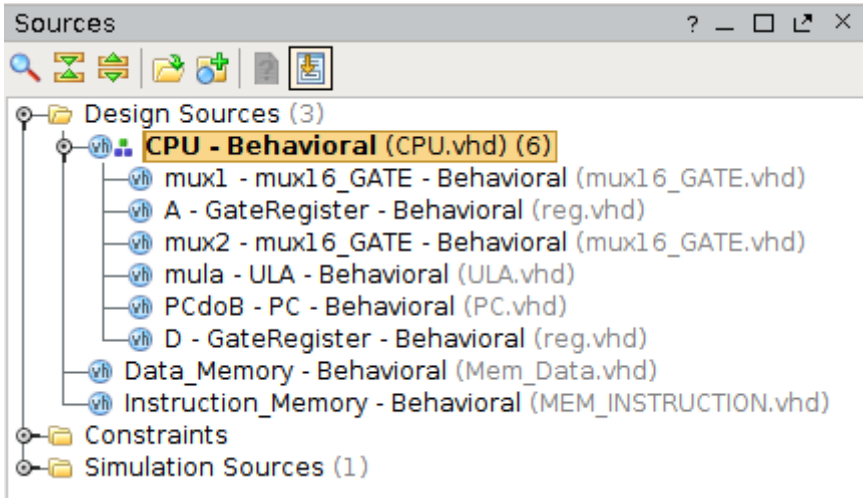
Atividade do Capítulo 5

Nome: Mateus Sousa Araújo – **Matrícula:** 374858
Nome: José Wesley Araújo – **Matrícula:** 374855

Para a realização deste trabalho fizemos a importação de todos os trabalhos até então já implementados anteriormente. Para isso fizemos a importação do registrador 16 bits, a ULA, o PC, e o flip flop D. No arquivo CPU.vhd temos toda a declaração destes componentes e a criação de diversos sinais. Esses sinais nos servirão de apoio para as conexões da implementação da CPU de acordo com o livro. Na CPU.vhd foi declarado componentes de cada trabalho antes implementado. Logo depois foram criados os sinais que serão correspondentes as ligações feitas de acordo com o diagrama apresentado abaixo.



Importação de arquivos

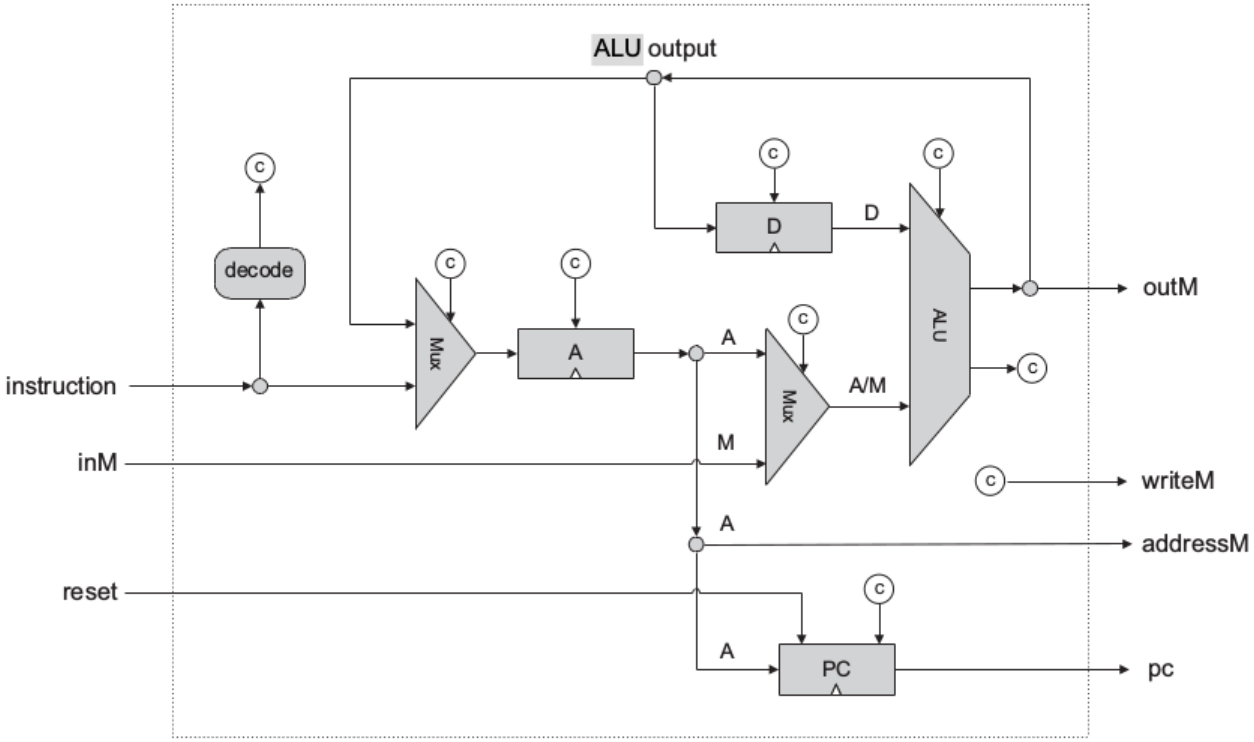


Diagrama de conecção entre componentes para implementação da CPU

Primeiramente fizemos um sinal para a saída da ula (saida_ula) com tamanho de 16 bits. Depois um sinal para a saída do MUX1 de tamanho 16 (saida_mux1). Um sinal para a saída do registrador A (saida_A). Uma saída para o multiplexador 2 (outM2). Uma saída para o flip flop D (saida_D). Um sinal para Unidade de controle (UC). Depois fizemos sinais correspondentes aos jumpers: jgt, jeq, jlt e jmp. Criamos um sinal de seleção (key). Fizemos mais outros 2 sinais para a ULA, (z_zero e s_neg).

Logo em seguida fizemos o mapeamento de cada componente de acordo com o diagrama acima. Temos para cada componente um bit de controle "c" que será distribuido pela unidade de controle. A UC fará o papel de pegar as instruções de 16 bits e fazer a distribuição de cada bit aos seus correspondentes controles "c". Fizemos o mapeamento do 1º mux do diagrama com os sinais de saída da ULA, a entrada da instrução, a seleção e a saída correspondente para este mux. Logo após fizemos o mapeamento para o Registrador A. De acordo com o diagrama apresnetado acima, o registrador A a saída do mux anterior, um bit de controle com uma instrução na posição 5, um clock e uma saída para este dispositivo. Para o próximo mux (MUX 2) fizemos o mapeamento de acordo com o diagrama apresentado acima. O MUX 2 recebe a saída do registrador A, uma entrada para o mux (inM), um bit de controle na posição 12 e uma saída (saida_A). Fizemos agora o mapeamento da ULA. A ULA recebe como entrada a saída do registrador D, a saída do MUX anterior (MUX2). O restante dos bits de instrução que sobraram irão para os bits de controle. Dessa forma, os bits na posição 11, 10, 9, 8, 7 e 6 irão para o controle da ULA. A ula possui um sinal de saída (saida_ula), e os sinais correspondentes para zero e negado (s_zero e s_neg). Logo após a ULA, mapeameamos o PC. O program counter de acordo com o diagrama apresentado recebe a saída do registrador A (saida_A), um clock (clk), um sinal de jmp, outro clock, um sinal de reset, e um sinal de saída para o PC. O flip flop D recebe como entrada a saída da ULA, um controle de instrução na posição 4, um clock e uma saída para este componente (saida_D). Depois disso, fizemos um sinal geral receber a saída do flip flop D (s_D <= saida_D). Abaixo segue um esquemático do código do que fizemos até aqui.

```
component GateRegister is
    Port ( input : in STD_LOGIC_VECTOR (15 downto 0);
           load  : in STD_LOGIC;
           clk   : in STD_LOGIC;
           output: out STD_LOGIC_VECTOR (15 downto 0));
end component;

component PC is
    Port ( input  : in STD_LOGIC_VECTOR (15 downto 0);
           inc    : in STD_LOGIC;
           load   : in STD_LOGIC;
           clk    : in STD_LOGIC;
           reset  : in STD_LOGIC;
           output : out STD_LOGIC_VECTOR (15 downto 0));
end component;

component mux16_GATE is
    Port ( a : in STD_LOGIC_VECTOR (15 downto 0);
           b : in STD_LOGIC_VECTOR (15 downto 0);
           sel : in STD_LOGIC;
           x : out STD_LOGIC_VECTOR (15 downto 0));
end component;
```

Declaração dos componentes do registrador A, do PC e do MUX16

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CPU is
    Port ( instruction  : in STD_LOGIC_VECTOR (15 downto 0);
           inM          : in STD_LOGIC_VECTOR (15 downto 0);
           reset        : in STD_LOGIC;
           clk          : in std_logic;
           outM         : out STD_LOGIC_VECTOR (15 downto 0);
           writeM       : out STD_LOGIC;
           address      : out STD_LOGIC_VECTOR (15 downto 0);
           pc_out       : out STD_LOGIC_VECTOR (15 downto 0);
           s_D          : out std_logic_vector (15 downto 0));
end CPU;

architecture Behavioral of CPU is

    component ULA is
        Port ( x    : in STD_LOGIC_VECTOR (15 downto 0);
               y    : in STD_LOGIC_VECTOR (15 downto 0);
               zx   : in STD_LOGIC;
               nx   : in STD_LOGIC;
               zy   : in STD_LOGIC;
               ny   : in STD_LOGIC;
               f    : in STD_LOGIC;
               no   : in STD_LOGIC;
               s    : out STD_LOGIC_VECTOR (15 downto 0);
               zr   : out STD_LOGIC;
               ng   : out STD_LOGIC);
    end component;

    component GateRegister is
        Port ( input : in STD_LOGIC_VECTOR (15 downto 0);
               load  : in STD_LOGIC;
               clk   : in STD_LOGIC;
               output: out STD_LOGIC_VECTOR (15 downto 0));
    end component;
```

Declaração dos componentes da CPU e da ULA

Uma observação importante a fazer é que já que estamos apenas importando componentes e juntando-os ao escopo do projeto, achamos não entrar em muitos detalhes sobre os mesmos, já que esses foram implementados anteriormente e já foram explicados nos trabalhos passados com imagens e simulações. Iremos explicar a continuação e as partes adicionais como a CPU geral e os arquivos dos dados e instruções de memória.

```
--=============================================================================================
--      SINAIS DA ULA, SAIDA DO MUX1, MUX2, REGISTRADOR D, REGISTRADOR A E DEMAIS NECESSÁRIOS
--=============================================================================================
    signal saida_ula: std_logic_vector (15 downto 0);
    signal saida_mux1 : std_logic_vector (15 downto 0);
    signal saida_A  : std_logic_vector (15 downto 0);
    signal outM2 : std_logic_vector (15 downto 0);
    signal saida_D  : std_logic_vector (15 downto 0);
    signal UC    : std_logic_vector (15 downto 0);
    signal jgt   : std_logic;
    signal key   : std_logic;
    signal jeq   : std_logic;
    signal jlt   : std_logic;
    signal jmp   : std_logic;
    signal s_zero   : std_logic;
    signal s_neg   : std_logic;


begin

    mux1 : mux16_GATE port map(saida_ula, instruction, key, saida_mux1);
    A    : GateRegister port map(saida_mux1, UC(5), clk, saida_A);
    mux2 : mux16_GATE port map(saida_A, inM, UC(12), outM2);
    mula : ULA port map(saida_D, outM2, UC(11), UC(10), UC(9), UC(8), UC(7), UC(6), saida_ula, s_zero, s_neg);
    PCdoB: PC port map(saida_A, clk, jmp, clk, reset, pc_out);
    D    : GateRegister port map(saida_ula, UC(4), clk, saida_D);

    s_D <= saida_D;
```

Declaração dos sinais de acordo com o diagrama e o mapeamento dos componentes

```
    process(clk)
    begin
        if falling_edge(clk) then
        outM <= saida_ula;
        end if;
    end process;

    with instruction(15) select
    UC <= "0110110000100000" when '0',
          instruction when others;

    writeM <= UC(3) and clk;
    address <= saida_A;
    key <= not UC(15);

    jgt <= UC(0) and (not s_zero) and (not s_neg);
    jeq <= UC(1) and s_zero;
    jlt <= UC(2) and (not s_zero) and s_neg;
    jmp <= jgt or jeq or jlt;


end Behavioral;
```

Processo com definição da instrução, clock, bits de instrução e escrita

Na figura acima definimos o nível de sensibilidade do clock com borda de descida. Fizemos a saída geral (outM) receber a saída da ula (saida_ula). Fizemos a definição de uma instrução fazendo UC receber “0110110000100000”. Fizemos a operação de escrita (WriteM) com o bit 15 de instrução. Fizemos o endereço receber a saída do registrador A e a seleção receber a negação do bit 15 da instrução UC. Em seguida fizemos as definições dos jumpers fazendo operações dos respectivos bits de instrução com as flags da ULA (s_zero e not s_neg).

Dados de memória (Mem_data.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Data_Memory is
    Port ( input   : in STD_LOGIC_VECTOR (15 downto 0);
           address : in STD_LOGIC_VECTOR (15 downto 0);
           load    : in STD_LOGIC;
           clk     : in STD_LOGIC;
           output  : out STD_LOGIC_VECTOR (15 downto 0));
end Data_Memory;

architecture Behavioral of Data_Memory is

    type memory is array(24576 downto 0) of std_logic_vector(15 downto 0);
    signal bank: memory;
    signal ris_edge: std_logic := '0';
```

Acima fizemos a implementação do arquivo responsável pelos dados de memória. Fizemos a criação de 4 variáveis de entrada e uma de saída. Input, endereço, load e clock e uma saída de output. Fizemos uma memória do tipo array com capacidade de 24578 posições em que cada posição fosse armazenada 16 posições. Fizemos um sinal “pente” do tipo memória e em seguida fizemos a definição do nível ativo do clock que será de descida.

```
begin

    process(clk)
    variable indice: integer;
    begin
        if rising_edge(clk) then
            indice := to_integer(unsigned(address));
            output <= pente(indice);
            ris_edge <= not ris_edge;
        end if;
    end process;

    process(load, ris_edge)
    variable indice: integer;
    begin
        if load = '1' then
            indice := to_integer(unsigned(address));
            pente(indice) <= input;
        end if;
    end process;

end Behavioral;
```

Acima temos um processo em que faremos uma leitura e armazenamento dos dados na memória. A cada pulso os dados são lidos e armazenados nas posições correspondentes de memória. Temos uma variável do tipo indice para controle de posição.

Instruções de memória (MEM_INSTRUCTION.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;
use STD.TEXTIO.all;
use ieee.std_logic_textio.all;

entity Instruction_Memory is
    Port ( address : in STD_LOGIC_VECTOR (15 downto 0);
           load    : in STD_LOGIC;
           clk     : in STD_LOGIC;
           output  : out STD_LOGIC_VECTOR (15 downto 0));
end Instruction_Memory;

architecture Behavioral of Instruction_Memory is

    type memory is array(32767 downto 0) of std_logic_vector(15 downto 0);
    signal pente: memory;
    signal ris_edge: std_logic := '0';
    file arquivo_dados : text;

begin

    process(clk)
    variable indice: integer;
    begin
        if rising_edge(clk) then
            indice := to_integer(unsigned(address));
            output <= pente(indice);
            ris_edge <= not ris_edge;
        end if;
    end process;

    process(load)
    variable indice       : integer;
    variable s_address    : std_logic_vector(15 downto 0) := x"0000";
    variable v_ILINE      : line;
    variable v_OLINE      : line;
    variable v_READ       : std_logic_vector(15 downto 0);
```

Acima temos um código em VHDL que lida com as instruções pegadas da memória. Foram criadas variáveis semelhantes a leitura e armazenamento na entidade. A única diferença é que foi criado um arquivo com dados. Para isso foi criado uma variável “arquivo_dados”. Foi criado um índice de controle e variáveis v_ILINE e v_OLINE para leitura do arquivo.

```
    begin
        if load = '1' then
            file_open(arquivo_dados, "/home/wesley/Documents/QUINTO_SEMESTRE/SISTEMAS_DIGITAIS/MEMORIA/file_posicoes.hack",  read_mode);

            while not endfile(arquivo_dados) loop
                readline(arquivo_dados, v_ILINE);
                read(v_ILINE, v_READ);
                indice := to_integer(unsigned(s_address));
                pente(indice) <= v_READ;
                s_address := std_logic_vector(unsigned(s_address) + 1);

            end loop;
            file_close(arquivo_dados);

        end if;
    end process;

end Behavioral;
```

Acima está o processo de leitura do arquivo. Um arquivo foi criado para modo de leitura onde o índice recebe o endereço fixado. A variável “pente” em cada índice do laço recebe o modo de leitura. No final, o endereço recebe um vetor de cada sinal de endereço somado com 1. No final da leitura do arquivo, o arquivo é fechado.

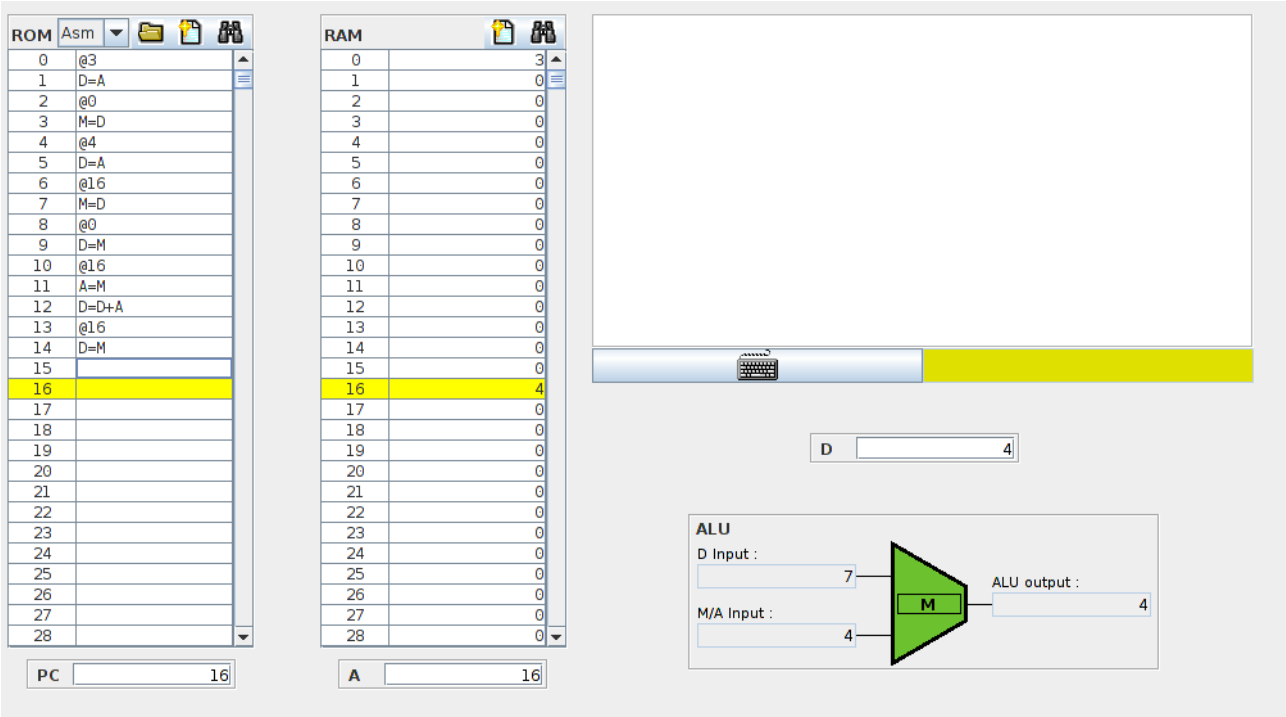
```
@3
D = A
@0
M = D  // m[0] = 3
@4
D = A
@sum
M = D
@0
D = M
@sum
A = M
D = A+D
@0

```

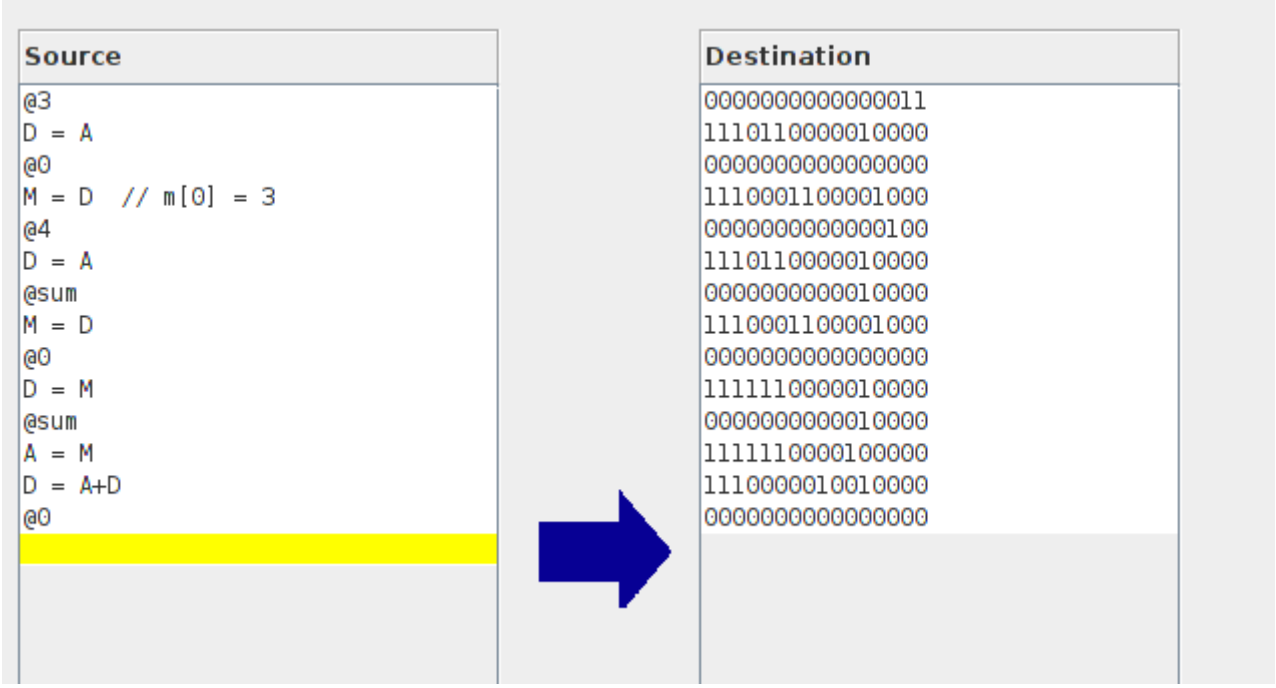
Acima implementamos um código que faz a soma de 2 números, 4 e 3 em Assembly Hack. Fizemos a conversão parao código binário para colocar no arquivo que foi gerado e fazer a simulação. Abaixo se encontra o binário correspondente as instruções do código da soma acima.

```
0000000000000011
1110110000010000
0000000000000000
1110001100001000
0000000000000100
1110110000010000
0000000000010000
1110001100001000
0000000000000000
1111110000010000
0000000000010000
1111110000100000
```

Geração de instruções a partir do Assembly

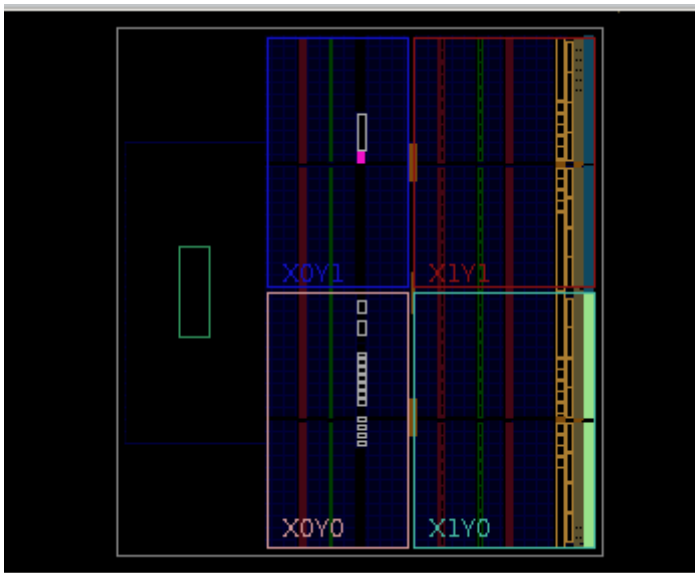


Código da soma em simulação na CPU Emulator

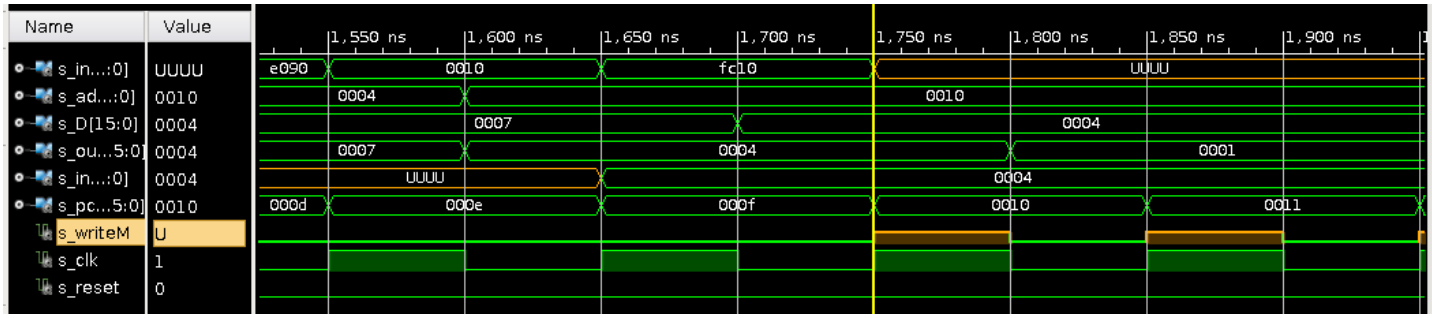


Geração de instruções a partir do simulador Assembler

Abaixo segue as simulações no vivado. Na simulação podemos perceber o valor da soma em cada instrução pelos valores de cada sinal/variável correspondente.



Device implementation



Behavioral Simulation