

COMP1521 26T1

Week 5 Lecture 1

Bitwise Operators and Floating Point

Assignment 1 is due Friday 6pm

Week 4 test: due thursday 9pm (MIPS basics, control, arrays)

Week 6 Flexibility Week

Week 6 (next week) is flexibility week so nothing is due then!

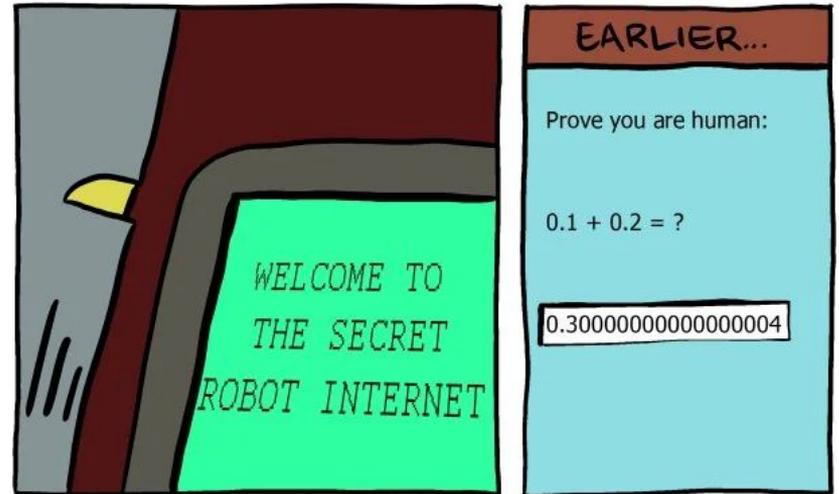
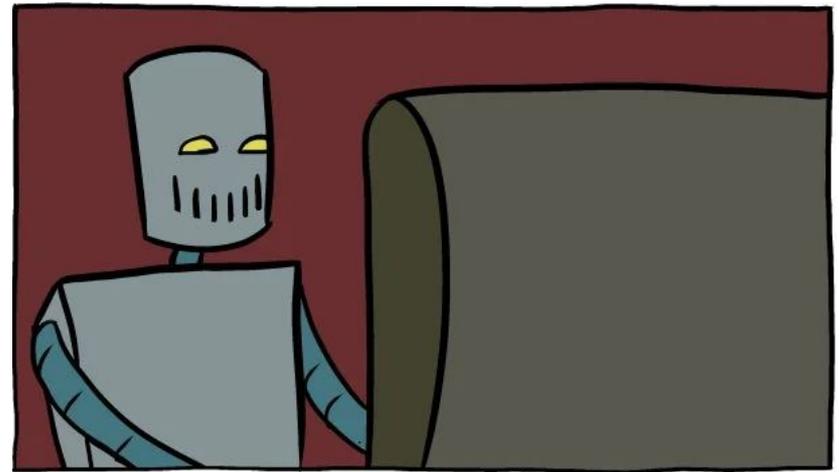
Week 5 lab: due monday midday week 7

Week 5 test: due thursday 9pm week 7 (MIPS strings)

Week 6 test: due thursday 9pm week 7 (bitwise operators C)

Today's Lecture

- Bitwise Operators
 - Recap
 - MIPS examples
 - C Coding examples
- Floating Point Representation



Recap Demo: bitwise.c

```
$ gcc bitwise.c print_bits.c -o bitwise
$ ./bitwise
Enter a: 23032
Enter b: 12345
Enter c: 3
    a = 0101100111111000 = 0x59f8 = 23032
    b = 0011000000111001 = 0x3039 = 12345
    ~a = 1010011000000111 = 0xa607 = 42503
a & b = 0001000000111000 = 0x1038 = 4152
a | b = 0111100111111001 = 0x79f9 = 31225
a ^ b = 0110100111000001 = 0x69c1 = 27073
a >> c = 0000101100111111 = 0x0b3f = 2879
a << c = 1100111111000000 = 0xcfc0 = 53184
```

cgi.cse.unsw.edu.au/~cs1521/26T1/topic/bitwise_operations/code/bitwise.c

Recap Exercise 1

Given the following declarations:

```
// a signed 8-bit value
uint8_t x = 0x55;
uint8_t y = 0xAA;
```

What is the value of each of these expressions?

```
uint8_t a = x & y;
uint8_t b = x ^ y;
uint8_t c = x | y;
uint8_t d = ~x
```

```
uint8_t e = x >> 1;
uint8_t f = y >> 2;
uint8_t g = y << 2;
```

Recap Exercise 2:

If bit positions are 0 to 7, with 0 being the least significant bit, how can we:

- 1) **Check** the bit at position **3** of a `uint8_t`?
- 2) **Set** the bit at position **3** of a `uint8_t`?
- 3) **Unset** the bit at position **3** of a `uint8_t`?

- 4) **Check** the bit at position **n** of a `uint8_t`?
- 5) **Set** the bit at position **n** of a `uint8_t`?
- 6) **Unset** the bit at position **n** of a `uint8_t`?

Code Demos

- `pokemon.c`
- `set_low_bits0.c`
- `set_low_bits.c`
- `set_bits_in_range.c`
- `extract_bits_in_range.c`
- `bitset.c` (left for you to look at)
- `mips_bits.s`

Demo: pokemon.c

```
$ gcc pokemon.c print_bits.c -o pokemon
$ ./pokemon
0000010000000000 BUG_TYPE
00000000000010000 POISON_TYPE
1000000000000000 FAIRY_TYPE
1000010000010000 our_pokemon type (1)

Poisonous
1001010000000000 our_pokemon type (2)

Scary
```

Demo: pokemon.c

```
#define FIRE_TYPE      0x0001
#define FIGHTING_TYPE 0x0002
#define WATER_TYPE    0x0004
#define FLYING_TYPE   0x0008
#define POISON_TYPE   0x0010
#define ELECTRIC_TYPE 0x0020
#define GROUND_TYPE   0x0040
#define PSYCHIC_TYPE  0x0080
#define ROCK_TYPE     0x0100
#define ICE_TYPE      0x0200
#define BUG_TYPE      0x0400
#define DRAGON_TYPE   0x0800
#define GHOST_TYPE    0x1000
#define DARK_TYPE     0x2000
#define STEEL_TYPE    0x4000
#define FAIRY_TYPE    0x8000
```

Demo: pokemon.c

```
$ gcc pokemon.c print_bits.c -o pokemon
$ ./pokemon
0000010000000000 BUG_TYPE
00000000000010000 POISON_TYPE
1000000000000000 FAIRY_TYPE
1000010000010000 our_pokemon type (1)

Poisonous
1001010000000000 our_pokemon type (2)

Scary
```

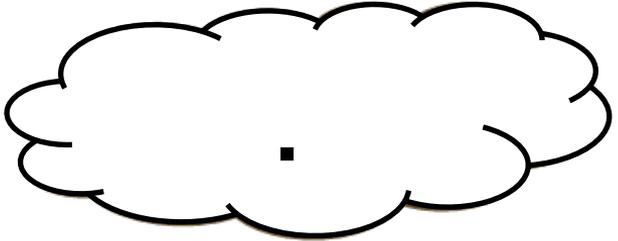
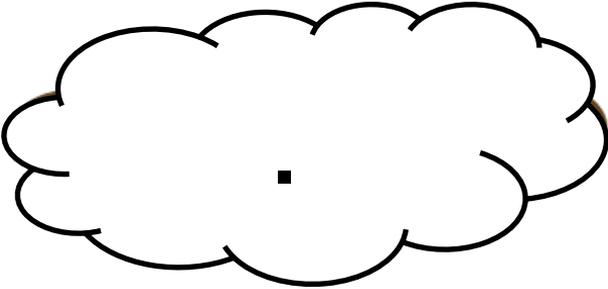

MIPS - Bit manipulation instructions

assembly	meaning	bit pattern
and r_d, r_s, r_t	$r_d = r_s \& r_t$	000000sssstttttddddd00000100100
or r_d, r_s, r_t	$r_d = r_s r_t$	000000sssstttttddddd00000100101
xor r_d, r_s, r_t	$r_d = r_s \wedge r_t$	000000sssstttttddddd00000100110
nor r_d, r_s, r_t	$r_d = \sim(r_s r_t)$	000000sssstttttddddd00000100111
andi r_t, r_s, I	$r_t = r_s \& I$	001100sssstttttIIIIIIIIIIIIIIIIIIII
ori r_t, r_s, I	$r_t = r_s I$	001101sssstttttIIIIIIIIIIIIIIIIIIII
xori r_t, r_s, I	$r_t = r_s \wedge I$	001110sssstttttIIIIIIIIIIIIIIIIIIII
not r_d, r_s	$r_d = \sim r_s$	pseudo-instruction

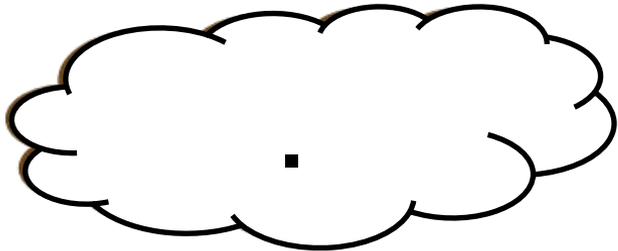
MIPS - Shift instructions

assembly	meaning	bit pattern
sllv r_d, r_t, r_s	$r_d = r_t \ll r_s$	000000s s s s s t t t t t d d d d d 00000000100
srlv r_d, r_t, r_s	$r_d = r_t \gg r_s$	000000s s s s s t t t t t d d d d d 00000000110
srav r_d, r_t, r_s	$r_d = r_t \gg r_s$	000000s s s s s t t t t t d d d d d 00000000111
sll r_d, r_t, I	$r_d = r_t \ll I$	00000000000t t t t t d d d d d I I I I I 000000
srl r_d, r_t, I	$r_d = r_t \gg I$	00000000000t t t t t d d d d d I I I I I 000010
sra r_d, r_t, I	$r_d = r_t \gg I$	00000000000t t t t t d d d d d I I I I I 000011

- **srl** and **srlv** shift zeroes into most-significant bit
 - This matches shift in C of unsigned values
- **sra** and **srav** propagate most-significant bit
 - This ensures the sign is maintained



Floating Point



Fractions in different bases

The decimal fraction 0.75 means

- $7 \cdot 10^{-1} + 5 \cdot 10^{-2} = 0.7 + 0.05 = 0.75$
- or equivalently $75/10^2 = 75/100 = 0.75$

Similarly 0.11_2 means

- $1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 0.5 + 0.25 = 0.75$
- or equivalently $3/2^2 = 3/4 = 0.75$

Similarly $0.C_{16}$ would mean

- $12 \cdot 16^{-1} = 0.75$
- or equivalently $12/16^1 = 3/4 = 0.75$

Note: We call the . a radix point rather than a decimal point when we are dealing with other bases.

Fractions in different bases

2^3	2^2	2^1	2^0
8_{10}	4_{10}	2_{10}	1_{10}

Fractions in different bases

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
8_{10}	4_{10}	2_{10}	1_{10}	0.5_{10}	0.25_{10}	0.125_{10}

Converting fractions to other bases

- The algorithm to convert a decimal fraction to another base is
 - Take the decimal (fractional) part of the number and **multiply** it by the base you are converting to.
 - The whole number part of the result becomes the next digit after the radix point in the converted number.
 - Repeat the process with the remaining fractional part.
 - Continue until the fractional part becomes zero or you have enough digits for the desired accuracy.

Note: This process does not always terminate because some fractions have repeating representations in certain bases.

Example: Converting Fractions

For example if we want to convert 0.3125_{10} to base 2

- $0.3125 * 2 = 0.625$
- $0.625 * 2 = 1.25$
- $0.25 * 2 = 0.5$
- $0.5 * 2 = 1.0$

Therefore $0.3125_{10} = 0.0101_2$

Exercise: Binary Fractional Conversion

Convert the following decimal values into binary

- 12.625
- 0.1

Fixed Point Representation

- A simple trick to represent fractional numbers as integers
 - every value is multiplied by a particular constant and stored as an integer
 - e.g. if constant is 1000 then 56125 represents 56.125
 - we could not represent 3.141592
- Used on small embedded processors without floating point hardware
- Major limitation is range:
 - 16 bits used for integer part and 16 bits for fraction (equivalent to a scale factor of 2^{16})
 - minimum step $2^{-16} \approx 0.000015$
 - maximum $2^{15} \approx 32768$

IEEE 754 Floating Point Representation

- The industry standard
 - Used by almost all computers
- Crucial to understand when working with numeric computations
- Understand precision and accuracy limitations
 - Why using them for finance is unwise
 - Why sometimes
 - $a + 1 == a$
 - Why code like
 - $\text{if } (x == y)$ is not a good idea



Floating Point Numbers

- C has 3 floating point types
 - **float** ... typically 32-bit quantity (lower precision, narrower range)
 - **double** ... typically 64-bit quantity (higher precision, wider range)
 - **long double** ... typically 128-bit quantity (but maybe only 80 bits used)
- Literal floating point values by default are **double**: 3.14159, 1.0/3, 1.0e-9
- Reminder: division of 2 ints gives an int e.g. $1 / 2 == 0$

Code demo: double_output.c

Floating Point Representation Issues

Representing floating point numbers with a fixed number of bits means:

- a finite number of bit patterns
- can only represent a finite subset of reals
 - almost all real values will have no exact representation
 - value of arithmetic operations may have no exact representation
- we must **round** to the **closest value** which can be exactly represented
 - this rounding introduces an error into our calculations
 - often, does not matter
 - sometimes ... can be disastrous
 - eg pacemakers, finance

Code demo: `double_imprecision.c`

Range of Floating Point Types

How do floating types have such a large range?

<code>float</code>	4 bytes	<code>min=1.17549e-38</code>	<code>max=3.40282e+38</code>
<code>double</code>	8 bytes	<code>min=2.22507e-308</code>	<code>max=1.79769e+308</code>

With the same number of bytes compare:

<code>unsigned int</code>	4 bytes	<code>min=0</code>	<code>max= 4294967295 (4.29497e+09)</code>
<code>unsigned long</code>	8 bytes	<code>min=0</code>	<code>max= (1.84467e+19)</code>

Code demo: `floating_types.c`

IEEE Standard: Exponential Representation

Idea: use **scientific notation**

- e.g $6.0221515 * 10^{23}$

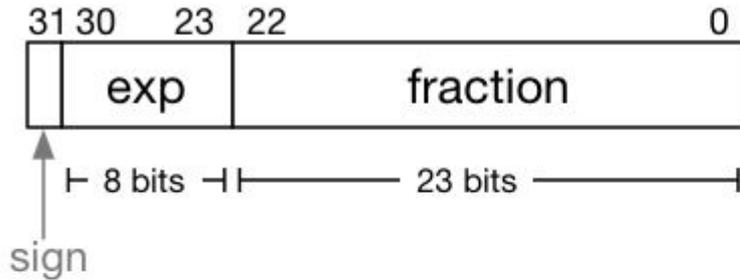
But in binary:

- $10.6875 = 1010.1011$
 $= 1.0101011 * 2^3$

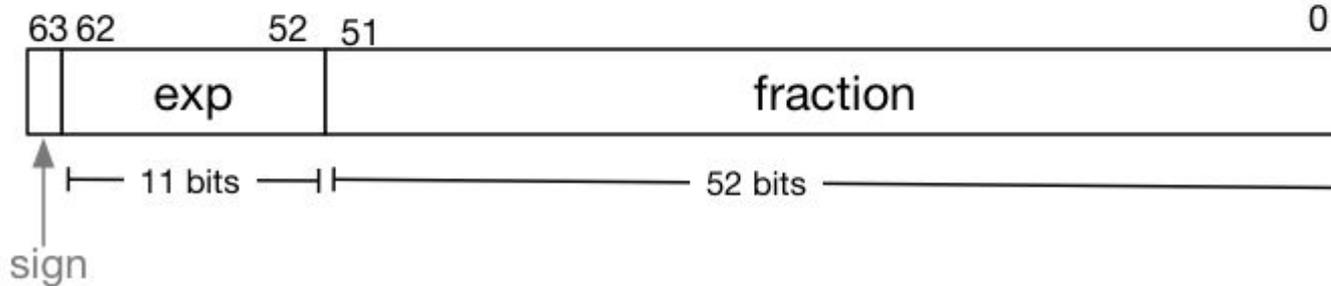
Allows a much bigger range of values to be represented than fixed point

- **8 bits** for the exponent can represent numbers from 10^{-38} .. 10^{38}
- **11 bits** for the exponent can represent numbers from 10^{-308} .. 10^{308}

IEEE 754 Standard



single precision



double precision

Note:
float in C is represented in this single precision format.
double in C is represented in this double precision format

Note: the fraction part is also called the significand or mantissa

IEEE 754 Standard: Sign and Fraction

Sign bit: 0 for positive, 1 for negative

Fraction:

We don't want multiple representations of the same number so we **normalise** it

- Use representation with exactly 1 digit in front of the radix point
 - (i.e. 1.1001×2^3 rather than 1100.1×2^0 or 11.001×2^2)
- Better to have only one representation (one bit pattern) representing a value
 - Multiple representations would make arithmetic slower on CPU

Weird hack: the first bit must be a one we don't need to store it

- As we long we have a special representation for zero
- To represent 1.1001×2^3 we would store **1001000000**... for the fraction.

IEEE 754 Standard: Exponent

- represented relative to a bias value B
 - to represent exponent of x , we would store $x+B$
 - for floats the **bias** is 127
- e.g. we were representing 1.1001×2^3 we would store $(3+127) = 130 = 10000010$ for a float
- How bias is calculated:
 - assume an 8-bit exponent, then bias $B = 2^{8-1} - 1 = 127$
 - 127 is in the middle of the 0-255 range of an 8-bit number
 - valid bit patterns for exponent 00000001 .. 11111110 (1..254)
 - exponent values we can represent -126 .. 127

IEEE 754 Example

150.75 = 10010110.11

// normalise fraction, compute exponent

= 1.001011011 × 2⁷

// determine sign bit,

// map fraction to 24 bits, (don't store the leading 1)

// map exponent to 8 bits after adding on the bias of 127

= 01000011000101101100000000000000

where red is sign bit, green is exponent, blue is fraction

Note: $B=127$, $e=2^7$, so exponent = $127+7 = 134 = 10000110$

Check using explain_float_representation.c or [Floating Point Calculator](#)

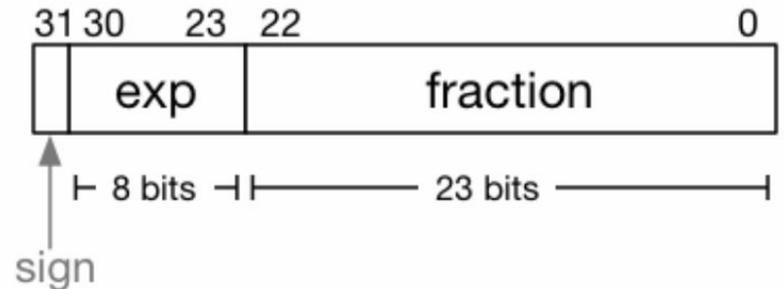
Exercise 2: Floating Point Conversions

Question 1: Convert the decimal number 1 to a floating point number in IEEE 754 single-precision format.

Question 2: Convert the following IEEE 754 single-precision floating point numbers to decimal.

1 10000000 11000000000000000000000000000000

0 01111110 10000000000000000000000000000000



IEEE 754 Standard: Special Cases

Value	Exponent	Fraction	Example
0 (+ve or -ve)	all 0's	all 0's	
inf (∞ and $-\infty$)	all 1's	all 0's	1.0/0
nan	all 1's	Not all 0's	0.0/0

IEEE 754 infinity.c

Representation of +- infinity : propagates sensibly through calculations

```
double x = 1.0/0.0;
printf("%lf\n", x);           //prints inf
printf("%lf\n", -x);         //prints -inf
printf("%lf\n", x - 1);     // prints inf
printf("%lf\n", 2 * atan(x)); // prints 3.141593
printf("%d\n", 42 < x);     // prints 1 (true)
printf("%d\n", x == INFINITY); // prints 1 (true)
```

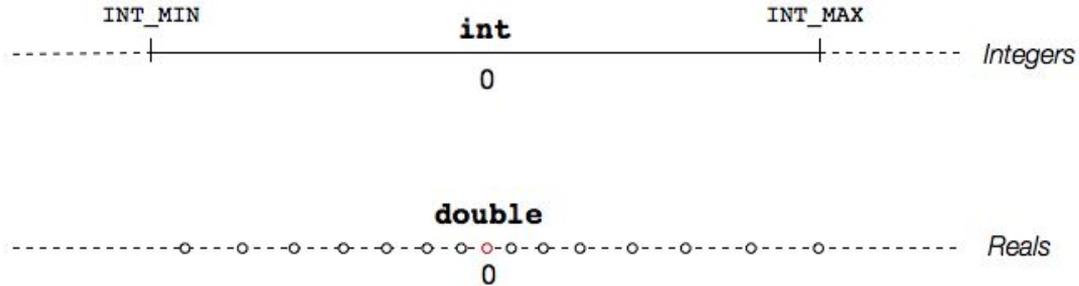
IEEE 754 nan.c

Representation for invalid results NaN (not a number)

- ensures errors propagate sensibly through calculations

```
double x = 0.0/0.0;
printf("%lf\n", x);           //prints nan
printf("%lf\n", x - 1);      // prints nan
printf("%d\n", x == x);      // prints 0 (false)
printf("%d\n", isnan(x));    // prints 1 (true)
```

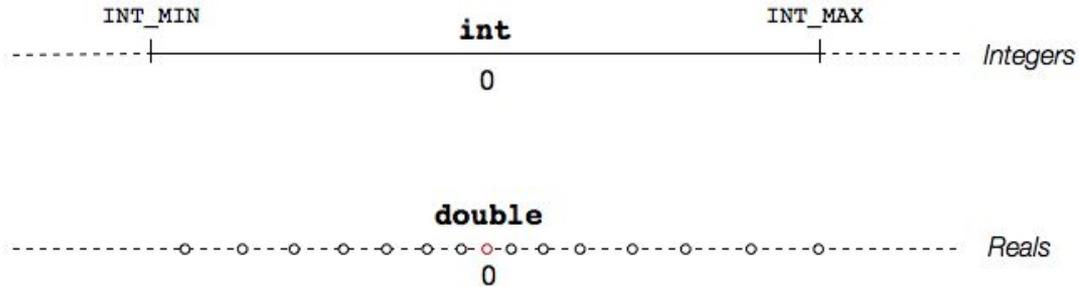
Distribution of Floating Point Numbers



integers ... subset (range) of the mathematical integers

- can represent all integer values in that subset
- each integer is 1 away from the next one and previous one
- all integers are represented accurately

Distribution of Floating Point Numbers



floating point ... subset of the mathematical real numbers

- floating point numbers not evenly distributed
 - numbers closer to 0 have higher precision which is good
 - representations get further apart as values get bigger
 - this works well for most calculations but can cause weird bugs

Distribution of Floating Point Numbers

A 64-bit **double** uses 52 bits for the fraction.

- Between 2^n and 2^{n+1} there are 2^{52} doubles evenly spaced
 - e.g. in the interval 2^{-42} and 2^{-43} there are 2^{52} doubles
 - and in the interval between 0.5 and 1 there are 2^{52} doubles
 - and in the interval between 1 and 2 there are 2^{52} doubles
 - and in the interval between 2 and 4 there are 2^{52} doubles
 - and in the interval between 4 and 8 there are 2^{52} doubles
 - and in the interval between 2^{42} and 2^{43} there are 2^{52} doubles

Distribution of Floating Point Numbers

- This results in doubles
 - near 0.001 - being about 0.00000000000000000002 apart
 - near 1000 - being about 0.00000000000002 apart
 - near 10000000000000000 - being about 0.25 apart
 - **above 2^{53} - doubles are more than 1 apart**

Code Demos

`double_disaster.c`

`explain_float_representation.c`

`double_catastrophe.c` (advanced example)

What did we learn today?

- Bitwise Operators
 - Recap
 - MIPS examples
 - C Coding examples
- Floating Point Representation

Next Lecture: Intro to Operating Systems and File Systems

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/ptY9X4Hg0J>

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



**In Australia Call Afterhours
UNSW Mental Health Support
Line**

1300 787 026
5pm-9am



Mind HUB

student.unsw.edu.au/mind-hub
Online Self-Help Resources



**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