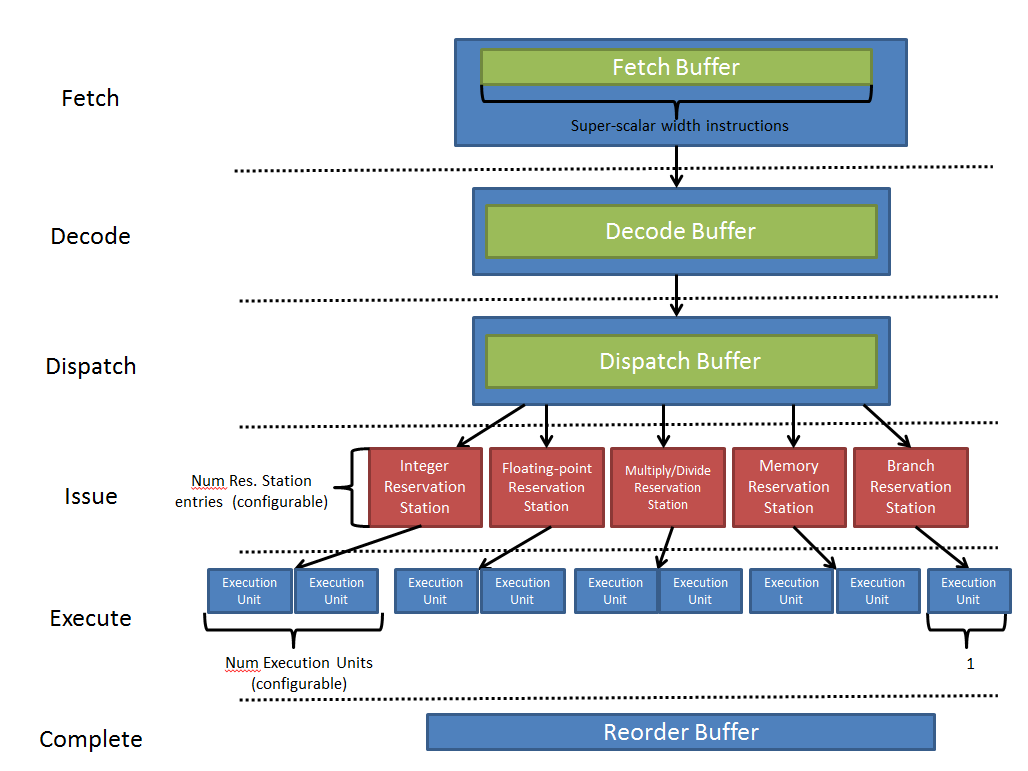
**1. General CPU Architecture**

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’s address 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.

**A. G-Share Branch Predictor**

**4. The Dispatch Stage**