Design, Implementation and Verification of the i281 Educational CPU

Team Elecpeeps123

Sagarika Sinha (24B1305), Vedika Doke (24B1231), Mansi Sharma (24B1302)

2025

This report details the design and verification of the i281 Educational CPU, a single-cycle processor implemented using VHDL. The project successfully met its primary objective by designing a datapath and control unit capable of executing a complex test program, the Bubble Sort algorithm, on an array stored in data memory. Instead of dumping the complete code, the report focuses on explaining the architectural choices, control signal derivation, flag handling, and memory interactions that make the design work. Selected VHDL snippets are included to highlight important ideas, while the full source code is submitted separately.

Contents

1 Introduction and Project Objectives

The i281 CPU project serves as a comprehensive exercise in digital system design, requiring the synthesis of a complete processor from fundamental components. The architecture is a synchronous, single-cycle design tailored to demonstrate foundational computer organization principles such as instruction fetch, decode, execute, memory access, and write-back.

The central goal was to verify the functional correctness of the CPU by running a non-trivial program—the Bubble Sort algorithm—entirely on the i281 hardware model. This validation confirms that the processor correctly handles:

(i) Sequential instruction fetch and non-sequential control flow (jumps and conditional branches).

(ii) Arithmetic operations (ADD/SUB/shift) and conditional flag generation.

(iii) Multi-port register file access in a single-cycle datapath.

(iv) Read/Write operations on separate code and data memories.

Beyond simple correctness, the project also aims to:

- Map a given instruction set specification to a clean VHDL implementation.
- Derive a hardwired control unit using boolean relationships between opcodes, flags and control signals.
- Build intuition about how architectural decisions (flags semantics, branch conditions, memory mapping) directly affect how easy it is to program and debug the CPU.

2 i281 High-Level Architecture

The i281 processor is structured around a unified datapath and a centralized control unit. Key specifications include an 8-bit data/register width and 6-bit code memory addresses, giving 64 instruction words.

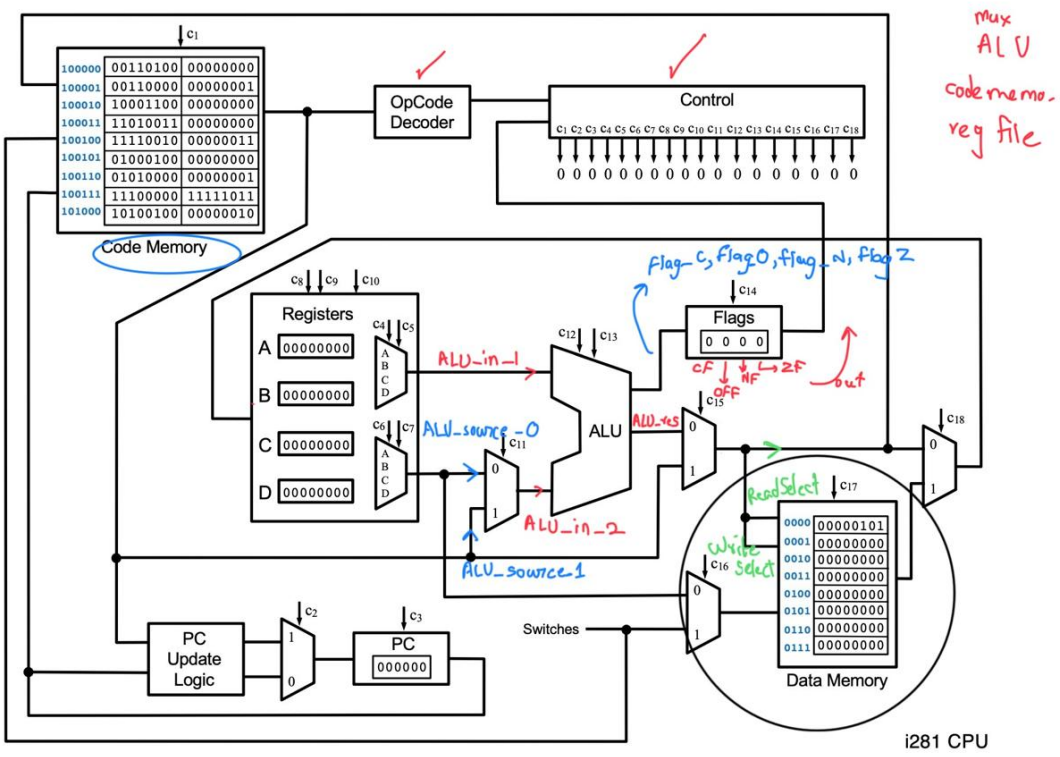

Figure 1: i281 CPU architecture showing Code Memory, Opcode Decoder, Control Unit, Register File, ALU, Flags Register, Data Memory, PC Update Logic and PC.

2.1 Architectural Components

- **Program Counter (PC):** A 6-bit register managed by `program_counter.vhd`. It holds the address of the current instruction and is updated every cycle.
- **PC Update Logic:** Implemented in `pc_update_logic.vhd`. It computes $PC + 1$ for sequential execution and $PC + 1 +$ offset for branches and jumps.
- **Code Memory (IMEM):** A 16-bit wide instruction memory, implemented in `code_memory.vhd`, storing the Bubble Sort program.
- **Opcode Decoder:** `opcode_decoder.vhd` converts the 8-bit opcode and register fields into one-hot instruction lines and register selectors.
- **Control Unit:** `control_logic.vhd` generates 18 control signals (C_1 to C_{18}) and register select lines from the decoded opcode and status flags.
- **Register File (RF):** `register_file.vhd` implements four 8-bit registers (A, B, C, D) with two asynchronous read ports and one synchronous write port.
- **ALU:** `alu.vhd` performs 8-bit arithmetic and shifts and produces the status flags ZF, NF, CF, OF.
- **Flags Register:** `flags_register.vhd` stores the four status flags with an explicit write enable (C_{14}), allowing compare-and-branch style instructions.

- **Data Memory (DMEM):** `data_memory.vhd` stores an array of 8-bit values for Bubble Sort along with general data.
- **Top-level CPU & Testbench:** `cpu.vhd` connects all modules structurally, and `cpu_tb.vhd` provides clock, reset and simulation control.

2.2 Program Counter and Branch Logic

The PC update path is shown conceptually in Fig. 2. A first adder computes $PC+1$ for the next sequential instruction. The low byte of the instruction provides an 8-bit signed offset used for branches and jumps. This offset is added to $PC+1$ in a second adder. A MUX, controlled by control signal C_2, selects whether the next PC is $PC+1$ (no branch taken) or $PC+1+$ offset (branch/jump taken).

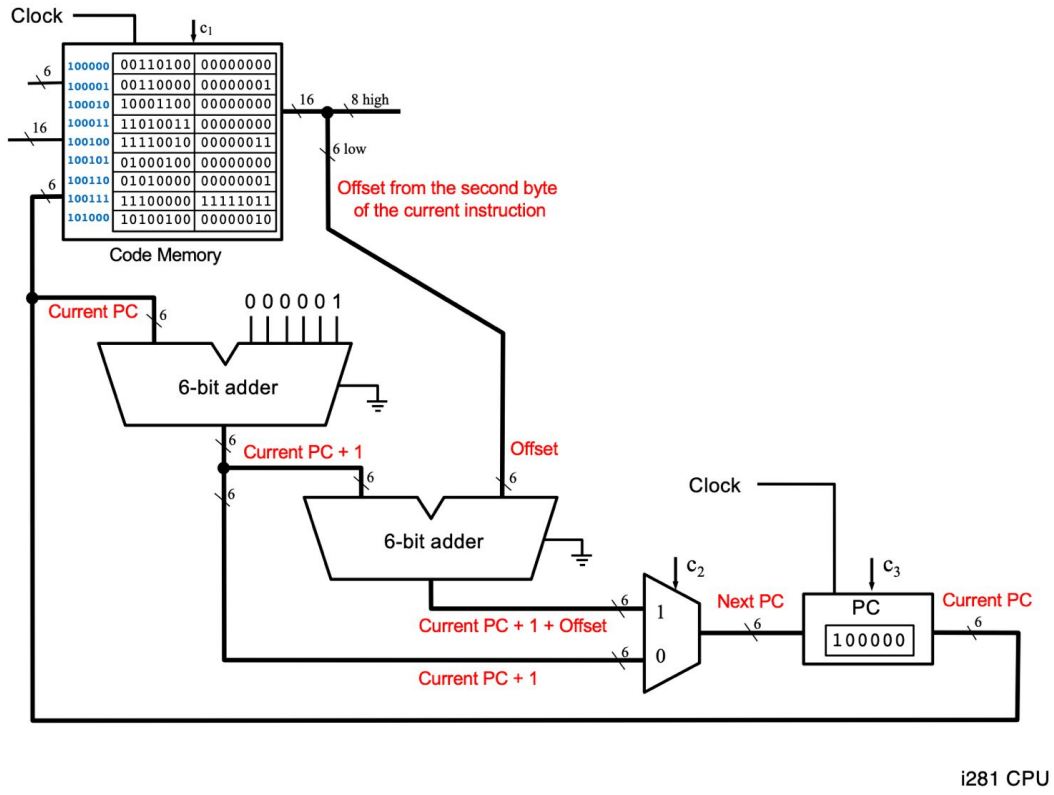

Figure 2: PC update datapath: computation of $PC+1$ and $PC+1+$ Offset and the selection of the next PC using control signal C_2.

This organisation keeps the PC path purely combinational with a single synchronous register at the end, making the reasoning about branches clear: whenever the control logic asserts C_2, execution is redirected by adding the offset to the sequential PC.

3 Instruction Set Architecture (ISA) and Encoding

The instruction set uses a 16-bit fixed format, primarily divided into a high-byte OpCode field and a low-byte operand field.

3.1 Instruction Format Breakdown

Table 1: i281 Instruction Field Allocation

Bits	Width	Meaning
15–8	8	OpCode, Register Source/Destination Selectors (X1,X0,Y1,Y0)
7–0	8	Immediate Value, Data Address, or PC Offset

The X and Y fields select registers A–D, while the low byte is interpreted depending on the instruction type:

- As an 8-bit immediate for `LOADI`/`ADDI`/`SUBI`.
- As a data memory address for `LOAD`/`STORE`.
- As a signed PC offset for branches and jumps.

3.2 Opcode Groups and Functionality

The processor supports 23 distinct instructions categorized by their function:

- **Data Transfer:** `LOADI`, `LOADP`, `LOAD`, `LOADF`, `STORE`, `STOREF`, `MOVE`, and input instructions. The `MOVE` instruction is efficiently implemented as an `ADD` where the second operand is zero, avoiding a separate datapath.
- **ALU Operations:** `ADD`, `ADDI`, `SUB`, `SUBI`, `SHIFTL`, `SHIFTR`, `CMP`. These typically require setting C_{10} (Register Write Enable) and C_{14} (Flags Write Enable) so that both the destination register and the status flags are updated.
- **Control Flow:** `JUMP`, `CMP`, and conditional branches (`BRZ`, `BRE`, `BRNE/BRNZ`, `BRG`, `BRGE`). These are critical for loop structures and conditional execution and primarily influence the PC-path select signal C_2.

4 Component Design and VHDL Implementation Details

This section describes the main hardware components and explains the rationale behind the VHDL implementation. Only representative code fragments are shown; the complete code is provided separately and summarised in Appendix A.

4.1 Arithmetic Logic Unit (ALU) and Flags

The 8-bit ALU (`alu.vhd`) is a purely combinatorial component. It accepts two 8-bit operands, A and B, and performs shifts, addition or subtraction depending on control bits C_{12} and C_{13}.

4.1.1 ALU Operation Control

Table 2: ALU Operation Selection based on Control Signals

C_{12}	C_{13}	Operation
0	0	Shift Left (SHIFTL)
0	1	Shift Right (SHIFTR)
1	0	Addition (ADD/ADDI/MOVE)
1	1	Subtraction / Compare (SUB/SUBI/CMP)

4.1.2 Flag Calculation Logic

The ALU combines unsigned and signed arithmetic to compute the four status flags correctly. A simplified version of the implementation is:

Listing 1: ALU Core and Flag Computation (Simplified)

```
process(A, B, c12, c13)
    variable Au, Bu : unsigned(7 downto 0);
    variable sum9   : unsigned(8 downto 0);
    variable tmp_s  : signed(8 downto 0);
    variable sel    : std_logic_vector(1 downto 0);
    variable res_v  : std_logic_vector(7 downto 0);
    variable c_out  : std_logic := '0';
    variable v_out  : std_logic := '0';
begin
    sel := c12 & c13;
    Au  := unsigned(A);
    Bu  := unsigned(B);

    case sel is
        when "00" =>            -- SHL
            res_v := A(6 downto 0) & '0';
            c_out := A(7);      -- bit shifted out

        when "01" =>            -- SHR
            res_v := '0' & A(7 downto 1);
            c_out := A(0);

        when "10" =>            -- ADD
            sum9  := ("0" & Au) + ("0" & Bu);
            res_v := std_logic_vector(sum9(7 downto 0));
            c_out := sum9(8);   -- carry bit

            tmp_s := resize(signed(A),9) + resize(signed(B),9);
            if (tmp_s > 127) or (tmp_s < -128) then
                v_out := '1';   -- signed overflow
            end if;

        when others =>          -- SUB / CMP
            res_v := std_logic_vector(Au - Bu);
```

```
            if Au < Bu then
                c_out := '0';   -- borrow
            else
                c_out := '1';   -- no borrow
            end if;
            v_out := (A(7) xor B(7)) and (A(7) xor res_v(7));
    end case;

    alu_res <= res_v;
    FLAG_z  <= '1' when res_v = "00000000" else '0';
    FLAG_n  <= res_v(7);
    FLAG_c  <= c_out;
    FLAG_o  <= v_out;
end process;
```

Unsigned arithmetic is used for the carry/borrow flag, while a signed 9-bit extended representation is used to detect signed overflow. This combination makes later branch decisions based on ZF, NF and OF align with standard signed integer semantics.

4.1.3 Flags Register

The flags register decouples the ALU from branching so that a compare and the subsequent branch can be separate instructions. Only instructions that conceptually update the status (ALU ops and `CMP`) assert C_{14}.

Listing 2: Flags Register: Latching Status Flags

```
entity flags_register is
    port (
        clock   : in std_logic;
        reset   : in std_logic;
        flag_c  : in std_logic;
        flag_o  : in std_logic;
        flag_n  : in std_logic;
        flag_z  : in std_logic;
        c14     : in std_logic;  -- Flags Write Enable
        cf      : out std_logic;
        off     : out std_logic;
        nf      : out std_logic;
        zf      : out std_logic
    );
end flags_register;

architecture Behavioral of flags_register is
begin
    process(clock, reset)
    begin
        if reset = '0' then
            zf <= '0'; cf <= '0'; nf <= '0'; off <= '0';
        elsif rising_edge(clock) then
            if c14 = '1' then
                zf <= flag_z; cf <= flag_c; nf <= flag_n; off <=
   flag_o;
```

```
            end if;
        end if;
    end process;
end Behavioral;
```

4.2 Control Unit Logic Synthesis

The control unit implements the instruction semantics by translating the one-hot opcode lines and register fields into the control signals C_1 to C_{18}. The starting point is the official control table, reproduced conceptually in Fig. 3.

	IMEM_WRITE_ENABLE	PROGRAM_COUNTER_MUX	PROGRAM_COUNTER_WRITE_ENABLE	REGISTERS_PORT0_SELECT1	REGISTERS_PORT0_SELECT0	REGISTERS_PORT1_SELECT1	REGISTERS_PORT1_SELECT0	REGISTERS_WRITE_SELECT1	REGISTERS_WRITE_SELECT0	REGISTERS_WRITE_ENABLE	ALU_SOURCE_MUX	ALU_SELECT1	ALU_SELECT0	FLAGS_WRITE_ENABLE	ALU_RESUT_MUX	DMEM_INPUT_MUX	DMEM_WRITE_ENABLE	REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

computed using the flags register

```
B1=  ZF
B2= ~ZF
B3= AND(~ZF, XNOR(NF, OF))
B4= XNOR(NF, OF)
```

Zero Flag (ZF)
Negative Flag (NF)
Overflow Flag (OF)

Figure 3: Control signal activity for each instruction. Branch conditions B_1–B_4 are expressed in terms of the flags (ZF, NF, OF).

4.2.1 Branch Condition Generation

The branch conditions are generated from the persistent flags stored in the `flags_register`. The logical derivation of the four branch signals (B_1 to B_4) and the PC select signal (C_2) is:

Listing 3: Control Logic: Branch Condition Generation

```
-- Flags mapping: ZF=Flags(0), NF=Flags(1), OFF=Flags(2)

B1 <= ZF;                                  -- B1 (BRE/BRZ)    : equal /
   zero
B2 <= not ZF;                              -- B2 (BRNE/BRNZ) : not equal
B3 <= (not ZF) and (not (NF xor OFF));     -- B3 (BRG)        : greater
   than
B4 <= not (NF xor OFF);                    -- B4 (BRGE)       : greater
   or equal

-- PC MUX select: branch or jump taken
c2 <= JUMP or
      (B1 AND BRE_BRZ)    or
      (B2 AND BRNE_BRNZ) or
      (B3 AND BRG)        or
      (B4 AND BRGE);
```

These equations implement signed comparisons using the standard relationships between ZF, NF and OF. C_2 is asserted exactly when a branch or jump is taken, causing the PC update logic to add the offset.

4.2.2 Other Key Control Signals

A few representative examples:

- C_{10}: Register Write Enable, active for all instructions that write to the RF (e.g., `LOADI`, `ADD`, `LOAD`, `MOVE`).
- C_{11}: ALU source MUX; selects between register and immediate as operand B.
- $C_{12,13}$: ALU operation selection lines as in Table 1.
- C_{14}: Flags Write Enable, active for ALU instructions and `CMP`.
- C_{17}: Data Memory Write Enable for `STORE`/`STOREF`.
- C_{18}: Register write-back source; selects DMEM output for `LOAD`/`LOADF`.

4.3 Register File Implementation

The Register File (`register_file.vhd`) is implemented as an array of four 8-bit registers, indexed by X and Y fields. Two combinational read ports provide operands to the ALU, while a single synchronous write port updates the destination register under the control of C_{10}.

Listing 4: Register File: Synchronous Write Logic

```
process(clock, reset)
begin
    if reset = '0' then
        Registers <= (others => (others => '0'));
    elsif rising_edge(clock) then
        if c10 = '1' then   -- Register write enable
```

```
            case Write_Select_Vector is
                when "00" => Registers(0) <= reg_file_input; -- A
                when "01" => Registers(1) <= reg_file_input; -- B
                when "10" => Registers(2) <= reg_file_input; -- C
                when "11" => Registers(3) <= reg_file_input; -- D
                when others => null;
            end case;
        end if;
    end if;
end process;
```

The read ports are implemented as simple multiplexers using X1,X0 and Y1,Y0 from the decoder, enabling single-cycle operand fetch for all arithmetic and memory instructions.

4.4 Memories, Top-Level and Testbench

Code and Data Memories. `code_memory.vhd` is a ROM initialised with the Bubble Sort program and a few test instructions. `data_memory.vhd` is an 8-bit RAM initialised with an unsorted array:

$$[7, 3, 2, 1, 6, 4, 5, 8].$$

The final state after execution is

$$[1, 2, 3, 4, 5, 6, 7, 8],$$

as confirmed in Section 5.1.

Top-level CPU. `cpu.vhd` is purely structural, instantiating the PC, PC update logic, memories, opcode decoder, control logic, register file, ALU and flags register, and wiring them according to Fig. 1.

Testbench. `cpu_tb.vhd` generates the clock and reset signals and runs the simulation for enough time for the sort to complete. A representative fragment is:

Listing 5: Testbench: Clock and Reset Generation

```
architecture Behavioral of cpu_tb is
    signal clock : std_logic := '0';
    signal reset : std_logic := '0';
    constant clock_period : time := 20 ns;
begin
    -- Instantiate CPU
    uut: entity work.cpu
        port map (
            clock => clock,
            reset => reset
        );

    -- Clock generation (50 MHz)
    clock_process : process
    begin
        clock <= '0';
        wait for clock_period/2;
```

```
        clock <= '1';
        wait for clock_period/2;
    end process;

    -- Reset pulse
    reset_process : process
    begin
        reset <= '0';
        wait for 2*clock_period;
        reset <= '1';
        wait;
    end process;
end Behavioral;
```

5 Verification Strategy: Bubble Sort Execution

The primary mechanism for verifying the i281 processor's functionality is the successful execution of the Bubble Sort program, loaded into the Code Memory. This program uses nested loops, index manipulation, conditional comparisons and data swaps, exercising almost all instructions and datapath components.

5.1 Data Memory Setup and Expected Output

The Data Memory (`data_memory.vhd`) is initialised with an unsorted array of 8-bit values:

- **Initial Unsorted Data:** x"07", x"03", x"02", x"01", x"06", x"04", x"05", x"08"
- **Expected Sorted Output:** x"01", x"02", x"03", x"04", x"05", x"06", x"07", x"08"

After running the simulation, the memory viewer in the CAD tool confirms that locations 0–7 contain the sorted sequence. A snapshot of this final state is shown in Fig. 4.

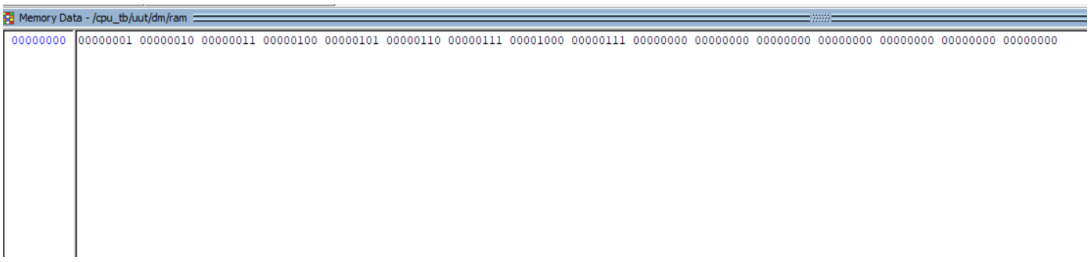

Figure 4: Data memory after Bubble Sort completion. Addresses 0–7 contain the sorted sequence 1,2,3,4,5,6,7,8.

5.2 Simulation Methodology

The behavioural testbench `cpu_tb.vhd` runs the CPU at a simulated clock frequency of 50 MHz (20 ns period) for several microseconds. This number of cycles is sufficient for the $\mathcal{O}(N^2)$ Bubble Sort algorithm to complete on an 8-element array. During simulation, the following signals are monitored:

- Global signals: clock, reset.
- Execution signals: Program Counter (PC), current instruction word.
- Flags: Zero (Z), Negative (N), Carry (C), Overflow (O).
- Register File contents: registers A–D.
- Data Memory: contents at addresses 0–7.

A condensed waveform view is shown in Fig. 5.

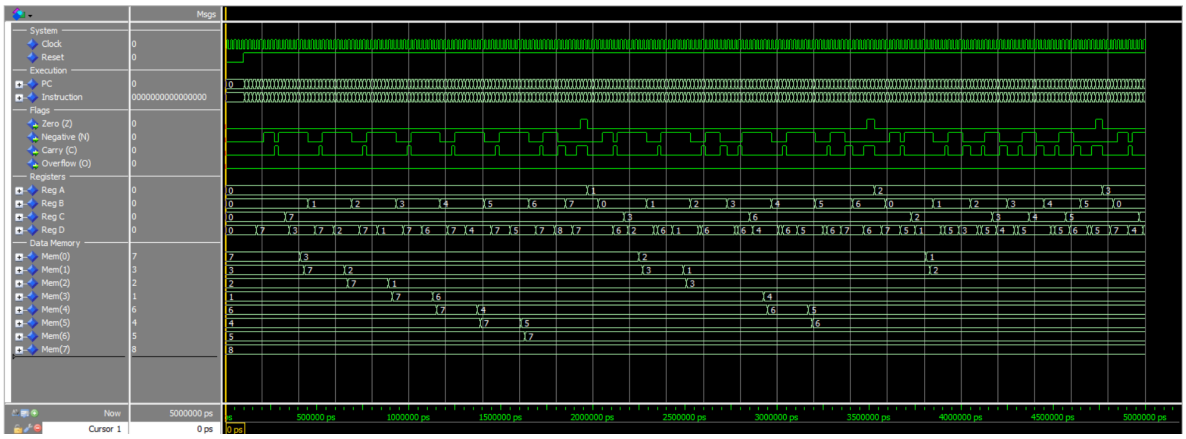

Figure 5: Simulation waveform of Bubble Sort execution. PC, Instruction, Flags, Registers and Data Memory locations 0–7 are observed over time.

5.2.1 Critical Instruction Trace

During the inner loop of Bubble Sort, the following micro-operations occur repeatedly:

(a) **Memory Access:** Two adjacent data elements are fetched using `LOADF` and placed into registers A and B. The effective address is formed by adding the base of the array and an index register.

(b) **Comparison:** The `CMP` instruction executes $A - B$ using the ALU subtraction mode. Control signal C_{14} is asserted so that the flags register stores the new ZF, NF and OF values derived from the result.

(c) **Branching:** The branch instruction `BRG` is fetched next. The control logic computes $B_3 = (\neg ZF) \wedge \neg(NF \oplus OF)$. If true, C_2 is asserted and PC jumps to the SWAP label, otherwise sequential execution continues.

(d) **Data Writeback:** In the SWAP block, the contents of A and B are written back to swapped locations using `STORE` or `STOREF`. This tests correct operation of C_{17} (DMEM write enable) and the register-to-memory datapath.

By placing cursors at key points in the waveform, it is possible to see each of these steps: the PC value, the fetched instruction, the change in flags after `CMP`, the subsequent change in C_2, and finally the swapping of data in memory.

6 Conclusion and Future Enhancement

The i281 single-cycle CPU was successfully implemented in VHDL, demonstrating a functional digital system that adheres to all project requirements. The execution of the Bubble Sort program confirms the operational integrity of the control unit, the ALU, and the register/memory interfaces. The simulation waveforms show that the program counter, flags, registers and data memory all evolve exactly as expected for the algorithm.

6.1 Performance Considerations

As a single-cycle design, the processor's maximum clock frequency ($f_{\max}$) is strictly limited by the total time delay of the longest combinational path, often referred to as the critical path:

$$t_{\text{critical}} \approx t_{\text{CodeMem\_read}} + t_{\text{Control}} + t_{\text{RegFile\_read}} + t_{\text{ALU}} + t_{\text{DataMem\_write}}.$$

All of these blocks must settle within one clock period. The design therefore favours clarity and simplicity rather than raw performance.

6.2 Recommendations for Future Work

1. **Pipelined Architecture:** Transition the design to a five-stage pipeline (Fetch, Decode, Execute, Memory, Write-back). This would reduce the critical path to the slowest single stage, allowing much higher clock frequencies. Pipeline hazards (data, control and structural) would have to be handled using forwarding and stall logic.

2. **Advanced ALU Design:** Replace the simple ripple-carry adder implied by `numeric_std` with a carry-lookahead or carry-select adder to reduce propagation delay for addition and subtraction.

3. **Interrupt Handling and I/O:** Adding interrupt support and simple memory-mapped I/O peripherals would move the CPU closer to a practical microcontroller.

4. **Expanded ISA and Wider Datapath:** Adding more addressing modes, additional registers, or moving to a 16-bit datapath would make the architecture more powerful and better suited to larger algorithms.

A Full VHDL Source Code Listings

The full VHDL source files are provided separately. This appendix presents representative excerpts from each file to illustrate structure and coding style.

A.1 flags_register.vhd

Listing 6: Source Excerpt: flags register.vhd

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

-- i281 Flags Register (ZF, NF, OF, CF)
entity flags_register is
    port (
        clock   : in std_logic; -- System Clock
        reset   : in std_logic; -- System Reset

        flag_c  : in std_logic;
        flag_o  : in std_logic;
        flag_n  : in std_logic;
        flag_z  : in std_logic;
        c14     : in std_logic; -- Flags Write Enable (C14)

        cf      : out std_logic;
        off     : out std_logic;
        nf      : out std_logic;
        zf      : out std_logic
    );
end flags_register;

architecture Behavioral of flags_register is
begin
    process(clock, reset)
    begin
        if reset = '0' then
            zf <= '0'; cf <= '0'; nf <= '0'; off <= '0';
        elsif rising_edge(clock) then
            if c14 = '1' then
                zf <= flag_z; cf <= flag_c; nf <= flag_n; off <=
   flag_o;
            end if;
        end if;
    end process;
end Behavioral;
```

A.2 alu.vhd

Listing 7: Source Excerpt: alu.vhd

```
library ieee;
```

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

entity alu is
  port(
    A       : in  std_logic_vector(7 downto 0);
    B       : in  std_logic_vector(7 downto 0);
    c12     : in  std_logic; -- ALU Select MSB
    c13     : in  std_logic; -- ALU Select LSB

    FLAG_c  : out std_logic;
    FLAG_o  : out std_logic;
    FLAG_n  : out std_logic;
    FLAG_z  : out std_logic;

    alu_res : out std_logic_vector(7 downto 0)
  );
end entity;

architecture behavioral of alu is
begin
  -- (Core process shown earlier in Listing \ref{lst:alu})
end architecture behavioral;
```

A.3 control_logic.vhd

Listing 8: Source Excerpt: control logic.vhd

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

entity control_logic is
    port (
        opcode_outputs : in std_logic_vector(26 downto 0);
        reset          : in std_logic;
        Flags          : in std_logic_vector(3 downto 0); -- ZF, NF,
    OF, CF

        c1  : out std_logic;
        c2  : out std_logic;
        c3  : out std_logic;
        -- ...
        c14 : out std_logic;
        c15 : out std_logic;
        c16 : out std_logic;
        c17 : out std_logic;
        c18 : out std_logic
    );
end entity control_logic;

architecture Behavioral of control_logic is
```

```
    -- Internal instruction and flag aliases
    signal NOOP, INPUTC, INPUTCF, INPUTD, INPUTDF, MOVE, LOADI_LOADP
    : std_logic;
    signal ADD, ADDI, SUB, SUBI, LOAD, LOADF, STORE, STOREF, SHIFTL,
    SHIFTR, CMP : std_logic;
    signal JUMP, BRE_BRZ, BRNE_BRNZ, BRG, BRGE : std_logic;

    signal X1, X0, Y1, Y0 : std_logic;
    signal ZF, NF, OFF    : std_logic;
    signal B1, B2, B3, B4 : std_logic;
begin
    -- Mapping from opcode_outputs to instruction aliases omitted
   for brevity...

    ZF  <= Flags(0); NF <= Flags(1); OFF <= Flags(2);

    -- Branch conditions and PC control: see Listing \ref{lst:branch
   }
    -- Other control signal equations are derived directly from Fig.
    \ref{fig:mapping}.
end architecture Behavioral;
```

A.4 Remaining Core VHDL Sources

A.4.1 Remaining Datapath Components

Listing 9: Source Skeleton: Register File.vhd

```
-- Synchronous Write Port (C8, C9, C10) and Asynchronous Read Ports
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
entity Register_File is
    port(
        clock         : in  std_logic;
        reset         : in  std_logic;
        -- ...
    );
end entity Register_File;

architecture Behavioral of Register_File is
    type reg_array is array(0 to 3) of std_logic_vector(7 downto 0);
    signal Registers : reg_array;
begin
    -- Read and write processes as shown in Listing \ref{lst:
   rf_write}
end architecture Behavioral;
```

Listing 10: Source Skeleton: program counter.vhd

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

entity program_counter is
    port(
```

```
        clock      : in  std_logic;
        reset      : in  std_logic;
        c3         : in  std_logic; -- PC write enable
        next_pc    : in  std_logic_vector(5 downto 0);
        current_pc : out std_logic_vector(5 downto 0)
    );
end program_counter;

architecture Behavioral of program_counter is
begin
    process(clock, reset)
    begin
        if reset = '0' then
            current_pc <= (others => '0');
        elsif rising_edge(clock) then
            if c3 = '1' then
                current_pc <= next_pc;
            end if;
        end if;
    end process;
end Behavioral;
```

Listing 11: Source Skeleton: pc update logic.vhd

```
-- Combinational Logic for PC + 1 and PC + 1 + Offset Calculation
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity pc_update_logic is
    port(
        pc     : in  std_logic_vector(5 downto 0);
        offset : in  std_logic_vector(5 downto 0);
        c2     : in  std_logic;
        next_pc: out std_logic_vector(5 downto 0)
    );
end pc_update_logic;

architecture Behavioral of pc_update_logic is
begin
    -- See Listing \ref{lst:pcblock} / \ref{lst:pc_logic} for logic.
end Behavioral;
```

A.4.2 Top-Level and Testbench

Listing 12: Source Skeleton: cpu.vhd (Structural Top-Level)

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

entity cpu is
    port(
        clock : in std_logic;
        reset : in std_logic
```

```
    );
end entity;

architecture struct of cpu is
    -- Component declarations for PC, ALU, RF, memories, etc.
begin
    -- Component instantiations and signal mapping
end struct;
```

Listing 13: Source Skeleton: cpu tb.vhd (Behavioral Testbench)

```
-- Generates clock/reset signals and runs the Bubble Sort program
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity cpu_tb is
end cpu_tb;

architecture Behavioral of cpu_tb is
    -- Clock, reset and CPU instantiation as in Listing \ref{lst:tb}
begin
end Behavioral;
```

B Appendix C: Control Signal Summary Table

The following table summarises the function of the key control signals for several representative instructions.

Table 3: Control Signal Activity for Key Instructions

Signal	Function	LOADI	MOVE	ADD	SUBI	LOAD	CMP	JUMP
C_1	IMEM Write Enable	0	0	0	0	0	0	0
C_2	PC MUX Select (Branch Taken)	0	0	0	0	0	0	1
C_3	PC Write Enable	1	1	1	1	1	1	1
C_{10}	Register Write Enable	1	1	1	1	1	0	0
C_{11}	ALU Source MUX (Immediate Select)	0	0	0	1	0	0	0
C_{12}	ALU Select MSB	0	1	1	1	0	1	0
C_{13}	ALU Select LSB	0	0	0	1	0	1	0
C_{14}	Flags Write Enable	0	0	1	1	0	1	0
C_{15}	ALU Result MUX (Immediate Pass)	1	0	0	0	0	0	0
C_{17}	DMEM Write Enable	0	0	0	0	0	0	0
C_{18}	Reg Writeback MUX (DMEM Data Select)	0	0	0	0	1	0	0

B.1 Control Signal Function Summary

- C_1: IMEM Write Enable. Used only for code/data input modes; inactive during normal program execution.
- C_2: PC MUX Select. 0 selects $PC + 1$; 1 selects $PC + 1 +$ Offset for taken branches and jumps.
- C_3: PC Write Enable. Held high in this single-cycle design so that PC updates every cycle.
- C_{10}: Register Write Enable. Enables data write to the selected Register File entry.
- C_{11}: ALU Source MUX. 0 selects register operand B; 1 selects the immediate field.
- C_{14}: Flags Write Enable. Allows the flags register to capture the ALU-computed flags.
- C_{15}: ALU Result MUX. 0 selects the ALU output; 1 bypasses the ALU and writes an immediate.
- C_{17}: DMEM Write Enable. Enables data to be written into Data Memory.
- C_{18}: Register Writeback MUX. 0 selects ALU/immediate result; 1 selects Data Memory output.