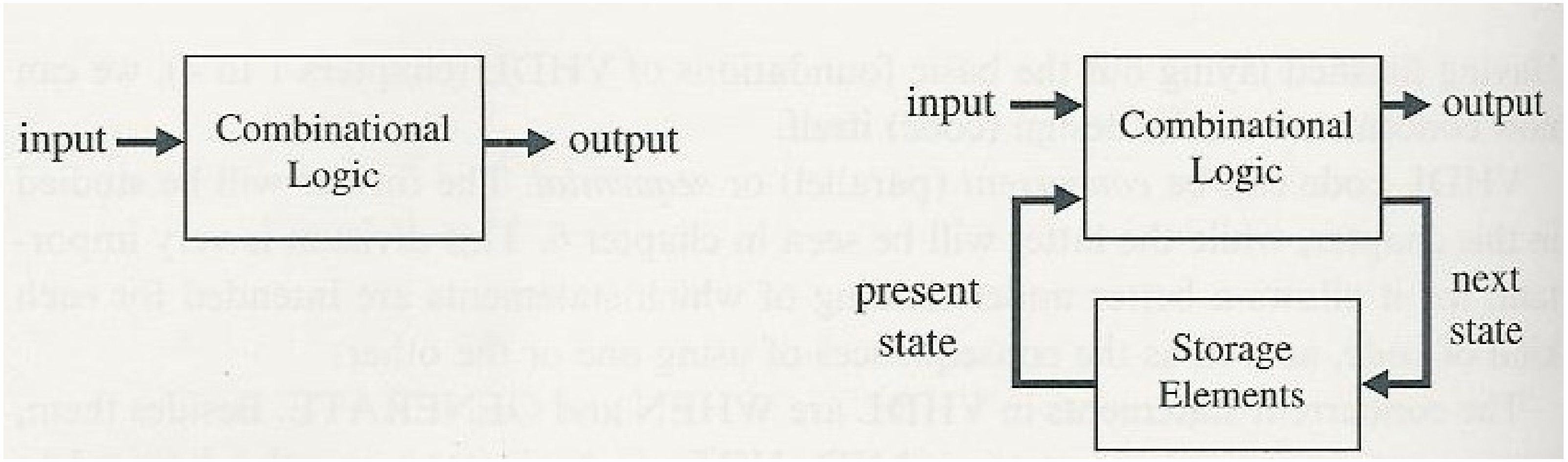
outlines

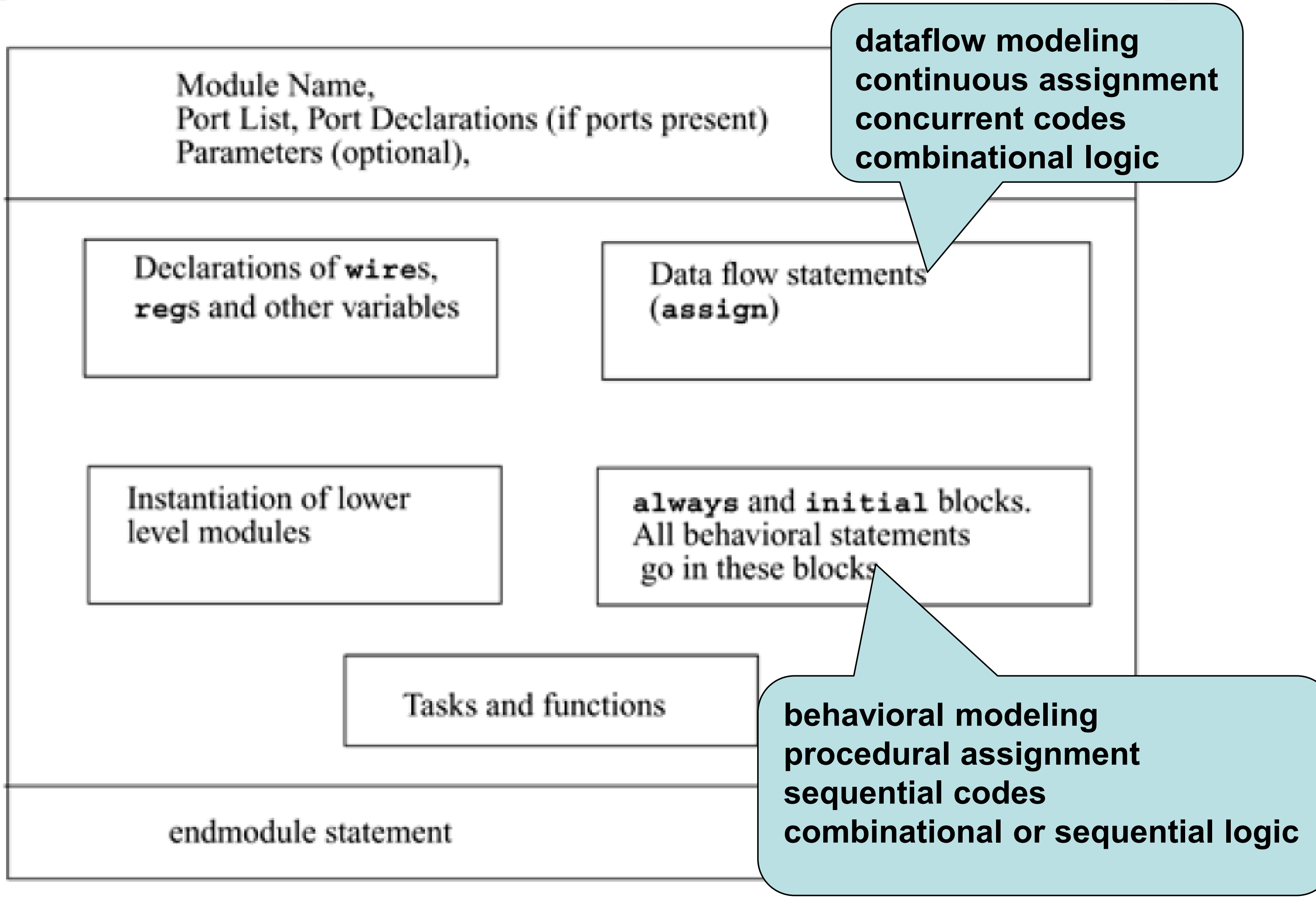
- VHDL concurrent code vs. Verilog continuous assignments
- VHDL sequential code vs. Verilog procedural assignments
- signals and variables
- finite state machine

combinational vs. sequential logic

- combinational logic
  - outputs depend only on *current* inputs
    - pure fee-forward datapath
  - no memory
    - no feeback loop
- sequential logic
  - outputs also depend on *previous* inputs
  - contain either latches or flip-flops
    - e.g., pipelined designs, finite state machine (FSM), …



Verilog Module (data-flow and behavioral levels)



Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks
Tasks and functions
endmodule statement
dataflow modeling
continuous assignment
concurrent codes
combinational logic
behavioral modeling
procedural assignment
sequential codes
combinational or sequential logic

Verilog Continuous Assignment (assign)

```
assign out = i1 & i2;     // out is a net; i1 and i2 are nets

assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];
// addr is a vector net; addr1 and addr2 are vector registers

assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
// left-hand side (LHS) is a concatenation of a scalar net and a vector net
// {c_out, sum[3:0]} is equal to c_out & sum(3 downto 0) in VHDL
```

- *the left-hand side (LHS) must always be scalar or vector net (**wire**) (cannot be **reg**)*
- evaluated as soon as the right-hand-side (RHS) operands changes (unlike the behavioral procedural assignments where the statements in a procedure are executed depending on the sensitivity list)
- *expressions combine operators and operands*

Verilog Procedural Assignments (inside **initial** or **always** blocks)

- two structured procedures
  - **always** (SystemVerilog uses **always_ff, always_comb, always_latch**)
  - **Initial** (not synthesizable)
- procedural assignments update LHS values of
  - **reg** (SystemVerilog uses **logic** to replace **reg** for avoiding confusion)
  - **integer**
  - **real**
  - **time**
- The value placed on a variable remain unchanged until another procedural assignment updates the variable with a different value
  - unlike continuous assignments (**assign**) in dataflow where one assignment can cause the value of RHS expression to be continuously placed on the LHS expression

Verilog behavioral statements

- procedural assignments
  - blocking (**=**) vs. non-blocking (**<=**)
- conditional statements (**if … else**…)
- multi-way branching (**case … endcase**)
- looping statement
  - **while, repeat, for, forever**
- sequential and parallel blocks
  - sequential block (**begin … end**)
  - parallel block (**fork … join**)
- timing control (**#**)

VHDL concurrent vs. sequential code

- VHDL code is inherently concurrent (parallel)
  - all the statements executed concurrently
  - independent of the order of concurrent statements
  - concurrent code is also called *dataflow* code
- Statements inside **PROCESS**, **FUNCTION**, **PROCEDURE** are *sequential*
  - the orders of the sequential statements affects the results
  - sequential statements can still be used to realize combinational logic
- However, **process** blocks are concurrent among blocks
  - every process block is executed *in parallel*
  - but statements inside a process block are executed *sequentially*

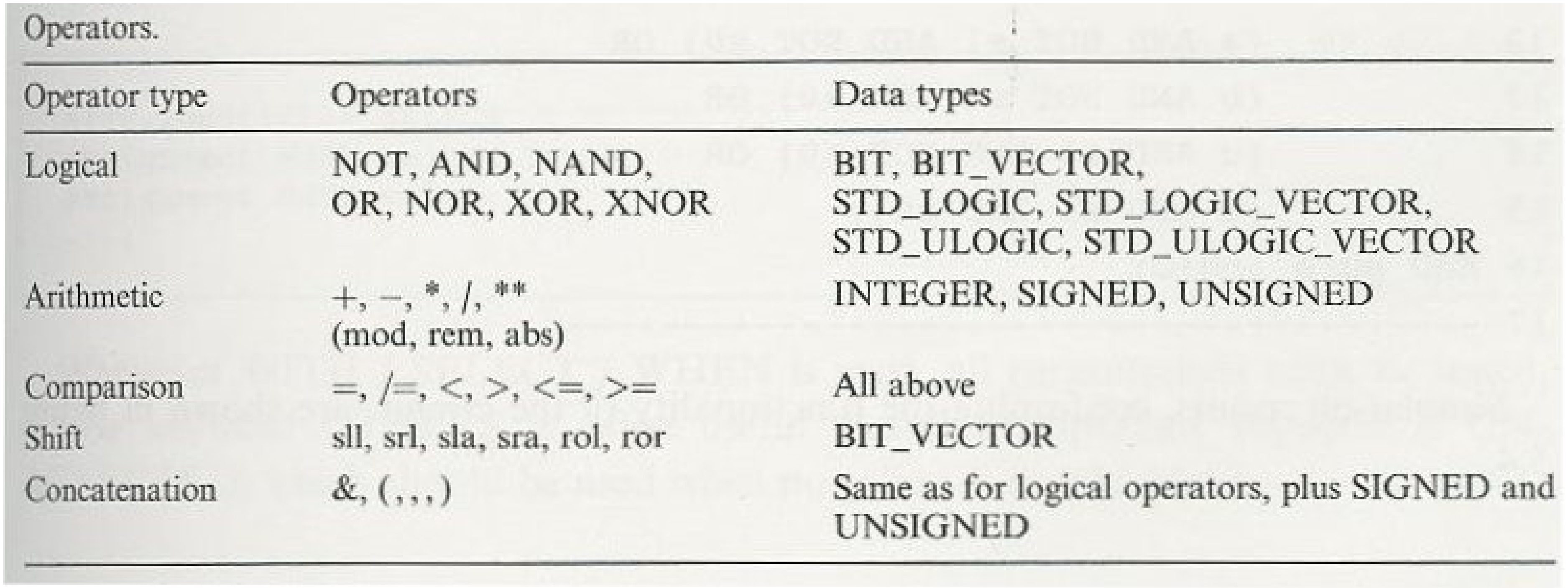
VHDL
Concurrent Code

VHDL concurrent code

- concurrent codes are used ***outside*** **process, function**, or **procedure**
  - operators
    - arithmetic, logical, relational, …
  - WHEN statements (realizing multiplexing)
    - **WHEN / ELSE**
    - **WITH / SELECT / WHEN**
  - **FOR/GENERATE**
  - BLOCK

VHDL Operators

- operators can be used to implement any combinational logic
  - logical. arithmetic, comparison, shift, concatenation



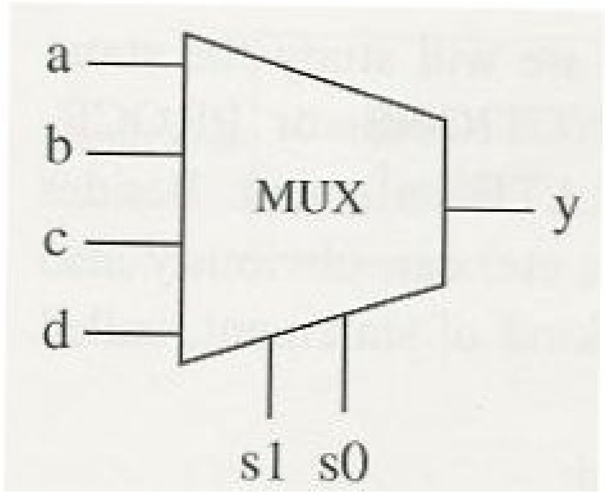
Operators.

Operator type	Operators	Data types
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmetic	+, −, *, /, ** (mod, rem, abs)	INTEGER, SIGNED, UNSIGNED
Comparison	=, /=, <, >, <=, >=	All above
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Concatenation	&, (, , ,)	Same as for logical operators, plus SIGNED and UNSIGNED

example (multiplexer #1)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
----------------------------------------
ENTITY mux IS
   PORT (a, b, c, d, s0, s1: IN STD_LOGIC;
                 y: OUT STD_LOGIC);
END mux;
----------------------------------------
ARCHITECTURE pure_logic OF mux IS
BEGIN
   y <=(a AND NOT s1 AND NOT s0) OR
        (b AND NOT s1 AND s0) OR
        (c AND s1 AND NOT s0) OR
        (d AND s1 AND s0);
END pure_logic;
```

$$y = a\bar{s}_1\bar{s}_0 + b\bar{s}_1 s_0 + c s_1 \bar{s}_0 + d s_1 s_0$$



WHEN

- **WHEN / ELSE**

  assignment WHEN condition ELSE

- **WITH / SELECT / WHEN**

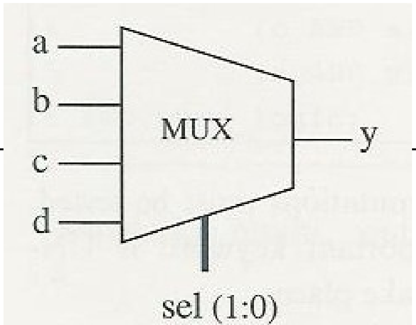
  WITH identifier SELECT

  assignment WHEN value

- similar to the conditional operator (**? :**) in Verilog

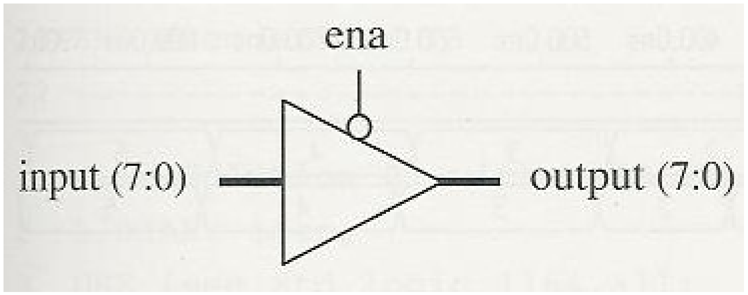
example (multiplexer #2)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY mux IS
PORT (a, b, c, d: IN STD_LOGIC;
sel: IN STD_LOGIC_VECTOR (1
    DOWNTO 0);
y: OUT STD_LOGIC);
END mux;
--------------------------------------------
ARCHITECTURE mux1 OF mux IS
BEGIN
y <=      a WHEN sel="00" ELSE
          b WHEN sel="01" ELSE
          c WHEN sel="10" ELSE
          d;
END mux1;
```



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY mux IS
PORT (a, b, c, d: IN STD_LOGIC;
sel: IN STD_LOGIC_VECTOR (1
    DOWNTO 0);
y: OUT STD_LOGIC);
END mux;
---------------------------------------------
ARCHITECTURE mux2 OF mux IS
BEGIN
WITH sel SELECT
y <= a WHEN "00",  -- notice "," not";"
      b WHEN "01",
      c WHEN "10",
      d WHEN OTHERS;
          -- cannot be "d WHEN "11" "
END mux2;
```

example (8-bit tri-state buffer)



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY tri_state IS
PORT (    ena: IN STD_LOGIC;
            input: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
          output: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END tri_state;
---------------------------------------------
ARCHITECTURE tri_state OF tri_state IS
BEGIN
           output <= input                WHEN (ena='0') ELSE
                        (OTHERS => 'Z');
END tri_state;
```

GENERATE

- FOR / GENERATE
  - Both the range limits should be static (not input parameter)
  - Avoid multiple-driven (unresolved) signal

```
SIGNAL x: BIT_VECTOR (7 DOWNTO 0);
SIGNAL y: BIT_VECTOR (15 DOWNTO 0);
SIGNAL z: BIT_VECTOR (7 DOWNTO 0);
…
G1: FOR i IN x'RANGE GENERATE
                z(i) <= x(i) AND y(i+8);
       END GENERATE;
```

```
NotOK: FOR i IN 0 TO 7 GENERATE
                    accum <= "11111111" WHEN ((a(i) AND b(i)) = '1' ELSE
                                  "00000000";   -- accum is multiple driven
            END GENERATE;
```

```
NotOK: FOR i IN 0 TO 7 GENERATE
                    accum <= accum + 1 WHEN x(i) = '1';
            END GENERATE;
```

Example (vector shifter)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY shifter IS
          PORT (  inp: IN     STD_LOGIC_VECTOR (3 DOWNTO 0);
                  sel: IN     INTEGER RANGE 0 TO 4;
                 outp: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END shifter;
---------------------------------------------
ARCHITECTURE shifter OF shifter IS
          SUBTYPE vector  IS STD_LOGIC_VECTOR (7 DOWNTO 0);
          TYPE         matrix IS  ARRAY (4 DOWNTO 0) OF vector;
          SIGNAL       row:  matrix;

BEGIN
          row(0) <= "0000" & inp;
          G1: FOR i IN 1 TO 4 GENERATE
                    row(i) <= row(i-1)(6 DOWNTO 0) & '0';
              END GENERATE;
          outp <= row(sel);
END shifter;
```

row(0): 0000$\underline{1111}$
row(1): 000$\underline{1111}$0
row(2): 00$\underline{1111}$00
row(3): 0$\underline{1111}$000
row(4): $\underline{1111}$0000

BLOCK

- Contains a set of concurrent statements
  - Make code more readable
- Simple BLOCK

```
Label: BLOCK
   [delcarative part]
BEGIN
   (concurrent statements)
END BLOCK Label;
```

- GUARDED BLOCK
  - Guarded statement executed only when guarded expression is TRUE

```
Lable: BLOCK (guarded expression)
   [delcarative part]
BEGIN
   (concurrent guarded and unguarded statements)
END BLOCK label;
```

Examples (simple and guarded blocks)

```
b1: BLOCK  -- simple block
          SIGNAL  a: STD_LOGIC;
     BEGIN
          a <= input_sig  WHEN ena ='1' ELSE 'z';
     END
END BLOCK b1;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--------------------------------
ENTITY latch IS
          PORT (d, clk: IN STD_LOGIC;
                              q: OUT STD_LOGIC);
END latch;
--------------------------------
ARCHITECTURE latch OF latch IS    -- latch
BEGIN
b1:       BLOCK (clk='1')  -- guarded block
          BEGIN
                    q <= GUARDED d; -- executed only when clk='1'
          END BLOCK b1;
END latch;
```

Example (synchronous reset DFF)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------
ENTITY dff IS
        PORT (d, clk, rst: IN    STD_LOGIC;
                       q: OUT STD_LOGIC);
END dff;
---------------------------------
ARCHITECTURE dff OF dff IS
BEGIN
                       -- DFF with synchronous reset
        b1: BLOCK (clk'EVENT AND clk='1')
        BEGIN
                q <= GUARDED '0' WHEN rst='1' ELSE d;
        END BLOCK b1;
END dff;
```

VHDL 2008

- concurrent code **WHEN** and **SELECT** can be used also in sequential code (inside **process** blocks)
- **WHEN** allows Boolean test, i.e., logic 1/0, in addition to the original true/false
- **SELECT?** was introduced to allows *don’t care* inputs

Boolean test in WHEN

```
-- traditional format

x <= '0' WHEN rst = '0' ELSE
      '1' WHEN a='0' OR b='1' ELSE
      '-' ;
```

```
-- VHDL 2008 support

x <= '0' WHEN NOT rst  ELSE
      '1' WHEN NOT a OR b ELSE
      '-' ;
```

don’t care in SELECT?

```
WITH interrupt SELECT?
   priority <= 4 WHEN “1---”,
            <= 3 WHEN “01--”,
            <= 2 WHEN “001-”,
            <= 1 WHEN “0001”,
            <= 0 WHEN OTHERS;
```

Summary of concurrent code

- Verilog
  - Continuous assignment (assign LHD = RHS) , ie.g.,

```
wire [7:0] a, b, c, w1, w2, w3;
wire s, carry;
assign w1 = a | b & c;  // & has higher priority order than |
 assign {carry, w2} = a + b;
assign w3 = s ? a : b;
```

- VHDL
  - Concurrent code (LHS <= RHS), e.g.,

```
library ieee;   use ieee.std_logic_unsigned;
signal a, b, c, w1, w2, w3: std_logic_vector (7 downto 0);
signal s, carry: std_logic;
signal out: std_logic_vector (15 downto 0);
w1 <= a or ( b and c );   -- and, or have same priority order
carry & w2 <= a + b;
w 3 <= a when s = ‘1’ else
          b;
```

VHDL
Sequential Code

Sequential codes

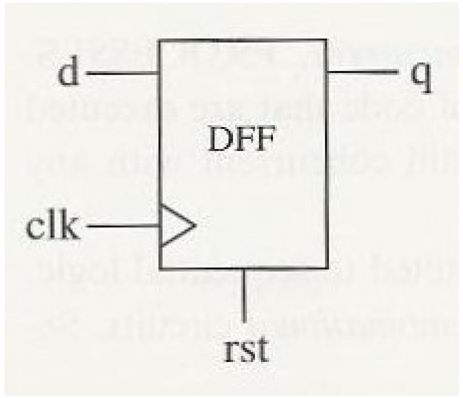
- Statements inside **PROCESS**, **PROCEDURE**, **FUNCTION** are executed sequentially
  - **IF / THEN / ELSE (ELSIF)**
  - **WAIT ON (FOR, UNTIL)**
  - **CASE / WHEN**
    - **cp. with WITH/SELECT/WHEN, WHEN/ELSE**
  - **FOR LOOP**
- cp. VHDL concurrent codes
  - logic operators such as **AND**, **XOR**
  - WHEN statement such as **WHEN/ELSE**, or **WITH/SELECT/WHEN**
  - **FOR/GENERATE**
  - **BLOCK**

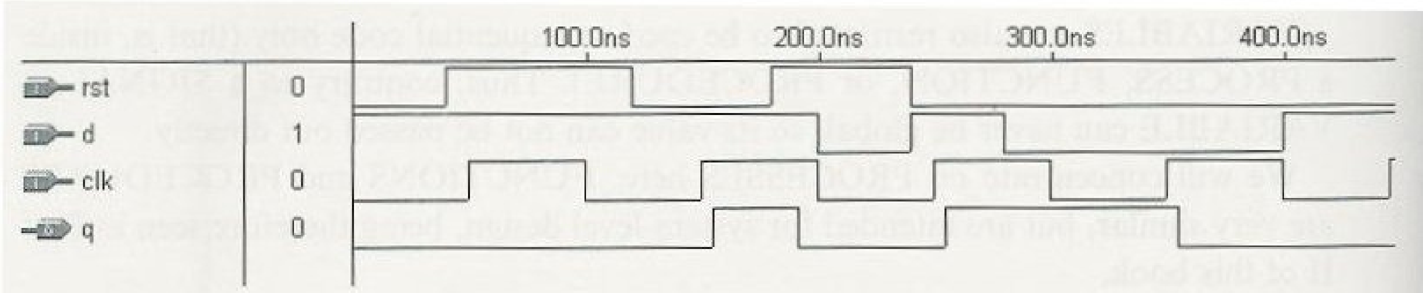
PROCESS

- Sequential section of VHDL code
  - With sequential statements of **IF, WAIT, CASE, LOOP**
  - Usually contains a sensitivity list (except for WAIT)
- PROCESS is executed every time a signal in the sensitivity list changes (or the condition related to WAIT is fulfilled)

Example (DFF with asynchronous reset)

```
LIBRARY ieee;  USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY dff  IS
PORT (d, clk, rst: IN STD_LOGIC; q: OUT STD_LOGIC);
END dff;
---------------------------------------
ARCHITECTURE behavior OF dff IS
BEGIN
    PROCESS (clk, rst)  -- sensitivity list clk, rst
    BEGIN
        IF (rst='1') THEN
                q <= '0';
        ELSIF (clk'EVENT AND clk='1') THEN
                q <= d;
        END IF;
    END PROCESS;
END behavior;
```





```
-- 8-bit register with synchronous reset
PROCESS      -- no sensitivity list
BEGIN
        WAIT UNTIL (clk’EVENT AND clk=‘1’);  -- first statement
        IF (rst = ‘1’) THEN output <= “00000000”;
        ELSIF (clk’EVENT AND clk=‘1’) THEN output <= input;
        END IF;
END PROCESS;
```

```
-- 8-bit register with asynchronous reset
PROCESS
BEGIN
        WAIT ON clk, rst;
        IF (rst = ‘1’) THEN output <= “00000000”;
        ELSIF (clk’EVENT AND clk=‘1’) THEN output <= input;
        END IF;
END PROCESS;
```

WAIT

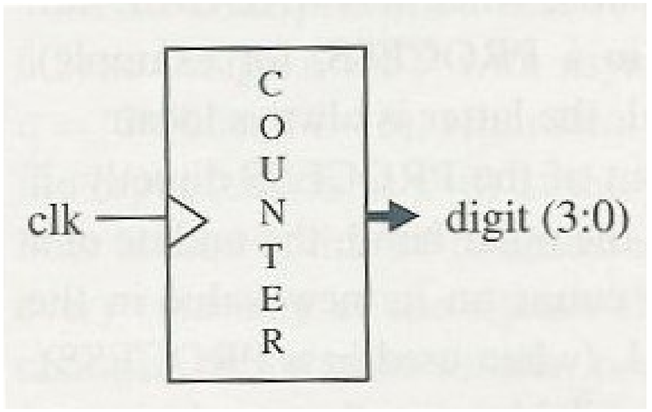
- PROCESS cannot have a sensitivity list when WAIT is employed
- **WAIT UNTIL** *signal_condition*;
  - wait until the condition is true
  - accepts **only one** signal
  - the first statement in the PROCESS
- **WAIT ON** *signal1*, *signal2*, …;
  - wait on at least one of the signals change
  - accepts **multiple signals**
- **WAIT FOR** *time*;
  - for simulation only (waveform generation for testbenches)

example (one-digit counter # 1)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY counter IS
        PORT (clk : IN STD_LOGIC;
                digit : OUT INTEGER RANGE 0 TO 9);
END counter;
---------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
        PROCESS         -- no sensitivity list
                VARIABLE temp : INTEGER RANGE 0 TO 10;
        BEGIN
                WAIT UNTIL (clk'EVENT AND clk='1');
                temp := temp + 1;
                IF (temp=10)  THEN  temp := 0;  END IF;
                digit <= temp;
        END PROCESS;
END counter;
```

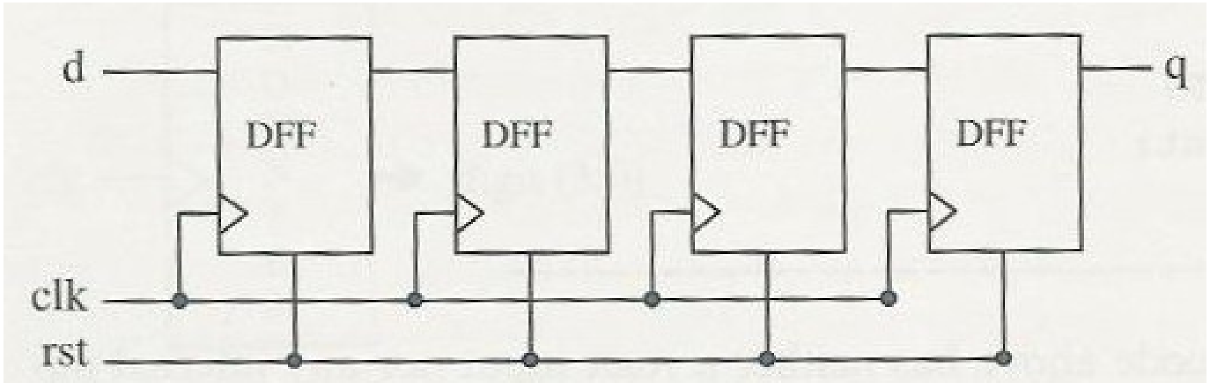
Example (one-digit counter #2)

```
LIBRARY ieee; USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY counter IS
PORT (clk : IN STD_LOGIC;    digit : OUT INTEGER RANGE 0 TO 9);
END counter;
---------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
        count: PROCESS (clk)
                VARIABLE temp : INTEGER RANGE 0 TO 10;
        BEGIN
                IF (clk'EVENT AND clk='1') THEN
                        temp := temp + 1;
                        IF (temp=10) THEN
                                temp := 0;
                        END IF;
                END IF;
                digit <= temp;
        END PROCESS count;
END counter;
```



Example (shift register)

```
ENTITY shiftreg IS
GENERIC (n: INTEGER := 4); -- # of stages
PORT (d, clk, rst: IN STD_LOGIC;    q: OUT STD_LOGIC);
END shiftreg;
--------------------------------------------------
ARCHITECTURE behavior OF shiftreg IS
        SIGNAL internal: STD_LOGIC_VECTOR (n-1 DOWNTO 0);
BEGIN
        PROCESS (clk, rst)
        BEGIN
                IF (rst='1') THEN
                        internal <= (OTHERS => '0');
                ELSIF (clk'EVENT AND clk='1') THEN
                        internal <= d & internal (internal'LEFT DOWNTO 1);
                END IF;
        END PROCESS;
        q <= internal(0);
END behavior;
```



CASE

- CASE (sequential) is similar to WHEN (combinational) in the concurrent codes
- but CASE allows *multiple assignments* in each test condition

```
CASE control IS
   WHEN  "00"        =>  x<=a;         y<=b;  -- two assignments
   WHEN  "01"        =>  x<=b;         y<=c;
   WHEN OTHERS   =>  x<="0000";   y<="zzzz";
END CASE;
```

  - WHEN / ELSE (concurrent statement) allows only *one assignment* in each condition

```
output <= "000"   WHEN (input='0' OR reset = '1') ELSE
          "001"  WHEN ctl = '1' ELSE
          "010";
```

FOR LOOP

```
[label:] FOR  identifier  IN  range  LOOP
        (sequential statements)
END LOOP [label];
```

```
FOR  i  IN  0 TO 5  LOOP
   x(i) <= enable AND w(i+2);
   y(0,i) <= w(i);
END LOOP;
```

```
FOR  i   IN   data'RANGE   LOOP
   CASE data(i) IS
      WHEN '0'           => count := count+1;
      WHEN OTHERS => EXIT;  -- terminate loop
   END CASE;
END LOOP;
```

```
FOR  i  IN  0 TO 15  LOOP
   NEXT WHEN i=skip;  -- jump to next iteration
    …
END LOOP;
```

WHILE / LOOP

[label:] **WHILE** condition **LOOP**

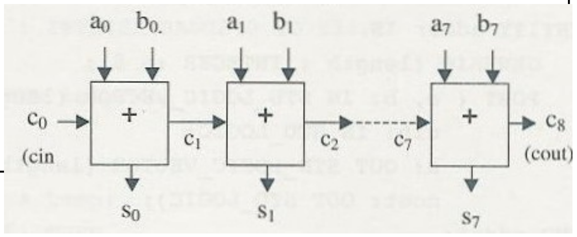
    (sequential statements)

END LOOP [label];

```
WHILE (i<10) LOOP
  WAIT UNTIL clk'EVENT AND clk='1';
  ....
END LOOP;
```

Example: adder

```
-- Solution 1: Generic, with VECTORS
LIBRARY ieee; USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY adder IS
GENERIC (length : INTEGER := 8);
PORT
(a, b: IN STD_LOGIC_VECTOR (length-1 DOWNTO 0);
cin: IN STD_LOGIC;
s: OUT STD_LOGIC_VECTOR (length-1 DOWNTO 0);
cout: OUT STD_LOGIC);
END adder;
---------------------------------------------
ARCHITECTURE ripple_carry_adder OF adder IS
BEGIN
PROCESS (a, b, cin)
VARIABLE carry : STD_LOGIC_VECTOR (length
DOWNTO 0);
BEGIN
carry(0) := cin;
FOR i IN 0 TO length-1 LOOP
   s(i) <= a(i) XOR b(i) XOR carry(i);
   carry(i+1) :=(a(i) AND b(i)) OR (a(i) AND  carry(i))
                    OR (b(i) AND carry(i));
END LOOP;
cout <= carry(length);
END PROCESS;
END adder;
```



```
-- Solution 2: non-generic, with INTEGERS
LIBRARY ieee;  USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY adder IS
PORT (a, b: IN INTEGER RANGE 0 TO 255;
             c0: IN STD_LOGIC;
               s: OUT INTEGER RANGE 0 TO 255;
             c8: OUT STD_LOGIC);
END adder;
---------------------------------------------
ARCHITECTURE adder OF adder IS
BEGIN
PROCESS (a, b, c0)
VARIABLE temp : INTEGER RANGE 0 TO 511;
BEGIN
IF (c0='1') THEN temp:=1;
                ELSE temp:=0;
END IF;
temp := a + b + temp;  -- behavioral description
IF (temp > 255) THEN c8 <= '1'; temp := temp – 256;
                         ELSE c8 <= '0';
END IF;
s <= temp;
END PROCESS;
END adder;
```

CASE vs. IF

- **IF**
  - usually infer priority decoder
  - however, optimization during synthesis might generate multiplexers
- **CASE**
  - never infer priority decoder
- After optimization, the following two codes implement the same physical multiplexer.

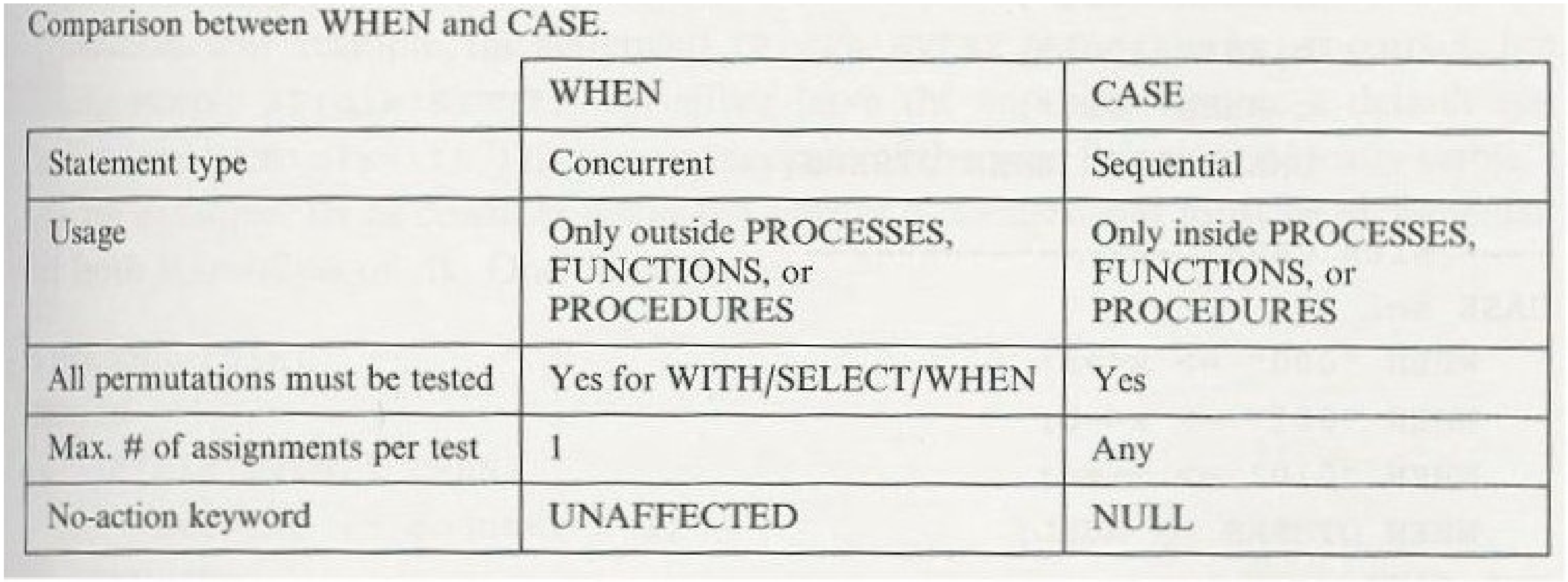
```
IF (sel="00") THEN x <= a;
ELSIF (sel="01") THEN x <= b;
ELSIF (sel="10") THEN x <= c;
ELSE x <= d;
```

```
CASE sel IS
WHEN "00"              => x <= a;
WHEN "01"              => x <= b;
WHEN "10"              => x <= c;
WHEN OTHERS            => x <= d;
END CASE;
```

CASE vs. WITHSELECT/WHEN

```
-- out side of PROCESS
WITH sel SELECT
x <= a            WHEN “000”,
     b            WHEN “001”,
     c            WHEN “010”,
     UNAFFECTED   WHEN OTHERS;
```

```
-- inside PROCESS
CASE sel IS
WHEN “000”     => x<=a;
WHEN “001”     => x<=b;
WHEN “010”     => x<=c;
WHEN OTHERS => NULL;
END CASE;
```



Comparison between WHEN and CASE.

	WHEN	CASE
Statement type	Concurrent	Sequential
Usage	Only outside PROCESSES, FUNCTIONS, or PROCEDURES	Only inside PROCESSES, FUNCTIONS, or PROCEDURES
All permutations must be tested	Yes for WITH/SELECT/WHEN	Yes
Max. # of assignments per test	1	Any
No-action keyword	UNAFFECTED	NULL

Bad clocking

- Assignments to the same signal at both transitions of the clock signal
  - Not synthesizable (no double-edge FFs available)
- **EVENT** attribute must be related to a test condition
  - Cannot use clk'**EVENT** only

```
PROCESS (clk)
BEGIN
   IF (clk'EVENT and clk='1') THEN
      counter <= counter +1;
   EISIF (clk'EVENT and clk='0') THEN
      counter <= counter +1;
   END IF;
END PROCESS;
-- need double-edge trigger FFs
-- signal counter is multiple driven
```

```
PROCESS (clk)
BEGIN
   IF (clk'EVENT) THEN
      counter <= counter +1;
END IF;
END PROCESS;
-- compiler might assume a default
-- test value, say "AND clk='1', or
-- error message "clk not locally stable"
```

Bad clocking (cont.)

- A signal in sensitivity list, but not in the PROCESS
  - the signal might be ignored
- use two-process code with assignments to different signals at clock rising edge and fall edge respectively

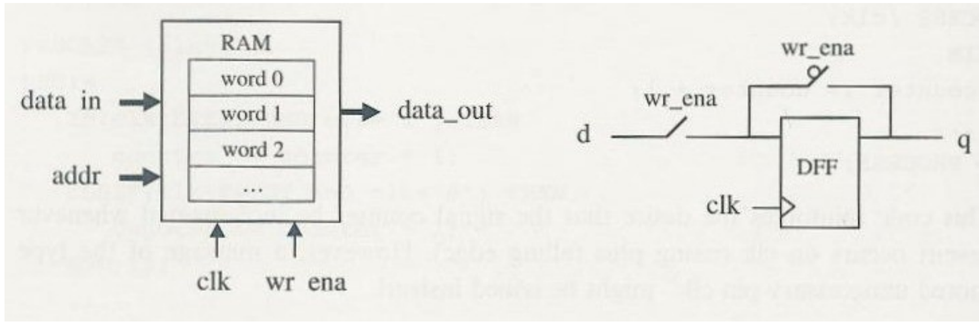
```
PROCESS (clk)
BEGIN
   counter <= counter +1;
END PROCESS;

--- clk does not appear in the PROCESS
--- this PROCESS might be ignored
```

```
--- correct 2-PROCESS code -----
PROCESS (clk)
BEGIN
   IF (clk'EVENT and clk='1') THEN
      x <= d;
   END IF;
END PROCESS;
---------------------------------------------------
PROCESS (clk)
BEGIN
   IF (clk'EVENT and clk='0') THEN
      y <= d;
   END IF;
END PROCESS;
```

Example: RAM (register-based)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------
ENTITY ram IS
GENERIC (bits: INTEGER := 8;                          -- # of bits per word
             words: INTEGER := 16);       -- # of words in the memory
PORT (wr_ena, clk: IN STD_LOGIC;
         addr: IN INTEGER RANGE 0 TO words-1;
         data_in: IN STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
         data_out: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
END ram;
---------------------------------------------------
ARCHITECTURE ram OF ram IS
TYPE vector_array IS ARRAY (0 TO words-1) OF STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
SIGNAL memory: vector_array;
BEGIN
PROCESS (clk, wr_ena)
BEGIN
IF (wr_ena='1') THEN
            IF (clk'EVENT AND clk='1') THEN
                        memory(addr) <= data_in;
            END IF;
END IF;
END PROCESS;
data_out <= memory(addr);
END ram;
```

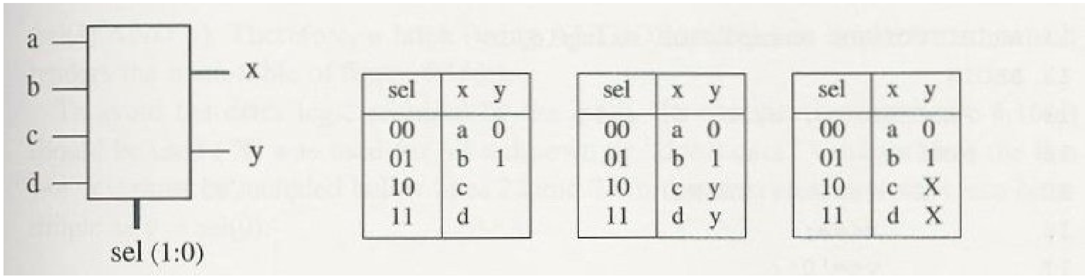


Completely specified truth table for combinational circuits

- All input signals used in the PROCESS appear in its sensitivity list
- All combinations of the input/output are included
- Otherwise, latches might be inferred

Bad combinational design

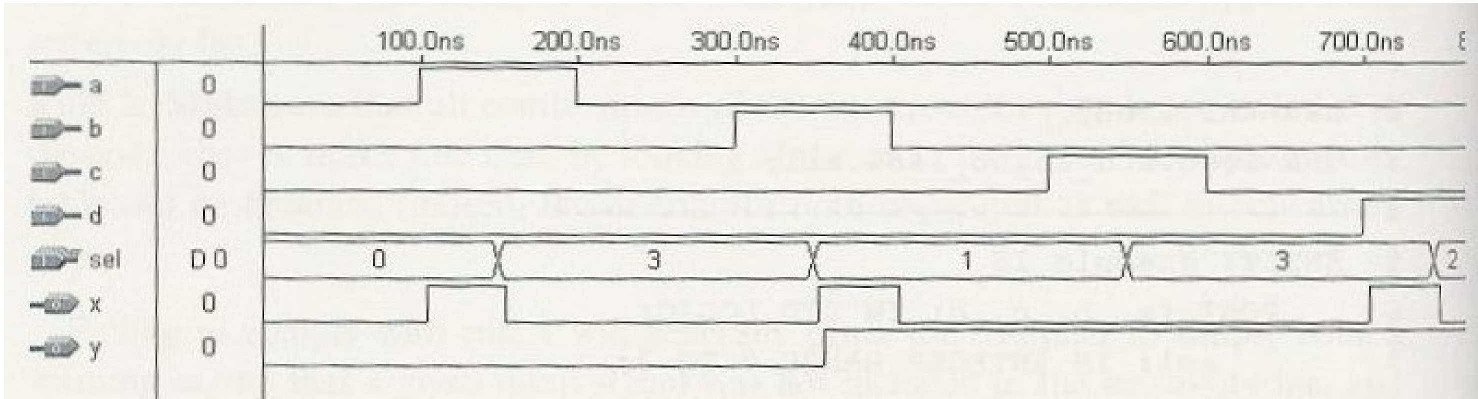
```
ENTITY example IS
PORT (a, b, c, d: IN STD_LOGIC;
sel: IN INTEGER RANGE 0 TO 3;
x, y: OUT STD_LOGIC);
END example;
-----------------------------------------
ARCHITECTURE example OF example IS
BEGIN
PROCESS (a, b, c, d, sel)
BEGIN
IF (sel=0)       THEN x<=a; y<='0';
ELSIF (sel=1) THEN x<=b;y<='1';
ELSIF (sel=2) THEN x<=c;           -- y is not specified when sel=2, infer latched y
ELSE x<=d;                          -- y is not specified when sel=3, infer latched y
END IF;
END PROCESS;
END example;
```



sel	x	y
00	a	0
01	b	1
10	c	
11	d	

sel	x	y
00	a	0
01	b	1
10	c	y
11	d	y

sel	x	y
00	a	0
01	b	1
10	c	X
11	d	X



VHDL 2008

- keyword **ALL** is allowed in PROCESS sensitivity list
- concurrent statements **WHEN** and **SELECT** are allowed in sequential code
- **IF** allows Boolean test (with logic 1/0) in addition to the original true/false test
- **CASE?** was introduced to allow *don’t care* input

Keyword ALL in PROCESS

-- reduce errors when implementing

-- combinational circuits with sequential codes

```
PROCESS (ALL)  -- Verilog: always @ (*)
BEGIN
… combinational logic
END PROCESS;
```

WHEN and SELECT in sequential code

```
-- traditional format


-- synchronous reset FF
PROCESS (clk OR clr)
BEGIN

IF (clk'EVENT AND clk='1') THEN
   IF (clr='1') THEN q <= '0';
   ELSE q <= d;
   END IF;
END IF;

END PROCESS;
```

```
-- VHDL 2008
-- shorter code


PROCESS (clk OR clr)
BEGIN

IF (clk'EVENT AND clk='1') THEN
   q <= '0 WHEN clr='1' ELSE
            d;
END IF;


END PROCESS;
```

Boolean test in IF statement

```
-- traditional format
IF (a=‘0’ AND b=‘0’) OR c=‘1’ THEN …

-- Boolean test supported in VHDL 2008
IF (NOT a AND NOT b) OR c THEN …
```

CASE?

```
-- concurrent code

WITH interrupt SELECT?
   priority <= 4 WHEN "1---",
            <= 3 WHEN "01--",
            <= 2 WHEN "001-",
            <= 1 WHEN "0001",
            <= 0 WHEN OTHERS;
```

```
-- sequential code

PROCESS (interrupt)
BEGIN

CASE? interrupt IS
   WHEN "1---"       =>  priority <= 4;
   WHEN "01--"       =>  priority <= 4;
   WHEN "001-"       =>  priority <= 4;
   WHEN "0001"       =>  priority <= 4;
   WHEN OTHERS =>  priority <= 4;
END CASE;

END PROCESS;
```

Summary of sequential code

- Verilog: procedural statements inside always blocks

```
wire [7:0] a, b, c;  wire s;
reg [7:0] w1, w2, w3;    reg carry;
always @ * begin
 w1 = a | b & c;  // & has higher priority order than |
 {carry, w2} = a + b;
w3 = s ? a : b;   end
```

- VHDL: sequential statements inside process blocks

```
library ieee;   use ieee.std_logic_unsigned;
signal a, b, c, w1, w2, w3, w4: std_logic_vector (7 downto 0);
signal s, carry: std_logic;
signal out: std_logic_vector (15 downto 0);
process all begin
variable tmp: std_logic_vector (7 down to 0) := “0000_0000”;
w1 <= a or ( b and c );   -- and, or have same priority order
carry & w2 <= a + b;
if s = ‘1’ then w3 <= a else w3 <= b; end if;
for i in tmp’range loop   if a(i) = ‘1’ then tmp := tmp+1; end if;    end loop;
w4 <= tmp;     end process;
```

signals and variables

CONSTANT

- Establish default values
- Declared in a PACKAGE, ENTITY, or ARCHITECTURE

CONSTANT name : type **:=** value;

```
CONSTANT set_bit : BIT := '1';
Type memory is array (0 to 2) of std_logic_vector (3 downto 0);
CONSTANT datamemory : memory := ( ('0', '0', '0', '0'),
                                  ('0', '0', '0', '1'),
                                  ('0', '0', '1', '1') );
-- used to describe ROM
```

SIGNAL

- Pass values in and out the circuit, as well as between its internal units
  - Represent real circuit interconnections (wires)
- *SIGNAL's update (inside PROCESS) is not immediate* (unlike VARIABLE whose value is updated immediately)

SIGNAL name : type [range] [:= initial value]

```
SIGNAL control : BIT := '0';
SIGNAL   count : INTEGER RANGE 0 TO 100;
SIGNAL       y : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

VARIABLE

- represent local information
  - used in PROCESS, FUNCTION, PROCEDURE
  - cannot be passed out directly
    - usually assigned to a SIGNAL before leaving process construct
  - *value update is immediate*

VARIABLE name : type [range] [:= init_value]

```
VARIABLE control : BIT := ‘0’;
VARIABLE   count : INTEGER RANGE 0 TO 100;
VARIABLE       y : STD_LOGIC_VECTOR (7 DOWNTO 0) := “10001000”;
```

SIGNAL vs. VARIABLE

- **SIGNAL (<=)**
  - Declared in a PACKAGE, ENTITY, ARCHITECTURE
  - *Global*
  - *Value updated after the conclusion of the present run of the PROCESS*
  - similar to Verilog non-blocking assign **<=**

- **VARIABLE (:=)**
  - Declared inside a Sequential code (such as PROCESS)
  - *Local*
  - *Value updated immediately*
  - similar to Verilog blocking assignment **=**
  - Must be assigned to a SIGNAL when VARIABLE passed out of the PROCESS directly

Examples: count ones

```
ENTITY count_ones IS
PORT (din: IN STD_LOGIC_VECTOR (7
DOWNTO 0);
ones: OUT INTEGER RANGE 0 TO 8);
END count_ones;
----------------------------------------
ARCHITECTURE not_ok OF count_ones IS
SIGNAL temp: INTEGER RANGE 0 TO 8;
BEGIN
PROCESS (din)
BEGIN

temp <= 0;
FOR i IN 0 TO 7 LOOP
   IF (din(i)='1') THEN temp <= temp + 1;
   END IF;
END LOOP;

ones <= temp;
END PROCESS;
END not_ok;

-- signal temp is NOT updated immediately
-- temp is multiple driven (1+8) times)
```

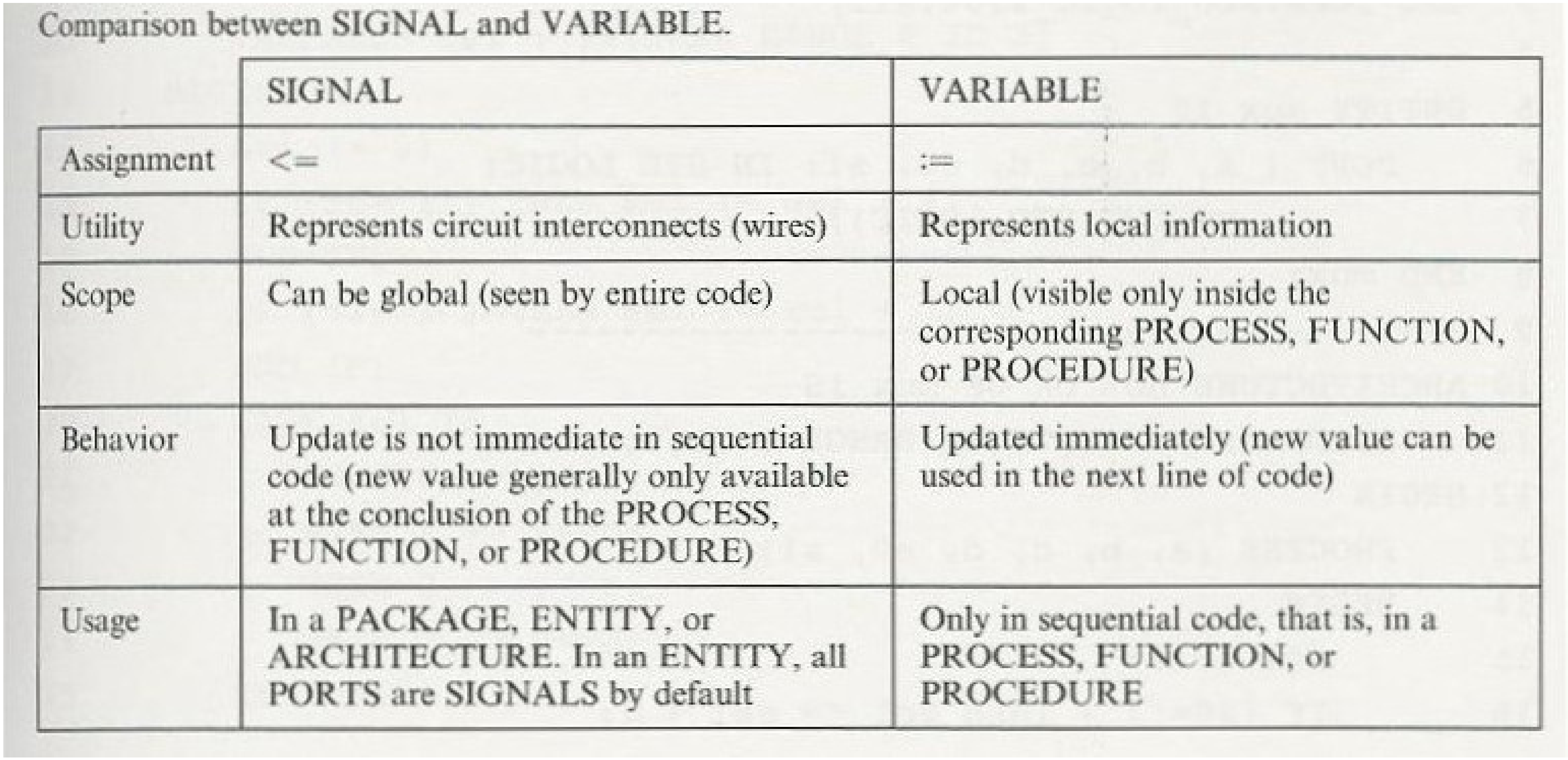
```
ENTITY count_ones IS
PORT (din: IN STD_LOGIC_VECTOR (7
DOWNTO 0);
ones: OUT INTEGER RANGE 0 TO 8);
END count_ones;
----------------------------------------
ARCHITECTURE ok OF count_ones IS
BEGIN
PROCESS (din)
VARIABLE temp: INTEGER RANGE 0 TO 8;
BEGIN

temp := 0;
FOR i IN 0 TO 7 LOOP
   IF (din(i)='1') THEN temp := temp + 1;
   END IF;
END LOOP;

ones <= temp;
END PROCESS;
END ok;

-- variable temp is updated immediately
-- no multiple-driven problem
```

SIGNAL vs. VARIABLE


Comparison between SIGNAL and VARIABLE.

	SIGNAL	VARIABLE
Assignment	<=	:=
Utility	Represents circuit interconnects (wires)	Represents local information
Scope	Can be global (seen by entire code)	Local (visible only inside the corresponding PROCESS, FUNCTION, or PROCEDURE)
Behavior	Update is not immediate in sequential code (new value generally only available at the conclusion of the PROCESS, FUNCTION, or PROCEDURE)	Updated immediately (new value can be used in the next line of code)
Usage	In a PACKAGE, ENTITY, or ARCHITECTURE. In an ENTITY, all PORTS are SIGNALS by default	Only in sequential code, that is, in a PROCESS, FUNCTION, or PROCEDURE

Process using variable

- values of variables updated immediately

```
entity dummy is
end dummy;

architecture var of dummy is
      signal trigger, sum: integer:=0;
begin
      process
      variable var1: integer:=1;
      variable var2: integer:=2;
      variable var3: integer:=3;
      begin
            wait on trigger;
            var1 := var2 + var3;               var1 = 2 + 3 = 5
            var2 := var1;                      var2 = 5
            var3 := var2;                      var3 = 5
            sum <= var1 + var2 + var3;         sum = 5 + 5 + 5 = 15 (after Δ)
      end process;
end var;
```

process using signal

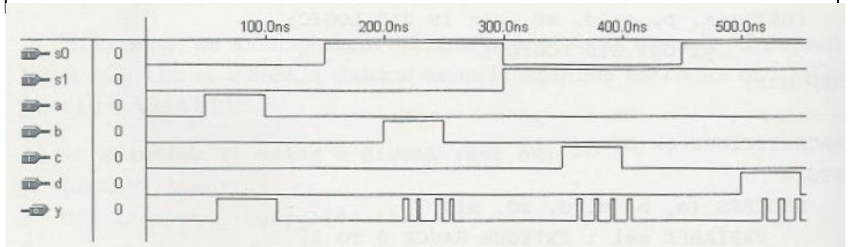
- signal values changed only after the end of current run

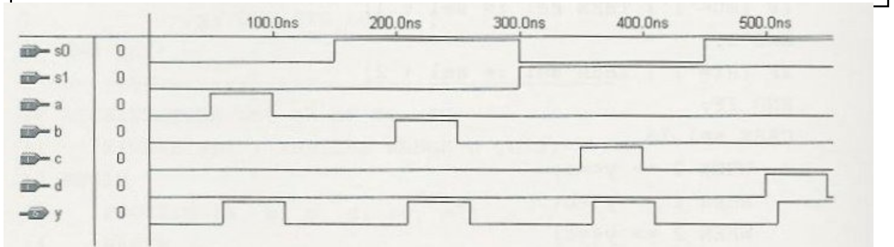
```
architecture sig of dummy is
      signal trigger, sum: integer:=0;
      signal sig1: integer:=1;
      signal sig2: integer:=2;
      signal sig3: integer:=3;
begin
      process
      begin
         wait on trigger;
         sig1 <= sig2 + sig3;                sig1 = 2 + 3 = 5       (after Δ)
         sig2 <= sig1;                       sig2 = 1               (after Δ)
         sig3 <= sig2;                       sig3 = 2               (after Δ)
         sum <= sig1 + sig2 + sig3;          sum = 1 + 2 + 3 = 6    (after Δ)
      end process;
end sig;
```

Example: MUX

```
ENTITY mux IS
PORT (a, b, c, d, s0, s1: IN STD_LOGIC;
y: OUT STD_LOGIC);
END mux;
-----------------------------------------
ARCHITECTURE not_ok OF mux IS
SIGNAL sel : INTEGER RANGE 0 TO 3;
BEGIN
PROCESS (a, b, c, d, s0, s1)
BEGIN
sel <= 0;
IF (s0='1') THEN sel <= sel + 1; END IF;
IF (s1='1') THEN sel <= sel + 2; END IF;
CASE sel IS
WHEN 0 => y<=a;
WHEN 1 => y<=b;
WHEN 2 => y<=c;
WHEN 3 => y<=d;
END CASE;
END PROCESS;
END not_ok;
```

```
ENTITY mux IS
PORT (a, b, c, d, s0, s1: IN STD_LOGIC;
y: OUT STD_LOGIC);
END mux;
-----------------------------------------
ARCHITECTURE ok OF mux IS
BEGIN
PROCESS (a, b, c, d, s0, s1)
VARIABLE sel : INTEGER RANGE 0 TO 3;
BEGIN
sel := 0;
IF (s0='1') THEN sel := sel + 1; END IF;
IF (s1='1') THEN sel := sel + 2; END IF;
CASE sel IS
WHEN 0 => y<=a;
WHEN 1 => y<=b;
WHEN 2 => y<=c;
WHEN 3 => y<=d;
END CASE;
END PROCESS;
END ok;
```

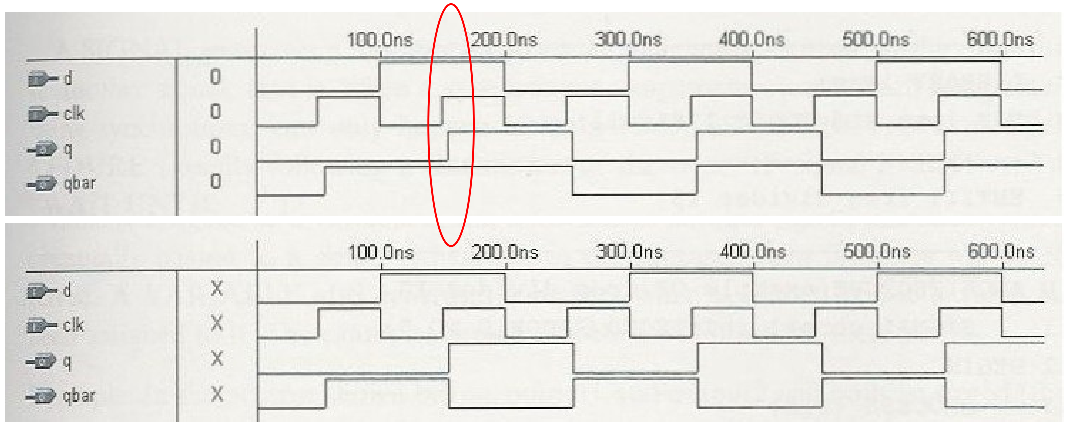




Example: DFF with q and qbar

```
ENTITY dff IS
PORT (d, clk: IN STD_LOGIC;
q: BUFFER STD_LOGIC;
qbar: OUT STD_LOGIC);
END dff;
----------------------------------------
ARCHITECTURE not_ok OF dff IS
BEGIN
PROCESS (clk)
BEGIN
IF (clk'EVENT AND clk='1') THEN
q <= d;
qbar <= NOT q;
END IF;
END PROCESS;
END not_ok;
-- qbar is not the inverse of signal q
```

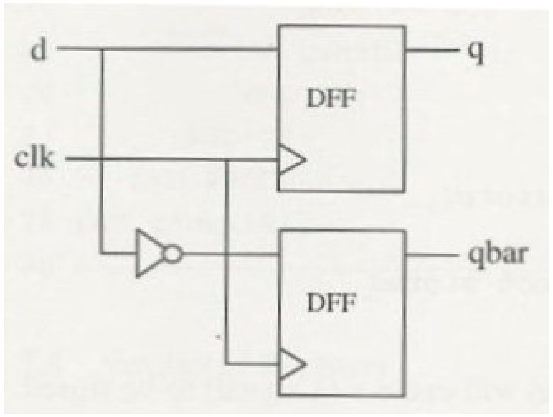
```
ENTITY dff IS
PORT (d, clk: IN STD_LOGIC;
q: BUFFER STD_LOGIC;
qbar: OUT STD_LOGIC);
END dff;
----------------------------------------
ARCHITECTURE ok OF dff IS
BEGIN
PROCESS (clk)
BEGIN
IF (clk'EVENT AND clk='1') THEN
q <= d;
END IF;
END PROCESS;
qbar <= NOT q;
END ok;
-- qbar assignment is concurrent statement
```

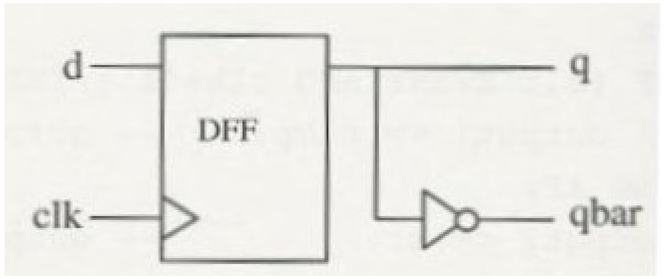


example: DFF with q and qbar

```
ENTITY dff IS
PORT (d, clk: IN STD_LOGIC;
q: OUT STD_LOGIC;
qbar: OUT STD_LOGIC);
END dff;
-------------------------------------------
ARCHITECTURE two_dff OF dff IS
BEGIN
PROCESS (clk)
BEGIN
IF (clk'EVENT AND clk='1') THEN
q <= d;     -- generates a register
qbar <= NOT d;      -- generates a register
END IF;
END PROCESS;
END two_dff;
```

```
ENTITY dff IS
PORT (d, clk: IN STD_LOGIC;
q: BUFFER STD_LOGIC;
qbar: OUT STD_LOGIC);
END dff;
-------------------------------------------
ARCHITECTURE one_dff OF dff IS
BEGIN
PROCESS (clk)
BEGIN
IF (clk'EVENT AND clk='1') THEN
q <= d;     -- generates a register
END IF;
END PROCESS;
qbar <= NOT q; -- uses logic gate NOT
END one_dff;
```





Inferring of registers

```
PROCESS (clk)
BEGIN
  IF (clk’EVENT and clk=‘1’) THEN
          output1 <= temp;
          output2 <= a;
END IF;
END PROCESS;
--- both output1 and output2 are stored (inferring TWO flip-flops)
```

```
PROCESS (clk)
BEGIN
IF (clk’EVENT and clk=‘1’) THEN output1 <= temp; END IF;
output2 <= a;
END PROCESS;
-- output1 stored in a flip-flop, output2 not stored
```

```
PROCESS (clk)
VARIABLE temp : BIT;
BEGIN
IF (clk’EVENT and clk=‘1’) THEN temp := a; END IF;
x <= temp;
END PROCESS;
-- signal x is stored when it leaves the PROCESS
```

example: counter #1, #2

count: out

```
ENTITY counter IS
PORT (clk, rst: IN BIT;
count: OUT INTEGER RANGE 0 TO 7);
END counter;
--------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
PROCESS (clk, rst)
VARIABLE temp: INTEGER RANGE 0 TO 7;
BEGIN
IF (rst='1') THEN temp:=0;
ELSIF (clk'EVENT AND clk='1') THEN
temp := temp+1;
END IF;
count <= temp;
END PROCESS;
END counter;
```

count: buffer

```
ENTITY counter IS
PORT (clk, rst: IN BIT;
count: BUFFER INTEGER RANGE 0 TO 7);
END counter;
--------------------------------------------
ARCHITECTURE counter OF counter IS
-- SIGNAL temp: INTEGER RANGE 0 TO 7;
BEGIN
PROCESS (clk, rst)
BEGIN
IF (rst='1') THEN count <= 0;
ELSIF (clk'EVENT AND clk='1') THEN
count <= count + 1;
END IF;
END PROCESS;
END counter;
```

INOUT: bidirectional input/output port

BUFFER: a signal is sent out but it must be used (read) internally

example: counter #2, #3

- two coding methods

count: buffer

```
ENTITY counter IS
PORT (clk, rst: IN BIT;
count: BUFFER INTEGER RANGE 0 TO 7);
END counter;
---------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
PROCESS (clk, rst)
BEGIN
IF (rst='1') THEN count <= 0;
ELSIF (clk'EVENT AND clk='1') THEN
count <= count + 1;
END IF;
END PROCESS;
END counter;
```

count: out

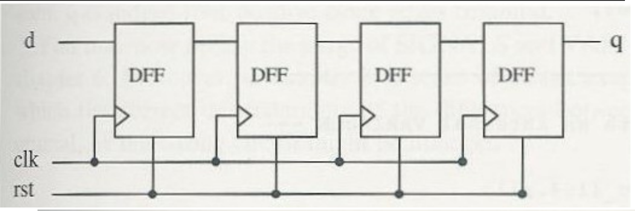
```
ENTITY counter IS
PORT (clk, rst: IN BIT;
count: OUT INTEGER RANGE 0 TO 7);
END counter;
---------------------------------------------
ARCHITECTURE counter OF counter IS
SIGNAL temp: INTEGER RANGE 0 TO 7;
BEGIN
PROCESS (clk, rst)
BEGIN
IF (rst='1') THEN count <= 0;
ELSIF (clk'EVENT AND clk='1') THEN
temp <= temp +1;
END IF;
END PROCESS;
count <= temp;
END counter;
```

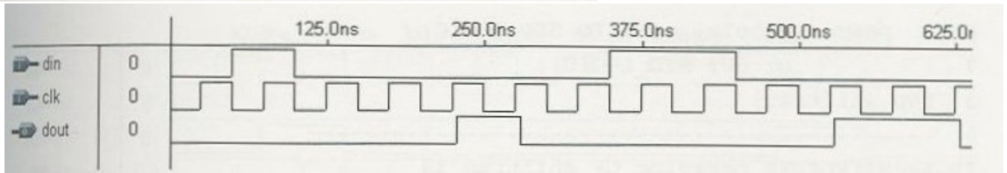
example: shift register #1

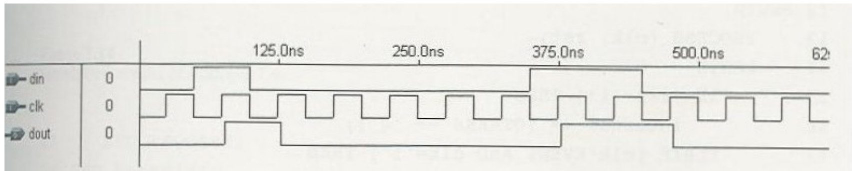
```
ENTITY shift IS PORT (din, clk:
IN BIT; dout: OUT BIT); END
shift;_
----------------------------------
ARCHITECTURE shift_OK1
OF shift IS BEGIN
PROCESS (clk)
VARIABLE a, b, c: BIT;
BEGIN
IF (clk'EVENT AND clk='1')
THEN                    -- OK
dout <= c;
c := b;
b := a;
a := din;
END IF;
END PROCESS;
END shift;
```

```
ENTITY shift IS PORT (din, clk:
IN BIT;dout: OUT BIT);
END shift;
-------------------------------------
ARCHITECTURE shift_OK2
OF shift IS
SIGNAL a, b, c: BIT;
BEGIN
PROCESS (clk)
BEGIN
IF (clk'EVENT AND clk='1')
THEN                      -- OK
a <= din;
b <= a;
c <= b;
dout <= c;
END IF;
END PROCESS;
END shift;
```

```
ENTITY shift IS PORT (din, clk:
IN BIT;dout: OUT BIT);
END shift;
------------------------------------
ARCHITECTURE
shift_not_OK OF shift IS
BEGIN
PROCESS (clk)
VARIABLE a, b, c: BIT;
BEGIN
IF (clk’EVENT AND clk=’1’)
THEN                  -- not OK
a := din;
b := a;
c := b;
dout <= c;
END IF;
END PROCESS;
END shift;
```



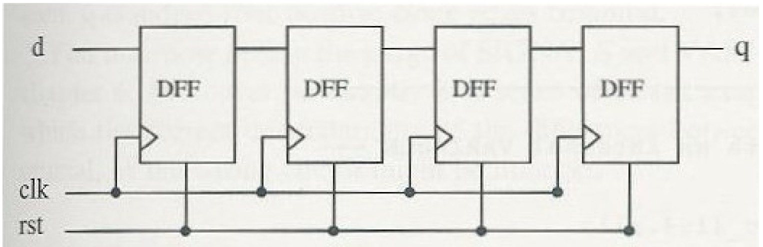


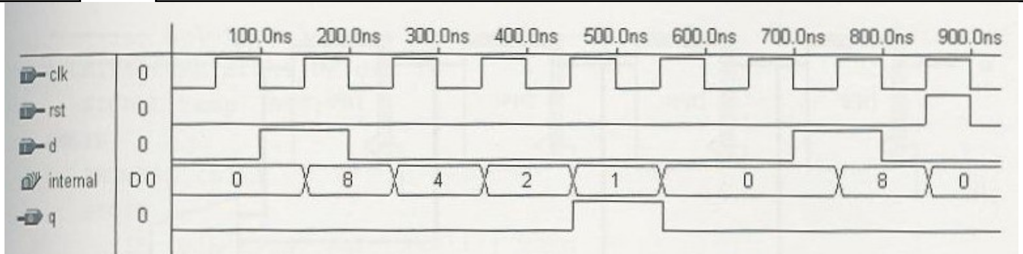


example: shift register #2 (both signal and variable are OK)

```
ENTITY shiftreg IS
PORT (d, clk, rst: IN STD_LOGIC;
                q: OUT STD_LOGIC);
END shiftreg;
---------------------------------------------
ARCHITECTURE behavior OF shiftreg IS
SIGNAL internal: STD_LOGIC_VECTOR
(3 DOWNTO 0);
BEGIN
PROCESS (clk, rst)
BEGIN
IF (rst='1') THEN
internal <= (OTHERS => '0');
ELSIF (clk'EVENT AND clk='1') THEN
internal <= d & internal(3 DOWNTO 1);
END IF;
END PROCESS;
q <= internal(0);
END behavior;
```

```
ENTITY shiftreg IS
PORT (d, clk, rst: IN STD_LOGIC;
                q: OUT STD_LOGIC);
END shiftreg;
---------------------------------------------
ARCHITECTURE behavior OF shiftreg IS
BEGIN
PROCESS (clk, rst)
VARIABLE internal: STD_LOGIC_VECTOR
(3 DOWNTO 0);
BEGIN
IF (rst='1') THEN
internal := (OTHERS => '0');
ELSIF (clk'EVENT AND clk='1') THEN
internal := d & internal(3 DOWNTO 1);
END IF;
q <= internal(0);
END PROCESS;
END behavior;
```





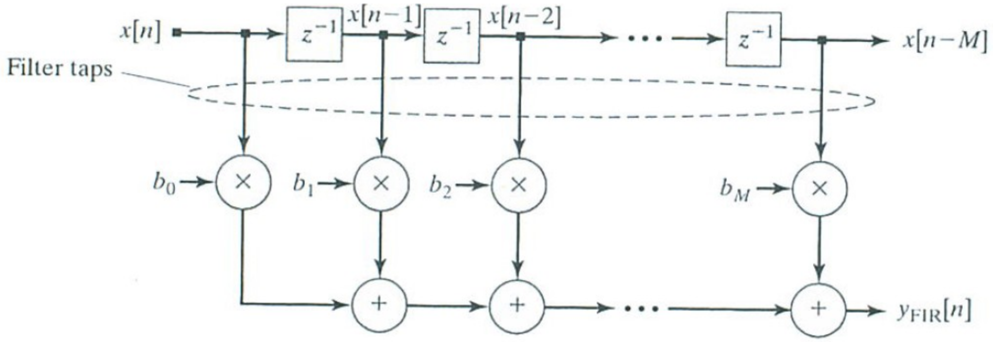
summary for signal vs. variable

- signal
  - correspond to real hardware wire
  - update is NOT immediately
  - caution for *multiple driven* problems
  - at clock edge, infer registers
- variable
  - used only inside process blocks
  - update is immediately (similar to C code)
    - easy for describing hardware operations
  - need to pass to signal before leaving process block

Digital filter

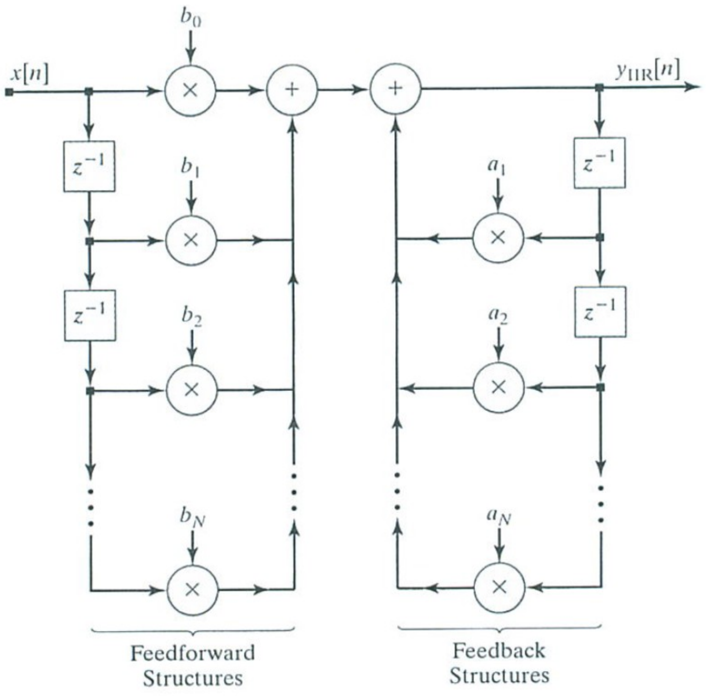
- Low-pass, high-pass, or bandpass filter
- Finite Impulse Response (FIR) filter

$$y_{FIR}[n] = \sum_{k=0}^{M} b_k x[n-k]$$



- Infinite Impulse Response (IIR) filter

$$y_{IIR}[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$



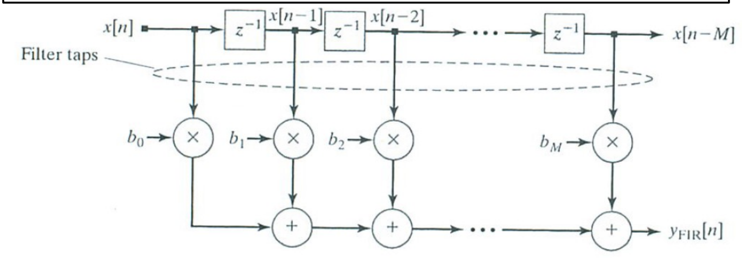
FIR Filter (Verilog)

- M shift register storing x[n-1], xpn-2[, …, x[n-M]
- for-loop for accumulation of pM+1 roducts
  - here, output is NOT stored in register

```
// wrong codes,
// Why?

…
reg [word_size_in-1:0] x[0:M];
reg [word_size_out-1:0] y;

y=0;
always @ (x)
begin
   for (i=0; i<=M; i++)
      y = y + x[i]*b[i];
end
…
```



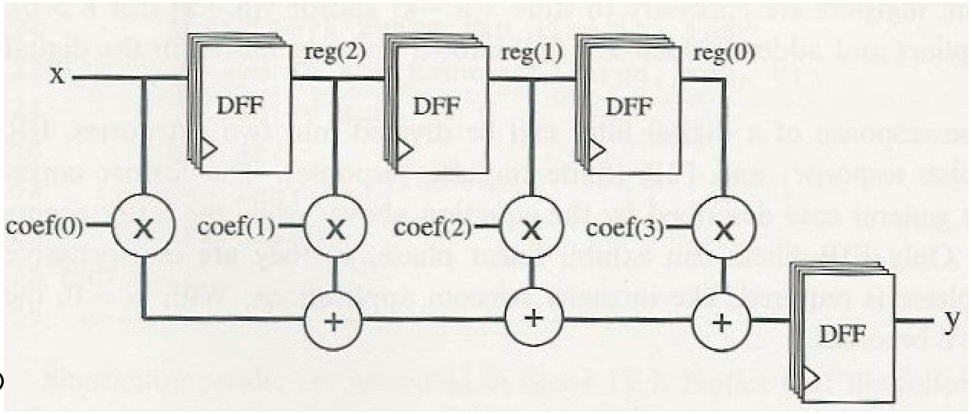
```
…
reg [word_size_in-1:0] x[0:M], temp[0:M+1];
reg [word_size_out-1:0] y;

temp[0]=0;
always @ (*)
begin
   for (i=0; i<=M; i++)
      temp[i+1] = temp[i] + x[i]*b[i];
end
assign  y = temp[M+1];  // un-stored output, or
//  always @ (posedge clk) y <= temp[M+1];


reg [15:0] x[0:M+1];
// shift registers that takes one input at each cycle
always @ (posedge clk)
   x[0:M+1] <= {in, x[0:M]};
…
```

FIR Filter (VHDL)

```
-------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all; -- package needed for SIGNED
---------------------------------------------------------
ENTITY fir IS
   GENERIC (n: INTEGER := 4; m: INTEGER := 4);
   -- n = # of coef., m = # of bits of input and coef.
   -- Besides n and m, CONSTANT (line 19) also need adjust
   PORT (x: IN SIGNED(m-1 DOWNTO 0);
                          clk, rst: IN STD_LOGIC;
                          y: OUT SIGNED(2*m-1 DOWNTO 0));
END fir;
---------------------------------------------------------
ARCHITECTURE rtl OF fir IS
   TYPE registers IS ARRAY (n-2 DOWNTO 0) OF SIGNED(m-1 DOWNTO 0);
   TYPE coefficients IS ARRAY (n-1 DOWNTO 0) OF SIGNED(m-1 DOWNTO 0)
   SIGNAL reg: registers;
   CONSTANT coef: coefficients := ("0001", "0010", "0011", "0100");
BEGIN
   PROCESS (clk, rst)
             VARIABLE acc, prod: SIGNED(2*m-1 DOWNTO 0) := (OTHERS=>'0');
             VARIABLE sign: STD_LOGIC;
   BEGIN
             ----- reset: --------------------------
             IF (rst='1') THEN
                          FOR i IN n-2 DOWNTO 0 LOOP
                                      FOR j IN m-1 DOWNTO 0 LOOP
                                                  reg(i)(j) <= '0';
                                      END LOOP;
                          END LOOP;
             ----- register inference + MAC: -------
             ELSIF (clk'EVENT AND clk='1') THEN
                          acc := coef(0)*x;
                          FOR i IN 1 TO n-1 LOOP
                                      sign := acc(2*m-1);
                                      prod := coef(i)*reg(n-1-i);
                                      acc := acc + prod;
                                      ---- overflow check: ------------
                                      IF (sign=prod(prod'left)) AND (acc(acc'left) /= sign)
                                                  THEN
                                                  acc := (acc'LEFT => sign, OTHERS => NOT sign);
                                      END IF;
                          END LOOP;
                          reg <= x & reg(n-2 DOWNTO 1);
             END IF;
             y <= acc;
   END PROCESS;
END rtl;
```



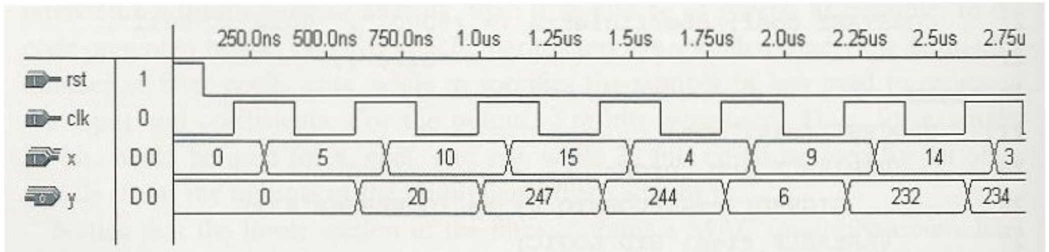
-- n-1 shift registers

-- m-bit data, coeff.

-- for-loop with variable acc

-- to accumulate n products

-- saturation accmulator



Quiz

- write a complete Verilog code that corresponds to the previous VHDL FIR filter
  - signed data type
  - reset for shift registers
  - saturation for accumulator

finte state machines (FSM)

single phase state machine

- Mealy machine

$output = \quad ent\_state,\ input)$

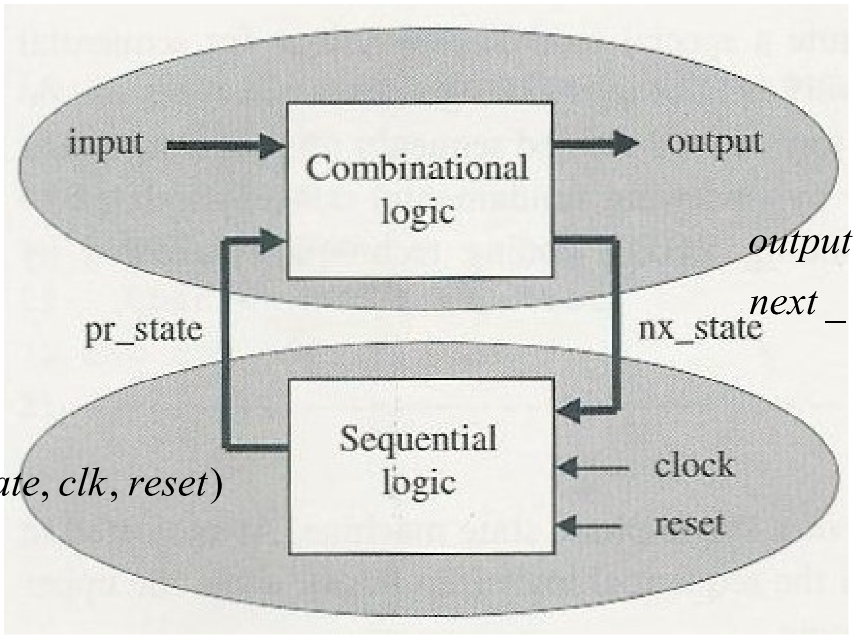
$next\_state = \quad ent\_state, input)$

  - output depends on both the present state and the current input
- Moore machine

$output = \quad ent\_state)$

$next\_state = \quad ent\_state, input)$

  - output depends only on the present state



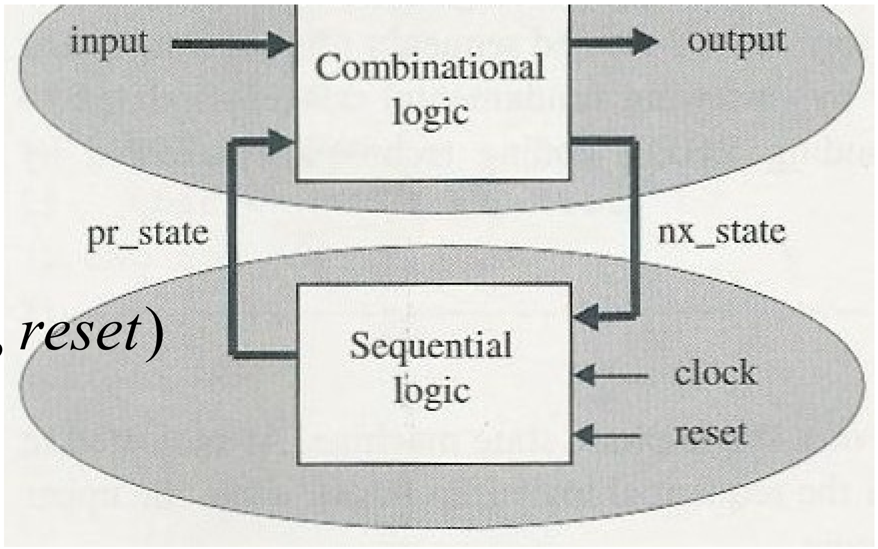
$output = \quad ent\_state,\ input)$

$next\_state = \quad ent\_state, input)$

$present\_state = \quad \_state, clk, reset)$

lower (sequential) section: generate “present state”

$$present\_state = \qquad \_state, clk, reset)$$



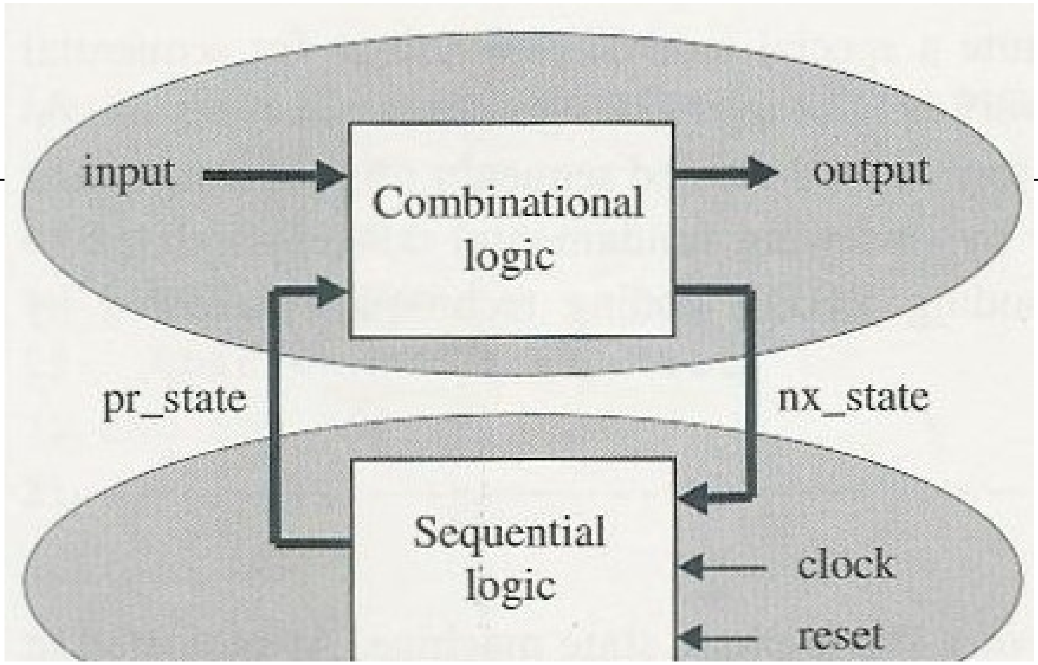
```
PROCESS (reset, clock)
BEGIN
  IF (reset = ‘1’) THEN
    pr_state <= state0;              -- initialization of pr_state
  ELSIF (clock’EVENT AND clock=‘1’) THEN
    pr_state <= nx_state;            -- infer registers to store present state
  END IF;
END PROCESS;
```

upper (combinational) section: generate “output” and “next state”

$output =$ $ent\_state,\ input)$

$next\_state =$ $ent\_state, input)$

```
PROCESS (input, pr_state);
BEGIN
CASE pr_state IS   -- infer pure combinational logic to generate output and next state
WHEN state0 =>
   IF (input = … ) THEN output <= <value>; nx_state <= state1; ELSE … END IF;
WHEN state1 =>
   IF (input = … ) THEN output <= <value>; nx_state <= state2; ELSE … END IF;
WHEN state2 =>
   IF (input = … ) THEN output <= <value>; nx_state <= state3; ELSE … END IF;
…
END CASE;
END PROCESS;
```



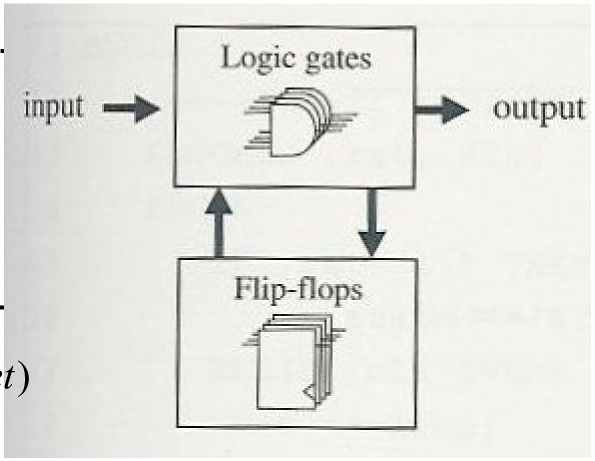
complete state machine template

```
ENTITY <entity_name> IS
PORT (input: IN <data_type>; reset, clock: IN_STD_LOGIC; output: OUT <data_type>;
END <entity_name>;
------------------------------------------------------------------------------------------
ARCHITECTURE <arch_name> OF <entity_name> IS
TYPE state IS (state0, state1, state2, …);
SIGNAL pr_state, nx_state: state;
BEGIN
----------------------- lower section (generate present state) --------
PROCESS (reset, clock)
BEGIN
   IF (reset = '1') THEN  pr_state <= state0;
   ELSIF (clock'EVENT AND clock='1') THEN pr_state <= nx_state;
   END IF;
END PROCESS;
---------------------------- upper section (generate output and next state ------------------------------
PROCESS (input, pr_state);
BEGIN
CASE pr_state IS
WHEN state0 => IF (input = … ) THEN output <= <value>; nx_state <= state1; ELSE … END IF;
WHEN state1 => IF (input = … ) THEN output <= <value>; nx_state <= state2; ELSE … END IF;
…
END CASE;
END PROCESS;
```

$present\_state = \quad \_state, clk, reset)$

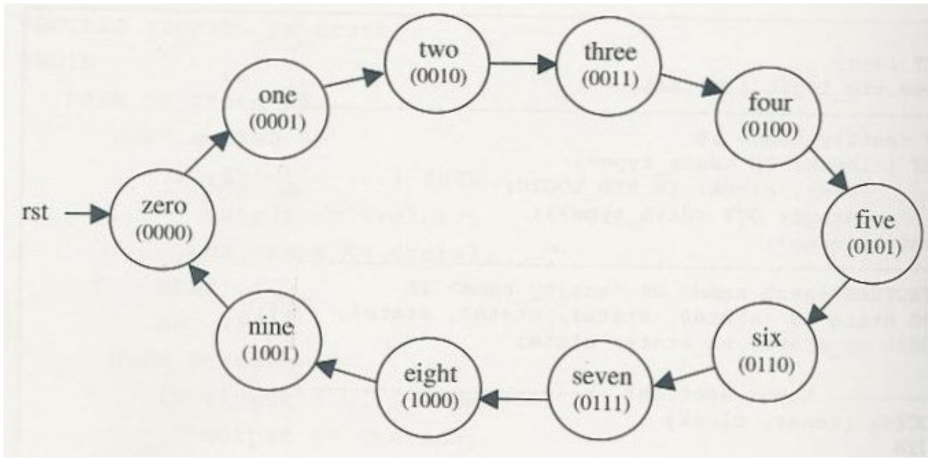
$output = \quad ent\_state, \ input)$

$next\_state = \quad ent\_state, input)$



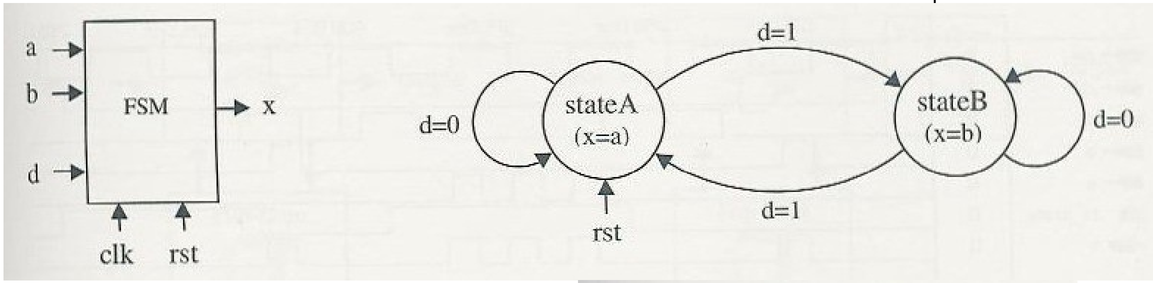
example: BCD counter

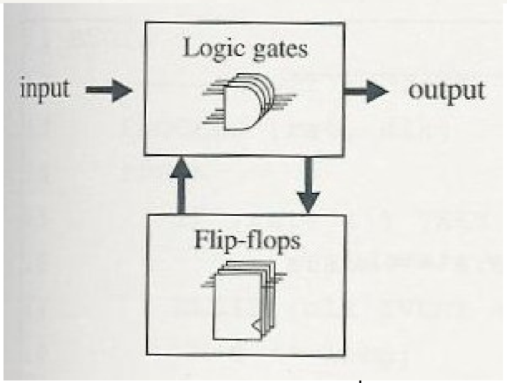
```
ENTITY counter IS
PORT (clk, rst: IN STD_LOGIC;     count: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
END counter;
----------------------------------------------
ARCHITECTURE state_machine OF counter IS
TYPE state IS (zero, one, two, three, four,  five, six, seven, eight, nine);
SIGNAL pr_state, nx_state: state;
BEGIN
------------- Lower section: generate present state----------------
PROCESS (rst, clk)
BEGIN
IF (rst='1') THEN pr_state <= zero; ELSIF (clk'EVENT AND clk='1') THEN pr_state <= nx_state; END IF;
END PROCESS;
------------- Upper section: generate output and next state ----------------
PROCESS (pr_state)
BEGIN
CASE pr_state IS
WHEN zero => count <= "0000"; nx_state <= one;
WHEN one => count <= "0001"; nx_state <= two;
WHEN two => count <= "0010"; nx_state <= three;
WHEN three => count <= "0011"; nx_state <= four;
WHEN four => count <= "0100"; nx_state <= five;
WHEN five => count <= "0101"; nx_state <= six;
WHEN six => count <= "0110"; nx_state <= seven;
WHEN seven => count <= "0111"; nx_state <= eight;
WHEN eight => count <= "1000"; nx_state <= nine;
WHEN nine => count <= "1001";nx_state <= zero;
END CASE;
END PROCESS;
END state_machine;
```



example: FSM #1 (swap state if d=1) (output is not stored)

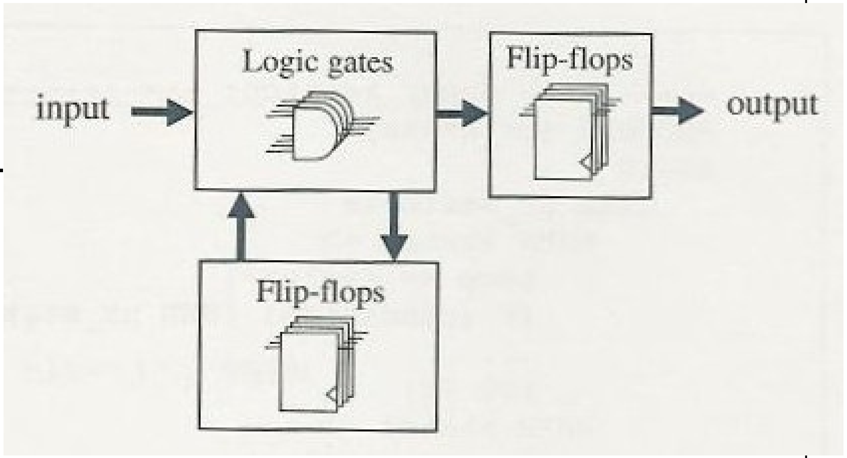
```
ENTITY simple_fsm IS PORT (a, b, d, clk, rst: IN BIT; x: OUT BIT);
END simple_fsm;
---------------------------------------------
ARCHITECTURE simple_fsm OF simple_fsm IS
TYPE state IS (stateA, stateB);
SIGNAL pr_state, nx_state: state;
BEGIN
----- Lower section: ----------------------
PROCESS (rst, clk)
BEGIN
IF (rst='1') THEN pr_state <= stateA;
ELSIF (clk'EVENT AND clk='1') THEN
            pr_state <= nx_state;
END IF;
END PROCESS;
---------- Upper section: ----------------
PROCESS (a, b, d, pr_state)
BEGIN
CASE pr_state IS
WHEN stateA => x <= a; IF (d='1') THEN nx_state <= stateB; ELSE nx_state <= stateA; END IF;
WHEN stateB => x <= b; IF (d='1') THEN nx_state <= stateA; ELSE nx_state <= stateB; END IF;
END CASE;
END PROCESS;
END simple_fsm;
```





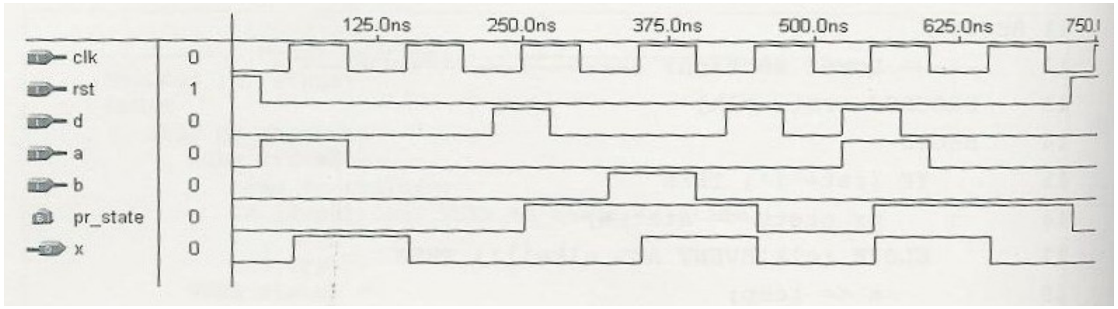
FSM Template with Stored Output

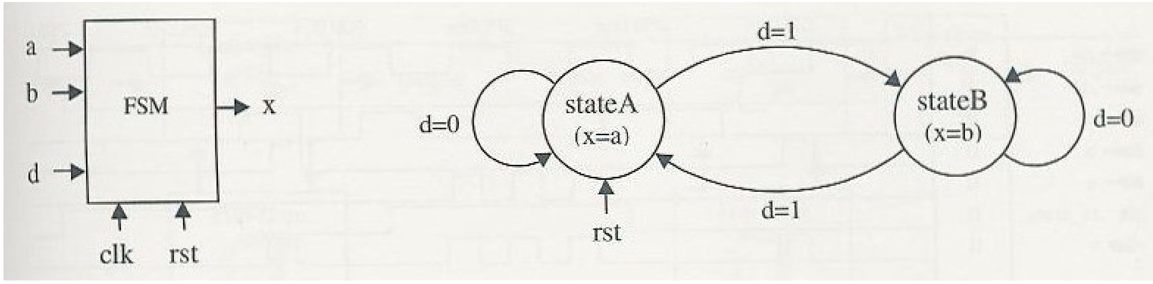
```
ENTITY <entity_name> IS
PORT (input: IN <data_type>; reset, clock: IN_STD_LOGIC; output: OUT <data_type>;
END <entity_name>;
-----------------------------------------------------------------------------------------------------------
ARCHITECTURE <arch_name> OF <entity_name> IS
TYPE state IS (state0, state1, state2, …);
SIGNAL pr_state, nx_state: state;
SIGNAL temp: <data_type>;
BEGIN-
---- lower section: generate pr_state  and store output-----
PROCESS (reset, clock)
BEGIN
   IF (reset = '1') THEN  pr_state <= state0;
   ELSIF (clock'EVENT AND clock='1') THEN
              output <= temp;
              pr_state <= nx_state;
   END IF;
END PROCESS;
------------------------------ upper section: generate output and next state -------------------------------------------
PROCESS (input, pr_state);
BEGIN
CASE pr_state IS
WHEN state0 => temp <= <value>; IF (condition ) THEN nx_state <= state1; … END IF;
WHEN state1 => temp <= <value>; IF (condition ) THEN nx_state <= state2; … END IF;
…
END PROCESS;
```



Example: simple FSM #2 (stored output)

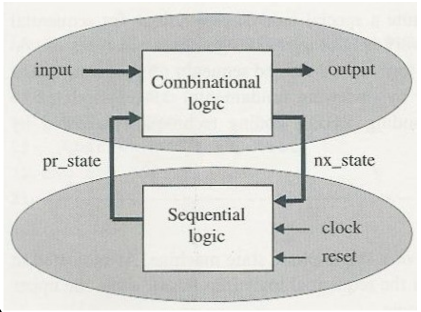
```
ENTITY simple_fsm IS PORT (a, b, d, clk, rst: IN BIT; x: OUT BIT); END simple_fsm;
----------------------------------------------
ARCHITECTURE simple_fsm OF simple_fsm IS
TYPE state IS (stateA, stateB);
SIGNAL pr_state, nx_state: state;
SIGNAL temp: BIT;
BEGIN
----- Lower section: ----------------------
PROCESS (rst, clk)
BEGIN
IF (rst='1') THEN pr_state <= stateA;
ELSIF (clk'EVENT AND clk='1') THEN
x <= temp; pr_state <= nx_state;
END IF;
END PROCESS;
---------- Upper section: ----------------
PROCESS (a, b, d, pr_state)
BEGIN
CASE pr_state IS
WHEN stateA => temp <= a; IF (d='1') THEN nx_state <= stateB;ELSE nx_state <= stateA;END IF;
WHEN stateB => temp <= b; IF (d='1') THEN nx_state <= stateA;ELSE nx_state <= stateB;END IF;
END CASE;
END PROCESS;
END simple_fsm;
```

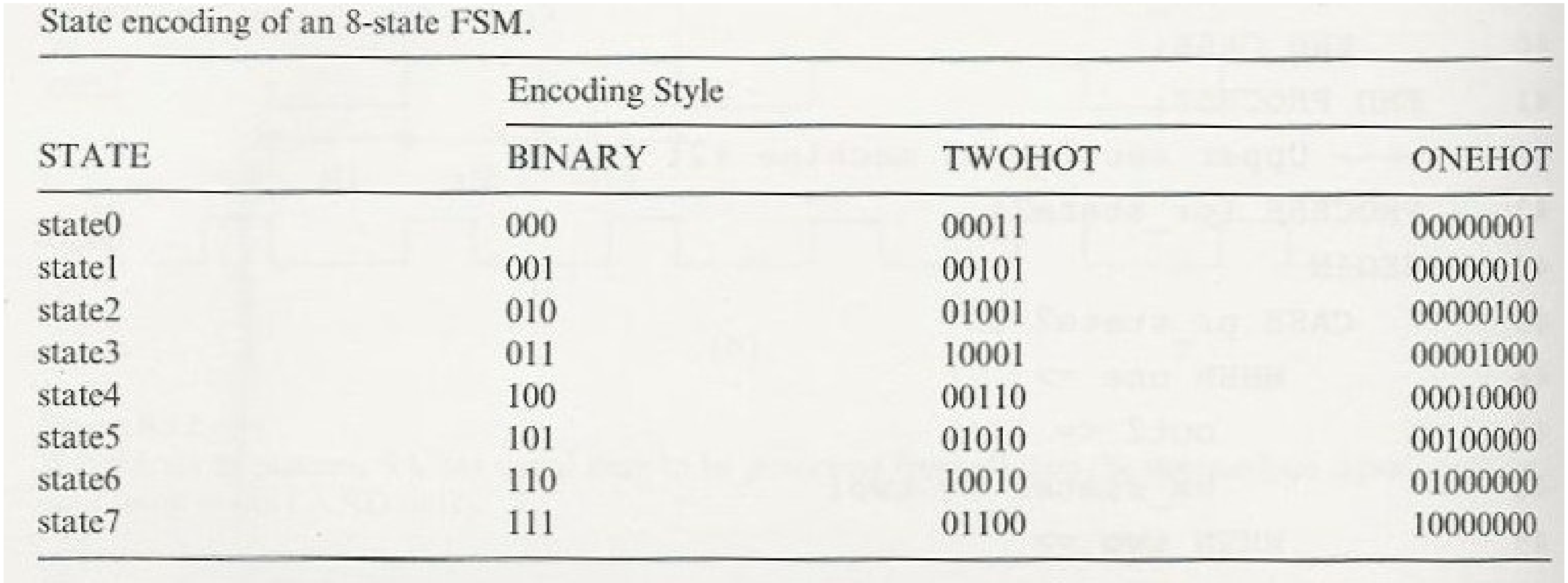




State encoding style (binary or one-hot)

- Binary state encoding
  - Requires least number of flip-flops (smaller lower part)
  - Needs more area in the upper part, and is slower
    - due to state decoding
  - In ASIC applications where total area is main goal
- One-hot state encoding
  - Requires largest number of flip-flops (larger lower part)
  - Needs least amount of logic area in the upper part and is fast
  - In FPGA applications (where FFs are abundant)

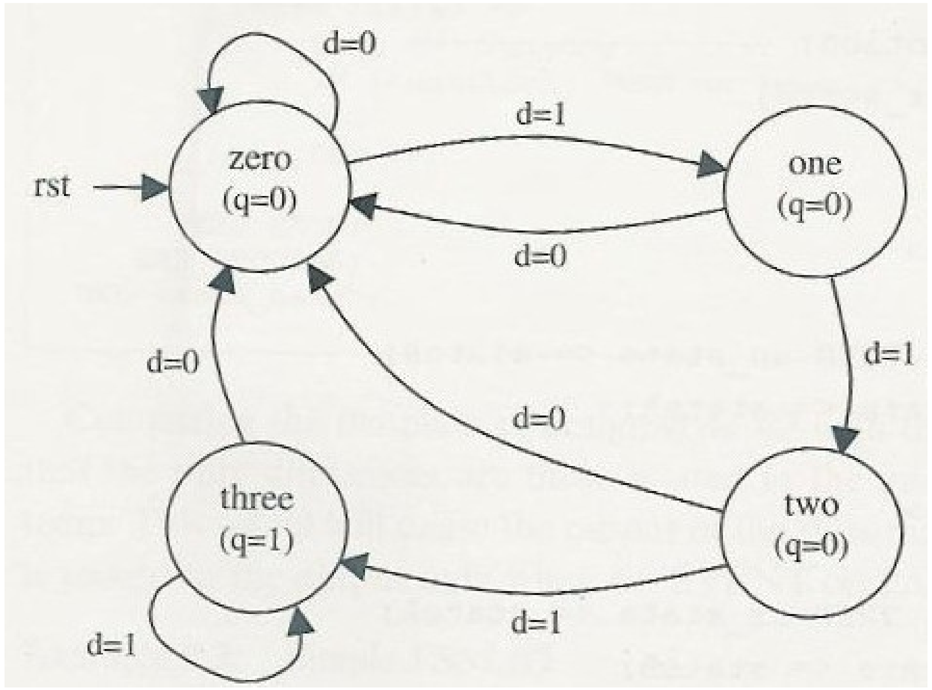



State encoding of an 8-state FSM.

	Encoding Style		
STATE	BINARY	TWOHOT	ONEHOT
state0	000	00011	00000001
state1	001	00101	00000010
state2	010	01001	00000100
state3	011	10001	00001000
state4	100	00110	00010000
state5	101	01010	00100000
state6	110	10010	01000000
state7	111	01100	10000000

Example : string detector (e.g., “111”)

```
-- detect a string sequence of “111”
ENTITY string_detector IS PORT (d, clk, rst: IN BIT; q: OUT BIT); END string_detector;
--------------------------------------------
ARCHITECTURE my_arch OF string_detector IS
TYPE state IS (zero, one, two, three);
SIGNAL pr_state, nx_state: state;
BEGIN
----- Lower section: --------------------
PROCESS (rst, clk) BEGIN
IF (rst='1') THEN
        pr_state <= zero;
ELSIF (clk'EVENT AND clk='1') THEN
        pr_state <= nx_state;
END IF;
END PROCESS;
---------- Upper section: ---------------
PROCESS (d, pr_state) BEGIN
CASE pr_state IS
WHEN zero => q <= ‘0’;IF (d=‘1’) THEN nx_state <= one; ELSE nx_state <= zero; END IF;
WHEN one => q <= ‘0’; IF (d=‘1’) THEN nx_state <= two; ELSE nx_state <= zero; END IF;
WHEN two => q <= ‘0’; IF (d=‘1’) THEN nx_state <= three; ELSE nx_state <= zero; END IF;
WHEN three => q <= ‘1’; IF (d=’0’) THEN nx_state <= zero; ELSE nx_state <= three; END IF;
END CASE;
END PROCESS;
END my_arch;
```

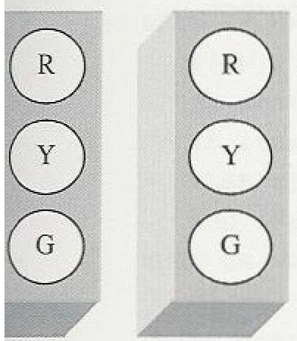


string detector for “101”

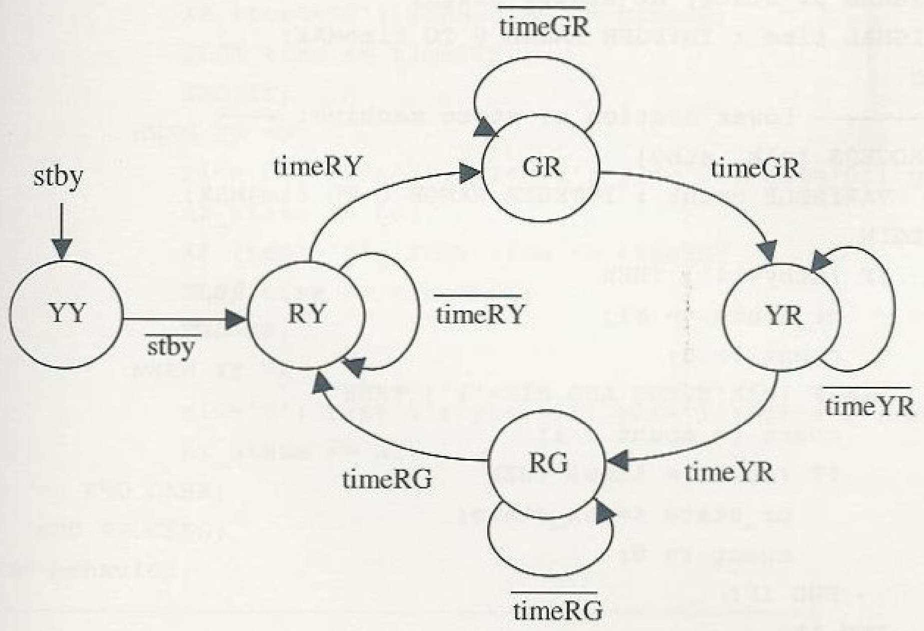
- quiz: write a code to detect string “101”
  - first draw the state diagram

Example: traffic light controller

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY tlc IS
PORT (clk, stby, test: IN STD_LOGIC;
r1, r2, y1, y2, g1, g2: OUT STD_LOGIC);
END tlc;
---------------------------------------------
ARCHITECTURE behavior OF tlc IS
CONSTANT timeMAX : INTEGER := 2700;
CONSTANT timeRG : INTEGER := 1800;
CONSTANT timeRY : INTEGER := 300;
CONSTANT timeGR : INTEGER := 2700;
CONSTANT timeYR : INTEGER := 300;
CONSTANT timeTEST : INTEGER := 60;
TYPE state IS (RG, RY, GR, YR, YY);
SIGNAL pr_state, nx_state: state;
SIGNAL time : INTEGER RANGE 0 TO timeMAX;
BEGIN
```



State	Operation Mode		
	REGULAR	TEST	STANDBY
	Time	Time	Time
RG	timeRG (30s)	timeTEST (1s)	---
RY	timeRY (5s)	timeTEST (1s)	---
GR	timeGR (45s)	timeTEST (1s)	---
YR	timeYR (5s)	timeTEST (1s)	---
YY	---	---	Indefinite

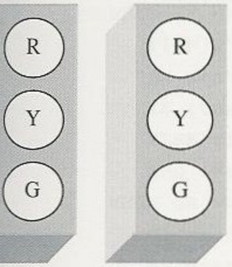

$\overline{\text{timeGR}}$
GR
timeRY
timeGR
stby
YY
$\overline{\text{stby}}$
RY
$\overline{\text{timeRY}}$
YR
$\overline{\text{timeYR}}$
RG
timeRG
timeYR
$\overline{\text{timeRG}}$

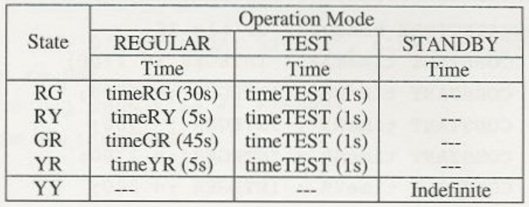
Example: traffic light controller (cont.)

```
-------- Lower section of state machine: ----
PROCESS (clk, stby)
VARIABLE count : INTEGER RANGE 0 TO timeMAX;
BEGIN
IF (stby='1') THEN pr_state <= YY; count := 0;
ELSIF (clk'EVENT AND clk='1') THEN
count := count + 1; IF (count = time) THEN pr_state <= nx_state; count := 0; END IF;
END IF;
END PROCESS;
-------- Upper section of state machine: ----
PROCESS (pr_state, test) BEGIN CASE pr_state IS
WHEN RG => r1<='1'; r2<='0'; y1<='0'; y2<='0'; g1<='0'; g2<='1';nx_state <= RY;
        IF (test='0') THEN time <= timeRG; ELSE time <= timeTEST; END IF;
WHEN RY => r1<='1'; r2<='0'; y1<='0'; y2<='1'; g1<='0'; g2<='0'; nx_state <= GR;
        IF (test='0') THEN time <= timeRY; ELSE time <= timeTEST; END IF;
WHEN GR => r1<='0'; r2<='1'; y1<='0'; y2<='0'; g1<='1'; g2<='0'; nx_state <= YR;
        IF (test='0') THEN time <= timeGR; ELSE time <= timeTEST; END IF;
WHEN YR => r1<='0'; r2<='1'; y1<='1'; y2<='0'; g1<='0'; g2<='0'; nx_state <= RG;
        IF (test='0') THEN time <= timeYR; ELSE time <= timeTEST; END IF;
WHEN YY => r1<='0'; r2<='0'; y1<='1'; y2<='1'; g1<='0'; g2<='0'; nx_state <= RY;
END CASE; END PROCESS;
END behavior;
```

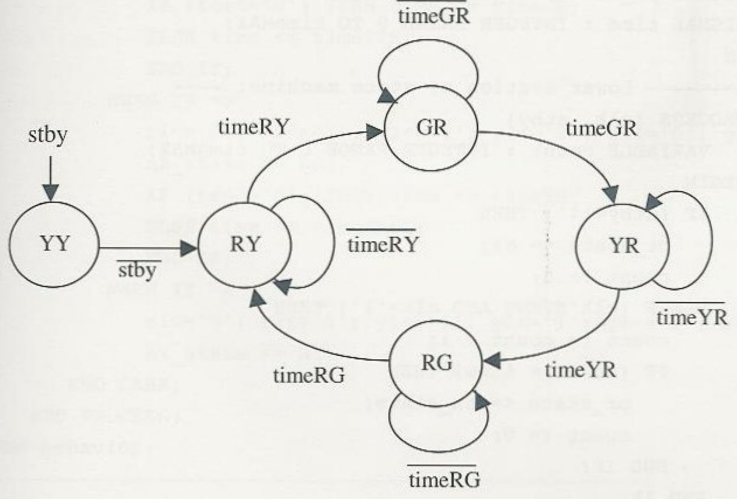
simulation results

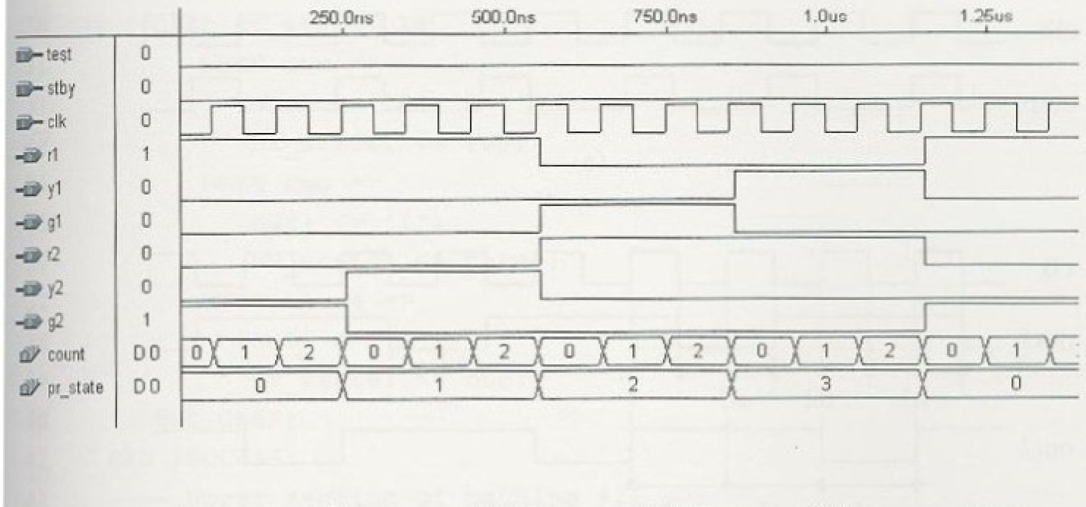
- 15 FFs synthesized
  - 3 for pr_state
  - 12 for count

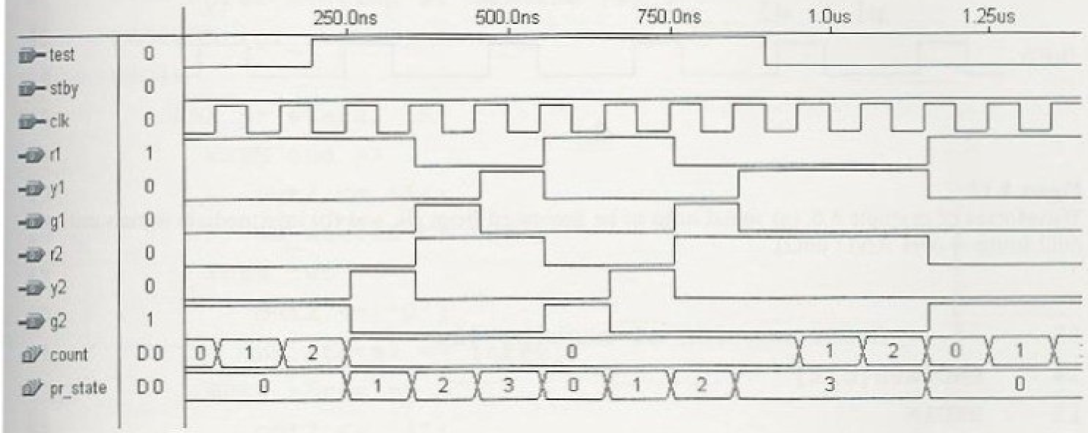


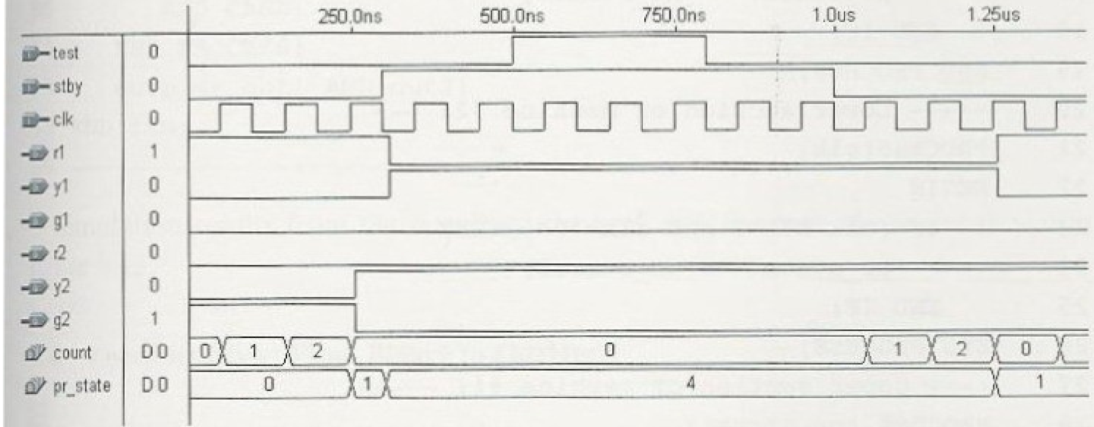


State	Operation Mode		
	REGULAR	TEST	STANDBY
	Time	Time	Time
RG	timeRG (30s)	timeTEST (1s)	---
RY	timeRY (5s)	timeTEST (1s)	---
GR	timeGR (45s)	timeTEST (1s)	---
YR	timeYR (5s)	timeTEST (1s)	---
YY	---	---	Indefinite


$\overline{\text{timeGR}}$
stby
timeRY
GR
timeGR
YY
$\overline{\text{stby}}$
RY
$\overline{\text{timeRY}}$
YR
$\overline{\text{timeYR}}$
RG
timeRG
timeYR
$\overline{\text{timeRG}}$





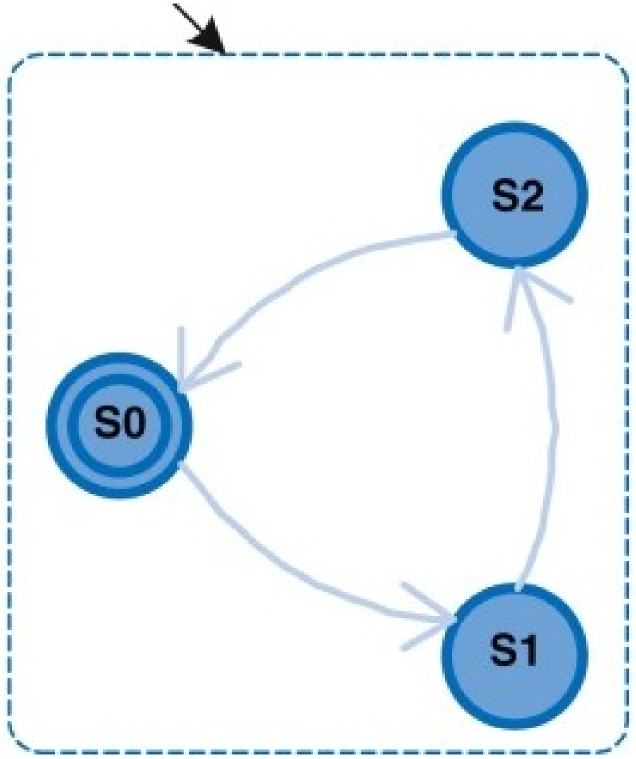


Divide-by-3 counter with 1/3 duty cycle (VHDL)

```
-- output 100_100_100 …
process (clk, reset) begin
if (reset = '1') then state <= S0;
elsif (clk'event and clk = '1') then
      state <= nextstate; end if;
end process;

process (all) begin
    case state is
      when S0 =>  nextstate <= S1;
      when S1 =>  nextstate <= S2;
       when S2 =>  nextstate <= S0;
end process;

y <= '1' when state = S0 else
   <= '0'; ;
```



Divide-by-3 counter (Verilog)

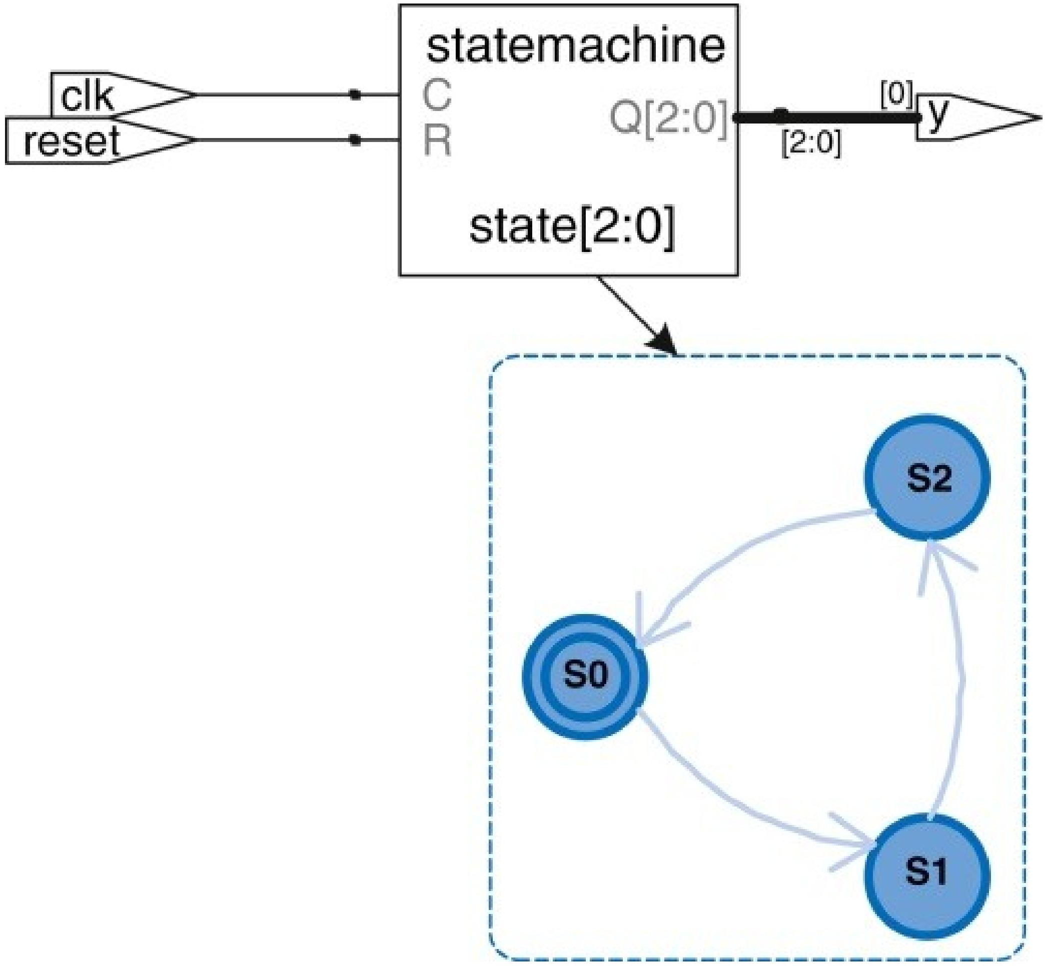
```
module divideby3FSM  (input  clk,  input  reset,  output  y);
reg [1:0] state, nextstate;
parameter S0 = 2'b00; //binary encoding
parameter S1 = 2'b01;
parameter S2 = 2'b10;

  // state register
  always @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always @ (*)
    case (state)
      S0:     nextstate = S1;
      S1:     nextstate = S2;
      S2:     nextstate = S0;
      default: nextstate = S0;
    endcase

  // output logic
  assign y = (state == S0); // duty cycle = 1/3 = 33%)

endmodule
```



freqiemcu divider with ~1/2 duty cycle

- the above divide-by-3 clock generator
  - generate 100_100_100_...
  - 3x period with duty cucle 1/3
- hwo to divide-by-5 freq divider with duty cycle = 3/5 ?

divide-by-5 clk with ~1/2 duty cycle (Verilog)

- count the accumulated number of clk ecges
  - clk_ou=1 if count = 5/2 = 2,
  - reset count and clk_out if count = M = 5
- wo methods

```
paratmer M=5;
reg clk_out;
reg. [3:0] counter;
always @ (posedge clk) begin
counter <= counter+1;
if (counter == M/2)
     clk_out <= 1;
else if (counter == M) begin
          clk_out <= 0;
          counter <= 0; end
end
end
```

```
always @ (posedge clk)
   state <= next+state;

always @ (*) begin
case state
s0: begin next_state = s1; clk_out =0; end
s1: begin next_state = s2; clk_out =0; end
s2: begin next_state = s3; clk_out =1; end
s3: begin next_state = s4; clk_out =1; end
s4: begin next_state = s0; clk_out =1; end
endcase
end
```

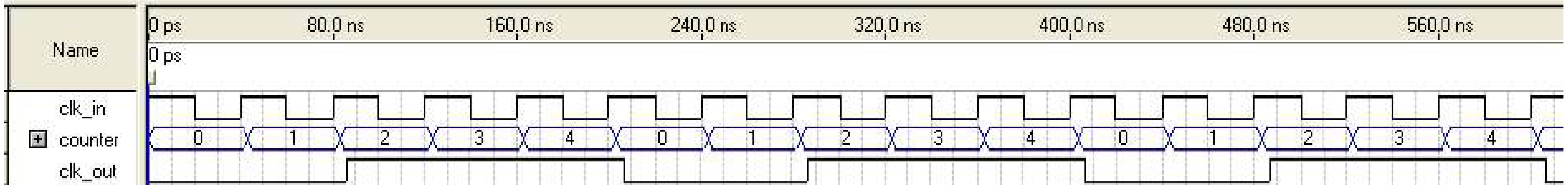
frequency divider (VHDL)

- clk_out: 00111_00111_00111 … (5x clk period)

```
---------------------------------------------
ENTITY clock_divider IS
   GENERIC (M: NATURAL := 5;
   PORT (clk_in: IN BIT;
         clk_out: OUT BIT);
END ENTITY;
---------------------------------------------
ARCHITECTURE circuit OF clock_divider IS
BEGIN
   PROCESS (clk_in)
      VARIABLE counter: NATURAL RANGE 0 TO M;
   BEGIN
      IF clk_in'EVENT AND clk_in='1' THEN
         counter := counter + 1;
         IF counter=M/2 THEN
            clk_out <= '1';
         ELSIF counter=M THEN
            clk_out <= '0';
            counter := 0;
         END IF;
      END IF;
   END PROCESS;
END ARCHITECTURE;
---------------------------------------------
```

```
-- M=5
process (clk) begin
variable counter: natural range 0 to M;
if (clk’event and clck=‘1’ then
  counter:=counter+1;
  if (counter = M/2 then
    clk_out <= ‘1’;
  elsif (counter = M) then
    clk_out <= ‘0’;
    counter := 0;
  end if;
end process;
```

Simulation results for *M*=5:



divide-by-5 counter (using FSM)

- design a divide-by-5 counter (frequency divider) with 3/5 duty cycle using finite state machine with one-hot state encoding
  - 5 states

  s1=“00001”=> next_state<=“00010”; clk_out<=‘0’;

  s2=“00010”=> next_state<=“”00100”; clk_out<=‘0’;

  s3=“00100”=> next_state<=“01000”; clk_out<=‘1’;

  s4=“01000”=> next_state<=“10000”; clk_out<=‘1’;

  s5=“10000” next_state<=“00001”; clk_out<=‘1’;

typical FSM applications

- controller
  - generate control signals for the entire system
- vendor machine
- traffic light controller
- 7-segment LED controller
- string detector (string matching)
- divide-by-n counter (frequency divider)