

i281 CPU

A Course Project for EE224 Digital Systems

Team Name: **googoogaga**

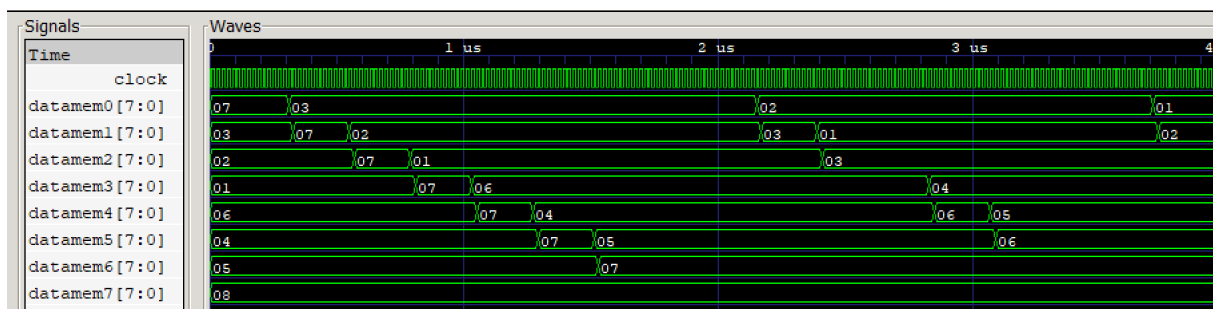
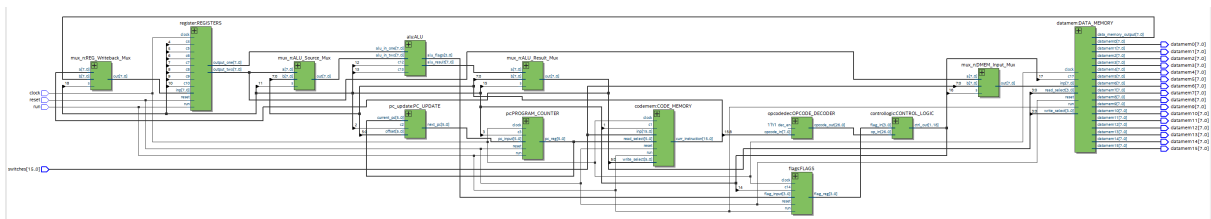
Team Members:

Visharad Srivastava (24B1202)

Shridhar Patil (24B1261)

Jai Bellare (24B1307)

Submission Date: **23 November, 2025**



Contents

1	Introduction	3
2	Architecture	3
2.1	Overview	3
2.2	Memories	4
2.3	Instructions and Opcode Decoder	5
2.4	Control Logic	6
2.5	ALU and Registers	7
3	Verilog implementation	7
3.1	Directory structure	7
4	Assembler	8
5	Running Bubble Sort, Memory Waveforms	8
6	Multi-cycle Implementation	9
6.0.1	Brief Description of Important States:	10

1 Introduction

In this report, we present our design, Verilog implementation, and simulation of the i281, an 8-bit CPU. This CPU was made for the course project of EE224 Digital Systems at IIT Bombay, instructed by Prof. Sachin Patkar. We follow the work of Prof. Alexander Stoytchev and his group at Iowa State University.

Find the Verilog code, waveforms and documentation at https://github.com/jjbel/i281_cpu

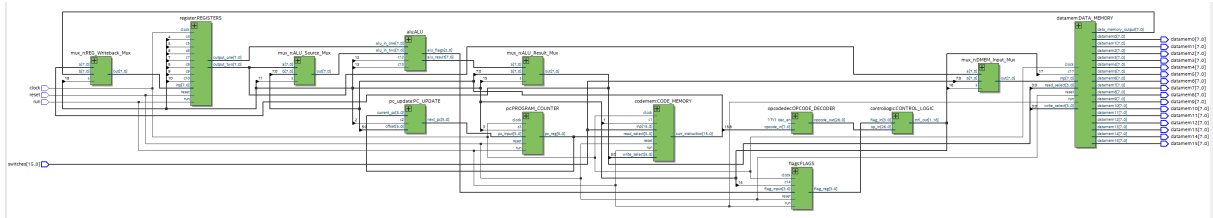


Figure 1: Netlist of the CPU

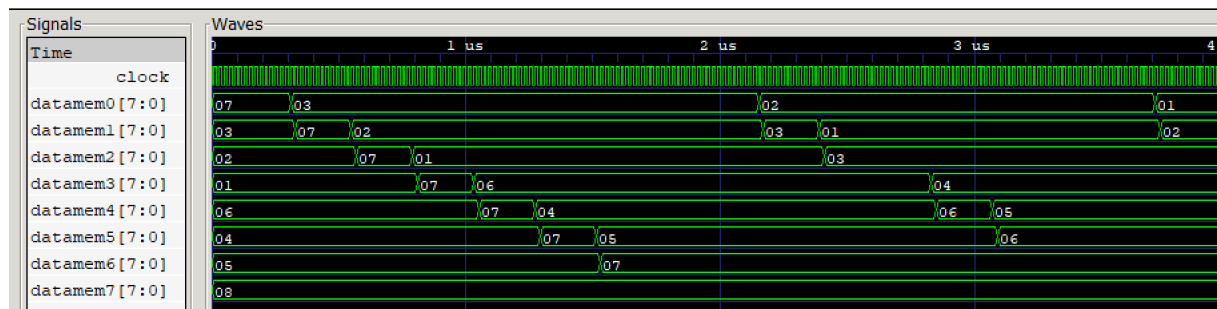


Figure 2: Bubble Sort Waveform

Below we explain the architecture and design choices, followed by a demo of running the bubble sort algorithm.

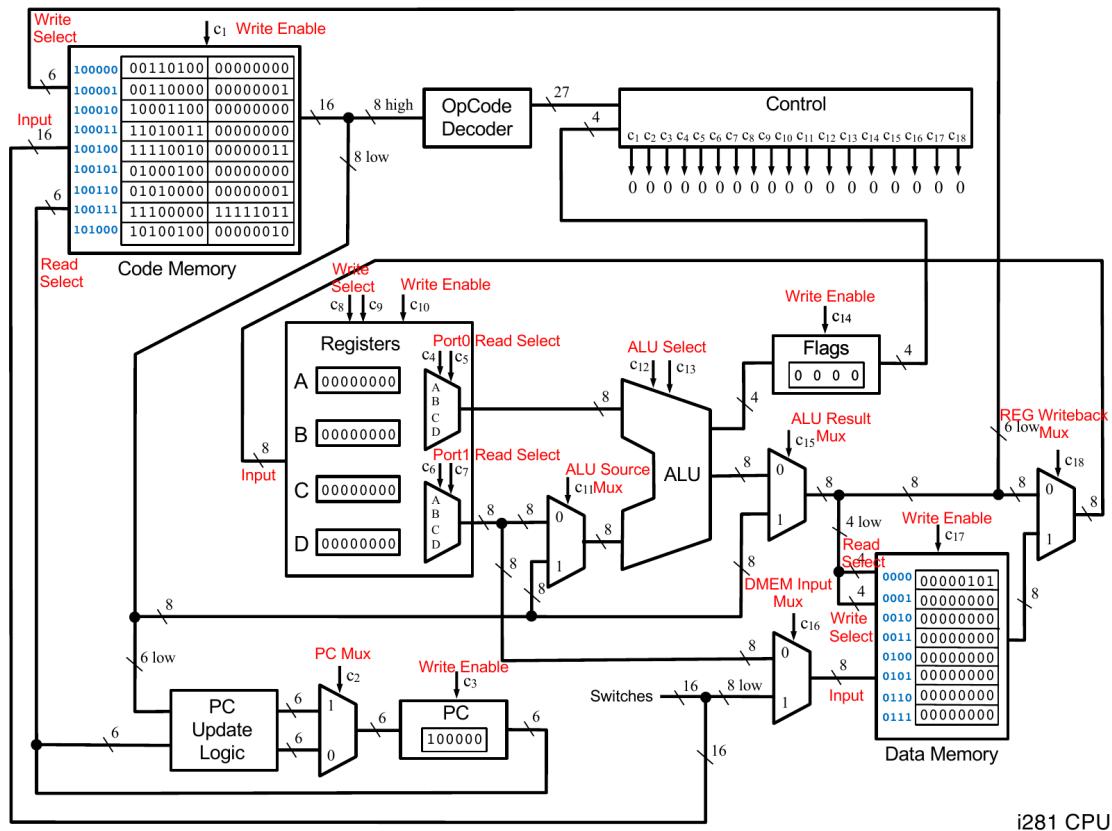
2 Architecture

2.1 Overview

The CPU is of the Harvard-style architecture, with separate cod (64x 16-bit) and data (16x 8-bit) memories.

The CPU is single-cycle, with a purely combinational control logic block. There are 26 assembly instructions, which map to 23 Opcodes. The control logic block converts the decoded opcode to 18 control signals c1 to c18.

The ALU supports addition, subtraction, left and right bitshifts. It operates on 4x 6-bit registers A,B,C,D.



Drawing conventions: inputs enter from the left, outputs exit from the right, control lines are vertical arrows on the top.

2.2 Memories

1. **Code Memory** : 64 rows of 16-bit registers, hence a 6-bit Program Counter (PC)
2. **Data Memory** : 16 rows of 8-bit registers

The 6-bit program counter specifies the current instruction.

2.3 Instructions and Opcode Decoder

NOOP	NO Operation
INPUTC	INPUT into Code memory
INPUTCF	INPUT into Code memory with oFfset
INPUTD	INPUT into Data memory
INPUTDF	INPUT into Data memory with oFfset
MOVE	MOVE the contents of one register into another
LOADI	LOAD Immediate value
LOADP	LOAD Pointer address
ADD	ADD two registers
ADDI	ADD an Immediate value to a register
SUB	SUBtract two registers
SUBI	SUBtract an Immediate value from a register
LOAD	LOAD from a data memory address into a register
LOADF	LOAD with an oFfset specified by another register
STORE	STORE a register into a data memory address
STOREF	STORE with an oFfset specified by another register
SHIFTL	SHIFT Left all bits in a register
SHIFTR	SHIFT Right all bits in a register
CMP	CoMPare the values in two registers
JUMP	JUMP unconditionally to a specified address
BRE	BRanch if Equal
BRZ	BRanch if Zero
BRNE	BRanch if Not Equal
BRNZ	BRanch if Not Zero
BRG	BRanch if Greater
BRGE	BRanch if Greater than or Equal

Figure 3: Assembly Instructions

There are 26 assembly instructions, which map to 23 Opcodes. LOADI/LOADP, BRE/BRZ and BRNE/BRNZ are aliased.

The decoder produces 27 outputs: a 23-bit one-hot opcode, and 4 outputs to select registers A,B,C or D.

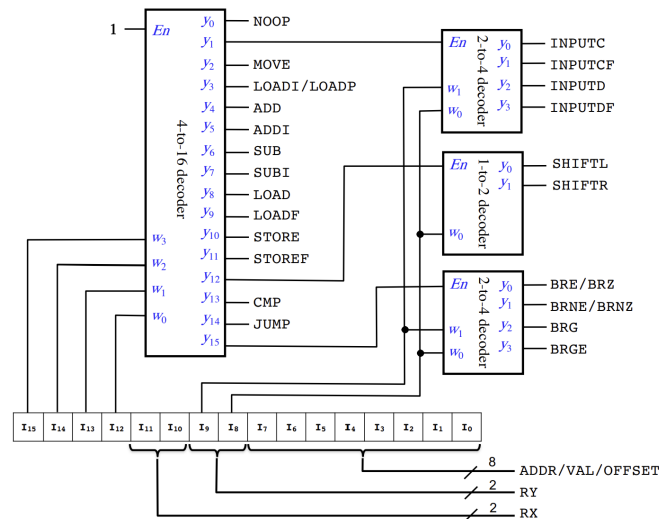


Figure 4: Opcode Decoder Diagram

Each 16-bit register of the code memory encodes an instruction:

1. $\text{instr}[15:12]$ majorly chooses the instruction, using a 4-to-16 decoder which produces a one-hot output
2. $\text{instr}[11:10]$ further choose which register A,B,C or D to use for the first operand

- instr[9:8] either choose which register to use for the second operand, or which of (INPUTC, INPUTCF, INPUTD, INPUTDF), or of (BRE/BRZ, BRNE/BRNZ, BRG, BRGE)
- instr[8] also decides whether to shift left or right
- The lower byte instr[7:0] is NOT routed through the decoder. It is either an 8-bit immediate value, or a 6-bit address for data memory or branching

Immediate values allow using compile-time constants for loading into registers, or for arithmetic operations.

Jumps specify a fixed 6-bit relative-jump to unconditionally move the program counter by. Branches allow conditionally jumping based on comparisons from the flags output of the ALU. Jumps and branches are necessary for control flow like if/else statements and loops.

2.4 Control Logic

Since the CPU is single-cycle, the control logic block is purely combinational. The control logic block maps the 27-bit opcode decoder output to 18 control signals c1 to c18.

18 control lines

	c ₁	c ₂	c ₃	c ₄	c ₅	c ₆	c ₇	c ₈	c ₉	c ₁₀	c ₁₁	c ₁₂	c ₁₃	c ₁₄	c ₁₅	c ₁₆	c ₁₇	c ₁₈
NOOP			1															
INPUTC	1																	
INPUTCF	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								
LOADI/LOADP			1					X1	X0	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															

23 one-hot encoded OPCODEs

Figure 5: Control Signal Output Table

The Control Signal Output Table specifies what outputs (C1 to c18) a given decoded opcode produces. For example, the ADD instruction:

- specifies the operands in c_4c_5 and c_6c_7
- sets c_{10} to route the ALU output to the destination register
- and chooses the ALU's addition operation in $c_{11}c_{12}$

2.5 ALU and Registers

The ALU supports 4 operations:

1. left shift (SHIFTL)
2. right shift (SHIFTR)
3. add (ADD)
4. subtract (SUB)

The ALU operates on 4x 8-bit registers: A,B,C,D.

3 Verilog implementation

Find the Verilog code, waveforms and documentation at https://github.com/jjbel/i281_cpu

We implemented each block in a separate Verilog module. For example the folder `i281_main/Control Logic` contains the control logic in `controllogic.v`, its testbench in `controllogic_tb.v`.

3.1 Directory structure

```
i281_main
|   dump.vcd
|   i281_main.qpf
|   i281_main.qsf
|   i281_toplevel.v
|   i281_toplevel_tb.v
---ALU
|   |   alu.v
|   |   alu_tb.v
---Assembler
|   |   BIOS_Hardcoded_High.v
|   |   BIOS_Hardcoded_Low.v
|   |   BubbleSort.asm
|   |   i281assembler.java
|   |   User_Code_High.v
|   |   User_Code_Low.v
|   |   User_Data.v
---Code Memory
|       codemem.v
---Control Logic
|   |   a.out
|   |   controllogic.v
|   |   controllogic_tb.v
---Data Memory
|       datamem.v
---Flags
|       flags.v
---OpCode Decoder
|   |   opcodedec.v
```

```

| | opcodedec_tb.v
---PC
| pc.v
---PC Update
| a.out
| pc_update.v
| pc_update_slide.png
| pc_update_tb.v
---Registers
| registers.v

```

4 Assembler

We use Prof. Stoytchev's assembler `i281assembler.java` to convert assembly code in a .asm file to machine code. It produces 4 verilog files: `BIOS_Hardcoded_High.v` and `BIOS_Hardcoded_Low.v` have an empty BIOS which just jumps to the user code at row 32. The user code is in `User_Code_Low.v` and `User_Code_High.v`. For example, this stores the bubble sort code. Finally, we give the input numbers to the bubble sort in `User_Data.v`. The `codemem` and `datamem` Verilog modules instantiate modules to load these hardcoded values on reset.

5 Running Bubble Sort, Memory Waveforms

Following is the contents of the data memory, registers, and flags while bubble sort is running. The array (7,3,2,1,6,4,5,8) is finally sorted in ascending order.

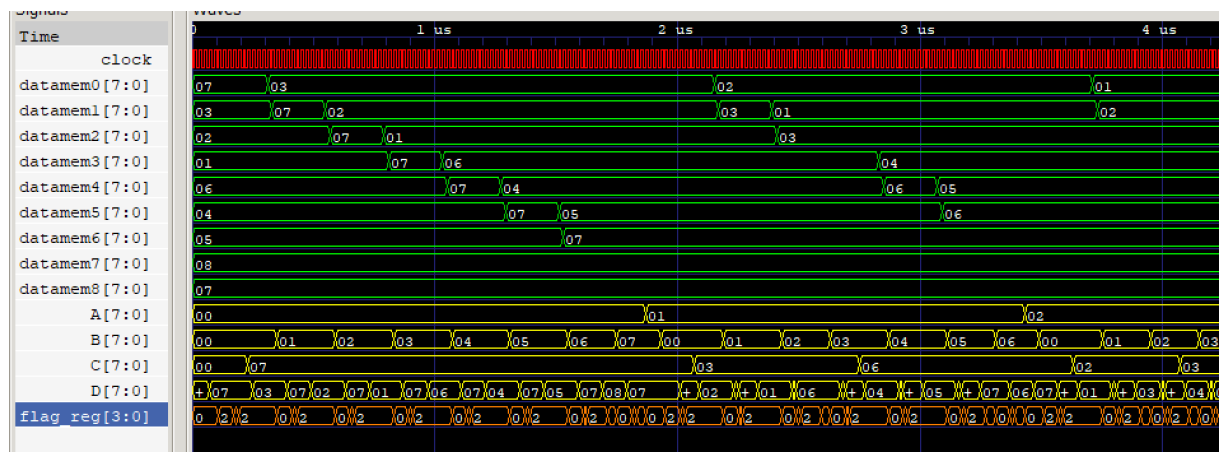


Figure 6: Bubble Sort Waveform

6 Multi-cycle Implementation

The Multicycle Implementation of i281 CPU requires us to create an FSM control block which generates a different FSM for each instruction. Since we are to use the same assembler, the Opcode decoder is kept the same.

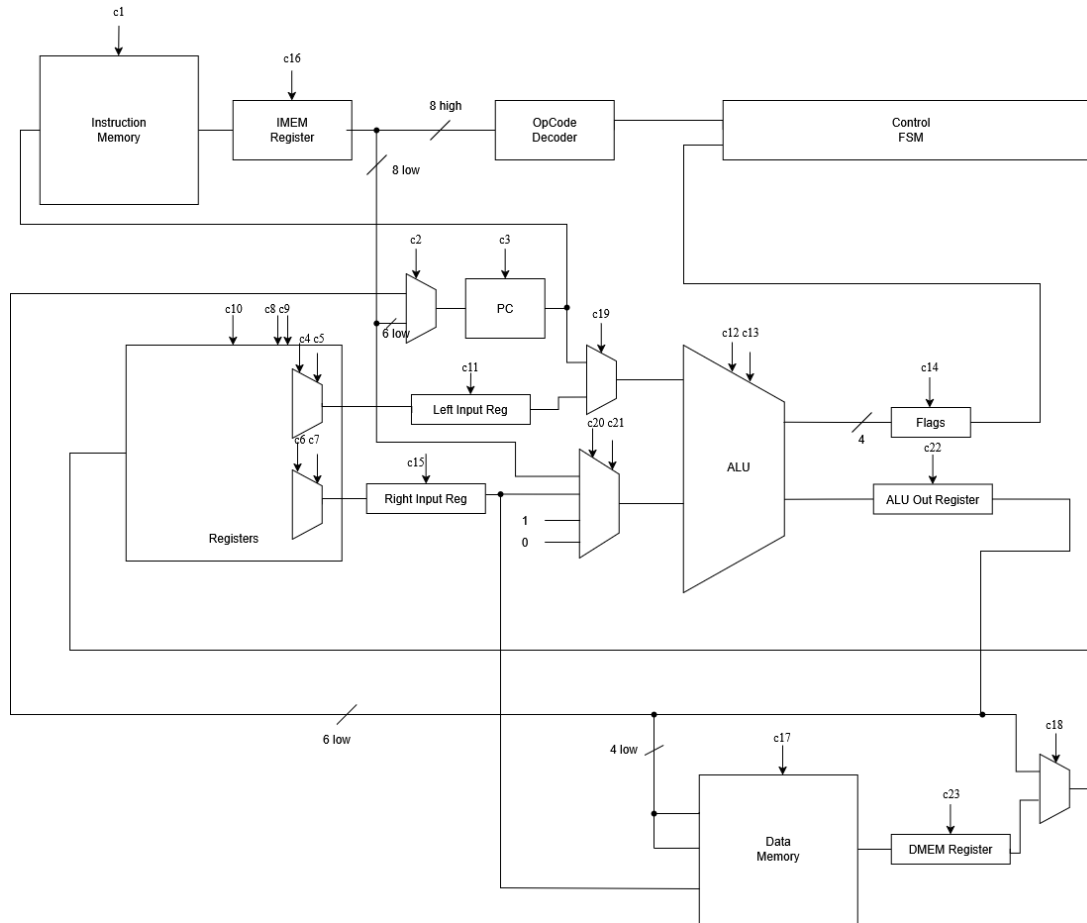


Figure 7: Datapath Diagram

A fixed set of states is first created, each of which has a definite output sequence of *c*[24:1]. Each instruction will be a FSM of a subset of these states. The set of states used is provided below:

```

1  localparam IF      = 5'd0,
2      ID             = 5'd1,
3      ExALU          = 5'd2,
4      ExADDR          = 5'd3,
5      ExBRANCH        = 5'd4,
6      ExJUMP          = 5'd5,
7      MemREAD         = 5'd6,
8      MemWRITE        = 5'd7,
9      WbALU           = 5'd8,
10     WbLOAD           = 5'd9,
11     ExLOAD           = 5'd10,
12     ExLOADI          = 5'd11,
13     ExLIR            = 5'd12,
14     ExMOVE           = 5'd13,
15     ExSWAPREG        = 5'd14;

```

A basic representation of the control logic FSM is as follows:

```
1 module controlfsm (...)
2     always@(posedge clock or posedge reset)
3         if(reset) begin
4             state <= IF;
5         end
6         else begin
7             state <= next_state
8         end
9
10    always@(*) begin
11        case(opcode[22:0])
12            instruction = f(opcode_in)
13        endcase
14    end
15
16    always@(*) begin
17        next_state = g(state,instruction)
18    end
19
20    always@(*) begin
21        case(state)
22            c(I)
23        endcase
24    end
25
26 endmodule
```

For example, a NOOP command will cycle from IF (Instruction fetch) to ID (Instruction Decode) and back to IF (instruction fetch), where the next instruction is run.

```
1 casez ({
2     state, instruction
3 })
4 {
5     IF, 5'bzzzzz //instruction is reg[4:0], any instruction has
6         next ID
7     } : begin
8         next_state = ID;
9     end
10    // NOOP OPERATION
11    {
12        ID, 5'd0
13    } : begin
14        next_state = IF;
15    end
16 end
```

6.0.1 Brief Description of Important States:

- IF: (Instruction Fetch): Next instruction is fetched into the IMEM Register. PC and value 8'b00000001 are loaded into ALU operands and addition is done to calculate $PC + 1$ (the default next address to read from). (This is a difference to single-cycle variant where PC what updated with a separate adder)
- ID: Instruction decode: Here values of register[RX], register [RY] are loaded into left input reg and right input reg for calculations. The instruction is decoded from the opcode provided using a binary decoder as opcode_in[22:0] are one-hot encoded.

