



**Universidade Federal da Paraíba
Centro de Informática**

Concepção Estruturada de Circuitos Integrados

Sarah Andrade Toscano de Carvalho - Matrícula: 20170022975

João Pessoa, 2020.

Sumário

MIPS32 Multiciclo.....	6
MIPS32.....	7
Arquitetura RISC.....	7
Conjunto de Instruções.....	7
Arquitetura Multiciclo	14
Implementação do modelo	14
Anexo I: Componentes Datapath	22
Flip-Flop com Reset (flopr).....	23
Descrição em System Verilog	23
Visão RTL	23
Golden Model.....	24
Vetores de Teste	24
Arquivo de TestBench	25
Simulação	26
Flip-Flop com Enable e Rst (flopenr 1 bit)	28
Descrição em System Verilog	28
Visão RTL	28
Golden Model.....	28
Vetores de Teste	29
Arquivo de TestBench	30
Simulação	31
Flip-Flop com Enable e Rst (flopenr 32 bits)	33
Descrição em SystemVerilog	33
Visão RTL	33
Mux: 2-1 (1 bit)	35
Descrição em System Verilog	35
Visão RTL	35
Golden Model.....	36
Vetores de Teste	36
Arquivo de TestBench	37
Simulação	38
Mux: 2-1 (5 bits).....	39
Descrição em System Verilog	39
Mux: 2-1 (32 bits)	40

Descrição em System Verilog	40
Mux: 3-1 (32 bits)	41
Descrição em System Verilog	41
Visão RTL	41
Golden Model.....	42
Vetores de Teste	42
Simulação	42
Mux: 4-1 (1 bit)	44
Descrição em System Verilog	44
Visão RTL	44
Golden Model.....	44
Vetores de Teste	45
Arquivo de TestBench	47
Simulação	48
Mux: 4-1 (32 bits)	50
Descrição em System Verilog	50
Mux: 8-1 (1 bit)	51
Descrição em System Verilog	51
Visão RTL	51
Golden Model.....	52
Vetores de Ouro	52
Simulação	53
Mux: 32-1 (1 bit)	55
Descrição em System Verilog	55
Visão RTL	55
Golden Model.....	56
Vetores de Teste	56
TestBench.....	57
Simulação	58
Decodificador: 5x32.....	59
Descrição em System Verilog	59
Visão RTL	60
Golden Model.....	60
Vetores de teste.....	62
Arquivo de TestBench	62
Simulação	64

Banco de Registradores (1 bit)	66
Descrição em System Verilog	67
Visão RTL	68
Golden Model.....	69
Vetores de Teste	70
TestBench.....	72
Simulação	73
Banco de Registradores (32 bits)	74
Descrição em System Verilog	74
Deslocador de Bits (32 <<2)	75
Descrição em System Verilog	75
Visão RTL	75
Golden Model.....	75
Vetores de Ouro	75
TestBench.....	77
Simulação	77
Deslocador de Bits (26 <<2)	78
Descrição em System Verilog	78
Visão RTL	78
Extensor de bits - 16x32	79
Descrição em System Verilog	79
Visão RTL	79
Golden Model.....	79
TestBench.....	80
Simulação	80
ULA (1 bit)	81
Descrição em System Verilog	81
Visão RTL	82
Golden Model.....	83
Vetores de Teste	83
TestBench.....	85
Simulação	86
ULA (32 bits).....	87
Descrição em System Verilog	87
Visão RTL	88
Golden Model.....	88

Vetores de teste.....	88
Simulação	89
PC, IR e MDR (32 bits).....	90
Descrição em System Verilog	90
Visão RTL	90
Golden Model.....	92
Vetores de Ouro	92
Simulação	93
Anexo II: Datapath	94
Funcionamento do DataPath	95
Conexão dos blocos	95
Fluxo de dados no Fetch.....	98
Descrição em System Verilog	99
Visão RTL.....	99
Golden Model	100
Vetores de Testes	100
TestBench	101
Simulação.....	102
Anexo III: Unidade de Controle	104
Máquina de Estados Finita - FSM.....	105
Metodologia de Síntese.....	105
Descrição dos modelos em System Verilog	107
Visão Máquina de Estados.....	109
Visão RTL	110
Golden Model.....	111
Vetores de Teste	112
TestBench.....	115
Simulação	115



**Universidade Federal da Paraíba
Centro de Informática**

MIPS32 Multiciclo

Relatório sobre a implementação do MIPS32 multiciclo como entrega do projeto da disciplina de Concepção Estrutural de Circuitos Integrados, ministrada pelo professor Antônio Carlos Cavalcante, no Centro de Informática da Universidade Federal da Paraíba.

João Pessoa, 2020.

MIPS32

O Mips32 de multiciclo é um microprocessador com arquitetura baseada em conjuntos de instruções simples. A arquitetura MIPS é uma das arquiteturas de computadores originais do tipo *Reduced Instruction Set* (RISC). Teve como consequência um aumento da eficiência das arquiteturas de computadores.

Um microprocessador multiciclo apresenta um conjunto completo instruções lógicas, aritméticas e imediatas. Neste projeto foi optado por implementar um conjunto de instruções reduzidas com as seguintes funções: LW, SW, ADD, SUB, AND, OR, NOR, XOR, SLT ADDI, ORI, NORI, XORI, SLTI, BEQ, BNE, J.

A seguir será dada uma ênfase nos tópicos referentes à arquitetura RISC e ao conjunto de instruções utilizados e a arquitetura multiciclo, respectivamente.

Arquitetura RISC

Em 1980, pesquisadores se opuseram ao conjunto de instruções complexas (CISC) em máquinas e iniciaram pesquisas relacionadas ao estudo e criação de um novo tipo de arquitetura. Desta forma, surgiram os processadores com conjuntos de instruções reduzidas (RISC), os quais buscavam maximizar o desempenho do sistema em relação a um processador CISC. A proposta RISC baseava-se na implementação de poucas instruções simples que pudessem ser combinadas entre si para obter o mesmo resultado de uma instrução complexa do CISC, porém com resultado em menor tempo.

Nesse contexto surgiu a arquitetura MIPS para microcontroladores, cujo conjunto de instruções foi dividido em categorias como:

- Aritméticas;
- Lógicas;
- Armazenamento/Leitura;
- *Jump e Branch*;
- Ponto Flutuante;
- Controle;
- *Shift e Move*;
- Gerenciamento de Memória;
- Especial.

Conjunto de Instruções

Neste trabalho serão abordadas algumas instruções dos quatro primeiros subconjuntos de instruções (aritmética, lógica, armazenamento/leitura, jump e branch) citados acima. Os formatos de instruções são compostos de 32 bits e podem ser classificados no formato R, I ou J.

Formato R

As instruções no formato R define a organização dos bits de funções lógicas, aritméticas e de deslocamento. O formato de preenchimento dos seus 32 bits é apresentado na Figura 1.

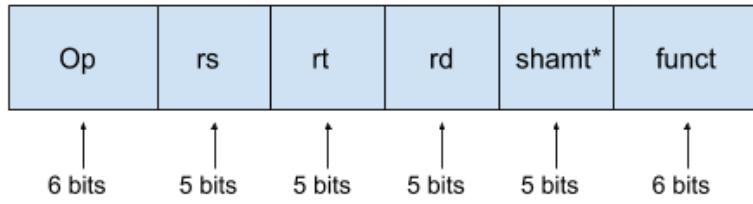


Figura 1 - Formato de instrução do tipo R.

Os campos destacados na Figura 1 são alocados de acordo com as seguintes divisões:

1. **Op** - código de operação que especifica a operação
2. **rs** - endereço de arquivo de registro do primeiro operando
3. **rt** - endereço de arquivo de registro do segundo operando
4. **rd** - endereço de arquivo de registro do destino do resultado
5. **shamt** - shift amount *(para instruções de deslocamento)
6. **funct** - código de função de operação.

No padrão de organização apresentado na Figura 1, o rs, rt e rd são registradores de propósito geral que armazenam os dados que estão sendo processados e o resultado da operação. E o campo Op é definido como 0.

Instruções Lógicas

Algumas instruções lógicas do formato R estão descritas na Tabela 1.

Instruções Lógicas	
AND	-
OR	-
Not Or	NOR
Exclusive Or	XOR

Tabela 1 - mnemônico das instruções lógicas.

AND

A operação lógica AND resulta em um valor lógico verdadeiro apenas quando seus dois operandos estiverem em nível lógico alto. Para que seja possível a realização desta conjunção, o campo *Op* deve ser nulo e o conjunto de bits que representam a função AND foi definido como sendo 100000.

Desta forma, a função lógica (definida no campo *funct*) será realizada com o conteúdo dos endereços dos registradores rs e rt. A sua representação deverá ser a seguinte:

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000	

6 5 5 5 5 6

Assim, a operação é realizada bit-a-bit entre os operandos e o resultado será salvo no endereço do rd.

OR

A função OR utiliza a mesma lógica de representação da AND, os seus operandos são salvos no endereço de rs e rt, é realizada a operação lógica entre eles e o resultado salvo no rd.

Diferentemente da operação AND, a função OR retorna o nível lógico alto sempre que quaisquer de sua entrada for 1. Desse modo, o código que define essa função é definido como 100101. O alocamento dos bits é realizado da seguinte forma:

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101	

6 5 5 5 5 6

NOR

A lógica de funcionamento da função NOR é a negação da função OR. O seu código é definido por 100111.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111	

6 5 5 5 5 6

XOR

De modo análogo as outras funções previamente definidas, os campos de registradores possuem as mesmas funções e esta operação pode ser representada pelo seguinte arranjo de bits.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110	

6 5 5 5 5 6

Instruções Aritméticas

O mesmo formato de representação utilizado até o momento também é válido para as operações aritméticas. Tais instruções são exemplificadas na Tabela 2.

Instruções Aritméticas	
Add Word	ADD
Subtract Word	SUB
Set on Less Than	SLT

Tabela 2 - mnemônico das instruções aritméticas.

ADD

A representação desta operação é indicada na Figura abaixo:

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000	6

Neste caso os valores contidos nos endereços de rs e rt são somados de modo lógico e atribuídos ao endereço contido em rd.

SUB

A operação de subtração possui o mesmo funcionamento do ADD, porém com o código de representação da função de subtração.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010	6

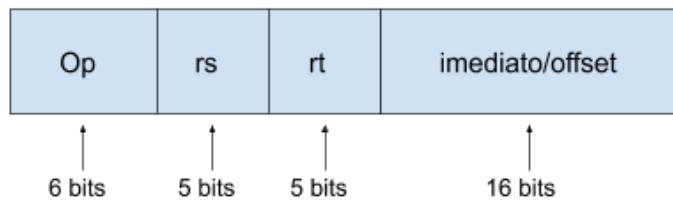
SLT

Esta operação (set on less-than - SLT) realiza a seguinte operação, compara os valores de rs e rt, caso rs seja menor que rt o resultado da operação é TRUE. Caso contrário, FALSE. A operação pode ser representada da seguinte forma

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010	6

Formato I

As instruções imediatas estão relacionadas a transferência de dados e são estruturadas conforme a seguinte estrutura:



É perceptível a ausência do registrador rd no formato R e do código da função. O código da função é definido, desta vez no campo Op.

Instruções Lógicas Imediatas

As seguintes operações lógicas podem ser implementadas no formato I.

Instruções Lógicas Imediatas	
Or Immediate	ORI
Exclusive Or Immediate	XORI
Add Immediate	ADDI

ORI

O conteúdo do imediato é completado a esquerda de zeros para que seja realizada a operação lógica. Desta forma seu resultado é salvo no endereço de rt.

31	26 25	21 20	16 15	0
	ORI 001101	rs 6	rt 5	immediate 16

ADDI

Nesse esquema o offset é utilizado como um dos operandos imediatos e é somado com o conteúdo do rs, cujo resultado da operação é salvo em rt. Tal representação é ilustrada abaixo.

31	26 25	21 20	16 15	0
	ADDI 001000	rs 6	rt 5	immediate 16

XORI

De modo análogo as outras operações, a função é executada entre os operandos do imediato e rs, e por sua vez, é salva em rt.

31	26 25	21 20	16 15	0
XORI 001110	rs 5	rt 5	immediate	16

Instruções Aritméticas Imediatas

O SLTI (do inglês, Set on Less Than Immediate) é um tipo de instrução que pode ser implementado de modo imediato.

SLTI

Utiliza-se do mesmo princípio lógico do SLT, porém a operação é realizada com o imediato e o rs, e salva em rt.

31	26 25	21 20	16 15	0
SLTI 001010	rs 5	rt 5	immediate	16

Instruções de Leitura e Armazenamento

As instruções de leitura e armazenamento são denominadas LW (do inglês, Load Word) e SW (do inglês, Store Word), respetivamente. Para essas instruções o rs é o registrador da fonte no qual é salvo o valor da base.

SW

A instrução SW armazena um valor que está em um registrador. O código de execução desta operação possui o seguinte formato: SW rt (registrador destino), offset(base). A máquina irá salvar o valor de rt no endereço de memória[offset+base]. Ou seja, é uma instrução de movimentação de dados do registrador para a memória.

31	26 25	21 20	16 15	0
SW 101011	base 5	rt 5	offset	16

LW

O LW é uma instrução de movimentação de dados do registrador para a memória, ou seja ele vai alocar o valor da memória[offset+base] no registrador rt.

31	26 25	21 20	16 15	0
LW 100011	base 5	rt 5		offset 16

Instruções de Branch (Desvio Condicional)

As instruções de branch seguem o mesmo formato das instruções de armazenamento. As operações de branch podem ser subdivididas entre Branch on Equal (BEQ) e Branch on Not Equal (BNE).

Branch on Equal

Essa instrução possui o seguinte formato (BEQ rs, rt, offset). Caso o valor de rs seja igual ao rt, a instrução é desviada para o label contido no offset.

31	26 25	21 20	16 15	0
BEQ 000100	rs 5	rt 5		offset 16

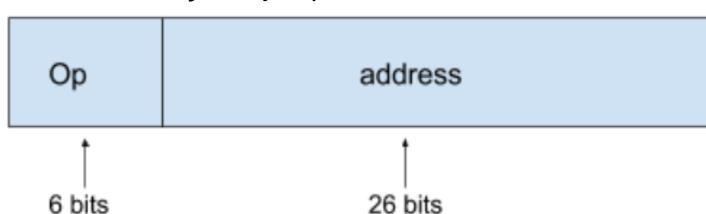
Branch on not Equal

De modo contrário, caso os valores sejam opostos o programa é desviado para o label.

31	26 25	21 20	16 15	0
BNE 000101	rs 5	rt 5		offset 16

Formato J

A instrução do tipo Jump deve ser formatada do seguinte modo: 6 bits para indicar a função e os outros 26 bits o endereço do jump.



Instruções de Jump

JUMP

O formato de organização do array é ilustrado abaixo. Essa instrução executa um salto para o label que está no index.

31	26 25	0
J 000010		instr_index 26

Arquitetura Multiciclo

Diferentemente da instrução monociclo que executa as instruções durante um ciclo de relógio, este tipo de implementação do MIPS baseia-se através da microarquitetura multiciclo. Neste tipo de microarquitetura as instruções são divididas em etapas/ciclos menores, no qual cada uma é realizada durante o ciclo de relógio. Deste modo não há a necessidade de duplicação de hardware que existe no monociclo, pois, as instruções mais simples são executadas mais rápido, consequentemente o custo do hardware é diminuído pela reutilização de blocos.

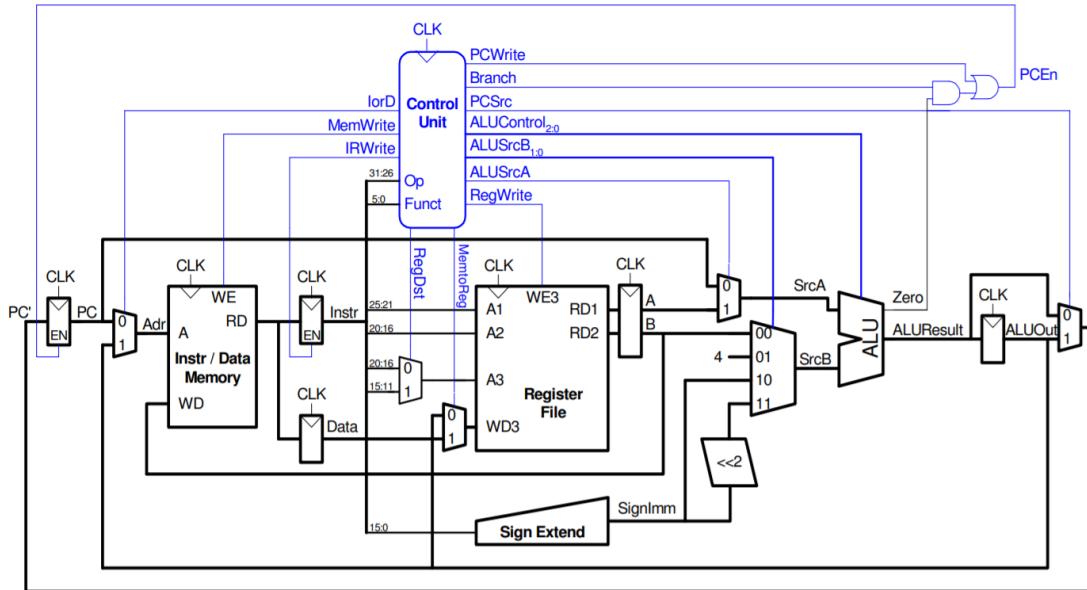
Implementação do modelo

O módulo do MIPS32 busca estabelecer a conexão entre a unidade de controle e o datapath. A implementação utilizada foi a multiciclo, na qual cada etapa da execução leva 1 ciclo de relógio. A implementação multiciclo permite uma unidade a ser usada mais de uma vez por instrução, desde que seja usada em diferentes ciclos de relógio. Esse compartilhamento pode ajudar a reduzir a quantidade de hardware necessária.

A capacidade de permitir instruções para obter diferentes números de ciclos de relógio e a capacidade compartilhar unidades funcionais na execução de uma única instrução são as principais vantagens de um design multiciclo [2].

A descrição da implementação dos componentes do datapath, o próprio datapath e a unidade de controle são apresentados nos anexos I, II e III, respectivamente.

A implementação do MIPS32 consiste na passagem dos parâmetros entre esses módulos, para que ambos possam ser interconectados. Dessa forma, ao realizar esta conexão, a interligação entre os blocos fica de acordo com a Figura mostrada abaixo, ou seja, a Figura apresenta o diagrama em blocos da estrutura interna do mips 32. Tal estrutura será modelada em system-verilog nas diantes Seções.



Hierarquia dos arquivos

A metodologia utilizada neste relatório para implementar a concepção estrutural do addac consiste em 6 passos:

1. Descrição do modelo em System Verilog;
2. Visão RTL;
3. Golden Model;
4. Vetores de Testes;
5. Arquivo de TestBench;
6. Simulação RTL.

Descrição dos modelos em System Verilog

Um bloco de hardware com entradas e saídas é chamado de módulo. Os dois estilos gerais para a descrição da funcionalidade do módulo são a comportamental, que descrevem o que o módulo faz. Enquanto, os modelos estruturais descrevem a sua construção a partir de partes simples; é uma aplicação de hierarquia. [Harris, 2013].

Neste tópico do relatório, são definidos os módulos dos componentes do MIPS32 de modo estrutural através de programas descritos em system verilog. Neste modo, portas lógicas e módulos definidos nos anexos são conectadas entre si para criar uma lógica de especificação para que o circuito realize uma função.

Como já dito anteriormente, o mips é a conexão entre o datapath e a unidade de controle, desta forma na Figura a seguir é exibido o código em systemverilog do mips, no qual ocorrem as passagens de parâmetros para as instâncias.

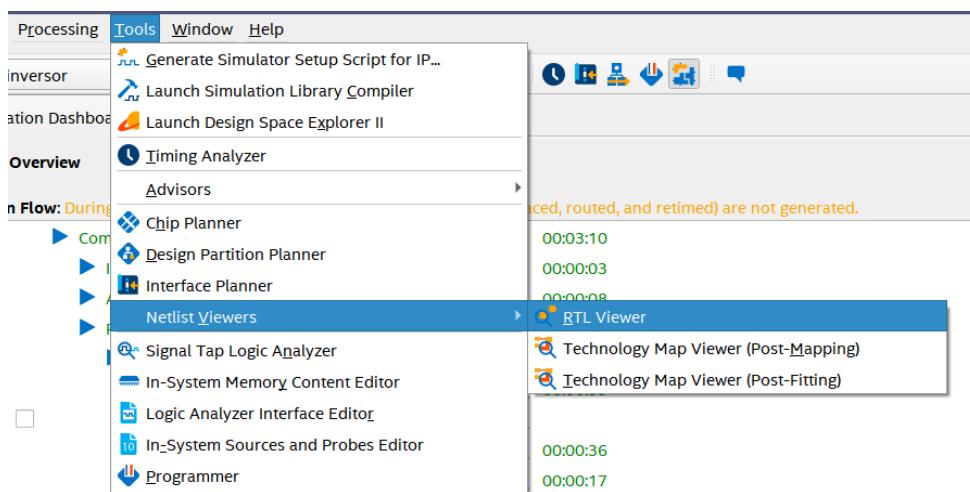
```

1 module mips(
2     input clk,
3     input rst,
4     input logic [31:0] RD,
5     output logic [31:0] ADR, WD,
6     output logic MemWrite, overflow
7 );
8
9     logic IorD;
10    logic ALUSrcA;
11    logic IRWrite;
12    logic PCWrite;
13    logic RegDst;
14    logic MemtoReg;
15    logic RegWrite;
16    logic Branch;
17    logic [1:0] ALUSrcB;
18    logic [1:0] ALUOp;
19    logic [1:0] PCSrc;
20    logic [05:0] OP;
21    logic [05:0] Funct;
22    logic BranchNE;
23    logic [2:0]ALUControl;
24
25 datapath datapath(clk,rst,IorD,ALUSrcA, ALUSrcB, PCSrc,IRWrite, PCWrite, RegDst,MemtoReg,
26                     RegWrite, Branch, RD, ALUControl,BranchNE,OP, Funct, ADR, WD, overflow);
27
28 control_unit FSM_controller(clk, reset, OP, Funct,MemtoReg,RegDst,IorD,ALUSrcA,IRWrite,
29                             MemWrite,PCWrite,Branch,RegWrite,BranchNE,PCSrc,ALUSrcB,ALUControl);
30
31

```

Visão RTL

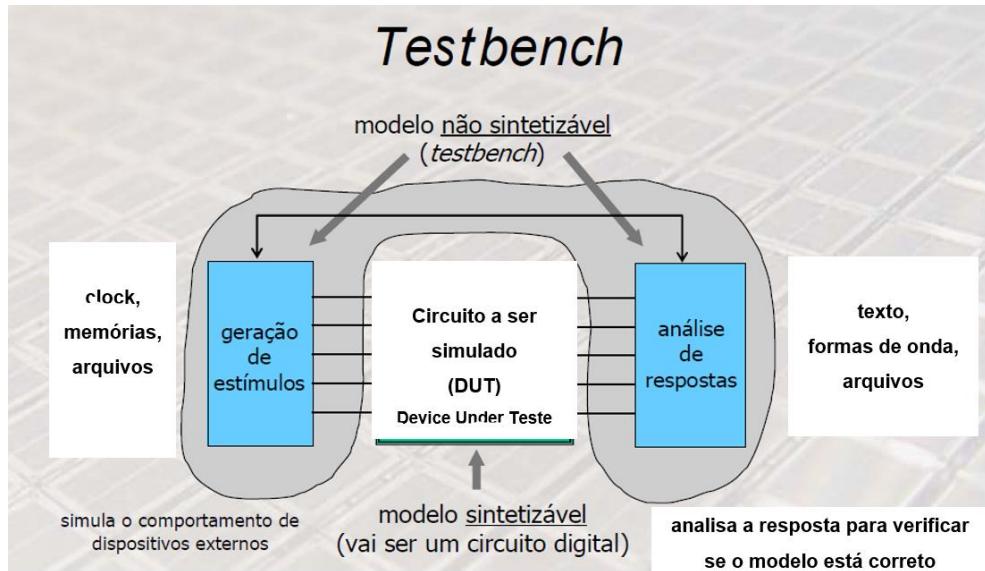
Após simular o arquivo .sv é gerado pelo Quartus o modelo RTL (do inglês, Register Transfer Level) do sistema modelado. Este modelo proporciona ao desenvolvedor do projeto uma visão de baixo nível do circuito modelado, na qual podem ser analisadas as conexões físicas internas do hardware e consequentemente a maneira em que o fluxo de sinal irá ocorrer. Para isto, é preciso acessar uma ferramenta do Quartus denominada “Netlist Viewers” e abrir a opção “RTL Viewer”, como ilustra a Figura abaixo.



Dessa maneira, a seguinte imagem foi gerada de acordo com a descrição física do circuito modelado.

TestBench

A lógica de execução de um testbench está ilustrada na Figura abaixo:



Dados os estímulos de entrada do circuito, será elaborado uma instanciação dos módulos definidos na descrição em systemverilog denominado DUV (design under verification). Desse modo, as entradas são inseridas nesse circuito, porém o sinal de resposta gerado por este sistema é comparado com o sinal obtido no modelo de referência, elaborado na linguagem C dos vetores de teste. Caso esses dados apresentem alguma inconsistência, o módulo do testbench emitirá uma mensagem de sinalização apontando a variável errada e seu bit de erro. Desse modo, conseguimos localizá-lo e consertá-lo.

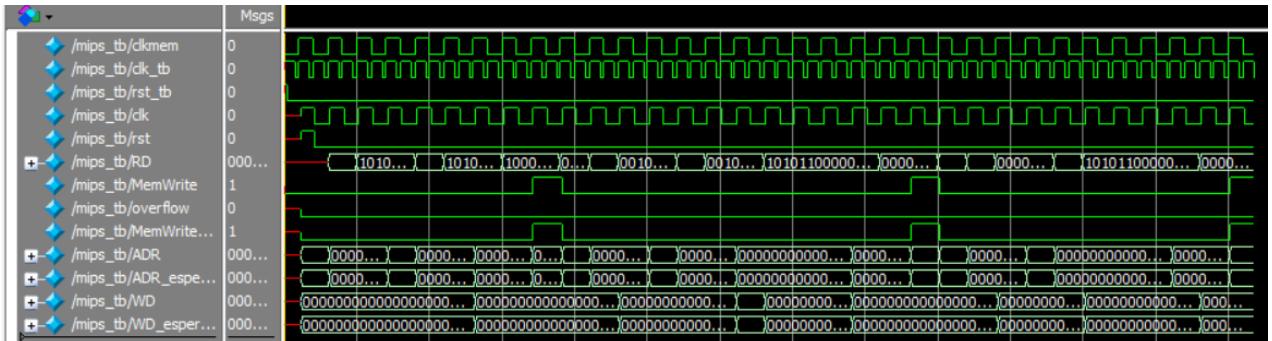
A descrição do Testbench foi realizada de acordo com o modelo indicado pelo professor. A interface com a memória do Mips32 foi incluída em forma de vetores. Dessa forma o array de bytes (8 bits) simula a memória.

```

28 initial begin
29     $display("Iniciando Testbench");
30
31     $display("clk|rst| RD | MemWrite | overflow| ADR | WD |");
32     $readmemb("C:/Users/satc1/OneDrive/Documentos/Concepcao/MIPS/simulation/modelsim/mips.tv",vectors);
33     counter=0; errors=0;
34     rst_tb = 1; #8; rst_tb = 0;
35
36     file = $fopen("C:/Users/satc1/OneDrive/Documentos/Concepcao/MIPS/simulation/modelsim/outuput_memory/memory.tv","w");
37
38     for(i=127; i>=0; i=i-1)
39         memory[i] = 8'b00000000;
40
41     {memory[0],memory[1],memory[2],memory[3]} = 32'b10001100000000001000000000100000; // Load Word
42     {memory[4],memory[5],memory[6],memory[7]} = 32'b10101100000000001000000000100100; // Store Word
43     {memory[8],memory[9],memory[10],memory[11]} = 32'b00011000000000001000000000101000; // Load Word
44     {memory[12],memory[13],memory[14],memory[15]} = 32'b00100000010000100000000000000001; // ADDI
45     {memory[16],memory[17],memory[18],memory[19]} = 32'b10101100000000001000000000000000; // Store Word
46     {memory[20],memory[21],memory[22],memory[23]} = 32'b0000100000000000000000000000000010011; // Jump
47     {memory[76],memory[77],memory[78],memory[79]} = 32'b0000000000100010001000000100000; // ADD
48     {memory[80],memory[81],memory[82],memory[83]} = 32'b10101100000000001000000000000000101100; // Store Word
49     {memory[32],memory[33],memory[34],memory[35]} = 32'b000000000000000000000000000000001111;
50     {memory[40],memory[41],memory[42],memory[43]} = 32'b000000000000000000000000000000001100;
51     clkmem = 0;
52     count = 0;
53     $fwrite(file,"-----Memoria inicial-----\n");
54
55     for(i=127; i>=0; i=i-1) begin
56         if(count == 4 ) begin
57             $fwrite(file,"\n");
58             count = 0;
59         end
60         if(count == 0)
61             $fwrite(file,"[0x2h]",127- i);
62
63         $fwrite(file,"%b",memory[127-i]);
64         count++;
65     end
66 end
67
68 always @ (posedge clk_tb) begin
69     if(~rst_tb) begin
70         clkmem = !clkmem;
71         #5
72         {clk,rst,ADR Esperado,WD Esperado,MemWrite Esperado} = vectors[counter];
73     end
74 end
75
76
77 always @ (posedge clkmem) begin
78     RD = {memory[ADR],memory[ADR+1],memory[ADR+2],memory[ADR+3]};
79     if(MemWrite)begin
80         {memory[ADR],memory[ADR+1],memory[ADR+2],memory[ADR+3]} = WD;
81     end
82 end
83
84 always @ (negedge clk_tb)
85     if(~rst_tb)
86     begin
87         aux_error = errors;
88         assert ( MemWrite == MemWrite_esperado )
89         else
90             begin
91                 $display("Error MemWrite: %b, expected %b", MemWrite, MemWrite_esperado);
92
93                 errors = errors + 1;
94             end
95             assert ( WD == WD_esperado )
96             else
97                 begin
98                     $display("Error WD: %b, expected %b", WD, WD_esperado);
99
100                errors = errors + 1;
101            end
102            assert ( ADR == ADR_esperado )
103            else
104                begin
105                    $display("Error ADR: %b, expected %b", ADR, ADR_esperado);
106
107                errors = errors + 1;
108            end
109        end
110        if(aux_error == errors)
111            $display(" | %b | OK", clk,rst,RD,MemWrite,overflow,ADR,WD);
112        counter++;
113
114        if(counter == $size(vectors)) begin
115            $display("Testes Efetuados = %0d", counter);
116            $display("Erros Encontrados = %0d", errors);
117            #1

```


Abaixo é possível visualizar os sinais da simulação RTL e realizar a análise do comportamento das variáveis simuladas bem como os seus resultados esperados.





**Universidade Federal da Paraíba
Centro de Informática**

Anexo I: Componentes Datapath

Neste anexo constam as descrições, modelagens, Golden models, vetores de teste e simulações dos blocos utilizados no dapopath.

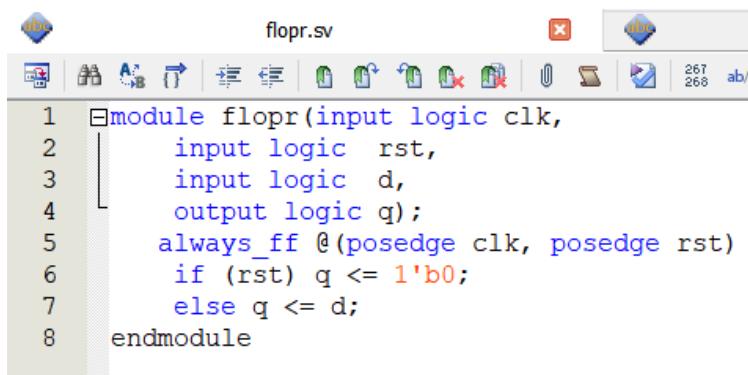
João Pessoa, 2020.

Flip-Flop com Reset (flop)

Descrição em System Verilog

Nas declarações always em SystemVerilog, e process, em VHDL, os sinais mantêm o seu valor antigo até que um evento na lista de sensibilidade aconteça e, explicitamente, causem a sua mudança. Consequentemente, tais códigos, com apropriadas listas de sensibilidade, podem ser utilizados para descrever circuitos sequenciais com memória. Por exemplo, o flip-flop possui apenas clk na sua lista de sensibilidade. Ele guarda o último valor de q até à próxima borda de subida de clk, mesmo se d muda temporariamente [Harris, Projeto Digital e Arquitetura de Computadores].

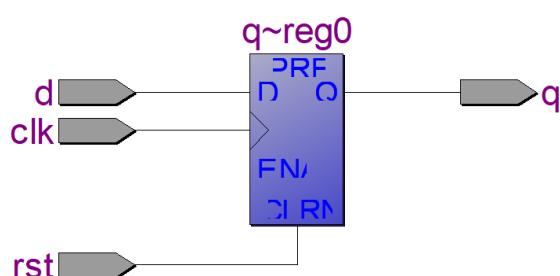
O Flip-flop com reset apresenta o seguinte comportamento, caso o reset esteja ativo, independentemente do valor do clock a sua saída será 0. Porém quando ele está desabilitado, o valor da saída é igual ao da entrada, porém apenas é carregado durante a borda de subida do clock. A descrição deste flip-flop é apresentada na imagem a seguir.



```
module flop(input logic clk,
             input logic rst,
             input logic d,
             output logic q);
    always_ff @(posedge clk, posedge rst)
        if (rst) q <= 1'b0;
        else q <= d;
endmodule
```

Visão RTL

Após simular o arquivo .sv é gerado pelo Quartus o modelo RTL (do inglês, Register Transfer Level) do sistema modelado. Este modelo proporciona ao desenvolvedor do projeto uma visão baixo nível do circuito modelado, na qual podem ser analisadas as conexões físicas internas do hardware e consequentemente a maneira em que o fluxo de sinal irá ocorrer. Para isto, é preciso acessar uma ferramenta do Quartus denominada “Netlist Viewers” e abrir a opção “RTL Viewer”, como ilustra a Figura 5.



Golden Model

Para validar o funcionamento do circuito modelado, é gerado seu modelo de referência comportamental, através de um programa que testa todas as suas possíveis entradas e obtém um sinal de saída, para cada uma delas. Desse modo foi elaborado um programa na linguagem C que simula o comportamento do bloco. A Figura abaixo representa esta simulação.

```
int flopr_func(){
    int q_ant = 0, q = 0;
    FILE *arquivo;
    int y = 0;
    arquivo = fopen("Flopr/Simulation/ModelSim/flopr.tv", "w");

    for (int clk = 0; clk < 2; clk++){
        for (int reset = 0; reset < 2; reset++){
            for (int d = 0; d < 2; d++){
                if (reset){
                    q = 0;
                    q_ant = 0;
                }
                if ((!reset) & (clk == 0)){
                    q = q_ant;
                }
                if ((!reset) & (clk == 1)){
                    q_ant = q;
                    q = d;
                }
                fprintf(arquivo, "%d_%d_%d_%d\n", clk, reset, d, q);
            }
        }
        fclose(arquivo);
    }
}
```

Vetores de Teste

Com a simulação do modelo de referência são gerados os vetores de testes, que são o modelo de referência do componente. Neste caso, ele representa todas as saídas corretas para quaisquer conjuntos de entradas. A tabela ilustra os vetores obtidos com o modelo de ouro.

CLK	RST	D	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0

1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Arquivo de TestBench

Para simular o arquivo descrito em system verilog e assegurar que o seu comportamento é equivalente ao obtido pelos vetores de testes, é realizado um arquivo de tesbench para simulá-lo no ModelSim.

```

1  `timescale 1ns/100ps
2  module flop_tb ();
3      logic d, clk, rst;
4      logic q, q_esperado; reset; clock;
5      logic [3:0]vectors[8];
6      int counter, errors, aux_error;
7
8      flop DUV(clk, rst, d, q);
9
10     initial begin
11         $display("Iniciando Testbench");
12         $display(" | CLK|Rst| D | Q |");
13         $display("-----");
14         $readmemb("../Simulation/ModelSim/flop_tb.v",vectors);
15         counter=0; errors=0;
16         reset = 1; #10; reset = 0;
17     end
18
19     always begin
20         clock=1; #6;
21         clock=0; #10;
22     end
23
24     always @ (posedge clock) begin
25         if(~reset) begin
26             {clk,rst,d,q_esperado} = vectors[counter]; //atualiza os valores na borda de subida
27         end
28     end
29

```

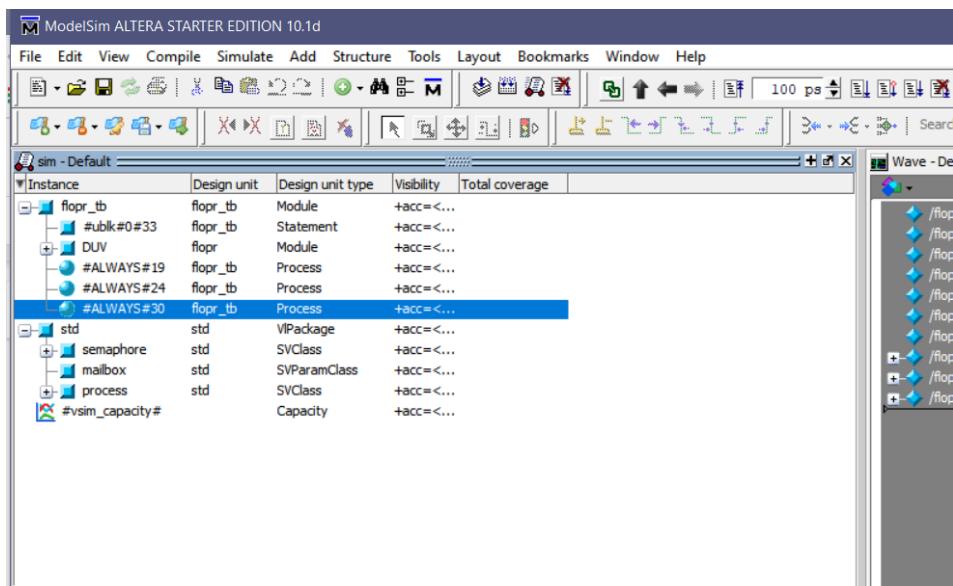
```

30     always @ (posedge clock) //Sempre (que o clock descer)
31         if (~reset)
32             begin
33                 aux_error = errors;
34                 assert (q == q_esperado)
35             end
36             begin
37                 errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
38                 end
39             if (aux_error == errors)
40                 $display ("| %b | %b | %b | %b | OK", clk, rst, d, q);
41             else
42                 $display ("| %b | %b | %b | %b | ERRO", clk, rst, d, q);
43
44             counter++; //Incrementa contador dos vetores de teste
45
46             if (counter == $size(vectors)) //Quando os vetores de teste acabarem
47             begin
48                 $display("Testes Efetuados = %0d", counter);
49                 $display("Erros Encontrados = %0d", errors);
50                 #10
51                 $stop;
52             end
53         end
54     endmodule

```

Simulação

Inicialmente, é realizada a criação de um projeto através do botão jumpstart, são adicionados os arquivos principais, testbench e o modelo de referência, ou seja, arquivo.sv, arquivo_tb.sv e arquivo.tb. Em seguida, é realizada a compilação de todos os arquivos através da sequência Compile->Compile all. Após apresentar uma mensagem no Transcript de que todos os arquivos foram executados sem falhas, é iniciada a simulação dos arquivos com os comandos Start Simulation -> Work -> arquivo_tb.sv.



A simulação é realizada no ModelSim, porém, foram efetuados 8 testes com 3 erros, como mostra a Figura.

```

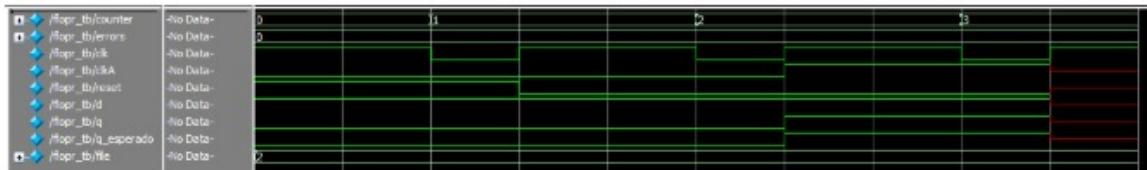
VSIM 48> run -all
# Iniciando Testbench
# |CLK|Rst| D | Q |
#
# | 0 | 0 | 0 | x | ERRO
# | 0 | 0 | 1 | x | ERRO
# | 0 | 1 | 0 | 0 | OK
# | 0 | 1 | 1 | 0 | OK
# | 1 | 0 | 0 | 0 | OK
# | 1 | 0 | 1 | 0 | ERRO
# | 1 | 1 | 0 | 0 | OK
# | 1 | 1 | 1 | 0 | OK
# Testes Efetuados = 8
# Erros Encontrados = 3

```

Os sinais encontrados são ilustrados abaixo:



No entanto, os arquivos foram ajustados e o resultado obtido sem erros foi o seguinte:



Flip-Flop com Enable e Rst (fopenr 1 bit)

Os registradores com enable respondem ao clock apenas quando a liberação (enable) é confirmada. Desse modo, ele apresenta a restrição de funcionamento do flip-flop anterior, que se baseia na ativação e desativação do reset.

Descrição em System Verilog

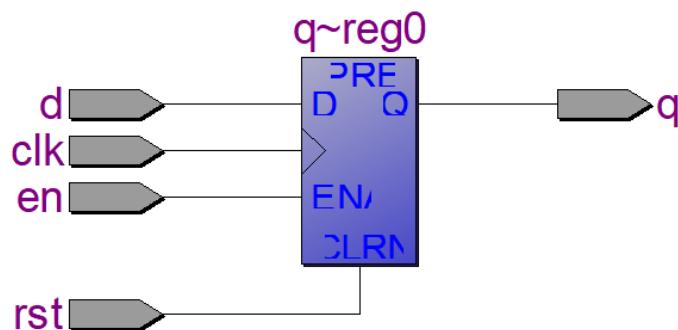
A descrição do arquivo é ilustrada na Figura abaixo, o que é modificado do flip-flop anterior é a inclusão do enable e a sua verificação para a atualização da saída.

```
module fopenr (input logic clk,
               input logic reset,
               input logic en,
               input logic d,
               output logic q);

    always_ff @ (posedge clk, posedge reset) begin
        if (reset) q <= 0;
        else if (en) q <= d;
    end
endmodule
```

Visão RTL

A visão RTL do componente é ilustrada abaixo.



Golden Model

O código do modelo de ouro deste componente também foi realizado em C e a lógica de implementação baseia-se na alteração dos valores do clock, enable, reset e a entrada de modo alinhado. Dentro da execução deste arquivo é gerado os vetores de testes em outro arquivo.

```

def flopenr(clk, rst, en, d):
    y = "x"

    if clk == 1 or rst == 1:
        if rst:
            y = 0
        else:
            if (en):
                y = d

    return y

from flopenr import *

arqOut = open("flopenr.tv", "w")

print("| clk | rst | en | d | y |")

for rst in range(0, 2, 1):
    for en in range(0, 2, 1):
        for clk in range(0, 2, 1):
            for d in range(0, 2, 1):
                y = flopenr(clk, rst, en, d)
                print("| {0} | {1} | {2} | {3} | {4} |".format(
                    clk, rst, en, d, y))
                arqOut.write("{0}_{1}_{2}_{3}_{4}\n".format(
                    clk, rst, en, d, y))

arqOut.close()

```

Vetores de Teste

Os vetores de testes são ilustrados na Tabela abaixo:

0_1_0_0_0
0_1_0_1_0
1_1_0_0_0
1_1_0_1_0
0_1_1_0_0
0_1_1_1_0
1_1_1_0_0
1_1_1_1_0
0_0_0_0_0
0_0_0_1_0
1_0_0_0_0
1_0_0_1_0
0_0_1_0_0
0_0_1_1_0
1_0_1_0_0
1_0_1_1_1

Arquivo de TestBench

O arquivo do testbench deste modelo é similar ao anterior, foram realizadas pequenas mudanças no código, como por exemplo, a inserção de novas variáveis e alteração na concatenação do vetor de teste. Este arquivo foi feito em system-verilog. E por conseguinte foi simulado no ModelSim.

```
`timescale 1ns/100ps

module flopenr_rb;
    parameter max_vectors = 16;
    int counter, errors, aux_error;

    logic clk, clkA, rst, rstA, en, d, q, q_esperado;
    logic [4:0] vectors[max_vectors];
    integer file;

    flopenr flopenr(.clk(clkA), .reset(rstA), .en(en), .d(d), .q(q));

    initial begin
        file = $fopen("flopenr_out.txt");
        $display("Iniciando o TestBench");
        $display("-----");
        $display("|CLK| RST| EN| D | Q |");
        $readmemb("flopenr.tv", vectors);
        counter = 0; errors = 0;
        rst = 1; #10; rst = 0;
    end

    always begin
        #15;
        clk = 1; #20;
        clk = 0; #10;
    end

    always @ (posedge clk) begin
        if (~rst)begin
            {d, rstA, en, clkA, q_esperado} = vectors[counter];
        end
    end
end
```

```

always @ (negedge clk) begin
    if (~rst) begin
        aux_error = errors;
        assert (q == q_esperado)
        else begin
            errors = errors + 1;
        end
    end
    if(aux_error == errors)
        $display(" | %b | %b | %b | %b | %b | OK", clkA, rstA, en, d, q);
    else
        $display(" | %b | %b | %b | %b | %b | ERRO", clkA, rstA, en, d, q);

    counter++;
    if (counter == max_vectors) begin
        $display("Testes Efetuados = %0d", counter);
        $display("Erros Encontrados = %0d", errors);
        #10
        $stop;
    end
end
endmodule

```

Simulação

A simulação foi realizada pelo ModelSim, através da mesma metodologia utilizada anteriormente. Inicialmente os arquivos do testbench e a descrição do componente são inseridos em um projeto criado diretamente no modelSim.

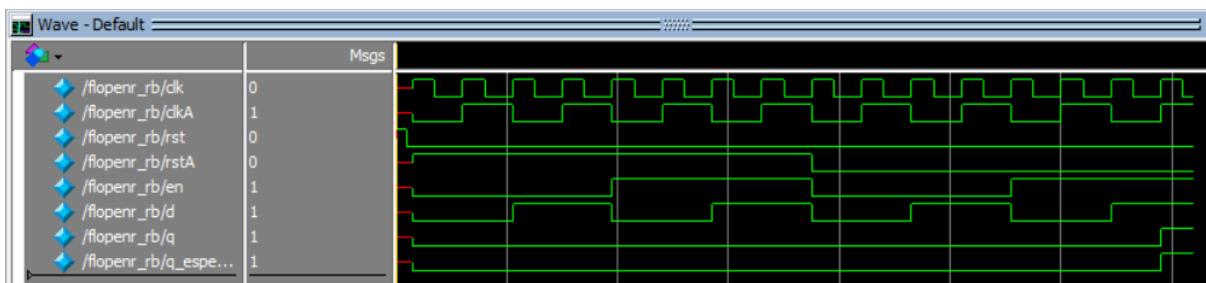
O Comportamento do acumulador equivale ao de um flip-flop com enable e reset. Neste caso, podemos ver que sua modelagem não apresenta erros. A sua saída equivale a entrada apenas quando o reset está desabilitado, o clock está em 1 e o enable ativo, ou seja na última e antepenúltima linha dos testbench.

```

# Iniciando o TestBench
# -----
# |CLK| RST| EN| D | Q |
# | 0 | 1 | 0 | 0 | 0 | OK
# | 1 | 1 | 0 | 0 | 0 | OK
# | 0 | 1 | 0 | 1 | 0 | OK
# | 1 | 1 | 0 | 1 | 0 | OK
# | 0 | 1 | 1 | 0 | 0 | OK
# | 1 | 1 | 1 | 0 | 0 | OK
# | 0 | 1 | 1 | 1 | 0 | OK
# | 1 | 1 | 1 | 1 | 0 | OK
# | 0 | 0 | 0 | 0 | 0 | OK
# | 1 | 0 | 0 | 0 | 0 | OK
# | 0 | 0 | 0 | 1 | 0 | OK
# | 1 | 0 | 0 | 1 | 0 | OK
# | 0 | 0 | 1 | 0 | 0 | OK
# | 1 | 0 | 1 | 0 | 0 | OK
# | 0 | 0 | 1 | 1 | 0 | OK
# | 1 | 0 | 1 | 1 | 0 | OK
# | 0 | 0 | 1 | 1 | 0 | OK
# | 1 | 0 | 1 | 1 | 1 | OK
# Testes Efetuados = 16
# Erros Encontrados = 0

```

Os sinais obtidos com a simulação do flip-flop com enable foram os seguinte:



Desta forma, não há erros no projeto deste componente pois durante a etapa da sua verificação foram encontrados 0 erros.

Flip-Flop com Enable e Rst (fopenr 32 bits)

Para montar esse registrador de 32 bits, foram utilizadas instâncias do módulo de 1 bit, anteriormente desenvolvido.

Descrição em SystemVerilog

As instâncias do módulo do registrador de 1 bit, acontecem na primeira linha de comando dentro do for. Desse módulo cada um dos 32 bits de entradas do registrador de 32 bits, são propagados para uma instância do registrador de 1 bit.

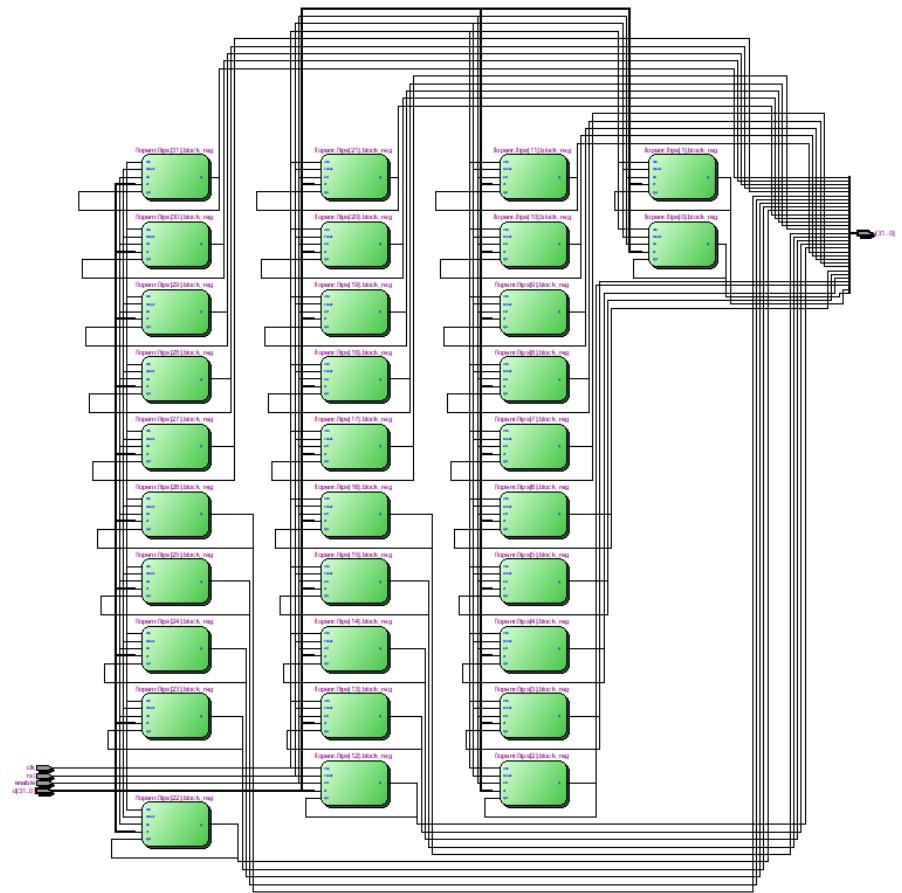
```
module reg32(rst, clk, enable, d, y);

    input logic [31:0] d;
    input logic clk, rst, enable;
    output logic [31:0] y;
    logic [31:0]qa;

    genvar i;
    generate
        for (i=0; i<32;i=i+1) begin: flips
            fopenr block_reg(clk, rst, enable, d[i], y[i],qa[i]); //instancia do flip
            assign qa[i] = y[i];
        end
    endgenerate
endmodule
```

Visão RTL

A visão em RTL permite a visualização das ligações entre os registradores de 1 bit e o registrador de 32.



Mux: 2-1 (1 bit)

O modelo de referência do arquivo mux.tv foi escrito com o bit equivalente a chave de seleção seguido das duas variáveis de entrada do mux e finalmente a variável de saída do bloco lógico. Dessa forma, sempre que o primeiro bit for equivalente a 0 a saída (o último bit) tem que ser igual ao bit equivalente a entrada A, ou seja, o segundo bit. Assim, a saída é igual ao segundo bit. Porém, se a chave de seleção estiver em 1 a saída é igual ao B.

A variável apresentada antes da interrogação é a condição do sistema, caso ela verdadeira a saída do sistema será a variável após a interrogação, caso contrário a variável após os dois pontos é atribuída a saída.

O código abaixo demonstra a linguagem para um multiplexador 2:1 com entradas e saídas de 1 bit utilizando o operador condicional.

Descrição em System Verilog

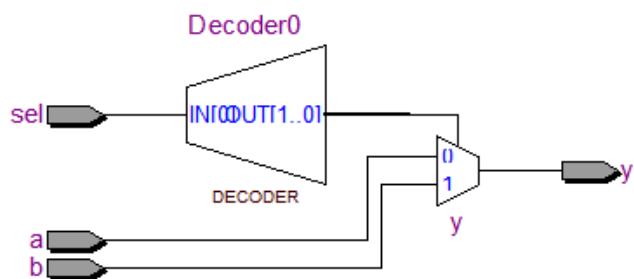
A descrição foi realizada de modo similar à encontrada no livro de arquitetura do Harris, e pode ser vista na Figura abaixo.

```
module mux (a,b,sel,y);
    input logic a,b,sel;
    output logic y;

    always@(*) begin
        case (sel)
            1'b0: y <= a;
            1'b1: y <= b;
        endcase
    end
endmodule
```

Visão RTL

A visão RTL do mux de 2 entradas foi gerada pelo quartus e resultou no seguinte esquemático.



Golden Model

O arquivo de geração dos vetores de teste é ilustrado abaixo.

```
int mux2_func()
{
    FILE *arquivo;
    int y = 0;
    arquivo = fopen("Mux2-1/Simulation/ModelSim/mux2.tv", "w");

    for (int Sel_0 = 0; Sel_0 < 2; Sel_0++)
    {
        for (int a = 0; a < 2; a++)
        {
            for (int b = 0; b < 2; b++)
            {
                if (!Sel_0)
                {
                    y = a;
                }
                else
                {
                    y = b;
                }
                fprintf(arquivo, "%d_%d_%d_%d\n", Sel_0, a, b, y);
            }
        }
    }
    fclose(arquivo);
}
```

Vetores de Teste

Sel0	a	b	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Arquivo de TestBench

O arquivo do testbench apresenta alterações no tamanho do *vectors* alterando-se para 8 linhas, que são o total das linhas impressas pelos vetores de testes. e a quantidade de bits para 4 que representa as 4 entradas/saidas do componente (a, b, sel y).

```
|timescale 1ns/100ps
module mux_tb;
    int counter, errors, aux_error;
    logic clk,rst;
    logic a , b, sel;
    logic y, y_esperado;
    logic [3:0]vectors[8];

    mux DUV(a, b, sel, y);

    initial begin
        $display("Iniciando Testbench");
        $display("|Sel| A | B | Y |");
        $display("-----");
        $readmemb("C:/Users/satc1/OneDrive/Documentos/Concepcao/Mux2-1/Simulation/ModelSim/mux2.tv",vectors);
        counter=0; errors=0;
        rst = 1; #10; rst = 0;
    end

    always begin
        clk=1; #30;
        clk=0; #5;
    end

    always @ (posedge clk) begin
        if(~rst)begin
            {sel,a, b, y_esperado} = vectors[counter];
        end
    end

    always @ (posedge clk) begin
        if(~rst)begin
            {sel,a, b, y_esperado} = vectors[counter];
        end
    end

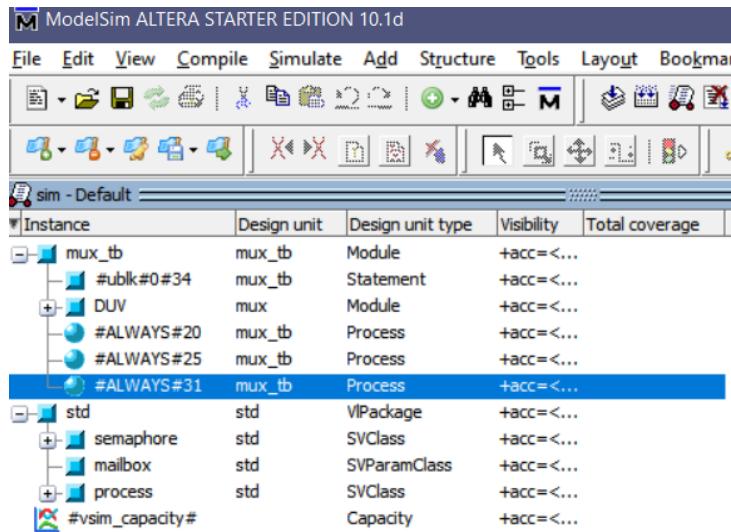
    always @ (negedge clk) //Sempre (que o clock descer)
    if(~rst)
    begin
        aux_error = errors;
        assert (y == y_esperado)
    end
    else
    begin
        errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
    end
    if(aux_error == errors)
        $display(" | %b | %b | %b | %b | OK", sel,a, b, y);
    else
        $display(" | %b | %b | %b | %b | ERRO", sel, a, b, y);

    counter++; //Incrementa contador dos vetores de teste

    if(counter == $size(vectors)) //Quando os vetores de teste acabarem
begin
    $display("Testes Efetuados = %0d", counter);
    $display("Erros Encontrados = %0d", errors);
    #10
    $stop;
end
end
endmodule
```

Simulação

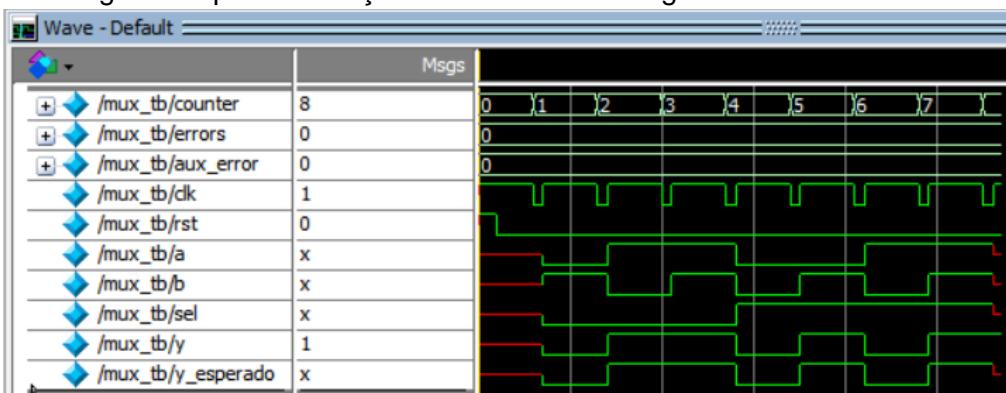
A inserção dos arquivos é realizada através de um projeto no Modelsim, como todos os outros casos anteriores.



A verificação do modelo simulado foi realizada com sucesso, resultando em 0 erros durante todos os 8 casos de testes dos vetores. Desse modo, o arquivo compara a resposta obtida pelo circuito com a esperada dos vetores de teste.

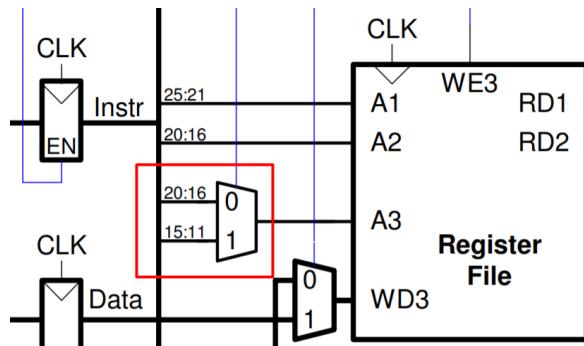
```
# Iniciando Testbench
# |Sel| A | B | Y |
# -----
# | x | x | x | x | OK
# | 0 | 0 | 1 | 0 | OK
# | 0 | 1 | 0 | 1 | OK
# | 0 | 1 | 1 | 1 | OK
# | 1 | 0 | 0 | 0 | OK
# | 1 | 0 | 1 | 1 | OK
# | 1 | 1 | 0 | 0 | OK
# | 1 | 1 | 1 | 1 | OK
# Testes Efetuados = 8
# Erros Encontrados = 0
```

Os sinais gerados pela simulação são ilustrados a seguir:



Mux: 2-1 (5 bits)

O multiplexador de duas entradas com 5 bits é necessário para conectar os bits das saídas do IR à entrada do banco de registradores. A Figura abaixo ilustra esta conexão destacada em vermelho.



Descrição em System Verilog

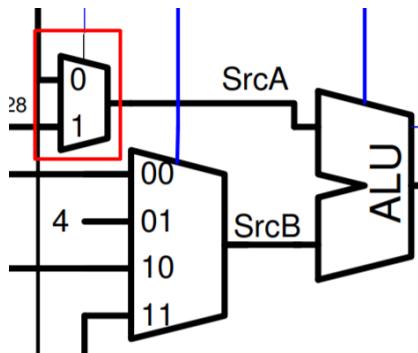
A modelagem deste mux, consistem 5 instâncias do mux 2-1 de 1 bit. A Figura a seguir descreve o código em systemverilog.

```
module mux2_5(a,b,sel,y);
    input logic [4:0]a;
    input logic [4:0]b;
    input logic sel;
    output logic [4:0]y;

    genvar i;
    generate
        for (i=0; i<5; i=i+1) begin: muxs5_2
            mux2 bloco(a[i], b[i], sel, y[i]);
        end
    endgenerate
endmodule
```

Mux: 2-1 (32 bits)

Este multiplexador é preciso na modelagem do datapath do mips. Este componente é posicionado no circuito de modo que a sua saída é a uma das entradas da ULA, com ilustra a Figura abaixo.



Descrição em System Verilog

A descrição consiste em um laço de repetição de 32 vezes que instancia o multiplexador 2x1.

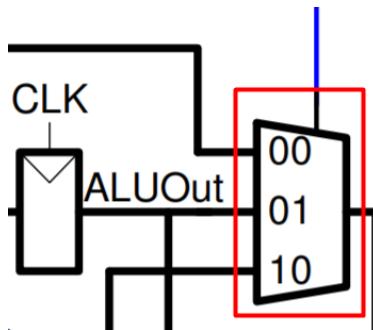
```
module mux2_32(a,b,sel,y);
    input logic [31:0]a;
    input logic [31:0]b;
    input logic sel;
    output logic [31:0]y;

    genvar i;
    generate
        for (i=0; i<32; i=i+1) begin: mux2_32
            mux2 mux22(a[i], b[i], sel, y[i]);
        end
    endgenerate
endmodule
```

Mux: 3-1 (32 bits)

O multiplexador 3 por 1 com entrada e saída de 32 bits, foi desenvolvido para ser utilizado no datapath. Tal multiplexador terá a sua saída futuramente como a entrada do Program Counter. Tal conexão só será estabelecida durante o desenvolvimento do caminho de dados, por enquanto iremos apenas nos ater ao desenvolvimento deste bloco lógico.

O posicionamento deste multiplexador se dá de modo que uma de suas entradas é a saída da ULA, e a saída do circuito do multiplexador é a entrada do PC. A Figura abaixo ilustra a utilização do bloco.



Descrição em System Verilog

O desenvolvimento do mux teria 4 possibilidades de entradas pelo fato de ter uma chave de seleção de dois bits, no entanto só tem três entradas. Desse modo caso a chave esteja em um estado impossível, que é o caso do 11, a saída retornará um lixo de memória.

```
module mux3(d0,d1,d2, Sel_1, Sel_0, y);

    input logic [31:0]d0, d1, d2;
    input logic Sel_1, Sel_0;
    output logic [31:0]y;
    logic [31:0]d3;//lixo, entrada impossivel

    assign y = Sel_1 ? (Sel_0 ? d3: d2)
                  : (Sel_0 ? d1 : d0);
endmodule
```

Visão RTL

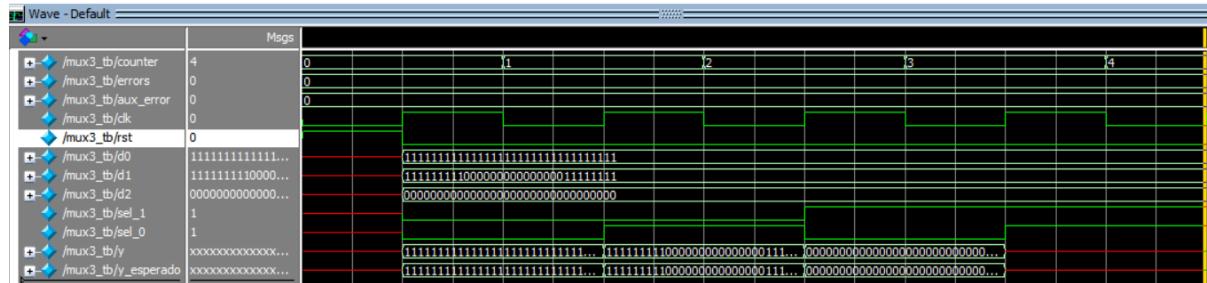
A visão RTL gerou dois mux 2 para 1 que constituem o mux 3 para 1.


```

VSIM 128> run -all
# Iniciando Testbench
# -----
# | D0 | D1 | D2 | Sel | Y |
# -----
# | 11111111111111111111111111111111 | 111111111000000000000000000000000000000000000000000000000000000000000000 | 00 | 11111111111111111111111111111111 | OK
# | 11111111111111111111111111111111 | 111111111000000000000000000000000000000000000000000000000000000000000000 | 01 | 111111111100000000000000000000000000000000000000000000000000000000000000 | OK
# | 11111111111111111111111111111111 | 111111111000000000000000000000000000000000000000000000000000000000000000 | 10 | 000000000000000000000000000000000000000000000000000000000000000000000000 | OK
# | 11111111111111111111111111111111 | 111111111000000000000000000000000000000000000000000000000000000000000000 | 11 | xxxxxxxxxxxxxxxxxxxxxxxxx | OK
# Testes Efetuados = 4
# Erros encontrados = 0
# Break in Module mux3_tb at C:/Users/satcl/OneDrive/Documentos/Concepcão/MIPS-Componentes/Mux3-1/TestBench/mux3_tb.sv line 56

```

Os sinais da Figura abaixo indicam os pulsos de transição entre as entradas e as saídas.



Mux: 4-1 (1 bit)

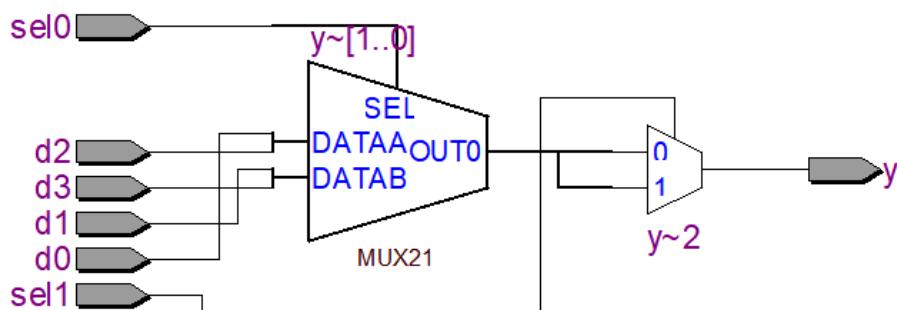
Este mux tem 4 entradas e 1 saída. A diferença na modelagem e descrição do componente baseia-se na inclusão de mais entradas e na escolha da saída do circuito com base nas chaves de seleção.

Descrição em System Verilog

A Descrição do circuito foi realizada de modo similar ao do livro do Harris.

```
module mux4( input logic d0, d1, d2, d3,
              input logic sel1, sel0,
              output logic y);
    assign y = sel1 ? (sel0 ? d3 : d2)
                  : (sel0 ? d1 : d0);
endmodule
```

Visão RTL



Golden Model

O código do modelo de ouro foi realizado de modo similar ao anterior, apenas com a inclusão de novas entradas e variação das chaves de seleção e de todas as entradas de modo alinhado.

```

int mux4_func()
{
    FILE *arquivo;
    int y = 0;
    arquivo = fopen("Mux4-1/Simulation/ModelSim/mux4.tv", "w");

    for (int Sel_0 = 0; Sel_0 < 2; Sel_0++)
    {
        for (int Sel_1 = 0; Sel_1 < 2; Sel_1++)
        {
            for (int a = 0; a < 2; a++)
            {
                for (int b = 0; b < 2; b++)
                {
                    for (int c = 0; c < 2; c++)
                    {
                        for (int d = 0; d < 2; d++)
                        {
                            if ((Sel_0 == 0) & (Sel_1 == 0))
                                y = a;
                            else if ((Sel_0 == 0) & (Sel_1 == 1))
                                y = b;
                            else if ((Sel_0 == 1) & (Sel_1 == 0))
                                y = c;
                            else
                                y = d;
                            fprintf(arquivo, "%d_%d_%d_%d_%d_%d\n", Sel_0, Sel_1, a, b, c, d, y);
                        }
                    }
                }
            }
        }
    }
}

```

Vetores de Teste

Como o mux possui 4 entradas e 2 chaves de seleções a quantidade de linhas para os vetores de teste é superior ao dos outros componentes modelados. Foram encontradas 64 linhas que representam todas as possíveis entradas e saídas deste multiplexador.

Sel0_Sel1_a_b_c_d_y
0_0_0_0_0_0_0
0_0_0_0_0_1_0
0_0_0_0_1_0_0
0_0_0_0_1_1_0
0_0_0_1_0_0_0
0_0_0_1_0_1_0
0_0_0_1_1_0_0
0_0_0_1_1_1_0
0_0_1_0_0_0_1
0_0_1_0_0_1_1
0_0_1_0_1_0_1
0_0_1_0_1_1_1
0_0_1_1_0_0_1
0_0_1_1_0_1_1
0_0_1_1_1_0_1
0_0_1_1_1_1_1
0_1_0_0_0_0_0

0_1_0_0_0_1_0
0_1_0_0_1_0_0
0_1_0_0_1_1_0
0_1_0_1_0_0_1
0_1_0_1_0_1_1
0_1_0_1_1_0_1
0_1_0_1_1_1_1
0_1_1_0_0_0_0
0_1_1_0_0_1_0
0_1_1_0_1_0_0
0_1_1_0_1_1_0
0_1_1_1_0_0_1
0_1_1_1_0_1_1
0_1_1_1_1_0_1
0_1_1_1_1_1_1
1_0_0_0_0_0_0
1_0_0_0_0_1_0
1_0_0_0_1_0_1
1_0_0_0_1_1_1
1_0_0_1_0_0_0
1_0_0_1_0_1_0
1_0_0_1_1_0_1
1_0_0_1_1_1_1
1_0_1_0_0_0_0
1_0_1_0_0_1_0
1_0_1_0_1_0_1
1_0_1_0_1_1_1
1_0_1_1_0_0_0
1_0_1_1_0_1_0
1_0_1_1_1_0_1
1_0_1_1_1_1_1
1_1_0_0_0_0_0
1_1_0_0_0_1_1
1_1_0_0_1_0_0
1_1_0_1_0_1_1
1_1_0_1_1_0_0
1_1_0_1_1_1_1
1_1_1_0_0_0_0
1_1_1_0_0_1_1
1_1_1_0_1_0_0
1_1_1_0_1_1_1
1_1_1_1_0_0_0
1_1_1_1_0_1_1
1_1_1_1_1_0_0
1_1_1_1_1_1_1

Arquivo de TestBench

A variável vectors apresenta 64 linhas para conter os vetores de testes e 7 bits para ter capacidade de armazenar todas as entradas/saídas (a,b,c,d,sel0,sel1,y).

```
`timescale 1ns/100ps
module mux4_tb;
    int counter, errors, aux_error;
    logic clk,rst;
    logic a, b, c, d, sel1, sel0;
    logic y, y_esperado;
    logic [6:0]vectors[64];

    mux4 DUV(a, b, c, d, sel1, sel0, y);

    initial begin
        $display("Iniciando Testbench");
        $display("|Sel0|Sel1| A | B | C | D | Y |");
        $display("-----");
        $readmemb(" .. /Simulation/ModelSim/mux4.tv",vectors);
        counter=0; errors=0;
        rst = 1; #10; rst = 0;
    end

    always begin
        clk=1; #30;
        clk=0; #5;
    end

    always @ (posedge clk) begin
        if(~rst)begin
            {a, b, c, d, sel1,sel0, y_esperado} = vectors[counter];
        end
    end
end
```

```

always @(posedge clk) begin
    if(~rst)begin
        {a, b, c, d, sel1,sel0, y_esperado} = vectors[counter];
    end
end

always @(negedge clk) //Sempre (que o clock descer)
begin
    aux_error = errors;
    assert (y == y_esperado)
end
begin
    errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
end
if(aux_error == errors)
    $display(" | %b | OK", sel0, sel1, a, b, c, d, y);
else
    $display(" | %b | ERRO", sel0, sel1, a, b, c, d, y);

counter++; //Incrementa contador dos vetores de teste

if(counter == $size(vectors)) //Quando os vetores de teste acabarem
begin
    $display("Testes Efetuados = %0d", counter);
    $display("Erros Encontrados = %0d", errors);
    #10
    $stop;
end
end
endmodule

```

Simulação

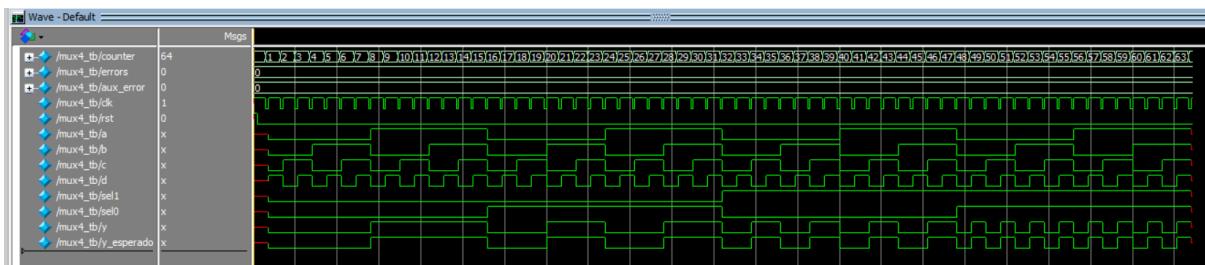
Todos os casos de testes foram validados com 0 erros encontrados nos 64 testes. A Figura abaixo ilustra parte dos prints apresentados pelo modelSim. Os casos de testes foram cortados para melhorar a visualização do arquivo.

```

# Iniciando Testbench
# |Sel0|Sel1| A | B | C | D | Y |
# -----
# | x | x | x | x | x | x | x | x | OK
# | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | OK
# | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | OK
# | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | OK
# | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | OK
# | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | OK
# | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | OK
# | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | OK
# | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | OK
# | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | OK
# | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | OK
# | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | OK
# | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | OK
# | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | OK
# | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | OK
# | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | OK
# | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | OK
# | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | OK
# | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | OK
# | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | OK
# | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | OK
# | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | OK
# | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | OK
# | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | OK
# | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | OK
# | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | OK
# | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | OK
# | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | OK
# | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | OK
# | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | OK
# | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | OK
# | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | OK
# | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | OK
# | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | OK
# | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | OK
# Testes Efetuados = 64
# Erros Encontrados = 0

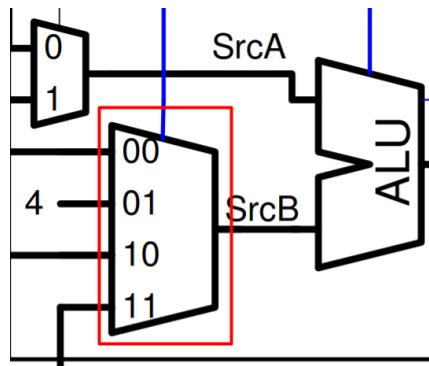
```

Os sinais encontrados foram os seguintes:



Mux: 4-1 (32 bits)

Este multiplexador deve ser implementado para conectar a entrada SrcB à ULA. Deste modo abaixo é ilustrada a sua modelagem.



Descrição em System Verilog

Esta descrição consiste em 32 instâncias do multiplexador 4x1 de 1 bits. As transmissões dos dados são realizadas através de um laço de repetição feito com o genvar.

```
module mux4_32 (d0,d1,d2,d3, Sel, y);
    input logic [31:0]d0,d1,d2,d3;
    input logic [1:0]Sel;
    output logic [31:0]y;

    genvar i;
    generate
        for (i=0; i<32; i=i+1) begin: blocomux4_32
            mux4 mux4_32(d3[i],d2[i],d1[i],d0[i], Sel[1:0], y[i]);
        end
    endgenerate
endmodule
```

Mux: 8-1 (1 bit)

Para isso foi realizado um mux de 8x1, com 2 instâncias dos mux 4x1 e uma do 2x1 já desenvolvidos nesse projeto.

Descrição em System Verilog

A descrição do módulo consiste em 2 instâncias do mux4x1 cujas saídas são denominadas intermediário, e são conectadas a entrada do mux2x1.

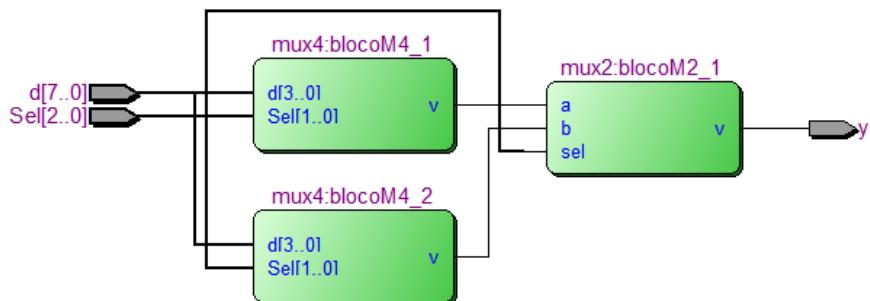
```
module mux8 (d, Sel, y);

    input logic [7:0] d;
    input logic [2:0] Sel;
    logic [1:0]intermediario;
    output logic y;

    mux4 blocoM4_1(d[3:0],Sel[1:0], intermediario[0]);
    mux4 blocoM4_2(d[7:4],Sel[1:0], intermediario [1]);
    mux2 blocoM2_1 (intermediario[0],intermediario[1], Sel[2], y);

endmodule
```

Visão RTL



Golden Model

O Golden Model consiste em laços de repetição aninhados com o intuito de alterar todos os bits das chaves de seleções e dos canais de entrada. O trecho do código é exemplificado abaixo.

```
int main()
{
    FILE *arquivo;
    int y = 0;
    int a = 0;
    int i = 0;
    arquivo = fopen("../Simulation/ModelSim/mux8.tv", "w");

    int Sel[3] = {0}, data[8] = {0};

    for (Sel[2] = 0; Sel[2] < 2; Sel[2]++)
        for (Sel[1] = 0; Sel[1] < 2; Sel[1]++)
            for (Sel[0] = 0; Sel[0] < 2; Sel[0]++)
                for (data[7] = 0; data[7] < 2; data[7]++)
                    for (data[6] = 0; data[6] < 2; data[6]++)
                        for (data[5] = 0; data[5] < 2; data[5]++)
                            for (data[4] = 0; data[4] < 2; data[4]++)
                                for (data[3] = 0; data[3] < 2; data[3]++)
                                    for (data[2] = 0; data[2] < 2; data[2]++)
                                        for (data[1] = 0; data[1] < 2; data[1]++)
                                            for (data[0] = 0; data[0] < 2; data[0]++)
                                            {
                                                a = Sel[0] * (1) + Sel[1] * (2) + Sel[2] * (4);
                                                y = data[a];
                                                fprintf(arquivo, "%d%d%d_%d%d%d%d%d%d%d%d\n", Sel[2], Sel[1], Sel[0],
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    fclose(arquivo);
}
```

Vetores de Ouro

Como no golden model existem 11 laços aninhados, temos que 2^{11} equivale a 2048, ou seja, temos 2048 linhas de vetores de teste. Neste relatório, serão ilustrado apenas alguns casos devido ao tamanho.

Sel[2:0] _ Data[7:0] _ Y
000_0000000_0
000_0000001_1
000_0000010_0
000_0000011_1
000_0000100_0
000_0000101_1
000_0000110_0
000_0000111_1
000_0001000_0
000_0001001_1
000_0001010_0
000_0001011_1
000_0001100_0
000_0001101_1
000_0001110_0
000_0001111_1
000_00010000_0
000_00010001_1
000_00010010_0
000_00010011_1
000_00010100_0
000_00010101_1
000_00010110_0

```

000_00010111_1
000_00011000_0
000_00011001_1
000_00011010_0
000_00011011_1
000_00011100_0
000_00011101_1
000_00011110_0
000_00011111_1
000_00100000_0
000_00100001_1
000_00100010_0
000_00100011_1
000_00100100_0
000_00100101_1
000_00100110_0
000_00100111_1
000_00101000_0
000_00101001_1
000_00101010_0
000_00101011_1
000_00101100_0
000_00101101_1
000_00101110_0
000_00101111_1
000_00110000_0
000_00110001_1
000_00110010_0
000_00110011_1
000_00110100_0
000_00110101_1
000_00110110_0
000_00110111_1
000_10001101_1
000_10001110_0
000_10001111_1

```

Simulação

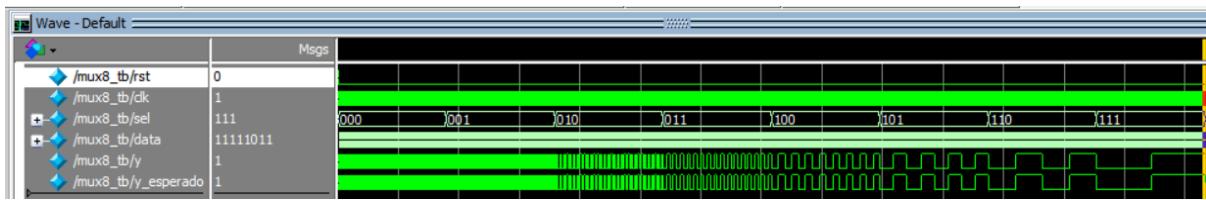
Para todos os 2048 casos executados não foi acusado nenhum erro, como ilustra a Figura a seguir.

```

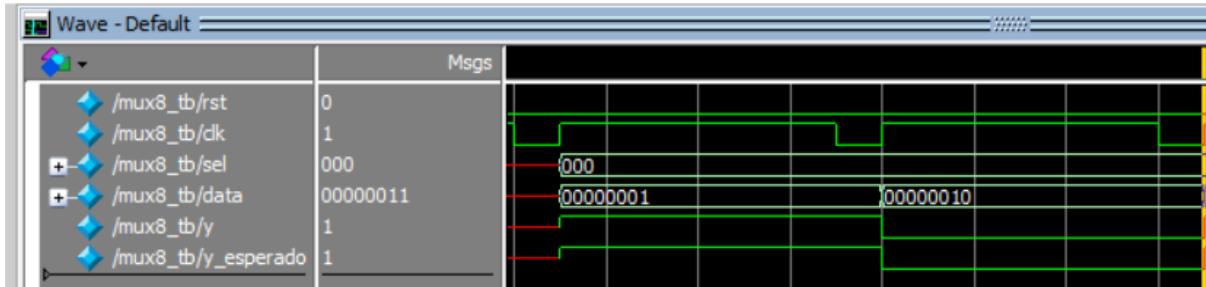
# | 111 | 11110000 | 1 | OK
# | 111 | 11110001 | 1 | OK
# | 111 | 11110010 | 1 | OK
# | 111 | 11110011 | 1 | OK
# | 111 | 11110100 | 1 | OK
# | 111 | 11110101 | 1 | OK
# | 111 | 11110110 | 1 | OK
# | 111 | 11110111 | 1 | OK
# | 111 | 11111000 | 1 | OK
# | 111 | 11111001 | 1 | OK
# | 111 | 11111010 | 1 | OK
# | 111 | 11111011 | 1 | OK
# | 111 | 11111100 | 1 | OK
# | 111 | 11111101 | 1 | OK
# | 111 | 11111110 | 1 | OK
# | 111 | 11111111 | 1 | OK
# Testes Efetuados = 2048
# Erros Encontrados = 0

```

Os sinais ficaram comprimidos uma vez que temos muitas amostras. Por isso a visualização do comportamento do circuito não dá para ser realizada a partir desta Figura abaixo.



No entanto a próxima Figura ilustra melhor a alteração a saída em função das entradas. Com a chave de seleção em 000, a saída deve receber o data[0]. Nesse caso, quando o valor do data[0] é alterado de 1 para 0, a saída também o acompanha.



Mux: 32-1 (1 bit)

Foi elaborado um mux de 32x1, com 4 instâncias dos mux 8_1 e uma do 4_1, ambos previamente modelado.

Descrição em System Verilog

```
module mux32 (d, Sel, y);

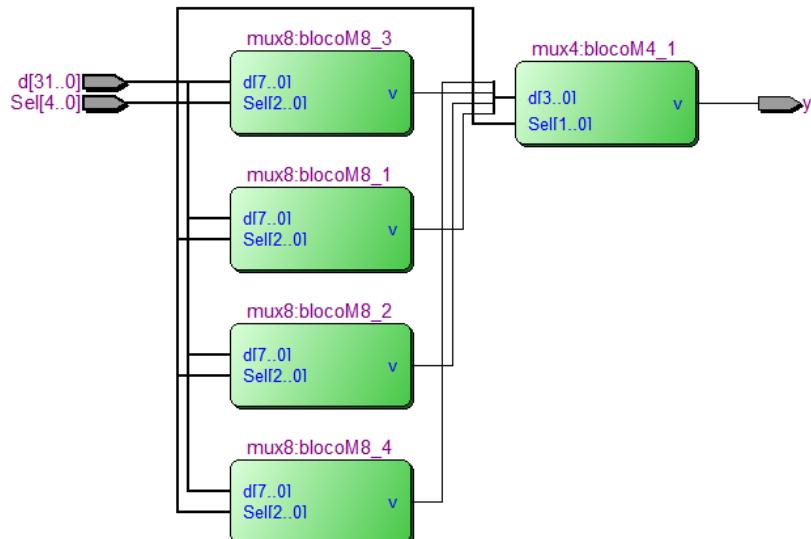
    input logic [31:0] d;
    input logic [4:0] Sel;
    output logic y;
    logic [3:0] intermediario;

    mux8 blocoM8_1(d[7:0], Sel[2:0], intermediario[0]);
    mux8 blocoM8_2(d[15:8], Sel[2:0], intermediario [1]);
    mux8 blocoM8_3(d[23:16], Sel[2:0], intermediario[2]);
    mux8 blocoM8_4(d[31:24], Sel[2:0], intermediario [3]);

    mux4 blocoM4_1 (intermediario[3:0], Sel[4:3], y);

endmodule
```

Visão RTL



Golden Model

```
int main()
{
    FILE *arquivo;
    int y = 0;
    int a = 0;
    int i = 0;
    arquivo = fopen("../Simulation/ModelSim/mux32.tv", "w");

    int Sel[5] = {0}, data[32] = {1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0};
    //Testa uma entrada para todas as variações de chaves
    for (Sel[4] = 0; Sel[4] < 2; Sel[4]++)
        for (Sel[3] = 0; Sel[3] < 2; Sel[3]++)
            for (Sel[2] = 0; Sel[2] < 2; Sel[2]++)
                for (Sel[1] = 0; Sel[1] < 2; Sel[1]++)
                    for (Sel[0] = 0; Sel[0] < 2; Sel[0]++)
                    {
                        a = Sel[0] * (1) + Sel[1] * (2) + Sel[2] * (4) + Sel[3] * (8) + Sel[4] * 16;
                        y = data[a];
                        fprintf(arquivo, "%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n", Sel[4], Sel[3], Sel[2], Sel[1], Sel[0], data[24], data[23], data[22], data[21], data[20], data[19], data[18], data[17], data[16], data[15], data[14], data[13], data[12], data[11], data[10], data[9], data[8], data[7], data[6], data[5], data[4], data[3], data[2], data[1], data[0], y);
                    }
    fclose(arquivo);
}
```

Vetores de Teste

Sel[4:0]_Data[31:0]_Y
00000_01010101011111000101010101010101_1
00001_01010101011111000101010101010101_0
00010_01010101011111000101010101010101_1
00011_01010101011111000101010101010101_0
00100_01010101011111000101010101010101_1
00101_01010101011111000101010101010101_0
00110_01010101011111000101010101010101_1
00111_01010101011111000101010101010101_0
01000_01010101011111000101010101010101_1
01001_01010101011111000101010101010101_0
01010_01010101011111000101010101010101_1
01011_01010101011111000101010101010101_0
01100_01010101011111000101010101010101_1
01101_01010101011111000101010101010101_0
01110_01010101011111000101010101010101_1
01111_01010101011111000101010101010101_0
10000_01010101011111000101010101010101_0
10001_01010101011111000101010101010101_0
10010_01010101011111000101010101010101_1
10011_01010101011111000101010101010101_1
10100_01010101011111000101010101010101_1
10101_01010101011111000101010101010101_1
10110_01010101011111000101010101010101_1
10111_01010101011111000101010101010101_0
11000_01010101011111000101010101010101_1
11001_01010101011111000101010101010101_0
11010_01010101011111000101010101010101_1
11011_01010101011111000101010101010101_0
11100_01010101011111000101010101010101_1

```

11101_0101010101111100010101010101010101_0
11110_01010101011111000101010101010101_1
11111_01010101011111000101010101010101_0

```

TestBench

```

`timescale 1ns/100ps
module mux32_tb;
    int counter, errors, aux_error;
    logic clk,rst;
    logic [31:0]data;
    logic [4:0]sel;
    logic y, y_esperado;
    logic [37:0]vectors[32];

    mux32 DUV(data[31:0], sel[4:0], y);

    initial begin
        $display("Iniciando Testbench");
        $display(" | Sel |          data           | Y |");
        $display("-----");
        $readmemb("C:/Users/satc1/OneDrive/Documentos/Concepcao/MIPS/Mux32-1/Simulation/ModelSim/mux32.tv",vectors);
        counter=0; errors=0;
        rst = 1; #15; rst = 0;
    end

    always begin
        clk=0; #15;
        clk=1; #15;
    end

    always @ (posedge clk) begin
        if(~rst)begin
            {sel[4:0], data[31:0], y_esperado} = vectors[counter];
        end
    end

    always @ (negedge clk) //Sempre (que o clock descer)
        if(~rst)
        begin
            aux_error = errors;
            assert (y == y_esperado)
        end
        else
        begin
            errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
            end
        if(aux_error == errors)
            $display(" | %b | %b | %b | OK", sel[4:0], data[31:0], y);
        else
            $display(" | %b | %b | %b | ERRO", sel[4:0], data[31:0], y);

        counter++; //Incrementa contador dos vetores de teste

        if(counter == $size(vectors)) //Quando os vetores de teste acabarem
        begin
            $display("Testes Efetuados = %0d", counter);
            $display("Erros Encontrados = %0d", errors);
            #10
            $stop;
        end
    end
endmodule

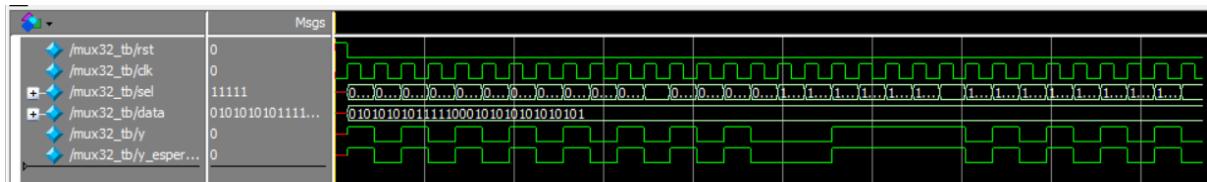
```

Simulação

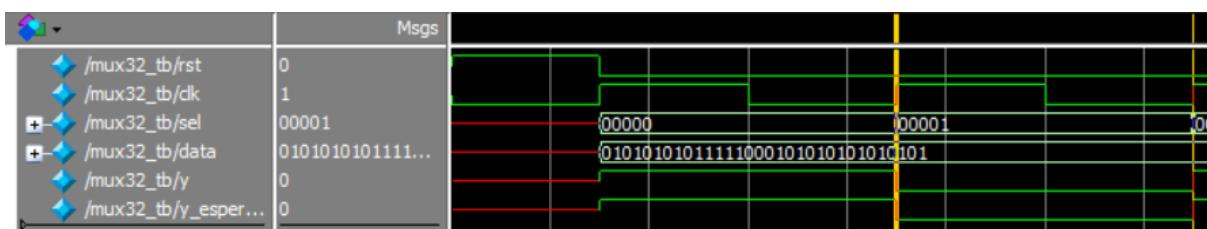
```

VSIM 66> run -all
# Iniciando Testbench
# | Sel | Data | Y |
# -----
# | 00000 | 010101010111100001010101010101 | 1 | OK
# | 00001 | 010101010111100001010101010101 | 0 | OK
# | 00010 | 010101010111100001010101010101 | 1 | OK
# | 00011 | 010101010111100001010101010101 | 0 | OK
# | 00100 | 010101010111100001010101010101 | 1 | OK
# | 00101 | 010101010111100001010101010101 | 0 | OK
# | 00110 | 010101010111100001010101010101 | 1 | OK
# | 00111 | 010101010111100001010101010101 | 0 | OK
# | 01000 | 010101010111100001010101010101 | 1 | OK
# | 01001 | 010101010111100001010101010101 | 0 | OK
# | 01010 | 010101010111100001010101010101 | 1 | OK
# | 01011 | 010101010111100001010101010101 | 0 | OK
# | 01100 | 010101010111100001010101010101 | 1 | OK
# | 01101 | 010101010111100001010101010101 | 0 | OK
# | 01110 | 010101010111100001010101010101 | 1 | OK
# | 01111 | 010101010111100001010101010101 | 0 | OK
# | 10000 | 010101010111100001010101010101 | 0 | OK
# | 10001 | 010101010111100001010101010101 | 0 | OK
# | 10010 | 010101010111100001010101010101 | 1 | OK
# | 10011 | 010101010111100001010101010101 | 1 | OK
# | 10100 | 010101010111100001010101010101 | 1 | OK
# | 10101 | 010101010111100001010101010101 | 1 | OK
# | 10110 | 010101010111100001010101010101 | 1 | OK
# | 10111 | 010101010111100001010101010101 | 0 | OK
# | 11000 | 010101010111100001010101010101 | 1 | OK
# | 11001 | 010101010111100001010101010101 | 0 | OK
# | 11010 | 010101010111100001010101010101 | 1 | OK
# | 11011 | 010101010111100001010101010101 | 0 | OK
# | 11100 | 010101010111100001010101010101 | 1 | OK
# | 11101 | 010101010111100001010101010101 | 0 | OK
# | 11110 | 010101010111100001010101010101 | 1 | OK
# | 11111 | 010101010111100001010101010101 | 0 | OK
# Testes Efetuados = 32
# Erros Encontrados = 0

```



Quando a chave de seleção é 000 a saída equivale ao MSB da entrada, ou seja, o 1. Quando a chave muda de valor para 001, o bit do data[1] é 0, por isso a saída fica em off.



Decodificador: 5x32

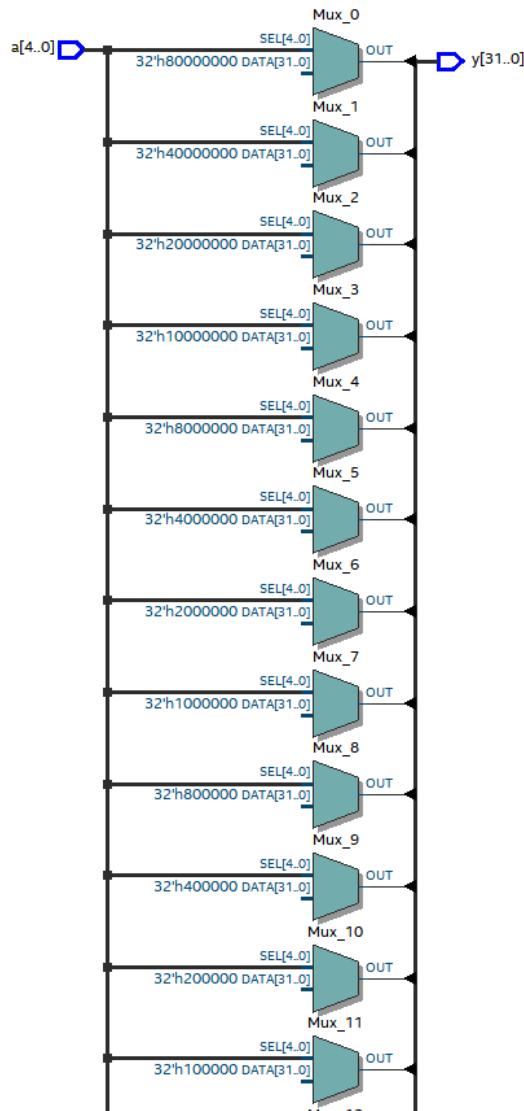
Este decodificador apresenta uma saída com 32 bits para cada combinação dos 5 bits de entrada. De acordo com o sinal de entrada, este componente ativa exatamente uma de suas saídas, sentando o bit equivalente a à sua entrada, ou seja, caso a entrada seja $01010_2=10_{10}$ a sua saída será um array de 32 bits com apenas o bit da posição 10 em 1, enquanto os outros permanecem em 0.

Descrição em System Verilog

A descrição do modelo em system verilog foi implementada utilizando a lógica do case, no qual todas as possíveis entradas são indicadas para as suas respectivas saídas. Neste caso foi utilizado um vetor de 5 bits denominado ‘a’ para indicar a entrada e ‘y’ com 32 bits para representar a saída. Os bits foram organizados do MSB para o LSB.

Visão RTL

Uma visão reduzida do esquemático RTL é apresentada na Figura abaixo. Pelo fato de possuir 32 saídas, o esquemático ficou extenso, por este modo foi colocado apenas uma parte. No entanto, o comportamento do modelo é similar ao que é apresentado na Figura. A entrada 'a' através dos mux seleciona a saída correta do componente, de acordo com o sinal presente na entrada.



Golden Model

O modelo de ouro deste componente foi gerado em C. A lógica de implementação utilizada foi a seguinte, na função main do programa são gerados fors de modo recursivo para simular a variação de todos os bits de entrada. Ao chegar no escopo do processo do bit MLB é chamada a função Calcula_Decimal que, dado os valores dos bits de entradas, retorna a posição do bit que deve ser setado no array de saída. Desta forma, com os dados de entrada e de saída, os vetores de teste podem ser gerados em um arquivo separado.

```
C:\> Users > satc1 > OneDrive > Documentos > Concepcao > Decoder_5x32 > GoldenModel > C main.c > ...
1 #include <stdio.h>
2 #include <math.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 #define peso_A 4
7 #define peso_B 3
8 #define peso_C 2
9 #define peso_D 1
10 #define peso_E 0
11
12 int calcula_decimal(int a, int b, int c, int d, int e){
13     int resultado=0;
14     resultado= (a*pow(2,peso_A))+ (b*pow(2,peso_B)) + (c*pow(2,peso_C))+ (d*pow(2,peso_D)) + (e*pow(2,peso_E));
15     printf("Decimal: %d\n", resultado);
16     return resultado;
17 }
18
int main(void){
    FILE *arquivo;
    arquivo = fopen("../Simulation/ModelSim/deco.tv", "w");
    char bits[32]={0};
    int set=0;
    int f=0;
    int a,b,c,d,e;
    for(a=0; a<2; a++){
        for(b=0; b<2; b++){
            for(c=0;c<2; c++){
                for(d=0; d<2; d++){
                    for(e=0; e<2; e++){
                        set = calcula_decimal(a,b,c,d,e);
                        bits[set]=1;
                        fprintf(arquivo, "%d%d%d%d%d ", a, b, c, d, e);
                        for(int i=31; i>=0; i--){
                            f=(int)bits[i];
                            fprintf(arquivo, "%d",f);
                            bits[i]=0;
                        }
                        fprintf(arquivo, "\n");
                    }
                }
            }
        }
    }
    return 0;
}
```

Vetores de teste

Os vetores de teste gerados pelo modelo de ouro são apontados na tabela abaixo. Tais valores, de modo análogo as descrições dos componentes anteriores, serão utilizadas na simulação do arquivo de testbench para servir como referência para validar os resultados obtidos na descrição em verilog do arquivo.

Input _ Output
5 bits 32 bits
00000_000000000000000000000000000000001
00001_0000000000000000000000000000000010
00010_00000000000000000000000000000000100
00011_000000000000000000000000000000001000
00100_0000000000000000000000000000000010000
00101_00000000000000000000000000000000100000
00110_000000000000000000000000000000001000000
00111_0000000000000000000000000000000010000000
01000_00000000000000000000000000000000100000000
01001_000000000000000000000000000000001000000000
01010_0000000000000000000000000000000010000000000
01011_0000000000000000000000000000000010000000000
01100_00000000000000000000000000000000100000000000
01101_000000000000000000000000000000001000000000000
01110_000000000000000000000000000000001000000000000
01111_000000000000000000000000000000001000000000000
10000_000000000000000010000000000000000000000
10001_000000000000000010000000000000000000000
10010_000000000000000010000000000000000000000
10011_000000000000000010000000000000000000000
10100_000000000000000010000000000000000000000
10101_000000000000100000000000000000000000000
10110_000000000000000010000000000000000000000
10111_000000000000000010000000000000000000000
11000_000000010000000000000000000000000000000
11001_000000010000000000000000000000000000000
11010_000000010000000000000000000000000000000
11011_000000010000000000000000000000000000000
11100_00010000000000000000000000000000000000000
11101_00100000000000000000000000000000000000000
11110_01000000000000000000000000000000000000000
11111_100

Arquivo de TestBench

O arquivo de TestBench foi realizado de modo similar aos outros arquivos de testes já descritos neste relatório. As alterações realizadas baseiam-se na definição das variáveis que representam a entrada e a saída do componente, neste caso o ‘a’ e o ‘y’. E a alteração dos parâmetros de concatenação da variável ‘vectors’, na qual se utiliza a entrada e o valor esperado da saída (equivalente ao descrito nos vetores de ouro).

```

`timescale 1ns/100ps
module deco_tb;
    int counter, errors, aux_error;
    logic clk,rst;
    logic [4:0] a;
    logic [31:0]y;
    logic [31:0]y_esperado;
    logic [36:0]vectors[32];

    decoder DUV(a,y);

    initial begin
        $display("Iniciando Testbench");
        $display(" | Input | Output");
        $display("-----");
        $readmem("C:/Users/satc1/OneDrive/Documentos/Concepcao/Decoder_5x32/Simulation/ModelSim/deco.tv",vectors);
        counter=0; errors=0;
        rst = 1; #10; rst = 0;
    end

    always begin
        clk=1; #30;
        clk=0; #5;
    end

    always @ (posedge clk) begin
        if(~rst)begin
            {a, y_esperado} = vectors[counter];
        end
    end

    always @ (negedge clk) //Sempre (que o clock descer)
        if(~rst)
        begin
            aux_error = errors;
            assert (y == y_esperado)
        end
        else
        begin
            errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
            end
        if(aux_error == errors)
            $display(" | %b | %b | OK",a, y);
        else
            $display(" | %b | %b | ERRO", a, y);

        counter++; //Incrementa contador dos vetores de teste
    end
    if(counter == $size(vectors)) //Quando os vetores de teste acabarem
    begin
        $display("Testes Efetuados = %0d", counter);
        $display("Erros Encontrados = %0d", errors);
        #10
        $stop;
    end
endmodule

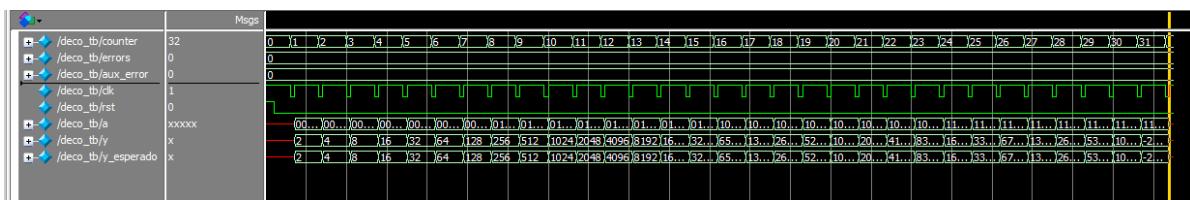
```

Simulação

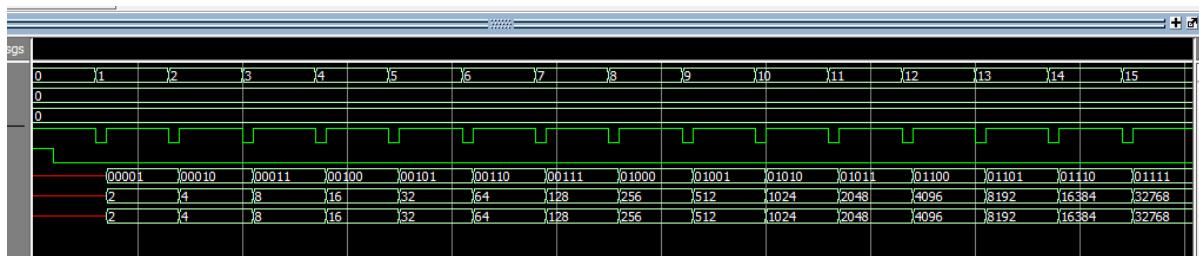
O resultado obtido através da simulação do ModelSim válida a descrição do decodificador, uma vez que não foram encontrados erros durante a comparação das saídas do modelo em systemverilog com os valores dos vetores de referência.

```
Transcript
add wave -position end sim:/deco_tb/y
add wave -position end sim:/deco_tb/y_esperado
VSIM 10> run -all
# Iniciando Testbench
# | Input | Output
# -----
# | xxxxx |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx | OK
# | 00001 | 000000000000000000000000000000010 | OK
# | 00010 | 0000000000000000000000000000000100 | OK
# | 00011 | 00000000000000000000000000000001000 | OK
# | 00100 | 000000000000000000000000000000010000 | OK
# | 00101 | 0000000000000000000000000000000100000 | OK
# | 00110 | 00000000000000000000000000000001000000 | OK
# | 00111 | 000000000000000000000000000000010000000 | OK
# | 01000 | 0000000000000000000000000000000100000000 | OK
# | 01001 | 00000000000000000000000000000001000000000 | OK
# | 01010 | 00000000000000000000000000000001000000000 | OK
# | 01011 | 00000000000000000000000000000001000000000 | OK
# | 01100 | 00000000000000000000000000000001000000000 | OK
# | 01101 | 00000000000000000000000000000001000000000 | OK
# | 01110 | 00000000000000000000000000000001000000000 | OK
# | 01111 | 00000000000000000000000000000001000000000 | OK
# | 10000 | 00000000000000000000000000000001000000000 | OK
# | 10001 | 00000000000000000000000000000001000000000 | OK
# | 10010 | 00000000000000000000000000000001000000000 | OK
# | 10011 | 00000000000000000000000000000001000000000 | OK
# | 10100 | 00000000000000000000000000000001000000000 | OK
# | 10101 | 00000000000000000000000000000001000000000 | OK
# | 10110 | 00000000000000000000000000000001000000000 | OK
# | 10111 | 00000000000000000000000000000001000000000 | OK
# | 11000 | 00000001000000000000000000000000000000000 | OK
# | 11001 | 00000001000000000000000000000000000000000 | OK
# | 11010 | 00000100000000000000000000000000000000000 | OK
# | 11011 | 00001000000000000000000000000000000000000 | OK
# | 11100 | 00010000000000000000000000000000000000000 | OK
# | 11101 | 00100000000000000000000000000000000000000 | OK
# | 11110 | 01000000000000000000000000000000000000000 | OK
# | 11111 | 10000000000000000000000000000000000000000 | OK
# Testes Efetuados = 32
# Erros Encontrados = 0
```

O comportamento dos sinais é apresentado na Figura abaixo, na qual podemos perceber que a cada variação da entrada o valor exibido em y é realmente igual ao valor esperado.

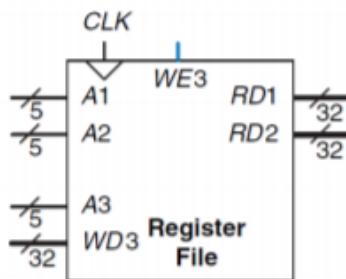


A análise dos sinais pode ser feita de forma mais pontual na Figura abaixo, que apresenta um zoom no início na simulação. A cada pulso de clock o valor da saída é alterado e permanece equivalente ao esperado.



Banco de Registradores (1 bit)

A arquitetura MIPS utiliza um banco de registradores para executar operações de leitura dos dados anteriormente gravados e de escrita de dados para modificar as informações internas. O banco de registradores tem dois portos de leitura (A1/RD1 e A2/RD2) e uma porta de escrita (A3/WD3). A Figura abaixo ilustra um banco de registradores com 2 portas de leitura e uma de escrita. No entanto é válido ressaltar que a Figura abaixo caracteriza um banco de dados de 32 bits com 32 registradores, nesta Seção do relatório está sendo desenvolvido o banco com 32 registradores de 1 bit. Desse modo o Rd1 e Rd2 serão apenas 1 bit.



As portas e suas respectivas funções do banco de registradores são descritas abaixo:

- A1, A2 - portas de entradas que identificam os registradores a serem lidos;
- A3 - porta de entrada que identifica o registrador de escrita;
- RD1 e RD2 - duas portas de leituras que mostram os dados armazenados nos registradores indicados por A1 ou A2;
- WD3 - Dados a serem gravados no registrador;
- We3 - enable para ativar a operação de escrita.

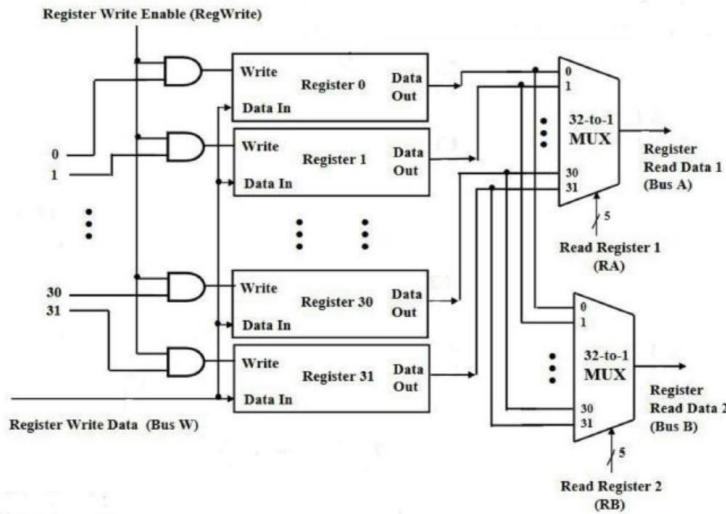
A lógica de organização de um banco de registradores consiste na ligação lógica de diversos registradores, neste caso, flip-flops do tipo enabled de 1 bit. Estes registradores devem ter sinais de controle em comum, como os sinais de relógio (clock), de carga (load) e de apagamento do conteúdo (clear).

A Figura a seguir exemplifica a modelagem do **sistema de escrita** de modo mais detalhado. Na qual, a entrada *write* desses registradores são operações lógicas entre o WE3 e os sinais de entrada da variável A3.

No entanto, A3 estará ligada a um decodificador 5x32 para transformar os 5 bits de entradas em 32 saídas. Desse modo, será a saída do decodificador que entrará na and com o WE3. Desse modo caso o *Write* esteja em nível lógico alto e o registrador esteja na borda de subida o dado oriundo do *data in* pode ser escrito e atualizado na saída *Data out*.

Dessa forma, todos os *Data outs* são conectados a um multiplexador de 32 canais, o qual será controlado pelas variáveis de entrada A1 e A2, assim como ilustra a Figura anterior. A saída do multiplexador, será então, o bit gravado no registrador de endereço indicado pelas chaves de seleção.

Como temos duas chaves e dois multiplexadores de 32 bits, podemos ler dois dados ao mesmo tempo, pois pode-se passar valores diferentes para A1 e A2 ao mesmo tempo.



Descrição em System Verilog

O código seja a mesma lógica da Figura anterior. Inicialmente os dados do A3 (5bits) são colocados em um decodificador 5x32 (linha 12 do código) com o intuito de estender a largura da palavra. Em seguida é realizada uma operação de AND entre o enable da operação write, WE3, e os dados de saída do decodificador. Para que desse modo seja verificado o enable do registrador (linha 17).

Nas linhas 16-18 foi realizado um laço de repetição que instancia 32 flip-flops, passando as respectivas saídas e entradas declaradas na Figura anterior. Por fim, são instanciados os 2 multiplexadores de 32 bits (linha 23 e 24), cuja as saídas resultam na leitura dos dados salvos nos registradores cujos endereços são selecionados pelas chaves A1 e A2.

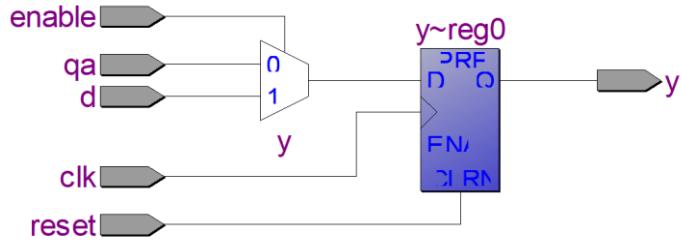
```

1  module banco_reg(clk, rst, A1, A2, A3, RD1, RD2, WD3, WE3);
2    input logic clk, WE3, rst;
3    input logic [4:0] A1, A2, A3;
4    input logic [31:0] WD3;
5    output logic RD1, RD2;
6
7    logic [31:0] output_decoder;
8    logic [31:0] and_output;
9    logic [31:0] reg_output;
10   logic [31:0] qa;
11
12  decoder decoder_block(A3,output_decoder); //Prepara a entrada da AND
13
14  genvar i;
15  generate
16    for (i=0; i<32;i=i+1) begin: forBancoReg
17      and (and_output[i],output_decoder[i], WE3);
18      flopenr block_reg(clk, rst, and_output[i], WD3[i], reg_output[i],qa[i]);
19      assign qa[i] = reg_output[i];
20    end
21  endgenerate
22
23  mux32 muxA(reg_output[31:0], A1[4:0],RD1);
24  mux32 muxB(reg_output[31:0], A2[4:0],RD2);
25
26 endmodule

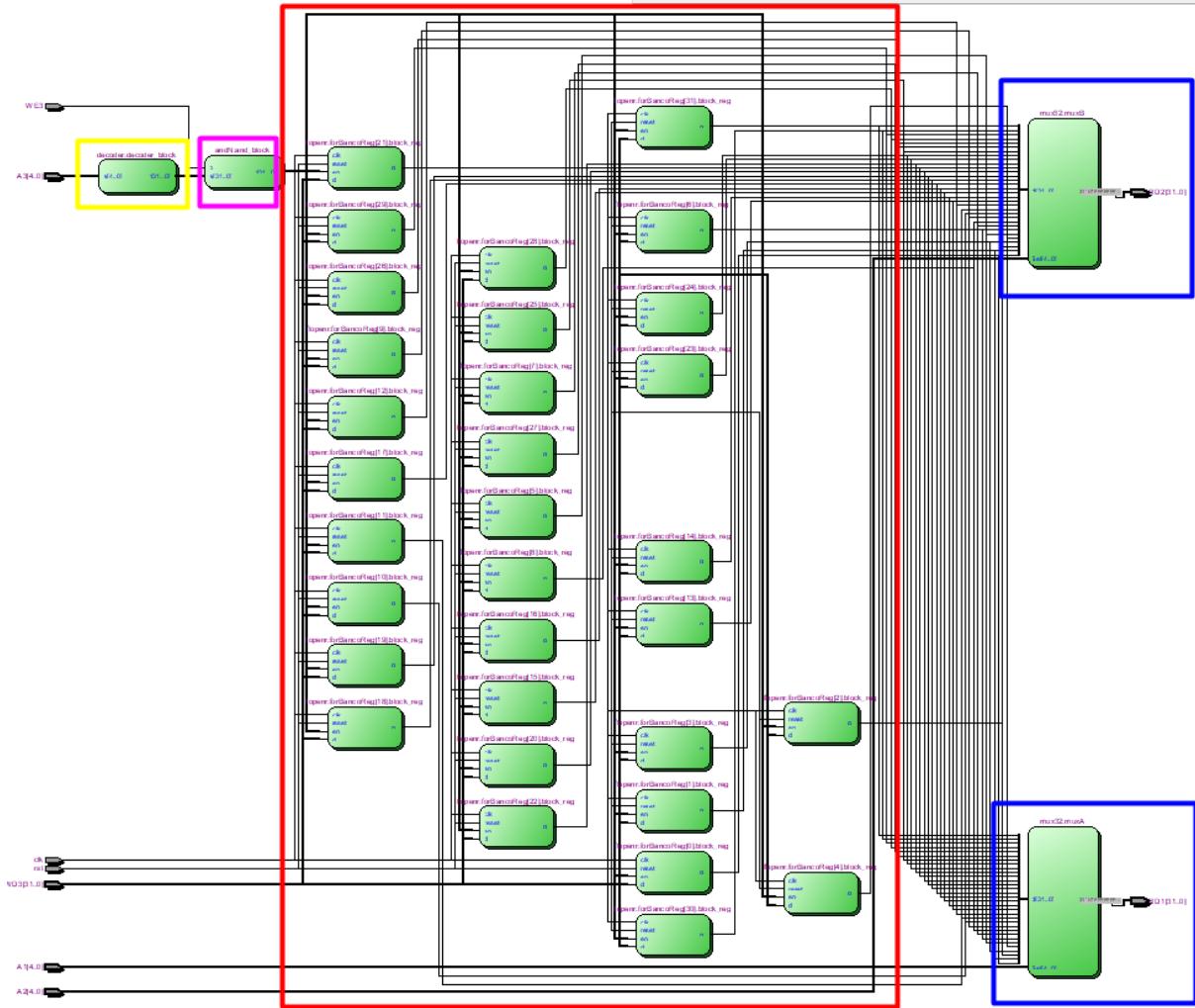
```

Visão RTL

O flip flop utilizado neste banco foi alterado para caso o seu enable seja 0, retornar a saída anterior. Desta forma, caso o valor do seu enable (oriundo da operação lógica de and entre o We3 e a saída do decodificador) seja falso, ou seja, esteja sendo realizada uma operação de leitura, a saída permanece com os dados que foram gravados anteriormente. Desta forma, o RTL de seus registradores ficou da seguinte forma:



A visão ampla do RTL é mostrada abaixo, na qual podemos perceber a instância de todos os blocos. O bloco em destaque na cor amarela é o decodificador. Em rosa, o módulo de AND. Em vermelho, todos os 32 registradores. E por fim, em azul, os 2 multiplexadores de 32 bits.



Golden Model

No início do código é definida um array de bits para representar o Wd3 durante toda a execução do código. Dessa forma ele é um parâmetro constante, neste código. Pois variar todos os seus bits iria acarretar em um esforço computacional considerável, uma vez que seriam mais 32 laços de repetições aninhados a sequência recursiva já existente. O modelo do Golden Model consiste na variação recursiva de todos os bits do reset, A1, A2, A3, reset, We3 e do clock, respectivamente. Tais laços de repetição são ilustrados abaixo.

```
for (rst = 0; rst < 2; rst++){
    for (int f = 0; f < 2; f++){ //A1
        for (int g = 0; g < 2; g++)
            for (int h = 0; h < 2; h++)
                for (int j = 0; j < 2; j++)
                    for (int k = 0; k < 2; k++) //A2
                        for (int l = 0; l < 2; l++) //A3
                            for (int m = 0; m < 2; m++)
                                for (int n = 0; n < 2; n++)
                                    for (int o = 0; o < 2; o++)
                                        for (int p = 0; p < 2; p++) //A3
                                            for (a = 0; a < 2; a++) //A3
                                                for (b = 0; b < 2; b++)
                                                    for (c = 0; c < 2; c++)
                                                        for (d = 0; d < 2; d++)
                                                            for (e = 0; e < 2; e++)
                                                                for (WE3 = 0; WE3 < 2; WE3++)
                                                                    for (clk = 0; clk < 2; clk++)
{
    A1[4] = f; A1[3] = g; A1[2] = h; A1[1] = j; A1[0] = k;
    A2[4] = l; A2[3] = m; A2[2] = n; A2[1] = o; A2[0] = p;
    A3[4] = a; A3[3] = b; A3[2] = c; A3[1] = d; A3[0] = e;
```

Dentro do for mais interno realiza-se a impressão das variáveis do reset e clock. Em seguida, é chamado o decodificador 5x32, o qual possui como parâmetros de entrada os 5 bits do A3. E são impressos mais alguns dados no arquivo.

```
fprintf(arquivo, "%d_%d_", rst, clk);

bits = decoder(A3[4], A3[3], A3[2], A3[1], A3[0]);

fprintf(arquivo, "%d%d%d%d%d_d_%d%d%d%d%d_d_",
        A1[4], A1[3], A1[2], A1[1], A1[0], A2[4], A2[3], A2[2], A2[1], A2[0]);
fprintf(arquivo, "%d%d%d%d%d_d_",
        A3[4], A3[3], A3[2], A3[1], A3[0]);
fprintf(arquivo, "%d_",
        WE3);

for (int i = 0; i < 32; i++)
{
    and[i] = bits[i] & WE3;
    //fprintf(arquivo, "%d", bits[i]); //Se o enable estiver off, sai 0 -- enable 1 recebe a saída do decodificador..
    //bits = saída do decodificador
}
//fprintf(arquivo, " ");

for (int i = 0; i < 32; i++)
    fprintf(arquivo, "%d", WD3[i]); //entrada
```

Por fim, os 32 registradores são instanciados como blocos de flip-flops enabled e são calculados os endereços dos registradores com base na entrada das chaves A1 e A2 para que as saídas dos multiplexadores possam retornar os dados escritos nessas respectivas posições. A Figura abaixo ilustra o término do programa.

```

for (int i = 0; i < 32; i++)
{
    output_reg[i] = fopenr(clk, rst, and[i], WD3[i], qa[i]);
    qa[i] = output_reg[i];
    //fprintf(arquivo, "%d", output_reg[i]); //Sai a entrada qnd o WE3=1 & CLK=1 & RST=0
}

//for (int i = 0; i < 32; i++)
//    fprintf(arquivo, "%d", output_reg[i]); //Sai um qnd o WE3=1 & CLK=1 & RST=0
pos = mux32(A1[4], A1[3], A1[2], A1[1], A1[0]);
RD1 = output_reg[31 - pos];
pos = mux32(A2[4], A2[3], A2[2], A2[1], A2[0]);
RD2 = output_reg[31 - pos];
if (count ≥ 3)
    fprintf(arquivo, "%d_%d\n", RD1, RD2); //Sai um qnd o WE3=1 & CLK=1 & RST=0
else
    fprintf(arquivo, "x_x\n"); //Esses sao os 3 primeiros casos onde o Qa é X

```

Vetores de Teste

Os vetores de testes consistiram em um arquivo com **262144** linhas, cada qual com 52 bits de dados, desse modo não é viável anexá-los a esse relatório. No entanto, a saída do RD1 e RD2 é 1 quando o rst é 0, clk é 1, e o valor da and com o WE3 e os valores do A3 equivalem a nível lógico alto.

A análise de alguns vetores é dada a seguir. Na Figura abaixo, são impressos os bits na seguinte ordem: reset, clock, A1, A2, A3, We3, saída do decodificador, Wd3, saída dos registradores, Rd1 e Rd2.

```
4  0_1_00000_00000_00000_1_000000000000000000000000000000001_10011001010110101100110100010101_0000000000000000000000000000000001_1_1
```

Desse modo com o clock e We3 ativos, o banco vai habilitar a escrita do bit da posição A3 do array Wd3, no registrador de endereço equivalente à saída do decodificador.

Nas linhas de testes consecutivas, houve variação do clock e do enable, no entanto ambos não ficaram em nível lógico alto ao mesmo tempo. O que pode caracterizar a operação de leitura. Nesse caso, a saída permanece a anterior, até que haja uma nova operação de escrita.

```
5  0_0_00000_00000_00001_0_0000000000000000000000000000000010_10011001010110101100110100010101_0000000000000000000000000000000000000001_1_1
6  0_1_00000_00000_00001_0_0000000000000000000000000000000010_10011001010110101100110100010101_0000000000000000000000000000000000000001_1_1
7  0_0_00000_00000_00001_1_0000000000000000000000000000000010_100110010101101100110100010101_0000000000000000000000000000000000000001_1_1
```

Na próxima linha de vetores, o clock ficou em nível lógico alto e o clock em borda de subida. Nesse caso foi gravado o segundo bit menos significativo do WD3, pois o A3 equivale a esta posição.

```
8  0_1_00000_00000_00001_1_0000000000000000000000000000000010_10011001010110101100110100010101_00000000000000000000000000000000000000001_1_1
```

De modo análogo, na subsequente operação de escrita, foi adicionado a próxima posição do vetor de entrada.

```
12  0_1_00000_00000_00010_1_00000000000000000000000000000000100_10011001010110101100110100010101_00000000000000000000000000000000101_1_1
```

Após todas as operações de escrita serem feitas, ou seja, após o A3 ser variado em todas as suas possíveis combinações binárias. As operações de leitura, possuem o seu chaveamento alterado. Na Figura a seguir, estas operações ficam mais claras, pois o na Saída 1, é lido o valor do registrador[3] e na saída 2 o registrador [31].

```
16269  0_0_00011_11111_00011_0_0000000000000000000000000000001000_10011001010110101100110100010101_10011001010110101100110100010101_0_1
```

No entanto, estas saídas de decodificador e registrador, foram colocadas apenas para melhor explicação do funcionamento e debugging do programa. Os vetores de testes são constituídos apenas das entradas e saídas do módulo do banco de registradores. A Tabela a seguir ilustra os mesmos casos de teste das 5 últimas Figuras.

linha	Vetor de referência
04	0_1_00000_00000_00000_1_10011001010110101100110100010101_1_1
05	0_0_00000_00000_00001_0_10011001010110101100110100010101_1_1
06	0_1_00000_00000_00001_0_10011001010110101100110100010101_1_1
07	0_0_00000_00000_00001_1_10011001010110101100110100010101_1_1
08	0_1_00000_00000_00001_1_10011001010110101100110100010101_1_1
12	0_1_00000_00000_00010_1_10011001010110101100110100010101_1_1
16269	0_0_00011_11111_00011_0_10011001010110101100110100010101_0_1

TestBench

O testbench foi realizado de modo similar a todos os outros, como é ilustrado na Figura abaixo. Houve a variação do clock a cada 10 instantes de tempo e a condição de assert foi modificada para verificar se as duas saídas do sistema estão coerentes.

```
1 `timescale 1ns/100ps
2 module bancoReg_tb;
3     int counter, errors, aux_error;
4     logic clk, we3, rst;
5     logic [4:0] A1, A2, A3;
6     logic [31:0] wd3;
7     logic rdi;
8     logic rd2;
9     logic clock, reset;
10    logic rdi_esperado, rd2_esperado;
11
12   logic [51:0]vectors[262144];
13   banco_reg DUV(clk, rst, A1[4:0], A2[4:0], A3[4:0], rd1, rd2, wd3[31:0],we3);
14
15 initial begin
16     $display("          Iniciando Testbench           ");
17     $display(" | rst | clk | en | A1 | A2 | A3 |      WD3      | RD1 | RD2 |");
18     $display("-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-");
19     $readmemb("C:/Users/satc1/OneDrive/Documentos/Concepcao/MIPS-Componentes/Banco_Reg/Simulation/ModelSim/banco_reg.tv",vectors);
20     counter=0; errors=0;
21     reset = 1; #20; reset = 0;
22 end
23
24 always begin
25     clock=0; #10;
26     clock=1; #10;
27 end
28
29 always @ (posedge clock) begin
30     if(~reset)begin
31         {rst, clk, A1[4:0], A2[4:0], A3[4:0], we3, wd3[31:0], rd1_esperado, rd2_esperado} = vectors[counter];
32     end
33 end
34
35 always @ (negedge clock) //Sempre (que o clock descer)
36     if(~reset)
37     begin
38         aux_error = errors;
39         assert ((rd1 == rd1_esperado) && (rd2 == rd2_esperado))
40     end
41     begin
42         errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
43     end
44     if(aux_error == errors)
45         $display(" | %b | OK ",rst, clk, we3, A1[4:0], A2[4:0], A3[4:0], wd3[31:0], rd1, rd2);
46     else
47         //if(aux_error!=errors)
48         $display(" | %b | ERRO ",rst, clk, we3, A1[4:0], A2[4:0], A3[4:0], wd3[31:0], rd1, rd2);
49     counter++; //Incrementa contador dos vetores de teste
50
51     if(counter == $size(vectors)) //Quando os vetores de teste acabarem
52     begin
53         $display("Testes Efetuados = %0d", counter);
54         $display("Erros Encontrados = %0d", errors);
55         #10;
56         $stop;
57     end
58 end
59 endmodule
```


Banco de Registradores (32 bits)

O banco de registradores a ser utilizado no mips deve ser de 32 bits, por tal razão o banco foi modelado para que o Rd1 e Rd2 apresentem 32 bits.

Descrição em System Verilog

Comparando esta modelagem com a implementação do banco de registradores de 1 bit, as alterações feitas, consistem em um aumento na variável que armazena a saída dos registradores. Anteriormente tínhamos a variável *output Reg* como sendo um vetor de 32 bits, no qual os dados eram gravados separadamente em cada um dos bits. Agora, temos a variável *array_reg*, como sendo 32 vetores de 32 bits.

Na Figura abaixo, é realizado 32 instâncias de um registrador de 32 bits abaixo do cálculo da and.

```
module banco_reg_32(clk, rst, A1, A2, A3, RD1, RD2, WD3, WE3);
    input logic clk, WE3, rst;
    input logic [4:0] A1, A2, A3;
    input logic [31:0] WD3;
    output logic [31:0] RD1;
    output logic [31:0] RD2;

    logic [31:0] output_decoder;
    logic [31:0] and_output;

    logic[31:0]array_reg[31:0];

    decoder decoder_block(A3,output_decoder); //Prepara a entrada da AND

    genvar i;
    generate
        for (i=0; i<32;i=i+1) begin: forBancoReg
            and (and_output[i],output_decoder[i], WE3);
            reg32 block_regs (rst, clk, and_output[i], WD3[i],array_reg[i]); //32 instancias dos 32 flips
        end
    endgenerate

    genvar k;
    generate
        for (k=0; k<32;k=k+1) begin: formuxs
            mux32 muxA(array_reg[k], A1[4:0],RD1[k]);
            mux32 muxB(array_reg[k], A2[4:0],RD2[k]);
        end
    endgenerate

endmodule
```

O reg 32 consiste em 32 instâncias de um flip flop de 1 bit, como mostra a figura abaixo.

```
module reg32(rst, clk, enable, d, y);

    input logic [31:0] d;
    input logic clk, rst, enable;
    output logic [31:0] y;
    logic [31:0]qa;

    genvar i;
    generate
        for (i=0; i<32;i=i+1) begin: flips
            flopenr block_reg(clk, rst, enable, d[i], y[i],qa[i]); //instancia do flip
            assign qa[i] = y[i];
        end
    endgenerate
endmodule
```

Deslocador de Bits (32 <<2)

Os shifters deslocam os bits e multiplicam ou dividem por potências de 2. Esse componente desloca um número binário para a esquerda, nesse caso, por isso o L(left) shift e desloca 2 bits.

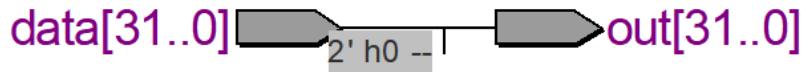
Descrição em System Verilog

Para realizar a implementação do deslocador o Systemverilog dispõe de um operador lógico que realiza essa função. Desse modo , não é necessário instanciar multiplexadores.

```
module LSHIFT2( input [31:0] data, output [31:0] out);
    assign out = data << 2;
endmodule
```

Visão RTL

A visão RTL é bastante simplificada, pois só é utilizado um operador.



Golden Model

O modelo golden model foi escrito em c++ e assim como no systemverilog o operador de deslocamento também existe nesta linguagem de programação. Por isso, o código consiste apenas na impressão e deslocamento dos dados.

```
#include <iostream>
#include <fstream>
#include <bitset>
#include <cstdio>
int main()
{
    std::ofstream saida("l2shift.tv", std::ios::out);
    for (long long i = 0; (long long)i < (long long)((long long)1 << (long long)16); i++)
    {
        saida << std::bitset<32>(i).tostring() << " " << (std::bitset<32>(i) << 2).to_string() << std::endl;
    }
}
```

Vetores de Ouro

Os vetores de referência apresentaram-se em mais de 65 mil linhas, desse modo alguns deles irão estar na figura abaixo, mas não todos.

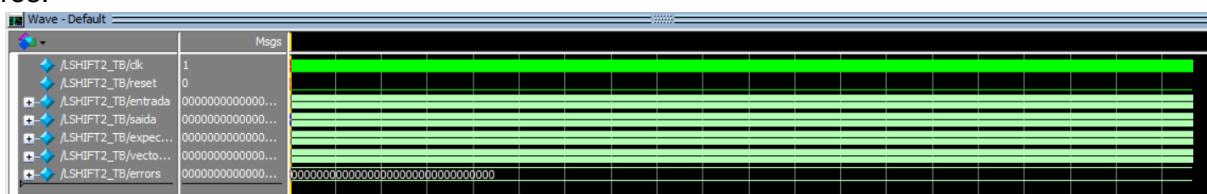
TestBench

O testbench é realizado de modo similar igual a todos os outros.

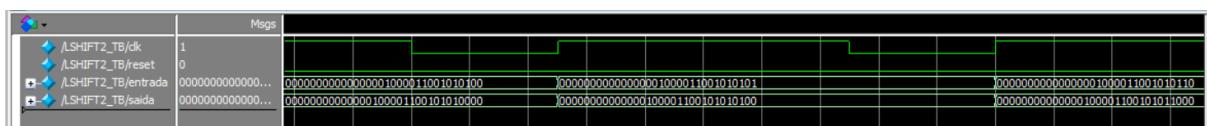
```
module LSHIFT2_TB();
    logic [63:0] testvectors [65536];
    logic clk, reset;
    logic [31:0] entrada;
    logic [31:0] saida;
    logic [31:0] expected_s;
    logic [31:0] vectornum, errors;
    LSHIFT2 DUV(.entrada(input), .saida(saida));
    initial begin
        $readmemb("lshift2.tv", testvectors);
        vectornum = 0;
        errors = 0;
        reset = 1;
        #27;
        reset = 0;
    end
    always begin
        clk = 1; #1000; clk = 0; #500;
    end
    always @ (posedge clk) begin
        #1
        {entrada, expected_s} = testvectors[vectornum];
    end
    always @ (negedge clk)
        if (~reset) begin
            if (saida != expected_s) begin
                $display("Error: inputs = %b", entrada);
                $display(" s = %b (%b expected)", saida, expected_s);
                errors = errors + 1;
            end else begin
                $display("OK: a: %b | saida: %b (%b expected)", entrada, saida, expected_s);
            end
            vectornum = vectornum + 1;
            if (vectornum == $size(testvectors)) begin
                $display("%d tests completed with %d errors", vectornum, errors);
                $stop;
            end
        end
    end
endmodule
```

Simulação

Na simulação todos os 65036 testes dos vetores de testes foram executados com 0 erros.



Na imagem abaixo, fica perceptível a coerência dos dados da simulação.



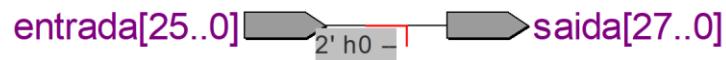
Deslocador de Bits (26 <<2)

Esse deslocador transforma a entrada de 26 bits em 28, a sua descrição em systemverilog é similar a anterior.

Descrição em System Verilog

```
module LSHIFT2_26(input [25:0] entrada, output [27:0] saida);
    assign saida = entrada << 2;
endmodule
```

Visão RTL



Extensor de bits - 16x32

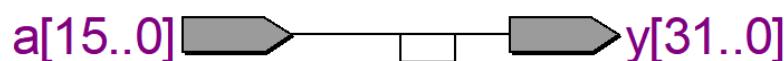
O extensor de bit, recebe uma entrada com 16 bits e a estende para 32 bits.

Descrição em System Verilog

A descrição em systemverilog é realizada de modo simples

```
module extensor( input logic [15:0] a,
                  output logic [31:0] y);
    assign y = {{16{a[15]}}, a};
endmodule
```

Visão RTL



Golden Model

Houve a variação dos 15 bits de entrada através de fors aninhados, e no mais interno os bits foram estendidos pelas operações básicas do c++.

```
outp[6] = inp[6];
for (inp[5] = 0; inp[5] < 2; inp[5]++){
    outp[5] = inp[5];
    for (inp[4] = 0; inp[4] < 2; inp[4]++){
        outp[4] = inp[4];
        for (inp[3] = 0; inp[3] < 2; inp[3]++){
            outp[3] = inp[3];
            for (inp[2] = 0; inp[2] < 2; inp[2]++){
                outp[2] = inp[2];
                for (inp[1] = 0; inp[1] < 2; inp[1]++){
                    outp[1] = inp[1];
                    for (inp[0] = 0; inp[0] < 2; inp[0]++){
                        outp[0] = inp[0];
                        for(int i = 15; i >= 0; i--){
                            out << inp[i];
                        }
                        out << "_";
                        for(int i = 31; i >= 0; i--){
                            out << outp[i];
                        }
                    }
                }
            }
        }
    }
}
```

TestBench

```
`timescale 1 ns / 1 ns
module extensor_tb();
    logic [47:0]testvectors[65536];
    logic clk, reset;
    logic [15:0]a;
    logic [31:0]saida, saidaexperado;
    logic [31:0]vectornum, errors;
    extensor DUV(a, saida);

    initial begin
        $readmemb("extensor.tv", testvectors);
        vectornum = 0;
        errors = 0;
        reset = 1;
        #27;
        reset = 0;
    end

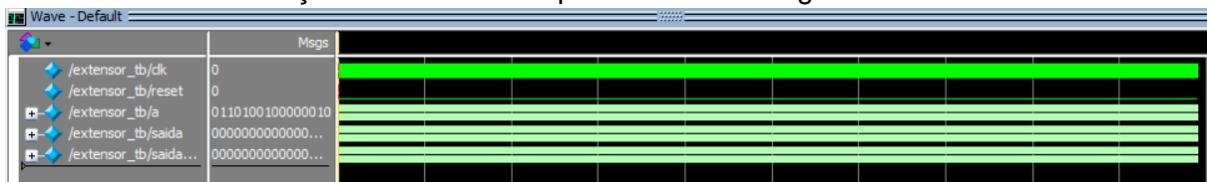
    always begin
        clk = 1; #10; clk = 0; #5;
    end

    always @ (posedge clk) begin
        #1
        {a, saidaexperado} = testvectors[vectornum];
    end// check results on falling edge of clk

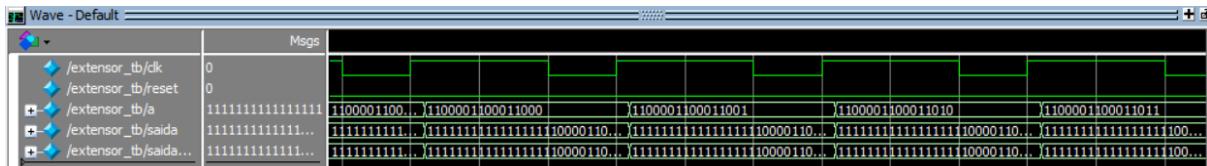
    always @ (negedge clk)
        if (~reset) begin // skip during reset
            if (saida != saidaexperado) begin // check result
                $display("Error: inputs = %b", {a});
                $display(" outputs = %b (%b esperado)", saida, saidaexperado);
                errors = errors + 1;
            end else begin
                $display("OK: a: %b | saida: %b (%b esperado)", a, saida, saidaexperado);
            end
            vectornum = vectornum + 1;
            if (vectornum == $size(testvectors)) begin
                $display("%d tests completed with %d errors", vectornum, errors);
                $finish;
            end
        end
    end
endmodule
```

Simulação

A onda da simulação ficou muito comprimida devido ao grande número de amostras.



Os dados do extensor podem ser visualizados na Figura abaixo, a cada pulso os dados de entrada são estendidos.



ULA (1 bit)

A ULA é uma unidade lógica que realiza operações lógicas e aritméticas como: Xor, AND, OR, ADD, SUB, etc.

Descrição em System Verilog

O código da ULA consiste inicialmente em um bloco always no qual é verificado o tipo de operação que deve ser realizada. Dessa forma, esta verificação ficou como ilustra a Figura abaixo.

```
module ula(input logic A, B, carry_in,less, add_sub,
           input logic[2:0]op,
           output logic Y, cout, s_addsub);

    logic s_B;
    logic s_and;
    logic s_or;
    logic s_xor;
    logic s_nor;

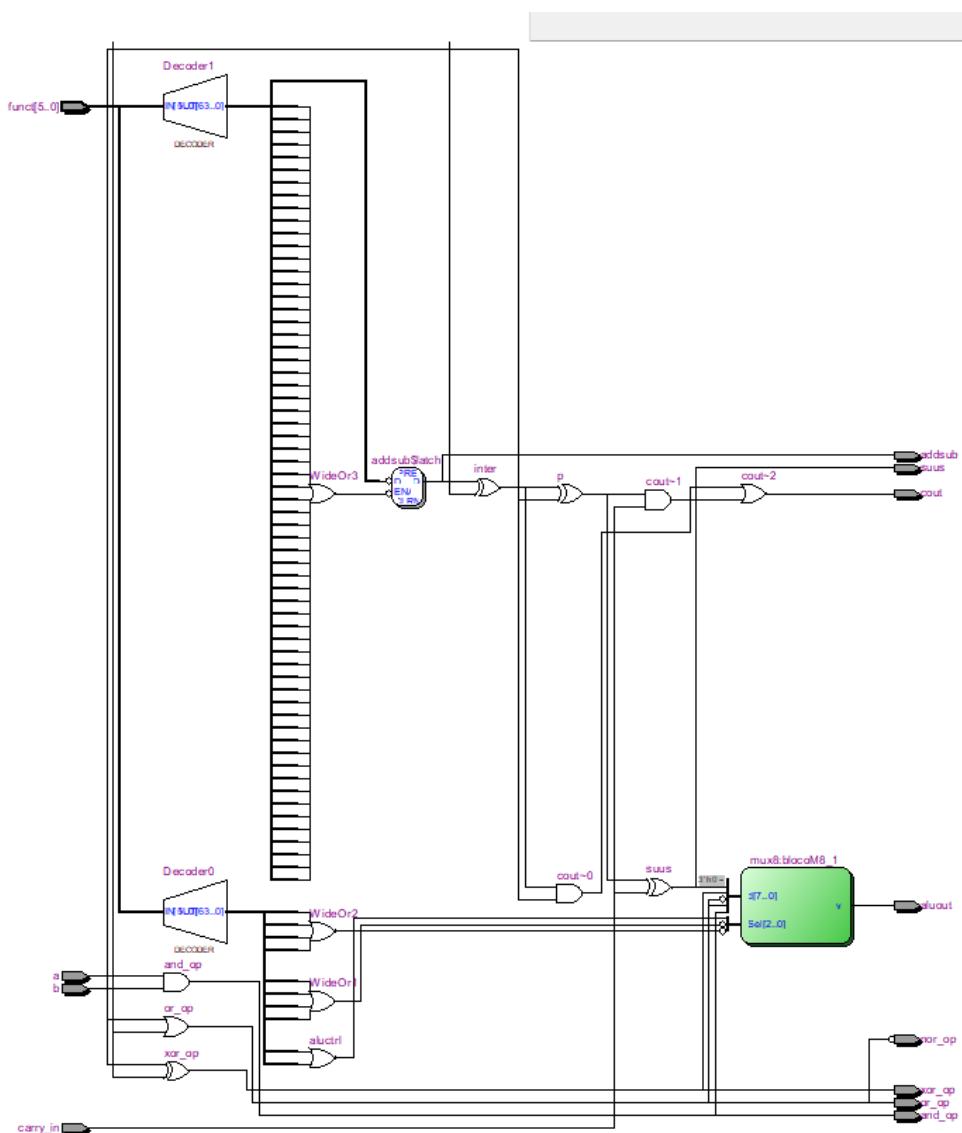
    assign s_B = B ^ add_sub;
    assign {cout,s_addsub} = A+s_B+carry_in;

    assign s_and = A&B;
    assign s_or = A|B;
    assign s_xor = A^B;
    assign s_nor = ~(A|B);

    always_comb begin
        case(op)
            3'b000:     Y = s_and;
            3'b001:     Y = s_or;
            3'b100:     Y = s_nor;
            3'b011:     Y = s_xor;
            3'b010:     Y = s_addsub;
            3'b110:     Y = s_addsub;
            3'b111:     Y = less;
            default:    Y = 1'b0;
        endcase
    end
endmodule
```

Por fim, uma vez já identificada a operação deve-se realizá-la:

Visão RTL



Golden Model

O golden model foi feito em C++ de modo similar ao modelo feito para testar a máquina de estados. Sua estrutura está separada em blocos, no quais implementam as funções aritméticas e lógicas da ULA separadamente. Na Figura abaixo é ilustrado o controle da ULA o qual é necessário para a função principal do programa.

```
#include <fstream>
#include <iostream>
#include <string>
#include <string.h>
using namespace std;

int controle(string funct, string aluctrl){
    if(!funct.compare("100000")){ //ADD
        aluctrl.assign("100");
        return 0;
    }
    else if(!funct.compare("100010")){ //SUB
        aluctrl.assign("100");
        return 1;
    }
    else if(!funct.compare("100100")){ //AND
        aluctrl.assign("000");
        return -1;
    }
    else if(!funct.compare("100101")){ //OR
        aluctrl.assign("001");
        return -1;
    }
    else if(!funct.compare("100111")){ //NOR
        aluctrl.assign("011");
        return -1;
    }
    else if(!funct.compare("100110")){ //XOR
        aluctrl.assign("010");
        return -1;
    }
}
```

Vetores de Teste

Os vetores de teste consistiram em 48 linhas, nas quais foram variadas o comando de função da ULA e os parâmetros de entrada ‘a’ e ‘b’.

funct_ addsub _ a_ b_ Ci_ and_or_xor_nor_S_Cout_Y
100000_0_0_0_0_0_0_0_1_0_0_0
100000_0_0_0_1_0_0_0_1_1_0_1
100000_0_0_1_0_0_1_1_0_1_0_1
100000_0_0_1_1_0_1_1_0_0_1_0
100000_0_1_0_0_0_1_1_0_1_0_1
100000_0_1_0_1_0_1_1_0_0_1_0
100000_0_1_1_0_1_1_0_0_0_1_0
100000_0_1_1_1_1_1_0_0_1_1_1
100010_1_0_0_0_0_0_0_1_1_1_1
100010_1_0_0_1_0_0_0_1_0_1_0
100010_1_0_1_0_0_1_1_0_0_0_0
100010_1_0_1_1_0_1_1_0_1_1_1
100010_1_1_0_0_0_1_1_0_0_0_0

100010_1_1_0_1_0_1_1_0_1_1_1
100010_1_1_1_0_1_1_0_0_1_0_1
100010_1_1_1_1_1_1_0_0_0_0_0
100100_1_0_0_0_0_0_0_1_1_1_0
100100_1_0_0_1_0_0_0_1_0_1_0
100100_1_0_1_0_0_1_1_0_0_0_0
100100_1_0_1_1_0_1_1_0_1_1_0
100100_1_1_0_0_0_1_1_0_0_0_0
100100_1_1_0_1_0_1_1_0_1_1_0
100100_1_1_1_0_1_1_0_0_1_0_1
100100_1_1_1_1_1_1_0_0_0_0_1
100101_1_0_0_0_0_0_0_1_1_1_0
100101_1_0_0_1_0_0_0_1_0_1_0
100101_1_0_1_0_0_1_1_0_0_0_1
100101_1_0_1_1_0_1_1_0_1_1_1
100101_1_1_0_0_0_1_1_0_0_0_1
100101_1_1_0_1_0_1_1_0_1_1_1
100101_1_1_1_0_1_1_1_1_0_0_0_1
100110_1_0_0_0_0_0_0_1_1_1_1
100110_1_0_0_1_0_0_0_1_0_1_1
100110_1_0_1_0_0_1_1_0_0_0_0
100110_1_1_0_0_0_1_1_0_0_0_0
100110_1_1_0_1_0_1_1_0_1_1_0
100110_1_1_1_0_1_1_1_1_0_0_0_0
100111_1_0_0_0_0_0_0_1_1_1_0
100111_1_0_0_1_0_0_0_1_0_1_0
100111_1_0_1_0_0_1_1_0_0_0_1
100111_1_0_1_1_0_1_1_0_1_1_1
100111_1_1_0_0_0_1_1_0_0_0_1
100111_1_1_0_1_0_1_1_0_1_1_1
100111_1_1_1_0_1_1_0_0_1_0_0
100111_1_1_1_1_1_1_0_0_0_0_0

TestBench

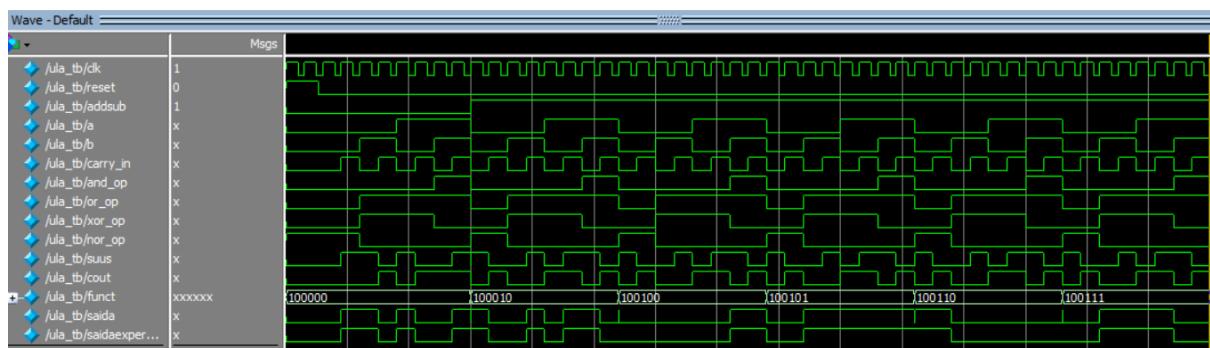
```
3  logic [16:0] testvectors[48];
4  logic clk, reset, addsub, a, b, carry_in, and_op, or_op, xor_op, nor_op, suus, cout;
5  logic addsub_esp, and_op_esp, or_op_esp, xor_op_esp, nor_op_esp, suus_esp, cout_esp;
6  logic [5:0] funct;
7  logic saida, saidaexperado;
8  logic [31:0] vectornum, errors;
9  ula DUV(funct, addsub, a, b, carry_in, and_op, or_op, xor_op, nor_op, suus, cout, saida);
10
11 initial begin
12   $readmemb("C:/Users/satcl/OneDrive/Documentos/Concepcao/MIPS-Componentes/ula/simulation/modelsim/ula.tv", testvectors);
13   vectornum = 0;
14   errors = 0;
15   reset = 1;
16   #27;
17   reset = 0;
18 end
19
20 always begin
21   clk = 1; #10; clk = 0; #5;
22 end
23 always @(posedge clk) begin
24   {funct, addsub_esp, a, b, carry_in, and_op_esp, or_op_esp, xor_op_esp, nor_op_esp, suus_esp, cout_esp, saidaexperado} = testvectors
25   end// check results on falling edge of clk-
26 always @(negedge clk)
27   if (~reset) begin // skip during reset
28     if (saida != saidaexperado) begin // check result
29       $display("| %b | Erro", funct, addsub, a, b, carry_in, and_op, or_op, xor_
30       $display(" output = %b (%b esperado)", saida, saidaexperado);
31       errors = errors + 1;
32     end else begin
33       $display("| %b | Ok", funct, addsub, a, b, carry_in, and_op, or_op, xor_
34     end
35   vectornum = vectornum + 1;
36   if (vectornum == $size(testvectors)) begin
37     $display("Testes Efetuados = %0d", vectornum);
38     $display("Erros Encontrados = %0d", errors);
39     #10;
40     $stop;
```

Simulação

O arquivo foi testado com os vetores de referência e apresentou 0 erros.

```
# ----- #
# | 100110 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Ok
# | 100110 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Ok
# | 100111 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Ok
# | 100111 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | Ok
# | 100111 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | Ok
# | 100111 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | Ok
# | 100111 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | Ok
# | 100111 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | Ok
# | 100111 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | Ok
# | 100111 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | Ok
# Testes Efetuados = 48
# Erros Encontrados = 0
```

Abaixo é apresentada a saída no formato de ondas da ULA.



ULA (32 bits)

Para a ULA de 32 bits foi realizada 32 instâncias da ULA de 1 bit, e foi adicionado o controle da ULA para o interior da ULA de 32.

Descrição em System Verilog

A descrição em systemverilog é apresentada a seguir, na qual há 32 instâncias da ULA de 1 bit. Nessa versão da ULA foi adicionada a saída do overflow que é calculado a partir de operações lógicas.

```
module ula_32(input logic [31:0] A, B,
               input logic [2:0] ulaControl,
               output logic [31:0] Y,
               output logic zero,
               output logic overflow);

    logic add_sub;
    logic [31:0] cout;
    logic [30:0] zerotest;
    logic less;

    assign add_sub = (ulaControl == 3'b111 | ulaControl == 3'b110) ? 1'b1 : 1'b0;

    ula int1(A[0], B[0], add_sub, less, add_sub, ulaControl, Y[0], cout[0]);
    or(zerotest[0], Y[0] , Y[1]);

    genvar i;

    generate
        for(i = 1; i < 31; i = i + 1) begin : generate_1bitalu_instances
            ula int2(A[i], B[i], cout[i - 1], 1'b0, add_sub, ulaControl, Y[i], cout[i]);
            or(zerotest[i], zerotest[i-1], Y[i+1]);
        end
    endgenerate

    not(zero, zerotest[30]);

    ula intlast(A[31], B[31], cout[30], 1'b0, add_sub, ulaControl, Y[31], cout[31], less);
    xor(overflow, cout[30], cout[31]);

endmodule
```


PC, IR e MDR (32 bits)

Todos esses três componentes: Program Counter (PC), Instruction Register (IR) e o Data Memory Register(MDR) são registrador de 32 bits. Tecnicamente a modelagem destes três componentes são iguais, por isto todos os códigos abaixo foram replicados para estes componentes.

Embora o comportamento seja igual, a função de cada registrador é diferente. O PC, por exemplo, indica o endereço da próxima instrução a ser executada, enquanto a sua saída, aponta para a instrução atual.

O comportamento do IR assemelha-se a um transmission gate o qual só permite a passagem da entrada quando está submetido às condições ideias de clock e enable. A sua saída será conectada, a posterior nas três entradas do banco de registradores.

Enquanto o MDR permite a transmissão dos dados à quarta entrada do banco de registradores que é o conteúdo do dado a ser escrito nos registradores.

O bloco lógico equivale ao seguinte ilustrado. Porém o IR e o MDR não possuem a entrada habilitadora. No entanto, elas foram setadas para um e foi utilizado o mesmo bloco.



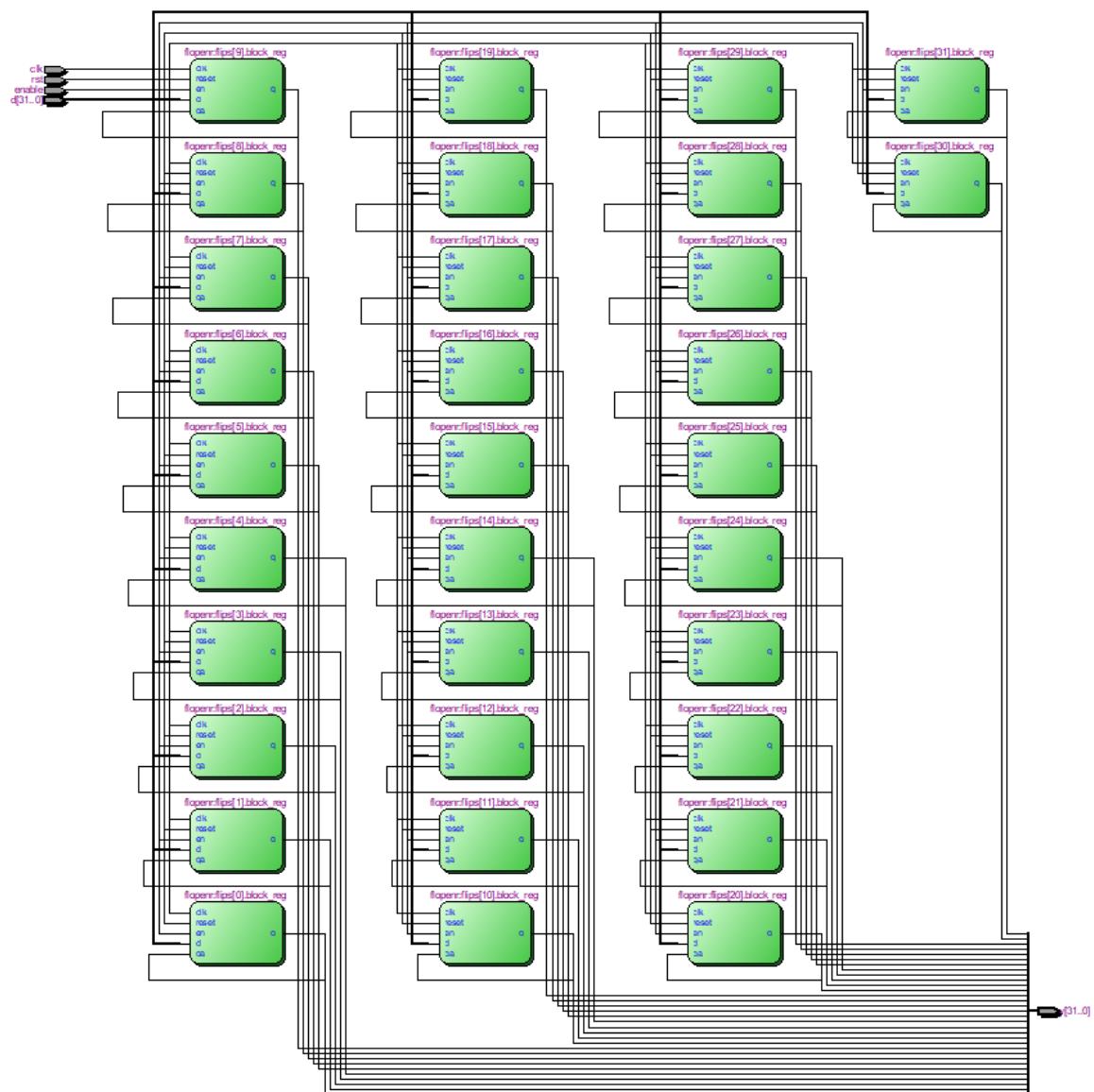
Descrição em System Verilog

Os módulos desses registradores resultam em uma saída com 32 bits, no entanto já foi desenvolvido anteriormente um registrador com enabled para 1 bit de dado. Desse modo, a descrição em verilog consiste em 32 instâncias desses blocos já desenvolvidos. Dessa forma utiliza-se o genvar para criar um for e atualizar o dado de cada posição do array e transmitir esses dados para a instância do registrador de 1 bit.

```
module pc(rst, clk, enable, d, y);  
    input logic [31:0] d;  
    input logic clk, rst, enable;  
    output logic [31:0] y;  
    logic [31:0]qa;  
  
    genvar i;  
    generate  
        for (i=0; i<32;i=i+1) begin: flips  
            flop#(1) block_reg(clk, rst, enable, d[i], y[i],qa[i]); //instancia do flip  
            assign qa[i] = y[i];  
        end  
    endgenerate  
  
endmodule
```

Visão RTL

O esquemático RTL ilustra a instância de 32 bloco de flip-flops (um para cada bit da entrada) cuja junção de todas as saídas resultam em um array 32 bits.



Golden Model

O modelo de ouro consiste em alterar os valores de entrada e para cada valor modificar o clock para analisar se a saída anterior seria mantida, caso o enable não estivesse ativo, independente da borda do clock.

```
int clk, enable, rst;
int d[5];
int d_32;
int y[32] = {0};
int a, b, c, f, e;
int qa[32] = {0};
int count = -2;

for (rst = 0; rst < 2; rst++)
    for (a = 0; a < 2; a++)
    { //d
        for (b = 0; b < 2; b++)
            for (c = 0; c < 2; c++)
                for (f = 0; f < 2; f++)
                    for (e = 0; e < 2; e++)
                        for (enable = 0; enable < 2; enable++)
                            for (clk = 0; clk < 2; clk++)
                            {
                                count += 1;

                                d[4] = a;
                                d[3] = b;
                                d[2] = c;
                                d[1] = f;
                                d[0] = e;

                                fprintf(arquivo, "%d_%d_%d_", rst, clk, enable);

                                d_32 = decoder(d[4], d[3], d[2], d[1], d[0]);

                                for (int i = 0; i < 32; i++)
                                {
                                    fprintf(arquivo, "%d", d_32[i]);
                                }
                                fprintf(arquivo, "_");
                                if (count < 2)
                                {
                                    for (int i = 0; i < 32; i++)
                                    {
```

Vetores de Ouro

Os vetores de ouro consistiram em 256 linhas, com alguns resultados como os ilustrados abaixo:

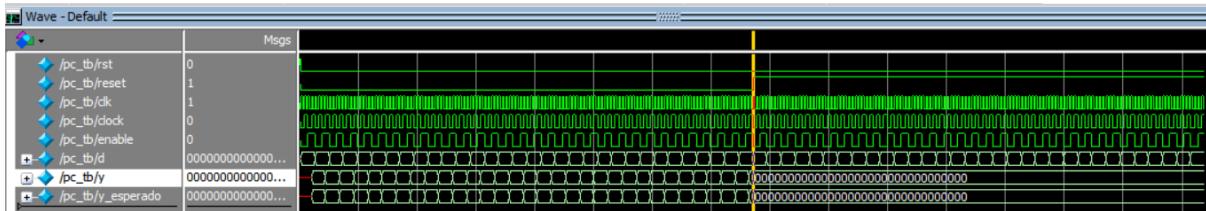
0_0_0_0000000000000000000000000000000010000_000000000000000000000000000000001000
0_1_0_0000000000000000000000000000000010000_000000000000000000000000000000001000
0_0_1_0000000000000000000000000000000010000_000000000000000000000000000000001000
0_1_1_0000000000000000000000000000000010000_0000000000000000000000000000000010000
0_0_0_00000000000000000000000000000000100000_0000000000000000000000000000000010000

Simulação

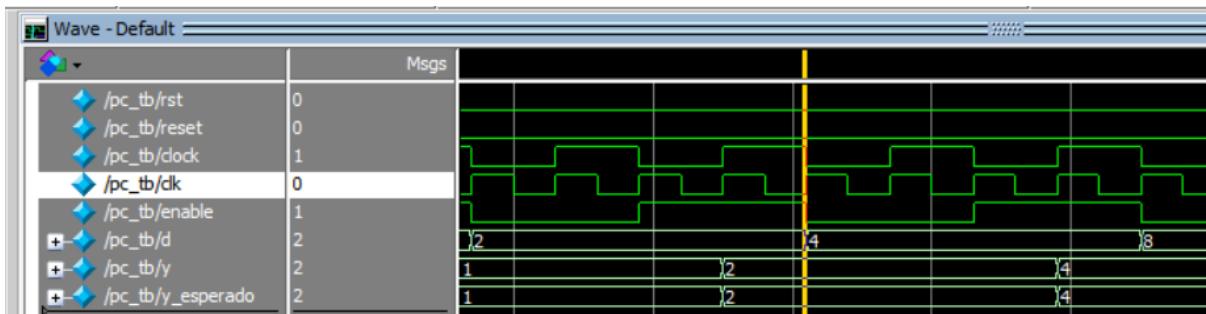
Os 256 casos foram verificados com o ModelSim e nenhum apresentou erro.

```
# | 1 | 1 | 1 | 0 | 01000000000000000000000000000000000000000000000000000000000 | 00000000000000000000000000000000000000000000000000000000000 | OK
# | 1 | 0 | 0 | 1 | 01000000000000000000000000000000000000000000000000000000000 | 00000000000000000000000000000000000000000000000000000000000 | OK
# | 1 | 1 | 1 | 1 | 01000000000000000000000000000000000000000000000000000000000 | 00000000000000000000000000000000000000000000000000000000000 | OK
# | 1 | 0 | 0 | 0 | 10000000000000000000000000000000000000000000000000000000000 | 00000000000000000000000000000000000000000000000000000000000 | OK
# | 1 | 1 | 0 | 0 | 10000000000000000000000000000000000000000000000000000000000 | 00000000000000000000000000000000000000000000000000000000000 | OK
# | 1 | 1 | 1 | 0 | 10000000000000000000000000000000000000000000000000000000000 | 00000000000000000000000000000000000000000000000000000000000 | OK
# | 1 | 0 | 1 | 1 | 10000000000000000000000000000000000000000000000000000000000 | 00000000000000000000000000000000000000000000000000000000000 | OK
# | 1 | 1 | 1 | 1 | 10000000000000000000000000000000000000000000000000000000000 | 00000000000000000000000000000000000000000000000000000000000 | OK
# Testes Efetuados = 256
# Erros Encontrados = 0
```

O marcador separa o reset durante a sua transição do nível lógico alto para o baixo.



A partir do marcador podemos ver quando a entrada é modificada para o valor 4 em radix decimal, no entanto os valores da saída só são alterados quando o enable é ativo e o clock está na borda de subida.





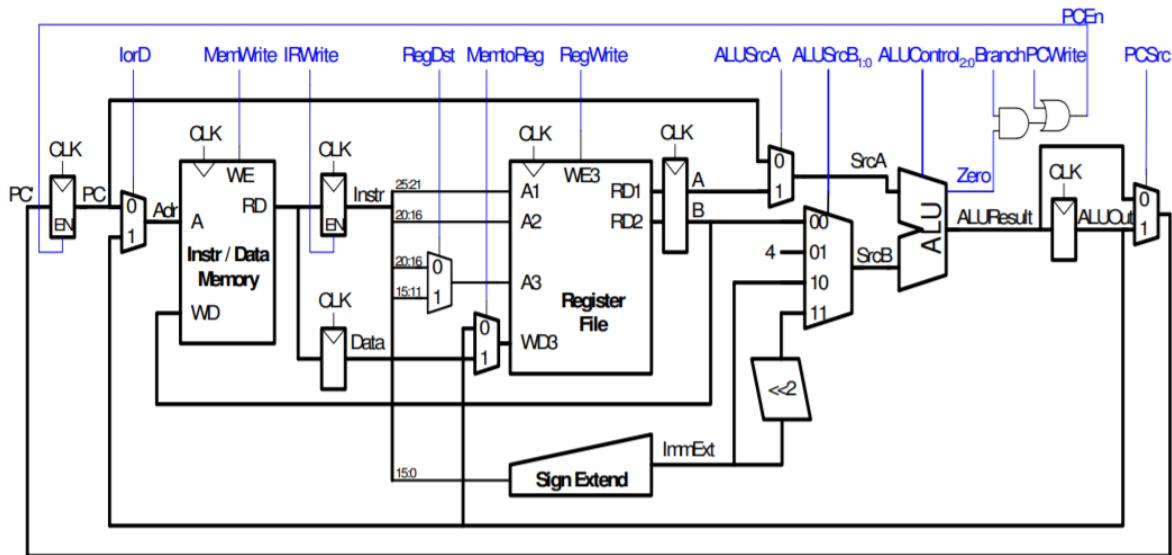
**Universidade Federal da Paraíba
Centro de Informática**

Anexo II: Datapath

Neste anexo é apresentada a lógica de funcionamento do datapath bem como o fluxo de dados mediante a execução de algumas instruções e a importância da interligação dos seus respectivos componentes.

Funcionamento do DataPath

A estrutura do datapath é composta pelos blocos anteriormente desenvolvidos. A Figura abaixo apresenta em preto a conexão entre todos estes componentes, em azul temos os sinais de entrada do datapath que são oriundos da unidade de controle.

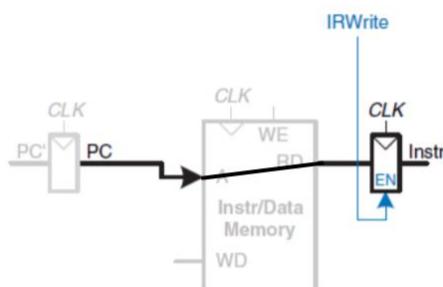


Coneção dos blocos

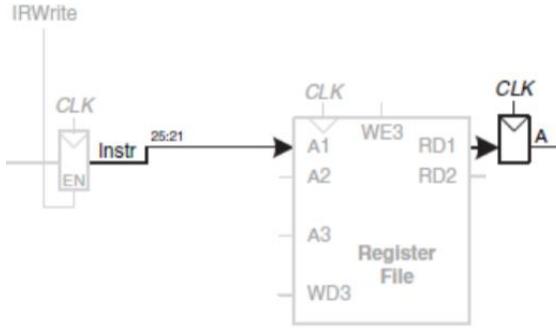
Para iniciarmos a montagem do datapath, o primeiro passo a ser executado é instanciar o bloco do PC. Este registrador contém o endereço da instrução a executar. Na implementação do MIPS desenvolvido neste relatório não será incluído o Data Memory no datapath, pois ele será um arquivo a parte com os dados de entrada.

Desse modo, a saída do PC é conectada diretamente nos registradores IR e MDR e seu valor é gravado nesses registradores. O IR recebe um sinal de habilitação, chamado IRWrite, que é ativado quando deve ser atualizado com uma nova instrução.

Para o MIPS desenvolvido neste relatório a conexão entre o PC e o IR fica da seguinte forma:

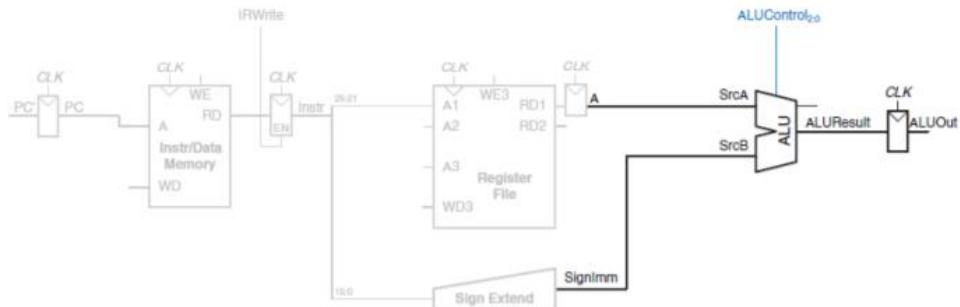


O próximo deve ser identificar dentre desse array de 32 bits a parte que contém o endereço de base e passá-la para a entrada do banco de registradores. Desse modo, são passados 5 bits para a entrada A1 do banco de registradores. Como ilustra a Figura a abaixo.

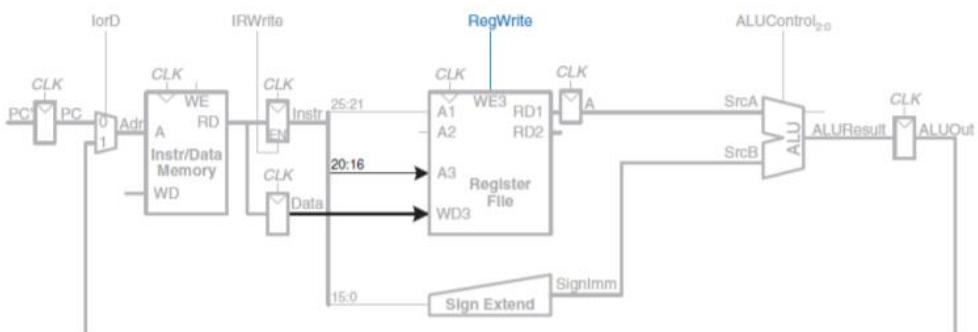


Desse modo, o banco lê o registro para RD1 e o armazena em um registrador. Para A instrução lw também requer um deslocamento. Outra parte dos dados que saem do IR são estendidas para 32 bits com o intuito de adicionar o offset ao endereço da base quando a entrada A da Figura anterior for a primeira entrada da ULA.

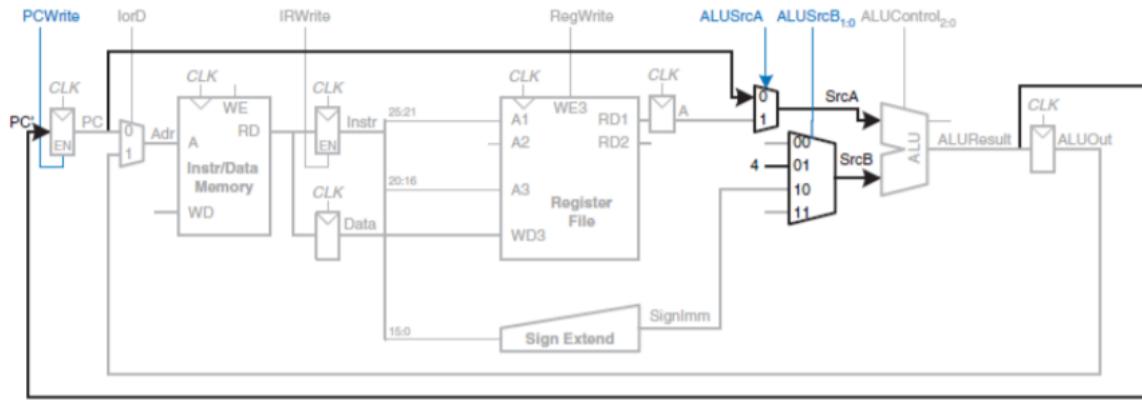
A ULA vai executar a soma dessas duas entradas e sua saída será armazenada em um registrador sem enable. A Figura abaixo ilustra essa conexão.



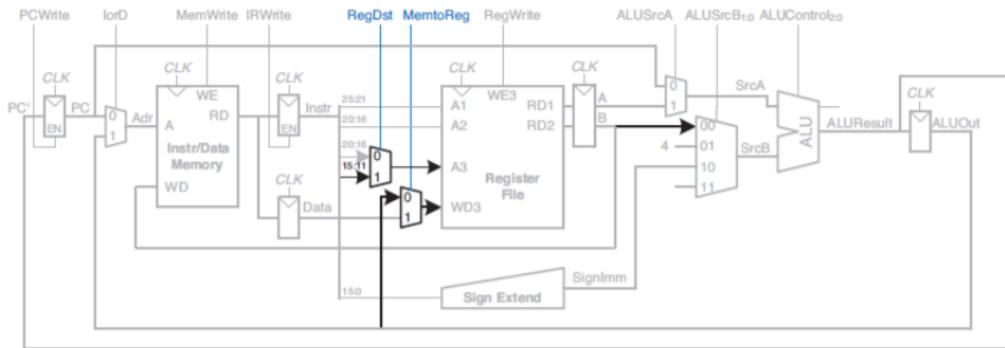
O Carregamento desses dados na memória se dá pela conexão do Aluout à entrada de um multiplexador que é colocado na saída do PC. Assim, o endereço de memória, Adr, é escolhido ou a partir do PC ou do ALUOut, a que vai definir isso será o enable desse mux que é o IorD. Desse modo, é inserido um registrador para carregar os dados do A2 no banco de registradores.



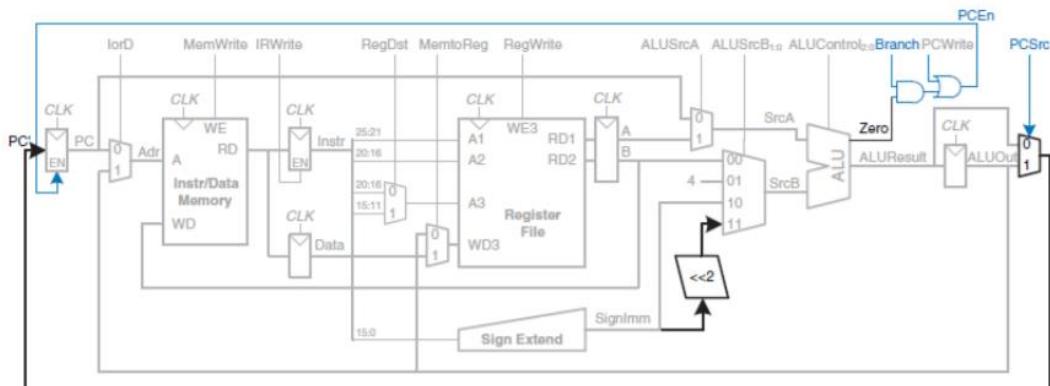
Para atualizar o valor da entrada do PC deve-se inserir multiplexadores de origem para escolher o PC e a constante 4 como entradas da ULA. E um multiplexador de duas entradas controladas para escolher entre o PC ou o A para ser o SrcA. Além disso, um multiplexador de quatro entradas controladas por ALUSrcB escolhe 4 ou Sign-Imm como SrcB. A Figura fica assim:



Para obter instruções do tipo-R, a instrução é novamente obtida, e os dois registros de origem são lidos a partir do banco de registros. A ALUSrcB1:0, a entrada de controle do multiplexador SrcB, é utilizado para escolher o registro B como o segundo registro de origem para a ALU.



Para as instruções beq, a instrução é outra vez obtida, e os dois registros de fonte são lidos do banco de registros. Para determinar se os registros são iguais, a ALU subtrai os registros e, quando o resultado é zero, a flag Zero é ativada. Assim, a ALU determina PC + 4 e coloca o resultado no program counter, como é realizado para as outras instruções. Noutro passo, a ALU utiliza este valor atualizado do PC para determinar PC mais o sinal estendido que é deslocado 2 vezes para a direita (multiplicar por 4). Assim, a Figura abaixo ilustra o caminho melhorado para o beq.

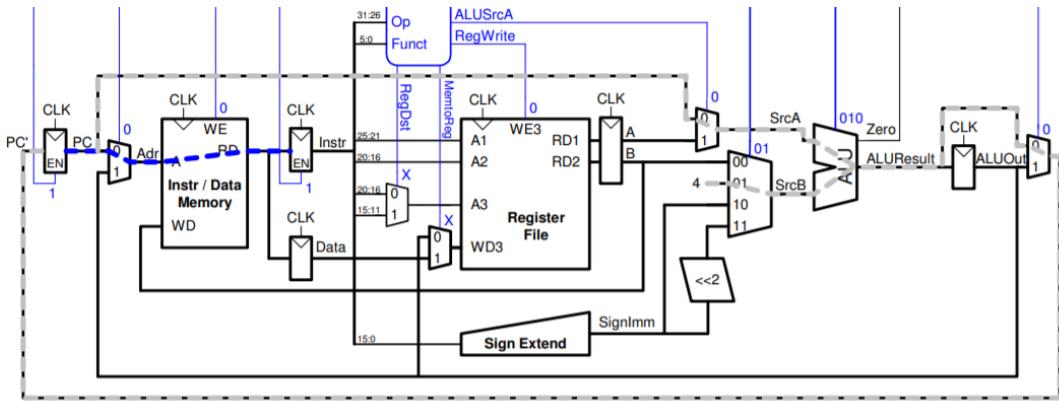


Dessa forma, todos os blocos que compõe o datapath foram conectados.

Fluxo de dados no Fetch

Para melhor explicar o fluxo dos dados através do datapath, irei explicar como se dá para a instrução do Fetch.

O fluxo de dados para atualizar o PC é ilustrado na Figura abaixo. Os caminhos em cinza indicam a transmissão do valor salvo em PC. Inicialmente o PC é inicializado com o valor 0, desse modo, esse valor é transmitido para o multiplexador que escolhe a entrada SrcA da ULA. Em estado de fetch, o ALUSrcA é 0 e o ALUSrcB é 01, dessa forma a ULA é carregada com 0 e 4. A sua saída resulta em o valor de PC+4, que por sua vez é encaminhada para a entrada de PC. Assim o valor de PC aponta para a próxima instrução.



Descrição em System Verilog

Como já descrito anteriormente, o datapath consiste na conexão de seus componentes. Desta forma o módulo em systemverilog se constitui das instâncias dos módulos previamente definidos. O código é apresentado abaixo, dentro dos parênteses são passados os parâmetros das instâncias, ou seja, as suas entradas e saídas.

```
assign inputPC = reset? 32'd0 : outputMUXtoPC;
logic [31:0]qa;

mux2 mux_BNE(Zero, !Zero, BranchNE, Zero_out);
assign PCEn = (PCWrite | (Branch&Zero_out));

pc PCreg(reset, clk, PCEn, inputPC, outputPC);
mux2_32 muxIORD (outputPC,ALUoutput,IorD,ADR); //primeiro uso do PC out
regbank_32 reg_bank(clk,RegWrite,reset,outputData, outputIR[25:21],outputIR[20:16],outputMux5Bit,RD1,RD2);
reg32 RD1reg (reset, clk, 1'b1, RD1, A);
reg32 RD2reg (reset, clk, 1'b1, RD2, B);

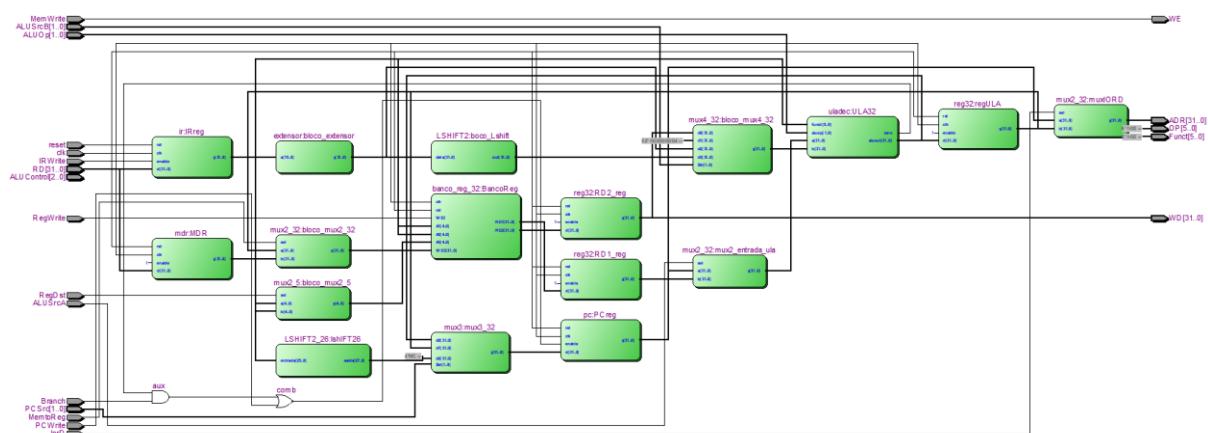
assign WD = B;

extensor Singmn (outputIR[15:0],outputExtensor);
LShiftT2 bacoLshift (outputExtensor,outputLShift);
mux2_32 mux2entradaaula (outputPC, A, ALUSrcA, SrcA); //srcA = entrada da ULA
mux4_32 bocomux432 (B, 32'00000000000000000000000000000000, outputExtensor,outputLShift, ALUSrcB, SrcB); //srcB = entrada da ULA

ula_32 ALU(SrcA,SrcB, ALUControl, ALUresult, Zero, overflow);
//uladec ULA32(outputIR[5:0],ALUOp,SrcA,SrcB,ALUresult,Zero);
reg32 regULA(reset, clk, 1'b1, ALUresult, ALUoutput);
LShiftT2_26 lshIFT26(outputIR[25:0], outputExtensorMuxPc);
mux3 mux332 ( ALUresult,ALUoutput,{outputPC[31:28], outputExtensorMuxPc[27:0]},PCSrc, outputMUXtoPC);
```

Visão RTL

O esquemático RTL ilustra todas as instâncias utilizadas no módulo do datapath.



Golden Model

Dada a complexidade de elaborar este golden model linguagem C, foi optado utilizar o python. A Lógica do programa baseia-se na análise do opcode e concatenação dos bits.

```
file.write(saidaBin, "00000100000", mnemonic, rs, rt, rd)
break
elif opcode == 2:
    file.write(saidaBin, "00000100101", mnemonic, rs, rt, rd)
break
elif opcode == 3:
    file.write(saidaBin, "00000100100", mnemonic, rs, rt, rd)
break
elif opcode == 5:
    file.write(saidaBin, "00000101010", mnemonic, rs, rt, rd)
break
elif opcode == 6:
    file.write(saidaBin, "00000100010", mnemonic, rs, rt, rd)
break
elif opcode == 9:
    file.write(saidaBin, mnemonic, tobinstr(imm, 26))
break

inst = ["addi", "add", "or", "and", "beq", "slt", "sub", "sw", "lw", "jmp", "end"]
regs = ['$0', '$at', '$v0', '$v1', '$a0', '$a1', '$a2', '$a3', '$t0', '$t1',
        '$t2', '$t3', '$t4', '$t5', '$t6', '$t7', '$s0', '$s1', '$s2', '$s3',
        '$s4', '$s5', '$s6', '$s7', '$t8', '$t9', '$k0', '$k1', '$gp', '$sp',
        '$fp', '$ra']

LEN_MEMORY = 1024
TAM_INSTR = 200
QTD_REGISTER = 32
QTD_INSTSET = 11

file = open('datapath.tv', 'w')

decode(instruction, mnemonic, rd, rs, rt, imm, opcode, PC)
mountBin(mnemonic, rs, rt, rd, imm, opcode)

file.close()
```

Vetores de Testes

Os vetores de testes, incialmente foram obtidos a partir de uma planilha completada manualmente, pois foi optado por realizá-la desta forma, uma vez que o Golden model exige um nível alto de complexidade para implementar. No entanto, as saídas estavam incorretas, dessa forma elaborou-se o Golden Model e o resultado foram 34 linhas de vetores de testes, como ilustra a Figura a seguir



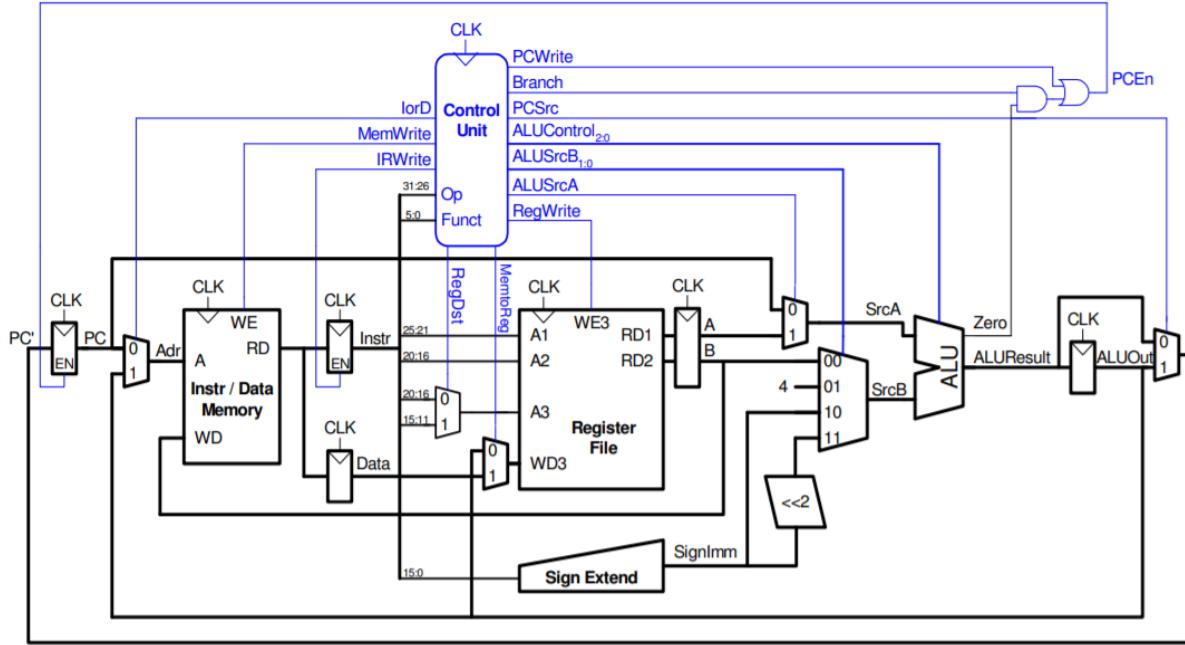
Universidade Federal da Paraíba
Centro de Informática

Anexo III: Unidade de Controle

Neste anexo é explicada a lógica de funcionamento da unidade de controle, bem como a sua implementação em system-verilog, Golden model e simulação.

Máquina de Estados Finita - FSM

A unidade de controle do MIPS tem por objetivo coordenar todas as operações e encaminhamento de dados habilitadores para o datapath. A Imagem abaixo ilustra a etapa final da implementação do mips. No entanto, nesse momento é importante apenas o entendimento do funcionamento da unidade de controle.

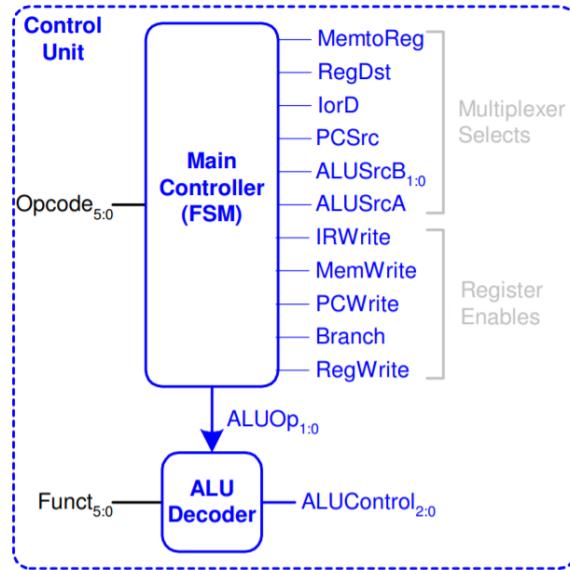


Em destaque em azul, temos a unidade de controle, que é modelada através de uma máquina de estados. Como é perceptível na Figura, todas as saídas da unidade de controle são variáveis habilitadoras ou chaves de seleção dos blocos que estão em preto.

Todos os componentes e barramentos em preto compõe o datapath do sistema. Ou seja, o mips é a conexão entre esse datapath e a unidade de controle.

Metodologia de Síntese

A implementação da unidade de controle é indicada na Figura a seguir. O módulo da FSM irá receber duas variáveis, o opcode e o funct. E através deles será determinado o estado da máquina. O controlador principal é uma FSM que aplica os sinais de controle de acordo com os estados do ciclo. A sequência desses sinais irá depender da instrução a ser executada. Neste relatório o controle da ULA foi implementado como algumas instruções interiores ao módulo.

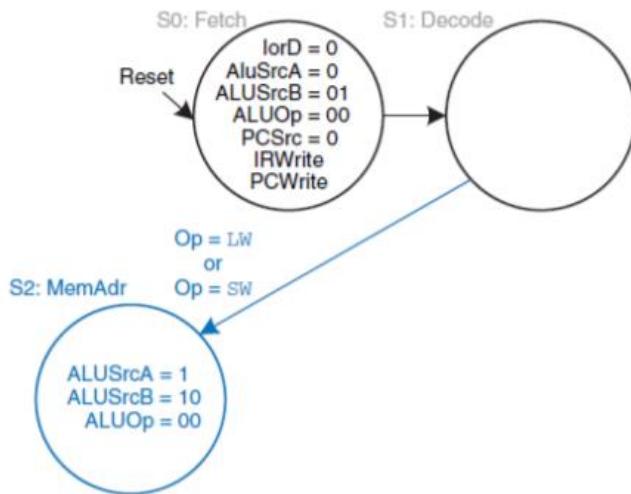


Os estados podem ser os seguintes:

Estado	Função
S0	Fetch
S1	Decodificação de Instrução (Decode)
S2	Cálculo do endereço efetivo (MemAdr)
S3	Leitura de Memória (Mem Read)
S4	Escrita de RF (Mem Writeback)
S5	Memwrite
S6	Executa
S7	ALU Writeback
S8	Branch
S9	ADDI executa
S10	ADDI writeback

Se a instrução é um load Word ou Store Word, o processador multi-ciclo determina o endereço adicionando o endereço base ao imediato com sinal estendido. Isso requer que **ALUSrcA** = 1 para seleccionar o registro A e que **ALUSrcB** = 10 para seleccionar **SignImm**. **ALUOp** = 00,

assim a ALU adiciona. O endereço efetivo é armazenado no registro ALUOut para uso no passo seguinte [Harris], isso pode ser ilustrado na Figura a seguir.



Descrição dos modelos em System Verilog

A máquina de estados finitos (FSM) consiste inicialmente em um always que verifica se a máquina foi resetada ou se é necessário avançar para o próximo estado.

```
always_ff @ (posedge clk, posedge reset)
  if (reset) state <= S0;
  else if (clk) state <= nextstate;
// next state logic
```

Em seguida inicia-se o bloco always que vai modificar os valores das variáveis sempre que o estado for alterado. E é definido todos os estados da máquina.

```

case (state)
S0: begin
    nextstate   <= S1;
    IorD      = 1'b0;
    AluSrcA   = 1'b0;
    AluSrcB   = 2'd1;
    ALUOp     = 2'd0;
    PCSrc    = 2'd0;
    IRWrite   = 1'b1;
    PCWrite   = 1'b1;
    AluSrcA = AluSrcA;
    RegWrite  = 1'b0;
    MemWrite  = 1'b0;
    Branch   = 1'b0;
    PCWrite  = 1'b0;
    MemtoReg = 1'b0;
    RegDst   = 1'b0;
end

S1: begin
    AluSrcA   = 1'b0;
    AluSrcB   = 2'd3;
    ALUOp     = 2'd0;
    IRWrite   = 1'b0;
    PCWrite   = 1'b0;
    case(opcode)
        6'b000000: nextstate <= S6;
        6'b100011: nextstate <= S2;
        6'b101011: nextstate <= S2;
        6'b000100: nextstate <= S8;
        6'b000101: nextstate <= S8;
        6'b001000: nextstate <= S9;
        6'b000010: nextstate <= S11;
    endcase
end

S2: begin
    AluSrcA   = 1'b1;
    AluSrcB   = 2'd2;
    ALUOp     = 2'd0;
    case(opcode)
        6'b100011: nextstate <= S3;
        6'b101011: nextstate <= S5;
        default: nextstate <= S0;
    endcase
end

S3: begin
    IorD      = 1'b1;
    if(reset) nextstate <= S0;
    else nextstate <= S4;
end

S4: begin
    nextstate   <= S0;
    RegDst    = 1'b0;
    MemtoReg  = 1'b1;
    RegWrite   = 1'b1;
end

S5: begin
    nextstate   <= S0;
    IorD      = 1'b1;
    MemWrite  = 1'b1;
end

```

Ao chegar no estado 6, temos as instruções são do tipo R. Desta forma, como elas indicam as operações lógicas e aritméticas temos que analisar o funct. Por isso, foi implementado um case desta variável no estado 6. Os estados 61,62,63... indicam o estado dessas operações.

```

S6: begin
    AluSrcA = 1'b1;
    AluSrcB = 2'd0;
    ALUOp   = 2'd2;
    case(func)
        6'b100000: nextstate <= S61;
        6'b100010: nextstate <= S62;
        6'b100100: nextstate <= S63;
        6'b100101: nextstate <= S64;
        6'b100111: nextstate <= S65;
        6'b100110: nextstate <= S66;
        default: nextstate <= S0;
    endcase
end

S61: if(reset) nextstate <= S0;
     else nextstate <= S7;
S62: if(reset) nextstate <= S0;
     else nextstate <= S7;
S63: if(reset) nextstate <= S0;
     else nextstate <= S7;
S64: if(reset) nextstate <= S0;
     else nextstate <= S7;
S65: if(reset) nextstate <= S0;
     else nextstate <= S7;
S66: if(reset) nextstate <= S0;
     else nextstate <= S7;

S7: begin
    nextstate <= S0;
    RegDst   = 1'b1;
    MemtoReg = 1'b0;
    RegWrite = 1'b1;
end

S8: begin
    nextstate <= S0;
    AluSrcA = 1'b1;
    AluSrcB = 2'd0;
    ALUOp   = 2'd1;
    PCSrc   = 2'd1;
    Branch   = 1'b1;
end

S9: begin
    if(reset) nextstate <= S0;
    else nextstate <= S10;
    AluSrcA = 1'b1;
    AluSrcB = 2'd2;
    ALUOp   = 2'd0;
end

S10: begin
    nextstate <= S0;
    RegDst   = 1'b0;
    MemtoReg = 1'b0;
    RegWrite = 1'b1;
end

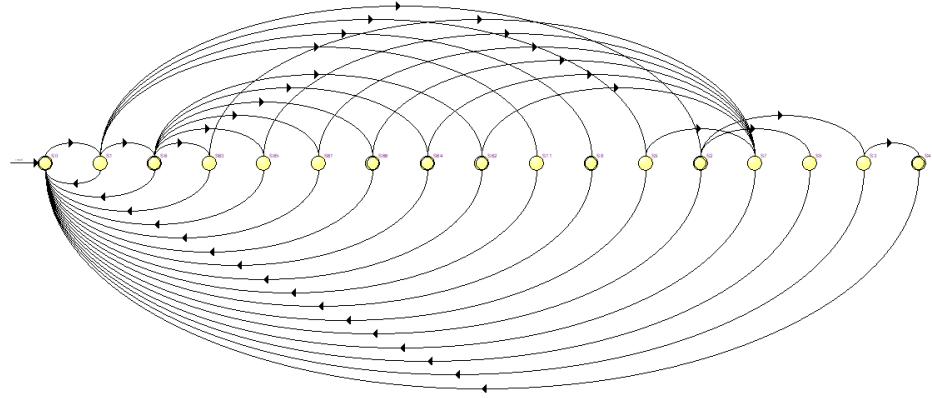
S11: begin
    nextstate <= S0;
    PCSrc   = 2'd2;
    PCWrite = 1'b1;
end

default: nextstate <= S0;

```

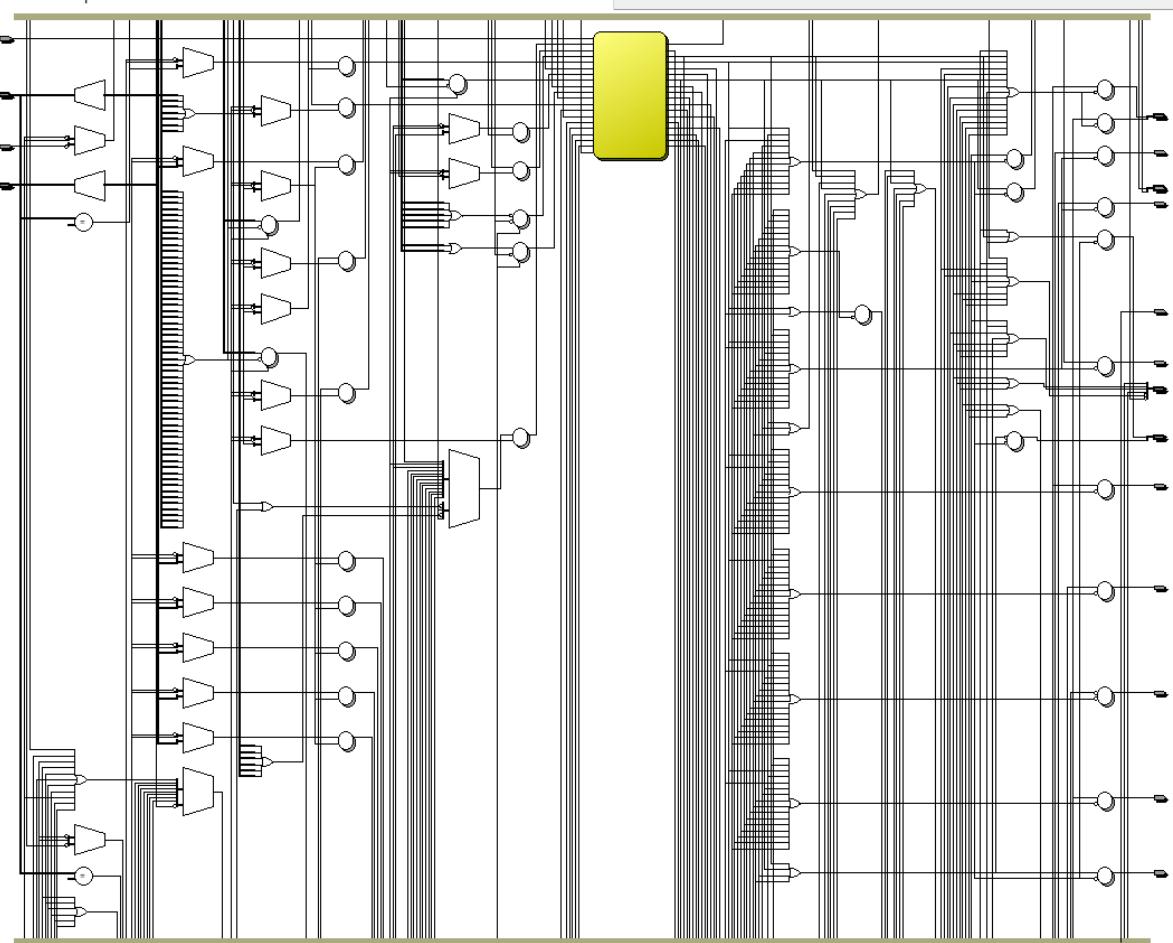
Visão Máquina de Estados

Num diagrama de transição de estado, os círculos representam os estados e os arcos representam as transições entre estados. As transições ocorrem no flanco ascendente do relógio; que não é representado no diagrama, porque ele está sempre presente num circuito sequencial síncrono. Além disso, o relógio controla quando devem ocorrer as transições, enquanto que o gráfico indica que transições ocorrem.



Visão RTL

O Arranjo dos blocos foi configurado conforme a visão RTL do circuito, é possível analisar que a inserção das variáveis de controle acrescenta maior complexidade e componentes que interligam os blocos.



Golden Model

A lógica de programação utilizada para verificar as instruções baseou-se nos seguintes fatos:

- Foi definido um padrão para os dados de entrada do arquivo, como sendo o primeiro bit, o sinal de reset (1 bit), seguido do opcode e da função: formato: *Reset_Opcode_Funct*;
- Lê-se então este arquivo;
- É realizada a detecção da transição/atual borda do clock;
- Detecta o estado atual da máquina de estados;
- Analisa-se o tipo de instrução e o seu próximo estado.

O código do programa escrito em c++ pode ser visto nas Figuras abaixo. A lógica de implementação utilizada baseia-se na verificação do código do arquivo de entrada. Na qual através de Switch cases e instruções condicionais como If e Else, o dado lido vai sendo definido para a sua função.

```
while (1)
{
    if (linha[0] == '1')
    {
        cout << "reset = 1, estado = 0";
        estado = 0;
        IorD = 0;
        AluSrcA = 0;
        AluSrcB[0] = 0;
        AluSrcB[1] = 1;
        ALUOp[0] = 0;
        ALUOp[1] = 0;
        PCSrc[0] = 0;
        PCSrc[1] = 0;
        IRWrite = 1;
        PCWrite = 1;
        out << "1." << clock << "" << opcode << "" << funct << "_" << IorD << "" << AluSrcA << "" <<
        | << "_" << estado << endl;
        cout << endl;
        IRWrite = 0;
        PCWrite = 0;
        terclock = antclock;
        antclock = clock;
        clock = terclock;
        break;
    }
    if (clock)
    {
        cout << "subida de clock, estado anterior " << estado << ", ";
        switch (estado)
        {
            case 0:
                IorD = 0;
                AluSrcA = 0;
                AluSrcB[0] = 0;
                AluSrcB[1] = 1;
                ALUOp[0] = 0;
                ALUOp[1] = 0;
                PCSrc[0] = 0;
                PCSrc[1] = 0;
                IRWrite = 1;
                PCWrite = 1;
                cout << "estado novo: " << estado;
                break;
            case 1:
                IorD = 1;
                AluSrcA = 1;
                AluSrcB[0] = 1;
                AluSrcB[1] = 0;
                ALUOp[0] = 1;
                ALUOp[1] = 0;
                PCSrc[0] = 1;
                PCSrc[1] = 0;
                IRWrite = 0;
                PCWrite = 0;
                cout << "estado novo: " << estado;
                break;
            case 2:
                IorD = 0;
                AluSrcA = 1;
                AluSrcB[0] = 0;
                AluSrcB[1] = 1;
                ALUOp[0] = 0;
                ALUOp[1] = 1;
                PCSrc[0] = 0;
                PCSrc[1] = 1;
                IRWrite = 1;
                PCWrite = 1;
                cout << "estado novo: " << estado;
                break;
            case 3:
                IorD = 1;
                AluSrcA = 0;
                AluSrcB[0] = 1;
                AluSrcB[1] = 0;
                ALUOp[0] = 1;
                ALUOp[1] = 0;
                PCSrc[0] = 1;
                PCSrc[1] = 0;
                IRWrite = 0;
                PCWrite = 0;
                cout << "estado novo: " << estado;
                break;
        }
    }
}
```

```

case 0:
    IorD = 0;
    AluSrcA = 0;
    AluSrcB[0] = 0;
    AluSrcB[1] = 1;
    ALUOp[0] = 0;
    ALUOp[1] = 0;
    PCSrc[0] = 0;
    PCSrc[1] = 0;
    IRWrite = 1;
    PCWrite = 1;
    cout << "estado novo: " << +estado;
    break;
case 1:
    AluSrcA = 0;
    AluSrcB[0] = 1;
    AluSrcB[1] = 1;
    ALUOp[0] = 0;
    ALUOp[1] = 0;
    if (!opcode.compare("100011") || !opcode.compare("101011"))
    {
        cout << "load/store detectado, indo para estado 2";
        estado = 2;
    }
    else if (!opcode.compare("000000"))
    {
        cout << "instrucao tipo R detectado";
        estado = 6;
    }
    else if (!opcode.compare("000100") || !opcode.compare("000101"))
    {
        cout << "instrucao tipo branch detectado";
        estado = 8;
    }
    else if (!opcode.compare("001000"))
    {
        cout << "instrucao ADDI detectada";
        estado = 9;
    }
    else if (!opcode.compare("000010"))
    {
        cout << "instrucao jump detectada";
        estado = 11;
    }
    break;
case 2:
    AluSrcA = 1;
    AluSrcB[0] = 1;
    AluSrcB[1] = 0;
    ALUOp[0] = 0;
    ALUOp[1] = 0;
    if (!opcode.compare("100011"))
    {
        cout << ", lw detectado, indo para estado 3";
        estado = 3;
        //    out << "0_" << clock << "" << opcode << "" << funct << "_3" << endl;
    }
    else if (!opcode.compare("101011"))
    {
        cout << ", sw detectado, indo para estado 5";
        estado = 5;
        //    out << "0_" << clock << "" << opcode << "" << funct << "_5" << endl;
    }
    break;
case 3:
    IorD = 1;
    cout << "estado novo: " << +estado;
    break;
case 4:

```

Vetores de Teste

O resultado da execução do golden model foi a seguinte tabela:

1_0_100011_000000_0000000
0_1_100011_000000_0000001
0_0_100011_000000_0000001
0_1_100011_000000_0000010
0_0_100011_000000_0000010

0_1_100011_000000_0000011
0_0_100011_000000_0000011
0_1_100011_000000_0000100
0_0_100011_000000_0000100
0_1_100011_000000_0000000
0_0_101011_000000_0000000
0_1_101011_000000_0000001
0_0_101011_000000_0000001
0_1_100011_000000_0000010
0_0_100011_000000_0000010
0_1_101011_000000_0000101
0_0_101011_000000_0000101
0_1_101011_000000_0000000
0_0_000000_100000_0000000
0_1_000000_100000_0000001
0_0_000000_100000_0000001
0_1_000000_100000_0000110
0_0_000000_100000_0000110
0_1_000000_100000_0111101
0_0_000000_100000_0111101
0_1_000000_100000_0000111
0_0_000000_100000_0000111
0_1_000000_100000_0000000
0_0_000000_100010_0000000
0_1_000000_100010_0000001
0_0_000000_100010_0000001
0_1_000000_100010_0000110
0_0_000000_100010_0000110
0_1_000000_100010_0111110
0_0_000000_100010_0111110
0_1_000000_100010_0000111
0_0_000000_100010_0000111
0_1_000000_100010_0000000
0_0_000000_100100_0000000
0_1_000000_100100_0000001
0_0_000000_100100_0000001
0_1_000000_100100_0000110
0_0_000000_100100_0000110
0_1_000000_100100_0111111
0_0_000000_100100_0111111
0_1_000000_100100_0000111
0_0_000000_100100_0000111
0_1_000000_100100_0000000
0_0_000000_100101_0000000
0_1_000000_100101_0000001
0_0_000000_100101_0000001
0_1_000000_100101_0000110
0_0_000000_100101_0000110
0_1_000000_100101_1000000
0_0_000000_100101_1000000
0_1_000000_100101_0000111

0_0_000000_100101_0000111
0_1_000000_100101_0000000
0_0_000000_100111_0000000
0_1_000000_100111_0000001
0_0_000000_100111_0000001
0_1_000000_100111_0000110
0_0_000000_100111_0000110
0_1_000000_100111_1000001
0_0_000000_100111_1000001
0_1_000000_100111_0000111
0_0_000000_100111_0000111
0_1_000000_100111_0000000
0_0_000000_100110_0000000
0_1_000000_100110_0000001
0_0_000000_100110_0000001
0_1_000000_100110_0000110
0_0_000000_100110_0000110
0_1_000000_100110_1000010
0_0_000000_100110_1000010
0_1_000000_100110_0000111
0_0_000000_100110_0000111
0_1_000000_100110_0000000
0_0_000100_000000_0000000
0_1_000100_000000_0000001
0_0_000100_000000_0000001
0_1_000100_000000_0001000
0_0_000100_000000_0001000
0_1_000100_000000_0000000
0_0_000101_000000_0000000
0_1_000101_000000_0000001
0_0_000101_000000_0000001
0_1_000101_000000_0001000
0_0_000101_000000_0001000
0_1_000101_000000_0000000
0_0_001000_000000_0000000
0_1_001000_000000_0000001
0_0_001000_000000_0000001
0_1_001000_000000_0001001
0_0_001000_000000_0001001
0_1_001000_000000_0000000
0_0_001000_000000_0000000
0_1_000010_000000_0000001
0_0_000010_000000_0000001
0_1_000010_000000_0001011
0_0_000010_000000_0001011
0_1_000010_000000_0000000
1_0_101011_000000_0000000

TestBench

O testbench checa os resultados durante a borda de descida e caso o reset não seja ativado. O arquivo com código é ilustrado abaixo.

```
module mips_tb();
    logic [15:0] testvectors[105];
    logic clk, reset, resete;
    logic [5:0] opcode, funct;
    logic saída, saídaesperado;
    logic [31:0] vectornum, errors;
    mips DUV(clk, resete, opcode, funct, saída);
    initial begin
        $readmem("C:/Users/satc1/Downloads/MIPS/ModelSim/Simulation/saida.tv", testvectors);
        vectornum = 0;
        errors = 0;
        reset = 1;
        #27;
        reset = 0;
    end
    always begin
        clk = 1; #10; clk = 0; #5;
    end
    always @(posedge clk) begin
        #1
        {clk, resete, opcode, funct, saídaesperado} = testvectors[vectornum];
    end// check results on falling edge of clk-
    always @ (negedge clk)
    begin
        if (~reset) begin // skip during reset
            if (saída != saídaesperado) begin // check result
                $display("Error: reset = %b, clock = %b, opcode = %b ,e funct = %b", {reseste}, {clk}, {opcode}, {funct});
                $display(" outputs = %b (%b esperado)", saída, saídaesperado);
                errors = errors + 1;
            end else begin
                $display("OK: reset: %b, clock: %b, opcode: %b, funct: %b | saída: %b (%b esperado)", resete, clk, opcode, funct, saída, saídaesperado);
            end
            vectornum = vectornum + 1;
            if (vectornum == $size(testvectors)) begin
                $display("%d tests completed with %d errors", vectornum, errors);
                $finish;
            end
        end
    end
endmodule
```

Simulação

A validação do modelo RTL foi realizada através da simulação do testbench no ModelSim. Esta simulação foi realizada diretamente na plataforma do ModelSim por questões de facilidade.

Inicialmente, deve-se criar o projeto com os arquivos do MIPS, tanto o .sv quanto o testbench.sv e ambos são compilados. Neste caso, não foi apresentando erros.

O arquivo então é simulado e obtém-se o seguinte resultado

