Relatório: Atividade 5

Camila Beatriz da Silva - RA: 2103214

Filipe Augusto Parreira Almeida - RA: 2320622

23 de maio de 2024

Conteúdo

1 Resumo

O presente relatório descreve o processo de desenvolvimento de um algoritmo que utiliza a estrutura de códigos sequenciais em **VHDL**, mais especificamente a estrutura *process*, que permite a execução de cada comando sequencialmente, algo que não é possível utilizando estruturas de código concorrente. O algoritmo desenvolvido utiliza como saída os leds da placa de desenvolvimento DE10-Lite, onde é simulado um comportamento semelhante ao de um bastão de LED com efeito cascata (tal analogia será melhor entendida na apresentação do resultado); para essa implementação, foi utilizado o *clock* presente na placa, e com sua manipulação definimos a velocidade em que o evento simulado ocorrerá.

2 Introdução

Para melhor entendimento dos processos que envolvem a implementação dos requisitos exigidos, há de se compreender os conceitos de código sequencial. A estrutura de código sequencial é utilizada para implementar circuitos sequenciais e também circuitos combinacionais. O código sequencial deve ser escrito dentro de um processo (***PROCESS***) ou então de um subprograma (***FUNCTION***). No presente relatório será abordado somente a estrutura de processos, pois o conceito de subprograma foge do escopo do mesmo.

2.1 PROCESS

A estrutura ***PROCESS*** permite a execução de código sequencial dentro da região principal de código, ***ARCHITECTURE***. O fato de ser sequencial permite que a estrutura de algoritmo seja semelhante à de uma linguagem de programação "convencional" (C, Python, Java), essa aproximação traz consigo métodos semelhantes aos das linguagens de programação usuais; logo, temos que em uma estrutura ***PROCESS*** é permitido somente as instruções *IF*, *CASE*, *LOOP* e *WAIT*.

Usualmente, a estrutura *PROCESS* tem a seguinte sintaxe:

```
1 [label:] PROCESS [(sensitivy_list)] [IS]
2   [declarative_part]
3 BEGIN
4   sequential_statements_parts
5 END PROCESS [label];
```

Código 1: Sintaxe PROCESS

Na parte declarativa é possível definir diversos parâmetros, porém, para o escopo deste relatório é útil uma introdução para dois tipos, as variáveis e os sinais. Para um melhor entendimento das diferenças entre entre esses dois tipos, segue uma tabela que denota essas diferenças:

Figura 1: Variáveis x Sinais

Regra	SIGNAL	VARIABLE
1. Local de declaração	ENTITY, ARCHITECTURE, PACKAGE ou BLOCK (proibido declarar em código seqüencial)	Somente em código sequencial (PROCESS ou subprograma), exceto quando é SHARED VARIABLE (frequentemente declarada na ARCHITECTURE)
2. Escopo	Pode ser global (visível no código inteiro)	Sempre local (visível somente no código seqüencial correspondente), exceto quando é SHARED VARIABLE (pode ser global, mas modificada por só um código seq.)
3. Atualização	Em código sequencial, o novo valor somente estará pronto ao final da presente execução do código	Atualização imediata (o novo valor pode ser usado na próxima linha de código)
4. Operador de assinalamento	Valores são assinalados usando "<=" Exemplo: sig <= 5;	Valores são assinalados usando ":=" Exemplo: var := 5;
5. Assinalamentos múltiplos	Somente um assinalamento é permitido	Aceita múltiplos assinalamentos (porque os mesmos são atualizados imediatamente)
6. Inferência de registradores	Flip-flops são inferidos quando um assinalamento a um sinal é feito na transição (borda) de outro sinal	Flip-flops são inferidos quando um assinalamento a uma variável é feito na transição (borda) de um sinal e o valor desta variável afeta o valor de algum sinal

Fonte: (PEDRONI, 2010)

Dentro da estrutura *PROCESS*, temos que suas instruções permitidas tem a seguinte sintaxe:

2.1.1 IF

Sua lógica, se comparada com a mesma instrução em linguagens convencionais, é semelhante, portanto, se comporta como uma estrutura condicional, podendo ter condições sequenciais, ***ELSIF***. Segue sua sintaxe:

```
[label:] IF conditions THEN
  assignments;
ELSIF conditions THEN
  assignments;

ELSE
  assignments;
END IF [label];
```

Código 2: Sintaxe IF

2.1.2 WAIT

Semelhante ao *IF*, porém menos utilizada, a instrução ***WAIT*** tem como principal uso a construção de formas de onda para simulação. Ela possui três formatos:

```
   - O processo aguarda at  que a condi  o seja atendida
[label:] WAIT UNTIL conditions;
   - O processo aguarda at  que um certo sinal mude de valor
[lable:] WAIT ON signals;
-- O processo aguarda at  que uma certa quantidade de tempo transcorra
[label:] WAIT FOR time;
```

Código 3: Sintaxe WAIT

2.1.3 CASE

Funcionamento semelhante a de uma estrutura de *IF's* sequenciais, é utilizado para facilitar a criação de circuitos combinacionais (tabelas-verdade) dentro de um processo. Funciona da mesma forma que a instrução ***SELECT***, utilizada em códigos concorrentes, porém, a palavra-chave ***OTHERS*** tem mais utilidade. Segue a sintaxe referente a instrução ***CASE***:

```
[label:] CASE expression IS
  WHEN value => assignments;
  WHEN value => assignments;

END CASE;

  LOOP
```

Código 4: Sintaxe CASE

Sua lógica permite a criação de múltiplas instâncias das mesmas atribuições, funcionando de forma semelhante a instrução ***GENERATE*** de códigos concorrentes. Ela possui quatro formas de *loop*:

```
   - FOR
[label:] FOR identifier IN range LOOP
(sequential_statements)
END LOOP [label];

   - WHILE
[label:] WHILE condition LOOP
(sequential_statements)
END LOOP [label];

-- EXIT
[label:] [FOR identifier IN range] LOOP

[exit_label] EXIT [loop_label] [WHEN condition];

END LOOP [loop_label];

   - NEXT
[label:] [FOR identifier IN range] LOOP

[next_label] NEXT [loop_label] [WHEN condition];

END LOOP [loop_label];
```

Código 5: Sintaxe tipos CASE

2.1.4 Exemplo Código Sequencial

Para uma melhor compreensão de todos os conceitos envolvendo códigos sequenciais, temos um exemplo que conta o número total de zeros à esquerda em um vetor de N bits (PEDRONI, 2010). Para isso, é utilizado um ***PROCESS*** que tem em sua lista de sensitividade, o vetor de bits de entrada **x**, é declarado internamente a variável *temp* para auxiliar na lógica, e dentro de sua estrutura é utilizando o laço repetição (*LOOP*) ***FOR***, onde dentro dele foi utilizado o comando ***EXIT***. Tal exemplo tem como intuito apresentar a estrutura completa de um código sequencial, caso se queira saber mais sobre a lógica utilizada no mesmo, consultar (PEDRONI, 2010, p. 456). Segue o código completo referente ao exemplo:

```
   ------------------------------------------------------------
ENTITY leading_zeros IS
  GENERIC (N: INTEGER := 8);
  PORT (x: IN BIT_VECTOR(N-1 DOWNTO 0);
    y: OUT NATURAL RANGE 0 TO N);
END ENTITY;
   ------------------------------------------------------------
ARCHITECTURE behavioral OF leading_zeros IS
BEGIN
  PROCESS (x)
VARIABLE temp: NATURAL RANGE 0 TO N;
BEGIN
temp := 0;
FOR i IN   x RANGE   LOOP
EXIT WHEN x(i)=   1   ;
```

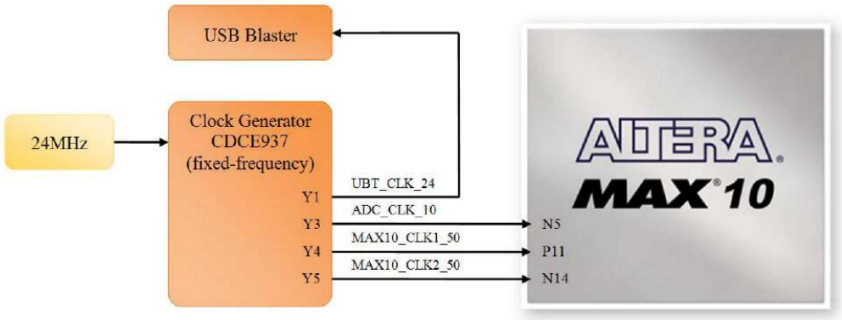
```
16 temp := temp + 1;
17 END LOOP;
18 y <= temp;
19 END PROCESS;
20 END ARCHITECTURE;
21    ---------------------------------------------------------
```

Código 6: Exemplo Código Sequencial

2.2 CLOCK

Para um melhor entendimento do processo de *loop* e temporização utilizado na implementação, foi utilizado o *clock* presente na placa de desenvolvimento. A placa de desenvolvimento DE10-Lite possui dois *clocks* de 50 MHz utilizados para implementação lógica, um *clock* de 10 MHz para conversão analógico digital, e um de 24 MHz conectado ao microcontrolador USB do USB Blaster. Abaixo segue uma representação em diagrama de blocos do sistema de *clocks* da placa de desenvolvimento e também uma tabela dos pinos disponíveis ao usuário.

Figura 2: Diagrama de Blocos Clock - DE10-Lite



Fonte: (TERASIC TECHNOLOGIES, 2020)

Figura 3: Tabela Pinos Clock - DE10-Lite

Signal Name	FPGA Pin No.	Description	I/O Standard
ADC_CLK_10	PIN_N5	10 MHz clock input for ADC (Bank 3B)	3.3-V LVTTL
MAX10_CLK1_50	PIN_P11	50 MHz clock input(Bank 3B)	3.3-V LVTTL
MAX10_CLK2_50	PIN_N14	50 MHz clock input(Bank 3B)	3.3-V LVTTL

Fonte: (TERASIC TECHNOLOGIES, 2020)

3 Implementação

A implementação em si foi feita com base no problema apresentado para a atividade atual. O problema consistia em, utilizando os LEDs presentes na placa de desenvolvimento, fazer os LEDs ligarem um por um, mantendo o LED anterior aceso, e quando todos os LEDs estiverem acesos, irem apagando um por um, assim formando um *loop*.

3.1 Algoritmo VHDL

A parte do código referente a declaração de bibliotecas/pacotes continua sendo a mesma dos relatórios anteriores, portanto ela será omitida, sendo apresentada somente no código completo.

3.1.1 Entidade

A entidade contém a declaração de uma constante global na seção ***GENERIC***, tal constante representa a frequência do *clock* utilizado da placa de desenvolvimento; há também a declaração das portas de entrada e saída utilizadas durante todo o código, como entrada temos o *clock* (**clk**), que é a entrada ligada ao pino referente ao *clock*; e temos o ***enable***, ***reset*** e os **set_velocidade**, que vai de 1 até o 5, o *enable* os *set_velocidade* são ***switches***, já o *reset* é um ***push button***; as entradas *enable*, *reset* e os *set_velocidade* assumem o **mesmo tipo**, que é ***STD_LOGIC***. Dentro da declaração das portas é declarada a saída, que é um vetor de tamanho 10 representado todos os LEDs da placa de desenvolvimento.

Abaixo segue o código referente a declaração da entidade:

```
1  -- Declara  o Entidade
2  ENTITY atividade_5 IS
3      GENERIC (
4        -- Constante de Frequencia de clock (Hz)
5          f_clk: integer := 50_000_000);
6      PORT (
7          clk: in STD_LOGIC;
8          ledS: out STD_LOGIC_VECTOR(9 DOWNTO 0);
9        enable, set_velocidade1, set_velocidade2, set_velocidade3,
   ↪ set_velocidade4, set_velocidade5, reset: in STD_LOGIC);
10 END ENTITY;
```

Código 7: Declaração da Entidade

3.1.2 Arquitetura

Na parte de arquitetura é onde se encontra basicamente toda a lógica empregada para a resolução do desafio proposto. A arquitetura, definida como *main*, tem declarado em seu interno a constante **tempo_piscar** que é um inteiro, e recebe metade do valor original do *clock*, essa atribuição representa meio segundo com base no *clock* da placa de desenvolvimento.

Em sua estrutura propriamente dita é composta somente pelo processo, que engloba toda a lógica. O processo em si tem em sua lista de sensibilidade três parâmetros, o *clock*, o *enable* e o *reset*. Logo abaixo é declarado sua lista de variáveis internas, temos:

- **contador**: utilizado para controlar o tempo de cada operação, acender ou apagar os leds.

- **contador_leds**: variável responsável por controlar os LEDs que vão acender.
- **contador_leds_apaga**: variável responsável por controlar os LEDs que vão apagar.
- **velocidade**: esta variável é tem como função servir como fator de divisão da tempo_piscar, operação esta utilizada para incrementar a variável contadora.

Na estrutura interna do *PROCESS* temos diversos *IF's* e *ELSIF's*, basicamente, o primeiro *IF* é o responsável por detectar se o botão de *reset* foi pressionado, caso sim, ele reseta todas as variáveis responsáveis pelo piscar dos LEDs. O *ELSIF* abaixo dele tem como função verificar se o "sistema" está ativo, caso esteja, e seja a borda de subida do *clock*, ele entra na estrutura de controle e executa seu código interno. E o último *ELSIF* mais externo é o responsável por verificar se o "sistema" está inativo, caso esteja, ele basicamente apaga todos os *LEDs* e reseta todos os contadores.

As estruturas internas dos *ELSIF's* estão melhor explicadas na própria estrutura de código referente a arquitetura, essa estrutura é denotada logo abaixo:

```
1  -- Arquitetura
2  ARCHITECTURE main OF atividade_5 IS
3      CONSTANT tempo_piscar: INTEGER := f_clk/2; -- Meio segundo
4  BEGIN
5
6      PROCESS(clk, enable, reset)
7          VARIABLE contador: integer RANGE 0 TO f_clk'high; -- Vari vel
       ↪ contadora
8        VARIABLE contador_leds: integer RANGE 0 TO 10 := 0;
9        VARIABLE contador_leds_apaga: integer RANGE 0 TO 10 := 0;
10       VARIABLE velocidade: integer RANGE 1 TO 20;
11     BEGIN
12
13       -- Resetar: Apaga todos os leds e reseta o loop
14       IF reset = '0' THEN
15         contador_leds := 0;
16         contador_leds_apaga := 0;
17         leds <= (others => '0');
18
19       -- Com enable ativo e borda de subida do clock
20       ELSIF rising_edge(clk) and enable = '1' THEN
21
22         -- Incrementa a variavel contador, enquanto ela for menor que
       ↪ uma parcela do clock
23         -- Essa parcela    alterada, de acordo com a variavel
       ↪ velocidade, quanto menor a parcela mais r pido
24           -- vai ser a velocidade do loop
25         IF contador < (tempo_piscar / velocidade) THEN
26           contador := contador + 1;
27
28         -- Quando o contador atinge o valor da parcela do clock
       ↪ verificado se todos os leds acenderam
29         -- ent o ele recome a o loop apagando os leds
30         ELSIF contador_leds = 10 THEN
31           leds(contador_leds_apaga) <= '0';
```

```
32            contador := 0;
33            contador_leds_apaga := contador_leds_apaga + 1;
34            IF contador_leds_apaga = 10 THEN
35              contador_leds := 0;
36              contador_leds_apaga := 0;
37            END IF;
38
39          -- Controla, verificando se o led atual e o proximo led esta
    ↪ aceso, se sim,    aceso o led atual
40          ELSIF not leds(contador_leds) and not leds(contador_leds + 1)
    ↪ THEN
41            leds(contador_leds) <= '1';
42            contador := 0;
43            contador_leds := contador_leds + 1;
44          END IF;
45
46          -- Por meio da ativa  o dos switches    definido o valor da
    ↪ variavel velocidade, em caso de mais
47            -- de um switch estar aceso, ele seta a velocidade do
    ↪ primeiro switch da esquerda para a direita
48          IF set_velocidade1 THEN
49            velocidade := 4;
50          ELSIF set_velocidade2 THEN
51            velocidade := 8;
52          ELSIF set_velocidade3 THEN
53            velocidade := 12;
54          ELSIF set_velocidade4 THEN
55            velocidade := 16;
56          ELSIF set_velocidade5 THEN
57            velocidade := 20;
58          ELSE
59            velocidade := 2;
60          END IF;
61
62        -- Verifica se o switch de enable esta ativo, se sim, ele apaga
    ↪ os leds e reseta o loop
63        ELSIF rising_edge(clk) and enable = '0' THEN
64          leds <= (others => '0');
65          contador_leds := 0;
66          contador_leds_apaga := 0;
67        END IF;
68
69      end process;
70
71  END ARCHITECTURE;
```

Código 8: Declaração da Arquitetura

3.1.3 Código Completo

Abaixo segue todo o código descrito anteriormente, contendo as partes de declaração de bibliotecas/pacotes, declaração de entidade e declaração de arquitetura:

```vhdl
-- Declara  o de Biblioteca
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declara  o Entidade
ENTITY atividade_5 IS
    GENERIC (
      -- Constante de Frequencia de clock (Hz)
        f_clk: integer := 50_000_000);
    PORT (
        clk: in STD_LOGIC;
        ledS: out STD_LOGIC_VECTOR(9 DOWNTO 0);
      enable, set_velocidade1, set_velocidade2, set_velocidade3,
    ↪ set_velocidade4, set_velocidade5, reset: in STD_LOGIC);
END ENTITY;

-- Arquitetura
ARCHITECTURE main OF atividade_5 IS
    CONSTANT tempo_piscar: INTEGER := f_clk/2; -- Meio segundo
BEGIN

    PROCESS(clk, enable, reset)
        VARIABLE contador: integer RANGE 0 TO f_clk'high; -- Vari vel
    ↪ contadora
      VARIABLE contador_leds: integer RANGE 0 TO 10 := 0;
      VARIABLE contador_leds_apaga: integer RANGE 0 TO 10 := 0;
      VARIABLE velocidade: integer RANGE 1 TO 20;
    BEGIN

      -- Resetar: Apaga todos os leds e reseta o loop
      IF reset = '0' THEN
        contador_leds := 0;
        contador_leds_apaga := 0;
        leds <= (others => '0');

      -- Com enable ativo e borda de subida do clock
      ELSIF rising_edge(clk) and enable = '1' THEN

        -- Incrementa a variavel contador, enquanto ela for menor que
    ↪ uma parcela do clock
        -- Essa parcela    alterada, de acordo com a variavel
    ↪ velocidade, quanto menor a parcela mais r pido
          -- vai ser a velocidade do loop
        IF contador < (tempo_piscar / velocidade) THEN
          contador := contador + 1;

        -- Quando o contador atinge o valor da parcela do clock
    ↪ verificado se todos os leds acenderam
        -- ent o ele recome a o loop apagando os leds
        ELSIF contador_leds = 10 THEN
          leds(contador_leds_apaga) <= '0';
```

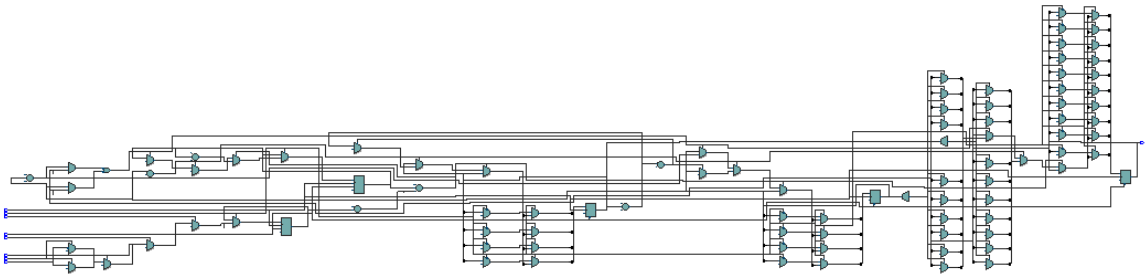
```
47            contador := 0;
48            contador_leds_apaga := contador_leds_apaga + 1;
49            IF contador_leds_apaga = 10 THEN
50              contador_leds := 0;
51              contador_leds_apaga := 0;
52            END IF;
53
54          -- Controla, verificando se o led atual e o proximo led esta
    ↪ aceso, se sim,    aceso o led atual
55          ELSIF not leds(contador_leds) and not leds(contador_leds + 1)
    ↪ THEN
56            leds(contador_leds) <= '1';
57            contador := 0;
58            contador_leds := contador_leds + 1;
59          END IF;
60
61          -- Por meio da ativa  o dos switches    definido o valor da
    ↪ variavel velocidade, em caso de mais
62            -- de um switch estar aceso, ele seta a velocidade do
    ↪ primeiro switch da esquerda para a direita
63          IF set_velocidade1 THEN
64            velocidade := 4;
65          ELSIF set_velocidade2 THEN
66            velocidade := 8;
67          ELSIF set_velocidade3 THEN
68            velocidade := 12;
69          ELSIF set_velocidade4 THEN
70            velocidade := 16;
71          ELSIF set_velocidade5 THEN
72            velocidade := 20;
73          ELSE
74            velocidade := 2;
75          END IF;
76
77        -- Verifica se o switch de enable esta ativo, se sim, ele apaga
    ↪ os leds e reseta o loop
78        ELSIF rising_edge(clk) and enable = '0' THEN
79          leds <= (others => '0');
80          contador_leds := 0;
81          contador_leds_apaga := 0;
82        END IF;
83
84      end process;
85
86  END ARCHITECTURE;
```

Código 9: Código Completo

3.2 Diagrama RTL

Com a compilação do código utilizando o ***Quartus***, é possível que se obtenha o diagrama ***RTL*** referente a implementação, dessa forma têm-se sua representação em formas de circuitos. Abaixo é apresentado o diagrama *RTL* gerado com base no código previamente analisado:

Figura 4: Diagrama RTL



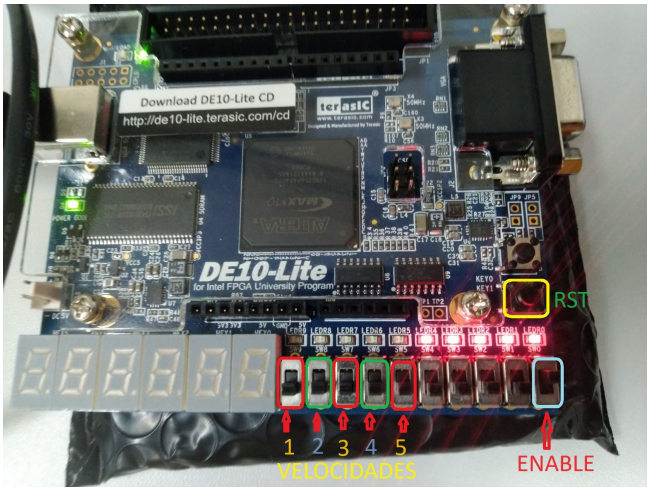
Fonte: Autoria Própria

Para uma melhor visualização do diagrama, também será anexado, no final deste arquivo, o arquivo com sua representação, permitindo assim que se faça zoom.

4 Conclusão

Para que seja possível a visualização dos resultados de toda a implementação, é anexado juntamente com este arquivo o vídeo que demonstra o funcionamento de todo o código *VHDL* implementado na placa de desenvolvimento. Sendo testado então todas as funcionalidades implementadas. Abaixo segue uma foto da placa de desenvolvimento indicando o mapeamento das portas de entrada:

Figura 5: Mapeamento de Portas



Fonte: Autoria Própria

Conclui-se que todas as funcionalidades idealmente pensadas para a solução do desafio proposto, foram atendidas na implementação, sendo também acrescentada a funcionalidade de controle de velocidade.

Referências

PEDRONI, Volnei A. **Eletrônica digital moderna e VHDL**. Rio de Janeiro, RJ: Elsevier, 2010. P. 619. ISBN 9788535234657.

TERASIC TECHNOLOGIES. **Manual da Placa de Desenvolvimento DE10-Lite**. [S.l.], 2020. Disponível em: `https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=China&No=1021&FID=a13a2782811152b477e60203d34b1baa`.