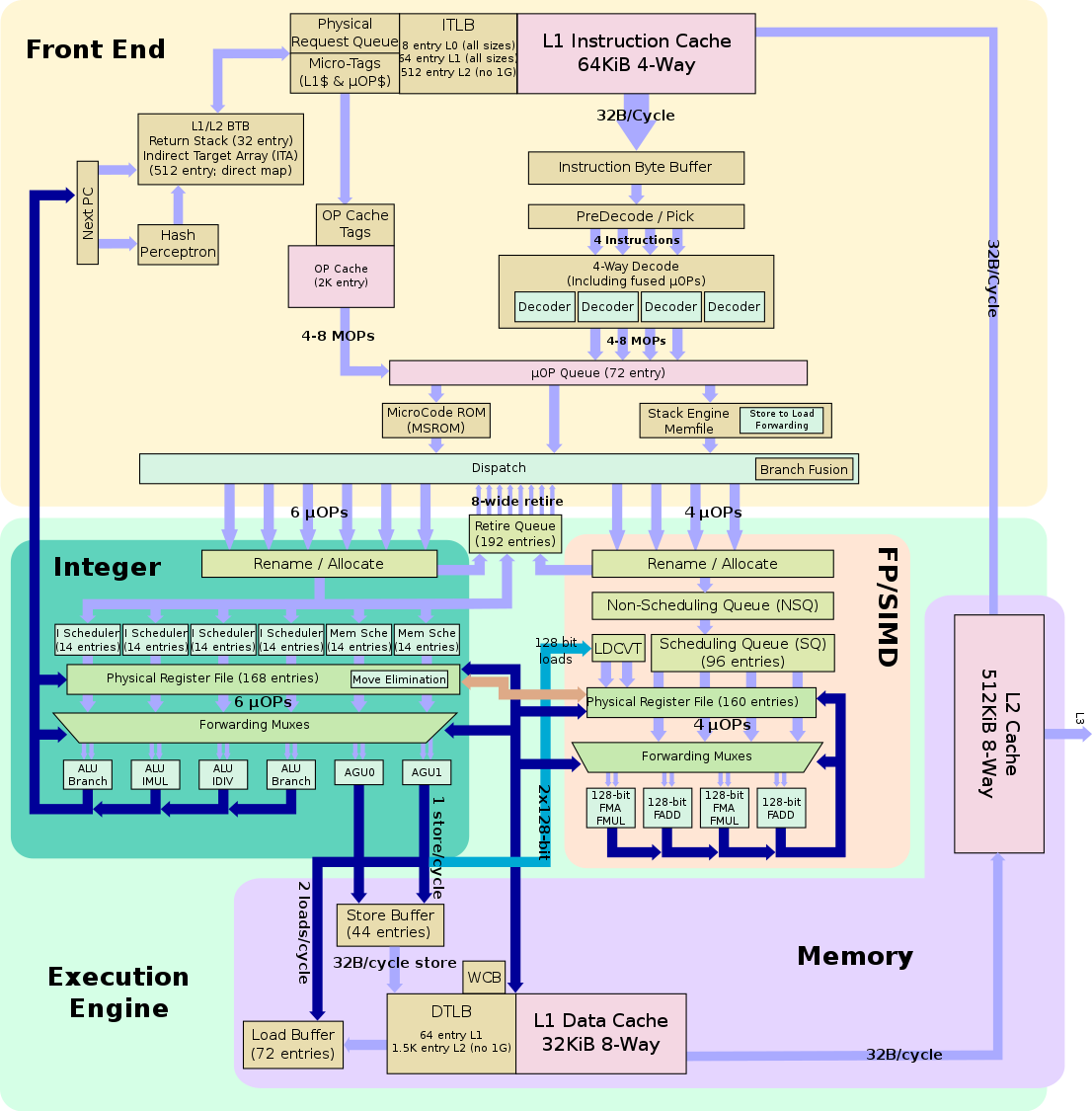
Closest to actual Article I can find(maybe someone can get it out of cate?): <https://web.archive.org/web/20181102185520/en.wikichip.org/wiki/amd/microarchitectures/zen>

* 1. Question refers to this: 
     1. Tracks return addresses on stack for branch target prediction of branch instructions that return (function calls)
     2. Based on the diagram for deciding how to dispatch uops. My prior knowledge also tells me that the microcode is used during the generation of uops, to allow for changing how instructions are executed after the fact.
     3. When addresses(relative to a register or absolute), are the same for both the load and store instructions. And the load occurs recently enough after the store, for the data to still be kept.
     4. In the Execution engine, when the store buffer/load buffer have pending stores/loads which can be forwarded.
     5. Move elimination allows for the elimination of instructions that move values from one register to another by simply renaming registers instead of actually moving the value in question. It is done in the physical register file b/c that would be the most natural place to do it since the mapping for registers are presumably stored there or nearby.
     6. Non scheduling queue is just a buffer, and the instructions within it cannot be scheduled i.e. put through to the execution units. They must first go through the scheduling queue.
     7. Since it allows FP instructions which are dependent on other instructions (esp. slow ones like loads), to sit in the NSQ while waiting and not pollute the SQ?

SQ more expensive hardware?

* + 1. Lots of FP code, dependent on memory (E.g. calculations, graphics). So we can initiate all memory accesses as soon as possible. To keep FP units in constant use.
  1. *“Decoding is the biggest weakness of* [*x86*](https://en.wikichip.org/wiki/x86)*, with decoders being one of the most expensive and complicated aspect of the entire microarchitecture”*

A simplified instruction set makes decode way simpler, so less expensive. Can maybe remove uOP cache (or atleast simplify). BP can be simpler (since currently does uOP prediction). Can maybe remove whole MOP idea.

Every instruction 32 bits => simplified fetching logic. No communication from decode to fetch for alignment.

* 1. Just from L1D Cache, normal load as expected.

Store to Load Forwarding can provide value, if a store occurred recently.

Not sure about a third?

Request from other cores by broadcasting the request

* + 1. Log2 cachelinesize = log2 64 = 6

Log2 totalsize/(cachelinesize \* associativity) = log2 64\*1024/(4\*64) = 8

First (from right) 6 are block offset, next 8 are set index. So: addres s[50:58]

* + 1. The bits for the page number. Not sure if a number is required? Dunno if so.b
    2. Two processes map a physical address to separate virtual addresses. E.g. COW’d files untouched. (synonyms)
    3. Probably since that’s what all decent CPUs do. Can’t find any evidence to support this tho.

PIPT is slow and outdated. VIVT is fast but we have issues with homonyms and synonyms that can’t be easily solved. PIVT is useless (one can parallelise the cache access and virtual to physical translation, if we have a hit we can return the data, if we have a miss, we have the physical address available to request in l2 cache /main memory)

* 1. See what L1 cache lines are taken (by aliasing). Therefore locality/size of secret key. Or locations of what tools might be used for calculation.

Same thing but for L2 - if that counts as another, and assuming ISA provides control where to put memory.

See what execution units are in use. E.g. if integer units in use, likely mathematical hash in use, instead of simple cipher.

See what uOPs have been used, with OP Cache. So identify what operations the algorithm might use. Particularly Zen has SHA in the instructions, so for sure check if this is used.

Use Store-to-load forwarding. Although it’s not clear how it works (so might not work), could time loads on other thread, and see if they’re fast i.e. mem in use.

* 1. When you are sure the memory won’t be used again for a long time, and don’t want to pollute the cache. Especially useful for vector store, as that’s 256 bits saved.
  2. To ensure the memory model consistency is ensured. As if we try to use the register when only one half has been retired, it would have a totally incorrect value.
     1. Potentially don’t have to execute the instruction at all. This could only be known after register read though, since we need to know the predicate register contents. So only execute units can be saved.
     2. In the execute stage, while reading register file.
     3. We would have already added the instruction to the retire queue, at dispatch. Although we could then potentially take it off. This depends on how the retire queue is implemented though (e.g. in BOOM 19-20 article, this would not be possible).
     4. On register read, just ignore that uOP entirely and don’t schedule it through to the execute units.
  3. Have to add ‘predicate masks’ to retire queue, adds extra hardware.

Vector instructions have high latency, so often will still be executing when predicate register known. Alternative: complex kill execution logic.

Often worth the costs, since vector instructions have such high latency, worth scheduling them as soon as possible.

Extra energy costs.

4.

* 1. Bimodal uses an n-bit saturating counter

2lev is a 2 level adaptive branch predictor which uses the history to index a table of 2-bit saturating counters

Combined uses a 2lev predictor with a table of n-bit saturating counters? <- Not sure

* 1. Return address stack keeps track of function calls and their returns. Predicts where a function will return to
  2. Branch target buffer predicts the target of a branch rather than if a branch is taken/not taken
  3. Updated when the front end is redirected to take a branch or jump. Crucially, at commit.

i. Need 9 bits for the set. Don’t need any for the block. So 32-9 = 23. So first 23 bits are for the tag

ii. Partial tags could mean multiple addresses could map to the same place in the BTB which would give the wrong prediction for some. Especially in non spatially local code e.g. calling kernel code frequently.

* 1. Number of cores
     1. Some programs might run more efficiently on multiple cores, and some might not have much possible threading
  2. L2 cache shared between cores?
     1. Is this possible?
  3. L3 cache enabled/shared?
     1. Energy efficiency depends on how much of the L3 cache is required and whether accessed memory overlaps
  4. Cache coherency mechanism, snooping or directory?
     1. Snooping may be faster for lower amount of cores, but costs more energy as more cores are used?
     2. Same thing for directory based coherency?
  5. Ability to turn off cache coherence (eg Weak/relaxed memory model)
     1. For programs that do not have multi-threaded race conditions, more energy efficient.
  6. Dynamic clock rate enabled?
     1. Some programs may be more energy efficient running at max clock rate for shorter period of time, compared to longer periods at lower clock rate/energy usage
  7. SIMT enabled, number of threads?
     1. SIMT means that the cpu pipeline can be efficiently utilised if necessary, but can occur energy overhead.