UFSC - UNIVERSIDADE FEDERAL DE SANTA CATARINA
CTC - CENTRO TECNOLÓGICO
INE - DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Sistemas Digitais

Relatório da Prática 3

—

Entrega marcada para 23/11/2022

PROFESSORA:

Cristina Meinhardt

EQUIPE:

Marcos Laureano Lacava
Otávio Augusto de Santana Jatobá
Tális Breda
Gabriela de Moura

Índice do Relatório

1. Introdução

Este documento serve de relatório para o trabalho de desenvolvimento de projetos de multiplicadores, tal como para a análise desses componentes no que tange à estrutura, uso de componentes, análise de tempo de execução e análise de comportamento.

Ao longo do relatório desenvolvemos a análise com o objetivo de classificar os modelos de multiplicadores em sua capacidade, área e velocidade e também para compará-los nesses critérios.

Neste projeto, abordamos dois algoritmos de multiplicação em nossos modelos: o algoritmo da soma sequencial e o algoritmo de Vedic.

2. Conceitos Fundamentais

2.1. FSMs

Uma máquina de estados finita (FSM - do inglês Finite State Machine) ou autômato finito é um modelo matemático usado para representar programas de computadores ou circuitos lógicos. O conceito é concebido como uma máquina abstrata que deve estar em um de um número finito de estados. A máquina está em apenas um estado por vez, este estado é chamado de estado atual. Um estado armazena informações sobre o passado, isto é, ele reflete as mudanças desde a entrada num estado, no início do sistema, até o momento presente. Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Uma ação é a descrição de uma atividade que deve ser realizada num determinado momento.

Tabela de Transições

Estado Inicial	Próximo Estado	Condition
S0	S0	not(inicio)
S0	S1	inicio
S1	S2	
S2	S5	A or B
S2	S3	not(A or B)
S3	S4	
S4	S2	
S5	S0	

Esta é a tabela de transição de estados. Ela possui uma coluna referente ao estado atual, uma referente ao próximo estado e uma com a condição de troca de estado.

Mapa de Estados

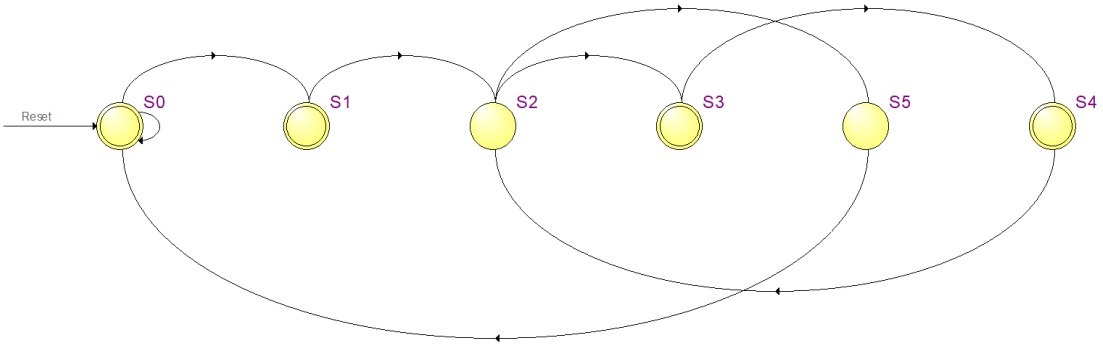


Figura 1: Mapa da máquina de estados, com setas de transição, gerada pelo Quartus

2.3. Algoritmos de Multiplicação

Algoritmos são sequências de instruções bem definidas utilizadas para solução de um problema específico. No nosso caso, o problema é encontrar o resultado da multiplicação entre dois números e, para isso, podemos utilizar diversos algoritmos que atacam esse problema.

Descrição do algoritmo do multiplicador V1 em pseudolinguagem:

```
0. Início
1. A ← entA; B ← entB; P ← 0; pronto ← 0;
2. Se B ≠ 0 então
3. Enquanto A ≠ 0 faça {
4. P ← P + B;
5. A ← A - 1; }
6. pronto ← 1;
```

Esse algoritmo funciona por receber duas entradas de dados A e B, e somar B a si mesmo A vezes. Uma vez que A chega a 0, o circuito termina, sobe uma *flag* de concluído, e retorna ao estado inicial. Assim que ele é concluído, obtemos o resultado da multiplicação entre A e B.

Iremos utilizar esse algoritmo na descrição do hardware que será aplicado para realização do cálculo em si, como demonstrado na arquitetura dos nossos multiplicadores.

3. Arquitetura Proposta

3.1. Multiplicador V1

3.1.1. **Toplevel:**

```
library IEEE;
use IEEE.std_logic_1164.all;

entity multiplicador is
generic(
	N: integer := 16 -- Mude o número de bits
	);

port(
	CLK, RST, INIT  :   in  std_logic;
	A, B            :     in  std_logic_vector(N-1 downto 0);
	R               :   out std_logic;
	S               :   out std_logic_vector(N-1 downto 0)
	);

end entity;

architecture arch of multiplicador is

	component bc is
	port (
	Reset, clk, inicio        : IN STD_LOGIC;
	Az, Bz                    : IN STD_LOGIC;
	pronto                    : OUT STD_LOGIC;
	ini, CA, dec, CP          : OUT STD_LOGIC
	);
	end component;

	component bo is
	generic (N: integer);
	port (
	clk                       : IN STD_LOGIC;
	ini, CP, CA, dec          : IN STD_LOGIC;
	entA, entB                : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
	Az, Bz                    : OUT STD_LOGIC;
	saida, conteudoA, conteudoB : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0)
	);
	end component;

	signal Az, Bz, ini, CA, dec, CP : std_logic;

begin

	bloco_controle:     bc          PORT MAP (RST, CLK, INIT,
	                          Az, Bz,
	                          R,
	                          ini, CA, dec, CP);

	bloco_operativo:    bo          GENERIC MAP (N)
	                          PORT MAP (CLK,
```

```
                                ini, CP, CA, dec,
                                A, B,
                                Az, Bz,
                                S);

end architecture; -- arch
```

3.1.2. **Componentes:**

Bloco de Controle:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bc IS
PORT (Reset, clk, inicio : IN STD_LOGIC;
      Az, Bz : IN STD_LOGIC;
      pronto : OUT STD_LOGIC;
      ini, CA, dec, CP: OUT STD_LOGIC );
END bc;

ARCHITECTURE estrutura OF bc IS
      TYPE state_type IS (S0, S1, S2, S3, S4, S5 );
      SIGNAL state: state_type;
BEGIN
      -- Logica de proximo estado (e registrador de estado)
      PROCESS (clk, Reset)
      BEGIN
      if(Reset = '1') THEN
            state <= S0 ;
      ELSIF (clk'EVENT AND clk = '1') THEN
            CASE state IS
            WHEN S0 =>
                  if (inicio = '0') then
                        state <= S0;
                  else
                        state <= S1;
                  end if;

            WHEN S1 =>
                  state <= S2;

            WHEN S2 =>
                  if (Az = '1' or Bz = '1') then
                        state <= S5;
                  else
                        state <= S3;
                  end if;

            WHEN S3 =>
                  state <= S4;

            WHEN S4 =>
                  state <= S2;

            WHEN S5 =>
```

```
                state <= S0;

            END CASE;
        END IF;
        END PROCESS;

        -- Logica de saida
        PROCESS (state)
        BEGIN
        CASE state IS
            WHEN S0 =>
            ini <= '0';
            CA <= '0';
            dec <= '0';
            CP <= '0';
            pronto <= '1';

            WHEN S1 =>
            ini <= '1';
            CA <= '1';
            dec <= '0';
            CP <= '0';
            pronto <= '0';

            WHEN S2 =>
            ini <= '0';
            CA <= '0';
            dec <= '0';
            CP <= '0';
            pronto <= '0';

            WHEN S3 =>
            ini <= '0';
            CA <= '0';
            dec <= '0';
            CP <= '1';
            pronto <= '0';

            WHEN S4 =>
            ini <= '0';
            CA <= '1';
            dec <= '1';
            CP <= '0';
            pronto <= '0';

            WHEN S5 =>
            ini <= '0';
            CA <= '0';
            dec <= '0';
            CP <= '0';
            pronto <= '0';

        END CASE;
        END PROCESS;
END estrutura;
```

Bloco Operativo:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY bo IS
GENERIC(N: integer);
PORT (clk : IN STD_LOGIC;
      ini, CP, CA, dec : IN STD_LOGIC;
      entA, entB : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      Az, Bz : OUT STD_LOGIC;
      saida, conteudoA, conteudoB : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END bo;

ARCHITECTURE estrutura OF bo IS

      COMPONENT registrador_r IS
      GENERIC (N: integer);
      PORT (clk,  reset, carga : IN STD_LOGIC;
      d : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
      END COMPONENT;

      COMPONENT registrador IS
      GENERIC (N: integer);
      PORT (clk, carga : IN STD_LOGIC;
      d : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
      END COMPONENT;

      COMPONENT mux2para1 IS
      GENERIC (N: integer);
      PORT ( a, b : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
             sel: IN STD_LOGIC;
             y : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
      END COMPONENT;

      COMPONENT somadorsubtrator IS
      GENERIC (N: integer);
      PORT (a, b : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      op: IN STD_LOGIC;
      s : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
      END COMPONENT;

      COMPONENT igualazero IS
      GENERIC (N: integer);
      PORT (a : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      igual : OUT STD_LOGIC);
      END COMPONENT;

      SIGNAL saimux1, saimux2, saimux3, sairegP, sairegA, sairegB, saisomasub: STD_LOGIC_VECTOR (N-1
DOWNTO 0);

BEGIN
      mux1: mux2para1 GENERIC MAP (N)
      PORT MAP (saisomasub, entA, ini, saimux1);

      regP: registrador_r GENERIC MAP (N)
```

```
    PORT MAP (clk, ini, CP, saisomasub, sairegP);

    regA: registrador GENERIC MAP (N => N)
    PORT MAP (clk, CA, saimux1, sairegA);

    regB: registrador GENERIC MAP (N => N)
    PORT MAP (clk, ini, entB, sairegB);

    mux2: mux2para1 GENERIC MAP (N => N)
    PORT MAP (sairegP, sairegA, dec, saimux2);

    mux3: mux2para1 GENERIC MAP (N => N)
    PORT MAP (sairegB, (0 => '1', others => '0'), dec, saimux3);

    somasub: somadorsubtrator GENERIC MAP (N => N)
    PORT MAP (saimux2, saimux3, dec, saisomasub);

    geraAz: igualazero GENERIC MAP (N => N)
    PORT MAP (sairegA, Az);

    geraBz: igualazero GENERIC MAP (N => N)
    PORT MAP (sairegB, Bz);

    saida <= sairegP; -- Saída do registrador acumulador
    conteudoA <= sairegA;
    conteudoB <= sairegB;

END estrutura;
```

3.1.3. **Análise de Ciclos:**

O número de ciclos do Multiplicador V1 é descrito pela seguinte equação, caso A > 0:

3b + 4

Isto é, **S0 + S1 + B (S2 + S3 + S4) + S2 + S5**

Caso A = 0, o número de ciclos é **4**,

Que pode ser dividido em: **S0 + S1 + S2 + S5**

Analisando estes fatores, percebemos que caso ocorra uma multiplicação entre um A pequeno e um B muito grande, muito tempo será desperdiçado fazendo somas triviais onde um algoritmo um pouco mais robusto poderia economizar tempo.

3.2. Multiplicador V2 (Vedic)

3.2.1. **Toplevel:**

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY multiplicador IS
GENERIC(
      N : integer := 8
);
PORT (CLK, INICIAR: in std_logic;
            A, B: in std_logic_vector(N-1 downto 0);
            R: out std_logic;
            S: out std_logic_vector((2*N)-1 downto 0));
END multiplicador;

ARCHITECTURE estrutura OF multiplicador IS

      COMPONENT bc IS
      PORT (CLK, INICIO : in std_logic;
                  ini, pronto : out std_logic);
      END COMPONENT;

      COMPONENT registrador IS
   GENERIC(N : integer);
      PORT
   (  load, clk: IN STD_LOGIC;
            data : in std_logic_vector (N-1 downto 0);
      q: OUT STD_LOGIC_vector(N-1 downto 0)
   );
      END COMPONENT;

      component fourbitmult IS
      PORT (clk: in std_logic;
                  A, B: in std_logic_vector(3 downto 0);
                  S: out std_logic_vector(7 downto 0));
      END component;

      component eightbitmult IS
      PORT (clk: in std_logic;
                  A, B: in std_logic_vector(7 downto 0);
                  S: out std_logic_vector(15 downto 0));
      END component;

      component sixteenbitmult is
      PORT (clk: in std_logic;
        A, B: in std_logic_vector(15 downto 0);
                  S: out std_logic_vector(31 downto 0));
      end component;

      Signal ini : std_logic;
      Signal A_in, B_in : std_logic_vector(N-1 downto 0);

BEGIN
```

```
	BC1: bc port map (CLK, INICIAR, ini, R);

	regA: registrador generic map (n) port map (ini, CLK, A, A_in);
	regB: registrador generic map (n) port map (ini, CLK, B, B_in);

	multi: if N = 4 generate
		mult4 : fourbitmult port map (CLK, A, B, S);
	end generate multi;
	mult1: if N = 8 generate
		mult8 : eightbitmult port map (CLK, A, B, S);
	end generate mult1;
	mult2: if N = 16 generate
		mult16 : sixteenbitmult port map (CLK, A, B, S);
	end generate mult2;


END estrutura;
```

3.2.1. **Componentes:**

Multiplicador de 4 bits:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fourbitmult IS
PORT (clk: in std_logic := '1';
	A, B: in std_logic_vector(3 downto 0);
	S: out std_logic_vector(7 downto 0));
END fourbitmult;

ARCHITECTURE estrutura OF fourbitmult IS

	component twobitmult IS
	PORT (clk: in std_logic;
		A, B: in std_logic_vector(1 downto 0);
		S: out std_logic_vector(3 downto 0));
	END component;

	component cla4bit is
	Port (A : in STD_LOGIC_VECTOR (3 downto 0);
		B : in STD_LOGIC_VECTOR (3 downto 0);
		Cin: in STD_LOGIC;
		S : out STD_LOGIC_VECTOR (3 downto 0);
		Cout : out STD_LOGIC);
	end component;

	component halfadder is
	port (A, B: in std_logic;
		S, Cout: out std_logic
	  );
```

```
	end component;

	signal b3b2, a3a2, b1b0, a1a0: std_logic_vector(1 downto 0);
	signal smult1, smult2, smult3, smult4: std_logic_vector(3 downto 0);
	signal scla1, scla2: std_logic_vector(3 downto 0);
	signal or2bit, sha1, coutha1, sha2, coutha2, coutcla1, coutcla2: std_logic;

	signal cla2in: std_logic_vector(3 downto 0);

BEGIN

	b3b2 <= B(3 downto 2);
	b1b0 <= B(1 downto 0);
	a3a2 <= A(3 downto 2);
	a1a0 <= A(1 downto 0);

	tbm1: twobitmult port map (clk, b3b2, a3a2, smult1);
	tbm2: twobitmult port map (clk, b3b2, a1a0, smult2);
	tbm3: twobitmult port map (clk, b1b0, a3a2, smult3);
	tbm4: twobitmult port map (clk, b1b0, a1a0, smult4);

	cla1: cla4bit port map (smult2, smult3, '0', scla1, coutcla1);

	cla2in(3 downto 2) <= smult1(1 downto 0);
	cla2in(1 downto 0) <= smult4(3 downto 2);

	cla2: cla4bit port map (cla2in, scla1, '0', scla2, coutcla2);

	or2bit <= coutcla1 or coutcla2;

	ha1: halfadder port map (smult1(2), or2bit, sha1, coutha1);
	ha2: halfadder port map (smult1(3), coutha1, sha2, coutha2);

	process (clk)
	begin
		if rising_edge(clk) then
			S(1 downto 0) <= smult4(1 downto 0);
			S(5 downto 2) <= scla2;
			S(6) <= sha1;
			S(7) <= sha2;
		end if;
	end process;


END estrutura;
```

Multiplicador de 8 bits:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY eightbitmult IS
PORT (clk: in std_logic;
            A, B: in std_logic_vector(7 downto 0);
            S: out std_logic_vector(15 downto 0));
END eightbitmult;

ARCHITECTURE estrutura OF eightbitmult IS

      component fourbitmult IS
      PORT (clk: in std_logic;
                  A, B: in std_logic_vector(3 downto 0);
                  S: out std_logic_vector(7 downto 0));
      END component;

      component cla8bit is
      Port (x_in      :  IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
        y_in      :  IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
        carry_in  :  IN   STD_LOGIC;
        sum       :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0);
        carry_out :  OUT  STD_LOGIC
       );
      end component;

      component halfadder is
      port (A, B: in std_logic;
             S, Cout: out std_logic
        );
      end component;

      signal b7654, a7654, b3210, a3210: std_logic_vector(3 downto 0);
      signal smult1, smult2, smult3, smult4: std_logic_vector(7 downto 0);
      signal scla1, scla2: std_logic_vector(7 downto 0);
      signal or2bit, sha1, coutha1, sha2, coutha2, sha3, coutha3, sha4, coutha4, coutcla1, coutcla2:
std_logic;

      signal cla2in: std_logic_vector(7 downto 0);

BEGIN

      b7654 <= B(7 downto 4);
      b3210 <= B(3 downto 0);
      a7654 <= A(7 downto 4);
      a3210 <= A(3 downto 0);

      mult1: fourbitmult port map (clk, b7654, a7654, smult1);
      mult2: fourbitmult port map (clk, b3210, a7654, smult2);
      mult3: fourbitmult port map (clk, b7654, a3210, smult3);
      mult4: fourbitmult port map (clk, b3210, a3210, smult4);

      cla2in(7 downto 4) <= smult1(3 downto 0);
      cla2in(3 downto 0) <= smult4(7 downto 4);
```

```
	cla1: cla8bit port map (smult2, smult3, '0', scla1, coutcla1);
	cla2: cla8bit port map (cla2in, scla1, '0', scla2, coutcla2);

	or2bit <= coutcla1 or coutcla2;

	ha1: halfadder port map (or2bit, smult1(4), sha1, coutha1);
	ha2: halfadder port map (coutha1, smult1(5), sha2, coutha2);
	ha3: halfadder port map (coutha2, smult1(6), sha3, coutha3);
	ha4: halfadder port map (coutha3, smult1(7), sha4, coutha4);


	process (clk)
	begin
		if rising_edge(clk) then
			S(3 downto 0) <= smult4(3 downto 0);
			S(11 downto 4) <= scla2;
			S(12) <= sha1;
			S(13) <= sha2;
			S(14) <= sha3;
			S(15) <= sha4;
		end if;
	end process;

END estrutura;
```

Multiplicador de 16 bits:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY sixteenbitmult IS
PORT (clk: in std_logic;
		A, B: in std_logic_vector(15 downto 0);
		S: out std_logic_vector(31 downto 0));
END sixteenbitmult;

ARCHITECTURE estrutura OF sixteenbitmult IS

	component eightbitmult IS
	PORT (clk: in std_logic;
			A, B: in std_logic_vector(7 downto 0);
			S: out std_logic_vector(15 downto 0));
	END component;

	component cla16bit is
	PORT (
	  A, B : in std_logic_vector (15 downto 0);
		  Ci: in std_logic;
		  S : out std_logic_vector(15 downto 0);
		  Co, PG, GG : out std_logic);
	end component;

	component halfadder is
	port (A, B: in std_logic;
		S, Cout: out std_logic
	  );
```

```
	end component;

	signal b15to8, a15to8, b7to0, a7to0: std_logic_vector(7 downto 0);
	signal smult1, smult2, smult3, smult4: std_logic_vector(15 downto 0);
	signal scla1, scla2, cla2in: std_logic_vector(15 downto 0);
	signal or2bit, coutcla1, coutcla2, pg1, pg2, gg1, gg2 : std_logic;
	signal sha, coutha: std_logic_vector(8 downto 1);

BEGIN

	b15to8 <= B(15 downto 8);
	b7to0 <= B(7 downto 0);
	a15to8 <= A(15 downto 8);
	a7to0 <= A(7 downto 0);

	mult1: eightbitmult port map (clk, b15to8, a15to8, smult1);
	mult2: eightbitmult port map (clk, b7to0, a15to8, smult2);
	mult3: eightbitmult port map (clk, b15to8, a7to0, smult3);
	mult4: eightbitmult port map (clk, b7to0, a7to0, smult4);

	cla2in <= smult1(7 downto 0) & smult4(15 downto 8);

	cla1: cla16bit port map (smult2, smult3, '0', scla1, coutcla1, pg1, gg1);
	cla2: cla16bit port map (cla2in, scla1, '0', scla2, coutcla2, pg1, gg1);

	or2bit <= coutcla1 or coutcla2;

	ha1: halfadder port map (or2bit, smult1(8), sha(1), coutha(1));

	halfadders : for i in 2 to 8 generate
		ha: halfadder port map (coutha(i-1), smult1(i+7), sha(i), coutha(i));
	end generate halfadders;

	process (clk)
	begin
		if rising_edge(clk) then
			S(7 downto 0) <= smult4(7 downto 0);
			S(23 downto 8) <= scla2;
			S(31 downto 24) <= sha(8 downto 1);
		end if;
	end process;

END estrutura;
```

3.2.3. **Análise de Ciclos:**

O número de ciclos do Multiplicador V2 é descrito pela seguinte equação, caso A > 0:

$\log_2 b$ + **2**

Isto é, **S0 + $\log_2$ b (S1) + S2**

Ou seja, para 8 bits, por exemplo, o número de estados será 5, sendo:

S0 + 3 (S1) + S2

4. Resultados

4.1. Multiplicador V1

4.1.1. **Netlists:**

FSM:

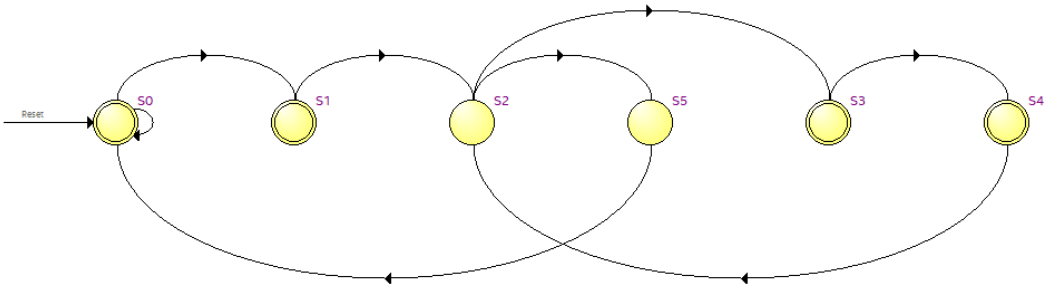


Figura 2: Mapa da máquina de estados, com setas de transição, gerada pelo Quartus

RTL VIEW:

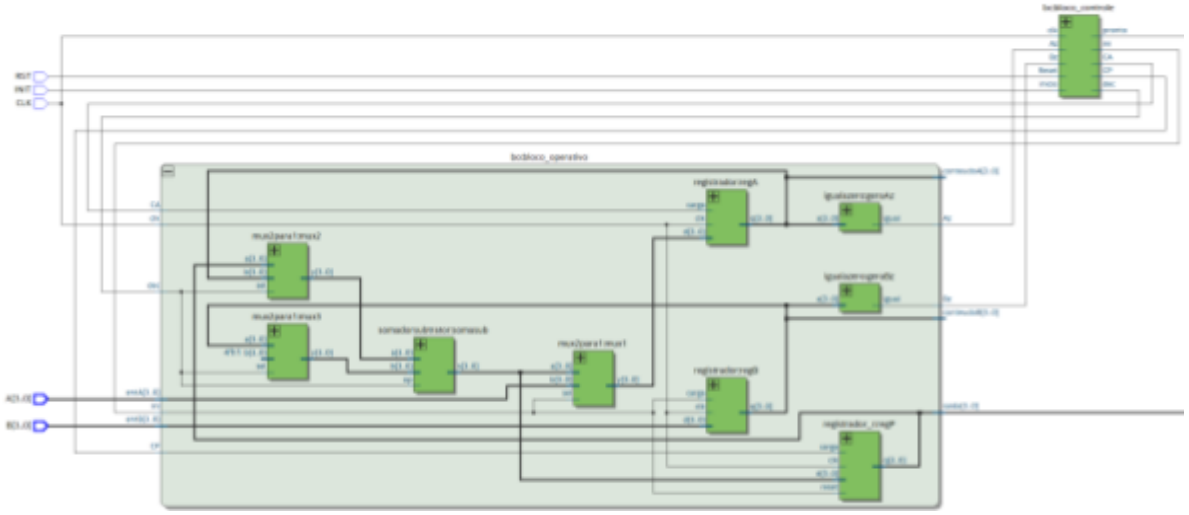


Figura 3: Diagrama RTL do circuito, gerado pelo Quartus

4.1.2. **Dados de Utilização da Placa:**

N = 16:

Family	**Cyclone IV GX**
Total Logic Elements	**83**
> Total combinational functions	81
> Dedicated logic registers	54
Total registers	**54**
Total Pins	**52**

N = 8:

Family	**Cyclone IV GX**
Total Logic Elements	**52**
> Total combinational functions	43
> Dedicated logic registers	30
Total registers	**30**
Total Pins	**30**

N = 4:

Family	**Cyclone IV GX**
Total Logic Elements	**29**
> Total combinational functions	24
> Dedicated logic registers	18
Total registers	**18**
Total Pins	**16**

Aqui vemos um crescimento linear dos componentes do multiplicador, seguindo, mais ou menos, 10 elementos de lógica por bit. Este "preço" barato reflete o comportamento insatisfatório deste bloco acelerador.

4.1.3. Verificação funcional de timing:

Clock to Output Times

	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	S[*]	CLK	9.409	9.475	Rise	CLK
1	S[5]	CLK	9.409	9.475	Rise	CLK
2	S[9]	CLK	8.179	8.147	Rise	CLK
3	S[1]	CLK	8.169	8.138	Rise	CLK
4	S[0]	CLK	7.952	7.836	Rise	CLK
5	S[6]	CLK	7.783	7.686	Rise	CLK
6	S[14]	CLK	7.729	7.659	Rise	CLK
7	S[2]	CLK	7.748	7.644	Rise	CLK
8	S[12]	CLK	7.708	7.605	Rise	CLK
9	S[11]	CLK	7.620	7.519	Rise	CLK
10	S[3]	CLK	7.562	7.485	Rise	CLK
11	S[7]	CLK	7.538	7.417	Rise	CLK
12	S[10]	CLK	7.446	7.377	Rise	CLK
13	S[13]	CLK	7.456	7.372	Rise	CLK
14	S[4]	CLK	7.461	7.368	Rise	CLK
15	S[8]	CLK	7.432	7.340	Rise	CLK
16	S[15]	CLK	7.281	7.163	Rise	CLK
2	R	CLK	8.203	8.254	Rise	CLK

Figura 4: Relatório de tempos de *clock* a saída gerado pelo Quartus para um circuito de 16 bits

Pior tempo de clock a saída para N = 16:

Clock Port	**Data Port (Saída)**	**Fall**
CLK	S[5]	9.475 ns

Clock to Output Times

	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	S[*]	CLK	8.435	8.391	Rise	CLK
1	S[4]	CLK	8.435	8.391	Rise	CLK
2	S[2]	CLK	7.526	7.395	Rise	CLK
3	S[0]	CLK	7.322	7.190	Rise	CLK
4	S[1]	CLK	7.303	7.179	Rise	CLK
5	S[3]	CLK	7.244	7.137	Rise	CLK
6	S[5]	CLK	6.955	6.856	Rise	CLK
7	S[7]	CLK	6.937	6.849	Rise	CLK
8	S[6]	CLK	6.649	6.564	Rise	CLK
2	R	CLK	6.808	6.906	Rise	CLK

Figura 5: Relatório de tempos de *clock* a saída gerado pelo Quartus para um circuito de 8 bits

Pior tempo de clock a saída para N = 8:

Clock Port	Data Port (Saída)	Fall
CLK	S[4]	8.435 ns

Clock to Output Times

	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	S[*]	CLK	7.604	7.577	Rise	CLK
1	S[1]	CLK	7.604	7.577	Rise	CLK
2	S[3]	CLK	6.882	6.798	Rise	CLK
3	S[0]	CLK	6.848	6.721	Rise	CLK
4	S[2]	CLK	6.538	6.469	Rise	CLK
2	R	CLK	6.914	6.999	Rise	CLK

Figura 6: Relatório de tempos de *clock* a saída gerado pelo Quartus para um circuito de 4 bits

Pior tempo de clock a saída para N = 4:

Clock Port	Data Port (Saída)	Fall
CLK	S[1]	7.604 ns

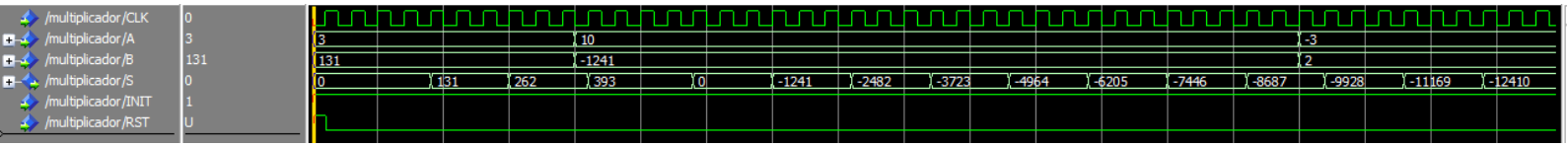
Nos piores casos, temos os seguintes dados:

	Número de ciclos	Período	Frequência
4 bits	34	9.475 ns	131.50 MHz
8 bits	514	8.435 ns	118.55 MHz
16 bits	65539	7.604 ns	95.46 MHz

A análise de timing revela o que já sabíamos sobre o caminho crítico. Entre as únicas saídas, o sinal R e o S (resultado), o resultado é o que demora mais para ficar pronto, pois ele envolve a saída de somadores.

Descrição dos Estímulos:

```
restart -force -nowave
delete wave *
add wave CLK
add wave A
add wave B
add wave S
add wave INIT
add wave RST
force CLK 0 0, 1 10ns -repeat 20ns
force RST 1 1ns, 0 11ns
force INIT 1 0ns
force A 0000000000000011 0ns, 0000000000001010 200ns, 1111111111111101 750ns
force B 0000000010000011 0ns, 1111101100100111 200ns, 0000000000000010 750ns
run 2000ns
```



Imagens da Simulação:

Figura 7: Diagrama de ondas gerado a partir de um testbench para o multiplicador V1 de 16 bits

Comportamento:

O comportamento do nosso multiplicador V1 ocorreu como esperado. Ele recebe 2 entradas, e quando estiver pronto e o valor INICIAR for 1, ele começa a multiplicação. Ele vai adicionando B ao resultado A vezes, enquanto nenhum for 0, momento quando ele fica pronto para multiplicar novamente.

Vemos também uma fraqueza ao multiplicar um B por um A negativo. O bloco irá tratar o A como um número sem sinal quando na verdade é um complemento de 2, e isso fará ele gastar muitos estados somando o valor até milhões de vezes, numa simples conta de -3 x 2. Isso é inviável, no mínimo.

4.2. **Multiplicador V2 (Vedic)**

4.2.1. **Netlists:**

FSM:

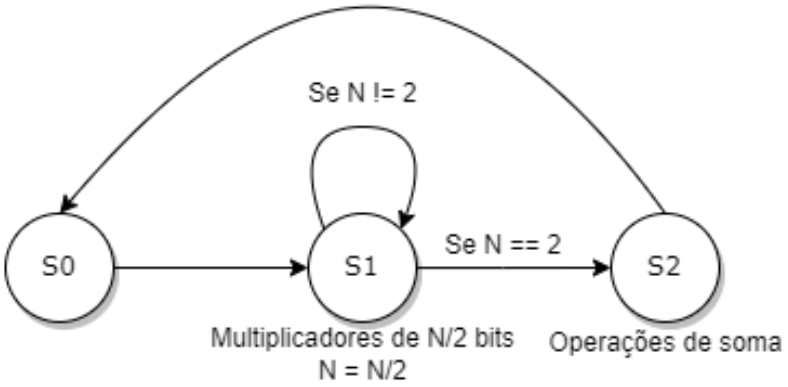


Figura 8: Máquina de estados do multiplicador V2

Bloco Operativo:

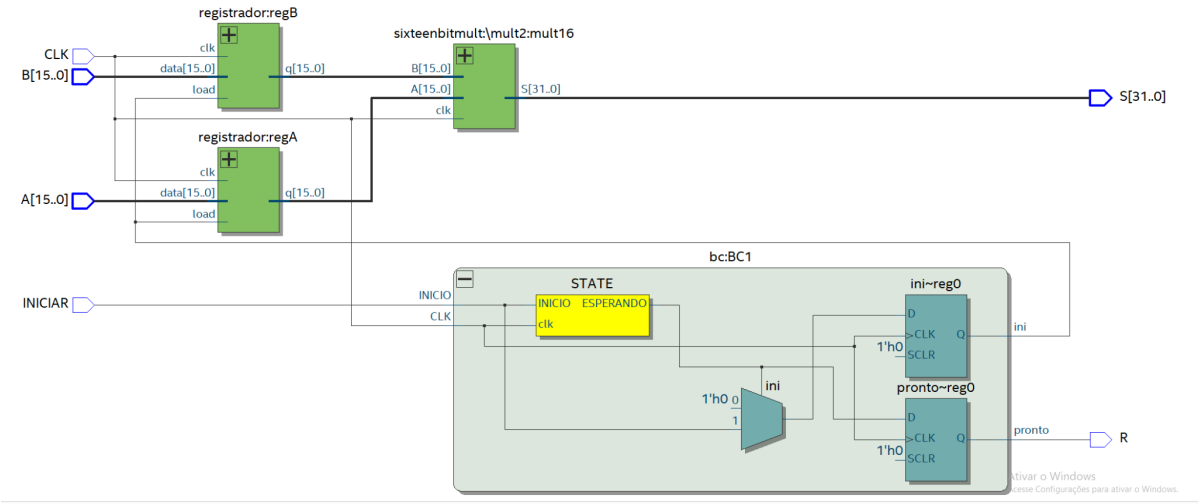


Figura 9: Diagrama RTL do multiplicador V2 gerado pelo Quartus

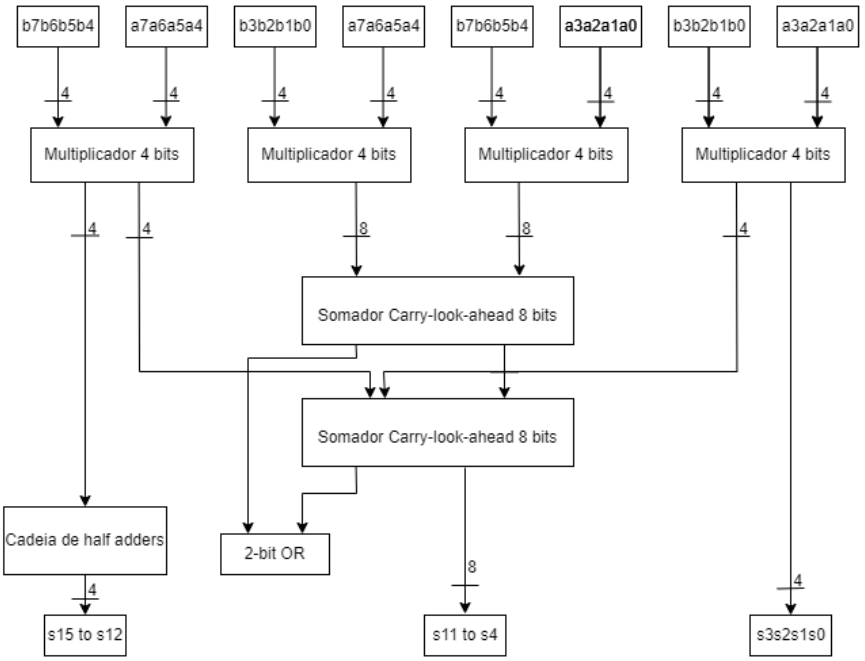


Figura 10: Diagrama do bloco operativo do multiplicador V2

4.2.2. **Dados de Utilização da Placa:**

N = 16:

Family	**Cyclone IV GX**
Total Logic Elements	**1023**
> Total combinational functions	1005
> Dedicated logic registers	480
Total registers	**480**
Total Pins	**67**

N = 8:

Family	**Cyclone IV GX**
Total Logic Elements	**219**
> Total combinational functions	212
> Dedicated logic registers	112
Total registers	**112**
Total Pins	**35**

N = 4:

Family	**Cyclone IV GX**
Total Logic Elements	**48**
> Total combinational functions	45
> Dedicated logic registers	24
Total registers	**24**
Total Pins	**19**

Nota-se, a partir dessa análise, que a quantidade de componentes cresce de forma exponencial de acordo com a quantidade de bits utilizados. Dessa forma, a quantidade de estados necessários é drasticamente reduzida, aumentando a eficiência do multiplicador.

4.2.3. **Verificação funcional de timing:**

N = 16:

Clock to Output Times

	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	S[*]	CLK	9.136	9.120	Rise	CLK
1	S[26]	CLK	9.136	9.120	Rise	CLK
2	S[4]	CLK	9.120	9.104	Rise	CLK
3	S[17]	CLK	8.967	8.860	Rise	CLK
4	S[31]	CLK	8.939	8.854	Rise	CLK
5	S[27]	CLK	8.602	8.545	Rise	CLK
6	S[29]	CLK	8.556	8.489	Rise	CLK
7	S[30]	CLK	8.495	8.449	Rise	CLK
8	S[12]	CLK	8.304	8.331	Rise	CLK
9	S[20]	CLK	8.476	8.325	Rise	CLK
10	S[28]	CLK	8.287	8.181	Rise	CLK
11	S[1]	CLK	8.187	8.134	Rise	CLK
12	S[5]	CLK	8.106	7.990	Rise	CLK
13	S[22]	CLK	7.963	7.849	Rise	CLK
14	S[7]	CLK	7.888	7.791	Rise	CLK
15	S[10]	CLK	7.835	7.772	Rise	CLK
16	S[14]	CLK	7.873	7.770	Rise	CLK
17	S[2]	CLK	7.804	7.745	Rise	CLK
18	S[23]	CLK	7.768	7.736	Rise	CLK
19	S[19]	CLK	7.779	7.699	Rise	CLK
20	S[24]	CLK	7.705	7.601	Rise	CLK
21	S[0]	CLK	7.625	7.573	Rise	CLK
22	S[3]	CLK	7.614	7.518	Rise	CLK
23	S[13]	CLK	7.563	7.469	Rise	CLK
24	S[21]	CLK	7.534	7.412	Rise	CLK
25	S[25]	CLK	7.446	7.358	Rise	CLK
26	S[11]	CLK	7.386	7.336	Rise	CLK
27	S[18]	CLK	7.383	7.291	Rise	CLK
28	S[16]	CLK	7.369	7.237	Rise	CLK
29	S[8]	CLK	7.343	7.214	Rise	CLK
30	S[6]	CLK	7.320	7.188	Rise	CLK
31	S[9]	CLK	7.276	7.184	Rise	CLK
32	S[15]	CLK	6.971	6.883	Rise	CLK

Figura 11: Relatório de tempos de *clock* a saída gerado pelo Quartus para um circuito de 16 bits

Pior tempo de clock a saída para N = 16:

Clock Port	Data Port (Saída)	Rise
CLK	S[26]	9.136

N = 8:

Clock to Output Times

	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	S[*]	CLK	8.297	8.161	Rise	CLK
1	S[14]	CLK	8.297	8.161	Rise	CLK
2	S[11]	CLK	8.147	8.030	Rise	CLK
3	S[13]	CLK	8.070	7.983	Rise	CLK
4	S[12]	CLK	8.063	7.976	Rise	CLK
5	S[4]	CLK	7.772	7.683	Rise	CLK
6	S[15]	CLK	7.754	7.691	Rise	CLK
7	S[5]	CLK	7.730	7.670	Rise	CLK
8	S[8]	CLK	7.699	7.696	Rise	CLK
9	S[0]	CLK	7.695	7.622	Rise	CLK
10	S[6]	CLK	7.689	7.580	Rise	CLK
11	S[9]	CLK	7.658	7.587	Rise	CLK
12	S[10]	CLK	7.528	7.499	Rise	CLK
13	S[1]	CLK	7.343	7.251	Rise	CLK
14	S[3]	CLK	7.305	7.238	Rise	CLK
15	S[7]	CLK	6.992	6.935	Rise	CLK
16	S[2]	CLK	6.991	6.930	Rise	CLK

Figura 12: Relatório de tempos de *clock* a saída gerado pelo Quartus para um circuito de 8 bits

Pior tempo de clock a saída para N = 8:

Clock Port	Data Port (Saída)	Rise
CLK	S[14]	8.297

N = 4:

Clock to Output Times

	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	S[*]	CLK	7.905	7.885	Rise	CLK
1	S[2]	CLK	7.905	7.885	Rise	CLK
2	S[5]	CLK	7.786	7.689	Rise	CLK
3	S[4]	CLK	7.785	7.676	Rise	CLK
4	S[6]	CLK	7.758	7.661	Rise	CLK
5	S[0]	CLK	7.609	7.581	Rise	CLK
6	S[7]	CLK	7.575	7.485	Rise	CLK
7	S[3]	CLK	7.170	7.088	Rise	CLK
8	S[1]	CLK	6.659	6.562	Rise	CLK

Figura 13: Relatório de tempos de *clock* a saída gerado pelo Quartus para um circuito de 3 bits

Clock Port	Data Port (Saída)	Rise
CLK	S[2]	7.905

Pior tempo de clock a saída para N = 5, Maior atraso é do S[2] = 7,905 ns:

Este multiplicador possui um tempo para a saída constante de 5 clocks para 16 bits, sendo esta a maior força dele.

Imagens da Simulação:

```
restart -force -nowave
delete wave *
add wave CLK
add wave A
add wave B
add wave S
add wave INICIAR

force CLK 0 0, 1 10ns -repeat 20ns
force INICIAR 1 0ns
force A 0000000000000011 0ns, 0000000000001010 65ns, 1111111111111101 120ns, 0111001111001000
180ns
force B 0000000010000011 0ns, 1111101100100111 65ns, 0000000000000010 120ns, 0001011010110110
180ns
run 2000ns
```

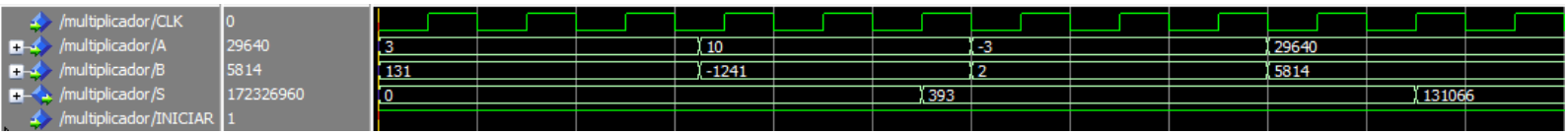


Figura 14: Diagrama de ondas gerado a partir de um testbench para o multiplicador V2 de 16 bits

Comportamento:

Este multiplicador funciona muito bem. Ele é extremamente rápido, dando resultados de forma constante em 5 ciclos de clock. Ele não dá overflow, porém com esta funcionalidade ele perdeu a capacidade de multiplicar número negativos, já que o bit de sinal se perde na conversão para o dobro de bits.

Ele possui uma FSM simples que controla quando ele começa a multiplicação (Sinal INICIAR = 1), e que espera o resultado sair nos 4 ciclos seguintes.

5. Conclusão

Analisando os dois multiplicadores temos a conclusão que o primeiro se torna inviável para operações que envolvam A como número negativo. No segundo multiplicador, ele troca a capacidade de multiplicar dois números negativos para evitar qualquer overflow, dobrando o tamanho de bits da saída.

O primeiro multiplicador, mesmo sendo mais barato, não compensa ser usado devido a seus vários problemas. O segundo, mesmo sendo exponencialmente mais caro em termos de componentes e área, multiplica com uma velocidade extrema, sendo assim, ele “se paga”.

Para 16 bits:	Multiplicador V1	Multiplicador Vedic
Frequência	95.46 MHz	126.50 MHz
Velocidade	9,475 ns	7,905 ns
Nº de ciclos	4 + 3B	5
Pior caso (Em clocks)	65539	5
Total Logic Elements	1023	83
Desvantagem	Overflow, Tempo, Números negativos	Números negativos

6. Referências

- ➔ VHDL - FPGA Tutorial
- ➔ FPGA designs with VHDL — FPGA designs with VHDL documentation
- ➔ https://pt.wikipedia.org/wiki/M%C3%A1quina_de_estados_finita
- ➔ https://www.chegg.com/homework-help/questions-and-answers/vhdl-code-4-bit-16-bit-carry-look-ahead-adder-need-create-32bit-carry-look-ahead-adder--al-q44742685
- ➔ https://gist.github.com/tpmckenzie/6999103
- ➔ https://allaboutfpga.com/carry-look-ahead-adder-vhdl-code/