Computer Systems

Avery Karlin

Fall 2015

Contents

Primary Textbook:
Teacher:

1 Chapter 1 - Introduction

1. Computer systems are viewed from a bottom up approach of how transistors run basic programs and a top down approach of how complicated programs are converted into simple logic commands
   (a) This is used to understand commands a computer does well and does badly to program more efficiently, and to understand how changes in technology change the speed of computing
   (b) C is used due to being high level enough to write large programs, but low level enough to allow direct modification of bits
2. The concept of abstraction is central to computer science, focusing on a higher level, rather than the component ideas to save time and mental effort, assuming the details work
   (a) On the other hand, it must go in turn with deconstruction, breaking down an abstract idea into more concert sub-ideas, the opposite of abstraction, in case there is a problem
3. The concept of viewing hardware and software as joint components of a single system, which must both be taken into account to design either, is another central idea, to design the most effective components
4. Processors/CPUs are the primary unit of a computer, used to direct the processing of information and perform the calculations required to process it, though there are other components to make use easier
   (a) Originally, they were made of large boards covered in integrated circuit packages, but now are just a single silicon microprocessor chip with millions of transistors
   (b) Memory is another major component, made up of a series of slots, each with an address and able to hold a single value, connected to CPU
   (c) Programs are sets of instructions executed one at a time, stored in memory, while the program counter contains the current/next instruction in the program, often just incrementing after each
5. The only limitations of computers are time and amount of memory, but otherwise all computers can do the same tasks, though not at the same pace
   (a) This is due to computers being universal computing devices, as digital machines which could be increased in precision unlike analog/physical machines, but which were not made for individual tasks
   (b) Turing in 1937 proposed the Turing machine, which would be able to carry out all computations of some type, and later began to define what computation is, abstracting tasks by a black box model, showing the task, input, and output, with no specification about how it is performed
      i. Turing's thesis states that a Turing machine can do all computations, such that improvements to it do not change the amount of computations, making a universal Turing machine able to simulate all different Turing machines
   (c) Computers/universal Turing machines are able to do any computations, due to being programmable
6. Computer problems must be converted into voltages to influence the flow of electrons which the computer is made of, made of a series of methods to allow carrying out of complex tasks
   (a) The levels of transformation are the levels of choice to convert the problem into an electron flow for the computer, starting with the statement in a natural/human language, which has too much ambiguity to give directly to the computer

   (b) The first transformation is to an algorithm, or a finite step by step procedure, with definiteness, or precisely stated, and effective compatibility, or able to be carried out by a computer,
      i. There are many possible algorithms, depending on the number of steps allowed concurrently by the computer, with many different speeds and lengths
   (c) After, it is converted to a mechanical/programming language, specifically created to avoid ambiguity, often designed for specific purposes
      i. High level languages are those far from the computer itself, often machine independent, such that they don't rely on the computer specifics, such as C
      ii. Low level languages are specific to the computer, such that there is one for each computer generally, called assembly
   (d) The third level is conversion into the computers instruction set architecture (ISA), or the specification of interface between programs and the hardware, generally x86 on Intel processors
      i. The ISA specifies the instructions and operations the computer can do, what inputs (operands) it requires, and the operand formats (data types) it is able to accept
      ii. It also contains mechanisms for the computer to find operands, called addressing modes, the number of unique memory locations, and the number of bits in each location
      iii. High level languages are converted to ISA format by a compiler, while low level languages are converted by an assembler
   (e) After, the ISA is transformed into an implementation, based on the microprocessor-specific microarchitecture
      i. Unlike the ISA, which specifies what the computer can do, the microarchitecture specifies how the computer actually performs those tasks
   (f) The microarchitecture is made out of logic circuits, determining the trade-off of cost and efficiency during manufacturing, each of which is further made out of a specific type of circuit materials, with its own specifications

2 Chapter 2 - Bits, Data Types, and Operations

1. The movement of electrons is controlled by devices reacting to the presence of voltage, rather than the amount for simplicity
   (a) These voltages are signified by binary digits/bits, 1 for voltage near the maximum, and 0 for voltage near zero (rounding due to device variation)
   (b) Bits are then combined to get a large number of distinct values
2. Data types are representations of values in which operations are encoded within the data types, mainly using 2's complement integers, ASCII codes, and floating points
   (a) Unsigned integers are used to identify locations and counts, represented by positional binary representation, ranging from 0 to $2^k - 1$ for k binary digits, added normally
   (b) Signed integers are used for arithmetic, with a leading 0 to signify positive, from 0 to $2^{k-1} - 1$ for k binary digits, using different systems for negative
      i. The signed magnitude system uses a leading 1 for a negative value, with a leading 1 on 0 signifying -0
      ii. The 1's complement system uses a leading 1 for negative values, signifying it is

        equivalent to switching every other binary digit in the value (complement) to get the magnitude
    iii. The 2's complement system is the standard data type, equivalent to the 1's complement - 1 for each negative value, found to be easiest to design hardware to do arithmetic on

3. Computers generally use an arithmetic and logic unit (ALU) for addition, converting two inputs to one output of the sum, performing it in the same column method with carrying as standard addition, without actually determining the values, leading to the 2's complement system
    (a) Thus, it is necessary for the sum of a number and its inverse to equal 0, ignoring all extra digits, such that only the correct number of digits is kept, and it is necessary for the sequence to be equal to one added, such that it works for results in the range, providing the advantage of the complements
    (b) The advantage of 2's complement over 1's complement is making full use of the maximum amount of digits, rather than having two versions of 0
    (c) The precision of the value is the number of numerical bits, while the range is the maximum magnitude of the bit sequence
4. Since addition is the main arithmetic bit operation, subtraction of bits is simply done by adding the additive inverse of the second value
    (a) Left shifts are equivalent to shifting all values to the left, adding a zero as the rightmost, dropping the overflow, equal to multiplying by 2 assuming no overflow
    (b) Arithmetic right shifts are shifting to right, adding the sign bit as the leftmost, while logical right shifts are shifting to the right, adding a 0 as the leftmost, equal to dividing by 2 for arithmetic shifts, and dividing unsigned by 2 for logical
        i. For odd values shifted right, the result is rounded based on the value in the 2nd rightmost spot
    (c) Sign extension/SEXT is used to shrink the amounts of bits used for a smaller number, removing all but one leading zero for positive, all but one leading one for negative
        i. Vice versa, the leading digits can be added, due to requiring the same number of bits used for adding or subtracting
    (d) Overflow is noted by the loss of the leading digit, such that the sum of two positive is negative, or vice versa, such it can be dealt with
        i. For unsigned addition, an outgoing carry denotes an overflow to be dealt with
5. Logical operations are based on viewing binary as true/1 and false/0, and are binary functions, requiring two logical inputs/bits
    (a) They are also able to be used on sequences of bits of the same length, testing corresponding bits from each
        i. Bit masks are sequences which are used to separate/modify specific bits from a sequence, using logical operations
    (b) Bit-wise AND returns 1 iff both inputs are 1, bit-wise OR (inclusive OR) returns 0 iff both inputs are 0, and bit-wise XOR (exclusive OR) returns 1 iff only one of the inputs is 1
    (c) The NOT/complement function takes a single input (unary function), inverting/switching each of the bits in the sequence
6. Bit vectors are binary sequences used to track if units are available/1 or busy/0, numbering units from right to left, starting with 0

7. Floating points are used to describe numbers of a high range, but low precision, in terms of scientific notation, such that for a 32 bit float, 1 signifies the sign (1 negative, 0 positive), then 8 for the range/exponent, then 23 for the precision/fraction
   (a) This is also called single, while doubles are similarly denoted, though the exponent is 11 bits, while the fraction is 52 bits
      i. Thus, doubles have an exponent from 0 to 2047 instead of 0 to 255, with a bias of 1023
   (b) Thus, the number is equal to (1 + fraction) * $2^{exponent-2^7+1}$, where the exponent is not allowed to be 0 or the maximum exponent
      i. If the exponent is 0, it is valued as if it is 1 under the IEEE 754 Standard for Floating Point Arithmetic
      ii. As a result, the 127 added to the exponent is called the bias, allowing for negative exponents as well
   (c) The fraction is found by converting the numerator and whole number to binary, then shifting to find the real exponent value, such that the fraction is the sequence except the leading 1
      i. This is done similarly to how it would be found for scientific notation, except that both parts are converted to binary before
      ii. The fraction can be thought to be continuing on the binary exponents, continuing as 2^{-1}, 2^{-2} and so forth
      iii. Due to converting non-binary denominator fractions into floating points, it can lead to various rounding errors, such that arithmetic with it would be close, but not exactly correct
         A. Thus, bounds of error are used to check the result, rather than equality functions
   (d) The sign bit is unrelated to the two's complement format
8. ASCII (American Standard Code for Information Interchange) converts character codes from input and output into unique 8 bit codes universally
   (a) As a result, keys generally have multiple associated codes, due to different characters able to be produced by each
9. Hexadecimal notation easily is produced from binary, taking 4 bit blocks (nibbles) into binary digits for human ease, used identically to binary of that form, able to be added similarly
   (a) An additional bit at the end is removed similarly to twos complement, and overflow is noted and dealt with by the same means
10. Hexadecimal notation is noted by an x before the number, while decimal notation is denoted by # before the number, generally in assembly languages

3 Chapter 3 - Digital Logic Structures

3.1 Building Blocks

1. Microprocessors are generally made of metal-oxide semiconductor (MOS) field effect transistors (MOSFET), divided into p and n types, each with three terminals, the gate, source, and drain
   (a) These are made up of a metal on top of the gate, with an oxide under, the main body made of Si semiconductor

        i. The N-region under the source and drain for a P-type, at the source and drain for an N-type, is made up of Si doped with P, while the P-region is made up of Si doped with Ga
        ii. The P-region doping attracts surplus electrons easily within, while the N-regions are more difficult to do so, forced out
        iii. Thus, current in an N-type gate pulls electrons from the body to the top, flowing out, while in a p-type, current causes it to be absorbed
    (b) For some amount of voltage, generally 2.9 V, supplied to the gate of an n-type, the switch turns on, such that electricity can move from the source to the drain, forming a closed circuit
        i. This is generally only denoted by shorthand in drawings by the gate, which is written out, while the other terminals are just drawn as wire, denoted overall by a parallel line next to the wire with the gate coming from it
        ii. p-type transistors have a small circle between the gate and the line leading to the gate
    (c) p-type transistors work the opposite manner, acting as a wire only when there is virtually no voltage, while acting as an open circuit if 2.9 V is placed in the gate
    (d) As a result, circuits with both types are called complementary metal-oxide semiconductor (CMOS) circuits
        i. These are complementary due to the N-type transistors on the bottom connected to the grounding in PDN format, and P-type transistors on the top, connected to power in PUN format
    (e) Closed wires/switches are also called shorted wires within a transistor, such that it must be connected to a grounding to prevent shorting
2. Logic gate structures are transistor circuits for the purposes of logical functions, AND, OR, and NOT/Inverter gates as the fundamental gates, depending only on the current input
    (a) These can be drawn either in pull-up network (PUN), drawing from p-type transistors connected to the original source, switching from in-series to parallel or vice versa when at the n-type
        i. In this case, AND is parallel, OR is in series, automatically adding a NOT without an inverter
        ii. It can also be down pull-down network (PDN), drawn from the grounding to the n-type, such that AND is in series, OR is parallel, automatically adding a NOT without an inverter, considered easier
    (b) NOT gates are made up of two MOS transistors of opposite types, current flowing into the gates, the input leading to the gates of both, with a power/voltage going into the source of the p-type
        i. The drain of the p-type is connected to the output and the source of the n-type, with the drain of the n-type grounded
        ii. Thus, if there is voltage to the gates, voltage flows from the source to the output, while otherwise, voltage flows from the inputs to the ground, with none to the output, the latter necessary to create a grounded circuit if broken
        iii. Inverters are drawn as a triangle with a small circle before the output wire, or can be drawn with just the small bubble/circle to show inversion as a shorthand
    (c) The NOR (Not OR) gate has two p-type in series, each with a separate input, and with each input connected to one of two parallel n-type resistors, which are connected in series

to the p-types, with the output between

   i. Each n-types are grounded, with voltage automatically flowing to the first p-type, such that it acts similarly to two NOT gates combined, such that the gate is grounded unless there is an output as well
   ii. OR gates are then created by adding an inverter to the output of the NOR gate
   iii. NOR gates are drawn as a crest-like shape, with an input to each horn, the output from the bottom point, with a small circle before the output wire, OR gates without the circle, XOR without a circle, but with an additional crest line mirroring the main one on the bottom

(d) NAND (Not AND) gates have two parallel p-type transistors, each connected to an input and a voltage source, joining to the output, each connected to one of two in series n-types connected to the ground
   i. As a result, the circuit is grounded only if the output is zero, such that there is current through both inputs, while otherwise current is through the output
   ii. AND gates as a result are just NAND gates with an added inverter after
   iii. NAND gates are drawn as a closed semi-circle with a small circle before the output line, with two input lines, while AND gates are drawn without the circle

(e) DeMorgan's Law states that NOT(A OR B) = (NOT A) AND (NOT B), showing the relationship between the NOT and OR gates
   i. It follows as a result that NOT(A AND B) = (NOT A) OR (NOT B) is also a valid form

(f) Boolean algebra is also often written in the form of $\cdot$ as AND, $+$ as OR, and $^-$ as NOT, with the order of operations as NOT, AND, then OR
   i. As a result, $X \cdot 0 = 0, X + 1 = 1, X \cdot 1 = X$, and $X + 0 = X$ as the main identities, with both associative properties of AND and OR, and distributive property of AND over OR and of OR over AND

(g) Karnaugh Maps convert a logic array to a statement, making one side of the square the possible combinations of half the variables, the other side the other half
   i. It is drawn such that any adjacent cells differ by one variable, including across borders, though not diagonally, such that the order is 00-01-11-10, instead of 00-01-10-11 which is sequential binary
   ii. Adjacent entries of some power of 2 as a result can be designated as an AND, signifying the change from each, adding each separate group together with an OR, even if containing values held by other groups

(h) The binary gates can be extended as a result to a larger number of inputs by extending the number of transistors that make up the gate, thought it is noted that more levels creates a delay of that magnitude

3.2 Combinational Circuits

1. Combinational logic circuits, or decision elements, are structures of logic gates, depending only on immediate input data provided, rather than any stored data, contrasting data storage structures

   (a) Decoders are a circuit which takes two inputs, and returns four bits based on which combination of the inputs is 1, called the opcode, used often for turning on a specific hardware unit

        i. It thus decodes two ordered inputs, returning 1000 if 00 is the input, 0100 if 01, 0010 if 10, and 0001 if 11
        ii. This is done by placing four AND gates in parallel, each with a varying number and sequence of inverters directly before them

(b) Mux, or multiplexers, select one of two inputs to use as output based on the select signal
    i. This is done by feeding the select signal and its inversion into two AND gates, each with one input going to it, combining the results in an OR gate to return the chosen value
    ii. This is extended to more inputs by increasing the number of possible select signals to the AND gate
    iii. This is drawn by a trapezoid with the select signal going to a leg, the inputs to the longer side, the output from the shorter, or as a rectangle
        A. Multiple select signals can be denoted by a slash on the wire with the number of signals written above the slash

(c) Full adders take three one bit inputs, a_i, b_i, and the $carry_i$, such that the two results are s_i and $carry_{i+1}$, used to sum two bits in a column, adding the three inputs into 8 series of AND gates, each with a different combination of inverters before
    i. The results of the OR gates are then fed into two OR gates, depending on how they combine for the truth table, to produce the sum of the column
    ii. Thus, for a 2s complement or unsigned numbers, since it can be added in columns just as a decimal number, each full adder can be used for a different column, taking the carry of the previous, to produce the sum of binary digits
    iii. These are drawn as either a square with an indication mark in the middle for a single unit with the carry going in from the right, out from the left, inputs from the top, output from the bottom
        A. For a complete full adder, it is drawn as a trapezoid with the inputs to the larger end, the inward carry to the top leg, and the output from the small end
    iv. Subtraction is thus done by a full added by negating one of the input bits, then adding a carry of one to the sum based on the rules of 2's complement
    v. Half adders are defined as an adder without an initial carry input, though still with a carry output

(d) The Programmable Logic Array (PLA) has an array of AND gates with a combination of inputs, such that for n inputs, there must be 2^n AND gates, each with varying negations
    i. These are then connected in some formation to an array of OR gates for each output, based on if the inputs produce each output
    ii. This as a result is able to be used to form any truth table in circuit form, consisting of only AND, OR, and NOT gates, such that those sets of gates are called logically complete
    iii. While PLAs are able to implement any circuit, they are often not the most efficient method of creating the circuit
    iv. Multiple bit inputs and outputs are denoted by a slash with the number of bits written next to the slash

(e) Arithmetic Logic Unit are thus made by a full adder, often combined with a decoder to separate between signals for AND, OR, ADD, or SUB, with a Mux to use the same adder for ADD and SUB

3.3 Memory

1. Basic storage elements are logical structures capable of storing some amount of data, contrasting combinational logic circuits
   (a) R-S latches can store a single bit, made of two NAND gates, the output of each connected to the input of the other, with two external inputs (S and R)
      i. In the quiescent state, S and R both begin as 1, such that whichever default input reaches first makes that output 1, at which point the other NAND gate returns 0, preserving the outputs
      ii. If the S gate returns 1, then the overall data value is 1, otherwise 0, such that S or R can be set to 0 quickly to make that gate return 1, setting the variable as one of the values
      iii. The term clearing the variable is used to denote setting it back to 0, such that clearing S or R makes that gate have a 1 output
      iv. Both inputs are not able to be set to 0, in which case the result will depend on the electrical properties of the transistors, rather than logic
      v. As a result, the S gate signal is taken separately to get the overall output of the circuit
      vi. Thus, it can either be drawn with both NAND gates with crossing outputs in the same direction, or with a cycle of NAND gates, taking the output of the S gate either way
   (b) Gated D latches allow it to be controlled when a latch is set or cleared, such that it is two NAND gates, each gaining input from D, the gate in series with the R gate inverted, such that it determines which is set
      i. Each of the D gates also have a write-enable signal, which determines if the byte is set in the first place, such that if it is activated briefly, one of the D gates stops signaling, setting the R-S latch based on D
   (c) Registers store a series of bits as a unit, made up of a series of gated D latches, designated as an array of bits, starting from [0] as the rightmost
      i. The same write-enable signal is given to each of the D latches, with a different D value given to each based on the value being stored
      ii. Subunits of the series of bits are designated $[l : r]$ where l is the value of the leftmost, r is the value of the rightmost, called a field of the register
2. Memory is made up of a large number of locations, each with a unique identifier, or address, each storing some amount of data, the amount of bytes/bits (1 byte = 8 bits), called the addressability of the memory
   (a) The total number of addresses is called the memory's address space, as some power of two based on the numbers of bits encoded as each memory address, such that the range is $[0, 2^n - 1]$
   (b) Most memory systems are byte addressable, such that they store a single byte per memory address, corresponding to one ASCII character, but computers are often 64-bit addressable to allow for double floating point numbers to be stored, used in large calculations
3. Memory is denoted as size m by n bits, where m is the address space, n is the addressability, made up of a decoder to determine which memory address row, called the word line, with the size of the addressability
   (a) The decoder sends the word line value into the mux, to determine which D latch provides

the value through the mux, sending it to each column of D-latch/mux combinations to give one bit of the word

   (b) The write enabled is connected to an AND gate with the decoder, such that the decoder signal must be set to the correct word line, along with the write enabled to set the latches
   (c) Each column of mux/D-latches is also connected to a D input value to set the byte if the write enabled is activated

3.4 Sequential Logic Circuit

1. Combinational logic circuits are those that have no ability to store data from the past, while storage circuits store data from the past
   (a) On the other hand, sequential logic circuits do both, such that they base the decisions on both the previous stored output and current input data, used for finite state machines
   (b) As a result, this is capable of monitoring sequences, such as a sequential rotation lock
2. Thus, sequential circuits rely on the state of the circuit, defined as the external data of the input and current setting fed in, combined with all prior operations in the sequence and the settings deemed relevant
   (a) Each step within the sequence is thus considered a state of the system, as a snapshot of the current characteristics of the system
   (b) Finite state machines are thus defined as a system with a finite number of states, external inputs, external outputs, as well as explicit specification of state transitions and what determines an external output value
      i. These are represented by a state diagram, drawing a circle for each state, with connections/arcs from the current state to each other subsequent/next state, each with the external input to provide that transition written at the base
      ii. The internal output values of the system are written near the state, designating the code provided for the state, with the external output written inside the state bubble
3. State transitions are often defined by a clock circuit, with a signal alternating between 0 and 1 cyclically in some amount of time, triggering a transition when on
   (a) Clocks are denoted as an input by a small triangle pointing into a circuit at the entry point, drawn with one edge as an edge of the circuit
   (b) The clock must have a long enough time that all charge transitions complete, creating the danger of overclocking
4. Thus, the combinational logic circuit aspect of the circuit is created based on the combination necessary, the storage element acting as a master-slave flip-flop instead of a gated D latch, which would modify the value stored immediately, rather than waiting until the next clock cycle
   (a) Thus, the master-slave flip-flop is made up of two D-latches for each byte, the output of one as the input of the other, the clock connected as the write-enabled, but negated for one, such that it is only able to change the value of one at a time
   (b) As a result, it writes the second when the clock is activated writing the value from the first, such that it is inputted into the circuit, and when the clock is deactivated, writes the first with the new value of the circuit
      i. The process of changing the feed-out value when the clock activates is called positive/rising edge triggered, while the process of changing the stored when it deactivates is called negative/falling edge triggered

        ii. The term clocked is used to refer to rising edge triggered generally, such that the register is written rising edge triggered, sent out falling edge triggered
            A. The PC is considered rising edge triggered as a result
        iii. The temporary register identifier bits are called register specifiers
    (c) Registers are generally used to refer to a set of flip-flops, rather than a set of latches, though may be used for either
5. Static RAM is thus made up of registers, maintaining value as long as power is applied, while Dynamic RAM uses a capacitor to hold the value instead, requiring refreshing due to leaking charge

4 Chapter 4 - Von Neumann Model

4.1 Hardware

1. Computers are made up of a program, or a set of instructions, each a well-defined work command of the smallest possible division, and a hardware device, such that a von Neumann model is simplest fundamental model for processing programs
    (a) It is made up of a memory, processing unit, input, output, and control unit (controlling the order of instructions processed), storing the program in the memory
    (b) In von Neumann/computer model diagrams, filled arrows are used to signify command/control signals, while unfilled arrows are used to signify data
2. The memory for a standard computer in modern day is 2^{28} by 8 bits, while the LC-3 has 2^{16} by 16 bits
    (a) The contents from memory are read by placing the location in the memory's address register (MAR), which then has the data stored there placed in the memory's data register (MDR), written by activating Write Enable signal, then writing the address and data value in each respectively
    (b) As a result, the MAR for an LC-3 has 16 bits, due to the address requiring that many to specify, while the MDR has 16 bits for one data piece as well
3. The processing unit is made up of at least one ALU, used for simple arithmetic and logic functions, with the length of the quantity evaluated called the word length, each value a word
    (a) In the LC-3, the word length is 16 bits, though most computers in modern day have either 32 bits or 64 bits (such as the SPARC-V9)
        i. Since the addressibility of the LC-3 is equal to a word, it can be called word-addressable
    (b) The ALU for an LC-3 is able to perform addition, bitwise NOT, and bitwise AND
    (c) Most processors also have a small amount of memory, called the TEMP, next to the ALU, to store data for calculations with multiple parts needed in the near future
        i. This is due to taking a long amount of time to access the main computer memory, relative to the time needed to do the calculation, generally with the TEMP having a certain amount of registers, each for one word
        ii. The TEMP is also called the register file, composed in the LC-4 of three MUX, a write enabled, an input, and creating two 16 bit outputs
        iii. In the LC-3, it has 8 registers (R0, R1, . . . , R7), while the SPARC-V9 has 32 registers

4. Input and output are made up of peripherals able to sense and display information, generally at the simplest the monitor and keyboard for an LC-3
   (a) A simple keyboard for an LC-3 requires two registers, a KBDR for the ASCII code of the last key hit and KBSR for maintaining the status of keys hit, while a simple monitor requires two registers, DDR for ASCII code being displayed and DSR for status information
5. The control unit keeps track of the order of instructions in the program, using the information register (IR)/program memory to contain the current instruction being executed
   (a) It also contains the address of the next instruction, stored within the program counter (PC), and a finite state machine to direct all activity taking input from the IR and from a clock (CLK)
      i. The clock is made up of a piezoelectric crystal oscillator to produce oscillating voltage, connected to an AND gate with a RUN latch as the other, determining if the computer runs
      ii. Older computers used the HALT instruction to turn off the RUN latch, while newer computers, such as the LC-3 use the operating system to do so to avoid wasting an opcode
   (b) The finite state machine has a series of command outputs, such as, within the LC-3, the two bit ALUK, controlling the operation of the ALU, or the GateALU, determining if the ALU output is sent out of the computer by the processor bus during that cycle
      i. The LD.MAR is the write-enabled signal for the MAR register, allowing the PC to be written into the MAR to get the subsequent command
      ii. The GatePC determines if the PC is connected to the processor bus, such that the data from it can be accessed at a particular point
      iii. The PCMUX allows it to choose the select line for modifying the PC in a specific manner, such as incrementing, giving the new value
      iv. The LD.PC is the write-enabled signal for the PC register, while the LD.IR is the write-enabled signal for the IR, allowing the values to be modified
      v. The GateMDR determines if the MDR is connected to the processor bus, such that the value from it can be sent to other parts of the system
      vi. The MARMUX determines where the address supplied to the MAR is from, either from a register or the PC, specified by the ADDR1MUX, which then determines what length immediate/zero to add to it by the ADDR2MUX
         A. The decision of which is controlled by a zero extended (positive sign) trap vector sent by the computer to determine which method is done
         B. The important of using the correct immediate length is noted by the risk of removing sections as overflow should it not be chosen correctly
      vii. It is noted that while these are used for LC-3, they are specific to a particular ISA, such that they are not universally created components

4.2 Instruction Processing

1. Instructions are made up of an opcode (command) on the left, written as 4 bits [15 : 12], in a 16 bit word, with the remaining 12 bits as the operands (data executing on)
   (a) Functions such as ADD or NOT are called an operate instruction, acting to process the data

   (b) Functions such as LDR are called a data movement instruction, due to moving data from one location to another
2. The commands are processed in the six phase instruction cycle, taking up a some number of machine/clock cycles, controlled by the finite state machine
   (a) It begins with the FETCH phase, taking the next instruction from memory by the address in the PC, loading it into the IR, after which the PC is incremented to point toward the next instruction, taking multiple cycles to access memory, the remaining steps each taking one
   (b) After, the DECODE phase studies the opcode to direct the command, sending an output line from the DECODE box to the processor in one clock cycle made up of all control signals (distinct from control commands), then the EVALUATE ADDRESS phase, which calculates any memory addresses needed
   (c) After, the FETCH OPERANDS phase takes the operands from either the memory from the EVALUATE ADDRESS phase, or from temporary storage, and then are used in the EXECUTE phase, finally using the STORE RESULT phase to finish the cycle
   (d) It is noted that while there are six phases, most commands such as the ADD (EVALUATE ADDRESS) and LDR (EXECUTE) do not require specific phases each
3. Control instructions are those which change the sequence of instructions, changing the PC after it is incremented, during the EXECUTE phase

5 Chapter 5 - LC-3 and LC-4

1. The LC-4 is a variation on the LC-3, using different formatting and cycle implementation in certain circumstances, but with generally similar conceptuals
2. The ISA specifies all functions of the computer that can be written for the computer in machine language, meaning Instructions Set Architecture for conversion from logic or high-level languages
   (a) The ISA is defined by the set of opcodes, data types of the operands, and addressing modes for a particular machine
3. The opcodes present in a particular ISA depend on the purpose of the computer, with varying amounts based on the level of complexity
   (a) Conversely, most computers prefer fewer large commands, rather allowing more variation in the instructions sent to maximize efficiency
   (b) The LC-3 has 19 different opcodes, with 1101 unspecified, each for a particular command, while the LC-4 uses 36 different opcodes, each with only 4 bits for it, such that it has certain opcodes with the first four bits overlapping, using additional bits to differentiate
4. Data types are representations of information that opcodes are able to operate on, supporting a specific format of data, such that LC-3 only supports 2's complement integers
5. Addressing modes specify where the operand is located, found either in the instruction mode (literal/immediate operand), register mode, or within memory
   (a) Addressing modes for within memory are either PC-relative, indirect, or base + offset modes, used only for data movement operands generally, using the other two for all operand types
   (b) Immediate operands are limited by the length of bits allowed within the command
6. Condition codes are single-bit registers found in the LC-3 and LC-4, set for any TEMP register writing function in LC-3, for any CMP function or any TEMP register writing in LC-4

   (a) The three registers are N (negative), Z (zero), P (positive), setting automatically to allow instruction sequencing as a result of the returned value
7. General purpose registers (GPR) are the registers/memory locations used for TEMP storage within the processor, each generally word-addressable, for the register addressing mode
   (a) It is noted that the same register can be used for both the source and destination
   (b) Loading is the process of moving memory to a register from the memory, while the process of moving away is storing
   (c) The LC-3 has seven data movement instructions, LD, LDR, LDI, LEA, ST, STR, and STI, while the LC-4 only has LDR and STR commands, with the first three bits after the opcode being the temporary register being used for loading/storing (destination register/DR)
      i. The temporary register identifier bits are called register specifiers
      ii. The remaining bits $[8:0]$ are the address generation bits, encoding the instructions for the data movement location/addressing mode
      iii. LDR and STR are the base + offset mode commands, with the leftmost three as the register for the base location (source register/SR), the remaining six bits as the literal offset value
      iv. LD and ST are PC-relative mode commands, providing all 9 bits as displacement from the current PC, after it has been incremented already
      v. LDI and STI are indirect address mode, providing all 9 bits as the displacement from the current PC, to a location which contains the address of the data in memory
      vi. LEA is immediate mode, only able to load into a register, adding the incremented PC counter to the immediate value, which is then loaded into the register, not needing memory access
   (d) The original valuing within an algorithm of variables is called the initialization
8. Control instructions are used to change the sequence of instructions, allowing conditional branches, unconditional jumps, subroutine/function calls, TRAP, and return from interrupt
   (a) The BR command is used for conditionals, checking some combination of NZP registers, if found, adding the incremented PC by the immediate offset given by the remaining bits for the new PC
      i. The use of all three registers is called an unconditional branch
      ii. Loops can either be run by iteratively, decreasing some counter each time the body of the loop executes, by iteration, or by sentinel, testing for an occurrence of some sentinel variable to stop the loop
         A. Sentinels are often unexpected character to search for, signifying the end of the sequence of expected characters
   (b) The JMP command allows movement to an address as an unconditional BR, with increased immediate length, such that there is a wider range
      i. The JMPR command allows changing the PC to any address contained within a register, to allow not using the offset within a wider range
   (c) The TRAP command sets R7 to the incremented PC, moving the PC to a memory address in the operating system, called an OS service call, the rightmost 8 bits as the trapvector
      i. The trapvector is an unsigned immediate that identifies the service call that the operating system is supposed to use
      ii. If the trapvector is 0, 0x8000 is used in the LC-4, while in the LC-3, an input

character from the keyboard is x23, output to the monitor is x21, outputting a string starting at the address in R0 (PUTS) is x22, and ending a program is x25

   A. The vector location signifies a request to move to an alternate program within the OS

 iii. The output and input character must be stored in R0 for the LC-3, which is the register character automatically stored and displayed

9. Coding in LC-3/LC-4 are done by using the mnemonic forms with spaces between the separate register codes, rather than the binary encoding or the semantic explanation
10. The global bus of the data path allows any structure within the computer to transmit 16 bits of information to any other structure, modifying the connections within the bus to determine the source and output, transferring one at a time, though some computers have multiple buses
    (a) Each component of the LC-3 has a tri-state device, drawn as a triangle, to allow the computers control to allow one device to send data through the bus at once, designated as the Gate control
    (b) On the receiving side, the LD (load enabled) control determines where the signal is being received
    (c) As a result, tracking the data path is the flow of data in the CPU through a single instruction cycle

6 Chapter 7 - Assembly Language

6.1 Assembly Programming

1. For programming-ease, machine language given by the ISA can also be denoted by assembly language, contained within each ISA, each command representing one ISA command
    (a) This is then often translated into a high-level programming language such as C, moving it closer toward human language, which are ISA-independent
        i. High-level languages provide less control over the individual commands, sacrificing it for readability
    (b) Each assembly command has a mnemonic name for opcodes, and allows addresses to be set equivalent to text symbolic addresses
2. The assembler program translates assembly language, also called assembling it, into machine code for the computer
3. Instruction lines begin with a label, then the symbolic opcode, symbolic or explicit (such as register codes) operands, and finally comments (written after a semi-colon)
    (a) The label acts as a symbolic address for the memory address at that label, either storing an instruction or data
        i. In LC-3 or LC-4, labels can be placed on assembler commands .FILL and .BLKW, setting that label as the current location within the assembler to define variable names
        ii. Labels are often used to designate the start of loops or functions, as well as the end of the program overall in LC-4
    (b) Immediate values are written non-symbolically, such that binary is prefaced by a b, hex by an x or 0x, and decimal by an #
    (c) Comments are purely to make the code more readable to other programmers, ignoring

all after the semicolon by the assembler
   i. Additional spaces and tabs are also used similarly, making the program more generally readable, though new lines can only be used at the end of a line properly
   ii. Comments are generally used at the start of the program to describe inputs, outputs, limitations, and the purpose of the program

4. Pseudo-ops/assembler directives are instructions directly to the assembler, not representing ISA opcodes, but rather to clarify the code for the assembler, denoted by a dot at the start
   (a) In LC-3, the .END command functions to tell the assembler when to stop reading, while the .ORIG replaces the .ADDR command
   (b) .FILL in LC-3 increments the current location within memory, then fills it with the value given, before incrementing again, while it fills the current value then increments in LC-4
   (c) On the other hand, pseudo-instructions are instructions that are used as shorthand for other instructions, converted into proper assembly instructions within the code
5. It is often wise to store the register values in some location, such that preexisting values are preserved
   (a) This is essential if the value of the register is modified within the program, while the original value is needed later

6.2 Assembly and Execution

1. The assembly process converts the assembly language into machine language by the assembler software, using the "assemble *filename*.asm *outfile*" command in LC-3 assembler
   (a) In LC-4, "as *filename output*" is used instead in the assembler, producing an object (.obj) file
   (b) The assembly and setup process in the LC-4 can be shortened by a script file, called by "script *filename*.txt", containing commands
      i. The reset command is used to reset all LC-4 registers to 0, while "set *register/PC value* is used to set the values of the PC and register before the program is run
      ii. The object file is loaded into the LC-4 by "ld *filename*"
      iii. Breakpoints in the program are set by "break set *label/address*"
2. The assembler requires a two-pass process due to the labels not necessarily being created before they are used
   (a) The first pass creates a symbol table, identifying the binary addresses corresponding to each label, while the second pass translates to machine code
   (b) During each pass, the current address code being created is kept track of by a location counter, incremented after
      i. As a result, the incremented PC for each command is the LC + 1
3. As a computer program is executed, it is done in executable image, made up of several object files linked together
   (a) Object files are either user-written and assembled, compiled from a higher level language, or built in libraries
   (b) Each object file is made up of ISA binary commands as a result
   (c) After it is linked into an executable file, the instruction cycle is used to run the file
4. In LC-3, multi-object files use ".EXTERNAL *variable*" in the assembly programs without the label, such that it would create a symbol entry for it, with a symbol notation for external
   (a) When the linker program is linking the modules, it identifies the values of each external

   variable
   (b) Programs working with each other can simply be loaded separately in, assuming no overlap in labels

7 Chapter 8 - I/O

1. I/O devices usually have at least two registers, one to display the current status, one for data
2. Older devices had special input/output instructions built-in to the ISO, while modern devices simply use standard memory instructions to special registers, called memory-mapped I/O
   (a) As a result, the MAR is loaded with the device register address, using logic to choose the device register rather than regular memory
3. Asynchronous I/O allows the devices to operate at a rate separate from that of the processor, due to the processor clock moving vastly faster than humans can input
   (a) Processing asynchronously requires a handshake protocol to coordinate, commonly using a flag 1-bit register, to indicate if the input/output has been processed up-to-date
      i. Thus, this Ready bit is set to 1 when data is inputted, 0 when processed for input, and 1 when received, 0 when displayed by output
      ii. The keyboard/monitor is then disabled until the data is processed properly, such that no data is missed
   (b) Synchronous I/O is a theoretical such that data is inputted at the same rate as the clock moves, such that a handshake is not needed
   (c) Interrupt-driven I/O is such that the keyboard notifies the processor when a key has been used, while polling I/O is such that the processor continuously checks the ready bit
      i. The program previously running does not have to be related to the I/O device, halted to allow the I/O to be dealt with, after which it is returned to as if there was no interruption
      ii. This is used to save processor power from having to constantly poll all I/O devices to receive characters
      iii. Thus, it requires an enabling mechanism to generate the interrupt signal and a mechanism to handle the signal
         A. For the signal to interrupt the processor, it must be more urgent than the task currently performed by the processor
         B. The ready bit determines if the signal would be sent, with the 14th bit of the status register as the interrupt enable, able to be written or cleared by the processor depending on if desired
         C. The interrupt request is the AND gate of the ready and interrupt enable bit
      iv. All processors have a level of urgency, PL7 at highest to PL0, such that a higher PL program causes the execution before other programs
         A. This is done by testing for the interrupt signal (INT) before FETCH phase, after STORE RESULT
         B. The priority encoder then determines which of the device interrupt signals is most important, and compares it to the current program
         C. After, the control unit saves the information needed to restart the program, and loads the PC with the new value, before FETCH

4. The keyboard in LC-3/LC-4 is made up of the keyboard data register (KBDR) and the keyboard status register (KBSR), contained within xFE02 and xFE00 respectively
    (a) Bits [7:0] within the KBDR are used to store the ASCII key value, the remainder containing 0
    (b) Bit [15] of the KBSR contains the ready bit, while the remainder contain 0
    (c) The display is similarly made up of the display data register (DDR) at xFE06 and the display status register (DSR) at xFE04, using similar storage distribution to the keyboard for console display
        i. In LC-4, it is called the ASCII display status/data register (ADDR/ADSR), stored in the same locations
    (d) For polling, they are not required to be writable by the CPU, though they are required to be for interrupted I/O
    (e) The timer measures the time in milliseconds, written from the initial set time in the Timer Interval Register (TIR) at xFE0A similarly
        i. The 15th bit of the Timer Status Register (TSR) at xFE08 is 1 when the timer expires, reset after the ready bit is read by the computer once it expires
        ii. This is used to prevent a program from running indefinitely timing out after some preset length of time
    (f) Video memory is stored in a 128 word horizontal length by 124 word vertical length array, starting in the top left from xC000, going up by one horizontally
        i. The memory stores the blue value in [4:0], green value in [9:5], and the red value in [14:10]
5. The memory-mapped I/O is controlled in LC-3 by the ADDR.CTL logic block, linking the MAR to the memory, such that the MIO.EN signal determines if movement from I/O memory is used
    (a) The R.W signal determines if data is being loaded or stored, coordinating the MAR, MDR, memory, and I/O
    (b) If the load signal is indicated, it controls the INMUX to pass the correct data from memory/IO to the MDR

8 Chapter 9 - TRAP Routines and Subroutines

1. Due to hardware I/O registers being accessed by a large number of programs, giving the programmer complete control could pose problems to other programs
    (a) Thus, hardware registers are considered privileged, requiring sufficient privilege to access it
    (b) As a result, I/O is generally only accessible by the TRAP instruction working through the OS, which has privilege to access them
        i. Within the LC-4, this is done by moving into supervisor mode, which is able to execute both user and non-user data
        ii. This is designated in the LC-4 by the Processor Status Register bit 15, containing 0 for user mode, 1 for the supervisor mode
            A. The NZP sequence is similarly stored in there, with NZP as bits [2:0] respectively
    (c) Thus, the TRAP instruction is said to create a service/system call, taking control of the computer from the program until after the call

2. The TRAP mechanism is made up of a set of service routines, built-in to the OS, a table of starting addresses for each instruction, the TRAP instruction itself, and a linkage to return control to the original program
   (a) Thus the TRAP instruction moves the PC to the routine, but creates the linkage to return to the program after (storing the incremented PC in R7), jumping at the end
   (b) Within the LC-4, it returns by the RTI instruction at the end of the TRAP instruction sequence, turning off supervisor mode
   (c) TRAP codes are written themselves with the assembler directive ".OS" at the start before ".CODE", written above both the table portion and the instruction portion of the file
3. TRAP is able to shut off the clock by the RUN latch, stored within bit 15 of the Machine Control Register, located in xFFFE
4. TRAP routines and subroutines, due to using registers within the routine, need to save the initial values of those registers beforehand, or during the routine, restoring when needed
   (a) Caller-save is used to describe if the calling/original program saves the registers, while callee-save describes if the routine saves the registers
   (b) Within the LC-4, TRAP commands are a combination, using caller-save for the general PC-restoring R7 register, callee save for any used by the individual routines
5. Subroutines are called similarly to OS TRAP routines, but rather written by user programmers, often contained within libraries
   (a) These are also called procedures or functions, used by a call/return mechanism, distinct due to the lack of OS privilege without risk of damaging other programs
   (b) The call mechanism calculates the starting address of the subroutine, loads it, and saves the return address for the return mechanism to load into the PC
   (c) This is done by JSR and JSRR, the latter moving to an address in a register, the former moving to the label by the offset
      i. JSR must have the label must be .FALIGNed, due to adding the bits after the first 4 as 0s, to extend the range
      ii. JSR is either the offset from x0000 or from x8000, depending on the original location
      iii. Both save the incremented PC within R7 for the return mechanism
   (d) Libraries are used to save time and prevent the programmer from having to know the details of every process required
      i. As a result, only a pseudo-op .EXTERNAL combined with documentation are needed by the programmer to use the library

9 Supplemental LC-4 Notes

1. The storage is made up of x0000-x1FFF for user code, x2000-x3FFF for user global variables, x4000-x6FFF for user dynamic storage/heap, and x7000-x7FFF for user local storage/stack
   (a) The trap vector table is then from x8000-x80FF, the OS from x8200-xBFFF, video output/memory from xC000-xFDFF, and xFE00-xFFFF for I/O device registers
   (b) Only the user portion can be accessed and modified, the code portion only able to be when loading a program, the memory portion from within the program
2. "reset" puts the computer back to the initial state, "as *filename filename*" assembles it, and "ld *filename*" loads it in

   (a) "set *register value*" is able to set an initial memory location, PSR, or temp registers
   (b) These system commands can be written within a script text file, run by "script *file-name*.txt"
      i. Unlike within LC-4 code where hexadecimal is default, within scripts, decimal is the default and binary is not permitted
3. In the documentation, PC + 1 means that the incremented PC is used, rather than that it is incremented a second time
4. .FILL automatically increments the address after being done
   (a) .BLKW moves the assembler address to after the block, setting all values to 0 within, and prevents the assembler from editing the value, returning an error if it tries, though the program is able to
   (b) .CODE and .DATA directives signify which portions of the memory are able to be written by the code at any point
   (c) .FALIGN moves the address to the next multiple of 16 within the assembler
   (d) .STRINGZ adds a x0000 character at the end of the string to designate the end
   (e) .CONST/.UCONST are replaced by the assembler to the correct value during assembly, such that it is shorthand, not a variable
5. CONST inserts a signed value (-FF-1, FF) into some memory location, removing prior data in that location
   (a) HICONST then adds an unsigned value (0, FF) into bits [15:8] of the memory location, preserving the remainder
   (b) Thus, when combined with CONST, can be used to fully set the memory location, removing the sign bit from CONST, such that it must be done after
6. Upper/lowercase letters for instructions or registers are irrelevant, such that they are corrected to capital by the compiler, except the nzp after BR, corrected to lowercase
   (a) BR is used as shorthand within code for BRnzp
   (b) Tabs are ignored by the assembler, but newlines imply the next instruction begins, assuming it is not a blank line
      i. Newlines are able to be used for an extra line spacing between instructions, and are able to separate the label from the instruction

10 Chapter 6 - Programming

1. Programming is divided into problem solving, converting the natural language problem into code that the computer can convert into electrons, and debugging, removing errors/bugs in the program
   (a) Thus, the first step is to convert the problem into an algorithm with finiteness, definiteness (precisely stated), and effective computibility (able to be done by a computer)
2. This is commonly done by structured programming/systematic decomposition, breaking the overall task into smaller steps, until it can be done by the computer
   (a) The three constructs are sequential (each task performed until completion without option), conditional (multiple tasks, possibly vacuous (doing nothing), based on a condition), or iterative (repeating a subtask until some test returns a value)
   (b) The constructs are also made such that all three to have one entrance and exit into it definitively, each assumed to be able to be done by the ISA/programming langauge

   (c) The process of breaking down the task is called stepwise refinement, breaking down each complicated task step by step
      i. The task is initially broken up into three sections, initialization, calculation, and output
      ii. Each is then divided into one or more constructs, until it is purely within manageable tasks for the programming language
   (d) The stepwise refinement is completed by replacing the human-termed variables with the actual computer variables initialized, followed by actual coding
3. Debugging is generally done by tracing the program, keeping track of the sequence of instructions and what it produced, to identify incorrect results
   (a) This is often done by breaking the program into modules, checking the results of each module, easily working with structural programming
   (b) For assembly language, this is done by setting specific values to see the resultant output, execute sequences using breakpoints or single instructions (single-stepping), and displaying values
      i. Breakpoints allow the result of each module or iteration, rather than each instruction, to be analyzed for errors
   (c) One common bug is often corner cases as well, working for the more extreme inputs, or infinite loops

11 Chapter 11 - Introduction to C

1. Higher-level languages were necessary as programs became more complicated to automate aspects of the process
   (a) This is done by providing symbolic names to values, without having to allocate storage or move data to access it
   (b) In addition, it allows expressive representation of calculations and tests, rather than specific line formatting, and improves readability of the code
   (c) It also has checks for bugs and error messages to aid the programmer, and has more extensive libraries
2. High-level languages can either be interpreted (executed by an interpreter program which reads it), or compiled (translated into an executable machine language)
   (a) Interpreters are also called virtual machines, executing the program as input data in a sequential manner, such as Perl, LISP, BASIC, LC-3, or LC-4
      i. This allows regions to be executed at a time, allowing easier debugging and code modifications, but longer execution
   (b) Compilation requires only one time for unlimited running, but requires analyzing the complete program as a full unit, used in C and FORTRAN
      i. This allows execution quicker and more efficient resource usage, but renders debugging during the process more difficult
3. C is made to allow writing compilers and operating systems, such that it is a high-level, but near-low-level language, allowing bitwise and memory manipulation
   (a) As a result, C has been a fundamental basis of many modern languages, such as Java or C++
   (b) C was made machine independent, also called portable by the ANSI C version, used by

        most compilers, specified by the American National Standards Institute
    (c) C is also considered a procedural language, made up of functions

4. C compilers are made up of a preprocessor, finding preprocessor directives (pseudo-ops), adding header files or constants
    (a) Preprocessor commands are begun by #, generally using "define" to create a compiler macro constant, and "include" to add the code from a header file in, not ended by a semicolon
        i. Adding header files in can be used to declare variables and functions, as well as define constants, which are standardized to multiple source files, often used to declare library functions
        ii. <> are used to surround library header files within the system library storage, while "" are used for files within the same directory
        iii. Macros can take arguments, specified in parenthesis next to the definition how it is evaluated to get the value
            A. After each occurance as a result, there must be parenthesis with the input values/variables to be evaluated, acting similar to a function call
            B. It then replaces the marco with the expression, inserting those variables, during compilation
    (b) The compiler then produces object modules, using a symbol table, parsing code in two stages (analysis/parsing into sections, synthesis/generating and optimizing code)
    (c) The linker then links all computer and user object modules together to create an executable image, finding library files in a specific OS location
5. Functions/subroutines are defined in code by a function definition, with the main() function, which must return an int, as the start of program execution
    (a) Functions are made up of variable declarations, and statements of the function (which express actions performed within the function)
    (b) Declarations and statements are ended with a semicolon, to allow the compiler to easily break it into blocks, though preprocessor commands do not
    (c) The main function can also have parameters, "int argc, char** argv", allowing command line inputs, where argc is the number of inputs including the executable file call itself and argv is the array of each string input
6. Input and output in C are done by library functions, commonly within "stdio.h", using formatted strings
    (a) Formatted strings have text to print and specifications on how to print values within the text, represented by format specifications starting with %
    (b) Output is written as "printf(*string, value1, ...*)", input done similarly by scanf(), either providing the specific value or the variable
        i. It is noted that scanf must have the values as pointer variables, providing where to store the input

12 Appendix D - The C Programming Language

1. Conventions within the C programming language is divided into source files (.c) and header files (.h)
    (a) The source files are compiled into an object, linked into an executable

- (b) The header files contain function, variable, structure, and type declarations, and preprocessor macros, using one for each source file, included at the top
- (c) Comments begin with "/*", ending with "*/", able to span multiple lines, unable to be nested or within characters/strings
- (d) Literal constant values are able to be used in expressions or used to initialize variables, defined bitwise as in assembly, explicitly written
  - i. Integer literals beginning with 0 are octal, 0x are hexadecimal, otherwise a decimal, with the minus sign before the marker
    - A. l or L are used as a marker for a long int, while u or U are used for an unsigned int
  - ii. Floating points are written with a decimal point and either a fractional, integer portion, or both, presided by a minus for negative values
    - A. Floating point can also be written in scientific notation, using either an e or E at the start of the integer exponent
    - B. It is defaulted as a double, though f or F can be used as a float marker, and l or L is used for a long double
  - iii. Character literals are surrounded by single-quotes, written in normal text, converted by the computer into ASCII
    - A. \0 or \x are able to be used to include a non-keyboard character by the ASCII code
    - B. Special characters often also have a character escape code to include it in a string, beginning by "\\"
  - iv. String literals are surrounded by double quotes, signified by the type char*, allocated in a special portion of memory for literal constants, automatically given the '
    0' null character at the end
    - A. It is noted that string literals cannot be used to fill an array after it has already been initialized
  - v. Enumerator constants are symbolic integer values, defined in an enumerator list of a declaration
  - vi. Literals are generally avoided, using constants instead to give symbolic representation to the value
- (e) C is freely formatted, such that spaces, tabs, new lines, and other formatting characters are able to be included when needed for readability
  - i. It is convention to comment at the start of code when it was last modified, by whom, inputs, outputs, and the purpose of the function
- (f) Keywords, used within the C language itself, are not able to be used as any form of object name
  - i. Object names are defined as the name of any named portion of memory, such as a variable or function

2. C basic data types are int (32 bits), float (32), double (64), and char (8), with char and int as integral types, the others as floating types
   - (a) While it is different sizes on some machines, all tend to follow the IEEE standard
   - (b) LC-4 assembly only allows integer types of variables, with 16 bit ints rather than 32
   - (c) The short int must be at least the size of a char in C, and the double must have a higher precision that that of a float
   - (d) "signed" and "unsigned" specifiers can be added to integral types, the former by default

    (e) "long" and "short" specifiers can be added to either extend or shorten the length of the integral types and can shorten or lengthen the precision and/or range of floating types, changing the bits by a factor of 2
    (f) "const" specifiers can be added for variables which are not changed to optimize compilation, requiring explicit initialization at the time of declaration
3. Enumerated types allow objects to be specified of constant value from 0 onward, defined by the specified, "enum *name* {*obj0 = first_constant, ..., objn*}", declared by "enim *name obj*"
    (a) The objects are called enumeration constants, set such that the first is 0 by default, progressing up integrally from the first
4. Derived types are extensions of basic types provided by C, consisting of pointers, arrays, structures, and unions
    (a) Arrays are sequences of objects, such that the element i+1 is at the location of i + sizeof(type), such that the type must be declared at the start
    (b) Pointers are objects containing the address of other objects, able to be manipulated in expresses by pointer arithmetic, adding integer values and comparing values
        i. Pointer arithmetic automatically adds or subtracts sizeof(type), where type is the pointer's type, from the pointer when adding or subtracting 1
    (c) Structures are aggregate types defined by "struct *tagid* { *type1 var1; ... typeN varN*;};", able to contain any other types within
        i. The tag ID allows variables of the structure to be declared, by "struct *tagid varName*", and allows multiple structures with the same member elements and identifiers to exist in the same program
        ii. Variables can also be declared without a tag, at the end of the struct definition for a single usage
        iii. It is allocated in memory by the same manner as fundamental types, given enough space for all member variables
        iv. Member elements are called by "*structVar.memberVar*", or called by a pointer to the struct by "*structVar-¿memberVar*"
            A. This is due to the variable name providing the location of the start of the struct, and the member variable name providing the offset from the start
    (d) Unions are a single object able to take on multiple different types during the program, storing enough data when declared for the largest possible type
        i. It is defined similar to a structure, with "union" replacing "struct", calling on member items the same, though only filling on at a time
    (e) Structs and unions can directly access a member element by *structUnionName.varName*, or if using a pointer to the struct/union, can use *structUnionPointer-¿varName*
        i. Structs and unions are automatically passed by value to functions, similar to all other variables, except arrays and pointers
    (f) "typedef *type name*" allows a type to gain a pseudonym, commonly used for derived types as shorthand
    (g) "sizeof(*type*)" returns the number of bytes occupied by the type, or the total size of the array/struct
5. The return statement, written "return *expression*;", terminates the current function, returning the expression value
    (a) All functions have an implicit return statement, with an undefined/void return value
    (b) The expression is automatically converted to the type which the function is defined to

return

   (c) "return;" is able to be used to terminate a function, without returning any value

6. I/O functions are within ¡stdio.h¿, consisting of "int getchar()" to read the next character from the standard input device/stdin
   (a) It uses the buffered I/O in the keyboard, taking in all prior characters when enter is pressed, giving each from the buffer when requested
   (b) "putchar(int c)" prints a character into the standard output often buffered, such that it isn't flushed until a newline is placed in the output stream
   (c) "int printf(char* format_string, var1, var2)" prints the format string, filling each conversion specification with the variable listed, returning the number of characters or a negative value for error
      i. %d is signed decimal, %o is octal, %x is lowercase hex, %X is uppercase hex, %c is character, %e is lowercase exponential notation, %E is uppercase exponential, %s is a string, %u is unsigned integer, %f is floating
      ii. Special characters can also be written starting with
      , such that
      n is newline or
      t is tab
   (d) "int scanf(char* format_string, *ptr1, *ptr2)" takes a stdin input based on the format string, putting it into each pointer given, returning the number of variables assigned
      i. It uses the same specifiers as printf, with the added %lf for a double floating point
      ii. It is noted that string pointers are regular string variables, such that it should not have the address operator added to it
7. String functions are stored within ¡string.h¿, including "int strcmp(char* A, char* B)" and "char* strcpy(char* A, char* B)", "strncpy(char* A, char*B, n)", where n is the max number of bytes to copy
   (a) Strcmp checks if each character is identical in A and B, at which point when a character is greater in ASCII, if it is in A, it returns ¿0, in B ¡0, if equal, 0
   (b) Strcpy copies every character from B into A
   (c) It also includes "strcat(char *d, char *s)", appending s to d, and "strlen(char* s)", returning the length of s not including the null character
8. Math functions are within ¡math.h¿, each taking doubles and returning doubles, including sin, cos, tan, exp (e^x), log (ln), sqrt, and pow(x, y) (x^y)
9. Utility functions are within <stdlib.h>, including "void *malloc(size_t size)", "void free(void *ptr)", "int rand()", and "void srand(unsigned int seed)"
   (a) malloc allocates the number of bytes given, within the heap, returning a pointer to it, with the parameter automatically typedefed to the unsigned integer size_t type returned by sizeof()
      i. Malloc returns a null pointer (the NULL macro, generally 0) if the memory cannot be allocated
      ii. free returns the allocated memory to the heap, having been previously allocated by malloc
      iii. Void* implies a generic data pointer, needing to be typecasted to the desired type, though free() automatically typecasts it
   (b) rand() generates a random integer based on the seed value, giving an integer from 0 to RAND_MAX (slightly more than 32.767)

    i. srand() allows reseeding of the rand() to produce a different pseudo-random integer

10. Character type functions are stored within "<ctype.h>", such as islower(), isupper(), tolower(), and toupper(), checking and modifying strings to the correct type of character
11. The assert command is contained within "<assert.h>", checking some condition, allowing the program to continue if true, but if false, ending the program and asserting an error into stderr

13 Chapter 12 - Variables and Operators

1. Variables hold values within a program, while operators are used to manipulate the values
   (a) Variables store data items, using symbolic representation, requiring declaration, written as "*storageClass typeQualifier type name;*", reserving that amount of memory for the variable before it is used
   (b) Declarations are also able to contain an initializer on the same line, such that it is written "*type name = value;*"
   (c) Each basic variable type (char, float, double, int) generally take up one memory location within C, though may contain differing numbers of bits (32 for int/float generally, 16 for char, 64 for double)
   (d) Variable names are called identifiers, with specific rules, such as only the first 31 characters being used, case-sensitive, and only alphanumeric and the underscore characters used
      i. Convention has all uppercase used for symbolic preprocessor values, a starting underscore for library, users using either camel-case or underscores between lowercase words
      ii. Variables are also often given names that allow the code to be more clear to readers
   (e) Variables are given a scope of local (in a specific block of code) or global (in the whole program) based on where they are declared
      i. Global variables are declared outside of any function or loop, while local variables are restricted to the function/control block in which they are defined
      ii. Blocks are defined as any portion of code enclosed within curly brackets, able to be defined without a defining use to change variable scope
      iii. Local variables are allowed to have the same name as local variables from other blocks, less vulnerable to be changed in other parts of the code, and more reusable
      iv. Variables have the scope and value of the most recently defined version, such that they can be nested, redeclaring locally within a block, reverting back to the original outside
   (f) Local variables are not cleared when declared, such that unless they are initialized, they could have a random value
      i. Global or other static storage class variables are automatically initialized to 0 when declared
2. Operators are combined with variables and literals to form an expression, grouped together to form a complete statement
   (a) Statements are terminated by a semicolon, generally modifying some variable or external factor
   (b) Expressions are defined as any legal combination of constants, variables, operators, and

function calls that evaluate to some specific type

- (c) Statements are able to be grouped together by curly brackets into a block/compound statement
- (d) The statement “;” is called the null statement
- (e) The assignment operator, =, evaluates the right hand side, assigning it to the variable on the left
  - i. Variables are then converted into the type the variable was declared as
- (f) Arithmetic operators return the highest length type given to the operator, with floating points considered longer than integral types, integers longer than characters
  - i. The main operators are * (multiply), / (divide), % (modulus), +(addition), and - (subtraction), with parentheses used to clarify the order of operations
  - ii. Modulus operators must have integer operands, but the others can have any variable types
  - iii. Expressions of two of the same type return the type designated by the operator, while for those of multiple types, the smaller is converted into larger bit size and integers are converted into floats
- (g) Bitwise operators include | (or), & (and), (̂xor), (not), << (left shift), and >> (arithmetic right shift), usable only on integer types
  - i. If an integer is shifting more than the amount of bits it takes up, it is considered undefined
- (h) Relational operators include > (greater than), >= ($\geq$), <= ($\leq$), < (less than), == (equal), and != (not equal), returning 1 if true, 0 if false
- (i) Logical operators evaluate values where nonzero is considered logically true, zero logically false, including && (and), —— (or), ! (not), returning 0 if false, 1 if true
  - i. AND and OR are short-circuit operators in C, such that if the value is found after only checking the left operand, the right is skipped
    - A. This results in any additional effects not taking place, such as incrementing the value of the right operator
- (j) Increment operators add or subtract 1 to a variable, returning the value of the variable (++, –)
  - i. If the operator is placed before the variable, it is in prefix form, returning the value of the variable after incrementing, placed after for postfix, such that it returns the value before
  - ii. This is used to change a variable value without the assignment operator
- (k) Joint assignment operators are combined arithmetic/bitwise and assignment operators, placing the assignment operator directly after without a space, such that the resultant value is assigned to the left variable
- (l) Conditional operators allow simple decisions to be made, such that it is written “x = a ? b : c”, meaning if a, then x = b, else x = c
- (m) The order of operations begins with left-associative (), [], -¿, and . operators, followed by postfix increment, then prefix increment (right-associative)
  - i. The indirection * operator, & address, unary (+/-), logical and bitwise not, “(type)” typecast, and “sizeof” operators then are next with right associativity
    - A. Unary operators are placed before the variable, such that - gives the negative of the operand, + gives the operand
    - B. Typecasting explicitly temporarily changes the type of the variable used such

that it is done "*(newType) var*"

   ii. It is then followed by *, /, %, then +, -, then shifts, then comparisons, then equality, then bitwise AND, then bitwise XOR, then bitwise OR, all left associativity
   iii. After is the logical AND, then logical OR, then ?: conditional expressions, all left associative
   iv. Last in the order is joint assignment operators with right associativity
   v. Associativity is the order for multiple operators with equal precedence

3. The compiler uses a symbol table, similar to LC-4, creating an entry each time a variable is declared, allocating the storage
   (a) Each symbol table entry states the name, type, memory location, and the scope of the variable
      i. The memory location is written in offset form, generally using negative offset
   (b) Variables are allocated within either the global data section or the run-time stack within memory, the former for "static" or global variables, the latter for all other variables
      i. The offset thus describes the offset from the start of the storage region, thus in terms of a pointer to the region, such that the global pointer (R4) is a built in pointer to the start of the global data section
      ii. Local variables are allocated in an activation record/stack frame/memory template, with the pointer to the final item called the frame pointer
         A. Each function call creates an activation record for that particular invocation, with a new frame pointer (R5) for the function
         B. When called, the activation record is put on the top of the run-time stack, moving the stack pointer (R6) to the new top of the stake, removing the record when the function completes
         C. As a result, variables declared locally first have an offset closer to 0, moving further negative after
      iii. The program itself is also given a portion of memory, as is the dynamically allocated data heap and the OS
4. Variables also have a storage class property, determining if it loses its value when the block in which it is declared completes, using the "static" declaration, acting as global variables
   (a) Local variables are by default in the automatic storage class, while global are by default in static
   (b) Local variables declared static are private to its block, such that it cannot be accessed/modified outside a block in which it is declared, but are able to be redeclared to be accessed further, not removed after the block
   (c) Automatic class variables can also be given the "register" qualifier, indicating it is commonly used, and thus should be placed in a register for efficiency
      i. On the other hand, if this cannot be done, it is ignored by the compiler
   (d) Both functions and variables can also have the qualifier "extern" added to it, stating the storage is defined in another object module for linking

14 Chapter 13 - Control Structures

1. Control structures allow the dividing of an algorithm into the three programming constructs
   (a) It is noted that all variables must be initialized outside the loop if it is to remain outside

the loop

2. Conditional constructs are divided into if and if else statements, written as “if (*conditional*) *action*; else *action*;
   (a) These are able to be nested to form else-if statements, automatically connected an else to the closed unattached if
   (b) Action blocks can be surrounded by curly brackets to allow multiple statements within
   (c) It is noted that == is the equality operator, while = is the assignment operator, commonly causing errors
   (d) 0 is a false conditional, while any nonzero value is considered to imply truth
3. Iterative constructs are divided into while, do-while, and for statements, the first written as “while (*conditional action*”
   (a) While loops are generally used for a sentinel condition, and can be used for a counter condition
   (b) For loops are used generally for counter controlled loops, written as “for (*variable-initialization; conditional; reinitialization*) *action*”
      i. This defines the initial counter state, then the method of modification after each iteration of the loop
   (c) Do-while loops are similar to while loops, except with the condition checked after each iteration, rather than before, written as “do *action* while *(text);*”
4. Other C constructs are the switch, break, and continue constructs, the first written as “switch (variable) { case *value*: *action* break; . . . default: *action*}
   (a) Each case must be checking for a constant value of the variable, executing it otherwise, then leaving the block by the break command
   (b) The break command is not required and can also be used to leave any other iterative control structure
      i. The continue statement is used to immediately move to the next loop iteration
   (c) The default is used if none of the cases are found, also not required for the construct
   (d) Case and default are both labels, such that the first line after them are not able to be a declaration

15 Chapter 16 - Pointers and Arrays

1. Pointers in C are defined as the address of a memory object, indirectly accessing them, while arrays are sequential lists of the same type of data, also indirectly accessed
2. Pointers allow arguments to be passed by memory instead of by value into a function, such that the memory itself can be changed, rather than simply modifying the variables in the function stack temporarily
   (a) The process of using pointers instead of calling a function by value is called calling by reference, giving the memory addresses
      i. Pointers also allow a function to return multiple values, since the return function can only provide a single variable
      ii. This then returns -1 to show an error, 0 to show no error, adjusting the input memory locations instead
   (b) Pointer variables are associated with the type of object they point to, such that it is declared by “*type *varname*”, storing the memory address

        i. It is declared like this due to stating that the variable is of that type when dereferenced
    (c) The address operator, &, returns the memory address of the variable, allowing it to be stored in the pointer, written "*pointer-var* = &*variable*;"
    (d) The indirection/dereference operator, *, can be used before a pointer variable to allow accessing the data stored at the location in the pointer
    (e) Null pointers are able to be made by initializing to "NULL", a built-in macro for an invalid pointer value
    (f) For types with more than one byte, the pointer to the variable containing it points to the starting point of the variable
    (g) Pointers are able to point to pointers, called double pointers, functioning similarly to single pointers, allowing passing a pointer by memory

3. Arrays are declared based on the type contained within by "*type name[size]*", unable to have their size changed after declaration
    (a) Arrays must be given an explicit size within code, rather than a variable size, or a syntax error will occur
    (b) Items are able to be accessed by index, such that the ith element in the array is taken by "*name[i-1]*"
        i. Indices are useful in being able to be represented by any integer representation, including symbolic
    (c) Arrays can be passed as parameters, automatically done by reference, due to the array variable name acting as a pointer to the first item in the array
        i. As a result, "*name[1]*" is equivelent to "*(*name* + 1)", such that the bracket operator automatically dereferences the variable at that location
            A. It is noted that while pointers can be reassigned as variables, array variables are fixed to the location for the duration of the stack
    (d) Arrays are also used for strings, acting as a character array, with a sentinel null character '
    0', called null-terminated strings
        i. String literals are written within double quots, rather than single quotes, written in printf() by the %s specifier
        ii. The compiler automatically adds the null character to the end of the string
        iii. The length of the string array states the maximum possible length of the string, but does not specify the actual string character length
            A. If the scanf() call exceeds the length of the array, it will overwrite variables stored after the string, writing remaining characters after the array
            B. This is due to the lack of protection built-in to C against exceeding array size to save time
        iv. It is noted that in C, scanf() only reads until the first white space for a string, keeping the remainder in a buffer for future scanf() calls
        v. String specific functions are found within the C standard library, declared in string.h

4. Higher dimensional arrays are considered to be an array of arrays, denoted by *type name[2D][1D]*, used in the same manner

16 Chapter 10 - The Stack

1. The stack is an abstract data type, such that it is defined by the operations used with it, rather than the implementation
   (a) It is based around the idea of Last In First Out (LIFO) storage, pushing an element on the top to add it, popping it to remove it
   (b) As each item on the stack is popped, the elements still within the stack move towards the top
2. Stacks are generally implemented by a stack pointer, keeping track of the top of the stack, with a fixed bottom, keeping the data in the same place in memory
   (a) The pop function then returns the top value, and removes it, such that the pointer moves down, while push adds a value to the top and moves the pointer up
      i. This creates the risk of an underflow, popping the value below the base of the stack, causing data storage errors, requiring checking for the bottom, returning 0 if allowed
      ii. Similarly, there is the risk of overflow, checking to make sure the stack isn't full, keeping a pointer at the top as well, returning 0 if allowed
   (b) Stack protocol states that data can only be accessed from the stack through the pop and push functions
3. While the LC-3 is a three-address machine, taking two input and one output registers, due to using general purpose registers for calculations, most calculators use stacks
   (a) Stack calculators are called zero-address machines, due to purely acting on the stack values
      i. On the other hand, most calculators have similar operations, implicitly pushing and pulling on a hardware level
   (b) This allows holding intermediate values on the stack, performing operations by popping out enough values, pushing the result
4. Interrupt-driven IO, in addition to relying on the interrupt bits and priority levels, require a process to manage the transfer of data
   (a) This process is made up of a initiating an interrupt, servicing it, and returning from it, such that when a higher priority device causes the INT signal to be expressed, it is prepared
   (b) First, the state of the program is preserved, saving the PC and PSR (which includes the original program priority in bits [10:8]), assumed that the service will save any registers modified
      i. In the LC-3, this is saved in the Supervisor stack, acting the same as the regular stack, but in a protected portion of memory, replacing the User stack for when supervisor mode is activated, using R6 as the stack pointer still
         A. As a result, there are two registers, Saved.SSP and Saved.USP to save the stack pointers when in the other stack
   (c) It then loads the service PC and PSR (privelege mode with NZP = 000), generally using 8-bit vectored interrupts, similar to TRAP vectors
      i. These are sent along with the priority level and the interrupt request signal at the start
      ii. The vector used by the processor is designated as the INTV, corresponding to a vector from x0100 to x01FF, after the trap vectors of x0000 to x00FF
   (d) Returning from the interrupt pops the PSR and PC from the supervisor stack and

restoring them, removing the remaining stack from the interrupt request

5. Queues are a similar abstract data type, acting through LIFO, or last in first out, built in the same manner

17 Chapter 14 - Functions

1. Functions are subroutines, creating building blocks for larger programs to be built, allowing abstraction
   (a) C is built around functions, using the main function to begin and end execution within a program
      i. The main function is required to return an integer type, generally using 0 to state no error
   (b) Functions are generally used if there is more than one call to it within a program, or if it is made into a library function
   (c) Functions within C have the property of being caller-independent, such that they do not change based on the caller
2. Functions are declared first by a function prototype at the header of the file, giving the compiler information about how the function should run/be written
   (a) This is written by "*return_type name(input1 name1, ..., inputn namen)*;"
      i. The return type can be any data type, or can be void for no return
      ii. If the function does not have a return statement, but does have a return type, the last statement value is returned
      iii. The parameters must be in the same order they are written in the definition of the function, called the formal parameter list in the definition
      iv. The prototype is not needed if the function is defined before it is used, but it is still considered correct syntax to include
   (b) Functions when called by the program or another function, take arguments, which are the specific inputted parameters
   (c) The function definition has a similar first line to the declaration, though without a semicolon, instead followed by brackets enclosing the function body
3. Each function has an activation record within memory for its global variables, created when the function activates
   (a) It cannot be placed at a specific location, due to the risk of recursion causing it to be overwritten, instead having each function get an unused region, by a stack
4. Functions are called, at which point the caller function copies the parameters onto the top of the run-time stack, right-to-left, and the callee function pushes its activation record data onto the stack
   (a) The callee then pushes a memory location for the return value to the top of the stack, and it saves any data needed for the original function, first as the return location
      i. This then includes the caller's frame pointer, copied as the dynamic link to return control to the caller
   (b) The callee also proceeds to allocate space for any local variables needed in the runtime stack
   (c) The frame pointer is then set to the beginning of the local variables, and the stack pointer to the top of the local variables

            i. This results in all of the functional data stored in the non-local variable region, but still considered within the activation record for the callee
   (d) After the function runs, it writes the return value, pops off the local variables, restores to the dynamic link and return address, returns control to the caller function
            i. The return value and parameters are then popped off by the caller function, after pushing the return value into the caller stack

18 Chapter 18 - I/O in C

1. The C standard library contains a large database of components for complex software, accessed by including the header file at the start, including preprocessor micros and function declarations
   (a) I/O functions are mainly contained within ¡stdio.h¿
   (b) The library functions themselves are linked from object files within some C library system directory if declarations are detected in the main source file
            i. Certain libraries are also dynamically linked, not linked when producing the executable program, but rather during execution each instance
2. I/O is based on streams, adding characters typed into the end of the stream, reading from the beginning, or vice versa for output, read from the streams as needed
   (a) The standard I/O streams are called stdin and stdout, controlled by getchar and putchar functions, contained in ¡stdio.h¿
            i. "void putchar(char)" adds the ASCII character inputted as the parameter to the output stream to print
            ii. "char getchar(void)" waits for the next character to be added to the input stream, by default the keyboard, returning the ASCII value
            iii. stderr is the third input stream, appearing similarly to stdout, except used purely as the console for errors
            iv. stdin, stdout, and stderr are built-in file pointers to refer to the input and output consoles in which the program is being run
   (b) Buffered I/O adds all characters into a buffer array, waiting for the ENTER/NEWLINE key to release them into the stream at once
            i. This allows the input to be confirmed by the user before being taken by the program, erasing mistakes
            ii. Buffered I/O is automatically built-in to C in putchar() and getchar() functions, and thus must be accounted for
   (c) Formatted I/O handle more variable types and longer sequences, using printf and scanf, performing all needed conversions by the conversion specifications in the format string, adding it to the stream
            i. Formatted functions may still be buffered, such that they require a newline to add it to the stream
            ii. The parameter for each conversion specifier in printf treats that portion of the stream as the type specified, not checking if it is properly that type
            iii. Scanf uses whitespace around designated regions of the inputted string cooresponding to the format string, indicating the end of the prior segment if not picked up by the format variable, used instead of an ending newline in the scanf

        A. On the other hand, it does check for the correct types, not assigning if mismatched, leaving the remainder in the stream

    iv. Since parameters are pushed right to left, the pointer to the format string is easily accessible within the stack, with the offset within the function run-time stack designating the number of variables, with the final at offset 0

        A. The format string is stored with other stirngs in the literal/constant memory section

    v. Printf used before scanf does not require a newline variable at the end to print it, automatically printing when the buffer is released before scanf

  (d) String I/O is performed similarly to formatted I/O, except either reading to or from strings, rather than from stdin/stdout, using "sprintf(*strPointer, formattedString, variables*)", similar for sscanf, using strPointer as the input/output string

3. I/O streams work with files, with stdin/stdout as special system files mapping to the keyboard and monitor, using more general functions to work with other files

  (a) fprintf and fscanf are used for file I/O, acting similarly to other I/O, but allowing the stream to be specified, used similarly but with an added first parameter for the file pointer

  (b) Thus, a file pointer must be declared first, with the type "FILE*", contained within ¡stdio.h¿, opening it by "*filePointer = fopen("filename", "operations")*"

    i. The file operations are r for reading, w for overwriting, a for appending, r+ for reading and appending

    ii. If the fopen fails, it will return a NULL file pointer

    iii. File pointers inherently point to a location in the file based on previous functions and if the operation used ("r"/"w" at start, "a" at the end), moving as a stream does

        A. This is able to be moved by "rewind(*filePointer*)" to the start of the file, and "fseek(*filePointer, offset, origin*)" to move to a particular location

        B. The origin is either SEEK_SET for the start, SEEK_CUR for the current position, or SEEK_END for the end of the file

    iv. "feof(*filePointer*)" returns 1 if the pointer is at the end of the file, and "ftell(*filePointer*)" returns the location in the file of the pointer

  (c) For scanf, the end of file is denoted by the EOF character macro, and is limited to reading 10000 characters at a time

  (d) After using the file, "fclose(*file-pointer*)" must be used to close and save the file

  (e) In addition, "int fgetc(*file-pointer*)", "fputc(*char, file-pointer*)", and "'int fgets(*str, char_num, file-pointer*)", "fputs(*str, file-pointer*)" read (get) and write (put) into files respectively for characters and strings

4. Binary files are those written in binary text, unable to be read by a computer except if converted into ASCII format

  (a) Binary files are read and written by "fread(*pointer, elementSize, elementNumber, filePointer*)" and "fread(*pointer, elementSize, elementNumber, filePointer*)", where pointer is the pointer to the array of data written, with elementNumber entries of elementSize

  (b) Binary files are written bite by bite in the order they are stored in memory, either big-endian (largest magnitude bite first) or little-endian (smallest magnitude bite first)

5. Command line piping sends stdin and stdout into files, such that scanf and printf work on the files piped into the program

19 Chapter 19 - Data Structures

1. Data structures to allow describing of real world objects is called object orientation, using multiple separate values within, held in different manners
   (a) Arrays are used for a fixed quantity ordered storage of the same data type, while structures are used for aggregates of fundamental types
   (b) Structs, like primitive data types, are passed by value, unlike arrays and pointers
2. Arrays of structures and pointers to structures can be created as for any other variable
3. Memory objects/variables in C are able to be allocated to the run-time stack, global data section, or the dynamic memory section/heap
   (a) The heap takes up a portion of memory below global data but above the runtime stack, growing downward toward the stack
   (b) This is managed by the memory allocator, with the program able to request a portion of the heap as needed during runtime, to allow unknown sizes of data structures without wasting space
      i. This contrasts global or stack memory, which is allocated at compile-time, rather than run-time
      ii. Memory allocated remains so until explicitly deallocated, done by the computer automatically on shutdown or within the code
      iii. Not freeing memory is called a memory leak, due to C not automatically doing garbage collection/freeing memory, like in Java
         A. Garbage is memory that is not connected to anything, unable to be accessed, yet not freed
   (c) Due to being contiguous memory, both array and pointer notation are able to be used, such that malloc can be thought to create a dynamically sized array
4. Linked lists store data as a sequential list of elements, similar to an array, but not sequentially adjacent, rather storing as a collection of data units/nodes, each containing a pointer to the next node
   (a) The head pointer points to the start of the list, each pointing to the next, with the final pointing to a NULL pointer
   (b) As a result, they cannot be accessed in a random order like an array, but rather must be viewed sequentially, but are more dynamic than an array
      i. Items are able to be added or removed at any point in the list, rather than just from the end like a dynamically sized array
   (c) Dummy head nodes are used often to eliminate the case of an empty list, serving as a placeholder, rather than a proper data node
   (d) Singly linked lists allow traversing in one direction, while doubly linked lists allow in both directions, adding an extra pointer for the previous node