Project 3: Designing a 32-bit CPU

Nash Kaminski - A20283999, **Contribution** - 25%
Adam Sumner - A20283081, **Contribution** - 25%
Bobby Unverzagt - A2028923, **Contribution** - 25%
Emilie Woog - A20265269, **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.

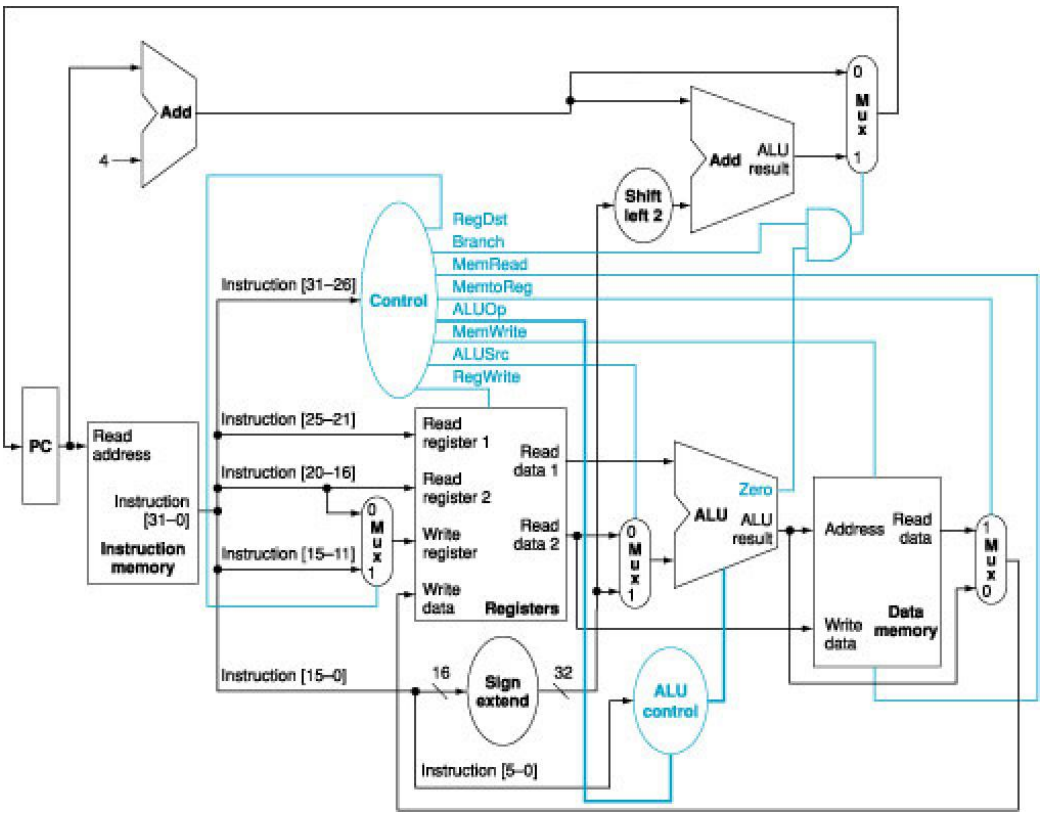

Figure 1: Datapath Overview

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	Example 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	nand $s5, $s1, $s2
000010_2	--	andi	andi $s6, $s2, 0x00FF
000000_2	000010_2	or	or $s8, $s1, $s2
000011_2	--	ori	ori $s7, $s1, 0x00FF

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. This memory is word addressable, as opposed to being byte addressable. 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. Because an ALU and Register File were previously implemented in earlier projects, it was necessary to extend their functionality to be able to handle 32-bit words. Once this was complete, this left the data memory entity to be completed so that it could be included in the processor entity. As mentioned earlier in Section 2.2, this entity contains an array of 256 words, and allows for reading and writing.

The processor entity combines all of the components into the desired datapath. It synchronizes the clock of the instruction memory, data memory, and register file so that the entire system is in sync with an external clock signal. The program counter is updated during the rising edge of the clock, and all writes happen on the falling edge. The processor relies on the control unit to carry out the instruction read from memory. Figure 2 shows the overview block diagram of the implemented datapath for the CPU.

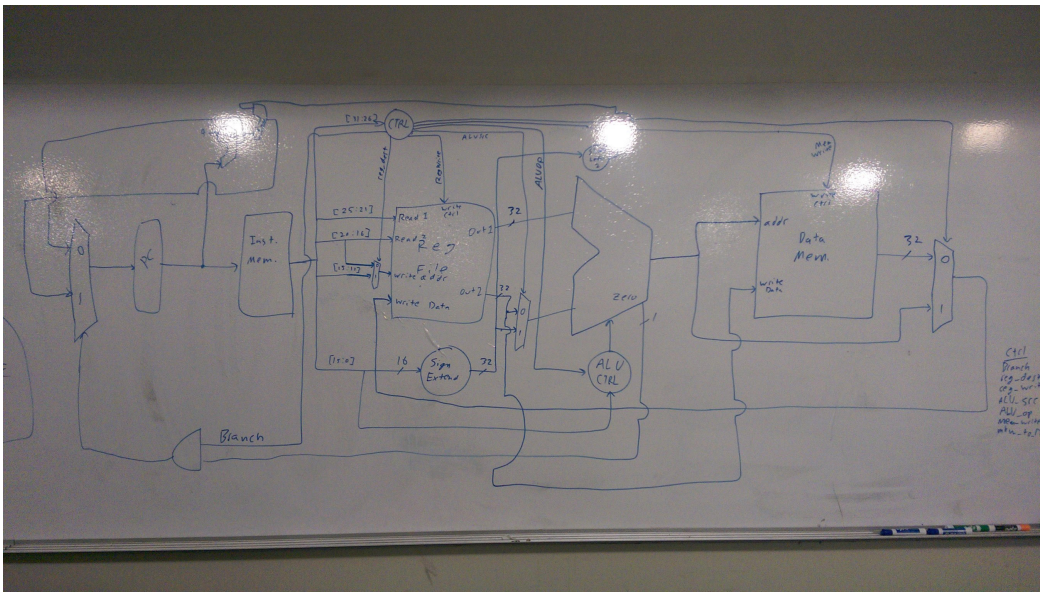

Figure 2: Implemented Data Path Overview

2.4 Control

The control lines can also be seen in Figure 2. It is a simple design of several signals acting as the `sel` line for 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. This unit is what determines which units will read/write, and what operations the ALU should perform.

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. After pipelining is implemented, hazard controls would need to coexist. This would allow for cool features of the processor to exist such as forwarding, making it a truly efficient piece of hardware.

4 Simulation Results

Once the processor was completely designed, it was necessary to write some test bench code. To test each instruction, data had to first be written to memory, along with the program being loaded onto the CPU. Due to the amount of signals involved in the CPU, not all will be shown in the simulation. The clock, contents of the registers, data memory, and program counter will only be shown. Data Memory addresses 0x00000001 $\rightarrow$ 0x00000005 were initialized with starting data. For simplicity, register numbers 1 $\rightarrow$ 8 are s registers. This is not the convention in a usual MIPS implemented processor, however, for testing the instructions, this assignment is arbitrary. Please refer to the Test Bench Code in the Appendix for a detailed view of the testing procedure. The execution of the program begins at 38ns. The tested instructions are:

1. `lw $s1, 1($zero)`
2. `sw $s1, 6($zero)`
3. `lw $s2, 2($zero)`

4. `add $s3, $s1, $s2`
5. `sub $s4, $s2, $s1`
6. `beq $s1, $s2, 100`
7. `lw $2, 4($zero)`
8. `nand $s5, $s1, $s2`
9. `andi $s6, $s2, 0x00FF`
10. `ori $s7, $s1, 0x00FF`
11. `or $s8, $s1, $s2`
12. `beq $s1, $s1, -0x000B`

Figure 3 shows the first instruction being executed. The data memory at address 0x00000001 holds the value 0xAAAAAAAA and register $s1 is subsequently loaded with the data 0xAAAAAAAA.

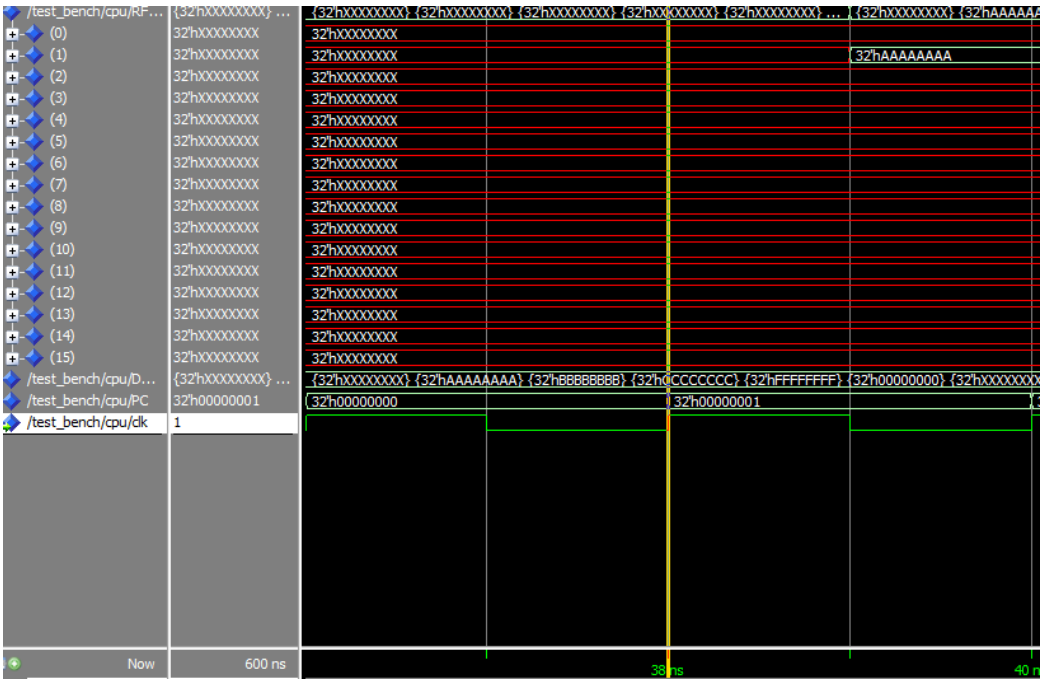

Figure 3: `lw $s1, 1($zero)`

Figure 4 shows the second instruction. The value of 0xAAAAAAAA in register $s1 is successfully stored into the data memory at address 0x00000006.

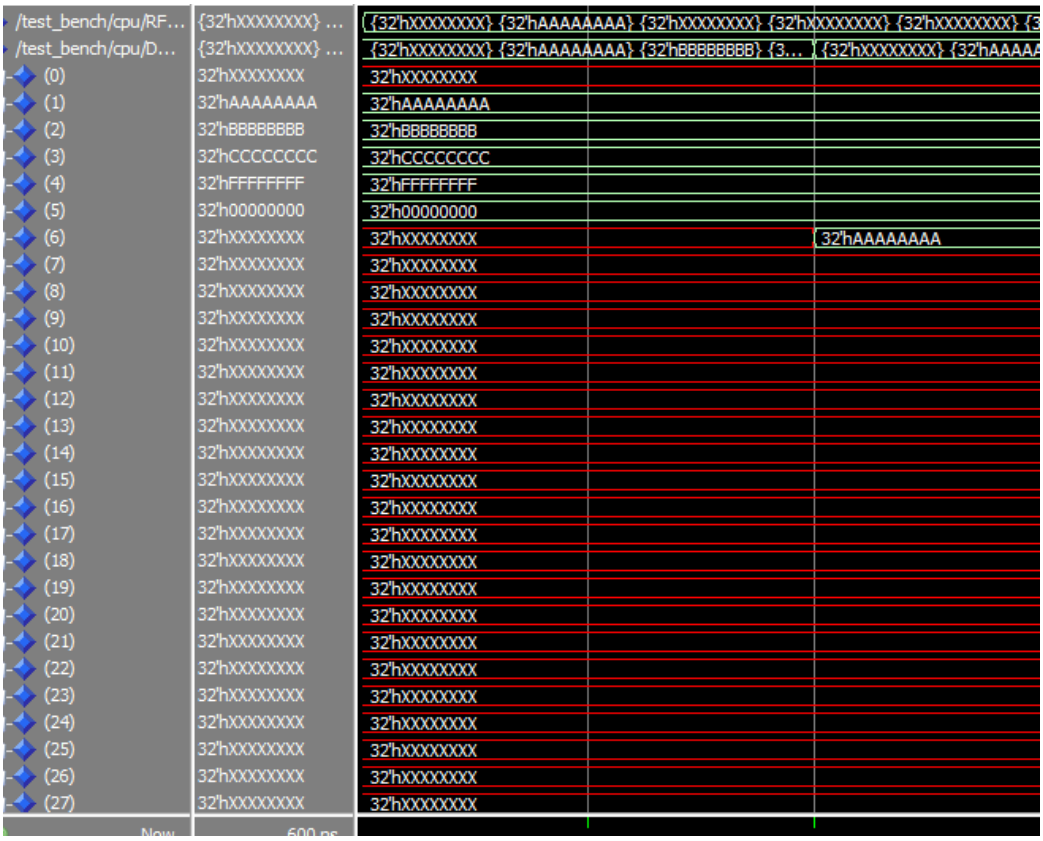

Figure 4: `sw $s1, 6($zero)`

Figure 5 shows the third instruction. The value of 0xBBBBBBBB at address 0x00000002 is successfully loaded into register $s2.

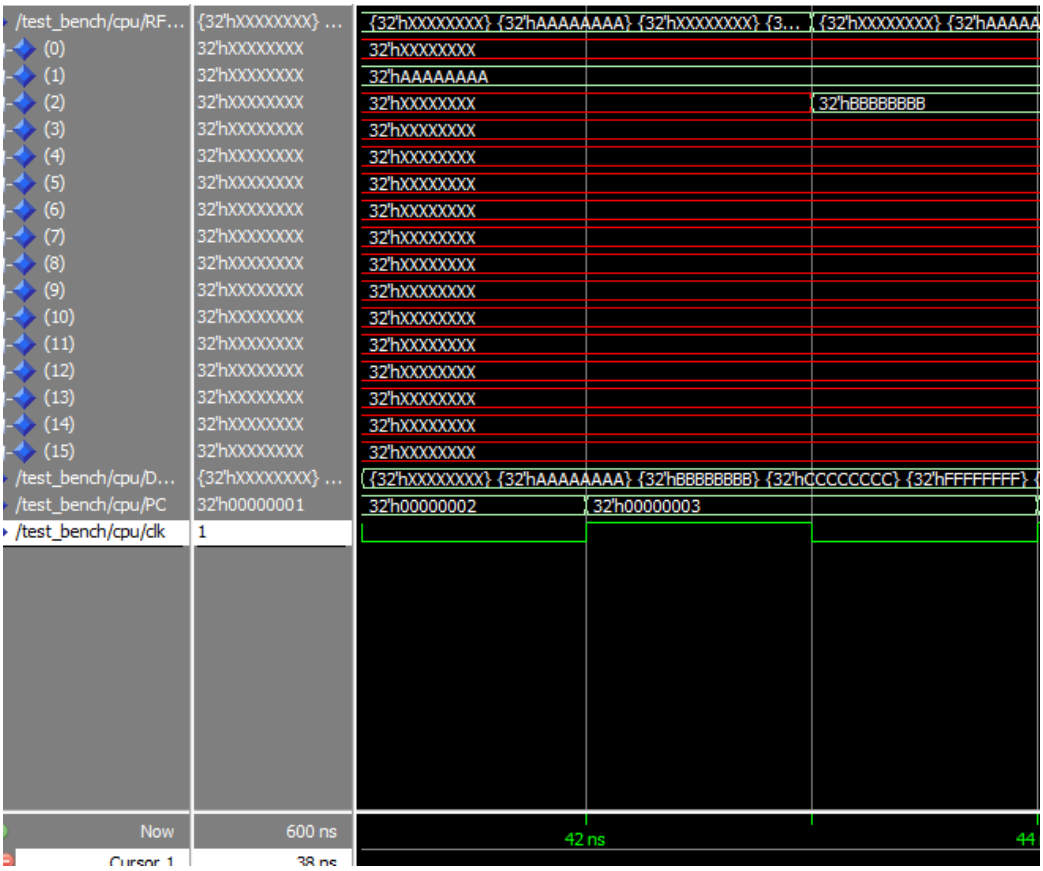

Figure 5: `lw $s2, 2($zero)`

Figure 6 shows the fourth instruction. The values of 0xAAAAAAAA and 0xBBBBBBBB are successfully added with the correct result of 0x66666665 being written into register $s3.

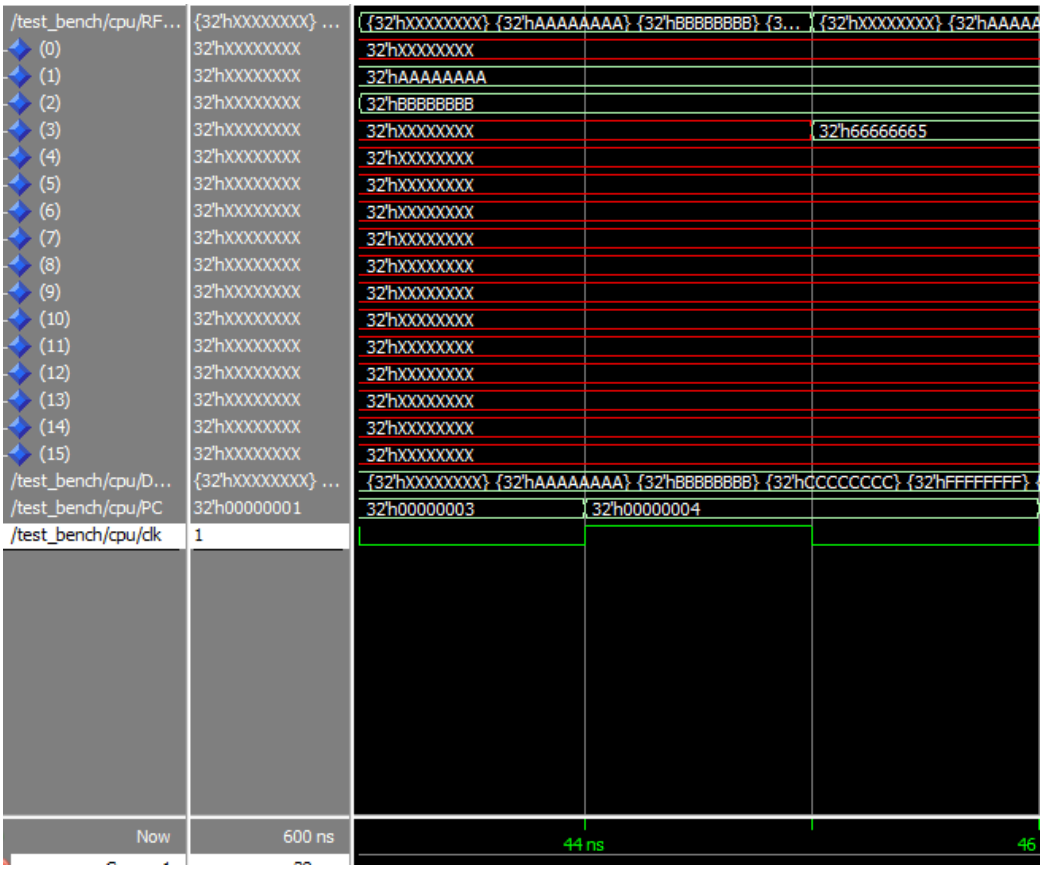

Figure 6: `add $s3, $s1, $s2`

Figure 7 shows the fifth instruction. The value of 0xAAAAAAAA in register $s1 is subtracted from register $s2. The result is 0x11111111 and is written back into register $s4.

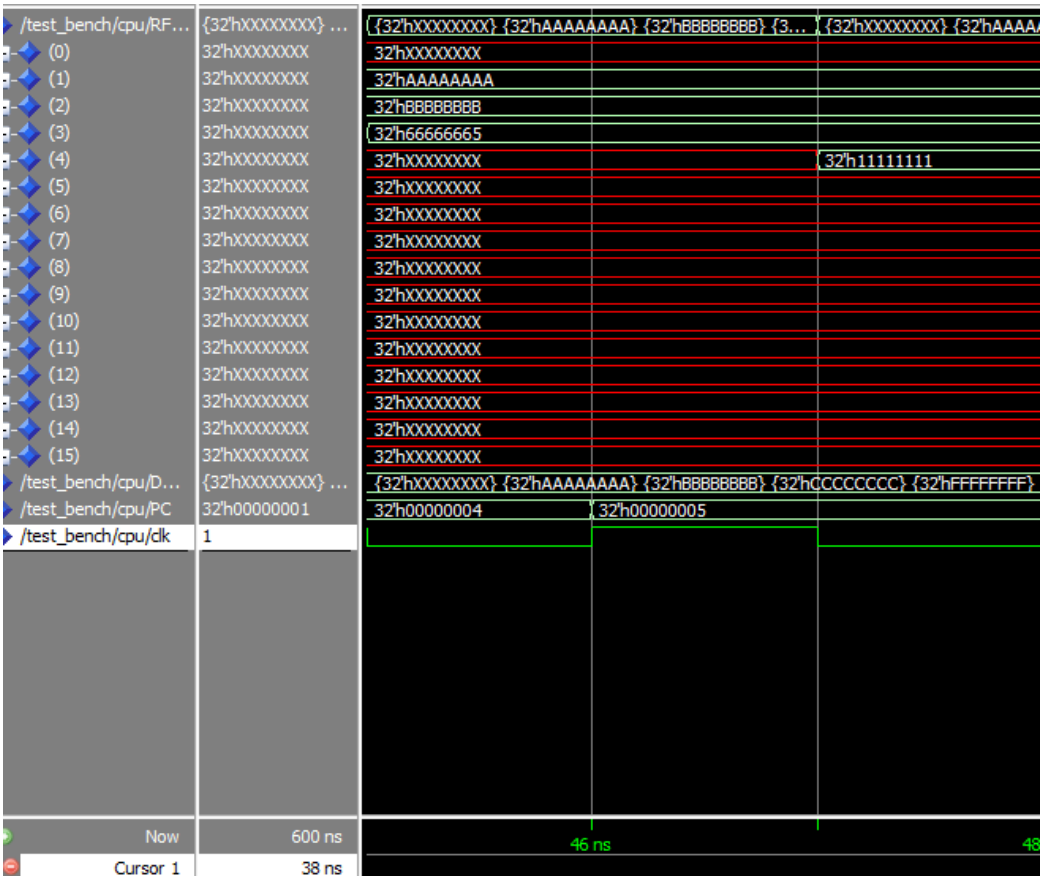

Figure 7: `sub $s4, $s2, $s1`

Figure 8 shows the sixth instruction. The value of 0xBBBBBBBB in register $s2 is compared to 0xAAAAAAAA in register $s1 for equivalence. Since they are not, the program counter increments to the next instruction.

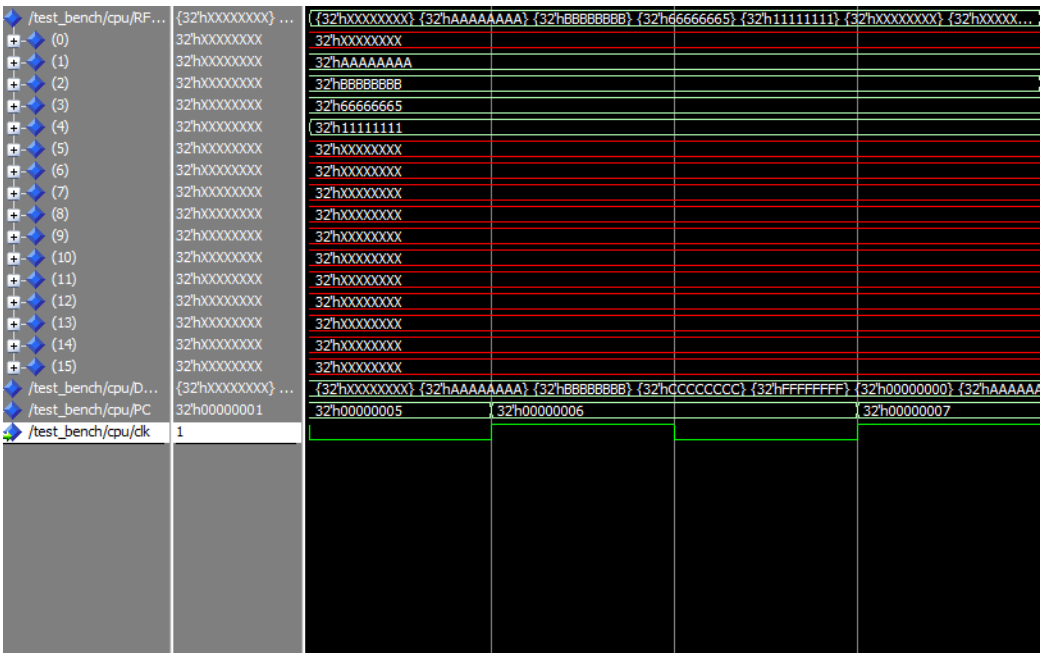

Figure 8: `beq $s1, $s2, 100`

Figure 9 shows the seventh instruction. The value of 0xFFFFFFFF at address 0x00000004 is loaded into register $s2 successfully.

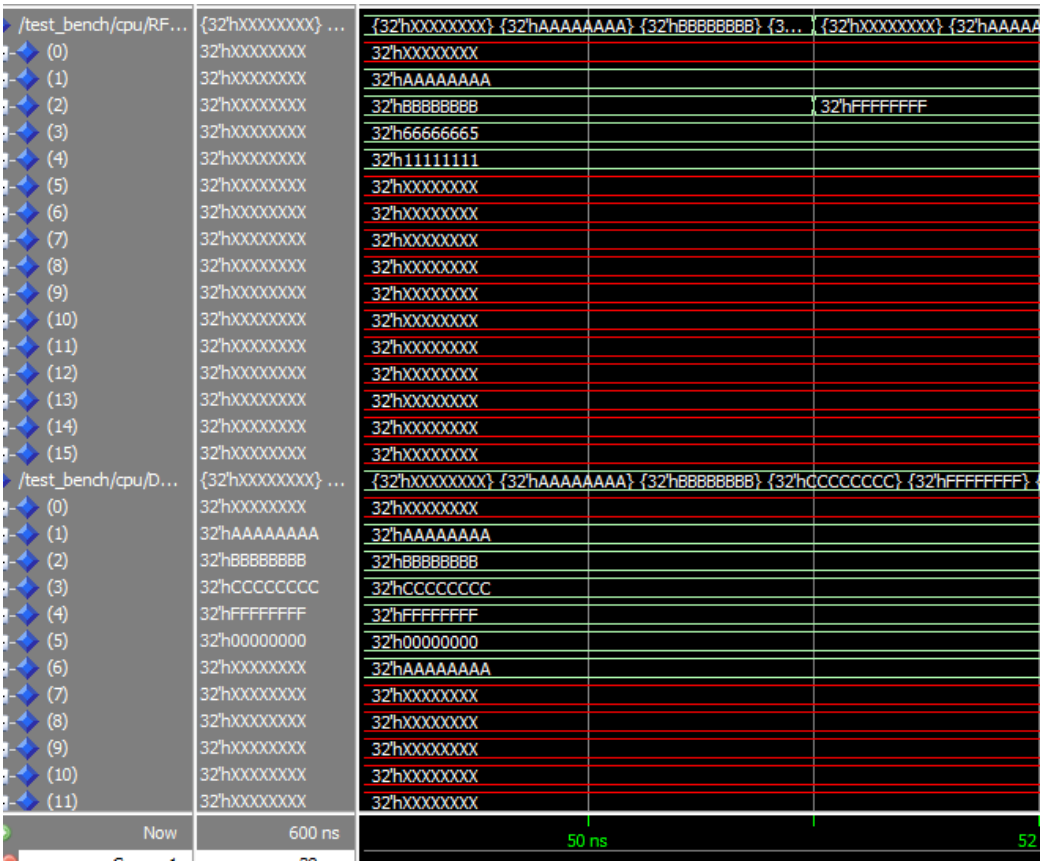

Figure 9: `lw $s2, 4($zero)`

Figure 10 shows the eighth instruction. The value of 0xFFFFFFFF in register $s2 and 0xAAAAAAAA in register are nanded and the result of 0x55555555 is both accurate and stored in register $s5.

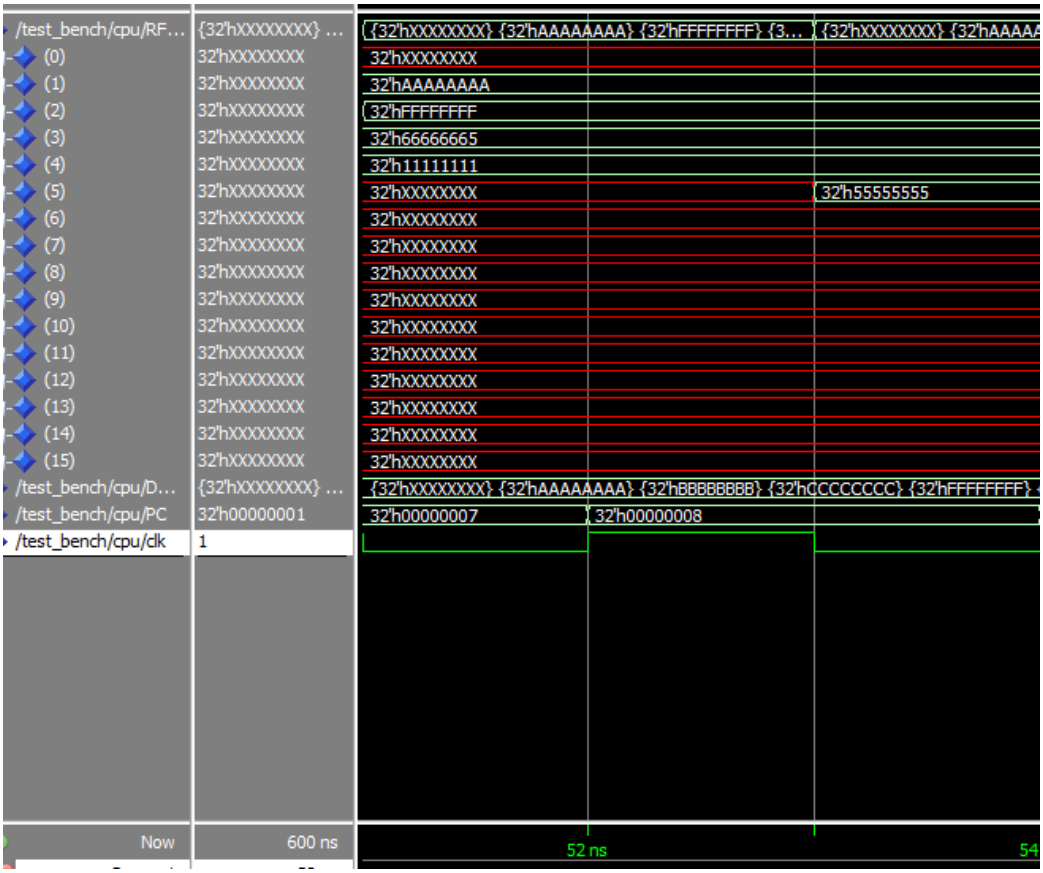

Figure 10: `nand $s5, $s1, $s2`

Figure 11 shows the ninth instruction. The immediate value of 0x000000FF is anded with the value of 0xFFFFFFFF in register $s2. The correct result of 0x000000FF is calculated and written into register $s6.

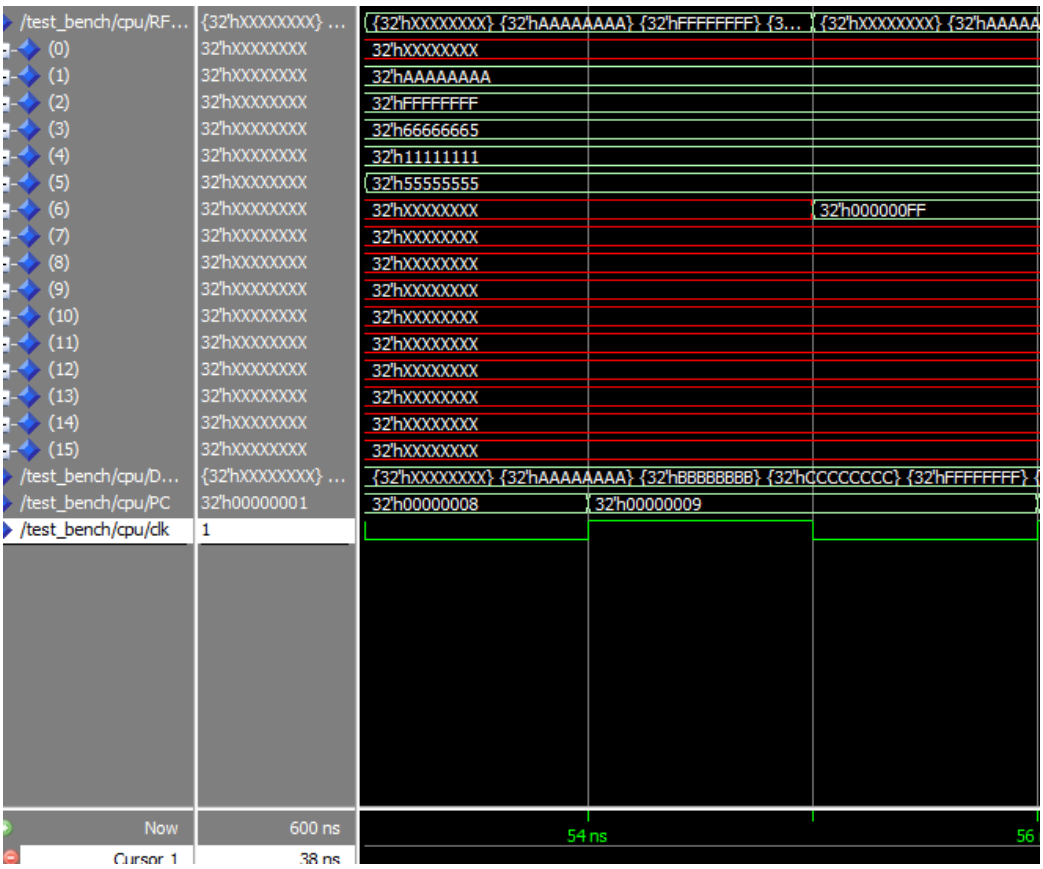

Figure 11: `andi $s6, $s2, 0x00FF`

Figure 12 shows the tenth instruction. The immediate value of 0x000000FF is or'ed with the value of 0xAAAAAAAA in register $s1. The correct result of 0xAAAAAAFF is written into register $s7.

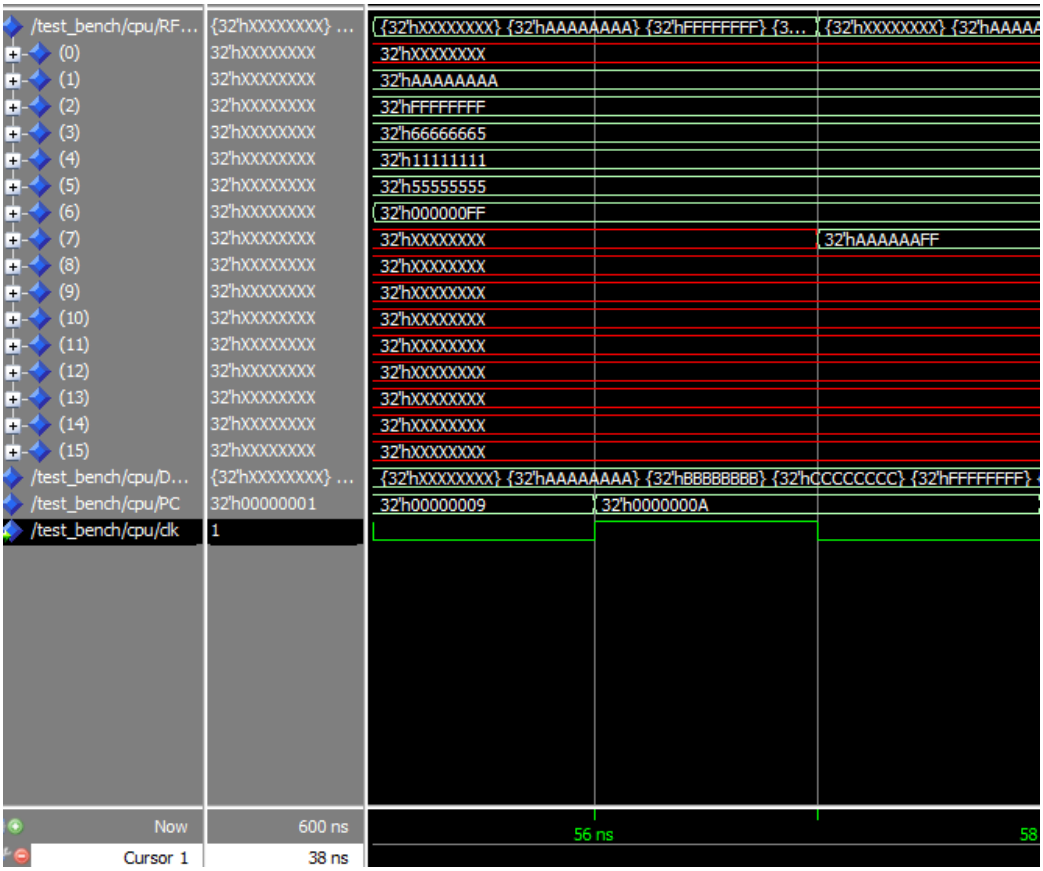

Figure 12: `ori $s7, $s1, 0x00FF`

Figure 13 shows the eleventh instruction. The value of 0xFFFFFFFF in register $s2 and 0xAAAAAAAA in register $s1 are or'ed. The correct result of 0xFFFFFFFF is written into register $s8.

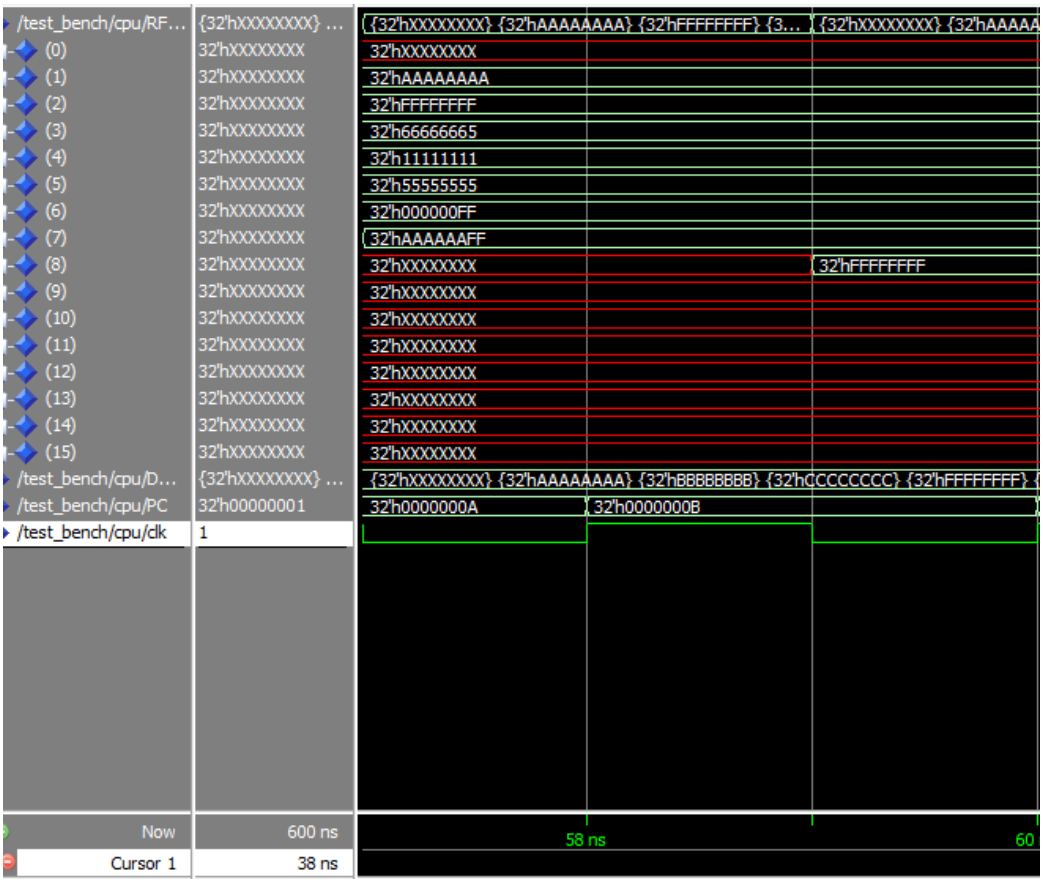

Figure 13: `or $s8, $s1, $s2`

Figure 14 shows the twelfth instruction. The value in register $s1 is checked for equivalence with itself. Since it is equal, the program counter branches backwards successfully back to 0x00000001. Thus the test program infinitely loops.

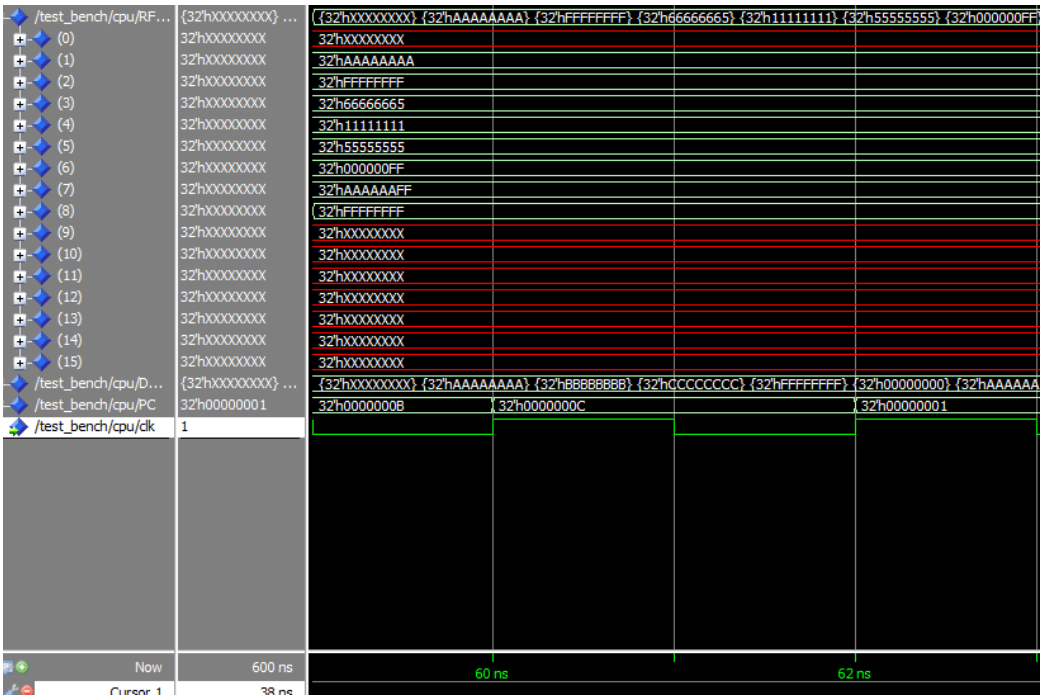

Figure 14: `beq $s1, $s1, -0x000B`

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 and can be expanded upon to become a more efficient piece of hardware.

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
     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,
      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
        instruction
110   with inst_in select
111     branch_o <= '0' when "100011",  --lw
```

```
112         '0' when "101011",  --sw
113         '1' when "000100",  --beq
114         '0' when "000010",  --andi
115         '0' when "000011",  --ori
116         '0' when others;
117     with inst_in select
118       reg_dest_o <= '0' when "100011",  --lw
119         '0' when "101011",  --sw
120         '0' when "000100",  --beq
121         '0' when "000010",  --andi
122         '0' when "000011",  --ori
123         'Z' when others;
124     with inst_in select
125       reg_write_o <= '1' when "100011", --lw
126         '0' when "101011",  --sw
127         '0' when "000100",  --beq
128         '1' when "000010",  --andi
129         '1' when "000011",  --ori
130         'Z' when others;
131     with inst_in select
132       ALU_src_o <= '1' when "100011", --lw
133         '1' when "101011",  --sw
134         '0' when "000100",  --beq
135         '1' when "000010",  --andi
136         '1' when "000011",  --ori
137         '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
      downto 0);
201 signal mData : mem_arr;
202 begin
203   data <= mData(to_integer(resize(unsigned(sel),8)));
204
205   wrProc: process(clk) is
206   begin
207     if falling_edge(clk) then
208     if(memWrite = '1') then
209           mData(to_integer(resize(unsigned(sel),8))) <=
      wData;
210     end if;
211     end if;
212   end process;
213 end behavioral;
214
215
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
      unsigned(inB));
240     nandres <= std_logic_vector(not(unsigned(inA) and
      unsigned(inB)));
241     orres <= std_logic_vector(unsigned(inA) or unsigned(
      inB));
242
243   -- Multiplexer
244   with ctl select
245     res <= add when "000",
246       sub when "001",
247       nandres when "010",
248       andres when "011",
249       orres when "100",
250       "00000000000000000000000000000000" when others;
251 end behavioral;
252
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
      downto 0);
274 signal dm_wrEn,dm_clk : std_logic;
275 signal dm_data,dm_addr,dm_wData : std_logic_vector(31
      downto 0);
276 signal PC : std_logic_vector(31 downto 0);
277 signal regA,regB,wData : std_logic_vector(31 downto 0);
278 signal selA,selB,selW : std_logic_vector(3 downto 0);
279 signal aluCtl : std_logic_vector(2 downto 0);
280 signal regWrite, regDest, regClk, dm_write, aluSrc,
      memtoreg : std_logic;
281 signal aluA,aluB,aluRes : std_logic_vector(31 downto 0);
282 signal branch, branchI, zero : std_logic := '0';
283 signal braAddr : std_logic_vector(15 downto 0);
284 signal op_code, func : std_logic_vector(5 downto 0);
285
286 begin
287   IM : entity work.dataMem port map(im_data,im_addr,
      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"FFFFFFFF";
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"10001100000000010000000000000001";
59    wait for 2 ns;
60    -- sw $1, 6($zero)
61    extPC <= x"00000002";
62    IMdata <= b"10101100000000010000000000000110";
```

```
63      wait for 2 ns;
64      -- lw $2, 2($zero)
65      extPC <= x"00000003";
66      IMdata <= b"10001100000000100000000000000010";
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"00000000100000100100000000110000";
75      wait for 2 ns;
76      -- beq $1, $2, 100
77      extPC <= x"00000006";
78      IMdata <= b"00010000001000100000000001100100";
79      wait for 2 ns;
80
81      -- lw $2, 4($zero)
82      extPC <= x"00000007";
83      IMdata <= b"10001100000000100000000000000100";
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"00010000001000011111111111110101";
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;
```