Discrete Event Simulator Report

25th March, 2024

1. Introduction

This report outlines the design and implementation of a discrete event simulator model for a computer architecture. The simulator is capable of modelling the behaviour of various components of a computer system through the use of events and an event queue. The primary objectives are to simulate the latency of the main memory and the latencies of different functional units, such as the ALU, multiplier, divider, and others. This report will describe the key components and design choices.

2. Implementation Details

2.1 Simulation of the program

The simulation of the program involves five stages of the processor, which are pipelined. Between each pair of stages, there is a latch, and each latch type is described by a separate class. The implemented stages include:

1. IF (Instruction Fetch) Stage: This stage has been implemented and is responsible for fetching instructions from memory. The instructions are then passed to the next stage through the IF-OF latch.
2. OF (Operand Fetch) Stage: This stage is responsible for decoding the instructions and preparing them for execution. The contents of the ID latch are determined by the specific instruction being processed.
3. EX (Execution) Stage: In this stage, arithmetic and logic operations are performed based on the decoded instructions. The contents of the EX latch depend on the operation being executed.
4. MA (Memory Access) Stage: This stage handles memory read

depend on the type of instruction and the data to be written back.

The simulation continues until an “end” instruction passes through the WB stage, at which point the simulation is marked as complete using setSimulationComplete().

2.2 Interlocks

The pipelined ToyRISC processor is an in-order pipeline implying the presence of Read after Write(RAW) Hazard and Branch Hazard which will be resolved by their respective Interlocks.

2.2.1 Data Interlock

RAW hazard is encountered when the source registers for the next coming instruction conflict with the destination register of the currently processing instruction. In the case of RAW hazard, the pipeline is stalled by passing two nop instructions. In a ToyRISC processor, RAW can be encountered for all instructions that use source registers, i.e. for all instructions except jmp instruction.

2.2.2 Branch Interlock

Branch hazard is encountered when the ”branch” instructions result in the ”isBranchTaken” to be true i.e. the instructions already in the IF and OF stages have to be nullified and new instructions are to be fetched. Branch hazard is possible in any of the conditional branch instructions i.e. **beq, blt, bgt, bne** and will definitely exist in **jmp** instruction.

2.3 Event Driven Simulation

In event-driven simulation, the simulator maintains an event queue that

stores events as tuples containing the following attributes:

- Event time: The time at which the event is scheduled to occur.

- Event type: A description of the type of event (e.g.,

MemoryRead,
MemoryResponse, ALUOperation,
etc.).

- Requesting element: The component or unit requesting the event.
- Processing element: The component or unit responsible

for
handling the event.

- Payload: Additional data or information associated with the event.

The simulation progresses in discrete time steps, with events scheduled to occur at specific times. When the current simulation time matches the event time, the corresponding processing element's handleEvent() function is invoked. Handling an event may lead to the generation of new events scheduled for the current or future clock cycles.

2.3.1 Component Design

To facilitate event-driven simulation, several components need to implement the Element interface. These components include main memory, functional units (e.g., ALU, multiplier, divider), and stages in the pipeline (e.g., Instruction Fetch).

2.3.2 Main Memory

The **MainMemory** component handles memory-related events, such as **MemoryRead**. It is responsible for reading data from memory and generating **MemoryResponse** events. The code snippet from the **MainMemory** class demonstrates the event handling logic for **MemoryRead** events.

provided in the **MainMemory** class demonstrates how **MemoryRead** events and **MemoryResponse** events are used to model main memory latency,

3.2 Modelling Functional Unit Latencies

The simulator can model the latencies of different functional units such as the ALU, multiplier, divider, etc. Events specific to these functional units can be defined, and their respective **handleEvent()** methods can simulate unit operation and event generation.

4. Test Case Performance

The following table summarises the performance of the simulator for each test case:

Test Case	Instructions	Dynamic Instructions	No. of Cycles	Instructions per Cycle
Fibonacci	21	78	3926	0.01986755
Descending	21	277	15325	0.018075041
Even or Odd	9	6	254	0.023622047
Prime	16	29	1402	0.020684736
Palindrome	16	49	2311	0.021202942

5. Conclusion

The discrete event simulator model described in this report provides a flexible and efficient way to model various components of a computer system. By using events and an event queue, it accurately simulates the behaviour of main memory and functional units while allowing for easy extension to other system components. This approach enables in-depth analysis of computer system performance