



ARQUITETURA DE COMPUTADORES

3º Projeto – CPU COM PIPELINE

Membros: Christopher de Oliveira Souza	RA: 18726430
Murilo De Paula Araujo	17747775
Leonardo Sanavio	18054395
Beatriz Morelatto Lorente	18071597

Professor: Roberto Santana De Rezende Edmar

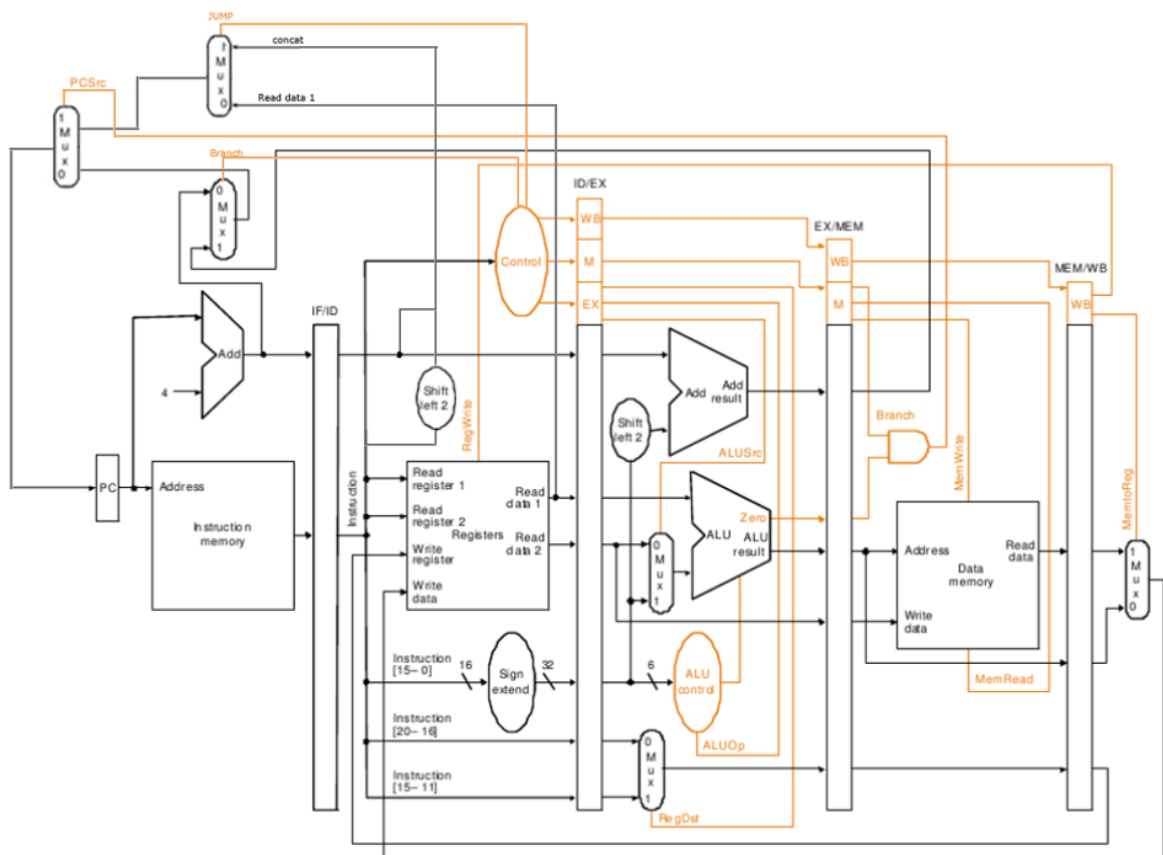
ÍNDICE

1. Descrição textual do projeto com a topologia da CPU
2. Especificação e detalhes relevantes do projeto
3. Desenvolvimento e descrição da implementação
4. Resultados obtidos e funcionamento
5. Conclusões e análise dos resultados obtidos
6. Bibliografia

1. INTRODUÇÃO

O projeto realizado é uma implementação de uma CPU com Pipeline em linguagem VHDL, o grupo partiu do seu datapath até sua real implementação no *Intel Quartus* onde foi programada, programando componente por componente, e em seguida implementada como um todo.

A implementação apresentada neste trabalho foi baseada no MIPS original de Hennessy e Patterson [HEN98] e na versão monociclo de [HAM00]. Na imagem abaixo, podemos observar o datapath com as alterações necessárias para realizar as instruções de jumps.



O MIPS possui três formatos de instruções diferentes: formato R, formato I e formato J. Onde as instruções do tipo R executam operações de lógicas e aritmética somente em dados armazenados nos registradores, as instruções do tipo I possuem dois tipos, as que fazem referência a memória como *load* e *store* e as instruções de *branch* (salto condicional).

A CPU tem que ser capaz de executar as seguintes instruções mostradas abaixo na tabela:

Categoria	Nome	Sintaxe da instrução	Significado	Formato	Notas	OPCODE
Arithmetic	Add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	R	Adds two registers	000001
	Subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	R	Subtracts two registers	000010
	Add immediate	addi \$1,\$2,CONST	$\$1 = \$2 + \text{CONST}$	I	Used to add constants	000011
	Sub immediate	subi \$1,\$2,CONST	$\$1 = \$2 - \text{CONST}$	I	Used to sub constants	000100
Data Transfer	Load word	lw \$1,CONST(\$2)	$\$1 = \text{Memory}[\$2 + \text{CONST}]$	I	Loads the word stored from: MEM[\$s2+CONST] and the following 3 bytes	000101
	Store word	sw \$1,CONST(\$2)	$\text{Memory}[\$2 + \text{CONST}] = \1	I	Stores a word into: MEM[\$2+CONST] and the following 3 bytes	000110
Logical	And	and \$1,\$2,\$3	$\$1 = \$2 \& \$3$	R	Bitwise and	000111
	And immediate	andi \$1,\$2,CONST	$\$1 = \$2 \& \text{CONST}$	I		001000
	Or	or \$1,\$2,\$3	$\$1 = \$2 \$3$	R	Bitwise or	001001
	Or immediate	ori \$1,\$2,CONST	$\$1 = \2CONST	I		001010
Conditional branch	Branch on equal	beq \$1,\$2,CONST	if (\$1 == \$2) go to PC+4+CONST	I	Goes to the instruction at the specified address if two registers are equal	001011
Unconditional	Jump	j CONST	goto address CONST	J	Unconditionally jumps to the instruction at the specified address	001100
jump	Jump register	jr \$1	goto address \$1	R	Jumps to the address contained in the specified register	001101

A memória do MIPS é endereçada a byte e, portanto, adiciona-se quatro para o cálculo do próximo endereço da palavra de 32 bits da memória, sendo os endereços de memória de 32 bits.

2. ESPECIFICAÇÃO

No projeto da CPU foi utilizado 5 registradores ao todo que são maiormente utilizados no armazenamento dos dados na passagem de um estágio para o outro. Formato e tipo das instruções utilizadas e aceitas pela CPU construída foram:

Instrução do Tipo R		
Campos	Bits	Notas
OPCODE	6	Operação básica da instrução (opcode)
RS	5	O primeiro registrador fonte
RT	5	O segundo registrador fonte
RD	5	O registrador destino
SHAMT	5	Shift Amount, para instruções de deslocamento
FUNCT	6	Function. Seleciona variações das operação especificada pelo opcode

Instrução do Tipo I		
Campos	Bits	Notas
OPCODE	6	Operação básica da instrução (opcode)
RS	5	O primeiro registrador fonte
RT	5	O segundo registrador fonte
ENDEREÇO	16	Endereço da instrução

Instrução do Tipo J		
Campos	Bits	Notas
OPCODE	6	Operação básica da instrução (opcode)
ENDEREÇO	16	Endereço da instrução

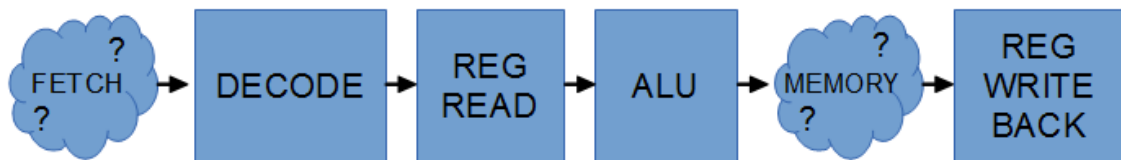
Foram utilizados vários registradores para armazenamento dos dados, abaixo temos a tabela de registradores e seus códigos:

Registrador	Código
\$R0	00000
\$R1	00001
\$R2	00010
\$R3	00011
\$R4	00100
\$R5	00101
\$R6	00110
\$R7	00111
\$R8	01000
\$R9	01001
\$R10	01010
\$R11	01011
\$R12	01100
\$R13	01101
\$R14	01110
\$R15	1111
\$R16	10000
\$R17	10001
\$R18	10010
\$R19	10011
\$R20	10100
\$R21	10101
\$R22	10110
\$R23	10111
\$R24	11000
\$R25	11001
\$R26	11010
\$R27	11011
\$R28	11100
\$R29	11101
\$R30	11110
\$R31	11111

A implementação do MIPS desenvolvida neste trabalho cobre um subconjunto das instruções do MIPS. Como podemos notar, as instruções do tipo R possuem um campo apenas para determinar qual operação será executada pelo ULA (Unidade Lógica Aritmética) da CPU.

3- DESENVOLVIMENTO

Funcionamento dos estágios de processamento de uma instrução:



Após o estudo da organização e arquitetura do processador, desenvolveu-se separadamente cada um dos componentes que compõem os estágios do pipeline do MIPS. Depois de testados individualmente, os estágios foram conectados e testados novamente.

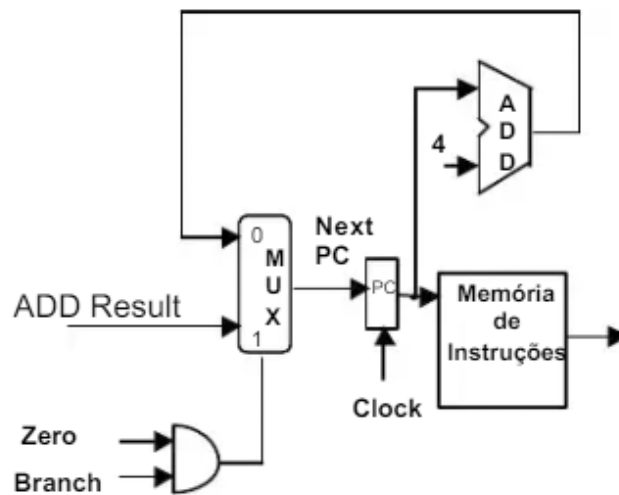
A implementação desenvolvida no trabalho trabalha com 5 estágios, o primeiro estágio de *fetch* (busca) da instrução, o estágio de decodificação da instrução, o de execução e o de acesso a memória e escrita de volta nos registradores (*Writeback*).

Desenvolvimento de cada estágio:

3.1 ESTAGIO DE BUSCA

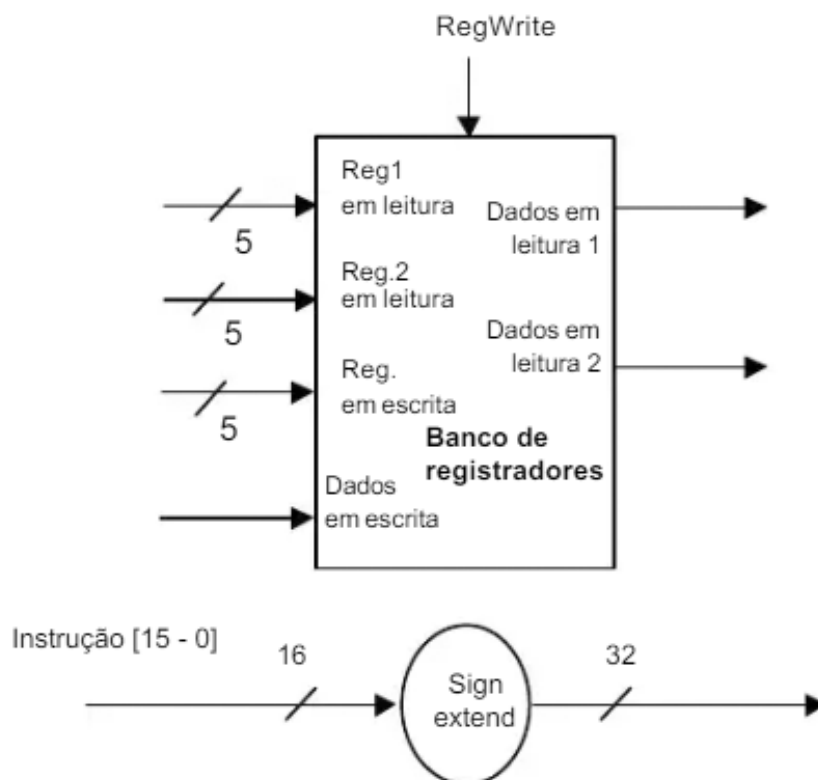
Este estágio é responsável pela busca da instrução que será executada, a partir da ordem definida no programa. Nele também é realizada a incrementação do PC, somando 4 para já calcular o endereço da próxima instrução a ser executada.

Representação no circuito:



3.2 Estágio de Decodificação

Neste estágio realiza-se a decodificação da instrução, e ele também é responsável pelo banco de registradores. O banco de registradores foi implementado como uma entidade separada, podendo ser determinado na instânciação o número de registradores. Representação no Circuito:



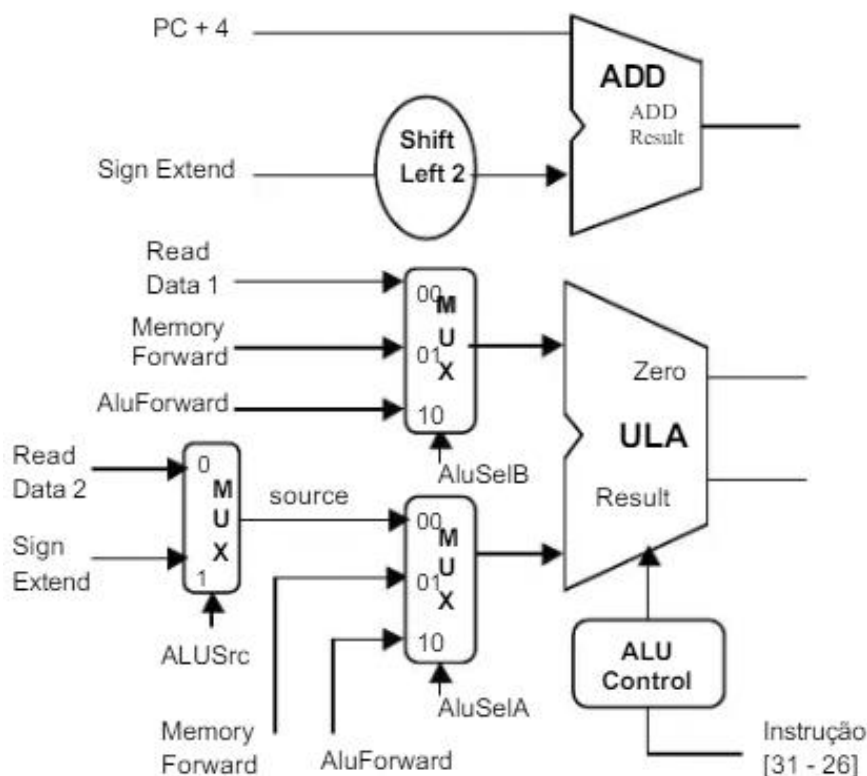
O banco de registradores implementado é um componente composto por um conjunto de registradores que podem ser acessados de forma organizada. podem ser executadas operações de leitura dos dados anteriormente gravados e de escrita de dados para modificar as informações internas. O banco de registradores é composto por 32 registradores de 32 bits cada.

3.3 ESTÁGIO DE EXECUÇÃO

Neste estágio, dependendo da instrução, existem duas unidades que as podem executar, a unidade para calculo de endereço de *branch* (apenas soma), e a ULA para realizar as operações aritmética e logica.

Segundo os sinais de controle gerados pela Unidade de Controle, poderá ser selecionado o segundo operando da ULA, que pode ser um dado lido de um registrador, como nos casos das instruções tipo R e de *branch*, ou um endereço, como nas instruções de *load* e *store*.

Representação no circuito:



3.3.1 UNIDADE LÓGICA ARITMÉTICA

O controle da ULA tem como entrada o opcode, os 6 bits mais significativos da palavra de instrução e mais os 6 bits menos significativos da instrução, gerando a partir destes um sinal de três bits para a ULA.

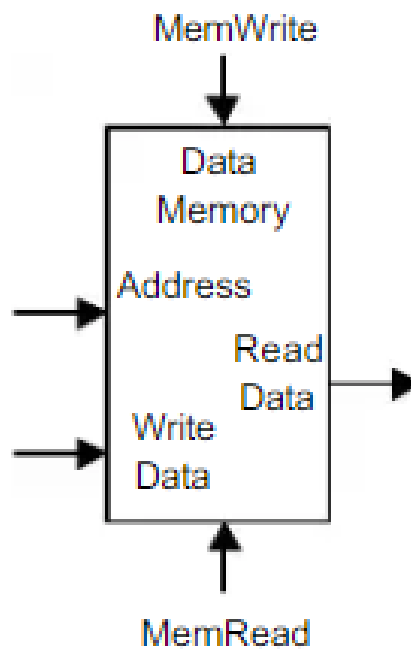
Abaixo está o modo definido pelo grupo para os *opcodes* das instruções aceitas pela ULA construída no projeto.

Category	Name	OPCODE
Arithmetic	Add	000001
	Subtract	000010
	Add immediate	000011
	Sub immediate	000100
Data Transfer	Load word	000101
	Store word	000110
Logical	And	000111
	And immediate	001000
	Or	001001
	Or immediate	001010
Conditional branch	Branch on equal	001011
Unconditional	Jump	001100
jump	Jump register	001101

3.4 ESTÁGIO DE MEMÓRIA

Neste estágio ocorre a leitura e/ou a escrita da na memória no mesmo ciclo. A escrita só é realizada no final do ciclo e se o sinal for recebido o sinal de *MemWrite*, que indica uma instrução de *store*.

Representação no circuito:



3.5 ESTÁGIO DE *WRITEBACK*

Nesta última etapa, o resultado da ULA é escrito no registrador de destino.

Os sinais de controle, gerados na UC (unidade de controle) definem de onde virá o valor de escrita (memória de dados ou resultado da ULA) e se haverá escrita no banco de registradores.

3.6 UNIDADE DE CONTROLE

É responsável por gerar os sinais de controle adequados, para ser feito o que se pede na instrução que será executada. Sinais de controle da CPU com pipeline:

RegWrite - sinal que habilita a escrita no banco de registradores;

MemToReg - sinal que indica que o dado a ser escrito no registrador de destino é um dado vindo da memória;

AluSrc - sinal que indica qual o segundo operando da ULA;

MemWrite - sinal que habilita a escrita na memória;

MemRead - sinal que habilita a leitura da memória;

Branch - sinal que indica uma instrução de branch;

RegDst - sinal utilizado para selecionar o registrado destino.

AluOp1 – primeiro operando da ULA.

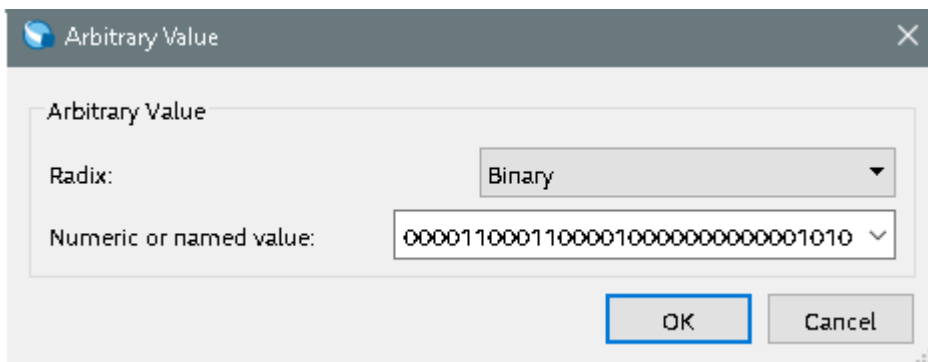
AluOp2 – segundo operando da ULA.

Jump – se a instrução for tipo J, o valor escrito no PC será o que foi incrementado em 4 no somador.

4. RESULTADOS

4.1 TESTES REALIZADOS

--Instrução ADDI



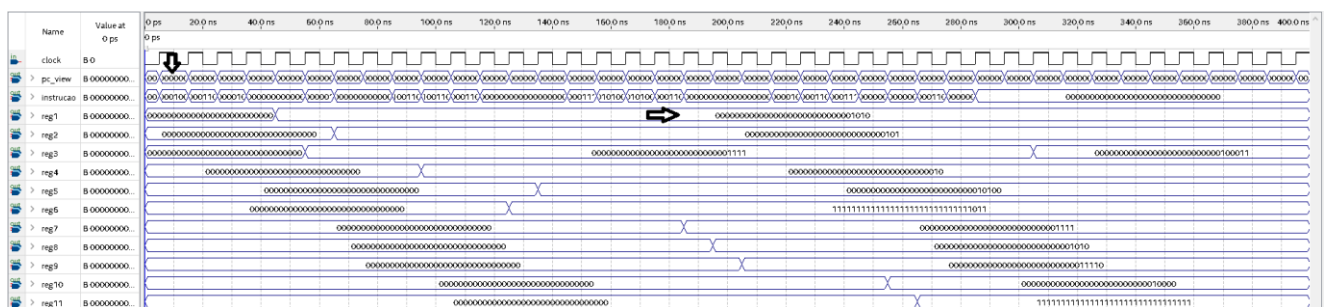
Arbitrary Value

Radix: Binary

Numeric or named value: 000011000110000100000000000001010

OK Cancel

-- ADDi \$1, \$3, 10



-- Instrução SUBI

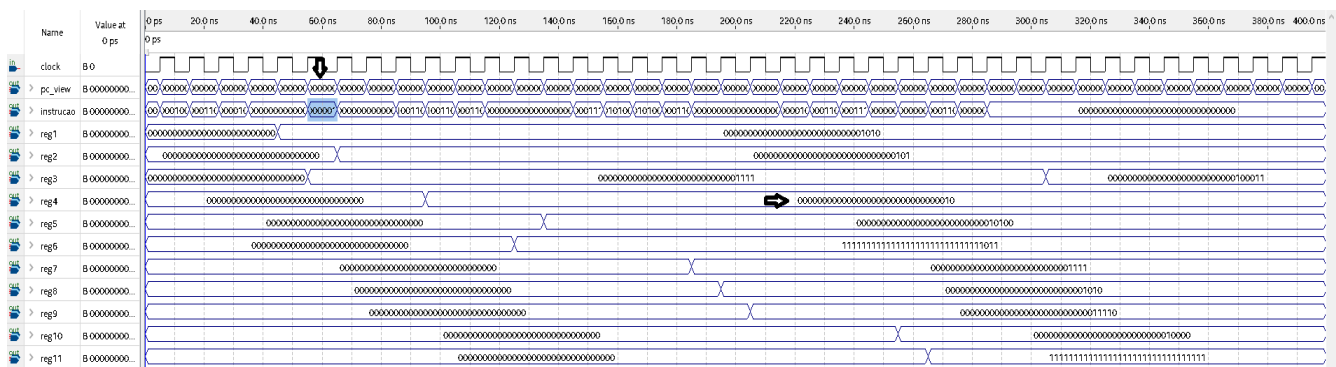
Arbitrary Value

Radix: Binary

Numeric or named value: 00010000001001000000000000001000

OK Cancel

-- SUBI \$4, \$1, 8



-- Instrução SUB

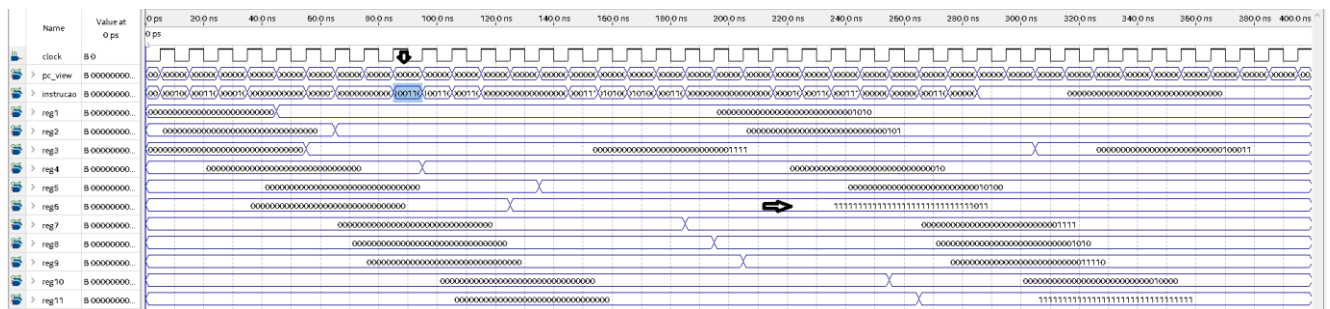
Arbitrary Value

Radix: Binary

Numeric or named value: 00001000001000110011000000000000

OK Cancel

-- SUB \$6, \$1, \$3



--Instrução ADD

Arbitrary Value

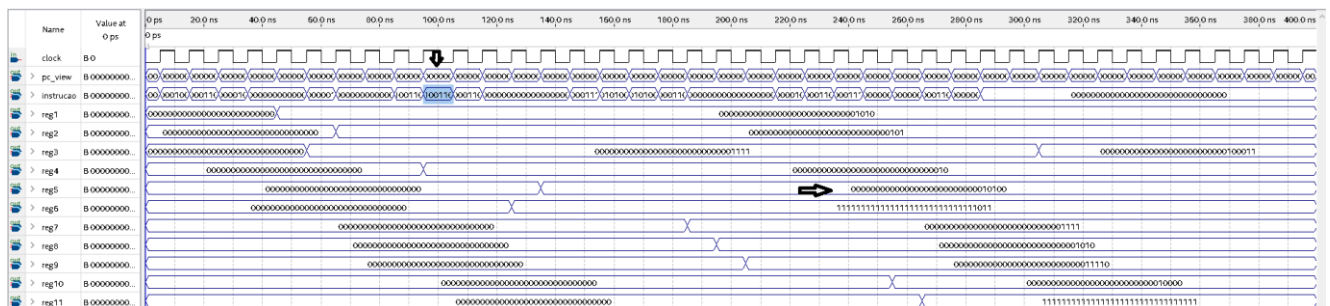
Arbitrary Value

Radix: Binary

Numeric or named value: 00000100010000110010100000000000

OK Cancel

-- ADD \$5, \$2, \$3



--Instrução SW

Arbitrary Value

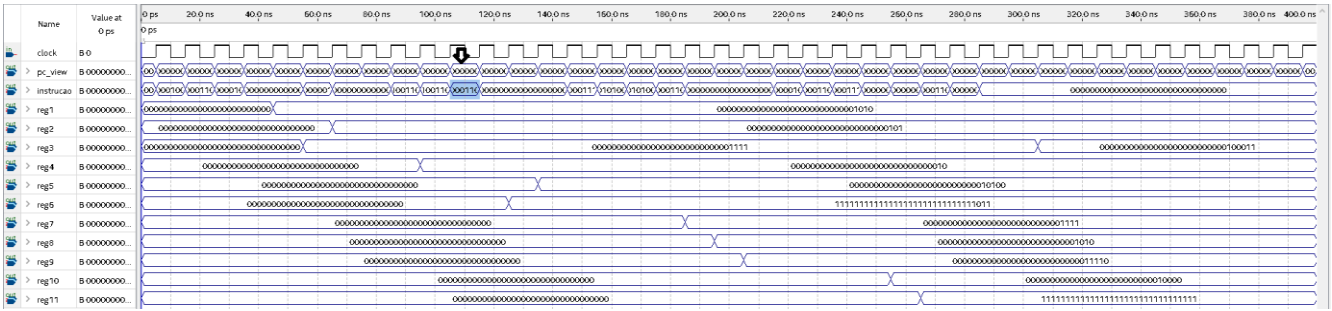
Arbitrary Value

Radix: Binary


Numeric or named value: 00011010000000110000000000000000

OK Cancel

-- SW \$3, 0(\$16)



-- Instrução LW



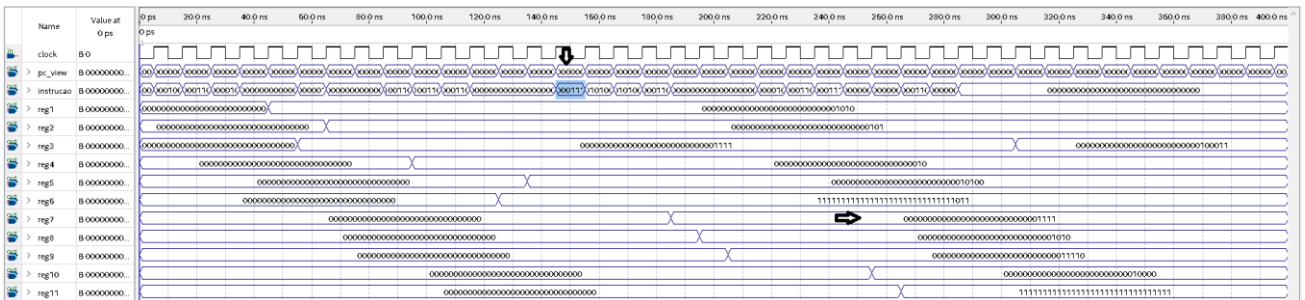
Arbitrary Value

Radix: Binary


Numeric or named value: 00010110000001110000000000000000

OK

-- LW \$7, 0(\$16)



-- Instrução AND



Arbitrary Value

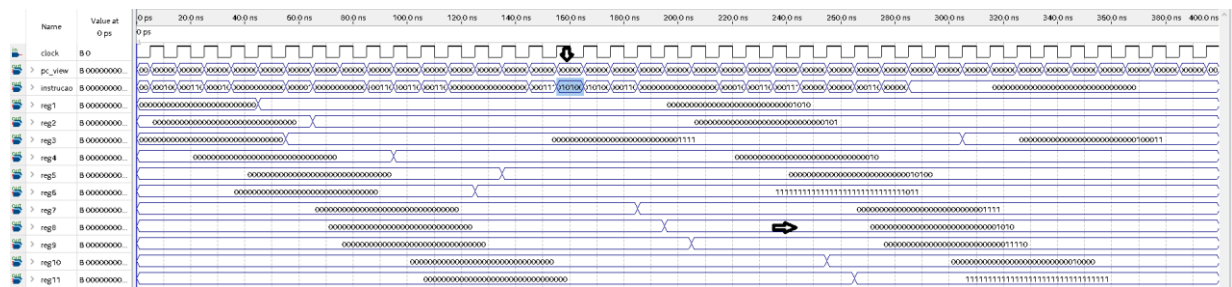
Arbitrary Value

Radix: Binary


Numeric or named value: 00011100011000010100000000000000

OK Cancel

-- AND \$8, \$3, \$1



-- Instrução OR



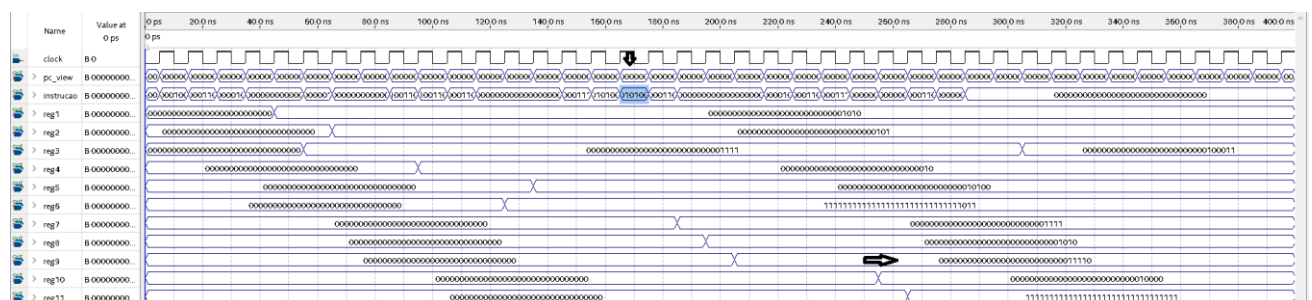
Arbitrary Value

Radix: Binary


Numeric or named value: 0010010010101000010100100000000000

OK Cancel

-- OR \$9, \$5, \$1



-- Instrução BEQ



Arbitrary Value

Radix: Binary

Numeric or named value: 00101100111000110000000000000100

OK Cancel


-- BEQ \$7,\$3 antes do salto

[illegible]

-- BEQ \$7,\$3 depois do salto

[illegible]

-- Instrução ANDi



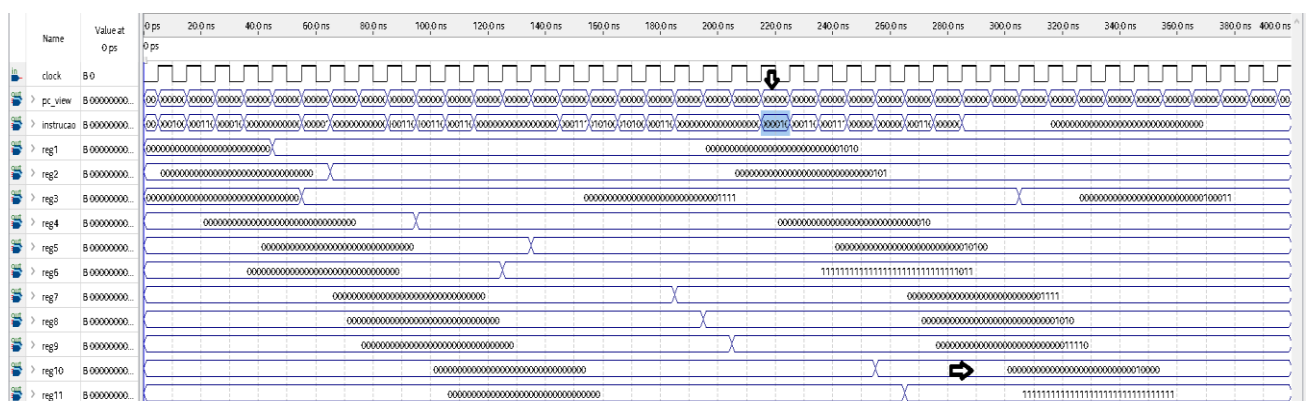
Arbitrary Value

Radix: Binary

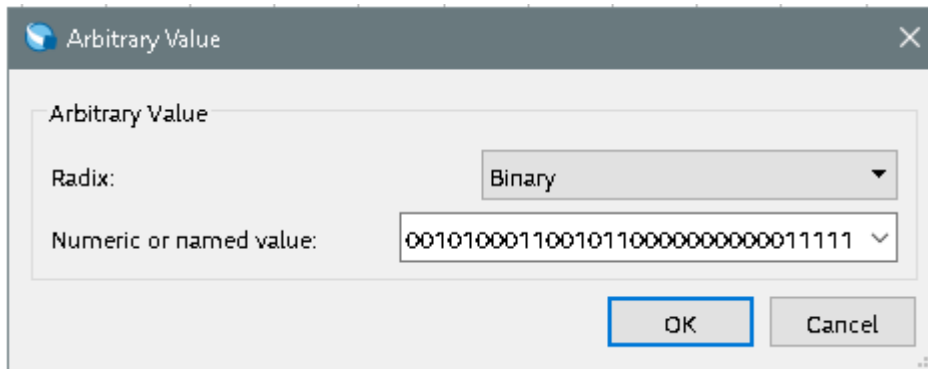
Numeric or named value: 001000001100101000000000000010000

OK

-- ANDi \$10, \$6, 16



-- Instrução ORi



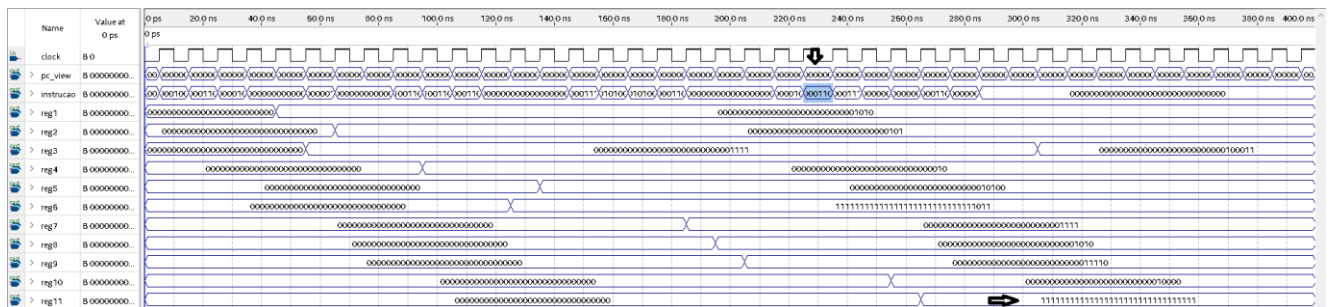
Arbitrary Value

Radix: Binary

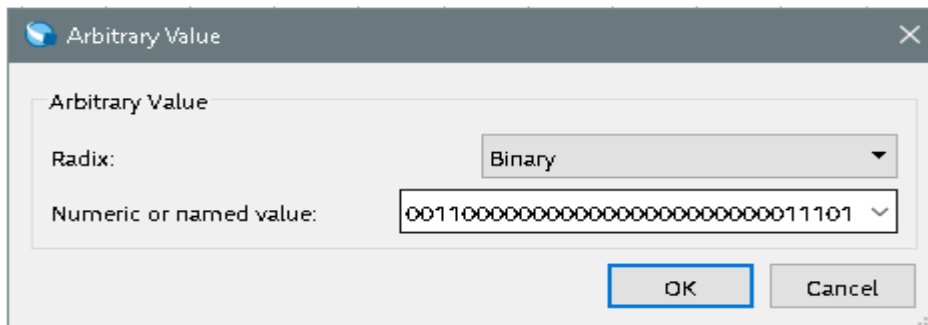
Numeric or named value: 00101000110010110000000000001111

OK Cancel

-- ORi \$11, \$6, 31



-- Instrução Jump



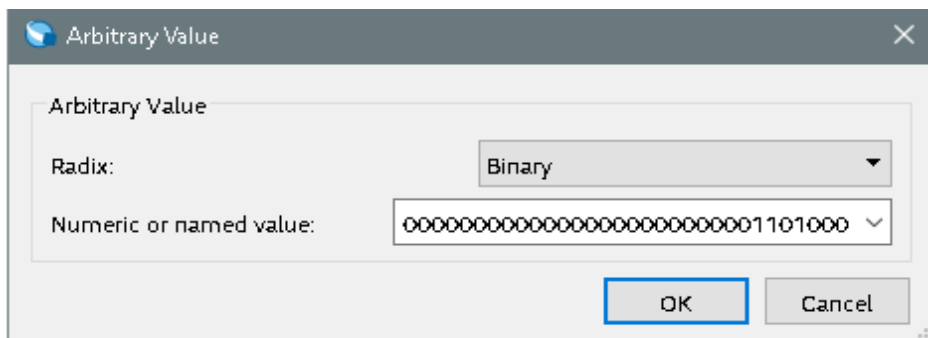
Arbitrary Value

Radix: Binary

Numeric or named value: 00110000000000000000000000001101

OK Cancel

-- Jump (116) antes do salto



Arbitrary Value

Radix: Binary

Numeric or named value: 00000000000000000000000000001101000

OK Cancel

5. CONCLUSÃO

Neste trabalho foi possível estar aprimorando os conhecimentos e manipulação com a linguagem de programação de VHDL. Podendo conhecer mais a fundo a aplicação de uma CPU e seus principais componentes, como ele funciona e como se comporta, podendo analisar as instruções e seus formatos e as suas execuções, observando o funcionamento e armazenamento nos registradores passo a passo no ciclo em pipeline com ajuda de “debug” criando no programa no waveform, observando também as saídas de cada componente e o que estará sendo salvo nos registradores que forem solicitados.

6. ANEXO: CÓDIGO VHDL

CPU:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Cpu_pipeline is
5
6  PORT ( clock:in std_logic;
7        reg1: out std_logic_vector(0 to 31);
8        reg2: out std_logic_vector(0 to 31);
9        reg3: out std_logic_vector(0 to 31);
10       reg4: out std_logic_vector(0 to 31);
11       reg5: out std_logic_vector(0 to 31);
12       reg6: out std_logic_vector(0 to 31);
13       reg7: out std_logic_vector(0 to 31);
14       reg8: out std_logic_vector(0 to 31);
15       reg9: out std_logic_vector(0 to 31);
16       reg10: out std_logic_vector(0 to 31);
17       reg11: out std_logic_vector(0 to 31);
18       instrucao: out std_logic_vector(0 to 31);
19       pc_view: out std_logic_vector(0 to 31)
20     );
21
22
23
24  end Cpu_pipeline;
25
26  architecture Behavioral of Cpu_pipeline is
27
28
29
30
31  component reg_f
32  port (regwrite: in std_logic;
33        clock: in std_logic;
34        read_register_1: in std_logic_vector(0 to 4);
35        read_register_2: in std_logic_vector(0 to 4);
36        write_register: in std_logic_vector(0 to 4);
37        write_data: in std_logic_vector(0 to 31);
38        read_data_1: out std_logic_vector(0 to 31);
39        read_data_2: out std_logic_vector(0 to 31);
40        reg1: out std_logic_vector(0 to 31);
41        reg2: out std_logic_vector(0 to 31);
42        reg3: out std_logic_vector(0 to 31);
43        reg4: out std_logic_vector(0 to 31);
44        reg5: out std_logic_vector(0 to 31);
45        reg6: out std_logic_vector(0 to 31);
46        reg7: out std_logic_vector(0 to 31);
47        reg8: out std_logic_vector(0 to 31);
```

```

48         reg9: out std_logic_vector(0 to 31);
49         reg10: out std_logic_vector(0 to 31);
50         reg11: out std_logic_vector(0 to 31);
51     );
52
53     end component;
54
55     component instruction_memory
56     port (
57         instruction: out std_logic_vector(0 to 31);
58         address: in std_logic_vector(0 to 31)
59     );
60
61     end component;
62
63     component soma32b
64     Port ( NUM1 : in STD_LOGIC_VECTOR (0 to 31);
65           out1 : out STD_LOGIC_VECTOR (0 to 31));
66
67     end component ;
68
69     component data_mem
70     port (address: in std_logic_vector(0 to 31);
71          clock: in std_logic;
72          mem_write: in std_logic;
73          write_data: in std_logic_vector(0 to 31);
74          mem_read: in std_logic;
75          read_data: out std_logic_vector(0 to 31)
76      );
77
78     end component;
79
80     component mini_mux_2
81     Port ( SEL : in STD_LOGIC;
82           A : in STD_LOGIC_VECTOR (0 to 4);
83           B : in STD_LOGIC_VECTOR (0 to 4);
84           X : out STD_LOGIC_VECTOR (0 to 4));
85
86     end component;
87
88     component mux_2
89     Port ( SEL : in STD_LOGIC;

```

```

95         A : in STD_LOGIC_VECTOR (0 to 31);
96         B : in STD_LOGIC_VECTOR (0 to 31);
97         X : out STD_LOGIC_VECTOR (0 to 31));
98
99     end component;
100
101     component reg_pc
102     port (pc: out std_logic_vector(0 to 31);
103          clock: in std_logic;
104          prox_pc: in std_logic_vector(0 to 31));
105
106     end component;
107
108     component sh1_2
109     PORT (
110         in1 : IN STD_LOGIC_VECTOR(0 TO 31) ;
111         out1 : OUT STD_LOGIC_VECTOR(0 TO 31) ) ;
112
113     end component;
114
115     component extend
116     PORT (
117         in1 : IN STD_LOGIC_VECTOR(0 TO 15) ;
118         out1 : OUT STD_LOGIC_VECTOR(0 TO 31)
119     );
120
121     END component;
122
123     component ALU
124     PORT (
125         aluop : IN STD_LOGIC_VECTOR(0 TO 1) ;
126         A, B : IN STD_LOGIC_VECTOR(0 TO 31) ;
127         Saída : OUT STD_LOGIC_VECTOR(0 TO 31);
128         zero: out std_logic
129     );
130
131     END component ;
132
133     component RegIFID
134     port (clock: in std_logic;

```

```

142         in_pc:    in    std_logic_vector(0 to 31);
143         out_pc:    out   std_logic_vector(0 to 31);
144
145         in_instr:  in    std_logic_vector(0 to 31);
146         out_instr: out   std_logic_vector(0 to 31));
147     end component;
148
149
150     component RegIDEX
151     port (clock:      in    std_logic;
152           in_WB:      in    std_logic_vector(0 to 1);
153           in_ME:      in    std_logic_vector(0 to 2);
154           out_WB:      out   std_logic_vector(0 to 1);
155           out_ME:      out   std_logic_vector(0 to 2);
156
157
158           ALUSrc_in:  in    STD_LOGIC;
159           RegDst_in:  in    STD_LOGIC;
160           ALUOp_in:   in    STD_LOGIC_VECTOR(0 TO 1);
161
162           ALUSrc_out: out   STD_LOGIC;
163           RegDst_out: out   STD_LOGIC;
164           ALUOp_out:  out   STD_LOGIC_VECTOR(0 TO 1);
165
166           in_pc:      in    std_logic_vector(0 to 31);
167           out_pc:      out   std_logic_vector(0 to 31);
168
169           in_read1:   in    std_logic_vector(0 to 31);
170           out_read1:  out   std_logic_vector(0 to 31);
171
172           in_read2:   in    std_logic_vector(0 to 31);
173           out_read2:  out   std_logic_vector(0 to 31);
174
175           in_imed:    in    std_logic_vector(0 to 31);
176           out_imed:   out   std_logic_vector(0 to 31);
177
178           in_rt:      in    std_logic_vector(0 to 4);
179           out_rt:      out   std_logic_vector(0 to 4);
180           in_rd:      in    std_logic_vector(0 to 4);
181           out_rd:      out   std_logic_vector(0 to 4));
182     end component;
183
184
185     component somador
186     PORT (
187         A: IN STD_LOGIC_VECTOR(0 TO 31);
188         B: IN STD_LOGIC_VECTOR(0 TO 31);

```

```

189         X: out STD_LOGIC_VECTOR(0 TO 31)
190     );
191     end component;
192
193
194     component RegEXMEM
195     port (clock:      in    std_logic;
196           in_WB:      in    std_logic_vector(0 to 1);
197           in_ME:      in    std_logic_vector(0 to 2);
198           out_WB:      out   std_logic_vector(0 to 1);
199           out_ME:      out   std_logic_vector(0 to 2);
200
201           in_pc:      in    std_logic_vector(0 to 31);
202           out_pc:      out   std_logic_vector(0 to 31);
203
204           in_result:  in    std_logic_vector(0 to 31);
205           out_result: out   std_logic_vector(0 to 31);
206
207           in_wrData:  in    std_logic_vector(0 to 31);
208           out_wrData: out   std_logic_vector(0 to 31);
209
210           in_regdst:  in    std_logic_vector(0 to 4);
211           out_regdst: out   std_logic_vector(0 to 4));
212     end component;
213
214
215     component RegMEMWB
216     port (clock:      in    std_logic;
217           in_WB:      in    std_logic_vector(0 to 1);
218           out_WB:      out   std_logic_vector(0 to 1);
219
220           in_rdData:  in    std_logic_vector(0 to 31);
221           out_rdData: out   std_logic_vector(0 to 31);
222
223           in_addr:    in    std_logic_vector(0 to 31);
224           out_addr:    out   std_logic_vector(0 to 31);
225
226           in_regdst:  in    std_logic_vector(0 to 4);
227           out_regdst: out   std_logic_vector(0 to 4));
228     end component;
229
230
231     component CONTROL
232     PORT(
233         PCSRC: OUT STD_LOGIC;
234         ALUSrc: OUT STD_LOGIC;
235

```

```

236     RegDst: OUT STD_LOGIC;
237     JM: OUT STD_LOGIC;
238
239     ALUOp: OUT STD_LOGIC_VECTOR(0 TO 1);
240     WB: OUT STD_LOGIC_VECTOR(0 TO 1);
241     MEM: OUT STD_LOGIC_VECTOR(0 TO 2);
242
243     OPCODE: in STD_LOGIC_vector(0 to 5)
244 );
245 END component;
246
247 component shift_jump
248
249     PORT (
250         in1 : IN STD_LOGIC_VECTOR(0 TO 25) ;--operandos
251         out1 : OUT STD_LOGIC_VECTOR(0 TO 27) ) ;--saida
252
253     end component;
254
255
256
257
258 signal pc_instr: std_logic_vector(0 to 31);
259 signal instructionIF: std_logic_vector(0 to 31);
260
261 signal pcsrc_mux_0: std_logic_vector(0 to 31);
262 signal pcsrc_mux_1: std_logic_vector(0 to 31);
263
264 signal prox_pc: std_logic_vector(0 to 31);
265
266 signal branch: std_logic;
267
268 signal pcselect_mux_0: std_logic_vector(0 to 31);
269
270 signal pc4_ID: std_logic_vector(0 to 31);
271
272 signal Instruction: std_logic_vector(0 to 31);
273
274 signal OPCODE: std_logic_vector(0 to 5);
275 signal Read_Register_1: std_logic_vector(0 to 4);
276 signal Read_Register_2: std_logic_vector(0 to 4);
277 signal Imediato: std_logic_vector(0 to 15);
278 signal Rt_ID: std_logic_vector(0 to 4);
279 signal Rd_ID: std_logic_vector(0 to 4);
280
281 signal RegWrite: std_logic;
282 signal Write_Register: std_logic_vector(0 to 4);

```

```

283 signal Write_Data: std_logic_vector(0 to 31);
284 signal Read_Data_1: std_logic_vector(0 to 31);
285 signal Read_Data_2: std_logic_vector(0 to 31);
286
287 signal Imediato_extendido_ID: std_logic_vector(0 to 31);
288
289 signal JumpMux: std_logic;
290 signal Jump_concat: std_logic_vector(0 to 31);
291
292 signal pc_select_mux_1: std_logic_vector(0 to 31);
293
294 signal PCSrc: std_logic;
295 signal instruction_shift_jump: std_logic_vector(0 to 31);
296
297 signal aluSrc_ID: std_logic;
298 signal regdst_ID: std_logic;
299 signal aluOP_ID: std_logic_vector(0 to 1);
300
301 signal WB_ID: std_logic_vector(0 to 1);
302 signal MEM_ID: std_logic_vector(0 to 2);
303 signal WB_EX: std_logic_vector(0 to 1);
304 signal MEM_EX: std_logic_vector(0 to 2);
305 signal WB_MEM: std_logic_vector(0 to 1);
306 signal MEM_MEM: std_logic_vector(0 to 2);
307 signal WB_WB: std_logic_vector(0 to 1);
308
309 signal aluSrc_ex: std_logic;
310 signal regdst_ex: std_logic;
311 signal aluOP_ex: std_logic_vector(0 to 1);
312 signal pc4_ex: std_logic_vector(0 to 31);
313 signal ULA_Src_A: std_logic_vector(0 to 31);
314 signal alusrc_mux_0: std_logic_vector(0 to 31);
315 signal alusrc_mux_1: std_logic_vector(0 to 31);
316 signal Imediato_extendido_ex: std_logic_vector(0 to 31);
317 signal regdst_mux_0: std_logic_vector(0 to 4);
318 signal regdst_mux_1: std_logic_vector(0 to 4);
319
320 signal imed_extendido_shiftado: std_logic_vector(0 to 31);
321 signal Branch_addr: std_logic_vector(0 to 31);
322
323 signal ULA_Zero: std_logic;
324 signal ULA_Src_B: std_logic_vector(0 to 31);
325 signal Resultado_ula: std_logic_vector(0 to 31);
326 signal regdst_mux_out: std_logic_vector(0 to 4);
327 signal endereco_mem: std_logic_vector(0 to 31);
328 signal write_data_memoria: std_logic_vector(0 to 31);
329 signal regdst_memoria: std_logic_vector(0 to 4);

```

```

330     signal Read_data_memoria: std_logic_vector(0 to 31);
331     signal memwrite: std_logic;
332     signal memread: std_logic;
333     signal branch_control: std_logic;
334     signal memtoreg_0: std_logic_vector(0 to 31);
335     signal memtoreg_1: std_logic_vector(0 to 31);
336     signal memtoreg: std_logic;
337
338     signal imediato_shift: std_logic_vector(0 to 27);
339
340
341
342 begin
343
344     IM: instruction_memory port map (instructionIF,pc_instr);
345     pc_4: soma32b port map (pc_instr, pcsrc_mux_0);
346
347     pc: reg_pc port map (pc_instr,clock, prox_pc);
348
349     mux_branch: mux_2 port map (branch,pcsrc_mux_1,pcsrc_mux_0,pcselect_mux_0);
350
351     ifid: RegIFID port map(clock,pcsrc_mux_0,pc4_ID,instructionIF,instruction);
352
353
354     OPCode      <= Instruction( 0 to 5);
355     Read_Register_1 <= Instruction( 6 to 10);
356     Read_Register_2 <= Instruction(11 to 15);
357     Imediato      <= Instruction(16 to 31);
358     Rt_ID         <= Instruction(11 to 15);
359     Rd_ID         <= Instruction(16 to 20);
360
361     banco_reg: reg_f port map (RegWrite,clock,Read_Register_1,Read_Register_2,
362     Write_Register,Write_Data,Read_Data_1,Read_Data_2,
363     reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11);
364
365     sign_extend: extend port map (Imediato, Imediato_extendido_ID);
366     jumptype_mux: mux_2 port map (JumpMux,Read_Data_1,Jump_concat,pc_select_mux_1);
367
368     pcsrc_mux: mux_2 port map (PCSrc,pc_select_mux_1,pcselect_mux_0,prox_pc);
369
370     shift: shift_jump port map(instruction(6 to 31),imediato_shift);
371
372
373
374     Jump_concat <= pc4_ID(0 to 3) & imediato_shift;
375
376     Ctrl: Control port map (PCSrc,ALUSrc_ID,regdst_ID,JumpMux,aluOP_ID,

```

```

377     WB_ID, MEM_ID, OPCode);
378
379
380     idex: RegIDEX port map (clock,WB_ID, MEM_ID, WB_EX, MEM_EX, ALUSRC_ID, regdst_ID,
381     aluop_ID, alusrc_ex, regdst_ex, aluop_ex, pc4_ID,
382     pc4_ex, Read_Data_1, ULA_Src_A, Read_Data_2, alusrc_mux_0, Imediato_extendido_ID,
383     Imediato_extendido_ex, rt_id, regdst_mux_0,
384     Rd_ID, regdst_mux_1);
385
386
387     calcula_branch: somador port map (pc4_EX, imed_extendido_shiftado, Branch_addr);
388
389     ula_principal: ALU port map (aluop_ex, ULA_Src_A, ULA_Src_B, Resultado_ULA, ULA_Zero);
390     alusrc_mux: mux_2 port map (Alusrc_ex, alusrc_mux_1, alusrc_mux_0, ULA_Src_B);
391     regdst_mux: mini_mux_2 port map (regdst_ex, regdst_mux_1, regdst_mux_0, regdst_mux_out);
392     shift_exec: shl_2 port map (Imediato_extendido_EX, imed_extendido_shiftado);
393
394
395
396     alusrc_mux_1 <= Imediato_extendido_EX;
397
398
399     exme: RegEXMEM port map (clock, WB_EX, MEM_EX, WB_MEM, MEM_MEM, Branch_addr, pcsrc_mux_1
400     , Resultado_ULA, endereco_mem,
401     alusrc_mux_0, write_data_memoria, regdst_mux_out, regdst_memoria);
402
403
404     memoria_dados: data_mem port map (endereco_mem, clock, memWrite, write_data_memoria,
405     memRead, Read_data_memoria);
406
407
408     memWrite <= MEM_MEM(0);
409     memRead <= MEM_MEM(1);
410     branch_control <= MEM_MEM(2);
411     branch <= branch_control and ula_zero;
412
413
414
415     mewb: RegMEMWB port map (clock, WB_MEM, WB_WB, Read_data_memoria, memtoreg_1,
416     endereco_mem, memtoreg_0, regdst_memoria,
417     Write_Register);
418
419
420     memtoreg_mux: mux_2 port map (Memtoreg, memtoreg_1, memtoreg_0, Write_Data);
421
422     MemToReg <= WB_WB(0);
423     RegWrite <= WB_WB(1);

```

```

424     instracao <= instruction;
425     pc_view <= pc_instr;
426
427
428
429 end Behavioral;

```


MUX_2:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity mux_2 is
5  port ( SEL : in  STD_LOGIC;
6        A   : in  STD_LOGIC_VECTOR (0 to 31);
7        B   : in  STD_LOGIC_VECTOR (0 to 31);
8        X   : out STD_LOGIC_VECTOR (0 to 31));
9  end mux_2;
10
11 architecture Behavioral of mux_2 is
12 begin
13     X <= A when (SEL = '1') else B;
14 end Behavioral;
```

REG_F:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity reg_f is
7  port ( regwrite: in std_logic;
8        clock: in std_logic;
9        read_register_1: in std_logic_vector(0 to 4);
10       read_register_2: in std_logic_vector(0 to 4);
11       write_register: in std_logic_vector(0 to 4);
12       write_data: in std_logic_vector(0 to 31);
13       read_data_1: out std_logic_vector(0 to 31);
14       read_data_2: out std_logic_vector(0 to 31);
15       reg1: out std_logic_vector(0 to 31);
16       reg2: out std_logic_vector(0 to 31);
17       reg3: out std_logic_vector(0 to 31);
18       reg4: out std_logic_vector(0 to 31);
19       reg5: out std_logic_vector(0 to 31);
20       reg6: out std_logic_vector(0 to 31);
21       reg7: out std_logic_vector(0 to 31);
22       reg8: out std_logic_vector(0 to 31);
23       reg9: out std_logic_vector(0 to 31);
24       reg10: out std_logic_vector(0 to 31);
25       reg11: out std_logic_vector(0 to 31)
26       );
27
28 end reg_f;
29
30 architecture regs of reg_f is
31     type register_type is array(0 to 31) of std_logic_vector(0 to 31);
32     signal registers: register_type;
33 begin
34
35     process(clock)
36     begin
37         if (clock'EVENT and clock = '1' and regwrite = '1' and not (write_register = "00000") ) then
38             registers(to_integer(unsigned(write_register))) <= write_data;
39         end if;
40     end process;
41
42     read_data_1 <= registers(to_integer(unsigned(read_register_1)));
43     read_data_2 <= registers(to_integer(unsigned(read_register_2)));
44     reg1 <= registers(1);
45     reg2 <= registers(2);
46     reg3 <= registers(3);
```

SOMADOR:

```
1  -- Declaração das bibliotecas utilizadas
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_signed.all;
5  USE ieee.numeric_std.all;
6
7  -- Entity ADDER_32_BITS
8  ENTITY somador IS
9
10     -- Declaração de todas as portas utilizadas
11     PORT (
12         A: IN STD_LOGIC_VECTOR(0 TO 31);
13         B: IN STD_LOGIC_VECTOR(0 TO 31);
14         X: out STD_LOGIC_VECTOR(0 TO 31)
15     );
16
17     -- Finalizando Entity
18     END somador;
19
20     -- Architecture ADDER_32_BITS
21     ARCHITECTURE ADDER OF somador IS
22     BEGIN
23
24         -- Saida recebe a soma das duas entradas
25         X <= A+B;
26
27         -- Finalizando Architecture
28     END;
```

CONTROL:

```
1  -- Declaração de todas as bibliotecas utilizadas
2  LIBRARY ieee;
3  USE ieee.STD_LOGIC_1164.all;
4  USE ieee.STD_LOGIC_unsigned.all;
5  USE ieee.numeric_std.all;
6
7  -- Declaração de ENTITY Control, mesmo nome do arquivo
8  ENTITY CONTROL IS
9  PORT( -- Declaração de todas as "portas" utilizadas
10      PCSRC: OUT STD_LOGIC := '0';
11      ALUSrc: OUT STD_LOGIC := '0';
12      RegDst: OUT STD_LOGIC := '0';
13      JM: OUT STD_LOGIC := '0';
14
15      ALUOp: OUT STD_LOGIC_VECTOR(0 TO 1) := "00";
16      WB: OUT STD_LOGIC_VECTOR(0 TO 1) := "00";
17      MEM: OUT STD_LOGIC_VECTOR(0 TO 2) := "000";
18
19      OPCODE: in STD_LOGIC_vector(0 to 5)
20  );
21  END CONTROL;
22
23  ARCHITECTURE CONTROL_UNIT OF CONTROL IS
24  BEGIN
25
26      process(OPCODE)
27      BEGIN
28
29          CASE OPCODE is -- Caso o OPCODE for igual a...
30
31              -- Enquanto for igual a 000000, realiza instrução NOP
32              WHEN "000000" =>
33                  PCSRC <= '0';
34                  ALUSrc <= 'X';
35                  ALUOp <= "XX";
36                  RegDst <= 'X';
37                  MEM <= "000";
38                  WB <= "00";
39
40              -- Enquanto for igual a 000001, realiza instrução ADD
41              WHEN "000001" =>
42                  PCSRC <= '0';
43                  ALUSrc <= '0';
44                  ALUOp <= "00";
45                  RegDst <= '1';
46                  MEM <= "000";
47                  WB <= "01";
```

```

48
49 -- Enquanto for igual a 000010, realiza instrução SUB
50 WHEN "000010" =>
51     PCSRC      <= '0';
52     ALUSrc     <= '0';
53     ALUOp      <= "01";
54     RegDst     <= '1';
55     MEM        <= "000";
56     WB         <= "01";
57
58 -- Enquanto for igual a 000011, realiza instrução ADDi
59 WHEN "000011" =>
60     PCSRC      <= '0';
61     ALUSrc     <= '1';
62     ALUOp      <= "00";
63     RegDst     <= '0';
64     MEM        <= "000";
65     WB         <= "01";
66
67 -- Enquanto for igual a 000100, realiza instrução SUBi
68 WHEN "000100" =>
69     PCSRC      <= '0';
70     ALUSrc     <= '1';
71     ALUOp      <= "01";
72     RegDst     <= '0';
73     MEM        <= "000";
74     WB         <= "01";
75
76 -- Enquanto for igual a 000101, realiza instrução LW
77 WHEN "000101" => -- LW
78     PCSRC      <= '0';
79     ALUSrc     <= '1';
80     ALUOp      <= "00";
81     RegDst     <= '0';
82     MEM        <= "010";
83     WB         <= "11";
84
85 -- Enquanto for igual a 000110, realiza instrução SW
86 WHEN "000110" => -- SW
87     PCSRC      <= '0';
88     ALUSrc     <= '1';
89     ALUOp      <= "00";
90     RegDst     <= 'X';
91     MEM        <= "100";
92     WB         <= "00";
93
94 -- Enquanto for igual a 000111, realiza instrução AND

```

```

95 WHEN "000111" => -- AND
96     PCSRC      <= '0';
97     ALUSrc     <= '0';
98     ALUOp      <= "10";
99     RegDst     <= '1';
100    MEM        <= "000";
101    WB         <= "01";
102
103 -- Enquanto for igual a 001000, realiza instrução ANDi
104 WHEN "001000" => -- ANDi
105     PCSRC      <= '0';
106     ALUSrc     <= '1';
107     ALUOp      <= "10";
108     RegDst     <= '0';
109     MEM        <= "000";
110     WB         <= "01";
111
112 -- Enquanto for igual a 001001, realiza instrução OR
113 WHEN "001001" => -- OR
114     PCSRC      <= '0';
115     ALUSrc     <= '0';
116     ALUOp      <= "11";
117     RegDst     <= '1';
118     MEM        <= "000";
119     WB         <= "01";
120
121 -- Enquanto for igual a 001010, realiza instrução ORi
122 WHEN "001010" => -- ORi
123     PCSRC      <= '0';
124     ALUSrc     <= '1';
125     ALUOp      <= "11";
126     RegDst     <= '0';
127     MEM        <= "000";
128     WB         <= "01";
129
130 -- Enquanto for igual a 001011, realiza instrução BEQ
131 WHEN "001011" => -- BEQ
132     PCSRC      <= '0';
133     ALUSrc     <= '0';
134     ALUOp      <= "01";
135     RegDst     <= 'X';
136     MEM        <= "001";
137     WB         <= "00";
138
139 -- Enquanto for igual a 001100, realiza instrução J
140 WHEN "001100" => -- J
141     PCSRC      <= '1';

```

```

142         JM          <= '0';
143         ALUSrc       <= 'X';
144         ALUOp        <= "XX";
145         RegDst       <= 'X';
146         MEM          <= "000";
147         WB           <= "00";
148
149         -- Enquanto for igual a 001101, realiza instrução JR
150         WHEN "001101" => -- JR
151             PCSRC     <= '1';
152             JM        <= '1';
153             ALUSrc    <= 'X';
154             ALUOp     <= "XX";
155             RegDst    <= 'X';
156             MEM       <= "000";
157             WB        <= "00";
158
159         -- Enquanto não for nenhum desses casos, não realiza nenhuma instrução
160         WHEN others =>
161             PCSRC     <= '0';
162             ALUSrc    <= 'X';
163             ALUOp     <= "XX";
164             RegDst    <= 'X';
165             MEM       <= "000";
166             WB        <= "00";
167
168         -- Finalizando o Case
169         END CASE;
170
171     -- Finalizando o Processo
172     END PROCESS;
173
174 -- Finalizando a ARCHITECTURE
175 END;

```

RegEXMEM:

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity RegEXMEM is
7  port (clock:      in    std_logic;
8        in_WB:      in    std_logic_vector(0 to 1);
9        in_ME:      in    std_logic_vector(0 to 2);
10       out_WB:      out   std_logic_vector(0 to 1) := "00";
11       out_ME:      out   std_logic_vector(0 to 2) := "000";
12
13       in_pc:       in    std_logic_vector(0 to 31);
14       out_pc:      out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
15
16       in_result:   in    std_logic_vector(0 to 31);
17       out_result:  out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
18
19       in_wrData:   in    std_logic_vector(0 to 31);
20       out_wrData:  out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
21
22       in_regdst:   in    std_logic_vector(0 to 4);
23       out_regdst:  out   std_logic_vector(0 to 4) := "00000");
24  end RegEXMEM;
25
26  architecture exe of RegEXMEM is
27  begin
28      process(clock)
29      begin
30          if (clock'EVENT and clock = '1') then
31              out_WB      <= in_WB;
32              out_ME      <= in_ME;
33              out_pc      <= in_pc;
34              out_result   <= in_result;
35              out_wrData   <= in_wrData;
36              out_regdst   <= in_regdst;
37          end if;
38      end process;
39  end;
40

```

RegIDEX:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity RegIDEX is
7  port (clock:      in    std_logic;
8        in_WB:      in    std_logic_vector(0 to 1);
9        in_ME:      in    std_logic_vector(0 to 2);
10       out_WB:      out   std_logic_vector(0 to 1) := "00";
11       out_ME:      out   std_logic_vector(0 to 2) := "000";
12
13
14       ALUSrc_in: in STD_LOGIC;
15       RegDst_in: in STD_LOGIC;
16       ALUOp_in: in STD_LOGIC_VECTOR(0 To 1);
17
18       ALUSrc_out: out STD_LOGIC := '0';
19       RegDst_out: out STD_LOGIC := '0';
20       ALUOp_out: out STD_LOGIC_VECTOR(0 To 1) := "00";
21
22
23       in_pc:      in    std_logic_vector(0 to 31);
24       out_pc:      out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
25
26       in_read1:   in    std_logic_vector(0 to 31);
27       out_read1:  out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
28
29       in_read2:   in    std_logic_vector(0 to 31);
30       out_read2:  out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
31
32       in_imed:    in    std_logic_vector(0 to 31);
33       out_imed:   out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
34
35       in_rt:      in    std_logic_vector(0 to 4);
36       out_rt:     out   std_logic_vector(0 to 4) := "00000";
37       in_rd:      in    std_logic_vector(0 to 4);
38       out_rd:     out   std_logic_vector(0 to 4) := "00000";
39   end RegIDEX;
40
41   architecture exe of RegIDEX is
42   begin
43   process(clock)
44   begin
45       if (clock'EVENT and clock = '1') then
46           out_WB <= in_WB;
47
48           out_ME <= in_ME;
49
50           ALUSrc_out <= ALUSrc_in;
51           RegDst_out <= RegDst_in;
52           ALUOp_out <= ALUOp_in;
53
54           out_pc <= in_pc;
55           out_read1 <= in_read1;
56           out_read2 <= in_read2;
57           out_imed <= in_imed;
58           out_rt <= in_rt;
59           out_rd <= in_rd;
60       end if;
61   end process;
62   end;
```

RegIFID:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity RegIFID is
7  port (Clock:      in    std_logic;
8
9        in_pc:      in    std_logic_vector(0 to 31);
10       out_pc:      out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
11
12       in_instr:    in    std_logic_vector(0 to 31);
13       out_instr:   out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
14   end RegIFID;
15
16   architecture exe of RegIFID is
17   begin
18   process(clock)
19   begin
20       if (clock'EVENT and clock = '1') then
21           out_instr <= in_instr;
22           out_pc <= in_pc;
23       end if;
24   end process;
25   end;
```

INSTRUCTION MEMORY:

```
1  -- Declaração de todas as bibliotecas utilizadas
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all;
5  use ieee.numeric_std.all;
6
7  -- Entity instruction_memory
8  entity instruction_memory is
9
10     -- Declaração das portas utilizadas no instruction_memory
11     port (
12         instruction: out std_logic_vector(0 to 31) := "00000000000000000000000000000000";
13         address: in std_logic_vector(0 to 31)
14     );
15
16     --Finalizando a entity instruction_memory
17 end instruction_memory;
18
19 -- Architecture instruction
20 architecture instruction of instruction_memory is
21
22     --Declaração do tipo e sinal
23     type memory_type is array(0 to 200) of std_logic_vector(0 to 7);
24     signal memory: memory_type;
25
26 begin
27
28     -- ADDi $1, $3, 10
29     memory(0) <= "00001100";
30     memory(1) <= "01100001";
31     memory(2) <= "00000000";
32     memory(3) <= "00001010";
33
34     -- ADDi $3, $2, 15
35     memory(4) <= "00001100";
36     memory(5) <= "01000011";
37     memory(6) <= "00000000";
38     memory(7) <= "00001111";
39
40     -- ADDi $2, $4, 5
41     memory(8) <= "00001100";
42     memory(9) <= "10000010";
43     memory(10) <= "00000000";
44     memory(11) <= "00000101";
```

```
48
49
50     -- NOP
51     memory(12) <= "00000000";
52     memory(13) <= "00000000";
53     memory(14) <= "00000000";
54     memory(15) <= "00000000";
55
56     -- NOP
57     memory(16) <= "00000000";
58     memory(17) <= "00000000";
59     memory(18) <= "00000000";
60     memory(19) <= "00000000";
61
62     -- subi $4,$1,8
63     memory(20) <= "00010000";
64     memory(21) <= "00100100";
65     memory(22) <= "00000000";
66     memory(23) <= "00001000";
67
68     -- NOP
69     memory(24) <= "00000000";
70     memory(25) <= "00000000";
71     memory(26) <= "00000000";
72     memory(27) <= "00000000";
73
74     -- NOP
75     memory(28) <= "00000000";
76     memory(29) <= "00000000";
77     memory(30) <= "00000000";
78     memory(31) <= "00000000";
79
80     -- SUB $6, $1, $3
81     memory(32) <= "00001000";
82     memory(33) <= "00100011";
83     memory(34) <= "00110000";
84     memory(35) <= "00000000";
85
86     -- add $5,$2, $3
87     memory(36) <= "00000100";
88     memory(37) <= "01000011";
89     memory(38) <= "00101000";
```

```

95     memory(39) <= "00000000";
96
97
98     -- sw $3,0($16)
99     memory(40) <= "00011010";
100    memory(41) <= "00000011";
101    memory(42) <= "00000000";
102    memory(43) <= "00000000";
103
104
105     -- NOP
106    memory(44) <= "00000000";
107    memory(45) <= "00000000";
108    memory(46) <= "00000000";
109    memory(47) <= "00000000";
110
111
112     -- NOP
113    memory(48) <= "00000000";
114    memory(49) <= "00000000";
115    memory(50) <= "00000000";
116    memory(51) <= "00000000";
117
118     -- NOP
119    memory(52) <= "00000000";
120    memory(53) <= "00000000";
121    memory(54) <= "00000000";
122    memory(55) <= "00000000";
123
124     -- lw $7,0($16)
125    memory(56) <= "00010110";
126    memory(57) <= "00000111";
127    memory(58) <= "00000000";
128    memory(59) <= "00000000";
129
130     -- and $8,$3,$1
131    memory(60) <= "00011100";
132    memory(61) <= "01100001";
133    memory(62) <= "01000000";
134    memory(63) <= "00000000";
135
136     -- or $9,$5,$1
137    memory(64) <= "00100100";
138    memory(65) <= "10100001";
139    memory(66) <= "01001000";
140    memory(67) <= "00000000";
141

```

```

142     -- beq $7,$3
143    memory(68) <= "00101100";
144    memory(69) <= "11100011";
145    memory(70) <= "00000000";
146    memory(71) <= "00000100";
147
148     -- NOP
149    memory(72) <= "00000000";
150    memory(73) <= "00000000";
151    memory(74) <= "00000000";
152    memory(75) <= "00000000";
153
154     -- NOP
155    memory(76) <= "00000000";
156    memory(77) <= "00000000";
157    memory(78) <= "00000000";
158    memory(79) <= "00000000";
159
160     -- NOP
161    memory(80) <= "00000000";
162    memory(81) <= "00000000";
163    memory(82) <= "00000000";
164    memory(83) <= "00000000";
165
166     -- ADDi $1, $3, 10
167    memory(84) <= "00001100";
168    memory(85) <= "01100001";
169    memory(86) <= "00000000";
170    memory(87) <= "00001010";
171
172     -- andi $10,$6,16
173    memory(88) <= "00100000";
174    memory(89) <= "11001010";
175    memory(90) <= "00000000";
176    memory(91) <= "00010000";
177
178     -- ori $11,$6,31
179    memory(92) <= "00101000";
180    memory(93) <= "11001011";
181    memory(94) <= "00000000";
182    memory(95) <= "00011111";
183
184     -- addi $15,$16,136
185    memory(96) <= "00001110";
186    memory(97) <= "00001111";
187    memory(98) <= "00000000";
188

```



```

189     memory(99) <= "10001000";
190
191
192
193     --j 116
194     memory(100) <= "00110000";
195     memory(101) <= "00000000";
196     memory(102) <= "00000000";
197     memory(103) <= "00011101";
198
199
200     -- NOP
201     memory(104) <= "00000000";
202     memory(105) <= "00000000";
203     memory(106) <= "00000000";
204     memory(107) <= "00000000";
205
206     -- NOP
207     memory(108) <= "00000000";
208     memory(109) <= "00000000";
209     memory(110) <= "00000000";
210     memory(111) <= "00000000";
211
212     -- ADDi $1, $3, 11
213     memory(112) <= "00001100";
214     memory(113) <= "01100001";
215     memory(114) <= "00000000";
216     memory(115) <= "00001011";
217
218     -- addi $3,$3,20
219     memory(116) <= "00001100";
220     memory(117) <= "01100011";
221     memory(118) <= "00000000";
222     memory(119) <= "00010100";
223
224     -- jr 15
225     memory(120) <= "00110101";
226     memory(121) <= "11100000";
227     memory(122) <= "00000000";
228     memory(123) <= "00000000";
229
230     -- NOP
231     memory(124) <= "00000000";
232     memory(125) <= "00000000";
233     memory(126) <= "00000000";
234     memory(127) <= "00000000";
235

```

```

236     -- NOP
237     memory(128) <= "00000000";
238     memory(129) <= "00000000";
239     memory(130) <= "00000000";
240     memory(131) <= "00000000";
241
242     -- ADDi $1, $3, 15
243     memory(132) <= "00001100";
244     memory(133) <= "01100001";
245     memory(134) <= "00000000";
246     memory(135) <= "00001111";
247
248     -- NOP
249     memory(136) <= "00000000";
250     memory(137) <= "00000000";
251     memory(138) <= "00000000";
252     memory(139) <= "00000000";
253
254
255     -- Inicializando processo com endereço
256     process (address)
257     begin
258         instruction <= memory(to_integer(unsigned(address))) & memory(to_integer(unsigned(address)) + 1)
259         & memory(to_integer(unsigned(address)) + 2) & memory(to_integer(unsigned(address)) + 3);
260
261         -- Finalizando processo com endereço
262     end process;
263 end;
264

```


DATA_MEM:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 use ieee.numeric_std.all;
5
6 entity data_mem is
7     port (address: in std_logic_vector(0 to 31);
8           clock:   in std_logic;
9           mem_write: in std_logic;
10          write_data: in std_logic_vector(0 to 31);
11          mem_read:  in std_logic;
12          read_data: out std_logic_vector(0 to 31)
13        );
14 end data_mem;
15
16 architecture imem of data_mem is
17     type mem_type is array(0 to 100) of std_logic_vector(0 to 7);
18     signal memory: mem_type;
19 begin
20
21     process(clock)
22     begin
23         if (clock'EVENT and clock = '1') then
24             if (mem_write = '1') then
25                 memory(to_integer(unsigned(address))) <= write_data(0 to 7);
26                 memory(to_integer(unsigned(address)) + 1) <= write_data(8 to 15);
27                 memory(to_integer(unsigned(address)) + 2) <= write_data(16 to 23);
28                 memory(to_integer(unsigned(address)) + 3) <= write_data(24 to 31);
29             end if;
30             if (mem_read = '1') then
31                 read_data <=
32                     memory(to_integer(unsigned(address))) &
33                     memory(to_integer(unsigned(address)) + 1) &
34                     memory(to_integer(unsigned(address)) + 2) &
35                     memory(to_integer(unsigned(address)) + 3);
36             else
37                 read_data <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
38             end if;
39         end if;
40     end process;
41 end;

```

REGMEMWB:

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6
7  entity RegMEMWB is
8  port (
9      clock:    in    std_logic;
10     in_WB:     in    std_logic_vector(0 to 1);
11     out_WB:    out   std_logic_vector(0 to 1) := "00";
12
13     in_rdData: in    std_logic_vector(0 to 31);
14     out_rdData: out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
15
16     in_addr:   in    std_logic_vector(0 to 31);
17     out_addr:  out   std_logic_vector(0 to 31) := "00000000000000000000000000000000";
18
19     in_regdst: in    std_logic_vector(0 to 4);
20     out_regdst: out   std_logic_vector(0 to 4) := "00000";
21 end RegMEMWB;
22
23 architecture exe of RegMEMWB is
24 begin
25     process(clock)
26     begin
27         if (clock'EVENT and clock = '1') then
28             out_WB      <= in_WB;
29             out_rdData   <= in_rdData;
30             out_addr     <= in_addr;
31             out_regdst   <= in_regdst;
32         end if;
33     end process;
34 end;

```

REGPC:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  -- Entidade reg_pc, registrador PC de 32 Bits
7  entity reg_pc is
8
9      -- Declaração da saída do registrador PC com 32 Bits
10     port (pc: out std_logic_vector(0 to 31) := "00000000000000000000000000000000");
11
12     -- Declaração da entrada Clock, verificando nível de subida ou descida
13     clock: in std_logic;
14
15     -- Declaração da entrada do próximo registrador PC com 32 Bits
16     prox_pc: in std_logic_vector(0 to 31));
17
18 end reg_pc; -- Finalizando a entidade reg_pc
19
20 architecture count of reg_pc is -- Architecture count
21 begin
22     process (clock, prox_pc) -- Inicializando processo com clock e próximo registrador pc
23     begin
24
25         if (clock'event and clock = '1') then -- Se ocorrer um evento de clock na subida
26
27             pc <= prox_pc; -- Registrador PC, recebe a instrução armazenada no registrador PC
28
29         end if; -- Finalizando if
30
31     end process; -- Finalizando processo
32
33 end; -- Finalizando
```

SOMA32B:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.std_logic_signed.all;
5
6
7
8  entity soma32b is
9      Port ( NUM1 : in STD_LOGIC_VECTOR (0 to 31);
10            out1 : out STD_LOGIC_VECTOR (0 to 31));
11 end soma32b ;
12
13 architecture Behavioral of soma32b is
14     Signal NUM2 :STD_LOGIC_VECTOR (0 to 31) := "00000000000000000000000000000100";
15
16 begin
17     out1 <= NUM1 + NUM2;
18
19 end Behavioral;
```

MINI_MUX:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  -- Entidade mini_mux_2, multiplexador de 2 bits
5  entity mini_mux_2 is
6
7
8      Port ( SEL : in STD_LOGIC; -- Entrada de seleção, podendo ser 1 ou 0
9            A : in STD_LOGIC_VECTOR (0 to 4); -- Entrada A do multiplexador
10           B : in STD_LOGIC_VECTOR (0 to 4); -- Entrada B do multiplexador
11           X : out STD_LOGIC_VECTOR (0 to 4)); -- Única saída do multiplexador
12 end mini_mux_2; -- Finalizando a entidade mini_mux_2
13
14 architecture Behavioral of mini_mux_2 is --Architecture Behavioral
15 begin
16     X <= A when (SEL = '1') else B;
17     -- Se a entrada "SEL" for igual a 1, a saída X recebe A, se não recebe B
18
19 end Behavioral; -- Finalizando Behavioral
```

SHIFT_JUMP:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.numeric_std.all;
4
5
6  entity shift_jump is
7  |
8  |   PORT (
9  |       in1 : IN STD_LOGIC_VECTOR(0 TO 25) ;--operandos
10 |       out1 : OUT STD_LOGIC_VECTOR(0 TO 27) ) ;--saida
11 |
12 |
13 |
14 |   end shift_jump;
15 |
16 | architecture Behavioral of shift_jump is
17 | | begin
18 | |   out1 <= in1(0 to 25) & "00";
19 | | end Behavioral;
```

SHIFTL_2:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.numeric_std.all;
4
5
6  entity shl_2 is
7  |
8  |   PORT (
9  |       in1 : IN STD_LOGIC_VECTOR(0 TO 31) ;--operandos
10 |       out1 : OUT STD_LOGIC_VECTOR(0 TO 31) ) ;--saida
11 |
12 |
13 |
14 |   end shl_2;
15 |
16 | architecture Behavioral of shl_2 is
17 | | begin
18 | |   out1 <= in1(2 to 31) & "00";
19 | | end Behavioral;
```

EXTEND:

```
1  -- Declaração de todas as bibliotecas utilizadas
2  LIBRARY ieee;
3  USE ieee.numeric_std.all;
4  USE IEEE.STD_LOGIC_1164.ALL;
5
6  -- Entity extend
7  ENTITY extend is
8  |
9  |   -- Declaração das portas utilizadas no extend
10 |   PORT (
11 |       in1 : IN STD_LOGIC_VECTOR(0 TO 15) ;-- Operandos
12 |       out1 : OUT STD_LOGIC_VECTOR(0 TO 31) -- Saída
13 |   );
14 |
15 |   --Finalizando a entity extend
16 |   END extend;
17 |
18 |
19 |   -- Architecture tb
20 |   ARCHITECTURE tb of extend is
21 | |
22 | |   begin
23 | |
24 | |       -- Utilizado para verificar o tamanho que seria 32 bits e cocatenar os vetores
25 | |       out1 <= std_logic_vector(resize(signed(in1), out1'length));
26 | |
27 | |   -- Finalizando o tb
28 | |   end tb;
```

ALU:

```
1  -- Declaração de todas as bibliotecas utilizadas
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_signed.all;
5
6  -- Entity ALU
7  ENTITY ALU IS
8
9      -- Declaração das portas utilizadas na ALU
10     PORT (
11         aluop : IN STD_LOGIC_VECTOR(0 TO 1) ; -- Aluop
12         A, B : IN STD_LOGIC_VECTOR(0 TO 31) ; -- Operandos
13         Saida : OUT STD_LOGIC_VECTOR(0 TO 31); -- Saida
14         zero : out std_logic -- Zero
15     );
16
17     --Finalizando a entity Ula
18     END ALU ;
19
20     -- Architecture Behavior
21     ARCHITECTURE Behavior OF ALU IS
22
23         -- Declaração do "aux" um sinal do tipo vetor de 31 posições
24         signal aux: std_logic_vector(0 to 31);
25
26     BEGIN
27
28         -- Inicializando o processo com Aluop, A e B
29         PROCESS (aluop, A, B)
30
31             BEGIN
32
33                 -- Caso aluop for igual a...
34                 CASE aluop IS
35                     |
36                     -- Enquanto for igual a 00, Realiza a soma entre operandos
37                     WHEN "00"=>
38                         aux <= (A + B);
39
40                     -- Enquanto for igual a 01, Realiza a subtração entre operandos
41                     WHEN "01"=>
42                         aux <= (A - B);
43
44                     -- Enquanto for igual a 10, Realiza and entre operandos
45                     WHEN "10"=>
46                         aux <= (A AND B);
47
48                     -- Enquanto for igual a 11, Realiza or entre operandos
49                     WHEN "11"=>
50                         aux <= (A OR B); --OR
51
52                     -- Enquanto for outro alem desses, saida recebe 0
53                     WHEN others => Saida <= "00000000000000000000000000000000";
54
55                     -- Fianlizando o Case
56                     END CASE ;
57
58                     -- Se Aux for igual a 0, zero recebe 1
59                     IF(aux = "00000000000000000000000000000000") THEN
60                         zero <= '1';
61
62                     -- Senão zero recebe 0
63                     ELSE
64                         zero <= '0';
65
66                     -- Finalizando o If
67                     END IF;
68
69                     -- Saida recebe aux
70                     Saida <= aux;
71
72                     --Finalizando o processo
73                     END PROCESS;
74
75                     -- Finalizando behavior
76                     END Behavior;
```