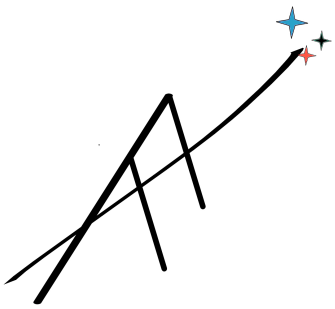
CPE390 Computer Architecture



Ad Astra Education

Dov Kruger

July 13, 2021

Author

Dov Kruger 

Acknowledgements

I would like to thank the giants who taught me. Danielle Nyman who modelled teaching excellence and taught me many things herself, Simcha Kruger who taught me how to research and provided an amazing personal example, Henry Mullish my first coauthor, Roger Pinkham a towering intellect and storehouse of mathematical knowledge, Stephen Bloom who introduced me to automata and rescued my Masters thesis, Klaus Sutner who aside from teaching me computer science gave a memorable lesson on the pronunciation of LaTeX, Alan Blumberg the messiest yet best programmer I know who taught me computational fluid dynamics and demonstrated how to debug anything, Yu-Dong Yao for giving me my first position at Stevens, and who believed in me sufficiently to give me the data structures course, Michael Bruno for an outstanding example of teaching, as well as teaching the dynamics of waves that so terrified me as an undergraduate and approving my position, and Min Song who supports my research and scholarship today. Thanks also to Moshe Kruger for creating the new Ad Astra Logo and for enriching my life in many ways. And thanks to Ellen who took care of my in this trying time when I had so many physical challenges and helped me get this done.

I would also like to thank some top students, TAs and graders who have contributed valuable corrections and insights over the years including Peter Ho, Pridhvi Myneni, E.J. Hannah and David Staronka.

Contributors

E.J. Hannah
David Staronka
Peter Ho

Contents

1. Course Introduction

Welcome to CPE-390!

This course will give you a background in how microprocessors work, with an emphasis on using that knowledge to write faster code. Because we are dealing with COVID, the labs are exclusively software and use a Raspberry Pi that you can buy and keep not only for this course but others including E-322, D4, CPE-487 (Digital System Design) and D6. A Pi will also enable you to pursue your own personal projects, which make for exceptional interview talking points and resume fillers. The first thing you need to do is read the next section on preparing to take this course and follow the links to see what you need to do to prepare.

The goals of this course are as follows:

- Learn how a computer works on a system level (not individual gate logic).
- Learn the components in a computer system and how they work together.
- Become reasonably proficient at writing ARM assembler code (on the Raspberry Pi).
- Learn to benchmark code.
- Identify sources of inefficiency in programs so that you can if necessary optimize it.
- Learn enough Unix commands to get around on a Raspberry Pi.
- Use the gcc toolchain, gdb, and other open source development tools.
- Debug and reverse-engineer code using an assembler debugger like gdb.
- Pair program using vscode.
- Build programs collaboratively using git, and in general work on shared documents using version control.
- Use a graphical debugger where possible to improve your debugging speed.
- Write code to optimize I/O performance to block-oriented devices like hard drives.
- Learn hardware standards such as IEEE-754 floating point, USB, PCIe, etc.

- Use primitive text editors like vi, or weird and esoteric editors like emacs when on a small embedded system supporting nothing else.

Obviously this is a lot. So you won't be tested on using tools. There will be homeworks where you will have to demonstrate knowledge of these tools.

In the beginning, with many of you not having received your Raspberry Pis, we will do homework and labs reviewing C++ and analyzing performance on your laptops. Accordingly, you will not only be using the gcc toolchain on the Pi, you will be asked to install it on your laptop as well. If you use Windows, there are two ways to do this: msys2 or Windows Subsystem for Linux (WSL). In the Mac, you would use clang which is basically equivalent to gcc, and on Linux you may use either, preferably both.

In one lab you will also benchmark performance comparing the Raspberry Pi against your PC.

In order to make this class as interactive as possible, I have also asked you to load Microsoft vscode, the open source editor that runs on Windows, Mac OSX and Linux (NOT Visual Studio, which is proprietary, Microsoft, and non-standard crap). This will allow you in class to edit code on my screen, so that I may virtually call students up to the board. Using vscode, I have been able to make class more interactive online that it was in person. However, I need your cooperation. If you choose not to interact you will learn less, but I cannot force you. If you are uncomfortable with being called in class, you may tell me. I will not force anyone to do it, nor will I penalize you gradewise, but I warn you, you will be penalizing yourself as you will learn much better if you actively try to participate in this way.

Once most of you have your Raspberry Pis, everyone is expected to install the software and have the experience of getting Linux up and running. Of course with enough students, there will always be problems. I have a limited ability to host some students via ssh logging into the Rasperry Pis in my house. If you cannot afford a Pi, or if you have a hardware failure you may meet with me about this. If there are any of you willing to help, I could use students willing to let others log into their Pi, in case of trouble. Also, if you are a strong programmer and willing to help others, we can really use more help. I always give massive extra credit for anyone willing to help others who are struggling, so in addition to being a nice thing to do, consider it insurance if you mess up on a test.

This course have been in constant development for 3 years. If you have any ideas on how to improve it after seeing how it works this semester, I invite your input, and even better, if you are willing to help develop new features. For anyone with work study money, I can pay for this.

1.1 Course Overview

As computer architecture is the study of how computers work, we will cover key the components of computer operation. By the end of the course, you will be able to design

computers and optimize them to perform computations faster. Below is a more detailed list of the material we will be looking at over the next 14 weeks. There is a lot to cover, and therefore some of the hardware topics we will ask you to learn on your own – read a short description and then take a quiz.

- Computer components and terminology
- Essential base notations and appropriate conversions and arithmetic
- C++ fundamentals
- How machines implement high-level languages
- Data types
- Integer and floating point arithmetic
- Circuits and implementation details
- Logic, gates and circuit symbols
- A/D and D/A conversion
- ARM Assembler code
  - Calling subroutines and parameter passing
  - Statements
  - Arrays
  - Passing by reference using pointers
  - Floating point instructions
- Memory Systems
- Compiler Optimizations
- Interrupts and Software Traps
- GPUs

1.2 Schedule of Lessons

In order to learn the above,

1.3 Prerequisites

Though there are no course requirements that will prevent you from registering for CPE-390, I am expecting you to come in with an understanding of loops and function calls in C++ from either E-115 or an appropriate course substitute. We will be hitting the ground running with C++ programming specifically. There will be a short review session at the start of the course, but it will not be sufficient if you are starting from nothing. Be prepared to spend some time familiarizing yourself with C++ syntax if it has been a while since you have last programmed.

1.4 Required Materials

To take this course, you will need to purchase your own Raspberry Pi. In the event of financial hardship, I ask that you contact me or your lab teaching assistant. If we have sufficient supply, we may be able to loan you one. [Note: Not in the time of COVID, but you may log in remotely to mine or a fellow student's]

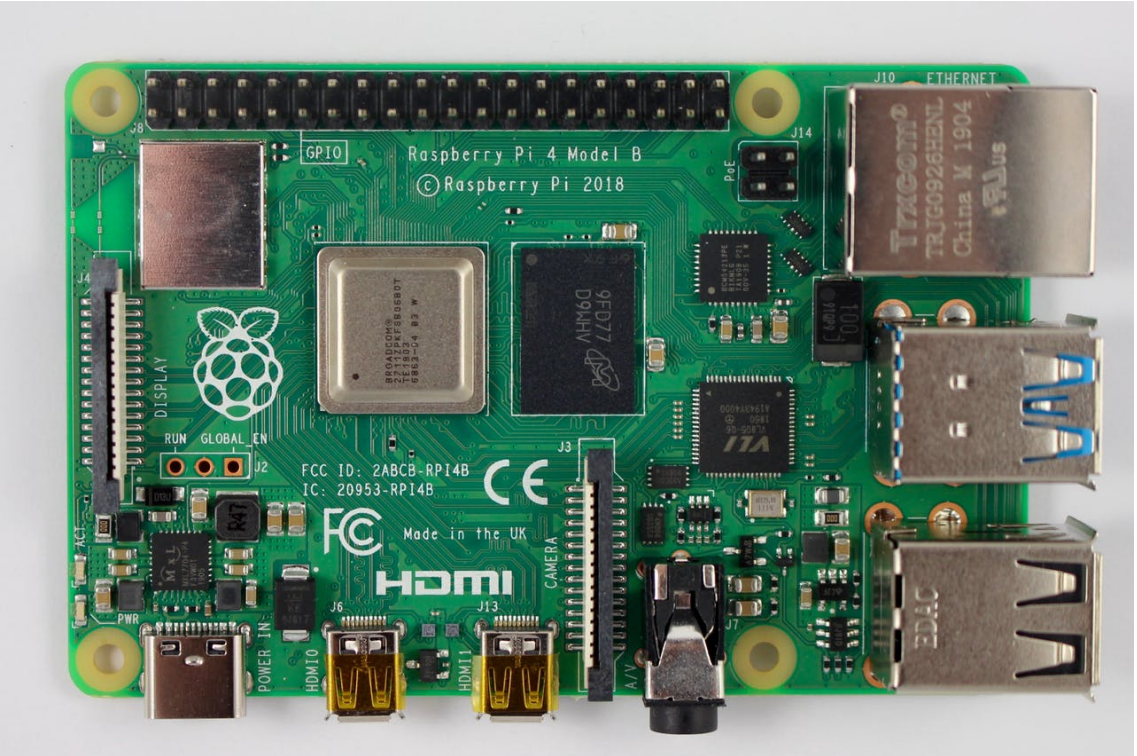
Buying a Raspberry Pi means buying: The Pi itself, a power supply, a memory card to put the operating system on. You also need the following: a memory card reader that you can write the memory card with, a micro-HDMI cable to send video to an HDMI screen, an HDMI screen like a TV, a keyboard and mouse. Once you boot up, you don't need any of the above. You can usually use a wireless connection from your laptop to connect to the Pi, but getting started is hard. Without COVID, any problems are easy. You just come in, get help in the lab, then go home with your machine working. Given that we are all isolated, this is much more challenging now. Most students are able to do this. If you cannot after trying with your TA and any friends, then we will try alternatives.

If you already own a Raspberry Pi, that is ok. If you have an old one it might be a bit slow, but you can do all the work we do in the course.

There are Pis for shared use in the lab, but everyone is expected to load an operating system and configure their own Pi as one of the first labs. As explained earlier, it can be very advantageous for you to have a Pi of your own.

The current model recommended for this class is the Raspberry Pi 4. The chart below explains some of the differences between the two models. If you have an old Raspberry Pi, you may use it, but you must load Raspbian and be capable of compiling c++ code on the machine. The Raspberry Pi4 runs hot, you should definitely either buy or acquire/convert a heatsink to cool the primary chip.

	Pi 3B+	Pi 4
Price	$35	$35 (1 GB RAM) $55 (4GB RAM) $75 (8GB RAM)
CPU	Broadcom BCM2837B0 Quad core Cortex-A53	Broadcom BCM2711 Quad core Cortex-A72
Clock Speed	1.4 GHz	1.5 GHz
GPU	VideoCore IV (250-400MHz)	VideoCore VI (500Mhz)
RAM	1 GB	1, 2, 4, 8Gb
RAM Type	LPDDR2 SDRAM	LPDDR4-3200 SDRAM
Bluetooth	4.0	5.0
USB	4x USB-A 2.0	2x USB-a 2.0, 2x USB-A 3.0, 1x USB-C
Display	1 HDMI	2 microHDMI



I strongly encourage you to order your Pi as early as possible, as there can be delays in purchasing some parts and we have much to cover in this class.

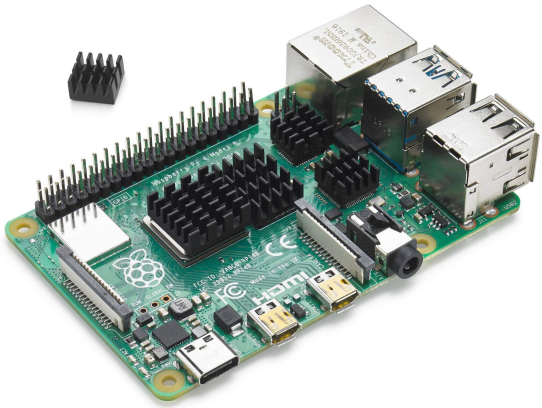
Here are some links that I recommend purchasing your Pi from. They may not be the cheapest by the time you read this, so I encourage you to shop around and look for better deals if at all possible.
Raspberry Pi 3B+:
https://www.pishop.us/product/raspberry-pi-3-model-b-armv8-with-1g-ram/
Raspberry Pi 4:
4Gb Model: https://www.pishop.us/product/raspberry-pi-4-model-b-4gb/
2Gb Model: https://www.pishop.us/product/raspberry-pi-4-model-b-2gb/
1Gb Model: https://www.pishop.us/product/raspberry-pi-4-model-b-1gb/

An interesting point to note is that the 3B+ Pi is phasing out, which means it is becoming MORE EXPENSIVE than the Pi 4.

You will also need a number of peripherals for your Pi to function. If you plan to purchase from one of the links above, then you will need to select all accessories except for the beginner's guide. If you plan to shop around and look for cheaper alternatives, the comprehensive list of peripherals you will need can be found below:

- A microSD card of at least 16 Gb
  - If purchased from a different site, you will need a microSD to SD adapter and some way to write onto an SD card
- An aluminum heat sink, specific to your 3B+/4 model
- A protective case, specific to your 3B+/4 model
- A power supply (MicroUSB for 3B+, USB-C for 4
- A display cable (HDMI for 3B+, microHDMI for 4

Note that the adapter will not do you any good if you do not have a card reader on your laptop. Most Windows-based PCs do have SD card readers, Macs definitely do not. If you don't have one, either get a preformatted card, or find a friend who has one you can use just to get started.



For anyone who wants to install full-blown Linux (64-bit Ubuntu for example) it is probably better to have a 4Gb machine for that, but if you have one feel free to go for it. In 64-bit mode, the raspberry Pi is considerably faster than 32-bit mode. When we use the Pi running Raspbian, we are effectively giving it a lobotomy, a good chunk of its brain is cut out. The down side is that you would have to learn instructions different than the ones I focus on in class, but I will support anyone who wishes to do this.

If you buy an SD card on your own, it will not come with Raspbian installed. In that case, you will need to download it from https://projects.raspberrypi.org/en/projects/noobs-install and move it onto your MicroSD card. You will need a laptop or some other computer/adapter that can write onto SD cards. Since this is a microSD card, you will likely need an SD to MicroSD adapter.

You will also need an HDMI Television or Monitor, and a USB keyboard or mouse to get you started. Once the Pi is installed, you will be able to log in directly from your laptop via Ethernet if your laptop has one, via WiFi through your laptop on Windows. It will be extremely hard for us to help you remotely, so this first part you must work through on your own, although we can try to help. As a backup plan, worst case you will need to connect the Pi to the TV for the duration of the course, which means you will need a USB keyboard, mouse, and a video cable.

If you purchase the Pi 4 and need a micro-HDMI to HDMI cable, or an appropriate converter, I recommend the following:
https://www.banggood.com/1_5m-150cm-Standard-HDMI-A-Cable-Mini-HDMI-C-Micro-HDMI-D-Adapter-Connector-p-1130254.html?rmmds=search&cur_warehouse=CN

1.5 Optional Materials

Although the course is not currently doing hardware labs while we are operating remotely, if you wish to get some equipment I can recommend great sources. The best deal on a DMM is $7.50 from Yourduino.com. These are inexpensive, but accurate – not junk.
http://www.yourduino.com/sunshop/index.php?l=product_detail&p=150

A Geekreit Arduino kit from banggood.com at the time of writing was $15.99:
https://www.banggood.com/Geekcreit-UNOR3-Basic-Starter-Kits-No-Battery-Version-Geekcreit-for-Arduino-products-that-work-with-official-Arduino-boards-p-1133595.html?rmmds=search&cur_warehouse=CN

1.6 Setting Up Software

To set up software on your laptop, please see our class repository:
`https://github.com/StevensDeptECE/CPE390/tree/master/Course%20Documentation`

The above URL has instructions on installing the gcc toolchain, gdb etc under msys2 on windows. On MacOSX you are on your own but since Mac is a Unix, it has a standard linux shell and clang and lldb which are installed as part of XCode which are very similar (in fact clang is almost compatible with gcc). If you have installed linux on your own you should be able to find out how to install all the desired tools, but of course the

TAs and I will be delighted to help you if you have a problem.

1.7 Further Reading

1. https://www.tomshardware.com/reviews/raspberry-pi-4-b,6193.html
2. https://en.wikipedia.org/wiki/Raspberry_Pi
3. https://www.raspberrypi.org/
4. https://developer.arm.com/docs/100095/latest/functional-description/about-the-cortex-a72-processor-functions
5. https://static.docs.arm.com/100095/0003/cortex_a72_mpcore_trm_100095_0003_05_en.pdf?_ga=2
1026908053.1582826689
6. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.100095_0001_02_en/way138151
7. https://medium.com/@ghalfacree/benchmarking-the-raspberry-pi-4-73e5afbcd54b

2. What is Computer Architecture?

2.1 Overview

2.2 Basic Terminology

Term	Definition
address bus	a bus used by the computer to tell memory or a peripheral what address is desired
bit	A single binary digit, either 0 or 1. This is stored on a digital computer as a low or high voltage. As computers have gotten faster and transistors have gotten smaller, the voltage of high has dropped because dropping the voltage is one way to reduce the resistive losses.
byte	8 bits. Invented by IBM. Since a hexadecimal digit represents 4 bits, a byte is two hex digits
word width	The size of the information that a computer can process in one operation. On a 32 bit computer, the ALU and registers are typically 32 bits. Often data paths to memory will be the same size.
bus	A collection of wires in parallel, shared by 2 or more parties to communicate.
core	A CPU, one of multiple on a single chip.
data bus	a bus used to send data, bidirectionally (the computer can read or write)
clock	The master timer used to synchronize operations in the computer
clock cycle	The period of the computer clock. For a 1GHz clock, the clock cycle is 10^{-9} seconds = 1ns.
asynchronous	Circuitry that runs without synchronization (without a clock)
synchronous	Circuitry in which a master clock is used to control the timing and operation of all components. Synchronous circuits are faster because they avoid the latency of waiting for an acknowledge.

block-oriented device	A device, usually mass storage that is designed to transfer data not one byte at a time, but a block at a time. This changes how software interacts with the device. For example, it is inefficient to write to a hard drive one byte at a time since the entire block must be written regardless of how few bytes are changed.
buffer	An area of memory used to marshal data before being sent to a peripheral at high speed.
cache	Fast memory used to keep recently-used data from main memory so it can be accessed again faster.
caching	General technique for storing data in faster memory so the average effective speed is higher.
chip	An integrated circuit in which many transistors are manufactured, essentially with a single unified error probability. Chip manufacture was revolutionary because it has brought down the probability of failure of any single component by a factor of billions. See MTBF, tubes.
CISC	Complicated Instruction Set Computer. An architecture in which many kinds of instructions, of different types and lengths, make decoding slower, but some instructions may offer performance advantages. Example: 80x86
RISC	Reduced Instruction Set Computer. Simple, uniform instructions that are easier to decode, therefore can be executed faster. Emphasis is on providing only instructions that speed up execution by simulating realistic situations and measuring time taken. Example: ARM
compiler	A program that reads in a high level program and generates machine language to execute the code on a particular computer. High level languages and compilers allow programmers to write portable programs, while assembler code is not portable and specific to a particular kind of computer.
CPU	The Central Processing Unit, the core of a computer that does the computing.
core	Another name for CPU, when there are multiple CPUs in a single chip. A 4-core chip has 4 separate CPUs inside, each typically with its own cache memory, some shared cache memory, and shared access to RAM.
floating point	An approximation for real numbers that uses a binary representation of a fraction (the mantissa) with an exponent to shift the binary point, and a sign bit. See: IEEE754
frame	TODO
frame buffer	TODO
handshake	TODO

hard drive	A storage device using spinning platters coated with magnetic media, on which data can be recorded. Hard drives are non-volatile, meaning they can retain their memory without power. Hard drives are mechanical and delicate, therefore vulnerable to damage from being dropped or other mechanical shock. They also require significant power to spin the platters at high speeds.
IEEE754	A standard for floating point computation including precise definition of how computation occurs for ordinary numbers and special values for uncountably infinite values (INF) and indeterminate (Not a Number, NaN).
instruction	A single set of bits to command a CPU to execute computation. The fundamental unit of machine language code.
integrated circuit	A single block of material on which tiny circuitry is engraved, usually using photography to etch a complicated pattern into photoresist, followed by acid to etch away the material beneath, then a different solution to remove the photoresist layer. Chips may require 30-40 steps to create multiple layers of circuitry.
kernel	The core of an operating system, since this is what runs the computer the kernel is typically run in privileged mode.
latency	The lag between a signal and the result, often due to propagation time which is limited to the speed of light.
memory	Memory typically means RAM, the main memory that is far faster than hard drives, but volatile.
memory manager	TODO
micro-processor	A CPU implemented in a chip.
micro-controller	A chip that contains a CPU and also has functions such as A/D conversion that allow it to interact with the physical worlds and control it. See SOC
MIMD	Multiple Instructions, Multiple Data. A system of parallel computation in which multiple computers run in parallel and communicate with each other.
noise	Unwanted signals impressed on circuits from the environment. Noise comes from the sun and cosmos, from radio and other electromagnetic interference.
operating system	The program that controls a computer and decides which operations are allowed.

parallel processing	A method of increasing speed of computers by running multiple instructions on multiple computers at the same time to perform work simultaneously.
pipeline	A technique to make computers faster by overlapping computation. Typical pipelining includes loading instructions, decoding instructions, executing instructions, loading data, computation.
RAM	Random Access Memory. The computer can send an address and request to read or write to/from RAM. RAM is typically volatile, meaning if power is lost all memory is gone. It is also significantly faster to access current generations of RAM sequentially, so while it may be accessed in random order, it is definitely slower to do so on current hardware.
register	The fastest memory in a computer. There are only a few registers (for example 16 on ARMv7, and 32 on ARMv8). Code which uses registers is the fastest. See cache and RAM. Code using only registers can run fast in parallel since the core is not using RAM which is shared by all cores.
register file	A group of registers that can be quickly brought in and out quickly so that the processor can call a function, or switch context quickly.
sector	A section of a track on a hard drive.
semiconductor	A material useful for creating devices such as diodes, transistors, and sensors. Semiconductors do not conduct well, nor do they insulate well, but tiny changes in light and heat can dramatically change their conductivity, and in the case of Field Effect Transistors (FETs), a tiny change in the gate voltage can dramatically change the resistance from source to drain.
signal	The desired part of a voltage in a circuit (see noise).
SIMD	Single Instruction, Multiple Data. A method of computing faster by executing many identical operations in parallel. Also called vectorized instructions.
SNR	Signal/Noise Ratio.Digital signals are highly resistant to noise, but as voltage keeps dropping for power reduction, the ratio of noise to the signal keeps rising. This requires ever-better shielding and noise reduction (avoiding long signal lines which can act as antennae.
SOC	System on a Chip. A single chip that contains the CPU, memory, I/O devices. Examples of SOCs include the chip running the Arduino and Raspberry Pi.
SSD	Sold State Drive. A block-oriented, non-volatile mass storage device that has no moving parts, uses less power, and is faster than a hard drive, built of NAND-flash. Downside, there is a limit to the number of times each block can be written before it degrades.

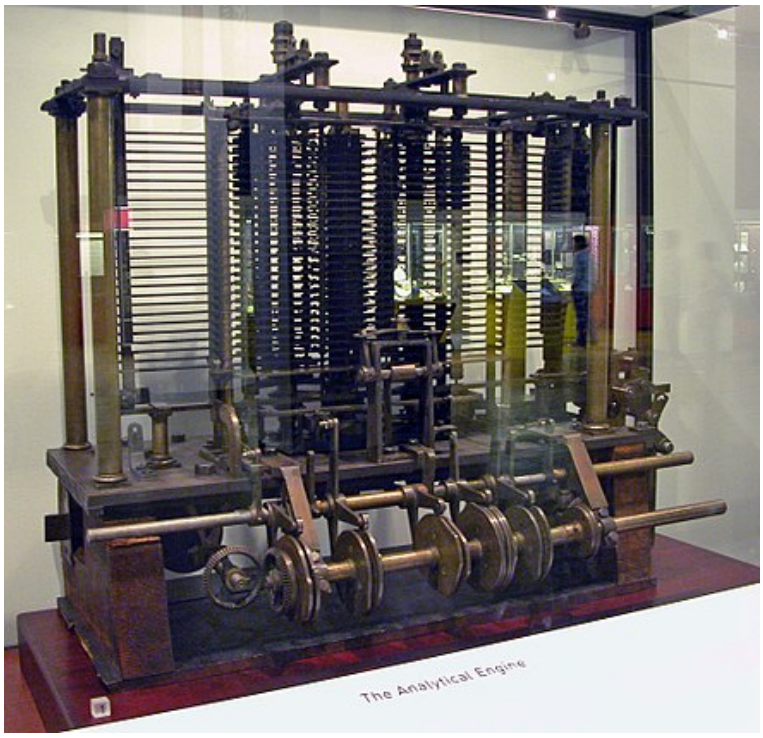
stack	A method of storing data for function calls. Stacks support push (adding values to the stack and subtracting from the stack pointer (SP) and pop (pulling values from the stack and adding to the SP).
superscalar	A computation architecture in which more than one instruction can be executed per clock cycle.
switching speed	The speed with which an individual transistor flips from low to high or high to low. Higher voltage increases the speed. Smaller transistors increase the speed because it takes time for electrons to travel. Grace Hopper famously said 1 foot = 1ns. It takes light 1 billionth of a second to travel 1 foot, so to make computation faster, shrink the distances. See: ballistic electrons.
system time	The amount of time spent inside the operating system kernel as opposed to inside the user's program. For example, once a program calls read or write kernel calls, the program must wait for the operation to complete.
track	A concentric circle of data on a hard drive that is broken into sectors.
vector processing	An attempt to make slow operations faster by designing circuitry that can pipeline and executing many of them overlapping. See SIMD.
VLIW	Very Long Instruction Word. A method of increasing the speed of CPU execution by reading many bits at once and considering executing multiple instructions simultaneously, potentially out of order if that does not affect results.
von Neumann architecture	A computer architecture where there is a single memory, and instructions are loaded from main memory, and a data is read and written from main memory as well. Because main memory is used for both purposes, computers are slower. This can be solved with cache since most code has loops, and most of the time instructions being executed have been executed before.
Harvard architecture	A computer architecture where there is one memory for instructions, and another for data. This is faster, but more complicated, and the instruction memory is often going to exceed what is needed in order to handle larger programs. As a result, this architecture is only used for special-purpose high-performance architectures.
wafer manufactured and cut apart to be put into individual chips.	A block of silicon or some other high-purity material from which hundreds of chips are

word	A single vector of bits that the processor can manipulate in a unit time. On a 32-bit computer words are 32 bits, and on a 64-bit computer, they are 64 bits.
XPoint	A new memory technology composed of a new kind of device (memristor), where the resistance can be changed from high to low and back. Currently faster than NAND-flash, and theoretically eventually as fast as RAM. The potential of XPoint would be to eliminate hard drives and have only a single memory that is non-volatile. This would be a radical change since currently, every time a machine boots up it must load software into RAM. With XPoint, loading can be eliminated since permanent storage could be as fast as RAM, and the memory is random- accessed by word, so there is no need to read an entire block.

2.3 History

The origin of computer architecture is often noted with Charles Babbage's work towards a mechanical computer in the 1800s. Though he never completed his work due to extending his concepts rather than finishing a simple machine, and then running out of funding, Babbage's steam-powered machines still showed promise and paved the way for machines to perform calculations. Notably, their architecture resembled that of a basic computer. Data and memory were separated, there were instruction-based operations, and his machines featured an I/O unit.

Babbage worked closely with Ada Lovelace Byron who was both a friend and math prodigy. In the 1840s, Babbage presented a lecture on his latest creation, the Analytical Engine. Lovelace was in attendance and helped translate his lecture into English. She also provided her own additional notes on the Analytical Engine, describing how it computed Bernoulli numbers. This is historically known as the first written algorithm, therefore making her the first programmer.



Babbage went on to design two new types of mechanical calculators, which were so large in scale that they required a steam engine to operate. The first was an automatic mechanical calculator (based upon his Difference Engine), which could automatically compute and print mathematical tables. The second was a programmable mechanical calculator (based upon his Analytical Engine). Though neither of these were successfully constructed in his lifetime, it inspired many future engineers to improve upon his designs. In 1937, Howard Aiken worked with IBM to design an Automatic Sequence Controlled Calculator (ASCC), or a general-purpose electromechanical computer, to use in World War II. This ASCC was based heavily upon the design for Babbage's Analytical Engine.

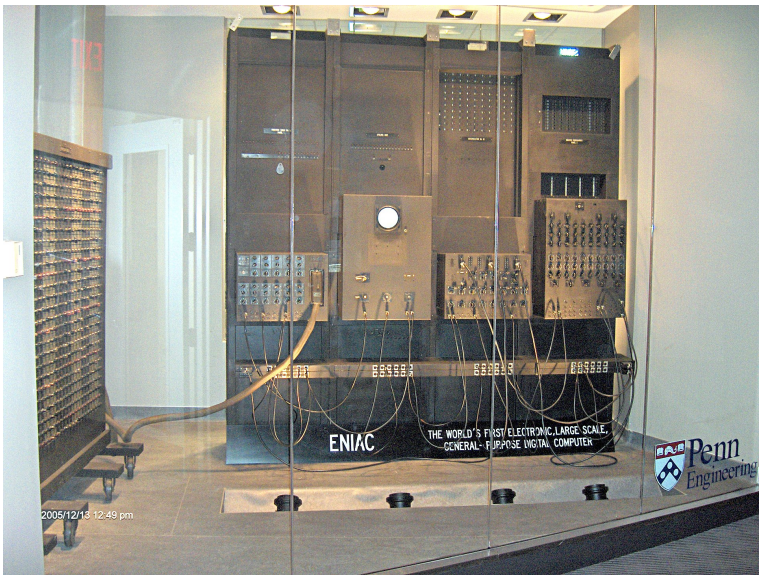
The Electronic Numerical Integrator and Computer (ENIAC) was the first programmable, electronic, general-purpose digital computer, completed in 1945. It was a modular computer, with many panels to perform different functions, such as addition, subtraction, and storing ten-digit decimal numbers in memory. It even featured high speed operation by having the panels send and receive numbers, perform computations, and trigger future operations all without moving any parts.

The ENIAC was based heavily upon vacuum tubes, requiring a total of 18,000 by the time it was retired in 1956. These vacuum tubes controlled electric current flow in a high vacuum between electrodes. The ENIAC ended up drawing so much power (150 kW) that the lights in Philadelphia were rumoured to dim when it was turned on.

The ENIAC's massive scale brought failure rate into consideration as a factor of computer architecture. With so many components, the probability that any one individual part fails becomes astronomical. Several tubes ended up burning out every day, leaving the ENIAC nonfunctional about half the time. In general, early computers faced a Mean Time Between Failure (MTBF) of just 30 minutes.

The ENIAC itself had engineers on staff at all time to find and replace broken tubes. The longest continuous operation ENIAC experienced was 116 hours, or almost 5 days, throughout its 11 years of operation.

For a better understanding of the ENIACs scale, below is an image including four of the ENIAC's panels. These ones specifically controlled the interface for the function table and the memory storage of 10-digit numbers.



The invention of the transistor had a huge impact on the speed and reliability of computers. In 1947 the first transistor was invented by John Bardeen and Walter Brattain under William Shockley at Bell Labs. In 1959 the first MOSFET was built by Mohamed Atalla and Dawon Kahng at Bell Labs. Transistors did not break as often as vacuum tubes and they were smaller and used less power.

Credit is often given to Robert Noyce from Fairchild Semiconductor and Jack Kilby of Texas Instruments for the first integrated circuit (IC), demonstrated on September 12th, 1958. By putting multiple transistors into a single package, this effectively reduced the probability of failure to a single entity, again dramatically increasing reliability, shrinking power and shrinking the size of computers. Their designs ended up being widely used by NASA from 1961 to 1965, just a few years before the first moon landing!

Just a few years after the first ICs was when Gordon Moore made the observation widely known today as Moore's Law, which we will talk about later in this chapter.

Intel released the first microprocessor, the Intel 4004, in November of 1971. Comprising of only 2,300 transistors and 640 bytes of memory, the 4004 was capable of 60,000 operations per second (OPS).

The next massive innovation was the Intel Pentium of March 22nd, 1993. This processor had a clock rate of 60 MHz and incorporated 3.1 million transistors, which was astronomical for the time.

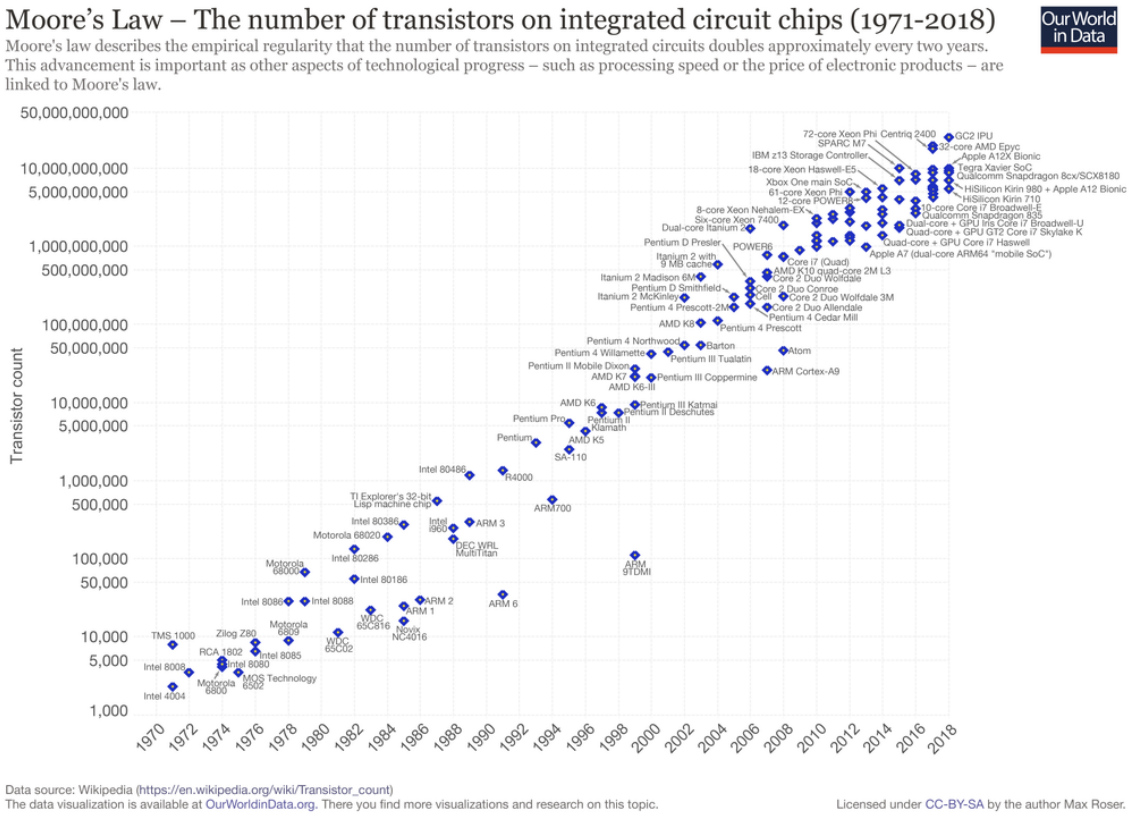
The most recent major advancement in computer architecture was the implementation of multi-core processors commercially. Rather than trying to focus on improving the OPS rate of processors with only one route to execute instructions, these new processors from AMD and Intel around 2004-2005 were able to execute multiple instructions at once in parallel.

2.4 Timeline of achievements

Invention	Year	More Information
Analytical Engine	1837	https://en.wikipedia.org/wiki/Analytical_Engine
Harvard Mark I	1944	https://en.wikipedia.org/wiki/Harvard_Mark_I
ENIAC	1946	https://en.wikipedia.org/wiki/ENIAC
UNIVAC 1	1951	https://en.wikipedia.org/wiki/UNIVAC_I
IBM 701	1952	https://en.wikipedia.org/wiki/IBM_701
CDC-6600	1964	https://en.wikipedia.org/wiki/CDC_6600
Intel 4004	1971	https://en.wikipedia.org/wiki/Intel_4004
Intel 8008	1972	https://en.wikipedia.org/wiki/Intel_8008
MOS 6502	1975	https://en.wikipedia.org/wiki/MOS_Technology_6502
Cray-1	1975	https://en.wikipedia.org/wiki/Cray-1
Motorola 68000	1979	https://en.wikipedia.org/wiki/Motorola_68000
Intel 80286	1982	https://en.wikipedia.org/wiki/Intel_80286
Motorola 68020	1984	https://en.wikipedia.org/wiki/Motorola_68020
Cray-2	1985	https://en.wikipedia.org/wiki/Cray-2
Intel 80386	1985	https://en.wikipedia.org/wiki/Intel_80386
Motorola 68030	1987	https://en.wikipedia.org/wiki/Motorola_68030
SPARC	1987	https://en.wikipedia.org/wiki/SPARC
Intel 80486	1989	https://en.wikipedia.org/wiki/Intel_80486
IBM PowerPC	1992	https://en.wikipedia.org/wiki/PowerPC
Intel i7	2008	https://en.wikipedia.org/wiki/List_of_Intel_Core_i7_microprocessors
ARMv8	2011	
AMD Ryzen		https://en.wikipedia.org/wiki/List_of_AMD_Ryzen_microprocessors
Tianhe-2	2013	https://en.wikipedia.org/wiki/Tianhe-2
TaihuLight		https://en.wikipedia.org/wiki/Sunway_TaihuLight
Apple A13 Bionic	2019	https://en.wikipedia.org/wiki/Apple_A13
IBM Summit	2019	https://en.wikipedia.org/wiki/Summit_(supercomputer)

2.5 Moore's Law

In 1965, the CEO and co-founder of Intel Gordon Moore observed trends in microelectronics and noted that every year the number of components (primarily transistors) per integrated circuit would double. This forecast was revisited in 1975, where Moore changed his outlook to doubling every two years. Though not based on empirical evidence, his prediction has become known as Moore's "law".



Initially, computers were increasing in speed every six to twelve months, as both number of circuits and clock speeds were increasing steadily. Today, the main enemy of speed has become heat dissipation which has dramatically reduced improvements in speed of individual circuits. For many years, improvements in performance of CPUs has come from improving the efficiency of the designs, by increasing parallel execution (multiple cores) because the number of circuits has continued to increase, and by increasingly efficient thermal management (shutting areas of the CPU off when not in use). As a result, performance has increased slowly, while the ratio of power/computation has continued to decrease faster. Clock speed has increased slowly as better cooling, better thermal management, and reduced voltage have been used to reduce power.

Moore's law still plays a key role today in long-term planning for companies in the semiconductor industry. It has guided research and developing targets, making it function almost like a self-fulfilling prophecy. Since around 2010 the pace of semiconductor advancement has fallen dramatically below that predicted by Moore's

law. Another difficulty in the last few years has been the dramatically increased cost of the smallest processes. While today, 5nm circuits are being produced, the cost of these circuits is vastly higher. As the size of the circuits has continued to decrease, it becomes necessary to reduce the size of dust that can come into contact with the wafer during manufacture. The smaller the circuits, the tighter the requirements and the more parts will fail after manufacture due to contamination that renders the chip non-functional. Thus even though 7nm and 5nm processes exist, most CPUs today are using 10nm to 14nm processes because the newer ones have yields that are too low for practicality and reasonable cost.

Transistors and their Nomenclature

Moore's Law, though applicable to many electronic components, is most often used when referring to transistors. Transistors are semiconductor devices used to switch and amplify electronic signals. When used to amplify signals, a small signal application on between a pair of terminals can control a much larger signal at a different pair of terminals. This principle, called *gain* is one you explore further in Design IV (E232) with Professor Ryan. When used as a switch, transistors can be used as an electronically-controlled toggle for the the on/off state of current in a circuit.

There are two types of transistors. Bipolar Junction Transistors (BJT) use a small current at the base to control a larger current between the emitter and collector. Field Effect Transistors (FETs) use a voltage applied to the gate to control the flow of current from source to drain.

The first successful transistor is often cited as the work of John Bardeen and Walter Brattain at AT&T's Bell Labs in New Jersey. Near the end of 1947, they observed the contact between two gold points and a germanium crystal amplifies the input power. This approach is that of a point-contact transistor, a transistor type that has become obsolete relative to BJTs and FETs. The image below is of a replica for this first transistor created in Bell Labs.

As you'll notice, this transistor is quite large. Nowadays, Metal-Oxide Semiconductor FET (MOSFET) which you will discuss in depth in Transport Phenomena in Solid State Devices (EE471) are on the scale of 10 nm down to 5nm. Most modern CPUs are built with transistors of this size. Newer processes including 3-nm and 2-nm transistors are planned for 2022 and 2023 respectively.

Progress has slowed dramatically because the smaller the process, the tighter the tolerances must be for precision alignment, dust particles, and material purity. Defects which were tolerable 30 years ago would now be bigger than several transistors. Intel has had trouble keeping up with the latest process, and Taiwan Semiconductor (TSM) is currently the leader in the 5nm process.

This definition does present a few problems, however. For starters, 1 nm is about the width of five silicon atoms. Knowing that, it becomes impossible after a certain point for Moore's Law to continue at the same pace. However, the nature of the name "7-nm transistors" is disconnected from the reality of transistor sizing. 7-nm transistors are considerable larger than 7 nm. Additionally, transistor-scaling progress has been made through more than one dimension/metric. The two most common are the metal half-pitch and the gate length. The metal half-pitch is half the distance from one metal interconnect to the start of the next one. The gate length is the space between the source

and drain terminals on FET transistors. As the half-pitches and gate lengths of transistors scale down, there is an atomic limit in place for how far they can shrink.

Current researchers have begun looking into stacking transistors vertically, which introduces the third dimension, referred to as "tier". Combined, these three metrics comprise the contacted gate pitch (G), metal pitch (M) and number of tiers (T) into a new GMT nomenclature. IEEE's International Roadmap for Devices and Systems, a well-known and respected prediction for the technologies of the near-future, is beginning to push this GMT nomenclature so that transistor representation more accurately reflects the developing technologies of our time.

2.5.1 References

"The Node is Nonsense" by Samuel K. Moore of the IEEE Spectrum.
https://computerhistory.org/blog/who-invented-the-transistor/

2.6 Designing For Speed

The goal of this course is to teach you how computing works so that you can write better programs. At the same time, by learning how computers work, you should be able to appreciate how to improve the design and you can take further courses to learn how to design CPUs.

The most obvious way to increase computer performance would be to increase clock speed but heat makes this extremely difficult. The following table summarizes many of the ways that can be used to improve the overall speed of CPUs. Some of these we will be covering in class. To illustrate the following topics we can state the equivalent performance improvement in a restaurant.

Clock Speed	Obviously, if every circuit in the computer becomes faster, the computer will be faster unless it is waiting for an I/O device in which case the peripherals would also need to be sped up. In a restaurant, this is analogous to hiring superheroes to be chefs.
Word Width	A 32-bit computer that is doing work on 64-bit numbers will have to perform multiple instructions. Doubling the circuitry to 64 bits makes processing those numbers faster. Unfortunately, for most computation we just don't need 128 bit numbers so there are diminishing returns to this approach. In a restaurant, this is analogous to serving more food by increasing portion sizes. That works if there are families willing to eat out of the same bowl, or very hungry customers who want more food but it does not allow serving more customers if they want normal portion sizes.
Latency	It takes time to request a value from memory and wait for it to arrive. This is related to the speed of light and also gate delay (the time it takes gates to react to changes in inputs). Short of making circuits faster we can design circuits to give more information when asked. So instead of reading one memory location at a time, every time the CPU makes a request it can get 4, 8 or 16 values. This is the principle behind burst transfers in DDR RAM, for example.
Pipelining	While doing one thing, try to have the CPU already starting on the next task. A cook can have multiple dishes cooking at the same time if they are all at different stages requiring waiting.
Multiple Execution Units	Since it can take time for a difficult operation like multiplication or division, give the CPU more than one, so while one unit is busy the other can be started This is analogous to giving a cook multiple burners to work on so they can cook more than one dish at once. There is still only one cook (the CPU) but they are much busier. And just like a cook can only manage so many dishes, a CPU cannot handle too many execution units.
Multi-core	If it is impossible to speed up the CPU any further, make multiple CPUs. This will not make an individual instruction execute any faster but if a program can be split into multiple parts (threads) it will get faster. The analogy is in a busy restaurant, hire more cooks, which generally won't speed any one dish but can get more dishes out to more customers.
Specialty	Some operations are slow using general purpose programming. If custom circuitry can make these faster, then specific operations like encryption, video encoding/data compression can be optimized. The restaurant analogy is buying custom equipment like a waffle maker to speed a particular kind of dish.
Power Management	Given that heat is the limiting factor, run one CPU at a higher clock cycle for a while to compute faster, then reduce the clock when the heat gets too high. The restaurant analogy is to make cooks work very hard for 15 minutes in a crush of customers, but realizing it's unsustainable, giving them a break after that.
Vectorization	Do many of the same operations at the same time. In a restaurant, instead of making soup for one, use a giant pot and cook soup for 200. It still takes the same amount of time to get the first dish out, but then we can serve many at high speed.
interleaving	When reading or writing memory, read multiple banks at the same time. In a restaurant, when pouring drinks lay out multiple cups and pour the drink quickly across the row.

The first factor we will look into is **bandwidth**, a measurement of the maximal rate of data transfer across a given path within a system. Bandwidth is commonly measured in terms of bits per second, or relevant multiples of it (bit/s, kbit/s, Mbit/s, Gbit/s, etc.). Bandwidth

clock speed The simplest if it is possible, speed up the circuits word width If data transfers are the limit, or size of word, double the word size pipelining decode, execute, load/store, multiply at the same time (overlapping) parallelism Run more programs at the same time vectorization: SIMD (Single Instruction Multiple Data) interleaving Have many banks of memory and get then all in parallel.

2.6.1 How do Computers get faster?

1. Faster circuits. Increased clock speed. drive circuits harder limiting factor: COOLING! make circuits smaller: limiting factor: our ability to build without defect

2. More parallel activity (increased word width). 4004 4-bit 700khz 8080 8-bit 1MHz 6502 8-bit 1MHz 80286 16-bit int16_t 80386 32-bit float int i7-xxxx 64-bit double where are the 128 bit and 256 bit computers? AVX, AVX-2, AVX-512 instructions (vectorized) 3. Pipelining (overlap actions that do not depend on each other)

3. Numerical Bases

Most human societies use base 10 arithmetic, invented because we have 10 fingers. Computers work with two state logic, high and low, and so it is natural to use base 2 where 0 is low, and 1 is high. For this reason, in computer architecture it is important to know the properties of the base 2 representation used, and how to convert to and from decimals.

Additionally, since a 32-bit computer would require 32 1s and 0s, it is more convenient and compact to use base 16 (hexadecimal) where each digit is 4 bits.

non-integer bases, we will not cover them in this course, as they will virtually never appear in the field.

3.1 Base Conversion

You are already familiar with decimal, as humanity depends heavily on base-10 representation for much of what it does. In decimal, we have ten digits, 0 through 9. These digits can be used to express almost any value, so long as you put them in the appropriate digit (decimal) place.

Binary is a base representation with only two values, 0 and 1. Binary is the only language that computers understand. Though we often program in higher level languages that are abstracted from this machine code, the job of the compiler is to translate what we write in languages such as C++ and Java into values of 0 and 1.

Hexadecimal is base-16, ranging from 0 to F. After reaching 9, hexadecimal follows A to F as you will see in the chart below.

Finally, we will look at octal, which is a base-8 representation.

Binary	Octal	Decimal	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Note a nasty gotcha in C++, C and Java:

```
int x = 33; // the decimal number we all know and love!
int y = 0x33; // this is base 16 because we can see the 0x 3*16 + 3 = 51
int z = 033; // ouch! leading zero means octal and if you do not know that and
    just threw on a leading zero for kicks you are in for a shock! 3*8+3 = 24
    decimal
```

3.2 Further Reading

4. Arithmetic

A large part of what computers do is compute numerical answers. This chapter describes whole-number (integer) calculations both unsigned and signed, and floating point (decimal) calculations.

4.1 Integer Arithmetic

Each bit of memory in a computer is low or high, which are represented as 0 or 1. A 1-bit number can therefore be 0 or 1. Combine two bits, and there are four combinations possible: 00, 01, 10, 11. Combining 8 bits give 2^8 combinations, and in general for n bits, 2^n. What these different values mean is subject to interpretation. In this section we are talking about unsigned and signed integers. The obvious choice where all bits are 0 is the value zero. Using base 2, each bit position represents a power of 2:

128	64	32	16	8	4	2	1

For example, the bits 101 represent $4 + 1 = 5$

The following table shows a few examples of 8-bit unsigned integers

128	64	32	16	8	4	2	1		
0	0	0	1	0	1	0	0	=	20
0	1	0	0	0	1	1	0	=	72
1	0	0	0	0	1	1	1	=	135

Writing out even 8 bits or 256 values is too big, so consider a hypothetical 3-bit computer. The values for unsigned integers are shown in the following table:

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

With only 3 bits, there is no room to represent numbers bigger than 7. Adding 1 to 7

results in the binary 1000, but since there is only room for the bottom 3 bits, the result is zero. For this reason, integer arithmetic can be viewed as a circle, and any computation is always modulo 8.

For signed numbers, the high bit is taken to be the sign. If the high bit is 0, the numbers is positive, if it is 1 then the number is negative. This means that the highest positive number is no longer 7 but 3. The question is how to arrange the negative values. The following table shows the way it is done on all computers. The reason is that with all 1 bits considered -1, this makes the math work just as it does for unsigned integers: $-1 + 1 = 0$

000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

This assignment of values is called twos-complement and is used on all computers today. One question is, how to compute what bits are needed for a particular negative number. Here is the algorithm for computing what bits for a negative 8-bit number -5.

- Determine the bits for 5 = 00000101
- Invert every bit 11111010
- Add 1 11111011

4.2 Floating Point arithmetic

Floating point is a computer approximation to real numbers. It is only an approximation because as usual, the computer has a finite precision, and real numbers are infinitely dense. A number in floating point has bits devoted to three purposes: the sign bit (positive or negative), the exponent controlling where the decimal point (really the binary point) is, and the mantissa which are the bits representing the leading digits of the number.

The IEEE754 standard now defines 5 types of float. We will focus on the oldest two: single (32-bit) and double (64-bit) precision. There is also half precision (16-bit) and quad precision (128-bit).

Single precision floating point has 1 bit for sign, 8 bits for the exponent and 23 bits for the exponent. Because leading zeros do not add information, there is an automatic leading 1 added to the front ofthe mantissa for effectively 24 bit resolution:

seeeeeeeemmmmmmmmmmmmmmmmmmm
In other words, the bits of the mantissa are 1mmmmmmmmmmmmmmmmmmm
The exponent ranges from 2^{-128} to 2^{127} and this means the range of floating point is +/- 1.2e+38. The magnitude can be as small as 1.2e-38.

The 24 bits give approximately seven digits accuracy, so floating point can represent any of the following. Each has a relative accuracy of 10^{-7} or 1 part in 10 million, but notice that the absolute error scales with the number. For a number like 10^{20} the absolute error could be 10^{13} which is in absolute terms, a large error.

1.234567
123.4567
1234567.
1.234567e+20
1.234567e+38
The value 0.0 is where all the bits are zero:

00000000000000000000000000000
Note that this is $1.0 * 2^{-128}$ but the hardware specifically recognizes this bit combination as zero. Note that the negative bit can be set for zero: 10000000000000000000000000000 and that this represents the value -0.0. This supports mathematics such as in calculus, where we approach zero from the left.

Internally, you can think of the exponent of the floating point number determining the position of the binary point.

1101101.10101010101010101010101

The digits to the left represent an integer. The digits to the right represent negative powers of 2 $(1/2)$, $(1/2^2)$, $(1/2^3)$, ...

So:

$.1 = 1/2$
$.01 = 1/4$
$.001 = 1/8$
and the number

$.101 = 5/8$ which is .625

Notice that any whole combination of powers of 2 is an exact number. The number 1.75 in decimal could be written 1.11 in binary, which is $1 + 1/2 + 1/4$.

However, there are numbers that are not powers of 2. For example, the fraction $1/3$ is a number that we are used to thinking of as a "repeating fraction" even in decimal. This is because there is no exact way to represent the fraction $1/3$ in base 10. There is similarly no exact way to represent the fraction in binary. There will always be a small error. In decimal $1/3 \neq 0.3333333$.

We are not used to thinking of $1/10$ as a repeating fraction because it fits well in our decimal system. However, $10 = 2 * 5$, and while 2 is a power of 2, 5 is not, so $1/10$ is also

a repeating fraction in binary.

Double precision floating point has 1 bit for sign, 11 bits for the exponent and 53 effective bits for the exponent including the hardwired leading digit 1. The following shows 53 mantissa bits which is what is stored:

seeeeeeeeeeemmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
The exponent ranges from 2^{-1024} to 2^{1023} and this means the range of floating point is +/- 1.2e+308. The magnitude can be as small as 1.2e-308.

The 53 bits give approximately 15 digits accuracy, so double precision can represent any of the following. Each has a relative accuracy of 10^{-15} .

1.23456789012345
3.14159265358979
123.4567012345
1234567890.12345
1.23456789012345e+20
1.23456789012345e+308
1.23456789012345e-308
-1.23456789012345e+308

4.2.1 Roundoff Error

4.2.2 Special Floating Point Values INF and NAN

Division by zero is impossible, and with integers we have seen division by zero causes a hardware trap that crashes the program unless caught. This is disasterous behavior if we are calculating a course in an airplane autopilot. We need the program to first and foremost not crash!. In calculus, while dividing by zero cannot be done, we can say what happens as the divisor approaches zero; the limit goes to infinity. The IEEE floating point standard supports this kind of operation.

```
double a = 1.0 / 0.0; // positive infinity
double b = -1.0 / 0.0; // negative infinity
double x = a + 1; // still infinity
double y = b * 2; // still infinity
double z = x + z; // NaN
```

The theory behind IEEE math is Kantor's infinities. Infinity is no longer countable. Once we reach infinity, we can no longer distinguish between $infinity + 1$ and $infinity * 2$. The numbers are uncountably large. When two infinitely powerful forces fight it out the result is NaN (Not a Number), as opposed to naan, which is delicious.

4.3 Big-Endian and Little-Endian

Big-endian and Little-endian refer to which "end" of a number is first in storage, either the largest or smallest, respectively. We write and work with numbers Big-endian if working by hand, but if we write a 32-bit (4 byte) number from a register into memory, it is written in little-endian order by default. This is because Intel has been little-endian and dominated the market. It's similar to why most of the world is writing technical papers in English – because the US was so dominant. The ARM does have the ability to change and be big-endian also.

On a little-endian computer, a 4-byte integer 5 (in hex, 0x00000005) if stored at location 2000 would be:

```
2000: 05 00 00 00
```

In other words, the low byte comes first. Similarly, the number 256 = 0x00000100 would be written:

```
2000: 00 01 00 00
```

On a big-endian computer, the high byte would come first, so the order is reversed:

```
2000: 00 00 00 05
```

As a programmer, the order in which bytes are stored within a number is not relevant unless you write out on one computer and read in on another. In that case, if bytes are written out in little-endian, then read in the opposite way, the data will be wrong.

The internet protocol TCP/IP was developed when Sun was dominant in networking and its computers were big-endian. For that reason, IP addresses are stored high-byte first. For example, the address of the Stevens gateway is 155.246.1.1 which would be written:

```
2000: 9B F6 01 01
```

If this were stored on an Intel CPU as an integer, each side would have to convert before sending, and then convert before reading, because that is the convention of the internet.

Another example is binary STL files. Because Intel has been so dominant, many formats just specify that binary data is stored in little-endian format. STL is a solid model standard that stores models to be manipulated by CAD packages and printed on 3d printers. For each triangular facet in a solid model, there are 12 numbers:

```
normalx normaly normalz
x1 y1 z1
x2 y2 z2
x3 y3 z3
16bits
```

Taking a look only at one representative point (1.0,2.0,3.0) the hexadecimal values would be 0x3f800000. 0x40000000 and 0x40400000. In little-endian order this is

```
00 00 80 3f 00 00 00 40 00 00 40 40
```

But for a programmer writing in Java, which was developed by Sun and is stored in big-endian format, this is backwards, and so any Java programmer reading this file has to know to reverse the bytes, like this:

```
RandomAccessFile raf = new RandomAccessFile(filename, "r");
FileChannel inChannel = raf.getChannel();
long fileSize = inChannel.size();
mem = ByteBuffer.allocate((int) fileSize);
inChannel.read(mem);
mem.flip();
for (int i = 0; i < fileSize; i++) {
  int bits = Integer.reverseBytes(mem.getInt(j));
  float f = Float.intBitsToFloat(bits); // f is now the right value after by
}
```

4.4 Further Reading

The IEEE 754 Floating point standard from 2009: https://drive.google.com/file/d/1CT_AumUC4iHwdXRIJ9Lx4fwhKDEhrfVx/view?usp=sharing

https://www.h-schmidt.net/FloatConverter/IEEE754.html

http://weitz.de/ieee/

A nice summary of floating point properties and mistakes
https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

The origin of the term big-endian and little-endian is from the book Gulliver's Travels
https://en.wikipedia.org/wiki/Gulliver%27s_Travels

https://en.wikipedia.org/wiki/STL_(file_format)

5. Digital Electronics

5.1 Logical Operations

Logic operators are symbols or words that evaluate two or more Boolean (true or false) expressions. The result of the operation depends on the logical operator and the initial values of the operands. The three basic logical operations are AND, OR, and NOT. Through combinations of just these three operations, you can create much more complex operations such as XOR, NAND, and more.

Throughout the next few sections of the text, you will see true and false for Boolean values represented as the binary digits of ”1” and ”0”, respectively.

5.1.1 Basic Operations

AND - Given two Boolean values, returns true if *both* A and B are *true* (Denoted A*B or AB).

OR - Given two Boolean values, returns true if *either* A or B are *true* (Denoted A+B).

NOT - Given one Boolean value A, returns the *opposite* of A (Denoted $\bar{A}$).

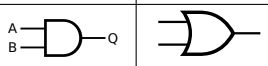
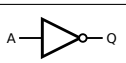
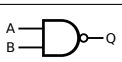
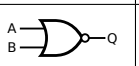
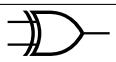
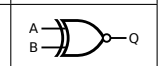
NAND - Given two Boolean values, returns true if *either* A or B are *false* (Denoted $\overline{AB}$).

NOR - Given two Boolean values, returns true if *both* A or B are *false* (Denoted $\overline{A+B}$).

XOR - Given two Boolean values, returns true if *either* A or B are true (Denoted $A \oplus B$).

XNOR - Given two Boolean values, returns true if *either* A or B are true (Denoted $\overline{A \oplus B}$).

The truth table below outlines the results of each seven operations given all possible inputs for the Booleans A and B. The chart also includes the respective gate symbols for each operator.

A	B	AND	OR	NOT A	NAND	NOR	XOR	XNOR
								
		AB	A + B	$\bar{A}$	$\overline{AB}$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$
0	0	0	0	1	1	1	0	1
0	1	0	1	1	1	0	1	0
1	0	0	1	0	1	0	1	0
1	1	1	1	0	0	0	0	1

The additional diagram below shows the Android logo robot as the different logical operators to help you get a better picture. The robot head represents the gate symbol of the corresponding logical operators.

5.1.2 De Morgan's Laws

De Morgan's Laws are a pair of transformation rules that relate AND, OR and NOT.

1. $\overline{AB} = \overline{A} + \overline{B}$
2. $\overline{A+B} = \overline{A} * \overline{B}$

5.1.3 Gate Properties

The logic gates we've been discussing are used in virtually all modern circuitry. However, given that they are being physically implemented, there are some other factors that we need to consider.

The first is what components we use to create them.

Because of implementation issues, CMOS (Complementary Metal Oxide Semiconductor) is typically NAND or NOR

You should know:

How to build a NOT out of NAND (tie inputs together) How to build a NOT out of NOR (tie inputs together) How to build an AND out NAND (build a not, and append NOT to the output of NAND)

Gate Delays: It takes a finite amount of time for a signal to propagate through a transistor, and through a gate. The speed of a computer is related to the speed of each gate, and the number of gates needed to get the answer.

Current time? 1ps (picosecond) Find out for a sample circuit!

Prefix	Order of Magnitude	Example of Modern Implementation
Exa-	10^{18}	current supercomputers approaching exa-scale ops/sec
Peta-	10^{15}	
Tera-	10^{12}	Hard Drive Storage: 4TB
Giga-	10^{9}	Motherboard RAM capacity: 64Gb or 128Gb (2-4 slots)
Mega-	10^{6}	L2 cache size
Kilo-	10^{3}	L1 cache size
milli-	10^{-3}	millisecond
micro-	10^{-6}	microsecond
nano-	10^{-9}	nanometers (size of circuits)
pico-	10^{-12}	picosecond (switching times)

5.2 Digital Components

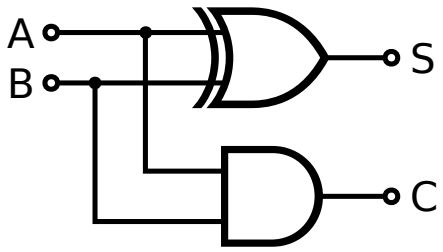
5.2.1 Adders

Half-Adder
Computers don't understand mathematics. The circuits within the Arithmetic Logic Unit (ALU) are just implementing the boolean logic we discussed previously. This logic happens to result in bit patters that are adding if we consider those bits to be integers. A half-adder circuit is the most basic part of addition. Taking two bits A and B as input, it is able to compute the result of their addition. When adding two numbers in binary, the result will of $0 + 0 = 0$, $0 + 1 = 1$, and $1 + 0 = 1$. However, when adding 1 + 1, you must perform a carry, moving a '1' into the next largest digit (or in this case, bit). For this reason, the half-adder has two outputs, often referred to as "sum" and "carry". The sum simply represents the XOR of the two inputs, as it will only have a value of '1' when

either A or B is 1, but not both. Carry can be represented by performing an AND of the two inputs, as the carry flag only turns on when both A and B are '1'. The truth table below reflects the logical outputs explained in this paragraph.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

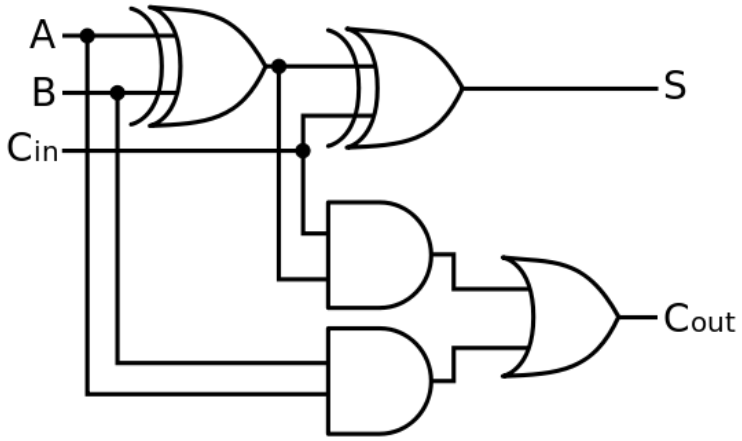
As was just mentioned, the Sum and Carry outputs can be represented by XOR and AND operations, respectively. Hence, we can represent a half adder using the logic gates explained earlier in this chapter.



Full Adder A half-adder is all well and good for performing addition on two bits. But what happens when we use the carry flag to move a value up one bit place? In that case, we will then need to add three bit values together to get the next result. Since our half-adder only takes two inputs, we can essentially string the output of one half-adder into another. To get our new sum, we will need to pass the initial sum through another XOR gate with our newly acquired 3rd bit, often called C_{in}. In the end, our final sum equates to A XOR B XOR C.

The value of C_{out}, our final carry, is slightly more complicated to find. Using the result of A AND B from before, we can OR that value with the result of C_{in} AND the original Sum. The truth table and logic gate equivalents of this can be found below.

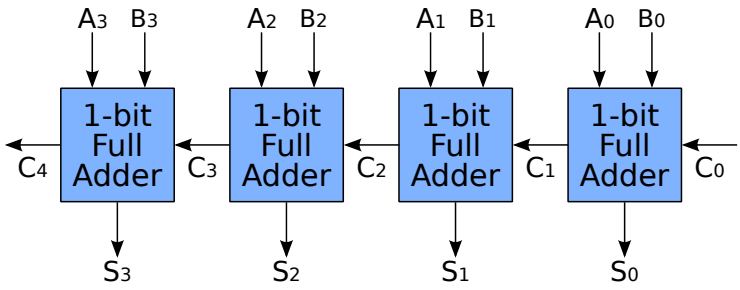
A	**B**	C_{in}	**Sum**	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Each circuit shown in the diagram above has two gate delays through the and and the or to the C_{out} which must in turn be fed to the next circuit. That means that the final bit will have $(n-1)(2)$ gate delays, in this case $n = 32$.

The full-adder allows us to accept the carry output from previous bits, performing more complex binary calculations without much extra hardware. This does still present a problem, though. The more bits we have to add together, the slower our circuit becomes. If you were to consider a 32-bit adder, there is lag time from the time the inputs A, B, and C_{in} change to the time the outputs are stable. This delay is directly based on wire delay and transistor switching rates. Currently, the fastest known transistor switches at 604GHz. Often times, wire delay (the amount of time for a wire to charge/discharge) is a more significant bottleneck than transistor switching rates.

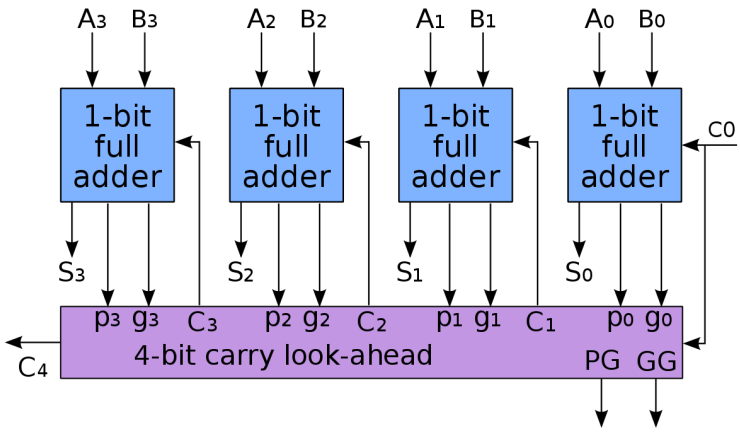
Ripple-carry Adder By combining multiple full adders, we are able to add N-bit numbers. Each adder creates a new carry $C_1, C_2, C_3, ...C_N$. Ripple-carry adders are made up completely of full-adders, with the exception of the first full adder. If the designer knows that the initial carry, C_{in} will always have a value of 0, then the first adder can be a half-adder. The diagram below is a high-level view of a ripple-carry adder.



The simple layout allows for fast design time, yet the actual execution time is relatively slow as each adder must wait for the carry bit of the previous adder.

Carry-lookahead Adder A carry-lookahead adder (CLA) reduces the problematic computation time of the ripple-carry adder's approach. Here, each adder creates two signals often referred to as P and G for each bit position. These signals are indicate whether a carry is *propagated* from a less significant bit position (at least one input is a 1), *generated* in that adder (both inputs are a 1), or killed in that adder (both inputs are 0). The P and G signal values are then analyzed to determine the carries for each bit position.

The result is a more complicated circuit, shown below, but the number of gate delays is significantly reduced. Instead of 32 gate delays from a 32-bit ripple-carry adder, using 4-bit CLAs to accomplish this task would mean there would only be 8 gate delays, from jumping across these groups of 4 to the highest group.

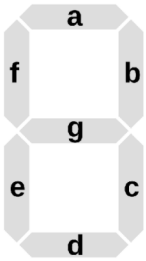


5.2.2 Decoders and Coders

Binary decoders are used to convert a certain amount of inputs, n, into 2^n unique outputs. Decoders are also able to have less than 2^n outputs, in which case certain outputs will repeat themselves. The easiest approach to fully understanding decoders is to analyze one.

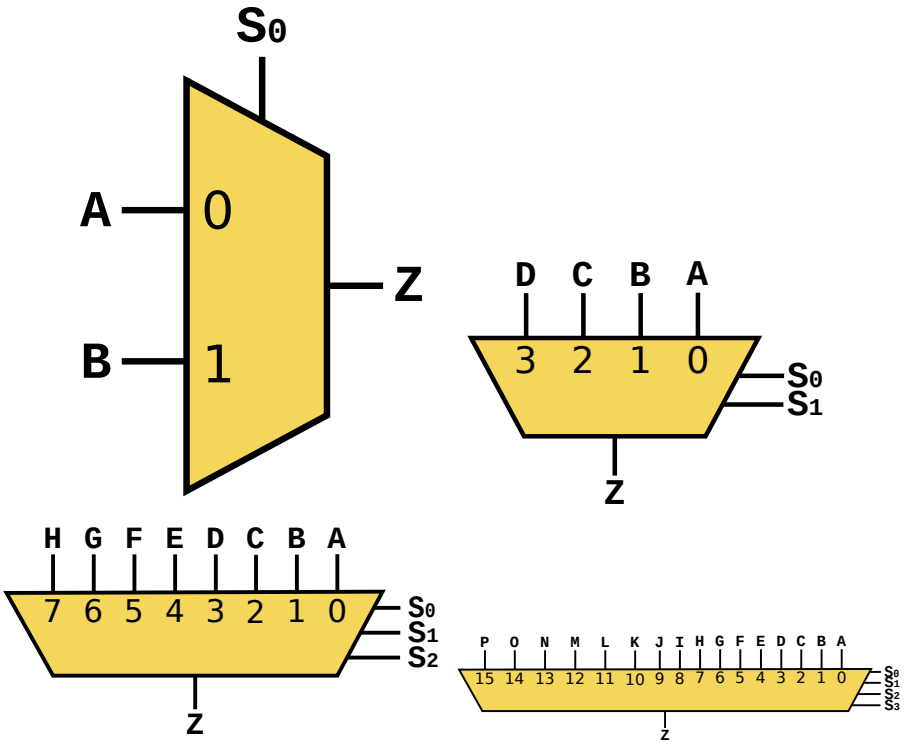
One common example of a decoder is a seven-segment display. Suppose you wanted to represent a hexadecimal value (0 to F) on the display shown below. Since a single

hexadecimal digit can be fully represented with four binary bits ($log_2 16 = 4$), we must create a decoder that converts our four binary inputs into the 16 unique outputs we can display on our seven-segment display.

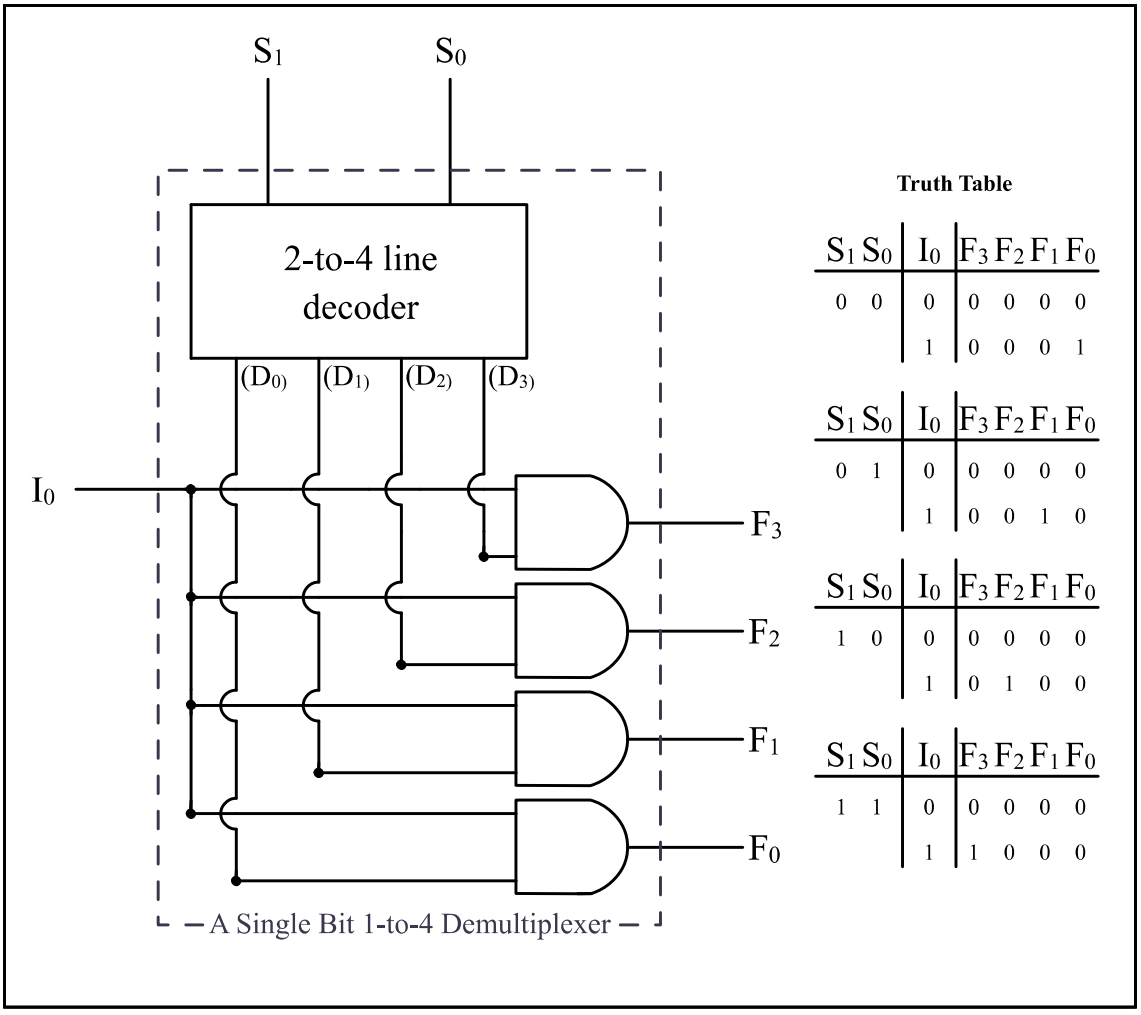


5.2.3 Multiplexers and Demultiplexers

Multiplexer - Often referred to as Mux, a multiplexer a data selector that chooses between n analog or digital input signals and sends it to a single output line. For example, A 2:1 or 2-to-1 Mux would be a multiplexer that has two lines of input and then chooses which of the two to send to the output line.



Demultiplexer - The opposite of a multiplexer would be a Demux. Rather than multiple outputs and one input, demultiplexers have one input and multiple outputs.



In order to share one line to transfer data, we could have a multiplexer selecting one of 2^n inputs, connected to a demultiplexer on the other side.

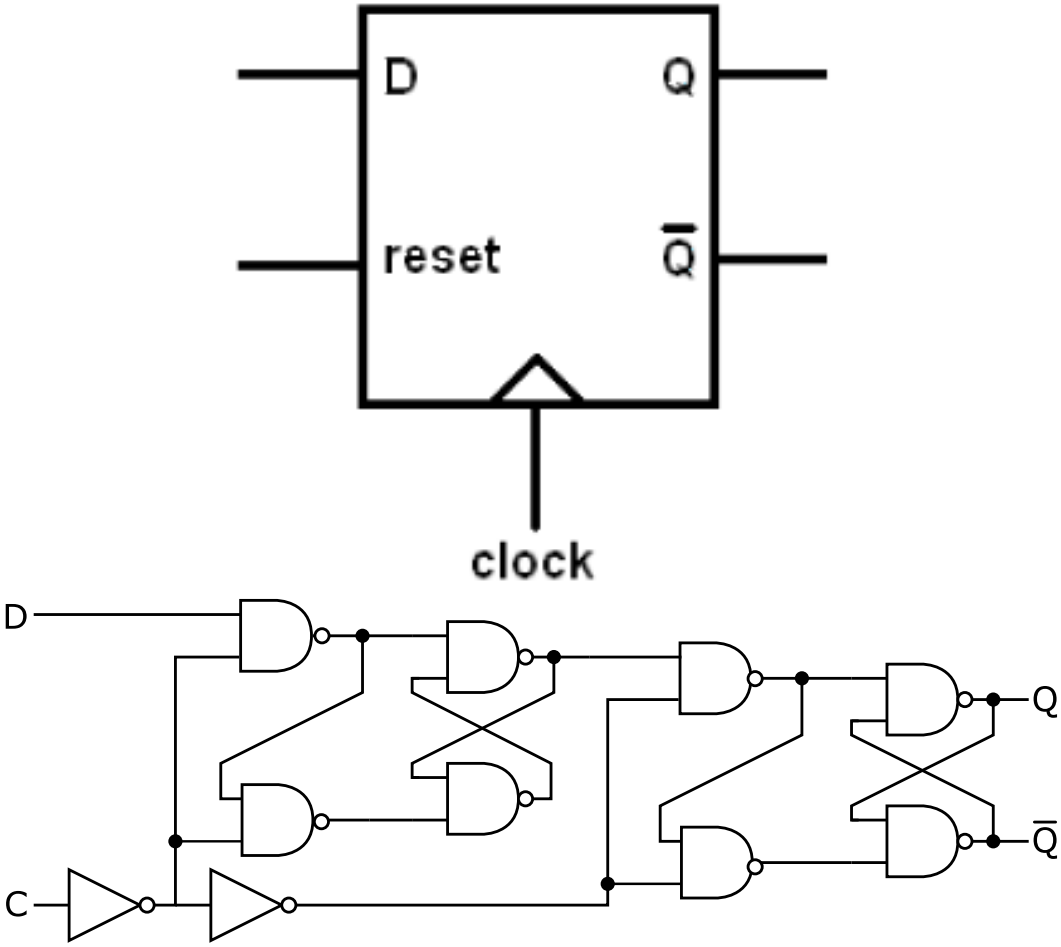
5.2.4 Latches

SR Latch
D Latch
JK Latch
T Latch

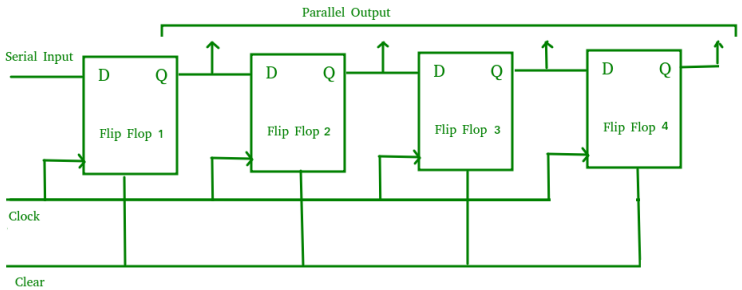
5.2.5 Flip Flops

SR Flip Flop
D-type Flip Flip - Flip-flops or latches are circuits used as data storage elements, capable of storing a single bit of data. They are the basic storage element of sequential logic inside and some types of computer memory. The D flip-flop is widely used as a memory cell. The nature of the ”data” flip-flop is such that whatever value is in the input on the

rising edge of the clock becomes the output. Otherwise, the output does not change. Below are two schematics of a D-type flip flop, the first at a higher level and the second made up of logic gates.



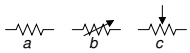
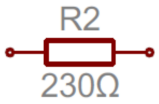
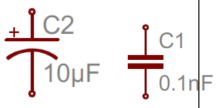
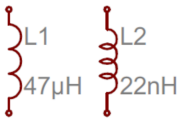
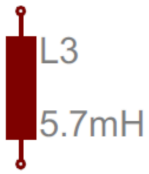
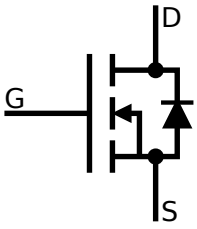
JK Flip Flop
T Flip Flop
Register A register is the fastest memory in a computer. Each bit is a D-type flip flip, and a 32-bit register is just 32 of those in parallel.

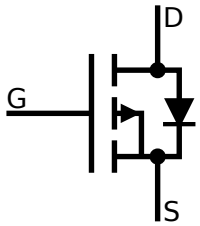
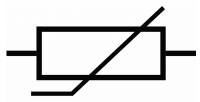


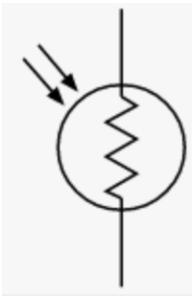
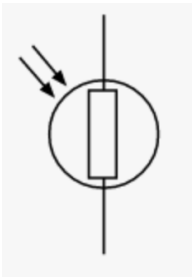
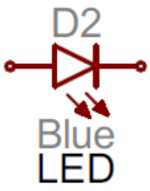
5.3 Exercises

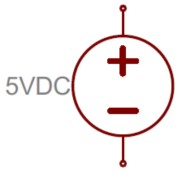
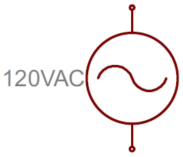
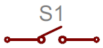
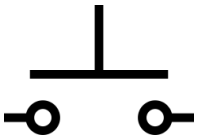
- Draw a logic design using AND, OR and NOT to enable an output when A = 1 and B = 0.
- Write out De Morgan's laws using AND, OR and NOT gates.

5.4 Essential Circuit Knowledge

Component Name	Description	US symbol	International Symbol
Resistor	Device with electrical resistance to reduce current flow		
Capacitor	Stores energy in an electric field. Can be polarized (image left) or non-polarized (image right)		
Inductor	A device that stores energy in a magnetic field by flowing electric current through a coil surrounding a magnetic core		
N-channel MOSFET	A Field Effect Transistor controlled by voltage to the gate with very high gate resistance. N channel FETs have the main channel doped negative and usually require the gate to be 10V higher than the source in order to be fully on		

P-channel MOSFET	A Field Effect Transistor controlled by voltage to the gate with very high gate resistance. P channel FETs have the main channel doped positive and usually require the gate to be 10V lower than the source in order to be fully on		
BJT	A Bipolar Junction Transistor (BJT) is a transistor that uses a small current to the base to control a larger current between the emitter and collector. Formed by joining N-type and P-type semiconductors either NPN or PNP with the middle layer much thinner controlling flow. BJTs require much more power than MOSFETS and MOSFETs dominate in computing, amplification and power applications, but BJTs are significantly cheaper.		
Variable resistor	A resistor with an adjustable resistance value		
Thermistor	A resistor with a resistance value dependent on temperature		

Photoresistor	A resistor with a resistance value dependent on light levels		
Diode	Only allows current to flow in one direction		
LED	Light-emitting diode, emits light when current flows through it		
Single Cell	A unit structure that generates electrical current by some means		
Battery	Cell(s) that store chemical energy and convert it into electricity as a source of power		

DC voltage source	Unidirectional flow of electric charge, used for charging batteries and by large power supplies		
AC voltage source	Current that periodically reverses direction, used by the main power grid and for appliances		
SPST switch	Single Pole Single Throw, has a single input and a single output. Often on-off switches		
SPDT switch	Single Pole Double Throw, has a single input and switches between connecting to two outputs		
Push button	A button that allows current to flow when pushed		
Ground	A reference point in a circuit to measure voltages, and also as a return path for electric current		

6. C++: Reviewing the Basics

6.1 Introduction

This chapter covers C++ features you are expected to know before taking this course, and the basic features of C++ we cover more deeply. The first section covers knowledge you should already have. Note that if you placed out of our intro programming course you presumably learned Java which is equivalent for these issues. These features will be reviewed, but quickly so please try to make sure you are familiar with them before we cover them and use the time to fill in any gaps in your knowledge.

Next, we cover data types in more detail. To understand how a computer implements a high level language you have to understand each data type, the finite range of numbers it can represent, and how much memory it requires.

We will cover loops, everyone is expected to be able to compute sums and products, and then once this is defined, to do it in assembler showing you undersand how the machine works.

We will cover arrays, blocks of memory containing a single type, and you will have to traverse the array processing those numbers in C++. Again, you will then have to do it in assembler and demonstrate that you can optimize code that is written inefficiently.

We will also cover object-oriented programming, where a class is defined containing different kinds of data. Classes are used to create objects. Functions that operate on objects are called methods. Since object-oriented programming is extremely popular, the dominant way of constructing large systems we will cover how C++ implements calls.

All that depends on understanding C++ basics first.

6.2 C++ Programming Review: What you should Already Know

The following basic skills you should already have mastered. While we will review, it is your responsibility to make sure you can do the following:

1. Understand that in C++, the preprocessor brings in files using #include that define commonly used code.

2. Recognize main as the special function that starts programs in C and C++.

3. Declare variables of the basic integer and floating point data types.

4. Understand that a given symbol name may only be defined once in one location. In other words the variable a can only be defined once in a function.

5. Identify the constants for basic types (integer and floating point).

6. Recognize a character constant that uses single quotes 'x'.

7. Recognize a string using double quotes "testing" and know that there is a hidden final character whose code has the value zero. So **char** s[] = "abcd"; contains an extra character

8. Be able to print values using cout and read in values from cin.

9. Recognize scientific notation in floating point (6.022e+23 or 6.67408e-11).

10. Identify the results of integer operations +, −, *, /, %.

11. Recognize the form operator= such as a += b; which is equivalent to a = a + b;.

12. Identify the results of floating point expressions +, −, *, /.

13. Recognize the most common functions in the math library **#include** <cmath>,

14. Write loops and identify correct end conditions (**for**, **while**).

15. Understand that loop variables in a for that are declared are only valid within that loop and may therefore be used again in another loop.

16. Be able to identify an infinite loop (a loop that never ends).

17. Be able to identify a loop that does nothing (a loop where the initial condition fails and executes 0 times).

18. Recognize that a for loop is just a different syntax for a while loop, that the two are equivalent.

19. Write functions and pass parameters.

20. Understand that passing values to a function normally is pass by value which copies the value.

21. Understand that variables inside each function are private to that function, not visible in the rest of the program (this is called scope).

All these features are summarized in this chapter. If there is anything you do not know, read the section, and look up on the internet if you need further explanation. Then, ask in class. We will build from this common knowledge base.

For example, you should be able to write every individual component in the following program or read a program containing these language features.

```
#include <iostream>
using namespace std; // means we do not have to write std::cout every time
int f(int x) { return x*x; }

int main() {

  // basic data types and constants you should be familiar with
  int a = 2;
  float b = 2.5f;
  double c = 1.25;
  double d = 6.022e+23;
  cout << a << ' ' << b << ' ' << c << ' ' << d << '\n';

  // basic integer operations
  int e = 2 + 3 * 4; // order of operations
  int f = 2 / 3;     // integer division truncates
  int g = 7 / 2;
  int h = 3 % 2;     // modulo
  cout << e << ' ' << f << ' ' << g << ' ' << h << '\n';
  e += 4 - 3; // operator= form: +=, -=, *=, /=, %=
  g *= 5 + 1; // first evaluates the right side, then multiplies g by 6
  cout << "e=" << e << " g=" << g << '\n';
}
```

The following program contains a while loop and a for loop. Both are equivalent except that the variable declared in the for loop is valid only within the for loop. This means that the more compact for loop has the advantage that you could write two loops back to back using the same variable name and it works, which it would not with the while version. (You may not redeclare a, but you may reassign it to a new starting value and write a second loop).

```
#include <iostream>
using namespace std;
int main() {
  int a = 2;
  while (a < 10) {
    cout << a;
    a += 3;
  }
  cout << '\n';

  /*
```

```
    note: this for loop is equivalent to the preceding
    while loop except that a here is local to the loop
  */
  for (int a = 2; a < 10; a += 3)
    cout << a;

  for (int a = 1; a <= 51; a += 10)
    cout << a;
  return 0;
}
```

The basic data types int and float you should already have encountered in an entry level program. We are being more rigorous about understanding them. You should be able to write a loop. At the very least, be ready to do the following:

1. Count up linearly
2. Count down linearly
3. Multiply (exponential)

If you can recognize the following you have the required background. If not, try to review loops by writing some. You should be able to identify what the following program prints. You should be able to identify loop termination conditions, knowing when the loop starts and stops.

```
#include <iostream>
using namespace std;
int main() {
  for (int i = 0; i < 10; i++)
    cout << i;
  cout << '\n';

  for (int i = 1; i <= 10; i++)
    cout << i;
  cout << '\n';

  for (int i = 1; i <= 256; i *= 2)
    cout << i;
  cout << '\n';
}
```

You should be able to sum a series and compute products. These are two basic patterns of computation. Note that in C and C++ variables defined in functions are on the stack (we will discuss exactly what that means in assembler later) and for speed, are not initialized unless you explicitly do so. If you do not initialize your variable, the value is random. It may even happen to work, but it is not reliable, and this is a serious error

The following program shows the pattern for computing the sum of the series from 3 to 23 in steps of 5,

Note that for loops containing a single line, the curly braces { } may be omitted. While we will review this rule, if you find easier, always use curly braces on your loops.

```
int main() {
  int sum = 0;  // create a variable and initialize it
  // start from the first number, and go while i is less than or equal to.
  for (int i = 3; i <= 23; i += 5)
    sum += i; // each time, add the loop variable onto sum

  cout << sum << '\n'; // print the answer
}
```

The following program computes the product of the numbers from 4 to 8.

```
int main() {
  int prod = 1;  // Notice for multiplication variable must start with 1

  for (int i = 4; i <= 8; i++) {
    prod *= i; // each time, add the loop variable onto sum
  }

  cout << prod << '\n'; // print the answer
}
```

It is important to understand how to write this code, but it is also important to memorize it. This is a recipe, a standard pattern of computation in many languages.

6.2.1 Arrays

You should also know about arrays, a single block of memory containing all the same elements, indexed by an integer position. The first position is at location [0]. For people who have learned Java not C++, be aware that the square brackets must come after the variable name. The following program walks through an array a and prints each element in order:

```
int main() {
  int x[] = {7, 6, 3, 1};
  for (int i = 0; i < 4; i++) {
    cout << x[i];
  }
}
```

You also should know how to write values into an array. For example, predict what is printed by the following program:

```
int main() {
  int x[10]; // x is full of random garbage
  for (int i = 0; i < 10; i++) {
    x[i] = 7-i;
  }
  for (int i = 0; i < 10; i++) {
    cout << x[i] << ' ';
  }
}
```

Putting arrays together with sums and products, you should be able to compute the sum or products of elements of an array. For example, the following program shows loops that compute the sum and product of the elements in the array x:

```
int main() {
  int x[] = {5, 4, 3, 7};

  int sum = 0;
  for (int i = 0; i < 4; i++) {
    sum += x[i];
  }
  int prod = 1;
  for (int i = 0; i < 4; i++) {
    prod *= x[i];
  }
}
```

Note that in the above case, if you added another element to x, the program would become wrong because both loops count 4 times. A technique to avoid this error is to define a constant describing the size of the array:

```
int main() {
  const int size = 4;
  int x[size] = {5, 4, 3, 7};

  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += x[i];
  }
  int prod = 1;
  for (int i = 0; i < size; i++) {
    prod *= x[i];
  }
}
```

In this case, if you add an extra element to the array, then the size exceeds 4, and the compiler will report an error, reminding you to change size. And every place in the program uses the symbol size so all the loops will calculate correctly.

6.3 C++ Fundamentals in More Depth

Even if you know the background material, there may be gaps in your knowledge. This section attempts to define completely the basic facts about C++ to aid in describing exactly how the computer works. After we cover this in C++, we will do it again in assembler, where you will see how the computer actually implements each C++ feature.

6.4 Data Types

C and C++ can generate very efficient code. Today, a good compiler can generate code that is as efficient as hand-written assembler in most circumstances. This lets programmers be more productive because it is definitely easier to write more code in a higher level language. However, the definition of C types was vague and depended on the computer. This means that a program running on different computers may have different results. Ideally, high level languages should increase programmer productivity and also make the program portable to any computer, and isolate the user somewhat from the machine. In the zeal to provide efficiency, the portability goal failed for C. After many years, in 2011 the C++11 standard defined portable types that would work the same everywhere. So why study and understand the original types? Because there are presumably billions of lines of code already written using them. Ideally you should not use these types (except for float and double which are standardized by IEEE), but rather the ones in the next section. Still, you have to know them. There is no standard for long double. More detail on floating point computation in a separate chapter.

Table 6.1: Non-portable types

Type	Required	Typical Computer	Arduino(8 bit)
int	>= 16 bits	32 bits	16 bits
short int (short)	16 bits <= short int <= int	16 bits	16 bits
long int long	32 bits <= long	32 bits? or 64?	32 bits
long long	>= 64 bits	64 bits	64 bits
char	Minimum addressable memory unit	1 byte	1 byte
float	32 bits/4 bytes	4 bytes (IEEE 754)	4 bytes
double	64 bits/8 bytes	8 bytes (IEEE 754)	(same as float)
long double		?? no standard	(same as float)
bool	true or false	1 byte	1 byte

all the integer types above may be signed or unsigned: (implementation-defined)

Portable types are the same on all computers, defined since C++11. If you are writing a program from scratch, you should ideally use these. If you do not need negative numbers and are representing a whole number, then unsigned integers are the natural

way to go. A good general principle is to try to use a data type that represents the values you want represented and if possible, does not represent values that are not legal.

Table 6.2: Portable types

Type	Size
int8_t / uint8_t	signed/unsigned 8 bits (1 byte)
int16_t / uint16_t	signed/unsigned 16 bits (2 bytes)
int32_t / uint32_t	signed/unsigned 32 bits (4 bytes)
int64_t / uint64_t	signed/unsigned 64 bits (8 bytes)

6.4.1 Best Practices with Data Types

You need to know all the data types to be able to read existing code. But best practice for a new program is to write code that is portable. Here are some rules to write better code.

1. If you are writing code involving positive integers, use an unsigned data type. If you use a signed one, what would happen if the user entered a negative value, or if you computed one? If the data type simply doesn't support negative numbers, that is one less problem to worry about.

2. Think carefully about how large the answers are going to need to be. If you need to handle giant numbers like $123x10^{50}$ then you need to use double. It has only 15 digits precision but can handle numbers up to 10^{308}. Never use single precision floating point. As you will see when we go deeper into that topic, the 7-8 digits accuracy can be lost very quickly, leaving an answer that is complete garbage.

6.5 Character Data

char in C/C++ is a single atomic unit of memory. Mostly this is one byte. But on the Cray-1, where for speed the smallest addressable memory was 8 bytes, then a char = 8 bytes.

char is a one byte integer. Typically signed but not guaranteed.

Since in g++ it is signed, the value ranges from -128 to 127.

```
char x = 'a';  // the ASCII value is 97
x = 98;        // same as 'b'
x++;           // now it is 'c'
```

6.6 Variable Initialization

Variables declared in a function are by default allocated on the stack. They are uninitialized by default (because it would take time to initialize them) and are therefore whatever random garbage was in memory on the stack when they are assigned that location.

Variables declared outside of a function are global. They may be accessed by any function anywhere in the program. They are preloaded with the code and are initialized to binary zero (all zero bits) when the program begins.

```
int myvar; // global variable outside function = 0 when the program is loaded
int glob2 = 3; // comes loaded with 3 from disk, ready before main is entered
int main() {
  int x;                // born when the main is entered,
                        // uninitialized (random value)
  int y = 2;            // born when main is entered, initialized
                        // to 2
  int a[10];            // born when main is entered, uninitialized
  int b[10] = {2};      // b[0] = 2, all other values initialized
                        // to zero when the function is entered
  int c[] = {1, 2, 3};  // c is exactly 3 elements
                        // 3*sizeof(int)=12 bytes on most computers
                        // on Arduino (8 bit computer) 3*2 = 6 bytes
  int c[3][4] = {{5, 4, 1, 4}, {-1, 2, -3}, {2, 5, 4}};
  /**
   * array contains
   * 5    4   1    4
   * -1   2   -3   0
   * 2    5   4    0
   *
   * physical order in memory is row-major order:
   * 5    4   1   4   -1    2   -3    0   2   5   4   0
   */
}
```

6.7 Operators

Table 6.3: Integer Arithmetic Operations

`int a = 2 + 3;`	integer addition can overflow
`int b = 2 - 3;`	can result in a negative for signed values, can overflow
`int c = 2 * 3;`	
`int c = 1000000 * 1000000;`	overflow, more than 2^{31}
`int d = 2 / 3;`	integer division, answer = 0
`int e = 19 / 10;`	integer division, answer = 1
`int f = 17 % 8;`	modulo (remainder)
`a += 3;`	`a = a + 3`
`a -= 2;`	`a = a - 2`
`a *= 4;`	`a = a * 4`
`a /= 5;`	a = a / 5;
a %= 6;	`a = a % 6`

6.8 Exercises

1. Write a program to read in your age, and output your age in seconds using the int datatype. This should not overflow at your age. However, if the number gets big enough, it will overflow. Find out the number beyond which your program does not produce the correct answer due to overflow.

2. Write 4 for loops to print the following:
0 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 10 10 9 8 7 6 5 4 3 2 1 0 blastoff! 1 2 4 8 16 32 64 128 256

3. Write a program to calculate the 3n+1 conjecture. Have the user type in a starting number. Then if the number is odd, multiply by 3 and add 1. If the number is even, divide by 2. Print the result and keep doing it until reaching 1.

4. Find the problem in the following loop which is supposed to print 3 6 9 12 15 18 21:

```
for (int i = 3; i > 21; i += 3)
  cout << i << ' ';
```

Examples of bitwise operations

Table 6.4: Bitwise Operations

`c = a & b`	a AND b (bitwise AND)
`c = a | b`	a OR b (bitwise OR)
`c = a ^ b`	a XOR b (bitwise XOR)
`c = ~a`	NOT a (bitwise NOT)
`c = a << 3`	shift a left by 3
`c = a >> 4`	shift a right by 4
`c = (a << 15) | (a >> 17)`	rotate a right by 17
`a <<= 2`	`a = a << 2`
`a >>= 3`	`a = a >> 3`
`a &= 1`	`a = a & 1`
`a |= 5`	`a = a | 5`

Table 6.5: Logical Operations

c = a && b	a AND b
c = a || b	a OR b
c = ! a	NOT a

6.9 Arrays

In C/C++ an array is a single block of memory containing many of the same type. The elements are accessed by position with array[0] being the first element. For an array of size n, the last element is in array[n-1].

```
uint32_t a[10];           // 10 elements of 4 bytes each, total
                          // size 40 bytes
uint64_t a[1000];         // 1000 elements, 8 bytes each, total
                          // size 8000 bytes
uint32_t a[3][4];         // 2d array of 3 rows, each 4 columns
                          // size = 3 * 4 * sizeof(uint32_t) =
                          // 12 * 4 = 48
uint16_t a[2][3][4][5]; // 4-D table size: 2 * 3 * 4 * 5 *
                          // sizeof(uint16) = 120 * 2 = 240 bytes
char s[80] = "ABC";       // s[0] = 'A' s[1] = 'B' s[2] = 'C'
                          // s[3] = '\0';
char s2[] = "testing";    // note size is 8 (don't forget the
                          // terminal '\0')
```

Arrays of char are called strings in C. Each letter is one byte (not good for Chinese/Japanese/Korean multi-byte letters). The string terminates with the character whose code is 0, written '\0'. Note this is different than the letter '0' whose code is 48.

Table 6.6: Comparison Operations

c = a < b	a is less than b
c = a <= b	a is less than or equal to b
c = a > b	a is greater than b
c = a >= b	a is greater than or equal to b
c = a == b	a is equal to b
c = a != b	a is not equal to b

6.10 C++ Objects and Pointers

6.10.1 Introduction to Object Oriented Programming

Object-oriented programming is a popular organizational technique to improve programmer productivity. By grouping data into objects, hiding the data from the outside world, programs can be broken down into smaller, hopefully reusable pieces that each can be tested separately. Object oriented programming has not quite lived up to its hype, but it is the most popular programming model in the world today.

In order to program in object-oriented style, programmers must define a new kind of entity. The built-in types like integers , floating point, and strings can be combined into classes. The following class definitions show a Fraction containing a numerator and denominator, and a person containing a first name

6.11 Memory Layout

The old C keyword struct and the newer C++ class allow defining a block of memory containing different types.

```
struct Person {
  char firstName[10];
  char lastName[10];
  int age;
  double weight;
};

class Vector {
  public:
    double x, y, z;
    Vector(double x, double y, double z) : x(x), y(y), z(z) {}
    double abs() const { return sqrt(x * x + y * y + z * z); }
};
```

The above code defines a new type called Person, which contains 4 fields: firstName,

lastName

6.12 Public and Private Symbols

Classes define

When you call a method, there is a hidden first parameter that is a pointer to the object in memory. In other words, given the method a.f(2,3) C++ will call the method f, passing three parameters:

```
f(pointer to a, 2, 3)
```

Of course it takes time to call functions and pass parameters, and object oriented programming is often gigantic sequences of calls to small functions. C++ will attempt to remove the call to the function and replace it by the code executed, if doing so will improve efficiency. This will be discussed later under Computer Optimization.

7. Assembly Language Overview

This chapter covers assembler programming, mostly ARM. We use ARM because it is dramatically simpler than Intel assembler. The disadvantage is that we need another computer to do it. That's why you have the Raspberry Pi. What makes the ARM CPU so much simpler than Intel? Why, if the design of the ARM is better, doesn't Intel simply change their design?

The answer is complicated, but essentially Intel started out with a 4 bit CPU (4004) in 1973. By 1983 IBM was creating the PC, and selected the intel 8088, largely for business reasons as the Motorola 68000 was dramatically better. The infusion of cash Intel got from the huge volume of business enabled them to stay ahead building each new generation of CPU, but there was a price – the reason everyone wanted an Intel was that it was compatible with earlier models. Each new CPU had to run old DOS programs. While they have managed to ditch some of the oldest and ugliest parts, they still emulate those in software, so today's Intel CPU is still probably capable of running the earliest DOS programs.

On the other hand, with no history, ARM was free to just design the simplest best processor they could. Intel is a CISC processor (Complex Instruction Set Computer). It has approximately 1500 instructions. Worse, the length of those instructions ranges from 2 bytes to 7 bytes. That means that if the CPU is looking at the current instruction, it isn't simple to figure out where the next instruction begins, because that depends on how long the current instruction is. This makes it much more complicated to pipeline (start to decode the next instruction while working on this one).

ARM by contrast is an example of a RISC (Reduced Instruction Set Computer). There are more like 90 instructions on the ARM though it is difficulty to find that number. And while older versions of ARM have experimented with some variations, the current version of ARM, and the subset we are studying is very simple: all instructions are 4 bytes (32 bits). The instructions are highly regular. Being all similar keeps the decoding instruction simple. This uses fewer transistors, leaving more available for other purposes. What can be done with a more efficient design?

- Use less power (just have fewer transistors)
- Less expensive (Processed silicon by weight is more expensive than diamond)
- Add more memory. Instead of complicated circuitry have more registers, more cache, etc.

- If each CPU uses few transistors, have more CPUs (cores) on a chip.

7.1 High Level Languages

So far, you have programmed in high level languages like C++. What happens when you compile a c++ program? When you type:

g++ mycode.c

The compiler g++ does the following:

- Compiles mycode.c to machine language

- Stores it in an object file mycode.o, though it may just immediately feed that into the linker and destroy the file (on windows using microsoft tools, mycode.obj)

- Runs the linker which combines the object file with the code you wrote and the c++ libraries that support common actions like printing to the screen, and builds an executable. If you refer to a function or variable that does not exist the linker will report an error, that a symbol is not found and refuse to build the executable.

Now that we are going to look at assembler programming, we will look at the layer that the computer sees, and learn the sequence of assembler (machine) language that executes the program works. By understanding how your code really works, you will learn how to make your code faster, and also learn what can go wrong in a language like C++ where you can go out of bounds in memory.

The following is the summary of Unix tools we will be using to build assembler programs. These tools are all available on Windows under msys2, in Linux and on the Mac.

g++	This is the GNU c++ compiler. If you compile an assembler program it will automatically do the right thing. It will compile .cc or .cpp files as c++, .s files as assembler.
as	This is the assembler itself. It is easier to use g++ which can also compile c++ and link pieces of programs together. When you compile an assembler program using g++ it runs the as command to compiling the assembler code.
ld	The unix linker. This combines one or more .o files and any libraries the program may be using into an executable.
file	Identifies file types by looking at the first few bytes (the magic number).
xxd	Dumps out the contents of a binary file in text and hexadecimal.
od	Octal dump, also can dump the contents of a binary file in hex or text.
objdump	Display the contents of an executable or object file.
gdb	Gnu Debugger, used to step through your programs to find out why they aren't working.
cgdb	A better version of gdb that has fewer screen refresh problems.
make	The make utility, used to store the instructions to efficiently build large programs made of many files.

Here are the key examples for using the programs above:

g++ myprog.cc	Compile your program into an executable named a.out
g++ -S myprog.cc	Compile your program into assembler so you can see what the assembler code looks like. Does not run.
g++ -O2 -S myprog.cc	Compile with optimization and convert to assembler. Unoptimized code is big and ugly. Optimized is not only faster but easier to understand. Usually when we look at how C++ generates code we will look at optimized code. Unfortunately the optimizer is so good that sometimes it mutates a program in ways that make it hard to understand. We will discuss that in class.
./a.out	Run the program a.out in the current directory. Note that Unix does not include the current directory in the path by default, you will have to type ./ in front of a.out or you will get the error "command not found"
g++ -c myprog.cc	Compile only (do not link) the program, outputting the object file myprog.o
g++ myprog.s	Compile an assembler program and generate an executable. Note that ARM machine language will only run on ARM computers like Raspberry Pi, and Intel machine language only on your laptop or desktop, assembler isn't portable at all. There are emulators which can run code from one computer on another, but since this is software simulating the other computer, the emulator is much slower.
as myprog.s	Compile an assembler program and generate the object file myprog.o
file a.out	Display the type of the file a.out.
xxd myprog.o	Dump out the contents of the object file in hexadecimal.
objdump −d myprog.o	Examine an obj file or executable in assembler.

Exercises

- Write a function f in C++ that takes two integer parameters and returns their sum. Compile using -O2 -S and show the name of the function internally in assembler and the instruction that does the addition.

- Compile the same function using -c and generate the object file. Show the output of running the file utility on the object file.

- Write a program in C++ that calls the nonexistent function f(). Compile it and show the error.

- Write a program in C++ that defines the function prototype **void** f (); and calls it from main. Show the error. This one is different than the above. What program is reporting the error?

- Write a program that prints hello world, compile it. Use the file utility to display the file type of the executable. Display the first 8 bytes of your program in hex using the xxd utility.

If you are interested in going further we will not be using the following tools, but you might want to know about them

ar	The unix archiver utility builds static code libraries (.a) out of one or more executables.
g++ -shared	The shared option in g++ is how to build a shared object library (.so)
valgrind	Tool to find memory bugs in C++.
ldd	Utility to display which dynamic libraries your program uses (also shows you if any are missing).

7.2 Levels of Machine Language

At the lowest level, computers run by being fed instructions which are bit patterns that tell the computer what to do. For example, the instruction 1110... might tell a computer to add two numbers and store the result. Binary is hard to read, so the next level would be to encode the numbers in base 16 (hexadecimal) where each digit represents 4 bits. This is still really cryptic, so programmers typically use an assembler, a programming language where the instructions are given names like mov, add, mul, etc. This makes the code easier to understand. Assembler also provides symbolic addresses. While a computer instruction will always be at a specific address in memory, most of the time the exact location does not matter. We can think in terms of symbols, and give names to sequence of code (subroutines).

7.3 Types of Assembly

In this course we cover ARM assembler because it is far simpler and better designed than Intel assembler. For anyone interested in Intel assembler (since that works on your laptops) there is a chapter giving an overview but it is not part of the course.

7.4 Getting Started: Labels, Assembler Directives, and GDB

An assembler program consists of machine language instructions, but it must also define symbols which then match up with addresses when the program is loaded. The following first program in ARM assembler defines the name main which is the entry point to the program. We do not know where main will be located in memory until we watch it being run in the debugger, but it will in the end match up to some location in memory like 1005c0. Most symbols in a program are only of interest within that program, so in order to report a symbol to the operating system (like main which it has to know in order to start your code) the symbol is declared global using the .global assembler directive.

The lone instruction in this program is "bx lr" which returns from the program. This is discussed later.

```
    .global main
main:
    bx lr
```

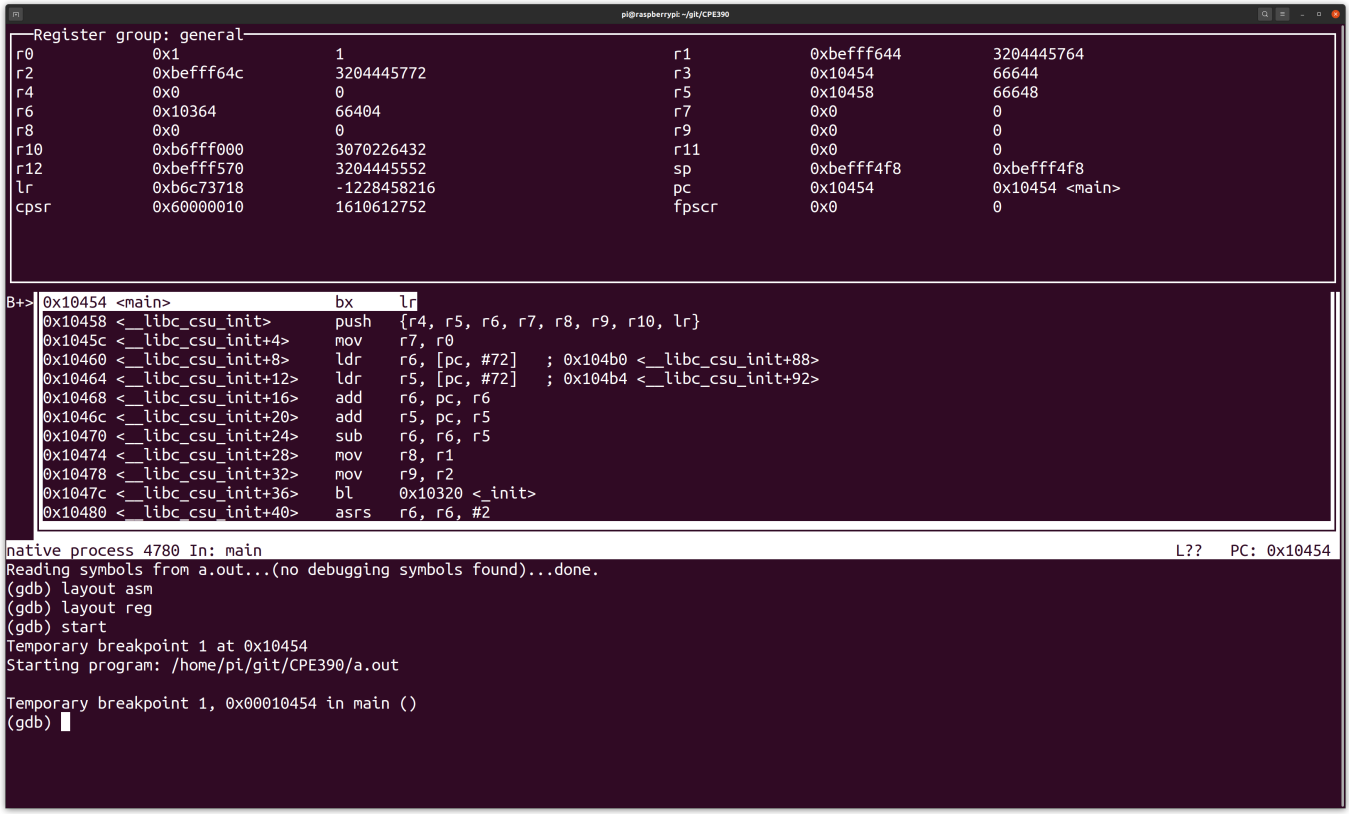
Intel assembler is covered in a separate chapter, and is not part of this course. However, just to show an Intel program once for comparative purposes, here is the same program in Intel assembler. Given the small size it does not look very different. Note that the assembler directive is spelled differently (.globl) and that the return instruction is called ret:

```
    .globl main
main:
    ret
```

In order to debug a program, you will need to run the debugger. gdb is built into Linux. It is command line oriented and not that pretty but it runs on everything. The biggest problem with gdb is that every time the program prints a newline the screen layout gets disturbed and you have to type control-L to refresh it. There is another debugger called cgdb that avoids this problem. Commands to install cgdb:

Msys2	pacman -S cgdb
Ubuntu	sudo apt install cgdb

For most of this course we will be using gdb to look at ARM assembler, showing the assembler code and the registers that comprise the state of the computer. Here is a screenshot of the ARM program running.



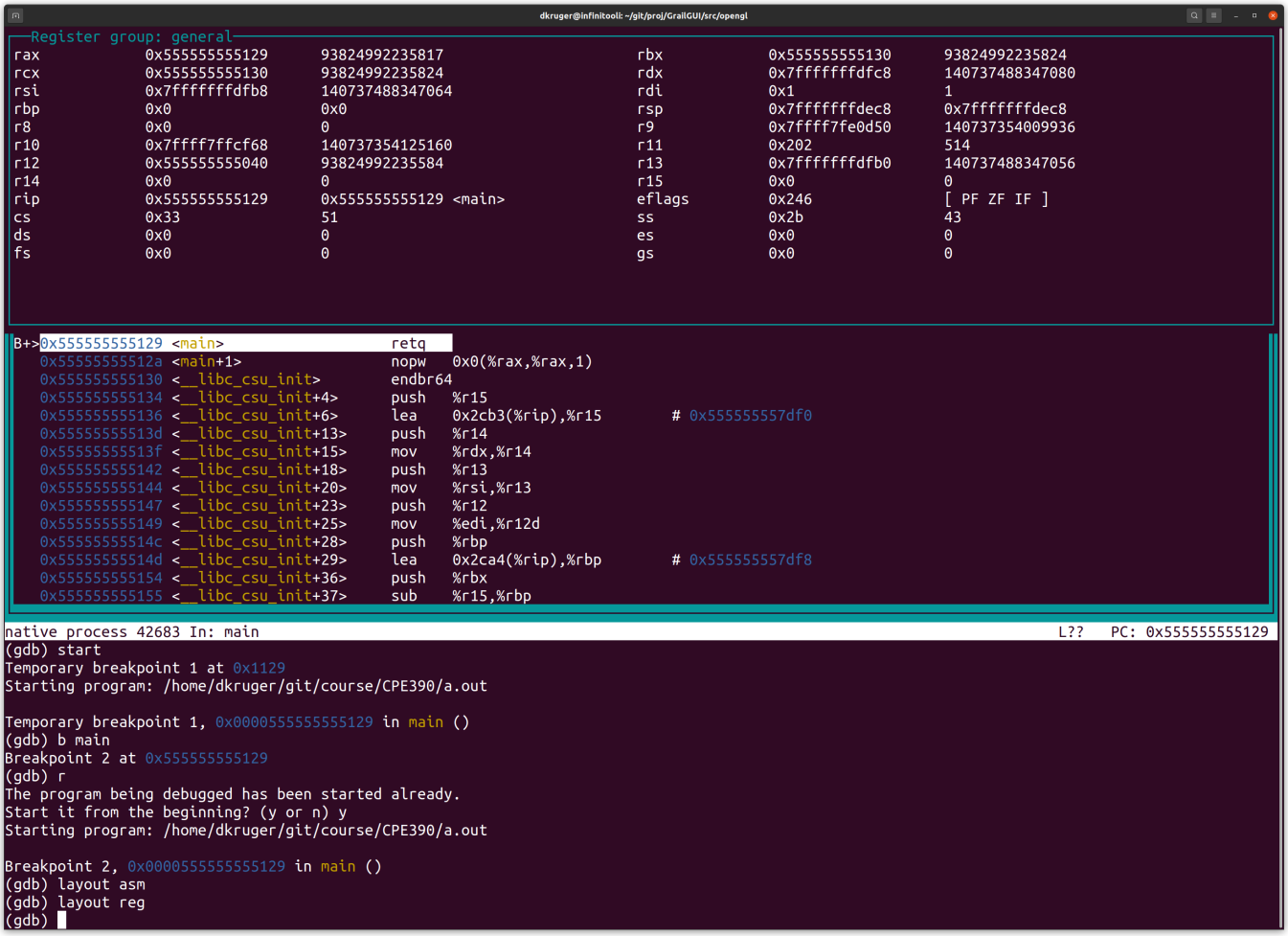
In order to get gdb to look this way we have to execute two commands:

```
layout asm
layout reg
```

Because you will typically want to do this every time you run the debugger, it is easier to put these two commands into a script file that is run automatically every time you enter gdb on the Rasperry pi. To do this, edit the file /.gdbinit and type the two commands in there.

In order to start the program, type the command start.

The following screenshot shows running the gdb debugger on Intel, having typed the commands ”layout asm” and ”layout reg” to show the assembler instructions and the integer registers. Note that the instruction ”ret” we typed turned into ”retq”. This is typical, there are often shorthands in assembler programming where the programmer describes what they want to do (return) and the assembler decides which specific instruction is most efficient in this case. Notice also that the registers are completely different names than the ones in ARM:



7.5 Instruction Set Overview

7.5.1 Introduction

Computers are machines to execute a set of instructions (the instruction set). We can group the ARM instructions into a number of categories.

1. Loading values into registers and storing them.
2. Integer math on registers (add, subtract, etc)
3. Floating point instructions
4. Comparison/testing (changing processor status)
5. Branching

7.5.2 Loading Registers

The ARM requires that all arithmetic must be performed on registers for speed. This means the first job is to load registers with values. The first instruction that can load values into registers is the mov (immediate mode). It takes a destination register and a number from 0 to 255. For example:

```
mov r0, #3
mov r1, #242
mov r2, #0xA0
```

loads register r0 with the number 3, register r1 with the value 242, and register r2 with the value 160 (A0 in hex is 10*16).

The mvn instruction (move and negate) loads the register with NOT the number

```
mvn   r0, #ff
mvn   r0, r1              @r0 = NOT r1

ldr   r0, =0x12345678     @r0 = 0x12345678 (hex)
ldr   r1, [r0]            @load 4 bytes of r1 from the location specified in
                          @register r0
ldrb  r1, [r0]            @load one byte (the low byte) of r1 from  location
                          @in r0
str   r1, [r0]            @store r1 into 4 bytes at r0
strb  r1, [r0]            @store the low byte of r1 into the location at r0
```

Since registers in ARMv7a are 32 bits, there must be a way to load an entire 32-bit value, and it is through the pseudo-instruction:

```
ldr r0, =0xF1234567
```

The above is actually an ldr instruction, but involves a more complicated mode, so just for now please trust that this can load all bits into a register.

The opposite of ldr is str. A register may be stored out to memory, but that involves more complicated addressing modes so it is discussed later.

7.5.3 Arithmetic Operations

Arithmetic on the ARM can only be performed on registers. All arithmetic instructions take a destination register and two source registers. The following table shows instructions for the common operations such as addition, subtraction, multiplication (there is no division or modulo on ARMv7), as well as AND, OR, XOR. For each of these arithmetic operations, there is a variant (immediate mode) where the second operand is a number from 0 to 7

Of course, there are some unary operations as well, such as negate, bitwise NOT, etc. The following instructions are common unary operators:

`mvn r0, r1`	$r0 \leftarrow$ NOT $r1$ (move and negate)

`add r0, r1, r2`	$r0 \leftarrow r1 + r2$
`add r0, r1, #3`	$r0 \leftarrow r1 + \#3$
`sub r0, r1, r2`	$r0 \leftarrow r1 - r2$
`sub r0, r1, #5`	$r0 \leftarrow r1 - 5$
`mul r0, r1, r2`	$r0 \leftarrow r1 * r2$
`mul r0, r1, #7`	$r0 \leftarrow r1 * r2$
`and r0, r1, r2`	$r0 \leftarrow r1$ AND $r2$
`and r0, r1, #6`	$r0 \leftarrow r1$ AND $r2$
`orr r0, r1, r2`	$r0 \leftarrow r1$ OR $r2$
`orr r0, r1, #4`	$r0 \leftarrow r1$ OR $r2$
`eor r0, r1, r2`	$r0 \leftarrow r1$ XOR $r2$
`eor r0, r1, #1`	$r0 \leftarrow r1$ XOR $r2$

There are also clever operations that are designed to reduce the number of instructions. If the CPU designers notice that often, programmers perform a mvn followed by and they try to make it faster. The following operations combine two simpler ones:

`mla r0, r1, r2`	$r0 \leftarrow r0 + r1 * r2$ (multiple and add)
`bic r0, r1, r2`	$r0 \leftarrow r0$ AND NOT $r1$ (bit clear)

Division is normally far slower than multiplication. Think of how you do it. The algorithms for multiplication and division in binary are essentially similar as long multiplication and division for humans. There is no integer division in ARMv7, and for that reason, it is even slower since division is a subroutine.

```
mov     r0, r6
mov     r1, r4
bl      __aeabi_idiv
mov     r3, r0  @ save results of div
bl      __aeabi_idivmod
@ r0 is the result of mod
```

7.5.4 Comparisons, Branching, and Conditional Execution

If statements and loops in high level languages are at core, decisions. In machine language this is broken into two pieces: the instruction that tests values, and then a subsequent instruction that goes somewhere if the first condition is true. You may wonder why assembler does not combine these two operations into a single instruction. This is explained below. For now, consider an if statement in C++:

```
if (x > 3) {
  x = x + 7;
}
```

The above code states that if x is greater than 3, that it should execute what is in the braces. Internally in assembler, the instruction is the reverse: if x is NOT greater than 3, skip the instruction doing the add. The test uses the CMP (compare) instruction, and skipping is done with a branch:

```
        cmp r0, #3      @ compare r0 to the number 3
        ble skip        @ branch if less than or equal to <=
        add r0, #7
skip:
```

The compare instruction subtracts the first parameter minus the second. That is, it computes $r0 - 3$ in this case, and sets 4 flags accordingly:

The Z flag (Zero) is 1 (true) if the result is zero, and 0 (false) if it is not zero.

The N flag (negative) is 1(true) if the result is negative, meaning if the high bit is set, and false otherwise.

The C flag (carry) is 1(true) if the result of an unsigned operation overflows the register.

The V flag (overflow) works the same as the C flag, but for signed operations. So if a number is positive 0x7FFFFFFF and then 1 is added 0x80000000 and changes sign, the V flag is set.

The ARM supports 14 condition codes:

eq	Z=1	equal
ne	Z=0	not equal
cs or hs	C=1	carry set or higher or same
cc or lo	C=0	carry clear or lower than
mi	N=1	minus (negative)
pl	N=0	plus (positive)
vs	V=1	overflow set
vc	V=0	overflow clear
hi	C=1 AND Z=0	higher
ls	C=0 AND Z=1	lower or same
ge	V=Z	greater than or equal to
lt	N=1	less than
gt	N=0	greater than
le	Z=1 OR N=1	equal
		always

Any instruction can be made conditional. At first, we are only considering branch: ble (branch if less than or equal to), bgt (branch if greater than), bne (branch if not equal), beq (branch if equal)

As it turns out, branching is slow. Normally, the CPU is very fast because while it is executing the current instruction it is already decoding the next one. This is called pipelining. Modern computers have deep pipelines and can begin working on multiple instructions in the future, but once a branch is reached, the CPU does not know whether the next instruction will be at location skip, or the add. Branching ruins the pipeline so it is preferable to avoid whenever possible.

The ARM has a clever way of making branching unnecessary for short if statements like the above. They can make any instruction conditional. The program above could be

written without a branch by making the add conditional:

```
cmp r0, #3     @ compare r0 to the number 3
addgt r0, #7
```

Note that the above code will take two clock cycles to execute whether the condition is true or not. It is always executing the add instruction, but the add is conditional and will not add unless $r0 > 3$. At a certain point, it does become worth it to take the hit and branch. For example, consider the if statement:

```
if (x > 3) {
  x = x + 7;
  y = y - x * 3;
  z = 5 * x;
} else {
  x -= 2;
  y *= 3;
}
```

Even if all the variables are in registers, the inside of the curly braces are now at least 4 instructions. Given that a multiply instruction is at least 3 clock cycles, the conditional code is now 1+3+1+3+1+3 = 12 clock cycles. Probably with pipelining, some of the shorter operations can happen during the multiplies. Still, the fact is, regardless of the way the branch goes, it will be executing all that. At a certain point, branches are still useful.

```
cmp r0, #3     @ compare r0 to the number 3
addgt r0, #7
mulgt   r1, r0, #3
subgt   r2, r2, r1
mulgt   r3, r0, #5
suble   r0, #2
mulle   r2, #3
```

7.5.5 Bit Manipulation

Using AND, OR, XOR and NOT, you can surgically insert and remove bits from a word leaving other bits unchanged. This also requires the ability to shift bits left and right (lsl, lsr). In this section we will also discuss arithmetic shift (asl, asr) and rotate right (ror). This section assumes you already know the truth tables for AND, OR, XOR and NOT.

For the purposes of this section we will call the rightmost bit position 0, and the leftmost bit position 31.

```
33322222222221111111110000000000
21098765432109876543210987654321O
01000101010110100011101001011011
```

Suppose you have a number in a register and want to set some bits to 1 (regardless of what they are now, and without affecting any of the other bits). The OR operation does exactly that because 1 OR anything is 1. So all you need is a mask with 1 in the places you want to be set to 1 and 0 everywhere else.

For example consider setting the bits from position 1 to 4 to all 1.

```
10110101 11000111 00110100 10001101 Value
00000000 00000000 00000000 00011110 Mask
```
When ORed, the result has all 1s in the desired bits.

```
10110101 11000111 00110100 10011111 Value
```
To set desired bits to zero, there are two ways. The first is to create a mask with zeros in the desired bits, and AND them

```
10110101 11000111 00110100 10001101 Value
11111111 11111111 11111111 11100001 Mask
```
Since anything AND 0 = 0, the result is zero in the desired bits.

```
10110101 11000111 00110100 10000001 Value
```
On the ARM, the way we can load 32-bit masks is with ldr. The mov instruction only supports numbers from 0 to 256. Because mov is faster, it would be nice to be able to load masks but for AND we need all ones. One trick is to use the bit clear bic instruction which is AND with NOT mask. Suppose you want to get rid of the low 3 bits in a number (the bits marked x)

```
10110101 11000111 00110100 10010xxx
```
In order to do this using AND, you need a mask with all ones, with zeros where the xs are:

```
11111111 11111111 11111111 11111000 mask
```
That would require ldr. Instead, specify the inverse of this mask:

```
00000000 00000000 00000000 00000111 mask
```
Because the mask is 7, it fits within an immediate mode bic instruction, no ldr required.

```
bic r0, #7
```

In order to determine whether a bit is 1 or 0, we need to zero out all the other bits. This means ANDing with a mask with a 1 in the desired location and zero everywhere else. The result will either be zero if the bit is 0, or something other than zero (the number will change according to the bit position). For example, suppose we want to test for the bit marked x in position 5:

```
00010101 01010110 00111111 01x00000
```
This requires the mask:

```
00000000 00000000 00000000 00100000 mask
```
ANDing with the number will result in zero if x is zero, and 32 if x is 1. Here is the ARM assembler code that corresponds:

```
ldr     r1, =0x0000020  @load the mask
ands    r1, r0          @AND with the original value
bne     bitIsSet        @ if the result is not zero, the bit is set
```

Testing bits is very common, and often there is no reason to store the resulting number; in fact it just destroys the value in register r1 in this case. We could store into a different

register, but if there are no available registers, this will just require more work pushing to save the register and popping to get it back. So instead ARM provides an instruction that tests bits without saving the result. It is similar to compare (cmp) but instead of subtracting the two operands it ANDs them. The instruction is called tst. The following example is similar to the above but leaves the mask intact in register r1, so it could be used repeatedly.

```
ldr    r1, =0x0000020  @load the mask
tst    r1, r0          @AND with the original value
bne    bitIsSet        @ if the result is not zero, the bit is set
```

7.5.6 Stack Operations

`push {r4}`
`@ equivalent to:`
`sub sp, #4`
`str r4, [sp]`
	`pop {r4}`
	`@ equivalent to:`
	`ldr r4, [sp]`
	`add sp, #4`

`push {r4,r5}`
`@ equivalent to:`
`sub sp, #8`
`str r4, [sp]`
`str r5, [sp, #4]`
	`pop {r4,r5}`
	`@ equivalent to:`
	`ldr r4, [sp]`
	`ldr r5, [sp, #4]`
	`add sp, #8`

The stack must be 8-byte aligned at all external interfaces (e.g. when calling a function), and it must ALWAYS be 4-byte aligned. (Section 6.2.1, AAPCS)

What is 8-byte alignment?:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4127.html

What is an external interface?:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13786.html

The requirement is to keep the stack on even 8 byte alignment when calling functions, so push registers in pairs when calling any external subroutine. If you're writing your own assembler code without calling any other functions, you just need to keep 4-byte alignment (if you only use push/pop instructions, you'll be fine.)

The stack is used to save the contents of registers so you can use them for whatever you need. Every time a function is called, in order to make room for all the variables in that function, space is allocated on the stack. For example, for the following function:

```
void f() {
  int x = 5;
  int y = 3;
}
```

```
_Z1fv:
  sub sp, sp, #8
  mov r3, #3
  mov r4, #5
  str r3, [sp]          @ y = 3
  str r4, [sp, #4]      @ x = 5
```

```
bx  lr
```

7.5.7 Procedure Call Standards (Section 6.1.1 AAPCS)

The ARM binary standard is a convention for calling functions used by C, C++ and other languages. In general, any function may use r0 to r3 and wipe them out, so if you call a function, parameters are passed using r0 to r3, and you must assume that any of the values in these registers could be destroyed.

For double precision parameter passing and scratch registers are d0 to d7.

As mentioned previously, r4-r8 and double precision registers d8-d15 must be pushed to the stack before they can be used. Any function is going to assume that these registers are left alone, so if they call your function, your job is to make sure that those registers are untouched when you leave. By pushing to the stack, you can do anything you want and then restore the values are the end.

The following functions calls have comments to the right showing which registers are used for each parameter.

```
void f(int a);                        // r0 = a
void f(int a, int b);                 // r0 = a, r1 = b
void f(int a, int b, int c);          // r0 = a, r1 = b, r2 = c
void f(int a, int b, int c, int d); // r0 = a, r1 = b, r2 = c, r3 = d

void f(int a, int b, int c, int d, int e)
// r0 = a, r1 = b, r2 = c, r3 = d, and e is pushed o

void f(int x[], int n);               // r0 = x (address of the array)
                                      // r1 = n
void f(double x);                     // d0 = x
void f(double x, double y);           // d0 = x, d1 = y
```

Passing Parameters By Reference

```
void f(int* a, int b);                 // r0 = a, r1 = b

int x;
f(&x, 3);
```

Mixed parameters

```
void f(int a, double x, int array[]);      // r0 = a, d0 = x, r1 = array

void f(int a, double x, double array[]);  // r0 = a, d0 = x, r1 = array
```

For return values, if the function return an integer or a pointer, it uses r0 If it returns a double, it return d0

7.5.8 Floating Point

Most CPUs have a separate execution unit for floating point, and ARM is no exception. There is a separate section, with its own registers and operations. Values can be copied over from the integer main CPU, and the main CPU is responsible for loading and saving from memory. This means that the floating point computation can be shut down to save power if it is not being used, and also that floating point computation can take place simultaneously with integer computation since it is not the same hardware.

There are 16 double precision registers d0 to d15. There are also 32 single precision registers s0 to s31, and each pair corresponds to one double precision register. Thus it is not possible to store a value in d0 and s0 or s1 at the same time since they use the same bits.

All operations on the ARM have suffixes .f32 for single precision and .f64 for double precision. There is also a vector engine called Neon that can do 128 bit operations, in other words 4 floating point at a time or 2 double. This is not covered in our course, but a section below will eventually describe it.

The following is a list of many (not all) of the operations supported by the floating point processor. All examples are double precision, for single precision just replace every suffix with .f32

vldr.f64 d0, [r0]	load floating point register d0 from memory location at r0
vstr.f64 d0, [r0]	store floating point register d0 to the memory location at r0
vpush.f64 d0	save to stack
vpush.f64 d0,d1	save registers, d0,d1
vpop.f64 d0	restore from stack
vmov.f64 d0, d1	d0 ← d1
vadd.f64 d0, d1, d2	d0 ← d1 + d2
vsub.f64 d0, d1, d2	d0 ← d1 - d2
vmul.f64 d0, d1, d2	d0 ← d1 * d2
vdiv.f64 d0, d1, d2	d0 ← d1 / d2
vmla.f64 d0, d1, d2	d0 ← d0 + d1 * d2 multiply and add
vabs.f64 d0, d1	d0 ← abs(d1)
vsqrt.f64 d0, d1	d0 ← sqrt(d1) (does not handle special cases like nan)
vneg.f64 d0, d1	d0 ← -d1 (negate)
vmov s15, r0	s15 = float(r0) convert to float
vcvt.f64.s32 d0, s15	d0 = double(s15)
vcmp.f64 d0, d1	compare floating point d0 to d1

Kruger's rule: NEVER USE FLOAT!!! (for computation). You will regret it almost every time. Exceptions:

1. For graphics, OpenGL uses float (because the screen is only 2k 4k 8k so coordinates don't have to be that accurate

2. If you are storing an array of millions of data points, and known accuracy ¡ 1 part in 10 million, no reason to store more useless digits. But do the computation in double.

For single precision float (untested!)

```
vldr.f32 s0, [r0]      @load floating point register from memory location
                       @at regular register
vpush.f32 {s0, s1}
vadd.f32  s0, s1, s2   @s0 = s1 + s2
vsub.f32  s0, s1, s2   @s0 = s1 - s2
vmul.f32  s0, s1, s2   @s0 = s1 * s2
vdiv.f32  s0, s1, s2   @d0 = d1 / d2
vmla.f32  s0, s1, s2   @d0 += d1 * d2
```

7.5.9 Vector Operations

[TBD]

7.6 RISC vs. CISC

CISC stands for Complex Instruction Set Computer. RISC stands for Reduced Instruction Set Computer.

Computers started simply. As the number of transistors grew, designers added features such as more registers. CPU designers would imagine scenarios in which a particular sequence of operations could be turned into many fewer by adding more complicated instructions. The goal of a CPU designer is to try to compute as fast as possible. Designers can make instructions faster with faster circuitry, but at a particular speed, they always want to do the best they can. The CISC approach is to design instructions that do more, and have a rich set of modes for thes instructions. For example, an add instruction on the Intel 80x86 can add register to register, or register to memory, or memory indexed by a register, ... many different ways of getting the data to add. The RISC approach is to streamline and have fewer kinds of instructions but make them faster. For example, there are only two kinds of values that can be added on ARMv8, adding values from registers or a number built into the instruction (immediate mode). The ARM has more registers (31) so that more values can be kept on chip and there is less need to load and store data.

John Cocke and a team at IBM are credited with inventing the first RISC computer at IBM in the 1970s. They tried to simplify the logic so that each instruction could be executed faster. His team's approach was to remove instructions that weren't frequently used and try to focus on speeding up the average computation rather than focusing on many special cases.

In the late 80s the DEC Vax CPU was a typical CISC machine. It had instructions to add and multiply. These could be used to evaluate a polynomial ax2+bx+c. But when designing the Vax, the team decided to add a function to evaluate a polynomial directly. The instruction was more complicated since it did many adds and multiplies. It was almost never used since no compilers knew how to turn high level code into that instruction. And in fact, the instruction ran slower than the large program composed of multiplies and adds. This example is often cited as why CISC is slower than RISC.

The RISC approach did not immediately win. Each time a new CISC or RISC design is built, it is faster than older models because the circuits are newer. Often, CPUs win for other reasons. Intel stayed ahead for years despite a highly complicated CISC architecture because customers wanted to be able to run existing software (compatibility) and because they made so much money, they were able to keep building new fabs and keep building processors using ever-smaller and faster circuits. Intel paid the performance penalty of all the crazy instructions in the 80x86 family because customers wanted compatibility more than anything else. Sun Microsystems was highly successful for a while with a RISC design, but in the end Intel outperformed their CPUs by continuing to shrink circuits, and put more circuits onto a chip while increasing clock speed.

Today, the Intel architecture (80x86) is a CISC design, but it has changed somewhat. Some of the instructions are no longer supported in hardware. They are software-emulated. In other words, rather than wasting silicon on building circuits to decode rarely-used instructions, they came up with a way to run a sequence of simpler instructions. This means that the rarely-used instructions run slower, but no one cares.

On the other hand a modern RISC architecture like ARM is far simpler than the 80x86, and has fewer instructions, but in certain cases, performance can be accelerated so much that even RISC processors use the CISC concept. Examples include instructions to accelerate complex operations such as encryption, and data compression. Even though these do not happen as often as adding and multiplying, they are so slow that having operations specifically designed to make them faster can overall speed up the computer.

Intel 80x86 has 1503 instruction from AAA So we have seen both sides learning from the mistakes of the past. CPUs are getting simpler so as to speed the execution of common operations like add and multiply, they are getting more registers

7.7 Registers

See: Section A2.3 Reference Manual

The address in the pc register is the next PCSR (Program Current Status Register) (Section B1.3.3 and A2.4 Reference Manual) Contains a lot of things but for our course, focusing on:

Z Zero condition flag. Was the result of the last computation zero or not?

Table 7.1: Registers

register	Abbreviation	Purpose
r0...r15		general registers
r9		platform register
r12	ip	linker scratch register
r13	lr	link register
r14	sp	stack pointer
r15	pc	program counter

N Negative condition flag. Was the result of the last computation negative or not?

C Carry condition flag. Did the last computation result in carry (e.g. unsigned addition)?

Floating point Registers s0..31 d0..15

Each d register has two registers as halves. d0 = [s0,s1] d1 =[s2,s3] You cannot write into s0 without destroying the value in d0, and vice versa.

7.8 Memory Modes

Load and store instructions can use the following modes:

```
ldr rd, =constant     @really internally this loads at some offset relative to
                      @the PC
ldr  rd, [rm]          @rd $\larrow$ 32-bit memory at location rm
ldr  rd, [rm, #offset] @rd $\larrow$ 32-bit memory at location rm + offset
ldr  rd, [rm, #4]!     @rm $\larrow$ rm +4 then rd $\larrow$ 32-bit memory at
   location rm
ldr  rd, [rm], #4      @rd $\larrow$ 32-bit memory at location rm, then rm +=
   4
str  rd, [rm]          @memory location rm $\larrow$ rd (32-bit)
ldrb rd, [rm]          @rd $\larrow$ 8 bits (one byte) at location rm
strb rd, [rm]          @memory location rm $\larrow$ low 8 bits from rd
```

7.8.1 Examples of using memory modes

For a function reading from an array, the C++ convention is usually to pass the name of the array (which is the location of/pointer to the first element) and the length of the array. For example:

```
int a[3] = {9, 5, 4};
int y = sum(a, 3);
```

On the ARM, the calling convention for integers and pointers is to pass the first parameter as r0 and the second as r1. This means r0 points to the memory, and r1 = 3.

The arm assembler should go in a loop repeating r1 times. Each time, it should load the element pointed to by r0, add it onto a sum, then at the end, make sure that r0 has the answer. The resulting code is:

```
        .global _Z3sumPii
_Z3sumPii:
        mov r2, #0
1:
        ldr  r3, [r0]      @ load in each number into r3
        add  r2, r2, r3    @ add onto r2
        subs r1, #1        @count down
        bne  1b            @keep going n times (until zero)
        // by the time the code gets here, r1 = 0
        // every time it is NOT zero we jump back up to 1:
        mov  r0, r2        @answer is expected in r0 (convention)
        bx   lr            @return to caller
```

8. Optimization

8.1 How do computers get faster?

8.2 What optimizations can programmers make?

8.3 Compilers

To know how to make your code fast, you must know what the compiler knows how to do, and most importantly, what it still cannot do. This is a constantly changing target because very smart people are constantly trying to make compilers generate better code. Their effort is like a silent boost in the performance of your computer – if you recompile old code and know how to use the optimizer.

Instruction	Description
g++ −g	Compile with debugging
g++ −O2	Compile with optimization. Code is far more efficient
g++ −O2	
g++ −g −O2	Compile with maximum normal optimization. If combined with -g, as you step through the program in the debugger it will no longer seem to step line by line because some lines of your code will just have disappeared. The optimizer is allowed to do whatever it wants as long as a legal program would still do the same thing. Of course if you have an error, the compiler could change the program, but that is your fault.
g++ −O3	Compile and use some "dangerous" assumptions that might change the results. I rarely see a difference between -O2 and -O3 anyway. For really optimizing floating point, you have to use your own brain still.
g++ -O2 -march=native	On the so-called "Intel" architecture (which include AMD or anyone making a "clone", and unfairly, let's give credit, some of those features were invented by AMD notably x86_64 the 64-bit extensions), compile so that the code is only good on your computer. This means that if you tried to run on an older machine like an 80386 or 80486, the code would not work, since some of the assembler instructions don't exist on those old machines. But the code does in certain cases run a lot faster. This will not make your typical counting loop run twice as fast. It's more for edge cases. I have seen about 30-40% maximum improvement, but not on most code. This will not use specialized instructions like those to make video compression or encryption more efficient. For those, people still hand-write assembler, and those can make tasks like compressing video go a lot faster, perhaps 2x or 3x.

8.4 Integer Optimization

If the compiler sees	turns into	Additional Comments
x*0	0	anything * 0 is zero, so no need to compute.
x*1	x	No need to multiply
x+0	x	No need to add
x−x		gcc does not seem capable of recognizing that this is zero.
x*2	x+x or x<<1	addition is faster than multiplication. If you examine the results from lab 7 and lab 8, multiply probably takes from 3-4 clock cycles whereas addition takes 1. Remember however that with a pipelined processor, it may not be quite as bad because you may be able to do other things while waiting for the multiply to complete.
$x * 2^n$	x << n	Left shift takes one clock. Multiplying in binary is like multiplying in decimal. Adding a zero on the end multiplies by the base. For example the decimal number 26 becomes 260 with a zero added which is 26 * 10. Similarly, the number 1001 (9) shifted left by 2 becomes 100100 which is 36 (which is multiplied by $2^2 = 4$).
$x/2^n$	x >> n	Right shift takes one clock
$x\%2^n$	x & 2^{n-1}	modulo 2^n is equivalent to the last n bits
`int sum = 0; for (int i = 0; i < n; i++) sum += i;`	**int** sum = n * (n+1) / 2;	They know the loop has a closed form equation

Note that in general, dead code is not generated. So if the compiler can determine that something is not used (the results of a loop, a variable) nothing is generated. The following code literally turns into nothing if optimization is turned on, as long as sumsq is not printed. In every case, the compiler knows exactly what is happening and it knows that removing the code does not change the outcome of the program because there is no outcome.

```
int f(int x) { return x*x; }

int main() {
  if (false) {
    cout << "This will never happen\n";
    for (int i = 0; i < 10; i++)
      cout << i;
```

```
  }
  const int n = 1000000000;
  for (int i = 0; i < n; i++)
    ;
  int sum = 0;
  for (int i = 0; i < n; i++)
    sum += i;
  int sumsq = sum*sum;

  for (int i = 0; i < n; i++)
    sum += f(i);
}
```

In this second example, nothing can be eliminated. The first loop executes f(i) n times. Since the compiler does not have the source code for the function f, it has to assume that something happens there. The sum in the second loop is squared and then the result is printed so the loop cannot be removed.

```
void f(int x);

int main() {
  const int n = 1000000000;
  for (int i = 0; i < n; i++)
    f(i); // the compiler does not know what f does
  int sum = 0;
  for (int i = 0; i < n; i++)
    sum += i;
  int sumsq = sum*sum;
  cout << sumsq;
}
```

8.5 Inline Optimization

C++ like many object-oriented languages has lots of little function calls. Calling functions takes a lot of time, so the optimizer will remove the call and inject the assembler of the function whenever possible. The first requirement is that the code is known. The optimizer isn't clairvoyant, so if you don't tell it the code, it cannot inline it.

Suppose you have a simple one-line function:

```
double f(double x) { return x*x; }

int main() {
  cout << f(3) << '\n';
}
```

In the above case, the optimizer can clearly determine at compile time what f(3) is. It

replaces the function call by x*x, then inserts 3 for x, which is 3*3. Then it evaluates the 9. The result is that the computation is completely eliminated and all that is left is printing the result:

```
int main() {
  cout << 9 << '\n';
}
```

Of course, it's not always known at compile time what the values are. In the following code the programmer reads in the variable from the keyboard.

```
int f(int x) { return x*x; }

int main() {
  int x;
  cin >> x;
  cout << f(x) << '\n';
}
```

In the second case, since the compiler does not know the value of x, it cannot precompute it, but it does not have to call a function. The call f(x) is replaced by x*x which is significantly faster. Let's take a look at the assembler generated for the call:

```
_Z1fi:
    mul r0, r0, r0
    bx  lr

main: ...
    ldr r0, [sp, #8]
    bl  _Z1fi
```

The total instruction executed is 4, which itself is a testament to how well designed ARM is. Older CPU designs using the stack would require more instructions and much slower. Compare this to the code executed for the inline code:

```
    ldr r0, [sp, #8]
    mul r0, r0, r0
```

There is less code generated, and less code executed, a double win. The only cost is that it takes extra time to figure out what to generate, so compilation is a bit slower.

But now, suppose the function f is specified as a function prototype, and is written elsewhere. In this case, no matter how smart the optimizer is, it cannot determine what code to inject, so it must execute the call to the function:

```
int f(int x);

int main() {
  cout << f(3) << '\n';
}
```

In some cases, however, it can actually be slower to inline code. The first thing to understand is that the savings is the time it takes to pass parameters, call a subroutine and return. So if the function contains a loop that executes many times, then any savings

will be a tiny percentage. A large subroutine will also have many instructions, and all those will have to be replicated every time the function is called. This means every time the function is called, the program will grow. For a function called in many places, the increased use of memory can overflow cache, resulting in lower performance. The following programs shows an example where a loop means that the time taken to call the function is less than 1 millionth the time. This function will typically not be inlined by the compiler – C++ will simply not do it.

```
double f(int n) {
  double sum = 0;
  for (int i = 0; i < n; i++)
    sum += exp(-i);
  return sum;
}
```

Usually, the compiler is quite right about not inlining. It is better at bookkeeping than any human, and it know what the cost is. However, there are a few exceptions. If you know that the function is only called once, then it is always better to inline than not. And even if there is a loop, if inlining would let some of the computation be constant and eliminated, that can be a win. Furthermore, if function f calls g which calls h, and all can be inlined, and if enough of the computation boils out, it can be a win. Usually however, the compiler knows best, and manually writing the code yourself will not be much of a win except in special circumstances.

8.6 Bit Twiddling

There is a special class of problem where clever tricks result in faster solutions. Over the years, some brilliant people have come up with clever solutions to compute specific problems that come up over and over again. Many of these are summarized in the website graphics.stanford.edu:
https://graphics.stanford.edu/~seander/bithacks.html

We are just going to look at a representative sample, and then mention that even better than these, sometimes a CPU designer will come up with hardware to compute results even faster.

Problem	Obvious way	Faster

toggle between n and 0	`if (n == 5)`
	`  n = 0;`
	`else`
	`  n = 5;`
		`n = 5 - n;`

Find power of 2 greater than *n*	`int power2greater(int n) {}`
	`  for (int i = 0, j = 1; i < 31;`
	`  i++, j <<= 1)`
	`  if (j > n)`
	`  return j;`
	`  return 0; // if there is no`
	`  bigger power of 2, too bad!`
	`}`
		`int power2greater(int n) {}`
		`  n = n | (n >> 16);`
		`  n = n | (n >> 8);`
		`  n = n | (n >> 4);`
		`  n = n | (n >> 2);`
		`  n = n | (n >> 1);`
		`  return n + 1;`

8.7 Floating Point Optimizations

In general, compilers are extremely conservative about floating point optimization because of roundoff error. If you look back at the section on floating point, you can see that associativity does not hold. That means changing the order of operations changes the answer subtly. Compilers therefore leave these kind of optimizations to the humans in general. There are some compilers which have flags for "aggressive" optimization of floating point which will allow such reorderings (Intel?) but I cannot find any such options in gcc.

Instead of	Write	Comments
a*power(x, 2) + b * x + c	((a * x + b) * x + c	Horner's form
c1 * x * c2	(c1*c2) * x	Group Constants
c1 * x + c2 * x	(c1+c2) * x	Undistribute manually, compiler won't
x / a + y / a + z / a	(1/a) * (x + y + z)	precompute inverse
abs(x*x + y*y)	x*x + y*y	compiler does not realize result is guaranteed to be positive
sqrt(x*x + y*y)	sqrt(x*x + y*y)	You can remove internal tests for negative (only in assembler)
`double xsq = x * x;` `sqrt(xsq) * sqrt(xsq);`	`double xsq = x*x`	Simplify algebra, compiler will not
(x + y)/2	(x + y) * 0.5	Precompute inverse

8.8 References

https://docs.oracle.com/cd/E19422-01/819-3693/ncg_goldberg.html

9. Computer Memory and Storage Overview

9.1 Introduction

Memory is an integral part of computation. There is almost no practical purpose in performing computations without memory to store the results in. This chapter will deep-dive into how different types of memory work, their advantages and disadvantages, as well as the evolution of memory overtime.

Memory acts as the storage system for anything the processor needs to perform computations, and the results of those computations. At the base level, memory is the generalization of *memory cells*, which store a value of either 0 (low) or 1 (high). Different combinations of these 0s and 1s make up any sort of program, file, etc. stored within the computer. In the next section, we will look into the different types of memory, specifically how their access procedures affect performance.

Before we look at the different types of memories, we will focus on three hardware components that make use of these different types. These are caches, random-access memory (RAM), and disks.

1. Cache - Often referred to as a high-level memory, the cache is a small storage component generally found on the processor. Cache design focuses minimizing the hit time, where a hit is an access to memory for reading or writing. To keep a low hit time, caches are often sized on the order of kilobytes to megabytes. Additionally, caches are built directly on the processor so that there is minimal time spent travelling with a bus cable to access memory. The lack of a bus and small storage size means bandwidth is generally a non-issue. Caches often hold the instruction, source, and destination addresses and values needed by the arithmetic-logic unit (ALU) when it executes instructions.

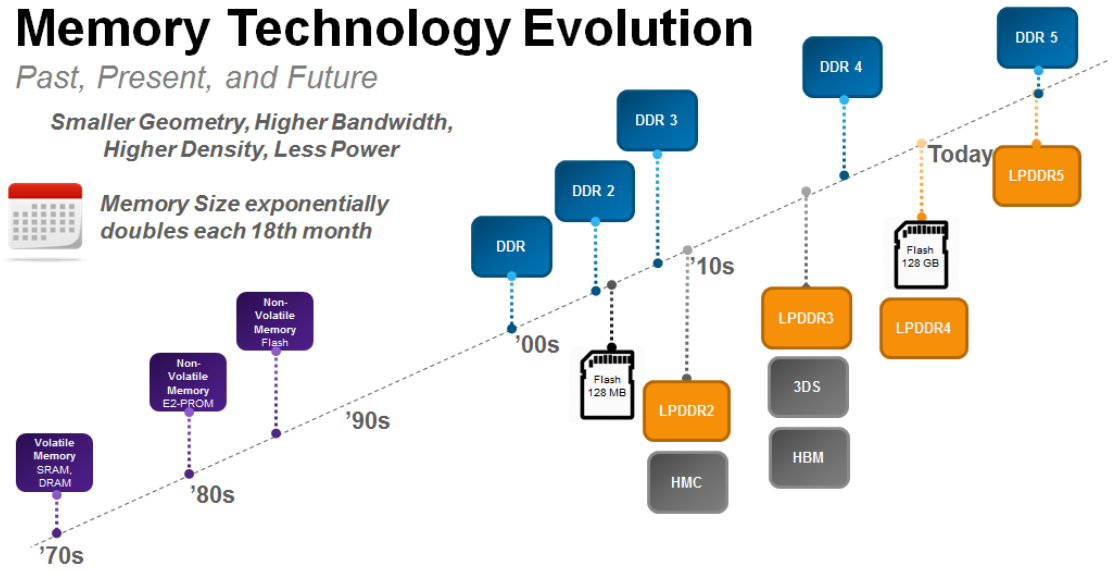
2. RAM - The cache's small size often means it needs to pull instructions and operands from a larger memory bank. Enter random-access memory, or RAM. It is substantially larger than the cache, in the range of gigabytes. RAM can be both *volatile* and *non-volatile*. Volatility in this context means the data is saved even if there is no power. Most modern RAM is based upon volatile techniques, meaning when the computer is turned off, anything that was stored in RAM is lost. As RAM is much larger than the cache, it is also much slower to search through for the sake of finding data. Additionally, RAM modules are located on the CPU board, not on

the processor itself. This means that there is a non-negligible access time required to reach the RAM modules in order to read data. Despite the warnings of the last two sentences, RAM is still ridiculously fast compared to the disk.

3. Disk - Throughout recent history, disks have been magnetic platters within a computer which can be read and written to. You may know them better as hard drives. Disks have immensely more storage, nowadays reaching into the terabytes. However, it takes much longer for these drives to be read and written into. They are connected to the CPU board using a memory bus (cable), meaning travel time and bandwidth take a significant toll on hit time. Additionally, the spinning platter needs physically orient to where the data is stored, which takes noticeably more time than the cache or RAM need to access internal data.

As you may have noticed, there are trade offs in memory hardware between size and speed. As you increase the size of memory, it becomes slower to access data. Computer architectures typically have multiple kinds of memory ranging from cache to disk as a way of optimizing the average speed of memory.

9.2 History of Memory



9.2.1 Old Fashioned Technology

Magnetic Core

Ancient technology, very slow compared to RAM but no moving parts. Can survive the radiation of atomic bombs so was used to record data on computers near atomic tests. The computers stopped working, but the data in the core remained.

Floppy Disk

Older and much slower, with a head that touches the surface. Floppies are pretty useless today.

Magnetic Tape

Slow access because you must somehow get to the right place on the tape. Today magnetic tape is still used for the highest volumes of data to be stored. Probably more reliable than hard drives if kept without being used for years, but no technologies today have a great shelf life, which is a real problem. For the largest volume of data, many tapes are handled by a robot that reaches the tape and puts it in the reader. This arrangement, called a tape library is even slower than tape, since it takes multiple seconds to access the correct tape and load it in a reader before the tape itself can be searched for the right place. Tape libraries are primarily for archival (infrequent access) storage.

Optical Disk

This includes CD-ROM, DVD, Blu-Ray, and subsequent implementations that were higher density but used only for data storage, not videos. Writeable disks require power and are a bit slow because a laser has to heat up the surface.

9.2.2 Modern Memory

Dynamic RAM (DRAM)

Only 2 transistors per bit, but much more complicated. Uses capacitors to store charge, and must be read and written every few milliseconds in order not to lose memory.

Static RAM (SRAM)

A circuit designed to retain a bit of data. SRAM is the fastest kind but uses 6 transistors per bit, so for main memory DRAM is more efficient.

SDRAM and DDR SDRAM

Synchronous DRAM (or anything) is a lot faster than asynchronous. With an asynchronous circuit, we have to wait for an acknowledge signal, and that takes time to propagate (latency) proportional to distance. Most DRAM today (DDR, DDR2, DDR3, DDR4, DDR5 coming up and in the graphics world GDDR5 and HBR) is synchronous. The speed of memory is the burst speed that data can be delivered. There is also the latency (delay) for reading or writing a new row and reaching the correct column.

In general, as speeds have increased for memory, the number of clocks for RAS and CAS delays has increased. Overall, everything has gotten faster of course, just not by as great a margin as the burst speed.

All things being equal, lower RAS and CAS delay numbers should mean faster access to memory. But all things are not equal. For a detailed explanation, see

The first generation of the current kind of SDRAM. DDR4 is the current generation of DDR memory, a lot faster, and with 8x preread buffer. Reading 8 locations at once is very fast IF AND ONLY IF you are reading sequentially. So when your program reads from memory jumping around, it's very slow today because we have no way of speeding that up. DDR5 is the next generation of DDR memory, not out yet. Note that even if manufacturers can design one, it takes 18 months once they do to get the whole ecosystem working together (motherboards have to be designed to use it).

GDDR SDRAM

Graphics DDR SDRAM (GDDR SDRAM) is a type of SDRAM designed specifically for graphics processing units (GPUs). It tends to have very high bandwidth because graphics requires a lot of data, and is very much oriented towards sequential reads.

HBM

A competing design for graphics memory that has higher performance than GDDR but is more expensive. Also, since it came later, GDDR was available on many grpahics cards already, and not only would it cost more money, it would delay the production of cards. Sometimes, the decisions made by manufacturers are not about performance but business considerations.

Hard Drives

Hard drives are a crazy technology. Imagine a disk spinning so fast that the wind it picks up is sufficient to carry a tiny magnetic reader with wings flying over the surface. Jarring the device, or a tiny spec of dirt can cause the magnetic read/write head to crash, gouging the surface and killing your data. Yet that is exactly what we have used for 30 years, and as long as you don't drop them or unseal them and let in dirt, they are actually amazingly reliable.

Hard drives are the most cost effective for non-volatile, large data storage but are slower than SSDs.

Flash

Also called NAND-Flash because it is composed of NAND gates. Flash memory is a lot faster than hard drives, at least 10x but it has a limited life. Too many reads can burn out the drive. Drive controllers usually carefully balance out the reads to all parts of the drive so that drives last for a few years. Hard drives may be slower but they still have advantages…

9.2.3 The Future of Memory

3D XPoint

3D XPoint memory is a totally new approach which is radical in that it could eventually replace RAM if it can be made as fast as current RAM (and there is reason to believe it can be), and it is byte addressable. It is also non-volatile, so it potentially could

revolutionize computing if it could be made as fast as RAM since there could be only a single level of memory in the computer. At the moment, the fastest XPoint memory is significantly faster than NAND flash, but not as fast as RAM.

9.3 General Characteristics of Memory

Some devices allow access of memory at fine-grained levels. RAM (both static and dynamic) allow accessing memory by an address. In order to increase bandwidth, memory may be organized into 64-bit data bus, or wide (for video RAM, currently, 256bit, 392bit, exist).

DDR RAM is designed to give faster access to sequential accesses. When a memory location is read, the hardware reads 4 or 8 in a burst. This means that if the next memory location is requested, it is already available, and dramatically increases the speed of memory.

Interleaving is used to provide more bandwidth. Today's PCs use two banks of memory, each 64-bit, so that when one location is requested, two simultaneous memories are read, each with 4-burst, for a total of 8 locations. This is how memory bandwidth can keep up with the processor.

It is slower to access memory non-sequentially as we can see in benchmarks when we try to read from non-sequential locations. But at least it allows any location can be read or written.

Hard drives, cd-roms and other spinning media are so slow by comparison, that it is not efficient to access individual bytes or words. When reading from these kind of drives, or from flash memory SSDs, the computer reads and writes data a block at a time. This fundamentally changes how we access these kinds of memory. It takes time to load a block from disk, so naturally we try to group all the data we can into a single block. All performance on hard drives is based around trying to keep data in as few blocks as possible, and trying to keep those blocks together on disk because moving the heads from one track to another takes milliseconds, a million times slower than RAM. By reading a lot of data at a time from each block, the data rate can be kept very high despite the lags for switching tracks, as long as data is read or written sequentially.

Currently, block devices are connected via interfaces including USB 2.0, USB 3.0 and serial ATA, which require few wires and are pretty fast, or a bus like PCI-e, which is even faster and requires a card plugged into a PC. For details on the connections between mass storage and computers, see the section on busses below.

9.3.1 Block Devices

Hard drives, floppy drives, and NAND-flash are all built in a way that makes accessing a single byte difficult. Accordingly, they are organized into much larger blocks. When

the computer reads from these types of memory, it reads an entire block at a time.

Hard drives are organized into sectors and tracks. The speed with which data is read/written depends on the speed at which the hard drive rotates multiplied by the density of the data on the track. Hard drives also have multiple platters with multiple heads so that multiple bits can be transferred in parallel. A read head must move to the correct track on the disk, and this physical motion takes multiple milliseconds. On most hard drives today, block size is 4k.

Unix uses files to give access to block devices. We can open a file and read or write bytes to it. The kernel calls open(), read(), write() and close() are used to access files. As we will see in experiments, reading small amounts is not efficient. Knowing that hard drives are designed to access a block at a time, reading or writing in sizes that are an integer multiple of the block size is most efficient. For example, given a block size of 4k, writing 4k or 8k is efficient. Writing 32k may be more efficient because it does take time to enter and leave the kernel, but there are diminishing returns. A buffer size of 1 million bytes might even be slower because writing will not begin until the program fills the buffer and calls write. For faster speed, it might be possible to use multiple threads, and work to prepare the next block while the current one is being written. However, no matter what approach is taken, the fundamental limit of speed

9.3.2 Busses and Interfaces

This self-study section is just designed to get you to learn about the range of technologies in use to connect two sides using a bus. Busses are shared wires used to transmit data. Some of these interfaces are designed to communicate at high speeds between extremely fast devices. Others are low speed, designed for simplicity and ease of building. SPI for example uses 4 wires and is easy to build connecting a Raspberry Pi or Arduino to peripherals. PCI-express is a very high speed interface that lets a desktop PC communicate with the devices on cards such as a video

9.4 Byte Alignment

Byte Alignment Computer memory is organized into words. The Raspberry Pi has a 64-bit bus, meaning it can read and write 64 bits (8 bytes) at a time. These memory locations are addressed as multiples of 8 as shown in the diagram below. It takes a single memory cycle to read or write memory from a memory location starting with a multiple of 8. Reads and writes starting with addresses that are not a multiple of 8 are legal. However, they are slower because the operation requires reading from two adjacent memory locations, and therefore takes longer.

In the table below, a read starting with location is highlighted in yellow, it reads all 8 bytes in one operation.

Address	+0	+1	+2	+3	+4	+5	+6	+7
00000000								
00000008								
00000010								
00000018								
00000020								

A second read starting at location 9 is not aligned, and therefore reads 7 bytes first, then 1 byte from the next location shown in green. This is twice as slow as the first. For this reason, most compilers will store values on even-aligned memory, where the alignment is defined by the width of the memory system (not the width of the registers, which in this case is half as much).

For this reason, in C++ when we declare a structure or class, each member is stored on an even alignment so it can be read in a single operation:

```
class Vehicle {
private:
  char      colorCode[2];
  uint32_t  numWheels;
  bool      amph;
  double    weight;
  char      mfg;
};
```

Relative location of fields in Vehicle:

0	colorCode[0]	colorCode[1]			numWheels			
8	amph							
16	weight							
24	mfg							

This object is 32 bytes with 16 bytes of wasted space, shown in blue.

By reordering the elements in your object, you can store the values more efficiently. C++ does not optimize the order for you because you may want to specify the order because the other side of a communication may have a defined order.

```
class Vehicle {
 private:
  char colorCode[2];
  bool amph;
  char manufacturer;
  uint32_t numWheels;
  double weight;
};
```

0	colorCode[0]	colorCode[1]	amph	mfg	numWheels
8	weight				

The second version of the object is 16 bytes

10. Interrupts

There are two categories of interrupts.

- Hardware interrupts that interrupt normal execution and cause a function to execute
- Software interrupts used to access an operating system

An interrupt is like a function call that is triggered by hardware. This is a fast way to get the computer to respond to something

Examples of incoming events:

- User presses a key
- Disk drive has read a block and is ready to transfer data
- A message is coming in on a serial port
- A message is coming in on a USB serial line

All these events have in common is that they are asynchronous

- We don't know when the event will happen
- We don't want to wait for it
- The computer should process data, but when the event occurs we need to respond quickly
- Examples of time sensitive interrupts
  - A key is pressed. The user wants to see that letter on the screen
  - A stream of bytes comes in on the network interface. The computer must respond.
  - A hard drive reads data. It needs to get rid of that data so it can get the next block.

The difference is that some events are higher priority than others. Devices like ethernet and disks are very high speed, and the computer may have to respond or risk losing data. (Actually, high speed devices typically write their results into memory directly, giving the computer more time to respond.)

The keystroke does not require the same speed but the user wants to see the letter they typed on the screen without lag.

For all these events, the solution is that when hardware does something, we want to wake up the computer. That's an interrupt.

An interrupt is a hardware-triggered call to a function

When the interrupt is done, it returns control to whatever was running

An interrupt can be interrupted, but only by one with higher priority.

On the Arduino, any digital pin can be intercepted using the function attachInterrupt:

```
attachInterrupt(digitalPinToInterrupt(2),buttonPressed1,RISING);
```

10.1 Software Traps

The second way interrupts are used is to make controlled function calls into the operating system. The operating system is a program that controls your computer. It can do anything, so we cannot have programmers calling pieces of it trying to achieve evil purposes. Instead of calling a function, we request a service using a software trap.

This is like calling a function except:

1. The caller does not say what address to execute. They can only request a number, like 'execute function 1". This looks up the address in a vector that is under the operating system's control.
2. When calling a software trap, the privilege level can be raised. So the call to the operating system can execute code that the user does not have permission to do. This is analogous to asking someone in a restaurant to cook something for you. You want to eat, you have the right to ask, but you cannot just go into the kitchen and do it yourself.

10.2 Exercises

1. Look up the Unix signal function and write a C program to trap when control C is pressed on the command line. Note this means that in order to kill the program you will have to get into another shell, view processes with the command like "ps -afe — grep yourprogram" and then kill -15 ¡processid¿

2. Connect a pushbutton on the Arduino. Write a program to print out numbers 10 per second counting up. Every time the pushbutton is pressed, reset the number to 0 using an interrupt.

10.3 Further Reading

https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/ https://linux-kernel-labs.github.io/refs/heads/master/lectures/interrupts.html https://peterdn.com/post/2019/02/03/hello-world-in-arm-assembly/

11. A/D and D/A Conversion

11.1 Introduction

Computers store discrete numbers digitally. This makes them resistant to losing data because each bit is either a 1 or 0 and errors can be detected. The world outside the computer is analog however and computers, particularly embedded systems must read in analog values so they can be stored. Sensors attached to computers typically turn the measurements into voltage, and the voltage is measured by an analog to digital converter (ADC). In the opposite direction, for a computer to change a voltage requires a digital to analog converter (DAC). Uses of DACs include audio, for example.

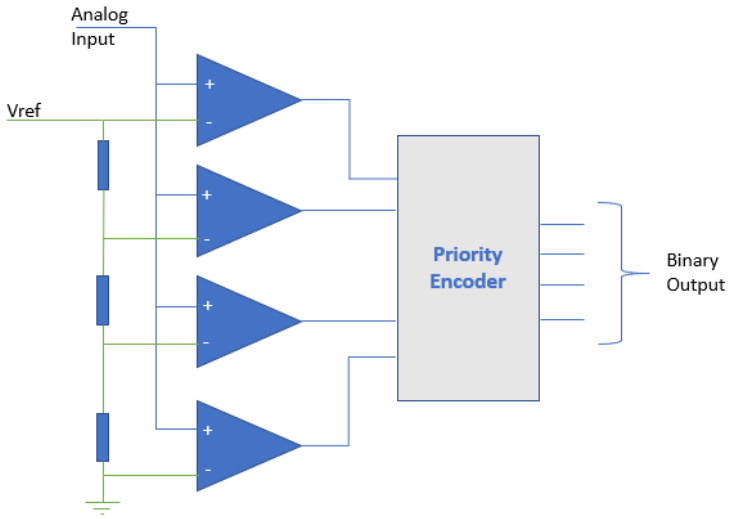
11.2 Different Kinds of AD/C

An ADC converts a continuous-time and continuous-amplitude analog signal to a discrete-time and discrete-amplitude digital signal. AD/C circuits use one or more op-amps to read in the voltage (or current) and measure it.

11.3 Flash Conversion

The first, most direct but most expensive method is called the flash converter. This uses a voltage divider and 2^n op-amps in parallel to achieve very fast conversion. However, the number of op-amps means that this kind of circuit is power hungry and the practical limit is 256 op-amps meaning 8-bit accuracy.

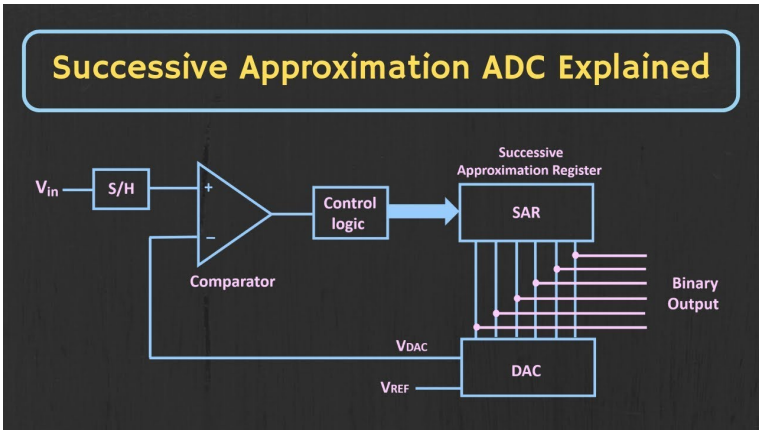
Examples of this kind of conversion include digital oscilloscopes. The scopes in the B123 lab use a chip that captures 1G sample/sec (1Gs/s), at an accuracy of 8 bits. For enough money, performance can go even higher. High end oscilloscopes capture 30 to 60Gs/s but these can cost $10k to $100k. RADAR on military aircraft is even more expensive but can capture and analyze billions of samples per second in realtime.



11.4 Successive Approximation (SAR)

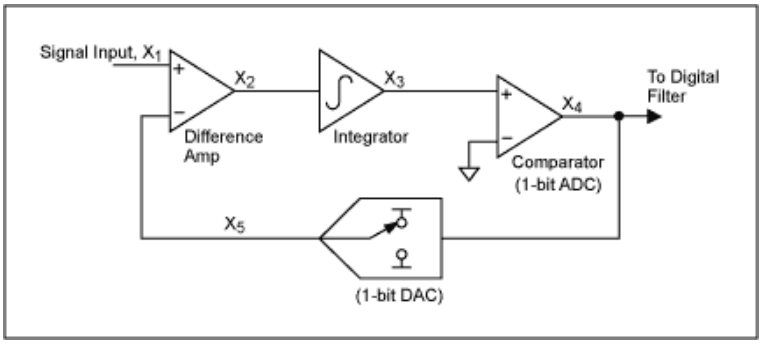
A successive-approximation ADC uses a comparator and a binary search to successively narrow a range that contains the input voltage. At each successive step, the converter compares the input voltage to the output of an internal digital to analog converter which initially represents the midpoint of the allowed input voltage range. At each step in this process, the approximation is stored in a successive approximation register (SAR) and the output of the digital to analog converter is updated for a comparison over a narrower range.

Each bit of answer takes a unit of time so this type of A/D conversion is comparatively slow, but it can certainly do audio sampling (44khz) and probably higher.



11.5 Sigma-Delta

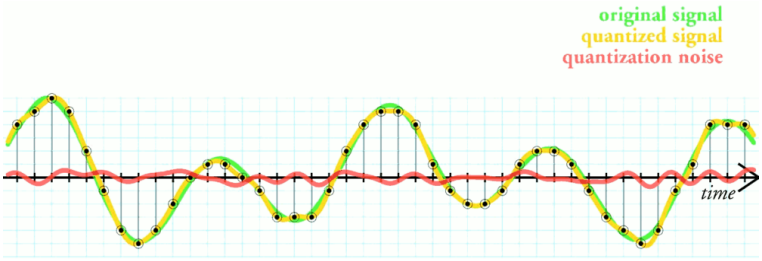
A sigma-delta (or delta-sigma) ADC oversamples the desired signal by a large factor and filters the desired signal band. Generally, a smaller number of bits than required are converted using a Flash ADC after the filter. The resulting signal, along with the error generated by the discrete levels of the Flash, is fed back and subtracted from the input to the filter. This negative feedback has the effect of noise shaping the error due to the Flash so that it does not appear in the desired signal frequencies. A digital filter (decimation filter) follows the ADC which reduces the sampling rate, filters off unwanted noise signal and increases the resolution of the output (sigma-delta modulation, also called delta-sigma modulation). [Wikipedia]



11.6 Errors

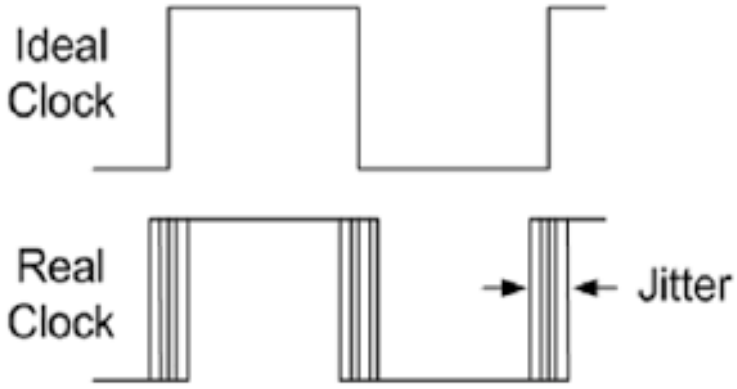
Every ADC has a number of bits accuracy. Some like successive approximation may have more or less accuracy depending on the amount of time used, but there is a maximum possible accuracy of the circuit. For an 8 bit ADC there are 256 different levels of voltage.

This means that for a value halfway between two levels, the maximum error is half of one level. This is called quantization error.

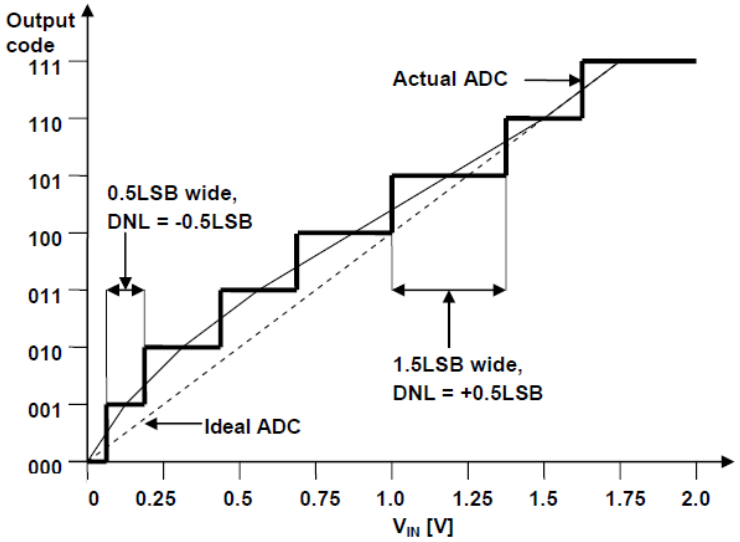


For example, if measuring a range from 0 to 5V, and there are 8 bits, or 256 voltage levels, then the difference between 0 and 1 is $\frac{5V}{256} = 0.019V$. A value of 0.085V is halfway between 0V and 0.019V and is therefore the maximum error.

Jitter is the error due to a clock that is not perfectly accurate.



Nonlinearity is due to the fact that the op-amps are not perfect, and therefore the voltage between 0 and 1 is not quite the same as 1 to 2 or from 1022 to 1023.



In general, A/D conversion is a tradeoff between speed, power, accuracy, noise and price. The simplest A/D converters use a single op-amp and iterate towards the correct solution. This takes time. A flash A/D converter uses many in parallel, which is fast but uses a lot of power and is expensive.

11.7 A Temperature Sensor

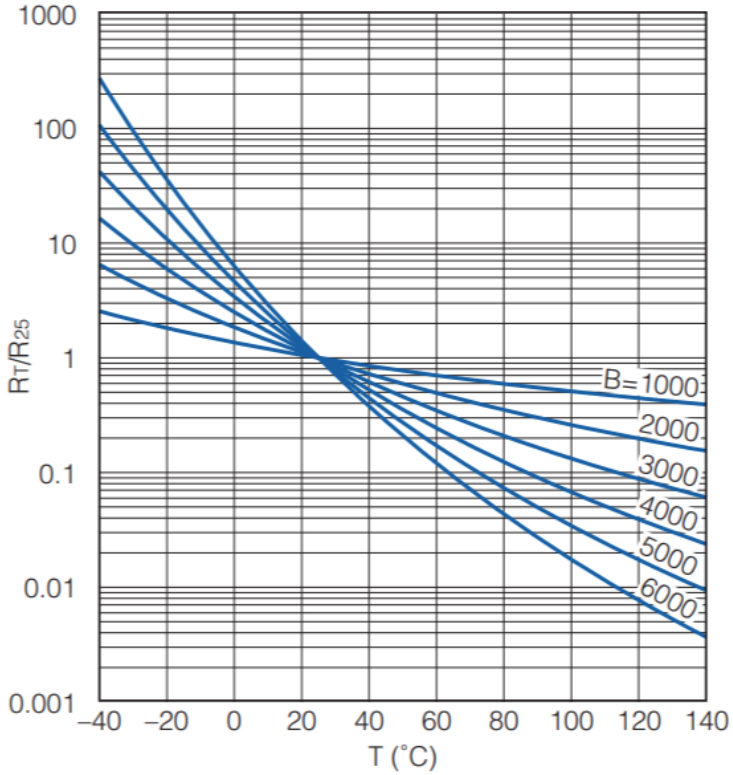
We humans are visual, so we typically have instruments that measure a quantity (like temperature) and turn it into something we can see (like a linear measurement).

Since computers are electrical, they must measure an electrical quantity. Most A/D converters measure voltage, but current can be measured by putting it through an extremely accurate, small resistor.

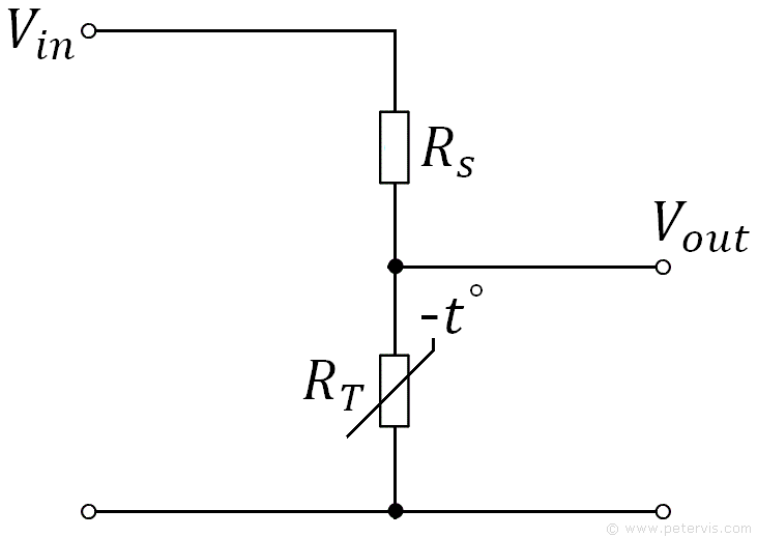
On a computer, it is far easier to turn the temperature into voltage, rather than implementing image recognition and trying to detect visually.

In this section, we will assume we have an A/D converter capable of reading in a voltage and converting to an integer. The remaining question is how to convert the desired input into voltage. This description will start with temperature, but the procedure for light would be similar, subsituting a photoresistor for the thermistor.

The first step is to build a circuit that turns temperature into voltage. There are components called thermistors whose resistance varies with temperature. The kind most often used are semiconductors, and as the temperature increases, there are sharply increasing numbers of carriers available, so the resistance drops. This type of thermistor is called a Negative Temperature Coefficient (NTC). The following diagram shows a curve of a typical thermistor's resistance as a function of temperature.



Only the resistance changes. If a voltage is applied across a thermistor, the voltage would remain stable, more varying amounts of current would flow. Instead, a thermistor can be used to build a temperature to voltage converter by creating a voltage divider with a fixed resistor.



If the fixed resistor is very large compared to the thermistor, then the value of the thermistor does not matter very much, the voltage across it will always be the same. Similarly, if the value of the resistor is very small compared to that of the thermistor, then most of the voltage drops across the thermistor regardless of its value. The ideal value of the resistor is the middle of the range. For example, if a thermistor is 5500Ω at 0 degrees C, and 500Ω at 100 degrees C, then the optimal value for measuring temperature is about halfway, or somewhere in the neighborhood of 3500Ω.

11.8 Further Reading

https://en.wikipedia.org/wiki/Analog-to-digital_converter
https://www.arrow.com/en/products/search?prodline=Analog+to+Digital+Converters+-+ADCs&selectedType=plNames https://www.maximintegrated.com/en/design/technical-documents/tutorials/1/1870.html https://www.silabs.com/community/blog.entry.html/2015/03/19/jitter_what_it_is-eVzF

12. Busses

12.1 Introduction

12.2 USB

12.3 PCIe

12.4 Ethernet and Limits to Line Bandwidth

Every 8 bits is translated into 10 bits for error detection and correction.

s each group of 8 data bits to four quinary symbols. Each quinary symbol is then line encoded using 4D-PAM5, which is a system that used five voltage levels, (similar to MLT-3 which uses three levels.)

Since 2 bits are represented for each quinary symbol and the clock rate is set at 125MHz, this gives 250Mbps data per twisted pair and therefore 1000Mbps for the whole cable.

The reason is that it is increasingly hard to drive signals and higher frequencies. Even for 1Gb, already the scheme uses multiple voltage levels and a slower signal rate.

For 10Gb, even though they have existed for years prices are still high and it is easy to see from the big heat sinks that power dissipation is also high.

12.5 Further Reading

http://units.folder101.com/cisco/sem1/Notes/ch7-technologies/encoding.htm#1000base-t https://en.wikipedia.org/wiki/Ethernet

13. Exploits

We have all heard about viruses, which can take over a computer and do damage. Our goal as users is not to run viruses, or to restrict permissions so that the operating system can stop them. But researchers are now finding vulnerabilities in the hardware that can lead to exploits based on fundamental properties of the hardware.

13.1 Spectre and Meltdown

13.2 Side Channel Attacks

A side channel attack is an attack that uses external equipment to measure something that changes as a result of computation. If the computer is leaking information, then it may be possible to discover secrets based on measurement. For example,

13.3 Further Reading

https://www.wired.com/story/air-gap-researcher-mordechai-guri/
https://threatpost.com/air-gap-attack-turns-memory-wifi/162358/
https://en.wikipedia.org/wiki/Stuxnet

14. Ethics

This chapter contains a number of links to readings on interesting areas of ethics. You may be asked to study a particular issue and take a stand, justifying your argument as a break from all the programming.

Fundamental ethical questions in computing include:

- Should there be control over encryption? Should everyone be able to keep secrets?
- Should software be proprietary?
- What is the Digital Millenium Copyright Act? Is it good or bad?
- What is net neutrality? Why should we care?
- Should there be a right to repair equipment?
- Should manufacturers have to make their devices easier to repair?
- Who is responsible for e-waste?
- Is it moral to break copyright by illegally copying software? books? movies?
- What is the law on copyright and how has it changed?
- What is the digital divide, and are there any issues of fairness?
- Governments are often concerned that cryptography can prevent them from finding out a criminal's secrets. Is this justified or not? How can governments weaken cryptography and what are the possible ramifications?
- Should car manufacturers be able to offer features in cars if it means they might be vulnerable to hacking?

14.1 Further Reading

https://digify.com/blog/digital-rights-management/
https://www.repair.org/stand-up
https://en.wikipedia.org/wiki/Digital_Millennium_Copyright_Act#:~:

text=The%20Digital%20Millennium%20Copyright%20Act,Intellectual%20Property%
20Organization%20(WIPO).&text=It%20also%20criminalizes%20the%20act,actual%
20infringement%20of%20copyright%20itself. https:
//locusmag.com/2021/01/cory-doctorow-neofeudalism-and-the-digital-manor/
https://www.defectivebydesign.org/
https://en.wikipedia.org/wiki/Digital_rights_management
https://en.wikipedia.org/wiki/Trusted_Computing
https://www.zdnet.com/article/you-cant-open-source-license-morality/
https://www.schneier.com/blog/archives/2021/06/
intentional-flaw-in-gprs-encryption-algorithm-gea-1.html
https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/
https://www.gnu.org/philosophy/can-you-trust.en.html

15. Further Topics

15.1 GPU

Technologies beyond the scope of this course

- GPU (Graphics Programming Unit) On the order of 3000 to 4000 processors, each perhaps 50% the speed of a core Compare: 32 cores in very high end desktop processors GPU is on the order of 1000 times faster than the regular CPU in a PC Memory: GDDR5, GDDR6, HBM
- MIMD (Multiple Instructions Multiple Data) Big supercomputers have 10^5 or 10^6 computers, each with its own memory, all passing mesages to each other. A custom Chinese supercomputer uses simple ARM cores, 256 per chip. Many chips Computing today is also limited by power, so there is concern over compute/Watt, not just absolute compute
- FPGA Field Programmable Grid Array Program the circuit. Instead of programming a computer to interpret a program, the circuit is the program. Extremely parallelizable Slower clock speed, low power Very efficient and the fastest way to implement very high end algorithms
- Future technologies

  optical computing

  quantum computing

16. C++ and Assembler Equivalences

16.1 Summary of Integer Operations for ARMV7a, ARMv8, and Intel

This section summarizes the core computer capabilities on ARMv7a (what a Raspberry Pi runs in 32-bit mode), an ARMv8 (a Raspberry Pi if run in 64-bit mode) and Intel. This is only the basic core of the CPU, not vectorized code which is in a different section.

The names shown are as defined in gnu assembler, running on Linux or msys2 under windows. There is alternate AT&T syntax which is different. On Windows there are other assemblers as well.

Topic	ARMv7a	ARMv8	x86
Registers	r0..r15 r15 pc r14 lr r13 sp r12	x0..x31 x31 = 0	%rax %rcx %rdx %rbx base pointer %rsp stack pointer %rbp base pointer %rsi source index %rdi destination index %r9 .. %r15

16.2 Integer Operations

Integer Ops In this section, we will see for each operator the corresponding assembler in ARMv7a, ARMv8, and Intel x86. You will notice that there is no divison operator for ARMv7, and so that calls a subroutine instead which is vastly slower. Both ARMv8 and Intel have a division instruction which is quite slow compared to addition (15 clocks on Intel I7 10th generation) but considerably faster than calling a subroutine and writing it in software.

C++	ARMv7a

```
int op0(int a, int b) {
        return a + b;
}
```

```
  add x0, x0, x1
  bx  lr
```

```
add r0, r0, r1
bx  lr
```

```
leal  (%rdi,%rsi), %eax
ret
```

```
int op1(int a, int b) {
        return a - b;
}
```

```
  sub x0, x0, x1
  bx  lr
```

```
sub r0, r0, r1
bx  lr
```

```
movl  %edi, %eax
subl  %esi, %eax
ret
```

```
int op2(int a, int b) {
        return a * b;
}
```

```
  mul x0, x0, x1
  bx  lr
```

```
mul r0, r0, r1
bx  lr
```

```
movl  %edi, %eax
imull %esi, %eax
ret
```

```
int op3(int a, int b) {
        return a / b;
}
```

```
  div x0, x0, x1
  bx  lr
```

```
[TBD]
bx  lr
```

```
movl  %edi, %eax
idivl %esi, %eax
ret
```

16.3 Integer Loops

The following table shows C++ code and the equivalent ARM assembler both for v7a, v8, and x86. Tables in this chapter may not be complete yet because testing all this code takes a great deal of time. All code is shown optimized, and because the compiler will optimize loops out of existence if they do nothing, we pass the loop variable into the function so the compiler cannot determine what happens and is forced to generate the code. Note that an empty loop that does nothing will be completely eliminated by the optimizer as dead code.

C++	ARMv7a

`void f0_1(int n) {`
`  for (int i = 0; i < n; i++)`
`  f(i);`
`}`
	`  mov  r1, #0`
	`1:`
	`  add  r1, #1`
	`  cmp  r1, r0`
	`  bne  1b`

`  mov  x1, #0`
`1:`
`  add  x1, #1`
`  cmp  x1, x0`
`  bne  1b`
	`  xorl  %ebx, %ebx`
	`1:`
	`  movl  %ebx, %edi`
	`  addl  $1, %ebx`
	`  call  _Z1fi@PLT`
	`  cmpl  %ebx, %ebp`
	`  jne  1b`

`void f0_2(int n) {`
`  int i = 0;`
`  do {`
`  f(i);`
`  i++;`
`  } while (i < n);`
`}`
	`  mov  r1, #0`
	`1:`
	`  add  r1, #1`
	`  cmp  r1, r0`
	`  bne  1b`

```
        mov     x1, #0
1:
        add     x1, #1
        cmp     x1, x0
        bne     1b
```

```
        xorl    %ebx, %ebx
1:
        movl    %ebx, %edi
        addl    $1, %ebx
        call    _Z1fi@PLT
        cmpl    %ebx, %ebp
        jne     1b
```

```
void f0_3(int n) {
  int sum = 0;
  for (int i = 1; i <= n; i++)
    sum += i;
  return sum;
}
```

```
    mov     r1, #0
    mov     r2, #1
    cmp     r0, #1
    ble     2f
1:
    add     r1, r2
    add     r2, #1
    cmp     r2, r0
    ble     1b
    mov     r0, r1
2:
    bx      lr
```

```
@TODO: does not properly set answer
    mov     x1, #0
    mov     x2, #1
    cmp     x0, #1
    bxle    lr
1:
    add     x1, x2
    add     x2, #1
    cmp     x2, x0
    ble     1b
    mov     x0, x1
2:
    bx      lr
```

```
    movl    $1, %eax
    xorl    %r8d, %r8d
1:
    addl    %eax, %r8d
    addl    $1, %eax
    cmpl    %edi, %eax
    jne     1b
    movl    %r8d, %eax
    ret
```

```
int f0_4(int n) {
  int prod = 1;
  for (int i = 2; i <= n; i++)
    prod *= i;
  return prod;
}
```

```
  addl  $1, %edi
  movl  $2, %eax
  movl  $1, %r8d
1:
  imull %eax, %r8d
  addl  $1, %eax
  cmpl  %edi, %eax
  jne   1b
  movl  %r8d, %eax
  ret
```

```
// Integer division
int f0_5(int n) {
        int divides = 1;
        for (int i = 1; i <= n; i++)
                divides /= i;
        return divides; // of course this is zero, but the compiler doesn't
}
```

```
  addl  $1, %edi
  movl  $1, %ecx
  movl  $1, %eax
1:
  cltd
  idivl %ecx
  addl  $1, %ecx
  cmpl  %edi, %ecx
  jne   1b
  ret
```

16.4 If Statements

16.5 Function Calls

16.6 Recursion

16.7 Object-Oriented Method Calls

16.8 Floating Point

17. Appendix B: Circuit Terminology

18. Appendix C: Common C++ Equivalents in Assembler

19. Linux Commands

This appendix is not to try to create a universal manual for Linux, but rather to alert you to the list of basic commands that everyone needs to know to get around, then the commands for development including g++, gdb, and git. Perhaps we might move those out to separate appendices.

19.1 Basic Systems Commands

`ls`	list the files in the current directory
ls −a	List all files in the current directory, including the hidden files starting with .
ls −l	List in long format showing permissions and file sizes
ls −R	List recursively
cd dir	change to directory dir
cd	change to my home directory
pwd	present working directory
mkdir [dirname]	create a directory with that name in the current directory
control−C	Cancels, exits running program
control−Z	Suspends current job running. Then type bg to make it go in the background
./xyz	runs xyz in the current directory. If you get tired of this you can add . to the path. Edit the file .profile in your home directory and add the command **export** PATH=.:$PATH$
mv file.cc newname.cc	renames file.cc in the current directory
`mv file.cc ..`	moves file.cc to the parent directory of this one
`mv file.cc ~/`	moves file.cc to the your home directory

`cp a.txt b.txt`	copy file to new name b.txt
cp *.txt ~/Documents	copy all files with suffix .txt Documents dir
cat file.cc	display file.cc to the screen
`cat c >>out`	read in file c and append to the end of out
cat a.txt b.txt >c.txt	concatenate a.txt and b.txt and write to c.txt
rm file.cc	Deletes file (no undo, be careful)
rm −r directory	DANGEROUS. Be careful. Delete a directory and everything under it recursively.
grep void file.cc	Search for string "void" within file.cc
less file	Display the file a page at a time, press space to keep displaying or q to quit
df	Disk free. Find out how much space remaining on all system drives
df −h	Report space in 1k units
df −h .	Report space in 1k units on current drive
du −sh dir	Find how much space is used by dir and all files under it.
find . −name "myfile.txt"	Find the named file somewhere under the directory .
locate filename	Use the precomputed location database to search for files, requires updatedb.
diff a b	Display the differences between file a and b
grep target *.cc	Search for the string target in all the files ending in .cc
grep −r target *.cc	Recursively do the same search in all dirs under this one
blockdev −getbsz /dev/sda1	Get the block size on the named device. Get the name from df

19.2 Networking Commands

ifconfig	Display the network ports available and their status
ping 192.168.1.1	check whether ip address is reachable
ssh pi@192.168.1.16	log in remotely to a computer via ssh using account pi
scp *.cc pi@192.168.1.16:temp	copy all files .cc to remote server in directory temp under account pi
ssh-keygen -t rsa	create public/private keypair in /.ssh
ssh-copy-id user@remotecomputer	copy the locally generated public key here to there so login can be passwordless subsequently.

19.3 Git Commands

git clone [url]	Get a git repo and create a directory mirroring it locally
git status	Shows which files are in queue to be moved to github
git add .	preps all files in current directory to be added to repo
git commit -m“[comment goes here]”	Attaches a comment to all files added to explain what you did
git push	copies all changes with the comments to the repo
git pull	copy any changes made to a git repo to the local copy here
git merge	take two different copies (like the local and remote) and merge them. Manual fixed may have to be made if git cannot figure out what to do. Typically if two people edit the same line, the human has to sort it out.

19.4 g++ Compiler Commands

g++ foo.cc	Compiles foo.cc, then links the resulting output to create an executable. The default name in unix is called a.out. In Windows it is a.exe
g++ foo.cc -o foo	Compiles and links foo.cc and creates an executable named foo (on windows, foo.exe)
g++ -c file.cc	Compiles file.cc and stores the object code to file.o The file is not linked, and no executable is created compile a c++ program with debugging, so you can see the variables and step through a line at a time using gdb
g++ -O2 foo.cc	Optimizing code so that the program runs faster.
g++ -g -O2 foo.cc	Compile a program with both debugging and optimization. You can do this, but since the optimizer is rewriting your program to be more efficient, the lines of your original program may no longer map cleanly to the machine language. And some variables may no longer exist.
g++ -O2 -march=native foo.cc	Compile a program using this CPU only. For example, on your laptop, older version of the 80x86 processor might no longer work, but the code, using features only available on this cpu, might work faster. This is, in general, the fastest code you can generate in C++ on the Intel platform.
g++ -g a.cc b.cc c.cc -o myprog	Compile a program that is split into 3 c++ source files (a.cc, b.cc, c.cc) and create the executable myprog. Notice that on linux/MacOSX, the program does not end with .exe, though on Windows it will.

g++ -S -O2 a.cc	Compile the c++ code in a.cc, optimize so it is fairly good, and instead of generating code, generate an assembler file a.s which shows the instructions to be generated. This is a great way of learning to code on any architecture supported by gcc because their code is pretty good most of the time.
g++ -g hw1.cc b.s -o hw1	Compile a c++ file that is given to you (hw1.cc), an assembler file that you write (b.s) and generate hw1. This is what you for each homework, although this command is written in a makefile so you do not have to type it.
g++ file .cc >log 2>logerr	Compiles file.cc to log, in case of error, redirect to logerr
make	Execute the instructions in the file Makefile to build a project. This is commonly used for C and C++, though make can be used for other things as well.
gdb a.out	Execute the debugger on the default executable name a.out
gdb executable	Execute the gnu debugger on a program. Requires -g flag if you want to see symbols and step through in C++.
g++ foo.s	Calls the assembler (as) to compile assembler code and then link to an executable called a.out by default
gprof executable	Execute the profiler which records how much time was taken by each function. This is used to help optimize programs.
valgrind executable	Find memory leaks using the valgrind tool

19.5 gdb Commands

b function	Set a breakpoint at named function. Debugger will stop there
r	Start running the program, or re-run from start.
c	Continue execution until the end or the next breakpoint
c 500	Continue execution 499 times and stop on the 500th breakpoint
p expr	Print any legal expression
p $reg	print the value of a register
p $reg = value	Set the value of a register.
disp x	Print out the contents of the variable x every time the debugger stops at a breakpoint. If x does not exist the debugger will say so.
layout src	Create a two-paned text mode debugger that interactively shows the source while stepping.
layout asm	Show assembler source while stepping
layout reg	Show the registers for assembler debugging
n	Execute one line in a high-level language (requires debugging information -g)
s	Step into a function in a high-level language (required debugging info)
ni	Next instruction in assembler
si	Step into function call in assembler
x/s $reg	Display memory at location specified by register as a string

b * 0x1234567	Set breakpoint at a specific memory location
clear *0x1234567	Clear breakpoint at a specific memory location

19.6 Low Level System Commands for Programmers

xxd file	display dump of the file showing binary data
objdump -d file	display machine language in executable
pmap -X ‘pidof myprog‘	run myprog, get the process id, and feed it to pmap to view memory use

19.7 Editors

You will need a programming editor to type your programs. Raspberry Pi does not seem to have many editors built for it, and in truth it is pretty slow for a big IDE. There are currently three choices on the Pi, and a few more editors you can use on a bigger computer remotely editing.

nano file.c	https://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/
emacs file.cc	emacs, a powerful but crazy old editor. You can see my cheatsheet for emacs here: https://drive.google.com/open?id=1RwtEh8nAVBFn7GxjrZZunsAQGs3egwL4
vi file.cc	vi is one of the original Unix editors. Powerful and cryptic. In order to start typing, press i to insert, ESC to get out of insert mode, and capital ZZ to save the current file and exit. Tutorial: https://www.howtoforge.com/vim-basics

19.8 Intel Assembler

Intel is a CISC processor, and each generation they have slapped on weird instructions that only work with certain registers. After more than 30 years, the resulting instruction set may be compatible with older instructions but it’s a mess. In this section we are not going to describe all the old instructions, just the new ones. Still, you will see the names of the registers are not exactly regular like ARM, and that is a legacy of all the old baggage riding along.

This appendix showed the modern, 64-bit PC instruction set, also called the AMD 64-bit extensions which all modern CPUs in this family (including Intel) use.

There are 16 64-bit registers. Particularly for Intel, there are multiple assemblers with different syntaxes, so for this section we are only considering the same Linux as assembler as we use for ARM. The register names start with percent (%) as follows:

%rax %rbx %rcx, %rsi, %rdi, %r8, %r9, %r10, %r11,%r12, %r13,%r14, %r15

The names have vestigial meanings: ax was the accumulator which got all answers from the ALU in the original 8-bit CPU. The registers si and di were used for source and destination in various memory copy instructions. These names are no longer relevant in the modern instruction set and you may consider them as registers 0 to 15 for the most part.

The registers may also be addressed as 32-bit and 16 bit. The 32-bit registers start with e:

%eax, %ebx, %ecx, ...

and the 16 bit registers are without the leading e.

Intel instructions in the Unix AT&T style go from left to right, the opposite of ARM. For example:

movl %ecx, %eax means eax $\leftarrow$ ecx.

Instructions are either 8,16,32 or 64 bits. The suffix b(byte), l(long, 32) and q(quad, 64) are used to indicate the type of instruction. For example, the add instruction is either addb, add, addl, or addq.

Arithmetic operations are all two operand instead of 3-operand. This means that one of the operands is also the destination, so one is always destroyed by the operation. In order to compute $rcx = rax + rdi$:

```
mov    %rdi, %rcx   ;rcx = rdi
add    %rax, %rcx
```

This tradeoff means that every ARM arithmetic instruction is the equivalent of 2 Intel instructions, but often it does not matter if the value in the register is changed, and in that case the move instruction is not needed. Moreover because there is less information in each instruction, they can be smaller. Fewer bytes means more instructions fitting into cache. At least that is the theory. In practice, the fact that Intel has wildly complicated instructions that take from 2 to 7 bytes makes decoding complicated and clearly slows them down.

Intel allows operands to not only be from register, but from memory. Thus instead of having load/store instructions, Intel can load as part of an arithmetic operation. This means that an arithmetic operation is not always one clock cycle. In fact it can be very slow if it is waiting for a load, which complicates the architecture. So for example, in order to add two elements in memory on the ARM we first have to load both into registers, then add the registers:

```
ldr r1, [r0], #4
ldr r2, [r0], #4
add r3, r1, r2
```

On the Intel we would have to load the first one in, then we could add the second to the first, but Intel does not have autoincrement mode and so requires extra instructions to advance the pointer through memory:

```
mov    (%rdi), %eax  @ 32 bit move comparable to above
add    $4, %rdi
add    (%rdi), %eax
add    $4, %rdi
```

Arithmetic instructions

The following table shows the different modes for a mov instruction. The same apply equally to add, sub, etc. The following shows 64 bit operations for each mode.

movq $1, %eax	immediate mode	move the constant 1 to register
movq %ebx, %eax	register-register	move %ebx to %eax
movq $24(%ebx), %eax	memory to register	move 16 bits at 24+%ebx to %eax
movq location(%ebx,%ecx,4), %eax	double indexed	eax ← location[ebx+ecx*4]

The following table shows the arithmetic instructions for Intel. Note the special instruction in the first row. Load Effective Address (LEA) is often used to add on Intel because unlike all the math functions, it is 3 operand like ARM.

leal (%rdi,%rsi), %rax	rax ← rdi + rsi
addq %rax, %rbx	rbx ← rax + rbx
subq %rax, %rbx	rbx ← rax - rbx
mulq %rax, %rbx	rbx ← rax * rbx
divq %rax, %rbx	rbx ← rax / rbx

19.8.1 Branching

Intel short branches are relative like ARM, but some branches are absolute. This means that code is not, by default position-independent, and there are some notable cases where this is a problem. For example, a shared memory library is loaded once for all programs, but it may not appear to be in the same memory for each task. On the ARM, branches are relative and code is by default, position independent. On Linux on the PC, code must be specifically compiled as position independent.

call	Call a function
ret	Return from function

Mnemonic	Jump if...	Flags
JO	overflow	OF = 1
JNO	not overflow	OF = 0
JS	sign	SF = 1
JNS	not sign	SF = 0
JE	equal	ZF = 1
JZ	zero	
JNE	not equal	ZF = 0
JNZ	not zero	
JB	below ¡ unsigned	CF = 1
JNAE	not above or equal	
JC	carry	
JNB	not below	CF = 0
JAE	above or equal $\geq$ unsigned	
JNC	not carry	
BE	below or equal $\leq$ unsigned	CF = 1 or ZF = 1
JNA	not above	
JA	above ¿ unsigned	CF = 0 and ZF = 0
JNBE	not below or equal	
JL	less than ¡ signed	SF $\neq$ OF
JNGE	not greater than or equal	
JGE	greater or equal $\geq$ signed	
JNL	not less	SF = OF
JLE	less or equal $\leq$ signed	ZF = 1 or SF $\neq$ OF
JNG	not greater	
JG	greater ¿ signed	ZF = 0 and SF = O
JNLE	not less or equal	
JP	parity	PF = 1
JPE	parity even	
JNP	not parity	PF = 0
JPO	parity odd	
JCXZ	CX register is 0	cx = 0
JECXZ	ECX register is 0	ecx = 0

19.8.2 Floating Point

Intel has 8 floating point registers in a stack. This vestigial engine is rarely used today. CPUs also began adding extra instructions for rapid execution of specialized computations such as compression, and this morphed into a vector engine. Intel developed extensions called SSE, SSE2, AVX, and AVX2. Since most of these are obsolete we will just focus on the current AVX2 capabilities which are a superset. Today, all desktop and laptop CPUs (probably not some low-power ones) since Intel 4th generation have AVX2 instructions. There is a newer standard AVX512, but it is not currently used very much and is only available on a few Intel models and not at all on AMD. It remains to be seen whether AVX512 will ever get popular.

For now, we are using the AVX2 registers because this is what is generated by g++ if you compile floating point code.

addsd	%xmm1, %xmm0	xmm0 ← xmm0 + xmm1
subsd	%xmm1, %xmm0	xmm0 ← xmm0 - xmm1
mulsd	%xmm1, %xmm0	xmm0 ← xmm0 * xmm1
divsd	%xmm1, %xmm0	xmm0 ← xmm0 / xmm1
sqrtsd	%xmm1, %xmm0	xmm0 ← $\sqrt{xmm1}$
pxor	%xmm1, %xmm0	xmm0 ← xmm0 XOR xmm1
ucomisd	%xmm1, %xmm0	unsigned double scalar comparison
movapd	%xmm1, %xmm0	xmm0 ← xmm1
movsd	offset(%rip), %xmm0	xmmo ← m64[pc+offset]

AVX and AVX2 vector instructions

VGATHERDPD Using dword indices specified in vm32x, gather double-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.

vxorpd	xmm1, xmm2, xmm3	xmm1 ← xmm2 XOR xmm3
VBROADCASTSS	xmm1, m32	Broadcast single-precision floating-point element in mem to four locations in xmm1.
VBROADCASTSS	ymm1, m32	Broadcast single-precision floating-point element in mem to eight locations in ymm1.
VBROADCASTSD	ymm1, m64	Broadcast double-precision floating-point element in mem to four locations in ymm1.
VBROADCASTF128	ymm1, m128	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.
VBROADCASTSS	xmm1, xmm2	Broadcast the low single-precision floating-point element in the source operand to four locations in xmm1.
VBROADCASTSS	ymm1, xmm2	Broadcast low single-precision floating-point element in the source operand to eight locations in ymm1.
VBROADCASTSD	ymm1, xmm2	Broadcast low double-precision floating-point element in the source operand to four locations in ymm1.
VPERMPS	ymm1, ymm2, ymm3/m256	Permute single-precision floating-point elements in ymm3/m256 using indices in ymm2 and store the result in ymm1.

VFMADD132SD	xmm1, xmm2, xmm3/m64	xmm1 ← xmm1 * xmm3/m64 + xmm2
VFMADD213SD	xmm1, xmm2, xmm3/m64	xmm1 ← xmm1 * xmm2 + xmm3
VFMADD231SD	xmm1, xmm2, xmm3/m64	xmm1 ← xmm2*xmm3/m64 + xmm1
VFNMADD132PD	xmm1, xmm2, xmm3/m128	xmm1 ← -xmm1 * xmm3/mem + xmm2
VFNMADD213PD	xmm1, xmm2, xmm3/m128	xmm1 ← -xmm1 * xmm2 + xmm3/mem
VFNMADD231PD	xmm1, xmm2, xmm3/m128	xmm1 ← -xmm2 * xmm3/mem + xmm1
VFNMADD132PD	ymm1, ymm2, ymm3/m256	ymm1 ← -ymm1 * ymm3/mem + ymm2
VFNMADD213PD	ymm1, ymm2, ymm3/m256	ymm1 ← -ymm1 * ymm2 + ymm3/mem
VFNMADD231PD	ymm1, ymm2, ymm3/m256	ymm1 ← -ymm2 * ymm3/mem ymm1
VFMSUB132PD	xmm1, xmm2, xmm3/m128	xmm1 ← xmm1 * xmm3/m64 - xmm2
VFMSUB213PD	xmm1, xmm2, xmm3/m128	xmm1 ← xmm1 * xmm2 - xmm3/m64
VFMSUB231PD	xmm1, xmm2, xmm3/m128	xmm1 ← xmm2 * xmm3/m64 - xmm1
VFMSUB132PD	ymm1, ymm2, ymm3/m128	ymm1 ← ymm1 * ymm3/m64 - ymm2
VFMSUB213PD	ymm1, ymm2, ymm3/m128	ymm1 ← ymm1 * ymm2 - ymm3/m64
VFMSUB231PD	xmm1, xmm2, xmm3/m128	ymm1 ← ymm2 * ymm3/m64 - ymm1

VEXTRACTF128	xmm1/m128, ymm2, imm8	m128[] ← ymm2
VGATHERDPD	xmm1, vm32x, xmm2	xmm1 ←
VGATHERQPD	xmm1, vm64x, xmm2	Using qword indices specified in vm64x, gather double-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VGATHERDPD	ymm1, vm32x, ymm2	Using dword indices specified in vm32x, gather double-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VGATHERQPD	ymm1, vm64y, ymm2	Using qword indices specified in vm64y, gather double-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.

19.9 Accessing Vector Operations from C++

You don't have to write code in assembler to use the AVX instructions. Instead, there are intrinsics, little functions in C++ that make them available. Of course, including intrinsics in C++ code makes it completely unportable. It not only will only work on the Intel platform, it will not even run on older machines that do not support AVX. But for high performance, it is sometimes worth it. The following examples show how to use AVX instructions within C++ programs using intrinsics.

Example: Slow dot product. This example is taken from a github repo shown below (jshitikl/avx2-examples). This is clearly not the best example of performance improvement as reading takes a significant amount of time, but it has the benefit of being simple. First, see the ordinary C++ code that computes a dot product one number at a time.

The first c code is ordinary dot product doing the math a single value at a time

```
double simple_dot_product(const double a[], const double b[], int n) {
  double answer = 0.0;
```

```
  for(int i = 0; i < n; ++i)
    answer += a[i]*b[i];
  return answer;
}
```

The next function uses AVX2 intrinsics to try to increase execution speed.

```
double dot_product(const double a[], const double b[], int n) {
  __m256d sum = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);

  /* Add up partial dot-products in blocks of 256 bits */
  for(int i = 0; i < n; i += 4) {
    __m256d x = _mm256_load_pd(a+i); // load 4 doubles in a burst
    __m256d y = _mm256_load_pd(b+i); // same
    __m256d z = _mm256_mul_pd(x,y);  // multiply each corresponding
    sum = _mm256_add_pd(sum, z); // add them in parallel
  }
  __m256d temp = _mm256_hadd_pd(sum, sum); // add each half
  __m128d sum_high = _mm256_extractf128_pd(temp, 1); // pull out each half
  __m128d sum_low = _mm256_castpd256_pd128(temp);
  __m128d result = _mm_add_pd(sum_high, sum_low); // add the two halves
  return ((double*)&result)[0];
}
```

Many intrinsics in the following table end with ps (8 floats) or pd (4 doubles)

_mm256_setzero_ps/pd	Returns a floating-point vector filled with zeros
_mm256_setzero_si256	Returns an integer vector whose bytes are set to zero
_mm256_set1_ps/pd	Fill a vector with a floating-point value
_mm256_set1_epi8/epi16 _mm256_set1_epi32/epi64	Fill a vector with an integer
_mm256_set_ps/pd	Initialize a vector
_mm256_set_epi8/epi16 _mm256_set_epi32/epi64	Initialize a vector with integers
_mm256_set_m128/m128d _mm256_set_m128i	Initialize a 256-bit vector with two 128-bit vectors
_mm256_setr_ps/pd	Initialize a vector in reverse order
_mm256_setr_epi8/epi16 _mm256_setr_epi32/epi64	Initialize a vector with integers in reverse order

19.10 Further Reading

https://en.wikipedia.org/wiki/Advanced_Vector_Extensions
https://www.felixcloutier.com/x86/ https://github.com/kshitijl/avx2-examples
https:
//www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX

20. Homework Problems

Simple homeworks for getting started in ARM assembler

- Given two registers r0 and r1, swap the values. That is, if r0 = 3 and r1 = 4, by the end of your code r0 = 4 and r1 = 3.
- Given a call from C++ to your arm function passing an integer in r0, count down from r0 down to 0. Then stop and return.
- Given a call from C++ to your arm function passing an integer in r0, count down from r0 down to 1. Then stop and return.
- Given a call from C++ to your arm function passing an integer in r0, count up from 0 to r0. Then stop and return.
- Given a call from C++ passing two integer values in r0 and r1, compute the sum and return in the standard way (r0 has the answer).
- Given a call from C++ passing two integer values in r0 and r1, sum all the numbers from r0 to r1. For example, if r0=3 and r1 = 5, then the answer should be $3+4+5=12$.
- Same as the previous function, but if r0 ¿ r1 the answer returned should be zero.

The following list contains C++ function prototypes and descriptions of what to compute. Your homework will refer to this list and specify whether you are to write in C++ or ARM assembler.

1. Compute the sum of the numbers from a to b.
   ```
   int sum(int a, int b);
   ```
2. Compute the product of the numbers from a to b.
3. compute n factorial. Determine which value of n will overflow a signed 32 bit int and write that in a comment at the top of your code.
4. Return the sum of the squares from a to b inclusive. Example:
   $sumsq(3,5) = 3*3+4*4+5*5$

5. Compute the arithmetic series $a + a + s + a + 2s + a + 3s + ...$ up to b
6. Compute the series $a * (a + s) * (a + 2s) * ...$ up to b
7. Compute the sum of the elements in array x from position 0 to n-1
8. Compute the product of the elements in array x from position 0 to n-1
9. Compute the sum of the elements in array x from position a to b inclusive
10. Compute the product of the elements in array x from position a to b inclusive
11. Compute the Root-mean-square of the elements in x. Square each element, sum them, then return the square root.

Given the following Matrix in C++, write the methods in C++ or assembler as directed

```
class Matrix {
private:
  uint32_t w, h; // width and height of the matrix
  double* p;     // p will point to a w * h rectangle of numbers in row-maj
public:
  Matrix(uint32_t w, uint32_t h, double v) : w(w), h(h), p(new double[w*h])
    for (int i = 0; i < w*h; i++)
      p[i] = v;
  }
  ~Matrix() {
    delete[] p;
  }
  void setAll(double v); // set all values in the matrix to v
  void setMainDiagonal(double v); // assuming w=h, set all elements on the
  double sumColumn(uint32_t c) const; // return the sum of column c
  double sumRow(uint32_t r) const; // return the sum of row r
  friend void operator+(const Matrix& a, const Matrix& b);
  friend void operator-(const Matrix& a, const Matrix& b);
  friend void operator*(const Matrix& a, const Matrix& b);
}

double median(double x[], int n); // assuming x is sorted, return the middl
// if there are an even number of elements, compute the average of the two

void curve(double x[], int n);

int main() {
  cout << sum(3, 5) << '\n';
}
```

- Calculate parity of an integer in ARM assembler

```
int parity(int v);
```

- Return the number of iterations that it takes for the 3n+1 problem to converge. Starting with a number (passed in r0) apply the followings rules:

  if the number is odd, multiply by 3 and add 1.
  if the number is even, divide by 2.

  repeat until the number converges to 1. Barring overflow, all numbers should converge.

- Find the maximum 3n+1 for all numbers ¡ 10 million. Write fibo(n) recursively in assembler and compute fibo(40).

- Write a function to accept an integer and reverse the bits in the integer.

  For example, the integer 5 = 00000000000000000000000000000101 The bit reverse of that is 10100000000000000000000000000000

  ```
  int bitreverse(int v);
  ```

- Write a function that takes an array of double precision numbers and the length of the array. replace every pair of numbers x[i], x[i+1] for i=0, 2, 4, ... by:

  ```
  temp = x[i] + x[i+1]
  x[i+1] = x[i] - x[i+1]
  x[i] = temp
  ```

- Write a bubblesort in ARM assembler:

  ```
  void bubblesort(int a[], int n);
  ```

  The bubblesort algorithm is two nested loops. The inner loop compares each element a[i] to the one to its right a[i+1]. To sort into ascending order, if a[i] ¿ a[i+1], swap the two values. The outer loop in the simplest form just executes the inside loop n-1 times.

- Write a function to perform Gram-Schmidt Orthogonalization in the following way.

  - Transpose the matrix so that the columns are rows which are sequential in memory, for speed.
  - Read in each column and write to a row of a temporary matrix. While doing this calculate the magnitude of the vector.
  - Invert the magnitude.
  - Normalize the row by multiplying by the inverse magnitude.
  - For each subsequent row, subtract the first

21. Labs

Labs These labs are the version stored in LaTeX. Eventually this will be the official lab, but for now the formatting here is still imperfect, and there may be bugs. Just use this material as an explanation of the general approach. Reading the lab material and references before lab each week will help you understand what the goal is and get the lab done efficiently.

22. Preparing C++ Environment, git, Building Code in Teams

Install gcc toolchain on your laptop if you have not already done so. If you are done, and have a Raspberry Pi, you may take some time to do that. Get help from the TA if you need it. If you have problems getting your programming environment up and running and the TA is busy, you can write programs and test online at a site like: http://repl.it/
https://www.jdoodle.com/c-online-compiler/

Install a programming editor. On the laptop you have your choice of many great ones but for class we need microsoft vscode. See the class repo for instructions on how to set up:

https:
//github.com/StevensDeptECE/CPE390/tree/master/Course%20Documentation
For Raspberry Pi we have only vi, emacs, nano. You can also configure vscode on your laptop to edit and run files remotely on the pi via ssh.

Edit a test file then

```
g++ test.cc              compiles test.cc to a.out or a.exe on MSYS2 or
g++ -g test.cc           compile with debugging
./a.out                  run the program in the current
gdb a.out                debug the program
```

see the following link for many options for compiler and debugger commands:
https://docs.google.com/document/d/
1NcpADjQByWPtD3Fwv38ZLwA5jrlzAD0m4O0Z5euEYYU/edit?usp=sharing

Verify that g++, gdb work, and what version you are using

```
g++ --version
gdb --version
```

Anything beyond gcc 4.9 will probably work, but any reasonably modern system should really have gcc 9 or later.

Create a github id if you do not already have one, see github101. As a student you can get some good deals. https:
//drive.google.com/open?id=1kSknsnRPqe1-DSIx-r9alMr1drEHzgzzXxqCkdlnHGk
Create a public/private keypair using ssh-keygen. Also see github101 above. Go to your github account, click on your icon (top-right) settings. Click on ssh keys and add your public key. Go to the github repository for your section:

LAB_A https://github.com/sit-ece-cpe390/LAB_B_2020S https://github.com/sit-ece-cpe390/LAB_C_2020S LAB_D click on the green button that says clone or download, make sure it says ssh Clone the repo:

```
git clone git@github.com:sit-ece-cpe390/whatever-your-repo-is-called.git
```

Add your name to contributors.txt in the repo (edit the file) Then on the command line, add your change:

```
git add contributors.txt
git commit -m"added my name to the list"
git push
```

Find out from your TA what your functions are, and write them. check into the group repo. When everyone puts in their code, the program will build. Do a git pull, compile, and run. Test your own functions before commit/pushing. Do not submit code that does not compile.

23. Setting Up Raspberry Pi

Install Raspbian on the Pi ssh-keygen -t rsa on the raspberry pi just like you did on your laptop. Add this to your github account Set up your raspberry pi to work without a screen. If you have no ethernet, see professor Lu's technique (https://github.com/kevinwlu/iot) that will email you the IP address of the pi on bootup. Note that this requires you to setup first with a screen, and you must be on a network that the pi knows. So this works at Stevens if you set it up, but you would have to setup for home as well.

If you have an ethernet port on your laptop and you are running Windows, the instructions are in: http://carbonstone.blogspot.com/2014/02/connecting-to-pi-from-laptops-ethernet.html

If you have an ethernet port on your laptop and you are running Linux see Peter Ho's pissh.sh Setup is described on the repo:
https://github.com/PeterHo8888/RPi-Ethernet

Copy HW1 from the course into your repo and check in the empty framework. You will be given an assignment link. Click to create your private repo. Clone your repo onto the raspberry pi. Install development packages including gcc, gdb on your Pi so you can build the homeworks. you can verify with: g++ –version gdb –version

HW1 write the assembler code to add the two numbers in the add.s file build HW1 and run your code. Demonstrate to the TA that HW1 works.

24. C++ Practice, and Working in Teams using git

If you have not completed from lab 1, put your name into the contributors.txt file. Use the following git commands

```
git pull                                    #to get the most recent version
#edit the file and add your name
#tell git that the contributors.txt is to be added to the repo
git add contributors.txt
git commit -m"put my name in!"  #add comment to explain what you did
git push
```

if git push fails, it may be because someone else pushed before you. You may have to pull again, then push. If two of you edit the same line of the same file, you may have to manually edit the file to resolve the conflict if git cannot figure it out. In this case it's easy: DON'T DELETE THE OTHER PERSON'S NAME!!! ADD YOURS. It's not always that easy to figure out how to merge your changes. If you had to merge, there will be lines like this in the file:

```
<<<<<<
your name
======
Their name
>>>>>>
```

In this case the decision is quite simple, keep the other person's name and add your own, either above or below. Remove the lines with:

```
<<<<
====
>>>>
```

Then add the new combined file in with the following commands:

```
git add contributors.txt                    #add the new version again!
git commit -m"merge"                #you just combined the two versions
git push                                    #push it up to the server
```

For your individual component, write the following program to compute the 3n+1 or Collatz conjecture.

- read in an integer variable n from the keyboard.
- if n is even, divide by 2
- if n is odd, multiply by 3 and add 1.
- repeat this until n = 1.

Example: n=5 $\rightarrow$ 16, 8, 4, 2, 1
n=17 $\rightarrow$ 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
For the group component of your work write one scalar function and one array function (TA will assign you which function to write). main calls all the functions. Together you will write one big program that implements all the functions. This is more complicated than contributors.txt because now you are all writing separate functions, but they are all called from main. Try not to delete each other's work! Help each other so you all finish. Your entire lab is one team.

Each time you add code, you will have to use the sequence git add, git commit, git push. If someone pushed before you, you will have to git pull, manually edit the file if there is a collision, and then git add, git merge, git push.

25. Getting Started with C++ on pi, and ARM assembler

It's ok to get help with these labs, but you must CITE if you get help from anyone. Implement the functions in hw02 (C++). Note these are in a separate file, so you must compile each one, then link together. The makefile is already written so this is pretty easy, but you should look at it because later we may give you a program to build on your own. Practice generating assembler code using g++ -O2 -S hw02.cc. Look at the assembler code and see what the mangled function you are calling is called. Use the debugger gdb to debug hw02 once you have it built. Use the commands: gdb hw02

In order to see the assembler you need to type:

```
layout asm
```

In order to see the registers you need to type:

```
layout reg
```

In order not to have to type these commands every time, edit the file in your home directory: /.gdbinit For example, using emacs:

```
emacs ~/.gdbinit
```

Type the following commands into .gdbinit and save the file. From now on, gdb will start with these enabled.

```
layout asm
layout reg
```

so you can see the machine language and step through instruction by instruction using si and ni. Build Homework hw04, the first assembler homework. Recall that ARM binary calling convention is that first two variables are r0 and r1. Your job is to compute the answer and store into r0 since that is the value returned. You will also have to return from subroutine. Discover correspondence between C++ loops/if statements and ARM assembler. Write 4 functions in C++:

```
void count(int n);                  // count up from 1 to n using a for loop
void countDown(int n);   // count down from n to 0 using a while loop
int sum(int a, int b);   // add the numbers from a to b inclusive ie sum
```

For each of these 3 functions, use g++ to compile optimized code and record the assembler generated.

Note that the instructions will consist of compare (CMP) and branch instructions.

```
CMP     r0, r1  @ compare register r0 and r1, and set the Processor sta
CMP is followed by a branch instruction, one of the following:
BGT             @branch if r0 > r1
BLT             @ branch if r0 < r1
BLE             @ branch if r0 <= r1
```

There are many different branch condition codes. Find out how many there are by looking up the condition codes in the ARMv6 reference manual.
https://static.docs.arm.com/ddi0419/d/DDI0419D_armv6m_arm.pdf

Look for "condition codes". Hint: 4 bits are allocated to conditions, which should give you an upper bound on how many there can be.

There are 4 kinds of branch instructions. Bcc (BGT, BLT, etc) will add to the current PC and jump to somewhere else BXcc (BXGT, BXLT, etc) will also add to the current PC. Find out the difference in the manual and explain why there are two instructions.

You may begin to do hw05 in the lab, getting help from the TA if you need it, or you may do it for homework after it is covered in lecture. You must write four functions. Your code for the homework should be BETTER than can be generated by C++, because we will have an additional condition they do not know about. We will guarantee that out calls to the four routines will always execute at least once. In other words, we guarantee that the parameter n will never be such that the loop should not execute. For example:

```
void count(int n) {
   for (int i = 0; i < n; i++)
     f();
}
```

will generate code that branches out of the loop if n ¡ 0, but if you know that n is greater than zero, you know that this loop will execute at least once and you can eliminate that unneeded code.

26. Bit Manipulation

In this lab you will write code to do bit manipulations. You will practice with AND, OR, XOR (EOR on ARM) and BIC (Bit Clear) to set and clear bits. Then, you will use shift instructions to move those bits to desired locations. You must know the notation in both C++ and assembler (see section C++ Assembler Equivalence

You need to know how AND, OR, an XOR can be used to surgically operate on bits, and how we use shifting to move those bits to wherever they are needed.

Deliverables for this lab: You will be writing code to implement the following operations in ARM assembler. Ideally you should also know how to do this in C++ as well.

```
void setBit(int v, int n);     // set the bit of v at position n to true
void clearBit(int v, int n);   // set the bit of v at position n to fals
void toggleBit(int v, int n); // Invert the bit at position n
void replaceBits(int v, int start, int end, int replacement);
double choose(int n, int r);
void crypt(char msg[], uint32_t len, char key);
```

The following are the details:

How to set a bit. This can be done by starting with 1, which is a bit in the rightmost position, and shifting it left to the desired position.

```
1 << 0  (is still 1)
1 << 1 (is 2)
1 << 2 (is 4)
00000000000000000000000000000001
shift to desired place
00000000000000000000100000000000
or with the other number. The bit is now set (1).
Write function setBit in both C++ and ARM assembler.

2. clearing a bit. Again start with 1
00000000000000000000000000000001
shift to desired place
00000000000000000001000000000000
invert
11111111111111111110111111111111
and it with the other number. The bit is now clear (0).
```

The ARM has a shorthand, an instruction called BIC and which AND with NOT. Use that
Write function clearBit in both C++ and ARM assembler.

3. Flipping a bit. Same as setting a bit but instead of OR use XOR.
Write function flipBit in both C++ and ARM assembler.

4. Write the following function in ARM assembler:

int replaceBits(int v, int mask, int rep);

the function takes the value v, clears all bits that are zero in mask then replace
For example:
replaceBit(0x789ABCFF, 0xFFFFFF00, 0x12);

should zero out the last 8 bits specified in the mask so the value is
789ABC00
then replace the bits by the third parameter
789ABC12
Write this function in ARM assembler only

5. Build a single number from pieces shifted in. We ask you to buildColor(r,g,b) wh
0x00rrggbb where rr gg and bb represent the color value in hex. Use shifting in as

Write function clearBit in both C++ and ARM assembler.

5. Write an optimized choose(n,r) which computes n! / (r! * (n-r)!) but cancels the
In this lab you will practice reading and writing arrays of data. These will be pas

Read in an array and calculate checksum (sum the bytes, return the answer modulo 25

Calculate CRC32, which is much better than checksum. See the c-code which will be p

Encrypt data by xoring with a single-letter key. For example:

```
void crypt(char msg[], uint32_t len, char key);
```

print out the resulting text in c++ by casting each letter to an integer:

```
for (int i ...)
```

```
cout << (int)msg[i] << ' ';
```

then call crypt again with the same key to get back the original message. Print it out to be sure that the decrypted message is still correct.

27. The Bomb Defuser Project

It's ok to get help with these labs, but you must CITE if you get help from anyone.

In this lab you are given an executable file and asked to run it. There are 4 stages to defuse, each requiring the correct input to move on to the next stage. If you type in the wrong inputs, the bomb will blow up (it will print a message "boom.") You must figure out the correct inputs to defuse the bomb. You are encouraged to help each other, but you should know that each student will be given a different randomly created program. Therefore, while you may give your friends help, you must help them step through with the debugger and figure out their code, as you can't just copy the answers because each program is different.

To receive points for this lab, you must demonstrate to your TA that the bomb defuses when you put in the 4 inputs for your executable. This means running the application outside of gdb.

Answer the following questions.

What is the function of scanf?

How did you find out that the program is calling scanf?

A person trying to reverse engineer a program must step through the logic. Seeing symbols like scanf and gets is a serious clue to what it is doing. We can make it a LOT HARDER for people trying to reverse engineer our code by removing those clues. Search how to strip out symbols in a linux executable, and show the command you would use to remove them if you wanted to make it harder.

28. Benchmarking: Intel and ARM

In this lab, you are going to figure out the speed of many operations on both your Intel-based laptop and the ARM-based Raspberry Pi. In the process, you will have to write some code in both C++ and assembler, and hopefully learn about the performance characteristics of CPUs and memory in general.

This lab is designed to teach you how to measure how long code takes, and how to use your measurements to understand what is taking the time. We will be using only simple tools, the clock() function which measures elapsed times. Note that the function may produce different results on Windows, so you should verify that the units are the same on both your laptop and pc. Find out the following information and record in the spreadsheet the TA provides:

- The model of the CPU on your laptop
- The model of your Raspberry Pi
- The model of the CPU on your Raspberry Pi
- The clock speed of your Raspberry Pi
- The clock speed of your laptop
- The kind of memory on your laptop
- The amount of RAM in your laptop
- The amount of RAM in your Raspberry Pi
- The amount of L1 cache for your laptop
- The amount of L1 cache on the ARM chip in your Pi

Edit the benchmark.cc code and complete the benchmarks that are not done. Run the benchmark and record the results for the following 4 cases.

Intel without optimization	g++ −g −c f.cc g++ −g benchmark.cc f.o
Intel with optimization	g++ −g −O2 −c f.cc g++ −g −O2 benchmark.cc f.o
Raspberry Pi without optimization	g++ −g −c f.cc g++ −g benchmark.cc f.o
Raspberry Pi with optimization	g++ −g −O2 −march=native −c f.cc g++ −g −O2 −march=native benchmark.cc f.o

Make a spreadsheet comparing these times. You should be able to draw a conclusion for every question in the code, for example:

"multiply is 3x slower than addition on the ARM"

Note that the Raspberry Pi is a slower machine than your laptop, but not every function is slow by the same ratio. We expect to see certain operations be much slower than others. For example, since there is no integer divide, anything that might use integer division would be very slow compared with a PC.

Post your laptop speed and select results in the spreadsheet shared for your section. Each student will write one row in the shared spreadsheet.

29. Benchmarking, continued

Using only optimized code, figure out how to measure the speed of double precision floating point addition/subtraction, multiplication, and division on the PC and ARM. Note that you have some code that goes in a loop computing additions, multiplications, and it is your job to write whatever you need to isolate the numbers you need. Record the time taken for n operations of:

- integer addition
- integer multiplication
- double precision floating point addition
- double precision floating point multiplication
- double precision floating point division

Languages in the C family will in general turn any constant expression into a constant, removing the computation. For example, if you write:

```
int a = 5 / 3;
```

the compiler does not actually compute 5/3 at runtime. Instead, since both 5 and 3 are constants, it computes the answer at compile time and simply stores the number 1 into the int variable a. Floating point, however, is known to have problems because it does not obey the associative law. That is a + b + c != a + (b + c). Because of this, the C++ compiler is afraid to rearrange terms and will not change the order. You have to do it.

Two new functions have been added to benchmark.cc

The first is deg2rad. The second is grav. Your job is to look at the source code, compile, look at the assembler code, and figure out how you can compute the same values faster. To do this, see if you can rearrange terms to compute the same numbers but in a different order.

For this lab, you do not need to worry about non-optimized code. We know optimized code is faster, so use g++ -O2. Just compile your answers on PC and raspberry pi and compare. Then write equivalent functions that you optimize, make sure your answers are the same, and compare the times. Due to floating point roundoff error, the error will never be exactly the same, but if it's the same to 5-6 digits, that is certainly good enough.

	PC	ARM
Original deg2rad		
Your deg2rad		
Original grav		
Your grav		

30. Crack My Software

Dovsoft has wonderful software that counts the number of primes up to n. Unfortunately, the company wants to be paid, and the program will terminate with an error if you do not have the license file. You will be given an individual software randomly generated that checks for a license file. Your job is to first, step through the code to determine what must be put into the license file in order to satisfy the program. After creating your license file so the code works, you must then crack the software by patching the binary executable so that the code will work, completely ignoring the license file.

Somewhere in the code, there is obviously a test (perhaps multiple), something like:

```
if (badLicenseFile) {
    exit();
}
```

// Figure out the test(s) and create the license file.

After demonstrating your correct license file, you must patch your binary file to skip the check entirely. You will want to take a look at the ARM Architecture Reference Manual for ARMv7-A. AN EXAMPLE IS SHOWN ON THE NEXT PAGE.

```
Section A8.8 - Instruction listing
 You want to use A1 encoding
Section A8.3 - Conditional execution
This is for suffixes on things like your branch instructions (BLT, BEQ, etc)
cond = 1110 for non-conditional
Ex: SUB has cond = 1110
cond = other value from Table A8-1 on page 288 (Section A8.3)
Ex: SUBGT has cond = 1100
S is the same as the S in SUBS, ADDS, etc.  It sets the APSR flags.
This is probably not necessary for this lab because you are bypassing checks.  No n
This means you should set S to 0.
Rd stands for Register destination, Rs = Register source
imm# = Immediate (constant) value (with restrictions, see Example 2.8)
```

Hint: You most likely will want to replace conditional branches with unconditional branches or NOP instructions... Section A8.8.119 (NOP), Section A8.3 for Conditional Execution

To patch the binary, you can use the provided patching application in the repository with your executable. An example is shown below, but the executable also prints instructions. Example 1:

========EXAMPLE OF ENCODING OF INSTRUCTION==========

	cond opcode S Rn Rd Rm	hex
add r1, r5, r3	1110 00 0 0100 0 0101 0001 00000 00 0 0011	E0851003
sub r1, r2, r4	1110 ?? ? ???? 0 0010 0001 00000 00 0 0100	E??21004
ands r1, r5, r3	1110 00 0 0100 1 0101 0001 00000 00 0 0011	E0851003
addgt r4, r7, r8	???? ?? ? ???? 0 0111 0100 00000 00 0 1000	???74008

For the above instructions, cond = 1110 because they are not conditional (they always execute). The opcode varies by instruction. The registers in the instruction are encoded in the fields labelled Rn, Rd and Rm. You can see that instructions that set the flags, like ands have $S = 1$. imm5 and type are for optionally shifting Rm, which we are not doing

Let's say we want to patch a.out and have a resulting b.out with the above instruction. Assuming that the address of the instruction we want to replace this with is at 0x10570:

```
$ ls
a.out patch patch.txt

$ cat patch.txt
10570: e0851003

$ ./patch patch.txt a.out b.out
Patching 10570 with: e0851003

$ ls
a.out b.out patch patch.txt
```

```
    ldr r3, =\#385
    add r1, r5, r3
```

The ldr pseudo-instruction loads #385 from someplace in memory, so the number 385 isn't stored in the LDR instruction. Show the license.txt file, the code working, and the patched executable that skips the license check to the TA.

31. Gravitation Simulator Optimization

Similar to the simple function in benchmark.cc but this one is a full gravity simulator. We won't bother testing how well it works, though given real velocities and positions, it should satisfactorily predict the motion of planets for a while.

In this lab the only goal is to get it to compute faster, so you can build once with optimization enabled, save the results, and then try to optimize the code. Because debugging is so hard with optimization on, you might want to turn it off so you can see what's happening more easily. Your goal is to get it to run approximately the same but faster. Can you identify computations that are redundant that can be eliminated, yet the C++ optimizer won't do it? Look at the code, try to combine constants, eliminate redundant computation. If there is a division by x being done multiple times, can you pre-compute an inverse 1/x and multiply by that?

Remember, given roundoff error, you will not be able to get the code to produce exactly the same. In fact, your optimized result is probably producing better results. So don't worry about small differences. On the other hand if you get radically different results, then you probably changed the program in some way.

We realize that this code will be hard for you because it is object-oriented, but you do not have to write any object code really. You only have to optimize the numerical code. The one exception is the hint on magsq() but if you look at mag() you just have to copy that function and change what it computes. Remember, the goal here is to take working code and make it faster. You don't have to fully understand all the code, just walk through the main loop when we are stepping forward in time, watch what it does, and come up with your code to do it faster.

In order to debug, we recommend stepping through the main loop. You can always print:

```
p bodies[0]
p bodies[1]
```

These are the objects. Each one contains all the information including position, v (velocity) and a (acceleration). p.bodies[0].v

Compile and run the gravsim.cc program and record the results. You may save them to disk by redirecting standard out as follows: There are hints in comments in the code. Look at them and try to implement them.

```
./gravsim >log
```

Examine the code in gravsim and try to make it faster without changing the results radically. You can look at the assembler code, rewrite in C++ at first (easy and

quick). But if you notice something that the compiler cannot do, like understand that the value you are computing the square root of is not negative, then you will have to write some in assembler. The minimum requirement of this lab is achieving some reasonable improvement in time, and explaining in a comment in your code what techniques worked. For going beyond the minimum and writing an assembler routine you get 30

32. Disassembler

See the ARM 32-bit reference manual, page A3-2
https://drive.google.com/file/d/1m2ldsn3tmGhos86thB7i-rS4xP3rZmeu/view?usp=sharing

The code framework is in the group git: StevensDeptECE/CPE-390-labX where X is your lab group.

Your TA will assign you a task within the disassembler. Work together with other teams to calculate the shared values, and write the function that is your team's assigned task.

33. Bitmap

This project will require you to write methods for an object using ARM assembler that manipulate a bitmap and demonstrate that you can put all your skills of array and bit manipulation into play. You are allowed to work in teams of 2.

- Refer to the C++ bitmap specification. You must implement all the methods that are prototyped. Reading and writing bitmaps is handled in C++ using the STB library, so you do not have to know anything about how png or jpeg files work.

34. System Calls

- Look up the Unix kernel interface to files: man open
- Opens a file using level 2 I/O (open or creat). This is a call direct to the kernel.
- Write a program that writes 4 bytes (all zero) to the file and then closes the file.
- Now repeat the writes, writing integer 0, 1, 2, ... up to 10 million to the file. Use the Linux time command to benchmark the time it takes on the Raspberry Pi to write this file.
- Find out the blocksize of the drive on raspberry Pi
- Write a second program that has a buffer of a multiple of the buffer size. You can try 1x the buffer size. Write numbers into memory until the buffer is full, then write the entire buffer to disk. Then start over and write again. Use the Unix time command to measure this second program which should be much, much faster.
- Use the Unix cmp utility to verify that the two files are the same:
  cmp file1 file2

35. Buffer Overflow Exploits

[This exercise is not complete yet, but we have code showing how you can corrupt the stack and execute arbitrary code, which is a common exploit] Given source code exploit/ex1.cc, figure out how to bypass the check and execute the shell at line [???]. For the second exploit, you don't get the source code. Figure out how to bypass the check (it's not as easy this time) and execute the shell.

36. Arduino: Digital Output and Input

Setup the Arduino environment on your laptop. https://arduino.cc

Install Fritzing for drawing diagrams of your circuits (or Eagle and/or KiCAD if you are more ambitious). https://fritzing.org/download/

Eagle used to be the most popular "real" CAD package but is now owned by Autodesk and they are more restrictive (they charge).
https://www.autodesk.com/products/eagle/free-download

KiCAD is a similar high-end open source CAD package.
https://kicad-pcb.org/download/windows/

Use Eagle or KiCAD if you want to be able to design boards in the future, but make sure you get your lab in, so please install Fritzing regardless because it is easy to learn. To learn how to use Eagle or KiCAD, see youtube.

- Create an Arduino directory code under your repo. This is where all the code should go.
- Run the blink demo and make sure you can blink the internal LED
- Create a breadboard circuit with an LED and a resistor in series, at least 100 ohms.
- Plug the LED into an external pin that is NOT pin13, and demonstrate that you can blink it.
- Create 3 LEDs and blink them sequentially, for 100ms each.
- Digital Input. If you have a partner, one of you can do a pulldown resistor, and the other can implement a pullup resistor. Make sure you understand both and that your code matches your circuit.
- Create a pushbutton and pulldown/pullup resistor. Write a loop that waits until the button is pressed. Then blink the light with a period of 0.2 seconds with a duty cycle of 50%.
- Show your circuits and diagram to the TA to get credit for the lab. Save and submit the code and a png of your CAD diagram. Write code that blinks the light at 20Hz as long as the button is held.

37. Power Transistors, H-bridge controller

For this lab work with a partner. Test the voltage of your battery pack and record it open circuit. Test the motor using the 6V battery packs and measure the current. Write down the model number of the power transistor you are given for turning on the motor. Look up the spec sheet for the power transistor and fill in the following:

- Is it N-channel or P-channel?
- Maximum source-drain voltage:
- Explain the significance of maximum source-drain voltage.
- Maximum gate-source voltage:
- Explain the significance of maximum gate-source voltage.
- Is this power transistor suitable for turning on the motor? If not, see your TA:
- What factors could make the power excessive on your power transistor causing it to burn out?
- Draw a circuit for the Arduino to control the motor using a motor, a 6V battery pack and a power transistor.
- Identify which pins on your MOSFET are ground, source, drain.
- Build a manual circuit where you control the gate by plugging it into either Vcc or ground.
- What would happen if there is no common ground?
- Turn the power transistor on by connecting a wire from gate to V_{CC}. The motor should start turning.
- Feel the power transistor with your finger. If it gets warm unplug immediately and debug, or ask the TA for help.
- IF YOU FRY A POWER TRANSISTOR, TELL THE TA. WE WILL NOT EXECUTE YOU (promise). Do not put the dead transistor back into the supply, forcing some other student to work with a dead one. You wouldn't want that done to you, right?
- Pull out the wire, the motor may continue to turn or stop. Why? Explain the term "floating"
- Connect the wire to ground. The motor should turn off. Check again for power transistor temperature and SHUT OFF IMMEDIATELY if it's too warm.

- Next you can design a circuit where a digital output on the Arduino controls the gate of the power transistor. The Arduino should only be connected to the gate, with ground on the Arduino joined with ground on the battery to establish a common ground.

Write a function on the arduino to set the power to the motor that takes the following parameters:

```
void power(int pin, int percent, int duration);
```

where the pin selects which pin to pulse, percent represents a percentage from 0 (off) to 100 (full power), and duration is in milliseconds. Test with a loop like the following:

```
void setup() {
  power(11, 100, 3000); // set pin 11 to 100\% power for 3 sec
  power(11, 50, 3000); // set pin 11 to 50\% power for 3 sec
  power(11, 20, 3000); // set pin 11 to 20\% power for 3 sec
```

Determine the duty cycle percentage for your motor at which the motor stops working. Note that this will not be the same for every motor, particularly because the batteries might be more or less fresh for your group.

Draw a schematic for H-bridge controller using P-type MOSFETS for the upper voltage, and N-type for the lower. If P-type power transistors are available, build the h-bridge controller. If not, you will have an opportunity to do so (optional) later in the course for extra credit.

38. Interrupts

Use an interrupt to demonstrate real=time response to an event while something else is happening.

On the Arduino, write a loop that blinks a light at 10 Hz with 50% duty cycle. This would normally mean that nothing else can happen, because while you are sleeping with delay, you are not computing anything else. You could write code that would check for the button being pressed, but that would be complicated, and if you only check at one point in the loop, there is an entire tenth of a second during which you would not detect the button being pressed. Instead, write an interrupt that responds to the event of the button being pressed and turns on the light, and another that turns off when the button is released. Turn on pulse width modulation on a digital output, and set the output low (10). Since 10/256 is small, there will be a series of short pulses. Look at the output with an oscilloscope and determine the frequency _____ and the duty cycle _____ of the signal. Plug the digital output into the input you are using as an interrupt. Now the computer will effectively have a button pressed quite fast. How many times per second is the button being pressed? _____

39. Using the DAC: Waveforms on the Arduino

Test the oscilloscope probes using the square wave output on the scope. Just verify that the wave is square. If not, review with your TA to make sure you remember how to calibrate the probes.

Using the DAC, create a sine wave of 440Hz. View on the oscilloscope.

Create a wave that is sin(440Hz) + sin(880Hz) and view on the oscilloscope.

Now create a wave that is 100Khz. This is much harder because this is pretty fast with respect to the speed of the computer. Instead of calculating sine, pre-calculate the waveform as integers and write code to output the wave directly.

Measure the period of the wave on the oscilloscope and determine the frequency of the wave. How close can you get to 100kHZ?

Build the op-amp circuit and audio amp shown in the following circuit and attach the small speaker to the output. Is the circuit capable of driving the speaker? Make a chord with A and C and demonstrate for the TA. Write out the schematic in your CAD package and submit as part of the lab.