Coduri sintetizabile

Constructii sintetizabile

ENTITY
- **ARCHITECTURE**
- **CONFIGURATION**
- **PACKAGE**
- **Concurrent signal assignments**
- **PROCESS**
- **SIGNAL**
- **VARIABLE (non-shared)**
- **CONSTANT**
- **IF-ELSE**
- **CASE**
- **Loops (fixed iteration)**
- **Multi-dimensional arrays**
- **PORT**
- **GENERIC (constant)**
- **COMPONENT**

Component & direct instantiation
- **GENERATE**
- **FUNCTION**
- **PROCEDURE**
- **ASSERT (constant false)**
- **WAIT (one per process)**
- **TYPE**
- **SUBTYPE**

Constructii nesintetizabile

- ACCESS
- •ASSERT
- •DISCONNECT
- •FILE
- •GROUP
- •NEW
- •Physical delay types
- •PROTECTED
- •SHARED VARIABLE
- •Signal assignment delays

- Incomplete sensitivity list in combinatorial PROCESS blocks may result in differences between RTL & gate-level simulations
  - Synthesis tool synthesizes as if sensitivity list complete

```
PROCESS (a, b)
  y <= a AND b AND c;
```

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

```
PROCESS (a, b, c)
    y <= a AND b AND c;
```

Correct way for the intended AND logic !

- **IF-ELSE** (like WHEN-ELSE concurrent assignment) structure implies prioritization & dependency
  - Nth clause implies all N-1 previous clauses not true
    - Beware of needlessly "ballooning" logic

Logical Equation

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

  - Consider restructuring IF statements
    - May flatten the multiplexer and reduce logic

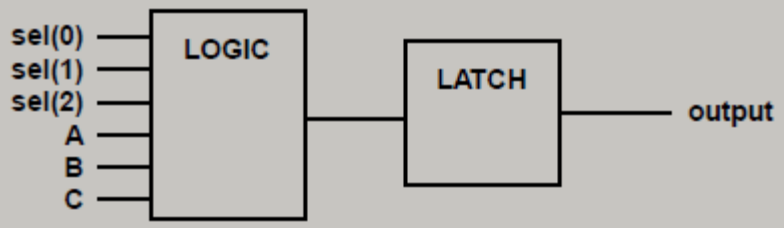
```
IF <cond1> THEN
    IF <cond2> THEN
```

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

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

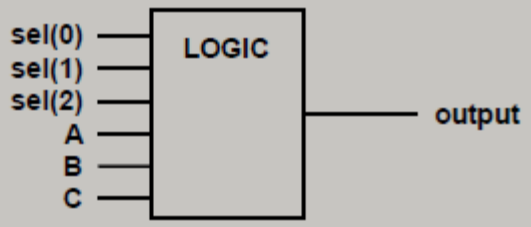
- Combinatorial processes that do not cover all possible input conditions generate latches

```
PROCESS (sel, a, b, c)
BEGIN
      IF sel = "001" THEN
            output <= a;
      ELSIF sel = "010" THEN
            output <= b;
      ELSIF sel = "100" THEN
            output <= c;
      END IF;
END PROCESS;
```



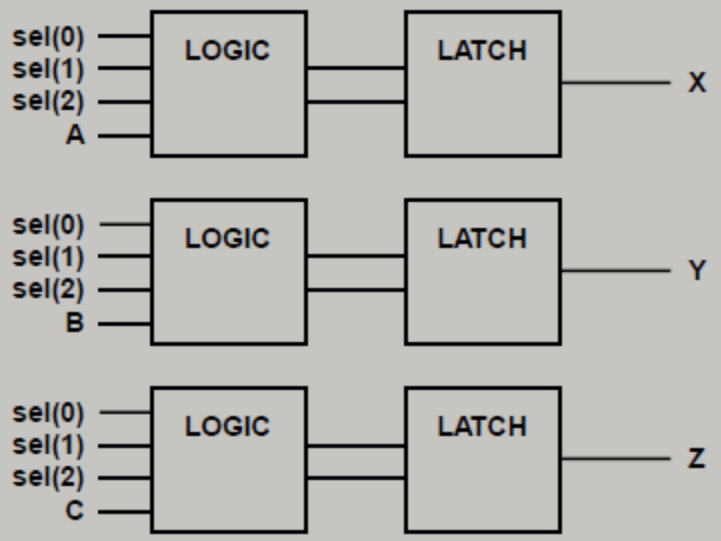
- Close all IF-ELSE structures
  - If possible, assign "don't care's" to else clause for improved logic optimization

```
PROCESS (sel, a, b, c)
BEGIN
    IF sel = "001" THEN
        output <= a;
    ELSIF sel = "010" THEN
        output <= b;
    ELSIF sel = "100" THEN
        output <= c;
    ELSE
        output <= (OTHERS => 'X');
    END IF;
END PROCESS;
```



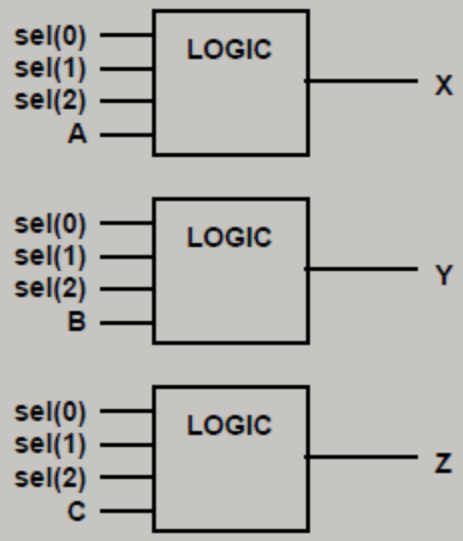
- Beware of building unnecessary dependencies
  - e.g. Outputs x, y, z are mutually exclusive, IF-ELSIF causes all outputs to be dependant on all tests & creates latches

```
PROCESS (sel,a,b,c)
BEGIN
      IF sel = "010" THEN
            x <= a;
      ELSIF sel = "100" THEN
            y <= b;
      ELSIF sel = "001" THEN
            z <= c;
      ELSE
            x <= '0';
            y <= '0';
            z <= '0';
      END IF;
END PROCESS;
```



- Separate IF statements and close

```
PROCESS (sel, a, b, c)
BEGIN
      IF sel = "010" THEN
      x <= a;
      ELSE
      x <= '0';
      END IF;
      IF sel = "100" THEN
      y <= b;
      ELSE
      y <= '0';
      END IF;
      IF  sel = "001" THEN
      z <= c;
      ELSE
      z <= '0';
      END IF;
END PROCESS;
```




```
PROCESS (sel, a, b, c)
BEGIN
      x <= '0';
      y <= '0';
      z <= '0';
      IF sel = "010" THEN
      x <= a;
      END IF;
      IF sel = "100" THEN
      y <= b;
      END IF;
      IF sel = "001" THEN
      z <= c;
      END IF;
END PROCESS;
```

- Conditions where output is undetermined

```
output: PROCESS (filter)
BEGIN
      CASE  filter IS
            WHEN idle =>
                  nxt <= '0';
                  first <= '0';
            WHEN tap1 =>
                  sel <= "00";
                  first <= '1';
            WHEN tap2 =>
                  sel <= "01";
                  first <= '0';
            WHEN tap3 =>
                  sel <= "10";
            WHEN tap4 =>
                  sel <= "11";
                  nxt <= '1';
      END CASE;
END PROCESS output;
```

sel missing

nxt missing

nxt missing

nxt & first missing

first missing

- *Undetermined output conditions implies memory*
- *Latch generated for **ALL** 3 outputs*

- Conditions where output is determined

```
output: PROCESS(filter)
BEGIN
        first <= '0';
        nxt <= '0';
        sel <= "00";
        CASE filter IS
                WHEN idle =>
                WHEN tap1 =>
                        first <= '1';
                WHEN tap2 =>
                        sel <= "01";
                WHEN tap3 =>
                        sel <= "10";
                WHEN tap4 =>
                        sel <= "11";
                        nxt <= '1';
                END CASE;
END PROCESS output;
```

Signals Initialized

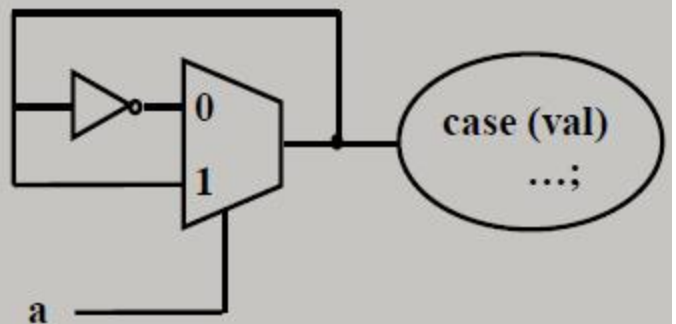
To remove latches & ensure outputs are never undetermined

- *Use signal initialization at beginning of case statement (case statement only deals with changes)*
- *Use don't cares ('-') for WHEN OTHERS clause, if design allows (for better logic optimization)*
- *Manually set output in each case*

```
ARCHITECTURE logic OF cmb_vari IS
BEGIN
    PROCESS(i0, i1, a)
        VARIABLE val :
                INTEGER RANGE 0 TO 1;
    BEGIN
        IF (a = '0') THEN
            val := val;
        ELSE
            val := val + 1;
        END IF;

        CASE val IS
            WHEN 0 =>
                q <= i0;
            WHEN OTHERS =>
                q <= i1;
        END CASE;
    END PROCESS;
END ARCHITECTURE logic;
```

Variable used without initialization

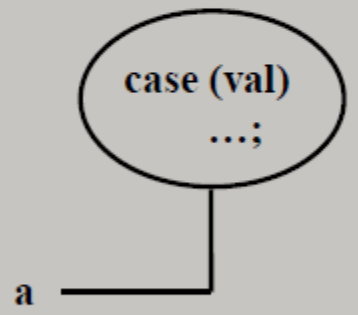


variable

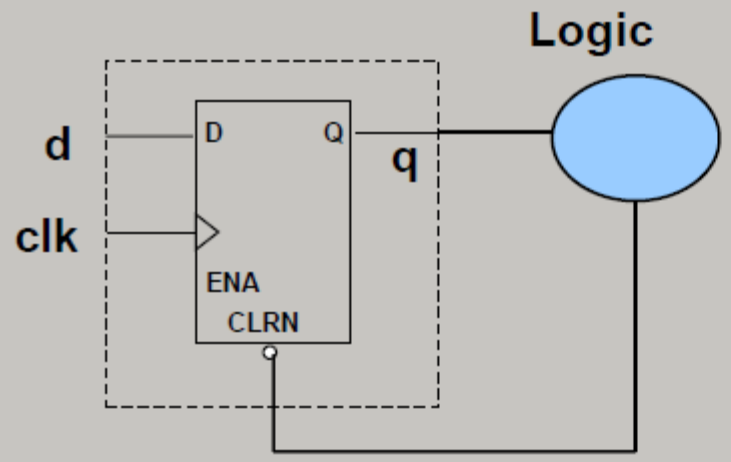
```
ARCHITECTURE logic OF cmb_vari IS
BEGIN
    PROCESS(i0, i1, a)
        VARIABLE val :
                INTEGER RANGE 0 TO 1;
    BEGIN
        val := 0;
        IF (a = '0') THEN
            val := val;
        ELSE
            val := val + 1;
        END IF;

        CASE val IS
            WHEN 0 =>
                q <= i0;
            WHEN OTHERS =>
                q <= i1;
        END CASE;
    END PROCESS;
END ARCHITECTURE logic;
```

*Assign initial value or signal to **variable***



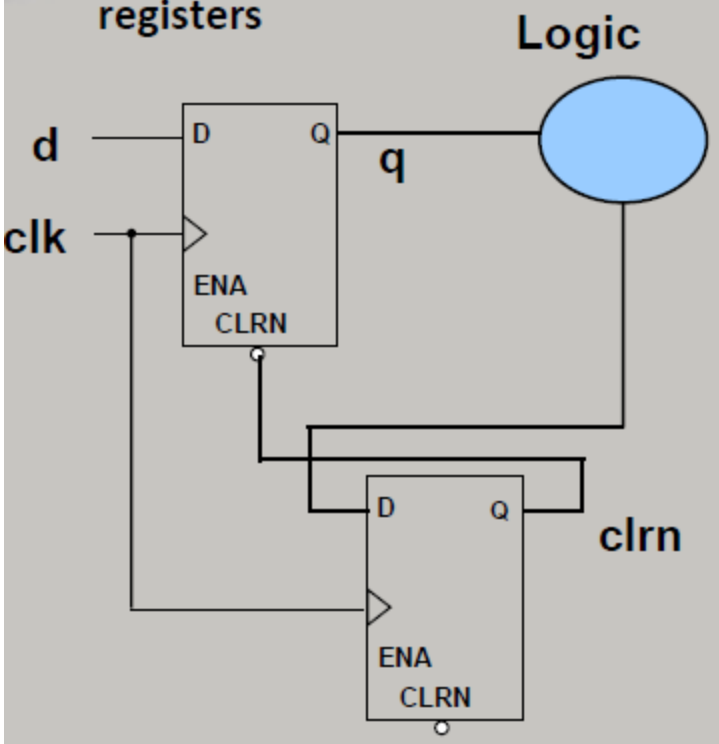
- Common cause of instability
- Behavior of loop depends on the relative propagation delays through logic
  - Propagation delays can change
- Simulation tools may not match hardware behavior




```
PROCESS (clk, clrn)
BEGIN
    IF clrn = '0' THEN
        q <= 0;
    ELSIF rising_edge (clk) THEN
        q <= d;
    END IF;
END PROCESS;

clrn <= (ctrl1 XOR ctrl2) AND q;
```

- All feedback loops should include registers



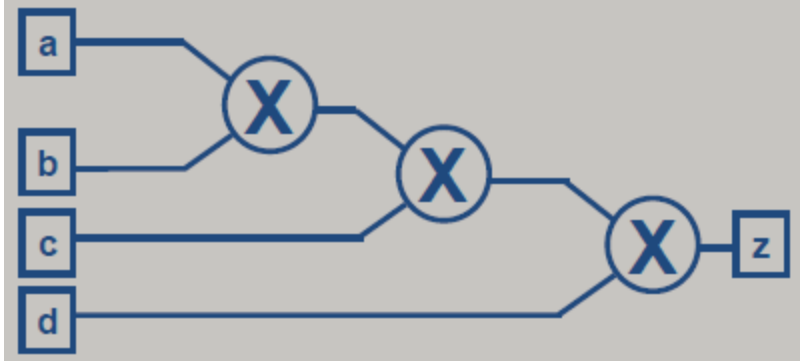

```
PROCESS (clk, clrn)
BEGIN
      IF clrn = '0' THEN
            q <= 0;
      ELSIF rising_edge (clk)
            q <= d;
      END IF;
END PROCESS;

PROCESS (clk)
BEGIN
      IF rising_edge (clk) THEN
            clrn <= (ctrl1 XOR ctrl2) AND q;
      END IF;
END PROCESS;
```

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

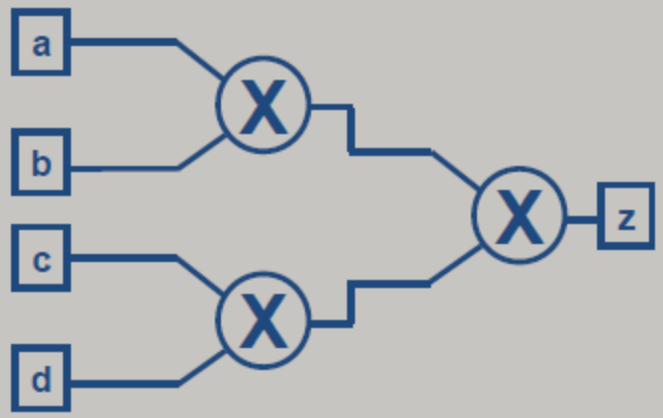
Unbalanced

z <= a * b * c * d



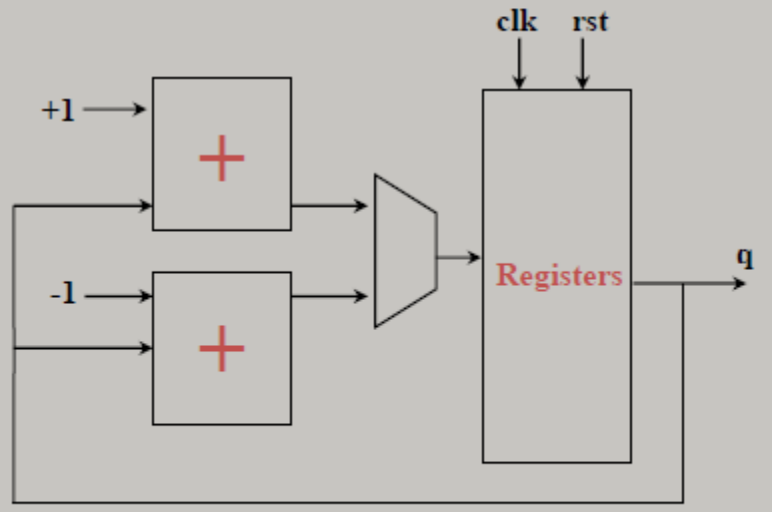
Balanced

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



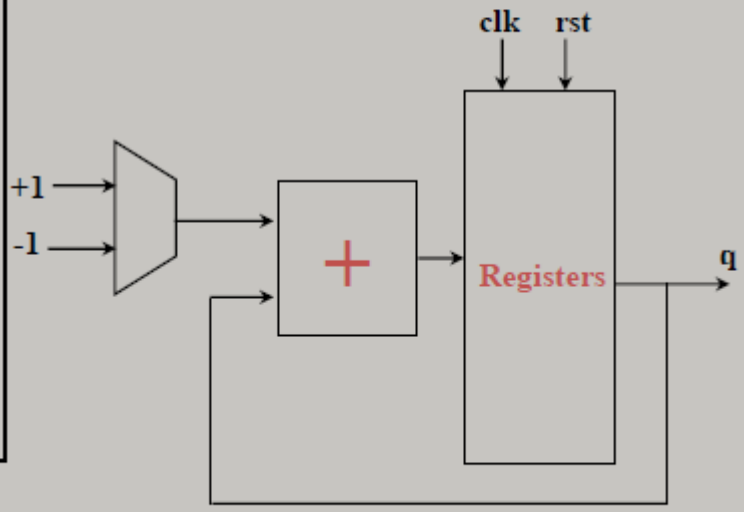
```
process(rst, clk)
variable tmp_q : std_logic_vector(7 DOWNTO 0);
begin
        if rst = '0' then
                tmp_q := (OTHERS => '0');
        elsif rising_edge(clk) then
                if updn = '1' then
                        tmp_q := tmp_q + 1;
        else
                        tmp_q := tmp_q - 1;
        end if;
        end if;
        q <= tmp_q;
end process;
```

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



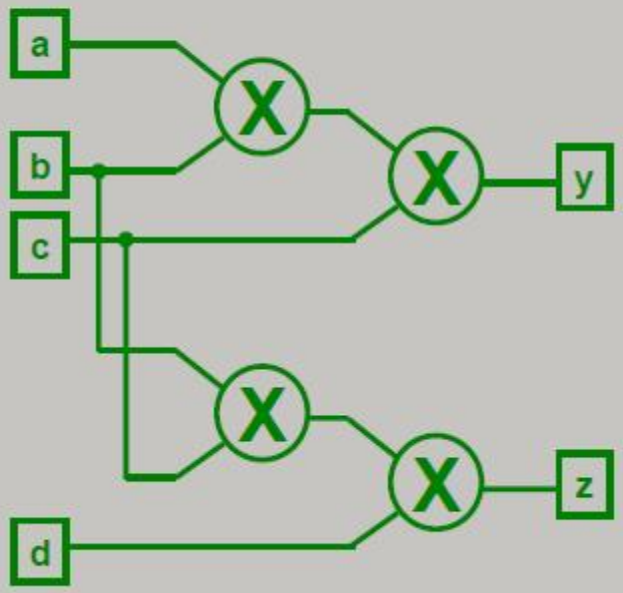
```
process(rst, clk)
variable tmp_q : std_logic_vector(7 DOWNTO 0);
variable dir : integer range -1 to 1;
begin
        if rst = '0' then
                tmp_q := (OTHERS => '0');
        elsif rising_edge(clk) then
                if updn = '1' then
                        dir := 1;
                else
                        dir := -1;
                end if;
                tmp_q := tmp_q + dir;
        end if;
        q <= tmp_q;
end process;
```

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



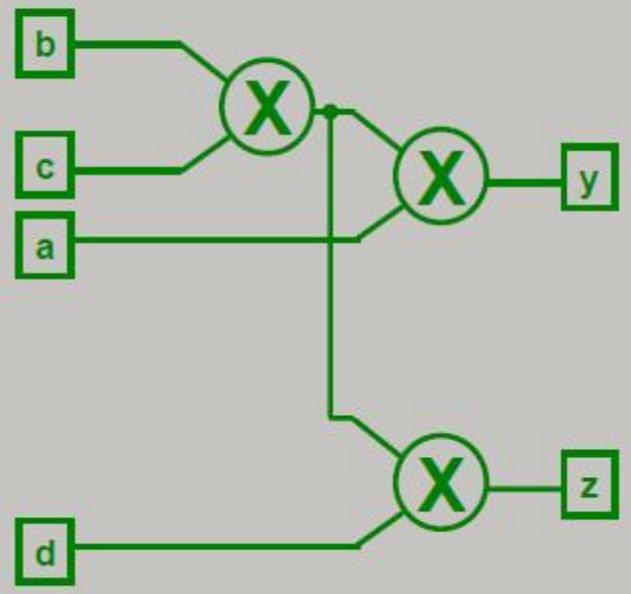
y <= a * b * c

z <= b * c * d



4 Multipliers!

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



3 Multipliers!

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