05-整数运算

ALU

- 算术逻辑单元：完成数据算术逻辑运算的部件
- ALU会根据运算结果设置Flags，存入寄存器
- Control Unit控制ALU操作和数据出入ALU的信号

门延迟

- 与门延迟：1级门延迟
- 或门延迟：1级门延迟
- 异或门延迟：3级门延迟

加减法

全加器

- $S_i = X_i \oplus Y_i \oplus C_{i-1}$：6ty
- $C_i = X_iC_{i-1} + X_iY_i + Y_iC_{i-1}$：2ty

串行进位加法器

- 将全加器进行串联
- 延迟（每一步的起始时间取决于输入中最慢计算得到的那个）：
  - $C_i = C_{i-1} + 2 \text{ ty} \rightarrow C_n = 2n \text{ ty}$
  - $S_1 = S_2 = 6$
  - $S_n = C_{i-1} + 3 \text{ ty} \rightarrow S_n = 2n + 1 \text{ ty } (n \geq 3)$
- 优点：元件少
- 缺点：慢，和数据的位数线性相关

全先行进位加法器 CLA

- 思路：预先并行计算部分输入值，提高效率
- 定义两个辅助函数：
  - 传播：$P_i = X_i + Y_i$（将前序结果"传递"下来）
  - 生成：$G_i = X_i Y_i$
- 此时展开加法的每一位，有：
  - C_0
  - $C_1 = G_1 + P_1 C_0$
  - $C_2 = G_2 + P_2 G_1 + P_2 P_1 C_0$
  - $C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_0$
  - ……
- 延迟
  - 计算G_i, P_i：$1\ \text{ty}$
  - 求出所有的C_i：$2\ \text{ty}$
    - 先并行计算所有的与：$1\ \text{ty}$
    - 再并行计算所有的或：$1\ \text{ty}$
  - 最后通过异或求出S_i：$3\ \text{ty}$
- 优点：效率高
- 缺点：元件多

部分先行加法器

- 串联多个CLA
- 权衡效率与复杂度
- 延迟
  - $8n$位加法，串联n个8-bit的CLA
  - 第一个计算出给下一个的C_8需要：$1 + 2 = 3\ \text{ty}$，S_i不与后续元件相关，时间忽略
  - 此后每一个的G_i, P_i都预先算好，计算出给下一个的C_{8i}需要$2\ \text{ty}$
  - 最后一个计算出C_{8n}需要$2\ \text{ty}$，S_{8n}需要$3\ \text{ty}$
  - 因此共需$2n + 4\ \text{ty}$
  - 例如32位就是$12\ \text{ty}$

溢出判断

无符号数

- 最高位产生进位

有符号补码数

- 出现溢出：两个数同号，但是结果和这两个数异号
- 判断方法
  - 比较符号：$X_n = Y_n$，且$S_n \neq X_n, Y_n \rightarrow \text{overflow} = \bar{S_n} X_n Y_n + S_n \bar{X}_n \bar{Y}_n$
  - 比较进位：$C_n \neq C_{n-1} \rightarrow \text{overflow} = C_n \oplus C_n - 1$

减法

- 加相反数
- 数据通路：
  - 对多路选择器和ALU输入加/减的信号
  - 多路选择器若为减则将输入取反输出
  - ALU的加减信号连接在C_0上（合在一起就是取反+1）

乘法

手算

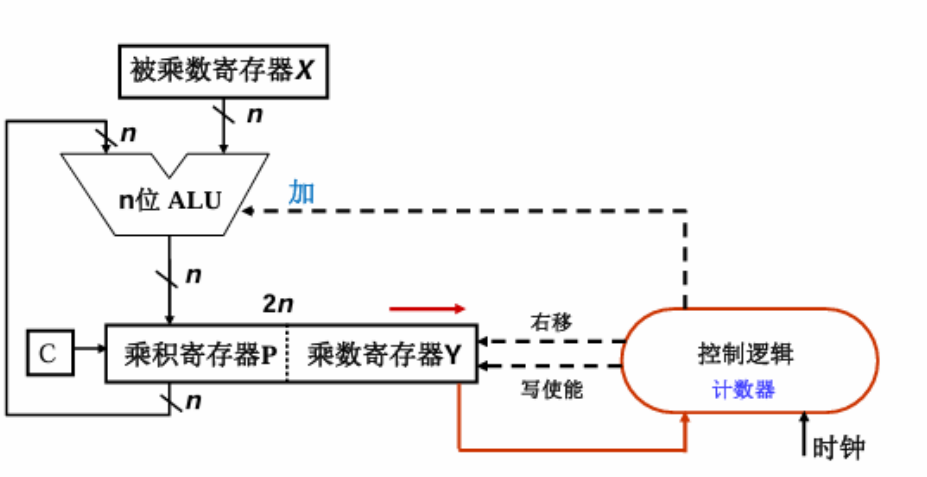
- $Y_i = 0$：**部分积**为0，否则为X
- 每一步部分积左移一位
- 求和

无符号数乘法

思路

- 基于手算思路
- 算一位求和一位，减少保存开销
- 相加时右移先前部分积的和而非左移部分积
- 若$Y_i = 0$，只移位

实现



无符号数乘法的实现

- Y算一位右移一位（算完就可以抛弃了），留出来的空间给不断右移的结果，二者此消彼长
- Y的最低位传给控制逻辑决定是否要加
- 最终结果的高32位在成绩寄存器中，低32位在乘数寄存器中

补码乘法

- 问题：积的补码不等于补码的积
- 简单办法：都转成原码进行无符号计算，将结果再转换为补码（开销太大）

布斯算法

> 补码数的真值表示：$Y = -Y_n \cdot 2^{n-1} + Y_{n-1} \cdot 2^{n-2} + ... + Y_2 \cdot 2^1 + Y_1 \cdot 2^0$

- 乘积：$X \cdot Y = X \cdot \left(-Y_n \cdot 2^{n-1} + Y_{n-1} \cdot 2^{n-2} + ... + Y_2 \cdot 2^1 + Y_1 \cdot 2^0\right)$
  - $= X \cdot$
    $\left((Y_{n-1} - Y_n) \cdot 2^{n-1} + (Y_{n-2} - Y_{n-1}) \cdot 2^{n-2} + ... + (Y_1 - Y_2) \cdot 2^1 + (Y_0 - Y_1) \cdot 2^0\right)$
    （让$Y_0 = 0$）
- 部分积：$P_{i+1} = 2^{-1} \cdot (P_i + X \cdot (Y_i - Y_{i+1}))$
- 流程：
  - 增加$Y_0 = 0$
  - 根据$(Y_i - Y_{i-1})$，在和上+X/-X/不变
  - 右移部分积
  - 重复n次
  - 在高位补充时**符号扩展**

除法

处理极端情况

被除数	除数	结果
0	非0	0
非0	0	“除数为0”异常
0	0	“除法错”异常
非0	非0	进一步运算

手算除法

- 在被除数左侧补0（符号位），将除数的最高位与被除数的次高位对齐
- 被除数是否够减除数
  - 若够减，减去，商写入1
  - 若不够减，商写入0
- 右移除数，重复上述步骤

无符号数除法

- $2n$bits的寄存器共享被除数（余数）和商，每做一位的运算，商和余数都左移一位

补码除法

> 本节中“减法”“够减”按以下定义拓展到补码的情况：
>
> - “减”：将余数（被除数）往0的方向靠近⇒（被除数与除数）同减异加
> - “加”：减的逆运算，（被除数与除数）同加异减
> - “够减”：余数（被除数）往0的方向靠近时不会越过0⇒“减”之前的余数和“减”完的余数同号

- 前面加n位符号扩展被除数，储存在余数&商寄存器中
- 判断“够减”
  - 若“够减”：“减”，商为1
  - 若“不够”：商为0

- **若除数和被除数异号，将商替换为相反数**
- 余数存在余数寄存器中

恢复余数除法

- 按照上面思路来，“不够减”时“加”回去，即“恢复余数”
- 弊端：恢复余数成本高

不恢复余数除法

- 思路：无论够不够减都继续，后一轮执行时根据商的结果进行加/减
  - 若上一位“够减”：“减”
  - 若上一位“不够减”：“加”（移位后，上一次多减的Y相当于要补$2Y$，此时加Y等效于恢复余数除法中的$-Y$

> [!NOTE]
>
> 由于不恢复余数会减过头（使得余数变号），“减”、“够减”每次的意义都有可能改变，因此直接用加、减表示，实际上“减”、“够减”的定义依然是使用的

- 实现
  - 前面加n位符号扩展被除数，储存在余数&商寄存器中
  - 计算余数：
    - 第一次：被除数“减”除数
    - 此后比较余数和除数符号：相同减，不同加
  - 商：
    - 新的余数和除数符号相同：为1
    - 不同：为0
  - 修正商：若除数和被除数异号，商+1
  - 修正除数：若被除数和余数异号，让余数加减除数，使之与被减数同号

注意事项

- 加法器实现的是**模 2n 无符号整数加法运算**，因为
  - $a-b=a_{补}-b_{补} \mod 2n$
  - $a+b=a_{补}+b_{补} \mod 2n$
  - 所以n位有符号整数加减运算都可在 n 位加法器中实现

上一页
04-校验码

下一页
06-浮点运算

最后更新于11个月前