Assignment 1 Report

- by ridoger, Haoyu Ren 124090521

1 Addition

1.1 Method

1.1.1 An naive way

Naturally we notice that if both a and b have a 1 at the same bit position, then a & b will also have a 1 at that bit, which indicates a carry is generated at that position; if we ignore the carry, then each bit of $a + b$ is 1 if and only if exactly one of a or b has a 1 at that bit. Thus we obtain

$$a + b = a_1 + b_1, \text{where } a_1 = a \wedge b << 1, b_1 = a \oplus b.$$

Note that since left-shifting fills the vacated bits with zeros, the least significant bit a_1 of a, equals to $(a \wedge b) << 1$, is always 0. Consequently, the two lowest-order bits of $(a_1 \wedge b_1) << 1$ will both be 0. Every time we repeat this process, one more 0 will be generated at the right side. Hence after 32 repeats, we finally get $a_{32} = 0$. We conclude that one of the approaches to implement `add()` just need to do the operation above for 32 times.

However this approach is slow and code looks ugly while loops are forbiddened. Inspired by the thought of Carry-Lookahead Adder[CLA], I implemented another approach (although it might be more proper to call my `add()` as Parallel Prefix Adder[PPA]).

1.1.2 My approach

Define carry propagation function $P_0(a, b) = a \oplus b$ and carry generation function $G_0(a, b) = a \wedge b$. Denote the bits of P_0 and G_0 by $p_0^i, g_0^i, i = 1, 2, ..., 32$ where p_0^1, g_0^1 are the right end of P_0, G_0. Note that a carry $g_0^i = 1$ triggers another carry $\Leftrightarrow g_0^i \wedge p_0^{i+1} = 1$.

We define that a sequence of 1 $S = \left\{p_k^i = 1, p_k^{i+1} = 1, ..., p_k^{i+j} = 1\right\} \subseteq P_k$ is a *path*, if $\forall g_k^i \in S$, we have either $g_0^i = 0$ or g_0^i is the right end of the sequence. We call $|S|$ the *length* of the path. Then, all carries that don't trigger another carry will be represented by a path with length 1.

```
G |= (G << 1) & P;
P &= P << 1;
G |= (G << 2) & P;
```

Meaning of the first line is similar to the operation in **Section 1.1.1**. By using $P_1 = P_0 \wedge (P_0 << 1)$ we can find end points for all possible propagation with len = 2, then we use $G_2 = G_1 \vee [(G_1 << 2) \wedge P_1]$ to expand the length of paths in G_1 if possible. **Here comes some different situations after left-shifting**:

a. **A path $\left\{g_1^i\right\}$ with length 1 in G_1.**

   We want to prove that those paths didn't expand in last step will not expand in this step.

   The only bit of G_2 that may be affected by g_1^i is g_2^{i+2}.

   $g_1^{i+1} = g_0^{i+1} \vee \left(g_0^i \wedge p_0^{i+1}\right) = 0 \quad \Longrightarrow \quad g_0^{i+1} = p_0^{i+1} = 0,$ then we have

$$
\begin{aligned}
g_2^{i+2} &= g_1^{i+2} \vee \left(g_1^i \wedge p_1^{i+2}\right) \\
&= \left[g_0^{i+2} \vee \left(g_0^{i+1} \wedge p_0^{i+2}\right)\right] \vee \left[g_1^i \wedge \left(p_0^{i+2} \wedge p_0^{i+1}\right)\right] \\
&= \left[g_0^{i+2} \vee \left(0 \wedge p_0^{i+2}\right)\right] \vee \left[1 \wedge \left(p_0^{i+2} \wedge 0\right)\right] \\
&= \left(g_0^{i+2} \vee 0\right) \vee 0 \\
&= \ g_0^{i+2}
\end{aligned}
$$

We showed that the value of g_2^{i+2} is regardless of g_1^i. Therefore, this path will be moved to G_2 while remaining unchanged without any side effect.

b. **A path $\left\{g_1^i, g_1^{i+1}\right\}$ with length 2.**

Note that $p_0^{i+1} = 1$ since $g_1^{i+1} = g_0^{i+1} \vee \left(g_0^i \wedge p_0^{i+1}\right) = 1$ and $g_0^{i+1} = 0$ by the definition.

If $g_1^{i+1} \wedge p_1^{i+3} = p_1^{i+3} = p_0^{i+3} \wedge p_0^{i+2} = 1$, then $p_0^{i+2} = 1$, which implies $p_1^{i+2} = p_0^{i+2} \wedge p_0^{i+1} = 1$. Hence $g_2^{i+3} = g_1^{i+3} \vee \left(g_1^{i+1} \wedge p_1^{i+3}\right) = g_1^{i+3} \vee (1 \wedge 1) = 1$. This is to say, a path must remain conjunctive after expansion. Thus a carry can propagate throughout the path without causing bugs.

Finally we point that a path expands iff $p_0^{i+2} = 1$, or both p_0^{i+2} and p_0^{i+3} equals 1, which is equivilant to say a carry can propagate step by step like what we did repeatedly in **Section 1.1.1**.

We repeat this process 5 times to find all path for propagation. Correctness of each step can be prove by following the idea above. The largest length can a path expand in a single step is 1, 2, 4, 8, 16, repectively. Since a path will keep expanding in each step utill reaching its maximum length, this 5 steps can expands the length of the path by up to $31 = 1 + 2 + 4 + 8 + 16$. When computing additon of 2 `int32_t` type integers, a carry can propagate at most 31 bits far, so 5 steps are enough for us.

Now we have G_5 that contains all paths with their maximum length. For each path, it terminates at g_5^i iff either another path begins here, or $p_4^{i+1} = 0 \Leftrightarrow p_0^{i+1} = 0$. Also, if a bit of a path $g_5^k = 1$, then $p_0^{k+1} = 1$. Hence only need to compute $P_0 \oplus (G_5 << 1)$ will we get the result of $a + b$ (recall that $P_0 = a \oplus b$).

Complete implementation:

```
int32_t add(int32_t a, int32_t b) {

int32_t P = a ^ b;
int32_t G = a & b;

int32_t temp_P = P;

G |= (G << 1) & P;
P &= P << 1;
G |= (G << 2) & P;
P &= P << 2;
G |= (G << 4) & P;
P &= P << 4;
G |= (G << 8) & P;
P &= P << 8;
G |= (G << 16) & P;

return temp_P ^ (G << 1);

}
```

1.2 Time Complexity

$O(\log n)$, where n is the number of bits.

1.3 Difficulties I met

Almost every step to make this function run appeared to be a big challenge for me. After implemented the method in **Section 1.1.1**, I just felt unpleasant about its inefficiency and ugliness. When I asked AI for help, none of them offered a better solution.

Don't know whether I should say it lucky or not, my roommate used to be an Oier. I asked him if he had some advice, and he told me that carries can propagate "exponentially". That's how my self-torture began. I googled to find what he means and found the existence of CLA. Wow. That concept was way ahead of its time for me, and it took me some time to understand what it is.

The clearest blog I found about CLA which also spent the most of my time to learn the thought of CLA

However, eventually it turned out that my code would only become uglier instead becoming of more elegant if I just simply copy the mechanism of CLA. But AI started to say some useful after I prompt them with "CLA". Chat-GPT gave a version of implementation that pretty close to the final version I used, except for gpt didn't update the value of P after each step. It worked well on all unit tests except the 2 random number addition. I spent another several hours to check the binary forms of the numbers to find why the function failed on this two. There's no doubt that I found nothing. Those binary numbers are not for human beings.

Unfortunately, I thought the codes gpt gave was junk since gpt couldn't tell how his code works. So I deleted both code and the dialogue with gpt. Hence I am not able to post it here.

After I gave up, then my roommate asked me to show him the code gpt gave, and pointed out the problem. I corrected it by accident. It runned, but I didn't know why it runned. Deepseek said nothing useful about it. Finally I spent 2 days to figure that out and learn how to write it down by typst. These shall be all difficulties I met when implementing `add()`.

1.4 Credit

- My roommate Heng Zhang who provided essential idea of the implementation.
- 一次型解决超前进位加法器——32 位 CLA 的实现 —— 叶逸昇

int32_t P = a ^ b;
int32_t G = a & b;

int32_t temp_P = P;

G |= (G << 1) & P;
P &= P << 1;
G |= (G << 2) & P;
P &= P << 2;
G |= (G << 4) & P;
P &= P << 4;
G |= (G << 8) & P;
P &= P << 8;
G |= (G << 16) & P;

return temp_P ^ (G << 1);
这段代码是怎么实现加法的

已思考（用时 37 秒）

这段代码实现的是**并行前缀加法器（Parallel Prefix Adder）**中的一种优化算法，用于高效计算两个32位整数的加法。它使用了**Kogge-Stone**或类似的前缀树结构。让我详细解释：

基本思想

1. **P（Propagate）**：`a ^ b` - 当某一位需要传播进位时（即a和b中只有一个为1）
2. **G（Generate）**：`a & b` - 当某一位会生成进位时（即a和b都为1）

算法步骤

1. 初始化

```c
int32_t P = a ^ b;    // 传播位
int32_t G = a & b;    // 生成位
int32_t temp_P = P;   // 保存初始P用于最终计算
```

2. 并行前缀计算（关键部分）

这个循环展开的部分是在**并行计算所有位的进位**：

这个循环展开的部分是在**并行计算所有位的进位**：

```c
// 第1轮：计算相邻2位的进位传播
G |= (G << 1) & P;  // 合并相邻的进位生成
P &= P << 1;        // 更新进位传播链

// 第2轮：计算相邻4位的进位传播
G |= (G << 2) & P;
P &= P << 2;

// 第3轮：计算相邻8位的进位传播
G |= (G << 4) & P;
P &= P << 4;

// 第4轮：计算相邻16位的进位传播
G |= (G << 8) & P;
P &= P << 8;

// 第5轮：计算相邻32位的进位传播
G |= (G << 16) & P;
// 最后不需要再更新P，因为已经覆盖全部32位
```

3. 计算结果

```c
return temp_P ^ (G << 1);
```

- `G << 1`：将生成的进位向左移动一位，得到每个位的实际进位
- `temp_P ^ (G << 1)`：初始的传播位与进位相加，得到最终和

举例说明

假设计算 `a = 5 (0101)` 和 `b = 3 (0011)`：

1. 初始：
   - `P = 0101 ^ 0011 = 0110`
   - `G = 0101 & 0011 = 0001`
   - `temp_P = 0110`
2. 经过并行前缀计算后：
   - G会计算出所有进位：最终 `G = 0011`
3. 最终结果：
   - `G << 1 = 0110`
   - `temp_P ^ (G << 1) = 0110 ^ 0110 = 0000`
   - 加上进位：`1000` (8)

时间复杂度

- 传统逐位进位：O(n) 线性时间
- 这种并行前缀方法：O(log n) 对数时间
- 对于32位，只需要5步（$\log_2 32 = 5$）

优点

1. **并行计算**：所有位的进位可以同时计算
2. **适合硬件实现**：在芯片设计中常用
3. **常数时间**：无论输入如何，步骤固定

这种算法是计算机体系结构中**算术逻辑单元(ALU)**实现快速加法的重要技术之一。

2 Subtraction

2.1 Method

One group only need one operation. There's no need for us to implement addition again. The only thing we need to do is to find the additive inverse of b, which is $1 + \sim b$ in computer.

```
return add(a, add(~b, 1));
```

2.2 Time Complexity

$O(\log n)$, the same with `add()`.

2.3 Difficulties

None.

2.4 Credit

Should be none. Maybe my algebra textbook should be mentioned? Then

Kostrikin, A. I. 2007. 代数学引论（第 1 卷：基础代数）. Translated by 张英伯. 2nd ed. Beijing: Higher Education Press. ISBN 978-7-04-020525-1

3 Multiplication

3.1 Method

Denote a by $\overline{a_1 a_2 ... a_{32}}$, where a_i is the i-th bit of a from right to left. Then

$$ab = \sum_{i=0}^{32} 2^{i-1} a_i b$$

We traversal through bits of a. If $a_i = 1$, left_shift b by i bits then add it to the result. After traversal is done, the sum will be the product.

3.2 Time Complexity

For each bit, an additional shift may occur; for each shift, an addition with complexity $O(\log n)$ is done to sum the result. Hence the complexity is $O(n \log n)$.

3.3 Difficulties

None

3.4 Credit

None

4 Division

4.1 Method

Naturally come up with the idea of long division, which is the method we use in daily life.

First, take the absolute value $|a|, |b|$ for convenience. Then try to find the maximal i that $(b << i) \leq a$. We do this by checking the sign of the difference between a and the multiplier. Since left-shifting b may lead to overflow and cause unexpected effect (e.g. $(2^{31} - 1) = -2 < 0$), we right-shift a and check the sign of $(a >> i) - b$ instead.

Every time we find the multiplier $(b << i)$ that smaller than a, we let `a = a - b << i` and plus 1 << i to the quotient. At last we get $a < b$ and the long division terminates. After restoring the sign of quotient, we return the result.

4.2 Time Complexity

n subtraction with complexity $O(\log n)$ is applied to check whether the multiplier is greater than the remainder, thus the complexity is $O(n \log n)$.

4.3 Difficultis

I had forgotten that signed integer uses arithmetic shift. That caused some chaos when checking the sign of differences. gpt pointed it out for me. After all, I didn't met any big problems.

4.4 Credit

```
int32_t divide(int32_t a, int32_t b) {

    int32_t result = 0;

    // Check for sign of quotient
    int32_t minus = a >> 31 ^ b >> 31;

    // Take absolute values
    if (a >> 31 & 1) a = add(~a, 1);
    if (b >> 31 & 1) b = add(~b, 1);

    for (int32_t i = 0; i < 32; ++i) {

        // Check whether divisor's multiplier > dividend
        // In case of overflow, we shrink the dividend instead of
expanding the divisor
        if (subtract(a >> subtract(31, i), b) >> 31 ^ 1) {
            a = subtract(a, b << subtract(31, i));    // Reminder
            result |= 1 << subtract(31, i);    // Add corresponding
value to quotient
        }

    }

    // Restore sign of quotient
    minus && (result = add(~result, 1));

    return result;

}
```

不要多管闲事，这个程序和常见实现有什么区别跟你没关系，现在告诉我为什么程序只会输出1或-1，别管其他任何事，只要告诉我为什么会这样

已思考 16s ›

问题的根本原因在于循环里的 `if` 条件永远为真，导致每一位都被设成 1，最后 `result` 变为全 1（`0xFFFFFFFF`），然后最后那条用 `&&` 做符号恢复的语句把它翻转成 `1` 或保留为 `-1`。下面逐步说明为什么会这样：

1. `minus` 的值是这样计算的：

```c
int32_t minus = a >> 31 ^ b >> 31;
```

因为 `a >> 31` 与 `b >> 31` 是算术右移（signed right shift），它们的值不是 0/1，而是 `0`（当数非负）或 `-1`（当数为负，二进制为 `0xFFFFFFFF`）。所以 `minus` 的可能值是 `0` 或 `-1`（不是布尔 0/1）。

2. 关键的 if 条件是：

```c
if (subtract(a >> subtract(31, i), b) >> 31 ^ 1) { ... }
```

按 C 的运算符优先级，这等价于：

```c
if ( (subtract(a >> (31 - i), b) >> 31) ^ 1 ) { ... }
```

现在看两种情况：

- 如果 `subtract(...) >= 0`，则 `(subtract(...) >> 31)` 为 `0`，`0 ^ 1 == 1` → 非零 → 条件为真。
- 如果 `subtract(...) < 0`，则 `(subtract(...) >> 31)` 为 `-1`（即 `0xFFFFFFFF`），`-1 ^ 1 == 0xFFFFFFFE`（等于 -2）→ 仍然非零 → 条件也为真。

因此不论 `subtract(...)` 是正还是负，`( ... >> 31) ^ 1` 都是非零，`if` 分支**始终进入**。

3. 由于 `if` 总是成立，循环里会对每一位都执行：

```c
result |= 1 << (31 - i);
```

因而最终 `result` 的所有位都被置为 1，即 `result == 0xFFFFFFFF`（作为 `int32_t` 是 `-1`）。

4. 最后的符号恢复语句：

```c
minus && (result = add(~result, 1));
```

这里 `minus` 为 `0` 或 `-1`。在 C 中 `&&` 的左侧被当作布尔判断，任何非零值被视为真；所以当 `minus == -1`（即原被除数与除数符号不同）时，右侧被执行，计算 `result = add(~result,1)`：对 `0xFFFFFFFF` 取反得 `0x00000000`，加 1 得 `0x00000001`（即 `1`）。当 `minus == 0`（符号相同）时，右侧不执行，`result` 保持 `0xFFFFFFFF`（即 `-1`）。

5. 结论（对应你观察到的行为）：
   - 若被除数与除数符号相同，函数返回 `-1`（`0xFFFFFFFF`）。
   - 若符号相反，函数返回 `1`（`0x00000001`）。

简而言之：`if` 的判断写法逻辑上是想检测"非负"/"负"，但由于算术右移对负数产生 `-1`（全 1），再与 `1` 做 XOR 后依然得到非零值，导致条件永真，从而把商的每一位都置为 1，最后只剩 `-1` 或其取反 `1` 两种结果。

5 Modulo

5.1 Method

Note that $a\%b = a - (a/b) * b$, we can call the function already implemented.

5.2 Time Complexity

`subtract()`, `multiply()`, `divide()` are called once respectively. The time complexity is $O(\log n) + O(n \log n) + O(n \log n) = O(n \log n)$

5.3 Difficulties

None

5.4 Credit

None

6 Thoughts about unit testing

I'm not sure what should be include in this part, so I'll just say something random.

In my opinion, unit testing is an important and useful tool for developers to find out bugs in their programs. It helps saving time and enhance the robustness. It is widely used in IT companies, and I think it is benefitial for every developer to master this skill.