Praful Mathur
Sergey Grabkovsky
Final Project - Spring 2011 Comp Arch
Prof. Ben Cordes

Development process:

We employed pair programming through the duration of the project. Usually one of us would work on the code, while the other overlooked checking for bugs and various errors in logic from a high-level. Additionally, as one of us was writing the code, the other would look for issues and find appropriate tests to resolve perceived weaknesses.

We initially developed the parser to ensure we were correctly reading in the input files and to understand the complexity of the simulator upfront. As we became more familiar with the code and our parser was completed we started developing data representations for instructions.

We started with a base Instruction object and then the rest of our instruction types inherited from the base Instruction class. Each Instruction class has methods for each pipeline stage (F, D, X, M, W), getters for source registers & destination register, and how to deal with the results.

The first type of instruction we modeled was the I-type. It was the simplest piece and all the instructions had straight forward implementations. As we had developed a decent model for I-type instruction, we started development on the simulator. The simulator was simple as we only had to read in a destination register, a source register, and an immediate value. Then we had to figure out how to implement the corresponding instructions: addi, andi, ori, and slti.

However, the difficulty came with dealing with branches. We initially decided to stall every instruction until branch reached the Execute stage and then branch to the appropriate address. Only after we had implemented all the other instructions and had run a few tests did we decide to implement branch prediction. We opted for the simpler implementation of Branch Not Taken prediction algorithm as it was easier to test the implementation worked correctly from the logs. Also, there were given CPIs that we could benchmark our implementation against. Furthermore it turned out Branch Not Taken was an easier than trying to stall the instruction and reduced our errors.

Later we added support for R-type and J-type instructions. All the R-type counterparts of I-type instructions were just a little more difficult as it had to read an additional source register. Though it was still fairly straightforward. The J-type instruction was pretty easy to implement as we already developed jr.

Once all of the instructions were implemented as we liked then we started to deal with the data hazards. The best way we decided was to create decorators which are syntactic support for function wrappers. This way we could implement each forwarding type as a different method and then flexibly apply them to different instructions. Additionally, we implemented getters in the data model to determine whether the previous destination registers matched any of the current source registers while we implemented forwarding.

How the simulator works:

The parser reads and tokenizes the input. After the input is tokenized, we replace everything with its representation and load it into the simulator. After the instructions have been loaded into memory, the run method starts fetching each instruction and then passing each instruction into the next stage of the pipeline while ensuring the previous stage is filled.

We determine forwarding based on the following conditions:

- X to X:
    - previous (n - 1) destination register is current (n) source register
    - the register is not $r0
- M to X:
    - (n -2) prev dest register is (n) current's source register
    - register is not $r0
    - X to X has not been employed
- M to M:
    - lw is followed by sw
    - previous (n - 1) destination register is current (n) source register

If there's a data hazard we're unable to forward, we implemented stalls to resolve.

Finally, we optimize our simulator by implementing branch predictions. We implemented branch not taken predictions:

- Predict branch not taken
    - Queue up previous pipeline stages with the next instructions in queue
    - If the branch is correctly taken, then the pipeline is OK and everything keeps going.
    - If the branch is incorrectly taken, then:
        - Flush all previous instructions in the pipeline.

        - Update PC to right value.
        - Fetch new correct instructions

Data Structures:

Registers - List

Registers are just a linear data type that we need to access by numerical indices.

Single Memory - List (I$ & D$ same)

We keep all the memory in the same list with no distinction between instruction caches and data caches. The PC is started at 0x1000 and incremented after each cycle and the next instruction is fetched at the new address.

Instruction - Objects

We made instructions objects so that each instruction's functionality can be encapsulated. Also, it lets us abstract similar functionality into super-classes which allowed us to keep the code simpler and easier to test. Finally, adding features such as decorators became easier to implement as the instructions had standard methods.

Arguments - Objects

We made arguments objects so that we can have a uniform way of getting similarly formatted data (i.e. numbers) out of any type of argument. For registers, that means getting the value in the register; for immediate values, it means simply returning the value; and for offsets, it means adding the value of the offset register to the register or the immediate that's in the parentheses.

Pipelined Timing Model:

- Instruction count is incremented after each instruction gets to write.
- Cycle count is incremented if there’s an instruction at any stage in the pipeline at the end of a cycle.

Performance Factors:

- Branch prediction - Not Taken
- Fowarding (M-M; M-X; X-X)