3.2. 整数加减法

3.2.1. 补码

计算机里的二进制串被读取时可能会被视为补码，关于补码的知识，可以参考《数字电路与逻辑设计》。重要的是二进制补码与十进制的转换，设 n 位二进制补码$x_{n-1}x_{n-2}\cdots x_1x_0$，则其十进制数为：

$$(-x_{n-1}2^{n-1}+x_{n-2}2^{n-2}+\cdots+x_12^1+x_02^0)_{10}=(x_{n-1}x_{n-2}\cdots x_1x_0)_2 \qquad 2.1$$

另外可以快速求二进制补码数（无论正负均适用）的相反数：将该数按位取反后再加 1。比如，计算一个负数二进制补码对应的十进制数，可以先求其相反数，再添加负号，避免计算过多的 2 的幂。

补码的加减法

因为减去一个数等于加上一个数的相反数，所以现只考虑补码的加法，分为：正数+正数，正数+负数和负数+负数。

正数的补码等于其原码；负数是对应正数的相反数，将该正数按位取反后加 1 得到负数的补码（注意不要和反码弄混了）。然后进行补码的加法运算（一般二进制加法），最后舍去最高位的进位，结果保留操作数的原长即可。

此外，也可以将补码转化为十进制数进行运算，二者互为检验。

3.2.2. 整数范围

MIPS 的 32 位操作数本身只是一个二进制串，只是被读取时会被看做无符号数和~~补码~~两种情况（一般情况下，有些指令会执行其他的读取方法）。

当 32 位 01 串被看做无符号数时，其表示的整数范围为$[0,2^{32}-1]$。

当 32 位 01 串被看做二进制补码时，其表示的整数范围为$[-2^{31},2^{31}-1]$。其中，最大正整数为 0x7FFF FFFF=$2^{31}-1$，最小负整数为 0x8000 0000=-2^{31}。

符号扩展

所谓二进制数的符号扩展，就是用更多的位数表示同一个数。对于补码正数或无符号数：用 0 进行扩展；对于补码负数：用 1 进行扩展。

MIPS 的一些指令，比如立即数指令，指令中会含有小于 32 位的二进制数，在对这些数进行操作前，须将其符号扩展到 32 位后再进行操作。

3.2.3. 溢出

无论 32 位 01 串被读取成什么形式的二进制数，计算机只会对其进行一般的二进制加减法。当计算结果超出 32 位 01 串可表示的整数范围时，便会产生溢出。

补码操作的溢出

MIPS 的非无符号数算术操作（add, sub, addi）将操作数看做二进制补码。

正数+正数，正数+负数和负数+负数的三种加法情况中，“正数+负数”永远不会产生溢出（都是使原真值减小）。“正数+正数”会产生上溢（overflow），表现正数和超出可以表示的整数范围变为负数；“负数+负数”会产生下溢（underflow），表现为负数和超出可以表示的整数范围变为正数：上溢和下溢统称为溢出（overflow），表现为符号的改变。

补码操作溢出时会产生“异常（exception）”，也叫作中断（interrupt）。

饱和（saturating）操作处理二进制补码溢出的情况时，结果会被设置为最大正数（正数+正数）或最小负数（负数+负数），一般更适合多媒体操作。

无符号数的溢出

MIPS 的无符号数算术操作（addu, subu, addiu）将操作数看做无符号数。

无符号数通常用于表示内存地址，因此这种情况下的溢出可以忽略，不会产生“异常”。

3.2.3.1. 溢出测试

MIPS 在溢出时会产生异常，但没有测试溢出的条件分支。可以通过一个 MIPS 指令序列来检测溢出，以有符号加法为例：

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor $t3, $t1, $t2 # Check if adders' signs differ
slt $t3, $t3, $zero # $t3 = 1 if signs differ
bne $t3, $zero, No_overflow # 正数+负数永远不会溢出

# 指令可以运行下一部分表示加数符号相同
xor $t3, $t0, $t1 # Check if sum and adders' signs differ
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow
```

因为 add 在溢出时会产生异常，所以使用 addu 来避免异常的产生，从而可以进行溢出检测指令序列的执行，而非跳转到处理异常的指令序列。

同样，对于无符号数的操作，我们也可以进行溢出测试，以无符号加法为例：

```
addu $t0, $t1, $t2 # $t0 = sum
nor $t3, $t1, $zero # $t3(unsigned) = NOT $t1 = 2^(32) - $t1 - 1
sltu $t3, $t3, $t2 # 2^(32) - 1 < $t1 + $t2 则置位
bne $t3,$zero,Overflow
```

Note

因为$A + \overline{A} = 1$，所以对于 n 位二进制串 a 有$a + \bar{a} = 1 \cdots 1(\text{n bits}) = 2^n - 1$。

3.2.3.2. 溢出异常

从本质上说，异常是一种过程调用，不过是在计划之外。

当某条指令产生溢出时，该指令的地址保存在一个寄存器中，MIPS 使用异常程序计数器（Exception Program Counter, EPC）来保存致使异常的指令地址。然后计算机会跳到一个预先设定好的地址去执行相应的异常处理指令序列（程序）。在某些条件下，异常处理程序执行完毕后，希望能够返回原程序继续执行，MIPS 使用指令 mfc0（move from system control）来将 EPC 中的地址存入一个通用寄存器，从而可以在执行完异常处理程序后通过寄存器跳转指令（jr）返回原程序继续执行。

由于返回原程序时，所有寄存器必须恢复到（可以通过栈操作）进入异常处理程序前的状态才能继续执行原程序，所以会产生一种困境：如果先恢复所有寄存器，则保存从 EPC 复制的地址的那个通用寄存器就会被破坏；如果恢复时保留那个存放放回地址的寄存器不变，虽然可以进行正确跳转，但意味着在程序执行的任何时刻，异常会导致一个寄存器的值无法被恢复。为了解决这一困境，MIPS 将寄存器$k0 和$k1 预留给操作系统，这些寄存器在异常时不会被恢复。

3.2.4. 算术逻辑单元（Arithmetic Logic Unit, ALU）

ALU 用来执行算术运算和逻辑运算。

ALU 的符号图如 Figure 3-4 所示。注意 ALU 的符号也可以表示加法器（Adder），在符号图上标明 ALU 和 Adder 加以区分。

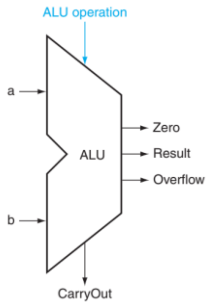

Figure 3-4 The Symbol Commonly Used to Represent an ALU

其中 ALU operation 是一个由 3 条控制线结合在一起的功能选择器，其取值与 ALU 对应的操作如 Figure 3-5 所示；Result 是输入 a 和 b 经 ALU 运算后的结果，均为 32 位；Zero 是 0 检测器，当 ALU 输出为全 0 时为 1；Overflow 输出溢出检测的结果，计算溢出时为 1。

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Figure 3-5 Values of ALU Control Lines Corresponding ALU Operations

一种 MIPS 的 32 位 ALU 的实现结构如所示。

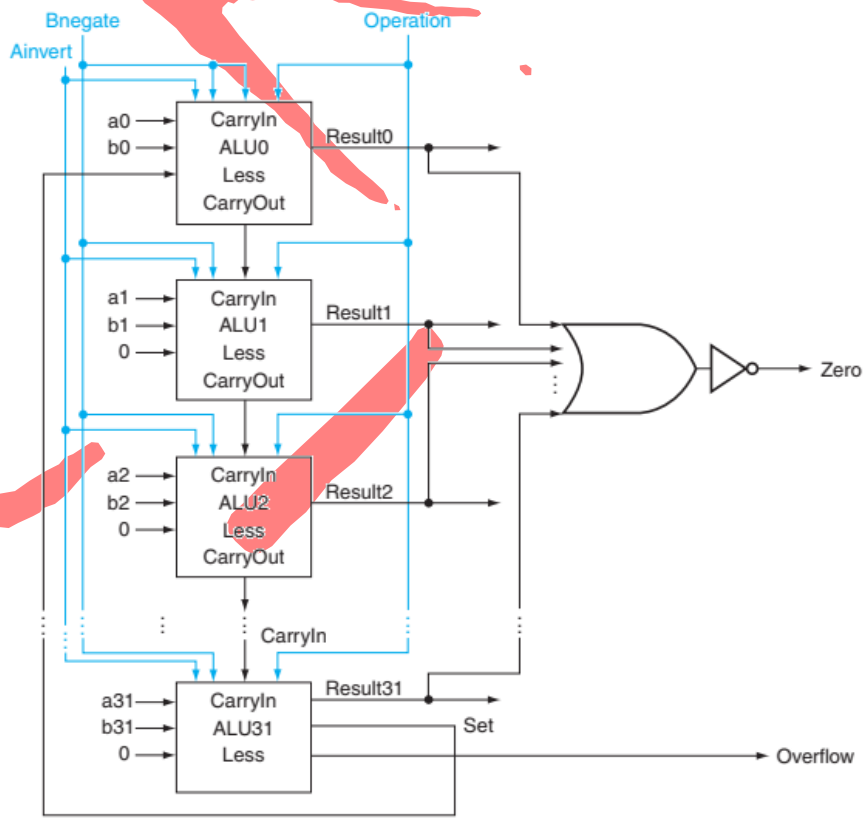

Figure 3-6 A 32-bit ALU Constructed from the 31 Copies of the 1-bit ALU

Ainvert（1 位），Bnegate（1 位）和 Operation（2 位）是 ALU 的 3 条控制线，用于控制三个数据选择器。其中 Ainvert 负责选择输入 a 还是$\bar{a}$，Bnegate 负责选择输入 b 还是$\bar{b}$，Operation 用于选择输入经过一系列门电路产生的不同结果作为最终结果输出。

例如减法的功能码为 0110，三条控制线的取值为 Ainvert（0），Bnegate（1）和 Operation（10）。Ainvert（0）和 Bnegate（1）表示取输入为 a 和$\bar{b}$，Operation（10）表示在 ALU 内部选择 a+$\bar{b}$（a-b）的结果进行输出。

Set 是用于指示小于则置位的比较结果，小于的比较可以通过 a-b 的结果与 0 进行比较实现。

行波进位（Ripple Carry）与超前进位（Lookahead Carry）

Figure 3-6 所示组织方式的 32 位 ALU 被称为行波进位加法器，意味着每一级的运算必须等待上一级运算产生的进位信号传入后方可进行，延迟高。

超前进位根据每一级的运算法则，其中a和b为输入，s为和，c为进位信号：

$$s = a \oplus b \oplus c_{in}$$

$$c_{out} = ab + c_{in}(a + b) \tag{3.1}$$

通过代入、展开等方法，表示出所有级加法器最终输出结果的逻辑表达式，用空间的复杂（超前进位的逻辑电路比行波进位复杂得多）换取时间上的低延迟。

定义进位传播（propagate）函数$p = a + b$，以及进位生成函数（generate）$g = ab$。则对于n位 LCA（lookahead carry adder），进位公式将重新书写如下：

$$\begin{cases} s_i = a_i \oplus b_i \oplus c_{i-1}, & i = 1, \cdots, n \\ c_i = g_i + c_{i-1} p_i, & i = 1, \cdots, n \\ c_0 = c_{in} \\ c_{out} = c_n \end{cases} \tag{3.2}$$

例如，一个 4 bit 的 LCA 进位信号的逻辑函数式为：

$$\begin{cases} c_1 = G_0 + c_0 P_0 \\ c_2 = G_1 + c_1 P_1 = G_1 + (G_0 + c_0 P_0) P_1 = G_1 + G_0 P_1 + c_0 P_1 P_0 \\ c_3 = G_2 + c_2 P_2 = G_2 + (G_1 + G_0 P_1 + c_0 P_1 P_0) P_2 = G_2 + G_1 P_2 + G_0 P_2 P_1 + c_0 P_2 P_1 P_0 \\ c_4 = G_3 + c_3 P_3 = G_3 + (G_2 + G_1 P_2 + G_0 P_2 P_1 + c_0 P_2 P_1 P_0) P_3 = G_3 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1 + c_0 P_3 P_2 P_1 P_0 \end{cases}$$

可以看出其逻辑电路非常复杂，但门延迟降低了。

一种方案是使用 4 位组 LCA 单元构成的 n 位超前进位加法器，其主要思想是进行级间抽象，Figure 3-7 是一个 16 位的 4 位组超前进位加法器。

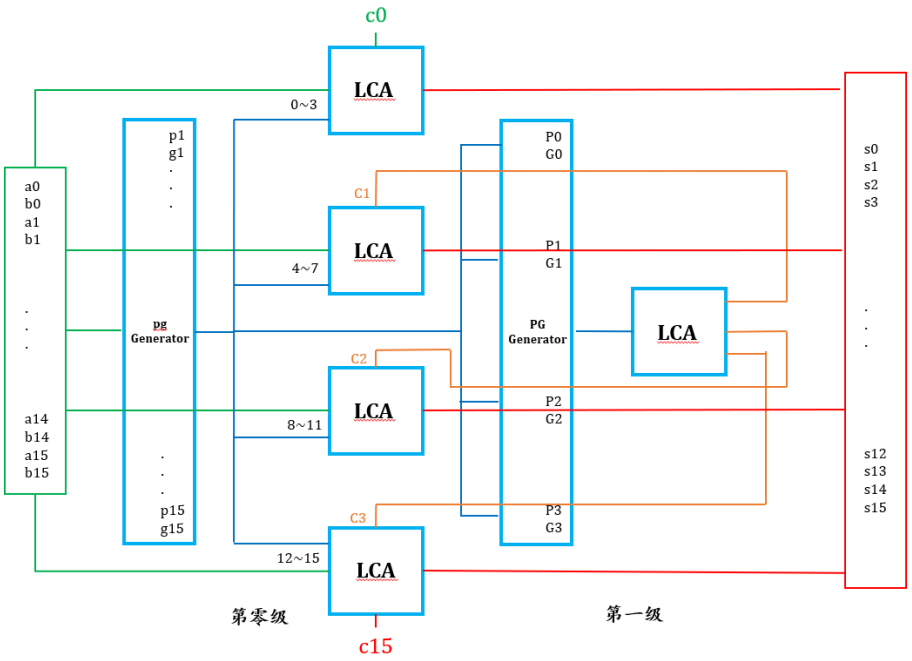

Figure 3-7 Four 4-bit ALUs Using Carry Lookahead to Form a 16-bit Adder

最开始，每 4 位构成一个 LCA，组成第零级的 n/4 个 LCA，需要 n/4 个进位信号来完成运算。第零级每 4 个 LCA 抽象成第 1 级的一个 LCA 单位，以此类推，直到第$\log_4 n$-1 级只有一个 LCA 单位（逻辑块）；如果操作数长度不是 4 的幂，则将其进行符号扩展。

pg 生成器利用输入的操作数生成所有的p_i和g_i，一是用于计算最终结果，而是用于计算下一级的 P 和 G。从第一级开始，PG 生成器是由下一级的 P 和 G（初

始条件为p_i和g_i）用来计算本级 P 和 G 的逻辑电路，而第一级可以利用本级的 P 和 G 返回第零级的进位信号。

除了第零级的 LCA，其他级的 LCA 只是抽象概念，并不执行完整的计算，实际上每一级只需要运行 PG 发生器即可，而将 LCA 的进位信号计算推迟到最后并行计算。第零级需要的进位信号依赖于整个加法器的进位输入，以及第一级的 P 和 G 计算出的进位信号。第 1 级的 P 和 G 依赖于第二级的 P 和 G，以此类推，第$\log_4 n$-1 级依赖于 pg 发生器计算的p_i和g_i。这种依赖和逻辑关系，用 Figure 3-8 表示为：

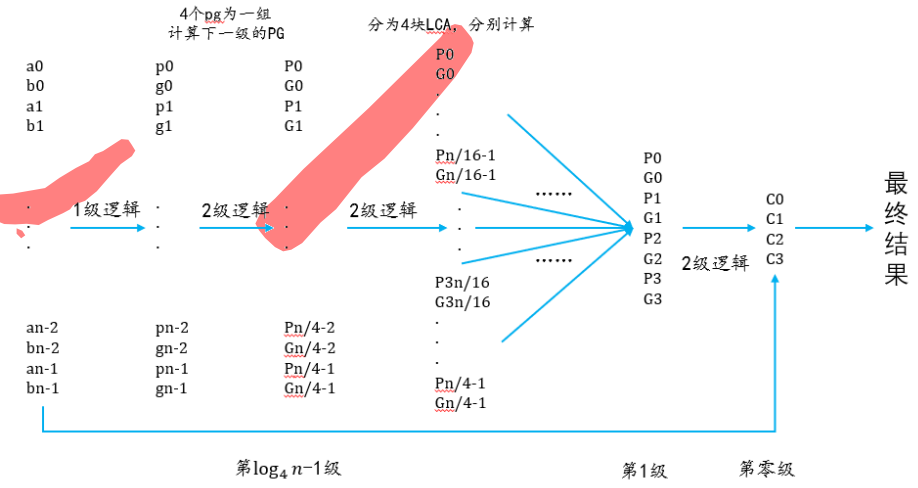

Figure 3-8 The Dependent and Logical Relation between Levels

Example 3.1

在 16 位和 32 位加法器两种情况下，比较行波进位加法器和超前进位加法器从最低位进位输入信号到最高位进位输出信号，路径上门延迟的数量。假设通过每个与门、或门和异或门的延迟相同。

行波进位加法器：

每一级（1 bit）单位加法器需要两个门延迟（c 都可以用析取范式表示，即先经过几个与门，与门的结果再经过或门）

16（32）位行波进位加法器共 16（32）级，共有 32（64）个门延迟

超前进位加法器：

32 不是 4 的幂，符号扩展为 64 位

16（32）位超前进位加法器共$\lceil \log_4 n \rceil$级，共有$2\lceil \log_4 n \rceil + 1$=5（7）个门延迟

Note

如 Figure 3-8 所示，除了第零级外，每一级的运算实际上只是一个 PG 发生器 PG 发生器可以写成析取范式，需要 2 个门延迟。

另外第一级运算出第零级的进位信号也需要 2 个门延迟，所以每一级运算共经过 2 个门延迟。

第一级的 pg 发生器同时计算p_i和g_i，经历 1 个门延迟。

所以 n 位 4 位组超前进位加法器从最低位进位输入到最高位进位输出之间的门延迟个数为$2\lceil \log_4 n \rceil + 1$。