



**INSTITUTO
FEDERAL**

Santa Catarina

Câmpus
São José

Dispositivos Lógicos Programáveis II

Implementação de PLL para Relógio Digital (Milisegundos)

Arthur Cadore Matuella Barcella e Gabriel Luiz Espindola Pedro

23 de Abril de 2024

Engenharia de Telecomunicações - IFSC-SJ

Sumário

1. Introdução	3
2. Implementação	3
2.1. Parte 1 - Adicionar Centésimo de Segundo ao Relógio	3
2.2. Parte 2 - Adicionar PLL	4
2.3. Parte 3 - Modificar contadores para BCD	7
2.4. Parte 4 - Modificar o r_reg para LFSR:	10
2.5. Parte 5 - Implementação na placa:	13
3. Conclusão	14
4. Códigos VHDL utilizados	14
4.1. bin2bcd	14
4.2. bcd2ssd:	15
4.3. timer:	15
4.4. top_timer:	17
4.5. single_clock_arch:	19

1. Introdução

Neste relatório, será apresentado o desenvolvimento de um relógio digital com precisão de milissegundos, utilizando um PLL (Phase-Locked Loop) para a geração de um sinal de clock de 5 kHz. O projeto foi desenvolvido utilizando a ferramenta Quartus Prime Lite Edition 20.1.0.720 e a placa de desenvolvimento DE2-115.

2. Implementação

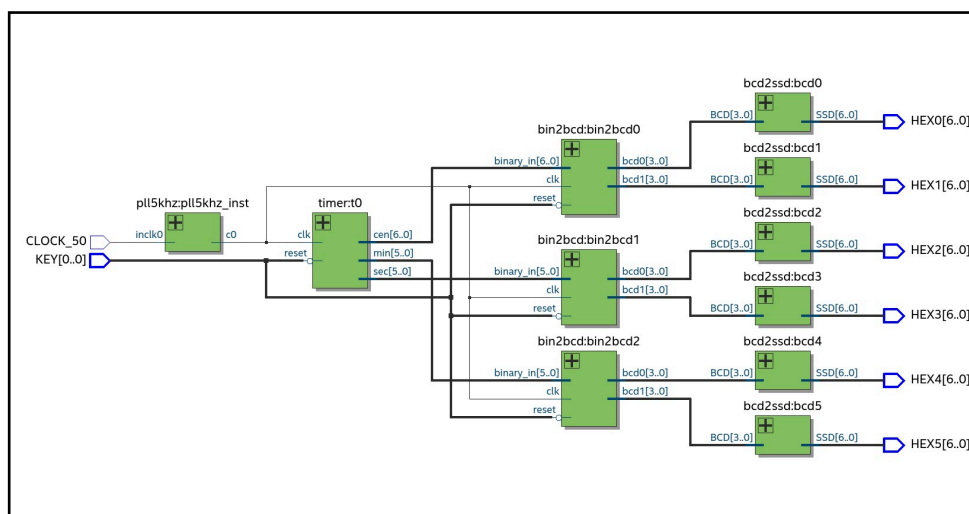
2.1. Parte 1 - Adicionar Centésimo de Segundo ao Relógio

A primeira etapa da implementação foi a adição de um contador de 100 para a contagem de centésimos de segundo. Para isso, foi utilizado um contador de 7 bits, que conta de 0 a 99, e um comparador para resetar o contador quando atingir o valor de 100.

Foi instanciado um novo componente de contagem ao circuito e dois conversores BDC2SSD para a impressão dos dígitos em um display de 7 segmentos.

O seguinte RTL foi gerado após a adição do contador de centésimos de segundo ao circuito:


Figure 1: Elaborada pelo Autor



RTL do circuito operando com PLL

Em seguida, verificamos a contagem através do modelsim e observamos que o contador de centésimos de segundo estava funcionando corretamente, contando de 0 a 99 e resetando para 0 após atingir o valor de 100, conforme a imagem abaixo:

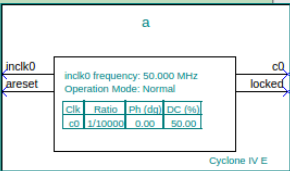
[illegible]



[About](#)
[Documentation](#)

1 Parameter Settings
2 PLL Reconfiguration
3 **Output Clocks**
4 EDA
5 Summary

clk c0 > clk c1 > clk c2 > clk c3 > clk c4



Cyclone IV E

c0 - Core/External Output Clock

Able to implement the requested PLL

☒ Use this clock

Clock Tap Settings

☒ Enter output clock frequency:
☐ Enter output clock parameters:
Clock multiplication factor
Clock division factor
Clock phase shift
Clock duty cycle (%)

Requested Settings

0.005 MHz

1

10000

0.00

50.00

Actual Settings

0.005000

1

10000

0.00

50.00

<< Copy

Description	Val
Primary clock VCO frequency (MHz)	...
Modulus for M counter	12
Modulus for N counter	1
Initial VCO phase offset for M counter	...

Per Clock Feasibility Indicators

c0 c1 c2 c3 c4

Note: The displayed internal settings of the PLL is recommended for use by advanced users only

Cancel

< Back

Next >

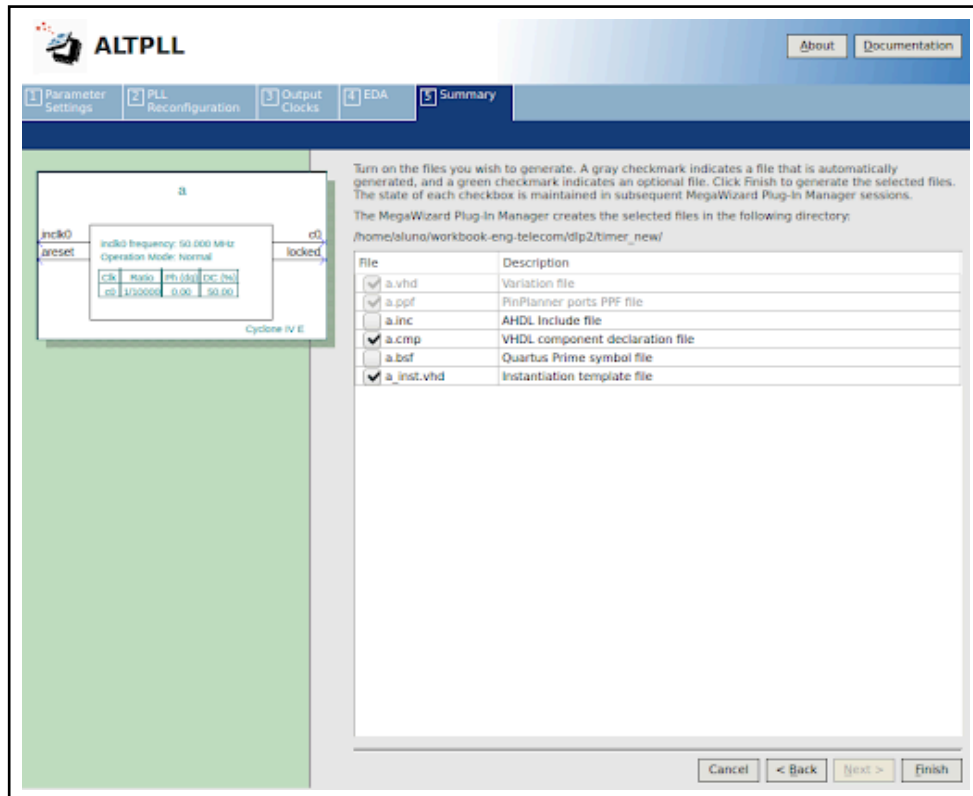
Finish

Engenharia de Telecomunicações - IFSC-SJ

Na própria ferramenta, ao inserir os valores de entrada e saída desejados para o circuito de PLL, o Quartus gera o código VHDL necessário para a configuração do circuito que irá controlar a seção analógica do PLL, assim sendo possível realizar a multiplicação ou divisão de frequência corretamente.

Ao finalizar a configuração, foi solicitado gerar os seguintes arquivos:

Figure 4: Elaborada pelo Autor



Configuração do PLL através da ferramenta ALT-PLL

Abaixo está uma sessão do código VHD gerado pelo Quartus, para a configuração VHDL do PLL, demais arquivos são necessários para realizar a instânciação do PLL como um componente do circuito principal.

```
1 GENERIC MAP (  
2   bandwidth_type => "AUTO",  
3   clk0_divide_by => 10000,  
4   clk0_duty_cycle => 50,  
5   clk0_multiply_by => 1,  
6   clk0_phase_shift => "0",  
7   compensate_clock => "CLK0",  
8   inclk0_input_frequency => 50000,  
9   intended_device_family => "Cyclone IV E",  
10  lpm_hint => "CBX_MODULE_PREFIX=pll",  
11  lpm_type => "altpll",  
12  operation_mode => "NORMAL",  
13  pll_type => "AUTO",
```

É possível notar na descrição acima frequência de entrada, a frequência de saída, o fator de divisão e o duty-cycle do circuito de PLL.

Esses parâmetros são necessários para determinar o formato da onda na saída do circuito, sendo que a frequência precisa ser dividida pelas 10.000 vezes para obter a frequência de 5 kHz.

O duty cycle é de 50% para que a onda seja simétrica, abaixo está uma imagem para ilustrar a diferença entre um duty-cycle de 50% entre 25% e 75%:

Figure 5: Elaborada pelo Autor

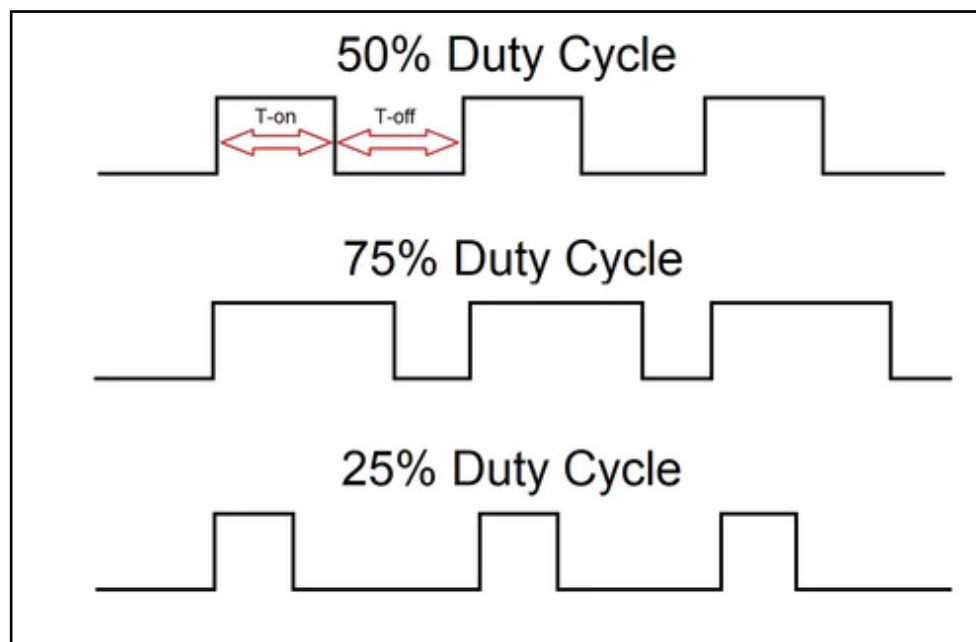
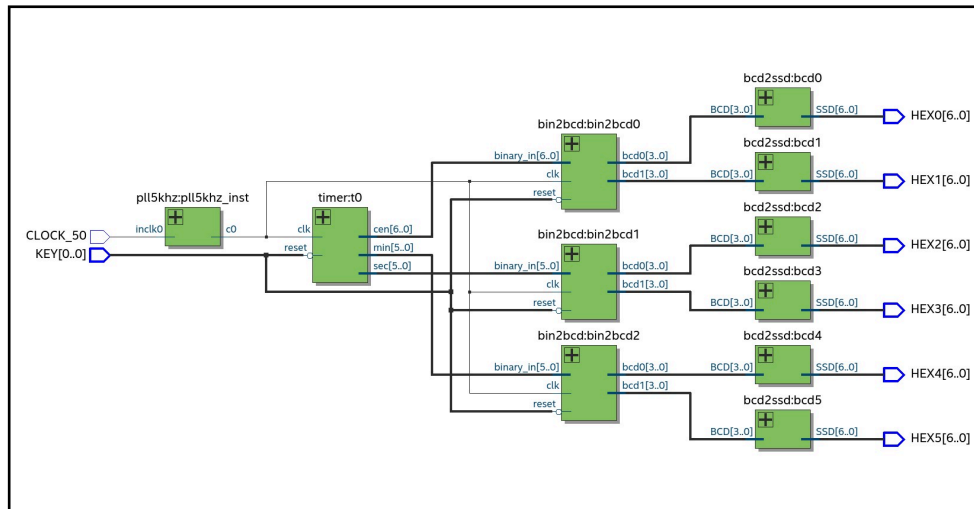


Ilustração de variação de duty-cycle

Com a adição do PLL no circuito, temos uma topologia RTL com um intermediário entre o clock de entrada e o clock de saída, como ilustrado abaixo.

Para ajustar a contagem do segundo, o componente de timer também teve que ser ajustado para contar 5.000 vezes mais rápido, ou seja, de 0 a 99,99 em 5.000 ms.

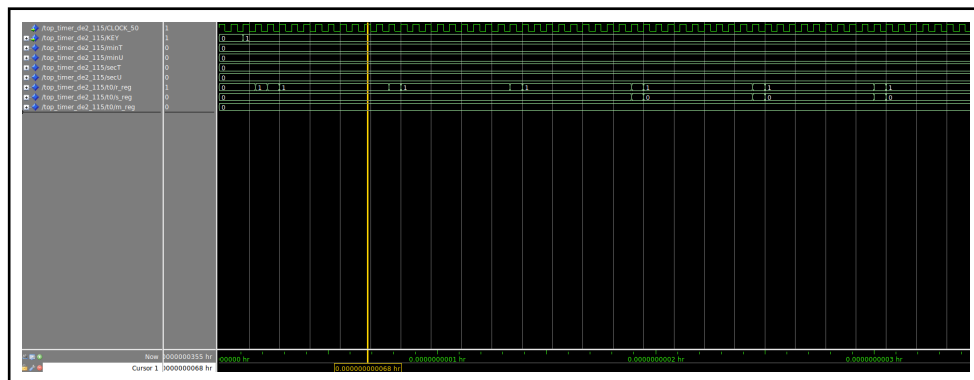
Figure 6: Elaborada pelo Autor



RTL do circuito operando com PLL

Em seguida, realizamos a simulação do circuito com a adição do PLL e verificamos seu funcionamento, conforme a imagem abaixo, a simulação apresenta a mesma característica de onda pois apenas o clock foi alterado, mantendo a contagem de centésimos de segundo correta:

Figure 7: Elaborada pelo Autor



Contagem de centésimos de segundo com PLL

2.3. Parte 3 - Modificar contadores para BCD

Para realizarmos a parte 3, alteramos o método de contagem do contador de centésimos de segundo para BCD (Binary-Coded Decimal), que é uma forma de representar números decimais utilizando 4 bits para cada dígito.

Diferentemente da topologia RTL nos casos anteriores, na parte 3, toda a contagem é feita diretamente em um único componente. Para isso, diversos sinais foram criados para armazenar a contagem atual e repassar o status da contagem para o próximo ciclo de clock.

```
1 ARCHITECTURE single_clock_arch OF timer IS
2     SIGNAL r_reg : unsigned(5 DOWNTO 0);
3     SIGNAL r_next : unsigned(5 DOWNTO 0);
4
```

```

5     SIGNAL s_u_reg, m_u_reg : unsigned(3 DOWNTO 0);
6     SIGNAL s_d_reg, m_d_reg : unsigned(3 DOWNTO 0);
7
8     SIGNAL s_u_next, m_u_next : unsigned(3 DOWNTO 0);
9     SIGNAL s_d_next, m_d_next : unsigned(3 DOWNTO 0);
10
11    SIGNAL s_en, m_en : STD_LOGIC;
12
13    SIGNAL c_u_reg, c_u_next : unsigned(3 DOWNTO 0);
14    SIGNAL c_en : STD_LOGIC;
15
16    SIGNAL c_d_reg, c_d_next : unsigned(3 DOWNTO 0);

```

Em seguida, a lógica de contagem foi implementada de maneira a contar os centesimos de segundo, segundos e minutos:

```

1  BEGIN
2      -- register
3      PROCESS (clk, reset)
4      BEGIN
5          IF (reset = '1') THEN
6              r_reg <= (OTHERS => '0');
7              s_u_reg <= (OTHERS => '0');
8              m_u_reg <= (OTHERS => '0');
9
10             s_d_reg <= (OTHERS => '0');
11             m_d_reg <= (OTHERS => '0');
12
13             c_u_reg <= (OTHERS => '0');
14             c_d_reg <= (OTHERS => '0');
15             ELSIF (rising_edge(clk)) THEN
16                 r_reg <= r_next;
17                 c_u_reg <= c_u_next;
18                 c_d_reg <= c_d_next;
19                 s_u_reg <= s_u_next;
20                 s_d_reg <= s_d_next;
21                 m_u_reg <= m_u_next;
22                 m_d_reg <= m_d_next;
23             END IF;
24         END PROCESS;
25
26         -- next-state logic/output logic for mod-1000000 counter
27         r_next <= (OTHERS => '0') WHEN r_reg = 49 ELSE
28             r_reg + 1;
29
30         c_en <= '1' WHEN r_reg = 49 ELSE
31             '0';
32
33         s_en <= '1' WHEN c_d_reg = 9 AND c_u_reg = 9 AND c_en = '1' ELSE
34             '0';
35
36         m_en <= '1' WHEN s_d_reg = 5 AND s_u_reg = 9 AND s_en = '1' ELSE
37             '0';
38
39         -- next-state logic/output logic for centisecond units
40         c_u_next <= (OTHERS => '0') WHEN (c_u_reg = 9 AND c_en = '1') ELSE
41             c_u_reg + 1 WHEN c_en = '1' ELSE

```



```

42     c_u_reg;
43
44     -- next-state logic/output logic for centisecond tens
45     c_d_next <= (OTHERS => '0') WHEN (c_d_reg = 9 AND c_u_reg = 9 AND c_en
= '1') ELSE
46         c_d_reg + 1 WHEN (c_u_reg = 9 AND c_en = '1') ELSE
47         c_d_reg;
48
49     -- next-state logic/output logic for second units
50     s_u_next <= (OTHERS => '0') WHEN (s_u_reg = 9 AND s_en = '1') ELSE
51         s_u_reg + 1 WHEN s_en = '1' ELSE
52         s_u_reg;
53
54     -- next-state logic/output logic for second tens
55     s_d_next <= (OTHERS => '0') WHEN (s_d_reg = 9 AND s_u_reg = 9 AND s_en
= '1') ELSE
56         s_d_reg + 1 WHEN (s_u_reg = 9 AND s_en = '1') ELSE
57         s_d_reg;
58
59     -- next-state logic/output logic for minute units
60     m_u_next <= (OTHERS => '0') WHEN (m_u_reg = 9 AND m_en = '1') ELSE
61         m_u_reg + 1 WHEN m_en = '1' ELSE
62         m_u_reg;
63
64     -- next-state logic/output logic for minute tens
65     m_d_next <= (OTHERS => '0') WHEN (m_d_reg = 9 AND m_u_reg = 9 AND m_en
= '1') ELSE
66         m_d_reg + 1 WHEN (m_u_reg = 9 AND m_en = '1') ELSE
67         m_d_reg;

```

E por fim, a saída desta contagem foi repassada para os displays de 7 segmentos para exibição:

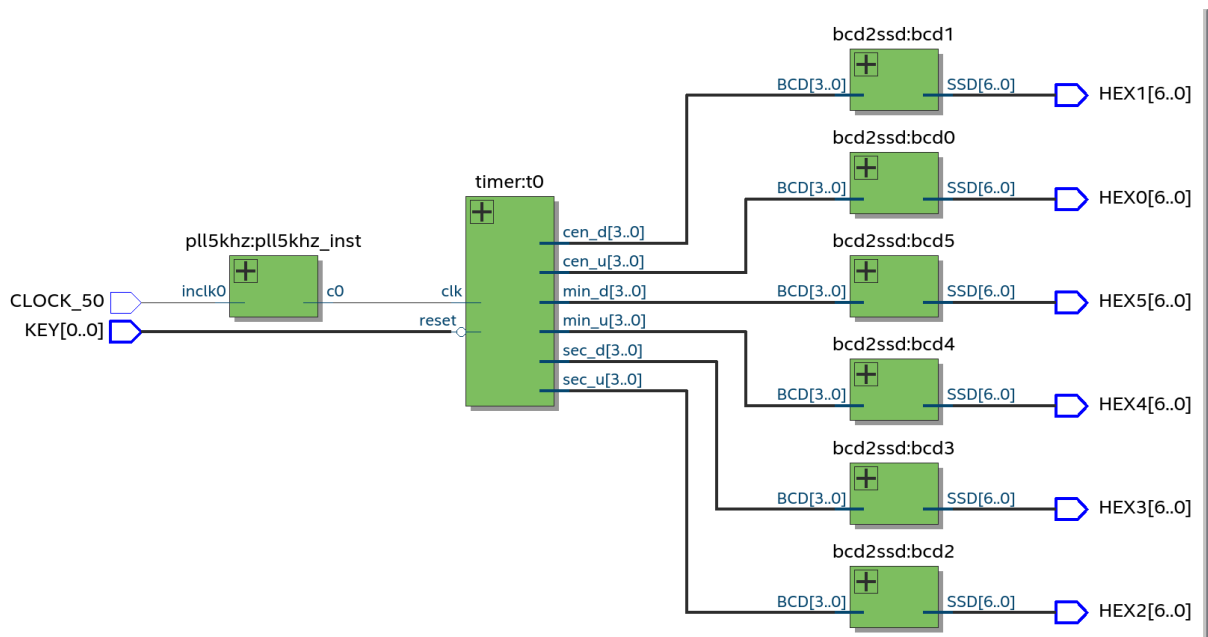
```

1     -- output logic
2     cen_u <= STD_LOGIC_VECTOR(c_u_reg);
3     cen_d <= STD_LOGIC_VECTOR(c_d_reg);
4
5     sec_u <= STD_LOGIC_VECTOR(s_u_reg);
6     sec_d <= STD_LOGIC_VECTOR(s_d_reg);
7
8     min_u <= STD_LOGIC_VECTOR(m_u_reg);
9     min_d <= STD_LOGIC_VECTOR(m_d_reg);
10  END single_clock_arch;

```

Com a implementação do contador BCD, podemos ver alteração na topologia do RTL, conforme apresentado abaixo:

Figure 8: Elaborada pelo Autor



Timer com contagem em BCD

2.4. Parte 4 - Modificar o r_reg para LFSR:

A parte 4 consiste em modificar o registrador r_reg para um registrador LFSR (Linear Feedback Shift Register), o objetivo é retirar o contador sequencial que é utilizado por padrão e substituir por um contador LFSR.

Essa modificação permite uma vantagem no circuito pois ao utilizar um contador sequencial comum, são necessários diversos registradores para armazenar o estado atual do número, nesta atividade por exemplo, utilizando contadores sequenciais, seriam necessários 13 registradores para armazenar o estado atual do contador de 13 bits, pois 13 bits geram 8.192 possibilidades o equivalente as 5000 contagens necessárias para o circuito.

Agora, ao substituírmos o contador sequencial por um LFSR, é possível realizar a contagem até 5000 com apenas 4 Taps (onde operações XOR são realizadas para gerar o próximo estado do contador) e 13 bits, o que reduz a quantidade de registradores necessários para armazenar o estado atual do contador, e também a complexidade e o tempo de propagação do circuito.

Desta forma, seguindo a planilha repassada professor, utilizamos a seguinte configuração para o LFSR, foi utilizado os seguintes parâmetros:

- Seed: 1111111111111
- Taps: [0, 2, 3, 12]
- bits: [12, 10, 9, 0] (Os bits são ordenados de acordo com a ordem de saída do LFSR, ou seja, inversos aos taps)

Entretanto, para utilizarmos o contador LFSR, é necessário também identificar qual a sequência de bits que estará 5000 contagens a frente da sequência considerada inicial, ou seja, no

nosso caso, a seed escolhida foi um vetor com 13x1, então, devemos identificar qual o vetor que estará 5000 casas a frente, para podermos resetar o contador e iniciar a contagem novamente.

Para isso, executamos o código Python abaixo para gerar a sequência LFSR e imprimir o estado do LFSR após 5000 contagens:

```
1 cadore: ~$ python3 find_LFSR.py
2 Estado do LFSR na contagem 5000: 1011111001001
```

O código Python utilizado possui a seguinte estrutura:

```
1 # -*- coding: utf-8 -*-
2
3 def lfsr(seed, taps, count):
4     # Converte o seed de string binária para uma lista de inteiros
5     state = [int(bit) for bit in seed]
6
7     # Função para calcular o próximo bit usando os taps
8     def next_bit(state, taps):
9         xor = 0
10        for t in taps:
11            xor ^= state[t]
12        return xor
13
14    for i in range(count):
15        new_bit = next_bit(state, taps) # Calcula o próximo bit
16        state = [new_bit] + state[:-1] # Desloca os bits para a direita e
17        insere o novo bit na frente
18        state_str = ''.join(map(str, state))
19        print(f"{i + 1}: {state_str}")
20
21    # Parâmetros
22    seed = '1111111111111'
23    taps = [0, 2, 3, 12]
24    count = 5000
25
26    # Gera a sequência LFSR e imprime todos os estados
27    lfsr(seed, taps, count)
```

Em seguida, alteramos a implementação do componente de contagem para utilizar o LFSR ao invés do contador sequencial, como ilustrado abaixo:

```
1 BEGIN
2     -- register
3     PROCESS (clk, reset)
4     BEGIN
5         IF (reset = '1') THEN
6             LFSR_reg <= SEED;
7             s_u_reg <= (OTHERS => '0');
8             m_u_reg <= (OTHERS => '0');
9
10            s_d_reg <= (OTHERS => '0');
11            m_d_reg <= (OTHERS => '0');
```

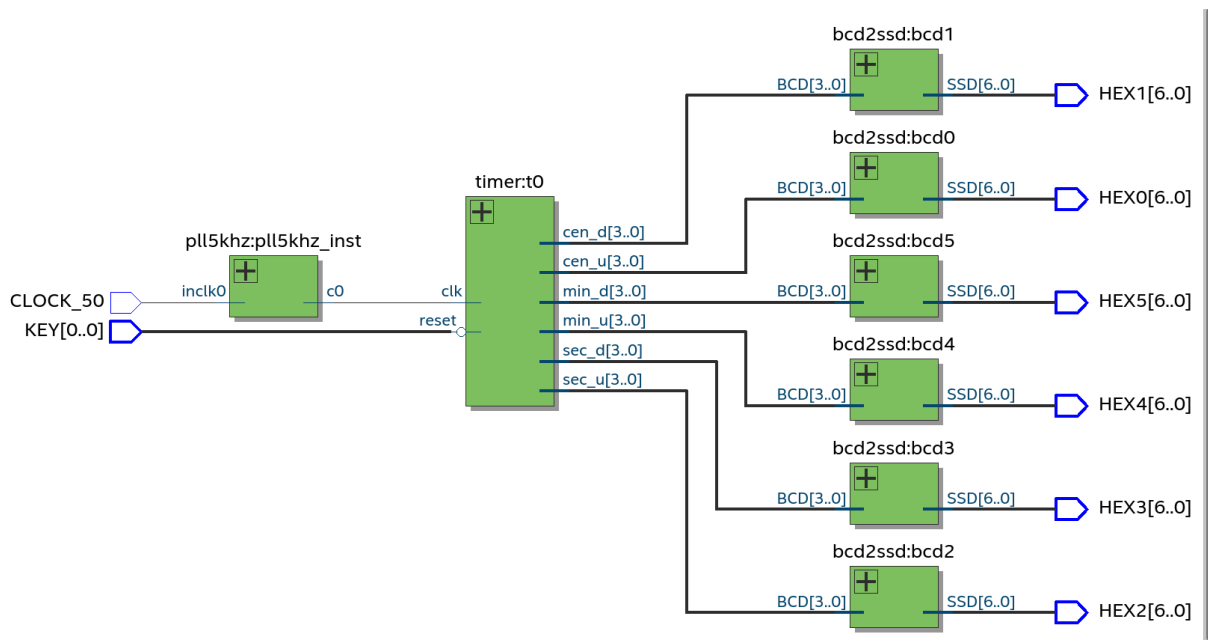
```

12
13         c_u_reg <= (OTHERS => '0');
14         c_d_reg <= (OTHERS => '0');
15     ELSIF (rising_edge(clk)) THEN
16         LFSR_reg <= LFSR_next;
17         c_u_reg <= c_u_next;
18         c_d_reg <= c_d_next;
19         s_u_reg <= s_u_next;
20         s_d_reg <= s_d_next;
21         m_u_reg <= m_u_next;
22         m_d_reg <= m_d_next;
23     END IF;
24 END PROCESS;
25
26 fb <= LFSR_reg(5) XOR LFSR_reg(0);
27
28 LFSR_next <= SEED WHEN LFSR_reg = CONST_RESET
29     ELSE
30     fb & LFSR_reg(5 DOWNTO 1);
31
32 c_en <= '1' WHEN LFSR_reg = CONST_RESET
33     ELSE
34     '0';
35
36 s_en <= '1' WHEN c_d_reg = 9 AND c_u_reg = 9 AND c_en = '1' ELSE
37     '0';
38
39 m_en <= '1' WHEN s_d_reg = 5 AND s_u_reg = 9 AND s_en = '1' ELSE
40     '0';
41
42 -- next-state logic/output logic for centisecond units
43 c_u_next <= (OTHERS => '0') WHEN (c_u_reg = 9 AND c_en = '1') ELSE
44     c_u_reg + 1 WHEN c_en = '1' ELSE
45     c_u_reg;
46
47 -- next-state logic/output logic for centisecond tens
48 c_d_next <= (OTHERS => '0') WHEN (c_d_reg = 9 AND c_u_reg = 9 AND c_en
= '1') ELSE
49     c_d_reg + 1 WHEN (c_u_reg = 9 AND c_en = '1') ELSE
50     c_d_reg;

```

Nesta modificação, apenas o componente de contagem foi alterado, desta forma, mantendo a estrutura do RTL igual:

Figure 9: Elaborada pelo Autor

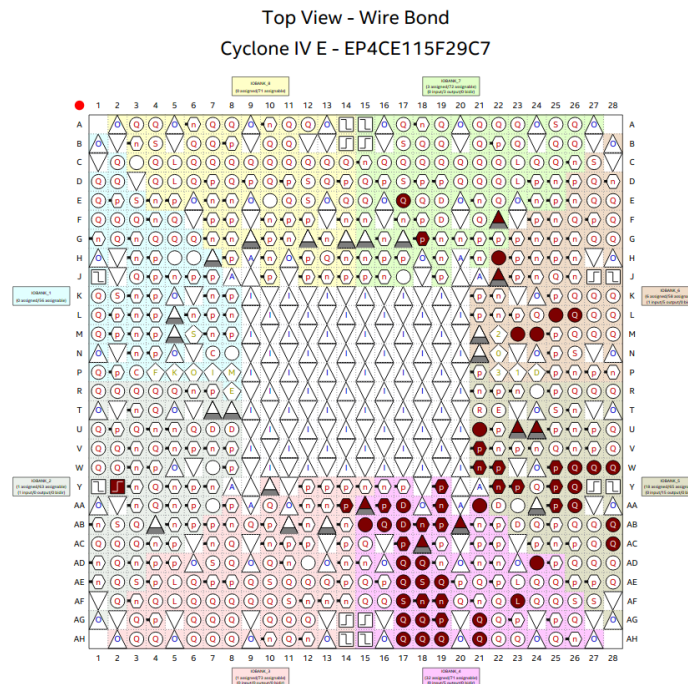


Timer com contagem em LFSR

2.5. Parte 5 - Implementação na placa:

Para cada etapa, realizamos a implementação na placa de desenvolvimento DE2-115, e verificamos o funcionamento do circuito.

Figure 10: Elaborada pelo Autor



Sinal de entrada no domínio do tempo

Desta forma, obtivemos os seguintes resultados:

Table 1: Elaborada pelo Autor

Implementacao	Área (LE)	Registradores
Parte 1	83	13.699
Parte 2	239	124
Parte 3	102	30
Parte 4	101	24

Tabela de resultados da implementação

3. Conclusão

A partir da implementação do PLL vista anteriormente, juntamente com a implementação de divisão de clock sem o uso do PLL, podemos concluir que a utilização de um PLL é muito útil para a geração de sinais de clock com frequências específicas de maneira confiável.

Isso pois o PLL é capaz de gerar sinais de clock com frequências específicas, além de possuir uma maior precisão e estabilidade em relação a outros métodos de geração de clock. Além disso, podemos concluir que a contagem de maneira não sequencial através de um LFSR é mais eficiente e consome menos recursos da FPGA, além de ser mais rápido e eficiente.

Isso pois o LFSR pois não precisar contar de maneira sequencial e necessitar apenas de operações básicas para funcionar, não é só mais eficiente em termos de consumo de recursos, mais também possui um tempo de operação menor, contribuindo para um tempo menor de propagação do circuito.

4. Códigos VHDL utilizados

Abaixo estão os demais códigos VHDL utilizados para a implementação do projeto.

4.1. bin2bcd

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity bin2bcd is
6      port (
7          A      : in  std_logic_vector (7 downto 0);
8          sd, su, sc : out std_logic_vector (3 downto 0)
9      );
10 end entity;
11
12 architecture ifsc_v1 of bin2bcd is
13     signal A_uns      : unsigned (7 downto 0);
14     signal sd_uns, su_uns, sc_uns : unsigned (7 downto 0);
15
16 begin
17     A_uns  <= unsigned(A);
18     sc_uns <= A_uns/100;

```

```

19     sd_uns <= A_uns/10;
20     su_uns <= A_uns rem 10;
21     sc      <= std_logic_vector(resize(sc_uns, 4));
22     sd      <= std_logic_vector(resize(sd_uns, 4));
23     su      <= std_logic_vector(resize(su_uns, 4));
24 end architecture;

```

4.2. bcd2ssd:

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity bcd2ssd is
5      port (
6          BCD : in std_logic_vector (3 downto 0);
7          SSD : out std_logic_vector (6 downto 0)
8      );
9
10 end entity;
11
12 architecture arch of bcd2ssd is
13 begin
14
15     with BCD select
16         SSD <= "1000000" when "0000",
17         "1111001" when "0001",
18         "0100100" when "0010",
19         "0110000" when "0011",
20         "0011001" when "0100",
21         "0010010" when "0101",
22         "0000011" when "0110",
23         "1111000" when "0111",
24         "0000000" when "1000",
25         "0011000" when "1001",
26         "0111111" when others;
27 end arch;

```

4.3. timer:

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4
5  ENTITY timer IS
6      PORT (
7          clk, reset : IN STD_LOGIC;
8          cen_u, cen_d : OUT STD_LOGIC_VECTOR(3 DOWNT0 0);
9          sec_u, sec_d : OUT STD_LOGIC_VECTOR(3 DOWNT0 0);
10         min_u, min_d : OUT STD_LOGIC_VECTOR(3 DOWNT0 0)
11     );
12 END timer;
13
14 ARCHITECTURE single_clock_arch OF timer IS
15     SIGNAL r_reg : unsigned(5 DOWNT0 0);

```

```

16     SIGNAL r_next : unsigned(5 DOWNTO 0);
17
18     SIGNAL s_u_reg, m_u_reg : unsigned(3 DOWNTO 0);
19     SIGNAL s_d_reg, m_d_reg : unsigned(3 DOWNTO 0);
20
21     SIGNAL s_u_next, m_u_next : unsigned(3 DOWNTO 0);
22     SIGNAL s_d_next, m_d_next : unsigned(3 DOWNTO 0);
23
24     SIGNAL s_en, m_en : STD_LOGIC;
25
26     SIGNAL c_u_reg, c_u_next : unsigned(3 DOWNTO 0);
27     SIGNAL c_en : STD_LOGIC;
28
29     SIGNAL c_d_reg, c_d_next : unsigned(3 DOWNTO 0);
30
31 BEGIN
32     -- register
33     PROCESS (clk, reset)
34     BEGIN
35         IF (reset = '1') THEN
36             r_reg <= (OTHERS => '0');
37             s_u_reg <= (OTHERS => '0');
38             m_u_reg <= (OTHERS => '0');
39
40             s_d_reg <= (OTHERS => '0');
41             m_d_reg <= (OTHERS => '0');
42
43             c_u_reg <= (OTHERS => '0');
44             c_d_reg <= (OTHERS => '0');
45         ELSIF (rising_edge(clk)) THEN
46             r_reg <= r_next;
47             c_u_reg <= c_u_next;
48             c_d_reg <= c_d_next;
49             s_u_reg <= s_u_next;
50             s_d_reg <= s_d_next;
51             m_u_reg <= m_u_next;
52             m_d_reg <= m_d_next;
53         END IF;
54     END PROCESS;
55
56     -- next-state logic/output logic for mod-1000000 counter
57     r_next <= (OTHERS => '0') WHEN r_reg = 49 ELSE
58         r_reg + 1;
59
60     c_en <= '1' WHEN r_reg = 49 ELSE
61         '0';
62
63     s_en <= '1' WHEN c_d_reg = 9 AND c_u_reg = 9 AND c_en = '1' ELSE
64         '0';
65
66     m_en <= '1' WHEN s_d_reg = 5 AND s_u_reg = 9 AND s_en = '1' ELSE
67         '0';
68
69     -- next-state logic/output logic for centisecond units
70     c_u_next <= (OTHERS => '0') WHEN (c_u_reg = 9 AND c_en = '1') ELSE
71         c_u_reg + 1 WHEN c_en = '1' ELSE
72         c_u_reg;
73
74     -- next-state logic/output logic for centisecond tens

```



```

75     c_d_next <= (OTHERS => '0') WHEN (c_d_reg = 9 AND c_u_reg = 9 AND c_en
= '1') ELSE
76         c_d_reg + 1 WHEN (c_u_reg = 9 AND c_en = '1') ELSE
77         c_d_reg;
78
79     -- next-state logic/output logic for second units
80     s_u_next <= (OTHERS => '0') WHEN (s_u_reg = 9 AND s_en = '1') ELSE
81         s_u_reg + 1 WHEN s_en = '1' ELSE
82         s_u_reg;
83
84     -- next-state logic/output logic for second tens
85     s_d_next <= (OTHERS => '0') WHEN (s_d_reg = 5 AND s_u_reg = 9 AND s_en
= '1') ELSE
86         s_d_reg + 1 WHEN (s_u_reg = 9 AND s_en = '1') ELSE
87         s_d_reg;
88
89     -- next-state logic/output logic for minute units
90     m_u_next <= (OTHERS => '0') WHEN (m_u_reg = 9 AND m_en = '1') ELSE
91         m_u_reg + 1 WHEN m_en = '1' ELSE
92         m_u_reg;
93
94     -- next-state logic/output logic for minute tens
95     m_d_next <= (OTHERS => '0') WHEN (m_d_reg = 5 AND m_u_reg = 9 AND m_en
= '1') ELSE
96         m_d_reg + 1 WHEN (m_u_reg = 9 AND m_en = '1') ELSE
97         m_d_reg;
98
99     -- output logic
100    cen_u <= STD_LOGIC_VECTOR(c_u_reg);
101    cen_d <= STD_LOGIC_VECTOR(c_d_reg);
102
103    sec_u <= STD_LOGIC_VECTOR(s_u_reg);
104    sec_d <= STD_LOGIC_VECTOR(s_d_reg);
105
106    min_u <= STD_LOGIC_VECTOR(m_u_reg);
107    min_d <= STD_LOGIC_VECTOR(m_d_reg);
108 END single_clock_arch;

```

4.4. top_timer:

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4
5  ENTITY top_timer_de2_115 IS
6      PORT (
7          CLOCK_50 : IN STD_LOGIC;
8          KEY : IN STD_LOGIC_VECTOR (0 DOWNT0 0);
9          HEX0 : OUT STD_LOGIC_VECTOR (6 DOWNT0 0);
10         HEX1 : OUT STD_LOGIC_VECTOR (6 DOWNT0 0);
11         HEX2 : OUT STD_LOGIC_VECTOR (6 DOWNT0 0);
12         HEX3 : OUT STD_LOGIC_VECTOR (6 DOWNT0 0);
13         HEX4 : OUT STD_LOGIC_VECTOR (6 DOWNT0 0);
14         HEX5 : OUT STD_LOGIC_VECTOR (6 DOWNT0 0)
15     );
16

```

```

17 END ENTITY;
18
19 ARCHITECTURE top_a3_2019_2 OF top_timer_de2_115 IS
20
21     COMPONENT timer IS
22     PORT (
23         clk, reset : IN STD_LOGIC;
24         cen_u, cen_d : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
25         sec_u, sec_d : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
26         min_u, min_d : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
27     );
28 END COMPONENT;
29
30 COMPONENT bin2bcd IS
31     GENERIC (N : POSITIVE := 16);
32     PORT (
33         clk, reset : IN STD_LOGIC;
34         binary_in : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
35         bcd0, bcd1, bcd2, bcd3, bcd4 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
36     );
37 END COMPONENT;
38
39 COMPONENT bcd2ssd
40     PORT (
41         BCD : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
42         SSD : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)
43     );
44 END COMPONENT;
45
46 COMPONENT pll5khz IS
47     PORT (
48         inclk0 : IN STD_LOGIC := '0';
49         c0 : OUT STD_LOGIC
50     );
51 END COMPONENT;
52
53 SIGNAL minT, minU : STD_LOGIC_VECTOR(3 DOWNTO 0);
54 SIGNAL secT, secU : STD_LOGIC_VECTOR(3 DOWNTO 0);
55 SIGNAL centT, centU : STD_LOGIC_VECTOR(3 DOWNTO 0);
56 SIGNAL min, sec : STD_LOGIC_VECTOR(5 DOWNTO 0);
57 SIGNAL cent : STD_LOGIC_VECTOR(6 DOWNTO 0);
58 SIGNAL r_reg, r_next : unsigned(22 DOWNTO 0);
59 SIGNAL reset : STD_LOGIC;
60 SIGNAL c0 : STD_LOGIC;
61
62 BEGIN
63
64     reset <= NOT KEY(0);
65
66     t0 : timer
67     PORT MAP(
68         clk => c0,
69         reset => reset,
70         cen_u => centU,
71         cen_d => centT,
72         sec_u => secU,
73         sec_d => secT,
74         min_u => minU,

```

```

75     min_d => minT
76 );
77
78 pll5khz_inst : pll5khz PORT MAP(
79     inclk0 => CLOCK_50,
80     c0 => c0
81 );
82
83 bcd0 : bcd2ssd
84 PORT MAP(
85     BCD => centU,
86     SSD => HEX0
87 );
88
89 bcd1 : bcd2ssd
90 PORT MAP(
91     BCD => centT,
92     SSD => HEX1
93 );
94
95 bcd2 : bcd2ssd
96 PORT MAP(
97     BCD => secU,
98     SSD => HEX2
99 );
100
101 bcd3 : bcd2ssd
102 PORT MAP(
103     BCD => secT,
104     SSD => HEX3
105 );
106
107 bcd4 : bcd2ssd
108 PORT MAP(
109     BCD => minU,
110     SSD => HEX4
111 );
112
113 bcd5 : bcd2ssd
114 PORT MAP(
115     BCD => minT,
116     SSD => HEX5
117 );
118
119 END top_a3_2019_2;

```

4.5. single_clock_arch:

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4  ENTITY timer IS
5      PORT (
6          clk, reset : IN STD_LOGIC;
7          cen_u, cen_d : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
8          sec_u, sec_d : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);

```

```

9      min_u, min_d : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
10    );
11  END timer;
12
13  ARCHITECTURE single_clock_arch OF timer IS
14    SIGNAL r_next : unsigned(5 DOWNTO 0);
15
16    SIGNAL s_u_reg, m_u_reg : unsigned(3 DOWNTO 0);
17    SIGNAL s_d_reg, m_d_reg : unsigned(3 DOWNTO 0);
18
19    SIGNAL s_u_next, m_u_next : unsigned(3 DOWNTO 0);
20    SIGNAL s_d_next, m_d_next : unsigned(3 DOWNTO 0);
21
22    SIGNAL s_en, m_en : STD_LOGIC;
23
24
25    SIGNAL c_u_reg, c_u_next : unsigned(3 DOWNTO 0);
26    SIGNAL c_en : STD_LOGIC;
27
28    SIGNAL c_d_reg, c_d_next : unsigned(3 DOWNTO 0);
29
30    CONSTANT CONST_RESET : unsigned(5 DOWNTO 0) := "000110";
31    CONSTANT SEED : unsigned(5 DOWNTO 0) := "111111";
32    SIGNAL fb : STD_LOGIC;
33    SIGNAL LFSR_reg: unsigned(5 DOWNTO 0);
34    SIGNAL LFSR_next: unsigned(5 DOWNTO 0);
35
36  BEGIN
37    -- register
38    PROCESS (clk, reset)
39    BEGIN
40      IF (reset = '1') THEN
41        LFSR_reg <= SEED;
42        s_u_reg <= (OTHERS => '0');
43        m_u_reg <= (OTHERS => '0');
44
45        s_d_reg <= (OTHERS => '0');
46        m_d_reg <= (OTHERS => '0');
47
48        c_u_reg <= (OTHERS => '0');
49        c_d_reg <= (OTHERS => '0');
50      ELSIF (rising_edge(clk)) THEN
51        LFSR_reg <= LFSR_next;
52        c_u_reg <= c_u_next;
53        c_d_reg <= c_d_next;
54        s_u_reg <= s_u_next;
55        s_d_reg <= s_d_next;
56        m_u_reg <= m_u_next;
57        m_d_reg <= m_d_next;
58      END IF;
59    END PROCESS;
60
61    fb <= LFSR_reg(5) XOR LFSR_reg(0);
62
63    LFSR_next <= SEED WHEN LFSR_reg = CONST_RESET
64    ELSE
65      fb & LFSR_reg(5 DOWNTO 1);
66
67    c_en <= '1' WHEN LFSR_reg = CONST_RESET

```

```

68         ELSE
69         '0';
70
71     s_en <= '1' WHEN c_d_reg = 9 AND c_u_reg = 9 AND c_en = '1' ELSE
72     '0';
73
74     m_en <= '1' WHEN s_d_reg = 5 AND s_u_reg = 9 AND s_en = '1' ELSE
75     '0';
76
77     -- next-state logic/output logic for centisecond units
78     c_u_next <= (OTHERS => '0') WHEN (c_u_reg = 9 AND c_en = '1') ELSE
79     c_u_reg + 1 WHEN c_en = '1' ELSE
80     c_u_reg;
81
82     -- next-state logic/output logic for centisecond tens
83     c_d_next <= (OTHERS => '0') WHEN (c_d_reg = 9 AND c_u_reg = 9 AND c_en
= '1') ELSE
84     c_d_reg + 1 WHEN (c_u_reg = 9 AND c_en = '1') ELSE
85     c_d_reg;
86
87     -- next-state logic/output logic for second units
88     s_u_next <= (OTHERS => '0') WHEN (s_u_reg = 9 AND s_en = '1') ELSE
89     s_u_reg + 1 WHEN s_en = '1' ELSE
90     s_u_reg;
91
92     -- next-state logic/output logic for second tens
93     s_d_next <= (OTHERS => '0') WHEN (s_d_reg = 5 AND s_u_reg = 9 AND s_en
= '1') ELSE
94     s_d_reg + 1 WHEN (s_u_reg = 9 AND s_en = '1') ELSE
95     s_d_reg;
96
97     -- next-state logic/output logic for minute units
98     m_u_next <= (OTHERS => '0') WHEN (m_u_reg = 9 AND m_en = '1') ELSE
99     m_u_reg + 1 WHEN m_en = '1' ELSE
100    m_u_reg;
101
102     -- next-state logic/output logic for minute tens
103     m_d_next <= (OTHERS => '0') WHEN (m_d_reg = 5 AND m_u_reg = 9 AND m_en
= '1') ELSE
104     m_d_reg + 1 WHEN (m_u_reg = 9 AND m_en = '1') ELSE
105     m_d_reg;
106
107     -- output logic
108     cen_u <= STD_LOGIC_VECTOR(c_u_reg);
109     cen_d <= STD_LOGIC_VECTOR(c_d_reg);
110
111     sec_u <= STD_LOGIC_VECTOR(s_u_reg);
112     sec_d <= STD_LOGIC_VECTOR(s_d_reg);
113
114     min_u <= STD_LOGIC_VECTOR(m_u_reg);
115     min_d <= STD_LOGIC_VECTOR(m_d_reg);
116 END single_clock_arch;

```