# CPU Design Project Part 3 - Datapath/Control Implementation

Joe Driscoll

March 8, 2019

# Contents

# 1 Component Implementations

This section offers brief descriptions of each of the components in the datapath. The code for each component is located in the report's appendix. Testing and verification of each component is discussed in detail in the later verification section. An important file that is not directly referenced in any of these sections is the types.vhd file, which implements many important data types used by the other components. This file is included at the beginning of the code appendix, and it defined things like opcodes, ALU operations, branch conditions, and control signal buses.

It should also be noted that the requirements document suggests that all sequential components have reset signals, but none of the sequential components are implemented with a reset signal. This is because the two sequential components implemented (the generic register and the register file) both perform initialization at startup. The generic register initializes to all zeros, and the register file also initializes to all zeros. In synthesis and implementation, the FPGA includes these initial states in the programmed implementation. In this sense, the initialization done in the code level (as opposed to including a reset port) performs the same task, just in different ways.

## 1.1 Program Counter

The program counter is just a simple register. To implement it (and any other potentially needed registers), a generic register model was created so that the program counter and other registers in the future could be instantiated as it. The code for the "generic_register" entity is shown in the appendix. In the top-level datapath, the PC will simply be an instantiated version of this generic model.

The generic register model (and thus the PC) contains four ports, namely "data_in", "data_out", "write_enable", and "clock". The "data_in" signal is simply written to the register if "write_enable" is high on a clock edge. The "data_out" signal simply outputs the value stored in the register at all times.

## 1.2 Register File

The register file is modeled internally as an array whose entries are sixteen bits. The register file is essentially a defined interface between the rest of the datapath and this array, where two elements (registers) in the array can be read at the same time and one may be written to.

More specifically, the register file consists of seven ports: "read_address_1", "read_address_2", "write_address", "data_in", "write_enable", "data_out_1", and "data_out_2". The two read address inputs determine which registers should be output on the two data out ports. The "data_in" port is written to the register number on the "write_address" port asynchronously when "write_enable" is high.

## 1.3 ALU

The ALU implements all of the logical and arithmetic operations that the CPU performs. This ALU can perform adds, subtracts, AND's, OR's, NOT's, shift rights, and shift lefts.

The ALU consists of five ports: "alu_op", "alu_in1", "alu_in2", "alu_out", and "alu_zero". The ALU simply outputs the result of the operations specified by "alu_op" with the two inputs out onto "alu_out". If this output is exactly zero, then the "alu_zero" port is driven high.

## 1.4 Control Unit

The control unit coordinates control signals throughout the datapath, like the ALU operation, value to load into the PC, and many other routing/interconnection parts.

The unit consists of five ports: "op_code", "in_condition", "pc_condition", "clock", and "control_signals", where the first four are inputs and the last is an output. Although the control signals are listed as a single output port, it is in reality a bus (record in VHDL, struct in C, etc.) that contains multiple different control signals. This port definition simply enforces good design, as the interface between the controller and the rest of the datapath does not need to be changed if new control signals are added later. In essence, the controller takes the op code of the current instruction, as well as two extra bits from the instruction to account for conditional branches, and outputs control signals based on those inputs. The clock simply lets

the controller know when a new instruction is ready to be interpreted. The "in_condition" signal is the two-bit branch condition from the instruction itself, while the "pc_condition" signal is the result of the last compare instruction, and the controller compares these two to determine if a branch should be taken.

# 2 Verification and Testing

Testbenches were created for each of the four components discussed in the previous section. The code for each of testbenches can be found in the appendix, along with the code for the actual implementation of the components.

## 2.1 Generic Register (PC)

The testbench created for the generic register (which the PC will be an instance of) simply writes all $2^{16}$ possible values to the register and makes sure the correct value is output when written to. When run, the testbench completes without error - demonstrating that the generic register performs without fault. Figure 1 shows a segment of the signals generated by the testbench. Of course, not all values are shown because they cannot be fit into a single screenshot.

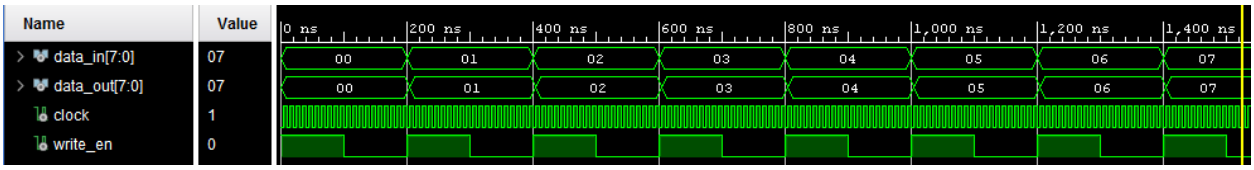


Figure 1. Signals generated by testbench for the generic register - simply writes and checks values.

## 2.2 Register File

The testbench created for the register file simply takes the register number $\rho$ and writes it to register number $N - \rho$, where N is the total number of registers. After the register numbers are written in this reverse order, all the registers are read using the two read ports (each reading in order, but starting from different register numbers) to verify the correct value was written and could be read back. When run, the testbench completes without fault - demonstrating the correct operation of the register file. Figures 2 and 3 show a small portion of the whole write and read process of the testbench.

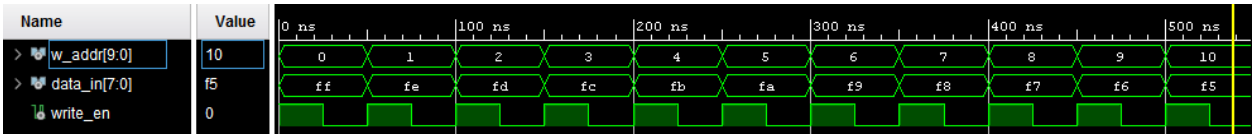


Figure 2. Beginning portion of register file writing in the testbench.

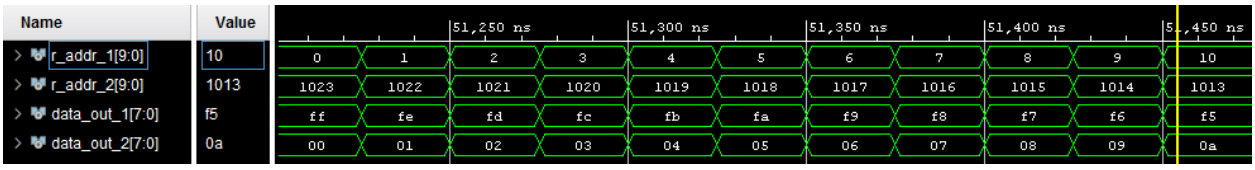


Figure 3. Beginning portion of register file reading in the testbench that verifies previous writes.

## 2.3 ALU

The testbench for the ALU was non-exhaustive, and it simply tried all possible combinations for a single input, the 201 combinations between -100 and 100 for the second input, and for each pair of these inputs,

it evaluated all seven arithmetic/logical operations. This took a few minutes to run, and some calculations reveal than an exhaustive test of the inputs could be run in approximately four hours. This was not done for time purposes, as well as the fact that a screenshot of the result would be practically meaningless since it would contain too much dense information. The testbench completes without fault, showing that the ALU performs correctly for all the operations it supports. Figure 4 shows a single pair of inputs for the testbench, demonstrating that each of the seven ALU operations are performing correctly for the inputs -15782 and 3.

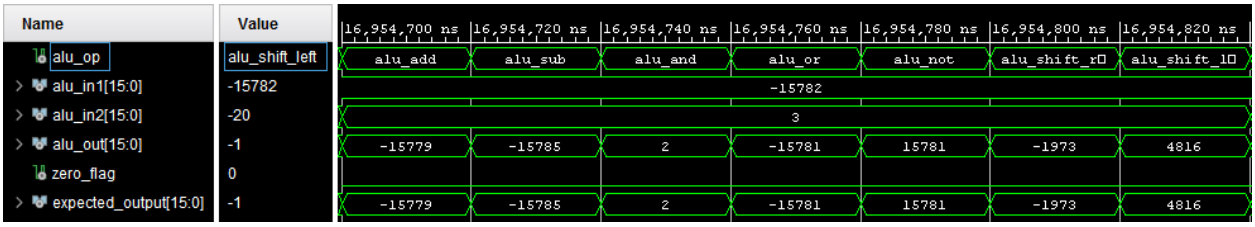


Figure 4. Segment of ALU testbench showing all seven ALU operations on inputs -15782 and 3.

## 2.4 Control Unit

The testbench for the control unit simply inputs every possible op code to the controller and checks that the correct control signals are generated for each one. Some extra work is needed to check the correctness of the branch instructions, since the branch signal depends on the two, two-bit condition inputs to the controller that come from the current instruction and from the PC register. For non-branch instructions, just the op code is varied and the correct list of control signals is used to check the outputs. For branch instructions, all $2^4$ possible branch conditions are tested to make sure the branch signal only activates when the two condition inputs are equal (this indicates the branch should be taken). The same table is used to ensure the control signals are correct, but the testbench checks the branch signal by logically OR-ing the branch bit into the list of default control signal values for the branch instructions.

Overall, the testbench ran without issue, demonstrating the correct operation of the control unit with respect to the design. Figure 5 and 6 respectively show the testbench in operation for non-branch instructions and for branch instructions. Not shown are the large table of control signal values that the output is compared with. The two tables that this comparison table is based off were shown in the second project report, and the tables are included in the appendix for reference.

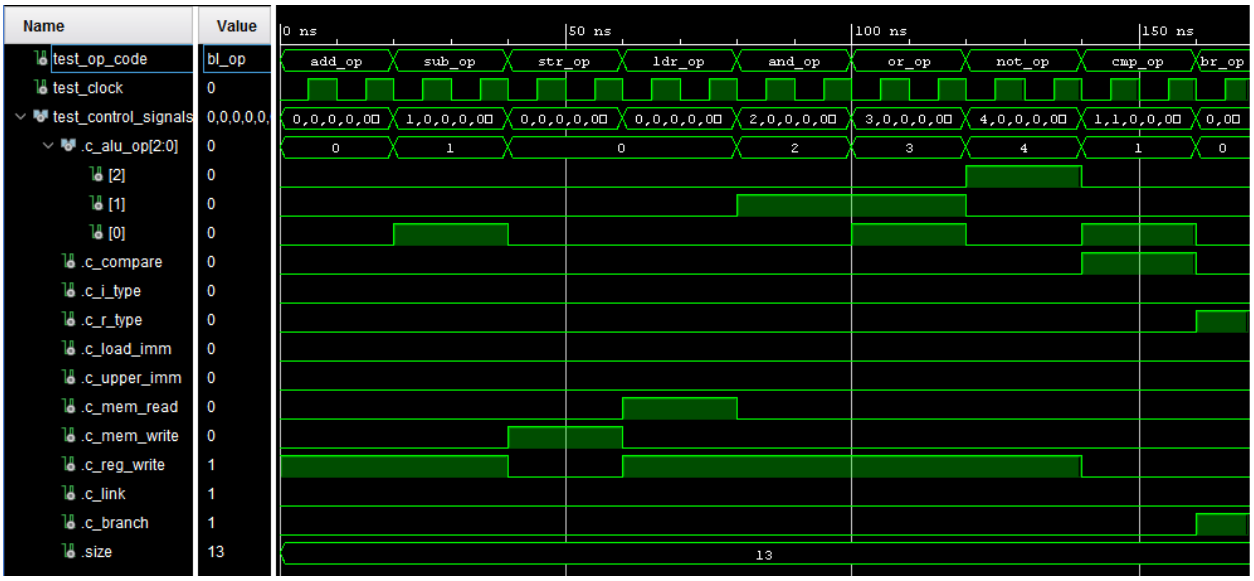


Figure 5. Portion of controller testbench on non-branch instructions.

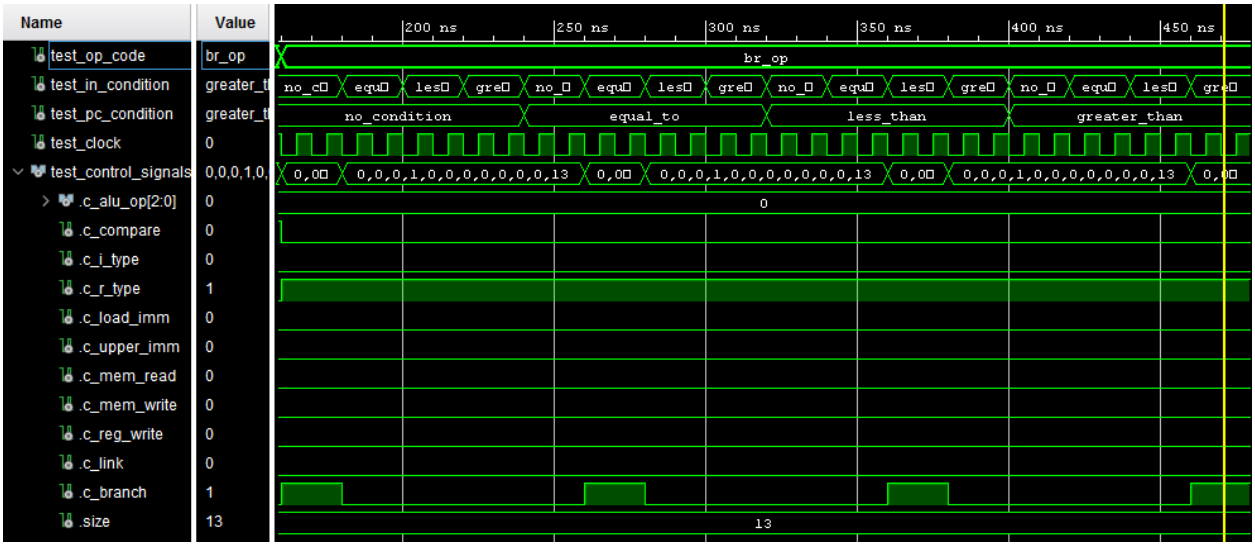


Figure 6. Portion of controller testbench on a branch instruction (br).

# 3 Design Changes

The new design has three major changes: the splitting of the main memory into an instruction, the removal of the instruction register, and data memory and the removal of the fetch signal. Single cycle data paths are very difficult to implement with a von Neumann style memory, so a Harvard architecture approach is now being taken to make the datapath operate in a simpler manner. Because of this, no instruction register or fetch control signal are needed, as the instruction will come directly from the instruction memory. Updated control signal tables are included in earlier sections of this report (i.e. the tables in this report do not have the fetch signal).

# A Implementation Code

## A.1 Custom Types

```
-- File: types.vhd
--
-- Declares some types common across implementations.


library ieee;
use ieee.std_logic_1164.all;


package types is

    -- Op code enumeration.
    type op_code_t is (add_op,
                       sub_op,
                       str_op,
                       ldr_op,
                       and_op,
                       or_op,
                       not_op,
                       cmp_op,
                       br_op,
                       b_op,
                       bl_op,
                       loadil_op,
                       loadiu_op,
                       addi_op,
                       lsr_op,
                       lsl_op);

    -- ALU operating type definitions.
    type alu_op_t is (alu_add,
                      alu_sub,
                      alu_and,
                      alu_or,
                      alu_not,
                      alu_shift_right,
                      alu_shift_left);

    -- Conditional branch types.
    type condition_t is (no_condition,
                         equal_to,
                         less_than,
                         greater_than);

    -- Bus that packs all control signals like a C struct.
    type control_signal_bus_t is record
        c_alu_op    : std_logic_vector(2 downto 0);
        c_compare   : std_logic;
        c_i_type    : std_logic;
        c_r_type    : std_logic;
        c_load_imm  : std_logic;
        c_upper_imm : std_logic;
        c_mem_read  : std_logic;
        c_mem_write : std_logic;
        c_reg_write : std_logic;
```

```
        c_link      : std_logic;
        c_branch    : std_logic;
        size        : integer;
    end record control_signal_bus_t;

end types;
```

## A.2 Generic Register (PC)

```
-- File: register.vhd
--
-- Implements a generic register model.


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


-- Register interface.
entity generic_register is

    generic (N : integer := 16);

    port    (data_in        : in  std_logic_vector((N - 1) downto 0);
             data_out       : out std_logic_vector((N - 1) downto 0);
             write_enable   : in  std_logic;
             clock          : in  std_logic);

end generic_register;

-- Register implementation.
architecture behavioral of generic_register is

    signal stored_value : std_logic_vector((N - 1) downto 0) := (others => '0');

begin

    process(clock)
    begin

        if rising_edge(clock) then

            if (write_enable = '1') then

                stored_value <= data_in;

            end if;

        end if;

    end process;

    data_out <= stored_value;

end behavioral;
```

## A.3 Register File

```
-- File: register_file.vhd
--
-- Implements a 2-port reading and 1-port writing
-- register file for use in the CPU.


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


-- Register file contains:
--     - 2 read input/output ports
--     - 1 write address port
--     - 1 data input port
--     - 1 write enable port
--
entity register_file is

    generic (REGISTER_SIZE  : integer := 8;
             ADDRESS_BITS   : integer := 4);

    port    (read_address_1 : in  std_logic_vector((ADDRESS_BITS - 1)  downto 0);
             read_address_2 : in  std_logic_vector((ADDRESS_BITS - 1)  downto 0);
             write_address  : in  std_logic_vector((ADDRESS_BITS - 1)  downto 0);
             data_in        : in  std_logic_vector((REGISTER_SIZE - 1) downto 0);
             write_enable   : in  std_logic;
             data_out_1     : out std_logic_vector((REGISTER_SIZE - 1) downto 0);
             data_out_2     : out std_logic_vector((REGISTER_SIZE - 1) downto 0));

end register_file;


architecture behavioral of register_file is

    -- Types for internal representation of register data.
    subtype register_t      is std_logic_vector((REGISTER_SIZE - 1) downto 0);
    type    register_file_t is array(((2**ADDRESS_BITS) - 1) downto 0) of register_t;

    -- Register file storage as a signal.
    signal register_file_storage : register_file_t := (others => (others => '0'));

begin

    -- Handle register writes.
    process(write_enable)
    begin

        if rising_edge(write_enable) then

            register_file_storage(to_integer(unsigned(write_address))) <= data_in;

        end if;

    end process;

    -- Constantly output read registers on the two output ports.
```

```
    data_out_1 <= register_file_storage(to_integer(unsigned(read_address_1)));
    data_out_2 <= register_file_storage(to_integer(unsigned(read_address_2)));

end behavioral;
```

## A.4 ALU

```
-- File: alu.vhd
--
-- Implements an ALU for the CPU project.


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


-- Import custom types.
use work.types.all;


entity alu is

    -- This generic should not really be changed.
    generic (IO_WIDTH : integer := 16);

    port    (alu_op    : in  alu_op_t;
             alu_in1   : in  std_logic_vector((IO_WIDTH - 1) downto 0);
             alu_in2   : in  std_logic_vector((IO_WIDTH - 1) downto 0);
             alu_out   : out std_logic_vector((IO_WIDTH - 1) downto 0) := (others => '0');
             zero_flag : out std_logic := '0');

end alu;


architecture behavioral of alu is

    -- Store results and assign after process for neatness.
    signal alu_result : std_logic_vector((IO_WIDTH - 1) downto 0);

    -- 0 comparison value for ALU output.
    constant zero_compare : std_logic_vector((IO_WIDTH - 1) downto 0) := (others => '0');

begin

    process(alu_op, alu_in1, alu_in2)

        -- Put inputs in integer variable for easier manipulation/readability.
        -- Only used in add and sub.
        variable alu_input_1 : integer;
        variable alu_input_2 : integer;

    begin

        case alu_op is

            when alu_add =>
```

```
                -- Cast the ALU inputs.
                alu_input_1 := to_integer(signed(alu_in1));
                alu_input_2 := to_integer(signed(alu_in2));
                alu_result  <= std_logic_vector(to_signed(alu_input_1 + alu_input_2, IO_WIDTH));

            when alu_sub =>

                -- Cast the ALU inputs.
                alu_input_1 := to_integer(signed(alu_in1));
                alu_input_2 := to_integer(signed(alu_in2));
                alu_result <= std_logic_vector(to_signed(alu_input_1 - alu_input_2, IO_WIDTH));

            when alu_and =>

                -- And the two inputs.
                alu_result <= alu_in1 and alu_in2;

            when alu_or =>

                -- Or the two inputs.
                alu_result <= alu_in1 or alu_in2;

            when alu_not =>

                -- Not just the first input.
                alu_result <= not alu_in1;

            when alu_shift_right =>

                -- Shift the first input to the right by the amount in the second input.
                alu_result <= std_logic_vector(shift_right(signed(alu_in1), to_integer(signed(alu_in2))));

            when alu_shift_left =>

                -- Shift the first input to the left by the amount in the second input.
                alu_result <= std_logic_vector(shift_left(signed(alu_in1), to_integer(signed(alu_in2))));

        end case;

    end process;

    -- Set the ALU output.
    alu_out   <= alu_result;

    -- Set the zero flag.
    zero_flag <= '1' when (alu_result = zero_compare) else '0';

end behavioral;
```

## A.5 Control Unit

```
-- File: control_unit.vhd
--
-- Implements the control unit for the CPU project.


-- Standard libraries and custom type definitions.
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.numeric_std.all;
use work.types.all;


-- Package that contains the control signals - basically packs the
-- control signals into a single bus using VHDL records (like a C-struct).
package control_unit_package is

    -- Function interface for casting std_logic_vector to control_signal_bus_t.
    function control_signal_bus(x : in std_logic_vector) return control_signal_bus_t;

    -- Function for converting control signal bus back to std_logic_vector.
    function to_std_logic_vector(x : in control_signal_bus_t) return std_logic_vector;

end package control_unit_package;


package body control_unit_package is

    -- Convert std_logic_vector to control signal bus (saves much code space).
    -- Note: assumes first three bits are alu_op.
    function control_signal_bus(x : in std_logic_vector) return control_signal_bus_t is

        variable output_bus : control_signal_bus_t;

    begin

        output_bus.c_alu_op     := x(12 downto 10);
        output_bus.c_compare    := x(9);
        output_bus.c_i_type     := x(8);
        output_bus.c_r_type     := x(7);
        output_bus.c_load_imm   := x(6);
        output_bus.c_upper_imm  := x(5);
        output_bus.c_mem_read   := x(4);
        output_bus.c_mem_write  := x(3);
        output_bus.c_reg_write  := x(2);
        output_bus.c_link       := x(1);
        output_bus.c_branch     := x(0);
        output_bus.size         := x'length;

        return output_bus;

    end function control_signal_bus;

    -- Function for converting control signal bus back to std_logic_vector.
    function to_std_logic_vector(x : in control_signal_bus_t) return std_logic_vector is

        variable output_vector : std_logic_vector((x.size - 1) downto 0);

    begin

        output_vector(12 downto 10) := x.c_alu_op;
        output_vector(9)  := x.c_compare;
        output_vector(8)  := x.c_i_type;
        output_vector(7)  := x.c_r_type;
        output_vector(6)  := x.c_load_imm;
        output_vector(5)  := x.c_upper_imm;
        output_vector(4)  := x.c_mem_read;
        output_vector(3)  := x.c_mem_write;
```

```
        output_vector(2)  := x.c_reg_write;
        output_vector(1)  := x.c_link;
        output_vector(0)  := x.c_branch;

        return output_vector;

    end function to_std_logic_vector;

end package body control_unit_package;
```

```
-- Standard libraries and custom type definitions.
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.types.all;
use work.control_unit_package.all;


-- Beginning of control unit interface and implementation.
entity control_unit is

    port (op_code           : in  op_code_t;
          in_condition      : in  condition_t;
          pc_condition      : in  condition_t;
          clock             : in  std_logic;
          control_signals   : out control_signal_bus_t);

end control_unit;


-- Basically a massive switch statement that activates control signals
-- when they are needed by an instruction.
architecture behavioral of control_unit is

    -- Number of control signals can change after updates to design.
    constant num_control_signals : integer := 13;
    subtype control_constant_t is std_logic_vector((num_control_signals - 1) downto 0);

    -- This table is based on the tables created for the second project report.
    constant ADD_SIGNALS    : control_constant_t := "0000000000100";
    constant SUB_SIGNALS    : control_constant_t := "0010000000100";
    constant STR_SIGNALS    : control_constant_t := "0000000001000";
    constant LDR_SIGNALS    : control_constant_t := "0000000010100";
    constant AND_SIGNALS    : control_constant_t := "0100000000100";
    constant OR_SIGNALS     : control_constant_t := "0110000000100";
    constant NOT_SIGNALS    : control_constant_t := "1000000000100";
    constant CMP_SIGNALS    : control_constant_t := "0011000000000";
    constant BR_SIGNALS     : control_constant_t := "0000010000000";
    constant B_SIGNALS      : control_constant_t := "0000000000000";
    constant BL_SIGNALS     : control_constant_t := "0000000000110";
    constant LOADIL_SIGNALS : control_constant_t := "0000001000100";
    constant LOADIU_SIGNALS : control_constant_t := "0000001100100";
    constant ADDI_SIGNALS   : control_constant_t := "0000100000100";
    constant LSR_SIGNALS    : control_constant_t := "1010100000100";
```

```
    constant LSL_SIGNALS    : control_constant_t := "1100100000100";

begin

    process(op_code, in_condition)

        -- Need to OR the constant patterns with branch taken signal based on input
        -- control signal ports.
        variable branch_setter : std_logic := '0';

        -- Need variables that store OR'd signals to preserve bit ordering.
        variable br_or : std_logic_vector(12 downto 0) := BR_SIGNALS;
        variable b_or  : std_logic_vector(12 downto 0) := B_SIGNALS;
        variable bl_or : std_logic_vector(12 downto 0) := BL_SIGNALS;

    begin

        -- Set branch based on input condition from instruction and condition in PC.
        if (in_condition = pc_condition) then
            branch_setter := '1';
        else
            branch_setter := '0';
        end if;

        -- OR in the branch control signal.
        br_or(0) := branch_setter;
        b_or(0)  := branch_setter;
        bl_or(0) := branch_setter;

        case op_code is

            -- Decode instruction.
            when add_op     => control_signals <= control_signal_bus(ADD_SIGNALS);
            when sub_op     => control_signals <= control_signal_bus(SUB_SIGNALS);
            when str_op     => control_signals <= control_signal_bus(STR_SIGNALS);
            when ldr_op     => control_signals <= control_signal_bus(LDR_SIGNALS);
            when and_op     => control_signals <= control_signal_bus(AND_SIGNALS);
            when or_op      => control_signals <= control_signal_bus(OR_SIGNALS);
            when not_op     => control_signals <= control_signal_bus(NOT_SIGNALS);
            when cmp_op     => control_signals <= control_signal_bus(CMP_SIGNALS);
            when br_op      => control_signals <= control_signal_bus(br_or);
            when b_op       => control_signals <= control_signal_bus(b_or);
            when bl_op      => control_signals <= control_signal_bus(bl_or);
            when loadil_op  => control_signals <= control_signal_bus(LOADIL_SIGNALS);
            when loadiu_op  => control_signals <= control_signal_bus(LOADIU_SIGNALS);
            when addi_op    => control_signals <= control_signal_bus(ADDI_SIGNALS);
            when lsr_op     => control_signals <= control_signal_bus(LSR_SIGNALS);
            when lsl_op     => control_signals <= control_signal_bus(LSL_SIGNALS);

        end case;

    end process;

end behavioral;
```

# B Verification Code

## B.1 Generic Register (PC) Testbench

```
-- File: register_test.vhd
--
-- Testbench for register component.


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


library work;
use work.all;


entity test_bench is
end test_bench;


architecture test of test_bench is

    -- Size of register to test.
    constant N : integer := 16;

    -- Signals to connect UUT.
    signal data_in  : std_logic_vector((N - 1) downto 0) := (others => '0');
    signal data_out : std_logic_vector((N - 1) downto 0);
    signal clock    : std_logic := '0';
    signal write_en : std_logic := '0';

begin

    -- Instantiate register.
    UUT : entity work.generic_register
        port map (data_in       => data_in,
                  data_out      => data_out,
                  write_enable  => write_en,
                  clock         => clock);

    -- Create 200 MHz clock.
    clock <= not clock after 5 ns;

    -- Testing patterns.
    process

        variable input_value : std_logic_vector((N - 1) downto 0):= (others => '0');

    begin

        for test_val in 0 to ((2**N) - 1) loop

            -- Set input of register to current loop value.
            input_value := std_logic_vector(to_unsigned(test_val, N));
            data_in <= input_value;

            -- Drive write enable.
```

```
            write_en <= '1';

            -- Wait for everything to settle and assert.
            wait for 100 ns;

            assert(input_value = data_out)
                report "Output data was not set correctly."
                severity FAILURE;

            write_en <= '0';
            wait for 100 ns;


        end loop;

        wait;

    end process;

end test;
```

## B.2 Register File Testbench

```
-- File: register_file_test.vhd
--
-- Testbench for register file component.


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


library work;
use work.all;


entity register_file_test_bench is
end register_file_test_bench;


architecture test of register_file_test_bench is

    -- Register file parameters.
    constant N : integer := 8;   -- Number of register bits.
    constant M : integer := 10; -- Number of registers in file.

    -- Signals to connect UUT.

    -- Inputs.
    signal r_addr_1   : std_logic_vector((M - 1) downto 0) := (others => '0');
    signal r_addr_2   : std_logic_vector((M - 1) downto 0) := (others => '0');
    signal w_addr     : std_logic_vector((M - 1) downto 0) := (others => '0');
    signal data_in    : std_logic_vector((N - 1) downto 0) := (others => '0');
    signal write_en   : std_logic := '0';

    -- Outputs.
    signal data_out_1 : std_logic_vector((N - 1) downto 0);
    signal data_out_2 : std_logic_vector((N - 1) downto 0);
```

```
begin

    -- Instantiate register file.
    UUT : entity work.register_file

        generic map (REGISTER_SIZE => N,
                     ADDRESS_BITS  => M)

        port map (read_address_1 => r_addr_1,
                  read_address_2 => r_addr_2,
                  write_address  => w_addr,
                  data_in        => data_in,
                  write_enable   => write_en,
                  data_out_1     => data_out_1,
                  data_out_2     => data_out_2);

    -- Testing patterns.
    process

        -- Input value to DUT and comparison variables for readability in read loop.
        variable input_value     : std_logic_vector((N - 1) downto 0):= (others => '0');
        variable compare_value_1 : std_logic_vector((N - 1) downto 0):= (others => '0');
        variable compare_value_2 : std_logic_vector((N - 1) downto 0):= (others => '0');

    begin

        -- For each register, write to it (total number of registers - register number).
        for reg_num in 0 to ((2**M) - 1) loop

            -- Set data_in current loop value.
            input_value := std_logic_vector(to_unsigned(((2**M) - 1) - reg_num, N));
            data_in <= input_value;

            -- Set write address to reg_num.
            w_addr <= std_logic_vector(to_unsigned(reg_num, M));

            -- Drive write enable.
            write_en <= '1';

            -- Wait for everything to settle and turn off write enable.
            wait for 25 ns;
            write_en <= '0';
            wait for 25 ns;

        end loop;

        -- Now read back values for each port, reading in opposite directions for each port.
        for reg_num in 0 to ((2**M) - 1) loop

            -- Get comparison values.
            compare_value_1 := std_logic_vector(to_unsigned(((2**M) - 1) - reg_num, N));
            compare_value_2 := std_logic_vector(to_unsigned(reg_num, N));

            -- Read from the ports (port 1 goes from low address to high address).
            r_addr_1 <= std_logic_vector(to_unsigned(reg_num, M));
            r_addr_2 <= std_logic_vector(to_unsigned(((2**M) - 1) - reg_num, M));
```

```
            -- Set write address to reg_num.
            w_addr <= std_logic_vector(to_unsigned(reg_num, M));

            -- Wait for everything to settle and turn off write enable.
            wait for 25 ns;

            -- Check correctness of both output port 1.
            assert(data_out_1 = compare_value_1)
                report "Output data on port 1 not set correctly."
                severity FAILURE;

            -- Check correctness of both output port 2.
            assert(data_out_2 = compare_value_2)
                report "Output data on port 2 not set correctly."
                severity FAILURE;

        end loop;

        -- End testbench.
        wait;

    end process;

end test;
```

## B.3 ALU Testbench

```
-- File: alu_test.vhd
--
-- Testbench for the ALU component.


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


-- Import custom types.
library work;
use work.types.all;
use work.all;


entity alu_test_bench is
end alu_test_bench;


architecture test of alu_test_bench is

    -- Signals for UUT instantiation.
    signal alu_op    : alu_op_t := alu_add;
    signal alu_in1   : std_logic_vector(15 downto 0) := (others => '0');
    signal alu_in2   : std_logic_vector(15 downto 0) := (others => '0');
    signal alu_out   : std_logic_vector(15 downto 0);
    signal zero_flag : std_logic;

    -- Signal used for comparing correct ALU output.
    signal expected_output : std_logic_vector(15 downto 0);
```

```
    -- Constant for comparison to output to all zero's.
    constant zeros_compare : std_logic_vector(15 downto 0) := (others => '0');

begin

    -- Instantiate ALU for testing.
    UUT : entity work.alu

        port map (alu_op    => alu_op,
                  alu_in1   => alu_in1,
                  alu_in2   => alu_in2,
                  alu_out   => alu_out,
                  zero_flag => zero_flag);

    -- Actual testing
    process

    begin
        -- Test all values for input_1, and selected values for input_2.
        for input_1 in (-2**14) to ((2**14) - 1) loop
            for input_2 in -100 to 100 loop

            -- Check correctness of ALU output for each operation.
            expected_output  <= std_logic_vector(to_signed(input_1 + input_2, 16));

            -- Set ALU inputs.
            alu_in1 <= std_logic_vector(to_signed(input_1, 16));
            alu_in2 <= std_logic_vector(to_signed(input_2, 16));


            -- First alu op.
            alu_op <= alu_add;
            wait for 20 ns;


            -- Addition with zero check.
            assert(alu_out = expected_output)
                report "Addition failed."
                severity FAILURE;

            assert((zero_flag = '0') xor (zeros_compare = expected_output))
                report "Addition zero-check failed."
                severity FAILURE;


            -- Next alu op type.
            alu_op <= alu_sub;
            expected_output  <= std_logic_vector(to_signed(input_1 - input_2, 16));
            wait for 20 ns;

            -- Subtraction with zero check.
            assert(alu_out = expected_output)
                report "Subtraction failed."
                severity FAILURE;

            assert((zero_flag = '0') xor (zeros_compare = expected_output))
                report "Subtraction zero-check failed."
                severity FAILURE;
```

```
-- Next alu op type.
alu_op <= alu_and;
expected_output  <= std_logic_vector(to_signed(input_1, 16))
                and std_logic_vector(to_signed(input_2, 16));
wait for 20 ns;

-- And with zero check.
assert(alu_out = expected_output)
    report "Logical AND failed."
    severity FAILURE;

assert((zero_flag = '0') xor (zeros_compare = expected_output))
    report "AND zero-check failed."
    severity FAILURE;


-- Next alu op type.
alu_op <= alu_or;
expected_output   <= std_logic_vector(to_signed(input_1, 16))
                 or  std_logic_vector(to_signed(input_2, 16));
wait for 20 ns;

-- Or with zero check.
assert(alu_out = expected_output)
    report "Logical OR failed."
    severity FAILURE;

assert((zero_flag = '0') xor (zeros_compare = expected_output))
    report "Or zero-check failed."
    severity FAILURE;


-- Next alu op type.
alu_op <= alu_not;
expected_output  <= not std_logic_vector(to_signed(input_1, 16));
wait for 20 ns;

-- Not with zero check.
assert(alu_out = expected_output)
    report "Logical NOT failed."
    severity FAILURE;

assert((zero_flag = '0') xor (zeros_compare = expected_output))
    report "NOT zero-check failed."
    severity FAILURE;


-- Next alu op type.
alu_op <= alu_shift_right;
expected_output   <= std_logic_vector(shift_right(to_signed(input_1, 16),
                                                  to_integer(to_signed(input_2, 16))));
wait for 20 ns;

-- Shift right with zero check.
assert(alu_out = expected_output)
    report "Shift right failed."
    severity FAILURE;
```

```
            assert((zero_flag = '0') xor (zeros_compare = expected_output))
                report "Shift right zero-check failed."
                severity FAILURE;


            -- Next alu op type.
            alu_op <= alu_shift_left;
            expected_output   <= std_logic_vector(shift_left(to_signed(input_1, 16),
                                                             to_integer(to_signed(input_2, 16))));
            wait for 20 ns;

            -- Shift left with zero check.
            assert(alu_out = expected_output)
                report "Shift left failed."
                severity FAILURE;

            assert((zero_flag = '0') xor (zeros_compare = expected_output))
                report "Shift left zero-check failed."
                severity FAILURE;

            end loop;
        end loop;

    wait;

    end process;

end test;
```

## B.4 Control Unit Testbench

```
-- File: controller_test.vhd
--
-- Testbench for controller component.


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


library work;
use work.all;
use work.types.all;
use work.control_unit_package.all;


entity controller_test_bench is
end controller_test_bench;


architecture test of controller_test_bench is

    -- This table is based on the tables created for the second project report.
    constant ADD_SIGNALS    : std_logic_vector(12 downto 0) := "0000000000100";
    constant SUB_SIGNALS    : std_logic_vector(12 downto 0) := "0010000000100";
    constant STR_SIGNALS    : std_logic_vector(12 downto 0) := "0000000001000";
    constant LDR_SIGNALS    : std_logic_vector(12 downto 0) := "0000000010100";
```

```
    constant AND_SIGNALS    : std_logic_vector(12 downto 0) := "0100000000100";
    constant OR_SIGNALS     : std_logic_vector(12 downto 0) := "0110000000100";
    constant NOT_SIGNALS    : std_logic_vector(12 downto 0) := "1000000000100";
    constant CMP_SIGNALS    : std_logic_vector(12 downto 0) := "0011000000000";
    constant BR_SIGNALS     : std_logic_vector(12 downto 0) := "0000010000000";
    constant B_SIGNALS      : std_logic_vector(12 downto 0) := "0000000000000";
    constant BL_SIGNALS     : std_logic_vector(12 downto 0) := "0000000000110";
    constant LOADIL_SIGNALS : std_logic_vector(12 downto 0) := "0000001000100";
    constant LOADIU_SIGNALS : std_logic_vector(12 downto 0) := "0000001100100";
    constant ADDI_SIGNALS   : std_logic_vector(12 downto 0) := "0000100000100";
    constant LSR_SIGNALS    : std_logic_vector(12 downto 0) := "1010100000100";
    constant LSL_SIGNALS    : std_logic_vector(12 downto 0) := "1100100000100";

    -- Signals used for testing the control unit under test.
    signal test_op_code             : op_code_t     := add_op;
    signal test_in_condition        : condition_t   := no_condition;
    signal test_pc_condition        : condition_t   := no_condition;
    signal test_clock               : std_logic     := '0';
    signal test_control_signals     : control_signal_bus_t;

begin

    -- Instantiate the controller.
    UUT : entity work.control_unit
        port map(op_code            => test_op_code,
                 in_condition       => test_in_condition,
                 pc_condition       => test_pc_condition,
                 clock              => test_clock,
                 control_signals    => test_control_signals);

    -- 200 MHz clock.
    test_clock <= not test_clock after 5 ns;

    process

        -- Variable used to determine if branch should be taken for a given instruction.
        variable branch_taken : std_logic_vector((ADD_SIGNALS'length - 1) downto 0) := (others => '0');

    begin

        -- Iterate through all op codes.
        for op_code in op_code_t range add_op to lsl_op loop

            -- Set new op code.
            test_op_code <= op_code;

            -- If instruction is a branch, then each condition must be done for each possible condition state.
            if ((op_code = br_op) or (op_code = b_op) or (op_code = bl_op)) then

                -- Iterate over all possible branch conditions in instruction.
                for i_condition in condition_t range no_condition to greater_than loop

                    -- Iterate over all possible compare conditions.
                    for pc_condition in condition_t range no_condition to greater_than loop

                        -- Set the conditions of current loop.
                        test_in_condition <= pc_condition;
                        test_pc_condition <= i_condition;
```

```
            -- See if branch should be taken based on conditions.
            if (pc_condition = i_condition) then
                branch_taken(0) := '1';
            else
                branch_taken(0) := '0';
            end if;

            -- Wait for output of controller to update.
            wait for 20 ns;

            -- Assert based on which of the three branch instructions is being executed.
            case op_code is

                when br_op =>

                    assert((BR_SIGNALS or branch_taken) = to_std_logic_vector(test_control_signals))
                        report "br failed."
                        severity FAILURE;

                when b_op =>

                    assert((B_SIGNALS or branch_taken) = to_std_logic_vector(test_control_signals))
                        report "b failed."
                        severity FAILURE;

                when bl_op =>

                    assert((BL_SIGNALS or branch_taken) = to_std_logic_vector(test_control_signals))
                        report "bl failed."
                        severity FAILURE;

            end case;

        end loop;

    end loop;

else

    wait for 20 ns;

    -- Big switch for every other op code.
    case op_code is

        when add_op     =>

            assert (to_std_logic_vector(test_control_signals) = ADD_SIGNALS)
                report "add failed"
                severity FAILURE;

        when sub_op     =>

            assert (to_std_logic_vector(test_control_signals) = SUB_SIGNALS)
                report "sub failed"
                severity FAILURE;

        when str_op     =>

            assert (to_std_logic_vector(test_control_signals) = STR_SIGNALS)
```

```
        report "str failed"
        severity FAILURE;

when ldr_op     =>

    assert (to_std_logic_vector(test_control_signals) = LDR_SIGNALS)
        report "ldr failed"
        severity FAILURE;

when and_op     =>

    assert (to_std_logic_vector(test_control_signals) = AND_SIGNALS)
        report "and failed"
        severity FAILURE;

when or_op      =>

    assert (to_std_logic_vector(test_control_signals) = OR_SIGNALS)
        report "or failed"
        severity FAILURE;

when not_op     =>

    assert (to_std_logic_vector(test_control_signals) = NOT_SIGNALS)
        report "not failed"
        severity FAILURE;

when cmp_op     =>

    assert (to_std_logic_vector(test_control_signals) = CMP_SIGNALS)
        report "cmp failed"
        severity FAILURE;

when loadil_op  =>

    assert (to_std_logic_vector(test_control_signals) = LOADIL_SIGNALS)
        report "loadil failed"
        severity FAILURE;

when loadiu_op  =>

    assert (to_std_logic_vector(test_control_signals) = LOADIU_SIGNALS)
        report "loadiu failed"
        severity FAILURE;

when addi_op    =>

    assert (to_std_logic_vector(test_control_signals) = ADDI_SIGNALS)
        report "addi failed"
        severity FAILURE;

when lsr_op     =>

    assert (to_std_logic_vector(test_control_signals) = LSR_SIGNALS)
        report "lsr failed"
        severity FAILURE;

when lsl_op     =>
```

```
                    assert (to_std_logic_vector(test_control_signals) = LSL_SIGNALS)
                        report "lsl failed"
                        severity FAILURE;

                end case;

            end if;

        end loop;

        wait;

    end process;

end test;
```

# C Control Signal Tables

Table 1: Control signal values for each instruction in the ISA.

| op | alu_op | cmp | i_type | r_type | ldi | upi |
|---|---|---|---|---|---|---|
| add | 000 | 0 | 0 | 0 | 0 | 0 |
| sub | 001 | 0 | 0 | 0 | 0 | 0 |
| str | 000 | 0 | 0 | 0 | 0 | 0 |
| ldr | 000 | 0 | 0 | 0 | 0 | 0 |
| and | 010 | 0 | 0 | 0 | 0 | 0 |
| or | 011 | 0 | 0 | 0 | 0 | 0 |
| not | 100 | 0 | 0 | 0 | 0 | 0 |
| cmp | 001 | 1 | 0 | 0 | 0 | 0 |
| br | XXX | 0 | 0 | 1 | 0 | 0 |
| b | XXX | 0 | 0 | 0 | 0 | 0 |
| bl | XXX | 0 | 0 | 0 | 0 | 0 |
| loadil | XXX | 0 | 0 | 0 | 1 | 0 |
| loadiu | XXX | 0 | 0 | 0 | 1 | 1 |
| addi | 000 | 0 | 1 | 0 | 0 | 0 |
| lsr | 101 | 0 | 1 | 0 | 0 | 0 |
| lsl | 110 | 0 | 1 | 0 | 0 | 0 |

Table 2: Control signal values for each instruction in the ISA. (cont.)

| op | m_read | m_write | r_write | link | branch |
|---|---|---|---|---|---|
| add | 0 | 0 | 1 | 0 | 0 |
| sub | 0 | 0 | 1 | 0 | 0 |
| str | 0 | 1 | 0 | 0 | 0 |
| ldr | 1 | 0 | 1 | 0 | 0 |
| and | 0 | 0 | 1 | 0 | 0 |
| or | 0 | 0 | 1 | 0 | 0 |
| not | 0 | 0 | 1 | 0 | 0 |
| cmp | 0 | 0 | 0 | 0 | 0 |
| br | 0 | 0 | 0 | 0 | 1 |
| b | 0 | 0 | 0 | 0 | 1 |
| bl | 0 | 0 | 1 | 1 | 1 |
| loadil | 0 | 0 | 1 | 0 | 0 |
| loadiu | 0 | 0 | 1 | 0 | 0 |
| addi | 0 | 0 | 1 | 0 | 0 |
| lsr | 0 | 0 | 1 | 0 | 0 |
| lsl | 0 | 0 | 1 | 0 | 0 |