计算机组成原理
CH3_计算机的算术运算

卢建良
lujl@ustc.edu.cn
2022年春季学期

提纲

3.1 引言

- 在32bit处理器中，可以表示2^{32}个整数
  - 无符号数：0 ~ 2^{32}-1
  - 有符号数：- 2^{31} ~ 2^{31}-1

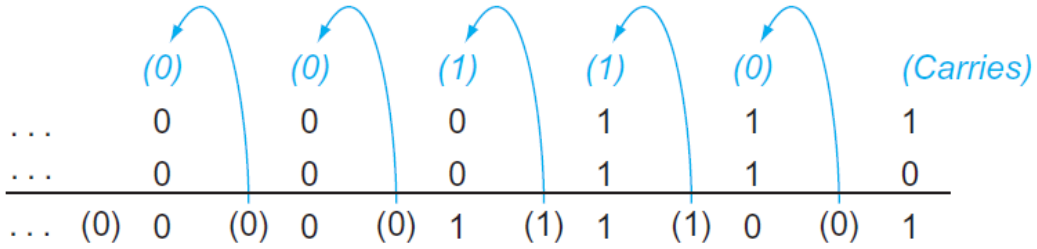
- 其它数字该如何表示?
  - 如何表示小数和其它实数?
  - 如果运算产生了大到无法表示的数，该如何处理?
  - 硬件如何实现乘法、除法运算?

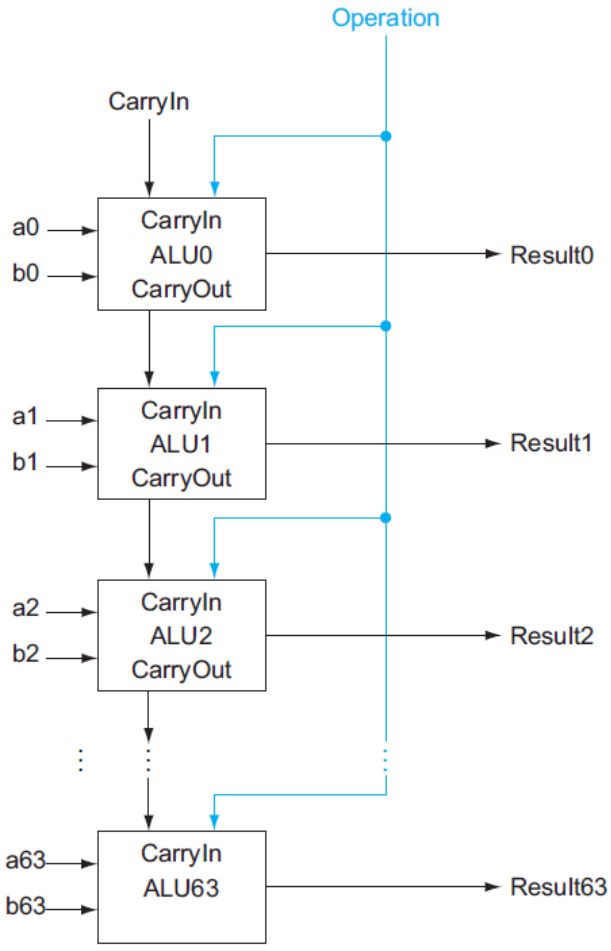
- 本章内容
  - 实数的表示
  - 算术的算法
  - 硬件结构

3.2 加法和减法

- 示例：7 + 6 = 13
  - 硬件实现可参考右图（附录A.5.2）

```
  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000111two = 7ten
+ 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000110two = 6ten
-----------------------------------------------------------------------------------
= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001101two = 13ten
```





- 当结果超出表示范围时，会发生溢出
- 正数与负数相加，不会溢出
- 两个正数相加，如果结果符号位为1，则溢出
- 两个负数相加，如果结果符号位为0，则溢出

3.2 加法和减法

- 示例： 7 – 6 = 1
  - 方式1：通过正常的减法算式来计算

$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\
- & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}
\end{array}
$$

  - 方式2：用二进制补码来表示-6，计算7 + （-6）

$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\
+ & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111010_{two} = -6_{ten} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}
\end{array}
$$

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

3.2 加法和减法

- 图像和多媒体处理操作经常涉及8bit、16bit的数据
  - 使用64bit的加法器，同时将进位链进行适当分割
  - 实现8*8bit、4*16bit、2*32bit的向量操作
  - SIMD： single-instruction, multiple-data

- 饱和操作
  - 当计算溢出时，结果设置为最大正数或最小负数
  - eg：收音机音量旋钮

3.2 加法和减法

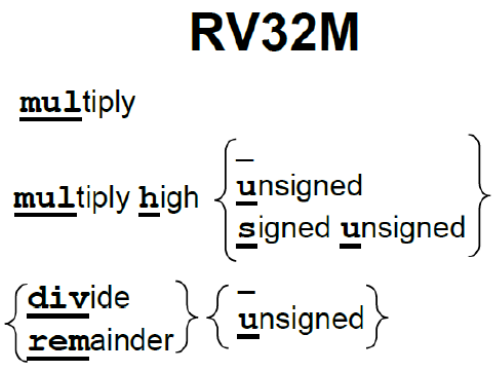
Check Yourself

Some programming languages allow two's complement integer arithmetic on variables declared byte and half, whereas RISC-V only has integer arithmetic operations on full words. As we recall from Chapter 2, RISC-V does have data transfer operations for bytes and halfwords. What RISC-V instructions should be generated for byte and halfword arithmetic operations?

1. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mul`, `div`, using `and` to mask result to 8 or 16 bits after each operation; then store using `sb`, `sh`.
2. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mul`, `div`; then store using `sb`, `sh`.

3.3 乘法

- RV32M指令集
  - 定义了可选的RV32M，它定义了整数乘法除法指令，总共8条指令



31 25	24 20	19 15	14 12	11 7	6 0	
0000001	rs2	rs1	000	rd	0110011	R mul
0000001	rs2	rs1	001	rd	0110011	R mulh
0000001	rs2	rs1	010	rd	0110011	R mulhsu
0000001	rs2	rs1	011	rd	0110011	R mulhu
0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

  - 示例：8 * 9 = 72（0x48）

Address	Code	Basic	
0x00000000	0x00800293	addi x5, x0, 8	1: li t0, 8
0x00000004	0x00900313	addi x6, x0, 9	2: li t1, 9
0x00000008	0x02628533	mul x10, x5, x6	3: mul a0, t0, t1

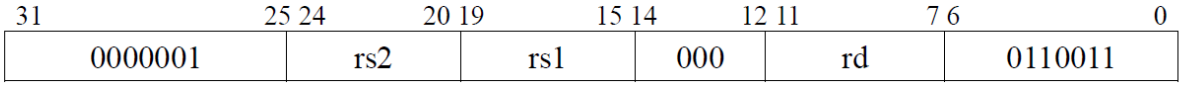
3.3 乘法

- 带符号的乘法
  - 将被乘数和乘数转换为正数进行计算，最后进行符号转换
- RISC-V中的乘法（RV64M）
  - mul，乘
  - mulh，乘法取高位
  - mulhu，无符号乘法取高位
  - mulhsu，有符号/无符号乘法取高位

mul rd, rs1, rs2 $x[rd] = x[rs1] \times x[rs2]$

乘*(Multiply)*. R-type, RV32M and RV64M.

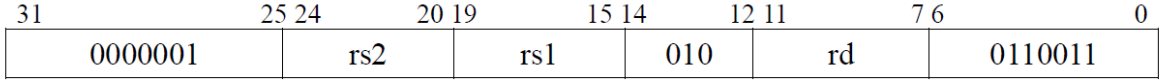
把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，乘积写入 x[*rd*]。忽略算术溢出。



mulhsu rd, rs1, rs2 $x[rd] = (x[rs1]_s \times_u x[rs2]) \gg_s XLEN$

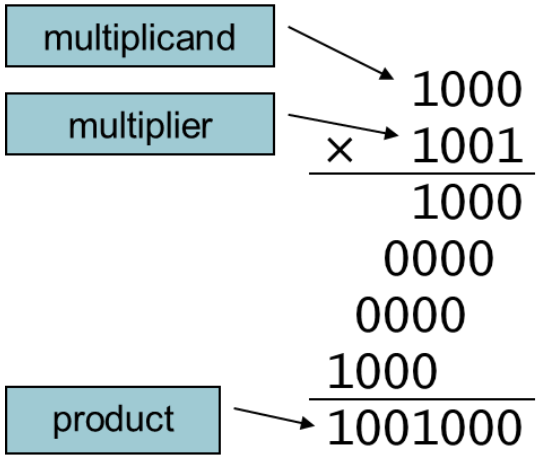
高位有符号−无符号乘*(Multiply High Signed-Unsigned)*. R-type, RV32M and RV64M.

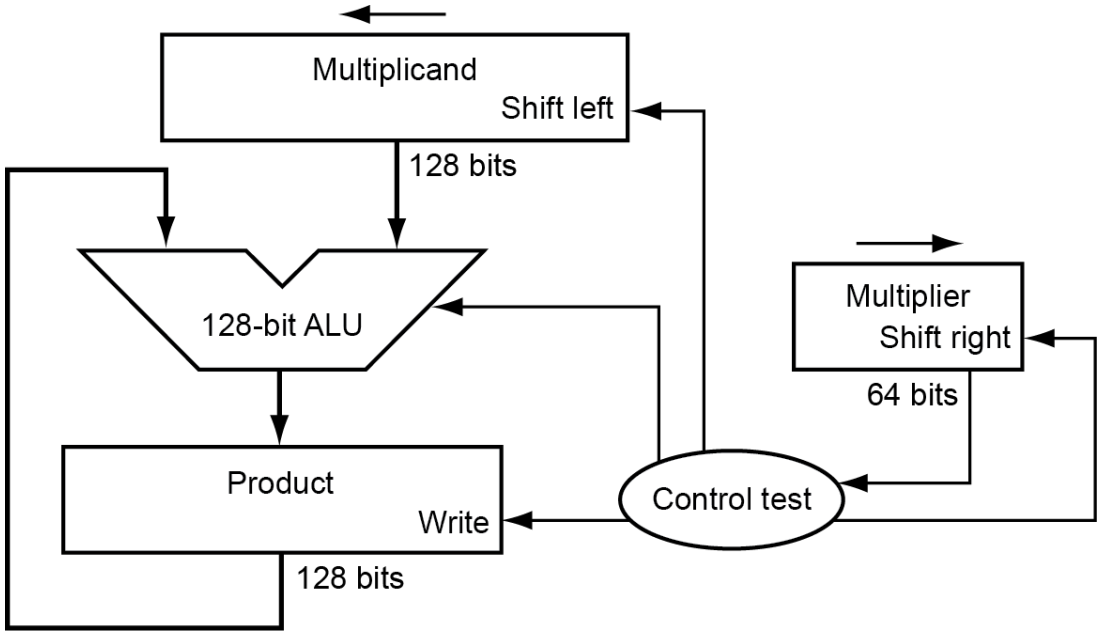
把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，x[*rs1*]为 2 的补码，x[*rs2*]为无符号数，将乘积的高位写入 x[*rd*]。



3.3 乘法

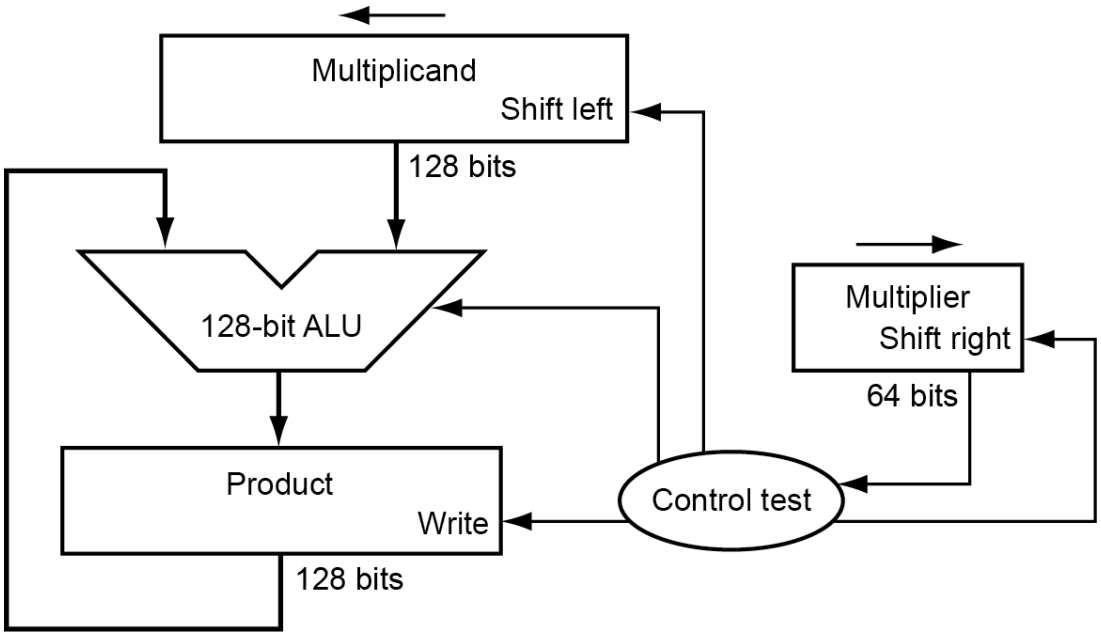
- 被乘数：multiplicand，乘法算式的第一个操作数
- 乘数：multiplier，乘法算式的第二个操作数
- 积：product，乘法算式的结果

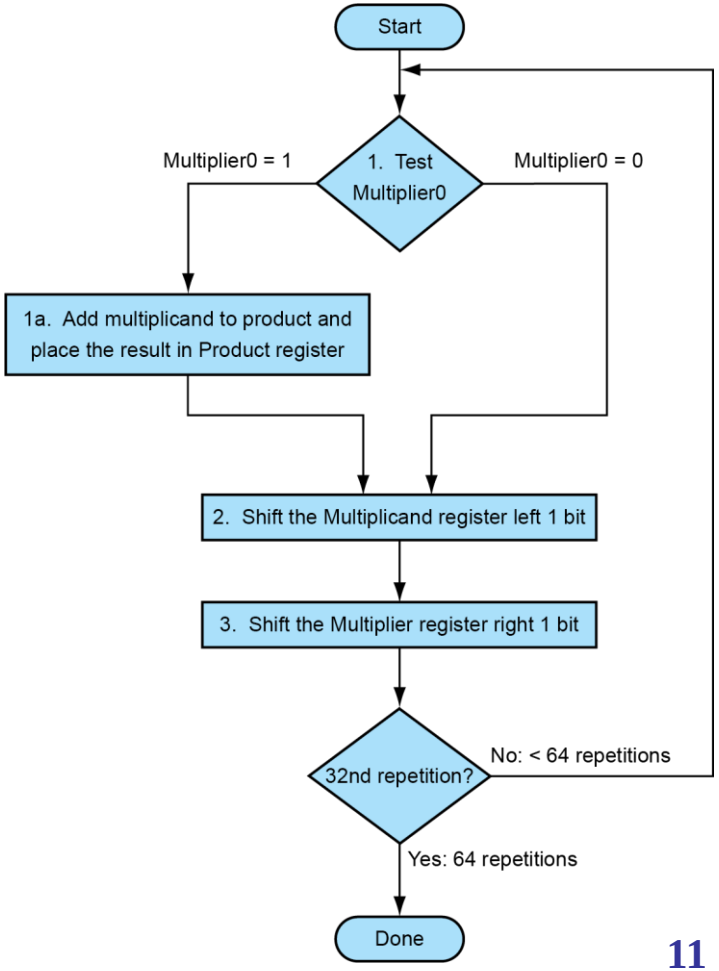




3.3 乘法

- 乘法器硬件结构及算法实现，耗时约200个时钟周期
  - 1. 检测乘数最低位
    - 为1，将被乘数加到积上，并存入寄存器
    - 为0，则跳过
  - 2. 将被乘数寄存器左移一位
  - 3. 将乘数寄存器右移一位





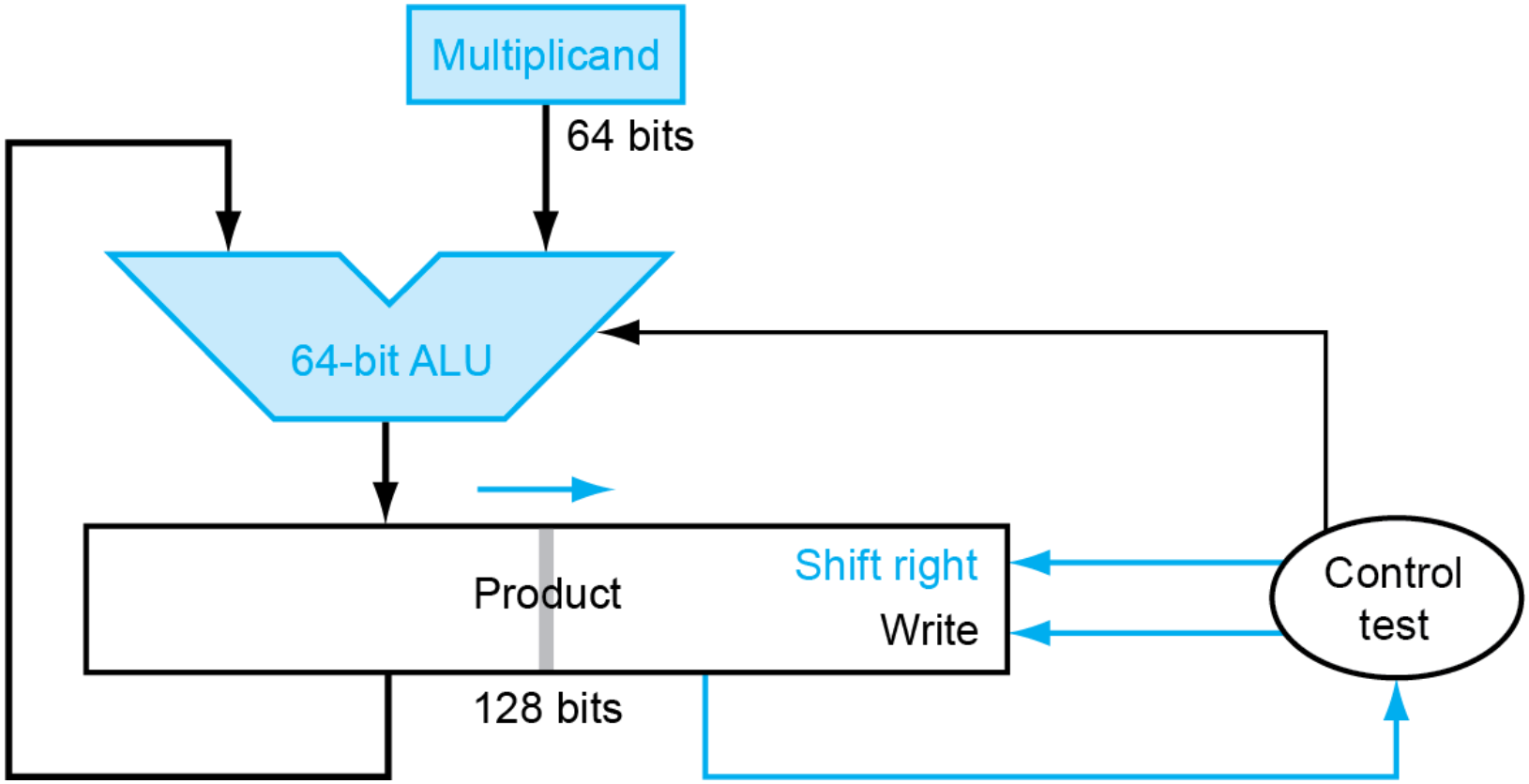
3.3 乘法

- 示例：2 * 3 (或者0010 * 0011)

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0011	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

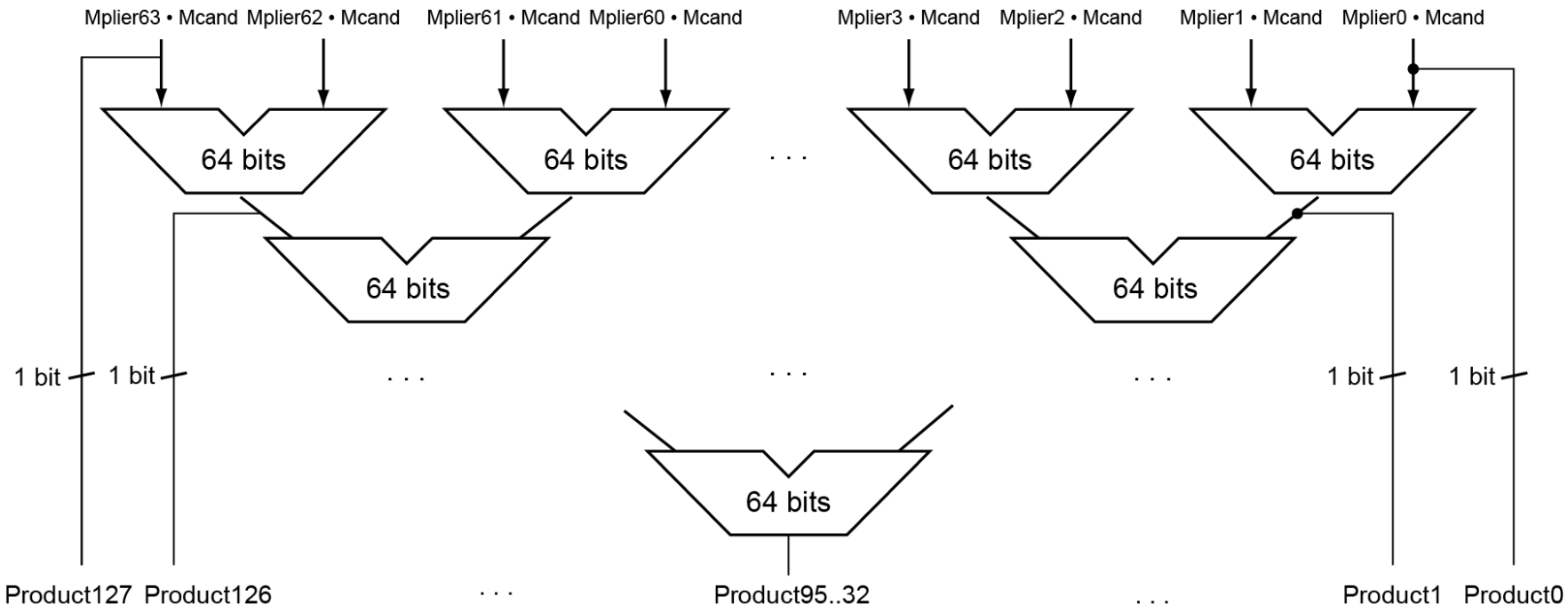
3.3 乘法

- 经过优化的乘法器，耗时约64个时钟周期



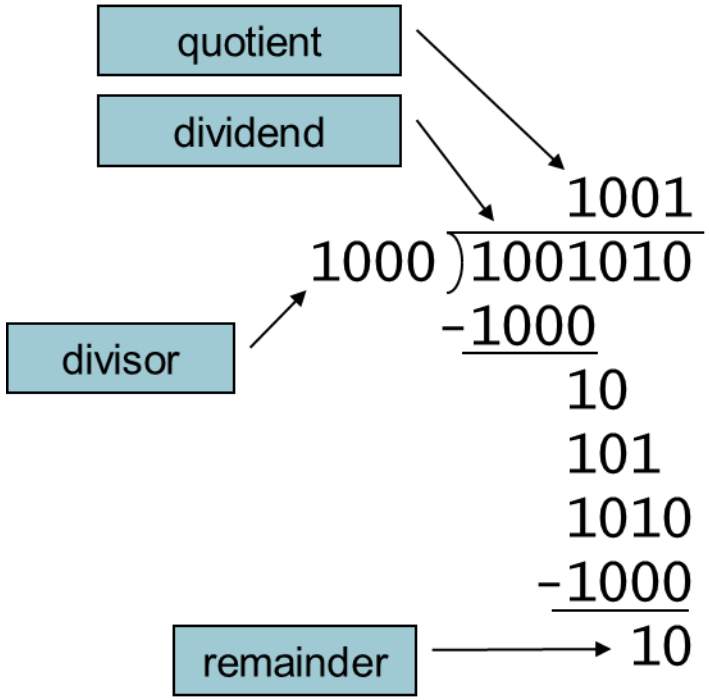
3.3 乘法

- 快速乘法，耗时约6个时钟周期
  - 使用多个加法器
  - 运算速度快于做6次加法
  - 可以通过流水线技术，同时支持多个乘法



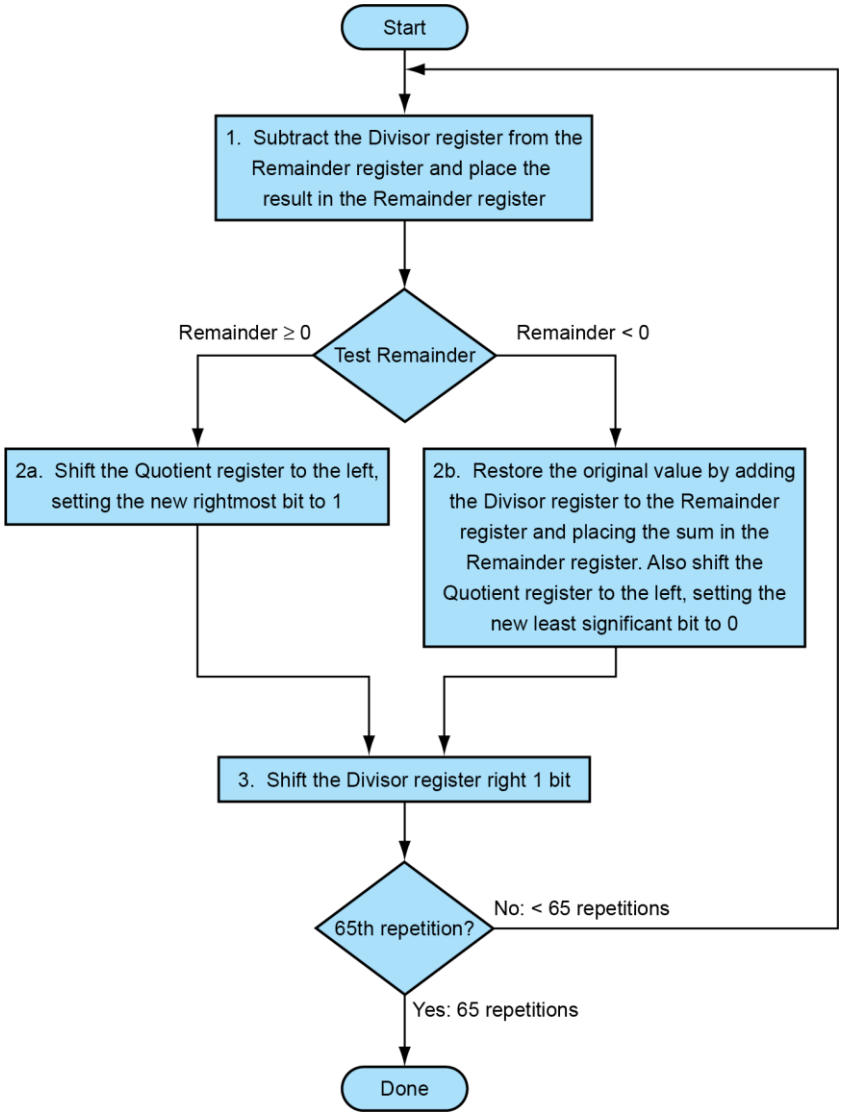
3.4 除法

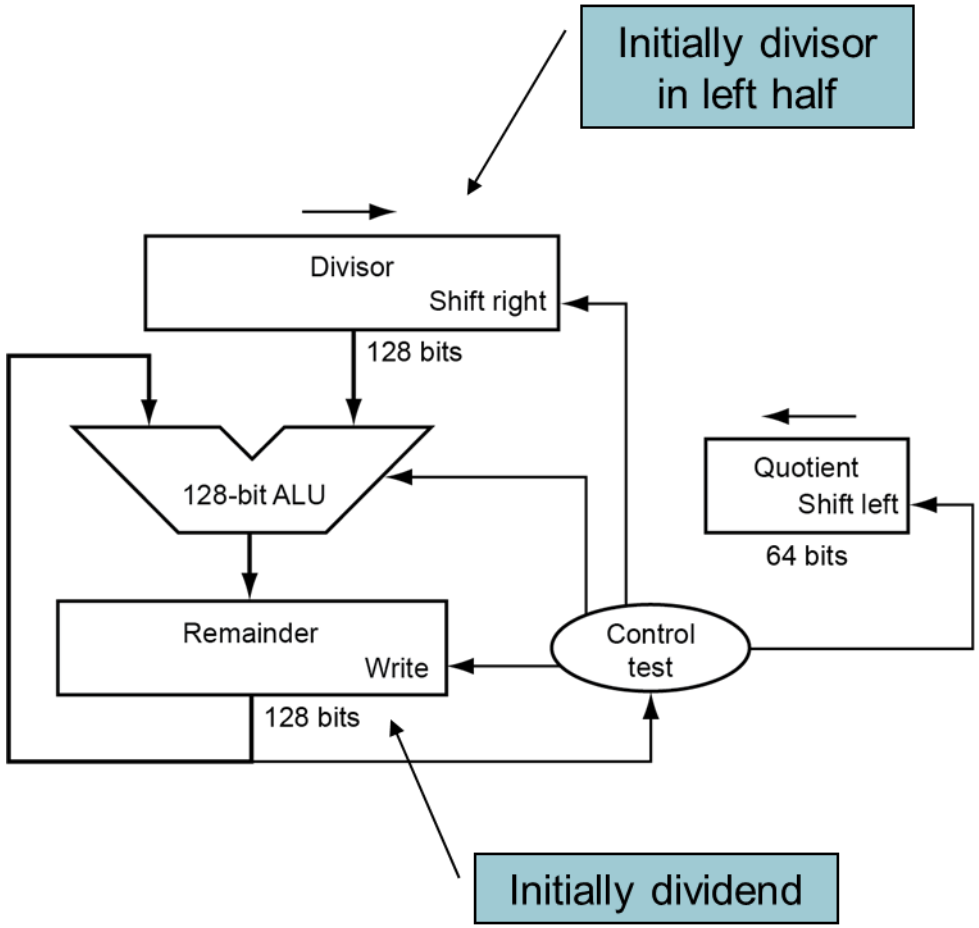
- 被除数：dividend
- 除数：divisor
- 商：quotient
- 余数：remainder
- 例：74 ÷ 8 = 9 余 2



- 说明：
  - 检查除数不为0
  - 在RV64中，两个源操作数和两个结果均为64位
  - 如除数和被除数都是正数，那商和余数也都是非负的（可以为0）
  - 对于有符号数的除法
    - 使用绝对值进行计算
    - 根据需要调整商和余数的符号

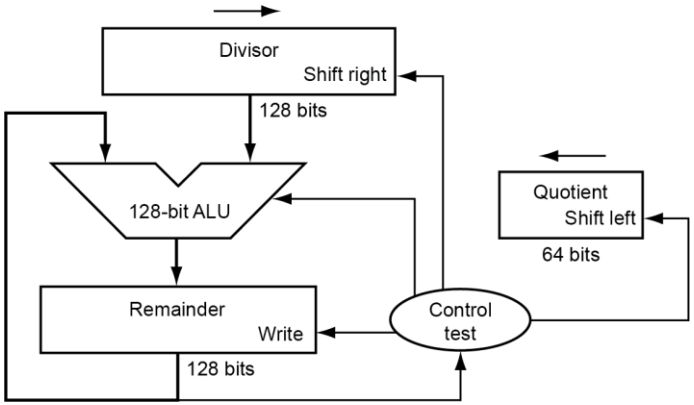
3.4 除法

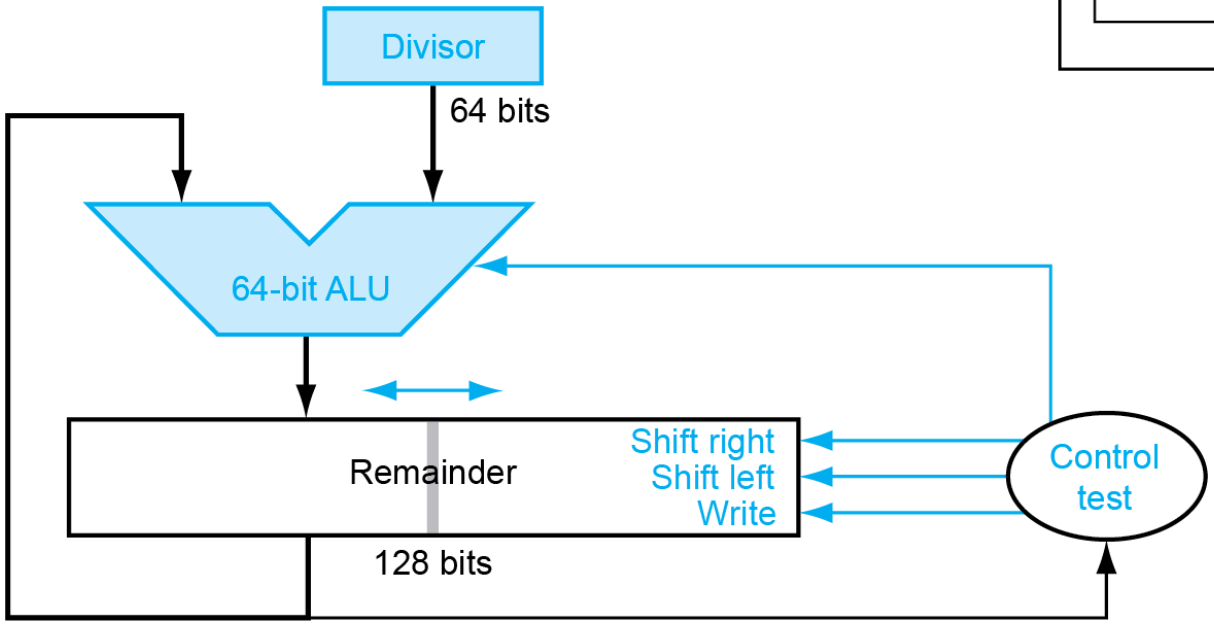

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
65th repetition?
No: < 65 repetitions
Yes: 65 repetitions
Done


Initially divisor in left half
Divisor
Shift right
128 bits
128-bit ALU
Quotient
Shift left
64 bits
Remainder
Write
Control test
128 bits
Initially dividend

3.4 除法

- 改进电路
  - 除数寄存器和ALU由128位降为64位
  - 取消商寄存器，与余数寄存器合用
  - 余数寄存器由128位改为129位





- 快速除法
  - 可自行调研，如：SRT除法技术

3.5 浮点运算

- RV32F：单精度浮点指令
- RV32D：双精度浮点指令
- RISC-V遵从IEEE 754-2008浮点标准

- RV32FD使用单独的32个F寄存器
- RV32F使用F寄存器的低32位

Floating-Point Computation

float { add | subtract | multiply | divide | square root | minimum | maximum } { .single | .double }

float { – negative } multiply { add | subtract } { .single | .double }

float move to .single from .x register

float move to .x register from .single

Comparison

compare float { equals | less than | less than or equals } { .single | .double }

Load and Store

float { load | store } { word | doubleword }

Conversion

float convert to { .single | .double } from .word { – unsigned }

float convert to .word { – unsigned } from { .single | .double }

float convert to .single from .double

float convert to .double from .single

Other instructions

float sign injection { – negative | exclusive or } { .single | .double }

float classify { .single | .double }

63 … 32	31 … 0	
	f0 / ft0	FP Temporary
	f1 / ft1	FP Temporary
	f2 / ft2	FP Temporary
	f3 / ft3	FP Temporary
	f4 / ft4	FP Temporary
	f5 / ft5	FP Temporary
	f6 / ft6	FP Temporary
	f7 / ft7	FP Temporary
	f8 / fs0	FP Saved register
	f9 / fs1	FP Saved register
	f10 / fa0	FP Function argument, return value
	f11 / fa1	FP Function argument, return value
	f12 / fa2	FP Function argument
	f13 / fa3	FP Function argument
	f14 / fa4	FP Function argument
	f15 / fa5	FP Function argument
	f16 / fa6	FP Function argument
	f17 / fa7	FP Function argument
	f18 / fs2	FP Saved register
	f19 / fs3	FP Saved register
	f20 / fs4	FP Saved register
	f21 / fs5	FP Saved register
	f22 / fs6	FP Saved register
	f23 / fs7	FP Saved register
	f24 / fs8	FP Saved register
	f25 / fs9	FP Saved register
	f26 / fs10	FP Saved register
	f27 / fs11	FP Saved register
	f28 / ft8	FP Temporary
	f29 / ft9	FP Temporary
	f30 / ft10	FP Temporary
	f31 / ft11	FP Temporary
32	32	

3.5 浮点运算

- RV32F包含26条指令（左图）
- RV32D包含26条指令（右图）

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	010	rd	000011	I flw
imm[11:5]		rs2	rs1	010	imm[4:0]	010011	S fsw
rs3	00	rs2	rs1	rm	rd	100001	R4
rs3	00	rs2	rs1	rm	rd	100011	R4
rs3	00	rs2	rs1	rm	rd	100101	R4
rs3	00	rs2	rs1	rm	rd	100111	R4
0000000		rs2	rs1	rm	rd	101001	R fadd.s
0000100		rs2	rs1	rm	rd	101001	R fsub.s
0001000		rs2	rs1	rm	rd	101001	R fmul.s
0001100		rs2	rs1	rm	rd	101001	R fdiv.s
0001100		00000	rs1	rm	rd	101001	R fsqrt.s
0010000		rs2	rs1	000	rd	101001	R fsgnj.s
0010000		rs2	rs1	001	rd	101001	R fsgnjn.s
0010000		rs2	rs1	010	rd	101001	R fsgnjx.s
0010100		rs2	rs1	000	rd	101001	R fmin.s
0010100		rs2	rs1	001	rd	101001	R fmax.s
1100000		00000	rs1	rm	rd	101001	R fcvt.w.s
1100000		00001	rs1	rm	rd	101001	R
1110000		00000	rs1	000	rd	101001	R fmv.x.w
1010000		rs2	rs1	010	rd	101001	R feq.s
1010000		rs2	rs1	001	rd	101001	R flt.s
1010000		rs2	rs1	000	rd	101001	R fle.s
1110000		00000	rs1	001	rd	101001	R fclass.s
1101000		00000	rs1	rm	rd	101001	R fcvt.s.w
1101000		00001	rs1	rm	rd	101001	R
1111000		00000	rs1	000	rd	101001	R fmv.w.x

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	011	rd	000011	I fld
imm[11:5]		rs2	rs1	011	imm[4:0]	010011	S fsd
rs3	01	rs2	rs1	rm	rd	100001	R4
rs3	01	rs2	rs1	rm	rd	100011	R4
rs3	01	rs2	rs1	rm	rd	100101	R4
rs3	01	rs2	rs1	rm	rd	100111	R4
0000001		rs2	rs1	rm	rd	101001	R fadd.d
0000101		rs2	rs1	rm	rd	101001	R fsub.d
0001001		rs2	rs1	rm	rd	101001	R fmul.d
0001101		rs2	rs1	rm	rd	101001	R fdiv.d
0001101		00000	rs1	rm	rd	101001	R fsqrt.d
0010001		rs2	rs1	000	rd	101001	R fsgnj.d
0010001		rs2	rs1	001	rd	101001	R fsgnjn.d
0010001		rs2	rs1	010	rd	101001	R fsgnjx.d
0010101		rs2	rs1	000	rd	101001	R fmin.d
0010101		rs2	rs1	001	rd	101001	R fmax.d
0100000		00001	rs1	rm	rd	101001	R fcvt.s.d
0100001		00000	rs1	rm	rd	101001	R fcvt.d.s
1010001		Rs2	rs1	010	rd	101001	R feq.d
1010001		rs2	rs1	001	rd	101001	R flt.d
1010001		rs2	rs1	000	rd	101001	R fle.d
1110001		00000	rs1	001	rd	101001	R fclass.d
1100001		00000	rs1	rm	rd	101001	R fcvt.w.d
1100001		00001	rs1	rm	rd	101001	R
1101001		00000	rs1	rm	rd	101001	R fmv.d.w
1101001		00001	rs1	rm	rd	101001	R

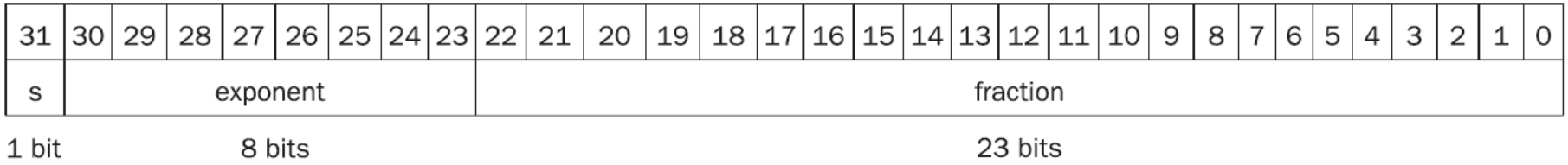
3.5 浮点运算

- 以科学计数法的形式表示二进制实数(IEEE 754标准)
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
  - 对应于C语言中的float（单精度）和double（双精度）类型

$$x = (-1)^S \times (1 + Fraction) \times 2^{(Exponent - Bias)}$$

single: 8 bits　　single: 23 bits
double: 11 bits　　double: 52 bits

  - S：符号位，0为正，1为负
  - F：尾数，单精度23位，双精度52位，无符号
  - E：指数，单精度8位，双精度11位，无符号
  - Bias：偏移值，单精度为127，双精度为1023



3.5 浮点运算

- 浮点表示实例：-0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - 符号位：S = 1
  - 尾数：Fraction = $1000...00_2$
  - 指数：Exponent = –1 + Bias
    - 单精度：–1 + 127 = 126 = 01111110_2
    - 双精度：–1 + 1023 = 1022 = 01111111110_2

- -0.75单精度表示： 10111111010000...00
- -0.75双精度表示： 10111111111010000...00

- 浮点加法：略
- 浮点乘法：略

3.6 并行性与计算机算术：子字并行

- 略

3.7 实例：x86中的SIMD扩展和高级向扩展

- 略

3. 8. 加速：子字并行和矩阵乘法

- 略

3. 9. 谬误与陷阱

- 略

3. 10. 本章小结

- 略