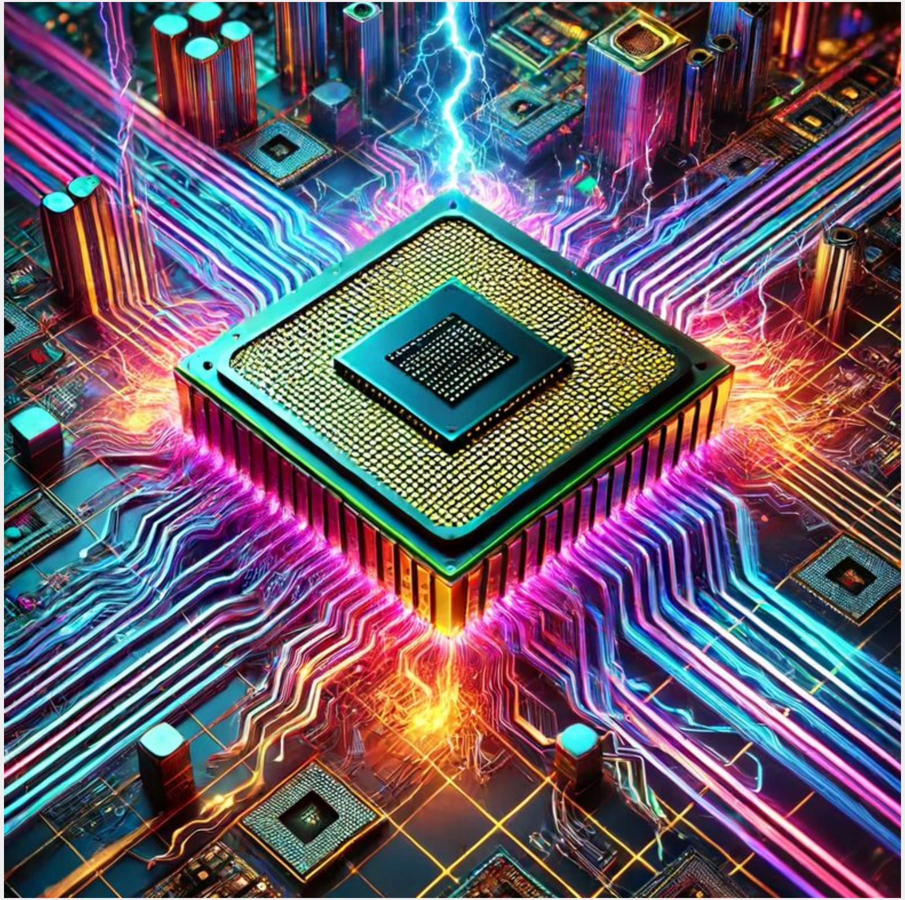
COMP1521 24T2 Lec06 part A

MIPS: Recap

2024

Hammond Pearce

Adapted from *Nothing*



Recap: MIPS Function skeleton

```
func:
        # [header comment]
func__prologue:
        begin
        push    $ra
        push    $s0
        push    $s1

func__body:
        # do stuff

        li      $a0, 42
        jal     foo         # foo(42)

        # foo return val in $v0

func__epilogue:
        pop     $s1
        pop     $s0
        pop     $ra
        end

        jr      $ra
```

Recap: A translation exercise

```
#include <stdio.h>

struct pizza_t {
    char size[10];
    int price_cents;
};

struct pizza_t pizza_options[3] = {
    {"small", 300},
    {"medium", 550},
    {"large", 800}
};
```

This should use everything from the prior lectures!

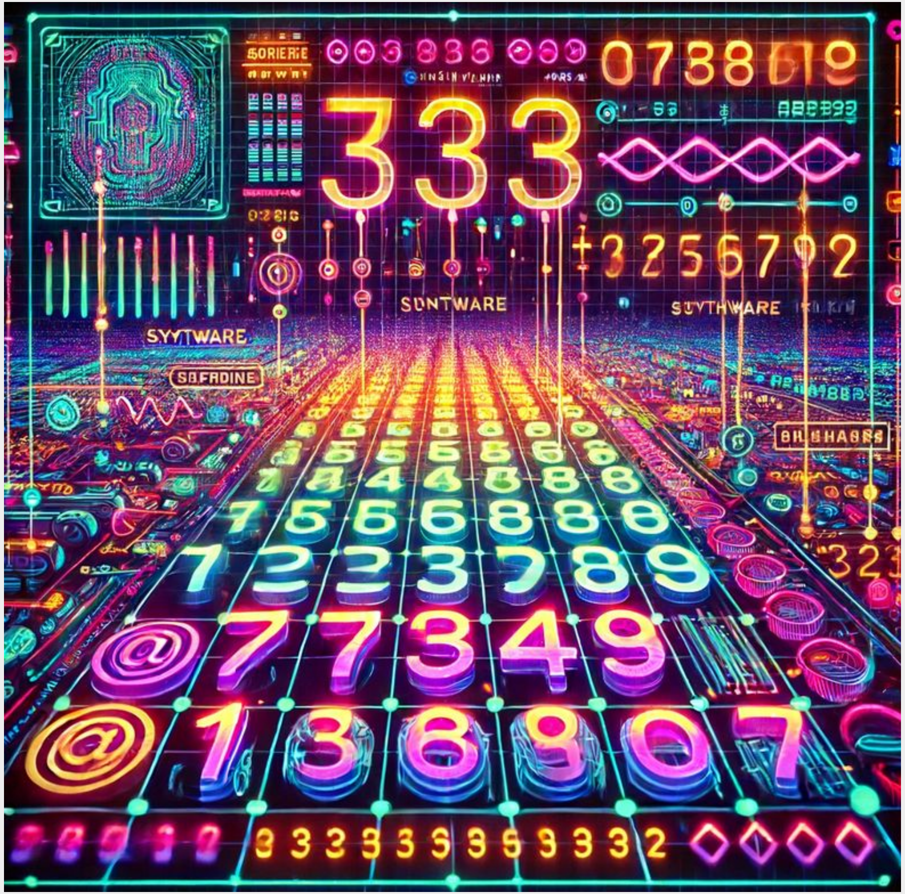
```
void print_pizza_t(struct pizza_t *pizza) {
    printf("Size: %s, ", pizza->size);
    printf("price: %d cents\n", pizza->price_cents);
}

void increase_price(struct pizza_t *pizza, int increase_cents) {
    pizza->price_cents += increase_cents;
}

int main() {
    printf("The available pizza options are:\n");
    for (int i = 0; i < 3; i++) {
        increase_price(&pizza_options[i], 100);
        print_pizza_t(&pizza_options[i]);
    }
    return 0;
}
```

COMP1521 24T2 Lec06 part B

Integers



2024
Hammond Pearce
Adapted from Andrew's Material

There are 10 types of students

There are 10 types of students

Those that understand binary,

There are 10 types of students

Those that understand binary,

And those that don’t

There are 10 types of students

Those that understand binary,

And those that don’t

-Andrew Taylor

What’s a number?

4750

What’s a number?

4750 - this is a number

What’s a number?

4750 - this is a number

It is equivalent to:

4000 + 700 + 50 + 0

What’s a number?

4750 - this is a number

It is equivalent to:

4000 + 700 + 50 + 0

If we assume it is base 10!

What’s a number?

4750 - this is a number

It is equivalent to:

4000 + 700 + 50 + 0

We can also write this as,

$$\mathbf{4*10^3 + 7*10^2 + 5*10^1 + 0*10^0}$$

What's a number?

4750 - this is a number

It is equivalent to:

4000 + 700 + 50 + 0

We can also write this as,

$$\mathbf{4*10^3 + 7*10^2 + 5*10^1 + 0*10^0 = 4750_{10}}$$

Base 10 is an arbitrary choice

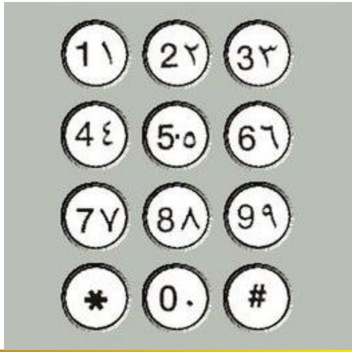
- Base 10 is also called “Decimal” (Deci = 10)

Base 10 is an arbitrary choice

- Base 10 is also called “Decimal” (Deci = 10)
- Possibly exists because we have 10 digits (fingers)
- Ancient Egyptians, Brahmi Numerals, Greek Numerals, Hebrew Numerals, Roman Numerals and Chinese Numerals:
  - All base 10!

(Aside)

- It was quite hard to do math in a lot of the base 10 notations
    - XVI times IIV plus L ?????
- The Hindu–Arabic numeral system simplified things greatly
- Popularized by **Al-Khwarizmi**
    - This guy wrote some neat books, including one called Al-Jabr, in the 800s
        - You might know some of his work!



What about some other bases?

What about some other bases?

- Let’s make a new number system which is base 7
- Here, $1216_7 = ?_{10}$

What about some other bases?

- Let’s make a new number system which is base 7
- Here, $1216_7 = ?_{10}$

$1 * 7^3 + 2 * 7^2 + 1 * 7^1 + 6 * 7^0 = ?$

What about some other bases?

- Let’s make a new number system which is base 7
- Here, $1216_7 = ?_{10}$

$$1 * 7^3 + 2 * 7^2 + 1 * 7^1 + 6 * 7^0 = ?$$

7^3	7^2	7^1	7^0
343_{10}	49_{10}	7_{10}	1_{10}

What about some other bases?

- Let’s make a new number system which is base 7
- Here, $1216_7 = ?_{10}$

$1 * 7^3 + 2 * 7^2 + 1 * 7^1 + 6 * 7^0 = ?$

7^3	7^2	7^1	7^0
343_{10}	49_{10}	7_{10}	1_{10}

$1 * 343 + 2 * 47 + 2 * 7 + 6 * 1 = ?$

What about some other bases?

- Let’s make a new number system which is base 7
- Here, $1216_7 = ?_{10}$

$1 * 7^3 + 2 * 7^2 + 1 * 7^1 + 6 * 7^0 = ?$

7^3	7^2	7^1	7^0
343_{10}	49_{10}	7_{10}	1_{10}

$1 * 343 + 2 * 47 + 2 * 7 + 6 * 1 = \mathbf{454_{10}}$

Computers like binary

- Binary is decimal using base 2 notation
  - Digits 0 and 1
  - Easy to represent using "electricity"

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

Computers like binary

- Binary is decimal using base 2 notation
  - Digits 0 and 1
  - Easy to represent using "electricity"

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

Question: What is 1011_2 in base 10?

Computers like binary

- Binary is decimal using base 2 notation
  - Digits 0 and 1
  - Easy to represent using "electricity"

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

Question: What is 1011_2 in base 10?

Answer: 1 * 8 + 0 * 4 + 1 * 2 + 1 * 1 = 11

More examples

Question: Convert 1101_2 to decimal?

More examples

Question: Convert 1101_2 to decimal?

Answer: 13

More examples

Question: Convert 1101_2 to decimal?

Answer: 13

Question: Convert 29_{10} to binary?

More examples

Question: Convert 1101_2 to decimal?

Answer: 13

Question: Convert 29_{10} to binary?

Answer: 11101

Binary numbers are hard to read!

- They get very long, very fast
- E.g. $12345678_{10} = 101111000110000101001110_2$

Binary numbers are hard to read!

- They get very long, very fast
- E.g. $12345678_{10} = 101111000110000101001110_2$
- Solution: Write numbers in hexadecimal!
- The *base* or *radix* is “16”, digits 0 1 2 3 4 5 6 7 8 9 A B C D E F

Binary numbers are hard to read!

- They get very long, very fast
- E.g. $12345678_{10} = 101111000110000101001110_2$
- Solution: Write numbers in hexadecimal!
- The *base* or *radix* is “16”, digits 0 1 2 3 4 5 6 7 8 9 A B C D E F

16^3	16^2	16^1	16^0
4096_{10}	256_{10}	16_{10}	1_{10}

Binary numbers are hard to read!

- They get very long, very fast
- E.g. $12345678_{10} = 101111000110000101001110_2$
- Solution: Write numbers in hexadecimal!
- The *base* or *radix* is “16”, digits 0 1 2 3 4 5 6 7 8 9 A B C D E F

16^3	16^2	16^1	16^0
4096_{10}	256_{10}	16_{10}	1_{10}

Example: $3AF1_{16} = ?_{10}$

Hexadecimal

- The *base* or *radix* is “16”, digits 0 1 2 3 4 5 6 7 8 9 A B C D E F

16^3	16^2	16^1	16^0
4096_{10}	256_{10}	16_{10}	1_{10}

Example: $3AF1_{16} = ?_{10}$

$3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0 = ?$

Hexadecimal

- The *base* or *radix* is “16”, digits 0 1 2 3 4 5 6 7 8 9 A B C D E F

16^3	16^2	16^1	16^0
4096_{10}	256_{10}	16_{10}	1_{10}

Example: $3AF1_{16} = ?_{10}$

$3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0 = 15089_{10}$

More hexadecimal examples

Question: Convert $1FF_{16}$ to decimal?

More hexadecimal examples

Question: Convert $1FF_{16}$ to decimal?

Answer: 511_{10}

More hexadecimal examples

Question: Convert $1FF_{16}$ to decimal?

Answer: 511_{10}

Question: Convert 11101_2 to hexadecimal?

Answer: $1D_{16}$

Binary -> Hexadecimal

- Binary is long e.g. $12345678_{10} = 101111000110000101001110_2$
- Solution: Write numbers in hexadecimal!

16^3	16^2	16^1	16^0
4096_{10}	256_{10}	16_{10}	1_{10}

How do we do this?

Binary -> Hexadecimal

- Binary is long e.g. $12345678_{10} = 101111000110000101001110_2$
- Solution: Write numbers in hexadecimal!

16^3	16^2	16^1	16^0
4096_{10}	256_{10}	16_{10}	1_{10}

“16” is a power of “2” – 4^2

Binary -> Hexadecimal

- Binary is long e.g. $12345678_{10} = 101111000110000101001110_2$
- Solution: Write numbers in hexadecimal!

16^3	16^2	16^1	16^0
4096_{10}	256_{10}	16_{10}	1_{10}

“16” is a power of “2” – 4^2

Separate the bits into groups of 4…

Binary -> Hexadecimal

- $12345678_{10} = 101111000110000101001110_2$
- $= 1011\ 1100\ 0110\ 0001\ 0100\ 1110_2$

Binary -> Hexadecimal

- $12345678_{10} = 101111000110000101001110_2$
- $= 1011\ 1100\ 0110\ 0001\ 0100\ 1110_2$
- Each 4 bit group now exactly fits **one** hexadecimal numeral!

Binary -> Hexadecimal

- 12345678_{10} = $101111000110000101001110_2$
- = $1011\ 1100\ 0110\ 0001\ 0100\ 1110_2$
- Each 4 bit group now exactly fits **one** hexadecimal numeral!

Base 10	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Base 16	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Base 2																0000

Binary -> Hexadecimal

- 12345678_{10} = $101111000110000101001110_2$
- = $1011\ 1100\ 0110\ 0001\ 0100\ 1110_2$
- Each 4 bit group now exactly fits **one** hexadecimal numeral!

Base 10	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Base 16	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Base 2	1111	1110	1101	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001	0000

Binary -> Hexadecimal

- 12345678_{10} = $101111000110000101001110_2$
- = $1011\ 1100\ 0110\ 0001\ 0100\ 1110_2$
- Each 4 bit group now exactly fits **one** hexadecimal numeral!

Base 10	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Base 16	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Base 2	1111	1110	1101	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001	0000

Simply “look up” the value!

Binary -> Hexadecimal

- 12345678_{10} = $101111000110000101001110_2$
- = $1011\ 1100\ 0110\ 0001\ 0100\ 1110_2$
- Each 4 bit group now exactly fits **one** hexadecimal numeral!

Base 10	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Base 16	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Base 2	1111	1110	1101	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001	0000

- = $1011\ 1100\ 0110\ 0001\ 0100\ 1110_2$
- = B C 6 1 4 E_{16}

More examples

Binary	01101111
=	
Hexadecimal	6F

More examples

Binary 01101111

=

Hexadecimal 6F

Hexadecimal A5

=

Decimal 10100101

Your turn

Question: Convert $1FF_{16}$ to binary?

Question: Convert 001111101_2 to hexadecimal?

Your turn

Question: Convert $1FF_{16}$ to binary?

Answer: $0001\ 1111\ 1111_2$

Question: Convert 001111101_2 to hexadecimal?

Answer: $3D_{16}$

Are there other powers of 2?

- Yes, octal (Base 3)
  - Used to be popular for binary numbers
  - Similar advantages to hexadecimal
- Group bits into 3:

Base 10	7	6	5	4	3	2	1	0
Base 8	7	6	5	4	3	2	1	0
Base 2	111	110	101	100	011	010	001	000

- Example: $72_8 = 111\ 010_2$

Are there other powers of 2?

- Yes, octal (Base 3)
  - Used to be popular for binary numbers
  - Similar advantages to hexadecimal
- Group bits into 3:

Base 10	7	6	5	4	3	2	1	0
Base 8	7	6	5	4	3	2	1	0
Base 2	111	110	101	100	011	010	001	000

- Example: $72_8 = 111\ 010_2 = 3A_{16} = 58_{10}$

A handy way to remember:

- In binary, each digit is 1 bit:
  - $010010001111101010111110010010111_2$

Remember:

- In binary, each digit represents 1 bit:
    - $01001000111110101011110010010111_2$
- In hexadecimal, each digit represents 4 bits:
    - $0100\ 1000\ 1111\ 1010\ 1011\ 1100\ 1001\ 0111_2$
    - $4\ \ 8\ \ F\ \ A\ \ B\ \ C\ \ 9\ \ 7_{16}$
- In octal, each digit represents 3 bits
    - $01\ 001\ 000\ 111\ 110\ 101\ 011\ 110\ 010\ 010\ 111_2$
    - $1\ \ 1\ \ 0\ \ 7\ \ 6\ \ 5\ \ 3\ \ 6\ \ 2\ \ 2\ \ 7_8$

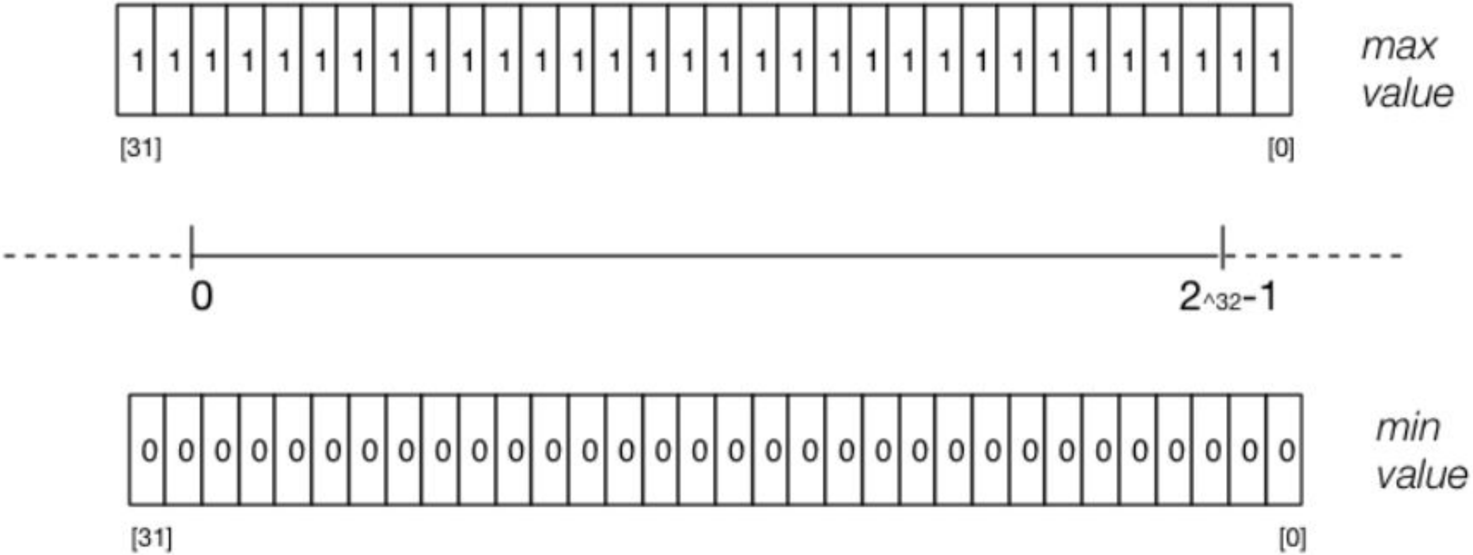
Constants in C and MIPS assembly

- A number beginning with 0x is hexadecimal
- A number beginning with 0 is octal
- A number beginning with 0b is binary
- Otherwise, it is decimal

```
printf("%d", 0x2A);     // prints 42
printf("%d", 052);      // prints 42
printf("%d", 0b101010); // prints 42
printf("%d", 42);       // prints 42
```

How big can numbers get?

- In MIPSY integers are “words”, which are 4 bytes == 32 bits
  - means we can store values from the range 0 .. 2^{32}-1

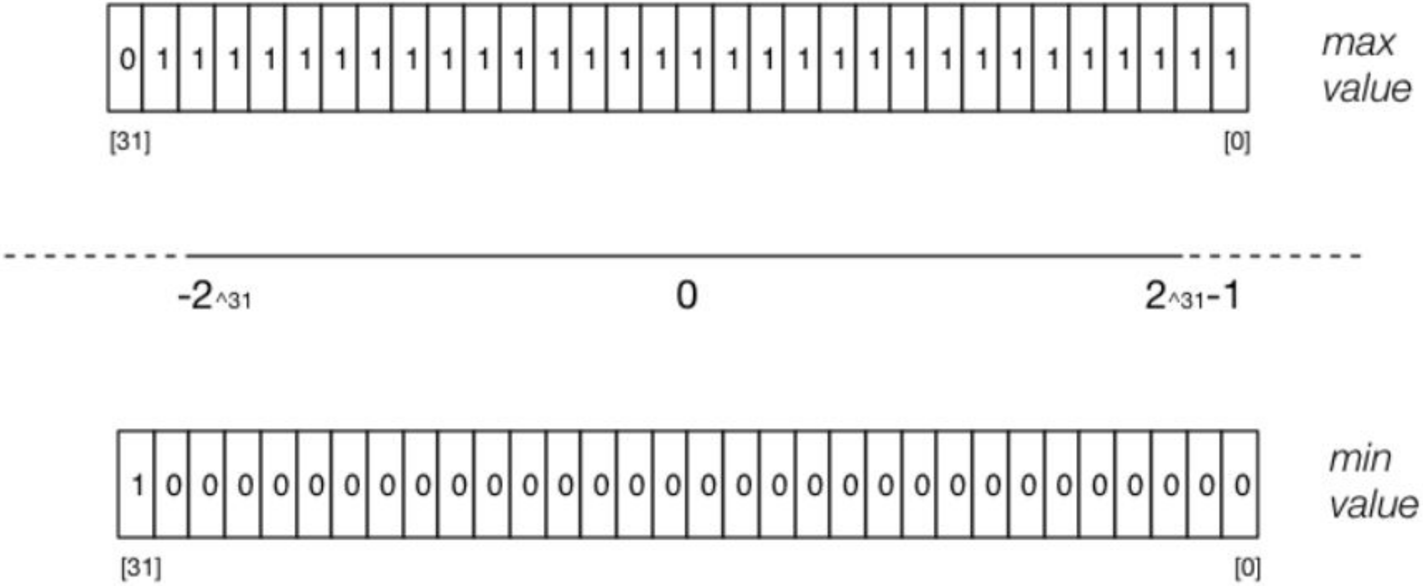


What about negative numbers?

- We call these “signed” numbers
- E.g. -6 has a “negative” sign, +6 has a “positive” sign

Range of negative values

- The maximum number of values stays the same
- But the range of values shift to $-2^{31} .. 2^{31}-1$



What do signed binary numbers look like?

- Modern computers use “two’s complement” for integers
- Positive integers and zero represented as normal,
- Negative integers represented in a way to make math ✨easy✨
  - For an *n*-bit binary number, the number $-b$ is $2^n - b$
  - E.g. 8-bit number “-5” is represented as $2^8 - 5 = 1111\ 1011_2$

Example

- Some simple code to examine 8-bit 2’s complement numbers:

```
for (int i = -128; i < 128; i++) {
    printf("%4d ", i);
    print_bits(i, 8);
    printf("\n");
}
```

- `gcc 8_bit_twos_complement.c print_bits.c -o 8_bit_twos_complement`

Example: Printing all 8-bit 2’s complement

```
$ ./8_bit_twos_complement
-128 10000000
-127 10000001
-126 10000010
...
-3 11111101
-2 11111110
-1 11111111
0 00000000
1 00000001
2 00000010
3 00000011
...
125 01111101
126 01111110
127 01111111
```

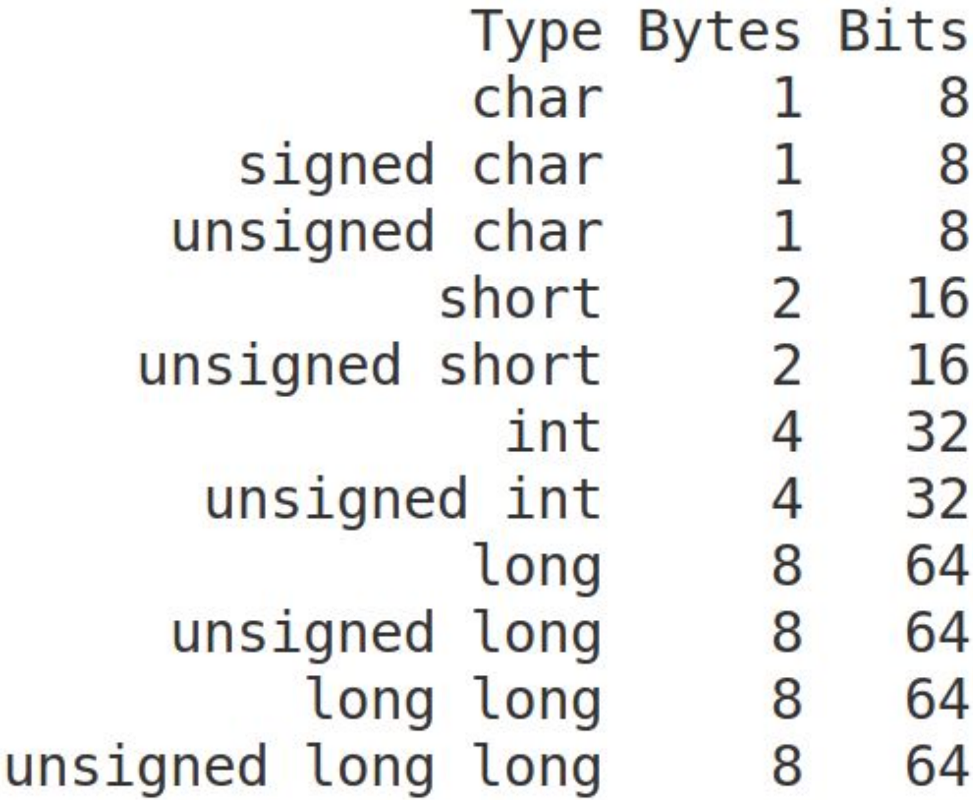
Example: print_bits_of_int.c

```
$ ./print_bits_of_int
Enter an int: 0
00000000000000000000000000000000
$ ./print_bits_of_int
Enter an int: 1
00000000000000000000000000000001
$ ./print_bits_of_int
Enter an int: -1
11111111111111111111111111111111
$ ./print_bits_of_int
Enter an int: 2147483647
01111111111111111111111111111111
$ ./print_bits_of_int
Enter an int: -2147483648
10000000000000000000000000000000
$
```

HW usually based on bytes, 1 byte = 8 bits

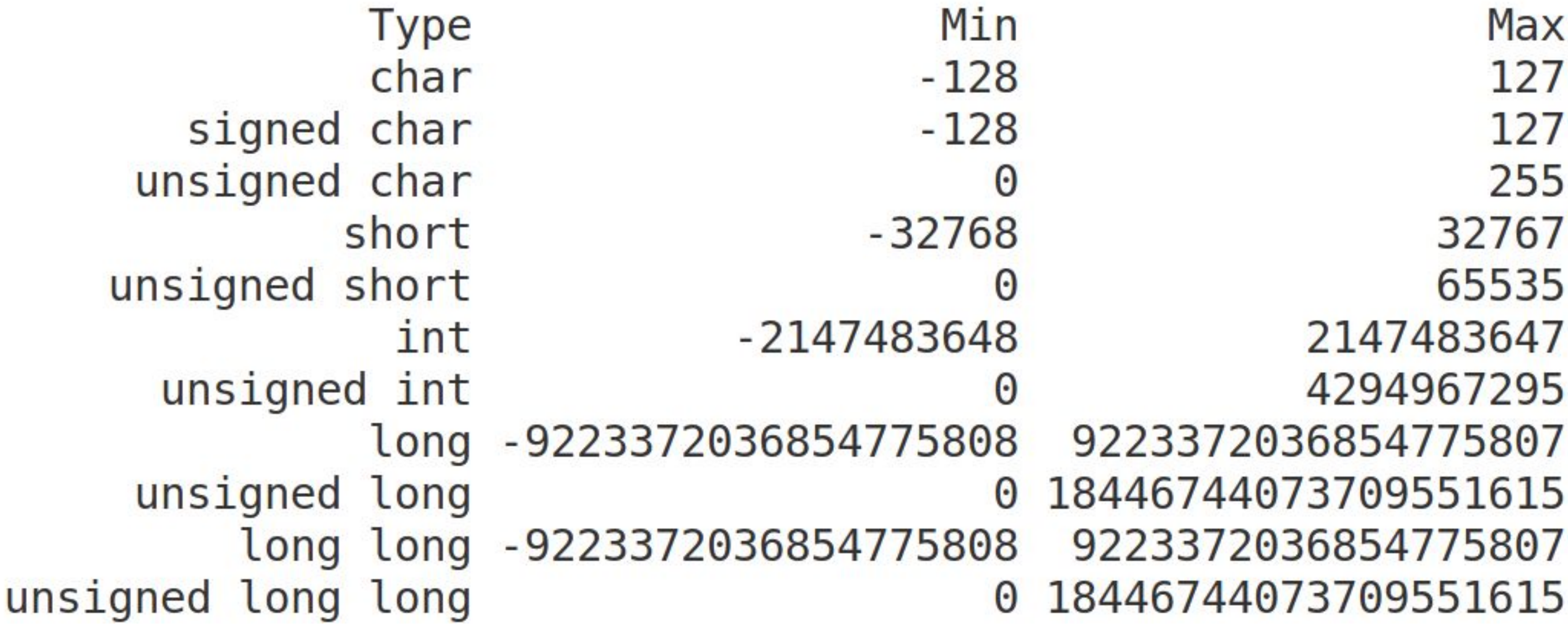
- C’s sizeof gives you number of bytes used for variable or type
  - sizeof variable - returns number of bytes to store variable
  - sizeof (type) - returns number of bytes to store type
- On CSE servers, C types have these sizes
  - char = 1 byte = 8 bits, 42 is 00101010
  - short = 2 bytes = 16 bits, 42 is 0000000000101010
  - int = 4 bytes = 32 bits, 42 is 00000000000000000000000000101010
  - double = 8 bytes = 64 bits, 42 = ?
- above are common sizes but not universal
- sizeof (int) might be 2 (bytes), or 4 (bytes)

integer_types.c - exploring integer types



```
              Type Bytes Bits
              char     1    8
       signed char     1    8
     unsigned char     1    8
             short     2   16
    unsigned short     2   16
               int     4   32
      unsigned int     4   32
              long     8   64
     unsigned long     8   64
         long long     8   64
unsigned long long     8   64
```

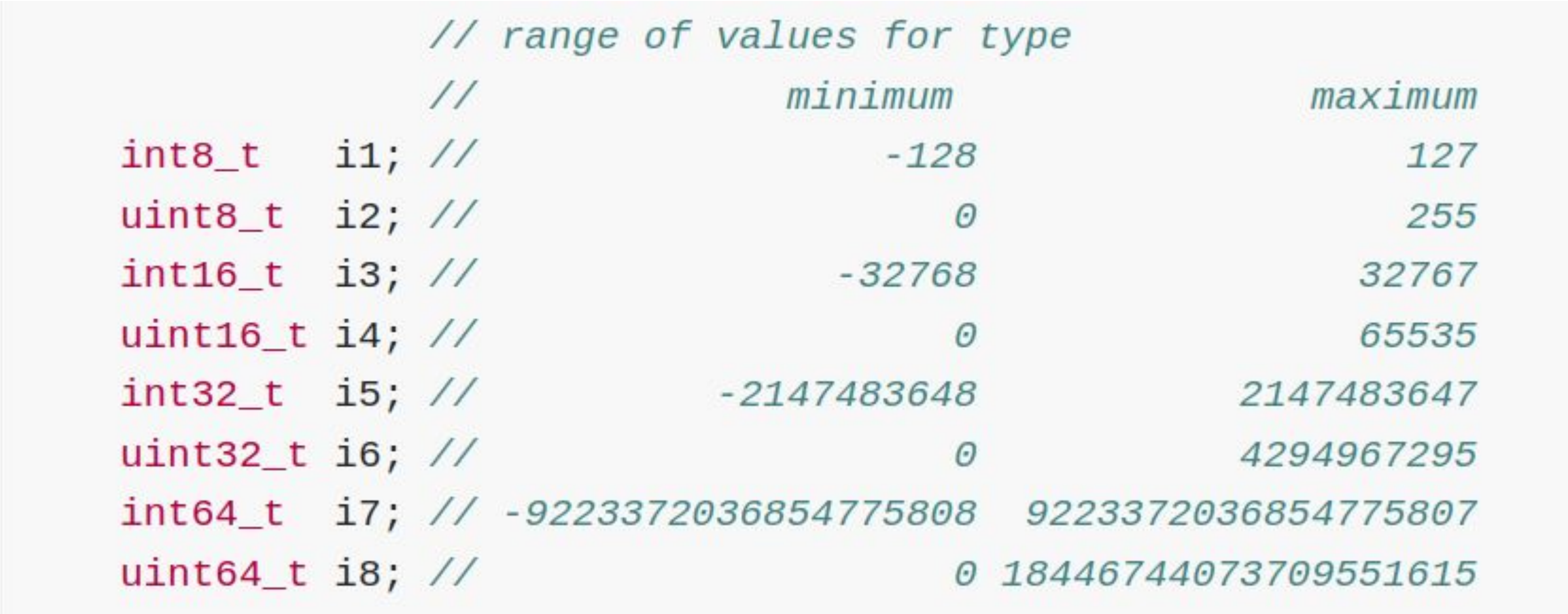
Exploring integer types



```
              Type                  Min                  Max
              char                 -128                  127
       signed char                 -128                  127
     unsigned char                    0                  255
             short               -32768                32767
    unsigned short                    0                65535
               int          -2147483648           2147483647
      unsigned int                    0           4294967295
              long -9223372036854775808  9223372036854775807
     unsigned long                    0 18446744073709551615
         long long -9223372036854775808  9223372036854775807
unsigned long long                    0 18446744073709551615
```

stdint.h - guaranteed size integer types

- `#include <stdint.h>` to get below int types (and more) with known sizes
- We use these a lot in COMP1521!



```
                  // range of values for type
                  //             minimum               maximum
    int8_t   i1; //                 -128                   127
    uint8_t  i2; //                    0                   255
    int16_t  i3; //               -32768                 32767
    uint16_t i4; //                    0                 65535
    int32_t  i5; //          -2147483648            2147483647
    uint32_t i6; //                    0            4294967295
    int64_t  i7; // -9223372036854775808   9223372036854775807
    uint64_t i8; //                    0  18446744073709551615
```