Latency and throughout ~ Calculating for SEQ v Pipe

* Latency
  + Time from start to finish for some operation
* Throughput
  + Rate at which operations complete usually operations/time
  + Can also write it as time/operation completed (still throughput)
  + 1 operation/time unit
    - 1 time unit/ operation completed
* Latency of 5 time units/operation
* Single-cycle processor: we do one instruction at a time, so time per instruction = time between when instructions complete
* Time per circuit = critical path: longest sequence of operations that needs to complete in a clock cycle
  + Single cycle: operations from getting the new PC (beginning of cycle) all the way to writing all the instructions’ values
  + Pipelined processor: operations for pipeline stage typically “between two pipline registers” (except: ends)
* Set cycle time = critical path length
* Subtly with critical paths are that not everything is on the critical path:
  + Single-cycle: operations which can be done in parallel
    - For example: adding X to the PC while reading registers
  + Pipelined: only the longest stage matters ( and only longest path through a stage)
* Calculating time: sum of things on critical path
  + But remember that pipeline registers take time (if there are pipeline registers)

RET and Load/Use Hazards

* IN the five-stage processor we used
* RET F DE M F
  + We need to forward the thing from memory to the fetch step
  + 3 cycles of stalling
* Load/Use hazard
* MRMOVQ -> R9 FDEMW
* ADDQ R9, R10
  + We need r9 from previous memory stage
  + Cycle of waiting even with forwarding
    - Anytime we read a value from memory and want to use it in execute probably forwarding from end of memory to end of decode
* No hazard
* ADDQ R8, R9 FDEMW
* SUBQ R8,R10
  + End of decode execute to end of decode since the value will be executed to in the E step

S2018 Q5 !! hazards in a four-stage pipeline

S2018 Q13 !! address mapping to same set

* 4 way, 18KB cache, 16 Blocks
* Offset.= location w/in block , 16 locations 🡪 4 bits (2^4 = 16)
* [set] index = number of the set, ??? sets
  + 16KB chache/16B blocks 🡪 1 blocks
  + 4 Blocks/set
  + 1K blocks/(4 blocks/set) = (1k/4) set = 256 sets
* Tag = everything else (in this case, 16 bit addresses – 8 – 4 = 4 bits)
* Same set as 0x1234
  + Tag index offset
  + 0x1 0x23 0x4
  + Determines which set to use
  + Same set
    - 0x1235
    - 0xF23F
  + Not same set:
    - 0x1A34
    - 0x1004

Direct-mapped v set-associative

* Direct-mapped ==== 1-way set associative
* N- way set associative means N places to put each block means can store N different with the same index bits
* Fully-associative ==== # blocks in cache – way set associative

Write strategies (write-alloc/no-alloc, etc.)

* Write-allocate: when the program writes, do I bring something into the chache because of it
  + Pro
    - Hopefully the program will read that value/near it
  + Con
    - Writes now become reads (rest of block) + writes
* Wirte-no-allocate:
  + When the program writes, modify value if I have it, otherwise send it to memory w/o any extra work
* Write-back:
  + When the pogram writes, don’t send it to memory until I need to
* Write-through
  + When the program writes, always send it to memory immediately.

High-level strategy for cache miss counting:

* Figure out how big blocks are in terms of what’s being accessed what values map to the same block?
* Figure out what values map to the same set
  + Rule: find # bytes in way, values separated by K\*#bytes in a way map to the same set.