Computer Architecture I and II

CS 350: Computer Organization & Assembler Language Programming

11/6: Solved

A. Why?

- Modern computers are much more complicated than the LC-3.

B. Outcomes

After this lecture, you should understand

- The basic ideas behind memory caching, instruction pre-fetching, and instruction pipelining.
- The difference between multiprogramming/multitasking and multiprocessing.

C. Speeding Up Computers: Change the ISA?

- One way to categorize ways to improve computer execution is by looking at whether they change the **instruction set architecture (ISA)**.
- Today we'll look at techniques for speeding up execution of the instructions as given. Next time we'll look at changing the instruction set.

D. Caching

- A **Memory Cache** is a small, fast memory that stores copies of values in main memory. When accessing memory, we check the cache first. If the cache contains often-used data, then our effective time to access memory decreases.
- We may use different caches to hold data vs instructions because they can differ in pattern of uses.
- It's not always the case that larger caches are better; the extra expense may not bring enough benefits.
- People have found that multiple levels of cache can help: If a search of the very fast/small level 1 cache fails, we go to the somewhat slower but less expensive level 2 cache before checking main memory. (We even see level 3 caches in some implementations.)

- **Write-back policy**: Writing a new value to memory actually changes the value in the cache; when should we actually update main memory?
  - In a **Write-Through Cache**, we update main memory every time a new value is stored; this keeps memory synchronized with the cache but is slower than in a **Write-Back Cache**, where we only write the value to main memory when it's evicted from the cache (presumably to make room for some other piece of data).
- Keeping the cache synchronized with memory ("**Cache Coherence**") is made harder by having I/O devices or other processes or CPUs that can update memory independently from the CPU.
- An **Instruction Cache** holds recently-used instructions. If the entire body of a frequently-executed loop fits into the cache, we can get excellent speedup; this is why we prefer programs with short loops over ones with long loops.
- We can try to *Pre-Fetch Instructions* (read the next instruction(s) into the cache from memory before we need them).
- Modern CPUs have sophisticated algorithms for trying to predict which instructions might be used. Branch prediction: Which side of a conditional branch might we take? Keep info on what was taken in the recent past? Fetch both sides of the branch?
- Once the instruction cache gets full, we need *Eviction Policies* to tell us which instructions to remove when we need room in the cache. E.g., *Least-Recently Used* is a popular policy

E. Instruction Pipelining

- **Instruction Pipelining** is where we overlap the processing of multiple instructions. We speed up decoding and increase the number of instruction executions per second by fetching instruction 1 while decoding instruction 2 while calculating effective addresses for instruction 3 and so on.
- Compare with memory caches, which speed up the Fetch instruction, Read Operands, and Write Results part of the instruction cycle.

- **Latency vs Throughput**: Latency is the time to execute one instruction; throughput is the number of instructions completed per unit time. **Instruction pipelining aims to increase throughput, not decrease latency**.
  - One analogy uses everyday plumbing: Latency is the time it takes for water to come out of the faucet when you turn it on; throughput is the rate at which water comes out once it does start coming out. (Latency depends on the length of the pipe; throughput depends on the diameter.)
  - Another analogy uses a bucket brigade: The only person tossing water on the fire is the final person, but that person gets to toss more water on per unit time because other people are moving buckets through the line.
- A *pipeline stall* is when one part of an instruction pipeline has to wait for some other part.
  - One cause is a data dependency (one instruction needs the result of another). To keep the pipeline full, modern computers use out-of-order execution: The control unit rearranges the order of instructions to reduce data dependency delays. When you toss conditional branches into the mix, you can find yourself speculatively executing an instruction (before you know you need it) but then having to un-execute it if your branch prediction was wrong.
- Using pipelining, we may be able to complete more than one instruction per clock cycle. A computer is superscalar, scalar, or sub-scalar depending on whether it can complete at most > 1, $= 1$, or < 1 instruction per clock cycle.

F. Multiprogramming/Multitasking vs Multiprocessing

- In **Multiprogramming (a.k.a. Multitasking)**, **the OS system lets multiple programs share a CPU** by running each program for a while and then switching to some other program.
  - In **Concurrent Execution**, **execution appears to be in parallel** but we actually only run one program at a time, each for a short quantum of time. By making the quantum small enough, we make programs look as though they are executing simultaneously.

- By mixing the execution of processes that want the CPU vs processes that want I/O devices, we work toward having processes do less waiting (thereby decreasing latency) and keeping the CPU and I/O devices busier (thereby increasing throughput).
- In **multiprocessing**, we have multiple CPUs (either on the same chip or different chips), so we can perform operations truly in parallel. Each processor has its own control unit and arithmetic-logic units; they share memory (though each processor might have a bit of local memory too).
- In **Asymmetric Multiprocessing**, the processors are different. Often this is done so that specific processors can be specially optimized for a particular kind of work. (A CPU with a video processor on a graphics card uses asymmetric multiprocessing.)
- In **Symmetric Multiprocessing** (**SMP**), we have identical processors sharing memory. This makes it easy to move programs from one processor to another, so it's easier to balance loads across processors. A **Multi-Core** computer has multiple processors on the same chip. Usually they are identical processors, so we have symmetric multiprocessing.
- When processors have individual caches there's another instance of the cache coherence problem (keeping them synchronized across processors); if processors share a cache, we have to be careful to arbitrate simultaneous requests.
- **Race conditions**: Whether we have multiple processors or multiple tasks running on one processor, if programs share a resource (typically variables in memory), we have to ensure that the overall computation is correct regardless of what order programs run in or how fast their processors are (if there are more than one of them).
    - In a **Race Condition**, the correctness of our computation depends on the relative speed of execution of different parts of the computation. Race conditions and synchronization make it hard to write correct programs that run concurrently or in parallel.

Processes and Threads

- A **process** is a program running with its data[*]. An analogy: A program is like a script for a play — it's a set of instructions. **A process is like a performance of a play** — you have resources like actors, props, and a stage allocated, and the actors are doing things: Saying lines, interacting with props, etc.
- A **thread of execution** is an execution path through a program (along with the data for the thread). Going back to the play analogy, a thread is like an individual actor. A play performance involves the coordinated actions of all the actors. Execution of a process can involve coordinated action between threads.
- Contemporary programs are often **multi-threaded** (involve multiple threads). You might have one thread listening to the keyboard, another listening to the trackpad, another doing computation, and another displaying things to the monitor.) In the same way, a play typically has multiple actors, though you can have one-performer plays where one actor speaks, say, Ebenezer Scrooge's lines and then changes to Bob Cratchit's lines, etc.
- Multithreading is used partly because it can make the program **easier to write and understand**. It's also used because **changing threads within a process is cheaper than changing processes**. Threads share the same context (the stage and props), but processes don't. To change processes, you have to clear the stage and get everyone involved in the other play onto the stage.
- Running multiple threads: One possibility is to run different threads simultaneously on different cores or processors. (In the analogy, it's like having multiple actors saying lines simultaneously.) Another is to run multiple threads concurrently on one core. We can run each thread for a slice of time and switch (similar to multiprogramming).
    - In Intel's hyperthreading approach, one core can simulate two cores by having two threads share the same instruction pipeline. Benchmarks indicate perhaps 20% increase in average computation speed.

[*] The nomenclature is confusing: Running multiple processes on one CPU is called **multiprogramming**. Having multiple processors is called **multiprocessing**.

G. Design of Instruction Set Architectures (ISAs)

- **Computer organization vs computer architecture**: In computer organization we study how to implement a given ISA. In computer architecture we study the design and implementation of ISAs.
- Last time we discussed computer organization: Various ways to run instructions faster (caching, pipelines, multiple cores or processors).
- Let's look at some basic computer architecture: Can we make Instruction Set Architectures (ISAs) that are better-suited for our programs and/or easier to implement in hardware?
- **Some factors that affect ISA design**:
  - **Datatypes supported**
  - **Operations: Basic op codes**
    - How many op codes should there be?
  - **Operands: How many per instruction?**
  - **Addressing modes**: Where can operands come from; where can results go to? Memory? Registers? Immediate? Stack?
  - Outside limitations: Technology, cost

H. Hardware Datatypes

- Common hardware datatypes
- For flexibility, you want various sizes of integers and floating-point numbers.
  - Integers: 8, 16, 32 bits?
  - Floating-point: 32, 64 bits?
  - Characters (more complicated with Unicode's 16-bit chars)
  - Strings? How terminated?
  - Vectors of data?

I. Operands and Addressing Modes

- Operands: **How many operands** do instructions have? E.g.,

- 3 addresses: `ADD r1, r2, r3   ; r1 = r2 + r3`
- 2 addresses: `ADD r1, r2    ; r1 = r1 + r2`
- 2 addresses: `ADD r, loc    ; r = r + Mem[loc]`
- 1 address: `ADD loc`
  `; accumulator = accumulator + Mem[loc]`
    - The one register is traditionally called the "accumulator."
- 0 addresses: `ADD`
    - Typically used for a machine with a built-in stack, such as HP RPN calculators: We pop off two values, add them, and push the result back onto the stack.

- **Addressing Modes:** How can we specify an operand / result?
- LC-3 illustrates many modes
    - Register [As in `ADD R1`, etc.]
    - Immediate [sits inside instruction]
    - Program counter + immediate offset
    - Base register + immediate offset
    - Indirect through PC + immediate offset
    - Memory-mapped I/O register
- Some other modes people have used:
    - Absolute: address inside instruction
    - Base register + index register + offset [nice for array lookups]
    - Address + length [nice for strings]
    - Register indirect with automatic pre/post-increment/decrement
        - E.g. use address in `R3` and add 1 to `R3`
        - Nice for indexing arrays with a loop
    - Stack

J. Microarchitecture, Microcode, and Microprogramming

- The **microarchitecture** of a computer is its implementation of the ISA.
    - What are the components of the processor, and how do they interact?
        - Data path: User registers, internal registers (like condition code, memory-is-ready, ...), execution units (like the ALU), pipeline components, etc.
        - Control path: how do we move/manipulate data?
        - Control signals (Tell memory to read; Tell memory to write; Memory says its ready; Tell ALU to add; Tell ALU to AND, ...
- Each ISA instruction consists of a sequence of lower-level microinstructions that specify what control signals and registers to look at and set.
- **Microinstructions**: A microinstruction causes internal state changes of the CPU. E.g., `IR` ← `MDR`
- A microsequencer selects the current microinstruction to execute.
    - One microinstruction per state, stored in a read-only Control Store
- The microcode for a computer is the set of its microinstructions, and microprogramming is the writing of microcode.
- **Example**: Branch instruction
    - Calc `FLAGS` = `IR[9:11]` `AND` `CC`; `if` `FLAGS` ≠ `000` then `PC` ← `PC` + `SignExtend`(`IR[0:8]`); `Go to state` "Start executing next instruction."

K. Separating Microarchitecture from Instruction Set Architecture

- The ISA is the specification for the microarchitecture
    - You can implement one ISA using different microarchitectures
    - Allows for tradeoffs between cost and speed of hardware.
    - Allows for creation of hardware without having to rewrite assembler code.
- History: IBM 360/370 family had complicated ISA
    - Many ways to represent numbers, many different ADD instructions, so implementation of the ISA was complicated.

- Different models had different capabilities (speed, memory, bus width, hardware vs software support for floating-point numbers).
- They invented downward compatibility so you could upgrade without rewriting your programs.
- Same thing happens today: Intel designs a new processor
  - It supports the x86 ISA because it's microprogrammed to.
  - Different processors may support the ISA in wildly different ways.

L. Complicated Instruction Set Architectures: Good or Bad?

Pros of complicated instruction set architectures

- **Having many op codes, addressing modes, and datatypes is nice**.
  - **Programs are easier to write.**
  - Less tedious for programmer, fewer errors
- **Higher code density** (oomph per byte of instruction)
  - Especially if you have instructions of different lengths. (E.g. different Add instructions.)
- **Programs fit into less memory**
  - Very helpful when memory was expensive.
  - Still helpful today because instruction access is faster, more instructions fit into caches.
- **Programs might run faster**
  - Frequently-used operations can be fine-tuned for silicon.

Cons of complicated instruction set architectures

- **Hardware design is harder and slower**
  - More combinations of factors to worry about
- **Hardware requires more resources**
  - You need more silicon and electrons
    - You have to skimp on registers & caches.

    - Fabrication is harder; you need bigger power supplies, more cooling.
    - Testing is more complicated
- **The hardware isn't necessarily faster**
    - Fine-tuning instructions for speed gets dropped from the schedule.
    - Exist instances where people wrote programs that were faster than the built-in hardware. (INDEX instructions on VAXen.)
- **Compilers are harder to write, slower, and don't generally produce optimal code**
    - It's hard to write compilers that take advantage of all the available instructions
    - So there are instructions that tend to never get used.
- **The clock rate tends to be lower**
    - The clock rate is determined by the slowest sub-operation the CPU needs to do.

M.RISC and CICS ("risk" & "sisk") Computers

- **RISC: Reduced Instruction Set Computer. The "Reduced" in RISC** refers to the amount of work needed to execute the average instruction.
    - Very successful RISC machine: ARM ("ARM" was "Advanced RISC Machine" architecture; now just ARM.) 32-bit RISC processor, low power and cost, high performance. ARM or ARM-based CPUs used in many portable & consumer devices.
- **Load/Store Architecture** One approach to RISC design involves separating memory load/store instructions from calculation instructions so their implementations can be optimized separately.
- Some other approaches to RISC design:
    - Reduce the number of instructions supported (simplifies instruction execution hardware)
    - Reduce the number of datatypes supported (simplifies calculation hardware)
    - Reduce the number of instruction formats (simplifies decoding hardware)

    - Reduce the number of addressing modes (simplifies address calculation hardware)
    - Reduce number of special cases by having an **orthogonal instruction set** (every kind of operation works with every addressing mode; compare with LC-3 where each operation supports only certain addressing modes).
    - Use freed-up chip space and power to support more registers and caches
    - Use higher-speed clocks.
- **Note: Fewer instructions and orthogonal design are not necessarily part of a RISC design**
- Difficulties using RISC computers
    - RISC programs tend to be longer because of the reduced ISA.
    - Compilers have to be rewritten to work with RISC hardware. The hope is that smart compilers can optimize code better than for more complex ISAs.
- **CISC: Complex Instruction Set Computers.** The name CISC was made up to designate older non-RISC designs. Examples include IBM System/360 etc, Intel 486 and on. CISC aims for small code size by using complex data types, complex instructions, and good hardware.
- **RISC vs CISC differences have diminished/blurred over time**
    - CISC computers have included faster, simpler instructions. CISC execution has improved with instruction pipelining and out-of-order execution. (These are microprogramming-level versions of the optimizations that smart RISC compilers were supposed to use.)
    - RISC designs have included more-complicated instructions. E.g., the ARM now includes load/store multiple registers; it also has auto increment/decrement modifiers.

Computer Architecture I

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Modern computers are much more complicated than the LC-3.

B. Outcomes

After this activity, you should be able to describe the basics of

- Caching, pipelining, multiprogramming, multiprocessing, race conditions, and processes and threads.

C. Questions

Computer Architecture I

1. Are larger memory caches always better than smaller ones?
2. How do short loops in programs and instruction caches relate?
3. What is instruction pipelining? Does it increase or decrease latency and throughput?
4. What is the difference between multiprogramming and multiprocessing?
5. Do multicore computers use symmetric or asymmetric multiprocessing? What about a video card vs our general-purpose CPU?
6. What's a race condition?
7. How are programs and processes different?
8. What is a thread of execution?
9. Why do people write multi-threaded programs?

Computer Architecture II

1. What's the basic difference between computer organization and computer architecture?
2. Name the main factors that affect the design of an instruction set architecture.
3. List some advantages to having a complex ISA

4. List some disadvantages to having a complex ISA
5. What are RISC computers? What characterizes RISC design? What characteristics are sometimes but not always found in RISC design.
6. What are CISC computers? How has RISC vs CISC design changed over time?

Activity 14 Solution

1. Surprisingly, no. There can be anomalous situations where enlarging a cache causes a slowdown.
2. Smaller loops have a better chance of having all their executable code fit inside the instruction cache, which speeds up execution.
3. In instruction pipelining we overlap the processing of multiple instructions. This increases throughput but does not decrease [12/9] latency.
4. In multiprogramming, the OS system runs multiple programs concurrently, by quickly switching from one program to another. In multiprocessing, we have multiple CPUs, so both programs execute in true parallel fashion.
5. If the multiple cores are identical, we have symmetric multiprocessing. Video cards and general-purpose CPUs are asymmetric.
6. Say two programs execute concurrently or in parallel and share a resource. If the correctness of overall execution depends on the relative speed of execution of the programs, we have a race condition.
7. A program is a set of instructions; a process is the execution of that program (sitting in memory, with resources allocated, etc).
8. A thread of execution is an execution path through a program.
9. Multi-threaded programs can be easier to write and understand than monolithic single-threaded programs. If different threads execute on different CPUs or cores, then actual execution can be faster.

Architecture II

1. In computer organization, we look at how to implement an ISA; computer architecture extends this by including ISA design.
2. What datatypes are supported; the number of opcodes; the number of operands per instruction; the number of addressing modes.
3. Programs are easier to write; you get higher code density, so programs fit into less memory. Also, programs may run faster

4. Hardware design is harder and slower. The hardware itself can require more resources (e.g. power, cooling). Programs may execute slower and with a lower clock rate. It's hard to write good compilers for complex ISAs.
5. RISC: Reduced Instruction Set Computer or Load-Store" architecture. "Reduced" in the sense of amount of work to execute an instruction. The "Reduced" in RISC refers to the amount of work needed to execute an instruction. Doesn't necessarily imply having fewer instructions or an orthogonal instruction set
6. CISC: Complex Instruction Set Computers. RISC vs CISC differences have diminished/blurred over time