Pontifícia Universidade Católica de Minas Gerais
Instituto de Ciências Exatas e Informática
Departamento de Engenharia de Computação

Relatório: Trabalho Prático 3

Barramento Bidirecional e Memória RAM

Professor: Antônio Hamilton Magalhães

Alunos: Bruno Luiz Dias Alves de Castro
Rafael Ramos de Andrade

Belo Horizonte
Campus Coração Eucarístico

24 de novembro de 2024

Conteúdo

1 Introdução

Durante as aulas da disciplina de Sistemas Reconfiguráveis, fomos introduzidos à linguagem VHDL. VHDL (**V**HSIC **H**ardware **D**escription **L**anguage) é uma linguagem de descrição de hardware. Com ela, podemos montar circuitos lógicos de maneira totalmente textual, o que garante à linguagem uma grande vantagem ante à soluções visuais.

1.1 Objetivos

O objetivo deste terceiro trabalho prático é a implementação de uma memória ram e uma porta com barramento bidirecional. Essas estruturas serão utilizadas na construção interna do processador e para interfacear a comunicação interna e externa, respectivamente. Uma breve descrição de cada um é apresentada à seguir:

1.1.1 port_io

O bloco port_io é utilizada na comunicação entre a parte interno e externa do processador. É constituído de um barramento bidirecional configurado por um registrador nomeado tris_reg, que define quais sinais serão entrada e saída.

1.1.2 ram_mem

O bloco de ram_mem, ou memória ram, são 4 registradores de propósito geral acessados através de endereços configurados na porta de endereçamento. Cada registrador é acessado por uma faixa de endereços, suportam acesso para escrita ou leitura.

2 port_io

O port_io é um circuito que possui uma estrutura de doiis registradores de 8 bits, e um barramento biderecional de entrada e saída, para intefaceamento com a estrutura. O registrados tris_reg armazena o estado de cada bit (que poode ser entrada e saída), e o registrador port_reg armazena dados, que podem ser escritos eplo usuário através do barramento de dados dbus_in.

As entradas e saídas do circuito são descritas na tabela a baixo:

Nome	Tamanho	Tipo	Descrição
nrst	1 bit	*Input*	Entrada de *reset* assíncrono.
clk_in	1 bit	*Input*	Entrada de *clock*.
abus_in	9 bits	*Input*	Entrada de endereçamento para os registradores internos.
dbus_in	8 bits	*Input*	Entrada de habilitação para escrita nos registradores
wr_en	1 bit	*Input*	Entrada de habilitação de escrita.
rd_en	1 bit	*Input*	Entrada de habilitação de leitura.
dbus_out	8 bits	*Output*	Barramento de saída de dados, com 8 bits.
port_io	8 bits	*Inout*	Porta bidirecional, com 8 bits.

Tabela 1: Entradas e Saídas de port_io

Para o port_io, também são utilizados 4 endereços internos, implementados através de estruturas *GENERIC*, para enderaçamento dos registrados otris_reg e port_reg. São eles:

Nome	Endereço	Descrição
port_addr	0b00000011	Especifica o endereço de escrita no registrador port_reg.
tris_addr	0b00000111	Especifica o endereço de escrita no registrador tris_reg.
alt_port_addr	0b10000000	Endereço alternativo a port_addr.
alt_tris_addr	0b11000000	Endereço alternativo a tris_addr.

2.1 Implementação

O circuitop port_io foi implementado utilizando a linguagem VHDL.

O código na íntegra está abaixo:

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

ENTITY port_io IS
    GENERIC (
        -- enderecos dos registradores port a tris
        port_addr: IN STD_LOGIC_VECTOR(8 DOWNTO 0) :=  "000000011";
        tris_addr: IN STD_LOGIC_VECTOR(8 DOWNTO 0) :=  "000000111";
        alt_port_addr: IN STD_LOGIC_VECTOR(8 DOWNTO 0) :=  "100000000";
        alt_tris_addr: IN STD_LOGIC_VECTOR(8 DOWNTO 0) :=  "110000000"
    );
    PORT (
        -- Processor Side
        -- Inputs
        nrst : IN STD_LOGIC;                              -- Reset
        clk_in: IN STD_LOGIC;                             -- Clock
        abus_in: IN STD_LOGIC_VECTOR(8 DOWNTO 0);         -- Enderecamento
        dbus_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);         -- Dados
        wr_en : IN STD_LOGIC;                             -- Enable escrita
        rd_en : IN STD_LOGIC;                             -- Enable leitura

        -- Outputs
        dbus_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0); -- Dados
```

```
26
27          -- Port side
28          port_io: INOUT STD_LOGIC_VECTOR(7 DOWNTO 0) -- bidirectional port
29      );
30  END ENTITY;
31
32  ARCHITECTURE port_io OF port_io IS
33      SIGNAL port_reg: STD_LOGIC_VECTOR(7 DOWNTO 0);
34      SIGNAL tris_reg: STD_LOGIC_VECTOR(7 DOWNTO 0);
35      SIGNAL latch: STD_LOGIC_VECTOR(7 DOWNTO 0);
36
37      SIGNAL en_tris_addr: STD_LOGIC;
38      SIGNAL en_port_addr: STD_LOGIC;
39  BEGIN
40      -- verifica quais dos registradores se encontra no estado ativo
41      en_tris_addr <= '1' WHEN (abus_in = tris_addr) OR (abus_in = alt_tris_addr) ELSE '0';
42      en_port_addr <= '1' WHEN (abus_in = port_addr) OR (abus_in = alt_port_addr) ELSE '0';
43
44      -- secao sequencial
45      PROCESS(nrst, clk_in, abus_in, en_tris_addr, tris_reg, port_reg)
46      BEGIN
47          IF nrst = '0' THEN
48              port_reg <= "00000000";
49              tris_reg <= "11111111";
50          -- parte sincroina
51          ELSIF RISING_EDGE(clk_in) THEN
52              IF en_tris_addr = '1' THEN
53                  IF(wr_en = '1') THEN
54                      -- escrita
55                      tris_reg <= dbus_in;
56                  END IF;
57              END IF;
58              IF en_port_addr = '1' THEN
59                  IF(wr_en = '1') THEN
60                      -- escrita
61                      port_reg <= dbus_in;
62                  END IF;
63              END IF;
64          END IF;
65      END PROCESS;
66
67      -- leitura
68      dbus_out <= tris_reg WHEN en_tris_addr = '1' AND rd_en = '1' ELSE
69                  latch WHEN en_port_addr = '1' AND rd_en = '1' ELSE "ZZZZZZZZ";
70
71      - atualizacao do latch
72      latch <= port_io WHEN en_port_addr = '0' OR rd_en = '0';
73
74      -- altera os valores de port_io para 0 ou Z
75      port_io(0) <= port_reg(0) WHEN tris_reg(0) = '0' ELSE 'Z';
76      port_io(1) <= port_reg(1) WHEN tris_reg(1) = '0' ELSE 'Z';
77      port_io(2) <= port_reg(2) WHEN tris_reg(2) = '0' ELSE 'Z';
78      port_io(3) <= port_reg(3) WHEN tris_reg(3) = '0' ELSE 'Z';
79      port_io(4) <= port_reg(4) WHEN tris_reg(4) = '0' ELSE 'Z';
80      port_io(5) <= port_reg(5) WHEN tris_reg(5) = '0' ELSE 'Z';
81      port_io(6) <= port_reg(6) WHEN tris_reg(6) = '0' ELSE 'Z';
82      port_io(7) <= port_reg(7) WHEN tris_reg(7) = '0' ELSE 'Z';
83
84  END port_io;
```

Listing 1: Código VHDL w_reg

2.1.1 Descrição do funcionamento

O circuito port_io consiste de de duas partes: uma síncrona e uma assíncrona. A parte síncrona é composta pelas funções de *reset* e escrita. Já a parte assícrona pelas funções de leitura e endereçamento.

Inicialmente, verifica-se se alguns dos registrados internos (port_reg e tris_reg) estão devidamente endereçados pela entrada abus_in. Para isso, utilizamos um *SIGNAL* auxiliar. O resultado será usado por ambas as partes síncrona e assíncrona.

Dentro do *PROCESS*, a primeira função é a de *reset* (ativo por nrst em baixo.), implementada de maneira assícrona (isto é, independente de uma borda de subida de *clock*). Quando

ocorrer, registrador port_reg é zerado, e o tris_reg tem todos os bits setados em '1'.

Ainda dentro do *PROCESS*, são implementadas as funções de escrita sícrona. Após uma borda de subida de *clock*, caso devidamente endereçado (feito anteriormente) e sinal wr_en ativo, o registrados correspondente é escrito.

Fora do *PROCESS*, de maneira assíncrona, são atualizadas a saída dbus_out, com wr_en ativo e enderaçamento correto, e o latch, caso não haja uma leitura de port_reg.

Os bits de port_io são setados individualmente a alta impedância caso estejam configurados como saída.

2.2 Simulação

Para testar nosso código VHDL e certificar-nos de que nosso circuito funciona de maneira esperada, simulamos alguns casos de testes utilizando o software Quatus II.

Os testes realizados foram os seguites:

1. Testa exemplo do relatório.
   - tris_reg = "00001111" ou 0x0F;
   - port_io = "ZZZZ1111" ou 0xZF;
   - port_reg = "10101111" ou 0xAF;
   - **Comportamento esperado:**
     - dbus_out = "10101111" ou 0xAF;
     - port_io_result = "10101111" ou 0xAF;
2. Teste com port_io entradas e saídas alternadas.
   - tris_reg = "10101010" ou 0xAA;
   - port_io = "1Z1Z1Z1Z";
   - port_reg = "11111111" ou 0xFF;
   - **Comportamento esperado:**
     - dbus_out = "11111111" ou 0xFF;
     - port_io_result = "10101010" ou 0xAA;
3. Teste com port_io saídas e entradas alternadas.
   - tris_reg = "01010101" ou 0x55;
   - port_io = "Z1Z1Z1Z1";
   - port_reg = "11111111" ou 0xFF;
   - **Comportamento esperado:**
     - dbus_out = "11111111" ou 0xFF;
     - port_io_result = "01010101" ou 0x55;

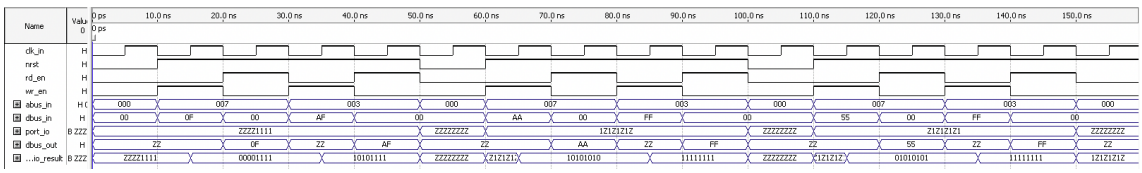

Figura 1: Simulação do circuito port_io

3 ram_mem

O bloco ram_mem é um circuito que funciona como uma memória RAM. Ele é dividido em 4 blocos: 3 de 80 bytes, e 1 de 16 bytes. A escrita e leitura em cada um desses blocos é invisível para o usuário. Isto é, apenas um espaço de endereçamento é utilizado, e o circuito deve manejar em qual bloco escrever, ou ler, de acordo com a especificação.

Nome	Tamanho	Tipo	Descrição
nrst	1 bit	*Input*	Entrada de *reset* assíncrono.
clk_in	1 bit	*Input*	Entrada de *clock*.
abus_in	9 bit	*Input*	Entrada de enderençamento.
dbus_in	8 bits	*Input*	Entrada de dados para escrita.
wr_en	1 bit	*Input*	Entrada de habilitação de escrita.
rd_en	1 bit	*Input*	Entrada de habilitação de leitura.
dbus_out	8 bits	*Output*	Saída de dados hailitada por rd_en.

Tabela 2: Entradas e Saídas de ram_mem

A memória é divida em 4 blocos, com espaço de endereçamento compartilhado. Dependendo do endereço sendo lido/escrito, um bloco de memória diferente deve ser acessado, de acordo com a tabela abaixo:

Bloco	Faixa de endereçamento
mem0	020h $\sim$ 06Fh (80 bytes). Em decimal: 32 a 111
mem1	0A0h $\sim$ 0EFh (80 bytes). Em decimal: 160 a 239
mem2	20h $\sim$ 16Fh (80 bytes). Em decimal: 288 a 367
mem_com	070h $\sim$ 07Fh (16 bytes). Em decimal: 112 a 127

Tabela 3: Entradas e Saídas de ram_mem

A área de memória mem_com também pode ser endereçada através dos endereços 0F0h $\sim$ 0FFh, 170h $\sim$ 17Fh ou 1F0h $\sim$ 1FFh. Dessa forma os bits 8 e 7 de abus_in não importam para o endereçamento dessa área específica, sendo utilizados apenas os bits 6 a 0.

3.1 Implementação

O circuito ram_mem foi implementado utilizando a linguagem VHDL.

O código na íntegra está abaixo:

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_unsigned.all;
4  USE ieee.numeric_std.all;
5
6  ENTITY ram_mem IS
7      PORT (
8          -- Inputs
9          nrst : IN STD_LOGIC;                              -- Reset
10         clk_in: IN STD_LOGIC;                             -- Clock
11         abus_in: IN STD_LOGIC_VECTOR(8 DOWNTO 0);         -- Enderecamento
12         dbus_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);         -- Dados
13         wr_en : IN STD_LOGIC;                             -- Enable escrita
14         rd_en : IN STD_LOGIC;                             -- Enable leitura
15
16         -- Outputs
17         dbus_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)       -- Dados
18     );
19 END ENTITY;
20
21 ARCHITECTURE ram_mem OF ram_mem IS
```

```
    TYPE mem_type0 IS ARRAY(0 TO 79) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
    TYPE mem_type1 IS ARRAY(0 TO 15) OF STD_LOGIC_VECTOR(7 DOWNTO 0);

    SIGNAL mem0: mem_type0;
    SIGNAL mem1: mem_type0;
    SIGNAL mem2: mem_type0;
    SIGNAL mem_com: mem_type1;
    SIGNAL addr_int : INTEGER RANGE 0 TO 511;

BEGIN
    addr_int <= TO_INTEGER(UNSIGNED(abus_in));

    PROCESS(clk_in, nrst, addr_int, rd_en)
    BEGIN
        IF RISING_EDGE(clk_in) THEN
            IF wr_en = '1' THEN
                CASE addr_int IS
                    -- mem0 80 bytes
                    WHEN 32 TO 111 =>
                        mem0(addr_int - 32) <= dbus_in;

                    -- mem1 80 bytes
                    WHEN 160 TO 239 =>
                        mem1(addr_int - 160) <= dbus_in;

                    -- mem2 80 bytes
                    WHEN 288 TO 367 =>
                        mem2(addr_int - 288) <= dbus_in;

                    -- mem_com 16 bytes
                    WHEN 112 TO 127 =>
                        mem_com(addr_int - 112) <= dbus_in;
                    WHEN 240 TO 255 =>
                        mem_com(addr_int - 240) <= dbus_in;
                    WHEN 368 TO 383 =>
                        mem_com(addr_int - 368) <= dbus_in;
                    WHEN 496 TO 511 =>
                        mem_com(addr_int - 496) <= dbus_in;

                    -- default
                    WHEN OTHERS =>
                END CASE;
            END IF;

        END IF;

        IF nrst = '0' THEN
            mem0 <= (OTHERS => (OTHERS => '0'));
            mem1 <= (OTHERS => (OTHERS => '0'));
            mem2 <= (OTHERS => (OTHERS => '0'));
            mem_com <= (OTHERS => (OTHERS => '0'));
        END IF;
    END PROCESS;

    PROCESS(clk_in, addr_int, rd_en, mem0, mem1, mem2, mem_com)
    BEGIN
        IF rd_en = '1' THEN
            CASE addr_int IS
                -- mem0 80 bytes
                WHEN 32 TO 111 =>
                    dbus_out <= mem0(addr_int - 32);

                -- mem1 80 bytes
                WHEN 160 TO 239 =>
                    dbus_out <= mem1(addr_int - 160);

                -- mem2 80 bytes
                WHEN 288 TO 367 =>
                    dbus_out <= mem2(addr_int - 288);

                -- mem_com 16 bytes
                WHEN 112 TO 127 =>
                    dbus_out <= mem_com(addr_int - 112);
                WHEN 240 TO 255 =>
                    dbus_out <= mem_com(addr_int - 240);
                WHEN 368 TO 383 =>
```

```
98                      dbus_out <= mem_com(addr_int - 368);
99                  WHEN 496 TO 511 =>
100                     dbus_out <= mem_com(addr_int - 496);
101
102                 -- default
103                 WHEN OTHERS =>
104             END CASE;
105         ELSE
106             dbus_out <= "ZZZZZZZZ";
107         END IF;
108     END PROCESS;
109 END ram_mem;
```

Listing 2: Código VHDL fsr_reg

3.1.1 Descrição do funcionamento

O circuito implementado possui duas partes: uma síncrona e outra assíncrona. A leitura é feita de forma síncrona, isto é, junto de uma borda de subida do *clock*, já a leitura é feita de forma assícrona.

Primeiramente, se converte o sinal de entrada abus_in para um inteiro, utilizando a função *TO_INTEGER()*. Com isso, podemos identificasr se o endereço sendo lido/escrito está dentro dos intervalos identificados.

Dentro da seção síncrona, após uma borda de subida de *clock*, os dados são escritos no bloco correto, caso rd_en esteja ativo. Já na parte assícrona, caso wr_en esteja ativo, o valor endereçado é colocado na saída, acessado de acordo com o bloco utilizado.

O reset é feito de maneira assícrona, e possui preferência sobre a escrita, pis é feita de maneira processural depois dela.

3.2 Simulação

Para testar nosso código VHDL e certificar-nos de que nosso circuito funciona de maneira esperada, simulamos alguns casos de testes utilizando o software Quartus II.

Os testes realizados foram os seguites:

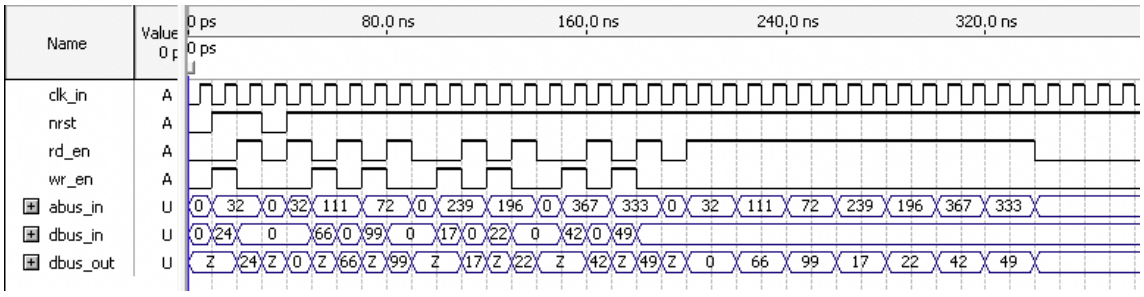

Figura 2: Simulação realizada com todos os blocos em conjunto.

3.2.1 Registrador mem0

1. Escrita e leitura com menor endereço possível.
   - abus_in = 32;
   - dbus_in = 24;
   - **Comportamento esperado:**
     - dbus_out = 24;
2. Testa leitura após reset.

- abus_in = 32;
- dbus_in = 0;
- **Comportamento esperado:**
  - dbus_out = 0;

3. Escrita e leitura com maior endereço possível.
   - abus_in = 111;
   - dbus_in = 66;
   - **Comportamento esperado:**
     - dbus_out = 66;
4. Escrita e leitura com endereço aleatório.
   - abus_in = 72;
   - dbus_in = 99;
   - **Comportamento esperado:**
     - dbus_out = 99;

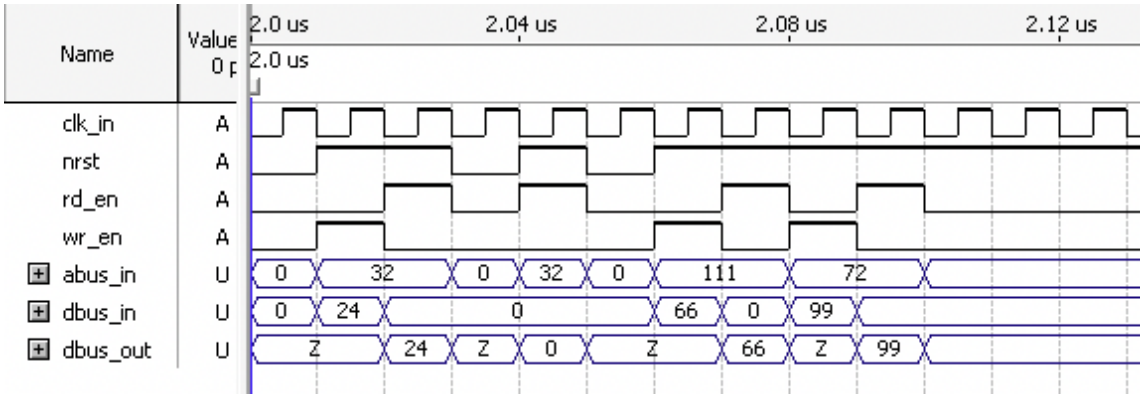

Figura 3: Simulação mem0

3.2.2 Registrador mem1

1. Escrita e leitura com menor endereço possível.
   - abus_in = 160;
   - dbus_in = 13;
   - **Comportamento esperado:**
     - dbus_out = 13;
2. Testa leitura após reset.
   - abus_in = 160;
   - dbus_in = 0;
   - **Comportamento esperado:**
     - dbus_out = 0;
3. Escrita e leitura com maior endereço possível.

- abus_in = 239;
- dbus_in = 17;
- **Comportamento esperado:**
  - dbus_out = 17;

4. Escrita e leitura com endereço aleatório.
   - abus_in = 196;
   - dbus_in = 22;
   - **Comportamento esperado:**
     - dbus_out = 22;

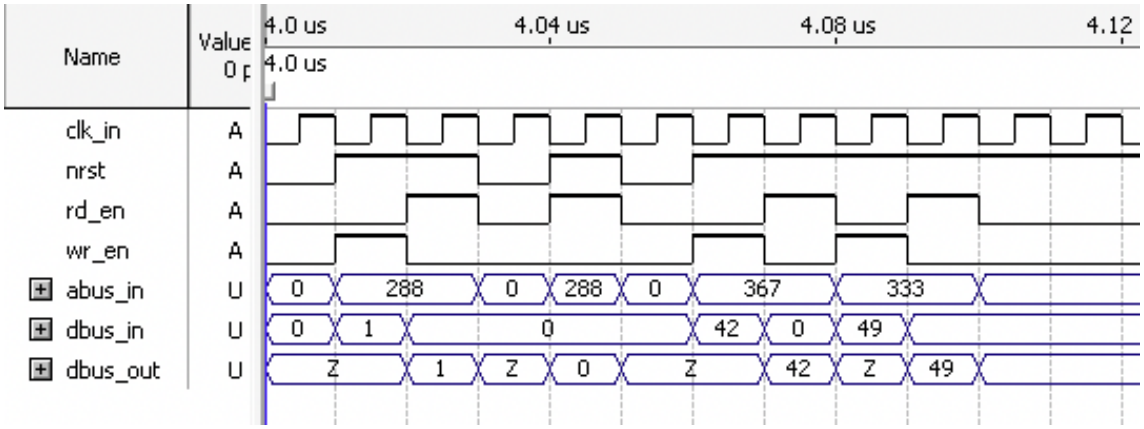

Figura 4: Simulação mem1

3.2.3 Registrador mem2

1. Escrita e leitura com menor endereço possível.
   - abus_in = 288;
   - dbus_in = 1;
   - **Comportamento esperado:**
     - dbus_out = 1;
2. Testa leitura após reset.
   - abus_in = 288;
   - dbus_in = 0;
   - **Comportamento esperado:**
     - dbus_out = 0;
3. Escrita e leitura com maior endereço possível.
   - abus_in = 367;
   - dbus_in = 17;
   - **Comportamento esperado:**
     - dbus_out = 17;
4. Escrita e leitura com endereço aleatório.

- abus_in = 333;
- dbus_in = 49;
- **Comportamento esperado:**
  - dbus_out = 49;

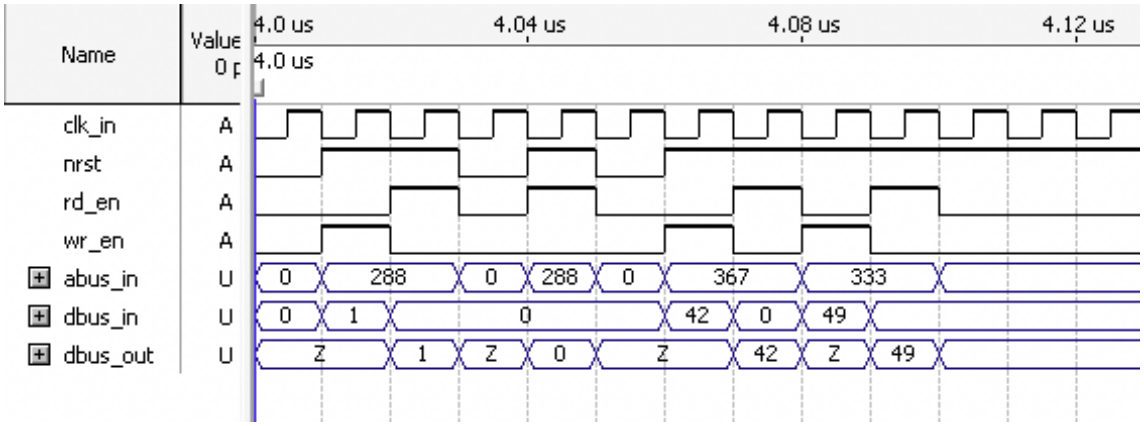

Figura 5: Simulação mem2

3.2.4 Registrador mem_com

1. Escrita e leitura com menor endereço possível.
   - abus_in = 112;
   - dbus_in = 24;
   - **Comportamento esperado:**
     - dbus_out = 24;
2. Testa leitura após reset.
   - abus_in = 112;
   - dbus_in = 0;
   - **Comportamento esperado:**
     - dbus_out = 0;
3. Escrita e leitura com endereço 127.
   - abus_in = 127;
   - dbus_in = 66;
   - **Comportamento esperado:**
     - dbus_out = 66;
4. Escrita e leitura com endereço 120.
   - abus_in = 120;
   - dbus_in = 100;
   - **Comportamento esperado:**
     - dbus_out = 100;
5. Escrita e leitura com endereço 245.

- abus_in = 245;
- dbus_in = 37;
- **Comportamento esperado:**
  - dbus_out = 37;

6. Escrita e leitura com endereço 370.
   - abus_in = 370;
   - dbus_in = 123;
   - **Comportamento esperado:**
     - dbus_out = 123;
7. Escrita e leitura com endereço 500.
   - abus_in = 500;
   - dbus_in = 8;
   - **Comportamento esperado:**
     - dbus_out = 8;

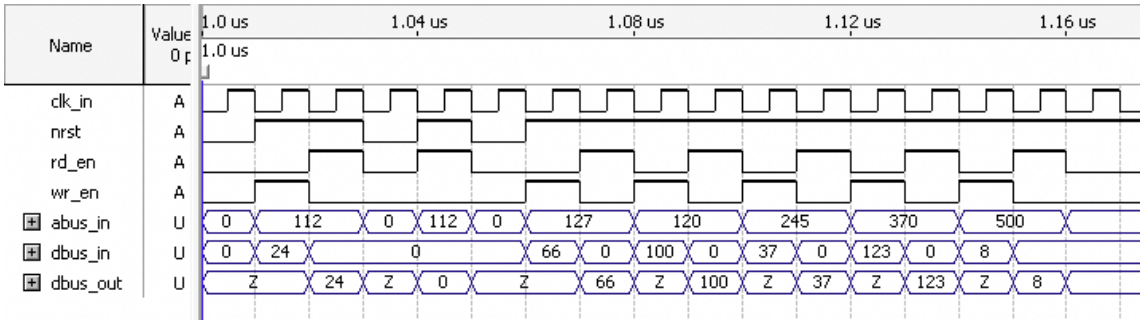

Figura 6: Simulação mem_com

4 Conclusão

A implementação de registradores através de liguagens de descrição de hardware, como a VHDL, são maneiras poderosas de construir circuitos computacionais de maneira menos complexa e mais rápida.

Apesar de parecerem "simples" à primeira vista, sua implementação possuem nuancias que exigem a atenção do engenheiro, e testes exaustivos para certificar que funcionam como deveriam.

Neste trabalho prático, aprendemos como construir registradores e pilhas para auxiliar a contrução de nossos circuitos computacionais, utilizando conhecimentos da prática anterior e novas técnicas, como a utilização do "PROCESS", que permite a execução de trechos sequenciais no nosso circuito, e a tuilização do *clock*, que permite a sincronização de rotinas no nosso cicuito.

Com os circuitos implementados, estamos mais confiantes nas nossas capacidades, e estamos um passo mais perto de implementar circuitos mais complexos como controladores e processadores.