Write-Up

Anjin

Dhruv

Hao- Worked on branch prediction and implementing the push\_pipeline\_stage function

Jay- Worked on cache data structure and associated methods, some editing

In our project we implemented several methods which allowed our pipeline to run accurately. First we ask the user to enter in a cache index, block size and associativity. We then make sure that we don’t not exceed the cache size of 10240. Next we dynamically create the cache and initialize the pipeline. We then go through and parse the information in our trace file.

Every time we parse one line, we check to see if the address is already in our cache, using iplc\_sim\_trap\_address. If it is in our cache, then we call the iplc\_sim\_LRU\_update\_on\_hit method which will simply add to our counter for the amount of cache hits and update the value in our cache. However, if the address is not in our cache then we call the iplc\_sim\_LRU\_replace\_on\_miss which goes through and looks to find either an empty spot in the cache or looks to find the lowest value in the cache and replace the old data with the new address’s data. We then also update the amount of cache misses. If we do have a miss, in our iplc\_sim\_parse\_instruction method we push the instruction through the pipeline. The purpose of doing this is that we do not double count cycles but instead have them overlap.

In the pipeline simulator, iplc\_sim\_push\_pipeline\_stage, we go through seven stages. The first stage is the writeback stage in we write our results into the register file. The second stage we check to see if the branch prediction was right. The third stage is to check for any LW delays. If there are any then we add 9 cycles to our cycle count. We are adding one additional cycle later one so that is why we do not add the full ten cycle delay. In the fourth stage we look for a SW memory access and data miss and add 9 cycles if it does happen. In the fifth date we just implement the total cycles by one. In the sixth stage we push through onto the next stage of the pipeline. In the seventh stage we reset the fetch stage to NOP.

We then go back into the iplc\_sim\_parse\_instruction and parse the instruction. We check to see if then instruction if “add”, “sll” or “ori” as if it is then we assign values to our dest\_reg, src\_reg and src\_reg. We then call the iplc\_sim\_process\_pipeline\_rtype method which basically assigns the dest\_reg, src\_reg and src\_reg to the Fetch in the pipeline. Next, we check to see if the instruction is “lui”, if it is we then call the iplc\_sim\_process\_pipeline\_rtype method again but this time we would assign the same value to dest\_reg as the last conditional but assign the values -1 to both src \_reg and src\_reg2. We then check to see if the instruction is “lw” or “sw”, if it is “lw” we then set the dest\_reg value and call the iplc\_sim\_process\_pipeline\_lw method which essentially sets the values of the dest\_reg, base\_reg and data\_address for the lw in the pipeline. If the instruction was “sw” however, then we assign the proper value to the src\_reg and call, iplc\_sim\_process\_pipeline\_sw. If the instruction is “beq” then we call iplc\_sim\_process\_pipeline\_branch. If the instruction was “jal”, “jr” or “j” then we call iplc\_sim\_process\_pipeline\_jump. If the instruction is “syscall” we call iplc\_sim\_process\_pipeline\_syscall. If the instruction is “nop”, then we call iplc\_sim\_process\_pipeline\_nop. All of these methods just assign values to the targeted pipeline. Lastly, we call iplc\_sim\_finalize which will ensure all instructions are done procceded and print important statistics.