ENGG*3380: Computer Organization

Mohamad Abou El Nasr, Haleh Vahedi

Final Project: 16-bit CPU

Group 5: Chizorom Anyalechi, Khoa Tran, Eric Cao

March 9, 2026

# Table of Contents

## Problem StatementA

The purpose of the project is to understand, design, implement, and validate a 16-bit CPU. Extending from Lab 5, the project introduces the program counter register, instruction memory instructions, and data memory design enabling implementations for J-type instruction. The completed design serves as the foundation for following courses in the Computer Engineering program.

## Assumptions and Constraints

- Instruction Memory: read_en is always '1' and write_en is always '0' since this memory is only used to read instructions from the text file for the CPU to process.
- The Program Counter (PC) register increments by 2 after each instruction.
    - If rd of bne instruction equals 1, pc = pc + offset + 2
    - For jump instruction, pc = jumpAddress
- Sout is controlled by reg_src from the control unit; output selects data_out of memory data, Sout of 16-bit ALU, and slt_input.
- Some components of VHDL code are provided with missing control signal handling and incomplete module port mapping.
- CPU test bench VHDL are not modified as instructions are provided in the text file.

## System Overview

See Figure A7 in Appendix A for the block diagram of the CPU.

The CPU consists of several components that allow the datapath to read and execute on instructions stored in memory. 11 instructions are implemented: add, subtract, bitwise AND, bitwise OR, add and subtract-immediate, set-on-less than, load and store-word, as well as branch-not-equal and unconditional jump.

### Control

The control unit decodes opcode and output signals relative to the operations. There are 9 output signals.

- alu_op(1:0) selects one of four operations supported by the 16-bit ALU: addition "00", subtraction "01", bitwise OR "10", and bitwise AND "11".
- alu_src controls input B of the 16-bit ALU to either select a value from the register file or an immediate value.
    - When set to '0', c_data (rs) from 16-bit register file is passed as input B. This implies the instruction R-type, as R-type instructions perform operations using two registers.

    - When set to ‘1’, the output of the sign extension module (immOut) is passed as input B. This is used for I-type instructions, where the operand is directly stored in the instruction.
- reg_dest is the selection line that allows the register file to read register rd at instruction(11:8). This is useful for implementing bne and sw, where the destination register must be read rather than written.
    - When set to ‘1’, the register file reads from rd and passes it as c_data. This allows bne to compare rs and rd. It also allows sw instructions to write rd to the memory.
    - In all other cases, reg_dest is set to ‘0’ to read from rt or instruction(3:0). Otherwise, the value from c_data is not used and the value of reg_dest does not matter.
- reg_load enables loading (i.e. storing) data to the register file. The instructions bne, jmp, and sw have reg_load set to ‘0’ as values in the register file do not change.
- reg_src(1:0) indicates the selection line of the 3:1 multiplexer selecting input from data memory, 16-bit ALU, or set-on-less-than module.
    - “00” is reserved for the lw instruction as data from memory is loaded to the register file.
    - “10” is reserved for the slt instruction to load sltinput to the register file.
    - “01” indicates selecting the 16-bit ALU result.
- mem_read enables reading from memory. It is set to ‘1’ for the lw instruction.
- mem_write enables writing to memory. It is set to ‘1’ for the sw instruction.
- branch and jump both handle the 3:1 multiplexer that selects the next value of the PC register. When both branch and jump are set to ‘0’, the PC register functions normally by iterating by 2.
    - With branch set to ‘1’ on a bne instruction, this signal is combined with the zero-detection from the ALU to only take the branch when the two registers are not equal. This new signal then selects the calculated branch target as input to the PC register.
    - With jump set to ‘1’ on a jmp instruction, the signal immediately selects the calculated jump target as the input to the PC register.

### Program Counter

The program counter is stored in the dedicated PC register which iterates by two on every clock cycle, and specific instructions (i.e. bne, jmp) modify the input to the PC register using the 3:1 multiplexer.

- On a bne instruction, the calculation performed is target = pc + offset + 2. The calculation is handled by the BNE calculator module (see Figure A4), which is selected by the control module when the two registers being compared are not equal.
- On a jmp instruction, the 4 most significant bits of the current program counter are concatenated with the 12-bit immediate value to form the jump target. This modification is performed by the JMP calculator (see Figure A5), which the jump signal from the control module selects as the next PC value.

### Register File

The register file was provided to us to use in the design of this CPU. It features a load signal, two address signals, and one data signal as input. There are two output data lines.

- Since every instruction (except jmp) reads from rs, it is directly connected to the register file as b_addr, whose contents are read to b_data.
- For instructions that load to the register file, the address at rd passed into a_addr controls the location to load the content at a_data. This data input is handled by the 3:1 multiplexer controlled by the control signal reg_src, which allows non-ALU data (lw, slt instructions) to be loaded to the register file.
- The address/data line C (c_addr, c_data) is connected to the ALU, but multiplexers controls which register file to read and whether to pass the data to the ALU. This notably facilitates the implementation of the bne and sw instructions.
    - For bne, register rd is compared with rs using the ALU's subtraction operation and zero-detection to determine if the branch is taken.
    - For sw, the contents of register rd (via c_data) are directly connected to the input of the memory module. The data at rd is written to memory using the memory address at rs, offset by the immediate value.

### Sign Extension

The sign extension module takes the 4-bit immediate input and extends the most significant bit to create a 16-bit immediate value: immOut(15:4) = immIn(3).

### ALU

The arithmetic logic unit supports four operations and input B is controlled by a 2:1 multiplexer, selecting either a register or immediate value as a second operand.

- The control signal alu_op dictates the ALU operation.
    - "00": addition
    - "01": subtraction
    - "10": bitwise OR

    - “11”:  bitwise AND
- Input B is controlled by alu_src from the control module.
    - ‘0’ selects c_data ($rs) from 16-bit register file. This is used by R-type instructions.
    - ‘1’ selects the immediate value from the sign extension module. This is used by I-type instructions.
- The zero detection signal is set to ‘1’ when the ALU result is 0.

## Analysis

See Figure D2, Figure D3 from Appendix C for the simulation waveforms.

Notable test cases:

- 0x7841 (slt $r8, $r4, $r1) at 1100ns: Indicates if $r4 < $r1, then $r8 = 0x0001, otherwise $r8 = 0x0000. At that rising edge, $r4 = 0x0003 and $r1 =  0x0001, $r4 > $r1. Therefore, $r8 = 0x0000 clock rising edge
- 0x7841 (slt $r8, $r4, $r1) at 3200ns: Indicates if $r4 < $r1, then $r8 = 0x0001, otherwise $r8 = 0x0000. At that rising edge, $r4 = 0x0000 and $r1 =  0x0001, $r4 > $r1. Therefore, $r8 = 0x0001 at the next clock rising edge
- 0xc352 (sw $r3, 2($r5)) at 400ns: Indicates storing data from register $r3 to memory[$r5 + 2]. The $r3 = 0x0008, $r5 = 0x000c, then the data is stored in memory[12+2] = memory[14]. First byte of $r3 (0x08) stored into memory[14], second byte of $r3 (0x00) stored into memory[15] at the next clock rising edge.
- 0x8236 (lw $r2, 6($r3)) at 600ns: Indicates loading data from memory[$r3 + 6] to $r2. $r3 = 0x0008. Therefore memory[$r3 + 6] = memory[8+6] = memory[14] = 0x08. Load memory[14] to $r2, therefore $r2 = 0x0008 at the next clock rising edge.
- 0x9802 (bne $r8, $r0, 2) at 1900ns: Indicates if $r8 != $r0, jumps to label, in this case jumps to pc + 2 + offset. $r8 = 0, $r0 = 0, $r8 = $r0. Therefore, next_pc = pc + 2. next_pc = 0x100a + 2 = 0x100c.
- 0x9802 (bne $r8, $r0, 2) at 3300ns: Indicates if $r8 != $r0, jumps to label, in this case jumps to pc + 2 + offset. $r8 = 1, $r0 = 0, $r8 != $r0. Therefore, next_pc = pc + 2 + offset (offset = 2). next_pc = 0x001a + 2 + 2 = 0x001e.
- 0xb00e (jump 0x00e) at 500ns: Indicates unconditional jump to label in this case program count at 0x00e. next_pc = 12-bit immediate. Before executing jump 0x00e, pc = 0x000a. After jump 0x00e is executed, pc = 0x000e.

## Errors and Issues

Several errors regarding the control unit were found and resolved before demonstrations were held. As seen in figure A1, erroneously setting the reg_dest signal resulted in choosing the wrong register for load-word, store-word, and branch-not-equal instructions. This was first noticed when a bne instruction was provided the wrong register to compare; instead of comparing rs with rd, the ALU compared rs with rt and so the branch condition failed. Similarly, the store-word instruction used the wrong memory address after receiving bad data from rt instead of an immediate as an offset value. These issues were identified and fixed during the lab period.

## Summary and Conclusion

The objective of this lab was to design a CPU that can load eleven types of instructions to memory, entering the design using VHDL and simulating its behaviour in Vivado Design Suite. The CPU uses a specific machine code format (op, rd, rs, rt/imm) to control the datapath, with the registers and memory positions modified as seen using the simulation tools by Vivado Design Suite.

Many components were provided from previous labs (e.g. register file, sign extend, ALU) and a select few were provided to us for the purposes of this lab (i.e. memory). Some components were created for the purposes of this lab, most notably the JMP and BNE target calculators to implement the instructions jmp and bne.

Lab periods allowed us to verify and demonstrate the functionality of the design to our peers and instructors. One such issue was the presence of off-by-one errors in the control module, which lead to logical errors in the simulation of our design. The lab period allowed us to catch the errors made during preparation.

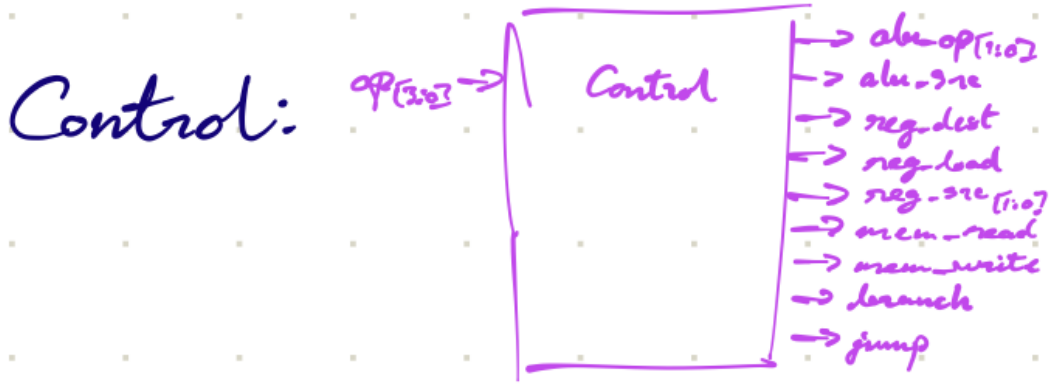
This cumulative report documents the knowledge gained and applied to create the 16-bit CPU according to an instruction standard, generating the desired behaviour for any of the eleven instructions loaded as machine code.

With the verification, demonstration, and reporting of our CPU design complete, we can definitively say that this project was a resounding success.

# Appendix

## Appendix A: Block Diagrams

**Figure A1.** Truth Table and Diagram for Control




| | (add) 0 | (sub) 1 | (and) 2 | (or) 3 | (addi) 4 | (subi) 5 | (slt) 7 | (lw) 8 | (bne) 9 | (jump) B | (sw) C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| alu_op | 0 | 1 | 2 | 3 | 0 | 1 | 1 | 0 | 1 | X | 0 |
| alu_src | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | X | 1 |
| reg_dest | 0 | 0 | 0 | 0 | X | X | X | X | (1) | X | (1) |
| reg_load | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| reg_src | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 0 | X | X | X |
| mem_read | X | X | X | X | X | X | X | 1 | X | X | X |
| mem_write | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| branch | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| jump | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**Figure A2.** Diagram for PC Register

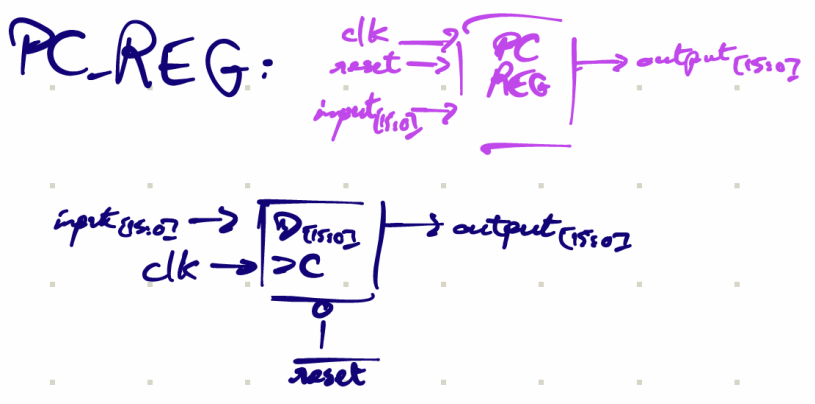



**Figure A3.** Diagram for Jump Instruction Calculator

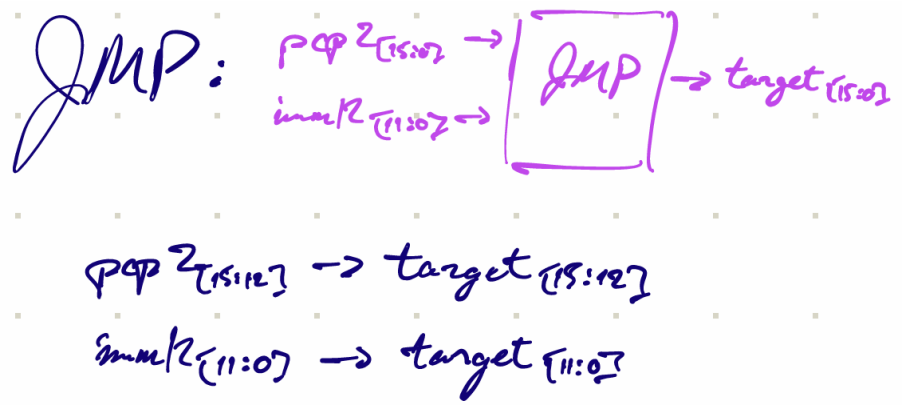



**Figure A4.** Diagram for Branch-Not-Equal Instruction Calculator

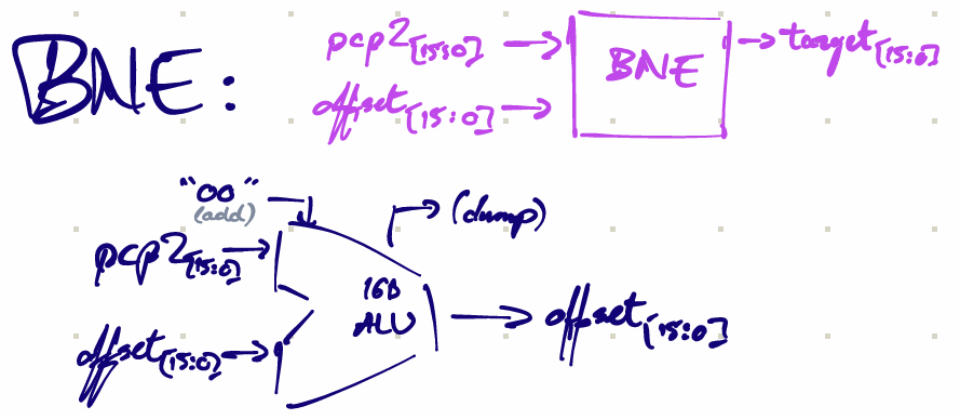


**Figure A5.** Diagram for Sign Extension

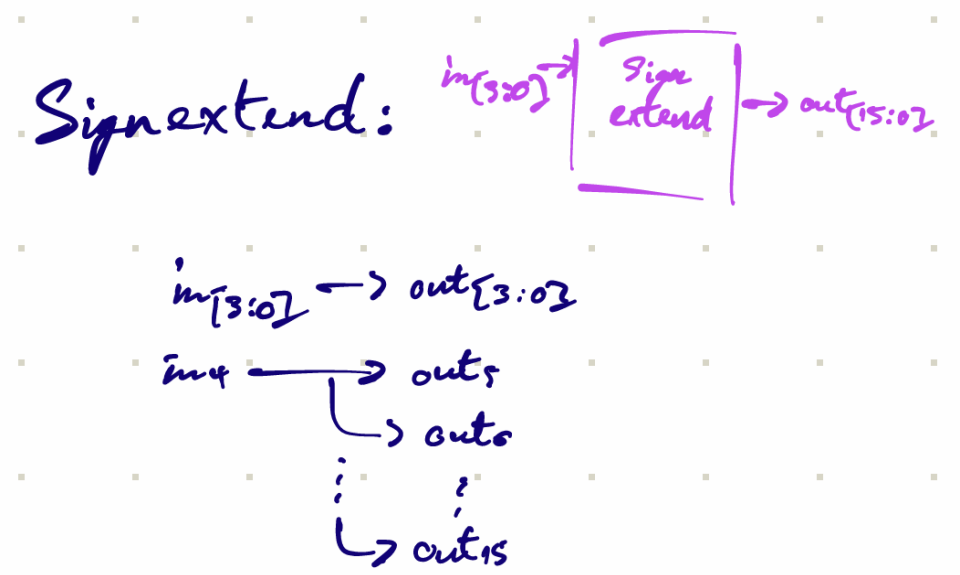



**Figure A6.** Block Diagram for 16-Bit ALU with Zero Detection

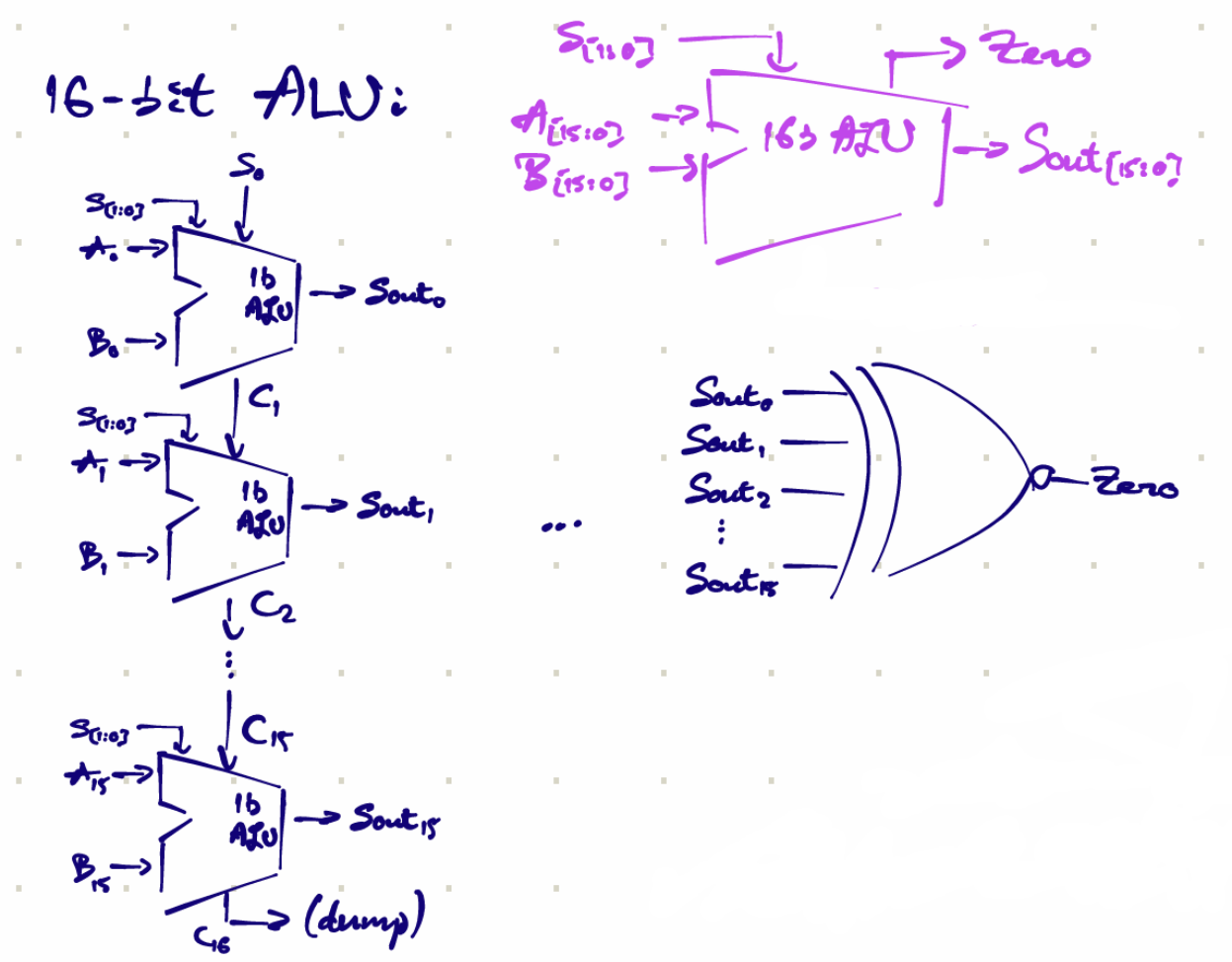
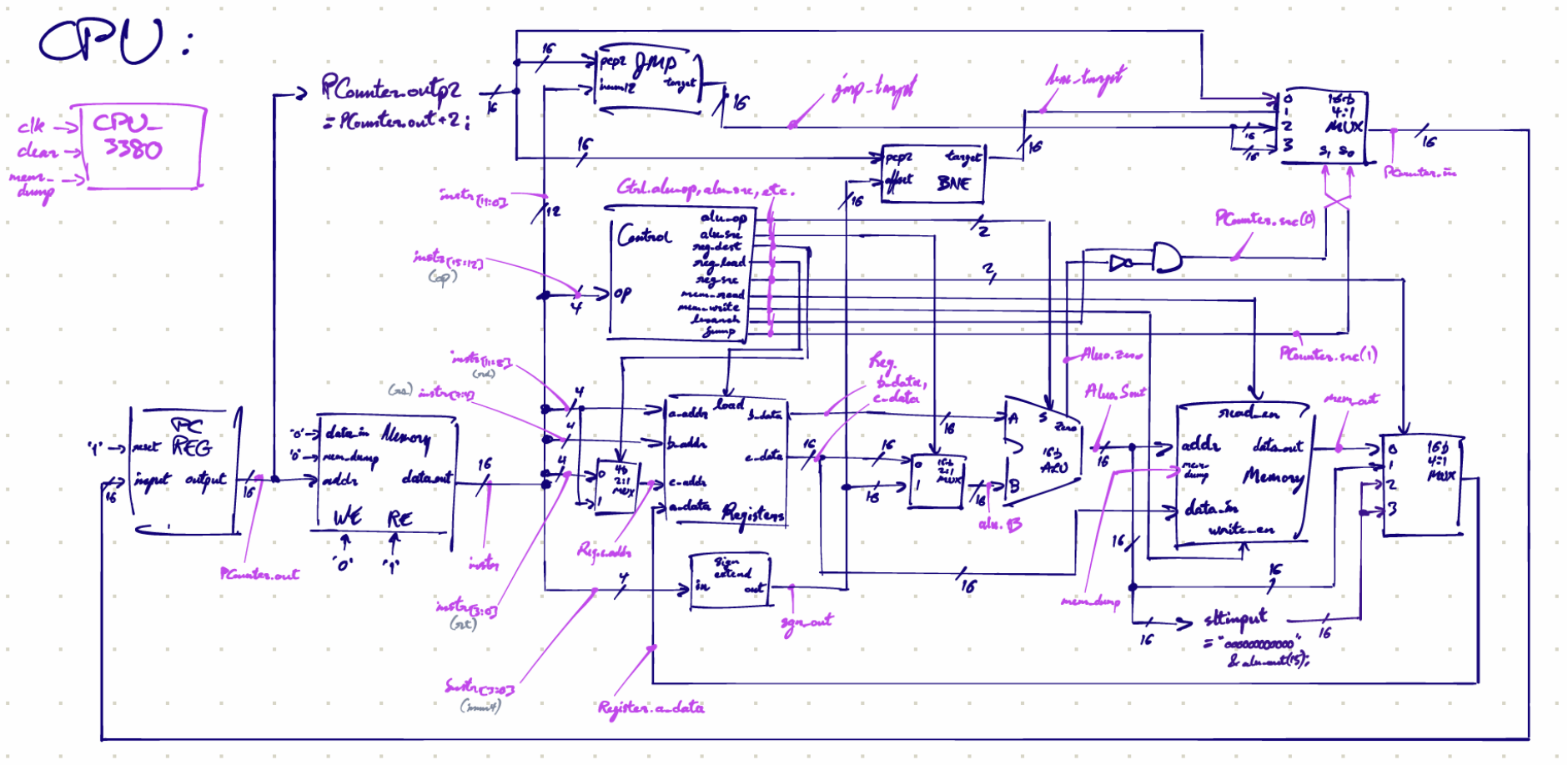


**Figure A7.** Block Diagram for CPU



## Appendix B: VHDL Code for New Modules

**Figure B1.** VHDL Code for Control

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

entity Control is
    port (
        op : in std_logic_vector(3 downto 0);
        alu_op : out std_logic_vector(1 downto 0);
        alu_src : out std_logic;
        reg_dest, reg_load : out std_logic;
        reg_src : out std_logic_vector(1 downto 0);
        mem_read, mem_write : out std_logic;
        branch, jump : out std_logic
    );
end Control;

architecture behav of Control is
begin
    alu_op <= "00" when op = x"0" else
                "01" when (op = x"1" or op = x"5" or op = x"7" or op = x"9") else
                "10" when op = x"2" else
                "11" when op = x"3" else
                "00";
    alu_src <= '1' when (op = x"4" or op = x"5" or op = x"8" or op = x"C") else '0';

    reg_dest <= '1' when (op = x"C" or op = x"9") else '0';
    reg_load <= '0' when (op = x"9" or op = x"B" or op = x"C") else '1';
    reg_src <= "00" when op = x"8" else
                "10" when op = x"7" else
                "01";

    mem_read <= '1' when op = x"8" else '0';
    mem_write <= '1' when op = x"C" else '0';

    branch <= '1' when op = x"9" else '0';
    jump <= '1' when op = x"B" else '0';
end behav;
```

**Figure B2.** VHDL Code for PC Register

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity PC_REG is
       port(
              clk, reset : in std_logic;
              input : in std_logic_vector(15 downto 0);
              output : out std_logic_vector(15 downto 0)
       );
end PC_REG;

architecture Behavioral of PC_REG is
begin
       process(reset, clk)
       begin
          if (reset = '0') then
              output <= (others => '0');
          elsif (rising_edge(clk)) then
              output <= input;
        end if;
       end process;
end Behavioral;
```

**Figure B3.** VHDL Code for Jump Instruction Calculator

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

entity JMP is
    port (
        pcp2 : in std_logic_vector(15 downto 0);
        imm12 : in std_logic_vector(11 downto 0);
        target : out std_logic_vector(15 downto 0)
    );
end JMP;

architecture behav of JMP is
begin
    target(15 downto 12) <= pcp2(15 downto 12);
    target(11 downto 0) <= imm12;
end behav;
```

**Figure B4.** VHDL Code for Branch-Not-Equal Calculator

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

entity BNE is
    port (
        pcp2, offset : in std_logic_vector(15 downto 0);
        target : out std_logic_vector(15 downto 0)
    );
end BNE;

architecture behav of BNE is

component ALU_16b is
    port (
        A, B : in std_logic_vector(15 downto 0);
        S : in std_logic_vector(1 downto 0);
        Sout : out std_logic_vector(15 downto 0);
        Zero : out std_logic
    );
end component;
signal Zero : std_logic;

begin
    alui: ALU_16b port map (
        A => pcp2,
        B => offset,
        S => "00",
        Sout => target,
        Zero => Zero
    );
end behav;
```

**Figure B5.** VHDL Code for Sign Extension

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

entity SignExtend is
    generic (
        in_width : integer := 4;
        out_width : integer := 16
    );
    port (
        input : in std_logic_vector(in_width-1 downto 0);
        output : out std_logic_vector(out_width-1 downto 0)
```

```
    );
end SignExtend;

architecture behav of SignExtend is
begin
    output(in_width-1 downto 0) <= input;
    output(out_width-1 downto in_width) <= (others => input(in_width-1));
end behav;
```

**Figure B6.** VHDL Code for 16-Bit ALU with Zero Detection

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ALU_16b is
    port (
        A, B : in std_logic_vector(15 downto 0);
        S : in std_logic_vector(1 downto 0);
        Sout : out std_logic_vector(15 downto 0);
        Zero : out std_logic
    );
end ALU_16b;

architecture behav of ALU_16b is

component ALU is
    port (
        A, B, Cin : in std_logic;
        S : in std_logic_vector(1 downto 0);
        Sout, Cout : out std_logic
    );
end component;
signal C : std_logic_vector(16 downto 0);
signal So : std_logic_vector(15 downto 0);

begin
    C(0) <= S(0);

    alus: for i in 0 to 15 generate
        alui: ALU port map (
            A => A(i),
            B => B(i),
            S => S,
            Cin => C(i),
            Cout => C(i+1),
            Sout => So(i)
        );
```

```
    end generate;

    Sout <= So;
    Zero <= '1' when So = x"0000" else '0';
end behav;
```

**Figure B7.** VHDL Code for CPU

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity CPU_3380 is
    port (
        clk, clear, mem_dump : in std_logic
    );
end CPU_3380;

architecture behav of CPU_3380 is

signal instr : std_logic_vector(15 downto 0);
signal op, rd, rs, rt : std_logic_vector(3 downto 0);
signal imm12 : std_logic_vector(11 downto 0);
signal imm4 : std_logic_vector(3 downto 0);

component Control is
    port (
        op : in std_logic_vector(3 downto 0);
        alu_op : out std_logic_vector(1 downto 0);
        alu_src : out std_logic;
        reg_dest, reg_load : out std_logic;
        reg_src : out std_logic_vector(1 downto 0);
        mem_read, mem_write : out std_logic;
        branch, jump : out std_logic
    );
end component;

type ctrl_type is record
    alu_op : std_logic_vector(1 downto 0);
    alu_src : std_logic;
    reg_dest, reg_load : std_logic;
    reg_src : std_logic_vector(1 downto 0);
    mem_read, mem_write : std_logic;
    branch, jump : std_logic;
end record ctrl_type;
```

```
signal ctrl : ctrl_type := (
    alu_op => "00",
    reg_src => "00",
    others => '0'
);

component ALU_16b is
    port (
        A, B : in std_logic_vector(15 downto 0);
        S : in std_logic_vector(1 downto 0);
        Sout : out std_logic_vector(15 downto 0);
        Zero : out std_logic
    );
end component;

type alur_type is record
    B, Sout : std_logic_vector(15 downto 0);
    Zero : std_logic;
end record alur_type;

signal alur : alur_type := (
    B => (others => '0'),
    Sout => (others => '0'),
    Zero => '0'
);

component BNE is
    port (
        pcp2, offset : in std_logic_vector(15 downto 0);
        target : out std_logic_vector(15 downto 0)
    );
end component;
signal bne_target : std_logic_vector(15 downto 0);

component JMP is
    port (
        pcp2 : in std_logic_vector(15 downto 0);
        imm12 : in std_logic_vector(11 downto 0);
        target : out std_logic_vector(15 downto 0)
    );
end component;
signal jmp_target : std_logic_vector(15 downto 0);

component SignExtend is
    generic (
        in_width : integer := 3;
        out_width : integer := 16
```

```
    );
    port (
        input : in std_logic_vector(in_width-1 downto 0);
        output : out std_logic_vector(out_width-1 downto 0)
    );
end component;
signal sgn_out : std_logic_vector(15 downto 0);

component Memory is
    generic (
        INPUT : string := "in.txt";
        OUTPUT : string := "out.txt"
    );
    port (
        clk : in std_logic;
        read_en : in std_logic;
        write_en : in std_logic;
        addr : in std_logic_vector(15 downto 0);
        data_in : in std_logic_vector(15 downto 0);
        data_out : out std_logic_vector(15 downto 0);
        mem_dump : in std_logic := '0'
    );
end component;
signal mem_out : std_logic_vector(15 downto 0);

component Registers is
    port(
        clk : in std_logic;
        clear : in std_logic;

        a_addr : in std_logic_vector( 3 downto 0);
        a_data : in std_logic_vector(15 downto 0);
        load : in std_logic;

        b_addr : in std_logic_vector( 3 downto 0);
        c_addr : in std_logic_vector( 3 downto 0);

        b_data : out std_logic_vector(15 downto 0);
        c_data : out std_logic_vector(15 downto 0)
    );
end component;

type reg_type is record
    a_data, b_data, c_data : std_logic_vector(15 downto 0);
    c_addr : std_logic_vector(3 downto 0);
end record reg_type;
```

```
signal reg : reg_type := (
    others => (others => '0')
);

component PC_REG is
    port(
        clk, reset : in std_logic;
        input : in std_logic_vector(15 downto 0);
        output : out std_logic_vector(15 downto 0)
    );
end component;

type pcreg_type is record
    input, pc_out, outputp2 : std_logic_vector(15 downto 0);
    src : std_logic_vector(1 downto 0);
end record pcreg_type;

signal pcreg : pcreg_type := (
    src => "00",
    others => (others => '0')
);

component mux_4_1 is
    generic (
        width : integer := 16
    );
    port (
        A3, A2, A1, A0 : in std_logic_vector(width-1 downto 0);
        S : in std_logic_vector(1 downto 0);
        Sout : out std_logic_vector(width-1 downto 0)
    );
end component;

component mux_2_1 is
    generic (
        width : integer := 16
    );
    port (
        A1, A0 : in std_logic_vector(width-1 downto 0);
        S : in std_logic;
        Sout : out std_logic_vector(width-1 downto 0)
    );
end component;

signal sltinput : std_logic_vector(15 downto 0);

begin
```

```
op <= instr(15 downto 12);
rd <= instr(11 downto 8);
rs <= instr(7 downto 4);
rt <= instr(3 downto 0);
imm12 <= instr(11 downto 0);
imm4 <= rt;

cpu_pc_register: PC_REG port map (
    clk => clk,
    reset => clear,
    input => pcreg.input,
    output => pcreg.pc_out
);

cpu_pc_mem: Memory generic map (INPUT => "Instr2.txt") port map (
    clk => clk,
    read_en => '1',
    write_en => '0',
    addr => pcreg.pc_out,
    data_in => (others => '0'),
    data_out => instr,
    mem_dump => '0'
);

pcreg.outputp2 <= pcreg.pc_out + 2;

cpu_pc_bne: BNE port map (
    pcp2 => pcreg.outputp2,
    offset => sgn_out,
    target => bne_target
);

cpu_pc_jmp: JMP port map (
    pcp2 => pcreg.outputp2,
    imm12 => imm12,
    target => jmp_target
);

cpu_pc_mux: mux_4_1 generic map (width => 16) port map (
    A0 => pcreg.outputp2,
    A1 => bne_target,
    A2 => jmp_target,
    A3 => jmp_target,
    S => pcreg.src,
    Sout => pcreg.input
);
```

```
    pcreg.src(1) <= ctrl.jump;
    pcreg.src(0) <= ctrl.branch and not alur.Zero;

    cpu_ctrl: Control port map (
        op => op,
        alu_op => ctrl.alu_op,
        alu_src => ctrl.alu_src,
        reg_dest => ctrl.reg_dest,
        reg_load => ctrl.reg_load,
        reg_src => ctrl.reg_src,
        mem_read => ctrl.mem_read,
        mem_write => ctrl.mem_write,
        branch => ctrl.branch,
        jump => ctrl.jump
    );

    cpu_reg_dest_mux: mux_2_1 generic map (width => 4) port map (
        A0 => rt,
        A1 => rd,
        S => ctrl.reg_dest,
        Sout => reg.c_addr
    );

    cpu_reg: Registers port map (
        clk => clk,
        clear => clear,
        load => ctrl.reg_load,
        a_addr => rd,
        a_data => reg.a_data,
        b_addr => rs,
        b_data => reg.b_data,
        c_addr => reg.c_addr,
        c_data => reg.c_data
    );

    cpu_sgn: Signextend generic map (in_width => 4, out_width => 16) port map (
        input => imm4,
        output => sgn_out
    );

    cpu_alu_src_mux: mux_2_1 generic map (width => 16) port map (
        A0 => reg.c_data,
        A1 => sgn_out,
        S => ctrl.alu_src,
        Sout => alur.B
    );
```

```
    cpu_alu16b: ALU_16b port map (
        A => reg.b_data,
        B => alur.B,
        S => ctrl.alu_op,
        Zero => alur.Zero,
        Sout => Alur.Sout
    );

    cpu_mem: Memory generic map (INPUT => "in.txt") port map (
        clk => clk,
        read_en => ctrl.mem_read,
        write_en => ctrl.mem_write,
        addr => alur.Sout,
        data_in => reg.c_data,
        data_out => mem_out,
        mem_dump => mem_dump
    );

    sltinput <= (
        0 => alur.Sout(15),
        others => '0'
    );

    cpu_reg_src_mux: mux_4_1 generic map (width => 16) port map (
        A0 => mem_out,
        A1 => alur.Sout,
        A2 => sltinput,
        A3 => sltinput,
        S => ctrl.reg_src,
        Sout => reg.a_data
    );

end behav;
```

## Appendix C: VHDL Code for Existing Modules

**Figure C1.** VHDL Code for Full Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Full_Adder is
    port (
        A, B, Cin : in std_logic;
        Sout, Cout : out std_logic
    );
end Full_Adder;

architecture Behavioral of Full_Adder is

begin
    Sout <= A xor B xor Cin;
       Cout <= (A and B) or ((A xor B) and Cin);
end Behavioral;
```

**Figure C2.** VHDL Code for 1-Bit ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ALU is
    port (
        A, B, Cin : in std_logic;
        S : in std_logic_vector(1 downto 0);
        Sout, Cout : out std_logic
    );
end ALU;

architecture behav of ALU is

component Full_Adder is
    port (
        A, B, Cin : in std_logic;
        Sout, Cout : out std_logic
    );
end component;
signal FA_B : std_logic := '0';
signal FA_Sout : std_logic_vector(0 downto 0) := "0";

component mux_4_1 is
    generic (
```

```
        width : integer := 1
    );
    port (
        A3, A2, A1, A0 : in std_logic_vector(width-1 downto 0);
        S : in std_logic_vector(1 downto 0);
        Sout : out std_logic_vector(width-1 downto 0)
    );
end component;
signal AandB, AorB, mux_Sout : std_logic_vector(0 downto 0) := "0";

begin
    FA_B <= B xor S(0);

    adder: Full_Adder port map (
        A => A,
        B => FA_B,
        Cin => Cin,
        Cout => Cout,
        Sout => FA_Sout(0)
    );

    AandB(0) <= A and B;
    AorB(0) <= A or B;

    mux: mux_4_1 generic map (width => 1) port map (
        A0 => FA_Sout,
        A1 => FA_Sout,
        A2 => AandB,
        A3 => AorB,
        S => S,
        Sout => mux_Sout
    );

    Sout <= mux_Sout(0);

end behav;
```

**Figure C3.** VHDL Code for 2-to-1 Multiplexer

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux_2_1 is
    generic (
        width : integer := 16
    );
    port (
        A1, A0 : in std_logic_vector(width-1 downto 0);
        S : in std_logic;
        Sout : out std_logic_vector(width-1 downto 0)
    );
end mux_2_1;

architecture behav of mux_2_1 is
begin
    Sout <= A0 when S = '0' else
            A1 when S = '1' else (others => 'X');
end behav;
```

**Figure C4.** VHDL Code for 4-to-1 Multiplexer

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux_4_1 is
    generic (
        width : integer := 16
    );
    port (
        A3, A2, A1, A0 : in std_logic_vector(width-1 downto 0);
        S : in std_logic_vector(1 downto 0);
        Sout : out std_logic_vector(width-1 downto 0)
    );
end mux_4_1;

architecture behav of mux_4_1 is
begin
    Sout <= A0 when S = "00" else
            A1 when S = "01" else
            A2 when S = "10" else
            A3 when S = "11" else
            (others => 'X');
end behav;
```

**Figure C5.** VHDL Code for Memory

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;


entity Memory is
   generic (
       INPUT : string := "in.txt";
       OUTPUT : string := "out.txt"
   );
    port (
        clk : in std_logic;
        read_en : in std_logic;
        write_en : in std_logic;
        addr : in std_logic_vector(15 downto 0);
        data_in : in std_logic_vector(15 downto 0);
        data_out : out std_logic_vector(15 downto 0);
        mem_dump : in std_logic := '0'
    );
end entity Memory;


architecture beh of Memory is
    type mem_type is array (0  to  33) of std_logic_vector(7  downto 0);

    impure function init_mem return mem_type is
        file in_file : text;
        variable in_line : line;
        variable byte : bit_vector(7  downto 0);
        variable mem : mem_type;
    begin
        if INPUT'length = 0 then
            return mem;
        end if;

        file_open(in_file, INPUT, READ_MODE);
        for row in mem_type'range loop
            readline(in_file, in_line);
            read(in_line, byte);
            mem(row) := to_stdlogicvector(byte);
        end loop;

        return mem;
    end function init_mem;
```

```
    signal mem : mem_type := init_mem;

begin
    read_p : process (read_en, mem, addr) is
    begin
        if read_en = '1' and conv_integer(addr(7 downto 0)) < 33 then -- NEW: ADD
SAFEGUARD
            data_out(7  downto 0) <= mem(conv_integer(addr(7 downto 0)));
            data_out(15 downto 8) <= mem(conv_integer(addr(7 downto 0))+1);
        end if;
    end process read_p;

    write_p : process (clk) is
    begin
        if rising_edge(clk) and write_en = '1' and conv_integer(addr(7 downto 0)) <
33 then
            mem(conv_integer(addr(7 downto 0))) <= data_in(7  downto 0);
            mem(conv_integer(addr(7 downto 0))+1) <= data_in(15 downto 8);
        end if;
    end process write_p;
end architecture beh;
```

**Figure C6.** VHDL Code for Register File

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

Entity Registers is
       port(
             clk    :     in     std_logic;
             clear  :     in  std_logic;

             a_addr:      in     std_logic_vector( 3 downto 0);
             a_data:      in     std_logic_vector(15 downto 0);
             load   :     in     std_logic;

             b_addr:      in     std_logic_vector( 3 downto 0);
             c_addr:      in     std_logic_vector( 3 downto 0);

             b_data:      out std_logic_vector(15 downto 0);
             c_data:      out std_logic_vector(15 downto 0)
             );
End Registers;

architecture syn of Registers is
```

```
	type ram_type is array (15 downto 0) of std_logic_vector(15 downto 0);
	signal REG   :     ram_type;
begin
	process(clk, load, clear)
	begin
		if (clear = '0') then
			REG(0) <= x"0000";
			REG(1) <= x"0000";
			REG(2) <= x"0000";
			REG(3) <= x"0000";

			REG(4) <= x"0000";
			REG(5) <= x"0000";
			REG(6) <= x"0000";
			REG(7) <= x"0000";

			REG(8) <= x"0000";
			REG(9) <= x"0000";
			REG(10)      <= x"0000";
			REG(11)      <= x"0000";

			REG(12)      <= x"0000";
			REG(13)      <= x"0000";
			REG(14)      <= x"0000";
			REG(15)      <= x"0000";
		elsif (clk'event and clk='1') then
			if (load = '1') then
				REG(conv_integer(a_addr)) <= a_data;
			end if;
		end if;
		REG(0) <= x"0000";
		REG(1) <= x"0001";
	end process;
	b_data <= REG(conv_integer(b_addr));
	c_data <= REG(conv_integer(c_addr));
end syn;
```

## Appendix D: VHDL Test Bench and Simulation

**Figure D1.** VHDL Test Bench for CPU

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


entity CPU_3380_Test is
end entity CPU_3380_Test;


architecture mixed of CPU_3380_Test is
    constant tick : time := 100 ns;
        constant RUN_TIME : integer := 18;
    signal reset, clock : std_logic;
        signal mem_dump : std_logic := '0';
begin
    uut : entity work.CPU_3380
        port map(
            clk           => clock,
            clear   => reset,
                          mem_dump    => mem_dump
        );

    driver : process is
    begin
        -- reset the system
        reset <= '0';
               wait for 50 ns;
        reset <= '1';

                    for i in  1 to RUN_TIME loop
                          wait for tick;
                    end loop;
        wait;
    end process driver;

    clock_p : process is
    begin
        for i in 0 to 50 loop
            clock <= '1'; wait for tick/2;
            clock <= '0'; wait for tick/2;
        end loop;
        wait;
    end process clock_p;
end architecture mixed;
```

**Figure D2.** Simulation Waveform with Registers

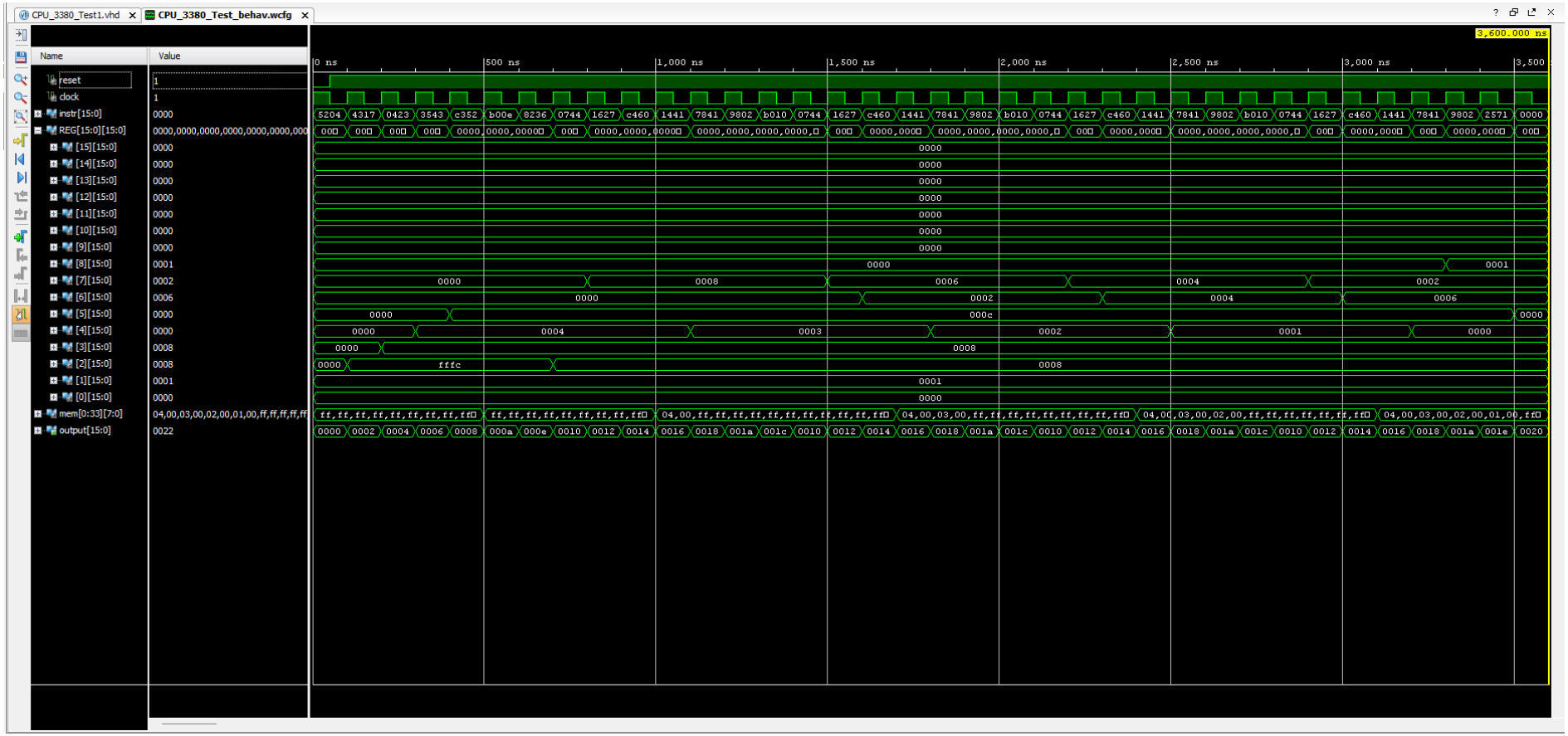


**Figure D3.** Simulation Waveform with Memory