

Project 3: Designing a 32-bit CPU

Adam Sumner - A20283081, **Contribution** - 25%

Bobby Unverzagt - A2028923, **Contribution** - 25%

Emilie Woog - A20265269, **Contribution** - 25%

Nash Kaminski - A20283999, **Contribution** - 25%

ECE 485

December 5th, 2015

1 Introduction

This goal of this project is to design a stripped down version of the MIPS processor. The processor will be a 32-bit version of the processor discussed in class and the text book, however, its instruction set will be a small subset of the MIPS processor's full capability.

1.1 Background Information

1.1.1 MIPS

MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA). It defines three types of instruction types: R (register), I (Immediate), and J (Jump). For the implementation that this project is focused on, only R and I instructions will be executed. R type instructions are the most common form of instructions. The format for an r-type instruction is:

Bits[31:26]	Bits[25:21]	Bits[20:16]	Bits[15:11]	Bits[10:6]	Bits[5:0]
opcode	Rs	Rt	Rd	shamt	funct

For this instruction, the opcode field is always 000000_2 , while the function code **funct** is used to determine which instruction is to be carried out. Rs and Rt are the two registers in which the operation reads and Rd is the destination of the result. Some instructions require a shift amount (**shamt**), so it is specified explicitly.

The I type instruction involves an immediate value, so the instruction format must accommodate this. The format of this type of instruction is:

Bits [31:26]	Bits [25:21]	Bits [20:16]	Bits [15:0]
opcode	Rs	Rt	immediate

For this instruction, the op code field is used to define the specific instruction, Rs is the register in which the operation acts on along with the immediate value as the other operand. Rt is the destination register in which the result is stored.

1.1.2 Datapath and Control

A datapath is a collection of functional units that perform data processing operations. It includes units such as a program counter, a register file, instruction memory, an ALU, data memory, and a control unit. Figure 1 shows a high level overview of a simple datapath with control.

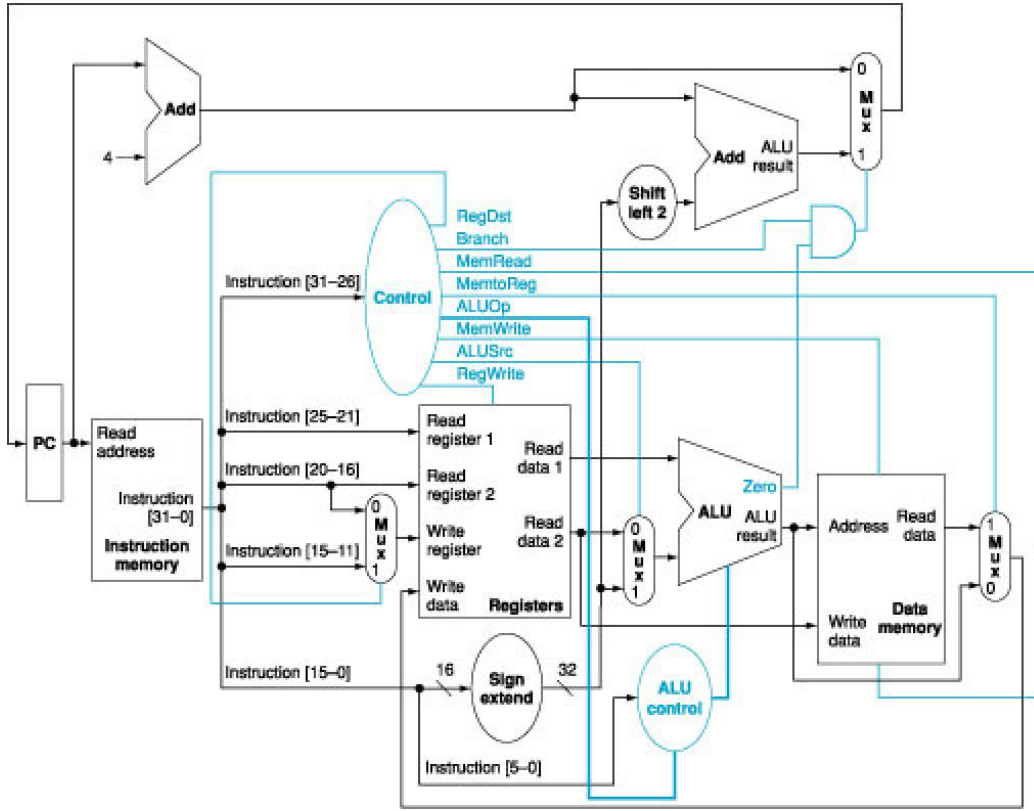


Figure 1: Datapath Overview

2 Design

2.1 Instruction Set

Table 1 shows the instructions that were chosen to be implemented in the CPU with the respective OpCode and Function Field for each instruction.

OpCode[31:26]	Function Field [5:0]	Instruction	Operation
100011 ₂	--	lw	lw \$t3, 200(\$s2)
101011 ₂	--	sw	sw \$t4, 100(\$t3)
000000 ₂	100000 ₂	add	add \$s3, \$t2, \$s2
000000 ₂	110000 ₂	sub	sub \$s3, \$t2, \$s2
000100 ₂	--	beq	beq \$s5, \$s2, 500
000000 ₂	000001 ₂	nand	tbd
000010 ₂	--	andi	tbd
000000 ₂	000010 ₂	or	tbd
000011 ₂	--	ori	tbd

Table 1: CPU Instruction Set

Because it was only required to implement 9 instructions, and the MIPS instruction set format requires 6 bits for op code and function field, it was an easy decision to choose these values for the implemented instructions. For all R-type instructions, the functions fields were chosen to be vastly different from one another to make debugging easier for the team. Likewise, the same approach was taken for the op code decisions for the I-type instructions.

2.2 Memory

For this project, it seemed unnecessary to implement memory of 4GB (2^{32}). It was chosen to use an array of 256 words instead. If need be, this memory size could be upgraded easily, so this choice does not hinder performance on the actual design of the CPU.

2.3 Datapath

Because of the simplicity of this design, the implemented datapath did not need to be modified by much from Figure 1. Therefore, the design of a single cycle datapath from the textbook acted as the skeletal structure of the final implementation. Figure 2 shows the overview block diagram of the implemented datapath for the CPU.

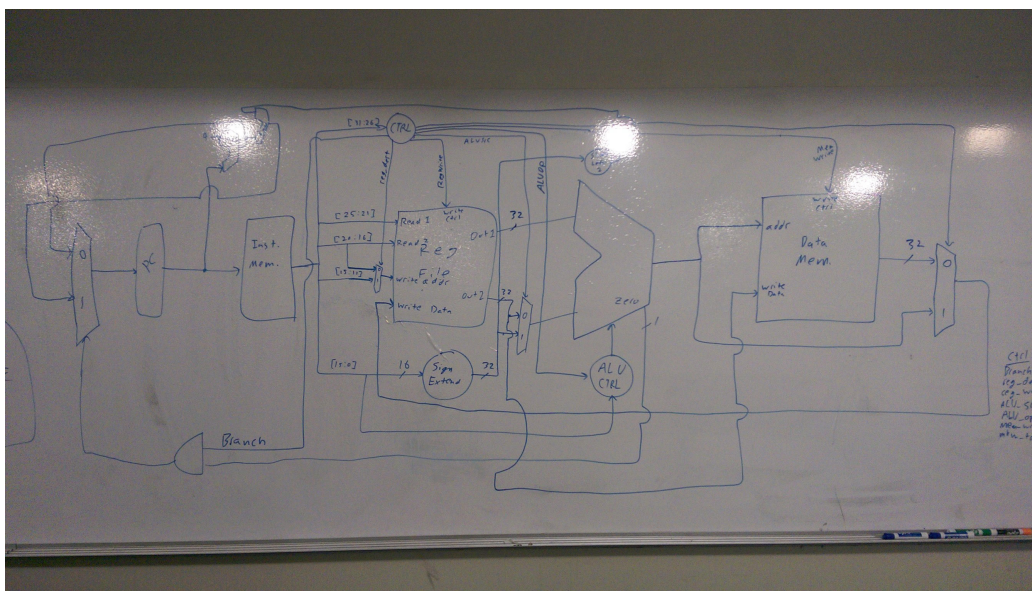


Figure 2: Implemented Data Path Overview

2.4 Control

The control can also be seen in Figure 2. It is a simple design of several signals acting as the `sel` line of a series of multiplexers. Based on the op codes and function field read from the instruction memory, the signals are asserted accordingly to relay the correct signals into the Register File, ALU, and Data memory.

3 Analysis

While this processor was optimized to be able to fully accomplish the tasks specified in the business requirements document, it could still be improved. In its current stage, it can be considered a bare bones prototype. To transform the current design into a processor on par with the current industry standard, a complete instruction set would have to be implemented. Furthermore, pipelining is a necessity to add. Any processor that doesn't implement pipelining is not making efficient use of its own components.

4 Simulation Results

Include blurb about test bench code and all figures of waveforms explaining each section of the test.

5 Conclusion

The design and implementation of a 32-bit CPU was a success. a set of 9 instructions were successfully implemented and verified with test bench code. All requested functionality was achieved. This 32-bit CPU can now be used in further projects.

Appendix

Listing 1: CPU Code

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity regFile is
6   port(
7     regA : out std_logic_vector(31 downto 0);
8     regB : out std_logic_vector(31 downto 0);
9     selA : in std_logic_vector(3 downto 0);
10    selB : in std_logic_vector(3 downto 0);
11    wData : in std_logic_vector(31 downto 0);
12    registerWrite : in std_logic;
13    selW : in std_logic_vector(3 downto 0);
14    clk : in std_logic);
15 end regFile;
16
17 architecture behavioral of regFile is
18   type reg_arr is array(0 to 15) of std_logic_vector(31
19     downto 0);
19   signal rData : reg_arr;
20   begin
21     with selA
```

```

22     select regA <= x"00000000" when b"0000",
23     rData(to_integer(unsigned(selA))) when others;
24 with selB
25     select regB <= x"00000000" when b"0000",
26     rData(to_integer(unsigned(selB))) when others;
27
28 wrProc: process(clk) is
29 begin
30     if falling_edge(clk) then
31         if(registerWrite = '1') then
32             rData(to_integer(unsigned(selW))) <= wData;
33         end if;
34     end if;
35 end process;
36 end behavioral;
37
38 ---
39
40 library ieee;
41 use ieee.std_logic_1164.all;
42 use ieee.numeric_std.all;
43
44 entity control is
45     port(
46         inst_in : in std_logic_vector(5 downto 0);
47         func    : in std_logic_vector(5 downto 0);
48         stall   : in std_logic;
49         branch  : out std_logic;
50         reg_dest : out std_logic;
51         reg_write : out std_logic;
52         ALU_src  : out std_logic;
53         ALU_op   : out std_logic_vector(2 downto 0);
54         mem_write : out std_logic;
55         mem_to_reg : out std_logic
56     );
57 end control;
58
59 architecture behavioral of control is
60     signal branch_o, reg_dest_o, reg_write_o, ALU_src_o,
61         mem_write_o, mem_to_reg_o : std_logic;

```

```

61  signal ALU_op_o : std_logic_vector(2 downto 0);
62  signal branch_f, reg_dest_f, reg_write_f, ALU_src_f,
    mem_write_f, mem_to_reg_f : std_logic;
63  signal ALU_op_f : std_logic_vector(2 downto 0);
64  begin
65  -- set intermediate signals incase of r-type
    instruction
66  with func select
67      branch_f <= '0' when "100000", --add
68          '0' when "110000", --sub
69          '0' when "000001", --nand
70          '0' when "000010", --or
71          '0' when others;
72  with func select
73      reg_dest_f <= '1' when "100000", --add
74          '1' when "110000", --sub
75          '1' when "000001", --nand
76          '1' when "000010", --or
77          'Z' when others;
78  with func select
79      reg_write_f <= '1' when "100000", --add
80          '1' when "110000", --sub
81          '1' when "000001", --nand
82          '1' when "000010", --or
83          'Z' when others;
84  with func select
85      ALU_src_f <= '0' when "100000", --add
86          '0' when "110000", --sub
87          '0' when "000001", --nand
88          '0' when "000010", --or
89          'Z' when others;
90  with func select
91      ALU_op_f <= "000" when "100000", --add
92          "001" when "110000", --sub
93          "010" when "000001", --nand
94          "100" when "000010", --or
95          "ZZZ" when others;
96  with func select
97      mem_write_f <= '0' when "100000", --add
98          '0' when "110000", --sub

```



```

99         '0' when "000001", --nand
100        '0' when "000010", --or
101        'Z' when others;
102    with func select
103        mem_to_reg_f <= '1' when "100000", --add
104        '1' when "110000", --sub
105        '1' when "000001", --nand
106        '1' when "000010", --or
107        'Z' when others;
108
109    -- set intermediate signals incase of non r-type
110    instruction
111    with inst_in select
112        branch_o <= '0' when "100011", --lw
113        '0' when "101011", --sw
114        '1' when "000100", --beq
115        '0' when "000010", --andi
116        '0' when "000011", --ori
117        '0' when others;
118    with inst_in select
119        reg_dest_o <= '0' when "100011", --lw
120        '0' when "101011", --sw
121        '0' when "000100", --beq
122        '0' when "000010", --andi
123        '0' when "000011", --ori
124        'Z' when others;
125    with inst_in select
126        reg_write_o <= '1' when "100011", --lw
127        '0' when "101011", --sw
128        '0' when "000100", --beq
129        '1' when "000010", --andi
130        '1' when "000011", --ori
131        'Z' when others;
132    with inst_in select
133        ALU_src_o <= '1' when "100011", --lw
134        '1' when "101011", --sw
135        '0' when "000100", --beq
136        '1' when "000010", --andi
137        '1' when "000011", --ori
138        'Z' when others;

```

```

138 with inst_in select
139     ALU_op_o <= "000" when "100011",  --lw
140     "000" when "101011",  --sw
141     "001" when "000100",  --beq
142     "011" when "000010",  --andi
143     "100" when "000011",  --ori
144     "ZZZ" when others;
145 with inst_in select
146     mem_write_o <= '0' when "100011",  --lw
147     '1' when "101011",  --sw
148     '0' when "000100",  --beq
149     '0' when "000010",  --andi
150     '0' when "000011",  --ori
151     'Z' when others;
152 with inst_in select
153     mem_to_reg_o <= '0' when "100011",  --lw
154     '1' when "101011",  --sw
155     '1' when "000100",  --beq
156     '1' when "000010",  --andi
157     '1' when "000011",  --ori
158     'Z' when others;
159
160 -- select from intermediate signals
161 with inst_in select
162     branch <= branch_f when "000000",
163     branch_o when others;
164 with inst_in select
165     reg_dest <= reg_dest_f when "000000",
166     reg_dest_o when others;
167 with inst_in select
168     reg_write <= reg_write_f when "000000",
169     reg_write_o when others;
170 with inst_in select
171     ALU_src <= ALU_src_f when "000000",
172     ALU_src_o when others;
173 with inst_in select
174     ALU_op <= ALU_op_f when "000000",
175     ALU_op_o when others;
176 with inst_in select
177     mem_write <= mem_write_f when "000000",

```

```

178         mem_write_o when others;
179     with inst_in select
180         mem_to_reg <= mem_to_reg_f when "000000",
181         mem_to_reg_o when others;
182 end behavioral;
183
184 ---
185
186 library ieee;
187 use ieee.std_logic_1164.all;
188 use ieee.numeric_std.all;
189
190 entity dataMem is
191     port(
192         data : out std_logic_vector(31 downto 0);
193         sel  : in  std_logic_vector(31 downto 0);
194         wData : in std_logic_vector(31 downto 0);
195         memWrite : in std_logic;
196         clk : in std_logic);
197 end dataMem;
198
199 architecture behavioral of dataMem is
200     type mem_arr is array(0 to 255) of std_logic_vector(31
201         downto 0);
202     signal mData : mem_arr;
203     begin
204         data <= mData(to_integer(resize(unsigned(sel),8)));
205
206         wrProc: process(clk) is
207         begin
208             if falling_edge(clk) then
209                 if(memWrite = '1') then
210                     mData(to_integer(resize(unsigned(sel),8))) <=
211                         wData;
212                 end if;
213             end if;
214         end process;
215     end behavioral;

```

```

216
217 ---
218 library ieee;
219 use ieee.std_logic_1164.all;
220 use ieee.numeric_std.all;
221
222 entity ALU is
223     port(
224         inA : in std_logic_vector(31 downto 0);
225         inB : in std_logic_vector(31 downto 0);
226         ctl : in std_logic_vector(2 downto 0);
227         res : out std_logic_vector(31 downto 0));
228 end ALU;
229
230 architecture behavioral of ALU is
231     signal add : std_logic_vector(31 downto 0);
232     signal sub : std_logic_vector(31 downto 0);
233     signal andres : std_logic_vector(31 downto 0);
234     signal nandres : std_logic_vector(31 downto 0);
235     signal orres : std_logic_vector(31 downto 0);
236     begin
237         add <= std_logic_vector(signed(inA)+signed(inB));
238         sub <= std_logic_vector(signed(inA)-signed(inB));
239         andres <= std_logic_vector(unsigned(inA) and
240             unsigned(inB));
241         nandres <= std_logic_vector(not(unsigned(inA) and
242             unsigned(inB)));
243         orres <= std_logic_vector(unsigned(inA) or unsigned(
244             inB));
245
246     -- Multiplexer
247     with ctl select
248         res <= add when "000",
249             sub when "001",
250             nandres when "010",
251             andres when "011",
252             orres when "100",
253             "00000000000000000000000000000000" when others;
254 end behavioral;
255

```

```

253 --
254
255 library ieee;
256 use ieee.std_logic_1164.all;
257 use ieee.numeric_std.all;
258 entity processor is
259     port(
260         extPC : in std_logic_vector(31 downto 0);
261         IMdata : in std_logic_vector(31 downto 0);
262         DMdata : in std_logic_vector(31 downto 0);
263         IMwrite : in std_logic;
264         DMwrite : in std_logic;
265         DMaddr : in std_logic_vector(31 downto 0);
266         stall : in std_logic;
267         clk : in std_logic
268     );
269 end processor;
270
271 architecture behavioral of processor is
272     signal im_wrEn, im_clk : std_logic;
273     signal im_data, im_addr, im_wData : std_logic_vector(31
274         downto 0);
275     signal dm_wrEn, dm_clk : std_logic;
276     signal dm_data, dm_addr, dm_wData : std_logic_vector(31
277         downto 0);
278     signal PC : std_logic_vector(31 downto 0);
279     signal regA, regB, wData : std_logic_vector(31 downto 0);
280     signal selA, selB, selW : std_logic_vector(3 downto 0);
281     signal aluCtl : std_logic_vector(2 downto 0);
282     signal regWrite, regDest, regClk, dm_write, aluSrc,
283         memtoreg : std_logic;
284     signal aluA, aluB, aluRes : std_logic_vector(31 downto 0);
285     signal branch, branchI, zero : std_logic := '0';
286     signal braAddr : std_logic_vector(15 downto 0);
287     signal op_code, func : std_logic_vector(5 downto 0);
288
289 begin
290     IM : entity work.dataMem port map(im_data, im_addr,
291         im_wData, im_wrEn, im_clk);

```

```

288  DM : entity work.dataMem port map(dm_data, dm_addr,
    dm_wData, dm_wrEn, dm_clk);
289  RF : entity work.regFile port map(regA, regB, selA,
    selB, wData, regWrite, selW, regClk);
290  ALU : entity work.ALU port map(aluA, aluB, aluCtl,
    aluRes);
291  CTRL : entity work.control port map(op_code, func,
    stall, branchI, regDest, regWrite, aluSrc, aluCtl,
    dm_write, memtoreg);

292
293  --all clocks synced
294  im_clk <= clk;
295  dm_clk <= clk;
296  regClk <= clk;
297
298  im_wData <= IMData;
299  im_wrEn <= IMWrite;
300  --allow testbench to initialize
301  process(clk)
302  begin
303  if(rising_edge(clk)) then
304      if(stall = '1') then
305          PC <= extPC;
306      elsif(branch = '1') then
307          PC <= std_logic_vector(unsigned(PC) + (
    unsigned(resize(signed(braAddr), 32))));
308      else
309          PC <= std_logic_vector(unsigned(PC) + x"1");
310      end if;
311  end if;
312  end process;
313  braAddr <= im_data(15 downto 0);
314  im_addr <= PC;
315  aluA <= regA;
316  with aluSrc
317      select aluB <= regB when '0',
318          std_logic_vector(unsigned(resize(signed(im_data
    (15 downto 0)), 32))) when '1',
319          x"00000000" when others;
320  with regDest

```

```

321         select selW <= im_data(19 downto 16) when '0',
322         im_data(14 downto 11) when '1',
323         "ZZZZ" when others;
324     with stall
325         select dm_addr <= DMaddr when '1',
326         aluRes when others;
327     with stall
328         select dm_wData <= DMdata when '1',
329         regB when others;
330     with stall
331         select dm_wrEn <= DMWrite when '1',
332         dm_write when others;
333
334     with memtoreg
335         select wData <= aluRes when '1',
336         dm_data when others;
337
338     with aluRes
339         select zero <= '1' when x"00000000",
340         '0' when others;
341
342     branch <= branchI and zero;
343
344     op_code <= im_data(31 downto 26);
345     func <= im_data(5 downto 0);
346
347     selA <= im_data(24 downto 21);
348     selB <= im_data(19 downto 16);
349 end behavioral;

```

Listing 2: Test Bench Code

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity test_bench is
6 end test_bench;
7
8 architecture behavioral of test_bench is
9     signal clk : std_logic;

```

```

10  signal extPC, IMdata, DMdata, DMaddr : std_logic_vector
    (31 downto 0) := x"00000000";
11
12  signal IMwrite, DMwrite, stall : std_logic := '0';
13  begin
14  cpu : entity work.processor port map(extPC, IMdata,
    DMdata, IMwrite, DMwrite, DMaddr, stall, clk);
15
16  -- clk process
17  clkgen: process
18  begin
19      clk <= '1';
20      wait for 1 ns;
21      clk <= '0';
22      wait for 1 ns;
23  end process;
24
25  tester: process
26  begin
27      -- init values
28      stall <= '1';
29      IMwrite <= '0';
30      DMwrite <= '1';
31      -- put some data into the DM
32      DMdata <= x"AAAAAAAA";
33      DMwrite <= '1';
34      DMaddr <= x"00000001";
35      wait for 2 ns;
36      DMdata <= x"BBBBBBBB";
37      DMwrite <= '1';
38      DMaddr <= x"00000002";
39      wait for 2 ns;
40      DMdata <= x"CCCCCCCC";
41      DMwrite <= '1';
42      DMaddr <= x"00000003";
43      wait for 2 ns;
44      DMdata <= x"FFFFFFF";
45      DMwrite <= '1';
46      DMaddr <= x"00000004";
47      wait for 2 ns;

```



```

48  DMdata <= x"00000000";
49  DMwrite <= '1';
50  DMaddr <= x"00000005";
51  wait for 2 ns;
52
53  — Now load program, start from address 1
54  DMwrite <= '0';
55  IMwrite <= '1';
56  — lw $1, 1($zero)
57  extPC <= x"00000001";
58  IMdata <= b"10001100000000001000000000000001";
59  wait for 2 ns;
60  — sw $1, 6($zero)
61  extPC <= x"00000002";
62  IMdata <= b"10101100000000001000000000000110";
63  wait for 2 ns;
64  — lw $2, 2($zero)
65  extPC <= x"00000003";
66  IMdata <= b"10001100000000001000000000000010";
67  wait for 2 ns;
68  — add $3, $1, $2
69  extPC <= x"00000004";
70  IMdata <= b"00000000001000100001100000100000";
71  wait for 2 ns;
72  — sub $4, $2, $1
73  extPC <= x"00000005";
74  IMdata <= b"00000000010000010010000000110000";
75  wait for 2 ns;
76  — beq $1, $2, 100
77  extPC <= x"00000006";
78  IMdata <= b"000100000010001000000000001100100";
79  wait for 2 ns;
80
81  — lw $2, 4($zero)
82  extPC <= x"00000007";
83  IMdata <= b"100011000000000010000000000000100";
84  wait for 2 ns;
85  — nand $5, $1, $2
86  extPC <= x"00000008";
87  IMdata <= b"00000000001000100010100000000001";

```

```

88     wait for 2 ns;
89     — andi $6, $2, 00FF
90     extPC <= x"00000009";
91     IMdata <= b"00001000010001100000000011111111";
92     wait for 2 ns;
93     — ori $7, $1, 00FF
94     extPC <= x"0000000A";
95     IMdata <= b"00001100001001110000000011111111";
96     wait for 2 ns;
97     — or $8, $1, $2
98     extPC <= x"0000000B";
99     IMdata <= b"00000000001000100100000000000010";
100    wait for 2 ns;
101    — beq $1, $1 -0x000B
102    extPC <= x"0000000C";
103    IMdata <= b"000100000010000111111111111110101";
104    wait for 2 ns;
105
106    — Begin execution here
107    wait for 2 ns;
108    IMwrite <= '0';
109    extPC <= x"00000000";
110    wait for 2 ns;
111    stall <= '0';
112
113    —allow enough time for processor to execute
        instructions
114    wait for 100 ns;
115
116    end process;
117 end behavioral;

```