EECS 370

Lab 2: Project 1 - LC2K ISA

Today's Discussion

- More on Binary: 2's complement & multiplication
- How to translate to LC2K
- How to write in LC2K

Negative Numbers Using 2's Complement

A way to represent integers using binary, both positive and negative

In 2's Complement, the most significant bit (MSB) is negative

For example, in a 4-bit number,
the MSB typically represents 2^3, but in 2's Complement,
the MSB ***instead*** represents -2^3

So, if the MSB is 0, the number will be ***positive (or zero)***
And, if the MSB is 1, the number will be ***negative***

The other bits remain the same, let's see how with an example:

Converting 3_{10} into -3_{10} with 4-bit Integers

With a 4-bit, 2's Complement, binary number, we represent 3 as

$0011_2 \rightarrow 0*(-2^3) + 0*2^2 + 1*2^1 + 1*2^0 = 3$

Conversely, we represent -3 with the same constraints as

$1101_2 \rightarrow 1*(-2^3) + 1*2^2 + 0*2^1 + 1*2^0 = -3$

To convert quickly and easily, ***flip the bits and add 1!***

```
3  = 0011 -flip-> 1100 -add1-> 1101 = -3
```


Comparison of Representations

Consider this 8 bit number: 0b11110000

Unsigned: $2^7 * 1 + 2^6 * 1 + 2^5 * 1 + 2^4 * 1 + 2^3 * 0 + 2^2 * 0 + 2^1 * 0 + 2^0 * 0$
$= 128 + 64 + 32 + 16 = 240$

Signed: $-2^7 * 1 + 2^6 * 1 + 2^5 * 1 + 2^4 * 1 + 2^3 * 0 + 2^2 * 0 + 2^1 * 0 + 2^0 * 0 =$
$-128 + 64 + 32 + 16 = -16$

Notice that $2^8 - 16 = 240$

To *sign extend* to a larger size, just copy the MSB:
0b1111111111110000 is -16 in 16-bit signed representation.

Addition in 2's Complement

Consider 8 + 4 = 12 = 0b1100 (unsigned 4-bit)

And, -8 + 4 = -4 = 0b1100 (signed 4-bit)

12 - (-4) = 2^4

Notice the identical binary values.

Also, - 1 = 0b1111111111111111 (signed 16-bit)

And, $2^{16} - 1$ = 65535 = 0b1111111111111111 (unsigned 16-bit)

This is why we use 2's complement! Addition is the same as normal binary

-1+1:
0b1111
+0b0001
→ 0b0000 (shift)

-3+2
0b1101
+0b0010
0b1111

Multiplying in Decimal (Project 1m)

In decimal, we multiply by "shifting"
each number as we go

Note the added 0 on the right when
multiplying the 2

We can perform the same operation
in binary to multiply

```
     104
x     26
     624
+   2080
    2704
```


Multiplying in Binary (Project 1m)

Since we always multiply by either
1 or 0, we either add to our running
sum, or don't.

We perform one "shift" by adding a
value to itself.

We get the value of each bit by
masking with almost all 0's, and
then checking if the result equals 0.

(Hint: Shift the mask also)

```
       1001
x      1011
       1001
      10010
     000000
+   1001000
    1100011
```

The Add instruction

The add instruction allows us to do some bit shifting:

`x + x == x * 2 == x << 1`

By doing this multiple times, we get multiplication:

`a * 2^b == a << b`

The Nor Instruction

NOR in LC2K is bitwise: First do OR with A | B, then NOT with ~result.

Recall from 270 or 203 that NOR is a universal gate; any gate can be built from it (AND, OR, etc.) https://en.wikipedia.org/wiki/NOR_logic

~A = ~(A | A) = NOR(A, A) Since A = A | A

A | B = ~(~(A | B)) = ~(NOR(A, B)) Double Negation

nor(nor(A,B), nor(A,B))

A & B = ~(~A | ~B) = NOR(~A, ~B) By DeMorgan's Laws

nor(nor(A,A), nor(B,B))

Combining Add and Nor

We can do subtraction by combining them:

A - B = A + (-B) = A + (~B + 1)

We can set or mask bitfields, like in Lab 1:

First shift with multiple add instructions, then do some nor operations.

How can you use this in your multiplication algorithm?

The Beq instruction

Beq always branches (sets PC) to *PC + **1** + offset.*

But if the offset is a label, it just branches to the label.

Review lec 4/5 & P1 Walkthrough for examples, and be comfortable with:

- If/Else statements
- While Loops
- Do-While Loops
- For Loops

The Beq instruction

Beq instructions implement conditional branches, but how do we do unconditional branches like in ARM?

When the 2 registers in beq are the same register, the branch is unconditional.

直接跳转 (if true)

Example: `beq 1 1 offset` is unconditional since r1 = r1 always.

Testing Project 1a

There are 2^{10} (add, nor) + $3*2^{22}$ (lw, sw, beq) + 2^6 (jalr) + 2 (noop, halt) = 12,584,002 possible LC2K instructions.

You *cannot* test all of these!

But, you can test every opcode, regA, regB, and destReg.

You should also test many types of offsets and .fill values:

- Numbers (16-bit 2's complement)
- Absolutely-resolved labels for lw/sw/.fill
- Relatively-resolved labels for beq

Problem 3: LC2K Assembler Test Cases

Take some time to write some test cases for Project 1a

Guidelines:

- Also submit corresponding machine code
- Must not have errors
- Must catch 3 bugs
- Each test must be 5 lines or fewer

Pro tip: Use to debug your Project 1a

LC2K to Machine Code Conversion Example

"five" is defined on line 8 (address 7) so memory[7] = 5

From spec.as

lw 0 1 five load reg1 with 5 (symbolic address)

I-type instructions (`lw`, `sw`, `beq`)	bits 24-22: `opcode` bits 21-19: `reg A` bits 18-16: `reg B` bits 15-0: `offsetField`

00000000100000010000000000000111
= **0x00810007**