CS2100

Jia Cheng

January 2021

1 Definitions

- MSB: Most significant bit
- LSB: Least significant bit

2 Conversion Table

Dec	Hex	Bin
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	a	1010
11	b	1011
12	c	1100
13	d	1101
14	e	1110
15	f	1111

3 Number representations

Note: The following terminology may be slightly confusing. $2s$ complement representation refers to the number representation format. $2s$ complement of a number N refers to how $-N$ is represented under the $2s$ complement representation.

Ditto for $1s$ complement representation and $1s$ complement of a number N.

3.1 Bs complement

Given a base B, positive integer n. The n-digit B's complement representation operates modulo B^n.

We now denote Bs as n-digit B's complement representation.

There are 3 interpretations of complement number systems.

1. The actual mathematical *object* O
2. The 2's complement *representation* R
3. The *value* N that we assign to the object

I will use $\mathbb{Z}/256\mathbb{Z}$ to illustrate. Denote $O_i = [i] = \{256k + i : k \in \mathbb{Z}\}$ for each $i \in \{0, \ldots, 255\}$.
Then fix any $i \in \{0, \ldots, 255\}$.
If $i \in \{0, \ldots, 127\}$, then value of O_i, $N(O_i) = i$.
If $i \in \{128, \ldots, 255\}$, then $N(O_i) = i - 256$.
Regardless of whether $i \in \{0, \ldots, 127\}$ or $i \in \{128, \ldots, 255\}$, the representation of O_i, $R(O_i)$ is just i.

From here onwards, if I use R, or N, it should be understood to mean $R(O_i)$, $N(O_i)$ respectively, where O_i is the relevant mathematical object. Note that a rigorous way to view R, N would be that they are mappings with domain $\mathbb{Z}/n\mathbb{Z}$.

In our previous example, $R : \mathbb{Z}/256\mathbb{Z} \to \{0, \ldots, 255\}$ and $N : \mathbb{Z}/256\mathbb{Z} \to \{0, \ldots, 127, -128, \ldots, -1\}$.

3.2 Fractional complements

3.2.1 (B-1)'s complement

Suppose a fractional number N has n integer digits and m fractional digits, then $(B-1)$'s complement of N is given by $B^n - B^{-m} - R$.

To see why, simply remove the decimal point by multiplying R by B^m (and then dividing B^m). Hence,

$$B^{-m}(B^{n+m} - 1 - B^m R) = B^n - B^{-m} - R$$

Suppose the *representation* of N is $R = (b_{n-1} \ldots b_0 b_{-1} \ldots b_{-m})$. If $b_{n-1} = 1$, then the *value* is $R - (B^n - B^{-m})$.

3.3 Sign extension to the left

3.3.1 Binary case

For positive numbers, i.e. numbers with the MSB 0, in both $2s$ and $1s$ complement, it suffices to pad the "left" of the number with 0's.

It is particularly easy to mentally prove the sign extension rules for $1s$, since for a negative number x, we simply invert it to get positive $-x$, apply the positive sign extension rules (pad with 0's), then invert it back.

Regardless, it is easy to prove the validity of the sign extension by doing summations.

For **both** 1s and 2s complement representations

- Sign extending number with 0 as MSB: Pad with 0's.
- Sign extending number with 1 as MSB: Pad with 1's.
- Note that the reason why I didn't say positive/negative here is because in $1s$ complement, there is a positive and a negative 0 value. Referring to the MSB would be the most general.

3.3.2 General case

3.4 Sign extension to the right

i.e. after the decimal point

Technically this is not called sign extension. But I don't know what is the technical term for this, so I'll just call this sign extension to the right for convenience.

1s complement (fractional)

- Sign extending number with 0 as MSB: Pad with 0's.
- Sign extending number with 1 as MSB: Pad with 1's.

2s complement (fractional)

- Pad with 0's regardless of MSB

3.5 Excess representation

Excess-n representation of a number x will be the binary representation of $x + n$.

Example Excess-127 representation of x is binary representation of $x+127$. The correspondence with $\mathbb{Z}/255\mathbb{Z}$ are as follows:
0 ... 127 128 ... 255 (mod 255)
0 ... 127 -128 ... -1 (2s complement)
-127 ... 0 1 ... 128 (excess-127)
-128 ... -1 0 ... 127 (excess-128)

4 MIPS Assembly

```
add (most R-type) $rd, $rs, $rt

sll $rd, $rt, shamt

addi (most I-type) $rt, $rs, immed

beq/bne $rs, $rt, immed # immed uses relative *word* addressing.
# Hence new PC = PC + 4 + immed * 4 if branch taken (PC uses *byte* addressing)

lw $rt, immed($rs) # loading Mem($rs + immed) into $rt
sw $rt, immed($rs) # storing $rt into Mem($rs + immed)
```

4.1 Uses of logical operators

With respect to manipulating (32-bit) binary sequences, there are some usual uses of logical operators.

1. `and` is used to force bits to 0.
    - Suppose bitmask is 00001101, then bits 1,4,5,6,7 are forced to 0. Another way of viewing this is that bits 0,2,3 are "extracted".
2. `or` is used to force bits to 1.
    - Suppose bitmask is 00001101, then bits 0, 2, 3 are forced to 1.

3. `xor`
    - Observe that $x \oplus 1 = x'$ and $x \oplus 0 = x$.
    - Hence, if bitmask is 00001101, then bits 0,2,3 are inverted and the other bits (1,4,5,6,7) remain the same.
    - Another application is this: If we know that bits 0,2,3 are of value 1 in a binary sequence B and we want to force those bits to 0, then we can xor B with bitmask 00001101. Of course, this can also be done with `and`.

4.2 Encoding space problem

The most general statement of this problem type is as follows: Given an ISA, with n types of instructions, type $T_1, \ldots, T_n$. Each instruction type T_i has an opcode of length l_i, with $i < j \implies l_i < l_j$. i.e. $l_1 < \cdots < l_n$.

Given that the encoding space is fully utilised and that there are at least 1 of each type of instructions encoded, what is the minimum and maximum number of instructions?

Define $l_0 = 0$.
Minimum

$$\sum_{1 \le i \le n-1} (2^{l_i - l_{i-1}} - 1) + 2^{l_n - l_{n-1}}$$

I will give an intuitive argument for the minimum case as I'm not too sure how to give a formal proof. We start by allocating space for the shortest instruction T_1. For the first l_1 bits, combinatorial analysis (product rule) tells us that there are 2^{l_1} "outcomes". Since all other instructions are longer than T_1, at least 1 of the first 2^{l_1} must be allocated for the other instructions of type T_2 to T_n. Since we want to minimise the total number of instructions, we will allocated exactly 1 of the first 2^{l_1} to the other instructions and $2^{l_1} - 1$ to instruction type T_1.
We repeat this argument for T_2, allocating $2^{l_2 - l_1} - 1$ of the next $2^{l_2 - l_1}$ outcomes (generated by $l_2 - l_1$ bits) to instructions of type T_2. And so on.
Finally, we reach the final instruction type T_n. Then the last $l_n - l_{n-1}$ bits generate $2^{l_n - l_{n-1}}$ outcomes, which we will all allocate to T_n.

Maximum

$$(n-1) + 2^{l_n} - \sum_{1 \le i \le n-1} 2^{l_n - l_i}$$

Argument: For $1 \le i \le n$, define S_i as the set of encoding allocated to instruction type T_i. By problem definition, $|S_i| \ge 1$. So obviously, to maximise $\sum_{1 \le i \le n} |S_i|$, our strategy is to minimise $|S_i|, 1 \le i < n$ so as to maximise $|S_n|$. In particular, we find a way to let $|S_i| = 1, \forall 1 \le i < n$.
Here we introduce the notion of making encoding "*unavailable*". For instance, when we allocate l_i bits to an instruction of type T_i, $i < n$, the remaining $n - i$ bits are "wasted", that is, they are made *unavailable*. So, in a certain sense, we lose $2^{l_n - l_i}$ encodings.

We know that if $\forall 1 \le i, j < n$(WLOG $i \le j$), then any instruction of type T_i cannot have the same first i bits as an instruction of type T_j. This implies that the encodings made *unavailable* by any 2 such instructions are disjoint.
To give a concrete example, suppose we have $n = 3$, $l_1 = 2, l_2 = 4, l_3 = 5$.
Let $(0, 0)$ be an encoding allocated for instruction A of type T_1. Let $(0, 1, 0, 0)$ be an encoding allocated for instruction B of type T_2.

Then the encoding made *unavailable* by A is $\{(0,0,0,0,0),\ldots,(0,0,1,1,1)\}$. The encoding made *unavailable* by B is $\{(0,1,0,0,0),(0,1,0,0,1)\}$. Notice how these 2 sets are disjoint since no 2 entries across the 2 respective sets can share the first $2 = l_1$ bits.

This idea of disjoint sets gives the intuition for why we can keep on subtracting $2^{l_n - l_i}$, overall subtracting away $\sum_{1 \leq i \leq n-1} 2^{l_n - l_i}$.

Note that the first $(n-1)$ refers to $\sum_{1 \leq i < n} |S_i|$, where $\forall 1 \leq i < n, |S_i| = 1$. That is, we only allocate 1 encoding for each instruction of type $T_1, \ldots, T_{n-1}$.

5 Boolean Algebra

5.1 Minterms and maxterms

On n variables $x_1, \ldots, x_n$, define the minterms and maxterms as follows:
Consider $m[x]$, where $x \in \{0, \ldots, 2^n - 1\}$, $x = (b_1 \ldots b_n)_2$.
Then for each i in $\{1, \ldots, n\}$, we have the literal x_i if $b_i = 1$, x_i' if $b_i = 0$.
Another way to remember this is that the product of literals forming $m[x]$ equals 1 when each b_i is substituted into x_i / x_i'.

Consider $M[x]$, where $x \in \{0, \ldots, 2^n - 1\}$, $x = (b_1 \ldots b_n)_2$.
Then for each i in $\{1, \ldots, n\}$, we have the literal x_i if $b_i = 0$, x_i' if $b_i = 1$.
Another way to remember this is that the sum of literals forming $M[x]$ equals 0 when each b_i is substituted into x_i / x_i'.

In short, when it comes to minterms, negate the 0's, when it comes to maxterms, negate the 1's