DOI: 10.53660/CLM-2074-23P21

RISC processor implementation 32-bit MIPS-based: an approach to teaching and learning

Implementação do processador RISC baseado em MIPS de 32 bits: uma abordagem para o ensino e aprendizagem

Received: 2023-09-03 | Accepted: 2023-10-05 | Published: 2023-10-11

Francisco Silva e Serpa
ORCID: https://orcid.org/0000-0002-4218-0674
Instituto Federal de Educação, Ciência e Tecnologia do Pará - Campus Parauapebas, Brasil
E-mail: francisco.serpa@ifpa.edu.br
Alan Marcel Fernandes de Souza
ORCID: https://orcid.org/0000-0002-1656-5714
Instituto Federal de Educação, Ciência e Tecnologia do Pará - Campus Parauapebas, Brasil
E-mail: alan.marcel@ifpa.edu.br
Hélio Fernando Bentzen Pessoa Filho
ORCID: https://orcid.org/0000-0002-3089-1639
Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco - Campus Afogados da Ingazeira, Brasil
E-mail: helio.filho@afogados.ifpe.edu.br
Kassio Derek Nogueira Cavalcante
ORCID: https://orcid.org/0009-0004-5632-9558
Instituto Federal de Educação, Ciência e Tecnologia do Ceará - Campus Cedro, Brasil
E-mail: Kassioderek@hotmail.com

ABSTRACT

This article describes the development of the design of a processor based on the RISC architecture, taking the 32-bit MIPS microprocessor as a basis. The RISC architecture, which stands for Reduced Instruction Set Computer, is characterized by having a reduced instruction set, aiming to optimize the processor's overall performance. The designed MIPS processor follows a 5-stage pipeline, which comprises the instruction fetch, instruction decode, execution, preparation and memory access phases. The main objective of this article is to carry out the structural development of the processor, using the hardware description language. This implies the creation of a Verilog representation that will later be used to generate the extraction of the processor's logic circuit. Furthermore, the project involves generating a timing diagram that illustrates the temporal behavior of processor operations and, ultimately, the physical implementation of the processor core. This work seeks to contribute knowledge in the field of computer architecture, providing a practical implementation of a RISC processor based on the 32-bit MIPS architecture, which can be relevant both for educational purposes and for practical applications in embedded systems.

Keywords: MIPS; RISC; Verilog

RESUMO

Este artigo descreve o desenvolvimento do *design* de um processador baseado na arquitetura RISC, tomando como base o microprocessador MIPS de 32 bits. A arquitetura RISC, que significa *Reduced Instruction Set Computer*, é caracterizada por ter um conjunto de instruções reduzido, visando otimizar o desempenho geral do processador. O processador MIPS projetado segue um *pipeline* de 5 estágios, que compreende as fases de busca de instruções, decodificação de instruções, execução, preparação e acesso à memória. O objetivo principal deste artigo é realizar o desenvolvimento estrutural do processador, empregando a linguagem de descrição de hardware. Isso implica na criação de uma representação em Verilog que posteriormente será utilizada para gerar a extração do circuito lógico do processador. Além disso, o projeto envolve a geração de um diagrama de tempo que ilustra o comportamento temporal das operações do processador e, por fim, a implementação física do núcleo do processador. Este trabalho busca contribuir conhecimentos no campo da arquitetura de computadores, fornecendo uma implementação prática de um processador RISC baseado na arquitetura MIPS de 32 bits, o que pode ser relevante tanto para fins educacionais quanto para aplicações práticas em sistemas embarcados.

Palavras-chave: MIPS; RISC; Verilog

INTRODUÇÃO

Os processadores RISC (*Reduced Instruction Set Computer*) foram criados com o objetivo de fabricar um tipo de microprocessador que utiliza um conjunto pequeno e simples de instruções invés de uma variedade de instruções complexas e de modos de endereçamento, como é o caso da arquitetura CISC (*Complex Instruction Set Computer*). E a MIPS (*Microprocessor without Interlocked Pipeline Stages*) *Technologies* foi uma fabricante de computadores que projetou e vendeu vários microprocessadores RISC começando com o processador MIPS R2000, no ano de 1980 (ROTH et al., 2016).

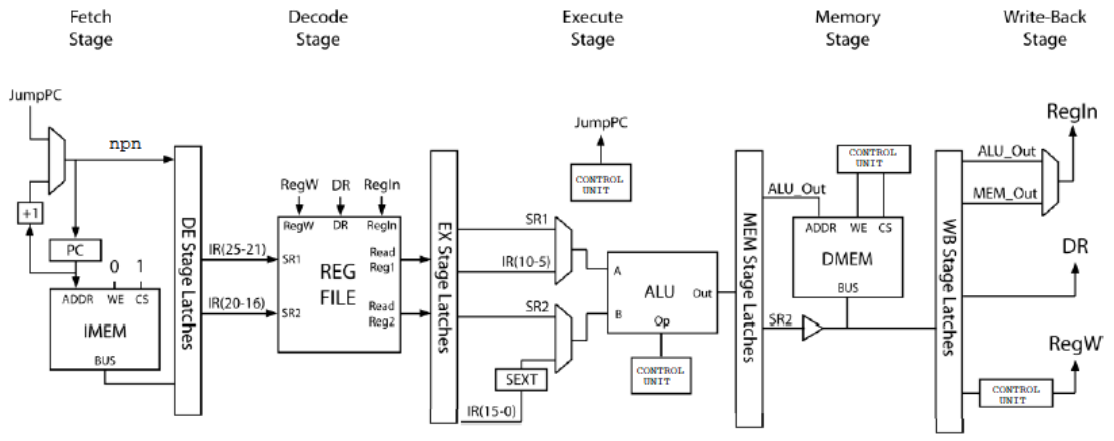
Além da simplicidade de instruções e modos de endereçamento, pode-se citar outros principais recursos característicos da maioria dos processadores RISC, como: comprimento uniforme da instrução de 32 bits, grande número de registros e arquitetura de carregamento e armazenamento. Esses recursos tornam o *design* RISC idealmente adequado para participar de uma tendência poderosa no mercado de processador embutido: o "*system-on-a-chip*" (JAIN, 2012). Existem várias empresas de semicondutores de implementação do processador RISC no "*system-on-a-chip*", como Atmel AVR, Micro Blaze, que são amplamente utilizados para aplicativos incorporados e DSP (*Digital Signal Processor*), devido a diferença no tempo necessário para acessar um registro em comparação com um local de memória, é muito mais rápido realizar operações em registradores de chip em vez da memória. A arquitetura permanece o mesmo para todos os processadores baseados em MIPS enquanto a implementações podem diferir como ciclo único, ciclo múltiplo e implementação em *pipeline*. Os microprocessadores RISC mais comuns são: ARM, SP ARC, MIPS e IBM's PowerPC (TOPIWALA & SARASWATHI, 2014).

Hoje em dia, a potência do equipamento e o parâmetro de desempenho são os aspectos mais importantes para aplicações embarcadas e portáteis (TOPIWALA & SARASWATHI, 2014). Porém, em qualquer circuito integrado, existe uma compensação entre potência, área e tempo de processamento que deve ser analisada. Neste artigo, será descrito e analisado o desenvolvimento de um projeto de microprocessador MIPS pelo diagrama de blocos RTL, código em Verilog, síntese e extração do circuito, tempo de processamento do circuito (*report timing*), simulação do diagrama de tempo das IOS *(Input/Output System*) e implementação física com auxílio do software Cadence®.

ARQUITETURA DO MICROPROCESSADOR MIPS

A arquitetura do microprocessador MIPS foi implementada usando cinco etapas em *pipeline*: *fetch, decode, execute, memory* e *write-black* (ASHOK & RAVI, 2017). A etapa de busca (*fetch*) utiliza um contador de programa (PC) que busca a instrução do módulo *memory* (MEM) e rastreia as instruções de 32 bits no formato hexadecimal, em seguida entra na etapa de decodificação (*Decode*) que identifica as instruções, conforme formatos preestabelecidos (R, I, J) mostrados na tabela 1. Devido aos formatos uniformes, os campos de instruções são usados diretamente para endereçamento do registro e geração de sinais de controle. Após a instrução ser identificada, a próxima etapa é executar (*Execute*) as instruções conforme o formato, as operações lógicas e aritméticas (ULA) ocorrem neste estágio, assim como operações de desvios (bne e beq) e salto (jr). E por fim, a quarta e quinta etapas são o acesso e regravação da memória (*memory* MEM) com objetivo de ler e escrever na memória de dados e em seguida gravar de volta o resultado no módulo *register*. A descrição da arquitetura em fases é mostrada na figura 1.

Figura 1 – *Datapath* do MIPS *Pipelined*



Fonte: ROTH et al. (2016)

Tabela 1 – Formato de instruções no MIPS

Format	Fields						Comments
	6 bits 31..26	5 bits 25..21	5 bits 20..16	5 bits 15..11	5 bits 10..6	6 bits 5..0	All MIPS instruction are 32 bits.
R-format	opcode	rs	rt	rd	shamt	F_Code (funct)	ALU instructions except immediate, Jump Register (JR)
I-format	opcode	rs	rt	address/immediate			Load, store, Immediate ALU, beq, bne
J-format	opcode	target address					Jump (J)

Fonte: ROTH et al. (2016)

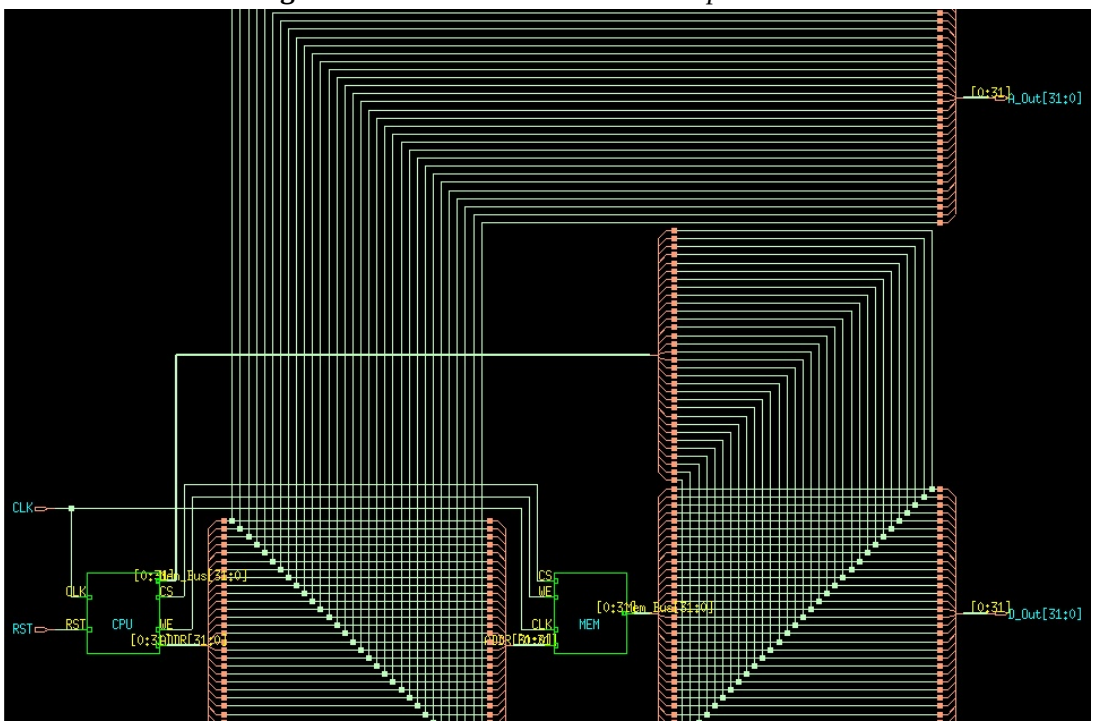
MATERIAIS E MÉTODOS

Os resultados apresentados neste trabalho foram obtidos através dos *softwares* RTL, *Genus Synthesis Solution, SimVision e Innovus* Cadence®. Os principais passos da metodologia de projeto foram: especificações das restrições de projeto no arquivo SDC (*Synopsys Design Constraints*) conforme apêndice I, verificação e aplicação do script TCL (*Tool Command Language*) para a síntese lógica conforme anexo I, Utilização de DFTs (*Design for test*) na arquitetura, utilização da biblioteca de células LEF (*Layout Exchange Format*), aplicação dos parâmetros de processo PVT (*process/voltage/temperature*), realização da verificação formal do circuito através do LEC (*Logic Equivalence Checker*), definição do módulo *top*, o qual faz a integração do *netlist* com os PADS de entrada, saídas e alimentação, definição do posicionamento dos PADs de entrada, saídas e alimentação, realização do *Floorplan* (relação de tamanho do circuito, percentual de ocupação e espaçamento dos extremos para o núcleo e vários outros detalhes) do circuito, realização do *Power Planning* do circuito adicionando os anéis e os *stripes* de VDD e GND do circuito, realização do *Placement* do circuito, realização do roteamento completo do circuito, verificação do *timing closure* de *setup* e *hold* para as etapas de pré-CTS, pós-CTS e pós-Route, inserção das *Spare Cells* e das *Filler Cells*; realização do *Metal fill*; realização das verificações finais do projeto, DRC (regras de projeto da terminologia utilizada), Geometria, Conectividade, densidade de metal; realização da análise de potência.

RESULTADO DAS SIMULAÇÕES

O processo de *synthesis* realizado pelos softwares Genus Synthesis Solution Cadence® transformará a descrição em Verilog (Anexo 1) do módulo Complete MIPS em um circuito em blocos com entradas e saídas, conforme figura 2. Foram adicionadas as restrições (*constraints*) de capacitância de comutação e de tempo das entradas, das saídas e do *clock* necessárias para verificação de desempenho interno do processador através da ferramenta *report timing*. As especificações de tempo estão descritas no anexo 2.

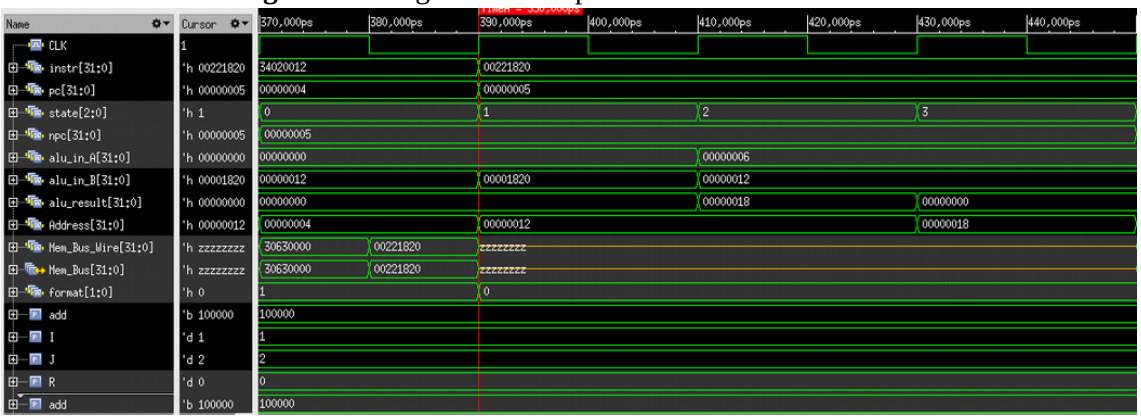
Figura 2 – *Schematic* do módulo *Complete* MIPS



Fonte: Autoria própria

O *status* das entradas e saídas de todo circuito podem ser analisados pelo diagrama de tempo simulado pelo *software SimVision* Cadence® conforme figura 3.

Figura 3 – Diagrama de tempo do circuito no *SimVision*



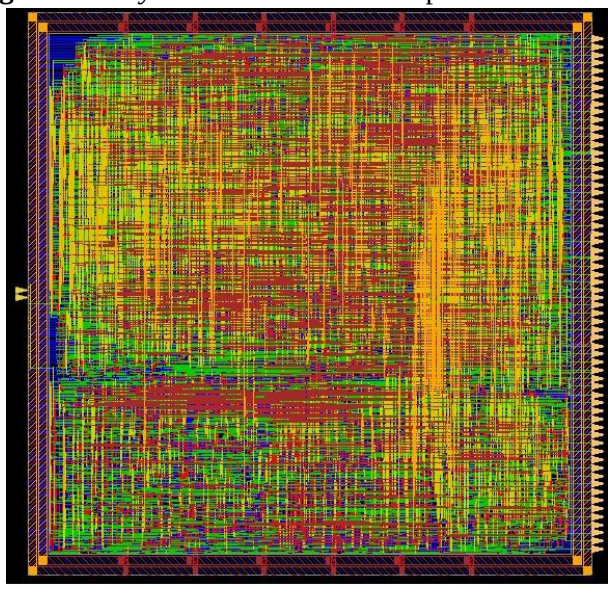
Fonte: Autoria própria

Na figura 3 mostra que o state 0 (*fetch*) buscará na memória a instrução armazenada 0022182016, já no state 1 *(decode*) será convertida em 10001000011000001000002 e conforme formato de instruções (Tabela 1) e descrição em Verilog (Anexo 1) é definida como formato R e a opção de operação como adição. Em seguida, no state 2 (*execute*) coletasse as informações dos endereços $1 (alu_in_A) e $2 (alu_in_B) e realiza-se a adição que fica armazenada em $3

(alu_result). Por fim, no state 3 (*memory*) a informação é repassada para *Address* e preparada para ser armazenada na memória.

O *layout* do processador MIPS é mostrado na figura 4 através do simulador *Innovus* Cadence®. Nesta etapa, foram especificadas as barras de alimentação em volta do núcleo (W= 12 um, S = 2 um, Offs=4 um), espessuras (metal 5 e 6) para VDD e VSS, as barras de alimentação cortando o núcleo na vertical (W = 6 um, S = 2 um, Set= 100 um), os pinos de entrada a esquerda e de saída a direita com espaçamento entre eles de 11,2 um, a adição das células digitais, preenchimento com os *fillers*, *pré-route* e a verificação de violação da geometria, DRC e conectividade.

Figura 4 – *Layout* do núcleo do microprocessador MIPS



Fonte: Autoria própria

Para verificação de desempenho utilizou-se a ferramenta *report timing* logo após a *synthesis*, conforme exibido no apêndice II, pode-se destacar: O *Slack* positivo de 68 ρs que representa a diferença entre os tempos requeridos e de chegada, e o *Data Path* de 2323 ρs que pela lista dos blocos digitais tem como caminho crítico o bloco CPU_g75/Z, o qual apresenta o maior atraso no processamento com *delay* de 511 ρs e *transition* de 546 ρs. Já as potências extraídas, pode-se destacar a Potência estática (*Leakage*) de 1,404 μW e a Potência dinâmica (*Switching*) de 0,145 W do módulo Complete_MIPS, conforme apêndice III.

CONCLUSÃO

Neste artigo, foi apresentado o *design* de processador RISC baseado em MIPS de 32 bits e implementado com funcionalidades de *pipeline*. Cada instrução foi executada em um ciclo de *clock* com 5 etapas. O *design* foi implementado usando Verilog-HDL e sintetizado usando o software RTL Cadence®. Este tipo de projeto pode ser usado para realizar simulações e testes da arquitetura RISC e principalmente tornar o ensino desse importante tópico da engenharia da computação menos abstrato, pois permite que o aluno pratique a teoria.

REFERÊNCIAS

ASHOK, Agineti; RAVI, V. ASIC design of MIPS based RISC processor for high performance. In: **2017 International Conference on Nextgen Electronic Technologies: Silicon to Software (ICNETS2)**. IEEE, 2017. p. 263-269.

JAIN, Neeraj. VLSI design and optimized implementation of a MIPS RISC processor using XILINX tool. **International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE)**, v. 1, n. 10, 2012.

TOPIWALA, Mohit N.; SARASWATHI, N. Implementation of a 32-bit MIPS based RISC processor using Cadence. In: 2014 **IEEE International Conference on Advanced Communications**, Control and Computing Technologies. IEEE, 2014. p. 979-983.

ROTH, Charles; JOHN, Lizy Kurian; LEE, Byeong Kil. **Digital System Design Using Verilog**, Cengage Learning US, 2016.

ANEXO I - CÓDIGO COMPLETO EM VERILOG

```
module Complete_MIPS(CLK, RST, A_Out, D_Out);
input CLK;
input RST;
output [31:0] A_Out;
output [31:0] D_Out;
wire CS, WE;
wire [31:0] ADDR, Mem_Bus;
assign A_Out = ADDR;
assign D_Out = Mem_Bus;
MIPS CPU(CLK, RST, CS, WE, ADDR, Mem_Bus);
Memory MEM(CS, WE, CLK, ADDR, Mem_Bus);
endmodule

module Memory(CS, WE, CLK, ADDR, Mem_Bus);
input CS, WE, CLK;
input [31:0] ADDR;
inout [31:0] Mem_Bus;
reg [31:0] data_out;
reg [31:0] RAM [0:127];
integer i;
initial begin
 RAM[1] = 32'h30000000;
...
 RAM[127] = 32'h00000000;
end
assign Mem_Bus = ((CS == 1'b0) || (WE == 1'b1)) ? 32'bZ : data_out;
always @(negedge CLK) begin
if((CS == 1'b1) && (WE == 1'b1))
RAM[ADDR] <= Mem_Bus[31:0];
data_out <= RAM[ADDR];
end
endmodule

`define opcode instr[31:26]
`define srl instr[25:21]
`define sr2 instr[20:16]
`define f_code instr[5:0]
`define numshift instr[10:6]
module MIPS (CLK, RST, CS, WE, ADDR, Mem_Bus);
input CLK, RST;
output reg CS, WE;
output [31:0] ADDR;
inout [31:0] Mem_Bus;
//special instructions (opcode == 000000), values of F code (bits 5-0):
parameter add = 6'b100000;
parameter sub = 6'b100010;
parameter xor1 = 6'b100110;
parameter and1 = 6'b100100;
parameter or1 = 6'b100101;
parameter slt = 6'b101010;
parameter srl = 6'b000010;
parameter sll = 6'b000000;
parameter jr = 6'b001000;
//non-special instructions, values of opcodes:
parameter addi = 6'b001000;
parameter andi = 6'b001100;
parameter ori = 6'b001101;
parameter lw = 6'b100011;
parameter sw = 6'b101011;
parameter beq = 6'b000100;
parameter bne = 6'b000101;
parameter j = 6'b000010;
//instruction format
parameter R = 2'd0;
parameter I = 2'd1;
parameter J = 2'd2;
//internal signals
reg [5:0] op, opsave;
wire [1:0] format;
reg [31:0] instr, pc, npc, alu_result;
wire [31:0] imm_ext, alu_in_A, alu_in_B, reg_in, readreg1, readreg2;
reg [31:0] alu_result_save;
reg alu_or_mem, alu_or_mem_save, regw, writing, reg_or_imm, reg_or_imm_save;
```

```
74  reg fetchDorI;
75  wire [4:0] dr, sr1, sr2;
76  reg [2:0] state, nstate;
77  //combinational
78  assign imm_ext = (instr[15] == 1)? {16'hFFFF, instr[15:0]} : {16'h0000,instr[15:0]};
    //Sign extend immediate field
79  assign dr = (format == R)? instr[15:11] : instr[20:16];
80  //Destination Register MUX (MUX1)
81  assign alu_in_A = readreg1;
82  assign alu_in_B = (reg_or_imm_save)? imm_ext : readreg2;
83  //ALU MUX (MUX2)
84  assign reg_in = (alu_or_mem_save)? Mem_Bus : alu_result_save;
85  //Data MUX
86  assign format = (`opcode == 6'd0)? R : ((`opcode == 6'd2)? J : I);
87  assign Mem_Bus = (writing)? readreg2 : 32'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
88  //drive memory bus only during writes
89  assign ADDR = (fetchDorI)? pc : alu_result_save;
90  //ADDR Mux
91  Register REG(CLK, regw, dr, `sr1, `sr2, reg_in, readreg1, readreg2);
92  initial begin
93  op = and1;
94  opsave = and1;
95  state = 3'b0;
96  nstate = 3'b0;
97  alu_or_mem = 0;
98  regw = 0;
99  fetchDorI = 0;
100 writing = 0;
101 reg_or_imm = 0;
102 reg_or_imm_save = 0;
103 alu_or_mem_save = 0;
104 end
105 always @(*)
106 begin
107 fetchDorI = 0; CS = 0; WE = 0; regw = 0; writing = 0; alu_result = 32'd0;
108 npc = pc; op = jr; reg_or_imm = 0; alu_or_mem = 0;
109 case (state)
110 0: begin //fetch
111 npc = pc + 32'd1; CS = 1; nstate = 3'd1;
112 fetchDorI = 1;
113 end
114 1: begin //decode
115 nstate = 3'd2; reg_or_imm = 0; alu_or_mem = 0;
116 if (format == J) begin //jump, and finish
117 npc = {pc[31:26], instr[25:0]};
118 nstate = 3'd0;
119 end
120 else if (format == R) //register instructions
121 op = `f_code;
122 else if (format == I) begin //immediate instructions
123 reg_or_imm = 1;
124 if(`opcode == lw) begin
125 op = add;
126 alu_or_mem = 1;
127 end
128 else if ((`opcode == lw)||(`opcode == sw)||(`opcode == addi))
129 op = add;
130 else if ((`opcode == beq)||(`opcode == bne)) begin
131 op = sub;
132 reg_or_imm = 0;
133 end
134 else if (`opcode == andi) op = and1;
135 else if (`opcode == ori) op = or1;
136 end
137 end
138 2: begin //execute
139 nstate = 3'd3;
140 if (opsave == and1) alu_result = alu_in_A & alu_in_B;
141 else if (opsave == or1) alu_result = alu_in_A | alu_in_B;
142 else if (opsave == add) alu_result = alu_in_A + alu_in_B;
143 else if (opsave == sub) alu_result = alu_in_A - alu_in_B;
144 else if (opsave == srl) alu_result = alu_in_B >> `numshift;
145 else if (opsave == sll) alu_result = alu_in_B << `numshift;
```

```
146   else if (opsave == slt) alu_result = (alu_in_A < alu_in_B)? 32'd1 : 32'd0;
147   else if (opsave == xorl) alu_result = alu_in_A ^ alu_in_B;
148   if (((alu_in_A == alu_in_B)&&(`opcode == beq)) || ((alu_in_A != alu_in_B)&&(`opcode ==
       bne))) begin
149   npc = pc + imm_ext;
150   nstate = 3'd0;
151   end
152   else if ((`opcode == bne)||(`opcode == beq)) nstate = 3'd0;
153   else if (opsave == jr) begin
154   npc = alu_in_A;
155   nstate = 3'd0;
156   end
157   end
158   3: begin //prepare to write to mem
159   nstate = 3'd0;
160   if ((format == R)||(`opcode == addi)||(`opcode == andi)||(`opcode == ori))
161   regw = 1;
162   else if (`opcode == sw) begin
163   CS = 1;
164   WE = 1;
165   writing = 1;
166   end
167   else if (`opcode == lw) begin
168   CS = 1;
169   nstate = 3'd4;
170   end
171   end
172   4: begin
173   nstate = 3'd0;
174   CS = 1;
175   if (`opcode == lw) regw = 1;
176   end
177   endcase
178   end //always
179   always @(posedge CLK) begin
180   if (RST) begin
181   state <= 3'd0;
182   pc <= 32'd0;
183   end
184   else begin
185   state <= nstate;
186   pc <= npc;
187   end
188   if (state == 3'd0) instr <= Mem_Bus;
189   else if (state == 3'd1) begin
190   opsave <= op;
191   reg_or_imm_save <= reg_or_imm;
192   alu_or_mem_save <= alu_or_mem;
193   end
194   else if (state == 3'd2) alu_result_save <= alu_result;
195   end //always
196   endmodule
197
198   module Register(CLK, RegW, DR, SR1, SR2, Reg_In, ReadReg1, ReadReg2);
199   input CLK, RegW;
200   input [4:0] DR, SR1, SR2;
201   input [31:0] Reg_In;
202   output reg [31:0] ReadReg1, ReadReg2;
203   reg [31:0] REG [0:31];
204   integer i;
205   initial begin
206   ReadReg1 = 0;
207   ReadReg2 = 0;
208   for (i = 0; i < 32; i = i+1) begin
209   REG[i] = 0;
210   end
211   end
212   always @(posedge CLK) begin
213   if(RegW == 1'b1)
214   REG[DR] <= Reg_In[31:0];
215   ReadReg1 <= REG[SR1];
216   ReadReg2 <= REG[SR2];
217   end
```

```
218  endmodule
219
220  module mips_testbench;
221  reg CLK;
222  wire CS, WE;
223  parameter N = 10;
224  reg[31:0] expected[N:1];
225  wire[31:0] Address, Address_Mux, Mem_Bus_Wire;
226  reg[31:0] AddressTB;
227  wire WE_Mux, CS_Mux,compare;
228  reg init, rst, WE_TB, CS_TB;
229  integer i;
230  MIPS CPU(CLK, rst, CS, WE, Address, Mem_Bus_Wire);
231  Memory MEM(CS_Mux, WE_Mux, CLK, Address_Mux, Mem_Bus_Wire);
232  assign Address_Mux = (init)? AddressTB : Address;
233  assign WE_Mux = (init)? WE_TB : WE;
234  assign CS_Mux = (init)? CS_TB : CS;
235  always
236  #10 CLK = ~CLK;
237  initial begin
238  expected[1] = 32'h00000006; // $1 content=6 decimal
239  expected[2] = 32'h00000012; // $2 content=18 decimal
240  expected[3] = 32'h00000018; // $3 content=24 decimal
241  expected[4] = 32'h0000000C; // $4 content=12 decimal
242  expected[5] = 32'h00000002; // $5 content=2
243  expected[6] = 32'h00000016; // $6 content=22 decimal
244  expected[7] = 32'h00000001; // $7 content=1
245  expected[8] = 32'h00000120; // $8 content=288 decimal
246  expected[9] = 32'h00000003; // $9 content=3
247  expected[10] = 32'h00412022; // $10 content=5th instr
248  CLK = 0;
249  end
250  always begin
251  rst = 1;
252  @(posedge CLK); //wait until posedge CLK
253  //Initialize the instructions from the testbench
254  init <= 1; CS_TB <= 1; WE_TB <= 1;
255  @(posedge CLK);
256  CS_TB <= 0; WE_TB <= 0; init <= 0;
257  @(posedge CLK);
258  rst <= 0;
259  for(i = 1; i <= N; i = i+1) begin
260  @(posedge WE); // When a store word is executed
261  @(negedge CLK);
262  if (Mem_Bus_Wire == expected[i]);
263  $stop;
264  end
265  end
266  endmodule
267
268
269
```

APÊNDICE I - SDC (*SYNOPSYS DESIGN CONSTRAINTS*)

```
## Logical/Physical synthesis constraints ##
## DEFINE VARS
set sdc_version 1.5
set_load_unit -picofarads 1

create_clock -name {CLK} -period 5 [get_ports {CLK}]
set_false_path -from [get_ports {RST}]

## INPUTS
set_input_delay -clock CLK -max 1 [all_inputs]
set_input_transition -min -rise 0.0005 [all_inputs]
set_input_transition -max -rise 1 [all_inputs]
set_input_transition -min -fall 0.0005 [all_inputs]
set_input_transition -max -fall 1 [all_inputs]

## OUTPUTS
set_output_delay -clock CLK -max 1 [all_outputs]
set_load -min 0.0005 [all_outputs]
set_load -max 1 [all_outputs]

```

APÊNDICE II - *REPORT TIMING*

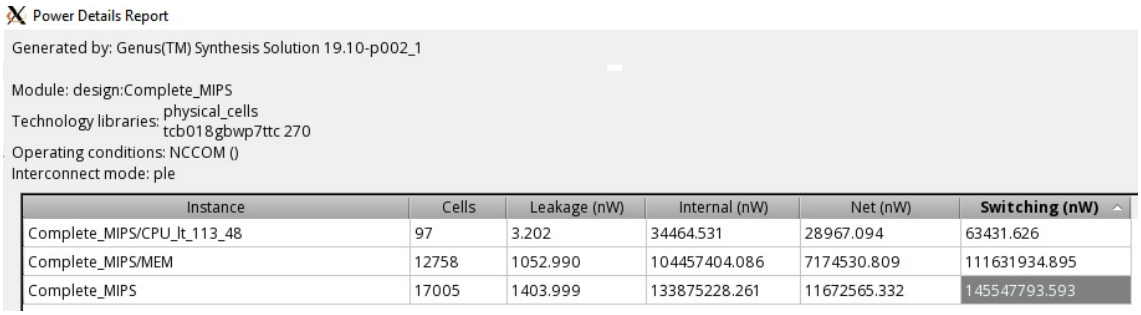
```
Path 1: MET (68 ps) Setup Check with Pin MEM/RAM_reg[1][8]/CPN->D
          Group: CLK
     Startpoint: (R) CPU_instr_reg[31]/CP
          Clock: (R) CLK
       Endpoint: (F) MEM/RAM_reg[1][8]/D
          Clock: (F) CLK

                    Capture       Launch
       Clock Edge:+    2500            0
       Src Latency:+      0            0
       Net Latency:+      0 (I)        0 (I)
           Arrival:=   2500            0

             Setup:-    108
     Required Time:=   2392
      Launch Clock:-      0
         Data Path:-   2323
             Slack:=     68

#-------------------------------------------------------------------------------------------------------------
#    Timing Point      Flags    Arc    Edge     Cell        Fanout  Load  Trans Delay Arrival Instance
#                                                                    (fF)   (ps)  (ps)   (ps)  Location
#-------------------------------------------------------------------------------------------------------------
  CPU_instr_reg[31]/CP -       -      R     (arrival)        5323     -     0     -       0    (-,-)
  CPU_instr_reg[31]/Q  -       CP->Q  R     EDFQD2BWP7T         5  33.2   106   256     256    (-,-)
  CPU_g11918/ZN        -       A1->ZN R     INR2XD0BWP7T        2  15.5   205   180     436    (-,-)
  CPU_g11915/ZN        -       A2->ZN F     ND3D1BWP7T          4  25.8   181   144     580    (-,-)
  CPU_g11818/ZN        -       A1->ZN F     IND2D1BWP7T         1  13.2    88   164     744    (-,-)
  CPU_g11816/ZN        -       A1->ZN R     NR2D2BWP7T          4  36.2   238   155     898    (-,-)
  CPU_g11805/Z         -       I->Z   R     BUFFD1P5BWP7T       2  27.8   117   129    1027    (-,-)
  CPU_g11804/ZN        -       I->ZN  F     INVD2BWP7T          2  39.4    68    60    1087    (-,-)
  CPU_g11802/ZN        -       I->ZN  R     CKND4BWP7T          8 108.6   156   108    1196    (-,-)
  CPU_g75/Z            -       OE->Z  F     BUFTD6BWP7T         4 1085.7  546   511    1707    (-,-)
  MEM/g21305/ZN        -       I->ZN  R     INVD6BWP7T        128 661.2   607   482    2189    (-,-)
  MEM/g20074/ZN        -       A2->ZN F     MOAI22D0BWP7T       1   9.5   196   134    2323    (-,-)
  MEM/RAM_reg[1][8]/D  <<<     -      F     DFND0BWP7T          1     -     -     0    2323    (-,-)
#-------------------------------------------------------------------------------------------------------------
```

APÊNDICE III - RELATÓRIO COMPLETO DE POTÊNCIAS



Power Details Report

Generated by: Genus(TM) Synthesis Solution 19.10-p002_1

Module: design:Complete_MIPS
Technology libraries: physical_cells
tcb018gbwp7ttc 270
Operating conditions: NCCOM ()
Interconnect mode: ple

Instance	Cells	Leakage (nW)	Internal (nW)	Net (nW)	Switching (nW)
Complete_MIPS/CPU_lt_113_48	97	3.202	34464.531	28967.094	63431.626
Complete_MIPS/MEM	12758	1052.990	104457404.086	7174530.809	111631934.895
Complete_MIPS	17005	1403.999	133875228.261	11672565.332	145547793.593