Sequential Circuits

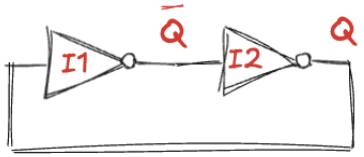
Index

Sequential Logic

- The output of **combinational logic** depends only on **current input** values
  - given a specification in the form of a truth table or boolean equation, we can create an optimized circuit to meet the specification
- The outputs of **sequential logic** depend on both **current and prior input** values
  - has **memory**
  - explicitly remember previous inputs
  - or it might **distil the prior inputs into a smaller amount of information** called the **state of the system**
    - a set of bits (**state variables**) that contain all the information about the past necessary to explain the future behaviour of the circuit
  - **latches** and **flip-flops** are simple sequential circuits that store **one bit of state**
- Sequential circuits are **complicated to analyse**
  - we can discipline ourselves to build only **synchronous** sequential circuits
  - combinational logic and banks of flip-flops containing the state of the circuit

Bistable element (1)

- The fundamental building block of **memory**: an element **with two stable states**
  - The inverters are cross-coupled
    - the input of I1 is the output of I2 and vice versa
    - the circuit has no inputs, but it does have two outputs



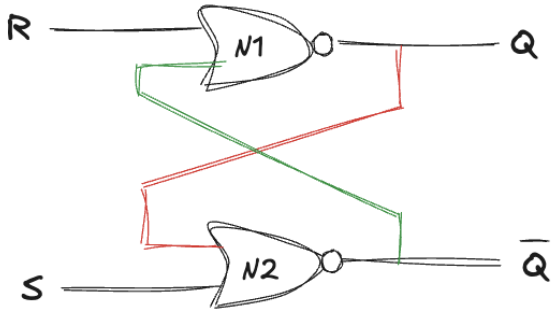
- This circuit is **cyclic**
  - Case 1: Q = 0
    - I2 receives a FALSE input, so it produces a TRUE output on $\bar{Q}$
    - I1 receives a TRUE input, so it produces a FALSE output on Q
    - this is consistent with the original assumption (Q = 0), so it is **stable**
  - Case 2: Q = 1
    - I2 receives a TRUE input and produces a FALSE output on $\bar{Q}$
    - I1 receives a FALSE input and produces a TRUE output on Q
    - this is again stable
- Because the cross-coupled inverters have **two stable states**, the circuit is said to be **bistable**

Bistable element (2)

- An element with N stable states conveys $\log_2 N$ bits of information, so a bistable element stores **one bit**
  - state of the cross-coupled inverters is contained in one binary state variable (Q)
  - the value of Q tells us everything about the past that is necessary to explain the future behaviour of the circuit
    - if Q=0, it will remain 0 forever
    - if Q=1, it will remain 1 forever
- When power is first applied to a sequential circuit, the initial state is **unknown** and usually **unpredictable**
  - it may differ each time the circuit is turned on
- Although the cross-coupled inverters can store a bit of information, they are not practical because the **user has no inputs to control the state**

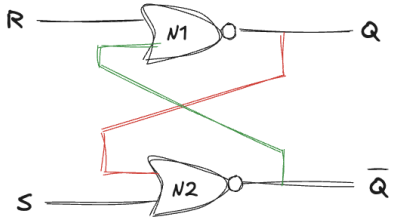
SR Latch (1)

- Two cross-coupled NOR gates



- Like the cross-coupled inverters, but its **state can be controlled** through the inputs
  - set (S) and reset (R) the output
- Case 1: R=1, S=0
  - N1 sees at least one TRUE input R, so it produces a FALSE output on Q
  - N2 sees both Q and S FALSE, so it produces a TRUE output on $\overline{Q}$
- Case 2: R=0, S=1
  - N1 receives inputs of 0 and $\overline{Q}$, however we don't yet know $\overline{Q}$
  - N2 receives at least one TRUE input S, so it produces a FALSE output on $\overline{Q}$
  - Now we can revisit N1, so the output Q is TRUE

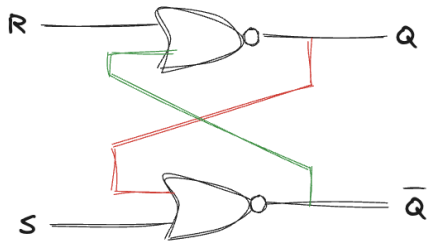
SR Latch (2)



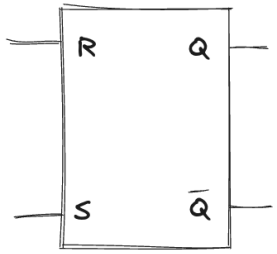
- Case 3: R=1, S=1
  - N1 and N2 both see at least one TRUE input, so each produces a FALSE output
- Case 4: R=0, S=0
  - N1 receives inputs of 0 and $\overline{Q}$, however we don't yet know $\overline{Q}$
  - N2 receives inputs of 0 and Q. however we don't yet know Q
  - now we are stuck, but we know that Q must either be 0 or 1
  - Case 4a: Q=0
    - N2 produces a TRUE output on $\overline{Q}$
    - N1 receives one TRUE input ($\overline{Q}$), so its output is FALSE, just as we had assumed
  - Case 4b: Q=1
    - N2 produces a FALSE output on $\overline{Q}$
    - N1 receives two FALSE inputs, so its output is TRUE, just as we had assumed
  - Putting this all together: when R and S are 0, Q will **remember** the old value
    - this circuit has **memory**

SR Latch (3)

- The truth table summarizes these four cases



	S	R	Q	$\overline{Q}$
(1)	0	0	Q_{prev}	$\overline{Q}_{prev}$
(2)	0	1	0	1
(3)	1	0	1	0
(4)	1	1	0	0



- The inputs S and R stand for **Set** and **Reset**
  - “set” a bit means to make it TRUE
  - “reset” a bit means to make it FALSE
- When neither input is asserted, Q **remembers** its old value
- Asserting both S and R simultaneously **doesn’t make sense**
  - it means the latch should be set and reset at the same time
  - the confused circuit responds by making both outputs 0

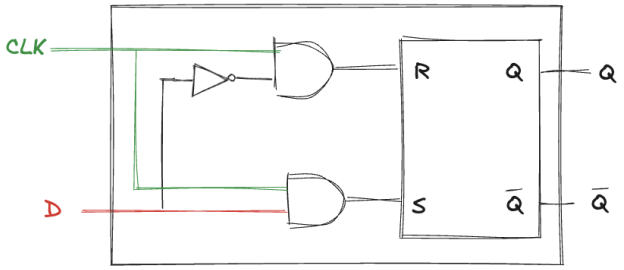
SR Latch VHDL

```
entity SR_LATCH is
    port( S: in std_logic;
          R: in std_logic;
          Q: out std_logic
    );
end SR_LATCH;
```

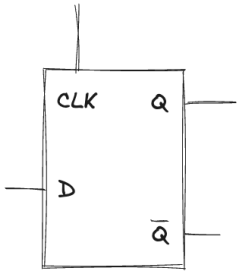
```
architetcture rtl of SR_LATCH is
begin
    latch: process(S, R)
    begin
        if( S='1' and R='0') then
            Q <= '1';
        elsif( S='0' and R='1') then
            Q <= '0';
        elsif( S='0' and R='0') then
            null;
        elsif( S='1' and R='1') then
            Q <= 'X';
        end if;
    end process;
end rtl;
```

D Latch

- SR latch is awkward
  - it behaves strangely when both S and R are simultaneously asserted
  - moreover, S and R conflate the **issues of what and when**
    - asserting one input determines **what** the state should be and **when** it should change
- Design is easier when these questions (what and when) are **separated**
- The D latch avoids the strange case of simultaneously asserted R and S inputs
  - **data input** D controls what the next state should be
  - **clock input** CLK controls when the state should change
  - if CLK=0, S and R are FALSE (regardless of D) and Q remembers its old value
  - if CLK=1, one AND is TRUE and the other is FALSE (depending on D) and Q = D
  - In all cases, $\overline{Q}$ is the complement of Q



CLK	D	S	R	Q	$\overline{Q}$
0	x	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	0	1	0	1
1	1	1	0	1	0



D Latch VHDL

```
entity D_LATCH is
    port( D: in std_logic;
          CLK: in std_logic;
          Q: out std_logic
        );
end D_LATCH;
```

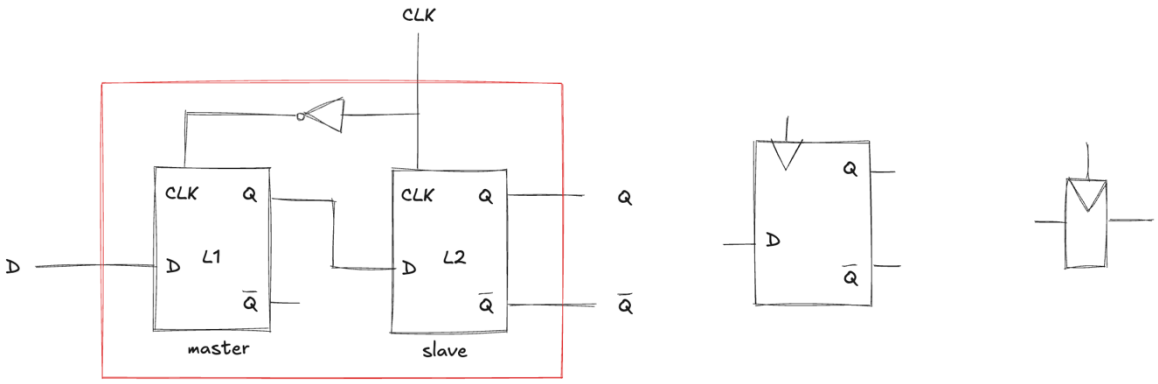
```
architetcture rtl of D_LATCH is
    begin
        latch: process(D, CLK)
        begin
            if( CLK = '1' ) then
                Q <= D;
            end if;
        end process;
end rtl;
```

Clock

- The clock controls **when** data flows through the latch
  - when CLK = 1, the latch is **transparent**
    - data at D flows through to Q as if the latch were just a buffer
  - when CLK = 0, the latch is **opaque**
    - it blocks the new data from flowing through to Q, and Q retains the old value

D Flip-Flop

- Two back-to-back D latches controlled by complementary clocks



  - L1 is called the **leader/master**
  - L2 is called the **follower/slave**

- It copies D to Q on the **rising edge** of the clock and remembers its state at all other times:
  - when CLK=0, the master is transparent and the slave is opaque
  - when CLK=1, the master goes opaque and the salve becomes transparent

- The **D input** specifies **what** the new state will be, the **clock edge** indicates **when** the state should be updated

D Flip-Flop VHDL

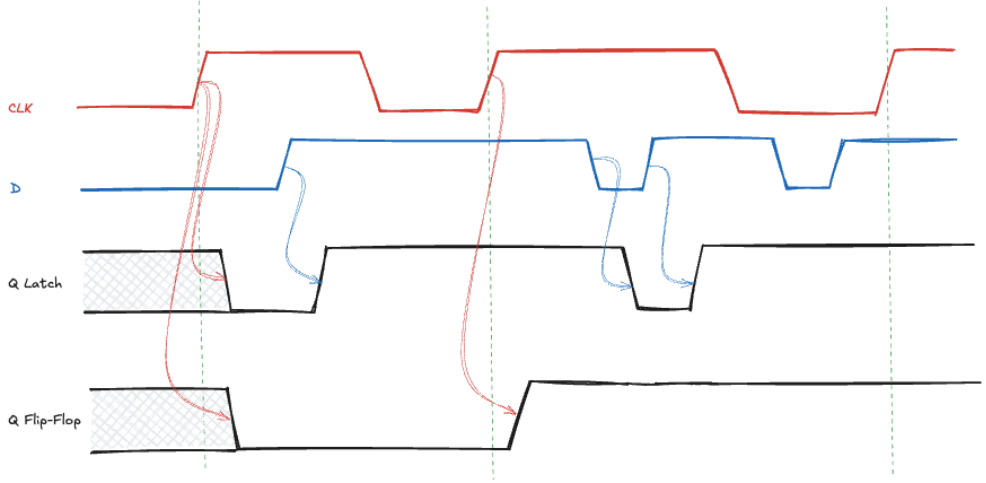
```
entity D_FF is
    port( CLK: in std_logic;
          D: in std_logic;
          Q: out std_logic
    );
end D_FF;
```

```
architecture rtl_rising_edge of D_FF is
    begin
        ff: process(CLK)
        begin
            if(CLK'event and CLK='1') then
                Q <= D;
            end if;
        end process;
end rtl_rising_edge;
```

```
state_register : process(CLK)
    begin
        if rising_edge(CLK) then
            Q <= Q_next;
        end if;
    end process;
```

Flip-Flop and Latch comparison

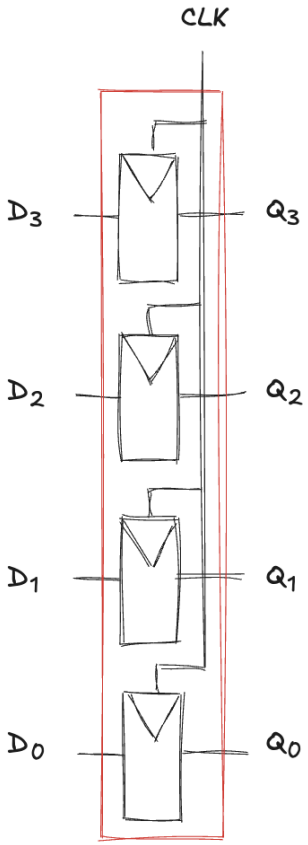
- Apply D and CLK inputs to a D latch and a D flip-flop to determine Q

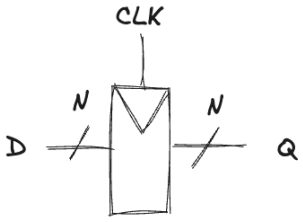


- The initial value of Q is **unknown**
- Consider the latch:
  - on the first CLK rising edge, D=0 so Q becomes 0
  - each time D changes while CLK=1, Q follows
  - when D changes while CLK=0, D is ignored
- Consider the flip-flop:
  - **on each CLK rising edge**, D is copied to Q
  - at all other times, Q retains its state

Register

- An N-bit register is a **bank** of N flip-flops that **share a common CLK**
  - all bits of the register are updated at the same time
  - registers are the **key building block of most sequential circuits**





Register VHDL

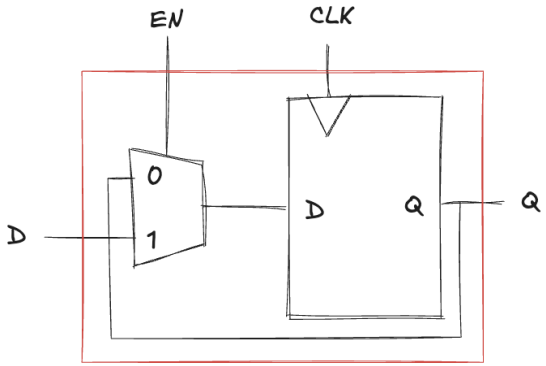
```
entity Register is
    generic(N : integer := 8);
    port(CLK : in  std_logic;
         D   : in  std_logic_vector(N-1 downto 0);
         Q   : out std_logic_vector(N-1 downto 0)
    );
end Register;
```

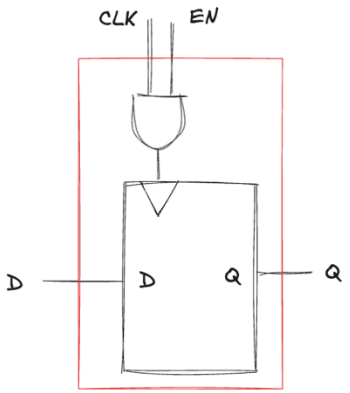
```
architecture structural of Register is
    component D_FF is
        port(CLK : in  std_logic;
             D   : in  std_logic;
             Q   : out std_logic
        );
    end component;
begin
    GEN_FF : for i in 0 to N-1 generate
        FF_i : D_FF
            port map(CLK => CLK, D => D(i), Q => Q(i));
    end generate GEN_FF;

end structural;
```

Enabled Flip-Flop

- Another input called **Enable** to determine if data is loaded on the clock edge
  - when Enable is TRUE, the flip-flop behaves like an ordinary D flip-flop
  - when Enable is FALSE, the flip-flop ignores the clock and retains its state
- Useful when we wish **to load a new value into a flip-flop only some of the time**, rather than on every clock edge
- Two ways to implement it
  - an input multiplexer:
    - the value D pass if Enable is TRUE
    - the old state is recycled if Enable is FALSE
  - the clock can be gated:
    - if Enable is TRUE, the CLK toggles normally
    - if Enable is FALSE, the CLK input is also FALSE
    - notice that Enable must not change while CLK=1 or the flip-flop see a **clock glitch** (switch at an incorrect time)
- Generally, **performing logic on the clock is a bad idea**





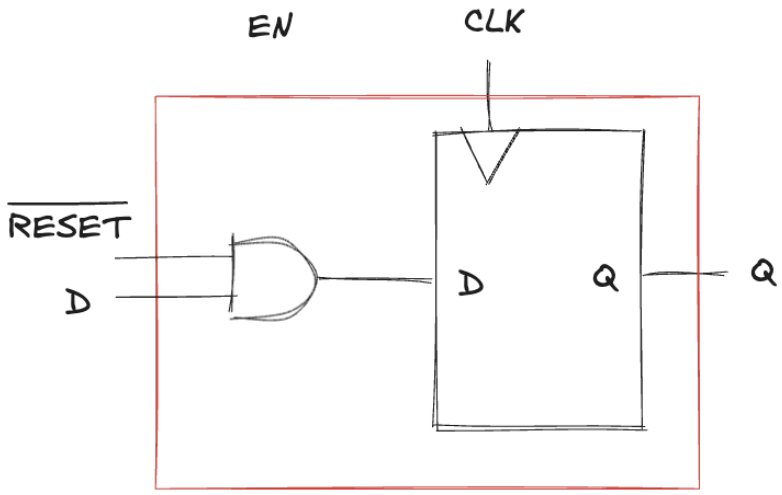
Enabled Flip-Flop VHDL

```
entity DFF_en is
    port(CLK : in  std_logic;
         EN  : in  std_logic;
         D   : in  std_logic;
         Q   : out std_logic
    );
end entity;
```

```
architecture rtl of DFF_en is
begin
    process(CLK)
    begin
        if rising_edge(CLK) then
            if EN='1' then
                Q <= D;
            end if;
        end if;
    end process;
end rtl;
```

Resettable Flip-Flop

- Another input, called **Reset**
  - when Reset is FALSE, the flip-flop behaves like an ordinary D flip-flop
  - when Reset is TRUE, the flip-flop ignores D and resets the output to 0
- Useful when we want to **force a known state into all the flip-flops** in a system when we first turn it on
- We can construct it from an ordinary D flip-flop and an AND gate



  - the reset is an **active low signal**, meaning that it performs its function when it is 0
  - by adding an inverter, the circuit could have accepted an **active high** reset signal

Resettable Flip-Flop

```
entity DFF_reset_low is
    port(CLK   : in  std_logic;
         RESET : in  std_logic;   -- active low
         D     : in  std_logic;
         Q     : out std_logic
    );
end entity DFF_reset_low;
```

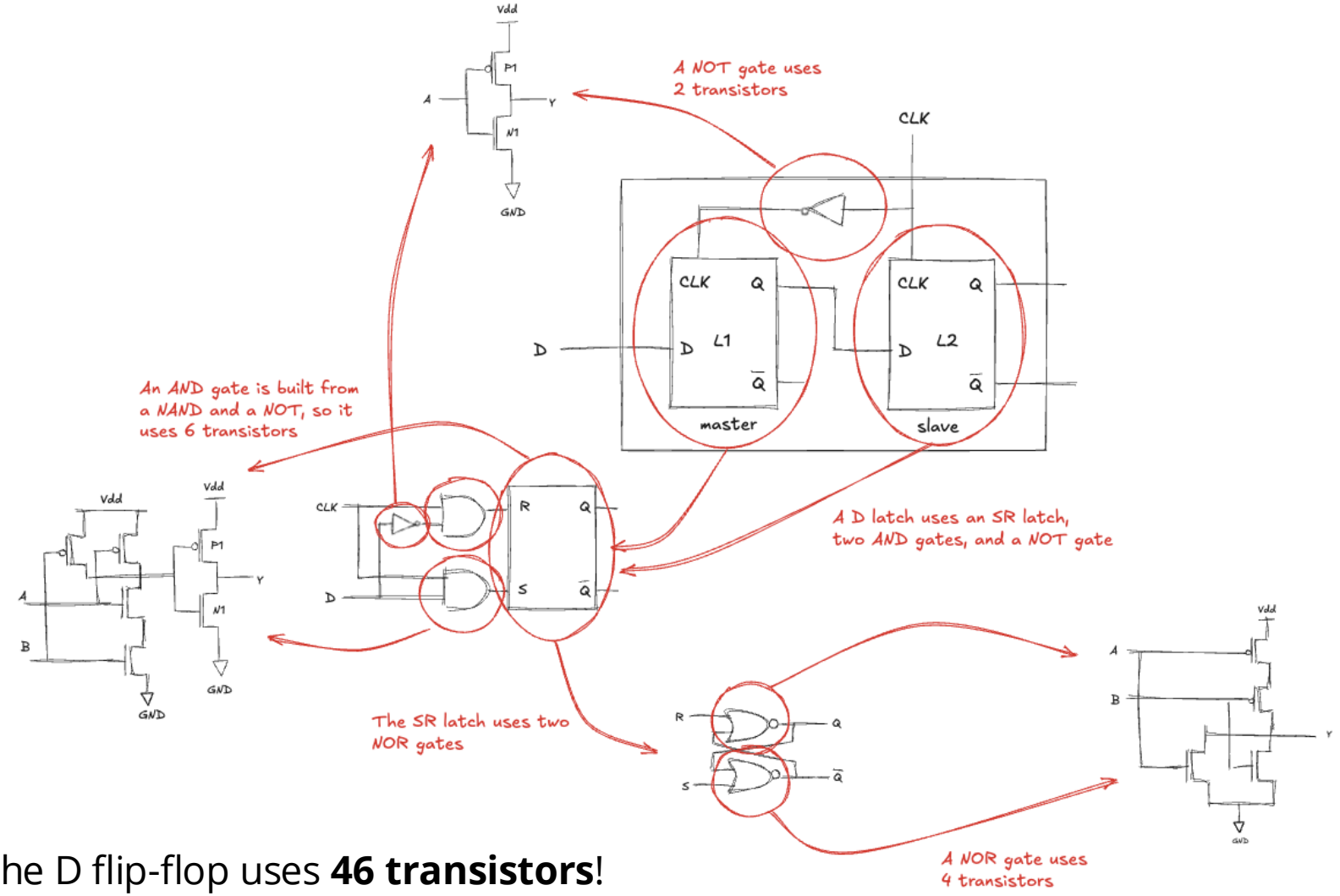
```
architecture rtl of DFF_reset_low is
    signal D_int : std_logic;
begin
    -- Active-low reset implemented via AND gate
    D_int <= D and RESET;

    process(CLK)
    begin
        if rising_edge(CLK) then
            Q <= D_int;
        end if;
    end process;

end architecture rtl;
```

Transistor-Level Design (1)

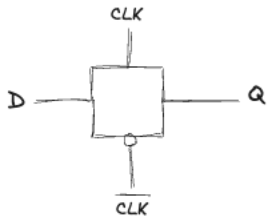
- How many transistors are needed to build the D flip-flop?



- The D flip-flop uses **46 transistors**!

Transistor-Level Design (2)

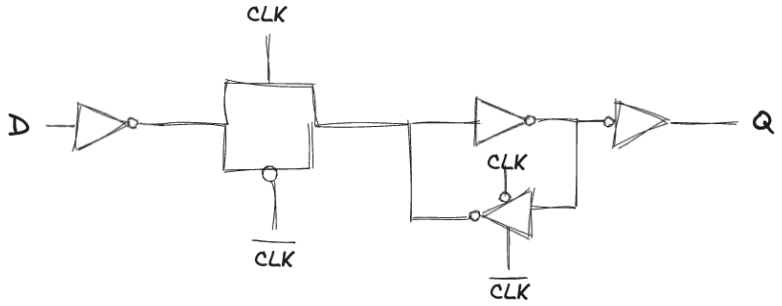
- Many transistors when built from logic gates!
- Its fundamental role is to be **transparent or opaque**, much **like a switch**
- A compact D latch can be constructed from **a single transmission gate**
  - when CLK=1, the transmission gate is ON, so D flows to Q and the latch is transparent
  - when CLK=0, the transmission gate is OFF, so Q is isolated from D and the latch is opaque



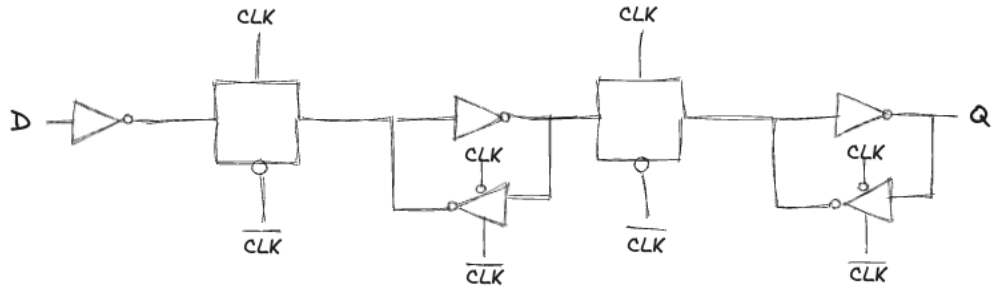
- Two major limitations:
  - **floating node**: when the latch is opaque, Q is not held at its value by any gates, after some time, noise and charge leakage may disturb the value of Q
  - **no buffers**
    - a spike of noise that pulls D to a negative voltage can turn on the nMOS transistor, making the latch transparent, even when CLK=0;
    - likewise, a spike on D above VDD can turn on the pMOS transistor even when CLK=0
    - neither the input of a transmission gate nor the state node of a sequential circuit **should ever be exposed to the outside world** where noise is likely

Transistor-Level Design (3)

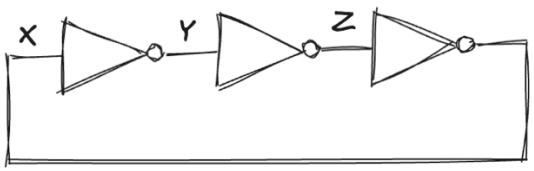
- A more robust (12-transistor) D latch used on modern commercial chips
  - still built around a clocked transmission gate
  - adds inverters to buffer the input and output
  - however, it is outside the scope of this course...



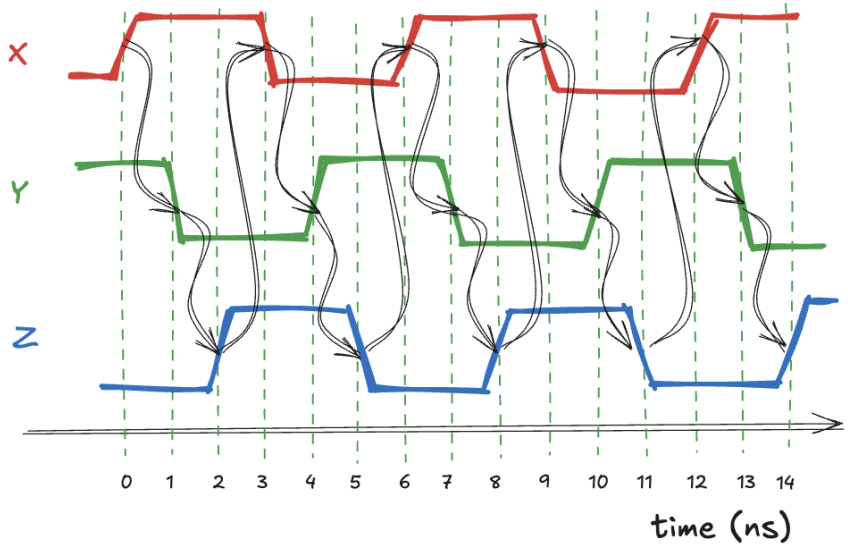
- So, a D flip-flop requires only 20 transistors:



Problematic circuit: astable circuits



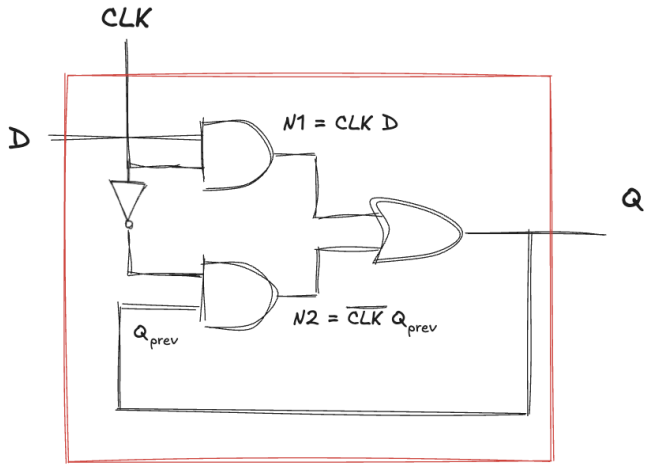
- Consider the three inverters tied in a loop
  - each inverter has a delay of 1ns
- Suppose X is initially 0, then Y=1, Z=0, and, hence, X=1
  - which is inconsistent with original assumption
  - the circuit has **no stable states** and is said to be **unstable** or **astable**
- The period of the oscillator depends on the propagation delay
  - if X rises at t=0, Y falls at 1ns, Z rises at 2ns, and X falls at 3ns
  - in turn, Y rises at 4ns, Z falls at 5ns, and X rises again at 6ns
  - then, the pattern will repeat: each node oscillates with a period of 6 ns
  - this circuit is called a **ring oscillator**



- The gate delay depends on how the inve supply voltage, and even the temperatur
  - the ring oscillator period is difficult to accurately predict

Problematic circuit: race condition (1)

- Consider the following circuits, it seems a good implementation of a D latch (it use fewer gates):

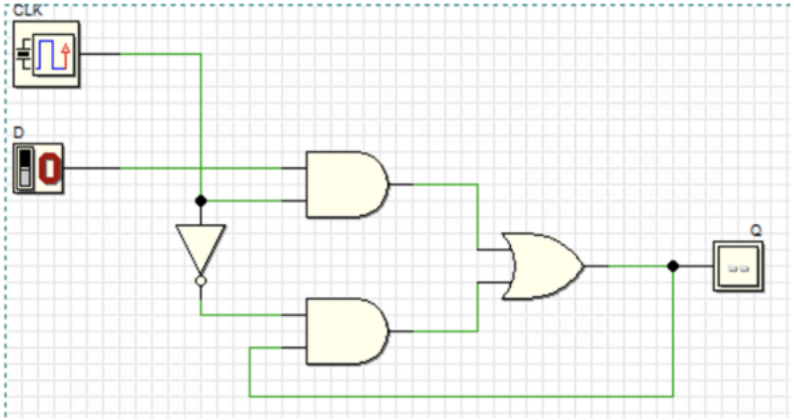


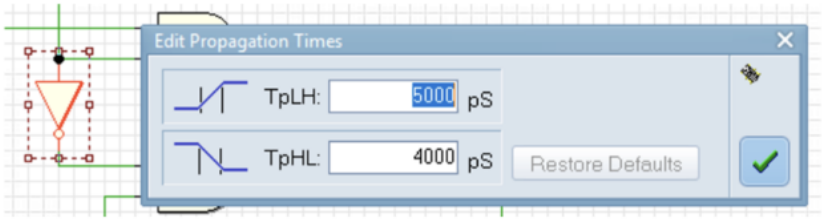
$$= CLK\ D + \overline{CLK}\ Q_{prev}$$

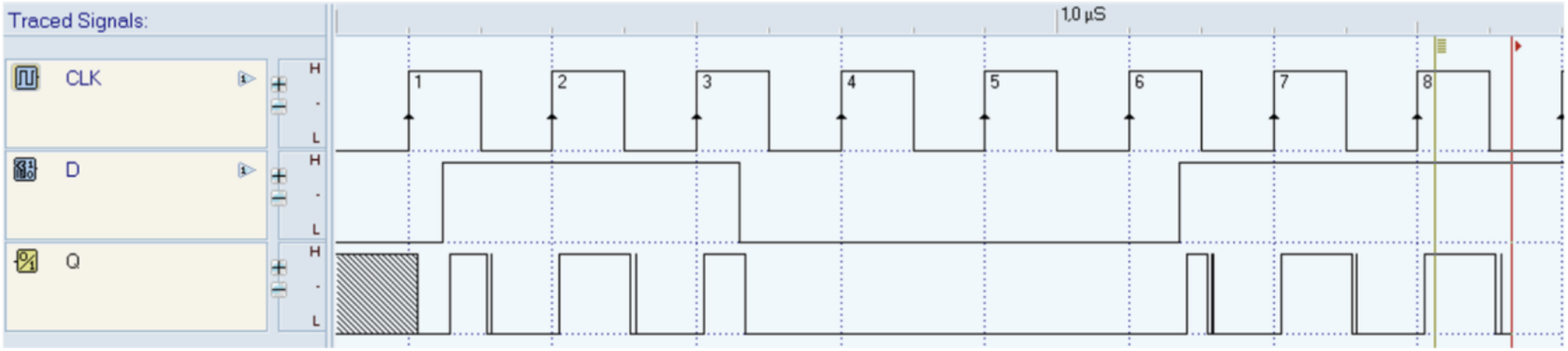
CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

- The circuit has a **race condition** that causes it to fail when certain gates are slower than others:
  - suppose that CLK=D=1, the latch is transparent and passes D through to make Q=1
  - now, CLK falls, the latch should remember its old value, keeping Q=1
  - however, if the **delay** through the inverter is longer than delays of AND and OR
    - N1 and Q may both fall before CLK rises
    - N2 will never rise, and Q becomes stuck at 0

Problematic circuit: race condition (2)







Problematic circuit: asynchronous circuits

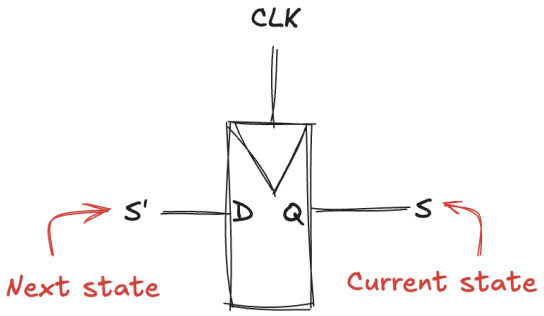
- The previous examples contain **loops** called **cyclic paths**
  - outputs are directly fed back to inputs
- The behaviour depends on which of the paths through logic gates is fastest
  - **asynchronous circuits**
  - a circuit works, an identical one built of gates with different delays may not work
  - a circuit works only at certain temperatures at which the delays are just right
  - these malfunctions are **extremely difficult to track down**
- We **break the cyclic paths** by **inserting registers**
  - transforms the circuit into a **collection of combinational logic and registers**
  - registers contain the **state of the system**, which **changes only at the clock edge**
  - the state is **synchronized** to the clock
  - if the clock is **sufficiently slow** all **races are eliminated**
    - inputs to all registers settle before the next clock edge

Synchronous circuits (1)

- A **synchronous sequential circuit** has a **clock input**, whose rising edges indicate a sequence of times at which state transitions occur
  - **current state** and **next state** distinguish the state of the system at the present from the state to which it will enter on the next clock edge

- **Rules of synchronous sequential circuit composition**
  - a circuit is a synchronous if it consists of interconnected elements, such that
    - every element is either a register or a combinational circuit
    - at least one circuit element is a register
    - all registers receive the same clock signal
    - every cyclic path contains at least one register

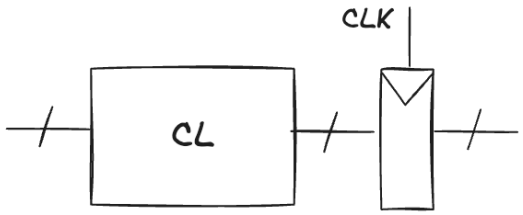
- A flip-flop is the simplest synchronous sequential circuit
  - it has one input, one clock, one output, and two states
  - the next state is D and the output Q is the current state



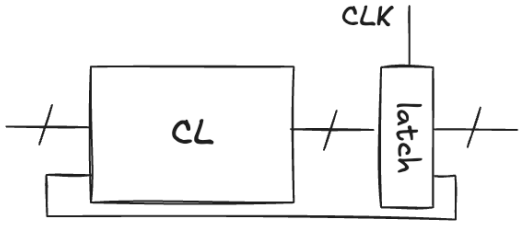
Synchronous circuits (2)

- Which of the following circuits are synchronous sequential circuits?

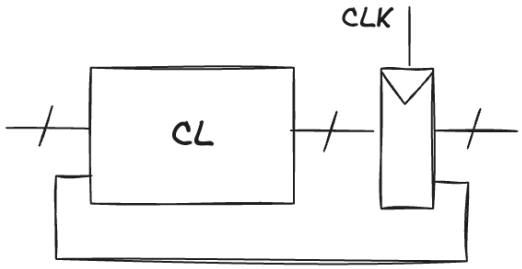
combinational
it has no registers



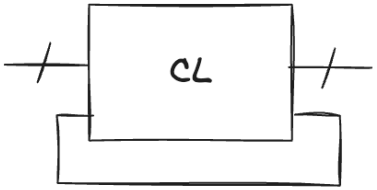
sequential
no feedback



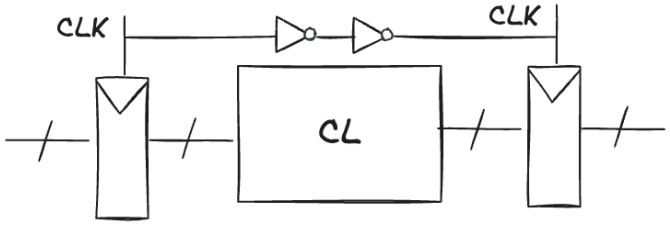
no combinational, no synchronous
it has a latch that is
neither a register
nor a combinational circuit



synchronous sequential

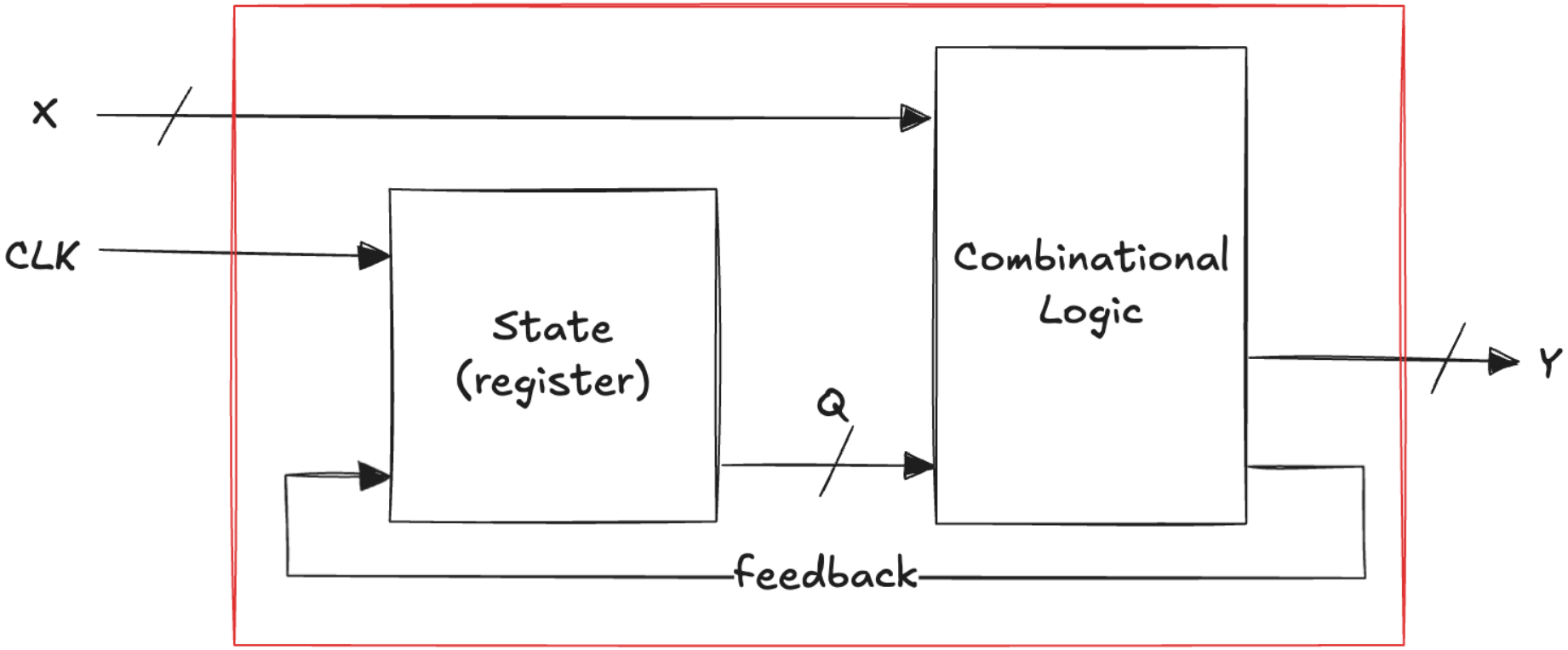


no combinational, no synchronous
it has a cyclic path
but no register in the path



mo synchronous sequential
the second register receives a different clock
signal than the first
delayed by two inverter delays

Synchronous circuits (3)



Synchronous circuits VHDL

```
entity SyncCircuit is
    generic(N : integer := 4   -- number of state bits);
    port(CLK : in  std_logic;
         X   : in  std_logic_vector(N-1 downto 0);  -- external inputs
         Y   : out std_logic_vector(N-1 downto 0)   -- outputs
    );
end SyncCircuit
```

```
architecture Behavioral of SyncCircuit is

    -- State register
    signal Q       : std_logic_vector(N-1 downto 0); -- current state
    signal Q_next  : std_logic_vector(N-1 downto 0); -- next state

begin

    -- 1. COMBINATIONAL LOGIC:
    comb_logic : process(X, Q)
    begin
        -- default assignments (mandatory for combinational logic)
        Q_next <= Q;
        Y      <= (others => '0');

        -- INSERT COMBINATIONAL BEHAVIOR HERE
        --   Q_next <= some_function_of(X, Q);
        --   Y      <= some_other_function_of(X, Q);
    end process;

    -- 2. SEQUENTIAL PROCESS:
    state_register : process(CLK)
    begin
        if rising_edge(CLK) then
            Q <= Q_next;
        end if;\
    end process;

end Behavioral;
```

Synchronous vs Asynchronous Circuits

- Asynchronous design is **more general** than synchronous design
  - the timing of the system is not limited by clocked registers
  - (just as analog circuits are more general than digital circuits)

- Synchronous circuits are **easier to design** than asynchronous circuits
  - (just as digital are easier than analog circuits)

- Despite decades of research on asynchronous circuits, virtually all digital systems are essentially synchronous

- Asynchronous circuits are **occasionally necessary** when communicating between **systems with different clocks** or when **receiving inputs at arbitrary times**
  - (just as analog circuits are necessary when communicating with the real world of continuous voltages)