1 Binary

1.1 Converting from Binary to Decimal

The goal of this course is to learn how programs are generated and executed. To begin, we will review binary representations, the way data is represented in computers.

Recall that binary data has no meaning on its own: we need to be told, come up with, or assume an interpretation for it.

Exercise: Give five different possible ways we could interpret the binary sequence 1000.

Solution: Here are some “natural” ways of interpreting this sequence:

- As an unsigned 4-bit number, this is 8.
- As a signed (two’s complement) 4-bit, this is -8.
- An array of four booleans, where the first is “true” and the next three are “false”.
- The “backspace” character (ASCII code 8).
- The address of the third word in the MIPS machine’s memory, where the first word is at address zero.

However, bits can mean anything you want. You could write a program that interprets this bit sequence as a picture of a cat, a piece of music, or the decimal number one thousand.

We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

\[ 11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0) = 16 + 8 + 2 = 26. \]

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

- \( 23 / 2 = 11 \) remainder 1
- \( 11 / 2 = 5 \) remainder 1
- \( 5 / 2 = 2 \) remainder 1
- \( 2 / 2 = 1 \) remainder 0
- \( 1 / 2 = 0 \) remainder 1

Then reading the remainders from bottom to top, we get 10111 for the binary representation of 23. To verify our result, we can work backwards: \( 10111 = 2^4 + 2^2 + 2^1 + 2^0 = 16 + 4 + 2 + 1 = 23 \). To write this as an 8-bit binary number, we would just add leading zeroes: 00010111.
Exercises:

• Convert the 8-bit unsigned binary number 01101001 into decimal.
  **Solution:** \(2^6 + 2^5 + 2^3 + 2^0 = 64 + 32 + 8 + 1 = 105\).

• Convert the following numbers into unsigned 8-bit binary:
  1. 35
  **Solution:** 35 breaks down into \(32 + 2 + 1 = 2^5 + 2^1 + 2^0\), which gives 00100011. You can also obtain this by repeated division:
    - \(35/2 = 17 \text{ remainder } 1\)
    - \(17/2 = 8 \text{ remainder } 1\)
    - \(8/2 = 4 \text{ remainder } 0\)
    - \(4/2 = 2 \text{ remainder } 0\)
    - \(2/2 = 1 \text{ remainder } 0\)
    - \(1/2 = 0 \text{ remainder } 1\)
  Reading bottom to top gives 100011, then adding leading zeroes gives 00100011.

2. 216
  **Solution:** The mental arithmetic for breaking this number down into powers of two is a bit harder than in the previous question. Let’s just use the division method:
    - \(216/2 = 108 \text{ remainder } 0\)
    - \(108/2 = 54 \text{ remainder } 0\)
    - \(54/2 = 27 \text{ remainder } 0\)
    - \(27/2 = 13 \text{ remainder } 1\)
    - \(13/2 = 6 \text{ remainder } 1\)
    - \(6/2 = 3 \text{ remainder } 0\)
    - \(3/2 = 1 \text{ remainder } 1\)
    - \(1/2 = 0 \text{ remainder } 1\)
  This gives 11011000. Working backwards, we verify that \(2^7 + 2^6 + 2^4 + 2^3 = 128 + 64 + 16 + 8 = 216\).

1.2 Two’s Complement

To represent negative numbers as well as positive ones, we use an encoding called “two’s complement”. Two’s complement encoding is based on modular arithmetic: instead of interpreting binary sequences as positive integers, we interpret them as integers modulo \(2^b\) (where \(b\) is the number of bits). This representation is nice because addition, subtraction and multiplication work the same way for ordinary integers as they do for integers modulo \(2^b\), so arithmetic is straightforward.

However, recall from MATH 135 that there are infinitely many integers which are congruent to a given integer modulo \(2^b\). We use a simple rule to choose a single integer corresponding to each bit sequence:

• If the leftmost bit is 0, just treat it as an unsigned binary number.
• If the leftmost bit is 1, interpret it as an unsigned binary number and then subtract $2^b$. This gives the smallest (in absolute value) negative number that is congruent to the unsigned binary number.

The range of values for unsigned $b$-bit binary representation ranges from 0 to $2^b - 1$, while the range of values for signed two’s complement $b$-bit binary representation ranges from $-2^{b-1}$ to $2^{b-1} - 1$. For example, for 8-bit binary, the unsigned range is 0 to 255 while the signed range is $-128$ to 127. Take note of the asymmetry in the signed range.

Here are two techniques for negating a two’s complement number:

• Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
• Flip all the bits to the left of the rightmost 1.

Proving that these techniques work is a fun mathematical exercise. These techniques give a useful way to convert between decimal and two’s complement binary. To convert a negative decimal number $-x$ to binary, just convert the positive number $x$ and then use one of the techniques to negate it. Likewise, to convert a two’s complement binary sequence to decimal, if the leftmost bit is 0 just treat it as an unsigned number; if the leftmost bit is 1, first negate the number, treat it as unsigned and convert it, and negate it again.

For example consider the 4-bit two’s complement number 1010. Using the first technique, we flip the bits to get 0101, then add 1 to get 0110, which is 6. So 1010 is $-6$ in decimal. The second technique gives the same result: flipping the bits to the left of the rightmost 1 gives 0110.

**Exercises:** Try the following in 8-bit two’s complement binary:

1. Convert $-12$ to binary using both of the two’s complement negation techniques.

   **Solution:** 12 in unsigned binary is 00001100. Flipping the bits gives 11110011. Adding 1 gives 11110100. The other method is to flip the bits to the left of the rightmost 1 in 00001100, which gives 11110100 as expected.

2. Sometimes the following fact is useful: in $b$-bit binary, the two’s complement representation of $-x$ is equal to the unsigned representation of $2^b - x$ (because they are congruent modulo $2^b$). Convert $-123$ to binary using this fact.

   **Solution:** We have $256 - 123 = 133$. We see that $133 = 128 + 4 + 1 = 2^7 + 2^2 + 2^0$ and so the binary representation of $-123$ is 10000101.

3. Use one of the two’s complement negation techniques to negate $-128$. What happens?

   **Solution:** The two’s complement representation of $-128$ is 10000000. If we flip the bits we get 01111111. Adding 1 gives... 01111111 again! The “negation” of $-128$ is just $-128$ again. This is because 128 is congruent to $-128$ modulo $2^8 = 256$.

2 Assembly Language

2.1 Assembly Language Instructions

Assembly languages are simple programming languages which let us perform simple arithmetic and logic operations and transfer values between registers and memory. We will use (a very simplified version of) MIPS assembly language in this course. For example, `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, just like assignment statements in most programming languages, e.g., $r3 = r1 + r2$. 

3
2.2 Registers

- Registers in (our simplified) MIPS architecture each hold 32 bits of information, and can be thought of as being similar to variables: they store data and allow the data to be accessed and manipulated.

- Some registers are special:
  - $0 always contains the value 0 and the contents of $0 cannot be modified in any way.
  - $31 is reserved for the return address of the program (or later in the course, procedures).
  - Registers $3, $29, and $30 are special by convention in this course.

Avoid using these special registers for things like computations or temporary values. Their intended purpose will be explained later on in the course.

2.3 Constant values

Most MIPS instructions in this course deal with values stored in registers instead of constant values. While real MIPS machines have instructions that allow you to do things like add a constant to a value in a register, the simplified MIPS machine in this course lacks these instructions.

To load constant values into registers, we use the Load Immediate and Skip (lis) instruction together with the .word directive. For example, this snippet of assembly language stores the value 7 into $5.

\[
\text{l}i\text{s }5
\]
\[
\text{.word } 7
\]

The “Load Immediate and Skip” name comes from the fact that the \textit{lis} instruction \textit{skips} over the constant value 7 after loading it into the register. If it wasn’t for the skip, the MIPS machine would try to execute the binary representation of 7 as if it was an instruction.

Constant values are often called \textit{immediate values} in the context of assembly language.

2.4 Assembling an Instruction

The process of converting assembly language into code the computer can process (i.e., machine code) is called \textit{assembling}. Soon we’re going to automate this process, but for now well 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.

There are two instruction formats: register and immediate (denoted R and I in the second-last column on the MIPS reference sheet). Suppose we want to assemble the register-format instruction \texttt{mult $5, $4}.

- First, we need to convert 5 into 5-bit binary: 00101. This is s, as seen on the reference sheet.
- Next, we need to convert 4 into 5-bit binary: 00100. This is t.
- Next, we simply need to replace the s and t values on the reference sheet for \texttt{mult} with these binary sequences. Since \texttt{mult} has no d register, 00000 is simply substituted in that position (and has already been substituted for you on the reference sheet). We end up with: 0000 0000 1010 0100 0000 0000 0001 1000
- Finally, we need to rewrite this as hexadecimal. Looking up a table or just converting each nibble (4-bit chunk) we end up with 0x00A40018.
Assembling immediate-format instructions such as `beq $0, $1, -2` is very similar, except they include a 16-bit two’s complement immediate value that you must convert from decimal (−2 in this example).

### 2.5 Exercise

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 that it behaves as you’d expect.

```
lis $5
.word 7
add $1, $1, $5
beq $0, $1, -2
jr $31
```

**Solution:** Here is the binary and hexadecimal encoding of each instruction:

```
lis $5
0000 0000 0000 0000 0010 1000 0001 0100
0x00002814

.word 7
0000 0000 0000 0000 0000 0000 0000 0111
0x00000007

add $1, $1, $5
0000 0000 0010 0101 0000 1000 0010 0000
0x00250820

beq $0, $1, -2
0001 0000 0000 0001 1111 1111 1111 1110
0x1001fffe

jr $31
0000 0011 1110 0000 0000 0000 0000 1000
0x03e00008
```

The final result:

```
.word 0x00002814
.word 0x00000007
.word 0x00250820
.word 0x1001fffe
.word 0x03e00008
```

This program adds 7 to the value in $1, and then checks if $1 is zero. If so, it branches back and performs the addition again, repeating until $1 is non-zero. The result is that the program computes the following function and stores its value in $1:

\[
 f(n) = \begin{cases} 
 n + 7, & \text{if } n \neq -7; \\
 7, & \text{if } n = -7. 
\end{cases}
\]
3 Setup

To access the student environment via the command line, you must `ssh` into your user account. Linux/Mac users can use Terminal and execute the command:

```bash
ssh userid@linux.student.cs.uwaterloo.ca
```

Windows users should use an ssh client called PuTTY, or Cygwin, or use a virtual machine.

3.1 Password-less ssh

Entering your password every time is tedious, but there are simple steps you can take to use password-less ssh.

Instructions for those on Windows machines using PuTTY can be found at [https://www.getfilecloud.com/blog/ssh-without-password-using-putty/](https://www.getfilecloud.com/blog/ssh-without-password-using-putty/).

For Linux/Mac users, you can enter the two following commands:

```bash
$> ssh-keygen -t rsa
```

Follow the prompt. If you used the default options by just pressing enter for all inputs, this will generate a pair of keys and put them in `~/.ssh/` with the name "`id_rsa`" for your private key, and `id_rsa.pub` for your public key.

After that you should append your new public key to `~/.ssh/authorized_keys`:

```bash
$> cat ~/.ssh/id_rsa.pub | ssh userid@linux.student.cs.uwaterloo.ca "cat >> ~/.ssh/authorized_keys"
```

Subsequent public keys can be appended to this file. This means that if you wanted to add another public key for another computer on this server, you would append the contents of the second `id_rsa.pub` file into a new line on the existing `authorized_keys` file.

You now can now access the student environment from your machine without a password.

3.2 .profile

- When you log into the CS environment there are a number of files that get executed. One of these files is `~/.profile`.
- For convenience, edit `.profile` to include the command `source /u/cs241/setup`. This will save you from having to source CS241 tools every time you ssh.
- Aliases, for commands you use often, should also be added to `.profile`. For example:
  ```bash
  alias xxd4='xxd -bits -cols 4'
  ```
- You can also write your own functions and make them available in `.profile` to simplify repeated tasks.
- Bonus tip for Unix users: You can similarly edit profile file for the shell you are using and add the following alias to simply logging into to the student environment.
  ```bash
  alias lse='ssh userid@linux.student.cs.uwaterloo.ca'
  ```
3.3 Further Reading

When it comes to editing files/programs, some of you prefer to use the student environment using Vim or Emacs (or Nano), while others prefer to edit on their local machines using a graphical editor such as Atom. Note that CS241 tools are only available on the student environment.

For those of you who wish to edit files on their local machine, you may want to mount the student environment as a network drive on your computer, to avoid having to copy files to/from the student environment for testing. You are not required to do this, but it has been helpful and time-saving for students in the past.

On Windows 10, you can mount the student environment as a network drive as follows. Note that this will only work while on campus and connected to Eduroam. If you want to do this off-campus, you will need to connect to the campus network through a VPN (https://uwaterloo.ca/information-systems-technology/services/virtual-private-network-vpn/about-virtual-private-network-vpn).

1. Open the “This PC” window (if you don’t know where to find it, open the start menu and type “This PC” to search for it).
2. In the toolbar at the top, click “Map network drive”.
3. In the “Folder” box, type \smb-files.student.cs.uwaterloo.ca\userID, where userID should be replaced with your Quest user ID.
4. Select the “Reconnect at sign-in” and “Connect using different credentials” boxes. Click “Finish”.
5. A login box should appear. If your username is already filled in, click “More choices” and then “Use a different account” to provide a different username.
6. For the username, enter CS-TEACHING\userID. For the password, enter your usual student environment password.

On Mac OSX the instructions are similar:
1. Open Finder and on the menu at the top, click “Go” then “Connect to Server”.
2. Enter the address \smb-files.student.cs.uwaterloo.ca\userID.
3. When prompted for a username and password, select “Registered User” and enter CS-TEACHING\userID for the username, and your usual student environment password.

There are also a variety of external programs that can accomplish this, not limited to SSHFS (https://www.digitalocean.com/community/tutorials/how-to-use-sshfs-to-mount-remote-file-systems-over-ssh), FileZilla, and Cyberduck. If you are unable to connect, the CSCF help desk (DC 2608) or MFCF help desk (MC 3017) may be able to assist you.