E-mail: emjo2109@student.miun.se
Kurs: ET061G

Digitalteknik lab 4

Emil Jons, ET061G, Lab 4 VHDL CPU

E-mail: emjo2109@student.miun.se
Kurs: ET061G

Designing the CPU

Entity decleration

The first task of implementing the CPU module into VHDL code largly consisted of copying and translating what was given in the instructions into the code. The large chunk of code that could be copied straight over was the entity decleration, the part that declares the port names, if its an input or output, and if it's a vector with a specific size or not. The VHDL for the entity named "cpu" is shown below, in (code 1) part of "cpu.vhd".

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

entity cpu is
Port (
clk : in STD_LOGIC;
update : in STD_LOGIC;
ax : in STD_LOGIC_VECTOR(3 downto 0);
opcode : in STD_LOGIC_VECTOR(3 downto 0);
result_add : in STD_LOGIC_VECTOR(4 downto 0);
result_mult : in STD_LOGIC_VECTOR(7 downto 0);
result_not : in STD_LOGIC_VECTOR(3 downto 0);
result_and : in STD_LOGIC_VECTOR(3 downto 0);
result_or : in STD_LOGIC_VECTOR(3 downto 0);
A : out STD_LOGIC_VECTOR(3 downto 0);
B : out STD_LOGIC_VECTOR(3 downto 0);
C : out STD_LOGIC_VECTOR(3 downto 0);
D : out STD_LOGIC_VECTOR(3 downto 0);
op_in1 : out STD_LOGIC_VECTOR(3 downto 0);
op_in2 : out STD_LOGIC_VECTOR(3 downto 0) );
end cpu;
```

Code 1: Entity declaration, in "cpu.vhd"

E-mail: emjo2109@student.miun.se
Kurs: ET061G

Architecture of the CPU

The next part of the code is the start of the architectural description, which describes how the CPU should behave depending on various inputs. The CPU operates by analyzing the inputs of the opcode port and performs instructions corresponding to the CPU instructions set given in the lab instructions. The CPU will update when two if statement is met, being if the update port is one, and if the clock updates from 0 to 1, or “rising edge”. This part of the code also declares the internals signals BX, CX, DX and connects them to the ports B, C, D, as well as connects the port AX to port A.

```vhdl
architecture behave of cpu is

signal BX, CX, DX : STD_LOGIC_VECTOR(3 downto 0);

    begin
    A <= AX;
    B <= BX;
    C <= CX;
    D <= DX;

    clock: process(clk) begin

    if rising_edge(clk) then
        if update = '1' then
            case opcode is
            when "0000" =>
                BX <= AX;
            when "0001" =>
                CX <= AX;
            when "0010" =>
                DX <= AX;
```

E-mail: emjo2109@student.miun.se
Kurs: ET061G

Continuing opcode cases

The code below continues to correlate each opcode to a certain instruction. What is to be taken into account when writing what each opcode should do is that it performs all actions in the opcode in a single clock cycle. This means that you can not read or update a value, in the same clock cycle as using that same value for a specific operation. You will eaither need to wait for another clock cycle in the same opcode, or perform one of the operations in the previous opcode to separate the operations into two cycles. This is done in the opcode "0110", where op_in1 and op_in2 are updated in the previous opcode in order to use them in the current. The updated values are now saved and can be used in the coming operations ("0110" to "1010"). Op_in1 & 2 are once again updated in the opcode "1010" to be used in the final opcodes. It should be noted that this way of updating the value will only work if the opcodes are executed in consecutive order, due to the them now depending on the previous opcode in order to work as intended.

```
            when "0011" =>
                BX <= "1010";
            when "0100" =>
                CX <= "0000";
            when "0101" =>
                DX <= "0000";
                op_in1 <= "0010";
                op_in2 <= "0011";
            when "0110" =>
                CX <= result_add(3 downto 0);
                DX <= "0101";
            when "0111" =>
                CX <= result_mult(3 downto 0);
                DX <= "0110";
            when "1000" =>
                CX <= result_not;
                DX <= "1101";
            when "1001" =>
                CX <= result_and;
                DX <= "0010";
            when "1010" =>
                CX <= result_or;
                DX <= "0011";
                op_in1 <= AX;
                op_in2 <= BX;
            when "1011" =>
                DX <= "000"&result_add(4);
                CX <= result_add (3 downto 0);
            when "1100" =>
                DX <= result_mult(7 downto 4);
                CX <= result_mult(3 downto 0);
            when "1101" =>
                CX <= result_not ;
                DX <= "0000";
            when "1110" =>
                CX <= result_and;
                DX <= "0000";
            when "1111" =>
                CX <= result_or ;
                DX <= "0000";
```

E-mail: emjo2109@student.miun.se
Kurs: ET061G

Final opcode cases and ending the process

The final part of the code is included as a required part of switch cases in VHDL. It handles what happens when something other than the specified inputs are set as inputs. In our case we specify all possible combinations, but the code block should be included anyways.

```
           when others =>
                op_in1 <= "0001";
                op_in2 <= "0000";
                BX <= "0000";
                CX <= "0000";
                DX <= "0000";
             end case;
        end if;
    end if;
    end process clock;
end behave;
```

E-mail: emjo2109@student.miun.se
Kurs: ET061G

CPU Simulation

To simulate and assess if the CPU behaves in a correct way a testbench was created. The first part of the testbench, consists of the initialization of the components. Some of the components are reused from previous labs, them being the adder, multiplier, inverter, AND, OR. They are declared in the code below.

```
Library IEEE;
use work.all;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.all;
entity cpu_tb is
end cpu_tb;
architecture behave of cpu_tb is
  component add4
    port (
      in_1, in_2 : in  std_logic_vector(3 downto 0);
      s_out  : out std_logic_vector(4 downto 0));
  end component;
   component mul4
    port (
      in_1, in_2 : in  std_logic_vector(3 downto 0);
      m_out  : out std_logic_vector(7 downto 0));
  end component;
  component invrt4
    port (
      in_1 : in  std_logic_vector(3 downto 0);
      i_out  : out std_logic_vector(3 downto 0));
  end component;
  component bit_and4
    port (
      in_1, in_2 : in  std_logic_vector(3 downto 0);
      a_out  : out std_logic_vector(3 downto 0));
  end component;
  component bit_or4
    port (
      in_1, in_2 : in  std_logic_vector(3 downto 0);
      o_out  : out std_logic_vector(3 downto 0));
  end component;
```

The component decleration for the CPU is also included. The next part defines what signals should be used, the name of them and what datatype they are. The clock and the update signal will simply be either ‘1’ or ‘0’. The rest of the signals should be vectors of four bits excluding the adder and multiplier which requires five and eight bits respectivly.

```
component cpu
    port (
      clk, update : in STD_LOGIC;
      ax : in STD_LOGIC_VECTOR(3 downto 0);
      opcode : in STD_LOGIC_VECTOR(3 downto 0);
      result_add : in STD_LOGIC_VECTOR(4 downto 0);
      result_mult : in STD_LOGIC_VECTOR(7 downto 0);
      result_not : in STD_LOGIC_VECTOR(3 downto 0);
      result_and : in STD_LOGIC_VECTOR(3 downto 0);
      result_or : in STD_LOGIC_VECTOR(3 downto 0);
      A, B, C, D : out STD_LOGIC_VECTOR(3 downto 0);
      op_in1 : out STD_LOGIC_VECTOR(3 downto 0);
      op_in2 : out STD_LOGIC_VECTOR(3 downto 0) );
  end component;

  signal in_clk, in_update : std_logic;
  signal in_ax, in_opcode, input1, input2, A_out, B_out, C_out, D_out :
std_logic_vector(3 downto 0);
  signal out_inv, out_and, out_or : std_logic_vector(3 downto 0);
  signal out_adder : std_logic_vector(4 downto 0);
  signal out_mult : std_logic_vector(7 downto 0);
```

The next part of the testbench consists of port mapping the ports to the signals. The port mapping for the components, are the same as in lab three. There are two input signals which are connected into every component, and all components give out a output depending on what component is being used. When looking at the port mapping for the CPU there are some noteworthy connections being made. They are the connections for opcode and AX ports, that will be looped though and obtain every possible combination of values from 0 to 15 for both AX and opcode. The ports op_in1 & op_in2 are connected to input1 and input2, this will in turn make the ports function as input for all the components. The output result from each of the components are connected to the respective ports in the CPU for them to be used as operations in the specific opcodes. Finally, the ports A, B, C, D are connected to the respective signal in order to read the value of the signals in the simulation.

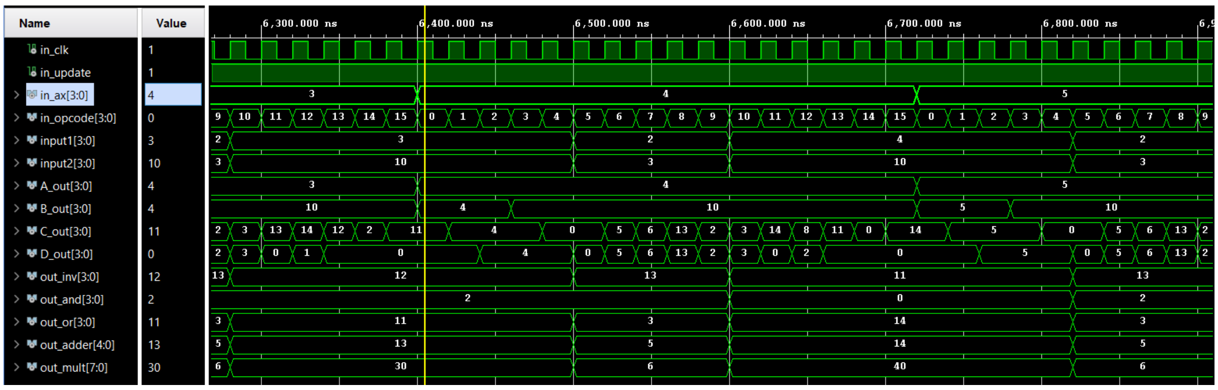
```
begin  -- behave
add4_0 : add4 port map (
    in_1            => input1,
    in_2            => input2,
    s_out           => out_adder);

mul4_0 : mul4 port map (
  in_1         => input1,
  in_2         => input2,
  m_out      => out_mult);

invrt4_0 : invrt4 port map (
    in_1            => input1,
    i_out           => out_inv);

bit_and4_0 : bit_and4  port map (
    in_1            => input1,
    in_2            => input2,
    a_out           => out_and);

bit_or4_0 : bit_or4  port map (
    in_1            => input1,
    in_2            => input2,
    o_out           => out_or);

cpu_0 : cpu  port map (
    clk              => in_clk,
    update           => in_update,
    ax               => in_ax,
    opcode           => in_opcode,
    op_in1           => input1,
    op_in2           => input2,
    result_add       => out_adder,
    result_mult      => out_mult,
    result_not       => out_inv,
    result_and       => out_and,
    result_or        => out_or,
    A                => A_out,
    B                => B_out,
    C                => C_out,
    D                => D_out);
```

The final part of the testbench includes the process which is used to define a test environment where specific values are set to test the components intended behavior. The process loops through all possible combinations of the signals “in_ax” and “in_opcode” and cycles the clock signal “clk” between 1 and 0 for every iteration. In additional to this the process also sets the update signal to a constant 1, and a start value for ax to “0000”.

```
process
   variable ax_vector, opcode_vector : std_logic_vector(3 downto
0):=(others=>'0');
begin  -- process
  in_update <= '1';
  in_ax <= "0000";
  for ax_vector in 0 to 15 loop
    for opcode_vector in 0 to 15 loop
      in_ax <= conv_std_logic_vector(ax_vector,4);
      in_opcode <= conv_std_logic_vector(opcode_vector,4);
      in_clk  <= '1';
      wait for 10 ns;
      in_clk  <= '0';
      wait for 10 ns;
    end loop;
  end loop;
end process;
end behave;
```

Simulating the CPU

When simulating the CPU, it can be verified that each signal behaves as intended. Going through each of the opcodes to see if they perform the desired action, for all possible inputs. Looking at the waveform below the yellow line is placed at the start of a cycle of opcodes. The result will be read out from C_out and D_out, which are the signals connected to the ports C and D. The five last rows of signals are the output signals for each of the arithmetic operation components. They can be used to verify that the opcode performs the correct operation and gives the correct output, the same as the corresponding component.


Name
Value
in_clk 1
in_update 1
in_ax[3:0] 4
in_opcode[3:0] 0
input1[3:0] 3
input2[3:0] 10
A_out[3:0] 4
B_out[3:0] 4
C_out[3:0] 11
D_out[3:0] 0
out_inv[3:0] 12
out_and[3:0] 2
out_or[3:0] 11
out_adder[4:0] 13
out_mult[7:0] 30
6,300.000 ns
6,400.000 ns
6,500.000 ns
6,600.000 ns
6,700.000 ns
6,800.000 ns