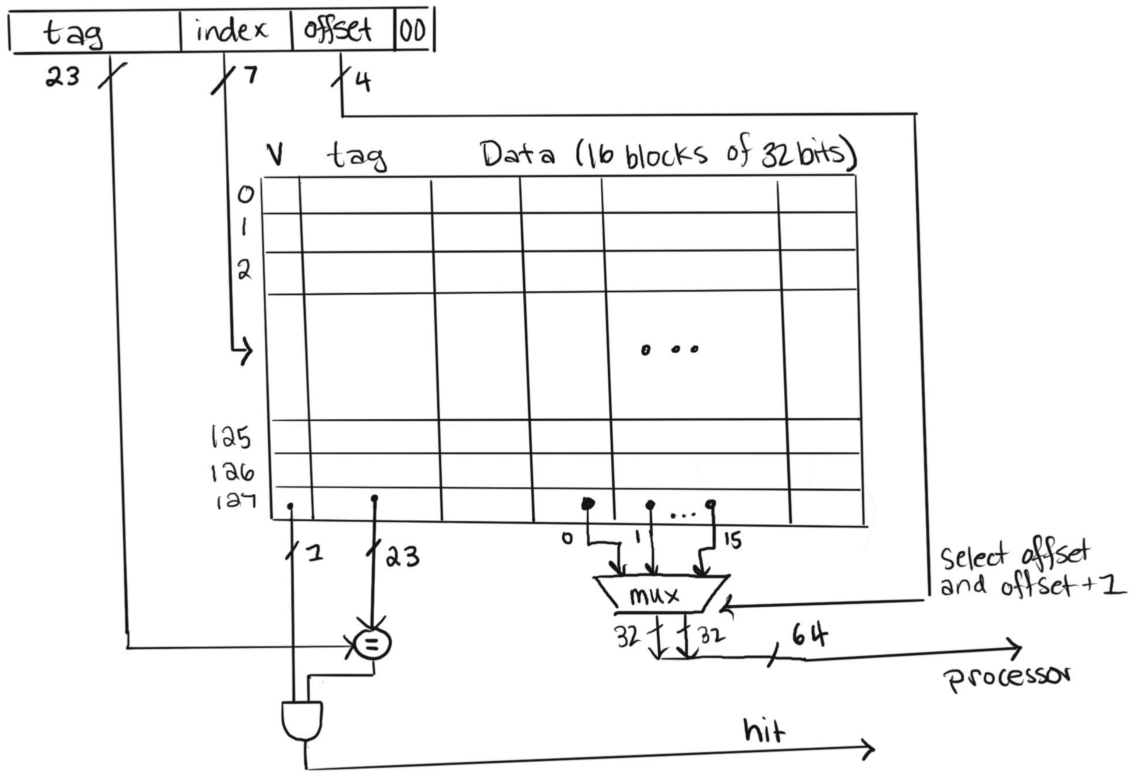
Final Report 6.1920

Alyssa Keirn and Keilee Northcutt
akeirn@mit.edu & keilee@mit.edu

In this project, we implemented a modified cache and a superscalar processor, and attempted to run it on an FPGA. To achieve this, we changed the cache return type, storage method, and addressing. We modified the connection between the cache and processor, changed the test cases to work with a new cache, redid the processor to execute two instructions at once, and modified the Konata visualization.

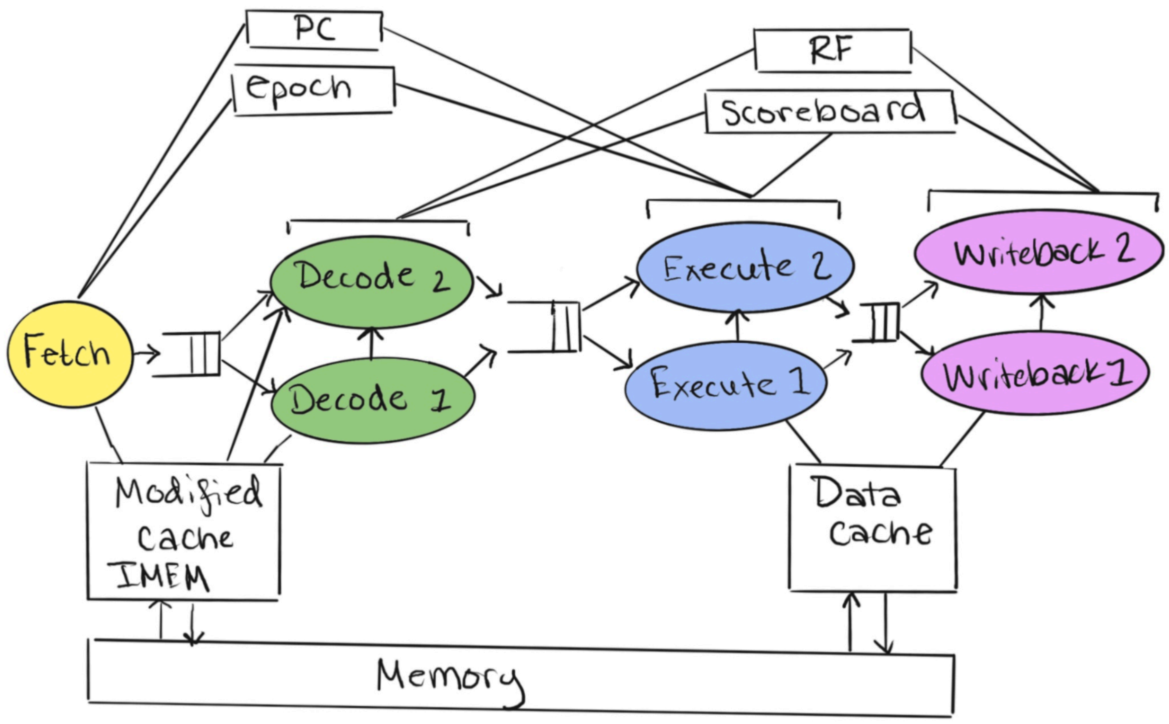
To create the cache, we used the cache from lab 5 as a starting point. This cache operated as a direct mapped cache, with 128, 512 bit lines. Each line was treated as a single entry. For our cache, we kept the 128, 512 bit lines, but split them up into 32 entries. Each entry could be addressed using the 4 offset bits of the address. See below for a diagram of the new cache.



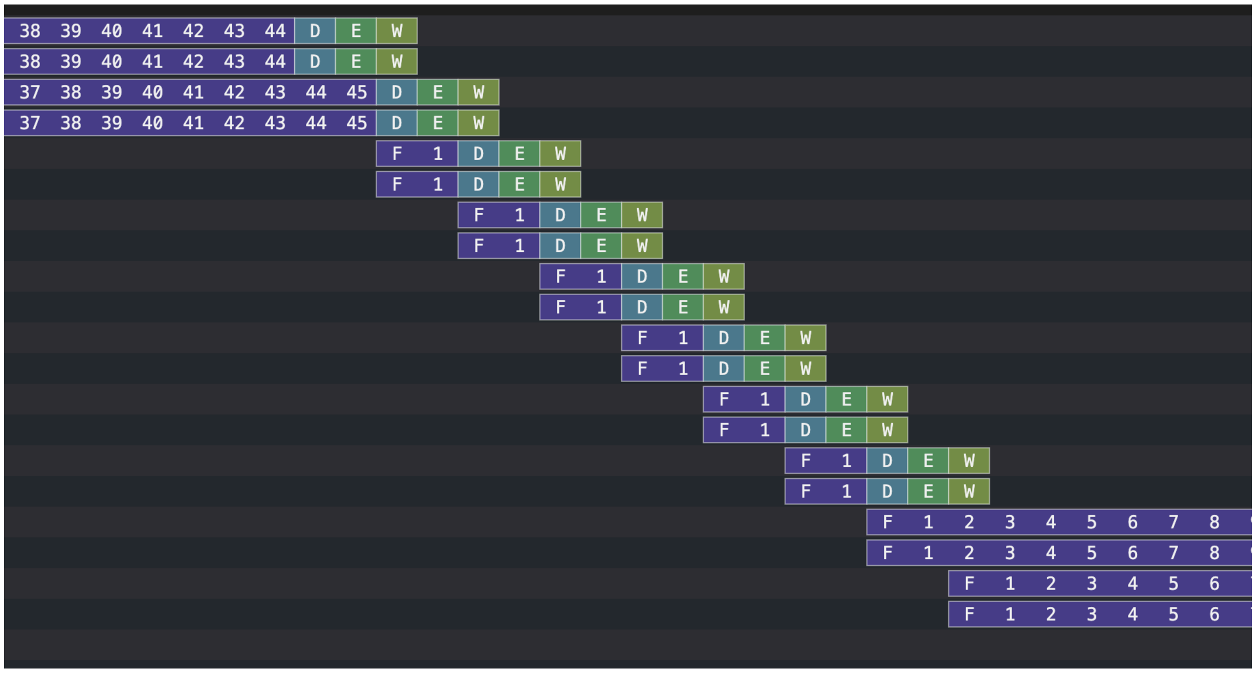
This involved changing the addressing in both the cache and the test bench. To allow the processor to do superscalar, we next changed the return type of the cache to include two entries and a valid bit. The first entry is the instruction at the requested pc. The second entry is the instruction at pc + 4, which in most cases is directly adjacent in the cache. The valid bit is used to signify whether the second entry in the return type is usable. This accounts for edges in the cache. For example, if the processor were to attempt to fetch an instruction at offset 15, the value at pc + 4 would fall on the next cache line. To simplify the cache, we only dealt with the case where both instructions fall on the same line, and returned an invalid bit if only one entry was accessible. To test this cache, we modified the Beveren test suite given for lab 5. To do this, we changed the return types for the cache and the reference. In the process, we changed the reference to work with a two port BRAM in order to allow both returned instructions to be checked at the same time. This had the added benefit of limiting the changes of the Beveren file to just adding a new statement to check if the second instruction was valid. We then checked our cache against the expected behavior of the reference. We ended up using this cache for both the instruction cache and data cache. If we were to improve on this project, we could instead create two unique caches, one with superscalar functionality and no store buffer, and one with a store buffer but no superscalar. This would be more efficient because the superscalar functionality is not very useful for the data cache, while the instruction cache never stores and thus has no use for a store buffer.

Once we modified the cache, we connected the cache to an unmodified processor from lab 4. We started by adding additional rules to the top level coding infrastructure to handle connecting to a cache rather than a BRAM instance. In order to connect the processor to the modified cache as opposed to the Main Memory structures originally provided in Lab 4, we created a cache to store instruction memory (“iCache”) and a cache to store data memory (“dCache”). Within the top level of the processor, these two caches are then connected to the Main Memory. Additionally, we implemented a FIFO in order to keep track of which of the two caches a Main Memory response belongs to. We then tested the connection by running the unmodified test bench from lab 4 on a known working processor, modified to drop the second instruction in the cache return type.

Finally, we implemented superscalar in the processor. To do this, we started by swapping all the FIFOs between stages for superscalar FIFOs, courtesy of Thomas Bourgeat. These FIFOs allow multiple things to be enqueued and dequeued between stages in the same cycle and are critical to allowing superscalar to function. An order must be kept with the enqueue and dequeue to make sure the instructions execute in order, but they must also be entered concurrently. We then duplicated the Decode, Execute, and Writeback stages. This simplified the implementation and was needed for concurrent execution. It allows the ALU and other logic to be used for two instructions at once. To prevent duplication of memory, we prevented any memory instructions from executing in the second pipeline. This causes some stalling, but is preferable to incurring the extreme cost of duplicate memories. To preserve order, we also modified the guards of each rule in order to ensure that the second pipeline only executes if the first has also executed during the current cycle. See below for a top level diagram.



We debugged and verified this processor by running it against the test suite provided for lab 4. We then modified the Konata rules to allow for multiple calls and checked the concurrency by displaying the modified processor’s output on Konata (see below).



Our processor successfully completes two instructions at a time and arrives at the correct end state. As seen in the figure, the time the cache takes to load a line is reflected in the occasionally long fetch time. We did however have a double-cycle hit fetch bug, where the fetch takes two cycles to complete.

We attempted to run our design on an FPGA via the virtual machine provided by the 6.1920 staff; we created two different cache files for iCache and dCache for the machine to build along with our pipelined processor for real hardware. However, we experienced issues with building on the virtual machine and did not have enough time to resolve them.

If we were to continue this project, we would modify the cache as mentioned above, and fix the double fetch bug. We would also like to try simple branch prediction with this processor if possible.