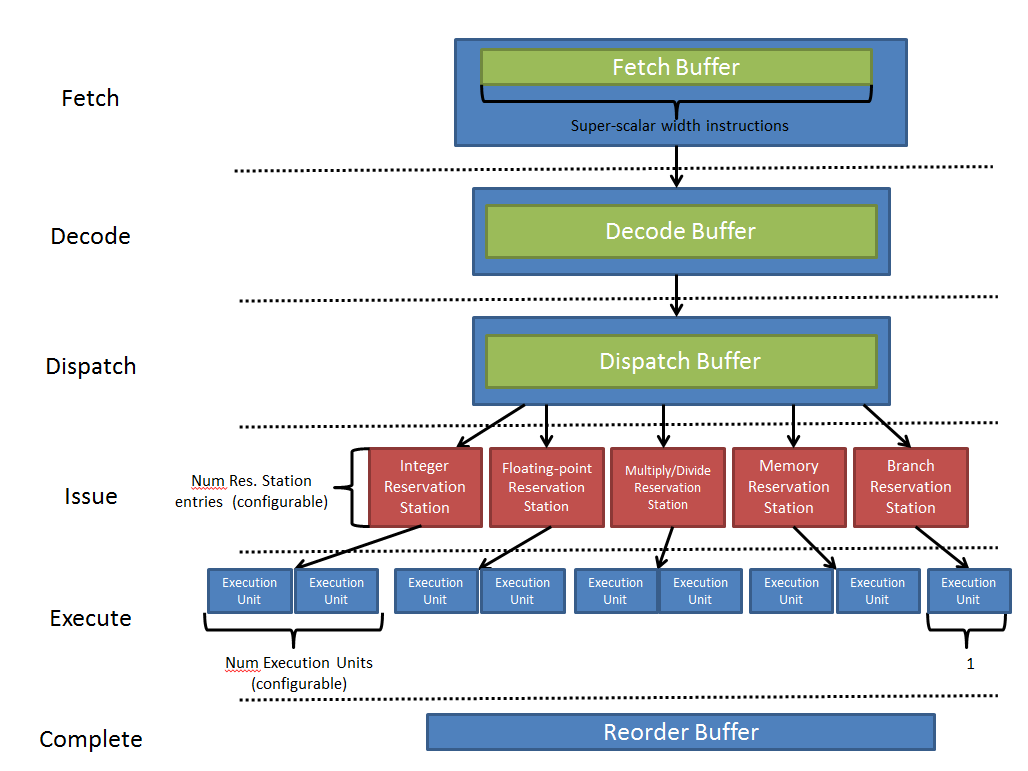
Analysis and Implementation of a Superscalar CPU Simulator

Benjamin Fortener and Andrew Kordik  
ECE 636 – Advanced Computer Architecture  
Dr. Tarek Taha  
Fall 2011

**1. General CPU Architecture**

Our CPU simulator implements a superscalar processor with six pipeline stages: the fetch stage, the decode stage, the dispatch stage, the issue stage, the execute stage, and the complete stage. It uses out of order execution in the issue and execute stages in order to extract parallelism from the code where possible. This out of order execution is achieved via execution-type specific reservation station buffers in the issue stage and multiple execution units for each type of instruction execution type. The instruction execution types are: Integer, Floating Point, Logical (although Logical instructions are treated as Integer), Memory, Multiply/Divide, Branch, and Nop (no-operation). The number of execution units is configurable except for the branch execution unit, of which there is only one. The high level CPU contains an architecture register file, a rename register file, a reorder buffer, the program counter, and a branch predictor along with the pipeline stages. A block overview of the CPU’s pipeline stage design is shown below in Figure 1. The following sections will cover each of the pipeline stages and the functional blocks within them in more detail.

  
Figure 1 – Overall block diagram of the CPU simulator’s pipeline stages.

**2. The Fetch Stage**

The fetch stage is probably the simplest of all the stages; its job is to read [*superscalar width*] instructions in from a trace file and put them into a buffer. Because the fetch stage is implemented in a CPU in-order, we used C#’s Queue class. Although we are reading from a trace file in our simulator, a real CPU is reading from its instruction cache and may run into a situation where the instruction fetched is not in the cache. In order to simulate this cache miss, we were given 1% as the percentage of time in which a fetch typically misses cache. If the fetch misses level 1 instruction cache, the penalty is 5 cycles, during which it is fetching from level 2 cache. It could also miss level 2 cache (typically 20% of the time), and the penalty for a level 2 cache miss is 200 cycles.

**A. Instructions**

Instructions in a CPU contain more information than the instructions themselves. Our instruction class holds the execution type of the instruction, the address (given by the trace file), the instruction’s sources and whether or not the sources are immediate, its destination, and whether or not it’s the last instruction in the trace. This allows us to easily pass the all the information needed about the instruction to all the different buffers it needs to go.

**B. Reading Instruction Sources and Destination Registers**

Although this portion most likely occurs in the decode stage in a real processor, we implemented it when we read in an instruction. Based on the instruction type and the way in which the assembly command is written, the source data and destination register may be located in different places on the trace file line. Most instructions typically use one way while some exceptions must be taken into account to assign the correct sources and destinations.

**C. Cache Misses**

In order to simulate a cache miss, we need to know what occurs in the real CPU when it misses cache. If a CPU tries to fetch an instruction from its instruction cache, and that instruction is not in the cache, it must pause all fetching until it brings in the instruction from a higher memory (either level 2 cache or RAM). In our case, it cannot find the instruction in level 1 cache 1% of the time and it cannot find the instruction in level 2 cache 20% of the time. It must pause 5 cycles if it found the instruction in level 2 cache and 200 cycles if it also missed level 2 and found the instruction in main memory. In order to simulate a cache miss, a random number is generated between 0 and 99, inclusive, and if that number is equal to 0 (as it will be 1% of the time), we generate another random number between 0 and 99. If that number is not less than 20, we only missed level 1 cache, and fetch is paused for 5 cycles. If it is less than 20, then we also missed level 2 cache, so fetch is paused for 200 cycles. After the pause amount, fetch reads in the amount of instructions defined by superscalar width and continues as before.

**3. The Decode Stage**

The decode stage contains the decode buffer and the branch predictor. It reads instructions from the fetch buffer until either it is full or the fetch buffer is empty. If an instruction is literally a branch instruction (not a branch execution type, which may include jump instructions), its address must be checked against the next instruction that was fetched to see if the branch was actually taken. We use our branch predictor module to predict whether or not the branch is taken. If the branch prediction is different than what actually occurred, the fetch stage and decode stage are paused until the branch instruction is finished executing, when the branch would be resolved and any invalid instructions would be flushed out. However, if the prediction was correct, the fetch and decode stages continue as normal.

1. **G-Share Branch Predictor**

The G-Share Branch Predictor consists of 512 independent 2-bit predictor state machines and a branch history shift register. The selection of a state machine is determined by the instruction address hashed with the branch history shift register. Each 2-bit state machine always starts in an initial state where the branches are predicted as not taken, it can then be updated based on the result of the branch instruction. The result of the branch instruction can be immediately calculated based on the address of the next instruction in the trace file.

**4. The Dispatch Stage**

The dispatch stage must check to see if the system is ready for an instruction to be dispatched before it can read from the decode buffer. Because when an instruction is dispatched, it’s dispatched to three places, all three of the places must have room for the instruction before dispatching. Once it has verified the system is ready for it, the dispatch stage reads an instruction from the decode buffer and dispatches it to the reorder buffer, a reservation station, and the rename register file.

**A. The Rename Register File**

The rename register file is an intermediate buffer for the results of instructions to be held before updating the architecture register file with the new results. When an instruction is placed in the rename register file, the architecture register file’s tag is updated to point to the RRF entry.

**B. The Reservation Station**

The reservation station is a buffer that holds multiple instructions of a single execution type. Upon dispatching to the reservation station, a check of the instruction’s sources are made to see if any of the sources need to be updated by another executing instruction before it can be executed. If all the sources are valid, the instruction is set to ‘ready’ in the reservation station. Otherwise, the sources are set to the tags for the rename register file entry of the registers they’re waiting on being updated.

**5. The Issue Stage**

The issue stage contains reservation stations for each execution type. On each cycle, the issue stage checks to see if one of the instructions within it is ready to be executed. If one is ready, it then checks to see if there is a functional unit available to execute it. If so, it sends the entry to the functional unit to be executed.

**6. The Execute Stage**

The execute stage has functional units for each execution type (configurable amount for all types except branches, of which there is one). During this stage, any execution unit that has an instruction checks to see how many clocks are left before the instruction is finished executing. If it is finished executing, it signals the reorder buffer that this instruction is finished and tells any instruction in the reservation stations that may be depending upon the resulting data that it is valid.

**7. The Complete Stage**

On each cycle, the complete stage checks the reorder buffer, starting from the front of the queue, to see how many instructions have finished executing. It can only complete [*superscalar width*] number of instructions per cycle. Because instructions must be completed in order, it can only complete instructions until it encounters an instruction that is not finished.

**8. Application Specific CPU Optimization**

The simulator is designed to support the configuration of the following parameters: Super-scalar factor, number of reservation station entries, number of functional units and number of reorder buffer entries. As an initial starting point, each was assigned a value of 20. This allowed for a large range to sweep through parameters and see which combination was the optimal. Each value was swept independently. The “mpeg2d” program was selected to use as the basis for the CPU optimization because the CPU parameters have a more pronounced effect on its behavior.

The two parameters which had the most effect on the instructions per cycle were the number of renaming table entries and the number of reorder buffer entries. Figures 2 and 3 below show the results of sweeping these parameters respectively. The large dips in the plots are due to the randomness associated with cache misses; however, the trend is apparent in each plot. From these plots we selected the optimal value for the number of renaming table entries as 17 and the optimal number of reorder buffer entries as 25.

Figure 2: Renaming Table settings sweep.

Figure 3: Reorder Buffer settings sweep.

The next parameter analyzed was the number of reservation station entries. As this parameter does not affect the instructions per cycle, it is optimized based off of how often the reservation stations are full. Figure 4 shows a comparison of the number of times the reservation stations are too full to issue and instruction with respect to the total number of reservation stations in the system. The optmimal value of 18 reservation station entires is selected to provide some flexibility under times of high cache misses.

Figure 4: Reservation Station settings sweep.

The super scalar size and number of functional units did not show significant change in any statistics. This is likely because the traces were short and because not all possible combinations and permutations of parameters were selected to demonstrate an effect. For example, if there are only 3 execution units, the capability to capture 20 instructions at a time is overshadowed. The inverse is also true. Therefor the super scalar size and number of functional units were set to be equal at 10 so that we can limit power consumption and reduce unused resources.

Using these optimal values as the starting point, the parameters are swept again to find a final set of optimal values. It was found that the optimal settings were: super scalar factor = 10, number of reservation stations = 18, number of rename register file entries = 28, number of reorder buffer entries =25. These settings yield 1.6 instructions per cycle with zero occurances of the register rename file full and 26 occurances of the reservation stations being full.

**9. Conclusions**

Our CPU simulator successfully ran through all the of the traces given over the multitude of parameter changes. It was very interesting to see the different results of each statistic for the different programs. From our results, we’ve shown that it is a working simulator and it can process more than one instruction per clock, which is what a superscalar processor allows.

After running all the files through the CPU simulator, it was surprising to see what little effect increasing the parameters had on the instructions per cycle. After some analysis, however, it made sense. When increasing one or two parameters, the others quickly became the bottleneck. For example, if the superscalar size was increased to a very large amount (eg, 40), many instructions were fetched in a single clock, but they quickly hit a wall when reservation stations became full or execution units all became busy. The process of optimizing the small amount of parameters we had was daunting; it is truly a daunting task to optimize a real processor for use on average applications across the board.

Given more time, this could easily be taken to a GUI, given that it is already implemented in C#. Some more granular timing analysis would serve it well; perhaps some of the more time-intensive portions of it could be implemented in C or C++ and called from a DLL. This would allow for the fast execution while still allowing for the nice structure and GUI features of C#. More in-depth and better statistical analysis could be added to make it a truly powerful tool for a CPU of this architecture.