乘法器与除法器

概论：乘除法与处理器

1. 乘法和除法对于处理器的重要性：没有乘法的话，需要很多条指令来实现乘法，除法亦然。
2. 乘法对于处理器的影响：大幅增加处理器延迟（特别是组合乘法），需要使用一些手段来降低延迟
3. 除法对于处理器的影响：除法无法组合化（查表除法消耗了太多资源，而且表地址比较大，延迟也相当可观），因此会造成处理器大幅停顿，以完成多周期的除法
4. 取模运算和除法是同量级的

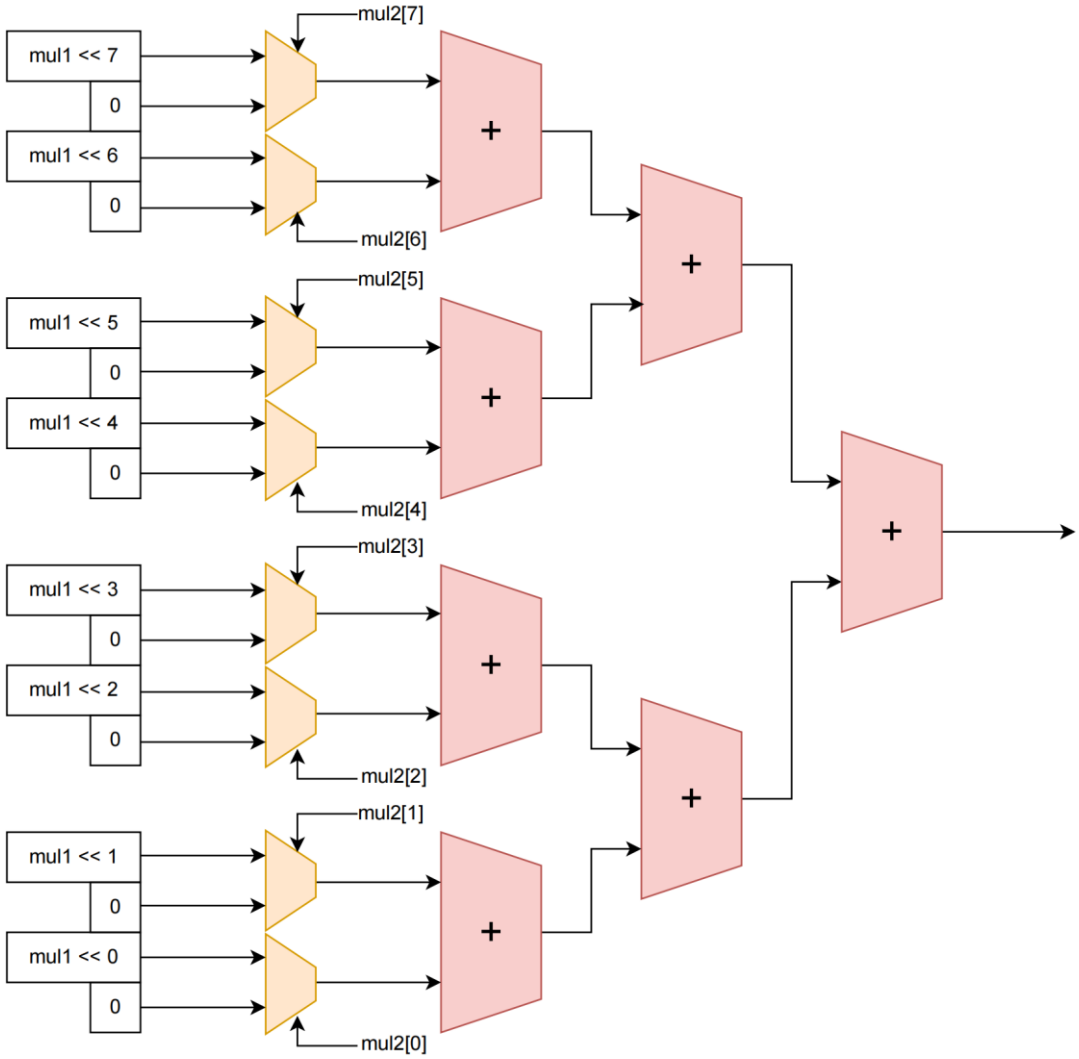
乘法器

简单移位乘法

1. 时序移位乘法的开销：64 个周期
2. 借鉴小学乘法的思路，计算 64 个部分和，不断将其相加即可
3. 硬件的思想：并行增加带宽，是否在这其中有可以并行的部分？

二叉树优化乘法

1. 使用 5 层二叉树，根据乘数，将被乘数适当移位，不断累加至只剩一个结果
2. 为了简单起见每一层二叉树需要使用 64 位加法器（经过精细的控制可以让前几层加法器位数减少一些，不过意义不大），加法器可以参考级内超前进位，级间串行进位



3. 超前进位加法器真的能超前进位吗？未必：

$$C_1 = G_0 + P_0 \cdot C_0$$

$$\mathbf{C_2} = G_1 + P_1 \cdot C_1$$
$$= G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0)$$
$$\mathbf{= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0}$$

$$\boxed{C_{i+1} = G_i + P_i \cdot C_i}$$

$$\mathbf{C_3} = G_2 + P_2 \cdot C_2$$
$$= G_2 + P_2 \cdot (G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0)$$
$$\mathbf{= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0}$$

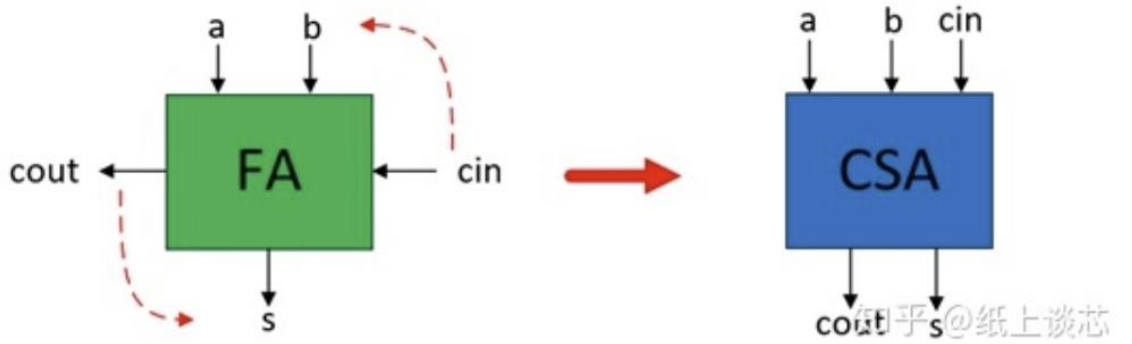
$$\mathbf{C_4} = G_3 + P_3 \cdot C_3$$
$$= G_3 + P_3 \cdot (G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0)$$
$$\mathbf{= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0}$$

可以看到，这里依然存在依赖关系，即便可以使用 MOS 管实现多输入与门，但因为电压的物理因素，一般 4 个输入的与门已经很危险了。所以，超前进位未必是真正的超前，二叉树乘法器的每一个加法器也会引入大量的延迟：

ALU 为 32 为串行加法，这里每一级加法器假如使用 4 位超前进位加法器，那么相当于 16 位串行进位加法器，只需要两级加法器就会使得延迟爆炸。三级流水乘法可以大体满足延迟需求

保留进位加法器

1. 保留进位加法器有 3 输入，2 输出，将三个数的加法转化为其和与每一位的进位：



---

2. 保留进位加法器的优势：任意位宽的加法器都不再依赖低位进位：

$$S = A \oplus B \oplus C$$

$$COUT = AB + BC + AC$$

这基本就是全加器的逻辑

3. 保留进位加法器可以把三个数的和转化为两个数的和，这也是类似于二叉树的方法进行计算。每过一层保留进位加法器，部分和数量减少 2/3，因此也只需要$\log_{1.5} n$层保留进位加法器就可以完成计算。当然，最后一层两个数相加，应当是一个真正的加法器来计算
4. 保留进位加法器虽然层数多，但每一层的延迟都远远小于真正的加法器

2 位 Booth 编码

1. 2 位 Booth 编码可以通过有效的编码来缩减乘法中部分积的个数，由 32 个部分积转化为 16 个部分积
2. 原理：对于有符号数的乘法$X \times Y$，可以使用如下公式来进行变换：

$$\begin{aligned}
&-y_{31} \times 2^{31} + y_{30} \times 2^{30} + y_{29} \times 2^{29} + \cdots + y_1 \times 2^1 + y_0 \times 2^0 \\
&= (y_{29} + y_{30} - 2 \times y_{31}) \times 2^{30} + (y_{27} + y_{28} - 2 \times y_{29}) \times 2^{28} + \cdots \\
&\quad + (y_1 + y_2 - 2 \times y_3) \times 2^2 + (y_{-1} + y_0 - 2 \times y_1) \times 2^0
\end{aligned}$$

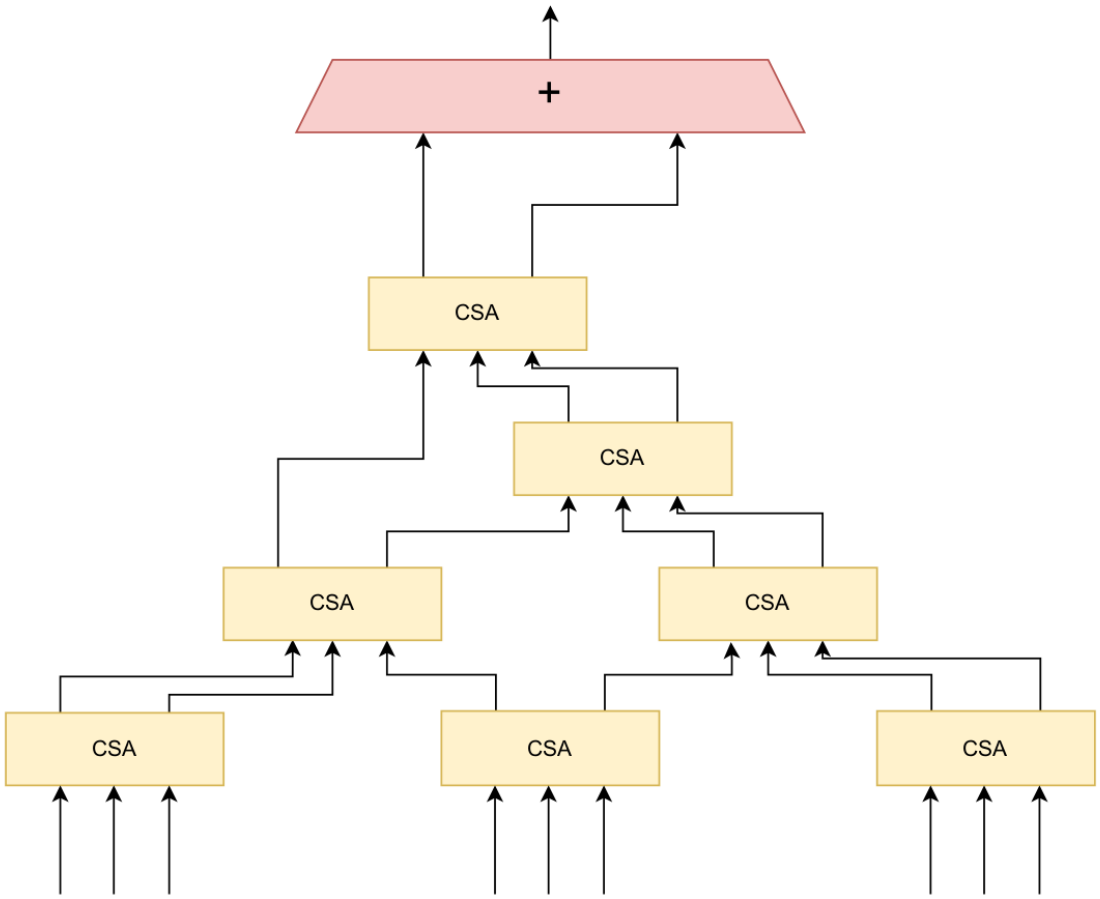
有了这样的新表示，就可以使用编码器对 2 的幂之前的系数进行编码：

Y_{i+1}	Y_i	Y_{i-1}	输出
0	0	0	0
0	0	1	+X
0	1	0	+X
0	1	1	+2X
1	0	0	-2X
1	0	1	-X
1	1	0	-X
1	1	1	0

其中，输出代表通过 Y 的相邻 3 位对 X 编码之后的结果，这和通过看 Y 的每一位来编码是一致的思路

简单华莱士树乘法

1. 简单华莱士树就是树型保留进位加法器的硬件结构，通过不停地使用保留进位加法器，来使得每一层部分积减少 1/3，最终缩减为 2 个数的加法，再通过一个全加器获得最后的结果
2. 在进入华莱士树之前，我们可以使用 2 位 Booth 编码来大幅度缩减部分积的个数。结构如下：

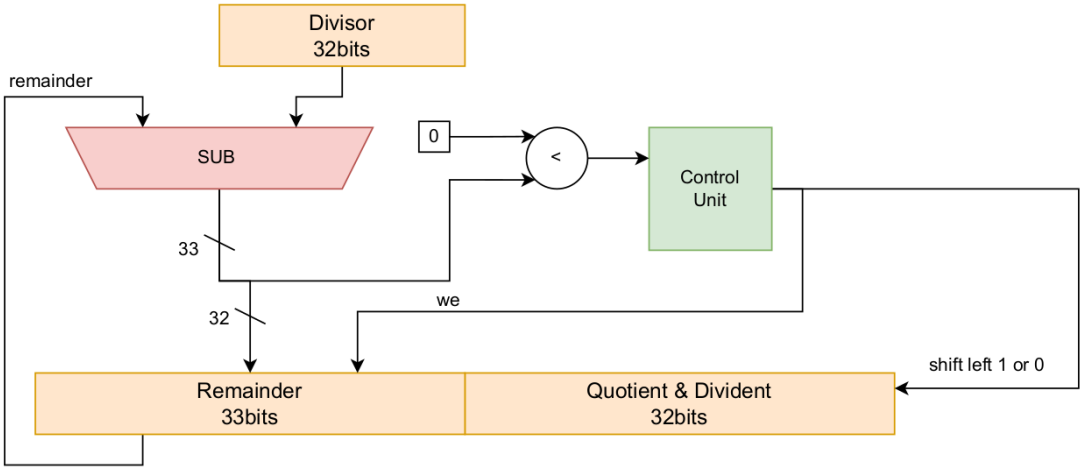


3. 无符号乘法：实现 33 位乘法器，乘数拓展一位，最高位 0 拓展，就可以实现无符号的乘法

除法器

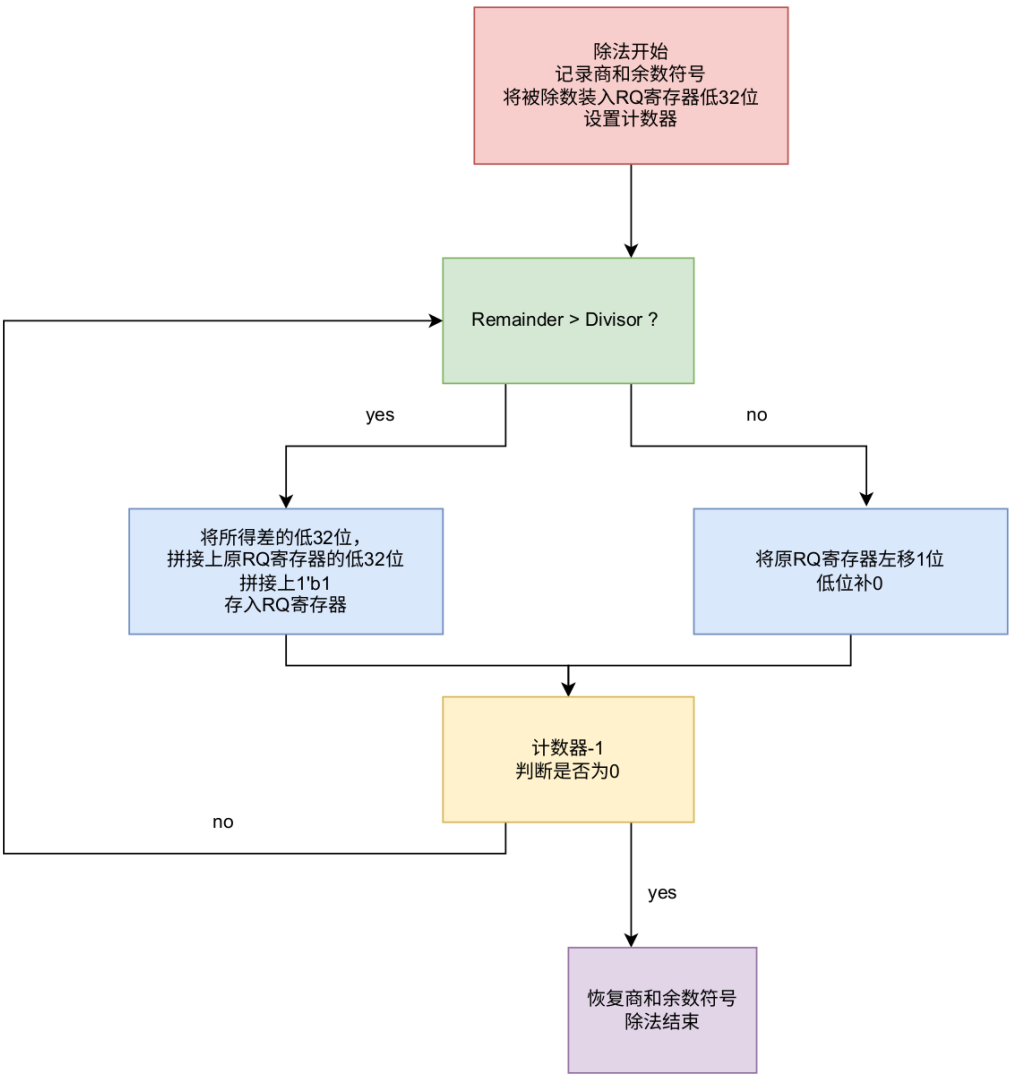
Radix-2 移位除法器

1. 使用小学除法竖式计算的思路，使用多周期对除法进行计算
2. 由于需要多周期，除法器必须由状态机控制，并和流水线功能单元进行握手
3. 在开始除法前，将符号记录下来，并把所有源操作数转换为其绝对值，除法完成后再变换操作数的正负性
4. 移位除法器的核心设计：



- 最下方的 RQ 寄存器，初始时要把被除数放到低位
- 不断试余数和除数差（一直假设在商的位置上写 1），如果差小于 0，那么这一位商为 0，需要左移 RQ 寄存器，来看下一位。如果差大于 0，那么把差的低 32 位放到 RQ 寄存器的高 32 位，并将剩余所有低位左移，低位补 1
- 除法结束后，注意余数此时在最高 32 位中（而不是在 RQ[63:32]中），恢复余数和除数的符号，除法完成

5. 移位除法器的逻辑控制图：



注意：中间部分并不用状态机实现，中间部分完全可以直接进行逻辑判断。除法器的状态机主要是和流水线握手并发送除法开始信号。

6. 时间开销：64 个周期（有点可惜）

提前启动优化

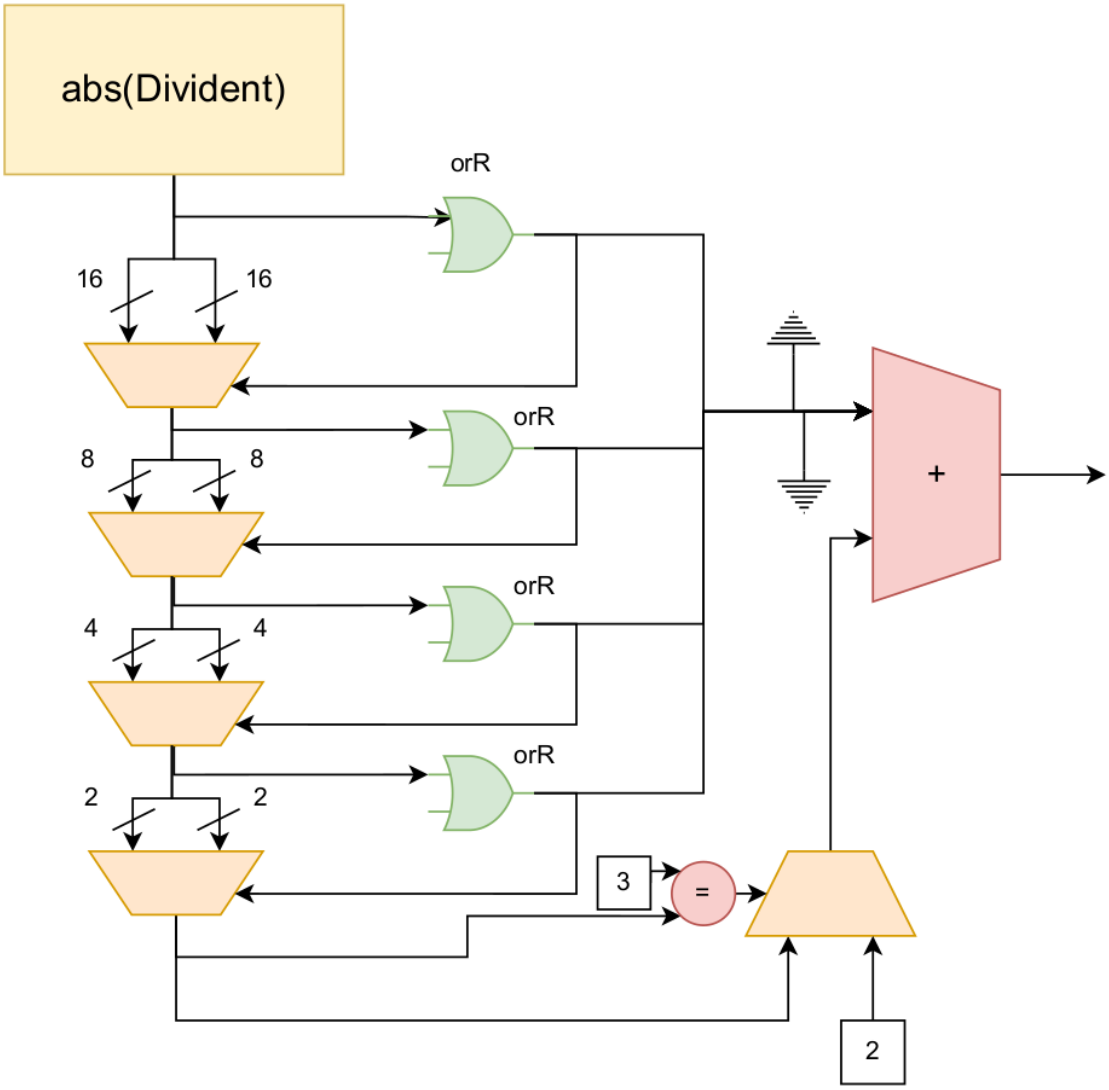
1. 在实际应用的大部分除法中，除数和被除数都不会很大，比如对于 2/1，这样的计算如果需要 64 个周期显然是过于离谱了。提前启动优化就是针对被除数绝对值较小的情况进行优化。
2. 对于除数小的情况，哪怕除数是 3，我们其实都比较难直接看出结果。对于除以 1 和除以 2 的情况，我们选择相信编译器，不会编译出这么诡异的代码。
3. 提前启动：找出被除数高位一共有多少 0。如果直接逐位扫描，尽管可以组合完成，但未免影响时序（考虑 32'b1）。这里可以用二分查找，求以 2 为底的对数取上整的方法来解决

4. 对数器的设计：

   32 位的对数加一应该是 6 位，我们可以使用二分查找找出其[4, 1]位的值，最后通过一次加法生成对数上整

   - 先看这个数最高 16 位是否有 1（可以通过缩位或运算实现），这个结果就是对数中第 4 位的值。如果没有 1，那么取出低 16 位。如果有 1，取出高 16 位
   - 再看取出的 16 位数中最高 8 位是否有 1，这个结果就是对数中第 3 位的值。如果没有 1，那么取出低 8 位。如果有 1，取出高 8 位
   - 以此类推，直到还剩两位数
   - 对于最后两位数，如果它的值是 3，我们需要把结果加上 2。否则，需要加上这个两位数。如此就得到了对数上整的值

5. 数据通路



6. 提前启动的优化：

- 将计数器置为被除数对数上整
- 将被除数右对齐到 RQ 的$32 - \log_2 Divident$位置，