EEL7020 – Sistemas Digitais
Aula 6: Circuitos sequenciais - Flip-flops

Prof. Djones Vinicius Lettnin

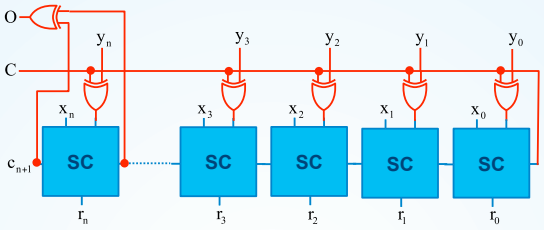
lettnin@eel.ufsc.br

http://lettnin.paginas.ufsc.br/

Disclaimer: slides adapted for EEL7020 by D. Lettnin from the original slides made available by the author E. Batista, F. Vahid.

Circuitos Aritméticos
Overflow

- Somador/Subtrator Paralelo com Detecção de Overflow:



Se $C = 0 \rightarrow R = X + Y$

Se $C = 1 \rightarrow R = X - Y$

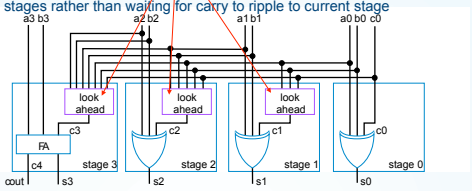
Se $O = 1 \rightarrow$ ocorreu overflow

Plano de Aula

- **Somadores**
- Flip-flops
  - Visão geral
  - NAND, NOR
  - FF-D, FF-T, FF-JK

Faster Adder – (Bad) Attempt at "Lookahead"

- Idea
  - Modify carry-ripple adder – For a stage's carry-in, don't wait for carry to ripple, but rather *directly compute* from inputs of earlier stages
    - Called "lookahead" because current stage "looks ahead" at previous stages rather than waiting for carry to ripple to current stage



Implementação de Somadores

CLA

Somador paralelo "*CARRY LOOKAHEAD*"

A	*B*	C_{in}	*S*	C_{out}	*Carry status*
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

A = B = 0
Remoção de *carry*
Se A = B = 0, então C_{out} será sempre 0, sem levar em consideração o C_{in}.

O FA em questão irá ignorar o *carry* recebido do FA do estágio anterior, "removendo" o *carry* recebido do restante do circuito.

Implementação de Somadores

CLA

Somador paralelo "*CARRY LOOKAHEAD*"

A	*B*	C_{in}	*S*	C_{out}	*Carry status*
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

A = B = 1
Geração de *carry*
Se A = B = 1, então C_{out} será sempre 1, sem levar em consideração o C_{in}.

O FA em questão irá ignorar o *carry* recebido do FA do estágio anterior, gerando sempre *carry* "1" para o próximo estágio.

Implementação de Somadores

CLA

Universidade Federal de Santa Catarina

Somador paralelo "*CARRY LOOKAHEAD*"

A	*B*	C_{in}	*S*	C_{out}	*Carry status*
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

A* ≠ *B
Propagação de *carry*

Se A ≠ B, então o FA em questão irá propagar o *carry* recebido do FA do estágio anterior, gerando sempre *carry* "1" para o próximo estágio.

Implementação de Somadores

CLA

Universidade Federal de Santa Catarina

Somador paralelo "*CARRY LOOKAHEAD*"

A	*B*	C_{in}	*S*	C_{out}	*Carry status*
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

Funções para o propagate e generate

Se $A_i \neq B_i$, ou seja, se A_i xor B_i for verdadeiro, o C_{in} é "propagado" para o C_{out}:

$P_i = A_i$ xor B_i

Implementação de Somadores

CLA

Universidade Federal de Santa Catarina

Somador paralelo "*CARRY LOOKAHEAD*"

A	*B*	C_{in}	*S*	C_{out}	*Carry status*
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

Funções para o propagate e generate

Se $A_i \neq B_i$, ou seja, se A_i xor B_i for verdadeiro, o C_{in} é "propagado" para o C_{out}:

$P_i = A_i$ xor B_i

Se $A_i = B_i$, então um carry será gerado no C_{out}:

$G_i = A_i$ and B_i

Implementação de Somadores

CLA

Universidade Federal de Santa Catarina

Somador paralelo "*CARRY LOOKAHEAD*"

Soma e *carry* podem ser obtidos utilizando as funções *propagate* e *generate*.

Assim, se $\mathbf{P_i = A_i}$ **xor** $\mathbf{B_i}$ e $\mathbf{G_i = A_i}$ **and** $\mathbf{B_i}$, logo:

$S_i = A_i$ xor B_i xor C_i => $\mathbf{S_i = P_i}$ **xor** $\mathbf{C_i}$

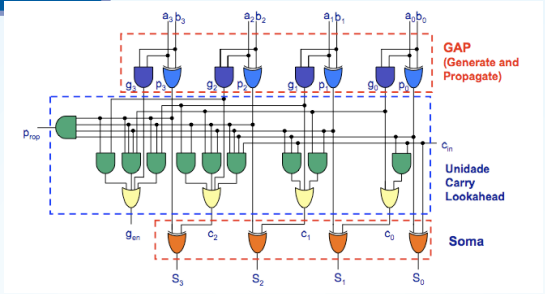
$C_{i+1} = A_iB_i$ or $C_i(A_i$ xor $B_i)$ => $\mathbf{C_{i+1} = G_i}$ **or** $\mathbf{C_iP_i}$

Para um somador de 4 bits, o *carry out* pode ser obtido da seguinte forma:

$C_1 = G_0$ or P_0C_0
$C_2 = G_1$ or $P_1C_1 = G_1$ or P_1G_0 or $P_1P_0C_0$
$C_3 = G_2$ or $P_2C_2 = G_2$ or P_2G_1 or $P_2P_1G_0$ or $P_2P_1P_0C_0$
$C_4 = G_3$ or $P_3C_3 = G_3$ or P_3G_2 or $P_3P_2G_1$ or $P_3P_2P_1G_0$ or $P_3P_2P_1P_0C_0$

Somador paralelo "*CARRY LOOKAHEAD*"

Universidade Federal de Santa Catarina



Implementação de Somadores

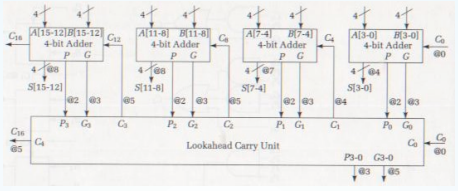
CLA

ANÁLISE

Universidade Federal de Santa Catarina

Somador paralelo "*CARRY LOOKAHEAD*"

- Na prática, somadores *carry lookahead* são utilizados para símbolos até 4 bits.
- Para símbolos maiores do que 4 bits, a complexidade da lógica de geração dos *carries* torna esse somador mais lento do que o *ripple carry*.
- Abordagem hierárquica – somador de 16 bits construído a partir de CLAs de 4 bits:



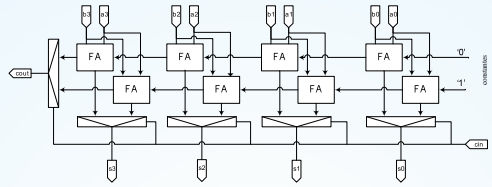
Fonte: Katz 1992.

Implementação de Somadores

CSA

Somador paralelo "*CARRY SELECT*"

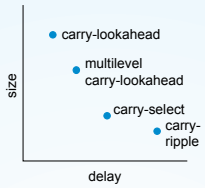
- Duplica o hardware do somador (FA) para cada bit para os dois valores possíveis de *carry*



- O *Cout* do estágio anterior (Cin na figura acima) é utilizado para selecionar nos multiplexadores a soma correta (S_3 .. S_0) e o Cout a ser enviado para o próximo estágio

13

Adder Tradeoffs



- Designer picks the adder that satisfies particular delay and size requirements
  - May use different adder types in different parts of same design
    - Faster adders on critical path, smaller adders on non-critical path

1

Plano de Aula

- Somadores
- Flip-flops
  - **Visão geral**
  - NAND, NOR
  - FF-D, FF-T, FF-JK

15

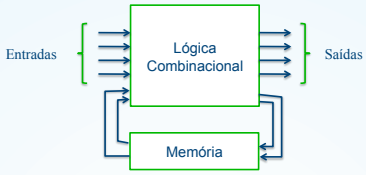
Lógica Seqüencial

- Combinacional x Seqüencial

16

Lógica Seqüencial

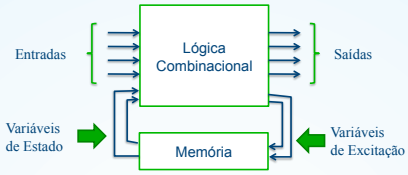
- Combinacional x Seqüencial



17

Lógica Seqüencial

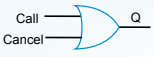
- Combinacional x Seqüencial



18

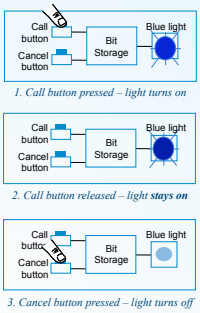
Example Needing Bit Storage

- Flight attendant call button
  - Press call: light turns on
    - ***Stays on*** after button released
  - Press cancel: light turns off
  - Logic gate circuit to implement this?



Doesn't work. Q=1 when Call=1, but doesn't stay 1 when Call returns to 0

Need some form of "feedback" in the circuit



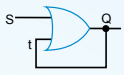
1. Call button pressed – light turns on

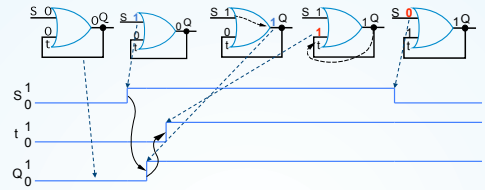
*2. Call button released – light **stays on***

3. Cancel button pressed – light turns off

First attempt at Bit Storage

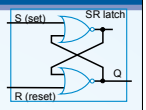
- We need some sort of feedback
  - Does circuit on the right do what we want?
    - No: Once Q becomes 1 (when S=1), Q stays 1 forever – no value of S can bring Q back to 0

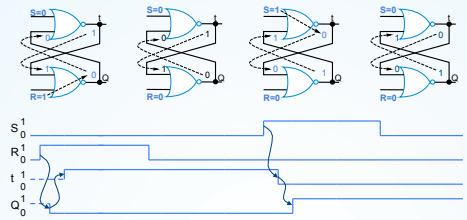




Bit Storage Using an SR Latch

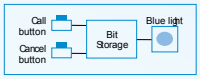
- Does the circuit to the right, with cross-coupled NOR gates, do what we want?
  - Yes! How did someone come up with that circuit? Maybe just trial and error, a bit of insight...

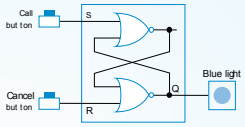




Example Using SR Latch for Bit Storage

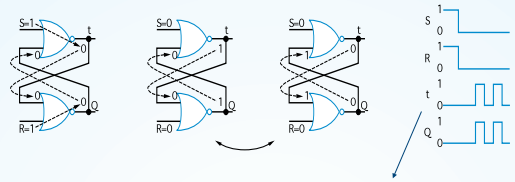
- SR latch can serve as bit storage in previous example of flight-attendant call button
  - Call=1 : sets Q to 1
    - Q stays 1 even after Call=0
  - Cancel=1 : resets Q to 0
- But, there's a problem...





Problem with SR Latch

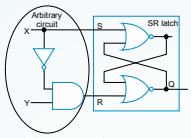
- Problem
  - If S=1 and R=1 simultaneously, we don't know what value Q will take



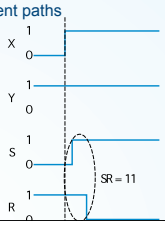
Q may oscillate. Then, because one path will be slightly longer than the other, Q will eventually settle to 1 or 0 – but we don't know which.

Problem with SR Latch

- Problem not just one of a user pressing two buttons at same time
- Can also occur even if SR inputs come from a circuit that supposedly never sets S=1 and R=1 at same time
  - But does, due to different delays of different paths

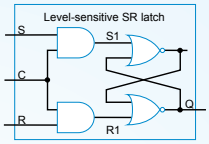


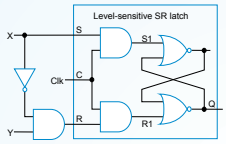
The longer path from X to R than to S causes SR=11 for short time – could be long enough to cause oscillation



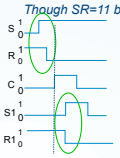
Solution: Level-Sensitive SR Latch

- Add enable input "C" as shown
  - Only let S and R change when C=0
    - Enure circuit in front of SR never sets SR=11, except briefly due to path delays
  - Change C to 1 only after sufficient time for S and R to be stable
  - When C becomes 1, the stable S and R value passes through the two AND gates to the SR latch's S1 R1 inputs.

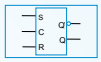




Though SR=11 briefly...

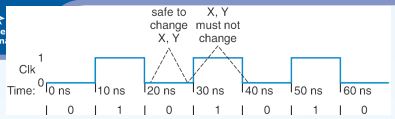


...S1R1 never = 11

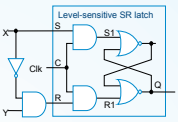


Level-sensitive SR latch symbol

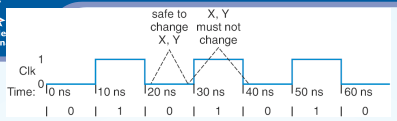
Clock Signals for a Latch



- How do we know when it's safe to set C=1?
  - Most common solution –make C pulse up/down
    - C=0: Safe to change X, Y
    - C=1: Must *not* change X, Y
    - We'll see how to ensure that later
  - ***Clock*** signal -- Pulsing signal used to enable latches
    - Because it ticks like a clock
  - Sequential circuit whose storage components all use clock signals: ***synchronous*** circuit
    - Most common type
    - Asynchronous circuits – important topic, but left for advanced course



Clocks

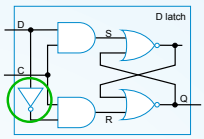


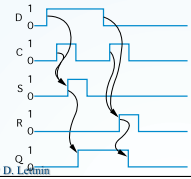
- ***Clock period***: time interval between pulses
  - Above signal: period = 20 ns
- ***Clock cycle***: one such time interval
  - Above signal shows 3.5 clock cycles
- ***Clock frequency***: 1/period
  - Above signal: frequency = 1 / 20 ns = 50 MHz
    - 1 Hz = 1/s

Freq	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	1 ns
100 MHz	10 ns
10 MHz	100 ns

Level-Sensitive D Latch

- SR latch requires careful design to ensure SR=11 never occurs
- D latch relieves designer of that burden
  - Inserted inverter ensures R always opposite of S

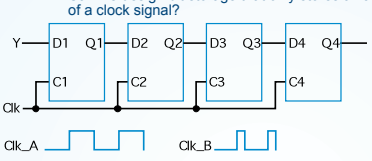




D latch symbol

Problem with Level-Sensitive D Latch

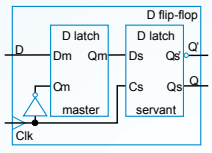
- D latch still has problem (as does SR latch)
  - When C=1, through how many latches will a signal travel?
  - Depends on for how long C=1
    - Clk_A -- signal may travel through multiple latches
    - Clk_B -- signal may travel through fewer latches
  - Hard to pick C that is just the right length
    - Can we design bit storage that only stores a value on the rising edge of a clock signal?

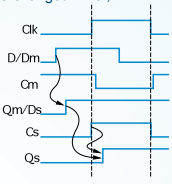


D Flip-Flop

- ***Flip-flop***: Bit storage that stores on clock edge, not level
- One design -- master-servant
  - Two latches, output of first goes to input of second, master latch has inverted clock signal
  - So master loaded when C=0, then servant when C=1
  - When C changes from 0 to 1, master disabled, servant loaded with value that was at D just before C changed -- i.e., value at D during rising edge of C

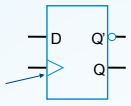
Note: Hundreds of different flip-flop designs exist



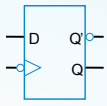


D Flip-Flop

The triangle means clock input, edge triggered



Symbol for rising-edge triggered D flip-flop

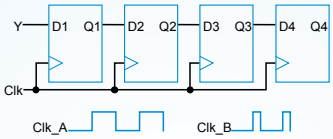


Symbol for falling-edge triggered D flip-flop

Internal design: Just invert servant clock rather than master

D Flip-Flop

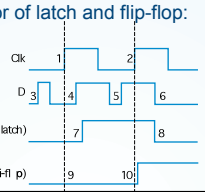
- Solves problem of not knowing through how many latches a signal travels when C=1
  - In figure below, signal travels through exactly one flip-flop, for Clk_A or Clk_B
  - Why? Because on rising edge of Clk, all four flip-flops are loaded simultaneously -- then all four no longer pay attention to their input, until the next rising edge. Doesn't matter how long Clk is 1.



Two latches inside each flip-flop

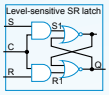
D Latch vs. D Flip-Flop

- Latch is level-sensitive: Stores D when C=1
- Flip-flop is edge triggered: Stores D when C changes from 0 to 1
  - Saying "level-sensitive latch," or "edge-triggered flip-flop," is redundant
  - Two types of flip-flops -- rising or falling edge triggered.
- Comparing behavior of latch and flip-flop:

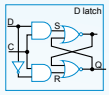


Bit Storage Summary

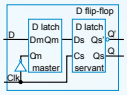
Feature: S=1 sets Q to 1, R=1 resets Q to 0. Problem: SR=11 yield undefined Q.



Feature: S and R only have effect when C=1. We can design outside circuit so SR=11 never happens when C=1. Problem: avoiding SR=11 can be a burden.



Feature: SR can't be 11 if D is stable before and while C=1, and will be 11 for only a brief glitch even if D changes while C=1. Problem: C=1 too long propagates new values through too many latches: too short may not enable a store.



Feature: Only loads D value present at rising clock edge, so values can't propagate to other flip-flops during same clock cycle. Tradeoff: uses more gates internally than D latch, and requires more external gates than SR – but gate count is less of an issue today.

- We considered increasingly better bit storage until we arrived at the robust D flip-flop bit storage

Plano de Aula

- Somadores
- Flip-flops
  - Visão geral
  - **NAND, NOR**
  - FF-D, FF-T, FF-JK

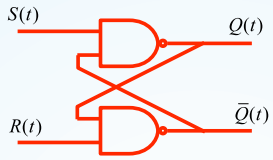
Lógica Seqüencial

Latches e Flip-flops

- Circuitos seqüenciais elementares
- Têm capacidade de armazenar informação
- Unidade elementar de memória => **1 bit**
- **Latch** – assíncrono
- **Flip-flop** – versão síncrona do **latch**

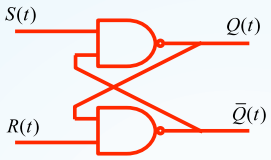
Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NAND:**



Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NAND:**



$Q(t)$, $\bar{Q}(t)$ — Complementares

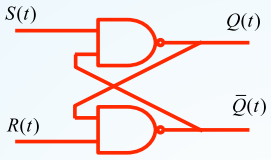
$S(t)$, $R(t)$ — Entradas atuais "Excitação"

$Q(t)$ ➡ Saída atual "estado"

$Q(t+1)$ ➡ Próximo estado

Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NAND:**

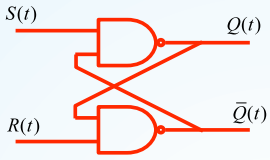


A) Tabela de Transição:

$S(t)$	$R(t)$	$Q(t)$	$Q(t+1)$	$\bar{Q}(t+1)$
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NAND:**

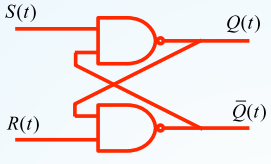


A) Tabela de Transição:

$S(t)$	$R(t)$	$Q(t)$	$Q(t+1)$	$\bar{Q}(t+1)$
0	0	0	1	1
0	0	1	1	1
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NAND:**



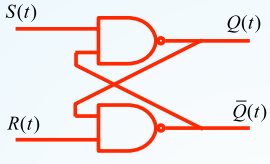
Proibidos

A) Tabela de Transição:

$S(t)$	$R(t)$	$Q(t)$	$Q(t+1)$	$\bar{Q}(t+1)$
0	0	0	1	1
0	0	1	1	1
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NAND:**

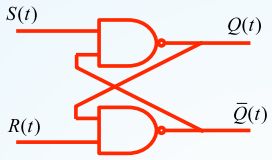


B) Tabela de Função:

$S(t)$	$R(t)$	$Q(t+1)$
0	0	
0	1	
1	0	
1	1	

Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NAND:**

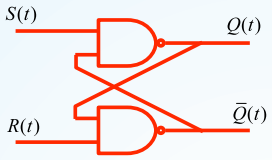


B) Tabela de Função:

$S(t)$	$R(t)$	$Q(t+1)$
0	0	Proibido
0	1	1 (Set)
1	0	0 (Reset)
1	1	

Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NAND:**

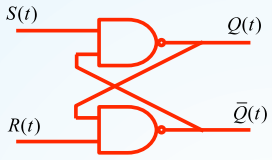


B) Tabela de Função:

$S(t)$	$R(t)$	$Q(t+1)$
0	0	Proibido
0	1	1 (Set)
1	0	0 (Reset)
1	1	$Q(t)$

Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NAND:**

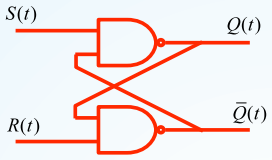


C) Tabela de Excitação:

$Q(t) \rightarrow Q(t+1)$		$S(t)$	$R(t)$
0	0		
0	1		
1	0		
1	1		

Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NAND:**

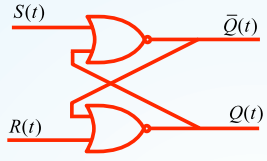


C) Tabela de Excitação:

$Q(t) \rightarrow Q(t+1)$		$S(t)$	$R(t)$
0	0	1	X
0	1	0	1
1	0	1	0
1	1	X	1

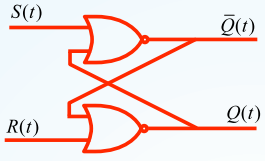
Lógica Seqüencial
Latch RS (Reset-Set)

- **Latch com portas NOR:**



Lógica Seqüencial
Latch RS (Reset-Set)

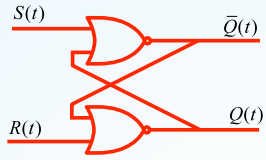
- **Latch com portas NOR:**



$S(t)$	$R(t)$	$Q(t+1)$
0	0	
0	1	
1	0	
1	1	

Lógica Seqüencial
Latch RS (Reset-Set)

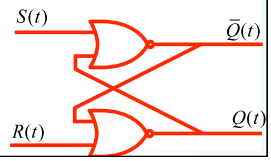
- **Latch com portas NOR:**



$S(t)$	$R(t)$	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	Proibido

Latches SR em VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity SR is
  port (S, R : in std_logic;
        Q : out std_logic_vector(1 downto 0)
       );
end SR;
architecture behavior of SRis
  signal A, B: std_logic;
begin
  A <= not (B or R);
  B <= not (A or S);
  Q(1) <= A;
  Q(0) <= B;
end behavior
```



Flip-Flops em VHDL

sensitivity list

```
process (A, B, C, D)
  begin
        A <= '1';
        B <= '1';
        B <= D;
        A <= not B;
        C <= A and '1';
        D <= C;
end process;
```

Sequêncial

Process

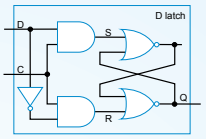
- **Os valores atribuídos aos sinais só são válidos após a execução do último comando(após "sair" do processo).**
- **Se existem várias atribuições a um mesmo sinal APENAS a última atribuição é válida na saída**

  ⇒ ex. sinal A no corpo do processo.

É permitido declarar

Latch D em VHDL

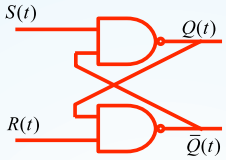
D	C	Q^{n+1}
0	1	0
1	1	1
X	0	Q^n
X	0	Q^n




```
library ieee;
use ieee.std_logic_1164.all;
entity D_latch is
  port (
        C: in std_logic;
        D: in std_logic;
        Q: out std_logic_vector(0 downto 1));
end D_latch;
architecture behv of D_latch is
begin
  process(C, D)
        begin
          if (C = '1') then
            Q(1) <= D;
            Q(0) <= not(D);
          end if;
  end process;
end behv
```

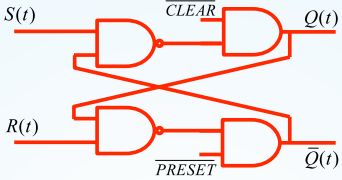

Lógica Seqüencial
Clock e Preset/Clear

- **Preset/Clear:**



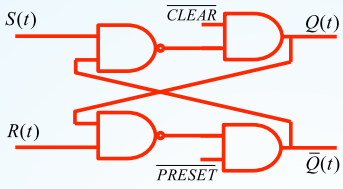
Lógica Seqüencial
Clock e Preset/Clear

- **Preset/Clear:**



Lógica Seqüencial
Clock e Preset/Clear

- **Preset/Clear:**



PRESET	CLEAR	$Q(t+1)$
0	0	Proibido
0	1	1
1	0	0
1	1	$Q(t)$

Plano de Aula

- Somadores
- Flip-flops
  - Visão geral
  - NAND, NOR
  - **FF-D, FF-T, FF-JK**

Lógica Seqüencial
Flip-Flop D (Data/Delay)

Função

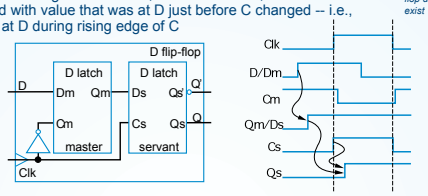
$D(t)$	$Q(t+1)$
0	0
1	1

CLK	$D(t)$	$Q(t+1)$
0	X	$Q(t)$
↑	$D(t)$	$D(t)$
1	X	$Q(t)$

D Flip-Flop: Mestre-Escravo

- ***Flip-flop***: Bit storage that stores on clock edge, not level
- One design -- master-servant
  - Two latches, output of first goes to input of second, master latch has inverted clock signal
  - So master loaded when C=0, then servant when C=1
  - When C changes from 0 to 1, master disabled, servant loaded with value that was at D just before C changed -- i.e., value at D during rising edge of C

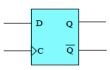
Note: Hundreds of different flip-flop designs exist



D Flip-Flop: Edge Triggered Flip-Flop

O Flip-flop D disparado pela borda ascendente
(ou Flip-flop D sensível à borda ascendente)

símbolo



circuito com portas nand

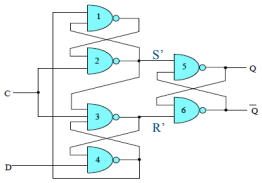
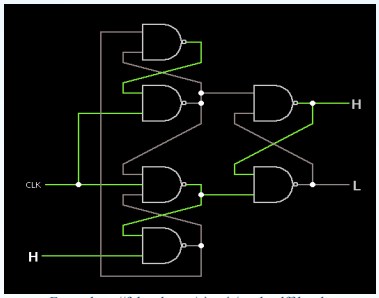


tabela de transição de estados

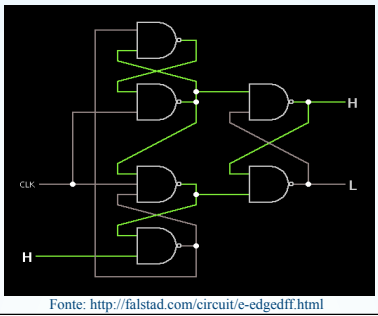
C	D	Q_{t+1}
↑	X	Q_t
↑	0	0
↑	1	1

D Flip-Flop: Edge Triggered Flip-Flop



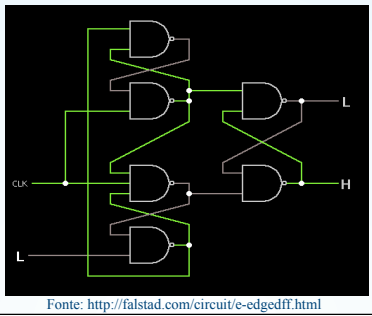
Fonte: http://falstad.com/circuit/e-edgedff.html

D Flip-Flop: Edge Triggered Flip-Flop



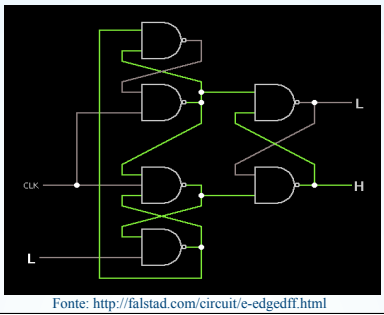
Fonte: http://falstad.com/circuit/e-edgedff.html

D Flip-Flop: Edge Triggered Flip-Flop



Fonte: http://falstad.com/circuit/e-edgedff.html

D Flip-Flop: Edge Triggered Flip-Flop



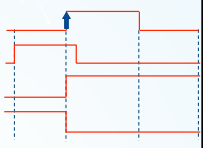
Fonte: http://falstad.com/circuit/e-edgedff.html

Flip-Flop D em VHDL

D	C	Q^{n+1}
0	↑	0
1	↑	1
X	0	Q^n
X	1	Q^n

```
library ieee;
use ieee.std_logic_1164.all;
entity FFD is
  port (
        C:  in std_logic;
        D:  in std_logic;
        Q: out std_logic_vector(0 downto 1));
end FFD;
architecture behv of FFD is
begin
  process(C, D)
       begin
          if (C'event  and C = '1') then
             Q(1) <= D;
             Q(0) <= not(D);
          end if;
  end process;
end behv
```

Não tem que manter D na duração de relógio C



Lógica Seqüencial

Flip-Flop T (Toggle – Comutação)

Lógica Seqüencial

Flip-Flop T (Toggle – Comutação)

Função

$T(t)$	$Q(t+1)$
0	$Q(t)$
1	$\bar{Q}(t)$

CLK	$T(t)$	$Q(t+1)$
0	X	$Q(t)$
↑	0	$Q(t)$
↑	1	$\bar{Q}(t)$

Lógica Seqüencial
Flip-Flop T (Toggle – Comutação)

Tabela de Excitação:

$Q(t) \rightarrow Q(t+1)$		$T(t)$
0	0	
0	1	
1	0	
1	1	

Lógica Seqüencial
Flip-Flop T (Toggle – Comutação)

Tabela de Excitação:

$Q(t) \rightarrow Q(t+1)$		$T(t)$
0	0	0
0	1	1
1	0	1
1	1	0

Lógica Seqüencial
Flip-Flop JK

$J(t)$	$K(t)$	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\bar{Q}(t)$

Lógica Seqüencial
Flip-Flop JK

$J(t)$	$K(t)$	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\bar{Q}(t)$

Tabela de Excitação:

$Q(t) \rightarrow Q(t+1)$		$J(t)$	$K(t)$
0	0		
0	1		
1	0		
1	1		

Lógica Seqüencial
Flip-Flop JK

$J(t)$	$K(t)$	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\bar{Q}(t)$

Tabela de Excitação:

$Q(t) \rightarrow Q(t+1)$		$J(t)$	$K(t)$
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

EEL7020 – Sistemas Digitais
Aula 6: Circuitos sequenciais - Flip-flops

Prof. Djones Vinicius Lettnin

lettnin@eel.ufsc.br

http://lettnin.paginas.ufsc.br/

Disclaimer: slides adapted for EEL7020 by D. Lettnin from the original slides made available by the author E. Batista, F. Vahid.