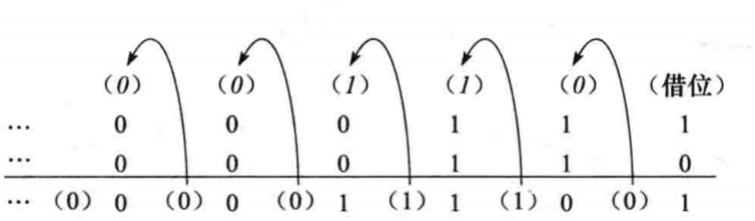
Lecture 6 计算机的整数运算

1. 整数加减法

基本操作

加法是计算机中必备的操作，数据从右到左逐 bit 相加，同时进位也相应地想做传播

$$
\begin{array}{lll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 & = 7_{10} \\
+ & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_2 & = 6_{10} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_2 & = 13_{10}
\end{array}
$$



减法可以采用加法实现，减数在简单的取反之后再进行加法操作

7_{10}减去6_{10}可以直接操作：

$$
\begin{array}{lll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 & = 7_{10} \\
- & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_2 & = 6_{10} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 & = 1_{10}
\end{array}
$$

或者通过加上 −6 的二进制补码来实现：

$$
\begin{array}{lll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 & = 7_{10} \\
+ & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_2 & = -6_{10} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 & = 1_{10}
\end{array}
$$

溢出判断

操作	操作数A	操作数B	表示结果有溢出的条件
$A+B$	$\geqslant 0$	$\geqslant 0$	<0
$A+B$	<0	<0	$\geqslant 0$
$A-B$	$\geqslant 0$	<0	<0
$A-B$	<0	$\geqslant 0$	$\geqslant 0$

加法

- 两个正操作数相加
    - 如果符号为1，则溢出
- 两个负操作数相加
    - 如果符号为0，则溢出

减法

- 正操作数减掉一个负操作数
    - 若符号为1，则溢出
- 负操作数减掉一个正操作数
    - 若符号为0，则溢出

忽略溢出

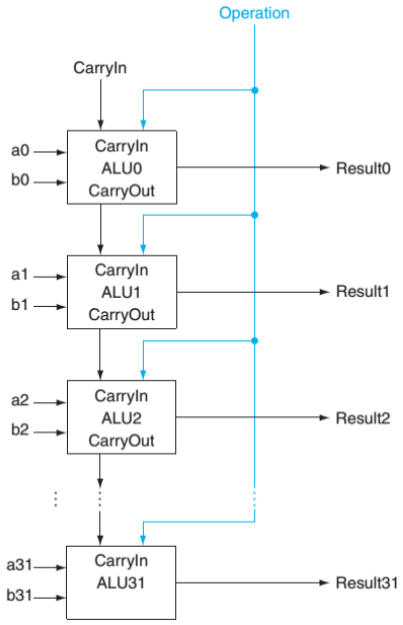
无符号计算通常忽略溢出

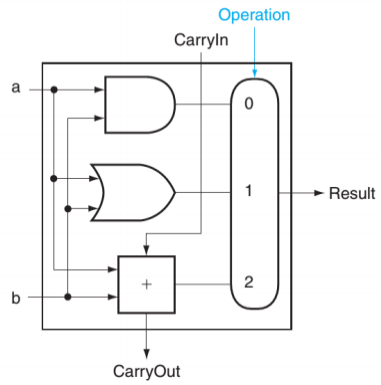
- 加法（add）、立即数加法（addi）、减法（sub）这三条指令在溢出时候产生异常
- 无符号加法（addu）、立即数无符号加法（addiu）和无符号减法（subu），这三条指令在溢出时不会产生异常

注意，addiu 进行的加法操作不会产生异常，但是指令中的立即数字段是有符号的，这个 16-bit 的立即数会符号拓展至 32-bit，即使操作是“无符号”的，立即数字段是有符号的

算数逻辑单元 ALU

用于执行加法、减法，通常也包括逻辑与、逻辑或等逻辑操作硬件





- op = 0，结果 = a & b
- op = 1，结果 = a | b
- op = 2，结果 = a + b

2. 整数乘法

基本操作

我们以十进制数来表示乘法的操作，例如，计算 $1000_{10} \times 1001_{10}$

被乘数 1000_{10}
乘数 x 1001_{10}
1000
0000
0000
1000
积 1001000_{10}

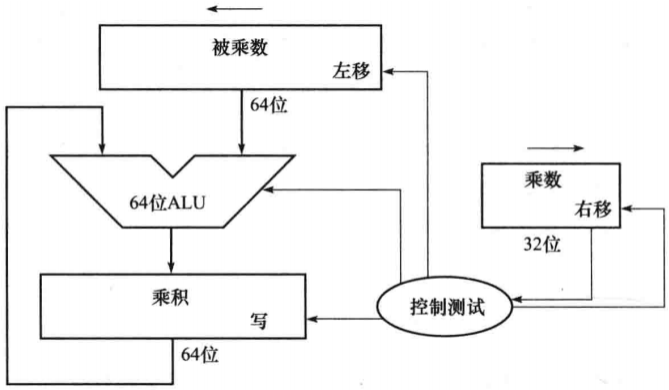
- 第一个操作数被称为被乘数 multiplicand
- 第二个操作数被称为乘数 multiplier

如果我们忽略符号位，若被乘数为 **n-bit**，乘数为 **m-bit**，则积的位数为 **n+m-bit**

符号位处理

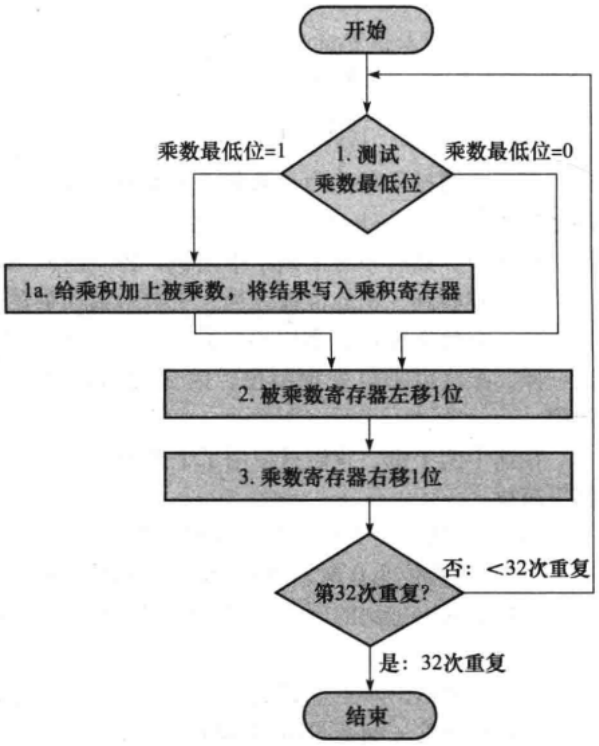
将被乘数和乘数都转化为正数，并记住原来的符号位，符号位不参与运算，再计算完毕后根据符号位直接填充

顺序的乘法算法



- ALU 为 64-bit
- 乘积寄存器为 64-bit
- 32-bit 的被乘数被扩展到 64-bit（高位补 0），每一次迭代向左边移动 1-bit
- 32-bit 的乘数每一次迭代向右边移动 1-bit
- 如果检测到乘数最右边为 0，则被乘数不会被加和到乘积寄存器中去
- 如果检测到乘数最右边为 1，则被乘数会被加和到乘积寄存器中去

流程图如下



示例

我们以 4-bit × 4-bit 的两个数操作为例，$1000_2 \times 1001_2$

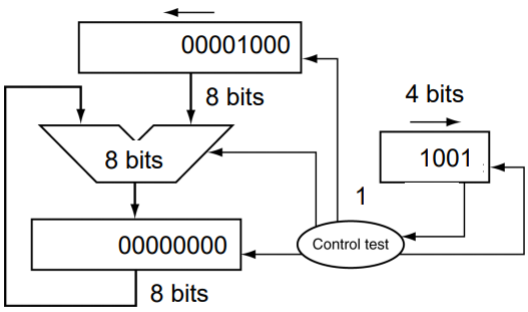
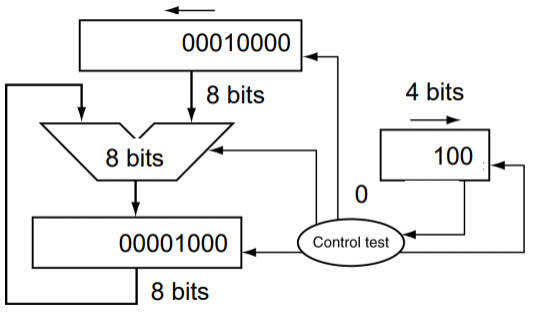
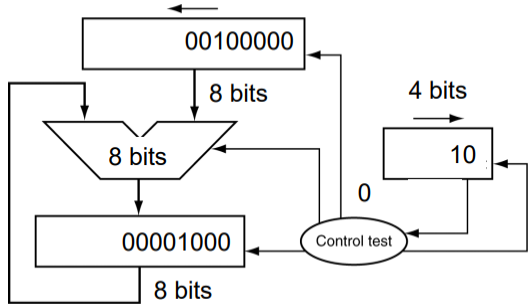
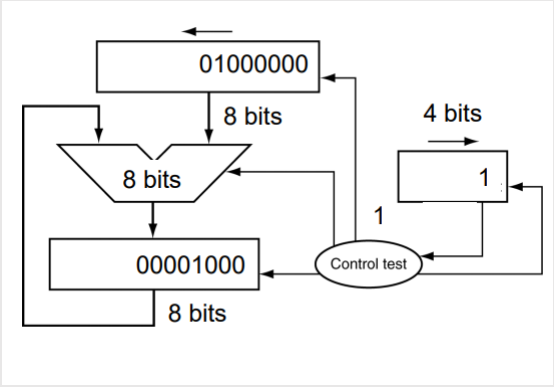
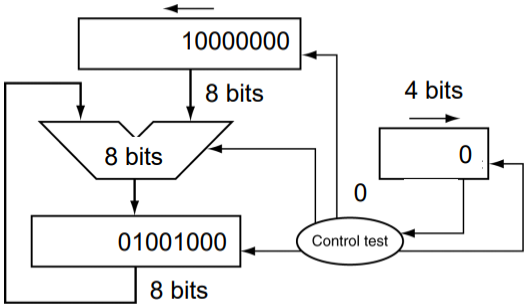
示意图	解释
	初始化被乘数被拓展到 8-bit，高位补 0 乘数最右位为 1， 被乘数加和到乘积里
	乘数最右位为 0， 被乘数不会加和到乘积里

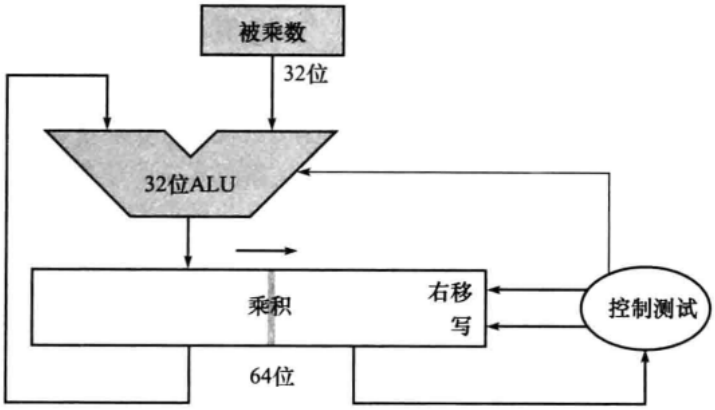
示意图	解释
	乘数最右位为 0， 被乘数不会加和到乘积里
	乘数最右位为 1， 被乘数加和到乘积里
	完成乘法操作

例题-乘法器流程

计算 $0010_2 \times 0011_2$ 的乘积

迭代次数	步骤	乘数	被乘数	乘积
0	初始值	001①	0000 0010	0000 0000
1	1a：1⇒乘积=乘积+被乘数	0011	0000 0010	**0000 0010**
	2：左移被乘数	0011	**0000 0100**	0000 0010
	3：右移乘数	**000①**	0000 0100	0000 0010
2	1a：1⇒乘积=乘积+被乘数	0001	0000 0100	**0000 0110**
	2：左移被乘数	0001	**0000 1000**	0000 0110
	3：右移乘数	**000⓪**	0000 1000	0000 0110
3	1：0⇒无操作	0000	0000 1000	0000 0110
	2：左移被乘数	0000	**0001 0000**	0000 0110
	3：右移乘数	**000⓪**	0001 0000	0000 0110
4	1：0⇒无操作	0000	0001 0000	0000 0110
	2：左移被乘数	0000	**0010 0000**	0000 0110
	3：右移乘数	**0000**	0010 0000	0000 0110

优化的乘法器



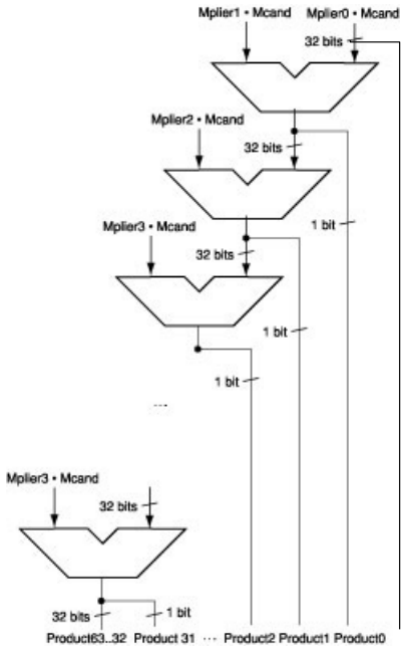
- ALU 为 32-bit
- 乘积寄存器为 64-bit
- 被乘数为 32-bit
- 乘数 32-bit 放在乘积寄存器的右半部分，高位补 0，每一次迭代乘积寄存器向右边移动 1-bit
- 若检测到乘积寄存器最右边为 1，则乘积的左半部分与被乘数相加

快速乘法器

之前的乘法器需要一个时钟来确保前面的加法在移位之前已经完成

下面的快速乘法器可以快速设置大多数输入——然后它必须等待每个相加的结果向下传播——更快，因为不涉及时钟，快速乘法器可以不需要长时钟周期，而**需要多个ALU**用于计算

我们可以乘法运算开始时检查乘数的 32-bit，用来判定被乘数是否需要加上，每个加法器输出被乘数与 1-bit 乘数相与的结果

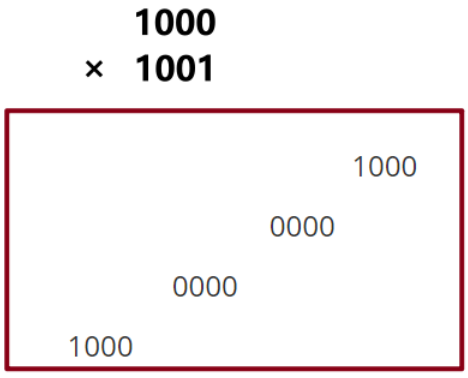


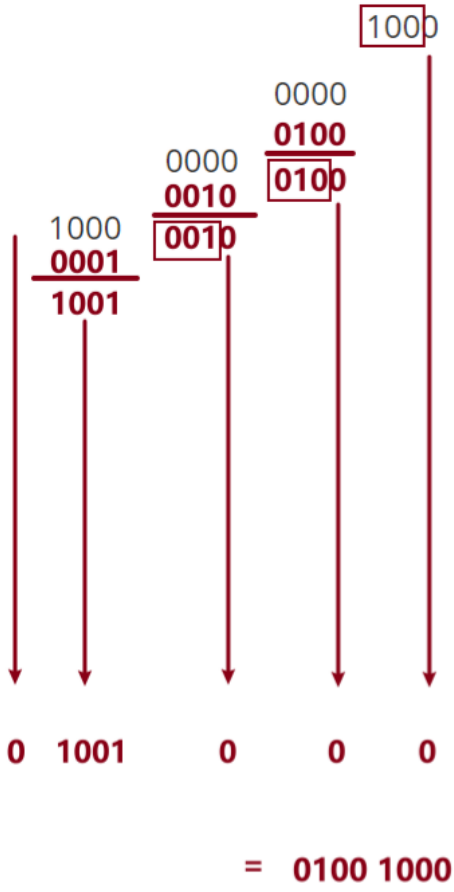
也可以将 32-bit 加法器组织成一个并行树，这样只需要等待 $log_2 32$ 的时间即可完成乘法

示例

运用快速乘法器计算 $1000_2 \times 1001_2$

首先将 $1000_2 \times 1001_2$ 按照比特位计算出乘数每比特位对应的结果





MIPS乘法

```
1  mult rs, rt     # rs * rt 有符号乘法
2  multu rs, rt    # rs * rt 无符号乘法
3  mul rd, rs, rt  # rt = rs * rt
```

可以测试 HI 查看乘积结果是否超过 32bit

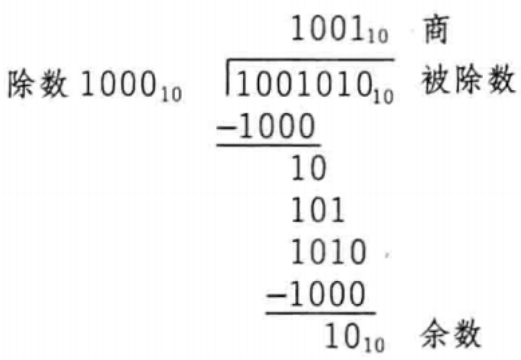
为了取得 32-bit 整数积，需要使用 `mflo` 指令，MIPS 使用伪代码对其进行了封装

MIPS 乘法指令**都忽略溢出**，需要手动检测

3. 整数除法

基本操作

我们以十进制数来表示除法的操作，例如，计算 $1001\ 010_{10} \div 1000_{10}$

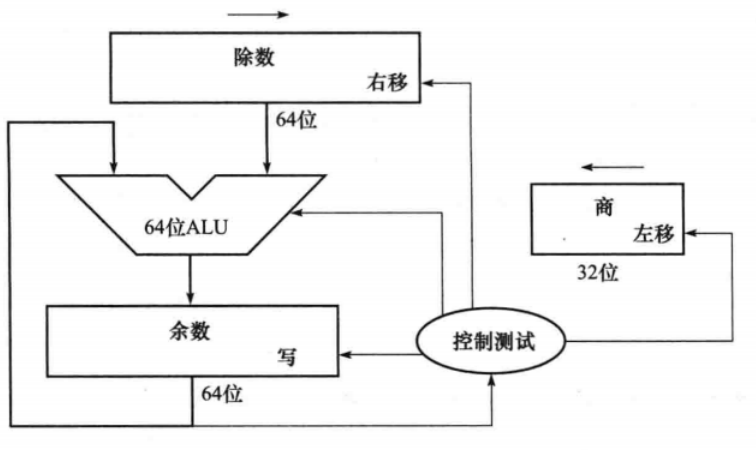


- 第一个操作数被称为被除数 dividend
- 第二个操作数被称为除数 divisor
- 除法结果被称为商 quotient
- 除法另一个结果被称为余数 remainder

除法的流程

1. 检查除数是否为 0
2. 若除数不是 0
    a. 如果除数 ≤ 被除数位
    商1位，相减1位
    b. 否则
    商0位，取下被除数位

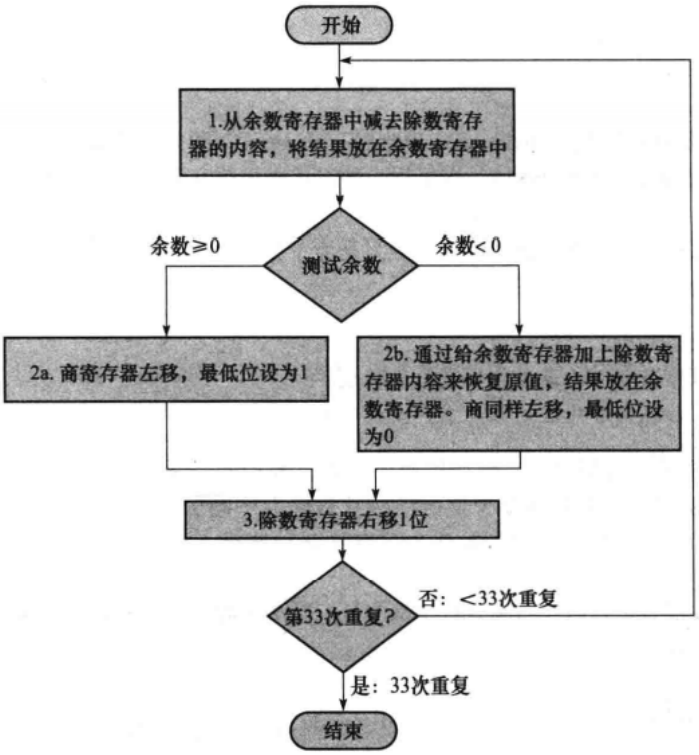
顺序的除法算法



- ALU 为 64-bit
- 商寄存器为 32-bit

- 除数寄存器为 64-bit，开始将除数放置在 64-bit 除数寄存器的左半边，每一次迭代向右边移动一位
- 余数寄存器为 64-bit，开始将被除数放置在 64-bit 余数寄存器的右半边
- 如果检测到乘数最右边为 0，则被乘数不会被加和到乘积寄存器中去
- 如果检测到乘数最右边为 1，则被乘数会被加和到乘积寄存器中去

流程图如下



示例

我们以 4-bit × 4-bit 的两个数操作为例，$0000\ 0111_2 \div 0010_2$

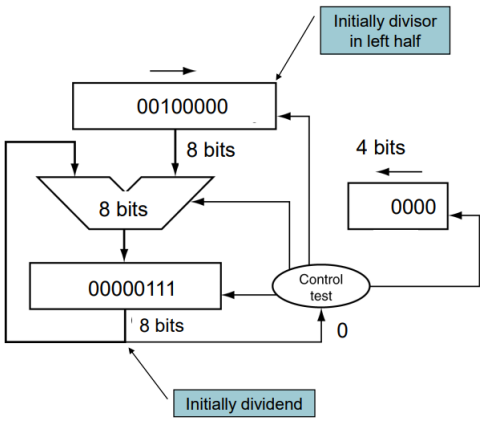
示意图	解释
	除数放置在除数寄存器，被拓展到 8-bit，低位补 0 被除数放置在余数寄存器，被拓展到 8-bit，高位补0 余数寄存器 - 除数寄存器 < 0 余数寄存器恢复，商寄存器右移 0 除数寄存器右移一位

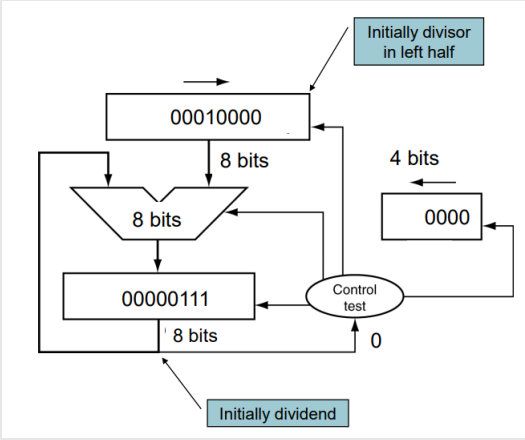
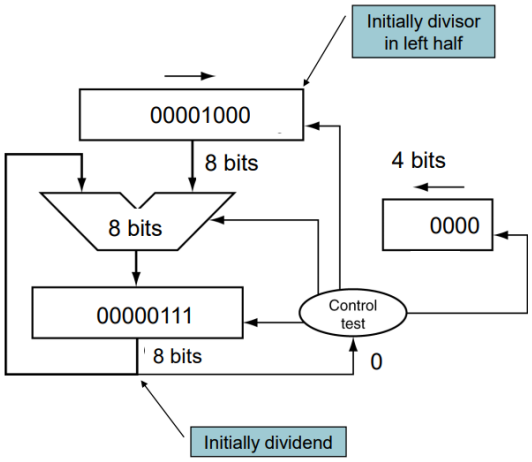
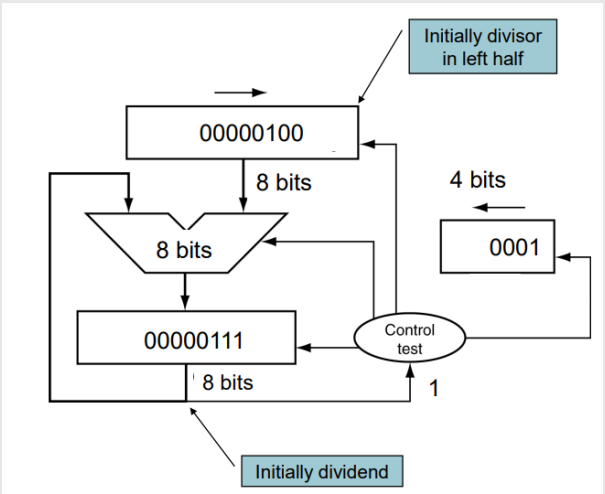
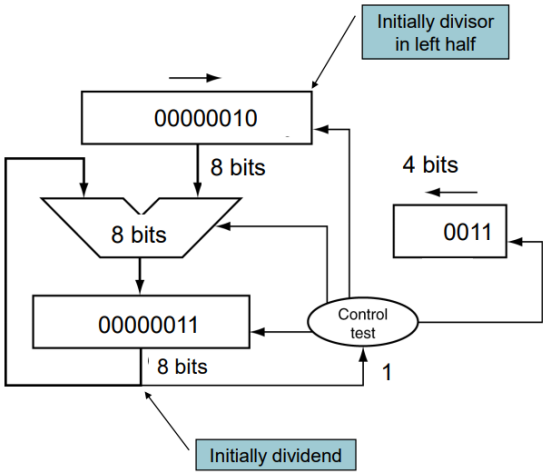
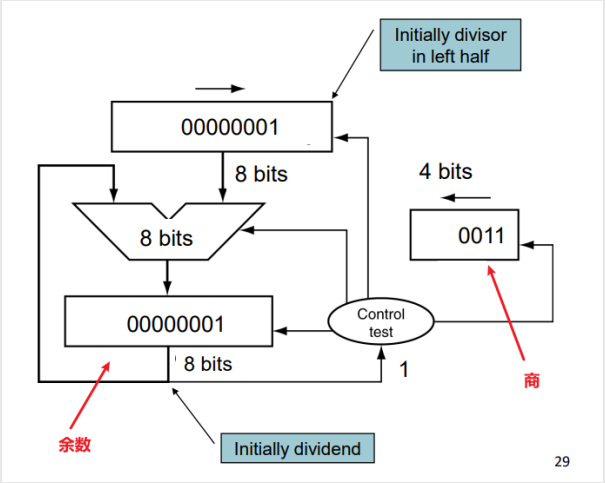
示意图	解释
	余数寄存器 - 除数寄存器 < 0 余数寄存器恢复，商寄存器左移 0 除数寄存器右移一位
	余数寄存器 - 除数寄存器 < 0 余数寄存器恢复，商寄存器左移 0 除数寄存器右移一位
	余数寄存器 - 除数寄存器 ≥ 0 商寄存器左移 1 除数寄存器右移一位
	余数寄存器 - 除数寄存器 ≥ 0 商寄存器左移 1 除数寄存器右移一位

示意图	解释
	完成操作

例题-除法器流程

计算 $0111_2 \div 0010_2$ 的商和余数

7 (0000 0111two) 除以2 (0010two)

Iter	Step	Quot	Divisor	Remainder
0	Initial values 11100111 +00010000	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 ➔ +Div, shift 0 into Q 商0	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
	商0	0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1 商0	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem >= 0 ➔ shift 1 into Q 商1	0001	0000 0100	0000 0011
	Shift Div right 余数	0001	0000 0010	0000 0011
5	Same steps as 4 商1	00011	0000 0001	0000 0001

迭代次数	步骤	商	除数	余数
0	初始值	0000	0010 0000	0000 0111
1	1：余数=余数-除数	0000	0010 0000	**①110 0111**
	2b：余数<0 ⇒+除数，商左移，上最低位上0	**0000**	0010 0000	**0000 0111**
	3：除数右移	0000	**0001 0000**	0000 0111
2	1：余数=余数-除数	0000	0001 0000	**①111 0111**
	2b：余数<0 ⇒+除数，商左移，上最低位上0	**0000**	0001 0000	**0000 0111**
	3：除数右移	0000	**0000 1000**	0000 0111
3	1：余数=余数-除数	0000	0000 1000	**①111 1111**
	2b：余数<0 ⇒+除数，商左移，上最低位上0	**0000**	0000 1000	**0000 0111**
	3：除数右移	0000	**0000 0100**	0000 0111
4	1：余数=余数-除数	0000	0000 0100	**⓪000 0011**
	2a：余数≥0 ⇒商左移，上最低位上1	**0001**	0000 0100	0000 0011
	3：除数右移	0001	**0000 0010**	0000 0011
5	1：余数=余数-除数	0001	0000 0010	**⓪000 0001**
	2a：余数≥0 ⇒商左移，上最低位上1	**0011**	0000 0010	0000 0001
	3：除数右移	0011	**0000 0001**	0000 0001

MIPS除法

```
div rs, rt      # rs/rt
divu rs, rt     # rs/rt
```

- HI：存放 32bit 余数
- LO：存放 32bit 商

MIPS 没有无溢出或除 0 检查，需要自行检查

使用 `mfhi`，`mflo` 获得结果

6. 处理溢出

不同语言的溢出异常检测

- C 和 Java 会忽略整数溢出
- Ada 和 Fortran 会检测溢出

MIPS 异常检测

MIPS 检测到溢出时会发生异常

它使用一个叫做异常程序计数器 Exception Program Counter，EPC 的寄存器来保存导致异常的指令地址，指令 mfc0 (move from system control) 用来将 EPC 存入通用寄存器，从而使 MIPS 软件可以通过寄存器跳转指令返回到导致异常的指令那里

溢出时的异常处理

- 保存**PC在异常程序计数器（EPC）**寄存器
- 跳转到预定义的处理程序地址
- mfc0 指令可以检索EPC值，在纠正操作后返回

MIPS 允许将寄存器 $k0 和 $ k1 预留给操作系统，这些寄存器在异常时不会恢复

7. 多媒体的算数处理

图形和多媒体处理操作

8bit 和16bit 数据的向量

- 将一个 64bit 的数据拆分成 8x8bit，4x16bit，2x32bit的向量，逐一相加
- **SIMD**（Single Instruction Multiple Data）：单指令，处理多数据

饱和 saturating

饱和意味着当计算结果溢出时，结果被设置为最大的整数/最小的负数

应用

- 改变音频或视频的音量或亮度