CS 241 – Week 1 Tutorial

Binary and assembly basics

Spring 2018

1 Binary

1.1 Binary basics

In school thus far you've largely done all of your arithmetic in base 10, but in math class you've likely seen other bases as well. Computers generally work in base 2 ("binary") and store all information as numbers. A binary *word* is simply a binary number, possibly with leading zeroes. A *bit* is a binary digit, and a *byte* is eight bits. Most computers have a standard *word size* and prefer to do their operations on numbers with that many bits.

1.1.1 Exercises

1. What is the meaning of the binary word 00101101?
2. Rewrite the number 1011_2 in decimal.

1.2 Converting positive decimal numbers to binary

We can convert positive numbers into binary by repeated division by 2. Using 23 as an example:

- $23/2 = 11$ remainder 1
- $11/2 = 5$ remainder 1
- $5/2 = 2$ remainder 1
- $2/2 = 1$ remainder 0
- $1/2 = 0$ remainder 1

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

1.2.1 Exercises

Convert the following numbers into unsigned 8-bit binary:

1. 35
2. 216

1.3 Converting negative decimal numbers to binary

If we also want to be able to represent negative numbers as well as positive ones, we use an encoding called "two's complement". The two's complement encoding can feel somewhat arbitrary, but is based off of the important property that $x + (-x) \equiv 0 \mod 2^b$. In two's complement, positive numbers always start with a 0 and negative numbers with a 1, but this doesn't just mean that $-x$ is written the same way as $+x$ but with a different first digit!

You can find the two's complement encoding of a number in b-bit binary in a number of ways. Two simple ones are:

- 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 $x + x^* = 1111 = 2^b - 1$. So since we picked $-x = x^* + 1$, we get $x + (-x) = x + x^* + 1 = 2^b - 1 + 1 = 2^b \equiv 0 \mod 2^b$.

- Instead of encoding $-x$, encode $2^b - x$.

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

Notice that if $1x$ is a two's complement number then $11x$ is the same number, just as $0x$ and $00x$ are the same number.

1.3.1 Exercises

Try the following in 8-bit binary:

1. Encode -12 using the first technique.
2. Encode -123 using the second technique.
3. Compute the 8-bit two's complement binary encodings of 15, -15, and $--15$ to convince yourself that $15 = --15$.
4. Repeat the above exercise using 128 instead of 15. What do you notice?

2 Assembly basics

2.1 Assembly instructions

Assembly languages are simple programming languages which let us make simple modifications to locations in memory, and especially special storage locations called registers. We will use a 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": notice that the destination comes first. Note that most instructions 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.
- $31 is reserved for the return address.
- $1 and $2 contain the program's arguments, but are not special once it begins running.
- We will make $3, $29, and $30 special by convention in this course.

Dont get in the habit of using registers for things other than their intended purpose, since later in the course we'll need to avoid using them for other things to make sure our compiler works!

2.3 Constant values

As we noticed earlier, most instructions don't accept constant values.

We solve this using the lis instruction with the .word directive to load a constant value into a register. For example,

```
lis $5
.word 7
```

Stores the constant 7 into $5.

2.4 Assembling assembly language

The process of turning assembly language into code the computer understands is called assembling. Soon we're going to automate this process, but for now we'll do it by hand. The MIPS assembly language reference sheet gives a template for how to assemble each instruction.

2.5 Assembling an instruction: register format

There are two formats given on the reference sheet for assembling instructions: 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 `mult $5, $4`.

- First, we need to convert 5 into a 5-bit binary word: 00101. This is "s", as seen on the reference sheet.
- Next, we need to convert 4 into a 5-bit binary word: 00100. This is "t"
- Next, we simply need to replace the "s" and "t" values on the reference sheet for `mult` with these binary words. Since `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 remembering the bit patterns we end up with `00A40018`.

Alternatively, if we wanted to follow the generic register-format template at the top of the sheet, we need to know the "f" value (the opcode for `mult`), which can only be found in the `mult` instruction's template and is 011000

2.6 Assembling an instruction: immediate format

Immediate-format instructions are assembled similarly to register-format instructions, except that the `i` value is a 16-bit two's complement number, and the opcode comes at the start of the instruction rather than at its end.

For example, if we want to assemble `lw $31, 23($11)`

- First, we need to convert 11 into a 5-bit binary word: 01011. This is "s".
- Next, we need to convert 31 into a 5-bit binary word: 11111. This is "t".
- Next, we need to convert 23 into a 16-bit binary word: 0000000000010111. This is "i".
- Next, we need to substitute the values for "s", "t", and "i" into the template for `lw`, in the same way that we did for register-format instructions. We end up with:

  `1000 1101 0111 1111 0000 0000 0001 0111`

- Finally, we need to rewrite this as hexadecimal. Looking up a table or just remembering the bit patterns we end up with `8D7F0017`.

2.7 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 it behaves as you'd expect.

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