CS 241 — Lecture 2

Bartosz Antczak *Instructor: Kevin Lanctot* *January 5, 2017*

2.1 More on representing numbers

2.1.1 Hexadecimal

It's hard to read binary because it's usually a long string of only 1's and 0's. To make these long strings easier to read, we can convert binary to hexadecimal.
To convert binary to hexadecimal, group the binary digits into groups of 4, then convert each group to their respective base-10 decimal. If the decimal is larger than 9, convert that digit to a new symbol ({A, B, C, D, E, F} respectively for {10, 11, 12, 13, 14, 15}).

Example 2.1.1. *Convert 101101000110 to hexadecimal*

1011	0100	0110	Binary
11	4	6	Base-10
B	4	6	Hexadecimal

Thus, $101101000110_2 = B46_{16}$.

Why use hexadecimal?

Humans represent numbers in decimal, and computer represent numbers in binary. Hexadecimal is a "compromise" between these two numbering systems because it's easy to convert to and from binary, thus allowing humans to better communicate with computers.

2.1.2 Interpreting data

What does 101010101010110101111 0 represent? Computers refer to everything as either a 0 or 1, so that string could be a number, instruction, a picture, or just about anything! We must keep track of what the data means.

Data representation

- **Bit**: a single 1 or 0
- **Nibble**: 1 hexadecimal digit = 4 bits
- **Byte**: 2 hexadecimal digits = 8 bits
- **Word**: (depends on the processor):
  - 32-bit architecture: 1 word = 4 bytes = 32 bits (in CS 241, we'll use a 32-bit architecture)
  - 64-bit architecture: 1 word = 8 bytes = 64 bits

ASCII

ASCII (American Standard Code for Information Interchange) is a method of representing data. ASCII represents numeric, alphabetic, or special characters as a decimal (i.e., 'A' in ASCII is 65_{10}, or ';' is 59_{10}, or `null` is 0_{10}).
A problem with ASCII is that it's only helpful for English characters — what about Russian, French, or Greek? A solution is *Unicode*, which is a standard for most languages.

2.2 MIPS Assembly Language

MIPS is an assembly language, which is what a high-level languages (such as C++, Python, or Racket) get converted into. MIPS then gets converted into machine code (which is simply binary).
Multiple revisions exist for MIPS, and we'll work with MIPS32, which is a 32-bit architecture.
MIPS stores data in *registers*. There are 32 registers, called \$0, ... , \$31. Three registers are reserved:

- \$0: always contains 0
- \$30: contains stack pointer
- \$31: contains return address

The instructions in MIPS are written out one statement per line and is split into two parts:

- **opcode**: the operation of the instruction (i.e., add or sub)
- **operands**: the data sources and destinations, which are either registers or memory locations in RAM

2.2.1 How MIPS is converted to machine code

Similarly to how a high-level language is converted to MIPS, there is a standard that's used to convert MIPS to machine code.
Say we want to convert `add $d, $s, $t` to machine language. The template for the `add` opcode is: 000 00*ss ssst tttt dddd* d000 0010 0000, where s, d and t are the respective binary conversions of the register numbers (i.e., \$2 has register number 2 which is then converted to 00010).
A list of these conversion templates will be provided on exams.

2.2.2 Basic MIPS instructions

- `add $d, $s, $t` — adds the contents of register `$s` and `$t` and stores it in register `$d`
- `lis $d`
  `.word i` (is actually a *pseudo instruction*, which means it's provided as a convenience and gets converted into other MIPS instructions, and just to clarify this instruction takes up two lines): loads the value of `i` into register `$d`
- `jr $s`: jumps to the address stored in register `$s`