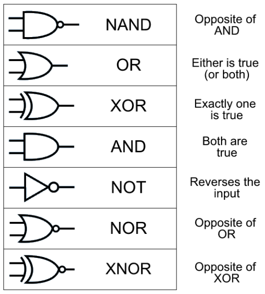
Systems Programming:

- Transistors and logical gates:
    - Hardware
        - A transistor is a semiconductor device that can be used to amplify or switch electronic signals
        - In this class, we are concerned with switching since that is what allows for digital computing
    - Transistor history
        - The concept of a transistor was first proposed by the Austrian Julius Lilienfeld in 1926
        - The first working transistor was made at Bell Labs in 1947
            - John Bardeen
            - Walter Brattain
            - William Shockley
    - Transistor
        - The basic digital transistor concept
            - Three connections
                - Signal In 1 (collector)
                - Signal In 2 (base)
                - Signal out (emitter)
            - When a voltage is applied to the base wire, current will flow from the collector to the emitter wire
                - Or, possibly, not flow, depending on the transistor type
        - This is accomplished via materials engineering that makes the N region, pictured here, conductive when a voltage is placed on the base wire
        - This acts something like a light switch you find on a wall
        - When a voltage is applied to the base wire, the switch is “on” (in the case of PNP transistors)
        - When a voltage is off, the switch is “off”
    - Predecessors
        - The first computers created actually used electromechanical relays
        - These are switches that are powered by magnetic fields applied to an input
        - Relays are still as common as switches but they are too slow for general-purpose computing
        - The first truly digital computers were created using vacuum tubes
        - Vacuum tubes can be used like transistors in a triode configuration
        - When a positive charge is applied to the grid surrounding the cathode, electrons will flow across the anode
        - When a negative charge is applied, electrons will stop flowing

        - This technology was used to build a computer that was instrumental in the Allies winning World War 2
- Transistor history
    - How can transistors be used as the basis for building digital systems?
        - Pretty easy!
            - High (or low) voltage indicates a 1
            - Low (or high) voltage indicates a 0
- Logical gates
    - Using this basic idea we can create something called a NAND (not and) gate
    - But before we get into that, let's take a quick look at logical operations and truth tables
- AND
    - By convention
        - 1 as high voltage = TRUE
        - 0 as low voltage = FALSE
    - A bitwise AND operator:
        - 1 if inputs A and B are both 1
- OR
    - The bitwise OR operator:
        - 1 if either A or B are 1
- NOT
    - The bitwise NOT operator:
        - 1 if A is 0
        - 0 if A is 1
- NAND
    - The bitwise NOT AND operator:
        - 1 if either A or B are 0
    - Turns out that NAND is very important
    - Somewhat surprisingly, all other logical operations can be created using a combination of NAND operations
        - This is also true of NOR
        - NAND and NOR are what is known as functionally complete
- Implementing logic
    - Using switches or transistors, we can implement logical operations digitally
    - Pictured at right is a logical AND circuit
    - If both buttons are pressed, the currency will flow through the transistors
    - This allows electricity to flow through the LED, lighting it up
    - The LED is our “output” for this simple digital system

    - If no button is pressed, no current is available to the LED
    - If either button is pressed, the currency is available to the LED
- Logical symbols
    - When designing digital systems, symbols are used for the various logical gates
        - NAND
        - OR
        - XOR
        - AND
        - NOT
        - NOR
        - XNOR



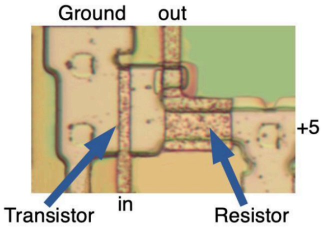
NAND	Opposite of AND
OR	Either is true (or both)
XOR	Exactly one is true
AND	Both are true
NOT	Reverses the input
NOR	Opposite of OR
XNOR	Opposite of XOR

- NAND gate
    - Recall that I said that NAND gates are special
    - NAND gates are functionally complete
    - All other gates can be constructed with them
- NOT gate
    - How could we create a NOT gate using a NAND gate?
    - Pretty easy, once you see it: simply run both inputs into a single NAND
    - If the input is 1, then both the inputs to the NAND are 1, and the output is 0
    - If the input is 0, then both inputs to the NAND are 0, and the output is 1
- And gate
    - Okay now we have NAND gates and NOT gates
    - How can we create an AND gate?
    - Again pretty easy once you see it: AND is just NOT NAND right?
    - So run a NAND through a NOT in series and you get an AND gate
- OR gate
    - How about an OR gate?
        - OR means either A or B
        - This is the same thing as saying NOT NOT A AND NOT B
            - NOT (NOT A) AND (NOT B)
        - So run A and B through NOT gates, then through a NAND gate
- AND SO ON
    - So using these techniques, we are able to construct all the different logical gates we might need for a digital system
    - Everything builds up from NAND gates
    - Now, how do we implement NAND gates using transistors?
- NPN vs. PNP transistors
    - It turns out there are two different major types of transistors

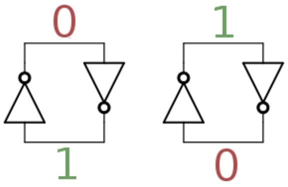
        - NPN transistors
            - Positive voltage allows current to flow
            - This is what we have been discussing mainly so far
        - PNP transistors
            - The application of voltage prevents current from flowing
            - Inverts the logic of the NPN transistor
        - Field-effect transistors
            - In order to make transistors smaller, our friends in EE have been creating smaller and smaller implementations of these basic building blocks
            - MOSFET technology has been used to reduce size and pack more transistors into smaller and smaller areas
            - The NPN and PNP equivalent in MOSFETs are called NMOS and PMOS
        - NAND gates and transistors
            - How could we implement a NAND using NPN transistors?
            - Here is the diagram for an NMOS NAND gate
                - If A and B are high, current will flow from Vcc to the ground
                - Otherwise, it will flow out (F)
            - A simple NAND implementation
            - A more modern and slightly more complex CMOS NAND gate
            - Note that the top transistors are PNP gates (thus voltage stops current) while bottom gates are NPN transistors
            - A physical layout of the NAND gate
            - CMOS uses both NPN/NMOS and PNP/PMOS-type semiconductors
                - Again, this is not an EE class
                - Just know that
                    - NPN/NMOS - currency flows if voltage is applied
                    - PNP/PMOS - currency flows if voltage is NOT applied
- Transistors and logical gates
    - Today we took a beginner's tour of the lowest level of digital computing
    - We discussed transistors, an electrical device that can be used for digital switching
    - We revisited the logical operators
        - We discussed how NAND can be used to create other logical operators
    - We looked at how to implement these logical operators using transistors
    - REMEMBER: IT IS JUST LIGHTNING RUNNING THROUGH SAND TO MAKE IT THINK

Introduction to Binary Computing: Storing and Adding Bits

- In this lecture, we will look at
    - The implementation of registers in the 8086 chip
    - The implementation of an adder (half and full) circuit
- The 8086 chip
    - Produced by Intel from 1978 to 1998
    - 16-bit chip
        - Supported 20-bit addressing
    - First in a series of x86 chips which, unexpectedly, would take over the world
    - Chip was a reaction to the delay of the iAPX 432 chip
    - iAPX 432 was an interesting chip
        - No user-facing registers
        - Stack machine
        - Hardware support for garbage collection, object-oriented programming
        - These ideas will resurface when we discuss the JVM
        -
- 8086 registers
    - Registers are very fast memory stores located close to the CPU
    - CPUs use registers for storing and mutating data
    - As with all binary systems, data is stored in 1s and 0s
    - NOT gate implementation
    - The in wire, when activated, opens a channel to the ground, causing current to flow to the ground
    - When not activated, the channel is blocked, so the output voltage is high
    - You can see how this inverts the signal in
    - The physical layout of a NOT gate
        - Ground and +5 carry electric current
        - Current on IN opens the resistor, causing the current to flow to the ground, rather than out

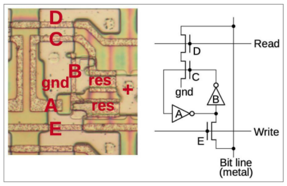


- Implementing a bit
    - How can we implement a stable bit value using NOT gates?
    - Chain them!
    - If the top is 0, it will stay 0 as it goes through the bottom gate
    - And vice versa

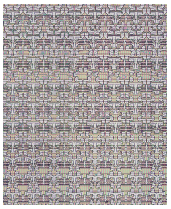


-
    - Adding reading and writing to the bit value
        - A read gate

        - A write gate
    - When the read gate is open, the value is written to the bit line
    - When the write gate is open, the value of the bit line is written to the register
    - Note that the read gate is reading the inverted value of the bit and inverting it again
    - The physical layout of this register bit
    - Note that A connection is much smaller than E
        - E will overpower A on write



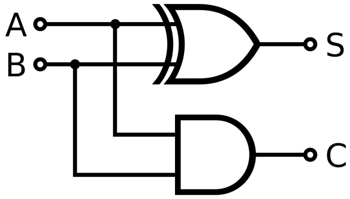
- 8086 registers
    - The register file: a collection of these register implementations, all wired together so that certain patterns read and write from certain register positions
    - Registers are not usually 1-bit, but rather 8, 16, 32, 64, or more
    - How is that done?
    - Just wire a bunch of them up next to one another
    - If you have an 8-bit computer, an 8-bit register is just 8 of these single-bit storage mechanisms sitting next to one another
    - Each one is connected to a different “bit line” (of the bus)
    - The read or write lines activate all of them at once and they take a value off the bus



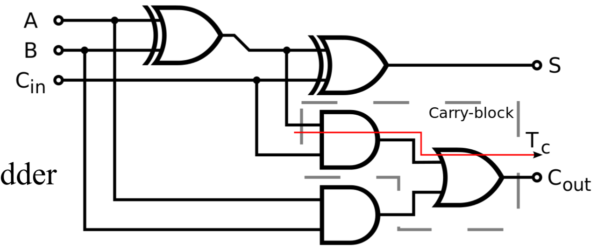
-
    - We’ll see this clearly when we go over the Scott CPU
    - Fun fact: In the x86 architecture, some registers can be partially addressed
    - Implementing this requires more wiring
    - More “ports” to register
    - There are even more complex registers available
        - We are not going to go into all the details
    - That’s a crash course of how registers work, let’s look at doing something useful with values that are stored in registers
- Adder circuitry
    - We have covered how a bit is stored in a register
    - Let's look at how to add two bits together
    - What does binary data look like?
    - Okay, so the carry bit
        - Just like with decimal math, we must carry overflow in binary math
        - Simpler since there are only two possible values: 1, 0
        - In the case of 1+1, we have a carry value to the 2s place
        - 1+1 = 10
            - There are 10 types of people in this world…

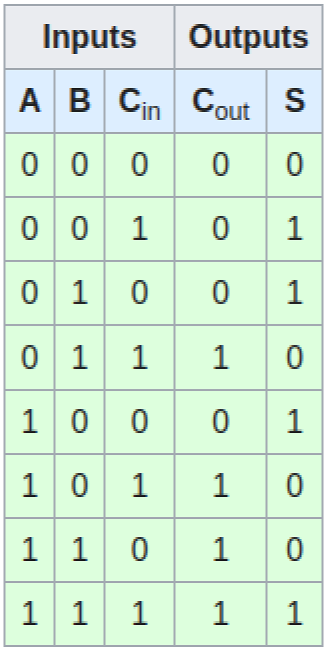
Addition		Result	Carry
0 + 0	=	0	0
0 + 1	=	1	0
1 + 0	=	1	0
1 + 1	=	0	1

-
    - This is called a half adder because it does not have an input for the carry bit from another binary addition

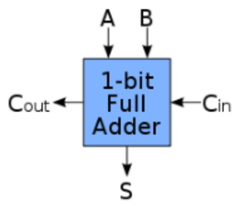


- A full adder has a Cin as well as a Cout
- Cout is true IF
    - A, B, and Cin are 1
    - A and Cin are 1
    - B and Cin are 1
- This is the logical layout of a full adder
- The full adder truth table



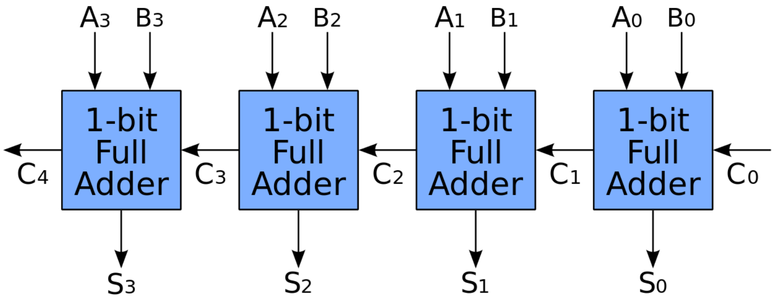


Inputs			Outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



- Schematic symbol for a 1-bit full adder

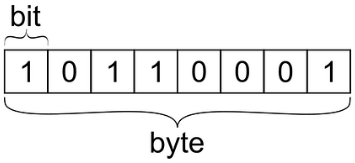
- Here we see a four-bit adder with a “ripple carry”
- The carry bit ripples through the gates from right to left
- This is slow and can be optimized with various additional wiring
    - Carry-lookahead



Bytes, Hex, and Integer Representation:

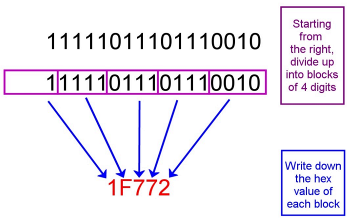
- In the last lecture we looked at
    - How binary data is stored in the computer

  - How an operation, addition, is implemented via logical gates
- In this lecture, we are going to move up a level and consider data representations
  - At the machine level, everything is binary
    - Different interpretive schemes can be imposed on this binary data
- Bytes:
  - The smallest group of bits is known as a byte
  - A byte is 8 bits
  - With 8 bits available we can represent 2^8 numbers
    - 0-255 unsigned



  - You will find the limit 255 in some funny places
  - CSS color specification
    - Each color in R, G, B can have a value of 0-255
- ASCII
  - ASCII is an example of a representation imposed on bytes
  - Developed from telegraph code
  - Work on standard began in 1960
    - Encodes 128 english characters into 7 bit integers
  - ASCII codes are shown at right
  - When you work with string literals in C, this is what is actually being stored in memory
  - NB: you can treat these as unsigned integers
    - We do that when converting a numeric char to its actual numeric value in the project
- Hex
  - Hexadecimal is another way to represent binary data
    - Unrelated to ASCII
    - Rather, and efficient way to specify groups of 4 bytes
  - Uses 16 characters to represent a half byte (a nibble)
    - 0-9, then A-F
  - In written representation, hex is typically prefixed with an 0x.
    - 0x00F
  - You can use this notation as a literal value in C, Java, etc…
- Converting binary to Hex
  - Pretty straight forward
    - Group in nibbles (groups of 4 bits)
    - 0-9 -> converts to the same numeric character
    - 10-15 -> A, B, C, D, E, F

Converting Binary to Hex

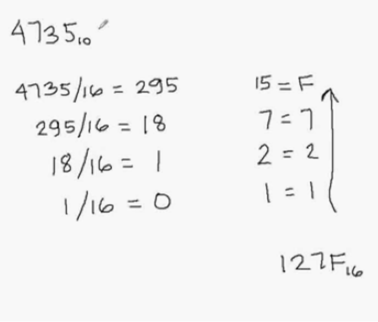


Starting from the right, divide up into blocks of 4 digits

Write down the hex value of each block

$11111011101110010_2 = 1F772_{16}$

- Converting decimal to Hex
  - A bit more work
    - If number is < 16, just convert it directly



4735_{10}

$4735/16 = 295$ $15 = F$

$295/16 = 18$ $7 = 7$

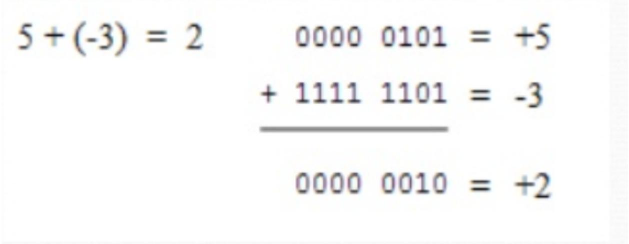
$18/16 = 1$ $2 = 2$

$1/16 = 0$ $1 = 1$

$127F_{16}$

- Else
  - Divide the number repeatedly by 16, writing down remainders, until value is <16
  - The last value is the first number in the hex, add remainders in reverse order

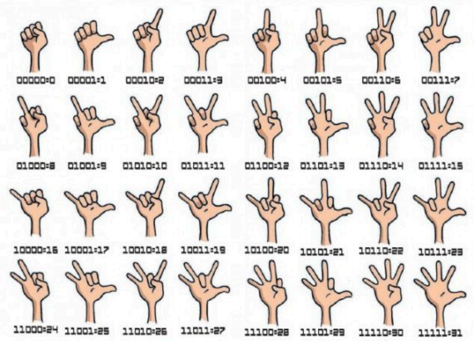
- Signed integers
  - How could we represent signed integers?
  - Option 1: a sign bit
    - By convention, let's say that the most significant bit is the sign bit
    - What number is this using this encoding?
  - Same bits as unsigned 177
  - The two different values we assigned to this bit pattern are the encoding we are using to interpret them
  - Signed integers typically do not solely use a sign bit
  - Instead, they use an encoding known as 2's complement
  - The encoding is shown at right
    - -128 is 10000000
    - -1 is 111111111
- Binary addition
  - Why on earth?
  - To understand why, you need to understand how binary math works
  - Let's look at addition
    - Note the carry bit from our earlier discussion on addition
  - Let's add 5 and -3
    - 5 rep: 0000 0101
    - -3 rep: 1111 1101
  - Woah, just normal binary math gives us the correct result
  - 2's complement has this wonderful feature
    - Math with negative numbers works in the same logical manner



```
5 + (-3) = 2      0000 0101 = +5
                + 1111 1101 = -3
                -----------
                  0000 0010 = +2
```

  - Converting a number to 2's complement:
    - Start with the positive binary rep
    - Subtract 1

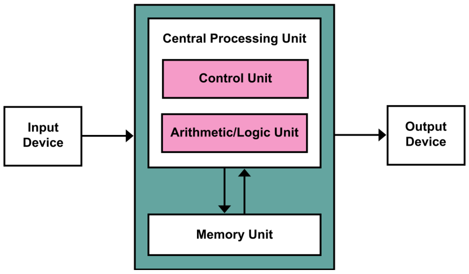
        - Flip the bits
    - Getting -3
        - Start with 3 = 0000 0011
        - Subtract 1, gives 2 = 0000 0010
        - Flip the bits = 1111 1101
    - Binary Finger counting: can count to 31 on one hand



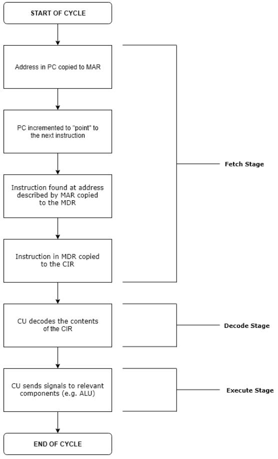
- We took a look at how binary data can be encoded
- We looked at ASCII, a character encoding
- We looked at hexadecimal encoding
- We looked at integer encoding
- We discussed 2s complement, a scheme for encoding signed integers
    - 2s complement has a really nice property of allowing standard addition to work with negative numbers without a change
- Remember it's just BITS

The CPU:

- Major components of the CPU chip
    - Adder (for memory, we will ignore)
    - Registers
        - Very fast memory stores located close to the CPU
        - Can interact with the ALU and one another quickly
    - Control unit
        - Directs the operation of the processor
        - Tells the memory, ALU and other devices how to respond to instructions being executed by the CPU
    - Decoding logic
        - Instruction are “decoded” into a series of control unit operations
        - Feeds the control unit with what to do for a given instruction
        - This is all done via logical gates based on the concepts we have discussed earlier
    - Microcode
        - Encoded logic for each instruction
        - Recall the x86 is a CISC architecture
        - With microcode, complex instructions can be reduced to a series of simpler instructions embedded in logic on the chip
    - Arithmetic logic unit (ALU)
        - Arithmetic logic unit - does most of the CPU work
        - Contains circuitry like the adder we looked at previously
            - More sophisticated than a simple ripple carry model
            - Also contains operations like shift, compare, etc…
        - Also contains temporary registers

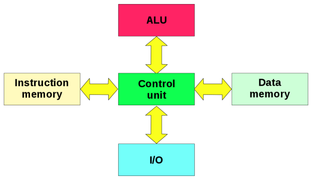


- The Von Neumann Architecture
    - This chip is an example of the Von Neumann Architecture
        - CPU with a control unit and ALU
            - Control unit contains a program counter and instruction register
        - Memory via a BUS
            - Stores data and instructions
        - I/O mechanisms
    - Note that instruction fetch and data fetch cannot be in parallel
        - They share a common bus with the memory unit
        - This is known as the Von Neumann Bottleneck


START OF CYCLE
Address in PC copied to MAR
PC incremented to "point" to the next instruction
Instruction found at address described by MAR copied to the MDR
Instruction in MDR copied to the CIR
Fetch Stage
CU decodes the contents of the CIR
Decode Stage
CU sends signals to relevant components (e.g. ALU)
Execute Stage
END OF CYCLE

-
    - The Fetch-Execute Cycle
        - The basic cycle of the CPU in the Von Neumann architecture is called the fetch-execute or fetch-decode-execute cycle
            - This cycle can be executed serially, or in more modern CPUs, in parallel
        - The following is a rough description of each step
        - Everything starts with the *Program Counter*
        - The program counter is a register that holds the address of the current instruction in memory to execute
        - It is called a counter because, in normal operations it is continually incremented
    - The fetch stage
        - Fetch stage steps
            - Address in program counter register is copied to the MAR
                - Memory address register
            - Program counter is incremented to next instruction position
            - The memory address register is a register that tells the memory subsystem what piece of data to put on the system bus
                - This is, of course, a simplified, view of things
            - The memory subsystem copies the instruction found at the address in the MAR into the MDR
                - The memory data register
            - The memory data register is the “outbox” of the memory sub-system
            - Kind of like the output of a vending machine
            - Finally the MDR is copied into the CIR
                - Current Instruction Register
            - The current instruction register is located at the heart of the control unit of the cip, and it is here that the instruction is turned from bits into actual execution of operations within the CPU
    - The decode stage
        - Decode stage steps

      - The control unit (or a sub-component, the decode unit) takes the instruction in the CU and determines what to do
      - Sends signals to the appropriate circuitry in the ALU to activate the correct logic
        - NB: a memory read may stall execution in this step to load a new value into the MDR
  - The execute stage
    - Execute stage steps
      - Based on signals received from the control unit after it has decoded an instruction, the ALU performs mathematical or logical functions on data stored in registers
      - The ALU writes values to locations (registers or memory, depending on the instruction type)
      - May update the PC to point to a new location
      - The ALU may write to the program counter register
        - To support things like: conditionals (if statements), iteration(for loops), or function calls
          - Functions are just “code in another place”
      - Now repeat the cycle really fast, like a few billion times per second
  - The interrupt stage
    - i/o devices need a way to signal to the CPU that input is ready
    - This is done with an interrupt
    - Interrupt handling forms another stage in the modern CPU cycle
    - Also used by the operating system to time share the CPU across multiple processes
      - A “timer interrupt” runs the OS code, which must pick the next process to be run on the CPU
      - OS code runs in a different mode and is able to move values off of the CPU to “suspend” a process and values onto the CPU to “resume” a process



- The harvard architecture
  - The Von Neumann architecture is not the only possibility
  - Early on the Harvard architecture was a competing layout
    - Separate pathways for instructions and data
  - Modern CPUs have moved towards this architecture by introducing separate caches for instructions and data
    - Sometimes called a modified harvard architecture
  - When instructions are retrieved from an instruction cache, a CPU acts like a harvard architecture
  - When instructions must load from memory, the CPU acts like a Von Neumann Architecture
  - This is typically hidden from the user/programmer
  - To us, the programmer, modern machines look like a Von Neumann Architecture, even if internally it usually acts like a Harvard Architecture device

- The Scott CPU
    - We will discuss the Scott CPU and each step in the Fetch/execute cycle using a simple 8-bit model
    - We will see higher level emulators later in the class: little man computer and MARS (MIPS emulator)
    - Scott CPU Assembly
        - In order to understand the Scott CPUs mechanics, we are going to have to introduce assembly language
        - Assembly language is a very low level programming language that typically corresponds 1-1 to machine code
    - Machine code is the binary code necessary to make a machine execute
    - You typically use an assembler to convert from assembly to machine code
    - As you might expect, the Scott CPU has a simple assembly language
    - Note: the Scott CPU is 8 bit
        - Has only 4 registers
        - Has only 256 memory slots
    - Some Scott CPU Assembly instructions
        - ADD, etc. <R1>, <R2>
        - AND, etc. <R1>, <R2>
        - CMP <R1>, <R2>
        - LD, ST <R1>, <R2>
            - Load memory pointed to by R1 to R2
        - DATA <R1>, <value>
            - Load value into R1
        - JA, JZ, JE
            - Jump if greater, zero if equal
- The CPU is the core component of a modern computer
- It consists of various components
    - Control unit
    - ALU
    - Registers
- Most CPUs use the Von Neumann Architecture
    - With the common fetch, decode, execute, (interrupt), repeat loop
- The harvard architecture lost initially, but has come back in modified form, via internal, hidden to the programmer, caches
- Next time we will drill in on implementation details of the CPU

Scott CPU - 1:

- The heart of the computer
- In the next three lectures we are going to do a deep dive on a simple CPU: The Scott CPU
    - 8 bit
    - 4 registers
    - 256 memory slots
    - Primitive assembly language

        - Kinda really simple
- Registers
    - Recall the lecture where we discussed how to store a single bit in an 8080 chip
    - In the Scott CPU there are several registers:
        - R0-R3 (general registers)
        - IAR - instruction address register
            - Aka the program counter
        - IR - instruction register
        - TEMP - temporary register
        - ACC - accumulator register
        - MAR - memory address register
        - The scoot CPU doesn't have a MDR (memory data register) because data is moved directly from RAM into destination registers
            - Unrealistic but makes things easier to understand
    - In the Scott CPU, registers are made up of a series of 8 one bit storage units (labeled M) in parallel
    - Note that there is a "set" line labeled "s"
    - The 's' line, when enabled, takes the value from the bus (discussed in a moment)
    - There is also an enable line (labeled "e") that puts the value of the register onto the bus
    - Together, all these features make up a register
- The Bus
    - In the Scott CPU there is an 8-bit bus that connects all components
        - Recall the "bit line" when we were looking at an 8086 registers
    - The "bus" is just a set of "wires" that connects everything together
    - To "move" data from one place to another:
        - The source memory location is enabled to place its value on the bus
        - The destination is set to take the new value
        - The source is disabled
    - On the Scott CPU this general series of operations is the same for both memory locations and registers
- The ALU
    - The ALU is the logical unit of the CPU
        - Actually pretty simple
        - A collection of logical operators on the 8 bits
        - A temporary input register
        - Some wiring to enable one of the operations
        - A bit of wiring for the flags register
        - An output register (the accumulator)
        - The flags register is used to store comparison values
            - Greater than
            - Equal
            - Zero
            - Etc

    - The flags and accumulator registers are where the results of computations in the ALU end up
- What about these operations we have available in the ALU?
- XOR, OR, AND, NOT
    - Bitwise logical operations
- SHR, SHL
    - Bitwise shift operations
- ADD
    - Binary integer addition
        - A ripple adder like we saw earlier
- NOT implementation
    - Note, only one input
        - “Not” is a unary rather than a binary operation
    - Again pretty simple
    - For each bit, apply a logical not
        - Remember: we know how to make a NOT out of a NAND
        - And we know how to make a NAND out of transistors
        - And we know how to make transistors out of sand
- OR, XOR, and AND implementations
    - Binary operations, take two values as inputs
    - For each bit, apply the given logical operation
    - At right, we see the ANDer
        - The first bits of A and B are anded, etc…
    - Remember: we know how to make an AND NOT out of a NAND
    - And we know how to make a NAND out of transistors
    - And we know how to make transistors out of sand
- SHIFT implementation
    - As with NOT, only one input
        - Shifts are unary
    - Could not get any simpler! No logical gates are even required!
    - Note that there is a carry bit that will go out to our flags register
    - What does a left shift do?
        - 0000 0010 - 2 decimal
        - 0000 0100 - 4 decimal
            - Well what happens if you “shift” a decimal number to the left?
                - 0000 0050
                - 0000 0500
    - Shifting is also useful for bit masking
    - If you want to test the value of a single bit into a byte, you can shift the value 1 to that and then bitwise-AND the bit with the value
- A basic bitwise test
    - 0101 0100 - to test
    - 0000 0001 - value 1

  - 0000 0100 - 1 << 2
  - 0000 0100 - bitwise AND of value 1 and value 3
  - if (value 4 is not 0) there was a bit in position 3 of value 1
- Adder implementation
  - Addition requires two inputs, so a binary operator
  - The adder here uses the basic digital addition logic we covered earlier
  - 8 full adders wired up in parallel to add the values up
  - Remember we know how to make all these logical gates out of NAND
  - And we know how to make a NAND out of transistors
  - And we we know how to make transistors out of sand
  - We have a rough understanding of how this CPU works all the way down
- Additional stuff
    - XOR can be used to detect equality and greater than booleans, set into the flags register
    - Zero is detected on output and placed into the flags register
    - Bus 1 is some logic to put the binary value 1 on the bus
  - Note that gates labeled E
  - These are enabler gates
  - Depending on which operation the control unit turns on, one of the operations will be enabled
  - Here is what an enabler looks like logically
  - A series of AND gates, hooked up to the enable signal as well as the input
- By placing this gate between the operation and the bus, the computer controls which value flows out to the accumulator
- A bit of control logic in the lower part of the ALU is responsible for activating the correct enabler
- How does that work?
  - Input 000 - adder enabled
  - Input 001 - orer enabled
  - Etc..
  - This is a 3x8 decoder
- Decoder
  - Takes n input bits
  - Sets one output bit from 2^n options
  - Recall n bits can represent 2^n numbers
  - A decoder will be referred to as nx(2^n)
    - Eg. 2x8
  - Here is a 2x4 decoder
    - 0/0 - first wire is on
    - 0/1 - second wire is on
    - 1/0 - third wire is on
    - 1/1 - fourth wire is on

        - This simple circuit can be used with enablers to select from 4 different operations
        - We will also see a decoder when we look at the memory subsystem
    - And that’s the whole 8 bit ALU!
    - Not so bad, is it? A lot going on but considered individually and ignoring some of the details, all pretty easy to understand
- Registers:
    - General purpose registers are quite simple in the Scott CPU
    - They are connected to the bus
    - They have a SET and an ENABLE input
        - When SET is on, the register takes the value from the bus
        - When ENABLE is on, the register puts its value on the bus
    - Internally we have eight of our little memory bits that we discussed previously
    - Each is connected to one wire on the bus
    - A set line is connected to the memory slots
    - When the set line is enabled the memory is updated to whatever is on the bus
    - On the other side we have a bunch of AND gates wired to an enable line
    - When the enable line is, er, enabled, the memory bits will be placed on the bus
    - Combining these two circuits gives us our register
        - A place to store binary values
        - A way to set the value from the bus
        - A way to put the value onto the bus
- The control unit
    - The control unit is the most complex part of the Scott CPU
    - Responsible to managing the rest of the computer
    - At a high level, the ALU:
        - Reads an instruction
        - Enables the correct gates to execute that instruction
        - Repeats
    - At a slightly lower level way to think about the CPU:
        - For any given “step”, the goal is to move an 8 bit value from one memory location to another memory location
            - Where “move” can also involve a transformation of the value through the ALU
        - ENABLE one memory location onto bus
        - SET another memory location to that value
    - Control Unit Components
        - Clock
        - Stepper
        - Instruction address register
        - Instruction register
        - And then a bunch of logic
    - Note that visually the Scott CPU is broken up into two functional areas
        - Enables on the left

        - Sets on the right
    - Ignore the details for now, just notice that we can enable and set various places in the CPU
        - Registers, ACC, RAM, IAR
    - Registers
        - IR - Instruction Register holds the current instruction
        - IAR - instruction address register holds the address of the current instruction
    - The clock
        - The clock emits three signals
            - Clk - a normal tick
            - Clk e - an enable signal that lasts 1 and .5 ticks
            - Clk s - a set signal that lasts .5 of a tick
        - Note that the clk e is attached to all the enable-side logic
        - And correspondingly, the clk s signal is tied to the set-side logic
        - Clock design
        - Signal needs to turn on and off at a certain rate
        - Conceptually this can be done by hooking a not gate back into itself and giving the wire a long enough path to travel
        - Creating the other two signals, clk e and clk s, involves taking two clocks and ANDING/ORING them together
        - A second clock, clk d, is half-offset from clk
        - Clk e is clk OR clk d
        - Clk s is clk AND clk d
        - Note that this sets things up perfectly for “moving” data between memory locations
        - Clk e enables some memory location onto the bus
        - Half a tick later clk s sets that data into a particular location
        - Half a tick later clk s goes to 0, closing the set window
        - Finally, another half a tick later, clk e closes the enable window
    - We’ve already learned about how roughly 60% of the CPU works
    - Next time we will discuss how the memory sub-system works, that funny stepper thingie, and how the Scott CPU decodes instructions, FUN!
- The CPU
    - Today we discussed aspects the the Scott CPU architecture
    - The bus is responsible for transmitting values between various memory locations
    - The ALU (arithmetic logic unit) is responsible for implementing
        - We looked at how to implement many of the common functions in an ALU at the logical level
    - There are four simple registers in the Scott CPU
    - The control unit is responsible for coordinating the rest of the chip
        - We discussed the basic layout and the clock
        - Next time we will discuss the remainder of the control unit as well as memory

- The control unit
    - The clock (review)
        - Recall that there are three (well, four) clocks in play
            - Clk - the cpu clock
            - Clk d - a hidden clock half offset
            - Clk e - the enable clock
            - Clk s - the set clock
        - For every cycle of clk, clk e and clk s enable as well
    - The stepper
        - Here again is the control unit of the Scott CPU
        - Note that the clock is attached to three different things:
            - Clk s - attached to the set-side logic
            - Clk e - attached to the enable side logic
            - Clk - attached to something labeled “stepper”
        - What is a stepper?
            - The stepper is a bit of digital logic that enables one outgoing wire
            - The outgoing wire activated moves to the next step on every clock cycle
            - When the last step is reached, it resets, thus cycling through each output
        - Internally this can be implemented using the following logical setup
            - Don’t worry about the details, just appreciate how cool it is that someone thought of this
        - Now we have a steady drumbeat of signals to make things go
    - We can focus in on the stepper, the enables and the sets
    - Recall that for each step, the clk e and clk s signals will be enabled
        - So each step has an opportunity to move data from one place to another on the bus
    - Let’s consider a very simple operation: adding R0 to R1
    - What steps would need to happen for this to occur?
        - R1 moved to TMP
        - R0 sent through ALU (with TMP) to ACC
        - ACC moved to R0
    - If you look closely at the wiring here, this is accomplished in steps 4, 5, and 6
        - For each step, a location is enabled and a location is set via an AND gate tied to the stepper, and the appropriate clock
    - Assuming that the ALU add was enabled as well, R0 now holds the sum of R0 and R1!
    - We can move pretty much any data from anywhere to anywhere, using the same technique
    - Of course we don’t want to hard wire these paths into the stepper, do we?
    - Instead we want to enable them conditionally, based on … an instruction
    - And here we get to the crux of the control unit: the instruction register and the instruction address register

    - Instruction register - holds the current instruction
    - Instruction address register - holds the address of the current instruction
- The first three steps of the stepper are always the same
    - 1 - enable the IAR and sent it to the Memory Address Register
        - Also bump the MAR and store it in the accumulator
    - 2 - enable the memory value at that location onto the bus and set it to that instruction register
    - 3 - load the ACC into the IAR
- Note that in step 1, there is also that funny Bus 1 thing
    - Makes a value of 1 available to the ALU immediately (no bus)
    - This allows the ALU to immediately add 1 to the instruction address, bumping it to the next value and saving it in the accumulator
    - So after step 3, we have the next instructions address in the IAR
- So the first 3 steps of the stepper load the next instruction into the instruction register and bump the IAR to the next value
    - Pretty cool!
- Now what?
    - Now let's look at an actual instruction
- Instructions are just binary 1s and 0s
- Here is the schema for ALU instructions
    - Always start with a 1
    - Next three bits are the OP
    - Next two bits are register A (read)
    - Next two bits are register B (read and written to)
- How can we make the registers we hard-wired earlier be enabled and set when needed?
- We can use a 2x4 decoder, decoding 2 bits into one of 4 registers
- Now the stepper can be wired into this whole scheme as well
- If the first instruction bit is 1 then
    - Step 4: enable register B in and move it to TMP (selecting from bits 6 and 7)
    - Step 5: enable register A and send it through the ALU (to be operated on with TMP) to ACC
    - Step 6: enable ACC and send the value to register B
- Note the output of bits 1, 2, and 3 to the ALU
- Recall the ALU setup
- Note the output of bits 1, 2, and 3 to the ALU
- Recall the ALU setup: it has a decoder as well, which is connected to an enabler for the specific operation
    - So the value put on the bus will run through the correct operation with the value in TMP and be written to ACC
- Pretty incredible, isn't it?
- Ok, so what has happened here?

- Based on an instruction, we have properly moved data through our CPU to add two registers
    - We decoded the instruction and executed it, 3 easy steps (4, 5, 6)
- And not only that, but in steps 1, 2, and 3, we bumped the IAR so that on the next clock cycle, we will do the same darned thing with the next instruction
- What about jumps? We don't want to just linearly execute instructions right?
- Of course!
    - So let's look now at a conditional jump instruction
- The Jump If instruction is too large to fit into a single byte
    - First byte - first four bits are the code for the instruction
        - Next four bytes select the condition to test in the FLAGS register
    - Second byte - the address to jump to
- Wiring looks a little hairy, but the first four bits are just selecting the logic shown
- Next four bits are ANDed with the values from FLAGS
- Now look at the steps, assuming this instruction is enabled
    - Step 4: enable bus 1 and IAR, setting MAR and ACC
        - Note: ACC is IAR + 1 where MAR is just IAR
    - Step 5: move ACC to IAR
    - Step 6: if the flags match up, move memory to IAR
- Pretty bonkers, but it all works out
    - If the flags aren't matched, then IAR points to the next byte past where the conditional address was
    - Some very smart folks thought all this up kids
- The Scott CPU has some other instructions
    - Data
    - load/store
- They are just more of the same
    - Enabling the right locations and setting the right locations, based on the instruction in IR
- The Scott CPU has a pretty clear fetch stage (steps 1-3)
- The decode and execute stages are a bit smeared into one another, but that's OK, they are smeared into one another in real CPUs too
- Pretty cool, we have gotten down to brass tacks here
- IT'S JUST LIGHTNING RUNNING THROUGH SAND
- Modern CPU's
    - The Scott CPU is obviously very simplified
    - Note that only one instruction at a time is being fetched and executed
        - 7 cycles per instruction
    - Modern CPUs feature pipelining: multiple instructions executing at once in various stages
        - IF - instruction fetch
        - ID - instruction decode
        - EX - instruction execute
        - MEM - memory access

      - WB - register write back
    - This is instruction parallelism
      - Multiple instructions are executing at the same time
      - Note that we are finishing (retiring) one instruction per clock cycle
        - A lot better at retiring one instruction every seven clock cycles in the Scott CPU!
    - Obviously this requires much more sophisticated engineering to keep all these instructions in flight
    - All is not perfect, however
      - Some instructions depend on the results of other instructions
      - Some stages may take longer than a clock cycle to complete
    - Here is an example of a stall caused by an instruction staying in the fetch stage for an extra cycle
    - The intel core i9 X-series is an extremely sophisticated machine
      - Perhaps the most sophisticated machine ever mass produced
    - But it still operates on the same basic principles as the Scott CPU
- The CPU
  - Whew!
  - Today we got down to brass tacks and looked at how the Control Unit works
    - The stepper marches things along in various stages
    - Each stage allows movement of data from one place to another
    - The control unit has a logic that decodes instructions into something like the ALU, etc… can act on
    - And on and on it goes…
  - We also briefly considered pipelining in modern CPUs
  - Next time: memory!

Memory:

- Basics and caching
- Last few lectures
  - We saw how the control logic unit of the Scott CPU works
  - But we glossed over memory
    - The Memory Address Register (MAR) and all that stuff on the right hand side
  - Time to take a look at that
- To understand how memory works, we need to again recall the concept of a decoder
- A decoder takes n bits and enables 1 of n^2 wires
- Here we see two 4x16 decoders, wired into the MARs top 4 and bottom 4 bits
- What will this do?
  - Given an address in MAR, exactly one top wire and one bottom wire will be activated
  - Where those two wires meet is 8 bits of memory
- At the intersection of the wires is an AND gate
  - If both the vertical AND horizontal wires are enabled, then this memory location is selected

    - If, additionally the set wire is enabled, the memory location is written to from the bus
    - If, alternatively, the enable wire is on, the memory location is put on the bus
- And that’s it
    - The memory chip is much more simpler and regular than the CPU chip
- Types of memory
    - SRAM - fast, expensive, used primarily in registers, caches
    - DRAM - slower, but much less costly, lower power consumption
    - Flash - slower still, but stable without power

Memory latency:

- In the Scott CPU, Memory and Registers had the same access time
- This is not realistic
- In fact, there are *huge* differences between various memory locations
- Registers are typically immediately available to the CPU
- RAM is 100x+ slower
- Disk is even worse
- Network….. Fuggedaboudit
    - NB: some folks will freak out about CPU perf in systems that touch the network
        - Lol
- The difference in speed between the CPU and memory has grown significantly over time
- Consequently, memory access time has become the most significant bottleneck for CPU performance in most workloads
- Because there is such a huge penalty for accessing RAM, modern computer systems have introduced caches
- Caches store data in faster, intermediate memory, allowing the CPU to access memory more quickly

Caching:

- Note that there are different levels of cache
    - Level 1 - very fast (almost as fast as registers) and very expensive
    - Level 2 - larger, slower, less expensive
    - Level 3 - even larger, slower still, less expensive than level 2
- On a modern multi-core CPU, each core has its own L1 and L2 cache
- There is a shared L3 cache
- Note that a significant share of the CPU is dedicated to cache
    - Caching is crucial to modern CPU performance
- All caches broadly are implemented the same way
    - This is a set of memory blocks (not a single address, a block of memory)
    - The memory blocks have a valid indicator
    - There is some mechanism for mapping a particular memory address into the cache
- Broadly, there are three strategies for mapping a memory address into a cache
    - Direct mapping
    - Associative mapping
    - Set associative mapping

- Direct mapping
    - Consider the memory address 0110 0000 1100
    - Take some set of bits and map them directly to a slot in the cache
    - We look at a middle five bits (bits 4-8) to determine which cache slot to look in
    - Because the value is 0000 1 for this memory address, we look at cache entry 1
    - Of course, many different addresses could have the bits 0001 in it, so we need to ensure that the cached value is indeed for
    - This is what the tag bits are for: they are stored in the cache and compared after the look up to ensure that they equal our top four bits
    - Assuming that the tag bits match, we have a hit
    - We can now index into the cache block by the lower 3 bits to get the cached value at that memory location
    - In this case, 100 binary means slot 4 of 8
    - Not too complicated
    - But why are we storing a block of memory? Why not store just an individual piece of memory?
    - And why did we use the middle bits for the set selection, rather than, say, the high bits?
- To answer this question you must recognize that computer programs tend to demonstrate locality of reference
    - If I ask for memory location N, it is very likely that I will soon ask for location N+1
- Here is some pseudocode for matrix manipulation
- Note that the arrays are being accessed over and over in locations very close (or identical) to the previous iteration
    - This algorithm demonstrates a lot of locality
- To return to our cache implementation
    - You can see how having a block of data makes sense in this algorithm
    - But what about the selection of the set bits? Why the middle? Why not the high bits?
- Well, consider an array that would span multiple blocks
- Would the high bits of two blocks of memory next to each other be different?
    - NO!
- So they would map to the same cache line
    - BAD! They would conflict with one another
    - This is why we use the lower bits for the set selection
- Set associative
    - A more advanced algorithm for caches is a Set Associative cache
    - The principles are essentially the same, but rather than one cache entry in a given set you can now have multiple cache entries
    - Rather than comparing a single tag, each tag is scanned for a match
    - More complex logic in your cache
    - Potentially better caching behavior
- Fully associative

        - Even more advanced (from an engineering standpoint) is a fully associative cache
        - There is no set selection stage
        - Simply scan the entire cache for a matching tag
        - Maximizes your cache utilization regardless of workload
        - Difficult to make both large and fast

Virtual Memory:

- The last topic we will briefly cover is Virtual memory
- Broadly, the computer treats memory as a contiguous array of bytes
    - These bytes may be physically addressed
- Physical addressing can lead to many issues
    - Collisions between programs
    - Security issues
    - Etc…
- To address this, the notion of virtual memory was introduced
- All addressed go through a translation step
- This allows all processes to see a uniform address space
- The operating system is in charge of ensuring that the proper translation tables are maintained for processes
- Note that two processes, via virtual memory, can share the same physical page
    - If the page is read only (e.g. a DLL)
    - If the page has read/write but has not yet been written to
        - Only after write do the processes need separate pages
- Here is the Core i7 memory schema
    - Note separate data instruction L1 caches
        - Harvard model?
    - Unified L2 cache
    - Huge L3 cache - shared
    - MMU to do address translation for Virtual Memory
        - Has its own set of caches
    - Memory controller connected to RAM

Memory

- The memory systems are much more regular, in theory, than the controller and ALU components of a CPU
- However, because memory latency is a large issue in modern CPU performance, increasing amounts of CPU engineering has gone into efficient memory usage
    - Caches in particular
- Virtual memory is a feature of modern processes that, in concert with the OS, provides a safe and consistent view of memory for processes

Introduction to Assembly:

- Little man computer
- Assembly Language
    - What is assembly?
        - Assembly language is a low level programming language

        - Typically just above binary instructions
        - There is usually a 1 to 1 correspondence between assembly instructions and machine instructions
    - Machine instructions == machine dependent
    - Assembly language is specific to a particular architecture
    - Recall the saying “C is portable assembly”
        - This is because the C compiler can target different architectures
    - Is C really portable assembly?
        - Assembly typically does not have: function calls, structs, arrays, loop constructs
        - Assembly typically does have: jumps, raw access to registers, it’s a ton of work to get anything done
    - I would say that’s a good joke, but no, as low level as C is, it is much more portable than assembly
    - The assembler
    - Takes assembly code and converts it into binary instructions
    - Part of the gcc tool chain:
        - C -> assembly -> object file -> executable file
    - Assembly code is typically a linear set of instructions, labels and directives
        - Assembly instructions correspond to single instructions on the CPU
        - Labels are used to refer to things by address
        - Directives can be comments, hints to the assembler, etc…
    - To understand assembly, you have to understand the underlying hardware
        - One of the difficult things about x86 is that the hardware is a little insane
    - We are going to start with very simple hardware instead
- MC6800 Assembly
    - The code at right is some assembly taken from the MC6802 processor
        - 1 Mhz
        - 64 Kbyte memory address capacity
        - 16-bit
        - 2 accumulator registers, A & B
        - Stack pointer (we will discuss more on stack pointers later)
        - Index register
        - Condition code register (AKA a flags register)
    - Note the explicit addresses and machine code (in hex form) in the program
    - This is how people used to deal with computers!
    - And some still do, in embedded programming
- Scott CPU Assembly
    - We have already seen some assembly for the Scott CPU
    - Much less “fiddly” than the motorola assembly at right
    - Closer, in some ways, to modern assembly
    - Still annoying in some ways
        - No labels, for example

        - Due to 8 bit architecture, some assembly instructions map to two bytes, which makes things more complex
- Little Man Computer or LMC
    - An extremely simple model of a computer
    - Proposed by Dr. Stuart Madnick in 1965
    - Models a simple von Neumann architecture machine
    - Has the basic operations of a modern computer, but much easier to understand
    - LMC simplifications
        - 100 memory slots
        - Addresses are in decimal
            - No binary or hex to confuse us
        - There is only one general purpose register, called the accumulator
        - Values between -999 and 999 are supported
    - LMC architecture
        - Program counter - points to the current instruction (starts at 0)
        - Instruction register - the current instruction being executed
        - Address register - an address associated with the current instruction (if any)
        - input/output - rudimentary I/O devices to read and print numbers
        - RAM - a continuous array of decimal values from position 0 to 100
        - Note that there is no distinction between instructions and data in memory
    - LMC execution cycle
        - Check the program counter
        - Fetch the instruction from that address
        - Increment the program counter
        - Decode the fetched instruction into the instruction and address registers
        - Fetch any data needed
        - Execute the instruction
        - Branch or store the result
        - Repeat!
    - LMC instruction set
        - LMC instructions are 3 decimals
        - First decimal indicates the instruction type
        - Next two decimals are optional arguments for that instruction
    - LMC - MATH
        - ADD
            - 1XX - adds the value stored in location XX to whatever value is currently stored in the Accumulator
        - SUB
            - 2XX - subtracts the value stored in location XX from whatever value is in the accumulator
    - LMC - Memory
        - STA

        - 3XX - stores the value in the accumulator into the memory location XX
    - LDA
        - 5XX - loads the value in the memory location XX into the accumulator
- LMC - Control Flow
    - BRA
        - 6XX - unconditionally sets the Program Counter to the memory location XX
    - BRZ
        - 7XX - branches to the location XX if the accumulator is zero
    - BRP
        - 8XX - branches to the location XX if the accumulator is 0 or positive
- LMC - Input/Output
    - INP
        - 901 - ask the user for numeric input, to be stored in the accumulator
    - OUT
        - 902 - write the current accumulator value to the output area
    - HLT/COB
        - 000 - end the program, halt, take a coffee break
    - DAT <value>
        - Used to indicate that the value should be stored at the memory location corresponding to this instruction

- LMC
    - That’s it!
    - A total of 10 instructions
    - But we can still do some interesting things with them
    - More complex architectures add more instructions and infrastructure (registers, etc…)
    - But the fundamentals are the same
    - We will be using the excellent LMC simulator
- LMC - Labels
    - A nice way to avoid having to hard code addresses into your assembly program
    - Provide a symbolic way to refer to jump targets or data loads
    - Here we are storing 42 at the 4th position and loading it via a label
    - Labels also become important as we implement things like loops or conditional branches in our assembly program
- Assembly Review
    - Assembly is very low level programming, typically just above machine code
    - Assembly programs consist mainly of linear sets of instructions, as well as data, assembly directives, etc…
    - We are going to begin working with assembly using the LMC architecture

        - Simple and fun!
        - Despite the simplicity, it shows us the core operations in any assembly language
    - Next time: implementing some stuff in LMC!

Using LMC Assembly:

- Do things with the little man computer
- LMC Review
    - Recall the LMC architecture
        - Program counter
        - Instruction and address register
        - Accumulator register for work
        - input/output areas
        - An ALU
        - 100 memory slots
    - LMC Execution Cycle
        - Check the program counter
        - Fetch the instruction from that address
        - Increment the program counter
        - Decode the fetched instruction into the instruction and address registers
        - Fetch any data needed
        - Execute the instruction
        - Branch or store the result
        - Repeat!
    - LMC Instructions
        - ADD addition
        - SUB subtraction
        - STA store to memory
        - LDA load from memory
        - BRA unconditional branch
        - BRZ branch if zero
        - BRP branch if positive
        - INP get user input, put in acc
        - OUT output acc to output area
        - HLT/COB halt
        - DAT data
- Add One
    - Our first program is going to be extremely simple
    - Ask the user for a number, add one to it, and then output the number
    - INP - get user input and put it in the accumulator
    - ADD ONE - add the number stored at the label ONE to the value in the accumulator
    - OUT - output the value in the accumulator to the output area
    - HLT - stop execution

    - ONE DAT 1 - store the value 1 at the “current” position and make it available for reference with the label “ONE”
    - Not too bad, right?
    - This is a simple program but gives you the general flavor of assembly programming
- Countdown
    - Let’s do something more complex: Print the numbers 10 to 1 in decreasing order
    - How could we do that?
    - LDA - load the number 10 into the accumulator
    - BRZ - if the accumulator is 0, branch to EXIT
    - OUT - Else, print the accumulator
    - SUB - subtract 1 from the accumulator
    - BRA - unconditionally jump back to the start of the loop
    - HLT - halt (exit)
    - ONE, TEN - Data slots to hold constants
    - This demonstrates two higher level programming language concepts:
        - Conditional logic (if statements)
        - Looping (for, do, while)
    - Compilers take your higher level programming language constructs, like the while loop in C, and create instructions like this
- MAX
    - So far we haven’t had to worry too much about data sizing
        - Our computations have fit in the accumulator
    - Sometimes not all the working data can fit in the available registers
    - What to do?
    - Max: “ask the user for two numbers and print the maximum of the two”
    - Now we are in trouble: we need to compute the difference between the two numbers, but we also need to keep the numbers around to print the correct value
    - So we have three values that we are interested in
        - The first number entered
        - The second number entered
        - The difference between them
    - We are going to have to store numbers somewhere in memory to make this all work
    - As a convention, to avoid stepping on the program memory, we will use the TOP of the memory space to store the values
    - This is a proto-stack!
        - If we create a calling convention (covered later) we could define a MAX function here
    - INP - get user input
    - STA 99 - Store the first number at position 99
    - INP - get user input again
    - STA 98 - store the second number at position 98
    - SUB 99 - subtract the number at position 99 from the accumulator

        - NB: the value in the accumulator is also the value at 98
        - If we had not stored the second value at 98, it would now be lost (unless we did some math)
    - Now the magic: the accumulator stores the difference between the two numbers
        - If the first number is greater than the second number, it is positive
        - Else, it is negative
    - BRP - branch if the number is positive to the instruction that loads the second number back into memory
    - Else LDA - load the first number into memory and then branch to PRINT
    - LDA - load the second number into memory
    - OUT - print the current number in the accumulator to output and halt
    - Note how the control flow works here, implementing an if/else
- MAX - Questions
    - How would we convert this from a MAX to a MIN computation?
    - How could we avoid storing the second value?
    - Can we avoid storing the first value?
- Practical LMC
    - In this lecture you have seen how to implement some basic algorithms in LMC assembly
    - Using branch instructions, we are able to implement rudimentary loops and conditional execution
    - We only have one working register, the accumulator, so we have to store values in memory if the working set of data is larger than one value

MIPS Assembly:

- Beyond the LMC
- LMC Review
    - Recall the LMC architecture
        - Program counter
        - Instruction and address registers
        - Accumulator register for work
        - input/output areas
        - An ALU
        - 100 memory slots
    - LMC Execution Cycle
        - Check the program counter
        - Fetch the instruction from that address
        - Increment the program counter
        - Decode the fetched instruction into the instruction and address registers
        - Fetch any data needed
        - Execute the instruction
        - Branch or store the result
        - Repeat!
    - LMC Instructions
        - ADD addition

        - SUB subtraction
        - STA store to memory
        - LDA load from memory
        - BRA unconditional branch
        - BRZ branch if zero
        - BRP branch if positive
        - INP get user input, put in acc
        - OUT output acc to output area
        - HLT/COB halt
        - DAT data
    - This is a great teaching tool
    - Reflects the layout of early hardware
    - Modern hardware looks different
    - We will move a step closer to modern hardware with a RISC simulator
- MIPS
    - We now turn to MIPS, a common teaching RISC architecture
        - RISC - reduced instruction set computer
            - Fewer instructions
            - More instructions needed to complete tasks
            - Simpler logic
        - Compared with CISC - complex instruction set computer
            - X86 is CISC
    - We will be looking at the 32 bit MIPS architecture
    - MIPS was introduced in 1985 and is extremely popular in teaching institutions
    - Despite its popularity in academia, it has not been widely adopted in the commercial space
    - Despite not being popular in industry, the architecture has been influential
    - ARM64, in particular, shares a lot of features with MIPS
        - ARM32 did not
    - With the advent of apple’s M chips, MIPS-like ideas are breaking into mainstream computing
    - Differences with LMC
        - MIPS has 32 data registers! vs 1 accumulator
        - Some registers have special functions, so you can’t use all of them
        - 32 bit memory space
        - Many more instructions
- MIPS Registers
    - Registers have a number as well as a name associated with them
    - In assembly programming you use the name
    - $zero is hardwired to zero
    - $at - temporary, used for pseudo instructions
    - $v0, $v1 - function return values
    - $a0-a3 - function parameters (not preserved)
    - $t0-t9-temporary values (not preserved)

        - $s0-s7 - temporary values (preserved)
        - $k0, $k1 - used for interrupt handling by the OS and hardware, DO NOT USE
        - $gp - global area pointer (points to heap memory)
        - $sp - stack pointer (points to stack memory)
        - $fp - frame pointer (related to the stack pointer)
        - $ra - return address (will make sense when we talk about how function calls work)
- MIPS
    - We will be using the MARS MIPS emulator
        - Written in java
        - Provides an editor with some autocompletion
        - Allows you to step through assembly instructions and see what's happening in the computer
    - Similarities to LMC
        - Same basic instruction cycle
            - Fetch instruction
            - Decode
            - Execute
            - Increment program counter register (PC)
        - The PC is visible in MARS UI
        - MIPS is a load/store architecture
        - Data must be loaded into register from memory
        - Mutation operations (+, -, etc…) are performed only between registers
        - Data is then stored back to memory
- MIPS basic instructions
    - MIPS instructions come in three major flavors:
        - R format (registers) - three register arguments
        - I format (immediate) - two register arguments + an immediate value
        - J format (jump) - no register arguments, 26 bits for address information
    - MIPS Instructions
        - LW - load 32-bit word to register
        - SW - store from register
        - ADD, SUB - signed math operators
            - Intermediate versions too
        - MOVE - copy from register to register
            - Move is a pseudo instruction and becomes an ADD under the covers
        - AND, OR, NOR - bitwise logical operators
            - Immediate versions available as well
- MIPS Immediates
    - Immediates
        - I-style instructions
        - Values are encoded in the instruction directly
        - Only two registers are involved in the computation:

        - ADDI $v0, $v0, 1
            - Add one to the value $v0 and save it back to $v0
- MIPS branch instructions
    - Branch instructions in MIPS
        - BEQ - branch on equal
        - BNE - branch on not equal
        - J - jump
        - JAL - jump and link (function related)
        - JR - jump based on the value of a register
- Pseudo-instructions
    - Many common instructions in MIPS assembly do not correspond 1-1 with machine instructions
    - Rather, they are transformed into other instructions
    - These are called pseudo-instructions
    - To the assembly programmer, they look like and act like regular instructions
    - They make working in MIPS assembly more pleasant, without adding additional complexity to the circuitry
    - Examples:
        - MOVE
        - BLT - branch less than
        - BGT - branch greater than
        - …
        - LI - load immediate value
        - LA - load address
    - For example
        - LI $v0, <value> becomes ADDI $v0, $zero, <value>
- MIPS
    - So, a lot more going on, but the basics of assembly remain the same
    - A lot of the complexity here is around supporting function calls, which we will discuss in our next lecture
- I/O in MIPS
    - Note that unlike LMC, we do not have instructions for getting input directly from the user
    - Instead, to do this, we will need to make system calls
    - This is a more realistic model for modern computers
- System Calls
    - A syscall in general is a call to the system (typically the OS) to perform some computation
    - This is typically done with some sort of calling convention
        - A register is used to select which system call to make
        - Arguments are passed via standard registers
            - Recall on MIPS, $a0-a4
- Storing Data
    - MIPS, like most modern assembly, splits code and data off into a separate area

        - .text is a code area
        - .data is a data area
        - Note that labels work as they do in LMC
            - The LA instruction loads the address of a given label
- I/O in MIPS
    - So, here we have a simple “hello world” assembly program
    - We load the intermediate value 4 into $v0
        - $v0 communicates which system call to make
    - We then load the address of the string into $a0, to pass through to the system
    - Finally we invoke the syscall instruction, which passes control over to the system
    - The next line sets $v0 to 10, which is the system exit code and invokes the system once again, which ends the program
    - Not so bad, right?
    - OK, how do we get input?
    - Another syscall
        - System call 5: read int
        - Results are saved to $v0
    - We move this value to a temp register so we can then print out a new string, and then the integer that was entered
    - You can find all the syscalls available in MARS either in the help menu or online
- MIPS Add One
    - Let’s implement some of our LMC programs in MIPS
    - Recall the LMC implementation for “Add one”
    - Pretty much the same as our previous MIPS example
    - We add the ADDI instruction to add an immediate value to the temporary register we are storing the value in
    - A lot more book keeping, but this is much more realistic assembly
- MIPS Count Down
    - Print the numbers 10 to 1 in decreasing order
    - LMC code
    - MIPS implementation
        - Move 10 into $t0
        - Print via a syscall
        - Subtract 1 from $t0 (using an immediate value)
        - Branch back to the loop if $t0 is greater than 0
    - Note that immediates make things clearer
- MAX
    - Max: “ask the user for two number and print the maximum of the two”
    - Recall that this was difficult in LMC because we had two values plus a difference value in play
        - Requires storing working data to memory
- MIPS MAX
    - Ask for two numbers via syscalls

    - The crux is at line 20, where we only move the result of the second syscall into $t0 if it is greater than the current value in $t0
        - Otherwise we jump over to the skip label
    - A little more difficult than LMC, but note that we didn't have to deal with memory at all
    - More registers means things are easier in this regard
- MIPS
    - Ok, that's a brief introduction to MIPS assembly code
        - More realistic model of modern computers (still not perfect)
        - Many more registers than LMC
        - Many more instructions
            - Especially the branching logic
        - Immediate values available with instructions
        - Syscalls for interacting with the system
    - Again, remember, it's just code!

Function Calls in MIPS Assembly: beyond the LMC

- MIPS
    - In the last lecture we looked at an introduction to MIPS
    - It was a little annoying that we had to keep writing all that code out to just print a string, wasn't it?
    - How could we fix that?
- Subroutines
    - Let's take that first print string and rearrange it so that we jump to some logic to print the string
    - We'll define the following calling convention:
        - $a1 holds the address of the string to print
    - We'll label the area we pull the logic out to print_str and we'll jump to it
    - Looks reasonable, so…. What happens?
    - Hmmm ok
    - We just ran off the end of the program and exited
    - We need some way to jump back to where the syscall was made….
    - Ok, let's add a return label and jump back there after the sub-routine has completed
    - It works!
    - But can we reuse this subroutine elsewhere?
        - No the Return address is hard-coded in the function body
    - Ok, so let's make that dynamic: use a register to store the address that we want to jump back to!
    - Turns out there is a register for this, $ra, the return address register
    - This becomes another part of our calling convention: before we call a function we need to push the return address register into the $ra register
    - Note that rather than using a jump instruction (j), we now use a jump register instruction (jr) to jump to the address in the $ra

  - Now we are getting somewhere: we can reuse this subroutine everywhere in our program by placing a label, where?
  - We place the return label on the instruction after the call instruction
  - Well, ok, we are reusing the print_str function
  - But is this any better than what we had before?
    - It’s actually a bit longer
    - That’s too bad
  - Note the pattern:
    - Jump to some other location and then return to the next instruction
  - This is such a common pattern that processors almost always have an instruction for it
  - On MIPS this is the jump and link (jal) instruction
  - The jump and link instruction will push the next instructions address into the $ra register and then jump
    - Program counter + 1
    - Now we are cooking with gas!
  - Note also we are making use of another special function-related instruction: jr or jump return
  - Jumping to the address stored in $ra is so common that it gets its own pseudoinstruction
  - Ok, so using all these tools, we are able to define and invoke a function in our assembly!
  - Doesn’t look like a C-function, but this is how it started!
  - Note that, technically, I am calling this a subroutine
  - Definitions vary, but when I use the term subroutine I mean “code that does not return a value, called only for its side effects”
  - We’ve managed to pretty well encapsulate the function and make it read well in our assembly
- Functions
  - Ok, but what about a more sophisticated function that does more than just call to the OS to print a string?
  - What if we wanted to, for example, actually return a value from a function?
  - Let’s take a look at a sum_to function that would look something like this in C:
    - Int sum_to(int x){
      - Int temp = 0;
      - while(x>0){
        - temp+=x–;
      - }
      - Return temp;
    - }
  - Since we are returning a value from this function we are going to need to extend our calling convention:
  - Ok, so far so good…
  - But how do we return a value from a function?

- Another calling convention:
    - We use $v0 to hold the return a value
- Here we have a function sum_to that sums the numbers from 0 to the given argument
- Note that there is nothing special about the label to sum_to
- It's a procedure simply because we follow the calling conventions
    - Using $a0-3 for arguments
    - Using $v0-1 for return values
- Note a few other things about our procedure
    - We use a temporary variable to keep track of the sum
    - We use the blez instruction - branch if less than or equal to zero
    - Note that we have to move $v0 to $a0 before the system call to avoid stomping on it
- So far, so good
- Note that there is nothing special about the label sum_to
- There is a “function” at this memory location simply because we follow the calling convention we have defined
- Functions aren’t real!
- Let’s look at the function implementation
    - We use a temporary variable to keep track of the sum
    - We use the blez instruction in the while loop:
        - Branch if less than or equal to zero
    - Note that we have to move $v0 to $a0 before the system call to avoid stomping on it
- Let’s run this code and see what happens
- Sure enough, it works!
- This is the assembly version of that while loop based C function we saw earlier
- All stitched together using the calling conventions we defined
- MIPS
    - Function calls are based on calling conventions
    - These calling conventions coordinate the caller and the callee
    - On MIPS:
        - $a0-$a3 are used to pass arguments
        - $v0-$v1 are used to store return values
        - $ra is used to store the return address to jump back to
    - Special instructions exist to assist in implementing function calls:
        - Jal - jump and link: jumps to the given address and saves the next instruction address into $ra
        - Ret - jumps back to the address stored in $ra
- But what if we wanted to implement this as a recursive function?
- How would we do that?
- Here is a naive first attempt
- First move $a0 into a temp register
- If the value is 0, move it into the return value register and return

- If not, subtract one from $a0 and call sum_to again
- Add the result of that to the temp register and fall through to return
- This looks somewhat reasonable, but when we run it, we get an infinite loop, why?
- When we call sum_to recursively, we stomp on the initial value of $ra from the main: code sequence
- We can never get it back
- Gone forever!
- Additionally, every time we call sum_to, it is going to stomp on the $t0 register!
- Complete disaster here folks
- Ok, so what do we need to do?
- > the stack has entered the chat …
- We need a place to store values when function calls are made, and the stack is where we are going to do that
- The stack is just a place in memory that is allowed to grow and shrink according to your needs for a given function call
- All it is in pointer/address
- You move the pointer down in memory to allocate some space for your data
- The stack typically starts at the top of memory and grows down
- A function call allocates a stack frame by bumping the stack pointer by some amount
- This memory area can then be used to store data
  - A return address
  - Local variables
  - Temporary computations
- The stack is crucial for allowing recursive functions
- Note that this doesn’t only mean directly recursive, but also includes indirectly recursive
- MIPS supports the stack concept with the $sp register, which points to the current stack address
- Let’s fix our code using $sp
- The first thing we need to do is bump the stack pointer down
  - The stack grows from the top of memory down on most platforms
- We are going to be storing two registers so we can restore them
  - $t0 - a temporary register
  - $ra - the return address
- MIPS is a 32 bit platform and both registers are 32 bit (1 word or 4 bytes) wide
- In MIPS, as on most platforms, memory is byte-addressed
  - That is, incrementing an address by 1 moves it to the next byte

Recursive Function Calls in MIPS Assembly

- Beyond the LMC
- In the last class we looked at an iterative implementation of the following C-style function:
- We used the following calling convention:

    - $a0….$a3 arguments
    - $v0, $v1 - return values
    - $ra - return address
- Remember functions aren't real: they are just instructions sitting in memory somewhere
- The calling conventions are what make them work
- What if we wanted to implement our algorithm recursively rather than iteratively:
- A first attempt:
- First move $a0 into a temp register
- If the value is 0, move it into the return value register and return
- If not, subtract one from $a0 and call sum_to again
- Add the result of that to the temp register and fall through to return
- This looks somewhat plausible
- However, when we run it, we get into an infinite loop
- Why?
- When we call sum_to recursively, we stomp on the initial value of $ra from the main: code sequence
- That return address is gone forever when we call the jump and link in the body of the sum_to function
- Additionally, every time we call sum_to, it is going to overwrite the $t0 register
- If we called any other functions, those could also overwrite the $t0 register!
- Ok, so what do we need to do?
- We need a place to store all these values when function calls are made
- In order to support recursive function calls (and, in fact, even non-recursive function calls) we need to define another convention
- That convention is called the stack
- The stack is just a place in memory that is allowed to grow and shrink according to your needs for a given function call
- "It" consists of a pointer to a memory address
- You move this stack pointer down in memory to allocate space on the stack
- This is because, conventionally, the stack starts at the to pof memory and grows down
- A function call allocates a stack frame by bumping the stack pointer by some amount
- This "stack frame", which is just some memory you claim for a specific function invocation, can then be used to store data, such as:
    - The return address
    - Any local variables
    - Any temporary computed values
- Again the stack is crucial for allowing recursive functions:
- Recursive functions require that the same named variable be able to hold multiple values at the same time
- Visually:
    - sum_to(3) -> temp = 3
    - sum_to(2) -> temp = 1
    - sum_to(1) -> temp = 0

- Some early languages explicitly forbid recursion because variables were only allowed to have one value at a time
    - This constraint actually made a fortran fast
- Once again, we are going to need some hardware support for the conceptual entity of a stack
- MIPS gives us the $sp, or stack pointer register, which points to the current address at the “top” of the stack
    - Remember the stack is “upside down”
- The first thing we need to do is bump the stack pointer down
    - The stack grows from the top of memory down on most platforms
- What registers do we need to save so that they can be restored after a function call is made?
- Two values get overwritten in the recursive sum_to function:
    - $t0 - the temporary register we are using to store the ongoing summed value
    - $ra - the return address to jump to when the function completes
- Now, MIPS is a 32 bit platform and both registers are 32 bit (1 word or 4 bytes) wide
- In MIPS, as on most platforms, memory is byte-addressed
    - Address “0” points to the first byte of memory
    - Address “1” points to the second byte in memory
- So, we need to move the stack pointer by a total of 8 bytes
    - 4 bytes for the 32-bit temporary value in $t0
    - 4 bytes for the 32-bit return address in $ra
- Hence, we subtract 8 from the current stack pointer
- Ok, next change
- Before we make the recursive call to sum_to, we need to store the values of $t0 and $ra onto the stack
- We do this by calling save word (sw) instruction with offsets from the new stack pointer
- The return address, $r0 is stored at offset 0 from the stack pointer
- The temporary value, $t0, is stored 4 bytes “higher” in memory
    - That's what the funny 4($sp) syntax means
    - You can think of it as stack[4], if stack were a byte array starting at the address $sp points to
- With our temporary values stored safely on the stack, we can then make a recursive call to sum_to
- When a function completes leaving its return value in $v0, we can then restore the values of $t0 and $ra from the stack, using the load word (lw) instruction
- We use the same offsets that we need to save the values, and we should get the original (pre call) values back!
- Magic, everything works now!
- Except.. There is one register we are modifying and not saving… which one?
- What about the $sp register?
- We are modifying it by subtracting 8 from it on line 15 right?
- Aren’t recursive calls also modifying it?
    - Yes they are

- But look down a bit further, just before our return jump
- Note that we are bumping the stack pointer back up the same amount that we bumped it down when we entered the procedure
- So, if our child call bumps the $sp pointer down (it will) then it will restore it before it comes back
- This means that the stack pointer will be the same value as it was before the current invocation of the function occurred
- This is the final component of our calling convention: who is responsible for preserving a register value
- Some registers must be preserved by the “caller”
    - The temporary registers
    - The argument registers
    - The return value registers
    - The return address
- Some registers must be preserved by the “callee”
    - The stack pointer
    - $s0-s7
- In our recursive sum_to() implementation we see both of these cases
- We save $ra and $t0 before making the recursive call
    - Caller responsibility
- We restore $sp before we return from an invocation
    - Callee responsibility
- The code before the function call is made is called the function prologue
- The code after the function call is made is called the function epilogue
- Together the two are sometimes called the function perilog
- As a side note, bumping the stack pointer down is how you can have a StackOverFlow error
- You bumped the stack pointer down too many times and ran out of memory due to too many function calls!
- So we now have a complete calling convention for our MIPS system
    - $a0-a3 - arguments
    - $ra - return addresses
    - $v0, $v1 - return values
    - $sp - stack pointer
    - + the caller & callee saved register conventions
- This is very similar to calling conventions on other platforms such as x86 or ARM
- Once again, important to stress that function calls aren’t real
- It is just an imaginary convention that we use to organize our code
- Nonetheless, software engineering without functions would be impossible
- An important philosophical point: things that aren’t “real” are sometimes as important (or maybe more so) than things that are!
    - Santa claus
- MIPS

- To properly implement the function call fiction on a register-based chip such as MIPS, it is necessary to have a “Stack”
- The stack consists of a pointer to memory where temporary values can be stored during function calls
  - For a given function invocation, the memory it uses is called its “stack frame”
- In order to properly utilize the stack, callers and callees must coordinate with one another to store and restore values at the proper times

Compilers: The link between high level and low level programming

- Getting from C to Assembly
  - Thus far we have been looking at low level computing
    - How to make logical gates out of sand
    - How to build complex logic out of simple logical gates (e.g. an adder)
    - How to drive complex logic with instructions (machine code)
    - How to program machines with low level assembly
  - Most programming today, thankfully, is not done at this level
  - Rather, it is done in a high-level language and compiled to assembly (or bytecode, etc…)
  - The tool used to do this is known as a compiler
  - I teach the compilers capstone course, CSCI 468, where we compile a high level language, CatScript, to JVM byte-code
  - Today we are going to look at the components of a compiler and how exactly C is compiled down to assembly
  - Yes, C is a high-level programming language!
    - May not seem like it, but it is!
- Components of a compiler
  - The major components of a compiler that targets assembly are:
    - scanning/lexing
    - Parsing
    - analysis/intermediate representations
    - Code generation
  - Scanning
    - Scanning also called lexing, tokenizing, or lexical analysis is the process of breaking text into individual tokens
    - This typically involves discarding meaningless white space, comments, etc…
  - Parsing
    - The next step is parsing
    - This is where the stream of tokens are turned into a Parse Tree
      - Also sometimes called an Abstract Tree (AST)
    - The parser can tell us if there is a syntax error
  - Analysis
    - With a parse tree it is then possible to do analysis of the code
      - “Does another variable have this variable's name?”

        - “Is the right hand side compatible with the left hand side?”
- Static analysis
    - Lexing through static analysis is considered the “front end”
    - At right is a diagram of the gcc internals
    - The “ends” of a compiler
        - Front end
        - Middle end
        - Back end
- Intermediate representations
    - In many compilers, the initial AST is transformed into a new tree structure that better captures or represents the final output
    - These new structures are known as an intermediate representations
- Optimization
    - Many compilers have a heavy focus on optimization
        - Producing the fastest possible code that is semantically compatible with the users program
    - Heavily optimized code can often run 3-10x faster than unoptimized code
- Code generation
    - The final step is when the compiler generates code
        - Sometimes called code gen
    - You may think of code gen as, for example, generating Java code
    - In this case, we are talking about generating assembly code
    - With the little man computer and the little man stack machine, you are seeing how assembly can be used to generate machine code
        - Also a kind of “code gen”
    - With a compiler, we are generating assembly
    - Allows us to impose more abstractions on top of assembly!
    - Let’s take a look at how compilation links high level programming to low level assembly by looking at the Godbolt compiler explorer
    - This compiler explorer lets us look at various source programs and how gcc, the GNU compiler, would turn that code into assembly for various target architectures
    - In our case, we will look at MIPS
        - Not exactly like the MIPS we have been looking at, but close!
- Doubling an integer
    - Consider the simple C program at right, which includes a main() method and a simple function, doubleIt() that doubles the value of an integer
    - gcc takes this code and puts it through the aforementioned stages of compilation:
        - It scans it into tokens
        - It parses it
        - It validates/analyzes the code
        - Finally, it generates MIPS assembly
    - Here is the generated code for doubleIt()

    - Let’s go through it line by line
- The first thing the function does is bump the stack pointer down by 24 bytes
    - As with most systems, the stack grows down in memory
    - This provides the function “working space” on the stack to store temporary values
- The next thing the code does is save fp (frame pointer) value out to the stack at an offset of 20
- It then moves the current stack pointer to the frame pointer register
    - All local variables are stored relative to the frame pointer
- The next thing it does is something kind of funny: it moves $4, which is $a0, the argument register, out to the stack
- It then moves the value right back off the stack into $v0
- Next comes a NOP instruction, inserted most likely to byte align the instructions
    - Machines often execute instructions most quickly when they are aligned on some multiple of 2
- Now comes a subtle one: the compiler emitted a shift left instruction, shifting the $v0 register once to the right
- Why?
- Hint: this corresponds with the expression “i + i”
- If you think about it, “i + i” is the equivalent of “2 * i”
- And if you remember from our earlier discussion on shift operators, shifting once to the left is the equivalent of multiplying by 2
- And remember: shifting is very fast when compared with an adder!
- This is an example of a compiler optimization finding its way into the low level generated code
    - On the other hand, what about all that useless data shuffling? And there’s more to come!
- Once the value in $v0 is doubled, it is once again saved out to the stack, now in an offset of 8 from the frame pointer
- But then it is loaded back into $v0!
    - Very inefficient
- We are now ready to return from the doubleIt() function
- We need to move the current frame pointer back to the stack pointer
- Then load the old frame pointer value from the stack
- Then bump the stack pointer back up to its initial value
- Finally, we have a jump register instruction to jump back to where the doubleIt() function was called from
- And a last little nop to keep everything properly aligned
- Things to notice
    - We once again see the symmetry around function calls with respect to the stack

        - Allocate some stack space at the start, save stuff out to it, deallocate the stack space at the end
      - We can see a clever compiler optimization around doubling a value
    - But what about that data shuffling?
      - On the other hand, there is an awful lot of pointless shuffling data from registers to the stack!
      - This could probably be reduced by increasing the optimization settings for our compiler
      - May also be detected by the MIPS hardware and optimized
    - This sort of thing is why early assembly programmers weren't always enamored with compilers: why would you put up with garbage assembly like this when you could just hand code it and make it right?
  - Main: calling a function
    - Now let's look at the code generated for that main function
    - It's a simple body that just calls the doubleIt() function with the value 2
    - We'll skip over the start and end of the function, this is the same rigamarole that we saw in the last function around the stack
    - Instead focus on the instructions on line 22 and 23
    - First, load the value 2 into $a0, the argument register
    - Next, use the magical jump and link instruction to jump to doubleIt() and push the next instruction's address into $ra
    - This is the crux of the function call abstraction, and this is how C function calls compile down to the related assembly
    - It may not look like much, but this is how we get from doubleIt(2) down to something that the computer can almost understand
    - This is one of the pieces of deep magic in this class! Please understand it at least at a high level!
- Compilers
  - Today we looked at how Compilers work, with a focus on the code generation aspect
  - It is code generation that links high level languages like C to low level assembly
  - The compiler also has other phases: scanning, parsing, etc…
  - Take CSCI 468 to learn more about those aspects of a compiler
  - Remember: there is no such thing as a function, but without them we wouldn't be able to build the software systems we do!

The Stack, The Heap & Pointers: C is glorified assembly

- The C mental model
  - You can think of C programs as being along the lines of programs in the LMSM: a giant array of bytes in memory
  - Like on the LMSM, C programs have different segments of memory used for different things
  - The "code" segment (aka Text segment) - the instructions of the program are stored here, typically read only

        - Corresponds to the low memory locations in the LMSM
        - The data & BSS segments - global and static variables are stored here
            - Known, fixed size
            - BSS is zero initialized
        - Similar to the area just after a LMSM program where we stored constants
- Memory segments in C
    - C has different segments of memory
        - The stack - where function parameters, local variables and other function related information is stored
        - The heap - an area where dynamically allocated memory lives
    - The stack - where function parameters, local variables and other function related information is stored
    - Similar to the upper memory location in the LMSM, where the stack pointer register points to
    - The heap - an area where dynamically allocated memory lives
    - Similar to our use of the higher 100s in the LMSM to store temporary runtime data
    - These four segments of memory make up the core content of a C program
    - Memory in C is byte-addressed
        - If you have an address in a register it points to a given byte in memory
    - Of the four memory segments, the two most important ones are the stack and the heap
    - The way each one of these behaves is different than the other, but C doesn't make a major syntactic distinction between them
        - A big source of confusion!
- The Stack
    - Let's first talk about the stack, since we've already been looking at it for function calls
    - Remember how the stack is used in function calls:
        - To store data that might be clobbered during function calls
        - To store return addresses
    - Additionally, local variables and temporary variables in C tend to be stored on the stack
    - The stack is the natural place to store any data associated with a function call
        - Hence, the term "the stack frame": the area where a given function call stores its information
    - In C, the stack is managed by the C runtime, rather than explicitly as in assembly
    - Remember, this is a big reason why C became so popular so quickly: it provided a good abstraction for function calls
    - Remember when we saw the stack pointer being incremented and decremented in assembly?
    - This was establishing the "stack frame", that is, the memory available on the stack for this particular function call

    - When the function completes, the stack frame is deallocated and the CPU jumps back to the return location
    - Here “deallocated” just means: the stack pointer is incremented up and the data below it becomes “garbage”
    - Note that memory for local variables is “automatically” reclaimed in C
    - A sort of garbage collection!
    - You don’t need to worry about allocating memory and deallocating memory for local variables in C, right?
    - The stack is why
    - Note that although we tend to think of stacks growing up, in C (and in most programming environments) the stack tends to grow down
    - This is how it works on the LMSM too
    - A recursive function demo will help demonstrate things
    - Notice that each function call has its own copies of the parameter i and the local variable i_address
    - What do you notice about i_address?
    - The addresses are going down in value
    - Again, this is because in C on x86, the stack starts at a high number and grows downward
    - The debugger in CLion shows the more intuitive upward layout
    - So, that’s the stack, and it’s very cool, it even garbage collects itself
    - If it's so great, why don’t we only use the stack to store stuff?
    - The stack is great and should be used for as much as possible, but the problem with the stack comes from the same places as its strength: it is tied to function invocations
    - What if data needs to live longer than a function invocation?
- The Heap
    - Enter the Heap
    - The heap is another selection of memory that is not nearly as structured as the stack
    - You can think of the heap as a huge pool of memory
    - You can take a handful of memory out whenever you’d like
    - But, and here’s the hard part, you had better put that memory back when you are done with it
    - In C, working with the heap usually means using two functions:
        - malloc()
        - free()
    - Note that these functions are part of <stdlib.h> they are not part of the C programming language!
    - malloc()
        - Here is an example using malloc()
        - We have a pointer to a string in the data segment
        - We allocate a 15 byte chunk of memory from the heap
        - We copy the string into it

    - We print it out
    - Finally, we free the memory
    - Things to notice
        - We had to pass in the amount of memory we wanted
            - What if we under-allocated?
        - The sizeof operator is commonly paired with malloc
            - Returns the number of bytes required by the given data type
    - Since the argument to malloc() is a variable, the size of the allocated memory can be dynamic
        - E.g. Based on user input
    - Note that the value returned by malloc is a pointer
    - We will talk about pointers in a sec
- calloc()
    - Another thing to note is that malloc() makes no guarantees about the memory it handed back to you
        - E.g. it could have garbage data from previous usages
    - If you want data zeroed, you can use the calloc() function
        - Note that you pass two arguments to calloc
        - Worse is better!
- malloc()
    - Alternatively, if you wanted to achieve the same guarantee with malloc, you would need to use the memset() function after you have allocated some memory
- free()
    - Data that you can allocate on the heap needs to be explicitly freed
    - Unlike the stack, there is no stack pointer to clean things up for you
    - If you don't free your data properly, you will get memory leaks
    - An additional danger is that if you free your data more than once, you will get double free error messages
    - In general, this means that when you allocate an object, you need to think about "who" owns the data
        - Who is going to free this?
- Behind the scenes
    - Behind the scenes, a memory allocator is doing the work of finding memory chunks that satisfy your malloc() requests
    - This memory allocator (malloc, get it?) is keeping track of what chunks of memory it has handed out and which have been handed back
    - Underneath it all is just a big chunk of memory "owned" by the memory allocator
    - Note that there isn't any real enforcement of this, it's just convention!
    - This "big chunk of memory" is actually managed by the operating system
    - When the memory allocator needs to grow the heap for more memory, it calls the lower level brk() and sbrk() methods

        - brk() and sbrk() directly manipulate the “program break value” which is the boundary of the heap
        - brk() sets the address to an explicit value
        - sbrk() increments it by a certain amount
    - malloc/free
        - The allocator uses the system calls sbrk() and brk() to ask for more heap memory if necessary
        - Turns out that you can use the brk and sbrk to play directly with memory allocations if you’d like!
            - NB: this can really screw with malloc()!
- Pointers
    - A pointer is a variable that contains the memory address of something
    - Pointers are difficult for many beginners to understand because they aren’t close to the hardware
    - But we still have pointers in LMSM
        - The stack pointer!
    - Pointers aren’t so bad as long as you have low level experience
        - In fact, for assembly programmers, they are obvious: how else would you refer to a big chunk of memory?
- Pointers in C
    - C has good support for pointers, but, unfortunately, they made what I consider some bad syntactic choices
    - The * character is used to declare and also to dereference pointers
        - Very confusing!
    - The function at right shows both uses of the * operator
    - Sometimes we want to go the other way: given a thing, get a pointer to that thing
    - For this we use the & operator
    - Note that we can get the address of anything
        - A stack-allocated variable
        - A heap-allocated variable
        - No difference!
    - What are the advantages of pointers?
        - They are small (just an integer)
        - You can pass around a reference to a large chunk of data, rather than the data itself
        - You can do pointer arithmetic
        - They are what malloc() returns :)
- Arrays in C
    - Arrays are usually easier for people to understand in C
    - Arrays are usually stack-allocated in C
    - Familiar array-indexing syntax we are all used to
    - Note that arrays in C do not encode their length
        - You have to know how large an array is to iterate over it properly
- Pointer/array correspondence

    - Pointers and arrays are deeply related to one another
        - This becomes apparent as you delve into pointer arithmetic
    - You can add to and subtract from pointers
        - This is known as “pointer arithmetic”
        - + and - operations bump the address n “slots”
    - In C, an array can be assigned to a pointer
    - The value of the pointer becomes the address of the first value in the array
    - However, pointers cannot be assigned to arrays
        - Nor can arrays be returned or passed into functions!
    - These restrictions are part of the reason why pointers tend to be used so heavily in C: they are just more flexible and allowed in more situations, and you can always convert an array to them easily
- Pointers
    - Note again that you can get a pointer to anything
    - This means that it can be unclear what the lifetime of the underlying data being pointed to is
    - Languages like Rust try to make this more explicit by enforcing a single “owner” for the underlying data
- Review
    - A C program consists of different segments of memory
    - Of these segments, the stack and the heap are the most important
    - The stack grows and shrinks automatically as functions are called
        - Memory allocated on the stack share the lifetime of the function
    - The heap grows and shrinks based on calls to malloc() and free()
        - Memory allocated on the heap has no particular lifetime, it must be managed explicitly by the programmer
    - Pointers are just memory addresses “pointing” to actual data somewhere in memory
        - This data can be anywhere: the stack, the heap, the BSS, etc…

Reintroduction to C: finally some code

- Today’s lecture:
    - A history of the C programming language
    - Why did C win?
    - Why are we still using C?
    - How is C easy?
    - How is C hard?
- The C programming language
    - Invented by Dennis Ritchie in 1972
    - Successor to the B programming language
        - Yes, there is an unrelated D programming language
    - Designed to help build the original Unix OS
    - Dennis Ritchie died in 2001

    - “Ritchie was under the radar. His name was not a household name at all, but… if you had a microscope and could look in a computer, you’d see his work everywhere inside”
- Hugely influential language
- A sampling of languages derived from it:
    - Java
    - C++
    - JavaScript
    - TypeScript
    - D
    - C#
- What was the big idea with C?
- Programming in assembly was hard especially as the program size grew
    - No notion of variables with names
    - No notion of scoping
    - Implicit “parameters” and lots of busywork around function calls
- C provided a lot of nice structures on top of assembly:
    - Easy function definition
    - Easy function invocation, including argument passing
    - Scoped local variables
    - Implicit stack pointer and register management
- C rapidly took over in place of assembly for general purpose development
- Assembly was relegated to special situations like embedded or extremely perf-sensitive work
    - Recall what the generated code we looked at last time looked like
- C, invented in 1972, is still widely used in industry
- Linux kernel development in done in C
- Many embedded systems
- Many core server systems
    - DBMS
    - Web Servers
    - DNS
- Were there any other alternatives?
- Yes! A big “other” language in the 70s and 80s: Lisp
- Lisp is
    - Garbage collected
    - Has a much larger/nicer standard library
    - Memory safe
    - Didn’t require header files
    - Etc…
- Lisp was a far more advanced programming language
- Very popular in early AI community due to flexibility and metaprogramming
- So why didn’t lisp win?
- A Lisp enthusiast, Richard Gabriel, tried to make sense of the situation

- A portion of his paper has come to be known as the “Worse is Better” argument
- In this paper, he contrasts two styles:
    - The MIT/Stanford approach
    - The Bell Labs/Berkeley approach
- MIT/Stanford focuses on “the right thing” no matter the complexity cost
- New Jersey/Berkeley focuses on simplicity of implementation
- Contrasts the styles along four possible design goals:
    - Simplicity of implementation
    - Correctness of implementation
    - Consistency of system
    - Completeness of systems
- MIT/Stanford
    - Simplicity - the design must be simple, both in implementation and interface. It is more important for the interface to be simple than the implementation.
    - Correctness - the design must be correct in all observable aspects. Incorrectness is simply not allowed.
    - Consistency - the design must not be inconsistent. A design is allowed to be slightly less simple and less complete to avoid inconsistency. Consistency is as important as correctness.
    - Completeness - the design must cover as many important situations as is practical. All reasonably expected cases must be covered. Simplicity is not allowed to overly reduce completeness.
- New Jersey/Berkeley
    - Simplicity - the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.
    - Correctness - the design must be correct in all observable aspects. It is slightly better to be simple than correct.
    - Consistency - the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.
    - Completeness - the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must be sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.
    - Gabriel argues that despite his biases, the worse is better has better survival characteristics
    - A wonderful example of a person reasoning against their priors
    - Wrote a series of papers essentially arguing with himself as to whether worse is better

        - I agree with that in some cases and certainly in the case of C, worse is better
        - Simplicity of implementation is an important design consideration and other considerations may need to be subordinated to it
            - “80/20” thinking
    - C was/is “worse is better”
    - It won
    - Nothing has really replaced it for systems programming
        - Maybe Rust?
        - Maybe
    - Deal with it…
    - C is a link between high-level programming concepts, like functions and low-level hardware concepts, like registers
    - It abstracts away the machine instructions, letting programmers work at a higher level of abstraction
    - At right is the canonical “hello world” program in C, compiled into MIPS assembly
    - Note the explicit stack management and register juggling in the assembly
    - Compare with the simplicity of the C program
    - Easy to see why C got so popular!
- Macros in C
    - What about that #include at the start of the program?
    - In C, #include is a macro
    - A macro is a bit of code that is executed at compile time
    - The result of that code execution is then substituted for the macro text and then compilation begins
    - #include angle brackets vs. strings
        - #include <foo.h> //look for foo.h on system path
        - #include “foo.h” //look for foo.h in the current directory, then the system path
    - At the top of header files you often see this #ifndef macro
    - Why?
    - Again, in C, macros are straight string substitution, so the header files are directly injected into the C file
    - If two header files referred to one another without this check, you’d have an infinite loop of inclusion
        - Worse is better is worse?
    - You can define your own macros in C using the #define macro
    - Here we create a square() “method” using #define
    - This will string substitute x*x for x
    - A faster square function since the logic is inlined!
    - What does it print?
    - Oops, the problem is that string substitution here gives us 2 + 2 * 2 + 2
    - Which binds as 2 + (2 * 2) + 2
    - Macros are dangerous and can lead to very hard to understand bugs and code

        - Takeaway: Macros are dangerous and can lead to very hard to understand bugs and code
- Header files
    - The header file concept is important in C because, in C, you cannot use a function or variable unless it has been declared
        - NB, this doesn't mean defined, only declared
    - Header files allow you to declare features so other files can refer to them
    - There is nothing magic about header/.h files: they are just a convention built on top of the fact that C lets/requires you to split declarations and definitions up
    - Worse is better
    - There is no name-spacing in header files or .c files
    - One giant global namespace
        - Compare with java, where you have packages
    - Prefixing functions is common to achieve namespacing
    - Worse is better
- Data Types in C
    - Common C data types
        - Int - usually means a 32 bit integer
        - Long long - a 64 bit integer
        - Unsigned long long - an unsigned 64 bit integer
        - Char - 8 bit integer, represents a character (usually)
        - Float - 32 bit floating point decimal number
        - Double - 64 bit floating point decimal number
    - C is relatively permissive in allowing casting between different data types
    - What do you think this code prints?
    - JavaScript is easy right?
        - JavaScript in contrast with C, only has one core numeric type: Number
        - Easy, right?
- Function definitions
    - A thing to really appreciate about C is how clean and understandable function declarations are
    - C is a functional programming language
        - Not a fancy one, like Haskell or OCaml, but still
    - The syntax has been hugely influential:
        - The function return type, followed by the function name, followed by an argument list
        - Arguments are comma separated and declared the familiar type name format
        - Curly braces around function body
    - This syntax influenced many later languages (e.g. Java)
- C standard library
    - The code at right also demonstrates calling a library function, printf()
    - This was another big contributor to the growth of C: especially on Unix, it came with a standard library for things like I/O

  - Seems boring today, but a big deal back then
  - Let’s take a look at the signature for printf()....
  - Uhhh, what?
  - This method signature shows how hard C can be…
  - Ignore the extern
  - Printf takes a pointer to a char
  - Const - “compiler, I will not assign a new value to this variable name”
  - _restrict - “Compiler, I will only use this pointer to access this data”
  - This is definitely worse is better stuff
  - A “good” language would abstract a lot of this away
    - E.g. provide a good string abstraction
  - But say what one will, C won and has shown tremendous staying power as a systems language

Strings, Structs, & Memory Segments: what’s really going on in C?

- In the last lecture we talked about the various memory segments of C
- Each has its own behavior and requirements for use
- Often things that look the same will act differently
  - Pointers to stuff, in particular: which segment is the pointer pointing to?
- Strings & Structs
  - This is particularly important when dealing with two types of data in C:
    - Strings
    - Structs
  - Strings
    - In C Strings are very raw and low level
    - They are, in reality, just a one-dimensional array of characters, terminated by a null (zero) character
    - Each character is 8 bits
      - Aka 1 byte
    - Here we have a five character string (sometimes called a “C String”)
    - These characters will be laid out in successive memory locations, just like an array
      - [‘h’, ‘e’, ‘l’, ‘l’, ‘o’]
    - Actually, I’m lying, there are six characters in this string
    - Why?
    - A hidden null or, literally, the value 0 (zero) terminate the string
    - Note that each address increased by one
    - Recall, C is byte addressed so a given raw address value (that is, a pointer) in memory refers to a specific byte (not, say, a 32 bit value)
    - Note that the string literal in C returns a pointer to a char
    - What memory segment does this pointer point to?
    - Is this string mutable?
      - E.g. Can we change “H” to “J”, and make it “Jello”?
    - Nope, this string is living in the data segment, so it is read only!

- If we want to mutate this string, we need to make a copy of it somewhere mutable
    - The stack (kinda hard)
    - The heap (where you'd normally do this)
- Ok, so let's use calloc() to allocate some memory on the heap, copy the string out to that, and then mutate it
- Hey, it works! Great!
- And, like good citizens, we free the memory afterwards
- Uh, wait a sec, how long did we say that the string "Hello" is?
- And we only allocated 5 bytes with calloc()
- But it still worked?!?
- Yep, C doesn't care
- It wrote right off the end of that memory segment you were allocated
- This didn't even cause an error
- If something else had been living next to the memory location we allocated, this would have stomped right on its value in memory
- C doesn't care
- The numeric value of the pointer here increases by one because the type of the pointer is char, which is 1 byte wide
- If the pointer had, instead, been to a 4 byte integer, the value of the pointer would have increased by 4
- Again, pointers are just addresses to locations in memory that can be dereferenced to get at the actual underlying data value
- Strings in C are uniquely terrible to work with, when compared with higher level languages
- Very easy to screw things up
    - Accidentally writing off the end of a char*
    - Accidentally handing out a char * reference to an immutable memory location
    - Etc…
- How often in life do you know exactly how large a string is going to be?
- Programming languages like Java provide abstractions like the String class, or the StringBuilder class, for easier manipulation/creation of strings
- C, instead, provides functions for string manipulation in the string.h standard library
- strcpy(), strcat(), etc…
- The safe versions of these functions start with strn and take an n, which is the max length to copy/index over in the first string
- Broadly, these are a huge pain to work with when compared with strings in other languages
- When I have to work with Strings in C, the first thing I do is create a small string library that hides all this away internally
    - I usually call it str_buf for String Buffer

    - You are going to need to deal with C strings in the next part of the project
    - In particular, you are going to need to tokenize assembly
        - (that is, split it up into chunks of text based on whitespace)
    - The standard function for doing this in C is strtok()
    - The way it works is that you pass a string in to tokenize, plus a separator, and it returns a pointer to the first token
    - You can then get the next token by passing NULL into strtok, which tells it "give me the next token of the last string you tokenized"
    - Wait, what?
    - Yes, strtok() keeps a global variable around pointing to the last string tokenized
    - Even worse, it mutates the original string passed into it, so this code actually segfaults!
    - We need to make a copy of the string to tokenize it!
    - Here is the fixed code, again we need to copy the string literal out to somewhere in memory that is mutable
    - Of course here it is easy to know that str is immutable
    - But what if it was an argument?
        - Hard to know!
        - Best to make a copy defensively, right?
    - This is showing how C is hard: there are a lot of hidden gotchas and semantics that you need to keep straight
    - I think this is a great example where object oriented programming could hide a lot of complexity from the end user
    - Ok, so, C-Strings kinda stink, but they are ubiquitous in low level programming and you'll need to work with them a bit in the next checkpoint
    - Sorry
- Structs
    - Moving on to Structs
    - Structs are the closest things to classes that C has, but they only hold data
        - It is actually possible to store function pointers in them, and some people emulate OO programming this way
    - Broadly, in large C programs, you have important structs as the first argument to a set of functions, often prefixed with the same name to make clear they all "belong together"
        - Again, I like OO programming, despite the bad name it has in some circles
    - Structs are defined with the struct keyword
    - If you do not create a typedef (type definition) for the struct as well, you have to include a struct keyword when you declare them too
    - If you include a typedef with the struct definition, then you can declare things in a more natural manner

        - It is very common to pass around pointers to structs
        - To access a field on a pointer to a struct, you need to dereference the pointer, then use the dot(.) operator to reference a given field
        - Note that structs can very easily be created on the stack
        - The first line allocates a book struct on the stack
        - The third line allocates a book struct on the heap
        - Note that C makes no distinction between pointers to these two memory locations
        - We could return either of these pointers from this function
        - Very dangerous if we return the first pointer!
        - In the second case, we have to determine who deallocates the struct!
        - C is hard
        - On the other hand, we could also pass these structs into functions
        - This is actually probably safe in both cases
            - The called function will live as long as the current function, so the stack data stays valid
            - Unless the called function stores the pointer out to a global value
        - C is hard
        - C calling conventions are pass by value
        - If you pass a struct into a function, it will make a whole copy of the struct
        - If you pass a pointer to a struct into a function, it will reference the original struct
        - Demo
- Review
    - C strings are a sequence of byte-sized ASCII characters, terminated by a null (0) character
    - C strings are touchy and error prone
    - You are going to need to use C strings in the next checkpoint, along with strtok(). Sorry.
    - Structs provide a way to define simple data structures in C
    - Structs contain only data, they are not objects like you are used to in Python or Java

More Low Level C: Welcome to the Machine

- In the last two lectures, we've been looking at relatively low level C
- Again, to program in C well it is necessary to have a good mental model for how a computer works
- A major goal of this class is to give you that model
- Review:
    - Remember, in C, memory is effectively treated like a huge byte array
    - That memory is divided up into segments
    - Each segment has its own quirks and behaviors
    - Consider the following code
    - Where does
        - Ptr1 point to?

        - Ptr2 point to?
        - Ptr3 point to?
    - What does assigning arr2 to ptr2 do?
    - Understanding which memory segment each of these pointers ends up pointing to and how that memory segment behaves is important for developing a good understanding of C
        - And for getting a good grade on the next quiz :)
- Bit manipulation
    - Again, C is very low level and you can treat nearly anything as just an array of bytes
        - Recall 1 byte is 8 bits
    - C also allows you to manipulate individual bits in memory via bitwise operators
    - Why on earth would you want to manipulate individual bits in memory?
    - What if you needed to store a bunch of booleans?
    - How many bits does a boolean take up, in theory?
    - You can efficiently store up to 64 booleans in a single long long (64-bit) integer!
    - Much more memory efficient than, say, 64 32-bit integers!
    - These sorts of things matter in embedded programming, etc…
    - Bit manipulation is based on simple boolean logic, where 1 is true and 0 is false
    - Recall our logical operators:
        - NOT
        - AND
        - OR
        - XOR
    - When applying a bitwise operation to a piece of data, each bit will have the given logical operation applied
    - So, in the case of a bitwise not, each of the bits in a given piece of data are inverted
    - Here is a single byte of data, w/ the hex value 0x2A (42) is inverted using the bitwise not operator, ~
    - For each bit, if the bit is 0, it becomes 1, if it is 1, it becomes 0
    - The resulting value is 0xD5 (213)
    - Another common bitwise operator is the bitwise AND, which uses a single ampersand (as opposed to the logical AND, which uses two ampersands)
    - Unlike the bitwise not, which is a unary operator, this is a binary operator
    - For each bit in both pieces of data, if both bits are 1, set the resulting bit to 1, otherwise set to 0
    - Here the two pieces of data only have only one bit where both are set to 1, so the resulting data has only one bit set to 1 (giving the decimal value 2)
    - Finally, there is one more important bit manipulation that doesn’t involve traditional logic: shifting
    - Shifting moves bits to the left or to the right some number of times
    - Here we have a left shift being applied using the left shift operator, <<

    - The shift operator takes a right hand argument that indicates how many bits to shift over
    - In this case, we are shifting everything over two bits
    - Recall that we saw that the gcc MIPS compiler emitted a shift operator for a multiplication
    - Again, this is because, in binary, a left shift is a multiplication by 2
        - Just like in base 10, a left shift is a multiplication by 10
    - Ok, so how is this all used to store individual booleans in larger data types?
    - Let's store 8 booleans in a single byte
    - Let's let the first bit indicate if a piece of data is Case Insensitive
    - The first step is to define a constant in C, CaseInsensitive, that has the left hand bit, and only the left hand bit, set to 1
    - This is the binary value 0b1 or just 1
    - Now given a byte, we can test if the byte is 1 or 0 by using the bitwise AND, the bit mask, and the input byte:
        - if(data & CaseInsensitive){
            - …do something
        - }
    - If we want to set the bit in data to 1, we can use the bitwise OR and that constant:
        - data = data | CaseInsenstive
    - Finally, if we want to clear the byte, we can use a bitwise AND and a bitwise NOT
    - data = data & ~CaseInsensitive
    - Given these three tools, you can now "pack" N booleans into a datatype of size N bits
    - Still occasionally used even in high level programming languages like Java
        - See java.util.BitSet for example
- Function pointers
    - C supports a notion of function pointers, that is, pointers to a given function in memory
    - The declaration syntax is pretty crazy:
        - The return type of the pointer
        - The name of the variable prefixed by a * enclosed in parens
        - The types of the function args
    - In this case, this says: a pointer to a function that has a void return type and no arguments
    - You can see that both func1() and func2() satisfy this type of signature
    - You can assign either to the funcPtr variable
    - You can then invoke the functions by calling the funcPointer
    - Question: which memory segment does the funcPtr variable point to?
    - Function pointers are useful in situations where you want a dynamic callback
    - A good example of that is the C threading library, pthreads
    - If you want to start a new thread of execution, you pass a function pointer into a function, pthread_create()

    - Here we pass the run_thread() function into the pthread_create() function, and it will be invoked in a new thread
    - We are not going to look at multithreading in C beyond this, we will look at it in Java
        - Much better implementation IMO
- Assembly in C
    - Recall that C has been called “glorified assembler”
    - I don’t really agree with that characterization, but there is some truth to it
    - There are two primary ways to interact with assembly from C
        - Include a reference to an assembly file
        - Inline assembly
    - Here we have a C file referring to functions defined in an assembly (asm) file
    - The symbols are defined as extern, which tells the C compiler “these things are going to come from elsewhere, just assume they will be there”
    - Later, a linker will link up the function calls in the C with their definitions from assembly
    - Here is what the add() function in the assembly file looks like
        - Note that it is defined in the text segment
    - This label will be what is used by the linker later on in the compilation process to figure out where to jump when the add function is invoked
    - Note that it is the assembler programmers responsibility to implement calling conventions properly!
    - Very easy to screw things up!
    - Another way to invoke assembly is to use inline assembly
    - The specifics of inline assembly very much depend on which compiler you are using
        - And, obviously, what architecture you are targeting!
    - At right is a gcc example that does a register-based add instruction (maybe to avoid the traffic with the stack we saw when we were looking at compiler output)
    - asm (instructions
    - : output operands
    - : input operands
    - : clobbers);
    - In this case, we have a simple add instruction
    - The res value is the output and is stored in a register
    - a & b should be stored in a register too
    - Obviously a niche need, but useful at times:
        - Overcome “bad code generation” by a given C compiler
        - To invoke features on a given CPU only available through assembly
- Review
    - Again, know how the various segments of memory work in C
    - We took a look at bitwise manipulation and, in particular, how to “pack” 8 bits into a byte of data so we could store 8 booleans in a byte
    - Function pointers allow you to pass functions to other functions

        - Useful for callback situations like when creating threads
    - Finally, we looked at how to integrate assembly code into C projects

Operating Systems: what do they do?

- The operating system
    - Early computers did not have an operating system
    - Programs were run “raw”, directly on the hardware
    - Did one “task” and then exited
    - Similar to the way the Little Man Computer simulator worked, but done with punch cards
    - Machines eventually started supporting “libraries”
    - Fixed location subroutines that could be jumped to to execute common needs
    - E.g. input/output via interrupts
    - As computers became more powerful, it became plausible that they might do more than one thing at the same time
    - Software needed to be created to manage multiple programs running on hardware concurrently
    - The “first” modern operating system according to most people was the atlas supervisor
    - Created for manchester universities atlas computer
    - As the name suggests, the term “operating system” was not yet a thing
    - The manchester computer was relatively sophisticated, being the first computer to support “virtual memory”
    - The atlas supervisor kept multiple programs running on the manchester computer, at the same time
    - Atlas supervisor:
        - “The most significant breakthrough in the history of operating systems.”
- What does an OS do?
    - The operating system grew out of systems that managed multiple batch processes at the same time
    - So the original activity of the operating system/supervisor was time sharing
- Time Sharing
    - The operating system grew out of systems that managed multiple batch processes at the same time
    - So the original activity of the operating system/supervisor was time sharing
    - That is, multiple users can be using the system at the same time
    - How can this be achieved?
    - What is a program, at the lowest level?
    - A collection of binary data in registers
        - Especially the program counter
    - To “switch” between user1 and user2
        - You need to “save” all of user1’s memory… somewhere
        - You need to “save” all of the registers for user1
        - You need to load user2’s memory and registers

    - Note that this is the exact same situation when you want to support multiple processes
    - We need some way to “suspend” a process by saving all its state, and then restoring it
    - This probably seems easy enough for registers
    - Just copy their values out to a “special” memory location somewhere and then restore them from that location
    - But what about memory?
        - Do we really want to copy all of the memory for a process or user out to some other location?
- Virtual memory
    - No, that’s crazy
    - Instead, modern hardware has what is known as virtual memory
    - Recall that, in a C-program, it looks like you have access to all of physical memory
    - This is a lie
    - Instead, what look like real, physical address to your C program are virtual addresses
    - These virtual addresses are translated into physical addresses via a data structure known as The Page Table
    - The Page Table is typically integrated directly into hardware for speed: the translation lookaside buffer or TLB
    - The operating system is responsible for saving this (much smaller) TLB out to memory, along with registers
    - This allows the OS to suspend processes relatively efficiently by only storing out a small amount of data to disk
    - Note that the OS does have a raw view of memory: it can write anywhere it wants in the system
    - This leads us to another feature of the OS
- Isolation
    - Modern CPUs have a notion of “modes” or “rings”
    - Various modes are allowed to perform various operations
    - By disallowing certain operations in certain modes, programs and users can be isolated from one another
    - So, a program might be run in Ring 3
    - In this ring, it is unable to manipulate, for example, the TLB or raw memory
    - The kernel (that is, the operating system), however, is able to manipulate the TLB
    - So how do these two programs work together?
    - Recall how we did a print in MIPS?
    - We issued a syscall instruction, which causes an interrupt
    - What happens here is that the OS “takes over”:
    - The CPU jumps to a special bit of code set up early on by the OS called the interrupt handler

    - This very special code begins the process of saving off the state of current processes
    - When it is done saving off the state of the process, it can then do some logic, driven by the value found in the v0 register
    - In this case, it prints a string to standard out
        - More on I/O in a bit
    - When it is done performing the given task, it can restore the process by restoring its registers and TLB and then resuming ring 4 operation
    - All of this requires significant hardware support
    - The OS and the hardware are closely bound together
- Scheduling
    - Now, of course, when the OS is done, it doesn't have to restore the process that called it: it just needs to note the value it is going to return (if any)
    - It could, instead, put a different process on the CPU
    - Eventually it will restore the process that called print, and it will look, to the process, like it just called the OS and got a result
    - But the other processes may have been run in the meantime
    - If the activity that the OS was asked to do is long running (e.g. requires I/O with disk or the network) it might be advisable to run a different process while that work is ongoing
- Preemptive scheduling
    - If the only time that we could switch processes was when a program made a syscall, what problem could arise?
    - What if you had an infinite loop in a program that didn't make a syscall?
    - Mac OS9, Windows 98, etc… all suffered from system hangs
    - Applications that were coded incorrectly could freeze the entire computer because the OS didn't have an opportunity to suspend them
    - The switch to OSX and Windows 2000 fixed this by introducing preemptive multitasking
    - In addition to syscalls, a timer integrated into the hardware triggered an interrupt to the OS, allowing the OS to suspend the current activity
- Input/Output
    - Another important feature of operating systems we will discuss is input/output
    - The OS typically contains code for working with various I/O devices
    - This includes interrupt handlers to deal with things like keyboard presses
    - Again, hardware is key here
    - Interrupts occur via
        - Disk returning data we asked for previously
        - Input from the keyboard
        - A clock interrupt
        - A printer notifying us that a print job is finished
    - The operating system is responsible for handling all of these interrupts
    - Additionally, the Operating System is responsible for sending data to these devices

        - This is typically accomplished via drivers
        - Drivers present a common interface to the OS that issues device specific instructions for a particular device
        - Common Input/Output
            - Printers
            - Keyboards
            - Mice
            - Monitors
            - Network
        - OS's often provide a sophisticated network stack
            - TCP/IP support, etc..
    - Graphical Interfaces
        - Early on, all interaction with computers was via text/teletypes
        - Operating systems provided functionality to make it possible for multiple teletypes to connect to a single system
        - Over time, graphical user interfaces became the standard for most desktop operating systems
        - Servers (e.g. systems running in the cloud) often do not have a graphical interface
            - Still need to be interacted with via the command line/text

Virtual Machines: the JVM

- The JVM, as the name suggests, is a virtual machine
- It emulates a simplified computing machine that can interpret Java bytecode
- Java bytecode is just a set of simple instructions, not unlike a simplified version of assembly code
- The JVM fundamentally operates on three types of values
    - Integers
        - Various sizes
    - Floating point
        - Various sizes
    - References
        - These are…pointers!
- The JVM has ClassLoaders, whose job is to load, well, classes, specified in the java bytecode format
- These classes can define methods, fields, etc… and the JVM provides infrastructure for executing the classes logic
- Garbage Collection
    - One of the wonderful features of the JVM is that it is garbage collected
    - References are occasionally scanned from the GC root references to determine if an object has anything pointing to it
        - If not, the object and memory are reclaimed
    - A GC root is something reachable within the current execution context
        - Static stuff
        - Things pointed to by the stack

- The JVM uses a “Mark and Sweep” algorithm
    - Start at the current GC roots
        - Mark all objects referred to recursively
    - Sweep all of memory, reclaiming anything not marked
- This is a simple and well understood algorithm
- It was noticed that most objects had either very short lives or very long lives
- This insight was used to create a mixed garbage collection mechanism
    - Objects begin in one area of memory
    - When they survive for a while, they are moved to another
- The two areas have different gc algorithms, tuned for their behavior
- This is called generational garbage collection
    - The shorter lived area uses an algorithm called stop-and-copy which is more efficient if you don’t have many live references

- Data types
    - Again, the JVM has three fundamental data types
        - Integers
        - Floating point
        - References
    - Note that there is no boolean
        - Booleans are represented with integers
        - chars/bytes - same
    - Integer and floating point values are called primitive values
    - All other stuff (including arrays of primitives) are reference values
    - This is the biggest distinction in the JVM: between primitives and references
    - In GC, primitives are ignored, references are traced
    - Most interesting things are references:
        - Strings
        - Lists
        - Etc..
    - Note that under the covers, reference values are pointers
    - When you call a method with a string parameter, you are really passing a pointer to that string in memory, not making a copy of the string
    - Java “hides the pointer”
- Autoboxing
    - The java has reference types that correspond to the various primitive types
        - Int - java.lang.integer
        - Char - java.lang.character
        - …
        - Float - java.lang.float
    - An interesting aspect of the JVM is that it supports autoboxing and unboxing
    - Here we have a primitive int stored in x
    - Yet we are allowed to assign it to a reference value object
    - Under the covers, java will convert x to a java.lang.integer
    - Convenient!

    - Also expensive…
    - This means you can kind of use primitives with things like lists
    - Here each primitive int autoboxed into an integer and added to the list
    - Spot the NullPointerException…
    - Int1 is a reference type, java.lang.integer
    - When it is used in a mathematical operation, it is automatically unboxed
    - If the reference is null, you get a NullPointerException!
- JVM architecture
    - The JVM runtime consists of
        - A stack per thread
        - A program counter per thread
    - The non-heap
        - Constants, code, strings, etc…
    - The heap
        - Split into "young" and "old" objects
- The Stack
    - Within a stack, you have frames, which correspond to function calls
    - A frame has
        - A return value
        - Local "slots" for variables
        - A reference to the current class
        - An operand stack
    - A method will shuffle variables to and from the local variable slots and use the operand stack to do computations
- The Operand Stack
    - The operand stack is used to do expression evaluation
    - Int x = 1 + 1 ends up looking like this
        - Op stack starts empty: []
        - Push 1 - [1]
        - Push 1 - [1, 1]
        - Add - [2]
        - Store to slot for x - [ ]
- Non-Heap Memory
    - This area contains things like strings, runtime constants, etc… as well as the class data
    - Class data includes constants, field references, method references, etc… as well as the actual bytecode implementation of methods of the class
    - At runtime, objects have pointers to their class data in order to properly look up methods, fields, etc…
- Heap Memory
    - The heap is where objects live
    - The JVM does not operate directly on objects, but rather operates on references
    - The stack never has an object on it, only ints, floats and pointers/references
    - Note again that the JVM has a generational garbage collector

    - The young generation has a faster gc algorithm, tuned for low survival rates
    - GC of the old generation happens infrequently, using traditional mark-and-sweep
- JIT compilation
    - Java started as a pure virtual machine
        - Basically an emulator
        - It was slow, especially on hardware from the late 90s
    - To address this, Sun Microsystems introduced JIT compilation under the name Hotspot
    - JIT compilation is when bytecode is, itself, compiled further into machine specific code
        - This is done at runtime and as needed
    - Can produce extremely fast code, but does incur some runtime cost
    - Generally, java will be about 20-50% slower than equivalent C
    - However,
        - Startup times will be significantly worse due to JVM overhead, JIT, etc…
        - Memory usage will almost certainly be worse as well
- What does JVM bytecode look like?
    - Consider the java class at right
    - This code is going to be compiled to bytecode for execution on the JVM
    - Just like C is compiled to assembly then assembled to machine code
    - My first bytecode
        - Ok, here we have our first step into the world of bytecode
        - What's going on?
        - Let's go line by line
        - Our frame (the data for the current method executing) has a few different data stores
            - Local variables (slots)
            - An operand stack
        - ILOAD 1 is saying "push the integer value in the second slot onto the operand stack"
            - Slots are base 0
- X86 and Java
    - Recall, the reason we are all using glorified cash register chips is due to the failure of the iAPX 432 chip
        - No user-facing registers
        - Stack machine
        - Hardware support for garbage collection, object oriented programming
    - The chip failed, but many of these ideas made their way into the JVM
    - And also into LMSM!
- The JVM
    - The JVM provides a virtual machine on top of physical machines
    - The language of this virtual machine is byte code
    - Conceptually very similar to assembly, but more advanced
    - It's turtles all the way down!

Concurrency: doing multiple things… at the same time

- Concurrency is the term we use for when logical operations overlap in time
- Concurrency is found at many layers in the computer
- The most obvious instance of concurrency on computers is application concurrency
- We all run multiple applications at the same time on our computers, phones, etc…
- Even on a single CPU with a single core, multiple programs can be run concurrently
- When one program blocks an I/O operation (e.g. disk or network activity) another can be run
- There are three main mechanisms for building concurrent programs on modern CPUs:
    - Processes
    - Threads
    - I/O multiplexing
- Processes
    - The simplest way to create concurrent programs is via processes
    - In C, this is accomplished with the venerable fork() function
        - __pid_t is an integer value, the process id of the newly created process
    - __pid_t value
        - Negative value: unable to create a child process (out of memory, etc…)
        - Zero: this is the newly created process
        - Positive: this is the original process, the pid is the Process ID (PID) of the child process
    - Both processes have a complete copy of memory
    - This includes any files, sockets, etc… that are open
    - The parent process must close any resources (e.g. open files) shared with the child or risk leaking them
    - The memory spaces, however, are separate
    - No shared memory between the processes, so no chance of corrupting one another's memory
    - But also no way to communicate easily with other processes
    - Process pros
        - Simple
        - Time tested
    - Process cons
        - Heavyweight
        - Awkward to communicate between processes
- Threads
    - A more advanced technique is to use threads
    - Threads are a lightweight alternative to processes
    - Threads share the same memory space
    - The Java Threading library is a mature, object-oriented API for working with threads
        - Much nicer than the standard pthreads API available in C
    - To create a new thread, create a new Thread() object and pass in a runnable object

    - The runnable becomes the activity that the thread performs
    - It does so in parallel with the original thread which created it
    - To begin executing the other thread, you invoke the start() method
    - Here is the implementation of that run() method
    - Simply prints out “running…” a few times and then exits
    - Note the call to Thread.sleep() - this will cause the current thread (that is, the thread executing this code) to pause for 1 second
    - So this code executes in the “main” JVM thread, and creates another thread and starts it
    - We want to wait until that newly created thread terminates, so we call join() on it
        - Why might we want to do this?
- Concurrency vs Parallelism
    - We have been talking about concurrency
    - But you are probably thinking about parallelism
    - They are not the same thing
    - It is possible to be concurrent and not parallel
        - E.g. a single threaded core
    - On modern computer systems, concurrency almost always implies parallelism: there are multiple cores available
    - The javascript runtime, in some ways, is an example of concurrency without parallelism
- Synchronization: doing things… at the same time. Safely.
    - Concurrency
        - If there is one concurrent activity on a shared resource, you need to coordinate activities to avoid issues
        - A traffic light coordinates access to a shared intersection
        - The same logic applies to concurrency in computing systems
        - Processes, since they do not share memory space, are less concerned with this issue
        - Threads, however, since they share memory, must be very careful
        - We will discuss two different tools for dealing with concurrency
        - Mutexes - a lock to isolate critical parts of the code to a single thread
        - Semaphores - a way to coordinate cooperation between threads
        - Mutexes
            - A mutex is a simple lock with two primary operations
            - lock() (sometimes wait())
                - Acquires the lock. If the lock is already held by someone else, the thread halts until it is released
            - unlock() (sometimes signal())
                - Releases the lock. If a thread is waiting on the lock, it is activated.
            - Consider the java code at right
                - If we create two threads to execute this method, what would happen?

            - Demo…
        - We need to synchronize these two threads
        - To do so, we can use the ReentrantLock class
            - Reentrant: if a thread has locked the lock, and tries to lock it again, the thread doesn't die
        - We use a lock to protect the critical section of the code
        - The place where shared data is both accessed and updated
        - Now the threads can properly interact with this shared data
        - DEMO
        - So you can see how to use locks to ensure that only one thread is updating shared data
    - Mutexes - deadlock
        - What if
            - Thread 1 acquires lock 1
            - Thread 2 acquires lock 2
            - Thread 1 attempts to acquire lock 2
            - Thread 2 attempts to acquire lock 1
        - Deadlock!!
        - To avoid deadlock you have a few options
            - Option 1: only one lock!
                - The Python GIL (Global Interpreter Lock)
                - Simple!
                - Destroys parallelism
            - Option 2: locks must be acquired in specific order
                - Better parallelism
                - Hard to enforce
            - Option 3: all locks must be acquired up front, in a single atomic action
                - Better parallelism
                - Hard to know in advance what locks an operation might need
        - For most projects, i recommend starting with option 1: a single lock
        - If it's good enough for python, it's good enough for us
    - Mutexes
        - Java provides other types of locks for specialized use cases
        - Readwritelock, for example, can be used to implement better concurrency for read-heavy/write-light use cases
        - Java also provides the synchronized keyword, which can effectively treat every object (i.e. non-primitive) as a lock
        - It appears that explicit locks are considered better form than using the synchronized mechanism
- Semaphores

        - Mutexes are about a single thread claiming and releasing sole ownership of a critical section
        - Semaphores, on the other hand, are about multiple threads coordinating with one another
        - We have already seen some code around thread coordination
        - The join() method here waits for the newly created thread to complete before this thread continues
        - Semaphores allow for more sophisticated coordination between threads
        - Here is a canonical example of a producer/consumer setup
        - The consumer acquires() the semaphore
            - If the semaphore's value is 0, it waits
        - The producer thread adds work to the queue and releases the semaphore
        - When this occurs, the consumer thread is woken up and can proceed to consume the value that has been produced
        - Note that we still synchronize the shared access of data using a lock
        - Semaphores are for coordinating the two threads, rather than for synchronization
- Review
    - Concurrency is when two logical operations can overlap
    - Concurrency and parallelism are not the same thing
    - The most popular concurrency tools are
        - Processes - heavy, simple
        - Threads - lightweight, complex
    - Concurrency with shared data requires coordination
    - Mutexes (locks) can be used to protect critical shared data
    - Semaphores can be used to coordinate activities between threads but are not usually used for protecting critical shared data

Concurrency 2: advanced concurrency

- Recall the concepts of concurrency and parallelism:
    - Concurrency is the term we use for when logical operations overlap in time
    - Parallelism is the term we use for when logical operations are being executed at the same time
- Threading is the most common concurrency model
- In order to safely and effectively work with shared resources in the presence of concurrency, it is necessary to use tools like mutexes/locks
- Java provides low level classes for dealing with concurrency
- Last time we looked at the ReentrantLock which implements locking for safe access to critical sections of code
- Today
    - Today we are going to look at higher-level concurrency tools that java makes available to you
    - Some concurrency tricks that may be useful at some point
    - Alternative concurrency models that avoid some of these complications
- Higher level concurrency

- atomicInteger - a class that wraps an integer and provides atomic that is “thread safe” access to it’s value
- In this code we create 5 threads that are all accessing the shared data atomicInteger
- You will recall from last time that to do this safely we needed to wrap access to the shared value in a Lock
- AtomicInteger abstracts that away and gives you atomic methods such as incrementAndGet() to safely increment and get the value
- DEMO
- atomicInteger is somewhat useful, but what about sharing data structures such as a list or Map?
- Consider some concurrent code that shares a hash map, putting and removing random strings from it
- Here there are five threads, all adding and removing random strings from a hashmap concurrently
- This code turns out to be extremely dangerous!
- If you were to allow this code to run long enough you would likely find one or more of the threads in a difficult-to-understand infinite loop!
- This shows the hazards of concurrent programming:
    - Difficult to understand semantics
    - Difficult to debug failures
    - Failures are non-deterministic: something may run just fine for a long time and then, suddenly, break
- Fortunately for us, Java offers a few solutions to this problem
- Solution 1, create a synchronized decorator of the map
- This will properly synchronize access and mutation within the hashmap so that there will not be internal corruption
- Unfortunately, since every get and put are synchronized, this means that only one thread at a time can update or read from the map
    - This hurts parallelism
- Java has a solution that allows for more parallelism but retains thread safety: ConcurrentHashMap
- ConcurrentHashMap will allow non-conflicting accesses to proceed in parallel, while still properly synchronizing updates to the maps internals
- ConcurrentHashMap is a widely used class in large, multi-threaded java applications
- As you would expect, there are other concurrent data structure classes that can be found in the java.util.concurrent package
- Consider the code at right
- We use a ConcurrentHashMap to keep track of the number of times a string has been accessed in the map
- Is this code thread safe?
- NO
- We are mutating the count value outside of any locking protection

- Thread 1: retrieves count for “A” which is 1 then pauses
- Thread 2: retrieves count for “A” which is still 1 then pauses
- Thread 1: updates the count to 2 and saves it
- Thread 2: updates the count to 2 and saves it
- Value is 2, even though 3 accesses have occurred!
- Again, this demonstrates how hard it is to get multithreaded code right
- To do this properly, you would need to put a lock around the entire read/update/save critical section
- Now things are safe, but you’ve destroyed the concurrency in this section of the code: only one thread at a time can update a count value
    - Maybe that’s OK, maybe not
- Again, this is why concurrency is so hard to get right
- So, we’ve looked at how to manage data that threads access, but what about managing threads themselves?
- We’ve already seen a few tools for helping coordinate threads:
    - join() - when called on a thread object, the current thread will wait until that thread exists to continue
    - Semaphores - can be used to coordinate threads (e.g. a producer/consumer situation)
- At right is some code that uses another tool: a latch
- A latch can be used to coordinate threads by offering two functions:
    - A countdown() method which decrements the latch count
    - An await() method which causes the current thread to wait until the latch is at 0
- Here we have five threads that all decrement the latch at some random time in the future
- The main thread waits until all these threads are complete before continuing
- Much cleaner than joining each thread
- Are threads free? Can we just have as many threads as we want?
- No
- Threads are expensive
- You often want to control the number of threads in your system
- To help you deal with this, Java offers Executors which provide a ThreadPoolExecutor
- A thread pool is a set of threads that can be reused to execute tasks or jobs
- A very common pattern in large applications
- The code at right creates a fixed size thread pool with 10 threads
- We submit tasks to the thread pool
- The tasks are farmed out to the threads to execute
- When a thread is done with a task it becomes available for a new one
- Contrast this with creating 100 threads to handle these 100 tasks
- That would certainly be better parallelism
- But it could also overwhelm your system with too many threads
- Typically better to cap parallelism

- Message Passing
    - If all of this seems complex and error prone to you, yes, yes it is
    - Concurrent programming using traditional approaches, even with good libraries like Java provides, is notoriously error prone
        - Even good programmers screw it up, sometimes badly
    - Some programming languages have address this problem by implementing message passing
    - At right is and example of Erlang, a famous message-passing based language
    - The counter process effectively acts like a very lightweight, single thread
    - Isolates all data locally to that module
    - The “function” is “invoked” via message passing
    - This approach has proven very effective for building highly concurrent, large scale systems
    - Adopted by some other programming languages as well (e.g. go)
    - As of today, still not widely adopted in the broader industry
    - Very popular in specific areas like telecom
    - Becoming more popular in other areas as well
- Review
    - Today we talked about some more advanced concurrency tools
        - atomicInteger - thread safe sharing of a single integer
        - concurrentHashMap - thread safe sharing of hashmap data structure
        - Latches - a simple way to coordinate multiple threads
        - threadPools - a way to keep the number of threads in your system under control by reusing existing threads
    - Even with these higher level constructs, concurrency can be very hard to get right!
    - Some languages use “message passing” with isolated environments between code units to implement simpler concurrency
        - Becoming more and more popular

Introduction to Networking in Java: Sockets, HTTP and HTML Parsing

- Networking
    - The core infrastructure that makes cloud computing possible is “the network”
    - Networking is an extremely complex topic
        - At right is the OSI Network model, a common framework for understanding how networking works
    - Since this is an overview, we won’t be going into the lower level details
    - However, before we move into the code, I want to mention two technologies
        - IP
        - TCP
    - IP: the Internet Protocol
        - A low level protocol for identifying hosts (computers) on the network and sending information to them, or receiving information from them
        - IP is where the term “ip address” comes from

    - Dig is command line tool available on linux, allowing you to turn a domain name into an IP address
    - This is done via DNS
        - The domain naming system
    - DNS is a distributed protocol for turning Domain Names into IP addresses
    - You may be familiar with URLs: Universal Resource Locators
    - URLs include a Domain Name
    - Here you can see that the domain name “yahoo.com” resolves to the IP address 98.137.11.164 etc…
    - Yahoo is a vintage website that was very popular a few decades ago
    - If you go to yahoo.com in a browser, the browser will actually connect to one of these IP addresses
- TCP: Transmission Control Protocol
    - TCP is protocol built on top of IP to reliably communicate information with a remote computer
    - IP has no notion of confirmation that a remote system has received a piece of information
    - There is actually another protocol built on top of IP that doesn’t make those guarantees: UDP
    - UDP is a fire-and-forget protocol used for streaming/gaming, etc…
    - Much faster, much lower latency
- TCP and IP technologies are so frequently used together that they are often referred to as TCP/IP
- TCP/IP is the most common protocol on the internet today, even in many places where it isn’t the ideal choice!
    - Worse is better!
- Writing Network Code
    - The primary abstraction that programmers use when writing networking code is known as a socket
    - Java has a pretty nice object oriented networking library found in the java.net package
    - Sockets come in two flavors:
        - Client sockets
        - Server sockets
    - Client sockets are sockets that connect to another computer
    - Server sockets are sockets that listen for a connection to the current computer
    - At right is some client socket code
    - We create an instance of a Socket class, giving it a server name and a port to connect to
        - Server name: resolved via DNS to an IP address for us
        - Port: A numeric value
    - Ports

        - Ports are a way for multiple programs to share the networking infrastructure of a computer without conflicting with one another
        - Many protocols have well known “default” ports:
            - HTTP: port 80
            - SSH: port 22
        - There is no law that a given program has to be running on a given port
        - Some operating systems restrict running on lower numbered ports to root or admin user-started processes only
        - Higher numbered ports are generally widely available and can be a little chaotic
    - Ok, so back to the code at right
    - We create a socket pointing to a given machine and port
    - We then create a PrintWriter that wraps the OutputStream of the socket
    - This will allow us to send data to the remote server
    - Question:
        - Why a PrintWriter?
        - Why not use the OutputStream directly?
    - OutputStream is low level, works in terms of bytes
    - PrintWriter allows us to work in higher-level terms
    - This is an example of the Decorator Pattern
    - Using the print writer, we send some lines of text to the remote system
    - Here we are sending code conforming to the HTTP protocol, asking for the root document
    - After we send data up to the server on the socket, ensuring it is all sent with a flush() call, we can listen for the response on the same socket
    - Again, we use the decorator pattern to get a BufferedReader
    - The BufferedReader will return lines of text or, if no lines are available, null
    - We can read the response and print it out to the console
    - Let’s take a look at the response! Demo!
    - Note at the end of the program, we close the writers and sockets
    - Necessary to free up OS-level resources associated with the socket
    - Turns out there can be resource leaks in java!
    - Ok, so that’s the client side of things
    - But we are talking to a server, listening on port 80 at montana.edu
    - What does the server side of things look like?
    - It’s a little bit more code, but sure looks familiar doesn’t it?
    - We have a new class, ServerSocket
        - The constructor takes a port to listen on
    - All that’s really new here is the accept() method
    - This is where we accept new connections when a client connects to the server

    - Once a client has connected, we are given back a… normal socket to communicate with them through
    - Since we are on the server, we reverse the reading and writing:
      - First, we read in the client’s request
      - Then we respond, echoing it back to them
    - Not too bad is it?
    - So what is all that gobbledygook stuff around “HTTP/1.0 blah blah blah”
    - We have been implementing very simple, very broken HTTP clients
    - Next time, we’ll take a look at HTTP, HTML, and Hypermedia!
- Sockets, HTTP, and HTML Parsing
  - Ok so what did we look at today?
  - We took a brief look at the networking layer model, including TCP/IP
  - Sockets, both client and server versions
    - Client sockets connect to a remote server
    - Server sockets listen for connections

HTTP, HTML, and Hypermedia: The Design of the Web

- The world wide web
  - In our last lecture, we looked at low level networking
  - In particular, we looked at how to use sockets to communicate between two computers over the network
  - We discussed some lower level protocols used on the internet:
    - IP - internet protocol
    - TCP - transmission control protocol
  - Today we are going to talk about a higher level system built on top of these: the world wide web
  - It's hard to understand, if you grew up with it, just how revolutionary the web was
  - Before the web, it was possible to just not immediately know sometime
    - You actually had to “wonder”: I wonder who the generals at Stalingrad were?
    - I wonder who invented corn syrup, etc…
  - The web is built on top of technologies built on top of the lower level technologies we discussed last time:
    - HTTP - the HyperText Transfer Protocol
    - HTML - the HyperText Markup Language
- HTTP
  - In our last lecture, we were actually using HTTP!
  - We wrote a very simple (and very broken) HTTP client and HTTP server
  - Let’s take a look at the protocol, which is built on top of TCP/IP
  - The HTTP specification is pretty simple by networking standards
    - Worse is better!
  - Keep in mind, this is just a basic text format, there is nothing magic here!
  - Note that an HTTP request has an “action”
    - GET - get a resource
    - POST - create/replace a resource

        - PUT - update/replace a resource
        - DELETE - remove a resource
    - The action tells the server what the request wants done
    - HTTP requests and responses have “headers” which are just pieces of text that are at the start of each, with the funny
        - <Name>: <value>
        - Syntax
    - Response headers, in particular, are important
    - They can tell a browser, for example, to go to an entirely different URL
    - Both requests and responses have optional bodies
    - The request at right has a form-encoded body of name/value pairs
    - A response body might be some HTML or possibly the binary data for an image
    - This is all pretty simple, but it’s still a pain to get right if you are dealing with raw sockets
    - Most programming languages have built in HTTP client libraries for working with HTTP servers without needing to get into the muck
    - At right we have rewritten out simple HTTP client to get the content at http://montana.edu to use the HTTPClient class, which is baked into Java
    - First, we create a URI
        - This is a URL as you are probably familiar from your browser’s address bar
    - We create a new HTTPclient using a factory method
    - We then build a request for the URI, using a BuilderPattern
    - Finally, we send the request, telling the client that we want the body to just be a string
        - Could also be a byte array or some other options
    - As you can see, this is a much nicer API for working with a remote HTTP system than dealing with raw sockets and strings
    - But, underneath, it’s doing the same stuff we did last lecture!
    - There are also higher level libraries for creating HTTP servers
    - One of my favorites is Javalin: https://javalin.io/
    - A very lightweight library that lets you create a webserver in java in minutes
    - At right is some simple Javalin code that will start a web server up on port 7070 that will respond to requests for the root path (/) with “Hello World”
        - Quick talk about closures
    - Easy peasy!
- HTML
    - HTTP, as the name suggests, was designed in order to transfer HyperText
    - The most popular HyperText in the world is HTML
    - HTML is a hypertext, which is a subset of a hypermedia
    - A hypermedia is any media that allows for non-linear interaction via hypermedia controls
    - At right is a simple HTML document:
        - It is an XML-like language

    - Tags are used to enclose content
- The reason this page is a hypermedia is because of that funny <a> tag
- This is called an anchor tag and it is a hypermedia control
- A proper web client will render this anchor tag as a link, with the text “htmx”
- If the user clicks on that link, they will be taken to the website: https://htmx.org
- A very good website
- So HTTP, HTML, and a piece of software called a browser come together to make this hypermedia system called the Web work
- It turns out that hypermedia systems are an interesting and distinct network architecture
- RPC
    - To understand how hypermedia is distinct as a network architecture we need to look at another alternative: Remote Procedure Calls (RPC)
    - The most popular RPC library is called gRPC, by google
    - Without getting into the gory details, network interactions are done in a functional manner, as if you are calling a function on the remote system
    - The gRPC library makes it look like you are just invoking a local function
        - Still a ton of junk, not a huge fan of google’s design sensibilities
- RPC and JavaScript
    - Today, the RPC style is very popular, even in web development
    - Front end JavaScript frameworks like React, Vue and Svelte allow you to build sophisticated user interfaces, and you are expected to sync the UI data with the server using what is effectively RPC
    - RPC in JavaScript typically uses a format called JSON
    - JSON is just a simple, JavaScript friendly syntax for presenting nested data
- Hypermedia vs RPC
    - Let’s look at two network responses that might be found on the web
    - The first is a hypermedia (HTML, yay!!)
    - The second is JSON (boo!)
    - These are both representations of a bank account in good standing
    - Note that the hypermedia representation encodes not just the balance, etc.. but also encodes all the actions available on the account
    - The JSON representation is more efficient: it simply encodes a status flag
    - However, the client is responsible for interpreting this information and displaying the appropriate actions
    - While the JSON encoding is more efficient, it requires a client to understand the concept of a bank account and a bank account status
    - The hypermedia representation does not require that of a client
        - The interface is uniform, that is just hypermedia
    - Now consider when an account goes into an overdrawn status
    - What do the responses look like?
    - Now consider when an account goes into an overdrawn status
    - What do the responses look like?

        - Note that the JSON response is largely the same: the status and value fields have changed
        - An RPC client must understand this status and render the appropriate UI
        - In the case of hypermedia response, the server simply renders a different set of links
        - The browser knows nothing about being "overdrawn" or, in fact, what a bank account even is
        - Because of this flexibility, hypermedia-based applications do not suffer from API churn issues: all "API" calls are surfaced through hypermedia and can change dramatically based on context, new features and so forth
        - Very flexible!
        - It is way, way beyond the scope of this class, but if it isn't already apparent, I am a big fan of the hypermedia network architecture
        - I have a library that I wrote to make hypermedia-based web applications more powerful: https://htmx.org
        - And I also wrote a book on how to build hypermedia systems:
            - https://hypermedia.systems
        - Will likely be teaching a class on this in Spring 2025!
- HTTP, HTML, and Hypermedia
    - HTTP is a simple protocol for transferring hypertext
    - Hypertext is a text form of hypermedia
    - The distinguishing characteristic of hypermedia is the presence of hypermedia controls
    - Hypermedia controls allow a user to interact in a non-linear manner with the media
    - The canonical hypermedia control is the anchor tag
    - Hypermedia is a unique network architecture when compared with, for example, the RPC style
    - Hypermedia is very flexible, at the cost of some representational efficiency

Cloud Computing: And now for something completely different

- We've been slowly moving up the abstraction hierarchy:
    - Started with transistors
    - assembly/machine code
    - C
    - Java/VM
    - Networking
- Now we hit the highest level of abstraction: The Cloud
- "The Cloud" is so abstract it is sometimes hard to even understand what it means
- Unfortunately the term has been taken by marketing and turned into a buzzword
- "Cloud computing is the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user."
- In plain english: the cloud is:
    - A bunch of computers working together

    - Tied together via network connections
    - Usually easily scalable (add another server, worker, etc either automatically or via a simple interface)
    - Typically managed, at the hardware level, by someone else
        - E.g. amazon
- Typically, although not always, these servers are
    - Located in a data center
    - Virtualized (multiple server instances per physical server)
    - Have a very fast local network backbone for inter-cloud communication
    - Have very fast connections to the internet
- History
    - 2006 - Amazon.com released the Elastic Compute Cloud (EC2)
    - 2008 - Google App Engine
    - 2010 - Microsoft Azure
    - 2010 - Rackspace OpenStack
    - 2011 - IBM SmartCloud
- Characteristics
    - EC2 is the largest cloud platform
    - What does it provide?
        - Scalable servers (easy to add or remove)
        - A web GUI and API for adding and removing them
        - Automated Scaling
        - Monitoring
        - Persistent Storage APIs
- Architecture
    - At right is a very basic cloud architecture
        - DNS - Domain Name System
        - Load Balancers
        - Databases
        - Cloud Storage
    - DNS
        - Originated in ARAPANET
        - Maps a name (e.g. montana.edu) to an IP address (e.g. 153.90.3.95)
        - Initially consisted of a single HOST.TXT file at Stanford Research
        - 1980s - turned into a network protocol for resolving
    - Load Balancer
        - Balances network requests between various application servers
        - Typically a relatively simple (and therefore reliable) server
        - May detect bad application servers and route away from them
            - Providing failover support in your cloud system
        - Load balancers may be session aware, to route a user to the same application server (if possible)
            - Why?
        - Load balancers may provide filtering or other services as well

        - gzip compression
        - Firewall support
    - Application servers
        - Where the application logic lives
            - E.g. a search node that is responsible for taking a search query and turning it into an HTML response
        - May be node, a J2EE server, a PHP server, .NET, etc…
    - Databases
        - Where the persistent data for an application is stored
        - RDBMs
        - Mongo
        - Etc…
    - This diagram shows a replication set
        - A replication set is part of clustering, which we will talk about later in the class
    - Cloud storage
        - Stores data and files related to the application
        - Why not use the DB?
            - DBs tend to perform poorly when delivering binary data, or even text data, compared with a file system
- CDNs
    - A CDN is a Content Delivery Network
    - A system for delivering static content efficiently
        - Static content - content that doesn't change
    - Typically sits in front of your load balancer
    - An origin server is the source of content (typically your cloud storage solution)
    - Assets (e.g. images) are resolved through the CDN and the request is routed to the closest physical CDN node
        - After initial request from the CDN node to the origin, no more requests
- Redis
    - Redis is a very popular data store frequently used in cloud architectures
    - Provides simple but powerful functionality:
        - Lists
        - Sets
        - Maps
        - Locks
        - Etc…
    - Redis is typically used as a cache layer
        - Alongside your database
        - Not between your database and application layer
    - The 440 project has a good, basic example of how Redis would be deployed in a cloud architecture
- Multi-datacenter Clouds

    - In order to survive major outages, a cloud may be split across multiple data centers
    - When this is done, you must decide on your data layout
        - 1 master DB?
        - Replication between peer databases?
- Cloud Computing
    - How does cloud computing match up with the computer system?
        - Load balancer - CPU control logic?
        - Application server - CPU
        - RDBMS - Memory/Disk
        - Redis - Cache
        - Storage - Disk
        - Network - Bus
- Developing on Cloud
    - Various terms have come to be used when discussing software deployed in a cloud
    - XaaS
        - SaaS - Software as a service
        - PaaS - Platform as a service
        - IaaS - Infrastructure as a service
    - SaaS - the highest level software deployed in a cloud system
    - Provides full software solution
        - Gmail
        - D2L
        - Etc..
    - PaaS - offers a full platform for developers to work with
    - .NET on Azure is a good example
    - AWS offers Elastic Beanstalk
        - Takes care of a lot of configuration for you
    - Heroku is another excellent example that we will use for our project
    - IaaS - offers a network abstraction over a particular piece of infrastructure
    - Most of AWS is IaaS
        - It gives you building blocks, not an entire platform
    - A good example would be an email delivery provider
    - Cloud computing has definitely gone to the industries head
    - FaaS - functions as a service
        - Individually deployed functions in a cloud
    - Sometimes called “serverless” because there isn’t a server? Wait, of course there is…
    - “Serverless” really means “you don’t run or pay for a server”
    - Amazon lambda is a good example
    - Also related to the idea of “microservices”
        - Small individual network services that do one thing
- Microservices

    - “Grug wonder wy big brain take hardest problem, factoring system correctly, and introduce network call too, seem very confusing to grug”
- Cloud == Amazon
    - One of the hard things about cloud computing is that there aren’t good standards
    - Kubernetes, a cloud-management system from google, is changing this to an extent
    - Unfortunately today, when people say “cloud” they usually mean AWS
- AWS
    - AWS stands for Amazon Web Services
    - A cloud hosting environment by amazon that provides various services
        - Virtual servers
        - Load balancers
        - CDNs (explained in a bit)
        - Storage
    - Extremely popular
    - I am not a fan of AWS
        - Confusing UI, lots of amazon-specific jargon
    - But it is the standard
    - As a career move, getting some AWS certifications would be a good idea
    - Still, I hate AWS
    - LeadDyno, my old startup, used AWS
    - Here is a screenshot of some of the resources it used
    - Two big ones:
        - EC2 (servers)
            - Server nodes
                - Web nodes
                - Worker nodes
            - Load balancers
        - S3 (storage)
            - Buckets
- Cloud Computing Advantages
    - Unfortunately the cloud computing industry has been loaded with buzzwords from the start
    - However, it is worth understanding some of these terms when discussing advantages of Cloud Computing
- Cloud Computing Buzzwords
    - Elastic Resource Capacity - Cloud systems have the ability to increase or reduce resources according to load
    - Utility pricing - pay as you go models (can be expensive during spikes!)
    - Virtualized resources - by running multiple virtual instances on one physical system, you can multiplex resource needs (CPU) over multiple workloads
        - Just hope you don’t get stuck on a server running a bitcoin miner

    - Self-service provision: you can spin up your own new instances via console, you don’t have to call someone to walk down and physically install a new server on a rack
        - Like when dinosaurs roamed the earth
    - Management automation - unfortunately this does not mean that management has been replaced with competent AI
    - Instead it means that you can automate your server management (e.g. autoscaling)
- Cloud Computing
    - Cloud computing is a network-centric deployment model for software
        - Often leverages virtualization to put multiple servers on a single physical device
    - Characterized by flexibility in
        - Provisioning new servers
        - Scaling resource needs up and down
    - Amazon/AWS is the biggest cloud platform
        - The cheapest place to run AWS servers is in Virginia
        - Why is that?

SQL in a Nutshell: CSCI 440 in 50 minutes

- In this lecture, I’m going to give you the entire CSCI 440 course in about 50 minutes
- Why?
    - Because the accreditors say so
    - It’s useful to see
- What is a Database?
    - A database is a system that organizes data into a collection of tables
    - A table is kind of like a spreadsheet, but the columns have a specific type
        - E.g. strings, integers, decimals, numbers, etc…
    - Rows in a database typically have a key associated with them
    - Here we have an albums tables with an albumId column
        - This is the key for this table
    - Rows can have Foreign Keys to other tables (or to themselves)
    - This table has a Foreign key to the Artists table
    - Note that the first row in the album table points to the artist with ArtistID 1
    - This artist is AC/DC
    - Foreign keys are analogous to pointers in C
    - A collection of tables with foreign keys are a schema
    - A schema can be represented graphically
    - We will be using the chinookdb schema for our examples, from the SQLite tutorial site
- SQL
    - SQL stands for Standard Query Language
    - It’s standard for creating, reading, updating, and deleting data in a database
        - CRUD
    - It is declarative, rather than imperative

- SQL - Select
    - A SQL select retrieves data from a table or collection of tables
    - Here we are selecting some fields from the tracks table
    - If you wish to retrieve all fields you can use the * operator
    - This will retrieve all fields of all rows from the tracks table
    - Often you wish to restrict the rows that are returned from a SELECT
    - To do this, you use the WHERE clause
    - This query will return all tracks whose albumid is 1
    - Constraints can be added with logical operators like AND or OR
    - This query will return all tracks whose albumid column is 1 and whose milliseconds (run time) is greater than 250 seconds
- SQL - Joins
    - Often you want to combine tables to produce some final information
    - This is achieved with the JOIN operator
    - Note that Album has a Foreign Key to Artist
        - This means that each album row has a pointer to artist, so multiple albums can belong to a single artist
    - If you want to return each album title as well as the artists names, you need to use a JOIN statement
    - The ON clause can be any sort of relational expression, but typically is equality on a foreign key
    - There are different types of JOINS
        - Inner join - only return exact matches
        - Outer join - returns rows that don't have a corresponding match
    - Most times you want an inner join
    - Joins can be combined with WHERE conditions
    - The WHERE condition can apply to any table in the query
        - In this case, artist or albums
- SQL - Order By
    - You can order the results of a query by using the ORDER BY clause
    - This example will return artists in ascending name
    - You can use the DESC keyword (or similar) to order the other direction
- SQL - IN, LIKE, BETWEEN
    - You can use IN expressions to test membership in a collection
    - You can use LIKE expressions to test string matching
        - % is the wildcard
    - You can use BETWEEN to express a range
- SQL - SubQueries
    - You can use the IN expression with nested sub-queries to create more complex
    - Here we find all customers whose support rep lives in Canada
- SQL - Insert
    - You can use the INSERT statement to create a new row in a table

        - A table specification may auto-populate some columns (e.g. the key for the table)
- SQL - Update
    - To update a value, use the UPDATE statement
    - Here we update the employee with the ID 3 to have the last name ‘Smith’
- SQL - Delete
    - To delete a value, use the DELETE statement
    - In the first example, we delete the row with the artistid of 1 in the artists_backup table
    - Note that this where clause is general and can match many rows, as in the second example, be careful!
- SQL - Group By
    - SQL supports aggregation of multiple rows into a single result via the GROUP BY clause
    - Here we group the tracks table by albumid
    - We use COUNT(trackid) to count how many trackids were grouped into a single result row
    - Aggregations usually include aggregation functions in the SELECT:
        - COUNT
        - AVG
        - MAX
        - MIN
        - SUM
    - Here we are summing up all the milliseconds of tracks on a given album
    - Very useful for reporting!
- SQL - Having
    - Sometimes you want to apply a condition to the results of the aggregation
    - To do so, you can use the HAVING clause
    - This will return all album ids with between 18 and 20 tracks on them
    - Note that this is distinct from the WHERE clause
    - The WHERE clause applies to the individual rows
    - The HAVING clause applies to the aggregated rows
    - Here we see both a WHERE clause and a HAVING clause
    - The WHERE clause filters to only tracks with an “A” in the name
    - The HAVING clause then filters any aggregated rows with a count outside the given range
- SQL - DDL
    - DDL (Data definition language) is the SQL used to define tables
    - CREATE creates a table
    - Here we create a contact table with a few fields
    - The contact_id field is an integer and is the primary key for this table
        - The primary key is the main column (or columns) to refer to a row by
        - First_name and last_name are Strings and may not be null

        - Email and phone are also strings and may not be null, and additionally must be unique in the table
    - SQL - DDL Foreign Keys
        - A foreign key constraint may be applied to tables to tell the database that a column refers to another table
        - Here we have a group_id column in the table suppliers, which refers to the group_id column in another table, supplier_groups
    - SQL - DDL Indexes
        - Databases support a data structure known as an index, which helps make queries run faster
        - Here we add an index on the email column in our contacts table
        - Email lookups will now be much faster
            - But contact inserts will be much slower
    - SQL - DDL Data Types
        - Every database supports a different group of column types
        - Some common ones:
            - VARCHAR, TEXT - Strings
            - Integer - integers (usually 64 bit)
            - Date, DateTime, Timestamp
            - Double, Float - Floating Point
            - Decimal - fixed precision
            - Blob - binary data
            - Bool, boolean - boolean
    - SQL - Transactions
        - Database support a concept of transactions
        - Allows multiple operations to either succeed or all fail as a single unit
        - Without transactions, these operations might fail halfway through and leave accounts in an inconsistent state!
    - SQL - ACID properties
        - Database transactions ensure that all changes are Atomic, Consistent, Isolated, and Durable (ACID)
            - Atomic - all succeed or all fail
            - Consistent - no ghost data
            - Isolated - as if no other transactions
            - Durable - written to persistent storage before success is reported
    - SQL - Java
        - Working with a database in Java is pretty easy
        - Use the JDBC API
            - Open a connection to the database
            - Create a statement
            - Execute a query
            - Iterate the results
        - Easy!
- SQL in a nutshell

    - Well that was a whirlwind tour of SQL
    - Play around with the SQLite tutorial if you are interested in learning more about SQL: https://www.sqlitetutorial.net
    - If you are interested in a deep dive into databases and a database-driven web application, consider taking CSCI 440 next year

Performance Tuning:

- Today we are going to discuss performance tuning: the process of taking code that is too slow and making it faster
- Before we get into the techniques for making code faster, let's talk about "too slow"
- What does "too slow" mean?
    - "This code looks like it won't run very fast. I'm going to optimize it"
- No, we aren't doing that
- Too slow, for us, means that an end user reported performance issues and we have verified them
- We must be very careful to avoid premature optimization
    - Why?
- Premature optimization almost always involves adding complexity to your application
- And what's the number 1 rule in software development?
    - Keep complexity under control!
- Ok, let's consider the following simple java application
- This is a simple fibonacci application that asks the user for an integer n and computes the nth fibonacci number
- Here we are using the classic recursive definition of fibonacci where fib(2) = 1 and fib(n) == fib(n-1) + fib(n-2) for n > 2
- A nice, elegant function and we release this as a command line tool for people to compute fibonacci numbers
- After we release the app and start getting performance complaints for numbers over ~45
- Let's see if we can verify these performance issues
- Ok, we have verified the performance issues, time to dive in
- First things first, we need to figure out how long the calculations are taking
- Let's add some code to do so to our entry point into the hot code
- Entry point: where a particular sub-system is entered from (often UI code or a controller in an MVC setup)
- Hot code: code that is being run repeatedly
- To time this function, we'll make use of System.currentTimeMillis()
- To capture the start time (in milliseconds) before we invoke the function
- Capture the end time after the function is done
- Log the difference between them
- Now let's retry with 44 and a few other numbers around 45
- There is obviously something bad going on here:
    - The algorithm is fast for low numbers but gets exponentially slower as we increase it
    - Remember your Big O notation class?
- Ok, next step, let's use a profiler

- A profiler is a piece of software that can be used to, well, profile some code, and tell you where perf issues are
- IntelliJ has a built in profiler
- Not the best, but free
- IntelliJ Demo
    - Summary
    - Bottom up/call tree
- Ok, so we have confirmed in the profiler that the fib() function is the issue
    - Profilers are much more useful when you have lots of complex methods rather than just one hotspot
- Next question, why is fib() a perf issue?
- To figure that out, we are going to throw some logging in
    - Note that logging will only make the performance worse, so we wouldn't leave this in production!
- Here is our simple logging statement
- We just log what's going on
    - Maybe if we want to be really we'd log the return value too
    - Start simple!
- When we've added logging to a method it is going to run slower, so we can use a smaller number initially to see what's going on
- Here we use 10
- What do you notice?
- We appear to be recalculating fib() for the same number over and over again
- Classic performance issue!
- This algorithm is O(2^n) or exponential
- That's why it blows out so hard perfwise at around 45
- Ok, so what's the fix?
- There are many options
- We could switch from recursion to iteration, for example
- We are going to use a classic performance technique: caching
- Caching
    - The idea in caching is that when you calculate (or retrieve) a value, you keep it around for reuse later
    - It's a core concept in both software and hardware
    - Caching is typically done with some sort of HashMap
        - Java has a nice hash map
    - After you calculate a value, you put it into a hashmap by a key
    - The next time you need it, you simply retrieve it from the cache
    - Here is our simple Java implementation
    - We declare a cache outside of the function, why?
    - When we call fib() we check the cache
    - If the value is already in there, just return it
    - If not, compute it
    - Before we return it, stick it into the cache for future use

    - With this cache in place, let’s take a look at the console
    - We only calculated the result once for each number!
    - Now let’s try it with 47
    - We’ve reduced the runtime from 7 seconds to 80 milliseconds…
    - So we’ve addressed our performance issue and now we can ship the fix (with the logging removed of course, but probably not) and be a hero to the company
    - Any problems with what we’ve done?
    - Probably not for something this simple, but keep in mind the quote about hard things in computer science …
    - Our cache is easy, the values never change
    - But if they did change, we would need to invalidate the cache
    - If you are seeing stale data in your app, think “did I cache this somewhere?”
- General Perf Considerations
    - Here are some general perf tips:
        - Hitting the network is orders of magnitude worse than almost any CPU-bound issue
        - If you hit the network (web, database, whatever) that’s where to start looking
    - String concatenation can also be an issue
        - Consider using StringBuilder in java or similar
    - If you are CPU bound, look first at the overall algorithms being used
        - Are there an O(n^2) or O(2^n) algorithms?
    - Resort to caching when necessary, but think through your cache invalidation mechanism
    - The fewer caches the better!
- Summary
    - Today we looked at performance testing
    - We discussed what an entry point and what hot code is
    - We looked at some tools for diagnosing perf issues
        - Timing
        - Profiling
        - Logging
    - And we looked at one way to address perf issues: caching
    - Remember: premature optimization is the root of all evil

Floating Point: Representing Decimals Efficiently

- Last lecture:
    - Remember earlier in the semester when we talked about binary representations?
        - Two’s complement for integer values
        - ASCII for character representation
        - Let’s loop back and consider decimal values
- Real numbers
    - Representing non-integer values
        - E.g. pi = 3.1415…

    - One obvious mechanism: dedicate a certain number of bits to the right hand side of the decimal
        - This is known as fixed point
- Fixed point
    - Advantages
        - Simple
        - Can use the same mathematical circuitry as integers
    - Disadvantages
        - Can only represent a small domain or number OR support a small number of decimal places
    - Fixed point is still used for specialized computations
        - E.g. finance
    - What decimal value is represented by 1010.1001?
    - Well, what decimal value is represented by 0003.1415?
        - 3 ones
        - 1 1/10th
        - 4 1/100th
        - 1 1/1000th
        - 5 1/10000th
    - You probably don't think of it this way, since you are so used to it, but that's the actual algorithm for understanding decimals
    - So, returning to 1010.1001 how should we read this?
        - 1 eight
        - 1 two (giving 10)
        - 1 one half (.5)
            - We are using binary so everything is in terms of 2s
        - 0 one fourths
        - 0 1 eighths
        - 1 one sixteenth
    - 10 and 9 sixteenths = 10.5625
- Floating Point
    - Realization
        - What if we allowed the point to float?
        - Dedicate a variable number of bits to the left hand and right hand side of the decimal point?
        - Dedicate a variable number of bits to the left hand and right hand side of the decimal point?
    - Advantages
        - Far greater range of numbers can be represented
        - No wasted leading or trailing 0 bits
    - Disadvantages
        - More complex
        - Different circuitry for mathematical operations
- Floating point history

    - First known use of floating point was in a electro-mechanical computing machine designed by Leonardo Torres y Quevado
        - Spanish engineer
        - Designed first computer game (chess) and was a pioneer in remote control
            - How come I’ve never heard of him? Good question.
    - Konrad Zuse
        - German engineer
        - Designed the world’s first programmable computer, the Z1
            - Included 24-bit binary floating point
        - Also developed the first high-level programming language, Plankalkul
            - How come I’ve never heard of him? Another good question.
    - 1950s to 1980s
        - Many competing floating point standards
            - IBM 7*
            - UNIVAC
            - Etc..
        - IEEE 754 - 1985
            - Established a floating point standard for the industry
                - Intel
                - Motorola
    - Intel 8087
        - The first x87 floating point coprocessor for the 8086 line of microprocessors
        - 20% to 500% faster for many operations
    - Eventually this functionality was folded into the main CPU with the advent of the 486 chip
    - William Kahan
        - Primary architect behind the IEEE 754 standard
        - The “father of floating point”
        - Wrote the program paranoia to test floating point implementations
            - Found the floating point bug in pentium division
        - Won the turing award for his contributions
    - Today X86-64
        - Includes registers for floating point values
        - 128 bits (!!!)
        - XMM0-XMM7 (part of x86-32 SSE)
        - XMM 8-15 (available in 64 bit mode only)
- Floating point details
    - IEEE 754 floating point representation
        - Various levels of precision
            - Half - 16 bits
            - Single - 32 bits
            - Double - 64 bits

      - Extended - 80 bits
      - Quad - 128 bits
  - IEEE 754 half
    - A total of 16 bits
      - 1 sign bit
      - 5 exponent bits
      - 10 significand (fraction) bits
    - Sign bit is obvious
      - 0 is positive
      - 1 is negative
  - Exponent
    - Exponent is a 5 bit value, giving 2^5 = 32 different possibilities
      - 00000 and 11111 have special values we will talk about in a moment
      - This leaves the values 30 through 1 for “normal” exponent values
    - Note that exponent is unsigned
    - We wish to express negative values for an exponent
    - To accomplish this, the exponent is biased at 15
      - The raw value has 15 subtracted from it to get the actual exponent value
      - This allows exponent values of 15 to -14 for the exponent
    - Consider the following exponent value:
      - 11100 - decimal 28
    - This would correspond to an exponent of 28 - 15 = 13, or 2^13 for this floating point number
    - IEEE 754 half
      - What about exponent value 00000?
        - Typically means 0
        - Can also mean subnormal numbers
          - Very small numbers below the “normal” floating point range
      - What about the exponent value?
        - Can mean infinity or NaN, depending on significand
- Significand (fraction)
  - Fraction is 10 bits
  - Gives us 2^10 (1024) possible values
    - Values are expressed in terms of x/1024th
      - E.g. 0000000001 -> 1/10124th
    - Significand value is added to 1 to get a number somewhere between 1 and 2
      - Except subnormal case, when 0 is added to it
- Float value calculation
  - Consider this 16 bit floating point number
  - Sign bit: 0 (positive)

    - Exponent 1 -> 2^(1-15) = 2^-14
    - Fraction = 0000000000 -> (1 + 0/1024)
- Binary fractions
    - We are used to decimal notation
    - What does 1.2345 mean in terms of fractions?
        - 1 + 2345/10000
    - What does 1.01011 mean in binary?
        - 1 + (001011/100000) binary
        - 1 + (11/32) decimal
        - 1 + (.34375) decimal
        - 1.34375
    - Another example:
    - 0.1011
        - 1011/10000 binary
        - 11/16 = 0.6875
    - Or look at it in terms of the two places…
- Float value calculation
    - Consider another 16 bit floating point number
    - Sign bit: 0 (positive)
    - Exponent 13 -> 2^(13-15) = 2^-2
    - Fraction = 0101010101 - > (1 + 341/1024)
    - This is the closest 16 bit floating point can represent 1/3rd
- Float precision
    - This is a serious limitation of floating point: it can only be an approximation of many values
- Float rounding rules
    - Floating point has different rounding rules
        - Round to nearest, ties to even
        - Round to nearest, ties away from zero
        - Round toward 0
        - Round toward positive infinite
        - Round toward negative infinity
    - To be compliant with the spec, CPUs must offer instructions that allow you to set the rounding mode for your system!
- Float precision implications
    - You must be very careful when using floating point!
    - Floating point is a bad idea when dealing with fixed precision numbers
        - E.g. money!
    - That’s OK though, our industry is smart enough to understand this…
    - Uhhhh
    - JavaScript uses 32 bit floats for numbers
    - So obviously true mathematical statements are …. False
    - People are increasingly writing software in JavaScript
    - The patriot missile incident

        - February 1991
        - Precision issue in an MIM-104 patriot missile battery prevented it from intercepting an incoming SCUD missile
        - 28 soldiers killed
- Float Special values
    - Float has some special values
        - A signed 0 value
            - Equal to one another
            - Division by one or the other leads to a signed infinity
        - Infinities
            - + and - infinities
            - Satisfy standard infinity math (e.g. 3 + + infinite = + infinite)
        - NaN (not a number)
            - Represents invalid values such as 1/10 or sqrt(-1)
- Larger floats
    - We have been looking at half floats, but this same logic generalizes to any length of bits
        - Float - typically 32 bits
        - Double - 64 bits
    - With more bits, more precision
        - Still not perfect precision however
- Floating point today
    - On x86-64
        - Floating point values are passed in separate registers
        - 128 bits of precision are available
        - XMM0-XMM7 (part of x86-32 SSE)
        - XMM 8-15 (available in 64-bit mode only)
        - Separate assembly instructions for working with them
- Floating Point
    - We took a look at how to represent non-integer values in binary
    - Initially fixed point was used
    - Floating Point representation is more efficient
        - But also more complex!
    - We took a look at 16 bit floats
        - Larger floats are just more of the same
    - And we looked at problems with Floating Point precision
        - JavaScript is a very terrible programming language
    - REMEMBER: IT’S JUST BITS!

CSCI 366 - Systems Programming Final Review

- Course goals
    - Introduce students to fundamental concepts in computer systems, including software environments and development tools, computer architecture and organization, concurrency, information management, network communications, and operating systems based on cloud computing

- Review
    - Today we will go over the most important concepts that I want you to take away from the class
    - Logical operators and what's special about NAND in particular*
    - How to implement NAND in silicon
    - How an adder can be built using logical gates
    - Integer and floating point representation * (floating point only)
    - The layout and function of the Scott CPU *
    - The Scott CPU memory implementation *
    - LMC Assembly *
    - The function call abstraction
    - Compilation
    - Bit masking *
    - Networking *
    - Each component of the basic cloud architecture diagram
- NAND Gate
    - NAND gates are special
        - NAND gates are functionally complete
        - All other logical gates can be constructed with them
- NOT Gate
    - To create a NOT, simply put both inputs into a NAND
- AND Gate
    - AND gate construction
    - Easy: NOT (A NAND B)
- OR Gate
    - A little more complicated
    - (NOT A) NAND (NOT B)
- A basic NAND gate
    - At right is the basic CMOS circuit to implement an NAND gate
    - If both inputs A and B are high, OUT is low
    - Otherwise, OUT is high
    - All additional logical circuits can be built on top of this
- Adder Circuitry
    - At right is a full adder
    - Called a full adder because it includes a carry in and carry out bit
    - Built out of NANDs, ORs, and ANDs
        - These latter two can be built out of more NANDs
    - In order to add multiple bits together, we can simply chain these adders together
    - This is known as a "ripple carry" adder
    - Real world adders are more sophisticated than this
- Two's Complement
    - In order to efficiently represent positive and negative integers, almost all computers use two's complement encoding

        - This makes the normal adder circuitry work for both signed and unsigned integers
- Binary Finger Counting
    - Remember you can count to 31 on one hand!
- Floating Point Details
    - Floating point is a different mechanism for representing numbers
    - Allows computers to represent decimal values rather than only integers
    - At right is a 16 bit float
    - Known as IEEE 754 Half
        - A total of 16 bits
            - 1 sign bit
            - 5 exponent bits
            - 10 significand (fraction) bits
    - Sign bit is obvious
        - 0 is positive
        - 1 is negative
    - Exponent
        - Sign bit is obvious
        - Exponent is a 5 bit value, giving 2^5 = 32 different possibilities
            - 00000 and 11111 have special values
            - This leaves the values 30 through 1 for “normal” exponent values
        - Note that the exponent is unsigned
        - We wish to express negative values for an exponent
        - To accomplish this, the exponent is biased at 15
            - The raw value has 15 subtracted from it to get the actual exponent value
            - This allows exponent values of 15 to -14 for the exponent
        - Consider the following exponent value:
            - 11100 - decimal 28
        - This would correspond to an exponent of 28 - 15 = 13, or 2^13 for this floating point number
- The Scott CPU
    - Please review the Scott CPU
    - In particular, please review the use of each register:
        - The instruction register
        - The instruction address register
        - The accumulator
        - The temp register
        - The general purpose registers R0 through R1
        - The memory access register
    - Each register performs a specific task
        - E.g. the instruction address register holds the address of the next instruction to execute
    - The scott CPU cycle consists of six-steps:

        - 1 - load the IAR to the MAR + bump IAR by 1 in the accumulator
        - 2 - load the memory location into the IR
        - 3 - move the accumulator into the IAR (advancing it by 1)
    - The next three steps depend on the particular instruction
    - An add instruction:
        - Moves a value from a general purpose register into the TEMP register
        - Puts another general purpose register onto the bus + enables the adder and sets the result into the accumulator
        - Moves the accumulator back to a general purpose register
- Memory
    - The Scott CPU memory subsystem makes uses of two decoders
    - A decoder takes n bits and enables 1 of n^2 wires
    - Here we see two 4x16 decoders, wired into the MARs top 4 and bottom 4 bits
    - Given an address in MAR, exactly one top wire and one bottom wire will be activated
    - Where those two wires meet is an 8 bit memory slot
    - This is the selected memory slot for either a read or a write
    - At the intersection of the wires is an AND gate
        - If both the vertical AND the horizontal wire are enabled, then this memory location is selected
        - If, additionally the set wire is enabled, the memory location is written to from the bus
        - If, alternatively, the enable wire is on, the memory location is put on the bus
- Little Man Computer
    - Assembly is the lowest level of programming language above machine code
    - Please review the basics of Little Man Computer Assembly:
        - How INP, OUT, and ADD work
        - How branching works
    - In LMC there is a 1 to 1 relationship between assembly and machine code
    - Not so in MIPS or LMSM assembly
        - Why?
- Functions
    - Function calls are a (perhaps the) fundamental concept to understand how high level computer languages work at the hardware level
    - Remember the JAL instruction:
        - Jumps to another area of code
        - Saves the next address in the return address register
    - Also important is the notion of a stack
    - Local function variables are stored on the stack so that things like recursion can work properly
- Compilers
    - A compiler is a piece of software that takes source and transforms it

    - Typically it takes high level source (like C) and transforms it into low level code (like assembly)
    - It takes code like we see at right
    - Doubling an integer
        - C provides much higher level abstractions, in particular the function call, that have to be done manually in assembly
        - Take CSCI 468 with me to learn about compilers!
- Bit Masking
    - Bit masking is a technique used sometimes in C programming
    - Bit masking can be used to test the value of a single bit in a larger piece of data
        - Useful for packing multiple booleans into a single long, for example
    - To get the value of a single bit, you will need to create a bit mask
    - A bit mask is the number 1 that has been shifted to the correct bit position
    - The left bit shift operator in C is <<
    - This moves all the bits n slots to the left in the number
    - Consider binary 1010 (decimal 10)
        - 1010 << 1 = 0100 (decimal 4)
        - 1010 << 2 = 1000 (decimal 8)
        - 1010 << 3 = 0000 (decimal 0)
    - Getting a bit mask for the nth bit
    - Testing the nth bit of a value
    - Setting the nth bit of a value
    - Clearing the nth bit of a value
- Networking
    - Recall that primary abstraction that programmers use when writing networking code is known as a socket
    - Java has a pretty nice object oriented networking library found in the java.net package
    - We discussed two different types of sockets
    - Please review what they are called and what each one does
- Cloud Architecture
    - At right is a very basic cloud architecture
        - DNS - Domain Name System
        - Load Balancers
        - Application Servers
        - Databases
        - Cloud Storage
    - Which of these components must stay up?
    - Which of them are able to reboot without serious consequences?
    - Why?
- Redis
    - Redis is a NoSQL data store
    - Remote Dictionary Service
    - Core concept is that of key-value pairs

        - Not unlike a giant hash table
    - Recall that redis gives us shared data structures over the network
    - In what situations would Redis be useful?
    - In what situations would Redis not be useful?
- Thank you
    - I hope you’ve enjoyed the class and feel like you have a better understanding of how computers work, from transistors up to cloud computing!