The Stored Program Concept
A limited repertoire you say?
Well … it’s just like Lego
Only with a surfeit of pieces

Mix, match and combine in ways
That are infinite
Literally and metaphorically

All leading to a machine
With limits defined by creativity

Frequently asked questions from the previous class survey

- Spatial vs Temporal locality
- Are more cores the way to go?
- Is there any redundancy built into the memory hierarchy?
Topics covered in this lecture

- Caching
  - Direct mapped
  - Associative
  - N-way associativity
- Stored Program Concept
- The van Neumann architecture

Cache lines

- For example, an 8 KB cache line is often organized as a set of 512 cache lines of 16 bytes each

8 KB with 512 16-byte cache lines
Types of caches

- Direct mapped caches
- Fully associative caches
- N-way associative caches
A **direct mapped cache** is also known as a one-way associative cache

- In a direct-mapped cache, a particular block of main memory is always loaded into—mapped to—the exact same cache line.
- This mapping is determined by a small number of bits in the data block’s memory address.

Direct-mapped Cache

- 9 bits (4 through 12) of the physical memory address provide an index to select one of the 512 cache lines within the cache ($2^9 = 512$).
- Bits 0 through 3 determine the particular byte within the 16-byte cache line.

8 KB with 512 16-byte cache lines
Problems with a direct mapped cache

- Two different memory addresses located on 8KB boundaries cannot both appear simultaneously in the cache.
- How many such addresses exist in our 32-bit system?
  - $2^{19}$ 8KB blocks exist in our system.
  - $2^{19}$ 512 ($2^9$) blocks of 16-bytes ($2^4$) each.
    - $2^{19} \cdot 2^9 \cdot 2^4 = 2^{32}$ (the size of the main memory in our example).

The ideal world: A fully associative cache

- The cache controller can place a block of bytes in any one of the cache lines present in the cache memory.
- While this is the most flexible cache system, the extra circuitry to achieve full associativity is expensive and, worse, can slow down the memory subsystem.
- Most L1 and L2 caches are not fully associative for this reason.
Trade-off space

- A fully associative cache is too complex, too slow, and too expensive to implement
- But a direct-mapped cache is too inefficient

A compromise: the n-way associative cache

- In an n-way set associative cache, the cache is broken up into sets of $n$ cache lines
- The CPU determines the particular set to use based on
  - Some subset of the memory address bits, just as in the direct-mapping scheme, and …
  - The cache controller uses a fully associative mapping algorithm to determine which one of the $n$ cache lines within the set to use
For example, an 8KB two-way set associative cache subsystem with 16-byte cache lines

- Organizes the cache into 256 cache-line sets with two cache lines each.
- Eight bits from the memory address determine which one of these 256 different sets will contain the data.
  - $2^8 = 256$
- Once the cache-line set is determined, the cache controller maps the block of bytes to one of the two cache lines within the set.

For example, an 8KB two-way set associative cache subsystem with 16-byte cache lines

- This means two different memory addresses located on 8KB boundaries (addresses having the same value in bits 4 through 11) can both appear simultaneously in the cache.
- However, a conflict will occur if you attempt to access a third memory location at an address that is an even multiple of 8KB.
An 8 KB two-way set associative cache subsystem with 16-byte cache lines

Eight bits (11 through 4) provide index to select one of 256 sets $2^8=256$

A cache line set comprising two cache lines

The cache controller chooses one of two different cache lines within the set $2^8=256$

16-byte cache line

8 KB with 2-way set associate cache with 256 sets of two (16-byte) cache lines each

What if we have a 4-way associative cache

- A four-way set associative cache puts four associative cache lines in each cache-line set
- In our example, 8KB cache, a four-way set associative caching scheme would have 128 cache-line sets with four cache lines each
- This would allow the cache to maintain up to four different blocks of data without a conflict, each of which would map to the same cache line in a direct-mapped cache
2-/4-way set associative vs direct mapped

- A 2- or 4-way set associative cache is
  - Much better than a direct-mapped cache and
  - Considerably less complex than a fully associative cache

Can we keep increasing the number of lines in each cache-line set?

- The more cache lines we have in each cache-line set, the closer we come to creating a fully associative cache
  - With all the attendant problems of complexity and speed
- Most cache designs are direct-mapped, two-way set associative, or four-way set associative
  - The various members of the 80x86 family make use of all three
Etymology Tidbit: Program “execution”

- **Execute**
  - Verb
  - Carry out or put into affect a plan, order, or course of action

- Contrary to some lore
  - No bits are actually “killed” when a program “executes”
The Stored Program Concept [1/2]

- Compared to all the machines around us, the most remarkable feature of the digital computer is its amazing versatility.
- Here is a machine with a finite and fixed hardware that can perform an infinite number of tasks:
  - From playing games to typesetting books to driving cars.
- This remarkable versatility—a boon that we have come to take for granted—is the fruit of a brilliant early idea called the stored program concept.

The Stored Program Concept [2/2]

- Formulated independently by several scientists and engineers in the 1930s.
- The stored program concept is still considered the most profound invention in, if not the very foundation of, modern computer science.
The Stored Program Concept: Like many scientific breakthroughs, the basic idea is simple

- The computer is based on a **fixed hardware** platform capable of executing a fixed repertoire of simple instructions
- At the same time, these instructions can be **combined like building blocks**, yielding **arbitrarily** sophisticated programs

Moreover, the logic of these programs is not embedded in the hardware

- As was customary in mechanical computers predating 1930
- Instead, the program’s code is **temporarily stored** in the computer’s memory, like data
  - Becoming what is known as **software**
Since the computer’s operation manifests itself to the user through the currently executing software …

- The same hardware platform can be made to **behave completely differently** each time it is loaded with a different program

---

**The stored program concept in models of computing**

- The stored program concept is the key element of both abstract and practical computer models
  - Notably the Turing machine (1936) and the von Neumann machine (1945)
- The Turing machine is an **abstract artifact** describing a deceptively simple computer
  - Is used mainly in theoretical computer science for analyzing the logical foundations of computation
- In contrast, the **von Neumann** machine is a **practical** model that informs the construction of almost all computer platforms today
The von Neumann architecture

- The von Neumann architecture is based on a Central Processing Unit (CPU)
  - Interacting with a memory device
  - Receiving data from some input device, and
  - Emitting data to some output device

- At the heart of this architecture lies the stored program concept:
  - The computer’s memory stores not only the data that the computer manipulates
  - But also, the instructions that tell the computer what to do

Make everything as simple as possible, but not simpler.
—Albert Einstein (1879–1955)
A generic van Neumann computer architecture

The computer’s Memory can be discussed from both physical and logical perspectives

- Physically, the memory is a **linear sequence of addressable bytes**, each having a unique address and a value.

- Logically, this address space serves two purposes: **storing data and storing instructions**.
  - Both the “instruction words” and the “data words” are implemented exactly the same way—as sequences of bits.
Instruction Memory

Before a high-level program can be executed on a target computer, it must first be translated into the machine language of the target computer.

Each high-level statement is translated into one or more low-level instructions.
- Written as binary values to a file called the binary, or executable, version of the program.

Before running a program:
- First load its binary version from a mass storage device, and
- Serialize its instructions into the computer’s instruction memory.

Instruction Memory

From the pure focus of computer architecture, how a program is loaded into the computer’s memory is considered an external issue.

What’s important is that when the CPU is called upon to execute a program:
- The program’s code will already reside in the computer’s memory.
Data Memory

- High-level programs are designed to manipulate abstract artifacts like variables, arrays, and objects.
- Yet at the hardware level, these data abstractions are realized by binary values stored in memory.
- In particular, following translation to machine language:
  - Abstract array processing and get/set operations on objects are reduced to reading and writing selected locations in memory.
The CPU is the centerpiece of the computer’s architecture

- The Central Processing Unit (CPU) is in charge of executing the instructions of the currently running program.
- Each instruction tells the CPU which computation to perform, which registers to access, and which instruction to fetch and execute next.
- The CPU executes these tasks using three main elements:
  - An Arithmetic Logic Unit (ALU)
  - A set of registers, and
  - A control unit

Arithmetic Logic Unit (ALU)

- The ALU chip is built to perform all the low-level arithmetic and logical operations featured by the computer.
- A typical ALU can add two given values, compute their bitwise And, compare them for equality, and so on ...
How much functionality should be packed into the ALU is a design decision

- In general, any function not supported by the ALU can be realized later, using system software running on top of the hardware platform.
- The trade-off is simple:
  - Hardware implementations are typically more efficient but result in more expensive hardware.
  - While software implementations are inexpensive and less efficient.

Why we use registers [1/2]

- While performing computations, the CPU is often required to store interim values temporarily.
- In theory, we could have stored these values in the RAM, but this would entail long-distance trips between the CPU and the RAM.
  - The CPU and the RAM are two separate chips.
Why we use registers

- Long distance trips between the CPU and RAM result in delays
- These delays would frustrate the CPU-resident ALU, which is an ultra-fast combinational calculator
- The result will be a condition known as starvation
  - Happens when a fast processor depends on a sluggish data store for supplying its inputs and consuming its outputs

To avert starvation and boost performance

- CPUs are equipped with a small set of high-speed (and relatively expensive) registers, acting as the processor’s immediate memory
- These registers serve various purposes:
  - Data registers store interim values
  - Address registers store values that are used to address the RAM
  - The program counter stores the address of the instruction that should be fetched and executed next
  - The instruction register stores the current instruction
Control

- A computer instruction is a structured package of agreed-upon micro-codes
  - Sequences of one or more bits that signal to different devices what to do
- Thus, before an instruction can be executed, it must first be decoded into its micro-codes
- Next, each micro-code is routed to its designated hardware device (ALU, registers, memory) within the CPU
  - Where it tells the device how to partake in the overall instruction execution
Fetch-Execute

- In each step (cycle) of the program’s execution:
  - The CPU fetches a binary machine instruction from the instruction memory,
  - Decodes it, and
  - Executes it

- As a side effect of the instruction’s execution, the CPU also **figures out which instruction** to fetch and execute next

- This repetitive process is sometimes referred to as the **fetch-execute cycle**

CPU trends

- Multiprocessor systems with multiple CPUs debuted in the 1980s to get higher performance than could be achieved with a single CPU

- As it turns out, though, it’s not that easy

- Dividing up a single program so that it can be parallelized to make use of multiple CPUs is an **unsolved problem in the general case**
  - Although it works well for several classes of problems
The contents of this slide-set are based on the following references

