



## L5: Simple Sequential Examples and VHDL



Acknowledgement: Nathan Ickes

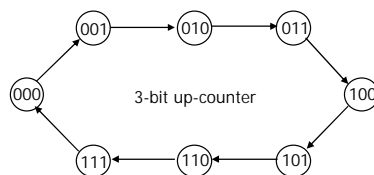
*Some slides are derived from slides used in past terms of 6.111*



## A Simple FSM: Counter



- **Counters**
  - Proceed through a well-defined state sequence
- **Many types of counters: binary, Gray-code**
  - 3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...
  - 3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...
- **Encoding of states: easy for counters – just use value**



	current state	next state
0	000	001   1
1	001	010   2
2	010	011   3
3	011	100   4
4	100	101   5
5	101	110   6
6	110	111   7
7	111	000   0



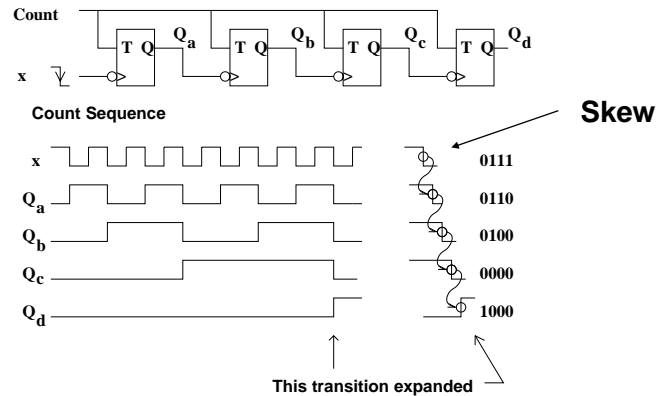
## Ripple Counter (Not Synchronous)



Simple counter – uses only registers

- Successive bits change on negative edge of the previous bit.
- Toggle rate fastest for the LSB

...but ripple architecture leads to large skew between outputs



Possible implementation problems

- Many programmable fabrics require all clock inputs to be driven by special global clock signals.

L5 6.111 Fall 2003 – Introductory Digital Systems Laboratory

3



## Structural Model for Ripple Counter



```

entity rip4 is
  port (clk, enb : in std_logic;
        count : out std_logic_vector(3 downto 0));
end rip4;
architecture structural of rip4 is
  component t_ff
    port (clk, t : in std_logic;
          q : out std_logic);
  end component;
  signal cntinside : std_logic_vector(3 downto 0);

begin
  count <= cntinside;
  bit_0 : t_ff
    port map (clk => clk, t => enb,
              q => cntinside(0));
  bit_1 : t_ff
    port map (clk => cntinside(0), t => enb,
              q => cntinside(1));
  bit_2 : t_ff
    port map (clk => cntinside(1), t => enb,
              q => cntinside(2));
  bit_3 : t_ff
    port map (clk => cntinside(2), t => enb,
              q => cntinside(3));
end structural;

```

Need to have a model for t\_ff in some file.

Why this?

Uses named association. Describes how flip-flops are wired together.

L5 6.111 Fall 2003 – Introductory Digital Systems Laboratory

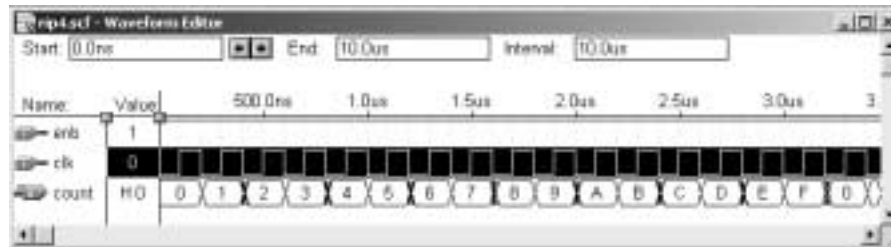
4



## Ripple Counter Simulation (MAX+plusII)



- Notice the ripple when a lot of flip-flops change.



## Modeling Sequential Logic using Process



```
[label:] process [(sensitivity_list)]
[declarations]
begin
{sequential_statement}
end process [label];
```

- A process is a wrapper for sequential statements.
  - Sequential statements model combinational and synchronous logic.
- When one of the signals in the *sensitivity list* changes, the sequential statements in the process are executed in sequence one time, scheduling signal changes.
- Signals are updated when end process is reached.
- Multiple processes are possible within an architecture. A process is concurrent with other concurrent statements.



## Concurrent vs. Sequential Statements



```

A <= B; -- 1
B <= C; -- 2
C <= D; -- 3

```

```

process (B, C, D)
begin
  A <= B; -- 1
  B <= C; -- 2
  C <= D; -- 3
end process;

```

T	δ	A	B	C	D	comments
0	+0	1	2	3	0	Initial state
1	+0	1	2	3	4	3 executes
1	+1	1	2	4	4	2 executes
1	+2	1	4	4	4	1 executes
1	+3	4	4	4	4	No further execution

T	δ	A	B	C	D	comments
0	+0	1	2	3	0	Initial state
1	+0	1	2	3	4	1,2,3 execute; update A,B,C one delta later
1	+1	2	3	4	4	1,2,3 execute; update A,B,C one delta later
1	+2	3	4	4	4	1,2,3 execute; update A,B,C one delta later
1	+3	4	4	4	4	No further execution

*During simulation, the scheduling and assigning of signals happen with a period known as simulation delta (delta cycle).*



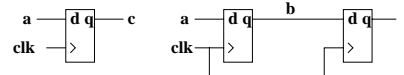
## One or Two Flip-flops?



```

-- Does this have one or two flip-flops?
library ieee;
use ieee.std_logic_1164.all;
entity oneortwo is
  port (a, clk : in std_logic;
        c : out std_logic);
end oneortwo;
architecture top of oneortwo is
  signal b : std_logic;
begin
  reg2: PROCESS (clk)
  BEGIN
    if rising_edge(clk) then
      b <= a;
      c <= b;
    end if;
  END PROCESS reg2;
end top;

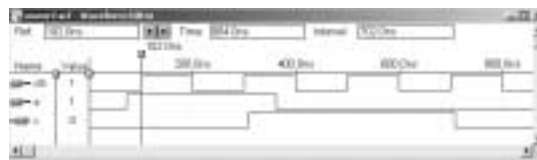
```



```

-- What if we
-- change the order
-- of these statements?
c <= b;
b <= a;

```





## Model for T Flip-flop



```
library ieee;
use ieee.std_logic_1164.all;

entity t_ff is
    port(clk, t : in std_logic;
         q : out std_logic);
end t_ff;

architecture behavioral of t_ff is

    signal qinside : std_logic;

begin
    process(clk)
    begin
        if (falling_edge(clk)) then
            if t = '1' then
                qinside <= not qinside;
            end if;
        end if;
    end process;
    q <= qinside;
end behavioral;
```

- Sequential logic is described using processes.
- The entire body of a process executes in one delta of simulation time.
- Individual processes are like individual concurrent statements and execute simultaneously.



## Modeling a D Flip-Flop in VHDL



```
library ieee;
use ieee.std_logic_1164.all;

entity d_ff is
    port ( clk : in std_logic;
          clear : in std_logic;
          d : in std_logic;
          q : out std_logic;
          nq : out std_logic);
end d_ff;

architecture behavioral of d_ff is

begin
    process (clear, clk)
    begin
        if clear = '1' then
            q <= '0';
            nq <= '1';
        elsif clk'event and clk = '1' then
            q <= d;
            nq <= not(d);
        end if;
    end process;
end behavioral;
```

- This D flip-flop has an asynchronous clear.
  - What would you do to get a synchronous clear?
- clk'event and clk = 1 is the same as rising\_edge(clk).



## Anatomy of a Process



```
process (clear, clk) ←  
begin  
  if clear = '1' then  
    q <= '0';  
    nq <= '1';  
  elsif clk'event and clk = '1' then  
    q <= d;  
    nq <= not(d);  
  end if;  
end process;
```

### **Sensitivity list:**

Contains a list of signals that trigger the process

- The process body is evaluated each time one of the signals in the sensitivity list changes state.
- Most processes contain at least the global clock and reset signals in their sensitivity lists.
- The sensitivity list is NOT used for synthesis.



## Anatomy of a Process



```
process (clear, clk)  
begin  
  if clear = '1' then  
    q <= '0';  
    nq <= '1';  
  elsif clk'event and clk = '1' then  
    q <= d;  
    nq <= not(d);  
  end if;  
end process;
```

**Variable declarations**  
(if any)

- Processes may optionally define their own local variables.
- Variables are updated when a sequential statement is executed.
- Signals are SCHEDULED to be updated when a statement is executed.



## Anatomy of a Process



```
process (clear, clk)
begin
  if clear = '1' then
    q <= '0';
    nq <= '1';
  elsif clk'event and clk = '1' then
    q <= d;
    nq <= not(d);
  end if;
end process;
```

**Sequential  
statements**

- Body of process consists of sequential statements.
- Statements are executed in order each time process is run.



## Using Wait Statements



- Wait statements delay the execution of the process until expression becomes true.
- No wait statements allowed in the body if there is a sensitivity\_list.
- wait until (boolean\_expression);  
wait on (signal\_name {,signal\_name});  
wait for (for time\_expression);
- A simple D FF

```
process
begin
  wait until (clk'event and clk='1');
  q <= d;
  nq <= not(d);
end process;
```



## Sequential Statements



Some example sequential statements:

### Assignment

```

<name> <= <value>; -- for signals
<name> := <value>; -- for variables

```

### If statements

```

if <condition> then
  <statements>
elsif <condition> then
  <other statements>
else
  <statements>
end if;

```

} Optional but recommended

### Case statements

```

case <signal/variable> is
  when <value1> => <statements>
  when <value2> => <statements>
  . . .
  when others => <statements>
end case;

```

- Absence of *else* may result in implicit memory (if all possible conditions are not specified).
- Conditions do not have to be mutually exclusive.

- *when* clauses must be mutually exclusive. Always use a *when others* at the end.



## Inferring Registers and Latches



- Latches are level-sensitive and are much less common than registers.

```

process (c)
begin
  if c'event and c = '1' then
    q <= d;
  -- else q <= q;
  -- makes memory explicit
  end if;
end process;

```

**This infers a register.**

Note that the body of the *if* statement only executes when a transition occurs on *c*.

```

process (c, d)
begin
  if c = '1' then
    q <= d;
  -- else q <= q;
  -- makes memory explicit
  end if;
end process;

```

**This infers a latch.**

The body of the *if* statement executes whenever a transition occurs on *c* or *d*.



## Synchronous Counter Implementation



- D flip-flop for each state bit
- Combinational logic based on encoding

C3	C2	C1	N3	N2	N1
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

$$N1 := \overline{C1}$$

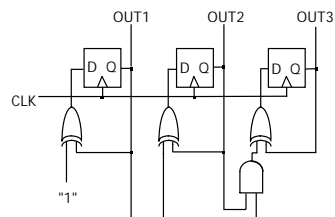
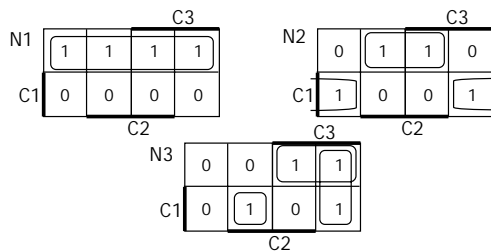
$$N2 := C1 \overline{C2} + \overline{C1} C2$$

$$:= C1 \text{ xor } C2$$

$$N3 := C1 C2 \overline{C3} + \overline{C1} C3 + \overline{C2} C3$$

$$:= C1 C2 \overline{C3} + (\overline{C1} + \overline{C2}) C3$$

$$:= (C1 C2) \text{ xor } C3$$



From [Katz93]

L5 6.111 Fall 2003 – Introductory Digital Systems Laboratory

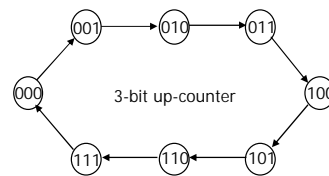
17



## Implementing with T Flip-Flop



QC	QB	QA	NC	NB	NA	TC	TB	TA
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1



**Decide to implement with Toggle Flip-flops.**

**What inputs must be presented to the T FFs to get them to change to the desired state bit?**

**This is called "Remapping the Next State Function".**

L5 6.111 Fall 2003 – Introductory Digital Systems Laboratory

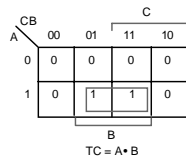
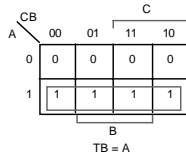
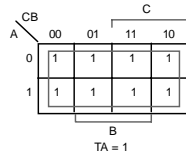
18



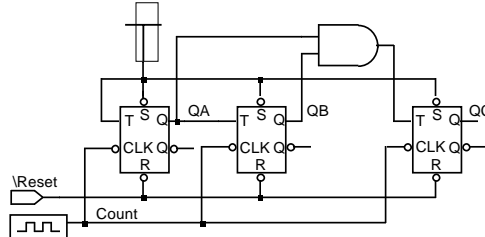
## With Toggle Inputs



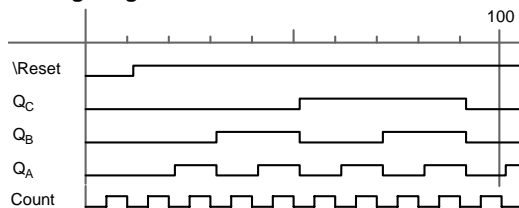
### K-maps for Toggle Inputs:



### Resulting Logic Circuit:



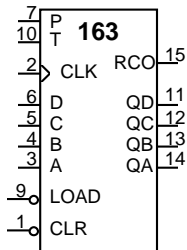
### Timing Diagram:



From [Katz93]



## Catalog Counter: 74163



74163 Synchronous 4-Bit Upcounter

### Synchronous Load and Clear Inputs

### Positive Edge-Triggered FFs

### Parallel Load Data from D, C, B, A

P, T Enable Inputs: both must be asserted to enable counting

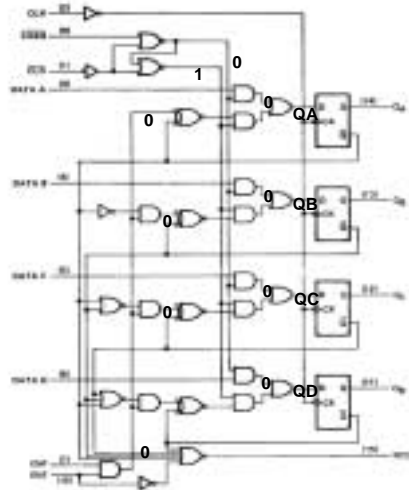
RCO: asserted when counter enters its highest state 1111 and T = 1, used for cascading counters "Ripple Carry Output"

CLR and LOAD are synchronous  
 If CLR = 0 then Q := 0  
 Else if LOAD then Q := D  
 Else if P \* T = 1 then Q := Q + 1  
 Else Q := Q

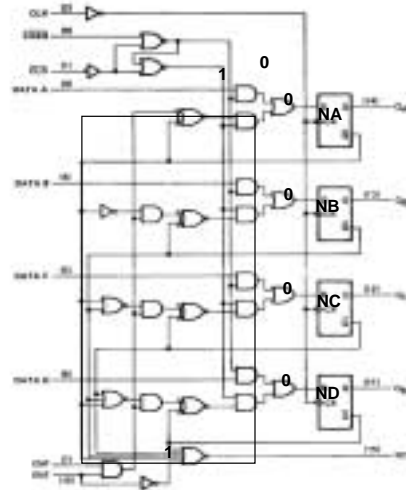




## Counter Modes (cont.)



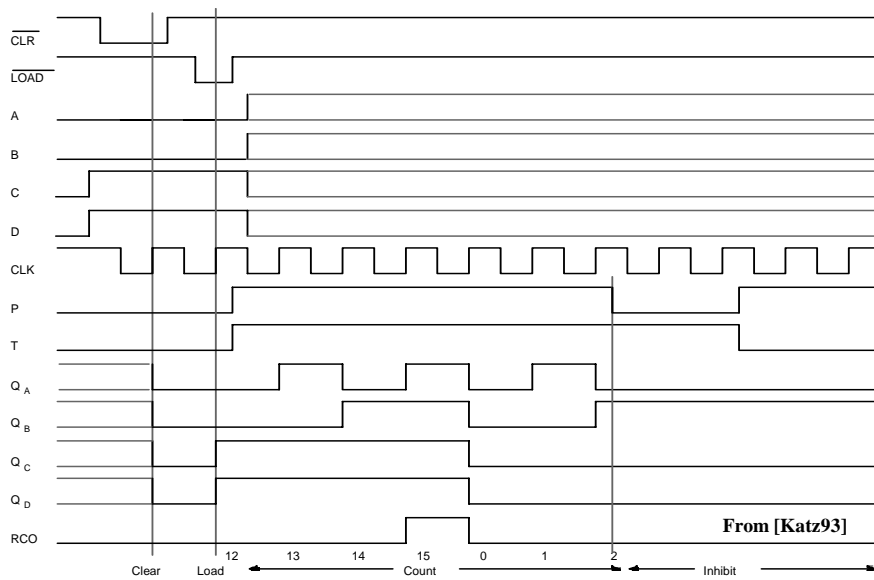
**CLR = 1, LOAD = 1, P T = 0**



**CLR = 1, LOAD = 1, P T = 1**



## 74163 Detailed Timing Diagram





## VHDL for 74163



```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity c74163 is
  port (LDN, CLRN, P, T, CLK: in std_logic;
        D : in std_logic_vector (3 downto 0);
        count : out std_logic_vector (3 downto 0);
        RCO : out std_logic);
end c74163;

architecture behavioral of c74163 is

  signal Q: std_logic_vector (3 downto 0);

begin
  count <= Q;
  RCO <= Q(3) and Q(2) and Q(1) and Q(0) and T;

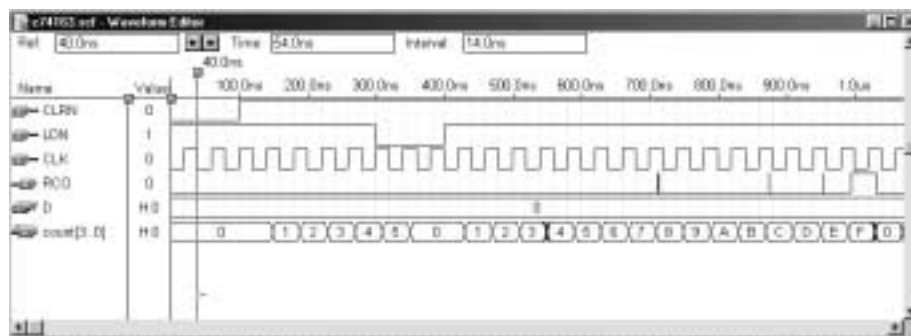
  process
  begin
    wait until (CLK'event and CLK = '1');
    if CLRN = '0' then Q<= "0000";
    elsif LDN = '0' then Q<= D;
    elsif (P and T) = '1' then Q<= Q+ 1;
    end if;
  end process;
end architecture behavioral;

```

- Our entire counter example can be defined in one process.
  - Behavioral, rather than structural specification
  - Synthesis tools will map behavior to logic.
  - This is the more typical style of coding.
  - Is CLRN synchronous or asynchronous?



## Simulation



**There should be no glitches on the output (ideally).  
Don't all FFs change at the same time?**

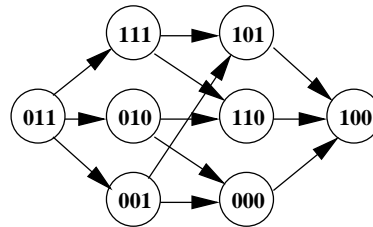


## Output Transitions

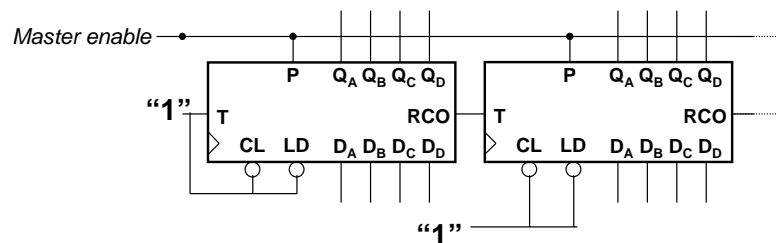


- Any time multiple bits change, the counter output needs time to settle.
- Even though all flip-flops share the same clock, individual bits will change at different times.
  - Clock skew, propagation time variations
- Can cause glitches in combinational logic driven by the counter
- The RCO can also have a glitch.

Care is required of the  
Ripple Carry Output:  
It can have glitches:  
Any of these transition  
paths are possible!



## Cascading 74163s



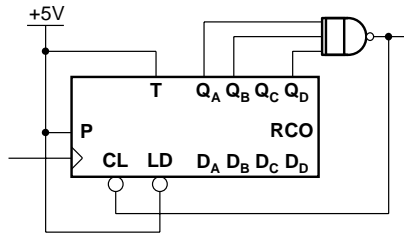
- '163 is enabled only if P and T are high.
- RCO goes high when T is a logic one and the '163 is about to roll over to 0000.
- Connecting RCO to T makes second '163 increment once whenever first '163 rolls over.
- P on all '163s is connected to master enable.



## Counter Tricks

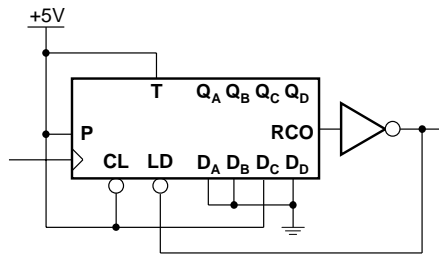


### Configured to automatically reset at 1011



Counts 0000, 0001, ..., 1010,  
1011, 0000, ...

### Configured to automatically load 0100



Counts 0100, 0101, ..., 1110,  
1111, 0100, ...



## Up/Down Counter with Enable



```

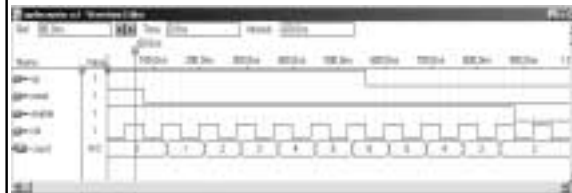
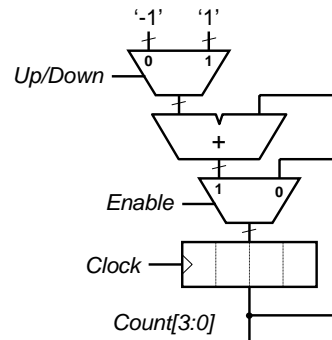
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity updowncounter is
port (clk : in std_logic;
      reset : in std_logic;
      up : in std_logic;
      enable : in std_logic;
      count : out std_logic_vector (3 downto 0));
end updowncounter;

architecture behavioral of updowncounter is
signal Q : std_logic_vector (3 downto 0);
begin
count <= Q;

process (clk, reset)
begin
if reset = '1' then
Q <= (others => '0');
elsif clk'event and clk = '1' then
if enable = '1' and up = '1' then
Q <= Q + 1;
elsif enable = '1' and up = '0' then
Q <= Q - 1;
end if;
end if;
end process;
end behavioral;

```

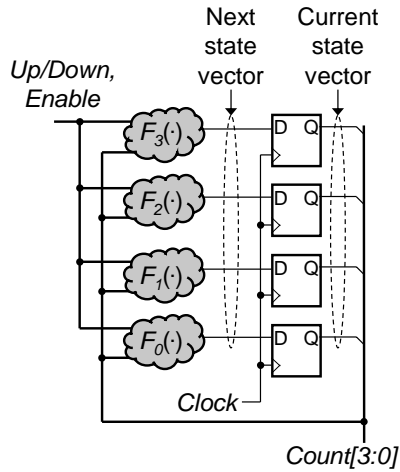




# The Up/Down Counter as an FSM



Next state functions  $F_0(), \dots, F_3()$  are derived from a state transition table.



Partial state transition table

Inputs		Current State				Next State			
<i>U/D</i>	<i>En</i>	$C_0$	$C_1$	$C_2$	$C_3$	$C_0'$	$C_1'$	$C_2'$	$C_3'$
...	...	...	...	...	...	...	...	...	...
0	1	0	0	0	0	1	1	1	1
0	1	1	0	0	0	0	0	0	0
0	1	0	1	0	0	1	0	0	0
0	1	1	1	0	0	0	1	0	0
1	1	0	0	1	0	1	1	0	0
1	1	1	0	1	0	0	0	1	0
1	1	0	1	1	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0
...	...	...	...	...	...	...	...	...	...

Karnaugh-maps, Boolean algebra, etc... used to simplify and implement state transition functions.