CS 241: Foundations of Sequential Programs

Chris Thomson

Winter 2013, University of Waterloo

Notes written from Gordon Cormack's lectures.

1 Introduction & Character Encodings

← January 7, 2013

1.1 Course Structure

The grading scheme is 50% final, 25% midterm, and 25% assignments. There are eleven assignments. Don't worry about any textbook. See the course syllabus for more information.

1.2 Abstraction

Abstraction is the process of removing or hiding irrelevant details. Everything is just a sequence of bits (binary digits). There are two possible values for a bit, and those values can have arbitrary labels such as:

- Up / down.
- Yes / no.
- 1 / 0.
- On / off.
- Pass / fail.

Let's say we have four projector screens, each representing a bit of up/down, depending on if the screen has been pulled down or left up (ignoring states between up and down). These screens are up or down independently. There are sixteen possible combinations:

Screen 1	Screen 2	Screen 3	Screen 4
Up (1)	Down (0)	Up (1)	Down (0)
Down (0)	Down (0)	Down (0)	Up (1)
⋮	⋮	⋮	⋮

Note that there are sixteen combinations because $k = 4$, and there are always 2^k combinations since there are two possible values for each of k screens.

1.3 Endianness

Let's consider the sequence 1010. This sequence of bits has a different interpretation when following different conventions.

- **Unsigned, little-endian**: $(1 \times 2^0) + (0 \times 2^1) + (1 \times 2^2) + (0 \times 2^3) = 1 + 4 = 5$.
- **Unsigned, big-endian**: $(0 \times 2^0) + (1 \times 2^1) + (0 \times 2^2) + (1 \times 2^3) = 2 + 8 = 10$.
- **Two's complement, little-endian**: $5 - 16 = -10$.
- **Two's complement, big-endian**: $10 - 16 = -6$.
- **Computer terminal**: LF (line feed).

Note that a two's complement number n will satisfy $-2^{k-1} \leq n < 2^{k-1}$.

1.4 ASCII

ASCII is a set of meanings for 7-bit sequences.

Bits	ASCII Interpretation
0001010	LF (line feed)
1000111	G
1100111	g
0111000	8

In the latter case, 0111000 represents the character '8', not the unsigned big- or little-endian number 8.

ASCII was invented to communicate text. ASCII can represent characters such as A-Z, a-z, 0-9, and control characters like ();!. Since ASCII uses 7 bits, $2^7 = 128$ characters can be represented with ASCII. As a consequence of that, ASCII is basically only for Roman, unaccented characters, although many people have created their own variations of ASCII with different characters.

1.5 Unicode

Unicode was created to represent more characters. Unicode is represented as a 32-bit binary number, although representing it using 20 bits would also be sufficient. The ASCII characters are the first 128, followed by additional symbols.

A 16-bit representation of Unicode is called **UTF-16**. However, there's a problem: we have *many* symbols ($> 1M$) but only $2^16 = 65,536$ possibilities to represent them. Common characters are represented directly, and there is also a 'see attachment' bit for handling the many other symbols that didn't make the cut to be part of the $65,536$. Similarly, there is an 8-bit representation of Unicode called **UTF-8**, with the ASCII characters followed by additional characters and a 'see attachment' bit.

The bits themselves do not have meaning. Their meaning is in your head – everything is up for interpretation.

1.6 A Message for Aliens

← January 9, 2013

In a computer, meaning is in the eye of the beholder. We must agree on a common interpretation – a convention. However, the English language and numbers also have their meaning determined by a set of conventions.

NASA wanted to be able to leave a message for aliens on a plaque on their spacecraft, however it was clear that aliens would not understand our language or even 0s and 1s. NASA wanted their message to be a list of prime numbers. They decided they would use binary to represent the numbers, but since 0s and 1s would be ambiguous to aliens, they used a dash (-) instead of 0, and 1 for 1. It's only a convention, but it's one that NASA determined aliens would have a higher chance of understanding.

1.7 Hexadecimal

Hexadecimal (hex) is base 16. It has sixteen case-insensitive digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f.

Why is hex useful? It makes conversions easy. We group bits into sequences of four:

$$\underbrace{\underbrace{0011}_{3}\underbrace{1010}_{A}}_{3A}$$

Conversions are made especially easy when the sequences of bits are lengthy:

$$\underbrace{\underbrace{10}_{2}\underbrace{1110}_{E}\underbrace{0111}_{7}\underbrace{0011}_{3}\underbrace{1011}_{B}\underbrace{1001}_{9}\underbrace{1000}_{8}\underbrace{0011}_{3}}_{2E73B983}$$

2 Stored Program Computers

Stored program computers are also known as the **Von Neumann architecture**. They group bits into standard-sized sequences.

In modern times, standard-sized sequences of bits are:

- **Bytes**. A byte is 8-bits (256 possible values). Example: 00111010.
- **Words**. A word is only guaranteed to be "more than a byte." Words are often 16-bit ($2^{16} = 65,536$ possible values), 32-bit ($2^{32} \approx 4 \times 10^9$), or 64-bit ($2^{64} \approx 10^{19}$).

2.1 Storage Devices

2.1.1 Registers

There are typically a finite number of fixed-sized sequence of bits, called **registers**. You can put bits in, peek at them, and modify them. A "64-bit CPU" just means it's a CPU that uses 64-bit words.

Calculators typically have 2-3 registers for recalling numbers and maintaining state.

There are a couple of downsides to registers. They're expensive to build, which is why there is a finite number of them. They're also difficult to keep track of.

2.1.2 RAM (Random Access Memory)

RAM is essentially a physical array that has **address lines**, **data lines**, and **control lines**. Data is fed into RAM using electrical lines. Data will remain in RAM until overwritten.

If you want to place a happy face character at address 100, you set the address lines to 100, the data lines to 10001110 (which is the Unicode representation of a happy face), and give the control lines a kick.

RAM could be implemented in several different ways. It could even be created with a **cathode ray tube**. The **core** method is synonymous with RAM, however. It involves a magnetic core, and the data remains magnetized after the magnet is removed. Bits are read by toggling the state (toggling the magnetic poles) and seeing if it was easier to toggle than expected (similar to unlocking an already-unlocked door), and then toggling back after. No one really uses magnetic cores anymore.

Capacitive memory (also known as dynamic RAM or **DRAM**) is still used today. It involves an insulator, and two conductive plates, one more negatively-charged than the other. The electrons will remain in their state even when the poles are removed. There is a problem, however. Insulators are not perfect – electrons will eventually make their way through the insulator. In order to alieviate this, we have to refresh the charge fairly often (every second, for instance).

Switches are typically used only for registers and cache. They produce more heat, but are much faster.

2.2 Control Unit Algorithm

The CPU contains a **control unit**, several **registers**, PC (**program counter**), and IR (**instruction register**), and is connected to RAM with electrical lines.

```
PC <- some fixed value (e.g. 0)
loop
  fetch the word of RAM whose address is in PC, put it in IR
  increment PC
  decode and execute the machine instructon that's in IR
end loop
```

IR would contain an instruction like "add register 1 to register 2, and put the result into register 7."

3 MIPS

← January 11, 2013

3.1 Unix

You'll need Unix to use MIPS in this course. Unix was originally created in the 1970s at AT&T Bell Labs. Unix is still popular today, especially for servers. Linux is a Unix dialect, and Mac OS X is also based on Unix.

Unix has three types of files:

- **Binary files**. A sequence of arbitrary bytes.
- **Text files**. A sequence of ASCII characters, with lines terminated by a LF / newline.
- **Tools**. These are programs, which are technically binary files.

3.1.1 Getting Access to a Unix System

If you use Linux or Mac OS X, you're in good shape. However, Windows is not Unix-based, so you'll have to pursue one of these alternative options:

- Install Linux. You can dual-boot it alongside Windows if you'd like, or you could install it inside a virtual machine.
- Install Cygwin. When installing it, choose to install everything.
- Login to the `student.cs` servers remotely. You can use PuTTY for that.

3.1.2 Commands You Should Know

- `ssh username@linux.student.cs.uwaterloo.ca` – logs you into the student.cs systems remotely through SSH.
- `cat unix_text_file.txt` – copies the contents of the file to the current terminal. If a non-text file is given to `cat`, incorrect output will result.
- `xxd -b unix_file.txt` – prints the binary representation of the file to the terminal. The numbers in the left column are the location in the file. If it's a Unix text file, the ASCII representation is presented on the right, with all non-printable characters printed as dots. `xxd` is not aware of newline characters – it arbitrarily splits the file into 16 bytes per line.
- `xxd unix_file.txt` – prints the hex representation of the file to the terminal. Identical to the previous command (`-b`) in every other way.
- `ls -l` – lists all files in the current directory in the long-listing form, which shows the number of bytes in the file, permissions, and more.

3.2 Getting Started with MIPS

The MIPS CPU uses 32-bit words since it's a 32-bit machine, and it's big-endian. You can use `xxd` to inspect MIPS files. MIPS has 32 registers (numbered 0 to 31).

At the end of our MIPS programs, we will copy the contents of register \$31 to the program counter (PC) to "return".

3.2.1 Running MIPS Programs

Upon logging in to the `student.cs` servers, run `source ~cs241/setup` in order to add the required executables to your `PATH`. Then, when given a MIPS executable called `eg0.mips`, you can run `java mips.twoints eg0.mips` in order to run the program.

`mips.twoints` is a Java program that requests values for registers \$1 and \$2 and then runs the given MIPS program.

3.2.2 Creating MIPS Programs

Start with `vi thing.asm` (or use your favorite editor). Inside this file, you'll create an **assembly language file**, which is a textual representation of the binary file you want to create. Each line in this file should be in the form `.word 0xabcdef12` (that is, each line should start with `.word 0x` – the `0x` is a convention that indicates that hex follows). You can add comments onto the end of lines, starting with a semi-colon (Scheme style).

Next, you'll need to convert your assembly language file into a binary file. You can do that by running `java cs241.wordasm < thing.asm > thing.bin`. You can then inspect `thing.bin` with `xxd` in hex, or in binary if you're masochistic.

A few important things you should know for developing MIPS programs:

- \$0 is a register that will always contain 0. It's special like that.
- \$30 points to memory that could be used as a stack.
- \$31 will be copied to the program counter at the end of execution in order to "return".
- You can specify register values using base 10 values or as hex values (if prefixed by `0x`).
- It takes 5-bits to specify a register, since $2^5 = 32$.
- It's convention to call S and T (as indicated in various documentation) **source registers**, and D is the **destination register**.
- MIPS uses two's complement numbers by default, unless specified otherwise.
- Loops and conditionals are accomplished by adding or subtracting from the program counter.

There is a MIPS reference sheet available on the course website that you'll find to be quite useful. It contains the binary representations for all MIPS instructions. Convert the binary into hex and put them into an assembly language file.

3.2.3 A Few Important MIPS Instructions

1. **Load Immediate & Skip** (`lis`): loads word from the program counter. Loads the next word of memory into the D register. You specify a `lis` instruction followed by an arbitrary word next. You need to also skip the appropriate number of bytes by incrementing the program counter.
2. **Set Less Than [Unsigned]** (`slt`): compares S to T. If S $<$ T, 1 is put into the D register, otherwise 0 is put into the D register.
3. **Jump Register** (`jr`): copies S to the program counter.
4. **Jump and Link Register** (`jalr`): assigns the program counter to register 31, then jumps to it.
5. **Branch on Equal** (`beq`): if S is equal to T, it adds the specified number to the program counter (times 4). There is also **Branch on Unequal** (`bne`) which does the opposite.