Multiplicação Através de Somas – Modelo PC-PO

Alunos: Anderson Bottega da Silva, Juliano Felipe Prass da Silva, Maycon de Queiroz Oliveira.

1 INTRODUÇÃO

Este trabalho teve como objetivo desenvolver um multiplicador através de somas com o modelo PC-PO (Parte Controle – Parte Operativa). O multiplicador implementado é de 8 bits utilizando somadores de 16 bits com os números em complemento de dois. Os somadores de 16 bits são CSAs (Carry Select Adders) que utilizam somadores CSAs de 8 bits internamente, que, por sua vez, utilizam somadores CLAs (Carry Look-Ahead Adders) de 4 bits.

1.1 MODELO PC-PO (PARTE CONTROLE – PARTE OPERATIVA)

A divisão de um sistema digital pode ser descrita em duas partes e são a cooperação entre dois blocos:

Parte Operativa: É todo o caminho dos dados que estão na CPU através do barramento, oriundos dos registradores e posteriormente de volta aos próprios registradores.

Parte Controle: Ativa a sequência de operações na Parte Operativa.

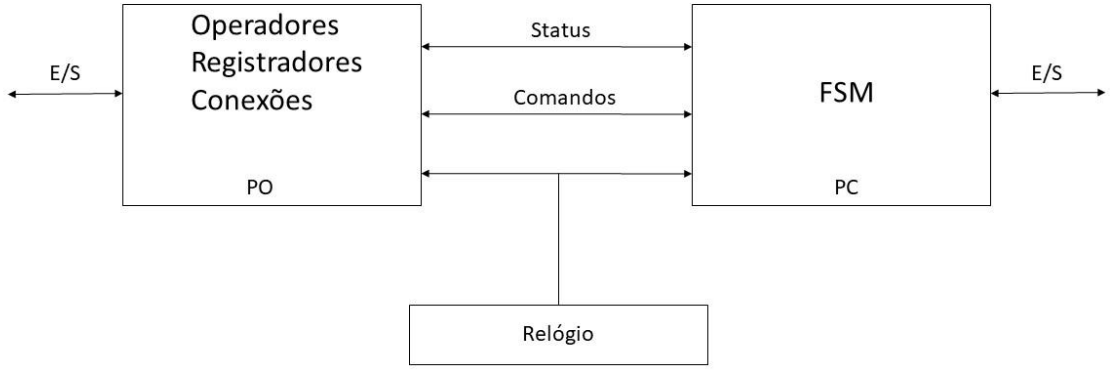

Figura 1.1: Modelo PC-PO

2 DESENVOLVIMENTO

Considerando que os sinais de entrada são 8 bits de tamanho, o resultado será de até 16 bits para multiplicações. Assim, fez-se um “expansor” para que as entradas de 8 bits se tornem de 16 bits. Para o

caso sinalizado, multiplica-se o bit mais significativo (*MSB, most significant bit*), seja este '1' ou '0'. A figura 2.1 mostra o componente para realizar a expansão de entradas em complemento de dois.

```
entity Expand is
   generic(DELAY : time := 4.0 ns);
   port(
      X : in  std_logic_vector(7 downto 0);
      Y : out std_logic_vector(15 downto 0)
   );
end Expand;

architecture Behavioral of Expand is
   signal ZeroFill : std_logic_vector(15 downto 0) := x"0000";
   signal OneFill  : std_logic_vector(15 downto 0) := x"FFFF";
begin
   ZeroFill(7 downto 0) <= X when X(7) = '0';
   OneFill (7 downto 0) <= X when X(7) = '1';

   Y <= ZeroFill when X(7) = '0' else
        OneFill;
end Behavioral;
```

Figure 2.1 Componente para transformar de 8 para 16 bits em complemento de dois.

Depois de expandidos os sinais de entrada, deve-se atribuir os sinais "Multiplicador" e "Multiplicando", como otimização no número de ciclos necessários para computar a multiplicação, escolhe-se o a entrada mínima absoluta como multiplicador (Se entradas são -2 e -3, o componente retorna 2) e a máxima como multiplicando (A máxima absoluta, mas com a saída mantendo o sinal; ex.: entre 2 e -3, componente retorna -3).

Dentro do "MinAbs", faz-se o módulo das duas entradas (Se o *MSB* de alguma entrada for "1", atribui-se o valor negado). Depois, mapeia-se as entradas, em módulo, para o componente mínimo. Este último seleciona o mínimo entre dois valores não sinalizados através da subtração do primeiro pelo segundo, caso o resultado negativo, o primeiro é o mínimo, caso contrário, positivo. Retorna-se o valor em módulo e uma flag que indica se o valor retornado era negativo ou não.

O "Máx" utiliza o componente anterior como auxílio para a escolha da entrada. O máximo faz essencialmente o oposto do "MinAbs", isto é, retorna o máximo em módulo. A diferença é que o primeiro retorna os valores ainda em complemento de dois, não sendo necessário posterior conversão.

O mínimo absoluto é utilizado como multiplicador (Para reduzir a quantia necessária de somas para o resultado final), e o máximo para o multiplicando. As figuras 2.2 e 2.3 mostram os componentes para a seleção do mínimo absoluto e o máximo, respectivamente.

```
entity MinAbs is
   generic(DELAY : time := 4.0 ns);
   port(
      X, Y : in  std_logic_vector(15 downto 0);
      NegF : out std_logic;
      S    : out std_logic_vector(15 downto 0)
   );
end MinAbs;

architecture Behavioral of MinAbs is
   component Min is
      generic(DELAY : time := 4.0 ns);
      port(
         X, Y : in  std_logic_vector(15 downto 0)
         S    : out std_logic_vector(15 downto 0)
      );
   end component;

   component Neg is
      generic(DELAY : time := 4.0 ns);
      port(
         X : in  std_logic_vector(15 downto 0);
         S : out std_logic_vector(15 downto 0)
      );
   end component;

   signal TempS, NX, NY, NegX, NegY :
   std_logic_vector(15 downto 0);
begin
   u_ca_t0 : Neg port map (X, NegX);
   u_ca_t1 : Neg port map (Y, NegY);

   NX <= X when X(15) = '0' else NegX;

   NY <= Y when Y(15) = '0' else NegY;

   u_ca0 : Min port map (NX, NY, TempS);

   S <= TempS;
   NegF <= '1' when ((X(15)='1' and NX=TempS)
   or (Y(15)='1' and NY=TempS)) else
         '0';
end Behavioral;
```

Figure 2.2 Componente mínimo absoluto. As entradas e saídas são iguais ao Max, exceto por uma flag de saída quando o multiplicador é negativo.

```
component Multiplexer16 is
   generic(DELAY : time := 4.0 ns);
   port(
      X, Y : in  std_logic_vector(15 downto 0)
      Sel  : in  std_logic;           -- Selec
      Sout : out std_logic_vector(15 downto 0)
   );
end component;

component MinAbs is
   generic(DELAY : time := 4.0 ns);
   port(
      X, Y : in  std_logic_vector(15 downto 0)
      NegF : out std_logic;
      S    : out std_logic_vector(15 downto 0)
   );
end component;

component Neg is
   generic(DELAY : time := 4.0 ns);
   port(
      X : in  std_logic_vector(15 downto 0);
      S : out std_logic_vector(15 downto 0)
   );
end component;

signal NegF, F1, F2, F3, sselect : std_logic;
signal TempS, NX, NY, NegX, NegY :
std_logic_vector(15 downto 0);
in
u_ca_t0 : Neg port map (X, NegX);
u_ca_t1 : Neg port map (Y, NegY);

NX <= X when X(15) = '0' else NegX;
NY <= Y when Y(15) = '0' else NegX;

u_ca0 : MinAbs port map (X, Y, NegF, TempS);

F1 <= '1' when (NX=TempS) else '0';
F2 <= '1' when (X=NegY)   else '0';
F3 <= '1' when (X=TempS)  else '0';

sselect <= F1 and not(F2 and F3);

u_ca1 : Multiplexer16
   port map (X, Y, sselect, S);
Behavioral;
```

Figure 2.3 Componente máximo.

O componente “Subtrator16” usado em ambos os casos é uma especialização do somador CSA16 (Mostrado mais detalhadamente mais a frente), onde a entrada “Y” é passada pela porta lógica “Not” e o Carry In (Do CSA16) é forçado em “1”, isto é, soma de “X” com o complemento de dois da segunda entrada. O componente “Neg” funciona analogamente, mas força “Y” como “0” na entrada do CSA16, ou seja, faz a negação do número (e.g. 2 para -2). O “Multiplexer16” é um Multiplexador de 16 bits, onde o seletor em zero seleciona “X” e em um, “Y”.

Depois de selecionados os fatores da multiplicação, utilizam-se dois “Multiplexer16” para fazer a inicialização dos dados, um para iniciar o acumulador do resultado em zero e o outro para carregar o valor do multiplicador no acumulador do multiplicador. A inicialização é feita com o sinal “init” ativado, caso este esteja desativado (Depois do segundo pulso de clock), o multiplexador carrega o resultado da operação anterior no acumulador.

Com o auxílio de dois registradores, os dados dos sinais acumuladores são carregados em variáveis de entrada dos somadores. O primeiro somador soma o que foi acumulado até então com o multiplicando. O segundo, decrementa o multiplicador em um (Soma com uma constante igual á -1). Os resultados podem então ser mapeados para os sinais acumuladores (nos multiplexadores) e carregados nos registradores. Por fim, um sinal extra é mantido como a negação do multiplicando acumulado (Será repassado para a saída caso o multiplicador seja negativo).

O Registrador (de 16 bits) apresenta uma porta para carregar algum dado e entrada de clock. Após 8 ns da descida do Clock, a entrada é mapeada para a saída. A figura 2.4 mostra o componente para 1 bit (São acoplados 16 para 16 bits).

```
entity Reg is
   generic(DELAY : time := 8.0 ns);
   port(
      Y    : in  std_logic;
      clk  : in  std_logic;
      load : in  std_logic;
      S    : out std_logic
   );
end Reg;

architecture Behavioral of Reg is
signal p : std_logic := '0';

begin
   S <= p;

   process(clk) is
   begin
      if (load = '1' and
      falling_edge(clk)) then
         p <= Y after DELAY;
      end if;
   end process;

end Behavioral;
```

Figure 2.4 Componente do registrador de 1 bit.

As figuras 2.5 e 2.6 mostram os sinais internos e o mapeamento dos componentes internos da arquitetura, respectivamente.

```
signal ExtX         : std_logic_vector(15 downto 0);
signal ExtY         : std_logic_vector(15 downto 0);

signal Multiplier   : std_logic_vector(15 downto 0);
signal Multiplicand : std_logic_vector(15 downto 0);

signal NotAntCand  : std_logic_vector(15 downto 0);
signal Acc_MultCand, AntCand : std_logic_vector(15 downto 0);
signal Acc_MultPler, AntPler : std_logic_vector(15 downto 0);
signal Temp_MulCand : std_logic_vector(15 downto 0);
signal Temp_MulPler : std_logic_vector(15 downto 0);

signal NegativeMultiplierFlag : std_logic;

constant zeros      : std_logic_vector(Multiplier'range) := (others => '0');
constant MinusOne   : std_logic_vector(15 downto 0) := x"FFFF";

signal Cin, Cout    : std_logic_vector(1 downto 0);
```

Figure 2.5 Sinais internos do Multiplicador.

```
Cin <= "00";
u_ex0 : Expand
   port map (Y, ExtY);
u_ex1 : Expand
   port map (X, ExtX);
u_const1 : MinAbs
   port map (ExtX, ExtY, NegativeMultiplierFlag, Multiplier);
u_const2 : Max
   port map (ExtX, ExtY, Multiplicand);

-- Load
u_mux0 : Multiplexer16
   port map(AntCand, x"0000", init, Acc_MultCand);
u_mux1 : Multiplexer16
   port map(AntPler, Multiplier, init, Acc_MultPler);
u_reg0 : Reg16
   port map(Acc_MultCand, Clock, '1', Temp_MulCand);
u_reg1 : Reg16
   port map(Acc_MultPler, Clock, '1', Temp_MulPler);

-- Sum and Decrement
u_ca1 : CSA16
   port map (Temp_MulCand, Multiplicand, Cin(0), Cout(0), AntCand);
u_dec : CSA16
   port map (Temp_MulPler, MinusOne,     Cin(1), Cout(1), AntPler);

-- Negate
u_negTem : Neg
   port map(AntCand, NotAntCand);
```

Figure 2.6 Mapeamento dos componentes internos do Multiplicador.

2.1 ESPECIFICAÇÃO DO SOMADOR CSA16

Na parte mais alta, o CSA16 utiliza três CSAs de 8 bits. Os 8 bits menos significativos são passados para um CSA8 com o Carry In do primeiro componente. Os bits restantes são duplicados para os outros dois CSAs, um com Carry In forçado em zero, outro em 1. O Carry out do primeiro somador de 8 bits é usado como seletor em um Multiplexador de 8 bits. A soma, consequentemente, converge para a porta de saída depois do tempo de um somador CSA8 (As partes são calculadas em paralelo) mais o tempo de seleção do Multiplexador (4 ns). A figura 2.7 mostra esta parte do somador.

```
architecture Behavioral of CSA16 is
   component CSA8 is
      generic(DELAY : time := 4.0 ns);
      port(
         X, Y : in  std_logic_vector(7 downto 0); -- X + Y
         T    : in  std_logic;           -- Transporte de entrada
         C    : out std_logic;           -- Transporte de saída
         S    : out std_logic_vector(7 downto 0) -- S = X + Y
      );
   end component;

   component Multiplexer8 is
      generic(DELAY : time := 4.0 ns);
      port(
         X, Y : in  std_logic_vector(7 downto 0); -- X + Y
         Sel  : in  std_logic;           -- Selector
         Sout : out std_logic_vector(7 downto 0) -- Saida
      );
   end component;

   component Multiplexer is
      generic(DELAY : time := 4.0 ns);
      port(
         X, Y : in  std_logic;       -- X + Y
         Sel  : in  std_logic;       -- Selector
         Sout : out std_logic        -- Saida
      );
   end component;

   -- vetor de carry
   signal cv  : std_logic_vector(1  downto 0); -- Carry temp
   signal sv  : std_logic_vector(15 downto 0); -- Signal temp
   signal sel : std_logic;             -- Sel temp

begin
   u_ca0 : CSA8
      port map(X(7 downto 0), Y(7 downto 0), T, sel, S(7 downto 0));
   u_ca1 : CSA8
      port map(X(15 downto 8), Y(15 downto 8), '0', cv(0), sv(7 downto 0));
   u_ca2 : CSA8
      port map(X(15 downto 8), Y(15 downto 8), '1', cv(1), sv(15 downto 8));
   u_ca3 : Multiplexer8              --Multiplex Result
      port map(sv(7 downto 0), sv(15 downto 8), sel, S(15 downto 8));
   u_ca4 : Multiplexer --Multiplex Carry
      port map(cv(0), cv(1), sel, C);
end Behavioral;
```

Figure 2.7 Arquitetura do componente CSA16.

O somador CSA8 funciona de forma análoga, mas ao invés de utilizar “CSA4” bits internamente, utiliza CLAs de 4 bits. A figura 2.8 ilustra a arquitetura do primeiro.

```
architecture Behavioral of CSA8 is
   component CLA is
      generic(DELAY : time := 4.0 ns);
      port(
         X, Y : in  std_logic_vector(3 downto 0); -- X + Y
         T    : in  std_logic;       -- Transporte de entrada
         C    : out std_logic;       -- Transporte de saÃda
         S    : out std_logic_vector(3 downto 0) -- S = X + Y
      );
   end component;

   component Multiplexer4 is
      generic(DELAY : time := 4.0 ns);
      port(
         X, Y : in  std_logic_vector(3 downto 0); -- X + Y
         Sel  : in  std_logic;       -- Selector
         Sout : out std_logic_vector(3 downto 0) -- Saida
      );
   end component;

   component Multiplexer is
      generic(DELAY : time := 4.0 ns);
      port(
         X, Y : in  std_logic;       -- X + Y
         Sel  : in  std_logic;       -- Selector
         Sout : out std_logic        -- Saida
      );
   end component;

   -- vetor de carry
   signal cv  : std_logic_vector(1 downto 0); -- Carry temp
   signal sv  : std_logic_vector(7 downto 0); -- Signal temp
   signal sel : std_logic;             -- Sel temp

begin
   u_ca0 : CLA
      port map(X(3 downto 0), Y(3 downto 0), T, sel, S(3 downto 0));
   u_ca1 : CLA
      port map(X(7 downto 4), Y(7 downto 4), '0', cv(0), sv(3 downto 0));
   u_ca2 : CLA
      port map(X(7 downto 4), Y(7 downto 4), '1', cv(1), sv(7 downto 4));
   u_ca3 : Multiplexer4                --Multiplex Result
      port map(sv(3 downto 0), sv(7 downto 4), sel, S(7 downto 4));
   u_ca4 : Multiplexer --Multiplex Carry
      port map(cv(0), cv(1), sel, C);
end Behavioral;
```

Figure 2.8 Arquitetura do componente CSA8.

O somador de 4 bits usa somadores completos simples (Que utiliza um meio somador internamente) e fórmulas para determinar antecipadamente o Carry In dos somadores completos intermediários em tempo de estabilização O($\log_2 n$). As fórmulas booleanas utilizadas são conseguidas a partir da expansão das expressões booleanas de um somador completo. São elas:

$$S_i = X_i \oplus Y_i \oplus C_i$$ -> i-ésima soma, onde C_i é o i-ésimo CarryIn

$$Cout_i = (X_i \wedge Y_i) \vee (C_i \wedge (X_i \vee Y_i))$$ -> i-ésimo CarryOut ou C_{i+1}

Para efeitos de simplificação, considera-se:

$$G_i = (X_i \wedge Y_i)$$ -> Gerador de Carry

$$P_i = (X_i \vee Y_i)$$ -> Propagador de Carry

Assim, pode-se reescrever a expressão para o Carry In $_{i+1}$ como:

$$C_{i+1} = G_i \vee (P_i \wedge C_i)$$

Dessa maneira, podemos explicitar as fórmulas para os "Carry In"s dos bits de 1 à 3 (Carry In $_0$ é dado como entrada do componente):

$$C_1 = G_0 \vee (P_0 \wedge C_0)$$

$$C_2 = G_1 \vee (P_1 \wedge C_1)$$

$$C_2 = G_1 \vee (P_1 \wedge (G_0 \vee P_0 \wedge C_0))$$

$$C_2 = G_1 \vee (P_1 \wedge G_0) \vee (P_1 \wedge P_0 \wedge C_0)$$

$$C_3 = G_2 \vee (P_2 \wedge G_1) \vee (P_2 \wedge P_1 \wedge G_0) \vee (P_2 \wedge P_1 \wedge P_0 \wedge C_0)$$

Nota-se que expandir a fórmula para mais bits (e.g. 8), torna a estratégia menos atrativa, devido ao número de portas lógicas necessárias a partir da expansão de C_3. Por esse motivo, utilizou-se a estratégia do Carry Select Adder para as outras etapas. A figura 2.9 mostra a arquitetura em VHDL do CLA.

```
   signal carry   : std_logic_vector(3 downto 0); -- Carry temp
   signal gener8  : std_logic_vector(3 downto 0); -- Generate temp
   signal propag8 : std_logic_vector(3 downto 0); -- Propagate temp

   signal trash   : std_logic_vector(2 downto 0);

begin
   carry(0)   <= T;
   gener8(0)  <= X(0) AND Y(0) after DELAY;
   propag8(0) <= X(0) OR Y(0)  after DELAY;

   u_ca0 : Adder
      port map(X(0), Y(0), carry(0), trash(0), S(0));

   carry(1)   <= gener8(0) OR (propag8(0) AND carry(0)) after 2*DELAY;
   gener8(1)  <= X(1) AND Y(1) after DELAY;
   propag8(1) <= X(1) OR Y(1) after DELAY;

   u_ca1 : Adder
      port map(X(1), Y(1), carry(1), trash(1), S(1));

   carry(2)   <= gener8(1) OR
                 (propag8(1) AND gener8(0)) OR
                 (propag8(1) AND propag8(0) AND carry(0))
            after 2*DELAY;
   gener8(2)  <= X(2) AND Y(2) after DELAY;
   propag8(2) <= X(2) OR Y(2) after DELAY;

   u_ca2 : Adder
      port map(X(2), Y(2), carry(2), trash(2), S(2));

   carry(3)   <= gener8(2) OR
                  (propag8(2) AND gener8(1)) OR
                  (propag8(2) AND propag8(1) AND gener8(0)) OR
                  (propag8(2) AND propag8(1) AND propag8(0) AND carry(0))
        AFTER 2*DELAY;

   u_ca3 : Adder
      port map(X(3), Y(3), carry(3), C, S(3));


end Behavioral;
```

Figure 2.9 CLA de 4 bits.

2.2 CONTROLE DO SINAL DE SAÍDA E TESTBENCH UTILIZADO

Após 1 ciclo de inicialização (Entrada “init” no componente “MultiplicadorSomador8”), são necessários “m” ciclos para o resultado convergir, onde “m” é o multiplicador, ou seja, ao multiplicar os números “5” e “-2”, levam-se 1+2 (Multiplicador é 2, já que o menor número absoluto é “|-2|“) ciclos para ter-se o resultado na saída. Caso multiplique-se por 0, o tempo é de apenas um ciclo. Devido ao sinal auxiliar “NotAntCand” (Figura 2.5), o tempo não muda para multiplicadores negativos.

O controle da saída é feito com um processo sensível ao sinal de clock. A figura 2.10 mostra esse trecho do código VHDL. A figura 2.11 mostra um exemplo de testbench para testar o Multiplicador.

```
process (Clock) is
begin
   if falling_edge(Clock) then
      if Multiplier = zeros then
         Sout <= zeros; --Mult por zero
      elsif AntPler = zeros then
         if NegativeMultiplierFlag = '1' then
            Sout <= NotAntCand; --Multiplier Negativo
         else
            Sout <= AntCand; -- Multiplier Positivo
         end if;
      end if;
   end if;
end process;
```

Figure 2.10 Controle de saída de resultados

```
architecture Behavioral of tb_mult is
   component MultiplicadorSomador8 is
      generic(DELAY : time := 4.0 ns);
      port(
         X, Y : in  std_logic_vector(7 downto 0);
         Clock: in  std_logic;
         init : in  std_logic;
         Sout : out std_logic_vector(15 downto 0)
      );
   end component;

   signal clk, init : std_logic := '0';

   signal X, Y : std_logic_vector(7 downto 0);
   signal Res  : std_logic_vector(15 downto 0);
begin
   clk <= not clk after 20 ns; --Clk de 40 ns

   u_top : MultiplicadorSomador8
      port map (X, Y, clk, init, Res);

   process
   begin
      init <= '1';
      X <= "11111101";
      Y <= "00000010";
      wait until falling_edge(clk);

      init <= '0';
      wait on Res;
      --Só para que a onda fique mais visível
      wait for 49 ns;

      assert false
            report "Stuff's done."
            severity failure;
   end process;

end Behavioral;
```

Figure 2.11 Exemplo de TestBench para o MultiplicadorSomador8.

2.3 DIFERENCIAÇÃO NA MULTIPLICAÇÃO DE NÃO SINALIZADOS

Para realizar multiplicações não-sinalizadas, tem-se que modificar levemente o funcionamento do MultiplicadorSomador8 e de seus componentes internos “Expand” e “Max”. Os outros, apesar de considerarem valores negativos, não precisam ser substituídos devido à expansão de sinal. Por exemplo, ao utilizar o componente “MinAbs”, não haverá diferença, pois, o bit mais significativo nunca será um (e.g. o componente não o considerará negativo), uma vez que a expansão de números não sinalizados sempre os preenche com zeros.

O componente nomeado “U_Max” faz a seleção do valor máximo entre as duas entradas de forma muito semelhante ao “Min”, descrito anteriormente: Realiza-se a subtração “X-Y”, se o *MSB* do resultado for um, seleciona-se Y, se não, X. A figura 2.12 mostra tal componente.

```
architecture Behavioral of U_Max is
   component Subtrator16 is
      generic(DELAY : time := 4.0 ns);
      port(
         X, Y : in  std_logic_vector(15 downto 0);
         C    : out std_logic;
         S    : out std_logic_vector(15 downto 0)
      );
   end component;

   component Multiplexer16 is
      generic(DELAY : time := 4.0 ns);
      port(
         X, Y : in  std_logic_vector(15 downto 0);
         Sel  : in  std_logic;
         Sout : out std_logic_vector(15 downto 0)
      );
   end component;

   signal X_Minus_Y : std_logic_vector(15 downto 0);
   signal C_Out : std_logic;
begin
   u_ca0 : Subtrator16
      port map (X, Y, C_Out, X_Minus_Y);

   u_ca1 : Multiplexer16
      port map (X, Y, X_Minus_Y(15), S);

end Behavioral;
```

Figure 2.12 U_Max.

A expansão do sinal pode ser simplificada para uma simples inclusão de zeros ao lado do *MSB*. A figura 2.13 mostra tal inclusão.

```
entity U_Expand is
   generic(DELAY : time := 4.0 ns);
   port(
      X : in  std_logic_vector(7 downto 0);
      Y : out std_logic_vector(15 downto 0)
   );
end U_Expand;

architecture Behavioral of U_Expand is
   signal ZeroFill :
   std_logic_vector(15 downto 0) := x"0000";
begin
   ZeroFill(7 downto 0) <= X;

   Y <= ZeroFill;
end Behavioral;
```

Figure 2.13 U_Expand

O Multiplicador não sinalizado (alcunhado de “U_MultiplicadorSomador8”) tem o mapeamento de portas e os sinais de controle levemente alterados. Ao invés de utilizar “Expand” e “Max”, substitui por “U_Expand” e “U_Max”, respectivamente e exclui a necessidade de manter um sinal acumulado negado adicional. A estrutura de controle de saída também pode ser simplificada. O testbench só precisa ser alterado no componente que utiliza para passar as entradas, de “MultiplicadorSomador8” para “U_MultiplicadorSomador8”. As figuras 2.14 e 2.15 mostram o mapeamento e o controle, respectivamente.

```
begin
   Cin <= "00";
   u_ex0 : U_Expand port map (Y, ExtY);
   u_ex1 : U_Expand port map (X, ExtX);
   u_const1 : MinAbs port map (ExtX, ExtY, Flag, Multiplier);
   u_const2 : U_Max  port map (ExtX, ExtY, Multiplicand);
   -- Load
   u_mux0 : Multiplexer16
      port map(AntCand, x"0000", init, Acc_MultCand);
   u_mux1 : Multiplexer16
      port map(AntPler, Multiplier, init, Acc_MultPler);
   u_reg0 : Reg16
      port map(Acc_MultCand, Clock, '1', Temp_MulCand);
   u_reg1 : Reg16
      port map(Acc_MultPler, Clock, '1', Temp_MulPler);
   -- Sum and Decrement
   u_ca1 : CSA16
      port map (Temp_MulCand, Multiplicand, Cin(0), Cout(0), AntCand);
   u_dec : CSA16
      port map (Temp_MulPler, MinusOne,     Cin(1), Cout(1), AntPler);
```

Figure 2.14 Mapeamento do "U_MultiplicadorSomador8".

```
process (Clock) is
begin
   if falling_edge(Clock) then
      if Multiplier = zeros then
         Sout <= zeros;
      elsif AntPler = zeros then
         Sout <= AntCand;
      end if;
   end if;
end process;
```

Figure 2.15 Controle de saída do "U_MultiplicadorSomador8".

3 FUNCIONAMENTO

O multiplicador de 8 bits recebe duas entradas e escolhe o menor valor absoluto entre elas, então considera este valor como n, o outro valor é somado n vezes. Se o valor de n for negativo, o resultado é invertido. O resultado é armazenado em 16 bits.

O somador CSA16 leva 20ns para convergir à um resultado, uma vez que o CSA8 leva 16 ns e o Multiplexador interno leva 4 ns para realizar a seleção. O CSA8, por sua vez, leva tal tempo pois o CLA leva 12ns e o multiplexador leva mais 4 ns. Os registradores levam 8 ns após a descida do Clock para carregar o resultado na saída.

Considerando os tempos, o Clock utilizado foi com um período de 40 ns, iniciando em zero (20 ns em zero, depois 20 ns em um, e assim sucessivamente). O multiplicador não sinalizado pode ser operado com um clock de 32 ns de período, devido à ausência de um negador de acumulador (Necessário no sinalizado para realizar multiplicação com multiplicadores negativos). As figuras 3.1 a 3.3 mostram testes de tempo com os somadores de 4 a 16 bits, em ordem.

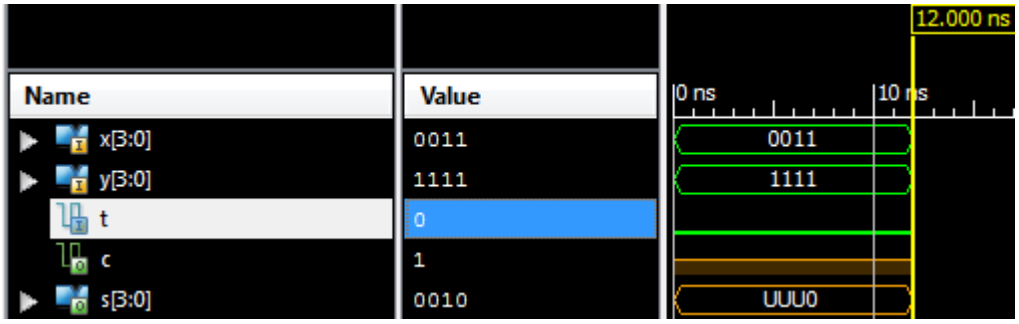

Figure 3.1 Teste com CLA 4 bits.

Figure 3.2 Teste com CSA8 bits.

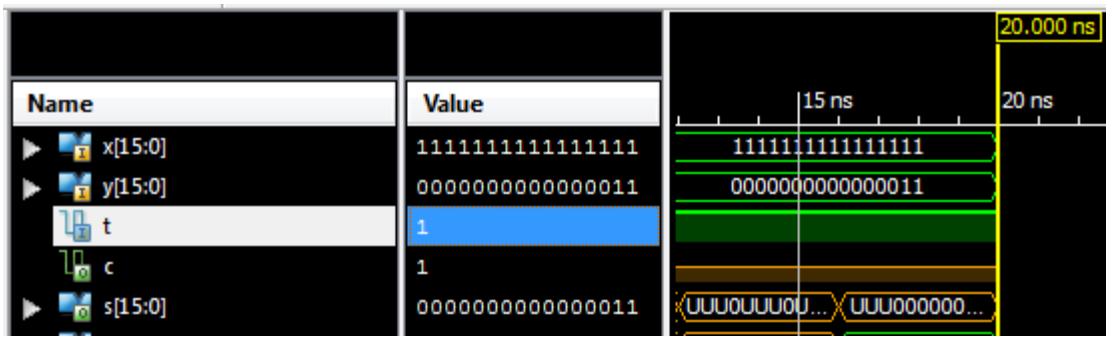

Figure 3.3 Teste com CSA 16 bits.

4 CONCLUSÃO

Há casos onde a estratégia implementada pode ser melhor que o Algoritmo de Multiplicação de Booth, por exemplo. Enquanto o alg. De Booth leva y (Sendo este o número de bits do multiplicador) ciclos de clock + 1 de inicialização, o multiplicador somador leva z (Sendo este o módulo do multiplicador) ciclos + 1 de inicialização. Quando se multiplica -3 * 2 (8 bits) usando Booth, por exemplo, ter-se-ia o resultado em 8 + 1 ciclos de clock; enquanto no implementado ter-se-ia o mesmo resultado em 2 + 1 ciclos. A figura 4.1 mostra tal exemplo.

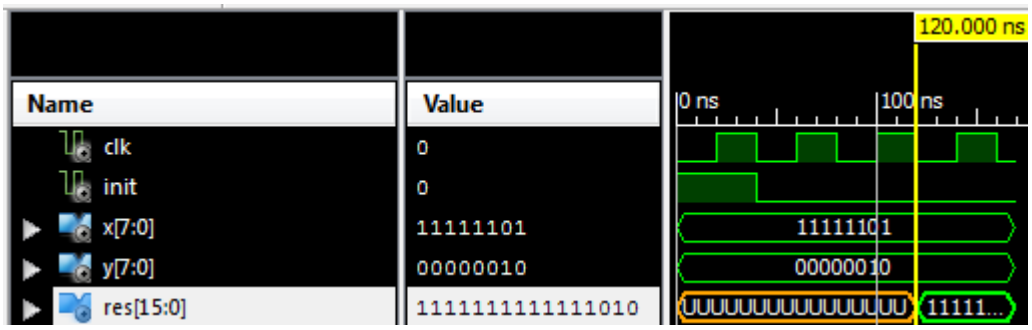

Figure 4.1 Multiplicação entre -3 e utilizando o "MultiplicadorSomador8".

Consequentemente, quando se tem um multiplicador cujo módulo é maior que o número de bits utilizado para representa-lo, o Multiplicador de Booth supera o somador. Como exemplo, considere a multiplicação sinalizada -126*-122 com 8 bits, enquanto Booth calcularia em 9 ciclos, o somador calcularia em 122 ciclos (|-122| + 1). A figura 4.2 mostra este exemplo.

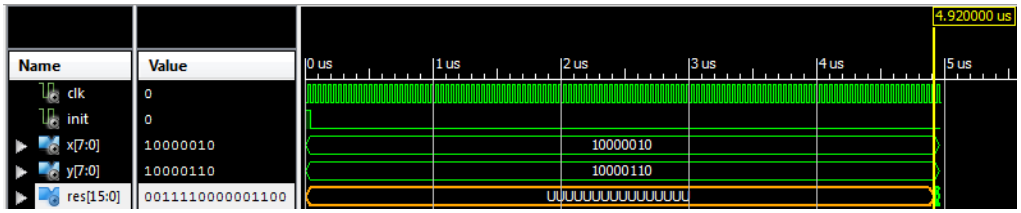

Figure 4.2 Multiplicação entre -126 e -122 utilizando o "MultiplicadorSomador8"

Outra nuance é que Booth apenas considera inteiros sinalizados, então caso fosse multiplicar 128*2 com 8 bits, necessitaria de 9 bits para que a multiplicação não fosse interpretada como -128*2, o que ocuparia mais recursos, um ciclo a mais de clock quando comparado com o mesmo algoritmo sinalizado de mesmo tamanho e um somador de tamanho equivalente (O de Booth utiliza somadores do mesmo número de bits que os operandos).

Obviamente, ainda seria melhor que o não sinalizado “Multiplicador Somador”, pois este precisaria alterar os componentes “Max” e “Expand”, como discutido na seção “Diferenciação na multiplicação de não sinalizados”, e ainda possui a característica de precisar de somadores maiores que os operandos da multiplicação e a grande quantia de ciclos com multiplicadores grandes.

5 ANEXOS

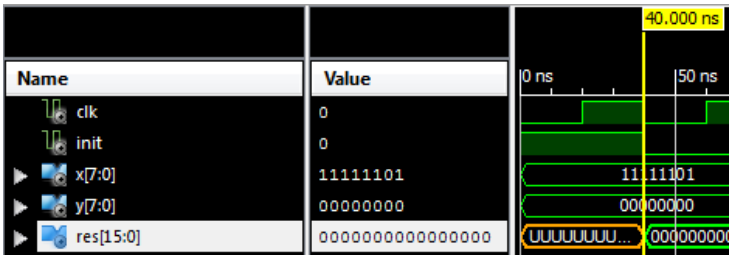

Figure 5.1 Multiplicação sinalizada entre -3 e 0. Resultado em 1 ciclo.

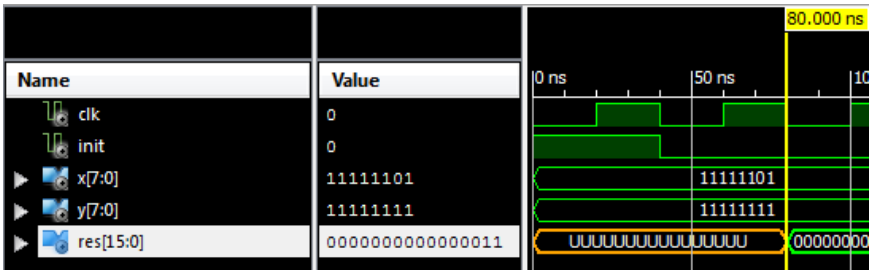

Figure 5.2 Multiplicação sinalizada entre -3 e -1. Resultado em 2 ciclos (Init + |-1|).

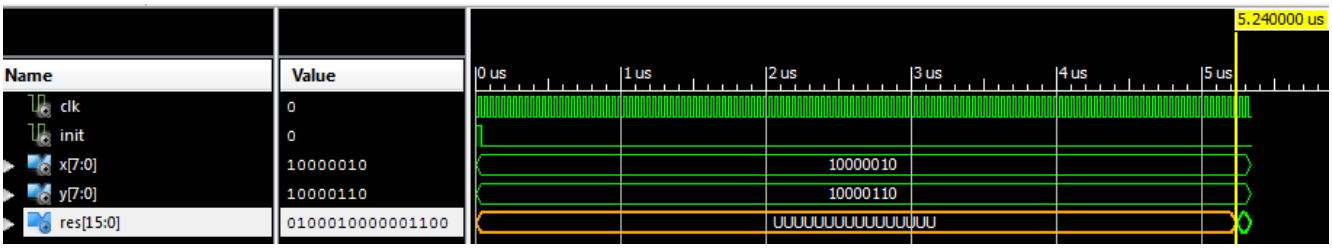

Figure 5.3 Multiplicação não sinalizada entre 130 e 134. Resultado em 131 ciclos (Init + |130|).

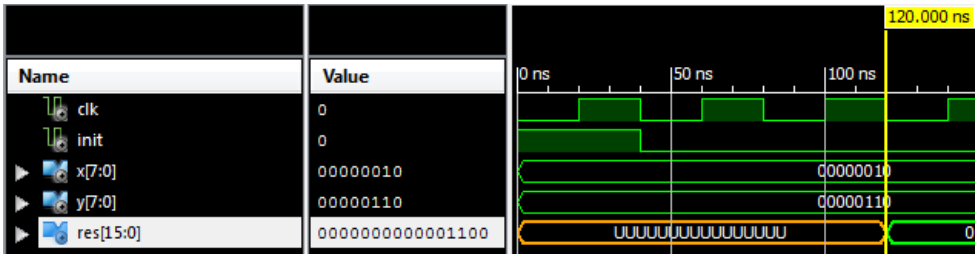

Figure 5.4 Multiplicação não sinalizada entre 2 e 3. Resultado em 3 ciclos (Init + |2|).