Atividade 2 - Dispositivos Lógicos Programáveis II

Daniel Tatsch, Nelson Alves, Schaiana Sonaglio
Engenharia de Telecomunicações
IFSC - Instituto Federal de Santa Catarina, São José, SC
Abril de 2018

1 Introdução

Este documento apresenta a modificação, os resultados e a análise de três exercícios propostos no capítulo sete do livro *RTL hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability* (CHU, 2006), feitos no *software* Quartus. Nestes exercícios, (*Listing* 7.13), (*Listing* 7.14) e (*Listing* 7.29), são feitas diferentes implementações de um circuito denominado *barrel shifter*, que propõe deslocar os *bits* de uma informação de entrada quantas posições fossem necessárias. Esta operação de *shift* pode ser feita deslocando os próprios *bits* da informação para direita (modo rotação - *rot_result*), adicionando *bits* '0' no começo da informação e deslocando os *bits* restantes para a direita (modo *modo lógico* - *logic_result*) ou adicionando o bit mais significativo da informação à sua frente e também deslocando os *bits* resultantes para direita (modo aritmético - *arith_result*). No último exercício, (*Listing* 7.29), a implementado foi feita em diferentes níveis, onde em cada nível é feita uma operação de *shift* de um determinado número de *bits*, anexando ao final do nível mais alto o total de *bits* da informação já deslocados nos níveis anteriores.

Em todos os exercícios foram analisados os tempos de propagação externo e interno (desconsiderando os tempos dos pinos de I/O) e a área ocupada por cada circuito gerado, utilizando como entrada um vetor '*a*' = "10100110101001100001111000011110"(tamanho: 32 *bits*). Além disso, para as questões um e três, foram inseridas restrições temporais para obter um circuito com menor atraso no caminho crítico, para isso, reduziu-se gradualmente o tempo máximo de propagação "tp"até atingir o menor valor de *delay* possível.

2 Implementação

Utilizando o software Quartus, foram implementados códigos VHDL que viabilizassem o proposto em cada exercício. O dispositivo utilizado foi o EP4CE115F29C7 e em todas as execuções foram verificados os tempos de propagação com e sem a utilização do logic lock.

2.1 Exercício 1

Neste exercício, foram modificadas as três operações do exercício 7.13: a rotacional, a lógica e a aritmética. Antes da modificação, as entradas *a* e *amt* eram de oito e cinco *bits*, respectivamente, e a saída *y* era de oito *bits*; após a modificação, a entrada *a* e a entrada *amt* passaram a ser de trinta e dois e cinco *bits*, respectivamente, e a saída *y* passou a ter também trinta e dois *bits*. Além da alteração do número de *bits*, houve um aprimoramento da operação aritmética, para que o bit mais significativo *a(31)* não precisasse ser repetido muitas vezes no código.

Na operação rotacional, alguns dos *bits* menos significativos, dependendo do valor de *amt*, foram colocados no lugar dos *bits* mais significativos, concatenando com o restante do número, que foi deslocado para a direita; por exemplo, na operação do *amt* assumindo o valor "00010", os *bits a(1)* e *a(0)* foram concatenados com o *bit a(31)* até o *bit a(2)*.

Na operação lógica, dependendo do valor de *amt*, zeros foram colocados no lugar dos *bits* mais significativos, concatenando com o restante do número, que foi deslocado para a direita; por exemplo, na operação do *amt* assumindo o valor "00010", os *bits a(1)* e *a(0)* assumiram o valor zero e foram concatenados com o *bit a(31)* até o *bit a(2)*, pertencentes ao número original.

Na operação aritmética, considerando *a* um número com sinal, o bit de sinal *a(31)* foi repetido no início do número uma quantidade variada de vezes; por exemplo, na operação do *amt* assumindo o valor "00010", o bit de sinal *a(31)* foi repetido duas vezes, nas posições *a(31)* e *a(30)*, e foi concatenado com o restante do número deslocado para a direita, ou seja, os *bits (a(31)* até *(a(2)*, suprimindo os bits *a(1)* e *a(0)*, até que o *bit* de sinal assumisse o número todo, como na operação "11111"ou *when others* do *amt*.

Abaixo, o código modificado e aprimorado:

```
library ieee;
use ieee.std_logic_1164.all;
entity shift3mode is
   port(
      a: in std_logic_vector(31 downto 0);
      lar: in std_logic_vector(1 downto 0);
      amt: in std_logic_vector(4 downto 0);
      y: out std_logic_vector(31 downto 0)
   );
end shift3mode ;

architecture direct_arch of shift3mode is
   signal logic_result, arith_result, rot_result:
      std_logic_vector(31 downto 0);
begin
   with amt select
   rot_result<=
    a                                          when "00000",
    a(0) & a(31 downto 1)                      when "00001",
    a(1  downto 0) & a(31 downto  2)           when "00010",
    a(2  downto 0) & a(31 downto  3)           when "00011",
    a(3  downto 0) & a(31 downto  4)           when "00100",
    a(4  downto 0) & a(31 downto  5)           when "00101",
    a(5  downto 0) & a(31 downto  6)           when "00110",
    a(6  downto 0) & a(31 downto  7)           when "00111",
    a(7  downto 0) & a(31 downto  8)           when "01000",
```

```
    a(8  downto 0) & a(31 downto  9)          when "01001",
    a(9  downto 0) & a(31 downto 10)          when "01010",
    a(10 downto 0) & a(31 downto 11)          when "01011",
    a(11 downto 0) & a(31 downto 12)          when "01100",
    a(12 downto 0) & a(31 downto 13)          when "01101",
    a(13 downto 0) & a(31 downto 14)          when "01110",
    a(14 downto 0) & a(31 downto 15)          when "01111",
    a(15 downto 0) & a(31 downto 16)          when "10000",
    a(16 downto 0) & a(31 downto 17)          when "10001",
    a(17 downto 0) & a(31 downto 18)          when "10010",
    a(18 downto 0) & a(31 downto 19)          when "10011",
    a(19 downto 0) & a(31 downto 20)          when "10100",
    a(20 downto 0) & a(31 downto 21)          when "10101",
    a(21 downto 0) & a(31 downto 22)          when "10110",
    a(22 downto 0) & a(31 downto 23)          when "10111",
    a(23 downto 0) & a(31 downto 24)          when "11000",
    a(24 downto 0) & a(31 downto 25)          when "11001",
    a(25 downto 0) & a(31 downto 26)          when "11010",
    a(26 downto 0) & a(31 downto 27)          when "11011",
    a(27 downto 0) & a(31 downto 28)          when "11100",
    a(28 downto 0) & a(31 downto 29)          when "11101",
    a(29 downto 0) & a(31 downto 30)          when "11110",
    a(30 downto 0) & a(31)                    when others;


with amt select
    logic_result<=
     a  when "00000",
     "0" & a(31 downto  1) when "00001",
     "00" & a(31 downto  2) when "00010",
     "000" & a(31 downto  3) when "00011",
     "0000" & a(31 downto  4) when "00100",
     "00000" & a(31 downto  5) when "00101",
     "000000" & a(31 downto  6) when "00110",
     "0000000" & a(31 downto  7) when "00111",
     "00000000" & a(31 downto  8) when "01000",
     "000000000" & a(31 downto  9) when "01001",
     "0000000000" & a(31 downto 10) when "01010",
     "00000000000" & a(31 downto 11) when "01011",
     "000000000000" & a(31 downto 12) when "01100",
     "0000000000000" & a(31 downto 13) when "01101",
     "00000000000000" & a(31 downto 14) when "01110",
     "000000000000000" & a(31 downto 15) when "01111",
     "0000000000000000" & a(31 downto 16) when "10000",
     "00000000000000000" & a(31 downto 17) when "10001",
     "000000000000000000" & a(31 downto 18) when "10010",
     "0000000000000000000" & a(31 downto 19) when "10011",
     "00000000000000000000" & a(31 downto 20) when "10100",
     "000000000000000000000" & a(31 downto 21) when "10101",
     "0000000000000000000000" & a(31 downto 22) when "10110",
     "00000000000000000000000" & a(31 downto 23) when "10111",
     "000000000000000000000000" & a(31 downto 24) when "11000",
     "0000000000000000000000000" & a(31 downto 25) when "11001",
```

```
      "00000000000000000000000000" & a(31 downto 26) when "11010",
      "00000000000000000000000000" & a(31 downto 27) when "11011",
      "00000000000000000000000000" & a(31 downto 28) when "11100",
      "00000000000000000000000000" & a(31 downto 29) when "11101",
      "00000000000000000000000000" & a(31)           when others;


with amt select
    arith_result<=
    a                                                 when "00000",
    a(31)                    & a(31 downto  1)        when "00001",
    (31 downto 30 => a(31)) & a(31 downto  2)         when "00010",
    (31 downto 29 => a(31)) & a(31 downto  3)         when "00011",
    (31 downto 28 => a(31)) & a(31 downto  4)         when "00100",
    (31 downto 27 => a(31)) & a(31 downto  5)         when "00101",
    (31 downto 26 => a(31)) & a(31 downto  6)         when "00110",
    (31 downto 25 => a(31)) & a(31 downto  7)         when "00111",
    (31 downto 24 => a(31)) & a(31 downto  8)         when "01000",
    (31 downto 23 => a(31)) & a(31 downto  9)         when "01001",
    (31 downto 22 => a(31)) & a(31 downto 10)         when "01010",
    (31 downto 21 => a(31)) & a(31 downto 11)         when "01011",
    (31 downto 20 => a(31)) & a(31 downto 12)         when "01100",
    (31 downto 19 => a(31)) & a(31 downto 13)         when "01101",
    (31 downto 18 => a(31)) & a(31 downto 14)         when "01110",
    (31 downto 17 => a(31)) & a(31 downto 15)         when "01111",
    (31 downto 16 => a(31)) & a(31 downto 16)         when "10000",
    (31 downto 15 => a(31)) & a(31 downto 17)         when "10001",
    (31 downto 14 => a(31)) & a(31 downto 18)         when "10010",
    (31 downto 13 => a(31)) & a(31 downto 19)         when "10011",
    (31 downto 12 => a(31)) & a(31 downto 20)         when "10100",
    (31 downto 11 => a(31)) & a(31 downto 21)         when "10101",
    (31 downto 10 => a(31)) & a(31 downto 22)         when "10110",
    (31 downto  9 => a(31)) & a(31 downto 23)         when "10111",
    (31 downto  8 => a(31)) & a(31 downto 24)         when "11000",
    (31 downto  7 => a(31)) & a(31 downto 25)         when "11001",
    (31 downto  6 => a(31)) & a(31 downto 26)         when "11010",
    (31 downto  5 => a(31)) & a(31 downto 27)         when "11011",
    (31 downto  4 => a(31)) & a(31 downto 28)         when "11100",
    (31 downto  3 => a(31)) & a(31 downto 29)         when "11101",
    (31 downto  2 => a(31)) & a(31 downto 30)         when "11110",
    (31 downto  1 => a(31)) & a(31)                   when others;


      with lar select
    y <= rot_result when "00",
         logic_result when "01",
         arith_result   when others;
end direct_arch;
```

Na Tabela 1, é o mostrado o número de pinos, o número de elementos lógicos, o tempo de propagação com e sem o logic lock e o tempo de propagação desconsiderando o tempo nos pinos de I/O, com e sem o logic lock.

Tabela 1 - Resultados obtidos com a Arquitetura *direct_arch*

Número de pinos	71
Número de elementos lógicos - modo normal	295
Número de elementos lógicos - modo aritmético	0
Tempo de propagação (sem logic lock)	20.748 ns
Tempo de propagação (com logic lock)	17.064 ns
Tempo de propagação interno (sem logic lock)	7.373 ns
Tempo de propagação interno (com logic lock)	5.792 ns

Na Tabela 2, são mostrados os mesmos dados da Tabela 1, após manipulação de tempo proposta. Observou-se uma melhoraria significativa para este circuito, saindo de um *delay* no caminho crítico de 5.792 ns para 3.645 ns, porém, houve aumento do número de elementos lógicos, que saiu de 295 para 303. Esta manipulação foi feita adicionando o comando *set_max_delay -from [get_ports *] -to [get_ports *] tp* no projeto feito no Quartus, que o instrui a fazer melhorias baseadas num tempo de propagação no caminho crítico específico.

Tabela 2 - Resultados obtidos após manipulação de tempo *shared_arch*

Número de pinos	71
Número de elementos lógicos - modo normal	303
Número de elementos lógicos - modo aritmético	0
Tempo de propagação (com logic lock)	9.940 ns
Tempo de propagação interno (com logic lock)	3.645 ns

Para simular o circuito implementado, foi desenvolvido um programa de teste (.vht - Figura 1) onde se variou a entrada '*amt*' a cada 10 us, representando todos os valores assumidos por esta entrada, para cada valor da entrada '*lar*'. Com isso, observou-se na simulação o deslocamento de todos os *bits* do vetor de entrada '*a*', com os três diferentes tipos de deslocamento apresentados. As Figuras **??**, **??** e **??** mostram todas as entradas e a saída '*y*' no *software* ModelSIM, exibindo alguns valores resultantes na saída, deslocando a informação e adicionando os *bits* correspondentes ao valor de '*lar*' de acordo com o valor de '*amt*'.

```
a <= "10100110101001100001111000011110";
amt <= "00000", "00001" AFTER 10us, "00010" AFTER 20us, "00011" AFTER 30us, "00100" AFTER 40us,
       "00101" AFTER 50us, "00110" AFTER 60us, "00111" AFTER 70us, "01000" AFTER 80us, "01001" AFTER 90us,
       "01010" AFTER 100us, "01011" AFTER 110us, "01100" AFTER 120us, "01101" AFTER 130us, "01110" AFTER 140us,
       "01111" AFTER 150us, "10000" AFTER 160us, "10001" AFTER 170us, "10010" AFTER 180us, "10011" AFTER 190us,
       "10100" AFTER 200us, "10101" AFTER 210us, "10110" AFTER 220us, "10111" AFTER 230us, "11000" AFTER 240us,
       "11001" AFTER 250us, "11010" AFTER 260us, "11011" AFTER 270us, "11100" AFTER 280us, "11101" AFTER 290us,
       "11110" AFTER 300us, "11111" AFTER 310us,
       "00000" AFTER 320us, "00001" AFTER 330us, "00010" AFTER 340us, "00011" AFTER 350us, "00100" AFTER 360us,
       "00101" AFTER 370us, "00110" AFTER 380us, "00111" AFTER 390us, "01000" AFTER 400us, "01001" AFTER 410us,
       "01010" AFTER 420us, "01011" AFTER 430us, "01100" AFTER 440us, "01101" AFTER 450us, "01110" AFTER 460us,
       "01111" AFTER 470us, "10000" AFTER 480us, "10001" AFTER 490us, "10010" AFTER 500us, "10011" AFTER 510us,
       "10100" AFTER 520us, "10101" AFTER 530us, "10110" AFTER 540us, "10111" AFTER 550us, "11000" AFTER 560us,
       "11001" AFTER 570us, "11010" AFTER 580us, "11011" AFTER 590us, "11100" AFTER 600us, "11101" AFTER 610us,
       "11110" AFTER 620us, "11111" AFTER 630us,
       "00000" AFTER 640us, "00001" AFTER 650us, "00010" AFTER 660us, "00011" AFTER 670us, "00100" AFTER 680us,
       "00101" AFTER 690us, "00110" AFTER 700us, "00111" AFTER 710us, "01000" AFTER 720us, "01001" AFTER 730us,
       "01010" AFTER 740us, "01011" AFTER 750us, "01100" AFTER 760us, "01101" AFTER 770us, "01110" AFTER 780us,
       "01111" AFTER 790us, "10000" AFTER 800us, "10001" AFTER 810us, "10010" AFTER 820us, "10011" AFTER 830us,
       "10100" AFTER 840us, "10101" AFTER 850us, "10110" AFTER 860us, "10111" AFTER 870us, "11000" AFTER 880us,
       "11001" AFTER 890us, "11010" AFTER 900us, "11011" AFTER 910us, "11100" AFTER 920us, "11101" AFTER 930us,
       "11110" AFTER 940us, "11111" AFTER 950us;
lar <= "00", "01" AFTER 320us, "10" AFTER 640us;
```

Figura 1 - Edição dos valores de entrada do circuito. Fonte: elaboração própria.

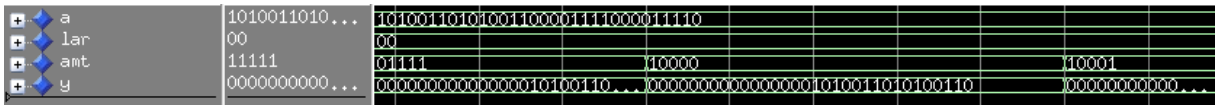

Figura 2 - Simulação do *logic_result*. Fonte: elaboração própria

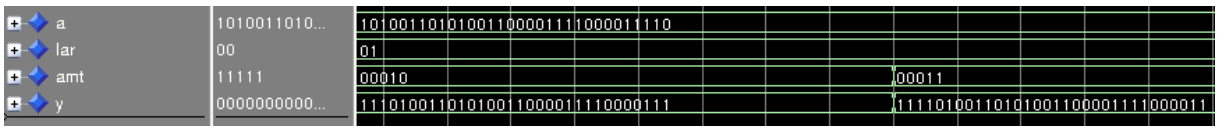

Figura 3 - Simulação do *arith_result*. Fonte: elaboração própria.

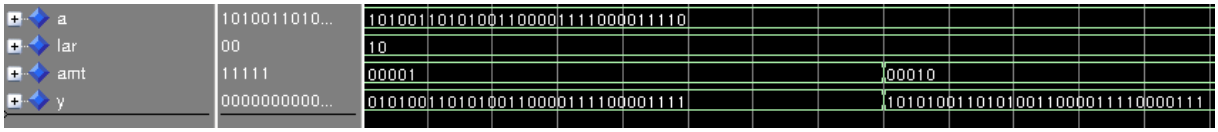

Figura 4 - Simulação do *rot_result*. Fonte: elaboração própria.

2.2 Exercício 2

Se analisarmos as três saídas das operações de deslocamento do circuito apresentado no exercício 1, verificamos que a única diferença entre os três tipos de *shifts* representados são os *bits* que preenchem o vetor de saída. Ao invés de atribuir cada uma das opções a diferentes sinais (*rot_result*, *logic_result* e *arith_result*), primeiramente verificamos qual o tipo de deslocamento que será feito, através da entrada '*lar*', e atribuirmos ao sinal *shift_in* um vetor contendo os dados para realizar a operação selecionada. Assim, os deslocamentos são realizados através desse sinal com o vetor de entrada, atribuindo o resultado diretamente à saída '*y*'.

Como esse exercício possui a mesma declaração de Entidade do exercício 1, a Arquitetura do circuito *barrel shifter* foi modificada e é apresentada no código a seguir:

```
architecture shared_arch of shift3mode_7_14 is
signal shift_in: std_logic_vector(31 downto 0);
begin
    with lar select
    shift_in <= (others=>'0')   when "00",
                (others=>a(31)) when "01",
                a               when others;
    with amt select
      y <= a                                          when "00000",
           shift_in(0)            & a(31 downto  1) when "00001",
           shift_in(1  downto 0) & a(31 downto  2) when "00010",
           shift_in(2  downto 0) & a(31 downto  3) when "00011",
           shift_in(3  downto 0) & a(31 downto  4) when "00100",
           shift_in(4  downto 0) & a(31 downto  5) when "00101",
           shift_in(5  downto 0) & a(31 downto  6) when "00110",
           shift_in(6  downto 0) & a(31 downto  7) when "00111",
           shift_in(7  downto 0) & a(31 downto  8) when "01000",
           shift_in(8  downto 0) & a(31 downto  9) when "01001",
           shift_in(9  downto 0) & a(31 downto 10) when "01010",
           shift_in(10 downto 0) & a(31 downto 11) when "01011",
           shift_in(11 downto 0) & a(31 downto 12) when "01100",
           shift_in(12 downto 0) & a(31 downto 13) when "01101",
           shift_in(13 downto 0) & a(31 downto 14) when "01110",
           shift_in(14 downto 0) & a(31 downto 15) when "01111",
           shift_in(15 downto 0) & a(31 downto 16) when "10000",
           shift_in(16 downto 0) & a(31 downto 17) when "10001",
           shift_in(17 downto 0) & a(31 downto 18) when "10010",
           shift_in(18 downto 0) & a(31 downto 19) when "10011",
           shift_in(19 downto 0) & a(31 downto 20) when "10100",
           shift_in(20 downto 0) & a(31 downto 21) when "10101",
           shift_in(21 downto 0) & a(31 downto 22) when "10110",
           shift_in(22 downto 0) & a(31 downto 23) when "10111",
           shift_in(23 downto 0) & a(31 downto 24) when "11000",
           shift_in(24 downto 0) & a(31 downto 25) when "11001",
           shift_in(25 downto 0) & a(31 downto 26) when "11010",
           shift_in(26 downto 0) & a(31 downto 27) when "11011",
           shift_in(27 downto 0) & a(31 downto 28) when "11100",
           shift_in(28 downto 0) & a(31 downto 29) when "11101",
           shift_in(29 downto 0) & a(31 downto 30) when "11110",
           shift_in(30 downto 0) & a(31)           when others;
end shared_arch;
```

Na opção em que ’*amt*’ = "00000"a entrada é enviada diretamente para a saída ’*y*’, já nos valores seguintes ocorre a concatenação do valor presente no sinal *shift_in* com os *bits* restantes de ’*a*’.

Com esta descrição de *hardware*, foram observados valores melhores do que os obtidos no exercício 1. Estes valores estão descritos na Tabela 3.

Tabela 3 - Resultados obtidos com a Arquitetura *shared_arch*

Número de pinos	71
Número de elementos lógicos - modo normal	255
Número de elementos lógicos - modo aritmético	0
Tempo de propagação (sem logic lock)	20.691 ns
Tempo de propagação (com logic lock)	15.234 ns
Tempo de propagação interno (sem logic lock)	6.407 ns
Tempo de propagação interno (com logic lock)	5.301 ns

Utilizando o mesmo arquivo de teste do exercício 1, foram realizadas simulações para cada operação da entrada *lar* e para cada valor de *amt*. A seguir, nas Figuras 5, 6 e 7, são apresentados alguns valores de entrada e saída nos modos *rot_result*, *logic_result* e *arith_result*.

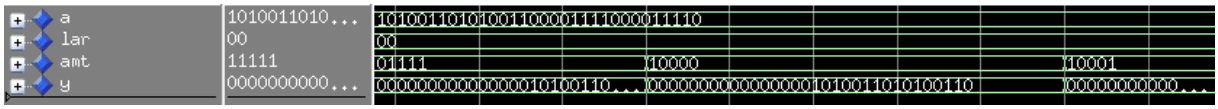

Figura 5 - Simulação do *logic_result*. Fonte: elaboração própria

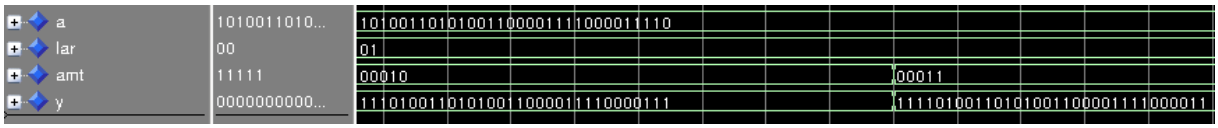

Figura 6 - Simulação do *arith_result*. Fonte: elaboração própria

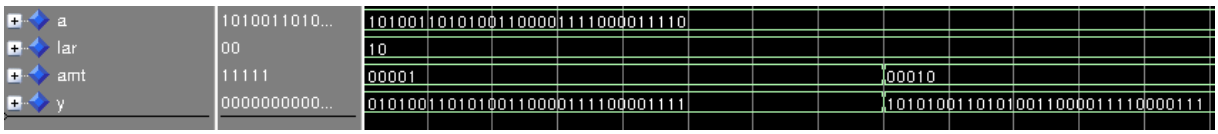

Figura 7 - Simulação do *rot_result*. Fonte: elaboração própria

2.3 Exercício 3

No último exercício proposto, temos um código que modifica a informação de entrada em uma rotação à direita de *n bits*. São três operações muito parecidas com a do exercício 2, porém, há uma mudança no número de rotações utilizadas. No exercício anterior, selecionávamos a quantidade de rotações dentro das condições de cada operação (*lar*), já, neste caso, cada operação combinacional possui definido quantas rotações vai realizar, fazendo com que cada condição dentro delas apenas varie quais os valores dos bits que serão adicionados, por exemplo, há rotações para 1, 2, 4, 8 e 16 bits, que são definidos conforme o valor de *amt*.

```
library ieee;
use ieee.std_logic_1164.all;
entity shift3mode is
    generic(
        n: integer:=32
    );
   port(
      a: in std_logic_vector(n-1 downto 0);
      lar: in std_logic_vector(1 downto 0);
      amt: in std_logic_vector(4 downto 0);
      y: out std_logic_vector(n-1 downto 0)
   );
end shift3mode ;

architecture multi_level_arch of shift3mode is
   signal le0_out, le1_out, le2_out,le3_out,le4_out:
      std_logic_vector(n-1 downto 0);
   signal le0_sin: std_logic;
   signal le1_sin: std_logic_vector(1 downto 0);
   signal le2_sin: std_logic_vector(3 downto 0);
    signal le3_sin: std_logic_vector(7 downto 0);
    signal le4_sin: std_logic_vector(15 downto 0);
begin
   -- level 0, shift 0 or 1 bit
   with lar select
    le0_sin <= '0'               when "00",
               a(n-1)      when "01",
               a(0)          when others;
   le0_out <= le0_sin & a(n-1 downto 1) when amt(0)='1' else
              a;
   -- level 1, shift 0 or 2 bits

with lar select
      le1_sin <=
         "00"                            when "00",
         (others => le0_out(n-1))      when "01",
         le0_out(1 downto 0)             when others;
   le1_out <= le1_sin & le0_out(n-1 downto 2)
                when amt(1)='1' else
              le0_out;
   -- level 2, shift 0 or 4 bits

    with lar select
```

```
      le2_sin <=
         "0000"                          when "00",
         (others => le1_out(n-1))      when "01",
         le1_out(3 downto 0)             when others;
   le2_out <= le2_sin & le1_out(n-1 downto 4)
               when amt(2)='1' else
            le1_out;

    -- level 3, shift 0 or 8 bits

    with lar select
      le3_sin <=
         "00000000"                  when "00",
         (others => le2_out(n-1))      when "01",
         le2_out(7 downto 0)             when others;
   le3_out <= le3_sin & le2_out(n-1 downto 8)
               when amt(3)='1' else
            le2_out;
    -- level 4, shift 0 or 16 bits

    with lar select
      le4_sin <=
         "0000000000000000"          when "00",
         (others => le3_out(n-1))      when "01",
         le2_out(15 downto 0)            when others;
   le4_out <= le4_sin & le3_out(n-1 downto 16)
               when amt(4)='1' else
            le3_out;
   -- output
   y <= le4_out;
end multi_level_arch ;
```

Conforme visto no código, apesar de as saídas de cada operação dependerem da saída da anterior, o valor de entrada *a* é alterado por apenas uma das operações combinacionais, fazendo com que elas trabalhem de forma independente. Observa-se também que, para cada rotação, seja de 1, 2, 4, 8 ou 16 *bits*, os valores dos novos *bits* variam conforme o valor da variável *lar*. As alterações ocorrem apenas no número de bits de entrada e saída, já que o *alt* e o *amt* não variam, pois, independentemente de quantos bits tiver a entrada, a operação é sempre a mesma.

A tabela 4 possui valores dos pinos, elementos lógicos e tempo de propagação, com e sem *logic lock*, para o caminho crítico geral e interno.

Com os dados obtidos, a proposta de melhoria de tempo de delay para este exercício é idêntica à do exercício 1: Foi criado um arquivo com extensão .sdc e inserido o comando *set_max_delay -from [get_ports *] -to [get_ports *] tp*, com a intenção de otimizar o tempo de propagação das entradas e saídas e também dos componentes internos do circuito. Concluída esta etapa, observou-se que o Quartus conseguiu otimizar o circuito de forma significativa, diminuindo aproximadamente 10% do valor anterior. Os resultados obtidos podem ser observados pela Tabela 5.

Tabela 4 - Resultados obtidos com a Arquitetura *multi_level_arch*

Número de pinos	71
Número de elementos lógicos - modo normal	197
Número de elementos lógicos - modo aritmético	0
Tempo de propagação (sem logic lock)	17.203 ns
Tempo de propagação (com logic lock)	15.108 ns
Tempo de propagação interno (sem logic lock)	4.272 ns
Tempo de propagação interno (com logic lock)	4.350 ns

Tabela 5 - Resultados obtidos com a Arquitetura *multi_level_arch*

Número de pinos	71
Número de elementos lógicos - modo normal	200
Número de elementos lógicos - modo aritmético	0
Tempo de propagação (com logic lock)	10.435 ns
Tempo de propagação interno (com logic lock)	3.281 ns

Após as simulações, foi criado um *testbench*, onde tivemos uma noção mais clara sobre o comportamento do código, conforme mostrado nas figuras 8, 9, 10, 11 e 12

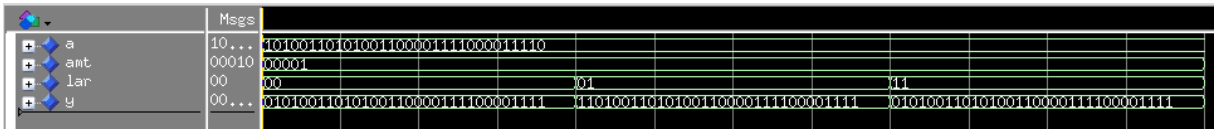

Figura 8 - Simulação do circuito no ModelSIM para *amt = '00001'*. Fonte: elaboração própria.

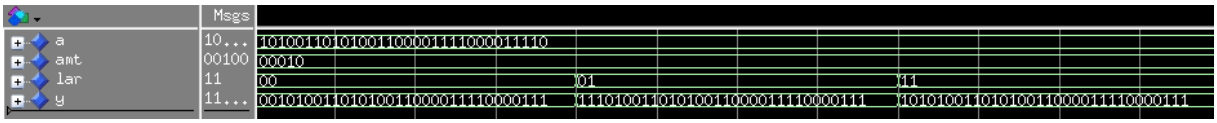

Figura 9 - Simulação do circuito no ModelSIM para *amt = '00010'*. Fonte: elaboração própria.

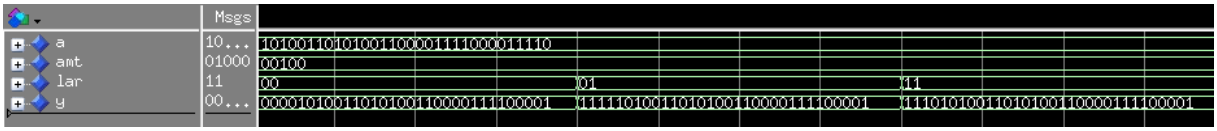

Figura 10 - Simulação do circuito no ModelSIM para *amt = '00100'*. Fonte: elaboração própria.

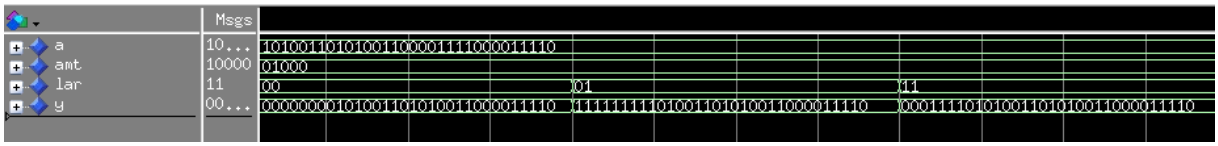

Figura 11 - Simulação do circuito no ModelSIM para *amt = '01000'*. Fonte: elaboração própria.

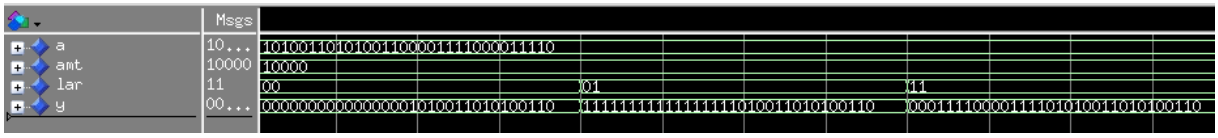

Figura 12 - Simulação do circuito no ModelSIM para *amt = '10000'*. Fonte: elaboração própria.

3 Considerações Finais

Após a resolução dos exercícios propostos, houve um maior entendimento das diferentes maneiras de implementar um mesmo circuito, priorizando um objetivo específico, como área ou *delay*. Mudanças simples no código deram resultados distintos, podendo ser melhorados em área ou em tempo de resposta. A resolução do exercício 1 era a menos complexa em termos de código, porém, não obteve resultados tão bons quanto às do exercício 2 e 3, que tinham códigos mais complexos. No exercício 3, apesar do código não ter o mesmo poder de número de rotações que os exercícios anteriores, a operação de rotação foi mantida, porém, com o uso das operações de condição de forma diferente, onde obtivemos uma resposta aproximadamente três vezes mais rápida em relação aos exercícios anteriores, mostrando que a mudança da topologia do circuito, diante de uma otimização no código, pode deixar a resposta muito mais ágil, podendo proporcionar uma implementação mais robusta, caso necessário em um momento futuro. Além de notar que manipulando o código podemos obter resultados melhores, observamos que a ferramenta também nos auxilia neste trabalho de melhoria, onde podemos setar um valor desejado de tempo de *delay* e contar com o auxílio dela para fazer modificações, tentando atender a esta exigência. Todas essas modificações em nível de código são úteis para circuitos de aplicação prática, onde, inclusive, até a disposição dos elementos influencia, não só o número deles e o tempo de resposta.

4 Referências

CHU, Pong P.. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Cleveland: Wiley-ieee Press, 2006. 694 p.