Week 15

Announcements

- Grades are up to date (except for HW 10)
- ADV8, ADV9, ADV10 submissions will be accepted for full credit until April 21

Lecture 11: Hardware pragmatics

Overview

- Intro
  - What are computers and what do they do (from a non-376 point of view)
  - Instruction set architecture (ISA)
- Journey from 270 to 370 to the real world
  - What is a cycle?
  - Single-cycle to pipelining
  - Doing better?
  - Moore and Dennard
- Contextualizing the CPU
  - Memory
  - Storage
  - Peripherals
  - Chipsets
  - System-on-chips (SoC)
- Buying and building computers
  - Navigating the lingo
  - Understanding your needs
- Answering your questions

Intro

What are computers and what do they do?

- They...

What are computers and what do they do?

- They...compute

What are computers and what do they do?

- They...compute
- They're calculators with extra steps

What are computers and what do they do?

- They...compute
- They're calculators with extra steps
  - (The extra steps being *control*: they have ways to figure out what operation to do next)

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them
  - "An contract between the hardware and the software"

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them
  - "An contract between the hardware and the software"
  - Hardware is the implementation, software is the thing using it

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them
  - "An contract between the hardware and the software"
  - Hardware is the implementation, software is the thing using it
- The ISA will specify what instructions it has (e.g. `add`, `udiv`, `mov`, `cmpxchg`) and what they do, what "registers" it has (e.g. `rax`, `rbx`), and how instructions are *encoded*

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them
  - "An contract between the hardware and the software"
  - Hardware is the implementation, software is the thing using it
- The ISA will specify what instructions it has (e.g. `add`, `udiv`, `mov`, `cmpxchg`) and what they do, what "registers" it has (e.g. `rax`, `rbx`), and how instructions are *encoded*
  - Most computers don't understand English, but we can easily design them to understand 1s and 0s (high and low voltages)

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
    - i.e. what operations can it do and how do you get it to do them
    - "An contract between the hardware and the software"
    - Hardware is the implementation, software is the thing using it
- The ISA will specify what instructions it has (e.g. `add`, `udiv`, `mov`, `cmpxchg`) and what they do, what "registers" it has (e.g. `rax`, `rbx`), and how instructions are *encoded*
    - Most computers don't understand English, but we can easily design them to understand 1s and 0s (high and low voltages)
    - We can assign patterns of 1s and 0s to different instructions

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them
  - "An contract between the hardware and the software"
  - Hardware is the implementation, software is the thing using it
- The ISA will specify what instructions it has (e.g. `add`, `udiv`, `mov`, `cmpxchg`) and what they do, what "registers" it has (e.g. `rax`, `rbx`), and how instructions are *encoded*
  - Most computers don't understand English, but we can easily design them to understand 1s and 0s (high and low voltages)
  - We can assign patterns of 1s and 0s to different instructions
- From here we get assembly languages since writing English is better than 1s and 0s

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them
  - "An contract between the hardware and the software"
  - Hardware is the implementation, software is the thing using it
- The ISA will specify what instructions it has (e.g. `add`, `udiv`, `mov`, `cmpxchg`) and what they do, what "registers" it has (e.g. `rax`, `rbx`), and how instructions are *encoded*
  - Most computers don't understand English, but we can easily design them to understand 1s and 0s (high and low voltages)
  - We can assign patterns of 1s and 0s to different instructions
- From here we get assembly languages since writing English is better than 1s and 0s
- Examples of ISAs: x86-64, ARMv8, RISC-V

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them
  - "An contract between the hardware and the software"
  - Hardware is the implementation, software is the thing using it
- The ISA will specify what instructions it has (e.g. `add`, `udiv`, `mov`, `cmpxchg`) and what they do, what "registers" it has (e.g. `rax`, `rbx`), and how instructions are *encoded*
  - Most computers don't understand English, but we can easily design them to understand 1s and 0s (high and low voltages)
  - We can assign patterns of 1s and 0s to different instructions
- From here we get assembly languages since writing English is better than 1s and 0s
- Examples of ISAs: x86-64, ARMv8, RISC-V
  - Some other ones: PowerPC, IA-64, SPARC, MIPS, AVR

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them
  - "An contract between the hardware and the software"
  - Hardware is the implementation, software is the thing using it
- The ISA will specify what instructions it has (e.g. `add`, `udiv`, `mov`, `cmpxchg`) and what they do, what "registers" it has (e.g. `rax`, `rbx`), and how instructions are *encoded*
  - Most computers don't understand English, but we can easily design them to understand 1s and 0s (high and low voltages)
  - We can assign patterns of 1s and 0s to different instructions
- From here we get assembly languages since writing English is better than 1s and 0s
- Examples of ISAs: x86-64, ARMv8, RISC-V
  - Some other ones: PowerPC, IA-64, SPARC, MIPS, AVR
- Implemented in the form of microarchitectures for a CPU (central processing unit)

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them
  - "An contract between the hardware and the software"
  - Hardware is the implementation, software is the thing using it
- The ISA will specify what instructions it has (e.g. `add`, `udiv`, `mov`, `cmpxchg`) and what they do, what "registers" it has (e.g. `rax`, `rbx`), and how instructions are *encoded*
  - Most computers don't understand English, but we can easily design them to understand 1s and 0s (high and low voltages)
  - We can assign patterns of 1s and 0s to different instructions
- From here we get assembly languages since writing English is better than 1s and 0s
- Examples of ISAs: x86-64, ARMv8, RISC-V
  - Some other ones: PowerPC, IA-64, SPARC, MIPS, AVR
- Implemented in the form of microarchitectures for a CPU (central processing unit)
  - Things like performance or exact implementation not specified by the ISA is figured out at this level

Instruction set architecture (ISA)

- A specification for what the computer is capable of and how to use it
  - i.e. what operations can it do and how do you get it to do them
  - "An contract between the hardware and the software"
  - Hardware is the implementation, software is the thing using it
- The ISA will specify what instructions it has (e.g. `add`, `udiv`, `mov`, `cmpxchg`) and what they do, what "registers" it has (e.g. `rax`, `rbx`), and how instructions are *encoded*
  - Most computers don't understand English, but we can easily design them to understand 1s and 0s (high and low voltages)
  - We can assign patterns of 1s and 0s to different instructions
- From here we get assembly languages since writing English is better than 1s and 0s
- Examples of ISAs: x86-64, ARMv8, RISC-V
  - Some other ones: PowerPC, IA-64, SPARC, MIPS, AVR
- Implemented in the form of microarchitectures for a CPU (central processing unit)
  - Things like performance or exact implementation not specified by the ISA is figured out at this level
  - Examples: Intel Haswell, AMD Zen, ARM Cortex-A57

Aside: CISC and RISC

- CISC: Complex Instruction Set Computer
  - "Complex" instructions that bundle multiple different operations together

Aside: CISC and RISC

- CISC: Complex Instruction Set Computer
  - "Complex" instructions that bundle multiple different operations together
  - You can have an instruction that gets something from memory, increments it, and stores it back into memory

Aside: CISC and RISC

- CISC: Complex Instruction Set Computer
  - "Complex" instructions that bundle multiple different operations together
  - You can have an instruction that gets something from memory, increments it, and stores it back into memory
- RISC: Reduced Instruction Set Computer

Aside: CISC and RISC

- CISC: Complex Instruction Set Computer
  - "Complex" instructions that bundle multiple different operations together
  - You can have an instruction that gets something from memory, increments it, and stores it back into memory
- RISC: Reduced Instruction Set Computer
  - "Simple" instructions that map to low level operations like doing a single add or a single load

Aside: CISC and RISC

- CISC: Complex Instruction Set Computer
  - "Complex" instructions that bundle multiple different operations together
  - You can have an instruction that gets something from memory, increments it, and stores it back into memory
- RISC: Reduced Instruction Set Computer
  - "Simple" instructions that map to low level operations like doing a single add or a single load
  - In contrast: you have 3 instructions: one to load from memory, one to increment, and one to store into memory

Aside: CISC and RISC

- CISC: Complex Instruction Set Computer
  - "Complex" instructions that bundle multiple different operations together
  - You can have an instruction that gets something from memory, increments it, and stores it back into memory
- RISC: Reduced Instruction Set Computer
  - "Simple" instructions that map to low level operations like doing a single add or a single load
  - In contrast: you have 3 instructions: one to load from memory, one to increment, and one to store into memory
- Lines are blurry, nothing is truly completely one or the other
  - x86 is a stereotypical CISC ISA
  - ARM is a stereotypical RISC ISA
  - Under the hood modern x86 implementations turn these x86 instructions into internal RISC-like operations

Journey from 270 to 370 to the real world

(basically glazing over a third of 370)

What is this cycle thing (and a bit of 270)

- From the circuit and logic level, things take time to do

What is this cycle thing (and a bit of 270)

- From the circuit and logic level, things take time to do
- How do you know when one operation starts and one ends?

What is this cycle thing (and a bit of 270)

- From the circuit and logic level, things take time to do
- How do you know when one operation starts and one ends?
- Add something to synchronize against: a **clock**

Setting the stage

- Our CPU will be interacting with a nebulous "memory" thing where instructions and data live
- Program counter (PC): our location in memory to get an instruction
- Registers: temporary hold space for operands and results for operations (e.g. to store the 2 and 5 for 2+5, and to store the resulting 7)

Single-cycle to pipeline

Single-cycle architecture

- Get an instruction, figure out what it does, do its operation, (accesses memory), store it, all in one cycle
- Pitfall:

Single-cycle to pipeline

Single-cycle architecture

- Get an instruction, figure out what it does, do its operation, (accesses memory), store it, all in one cycle
- Pitfall:
  - What if one instruction takes wayyyy longer to do than the rest?

Single-cycle to pipeline

Single-cycle architecture

- Get an instruction, figure out what it does, do its operation, (accesses memory), store it, all in one cycle
- Pitfall:
  - What if one instruction takes wayyyy longer to do than the rest?
  - Clock frequency is tied to the slowest instruction

Single-cycle to pipeline

Single-cycle architecture

- Get an instruction, figure out what it does, do its operation, (accesses memory), store it, all in one cycle
- Pitfall:
  - What if one instruction takes wayyyy longer to do than the rest?
  - Clock frequency is tied to the slowest instruction
  - Even if one instruction takes only 1 ns to do but another takes 100 ns to do, since we only have one clock* but everything has to fit in one cycle, we're stuck with a clock period of 100 ns

Single-cycle to pipeline

Single-cycle architecture

- Get an instruction, figure out what it does, do its operation, (accesses memory), store it, all in one cycle
- Pitfall:
  - What if one instruction takes wayyyy longer to do than the rest?
  - Clock frequency is tied to the slowest instruction
  - Even if one instruction takes only 1 ns to do but another takes 100 ns to do, since we only have one clock* but everything has to fit in one cycle, we're stuck with a clock period of 100 ns
  - We could've executed 100 of the short instruction in that one cycle!

Multi-cycle architecture

- Split processing an instruction into multiple steps, each step with a cycle

Multi-cycle architecture

- Split processing an instruction into multiple steps, each step with a cycle
- Get an instruction (fetch), figure out what it does (decode), do its operation (execute), potentially access memory (memory), store it (write-back)

Multi-cycle architecture

- Split processing an instruction into multiple steps, each step with a cycle
- Get an instruction (fetch), figure out what it does (decode), do its operation (execute), potentially access memory (memory), store it (write-back)
    - Known as the "instruction cycle"

Multi-cycle architecture

- Split processing an instruction into multiple steps, each step with a cycle
- Get an instruction (fetch), figure out what it does (decode), do its operation (execute), potentially access memory (memory), store it (write-back)
  - Known as the "instruction cycle"
  - Maybe certain instructions will need to take more cycles to execute
  - Example: addition only needs 1 cycle for execution, multiplication needs 8 execution, but multiplication needing more cycles doesn't bottleneck overall clock frequency

Multi-cycle architecture

- Split processing an instruction into multiple steps, each step with a cycle
- Get an instruction (fetch), figure out what it does (decode), do its operation (execute), potentially access memory (memory), store it (write-back)
  - Known as the "instruction cycle"
  - Maybe certain instructions will need to take more cycles to execute
  - Example: addition only needs 1 cycle for execution, multiplication needs 8 execution, but multiplication needing more cycles doesn't bottleneck overall clock frequency
- Pitfall:

Multi-cycle architecture

- Split processing an instruction into multiple steps, each step with a cycle
- Get an instruction (fetch), figure out what it does (decode), do its operation (execute), potentially access memory (memory), store it (write-back)
  - Known as the "instruction cycle"
  - Maybe certain instructions will need to take more cycles to execute
  - Example: addition only needs 1 cycle for execution, multiplication needs 8 execution, but multiplication needing more cycles doesn't bottleneck overall clock frequency
- Pitfall:
  - Seems wasteful to have hardware to fetch just sitting there while you're executing...

Pipelined architecture

- Make full use of the hardware for each part of the instruction cycle

Pipelined architecture

- Make full use of the hardware for each part of the instruction cycle
- Make the instruction cycle like an assembly line
  - Fetch works on i0, Decode works on i1, Execute works on i2, Memory works on i3, Write-back works on i4

Pipelined architecture

- Make full use of the hardware for each part of the instruction cycle
- Make the instruction cycle like an assembly line
  - Fetch works on i0, Decode works on i1, Execute works on i2, Memory works on i3, Write-back works on i4
  - Each cycle we're popping of an instruction off the end of the pipeline!

Pipelined architecture

- Make full use of the hardware for each part of the instruction cycle
- Make the instruction cycle like an assembly line
  - Fetch works on i0, Decode works on i1, Execute works on i2, Memory works on i3, Write-back works on i4
  - Each cycle we're popping of an instruction off the end of the pipeline!
- Pitfall:

Pipelined architecture

- Make full use of the hardware for each part of the instruction cycle
- Make the instruction cycle like an assembly line
  - Fetch works on i0, Decode works on i1, Execute works on i2, Memory works on i3, Write-back works on i4
  - Each cycle we're popping of an instruction off the end of the pipeline!
- Pitfall:
  - What if we have dependencies between instructions? E.g. the result of i3 is used by i2

Pipelined architecture

- Make full use of the hardware for each part of the instruction cycle
- Make the instruction cycle like an assembly line
  - Fetch works on i0, Decode works on i1, Execute works on i2, Memory works on i3, Write-back works on i4
  - Each cycle we're popping of an instruction off the end of the pipeline!
- Pitfall:
  - What if we have dependencies between instructions? E.g. the result of i3 is used by i2
  - What if we have a conditional branch? What happens to i0 and i1?

Pipelined architecture

- Make full use of the hardware for each part of the instruction cycle
- Make the instruction cycle like an assembly line
  - Fetch works on i0, Decode works on i1, Execute works on i2, Memory works on i3, Write-back works on i4
  - Each cycle we're popping of an instruction off the end of the pipeline!
- Pitfall:
  - What if we have dependencies between instructions? E.g. the result of i3 is used by i2
  - What if we have a conditional branch? What happens to i0 and i1?
  - These are known as "hazards" and impact performance

Pipelined architecture

- Make full use of the hardware for each part of the instruction cycle
- Make the instruction cycle like an assembly line
  - Fetch works on i0, Decode works on i1, Execute works on i2, Memory works on i3, Write-back works on i4
  - Each cycle we're popping of an instruction off the end of the pipeline!
- Pitfall:
  - What if we have dependencies between instructions? E.g. the result of i3 is used by i2
  - What if we have a conditional branch? What happens to i0 and i1?
  - These are known as "hazards" and impact performance
  - The cop-out method is to just stop the pipeline and wait for all of these to be resolved

Doing better

- Within the pipelineing framework we can add bypasses to have the result of execute or memory be used immediately

Doing better

- Within the pipelineing framework we can add bypasses to have the result of execute or memory be used immediately
- We can cleverly program so that these "hazards" don't happen

Doing better

- Within the pipelineing framework we can add bypasses to have the result of execute or memory be used immediately
- We can cleverly program so that these "hazards" don't happen
  - Known as "static scheduling"

Doing better

- Within the pipelineing framework we can add bypasses to have the result of execute or memory be used immediately
- We can cleverly program so that these "hazards" don't happen
  - Known as "static scheduling"
  - Interleave non-dependent instructions between dependent instructions

Doing better

- Within the pipelineing framework we can add bypasses to have the result of execute or memory be used immediately
- We can cleverly program so that these "hazards" don't happen
  - Known as "static scheduling"
  - Interleave non-dependent instructions between dependent instructions
  - Depends on a microarchitecture: what if you run the program on a different one?

Out-of-order execution (EECS 470)

- The "dynamic scheduling" counterpart

Out-of-order execution (EECS 470)

- The "dynamic scheduling" counterpart
- Schedule when instructions run based on availability of a resource and if the data is available

Out-of-order execution (EECS 470)

- The "dynamic scheduling" counterpart
- Schedule when instructions run based on availability of a resource and if the data is available
- Have the sense of different functional units that do different jobs (e.g. arithmetic and logic, memory access)

Out-of-order execution (EECS 470)

- The "dynamic scheduling" counterpart
- Schedule when instructions run based on availability of a resource and if the data is available
- Have the sense of different functional units that do different jobs (e.g. arithmetic and logic, memory access)
- Maybe we have an arithmetic instruction waiting on a long-running load, and some other non-dependent arithmetic instruction after it

Out-of-order execution (EECS 470)

- The "dynamic scheduling" counterpart
- Schedule when instructions run based on availability of a resource and if the data is available
- Have the sense of different functional units that do different jobs (e.g. arithmetic and logic, memory access)
- Maybe we have an arithmetic instruction waiting on a long-running load, and some other non-dependent arithmetic instruction after it
- We can run that non-dependent instruction since there's no dependency and the arithmetic unit is being unused!

Out-of-order execution (EECS 470)

- The "dynamic scheduling" counterpart
- Schedule when instructions run based on availability of a resource and if the data is available
- Have the sense of different functional units that do different jobs (e.g. arithmetic and logic, memory access)
- Maybe we have an arithmetic instruction waiting on a long-running load, and some other non-dependent arithmetic instruction after it
- We can run that non-dependent instruction since there's no dependency and the arithmetic unit is being unused!
- Handling this requires much more complex control hardware...

Superscalar

- Add more hardware so we can have two pipelines going!

Superscalar

- Add more hardware so we can have two pipelines going!
    - Fetch 2, decode 2, execute 2...

Superscalar

- Add more hardware so we can have two pipelines going!
  - Fetch 2, decode 2, execute 2...
  - The number of pipelines hardware are known as "ways"
  - 2-way superscalar, 4-way superscalar etc.

Superscalar

- Add more hardware so we can have two pipelines going!
  - Fetch 2, decode 2, execute 2...
  - The number of pipelines hardware are known as "ways"
  - 2-way superscalar, 4-way superscalar etc.
- Hazards only get more painful...

Superscalar

- Add more hardware so we can have two pipelines going!
  - Fetch 2, decode 2, execute 2...
  - The number of pipelines hardware are known as "ways"
  - 2-way superscalar, 4-way superscalar etc.
- Hazards only get more painful...
- But we can marry it with OoO for a superscalar out-of-order CPU!

Superscalar

- Add more hardware so we can have two pipelines going!
  - Fetch 2, decode 2, execute 2...
  - The number of pipelines hardware are known as "ways"
  - 2-way superscalar, 4-way superscalar etc.
- Hazards only get more painful...
- But we can marry it with OoO for a superscalar out-of-order CPU!
  - Thus grows our hardware complexity

Aside: Moore and Dennard

- You may have heard of this Moore's law thing

Aside: Moore and Dennard

- You may have heard of this Moore's law thing
  - Pattern where the number of transistors per unit area doubles every two years

Aside: Moore and Dennard

- You may have heard of this Moore's law thing
  - Pattern where the number of transistors per unit area doubles every two years
  - More transistors = more stuff we can do for the same amount of area

Aside: Moore and Dennard

- You may have heard of this Moore's law thing
  - Pattern where the number of transistors per unit area doubles every two years
  - More transistors = more stuff we can do for the same amount of area
- The unsung hero of yesteryear was Dennard scaling

Aside: Moore and Dennard

- You may have heard of this Moore's law thing
  - Pattern where the number of transistors per unit area doubles every two years
  - More transistors = more stuff we can do for the same amount of area
- The unsung hero of yesteryear was Dennard scaling
  - Power density stayed constant

Aside: Moore and Dennard

- You may have heard of this Moore's law thing
  - Pattern where the number of transistors per unit area doubles every two years
  - More transistors = more stuff we can do for the same amount of area
- The unsung hero of yesteryear was Dennard scaling
  - Power density stayed constant
  - As transistors got smaller (so we could fit more), they also used less power per transistor and could operate faster

Aside: Moore and Dennard

- You may have heard of this Moore's law thing
  - Pattern where the number of transistors per unit area doubles every two years
  - More transistors = more stuff we can do for the same amount of area
- The unsung hero of yesteryear was Dennard scaling
  - Power density stayed constant
  - As transistors got smaller (so we could fit more), they also used less power per transistor and could operate faster
  - This let us increase clock frequencies as Moore's law went on while keeping the same power consumption

Aside: Moore and Dennard

- You may have heard of this Moore's law thing
  - Pattern where the number of transistors per unit area doubles every two years
  - More transistors = more stuff we can do for the same amount of area
- The unsung hero of yesteryear was Dennard scaling
  - Power density stayed constant
  - As transistors got smaller (so we could fit more), they also used less power per transistor and could operate faster
  - This let us increase clock frequencies as Moore's law went on while keeping the same power consumption
- Dennard scaling fell apart around 2005: since then we've been hovering around 1-4 GHz

Aside: Moore and Dennard

- You may have heard of this Moore's law thing
    - Pattern where the number of transistors per unit area doubles every two years
    - More transistors = more stuff we can do for the same amount of area
- The unsung hero of yesteryear was Dennard scaling
    - Power density stayed constant
    - As transistors got smaller (so we could fit more), they also used less power per transistor and could operate faster
    - This let us increase clock frequencies as Moore's law went on while keeping the same power consumption
- Dennard scaling fell apart around 2005: since then we've been hovering around 1-4 GHz
- Who will save us now?

Multicore

- If we can't clock faster, let's have two CPU cores with these extra transistors!
  - (and reduce the overall clock frequency so we don't draw too much power)
  - A "core" is encompasses one of those pipeline things (including the superscalar's multiple)

Multicore

- If we can't clock faster, let's have two CPU cores with these extra transistors!
  - (and reduce the overall clock frequency so we don't draw too much power)
  - A "core" is encompasses one of those pipeline things (including the superscalar's multiple)
- You're probably not running only one program at once: Chrome and Vim have nothing to do with each other

Multicore

- If we can't clock faster, let's have two CPU cores with these extra transistors!
  - (and reduce the overall clock frequency so we don't draw too much power)
  - A "core" is encompasses one of those pipeline things (including the superscalar's multiple)
- You're probably not running only one program at once: Chrome and Vim have nothing to do with each other
- These are different threads: different *contexts* of execution that have their own "instruction stream" and register values
  - (the OS handles the nasty business of switching between them: that's 482 stuff though)

Multicore

- If we can't clock faster, let's have two CPU cores with these extra transistors!
  - (and reduce the overall clock frequency so we don't draw too much power)
  - A "core" is encompasses one of those pipeline things (including the superscalar's multiple)
- You're probably not running only one program at once: Chrome and Vim have nothing to do with each other
- These are different threads: different *contexts* of execution that have their own "instruction stream" and register values
  - (the OS handles the nasty business of switching between them: that's 482 stuff though)
- Since these are different contexts, their instructions don't have any dependency on each other

Multicore

- If we can't clock faster, let's have two CPU cores with these extra transistors!
  - (and reduce the overall clock frequency so we don't draw too much power)
  - A "core" is encompasses one of those pipeline things (including the superscalar's multiple)
- You're probably not running only one program at once: Chrome and Vim have nothing to do with each other
- These are different threads: different *contexts* of execution that have their own "instruction stream" and register values
  - (the OS handles the nasty business of switching between them: that's 482 stuff though)
- Since these are different contexts, their instructions don't have any dependency on each other
- We can just to toss each of these threads onto separate cores

Multicore

- If we can't clock faster, let's have two CPU cores with these extra transistors!
  - (and reduce the overall clock frequency so we don't draw too much power)
  - A "core" is encompasses one of those pipeline things (including the superscalar's multiple)
- You're probably not running only one program at once: Chrome and Vim have nothing to do with each other
- These are different threads: different *contexts* of execution that have their own "instruction stream" and register values
  - (the OS handles the nasty business of switching between them: that's 482 stuff though)
- Since these are different contexts, their instructions don't have any dependency on each other
- We can just to toss each of these threads onto separate cores
- Problems arise when it comes to memory however:

Multicore

- If we can't clock faster, let's have two CPU cores with these extra transistors!
    - (and reduce the overall clock frequency so we don't draw too much power)
    - A "core" is encompasses one of those pipeline things (including the superscalar's multiple)
- You're probably not running only one program at once: Chrome and Vim have nothing to do with each other
- These are different threads: different *contexts* of execution that have their own "instruction stream" and register values
    - (the OS handles the nasty business of switching between them: that's 482 stuff though)
- Since these are different contexts, their instructions don't have any dependency on each other
- We can just to toss each of these threads onto separate cores
- Problems arise when it comes to memory however:
    - If you have a "cache", what happens when one core writes to its cache and another core reads from the same location? (cache coherency)
    - If two cores access two memory locations, who writes/reads first? (memory consistency)
    - Fortunately, smart people have designed solutions for this

Multicore

- If we can't clock faster, let's have two CPU cores with these extra transistors!
  - (and reduce the overall clock frequency so we don't draw too much power)
  - A "core" is encompasses one of those pipeline things (including the superscalar's multiple)
- You're probably not running only one program at once: Chrome and Vim have nothing to do with each other
- These are different threads: different *contexts* of execution that have their own "instruction stream" and register values
  - (the OS handles the nasty business of switching between them: that's 482 stuff though)
- Since these are different contexts, their instructions don't have any dependency on each other
- We can just to toss each of these threads onto separate cores
- Problems arise when it comes to memory however:
  - If you have a "cache", what happens when one core writes to its cache and another core reads from the same location? (cache coherency)
  - If two cores access two memory locations, who writes/reads first? (memory consistency)
  - Fortunately, smart people have designed solutions for this
- (My group did a multicore *n*-way superscalar OoO processor for EECS 470)

Multithreading

- On a superscalar CPU we're going to have multiple sets of hardware, like multiple ALUs
- What if a thread can't use them all?

Multithreading

- On a superscalar CPU we're going to have multiple sets of hardware, like multiple ALUs
- What if a thread can't use them all?
- We can have a second thread! (if we pay the cost of supporting another context)
- These are known as "hardware threads"

Multithreading

- On a superscalar CPU we're going to have multiple sets of hardware, like multiple ALUs
- What if a thread can't use them all?
- We can have a second thread! (if we pay the cost of supporting another context)
- These are known as "hardware threads"
  - Software threads are handled by the OS: you can have thousands of threads active, but only a handful actually get to use the CPU at any given time

Multithreading

- On a superscalar CPU we're going to have multiple sets of hardware, like multiple ALUs
- What if a thread can't use them all?
- We can have a second thread! (if we pay the cost of supporting another context)
- These are known as "hardware threads"
  - Software threads are handled by the OS: you can have thousands of threads active, but only a handful actually get to use the CPU at any given time
- This method is to improve the utilization of the resources of the core

Multithreading

- On a superscalar CPU we're going to have multiple sets of hardware, like multiple ALUs
- What if a thread can't use them all?
- We can have a second thread! (if we pay the cost of supporting another context)
- These are known as "hardware threads"
  - Software threads are handled by the OS: you can have thousands of threads active, but only a handful actually get to use the CPU at any given time
- This method is to improve the utilization of the resources of the core
- Different types: fine-grained, coarse-grained, simultaneous multithreading (SMT)

Multithreading

- On a superscalar CPU we're going to have multiple sets of hardware, like multiple ALUs
- What if a thread can't use them all?
- We can have a second thread! (if we pay the cost of supporting another context)
- These are known as "hardware threads"
    - Software threads are handled by the OS: you can have thousands of threads active, but only a handful actually get to use the CPU at any given time
- This method is to improve the utilization of the resources of the core
- Different types: fine-grained, coarse-grained, simultaneous multithreading (SMT)
    - Fine-grained: switch thread after every cycle

Multithreading

- On a superscalar CPU we're going to have multiple sets of hardware, like multiple ALUs
- What if a thread can't use them all?
- We can have a second thread! (if we pay the cost of supporting another context)
- These are known as "hardware threads"
    - Software threads are handled by the OS: you can have thousands of threads active, but only a handful actually get to use the CPU at any given time
- This method is to improve the utilization of the resources of the core
- Different types: fine-grained, coarse-grained, simultaneous multithreading (SMT)
    - Fine-grained: switch thread after every cycle
    - Coarse-grained: switch thread when a long latency event occurs (cache miss, page fault)

Multithreading

- On a superscalar CPU we're going to have multiple sets of hardware, like multiple ALUs
- What if a thread can't use them all?
- We can have a second thread! (if we pay the cost of supporting another context)
- These are known as "hardware threads"
    - Software threads are handled by the OS: you can have thousands of threads active, but only a handful actually get to use the CPU at any given time
- This method is to improve the utilization of the resources of the core
- Different types: fine-grained, coarse-grained, simultaneous multithreading (SMT)
    - Fine-grained: switch thread after every cycle
    - Coarse-grained: switch thread when a long latency event occurs (cache miss, page fault)
    - SMT: multiple threads can have an instruction run at the same time

Multithreading

- On a superscalar CPU we're going to have multiple sets of hardware, like multiple ALUs
- What if a thread can't use them all?
- We can have a second thread! (if we pay the cost of supporting another context)
- These are known as "hardware threads"
  - Software threads are handled by the OS: you can have thousands of threads active, but only a handful actually get to use the CPU at any given time
- This method is to improve the utilization of the resources of the core
- Different types: fine-grained, coarse-grained, simultaneous multithreading (SMT)
  - Fine-grained: switch thread after every cycle
  - Coarse-grained: switch thread when a long latency event occurs (cache miss, page fault)
  - SMT: multiple threads can have an instruction run at the same time
- SMT is the one implemented nowadays, usually with 2 threads per core
  - Microarchitectural details differ

Multithreading

- On a superscalar CPU we're going to have multiple sets of hardware, like multiple ALUs
- What if a thread can't use them all?
- We can have a second thread! (if we pay the cost of supporting another context)
- These are known as "hardware threads"
    - Software threads are handled by the OS: you can have thousands of threads active, but only a handful actually get to use the CPU at any given time
- This method is to improve the utilization of the resources of the core
- Different types: fine-grained, coarse-grained, simultaneous multithreading (SMT)
    - Fine-grained: switch thread after every cycle
    - Coarse-grained: switch thread when a long latency event occurs (cache miss, page fault)
    - SMT: multiple threads can have an instruction run at the same time
- SMT is the one implemented nowadays, usually with 2 threads per core
    - Microarchitectural details differ
- Operating systems expose these threads as "logical cores" (as opposed to "physical cores")

SIMD, SIMT, and GPUs

- The stuff from before is what is known as SISD: "Single instruction, single data"

SIMD, SIMT, and GPUs

- The stuff from before is what is known as SISD: "Single instruction, single data"
- SIMD: "Single instruction, multiple data"
  - Do the same operation for multiple pieces of data
  - Skip out on the costs of having to do the instruction cycle for each piece of data
  - Implemented on normal CPUs as ISA extensions
  - Examples: Intex SSE and AVX-512, ARM NEON

SIMD, SIMT, and GPUs

- The stuff from before is what is known as SISD: "Single instruction, single data"
- SIMD: "Single instruction, multiple data"
  - Do the same operation for multiple pieces of data
  - Skip out on the costs of having to do the instruction cycle for each piece of data
  - Implemented on normal CPUs as ISA extensions
  - Examples: Intex SSE and AVX-512, ARM NEON
- SIMT: "Single instruction, multiple thread"
  - SIMD can't handle conditional branches
  - Like multithreading and SIMD had a baby
  - Conditional branches still hurt
  - Modern GPUs follow this pattern

SIMD, SIMT, and GPUs

- The stuff from before is what is known as SISD: "Single instruction, single data"
- SIMD: "Single instruction, multiple data"
  - Do the same operation for multiple pieces of data
  - Skip out on the costs of having to do the instruction cycle for each piece of data
  - Implemented on normal CPUs as ISA extensions
  - Examples: Intex SSE and AVX-512, ARM NEON
- SIMT: "Single instruction, multiple thread"
  - SIMD can't handle conditional branches
  - Like multithreading and SIMD had a baby
  - Conditional branches still hurt
  - Modern GPUs follow this pattern
- Modern GPUs serve as massively parallel computation units: you can envision them as a collection of very weak cores
- Their strength comes from sheer numbers: the workloads you put on them are highly parallelizable
  - E.g. halve the value of each pixel on the screen

Contextualizing the CPU

System

- CPUs don't exist in a vacuum: they exist in an interconnected computer *system*
- There are other things like memory, storage, and peripherals

Memory

- In this context, we're talking about "volatile" storage
- This serves as "fast" access storage for use while running

Memory

- In this context, we're talking about "volatile" storage
- This serves as "fast" access storage for use while running
- Otherwise known as RAM ("random access memory")
- Two flavors: SRAM and DRAM

Memory

- In this context, we're talking about "volatile" storage
- This serves as "fast" access storage for use while running
- Otherwise known as RAM ("random access memory")
- Two flavors: SRAM and DRAM
    - SRAM is faster but more expensive (less dense: around 6 transistors per bit)

Memory

- In this context, we're talking about "volatile" storage
- This serves as "fast" access storage for use while running
- Otherwise known as RAM ("random access memory")
- Two flavors: SRAM and DRAM
  - SRAM is faster but more expensive (less dense: around 6 transistors per bit)
  - DRAM is slower but much cheaper (1 transistor per bit)
  - DRAM also has to be manufactured on a separate chip due to a different manufacturing process

Memory

- In this context, we're talking about "volatile" storage
- This serves as "fast" access storage for use while running
- Otherwise known as RAM ("random access memory")
- Two flavors: SRAM and DRAM
  - SRAM is faster but more expensive (less dense: around 6 transistors per bit)
  - DRAM is slower but much cheaper (1 transistor per bit)
  - DRAM also has to be manufactured on a separate chip due to a different manufacturing process
- DRAM is often used as the bulk of memory, known as "main memory"

Memory

- In this context, we're talking about "volatile" storage
- This serves as "fast" access storage for use while running
- Otherwise known as RAM ("random access memory")
- Two flavors: SRAM and DRAM
  - SRAM is faster but more expensive (less dense: around 6 transistors per bit)
  - DRAM is slower but much cheaper (1 transistor per bit)
  - DRAM also has to be manufactured on a separate chip due to a different manufacturing process
- DRAM is often used as the bulk of memory, known as "main memory"
  - Since it's off chip, DRAM is actually kind of slow to use

Memory

- In this context, we're talking about "volatile" storage
- This serves as "fast" access storage for use while running
- Otherwise known as RAM ("random access memory")
- Two flavors: SRAM and DRAM
  - SRAM is faster but more expensive (less dense: around 6 transistors per bit)
  - DRAM is slower but much cheaper (1 transistor per bit)
  - DRAM also has to be manufactured on a separate chip due to a different manufacturing process
- DRAM is often used as the bulk of memory, known as "main memory"
  - Since it's off chip, DRAM is actually kind of slow to use
- SRAM is typically used on-chip by the core as "cache" to keep a fast access copy of DRAM contents

Memory

- In this context, we're talking about "volatile" storage
- This serves as "fast" access storage for use while running
- Otherwise known as RAM ("random access memory")
- Two flavors: SRAM and DRAM
  - SRAM is faster but more expensive (less dense: around 6 transistors per bit)
  - DRAM is slower but much cheaper (1 transistor per bit)
  - DRAM also has to be manufactured on a separate chip due to a different manufacturing process
- DRAM is often used as the bulk of memory, known as "main memory"
  - Since it's off chip, DRAM is actually kind of slow to use
- SRAM is typically used on-chip by the core as "cache" to keep a fast access copy of DRAM contents
  - Think about how multicore mucks this up

Storage

- In this context, we're talking about "non-volatile" storage
- We're in the age of hard disc drives (HDD) and solid state drives (SSD)

Storage

- In this context, we're talking about "non-volatile" storage
- We're in the age of hard disc drives (HDD) and solid state drives (SSD)
- HDDs use spinning magnetized discs to store information

Storage

- In this context, we're talking about "non-volatile" storage
- We're in the age of hard disc drives (HDD) and solid state drives (SSD)
- HDDs use spinning magnetized discs to store information
    - Slow: have to wait for the "read head" to move to the right "track" and wait for the disc to spin to the right "sector"
    - On the order of milliseconds

Storage

- In this context, we're talking about "non-volatile" storage
- We're in the age of hard disc drives (HDD) and solid state drives (SSD)
- HDDs use spinning magnetized discs to store information
  - Slow: have to wait for the "read head" to move to the right "track" and wait for the disc to spin to the right "sector"
  - On the order of milliseconds
  - Cheap

Storage

- In this context, we're talking about "non-volatile" storage
- We're in the age of hard disc drives (HDD) and solid state drives (SSD)
- HDDs use spinning magnetized discs to store information
  - Slow: have to wait for the "read head" to move to the right "track" and wait for the disc to spin to the right "sector"
  - On the order of milliseconds
  - Cheap
- SSDs are called "solid state" because there are no moving parts
  - Typically implemented as flash memory

Storage

- In this context, we're talking about "non-volatile" storage
- We're in the age of hard disc drives (HDD) and solid state drives (SSD)
- HDDs use spinning magnetized discs to store information
  - Slow: have to wait for the "read head" to move to the right "track" and wait for the disc to spin to the right "sector"
  - On the order of milliseconds
  - Cheap
- SSDs are called "solid state" because there are no moving parts
  - Typically implemented as flash memory
  - Fast! No moving parts with the magic of electrons
  - On the order of microseconds

Storage

- In this context, we're talking about "non-volatile" storage
- We're in the age of hard disc drives (HDD) and solid state drives (SSD)
- HDDs use spinning magnetized discs to store information
  - Slow: have to wait for the "read head" to move to the right "track" and wait for the disc to spin to the right "sector"
  - On the order of milliseconds
  - Cheap
- SSDs are called "solid state" because there are no moving parts
  - Typically implemented as flash memory
  - Fast! No moving parts with the magic of electrons
  - On the order of microseconds
  - Expensive :(
  - There's different "tiers", where they can sacrifice reliability for density while maintaining price (SLC, MLC, TLC)

Peripherals

- There's other peripherals as well
- Networking peripherals: Ethernet and Wifi controllers
- Accelerators: GPUs
- Input: Keyboards
- Output: Audio, display controllers
- USB

These together form a fully functioning computer system

- The most stereotypical is your desktop or laptop computer
- You have a **motherboard** connecting the components of a **chipset**

These together form a fully functioning computer system

- The most stereotypical is your desktop or laptop computer
- You have a **motherboard** connecting the components of a **chipset**
- The chipset specifies set of components and features for a computer system and controls the flow of data between them

These together form a fully functioning computer system

- The most stereotypical is your desktop or laptop computer
- You have a **motherboard** connecting the components of a **chipset**
- The chipset specifies set of components and features for a computer system and controls the flow of data between them
  - E.g. Intel Z170 supports LGA 1151 socket Intel CPUs, has four DIMM slots for DDR4 RAM etc.

These together form a fully functioning computer system

- The most stereotypical is your desktop or laptop computer
- You have a **motherboard** connecting the components of a **chipset**
- The chipset specifies set of components and features for a computer system and controls the flow of data between them
  - E.g. Intel Z170 supports LGA 1151 socket Intel CPUs, has four DIMM slots for DDR4 RAM etc.
- There's a bunch of little additional specialized chips doing their little jobs ("chip" "set")

These together form a fully functioning computer system

- The most stereotypical is your desktop or laptop computer
- You have a **motherboard** connecting the components of a **chipset**
- The chipset specifies set of components and features for a computer system and controls the flow of data between them
    - E.g. Intel Z170 supports LGA 1151 socket Intel CPUs, has four DIMM slots for DDR4 RAM etc.
- There's a bunch of little additional specialized chips doing their little jobs ("chip" "set")

But wait, there's more

- Enter...the system-on-chip (SoC)

These together form a fully functioning computer system

- The most stereotypical is your desktop or laptop computer
- You have a **motherboard** connecting the components of a **chipset**
- The chipset specifies set of components and features for a computer system and controls the flow of data between them
  - E.g. Intel Z170 supports LGA 1151 socket Intel CPUs, has four DIMM slots for DDR4 RAM etc.
- There's a bunch of little additional specialized chips doing their little jobs ("chip" "set")

But wait, there's more

- Enter...the system-on-chip (SoC)
- On a single given chip, put not just the CPU cores but also the memory and other peripherals!
  - Allows for better integration of components for optimization
  - Some SoCs implement their main memory as SRAM!

These together form a fully functioning computer system

- The most stereotypical is your desktop or laptop computer
- You have a **motherboard** connecting the components of a **chipset**
- The chipset specifies set of components and features for a computer system and controls the flow of data between them
  - E.g. Intel Z170 supports LGA 1151 socket Intel CPUs, has four DIMM slots for DDR4 RAM etc.
- There's a bunch of little additional specialized chips doing their little jobs ("chip" "set")

But wait, there's more

- Enter...the system-on-chip (SoC)
- On a single given chip, put not just the CPU cores but also the memory and other peripherals!
  - Allows for better integration of components for optimization
  - Some SoCs implement their main memory as SRAM!
- Desktop and laptop CPUs have gradually shifted over to look more like SoCs

Mobile SoCs

- Mobile phones use specially designed SoCs
- Pretty much all use the ARMv8 ISA for their cores
- Phone manufacturers have to pick out an SoC from vendors like Qualcomm and Samsung

Mobile SoCs

- Mobile phones use specially designed SoCs
- Pretty much all use the ARMv8 ISA for their cores
- Phone manufacturers have to pick out an SoC from vendors like Qualcomm and Samsung
  - (Apple is an exception as they design theirs in-house)

Mobile SoCs

- Mobile phones use specially designed SoCs
- Pretty much all use the ARMv8 ISA for their cores
- Phone manufacturers have to pick out an SoC from vendors like Qualcomm and Samsung
  - (Apple is an exception as they design theirs in-house)
- The SoC designers themselves figure out what they're going to put on the chip
  - Some opt for designing their own microarchitecture (Apple), some use a reference one from ARM

Mobile SoCs

- Mobile phones use specially designed SoCs
- Pretty much all use the ARMv8 ISA for their cores
- Phone manufacturers have to pick out an SoC from vendors like Qualcomm and Samsung
  - (Apple is an exception as they design theirs in-house)
- The SoC designers themselves figure out what they're going to put on the chip
  - Some opt for designing their own microarchitecture (Apple), some use a reference one from ARM
- One important aspect is power: mobile devices run off of a battery, so minimizing power consumption is important

Mobile SoCs

- Mobile phones use specially designed SoCs
- Pretty much all use the ARMv8 ISA for their cores
- Phone manufacturers have to pick out an SoC from vendors like Qualcomm and Samsung
  - (Apple is an exception as they design theirs in-house)
- The SoC designers themselves figure out what they're going to put on the chip
  - Some opt for designing their own microarchitecture (Apple), some use a reference one from ARM
- One important aspect is power: mobile devices run off of a battery, so minimizing power consumption is important
  - Still, you don't want to tank performance by using a low power microarchitecture

Mobile SoCs

- Mobile phones use specially designed SoCs
- Pretty much all use the ARMv8 ISA for their cores
- Phone manufacturers have to pick out an SoC from vendors like Qualcomm and Samsung
  - (Apple is an exception as they design theirs in-house)
- The SoC designers themselves figure out what they're going to put on the chip
  - Some opt for designing their own microarchitecture (Apple), some use a reference one from ARM
- One important aspect is power: mobile devices run off of a battery, so minimizing power consumption is important
  - Still, you don't want to tank performance by using a low power microarchitecture
- One solution for this is mixing big and small cores

Mobile SoCs

- Mobile phones use specially designed SoCs
- Pretty much all use the ARMv8 ISA for their cores
- Phone manufacturers have to pick out an SoC from vendors like Qualcomm and Samsung
  - (Apple is an exception as they design theirs in-house)
- The SoC designers themselves figure out what they're going to put on the chip
  - Some opt for designing their own microarchitecture (Apple), some use a reference one from ARM
- One important aspect is power: mobile devices run off of a battery, so minimizing power consumption is important
  - Still, you don't want to tank performance by using a low power microarchitecture
- One solution for this is mixing big and small cores
  - "Big" cores are powerful, and energy-hungry that take full advantage of the enhancements from before

Mobile SoCs

- Mobile phones use specially designed SoCs
- Pretty much all use the ARMv8 ISA for their cores
- Phone manufacturers have to pick out an SoC from vendors like Qualcomm and Samsung
  - (Apple is an exception as they design theirs in-house)
- The SoC designers themselves figure out what they're going to put on the chip
  - Some opt for designing their own microarchitecture (Apple), some use a reference one from ARM
- One important aspect is power: mobile devices run off of a battery, so minimizing power consumption is important
  - Still, you don't want to tank performance by using a low power microarchitecture
- One solution for this is mixing big and small cores
  - "Big" cores are powerful, and energy-hungry that take full advantage of the enhancements from before
  - "Small" cores are smaller, less powerful, and less energy-hungry by opting for simpler designs

Mobile SoCs

- Mobile phones use specially designed SoCs
- Pretty much all use the ARMv8 ISA for their cores
- Phone manufacturers have to pick out an SoC from vendors like Qualcomm and Samsung
  - (Apple is an exception as they design theirs in-house)
- The SoC designers themselves figure out what they're going to put on the chip
  - Some opt for designing their own microarchitecture (Apple), some use a reference one from ARM
- One important aspect is power: mobile devices run off of a battery, so minimizing power consumption is important
  - Still, you don't want to tank performance by using a low power microarchitecture
- One solution for this is mixing big and small cores
  - "Big" cores are powerful, and energy-hungry that take full advantage of the enhancements from before
  - "Small" cores are smaller, less powerful, and less energy-hungry by opting for simpler designs
- Not all programs need all that horsepower: perhaps during a low workload we can use only the small cores and turn off the big cores.

Buying and building computers

y tho

- You may be wondering why I decided to basically recap EECS 270, 370, 470, and a bit of 427

y tho

- You may be wondering why I decided to basically recap EECS 270, 370, 470, and a bit of 427
- My big idea is that computers are systems: things have to be balanced out for them to work properly

y tho

- You may be wondering why I decided to basically recap EECS 270, 370, 470, and a bit of 427
- My big idea is that computers are systems: things have to be balanced out for them to work properly
  - What good is a super fast CPU if you're being bottlenecked by a junk HDD?

y tho

- You may be wondering why I decided to basically recap EECS 270, 370, 470, and a bit of 427
- My big idea is that computers are systems: things have to be balanced out for them to work properly
    - What good is a super fast CPU if you're being bottlenecked by a junk HDD?
- Having a bit of understanding about each of the various parts can help you put together a well functioning computer

y tho

- You may be wondering why I decided to basically recap EECS 270, 370, 470, and a bit of 427
- My big idea is that computers are systems: things have to be balanced out for them to work properly
  - What good is a super fast CPU if you're being bottlenecked by a junk HDD?
- Having a bit of understanding about each of the various parts can help you put together a well functioning computer
- This has a very real world application for all of us (not just the CEs): buying a computer

Spec'ing a computer

- Ultimately, the specs of your computer will come down to your use: the highest end Macbook probably isn't needed if all you do is look at emails and watch Netflix
  - However, all of you do some sort of computer science, so the bottom of the barrel isn't necessarily helpful either

Spec'ing a computer

- Ultimately, the specs of your computer will come down to your use: the highest end Macbook probably isn't needed if all you do is look at emails and watch Netflix
    - However, all of you do some sort of computer science, so the bottom of the barrel isn't necessarily helpful either
- Also, keep in mind some sort of budget. If your budget is infinite, you might as well get the best thing.

Spec'ing a computer

- Ultimately, the specs of your computer will come down to your use: the highest end Macbook probably isn't needed if all you do is look at emails and watch Netflix
    - However, all of you do some sort of computer science, so the bottom of the barrel isn't necessarily helpful either
- Also, keep in mind some sort of budget. If your budget is infinite, you might as well get the best thing.
- First, I'm going to go some over general information. At the end I'll give my own personal preferences.

Laptop vs Desktop

- There's some things to keep in mind when deciding for either

Laptop vs Desktop

- There's some things to keep in mind when deciding for either
- Laptops are small and cramped: they don't dissipate heat as well
  - Clock frequencies tend to be lower for laptops
  - Tend towards the weaker versions of CPUs and GPUs
  - Premium for smaller form factor

Laptop vs Desktop

- There's some things to keep in mind when deciding for either
- Laptops are small and cramped: they don't dissipate heat as well
  - Clock frequencies tend to be lower for laptops
  - Tend towards the weaker versions of CPUs and GPUs
  - Premium for smaller form factor
- When building a desktop, keep in mind the power budget as well: each component has a "thermal design power" which is the highest power it was designed to operate at
  - A power supply to match/exceed the sum of the TDPs will be needed

To Apple or not to Apple

- This is a very personal decision

To Apple or not to Apple

- This is a very personal decision
- This depends on:
  - How invested you are in the Apple ecosystem
  - How much you need macOS: do you need to run certain applications that are only on macOS?
  - Do you want to develop applications for Apple products?
  - How much do you like having a native POSIX system (as opposed to Windows)?
  - How much do you like the Retina display?
  - Are you willing to pay a premium for an Apple product? (they do have nice design and build quality)

CPU

- The main contenders for laptop/desktop CPUs are Intel and AMD

CPU

- The main contenders for laptop/desktop CPUs are Intel and AMD
- AMD is particularly well known for their higher performance per dollar
  - They also offer more cores than equivalent-tier Intel processors

CPU

- The main contenders for laptop/desktop CPUs are Intel and AMD
- AMD is particularly well known for their higher performance per dollar
  - They also offer more cores than equivalent-tier Intel processors
- Intel has been known for having higher single-threaded performance (how much higher is debatable)
  - This will help out with applications that really need to push one or two threads

CPU

- The main contenders for laptop/desktop CPUs are Intel and AMD
- AMD is particularly well known for their higher performance per dollar
    - They also offer more cores than equivalent-tier Intel processors
- Intel has been known for having higher single-threaded performance (how much higher is debatable)
    - This will help out with applications that really need to push one or two threads
- SMT is called "Hyper-threading" by Intel
    - AMD Zen chips (Ryzen) also have SMT
    - Most places showing CPU specs will also have a core/thread count for the CPU

GPU

- The main contenders for **discrete** laptop/desktop GPUs are NVIDIA and AMD
- Intel also provides **integrated** GPUs on their CPU chips, while AMD provides theirs on their "APU" chips
  - Big caveat here: there are AMD processors for desktops builds don't have integrated GPUs!

GPU

- The main contenders for **discrete** laptop/desktop GPUs are NVIDIA and AMD
- Intel also provides **integrated** GPUs on their CPU chips, while AMD provides theirs on their "APU" chips
  - Big caveat here: there are AMD processors for desktops builds don't have integrated GPUs!
- Integrated GPUs are provided alongside a CPU and is there as the display controller as well as to provide okay-ish graphics support
  - If you don't play computer games, do GPGPU programming, or use any software that may use the GPU for acceleration (Photoshop, video editing, CAD), an integrated GPU is fine
  - As a side note, some AMD APUs are actually pretty decent for gaming
  - As a side side note, some newer Intel iGPUs are actually competitive with AMD APUs

GPU

- The main contenders for **discrete** laptop/desktop GPUs are NVIDIA and AMD
- Intel also provides **integrated** GPUs on their CPU chips, while AMD provides theirs on their "APU" chips
  - Big caveat here: there are AMD processors for desktops builds don't have integrated GPUs!
- Integrated GPUs are provided alongside a CPU and is there as the display controller as well as to provide okay-ish graphics support
  - If you don't play computer games, do GPGPU programming, or use any software that may use the GPU for acceleration (Photoshop, video editing, CAD), an integrated GPU is fine
  - As a side note, some AMD APUs are actually pretty decent for gaming
  - As a side side note, some newer Intel iGPUs are actually competitive with AMD APUs
- Discrete GPUs are separate GPUs optimized for performance
  - There's a vast (and confusing) lineup of them: there's gaming, professional visualization, and scientific compute variants
  - NVIDIA has the highest performance
  - AMD has better performance per dollar
  - CUDA is exclusive to NVIDIA GPUs: some software may be optimized for it or only support CUDA
  - NVIDIA has a lot of deep learning library support for their GPUs

GPU

- The main contenders for **discrete** laptop/desktop GPUs are NVIDIA and AMD
- Intel also provides **integrated** GPUs on their CPU chips, while AMD provides theirs on their "APU" chips
  - Big caveat here: there are AMD processors for desktops builds don't have integrated GPUs!
- Integrated GPUs are provided alongside a CPU and is there as the display controller as well as to provide okay-ish graphics support
  - If you don't play computer games, do GPGPU programming, or use any software that may use the GPU for acceleration (Photoshop, video editing, CAD), an integrated GPU is fine
  - As a side note, some AMD APUs are actually pretty decent for gaming
  - As a side side note, some newer Intel iGPUs are actually competitive with AMD APUs
- Discrete GPUs are separate GPUs optimized for performance
  - There's a vast (and confusing) lineup of them: there's gaming, professional visualization, and scientific compute variants
  - NVIDIA has the highest performance
  - AMD has better performance per dollar
  - CUDA is exclusive to NVIDIA GPUs: some software may be optimized for it or only support CUDA
  - NVIDIA has a lot of deep learning library support for their GPUs

RAM

- RAM speed isn't super critical for general purpose use
- The supposed performance improvement DDR4 has over DDR3 isn't really noticeable in the whole system
  - DDR4 vs DDR3 is more a matter of what motherboard/chipset/laptop you're getting

RAM

- RAM speed isn't super critical for general purpose use
- The supposed performance improvement DDR4 has over DDR3 isn't really noticeable in the whole system
  - DDR4 vs DDR3 is more a matter of what motherboard/chipset/laptop you're getting
- Nowadays 8 GB is a decent amount to have (dang web browsers)
  - 4 GB is fine for someone just doing light browsing/document editing/email writing

Storage

- The amount of storage is a super personal decision

Storage

- The amount of storage is a super personal decision
- SSD is overall better than HDD, just quite a bit more expensive

Storage

- The amount of storage is a super personal decision
- SSD is overall better than HDD, just quite a bit more expensive
- Laptop manufacturers put a premium on additional space...

Storage

- The amount of storage is a super personal decision
- SSD is overall better than HDD, just quite a bit more expensive
- Laptop manufacturers put a premium on additional space...
- If you end up with a system with both, HDDs are excellent for storing things like movies, music, and photos: they're relatively big and they don't require a lot of random access

Storage

- The amount of storage is a super personal decision
- SSD is overall better than HDD, just quite a bit more expensive
- Laptop manufacturers put a premium on additional space...
- If you end up with a system with both, HDDs are excellent for storing things like movies, music, and photos: they're relatively big and they don't require a lot of random access
- Performance metrics are read speed and write speeds

Storage

- The amount of storage is a super personal decision
- SSD is overall better than HDD, just quite a bit more expensive
- Laptop manufacturers put a premium on additional space...
- If you end up with a system with both, HDDs are excellent for storing things like movies, music, and photos: they're relatively big and they don't require a lot of random access
- Performance metrics are read speed and write speeds

SSDs

- Different "tiers": SLC, MLC, TLC; these are how many bits they stuff into a single cell
- SLC: "single-layer cell", most expensive, most reliable
- MLC: "multi-layer (2) cell" less expensive, less reliable
- TLC: "triple-layer cell" even less expensive; these are fairly common ones on the market
- Common form factors: 2.5" and m.2

Storage

- The amount of storage is a super personal decision
- SSD is overall better than HDD, just quite a bit more expensive
- Laptop manufacturers put a premium on additional space...
- If you end up with a system with both, HDDs are excellent for storing things like movies, music, and photos: they're relatively big and they don't require a lot of random access
- Performance metrics are read speed and write speeds

SSDs

- Different "tiers": SLC, MLC, TLC; these are how many bits they stuff into a single cell
- SLC: "single-layer cell", most expensive, most reliable
- MLC: "multi-layer (2) cell" less expensive, less reliable
- TLC: "triple-layer cell" even less expensive; these are fairly common ones on the market
- Common form factors: 2.5" and m.2

HDDs

- Their speeds determine how fast they are
- Generally 5400 RPM for laptops and 7200 RPM for desktops
- Different form factors: 2.5" (laptops) and 3.5" (desktops)

Common misconceptions

1. More cores = this one application will go faster
    - More cores only means more capability for running threads simultaneously
    - If you depend on an application that happens to be a single threaded application, more cores aren't going to help

Common misconceptions

1. More cores = this one application will go faster
   - More cores only means more capability for running threads simultaneously
   - If you depend on an application that happens to be a single threaded application, more cores aren't going to help
2. This CPU has a higher clock frequency than this other one: of course it's faster
   - GHz means more cycles a second, but it doesn't govern how much "work" goes into a cycle
   - The overall speed of your CPU depends on its microarchitecture: you can't compare GHz between different microarchitectures

Common misconceptions

1. More cores = this one application will go faster
    - More cores only means more capability for running threads simultaneously
    - If you depend on an application that happens to be a single threaded application, more cores aren't going to help
2. This CPU has a higher clock frequency than this other one: of course it's faster
    - GHz means more cycles a second, but it doesn't govern how much "work" goes into a cycle
    - The overall speed of your CPU depends on its microarchitecture: you can't compare GHz between different microarchitectures
3. More RAM = faster
    - It may feel frightening, but using 80% of your RAM isn't necessarily a bad thing
    - More RAM can help certain applications (i.e. the time-space tradeoff) but there's a certain point where there's more than enough to go around
    - The main problem that comes from RAM is if you don't have enough of it: when that happens your HDD/SSD is used to extend memory by swapping processes to your HDD/SSD

My preferences

Caveats

- My experiences are heavily driven by an engineering undergrad
  - CAD tools, scientific computing, and HDL synthesis

My preferences

Caveats

- My experiences are heavily driven by an engineering undergrad
  - CAD tools, scientific computing, and HDL synthesis

For a general purpose laptop for computer science and engineering related work I prefer:

- A decent enough CPU, nothing too fancy: something with 4 cores (preferably with SMT)
  - Intel i5 and i7, AMD Ryzen 5 and Ryzen 7 are around what I like

My preferences

Caveats

- My experiences are heavily driven by an engineering undergrad
    - CAD tools, scientific computing, and HDL synthesis

For a general purpose laptop for computer science and engineering related work I prefer:

- A decent enough CPU, nothing too fancy: something with 4 cores (preferably with SMT)
    - Intel i5 and i7, AMD Ryzen 5 and Ryzen 7 are around what I like
- At least 8 GB of RAM, preferably 16 GB
    - I open A LOT of browser tabs

My preferences

Caveats

- My experiences are heavily driven by an engineering undergrad
  - CAD tools, scientific computing, and HDL synthesis

For a general purpose laptop for computer science and engineering related work I prefer:

- A decent enough CPU, nothing too fancy: something with 4 cores (preferably with SMT)
  - Intel i5 and i7, AMD Ryzen 5 and Ryzen 7 are around what I like
- At least 8 GB of RAM, preferably 16 GB
  - I open A LOT of browser tabs
- An NVIDIA GPU, preferably at least at the GTX/RTX xx60 level
  - I say that I do GPU programming, but really I use it for games

My preferences

Caveats

- My experiences are heavily driven by an engineering undergrad
  - CAD tools, scientific computing, and HDL synthesis

For a general purpose laptop for computer science and engineering related work I prefer:

- A decent enough CPU, nothing too fancy: something with 4 cores (preferably with SMT)
  - Intel i5 and i7, AMD Ryzen 5 and Ryzen 7 are around what I like
- At least 8 GB of RAM, preferably 16 GB
  - I open A LOT of browser tabs
- An NVIDIA GPU, preferably at least at the GTX/RTX xx60 level
  - I say that I do GPU programming, but really I use it for games
- At least 256 GB of SSD space, preferably with a supplemental HDD (or a microSD card slot)
  - 128 GB of SSD is alright with me if supplemented by an HDD (or a microSD card slot)

My preferences

Caveats

- My experiences are heavily driven by an engineering undergrad
  - CAD tools, scientific computing, and HDL synthesis

For a general purpose laptop for computer science and engineering related work I prefer:

- A decent enough CPU, nothing too fancy: something with 4 cores (preferably with SMT)
  - Intel i5 and i7, AMD Ryzen 5 and Ryzen 7 are around what I like
- At least 8 GB of RAM, preferably 16 GB
  - I open A LOT of browser tabs
- An NVIDIA GPU, preferably at least at the GTX/RTX xx60 level
  - I say that I do GPU programming, but really I use it for games
- At least 256 GB of SSD space, preferably with a supplemental HDD (or a microSD card slot)
  - 128 GB of SSD is alright with me if supplemented by an HDD (or a microSD card slot)
- At least a 1080p display (preferably higher, but lower than 4K)
  - 1080p provides the bare minimum amount of real estate I want
  - 4K can cause some integrated GPUs to choke up

Low hanging fruit

- Honestly, CPU isn't a *huge* deal for most general programming work (especially in school)
    - Mileage may vary if you have to do a lot of data processing

Low hanging fruit

- Honestly, CPU isn't a *huge* deal for most general programming work (especially in school)
  - Mileage may vary if you have to do a lot of data processing
- The biggest quality of life improvement you can get is probably getting an SSD with enough space and enough RAM for how many browser tabs/windows you like to open

PC Building

- I personally find PC building to be a fun activity

PC Building

- I personally find PC building to be a fun activity
- Not difficult to actually assemble

PC Building

- I personally find PC building to be a fun activity
- Not difficult to actually assemble
- The biggest hurdle is picking out the parts

PC Building

- I personally find PC building to be a fun activity
- Not difficult to actually assemble
- The biggest hurdle is picking out the parts
  - Balancing out parts, making sure they fit, making sure you're under your power budget etc.

PC Building

- I personally find PC building to be a fun activity
- Not difficult to actually assemble
- The biggest hurdle is picking out the parts
  - Balancing out parts, making sure they fit, making sure you're under your power budget etc.
- I'm not going to show you since I don't have spare parts laying around

PC Building

- I personally find PC building to be a fun activity
- Not difficult to actually assemble
- The biggest hurdle is picking out the parts
  - Balancing out parts, making sure they fit, making sure you're under your power budget etc.
- I'm not going to show you since I don't have spare parts laying around
- If you're interested but don't know where to start, you can look online or ask me

PC Building

- I personally find PC building to be a fun activity
- Not difficult to actually assemble
- The biggest hurdle is picking out the parts
  - Balancing out parts, making sure they fit, making sure you're under your power budget etc.
- I'm not going to show you since I don't have spare parts laying around
- If you're interested but don't know where to start, you can look online or ask me
- PCPartPicker is an addicting tool to help put a build together

Questions

How similar is Python to C++ and is the syntax of Python the main reason why it's considered to be much easier than C++?

How similar is Python to C++ and is the syntax of Python the main reason why it's considered to be much easier than C++?

- A lot of this is subjective, but I'll give my point of view

How similar is Python to C++ and is the syntax of Python the main reason why it's considered to be much easier than C++?

- A lot of this is subjective, but I'll give my point of view
- I consider Python and C++ to be nearly completely different animals as languages

How similar is Python to C++ and is the syntax of Python the main reason why it's considered to be much easier than C++?

- A lot of this is subjective, but I'll give my point of view
- I consider Python and C++ to be nearly completely different animals as languages
- I would hazard to say that the syntax being much cleaner and more concise than C++ makes it a lot easier

How similar is Python to C++ and is the syntax of Python the main reason why it's considered to be much easier than C++?

- A lot of this is subjective, but I'll give my point of view
- I consider Python and C++ to be nearly completely different animals as languages
- I would hazard to say that the syntax being much cleaner and more concise than C++ makes it a lot easier
  - It can come at a cost, however...

Happy Belated Birthday!

Happy Belated Birthday!

Thank you! 😃

How is Python built under the hood? I'm pretty sure it's done in C, correct?

How is Python built under the hood? I'm pretty sure it's done in C, correct?

- Note that Python is just a language with specifications for behaviors: anyone can actually implement an interpreter for it

How is Python built under the hood? I'm pretty sure it's done in C, correct?

- Note that Python is just a language with specifications for behaviors: anyone can actually implement an interpreter for it
- There is the reference implementation by the creator of Python, called CPython, which is implemented in C (...and Python)

How is Python built under the hood? I'm pretty sure it's done in C, correct?

- Note that Python is just a language with specifications for behaviors: anyone can actually implement an interpreter for it
- There is the reference implementation by the creator of Python, called CPython, which is implemented in C (...and Python)
  - Under the hood it compiles the Python into a **bytecode** for its virtual machine (kinda like how Java works)

How is Python built under the hood? I'm pretty sure it's done in C, correct?

- Note that Python is just a language with specifications for behaviors: anyone can actually implement an interpreter for it
- There is the reference implementation by the creator of Python, called CPython, which is implemented in C (...and Python)
  - Under the hood it compiles the Python into a **bytecode** for its virtual machine (kinda like how Java works)
- There's other implementations as well: https://en.wikipedia.org/wiki/Python_(programming_language)#Implementations

Also, is there work being done to speed up Python and optimize it? I've heard that Python can be particularly slow sometimes when compared to C++ or C. (1/2)

- Part of it boils down to the nature of the languages: Python is a fairly high-level language providing for lots of abstractions. Compared to it, C and C++ are low level languages

Also, is there work being done to speed up Python and optimize it? I've heard that Python can be particularly slow sometimes when compared to C++ or C. (1/2)

- Part of it boils down to the nature of the languages: Python is a fairly high-level language providing for lots of abstractions. Compared to it, C and C++ are low level languages
- C and C++ by design are much closer to how the hardware works (C is sometimes described as a "portable assembler")

Also, is there work being done to speed up Python and optimize it? I've heard that Python can be particularly slow sometimes when compared to C++ or C. (1/2)

- Part of it boils down to the nature of the languages: Python is a fairly high-level language providing for lots of abstractions. Compared to it, C and C++ are low level languages
- C and C++ by design are much closer to how the hardware works (C is sometimes described as a "portable assembler")
- C and C++ get compiled into actual instructions as run by the CPU

Also, is there work being done to speed up Python and optimize it? I've heard that Python can be particularly slow sometimes when compared to C++ or C. (1/2)

- Part of it boils down to the nature of the languages: Python is a fairly high-level language providing for lots of abstractions. Compared to it, C and C++ are low level languages
- C and C++ by design are much closer to how the hardware works (C is sometimes described as a "portable assembler")
- C and C++ get compiled into actual instructions as run by the CPU
- CPython compiles to a bytecode meant for its virtual machine, whose implementation interprets the bytecode

Also, is there work being done to speed up Python and optimize it? I've heard that Python can be particularly slow sometimes when compared to C++ or C. (2/2)

- Part of it boils down to the nature of the languages: Python is a fairly high-level language providing for lots of abstractions. Compared to it, C and C++ are low level languages

Also, is there work being done to speed up Python and optimize it? I've heard that Python can be particularly slow sometimes when compared to C++ or C. (2/2)

- Part of it boils down to the nature of the languages: Python is a fairly high-level language providing for lots of abstractions. Compared to it, C and C++ are low level languages
- What this boils down to is:

Also, is there work being done to speed up Python and optimize it? I've heard that Python can be particularly slow sometimes when compared to C++ or C. (2/2)

- Part of it boils down to the nature of the languages: Python is a fairly high-level language providing for lots of abstractions. Compared to it, C and C++ are low level languages
- What this boils down to is:
  - C and C++ programs are straight up lists of instructions for a CPU to run: a program

Also, is there work being done to speed up Python and optimize it? I've heard that Python can be particularly slow sometimes when compared to C++ or C. (2/2)

- Part of it boils down to the nature of the languages: Python is a fairly high-level language providing for lots of abstractions. Compared to it, C and C++ are low level languages
- What this boils down to is:
  - C and C++ programs are straight up lists of instructions for a CPU to run: a program
  - Python scripts are programs interpreted and compiled by a program to instructions for a virtual CPU to run, which is another program in itself

Also, is there work being done to speed up Python and optimize it? I've heard that Python can be particularly slow sometimes when compared to C++ or C. (2/2)

- Part of it boils down to the nature of the languages: Python is a fairly high-level language providing for lots of abstractions. Compared to it, C and C++ are low level languages
- What this boils down to is:
  - C and C++ programs are straight up lists of instructions for a CPU to run: a program
  - Python scripts are programs interpreted and compiled by a program to instructions for a virtual CPU to run, which is another program in itself
- These additional layers hurt the level of performance that Python has compared to C/C++

Also, is there work being done to speed up Python and optimize it? I've heard that Python can be particularly slow sometimes when compared to C++ or C. (2/2)

- Part of it boils down to the nature of the languages: Python is a fairly high-level language providing for lots of abstractions. Compared to it, C and C++ are low level languages
- What this boils down to is:
  - C and C++ programs are straight up lists of instructions for a CPU to run: a program
  - Python scripts are programs interpreted and compiled by a program to instructions for a virtual CPU to run, which is another program in itself
- These additional layers hurt the level of performance that Python has compared to C/C++
- There are other interpreters for Python which may do other tricks such as just-in-time (JIT) compilation where some of the script/bytecode can be futher compiled into real CPU instructions to run
  - Java Virtual Machine implementations take advantage of this a lot

What exactly is object code?

What exactly is object code?

- Object code is a chunk of "code" produced by a compiler/assembler

What exactly is object code?

- Object code is a chunk of "code" produced by a compiler/assembler
- When we refer to "object code" it's usually in the sense that it's only a part of a bigger program

What exactly is object code?

- Object code is a chunk of "code" produced by a compiler/assembler
- When we refer to "object code" it's usually in the sense that it's only a part of a bigger program
- i.e. it can't exist purely on its own as it's missing info such as where other functions or variables are (could be in another object code file)

What exactly is object code?

- Object code is a chunk of "code" produced by a compiler/assembler
- When we refer to "object code" it's usually in the sense that it's only a part of a bigger program
- i.e. it can't exist purely on its own as it's missing info such as where other functions or variables are (could be in another object code file)
- When we "link" object code together we're resolving this missing info, and when everything is resolved we have something that's actually executable

What exactly is object code?

- Object code is a chunk of "code" produced by a compiler/assembler
- When we refer to "object code" it's usually in the sense that it's only a part of a bigger program
- i.e. it can't exist purely on its own as it's missing info such as where other functions or variables are (could be in another object code file)
- When we "link" object code together we're resolving this missing info, and when everything is resolved we have something that's actually executable
- This is why we get errors from the *linker* (`ld`) whenever we have undefined function: it hasn't figured out where the function is, which could either be from an unlinked object or unlinked shared object ("dynamic library", "shared library")

What exactly is object code?

- Object code is a chunk of "code" produced by a compiler/assembler
- When we refer to "object code" it's usually in the sense that it's only a part of a bigger program
- i.e. it can't exist purely on its own as it's missing info such as where other functions or variables are (could be in another object code file)
- When we "link" object code together we're resolving this missing info, and when everything is resolved we have something that's actually executable
- This is why we get errors from the *linker* (`ld`) whenever we have undefined function: it hasn't figured out where the function is, which could either be from an unlinked object or unlinked shared object ("dynamic library", "shared library")
    - (or just not implemented entirely because you mistyped the name of the function)

Thank you all for a great semester!