COMP1521 25T1

Week 4 Lecture 2

Bitwise Operators

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

Announcements

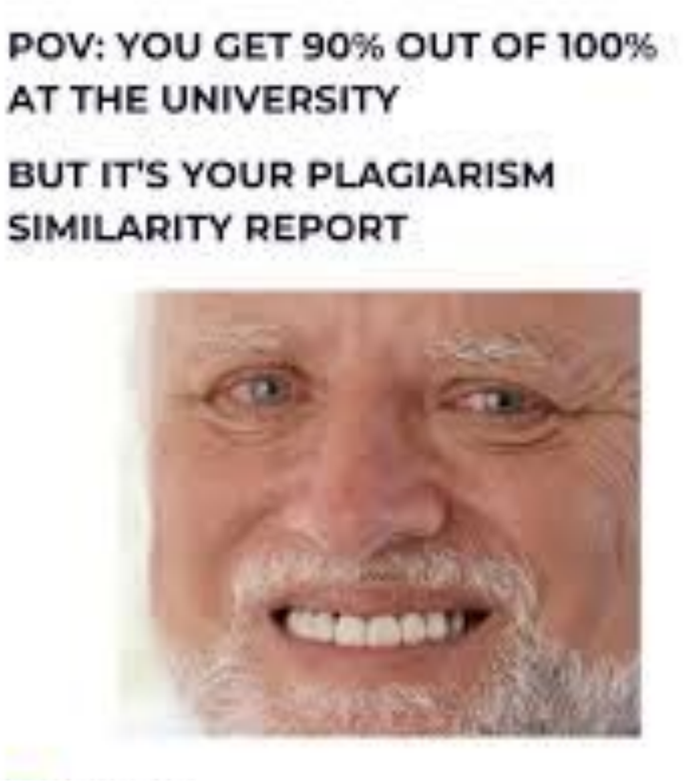
- Week 3 Test Due Tomorrow: Thursday 21:00:00.
  - Many people have not done it yet!
- Census Date : Thursday 13th March
- Assignment 1 Due: Week 5 Friday (next week) at 6pm
- See Help Sessions Schedule

Plagiarism

Get help from the right places

- staff in lectures, tuts, labs
- forum, Help Sessions, Revision Sessions

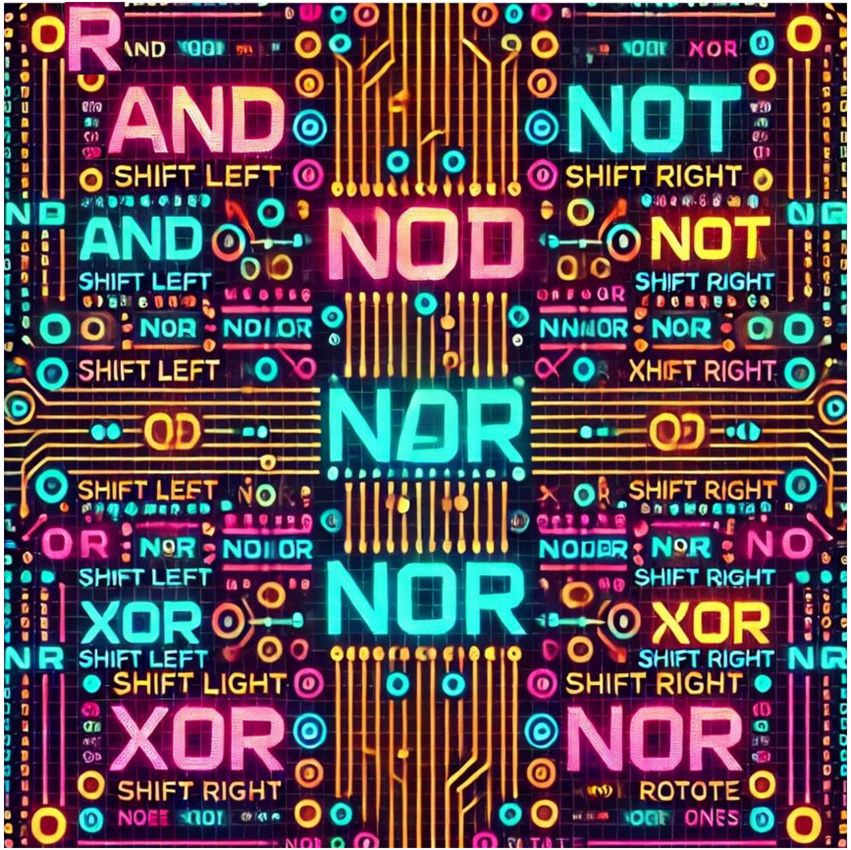
- Do not get ‘help’ or submit code from external sources like:
    - ChatGPT, external tutors, other people’s code etc
- We run plagiarism checking on all submissions



student.unsw.edu.au/plagiarism

Today's Lecture

- Integers Recap Exercises
- End of last lecture
  - Loading in MIPS
  - Endian in C
- Bitwise Operators



Recap: Bit and Bytes

What does this represent?

10110110111110001110110101110110

Recap: Bit and Bytes

What does this represent?

10110110111110001110110101110110

We can’t know without knowing its type!

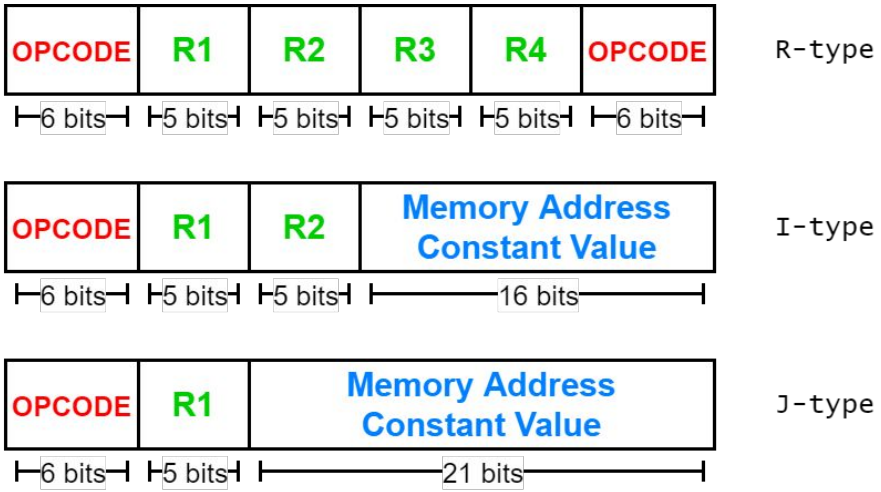
Is it: int, unsigned int, float, unicode character, MIPS instruction?

What MIPS instruction is this?

`0x01288820 =`

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`

What MIPS instruction is this?

`0x01288820 =`

`0000 0001 0010 1000 1000 1000 0010 0000`

What MIPS instruction is this?

`0x01288820 =`

`000000 01001 01000 10001 00000100000`

What MIPS instruction is this?

`0x01288820 =`

`000000 01001 01000 10001 00000100000`

`add`

What MIPS instruction is this?

`0x01288820 =`

`000000 01001 01000 10001 00000100000`

`add $17`

What MIPS instruction is this?

`0x01288820 =`

`000000 01001 01000 10001 00000100000`

`add $17, $9`

What MIPS instruction is this?

```
0x01288820 =

000000 01001 01000 10001 00000100000

add $17, $9, $8
```

What MIPS instruction is this?

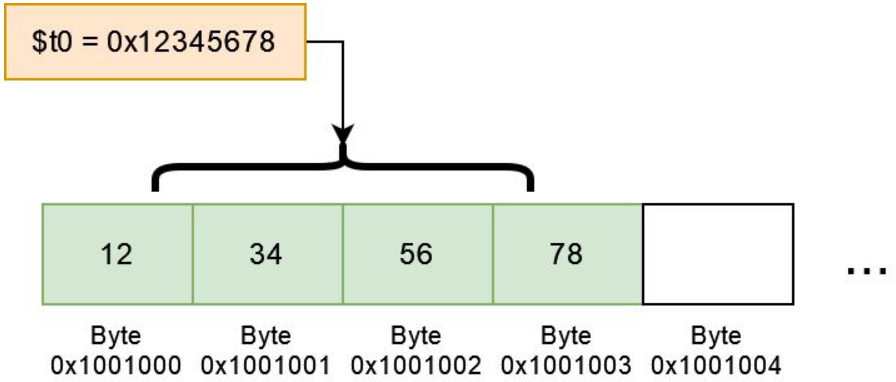
```
0x01288820 =

000000 01001 01000 10001 00000100000

add $17, $9, $8

add $s1, $t1, $t0
```

Let’s type it into mipsy web to check!

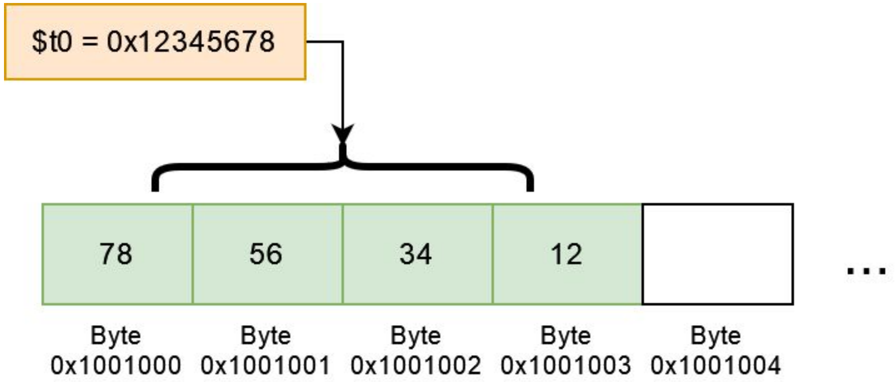
Recap: New concept: Endian-ness

- “What order to put things in” is a hard question to answer
- Two schools of thought:
  - **Big**-endian: MSB at the “low address” - big bits “first!”
  - **Little**-endian: LSB at the “low address” - little bits “first!”

BIG:



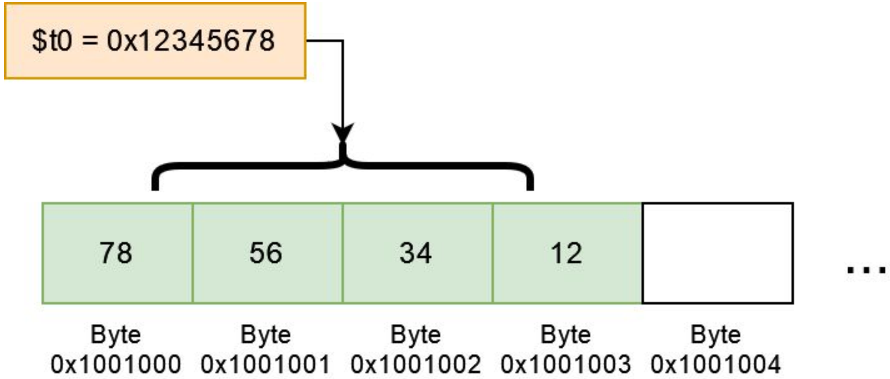
LITTLE:



Code example

- Mipsy-web is **little-endian**

```
        .text
main:
        li $t0, 0x12345678
        sw $t0, my_word

        .data
my_word:
        .space 4
```



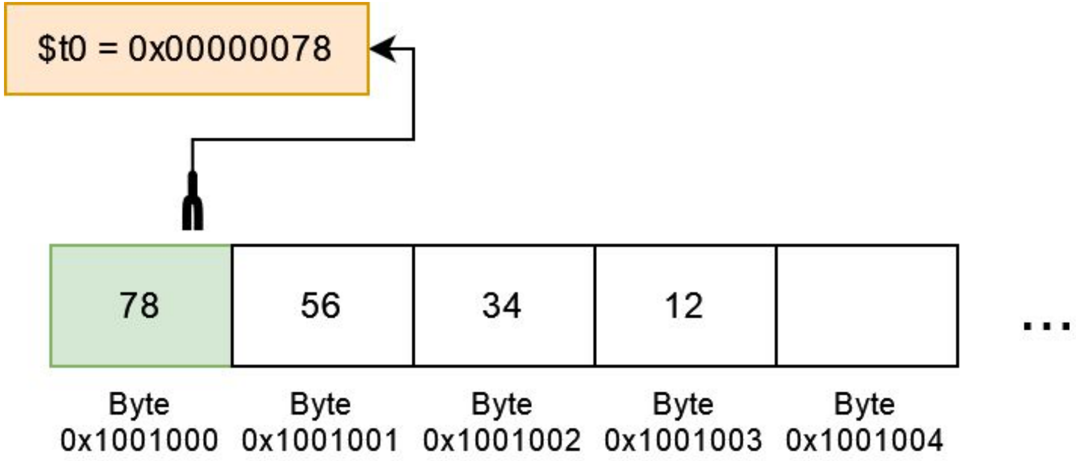
Loading bytes, half-words

The results of these will depend on endianness:

- `lh/lb` assume the loaded byte/halfword is signed
  - The destination register top bits are set to the sign bit
- `lhu/lbu` for doing the same thing, but unsigned

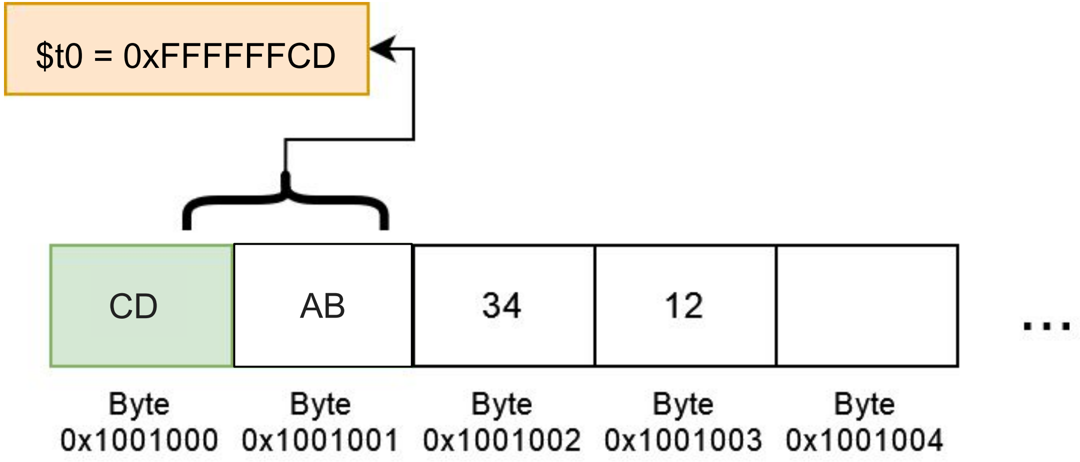
Loading Examples: lb

```
.text
main:
        lb $t0, my_label
.data
my_label:
        .word 0x12345678
```



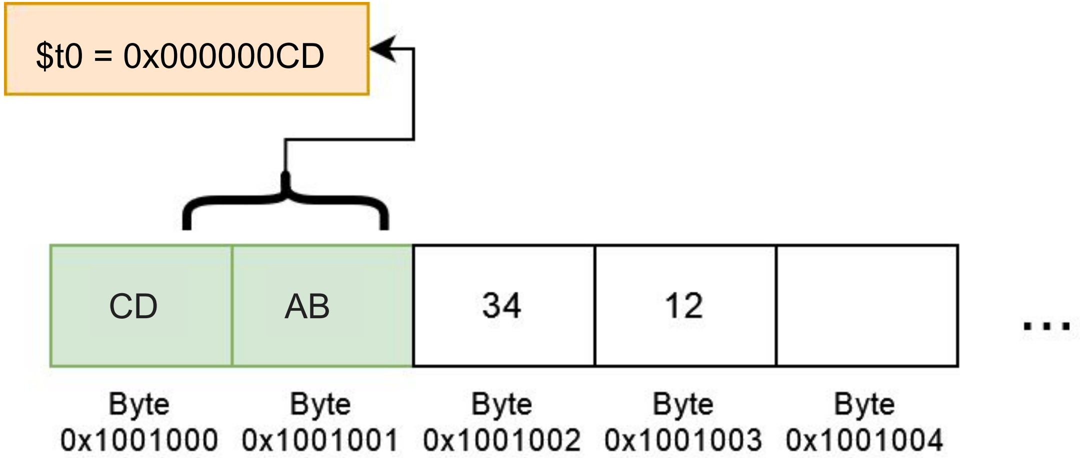
Loading Examples Negative: lb

```
.text
main:
        lb $t0, my_label
.data
my_label:
        .word 0x1234ABCD
```



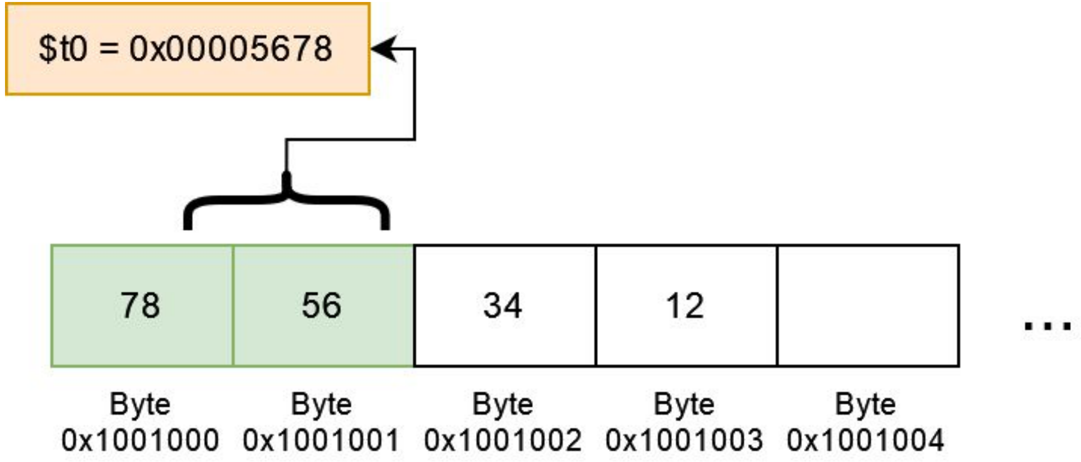
Loading Examples: lbu

```
.text
main:
        lbu $t0, my_label
.data
my_label:
        .word 0x1234ABCD
```



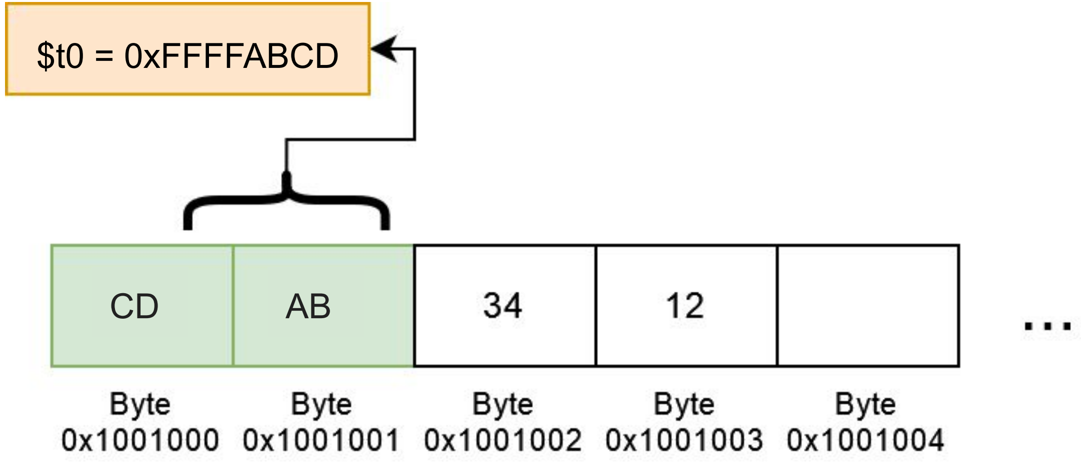
Loading Examples: lh

```
.text
main:
        lh $t0, my_label
.data
my_label:
        .word 0x12345678
```



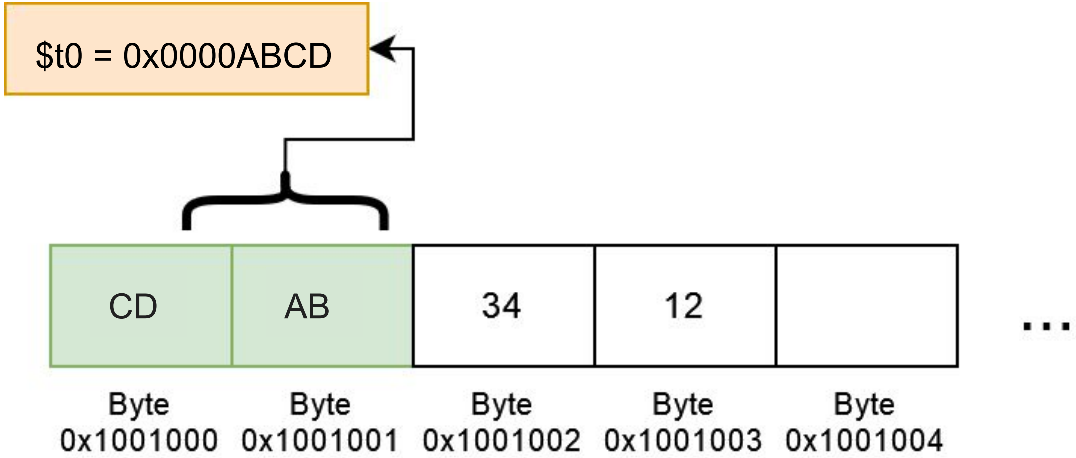
Loading Examples Negative: lh

```
.text
main:
        lh $t0, my_label
.data
my_label:
        .word 0x1234ABCD
```



Loading Examples Negative: lhu

```
.text
main:
        lhu $t0, my_label
.data
my_label:
        .word 0x1234ABCD
```



Endianness in C

endianness.c

Bitwise Operators

Why Learn Bitwise Operators

Used extensively in this course and also:

- Optimisation
- Embedded Systems
- Data compression
- Security and Cryptography
- Graphics
- Computer Networks

Bitwise Operations

- CPUs provide instructions which implement bitwise operations
  - Provide us ways to manipulating the individual bits of a value.
  - MIPS provides 13 bit manipulation instructions
  - C provides 6 bitwise operators

```
&  bitwise AND
|  bitwise OR
^  bitwise XOR (eXclusive OR)
~  bitwise NOT
<< left shift
>> right shift
```

Logical AND (&&) vs Bitwise AND (&)

- && works on whole values
  - We usually use it in conditions like:
    - if (x > 10 && x < 20)
- & works on every individual bit in each value
  - We use it to modify and/or extract bit information from values

Bitwise AND (&)

- takes two values (eg. **a & b**) and performs a logical AND between pairs of corresponding bits
  - resulting bits are set to 1 if **both** the original bits in that column are 1

Example:

	128	64	32	16	8	4	2	1
	0	0	1	0	0	1	1	1
&	1	1	1	0	0	0	1	1
	0	0	1	0	0	0	1	1

&	0	1
0	0	0
1	0	1

Used for eg. checking if particular bits are set (that is, set to 1) or unsetting bits (that is, setting them to 0)

Exercise: &

For any given bit value, x what is:

x & 0 = ?

x & 1 = ?

Exercise: &

For any given bit value, x what is:

x & 0 = 0

x & 1 = x

Bit Masks

We can create bit patterns to help us isolate the bits we are interested in! We call these masks!

For example:

```
int8_t x = 0x13;                //00010011
int8_t mask = 0x7;              //00000111  &
int8_t result = x & mask;
```

Bit Masks

We can create bit patterns to help us isolate the bits we are interested in! We call these masks!

For example:

```
int8_t x = 0x13;               //00010011
int8_t mask = 0x7;             //00000111  &
int8_t result = x & mask;
```

Bit Masks

We can create bit patterns to help us isolate the bits we are interested in! We call these masks!

For example:

```
int8_t x = 0x13;                //00010011
int8_t mask = 0x7;              //00000111  &
int8_t result = x & mask;       //00000011
```

bit_ops_and.c

Checking if a number is odd

The obvious way to check if a number is odd in C:

```
int is_odd(int n) {
    return n % 2 != 0;
}
```

Checking if a number is odd

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

What pattern do you see in the binary representation of odd numbers?

Checking if a number is odd

Decimal	Binary
0	000
1	00**1**
2	010
3	01**1**
4	100
5	10**1**
6	110
7	11**1**

What pattern do you see in the binary representation of odd numbers?

They all have a 1 as the least significant bit.

We can check that bit to see if it is 1.
If it is it is odd!

Checking if a number is odd

```c
int is_odd(int n) {
    return n & 1;
}
```

If the value is **ODD** (eg 39):

	128	64	32	16	8	4	2	1
	0	0	1	0	0	1	1	1
&	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	1

If the value is **EVEN** (eg 38):

	128	64	32	16	8	4	2	1
	0	0	1	0	0	1	1	0
&	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	0

Bitwise OR (|)

- takes two values (eg. **a | b**) and performs a logical OR between pairs of corresponding bits
    - resulting bits are set to 1 if **at least** one of the original bits are 1

Example:

	0	0	1	0	0	1	1	1
|	1	1	1	0	0	0	1	1
	1	1	1	0	0	1	1	1

|	0	1
0	0	1
1	1	1

Used for eg. setting particular bits (ie set to 1)

Bit Masks with |

For any given bit value, x what is:

x | 0 = ?

x | 1 = ?

Bit Masks with |

For any given bit value, x what is:

x | 0 = x

x | 1 = 1

Bit Masks with |

For any given bit value, x what is:

x | 0 = x

x | 1 = 1

For example:

```
int8_t x = 0x13;             //00010011
int8_t mask = 0x7;           //00000111
int8_t result = x | mask;    //00010111
```

Bitwise Negation (~)

- takes a single value (eg. ~**a**) and performs a logical negation on each bit

Example:

~	0	0	0	1	0	1	1	0
	1	1	1	0	1	0	0	1

~	0	1
	1	0

Note: This does NOT mean making a number negative!

Bit Masks with ~

We can have a mask we can use to both set bits and unset bits

- Example:
  - mask 0x7 with | to set the least significant 3 bits
  - negate that mask and use it with & to unset the least significant 3 bits

Bit Masks with ~

We can have a mask we can use to both set bits and unset bits

- Example:
    - mask 0x7 with | to set the least significant 3 bits
    - negate that mask and use it with & to unset the least significant 3 bits

For example:

```
int8_t x = 0x13;                //00010011
int8_t mask = ~0x7;             //11111000
int8_t result = x & mask;       //00010000
```

Bitwise XOR (^)

- takes two values (eg. **a ^ b**) and performs an exclusive OR between pairs of corresponding bits
  - resulting bits are set to 1 if **exactly** one of the original bits are 1

Example:

	0	0	1	0	0	1	1	1
^	1	1	1	0	0	0	1	1
	1	1	0	0	0	1	0	1

^	0	1
0	0	1
1	1	0

Used for eg. cryptography, flipping a bit

Bit Masks with ^

For any given bit value, x what is:

x ^ 0 = x

x ^ 1 = ~x (flips the bit)

Bit Masks with ^

For any given bit value, x what is:

x ^ 0 = x

x ^ 1 = ~x (flips the bit)

For example:

```
int8_t x = 0x13;           //00010011
int8_t mask = 0x7;         //00000111
int8_t result = x & mask;  //00010100
```

Bit Masks with ^

For any given bit value, x what is:

x ^ 0 = x

x ^ 1 = ~x (flips the bit)

For example:

```
int8_t x = 0x13;            //00010011
int8_t mask = 0x7;          //00000111
int8_t result = x & mask;   //00010100
```

What happens if I apply the mask again?

Exercise 1:

- Evaluate the following:
  - 5 && 6
  - 5 & 6
- How many beers did the software developer drink?

Left Shift (<<)

- takes a value and a small positive integer *x* (eg. `a << x`)
- shifts each bit *x* positions to the left
  - any bits that fall off the left vanish
  - new 0 bits are inserted on the right
  - result contains the same number of bits as the input
- Example:

```
  1 1 1 0 0 0 1 1 << 2
  ---------------
  1 0 0 0 1 1 0 0
```

Implications of left shift

What does this mean mathematically?

Implications of left shift

What does this mean mathematically?

Expression	Result Binary	Result Decimal
00000001 << 1	00000010	2
00000001 << 2	00000100	4
00000001 << 3	00001000	8
00000001 << 4	00010000	16

Implications of left shift

What does this mean mathematically? Multiplies by powers of 2!

Expression	**Result Binary**	**Result Decimal**
00000001 << 1	00000010	2
00000001 << 2	00000100	4
00000001 << 3	00001000	8
00000001 << 4	00010000	16

Demo: shift_as_multiply.c

Right Shift (>>)

- takes a value and a small positive integer *x* (eg. `a >> x`)
- shifts each bit *x* positions to the right
  - any bits that fall off the right vanish
  - new 0 bits are inserted on the left (for unsigned types)
  - result contains the same number of bits as the input
- Example:

```
1 1 1 0 0 0 1 1 >> 2
---------------
0 0 1 1 1 0 0 0
```

Used for eg looping through 1 bit at a time

Implications of right shift

What does this mean mathematically?

Implications of right shift

What does this mean mathematically? $16_{10} == 00010000_2$

Expression	Result Binary	Result Decimal
00010000 >> 1	00001000	8
00010000 >> 2	00000100	4
00010000 >> 3	00000010	2
00010000 >> 4	00000001	1

Implications of right shift

What does this mean mathematically? 16_{10} == 00010000_2

Expression	Result Binary	Result Decimal
00010000 >> 1	00001000	8
00010000 >> 2	00000100	4
00010000 >> 3	00000010	2
00010000 >> 4	00000001	1

Divides by powers of 2

Implications of right shift

But what about situations like this? We lose some bits!

0111 >> 1   == 0011

This is the same as 7/2 == 3 with integer division!

Issues with shifting

- Shifts involving negative values may not be portable, and can vary across different implementations
- Common source of bugs in COMP1521 (and elsewhere)
- Always use unsigned values/variables when shifting to be safe/portable

Demo: shift_bug.c

Code Demos

- get_nth_bit.c
- xor.c
- pokemon.c
- set_low_bits0.c
- set_low_bits.c

Demo: pokemon.c

```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
```

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 << 1;
uint8_t d = y << 2;
```

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

MIPS - Bit manipulation instructions

assembly	meaning	bit pattern
and r_d, r_s, r_t	$r_d = r_s \& r_t$	000000ssssstttttddddd00000100100
or r_d, r_s, r_t	$r_d = r_s \mid r_t$	000000ssssstttttddddd00000100101
xor r_d, r_s, r_t	$r_d = r_s \wedge r_t$	000000ssssstttttddddd00000100110
nor r_d, r_s, r_t	$r_d = \sim(r_s \mid r_t)$	000000ssssstttttddddd00000100111
andi r_t, r_s, I	$r_t = r_s \& \mathrm{I}$	001100ssssstttttIIIIIIIIIIIIIIII
ori r_t, r_s, I	$r_t = r_s \mid \mathrm{I}$	001101ssssstttttIIIIIIIIIIIIIIII
xori r_t, r_s, I	$r_t = r_s \wedge \mathrm{I}$	001110ssssstttttIIIIIIIIIIIIIIII
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	$r_d = r_t \ll r_s$	000000ssssstttttddddd00000000100
srlv r_d, r_t, r_s	$r_d = r_t \gg r_s$	000000ssssstttttddddd00000000110
srav r_d, r_t, r_s	$r_d = r_t \gg r_s$	000000ssssstttttddddd00000000111
sll r_d, r_t, I	$r_d = r_t \ll$ I	00000000000tttttdddddIIIII000000
srl r_d, r_t, I	$r_d = r_t \gg$ I	00000000000tttttdddddIIIII000010
sra r_d, r_t, I	$r_d = r_t \gg$ I	00000000000tttttdddddIIIII000011

1

- **`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

MIPS Code Demos

- odd_even.s
- mips_bits.s
- mips_negative_shifts.s

What did we learn today?

- Integer representation recap
- Bitwise Operators
- Next lecture:
    - Floating Point Data

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/NKVTwTXixC

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**