CentraleSupélec

Architecture des ordinateurs
Report

TD2
Designing a simple datapath in VHDL

Group :
Daniel Stulberg Huf
Lawson Oliveira Lima
Lucas Vitoriano de Queiroz Lira

Professors :
Salvador Perea Ruben
Hiet Guillaume

December 16, 2022

Contents

1 Introduction

In this report, we will cover the process of simulate how a real hardware deal with instructions, registers, and arithmetic operations using VHDL. The decisions we made regarding the design of the architecture will be explained along with the source code of each step. All the simulations were ran using the Vivado software.

2 Basic ALU design

In this first step, the basic ALU with 4 instructions was built. The source code written in alu.vhd is shown below.

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

entity alu is
  port(in1          : in  std_logic_vector(31 downto 0);
       in2          : in  std_logic_vector(31 downto 0);
       op           : in  std_logic_vector(1 downto 0);
       res          : out std_logic_vector(31 downto 0));
end alu;

architecture behav of alu is
begin
  process (in1, in2, op)
  begin
    case op is
      when "00" =>
        res <= std_logic_vector(signed(in1) + signed(in2));
      when "01" =>
        res <= std_logic_vector(signed(in1) - signed(in2));
      when "10" =>
        res <= in1 and in2;
      when "11" =>
        res <= in1 or in2;
      when others =>
        null;
    end case;
  end process;
end;
```

Inside the architecture, the When statement was used to treat all the 2-bit input cases. For the operations of addition and subtraction, it was necessary to convert the inputs to "signed" in order to perform such arithmetic operations, and then reverse it back to "std_logic_vector" composite type to assign it to the output. As for the AND and OR operations, we directly used the corresponding logical operators between the two inputs and then assigned the result to the output. In other cases, the null statement will perform no action.

The result of the testbench simulation can be seen in the figure 1.

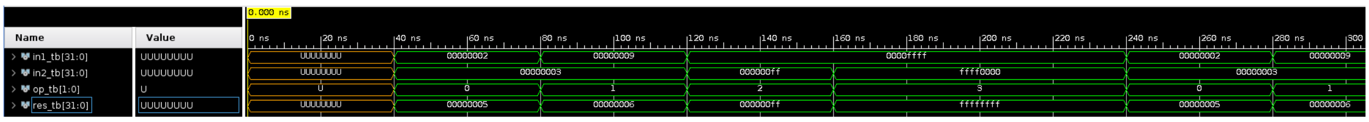


Figure 1: Simulation for the ALU benchmark.

3 Basic ALU + Output Register

In the second step, an output register was added after the ALU. The source code written in **alu_reg.vhd** is shown below.

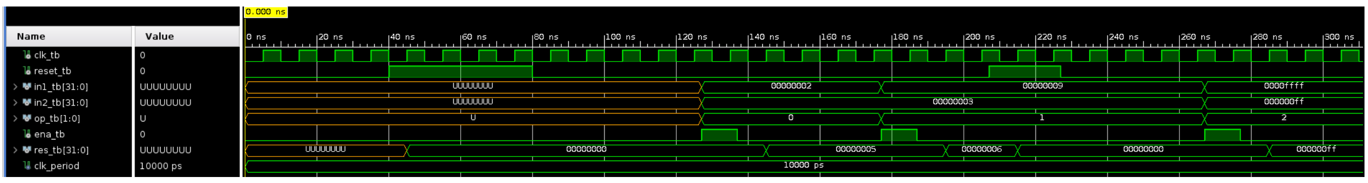
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu_reg is
  port(clk          : in  std_logic;
       reset        : in  std_logic;
       in1, in2     : in  std_logic_vector(31 downto 0);
       op           : in  std_logic_vector(1 downto 0);
       ena          : in  std_logic;
       res          : out std_logic_vector(31 downto 0));
end alu_reg;

architecture behav of alu_reg is

signal alu_out      : std_logic_vector(31 downto 0);
signal alu_op       : std_logic_vector(1 downto 0);
signal RegWrite     : std_logic;

begin

--------------------------------------------------------------------------------
-- Control unit
-- + Generate the ALU control signal to select the required operation
--    | In a real CPU design this signal would result from decoding the instruction
--    | Suppose the ALU operation is already decoded and available in "Op" input
-- + Generate the write enable signal for the ALU result register
--------------------------------------------------------------------------------
process (clk)
begin
    if rising_edge(clk) then
        if (reset = '1') then
            alu_op      <= (others => '0');
            RegWrite    <= '0';
        elsif (ena = '1') then
            -- Complete the code for alu_op and RegWrite
            alu_op <= op;
            RegWrite <= '1';
        else
            -- Complete the code to make RegWrite active only 1 clock cycle
            RegWrite <= '0';
        end if;
    end if;
end process;

--------------------------------------------------------------------------------
-- ALU
--------------------------------------------------------------------------------
-- Use the ALU from before
```

```

process (in1, in2, alu_op)
  begin
    case alu_op is
      when "00" =>
        alu_out <= std_logic_vector(signed(in1) + signed(in2));
      when "01" =>
        alu_out <= std_logic_vector(signed(in1) - signed(in2));
      when "10" =>
        alu_out <= in1 and in2;
      when "11" =>
        alu_out <= in1 or in2;
      when others =>
        null;
    end case;
  end process;

--------------------------------------------------------------------------------
-- Output register to save ALU result
--------------------------------------------------------------------------------
-- Complete the code for output res
process (clk)
    begin
        if rising_edge(clk) then
            if(RegWrite = '1') then
                res <= alu_out;
            elsif(reset = '1') then
                res <= (others => '0');
            end if;
        end if;
end process;
end;
```

In this architecture, the design is split between control unit, ALU, and the output register. Moreover, the clock signal is being used as the design is now synchronous and sequential. For the control unit part, the process will only be activated once there is a rising edge in the clock counter. In the case where the reset is not activated and the ALU is enabled to receive signals, the variable alu_op will receive op, since we want the ALU to perform operations in this situation. RegWrite will also be activated because we want the results to be saved in the output register at this moment.

For the case when both reset and ALU are deactivated, we assigned the value '0' to RegWrite so that it will also become inactive (it will be activated only in the next clock cycle). As for the output register part, the output res will receive the signal from alu_out just when the clock is rising and the saving is enabled. Otherwise, if the clock is rising and the reset is activated, res will also be reset.

The result of the testbench simulation can be seen below



Name	Value
clk_tb	0
reset_tb	0
in1_tb[31:0]	UUUUUUUU
in2_tb[31:0]	UUUUUUUU
op_tb[1:0]	U
ena_tb	0
res_tb[31:0]	UUUUUUUU
clk_period	10000 ps

Figure 2: Simulation for the ALU+Output Register benchmark.

We can notice in the image that for a moment the output is undefined, this is because of the alu_op, as it is also undefined the alu doesn't have a operation to do, so his output is floating.

4 Basic Datapath: ALU + Register File

In the third step, a register file, instead of a unique output register, was added together with the ALU. The source code written in alu_regfile.vhd is shown below.

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

entity alu_regfile is
  port(clk      : in  std_logic;
       reset    : in  std_logic;
       op       : in  std_logic_vector(1 downto 0);
       Rreg1    : in  std_logic_vector(2 downto 0);
       Rreg2    : in  std_logic_vector(2 downto 0);
       Wreg     : in  std_logic_vector(2 downto 0);
       ena      : in  std_logic;
       res      : out std_logic_vector(31 downto 0);
       init     : in  std_logic);
end alu_regfile;


architecture behav of alu_regfile is

-- ALU signals
signal alu_in1  : std_logic_vector(31 downto 0);
signal alu_in2  : std_logic_vector(31 downto 0);
signal alu_out  : std_logic_vector(31 downto 0);
signal alu_op   : std_logic_vector(1 downto 0);

-- Register file
type mem_array is array(7 downto 0) of std_logic_vector(31 downto 0);
signal reg_file: mem_array;
signal RegWrite : std_logic;

begin

--------------------------------------------------------------------------------
-- Control unit
-- + Generate the ALU control signal to select the required operation
--    | In a real CPU design this signal would result from decoding the instruction
--    | Suppose the ALU operation is already decoded and available in "op" input
-- + Generate the write signal to store the result in the register file
--------------------------------------------------------------------------------
-- Use the control unit from before
process (clk)
begin
    if rising_edge(clk) then
        if (reset = '1') then
            alu_op      <= (others => '0');
            RegWrite    <= '0';
        elsif (ena = '1') then
            alu_op <= op;
            RegWrite <= '1';
        else
            RegWrite <= '0';
        end if;
    end if;
end process;

--------------------------------------------------------------------------------
-- Register file
-- Write: synchronous with the clock edge, write enable signal
-- Read: combinational outputs, no read enable signal
--------------------------------------------------------------------------------
process (clk)
begin
    if rising_edge(clk) then
        -- Complete the code for reg_file: check multi-ported memories on the slides
        if(RegWrite = '1') then
            reg_file(to_integer(unsigned(Wreg))) <= alu_out;
        end if;
        -- Dirty hack to initialize the register file for a simple, meaningful simulation.
        -- So, only for simulation purposes in this special case.
        -- This is NOT the way to do it for a real design
        if (reset = '1') then
          reg_file <= (0 => (others => '0'),
            1 => x"00000000",
            2 => x"00000000",
            3 => x"00000000",
            4 => x"00000000",
            5 => x"00000000",
            6 => x"00000000",
            7 => x"00000000",
            others => (others => '0'));

        elsif(init = '1') then
            reg_file <= (0 => (others => '0'),
            1 => x"00000001",
            2 => x"00000002",
            3 => x"00000003",
            4 => x"00000004",
            5 => x"00000005",
            6 => x"00000006",
            7 => x"00000007",
            others => (others => '0'));
        end if;
    end if;
end process;

-- Complete the code
-- Register file outputs are the ALU inputs
alu_in1 <= reg_file(to_integer(unsigned(Rreg1)));
alu_in2 <= reg_file(to_integer(unsigned(Rreg2)));



--------------------------------------------------------------------------------
```

```
-- ALU
--------------------------------------------------------------------------------
-- Use the ALU from before
process (alu_op,alu_in1,alu_in2)
  begin
    case alu_op is
      when "00" =>
        alu_out <= std_logic_vector(signed(alu_in1) + signed(alu_in2));
      when "01" =>
        alu_out <= std_logic_vector(signed(alu_in1) - signed(alu_in2));
      when "10" =>
        alu_out <= alu_in1 and alu_in2;
      when "11" =>
        alu_out <= alu_in1 or alu_in2;
      when others =>
        null;
    end case;
end process;

--------------------------------------------------------------------------------
-- Output
--------------------------------------------------------------------------------
-- Assign the output
res <= alu_out;
end behav;
```

In this architecture, the design is split between control unit, ALU, and the register file. For the write register signal in the register file, the ALU output will only by written in the desired register of the register file once there is a rising edge in the clock and when RegWrite is activated (which automatically implies that the enable signal is also activated). Wreg is the desired register in which data will be written, and is represented as an array of bits, and the register file can be seen as a matrix of 8 lines (registers) and 32 columns (array of bits). Since the register file expects one register in which data coming from the ALU output will be written, we need to convert the array of bits to its corresponding integer.

We also added the reset case inside the rising edge condition, which will erase all the content from the register file. For the register file outputs to be the ALU inputs, the same conversion operation done before is being done again in order to access and send the content from the register file (by the two read registers) to the ALU. Finally, the ALU output was assigned to the output port of the architecture. The result of the testbench simulation can be seen below.

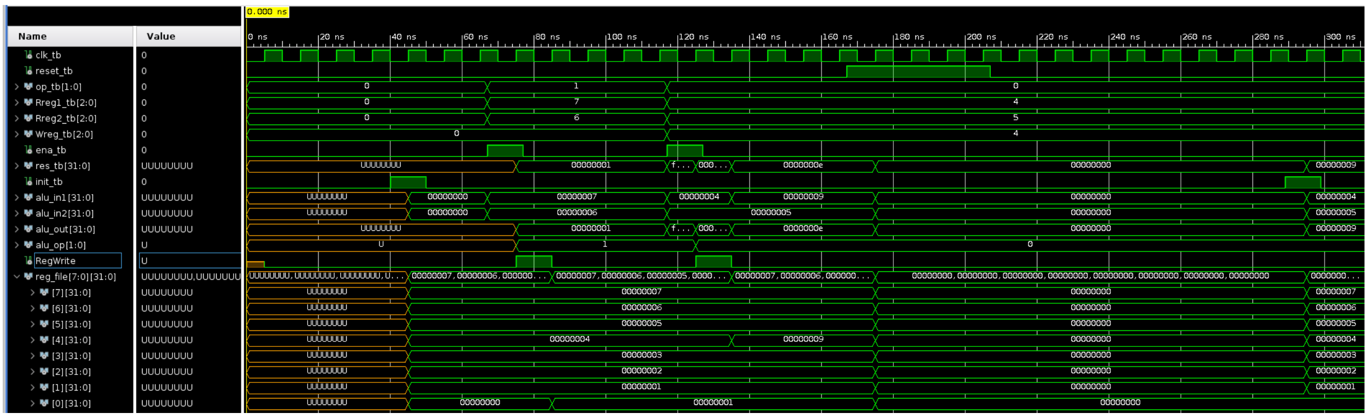


Figure 3: Basic datapath simulation.

5 Bonus question

Finally, for the bonus question, we have to read one instruction in the format RV32 and make the procedure respective to that function. The respective code to do that was written in datapath_and_control_regfile.vhd and is shown below.

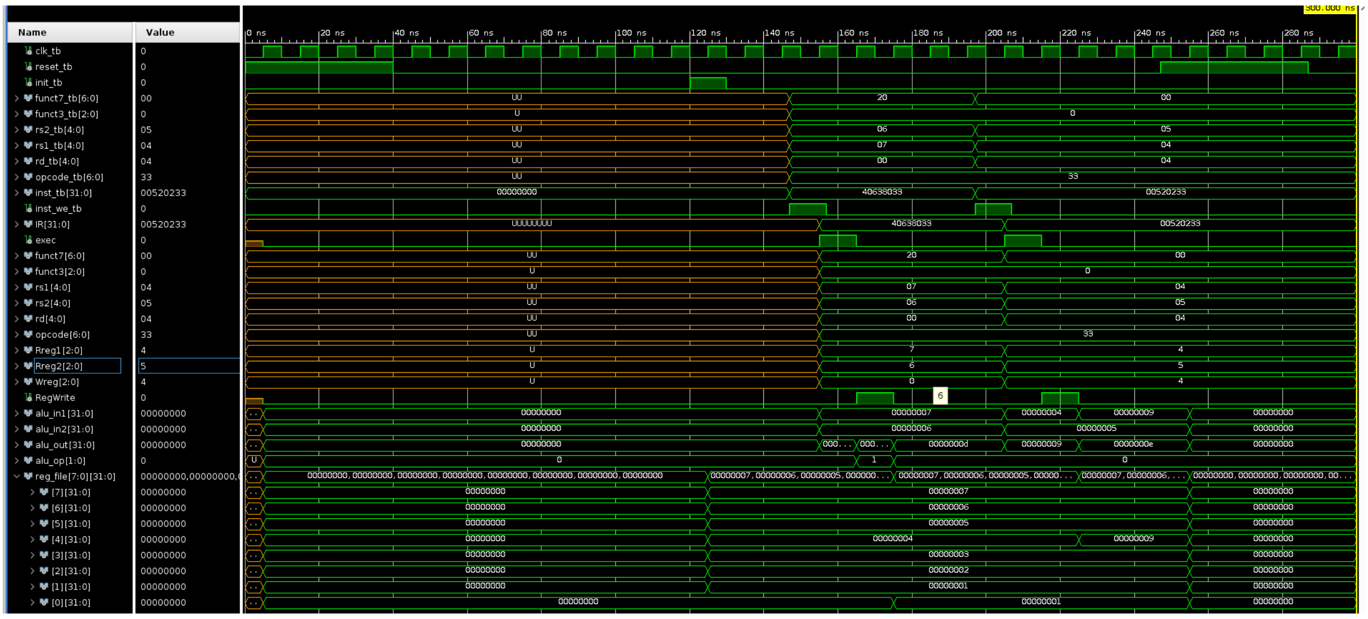
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity datapath_and_control is
  port(clk      : in  std_logic;
       reset    : in  std_logic;
       inst     : in  std_logic_vector(31 downto 0);
       inst_we  : in  std_logic;
       init     : in  std_logic);
end datapath_and_control;


architecture behav of datapath_and_control is
-- ALU signals
signal alu_in1  : std_logic_vector(31 downto 0);
signal alu_in2  : std_logic_vector(31 downto 0);
signal alu_out  : std_logic_vector(31 downto 0);
signal alu_op   : std_logic_vector(1 downto 0);

-- Register file
type mem_array is array(7 downto 0) of std_logic_vector(31 downto 0);
signal reg_file: mem_array;
-- Why size 3 ?
signal Rreg1    : std_logic_vector(2 downto 0);
signal Rreg2    : std_logic_vector(2 downto 0);
signal Wreg     : std_logic_vector(2 downto 0);

signal RegWrite : std_logic;

-- Instruction fields
signal IR       : std_logic_vector(31 downto 0);
signal funct7   : std_logic_vector(6 downto 0);
signal funct3   : std_logic_vector(2 downto 0);
signal rs1      : std_logic_vector(4 downto 0);
signal rs2      : std_logic_vector(4 downto 0);
signal rd       : std_logic_vector(4 downto 0);
signal opcode   : std_logic_vector(6 downto 0);

signal exec     : std_logic;

begin

--------------------------------------------------------------------------------
-- Instruction register
--------------------------------------------------------------------------------
-- Complete the code for the signals: IR and exec
--  IR: the instruction register
--  exec: we generate an auxiliary signal pulse, 1 clock cycle wide,
-- to enable the control unit


---------------------------------------------------------------------------------
-- Control unit
-- + 1. Generate the ALU control signal alu_op to select the required operation
--     | In a real CPU design this signal would result from decoding the instruction
--     | Suppose the ALU operation is already decoded and available in "op" input
-- + 2. Generate the write signal RegWrite to store the result in the register file
-- + 3. Extract the address of source and destination operands in the register file
---------------------------------------------------------------------------------


-- Let's make it a bit closer to RISC-V
-- Define RISC-V R-type instruction field names
funct7  <= IR(31 downto 25);
rs2     <= IR(24 downto 20);
rs1     <= IR(19 downto 15);
funct3  <= IR(14 downto 12);
rd      <= IR(11 downto 7);
opcode  <= IR(6 downto 0);

-- 1. and 2.
-- Complete the code for alu_op and RegWrite
process (clk)
begin
    if rising_edge(clk) then
      if (inst_we = '1') then
          exec <= '1';
          IR <= inst;
      else
        exec <= '0';
      end if;
    end if;
end process;

Rreg1   <= rs1(2 downto 0);
Rreg2   <= rs2(2 downto 0);
Wreg    <= rd(2 downto 0);

--Control Unit
process(clk)
  begin
    if rising_edge(clk) then
      if (exec = '1') then
        RegWrite <= '1';
        case( funct7 ) is
          when "0100000" =>
            alu_op <= "01";
          when "0000000" =>
            case( funct3 ) is
              when "000" =>
```

```
                alu_op <= "00";
              when "110" =>
                alu_op <= "10";
              when "111" =>
                alu_op <= "11";
              when others =>
                alu_op <= "00";
            end case ;
            when others =>
              alu_op <= "00";
          end case ;
      else
        RegWrite <= '0';
        alu_op <= "00";
      end if;
    end if;
end process;
-- 3.
-- Complete the code for the source and destination operands
-- Since our regfile and datapath is a stripped down version,
-- we keep only the bits required to address the 8 registers of our register file


--------------------------------------------------------------------------------
-- Register file
-- Write: synchronous with the clock edge, write enable signal
-- Read: combinational outputs, no read enable signal
--------------------------------------------------------------------------------
-- Use the same register file as before...
alu_in1 <= reg_file(to_integer(unsigned(Rreg1)));
alu_in2 <= reg_file(to_integer(unsigned(Rreg2)));


process (clk)
begin
    if rising_edge(clk) then
        -- Complete the code for reg_file: check multi-ported memories on the slides
        if(RegWrite = '1') then
            reg_file(to_integer(unsigned(Wreg))) <= alu_out;
        end if;
        -- Dirty hack to initialize the register file for a simple, meaningful simulation.
        -- So, only for simulation purposes in this special case.
        -- This is NOT the way to do it for a real design
        if (reset = '1') then
          reg_file <= (0 => (others => '0'),
            1 => x"00000000",
            2 => x"00000000",
            3 => x"00000000",
            4 => x"00000000",
            5 => x"00000000",
            6 => x"00000000",
            7 => x"00000000",
            others => (others => '0'));

        elsif(init = '1') then
            reg_file <= (0 => (others => '0'),
            1 => x"00000001",
            2 => x"00000002",
            3 => x"00000003",
            4 => x"00000004",
            5 => x"00000005",
            6 => x"00000006",
            7 => x"00000007",
            others => (others => '0'));
        end if;
    end if;
end process;


--------------------------------------------------------------------------------
-- ALU
--------------------------------------------------------------------------------
-- Use the same ALU as before ...
process (alu_op,alu_in1,alu_in2)
  begin
    case alu_op is
      when "00" =>
        alu_out <= std_logic_vector(signed(alu_in1) + signed(alu_in2));
      when "01" =>
        alu_out <= std_logic_vector(signed(alu_in1) - signed(alu_in2));
      when "10" =>
        alu_out <= alu_in1 and alu_in2;
      when "11" =>
        alu_out <= alu_in1 or alu_in2;
      when others =>
        null;
    end case;
end process;
end behav;
```

Inside the architecture, the design is split between the process responsible to deal with the instruction received, the control unit, ALU, and the register file. Because of the exec signal, the control unit will only send anything when the instruction is sent, exec will change only when we receive inst_we. After the send of the exec signal, the control unit will change his outputs, which are responsible to control where the data will be written and which registers are necessary to perform the operations. For the register file, it will only change a register inside if RegWrite is 1 and, as the ALU isn't dependent of anything, it will always show a output.

We also added the reset case inside the rising edge condition, which will erase all the content from the register file. For the register file outputs to be the ALU inputs, the same conversion operation done before is being done again in order to access and send the content from the register file (by the two read registers) to the ALU. Finally, the ALU output was assigned to the wire of WriteRegister, to send the new value to be saved in the RegisterFile.

The result of the testbench simulation can be seen below.



Name	Value
clk_tb	0
reset_tb	0
init_tb	0
funct7_tb[6:0]	00
funct3_tb[2:0]	0
rs2_tb[4:0]	05
rs1_tb[4:0]	04
rd_tb[4:0]	04
opcode_tb[6:0]	33
inst_tb[31:0]	00520233
inst_we_tb	0
IR[31:0]	00520233
exec	0
funct7[6:0]	00
funct3[2:0]	0
rs1[4:0]	04
rs2[4:0]	05
rd[4:0]	04
opcode[6:0]	33
Rreg1[2:0]	4
Rreg2[2:0]	5
Wreg[2:0]	4
RegWrite	0
alu_in1[31:0]	00000000
alu_in2[31:0]	00000000
alu_out[31:0]	00000000
alu_op[1:0]	0
reg_file[7:0][31:0]	00000000,00000000,
[7][31:0]	00000000
[6][31:0]	00000000
[5][31:0]	00000000
[4][31:0]	00000000
[3][31:0]	00000000
[2][31:0]	00000000
[1][31:0]	00000000
[0][31:0]	00000000

Figure 4: Bonus question.

In this case the wires where the Instruction is separated in sub functions are undefined in the beginning of the simulation because they receive directly a chop of the instructions, and as this last is also undefined they can't be assigned to a value.

6 Conclusion

To conclude, all the operations in the three source codes are working as expected, and the simulation results are very similar to those presented in the instructions. In addition, we learn in this practical work how a processor works, from interpretation to task execution.