Interesting Notes

Alex Clemmer
Student number: u0458675

ISA and compiler determine instruction count for a given program.

Where/how hazards occur: In a pipelined process, instructions and data tend to move left to right through the stages of execution. That is, an instruction is fetched, then decoded, and so on, moving to the right until completion.

When this is not the case – for example, when we writeback, we are in the WB stage, and we send back the data into the register file, which is in the ID stage – is where we get hazards.

There are, in this model, two cases where this happens: once for writebacks, and once for instruction selection (*i.e.*, when we decide whether to branch, jump, or just increment).

Reading and writing registers in one cycle: The book first makes a "logical" distinction between registers written to during WB and registers read from during ID. This is why they shade only half the boxes in the pipelining diagrams. But further, the registers are assumed to be written in the first half of the cycle and read during the second half.

What I-Mem really is: Instruction memory, or I-Mem is really just the memory that holds the loaded instructions from some program. Since the instructions are literally specified by the program, we can just load them into I-Mem.

WB has no pipeline register: This would be redundant. Every stage of the pipeline should update the state of the processor, and the pipeline registers are there simply to hold this state. Since writing to the register during WB tracks the state anyway, there's no real point in having a pipeline register also.

The PC is basically a pipeline register: The PC can be thought of as a pipeline register that just supplies information to IF – it just keeps track of the state of the processor at a certain point. Except for one thing: when there is an exception, we discard all the pipeline registers (*why?*), where we MUST KEEP the PC.

The relationship between cache size and latency: Interestingly, cache size is not directly related to latency. Cache size in some ways has an effect on *when* we miss (and thus, when request latency is worst), but the time it takes to get that information is independent of this: we may have to travel across a LAN in addition to seeking on the disk, or we may have to query some other server.

In other words, the architecture of system(s) ends up determining the latency, not the size of the cache.

Why organize caches into a hierarchy: There are two basic types of locality: spatial and temporal. The highest level cache addresses the fact that access times for the most recent-accessed data should be fast, and the next-highest addresses the fact that most data accessed is accessed in close proximity of other recently-accessed data.

That is, L1 cache is small and fast; data on it is meant to be accessed repeatedly. L2 is larger and almost as fast; it is meant to supply the data in close proximity to the stuff in the L1 cache. In this way, when the hit rate is very high, then the memory in cache appears to be as fast as L1 and as large as L2.

Memory is usually structured in true hierarchies: In most memory models, if data is in memory level i, it is also in level $i + 1$, assuming that there is such a level (and there would not be if i is the hard drive.

Does communication bandwidth or processing bandwidth matter more? Depends. Look at the model of some computer on the internet: clearly communications is the bottleneck for the browser, as it doesn't have to compute a whole lot. But for the proxy server, it's both processing and communications bandwidth: faster servers are expensive, and communication BW is expensive too. In servers, it's really both, but more the processing BW. Processing power is expensive.

Should cache lines be large or small? Tough call. Increasing their size makes us miss less often, but also makes misses cost more time. The larger the cache line, the more time it takes to populated it with fresh data.

But there's more to it. The miss rate drops off as the cache sizes get larger – perhaps because after a certain threshold programs tend to just jump outside of the scope of the cache line, rather than incidentally using some variable that is close by, but simply out of scope.

Why not use early restart for standard memory (rather than just I-Mem)? We might, but it's less successful. With I-Mem, we tend to fetch sequentially from cache. So when we have a miss, we can begin the fetch of a new cache line and just restart the instruction as the word we need becomes available (rather than waiting for the whole line to show up). We can continue to fetch instructions "just in time" until the transfer of memory is complete.

With regular data caches, memory access tends to be more erratic. If we incur a miss and start executing immediately when the word we need becomes available, all other fetches must wait until their instruction is loaded. In the case of just waiting for the data that's in the same cache line to load, the time is simply unpredictable. In the case of a cache miss, we won't know and will wait for the memory to load into cache before we start loading a different line into cache.

What's the difference between a block address and a byte address? If we say we have some byte address, and we want to know what block address it corresponds to, this is how we derive it:

$$\text{block address} = \frac{\text{byte address}}{\text{block size (in bytes)}} \% \text{ \# of blocks in cache}$$

In other words, we take the ordinal of a specific byte in a sequence of bytes (*e.g.*, byte number 10 in a sequence of 20), and divide that by the number of bytes in a block. Then we take that number (it should be a natural number) and mod it by the number of blocks in the cache. This gives us the specific place our bite would be mapped to in a cache.

Highly useful when we've indexed some information and need to know where to find it in a large cache.

What is a "streaming" workload? Streaming workloads bring in a lot of data but reuse almost none of it. This is in contrast to most other types of workloads, which may take in a lot of data, but also tend to reuse it at least a little.

Questions

Writing regs on clock edge If contents of a register only change on clock edge, how is writing happening in the first half of the cycle, while reading happens the second half? Shouldn't this be reversed?