CMPE 120 - Computer Organization and Architecture

- Learning Objectives
- Reference Material
  - Tools
  - Books and Online Resources
- Basics
  - Number Systems
  - Python Number Converter
  - Storage Units
  - Gates
- Programming Languages
  - How program is compiled in C
- Assignments
  - Git
  - x86 Dissassembly
- CPU Arhitecture
  - CPU Architecture Basics
- Operating System
  - System Call

- How OS launches a program
- Fundamentals of an OS

Learning Objectives

CLO

Upon successful completion of this course, students will be able to:
1. Understand digital logic and how it is used to build a computer system.
2. Explain how CPU functions to run a software program.
3. Develop assembly programs to control the operation of the CPU.
4. Understand the format of instructions and their operations.
5. Understand the role of the other components of a computer system such as buses and memories and how
they work together.

Lecture plan:

- Number System (CLO 1)
  - Binary
  - Hex
  - ASCII
- Units (CLO 1)
  - Kibibytes, Kilobytes etc.
- Hardware Architecture (CLO 2)
  - Data bus
  - Address bus
- Microcontrollers
  - On-chip peripherals review
- Compilers and Programming Languages (CLO 3, CLO 4)
  - Compiled vs. Interpreted languages
  - C compiler
  - Hands-on with a compiler

Reference Material

Reference Material

Tools

In the past, we needed to setup special tools on a local computer (i.e.: your laptop) to test software. In the modern era, the advanced made by software developers have led us to several tools we can use to understand a machine's instruction set.

1. Python Interpreters
    - https://www.programiz.com/python-programming/online-compiler/
    - Python interpreter
2. Assembly and Emulators
    - WeMips
    - Gobolt
3. Logic Emulators
    - Logic.ly/demo
    - https://circuitverse.org/simulator

Reference Material

Books and Online Resources

Really awesome book from Robert Plantz:

- Introduction to Computer Organization

Books:

- Structured Computer Organization 6th Edition
- Computer Organization and Design

Online Resources

- https://thinkingeek.com/arm-assembler-raspberry-pi/
- https://azeria-labs.com/writing-arm-assembly-part-1/

Basics

Number Systems

Number Types

The number system holds significance in terms of writing and expressing code to a computer, typically in a programming language. Note that we (as humans) do not use hex or binary numbers that much outside of the computer science domain. For example, we don't walk into a supermarket and read prices in binary such as `$0x10` :)

Often times in programming, we need to express numbers more quickly, and we might say int `x = 0x10000000` to quickly indicate 32-bit value with `bit31` set to 1. Notation `x = 0x10000000` is easier than writing `x = 268435456` which would be more cryptic for a programmer to realize the significance of because the reader of the programming code will not be able to quickly realize that it is specifically setting `bit31` to value of `1`.

Decimal

Typical numbers we are familiar with are decimals which are technically "base 10" numbers. So an ordinary number that we may be aware of such as 123 can be written as 123_{10}.

The number 123 could also be written as:

$1*10^2 + 2*10^1 + 3*10^0$ which is equal to `100 + 20 + 3 =`123_{10}

Binary

Binary numbers are always 1s and 0s only. Similar to decimal numbers, binary numbers increase in powers of 2, rather than powers of 10. Binary numbers are written by with the "0b" notation, such as 0b1100

For example, binary 101 or `0b`101 can be written as:

$1*2^2 + 0*2^1 + 1*2^0$ which is equal to $4 + 0 + 1 = 5_{10}$

Hex

One digit of a hex number can count from 0-15, but since we have to represent the hex number using a single character, the numbers 0-9 are usual numbers, and the numbers 10-15 are represented by A, B, C, D, E, F

Where decimal is a power of 10, and binary is power of 2, hex numbers are powers of 16. Hex numbers are written with the "0x" notation, such as 0x10.

For example, hex `0x12` can be written as:

$1*16^1 + 2*16^0$ which is equal to $16 + 2 = 18_{10}$

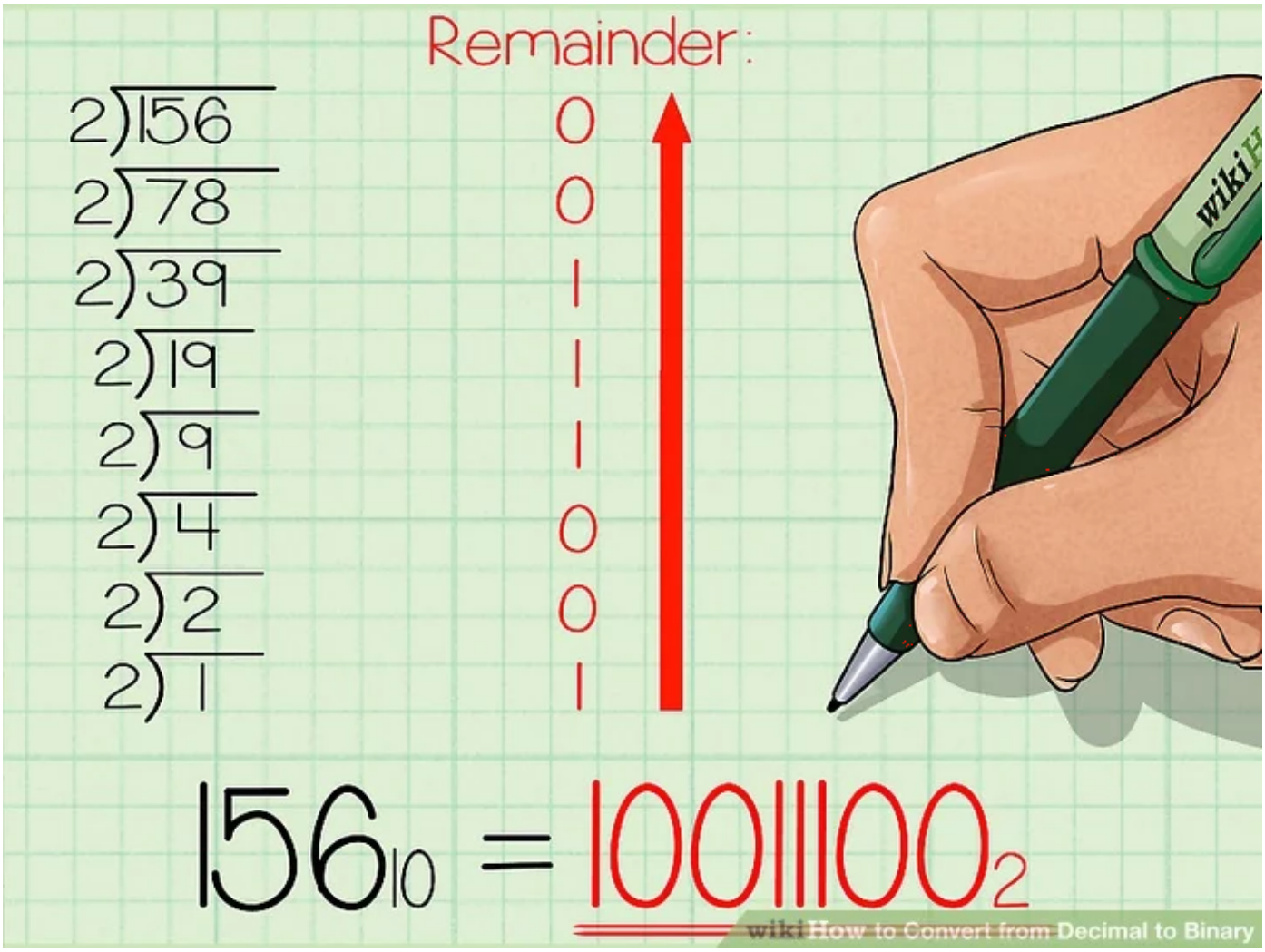
As another example, hex `0xC5` can be written as:

$12*16^1 + 5*16^0$ which is equal to $192 + 5 = 197_{10}$

Exercises

Decimal to Binary

Decimal (base 10) numbers can be converted in a couple of different ways as described here. One of the methods is to continue dividing by 2 and note down the remainder as described in the image below. The article above also describes a potentially faster method of conversion so be sure to read it!



Please try converting the following to binary:

1. 125
2. 255
3. 500

Decimal to Hex

Decimal to hex is similar to Decimal to Binary except that we are dealing with powers of 16 rather than powers of 2.

My favorite method of conversion from decimal to hex is to first convert the number to binary. For example, let's start with a large number such as 23912. We can use the Decimal to Binary method to convert this first to binary:

- 23912_{10}

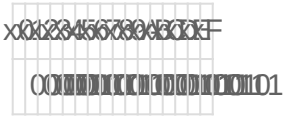
- `0b101110101101000`
- Split it up to nibbles:
  - `0b101 1101 0110 1000`
- Then use the lookup table listed in Hex to Binary:
  - `0x5D68`

Please try converting the following to hex:

1. 125
2. 255
3. 500

Hex to Binary

The following table can be utilized to convert hex to binary very instantly:



First row is HEX, and the second row is binary. For whatever hex number we wish to convert, we simply locate its equivalent in binary. For instance, if we wish to convert `0x5` to binary, it is `0b0101`, and `0xA5` would be `0b1010.0101` as you can convert one "nibble" (4-bits) at a time.

Let's take another example to convert `0x1BF` to binary; simply break it down by "nibbles":

- `0x1` --> `0b0001`
- `0xB` --> `0b1011`
- `0xF` --> `0x1111`
- Answer: `0b0001 1011 1111`

Please try converting the following to binary:

1. 0x55
2. 0x125
3. 0x40000000

Hex to Decimal

For Hex to Binary, we used a lookup table as a "cheat code" :). For Hex to decimal, it would be easier to re-write the numbers as powers of 16. For example, to convert `0x1BF` to decimal, we can break it down

to:

- `0x1` --> `1 * 16`^{2} --> `256`
- `0xB` --> `11 * 16`^{1} --> `176`
- `0xF` --> `15 * 16`^{0} --> `15`
- `256+176+15 = 447`

Please try converting the following to decimal:

1. 0x55
2. 0x125
3. 0x40000000

Basics

Python Number Converter

Generally speaking, practiced skill cannot be easily forgotten. It is far better to go through the process and practice converting a number, rather than to memorize the process.

Before we get started, have a look at the Tools Page to get started with a Python Interpreter we could use for this exercise.

Number to Printable Hex

```
def nibble_to_ascii(nibble: int) -> str:
  """
  This is a comment
  Input: Nibble (4-bits)
  Output: Single character HEX as a string
  Example: Input = 10, Output = 'A'
  Example: Input = 8,  Output = '8'
  """
  table = ['0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F']
  return table[nibble]
def to_hex(number: int) -> str:
    """
    This is a comment
    Input: Number (integer)
    Output: String
    Example: Input = 43605, Output = "0xAA55"
    """
    answer = ""

    # Forever loop
    while True:
```

```
        # Integer divide using the // operator
        quotient = number // 16
        # Get the remainder using the % operator
        remainder = number % 16

        # Accumulate result
        answer = nibble_to_ascii(remainder) + answer
        # Set the number we need to use for next time
        number = quotient

        # We break the "loop" when division turns to zero
        if (quotient == 0):
            break

    return "0x" + answer
print(to_hex(123456789))
print(to_hex(0b1010101))print(to_hex(0xDEADBEEF))
```

Exercise

Write a function `to_binary()` that takes a number, and returns the string equivalent version of the number in binary. You can borrow the template above of `to_hex()` function and most of the logic might be similar except that we would be dividing number by 2 rather than 16.

Basics

Storage Units

I fear that most of the technical articles on the Internet misinterpret some of the common storage units.

THERE'S BEEN A LOT OF CONFUSION OVER 1024 vs 1000, KBYTE vs KBIT, AND THE CAPITALIZATION FOR EACH.

HERE, AT LAST, IS A SINGLE, DEFINITIVE STANDARD:

SYMBOL	NAME	SIZE	NOTES
kB	KILOBYTE	1024 BYTES OR 1000 BYTES	1000 BYTES DURING LEAP YEARS, 1024 OTHERWISE
KB	KELLY-BOOTLE STANDARD UNIT	1012 BYTES	COMPROMISE BETWEEN 1000 AND 1024 BYTES
KiB	IMAGINARY KILOBYTE	1024 √-1 BYTES	USED IN QUANTUM COMPUTING
kb	INTEL KILOBYTE	1023.937528 BYTES	CALCULATED ON PENTIUM F.P.U.
Kb	DRIVEMAKER'S KILOBYTE	CURRENTLY 908 BYTES	SHRINKS BY 4 BYTES EACH YEAR FOR MARKETING REASONS
KBa	BAKER'S KILOBYTE	1152 BYTES	9 BITS TO THE BYTE SINCE YOU'RE SUCH A GOOD CUSTOMER

This article does a good job at clearly providing the relevant information:

> “ A **kilobyte** is made up of either 1,000 or 1,024 bytes. This distinction can be a little

> tricky and has to do with the difference between binary math (which computers rely on) and base-10 math (which most humans use in daily life). In practical terms, both definitions of kilobyte are used. In some cases, a distinction will be made between a kilobyte (1,000 bytes) and a kibibyte (1,024 bytes), though this is less common.

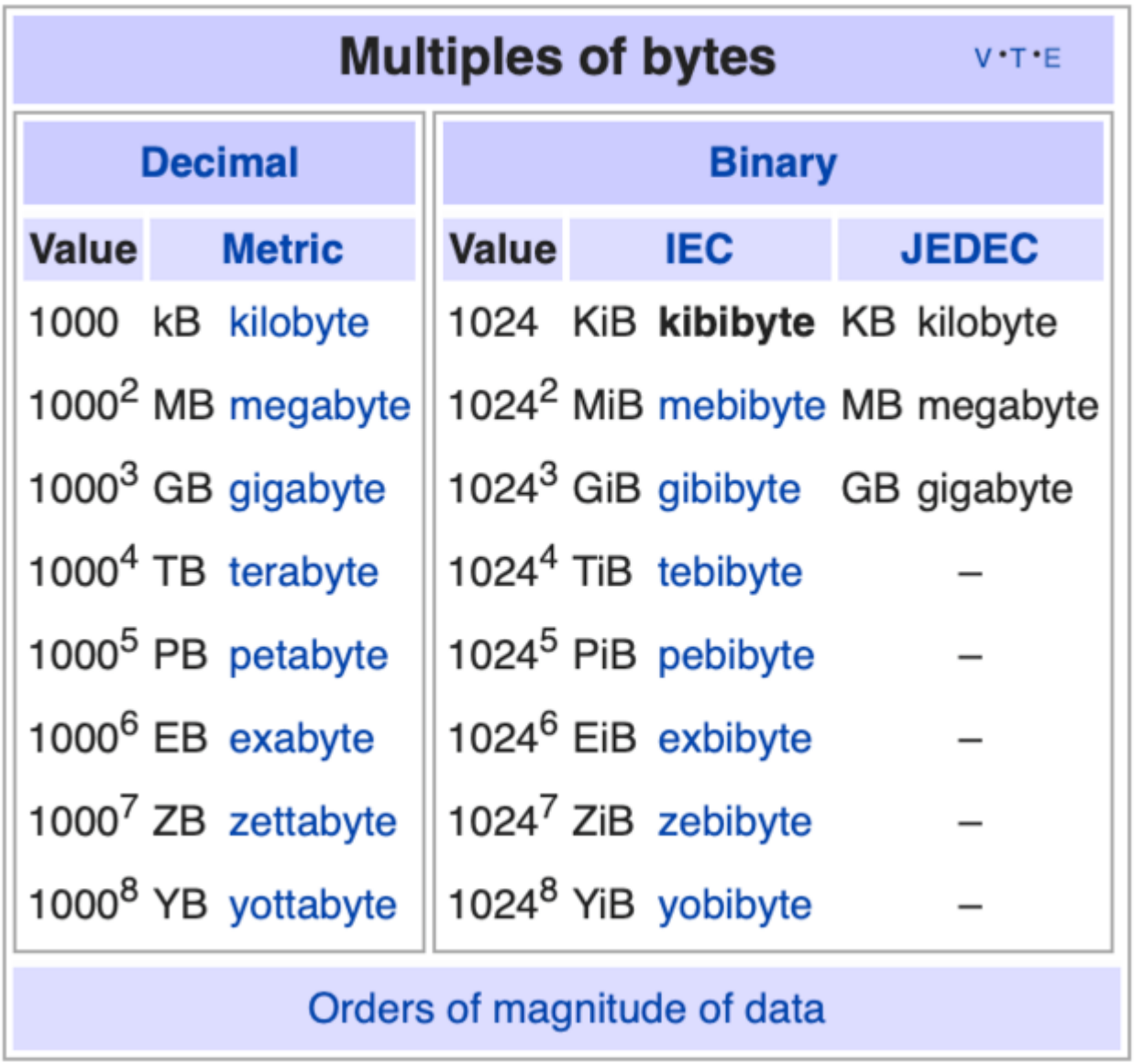
The Real Story

Apart from the funny picture above (Baker's Kilobyte?), the real story can be uncovered by referencing the picture below. Thanks to **this original article** that does a great job at providing the valuable information.

Decimal Prefix (SI)	Value	Value (1000)	Binary Prefix (IEC)	Value	Value (1024)
kilo (k)	10^3	1000	kibi (ki)	2^{10}	1024
mega (M)	10^6	1000^2	mebi (Mi)	2^{20}	1024^2
giga (G)	10^9	1000^3	gibi (Gi)	2^{30}	1024^3
tera (T)	10^{12}	1000^4	tebi (Ti)	2^{40}	1024^4
peta (P)	10^{15}	1000^5	pebi (Pi)	2^{50}	1024^5
exa (E)	10^{18}	1000^6	exbi (Ei)	2^{60}	1024^6
zetta (Z)	10^{21}	1000^7	zebi (Zi)	2^{70}	1024^7
yotta (Y)	10^{24}	1000^8	yobi (Yi)	2^{80}	1024^8

So while most people might misinterpret "kilo" as 1024 when it comes to storage units, the right way is thus "kibibytes". It would be an interesting conversation to discuss kibibytes as most people may not be aware, and this would make you look incredibly smart (and correct) :)

Here is another great image for reference:



Multiples of bytes V·T·E

Decimal			Binary				
Value		Metric	Value		IEC	JEDEC	
1000	kB	kilobyte	1024	KiB	**kibibyte**	KB	kilobyte
1000^2	MB	megabyte	1024^2	MiB	mebibyte	MB	megabyte
1000^3	GB	gigabyte	1024^3	GiB	gibibyte	GB	gigabyte
1000^4	TB	terabyte	1024^4	TiB	tebibyte	–	
1000^5	PB	petabyte	1024^5	PiB	pebibyte	–	
1000^6	EB	exabyte	1024^6	EiB	exbibyte	–	
1000^7	ZB	zettabyte	1024^7	ZiB	zebibyte	–	
1000^8	YB	yottabyte	1024^8	YiB	yobibyte	–	

Orders of magnitude of data

Based on the image above, the following should be used using capital letter first, then lowercase i and then finally capital B for bytes.

- KiB
- MiB
- GiB
- TiB
- PiB
- etc.

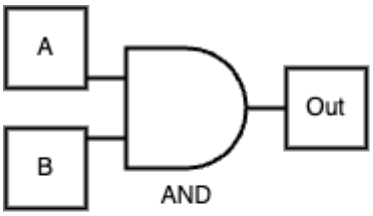
Reference Articles

- https://danielmiessler.com/blog/the-difference-between-kilobytes-and-kibibytes/
- https://study.com/learn/lesson/data-storage-units-kb-mb-gb-tb.html
- https://ozanerhansha.medium.com/kilobytes-vs-kibibytes-d77eb2ff6c2a

Gates

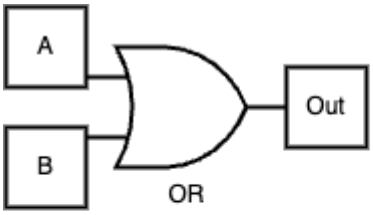
Diagram link

AND



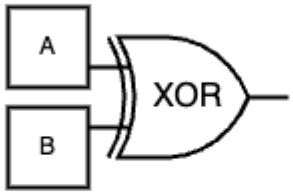
A	B	AND	NAND
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

OR



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

XOR



Equation:
(A * -B) + (-A * B)
(-A + -B) * (A + -B)

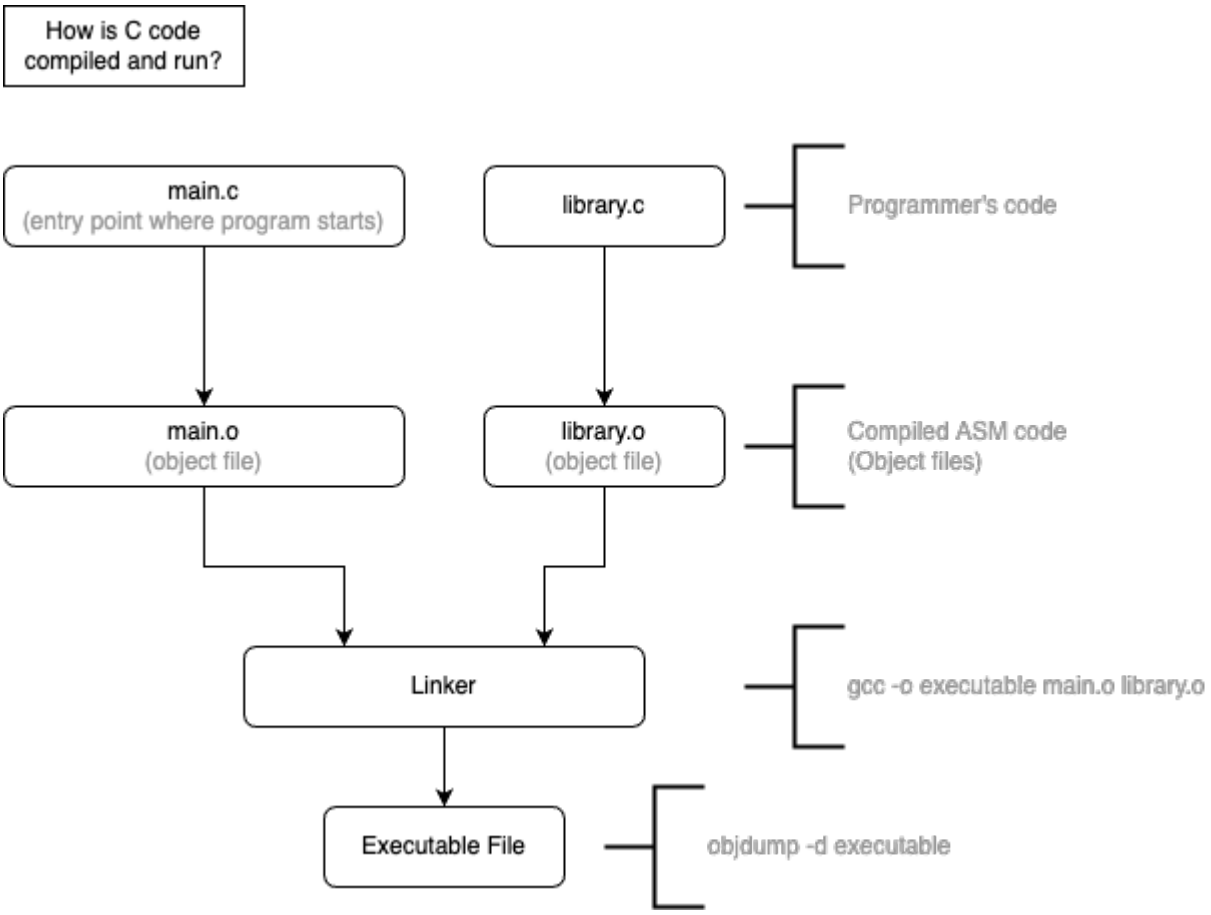
A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Programming Languages

Programming Languages

How program is compiled in C

Assuming that there are two files to be compiled in C, the overall flow to yield an executable is the following:



The commands used to compile the two files above are:

```
gcc -c library.c
gcc -c entry_point.c
gcc -o executable library.o entry_point.o
# Run the program:./executable
```

Assignments

Git

This is definitely not an exhaustive tutorial about learning Git... Google would be better to reveal several great tutorials about Git. What we focus on instead is a simplistic workflow about publishing a "Pull Request" in Git.

What is Gitlab?

Gitlab provides services that allow hosting your project on a remote repository and provides additional features that help in continuous integration and deployment. Such as code sharing, code review, and bug tracking.

Part 0: Setup Gitlab account

For better or worse, we have decided to use Gitlab.com for the repository. You are also required to use this Gitlab repository because that keeps the entire class aligned to a single server type and reduces fragmentation while increasing the efficiency of the teacher and the ISA team.

For this part, establish your Gitlab.com account.

How to set up a Gitlab account?

1. Go to GitLab.com and create an account.
2. Sign in to GitLab.

In addition, also install Git to your machine such that you can successfully execute the `git` Commands from a terminal.

1. Download git from GIT_Install and install git.
2. Check git is installed on your system by using the `git --version` command in the terminal.

Part 1: Create Project

Use the Gitlab UI from their website to create a new project. After you setup a new project, you will clone the project on your laptop.

Change project visibility

Set the visibility to public such that we can access your repository conveniently.

1. Go to your newly forked project's **Settings**
2. Change **Visibility Level** to Public

Part 2: Clone Repository

The first thing you want to do before you init is to add a project on the Git website to see the "Setting up a new Git repository" section. If you have a folder with code that is not on Git, and you wish to put it on the Git server, then you need to initialize Git into your folder. This creates a `.git` folder, and the current directory is now a Git repository. The `.git` folder contains Git information such as branches. Initializing your folder is local to your computer and does not yet upload onto the server.

For this assignment, we will instead clone a repository rather than initializing the repository. For the repository you created, you can "clone" it into a directory and start working on it. Cloning it will download the entire repository as well as a `.git` folder. Note that the clone is different from "pull". This will be explained later. Just use this command once at the beginning of the project unless you want multiple folders.

```
# Downloads entire repository to current directory
$ git clone <repo>
# Downloads entire repository to selected directory$ git clone <repo> <directory>
```

Part 3: Branch Workflow

The process of checking-in new code to your forked repository will involve "Branch Workflow". There are actually a number of ways to contribute code to your repository, and the branch workflow is just one of them that we will choose to use.

We are not going to discuss that in detail because it is already captured well in **this awesome article**.

We will summarize the process that you will use to do this. The `$` indicates the commands you should try.

```
# See what is going on
$ git status
On branch master
# Create a new "branch" of code to work on
# You can use any name, and feature/foo is just a convention
$ git checkout -b feature/hw1
Switched to a new branch 'hw1'
# Add or modify a file we want
$ touch file.txt
# Tell git to add it to be committed
$ git add file.txt
# Check what is going on
$ git status
On branch feature/hw1
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new file: file.txt
# Commit the change with a message
$ git commit -m "added file.txt"
[feature/hw1 5f76839] added file.txt
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 file.txt
# Check what is going on
$ git status
On branch feature/hw1nothing to commit, working tree clean
```

Part 4: Merge Request (MR)

The typical name of a request to merge code is called a "Pull Request" or a "Merge Request". This is the chance to review the code and merge the code. In the end, `Part 3` you have a branch that only exists on your computer. In case you lose your computer or your storage device dies, then you **will lose**

any work even though you have "committed" a change.

The distinction is that a commit only commits to your storage device, but does not send the data or the branch to the Git server. To actually push the code to the Git server, simply type `git push origin head`.

```
$ git push origin head
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 262 bytes | 262.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
remote:
remote: To create a merge request for feature/gpio_blinky_in_periodics, visit:
remote:   https://gitlab.com/sjtwo-c-dev/sjtwo-c/-/merge_requests/new?merge_request%5Bsource_branch%5D=
remote:
To gitlab.com:sjtwo-c-dev/sjtwo-c.git
 * [new branch]      head -> feature/gpio_blinky_in_periodics
```

This command will generate a URL for you, so copy and paste this URL to your web browser. For example, the URL above is:
`https://gitlab.com/sjtwo-c-dev/sjtwo-c/-/merge_requests/new?merge_request%5Bsource_branch%5D=feature%2Fh`

This will lead you to generate your "Merge Request". At the end of the web page that loads, click on "Submit Merge Request". At this point, you can view the changes, get feedback from others, and if the code looks good, you can then merge the code. But wait ... rarely will you be able to merge code without iterating and revising it, and that is what Part 5 is for.

Part 5: Revise an MR

Granted that you have an MR already out there, and you have got feedback from others, this section will teach you how to revise or amend your code.

```
# Modify any code
# In this case, we will dump 'hello' to our previously committed file: file.txt
$ echo "hello" >> file.txt
```

```
# Check what is going on
$ git status
On branch feature/hw1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
modified:   file.txt
# Add the file we want to re-commit (another commit on top of previous)
$ git add file.txt
$ git commit -m "Added hello to file.txt"
# Update the remote branch and the Merge Request$ git push origin head
```

After the `git push` command, your MR will be updated on the browser. This way, you can continue to revise your MR per the suggestions of other people. When you are satisfied with your MR, you can seek approval and officially hit the Merge button on the Gitlab.com webpage.

Part 6: Final Step

After you have merged your MR, it is time to go back to the master branch and grab the latest changes. Other users may have merged their code also, so pulling the latest master branch is going to get you the latest and greatest code.

```
# Go to the master branch
$ git checkout master
# Pull the latest master$ git pull origin master
```

Part 7: Going beyond . . .

There is of course A LOT more to Git, but once you master the basics, you can then Google your way through the rest of the world you will face such as:

- Handling merge conflicts
- Check out other people's branches

Rebase on the latest master branch.

```
$ git status
# Assume that you are on your feature branch
$ git checkout master
$ git pull origin master
# Go back to the previous branch you were working with (feature)
$ git checkout -
# Apply our commits to the latest master$ git rebase master
```

Part 8: Steps to create MR for Lab Submissions

The process of checking-in new code to your forked repository will involve "Branch Workflow" as explained in **PART 3**. The following steps will help you to add new code/files for each of your lab submissions.

- Goto cmd OR terminal OR git bash. CD to the location of the cloned project(cd sjtwo-c/projects/lpc40xx_freertos/l5_application) and run the following commands.

```
 # You can use any name, it's better to use lab with the number as a branch name.
 # such as lab1,lab2
$ git checkout -b lab1
Switched to a new branch 'lab1'
# Add or modify files as per the given lab assignment
# for example lab 1 requires two files
$ touch lab_multitasks.c
$ touch lab_multitasks.h
# Check what is going on
$ git status
On branch lab1
Untracked files:
  (use "git add <file>..." to include in what will be committed)
lab_multitask.c
lab_multitask.h
```

```
# Tell git to add it to be committed
$ git add .
# Check what is going on
$ git status
On branch lab1
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
	new file:   lab_multitask.c
	new file:   lab_multitask.h
# Commit the change with a message
$ git commit -m "added lab1 files"
[lab1 e88f23d] added lab1 files
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 projects/lpc40xx_freertos/l5_application/lab_multitask.c
 create mode 100644 projects/lpc40xx_freertos/l5_application/lab_multitask.h
# Check what is going on
$ git status
On branch lab1
nothing to commit, working tree clean
# Update the remote branch and the Merge Request
$ git push origin head
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 262 bytes | 262.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
remote:
remote: To create a merge request for lab1, visit:
remote:   https://gitlab.com/sjtwo-c-dev/sjtwo-c/-/merge_requests/new?merge_request%5Bsource_branch%5D=
remote:
To gitlab.com:sjtwo-c-dev/sjtwo-c.git
 * [new branch]      head -> lab1


 # Assume that you are on your lab branch
 # To comeback to master branch
```

```
$ git checkout master
```

- Create a merge request for each lab and use the merge request URL for your lab submissions.
  - Please follow **PART 4** to generate and submit "Merge Request" on Git.
  - After submitting a merge request you will receive a new URL on the browser. Use that URL for your canvas submission.
- Follow the same steps for creating the next lab branch(such as hw1), add new files to the hw branch(such as `hw.py`), and create a merge request for the submission after completing your GPIO driver.

x86 Dissassembly

Purpose of this assignment is to reverse engineer x86 assembly language of C code.

```
#include <stdio.h>
#include <stdlib.h>
__attribute__((noinline))
int sum(int a, int b) {
    return a + b;
}
__attribute__((noinline))
void print_the_value(int value) {
    printf("%d", value);
}
int entry_point() {
    int a = rand();
    int b = rand();
    int result = sum(b, a);
    print_the_value(result);}
```

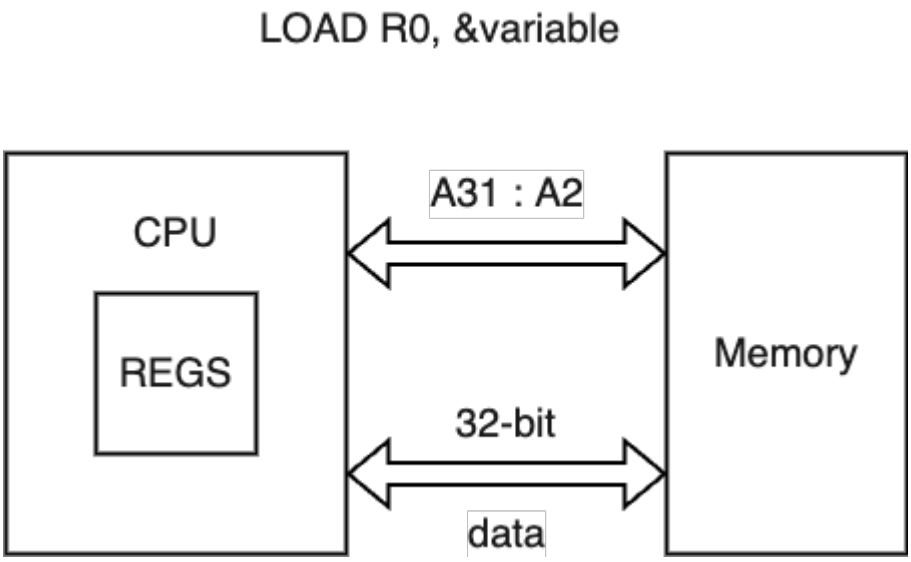
Problem:

1. Copy and paste the code below to https://godbolt.org
2. Use the following compiler for the ASM code generation: `x86-64 gcc 12.2`
3. Under the compiler options, use `-Os`
4. Explain each and every single line of the ASM program
    - You do not have to explain the C code
    - Attempt to provide the WHY rather than the WHAT. For example, do not just say "Moving R1 to R0" but state that "Moving R1 to R0 such that we can pass that as the first parameter to printf"

CPU Arhitecture

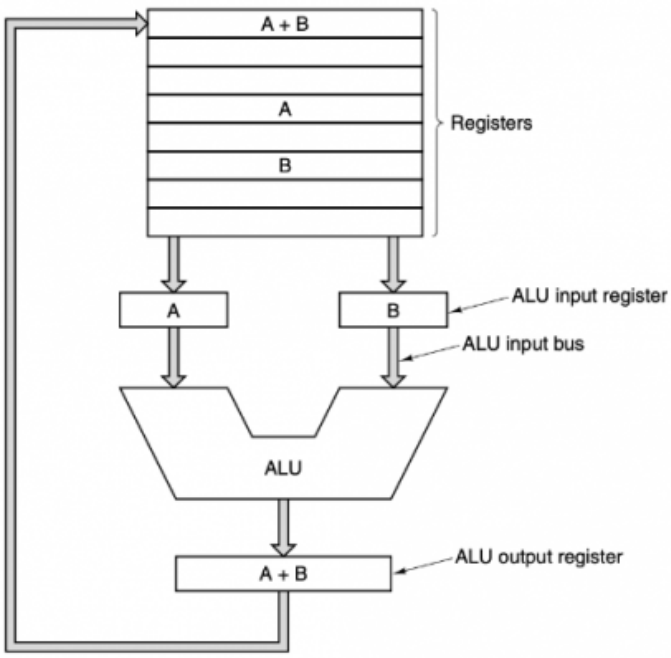
CPU Architecture Basics

Physical layout depiction of a CPU:



Note the following:

- CPU (ALU) is only connected to Registers
- Registers are connected to memory
- CPU can process an instruction to add to numbers but this operation can only be applied to registers
    - We cannot apply mathematical operations on the memory itself



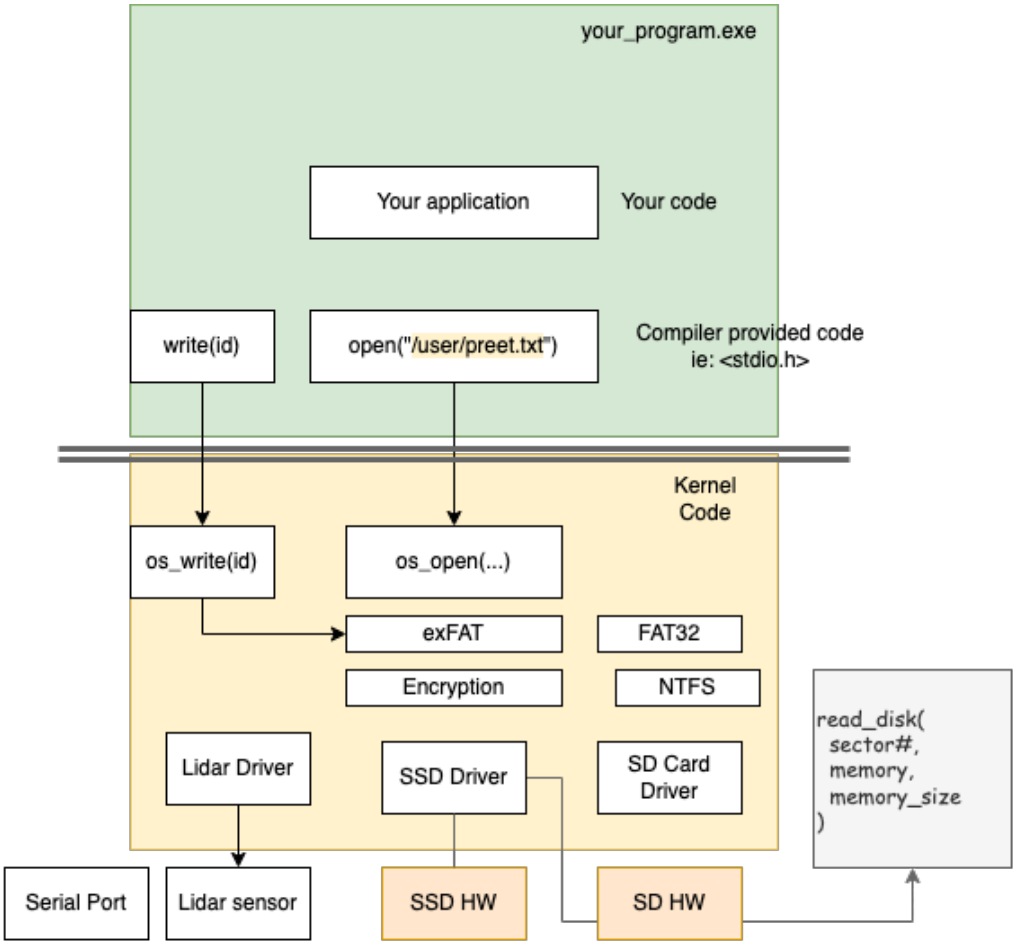

```
add r0, r1, r2  # r0 = r1+r2
add C, A, B
```

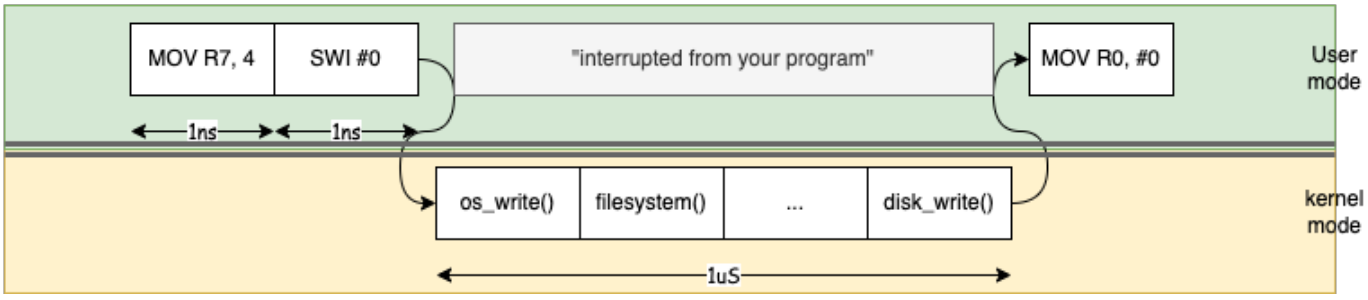
Registers

- Purpose of Registers

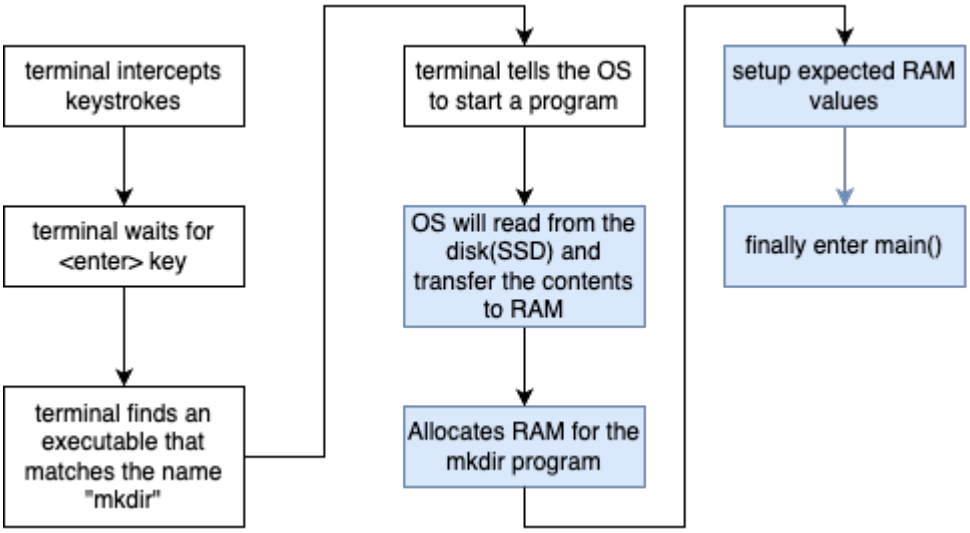
Operating System

System Call


your_program.exe
Your application
Your code
write(id)
open("/user/preet.txt")
Compiler provided code
ie: <stdio.h>
Kernel
Code
os_write(id)
os_open(...)
exFAT
FAT32
Encryption
NTFS
Lidar Driver
SSD Driver
SD Card
Driver
read_disk(
sector#,
memory,
memory_size
)
Serial Port
Lidar sensor
SSD HW
SD HW


MOV R7, 4
SWI #0
"interrupted from your program"
MOV R0, #0
User
mode
1ns
1ns
os_write()
filesystem()
...
disk_write()
kernel
mode
1uS

How OS launches a program



Fundamentals of an OS

What is an OS?

A good example can be reference at section 6.5 of Structured Computer Organization.

Fundamentally, an OS provides services:

- File services (open, read, write, close etc.)
- Networking services (open, read, write, close internet socket)
  - Berkeley socket interface
- Multi-tasking services
  - Creating multiple threads
  - Dictating priorities for threads

Interrupt

An interrupt is an asynchronous function call that can interrupt normal flow of a program.

```
// HW and OS work together to invoke this function
// whenver a keyboard key is pressed
// Regardless of where you are at your_program()
// this function can be invoked ascynhronously
void interrupt_keyboard(void) {
}
// Thread that never exits
// This calls all the sub-functions synchronously
void your_program(void) {
  while (forever) {
    check_for_brake_pedal();
    actuate_brakes();
```

```
  }}
```

Kernel vs. User Space

```
// OS use "SWI" or "Software Interrupt"
// "Software Interrupt" really means a "Deliberate Interrupt request to the HW"
void deliberate_interrupt(void) {
}
//
void your_program(void) {
  while (forever) {
    int file = open("file.txt");
  }
}
// Psuedo-code for open
// This is what the open function looks like inside the OS
void open(filename) {
  R4 = open_request_number;
  R5 = filename
  SWI
}
```

Virtual Memory

Virtual memory is not purely virtual memory, it is virtual memory addresses that maps to real and physical hardware memory.

POSIX Interface