Development Phase

Guidelines and Reference Design

Dr. Dheya Mustafa

9/10/2024

1. Overview

This document outlines the requirements and guidelines for the second phase of the JoSDC'24 Competition. The three phases are: (i) Qualifying (ii) Development and (iii) Innovation

1. **Qualifying Phase**: Teams focus on verification and debugging. The goal is to demonstrate a basic working design that can execute a minimal set of MIPS instructions. Due date **13\10\2024**

2. **Development Phase**: The second phase involves expanding and refining the design. Teams should focus on supporting an extended instruction set, handling hazards, and introducing a pipelined architecture to improve performance.

3. **Innovate Phase**: The final stage emphasizes creative improvements and optimizations. Teams can propose and implement advanced features to enhance performance, efficiency, or functionality.

2. Development Phase Details

Development Phase includes three major tasks: Basic processor, Pipelined processor, and a comparative evaluation of both processors.

2.1 Basic Processor:

Objective: Create a sequential (single-cycle\multi-cycle\non-pipelined) processor that supports a basic set of MIPS instructions. This will serve as the foundation for future task.

Basic Requirements:

- Implement the following instructions in hardware (not pseudo-instruction):

- **Arithmetic and Logic**: `ADD`, `SUB`, `AND`, `OR`, `NOR`, `XOR`, `XORI`, `SLT`, `SGT`, `ADDI`, `ORI`.

- **Branches & Jumps**: `BEQ`, `BNE`, `JR`, `JAL`.

- **Load and Store**: `LW`, `SW`.

- **Shift Operations**: `SLL`, `SRL`.

- implement pseudo-instruction: `**BLTZ**`, `**BGEZ**`,
- Ensure that the design handles basic data dependencies and control flow.
- Verify the functionality using comprehensive test cases covering all instructions
- Teams must provide a brief report detailing the design (datapath, control unit), any modifications, and a summary of test results.
- **Functional and performance testing** are critical to ensure that your processor works as intended. You should aim for a high **coverage percentage** during your tests, verifying that all instructions and corner cases are handled correctly.

Table 1 lists the required CPU instructions, their semantics, and examples using MIPS assembly language format.

Table 1 Required Instructions

Instruction	Semantics	Example
ADD	Adds two registers and stores the result in a register	`ADD $t0, $t1, $t2`
SUB	Subtracts the second register from the first and stores the result	`SUB $t0, $t1, $t2`
AND	Performs a bitwise AND operation on two registers	`AND $t0, $t1, $t2`
OR	Performs a bitwise OR operation on two registers	`OR $t0, $t1, $t2`
XOR	Performs a bitwise XOR operation on two registers	`XOR $t0, $t1, $t2`
NOR	Performs a bitwise NOR operation on two registers	`NOR $t0, $t1, $t2`
SLT	Sets a register to 1 if the first register is less than the second	`SLT $t0, $t1, $t2`
LW	Loads a word from memory into a register	`LW $t0, 4($t1)`
SW	Stores a word from a register into memory	`SW $t0, 4($t1)`
BEQ	Branches if the contents of two registers are equal	`BEQ $t0, $t1, label`
BNE	Branches if the contents of two registers are not equal	`BNE $t0, $t1, label`
J	Jumps to the specified address	`J label`
JR	Jumps to the address contained in a register	`JR $ra`
JAL	Jumps to a specified address and links the return address in `$ra`	`JAL label`
ADDI	Adds an immediate value to a register	`ADDI $t0, $t1, 10`
ORI	Performs a bitwise OR operation with an immediate value	`ORI $t0, $t1, 10`
XORI	Performs a bitwise XOR operation with an immediate value	`XORI $t0, $t1, 10`
ANDI	Performs a bitwise AND operation with an immediate value	`ANDI $t0, $t1, 10`
SLL	Shifts a register value left by a specified number of bits	`SLL $t0, $t1, 4`
SRL	Shifts a register value right by a specified number of bits	`SRL $t0, $t1, 4`
SLTI	Sets a register to 1 if a register value is less than an immediate	`SLTI $t0, $t1, 20`
SGT	Sets a register to 1 if the first register is greater than the second	`SGT $t0, $t1, $t2`

Memory model

MIPS is 32-bit architecture, with 4GB memory space. However, due to FPGA limited size, we encourage you to use a small memory size tha sufficient to illustrate the design functionality. For example you can implement 1kB memory, utilizing the least 10 significant bits of PC. You

should clearly state you design choices. Describe memory model (byte addressable\ word addressable, little 3ndian\big endian).

2.2. Pipelined processor

Objective: Enhance the design to implement a pipelined processor.

Requirements:
We suggest implementing a 5-stage pipelined design as follows:

IF (Instruction Fetch)	**ID** (Instruction Decode)	**EX** (Execute)	**MEM** (Memory Access)	**WB** (Write Back)

- You can use any number of stages, with reasonable justification of your design choices in terms of its impact on performance, area, or power.
- Address and resolve data, control, and structural hazards using stalling, forwarding, branch prediction and flushing mechanisms.

- The pipeline must handle all types of **hazards**, including structural, data, and control hazards. To resolve these, you may need to implement mechanisms like **forwarding** and **stalling/flushing**, as necessary.

- The **new, modified, and final Datapath** should be clearly explained and presented in the report, with visual diagrams that help illustrate the changes and overall structure of the pipeline.

- All modifications and features related to the pipelined architecture must be clearly explained and documented. You should also explain the **flow of execution** through the pipeline stages, showing how instructions progress from fetch to write-back.

- Similar to the single-cycle design, you need a thorough **testing plan** that demonstrates functional and performance testing. You must cover all **critical cases** and ensure that hazards are correctly handled, while providing **performance results** to evaluate your pipelined design.

- All edits, modifications, or new features introduced to meet the project requirements should be clearly documented. This includes a thorough, **well-organized report** that outlines your design procedure, testing plan, and insights into your approach.

2.3 Comparative evaluation

Objective: Provide a detailed comparison between the basic and pipelined designs using various metrics (performance, power, area, etc).

Requirements:

- Measure clock cycles, execution time, and throughput for both single-cycle and pipelined designs.
- Develop different benchmark programs and use them to compare between the two processors. provide machine code for each benchmark.
- ensure that the design functions correctly and meets all required specifications.
- The comparison should first ensure that **both designs pass all functionality benchmarks** with correct results. Both designs must be tested on the same set of benchmarks to ensure fair comparison.
- **Time comparison** is a crucial part of the evaluation. Teams must apply suitable **timing constraints** to the clock using the techniques learned in training and the first phase. Measure key performance metrics such as **execution time**, **clock frequency**, **number of cycles**, and any additional relevant metrics.
- You are free to apply any **testing plan** for performance evaluation, but both designs should be tested against the benchmarks provided by the committee. You are also welcome to introduce **additional test cases or benchmarks** of your own if needed. Appendix A suggests a detailed test plan for you.
- The **size of the design in terms of logic elements** should be considered in your evaluation.
- Teams should provide a **comparison of advantages and disadvantages** for each design, outlining where each excels and where its weaknesses lie. Clearly describe in which scenarios the single-cycle design performs better and in which cases the pipelined design is more efficient.

  Finally, as part of your **performance evaluation conclusion**, you may present your **future improvement ideas** based on your findings. Since the design will continue to be refined in future phases (if qualified), you can discuss which areas of your design you plan to improve and how you intend to enhance its performance.

3. Testing and Verification Tools

Usually, Engineers build these tools to facilitate testing process.

3.1. Assembler:

While you can generate machine code manually, it is helpful to develop a simple assembler that translates your processor's instructions into machine code. This tool can be implemented in any high-level programming language like Python, C++, or Java.

Key Features:

- **Support for all instructions** used in your processor design.

- **Basic translation functionality**: Convert assembly instructions directly into their binary/Hex form.

This tool should be simple and quick to develop, primarily to help with testing and running your processor efficiently.

3.2 Cycle Accurate Simulator

In this part of the project, teams are required to implement a **Cycle Accurate Simulator** for the 5-stage pipelined processor that was previously implemented. The simulator can be written in any programming language (python, java, **C++)**.

The simulator should support all the instructions covered in both the single-cycle and pipelined designs. It is important to ensure the simulator is **scalable** and adaptable for future enhancements, as it will play a crucial role in the next phase when your design evolves.

Key Features of the Cycle Accurate Simulator:

1. **Trace Execution:**
    - Captures cycle-by-cycle traces, showing instruction fetches, memory accesses, and pipeline stalls.
2. **Data Forwarding & Hazard Detection:**
    - Implements data forwarding and detects hazards, adding stalls as needed.
3. **Performance Counters:**
    - Tracks metrics like instruction count, cycles, and stalls.
4. **Pipeline Visualization:**
    - Provides a visual representation of instruction flow through the pipeline stages.

4. Conclusion and Phase Submission

At the end of the development phase, teams are required to **submit** the following:

1. **Full Quartus project folder for basic processor:** both assembly code files and machine code file of the bechmarks should be submitted in a subdirectory named "**bench**" with proper "**readme**" file. The project should run with zero errors.

2. **Full Quartus project folder for** pipelined processor:

both assembly code files and machine code file of the bechmarks should be submitted in a subdirectory named "**bench**" with proper "**readme**" file. The project should run with zero errors.

3. **Full report following provided template** detailing your work, including:
    - **Design explanations** for both the single-cycle and pipelined processors.
    - **Modifications and improvements** made during the phase.
    - **Testing plan** and the results from functional and performance testing.
    - **Performance comparison** between the two designs.
    - **Signal analysis** and any other relevant documentation covered in previous sections.
    - **Executive summary**: (max 5 pages, font size 11pts) To summarize the core content of the document. It should provide the key points, findings, recommendations, and conclusions. Give committee members a high-level understanding for quick evaluation.
4. You may also submit **additional materials** such as:
    - **Supplementary diagrams** to better explain your Datapath or control unit.
    - **Additional benchmarks or test cases** that you applied to validate your design.
    - **Bonus tasks**, such as the instruction-to-machine code converter or Cycle Accurate Simulator (if implemented).

Submission Method:

Submission will be through JoSDC24 Google classroom.

Submission deadline: December 1, 2024