DESCRIPTION OF DIGITAL CIRCUITS USING VHDL

Combinatinal circuits

Sequential circuits

Design organization. Generic design

Iterative operations

Authors: Luis Entrena Arrontes, Celia López, Mario García, Enrique San Millán, Marta Portela, Almudena Lindoso

Outline

- ☞ **Condicional sentences**
- ☞ **Rules for design of combinational circuits**
- ☞ **Examples and exercises**

Conditional sentences

Sequential sentences

- ☞ **IF...THEN...ELSE**
- ☞ **CASE...IS...WHEN**

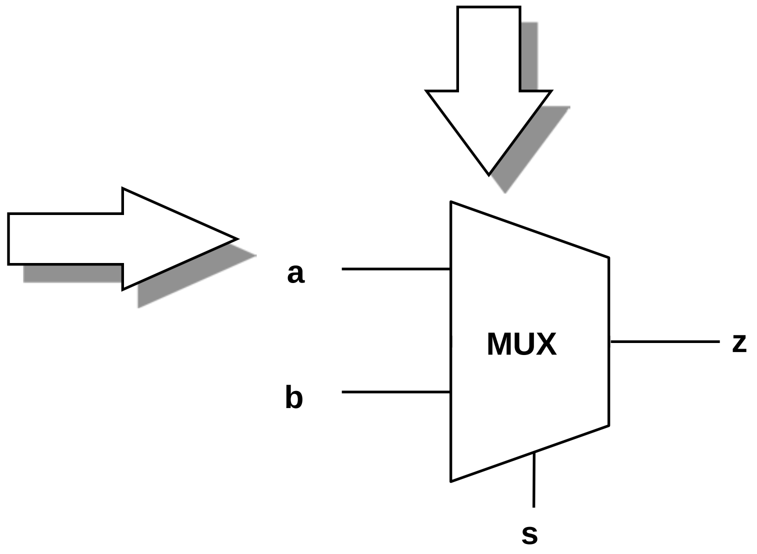
Concurrent sequences

- ☞ **... WHEN ...**
  (conditional assignment)
- ☞ **WITH ... SELECT**
  (selection assingment)

Multiplexer 2:1

```
-- Sequential description
PROCESS (a, b, s)
BEGIN
        IF s = '0' THEN
                z <= a;
        ELSE
                z <= b;
        END IF;
END PROCESS;
```

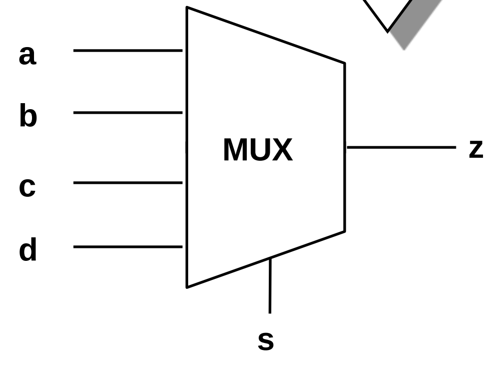
```
-- Concurrent description
z <= a WHEN s = '0'
      ELSE b;
```



Multiplexer 4:1

```
-- Sequential description
PROCESS (a, b, c, d, s)
BEGIN
      CASE s IS
            WHEN "00" =>     z <= a;
            WHEN "01" =>     z <= b;
            WHEN "10" =>     z <= c;
            WHEN OTHERS => z <= d;
            END CASE;
END PROCESS;
```

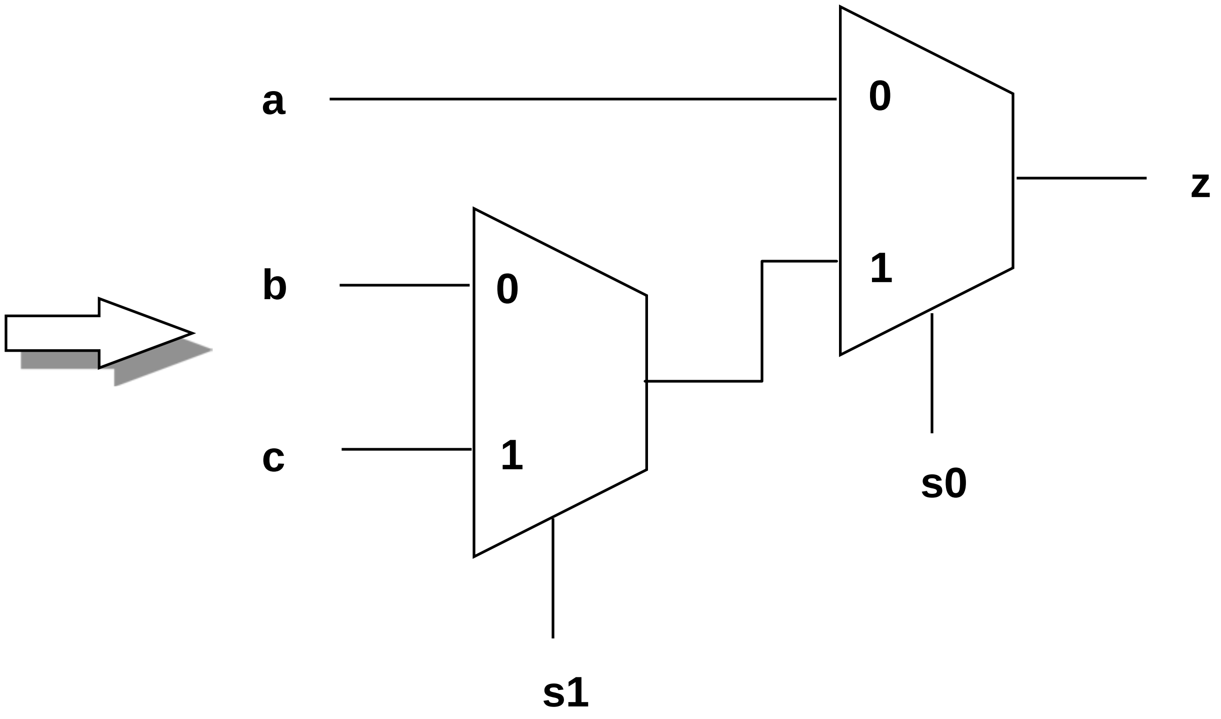
```
-- Concurrent description
WITH s SELECT
z <=      a WHEN "00",
          b WHEN "01",
          c WHEN "10",
          d WHEN OTHERS;
```



Diferences between IF and CASE

☞ **IF...THEN...ELSE imposes priority in the selection inputs**

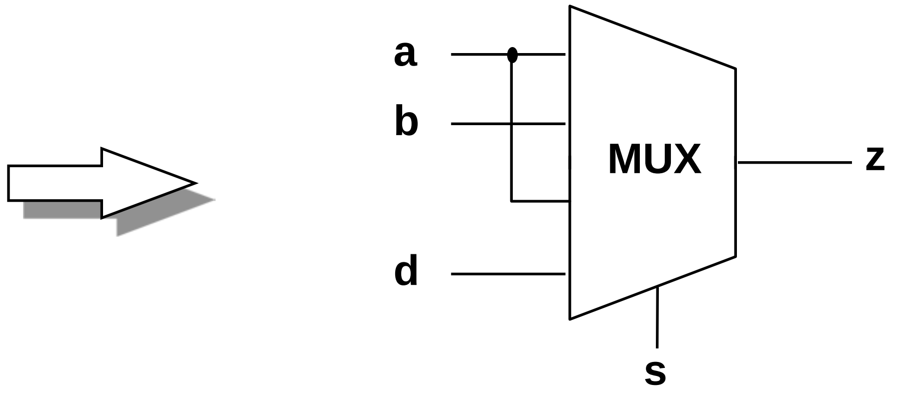
```
IF s0 = '0' THEN
        z <= a;
ELSIF s1 = '0' THEN
        z <= b;
ELSE
        z <= c;
END IF;
```



Diferences between IF and CASE

☞ CASE does not impose priority in the selection inputs

```
PROCESS (a, b, c, d, s)
BEGIN
        CASE s IS
                WHEN "00" | "10" =>     z <= a;
                WHEN "01" =>            z <= b;
                WHEN OTHERS =>          z <= d;
                END CASE;
END PROCESS;
```



Truth tables

```
PROCESS (a, b, c)
        VARIABLE inputs: STD_LOGIC_VECTOR (2 DOWNTO 0);
BEGIN
        inputs  := a & b & c;
        CASE inputs IS
                WHEN "000" =>       f <= '1';
                WHEN "001" =>       f <= 'X';
                WHEN "010" =>       f <= '0';
                WHEN "011" =>       f <= '1';
                WHEN "100" =>       f <= '0';
                WHEN "101" =>       f <= '1';
                WHEN OTHERS =>      f <= 'X';
        END CASE;
END PROCESS;
```

Rules for design of combinational circuits

☞ **Condicions that are necessary for a process to infere a correct combinational circuit:**

- ❑ ***If a signal is assigned in a conditional process o conditional concurrent sentence, then it needs to have assigned a value for all the possible branches or conditions.***
- ❑ ***The sensitivity list of a process must contain all the signal that are read in the process, i.e. all the signal which value is used in the process, and therefore are inputs in the synthesized circuit.***
- ❑ ***Variables used in processes have to be used as intermediate values, i.e. they have to be written before being read.***

Rules for design of combinational circuits

☞ **Some consequences of the previous rules:**

- ❑ **To cover all the possible cases, it is convenient that every IF sentence or conditional assignment is followed by an ELSE clause, and every CASE or selection assignment is followed by an WHEN OTHERS clause.**
- ❑ **Loop references should not be created, i.e. outputs that depend on themselves, because those loops are asynchronous.**
- ❑ **Use of signals is recommended over variables whenever it is possible.**

Examples (mistakes and correct solution)

```
PROCESS( s, a)
BEGIN
        IF s = ‘0’ THEN
                z <= a;
        END IF;
END PROCESS;
```

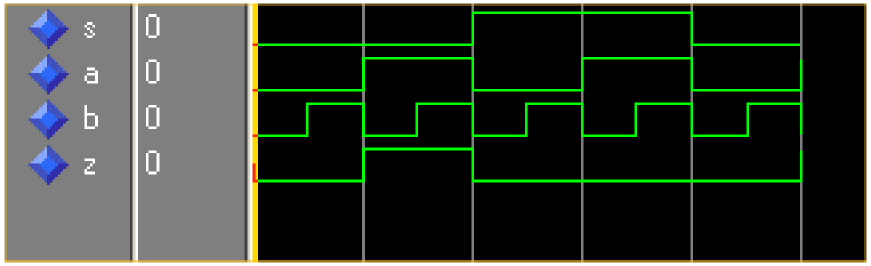
```
PROCESS( s, a)
BEGIN
        IF s = ‘0’ THEN
                z <= a;
        ELSE
                z <= z;
        END IF;
END PROCESS;
```

```
PROCESS( s, a)
BEGIN
        IF s = ‘0’ THEN
                z <= a;
        ELSE
                z <= b;
        END IF;
END PROCESS;
```

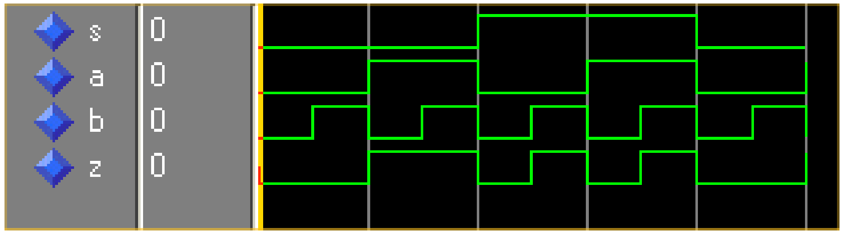
```
PROCESS( s, a, b)
BEGIN
        IF s = ‘0’ THEN
                z <= a;
        ELSE
                z <= b;
        END IF;
END PROCESS;
```

Effect of the sensitivity list in simulation

☞ **Incorrect sensitivity list (s, a)**



☞ **Correct sensitivity list (s, a, b)**



Exercise

- Design a 2:4 decoder
  - Using sequential sentences
  - Using concurrent sentences
- Declaration of the entity:

```
ENTITY decodificador IS
            PORT (  a: IN STD_LOGIC_VECTOR (1 DOWNTO 0);

                    z: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));

END decodificador;
```

Exercise

- Design a 4-bit comparator
  - ❑ Using sequential sentences
  - ❑ Using concurrent sentences
- Declaration of the entity:

```
ENTITY comparator IS
            PORT (  a, b: IN STD_LOGIC_VECTOR (3 DOWNTO 0);

                    less_than, greater_than, equal: OUT STD_LOGIC);

END comparator;
```

Exercise

- **Design a 2-bit encoder with priority**
  - ❑ **Using sequential sentences**
  - ❑ **Using concurrent sentences**
- **Declaration of the entity:**

```
ENTITY codificador IS
          PORT (  a: IN STD_LOGIC_VECTOR (3 DOWNTO 0);

                  z: OUT STD_LOGIC_VECTOR (1 DOWNTO 0);

                  gs: OUT STD_LOGIC);

END decodificador;
```

Solution: decoder

Solution: comparator

Solution: encoder

DESCRIPTION OF DIGITAL CIRCUITS USING VHDL

Combinatinal circuits

Sequential circuits

Design organization. Generic design

Iterative operations

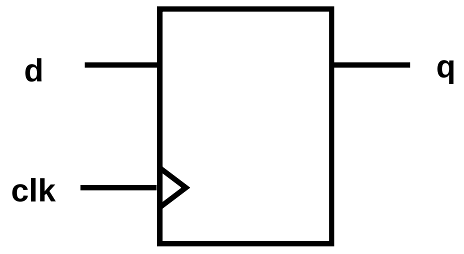
Outline

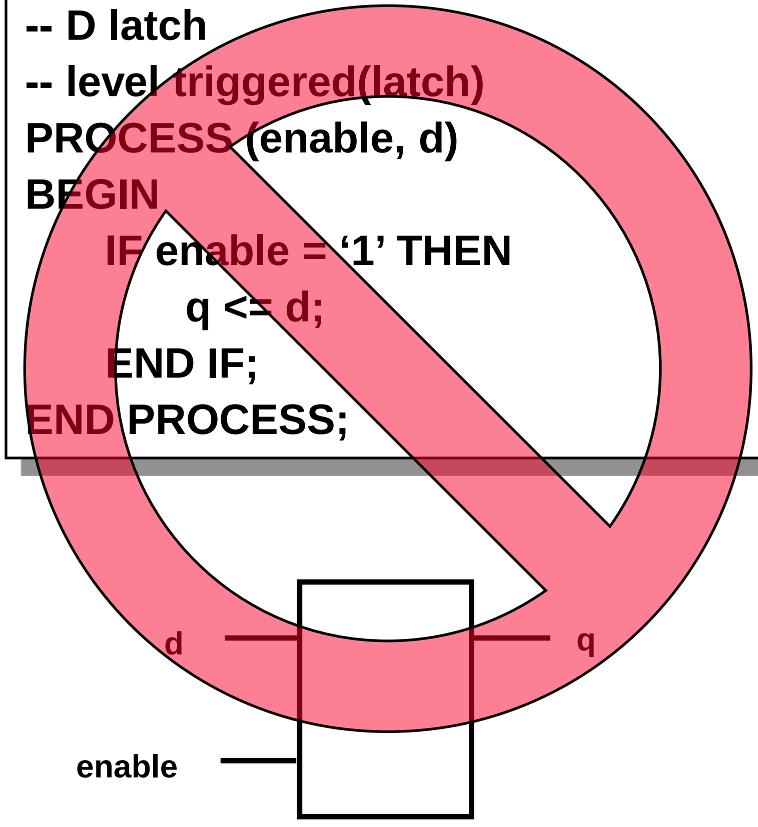
- ☞ **Registers and flip-flops**
- ☞ **Rules for design of sequential circuits**
- ☞ **Counters**
- ☞ **Finite state machines**

Bistables

```
-- D flip-flop
-- rising edge triggered
PROCESS (clk)
BEGIN
      IF clk'EVENT AND clk = '1' THEN
            q <= d;
      END IF;
END PROCESS;
```

```
-- D latch
-- level triggered(latch)
PROCESS (enable, d)
BEGIN
      IF enable = '1' THEN
            q <= d;
      END IF;
END PROCESS;
```





ONLY SYNCHRONOUS DESIGN

A register

- **If q is a vector, then it is created a register with as many flip-flops as the vector dimension specifies.**

```
-- Registro de 8 bits
SIGNAL q, d: STD_LOGIC_VECTOR (7 DOWNTO 0);
...
PROCESS (clk)
BEGIN
        IF clk’EVENT AND clk = ‘1’ THEN
                q <= d;
        END IF;
END PROCESS;
```

Flip-flop with asynchronous inputs

- Asynchronous inputs must be included in the sensitivity list
- Note: Control signals priority is given by the IF ... ELSIF ... ELSIF ... order

```
PROCESS (preset, clr, clk)
BEGIN
    IF preset = ‘1’ THEN
        q <= ‘1’;
    ELSIF clr = ‘1’ THEN
        q <= ‘0’;
    ELSIF clk’EVENT AND clk = ‘1’ THEN
        q <= d;
    END IF;
END PROCESS;
```

preset has priority over clr

Flip-flop with synchronous control inputs

☞ **Synchronous inputs are not included in the sensitivity list (changes in these inputs depend on the clock signal)**

```
PROCESS (reset, clk)
BEGIN
     IF reset = '1' THEN
          q <= '0';
     ELSIF clk'EVENT AND clk = '1' THEN
          IF enable = '1' THEN
            q <= d;
          END IF;
     END IF;
END PROCESS;
```

} **Asynchronous part**

} **Synchronous part**

Rules for desing of synchronous sequential circuits

- ☞ **Must be described using processes**
- ☞ **There must be a clock edge condition**
    - ✓ IF clk’EVENT AND clk = ‘1’ THEN
    - ✓ IF clk’EVENT AND clk = ‘0’ THEN
    - ❑ **After that condition the ELSE clause cannot be used**
    - ❑ **If a signal is assigned, it is not needed to cover all the cases (in non-covered cases the signal maintains its value)**
- ☞ **Sensitivity list: clock + asynchronous inputs (reset)**
    - ❑ **Data inputs and synchronous inputs do not appear in the sensitivity list.**

Rules for design of synchronous sequential circuits

- ☞ **“Template” for design of synchronous sequential circuits**

```
PROCESS( <clock + asynchronous inputs>)
BEGIN
    IF <asynchronous initialization condition> THEN
        < asynchronous initialization >
    ELSIF <active clock edge> THEN
        -- synchronous behavior
    END IF;
END PROCESS;
```

Examples

```
PROCESS (reset, clk)
BEGIN
        IF reset = ‘0’ THEN
                q <= “0101”;
        ELSIF clk’EVENT AND clk = ‘1’ AND enable = ‘1’ THEN
                q <= d;
        END IF;
END PROCESS;
```

Examples

```
PROCESS (reset, clk)
BEGIN
    IF reset = ‘0’ THEN
        q <= “0101”;
    ELSIF enable = ‘1’  THEN
        IF clk’EVENT AND clk = ‘1’ THEN
            q <= d;
        END IF;
    END IF;
END PROCESS;
```

Examples

```
PROCESS (reset, clk)
BEGIN
    IF reset = ‘1’ THEN
        IF clk’EVENT AND clk = ‘1’ THEN
            IF enable = ‘1’ THEN
                    q <= d;
            END IF;
        END IF;
    ELSE
        q <= “0101”;
    END IF;
END PROCESS;
```

Inference of registers and flip-flops

☞ **Every signal that is assigned between *IF clk'EVENT* ... and *END IF* infiere a flip-flop/register**

☞ **Every variable that is read between *IF clk'EVENT* ... and *END IF,* before being written, infere a flip-flop/register**

Examples

```
-- Example 1 (inferes 1 flip-flop)
PROCESS (clk)
    VARIABLE v: STD_LOGIC;
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        v := d;
        q <= v;
    END IF;
END PROCESS;
```

```
-- Example 2 (inferes 2 flip-flops)
-- Shift register, but NOT recommended
PROCESS (clk)
    VARIABLE v: STD_LOGIC;
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        q <= v;
        v := d;
    END IF;
END PROCESS;
```

Examples

```
-- Example 3 (inferes 2 flip-flops)
-- Shift register (recommended)
SIGNAL v: STD_LOGIC;
…
PROCESS (clk)
    BEGIN
    IF clk’EVENT AND clk = ‘1’ THEN
        v <= d;
        q <= v;
    END IF;
END PROCESS;
```

Examples

```
-- Example 4 (inferes 2 flip-flops)
PROCESS (clk)
      VARIABLE a: STD_LOGIC;
BEGIN
      IF clk'EVENT AND clk = '1' THEN
            a := a +1;
            q <= a;
      END IF;
END PROCESS;
```

```
-- Example 5 (inferes 1 flip-flop)
PROCESS (clk)
      VARIABLE a: STD_LOGIC;
BEGIN
      IF clk'EVENT AND clk = '1' THEN
            a := a +1;
      END IF;
       q <= a;
END PROCESS;
```

☞ **But use of signals instead of variables is recommended**

Examples

```
-- Combinational adder
PROCESS (a, b)
BEGIN
      s <= a + b;
END PROCESS;
```

```
-- Adder followed by a register
PROCESS (clk)
BEGIN
      IF clk’EVENT AND clk = ‘1’ THEN
            s <= a + b;
      END IF;
END PROCESS;
```

Examples

```
-- Adder with registered inputs and outputs
PROCESS (clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        a1 <= a;
        b1 <= b;
        s <= a1 + b1;
    END IF;
END PROCESS;
```

```
-- Adder with registered inputs and outputs
-- Input and output registers with reset input
PROCESS (reset, clk)
BEGIN
    IF reset = '1' THEN
        a1 <= (OTHERS => '0');
        b1 <= (OTHERS => '0');
        s <= (OTHERS => '0');
    ELSIF clk'EVENT AND clk = '1' THEN
        a1 <= a;
        b1 <= b;
        s <= a1 + b1;
    END IF;
END PROCESS;
```

Examples

- **A register may not have reset, but a reset signal needs always an associated register!**

```
-- Adder followed by register
-- Incorrect initialization!
PROCESS (clk)
BEGIN
        IF reset = ‘1’ THEN
                a <= (OTHERS => ‘0’);
                b <= (OTHERS => ‘0’);
                s <= (OTHERS => ‘0’);
        ELSIF clk’EVENT AND clk = ‘1’ THEN
                s <= a + b;
        END IF;
END PROCESS;
```

Exercises

☞ **Design a JK flip-flop falling-edge triggered**

```
ENTITY jk_flipflop IS
      PORT (   reset:    IN STD_LOGIC;
                clk:     IN STD_LOGIC;
                j, k:    IN STD_LOGIC;
                q:       OUT STD_LOGIC);
END k_flipflop;
```

☞ **Design a shift register**

```
ENTITY shift_reg IS
      PORT (    reset:   IN STD_LOGIC;
                clk:     IN STD_LOGIC;
                es:      IN STD_LOGIC;
                q:       OUT STD_LOGIC_VECTOR (0 to 3));
END shift_reg;
```

Solution: JK flip-flop

Solution: Shift register

Counters

```
PROCESS (reset, clk)
BEGIN
      IF reset = ‘1’ THEN
            q <= “00000000”;
      ELSIF clk’EVENT AND clk = ‘1’ THEN
            IF enable = ‘1’ THEN
                        q <= q + 1;
            END IF;
      END IF;
END PROCESS;
```

Counters

```
PROCESS (reset, clk)
    VARIABLE c: INTEGER RANGE 0 TO 255;
BEGIN
    IF reset = '1' THEN
        c := 0;
    ELSIF clk'EVENT AND clk = '1' THEN
        IF enable = '1' THEN
            IF c = 255 THEN
                c := 0;
            ELSE
                c := c + 1;
            END IF;
        END IF;
    END IF;
    q <= c;
END PROCESS;
```

Necessary to avoid overflow in simulation

q declared as integer

If q is std_logic_vector, then

q <= CONV_STD_LOGIC_VECTOR(c, 8);

Exercise

☞ **Design a 4-bit up-counter with carry output**

```
ENTITY cont4 IS
      PORT (  reset:          IN STD_LOGIC;
              clk:            IN STD_LOGIC;
              q:              OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
              carry_out:      OUT STD_LOGIC);
END cont4 ;
```

Exercise

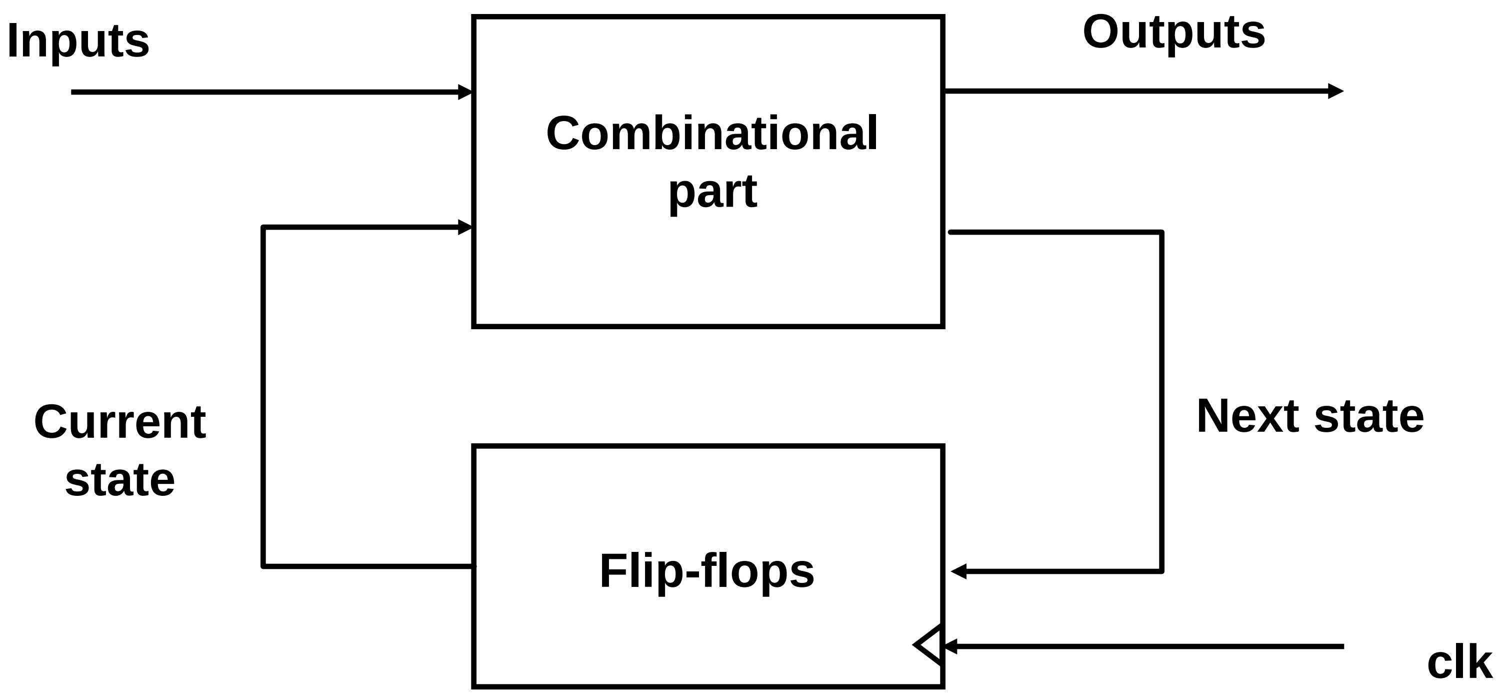
☞ **Design a up/down counter with enable and load inputs**

```
ENTITY cont_up_down IS
        PORT (   reset:          IN STD_LOGIC;
                 clk:            IN STD_LOGIC;
                 load:           IN STD_LOGIC;
                 d:              IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                 ud:             IN STD_LOGIC;
                 enable:         IN STD_LOGIC;
                 q:              OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END cont_up_down;
```

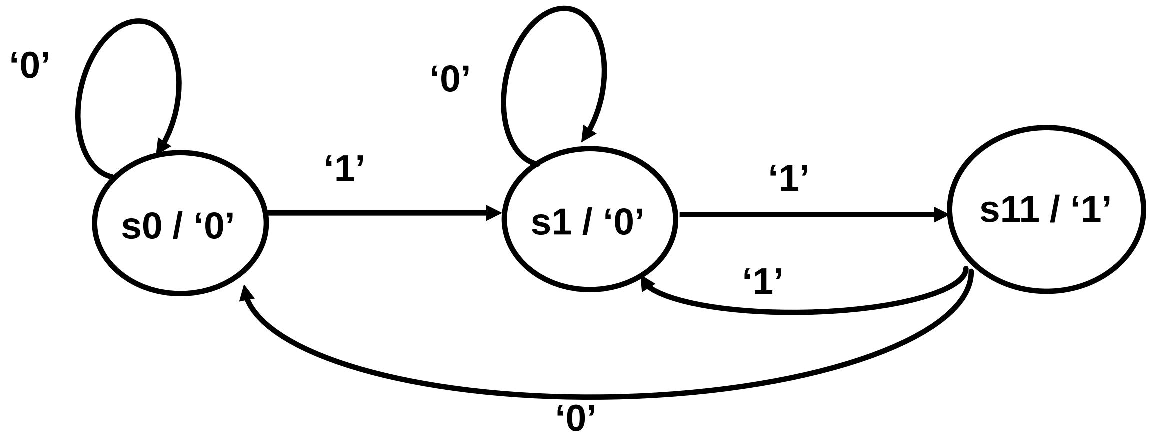
Solution: 4-bit up-counter

Solution: up/down counter with enable and load inputs

Finite state machines



Example: Moore’s machine



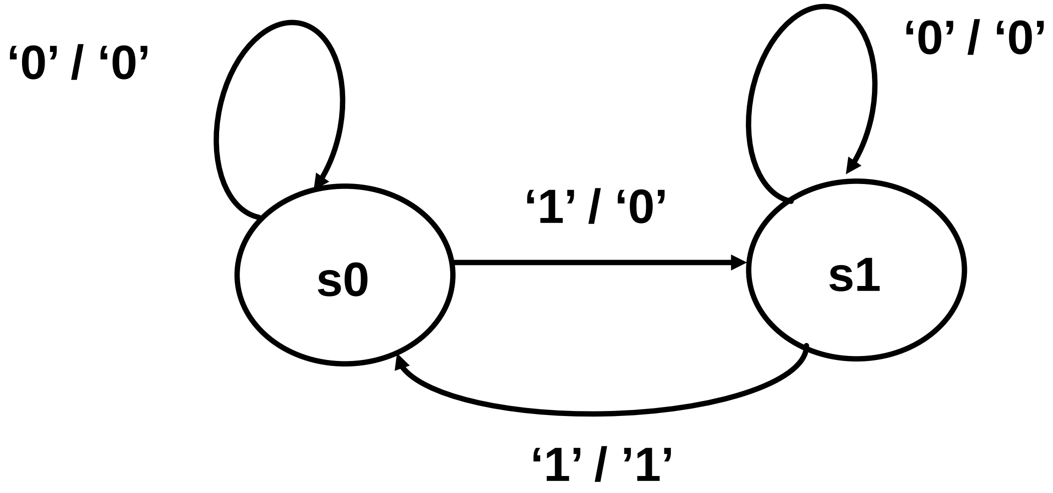
```
ENTITY machine is
    PORT(  clk:   in std_logic;
           reset: in std_logic;
           a:     in std_logic;
           z :    out std_logic);
END machine;
```

Example: Moore’s machine

```
ARCHITECTURE moore OF machine IS
   TYPE state IS (s0, s1, s11);
   SIGNAL current_state, next_state: state;
BEGIN
   PROCESS (clk, reset)
   BEGIN
      IF reset = '1' THEN
         current_state <= s0;
      ELSIF clk'EVENT AND clk = '1' THEN
         current_state <= next_state;
      END IF;
   END PROCESS;
....
```

```
   PROCESS (current_state, a)
   BEGIN
      CASE current_state IS
      WHEN s0 =>  z <= '0';
            IF a = '0' THEN
                  next_state <= s0;
            ELSE
                  next_state <= s1;
            END IF;
       WHEN s1 => z <= '0';
            IF a = '0' THEN
                  next_state <= s1;
            ELSE
                  next_state <= s11;
            END IF;
       WHEN s11 => ...
       ....
      END CASE;
   END PROCESS;
END moore;
```

Example: Mealy's machine



```
ENTITY machine is
    PORT(  clk:   in std_logic;
           reset: in std_logic;
           a:     in std_logic;
           z :    out std_logic);
END machine;
```

Example: Mealy's machine

```
ARCHITECTURE mealy OF machine IS
   TYPE state IS (s0, s1);
   SIGNAL current_state, next_state: state;
BEGIN
   PROCESS (clk, reset)
   BEGIN
      IF reset = '1' THEN
          current_state <= s0;
      ELSIF clk'EVENT AND clk = '1' THEN
          current_state <= next_state;
      END IF;
   END PROCESS;
....
```

```
    PROCESS (current_state, a)
    BEGIN
       CASE current_state IS
       WHEN s0 =>  z <= '0';
             IF a = '0' THEN
                 next_state <= s0;
             ELSE
                 next_state <= s1;
             END IF;
        WHEN s1 =>
             IF a = '0' THEN
                z <= '0';
                 next_state <= s1;
             ELSE
                z <= '1';
                 next_state <= s11;
             END IF;
       END CASE;
    END PROCESS;
...
```