Imperial College London

Lecture 1

Introduction to Instruction Architectures & Compiler

Prof Peter YK Cheung
Imperial College London

URL: www.ee.ic.ac.uk/pcheung/teaching/EIE2-IAC/
E-mail: p.cheung@imperial.ac.uk

Welcome to this 2nd year module on Instruction Architectures and Compiler (IAC). This module is one that makes EIE different from EEE. It provides you with both practical and theoretic knowledge in the design of a modern processor architecture, and how to write a compiler for it.

The emphasis of my part of the module (Autumn term) will be on the instruction set architecture (ISA), how to design an entire processor from scratch, and how to create a simulation model for the processor and its associated memory.

Please note that I do not put my notes on BlackBoard because I always make my notes open-source (i.e. anyone can access them with or without an Imperial College account). Instead, I maintain a course webpage shown in the slide here, and a Github repo where I put the Lab stuff. These are updated week-by-week as we progress through the term. Everything I cover: lecture notes, problem sheets, solutions, sample exam paper, experiment instructions, design solutions and useful resources are all included here:

http://www.ee.ic.ac.uk/pcheung/teaching/EIE2-IAC/

https://github.com/EIE2-IAC-Labs/

I strongly recommend that you BOOKMARK these two links.

Intended learning outcomes

- **Theory**
  - How pipelined CPUs with caches process instructions (Autumn)
  - How a compiler turns code into instructions (Autumn)
  - The hardware-software interface between compiler and CPUs (A+S)
- **Practise**
  - Creating a CPU in System Verilog (Autumn)
  - Creating a compiler in C++ (Spring)
- **Skills**
  - Improved knowledge of RTL languages and tools (Autumn)
  - Increased proficiency in C++ and software (Spring)

By the end of the Autumn term, you will be able to:

- Use design essential digital hardware using SystemVerilog
- Design a multi-stage pipeline processor compliant to RISC-V ISA
- Use common software tools and languages such as Git/Github, VSCode, Markdown, SystemVerilog, C++
- Use digital design tools such as Verilator, gtkWave to verify that your design functioning correctly
- Use the CPU with memory to execute assembly and C programmes
- Write testbenches to verify correctness of digital hardwarwe

This module is very demanding on your time and commitment. You also learn a lot as a result.

Learning approach for Instruction Architecture (Autumn Term)

- **Lectures: ~ 2 hours per week (Tuesday 4pm – 6pm)**
  - In person, cover theoretical stuff + introduction to Lab/Project
- **Reading: ~ 2-3 hours per week (untimetabled)**
  - Sections to read given in lecture
- **Supervised Lab (2 - 4 hours Thursday and/or Friday)**
  - Pair-based learning with Lab instructions in first half of term
  - Team-based project to design RISC-V processor in 2nd half of term
  - Team working in groups of 4, but individually assessed
  - Lab Sessions also serve as Tutorial Sessions – you can ask staff or UTA questions about course materials
  - Complete your partner declaration survey here:

    https://forms.office.com/e/TtBK3asFr4

All lectures and lab sessions for the Autumn term part of the module will be in person. There will NOT be live streaming but all sessions will be recorded. The recordings will be uploaded and made available a fews days after the sessions happen.

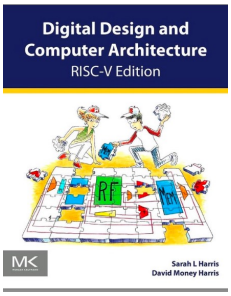
The class is divided into Group A and Group B. Each group is scheduled for a two-hours lab session each week (except mid-term week):

- Group A – Friday 10.00 – 12.00 in Room 305
- Group B – Thursday 9.00 – 11.00 in Room 305

You are to work in pairs. Each student must find a lab partner in your group and notify me via the link to a MS survey form no later than Monday 17 October.

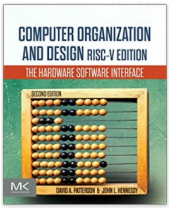
Autumn : Course Textbook

- Digital Design and Computer Architecture (RISC-V Edition) by Sarah Harris and David Harris.



https://www.vlebooks.com/product/openreader?id=IMPERIALBB&accId=8994656&isbn=9780128200650

- Computer Organization and Design RISC-V Edition, Patterson and Hennessy (~£77), electronic copy available at:



https://library-search.imperial.ac.uk/permalink/44IMP_INST/mek6kh/alma991000613172401591

The first book here is the recommended textbook. My course will be substantively based on this book and I will also be using some of the slides provided by the authors.

You can get free access to the digital version of this book via Imperial College library repository here:

https://www.vlebooks.com/product/openreader?id=IMPERIALBB&accId=8994656&isbn=9780128200650

Of course, this is only available to Imperial College students with an Imperial account.

The second textbook is both famous and popular, but less useful for my module because it has less of the hardware elements that are included in my syllabus. You can also get this via the link:

https://library-search.imperial.ac.uk/permalink/44IMP_INST/mek6kh/alma991000613172401591

Assessment – Entire module (10 ECTS)

- **The course uses three modes of assessment:**
  - **Labs (10% or 1 ECTS):**
    - Autumn (5%): Mid-term quiz on Lab experiments
    - Spring (5%): Tools for building compilers
  - **Coursework (50% or 5 ECTS):**
    - Autumn (25%): building a working RISC-V processor
    - Spring (25%): building a working C compiler
  - **Final exam (40% or 4 ECTS):**
    - Assessed knowledge of CPUs and compilers
    - Questions cover both topics of architecture and compiler

This module is worth 10 ECTS for two terms materials.

The assessment will consist of a short mid-term quiz on the Lab experiments you conduct in the first half of Autumn term, a team project to be completed by the end of the term, and a final Summer term written examination.

These modes of assessments is mirrored in the Spring term part of the module.

Labs and coursework for Autumn Term

- **First half :**
  - 4 Lab Sessions to teach digital design with SystemVerilog
  - Work in pairs – you choose your own lab partner
  - Expect to keep a logbook on **git**
- **Mid-term : assessment of lab (5%)**
  - Online quiz on Lab 0 to Lab 3 – multiple choice + evidence on git
- **Second half : assessment on project (25%)**
  - Work in teams of 4 from two pairs (I choose)
  - Design a working RISC-V processor in SystemVerilog
  - Four tasks already partitioned – you allocate responsibilities
  - Assessed both as a team and individually (details later)

Overview on

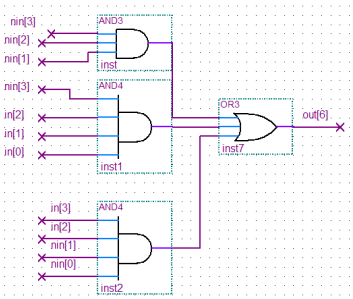
Digital Hardware Design

Part of the learning outcome of this module is to teach designing digital hardware using a hardware description language (HDL). In our case, the HDL is SystemVerilog.

You have been introduced to digital design in Year 1. Here is a recap.

How to describe/specify digital circuits?

Schematic diagram & gates



Boolean equation

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0

Truth table

in[3..0]	out[6:0]
0000	1000000
0001	1111001
0010	0100100
0011	0110000
0100	0011001
0101	0010010
0110	0000010
0111	1111000
1000	0000000
1001	0010000

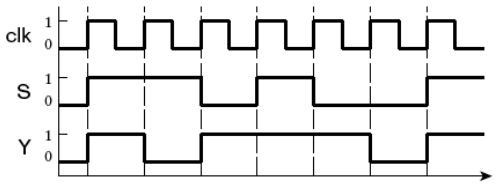
Hardware Description Language (HDL)

3-to-1 MUX

```
module mux32three(i0,i1,i2,sel,out);
input [31:0] i0,i1,i2;
input [1:0] sel;
output [31:0] out;
reg [31:0] out;

always @ (i0 or i1 or i2 or sel)
begin
  case (sel)
    2'b00: out = i0;
    2'b01: out = i1;
    2'b10: out = i2;
    default: out = 32'bx;
  endcase
end
endmodule
```

Timing Diagram



PYKC 7 Oct 2025 EIE2 Instruction Architectures & Compilers Lecture 1 Slide 8

In the first year, you learned about the different ways of describing or specifying a digital circuit.

1. Schematic diagrams with gates – this method is the first thing you learned and it is easy to understand. However, as will be seen in Lecture 3, this is not necessarily the best way to specify a large digital system.

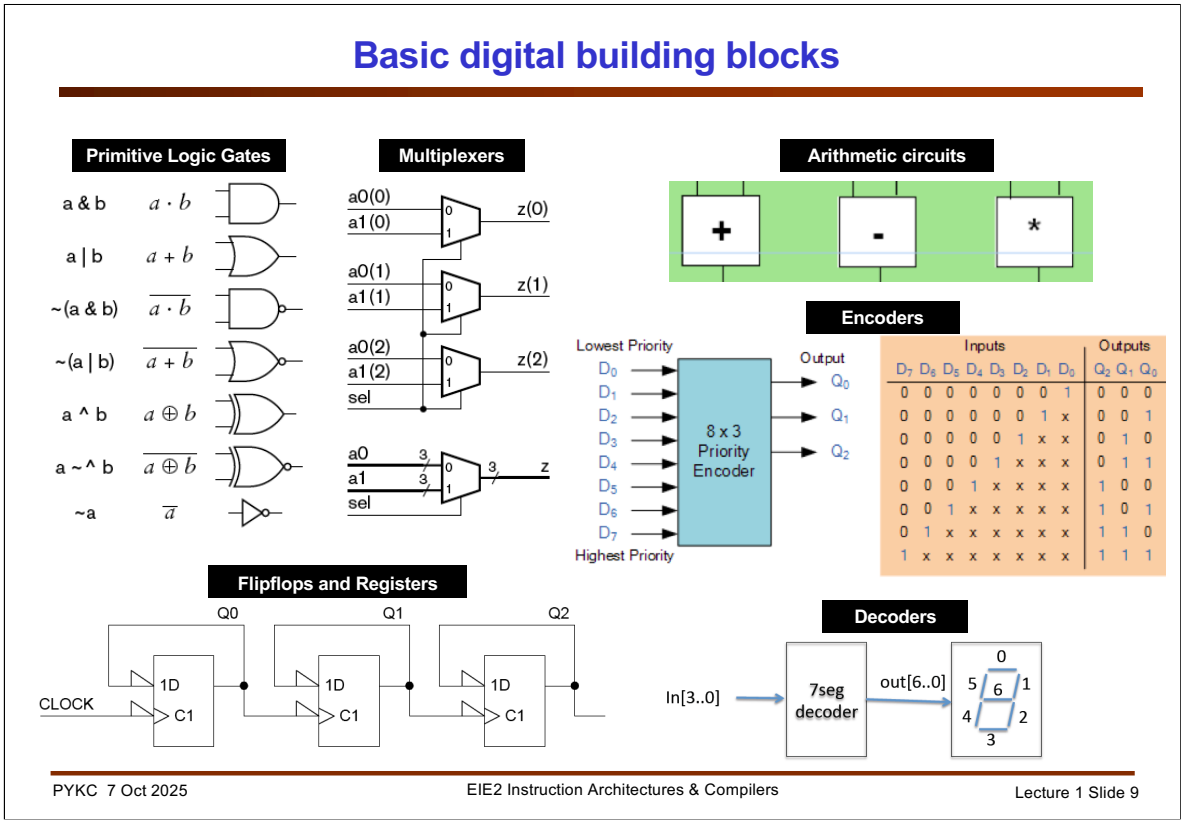
2. Boolean equations – this provides a formal way to express logical relationships between Boolean variables. Useful when designing on paper, but less useful in practice. In particular, we rarely use Boolean algebra to perform logic simplification in real-life!

3. Truth Tables – this is a universal way to describe the behaviour of a circuit and we continue to use this in datasheets or even in actual designs.

4. Timing diagrams – this is a useful way to explain behaviour of sequential circuits and is used in datasheets. However, not that useful as a method to specify a circuit in a CAD system.

5. Hardware Description Languages (HDLs) – you have already done a bit of this last year. HDL is the way that most digital designs are specified nowadays. For this course, we will be using SystemVerilog (SV) HDL, which is one that is very closed to the C language. It is also used extensively for designing integrated circuits such as ASICs and other type of chips. It is also popular in North America and in Asia.

Another popular HDL is VHDL. I personally find VHDL too wordy (verbose). Finally, there are now emerging higher level languages such as OpenCL, which is an attemp to make hardware design more like programming a computer. This topic is left to later years.



You have also learned about the various building blocks for digital electronics.

1. Primitive gates – We have the basic AND, OR, NAND, NOR, XOR and XNOR gates.

2. Multiplexers MUXs – These are really useful component. Shown here is a 2-to-1 MUX with two data inputs and one select input. The output is one or the other depending on the select input (sel). We often put a number of these together to provide multiplexing function to a multi-bit data word (as shown here with two 3-bit numbers).

3. Arithmetic circuits – Commonly found are adders and multipliers. Subtractor can be built from an adder if we use 2's complement representation of signed integers.

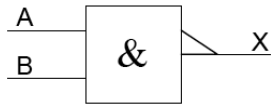
4. Encoders/Decoders – These two are related. **Encoding** is a logic module that reduces (encodes) a large number of bits and produces fewer output bits. **Decoders** are the opposite. Shown here is a 7-segment display decoder, where 4 input bits are decoded into 7 logic signals to drive the seven segments of the display. The **encoder** here is known as a **priority encoder**. It produces a 3-bit output showing where the first '1' is encounters from the most-significant bit D7 to the least significant bit D0.

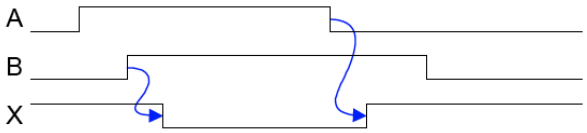
5. Flipflops and Registers – These are the building blocks for all sequential circuits. As will be seen later, we really only use one type of flipflop – the D-FF.

These are all important components that all digital circuit designers need to be familiar with. However, nowadays, we rarely design large digital systems at such low levels. Instead we generally try to express these building blocks in a more abstract manner in a hardware description language (as we will see in later lectures).

In addition to these basic blocks, we also have memory devices and processor blocks. These are topics that we will cover towards the end of this module.

Cause & Effect





Input B going high causes X to go low

Input A going low causes X to go high

Propagation Delay:
The time delay between a cause (an input changing) and its effect (an output changing), assuming output load capacitance of 30pF.

Example: 74AC00: Advanced CMOS 2-input NAND gate

	min	typ	max	
A↑ to X↓ (t_{PHL})	1.5	4.5	6.5	ns
A↓ to X↑ (t_{PLH})	1.5	6.0	8.0	ns

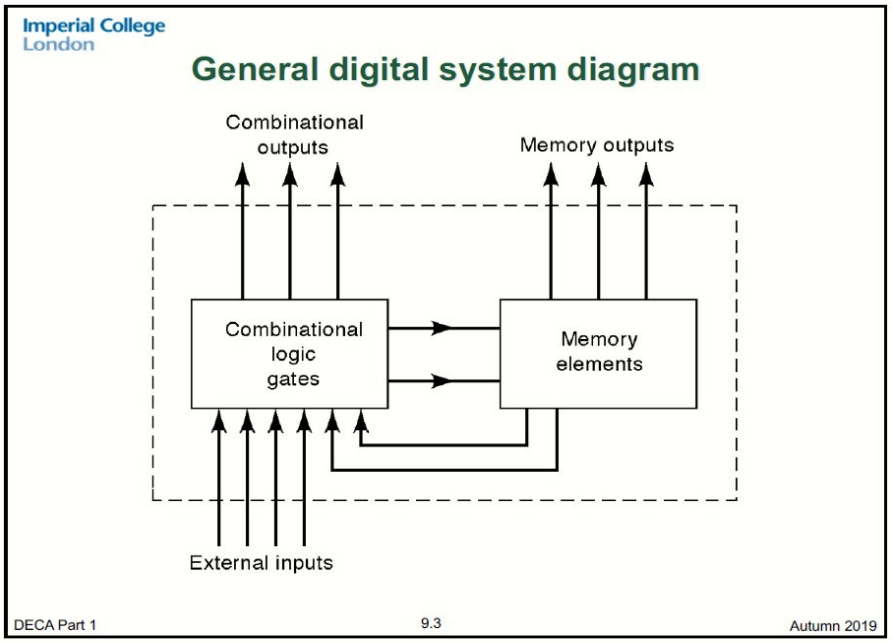
t_{PHL} and t_{PLH} refer to the direction that the output changes: high-to-low or low-to-high.

All digital circuits exhibit propagation delays. Here it shows the delay table for a "**discrete logic**" CMOS NAND gate. The delay could be in the region of nanoseconds. However, with the FPGA chips we use for this module, the internal "gate" propagation delay are in tens of picoseconds, which is much faster than discrete logic. As can be seen later, the "gates" inside the FPGA are also much more complex than a simple NAND gate. In fact, they are not gates at all!

Also note that propagation delay depends on the "**cause**" (input rising or falling, and on the gradient of the edges) and the "**effect**" (output rising or falling). Delay also depends on what are connected to the output (i.e. the loading). As can be seen in the example here, the rising edge A to falling edge X delay is lower than that of A falling to X rising.

Note that I use an arrow to indicate the cause (the blunt end) and the effect (the pointed end) in a timing diagram.

Combinatorial and sequential logic



All digital systems can be generalized into two types of circuits: combinational and sequential.

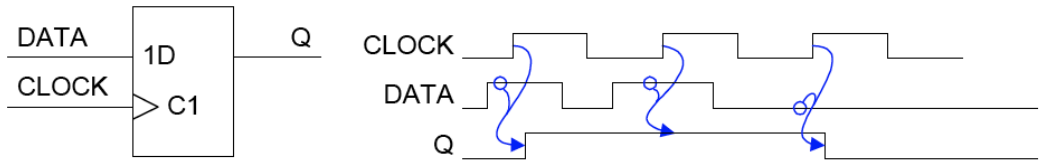
Combinational circuits do not have states. All outputs are immediately produced depending only on instantenous inputs.

Sequential circuits consist of combinational circuits AND STORAGE (such as flip-flops or memory elements). Memory elements store the “state” of the circuit.

All sequential circuits we use in this module are SYNCHRONOUS. That is, the state of the circuits changes only on the active edge of a clock signal. The active edge can be rising (positive) or falling (negative) edge of the clock signal.

In general, a digital system can have more than one clock signal (we call this a clock domain). However, for this module, we restrict ourselves to only a single clock domain.

D-Flipflop (1)



Notation:

>	input effect happens on the rising edge
C1	C ⇒ Clock input, 1 ⇒ This input *is* input number 1.
1D	D ⇒ Data input, 1 ⇒ This input *is controlled by* input number 1.

The meaning of a number depends on its position:

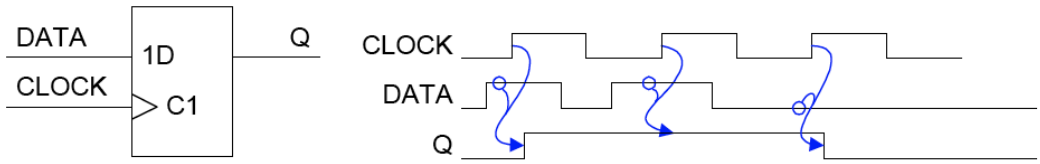
A number *after* a letter is used to identify a particular input.
A number *before* a letter means that this input is controlled by one of the other inputs.

You learned about various types of flipflops (FFs) in the first year. In fact, all you need is the D-FF. With a D-FF, you can construct circuits to behave like various types of flipflops: Toggle (T-FF), set-reset (SR-FF) or a JK-FF.

Therefore in this course, we will ONLY use D-FF for everything. This is in fact what happens in practical designs.

We use the IEEE standards for the symbol here. C mean clock input, the number 1 is a numerical label (as clock 1). D is for data input, and 1D means this input is controlled by input 1. Q is the flipflop output.

D-Flipflop (2)



Cause and Effect:

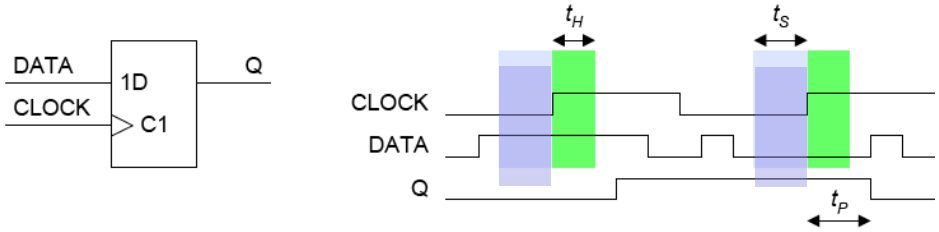
- CLOCK↑ causes Q to change after a short delay. This is the *only* time Q ever changes.
- The value of D *just before* CLOCK↑ is the new Q.
- Propagation delay CLOCK↑ to Q is typically 1 ns.
- Propagation delay DATA to Q **does not make sense** since DATA changing does not cause Q to change.

Timing and delay parameters for flipflop is different from that with gates. Shown here is a D-FF that responses to a rising edge on the clock signal. A D-FF is like a camera, taking a "picture" from the scene (input is D). The clock input C1 is like the trigger on the camera – when pressed it samples the input and take a picture. The "cause" here is the rising edge of the CLOCK and the "effect" is the Q output sampling the D input, and keep the value until the next rising edge of the clock.

The delay here is from CLOCK rising edge to Q output changing. However, for the D-FF to work properly, there are two other timing parameters which are important: the **setup time** and the **hold time**. I will be talking about these in a later lecture.

Setup and Hold Times

The DATA input to a flipflop or register must not change at the same time as the CLOCK.



Setup Time: DATA must reach its new value at least t_S before the CLOCK↑ edge.

Hold Time: DATA must be held constant for at least t_H after the CLOCK↑ edge.

- Typical values for a register: t_S = 5 ns, t_H =3 ns (discrete logic/ I/O circuit)
  t_S = -50ps, t_H = 0.2 ns (internal LE)
- The setup and hold times define a window around each CLOCK ↑ edge within which the DATA **must not change**.
- If these requirements are not met, the Q output may oscillate for many nanoseconds before settling to a stable value.

Registers (D-FFs) are used everywhere in digital circuits. Using registers has the advantage of: 1) **synchronising** all activities to a clock signal; 2) **isolate** different part of the digital systems between registers (because the registers block the signal until the next active edge of the clock; 3) makes timing consideration much easier to handle.

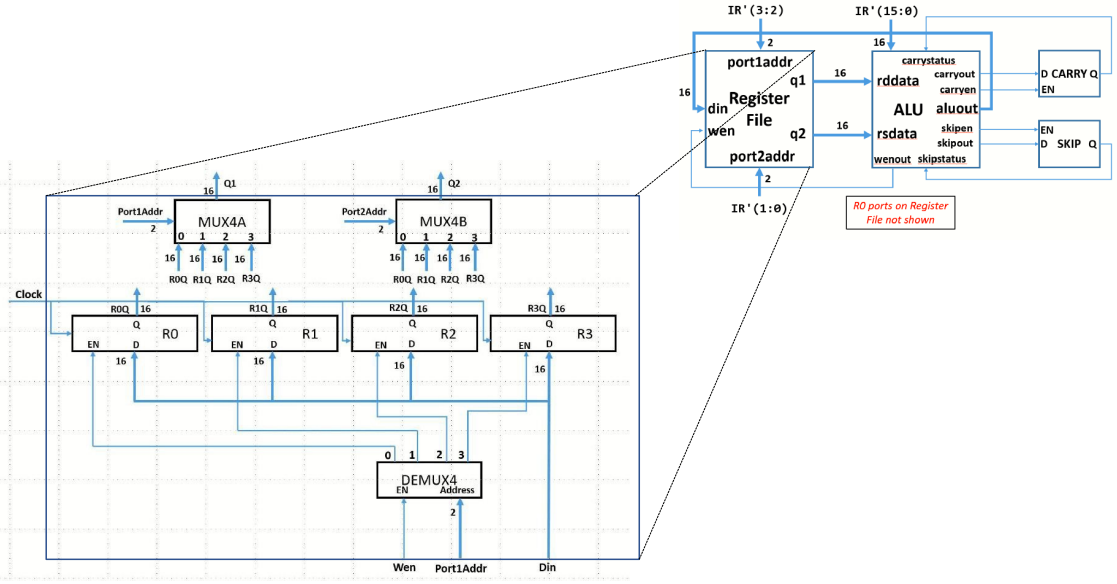
In the circuit shown here, the D-flipflop is triggered on the rising edge of the clock. The value in DATA is sampled and stored, and keep as output Q. However, for reliable operations, DATA MUST BE STABLE some time before the rising edge of CLOCK. This time is known as **setup time** t_S. This time is needed because there is internal propagation of the DATA signal which must be taken into account. As a result, for the D-flipflop to work, such internal delay is specified as the flipflop setup time requirement.

Similarly, DATA MUST BE STABLE and holds its value some time after the rising edge of CLOCK. This time is known as hold time t_H.

What happens if data changes within the setup/hold time window? The Q output becomes unknown (could be '1' or '0', or at a voltage level that is between the two). Eventually Q will go to '0' or '1', but the time it takes to reach the stable Q value is random! Such a state of the flipflop is known as a "**metastable**" state.

In this module, we are only concerned with a high-level FUNCTIONAL view of digital circuit. That is, we assume that the digital circuit is control with a clock where NO timing violation ever occurs. We therefore will not worry about setup and hold time in our module. All digital signals and outputs change on each clock cycle.

Design Hierarchy



You will also be design circuits that have hierarchy. Shown here is a simple CPU with a register file as a component. Expanding this Register file may show that it is made up of numerous registers, multiplexers and demultiplexers.

You must design your circuits so that components can be reused as much as possible. For example, a good multiplexer design should be usable for different number of data bits.

Overview on

Instruction Set Architecture (ISA)

One of the most important theme of this module is Instruction Set Architecture. Here is some of the basic terminology and concepts that you need to know.

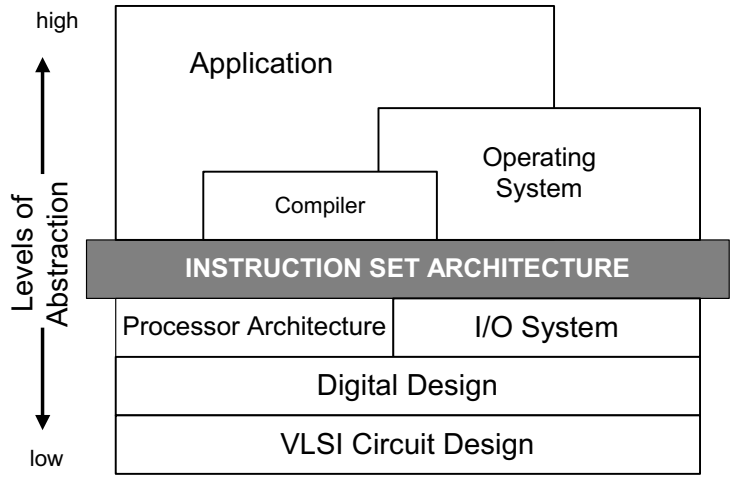
Eight Great Ideas in Computer Architecture

1. **Design for Moore's Law**
2. **Use Abstraction to Simplify Design**
3. **Make the Common Case Fast (RISC philosophy)**
4. **Performance via Parallelism**
5. **Performance via Pipelining**
6. **Performance via Prediction**
7. **Hierachy of Memories**
8. **Dependability via Redundancy**

Computer architecture is a field of study among electronic engineers and computer scientists for decades. However, in the past 30 years, this field has progressed so much that the current performance of the latest processor is beyond what we expect just a decade earlier.

The reason for this rapid progress is due to the eight great ideas introduced by computer architects.

What is "Computer Architecture" ?



- Key: Instruction Set Architecture (ISA)
- Different levels of abstraction

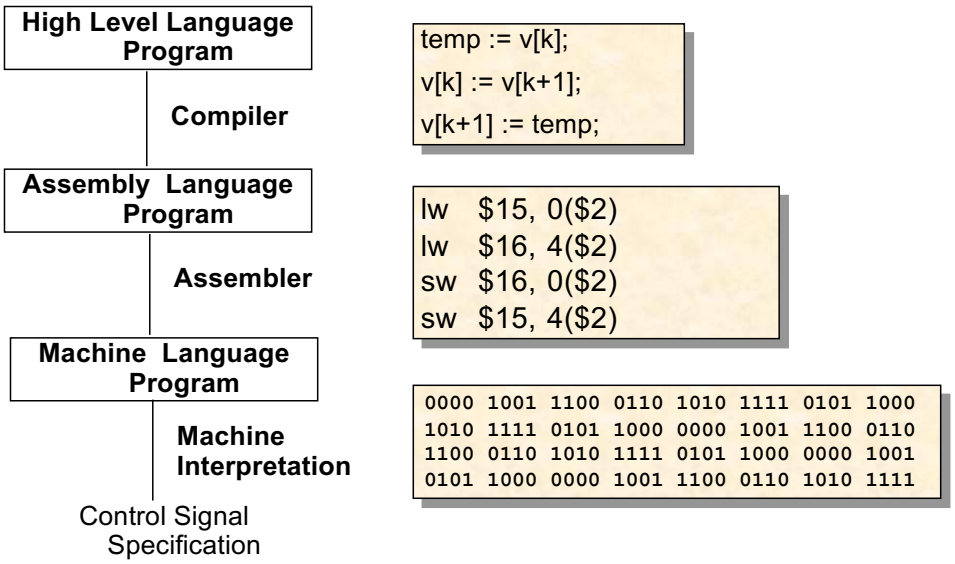
This slides shows a high level view of a computer system, from the highest abstraction level to the lowest abstraction level.

At the top are application programmes written in a high-level language such as C++ or Python. Such programmes needs tools such as compiler, editor, assembler, linker etc. The computer system must also run an operating system, which shield the user from the detail hardware structure. For example, OS provides easy means for the user to input data, communicate with other computers via network services etc.

At the lowest level, we have the integrated circuit as transistors and interconnects. These are implementations of circuits designed by engineers such as yourselves. The design is based on processor architecture – the topic of this module. The I/O system allows interfacing between the processor and the external world with standard protocols such as UART, USB, SPI etc.

Right in the centre that links together the abstract level at the top and the physical and logical level at the bottom is the Instruction Set Architecure (ISA). The ISA provides an anchor to everything in a computer system.

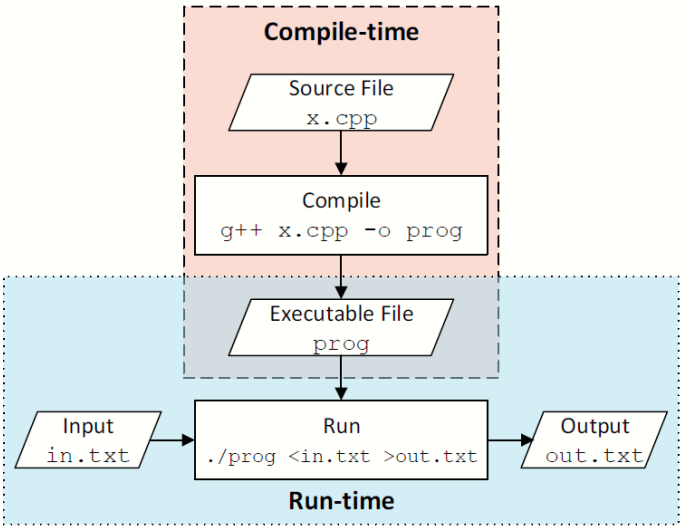
Levels of representation in computers



Programs for a computer can be in various level of representations. The highest is of course the high-level language such as C or C++. This must be translated to assembly language using the ISA using a compiler. The output of the compiler are assembly language instructions that must than be further mapped to machine code program (binary).

The machine code instructions are what the CPU use to run your program. These are decoded into different control signals which govern the internal working of the processor.

Compile-time and Run-time



You should be very familiar with this by now. Here a slides that show the compilation process, generating an executable file which you run with inputs and outputs.

This module will teach you both the design and implementation of a working processor based on the RISC-V ISA, and how to create your own compiler. By the end of the Spring term, you should have your own compiler that compiles C program to run on your RISC-V process design.

What is "Instruction Set Architecture (ISA)"?

- ". . . the attributes of a [computing] system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation."
  - ➢ Amdahl, Blaaw, and Brooks, 1964

ISA includes:-

- Organization of Programmable Storage
- Data Types & Data Structures: Encodings & Representations
- Instruction Formats
- Instruction (or Operation Code) Set
- Modes of Addressing and Accessing Data Items and Instructions
- Exceptional Conditions

At the centre of a computer architecture is its ISA. Currently the three most popular ISA are: Intel's X86 and its variants, ARM's processor ISA, and finally the RISC-V ISA.

Only the RISC-V ISA is open source. This is partly why it is chosen for this module.

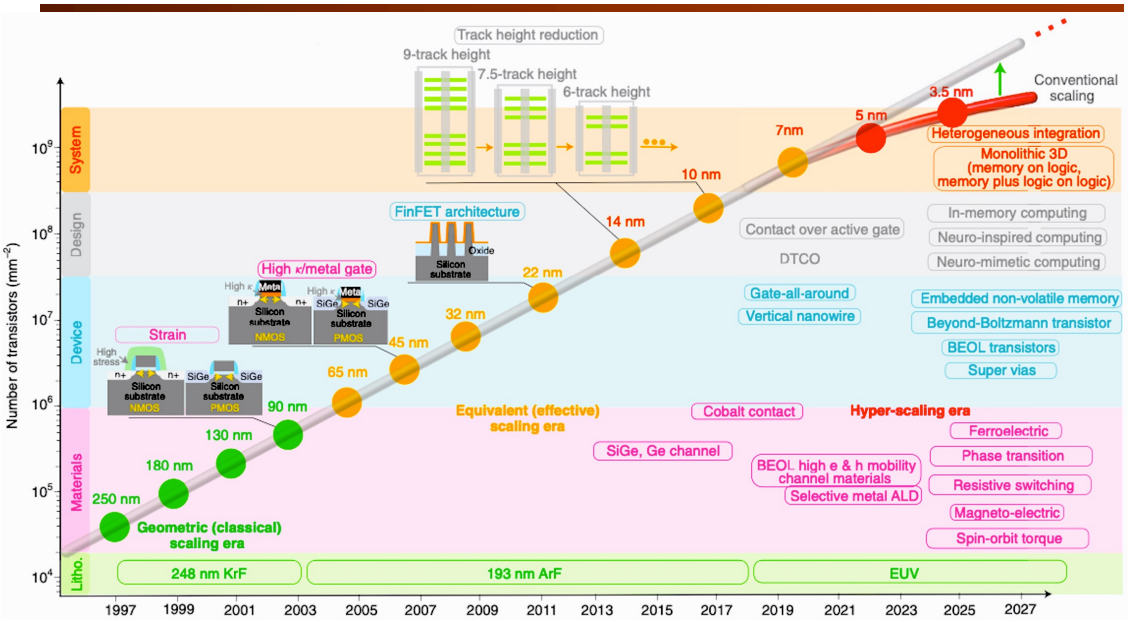
The ISA includes all the definitions listed above in the slides. You should know each of these charactersitics of the RISC-V ISA by the end of the term. Otherwise your processor will not conform to the RISC-V standard, and will no be able to run the executable files compiled using RISC-V compilers.

Instruction Set Architecture (ISA)

- A very important abstraction
  - interface between hardware and low-level software
  - standardizes instructions, machine language bit patterns, etc.
  - advantage: *different implementations of the same architecture*
  - disadvantage: *sometimes prevents using new innovations*

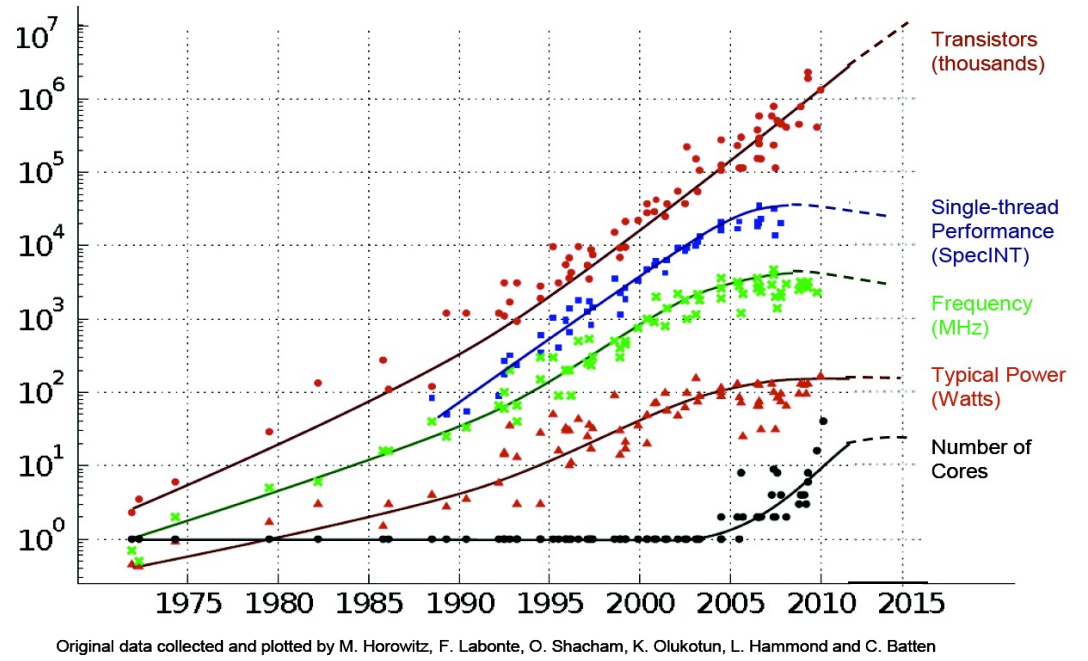
- Modern instruction set architectures:
  - ARM, Intel x86, **RISC-V**, Xtensa LX6/LX7 (used in ESP32)

Technology: Logic Density



Before we dive into the RISC-V ISA, let us examine the progress of integrated circuit technology. This a graph showing Moore's law in action. The trend is a straight line with the number of transistors plot in log scale and the years plotted in linear scale. This is the result of technology scaling (i.e. transistors getting smaller in dimension) and die size (i.e. area of a chip) getting larger while the manufacturing yield is maintained.

Processor Speed Improvements

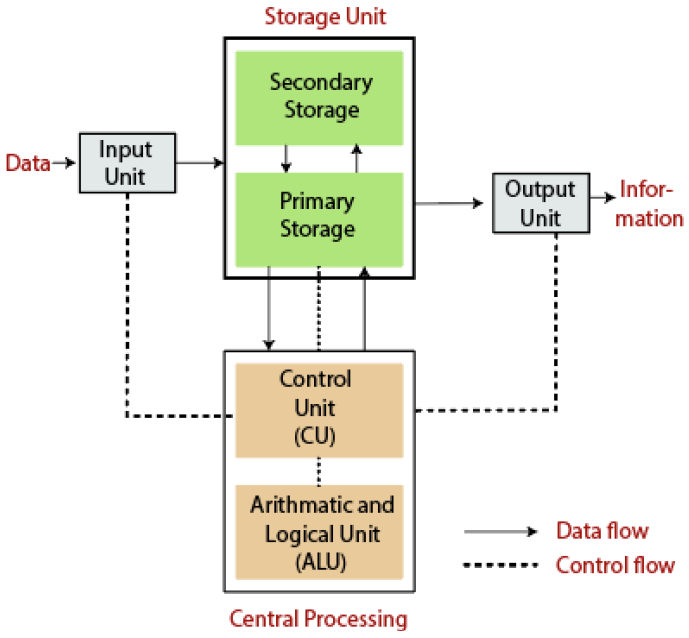


Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten

However, it is clear from this slide that while transistor per chip keeps increasing, the clock speed and the power dissipation per chip is leveling off or even dropping. The performance improvement in a computer system is now sustained not by technology shrinkage, but by having many threads on many cores.

In this module, we only concern ourselves with a single CPU core. This lay the foundation for later years elective modules on many-core designs.

A Typical Computer System with I/O



A typical computer system consists of the CPU, memory storage, input units and output units. There are two types of digital signals involved: data signals on the datapath, and the control signals.

RISC-V Characteristics

- Emphasis on simplicity and regularity
  - 32-bit instructions
- Smaller is faster
  - Small register file and fewer instructions
- Optimise for common cases
  - e.g. include support for constants
- One ISA family with different variants

Name	Description	Version	Status	Instruction Count
RV32I	Base Integer Instruction Set - 32-bit	2.1	Frozen	49
RV32E	Base Integer Instruction Set (embedded) - 32-bit, 16 registers	1.9	Open	Same as RV32I
RV64I	Base Integer Instruction Set - 64-bit	2.0	Frozen	14
RV128I	Base Integer Instruction Set - 128-bit	1.7	Open	14

- Open-source (up to a certain extend)

The RISC-V architecture has thse characteristics: It uses regular structure with 32 (or 64) internal registers, with 32 (or 64) bit instructions. (We only use 32 register, 32-bit version of RISC-V in this module).

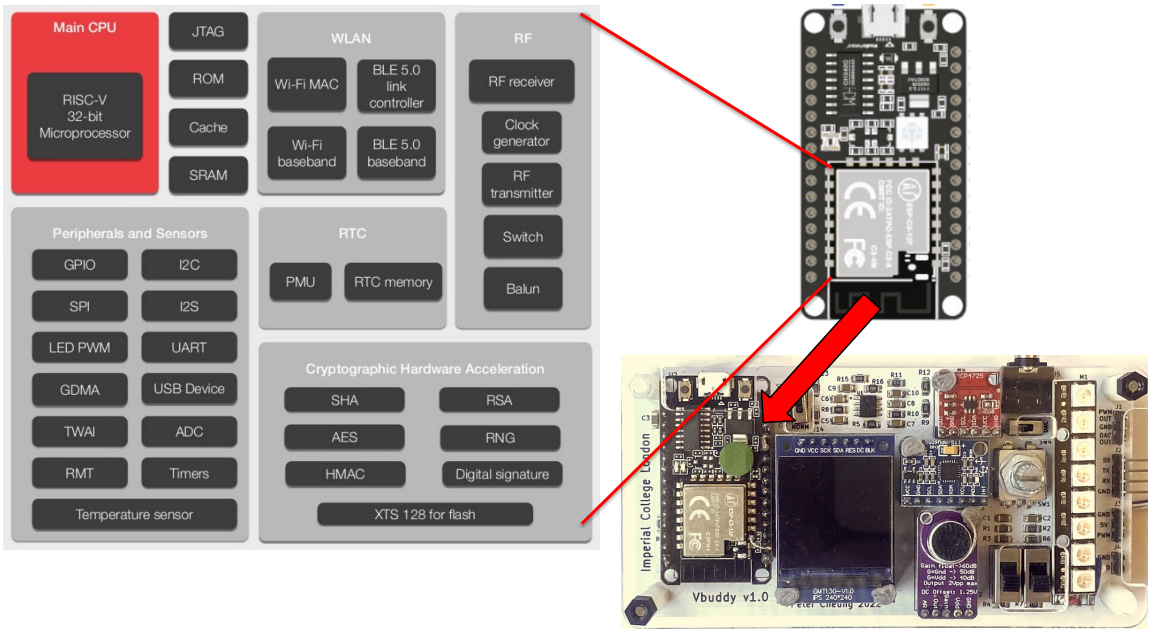
The design philosophy of RISC-V is that only the most common instructions are used. This is true to the name of Reduced Instruction Set Computer (or RISC).

Since there are fewer instructions, a designer (such as yourself) can optimize the internal hardware to run these instructions faster.

RISC-V is not a single ISA. The specification includes multiple variants and extensions.

Extension				
M	Standard Extension for Integer Multiplication and Division	2.0	Frozen	8
A	Standard Extension for Atomic Instructions	2.0	Frozen	11
F	Standard Extension for Single-Precision Floating-Point	2.0	Frozen	25
D	Standard Extension for Double-Precision Floating-Point	2.0	Frozen	25
G	Shorthand for the base and above extensions	n/a	n/a	n/a
Q	Standard Extension for Quad-Precision Floating-Point	2.0	Frozen	27
L	Standard Extension for Decimal Floating-Point	0.0	Open	Undefined Yet
C	Standard Extension for Compressed Instructions	2.0	Frozen	36
B	Standard Extension for Bit Manipulation	0.90	Open	42
J	Standard Extension for Dynamically Translated Languages	0.0	Open	Undefined Yet
T	Standard Extension for Transactional Memory	0.0	Open	Undefined Yet
P	Standard Extension for Packed-SIMD Instructions	0.1	Open	Undefined Yet
V	Standard Extension for Vector Operations	0.7	Open	186
N	Standard Extension for User-Level Interrupts	1.1	Open	3
H	Standard Extension for Hypervisor	0.0	Open	2
S	Standard Extension for Supervisor-level Instructions	1.12	Open	7

ESP32-C3 Microcontroller – Chip, module, board



You will be designing in a Team of 4 students your own version of RISC-V processor RV32-IM (32-bit integer with multiply-divide).

Shown here is the RISC-V CPU in a microcontroller by Espressif called ESP-C3, which is used in a module by AI Thinker, the ESP-32-12F-kit.

This module is used in a PCB board (called Vbuddy) designed by me to support the laboratory session of this course. More about this next week when you will start the in-person lab sessions.

Lab 0 – Setting up the Development Environment

- Follow instructions at the following Github page:

https://github.com/EIE2-IAC-Labs/Lab0-devtools.git

- You need to install and learn to use the following tools:
  - **VS Code** – the Integrated Development Environment (IDE)
  - **Verilator** – compile SystemVerilog HDL into C++
  - **RISC-V GNU toolchain** – open-source tools for RISC-V including compiler, assembler, simulator etc.
  - **gtkWave** – view waveforms generator by Verilator model for debugging
  - **Git and Github** – to record your work and your design (for assessment)

- Optional but useful to learn and install:
  - **Linux commands** – only basic ones
  - **Markdown language (MD)** – used with Git, Github
  - **Obsidian** – Cross-platform open-source note taking tool
  - **Make utilities** – used to compile and manage software build
  - **Bash** – basic scripting language all EIE students should know

This module requires you to learn many transferable skills and become familiar with software tools that are applicable elsewhere. To prepare you for the in-person lab session next week, you have completde in your own time (and at your own pace) Lab 0. The purpose is to install the software tools required for lab coursework and project this term.

Go to the link shown here and follow the instructions to install the software required.

https://github.com/EIE2-IAC-Labs/Lab0-devtools.git

A group of students from last year (Clemen Kok, Aranya Gupta, Frishikesh Venkatesh and Daniel Coroama) created a github repo to supplement my teaching materials. This repo contains useful links to additional reference on tools and skills useful not only to this module, but to the entire EIE program as a whole. Check this out here:

https://github.com/clemenkok/IAC0

8 most useful Linux Commands

Command	Example	Description
1. **ls**	ls ls -alF	Lists files in current directory List in long format
2. **cd**	cd tempdir cd .. cd ~dhyatt/web-docs	Change directory to tempdir Move back one directory Move into dhyatt's web-docs directory
3. **mkdir**	mkdir graphics	Make a directory called graphics
4. **rmdir**	rmdir emptydir	Remove directory (must be empty)
5. **cp**	cp file1 web-docs cp file1 file1.bak	Copy file into directory Make backup of file1
6. **rm**	rm file1.bak rm *.tmp	Remove or delete file Remove all file
7. **mv**	mv old.html new.html	Move or rename files
8. **more**	more index.html	Look at file, one page at a time

In addition to all the software tools that you need to learn to complete this module (such as VS code, Verilator, GTKwave, Git etc., you will also learn many other skills highly relevant to becoming a good Computer Engineer. One of the skills is to be able to use commands in a terminal window.

Here is a table containing 8 most useful Linux command that you need to learn. There is a 9th one: ”chmod”. I left this out for now, but you will encounter this later.

Basic Markdown Syntax

Element	Markdown Syntax

Heading	`# H1`
	`## H2`
	`### H3`

Bold	`**bold text**`
Italic	`*italicized text*`
Blockquote	`> blockquote`

Ordered List	`1. First item`
	`2. Second item`
	`3. Third item`

Unordered List	`- First item`
	`- Second item`
	`- Third item`

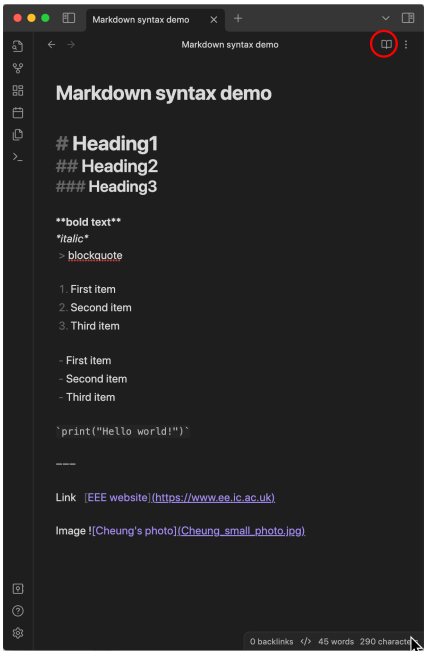
Code	`` `code` ``
Horizontal Rule	`---`
Link	`[title](https://www.example.com)`
Image	`![alt text](image.jpg)`

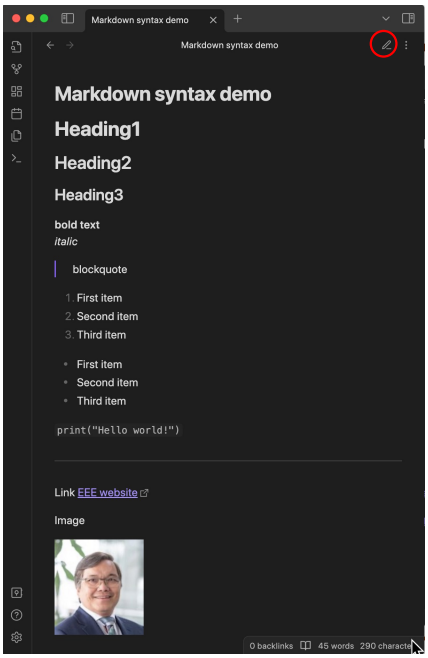
For this module, Git is being used extensively for me to disseminate lab instructions, for you to post your logbook and results, and finally for your team to collaborate on the project.

Git uses a simple language called “Markdown” or MD. This is a much simplified version of HTML and provide ways of formatting text easily.

Here are the few most important Markdown syntax that you should know. There are extensions to this list, but you can do most things with just these few Markdown “commands”

Obsidian – Best note taking app?





Obsidian is a free-to-use note taking app that uses Markdown as its language. Many people have started using this multi-platform freeware (personal use only) for taking notes and building their own repository of information on different topics.

I strongly recommend you watch this YouTube video to find out why Obsidian is fast becoming the most popular note taking app:

https://www.youtube.com/watch?v=DbsAQSIKQXk

Here is my Obsidian notes on the basic syntax for Markdown and the resulting formatted page. You can toggle between the two by clicking the icon at the red circle.