Imperial College London

Lecture 13
The CPU and the Pyboard

Professor Peter YK Cheung
Dyson School of Design Engineering

URL: www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/
E-mail: p.cheung@imperial.ac.uk

In this lecture, we will look at how storage (or memory) works with processor in a computer system. This is in preparation for the next lecture, in which we will examine how a microprocessor actually works inside.

A Very Simple Central Processing Unit

- Based on von Neumann model
- Stored program and data in memory
- Central Processing Unit (CPU) contains:
  - Arithmetic/Logic Unit (ALU)
  - Control Unit
  - Registers

- Look into memory, sees '1' and '0'. Meaning depends on context.
- All computers requires at least three types of signals (buses):
  - address bus - which location
  - data bus - carries the contents of the location
  - control bus - governs the information transfer

In the last lecture we have looked at the overall structure of a computer, which has three main components: CPU, memory and IO, all connected together via three buses: address bus, data bus and control bus.

In this lecture we will look at the CPU in some detail.

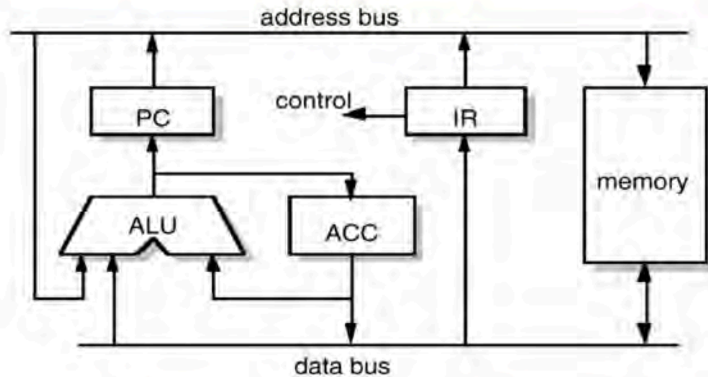
A CPU has a number of modules inside. These include a arithmetic and logic unit, a control unit and at least one, but usually many more, registers.

Remember registers are just a bunch of D-FFs.

Modern computers and microprocessors are based on the von Neumann model which uses memory to store both the instruction codes (i.e. the program) and the data.

It is important to remember and understand that if you could open the top of a memory chip and read the values stored inside the chip, you will see '1's and '0's. What they mean depends on where they are stored and the context. You cannot tell by just looking at memory which word is an instruction word, and which word is data. For that, we need some other information.

A Very Simple CPU



If you look inside a simple CPU, you are going to find these modules:

PC = **P**rogram **C**ounter – it stores the **address** of the NEXT instruction to be executed

IR = **I**nstruction **R**egister – It stores the current instruction binary code (called machine code) to be executed

ALU = **A**rithmetic/**L**ogic **U**nit – It performs the actual arithmetic or logical operations

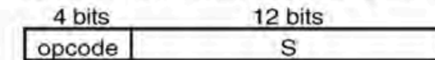
ACC = **Acc**umulator (or result register) – It stores the temporary data or result

In modern CPUs, there are many other modules. For example, instead of having only one ACC, there could be many more temporary registers. For example, the ARM processor in the Pyboard has 16 registers (including one for the PC). In addition to the ALU, there is also a floating point unit (FPU) and even an additional computational engine known as Adaptive Real-Time Accelerator (ART).

In order for you to understand how a CPU does its job, let us now create one that is very simple and only having the few modules shown above.

A Very simple CPU

- Let us design a simple processor MU0 with 16-bit instruction and minimal hardware:-
  - Program Counter (PC) - holds address of the next instruction to exec
  - Accumulator (ACC) - holds data being processed
  - Arithmetic Logic Unit (ALU) - performs operations on data
  - Instruction Register (IR) - holds current instruction code being executed
- Let us further assumes that the processor only has 8 instructions and can only access a maximum of 4k byte (2^{12}) of memory.
- The 16-bit instruction code (machine code) has a format:



- Note top 4 bits define the operation code (opcode) and the bottom 12 bits define the memory address of the data

This simple CPU uses instruction code that has 16-bits.

We assume that there is only 8k byte of memory and the processor is a 16-bit processor meaning that it process data in 16-bit words. (ARM normally process data in 32-bits or 64-bits.) Therefore the processor memory is organised as 4k x 16 bits. 4k words require 12 address bits.

Shown here is the format of the instruction. The top 4 bits are used to specify **the operation** to be performed. This is known as the **opcode**.

With 4-bit opcode, we could specify 16 operations, but we will only create 8 operations with opcode going from 0 to 7.

The remaining lower significant 12 bits in the instruction word are used to specify the **memory address** for the operation if required. We will see what that means in the next few slides.

Instruction Set

Instruction	Opcode		Effect
LDA S	0000	0	ACC := mem_{16}[S]
STO S	0001	1	mem_{16}[S] := ACC
ADD S	0010	2	ACC := ACC + mem_{16}[S]
SUB S	0011	3	ACC := ACC - mem_{16}[S]
JMP S	0100	4	PC := S
JGE S	0101	5	if PC := S
JNE S	0110	6	if PC := S
STP	0111	7	stop

Here is our simple instruction set with only 8 instructions. Here are the explanations for all 8 instructions. These instructions are shown as three letter code (known as mnemonics) to indicate what each instruction is suppose to do. This is known as **Assembly Language** program. However, the CPU can only understand binary code. Therefore shown here are the opcodes for each instructions forming the 4 most-significant bits (MSBs). Then each instruction is followed by a 12-bit memory address S. The binary form of the instruction is known as **Machine Code** program.

LDA S - Load Accumulator with the content at memory location S

STO S - Store Accumulator value to memory location S

ADD S - Add the data in memory location S to ACC and store the result back to ACC

SUB S - Subtract the data in memory location S from ACC and store the result back to ACC

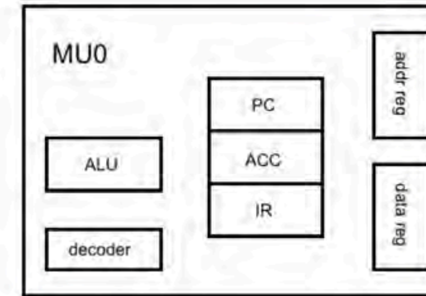
JMP S - Jump to instruction at memory address S

JGE S - Jump to instruction at memory address S if the previous ALU operation is greater than or equal to zero

JNE S - Same as before but for the case where the previous ALU operation does not equal to zero

STP - Stop the processor. Since this instruction does not require any memory address, it only uses the top 4-bits of the instruction word

Caught in the Act!

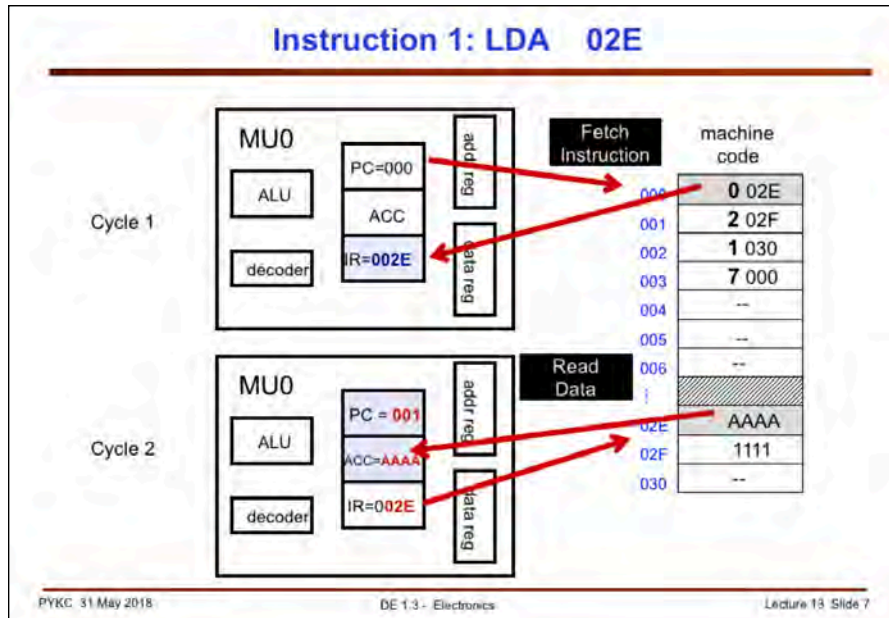


Address	Assembly program	machine code
000	LDA 02E	**0** 02E
001	ADD 02F	**2** 02F
002	STO 030	**1** 030
003	STP	**7** 000
004	–	–
005	–	–
006	–	–
⋮		
02E	AAAA	AAAA
02F	1111	1111
030	–	–

- CPU reading the first op-code

Now let us assume that at memory location 000 (hex) to 003, there are already stored four instructions. Furthermore, we also assume that in memory locations 02E and 02F (in hex again) stores two numbers AAAA and 1111 to be added together. The result of this addition, which is BBBB, will be stored back to memory location 030.

Let us now see how these instructions are executed by the CPU, one instruction at a time.



Firstly, we assume that we start from PC = 000, i.e. on reset, the program counter is zeroed.

This means that the PC is pointing to address 000 in memory where the first instruction code is stored. It is 002E, which is LDA 02E instruction, meaning that it should read the 16-bit data from memory location 02E and stores this in the accumulator.

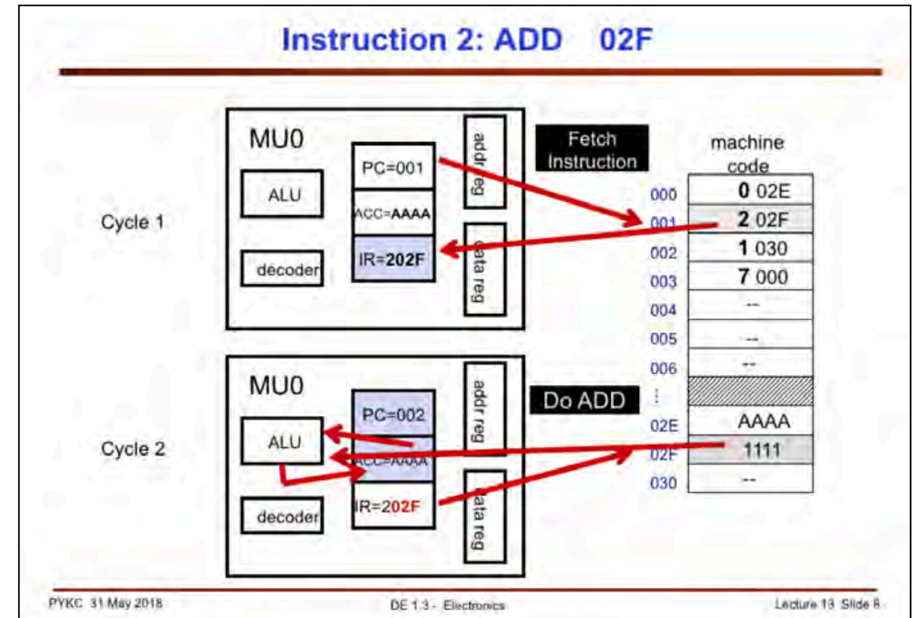
So in the first clock cycle, the PC is pushed out via the address register onto the address bus. A memory read operation is performed and the instruction code 002E is fetched and put into the instruction register IR. This is the instruction fetch cycle.

The opcode part (top 4-bits) of the instruction is passed through the instruction decoder unit and the CPU is ready to "execute" the opcode by performing a data-read-from-memory operation.

This operation is done in the second clock cycle.

The bottom 12-bit of the instruction stored in IR, 02E, is now pushed to the address register then onto the address bus. The data stored at location 02E (which is AAAA) is read from this memory location and stored in ACC.

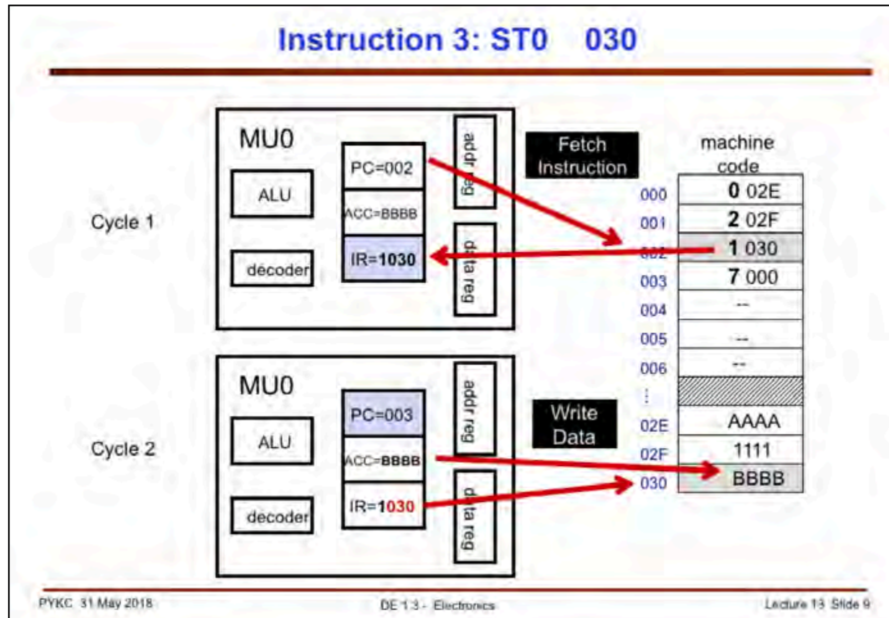
Now something else ALWAYS happens. After the PC is used to fetch an instruction, its value is ALWAYS automatically incremented. In other words, the PC always counts up once it is used (hence the name program counter). This is because we want it to always point to the **next** instruction.

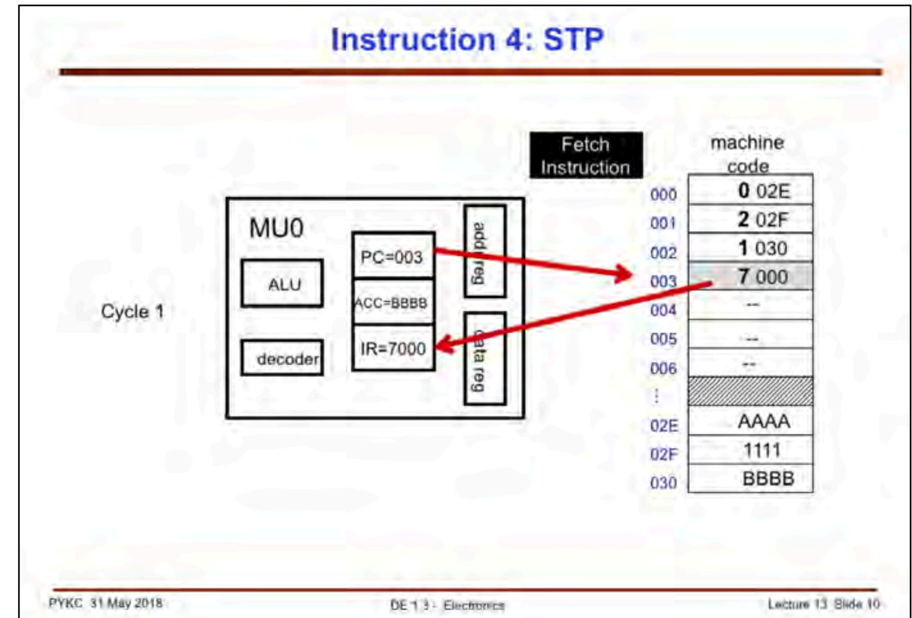


The second instruction is to add the previously fetch data to the data stored in memory location 02F.

Again in the first clock cycle, the PC value is send to the address bus, and the instruction code 202F is fetched and stored in IR as shown.

On the next cycle, the opcode (2) is decoded by the CPU and force an add operation to be performed. This involves the CPU sending the lower 12-bit of the instruction code (02F) to the address bus and performs a memory read. The data from memory 02F (i.e. 1111) is send, together with the value stored in ACC, to the ALU and the opcode tells the ALU to do an addition. The result BBBB is stored back to ACC.





In this instruction, the contents of ACC is stored to memory location 030.

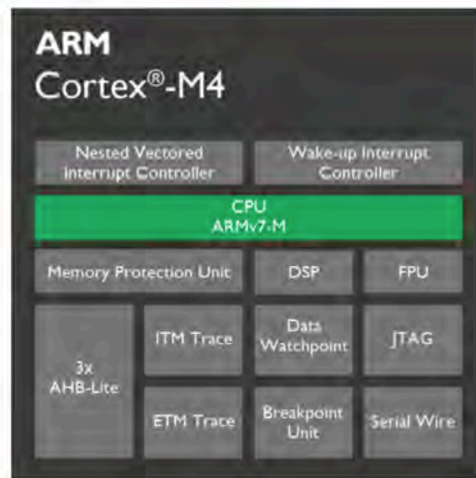
Finally, the last instruction is to tell the CPU to stop. This instruction does not involve any memory read AFTER fetching the instruction. It therefore only use the top four bit of the instruction for opcode, and it only takes one clock cycle. This is shown on the next slide.

To summarise

The operation of most processors are governed by a clock signal. For this simple CPU, we assume that:

1. The number of clock cycles taken by an instruction is the same as the number of memory access it makes.
2. LDA, STO, ADD, SUB therefore takes 2 clock cycles each: one to fetch (and decode) the instruction, a second to fetch (and operate on) the data
3. JMP, JGE, JNE, STP only need one memory read and therefore can be executed in one clock cycle.
4. Program counter (PC) - its content is incremented every time it is used (i.e. it also points to the next instruction).
5. The processor must start from a known state. Therefore, there is always a reset signal to initialise the processor on power-up.
6. Assume MU0 will always reset to start execution from address 000_{16}.
7.
8. Microprocessors performs operations depending on instruction codes stored in memory
9. Instruction usually has two parts:
   - Opcode - determines what is to be done
   - Operand - specifies where/what is the data
10. Memory contains both program and data. A peek into memory will tell you very little except a bunch of ‘1’s and ‘0’s
11. Program area and data area in memory are usually well separated
12. ALU is responsible for arithmetic and logic operations
13. There is always at least one register known as **accumulator** where the result from ALU is stored
14. There is usually one or more general purpose register for storing results or memory addresses temporarily
15. Fetching data from inside the CPU is much faster than from external memory

ARM Cortex-M4 Processor



The microprocessor inside the Pyboard is known as ARM Cortex-M4. The central part of this processor is the ARM 7 CPU. However, this ARM core (as it is called) is more than just the CPU. Surrounding the CPU are also many other useful modules that makes the ARM much easier to use. For example, it contains various instruments in order to capture various data go to and from the ARM processor. It has protection circuits and other digital circuitry to makes the interface between the CPU and everything external to it much easier. The most important here is the 3x AHB (Arm Higher Performrance Bus) which allows the CPU to talk to the peripheral devices that attach to it.

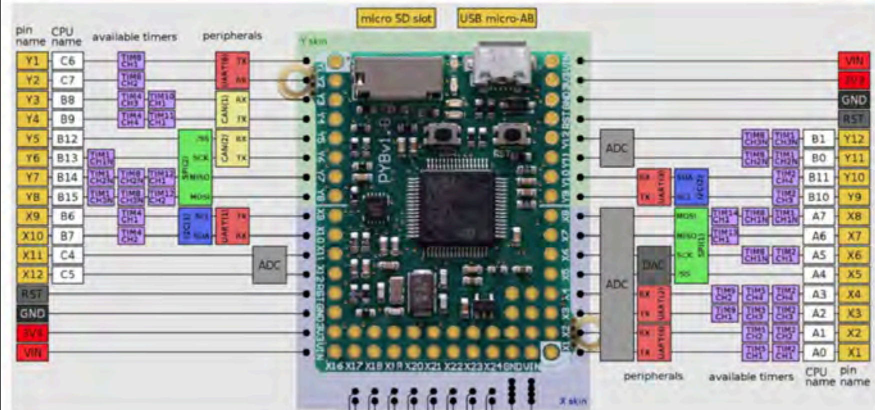
STM32F405 Microcontroller in Pyboard

We are not only just using the ARM CPU. Instead the Pyboard has a microcontroller as its main engine. This microntroller, the STM32F405 is made by ST Micro, and here is a overview block diagram for this chip.

The ARM Cortex-M4 (which has all the stuff from the last slide) ia only a small part of the entire chip – it is shown in dark blue here. All the light blue parts are added to the ARM core by ST Micro. These provides many useful functions such as many timers for control of motors and sensors, A-to-D converters and D-to-A converters, and lots of different logic circuits to connect the microcontroller to the outside world. (We will look at this connectivity in two weeks, in the Lecture entitled "LINK').

It is interesting to note that ARM does not make chips. They provide the ARM-Cortex M4 design to ST Micro (who pays a royalty to ARM for the IP). ST Micro then design all the other stuff around the ARM core to make their product. ARM was the first company to make this fabless IP business model work. It ended up with the world using ARM processor more than they use Intel processors because many foundries (i.e. those people who make the semiconductor chips) can design and make products with ARM.

The Pyboard



The Pyboard further add to the ST Micro chip by adding an accelerometer, a MicroSD card reader, all sort of power regulation and protection circuits.

This is a **QuickRef Guide** from MicroPython, the maker of the Pyboard. It shows which pins on the Pyboard is used for what. Mostof them are programmable, meaning that they have multiple functions. We enable them for a specify function as required. For example, in Lab 4, we used UART(6) for task 6, 7 and 8. This involves pins Y1 and Y2. If we were not using these pins for UART, we could program the pins to drive LEDs or one of the timer pins.

In the Team Project, you will only be using limited features on the Pyboard and the STM microcontroller. Lab 4 provides examples to you as to how to program the Pyboard to do things that are useful to your design projects. Don't worry about the rest – you can find those out for yourself if you have time.

Programming the Pyboard

- After power up, Pyboard will run the script boot.py on the flash drive (without SD card) or microSD card if it is there.
- **boot.py** is shown below (Lab1). This will led to running **main.py**.
- You can modify **main.py** to other scripts such as task1.py etc. as instructed in Lab 4.
- A better way to do thing is to use the following.
- Inside **main.py**, ask for a task number and use if-elif to select which script run:

```
task =  int(input("Enter task:  "))
if task == 1:
    execfile('task1.py')
elif task == 2:
    execfile('task2.py')
..........
```

```
import machine
import pyb
pyb.main('main.py') # main script to run after this one
```


In Lab 4, you brought to the surface of the BB the Pyboard. As soon as you plug in your computer, the Pyboard appears as a USB memory drive. You can see a file called boot.py. This is the program that Pyboard will first run. Inside the program is the function call: pyb.main('main.py'). pyb is the library that provides Pyboard specific functions. The function you use is pyb.main(), which direct the Pyboard to the main program that it should then execute.

I told you in the Lab instruction that you could change the name of main.py to something else, such as task1.py etc.

An even better way to do this (and I got this from Benedict Greenberg during the lab) is to do something shown on the right. Instead of changing the main program to run in boot.py, you could modify main.py as shown here. main.py now reads in the task number, and then use the standard Python call execfile(file_name) to redirect Python to run or execute another Python file.

The first line in this program prints a message on your terminal, and wait for you to type in a number. This number is store in the variable task.

The if-elif statement select which Python program to run. Now you just ask taskx.py to the USB disk as you progress in the Lab. Very elegant!

Micropython Documentation

MicroPython

Quick reference for the pyboard
General information about the pyboard
MicroPython tutorial
MicroPython libraries
The pyboard hardware
Datasheets for the components on the pyboard
Datasheets for other components
MicroPython license information
MicroPython documentation contents

Docs »

MicroPython documentation

Welcome! This is the documentation for MicroPython v1.8, last updated 01 Jun 2016.

MicroPython runs on a variety of systems and each has their own specific documentation. You are currently viewing the documentation for **the pyboard**.

Documentation for MicroPython and the pyboard:

Quick reference for the pyboard
pinout for the pyboard and snippets of useful code

Language Reference
information about MicroPython specific language features

General information about the pyboard
read this first for a quick overview

The pyboard hardware
schematics, dimensions and component datasheets

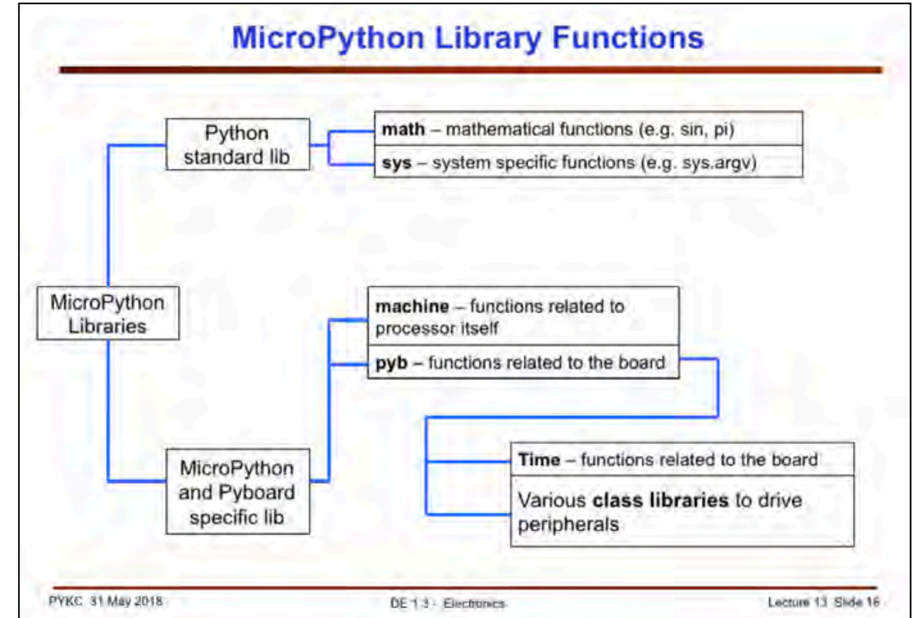
Tutorials and code examples

Guide for the pyboard on Windows

MicroPython is quite a large system and there are too much to learn. It is also good if you learn to read instructions from website, rather than just been spoon fed by me. So, go to:

http://docs.micropython.org/en/latest/pyboard/

You will see this page. There are lots of useful documents here. I will now give you an summary of the most important ones that are relevant to your Team Project.



MicroPython provides many library functions in order to make programming this board much easier. There are basically two main categories of library functions:

1. A subset of the standard Python library you can find everywhere (but implemented for MicroPython). Of these, the two that you would need are: math and sys.
2. There is a bunch of library functions that are written for the Pyboard specifically. The two that you need are: machine and pyb. Machine library provide top level control of the board, and pyb provide detail control. pyb is the most important library for you to know. Almost all functions we use so far are important from the pyb library.
3. Within pyb, there are some that are quite generic, such as Time (different from Timer). The functions provided by Time such as pyb.delay() and pyb.udelay() are most useful. They provide real time delay (quite accurate) in milliseconds and microseconds.
4. Then there is a large collection of peripheral specific libraies (they are written as object-oriented class libraries). You will be using many functions from this.

pyb - Class Library

pyb Classes

class Accel – accelerometer control
class ADC – analog to digital conversion
class DAC – digital to analog converson (2 channels)
class LED – LED objects to control on board LEDs
class Pin – control I/O pins (X1 to X12, X17 to X22, Y1 to Y12)
class Timer – control internal timers (14 timers)
class TimerChannel – setup a channel for a timer
class UART– two way serial communication (1 to 6)

The pyb class libraries includes those shown here. In Lab 4, you used the following library classes: LED, ADC, Pin (everything), Timer (these are functions to control internal timer circuits), TimerChannel (this is a further extension to control the timers, but at a high-level of abstraction) and UART. I have provided examples in Lab 4 for you.

You don't need to know all these libraries. You should learn using this approach:

1. Learn from the example code I gave you.
2. If these do not do what you need, be clear what function you need to perform.
3. Look for a function in the library that is likely to give you what you needed.
4. Failing that, ask me or your friends and classmates, or your team project supervisor.

A video on "How a CPU is made?"

This video explains how a CPU or a microprocessor is made. You can find this video on youtube:

https://www.youtube.com/watch?v=qm67wbB5GmI