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

1. Introdução

Este trabalho é um relatório da implementação de um processador MIPS32 Multiciclo. Será usado como base a arquitetura mostrada durante as aula de Concepção, 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.

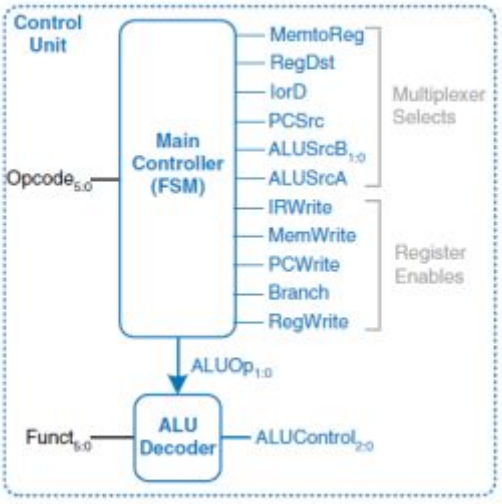


Figura 1: Estrutura da unidade de controle

O Main Controller tem como entrada o Opcode da instrução 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.

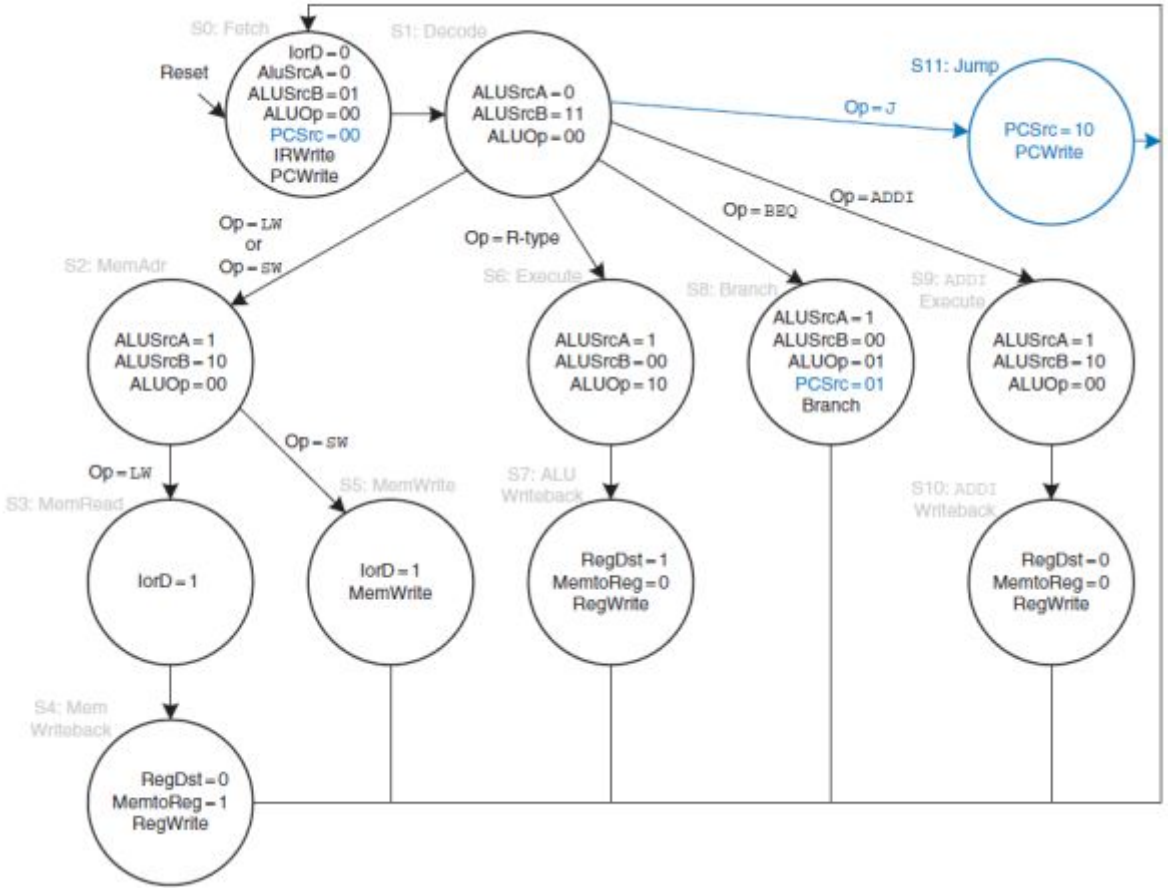


Figura 2: Máquina de estados **usada como base**

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 a 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.

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

```
# instructions opcode
LW = 0b100011
SW = 0b101011
R_TYPE = {0b000000}
BEQ = 0b000100
BNE = 0b000101
J = 0b000010
ADDI = 0b001000
ORI = 0b001101
XORI = 0b001110
SLTI = 0b001010
I_TYPE = {ADDI, ORI, XORI, SLTI}
```

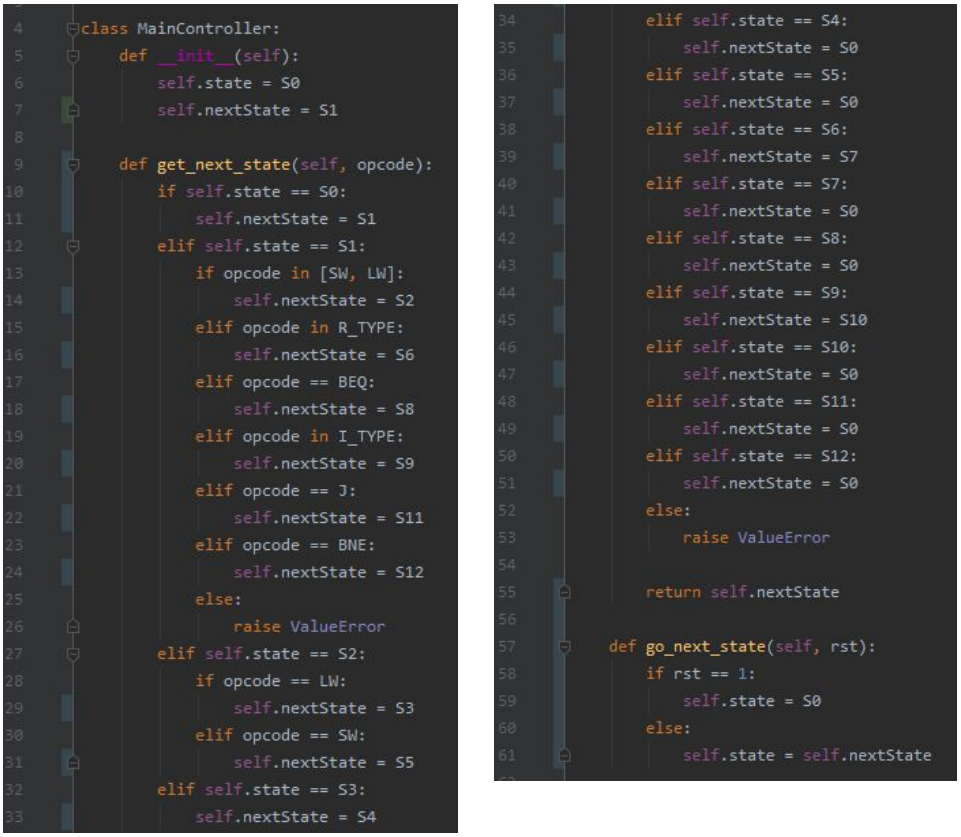
```
# R-Type Functs
ADD = 0b100000
SUB = 0b100010
AND = 0b100100
OR = 0b100101
NOR = 0b100111
XOR = 0b100110
SLT = 0b101010
```

Figura 3: definitions.py

Para gerar os vetores de teste, foi criado 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.

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

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.



```python
class MainController:
    def __init__(self):
        self.state = S0
        self.nextState = S1

    def get_next_state(self, opcode):
        if self.state == S0:
            self.nextState = S1
        elif self.state == S1:
            if opcode in [SW, LW]:
                self.nextState = S2
            elif opcode in R_TYPE:
                self.nextState = S6
            elif opcode == BEQ:
                self.nextState = S8
            elif opcode in I_TYPE:
                self.nextState = S9
            elif opcode == J:
                self.nextState = S11
            elif opcode == BNE:
                self.nextState = S12
            else:
                raise ValueError
        elif self.state == S2:
            if opcode == LW:
                self.nextState = S3
            elif opcode == SW:
                self.nextState = S5
        elif self.state == S3:
            self.nextState = S4
        elif self.state == S4:
            self.nextState = S0
        elif self.state == S5:
            self.nextState = S0
        elif self.state == S6:
            self.nextState = S7
        elif self.state == S7:
            self.nextState = S0
        elif self.state == S8:
            self.nextState = S0
        elif self.state == S9:
            self.nextState = S10
        elif self.state == S10:
            self.nextState = S0
        elif self.state == S11:
            self.nextState = S0
        elif self.state == S12:
            self.nextState = S0
        else:
            raise ValueError

        return self.nextState

    def go_next_state(self, rst):
        if rst == 1:
            self.state = S0
        else:
            self.state = self.nextState
```

Figura 4: Implementação do GM do Main Controller

```python
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

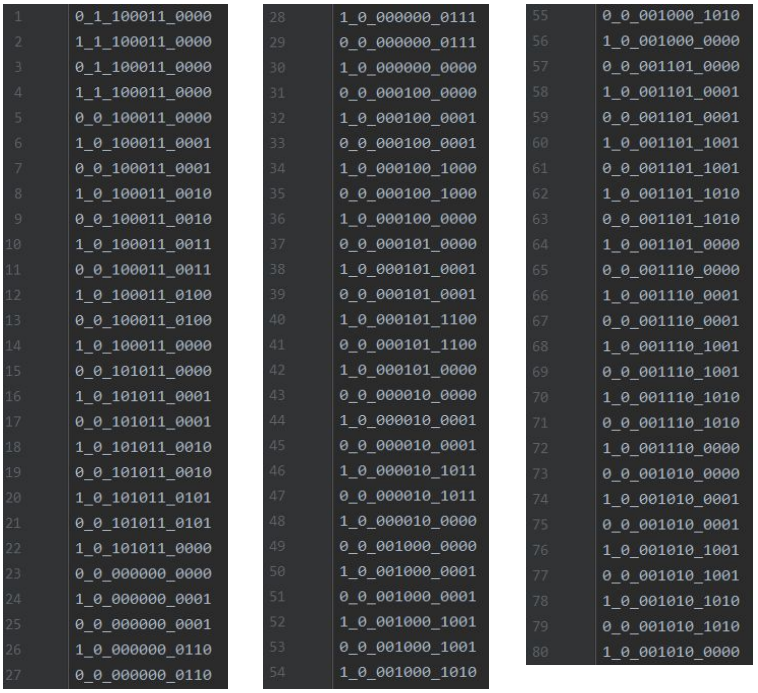


Figura 6: main_controller.tv

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.

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

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;
```

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.

```
20   State thisState, nextState;
21   assign state = thisState;
```

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).

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

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.

```
`timescale 1ns/100ps
module main_controller_tb;

    int counter, errors, aux_error;
    logic clk, rst;
    integer file;

    logic clock, reset;
    logic [5:0] opcode;
    logic [3:0] state, state_esperado;

    parameter max_vectors = 78;
    logic [11:0] vectors[max_vectors];

    main_controller dut(clock, reset, opcode, state);

    initial begin
        counter = 0; errors = 0;
        rst = 1'b1; #12; rst = 0;
        clk=0;

        if(~rst) begin
            $readmemb("main_controller.tv", vectors);
        end

        file = $fopen("main_controller_out.txt");

        $display("Iniciando Testbench");
        $display("---------------");
        $display("| clk | rst | opcode | state |");

        $fwrite(file, "Iniciando Testbench");
        $fwrite(file, "---------------");
        $fwrite(file, "| clk | rst | opcode | state |");
    end

    always begin
        clk = 1; #10;
        clk = 0; #5;
    end

    always @(posedge clk) begin
        if(~rst) begin
            {clock, reset, opcode, state_esperado} = vectors[counter];
        end
    end

    always @(negedge clk)   //Sempre (que o clock descer)
        if(~rst) begin
            if(state_esperado !== 6'bx) begin
                aux_error = errors;

                for(int i = 0; i < 6; i++) begin
                    assert (state[i] === state_esperado[i]) else begin
                        //Mostra mensagem de erro se a saída do DUT for diferente da saída esperada
                        $error("Erro S na linha %d bit %d, saida = %b, (%b esperado)", counter+1, i, state[i], state_esperado[i]);

                        errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
                    end
                end

                if(aux_error === errors) begin // Nao houve erro
                    $display("| %b |  %b   |  %b   |  %b  | OK", clock, reset, opcode, state);
                    $fwrite(file, "| %b |  %b   |  %b   |  %b  | OK", clock, reset, opcode, state);
                end

                if(counter+1 == max_vectors) begin
                    $display("Testes Efetuados  = %0d", counter+1);
                    $display("Erros Encontrados = %0d", errors);
                    $fwrite(file, "Testes Efetuados  = %0d", counter+1);
                    $fwrite(file, "Erros Encontrados = %0d", errors);
                    #10
                    $stop;
                end
            end
            counter++; //Incrementa contador dos vertores de teste
        end
endmodule
```

Figura 13: main_controller_tb.sv

2.1.1.3. Execução da Simulação

Abaixo temos o resultado da simulação feita com o ModelSim.

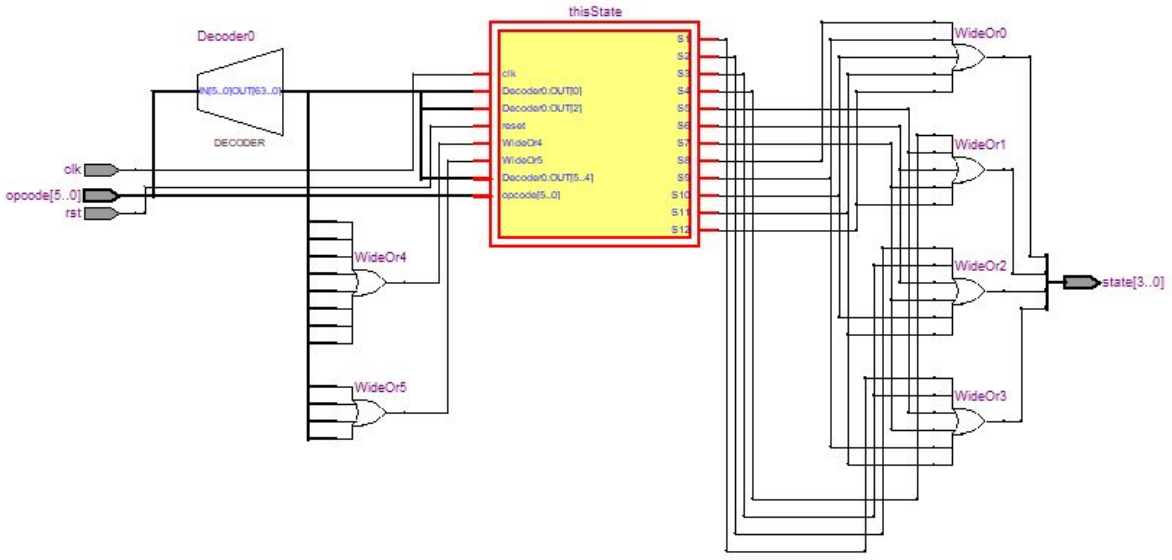


Figura 14: Visualização RTL do Main Controller

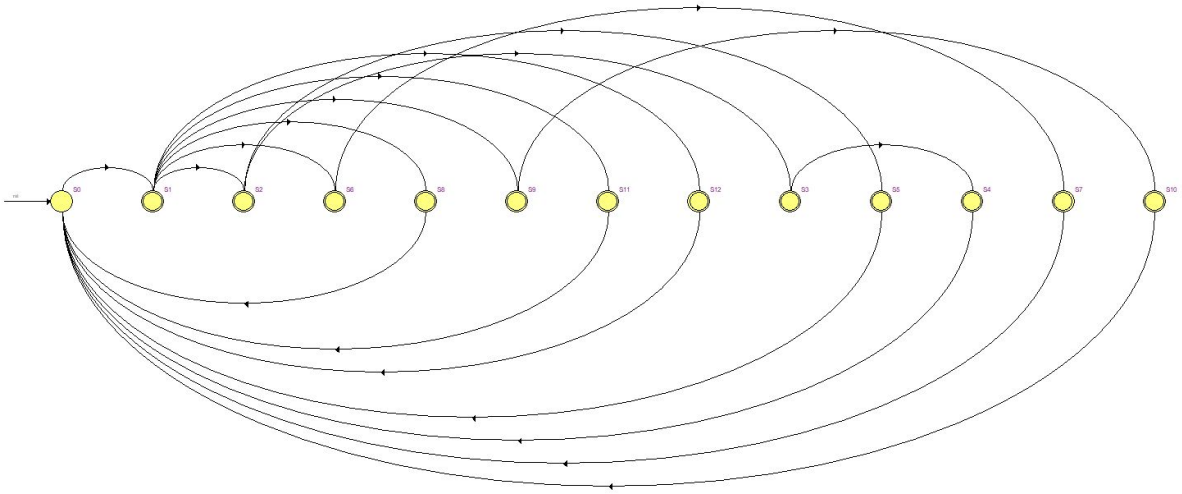


Figura 15: Diagrama de transição de estados

```
# |  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

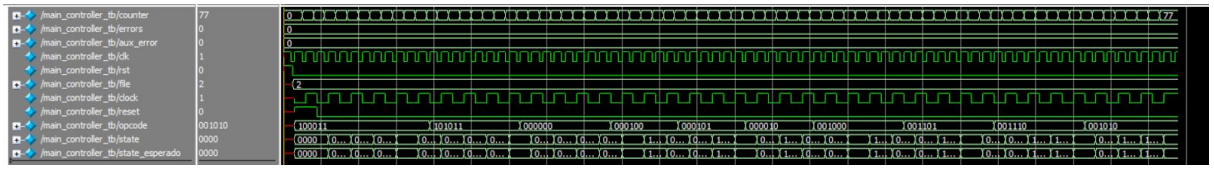


Figura 17: Visualização de onda da simulação RTL

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.

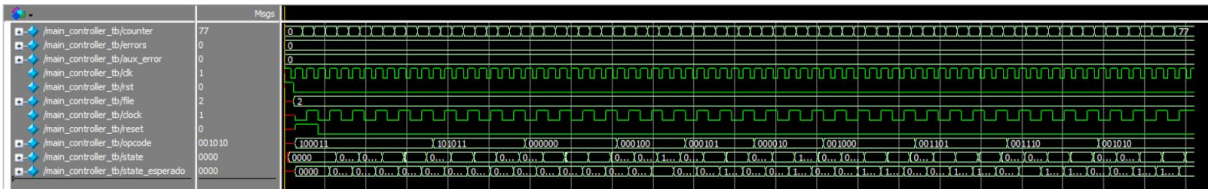


Figura 18: Visualização de onda da simulação Gate Level

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

```python
class MainController:
    def __init__(self):
        self.state = S0
        self.nextState = S1
        self.MemtoReg = 0b0
        self.RegDst = 0b0
        self.IorD = 0b0
        self.PCSrc = 0b00
        self.ALUSrcB = 0b01
        self.ALUSrcA = 0b0
        self.IRWrite = True
        self.MemWrite = False
        self.PCWrite = True
        self.BranchEQ = False
        self.BranchNE = False
        self.RegWrite = False
        self.ALUOp = 0b00

    def get_next_state(self, opcode):
        # atribui valor padrao aos sinais
        # para depois atribuir somente os
        # sinais do estado atual
        self.MemtoReg = 0b0
        self.RegDst = 0b0
        self.IorD = 0b0
        self.PCSrc = 0b00
        self.ALUSrcB = 0b01
        self.ALUSrcA = 0b0
        self.IRWrite = False
        self.MemWrite = False
        self.PCWrite = False
        self.BranchEQ = False
        self.BranchNE = False
        self.RegWrite = False
        self.ALUOp = 0b00

        if self.state == S0:
            self.IorD = 0b0
            self.ALUSrcA = 0b0
            self.ALUSrcB = 0b01
            self.ALUOp = 0b00
            self.PCSrc = 0b00
            self.IRWrite = True
            self.PCWrite = True

            self.nextState = S1

        elif self.state == S1:
            self.ALUSrcA = 0b0
            self.ALUSrcB = 0b11
            self.ALUOp = 0b00

            if opcode in [SW, LW]:
                self.nextState = S2
            elif opcode in R_TYPE:
                self.nextState = S6
            elif opcode == BEQ:
                self.nextState = S8
            elif opcode in I_TYPE:
                self.nextState = S9
            elif opcode == J:
                self.nextState = S11
            elif opcode == BNE:
                self.nextState = S12
            else:
                raise ValueError

        elif self.state == S2:
            self.ALUSrcA = 0b1
            self.ALUSrcB = 0b10
            self.ALUOp = 0b00

            if opcode == LW:
                self.nextState = S3
            elif opcode == SW:
                self.nextState = S5

        elif self.state == S3:
            self.IorD = 0b1

            self.nextState = S4

        elif self.state == S4:
            self.RegDst = 0b0
            self.MemtoReg = 0b1
            self.RegWrite = True

            self.nextState = S0

        elif self.state == S5:
            self.IorD = 0b1
            self.MemWrite = True

            self.nextState = S0

        elif self.state == S6:
            self.ALUSrcA = 0b1
            self.ALUSrcB = 0b00
            self.ALUOp = 0b10

            self.nextState = S7

        elif self.state == S7:
            self.RegDst = 0b1
            self.MemtoReg = 0b0
            self.RegWrite = True

            self.nextState = S0

        elif self.state == S8:
            self.ALUSrcA = 0b1
            self.ALUSrcB = 0b00
            self.ALUOp = 0b01
            self.PCSrc = 0b01
            self.BranchEQ = True

            self.nextState = S0

        elif self.state == S9:
            self.ALUSrcA = 0b1
            self.ALUSrcB = 0b10
            self.ALUOp = 0b11

            self.nextState = S10

        elif self.state == S10:
            self.RegDst = 0b0
            self.MemtoReg = 0b0
            self.RegWrite = True

            self.nextState = S0

        elif self.state == S11:
            self.PCSrc = 0b10
            self.PCWrite = True

            self.nextState = S0

        elif self.state == S12:
            self.ALUSrcA = 0b1
            self.ALUSrcB = 0b00
            self.ALUOp = 0b01
            self.PCSrc = 0b01
            self.BranchNE = True

            self.nextState = S0

        else:
            raise ValueError

        return self.nextState

    def go_next_state(self, rst):
        if rst == 1:
            self.state = S0
        else:
            self.state = self.nextState
```

Figura 20: Golden Model com sinais de controle

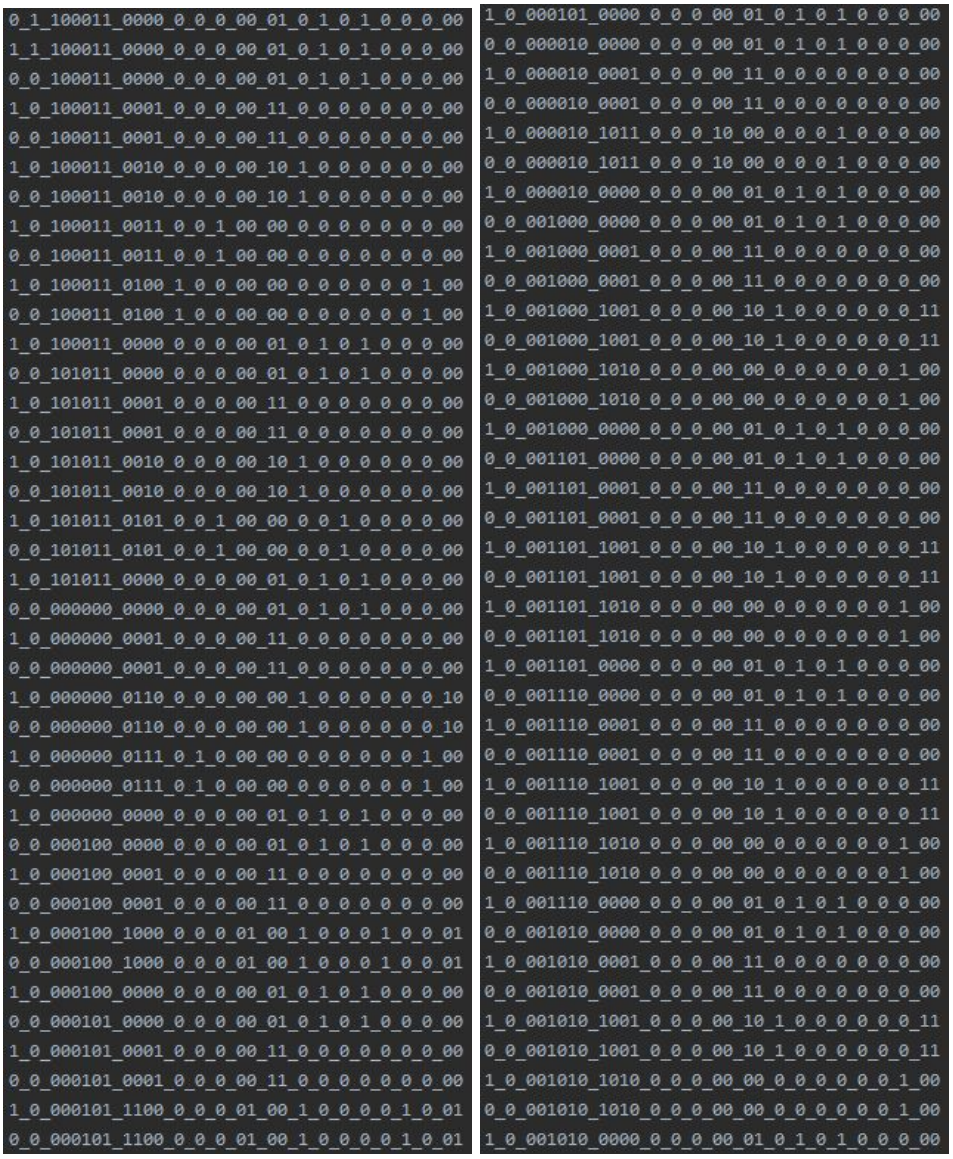
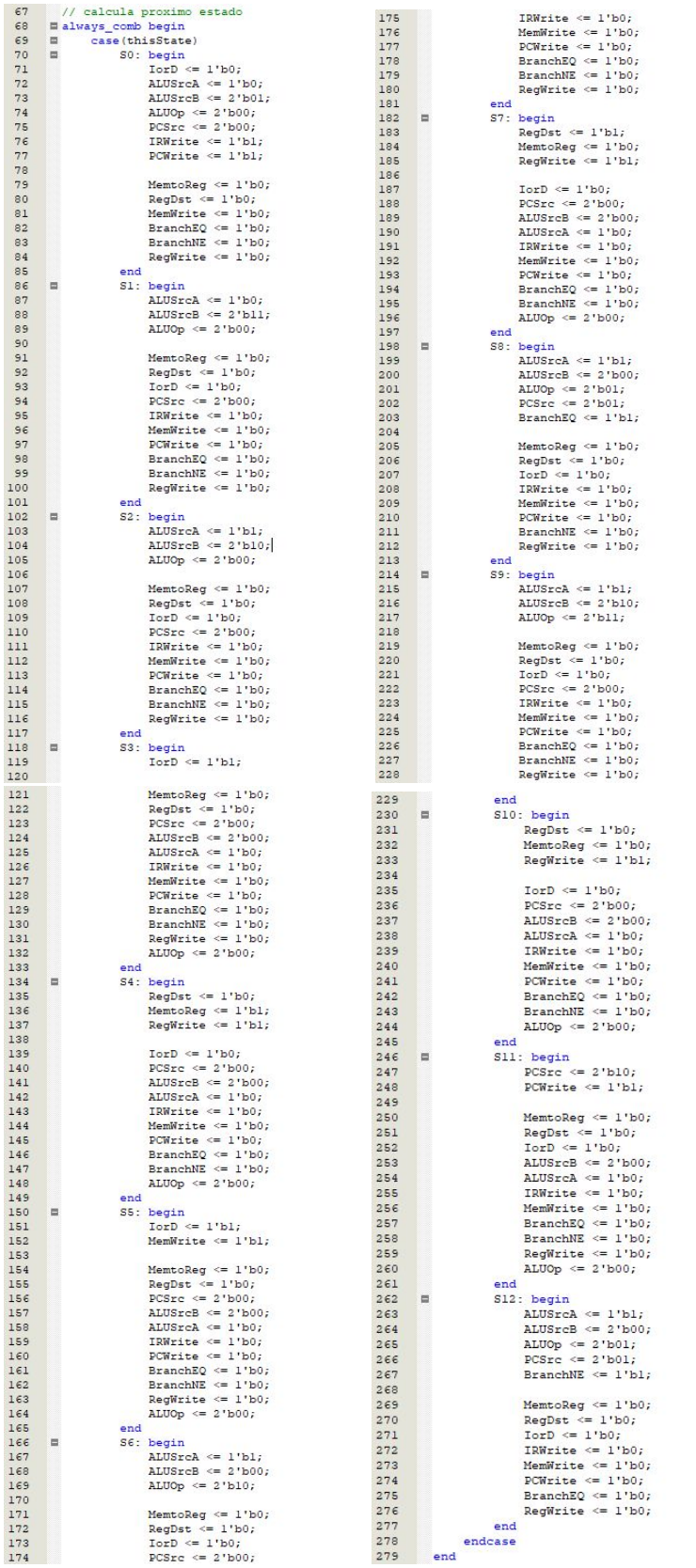


Figura 21: Vetores de teste com sinal de controle

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.



```
// calcula proximo estado
always_comb begin
    case(thisState)
        S0: begin
            IorD <= 1'b0;
            ALUSrcA <= 1'b0;
            ALUSrcB <= 2'b01;
            ALUOp <= 2'b00;
            PCSrc <= 2'b00;
            IRWrite <= 1'b1;
            PCWrite <= 1'b1;

            MemtoReg <= 1'b0;
            RegDst <= 1'b0;
            MemWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            RegWrite <= 1'b0;
        end
        S1: begin
            ALUSrcA <= 1'b0;
            ALUSrcB <= 2'b11;
            ALUOp <= 2'b00;

            MemtoReg <= 1'b0;
            RegDst <= 1'b0;
            IorD <= 1'b0;
            PCSrc <= 2'b00;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            RegWrite <= 1'b0;
        end
        S2: begin
            ALUSrcA <= 1'b1;
            ALUSrcB <= 2'b10;
            ALUOp <= 2'b00;

            MemtoReg <= 1'b0;
            RegDst <= 1'b0;
            IorD <= 1'b0;
            PCSrc <= 2'b00;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            RegWrite <= 1'b0;
        end
        S3: begin
            IorD <= 1'b1;

            MemtoReg <= 1'b0;
            RegDst <= 1'b0;
            PCSrc <= 2'b00;
            ALUSrcB <= 2'b00;
            ALUSrcA <= 1'b0;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            RegWrite <= 1'b0;
            ALUOp <= 2'b00;
        end
        S4: begin
            RegDst <= 1'b0;
            MemtoReg <= 1'b1;
            RegWrite <= 1'b1;

            IorD <= 1'b0;
            PCSrc <= 2'b00;
            ALUSrcB <= 2'b00;
            ALUSrcA <= 1'b0;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            ALUOp <= 2'b00;
        end
        S5: begin
            IorD <= 1'b1;
            MemWrite <= 1'b1;

            MemtoReg <= 1'b0;
            RegDst <= 1'b0;
            PCSrc <= 2'b00;
            ALUSrcB <= 2'b00;
            ALUSrcA <= 1'b0;
            IRWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            RegWrite <= 1'b0;
            ALUOp <= 2'b00;
        end
        S6: begin
            ALUSrcA <= 1'b1;
            ALUSrcB <= 2'b00;
            ALUOp <= 2'b10;

            MemtoReg <= 1'b0;
            RegDst <= 1'b0;
            IorD <= 1'b0;
            PCSrc <= 2'b00;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            RegWrite <= 1'b0;
        end
        S7: begin
            RegDst <= 1'b1;
            MemtoReg <= 1'b0;
            RegWrite <= 1'b1;

            IorD <= 1'b0;
            PCSrc <= 2'b00;
            ALUSrcB <= 2'b00;
            ALUSrcA <= 1'b0;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            ALUOp <= 2'b00;
        end
        S8: begin
            ALUSrcA <= 1'b1;
            ALUSrcB <= 2'b00;
            ALUOp <= 2'b01;
            PCSrc <= 2'b01;
            BranchEQ <= 1'b1;

            MemtoReg <= 1'b0;
            RegDst <= 1'b0;
            IorD <= 1'b0;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchNE <= 1'b0;
            RegWrite <= 1'b0;
        end
        S9: begin
            ALUSrcA <= 1'b1;
            ALUSrcB <= 2'b10;
            ALUOp <= 2'b11;

            MemtoReg <= 1'b0;
            RegDst <= 1'b0;
            IorD <= 1'b0;
            PCSrc <= 2'b00;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            RegWrite <= 1'b0;
        end
        S10: begin
            RegDst <= 1'b0;
            MemtoReg <= 1'b0;
            RegWrite <= 1'b1;

            IorD <= 1'b0;
            PCSrc <= 2'b00;
            ALUSrcB <= 2'b00;
            ALUSrcA <= 1'b0;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            ALUOp <= 2'b00;
        end
        S11: begin
            PCSrc <= 2'b10;
            PCWrite <= 1'b1;

            MemtoReg <= 1'b0;
            RegDst <= 1'b0;
            IorD <= 1'b0;
            ALUSrcB <= 2'b00;
            ALUSrcA <= 1'b0;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            BranchEQ <= 1'b0;
            BranchNE <= 1'b0;
            RegWrite <= 1'b0;
            ALUOp <= 2'b00;
        end
        S12: begin
            ALUSrcA <= 1'b1;
            ALUSrcB <= 2'b00;
            ALUOp <= 2'b01;
            PCSrc <= 2'b01;
            BranchNE <= 1'b1;

            MemtoReg <= 1'b0;
            RegDst <= 1'b0;
            IorD <= 1'b0;
            IRWrite <= 1'b0;
            MemWrite <= 1'b0;
            PCWrite <= 1'b0;
            BranchEQ <= 1'b0;
            RegWrite <= 1'b0;
        end
    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 saida 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

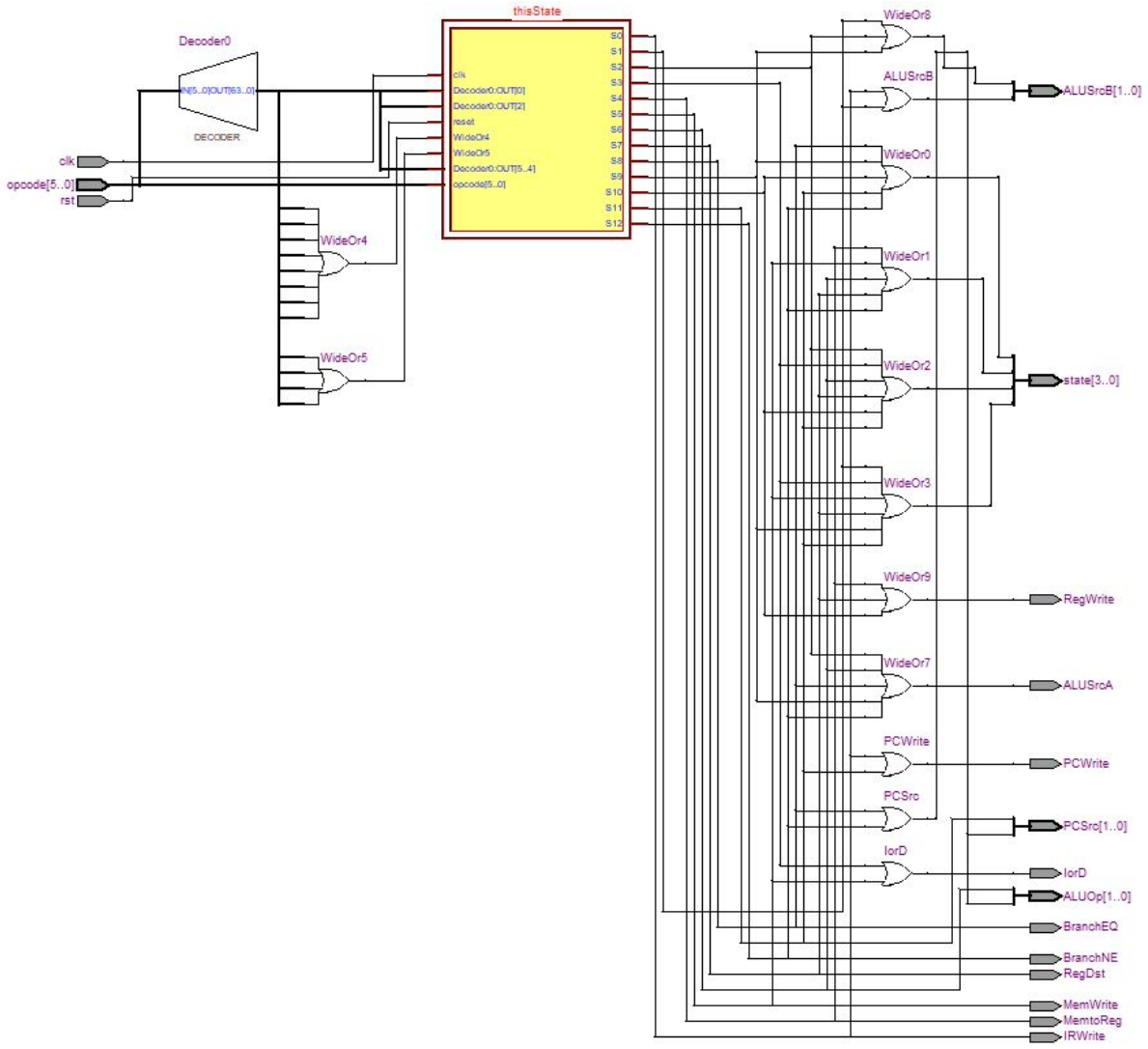


Figura 27: Visualização RTL

```
#  |  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

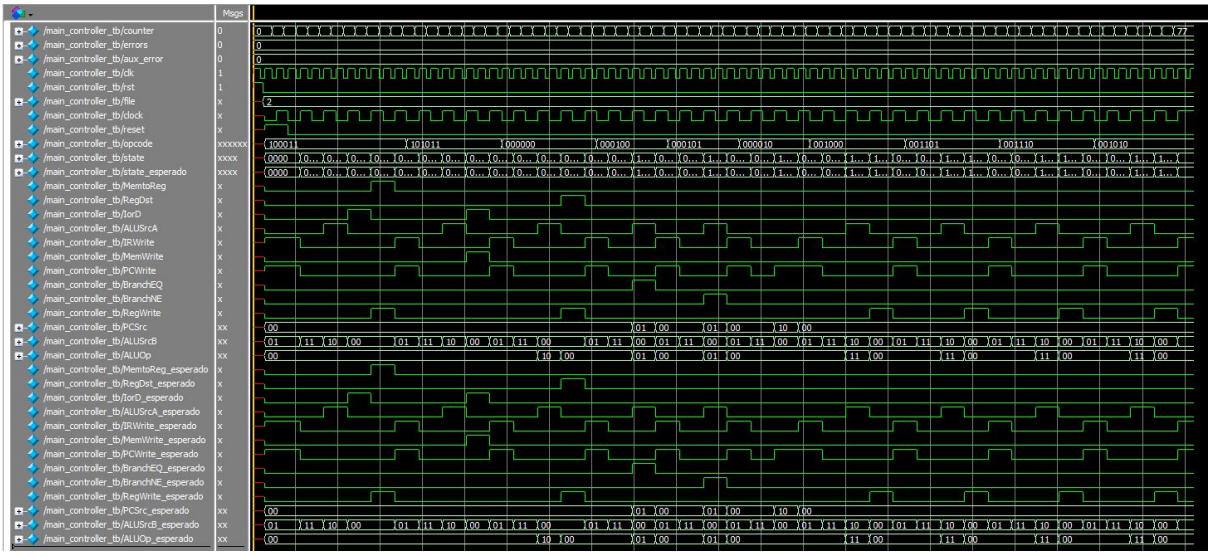


Figura 29: Visualização de Ondas

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).

```
class AluFunctions:
    AND = 0b000
    OR = 0b001
    ADD = 0b010
    SUB = 0b110
    SLT = 0b111
    NOR = 0b101
    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.

```
def alu_decoder(alu_op, opcode, funct):
    alu_f = None

    if alu_op == 0b00:
        alu_f = AluFunctions.ADD

    elif alu_op == 0b01:
        alu_f = AluFunctions.SUB

    elif alu_op == 0b10:
        if funct == ADD:
            alu_f = AluFunctions.ADD
        elif funct == SUB:
            alu_f = AluFunctions.SUB
        elif funct == AND:
            alu_f = AluFunctions.AND
        elif funct == OR:
            alu_f = AluFunctions.OR
        elif funct == NOR:
            alu_f = AluFunctions.NOR
        elif funct == XOR:
            alu_f = AluFunctions.XOR
        elif funct == SLT:
            alu_f = AluFunctions.SLT
        else:
            raise ValueError

    elif alu_op == 0b11:
        if opcode == ADDI:
            alu_f = AluFunctions.ADD
        elif opcode == ORI:
            alu_f = AluFunctions.OR
        elif opcode == XORI:
            alu_f = AluFunctions.XOR
        elif opcode == SLTI:
            alu_f = AluFunctions.SLT
        else:
            raise ValueError
    else:
        raise ValueError

    return alu_f
```

Figura 31: Implementação do ALU Decoder

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

```python
def create_alu_decoder_tvs():
    R = 0b000000
    X = 0b000000
    operations = [(0b00, X, X), (0b01, X, X), (0b10, R, ADD), (0b10, R, SUB), (0b10, R, AND), (0b10, R, OR),
                  (0b10, R, NOR), (0b10, R, XOR), (0b10, R, SLT), (0b11, ADDI, X), (0b11, ORI, X), (0b11, XORI, X),
                  (0b11, SLTI, X)]

    out = open('../simulation/modelsim/alu_decoder.tv', 'w')

    for ALUOp, opcode, funct in operations:
        ALUControl = alu_decoder(ALUOp, opcode, funct)
        out.write("{:02b}_{:06b}_{:06b}_{:03b}\n".format(ALUOp, opcode, funct, ALUControl))

    out.close()
```

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

```
`timescale 1ns/100ps
module alu_decoder_tb;

    int counter, errors, aux_error;
    logic clk, rst;
    integer file;

    logic [1:0] ALUOp;
    logic [5:0] Opcode, Funct;
    logic [2:0] ALUControl, ALUControl_esperado;

    parameter max_vectors = 13;
    logic [30:0] vectors[max_vectors];

    alu_decoder dut(ALUOp, Opcode, Funct, ALUControl);

    initial begin
        counter = 0; errors = 0;
        rst = 1'b1; #12; rst = 0;
        clk=0;

        if(~rst) begin
            $readmemb("alu_decoder.tv", vectors);
        end

        file = $fopen("alu_decoder_out.txt");

        $display("Iniciando Testbench");
        $display("---------------");
        $display("| ALUOp | Opcode | Funct  | ALUControl |");

        $fwrite(file, "Iniciando Testbench");
        $fwrite(file, "---------------");
        $fwrite(file, "| ALUOp | Opcode | Funct  | ALUControl |");
    end

    always begin
        clk = 1; #11;
        clk = 0; #5;
    end

    always @(posedge clk) begin
        if(~rst) begin
            {ALUOp, Opcode, Funct, ALUControl_esperado} = vectors[counter];
        end
    end

    always @(negedge clk)   //Sempre (que o clock descer)
        if(~rst) begin
            if(ALUControl_esperado !== 3'bx) begin
                aux_error = errors;

                for(int i = 0; i < 3; i++) begin
                    assert (ALUControl[i] === ALUControl_esperado[i]) else begin
                        //Mostra mensagem de erro se a saída do DUT for diferente da saída esperada
                        $error("ALUControl[%d] = %b (%b esperado) (Linha %d)", i, ALUControl[i],
                                                                    ALUControl_esperado[i], counter+1);

                        errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
                    end
                end

                if(aux_error === errors) begin // Nao houve erro
                    $display("|  %b    | %b | %b |    %b     | OK", ALUOp, Opcode, Funct, ALUControl);
                    $fwrite(file, "|  %b    | %b | %b |    %b     | OK", ALUOp, Opcode, Funct, ALUControl);
                end

                if(counter+1 == max_vectors) begin
                    $display("Testes Efetuados  = %0d", counter+1);
                    $display("Erros Encontrados = %0d", errors);
                    $fwrite(file, "Testes Efetuados  = %0d", counter+1);
                    $fwrite(file, "Erros Encontrados = %0d", errors);
                    #10
                    $stop;
                end
            end
            counter++; //Incrementa contador dos vertores de teste
        end
endmodule
```

Figura 35: ALU Decoder Testbench

2.2.3. Execução da simulação

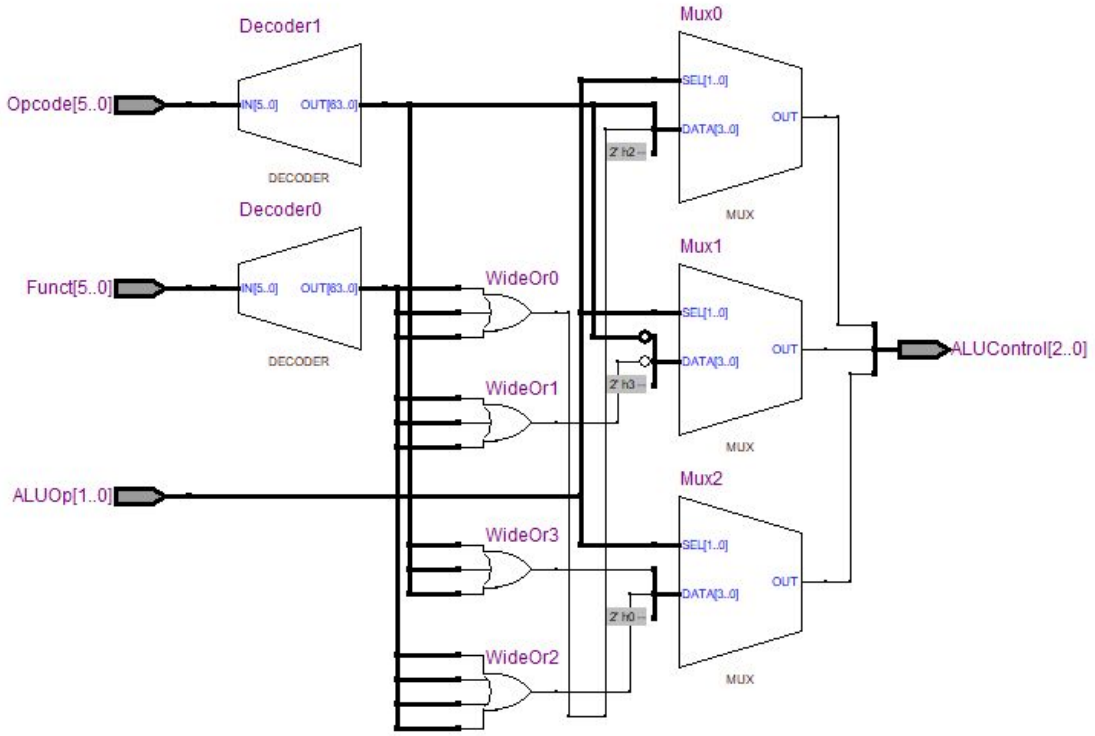


Figura 36: Visualização do RTL

```
# 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

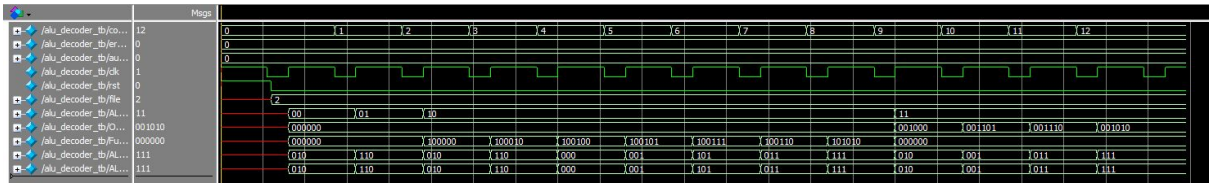


Figura 38: Visualização das ondas - simulação RTL

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.

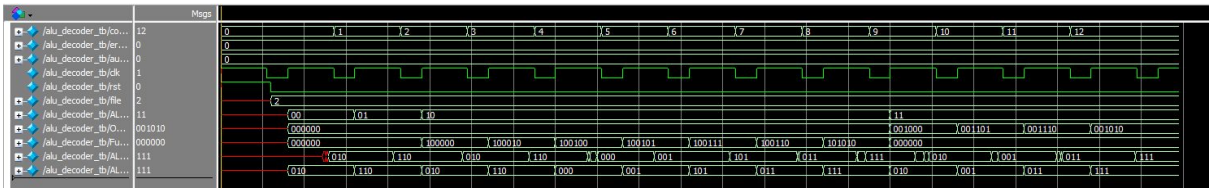


Figura 39: Visualização das ondas - Gate Level

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.

```python
class ControlUnit:
    def __init__(self):
        mc = MainController()
        self.mc = mc
        self.state = mc.state
        self.nextState = mc.nextState
        self.MemtoReg = mc.MemtoReg
        self.RegDst = mc.RegDst
        self.IorD = mc.IorD
        self.PCSrc = mc.PCSrc
        self.ALUSrcB = mc.ALUSrcB
        self.ALUSrcA = mc.ALUSrcA
        self.IRWrite = mc.IRWrite
        self.MemWrite = mc.MemWrite
        self.PCWrite = mc.PCWrite
        self.BranchEQ = mc.BranchEQ
        self.BranchNE = mc.BranchNE
        self.RegWrite = mc.RegWrite
        self.ALUControl = None

    def get_next_state(self, Opcode, Funct):
        self.mc.get_next_state(Opcode)

        self.state = self.mc.state
        self.nextState = self.mc.nextState
        self.MemtoReg = self.mc.MemtoReg
        self.RegDst = self.mc.RegDst
        self.IorD = self.mc.IorD
        self.PCSrc = self.mc.PCSrc
        self.ALUSrcB = self.mc.ALUSrcB
        self.ALUSrcA = self.mc.ALUSrcA
        self.IRWrite = self.mc.IRWrite
        self.MemWrite = self.mc.MemWrite
        self.PCWrite = self.mc.PCWrite
        self.BranchEQ = self.mc.BranchEQ
        self.BranchNE = self.mc.BranchNE
        self.RegWrite = self.mc.RegWrite
        self.ALUControl = alu_decoder(self.mc.ALUOp, Opcode, Funct)

    def go_next_state(self, rst):
        self.mc.go_next_state(rst)
        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.

```
def create_control_unit_tvs():
    file = open('control_unit_input.txt', 'r')
    out = open('../simulation/modelsim/control_unit.tv', 'w')
    clk = 0
    c = ControlUnit()
    rst = 1
    Opcode = None
    Funct = 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)
            Funct = int(inp[9:15], 2)

        if clk == 1 or rst == 1:
            c.go_next_state(rst)
            c.get_next_state(Opcode, Funct)
            if c.state == S0:
                finished = True

        sout = "{:1b}_{:1b}_{:06b}_{:06b}_{:04b}_{:1b}_{:1b}_{:1b}_{:02b}_{:02b}_{:1b}_{:1b}_{:1b}_{:1b}_{:1b}" \
               "_{:1b}_{:03b}\n".format(clk, rst, Opcode, Funct, c.state, c.MemtoReg, c.RegDst, c.IorD, c.PCSrc,
                                        c.ALUSrcB, c.ALUSrcA, c.IRWrite, c.MemWrite, c.PCWrite, c.BranchEQ, c.BranchNE,
                                        c.RegWrite, c.ALUControl)
        print(sout, end="")
        out.write(sout)

        clk = 1 - clk

    out.close()
    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 Controler 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, saida = %b, (%b esperado)", counter+1, 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

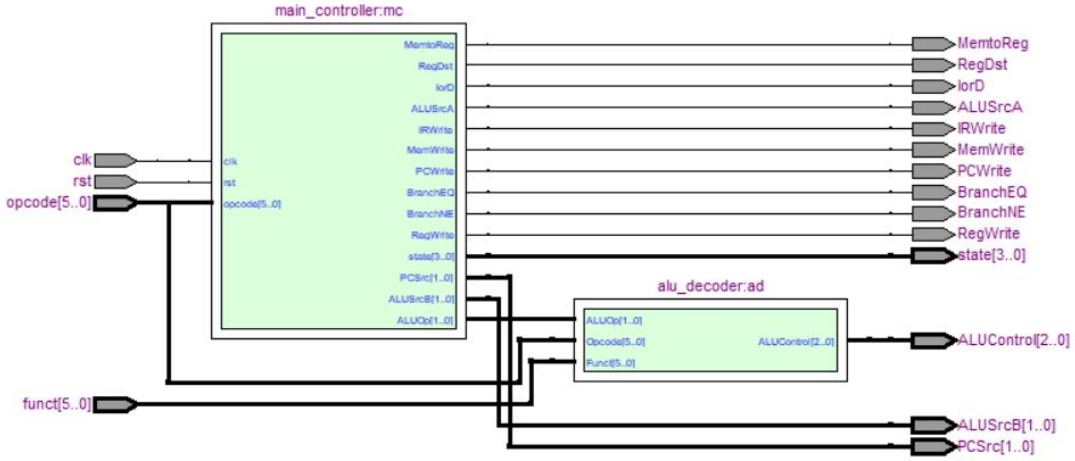


Figura 46: Visualização do RTL

```
#  |  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

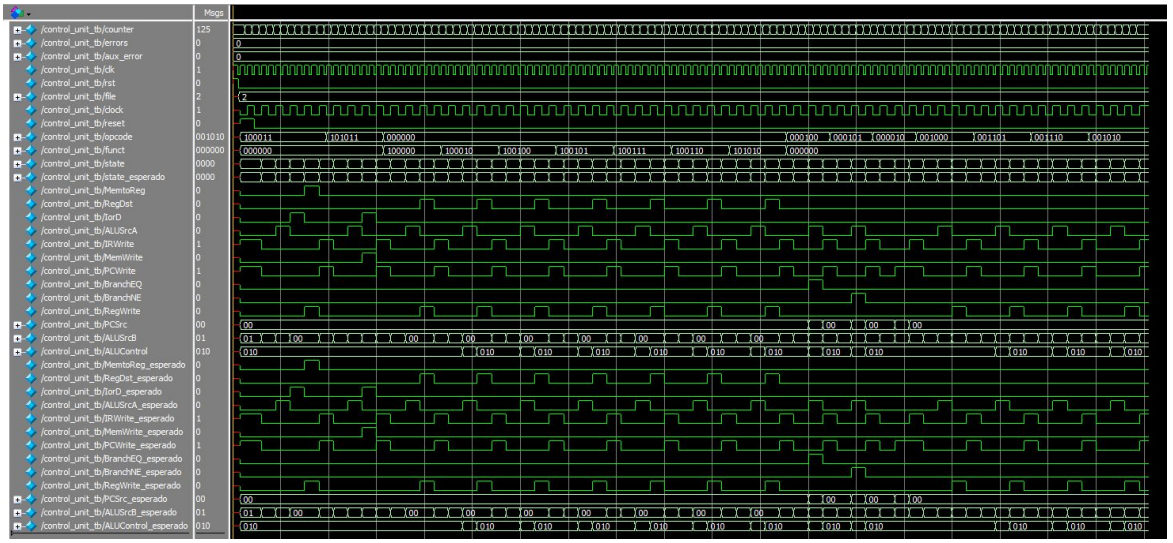


Figura 48: Visualização de onda

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.