COMP1521 24T2 Lec01

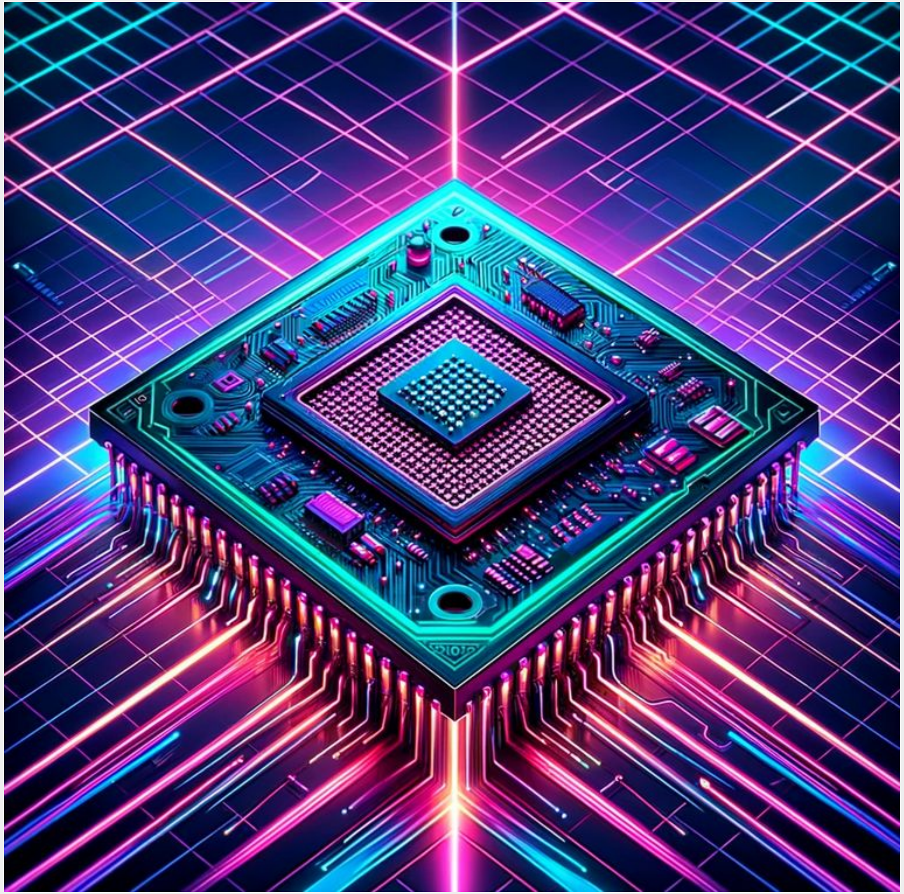
MIPS:

An Introduction

2024

Hammond Pearce

Adapted from Abiram's slides



MIPS

What is a computer?

- A machine that “computes”
- A machine that executes a program
- How do we make a machine that executes a program?

What is a program? How do they execute?

In COMP1[59]11:

- We run a compiler (dcc?)
- **./hello**
- profit ??

What's going on here? What's even in hello?

So what is a “program”??

- A program is a set of instructions and data

For example:

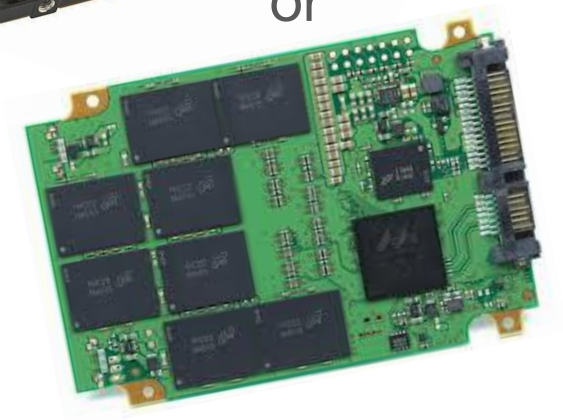
So how do we execute the program?

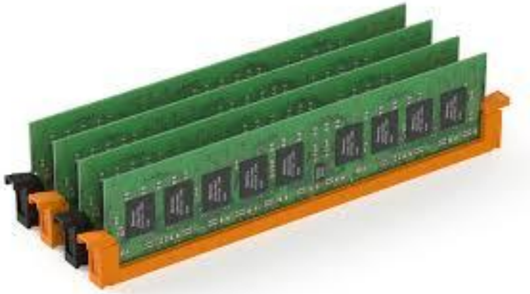
- The program is a set of instructions and data… somewhere
  - Maybe a “hard disk”
  - Long-term, **non-volatile**

- We load the program into “memory” - RAM!
  - RAM is like a massive 1D array which we divide into sections
  - It has addresses, which are like indexes into that array
  - RAM is **volatile**

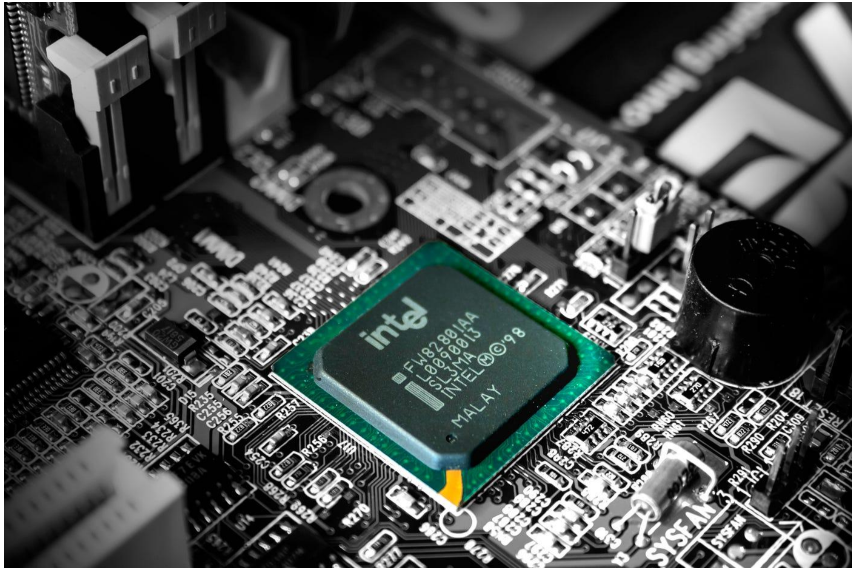
Disks and RAM

or





And then… the CPU “runs” the program!



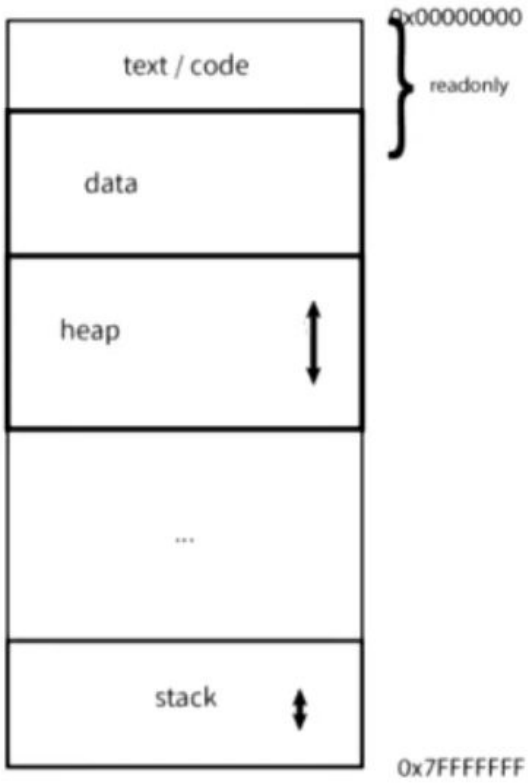
Programs and Program Memory

- A program contains information on how to set up memory
  - What instructions need to be followed?
  - What data do we need to load into the memory?
    - Variables?
      - Globals and locals

- Then, during operation, we might request *more* memory
  - malloc
  - So greedy

- Where do we put all these things?

A program's memory map



How does the CPU know what to do?

- There are a finite number of possible instructions
  - We assemble programs by combining the instructions in sequences

- E.g if we have just "x = a + b" - how do we get "y = a + b + c"?
  - temp = a + b
  - y = temp + c

- The CPU is built to execute all the possible instructions
- i.e. the CPU implements an "Instruction Set Architecture"

Some ISAs

MIPS, ARM, x86, Itanium, x86_64, Power, AVR, PIC, RISC-V …

↓

Lots of possible implementations

↓

Lots of possible uses/users

↓

E.g. games consoles: →

Year	Console	Architecture	Chip	MHz
1995	PS1	MIPS	R3000A	34
1996	N64	MIPS	R4200	93
2000	PS2	MIPS	Emotion Engine	300
2001	xbox	x86	Celeron	733
2001	GameCube	Power	PPC750	486
2006	xbox360	Power	Xenon (3 cores)	3200
2006	PS3	Power	Cell BE (9 cores)	3200
2006	Wii	Power	PPC Broadway	730
2013	PS4	x86	AMD Jaguar (8 cores)	1800
2013	xbone	x86	AMD Jaguar (8 cores)	2000
2017	Switch	ARM	NVidia TX1	1000
2020	PS5	x86	AMD Zen 2 (8 cores)	3500
2020	xboxs	x86	AMD Zen 2 (8 cores)	3700
2022	steam deck	x86	AMD Zen 2 (4 cores)	3500

What can instructions do?

- **Load/store**: Got data? Need to load it! Need to store it!
- **Computations**: eg. add, subtract, multiply, divide, bitwise
- **Branch**: jump to execute different instructions
  - Can’t have logic (eg. if statements) if our program continues linearly
- **System calls**: phone-a-friend
- **Coprocessor**: Do hard, special things needing special hardware
  - E.g. floating point math

Do we write the instructions directly?

- Not often…
  - But sometimes we do!
  - (Someone has to!)
- Instead, we tend to write in a higher-level *compiled* language
  - C, C++, D, Go, Zig, Rust, Java, Swift, and many more…
- A *compiler* will input programs in these languages and *output* the corresponding assembly instructions

Assembly

Instructions are really just 0s and 1s

- Would be a *pain* to read/write literal instructions
- Instead, we use **assembly** language to form a human-readable representation of each instruction
  - Each instruction we write in assembly language ***typically*** represents a single CPU instruction
  - An assembler translates this to binary CPU instructions

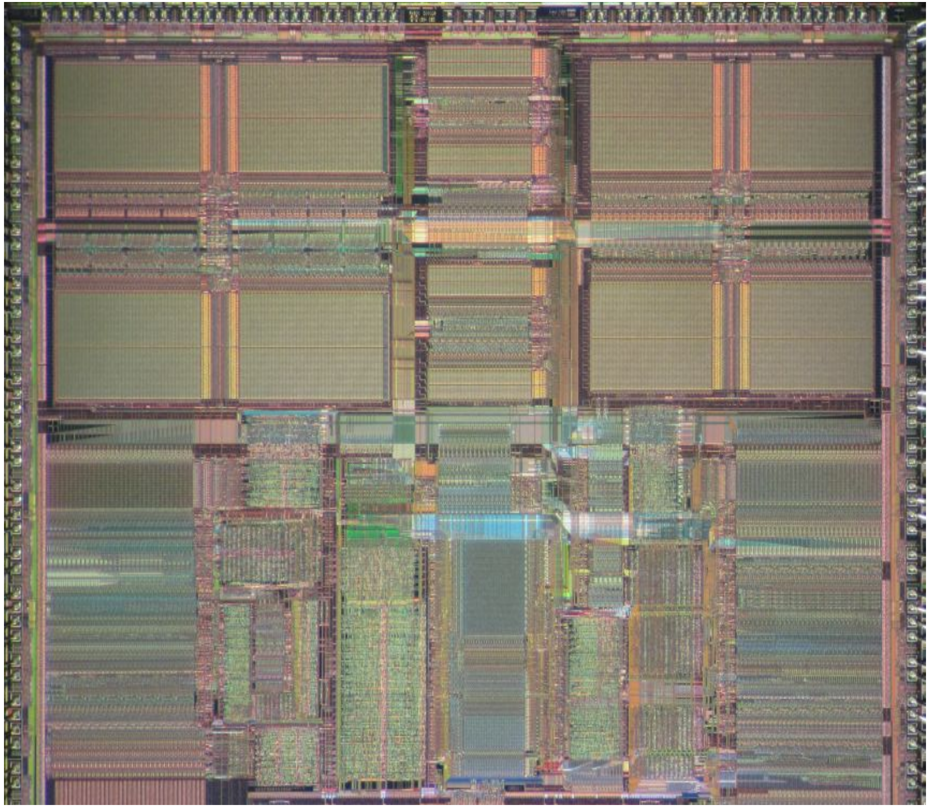
So, to recap: how do we make a program?

- We have a program in some language (e.g. C)
- We have a processor that runs some ISA (e.g. MIPS)
- We **compile** the program into **assembly** (and a binary)
- The binary is stored to a file

Then…

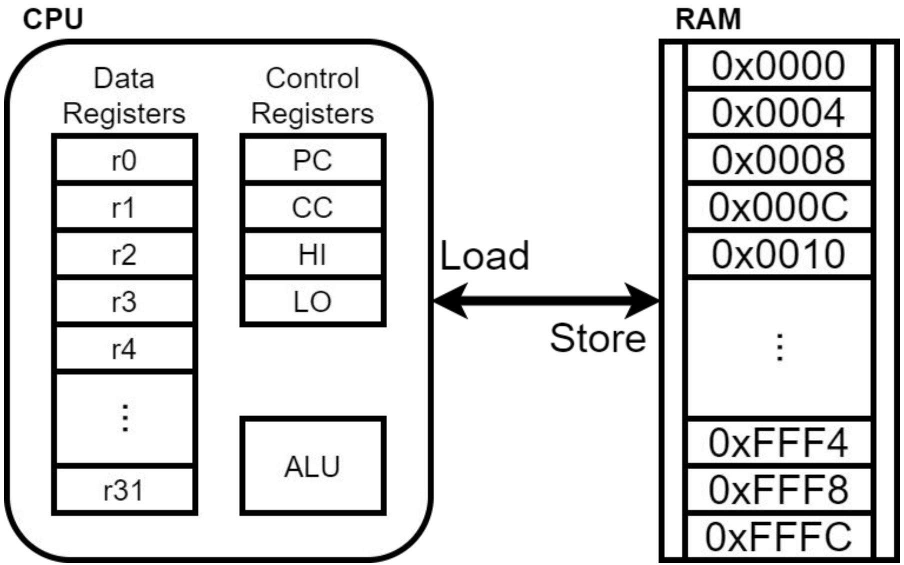
- The program is loaded into memory
- The CPU is pointed at the memory
- And we are off!

More about CPUs



What’s in there?

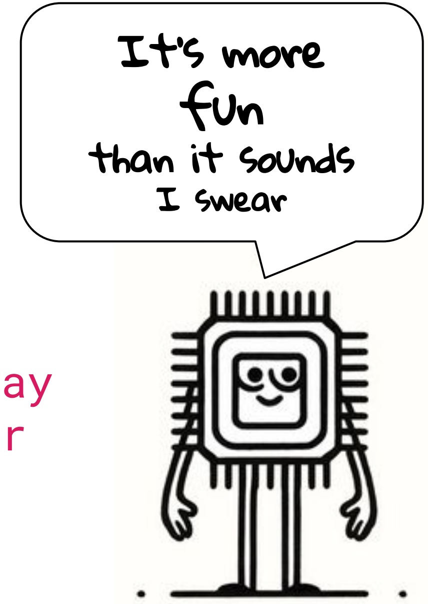
- a set of data registers
- a set of control registers
- a control unit
- an arithmetic-logic unit
- a floating-point unit
- caches
- connection to Memory/RAM



A day in the life of a CPU - as C code

```c
int program_counter = START_ADDRESS;

while (1) {
    // Fetch an instruction from memory
    int instruction = memory[program_counter];
    // Move to the next instruction
    program_counter++;
    // Execute the next instruction
    execute(instruction, &program_counter);
                  // ^ note: some instructions may
                  //   modify the program counter
}
```



So… writing instructions ourselves?

In this course we write assembly **ourselves** instead of compiling.

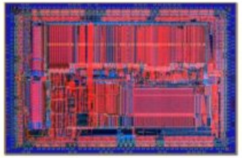
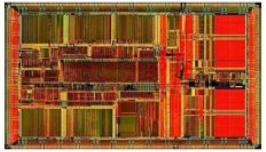
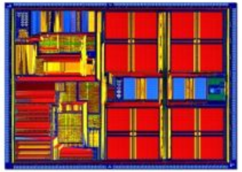
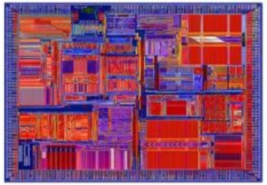
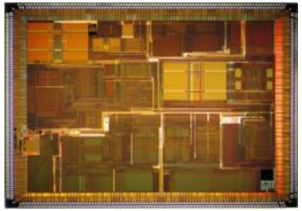
But why would anyone do that?

- To optimise code for performance
  - Less instructions = faster to execute = saving picoseconds!
- To write for edge cases not supported by compilers
  - eg. writing code to interact directly with a device (i.e. *drivers*)
- To learn how a compiled program executes
  - Primary reason in this course
  - Can be helpful when debugging
  - Also handy to identify security vulnerabilities and exploit binaries
- And sometimes, someone has to!
  - E.g. who's going to make your compiler in the first place?

So why “MIPS”?

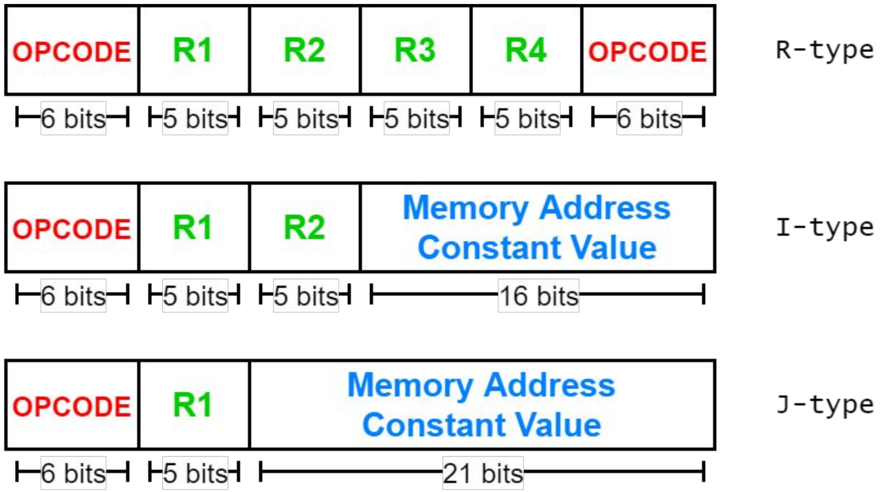
- Once used from game consoles to supercomputers
  - Still used in routers and TVs
- Considerable learning resources available
- Inspired many other ISAs
  - If you know MIPS, you can easily branch to ARM, RISC-V, and others
- All ISAs have tradeoffs
  - Some focus on performance and special features
- MIPS is “simple” yet powerful - good foundation for knowledge

More about MIPS

	MIPS R2000	MIPS R3000	MIPS R4000	MIPS R5000	MIPS R10000	MIPS R12000
						
Year	**1985**	**1988**	**1992**	**1996**	**1995**	**1998**
MIPS ISA	MIPS I (32-bit)	MIPS I (32-bit)	MIPS III (64-bit)	MIPS IV (64-bit)	MIPS IV (64-bit)	MIPS IV (64-bit)
Transistor count	110k	110k	2.3 – 4.6m	3.7m	6.8m	7.15m
Process node	2 µm	1.2 µm	0.35 µm	0.32 µm	0.35 µm	0.25 µm
Die size	80 mm^2	40 mm^2	84 – 100 mm^2	84 mm^2	350 mm^2	229 mm^2
Speed	12 – 33 MHz	20 – 40 MHz	50 - 250 MHz	150 – 266 MHz	180 – 360 MHz	270 – 400 MHz
Flagship devices	DECstation 2100 and 3100 workstations	***Sony PlayStation game console*** SGI IRIS and Indigo workstations ***NASA New Horizons space probe***	***Nintendo N64 game console*** Carrera Computers and DeskStation Technology PCs (Windows NT) SGI Onyx, Indigo, Indigo2, and Indy workstations	SGI O2 and Indy workstations Cobalt Qube servers HP LJ4000 laser printers	SGI Indigo2 and Octane workstations ***SGI Onyx and Onyx2 supercomputers*** NEC Cenju-4 supercomputers Siemens Nixdorf servers	SGI Octane 2, Onyx 2, and Origin workstations

What do MIPS instructions look like?

- 32 bits long
- Specify:
    - An operation
        - (The thing to do)
    - 0 or more operands
        - (The thing to do it over)



- For example:

`00100001000010010000000000001100`

`addi $t1, $t0, 12`

"But I don't have a MIPS CPU!"

- True (probably).
- We can't run our MIPS instructions directly on x86_64/ARM.
- But, we can emulate them using *mipsy*
- recreates the behaviour of a real MIPS CPU
  - written by Zac* (past course admin, now graduated/lecturing COMP6991)
  - can download on your own machine: https://github.com/insou22/mipsy/
  - comes with a **command-line interface** to run in your terminal
- **mipsy_web** runs entirely in your browser
  - by Shrey*, on course website: https://cgi.cse.unsw.edu.au/~cs1521/mipsy
- **vscode extension**
  - written by Xavier 🎉 - can download the 'mipsy editor features' extension

* some contributions from Josh Harcombe, Dylan Brotherston and Abiram

Can we write some MIPS?

Soon™

All about registers

- Most CPU architectures perform operations over **registers**
- They are part of the processor itself, not the memory
- Speed advantages:
  - Memory is fast, but not as fast as the CPU
  - Caches store some memory for faster access
    - Usually not as fast as registers!
- Simplifies processor design considerably

All about registers

- MIPS specifies 32 general-purpose registers
  - 32-bits each, same size as a typical C integer - coincidence?
  - Floating point registers (not used in COMP1521)
  - Hi/Lo special registers for multiply and divide (not important in this course)
  - Program counter
    - Keeps track of which instruction to fetch and execute next
    - Modified by branch and jump instructions

Registers

*Almost **all*** of our computations happen between registers!

Want to multiply 2 and 3 and store the result

Load 2 and 3 into registers:

```
li $t0, 2

li $t1, 3
```

And store the result:

```
mul $t2, $t0, $t1
```

More about registers

Registers are denoted by a $ and can be referred to using a number ($0…$31) or by symbolic names ($zero…$ra)

$zero ($0) is special!

- Always has the value 0 -> attempts to change it have no effect

$ra ($31) is also special!

- Directly affected by two instructions we use in Week 3

More about registers

Can use the other 30 registers however we want, technically, but:

There are conventions to prevent utter chaos and madness

(Discussed more in next week's tutorials and Week 3 lectures)

Number	Names	Conventional Usage
0	zero	Constant 0
1	at	Reserved for assembler
2,3	v0,v1	Expression evaluation and results of a function
4..7	a0..a3	Arguments 1-4
8..16	t0..t7	Temporary (not preserved across function calls)
16..23	s0..s7	Saved temporary (preserved across function calls)
24,25	t8,t9	Temporary (not preserved across function calls)
26,27	k0,k1	Reserved for Kernel use
28	gp	Global Pointer
29	sp	Stack Pointer
30	fp	Frame Pointer
31	ra	Return Address (used by function call instructions)

More about registers

Convention says `$t0` to `$t9` can be used however you want

Will also need `$v0`, `$a0`, `$ra` for certain things at the moment

Should not need to use any other registers (yet)

We will cover the other registers when we talk about functions in Week 3

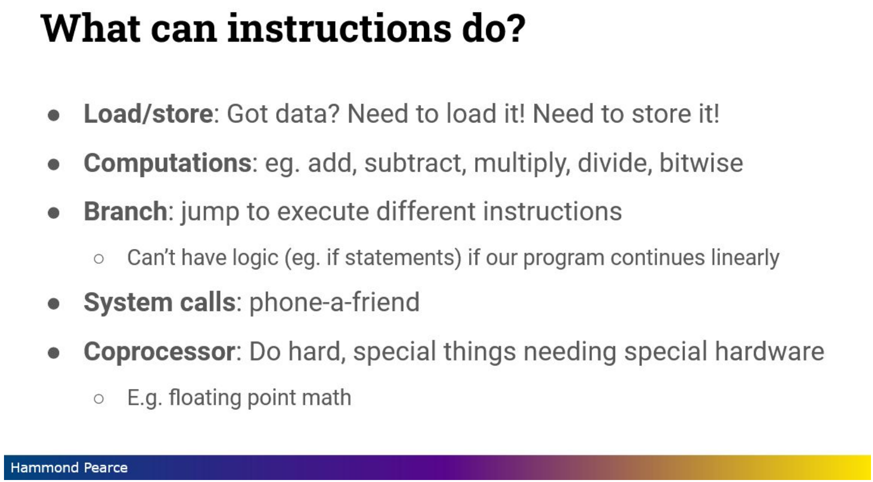
Now let's make something

Your turn

Our programs are useless

System calls

- We mentioned these earlier
- System call ==
  - Hi system friend
  - Can you do this thing for me
  - Thanks
- What sort of things?


What can instructions do?

- **Load/store**: Got data? Need to load it! Need to store it!
- **Computations**: eg. add, subtract, multiply, divide, bitwise
- **Branch**: jump to execute different instructions
  - Can't have logic (eg. if statements) if our program continues linearly
- **System calls**: phone-a-friend
- **Coprocessor**: Do hard, special things needing special hardware
  - E.g. floating point math

Hammond Pearce

System calls

- None of the instructions we have access to can interact with the outside world (eg. printing, scanning)
- Instead, we request the operating system to perform these tasks for us - this process is called a **system call**
- The operating system can access privileged instructions on the CPU (eg. communicating to other devices)
- *mipsy* simulates a very basic operating system
- Will explore real system calls in the second half of the course

Common mipsy syscalls

Service	$v0	Arguments	Returns
`printf("%d")`	1	int in `$a0`	
`fputs`	4	string in `$a0`	
`scanf("%d")`	5	none	int in `$v0`
`fgets`	8	line in `$a0`, length in `$a1`	
`exit(0)`	10	none	
`printf("%c")`	11	char in `$a0`	
`scanf("%c")`	12	none	char in `$v0`

We don't use syscalls 8 and 12 much in COMP1521

Most input will be integers

More ✨advanced✨ syscalls

Service	$v0	Arguments	Returns
`printf("%f")`	2	float in `$f12`	
`printf("%lf")`	3	double in `$f12`	
`scanf("%f")`	6	none	float in `$f0`
`scanf("%lf")`	7	none	double in `$f0`
`sbrk(nbytes)`	9	nbytes in `$a0`	address in `$v0`
`open(filename, flags, mode)`	13	filename in `$a0`, flags in $a1, mode $a2	fd in `$v0`
`read(fd, buffer, length)`	14	fd in `$a0`, buffer in $a1, length in $a2	number of bytes read in `$v0`
`write(fd, buffer, length)`	15	fd in `$a0`, buffer in $a1, length in $a2	number of written in $v0
`close(fd)`	16	fd in `$a0`	
`exit(status)`	17	status in `$a0`	

Probably only used for challenge exercises in COMP1521

The system call workflow

- We specify which system call we want in `$v0`
  - eg. `print_int` is syscall 1:

```
li $v0, 1
```

- We specify arguments (if any)

```
li $a0, 42
```

- We transfer execution to the operating system
  - The OS will fulfil our request if it looks sane

```
syscall
```

- Some syscalls may return a value - check syscall table

MIPS and mipsy documentation

Literally your best friend (it'll even be there for you in the exam 🥺)

COMP1521 - 23T2 Outline Timetable Forum

MIPS Instruction Set

An overview of the instruction set of the MIPS32 architecture as implemented by the mipsy and SPIM emulators. Adapted from reference documents from the University of Stuttgart and Drexel University, from material in the appendix of Patterson and Hennessey's *Computer Organization and Design*, and from the MIPS32 (r5.04) Instruction Set reference.

Now we can say hello world