i281 CPU

Bubble Sort Algorithm Implementation

EE 224 - Digital Systems Course Project

Team Members:

Tushar Yadav	(24B1257)
Maitreya Rahul Shrimale	(24B1277)
Aarush Gupta	(24B1271)

Date: November 20, 2025

Indian Institute of Technology Bombay

Contents

1 Introduction

The objective of this project is to design and implement a simplified 8-bit CPU architecture (i281) capable of executing a bubble sort algorithm. The processor demonstrates fundamental digital system concepts including instruction fetch-decode-execute cycles, register file management, arithmetic logic unit (ALU) operations, and data memory manipulation. This report details the complete design, implementation, and verification of the system.

2 Project Specifications

2.1 Architecture Overview

The i281 processor is a 8-bit simplified CPU with the following specifications:

- **Instruction Width:** 16 bits
- **Data Width:** 8 bits
- **Address Width:** 6 bits (for Code Memory), 4 bits (for Data Memory)
- **Code Memory:** 64 words × 16 bits
- **Data Memory:** 16 words × 8 bits
- **Register File:** 4 registers (A, B, C, D) × 8 bits each
- **Control Signals:** 18 control lines (C1–C18)
- **Flags:** Zero, Negative, Carry, Overflow

2.2 Instruction Set

The processor supports 24 instruction types organized as follows:

Table 1: Instruction Set Summary

Category	**Instructions**
No Operation	NOOP
Data Transfer	INPUTC, INPUTCF, INPUTD, INPUTDF
Register Operations	MOVE, LOADI, LOADP
Arithmetic	ADD, ADDI, SUB, SUBI
Memory Access	LOAD, LOADF, STORE, STOREF
Bit Operations	SHIFTL, SHIFTR
Comparison	CMP
Control Flow	JUMP, BRE, BRZ, BRNE, BRNZ, BRG, BRGE

3 File Structure and Architecture

3.1 Design Hierarchy

The implementation consists of the following Verilog modules organized in a hierarchical structure:

Table 2: Module Hierarchy and Functionality

Module Name	**Functionality**
`i281_cpu.v`	Top-level processor module that instantiates and interconnects all sub-modules
`cmem.v`	Instruction memory (64 × 16 bits) with asynchronous read and synchronous write
`opcode_decoder.v`	Decodes 8-bit opcode field and extracts register select bits
`control.v`	Generates 18 control signals based on opcode and flag signals
`pc.v`	Program counter register with synchronous write and reset
`pcupdatelogic.v`	Computes next PC value (sequential or branching)
`registerfile.v`	4-register file with dual-port read and single-port write
`alu.v`	Arithmetic/Logic Unit supporting add, subtract, and shift operations
`flags_register.v`	Status register (Zero, Negative, Carry, Overflow flags)
`dmem.v`	Data memory (16 × 8 bits) with asynchronous read and synchronous write
`i281_cpu_tb.v`	Testbench for simulation and verification

3.2 Module Descriptions

3.2.1 Top-Level Module (i281_cpu.v)

The top-level module integrates all sub-modules and implements the critical interconnections. It performs the following functions:

- Manages the instruction fetch-decode-execute cycle
- Routes signals between instruction memory, decoder, control unit, register file, ALU, and data memory
- Implements multiplexers for data path routing (C11, C15, C16, C18)
- Provides clock and reset signals to all synchronous components

3.2.2 Code Memory (cmem.v)

Implements a 64-word instruction memory with the following characteristics:

- **Capacity:** 64 × 16 bits
- **Read Access:** Asynchronous (combinational)
- **Write Access:** Synchronous (edge-triggered)
- **Initial State:** Pre-loaded with bubble sort algorithm instructions

The memory is partitioned into BIOS code (addresses 0–31) and user code (addresses 32–63).

3.2.3 Opcode Decoder (opcode_decoder.v)

Decodes the 8-bit opcode field (bits [15:8] of instruction) and:

- Generates 23 one-hot encoded output signals (one per instruction type)
- Extracts register selection bits: X1, X0 (bits [11:10]) and Y1, Y0 (bits [9:8])
- Provides combinational logic for immediate instruction identification

3.2.4 Control Unit (control.v)

The control unit implements the instruction decode logic and generates 18 control signals:

- **C1:** Code memory write enable (for BIOS)
- **C2:** Program counter multiplexer select (sequential vs. branch)
- **C3:** Program counter write enable
- **C4–C7:** Register file read port selection
- **C8–C10:** Register file write port control
- **C11:** ALU input source multiplexer (register vs. immediate)
- **C12–C13:** ALU operation selection
- **C14:** Flags register write enable
- **C15:** ALU result multiplexer
- **C16–C17:** Data memory control
- **C18:** Register writeback multiplexer

3.2.5 Program Counter and Update Logic

PC Module (pc.v): A 6-bit synchronous counter that:

- Loads new address on clock edge when C3 (write enable) is asserted
- Performs asynchronous reset to address 32 (user code start)
- Outputs current address to code memory address bus

PC Update Logic (pcupdatelogic.v): Computes the next PC value:

- If C2 = 0: Sequential execution (PC + 1)
- If C2 = 1: Branch execution (PC + offset)
- Offset is extracted from instruction bits [5:0]

3.2.6 Register File (registerfile.v)

A 4-register array (A, B, C, D) with:

- **Port 0:** Read-only, controlled by C4–C5 (source for ALU input A)
- **Port 1:** Read-only, controlled by C6–C7 (source for ALU input B or memory data)
- **Write Port:** Single write port controlled by C8–C10
- **Data Width:** 8 bits per register
- **Reset Behavior:** Asynchronous reset sets all registers to 0x00

3.2.7 Arithmetic Logic Unit (alu.v)

Performs dual-function operations controlled by C12 and C13:

- **Arithmetic Mode (C12=1):** Addition or subtraction with carry/overflow detection
- **Shift Mode (C12=0):** Left shift (C13=0) or right shift (C13=1)
- **Flag Generation:** Produces Zero, Negative, Carry, Overflow flags

Arithmetic Operation:

$$\text{result} = A + \begin{cases} \sim B + 1, & \text{if } C13 = 1 \quad \text{(subtraction)} \\ B, & \text{if } C13 = 0 \quad \text{(addition)} \end{cases} \tag{1}$$

Shift Operation:

$$\text{result} = \begin{cases} A \gg 1, & \text{if } C13 = 1 \quad \text{(right shift)} \\ A \ll 1, & \text{if } C13 = 0 \quad \text{(left shift)} \end{cases} \tag{2}$$

3.2.8 Flags Register (flags_register.v)

Stores status flags updated by ALU operations:

- Zero (Z): Set when ALU result is zero
- Negative (N): Set when ALU result MSB is 1 (signed interpretation)
- Carry (C): Set on arithmetic carry or shifted-out bit
- Overflow (O): Set on signed arithmetic overflow

Flags are written synchronously when C14 (flags write enable) is asserted.

3.2.9 Data Memory (dmem.v)

The data memory is implemented as a 16-word × 8-bit register array and supports both read and write operations.

- **Capacity:** 16 × 8 bits
- **Read Access:** Asynchronous (combinational)
- **Write Access:** Synchronous (edge-triggered)

- **User-Configurable Initialization:** Loaded from an external HEX file using `$readmemh()`

Memory Initialization Using External File:

To allow full user customization of input values, the memory contents are loaded from an external file at runtime. This enables testing multiple datasets without modifying Verilog code.

Listing 1: Data Memory Initialization Using External File

```
initial begin
    if (!$readmemh("dmem.mem", memory)) begin
        $display("ERROR: Could not read dmem.mem file!");
    end else begin
        $display("Data Memory initialized from dmem.mem");
    end
end
```

Sample `dmem.mem` File (HEX Format):

```
07 03 02 01 06 04 05 08 07 00 00 00 00 00 00 00
```

3.3 RTL Schematic View

The synthesized RTL schematic of the i281 processor illustrates the hardware realization of the architecture. It highlights the interconnections between the Control Unit, Datapath (ALU, Register File), and Memory interfaces.

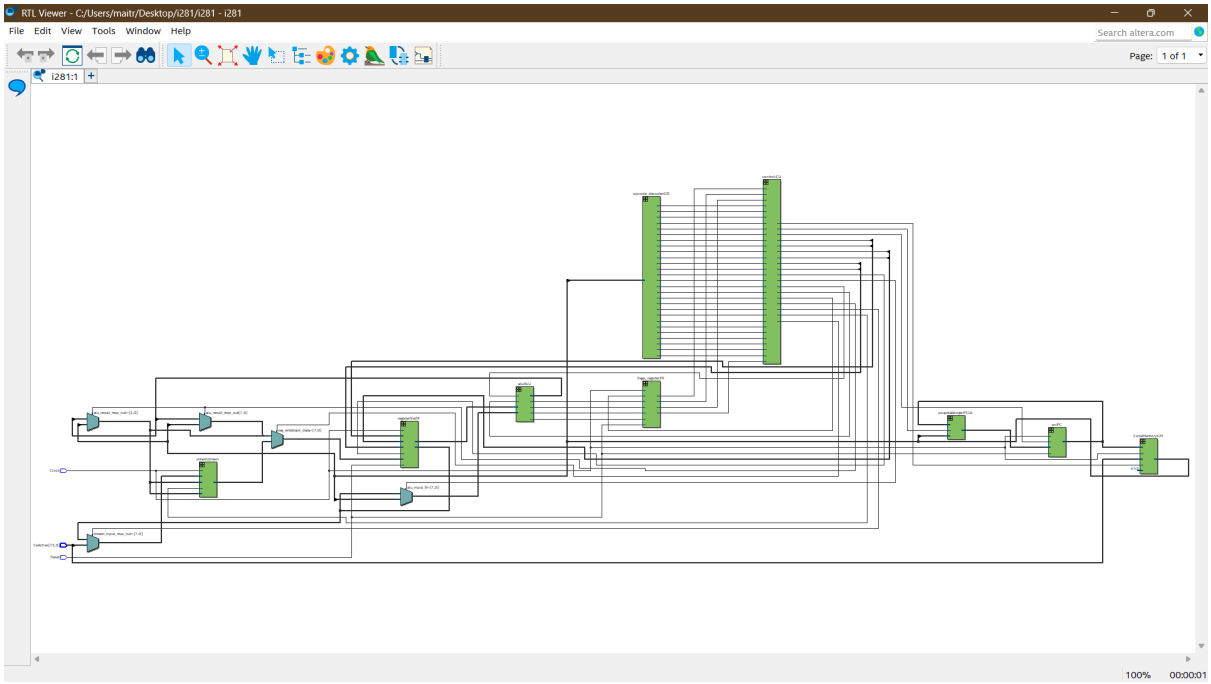
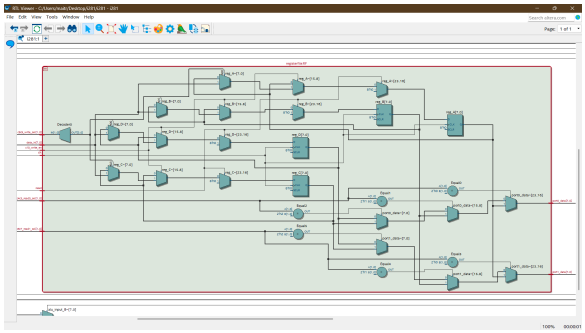
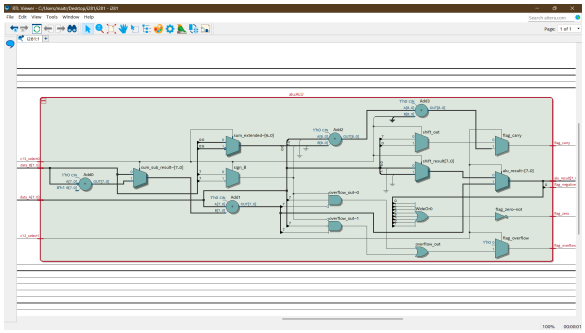


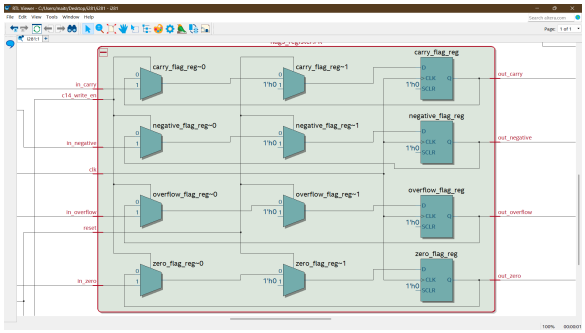
Figure 1: RTL Netlist Schematic of the i281 Processor.



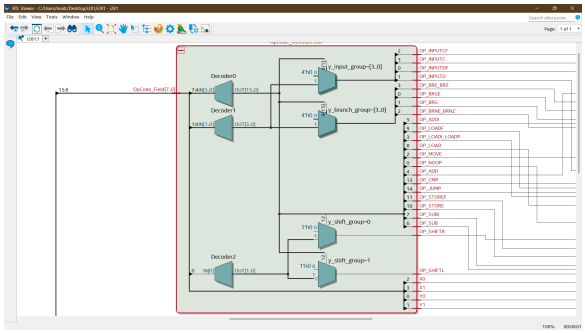
(a) Register File RTL Schematic



(b) ALU RTL Schematic



(c) Flags Register RTL View



(d) Opcode Decoder RTL View

Figure 2: RTL Views of Major CPU Submodules

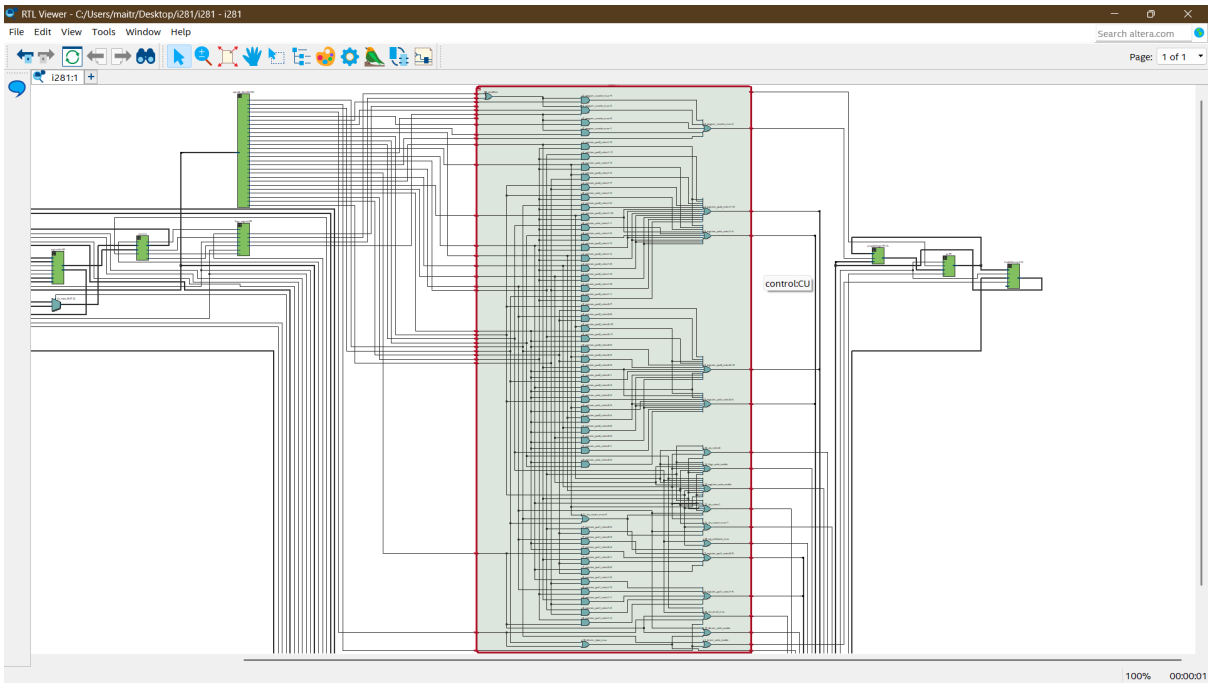
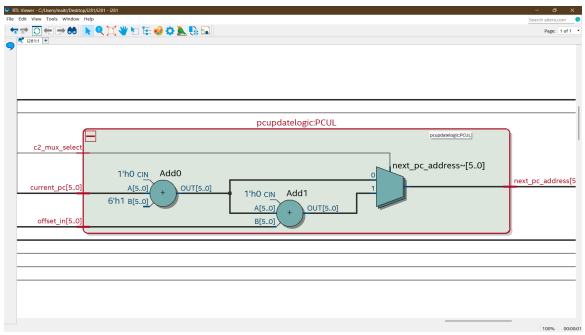
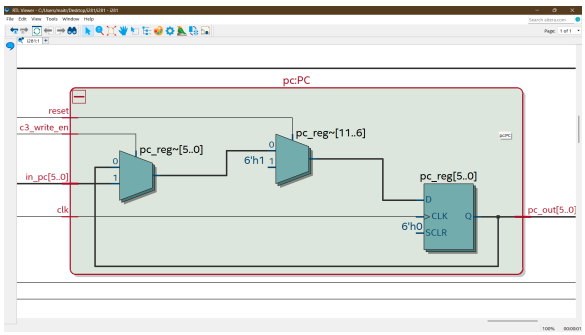


Figure 3: Control Unit RTL View



(a) PC Update Logic Module



(b) Program Counter (PC) RTL Schematic

Figure 4: PC Submodules: Update Logic and Register Implementation

3.3.1 Testbench (i281_cpu_tb.v)

The testbench instantiates the DUT and:

- Generates a 10 ns clock signal
- Applies reset for 2 clock cycles
- Runs simulation for 1500 clock cycles
- Dumps waveform to VCD file for GTKWave visualization
- Calls print task to display final memory contents

4 Simulation Results and Verification

4.1 Simulation Environment & Script

The verification was performed using the ModelSim environment. A Tcl script was created to automate the compilation, simulation initialization, and waveform configuration. This ensures a consistent and reproducible verification process.

Listing 2: ModelSim Simulation Script (sim.do)

```
# 1. Clean and Initialize
if {[file exists work]} { vdel -all }
vlib work
vmap work work

# 2. Compile (Order: Sub-modules -> Top -> Testbench)
vlog pc.v pcupdatelogic.v alu.v flags_register.v registerfile.v
   control.v opcode_decoder.v dmem.v cmem.v i281_cpu.v i281_cpu_tb.v

# 3. Start Simulation
vsim -voptargs="+acc" work.i281_cpu_tb

# 4. Configure Waveforms
radix -hex

# Top Level
add wave -noupdate -divider "Top Level"
add wave -noupdate /i281_cpu_tb/clk
add wave -noupdate /i281_cpu_tb/reset
add wave -noupdate /i281_cpu_tb/Switches
```

```
# CPU Internals
add wave -noupdate -divider "CPU Internals"
add wave -noupdate -label "PC Address" /i281_cpu_tb/DUT/
   pc_out_cm_addr
add wave -noupdate -label "OpCode" -bin /i281_cpu_tb/DUT/
   OpCode_Field
add wave -noupdate -label "ALU Result" /i281_cpu_tb/DUT/alu_result

# Flags
add wave -noupdate -divider "Flags"
add wave -noupdate -label "Zero (Z)"      /i281_cpu_tb/DUT/out_zero
add wave -noupdate -label "Negative (N)"  /i281_cpu_tb/DUT/
   out_negative
add wave -noupdate -label "Carry (C)"     /i281_cpu_tb/DUT/out_carry
add wave -noupdate -label "Overflow (O)"  /i281_cpu_tb/DUT/
   out_overflow

# Register File Details
add wave -noupdate -divider "Register Ports"
add wave -noupdate -label "Port 0 Data"      /i281_cpu_tb/DUT/RF/
   port0_data
add wave -noupdate -label "Port 1 Data"      /i281_cpu_tb/DUT/RF/
   port1_data
add wave -noupdate -label "Reg Write En"     /i281_cpu_tb/DUT/RF/
   c10_write_en
add wave -noupdate -label "Reg Write Data"   /i281_cpu_tb/DUT/RF/
   data_in

add wave -noupdate -divider "Specific Registers"
add wave -noupdate -label "Reg A" /i281_cpu_tb/DUT/RF/reg_A
add wave -noupdate -label "Reg B" /i281_cpu_tb/DUT/RF/reg_B
add wave -noupdate -label "Reg C" /i281_cpu_tb/DUT/RF/reg_C
add wave -noupdate -label "Reg D" /i281_cpu_tb/DUT/RF/reg_D

# Data Memory Array
add wave -noupdate -divider "Data Memory"
add wave -noupdate -label "Memory Array" -expand /i281_cpu_tb/DUT/
   dmem/memory

# 5. Run and Zoom
run -all
wave zoom full
```

4.2 Waveform Analysis

The simulation successfully demonstrates correct CPU operation with all key signals transitioning as expected. Figure 5 illustrates the global timing diagram of the processor executing the Bubble Sort algorithm.

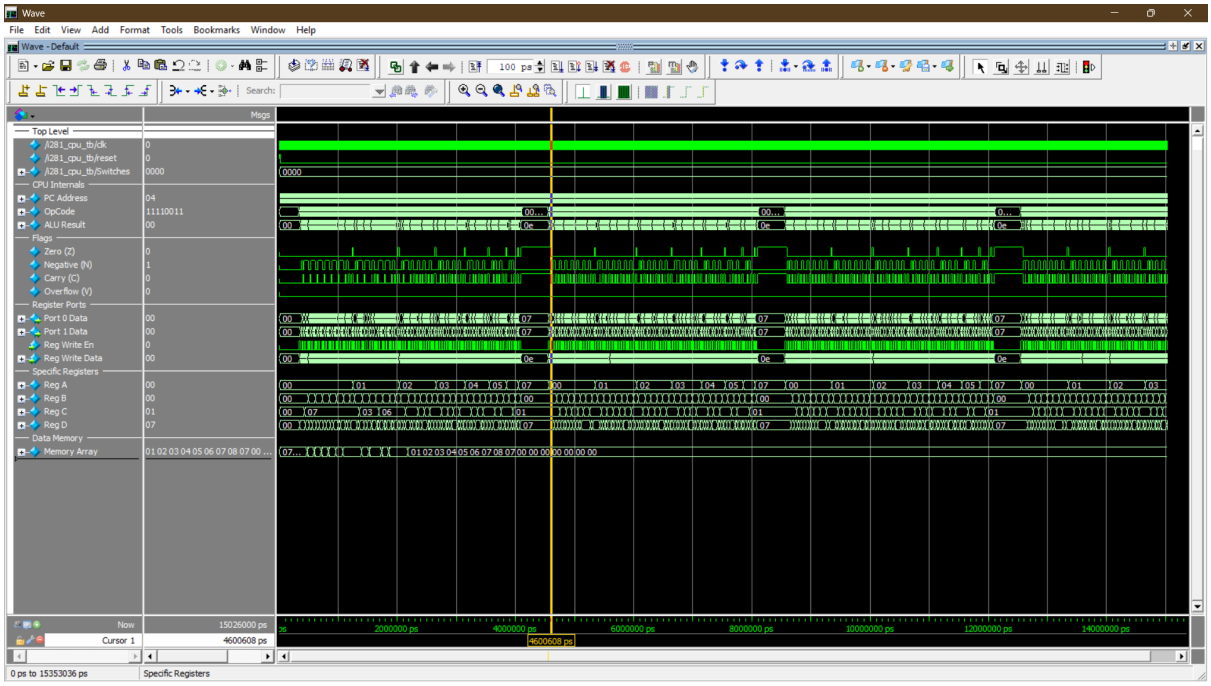


Figure 5: Simulation Waveform showing PC updates, ALU operations, Flag transitions, and Register contents.

Observations from the waveform:

- **Program Counter:** Increments sequentially during normal instruction execution and jumps correctly during branch operations
- **Instruction Fetching:** Code memory correctly outputs 16-bit instructions based on PC address
- **Register File:** Register contents update correctly on write operations and are correctly read via dual read ports
- **ALU Operations:** Arithmetic results and flags generate correctly for all instruction types
- **Data Memory:** Array elements are correctly read and written during bubble sort execution
- **Flags:** Status flags transition appropriately during comparison and branch operations

4.3 Register File Contents

Throughout execution, the register file dynamically updates with working variables and intermediate results:

4.4 Data Memory Contents

The initial contents of data memory (addresses `0x0` to `0x7`):

Table 3: Initial Unsorted Array Stored in Data Memory

Address	Decimal	Hex	Binary
0x0	7	0x07	0000 0111
0x1	3	0x03	0000 0011
0x2	2	0x02	0000 0010
0x3	1	0x01	0000 0001
0x4	6	0x06	0000 0110
0x5	4	0x04	0000 0100
0x6	5	0x05	0000 0101
0x7	8	0x08	0000 1000

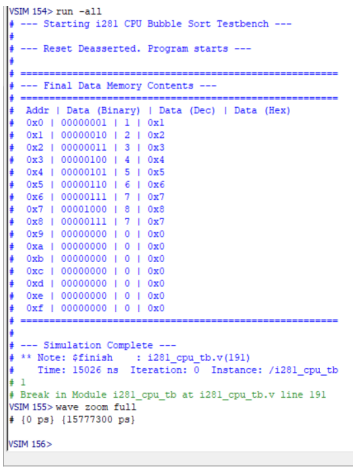


Figure 6: Simulation Transcript showing the Final Data Memory Contents (Sorted Array).

After 1500 clock cycles (algorithm completion), the array is sorted in ascending order. This is confirmed by the simulation transcript shown in Figure 6.

5 Design Challenges and Solutions

5.1 Instruction Scheduling

Challenge: Avoiding read-after-write hazards while maintaining single-cycle operation.
Solution:

- Placed register file write on rising edge clock
- Placed ALU operations combinationally after register reads
- Ensured sufficient setup time through careful signal routing

5.2 Memory Initialization

Challenge: Reliably initializing both code and data memory with pre-programmed values.
Solution:
Initially, the data memory was hardcoded using `initial` blocks. To make the design more flexible and user-configurable, we replaced this approach with external memory loading using the commands `$readmemh` (HEX format) or `$readmemb` (binary format).

This allows the user to modify the array without editing Verilog code.

Listing 3: Data Memory Initialization Using External File

```
initial begin
    if (!$readmemh("dmem.mem", memory)) begin
        $display("ERROR: dmem.mem file not found!");
    end else begin
        $display("Data Memory loaded successfully from dmem.mem");
    end
end
```

Example of dmem.mem (User Input Array):

```
07  03  02  01  06  04  05  08  07  00  00  00  00  00  00  00
```

Advantages of this method:

- User can modify input data without changing Verilog code.
- Easier testbench automation with different datasets.
- Increases reusability and scalability of the CPU design.

5.3 Flag Management

Challenge: Correctly generating and maintaining status flags across diverse instruction types.
Solution:

- Designed dedicated flags register module with synchronous write
- Control unit selectively enables flag updates only when needed
- ALU provides all four flags (Zero, Negative, Carry, Overflow) simultaneously

6 Conclusion

The i281 processor design successfully demonstrates fundamental digital system principles through a working implementation of a 8-bit CPU capable of executing non-trivial algorithms. The modular Verilog design facilitates verification and potential future enhancements. Simulation results confirm correct operation of all major CPU subsystems including instruction fetching, decoding, register operations, ALU computations, and data memory access.

The bubble sort implementation on this processor validates the hardware design through a practical application, demonstrating both the capability and efficiency of the architecture.