ASSIGNMENT 5

Sachin Kumar(220120019), Amogh R (220010005)

1 Introduction

In this assignment, we have modified the 5-stage pipelined processor to model the delays of the Main Memory unit and Execute unit using discrete event simulation.

2 Discrete Event Simulation

The simulator uses events and event handling to simulate delays in the processor and memory. The delays are modeled in the Instruction Fetch and the Memory Access stages for memory delays, and Execute stage for Arithmetic/Logical operation delays.

Four kinds of events are defined in the program:

1. `MemoryReadEvent`: Used in IF and MA
2. `MemoryWriteEvent`: Used in MA
3. `MemoryResponseEvent`: Used in IF and MA
4. `ExecutionCompleteEvent`: Used in EX

The discrete event simulation works as follows:

- An event is added to the `EventQueue` by a stage in the pipeline. The event fire delay is used as priority. Higher priority imply lower delays.
- The stage after enqueueing an event sets itself to busy until a response event is recieved. For handling the response events, `handleEvent()` functions are defined in the stages where required.
- Each cycle, the queue polls and fires the event which is polled. The `handleEvent()` function defined in the IF, EX or MA stages work on this event (depending on the processing element for the event).
- Once an event is handled, the processing element sets itself to not busy and waits for the next instruction to appear in its input latch.
- A stage when busy cannot accept new instructions in the input latch. This causes the pipeline to stall until the event is handled.

The individual stages handle events as follows:

1. The IF stage, when `performIF()` is called, checks whether the pipeline is stalled by the OF stage or if there is a pending fetch event (`MemoryReadEvent`). If not, it procceds to enqueue a `MemoryReadEvent` into the `EventQueue`. After the specified latency, it handles a `MemoryResponseEvent` by pushing the instruction recieved into the `IF_OF_Latch`.
2. The EX stage, when `performEX()` is called, checks whether the pipeline is stalled by the MA stage or if there is a pending execute event (`ExecutionCompleteEvent`). If not, it procceds to enqueue a `ExecutionCompleteEvent` into the `EventQueue`. After the specified latency, it handles a `ExecutionCompleteEvent` by pushing the ALUResult recieved into the `EX_MA_Latch`.
3. The MA stage, when `performMA()` is called, checks whether the pipeline has a pending read event (`MemoryReadEvent`) or a pending write event (`MemoryWriteEvent`) at the MA stage . If not, it procceds to enqueue a `MemoryReadEvent` or a `MemoryWriteEvent` into the `EventQueue` for `load` and `store` instructions respectively. After the specified latency, it handles a `MemoryResponseEvent` by pushing the data recieved into the `MA_RW_Latch`.
4. Branching when the IF stage is busy are handled by having the IF stage set the `isBranchWhenBusy` signal. However, the branch is not handled until the current instruction fetch does not finish, i.e., it does not switch to the new `branchPC` immediately after recieving the branch signal. Rather, it waits for the current instruction to be fetched, and sets it to a `nop` immediately.

3 Testing

The simulator was tested on the following files using default configuration:

`MainMemoryLatency` : 40 cycles

`DividerLatency` : 10 cycles

`MultiplierLatency` : 4 cycles

`ALULatency` : 1 cycle

Program file	No. instructions executed	No. cycles taken	Inst. per cycle
evenorodd.out	5	249	0.02008
prime.out	16	1364	0.01173
palindrome.out	41	2249	0.01823
fibonacci.out	51	3764	0.01355
descending.out	136	14805	0.00918

4 Conclusion

The IPC is hugely affected when there is a main memory latency of 40 cycles. The EX stage ALU Latency minimally affects the IPC. Programs with lots of loads and stores, and divisions/multiplications take a huge hit in performance (e.g. descending.out).