



**UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA**

INTRODUÇÃO À MICROELETRÔNICA 2023.1

GERADOR DE NÚMEROS PSEUDO-ALEATÓRIOS COM LFSR

Giovanni Bruno Travassos de Carvalho - 11506849
Rafael de Melo Oliveira - 20200013481

Professor: Hugo Leonardo Davi de Souza Cavalcanti

Novembro, 2023

Sumário:

1. Introdução.....	2
2. Metodologia.....	2
3. Resultado.....	5
4. Referencias:.....	8

1. Introdução

O objetivo desse relatório é mostrar o processo de desenvolvimento do projeto final da disciplina de Introdução à Microeletrônica no semestre 2023.1, que busca construir um gerador de números pseudo-aleatórios (RNG, *random number generator*) de 8 bits com realimentação linear. O trabalho começou com uma descrição comportamental em alto nível e uma geração de padrões de teste e passou por vários passos, que serão especificados, até chegar em um layout físico e uma simulação precisa da operação de geração dos números aleatórios. Todos os arquivos obtidos estarão anexados a esse relatório.

2. Metodologia

Para criação do gerador de números pseudo-aleatórios precisamos construir um registrador de deslocamento com realimentação linear (LFSR, *linear feedback shift register*), utilizando a função XOR. O LFSR é um registrador de deslocamento cujo bit de entrada é uma função linear de seu estado anterior. O valor inicial do LFSR é chamado de *seed* (semente) e, como a operação do registrador é determinística, o fluxo de valores produzido pelo registrador é completamente determinado pelo seu estado atual (ou anterior). Da mesma forma, como o registrador possui um número finito de estados possíveis, ele deverá eventualmente entrar em um ciclo de repetição. No entanto, um LFSR com uma função de feedback bem escolhida pode produzir uma sequência de bits que parece aleatória e tem um ciclo muito longo.

O polinômio de realimentação usado para um registrador de 8 bits é definido por:

$$x^8 + x^6 + x^5 + x^4 + 1$$

Esse polinômio é utilizado para máxima realimentação linear possível com 8 bits. Seu período é definido por $(2^n - 1)$, onde n é o número de bits, portanto o período é igual a 255, ou seja, o gerador irá criar 255 números pseudo-aleatórios antes de repetir algum número. Esse resultado será provado no final.

O gerador NPA construído funciona basicamente pegando um número binário de 8 bits como entrada e aplicando a função XOR em alguns de seus bits levando em consideração o polinômio escolhido, de maneira que o número não irá se repetir. Esse novo número gerado é usado como entrada de realimentação do mesmo circuito.

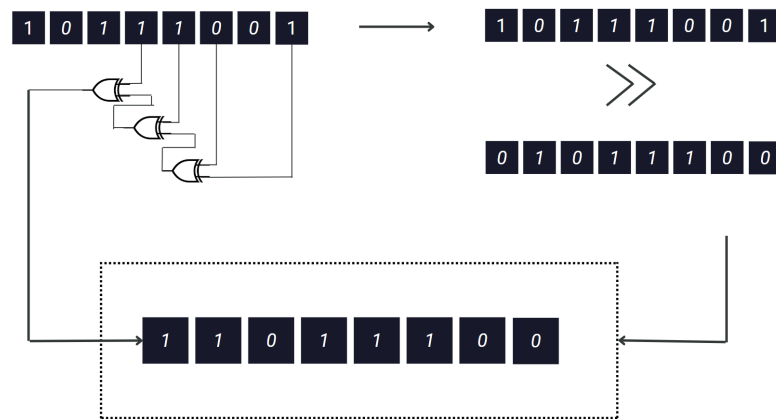


Figura 1: Ilustração do funcionamento do LFSR

Com o método de construção para o gerador estabelecido, antes de passar para o desenvolvimento de uma linguagem de descrição de hardware, primeiro será obtido um padrão de testes para confirmar uma sequência de saídas esperadas pelo circuito. Por meio da biblioteca “*genpat.h*” é possível obter uma simulação de forma menos complexa para conseguir ter um parâmetro comparativo ao se criar o layout físico.

```

1 #include <stdio.h>
2 #include <stdint.h>
3 #include "genpat.h"
4
5 #define LFSR_WIDTH 8
6 #define LFSR_TAP_BIT1 1
7 #define LFSR_TAP_BIT2 3
8 #define LFSR_TAP_BIT3 4
9 #define LFSR_TAP_BIT4 5
10
11 void updateLFSR(uint8_t*state){
12     uint8_t lsb=*state & 1;
13     uint8_t feedback = ((*state>>(LFSR_TAP_BIT1 - 1)) & 1)^
14         ((*state>>(LFSR_TAP_BIT2 - 1)) & 1)^
15         ((*state>>(LFSR_TAP_BIT3 - 1)) & 1)^
16         ((*state>>(LFSR_TAP_BIT4 - 1)) & 1);
17     *state>>=1;
18     if(feedback){
19         *state^=(1<<7); //Bit mais significativo(MSB)
20     }else{
21         *state^=(0<<7); //Bit mais significativo(MSB)
22     }
23 }
24 char *inttostr(int inteiro) {
25     char *str = (char *)malloc(32 * sizeof(char));
26     if (str != NULL) {
27         sprintf(str, "%d", inteiro);
28     }
29     return str;
30 }
31
32 int main(){
33     uint8_t lfsr_state=0xB8;
34     int tempo = 0;
35
36     DEF_GENPAT("lfsr_sim");
37     DECLAR("lfsr_state",":2","B",IN,"7 down to 0","");
38
39     int i;
40     for(i = 0;i<256;i++){
41         updateLFSR(&lfsr_state);
42         AFFECT(inttostr(tempo),"lfsr_state",inttostr(lfsr_state));
43         tempo++;
44     }
45     SAV_GENPAT();
46     return 0;
47 }

```

Figura 2: Código que gera o padrão de testes utilizando o ‘genpat.h’

Após compilar o código por meio do comando “alliance-genpat -v {nome do arquivo}” é gerado o arquivo *.pat* com as saídas processadas sendo possível agora utilizá-las para visualização e método de comparação.

Utilizamos a abordagem *Top-Down* para concepção do circuito, então começamos com uma descrição em VHDL comportamental de alto nível, sem muita preocupação com a implementação, que será convertida para um circuito implementável em VHDL comportamental, e finalmente, criado o layout físico das células padrão.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6 entity LFSR is
7     Port ( clk : in STD_LOGIC;
8           rst : in STD_LOGIC;
9           lfsr_state : in STD_LOGIC_VECTOR(7 downto 0) := "01111111";
10          lfsr_out : out STD_LOGIC_VECTOR(7 downto 0);
11          vdd : in STD_LOGIC;
12          vss : in STD_LOGIC );
13 end LFSR;
14
15
16 architecture Behavioral of LFSR is
17     signal XOR_result : STD_LOGIC;
18     signal clk_borda : bit;
19 begin
20     process (clk, rst)
21     begin
22         if rising_edge(clk) then
23             clk_borda <= NOT clk_borda; -- detecta a borda
24             if clk_borda = '1' then
25                 XOR_result <= lfsr_state(7) xor lfsr_state(5)
26                 xor lfsr_state(4) xor lfsr_state(3);
27                 lfsr_out <= XOR_result & lfsr_state(7 downto 1);
28             end if;
29         end if;
30     end process;
31 end Behavioral;
32

```

Figura 2: geradorNPA.vhdl

Com essa descrição comportamental do circuito feita, utilizamos a ferramenta **vasy** para convertê-la para um conjunto de VHDL implementável pelas demais ferramentas que serão utilizadas (“*.vbe*”).

O arquivo *.vbe* obtido foi otimizado através da ferramenta **boom**, que otimiza uma descrição comportamental usando uma representação RBDD (*Reduced Ordered Binary Decision Diagram*) de sua função lógica. Além disso, provamos a equivalência entre os arquivos obtidos com a ferramenta **proof**.

```

===== Files, Options and Parameters =====
First VHDL file      = gerador5.vbe
Second VHDL file     = gerador_5o.vbe
The common auxiliary signals are kept
Errors are displayed
=====

Compiling 'gerador5' ...
Compiling 'gerador_5o' ...
Looking for the common auxiliary signals :

---> final number of nodes = 57(33)

Running Ab12Bdd on `gerador_5o`
=====

```

Figura 3: Otimização e prova de equivalência

Após, usamos a ferramenta **boog** (*Binding and Optimizing on Gates*) para mapear uma descrição comportamental em uma biblioteca de células padrão predefinidas. Essa ferramenta constrói uma rede booleana equivalente à descrição otimizada obtida com **boom**. Então, para cada função booleana de cada nó da rede ele tenta encontrar uma célula ou conjunto de células que implemente aquela função. O resultado é um arquivo “.vst” que será uma descrição estrutural baseada nas células da biblioteca *sxlib* da Alliance. Em seguida otimizamos esse arquivo utilizando a ferramenta **loon** (*Light Optimizing On Nets*).

```
50% area - 50% delay optimization
Reading file 'gerador_t.vst'...
Reading lib '/usr/alliance/cells/sxlib'...
Capacitances on file 'gerador_t.vst'...
Delays on file 'gerador_t.vst'...914 ps
Area on file 'gerador_t.vst'...74000 lamda (with over-cell routing)
Details...
    sff2_x4: 9 (72%)
    buf_x2: 8 (10%)
    xr2_x1: 3 (9%)
    sff1_x4: 1 (6%)
    inv_x2: 1 (1%)
    Total: 22
Worst RC on file 'gerador_t.vst'...274 ps
Inserting buffers on critical path for file 'gerator_to.vst'...None inserted
Improving RC on critical path for file 'gerator_to.vst'...910 ps
Improving all RC for file 'gerator_to.vst'...
Worst RC on file 'gerator_to.vst'...66 ps
Area on file 'gerator_to.vst'...75000 lamda (with over-cell routing)
Details...
    sff2_x4: 9 (72%)
    buf_x2: 8 (10%)
    xr2_x1: 3 (9%)
    sff1_x4: 1 (6%)
    inv_x8: 1 (2%)
    Total: 22
Critical path (no warranty)...910 ps from 'rtlalc_0 0' to 'lfsr_out 0'
Saving file 'gerator_to.vst'...
Saving critical path in xsch color file 'gerator_to.xsc'...
End of loon...
```

Figuras 4: Resultados da otimização do atraso crítico

Podemos observar que a otimização aumentou a área, o não otimizado tem área de 74 mil lambda, já o otimizado tem área de 75 mil lambda, e o número de nós continua o mesmo. Porém o caminho crítico de dados diminui, de 274 ps no não otimizado passa para 66 ps no otimizado, ou seja, o atraso máximo é menor no otimizado. Isso acontece pois os componentes no caminho crítico são bufferizados ao aumentar a largura desses componentes.

3. Resultado

Agora utilizamos a ferramenta **alliance-ocp** para posicionar as células padrão no circuito que será criado, e a ferramenta **nero** para rotear essas células, com isso obtemos um arquivo “.ap” e conseguimos visualizar o circuito gerado através da descrição comportamental utilizando o comando **xsch**.

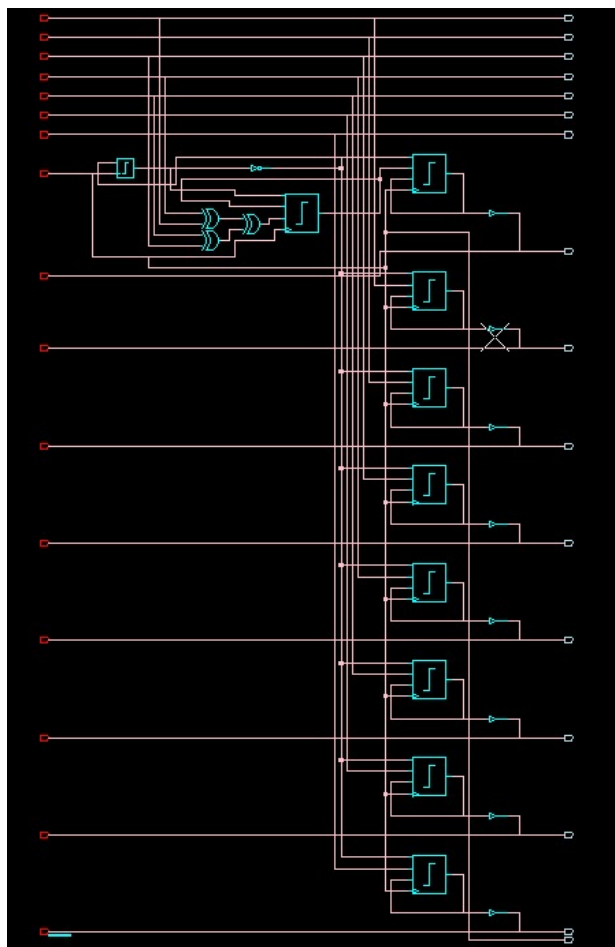


Figura 5: Circuito obtido.

Uma vez descrito o comportamento procedural e gerado o arquivo de padrão de testes com a biblioteca *genpat.h*, podemos visualizar esse padrão de testes gerado. O arquivo de padrão *.pat* é um arquivo de texto, mas com um formato rígido que pode ser usado para simulação e que pode ser visualizado graficamente com a ferramenta *xpat*.

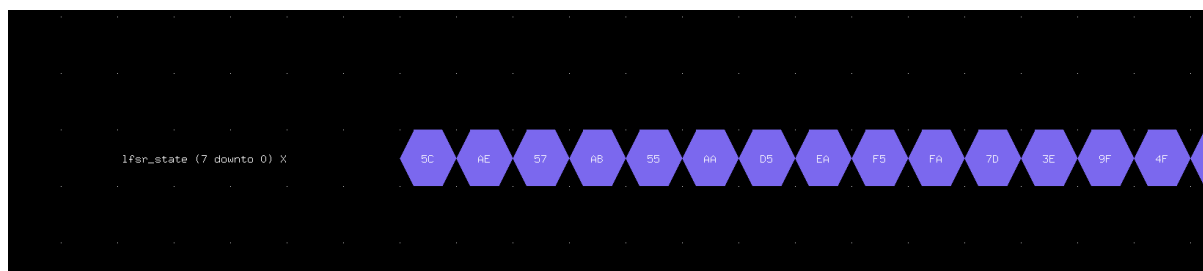


Figura 6: Visualização dos padrões de teste.

```

<      0 ps>      : 01011100 ;
<      1 ps>      : 10101110 ;
<      2 ps>      : 01010111 ;
<      3 ps>      : 10101011 ;
<      4 ps>      : 01010101 ;
<      5 ps>      : 10101010 ;
<      6 ps>      : 11010101 ;
<      7 ps>      : 11101010 ;
<      8 ps>      : 11110101 ;
<      9 ps>      : 11111010 ;
<     10 ps>      : 01111101 ;
<     11 ps>      : 00111110 ;
<     12 ps>      : 10011111 ;
<     13 ps>      : 01001111 ;
<     14 ps>      : 10100111 ;
<     15 ps>      : 01010011 ;
<     16 ps>      : 00101001 ;
<     17 ps>      : 00010100 ;
<     18 ps>      : 00001010 ;
<     19 ps>      : 10000101 ;
<     20 ps>      : 01000010 ;
<     21 ps>      : 00100001 ;
<     22 ps>      : 10010000 ;
<     23 ps>      : 11001000 ;
<     24 ps>      : 11100100 ;
<     25 ps>      : 11110010 ;
<     26 ps>      : 11111001 ;
<     27 ps>      : 11111100 ;
<     28 ps>      : 11111110 ;

```

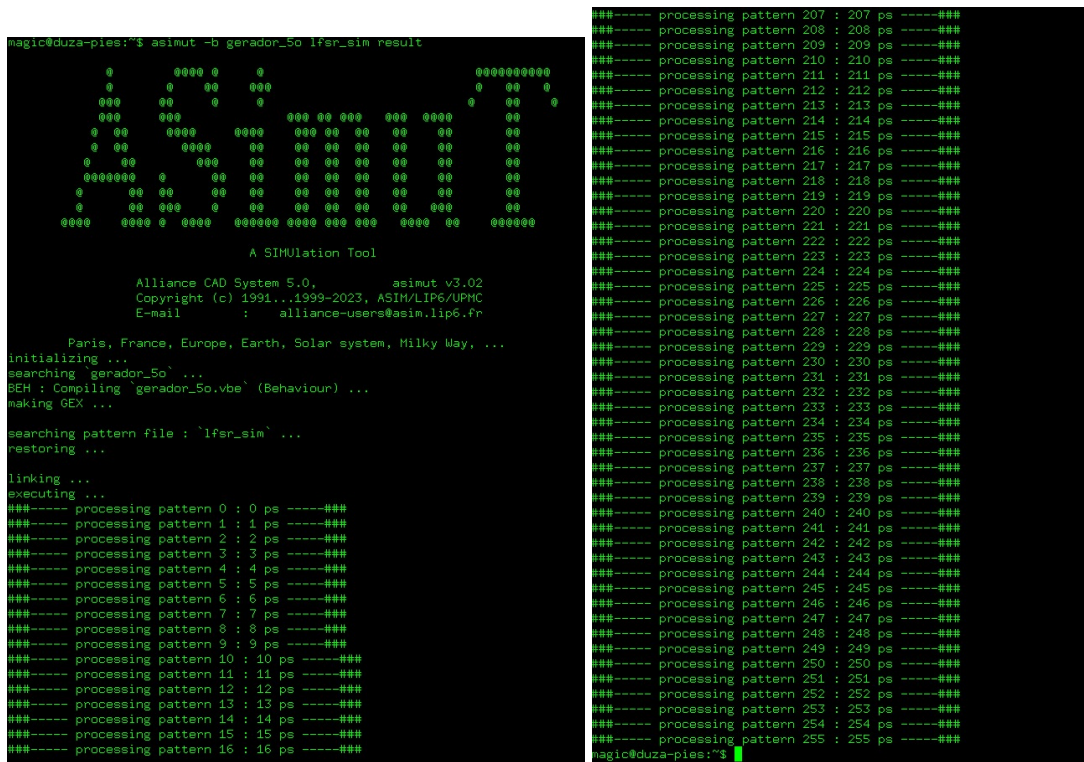
Figura 7: 'lfsr_sim.pat' gerado pelo *genpat.h*.

O arquivo é uma série temporal, em que cada linha corresponde a um instante de tempo, e cada coluna contém os valores em cada instante de uma variável. Ao observarmos os padrões de teste gerados, podemos ver que o primeiro número se repete apenas após 255 números gerados, e nenhum outro número gerado se repete antes. Ou seja, o programa gerou 255 números aleatórios sem repetição.

O arquivo com o padrão de testes pode ser usado para testar o funcionamento do circuito descrito em VHDL. Essa simulação é feita com a ferramenta *asimut* que faz uma comparação, lendo os valores dos dados de entrada e comparando com as saídas produzidas pelo circuito com aquelas prescritas no arquivo de padrão.

O *asimut* produz um arquivo de padrão de teste com o resultado da comparação e indica a presença de erros. Porém precisamos tratar o atraso de propagação das portas lógicas, o mais preciso é simular com um atraso correto, tabelado nas células padrão. Para isso, ao invés de usar a descrição em VHDL criada no início, iremos usar o resultado do roteiro de obtenção do layout físico para extrair um modelo de descrição estrutural (.vst) com os atrasos realistas listados nos arquivos .vbe das instâncias componentes.

Utilizamos a ferramenta *cougar* para extrair o modelo .vst do layout físico .ap que foi obtido após o *asimut/nero*. Esse modelo extraído é usado na simulação junto ao arquivo de padrões gerado pelo *genpat*.



```

magic@duza-ples:~$ asimut -b gerador_50 lfr_sim result

      0 0000 0 0 000000000
      0 0 00 000 0 00 0
      000 000 0 0 000 000 000 0000
      000 000 000 000 000 00 00 00
      0 00 0000 0000 000 00 00 00 00
      0 00 00 0000 00 00 00 00 00 00
      0000000 0 00 00 00 00 00 00 00
      0 00 00 00 00 00 00 00 00 00
      0 00 000 0 00 0000 000 00 0000
      0000 0000 0 0000 00000 000 00 000000

A SIMULATION Tool

Alliance CAD System 5.0, asimut v3.02
Copyright (c) 1991..1999-2023, ASIM/LIP6/UPMC
E-mail : alliance-users@asim.lip6.fr

Paris, France, Europe, Earth, Solar system, Milky Way, ...
initializing ...
searching `gerador_50` ...
BEH : Compiling `gerador_50.vbe` (Behaviour) ...
making GEX ...

searching pattern file : `lfr_sim` ...
restoring ...

linking ...
executing ...
##### processing pattern 0 : 0 ps -----###
##### processing pattern 1 : 1 ps -----###
##### processing pattern 2 : 2 ps -----###
##### processing pattern 3 : 3 ps -----###
##### processing pattern 4 : 4 ps -----###
##### processing pattern 5 : 5 ps -----###
##### processing pattern 6 : 6 ps -----###
##### processing pattern 7 : 7 ps -----###
##### processing pattern 8 : 8 ps -----###
##### processing pattern 9 : 9 ps -----###
##### processing pattern 10 : 10 ps -----###
##### processing pattern 11 : 11 ps -----###
##### processing pattern 12 : 12 ps -----###
##### processing pattern 13 : 13 ps -----###
##### processing pattern 14 : 14 ps -----###
##### processing pattern 15 : 15 ps -----###
##### processing pattern 16 : 16 ps -----###

##### processing pattern 207 : 207 ps -----###
##### processing pattern 208 : 208 ps -----###
##### processing pattern 209 : 209 ps -----###
##### processing pattern 210 : 210 ps -----###
##### processing pattern 211 : 211 ps -----###
##### processing pattern 212 : 212 ps -----###
##### processing pattern 213 : 213 ps -----###
##### processing pattern 214 : 214 ps -----###
##### processing pattern 215 : 215 ps -----###
##### processing pattern 216 : 216 ps -----###
##### processing pattern 217 : 217 ps -----###
##### processing pattern 218 : 218 ps -----###
##### processing pattern 219 : 219 ps -----###
##### processing pattern 220 : 220 ps -----###
##### processing pattern 221 : 221 ps -----###
##### processing pattern 222 : 222 ps -----###
##### processing pattern 223 : 223 ps -----###
##### processing pattern 224 : 224 ps -----###
##### processing pattern 225 : 225 ps -----###
##### processing pattern 226 : 226 ps -----###
##### processing pattern 227 : 227 ps -----###
##### processing pattern 228 : 228 ps -----###
##### processing pattern 229 : 229 ps -----###
##### processing pattern 230 : 230 ps -----###
##### processing pattern 231 : 231 ps -----###
##### processing pattern 232 : 232 ps -----###
##### processing pattern 233 : 233 ps -----###
##### processing pattern 234 : 234 ps -----###
##### processing pattern 235 : 235 ps -----###
##### processing pattern 236 : 236 ps -----###
##### processing pattern 237 : 237 ps -----###
##### processing pattern 238 : 238 ps -----###
##### processing pattern 239 : 239 ps -----###
##### processing pattern 240 : 240 ps -----###
##### processing pattern 241 : 241 ps -----###
##### processing pattern 242 : 242 ps -----###
##### processing pattern 243 : 243 ps -----###
##### processing pattern 244 : 244 ps -----###
##### processing pattern 245 : 245 ps -----###
##### processing pattern 246 : 246 ps -----###
##### processing pattern 247 : 247 ps -----###
##### processing pattern 248 : 248 ps -----###
##### processing pattern 249 : 249 ps -----###
##### processing pattern 250 : 250 ps -----###
##### processing pattern 251 : 251 ps -----###
##### processing pattern 252 : 252 ps -----###
##### processing pattern 253 : 253 ps -----###
##### processing pattern 254 : 254 ps -----###
##### processing pattern 255 : 255 ps -----###
magic@duza-ples:~$

```

Figura 8: Execução da simulação com asimut.

Por fim, foi gerado um arquivo de padrões de teste final, que nos mostra que nenhum erro foi observado durante a simulação com atraso real tabelado. Conclui-se então que o layout físico gerado foi bem sucedido.

4. Referencias:

<https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=76b0445329d967a5c564642232433203fe8357e8>

https://en.wikipedia.org/wiki/Linear-feedback_shift_register

https://edisciplinas.usp.br/pluginfile.php/5400280/mod_resource/content/1/SLIDES-Aula23-Parte-II-LinearFeedbackShiftRegisters-LFSRs-2020S1.pdf

https://edisciplinas.usp.br/pluginfile.php/5391242/mod_resource/content/1/SLIDES-Aula23-Registradores%20%2B%20Contadores-2020.pdf

<https://www.eng.auburn.edu/~nelson/courses/elec4200/Slides/VHDL%203%20Sequential.pdf>

<https://dcc.ufrj.br/~gabriel/circlog/VHDL-2.pdf>