Gated-Clock SR Flip-Flop (Latch Enable)

\[
Q \leftarrow (S \text{ NAND } LE) \text{ NAND } NQ;
\]

\[
NQ \leftarrow (R \text{ NAND } LE) \text{ NAND } Q;
\]

Synchronous terminology:
Set and Reset

Asynchronous terminology:
Preset and Clear

Latches require that during the gated-clock the data must also be stable (i.e. S and R) at the same time.

Suppose each gate was 5ns: how long does the clock have to be enabled to latch the data?

Answer: 15ns
Structural SR Flip-Flop (Latch)

### NAND Truth Table

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>(Q_{n+1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(Q_n)</td>
</tr>
</tbody>
</table>

### VHDL Code

```vhdl
ENTITY Latch IS
  PORT(R, S: IN std_logic; Q, NQ: OUT std_logic);
END ENTITY;

ARCHITECTURE latch_arch OF Latch IS
BEGIN
  Q <= R NAND NQ;
  NQ <= S NAND Q;
END ARCHITECTURE;
```
Inferring Behavioral Latches: Asynchronous

ARCHITECTURE Latch2_arch OF Latch IS
BEGIN
  PROCESS (R, S) BEGIN
    IF R = '0' THEN
      Q <= '1'; NQ <= '0';
    ELSIF S = '0' THEN
      Q <= '0'; NQ <= '1';
    END IF;
  END PROCESS;
END ARCHITECTURE;

Sensitivity list of signals:
Every time a change of state or event occurs on these signals this process will be called

Sequential Statements
ARCHITECTURE Latch_arch OF GC_Latch IS BEGIN
PROCESS (R, S, LE) BEGIN
IF LE='1' THEN
  IF R='0' THEN
    Q <= '1'; NQ<='0';
  ELSIF S='0' THEN
    Q <= '0'; NQ<='1';
  END IF;
END IF;
END PROCESS;
END ARCHITECTURE;
Rising-Edge Flip-flop

D-Type Flip Flop

Q <= D when Clk changes from '0' to

Dashed vertical lines indicate when an input to the flip flop has changed.

Clk
D
Q

t1 t2 t3 t4 t5 t6 t7 t8
Rising-Edge Flip-flop logic diagram

Do not want to code this up as combinatorial logic! Too much work!
Inferring D-Flip Flops: Synchronous

ARCHITECTURE Dff_arch OF Dff IS
BEGIN
    PROCESS (Clock) BEGIN
        IF Clock'EVENT AND Clock='1' THEN
            Q <= D;
        END IF;
        END IF;
    END PROCESS;
END ARCHITECTURE;

Notice the Process does not contain D:
PROCESS(Clock, D)

Sensitivity lists contain signals used in conditionals (i.e. IF)

Clock’EVENT is what distinguishes a D-FlipFlip from a Latch
Inferring D-Flip Flop: rising_edge

ARCHITECTURE Dff_arch OF Dff IS BEGIN
    PROCESS (Clock) BEGIN
        IF Clock'EVENT AND Clock='1' THEN
            Q <= D;
        END IF;
    END PROCESS;
END ARCHITECTURE;

ARCHITECTURE dff_arch OF dff IS BEGIN
    PROCESS (Clock) BEGIN
        IF rising_edge(Clock) THEN
            Q <= D;
        END IF;
    END PROCESS;
END ARCHITECTURE;

Alternate and more readable way is to use the rising_edge function
ARCHITECTURE dff_reset_arch OF dff_reset IS BEGIN

PROCESS (Clock, Reset) BEGIN

    IF Reset = '1' THEN -- Asynchronous Reset
        Q <= '0'
    ELSIF rising_edge(Clock) THEN -- Synchronous
        Q <= D;
    END IF;
END PROCESS;

END ARCHITECTURE;
Inferring D-Flip Flops: Synchronous Reset

PROCESS (Clock, Reset) BEGIN
    IF rising_edge(Clock) THEN
        IF Reset='1' THEN
            Q <= '0'
        ELSE
            Q <= D;
        END IF;
    END IF;
END IF;
END PROCESS;

PROCESS (Clock, Reset) BEGIN
    IF Reset='1' THEN
        Q <= '0'
    ELSIF rising_edge(Clock) THEN
        Q <= D;
    END IF;
END IF;
END PROCESS;

Synchronous Reset
Synchronous FF

Asynchronous Reset
Synchronous FF
D-Flip Flops: Asynchronous Reset & Preset

PROCESS (Clock, Reset, Preset) BEGIN
  IF Reset='1' THEN --highest priority
    Q <= '0';
  ELSIF Preset='1' THEN
    Q <= '0';
  ELSIF rising_edge(Clock) THEN
    Q <= D;
  END IF;
END PROCESS;

END PROCESS;
VHDL clock behavioral component

ENTITY clock_driver IS
    GENERIC (Speed: TIME := 5 ns);
    PORT (Clk: OUT std_logic);
END;

ARCHITECTURE clock_driver_arch OF clock_driver IS
    SIGNAL Clock: std_logic := '0';
BEGIN
    Clk <= Clk XOR '1' after Speed;
    Clock <= Clk;
END ARCHITECTURE;

CONFIGURATION  clock_driver_cfg OF clock_driver IS
    FOR clock_driver_arch END FOR;
END CONFIGURATION;
Synchronous Sequential Circuit

Issues: Specification, design, clocking and timing
Abstraction: Finite State Machine

- A Finite State Machine (FSM) has:
  - $K$ states, $S = \{s_1, s_2, \ldots, s_K\}$, initial state $s_1$
  - $N$ inputs, $I = \{i_1, i_2, \ldots, i_N\}$
  - $M$ outputs, $O = \{o_1, o_2, \ldots, o_M\}$
  - Transition function $T(S, I)$ mapping each current state and input to a next state
  - Output function $O(S)$ mapping each current state to an output

- Given a sequence of inputs the FSM produces a sequence of outputs which is dependent on $s_1$, $T(S, I)$ and $O(S)$
FSM Representations

State Transition Graph

Initial state

\[ \begin{align*}
& s_1 \\
& s_2 \\
& s_3
\end{align*} \]

State Transition Table

\[
\begin{array}{ccc}
T(S, I) \\
0 & s_1 & s_2 \\
1 & s_1 & s_3 \\
0 & s_2 & s_1 \\
1 & s_2 & s_2 \\
0 & s_3 & s_1 \\
1 & s_3 & s_3
\end{array}
\]

\[
\begin{array}{c}
O(S) \\
s_1 & 00 \\
s_2 & 10 \\
s_3 & 11
\end{array}
\]

Inputs: 0 1 0

Outputs: 00

\[
\begin{align*}
& t \\
& t+1 \\
& t+2
\end{align*} \]
Moore Machines

- So far we considered Moore machines where the output \( Q \) is a function of only the current state \( Q \).

- Moore FSM State Transition Graph
Simple Design Example

- Design a FSM that outputs a 1 if and only if the number of 1’s in the input sequence is odd

ENTITY FSM_Parity IS
  PORT (i1: std_logic;
        o1: OUT std_logic;
        CLK: IN std_logic; --Clock
        RST: IN std_logic --Reset
  ); END;
State Encoding is sequentially done by VHDL

TYPE FSMStates IS (s1, s2); -- s1=0, s2=1
SIGNAL State, NextState: FSMStates;

• State Encoding: Choose a unique binary code for each $s_i$ so the combinational logic can be specified
  – Choose $s_1 = 0$ and $s_2 = 1$
  – Choose $s_1 = 1$ and $s_2 = 0$

– The non-sequential case requires the following
ATTRIBUTE FSMencode: string;
ATTRIBUTE FSMencode of FSMStates: TYPE IS “1 0”;
PROCESS (State, i1) BEGIN

CASE State IS

WHEN s1 => if i1='1' then NextState <= s2;
else NextState <= s1; end if;

WHEN s2 => if i1='1' then NextState <= s1;
else NextState <= s2; end if;

WHEN OTHERS => NextState <= NextState;

END CASE;

END PROCESS;

--- Diagram ---

- **Even**
  - $o_1 = 0$
  - $i_1 = 0$ to $s_1$
  - $s_1$ to $i_1 = 1$
  - $i_1 = 0$

- **Odd**
  - $o_1 = 1$
  - $i_1 = 1$ to $s_2$
  - $s_2$ to $i_1 = 0$
  - $i_1 = 1$
ARCHITECTURE FSM_Parity_arch OF FSM_Parity IS
TYPE FSMStates IS (s1, s2);
SIGNAL State, NextState: FSMStates;
BEGIN

PROCESS (State, i1) BEGIN
CASE State IS
  WHEN s1 => if i1='1' then NextState <= s2;
   else NextState <= s1; end if;
  WHEN s2 => if i1='1' then NextState <= s1;
   else NextState <= s2; end if;
  WHEN OTHERS => NextState <= NextState;
END CASE;
END PROCESS;

WITH State SELECT
  o1 <= '0' WHEN s1,
       '1' WHEN s2,
       '1' WHEN OTHERS;   - - X, L, W, H, U
Alternative: less coding

ARCHITECTURE FSM_Parity_arch OF FSM_Parity IS
    TYPE FSMStates IS (s1, s2);
    SIGNAL State, NextState: FSMStates;
BEGIN
    PROCESS (State, i1) BEGIN
        CASE State IS
            WHEN s1 =>
                if i1='1' then
                    NextState <= s2;
                else
                    NextState <= s1;
                end if;
                o1 <= '0';
            WHEN s2 =>
                if i1='1' then
                    NextState <= s1;
                else
                    NextState <= s2;
                end if;
                o1 <= '1';
            WHEN OTHERS =>
                o1 <= '1';
                NextState <= NextState;
        END CASE;
    END PROCESS;

Important Note:
every input to the state machine must be in the PROCESS sensitivity list.

Important Note: every WHEN must assign the same set of signals: i.e. NextState and o1. If you miss one assignment, latches will show up!
FSM controller: **NextState** Process

```vhdl
PROCESS (CLK, RST) BEGIN
  IF RST='1' THEN -- Asynchronous Reset
    State <= s1;
  ELSIF rising_edge(CLK) THEN
    State <= NextState;
  END IF;
END PROCESS;

END ARCHITECTURE;

CONFIGURATION FSM_Parity_cfg OF FSM_Parity IS
  FOR FSM_Parity_arch
    END FOR;
END CONFIGURATION;
```
Choose $s_1 = 0$ and $s_2 = 1$

Choose $s_1 = 1$ and $s_2 = 0$
Coke Machine Example

- Coke costs $0.10
- Only nickels and dimes accepted
- FSM inputs:
  - 5: Nickel
  - 10: Dime
  - Coke: Give me a coke
  - Return: Give me my money back
- FSM outputs:
  - Drop: Drop a coke
  - Ret5: Return $0.05
  - Ret10: Return $0.10
Coke Machine State Diagram

Assumption: At most one input among Coke, 5, 10, and Return is asserted

* represents all unspecified transitions from state

Does this work? ________________
After Return input, any input in the next cycle is ignored!
Assignment #6

a) Write the VHDL synchronous code *(no latches!)* and test bench for the coke II machine. Note: the dc_shell synthesis analyze command will tell you if you inferred latches. Hand code and simulation using the Unix script command.

b) Synthesize the your design and hand in the logic diagram, Unix script include cell, area, timing report.