UF Herbert Wertheim College of Engineering
Department of Electrical & Computer Engineering
UNIVERSITY of FLORIDA

Michael White
11/9/25
EEL3701C - Fall 2025

Lab 6: CPU Design

Requirements Not Met

N/A

Problems Encountered

Getting the clocks to work properly for NANDA was a bit of a hassle and I still struggle to demo it properly. Not sure if its user error or code error.

Applications

CPUs are literally used by all modern computing to solve complex problems. Knowing how to program them not only makes you valuable as an electrical engineer because you can make thinking rocks, but you also understand clocks, instructions, registers, etc. that are applicable to all sorts of hardware like ASICs.

Herbert Wertheim College of Engineering
Department of Electrical & Computer Engineering
UNIVERSITY of FLORIDA

Michael White
11/9/25
EEL3701C - Fall 2025

Pre-Lab Questions

Part 1 – The Controller

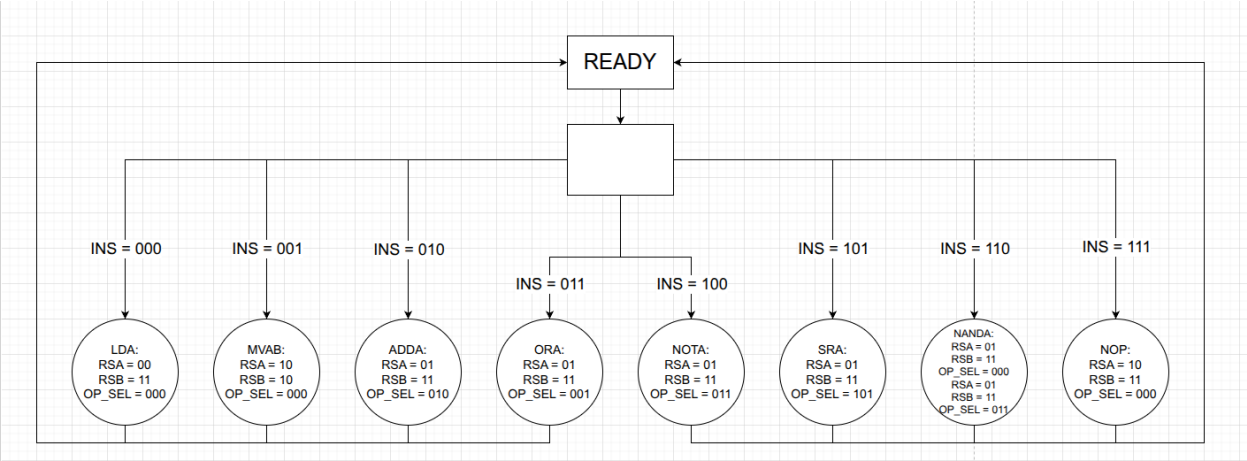

Figure 1: Instruction Flowchart

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

entity controller is
    port (
        clk   :  in std_logic;
        rst   :  in std_logic;  -- asynchronous, active-low
        ins   :  in std_logic_vector(2 downto 0);

        ready_out   :  out std_logic;
        rsa         :  out std_logic_vector(1 downto 0);
        rsb         :  out std_logic_vector(1 downto 0);
        op_sel      :  out std_logic_vector(2 downto 0)
    );
end entity controller;

architecture behavioral of controller is
    type t_state is (READY, EXECUTE);
    signal state   : t_state;
    signal step    : std_logic := '0';  -- indicates which step of NANDA its on
begin
    process(clk, rst) is
    begin
        if rst = '0' then
            -- reset values hold register info
            state <= READY;
            ready_out <= '1';
            rsa <= "10";
            rsb <= "11";
            op_sel <= "000";
            step <= '0';
        elsif rising_edge(clk) then
            case state is
                when READY =>
                    step <= '0';
                    ready_out <= '1';
                    rsa <= "10";
                    rsb <= "11";
                    op_sel <= "000";
                    -- should I hold until an instruction is recieved? What does that even mean?
                    if ins = "111" then
                        state <= READY;
                    else
                        state <= EXECUTE;
                    end if;
                when EXECUTE =>
                    ready_out <= '0';
                    -- LDA
                    if (ins = "000") then
                        rsa <= "00";
                        rsb <= "11";
                        op_sel <= "000";
                        state <= READY;
                    -- MVAB
                    elsif (ins = "001") then
                        rsa <= "10";
                        rsb <= "10";
                        op_sel <= "000";
                        state <= READY;
                    -- ADDA
                    elsif (ins = "010") then
                        rsa <= "01";
                        rsb <= "11";
                        op_sel <= "010";
                        state <= READY;
                    -- ORA
                    elsif (ins = "011") then
                        rsa <= "01";
                        rsb <= "11";
                        op_sel <= "001";
                        state <= READY;
                    -- NOTA
                    elsif (ins = "100") then
                        rsa <= "01";
                        rsb <= "11";
                        op_sel <= "011";
                        state <= READY;
                    -- SRA
                    elsif (ins = "101") then
                        rsa <= "01";
                        rsb <= "11";
                        op_sel <= "101";
                        state <= READY;
                    -- NANDA
                    elsif (ins = "110") then
                        if (step = '0') then
                            rsa <= "01";
                            rsb <= "11";
                            op_sel <= "000";
                            step <= '1';
                            state <= EXECUTE;
                        else
                            rsa <= "01";
                            rsb <= "11";
                            op_sel <= "011";
                            step <= '0';
                            state <= READY;
                        end if;
                    -- NOP
                    else
                        rsa <= "10";
                        rsb <= "11";
                        op_sel <= "000";
                        step <= '0';
                        state <= READY;
                    end if;
            end case;
        end if;
    end process;
end architecture behavioral;
```

Figure 2: VHDL for the controller

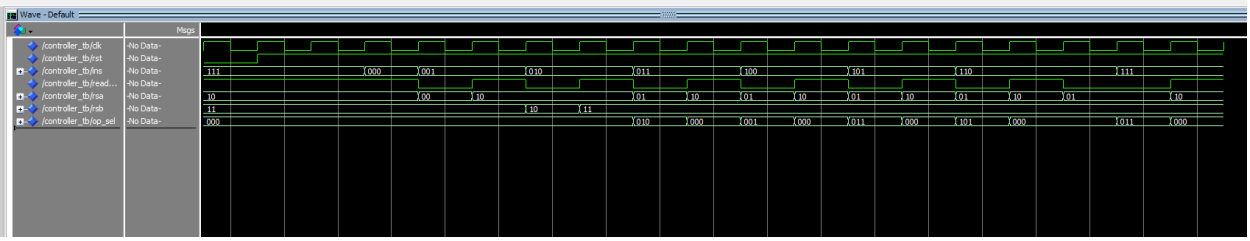

Figure 3: ModelSim Waveform Testbench for the Controller

Annotations:

- All instructions (000 through 111) are shown.
- All RSA, RSB, and OP_SEL values line up with the expected values in the flowchart.
- The output values update right after the rising edge of the clock.
- Ready is true, except during the execution of instructions.

Part 2 – Instruction Register and Connections

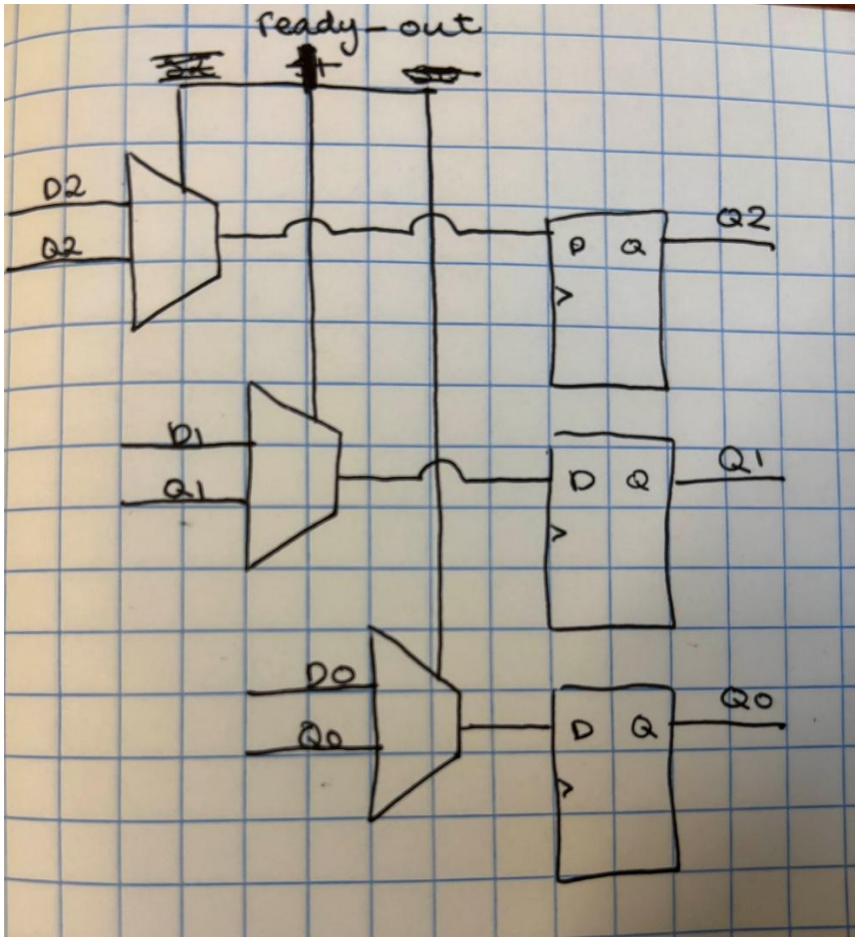

Figure 4: Hand-drawn Block Diagram for the Instruction Register

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

entity instruction_register is
   port (
      clk      :  in std_logic;
      update   :  in std_logic;  -- ready_out
      next_ins :  in std_logic_vector(2 downto 0);

      ins    :  out std_logic_vector(2 downto 0)
   );
end entity instruction_register;

architecture behavioral of instruction_register is
   signal ins_reg : std_logic_vector(2 downto 0);
begin
   process(clk)
   begin
      if rising_edge(clk) then
         if (update = '1') then
            ins_reg <= next_ins;
         end if;
      end if;
   end process;

   ins <= ins_reg; -- Apparently this models register/output behavior better?
end architecture;
```

Figure 5: VHDL for the Instruction Register

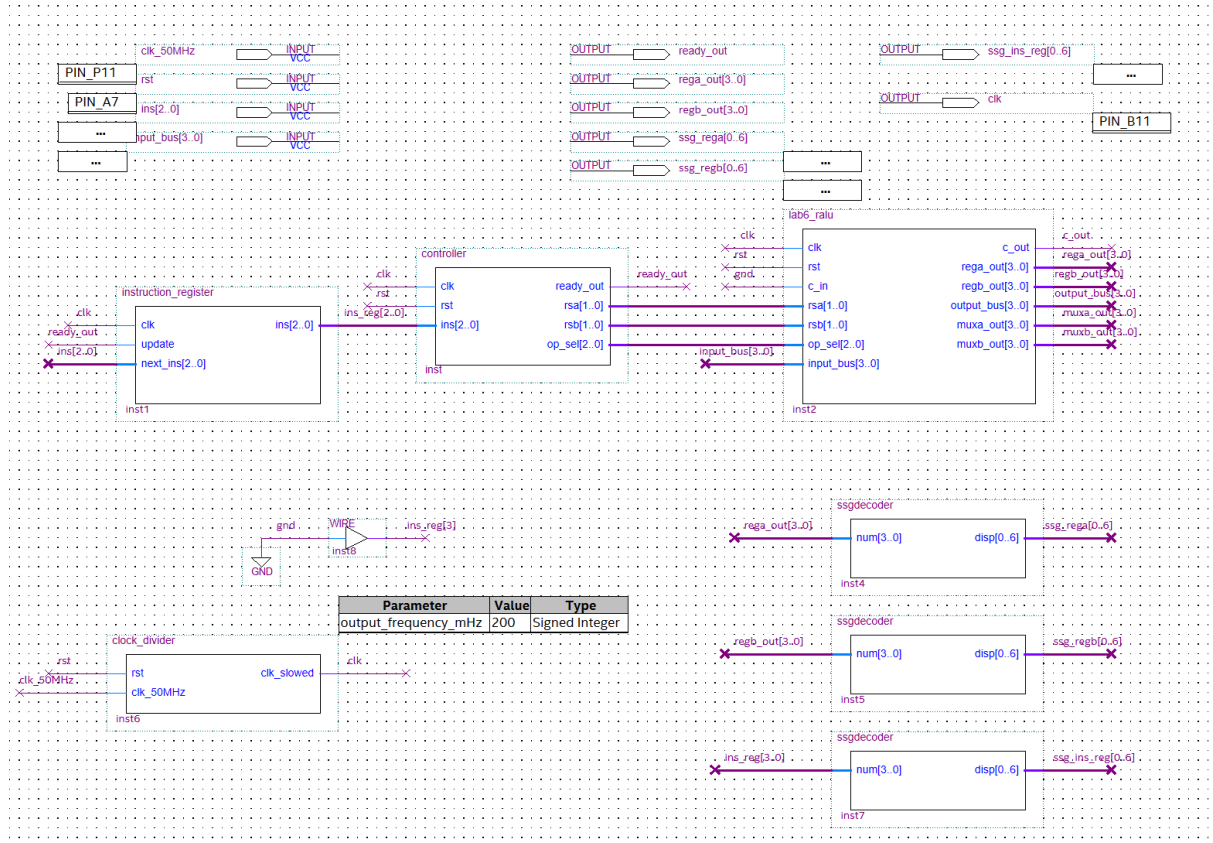

Figure 6: Block Diagram for the CPU

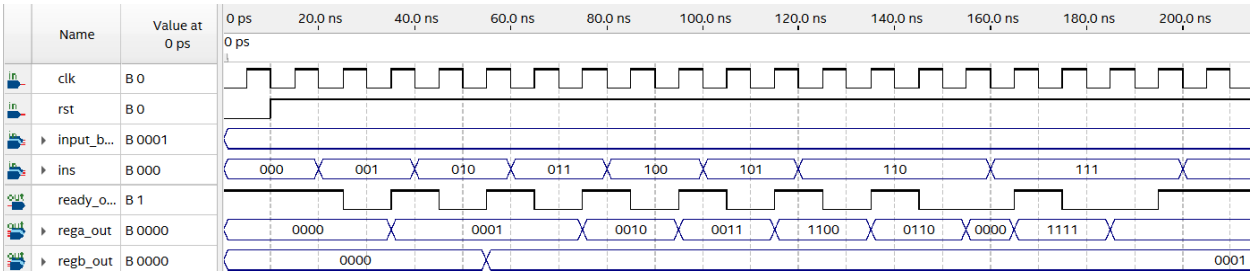

Figure 7: Waveform Simulation for the CPU

Annotations:

- Although not shown above, the input_bus = 0001.
- 000 (Load A) correctly updates register A to the value of the input bus (0001) on the rising edge of the clock and sets ready_out to 0.
- 001 (Move A) correctly updates register B to the value of register A (0001).
- 010 (Add A) correctly updates register A to 0001 + 0001 = 0010.
- 011 (Or A) correctly updates register A to 0010 or 0001 = 0011.
- So on, and so forth.

Part 3 - Programming

Instruction	Input Bus	REGA	REGB
LDA (000)	0101 (5)	0101 (5)	?
MVAB (001)	0101 (5)	0101 (5)	0101 (5)
LDA (000)	1100 (C)	1100 (C)	0101 (5)
ORA (011)	1100 (C)	1101 (D)	0101 (5)
MVAB (001)	1100 (C)	1101 (D)	1101 (D)
LDA (000)	0011 (3)	0011 (3)	1101 (D)
NANDA (110) – step1	0011 (3)	0001 (1)	1101 (D)
NANDA (110) – step2	0011 (3)	1110 (E)	1101 (D)
NOTA (100)	0011 (3)	0001 (1)	1101 (D)
MVAB (001)	0011 (3)	0001 (1)	0001 (1)
LDA (000)	0111 (7)	0111 (7)	0001 (1)
ADDA (010)	0111 (7)	1000 (8)	0001 (1)

Table 1: Assembly Program Chart for REGA = 7 + (3 AND (5 OR C))

1. The advantage of making the controller a state machine, rather than combinational logic is that everything in sync, so that we know when to expect and update values and so that the register and alu can be feed the right values at the right time. Essentially, it keeps everything clean and helps prevent glitches.

2.

Instruction	Input Bus	REGA	REGB
LDA (000)	1110 (E)	1110 (E)	?
SRA (101)	1110 (E)	0111 (7)	?
MVAB (001)	1110 (E)	0111 (7)	0111 (7)
LDA (000)	0101 (5)	0101 (5)	0111 (7)
ORA (011)	0101 (5)	0111 (7)	0111 (7)
NOTA (100	0101 (5)	1000 (8)	0111 (7)
MVAB (001)	0101 (5)	1000 (8)	1000 (8)
LDA (000)	0110 (6)	0110 (6)	1000 (8)
ADDA (010)	0110 (6)	1110 (14)	1000 (8)
MVAB (001)	0110 (6)	1110 (14)	1110 (14)

Table 2: Assembly Program Chart for REGB = 6 + (5 NOR (E / 2))