Computer Architecture Laboratory
Assignment 5

Upgrade the simulator to a discrete event simulator.

Inputs and Outputs

The inputs and outputs stay the same. The inputs are the configuration file, the path where the statistics file is to be created, and the object file. The output is the statistics file. Report throughput in terms of instructions per cycle in the statistics file.

New Source Files

Place the new source files: `Element.java`, `Event.java`, `EventQueue.java`, `ExecutionCompleteEvent.java`, `MemoryReadEvent.java`, `MemoryResponseEvent.java`, `MemoryWriteEvent.java` in the `generic` package.

Updation of `Simulator.java`

- Add the following data member to the class: `static EventQueue eventQueue;`.
- Add the following line to the constructor: `eventQueue = new EventQueue();`.
- Update the loop in `simulate()` to look like this:

```
while(not end of simulation)
{
        performRW
        performMA
        performEX
        eventQueue.processEvents();
        performOF
        performIF
        increment clock by 1
}
```

- Add this function to the class:

```
public static EventQueue getEventQueue()
{
        return eventQueue;
}
```

Discrete Event Simulator Model

An event is a tuple of the form: <`event_time, event_type, requesting_element, processing_element, payload`>. The event queue is a list of events ordered by time. An event is said to "*fire*" when the current clock cycle is equal to the `event_time`. When this happens, the `handleEvent()` function of the `processing_element` is invoked. Handling of an event may in turn lead to more events being generated, for the same clock cycle, or for some future clock cycle.

Decide which units you wish to work using events, and which directly through function calls from the main loop. For all units that you believe will recieve events, make them implement the `Element` interface. This will then require you to implement a `handleEvent()` function for that unit.

You may create more Event classes, or modify the existing ones.

Example

Below is a brief illustration of the Instruction Fetch stage. It is by no means complete. It is only to give you the basic idea.

```
public void performIF()
{
    if(IF_EnableLatch.isIF_enable())
    {
        if(IF_EnableLatch.isIF_busy())
        {
            return;
        }

        Simulator.getEventQueue().addEvent(
            new MemoryReadEvent(
                Clock.getCurrentTime() + Configuration.mainMemoryLatency,
                this,
                containingProcessor.getMainMemory(),
                containingProcessor.getRegisterFile().getProgramCounter()));

        IF_EnableLatch.setIF_busy(true);
    }
}

@Override
public void handleEvent(Event e) {
    if(IF_OF_Latch.isOF_busy())
    {
        e.setEventTime(Clock.getCurrentTime() + 1);
        Simulator.getEventQueue().addEvent(e);
    }
    else
    {
        MemoryResponseEvent event = (MemoryResponseEvent) e;
        IF_OF_Latch.setInstruction(event.getValue());
```

```
            IF_OF_Latch.setOF_enable(true);
            IF_EnableLatch.setIF_busy(false);
        }
    }
```

Below is the code snippet from the `MainMemory.java` class.

```
@Override
public void handleEvent(Event e) {
    if(e.getEventType() == EventType.MemoryRead)
    {
        MemoryReadEvent event = (MemoryReadEvent) e;
        Simulator.getEventQueue().addEvent(
            new MemoryResponseEvent(
                Clock.getCurrentTime(),
                this,
                event.getRequestingElement(),
                getWord(event.getAddressToReadFrom())));
    }
}
```

To Be Submitted

- A zip of the source files. They have to pass the test cases given for the previous assignment.
- A report that contains a table with
  - the number of cycles taken by each benchmark program,
  - the throughput in terms of instructions per cycle.

  Comment on your observations. Correlate with the nature of the benchmarks.