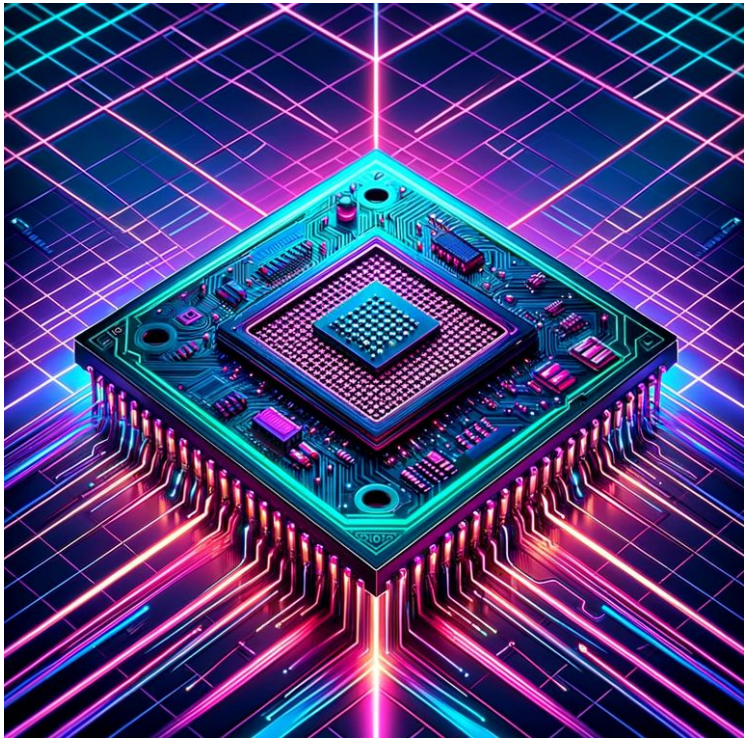
COMP1521 25T1

Week 1 Lecture 1

Course Introduction and MIPS Introduction

Today's Lecture

- Welcomes and Introductions
- How COMP1521 works
- How to get help
- How does a program run?
- A first look at MIPS assembler

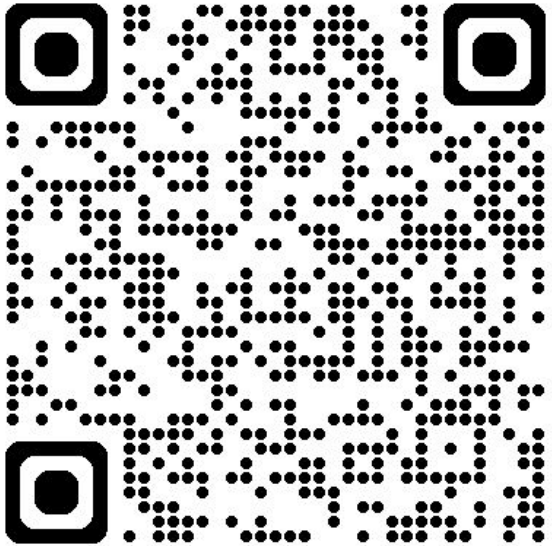


Please join the Youtube chat
and ask questions :)

Course Website

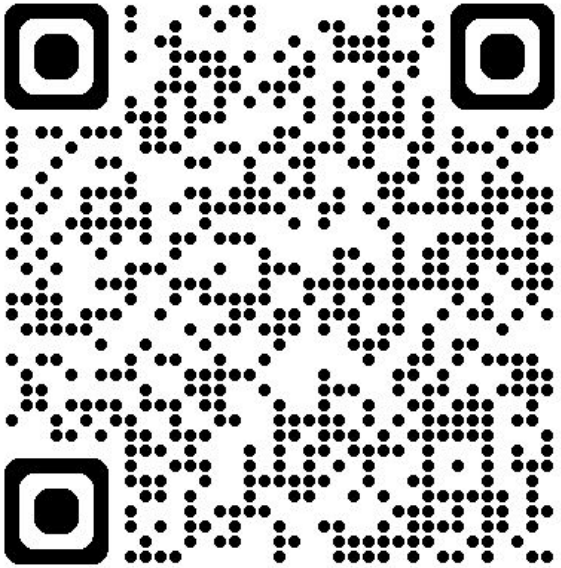
https://cgi.cse.unsw.edu.au/~cs1521/25T1

- All course information is on our course website
  - Please bookmark it
- Please read the course outline thoroughly
- We don’t use Moodle much. Just blackboard collaborate for online tutorials and help sessions



Course Staff: Who are we?

- Lecturer: Angela Finlayson
- Admins:
  - JJ Roberts-White
  - Anna Brew
  - Xavier Cooney
  - Abiram Nadarajah
  - Dylan Brotherston
- Lecture Moderators:
  - Tasfia Ahmed
  - JJ Roberts-White
- And an Amazing team of tutors!!!!



https://cgi.cse.unsw.edu.au/~cs1521/25T1/team/

COMP1521 Students: Who are you?

- Most students in this course have completed COMP1511 or COMP1911 which covers **fundamental** C programming.

- This week's tuts and labs:
  - review/strengthen assumed C knowledge
  - cover non-assumed C knowledge including recursion

- For anyone who needs more practice with C fundamentals:
  - Revision sessions in week 2 will help you to revise important concepts such as structs, pointers, malloc and recursion

Assumed C Knowledge

Design an algorithmic solution

Describe your solution in C code, using:

- variables, assignment, tests (`==`, `!`, `<=`, `&&`, etc)
- `if`, `while`, `scanf()`, `printf()`
- functions, `return`, prototypes, `*.h`, `*.c`
- arrays, structs, pointers, `malloc()`, `free()`

Not Assumed Knowledge

We do not assume you know:

- Recursion, for loops
  - These will be covered in week 1 tutorials
- Bit operations, File operations
  - These will be major topics taught in this course

You do not need to know:

- Linked lists

Course Goals

COMP1511/1911 ...

- gets you thinking like a **programmer**
  - How can we write a program?
- solving problems by developing programs
- expressing your solution in the C language

COMP1521 …

- gets you thinking like a **systems programmer**
  - How can we create systems that can run a program?
- and better able to reason about your C programs

Course Expectations

We also expect COMP1521 students to become more independent with their programming:

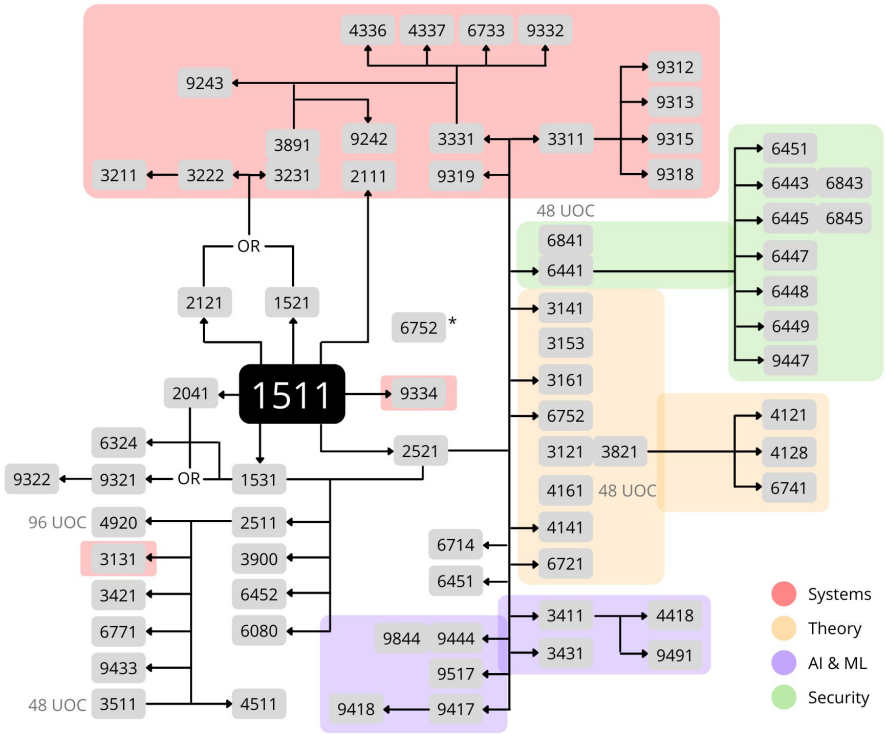
- further develop linux/command line skills
- further develop coding and debugging skills
- become less reliant on autotests and think more about your own test cases
- get used to reading manuals and documentation

COMP1511/1911

COMP1521

Course Context

The CSE Course Map



Major Themes

Goal: you are able to understand execution of software in detail

- software components of modern computer systems
- how C programs execute (at the machine level)
- how to write (MIPS) assembly language
- how computers represent data including integers & floats & emoji 😀🎈💻
- how operating systems are structured
- Unix/Linux system-level programming particularly file operations
- introduction to processes, thread and concurrency

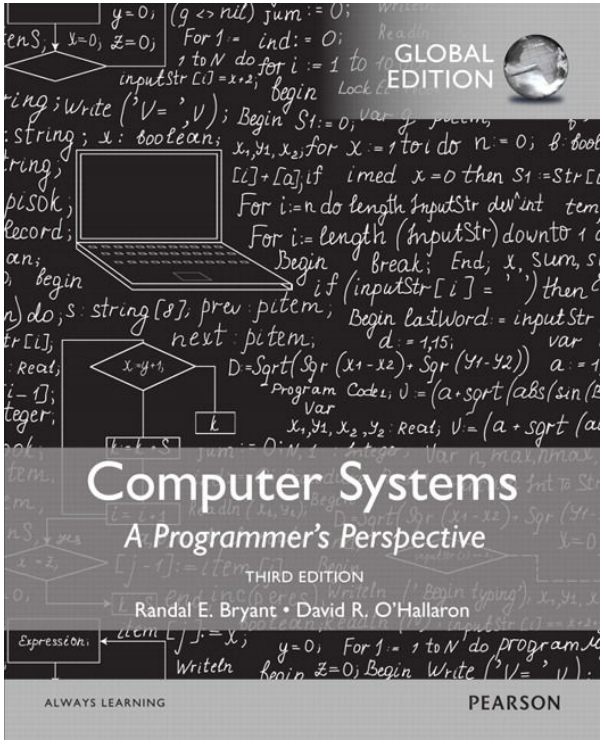
Textbook

There is no prescribed textbook for COMP1521.

Recommended reference:
Computer Systems: A Programmer's Perspective, Bryant and O'Hallaron

- covers most topics, and quite well
- but uses a different machine code

Available in UNSW Bookshop



Systems and Tools

- All tools available on the **CSE lab** machines (Debian Linux)
  - can use **VLAB** or **SSH** to connect to CSE from home
- Compilers:
  - **`dcc`** on CSE machines (**`clang`** or **`gcc`** elsewhere)
- Assembly language:
  - **`mipsy`** (**`mipsy_web`** online, **`vscode`** extension)
- Use your own favourite text editor:
  - **`vscode`**, **`ed`**, **`vim`**, **`emacs`**, **`nano`**, **`gedit`** etc.
- Other tools: **`make`**, **`man`**, **`bc -ql`**, **`python3`**, etc.
- Learn to love the **shell** and **command-line** ... very useful!

The Linux Manual (man)

The linux manual (**`man`**) is divided into the following sections:

- Section 1: Executable programs or shell commands eg. **`ls`**, **`cp`**
- Section 2: System calls (we will be looking at many of these in later weeks)
- Section 3: Library calls eg. **`strcpy`**, **`scanf`**

For example:

To find information about the C function getchar type

```
man 3 getchar
```

The Linux Manual (man)

- There are also other sections we won't be using so much
- You can find more information about man using the command **`man man`** which shows the manual page about the manual.
- You can get more information about individual sections by using **`man 1 intro`**, **`man 2 intro`** etc.

Advice: **man** will be available in the exam. Get used to using it!

Course Format

- Weekly Lectures 2 x 2 hours
- Weekly tut/labs 3 hour blocks
- Weekly tests done in your own time starting in week 3
- 2 Major Assignments
- 1 Final Exam in person

Lectures 2x2 Hours a Week

Monday 14:00 - 16:00: Ainsworth G03 (K-J17-G03)

- In Person and Live Streamed via YouTube (with live chat)
- There is usually space in lecture hall so come along even if you are in webstream!!

Wednesday 14:00 - 16:00:

- Live streamed via YouTube (with live chat)

All lectures recorded!

Lectures 2x2 Hours a Week

- Present a brief overview of theory
- Focus on practical demonstrations of coding
  - Problem-solving, testing, debugging
- If you have a question during the lecture:
  - Put your hand up and ask
  - Ask in live chat
- Please be respectful of others - everyone is here to learn
  - Don’t be noisy
  - Be kind to one another in the chat and of course in person too :)

Lectures 2x2 Hours a Week

- Resources:
  - All lectures recorded and linked from course home page.
  - Lecture slides available on the web before lecture.
  - Live code from lectures released during/after lecture
  - Each lecture topic has extra polished code examples and more detailed course notes available too

Tut-labs

- 3-hour tut-labs
  - start week 1
  - run each week (except week 6)
- Each class is a 1 hour tutorial, followed by a 2 hour lab
- Most of our tut-labs are face to face classes
- Online tut-labs are delivered via Blackboard Collaborate (accessed via Moodle)

Tutorials

- To get the best out of tutorials
  - attempt the problems yourself beforehand
  - not marked, and no submission
  - but you will learn more if you try the problems yourself
  - extra questions you can use for revision that won't get covered in class time

Do **not** keep quiet in tutorials: talk, discuss, ask questions, answer questions

Labs

Each tutorial is followed by a two-hour lab class.

- Several exercises, mostly small coding tasks
- Build skills needed for assignments, exam
- Done **individually**

Submitted via `give` before Monday 12:00 (midday) in the following week.

Lab 1 is an exception. It is due Monday 12:00 (midday) week 3.

Labs

Automarked (with partial marks) : 15% of final mark

- There will be seen autotests and unseen autotests

Labs may include challenge exercises:

- may be silly, confusing, or impossibly difficult
- almost full marks (95+%) possible **without** completing any challenge exercises

Flexibility Week and Public Holidays

Flex week (Week 6):

- No lectures, tutorials or labs
- There may be optional revision activities and help sessions

Public Holidays:

- Lecture will be pre-recorded to make up for Monday Week 10
- An alternative time for tutorials will be arranged by your tutor and/or you may attend another tut/lab in the same week if for Friday Week 9, Monday Week 10 and Friday Week 10.

Tests

From week 3, and every week after (including week 6):

- Released on Thursday 3pm
- due exactly one week later
- Submitted via **`give`**

Gives an immediate reality-check on your progress

Tests

Conditions:

- done in your own time under self-enforced **exam conditions**.
- time limit of 1 hour
- can keep working after hour for 50% cap on mark

Marking:

- automarked (with partial marks)
- best 6 of 8 tests contribute 10% of final mark
- any violation of test conditions -> 0 for whole component

Assignments

Ass1: Assembly (MIPS) Programming, weeks 3 - 5, 15%

Ass2: C Systems Programming, weeks 7 - 9, 15%

Assignments give you experience with larger programming problems than lab exercises

Assignments will be carried out **individually**

Assignment Tips

- They **always** take longer than you expect.
- Don't leave them to the last minute.
- Get help from appropriate sources - help sessions, forum, tutors in your lab
- Don’t copy or use generative AI
- Standard UNSW late penalties apply
  - 5% per day for 5 days, computed hourly
  - The penalty is 5% of the maximum possible assignment mark
  - The penalty is deducted from your actual mark

Final Exam

In-person 3-hour practical exam: in CSE labs, on CSE lab computers

- You must be in Sydney to sit the exam during the exam period
- limited environment: you get the tools and software of a lab computer, not your own computer
- You don't get access to your normal CSE account, so no custom configuration files and no course website available.
- no `dcc-help` or `autotest-help`
- hurdle: you must score 18+/45 (40%) on the final exam to pass course

Assessment

- 15% Labs
- 10% Tests
- 15% Assignment 1 --- due end of week 5
- 15% Assignment 2 --- due start of week 10
- 45% Final Exam

Above marks may be scaled to ensure an appropriate distribution

Assessment Hurdle

To pass, you must:

- score >= 50/100 overall
- score >= 18/45 on final exam

For example if you get:

- 55/100 overall in the course
- 17/45 on final exam

You will get a grade of **55 UF**
You will not get a grade of 55 PS

Code of Conduct

- CSE offers and inclusive learning environment for all students
- In anything connected to UNSW, including social media, the following are considered to be student misconduct and won't be tolerated
  - racist/sexist/offensive language or images
  - Sexually inappropriate behaviour
  - bullying , harassing or aggressive behaviour
  - Invasion of privacy
- Show respect to your fellow students and the course staff.

Plagiarism

- Cheating of any kind constitutes academic misconduct and carries a range of penalties
- Examples academic misconduct:
  - Groupwork on individual assignments (discussion OK)
  - Allowing another student to copy your work
  - Getting hacker cousin to code for you
  - Purchasing a solution to the assignment.

Plagiarism

- Labs, Tests and Assignments must be entirely your own work
  - You can not work on labs tests or assignments as a pair or in a group
- Plagiarism will be checked and penalised
  - Plagiarism may result in suspension from UNSW
  - Scholarship students may lose scholarship
  - International students may lose visa
  - Supplying your work to any other person is also considered plagiarism
- More information can be found in the course outline

Generative AI

- You may be able to see the issues with these AI generated images
  - Will you see all issues with AI generated MIPS or C code ?
- Will AI generate the same code for other students?
- What will you do in the final exam without AI?

Use of Generative AI Tools

- Use of generative AI tools including GitHub Copilot, ChatGPT **not permitted in COMP1521**
  - later courses will likely allow use of these tools
- dcc-help, autotest-help are specialized generative AI tools designed for CSE students
  - use of dcc-help and autotest-help **is permitted** in COMP1521
  - however dcc-help and autotest-help will **not be** available in the exam

Use of Generative AI Tools

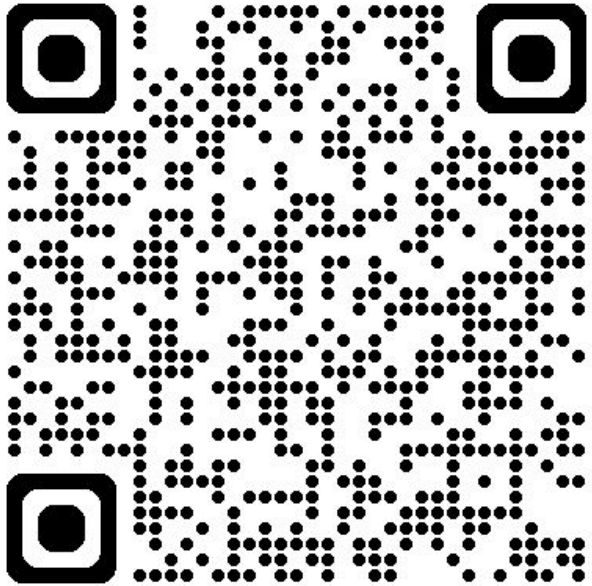
- Generative AI tools, e.g. GitHub Copilot, ChatGPT have great potential to assist coders however:
  - Code they generate often has subtle errors & security vulnerabilities
  - often generate poor code or unusual code
  - expert coders (hopefully) can spot these problems
  - students learning something new don't yet have this understanding
  - Use of tools such as Copilot, ChatGPT may slow you getting this understanding

How to Pass this Course

- coding is a **skill** that improves with practice
  - the more you practice, the easier you will find assignments/exams
  - do the lab exercises **yourself**
  - do the weekly tests **yourself**
  - do the assignments **yourself**
  - practice programming outside classes
  - do revision lab exercises
  - do extra revision tutorial questions like a mini prac exam
- Get help when needed from course staff!

Course Content Related Help

- Ask questions in lectures and in lecture chat
- Ask Questions in tuts and labs!
- **Forum:**
  - Post all your questions here
  - Feel free to answer other's questions
  - Don't post your code publicly in the forum



https://discourse01.cse.unsw.edu.au/25T1/COMP1521/

Course Content Related Help

- **Help Sessions:**
  - Good place to get one-on-one help outside of normal lab/tutorial times
  - There are optional drop in sessions
- **Revision Sessions:**
  - Optional group sessions to revise relevant topics
  - Booking required
  - Week 2: C revision (2d arrays strings, pointers, structs, malloc) and recursion

Schedules coming out soon

Admin Related Help

- **Course Administration Issues:**
  - Email: cs1521@cse.unsw.edu.au
- **Enrollment Issues:**
  - https://nucleus.unsw.edu.au/en/contact-us
- **cse course account issues**: CSE Help Desk
  http://www.cse.unsw.edu.au/~helpdesk/
- **Special consideration:**
  - https://student.unsw.edu.au/special-consideration
- **Equitable Learning Plans:**
  - https://www.student.unsw.edu.au/equitable-learning

Acknowledgement

Course Material has been drawn from:

- **Introduction to Computing Systems**: **from bits and gates to C and beyond**, Patt and Patel
- **The Elements of Computer Systems: Building a modern computer system from first principles**, Nisan and Schocken
- COMP2121 Course Web Site, Parameswaran and Guo
- Past COMP1521 lecturers, admin, and tutors

Always give credit to your sources

MIPS: An Introduction

Adapted from Abiram Nadarajah, Hammond Pearce,
Andrew Taylor and John Shepherd's slides

What is a program? How do they execute?

In COMP1511/1911:

- We run a compiler (dcc?)
  - `dcc -o hello hello.c`
- We run our program
  - `./hello`

What's going on here? What's in hello? Where is it stored?

What is a program? Where is it stored?

- A program is a set of instructions and data
  - In binary format (0s and 1s)
- A program is often stored as a file on a “hard disk drive” (HDD) or “solid state drive” (SSD)
  - Long-term, **non-volatile (keeps contents when power goes off)**

HDD

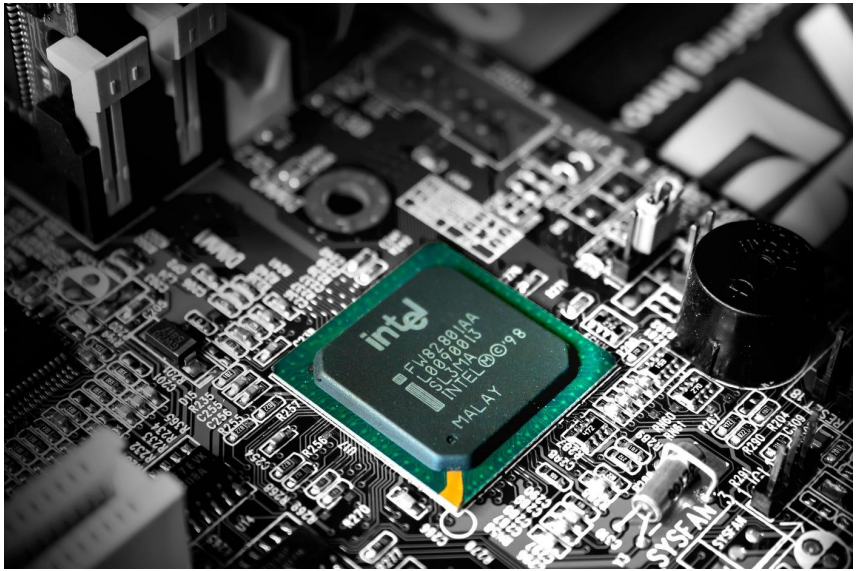
SSD

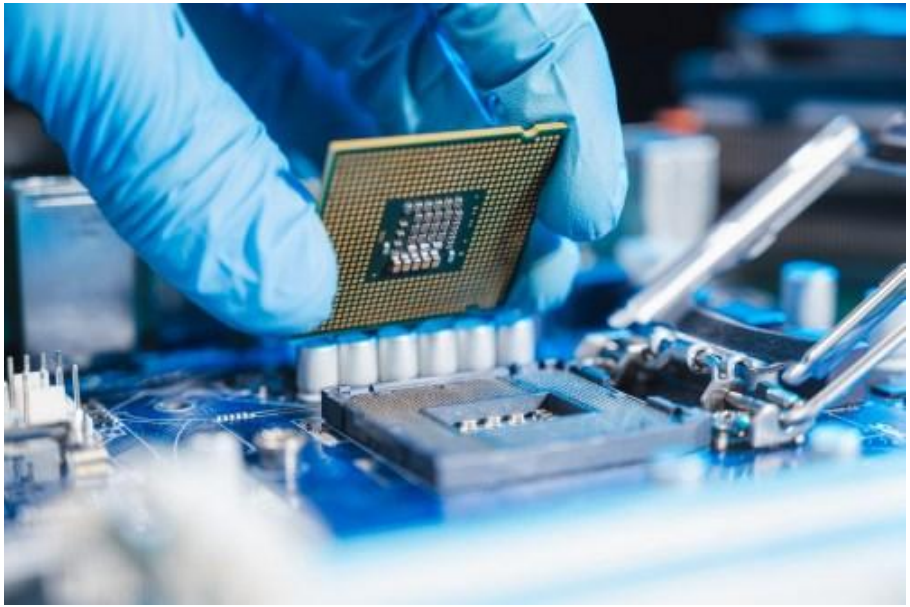
So how do we execute the program?

- The program needs to be loaded into **memory** - RAM!
  - RAM is like a massive 1D array
  - It has addresses, which are like indexes into that array
  - RAM is much faster than SSD or HDD, but more expensive
  - RAM is **volatile**
    - **Power goes off and everything is lost from RAM**

RAM

And then… the CPU executes the program!





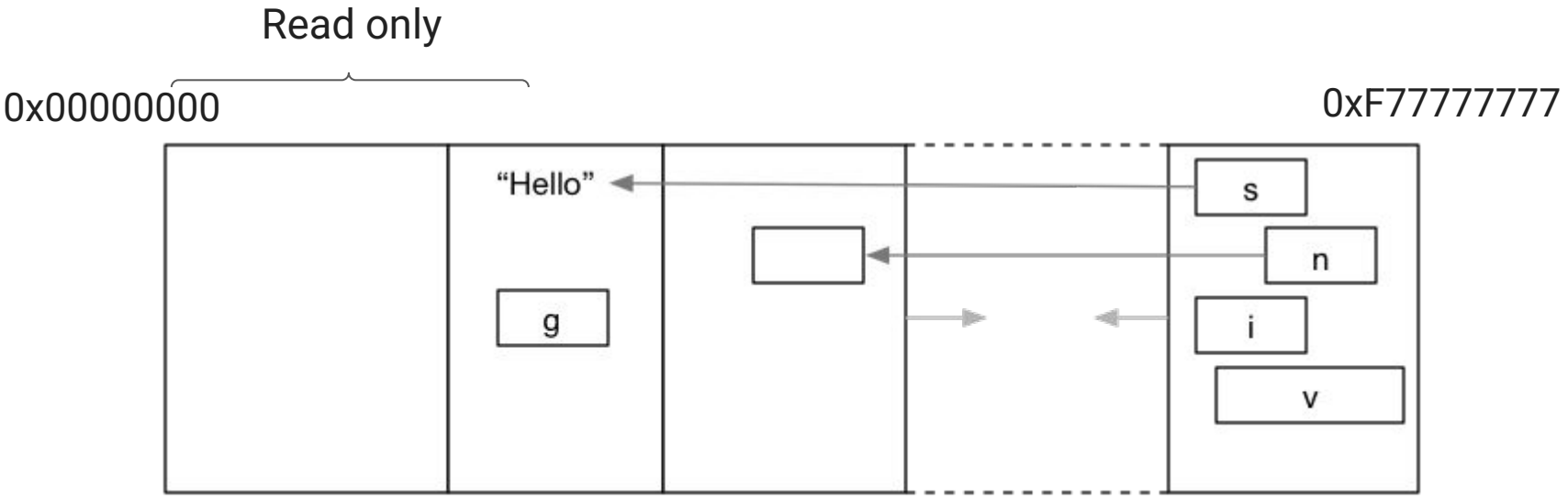
Loading our C program into Memory

- **Programs** contain information that needs to be loaded into the appropriate segments of memory so the program can execute.
- Segments include
  - **Text/code** segments:
    - Stores program instructions
    - Typically readonly and fixed size
  - **Data** segments:
    - Readonly section for string literals
    - Writable section for global variables
    - Fixed size

C Memory during Program execution

- **Heap**:
  - dynamically allocated memory
  - may grow when we malloc
  - may shrink when we free
- **Stack**:
  - local variables, parameters automatically managed
  - grows when functions are called
  - shrinks when functions return

C Program Memory Map



text/code

machine code for program instructions

data

global vars and string literals

heap

malloced things

stack

local vars and parameters

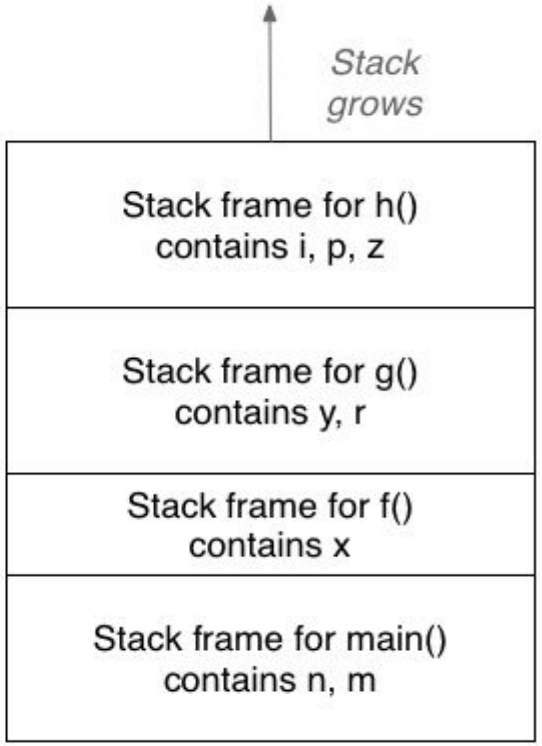
The Stack

```c
int main(void) {
    int n, m;
    n = 5;
    m = f(n);
    return 0;
}

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

```c
int g(int y) {
    int r = 4 * h(y);
    return r;
}

int h(int z) {
    int i;
    int p = 1;
    for (i = 1; i < z; i++) {
        p = p * i;
    }
    return p;
}
```



Infinite Recursion Demo

A “good” way to use up the stack and crash your program is to “accidentally” create “infinite” recursion.

Recursion is when a function directly (or indirectly) calls itself

```
// A recursive function that has no stopping condition
void f(int x) {
    printf("%d\n", x);
    f(x + 1);
}
```

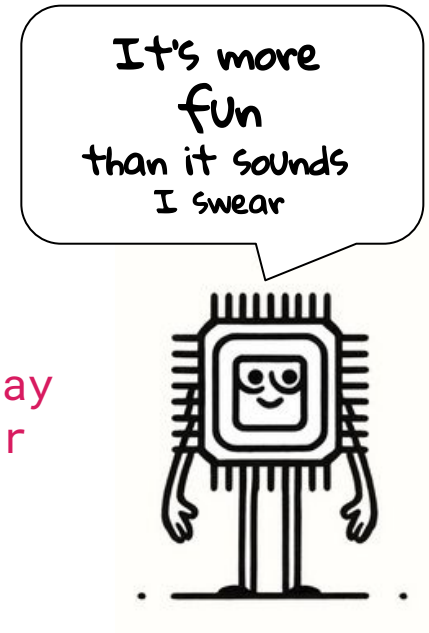
The CPU

- We have our instructions in memory (RAM)
- The CPU can
  - **fetch** an instruction from memory
  - **decodes** the instructions to work out what it should do
  - **executes** the instruction!

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
}
```



What can instructions do?

- **Computations**: eg. add, subtract, multiply, divide, bitwise
- **Load/store**: Load data from RAM! Store data to RAM!
- **Branch**: jump to execute different instructions
    - Can’t have logic (eg. if statements, while loops) if our program continues linearly
- **System calls**: request to the operating system to do something
- Many more things too!

Machine Code vs Assembly Code

Machine Code Instructions are really just 0s and 1s (binary data)

- 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 code ***typically*** represents a single CPU instruction
    - An assembler translates the assembly code to binary CPU instructions

Example Assembly Code Instruction

- For example: We might write in assembly:

```
addi $t1, $t0, 12
```

- And the assembler might generate the following machine code instruction:

```
00100001000010010000000000001100
```

- CPUs can't run assembly code directly; they can only execute machine code

Compiling to Assembly Code

- We usually just compile our code in one step to create our executable program.
  - `gcc -o hello hello.c`
- When we compile our code, the compiler first generates assembly code.
- To see this intermediate step we can type in:
  - `gcc -s hello.c`
  - and the assembly code it produces is in `hello.s`
- Will this generate the same assembly code on a different machine?

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

- We have a program in some language (e.g. C)
- We **compile** the program into **assembly** and it is assembled into a binary
- The binary is stored to a file

Then to execute it…

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

Writing Assembly code

- Usually we tend to write in a higher-level *compiled* language
  - C, C++, Go, Rust, Java, Swift, and many more…
  - A **compiler** will input programs in these languages and *output* the corresponding assembly instructions
- In this course we write assembly code **ourselves**
  - The main reason in this course is to **understand** how a compiled program executes
    - Can be helpful when debugging
    - Also handy to identify security vulnerabilities and exploit binaries (see COMP6447)

Writing Assembly Code

- Other reasons for writing assembly code:
  - To **optimise** code for performance
    - Less instructions = faster to execute = saving picoseconds!
  - Sometimes it's necessary
    - eg. writing code to interact directly with a device (i.e. *drivers*)
  - And sometimes, someone has to!
    - e.g. who's going to make your compiler in the first place?

Instruction Set Architectures (ISAs)

- Different types of CPUs implement different Instruction Set Architectures (ISAs)
  - In other words different types of CPUs may speak different languages or understand different sets of instructions
- ISAs define a finite set of instructions
  - These “simple” instructions can be combined to compute anything
- Examples of ISAs are
  - x86, ARM, RISC-V, MIPS

MIPS

So why MIPS?

- In COMP1521 we learn the **MIPS** instruction set architecture
- 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
- MIPS is **simple** yet powerful - good foundation for knowledge

But I don’t have a MIPS CPU!

- True (probably).
- Your laptop probably has x86 (PCs or older Mac) or ARM (newer Mac)
- We can’t run our MIPS instructions directly on our CPUs.
- But, we can emulate them using *mipsy*
  - software that recreates the behaviour of a real MIPS CPU
  - written by Zac* (past course admin, now graduated/lecturing COMP6991)
  - can run on CSE machines (including vlab)
  - can also download on your own machine: https://github.com/insou22/mipsy/
  - comes with a **command-line interface** to run in your terminal

Running a mips program

- **mipsy command line**
  - `1521 mipsy hello.s`
- **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

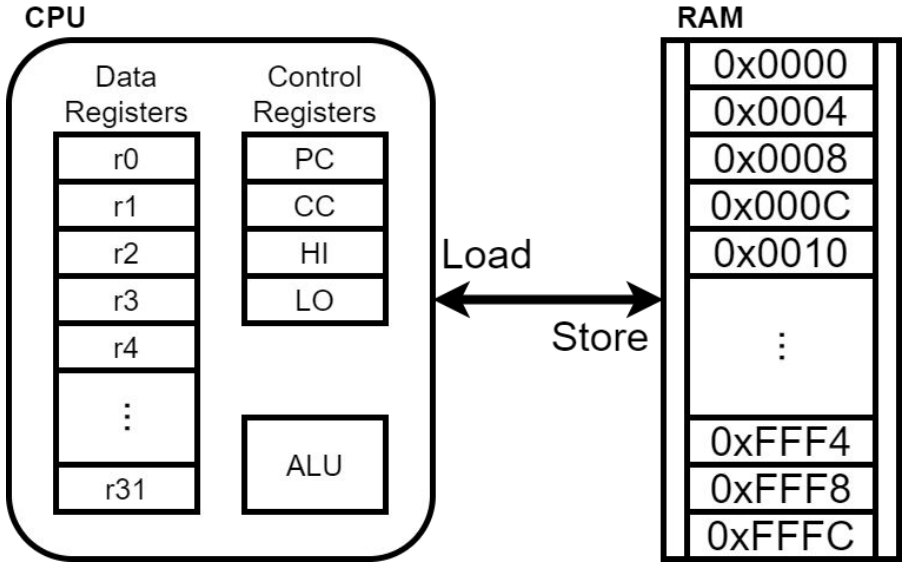
* some contributions from Josh Harcombe, Dylan Brotherston and Abiram

Can we write some MIPS?

Soon™

What's in a MIPS CPU?

- 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



Registers

- Most CPU architectures perform operations over **registers**
- They are part of the processor itself, not the memory
- Speed advantages:
  - Memory is fast, CPU is faster!
- There are only a small number of registers
- Values stored in memory must be loaded into registers for the CPU to perform computations on them.

MIPS 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

MIPS registers to use for now

- For now we will mainly use $t0 to $t9 registers for general purpose calculations
- Will also need `$v0`, `$a0` for certain things too.
- $zero ($0) is special!
  - Always has the value 0 -> attempts to change it have no effect
- $ra is also special!
  - We use it at the end of every program

MIPS Computations with 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
```

Let’s try it!
Open up mipsy_web and code along!

Simple Program Template

Here is a bare bones template to put instructions in to run them:

```
main:
        # YOUR CODE GOES IN HERE


        li      $v0                 # return 0
        jr      $ra
```

Your turn

- Code this up in mipsy_web.
  - Set $t0 to 10
  - Set $t1 to 7
  - Subtract $t1 from $t0 and store in $t2
  - Add 5 to $t2

What expression is this equivalent to?

Do you end up with the correct answer in $t2?

But how can we do input and output?

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

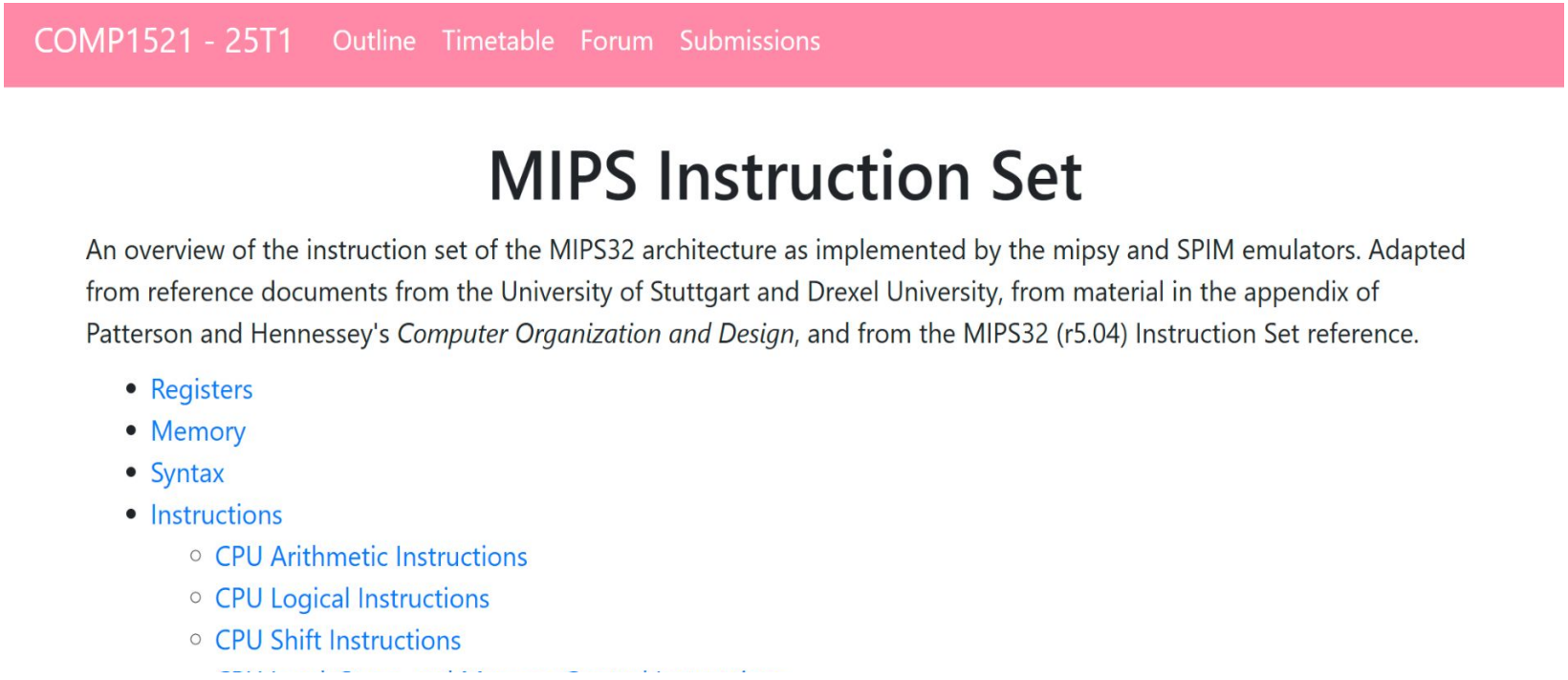
Let’s try to print out the number 42

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 fulfill 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 🥺)



Aside: Hexadecimal

0x in C and mipsy means hexadecimal.

Hexadecimal uses 16 digits. It uses 0-9 then A-F

We will learn more about this later in the course.

Decimal	Hexadecimal	Decimal	Hexadecimal
0	0	10	A
1	1	11	B
2	2	12	C
3	3	13	D
4	4	14	E
5	5	15	F
6	6	16	10
7	7	17	11
8	8	18	12
9	9	19	13

Aside: Hexadecimal

We often use Hexadecimal to represent addresses and other binary data like instructions.

- Easier for humans to read than binary
  - 8 hex digits can represent 32 bits
- Maps more nicely to binary than decimal

What did we learn today?

- **Admin:** How the course is run
- **Concepts:** How programs run!
- **Introduction to MIPS:**
  - Running MIPS programs
  - Writing simple programs with simple instructions
  - Simple system calls to print out data

What will we learn next lecture

- **MIPS Basics:**
  - More MIPS instructions and examples
  - Using system calls to read in integer and character data
  - Understanding how to work with strings and how hello.s works
- **MIPS Control:**
  - if statements
  - loops

Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/KYZBvyLhED

Reach Out

Content Related Questions:
Forum

Admin related Questions email:
cs1521@cse.unsw.edu.au

Student Support | I Need Help With…

My Feelings and Mental Health
Managing Low Mood, Unusual Feelings & Depression

- **Mental Health Connect** — student.unsw.edu.au/**counselling** Telehealth
- **Mind HUB** — student.unsw.edu.au/**mind-hub** Online Self-Help Resources
- **In Australia Call Afterhours UNSW Mental Health Support Line** — 1300 787 026 5pm-9am
- **Outside Australia Afterhours 24-hour Medibank Hotline** — +61 (2) 8905 0307

Uni and Life Pressures
Stress, Financial, Visas, Accommodation & More

- **Student Support Indigenous Student Support** — student.unsw.edu.au/**advisors**

Reporting Sexual Assault/Harassment

- **Equity Diversity and Inclusion (EDI)** — edi.unsw.edu.au/**sexual-misconduct**

Educational Adjustments
To Manage my Studies and Disability / Health Condition

- **Equitable Learning Service (ELS)** — student.unsw.edu.au/**els**

Academic and Study Skills

- **Academic Language Skills** — student.unsw.edu.au/**skills**

Special Consideration
Because Life Impacts our Studies and Exams

- Special Consideration — student.unsw.edu.au/**special-consideration**