1 Binary

1.1 The basics

To begin, we will review the most basic concept of data representation in computers, binary.

- Recall that binary data has no meaning on its own: we need to be told (or come up with) an interpretation for it. Give five different possible ways we could interpret 1101.

- Convert the 8-bit binary number 01101001 into decimal. (because of the leading 0, it could be unsigned or signed)

- What is the decimal range of unsigned n-bit binary numbers?

We can convert positive numbers into binary by repeated division by 2. For example, to convert 23 into binary:

- \[ 23 / 2 = 11 \text{ remainder 1} \]
- \[ 11 / 2 = 5 \text{ remainder 1} \]
- \[ 5 / 2 = 2 \text{ remainder 1} \]
- \[ 2 / 2 = 1 \text{ remainder 0} \]
- \[ 1 / 2 = 0 \text{ remainder 1} \]

Then reading digits from bottom to top, we get \(23_{10} = 10111_2\). To verify this, we can work backwards: \(10111_2 = 2^4 + 2^3 + 2^1 + 2^0 = 16 + 4 + 2 + 1 = 23\). Leading digits are just 0, so in 8-bit binary we would write \(00010111\).

We can also convert positive numbers into binary by repeatedly subtracting the largest power of two.

- Convert the following numbers into unsigned 8-bit binary:
  1. 35
  2. 216

1.2 Two’s Complement

If we also want to be able to represent negative numbers as well as positive ones, we can use an encoding called “two’s complement”. The simplest way to use two’s complement is to encode the number as if it were unsigned (so if you want to encode \(-123\), encode \(123\)), then apply the two’s complement algorithm presented last week: flip all bits and add 1.
• Why does this work? Call $x^*$ the result of flipping all the bits of $x$ (so for example, if $x = 1011$ then $x^* = 0100$). Then $x + x^* = 1111 = 2^4 - 1$. So since we picked $-x = x^* + 1$, we get $x + (-x) = x + x^* + 1 = 2^4 - 1 + 1 = 2^4 \equiv 0 \mod 2^4$.

The second way of encoding $-x$ is to instead encode $2^4 - x$.

• Why does this work? $x + (-x) = x + 2^4 - x = 2^4 \equiv 0 \mod 2^4$.

Try the following in 8-bit binary:

1. Encode $-12$ using the two’s complement algorithm.
2. Encode $-123$ using the second way above.
3. Convince yourself that, for example, $- - 15 = 15$ by either two’s complement technique listed above. What is $- - 128$?

Side Question: Why do we use two’s complement instead of signed magnitude?

2 Assembly basics

2.1 Assembly instructions

Assembly languages are simple programming languages which let us load and store values to/from memory and in registers, and to perform arithmetic operations. We will use MIPS assembly language in this course. For example, \texttt{add $3$, $1$, $2$} means “add together the values in registers 1 and 2 and place the result in register 3.” Note that the destination comes first. Note that most instructions you can use in the course cannot take constant values as arguments!

2.2 Registers

• Registers in MIPS each hold 32 bits of information, and can be thought of as the variables that we are given to work with in our program.

• Some registers are special:
  – $0$ is always 0, and cannot be modified in any ways.
  – $31$ is reserved for the return address. If you lose the value of this register, your program cannot return.
  – We will make $3$, $29$, and $30$ special by convention in this course.

Try to get in the habit of using them only for their intended purpose, which will explained later on in the course.
2.3 Constant values

Most MIPS instructions in this course deal with registers instead of constant (immediate) values.
We solve this using the load immediate and skip (lis) instruction with the .word directive to load a value into a register. For example,

```assembly
lis $5
.word 7
```
Stores the value 7 into $5.

2.4 Assembling assembly language

The process of turning assembly language into machine code a computer understands is called assembling. Soon we will give you access to an assembler, but for now we’ll do it by hand. The MIPS assembly language reference sheet (https://www.student.cs.uwaterloo.ca/~cs241/mips/mipsref.pdf) gives a template for how to assemble each 32-bit instruction.

Assemble the following program by first writing out its binary representation, then converting it to a hexadecimal representation, and finally using cs241.wordasm to make it executable. What does it do? Run it with mips.twoints to verify it behaves as you’d expect.

```assembly
lis $5
.word 7
slt $6, $1, $5
beq $6, $0, 1
add $1, $1, $5
jr $31
```