Chris Kastelic
Zak Nilsen

UST – 3400 Pipeline Simulator

The UST-3400 pipeline simulator takes in a machine code file and simulates executing each instruction under a pipeline design. In other words, the instructions are executed (for the most part) concurrently, rather than one instruction per cycle. The result is that each instruction requires, at least, five cycles to complete; however, the throughput is increased due to the ability to run multiple instructions at once.

The pipeline has five stages (and an end stage) with five buffers. The stages are broken into (in order of execution): instruction fetch, instruction decode, execute, memory, write-back, and the end stage. The buffers are "placed" in between each stage to maintain data, and, therefore, are named: IFID, IDEX, EXMEM, MEMWB, and WBEND, respectively. It is this breaking down into "chunk," or stages, that allows a pipeline to execute multiple instructions at once. Each stage (except the end stage) requires one cycle to complete, resulting in the five cycle per instruction minimum.

First, an instruction is fetched, and the program counter is updated. After the data is updated, it is passed into the IFID buffer, which marks the end of the first cycle. Upon the start of each new cycle, a new instruction is fetched into the first stage, thus allowing multiple instructions to be executed at once. From the IFID buffer, the instruction enters the decode stage, where the instruction is broken down into its bitwise components. Also, it is in this stage where the register contents are read and stored. Then, the simulator passes the data into the appropriate buffers (i.e., instruction fetch to IFID and IFID to IDEX), and the cycle ends.

From the IDEX buffer, the instruction enters the execute stage. In this stage, the ALU is updated, and the majority of hazards are detected, including load stalls and data forwarding. A load stall occurs when the destination register of a *lw* is used in the following instruction. The data has yet to enter the stage where write-back occurs, therefore a stall must be implemented (i.e., a *noop* instruction is inserted into the pipleline after the *lw* instruction). This allows the following instruction to be forwarded the necessary data when needed. In relation, all data forwarding is detected and performed in the beginning of this stage. This occurs when register values have yet to be updated with data that is further along in the pipeline. The branch target is also updated in the execute stage, to be used later if necessary. Once all hazards have been resolved and the data properly updated, the simulator pushes all stages of data into the next buffers and ends the cycle.

From the EXMEM buffer, the instruction enters the memory stage. This stage will determine what data needs to be written back, if any. Also, control hazards (i.e., *beq* hazards) are resolved in this stage. The branch is assumed to be not-taken (i.e., no changes must be made to the pipeline). In the case of a misprediction, the instructions that are following the *beq* in the pipeline are no longer valid. In order to resolve this problem, the pipeline is "flushed" by inserting *noop* instructions following the *beq*. The program counter is then updated to the branch target (which was passed from the buffer), and the correct instruction is fetched the next cycle. Also, in this stage, memory is written to (if executing a *sw* instruction). From here, the instruction enters the MEMWB buffer, where it is considered to be retired. All other instructions have the proper data pushed to the next buffer, and the cycle ends. The last stage the instruction goes through is the write-back stage. Here, data is written back to the proper register (*add, nand, lw*). From here, all buffers are updated and the cycle ends.

During runtime, the simulator will print out the state of the pipeline. This includes the contents of the data memory and the registers. It also includes the instruction held at each stage of the buffer at the end of the cycle as well as the values held within each buffer. Upon program completion, it will print a final set of statistics that include: total cycles, total number of instructions fetched, total number of instructions retired, total number of *beq*'s executed, and the total number of mispredictions (i.e., branches taken).

A general shortcoming to this program is that *jalr* instructions were not implemented for simplicity. This type of instruction would result in a number of additional hazards and complications. In addition, it is unclear whether the *sw* implementation is completely accurate. It is believed to be functioning properly, though little time was spent in debugging this instruction. Also, some minor improvements could be made to code structure for clarity and simplicity.

There were many difficulties encountered during this project. To start, the team got off to a slow start due to unforeseen circumstances. This put the project behind schedule from the start. In addition, the concepts in this project were difficult to translate into code. In particular, the data forwarding was an elusive concept. This obstacle was overcome by receiving help from the professor and by drawing out diagrams of the pipeline. Another sticking point was incorporating multiple hazards into the code. It was difficult to understand exactly when certain actions should take place within the code. Again, this obstacle was overcome by receiving help and drawing diagrams. In addition to drawing diagrams, extensive testing, while comparing the project simulator's output to the reference simulator's output, was extremely useful for debugging specific occurrences of hazards. Apart from these main points, there were several smaller sticking points that were overcome by sheer perseverance and effort.