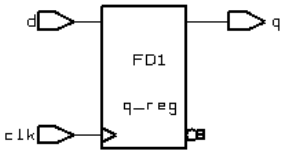
```
P1_L: process (CLK, D) begin
    if CLK='1' then    Q <= D;
    end if;
end process P1_L;

P2_FL: process (CLK) begin
    if CLK='1' then    Q<=D;
    end if;
end process P2_FL;

P1_FF: process (CLK) begin
    if CLK='1' and
       CLK'event then  Q<=D;
    end if;
end process P1_FF;
```



- **Simulation OK but not synthesis!**

```
-- Warning: Variable 'd' is being read
-- in routine .. line .. in file '..',
-- but is not in the process sensitivity
-- list of the block which begins
-- there.   (HDL-179)
```

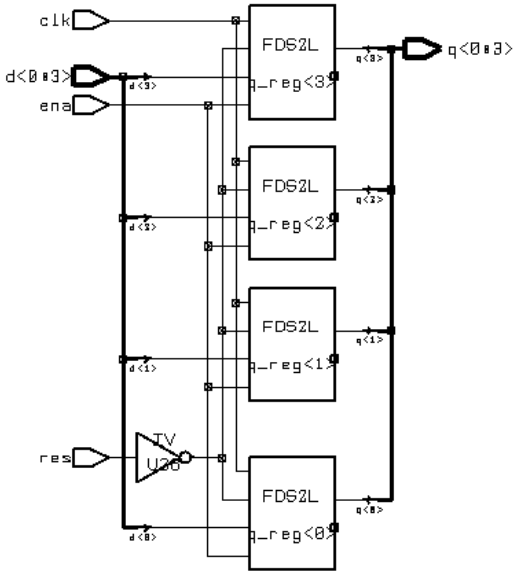
- **Result - latch**



Flip-flops

- **Process + reset & enable**

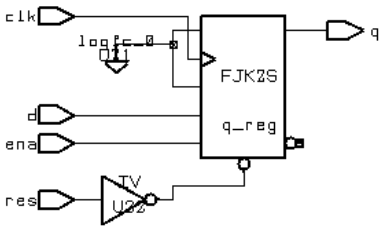
```
P3_FF: process (CLK)
begin
  if CLK='1' and CLK'event then
    if    RES='1' then
      Q<=(others=>'0');
    elsif ENA='1' then
      Q<=D;
    end if;
  end if;
end process P3_FF;
```



Flip-flops

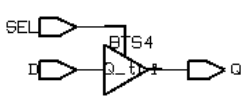
- Process + asynchronous reset & enable

```
P4_FF: process (RES,CLK)
begin
  -- asynchronous reset
  if    RES='1' then  Q <= '0';
  elsif CLK='1' and CLK'EVENT then
    if  ENA='1' then  Q <= D;
    end if;
  end if;
end process P4_FF;
```



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. CLK'event and CLK='1', in the process
  - usually, only one edge expression is allowed per process
    - different processes 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 <= D when SEL='1' else 'Z';



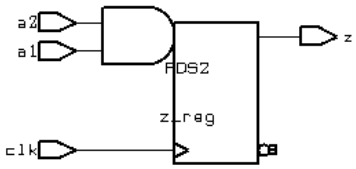
  - the target signal will infer a latch (latches) when the target signal is not assigned with a value in every conditional branch, and the edge expression is missing
  - a combinational circuit will be synthesized otherwise

- It is a good practice to isolate flip-flops, latches and three-state buffers inferences to ensure design correctness

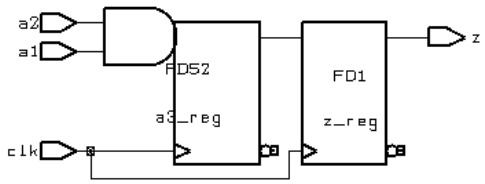
Signal versus variable

- The hardware resulting from synthesis of variables or signals differs: either nothing, wires, or memory elements

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




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



Arithmetics

- Overloaded arithmetic operations:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

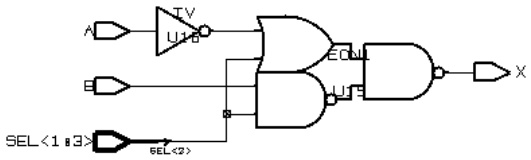
- Sources of the named packages are in the directory: $SYNOPSYS/packages/IEEE/src
  - $SYNOPSYS/packages is the root directory for all Synopsys packages
- Be careful with '*', '/', '**' - extremely chip area consuming
  - Safe in some special cases - multiplication by power of two
- Use parenthesis to group a set of gates

Initial values

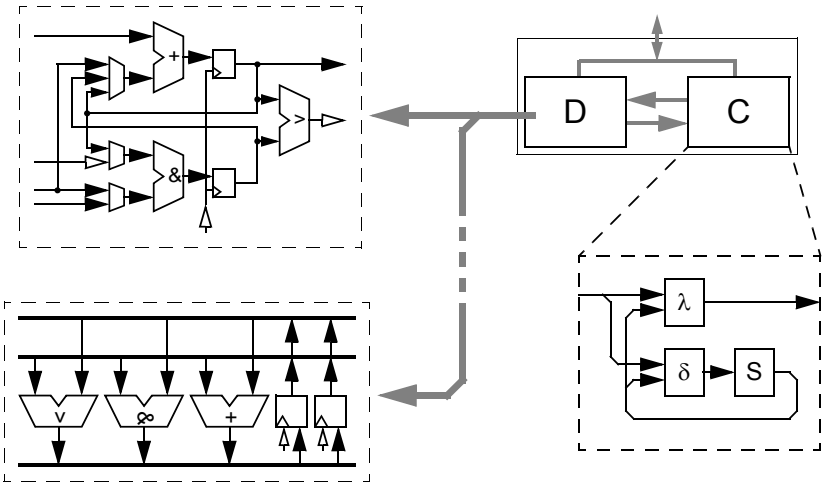
- The default values inherited from type or subtype definitions
- The explicit initialization that is given when the object is declared
- A value assigned using a statement at the beginning of a process
- Only the last case is supported by synthesis tools.
- Usually, a part of the synthesizable code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

Don’t care values and synthesis

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
architecture C of DONT_CARE is
  signal SEL: STD_LOGIC_VECTOR(1 TO 3);
  signal A, B, X: STD_LOGIC;
begin
  process (A, B, SEL) begin
    case SEL is
      when ”001” =>    X <= A;
      when ”010” =>    X <= B;
      when others =>   X <= '-';
    end case;
  end process;
end C;
```



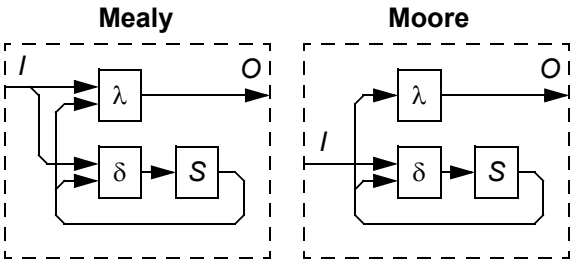
Data-part & control-part



- one unit – one process
  - functional units – combinational processes [all inputs in the sensitivity list]
  - storage units – clocked processes [activation at clock edge]

FSM in VHDL

- **FSM: M = (S, I, O, δ, λ)**
  - **S: set of states**
  - **I: set on inputs**
  - **O: set of outputs**
  - **δ: transition function - $\delta: S \times I \rightarrow S$**
  - **λ: output function - $\lambda: S \times I \rightarrow O$**



- **How many processes?**
  - **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 as a single process

```
-- RESET is the asynchronous reset, CLK is the clock
-- STATE is a variable (or signal) memorizing the current state
process(RESET,CLK)
begin
    if RESET='1' then          -- asynchronous reset
        STATE <= S_INIT;
    elsif CLK='1' and CLK'EVENT then
        case STATE is
            when S_INIT => if I0='1' then STATE <= S5; end if;
            when ... => ...
        end case;
    end if;
end process;
```

Three process FSM

- **storage elements, transition function & output function**

```
architecture B of FSM is
    type TYPE_STATE is (S_INIT,S1,...Sn);
    signal CURRENT_STATE, NEXT_STATE : TYPE_STATE;
begin
  P_STATE: process begin -- sequential process
    wait until CLK'EVENT and CLK='1';
    if RESET ='1' then CURRENT_STATE <= S_INIT;
    else               CURRENT_STATE <= NEXT_STATE; end if;
  end process P_STATE;
```

continue...

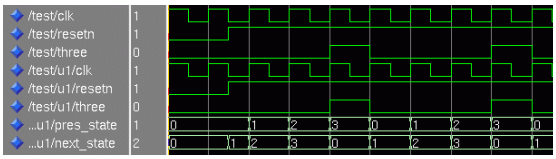
Three process FSM

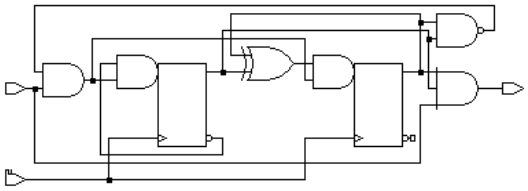
```
  P_NEXT_STATE: process (I0, ..., CURRENT_STATE) begin
    NEXT_STATE <= CURRENT_STATE;
    case CURRENT_STATE is
      when S_INIT => if I0='1' then NEXT_STATE <= S5; end if;
      when ... => ...
    end case;
  end process P_NEXT_STATE;

  P_OUTPUTS: process (CURRENT_STATE) begin
    case CURRENT_STATE is
      when S_INIT => O <= '0';
      when ... => ...
    end case;
  end process P_OUTPUTS;
end B;
```

FSM #2 – description styles & synthesis

Two processes (modulo-4 counter)

```
library IEEE; use IEEE.std_logic_1164.all;
entity counter03 is
  port ( clk: in bit;
         resetn: in std_logic;
         three: out std_logic );
end entity counter03;
architecture fsm2 of counter03 is
  subtype state_type is integer range 0 to 3;
  signal pres_state, next_state: state_type := 0;
begin
  process (clk) begin -- State memory
    if clk'event and clk = '1' then
      pres_state <= next_state;
    end if;
  end process;
-- Next state & output functions
  process (resetn, pres_state) begin
    three <= '0';
    if resetn='0' then    next_state <= 0;
    else
      case pres_state is
      when 0 to 2 => next_state <= pres_state + 1;
      when 3 => next_state <= 0;  three <= '1';
      end case;
    end if;
  end process;
end architecture fsm2;
```





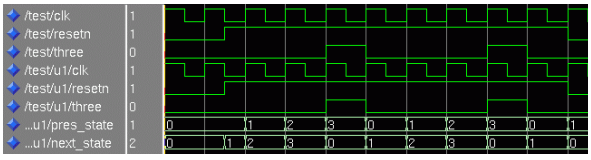
22 gates / 3.70 ns

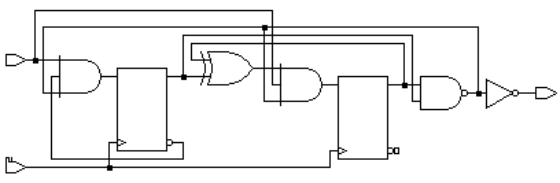
TTÜ1918
Department of Computer Engineering

FSM #2 – description styles & synthesis

Three processes (modulo-4 counter)

```
library IEEE; use IEEE.std_logic_1164.all;
architecture fsm3 of counter03 is
  subtype state_type is integer range 0 to 3;
  signal pres_state, next_state: state_type := 0;
begin
  process (clk) begin -- State memory
    if clk'event and clk = '1' then
      pres_state <= next_state;
    end if;
  end process;

  -- Next state function
  process (resetn, pres_state) begin
    if resetn='0' then    next_state <= 0;
    else
      if pres_state=3 then    next_state <= 0;
      else    next_state <= pres_state + 1;
      end if;
    end if;
  end process;
  -- Output function
  process (resetn, pres_state) begin
    if pres_state=3 then    three <= '1';
    else                    three <= '0';
    end if;
  end process;
end architecture fsm3;
```



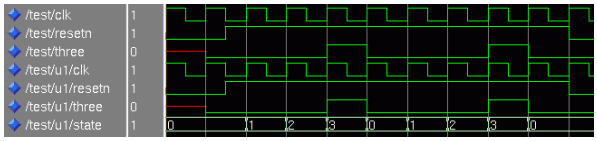


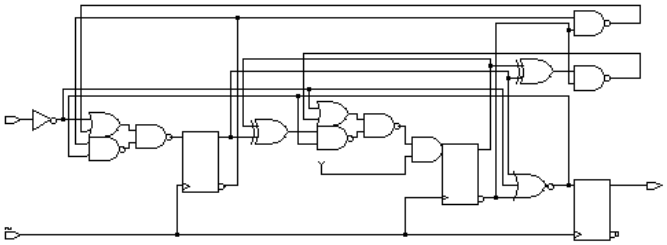
23 gates / 4.36 ns

FSM #2 – description styles & synthesis

One process (modulo-4 counter)

```
library IEEE; use IEEE.std_logic_1164.all;
architecture fsm1 of counter03 is
  subtype state_type is integer range 0 to 3;
  signal state: state_type := 0;
begin
  process (clk) begin
    if clk'event and clk = '1' then
      three <= '0';
      if resetn='0' then    state <= 0;
      else
        case state is
        when 0 | 1 => state <= state + 1;
        when 2 => state <= state + 1;  three <= '1';
        when 3 => state <= 0;
        end case;
      end if;
    end if;
  end process;
end architecture fsm1;

// Another version to build the process
  process begin
    wait on clk until clk='1';
    three <= '0';
    if resetn='0' then    state <= 0;
    else
    -- and so on...
```





38 gates / 5.68 ns

Using generics

```
entity AND_N is
    generic (N: POSITIVE);
    port     (DIN: in BIT_VECTOR(1 to N); R: out BIT);
end AND_N;

architecture A1 of AND_N is
    signal INTER: BIT_VECTOR(1 to N);
begin
    INTER(1) <= DIN(1);
    L: for I in 1 to N-1 generate
        INTER(I+1) <= DIN(I+1) and INTER(I);
    end generate;
    R <= INTER(N);
end A1;

C1: AND_N generic map (N=>12) port map(IN_DATA, OUT_DATA);
```

for-loop versus while-loop?

- **May be tool dependent!**
  - Design Compiler (Synopsys): *for* - parallel, *while* - sequential
  - Leonardo (Mentor Graphics): depending on the timing constructions

- **for-loop**
  - parallel implementation
  - no timing control (wait) in the loop body

```
for i in 0 to 7 loop
  x(i) := a(i) and b(i);
end loop;
```

- **while-loop**
  - sequential implementation
  - timing control (wait) required in the loop body

```
i := 0;
while i<7 loop
  data(i) := in_port;
  wait on clk until clk='1';
  i := i + 1;
end loop;
```


Multiple wait statements

- **VHDL 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*c1 + c*c2 + d
  - Timing constraint - 3 clock periods

```
process
  variable av, bv, cv, dv: ...;
begin
  av:=a; bv:=b; cv:=c; dv:=d;
  wait on clk until clk='1';
  wait on clk until clk='1';
  x <= av + bv * c1 + cv * c2 + dv;
  wait on clk until clk='1';
end process;
```

Multiple wait statements

- **Behavioral interpretation may lead to an unoptimal solution**

```
process
  variable av, bv, cv, dv: ...;
begin
  av:=a; bv:=b; cv:=c; dv:=d;
  wait on clk until clk='1';
  wait on clk until clk='1';
  x <= av + bv * c1 + cv * c2 + dv;
  wait on clk until clk='1';
end process;
```

2 multipliers & 3 adders

Behavioral Synthesis
(High-Level Synthesis)

```
process
  variable av, bv, cv, dv: ...;
  variable r1, r2: ...;
begin
  av:=a; bv:=b; cv:=c; dv:=d;
  r1 := av + dv;    r2 := bv * c1;
  wait on clk until clk='1';
  r1 := r1 + r2;    r2 := cv * c2;
  wait on clk until clk='1';
  x <= r1 + r2;
  wait on clk until clk='1';
end process;
```

1 multiplier & 1 adder

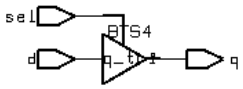
Inserting wait statements

- **VHDL semantics preserved for inputs/outputs**
  - **targeting as-fast-as-possible (AFAP) schedules**
- **16-tap FIR filter**
  - **new input and output data at every rising flank of sys_clk (sampling clock)**
  - **internal clock can be added**
- **How to implement loops?**
  - **1st - in parallel (shift-register)**
  - **2nd - sequentially**
    - **multiply-and-accumulate (MAC)**
    - **ROM for coefficients**

```
process
  variable sum: ...;
  variable buff: ...; -- array (0 to 15)
begin
  for i in 15 downto 1 loop
    buff(i):=buff(i-1);
  end loop;
  buff(0):=data_in;    sum:=0;
  for i in 0 to 15 loop
    sum := sum + buff(i) * coeff(i);
  end loop;
  x <= sum;
  wait on sys_clk until sys_clk='1';
end process;
```

Verilog – 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

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
```

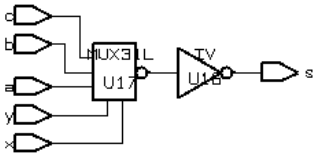
- **In case of single synchronization process there is no need to "remember" at which synchronization point it was stopped –> such behavior does not imply memorization**

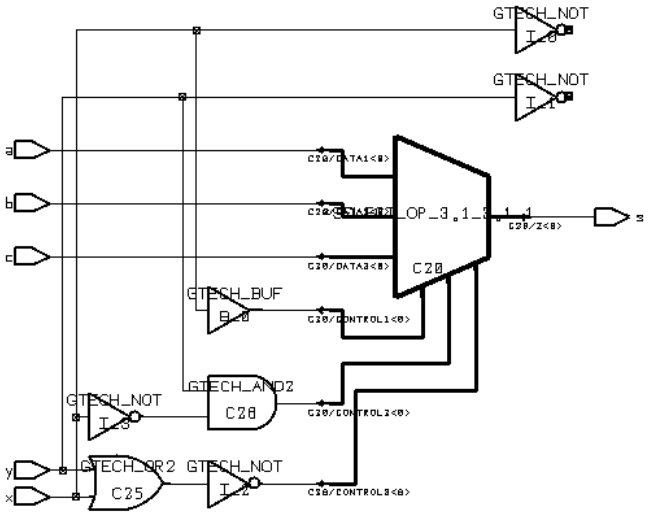
Complex assignments

- **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;
```

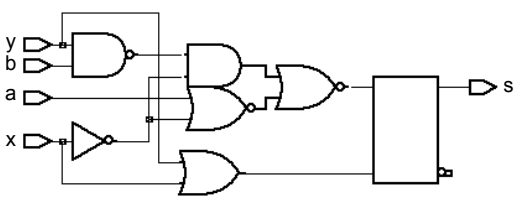


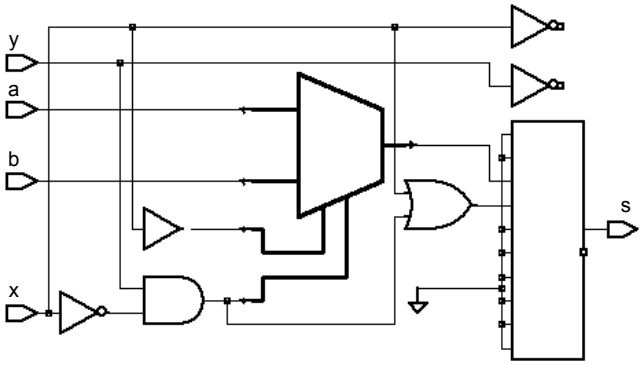


Complex assignments

- **Memory element generated:**

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



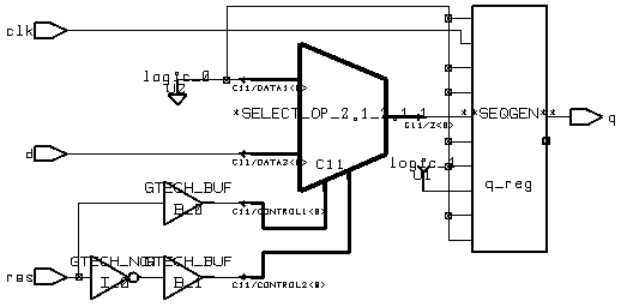


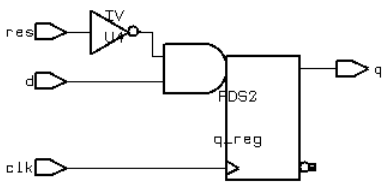
Flip-flops

```
always @(posedge clk) q = d;

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
```

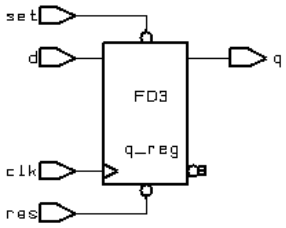




Flip-flops

- asynchronous reset

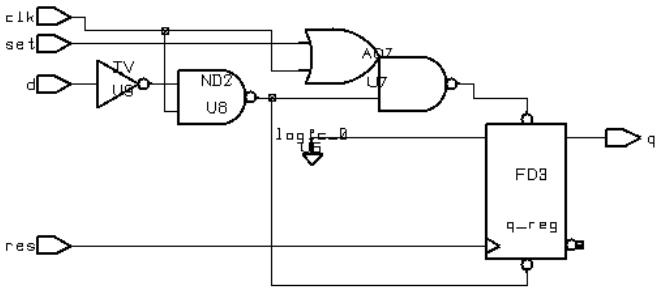
```
always
  @(posedge res or
    posedge clk)
  if (res==1) q = 0;
  else        q = d;

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!

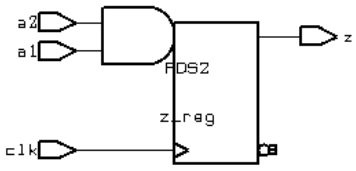
```
always @(posedge res or posedge set or posedge clk)
  if (clk==1)       q = d;
  else if (set==1)  q = 1;
  else              q = 0;
```

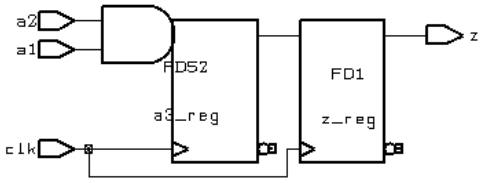


Blocking versus non-blocking

```
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
```



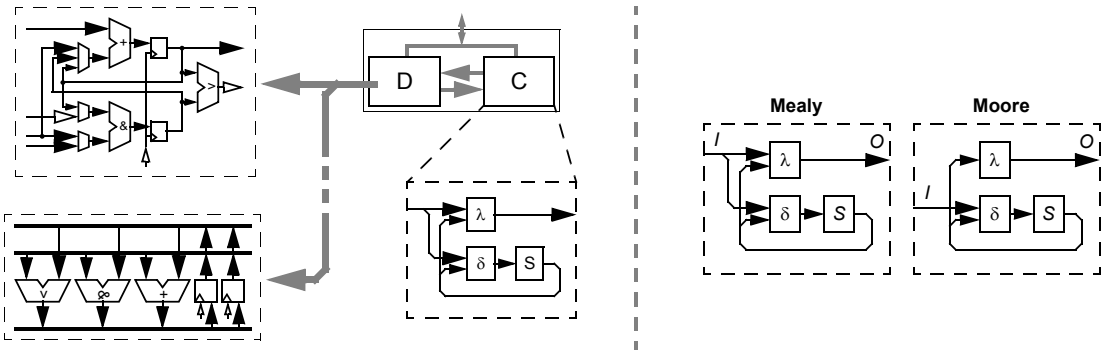


Compare – signal versus variable in VHDL

```
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

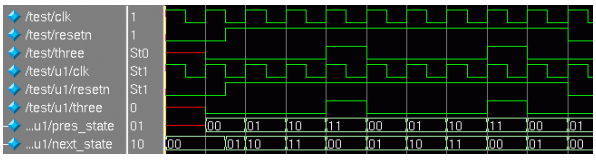


- one unit – one process
  - functional units – combinational processes [all inputs in the sensitivity list]
  - storage units – clocked processes [activation at clock edge]
- FSM: M = (S, I, O, δ, λ) – process per block
- Three processes – (1) transition function, (2) output function, (3) state register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

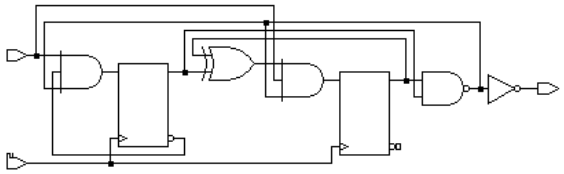
FSM - description styles

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
  always @(pres_state)  // Output function
    if (pres_state==3)  three = 1;
    else                three = 0;
endmodule
```



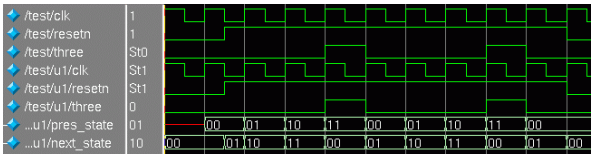
23 gates / 4.36 ns



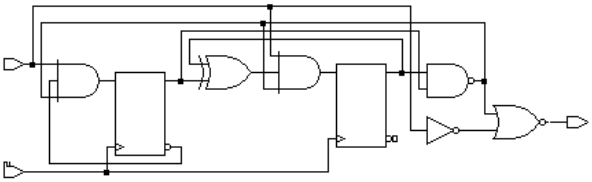
FSM - description styles

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
```



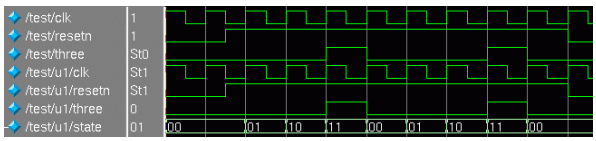
24 gates / 4.36 ns



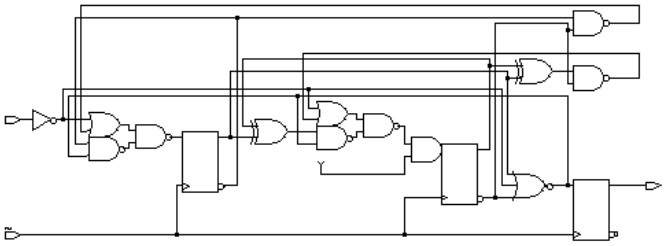
FSM - description styles

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...
```



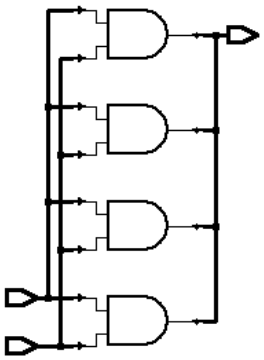
38 gates / 5.68 ns



for-loop versus while-loop?

- **Is tool dependent!**
  - **Design Compiler (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
```



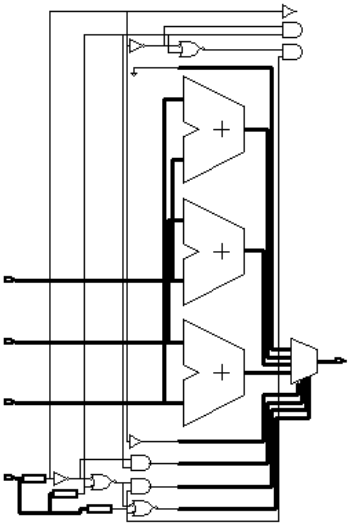
Behavioral RTL vs. “pure” RTL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity  test  is
  port ( a, b, c: in unsigned(7 downto 0);
         x: in unsigned(2 downto 0);
         o: out unsigned(7 downto 0) );
end test;
architecture bhv of test is begin
process (a, b, c, x)
  constant x2: unsigned(2 downto 0):=”010”;
  constant x3: unsigned(2 downto 0):=”011”;
  constant x6: unsigned(2 downto 0):=”110”;
begin
  if    x=x2 then  o <= a+b;
  elsif x=x3 then  o <= a+c;
  elsif x=x6 then  o <= b+c;
  else             o <= (others=>'0');
  end if;
end process;
end architecture bhv;
```

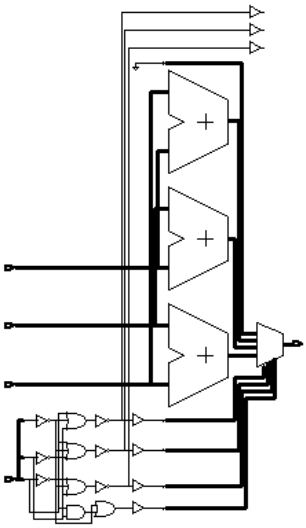
```
module test (a, b, c, x, o);
  input [7:0] a, b, c;
  input [2:0] x;
  output [7:0] o; reg [7:0] o;
  always @(a or b or c or x)
    if      (x==2)  o <= a+b;
    else if (x==3)  o <= a+c;
    else if (x==6)  o <= b+c;
    else            o <= 0;
endmodule // test
```


Behavioral RTL vs. “pure” RTL

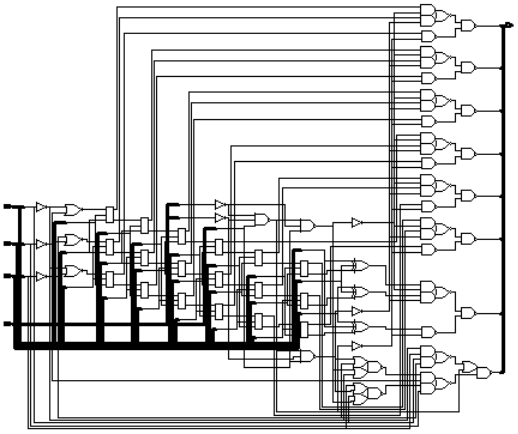
VHDL



Verilog

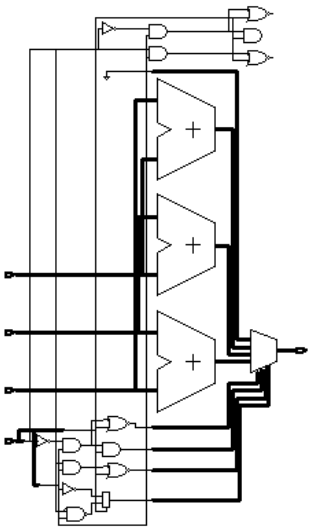


220 gates / 11.57 ns

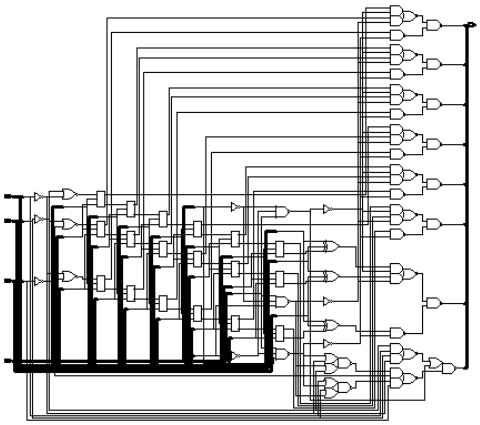


Behavioral RTL vs. “pure” RTL

```
architecture bhv2 of test is begin
process (a, b, c, x) begin
  case x is
  when ”010” =>   o <= a+b;
  when ”011” =>   o <= a+c;
  when ”110” =>   o <= b+c;
  when others =>  o <= (others=>'0');
  end case;
end process;
end architecture bhv2;
```



220 gates / 11.57 ns

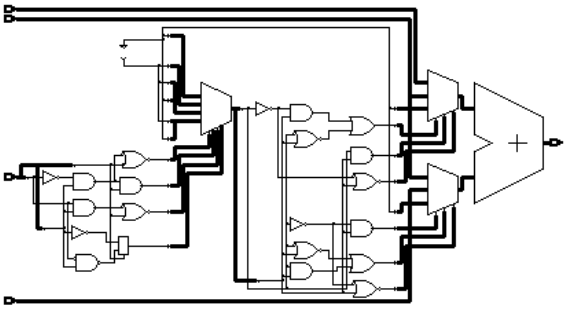


Behavioral RTL vs. “pure” RTL

```
architecture rtl of test is
  signal a1, a2: unsigned(7 downto 0);
  signal dc: unsigned(1 downto 0);
begin
dec: process (x) begin
  case x is
  when ”010” =>   dc <= ”01”;
  when ”011” =>   dc <= ”10”;
  when ”110” =>   dc <= ”11”;
  when others =>  dc <= ”00”;
  end case;
end process dec;
m1: process (a, b, dc) begin
  case dc is
  when ”01” =>    a1 <= a;
  when ”10” =>    a1 <= a;
  when ”11” =>    a1 <= b;
  when others =>  a1 <= (others=>'0');
  end case;
end process m1;
m2: process (b, c, dc) begin
  -- ...
end process m2;
o <= a1 + a2;
end architecture rtl;
```

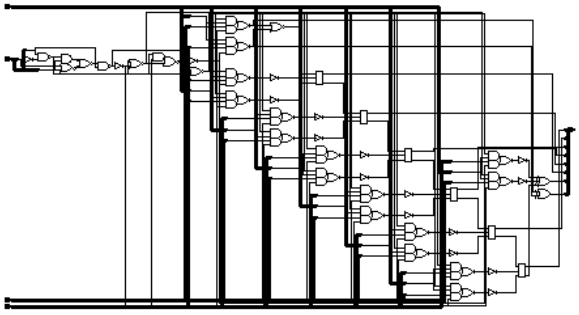
```
module test (a, b, c, x, o);
  input [7:0] a, b, c;
  input [2:0] x;
  output [7:0] o;
  reg [7:0] a1, a2;
  reg [2:0] dc;
  always @(x)
    if      (x==2)  dc = 1;
    else if (x==3)  dc = 2;
    else if (x==6)  dc = 3;
    else            dc = 0;
  always @(a or b or dc)
    if      (dc==1)  a1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

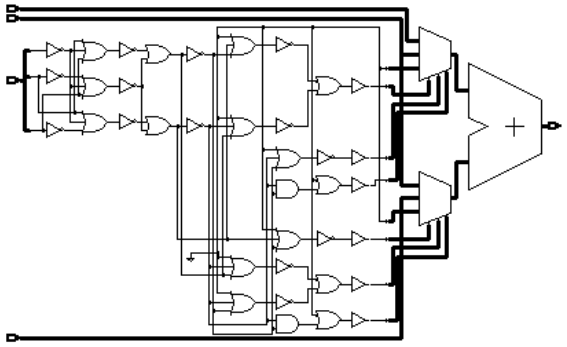
Behavioral RTL vs. “pure” RTL



VHDL

117 gates / 19.2 ns





Verilog

Adder / Subtracter

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity  add_sub  is
  port ( a, b: in unsigned(7 downto 0);
         x: in std_logic;
         o: out unsigned(7 downto 0) );
end add_sub;
architecture bhv of add_sub is begin
process (a, b, x) begin
  if x='0' then  o <= a+b;
  else           o <= a-b;  end if;
end process;
end architecture bhv;

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
```

145 gates / 11.64 ns

```
architecture dfl of test5 is
  signal a1, b1, o1: unsigned(8 downto 0);
begin
  a1 <= a & '1';
  b1 <= b & '0' when x='0' else
    unsigned(not std_logic_vector(b)) &
    '1';
  o1 <= a1+b1;
  o <= o1(8 downto 1);
end architecture dfl;

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

87 gates / 12.45 ns