Chapter 3

Arithmetic for Computers
- Introduction

Review: MIPS Design Principles

- Simplicity
  - Fixed size instructions.
  - Small number of instruction formats.
  - Opcode always the first 6 bits.
- Smaller is faster
  - Limited instruction set.
  - Limited number of registers in register file.
  - Limited number of addressing modes.
- Make the common case fast
  - Arithmetic operands from the register file only.
  - Allow instructions to contain immediate operands and branch targets.
Arithmetic for Computers

- Operations on integers
  - Addition and subtraction.
  - Multiplication and division.
  - Dealing with overflow.
- Floating-point numbers
  - Representation and operations.
  - Dealing with overflow and underflow.

Binary Representation

- The binary number

\[ 01011000 \ 00010101 \ 00101110 \ 11100111 \]

represents the decimal quantity:
\[ 0 \times 2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + \ldots + 1 \times 2^{0} \]

- A 32-bit word can represent \( 2^{32} \) numbers between 0 and \( 2^{32} - 1 \)
  - this is known as the unsigned representation, assuming that numbers are always positive.
Negative Numbers

32 bits can only represent $2^{32}$ numbers — if we wish to also represent negative numbers, we can represent $2^{31}$ positive numbers (incl zero) and $2^{31}$ negative numbers

\[
\begin{align*}
0000 0000 0000 0000 0000 0000 0000 0000_{\text{two}} &= 0_{\text{ten}} \\
0000 0000 0000 0000 0000 0000 0000 0001_{\text{two}} &= 1_{\text{ten}} \\
\ldots \\
0111 1111 1111 1111 1111 1111 1111 1111_{\text{two}} &= 2^{31} - 1 \\
1000 0000 0000 0000 0000 0000 0000 0000_{\text{two}} &= -2^{31} \\
1000 0000 0000 0000 0000 0000 0000 0001_{\text{two}} &= -(2^{31} - 1) \\
1000 0000 0000 0000 0000 0000 0000 0010_{\text{two}} &= -(2^{31} - 2) \\
\ldots \\
1111 1111 1111 1111 1111 1111 1111 1110_{\text{two}} &= -2 \\
1111 1111 1111 1111 1111 1111 1111 1111_{\text{two}} &= -1
\end{align*}
\]

2’s Complement

\[
\begin{align*}
0000 0000 0000 0000 0000 0000 0000 0000_{\text{two}} &= 0_{\text{ten}} \\
0000 0000 0000 0000 0000 0000 0000 0001_{\text{two}} &= 1_{\text{ten}} \\
\ldots \\
0111 1111 1111 1111 1111 1111 1111 1111_{\text{two}} &= 2^{31} - 1 \\
1000 0000 0000 0000 0000 0000 0000 0000_{\text{two}} &= -2^{31} \\
1000 0000 0000 0000 0000 0000 0000 0001_{\text{two}} &= -(2^{31} - 1) \\
1000 0000 0000 0000 0000 0000 0000 0010_{\text{two}} &= -(2^{31} - 2) \\
\ldots \\
1111 1111 1111 1111 1111 1111 1111 1110_{\text{two}} &= -2 \\
1111 1111 1111 1111 1111 1111 1111 1111_{\text{two}} &= -1
\end{align*}
\]

This format can directly undergo addition without any conversions.
Example

• Compute the 32-bit 2’s complement representations for the following decimal numbers:
  5, -5, -6

  5:   0000 0000 0000 0000 0000 0000 0000 0101
  -5:  1111 1111 1111 1111 1111 1111 1111 1011
  -6:  1111 1111 1111 1111 1111 1111 1111 1010
Signed / Unsigned

- The hardware recognizes two formats:
  - unsigned (corresponding to the C declaration `unsigned int`)
    -- all numbers are positive, a 1 in the most significant bit just means it is a really large number
  - signed (C declaration is `signed int` or just `int`)
    -- numbers can be +/-, a 1 in the MSB means the number is negative

This distinction enables us to represent twice as many numbers when we’re sure that we don’t need negatives.

MIPS Instructions

Consider a comparison instruction:
```
slt $t0, $t1, $zero
```
and $t1 contains the 32-bit number 1111 01…01

What gets stored in $t0?
MIPS Instructions

Consider a comparison instruction:

\[
\text{slt} \quad \$t0, \$t1, \$zero
\]

and $t1$ contains the 32-bit number $1111\ 01\ldots01$

What gets stored in $t0$?
The result depends on whether $t1$ is a signed or unsigned number – the compiler/programmer must track this and accordingly use either slt or sltu.

\[
\text{slt} \quad \$t0, \$t1, \$zero \quad \text{stores} \quad 1 \text{ in } \$t0
\]

\[
\text{sltu} \quad \$t0, \$t1, \$zero \quad \text{stores} \quad 0 \text{ in } \$t0
\]

Sign Extension Review

- Occasionally, 16-bit signed numbers must be converted into 32-bit signed numbers – for example, when doing an add with an immediate operand.

- The conversion is simple: take the most significant bit and use it to fill up the additional bits on the left – known as sign extension.

So $2_{10}$ goes from $0000\ 0000\ 0000\ 0010$ to $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010$

and $-2_{10}$ goes from $1111\ 1111\ 1111\ 1110$ to $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110$
Alternative Representations

The following two (intuitive) representations were discarded because they required additional conversion steps before arithmetic could be performed on the numbers.

- **sign-and-magnitude**: the most significant bit represents +/- and the remaining bits express the magnitude.

- **one’s complement**: -x is represented by inverting all the bits of x

Both representations above suffer from two zeroes.

Recap

- 2’s complement representation.
- Signed vs. Unsigned number representation.