* Victim cache:
  + Victim cache will be a fully associative cache that contains 16 blocks, whenever there is dirty eviction, the dirty data will be stored into the victim cache so that cache can immediately fetch missed data from the main memory without writeback. The clean data being evicted will be also temporarily stored in the victim cache. Whenever cache reads data from main memory, it will first check if the data is already in the victim cache, if it’s, then the cache can immediately take it. If the victim cache is full when it needs to take in a block, it will write back a lru block to make space.
  + The lowest level cache will check if the data to read is in the victim cache, if it is then take it, else go to the main memory as usual. If there is an eviction in the lowest cache, it will not write the data back to the main memory, instead, it will try to insert it into the victim cache. So there are basically two paths from the cache to memory with one of the paths containing the victim cache.
* L2 cache:
  + L2 cache will just be another cache after the cache in the first level. When missing, the first level cache will read from the second level cache. The first-level cache will also write back the data to the second-level cache. If there is still a miss in the second level cache, the second level will read from the main memory then bring the data to the first data cache. The logic of writing back data of second-level cache is the same as L1 cache.
* Pipelined cache: first instantiate BRAM Arrays:
  + Mp3 Appendix B for data, tag, valid, dirty, IRU.
  + Reusing the Cache from MP3, but divide to two stages.
    - Pipelined Icache: one stage to determine hit/miss, one stage for returning data.
      * Reroute datapath to prevent stalling: current datapath get data in one cycle so instruction data can be gathered before stage\_if\_id, we reroute it so that instruction data will be gathered at decode stage instead to prevent stalling.
    - Pipelined dcache: one stage to determine hit/miss, one stage for returning data.
      * Right now the dcache\_read signal is outputed right after stage\_ex\_mem, we have to get it directly from control\_word before stage\_ex\_mem. We have to create additional stalling logic to compensate for dcache read being directly from control\_word.
      * Two-cycle later we successfully retrieve data or stage\_mem\_wb.
* Four-way associative cache:
  + Each set in the cache stores four lines of data. Data is 32 bytes. If we keep using 8 as the number of sets we will have a cache size of 1kb.
  + Pseudo LRU: three bits, 4 valid arrays, 4 dirty arrays, 4 cache data arrays and 4 tag arrays. keep the tag bit, index, and offset the same.
* Tournament Branch Predictor:
  + The branch prediction mechanism for our design will consist of a two layer predictor table. The first layer will contain a local and a global branch prediction. The local predictor will function as a basic branch translation buffer, using the PC of the branch instruction as a hash into the table, which contains the 2-bit branch prediction and the PC target address. The other branch predictor in the first layer of the tournament will be a global branch predictor. The global predictor functions very similarly to the local branch predictor; however, it will account for the last N branches and whether they were taken. Using an N-bit array which will be bit shifted left for every branch prediction. If the branch is taken, the rightmost bit will be set to 1. By concatenating this N-bit array with the PC address of the branch instruction, we can create a hash into the global branch history table, and use the same 2-bit predictor and target mechanism as the local branch history table.
  + The second layer of the branch predictor will choose whether to take the prediction from the local or the global predictor. This will use a simple 2-bit mechanism, starting with 00 to represent “strongly local.” If the local predictor is incorrect, each history table will be updated and the 2nd level predictor will move to 01 meaning “weakly local.” As the program continues, it is expected that the predictor will move towards the global predictor as the global predictor gains more information about the program.
  + With every prediction, the PC will be updated with the predicted address of the branch. To do this, the branch predictor will pass a flag through the pipeline to indicate the result of the prediction. If the branch is predicted to be taken, the PC will move to the target address and proceed until this jump is confirmed in the execute phase. If the prediction is correct, the pipeline will continue as normal and update both branch history tables to reflect the result; however, if this prediction is incorrect, the pipeline will be flushed and the correct PC address will be loaded, updating the branch prediction mechanism to reflect the missed prediction.