COEN 316
Computer Architecture and Design

Lab Section: DJ-X

Lab #4: CPU

Kevin Mandiouba

Lab Instructor: Anatoly Syutkin Due

Date: December 2nd, 2025

"I certify that this submission is my original work and meets the Faculty's Expectations of Originality"

Kevin Mandiouba

December 2nd, 2025

Table of Contents

Table of Figures

1. Objectives

The objective of this lab was to complete our CPU. This includes the assembly of all the components designed in the previous labs, together with the remaining components needed for the full design. These components include the PC register, sign extension unit, I-Cache, and D-Cache. The assembly would then be followed by the incorporation of the Control Unit to produce the corresponding control signals which depend on the opcode and function. The resulting process would then be followed by the verification that the complete CPU executes all the instructions correctly using the I-Cache.

2. Design

In the lab, the design was in two phases. In the first phase, we completed the design of the datapath by including the remaining components in the design, such as the PC register, I-Cache, D-Cache, and Sign Extension, and then combined them with the components from the earlier labs. The completed design was then simulated using ModelSim by manually inputting the signals and testing the first four instructions available in our I-Cache.

The second part involved the implementation and integration of the Control Unit to the datapath in order to create the complete CPU. The Control Unit produces all the controls using the opcode and the function field, according to the specifications given in the lab handout.

2.1. Datapath

In our implementation of the datapath, we essentially adopted the layout provided in the lab 4 guide (Figure 1), and it involved several components that we had designed in the earlier lab sessions.

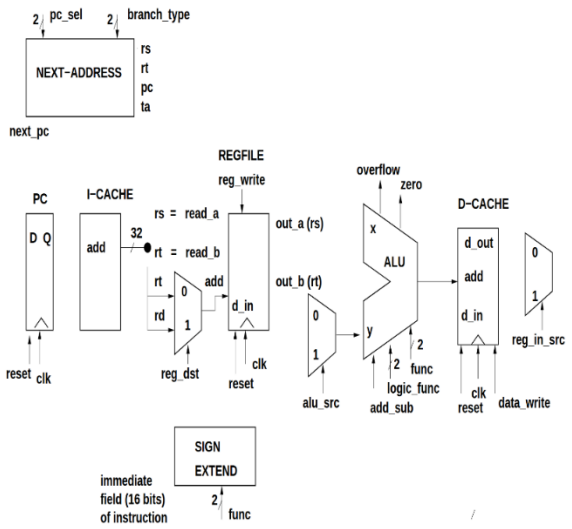


Figure 1: Lab handout's CPU Datapath

In this lab, we added the remaining components necessary to form the CPU and then interconnected them following the design provided in the guide.

2.1.1. PC register

The PC register adds a 32-bit register, whose operation takes place on the rise edge of the clock. The register resets to zero when the `reset = 1`, and the next PC value reloads every cycle, as illustrated in our `.vhd` file (Figure 2). The PC always operates on the 32 bits, even though only the lower portions is used for the lab, and only the lower 5 bits are used in the I-Cache in selecting the instructions. The output of the PC continuously goes to the I-Cache in order to fetch the next instruction.

2.1.2. Instruction cache (I-Cache)

The I-Cache stores our test MIPS program, which we use in testing the functionality of our design. The I-Cache, as described in Figure 3, takes the 32-bit PC input, where it uses the lower 5 bits, `pc(4 downto 0)`, to fetch an instruction from the instruction memory. Each address returns a pre-defined 32-bit instruction in our test program. This has helped us in manually testing different instructions in the datapath. The I-Cache output goes directly to the Register File, ALU, Sign Extension module, and Next-Address Unit through the datapath.

2.1.3. Register File

The Register File has 32 registers, each 32 bits long. The register file has two asynchronous read ports (`out_a` and `out_b`) and one synchronous write port. The register file read addresses (`read_a` and `read_b`) are directly taken from the instruction operands (`rs` and `rt`), and the register file write address is either `rt` or `rd`, depending on the value of `reg_dst`, has described in Figure 4. On each clock rising edge, when `reg_write = '1'`, the `din` signal's value gets written into the register. The purpose of this component is to pass the present values of the source registers to the ALU and collect the results either from the ALU operation or from the D-Cache. Like the PC the registers also get reset when `reset = '1'`.

2.1.4. Arithmetic Logic Unit (ALU)

The ALU is depicted in Figure 5. The ALU does all the arithmetic, logical, SLT, and LUI operations needed in our instruction set. The ALU takes in two inputs, each 32-bits. These inputs come from the register `rs` and either the `rt` register or the sign-extended immediate, based on the `alu_src` signal.

The operation performed on the inputs is determined through the use of the `add_sub`, `logic_func`, and `func` control signals. Besides obtaining the 32-bit output, the ALU produces the `zero` and `overflow` signals. The output goes either to the Register File for write-back or the D-Cache for the execution of the memory operation.

2.1.5. Sign Extension

The Sign Extension unit, depicted in Figure 6, sign-extends the 16-bit immediate field of the instruction, producing a 32-bit value. The operation depends on the two-bit `func` signal, where LUI sign-extends the immediate into the upper 16 bits and sets the lower bits to zero, arithmetic and SLT sign-extend the immediate, and logicals zero-extend the immediate.

Table 1: Sign extension formats

func	instruction type	sign extension	comments
00	load upper immediate	$i_{15}i_{14}i_{13}..i_1i_0$ 000..00	pad with 16 0s at least significant positions
01	set less immediate	$i_{15}i_{15}...$ $i_{15}i_{14}i_{13}..i_1i_0$	arithmetic sign extend (pad high order with copy of immediate sign bit i_{15})
10	arithmetic	$i_{15}i_{15}...$ $i_{15}i_{14}i_{13}..i_1i_0$	arithmetic sign extend (pad high order with copy of immediate sign bit i_{15})
11	logical	00...00 $i_{15}i_{14}i_{13}..i_1i_0$	high order 16 bits padded with 0s

The extension value now serves as the other input to the ALU when `alu_src = 1`. This enables immediate instructions in the datapath.

2.1.6. Data cache (D-Cache)

The Data Cache is depicted in Figure 7. The cache has a small 32×32 memory, where the ALU output sets the addresses, and the register `rt` value sets the input data in the store operation. When the clock goes high, the D-Cache stores the data in the memory when `data_write = 1`; else, the D-Cache produces the stored value at the specified address. The output from the D-Cache goes back to the Register File in the load operation when the `reg_in_src = 0`.

2.1.7. Next-address unit

The Next-Address Unit calculates the next PC value depending on the instruction type and the branching conditions. As can be seen in the code in Figure 8, it receives inputs from the present PC value, values of `rs` and `rt`, 26-bit target address, and control signals `branch_type` and `pc_sel`. Through selecting among these options, the next PC value is making sure to guide the flow of the CPU according to the type of instruction being read.

2.1.8. Assembly

The last step in the design process of the datapath was putting all the components together and making sure that the flow of data between the PC, the caches, the Register File, the ALU, the Sign Extension unit, and the Next-Address unit was correct. The datapath has the structure from the lab handout and is depicted in Figure 15.

Table 2: Single bit control signals

Control signal	value = 0	value = 1
reg_write	do not write into register file	write into register file
reg_dst	rt is the destination register	rd is the destination register
reg_in_src	d_out of data_cache is the d_in to the register file	ALU output is the d_in to the register file
alu_src	out_b of register file (rt) is the y input of the ALU	sign extended immediate is the y input of the ALU
add_sub	ALU operation = addition	ALU operation = subtraction
data_write	do not write into data cache	write into data cache

In order to check the functionality of the datapath, we manually provided the appropriate controls through the `.do` file. The different signals in the `.do` file and their interpretations are listed in Table 2 and Table 3. Table 2 lists the single-bit signals, and Table 3 lists the two-bit signals. These tables explain the flow of data in the datapath.

Table 3: Two bits control signals

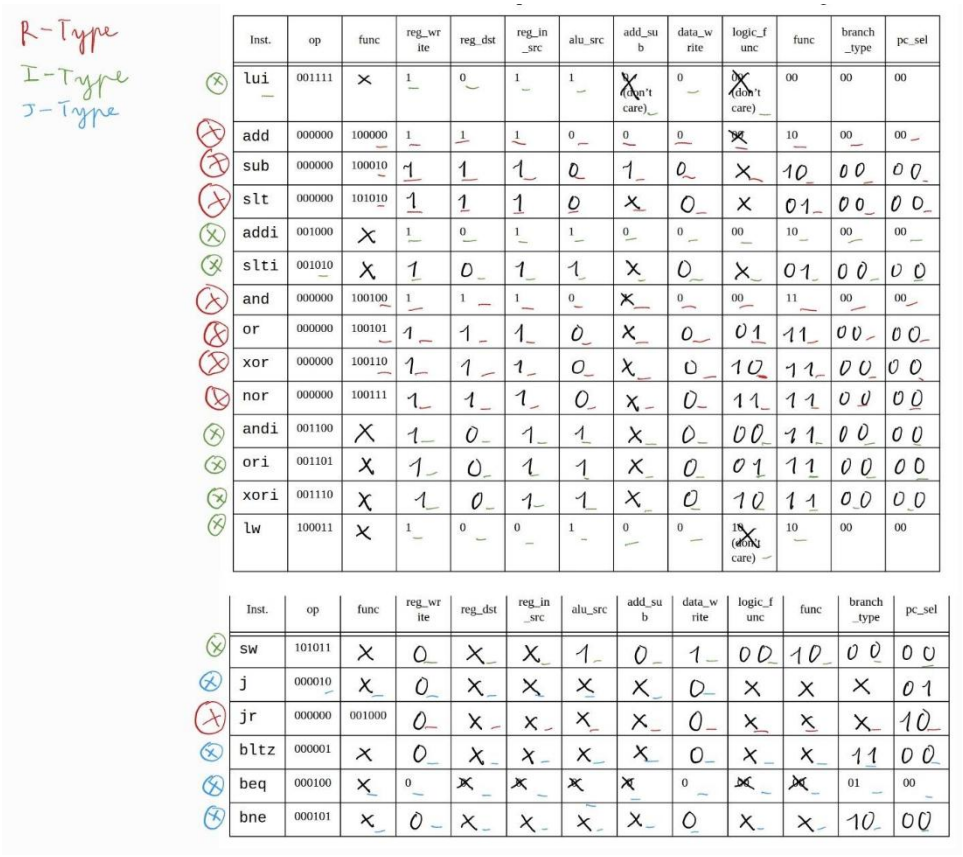
Control signal	value = 00	value = 01	value =10	value = 11
logic_func	AND	OR	XOR	NOR
func	load upper immediate	set less	arithmetic	logic
branch_type	no branch	beq	bne	bltz
pc_sel	no jump (PC+1, or PC+target address if branch condition is true)	jump (PC = target address)	jump register (PC = rs)	not used

Using the corresponding control values in the first four instructions in our I-Cache, we were able to verify our datapath before combining it with the Control Unit.

2.2. Control unit

The Control Unit produces all the control signals necessary for each instruction. The Control Unit has been designed based on the opcode and the function field, as summarized in Table 4. The Control Unit produces signals like `reg_write`, `alu_src`, `func`, `logic_func`, `branch_type`, and `pc_sel`, for each instruction, in order to make the datapath operate correctly.

Table 4: 20 instructions with opcode and function fields and control signals



Inst.	op	func	reg_write	reg_dst	reg_in_src	alu_src	add_sub	data_write	logic_func	func	branch_type	pc_sel
lui	001111	X	1	0	1	1	X (don't care)	0	XX (don't care)	00	00	00
add	000000	100000	1	1	1	0	0	0	XX	10	00	00
sub	000000	100010	1	1	1	0	1	0	X	10	00	00
slt	000000	101010	1	1	1	0	X	0	X	01	00	00
addi	001000	X	1	0	1	1	0	0	00	10	00	00
slti	001010	X	1	0	1	1	X	0	X	01	00	00
and	000000	100100	1	1	1	0	X	0	00	11	00	00
or	000000	100101	1	1	1	0	X	0	01	11	00	00
xor	000000	100110	1	1	1	0	X	0	10	11	00	00
nor	000000	100111	1	1	1	0	X	0	11	11	00	00
andi	001100	X	1	0	1	1	X	0	00	11	00	00
ori	001101	X	1	0	1	1	X	0	01	11	00	00
xori	001110	X	1	0	1	1	X	0	10	11	00	00
lw	100011	X	1	0	0	1	0	0	10 (don't care)	10	00	00

Inst.	op	func	reg_write	reg_dst	reg_in_src	alu_src	add_sub	data_write	logic_func	func	branch_type	pc_sel
sw	101011	X	0	X	X	1	0	1	00	10	00	00
j	000010	X	0	X	X	X	X	0	X	X	X	01
jr	000000	001000	0	X	X	X	X	0	X	X	X	10
bltz	000001	X	0	X	X	X	X	0	X	X	11	00
beq	000100	X	0	X	X	X	X	0	XX	XX	01	00
bne	000101	X	0	X	X	X	X	0	X	X	10	00

Our code implements a logic that first checks the `opcode` and, in the case of R-type instructions, the `funct` field. Every case statement sets the corresponding control signals necessary for each instruction, as discussed earlier. The controls then directly connect to the datapath, as shown in the complete CPU design in Figure 16.

By driving the datapath in the Control Unit instead of manually forcing each control line, the Control Unit completes the CPU and enables the execution of an entire program written in the I-Cache.

3. Results and Discussion

All the test cases performed on the datapath and the entire CPU worked in ModelSim. The output was as expected in all the instructions in our program, and all the control signals worked as expected.

When satisfied with the execution process, the test on the entire CPU was performed using the Control Unit, which provided the signals. The waveforms obtained for both parts are available in the Appendix.

3.1. Datapath

In order to test our datapath, all the necessary signals were manually applied using our `.do` file. The first four instructions present in the I-Cache were tested, and each step of their execution was checked in the waveform and behaved as expected.

3.2. Control unit

After validating the datapath, the Control Unit was enabled so that all the control signals could be generated automatically using the opcode and the function fields. The entire program in the I-Cache was simulated. The actions performed on the Register File, ALU, Memory, and branching in the entire program corresponding to different types of instructions were all correct.

4. Question

Based on the FPGA resource utilization report (available in the Appendix) of individual CPU, it consumes 685 LUTs (1.08%), 1 221 registers (0.96%), and 421 slices (2.66%) on the FPGA. In terms of the scalability of the design, it can be observed that it is possible to insert five or ten of such CPUs in the given top-level designs such as `Five_CPUs` and `Ten_CPUs` (Figure 13 and Figure 14, respectively), in which each of the CPU components has been instantiated and labeled as `DONT_TOUCH` in order to ensure that Vivado doesn't optimize them away.

From the reports, the results are almost linear. In the case of five CPUs, it consumes 3 506 LUTs (5.53%) and 6 105 registers (4.81%) compared to the ten CPU case that consumes 6 987 LUTs (11.02%) and 12 210 (9.63%) registers. Slice usage also increases linearly from 3 982 slices for the five CPU case to 7 269 for the ten CPU case.

By following this trend, we can determine the total number of CPUs that can potentially be placed on the FPGA while disregarding interconnection. Ten CPUs use only about 11% and 45% of LUTs and slices respectively, indicating that about twenty CPUs can still be accommodated in the slice limitation budget. However, after that, the LUTs of the FPGA would be exceeded, as the replication of the CPU one hundred times would need over 63,400 LUTs.

5. Conclusion

In this lab, we were able to finish the design of our MIPS CPU by implementing the entire datapath and combining it with the Control Unit. All the functional units, including those from the past labs and the units designed in the present lab, performed the test program correctly. The simulations confirmed the functionality of our design, complying with the desired instruction cycle, and the arithmetic, logical, memory, and branching instructions were correctly performed.

6. Appendix

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity PC_REG is
port(   clk     : in std_logic;
        reset   : in std_logic;
        next_pc : in std_logic_vector(31 downto 0);
        pc      : out std_logic_vector(31 downto 0)
    );
end PC_REG;

architecture behavior of PC_REG is
begin
   process(clk, reset)
   begin
        if reset = '1' then
            pc <= (others => '0');
        elsif rising_edge(clk) then
            pc <= next_pc;
        end if;
    end process;
end behavior;
```

Figure 2: PC_REG.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity I_CACHE is
   port(
      pc    : in  std_logic_vector(31 downto 0);
      instr : out std_logic_vector(31 downto 0)
   );
end I_CACHE;

architecture behavior of I_CACHE is
   signal address : std_logic_vector(4 downto 0);
begin

   address <= pc(4 downto 0);

   process(address)
   begin
      case address is

         -- 0: addi $1, $0, 5
         when "00000" =>  -- PC = 0
            instr <= "00100000000000010000000000000101";

         -- 1: addi $2, $0, 10
         when "00001" =>  -- PC = 1
            instr <= "00100000000000010000000000001010";

         -- 2: add $3, $1, $2   (r3 = 15)
         when "00010" =>  -- PC = 2
            instr <= "00000000001000100001100000100000";

         -- 3: sub $4, $2, $1   (r4 = 5)
         when "00011" =>  -- PC = 3
            instr <= "00000000010000100100000000100010";

         -- 4: sw $3, 0($0)  (mem[0] = 15)
         when "00100" =>  -- PC = 4
            instr <= "10101100000000110000000000000000";

         -- 5: lw $5, 0($0)  (r5 = 15, matches r3)
         when "00101" =>  -- PC = 5
            instr <= "10001100000001010000000000000000";

         -- 6: init r6 = 0x2000_0000 (no overflow on first add, overflow on second)
         when "00110" =>  -- PC = 6
            instr <= "00111100000001100010000000000000";

         -- 7: r6 = r6 + r6  (Loop for overflow)
         when "00111" =>  -- PC = 7
            instr <= "00000000110001100011000000100000";

         -- 8: beq r3, r5, +1  (15 == 15 → taken, skip PC9)
         when "01000" =>  -- PC = 8
            instr <= "00010000011001010000000000000001";

         -- 9: will be skipped if beq works
         when "01001" =>  -- PC = 9
            instr <= "00100000000001110000000000000001";

         -- 10: r7 = -1
         when "01010" =>  -- PC = 10
            instr <= "00100000000001111111111111111111";

         -- 11: bltz r7, +1 (taken, skip PC12)
         when "01011" =>  -- PC = 11
            instr <= "00000100111000000000000000000001";

         -- 12: skipped by bltz
         when "01100" =>  -- PC = 12
            instr <= "00100000000001100000000000101010";

         -- 13: j 7 (loop back to overflow instruction)
         when "01101" =>  -- PC = 13
            instr <= "00001000000000000000000000000111";

         -- Default: NOP
         when others =>
            instr <= (others => '0');  -- nop

      end case;
   end process;
end behavior;
```

Figure 3: I_CACHE.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity REGFILE is
port(   din             : in std_logic_vector(31 downto 0);
        reset           : in std_logic;
        clk             : in std_logic;
        write           : in std_logic;
        read_a          : in std_logic_vector (4 downto 0);
        read_b          : in std_logic_vector(4 downto 0);
        write_address   : in std_logic_vector(4 downto 0);
        out_a           : out std_logic_vector(31 downto 0);
        out_b           : out std_logic_vector(31 downto 0));
end REGFILE ;

architecture behavior of REGFILE is

        type register_array is array (31 downto 0)
                of std_logic_vector(31 downto 0);

        signal registers: register_array :=(others => (others => '0'));

begin

        out_a <= registers(to_integer(unsigned(read_a)));
        out_b <= registers(to_integer(unsigned(read_b)));

        process(clk, reset)
        begin
                if reset = '1' then
                        registers <=(others => (others => '0'));
                elsif rising_edge(clk) then
                        if write = '1' then
                                registers(to_integer(unsigned(write_address))) <= din;
                        end if;
                end if;
        end process;
end behavior;
```

Figure 4: REGFILE.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity ALU is
Port(x, y: in std_logic_vector(31 downto 0);

    --two input operands
    add_sub: in std_logic;
    -- 0 = add, 1 = sub

    logic_func: in std_logic_vector(1 downto 0);
    -- 00 = AND, 01 = OR, 10 = XOR, 11 = NOR)

    func: in std_logic_vector(1 downto 0);
    -- 00 = LUI, 01 = SETLESS, 10 = ARITH, 11 = LOGIC

    output: out std_logic_vector(31 downto 0);

    overflow: out std_logic;

    zero: out std_logic);

end ALU;

architecture behavior of ALU is

    signal a, b     : signed(31 downto 0);
    signal sum_diff : signed(31 downto 0);

    --Overflow
    signal add_o : std_logic;
    Signal sub_o : std_logic;

    -- Results
    signal arith_r  : std_logic_vector(31 downto 0);
    signal logic_r  : std_logic_vector(31 downto 0);
    signal slt_r    : std_logic_vector(31 downto 0);
    signal slt_diff : signed(31 downto 0);
    signal lui_r    : std_logic_vector(31 downto 0);

    constant ZERO32 : signed(31 downto 0) := (others => '0');

begin
    a <= signed(x);
    b <= signed(y);

    --Arithmetic
    with add_sub select
        sum_diff <= a + b when '0',
                    a - b when others;

    arith_r <= std_logic_vector(sum_diff);

    slt_diff <= a - b;
    slt_r    <= (31 downto 1 => '0') & slt_diff(31);

    -- Logic Function
    with logic_func select
    logic_r <=  (x and y)   when "00",
                (x or y)    when "01",
                (x xor y)   when "10",
                not(x or y) when others;

    -- LUI
    lui_r <= y;

    -- Output
    with func select
    output <=   lui_r   when "00",
                slt_r   when "01",
                arith_r when "10",
                logic_r when others;

    -- Flags

        -- Overflow
        add_o   <= '1' when (x(31) = y(31)) and (sum_diff(31) /= x(31)) else '0';
        sub_o   <= '1' when (x(31) /= y(31)) and (sum_diff(31) /= x(31)) else '0';
        overflow <= add_o when add_sub = '0' else sub_o;

        -- zero
        zero <= '1' when sum_diff = to_signed(0, 32) else '0';

end behavior;
```

Figure 5: ALU.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity SIGN_EXTEND is
port(   imm             : in std_logic_vector(15 downto 0);
        func            : in std_logic_vector(1 downto 0);
        sign_ext_out    : out std_logic_vector(31 downto 0)
    );
end SIGN_EXTEND;

architecture behavior of SIGN_EXTEND is
begin

    process(func, imm)
    begin
        case func is
            --LUI
            when "00" =>
                sign_ext_out(31 downto 16) <= imm;
                sign_ext_out(15 downto 0) <= (others => '0');

            --SLI and ARITH
            when "01" | "10" =>
                sign_ext_out <= std_logic_vector(resize(signed(imm), 32));

            --Logical
            when "11" =>
                sign_ext_out <= std_logic_vector(resize(unsigned(imm), 32));

            when others =>
                sign_ext_out <= (others => '0');
        end case;
    end process;
end behavior;
```

Figure 6: SIGN_EXTEND.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity D_CACHE is
port(   clk         : in std_logic;
        reset       : in std_logic;
        data_write  : in std_logic;
        addr        : in std_logic_vector(31 downto 0);
        d_in        : in std_logic_vector(31 downto 0);
        d_out       : out std_logic_vector(31 downto 0)
    );
end D_CACHE;

architecture behavior of D_CACHE is

    signal address: std_logic_vector(4 downto 0);

    type location_array is array (31 downto 0)
              of std_logic_vector(31 downto 0);

        signal locations: location_array :=(others => (others => '0'));
begin

    address <= addr(4 downto 0);
    d_out <= locations(to_integer(unsigned(address)));

    process(clk, reset)
    begin
        if reset = '1' then
            locations <= (others => (others => '0'));
        elsif rising_edge(clk) then
            if data_write = '1' then
                locations(to_integer(unsigned(address))) <= d_in;
            end if;
        end if;
    end process;
end behavior;
```

Figure 7: D_CACHE.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity NEXT_ADDRESS is
port(   rt, rs          : in std_logic_vector(31 downto 0);
        -- two register inputs
        pc              : in std_logic_vector(31 downto 0);
        target_address  : in std_logic_vector(25 downto 0);
        branch_type     : in std_logic_vector(1 downto 0);
        pc_sel          : in std_logic_vector(1 downto 0);
        next_pc         : out std_logic_vector(31 downto 0));
end NEXT_ADDRESS;

architecture behavior of NEXT_ADDRESS is

    signal offset       :  std_logic_vector(15 downto 0);
    signal sign_extend  :  signed(31 downto 0);
    signal jump_address :  std_logic_vector(31 downto 0);
    signal no_jump      :  std_logic_vector(31 downto 0);
    signal branching    :  signed(31 downto 0);
    signal reg_comp     :  std_logic;
    signal slt          :  std_logic;
begin

    -- Sign extend
    offset <= target_address(15 downto 0);
    sign_extend <= resize(signed(offset), 32);

    -- COMP rs, rt (BEQ = 1, BNE = 0)
    reg_comp <= '1' when (rs = rt) else '0';

    -- SLT (if rs < 0, slt = 1)
    slt <= '1' when (signed(rs) < 0) else '0';

    -- No Unconditioanl Jump
    process(branch_type, reg_comp, slt, sign_extend)
    begin
        case branch_type is
            when "00" => -- No Branch
                branching <= to_signed(1, 32);

            when "01" => -- BEQ
                if (reg_comp = '0') then branching <= to_signed(1, 32);
                else branching <= sign_extend + to_signed(1, 32);
                end if;

            when "10" => -- BNE
                if (reg_comp = '1') then branching <= to_signed(1, 32);
                else branching <= sign_extend + to_signed(1, 32);
                end if;

            when others => -- BLTZ
                if (slt = '0') then branching <= to_signed(1, 32);
                else branching <= sign_extend + to_signed(1, 32);
                end if;
        end case;
    end process;

    no_jump <= std_logic_vector(signed(pc) + branching);

    -- Jump to target address
    jump_address <= (5 downto 0 => '0') & target_address;

    -- PC_sel functionality
    with pc_sel select
    next_pc <=  no_jump         when "00",
                jump_address    when "01",
                rs              when "10",
                (others => '0') when others;
end behavior;
```

Figure 8: NEXT_ADDRESS.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity DATAPATH is
port(
    clk         : in std_logic;
    reset       : in std_logic;
    pc_sel      : in std_logic_vector(1 downto 0);
    branch_type : in std_logic_vector(1 downto 0);
    alu_src     : in std_logic;
    reg_dst     : in std_logic;
    reg_write   : in std_logic;
    reg_in_src  : in std_logic;
    data_write  : in std_logic;
    add_sub     : in std_logic;
    logic_func  : in std_logic_vector(1 downto 0);
    func        : in std_logic_vector(1 downto 0);
    instruction : out std_logic_vector(31 downto 0);
    alu_zero    : out std_logic;
    alu_overflow: out std_logic;
    rs_path     : out std_logic_vector(31 downto 0);
    rt_path     : out std_logic_vector(31 downto 0);
    pc_path     : out std_logic_vector(31 downto 0)
);
end DATAPATH;

architecture behavior of DATAPATH is
    signal pc_sig           : std_logic_vector(31 downto 0);
    signal next_pc_sig      : std_logic_vector(31 downto 0);

    signal rs, rt, rd       : std_logic_vector(4 downto 0);
    signal rs_out, rt_out   : std_logic_vector(31 downto 0);
    signal instr_sig        : std_logic_vector(31 downto 0);
    signal ta               : std_logic_vector(25 downto 0);
    signal imm_sig          : std_logic_vector(15 downto 0);
    signal sign_ext_sig     : std_logic_vector(31 downto 0);
    signal alu_out          : std_logic_vector(31 downto 0);
    signal data_out         : std_logic_vector(31 downto 0);
    signal alu_y            : std_logic_vector(31 downto 0);
    signal reg_din          : std_logic_vector(31 downto 0);
    signal reg_write_addr   : std_logic_vector(4 downto 0);
begin

    -- For control unit debugging
    rs_path <= rs_out;
    rt_path <= rt_out;
    pc_path <= pc_sig;

    ta <= instr_sig(25 downto 0);
    rs <= instr_sig(25 downto 21);
    rt <= instr_sig(20 downto 16);
    rd <= instr_sig(15 downto 11);
    imm_sig <= instr_sig(15 downto 0);

    instruction <= instr_sig;

    alu_y <= rt_out when alu_src = '0' else sign_ext_sig;

    reg_din <= data_out when reg_in_src = '0' else alu_out;

    reg_write_addr <= rt when reg_dst = '0' else rd;

    -- Program Counter (PC)
    U_PC_REG: entity work.PC_REG
     port map(
        clk     => clk,
        reset   => reset,
        next_pc => next_pc_sig,
        pc      => pc_sig
    );

    -- Next Address Unit
    U_NEXT_ADDRESS: entity work.NEXT_ADDRESS
     port map(
        rt             => rt_out,
        rs             => rs_out,
        pc             => pc_sig,
        target_address  => ta,
        branch_type     => branch_type,
        pc_sel          => pc_sel,
        next_pc         => next_pc_sig
    );

    -- I-Cache
    U_I_CACHE: entity work.I_CACHE
     port map(
        pc   => pc_sig,
```

```
        instr => instr_sig
    );

    -- Sign extend
    U_SIGN_EXTEND: entity work.SIGN_EXTEND
     port map(
        imm            => imm_sig,
        func           => func,
        sign_ext_out    => sign_ext_sig
    );

    -- Arithmethic Logic Unit
    U_ALU: entity work.ALU
     port map(
        x => rs_out,
        y => alu_y,
        add_sub => add_sub,
        logic_func => logic_func,
        func => func,
        output => alu_out,
        overflow => alu_overflow,
        zero => alu_zero
    );

    -- D-Cache(32x32 locations)
    U_D_CACHE : entity work.D_CACHE
    port map(
        clk        => clk,
        reset      => reset,
        data_write => data_write,
        addr       => alu_out,
        d_in       => rt_out,
        d_out      => data_out
    );

    -- Register File (32x32 registers)
    U_REGFILE: entity work.REGFILE
     port map(
        din            => reg_din,
        reset          => reset,
        clk            => clk,
        write          =>reg_write,
        read_a         => rs,
        read_b         => rt,
        write_address   => reg_write_addr,
        out_a          => rs_out,
        out_b          => rt_out
    );
end behavior;
```

Figure 9: DATAPATH.vhd

```
add wave clk
add wave reset
add wave pc_sel
add wave branch_type
add wave alu_src
add wave reg_dst
add wave reg_write
add wave reg_in_src
add wave data_write
add wave add_sub
add wave logic_func
add wave func

add wave instr_sig
add wave rs
add wave rt
add wave rd
add wave rs_out
add wave rt_out
add wave reg_write_addr
add wave reg_din
add wave alu_y
add wave alu_out
add wave data_out
add wave pc_sig
add wave next_pc_sig

radix hex

#--------------------------------
# Defaults (safe) from t = 0
#--------------------------------
force pc_sel      "00" 0ns
force branch_type "00" 0ns
force alu_src     0    0ns
force reg_dst     0    0ns
force reg_write   0    0ns
force reg_in_src  0    0ns
force data_write  0    0ns
force add_sub     0    0ns
force logic_func  "10" 0ns
force func        "00" 0ns
force reset 1
force clk 0
run 2

force reset 0
run 2

#--------------------------------
# addi $1, $0, 5
#--------------------------------
force reg_write '1';
force reg_dst    '0';
force reg_in_src '1';
force alu_src    '1';
force add_sub    '0';
force data_write '0';
force logic_func "00";
force func       "10";
force branch_type "00";
force pc_sel      "00";
run 2

force clk 1
run 2

#--------------------------------
# addi $2, $0, 10
#--------------------------------
force clk 0
force reg_write '1';
force reg_dst    '0';
force reg_in_src '1';
force alu_src    '1';
force add_sub    '0';
force data_write '0';
force logic_func "00";
force func       "10";
force branch_type "00";
force pc_sel      "00";
run 2

force clk 1
run 2

#--------------------------------
# add $3, $1, $2   (r3 = 15)
#--------------------------------
force clk 0
force reg_write '1';
force reg_dst   '1';
force reg_in_src '1';
```

```
force alu_src    '0';
force add_sub    '0';
force data_write '0';
#logic_func "00"; don't care
force func       "10";
force branch_type "00";
force pc_sel      "00";
run 2

force clk 1
run 2

#-------------------------------
# sub $4, $2, $1   (r4 = 5)
#-------------------------------
force clk 0
force reg_write '1';
force reg_dst   '1';
force reg_in_src '1';
force alu_src    '0';
force add_sub    '1';
force data_write '0';
#logic_func "00"; don't care
force func       "10";
force branch_type "00";
force pc_sel      "00";
run 2
```

Figure 10: DATAPATH.do

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity CPU is
port(
   reset : in std_logic;
   clk : in std_logic;
   rs_out, rt_out : out std_logic_vector(3 downto 0);
   --output ports from register file
   pc_out : out std_logic_vector(3 downto 0); --pc reg
   overflow, zero : out std_logic
);
end CPU;

architecture behavior of CPU is
   signal opcode           : std_logic_vector(5 downto 0);
   signal funct            : std_logic_vector(5 downto 0);

   signal pc_sig           : std_logic_vector(31 downto 0);
   signal rs_sig, rt_sig   : std_logic_vector(31 downto 0);
   signal pc_sel           : std_logic_vector(1 downto 0);
   signal branch_type      : std_logic_vector(1 downto 0);
   signal alu_src          : std_logic;
   signal reg_dst          : std_logic;
   signal reg_write        : std_logic;
   signal reg_in_src       : std_logic;
   signal data_write       : std_logic;
   signal add_sub          : std_logic;
   signal logic_func       : std_logic_vector(1 downto 0);
   signal func             : std_logic_vector(1 downto 0);
   signal instruction      : std_logic_vector(31 downto 0);
begin

   rs_out  <= rs_sig(3 downto 0);
   rt_out  <= rt_sig(3 downto 0);
   pc_out  <= pc_sig(3 downto 0);

   opcode  <= instruction(31 downto 26);
   funct   <= instruction(5 downto 0);

   process(opcode, funct)
   begin
      reg_write   <= '0';
      reg_dst     <= '0';
      reg_in_src  <= '0';
      alu_src     <= '0';
      add_sub     <= '0';
      data_write  <= '0';
      logic_func  <= "00";
      func        <= "00";
      branch_type <= "00";
      pc_sel      <= "00";

      -- What is commented out is don't care
      case opcode is

         -- R-type instructions
         when "000000" =>
            case funct is

               -- ADD
               when "100000" =>
               reg_write   <= '1';
               reg_dst     <= '1';
               reg_in_src  <= '1';
               alu_src     <= '0';
               add_sub     <= '0';
               data_write  <= '0';
               -- logic_func  <= "00";
               func        <= "10";
               branch_type <= "00";
               pc_sel      <= "00";

               -- SUB
               when "100010" =>
               reg_write   <= '1';
               reg_dst     <= '1';
               reg_in_src  <= '1';
               alu_src     <= '0';
               add_sub     <= '1';
               data_write  <= '0';
               -- logic_func  <= "00";
               func        <= "10";
               branch_type <= "00";
               pc_sel      <= "00";

               -- SLT
               when "101010" =>
               reg_write   <= '1';
               reg_dst     <= '1';
               reg_in_src  <= '1';
```

```
            alu_src     <= '0';
            -- add_sub     <= '1';
            data_write  <= '0';
            -- logic_func  <= "00";
            func        <= "01";
            branch_type <= "00";
            pc_sel      <= "00";

            -- AND
            when "100100" =>
            reg_write   <= '1';
            reg_dst     <= '1';
            reg_in_src  <= '1';
            alu_src     <= '0';
            -- add_sub     <= '1';
            data_write  <= '0';
            logic_func  <= "00";
            func        <= "11";
            branch_type <= "00";
            pc_sel      <= "00";

            -- OR
            when "100101" =>
            reg_write   <= '1';
            reg_dst     <= '1';
            reg_in_src  <= '1';
            alu_src     <= '0';
            -- add_sub     <= '1';
            data_write  <= '0';
            logic_func  <= "01";
            func        <= "11";
            branch_type <= "00";
            pc_sel      <= "00";

            -- XOR
            when "100110" =>
            reg_write   <= '1';
            reg_dst     <= '1';
            reg_in_src  <= '1';
            alu_src     <= '0';
            -- add_sub     <= '1';
            data_write  <= '0';
            logic_func  <= "10";
            func        <= "11";
            branch_type <= "00";
            pc_sel      <= "00";

            -- NOR
            when "100111" =>
            reg_write   <= '1';
            reg_dst     <= '1';
            reg_in_src  <= '1';
            alu_src     <= '0';
            -- add_sub     <= '1';
            data_write  <= '0';
            logic_func  <= "11";
            func        <= "11";
            branch_type <= "00";
            pc_sel      <= "00";

            -- Jump rs
            when "001000" =>
            reg_write   <= '0';
            -- reg_dst     <= '1';
            -- reg_in_src  <= '1';
            -- alu_src     <= '0';
            -- add_sub     <= '1';
            data_write  <= '0';
            -- logic_func  <= "10";
            -- func        <= "11";
            -- branch_type <= "00";
            pc_sel      <= "10";

            when others =>
            null;
        end case;

    -- I-type instructions
    when "001000" => -- ADDI
        reg_write   <= '1';
        reg_dst     <= '0';
        reg_in_src  <= '1';
        alu_src     <= '1';
        add_sub     <= '0';
        data_write  <= '0';
        logic_func  <= "00";
        func        <= "10";
        branch_type <= "00";
        pc_sel      <= "00";

    when "001010" => -- SLTI
        reg_write   <= '1';
        reg_dst     <= '0';
```

```
            reg_in_src  <= '1';
            alu_src     <= '1';
            -- add_sub     <= '0';
            data_write  <= '0';
            -- logic_func  <= "00";
            func        <= "01";
            branch_type <= "00";
            pc_sel      <= "00";

        when "001100" => -- ANDI
            reg_write   <= '1';
            reg_dst     <= '0';
            reg_in_src  <= '1';
            alu_src     <= '1';
            -- add_sub     <= '0';
            data_write  <= '0';
            logic_func  <= "00";
            func        <= "11";
            branch_type <= "00";
            pc_sel      <= "00";

        when "001101" => -- ORI
            reg_write   <= '1';
            reg_dst     <= '0';
            reg_in_src  <= '1';
            alu_src     <= '1';
            -- add_sub     <= '0';
            data_write  <= '0';
            logic_func  <= "01";
            func        <= "11";
            branch_type <= "00";
            pc_sel      <= "00";

        when "001110" => -- XORI
            reg_write   <= '1';
            reg_dst     <= '0';
            reg_in_src  <= '1';
            alu_src     <= '1';
            -- add_sub     <= '0';
            data_write  <= '0';
            logic_func  <= "10";
            func        <= "11";
            branch_type <= "00";
            pc_sel      <= "00";

        when "001111" => -- LUI
            reg_write   <= '1';
            reg_dst     <= '0';
            reg_in_src  <= '1';
            alu_src     <= '1';
            -- add_sub     <= '0';
            data_write  <= '0';
            -- logic_func  <= "00";
            func        <= "00";
            branch_type <= "00";
            pc_sel      <= "00";

        when "100011" => -- Load (lw)
            reg_write   <= '1';
            reg_dst     <= '0';
            reg_in_src  <= '0';
            alu_src     <= '1';
            add_sub     <= '0';
            data_write  <= '0';
            -- logic_func  <= "00";
            func        <= "10";
            branch_type <= "00";
            pc_sel      <= "00";

        when "101011" => -- Store (sw)
            reg_write   <= '0';
            -- reg_dst     <= '0';
            -- reg_in_src  <= '1';
            alu_src     <= '1';
            add_sub     <= '0';
            data_write  <= '1';
            -- logic_func  <= "00";
            func        <= "10";
            branch_type <= "00";
            pc_sel      <= "00";

        -- J-type instructions
        when "000010" => -- Jump target address (J)
            reg_write   <= '0';
            -- reg_dst     <= '0';
            -- reg_in_src  <= '0';
            -- alu_src     <= '1';
            -- add_sub     <= '0';
            data_write  <= '0';
            -- logic_func  <= "00";
            -- func        <= "10";
            -- branch_type <= "00";
            pc_sel      <= "01";
```

```
        when "000001" => -- BLTZ
            reg_write   <= '0';
            -- reg_dst     <= '0';
            -- reg_in_src  <= '0';
            -- alu_src     <= '1';
            -- add_sub     <= '0';
            data_write  <= '0';
            -- logic_func  <= "00";
            -- func        <= "10";
            branch_type <= "11";
            pc_sel      <= "00";

        when "000100" => -- BEQ
            reg_write   <= '0';
            -- reg_dst     <= '0';
            -- reg_in_src  <= '0';
            -- alu_src     <= '1';
            -- add_sub     <= '0';
            data_write  <= '0';
            -- logic_func  <= "00";
            -- func        <= "10";
            branch_type <= "01";
            pc_sel      <= "00";

        when "000101" => -- BNE
            reg_write   <= '0';
            -- reg_dst     <= '0';
            -- reg_in_src  <= '0';
            -- alu_src     <= '1';
            -- add_sub     <= '0';
            data_write  <= '0';
            -- logic_func  <= "00";
            -- func        <= "10";
            branch_type <= "10";
            pc_sel      <= "00";
        when others =>
        null;
    end case;
  end process;

  -- Datapath
  U_DATAPATH : entity work.DATAPATH
  port map(
    clk         => clk,
    reset       => reset,
    pc_sel      => pc_sel,
    branch_type => branch_type,
    alu_src     => alu_src,
    reg_dst     => reg_dst,
    reg_write   => reg_write,
    reg_in_src  => reg_in_src,
    data_write  => data_write,
    add_sub     => add_sub,
    logic_func  => logic_func,
    func        => func,
    instruction => instruction,
    alu_zero    => zero,
    alu_overflow=> overflow,
    rs_path     => rs_sig,
    rt_path     => rt_sig,
    pc_path     => pc_sig
  );

end behavior;
```

Figure 11: CPU.vhd

```
add wave clk
add wave reset
add wave zero
add wave overflow
add wave -unsigned pc_out
add wave -unsigned rs_out
add wave -unsigned rt_out

radix hex

force reset 1
force clk 0
run 2

force reset 0
run 2

force -repeat 10ns clk 0 0ns, 1 3ns

run 310ns
```

Figure 12: CPU.do

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_signed.all;

entity Five_CPUs is
port(   reset : in std_logic;
        clk     : in std_logic);
end Five_CPUs;


architecture behavior of Five_CPUs is
    component cpu is
    port(   reset           : in std_logic;
            clk             : in std_logic;
            rs_out, rt_out  : out std_logic_vector(3 downto 0);
            pc_out          : out std_logic_vector(3 downto 0);
            overflow, zero  : out std_logic);
    end component ;

    -- declare internal signals used in the component
    -- port map statements
    signal rs_out0, rs_out1, rs_out2, rs_out3, rs_out4  : std_logic_vector(3 downto 0);
    signal rt_out0, rt_out1, rt_out2, rt_out3, rt_out4 : std_logic_vector(3 downto 0) ;
    signal pc_out0, pc_out1, pc_out2, pc_out3, pc_out4 : std_logic_vector(3 downto 0) ;
    signal overflow0, overflow1, overflow2, overflow3, overflow4 : std_logic;
    signal zero0, zero1, zero2, zero3, zero4 : std_logic;

    for CPU_0, CPU_1, CPU_2, CPU_3, CPU_4 : cpu use entity WORK.cpu(behavior);
    attribute DONT_TOUCH : string;
    attribute DONT_TOUCH of CPU_0 : label is "TRUE";
    attribute DONT_TOUCH of CPU_1 : label is "TRUE";
    attribute DONT_TOUCH of CPU_2 : label is "TRUE";
    attribute DONT_TOUCH of CPU_3 : label is "TRUE";
    attribute DONT_TOUCH of CPU_4 : label is "TRUE";

begin
    -- component instantiation
    CPU_0: cpu port map(reset => reset, clk => clk, rs_out => rs_out0,
    rt_out => rt_out0, pc_out => pc_out0, overflow => overflow0, zero
    => zero0);

    CPU_1: cpu port map(reset => reset, clk => clk, rs_out => rs_out1,
    rt_out => rt_out1, pc_out => pc_out1, overflow => overflow1, zero
    => zero1);

    CPU_2: cpu port map(reset => reset, clk => clk, rs_out => rs_out2,
    rt_out => rt_out2, pc_out => pc_out2, overflow => overflow2, zero
    => zero2);

    CPU_3: cpu port map(reset => reset, clk => clk, rs_out => rs_out3,
    rt_out => rt_out3, pc_out => pc_out3, overflow => overflow3, zero
    => zero3);

    CPU_4: cpu port map(reset => reset, clk => clk, rs_out => rs_out4,
    rt_out => rt_out4, pc_out => pc_out4, overflow => overflow4, zero
    => zero4);
end behavior;
```

Figure 13: Five_CPUs.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_signed.all;

entity Ten_CPUs is
port(   reset : in std_logic;
        clk     : in std_logic);
end Ten_CPUs;


architecture behavior of Ten_CPUs is
    component cpu is
    port(   reset          : in std_logic;
            clk            : in std_logic;
            rs_out, rt_out  : out std_logic_vector(3 downto 0);
            pc_out          : out std_logic_vector(3 downto 0);
            overflow, zero  : out std_logic);
    end component ;

    -- declare internal signals used in the component
    -- port map statements
    signal rs_out0, rs_out1, rs_out2, rs_out3, rs_out4  : std_logic_vector(3 downto 0);
    signal rs_out5, rs_out6, rs_out7, rs_out8, rs_out9  : std_logic_vector(3 downto 0);

    signal rt_out0, rt_out1, rt_out2, rt_out3, rt_out4 : std_logic_vector(3 downto 0) ;
    signal rt_out5, rt_out6, rt_out7, rt_out8, rt_out9 : std_logic_vector(3 downto 0) ;

    signal pc_out0, pc_out1, pc_out2, pc_out3, pc_out4 : std_logic_vector(3 downto 0) ;
    signal pc_out5, pc_out6, pc_out7, pc_out8, pc_out9 : std_logic_vector(3 downto 0) ;

    signal overflow0, overflow1, overflow2, overflow3, overflow4 : std_logic;
    signal overflow5, overflow6, overflow7, overflow8, overflow9 : std_logic;

    signal zero0, zero1, zero2, zero3, zero4 : std_logic;
    signal zero5, zero6, zero7, zero8, zero9 : std_logic;

    for CPU_0, CPU_1, CPU_2, CPU_3, CPU_4,
        CPU_5, CPU_6, CPU_7, CPU_8, CPU_9 : cpu use entity WORK.cpu(behavior);
    attribute DONT_TOUCH : string;
    attribute DONT_TOUCH of CPU_0 : label is "TRUE";
    attribute DONT_TOUCH of CPU_1 : label is "TRUE";
    attribute DONT_TOUCH of CPU_2 : label is "TRUE";
    attribute DONT_TOUCH of CPU_3 : label is "TRUE";
    attribute DONT_TOUCH of CPU_4 : label is "TRUE";
    attribute DONT_TOUCH of CPU_5 : label is "TRUE";
    attribute DONT_TOUCH of CPU_6 : label is "TRUE";
    attribute DONT_TOUCH of CPU_7 : label is "TRUE";
    attribute DONT_TOUCH of CPU_8 : label is "TRUE";
    attribute DONT_TOUCH of CPU_9 : label is "TRUE";

begin
    -- component instantiation
    CPU_0: cpu port map(reset => reset, clk => clk, rs_out => rs_out0,
    rt_out => rt_out0, pc_out => pc_out0, overflow => overflow0, zero
    => zero0);

    CPU_1: cpu port map(reset => reset, clk => clk, rs_out => rs_out1,
    rt_out => rt_out1, pc_out => pc_out1, overflow => overflow1, zero
    => zero1);

    CPU_2: cpu port map(reset => reset, clk => clk, rs_out => rs_out2,
    rt_out => rt_out2, pc_out => pc_out2, overflow => overflow2, zero
    => zero2);

    CPU_3: cpu port map(reset => reset, clk => clk, rs_out => rs_out3,
    rt_out => rt_out3, pc_out => pc_out3, overflow => overflow3, zero
    => zero3);

    CPU_4: cpu port map(reset => reset, clk => clk, rs_out => rs_out4,
    rt_out => rt_out4, pc_out => pc_out4, overflow => overflow4, zero
    => zero4);

    CPU_5: cpu port map(reset => reset, clk => clk, rs_out => rs_out5,
    rt_out => rt_out5, pc_out => pc_out5, overflow => overflow5, zero
    => zero5);

    CPU_6: cpu port map(reset => reset, clk => clk, rs_out => rs_out6,
    rt_out => rt_out6, pc_out => pc_out6, overflow => overflow6, zero
    => zero6);

    CPU_7: cpu port map(reset => reset, clk => clk, rs_out => rs_out7,
    rt_out => rt_out7, pc_out => pc_out7, overflow => overflow7, zero
    => zero7);

    CPU_8: cpu port map(reset => reset, clk => clk, rs_out => rs_out8,
    rt_out => rt_out8, pc_out => pc_out8, overflow => overflow8, zero
    => zero8);

    CPU_9: cpu port map(reset => reset, clk => clk, rs_out => rs_out9,
    rt_out => rt_out9, pc_out => pc_out9, overflow => overflow9, zero
    => zero9);
end behavior;
```

Figure 14: Ten_CPUs.vhd

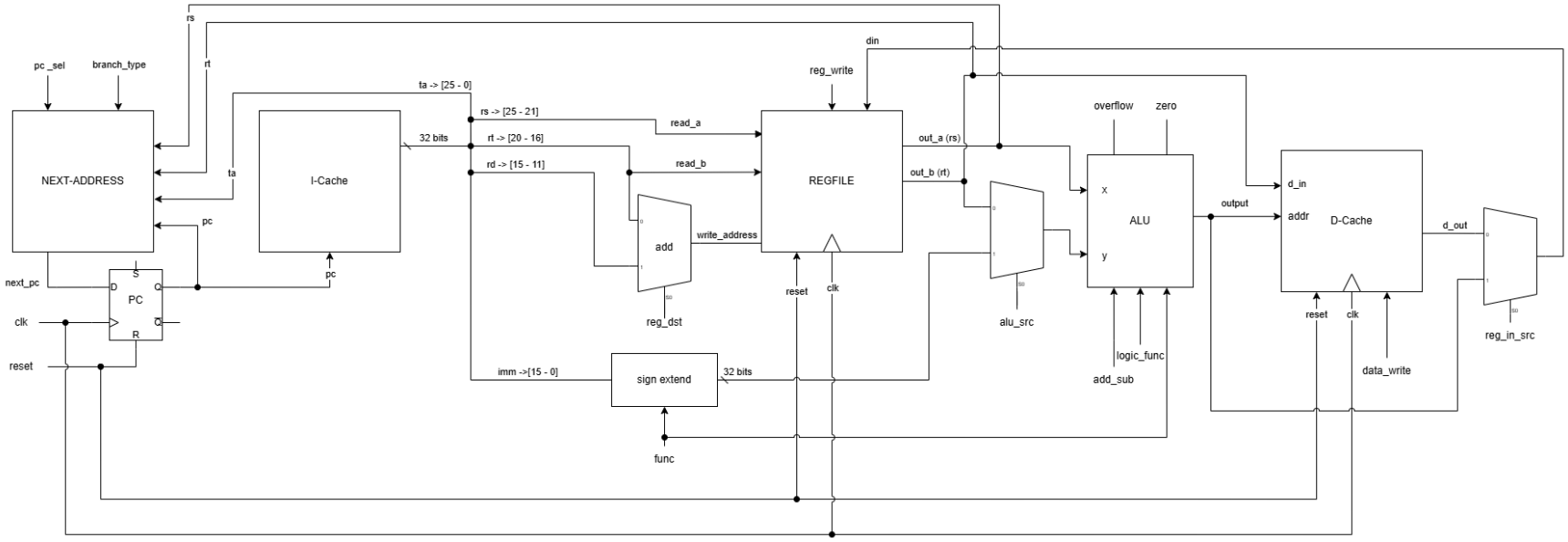


Figure 15: Complete Datapath design

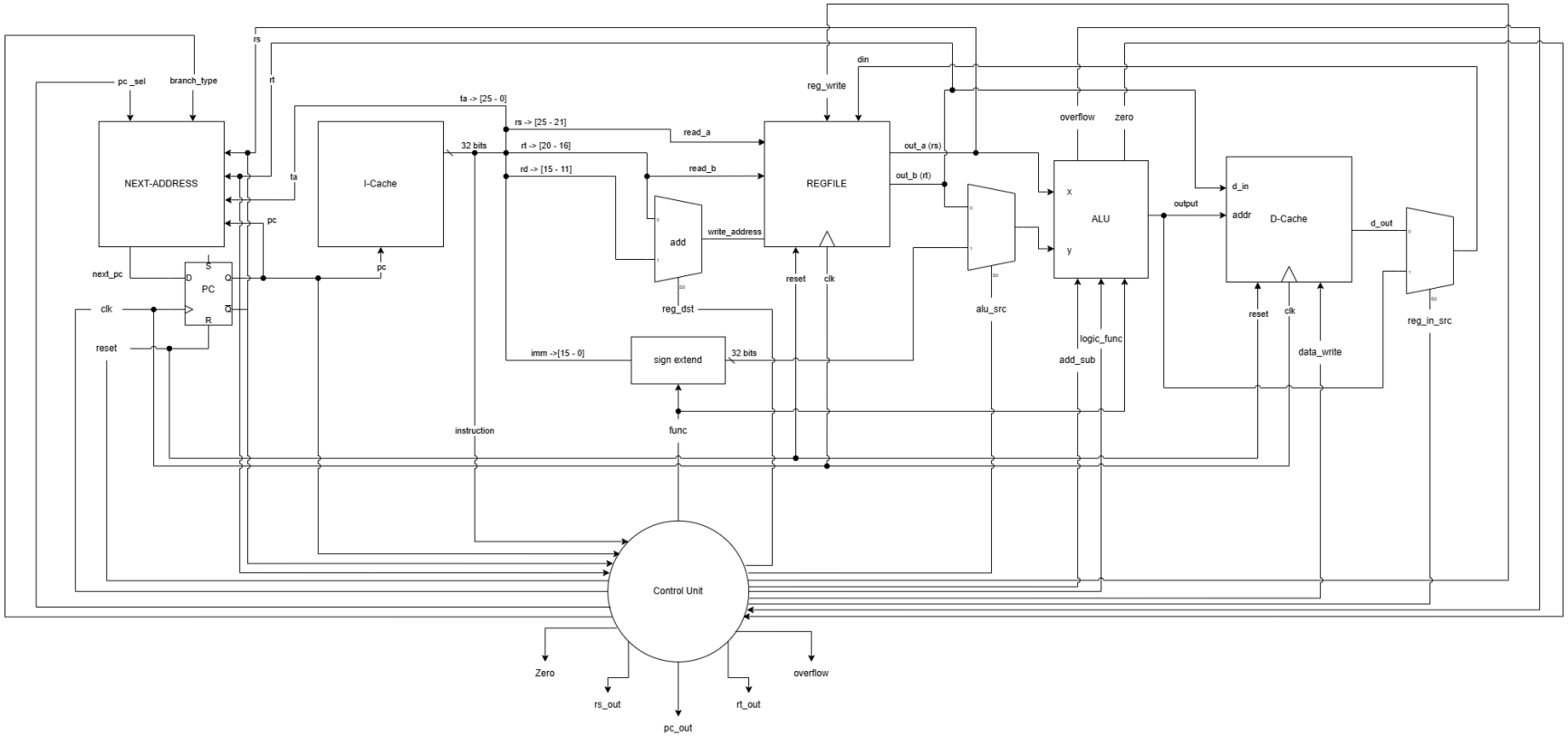
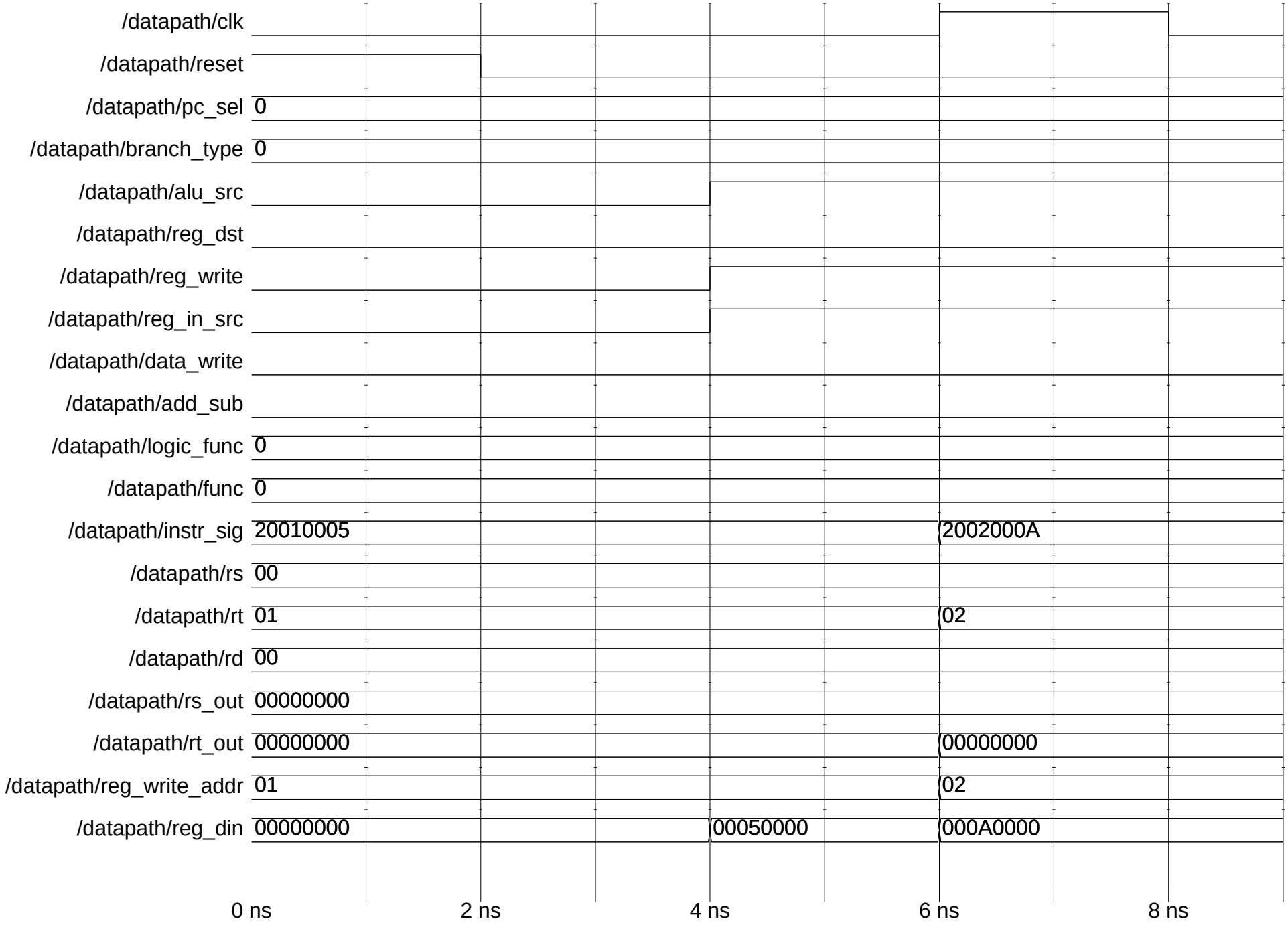
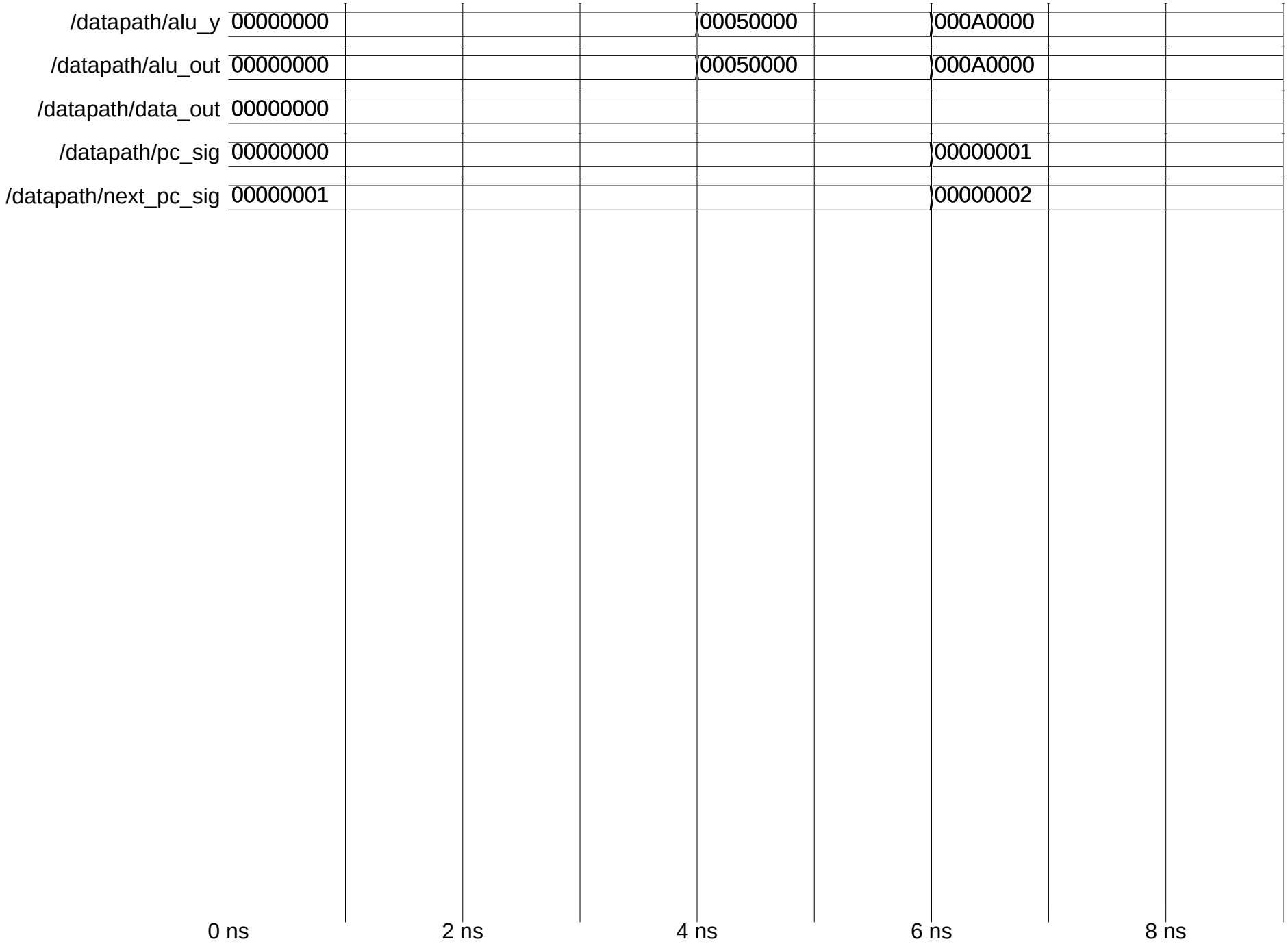


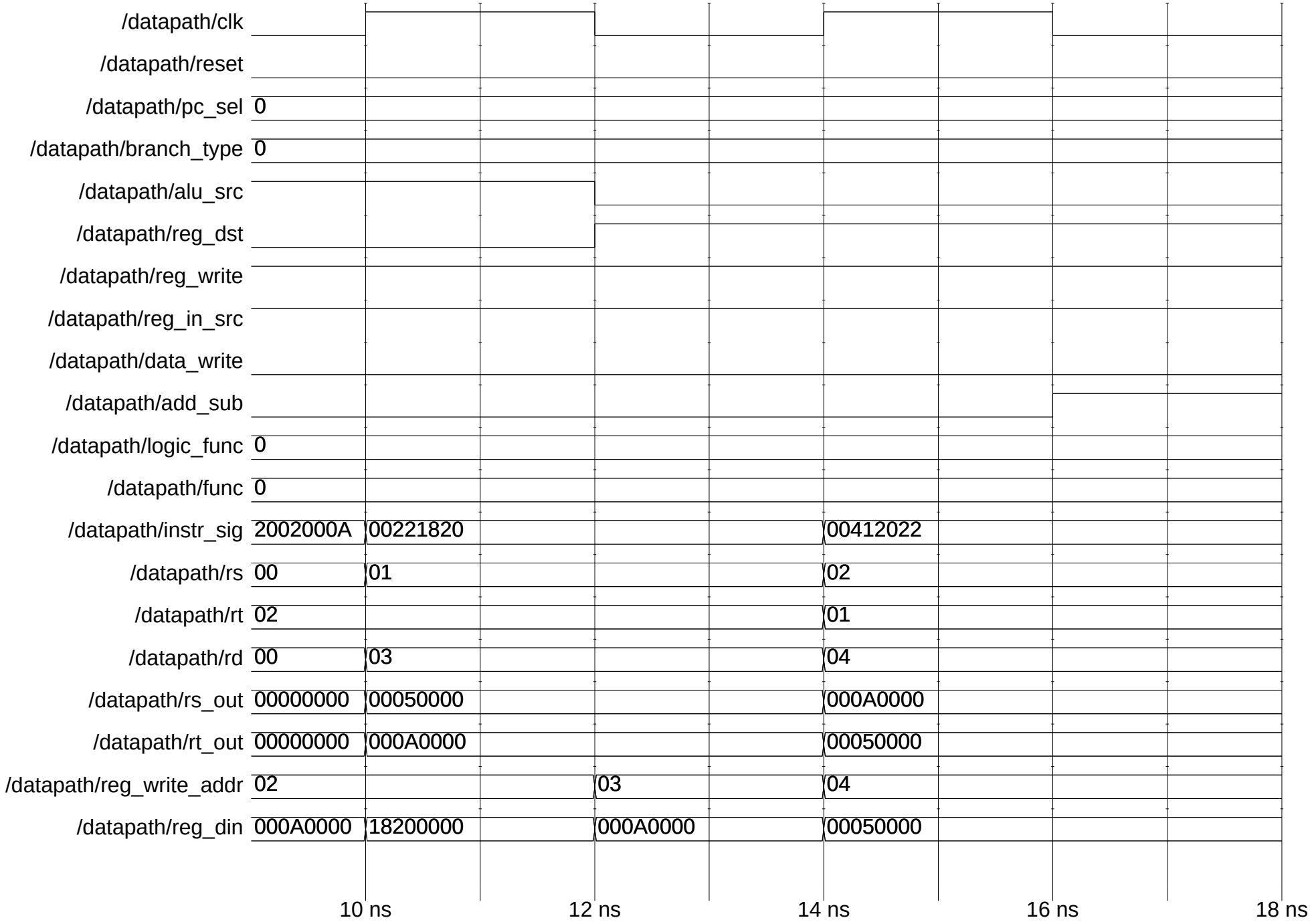
Figure 16: Complete CPU design



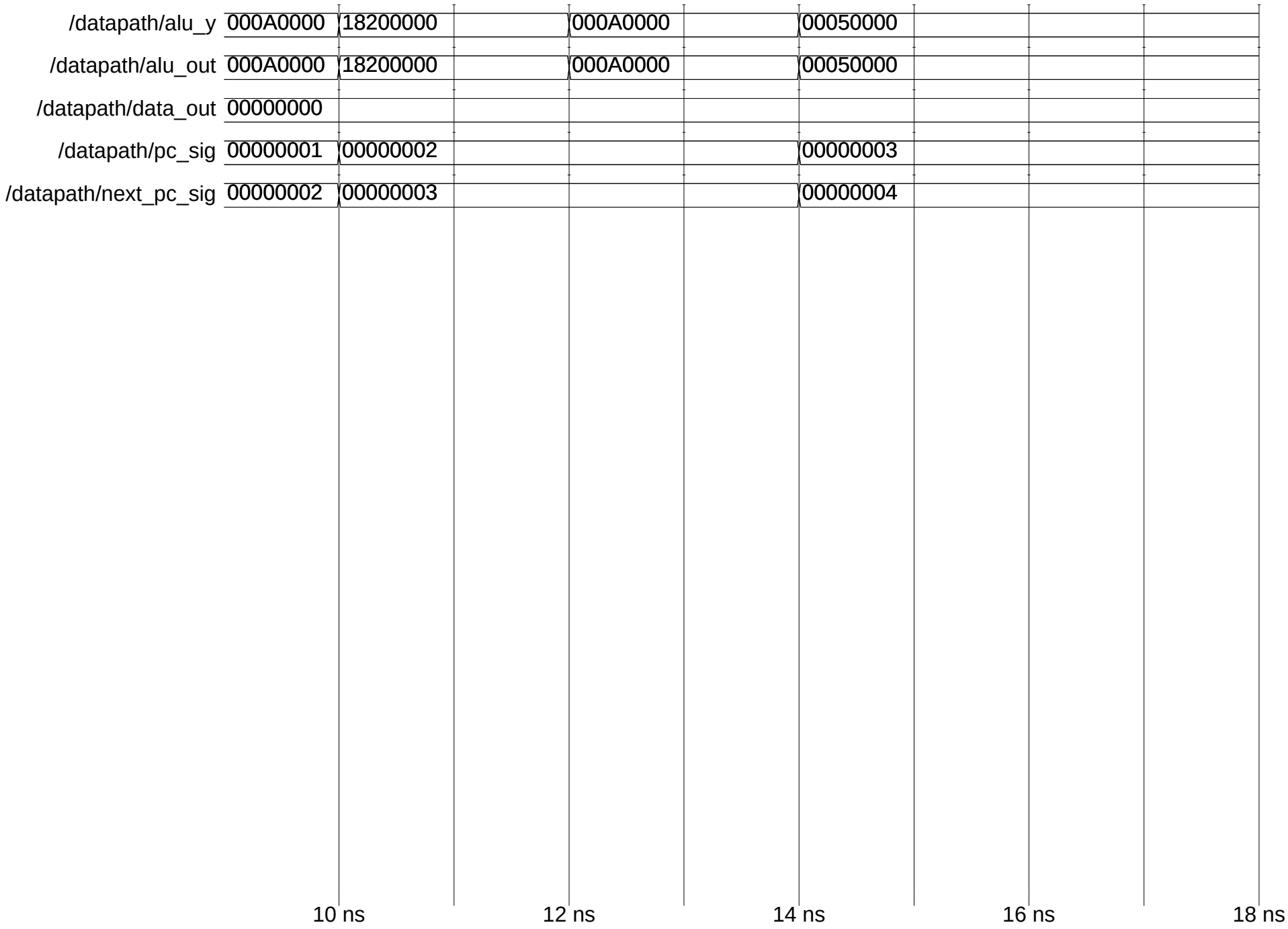
Entity:datapath Architecture:behavior Date: Tue Nov 25 11:04:16 AM EST 2025 Row: 1 Page: 1



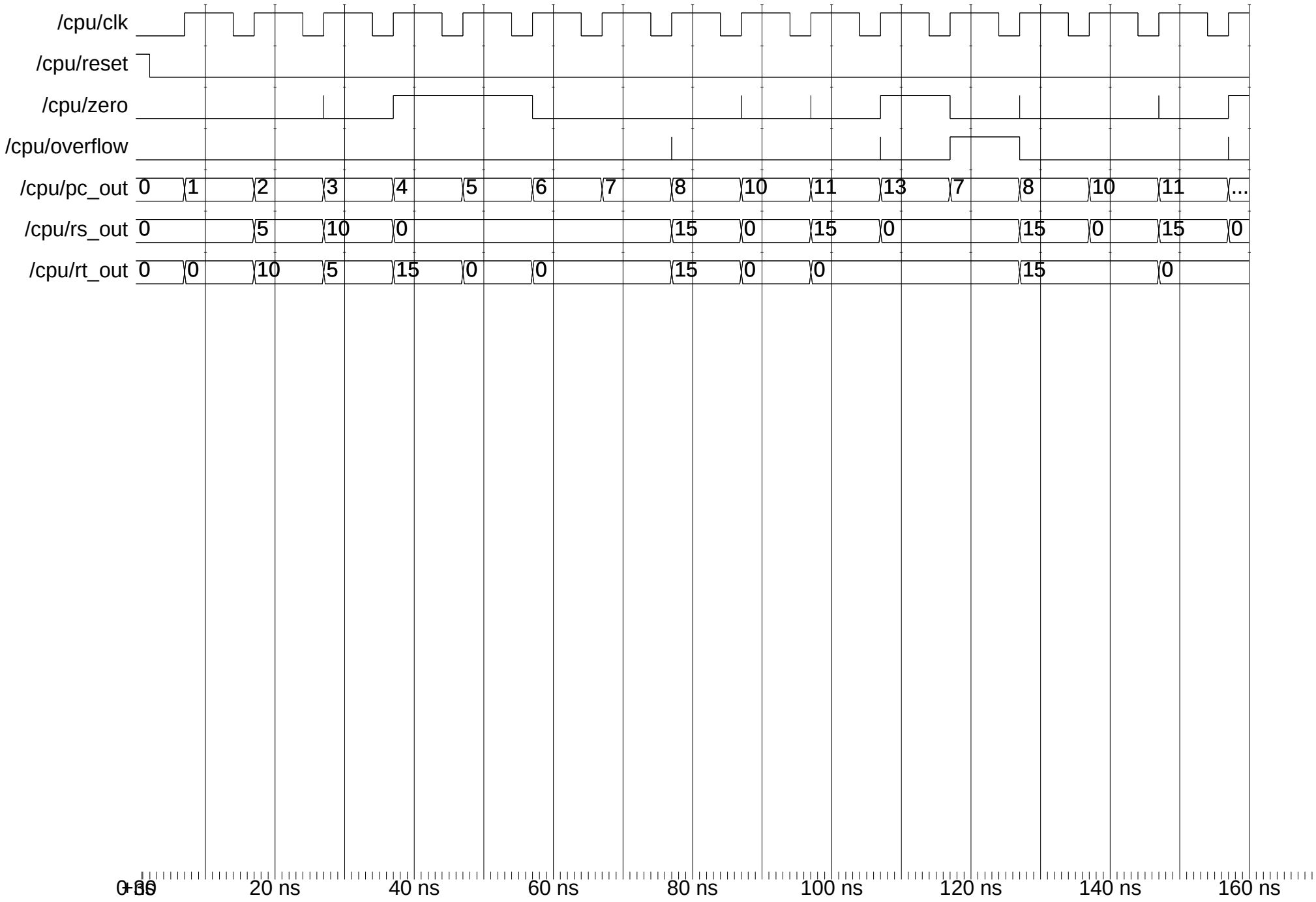
Entity:datapath Architecture:behavior Date: Tue Nov 25 11:04:16 AM EST 2025 Row: 1 Page: 2



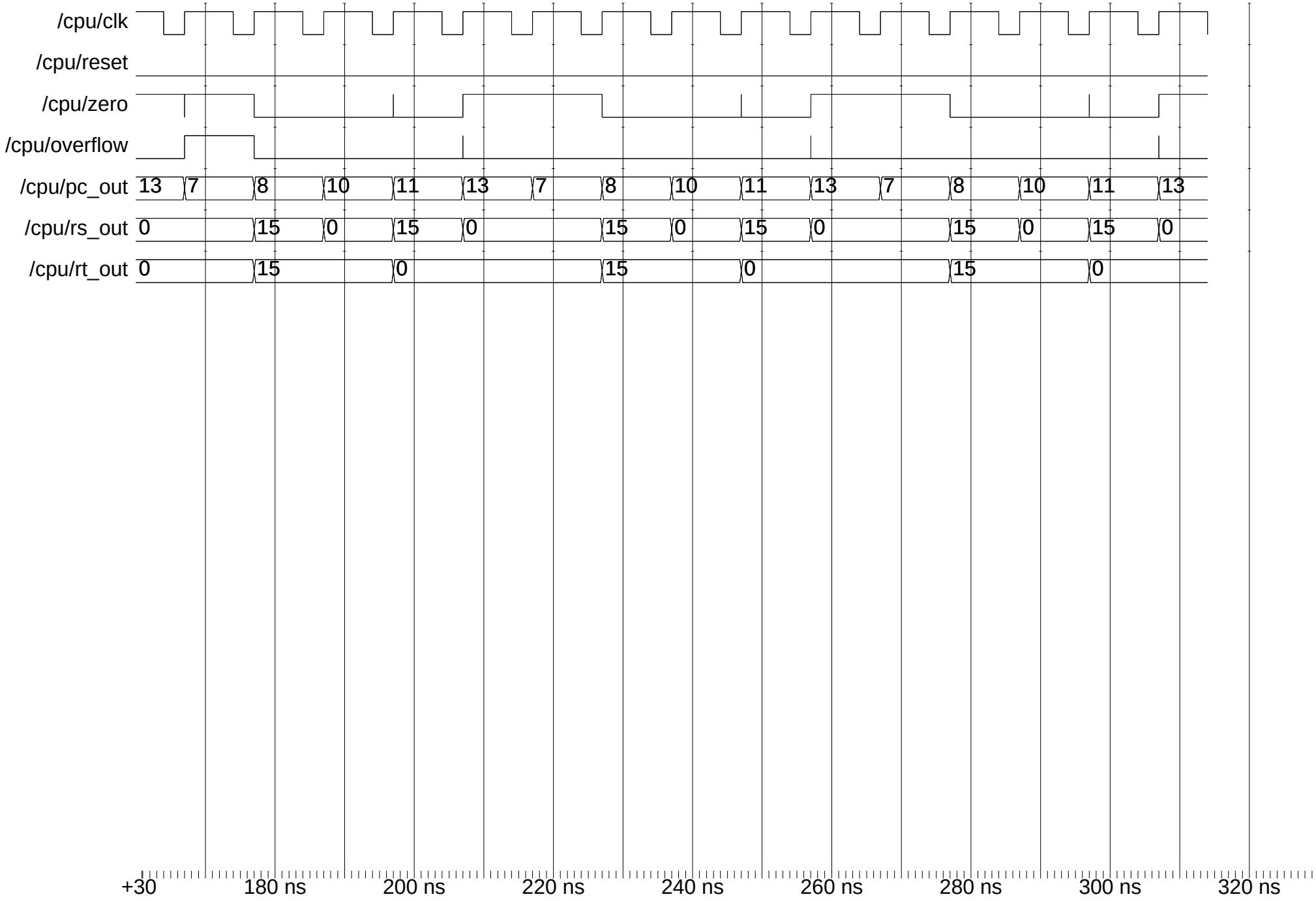
Entity:datapath Architecture:behavior Date: Tue Nov 25 11:04:16 AM EST 2025 Row: 2 Page: 3



Entity:datapath Architecture:behavior Date: Tue Nov 25 11:04:16 AM EST 2025 Row: 2 Page: 4



Entity:cpu Architecture:behavior Date: Tue Nov 25 11:06:48 AM EST 2025 Row: 1 Page: 1



Entity:cpu Architecture:behavior Date: Tue Nov 25 11:06:48 AM EST 2025 Row: 2 Page: 2

```
################################## One CPU ##################################

1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs              |  685 |     0 |          0 |     63400 |  1.08 |
|   LUT as Logic          |  685 |     0 |          0 |     63400 |  1.08 |
|   LUT as Memory         |    0 |     0 |          0 |     19000 |  0.00 |
| Slice Registers         | 1221 |     0 |          0 |    126800 |  0.96 |
|   Register as Flip Flop | 1221 |     0 |          0 |    126800 |  0.96 |
|   Register as Latch     |    0 |     0 |          0 |    126800 |  0.00 |
| F7 Muxes                |  173 |     0 |          0 |     31700 |  0.55 |
| F8 Muxes                |   62 |     0 |          0 |     15850 |  0.39 |
+-------------------------+------+-------+------------+-----------+-------+

2. Slice Logic Distribution
---------------------------

+--------------------------------------------+------+-------+------------+-----------+-------+
|                  Site Type                 | Used | Fixed | Prohibited | Available | Util% |
+--------------------------------------------+------+-------+------------+-----------+-------+
| Slice                                      |  421 |     0 |          0 |     15850 |  2.66 |
|   SLICEL                                   |  268 |     0 |            |           |       |
|   SLICEM                                   |  153 |     0 |            |           |       |
| LUT as Logic                               |  685 |     0 |          0 |     63400 |  1.08 |
|   using O5 output only                     |    0 |       |            |           |       |
|   using O6 output only                     |  673 |       |            |           |       |
|   using O5 and O6                          |   12 |       |            |           |       |
| LUT as Memory                              |    0 |     0 |          0 |     19000 |  0.00 |
|   LUT as Distributed RAM                   |    0 |     0 |            |           |       |
|     using O5 output only                   |    0 |       |            |           |       |
|     using O6 output only                   |    0 |       |            |           |       |
|     using O5 and O6                        |    0 |       |            |           |       |
|   LUT as Shift Register                    |    0 |     0 |            |           |       |
|     using O5 output only                   |    0 |       |            |           |       |
```

```
|     using O6 output only                 |    0 |       |            |           |       |
|     using O5 and O6                      |    0 |       |            |           |       |
| Slice Registers                          | 1221 |     0 |          0 |    126800 |  0.96 |
|   Register driven from within the Slice  |   10 |       |            |           |       |
|   Register driven from outside the Slice | 1211 |       |            |           |       |
|     LUT in front of the register is unused |  926 |       |            |           |       |
|     LUT in front of the register is used |  285 |       |            |           |       |
| Unique Control Sets                      |   39 |       |          0 |     15850 |  0.25 |
+------------------------------------------+------+-------+------------+-----------+-------+


################################## Five CPUs ###################################
1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs              | 3506 |     0 |          0 |     63400 |  5.53 |
|   LUT as Logic          | 3506 |     0 |          0 |     63400 |  5.53 |
|   LUT as Memory         |    0 |     0 |          0 |     19000 |  0.00 |
| Slice Registers         | 6105 |     0 |          0 |    126800 |  4.81 |
|   Register as Flip Flop | 6105 |     0 |          0 |    126800 |  4.81 |
|   Register as Latch     |    0 |     0 |          0 |    126800 |  0.00 |
| F7 Muxes                |  865 |     0 |          0 |     31700 |  2.73 |
| F8 Muxes                |  310 |     0 |          0 |     15850 |  1.96 |
+-------------------------+------+-------+------------+-----------+-------+

2. Slice Logic Distribution
---------------------------

+--------------------------------------------+------+-------+------------+-----------+-------+
|                  Site Type                 | Used | Fixed | Prohibited | Available | Util% |
+--------------------------------------------+------+-------+------------+-----------+-------+
| Slice                                      | 3982 |     0 |          0 |     15850 | 25.12 |
|   SLICEL                                   | 3035 |     0 |            |           |       |
|   SLICEM                                   |  947 |     0 |            |           |       |
```

LUT as Logic	3506	0	0	63400	5.53
using O5 output only	0				
using O6 output only	3367				
using O5 and O6	139				
LUT as Memory	0	0	0	19000	0.00
LUT as Distributed RAM	0	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	0				
LUT as Shift Register	0	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	0				
Slice Registers	6105	0	0	126800	4.81
Register driven from within the Slice	51				
Register driven from outside the Slice	6054				
LUT in front of the register is unused	4645				
LUT in front of the register is used	1409				
Unique Control Sets	191		0	15850	1.21

################################## Ten CPUs ##################################
1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	6987	0	0	63400	11.02
LUT as Logic	6987	0	0	63400	11.02
LUT as Memory	0	0	0	19000	0.00
Slice Registers	12210	0	0	126800	9.63
Register as Flip Flop	12210	0	0	126800	9.63
Register as Latch	0	0	0	126800	0.00
F7 Muxes	1730	0	0	31700	5.46
F8 Muxes	620	0	0	15850	3.91

+-------------------------+-------+-------+------------+-----------+-------+

2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	7269	0	0	15850	45.86
SLICEL	5306	0			
SLICEM	1963	0			
LUT as Logic	6987	0	0	63400	11.02
using O5 output only	0				
using O6 output only	6624				
using O5 and O6	363				
LUT as Memory	0	0	0	19000	0.00
LUT as Distributed RAM	0	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	0				
LUT as Shift Register	0	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	0				
Slice Registers	12210	0	0	126800	9.63
Register driven from within the Slice	103				
Register driven from outside the Slice	12107				
LUT in front of the register is unused	9169				
LUT in front of the register is used	2938				
Unique Control Sets	381		0	15850	2.40