Khaled Hamza
2024-02-26

ET061G

Experiment 4 in Electrical engineering, ET061G

by Khaled Hamza

Mittuniversitetet

MID SWEDEN UNIVERSITY

Purpose:

The purpose of this lab is to construct a CPU component by reusing elements from previous labs. The CPU operates using a 4-bit opcode as its instruction set, with execution speed controlled by a clock pulse (clk). The CPU sends 4-bit values (via op_in1 and op_in2) to an Arithmetic Logic Unit (ALU) to perform operations based on the opcode. The goal is to ensure proper handling of data assignments and calculations to avoid errors caused by concurrent execution in VHDL, such as using outdated values instead of updated ones.

Theory:

The CPU design (Figure 1) uses a 4-bit opcode to define instructions. The instruction set is divided into two categories:

1. **Opcodes 0–5**: Assign predefined values to outputs.
2. **Opcodes 6–15**: Perform calculations using operators (e.g., addition, subtraction).

In VHDL, all code within a process block executes **concurrently**, meaning values assigned in the same block cannot be immediately used for calculations unless managed carefully. For example, to compute A + B (opcode 6), values for A and B must first be assigned in earlier steps. If not handled properly, operations may rely on stale data. To resolve this, two methods are proposed:

- **Synchronized timing**: Assign variables on the **rising edge** of the clock and perform calculations on the **falling edge**.
- **Multi-clock processing**: Use additional clock pulses to separate assignment and calculation phases for different opcodes.

Implementation:

1. **Variable Assignment on Clock Edges**:
    - Declare variables to store intermediate values (e.g., A and B).
    - Assign values to these variables on the **rising edge** of the clock.
    - Perform ALU calculations (e.g., A + B) on the **falling edge** to ensure updated values are used.
2. **Multi-Step Clock Processing**:
    - Use additional clock cycles to handle complex opcodes. For example:
        - **Cycle 1**: Assign values to A and B (opcodes 0–5).
        - **Cycle 2**: Execute the ALU operation (opcodes 6–15) using the updated values.

This ensures that assignments and calculations are temporally separated, preventing race conditions and ensuring correct results. Proper synchronization is critical to avoid dependencies on outdated data in concurrent VHDL execution.

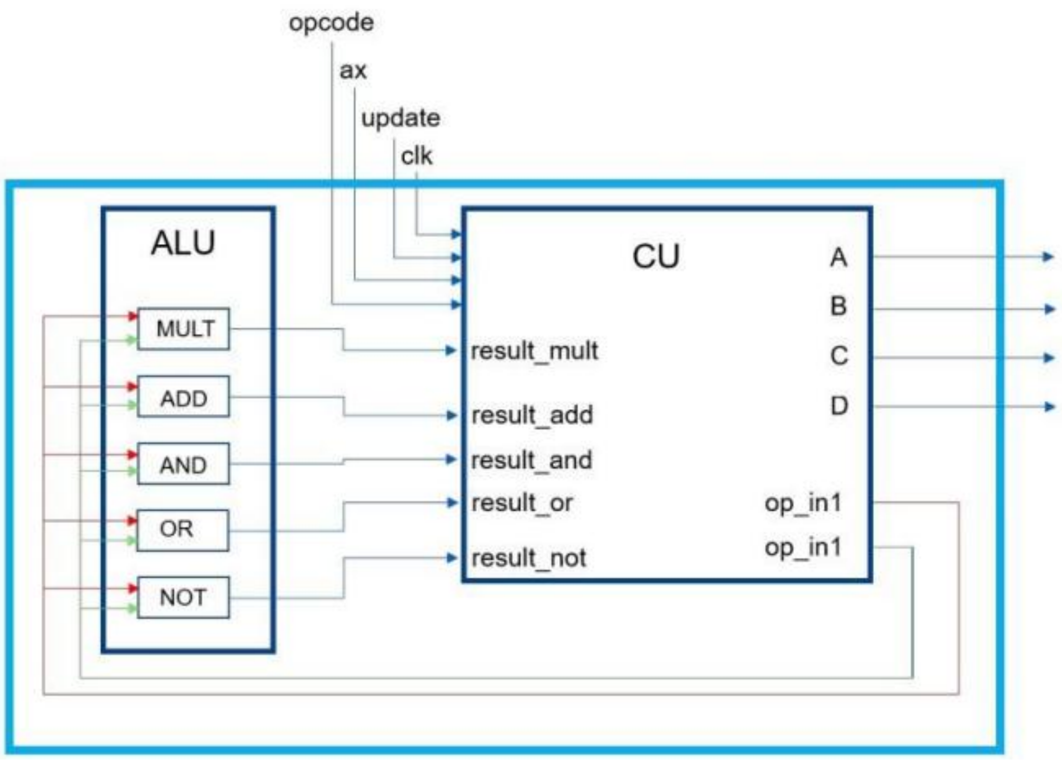

Figure 1: Images of the design of the CPU. It is taken from lab instruction.

1.1.1. CPU code

The entity section of the CPU code is designed according to the lab instructions, following the CPU architecture illustrated in Figure 1.

In this architecture, a process(clk) is used to execute instructions based on the opcode when the clock pulse changes. Within this process, two if-statements control the timing of command execution:

1. The first if-statement triggers on the rising edge of the clock pulse. It uses a case statement to assign values to various internal signals or variables.

2. The second if-statement executes on the falling edge of the clock pulse. Here, a case opcode structure directs arithmetic and logic operations in the ALU, as depicted in Figure 1.

This dual-edge approach ensures calculations in the ALU use updated variable values (from the rising edge) rather than outdated ones, improving result accuracy. By separating variable updates (rising edge) and ALU operations (falling edge), the design avoids race conditions and aligns with the architecture's timing requirements.

The full CPU code is provided below.

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.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;

architecture Behavioral of cpu is

signal BX, CX, DX : STD_LOGIC_VECTOR(3 downto 0) :=(others=>'0');

signal op_1, op_2 : STD_LOGIC_VECTOR(3 downto 0) := (others=>'0');

begin

A <= ax;

B <= BX;

C <= CX; D <= DX; op_in1 <= op_1; op_in2 <= op_2; process(clk) begin

if rising_edge(clk) and update = '1' then

case opcode is

when "0000" =>
```

```
BX <= ax; when

"0001" => CX

<= ax; when

"0010" => DX

<= ax; when

"0011" => BX

<= "0000"; when

"0100" => CX

<= "0000"; when

"0101" => DX

<= "0000"; when

"0110" => op_1

<= "0010"; op_2

<= "0011"; when

"0111" => op_1

<= "0010"; op_2

<= "0011"; when

"1000" => op_1

<= "0010"; when

"1001" => op_1

<= "0010"; op_2

<= "0011"; when

"1010" => op_1

<= "0010"; op_2

<= "0011"; when

"1011" => op_1

<= ax; op_2 <=

BX; when "1100"

=> op_1 <= ax;

op_2 <= BX; when

"1101" => op_1
```

```
<= ax; when

"1110" => op_1

<= ax; op_2 <=

BX; when "1111"

=> op_1 <= ax;

op_2 <= BX; when

others => null;

end case; end if;

if falling_edge(clk) then

case opcode is

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";

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";

when others =>

null; end case;

end if;

end process;

end Behavioral;
```

1.1.2. Test bench

The testbench file functions similarly to those used in Labs 2 and 3, with the key distinction that this testbench simulates a clock pulse. It begins by declaring the necessary components and ports using component and port map. Following this, signal variables required for the simulation are declared, matching the component's ports. To ensure thorough verification, a for-loop is implemented to test all combinations of CPU opcodes. Within this loop, the clock pulse is controlled with a period of 20 ns (10 ns high and 10 ns low), simulating realistic timing conditions. By iterating through all opcodes while generating the clock signal, the testbench rigorously validates the CPU's behavior across various scenarios and timing constraints, ensuring the design meets technical specifications.

The full code for the test bench is provided below.

```
library IEEE; use

IEEE.STD_LOGIC_1164.ALL; use

IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity cpu_tb is

end cpu_tb;

architecture Behavioral of cpu_tb is

component add4 port(in_1: in

STD_LOGIC_VECTOR(3 downto 0); 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
STD_LOGIC_VECTOR(3 downto 0); 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
STD_LOGIC_VECTOR(3 downto 0); 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 STD_LOGIC_VECTOR(3 downto 0);
in_2: in STD_LOGIC_VECTOR(3 downto 0); o_out:
out STD_LOGIC_VECTOR(3 downto 0));
end component;
component cpu 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 component;

signal clk_in: STD_LOGIC; signal update_in: STD_LOGIC;

signal ax_in: STD_LOGIC_VECTOR(3 downto 0); signal

opcode_in: STD_LOGIC_VECTOR(3 downto 0); signal add_1:

STD_LOGIC_VECTOR(4 downto 0); signal mult_1:

STD_LOGIC_VECTOR(7 downto 0); signal inv_1:

STD_LOGIC_VECTOR(3 downto 0); signal and_1:

STD_LOGIC_VECTOR(3 downto 0); signal or_1:

STD_LOGIC_VECTOR(3 downto 0); signal input1:

STD_LOGIC_VECTOR(3 downto 0); signal input2:

STD_LOGIC_VECTOR(3 downto 0); signal ax_temp:

STD_LOGIC_VECTOR(3 downto 0):="0000";

signal BX, CX, DX: STD_LOGIC_VECTOR(3 downto 0):="0000";

begin cpu_tb : cpu port

Map( clk => clk_in,

update => update_in,

AX => ax_in, opcode

=> opcode_in,

result_add => add_1,

result_mult => mult_1,

result_not => inv_1,

result_and => and_1,

result_or => or_1,

A => ax_temp,

B => BX,
C => CX, D => DX, op_in1 => input1,
op_in2 => input2
);
add4_0: add4 port map(
in_1 => input1, in_2
=> input2, s_out =>
add_1
);
mul4_0: mul4 port map(
in_1 => input1, in_2
=> input2,
m_out => mult_1
);
invrt4_0: invrt4 port map(
in_1 => input1, i_out =>
inv_1
);
bit_and4_0: bit_and4 port map(
in_1 => input1, in_2 =>
input2,
a_out => and_1
);
bit_or4_0: bit_or4 port map(
in_1 => input1, in_2 =>
input2,
o_out => or_1
);
process
variable a_vector : std_logic_vector(3 downto 0):=(others=>'0');
begin --process
update_in <= '1'; ax_in <=
```

"0000"; for a_vector in 0

to 15 loop

opcode_in <= conv_std_logic_vector(a_vector,4);

clk_in <= '1'; wait for 10 ns; clk_in <= '0'; wait

for 10 ns; end loop; -- a-vector end process; end

Behavioral;

Results and Discussions:

The results of running the testbench file are shown in Figure 2. The simulation illustrates the opcode transitioning from **0000 to 1111** (binary), covering all 16 possible combinations. As seen in the waveform, all commands defined in the lab instructions execute correctly, confirming that the CPU operates as expected.

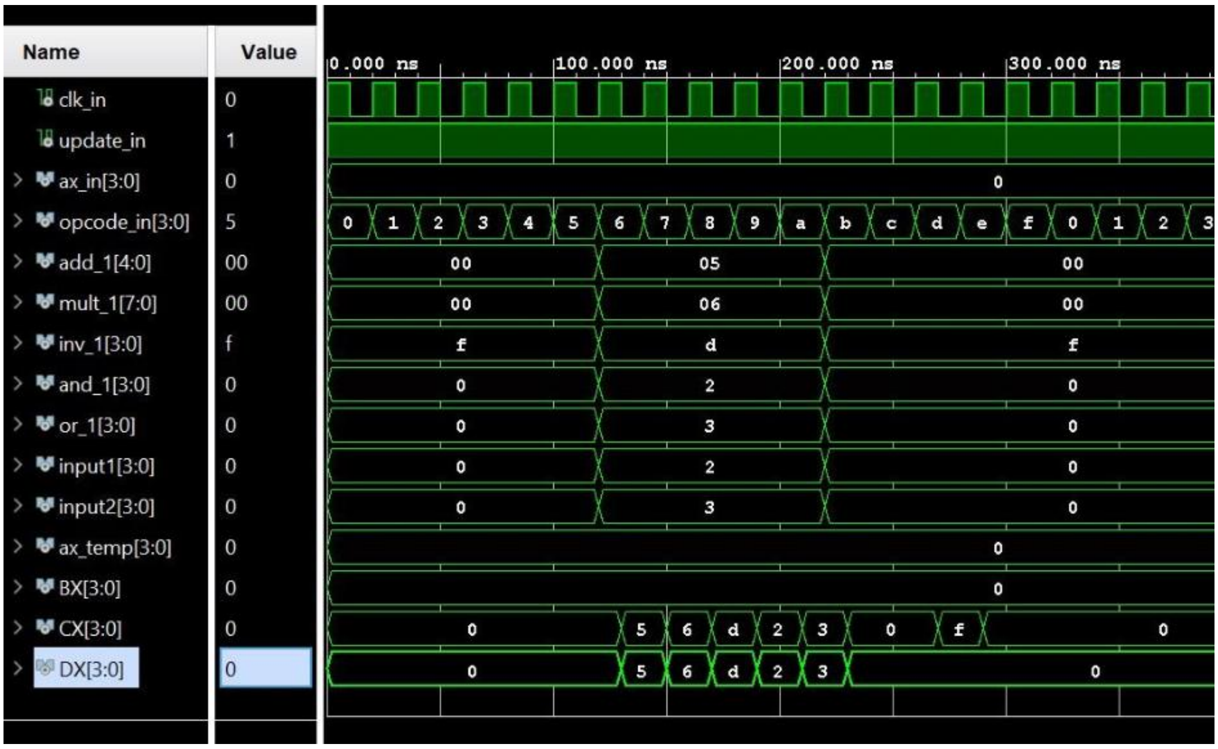

Figure 2: Simulation result of the cpu test bench.

References:

https://www.xilinx.com/support/download.html

https://docs.xilinx.com/v/u/2019.2-English/ug901-vivado-synthesis