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 5$^{\text{th}}$, 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.

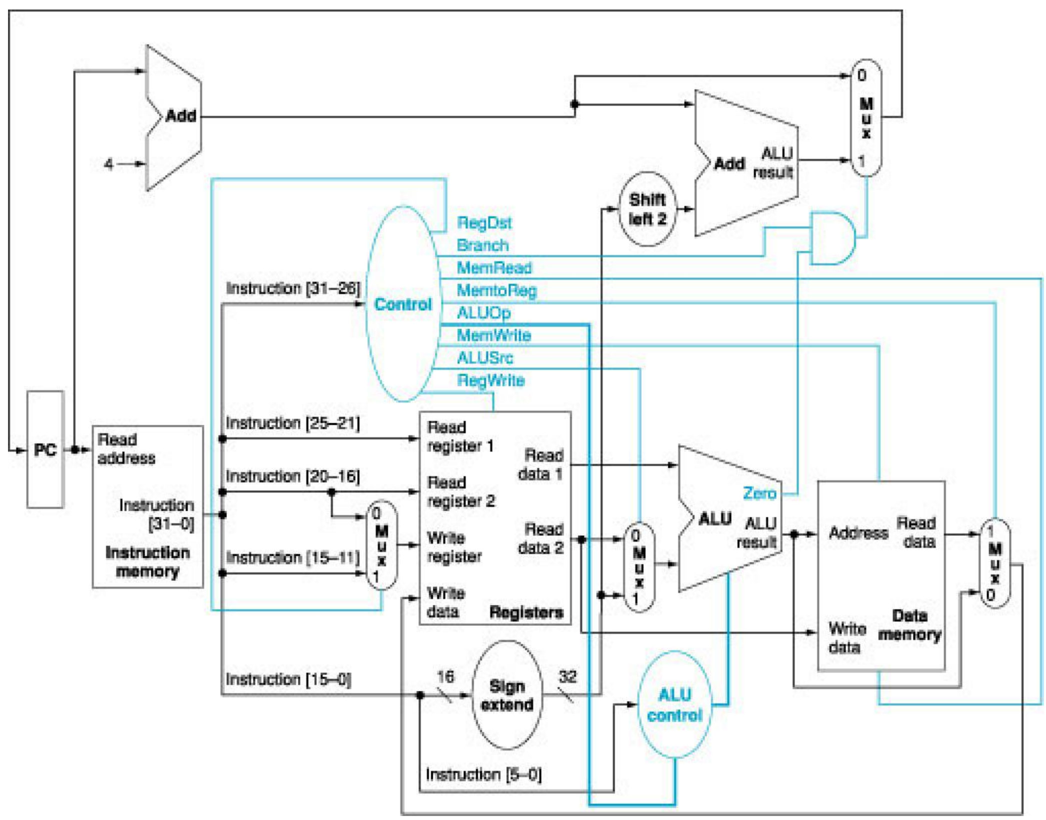

Figure 1: Datapath Overview

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 $5, $1, $2`
000010_2	`--`	`andi`	`andi $6, $2, 00FF`
000000_2	000010_2	`or`	`or $8, $1, $2`
000011_2	`--`	`ori`	`ori $7, $1, 00FF`

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. 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.

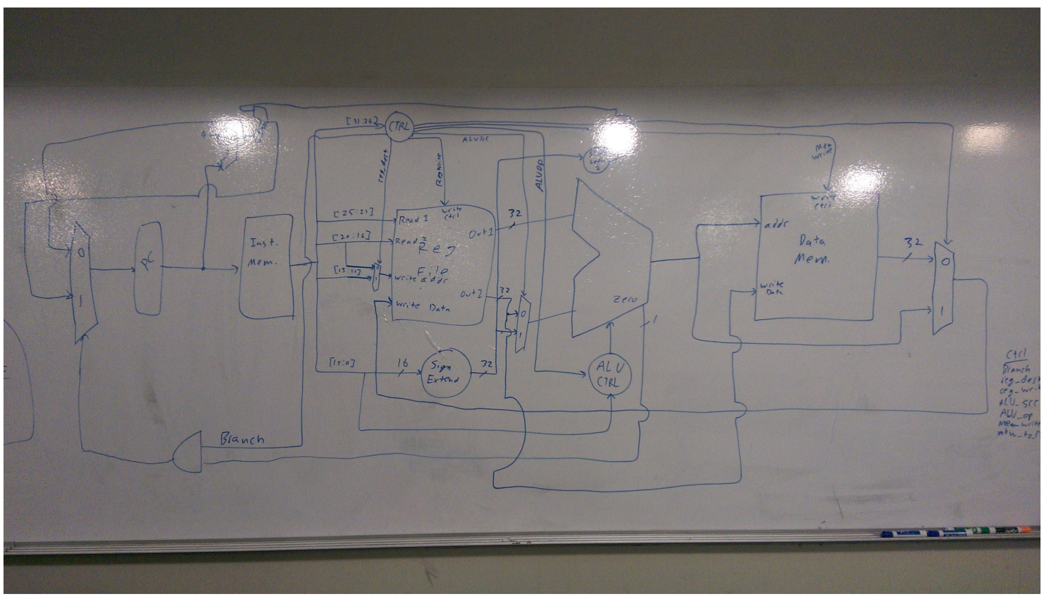

Figure 2: Implemented Data Path Overview

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 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. 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 ensure the functionality it provided was desired. 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 $00000001_{16} \rightarrow 00000005_{16}$ were initialized with starting data. Please refer to the Test Bench 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 $1, 1($zero)`
2. `sw $1, 6($zero)`
3. `lw $2, 2($zero)`
4. `add $3, $1, $2`
5. `sub $4, $2, $1`
6. `beq $1, $2, 100`
7. `lw $2, 4($zero)`

8. nand $5, $1, $2

9. andi $6, $2, 0x00FF

10. ori $7, $1, 0x00FF

11. or $8, $1, $2

12. beq $1, $1, -0x000B

Figure 3 shows the first instruction being executed. The data memory at address 0x00000001 holds the value 0xAAAAAAAA and register $1 is subsequently loaded with the data 0xAAAAAAAA.

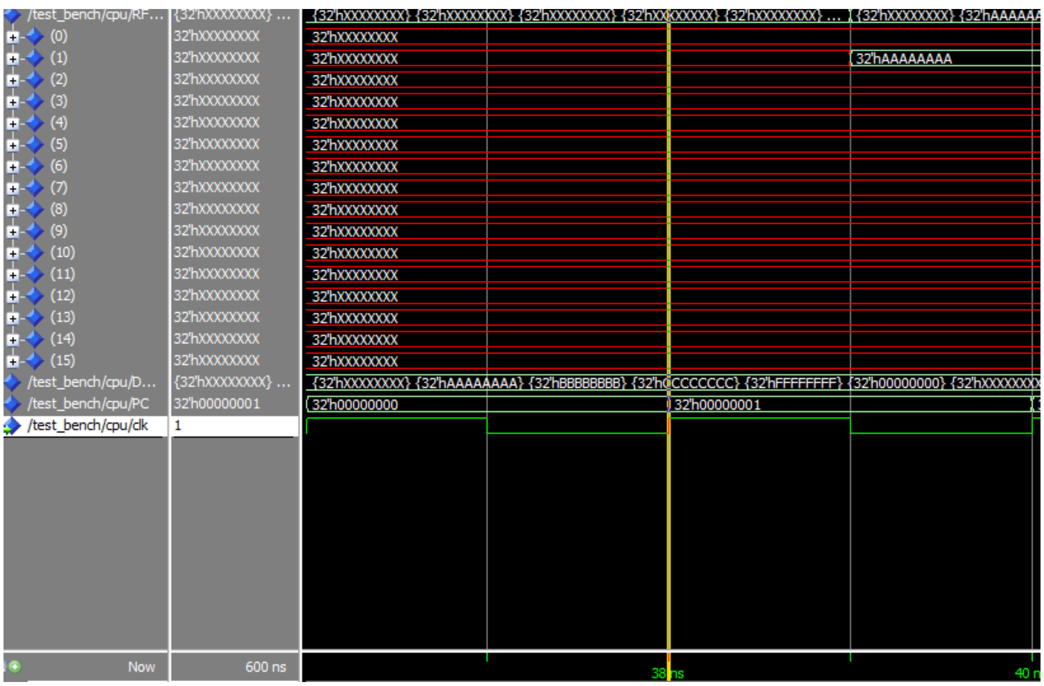

Figure 3: `lw $1, 1($zero)`

Figure 4 shows the second instruction. The value of 0xAAAAAAAA in register $1 is successfully stored into the data memory at address 0x00000006.

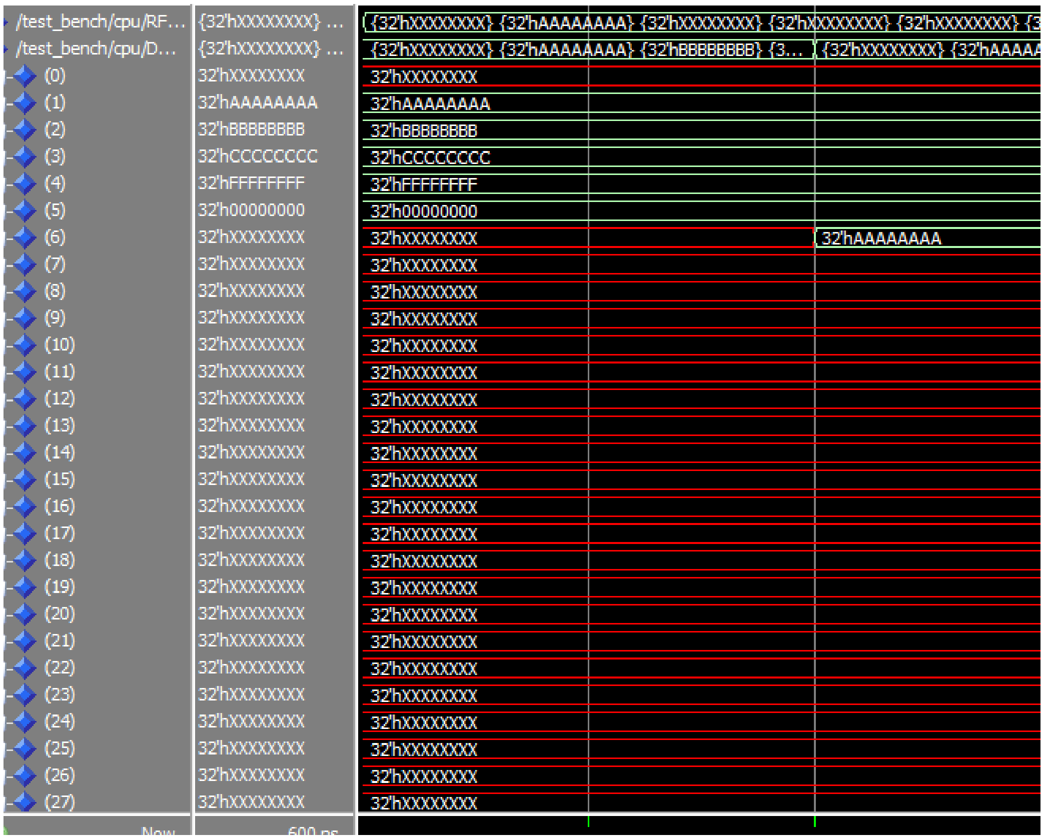

Figure 4: `sw $1, 6($zero)`

Figure 5 shows the third instruction. The value of 0xBBBBBBBB at address 0x00000002 is successfully loaded into register $2.

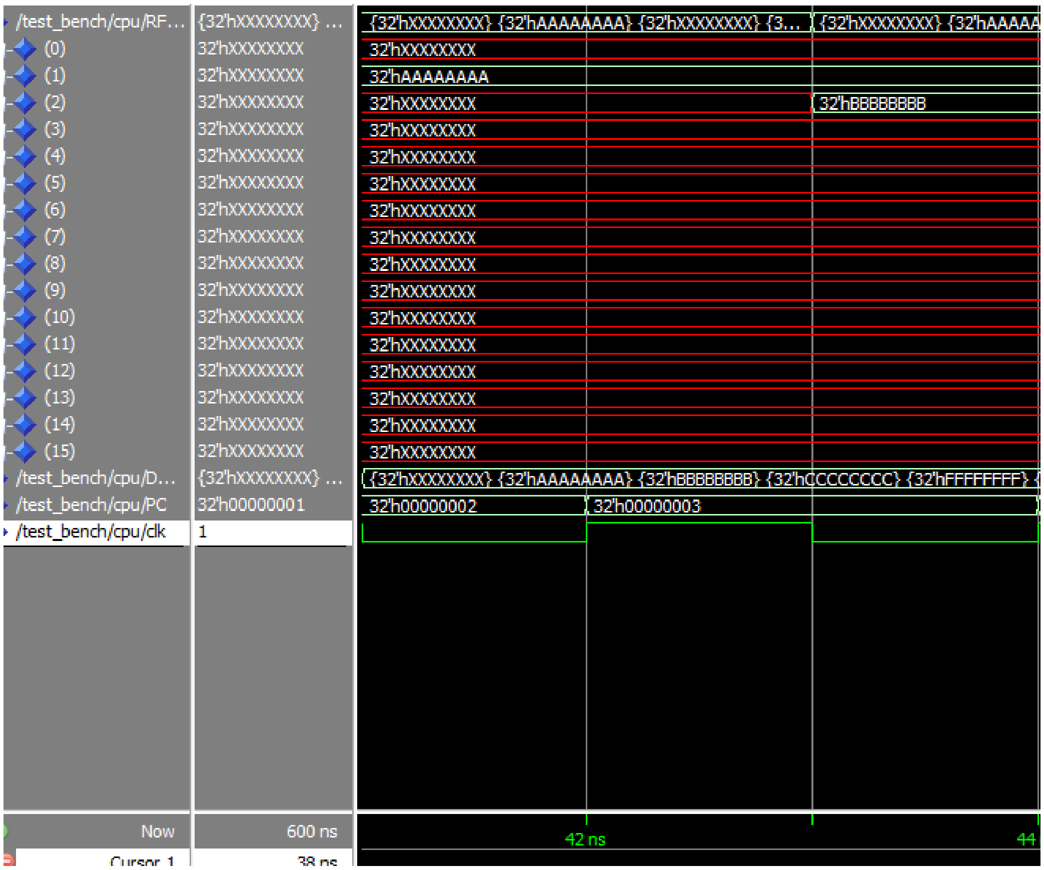

Figure 5: `lw $2, 2($zero)`

Figure 6 shows the fourth instruction. The values of 0xAAAAAAAA and 0xBBBBBBBB are successfully added with the result being 0x66666665 and writing this value into register $3.

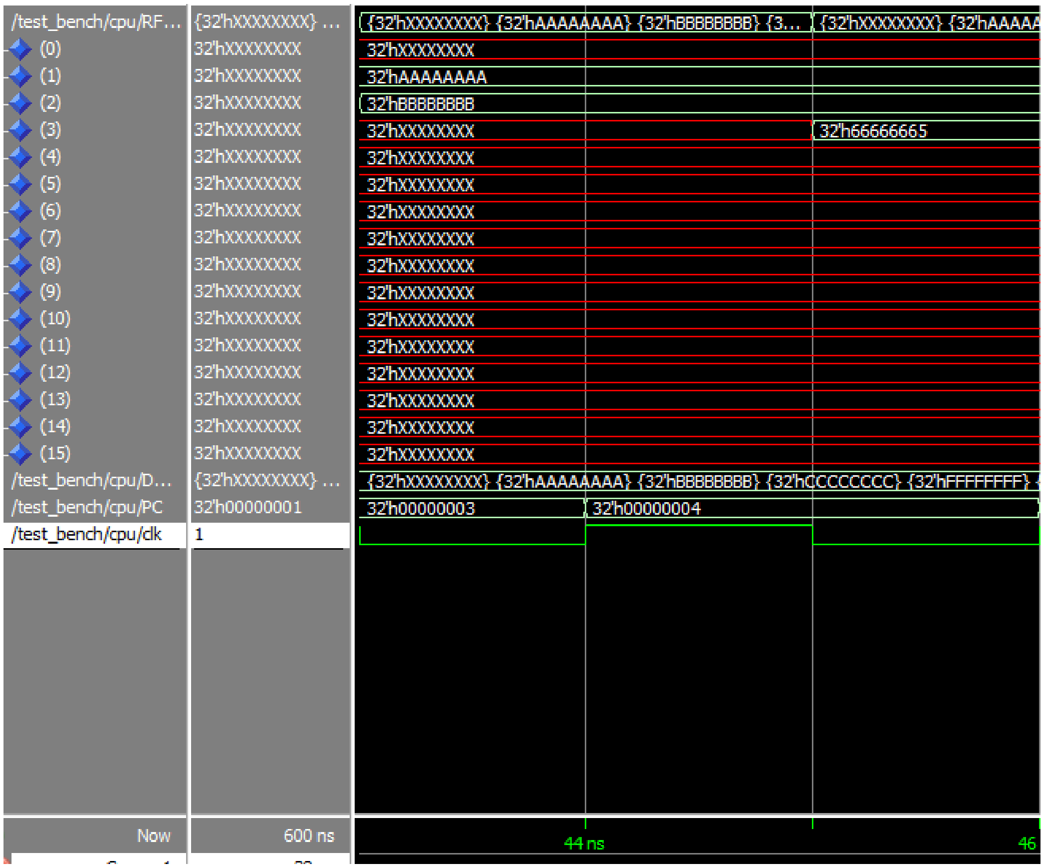

Figure 6: `add $3, $1, $2`

Figure 7 shows the fifth instruction. The value of 0xAAAAAAAA in register \$1 is subtracted from register \$2. The result is 0x11111111 and is written back into register \$4.

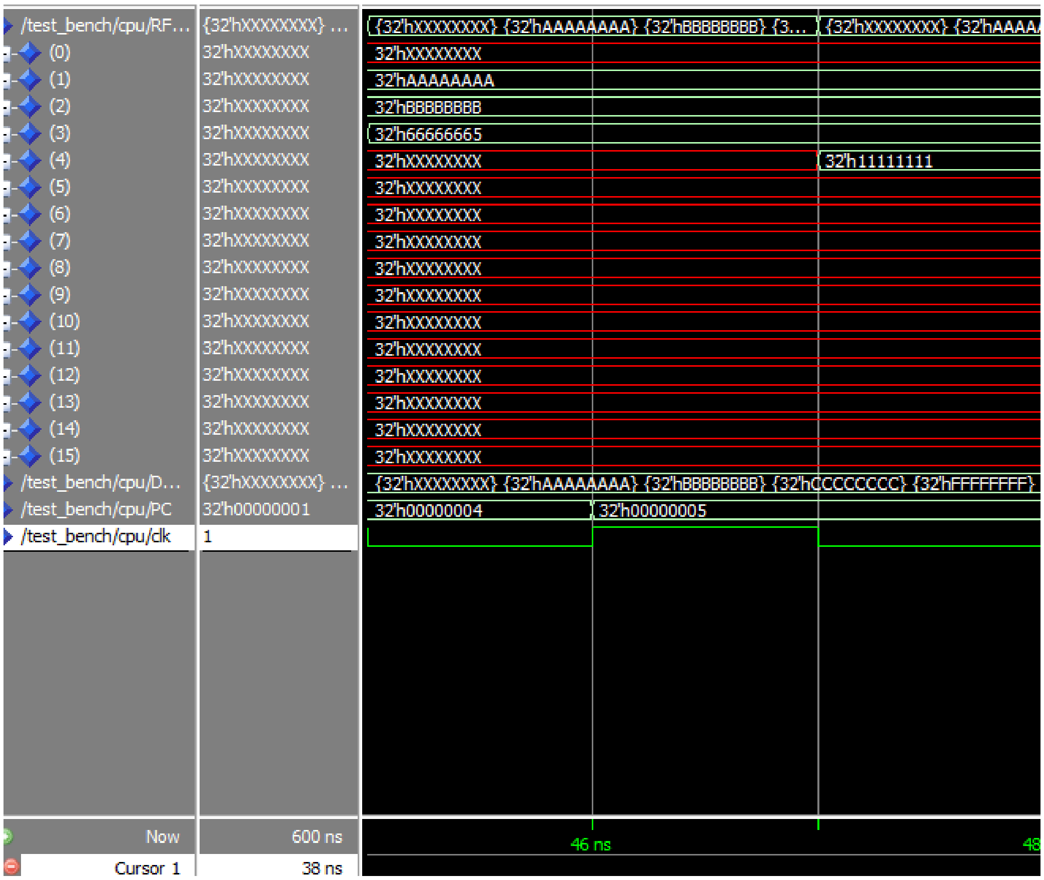

Figure 7: `sub $4, $2, $1`

Figure 8 shows the sixth instruction. The value of 0xBBBBBBBB in register \$2 is compared to 0xAAAAAAAA in register \$1 for equivalence. Since they are not, the program counter increments to the next instruction.

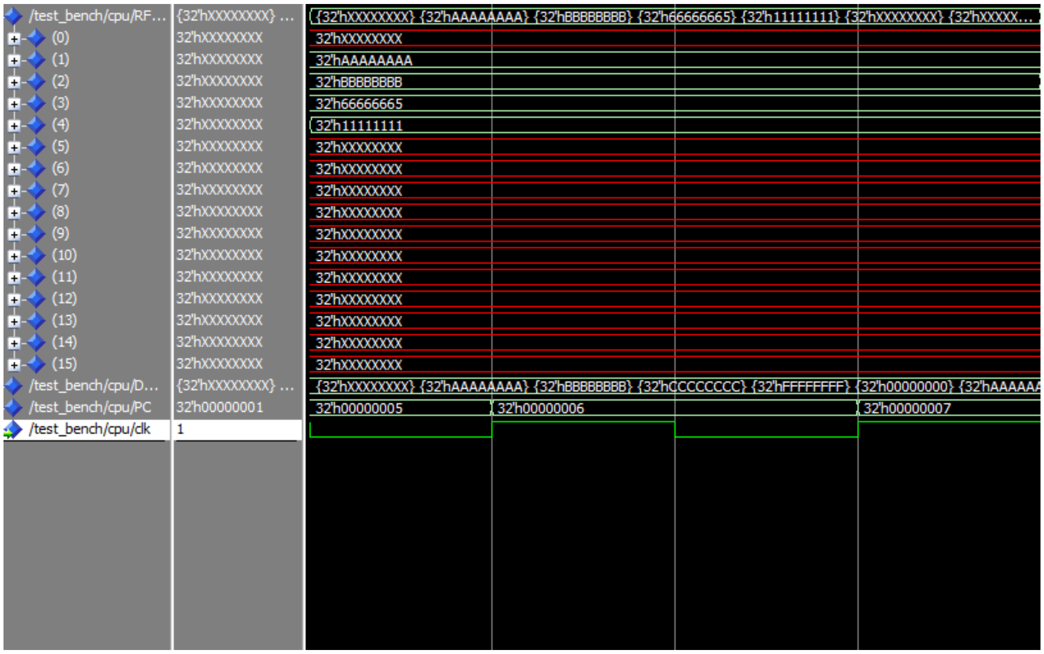

Figure 8: `beq $1, $2, 100`

Figure 9 shows the seventh instruction. The value of 0xFFFFFFFF at address 0x00000004 is loaded into register $2 successfully.

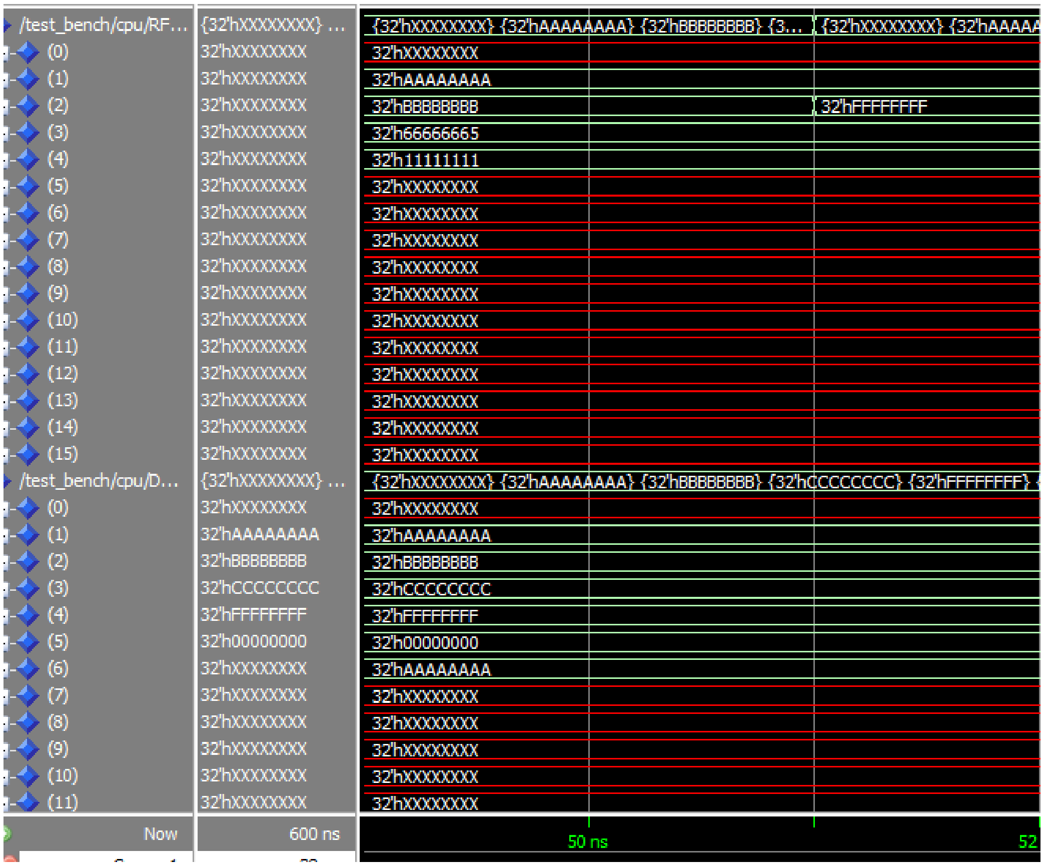

Figure 9: `lw $2, 4($zero)`

Figure 10 shows the eight instruction. The value of 0xFFFFFFFF in register \$2 and 0xAAAAAAAA in register are nanded and the result of 0x55555555 is both accurate and stored in register \$5.

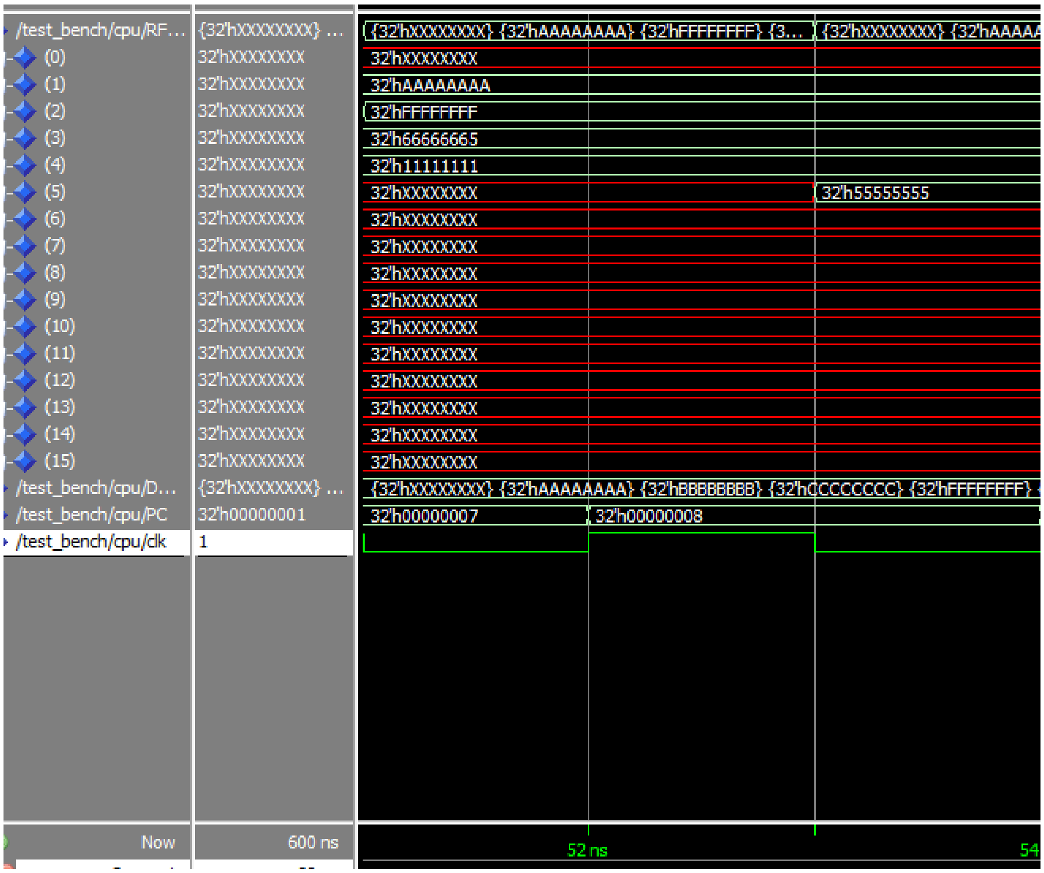

Figure 10: `nand $5, $1, $2`

Figure 11 shows the ninth instruction. The immediate value of 0x000000FF is anded with the value of 0xFFFFFFFF in register \$2. The correct result of 0x000000FF is calculated and stored in register \$6.

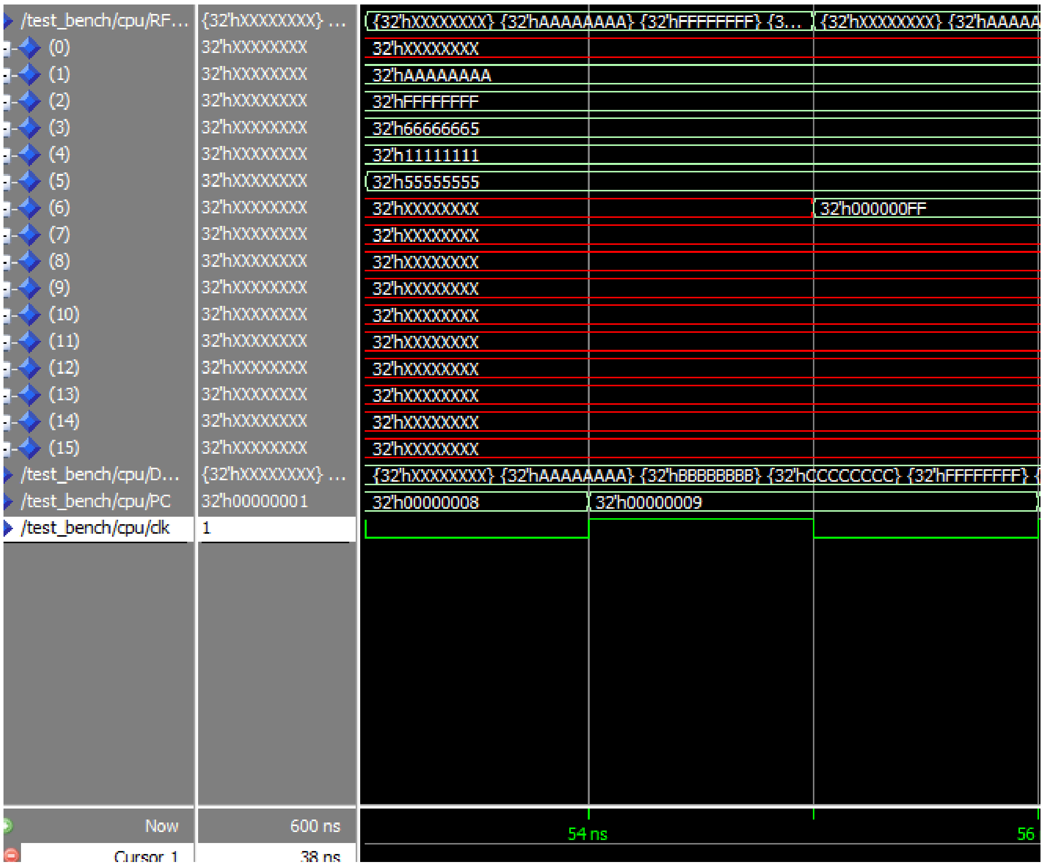

Figure 11: `andi $6, $2, 0x00FF`

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

```
library ieee;
```

```vhdl
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity regFile is
  port(
  regA : out std_logic_vector(31 downto 0);
  regB : out std_logic_vector(31 downto 0);
  selA : in std_logic_vector(3 downto 0);
  selB : in std_logic_vector(3 downto 0);
  wData : in std_logic_vector(31 downto 0);
  registerWrite : in std_logic;
  selW : in std_logic_vector(3 downto 0);
  clk : in std_logic);
end regFile;

architecture behavioral of regFile is
type reg_arr is array(0 to 15) of std_logic_vector(31
    downto 0);
signal rData : reg_arr;
begin
  with selA
    select regA <= x"00000000" when b"0000",
    rData(to_integer(unsigned(selA))) when others;
  with selB
    select regB <= x"00000000" when b"0000",
    rData(to_integer(unsigned(selB))) when others;

  wrProc: process(clk) is
  begin
    if falling_edge(clk) then
    if(registerWrite = '1') then
      rData(to_integer(unsigned(selW))) <= wData;
    end if;
    end if;
  end process;
end behavioral;

---

library ieee;
```

```vhdl
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity control is
  port(
    inst_in : in std_logic_vector(5 downto 0);
    func : in std_logic_vector(5 downto 0);
    stall : in std_logic;
    branch : out std_logic;
    reg_dest : out std_logic;
    reg_write : out std_logic;
    ALU_src : out std_logic;
    ALU_op : out std_logic_vector(2 downto 0);
    mem_write : out std_logic;
    mem_to_reg : out std_logic
  );
end control;

architecture behavioral of control is
  signal branch_o, reg_dest_o, reg_write_o, ALU_src_o, mem_write_o, mem_to_reg_o : std_logic;
  signal ALU_op_o : std_logic_vector(2 downto 0);
  signal branch_f, reg_dest_f, reg_write_f, ALU_src_f, mem_write_f, mem_to_reg_f : std_logic;
  signal ALU_op_f : std_logic_vector(2 downto 0);
begin
  -- set intermediate signals incase of r-type instruction
  with func select
    branch_f <= '0' when "100000",  --add
      '0' when "110000",  --sub
      '0' when "000001",  --nand
      '0' when "000010",  --or
      '0' when others;
  with func select
    reg_dest_f <= '1' when "100000",  --add
      '1' when "110000",  --sub
      '1' when "000001",  --nand
      '1' when "000010",  --or
      'Z' when others;
```

```vhdl
with func select
  reg_write_f <= '1' when "100000", --add
    '1' when "110000",  --sub
    '1' when "000001",  --nand
    '1' when "000010",  --or
    'Z' when others;
with func select
  ALU_src_f <= '0' when "100000", --add
    '0' when "110000",  --sub
    '0' when "000001",  --nand
    '0' when "000010",  --or
    'Z' when others;
with func select
  ALU_op_f <= "000" when "100000",  --add
    "001" when "110000",  --sub
    "010" when "000001",  --nand
    "100" when "000010",  --or
    "ZZZ" when others;
with func select
  mem_write_f <= '0' when "100000", --add
    '0' when "110000",  --sub
    '0' when "000001",  --nand
    '0' when "000010",  --or
    'Z' when others;
with func select
  mem_to_reg_f <= '1' when "100000",  --add
    '1' when "110000",  --sub
    '1' when "000001",  --nand
    '1' when "000010",  --or
    'Z' when others;

-- set intermediate signals incase of non r-type instruction
with inst_in select
  branch_o <= '0' when "100011",  --lw
    '0' when "101011",  --sw
    '1' when "000100",  --beq
    '0' when "000010",  --andi
    '0' when "000011",  --ori
    '0' when others;
```

```
with inst_in select
  reg_dest_o <= '0' when "100011",  --lw
    '0' when "101011",  --sw
    '0' when "000100",  --beq
    '0' when "000010",  --andi
    '0' when "000011",  --ori
    'Z' when others;
with inst_in select
  reg_write_o <= '1' when "100011", --lw
    '0' when "101011",  --sw
    '0' when "000100",  --beq
    '1' when "000010",  --andi
    '1' when "000011",  --ori
    'Z' when others;
with inst_in select
  ALU_src_o <= '1' when "100011", --lw
    '1' when "101011",  --sw
    '0' when "000100",  --beq
    '1' when "000010",  --andi
    '1' when "000011",  --ori
    'Z' when others;
with inst_in select
  ALU_op_o <= "000" when "100011",  --lw
    "000" when "101011",  --sw
    "001" when "000100",  --beq
    "011" when "000010",  --andi
    "100" when "000011",  --ori
    "ZZZ" when others;
with inst_in select
  mem_write_o <= '0' when "100011", --lw
    '1' when "101011",  --sw
    '0' when "000100",  --beq
    '0' when "000010",  --andi
    '0' when "000011",  --ori
    'Z' when others;
with inst_in select
  mem_to_reg_o <= '0' when "100011",  --lw
    '1' when "101011",  --sw
    '1' when "000100",  --beq
    '1' when "000010",  --andi
```

```
      '1' when "000011",  --ori
      'Z' when others;

  -- select from intermediate signals
  with inst_in select
    branch <= branch_f when "000000",
      branch_o when others;
  with inst_in select
    reg_dest <= reg_dest_f when "000000",
      reg_dest_o when others;
  with inst_in select
    reg_write <= reg_write_f when "000000",
      reg_write_o when others;
  with inst_in select
    ALU_src <= ALU_src_f when "000000",
      ALU_src_o when others;
  with inst_in select
    ALU_op <= ALU_op_f when "000000",
      ALU_op_o when others;
  with inst_in select
    mem_write <= mem_write_f when "000000",
      mem_write_o when others;
  with inst_in select
    mem_to_reg <= mem_to_reg_f when "000000",
      mem_to_reg_o when others;
end behavioral;

---

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dataMem is
  port(
  data : out std_logic_vector(31 downto 0);
  sel : in std_logic_vector(31 downto 0);
  wData : in std_logic_vector(31 downto 0);
  memWrite : in std_logic;
  clk : in std_logic);
```

```
end dataMem;

architecture behavioral of dataMem is
type mem_arr is array(0 to 255) of std_logic_vector(31 downto 0);
signal mData : mem_arr;
begin
  data <= mData(to_integer(resize(unsigned(sel),8)));

  wrProc: process(clk) is
  begin
    if falling_edge(clk) then
    if(memWrite = '1') then
          mData(to_integer(resize(unsigned(sel),8))) <= wData;
    end if;
    end if;
  end process;
end behavioral;



---
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALU is
    port(
    inA : in std_logic_vector(31 downto 0);
    inB : in std_logic_vector(31 downto 0);
    ctl : in std_logic_vector(2 downto 0);
    res : out std_logic_vector(31 downto 0));
end ALU;

architecture behavioral of ALU is
signal add : std_logic_vector(31 downto 0);
signal sub : std_logic_vector(31 downto 0);
signal andres : std_logic_vector(31 downto 0);
signal nandres : std_logic_vector(31 downto 0);
```

```
signal orres : std_logic_vector(31 downto 0);
    begin
    add <= std_logic_vector(signed(inA)+signed(inB));
    sub <= std_logic_vector(signed(inA)-signed(inB));
    andres <= std_logic_vector(unsigned(inA) and unsigned(inB));
    nandres <= std_logic_vector(not(unsigned(inA) and unsigned(inB)));
    orres <= std_logic_vector(unsigned(inA) or unsigned(inB));

  -- Multiplexer
  with ctl select
    res <= add when "000",
      sub when "001",
      nandres when "010",
      andres when "011",
      orres when "100",
      "00000000000000000000000000000000" when others;
end behavioral;

--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity processor is
    port(
    extPC : in std_logic_vector(31 downto 0);
    IMdata : in std_logic_vector(31 downto 0);
    DMdata : in std_logic_vector(31 downto 0);
    IMwrite : in std_logic;
    DMwrite : in std_logic;
    DMaddr : in std_logic_vector(31 downto 0);
    stall : in std_logic;
    clk : in std_logic
);
end processor;

architecture behavioral of processor is
```

```
signal im_wrEn,im_clk : std_logic;
signal im_data,im_addr,im_wData : std_logic_vector(31 downto 0);
signal dm_wrEn,dm_clk : std_logic;
signal dm_data,dm_addr,dm_wData : std_logic_vector(31 downto 0);
signal PC : std_logic_vector(31 downto 0);
signal regA,regB,wData : std_logic_vector(31 downto 0);
signal selA,selB,selW : std_logic_vector(3 downto 0);
signal aluCtl : std_logic_vector(2 downto 0);
signal regWrite, regDest, regClk, dm_write, aluSrc, memtoreg : std_logic;
signal aluA,aluB,aluRes : std_logic_vector(31 downto 0);
signal branch, branchI, zero : std_logic := '0';
signal braAddr : std_logic_vector(15 downto 0);
signal op_code, func : std_logic_vector(5 downto 0);

begin
  IM : entity work.dataMem port map(im_data,im_addr,im_wData,im_wrEn,im_clk);
  DM : entity work.dataMem port map(dm_data,dm_addr,dm_wData,dm_wrEn,dm_clk);
    RF : entity work.regFile port map(regA,regB,selA,selB,wData,regWrite,selW,regClk);
    ALU : entity work.ALU port map(aluA, aluB, aluCtl, aluRes);
  CTRL : entity work.control port map(op_code, func, stall, branchI, regDest, regWrite, aluSrc, aluCtl, dm_write, memtoreg);

    --all clocks synced
    im_clk <= clk;
    dm_clk <= clk;
    regClk <= clk;

    im_wData <= IMData;
    im_wrEn <= IMWrite;
    --allow testbench to initialize
    process(clk)
    begin
```

```
    if(rising_edge(clk)) then
        if(stall = '1') then
            PC <= extPC;
        elsif(branch = '1') then
            PC <= std_logic_vector(unsigned(PC) + (unsigned(resize(signed(braAddr), 32))));
        else
            PC <= std_logic_vector(unsigned(PC) + x"1");
        end if;
    end if;
    end process;
    braAddr <= im_data(15 downto 0);
    im_addr <= PC;
    aluA <= regA;
    with aluSrc
        select aluB <= regB when '0',
        std_logic_vector(unsigned(resize(signed(im_data(15 downto 0)), 32))) when '1',
        x"00000000" when others;
    with regDest
        select selW <= im_data(19 downto 16) when '0',
        im_data(14 downto 11) when '1',
        "ZZZZ" when others;
    with stall
        select dm_addr <=  DMaddr when '1',
        aluRes when others;
    with stall
        select dm_wData <= DMdata when '1',
        regB when others;
    with stall
        select dm_wrEn <= DMWrite when '1',
    dm_write when others;

    with memtoreg
        select wData <= aluRes when '1',
        dm_data when others;

    with aluRes
        select zero <= '1' when x"00000000",
        '0' when others;
```

```
    branch <= branchI and zero;

    op_code <= im_data(31 downto 26);
    func <= im_data(5 downto 0);

    selA <= im_data(24 downto 21);
    selB <= im_data(19 downto 16);
end behavioral;
```

Listing 2: Test Bench Code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity test_bench is
end test_bench;

architecture behavioral of test_bench is
  signal clk : std_logic;
  signal extPC, IMdata, DMdata,DMaddr : std_logic_vector
   (31 downto 0) := x"00000000";

  signal IMwrite, DMwrite, stall : std_logic := '0';
begin
  cpu : entity work.processor port map(extPC, IMdata,
   DMdata, IMwrite, DMwrite, DMaddr, stall, clk);

  -- clk process
  clkgen: process
  begin
    clk <= '1';
    wait for 1 ns;
    clk <= '0';
    wait for 1 ns;
  end process;

  tester: process
  begin
    -- init values
```

```
stall <= '1';
        IMwrite <= '0';
        DMwrite <= '1';
-- put some data into the DM
DMdata <= x"AAAAAAAA";
DMwrite <= '1';
DMaddr <= x"00000001";
wait for 2 ns;
DMdata <= x"BBBBBBBB";
DMwrite <= '1';
DMaddr <= x"00000002";
wait for 2 ns;
DMdata <= x"CCCCCCCC";
DMwrite <= '1';
DMaddr <= x"00000003";
wait for 2 ns;
DMdata <= x"FFFFFFFF";
DMwrite <= '1';
DMaddr <= x"00000004";
wait for 2 ns;
DMdata <= x"00000000";
DMwrite <= '1';
DMaddr <= x"00000005";
wait for 2 ns;

-- Now load program, start from address 1
DMwrite <= '0';
IMwrite <= '1';
-- lw $1, 1($zero)
extPC <= x"00000001";
IMdata <= b"10001100000000010000000000000001";
wait for 2 ns;
-- sw $1, 6($zero)
extPC <= x"00000002";
IMdata <= b"10101100000000010000000000000110";
wait for 2 ns;
-- lw $2, 2($zero)
extPC <= x"00000003";
IMdata <= b"10001100000000100000000000000010";
wait for 2 ns;
```

```
    -- add $3, $1, $2
    extPC <= x"00000004";
    IMdata <= b"00000000001000100001100000100000";
    wait for 2 ns;
    -- sub $4, $2, $1
    extPC <= x"00000005";
    IMdata <= b"00000000010000010010000000110000";
    wait for 2 ns;
    -- beq $1, $2, 100
    extPC <= x"00000006";
    IMdata <= b"00010000001000100000000001100100";
    wait for 2 ns;

    -- lw $2, 4($zero)
    extPC <= x"00000007";
    IMdata <= b"10001100000000100000000000000100";
    wait for 2 ns;
    -- nand $5, $1, $2
    extPC <= x"00000008";
    IMdata <= b"00000000001000100010100000000001";
    wait for 2 ns;
    -- andi $6, $2, 00FF
    extPC <= x"00000009";
    IMdata <= b"00001000010001100000000011111111";
    wait for 2 ns;
    -- ori $7, $1, 00FF
    extPC <= x"0000000A";
    IMdata <= b"00001100001001110000000011111111";
    wait for 2 ns;
    -- or $8, $1, $2
    extPC <= x"0000000B";
    IMdata <= b"00000000001000100100000000000010";
    wait for 2 ns;
    -- beq $1, $1 -0x000B
    extPC <= x"0000000C";
    IMdata <= b"00010000001000011111111111110101";
    wait for 2 ns;

-- Begin execution here
    wait for 2 ns;
```

```
      IMwrite <= '0';
      extPC <= x"00000000";
      wait for 2 ns;
      stall <= '0';

--allow enough time for processor to execute instructions
      wait for 100 ns;

   end process;
end behavioral;
```