UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
CONCEPÇÃO ESTRUTURADA DE CIRCUITOS INTEGRADOS

PROJETO MIPS – ENTREGA 7

THIAGO ALVES DE ARAUJO

2016019787

Sumário

1 Tabela de figuras

2 Unidade de Controle

Agora vamos desenvolver a unidade de controle do MIPS 32. Abaixo podemos ver uma ilustração do diagrama dos estados da maquina de estados finita. Vamos implementar as operações LW, SW, ADD, SUB, AND, OR, NOR, XOR, SLT, SUBI, ADDI, ORI, XORI, NORI, SLTI, BEQ, BNE, J

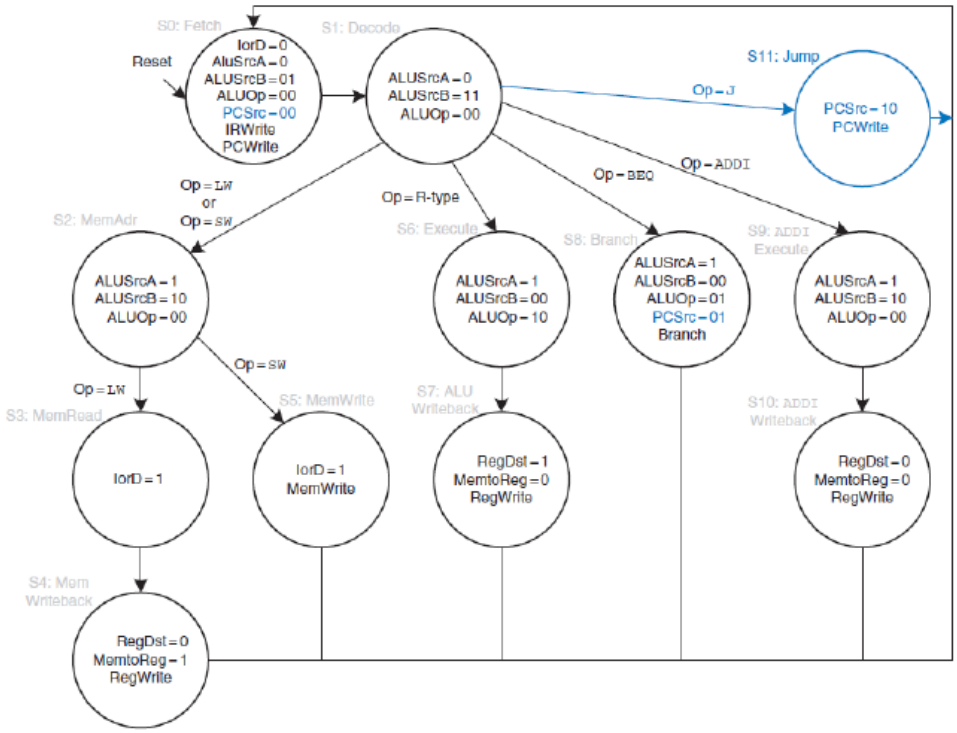

Figura 1 Unidade de controle - Diagrama de estados

2.1 Goldem model

Abaixo segue o trecho do código utilizado para gerar os *goldem vectors.* O código foi desenvolvido em C e simula as sequencias de estados para cada operação implementada para gerar os *goldem vectors*

```
int main(){
    ofstream file("control_unit.txt");

    string rst = "0";
    vector<string> clk = {"0","1"};
    vector<string> opcode = {"100011","101011","000000","000000","000000","000000","000000","000100","001000","000010","000000","000000","001101",
    vector<string> funct  = {"100000","100000","100000","100010","100100","100101","101010","100000","100000","100000","100111","100110","100000",

    //                          LW    ,    SW ,    ADD  ,   SUB  ,  AND   ,   OR   ,  SLT  ,    BEQ  ,   ADDI ,   J  ,    NOR  ,   XOR,   ORI

    string MemtoReg = "0",RegDst = "0",IorD = "0",PCSrc = "00",ALUSrcA = "0",IRWrite = "1",MemWrite = "0",PCWrite = "1",Branch = "0",RegWrite = "(

    int i = 0;

    int j = 0;
    int k = 0;
    int l = 0;
    while(i != 16){
        if(k > 1)
            k = 0;

        if(l < 2){
            rst = "1";
            j = 0;
        }
        else
```

Figura 2 Unidade de controle - Goldem Model

2.2 Goldem vector

A partir do *goldem model* criado anteriormente, geramos o seguinte vetor de testes. Vamos utilizar 126 vetores de teste para verificar o funcionamento de todas as instruções.

```
0_1_100011_100000_010_0_01_0_0_1_0_00_1_0_0_0_0
1_1_100011_100000_010_0_01_0_0_1_0_00_1_0_0_0_0
0_0_100011_100000_010_0_01_0_0_1_0_00_1_0_0_0_0
1_0_100011_100000_010_0_11_0_0_0_0_00_0_0_0_0_0
0_0_100011_100000_010_0_11_0_0_0_0_00_0_0_0_0_0
1_0_100011_100000_010_1_10_0_0_0_0_00_0_0_0_0_0
0_0_100011_100000_010_1_10_0_0_0_0_00_0_0_0_0_0
1_0_100011_100000_010_0_00_0_1_0_0_00_0_0_0_0_0
0_0_100011_100000_010_0_00_0_1_0_0_00_0_0_0_0_0
1_0_100011_100000_010_0_00_0_0_0_1_00_0_0_1_0_0
0_0_100011_100000_010_0_00_0_0_0_1_00_0_0_1_0_0
1_0_100011_100000_010_0_01_0_0_1_0_00_1_0_0_0_0
0_0_101011_100000_010_0_01_0_0_1_0_00_1_0_0_0_0
1_0_101011_100000_010_0_11_0_0_0_0_00_0_0_0_0_0
0_0_101011_100000_010_0_11_0_0_0_0_00_0_0_0_0_0
1_0_101011_100000_010_1_10_0_0_0_0_00_0_0_0_0_0
0_0_101011_100000_010_1_10_0_0_0_0_00_0_0_0_0_0
1_0_101011_100000_010_0_00_0_1_0_0_00_0_0_0_1_0
0_0_101011_100000_010_0_00_0_1_0_0_00_0_0_0_1_0
1_0_101011_100000_010_0_01_0_0_1_0_00_1_0_0_0_0
0_0_000000_100000_010_0_01_0_0_1_0_00_1_0_0_0_0
1_0_000000_100000_010_0_11_0_0_0_0_00_0_0_0_0_0
0_0_000000_100000_010_0_11_0_0_0_0_00_0_0_0_0_0
1_0_000000_100000_010_1_00_0_0_0_0_00_0_0_0_0_0
0_0_000000_100000_010_1_00_0_0_0_0_00_0_0_0_0_0
1_0_000000_100000_010_0_00_0_0_0_0_00_0_1_1_0_0
0_0_000000_100000_010_0_00_0_0_0_0_00_0_1_1_0_0
```

Figura 3 Unidade de controle - Goldem vectors

2.3 Testbench

Abaixo podemos ver o código em SystemVerilog responsável por testar a unidade de controle. O testbench segue o padrão dos códigos desenvolvidos anteriormente.

```
1  `timescale 1ns/100ps
2
3  module controle_tb;
4
5  int counter, errors, aux_error;
6  //entradas
7  logic clk,rst,clk_tb,rst_tb;
8  logic [5:0]opcode, funct;
9  //saidas
10 logic MemtoReg, RegDst, IorD, ALUSrcA, IRWrite, MemWrite, PCWrite, Branch, RegWrite, BranchNE;
11 logic [1:0]PCSrc, ALUSrcB;
12 logic [2:0]ALUControl;
13 //saidas esperadas
14 logic MemtoReg_esperado, RegDst_esperado, IorD_esperado, ALUSrcA_esperado, IRWrite_esperado, MemWrite_esperado, PCWrite_esperado, Branch_esperado, RegWrite_es
15 logic [1:0]PCSrc_esperado, ALUSrcB_esperado;
16 logic [2:0]ALUControl_esperado;
17
18 logic [30:0]vectors[126];
19
20 controle dut(.clk(clk),.rst(rst),.opcode(opcode),.funct(funct),.MemtoReg(MemtoReg),.RegDst(RegDst),.IorD(IorD),.ALUSrcA(ALUSrcA),.IRWrite(IRWrite),
21 .MemWrite(MemWrite),.PCWrite(PCWrite),.Branch(Branch),.RegWrite(RegWrite),.BranchNE(BranchNE),.PCSrc(PCSrc),.ALUSrcB(ALUSrcB),.ALUControl(ALUControl));
22 initial begin
23     $display("Iniciando Testbench");
24     $display("|clk|rst| opcode | funct  |ALUControl|ALUSrcA|ALUSrcB|Branch|IorD|IRWrite|MemtoReg|PCSrc|PCWrite|RegDst|RegWrite|MemWrite|BranchNE|");
25     $readmemb("controle_tv.tv",vectors);
26     counter=0; errors=0;
27     rst_tb = 1; #8; rst_tb = 0;
28 end
29
30 always
31     begin
32     clk_tb=1; #50;
33    clk_tb=0; #10;
34     end
```

```
always @(posedge clk_tb)
   begin
      {clk,rst,opcode,funct,ALUControl_esperado,ALUSrcA_esperado,ALUSrcB_esperado,Branch_esperado,IorD_esperado,IRWrite_esperado,MemtoReg_esperado,PCSrc_espera
   end

always @(negedge clk_tb)
      if(~rst_tb)
      begin
         aux_error = errors;
         assert (MemtoReg === MemtoReg_esperado)
         else
         begin
               $display("Error MemtoReg: %b, expected %b", MemtoReg, MemtoReg_esperado);

         errors = errors + 1;
         end
         assert (RegDst === RegDst_esperado)
         else
         begin
               $display("Error RegDst: %b, expected %b", RegDst, RegDst_esperado);

         errors = errors + 1;
         end
         assert (IorD === IorD_esperado)
         else
         begin
               $display("Error IorD: %b, expected %b", IorD, IorD_esperado);

         errors = errors + 1;
         end
         assert (PCSrc === PCSrc_esperado)
         else
         begin
               $display("Error PCSrc: %b, expected %b", PCSrc, PCSrc_esperado);

```

Figura 4 Unidade de controle - Testbench

2.4 Modelo duv

Para o modulo da unidade de controle temos as entradas de clock, reset, o Opcode e o funct. As saídas são todos os sinais de controle (MemtoReg, RegDst, IorD, ALUSrcA, IRWrite, MemWrite, PCWrite, Branch, RegWrite, BranchNE, PCSrc, ALUSrcB, ALUControl) que a unidade de controle envia para o caminho de dados.

```
module controle(
                     input clk, rst,
                     input logic [5:0]opcode, funct,
                     output logic MemtoReg, RegDst, IorD, ALUSrcA, IRWrite, MemWrite, PCWrite, Branch, RegWrite, BranchNE,
                     output logic [1:0]PCSrc, ALUSrcB,
                     output logic [2:0]ALUControl);

 logic [2:0]ALUOp;

fsm fsm(.clk(clk),.rst(rst),.opcode(opcode),.MemtoReg(MemtoReg),.RegDst(RegDst),.IorD(IorD),.ALUSrcA(ALUSrcA),.IRWrite(IRWrite),
.MemWrite(MemWrite),.PCWrite(PCWrite),.Branch(Branch),.RegWrite(RegWrite),.BranchNE(BranchNE),.PCSrc(PCSrc),.ALUSrcB(ALUSrcB),.ALUOp(ALUOp));

 alu_decoder alu_decoder(.funct(funct),.ALUOp(ALUOp),.ALUControl(ALUControl));

 endmodule
```

Figura 5 Unidade de controle - Controle.sv

Dentro da modulo controle.sv utilizamos uma maquina de estados finita (fsm) que representa todos os estados mostrados anteriormente. Abaixo podemos ver um trecho do código do fsm

```
module fsm (input clk, rst,
            input logic [5:0]opcode,
            output logic MemtoReg, RegDst, IorD, ALUSrcA, IRWrite, MemWrite, PCWrite, Branch, RegWrite, BranchNE,
            output logic [1:0]PCSrc, ALUSrcB,
            output logic [2:0]ALUOp);

   parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4, S5 = 5, S6 = 6, S7 = 7, S8 = 8, S9 = 9, S10 = 10, S11 = 11, S12 = 12, S13 = 13, S14 = 14, S15 = 15;
   reg[3:0] state;

   always@(state) begin
      case(state)

         S0: begin  // Fetch
            IorD    = 1'b0;
            ALUSrcA  = 1'b0;
            ALUSrcB  = 2'b01;
            ALUOp    = 3'b000;
            PCSrc    = 2'b00;
            IRWrite  = 1'b1;
            PCWrite  = 1'b1;

            BranchNE = 1'b0;
            MemtoReg = 1'b0;
            RegDst   = 1'b0;
            MemWrite = 1'b0;
            Branch   = 1'b0;
            RegWrite = 1'b0;
         end
```

Figura 6 Unidade de controle - Fsm.sv

Dentro do fsm também utilizamos um decodificador para a ula que servirá para que ela funcione perfeitamente nas operações do tipo R

```
1  module alu_decoder( input logic [5:0]funct, input logic [2:0]ALUOp, output logic [2:0]ALUControl);
2
3
4  always_comb begin
5     case(ALUOp)
6        3'b000: ALUControl = 3'b010; //ADDI, SW, LW
7        3'b001: ALUControl = 3'b110;//BEQ
8        3'b010: begin
9           case(funct)
10             6'b100000: ALUControl = 3'b010;//ADD
11             6'b100010: ALUControl = 3'b110;//SUB
12             6'b100100: ALUControl = 3'b000;//AND
13             6'b100101: ALUControl = 3'b001;//OR
14             6'b101010: ALUControl = 3'b111;//SLT
15             6'b100110: ALUControl = 3'b011;//XOR
16             6'b100111: ALUControl = 3'b100;//NOR
17             default:   ALUControl = 3'b000;//default(ADD)
18          endcase
19       end
20
21       3'b011: ALUControl = 3'b111;//SLTI
22       3'b100: ALUControl = 3'b110;//BNE
23       //3'b101: ALUControl = 3'b001;//ORI
24       3'b110: ALUControl = 3'b001;//ORI
25       3'b111: ALUControl = 3'b011;//XORI
26       default: ALUControl = 3'b000;
27    endcase
28 end
29
30 endmodule
```

Figura 7 Unidade de controle - Alu Decoder.sv

2.5 Simulação RTL

Com o modulo pronto, podemos iniciar a simulação RTL. Como podemos ver na imagem abaixo, o modulo está se comportando como o esperado e todos os sinais de controle estão corretos.

```
# | 1 | 0 | 000000 | 100110 |    011    |   1   |   00   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 000000 | 100110 |    011    |   1   |   00   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 000000 | 100110 |    010    |   0   |   00   |   0   | 0 | 0 |   0   | 00 |   0   | 1 |    1    |   0   |   0   | OK
# | 0 | 0 | 000000 | 100110 |    010    |   0   |   00   |   0   | 0 | 0 |   0   | 00 |   0   | 1 |    1    |   0   |   0   | OK
# | 1 | 0 | 000000 | 100110 |    010    |   0   |   01   |   0   | 0 | 1 |   0   | 00 |   1   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 001101 | 100000 |    010    |   0   |   01   |   0   | 0 | 1 |   0   | 00 |   1   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 001101 | 100000 |    010    |   0   |   11   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 001101 | 100000 |    010    |   0   |   11   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 001101 | 100000 |    001    |   1   |   10   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 001101 | 100000 |    001    |   1   |   10   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 001101 | 100000 |    010    |   0   |   00   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    1    |   0   |   0   | OK
# | 0 | 0 | 001101 | 100000 |    010    |   0   |   00   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    1    |   0   |   0   | OK
# | 1 | 0 | 001101 | 100000 |    010    |   0   |   01   |   0   | 0 | 1 |   0   | 00 |   1   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 001110 | 100000 |    010    |   0   |   01   |   0   | 0 | 1 |   0   | 00 |   1   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 001110 | 100000 |    010    |   0   |   11   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 001110 | 100000 |    010    |   0   |   11   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 001110 | 100000 |    011    |   1   |   10   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 001110 | 100000 |    011    |   1   |   10   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 001110 | 100000 |    010    |   0   |   00   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    1    |   0   |   0   | OK
# | 0 | 0 | 001110 | 100000 |    010    |   0   |   00   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    1    |   0   |   0   | OK
# | 1 | 0 | 001110 | 100000 |    010    |   0   |   01   |   0   | 0 | 1 |   0   | 00 |   1   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 001010 | 100000 |    010    |   0   |   01   |   0   | 0 | 1 |   0   | 00 |   1   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 001010 | 100000 |    010    |   0   |   11   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 001010 | 100000 |    010    |   0   |   11   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 001010 | 100000 |    111    |   1   |   10   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 001010 | 100000 |    111    |   1   |   10   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 001010 | 100000 |    010    |   0   |   00   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    1    |   0   |   0   | OK
# | 0 | 0 | 001010 | 100000 |    010    |   0   |   00   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    1    |   0   |   0   | OK
# | 1 | 0 | 001010 | 100000 |    010    |   0   |   01   |   0   | 0 | 1 |   0   | 00 |   1   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 000101 | 100000 |    010    |   0   |   01   |   0   | 0 | 1 |   0   | 00 |   1   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 000101 | 100000 |    010    |   0   |   11   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 0 | 0 | 000101 | 100000 |    010    |   0   |   11   |   0   | 0 | 0 |   0   | 00 |   0   | 0 |    0    |   0   |   0   | OK
# | 1 | 0 | 000101 | 100000 |    110    |   1   |   00   |   1   | 0 | 0 |   0   | 01 |   0   | 0 |    0    |   0   |   1   | OK
# | 0 | 0 | 000101 | 100000 |    110    |   1   |   00   |   1   | 0 | 0 |   0   | 01 |   0   | 0 |    0    |   0   |   1   | OK
# | 1 | 0 | 000101 | 100000 |    010    |   0   |   01   |   0   | 0 | 1 |   0   | 00 |   1   | 0 |    0    |   0   |   0   | OK
# Testes Efetuados  = 126
# Erros Encontrados = 0
```

Figura 8 Unidade de controle - Transcript da simulação RTL

Também podemos ver o *RTL view* gerado a partir do modulo testado.

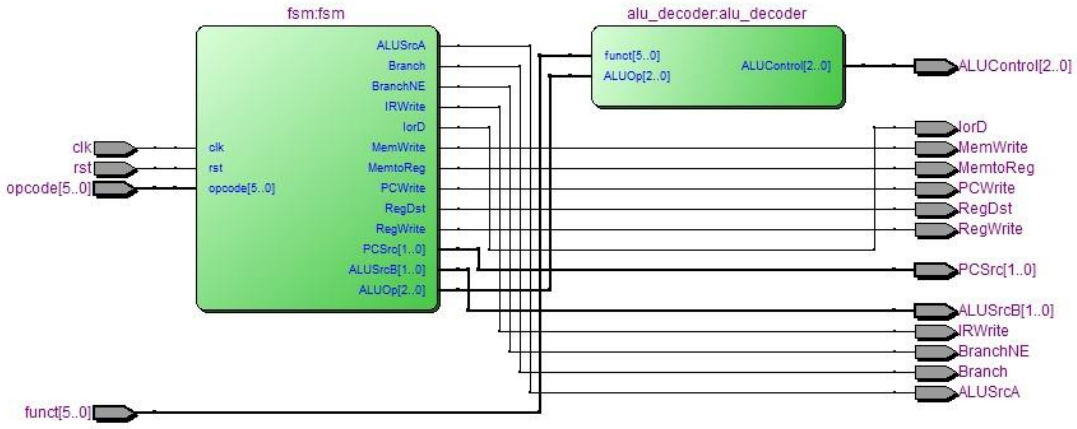

Figura 9 Unidade de controle – RTL view

2.6 Simulação *Gate Level*

Agora iniciamos a simulação Gate Level. Iniciamos o clock com um valor consideravelmente baixo e vamos subindo até que nenhum erro seja encontrado. Para este modulo, foram feitos três testes. É possível observar que à medida que o clock aumenta, o número de erros vai diminuindo. Como foram feitos muito testes, abaixo esta apenas um trecho do transcript, porém, podemos ver o número de testes efetuados e o número e erros.

```
# Error ALUSrcB: 01, expected 11
# | 1 | 0 | 001010 | 100000 |    010    |  0  |  01  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | ERROR
# | 0 | 0 | 001010 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# Error ALUControl: 010, expected 111
# | 1 | 0 | 001010 | 100000 |    010    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | ERROR
# | 0 | 0 | 001010 | 100000 |    111    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# Error ALUSrcA: 1, expected 0
# Error ALUSrcB: 10, expected 00
# Error ALUControl: 110, expected 010
# | 1 | 0 | 001010 | 100000 |    110    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | ERROR
# | 0 | 0 | 001010 | 100000 |    010    |  0  |  00  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | OK
# | 1 | 0 | 001010 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 000101 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# Error ALUSrcB: 01, expected 11
# | 1 | 0 | 000101 | 100000 |    010    |  0  |  01  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | ERROR
# | 0 | 0 | 000101 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# Error Branch: 0, expected 1
# Error ALUSrcB: 10, expected 00
# Error ALUControl: 010, expected 110
# | 1 | 0 | 000101 | 100000 |    010    |  1  |  10  |  0  |  0  |  0  |  0  | 01 |  0  |  0  |  0  |  0  |  1  | ERROR
# | 0 | 0 | 000101 | 100000 |    110    |  1  |  00  |  1  |  0  |  0  |  0  | 01 |  0  |  0  |  0  |  0  |  1  | OK
# Error Branch: 1, expected 0
# Error ALUControl: 110, expected 010
# | 1 | 0 | 000101 | 100000 |    110    |  0  |  01  |  1  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | ERROR
# Testes Efetuados  = 126
# Erros Encontrados = 69
```

Figura 10 Unidade de controle - clk #8

```
# Error ALUControl: 111, expected 011
# | 1 | 0 | 001110 | 100000 |    111    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | ERROR
# | 0 | 0 | 001110 | 100000 |    011    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# Error ALUSrcB: 10, expected 00
# Error ALUControl: 110, expected 010
# | 1 | 0 | 001110 | 100000 |    110    |  0  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | ERROR
# | 0 | 0 | 001110 | 100000 |    010    |  0  |  00  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | OK
# | 1 | 0 | 001110 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 001010 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 001010 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 001010 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 001010 | 100000 |    111    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 001010 | 100000 |    111    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# Error ALUSrcB: 10, expected 00
# Error ALUControl: 110, expected 010
# | 1 | 0 | 001010 | 100000 |    110    |  0  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | ERROR
# | 0 | 0 | 001010 | 100000 |    010    |  0  |  00  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | OK
# | 1 | 0 | 001010 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 000101 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 000101 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 000101 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# Error ALUSrcB: 10, expected 00
# | 1 | 0 | 000101 | 100000 |    110    |  1  |  10  |  1  |  0  |  0  |  0  | 01 |  0  |  0  |  0  |  0  |  1  | ERROR
# | 0 | 0 | 000101 | 100000 |    110    |  1  |  00  |  1  |  0  |  0  |  0  | 01 |  0  |  0  |  0  |  0  |  1  | OK
# | 1 | 0 | 000101 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# Testes Efetuados  = 126
# Erros Encontrados = 20
```

Figura 11 Unidade de controle - clk #9

```
# | 0 | 0 | 001101 | 100000 |    010    |  0  |  00  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | OK
# | 1 | 0 | 001101 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 001110 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 001110 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 001110 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 001110 | 100000 |    011    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 001110 | 100000 |    011    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 001110 | 100000 |    010    |  0  |  00  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | OK
# | 0 | 0 | 001110 | 100000 |    010    |  0  |  00  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | OK
# | 1 | 0 | 001110 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 001010 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 001010 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 001010 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 001010 | 100000 |    111    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 001010 | 100000 |    111    |  1  |  10  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 001010 | 100000 |    010    |  0  |  00  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | OK
# | 0 | 0 | 001010 | 100000 |    010    |  0  |  00  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  1  |  0  |  0  | OK
# | 1 | 0 | 001010 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 000101 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 000101 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 0 | 0 | 000101 | 100000 |    010    |  0  |  11  |  0  |  0  |  0  |  0  | 00 |  0  |  0  |  0  |  0  |  0  | OK
# | 1 | 0 | 000101 | 100000 |    110    |  1  |  00  |  1  |  0  |  0  |  0  | 01 |  0  |  0  |  0  |  0  |  1  | OK
# | 0 | 0 | 000101 | 100000 |    110    |  1  |  00  |  1  |  0  |  0  |  0  | 01 |  0  |  0  |  0  |  0  |  1  | OK
# | 1 | 0 | 000101 | 100000 |    010    |  0  |  01  |  0  |  0  |  1  |  0  | 00 |  1  |  0  |  0  |  0  |  0  | OK
# Testes Efetuados  = 126
# Erros Encontrados = 0
```

Figura 12 Unidade de controle - clk #10

Por fim, podemos ver os sinais de entrada e saída do modulo.

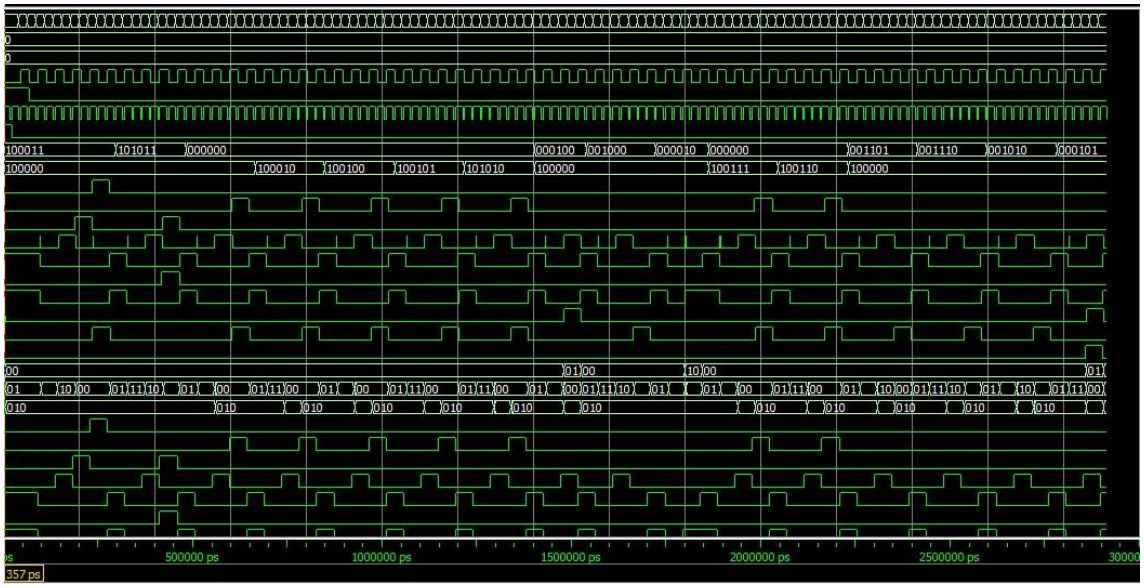

Figura 13 Unidade de controle - SInais de entrada e saída