ASSIGNMENT 4

Sachin Kumar(220120019), Amogh R (220010005)

1 Introduction

In our latest assignment, we've made significant strides in refining our Java-based simulation of a 5-stage pipelined processor tailored for the ToyRISC architecture.
Now the processor have interlock feature like **lock disabling for Instruction Fetch(IF), BranchTake addition, and the utilization of a RegisterLockVector**, `represented by an integer array of size 32`. Unlike our prior iteration, where instructions progressed sequentially, our current implementation enables each stage to process one instruction per cycle. This pivotal refinement ensures the accurate sequencing of instruction execution while effectively mitigating hazards. Overall, these advancements represent a substantial leap forward in both the functionality and efficiency of our simulated processor.

2 Working of the processor

The simulator closely emulates the behavior of a 5-stage pipelined processor, employing several crucial mechanisms for effective operation.

2.1 Control Unit and Control Signals

A vital component of our implementation is the `ControlUnit`, an independent entity responsible for generating `ControlSignals` for each instruction. These signals, encapsulated within the `ControlSignals` object, comprise boolean arrays representing operational and miscellaneous signals necessary for instruction execution.

2.2 Processor Operation

The processor operates through the following steps:

1. **Initialization**: The main function invokes the simulator, which loads the object file into memory using the `loadProgram()` function. At this stage, the processor initializes with a flag, `isIdle`, set to true, indicating that it has yet to commence program execution.
2. **Instruction Fetch (IF) Stage**: The processor begins execution by cycling through its stages. Initially, the IF stage fetches an instruction, transitioning the processor's state from idle to active.

3. **Pipeline Execution**: Subsequently, each instruction progresses through its respective stages, facilitated by inter-stage latches such as IF-OF, OF-EX, EX-MA, and MA-RW. These latches ensure data transfer and synchronization between stages.

4. **Cycle Synchronization**: As the count of cycles precisely aligns with the number of instructions executed, synchronization between the progression of instructions and the cycle count is ensured..

5. **Completion**: Upon encountering the `end` instruction, signifying the conclusion of program execution, the processor reverts to an idle state. The simulator concludes the loop, capturing relevant statistics and writing them to a file.

2.3 Mechanisms for Hazard Handling

To mitigate potential hazards and ensure smooth execution, our processor employs several mechanisms:

- **Lock Disabling for Instruction Fetch**: By disabling instruction fetching under certain conditions, such as during branch prediction or when **RAW** hazards are detected, we prevent fetching incorrect or invalid instructions.
- **BranchTaken Addition**: BranchTaken help us accurately predict branch outcomes and adjust the instruction flow accordingly, therefore optimizing the performance of the processor. It add OX0000000 instead of `nop`.
- **RegisterLockVector**: Utilizing an integer array of size 32, our system efficiently manages register access, incrementing the corresponding array element by one upon each utilization of the register. Conversely, it decrements the array element by one once the register is no longer in use. For instance, if register 22 is accessed for the first time, the array element at index 22 is incremented by one. Subsequent accesses to register 22 result in incrementing the array element further. Upon completion of the task involving register 22, the array element is decremented by one. Upon the completion of all tasks, the array element is reset to zero, effectively preventing hazards arising from simultaneous accesses or modifications to registers.

These mechanisms, coupled with the pipelined architecture and interlocks, enhance the efficiency, reliability, and accuracy of our simulated processor, aligning it closely with real-world counterparts.

3 Testing

The simulator was tested on the following files from assignment 1:

1. `descending.asm`

   Number of instructions: 288

   Number of cycles: 288

2. `fibonacci.asm`

    Number of instructions: 114

    Number of cycles: 114

3. `even_odd.asm`

    Number of instructions: 222

    Number of cycles: 222

4. `prime.asm`

    Number of instructions: 11

    Number of cycles: 11

5. `palindrome.asm`

    Number of instructions: 31

    Number of cycles: 31