Universidade Federal do Rio Grande do Norte
Centro de Tecnologia - CT
Departamento de Engenharia Elétrica

Modelo de Relatório Técnico
ELE1717 - Grupo 03 - Problema 02 - Implementação

Anny Beatriz Pinheiro Fernandes
Arthur Felipe Rodrigues Costa
Isaac de Lyra Junior
Wesley Brito da Silva

Natal, 6 de julho de 2021

Resumo

Este relatório tem o objetivo de demonstrar a construção e implementação de uma CPU de 16 instruções, que foi idealizada pelo grupo 3 da semana passada. Esta CPU (Unidade Central de Processamento) foi projetada com o objetivo de realizar algumas instruções já pré-definidas pelas especificações de projeto. Essas instruções possuem o objetivo de realizar operações aritméticas em um bloco de ULA (Unidade Lógica Aritmética). Neste projeto, foi feito uma simulação, adicionando nas intruções, um número específico em binário de 8 bits, convertendo-o em uma representação de caracteres no padrão ASCII.

Palavras-chave: Sistemas Digitais, Memória RAM, Memória ROM, Assembly, VHDL, CPU, ULA.

Lista de Imagens

Sumário

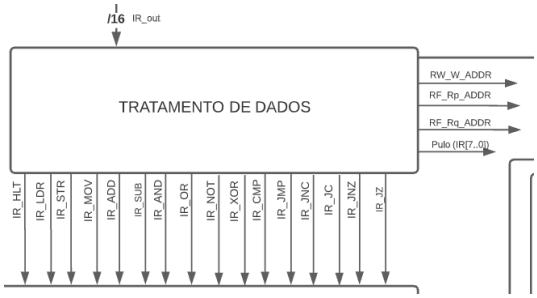
1 IMPLEMENTAÇÃO

Nesta seção serão apresentadas as soluções propostas, implementadas com algumas ressalvas, já que algumas alterações foram necessárias.

1.1 Correções do projeto

Anteriormente, o bloco IR era responsável por fazer um breve destacamento dos bits de entrada, os separando na saída em 4 variáveis. Seriam essas o OP[15:12], ADDR[11:8], B[7:4] e C[3:0]. Além dessas saídas, também tinha um variável de 8 bits que seria responsável pelo enderaçamento da memória de dados. Esta tinha o nome de B+C[7:0], pois se caracterizava como a concatenação das variáveis B e a C. Em nosso entendimento, o IR seria um registrador responsável para armazenar a saída de 16 bits mais recente da memória rom. Por isso, foi descidido retirar este "processamento"deste bloco e adicionado em um novo bloco, chamado de TRATAMENTO DE DADOS. Este pode ser observado na Figura 1 abaixo.

Figura 1 – Bloco Tratamento de Dados.



Fonte: Elaborado pelos autores.

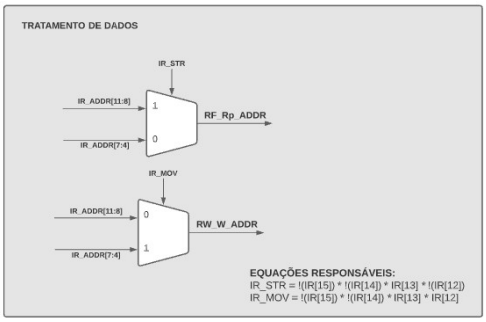
Para elaborar o bloco TRATAMENTO DE DADOS, foram feitas equações com o intuíto de separar quais seriam os OPCODEs que iriam entrar na MDE. Na Figura 2a podemos ver qual foi a lógica inserida no bloco. Sendo também que cada variável representa o estado desejado. Na Figura 2b, podem ser vistas quais serão as equações responsáveis pelo encaminhamento dos endereços dos registradores utilizados no BANCO DE REGISTRADORES.

Também foi retirado o load da memória de intrução, pois foi visto que bastava apenas os registradores terem, já que seriam eles que iriam ler a ROM.O bloco do PC, em sua essência, não foi modificado. Porém o anterior era apenas um registrador que recebia o valor de um somador. O atual se tornou um só com esse somador e o mux. Esta

Figura 2 – TRATAMENTO DE DADOS

(0000) IR_HLT = !(IR[15]) * !(IR[14]) * !(IR[13]) * !(IR[12])
(0001) IR_LDR = !(IR[15]) * !(IR[14]) * !(IR[13]) * IR[12]
(0010) IR_STR = !(IR[15]) * !(IR[14]) * IR[13] * !(IR[12])
(0011) IR_MOV = !(IR[15]) * !(IR[14]) * IR[13] * IR[12]
(0100) IR_ADD = !(IR[15]) * IR[14] * !(IR[13]) * !(IR[12])
(0101) IR_SUB = !(IR[15]) * IR[14] * !(IR[13]) * IR[12]
(0110) IR_AND = !(IR[15]) * IR[14] * IR[13] * !(IR[12])
(0111) IR_OR = !(IR[15]) * IR[14] * IR[13] * IR[12]
(1000) IR_NOT = IR[15] * !(IR[14]) * !(IR[13]) * !(IR[12])
(1001) IR_XOR = IR[15] * !(IR[14]) * !(IR[13]) * IR[12]
(1010) IR_CMP = IR[15] * !(IR[14]) * IR[13] * !(IR[12])
(1011) IR_JMP = IR[15] * !(IR[14]) * IR[13] * IR[12]
(1100) IR_JNC = IR[15] * IR[14] * !(IR[13]) * !(IR[12])
(1101) IR_JC = IR[15] * IR[14] * !(IR[13]) * IR[12]
(1110) IR_JNZ = IR[15] * IR[14] * IR[13] * !(IR[12])
(1111) IR_JZ = IR[15] * IR[14] * IR[13] * IR[12]

(a) Equações

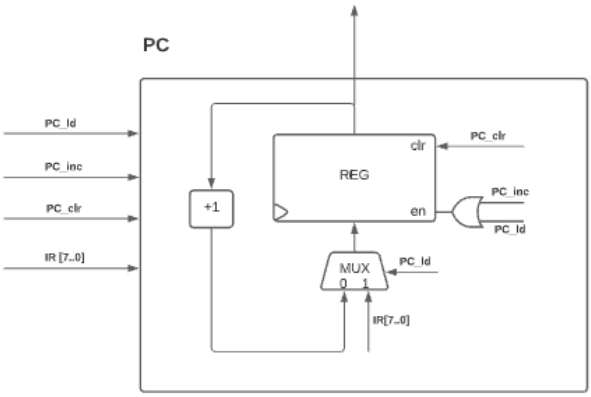


(b) Multiplexador

Fonte: Elaborado pelos autores.

alteração foi feita na visão de tornar o PC como o próprio contador sugerido. Podemos ver na Figura 3.

Figura 3 – Bloco PC do processador.

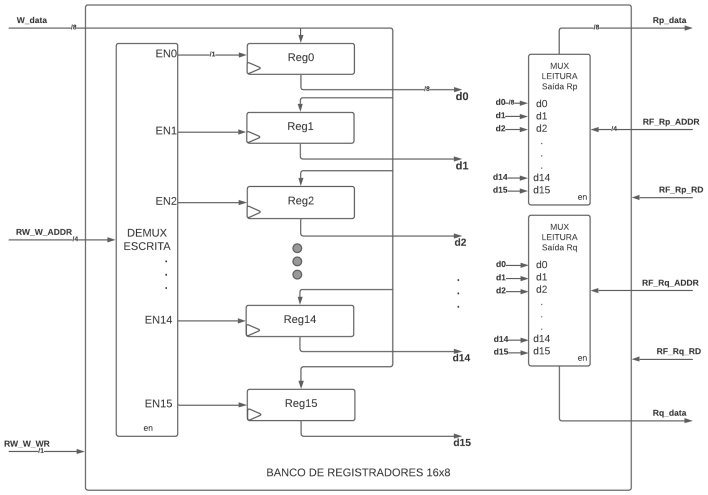


Fonte: Elaborado pelos autores.

Também foi pensado e adicionado como seria um banco de registradores, pois no relatório passado não havia uma clara definição de como deveria ser implementado. Devia a isso esse parágrafo foi adicionado. Na Figura 4, podemos a formação do banco. Percebe-se, que na sua entrada temos um demux responsável pelo endereçamento que indicará quais registradores devem ser utilizados. A entrada dos dados vai diretamento para os registradores e suas saídas irão para dois multiplexadores, sendo esses responsáveis por endereçar os dados na porta das entradas da ULA.

O antigo BLOCO DE CONTROLE recebia do IR, os valores já filtrados com o OPCODE e os endereços que seriam encaminhados ao banco de registradores. Na mudança, agora ele é unicamente a MDE. Então, recebe os valores do bloco TRATAMENTO DE DADOS, que são 16 flags indicadoras de qual opcode está sendo escolhido nas instruções e sua saída correspondem as mesmas da tabela da MDE que se encontra nos anexos. O

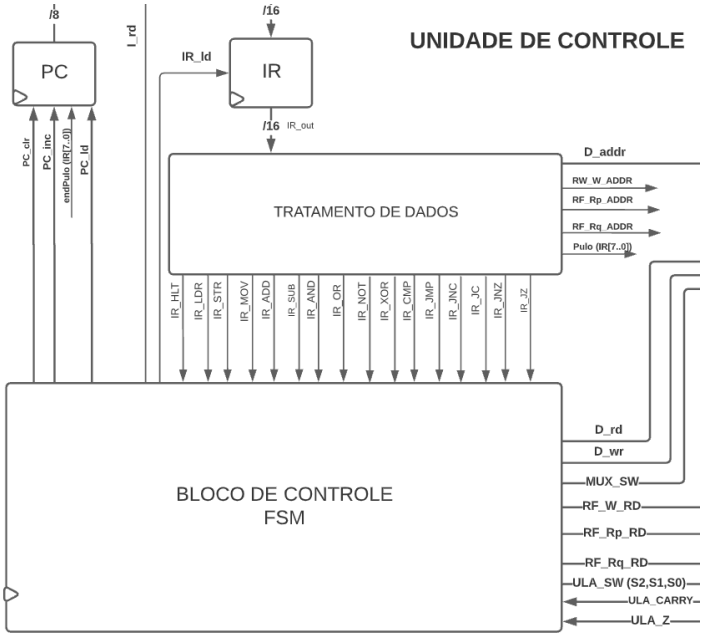
Figura 4 – Banco de Registradores.



Fonte: Elaborado pelos autores.

conjunto que reúne os blocos PC, IR, Tratamento de Dados e Bloco de controle (MDE) forma a unidade de controle como podemos ver na Figura 5.

Figura 5 – Unidade de controle.



Fonte: Elaborado pelos autores.

1.1.1 Máquina de Estado de Baixo Nível

A MDE, antes das mudanças, tinha como entrada apenas 4 flags que se compreendiam como os bits indicadores do OPCODE vindo do IR. Agora, para evitar que a MDE trabalhasse diretamente com dados, então foi feito um tratamento dos dados em um bloco anterior e acrescentamos essas variáveis que indicam qual é o código da instrução

a ser utilizada no momento. Adicionamos uma saída habilitadora para o registrador de escrita do BLOCO DE REGISTRADOR. Também foram inseridas outras 3 variáveis como entrada, sendo elas, as duas saídas da ULA (ULA_Z e ULA_Carry) e a reset que, como o nome sugere, serve para forçar um reset nos registradores, fazendo com que todos estejam devidamente inicializados.

Além disso, foi retirado o estado SALTO, pois tinha a mesma ideia de JMP, por isso, este último ficou no lugar do do estado retirado.Já que anteriormente a MDE tinha 20 estados, agora a atualizada possui 19 estados, sendo esses: INICIO, BUSCA, DEC, HLT, LDR, STR, MOV, ADD, SUB, AND, OR, NOT, XOR, CMP, JMP, JNC, JC, JNZ, JZ.

Quando o processo estiver no estado DEC (decodificar), a depender de qual flag representadora do OPCODE estiver ativada, irá selecionar qual deverá ser o próximo estado.

1.2 MEMÓRIA RAM E ROM

Para a memória RAM foram adicionadas 4 entradas e 1 saída, são elas: a do endereçamento da memória, o habilitador de leitura e de escrita, os dados de entrada e a saída dos dados de 8 bits. O endereçamento da memória virá do bloco TRATAMENTO DE DADOS e os habilitadores de leitura e o de escrita são derivados na própria MDE. A sua entrada de dados que nos permite escrever na memória, são carregados da própria ULA e a saída de 8 bits irá diretamente para o MUX_2x1. Na Figura 6 vemos como foi elaborado.

Na memória ROM (responsável por armazenar as instruções), temos uma entrada e uma saída. A entrada de 8 bits, vinda do PC, consiste em selecionar qual instrução será a escolhida e a sua saída deve ser de 16 bits, pois contém o OPCODE (4 bits), endereço do registrador (4 bits) e o da memória de dados (8 bits). Na Figura 7.

1.3 UNIDADE DE CONTROLE

A unidade de controle foi elaborada como a entidade "mãe", carregando como componentes o PC, o IR, T_DATA e o BLOCKCON. O port map do PC foram passados o reset, o enable, o load para carregar os pulos, o valor do pulo, o clock e a sua saída MEM_ADDR. No IR temos a entrada do reset, o enable, o clock, os valores de 16 bits e sua saída também de 16 bits. Vale ressaltar que o PC_inc do PC e o IR_ld do IR foram considerados como uma só variável: o enable. NO BLOCKCON são passados os valores das equações realizadas no componente do tratamento de dados, e as flags das saídas da ULA.

Figura 6 – Código da memória RAM.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity MEM_DATA is
  port (
    clk : in std_logic ;
    we : in std_logic ;
    addr : in std_logic_vector ( 7 downto 0);
    datai : in std_logic_vector (7 downto 0);
    datao : out std_logic_vector (7 downto 0));
end MEM_DATA;

architecture ckt of MEM_DATA is
  type memoria_ram is array (0 to 255) of std_logic_vector (7 downto 0);
  signal RAM : memoria_ram := (0 => "00000001",
                               1 => "00000010",
                               2 => "01000000",
                               others => "00000000");
  begin
  process (clk) begin
    if rising_edge(clk) then
      if we = '1' then
        RAM(conv_integer(addr))<=datai;
      end if;
        datao <= RAM(conv_integer(addr));
      end if;
      end process ;
end ckt;
```

Fonte: Elaborado pelos autores.

Figura 7 – Código da memória ROM.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity MEM_INST is
    port (
        clk : in std_logic ;
        addr : in std_logic_vector ( 7 downto 0);
        data : out std_logic_vector (15 downto 0));
end MEM_INST;

architecture ckt of MEM_INST is
    type memoria_rom is array (0 to 255) of std_logic_vector (15 downto 0);
    signal ROM : memoria_rom := (0 => X"1100",
                                 1 => X"1701",
                                 2 => X"4217",
                                 others => X"0000");
    begin
    process (clk) begin
        if rising_edge(clk) then
            data <= ROM(conv_integer(addr ));
        end if;
    end process ;
end ckt;
```

Fonte: Elaborado pelos autores.

1.3.1 Bloco PC.

Para o bloco PC (Program Counter), foram utilizadas 4 entradas de dados, sendo duas delas, PC_ld (load) e PC_inc (enable), direcionadas para as entradas de uma porta lógica do tipo OR, que tem em sua saída o sinal en_reg, que servirá de enable do REG dentro do bloco PC, o PC_ld também servirá de seletor para o MUX, onde os dados de entrada à serem selecionados são os 8 bits da entrada IR, a terceira do nosso bloco, ou os 8 bits vindos da saída de um incrementador, que, por sua vez, recebe o sinal da saída do REG.

A quarta e última entrada é o sinal PC_clr, que limpará os dados do REG quando seu valor for igual à 1. Na Figura 8 temos o código de como foi implementado.

Figura 8 – Código do PC.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity PC is
5      port(PC_clr, PC_inc, PC_ld : in STD_LOGIC;
6                  clk : in STD_LOGIC;
7                   IR : in STD_LOGIC_VECTOR(7 downto 0);
8                   IR_out : out STD_LOGIC_VECTOR(7 downto 0));
9  end PC;
10 architecture ckt of PC is
11     component MUX21_8
12     port (A, B : in STD_LOGIC_VECTOR(7 downto 0);
13           S: in STD_LOGIC;
14           O: out STD_LOGIC_VECTOR(7 downto 0));
15     end component;
16     component reg1 is
17     port( clk, D, reset, en : in STD_LOGIC;
18                   S : out STD_LOGIC);
19     end component;
20     component somador8bits is
21     port( A, B: in STD_LOGIC_VECTOR(7 downto 0);
22         O: out STD_LOGIC_VECTOR(7 downto 0);
23         Carry: out STD_LOGIC
24     );
25     end component;
26
27     signal en_reg, clr_reg, outSomador : STD_LOGIC;
28     signal out_soma, out_reg, out_mux : STD_LOGIC_VECTOR(7 downto 0);
29
30     begin
31         en_reg <= PC_inc OR PC_ld;
32
33         M0: MUX21_8 port map (out_soma, IR, PC_ld, out_mux);
34         R0: reg1 port map (clk, out_mux(0), PC_clr, en_reg, out_reg(0));
35         R1: reg1 port map (clk, out_mux(1), PC_clr, en_reg, out_reg(1));
36         R2: reg1 port map (clk, out_mux(2), PC_clr, en_reg, out_reg(2));
37         R3: reg1 port map (clk, out_mux(3), PC_clr, en_reg, out_reg(3));
38         R4: reg1 port map (clk, out_mux(4), PC_clr, en_reg, out_reg(4));
39         R5: reg1 port map (clk, out_mux(5), PC_clr, en_reg, out_reg(5));
40         R6: reg1 port map (clk, out_mux(6), PC_clr, en_reg, out_reg(6));
41         R7: reg1 port map (clk, out_mux(7), PC_clr, en_reg, out_reg(7));
42         S0: somador8bits port map (out_reg, "00000001", out_soma, outSomador);
43 end ckt;
```

Fonte: Elaborado pelos autores.

1.3.2 Bloco IR

O bloco IR é composto por uma entrada *data* com 16 bits, onde ele irá particionar em 4 saídas, sendo elas *OP*, *ADDR*,*B* e *C*, onde cada uma tem um significado. *OP* é composto pelas posições de 15 à 12 do valor de *data* e esses 4 bits são os responsáveis pela escolha do opcode que o Bloco de Controle irá realizar. *ADDR* será composto pelos bits de 11 à 8, e corresponderá ao endereçamento do resultado da operação realizada pelo Bloco. *B* que receberá as posições de 7 à 4, esses 4 bits serão correspondentes ao endereço onde está guardado o valor de reg B. E por fim, na saída *C* irão os bits das posições de 3 à 0, que correspondem ao endereço do valor que está guardado o reg C. A Figura 9 mostra o código referente à implementação do resgitrador de instruções aqui explicado.

Figura 9 – Código do IR.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity IR is
4      port( data : in std_logic_vector(15 downto 0);
5          clk, clr, en_IR : in std_logic;
6          O: out std_logic_vector(15 downto 0));
7  end IR;
8  architecture ckt of IR is
9  component FFD2 is
10     port( clk, D, P, C : in STD_LOGIC;
11                 en : in STD_LOGIC;
12               q, qn : out STD_LOGIC);
13 end component;
14 COMPONENT MUX21_16 is
15     port (A, B : in STD_LOGIC_VECTOR(15 downto 0);
16           S: in STD_LOGIC;
17           O: out STD_LOGIC_VECTOR(15 downto 0));
18 end COMPONENT;
19
20 signal Q, D, qn: std_logic_vector(15 downto 0);
21 begin
22     M0: MUX21_16 port map(Q, data, en_IR, D);
23
24     FFD0: FFD2 PORT MAP (clk, D(0),   '0', clr, en_IR,Q(0), qn(0));
25     FFD1: FFD2 PORT MAP (clk, D(1),   '0', clr, en_IR,Q(1), qn(1));
26     FFD_2: FFD2 PORT MAP (clk, D(2),  '0', clr, en_IR,Q(2), qn(2));
27     FFD3: FFD2 PORT MAP (clk, D(3),   '0', clr, en_IR,Q(3), qn(3));
28     FFD4: FFD2 PORT MAP (clk, D(4),   '0', clr, en_IR,Q(4), qn(4));
29     FFD5: FFD2 PORT MAP (clk, D(5),   '0', clr, en_IR,Q(5), qn(5));
30     FFD6: FFD2 PORT MAP (clk, D(6),   '0', clr, en_IR,Q(6), qn(6));
31     FFD7: FFD2 PORT MAP (clk, D(7),   '0', clr, en_IR,Q(7), qn(7));
32     FFD8: FFD2 PORT MAP (clk, D(8),   '0', clr, en_IR,Q(8), qn(8));
33     FFD9: FFD2 PORT MAP (clk, D(9),   '0', clr, en_IR,Q(9), qn(9));
34     FFD10: FFD2 PORT MAP (clk, D(10), '0', clr, en_IR,Q(10), qn(10));
35     FFD11: FFD2 PORT MAP (clk, D(11), '0', clr, en_IR,Q(11), qn(11));
36     FFD12: FFD2 PORT MAP (clk, D(12), '0', clr, en_IR,Q(12), qn(12));
37     FFD13: FFD2 PORT MAP (clk, D(13), '0', clr, en_IR,Q(13), qn(13));
38     FFD14: FFD2 PORT MAP (clk, D(14), '0', clr, en_IR,Q(14), qn(14));
39     FFD15: FFD2 PORT MAP (clk, D(15), '0', clr, en_IR,Q(15), qn(0));
40
41     O <= Q(15 downto 0);
42
43 end ckt;
```

Fonte: Elaborado pelos autores.

1.3.3 Bloco de Controle

Como foi descrito anteriormente, o bloco de controle é a Máquina de Estados do projeto. Ela vai iniciar em INICIO,onde irá inicializar os registradores do PC com zero, depois passará para o estado BUSCA, que a partir deste será habilitado os registradores IR e o PC, para que dê início a contagem e o armazenamento das instruções. Após isso, será encaminhado para o estado DEC e nele começarão a serem consideradas as 16 flags, de entradas, indicadoras de quais estados sucederão.

O LDR expedita a escrita em um dos registradores, permitindo o dado ser lido da memória RAM (mediante a variável D_rd), além de permitir a escrita no banco por meio da variável RF_W_ wr e a habilitação se dá através do mux através da MUX_SW. No STR, a escrita na memória (variável D_wr) permite que os dados (advindos do registrador Rp) sejam inseridos. No MOV e o NOT, a escrita (RF_W_rd) será habilitada deixando o dado adentrar no registrador Rq. O ADD, o SUB, o AND, o OR, o XOR e o CMP, dispõem as leituras dos registradores Rp e Rq para que seja escrito no W. O JMP colocará o PC_ld em 1, permitindo a inserção de um valor no PC. Já para o JNC, JC, JNZ e JZ, como já visto antes, a depender das duas saídas da ULA, irá para o JMP ou irá para BUSCA, reiniciando todo o processo.

1.3.4 Tratamento de Dados

Como citado anteriormente, foi necessária a criação de um bloco para realizar o tratamento de dados vindos da memória ROM. Estes dados são instruções e possuem 16 bits, onde os 4 mais significativos é o código OPCODE que define qual função deve ser realizada pela CPU naquele momento. Para que a MDE lidasse apenas com variáveis booleanas e enviasse apenas sinais de controle, o bloco tratamento de dados recebe os 16 bits, e envia todos os endereaçamentos necessários para o bloco operacional e também envia 16 sinais de controle que na MDE indicam qual estado atual o processo se encontra e qual estado deve ser o próximo. Na figura 10 se encontra o código que realiza o funcionamento do bloco aqui explicado.

No código citado, são realizados dois PORT MAP para implementar a seleção do conjunto de bits - vindo da instrução atual IR - nas funções MOV e STR para as variáveis de endereço W_ADDR e RP_ADDR. Quando o estado atual é o MOV, o endereço correto a ser passado para a variável W_ADDR é a sequência de bits IR[7...4] e já no caso do estado STR a sequência a ser passada para a variável RP_ADDR é a sequência IR[11...8] e esta é a seleção feita pelo multiplexador adicionado como componente na entidade T_DATA. Além disto, são implementadas 16 equações booelanas com os bits IR[15...12] para que as variáveis, como IR_HLT, receba o valor 1 quando o OPCODE for referente à função desejada, como no exemplo seria a função HLT. Com isso, todo o tratamento de dado e endereçamento necessário é realizado neste bloco, deixando que a MDE lide

Figura 10 – Bloco de Tratmento de Dados.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  ENTITY T_DATA is
5      PORT(IR: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
6      PULO, D_ADDR: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
7      W_ADDR, RP_ADDR, RQ_ADDR: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
8      IR_HLT, IR_LDR, IR_STR, IR_MOV, IR_ADD, IR_SUB, IR_AND, IR_OR, IR_NOT,
9      IR_XOR, IR_CMP, IR_JMP, IR_JNC, IR_JC, IR_JNZ, IR_JZ: OUT STD_LOGIC);
10 END T_DATA;
11 ARCHITECTURE CKT OF T_DATA IS
12 COMPONENT MUX21_4 IS
13     port(A, B : in std_logic_vector(3 downto 0);
14         S : in std_logic;
15         O: out std_logic_vector(3 downto 0));
16 END COMPONENT;
17 SIGNAL STR, MOV: STD_LOGIC;
18 BEGIN
19 MUX0: MUX21_4 PORT MAP(IR(11 DOWNTO 8), IR(7 DOWNTO 4), MOV, W_ADDR);
20 MUX1: MUX21_4 PORT MAP(IR(7 DOWNTO 4), IR(11 DOWNTO 8), STR, RP_ADDR);
21
22 IR_HLT  <= (NOT IR(15) AND NOT IR(14) AND NOT IR(13) AND NOT IR(12));
23 IR_LDR  <= (NOT IR(15) AND NOT IR(14) AND NOT IR(13) AND IR(12));
24 STR     <= (NOT IR(15) AND NOT IR(14) AND IR(13) AND NOT IR(12));
25 MOV     <= (NOT IR(15) AND NOT IR(14) AND IR(13) AND IR(12));
26 IR_ADD  <= (NOT IR(15) AND IR(14) AND NOT IR(13) AND NOT IR(12));
27 IR_SUB  <= (NOT IR(15) AND IR(14) AND NOT IR(13) AND IR(12));
28 IR_AND  <= (NOT IR(15) AND IR(14) AND IR(13) AND NOT IR(12));
29 IR_OR   <= (NOT IR(15) AND IR(14) AND IR(13) AND IR(12));
30 IR_NOT  <= (IR(15) AND NOT IR(14) AND NOT IR(13) AND NOT IR(12));
31 IR_XOR  <= (IR(15) AND NOT IR(14) AND NOT IR(13) AND IR(12));
32 IR_CMP <= (IR(15) AND NOT IR(14) AND IR(13) AND NOT IR(12));
33 IR_JMP  <= (IR(15) AND NOT IR(14) AND IR(13) AND IR(12));
34 IR_JNC  <= (IR(15) AND IR(14) AND NOT IR(13) AND NOT IR(12));
35 IR_JC   <= (IR(15) AND IR(14) AND NOT IR(13) AND IR(12));
36 IR_JNZ  <= (IR(15) AND IR(14) AND IR(13) AND NOT IR(12));
37 IR_JZ   <= (IR(15) AND IR(14) AND IR(13) AND IR(12));
38 IR_STR <= STR;
39 IR_MOV <= MOV;
40 RQ_ADDR <= IR(3 DOWNTO 0);
41 PULO <= IR(7 downto 0);
42 D_ADDR <= IR(7 downto 0);
43
44 END CKT;
```

Fonte: Elaborado pelos autores.

apenas com sinais de controle.

1.4 BLOCO OPERACIONAL

1.4.1 ULA

A ULA consiste na elaboração de diversos blocos combinacionais contidos em um. No caso, o somador foi elaborado com um meio somador, depois um somador de 1 bit, posteriormente chamado 8 vezes em outra entidade. Na subtração foi feita o complemento de 2, para isso foi utilizado as inversões da entrada B. O AND, o OR, XOR e o CMP foram utilizados apenas em uma entidade, cada um. Logo após foram chamados na entidade

principal que é a ULA. O MOV foi encaminho diretamente para a saída sem alterações de dados. Por fim, foi adicionado um MUX que vai permitir sair da ULA o resultado da operação selecionada de acordo com a chave seletora. Essa mesma chave vai indicar qual deverá ser o carry da operação. Já a saída indicadora que o resultado da operação é zero, vai ser também uma saída do mux.

1.4.2 Banco de Registradores

O Banco de registradores recebe um valor de 8 bits como entrada, sendo que como ele irá tratar esse dado irá ser processado, vai depender de qual dos outros sinais que ele recebe estará ativo.

Se o sinal for o de RF_W_RD, o bloco de controle irá ler os 4 bits que chegam à ela pelo RF_W_ADDR, e endereçar o valor da saída do MUX para aquele espaço da memória.

Agora, se os sinais que o bloco receber for os de RF_Rp_RD e/ou RF_Rq_RD (dependendo da operação), o bloco de controle irá ler os 4 bits das saídas de RF_Rp_ADDR e/ou de RF_Rq_ADDR, pegar os valores dos registradores nesses respectivos endereços e enviá-los para as saídas Rp_data e/ou Rq_data.

1.4.3 MUX 2x1

Este multiplexador foi elaborado com 3 entradas e uma saída. Duas das entradas são definidas como 8 bits e a outra é a chave seletora. A sua saída também é de 8 bits. Na Figura 11 podemos ver como foi elaborado o vhdl desse bloco.

Figura 11 – Código do Multiplexador 2x1.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity MUX21_8 is
5       port (A, B : in STD_LOGIC_VECTOR(7 downto 0);
6             S: in STD_LOGIC;
7             O: out STD_LOGIC_VECTOR(7 downto 0) );
8   end MUX21_8;
9
10  architecture ckt of MUX21_8 is
11
12  begin
13
14      O(0) <= (B(0) and S) or (A(0) and (not S));
15      O(1) <= (B(1) and S) or (A(1) and (not S));
16      O(2) <= (B(2) and S) or (A(2) and (not S));
17      O(3) <= (B(3) and S) or (A(3) and (not S));
18      O(4) <= (B(4) and S) or (A(4) and (not S));
19      O(5) <= (B(5) and S) or (A(5) and (not S));
20      O(6) <= (B(6) and S) or (A(6) and (not S));
21      O(7) <= (B(7) and S) or (A(7) and (not S));
22
23  end ckt;
24
```

Fonte: Elaborado pelos autores.

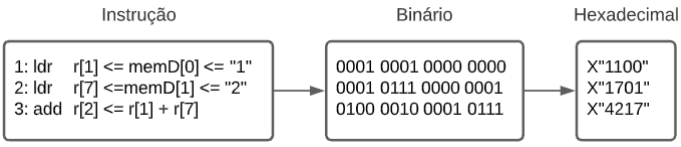
2 Resultados

Os blocos e simulações dos códigos foram rodados todos em VHDL (VHSIC Hardware Description Language) no software ModelSim.

2.1 Simulação simples

Para testar o funcionamento correto da CPU, fizemos um algoritmo simples em Assembly que utiliza as funções de carregamento de dados da memória com a operação de adição da ULA. Na Figura 12 é possível ver as três instruções pensadas, a primeira instrução carrega no registrador de endereço ”0001” o dado que está armazenado no endereço 00000000 da memória de dados(RAM), que no nosso caso possui o valor ”1” armazenado, a segunda instrução faz o mesmo processo, porém colocando no registrador de endereço ”0111” o valor que está armazenado no endereço ”00000001”, que possui o valor de ”2”, na terceira e última instrução a CPU irá armazenar no registrador de endereço ”0010” o resultado da adição entre os valores armazenados nos registradores cujo endereço é ”0001” e ”0111”, respectivamente.

Figura 12 – Instruções de carregamento e adição



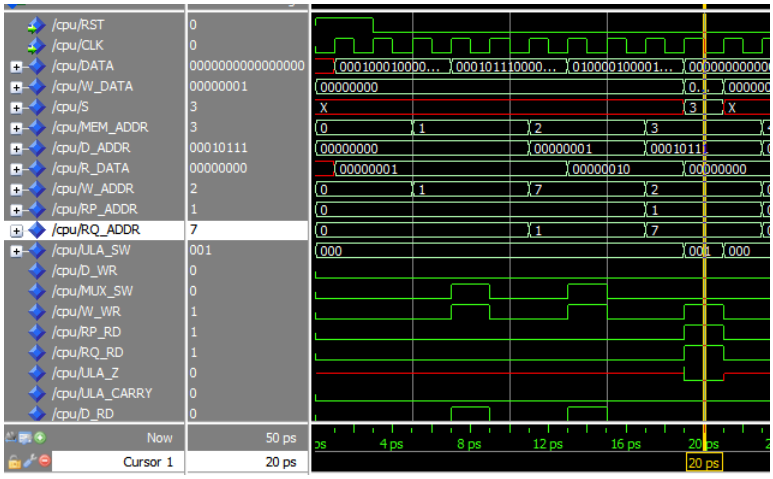
Fonte: Elaborado pelos autores.

A simulação em ModelSim do algoritmo explicado anteriormente pode ser vista na Figura 13.

2.2 Conversão de numéros binários para número decimal (Padrão ASCII para cada dígito)

Testada a funcionalidade da CPU com a simulação básica explicada no tópico anterior, o próximo passo é realizar a implementação de um programa que realize a conversão de um número bináio de 8 bits para um decimal em padrão ASCII usando de assembly na inserção das instruções na memória ROM. Na Figura 15 temos o resultado da simulação da instrução descrita abaixo.

Figura 13 – Simulação Simples em ModelSim



Fonte: Elaborado pelos autores.

Para a operação de converter um número em binário de 8 bits para um decimal em padrão ASCII, primeiro foi necessário dar o load dos valores necessários nos registradores, no exemplo do grupo, o número escolhido foi 125, então, nesse caso, os valores nos 5 registradores foram: reg0 = 100 (0110 0100), reg1 = 10 (0000 1010), reg2 = 48 (0011 0000), reg3 = 125 (0111 1101) e reg4 = 1 (0000 0001).

Após dado o load nos registradores, e considerando que os outros já se iniciam com o valor 0, e não um lixo, a primeira operação pode ser realizada, que é a de uma subtração entre 125 e 100, salvando o resultado no reg5, e no reg6, somando 0 + 1.

Como o resultado no reg5 (sexto) é 25, e esse valor é menor que 100, então já temos o primeiro dígito para transformar para decimal em ASCII, o valor de 1 no reg6 (sétimo).

Para as dezenas, vamos então usar o reg7 para contar as dezenas, mas primeiro, realizar 25 - 10 e guardar no reg5, após isso, no reg7 fazer a soma de 0 + 1, como sobrou 15 e esse valor é maior que 10, então pode ser realizado novamente a mesma operação, então reg5 salva 15 - 10, e reg7 salva 1 + 1, assim, finalmente, com o reg5 tendo o valor de 5, que foi o que sobrou para a grandeza das unidades, e o reg7 com o valor de 2 com a grandeza das dezenas.

E finalmente para o último passo, consultando a tabela ASCII, vemos que o número 1 é representado por 49 na tabela, sendo então somente necessário somar 48.

Então para finalizar o último passo, adiciona-se o reg2, que recebeu o valor de 48 lá no início, aos registradores que guardaram cada dígito do número escolhido.

O reg6 guardou o 1, mais 48 = sairá 49, que vale 1 na tabela ASCII.

O reg7 guardou o 2, mais 48 = sairá 50, que vale 2 na tabela ASCII.

E por fim, o reg5 que guardou o 5, mais 48 = sairá 53, que vale 5 na tabela ASCII.

Todo esse algoritmo pode ser observado em linhas de código usando do assembly para inserir as instruções em hexadecimal na figura 14 abaixo. E, para comprovar a funcionalidade tanto da CPU quanto do programa desenvolvido para conversão, na figura 15 vemos a simulação e o resultado da conversão do número 125 para decimal padrão ASCII se localiza da seguinte forma:

- Para o dígito 1, temos sua conversão armazenada no banco de registradores e aparece no tempo de 50ps como o valor 49;
- Para o dígito 2, temos sua conversão armazenada no banco de registradores e aparece no tempo de 80ps como o valor 50;
- Para o dígito 5, temos sua conversão armazenada no banco de registradores e aparece no tempo de 85ps como o valor 53;

Com isso, observamos a efetividade e funcionalidade da CPU e do programa de conversão de binário para decimal padrão ASCII.

Figura 14 – Conversão em linhas de código Hexadecimal

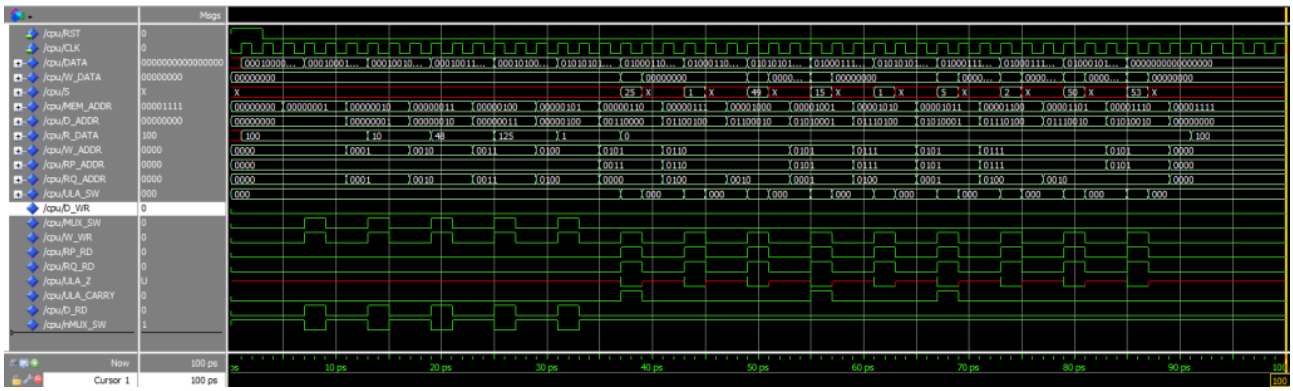
```
--mem0 => 0110 0100 = 100 em decimal
--mem1 => 0000 1010 = 10 em decimal
--mem2 => 0011 0000 = 48 em decimal
--mem3 => 0111 1101 = 125 em decimal
--mem4 => 0000 0001 = 1 em decimal

"X1000"-- => load no reg0 do valor da memoria 0 (100)
"X1101"-- => load no reg1 do valor da memoria 1 (010)
"X1202"-- => load no reg2 do valor da memoria 2 (048)
"X1303"-- => load no reg3 do valor da memoria 3 (125)
"X1404"-- => load no reg4 do valor da memoria 4 (001)
--OBS: Considera-se que os regs jsn iniciam com 0.

"X5530"-- => guardar no reg5 => 125 - 100 = 25
"X4664"-- => guardar no reg6 => reg6(0) + reg4(1) => 0 + 1 => 1 -> centena
"X4662"-- => guardar no reg6 => reg6(1) + reg2(48) => 1 + 48 => 49 -> 1 na tabela ASCII
"X5551"-- => guardar no reg5 => reg5(25) - reg1(10) = 15
"X4774"-- => guardar no reg7 => reg7(0) + reg4(1) => 0 + 1 => 1 -> dezena
"X5551"-- => guardar no reg5 => reg5(15) - reg1(10) => 5
"X4774"-- => guardar no reg7 => reg7(1) + reg4(1) => 1 + 1 => 2 -> dezena
"X4772"-- => guardar no reg7 => reg7(2) + reg2(48) => 2 + 48 => 50 -> 2 na tabela ASCII
"X4552"-- => guardar no reg5 => reg5(5) + reg2(48) => 5 + 48 => 53 -> 5 na tabela ASCII
```

Fonte: Elaborado pelos autores.

Figura 15 – Simulação em ModelSim: Conversão de binário para decimal



Fonte: Elaborado pelos autores.

3 CONCLUSÃO

Neste trabalho foi implementada a construção de uma Unidade Central de Processamento a partir da construção de componentes menores em VHDL, como o Bloco Operacional que contém a ULA e um banco de registradores. A CPU deveria ser capaz de executar um conjunto de instruções de 16 bits, pré-definidas na memória ROM. De acordo com os resultados obtidos na simulação básica - realizada para testar o funcionamento básico da CPU - é possível afirmar que o projeto foi bem sucessido. Os resultados obtidos representam o funcionamento pleno e correto da máquina construída.

Ainda para comprovar o desempenho correto do circuito implementado em VHDL, foi realizado um outro teste mais avançado. Foi desenvolvido um código, em assembly, que a realiza a conversão de um número binário de 8 bits em um número decinal padrão ASCII. De acordo com os resultados, o funcionamento é o esperado e assim concluímos que o projeto foi executado com sucesso.

Referências

VAHID, F. *Sistemas Digitais: Projeto, Otimização e HDLs*. [S.l.]: Artmed Bookman, 2008.

ANEXO A – Relato semanal

Líder: Anny Beatriz Pinheiro Fernandes

A.1 Equipe

Tabela 1 – Identificação da equipe

Função no grupo	**Nome completo do aluno**
Redator	Wesley Brito da Silva
Debatedor	Arthur Felipe Rodrigues Costa
Videomaker	Isaac de Lyra Junior
Auxiliar	N/A

Fonte: Produzido pelos autores.

A.2 Defina o problema

O presente trabalho propõe a implementação do projeto desenvolvido para a construção, em VHDL e assembly, de uma unidade central de processemaneto (CPU) que utiliza de uma memória de dados (RAM) e de uma memória de instruções (ROM), ambas alimentadas pelos projetistas. A ROM possui 256 instruções de 16 bits em assembly e a RAM possui 256 endereços de 8 bits que podem assumir qualquer valor desejado.

O projeto possui, além dos componentes já citados, uma unidade de controle e um bloco operacional com os quais a CPU deve ser capaz de realizar uma sequência de operações. E, para que o funcionamento do componente seja comprovado, é requerido o desenvolvimento de dois programas em assembly. Um que realize a conversão de um número binário de 8 bits em decinal de acordo com a tabela ASCII para cada dígitp e outro que realize a multiplicação de dois números de 8 bits, resultando em 16 bits.

Logo, o projeto consiste em implementar - em VHDL - uma CPU capaz de executar de forma autônoma um conjunto de intruções pré-definidas estabelecidas pela memória de instruções.

A.3 Registro de *brainstorming*

No primeiro dia, terça pós-aula, fizemos um breve encontro no qual foram decididos os cargos de cada membro e definidos tópicos/atividades a serem realizadas até a próxima reunião.Tais atividades foram voltadas ao estudo do relatório de projeto a ser implementado.

No mesmo encontro definimos um horário padrão de reunião diário que poderia ser alterado de acordo com a necessidade do grupo e que na próxima reunião seriam definidos os componentes que seriam implementados durante a semana.

No segundo dia, quarta, realizamos o levantamento de quais componentes iriam ser necessários a implementação em VHDL e também realizamos alterações necessárias no projeto atual para que o desenvolvimento do trabalho fosse o melhor possível. Ao fim da reunião, definimos um conjunto de componentes para cada membro implementar até à quinta de 20h. Ou seja, até a próxima reunião.

Na quinta, tiramos dúvidas dos componentes que foram designados a cada membro. Finalizamos aqueles que não estavam prontos, em grupo, realizamos outras alterações/correções no projeto novamente e definimos o que seria feito na reunião da sexta.

Na sexta realizamos a construção dos componentes maiores, como a construção do bloco operacional e unidade de controle, os quais utilizam dos componentes menores (exemplo: banco de registradores). Ainda em reunião definimos que no próximo encontro seria realizada a adaptação/correção de problemas nos códigos até então desenvolvidos, se necessário e iríamos realizar a construção do maior componente, a CPU.

No sábado, o trabalho foi iniciado às 13h (pela líder) para construção de imagens e correção das tabelas que iriam para o relatório e às 15h todo o grupo se reuniu para finalizar o trabalho. Diante dos problemas encontrados no código, fomos resolvendo em conjunto e também fomos elecando os componentes já finalizados e totalmente testados. O próximo encontro seguiu o mesmo ritmo de correção e finalização dos maiores componentes adicionando o estudo sobre a implementação das instruções em assembly na memória ROM.

Na segunda, último encontro do grupo, foram realizadas as finalizações no código, as simulações necessárias, escrita dos relatórios e produção do vídeo.

A.4 Pontos-chaves

Os pontos-chaves para a implementação deste projeto foi o entendimento de cada componente como um bloco a ser utilizado, entender a posição do bloco de controle diante dos tratamentos de dados vindos da memória de instrução, entender a comunicação existente entre os blocos e as memórias e entender a aplicação de instruções em assembly para coordenar o funcionamento da CPU como um todo.

A.5 Questões de pesquisa

- Memória ROM;

- Memória RAM;
- Banco de registradores;
- Assembly e suas aplicações na CPU;
- Tratamento de dados pela Unidade de Controle.

A.6 Planejamento da pesquisa

Inicialmente, já no primeiro dia de reunião foi requerido o estudo do relatório do projeto, estudo do livro VAHID e também da pesquisa a respeito da aplicação de assembly para o desenvolvimento do projeto. Nos dias subsequentes, toda pesquisa desenvolvida foi de acordo com a necessidade de cada membro diante do que era necessário para a realização das atividades requeridas para o trabalho. Ao chegar na etapa de implementação do assembly, todos os membros realizaram uma discursão sobre o tema e compartilhamos conhecimento e ficamos todos ao mesmo nível neste tópico.

ANEXO B – Tabela de Transição de Estados