



**UNIVERSIDADE FEDERAL DA PARAÍBA**  
**CENTRO DE INFORMÁTICA**

**CONCEPÇÃO ESTRUTURADA DE CIRCUITOS INTEGRADOS**  
**RELATÓRIO DA IMPLEMENTAÇÃO DO MIPS**

**GABRIEL DE OLIVEIRA MOURA SOARES - 20160163998**

**João Pessoa**

# Sumário

<b>1. Introdução</b>	<b>2</b>
<b>2. Unidade de controle</b>	<b>2</b>
2.1. Main Controller (FSM)	3
2.1.1. Estados	4
2.1.1.1. Golden Model	4
2.1.1.2. Implementação em System Verilog	6
2.1.1.3. Execução da Simulação	9
2.1.2. Sinais de controle	11
2.1.2.1. Golden Model	11
2.1.2.2. Implementação em System Verilog	13
2.1.2.3. Execução da Simulação	17
2.2. ALU Decoder	18
2.2.1. Golden Model	20
2.2.2. Implementação em System Verilog	21
2.2.3. Execução da simulação	24
2.3. Montando Unidade de Controle	25
2.3.1. Golden Model	25
2.3.2. Implementação em System Verilog	27
2.3.3. Execução da simulação	28

## 1. Introdução

Este trabalho é um relatório da implementação de um processador MIPS32 Multiciclo. Será usado como base a arquitetura mostrada durante as aulas de Conceção, mas será estendida para dar suporte às seguintes operações: LW, SW, ADD, SUB, AND, OR, NOR, XOR, SLT, ADDI, ORI, XORI, SLTI, BEQ, BNE, J.

## 2. Unidade de controle

Começaremos a implementação pela unidade de controle do MIPS. Ele é dividido em duas partes: o Main Controller e o ALU Decoder, como mostra a Figura 1.

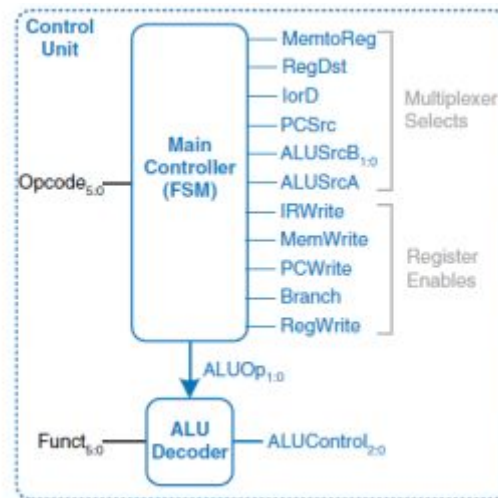


Figura 1: Estrutura da unidade de controle

O Main Controller tem como entrada o Opcode da instrução e é implementado como uma máquina de estados finitos (FSM - Finite State Machine em inglês), onde cada estado é um ciclo de alguma instrução. Apenas ele fornece a maioria dos sinais de controle que vai para o caminho de dados, com exceção apenas do ALUControl que dita a operação da unidade lógica aritmética (ULA ou ALU - Arithmetic Logic Unit em inglês).

O ALU Decoder é responsável por fornecer o sinal de controle da ULA. Para isso o Main Controller oferecerá o ALUOp que diz qual operação deverá ser feita pela ULA, mas no caso de uma operação do Tipo-R durante a etapa de execução o ALU Decoder precisará do Funct para saber qual operação deverá ser executada. Mais adiante veremos que no caso do jogo de instruções que estamos implementando, o ALU Decoder também precisará receber o Opcode como entrada, para calcular a operação durante o ciclo de execução de operações do Tipo I.

## 2.1. Main Controller (FSM)

Como citado, o Main Controller é uma máquina de estados, para implementá-la, foi usada como base a máquina mostrada na Figura 2.

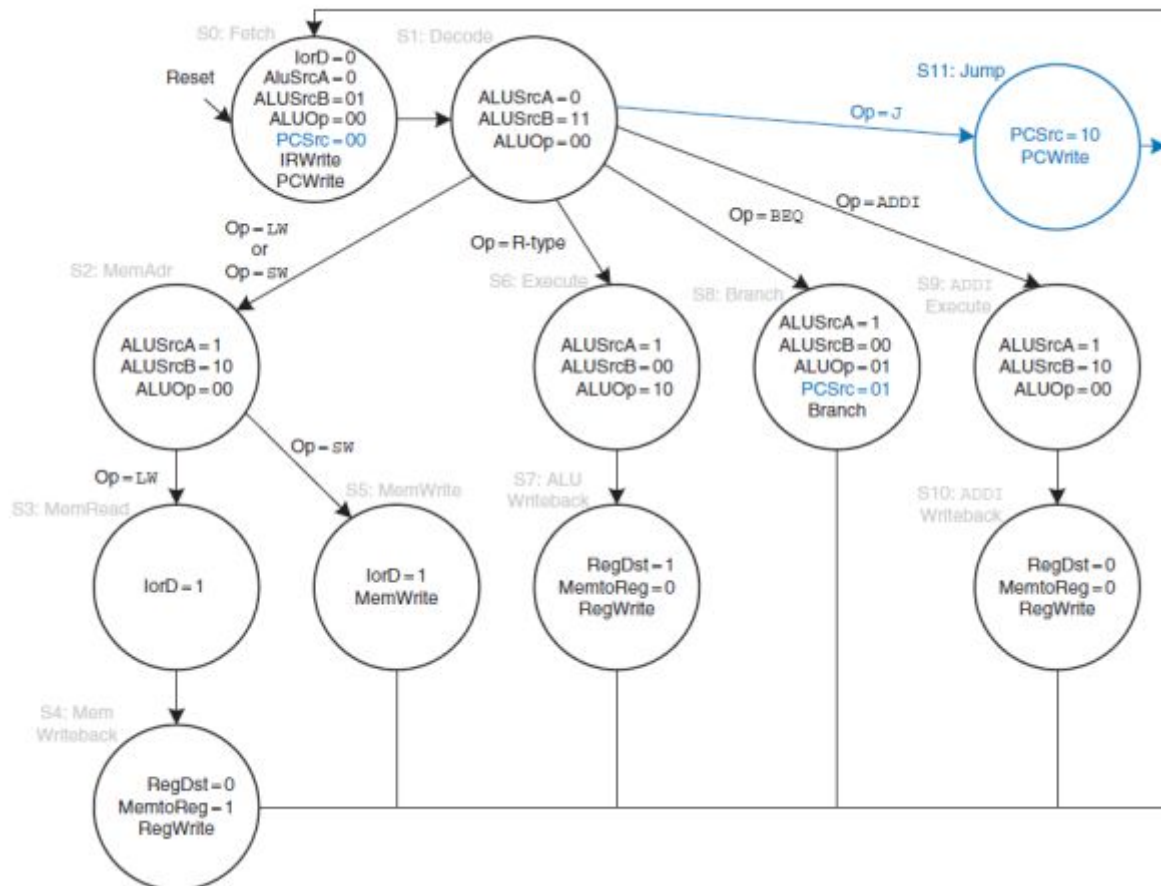


Figura 2: Máquina de estados usada como base

Essa máquina de estados dava suporte apenas a 1 operação do tipo I, o ADDI, mas podemos usar os mesmos estados para todas as outras operações do tipo I, bastando apenas mudar o  $ALUOp$  para que o ALU Decoder possa determinar corretamente qual a operação que a ULA vai fazer.

A máquina da Figura 2, também não dava suporte a operação BNE, por isso foi criado mais um estado, para dar suporte a essa operação: quando estiver no estado S1 e a operação foi um BNE, o próximo estado será o S12 (novo) que no próximo ciclo irá voltar para o S0. Além disso, a flag “Branch” será substituído por 2 flags: “BranchEQ” e “BranchNE” que juntamente com a flag “Zero” da ULA determina se haverá o branch. Nesse caso para determinar se haverá o branch (RB - Realizar Branch), faremos a seguinte operação (através dos componentes lógicos):

$$RB = (BranchEQ \wedge Zero) \vee (BranchNE \wedge \neg Zero)$$

### 2.1.1. Estados

Começaremos a implementação da máquina de estados apenas com os estados e suas transições, sem se preocupar com os sinais de controle. A implementação segue a máquina descrita anteriormente.

#### 2.1.1.1. Golden Model

O Golden Model foi implementado na linguagem Python. Primeiro criamos um arquivo onde definimos algumas constantes que ajudarão na implementação, adicionamos as constantes para representar os estados S0..S12, constantes que armazenam o opcode das instruções, e também constantes que armazenam os functs das operações do tipo R. Todas essas constantes podem ser vistas na Figura 3. E na Figura 4 vemos a implementação do MainController.

```

1  # states
2  S0 = 0 # Fetch
3  S1 = 1 # Decode
4  S2 = 2 # MemAdr
5  S3 = 3 # MemRead
6  S4 = 4 # Mem Writeback
7  S5 = 5 # MemWrite
8  S6 = 6 # R-type Execute
9  S7 = 7 # ALU Writeback
10 S8 = 8 # Branch EQ
11 S9 = 9 # I-Type Execute
12 S10 = 10 # I-Type Writeback
13 S11 = 11 # Jump
14 S12 = 12 # Branch NE

16 # instructions opcode
17 LW = 0b100011
18 SW = 0b101011
19 R_TYPE = {0b000000}
20 BEQ = 0b000100
21 BNE = 0b000101
22 J = 0b000010
23 ADDI = 0b001000
24 ORI = 0b001101
25 XORI = 0b001110
26 SLTI = 0b001010
27 I_TYPE = {ADDI, ORI, XORI, SLTI}

29 # R-Type Funct
30 ADD = 0b100000
31 SUB = 0b100010
32 AND = 0b100100
33 OR = 0b100101
34 NOR = 0b100111
35 XOR = 0b100110
36 SLT = 0b101010

```

Figura 3: definitions.py

Para gerar os vetores de teste, foi criada uma função que recebe um arquivo de entrada com o sinal de reset, e todos os opcodes que englobam as operações suportadas por essa implementação.. Ao lado podemos ver o conteúdo desse arquivo. Na Figura 5 podemos ver a implementação dessa função.

Essa função gera um arquivo com os vetores de teste, em que a cada transição do clock é gravado os sinais de clock, reset, o opcode de entrada e o estado esperado naquele momento. A Figura 6 mostra o conteúdo desse arquivo.

```

1_100011
1_100011
0_100011
0_101011
0_000000
0_000100
0_000101
0_000010
0_001000
0_001101
0_001110
0_001010

```

```

4 class MainController:
5     def __init__(self):
6         self.state = S0
7         self.nextState = S1
8
9     def get_next_state(self, opcode):
10        if self.state == S0:
11            self.nextState = S1
12        elif self.state == S1:
13            if opcode in [SW, LW]:
14                self.nextState = S2
15            elif opcode in R_TYPE:
16                self.nextState = S6
17            elif opcode == BEQ:
18                self.nextState = S8
19            elif opcode in I_TYPE:
20                self.nextState = S9
21            elif opcode == J:
22                self.nextState = S11
23            elif opcode == BNE:
24                self.nextState = S12
25            else:
26                raise ValueError
27        elif self.state == S2:
28            if opcode == LW:
29                self.nextState = S3
30            elif opcode == SW:
31                self.nextState = S5
32            elif self.state == S3:
33                self.nextState = S4
34
35        elif self.state == S4:
36            self.nextState = S0
37        elif self.state == S5:
38            self.nextState = S0
39        elif self.state == S6:
40            self.nextState = S7
41        elif self.state == S7:
42            self.nextState = S0
43        elif self.state == S8:
44            self.nextState = S0
45        elif self.state == S9:
46            self.nextState = S10
47        elif self.state == S10:
48            self.nextState = S0
49        elif self.state == S11:
50            self.nextState = S0
51        elif self.state == S12:
52            self.nextState = S0
53        else:
54            raise ValueError
55
56        return self.nextState
57
58    def go_next_state(self, rst):
59        if rst == 1:
60            self.state = S0
61        else:
62            self.state = self.nextState

```

Figura 4: Implementação do GM do Main Controller

```

def create_main_controller_tvs():
    file = open('main_controller_input.txt', 'r')
    out = open('../simulation/modelsim/main_controller.tv', 'w')
    clk = 0
    main_controller = MainController()
    rst = 1
    opcode = None
    finished = True

    while True:
        if finished:
            finished = False
            inp = file.readline()
            if not inp or inp is None:
                break

            rst = int(inp[0])
            opcode = int(inp[2:8], 2)

            if clk == 1 or rst == 1:
                main_controller.go_next_state(rst)
                main_controller.get_next_state(opcode)
                if main_controller.state == S0:
                    finished = True

            sout = "{:1b}_{:1b}_{:06b}_{:04b}".format(clk, rst, opcode, main_controller.state)
            print(sout)
            out.write("{}\n".format(sout))

            clk = 1 - clk

    out.close()
    file.close()

```

Figura 5: Gerador dos TVs do Main Controller

1	0_1_100011_0000	28	1_0_000000_0111	55	0_0_001000_1010
2	1_1_100011_0000	29	0_0_000000_0111	56	1_0_001000_0000
3	0_1_100011_0000	30	1_0_000000_0000	57	0_0_001101_0000
4	1_1_100011_0000	31	0_0_000100_0000	58	1_0_001101_0001
5	0_0_100011_0000	32	1_0_000100_0001	59	0_0_001101_0001
6	1_0_100011_0001	33	0_0_000100_0001	60	1_0_001101_1001
7	0_0_100011_0001	34	1_0_000100_1000	61	0_0_001101_1001
8	1_0_100011_0010	35	0_0_000100_1000	62	1_0_001101_1010
9	0_0_100011_0010	36	1_0_000100_0000	63	0_0_001101_1010
10	1_0_100011_0011	37	0_0_000101_0000	64	1_0_001101_0000
11	0_0_100011_0011	38	1_0_000101_0001	65	0_0_001110_0000
12	1_0_100011_0100	39	0_0_000101_0001	66	1_0_001110_0001
13	0_0_100011_0100	40	1_0_000101_1100	67	0_0_001110_0001
14	1_0_100011_0000	41	0_0_000101_1100	68	1_0_001110_1001
15	0_0_101011_0000	42	1_0_000101_0000	69	0_0_001110_1001
16	1_0_101011_0001	43	0_0_000010_0000	70	1_0_001110_1010
17	0_0_101011_0001	44	1_0_000010_0001	71	0_0_001110_1010
18	1_0_101011_0010	45	0_0_000010_0001	72	1_0_001110_0000
19	0_0_101011_0010	46	1_0_000010_1011	73	0_0_001010_0000
20	1_0_101011_0101	47	0_0_000010_1011	74	1_0_001010_0001
21	0_0_101011_0101	48	1_0_000010_0000	75	0_0_001010_0001
22	1_0_101011_0000	49	0_0_001000_0000	76	1_0_001010_1001
23	0_0_000000_0000	50	1_0_001000_0001	77	0_0_001010_1001
24	1_0_000000_0001	51	0_0_001000_0001	78	1_0_001010_1010
25	0_0_000000_0001	52	1_0_001000_1001	79	0_0_001010_1010
26	1_0_000000_0110	53	0_0_001000_1001	80	1_0_001010_0000
27	0_0_000000_0110	54	1_0_001000_1010		

Figura 6: main\_controller.tv

### 2.1.1.2. Implementação em System Verilog

Na implementação em system verilog, primeiro definimos o módulo, com as entradas dos sinais de clock, reset, e o opcode da instrução que está sendo executada, e por enquanto a saída será apenas o estado que a máquina se encontra.

```

1 module main_controller(
2     input logic clk, rst,
3     input logic [5:0] opcode,
4     output logic [3:0] state
5 );
6

```

Figura 7: Definição do módulo Main Controller

Depois foi criado uma enum State para representar os 13 estados, ao qual, a cada um é atribuído um número de 3 bits, sequencial de acordo com a ordem de definição.

```

6
7 typedef enum logic [3:0] {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12} State;
8

```

Figura 8: Definição da enum de estados

Depois criamos parâmetros que atribuem um nome para cada código opcode, apenas para facilitar a legibilidade do código a seguir.



```

9    localparam [5:0] LW = 6'b100011,
10    SW = 6'b101011,
11    R_TYPE = 6'b000000,
12    BEQ = 6'b000100,
13    BNE = 6'b000101,
14    J = 6'b000010,
15    ADDI = 6'b001000,
16    ORI = 6'b001101,
17    XORI = 6'b001110,
18    SLTI = 6'b001010;
19

```

Figura 9: Criando parâmetros para os opcodes

Agora criamos a variável 'thisState' que vai representar o estado atual que a máquina se encontra e a variável 'nextState' que representa o estado que a máquina irá no próximo ciclo. Além disso, atribuímos 'thisState' a saída 'state' do módulo.

```

19
20    State thisState, nextState;
21    assign state = thisState;
22

```

Figura 10: Variáveis de estado

E então criamos um bloco que atribui o próximo estado ao estado atual em toda subida de clock. Além disso, sempre que houver um sinal de reset, o estado atual volta para o S0, independente do clock (assíncrono).

```

22
23    // muda o estado
24    always_ff @(posedge clk, posedge rst) begin
25        if(rst) thisState <= S0;
26        else thisState <= nextState;
27    end
28

```

Figura 11: Reset e atribuição do estado na subida do clock

E finalmente temos o bloco que calcula o próximo estado toda vez que o estado atual muda (combinacional).



```

// calcula proximo estado
always_comb begin
    case(thisState)
        S0: nextState <= S1;
        S1: case(opcode)
            LW: nextState <= S2;
            SW: nextState <= S2;
            R_TYPE: nextState <= S6;
            BEQ: nextState <= S8;
            ADDI: nextState <= S9;
            ORI: nextState <= S9;
            XORI: nextState <= S9;
            SLTI: nextState <= S9;
            J: nextState <= S11;
            BNE: nextState <= S12;
            default: nextState <= S2;
        endcase
        S2: case(opcode)
            LW: nextState <= S3;
            SW: nextState <= S5;
            default: nextState <= S3;
        endcase
        S3: nextState <= S4;
        S4: nextState <= S0;
        S5: nextState <= S0;
        S6: nextState <= S7;
        S7: nextState <= S0;
        S8: nextState <= S0;
        S9: nextState <= S10;
        S10: nextState <= S0;
        S11: nextState <= S0;
        S12: nextState <= S0;
    endcase
end

```

Figura 12: Cálculo do próximo estado

Na implementação do testbench, é instanciado o main\_controller, e para cada vetor de teste no arquivo criado pelo Golden Model é atribuído o clock, reset e opcode, e então é comparado os estados de saída.

```

1  `timescale 1ns/100ps
2  module main_controller_tb;
3
4      int counter, errors, aux_error;
5      logic clk, rst;
6      integer file;
7
8      logic clock, reset;
9      logic [5:0] opcode;
10     logic [3:0] state, state_esperado;
11
12     parameter max_vectors = 78;
13     logic [11:0] Vectors[max_vectors];
14
15     main_controller dut(clock, reset, opcode, state);
16
17     initial begin
18         counter = 0; errors = 0;
19         rst = 1'b1; #12; rst = 0;
20         clk=0;
21
22         if(~rst) begin
23             $readmemb("main_controller.tv", vectors);
24         end
25
26         file = $fopen("main_controller_out.txt");
27
28         $display("Iniciando Testbench");
29         $display("-----");
30         $display("| clk | rst | opcode | state |");
31
32         $fwrite(file, "Iniciando Testbench");
33         $fwrite(file, "-----");
34         $fwrite(file, "| clk | rst | opcode | state |");
35     end
36
37     always begin
38         clk = 1; #10;
39         clk = 0; #5;
40     end
41
42     always @(posedge clk) begin
43         if(~rst) begin
44             {clock, reset, opcode, state_esperado} = vectors[counter];
45         end
46     end
47
48     always @(negedge clk) //Sempre (que o clock descer)
49     if(~rst) begin
50         if(state_esperado !== 6'bx) begin
51             aux_error = errors;
52
53             for(int i = 0; i < 6; i++) begin
54                 assert (state[i] == state_esperado[i]) else begin
55                     //Mostra mensagem de erro se a saída do DUT for diferente da saída esperada
56                     $error("Erro S na linha %d bit %d, saída = %b, (%b esperado)", counter+1, i, state[i], state_esperado[i]);
57                     errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
58                 end
59             end
60
61             if(aux_error == errors) begin // Nao houve erro
62                 $display("| %b | %b | %b | %b | OK", clock, reset, opcode, state);
63                 $fwrite(file, "| %b | %b | %b | %b | OK", clock, reset, opcode, state);
64             end
65
66             if(counter+1 == max_vectors) begin
67                 $display("Testes Efetuados = %0d", counter+1);
68                 $display("Erros Encontrados = %0d", errors);
69                 $fwrite(file, "Testes Efetuados = %0d", counter+1);
70                 $fwrite(file, "Erros Encontrados = %0d", errors);
71                 #10
72                 $stop;
73             end
74         end
75         counter++; //Incrementa contador dos vetores de teste
76     end
77 endmodule
78

```

Figura 13: main\_controller\_tb.sv

### 2.1.1.3. Execução da Simulação

Abaixo temos o resultado da simulação feita com o ModelSim.

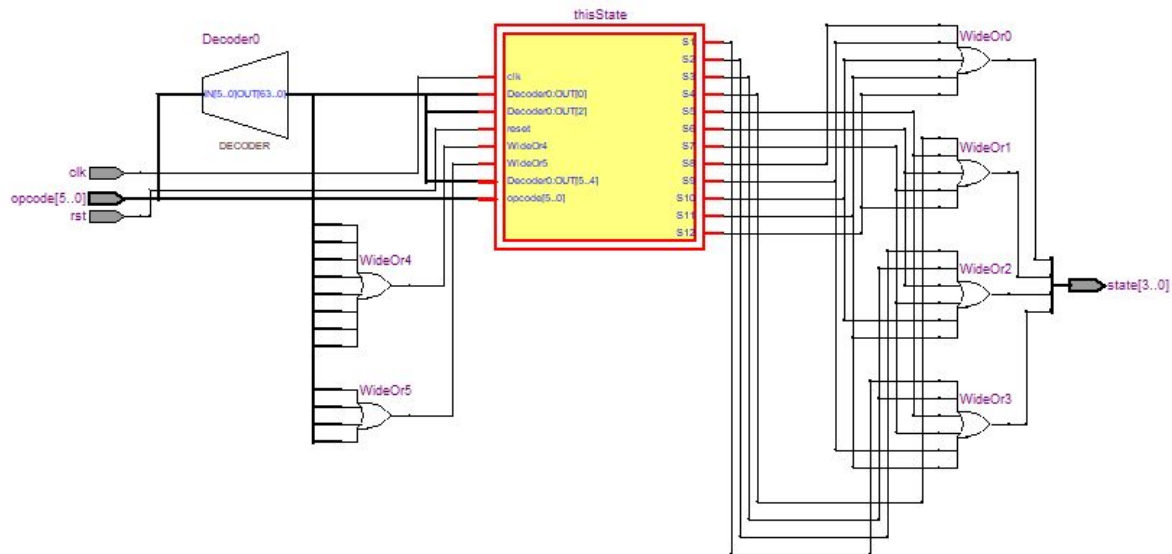


Figura 14: Visualização RTL do Main Controller

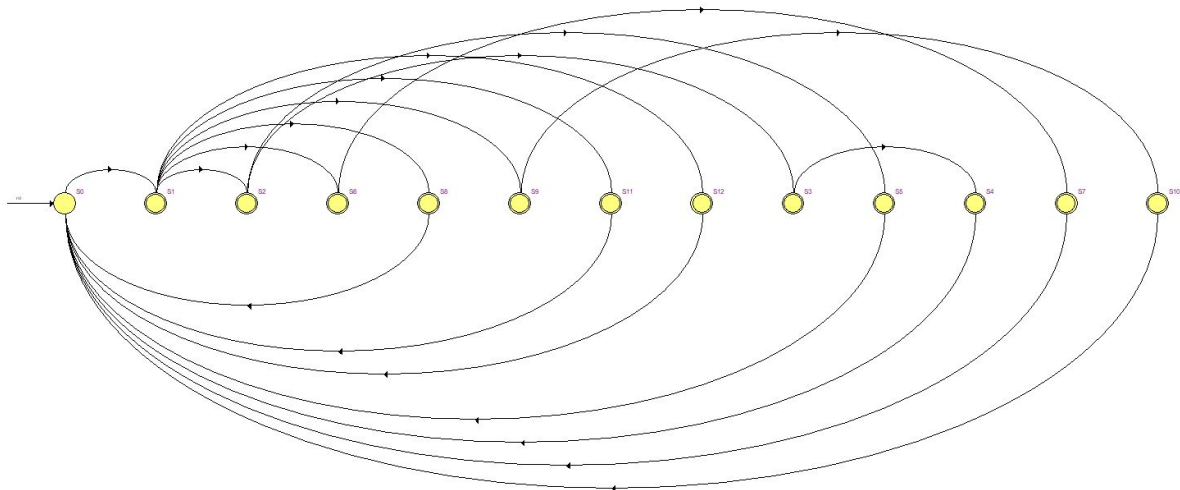


Figura 15: Diagrama de transição de estados

```
# | 1 | 0 | 001010 | 0001 | OK
# | 0 | 0 | 001010 | 0001 | OK
# | 1 | 0 | 001010 | 1001 | OK
# | 0 | 0 | 001010 | 1001 | OK
# | 1 | 0 | 001010 | 1010 | OK
# | 0 | 0 | 001010 | 1010 | OK
# | 1 | 0 | 001010 | 0000 | OK
# Testes Efetuados = 78
# Erros Encontrados = 0
```

Figura 16: Últimas linhas do resultado da simulação

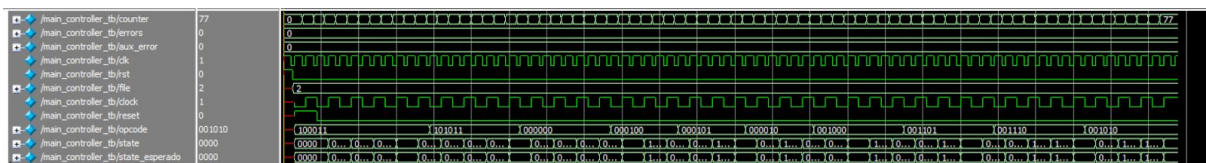


Figura 17: Visualização de onda da simulação RTL

Foi feita a simulação Gate Level e encontrado que o menor tempo de clock em nível um de forma que a simulação obtivesse 0 erros foi 9 unidades de tempo.

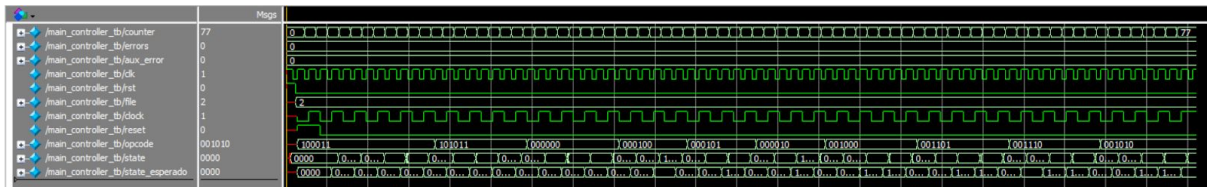


Figura 18: Visualização de onda da simulação Gate Level

## 2.1.2. Sinais de controle

Agora que a máquina está passando por todos os estados corretamente, agora basta adicionar os sinais de controle em cada estado de acordo com a máquina mostrada na Figura 2, e as alterações descritas na seção 2.1.

### 2.1.2.1. Golden Model

No Golden Model basta apenas adicionar os sinais de controle e alterá-los em cada estado. O resultado é mostrado na Figura 20. Observe que antes de atribuir os sinais, todos os sinais são atribuídos valores padrão, para se ter certeza que apenas os sinais corretos serão atribuídos no estado.

Na função que gera os vetores de teste, basta apenas modificar o código que escreve os resultados no arquivo, para incluir todos os sinais de controle, como pode ser visto na Figura 19.

```
sout = "{:1b}_{:1b}_{:06b}_{:04b}_{:1b}_{:1b}_{:1b}_{:02b}_{:02b}_{:1b}_{:1b}_{:1b}_{:1b}_{:1b}_{:1b}_{:1b}" \
      "_{:02b}\n".format(clk, rst, opcode, mc.state, mc.MemtoReg, mc.RegDst, mc.IorD, mc.PCsrc, mc.ALUSrcB,
                        mc.ALUSrcA, mc.IRWrite, mc.MemWrite, mc.PCWrite, mc.BranchEQ, mc.BranchNE,
                        mc.RegWrite, mc.ALUOp)
print(sout, end="")
out.write(sout)
```

Figura 19: Escrevendo sinais de controle nos vetores de teste

```

3
4 class MainController:
5     def __init__(self):
6         self.state = S0
7         self.nextState = S1
8         self.MemtoReg = 0b0
9         self.RegDst = 0b0
10        self.IorD = 0b0
11        self.PCSrc = 0b00
12        self.ALUsrcB = 0b01
13        self.ALUsrcA = 0b0
14        self.IRWrite = True
15        self.MemWrite = False
16        self.PCWrite = True
17        self.BranchEQ = False
18        self.BranchNE = False
19        self.RegWrite = False
20        self.ALUOp = 0b00
21
22    def get_next_state(self, opcode):
23        # atribui valor padrao aos sinais
24        # para depois atribuir somente os
25        # sinais do estado atual
26        self.MemtoReg = 0b0
27        self.RegDst = 0b0
28        self.IorD = 0b0
29        self.PCSrc = 0b00
30        self.ALUsrcB = 0b01
31        self.ALUsrcA = 0b0
32        self.IRWrite = False
33        self.MemWrite = False
34        self.PCWrite = False
35        self.BranchEQ = False
36        self.BranchNE = False
37        self.RegWrite = False
38        self.ALUOp = 0b00
39
40        if self.state == S0:
41            self.IorD = 0b0
42            self.ALUsrcA = 0b0
43            self.ALUsrcB = 0b01
44            self.ALUOp = 0b00
45            self.PCSrc = 0b00
46            self.IRWrite = True
47            self.PCWrite = True
48
49            self.nextState = S1
50
51        elif self.state == S1:
52            self.ALUsrcA = 0b0
53            self.ALUsrcB = 0b11
54            self.ALUOp = 0b00
55
56            if opcode in [SW, LW]:
57                self.nextState = S2
58            elif opcode in R_TYPE:
59                self.nextState = S6
60            elif opcode == BEQ:
61                self.nextState = S8
62            elif opcode in I_TYPE:
63                self.nextState = S9
64            elif opcode == J:
65                self.nextState = S11
66            elif opcode == BNE:
67                self.nextState = S12
68            else:
69                raise ValueError
70
71        elif self.state == S2:
72            self.ALUsrcA = 0b1
73            self.ALUsrcB = 0b10
74            self.ALUOp = 0b00
75
76            if opcode == LW:
77                self.nextState = S3
78            elif opcode == SW:
79                self.nextState = S5
80
81        elif self.state == S3:
82            self.IorD = 0b1
83
84            self.nextState = S4
85
86        elif self.state == S4:
87            self.RegDst = 0b0
88            self.MemtoReg = 0b1
89            self.RegWrite = True
90
91            self.nextState = S0
92
93        elif self.state == S5:
94            self.IorD = 0b1
95            self.MemWrite = True
96
97            self.nextState = S0
98
99        elif self.state == S6:
100            self.ALUsrcA = 0b1
101            self.ALUsrcB = 0b00
102            self.ALUOp = 0b10
103
104            self.nextState = S7
105
106        elif self.state == S7:
107            self.RegDst = 0b1
108            self.MemtoReg = 0b0
109            self.RegWrite = True
110
111            self.nextState = S0
112
113        elif self.state == S8:
114            self.ALUsrcA = 0b1
115            self.ALUsrcB = 0b00
116            self.ALUOp = 0b01
117            self.PCSrc = 0b01
118            self.BranchEQ = True
119
120            self.nextState = S0
121
122        elif self.state == S9:
123            self.ALUsrcA = 0b1
124            self.ALUsrcB = 0b10
125            self.ALUOp = 0b11
126
127            self.nextState = S10
128
129        elif self.state == S10:
130            self.RegDst = 0b0
131            self.MemtoReg = 0b0
132            self.RegWrite = True
133
134            self.nextState = S0
135
136        elif self.state == S11:
137            self.PCSrc = 0b10
138            self.PCWrite = True
139
140            self.nextState = S0
141
142        elif self.state == S12:
143            self.ALUsrcA = 0b1
144            self.ALUsrcB = 0b00
145            self.ALUOp = 0b01
146            self.PCSrc = 0b01
147            self.BranchNE = True
148
149            self.nextState = S0
150
151        else:
152            raise ValueError
153
154        return self.nextState
155
156    def go_next_state(self, rst):
157        if rst == 1:
158            self.state = S0
159        else:
160            self.state = self.nextState
161

```

Figura 20: Golden Model com sinais de controle



0_1_100011_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	1_0_000101_0000_0_0_0_00_01_0_1_0_1_0_0_0_00
1_1_100011_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	0_0_000010_0000_0_0_0_00_01_0_1_0_1_0_0_0_00
0_0_100011_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	1_0_000010_0001_0_0_0_00_11_0_0_0_0_0_0_0_00
1_0_100011_0001_0_0_0_00_11_0_0_0_0_0_0_0_00	0_0_000010_0001_0_0_0_00_11_0_0_0_0_0_0_0_00
0_0_100011_0001_0_0_0_00_11_0_0_0_0_0_0_0_00	1_0_000010_1011_0_0_0_10_00_0_0_0_1_0_0_0_00
1_0_100011_0010_0_0_0_00_10_1_0_0_0_0_0_0_00	0_0_000010_1011_0_0_0_10_00_0_0_0_1_0_0_0_00
0_0_100011_0010_0_0_0_00_10_1_0_0_0_0_0_0_00	1_0_000010_0000_0_0_0_00_01_0_1_0_1_0_0_0_00
1_0_100011_0011_0_0_1_00_00_0_0_0_0_0_0_0_00	0_0_001000_0000_0_0_0_00_01_0_1_0_1_0_0_0_00
0_0_100011_0011_0_0_1_00_00_0_0_0_0_0_0_0_00	1_0_001000_0001_0_0_0_00_11_0_0_0_0_0_0_0_00
1_0_100011_0100_1_0_0_00_00_0_0_0_0_0_0_1_00	0_0_001000_0001_0_0_0_00_11_0_0_0_0_0_0_0_00
0_0_100011_0100_1_0_0_00_00_0_0_0_0_0_0_1_00	1_0_001000_1001_0_0_0_00_10_1_0_0_0_0_0_0_11
1_0_100011_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	0_0_001000_1001_0_0_0_00_10_1_0_0_0_0_0_0_11
0_0_101011_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	1_0_001000_1010_0_0_0_00_00_0_0_0_0_0_0_1_00
1_0_101011_0001_0_0_0_00_11_0_0_0_0_0_0_0_00	0_0_001000_1010_0_0_0_00_00_0_0_0_0_0_0_1_00
0_0_101011_0001_0_0_0_00_11_0_0_0_0_0_0_0_00	1_0_001000_0000_0_0_0_00_01_0_1_0_1_0_0_0_00
1_0_101011_0010_0_0_0_00_10_1_0_0_0_0_0_0_00	0_0_001101_0000_0_0_0_00_01_0_1_0_1_0_0_0_00
0_0_101011_0010_0_0_0_00_10_1_0_0_0_0_0_0_00	1_0_001101_0001_0_0_0_00_11_0_0_0_0_0_0_0_00
1_0_101011_0101_0_0_1_00_00_0_0_1_0_0_0_0_00	0_0_001101_0001_0_0_0_00_11_0_0_0_0_0_0_0_00
0_0_101011_0101_0_0_1_00_00_0_0_1_0_0_0_0_00	1_0_001101_1001_0_0_0_00_10_1_0_0_0_0_0_0_11
1_0_101011_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	0_0_001101_1001_0_0_0_00_10_1_0_0_0_0_0_0_11
0_0_000000_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	1_0_001101_1010_0_0_0_00_00_0_0_0_0_0_0_1_00
1_0_000000_0001_0_0_0_00_11_0_0_0_0_0_0_0_00	0_0_001101_1010_0_0_0_00_00_0_0_0_0_0_0_1_00
0_0_000000_0001_0_0_0_00_11_0_0_0_0_0_0_0_00	1_0_001101_0000_0_0_0_00_01_0_1_0_1_0_0_0_00
1_0_000000_0110_0_0_0_00_00_1_0_0_0_0_0_0_10	0_0_001110_0000_0_0_0_00_01_0_1_0_1_0_0_0_00
0_0_000000_0110_0_0_0_00_00_1_0_0_0_0_0_0_10	1_0_001110_0001_0_0_0_00_11_0_0_0_0_0_0_0_00
1_0_000000_0111_0_1_0_00_00_0_0_0_0_0_0_1_00	0_0_001110_0001_0_0_0_00_11_0_0_0_0_0_0_0_00
0_0_000000_0111_0_1_0_00_00_0_0_0_0_0_0_1_00	1_0_001110_1001_0_0_0_00_10_1_0_0_0_0_0_0_11
1_0_000000_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	0_0_001110_1001_0_0_0_00_10_1_0_0_0_0_0_0_11
0_0_000100_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	1_0_001110_1010_0_0_0_00_00_0_0_0_0_0_0_1_00
1_0_000100_0001_0_0_0_00_11_0_0_0_0_0_0_0_00	0_0_001110_1010_0_0_0_00_00_0_0_0_0_0_0_1_00
0_0_000100_0001_0_0_0_00_11_0_0_0_0_0_0_0_00	1_0_001110_0000_0_0_0_00_01_0_1_0_1_0_0_0_00
1_0_000100_1000_0_0_0_01_00_1_0_0_0_1_0_0_01	0_0_001010_0000_0_0_0_00_01_0_1_0_1_0_0_0_00
0_0_000100_1000_0_0_0_01_00_1_0_0_0_1_0_0_01	1_0_001010_0001_0_0_0_00_11_0_0_0_0_0_0_0_00
1_0_000100_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	0_0_001010_0001_0_0_0_00_11_0_0_0_0_0_0_0_00
0_0_000101_0000_0_0_0_00_01_0_1_0_1_0_0_0_00	1_0_001010_1001_0_0_0_00_10_1_0_0_0_0_0_0_11
1_0_000101_0001_0_0_0_00_11_0_0_0_0_0_0_0_00	0_0_001010_1001_0_0_0_00_10_1_0_0_0_0_0_0_11
0_0_000101_0001_0_0_0_00_11_0_0_0_0_0_0_0_00	1_0_001010_1010_0_0_0_00_00_0_0_0_0_0_0_1_00
1_0_000101_1100_0_0_0_01_00_1_0_0_0_0_1_0_01	0_0_001010_1010_0_0_0_00_00_0_0_0_0_0_0_1_00
0_0_000101_1100_0_0_0_01_00_1_0_0_0_0_1_0_01	1_0_001010_0000_0_0_0_00_01_0_1_0_1_0_0_0_00

Figura 21: Vetores de teste com sinal de controle

### 2.1.2.2. Implementação em System Verilog

Na implementação em system verilog, devemos adicionar cada sinal de controle como uma saída na declaração do módulo, criar um novo bloco 'always\_comb' que atribui a saída de cada estado. A Figura 22 mostra esse bloco.

```

67 // calcula proximo estado
68 always_comb begin
69     case (thisState)
70     S0: begin
71         IorD <= 1'b0;
72         ALUSrcA <= 1'b0;
73         ALUSrcB <= 2'b01;
74         ALUOp <= 2'b00;
75         PCSrc <= 2'b00;
76         IRWrite <= 1'b1;
77         PCWrite <= 1'b1;
78
79         MemtoReg <= 1'b0;
80         RegDst <= 1'b0;
81         MemWrite <= 1'b0;
82         BranchEQ <= 1'b0;
83         BranchNE <= 1'b0;
84         RegWrite <= 1'b0;
85     end
86     S1: begin
87         ALUSrcA <= 1'b0;
88         ALUSrcB <= 2'b11;
89         ALUOp <= 2'b00;
90
91         MemtoReg <= 1'b0;
92         RegDst <= 1'b0;
93         IorD <= 1'b0;
94         PCSrc <= 2'b00;
95         IRWrite <= 1'b0;
96         MemWrite <= 1'b0;
97         PCWrite <= 1'b0;
98         BranchEQ <= 1'b0;
99         BranchNE <= 1'b0;
100        RegWrite <= 1'b0;
101    end
102    S2: begin
103        ALUSrcA <= 1'b1;
104        ALUSrcB <= 2'b10;
105        ALUOp <= 2'b00;
106
107        MemtoReg <= 1'b0;
108        RegDst <= 1'b0;
109        IorD <= 1'b0;
110        PCSrc <= 2'b00;
111        IRWrite <= 1'b0;
112        MemWrite <= 1'b0;
113        PCWrite <= 1'b0;
114        BranchEQ <= 1'b0;
115        BranchNE <= 1'b0;
116        RegWrite <= 1'b0;
117    end
118    S3: begin
119        IorD <= 1'b1;
120
121        MemtoReg <= 1'b0;
122        RegDst <= 1'b0;
123        PCSrc <= 2'b00;
124        ALUSrcB <= 2'b00;
125        ALUSrcA <= 1'b0;
126        IRWrite <= 1'b0;
127        MemWrite <= 1'b0;
128        PCWrite <= 1'b0;
129        BranchEQ <= 1'b0;
130        BranchNE <= 1'b0;
131        RegWrite <= 1'b0;
132        ALUOp <= 2'b00;
133    end
134    S4: begin
135        RegDst <= 1'b0;
136        MemtoReg <= 1'b1;
137        RegWrite <= 1'b1;
138
139        IorD <= 1'b0;
140        PCSrc <= 2'b00;
141        ALUSrcB <= 2'b00;
142        ALUSrcA <= 1'b0;
143        IRWrite <= 1'b0;
144        MemWrite <= 1'b0;
145        PCWrite <= 1'b0;
146        BranchEQ <= 1'b0;
147        BranchNE <= 1'b0;
148        ALUOp <= 2'b00;
149    end
150    S5: begin
151        IorD <= 1'b1;
152        MemWrite <= 1'b1;
153
154        MemtoReg <= 1'b0;
155        RegDst <= 1'b0;
156        PCSrc <= 2'b00;
157        ALUSrcB <= 2'b00;
158        ALUSrcA <= 1'b0;
159        IRWrite <= 1'b0;
160        PCWrite <= 1'b0;
161        BranchEQ <= 1'b0;
162        BranchNE <= 1'b0;
163        RegWrite <= 1'b0;
164        ALUOp <= 2'b00;
165    end
166    S6: begin
167        ALUSrcA <= 1'b1;
168        ALUSrcB <= 2'b00;
169        ALUOp <= 2'b10;
170
171        MemtoReg <= 1'b0;
172        RegDst <= 1'b0;
173        IorD <= 1'b0;
174        PCSrc <= 2'b00;
175
176        IRWrite <= 1'b0;
177        MemWrite <= 1'b0;
178        PCWrite <= 1'b0;
179        BranchEQ <= 1'b0;
180        BranchNE <= 1'b0;
181        RegWrite <= 1'b0;
182    end
183    S7: begin
184        RegDst <= 1'b1;
185        MemtoReg <= 1'b0;
186        RegWrite <= 1'b1;
187
188        IorD <= 1'b0;
189        PCSrc <= 2'b00;
190        ALUSrcB <= 2'b00;
191        ALUSrcA <= 1'b0;
192        IRWrite <= 1'b0;
193        MemWrite <= 1'b0;
194        PCWrite <= 1'b0;
195        BranchEQ <= 1'b0;
196        BranchNE <= 1'b0;
197        ALUOp <= 2'b00;
198    end
199    S8: begin
200        ALUSrcA <= 1'b1;
201        ALUSrcB <= 2'b00;
202        ALUOp <= 2'b01;
203        PCSrc <= 2'b01;
204        BranchEQ <= 1'b1;
205
206        MemtoReg <= 1'b0;
207        RegDst <= 1'b0;
208        IorD <= 1'b0;
209        IRWrite <= 1'b0;
210        MemWrite <= 1'b0;
211        PCWrite <= 1'b0;
212        BranchNE <= 1'b0;
213        RegWrite <= 1'b0;
214    end
215    S9: begin
216        ALUSrcA <= 1'b1;
217        ALUSrcB <= 2'b10;
218        ALUOp <= 2'b11;
219
220        MemtoReg <= 1'b0;
221        RegDst <= 1'b0;
222        IorD <= 1'b0;
223        PCSrc <= 2'b00;
224        IRWrite <= 1'b0;
225        MemWrite <= 1'b0;
226        PCWrite <= 1'b0;
227        BranchEQ <= 1'b0;
228        BranchNE <= 1'b0;
229        RegWrite <= 1'b0;
230    end
231    S10: begin
232        RegDst <= 1'b0;
233        MemtoReg <= 1'b0;
234        RegWrite <= 1'b1;
235
236        IorD <= 1'b0;
237        PCSrc <= 2'b00;
238        ALUSrcB <= 2'b00;
239        ALUSrcA <= 1'b0;
240        IRWrite <= 1'b0;
241        MemWrite <= 1'b0;
242        PCWrite <= 1'b0;
243        BranchEQ <= 1'b0;
244        BranchNE <= 1'b0;
245        ALUOp <= 2'b00;
246    end
247    S11: begin
248        PCSrc <= 2'b10;
249        PCWrite <= 1'b1;
250
251        MemtoReg <= 1'b0;
252        RegDst <= 1'b0;
253        IorD <= 1'b0;
254        ALUSrcB <= 2'b00;
255        ALUSrcA <= 1'b0;
256        IRWrite <= 1'b0;
257        MemWrite <= 1'b0;
258        BranchEQ <= 1'b0;
259        BranchNE <= 1'b0;
260        RegWrite <= 1'b0;
261        ALUOp <= 2'b00;
262    end
263    S12: begin
264        ALUSrcA <= 1'b1;
265        ALUSrcB <= 2'b00;
266        ALUOp <= 2'b01;
267        PCSrc <= 2'b01;
268        BranchNE <= 1'b1;
269
270        MemtoReg <= 1'b0;
271        RegDst <= 1'b0;
272        IorD <= 1'b0;
273        IRWrite <= 1'b0;
274        MemWrite <= 1'b0;
275        PCWrite <= 1'b0;
276        BranchEQ <= 1'b0;
277        RegWrite <= 1'b0;
278    end
279 endcase
end

```

Figura 22: Atribuição dos sinais de controle para cada estado



Agora vamos alterar o testbench, para verificar a corretude dos sinais de controle. Primeiro precisamos criar uma variável para cada sinal de controle, assim como mais uma variável para o sinal esperado, que será lido nos vetores de teste.

```
logic MemtoReg, RegDst, IorD, ALUSrcA,
IRWrite, MemWrite, PCWrite, BranchEQ, BranchNE, RegWrite;
logic [1:0] PCSrc, ALUSrcB, ALUOp;

logic MemtoReg_esperado, RegDst_esperado, IorD_esperado, ALUSrcA_esperado,
IRWrite_esperado, MemWrite_esperado, PCWrite_esperado, BranchEQ_esperado,
BranchNE_esperado, RegWrite_esperado;
logic [1:0] PCSrc_esperado, ALUSrcB_esperado, ALUOp_esperado;
```

Figura 23: Criação das variáveis dos sinais de controle

Ao instanciar o main controller, precisamos passar essas variáveis que estão definidas como saída do modulo.

```
main_controller dut(clock, reset, opcode, state, MemtoReg, RegDst, IorD,
ALUSrcA, IRWrite, MemWrite, PCWrite, BranchEQ, BranchNE, RegWrite, PCSrc,
ALUSrcB, ALUOp);
```

Figura 24: Instanciando o Main Controller com os sinais de saída

Na leitura dos vetores de teste, precisamos ler também esses sinais.

```
always @(posedge clk) begin
    if(~rst) begin
        {clock, reset, opcode, state_esperado, MemtoReg_esperado, RegDst_esperado, IorD_esperado,
        PCSrc_esperado, ALUSrcB_esperado, ALUSrcA_esperado, IRWrite_esperado, MemWrite_esperado,
        PCWrite_esperado, BranchEQ_esperado, BranchNE_esperado, RegWrite_esperado, ALUOp_esperado} = vectors[counter];
    end
end
```

Figura 25: Leitura dos vetores de teste

E agora basta fazer as asserções para cada sinal de controle do main controller (Figura 26).

```

assert(MemtoReg == MemtoReg_esperado) else begin
    $error("MemtoReg = %b na linha %d", MemtoReg, counter+1);
end

assert(RegDst == RegDst_esperado) else begin
    $error("RegDst = %b na linha %d", RegDst, counter+1);
end

assert(IorD == IorD_esperado) else begin
    $error("IorD = %b na linha %d", IorD, counter+1);
end

assert(ALUSrcA == ALUSrcA_esperado) else begin
    $error("ALUSrcA = %b na linha %d", ALUSrcA, counter+1);
end

assert(IRWrite == IRWrite_esperado) else begin
    $error("IRWrite = %b na linha %d", IRWrite, counter+1);
end

assert(MemWrite == MemWrite_esperado) else begin
    $error("MemWrite = %b na linha %d", MemWrite, counter+1);
end

assert(PCWrite == PCWrite_esperado) else begin
    $error("PCWrite = %b na linha %d", PCWrite, counter+1);
end

assert(BranchEQ == BranchEQ_esperado) else begin
    $error("BranchEQ = %b na linha %d", BranchEQ, counter+1);
end

assert(BranchNE == BranchNE_esperado) else begin
    $error("BranchNE = %b na linha %d", BranchNE, counter+1);
end

assert(RegWrite == RegWrite_esperado) else begin
    $error("RegWrite = %b na linha %d", RegWrite, counter+1);
end

for(int i = 0; i < 2; i++) begin
    assert(PCSrc[i] == PCSrc_esperado[i]) else begin
        $error("Erro PCSrc na linha %d bit %d, saida = %b, (%b esperado)", counter+1, i, PCSrc[i], PCSrc_esperado[i]);
        errors = errors + 1;
    end
end

for(int i = 0; i < 2; i++) begin
    assert(ALUSrcB[i] == ALUSrcB_esperado[i]) else begin
        $error("Erro ALUSrcB na linha %d bit %d, saida = %b, (%b esperado)", counter+1, i, ALUSrcB[i], ALUSrcB_esperado[i]);
        errors = errors + 1;
    end
end

for(int i = 0; i < 2; i++) begin
    assert(ALUOp[i] == ALUOp_esperado[i]) else begin
        $error("Erro ALUOp na linha %d bit %d, saida = %b, (%b esperado)", counter+1, i, ALUOp[i], ALUOp_esperado[i]);
        errors = errors + 1;
    end
end

```

Figura 26: Asserções dos sinais de controle do Main Controller

### 2.1.2.3. Execução da Simulação

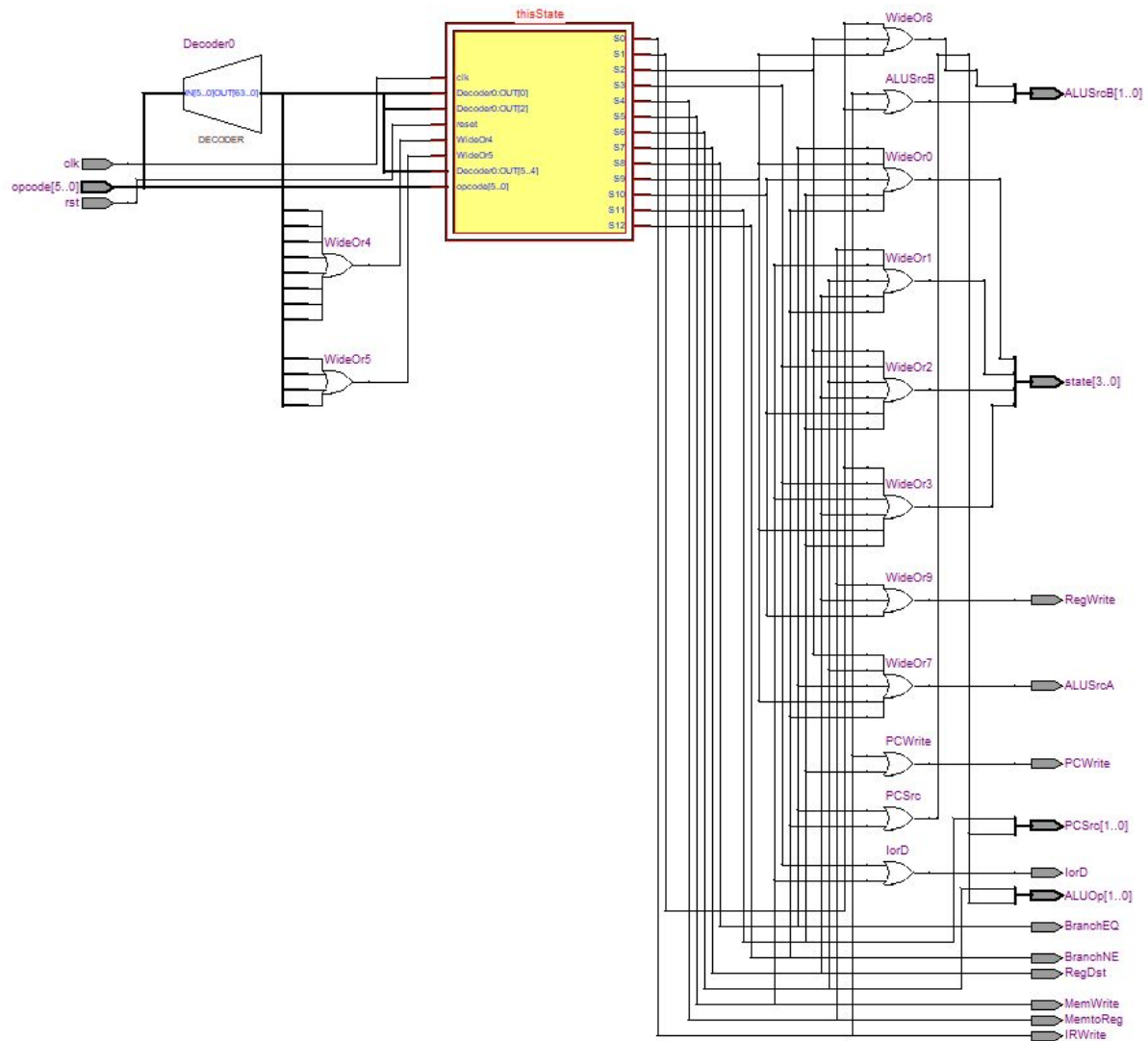


Figura 27: Visualização RTL

```
# | 0 | 0 | 001110 | 1010 | 0 | 0 | 0 | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 00 | OK
# | 1 | 0 | 001110 | 0000 | 0 | 0 | 0 | 00 | 01 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 00 | OK
# | 0 | 0 | 001010 | 0000 | 0 | 0 | 0 | 00 | 01 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 00 | OK
# | 1 | 0 | 001010 | 0001 | 0 | 0 | 0 | 00 | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | OK
# | 0 | 0 | 001010 | 0001 | 0 | 0 | 0 | 00 | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | OK
# | 1 | 0 | 001010 | 1001 | 0 | 0 | 0 | 00 | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | OK
# | 0 | 0 | 001010 | 1001 | 0 | 0 | 0 | 00 | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | OK
# | 1 | 0 | 001010 | 1010 | 0 | 0 | 0 | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 00 | OK
# | 0 | 0 | 001010 | 1010 | 0 | 0 | 0 | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 00 | OK
# | 1 | 0 | 001010 | 0000 | 0 | 0 | 0 | 00 | 01 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 00 | OK
# Testes Efetuados = 78
# Erros Encontrados = 0
```

Figura 28: Resultado de alguns casos de teste

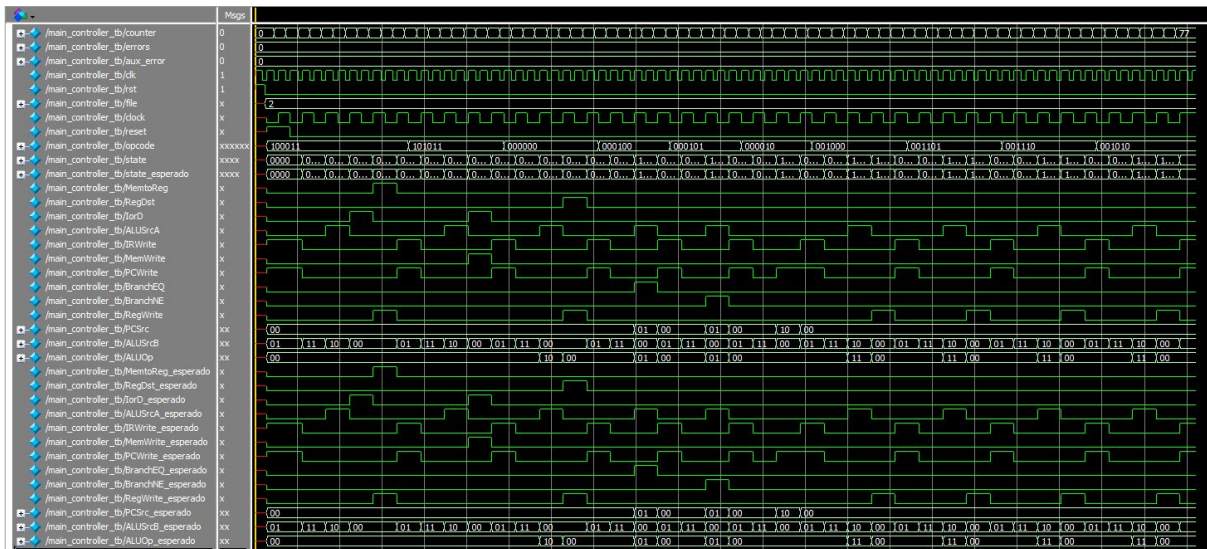


Figura 29: Visualização de Ondas

## 2.2. ALU Decoder

O ALU Decoder é responsável por gerar o sinal de controle da ULA. Para isso ele recebe como entrada o 'ALUOp' do Main Controller, e o 'opcode' e 'funct' da instrução. A primeira entrada que o ALUDecoder vai observar para definir a saída é o ALUOp, que funciona de acordo com a Tabela 1.

ALUOp	Significado
00	ADD
01	SUB
10	Olhar Funct
11	Olhar Opcode

Tabela 1: Significado OPCODE

Quando o OPCODE é '10' significa que se está realizando uma operação do tipo R, e portanto deve ser verificado o 'funct' para determinar qual operação será realizada na ULA. A Tabela 2 mostra a operação a ser realizada para cada funct.

Já quando o OPCODE é '11' é porque está sendo realizada uma operação do tipo I, e portanto o opcode deve ser comparado com os opcodes das operações do tipo I suportadas, para que seja determinado qual será a operação que a ULA executará. A Tabela 3 mostra os casos possíveis.

Funct	Operação
100000	ADD
100010	SUB
100100	AND
100101	OR
100111	NOR
100110	XOR
101010	SLT

Tabela 2: Operação Tipo-R por Funct

Opcode	Operação
001000	ADD
001101	OR
001110	XOR
001010	SLT

Tabela 3: Operação Tipo-I pelo Opcode

Com essas informações já é possível determinar a operação da ULA para todos os casos das operações suportadas nessa implementação. O sinal de controle da ULA é de 3 bits, a Tabela 4 mostra o sinal de controle para cada operação.

Operação	ALUControl
ADD	010
SUB	110
AND	000
OR	001
NOR	101
XOR	011
SLT	111

Tabela 4: ALUControl de cada operação

### 2.2.1. Golden Model

A implementação do Golden Model não tem dificuldade, basta apenas checar as variáveis de entrada da forma que foi descrita anteriormente. Primeiro atribuímos cada código ALUControl a uma constante com o nome da operação (Figura 30).

```

4      class AluFunctions:
5          AND = 0b000
6          OR = 0b001
7          ADD = 0b010
8          SUB = 0b110
9          SLT = 0b111
10         NOR = 0b101
11         XOR = 0b011

```

Figura 30: Constantes para representar operações da ULA

E então foi criada a função que faz todo o procedimento do ALU Decoder como descrito acima.

```

14 def alu_decoder(alu_op, opcode, funct):
15     alu_f = None
16
17     if alu_op == 0b00:
18         alu_f = AluFunctions.ADD
19
20     elif alu_op == 0b01:
21         alu_f = AluFunctions.SUB
22
23     elif alu_op == 0b10:
24         if funct == ADD:
25             alu_f = AluFunctions.ADD
26         elif funct == SUB:
27             alu_f = AluFunctions.SUB
28         elif funct == AND:
29             alu_f = AluFunctions.AND
30         elif funct == OR:
31             alu_f = AluFunctions.OR
32         elif funct == NOR:
33             alu_f = AluFunctions.NOR
34         elif funct == XOR:
35             alu_f = AluFunctions.XOR
36
37     elif funct == SLT:
38         alu_f = AluFunctions.SLT
39     else:
40         raise ValueError
41
42     elif alu_op == 0b11:
43         if opcode == ADDI:
44             alu_f = AluFunctions.ADD
45         elif opcode == ORI:
46             alu_f = AluFunctions.OR
47         elif opcode == XORI:
48             alu_f = AluFunctions.XOR
49         elif opcode == SLTI:
50             alu_f = AluFunctions.SLT
51         else:
52             raise ValueError
53     else:
54         raise ValueError
55
56     return alu_f

```

Figura 31: Implementação do ALU Decoder

Foi gerado um caso de teste para cada possível caso do ALU Decoder.



```

58 def create_alu_decoder_tvs():
59     R = 0b000000
60     X = 0b000000
61     operations = [(0b00, X, X), (0b01, X, X), (0b10, R, ADD), (0b10, R, SUB), (0b10, R, AND), (0b10, R, OR),
62                  (0b10, R, NOR), (0b10, R, XOR), (0b10, R, SLT), (0b11, ADDI, X), (0b11, ORI, X), (0b11, XORI, X),
63                  (0b11, SLTI, X)]
64
65     out = open('../simulation/modelsim/alu_decoder.tv', 'w')
66
67     for ALUOp, opcode, funct in operations:
68         ALUControl = alu_decoder(ALUOp, opcode, funct)
69         out.write("{:02b}_{:06b}_{:06b}_{:03b}\n".format(ALUOp, opcode, funct, ALUControl))
70
71     out.close()
72

```

Figura 32: Função que gerou os casos de teste do ALU Decoder

```

00_000000_000000_010
01_000000_000000_110
10_000000_100000_010
10_000000_100010_110
10_000000_100100_000
10_000000_100101_001
10_000000_100111_101
10_000000_100110_011
10_000000_101010_111
11_001000_000000_010
11_001101_000000_001
11_001110_000000_011
11_001010_000000_111

```

Figura 33: Casos de teste gerados

### 2.2.2. Implementação em System Verilog

A implementação do ALU Decoder em verilog também é bem direta, e não há dificuldades e é mostrado na Figura 34.

O testbench também é simples e direto, e é mostrado na Figura 35.



```

module alu_decoder(
    input logic [1:0] ALUOp,
    input logic [5:0] Opcode, Funct,
    output logic [2:0] ALUControl
);

localparam [2:0] AND = 3'b000,
OR = 3'b001, ADD = 3'b010,
SUB = 3'b110, SLT = 3'b111,
NOR = 3'b101, XOR = 3'b011;

always_comb begin
    case(ALUOp)
        2'b00: ALUControl <= ADD;
        2'b01: ALUControl <= SUB;
        2'b10: begin
            case(Funct)
                6'b100000: ALUControl <= ADD;
                6'b100010: ALUControl <= SUB;
                6'b100100: ALUControl <= AND;
                6'b100101: ALUControl <= OR;
                6'b100111: ALUControl <= NOR;
                6'b100110: ALUControl <= XOR;
                6'b101010: ALUControl <= SLT;
                default: ALUControl <= ADD;
            endcase
        end
        2'b11: begin
            case(Opcode)
                6'b001000: ALUControl <= ADD;
                6'b001101: ALUControl <= OR;
                6'b001110: ALUControl <= XOR;
                6'b001010: ALUControl <= SLT;
                default: ALUControl <= ADD;
            endcase
        end
    endcase
end

```

Figura 34: ALU Decoder em SV

```

1  `timescale 1ns/100ps
2  module alu_decoder_tb;
3
4      int counter, errors, aux_error;
5      logic clk, rst;
6      integer file;
7
8      logic [1:0] ALUOp;
9      logic [5:0] Opcode, Funct;
10     logic [2:0] ALUControl, ALUControl_esperado;
11
12     parameter max_vectors = 13;
13     logic [30:0] vectors[max_vectors];
14
15     alu_decoder dut(ALUOp, Opcode, Funct, ALUControl);
16
17     initial begin
18         counter = 0; errors = 0;
19         rst = 1'b1; #12; rst = 0;
20         clk=0;
21
22         if(~rst) begin
23             $readmemb("alu_decoder.tv", vectors);
24         end
25
26         file = $fopen("alu_decoder_out.txt");
27
28         $display("Iniciando Testbench");
29         $display("-----");
30         $display("| ALUOp | Opcode | Funct | ALUControl |");
31
32         $fwrite(file, "Iniciando Testbench");
33         $fwrite(file, "-----");
34         $fwrite(file, "| ALUOp | Opcode | Funct | ALUControl |");
35     end
36
37     always begin
38         clk = 1; #11;
39         clk = 0; #5;
40     end
41
42     always @(posedge clk) begin
43         if(~rst) begin
44             {ALUOp, Opcode, Funct, ALUControl_esperado} = vectors[counter];
45         end
46     end
47
48     always @(negedge clk) //Sempre (que o clock descer)
49     if(~rst) begin
50         if(ALUControl_esperado != 3'bx) begin
51             aux_error = errors;
52
53             for(int i = 0; i < 3; i++) begin
54                 assert (ALUControl[i] == ALUControl_esperado[i]) else begin
55                     //Mostra mensagem de erro se a saída do DUT for diferente da saída esperada
56                     $error("ALUControl[%d] = %b (%b esperado) {Linha %d}", i, ALUControl[i],
57                         ALUControl_esperado[i], counter+1);
58
59                     errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
60                 end
61             end
62
63             if(aux_error == errors) begin // Nao houve erro
64                 $display("| %b | %b | %b | %b | OK", ALUOp, Opcode, Funct, ALUControl);
65                 $fwrite(file, "| %b | %b | %b | %b | OK", ALUOp, Opcode, Funct, ALUControl);
66             end
67
68             if(counter+1 == max_vectors) begin
69                 $display("Testes Efetuados = %0d", counter+1);
70                 $display("Erros Encontrados = %0d", errors);
71                 $fwrite(file, "Testes Efetuados = %0d", counter+1);
72                 $fwrite(file, "Erros Encontrados = %0d", errors);
73                 #10
74                 $stop;
75             end
76         end
77         counter++; //Incrementa contador dos vetores de teste
78     end
79 endmodule

```

Figura 35: ALU Decoder Testbench

### 2.2.3. Execução da simulação

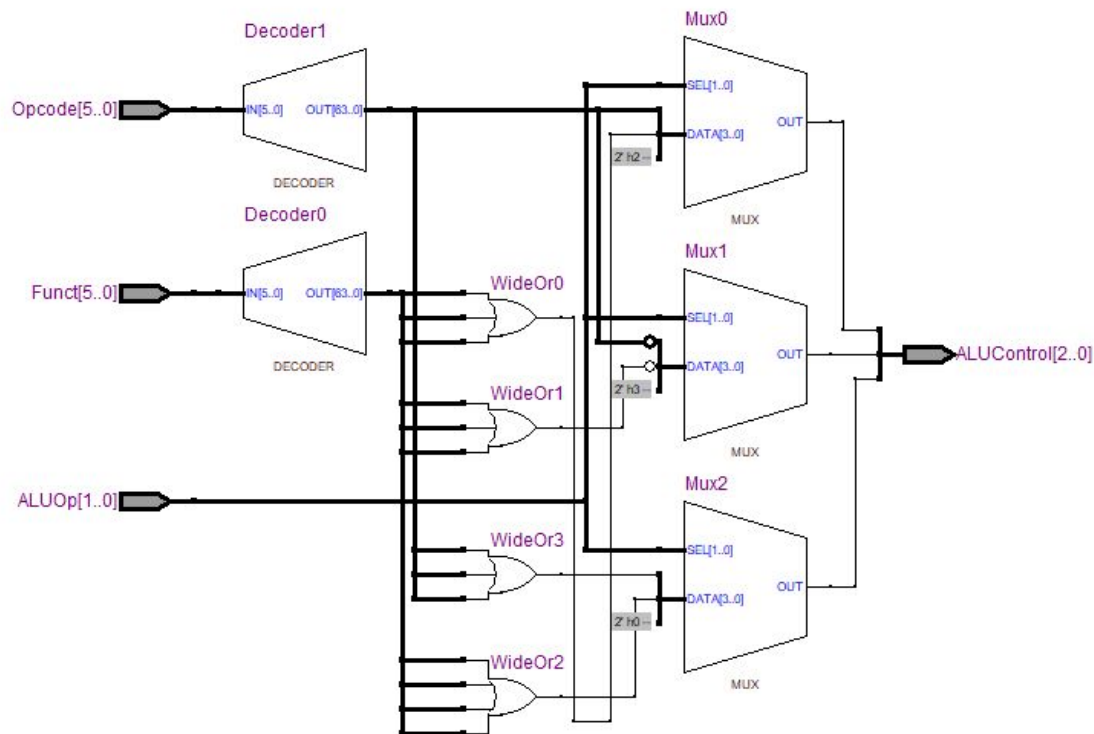


Figura 36: Visualização do RTL

```
# Iniciando Testbench
#
# | ALUOp | Opcode | Funct | ALUControl |
# | 00 | 000000 | 000000 | 010 | OK
# | 01 | 000000 | 000000 | 110 | OK
# | 10 | 000000 | 100000 | 010 | OK
# | 10 | 000000 | 100010 | 110 | OK
# | 10 | 000000 | 100100 | 000 | OK
# | 10 | 000000 | 100101 | 001 | OK
# | 10 | 000000 | 100111 | 101 | OK
# | 10 | 000000 | 100110 | 011 | OK
# | 10 | 000000 | 101010 | 111 | OK
# | 11 | 001000 | 000000 | 010 | OK
# | 11 | 001101 | 000000 | 001 | OK
# | 11 | 001110 | 000000 | 011 | OK
# | 11 | 001010 | 000000 | 111 | OK
# Testes Efetuados = 13
# Erros Encontrados = 0
```

Figura 37: Resultado da simulação

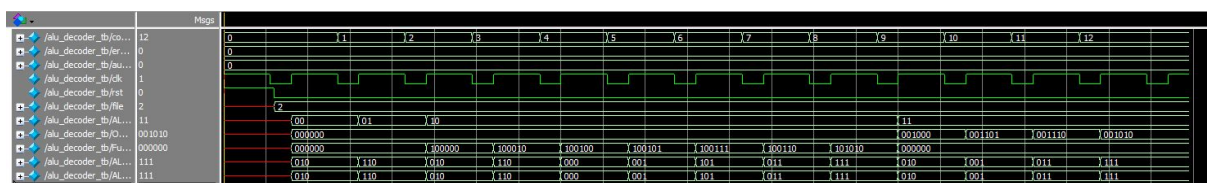


Figura 38: Visualização das ondas - simulação RTL

Durante a simulação Gate Level foi observado que o menor tempo em que o nível de clock pode permanecer alto para que ainda se tenha 0 erros, foi 11 unidades de tempo.

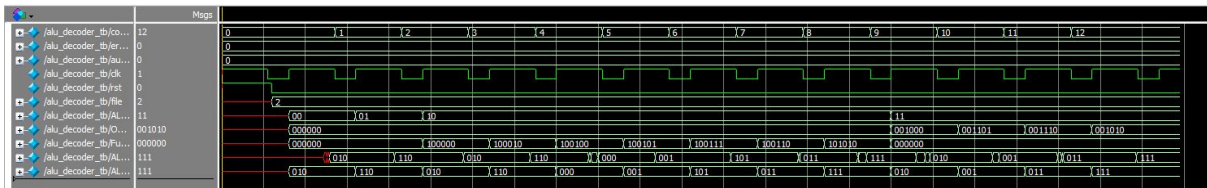


Figura 39: Visualização das ondas - Gate Level

## 2.3. Montando Unidade de Controle

Agora que temos o Main Controller e o ALU Decoder devidamente implementados e testados, basta agora conectá-los para formar a unidade de controle completa. Essa conexão é feita de acordo com o mostrado na Figura 1, com a adição da entrada Opcode no ALU Decoder.

### 2.3.1. Golden Model

O Golden Model basicamente cria uma instância do Main Controller, e delega a ela quase todas as responsabilidades. Apenas na troca de estado que é feito o calculo do ALUControl através do ALU Decoder.

```

6 class ControlUnit:
7     def __init__(self):
8         mc = MainController()
9         self.mc = mc
10        self.state = mc.state
11        self.nextState = mc.nextState
12        self.MemtoReg = mc.MemtoReg
13        self.RegDst = mc.RegDst
14        self.IorD = mc.IorD
15        self.PCSrc = mc.PCSrc
16        self.ALUSrcB = mc.ALUSrcB
17        self.ALUSrcA = mc.ALUSrcA
18        self.IRWrite = mc.IRWrite
19        self.MemWrite = mc.MemWrite
20        self.PCWrite = mc.PCWrite
21        self.BranchEQ = mc.BranchEQ
22        self.BranchNE = mc.BranchNE
23        self.RegWrite = mc.RegWrite
24        self.ALUControl = None
25
26    def get_next_state(self, Opcode, Funct):
27        self.mc.get_next_state(Opcode)
28
29        self.state = self.mc.state
30        self.nextState = self.mc.nextState
31        self.MemtoReg = self.mc.MemtoReg
32        self.RegDst = self.mc.RegDst
33        self.IorD = self.mc.IorD
34        self.PCSrc = self.mc.PCSrc
35        self.ALUSrcB = self.mc.ALUSrcB
36        self.ALUSrcA = self.mc.ALUSrcA
37        self.IRWrite = self.mc.IRWrite
38        self.MemWrite = self.mc.MemWrite
39        self.PCWrite = self.mc.PCWrite
40        self.BranchEQ = self.mc.BranchEQ
41        self.BranchNE = self.mc.BranchNE
42        self.RegWrite = self.mc.RegWrite
43        self.ALUControl = alu_decoder(self.mc.ALUOp, Opcode, Funct)
44
45    def go_next_state(self, rst):
46        self.mc.go_next_state(rst)
47        self.state = self.mc.state

```

Figura 40: Control Unit - Golden Model

A função que gera os vetores de teste é bem semelhante a função do main controller, com a diferença que precisamos ler o funct da operação, e escrevemos no arquivo o ALUControl, no lugar do ALUOp que tinha no main controller. A entrada desta função é um arquivo que contém o Opcode e Funct de todas as funções suportadas, há também um sinal de reset no início do arquivo.

```

50 def create_control_unit_tvs():
51     file = open('control_unit_input.txt', 'r')
52     out = open('../simulation/modelsim/control_unit.tv', 'w')
53     clk = 0
54     c = ControlUnit()
55     rst = 1
56     Opcode = None
57     Funct = None
58     finished = True
59
60     while True:
61         if finished:
62             finished = False
63             inp = file.readline()
64             if not inp or inp is None:
65                 break
66
67             rst = int(inp[0])
68             Opcode = int(inp[2:8], 2)
69             Funct = int(inp[9:15], 2)
70
71             if clk == 1 or rst == 1:
72                 c.go_next_state(rst)
73                 c.get_next_state(Opcode, Funct)
74                 if c.state == S0:
75                     finished = True
76
77             sout = "{:1b}_{:1b}_{:06b}_{:06b}_{:04b}_{:1b}_{:1b}_{:1b}_{:02b}_{:1b}_{:1b}_{:1b}_{:1b}_{:1b}_{:1b}" \
78                 "{:1b}_{:03b}\n".format(clk, rst, Opcode, Funct, c.state, c.MemtoReg, c.RegDst, c.IorD, c.PCSrc,
79                 c.ALUSrcB, c.ALUSrcA, c.IRWrite, c.MemWrite, c.PCWrite, c.BranchEQ, c.BranchNE,
80                 c.RegWrite, c.ALUControl)
81             print(sout, end="")
82             out.write(sout)
83
84             clk = 1 - clk
85
86     out.close()
87     file.close()

```

Figura 41: Função que gera os TVs da unidade de controle

```

1_100011_000000
1_100011_000000
0_100011_000000
0_101011_000000
0_000000_100000
0_000000_100010
0_000000_100100
0_000000_100101
0_000000_100111
0_000000_100110
0_000000_101010
0_000100_000000
0_000101_000000
0_000010_000000
0_001000_000000
0_001101_000000
0_001110_000000
0_001010_000000

```

Figura 42: Vetores de entrada do gerador de TVs



```

0_1_100011_000000_0000_0_0_0_00_01_0_1_0_1_0_0_0_010
1_1_100011_000000_0000_0_0_0_00_01_0_1_0_1_0_0_0_010
0_0_100011_000000_0000_0_0_0_00_01_0_1_0_1_0_0_0_010
1_0_100011_000000_0001_0_0_0_00_11_0_0_0_0_0_0_0_010
0_0_100011_000000_0001_0_0_0_00_11_0_0_0_0_0_0_0_010
1_0_100011_000000_0010_0_0_0_00_10_1_0_0_0_0_0_0_010
0_0_100011_000000_0010_0_0_0_00_10_1_0_0_0_0_0_0_010
1_0_100011_000000_0011_0_0_1_00_00_0_0_0_0_0_0_0_010
0_0_100011_000000_0011_0_0_1_00_00_0_0_0_0_0_0_0_010
1_0_100011_000000_0100_1_0_0_00_00_0_0_0_0_0_0_1_010
0_0_100011_000000_0100_1_0_0_00_00_0_0_0_0_0_0_1_010
1_0_100011_000000_0000_0_0_0_00_01_0_1_0_1_0_0_0_010
0_0_101011_000000_0000_0_0_0_00_01_0_1_0_1_0_0_0_010
1_0_101011_000000_0001_0_0_0_00_11_0_0_0_0_0_0_0_010
0_0_101011_000000_0001_0_0_0_00_11_0_0_0_0_0_0_0_010
1_0_101011_000000_0010_0_0_0_00_10_1_0_0_0_0_0_0_010
0_0_101011_000000_0010_0_0_0_00_10_1_0_0_0_0_0_0_010

```

Figura 43: Alguns vetores de teste gerados

### 2.3.2. Implementação em System Verilog

A implementação em System Verilog é ainda mais simples, bastando apenas fazer a conexão do Main Controller e do ALU Decoder da forma que foi vista na Figura 1.

```

module control_unit(
    input logic clk, rst,
    input logic [5:0] opcode, funct,
    output logic [3:0] state,
    output logic MemtoReg, RegDst, IorD, ALUSrcA,
    IRWrite, MemWrite, PCWrite, BranchEQ, BranchNE, RegWrite,
    output logic [1:0] PCSrc, ALUSrcB,
    output logic [2:0] ALUControl
);

    logic[1:0] ALUOp;
    main_controller mc(
        clk, rst, opcode, state, MemtoReg, RegDst, IorD,
        ALUSrcA, IRWrite, MemWrite, PCWrite, BranchEQ,
        BranchNE, RegWrite, PCSrc, ALUSrcB, ALUOp
    );
    alu_decoder ad(ALUOp, opcode, funct, ALUControl);

endmodule

```

Figura 44: Implementação da unidade de controle em SV

A implementação do testbench também ficou bastante semelhante ao testbench do Main Controller. A diferença está que agora é necessário ler o Funct dos vetores de teste, e avaliar a saída ALUControl no lugar da ALUOp.

```

57 always @(posedge clk) begin
58     if(~rst) begin
59         {clock, reset, opcode, funct, state_esperado, MemtoReg_esperado, RegDst_esperado, IorD_esperado,
60         PCSrc_esperado, ALUSrcB_esperado, ALUSrcA_esperado, IRWrite_esperado, MemWrite_esperado,
61         PCWrite_esperado, BranchEQ_esperado, BranchNE_esperado, RegWrite_esperado, ALUControl_esperado} = vectors[counter];
62     end
63 end

133 for(int i = 0; i < 3; i++) begin
134     assert (ALUControl[i] == ALUControl_esperado[i]) else begin
135         $error("Erro ALUControl na linha %d bit %d, saída = %b, (%b esperado)", counter+i, i, ALUControl[i], ALUControl_esperado[i]);
136         errors = errors + 1;
137     end
138 end

```

Figura 45: Mudanças no testbench do Control Unit em relação ao do Main Controller

### 2.3.3. Execução da simulação

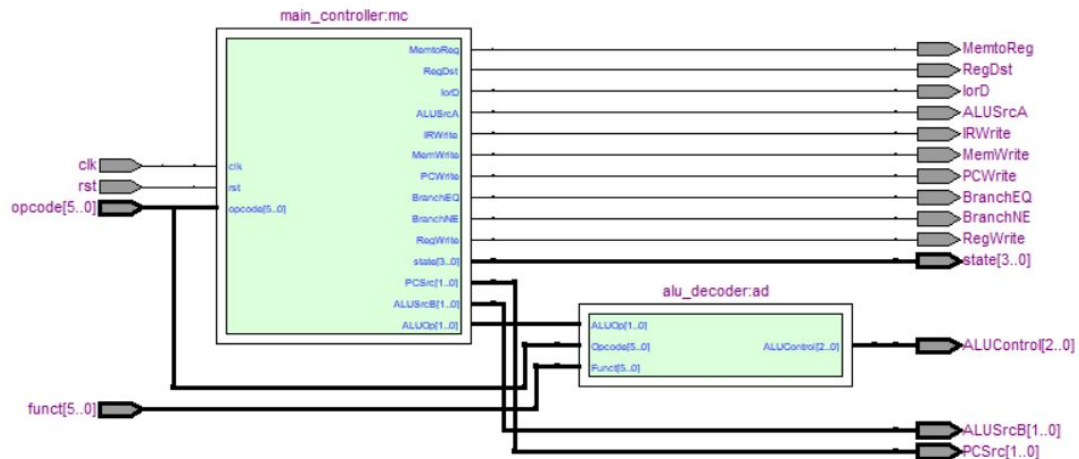


Figura 46: Visualização do RTL

```

# | 1 | 0 | 001110 | 000000 | 0000 | 0 | 0 | 0 | 00 | 01 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 010 | OK
# | 0 | 0 | 001010 | 000000 | 0000 | 0 | 0 | 0 | 00 | 01 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 010 | OK
# | 1 | 0 | 001010 | 000000 | 0001 | 0 | 0 | 0 | 00 | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 010 | OK
# | 0 | 0 | 001010 | 000000 | 0001 | 0 | 0 | 0 | 00 | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 010 | OK
# | 1 | 0 | 001010 | 000000 | 1001 | 0 | 0 | 0 | 00 | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 111 | OK
# | 0 | 0 | 001010 | 000000 | 1001 | 0 | 0 | 0 | 00 | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 111 | OK
# | 1 | 0 | 001010 | 000000 | 1010 | 0 | 0 | 0 | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 010 | OK
# | 0 | 0 | 001010 | 000000 | 1010 | 0 | 0 | 0 | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 010 | OK
# | 1 | 0 | 001010 | 000000 | 0000 | 0 | 0 | 0 | 00 | 01 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 010 | OK
# Testes Efetuados = 126
# Erros Encontrados = 0

```

Figura 47: Resultado dos últimos vetores de teste

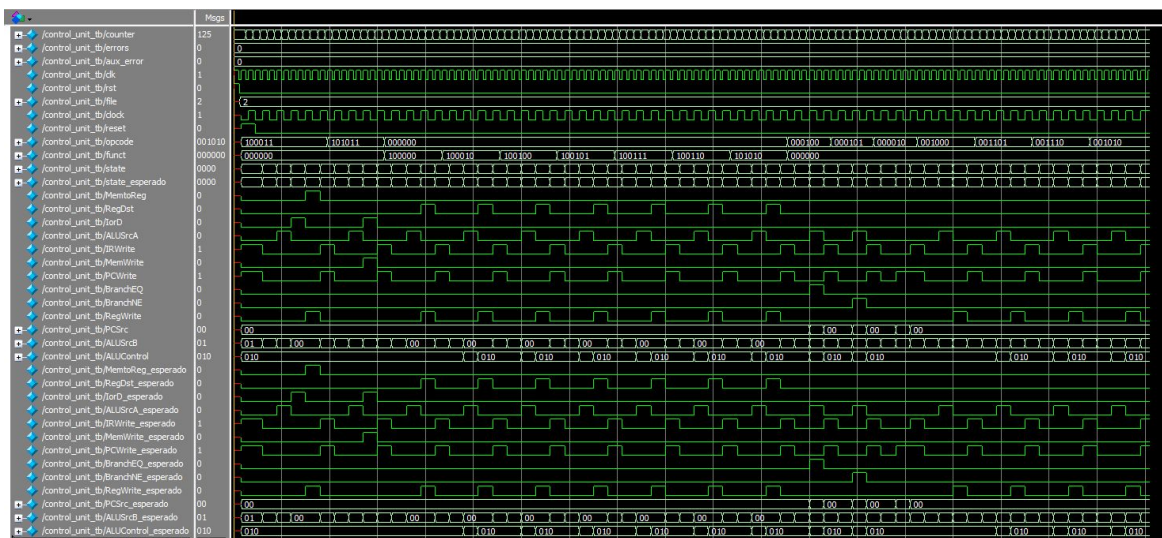


Figura 48: Visualização de onda



Com isso é concluída a implementação e teste da unidade de controle, agora que ela já está funcionando corretamente, podemos começar a implementação do Data Path para construir o processador completo.