CS 4515 Computer Architecture Notes

Christopher Myers

June 13, 2020

Contents

1 Introduction: Basics of computer architecture

Originally computer architecture used to be about instruction set design — what sorts of instructions are available, data types, etc. Nowadays nobody writes in assembly or machine code anymore (except perhaps for some very niche cases). Technically the instruction set is now a private contract between the hardware architect and the compiler writer, with very few exceptions. Device interfaces are similarly private contracts between hardware developers and the OS (or device driver writers). As a point of interest, Apple has changed the Macintosh instruction set twice in its lifetime, and even changed the endianness once, all without upsetting customers or existing software.

Anyways, computer architecture ultimately addresses hurdles of implementation in computers — speed, power, energy, size, and appropriateness for applications. This is a very quantitative approach, with lots of simulations and analysis involved to determine what's best. Current computer performance is reaching a cap of sorts, with 50% growth per year in the 90's but only 2-3% growth per year in modern times. Instruction-level parallelism is no longer viable on its own to increase performance, with other models such as thread-level parallelism (TLP) and data-level parallelism (DLP). Hardware size statistics remain high, however, with transistor density increasing 35% in one year, die size decreasing by 10-20% a year, DRAM capacity increasing 25-40% per year (although slowing), flash RAM capacity increasing 50-60% per year, and hard drives increasing 40% a year.

1.1 Processor types

"RISC" stands for "reduced instruction set computer", wherein the CPU contains very few instructions and few high-level features, but can run much faster and is much simpler as a result. This is in contrast to CISC, or "complex instruction set computer", which are machines that include more complex instructions that originally were intended to make life easier for assembly language programmers. For example, the IBM 360 included a "edit and mark" instruction that did something similar to C's strtok function.

RISC machines focus on the common case and leave software to handle special cases. Each instruction is exactly one word and does exactly one thing, with simple addressing modes (e.g. register + offset). Each instruction also takes exactly one cycle, with the exception of floating point arithmetic which is always done on a separate coprocessor.

As it turns out, Intel x86 and x86_64 machines are really RISC machines with microcode implementing the more complicated instructions. The core, internal instructions are RISC-like in nature and have simple addressing modes, with complex instructions just subroutines of core instructions, stored and invoked in hardware.

Ultimately, RISC machines are used for performance reasons first, then for parallelism, which as it turns out is a good chunk of this course. This is where Amdahl's law becomes relevant. Let P be the ratio of time in the parallelizable portion of an algorithm to the total time of the algorithm. If T_S is execution time in a serial environment, then:

$$T_P = T_S \times ((1 - P) + \frac{P}{N}) \tag{1}$$

In other words: parallelizing has diminishing returns. This is sometimes described as "Amdahl's disheartening law".

2 Pipelining

There are three (and now, thanks to Google, one extra) types of parallelism: data parallelism, task parallelism, and pipelining (plus Google's massive parallelism).

Recall the von Neumann model: fetch one instruction from memory, increment the program counter, read registers as needed, perform one integer operation, access memory, and write to registers. Each phase is independent of the others and each requires different circuitry. This means that these phases of execution can be laid out horizontally on a timeline, with different instructions executing in each phase at each time; this allows each circuit to be busy at all times, instead of idling while waiting for something to do. Nearly all processors since the mid-1980's have been designed to accommodate pipelining, and in fact some RISC machine architectures are specifically designed with pipelining in mind.

For example, a typical RISC pipeline looks a bit like this:

1. Instruction fetch
2. Instruction decode and register fetch
3. Execution *or* address calculation
4. Memory access
5. Write-back to register

Note that while every architecture partitions its instruction set differently, this is a textbook example of pipelining. Unfortunately, pipelining isn't that simple, since there are hazards: structural hazards, data hazards, and control hazards.

2.1 Structural hazards

Structural hazards are when there are resource conflict, data hazards are when instructions' dependencies include results of previous instructions, and control hazards are when the code executed depends on what control flow branch is taken. Some good examples of structural hazards are series of instructions that would result in multiple words read per cycle (not allowed), or cases where the instruction cache and the data cache would have to be the same. When situations caused by structural hazards arise, stalls in execution occur, wherein an instruction must wait to be pipelined until the previous instruction reaches the right point in its pipeline. This is potentially something that can be accounted for by the compiler writer.

2.2 Data hazards

There are three major data hazards:

- **Read after write:** an instruction wants to read from memory or a register that is written to by the previous instruction. Compiler writers call this a dependence.
- **Write after read:** An instruction writes a new value after a previous instruction accesses it, potentially before the old version is read. Not a problem so far with the material covered. Compiler writers call this antidependence.
- **Write after write:** The second instruction writes a value to the same register as the previous instruction. Compiler writers call this an output dependency, since the second instruction to execute should rightfully output the final value to the register.

Some hazards can be avoided using forwarding, in which a piece of hardware recognizes the hazard and sets the ALU that executes the instruction to use its output as the data source for its next computation, rather than using the [old] data fetched from the register. The compiler or assembly prorammer can also simply execute instructions in a different order in order to help optimize for instruction pipelining.

2.3 Control hazards

Control hazards are where one of the instructions used involves a branch, but the actual branch taken won't be known for sure until the relevant instructions finish executing. Branch stalls do have a penalty, but that penalty can be minimized. Processors often perform speculative execution on branches they guess will be taken, and discard the results of that execution if the guess was incorrect; this is one method to avoid branch stalls. The branch could also be defined to take place before a following instruction (???) which can help optimize for branch prediction.

Modern processors use dynamic branch prediction over static branch prediciton. The simplest scheme is a "branch history table" that store a 1-bit cache of recent branch instructions; instructions in that direction are fetched. If the prediction is incorrect, the bit is inverted and stored again. This is sub-optimal, so a simple state machine operating on *two* bits can be used instead. The bits on or off depending whether the branch was taken or not taken — a 11 means taken twice, a 00 means not taken twice, a 10 means taken once, and a 01 means

not taken once. The number at the front tells what prediction will actually be made; for example, if a 01 gets a not-taken response, it'll be changed to 00 to indicate that it happened twice, and the prediction is more stable. Sort of, it's complicated without using a real state machine diagram.

Predictors can also be correlated — that is, if one conditional depends on input to another, the processor can take a guess at how the second branch goes based on what happened to the first. This is rare and tricky to get right, but it can happen.

2.4 Miscellaneous Hazards

Sometimes code generates exceptions, like an illegal opcode or a divide by zero. Sometimes interrupts are generated, which are hardware (or sometimes software for soft IRQs) signals to tell the processor to switch to a new instruction stream.

3 Caching

Caching is one of the most significant things that can be done to improve peformance, by storing needed data in memory.

3.1 Memory Types

- **Static RAM (SRAM):** An array of cells where power flows constantly. Data is retained until overwritten or the memory is powered off. SRAM is very fast to both read and write. The caches inside processors (L1-3) are made from SRAM, as are DRAM buffers and other high-performance needs.
- **Dyanmic RAM (DRAM):** Uses one transitor and one capacitor per cell. Data is stored as a charge on the capacitor. DRAM is much slower than SRAM but is also much denser. The capacitors are discharged on read, so they must be rewritten after each read. The capacitors also "leak" over time and must be periodically recharged, so DRAM similarly loses data on poweroff.[1]
- **Flash RAM:** Non-volatile, holds data when powered off. Underlying hardware uses a floating gate. Flash RAM is written in large blocks, an entire block at a time, and if you want to write something to the block, the whole thing has to be erased first. It only lasts about 10^5 erase/write cycles, though. Requires support circuitry to manage blocks, map logical to physical blocks, and cache data for writing and reading. Flash RAM is used for SSDs, cellphones, and environments with vibrations, typically for long-term storage.

3.2 Basics

Caches are parts of memory (in one way or another) that are used to store frequently-accessed data or data assumed to be accessed soon, as an alternate fetch location from some larger, slower data source. For example, browsers store data loaded from a page in their cache so that they don't have to re-fetch it if data from that page (like an image) is needed again.

3.3 Processor caches

Processor caches are organized into multiple levels, with each level smaller, faster, closer, and more expensive than the next lower level. Lower levels have longer latency to access, but are still much faster than RAM all the same. These caches are on the order of 64kb for an L1 cache, 256kb for an L2, and 4-8 MB for an L3.

Caches are essential for improving the speed of computing. Since 1980, memory has gotten 10x faster, but processors have gotten *10,000 times* faster. Caches are organized into sets, entries, and blocks (or S, E, B). There are $E = 2^e$ entries per set and $S = 2^s$ sets, and $B = 2^b$ bytes per cache block (the actual stored data). The ultimate cache size is $C = S \times E \times B$. The address of a word in the cache starts with t bits for a tag, s bits for the set index, and b bits for the block offset. To read from the cache, the set is first located and the lines in

[1] Interestingly there is something called a "cold boot" attack where DRAM is cooled down to help keep capacitor charge, then the computer is restarted with the goal of keeping its original memory intact, allowing an attacker to to read it.

it are checked for matching tags. If one is found and the line is marked as valid, the read operation results in a cache hit.

3.4 Cache Optimizations

The three most immediately obvious optimizations are to increase block size, make caches bigger, and ncrease associativity, all to decrease miss rate. Implementing multilevel caches can also help reduce the miss penalty, as can giving priority to read misses over writes. Finally, you can avoid address translation during indexing of the cache to reduce hit time. *Technically*, with a bit of handwaving, physical memory is a cache of virtual memory. Page faults are therefore like cache misses, and page replacement is eqiuvalent to flushing cache entries.

4 Superscalar Processors

Superscalar processors are processors that can execute more than one instruction per cycle, in a single processor, from a single instruction stream. This requires the ability to fetch more than one instruction, multiple instruction units, the ability to deal with multiple control and data hazards on each cycle, and the ability to write and/or forward results. This all needed to get a CPI (cycles per instruction) of less than 1.

The basic idea involves a data structure called a scoreboard, which records the status of all functional units, the status of instructions, hazards, and the status of registers (more than just the standard architecture registers). The steps of execution are:

1. The scoreboard checks the functional unit needed by an instruction. If it's free and does not share a destination register with any other active instructoin, the instruction is **issued** to the functional unit. This guarantees there are no write-after-write hazards. If there is a hazard, the instruction issue stalls and no more instructions will be issued until the hazards clear. This replaces part of the instruction decode (ID) step.
2. The scoreboard monitors source operand availability. An operand is available if no earlier issued active instruction will write it. When available, a functional unit may **read operand(s)** from register(s), and execution begins. This resolves read-after-write hazards. Instructions may be sent into execution out-of-order.
3. The functional unit begins **execution** upon receiving operands. When the result is ready (and this may take longer than one cycle), the functional unit notifies the scoreboard.
4. The scoreboard checks for write-after-read hazards after completion of execution, stalling the completing instruction if necessary.

4.1 Parts of the Scoreboard

The scoreboard includes the instruction status and the status of different functional units:

- Busy
- Op — operation to perform
- Fi — destination register
- Fj, Fk — source register numbers
- Qj, Qk — functional units producing source register Fj, Fk
- Rj, Rk — flags indicating whether Fj, Fk are ready and not yet read. Set to "no" after operands are read.

5 Dynamic Scheduling

Under dynamic scheduling, hardware rearranges the execution order of instructions for the purpose of avoiding stalls. This simplifies the job of compiler writers and helps add tolerance for unpredictable delays like cache misses.

6 Thread-level Parallelism

The four main types of thread-level parallelism are:

- **SISD**: Single instruction stream, single data stream. Basic uniprocessor.
- **SIMD**: Single instruction stream, multiple data stream. Also known as vector architecture.
- **MISD**: Multiple instruction stream, single data stream. No commercial system like this has ever been built... except for the Space Shuttle control system, and possibly Orion, in which computers "vote" on the outcome.
- **MIMD**: Multiple instruction stream, multiple data stream. Basic multiprocessor.

The main focus of this section is MIMD, which is an implementation of thread-level parallelism.

Thread level parallelism involves having multiple processors in the same CPU, with shared memory but independent registers, program counters, etc. These are used to gain further performance gains since it's tough to get processor speeds above about 4.0 GHz due to thermal problems. There's also a limit to the benefits you can get out of instruction-level parallelism, further motivating thread-level parallelism.

Unfortunately taking advantage of this does require some software changes. Software must be made multi-threaded, which is easier if the software performs multiple independent tasks. The two main problems involved in threading are the sharing of caches and synchronization. In SMP (symmetric multiprocessors), accessing memory is uniform between processors and involves sharing a bus; this makes coherency simple.