Birzeit University

Department of Electrical & Computer Engineering

First Semester 2024/2025

HW Design Lab

Project#1 – Physical Design

Objective:

The objective of this project is to provide hands-on experience in digital design and physical implementation by scaling an existing 8-bit Processor design to a 32-bit Processor. This involves modifying the provided 8-bit Processor code to support 32-bit operations, updating data paths, control units, and memory interfaces while understanding the implications of increased complexity and performance. The project also covers the entire physical design process, including Synthesis, Floor-planning, Placement, Routing, and Clock tree synthesis, ensuring that the final layout meets timing, power, and area constraints. Comprehensive documentation of the design process, challenges, and solutions, along with a final presentation and demonstration of functionality, are integral parts of this project to solidify practical knowledge in digital and physical design principles.

Note1: The project specifications and codes are taken from this site
https://studentsxstudents.com/simple-8-bit-processor-design-and-verilog-implementation-part-1-8735fac284b
https://studentsxstudents.com/simple-8-bit-processor-design-and-verilog-implementation-part-2-f1465be2cbe2

Note2: We copied all the needed codes to this link
https://birzeit0-my.sharepoint.com/personal/rzabadi_birzeit_edu/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Frzabadi%5Fbirzeit%5Fedu%2FDocuments%2FHandware%20codes&ga=1

8-bit Processor Design and Verilog implementation:

The implementation will demonstrate the design of a simple 8-bit single-cycle processor, incorporating an ALU, a register file, and control logic using Verilog HDL. The design requires a straightforward instruction set architecture. For this processor, the instructions implemented include add, sub, and, or, mov, loadi, j, and beq. All instructions follow a uniform structure as depicted in the accompanying diagram as shown below.

OpCode 31:24	Destination 23:16	Source 1 15:8	Source 2 7:0

Instruction format:

The instruction format is 32 bits in size, divided into four 8-bit sections:

1. **OpCode:** specifies the basic operation to be performed according to the instruction set.
2. **destination part** indicates the register number where data will be written.
3. **Source 1 and Source 2** specify the register numbers from which data is read.
4. **Source 2** also serving as an immediate value in certain instructions.

8-bit single-cycle processor which includes:

- **ALU:**

This ALU will support five operations: move (used in both `loadi` and `move` instructions), add, sub, and or. Below is a simplified diagram illustrating the structure of the ALU.

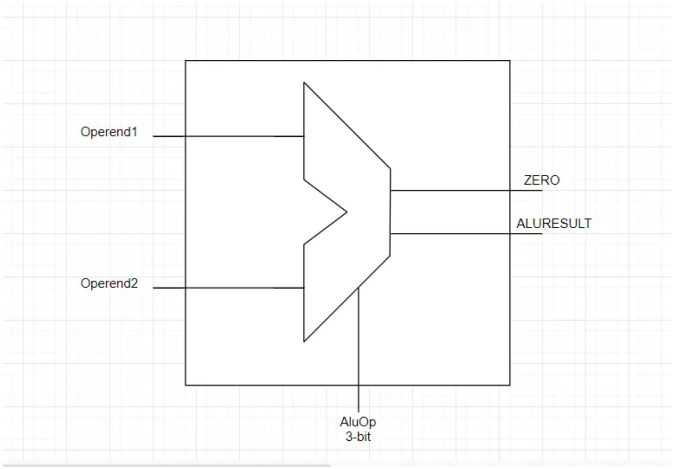

Figure 1 alu diagram

- AluOp is the 3-bit signal which is generated by Opcode decoding
- Operend1 and Operend2 will give the values to do the operation.
- ALURESULT will output the result
- ZERO is the signal that shows whether the result is zero or not. It is used in beq instruction.

Table 1 outlines the specific functions that need to be implemented.

Select	Function	Description	Supported Instructions	Latency
000	Forward	(forward DATA2 into RESULT) DATA2 -> RESULT	loadi, mov	#1
001	ADD	(add DATA1 and DATA2) DATA 1 + DATA2 -> RESULT	add, sub	#2
010	AND	(bitwise AND on DATA 1 with DATA2) DATA 1& DATA2 -> RESULT	and	#1
011	OR	(bitwise OR on DATA 1 with DATA2) DATA 1 I DATA2 -> RESULT	or	#1
1xx	Rederved	Reserved for future functional units	-	-

You can find the Verilog code for ALU here

- **Register:**

In this design, a register module is created with eight 8-bit registers. The structure of the register module is outlined below.

- The module has six inputs and two outputs.
- Writing to the registers occurs on the positive edge of the CLK signal, while reading is performed asynchronously.
- A WriteEnable control signal determines whether data should be written or not during each positive edge of the clock.
- ReadAddress signals specify which registers to read
- WriteAddress specifies the register to which data should be written.

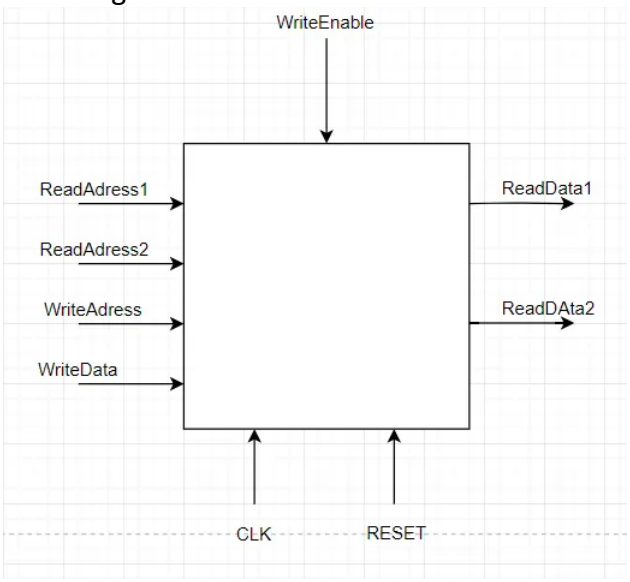

Figure 2 register diagram

You can find the Verilog code for Register here

- **CPU:**

To compose a working CPU using the previously discussed components, we will implement the instruction fetching mechanism along with a program counter (PC) register that tracks the address of the next instruction to be executed. The top-level module will be defined as module cpu(PC, INSTRUCTION, CLK, RESET).

- Instruction Fetching: We will design a combination logic unit to decode the fetched instruction, extract the opcode, source or destination registers, and immediate values. Based on the opcode, the control signals will be generated and sent to the register file, ALU, and other components of the CPU.
- Control Signals: The control signals will be generated dynamically during the fetch stage, according to the extracted opcode, and then sent to the corresponding components (register file, ALU, etc.).
- Program Counter (PC): The PC will keep track of the address of the next instruction. To fetch the next instruction, the PC value will be incremented by 4, since each instruction is 4 bytes in length.
- Timing Delays: To simulate realistic instruction fetching and decoding, we will add artificial timing delays:
    - Instruction Decode: Takes one time unit (#1).
    - PC Update: Takes one time unit (#1).
    - Instruction Memory Read: Takes two time units (#2).

To implement a dedicated adder for incrementing the Program Counter (PC) by 4, we can design a simple module that adds 4 to the current value of the PC at every clock cycle. This increment will ensure that the PC points to the next instruction in memory, as each instruction is assumed to be 4 bytes (32 bits).

```
//this    adder    is    to    increment    pc    value    with    4
module                                        adder(PCINPUT,RESULT);
 input                          [31:0]                          PCINPUT;
 output                          [31:0]                          RESULT;
 reg                                                             RESULT;
 always@(PCINPUT)
 begin
 RESULT                  =                PCINPUT+                4;
 end
endmodule
```

Figure below shows the full overview of the CPU diagram

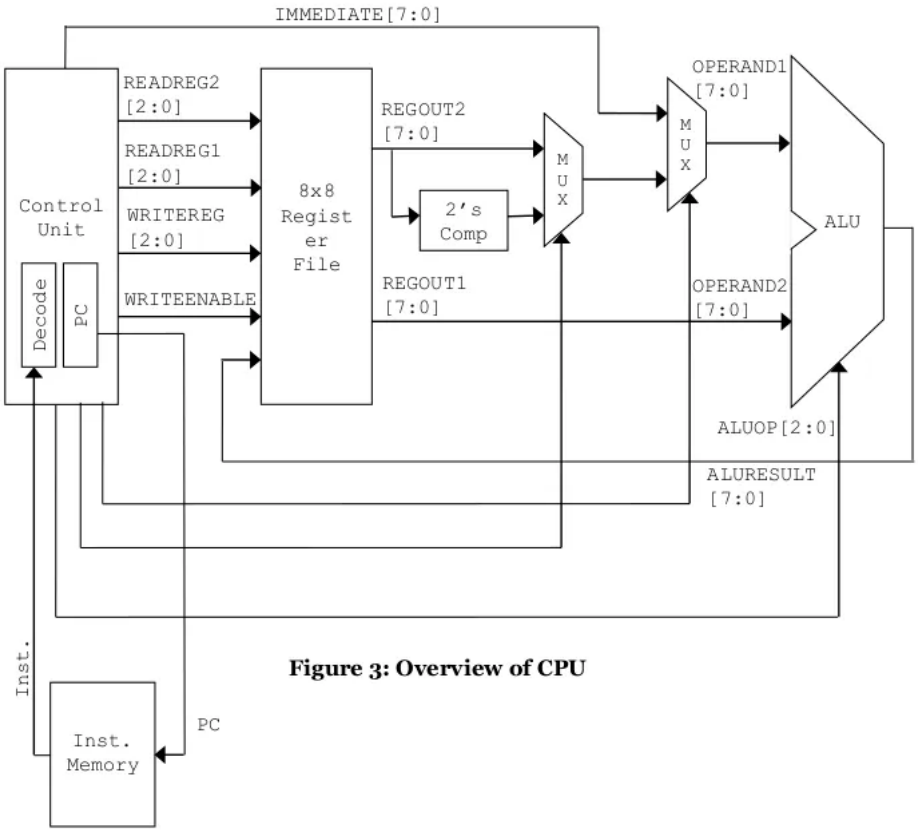

Figure 3: Overview of CPU

Figure 3 CPU diagram

-Note as shown in Figure 3 we need to implement the multiplexer and the two's complement modules.

You can find the Verilog code of CPU , multiplexer and two's complement here

Project Phases and Implementation Steps:

1. Initial Analysis and Planning:
   - **Code Review:** Understand the architecture and functionality of the provided 8-bit CPU code.
   - **Design Specification:** Draft a document outlining the key differences and requirements for a 32-bit CPU compared to an 8-bit CPU (e.g., data path, control unit, instruction set).
   - **Block Diagram Creation:** Draw a high-level block diagram that maps out the components and interconnections needed for the 32-bit CPU.

2. Modification and Expansion of Code:
   - **Data Path Expansion:** Modify the data path width from 8-bit to 32-bit, including registers, ALU (Arithmetic Logic Unit), and buses.
   - **Instruction Set Adjustment:** Update the instruction decoder and control unit to handle 32-bit instructions.
   - **Debug and Simulate:** Simulate the modified code to ensure correct functionality using HDL simulators .
3. Physical Design Steps:
   - **Synthesis**: Use synthesis tools (e.g., Synopsys Design Compiler) to convert the modified HDL code into a gate-level netlist.
   - **Floorplanning and Power Planning:** Plan the physical layout of the CPU components to optimize space, timing, and power.
     - Define chip dimensions and placement of major blocks (e.g., ALU, Control Unit, Register File).
     - Consider power planning (Power/Ground grids).

   - **Placement:** Place the cells on the die according to the floorplan.
     - Automate the placement of standard cells using provided tools.
     - Optimize for area and timing.

   - **Clock Tree Synthesis (CTS):** Implement a clock distribution network that ensures minimal skew and latency.
     - Design and analyze the clock distribution network.
     - Minimize clock skew and power consumption.

   - **Routing:** Connect the components through routing, ensuring minimal delay and avoiding congestion.
     - Perform global and detailed routing to connect all the components.
     - Resolve routing congestion issues.

4. Verification and Validation:
   - **Static Timing Analysis (STA):** Ensure that the design meets timing requirements using tools like Synopsys PrimeTime.
   - **Power Analysis**: Analyze power consumption and identify potential areas for optimization.

- **Design Rule Check (DRC) and Layout Versus Schematic (LVS):** Verify the layout meets fabrication rules and matches the schematic.
- **Post-Layout Simulation:** Run simulations to validate the final design's functionality and timing after layout completion.

Final Guidelines:

1) This is individual project.
2) Deadline till midnight of Saturday 28/12/2024
3) Report is needed, in the report you should show the used commands and the screenshots of your design.
4) The icc2 user-guide can be used to see command options and try variety of different options and show the output
5) For example, in the placement command you can try the option of timing driven, congestion driven, and area driven placements. You can try to add your own Macros, blockages …etc. You can show whatever options you read about and want to demonstrate …

GOOD LUCK