Lecture 5

Prepared by: Dr. Alireza Kavianpour

VHDL Architecture for 2-Bit Gray Code Counter

```
ARCHITECTURE circuit OF gray2bit IS
BEGIN
PROCESS (CLK, CLR)
BEGIN
IF (CLR = ‘0’) THEN QOUT <= 0;
ELSIF (CLK’EVENT AND CLK = ‘1’) THEN
IF QOUT = 0 THEN QOUT <= 1;
ELSIF QOUT = 1 THEN QOUT <= 3;
ELSIF QOUT = 3 THEN QOUT <= 2;
ELSIF QOUT = 2 THEN QOUT <= 0;
END IF;
END IF;
END PROCESS;
END circuit;
```

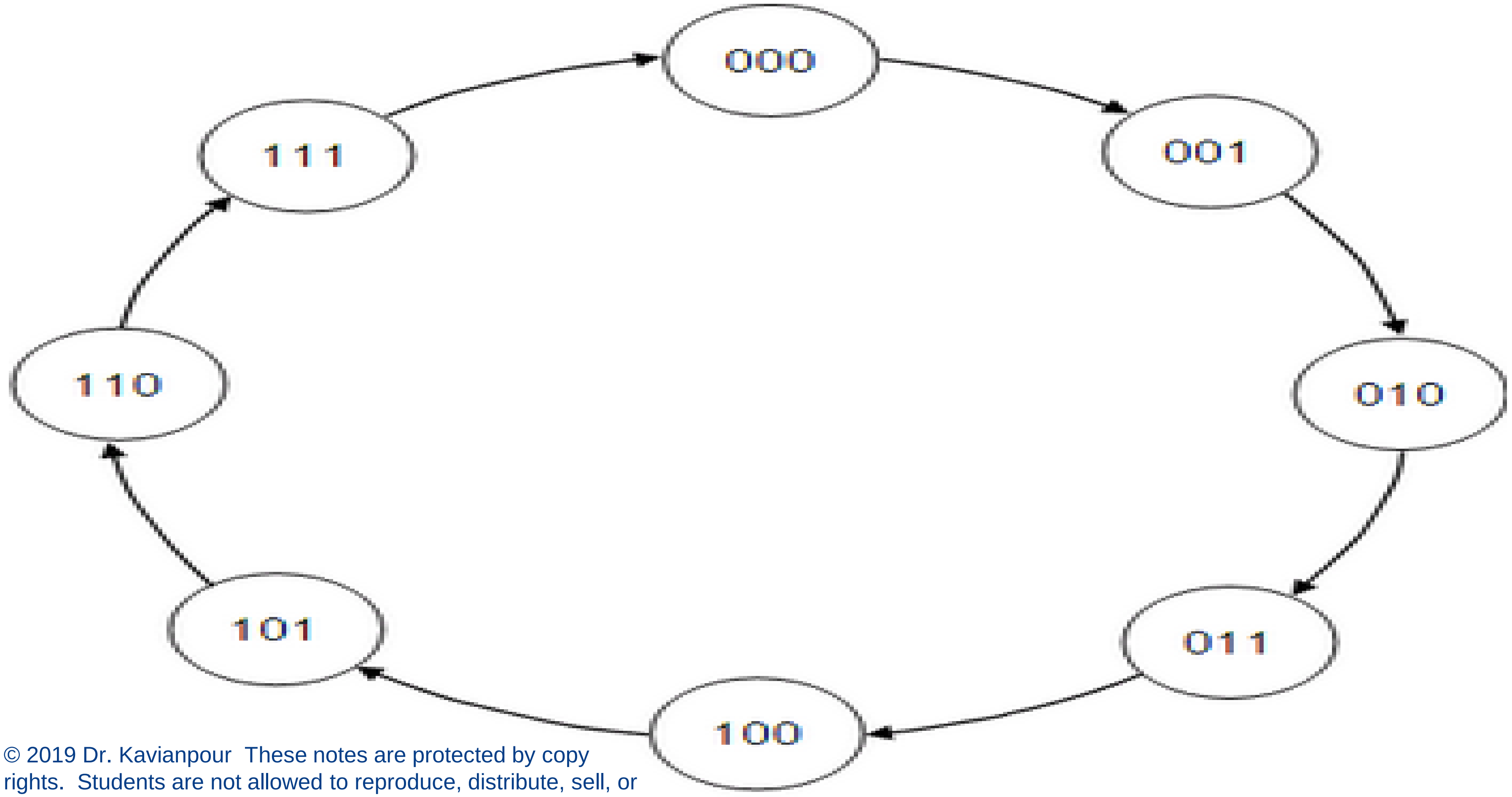
VHDL Architecture for 3-Bit Gray Code Counter

```
ARCHITECTURE circuit OF gray3bit IS
BEGIN
PROCESS (CLK, CLR)
BEGIN
IF (CLR = ‘0’) THEN QOUT <= 0;
ELSIF (CLK’EVENT AND CLK = ‘1’) THEN
IF QOUT = 0 THEN QOUT <= 1;
ELSIF QOUT = 1 THEN QOUT <= 3;
ELSIF QOUT = 3 THEN QOUT <= 2;
ELSIF QOUT = 2 THEN QOUT <= 6;
ELSIF QOUT =6 THEN QOUT <= 7;
ELSIF QOUT = 7 THEN QOUT <= 5;
ELSIF QOUT = 5 THEN QOUT <= 4;
ELSIF QOUT = 4 THEN QOUT <= 0;
END IF;
END IF;
END PROCESS;
END circuit;
```

State diagrams

- State diagrams are used to graphically indicate the operation of a logic circuit.
- State diagrams are useful for designing logic circuits in which events occur in a fixed order.
- Circuits having this characteristic are known as finite state machines (FSMs).
- Counters are a special case of FSMs and are well-suited to be described using state diagrams.
- A state diagram for a counter shows how the counts, or states, are related. Figure below is an example of a 3-bit binary up-counter.

State Diagrams for Counters

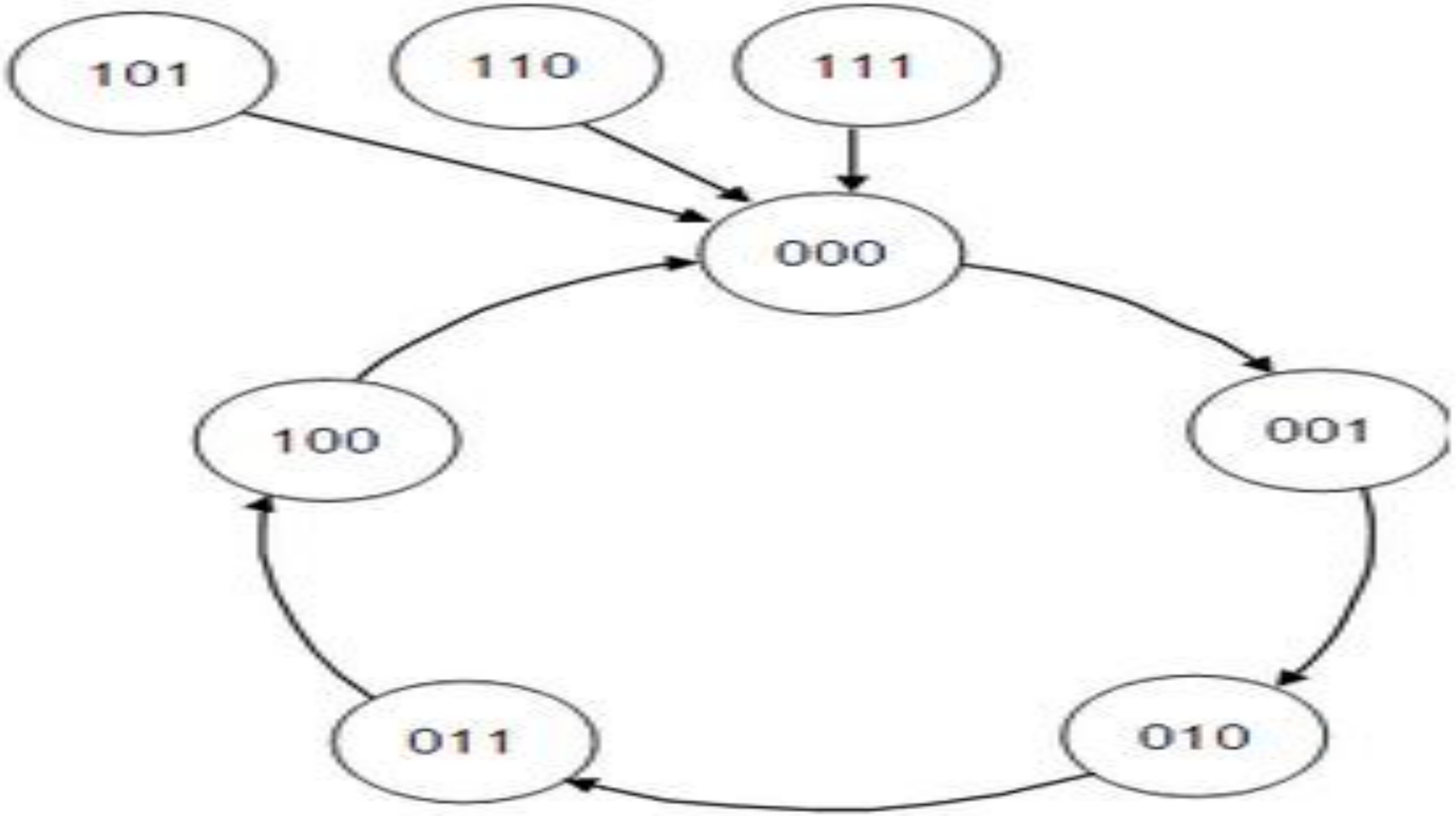


$$\begin{tikzcd}
 & 111 \arrow[r] & 000 \arrow[r] & 001 \arrow[d] \\
110 \arrow[ur] & & & 010 \arrow[d] \\
 & 101 \arrow[ul] & 100 \arrow[l] & 011 \arrow[l]
\end{tikzcd}$$

Truncated Binary Counters

- There are times when a non-2^N modulus is desired.
- Consider a simple case of needing a MOD 5 counter (note that if we use binary up-counters, the count typically ranges between 0 and N-1). Figure below shows the state diagram for this case.

MOD 5 Binary Up-Counter State Diagram

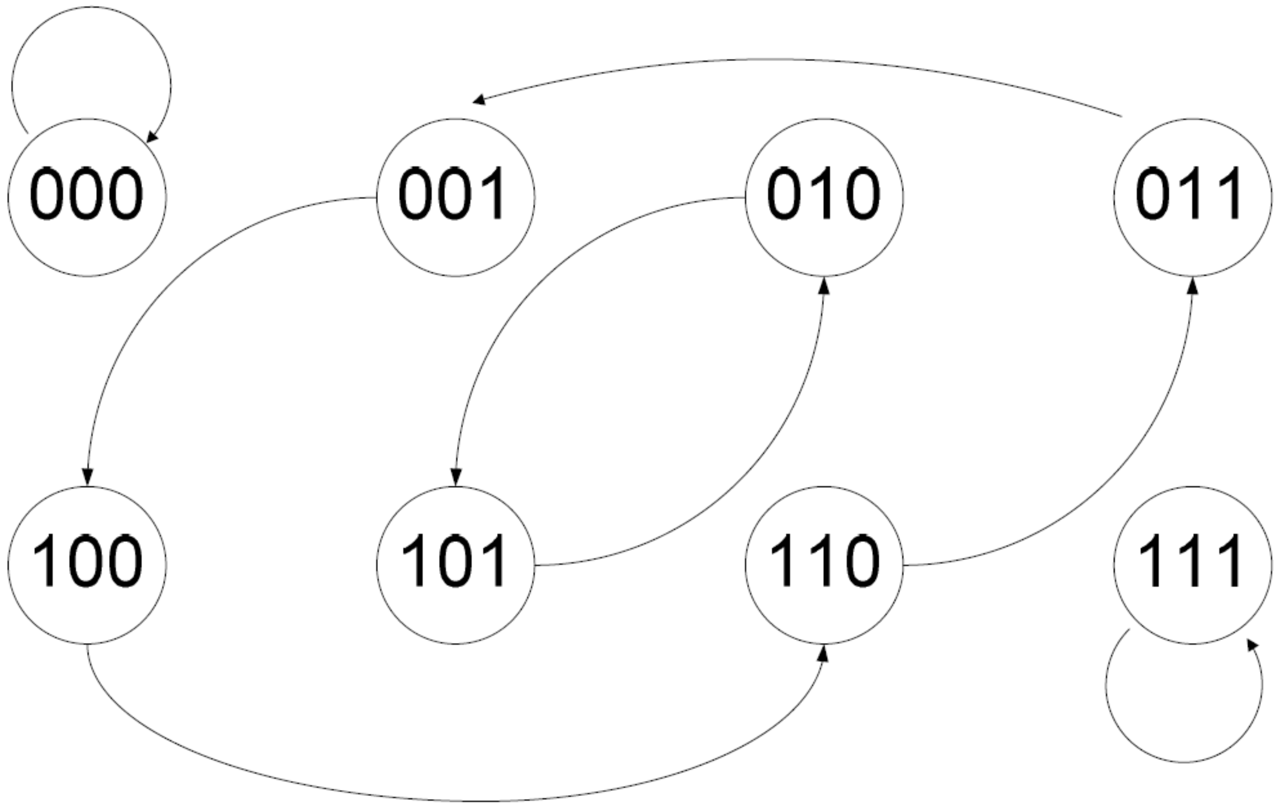


Design of Sequential Circuit

- 1-State Diagram
- 2-State table
- 3-Transition Table
- 4-Excitation table
- 5-Boolean expression

Design of Sequential Circuit

Given the following state diagram:



Design of Sequential Circuit

(1) Construct the next-state table.

Q2	Q1	Q0	Q2(next)	Q1(next)	Q0(next)
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	1	1
1	1	1	1	1	1

Design of Sequential Circuit

(2) Derive the next state expressions

$$Q_2(next) = Q_2'Q_1'Q_0 + Q_2'Q_1Q_0' + Q_2Q_1Q_0 + Q_2Q_1'Q_0'$$

$$= Q_2'(Q_1 \oplus Q_0) + Q_2(Q_1 \oplus Q_0)'$$

$$= Q_2 \oplus Q_1 \oplus Q_0$$

or

$$Q_2(next) = Q_2 XNOR Q_1 XNOR Q_0$$

or

$$Q_2(next) = Q_2' XNOR (Q_1 \oplus Q_0)$$

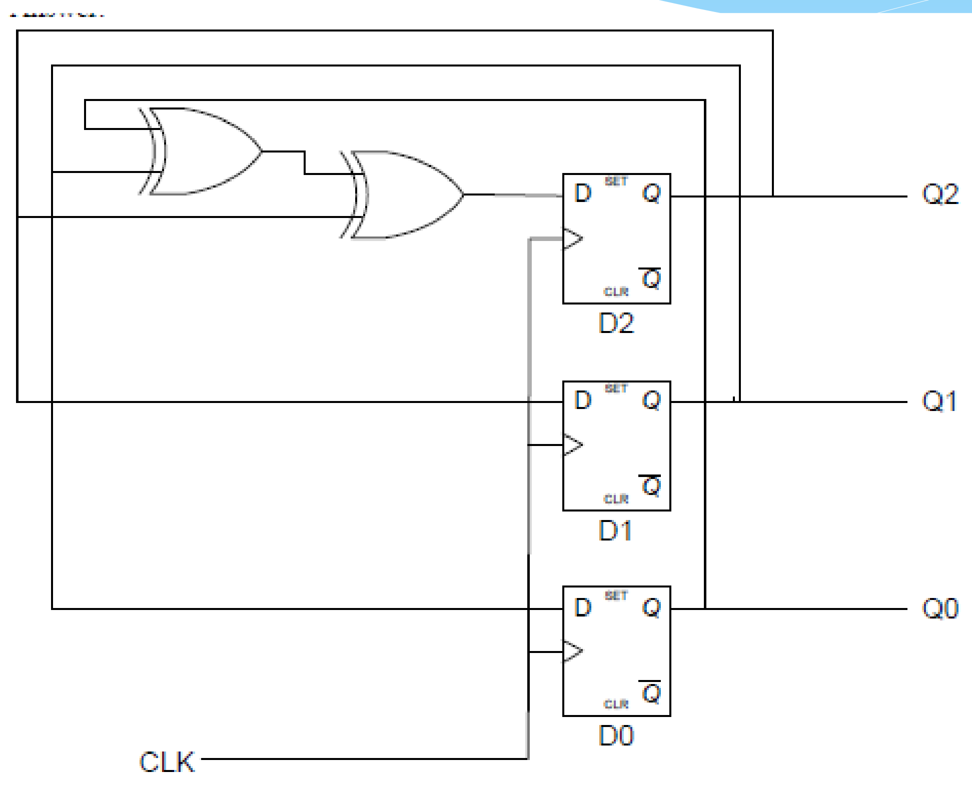
Design of Sequential Circuit

$$Q_1(next) = Q_2$$

$$Q_0(next) = Q_1$$

(3) Based on the next state table you have derived, construct a sequential circuit using only three D flip-flops and two 2-input gates (choose from AND, OR, NAND, NOR, XOR, XNOR).

Design of Sequential Circuit



Excitation for JK Flip Flops

- JK Flip Flop

Q→ Qnext	JK
0--→0	0-
0--→1	1-
1--→0	-1
1--→1	-0

Excitation for SR Flip Flops

- SR Flip Flop

Q→ Qnext	SR
0→0	0-
0→1	10
1→0	01
1→1	-0

Excitation for T Flip Flops

* T Flip Flop

Q→ Qnext	T
0--→0	0
0--→1	1
1--→0	1
1--→1	0

Excitation for D Flip Flops

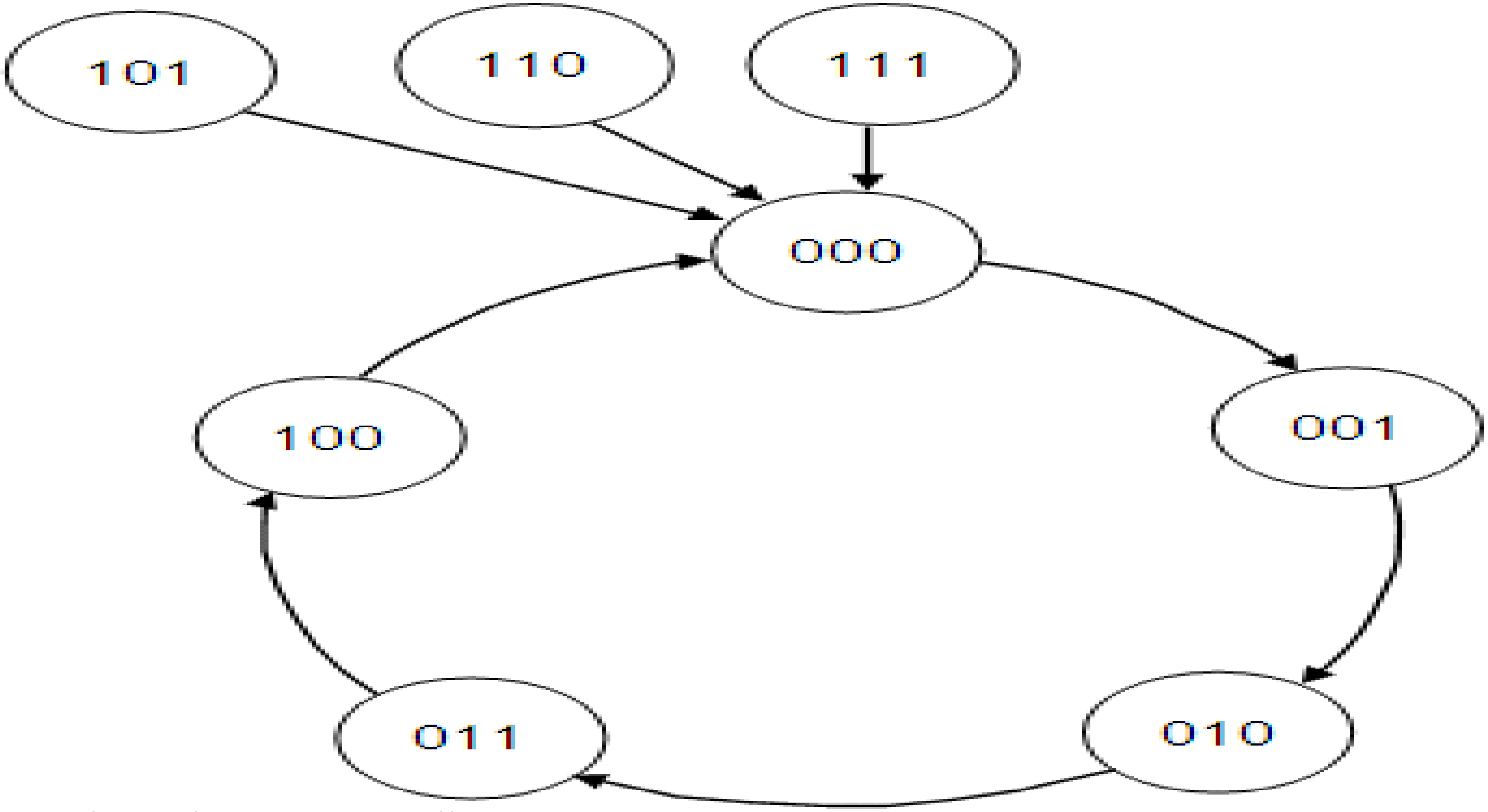
* D Flip Flop

Q→ Qnext	D
0--→0	0
0--→1	1
1--→0	0
1--→1	1

TYPE Statement State Diagram

- The TYPE statement defines a new signal type and the allowed values for that signal type. For example, we know that a BIT signal type has two values: 0 and 1. When we use the TYPE statement, we define the name of the new signal type and the values that type can have.
- For the MOD 5 counter, the definition of the TYPE statement would look like this:
- **ARCHITECTURE fsm1 OF counter5 IS**
  TYPE STATE_TYPE IS (S7,S6,S5,S4,S3,S2,S1,S0);
  SIGNAL state :STATE_TYPE;

Mod 5 counter



Mod 5 counter

- The TYPE statement is used to define the eight identified states for this counter (S7–S0). The SIGNAL declaration is necessary, because the values for the term “state” are not inputs to or outputs from the circuit. The STATE_TYPE notation identifies the form of the term “state”, which was specified in the TYPE statement. The combination of these two statements creates a SIGNAL named “state” with eight possible values, S7–S0.

Mod 5 counter

- **type state_type is (s0,s1,s2,s3,s4,s5,S5,s6,s7);**
- **signal state: state_type;**
- **begin**
- **PROCESS(CLK)**

```
BEGIN
    IF (CLK'EVENT AND CLK = '1') THEN
        CASE state IS
            WHEN s0 => state <= s1;
             ----------
```

- **End process**
- **WITH state SELECT**

```
Q <= "000" WHEN s0,
“001" WHEN s1,
```

  - **-------**

Loop Statement

- Loop statements are another method of creating a control structure within a **PROCESS**. In VHDL, loops are used to specify repeating sequences of behavior in a circuit.
- There are three types of loops available in VHDL: the **FOR** loop, the **WHILE** loop, and infinite loops. In this discussion, only the **FOR** loop is covered. The **FOR** loop has the following general format:
- **FOR identifier IN range LOOP**
  sequential statements_1
  sequential statements_2
  …
  sequential statements_n
  END LOOP;

Loop Statement

```
ARCHITECTURE behavior OF mux_4_1 is
BEGIN
PROCESS(S,I)
BEGIN
FOR n IN 0 to 3 LOOP
IF S = n THEN
Y <= I(n);
END IF;
END LOOP;
END PROCESS;
END behavior;
```

Use of Generate

```
entity rippleadder is
port (a, b : in unsigned(63downto 0);
cin : in std_logic;
sum : out unsigned(63downto 0);
cout : out std_logic);
end rippleadder;
architecture imp of rippleadder is
signal c : unsigned(64 downto 0);
begin
c(0) <= cin;
G1: for m in 0 to 63generate
sum(m) <= a(m) xor b(m) xor c(m);
c(m+1) <= (a(m) and b(m)) or (b(m) and c(m)) or (a(m) and c(m));
end generate G1;
cout <= c(64);
end imp;
```

Variables

- **Signals vs. Variables:**
- Variables can only be used inside processes,
- signals can be used inside or outside processes.
- Any variable that is created in one process cannot be used in another process, signals can be used in multiple processes *though they can only be assigned in a single process*.
- Variables need to be defined after the keyword *process* but before the keyword *begin*. Signals are defined in the architecture before the *begin* statement.
- Variables are assigned using the **:=** assignment symbol. Signals are assigned using the **<=** assignment symbol.
- Variables that are assigned immediately take the value of the assignment. Signals depend on if it's combinational or sequential code to know when the signal takes the value of the assignment.

Variables

```
signal a,b : std_logic_vector(0 to 4);

process (CLK)
    variable var : std_logic_vector(0 to 4);
    begin
        if (rising_edge(clk)) then
            var := '11111';
            a <= var;
            b <= var;
        end if;
end process;
```