Computer Organization and Design

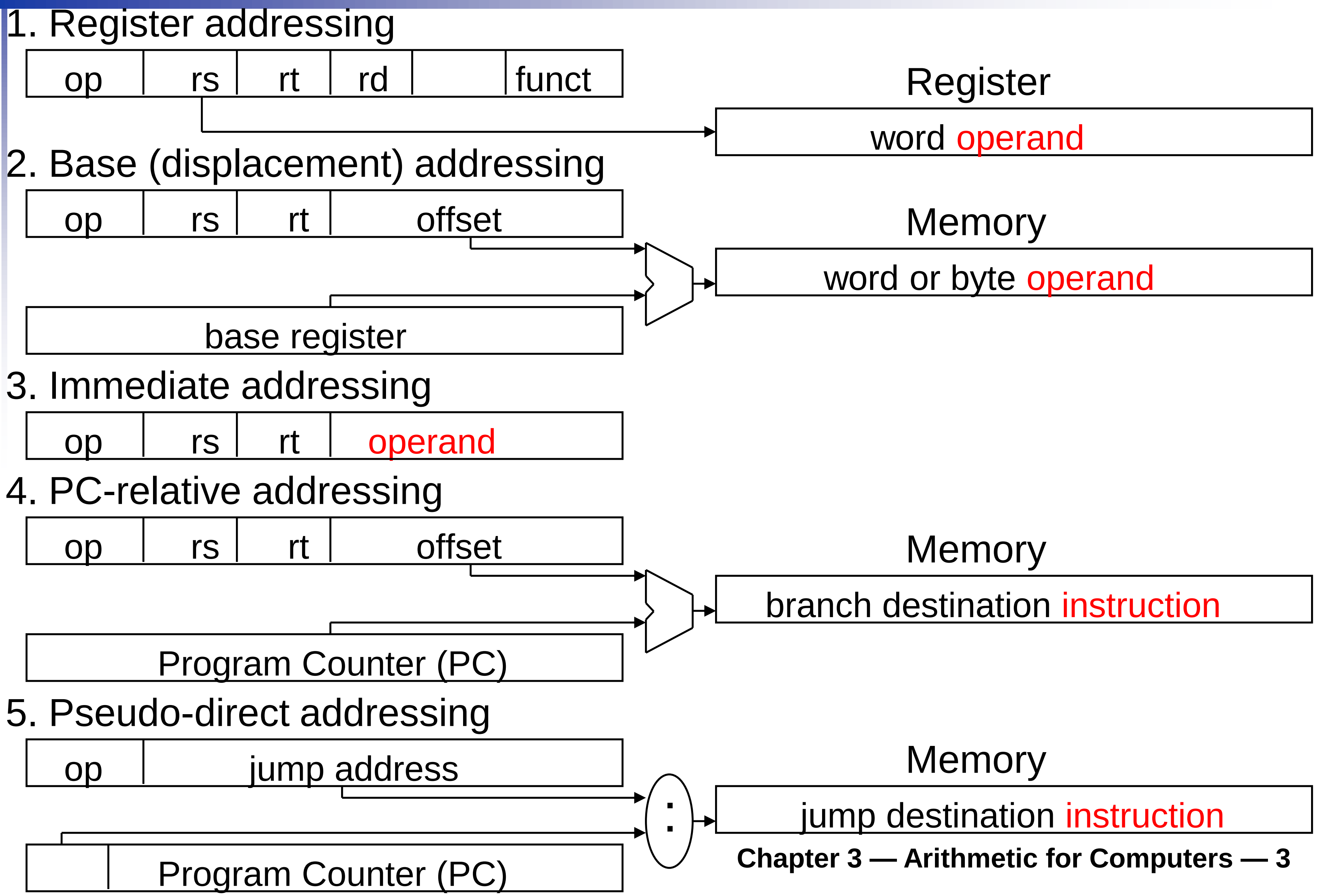
Arithmetic for Computers

Jiang Zhong
zhongjiang@cqu.edu.cn

Review: MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

Review: MIPS Addressing Modes



Review: Number Representations

- 32-bit signed numbers (2's complement):

maxint

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$

...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$

...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$

minint

MSB $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

LSB

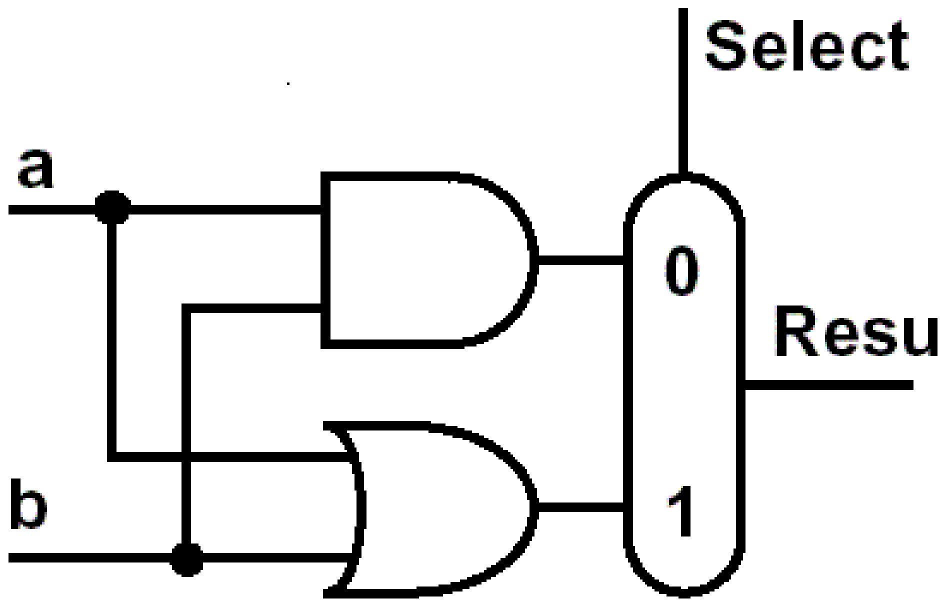
- Converting <32-bit values into 32-bit values
  - copy the most significant bit (the sign bit) into the "empty" bits

```
0010   -> 0000 0010
1010   -> 1111 1010
```

  - sign extend   versus   zero extend   (`lb` vs. `lbu`)

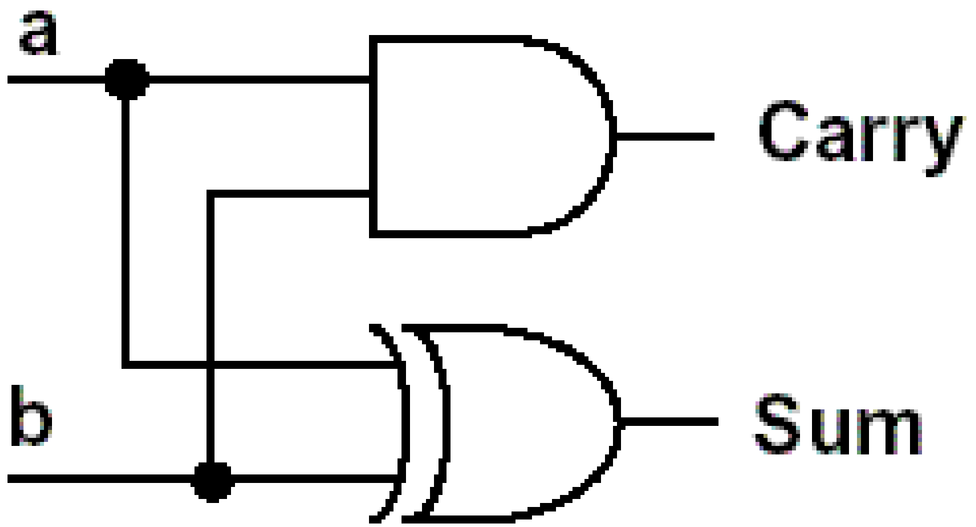
Constructing an ALU

- Step by step: build a single bit ALU and expand it to the desired width
- First function: logic AND and OR



A half adder

- Sum = $\bar{a}\,b + a\,\bar{b}$
- Carry = a b



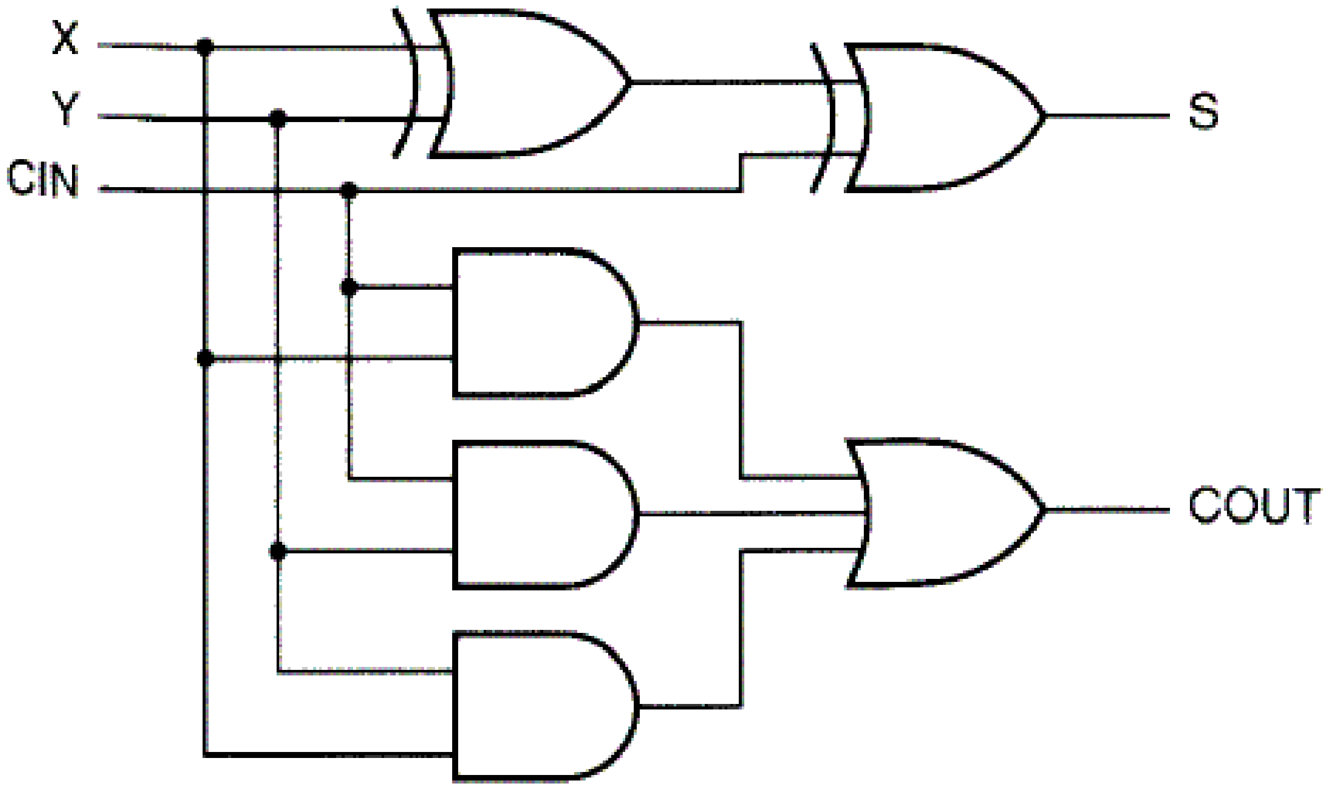
A full adder

- Accepts a carry in
- Sum = A xor B xor CarryIn
- CarryOut = B CarryIn + A CarryIn + A B

Inputs			Outputs		Comments
A	B	CarryIn	CarryOut	Sum	(two)
0	0	0	0	0	0+0+0=00
0	0	1	0	1	0+0+1=01
0	1	0	0	1	0+1+0=01
0	1	1	1	0	0+1+1=10
1	0	0	0	1	1+0+0=01
1	0	1	1	0	1+0+1=10
1	1	0	1	0	1+1+0=10
1	1	1	1	1	1+1+1=11

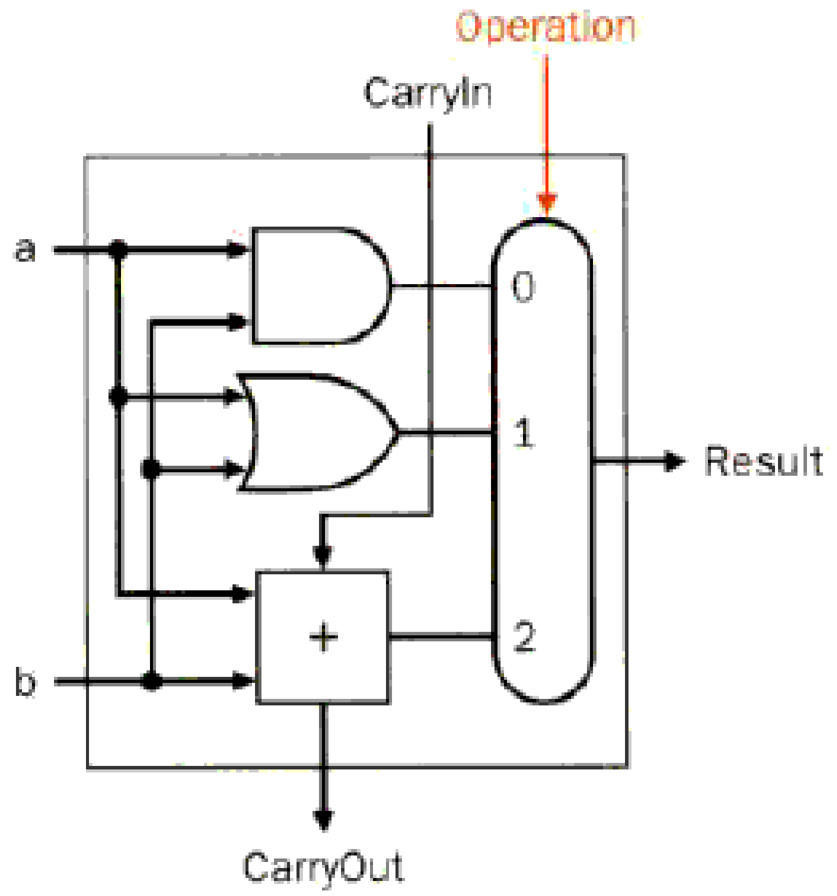
Full adder

- Full adder in 2-level design



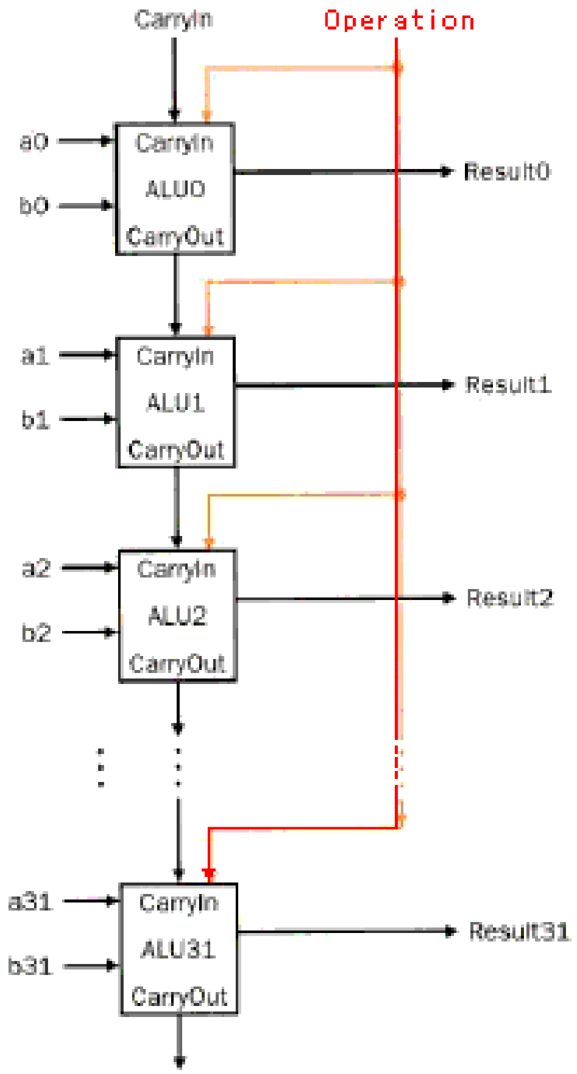
1 bit ALU

- ALU
  - AND
  - OR
  - ADD
- Cascade Element



Basic 32 bit ALU

- Inputs parallel
- Carry is cascaded
- Ripple carry adder
- Slow, but simple
- 1st Carry In = 0

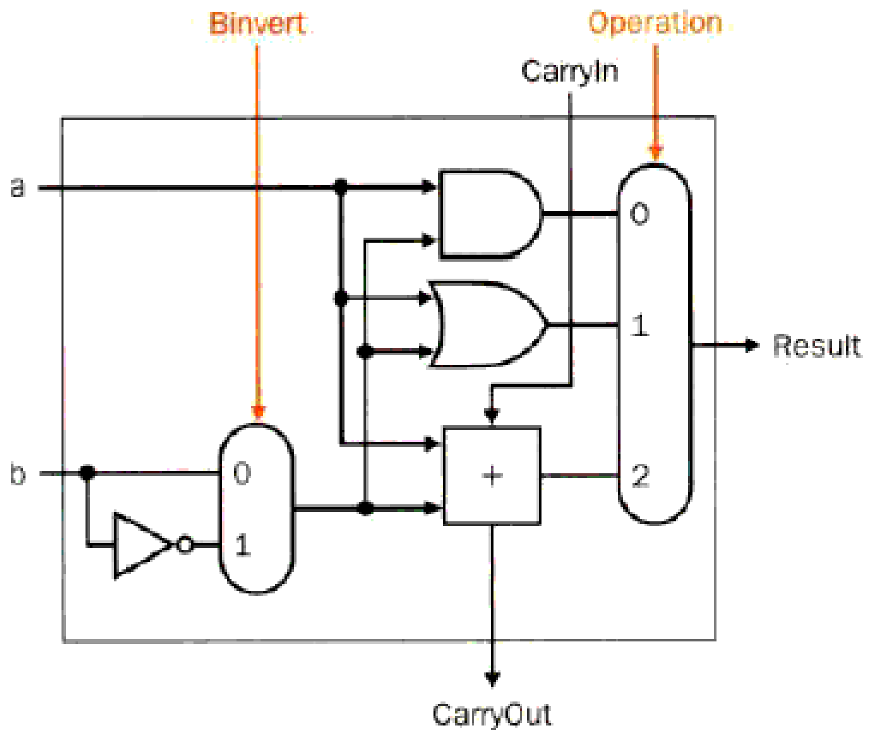


Extended 1 bit ALU

- Subtraction

  a - b

  - Inverting b
  - 1st CarryIn= 1

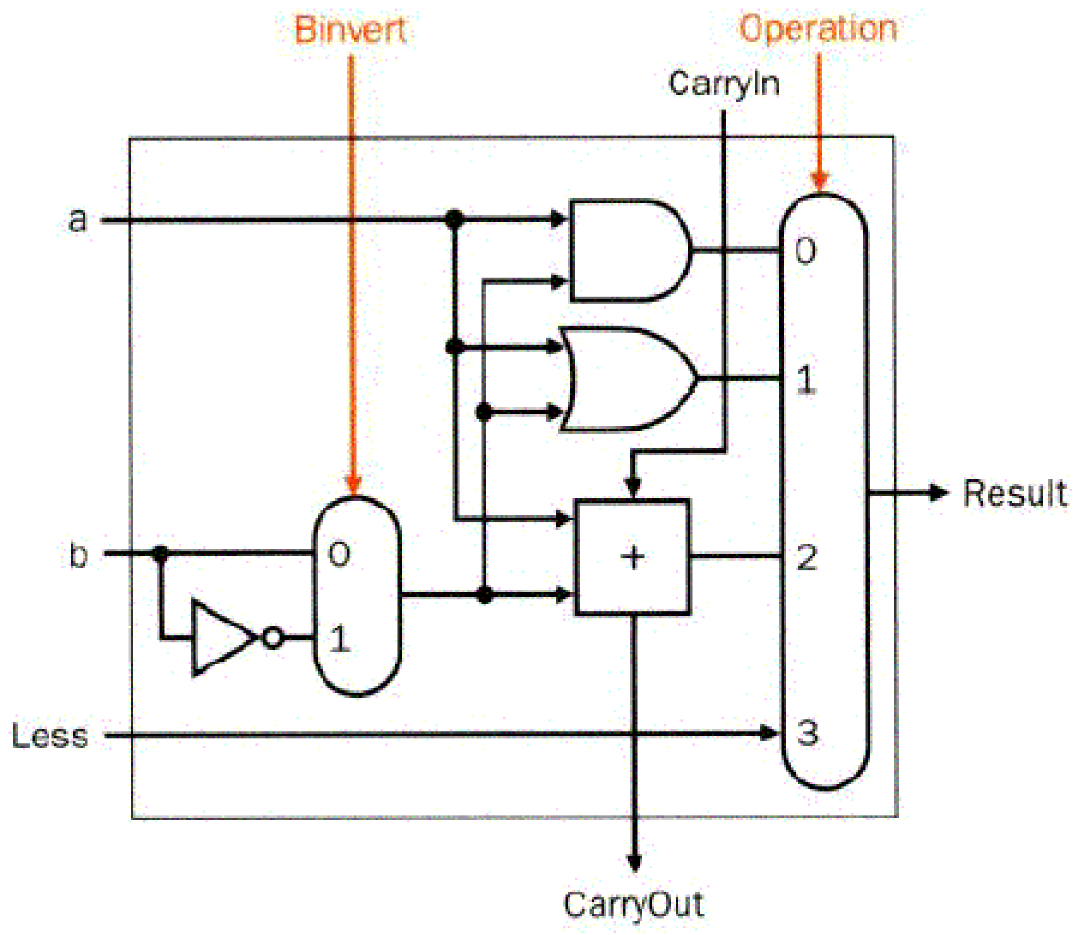


Extended 1 bit ALU

- Functions
  - AND
  - OR
  - Add
  - Subtract
- Missing: comparison
  - Slt rd,rs,rt
  - If rs < rt, rd=1, else rd=0
  - All bits = 0 except the least significant
  - Subtraction (rs - rt), if the result is negative -> rs < rt
  - Use of sign bit as indicator

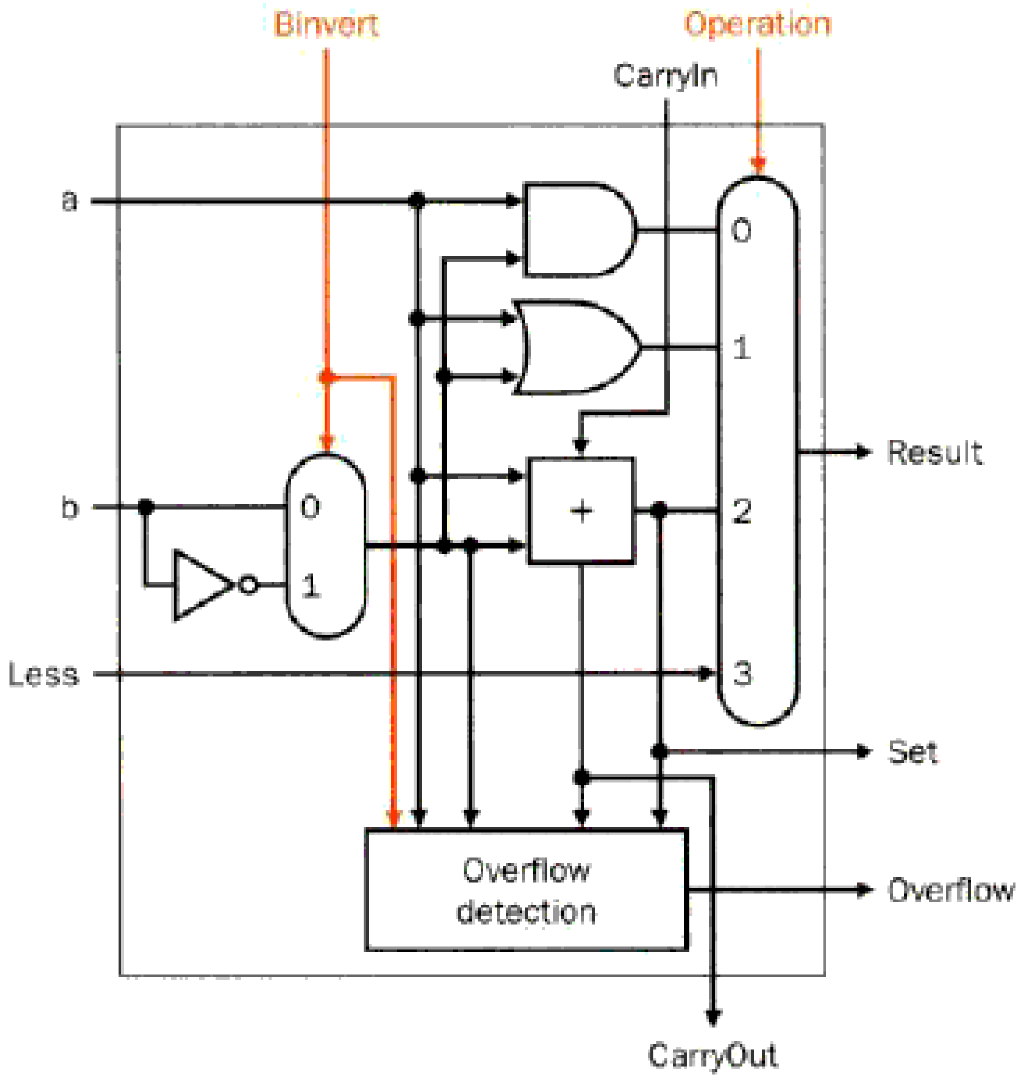
Extended 1 bit ALU

- ALU bit with input for Less data



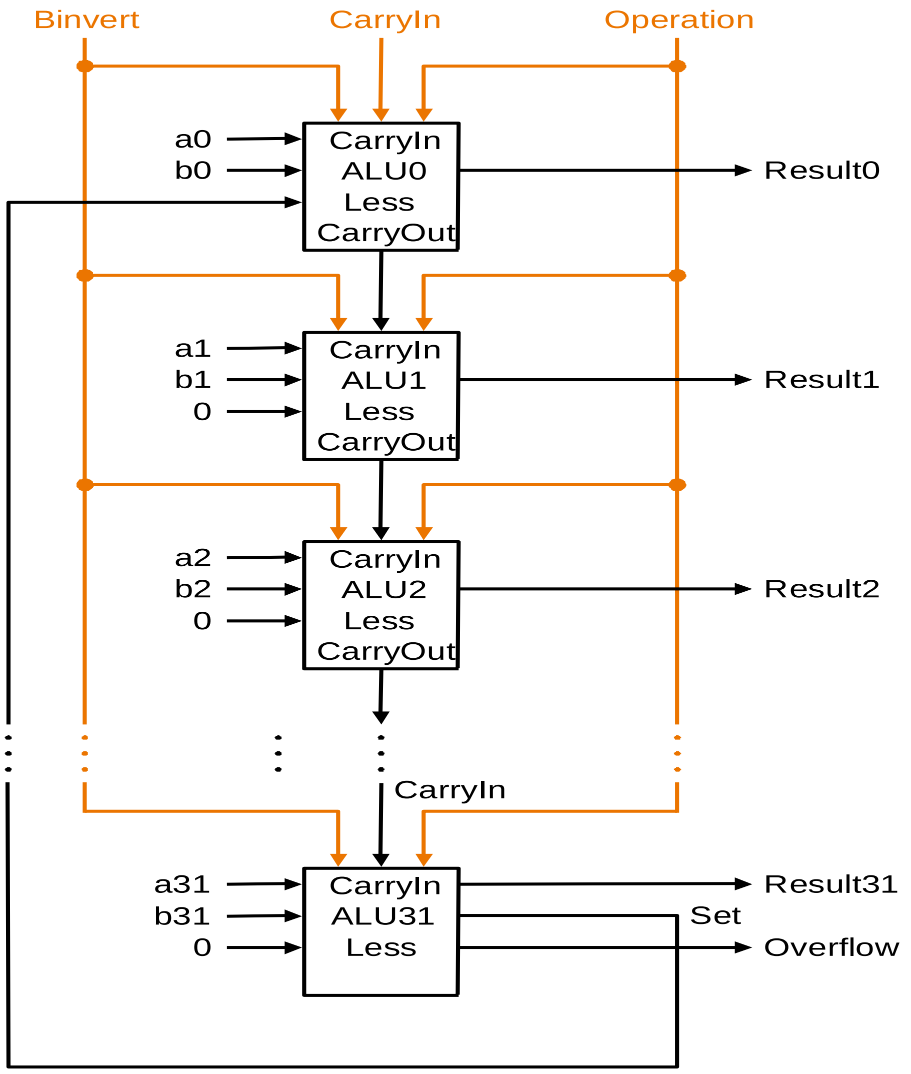
Most significant bit

- Set for comparison
- Overflow detect



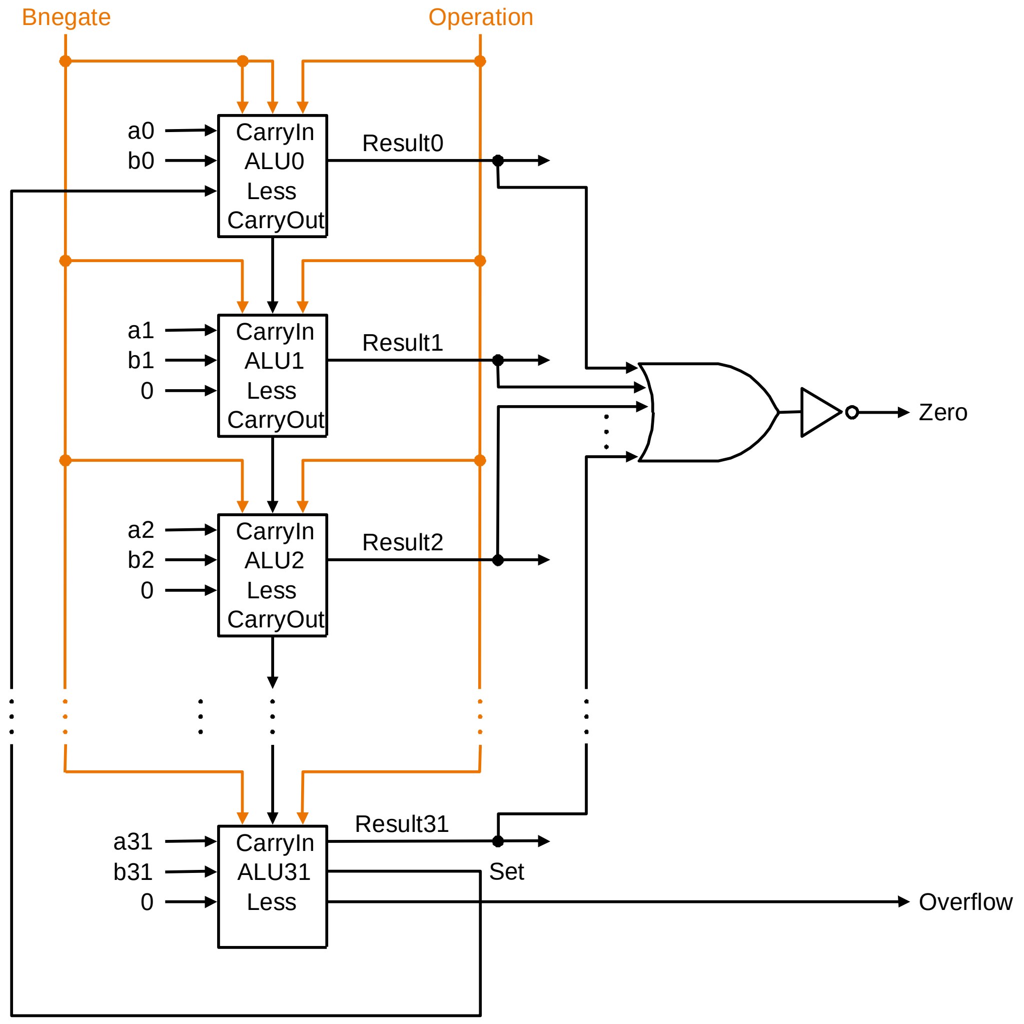
Complete ALU

- Input
  - A
  - B
- Control lines
  - Binvert
  - Operation
  - Carryin
- Output
  - Result
  - Overflow



Complete ALU

- Add a Zero detector

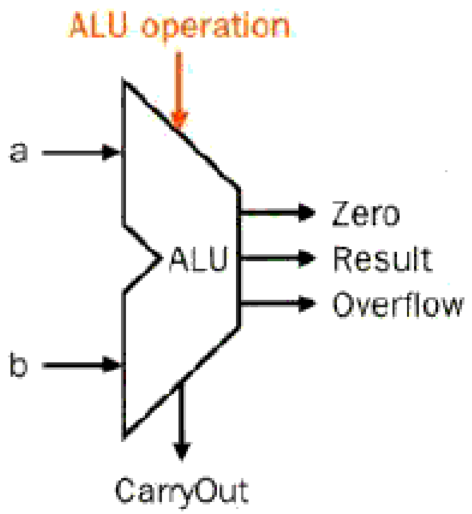


ALU symbol & control

- Function table

ALU Control Lines	Function
000	And
001	Or
010	Add
110	Sub
111	Set on less than

- Symbol of the ALU

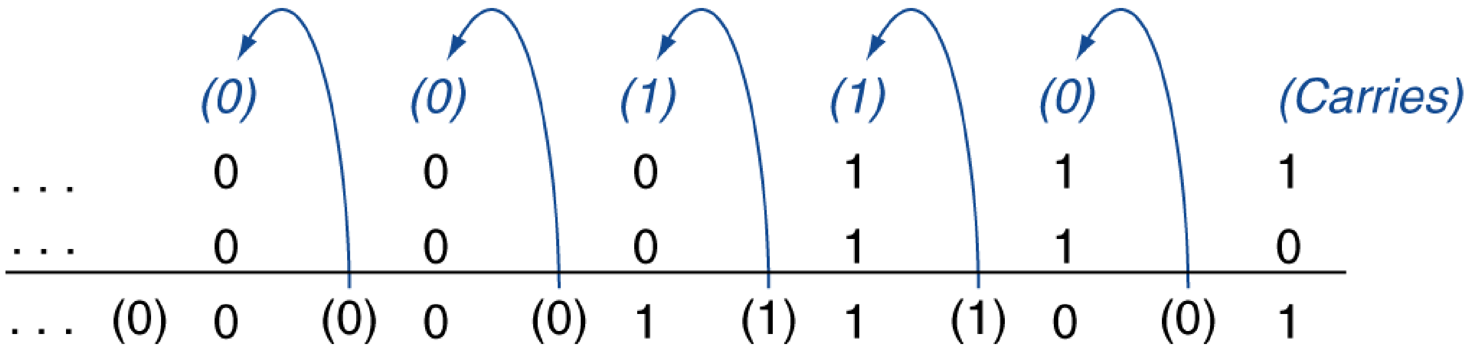


Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
- What about fractions and real numbers?
  - Representation and operations
- How are overflow scenarios handled?
  - e.g. An operation creates a number bigger than can be represented
- How does hardware really multiply and divide numbers?

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign bit is 1
  - Adding two –ve operands
    - Overflow if result sign bit is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

```
+7:     0000 0000 … 0000 0111
–6:     1111 1111 … 1111 1010
+1:     0000 0000 … 0000 0001
```

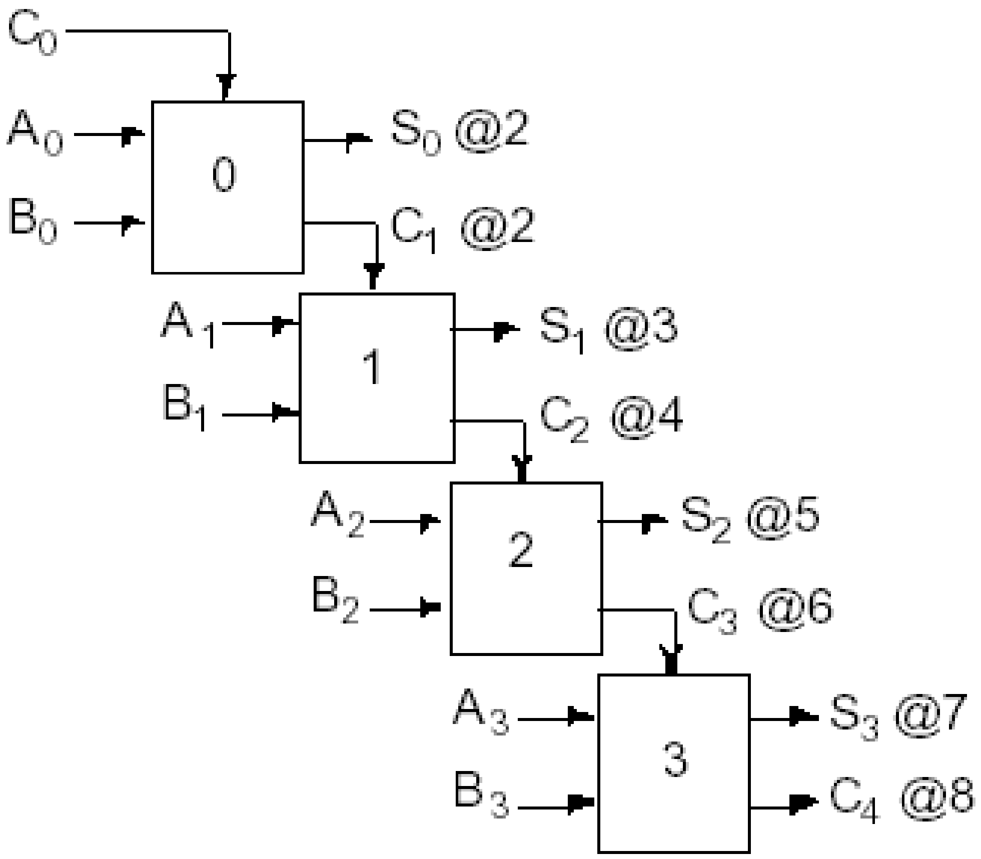
- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign bit is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign bit is 1

Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - C compilers use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

Speed considerations

- Delay of one adder
  - 2 time units
- Total delay for stages: 2n unit delays
- Not appropriate for high speed application



Fast adders

- All functions can be represented in 2-level logic.
- But:
  - The number of inputs of the gates would drastically rise
- Target:

  Optimum between speed and size

Fast adders

- Carry look-ahead adder
  - Calculating the carries before the sum is ready
- Carry skip adder
  - Accelerating the carry calculation by skipping some blocks
- Carry select adder
  - Calculate two results and use the correct one
- ...

Carry look ahead adder (CLA)

- Separation of
  - add operation
  - carry calculation
- Factorisation
  - $C_{i+1} = (b_i * c_i) + (a_i * c_i) + (a_i * b_i)$

    $= (a_i * b_i) + (a_i + b_i) * c_i$
  - Generate $g_i = a_i * b_i$
  - Propagate $p_i = a_i + b_i$

Carry look ahead adder

- Ci+1 = gi + pi * ci
- Carry generate: gi = ai * bi
  - If a and b are '1' ->

    we always have a carryout independent of ci
- Carry propagate: pi = ai+ bi
  - If only one of a and b is '1' ->

    the carry out depends on the carry in
  - pi propagates the carry

Four bit carry look ahead adder

- c1 = g0 + (p0 * c0)
- c2 = g1 + p1*c1 = g1 + (p1 * g0) + (p1 * p0 * c0)
- c3 = g2 + p2*c2 = g2 + (p2 * g1) + (p2 * p1 * g0) + (p2 * p1 * p0 * c0)
- c4 = g3 + p3*c3 = g3 + (p3 * g2) + (p3 * p2 * g1)
  +(p3 * p2 * p1 * g0) + (p3 * p2 * p1 * p0 * c0)

COMMENT:

This kind of adder will be faster than the ripple carry adder, and smaller than the adder with the tow-level logic.

PROBLEM:

If the number of the adder bits is very large, this kind of adder will be too large. So we must seek more efficient ways.

Four bit carry look ahead adder

Let's consider a 16-bit adder.

Divide 16 bits into 4 groups.Each group has 4 bits.

As we know:

$c4 = g3 + p3*g2 + p3*p2*g1 + p3*p2*p1*g0 + p3*p2*p1*p0*c0$

So,we can get the following:

$c8 = g7 + p7*g6 + p7*p6*g5 + p7*p6*p5*g4 + p7*p6*5*p4*c4$

$c12 = g11+p11*g10+p11*p10*g9+p11*p10*p9*g8+p11*p10*p9*p8*c8$

$c16=g15+p15*g14+p15*p14*g13+p15*p14*p13*g12+p15*p14*p13*p12*c12$

Assume:

$G0= g3 + p3*g2 + p3*p2 *g1 +p3*p2*p1*g0$

$G1= g7 + p7*g6 + p7*p6*g5 + p7*p6*p5*g4$

$G2= g11+p11*g10+p11*p10*g9+p11*p10*p9*g8$

$G3= g15+p15*g14+p15*p14*g13+p15*p14*p13*g12$

$P0= p3 * p2 * p1 * p0$

$P1= p7 * p6 * p5 * p4$

$P2= p11 * p10 * p9 * p8$

$P3= p15 * p14 * p13 * p12$

Four bit carry look ahead adder

Then we get:

$c4=G0+P0*c0$; $c8=G1+P1*c4$

$c12=G2+P2*c8$; $c16=G3+P3*c12$

Assume:$C1=c4,C2=c8,C3=c12,C4=c16$

Then:

$C1=G0+P0*c0$; $C2=G1+P1*C1$

$C3=G2+P2*C2$; $C4=G3+P3*C3$

And,we can further get:

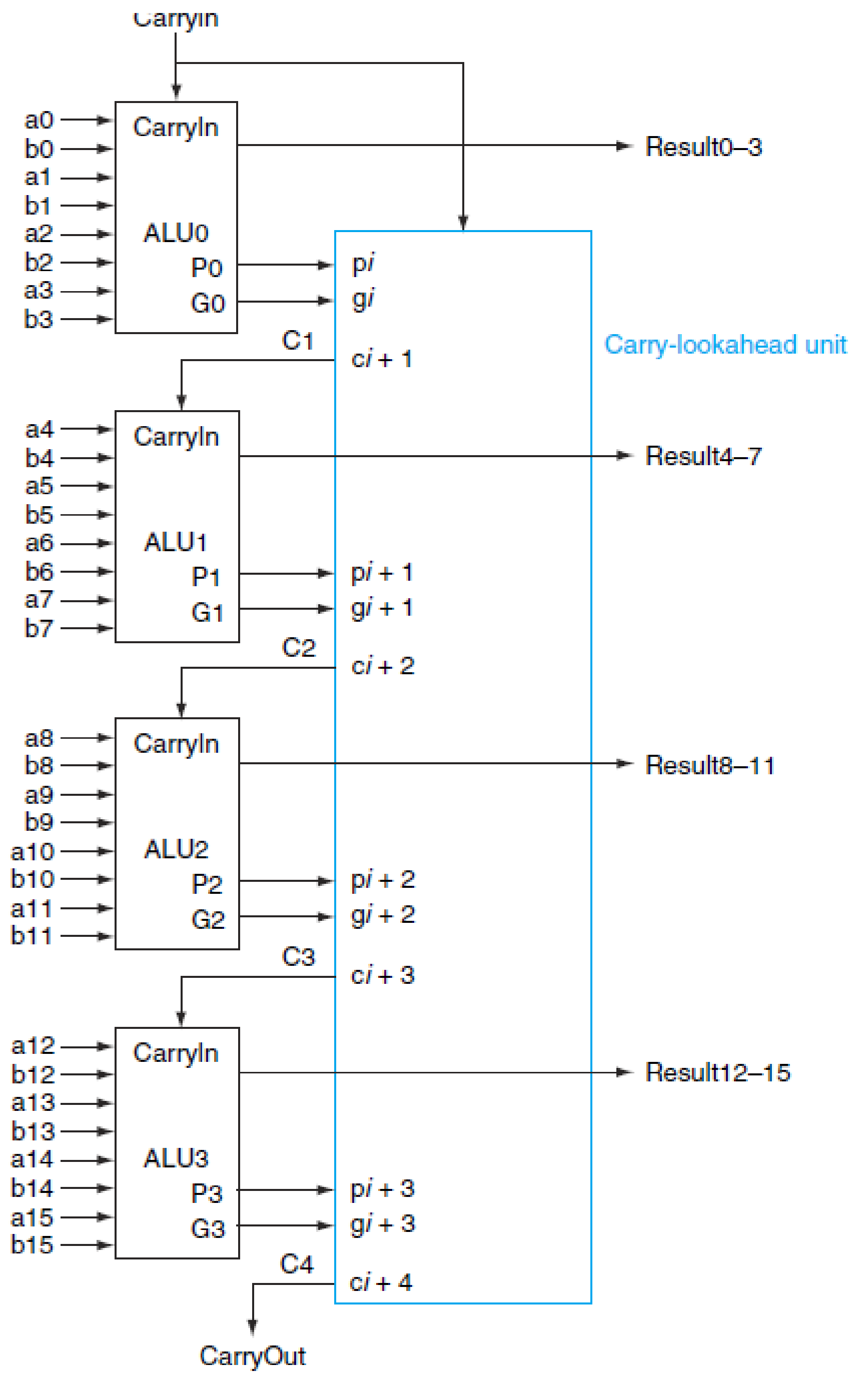
$C1=G0+P0*c0$;

$C2= G1+P1*C1 = G1+P1*G0+ P1* P0*c0$

$C3=G2+P2*C2 = G2+P2*G1 + P2* P1*G0+ P2* P1* P0*c0$

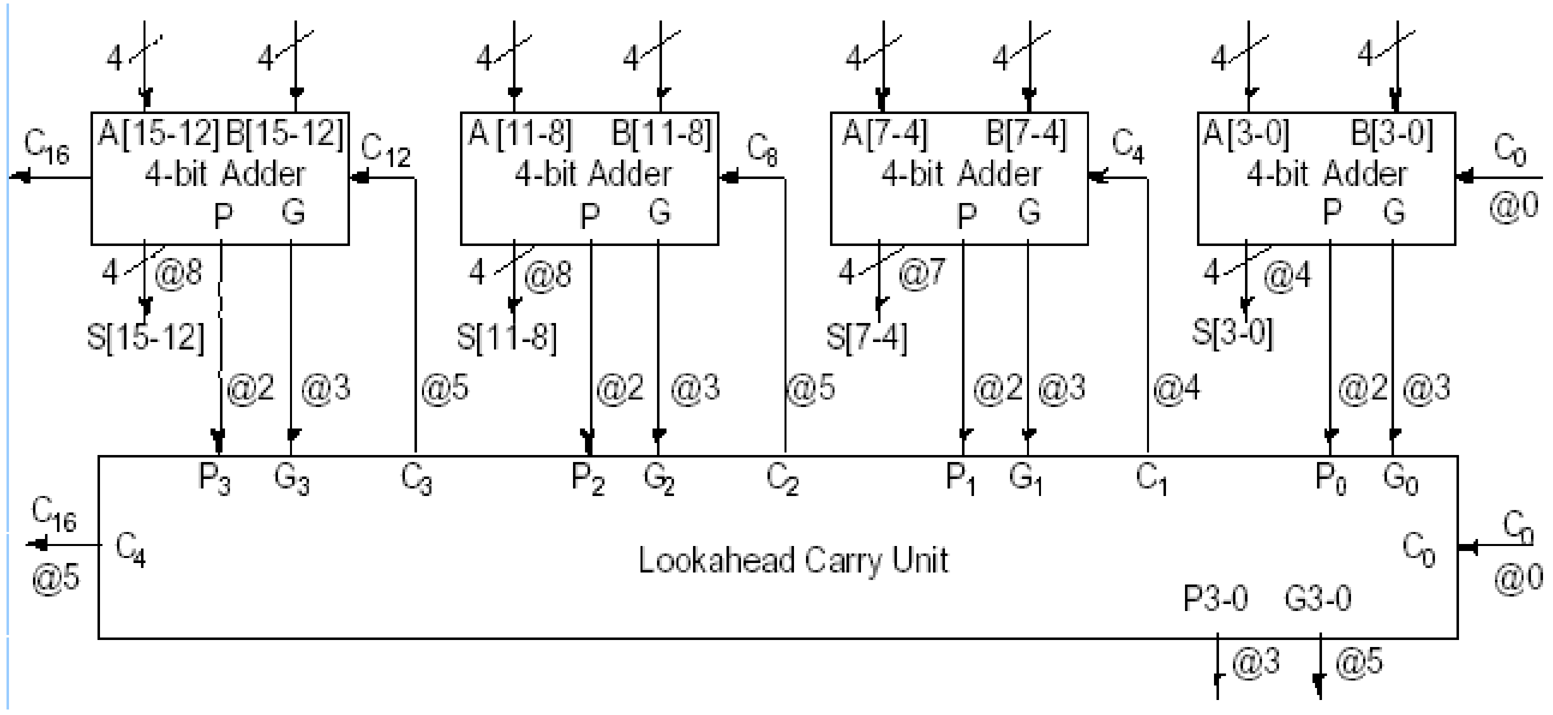
$C4=G3+P3*C3= G3+P3*G2+P3*P2*G1+P3*P2*P1*G0+P3*P2*P1* P0*c0$

Four 4-bit ALUs using carry lookahead to form a 16-bit adder.

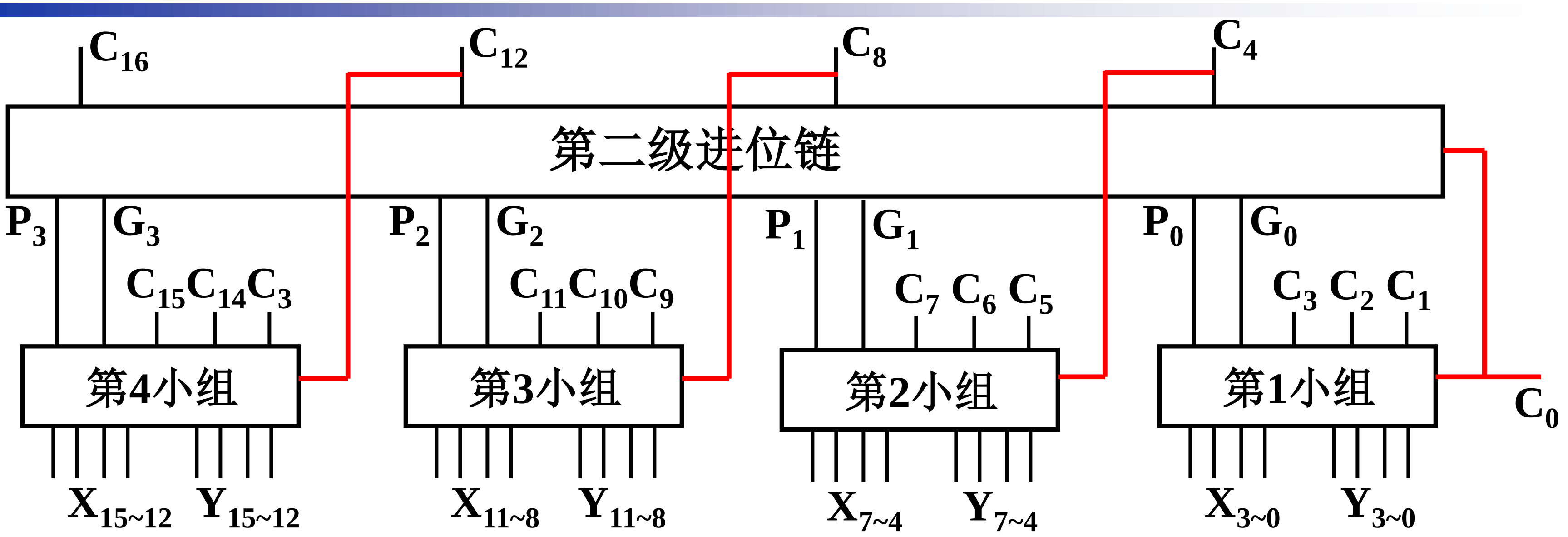


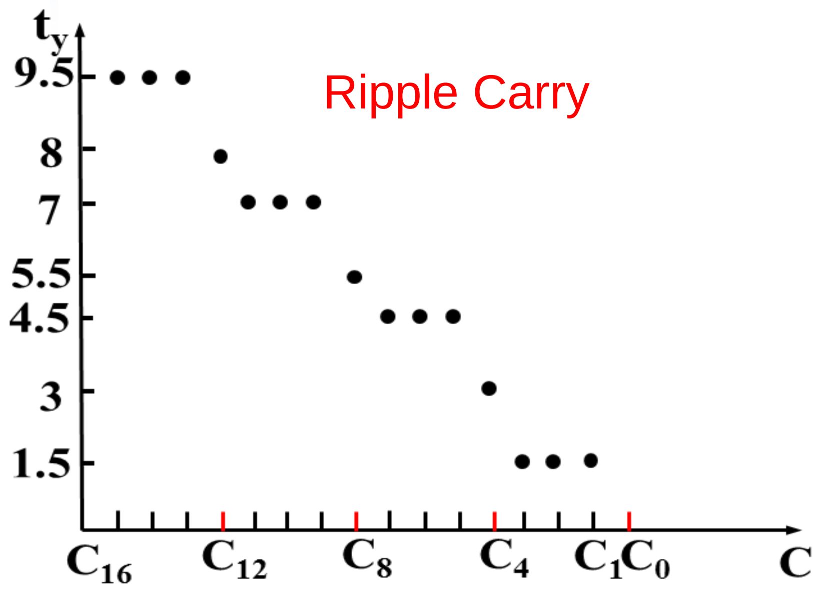
Hybrid CLA + Ripple carry

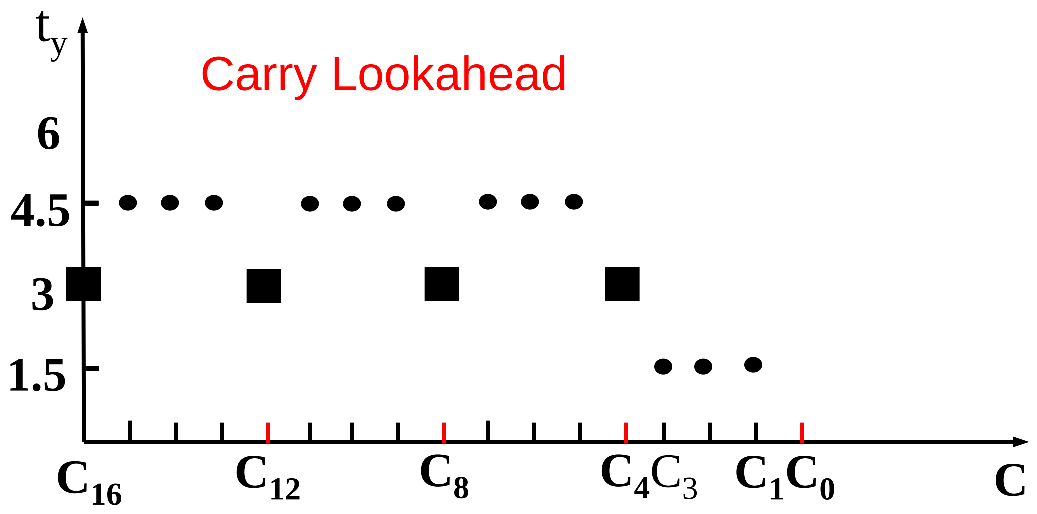
- Realisation:
  - Ripple carry adders and



Suppose Time (AND) =0.5 T , Time(Or)=1.0 T

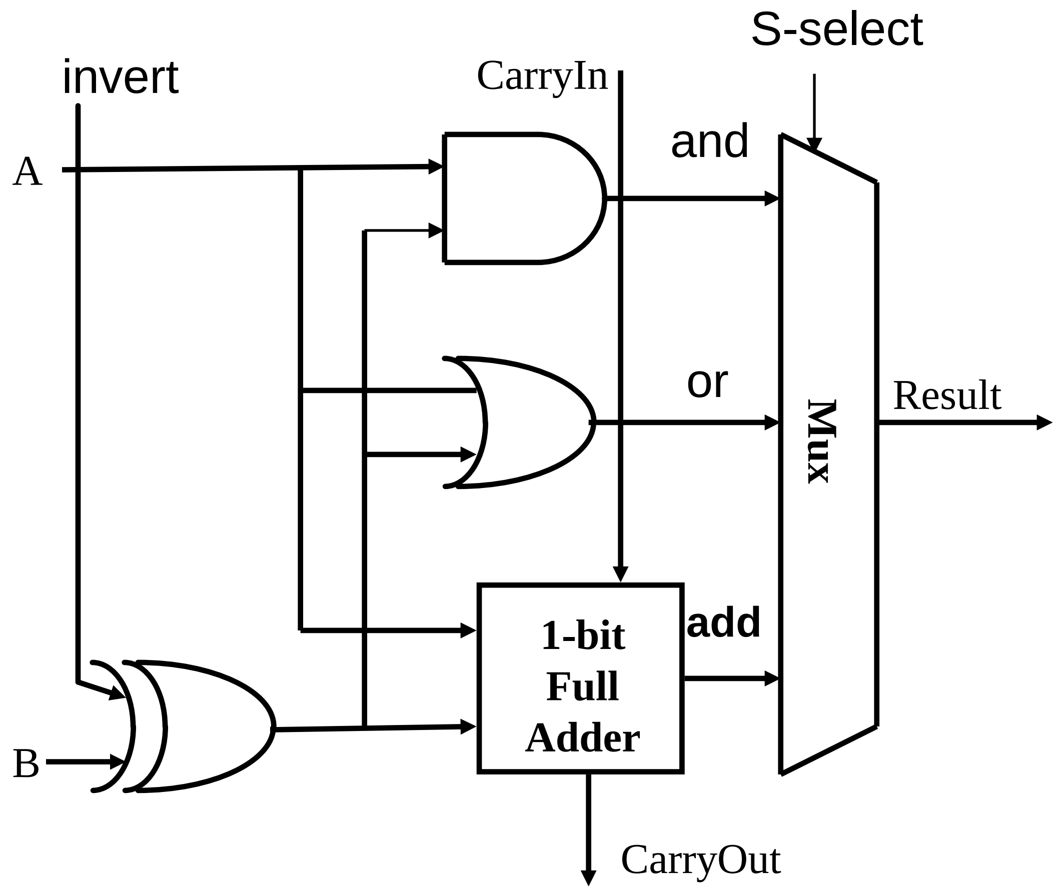






Complete ALU

- A - B = A + (– B)
  - form two complement by invert and add one

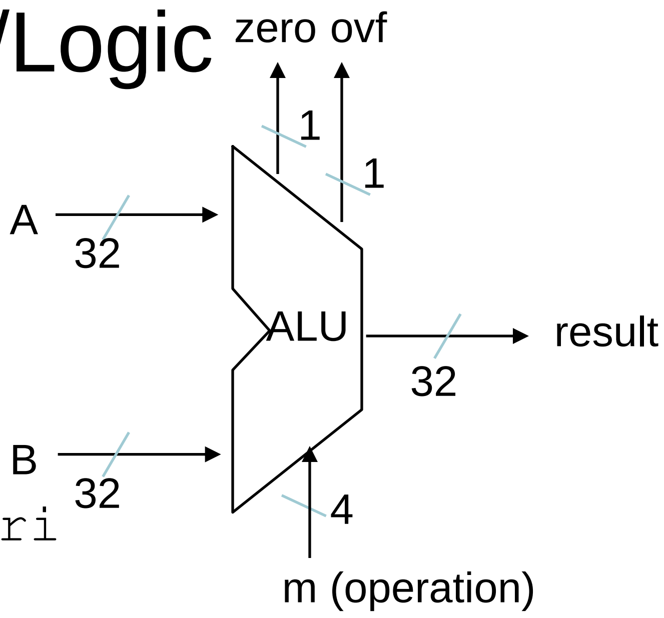


Set-less-than? – left as an exercise

MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

```
add, addi, addiu, addu
sub, subu
mult, multu, div, divu
sqrt
and, andi, nor, or, ori, xor, xori
beq, bne, slt, slti, sltiu, sltu
```

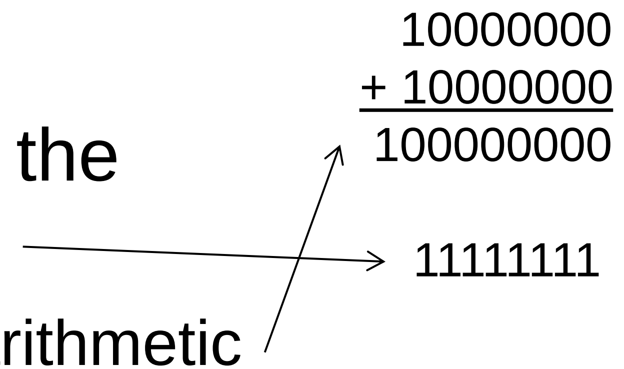


- With special handling for
  - sign extend – `addi, addiu, slti, sltiu`
  - zero extend – `andi, ori, xori`
  - overflow detection – `add, addi, sub`

Review Appendix C (from CD or lecture page) for more details on ALU design

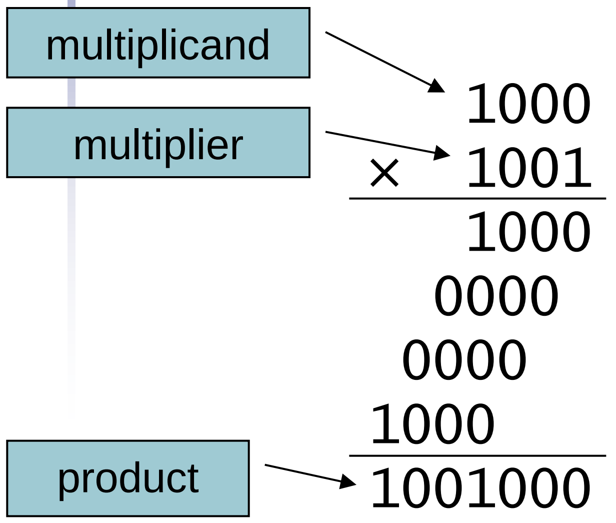
Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is set to the largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

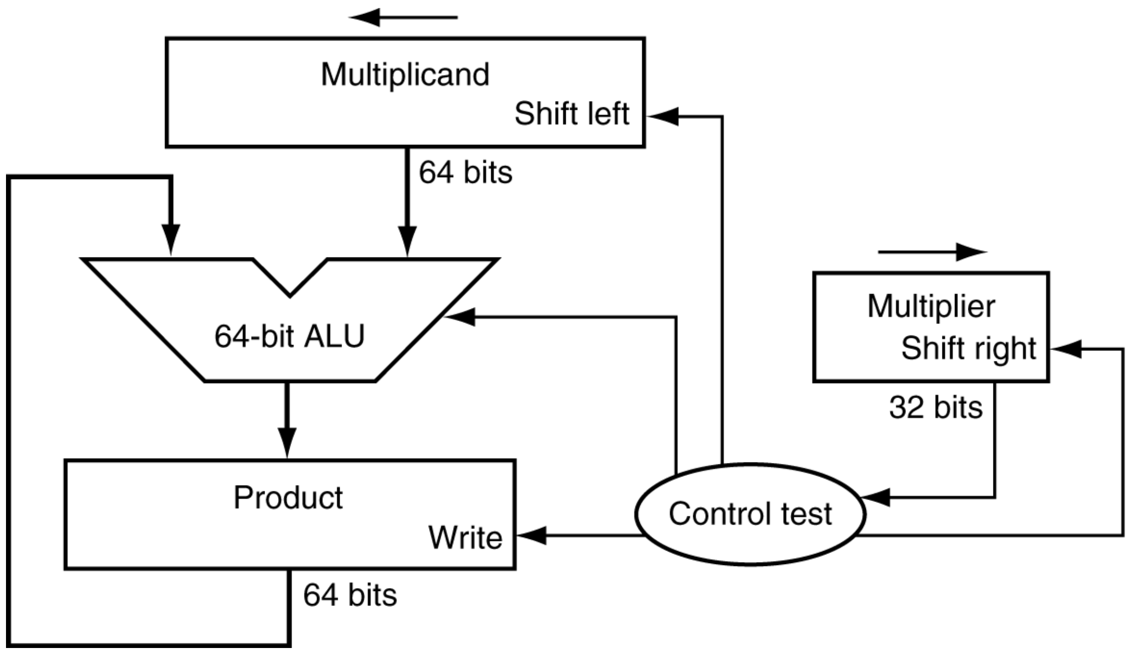


Multiplication

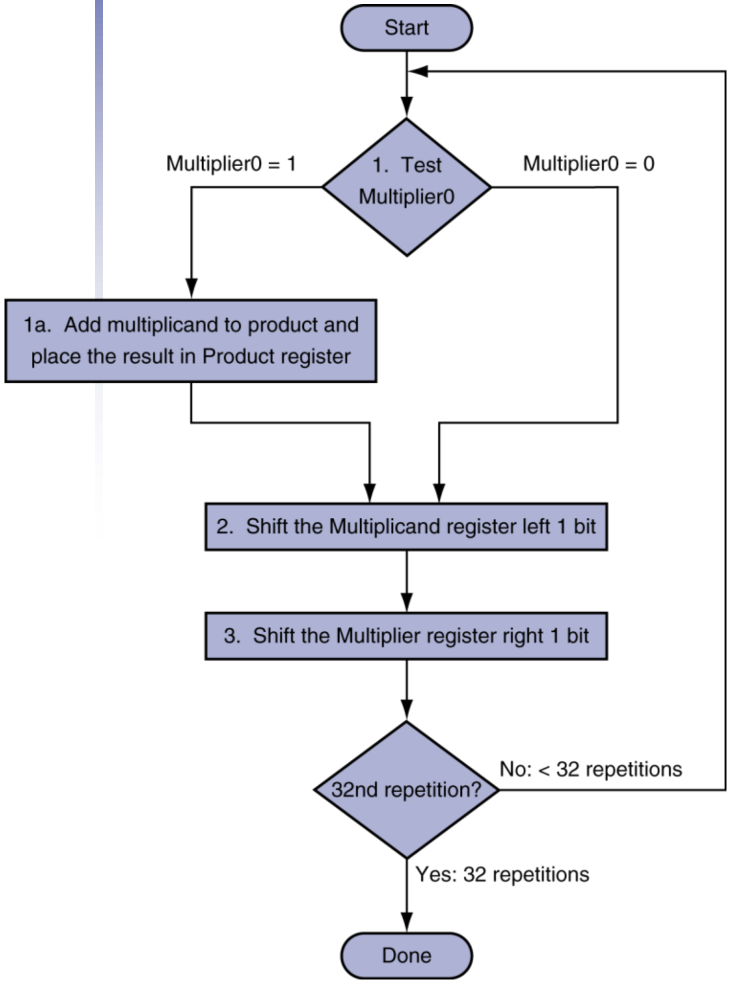
- Start with long-multiplication approach



Length of product is the sum of operand lengths

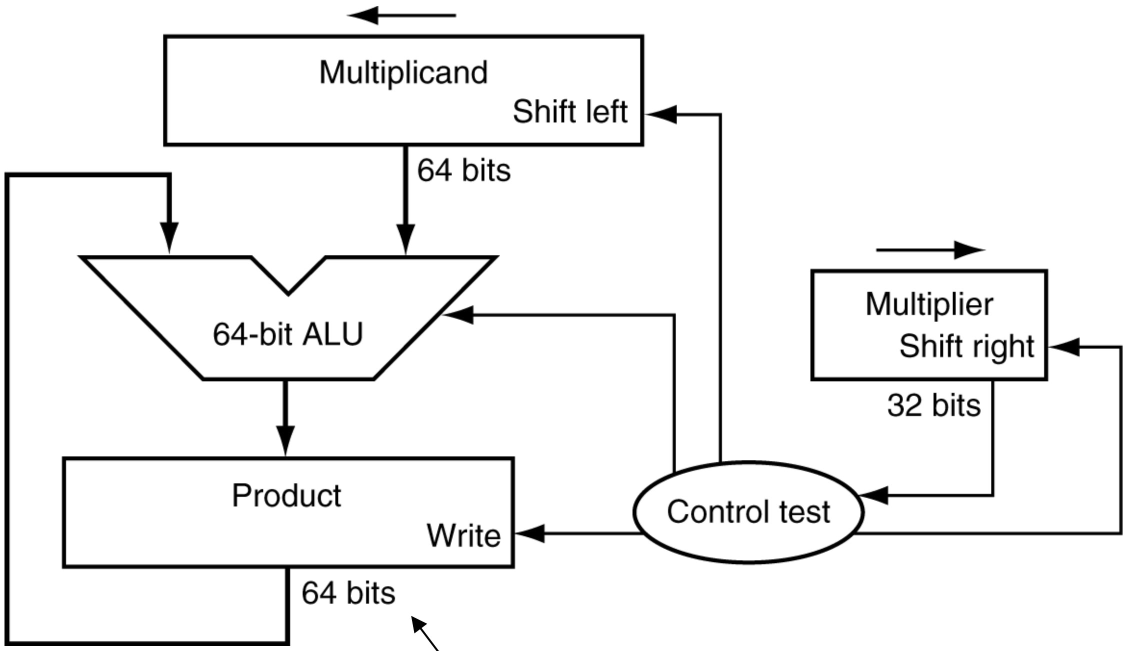


Multiplication Hardware


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Sequential Implementation

~100 cycles for multiplying 2 32-bit values



Initially 0

the recurrence formula for product

let z_i denotes the ith product, we could get the recurrence formula

$$
\begin{aligned}
z_0 &= 0,\\
z_1 &= 2^{-1}(y_n x+z_0)\\
z_2 &= 2^{-1}(y_{n-1}x+z_1)\\
&\vdots\\
z_i &= 2^{-1}(y_{n-i+1}x+z_{i-1})\\
&\vdots\\
z_n &= x\cdot y = 2^{-1}(y_1 x+ z_{n-1})
\end{aligned}
$$

特点：每次只需要相加两个数，然后右移一位。且相加的两个数（部分积和位积）都只有n位，因而不需要2n位的加法器。

Let x,y are fractional fixed-point both x and y less then 1.

$x=0.x_1x_2......x_n <1$

$y=0.y_1y_2......y_n <1$

The product of x and y is

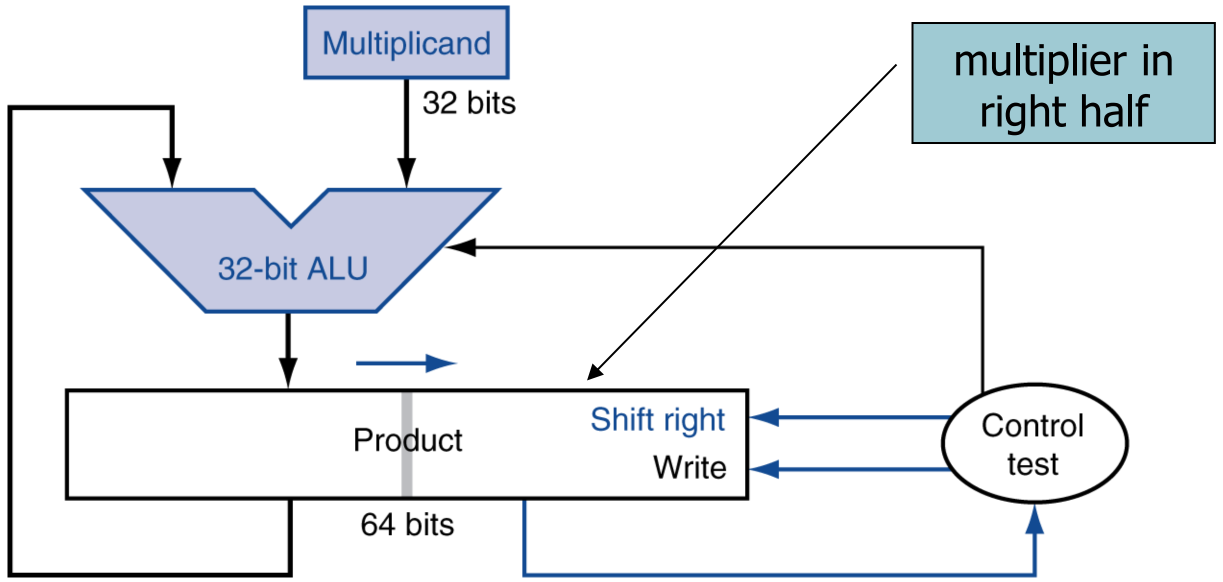
$$x \cdot y=x(0.y_1y_2......y_n)$$

$$=x(y_1 2^{-1}+y_2 2^{-2}+\ldots+y_n 2^{-n})$$

$$=2^{-1}(y_1x+2^{-1}(y_2x+2^{-1}(\ldots+2^{-1}(y_{n-1}x+2^{-1}(y_nx+0))\ldots)))$$

Optimized Multiplier

- Perform steps in parallel: add/shift



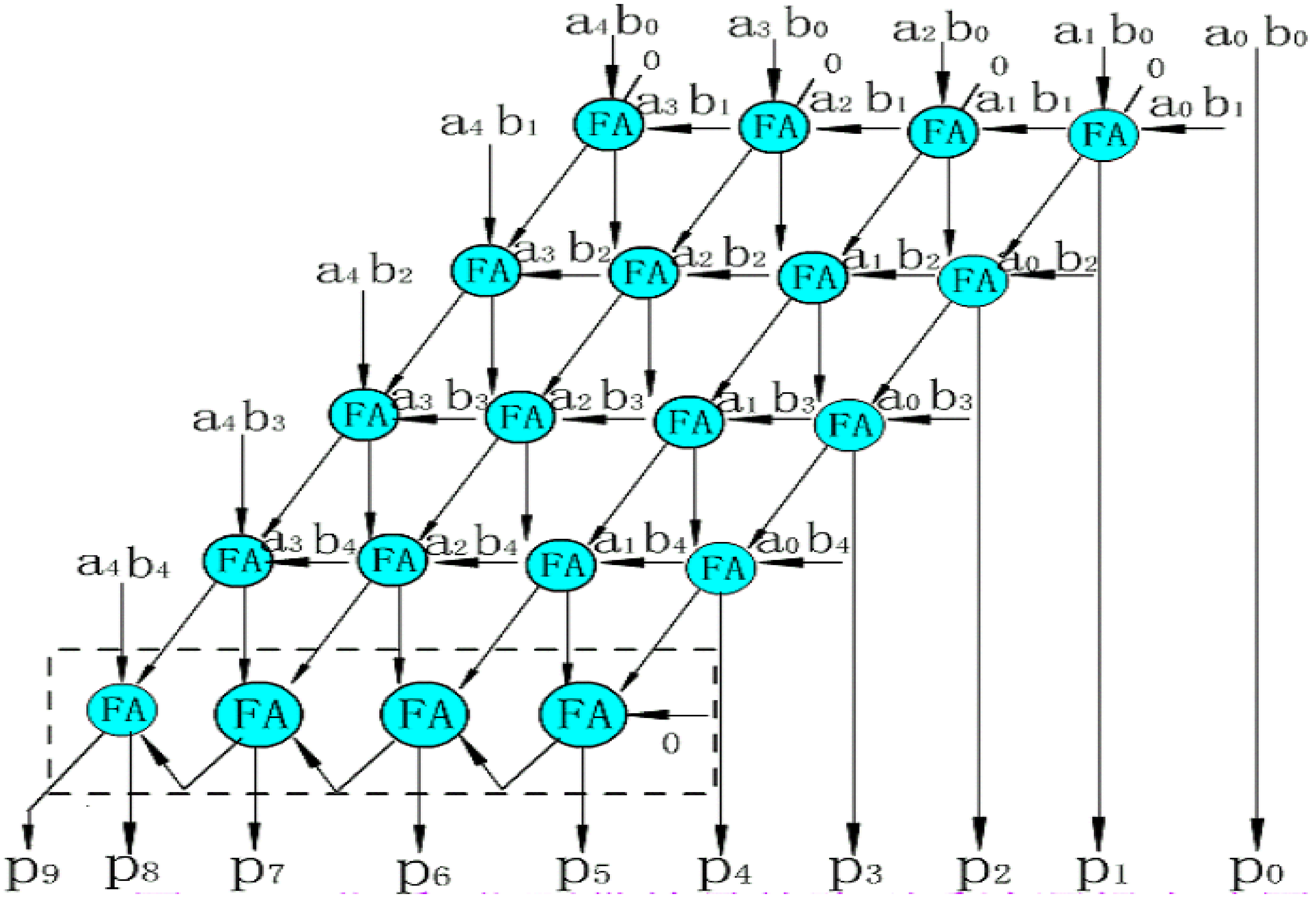
- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low
  - ~32 cycles for multiplying 2 32-bit values

例：x=0.1101，y=0.1011，求 x·y 。

部分积	乘数	说明
0 0.0 0 0 0	y_f 1 0 1 1	$z_0=0$
+ 0 0.1 1 0 1		$y_4=1$, +x
0 0.1 1 0 1		
→ 0 0.0 1 1 0	1 y_f 1 0 1	右移，得z_1
+ 0 0.1 1 0 1		$y_3=1$, +x
0 1.0 0 1 1		
→ 0 0.1 0 0 1	1 1 y_f 1 0	右移，得z_2
+ 0 0.0 0 0 0		$y_2=0$, +0
0 0.1 0 0 1		
→ 0 0.0 1 0 0	1 1 1 y_f 1	右移，得z_3
+ 0 0.1 1 0 1		$y_1=1$, +x
0 1.0 0 0 1		
→ 0 0.1 0 0 0	1 1 1 1 y_f	右移，得$z_4=x\cdot y$

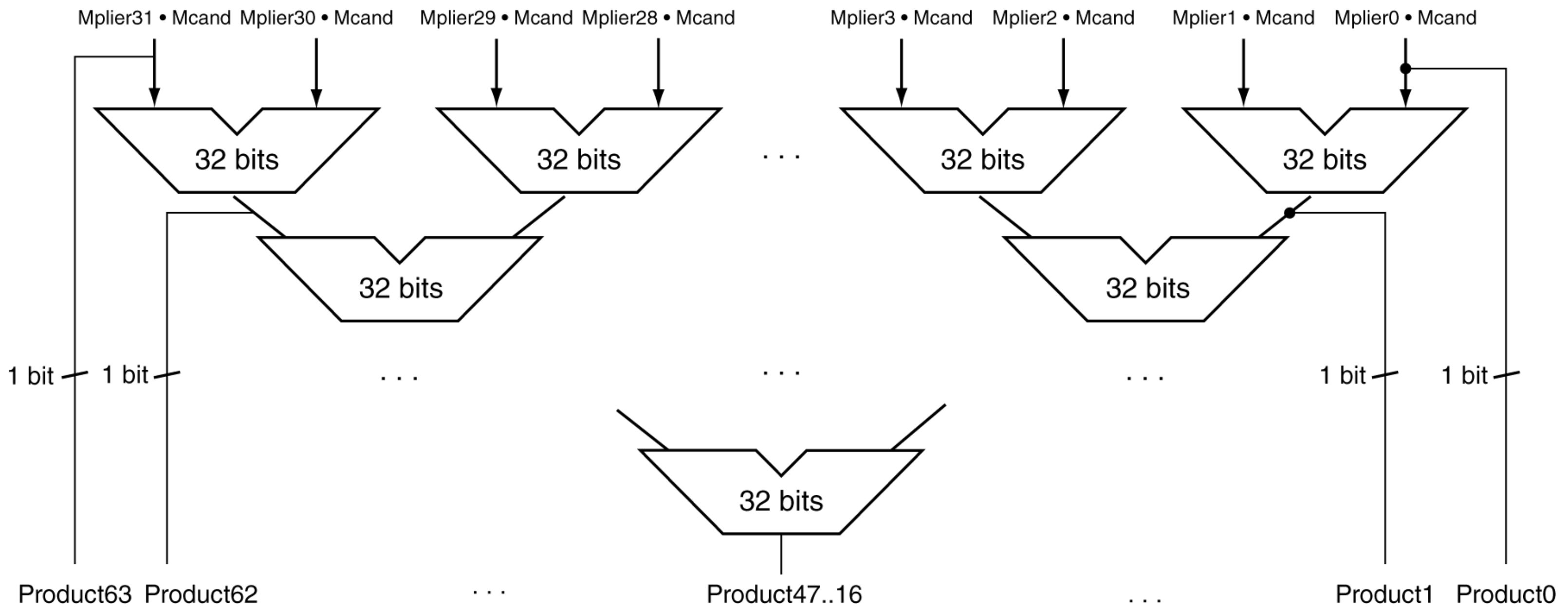
最后结果： $x\cdot y=0.10001111$

Even Faster Multiplier



Even Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff
    - 31 adders → $\log_2(31)$ = ~5 cycles for multiplying 2 32-bit values

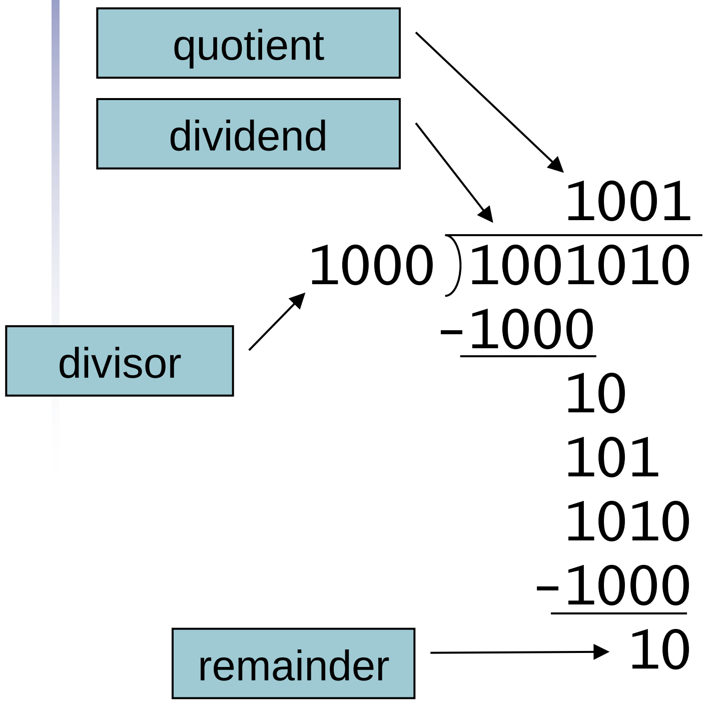


- Can be pipelined
  - Several multiplication performed in parallel

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

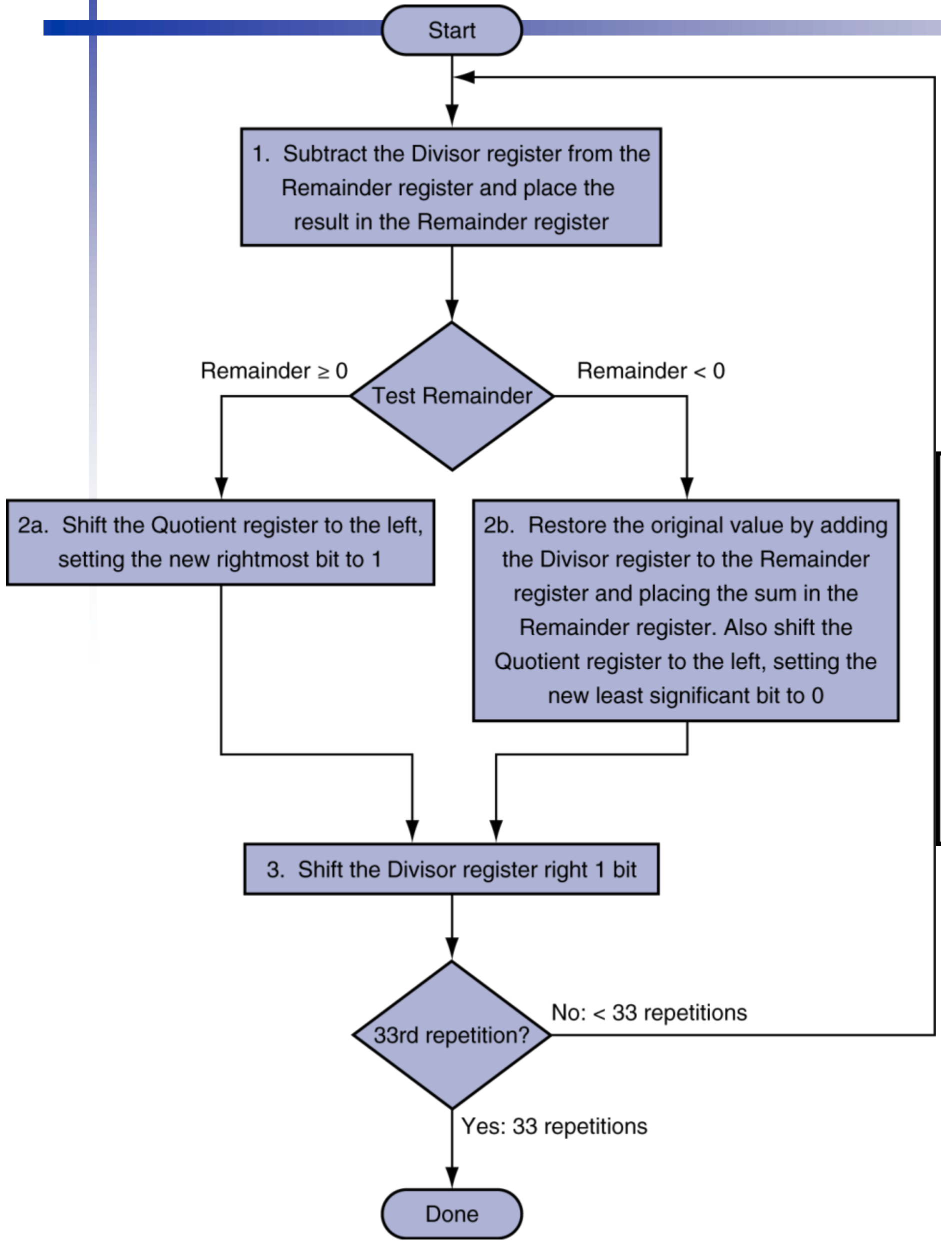
Division



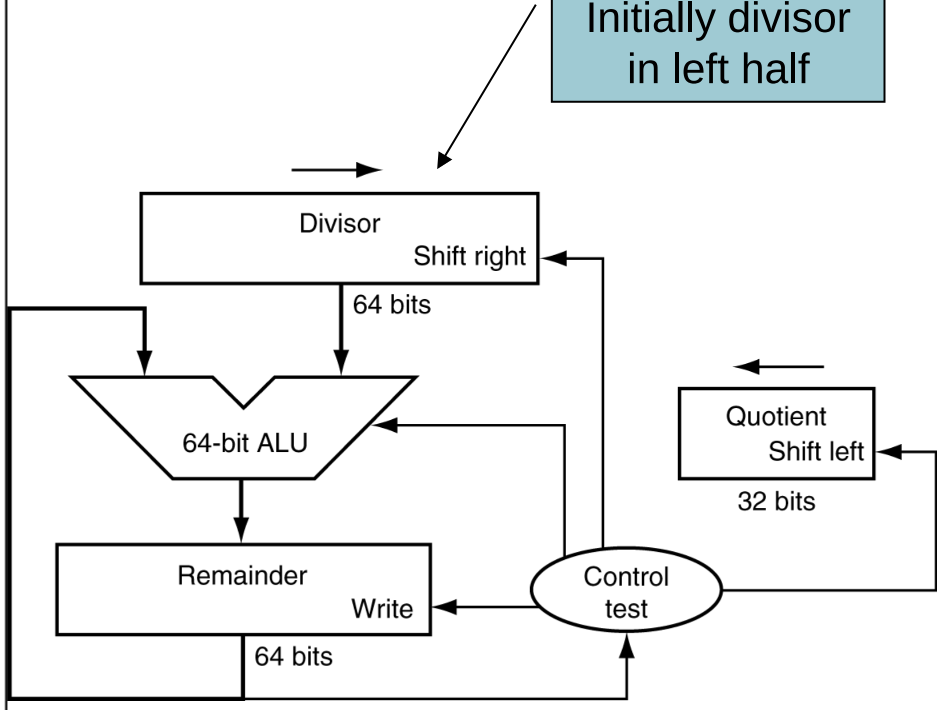
n-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Division Hardware



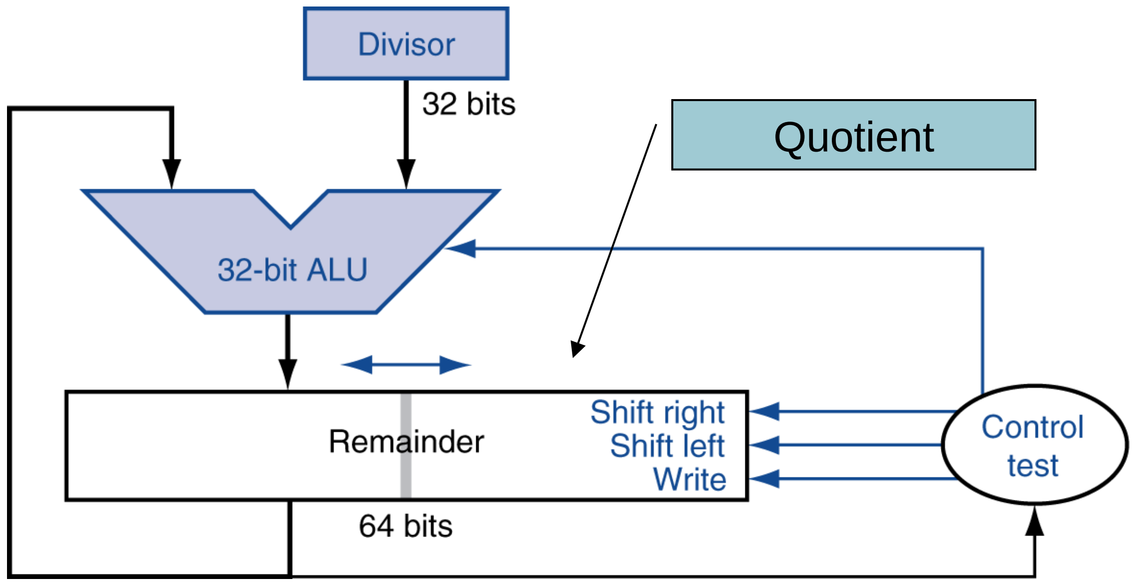
Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to 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
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



Initially divisor in left half

Initially dividend

Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

$[x]_{原} = 0 .1001$

$[y]_{补}=0.1011$

$[-y]_{补}=1.0101$

	被除数x / 余数 r	商q	说明
	0 0.1 0 0 1		
$+[-y]_{补}$	1 1.0 1 0 1		x减y
	1 1.1 1 1 0		余数 $r_0<0$，商“0”
$+ [y]_{补}$	0 0.1 0 1 1		恢复余数
	0 0.1 0 0 1		r_0'
←	0 1.0 0 1 0	0	商0移入q，r_0'左移
$+[-y]_{补}$	1 1.0 1 0 1		减y
	0 0.0 1 1 1		$r_1>0$，商“1”
←	0 0.1 1 1 0	0. 1	商1移入q，r_1左移
$+[-y]_{补}$	1 1.0 1 0 1		减y
	0 0.0 0 1 1		$r_2>0$，商“1”
←	0 0.0 1 1 0	0. 1 1	商1移入q，r_2左移
$+[-y]_{补}$	1 1.0 1 0 1		减y
	1 1.1 0 1 1		$r_3<0$，商“0”
$+[y]_{补}$	0 0.1 0 1 1		恢复余数
	0 0.0 1 1 0		$r_3'=2r_2$
←	0 0.1 1 0 0	0. 1 1 0	商0移入q，r_3'左移
$+[-y]_{补}$	1 1.0 1 0 1		减y
	0 0.0 0 0 1		$r_4>0$，商“1”
←	0 0.0 0 0 1	0. 1 1 0 1	商1移入q，r_4不左移

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  - Guesses quotient bits using a table lookup based on upper bits of dividend, remainder
    - Subsequent steps correct wrong guesses
  - Still require multiple steps
- Other faster dividers exist
  - nonrestoring dividers, nonperforming dividers, …

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt` / `divu rs, rt`
  - Use `mfhi`, `mflo` to access result
  - No overflow or divide-by-0 checking
    - Software must perform checks if required

浮点数的表示方法

$9 \times 10^{-28} = 0.9 \times 10^{-27}$

$2 \times 10^{33} = 0.2 \times 10^{34}$

—— $N=10^{E} \cdot M$ （十进制表示）

任意一个十进制数 N 可以写成

$$N = 10^{E} \times M$$

计算机中一个任意进制数 N 可以写成

$$N = R^{e} \times m$$

m ：尾数，是一个纯小数。

e ：浮点的指数，是一个整数。

R ：基数，对于二进计数值的机器是一个常数，一般规定 R 为2，8或16。

一个机器浮点数由阶码和尾数及其符号位组成：

尾数：用定点小数表示，给出有效数字的位数，
决定了浮点数的表示精度；

阶码：用定点整数形式表示，指明小数点在数据中的位置，决定了浮点数的表示范围。

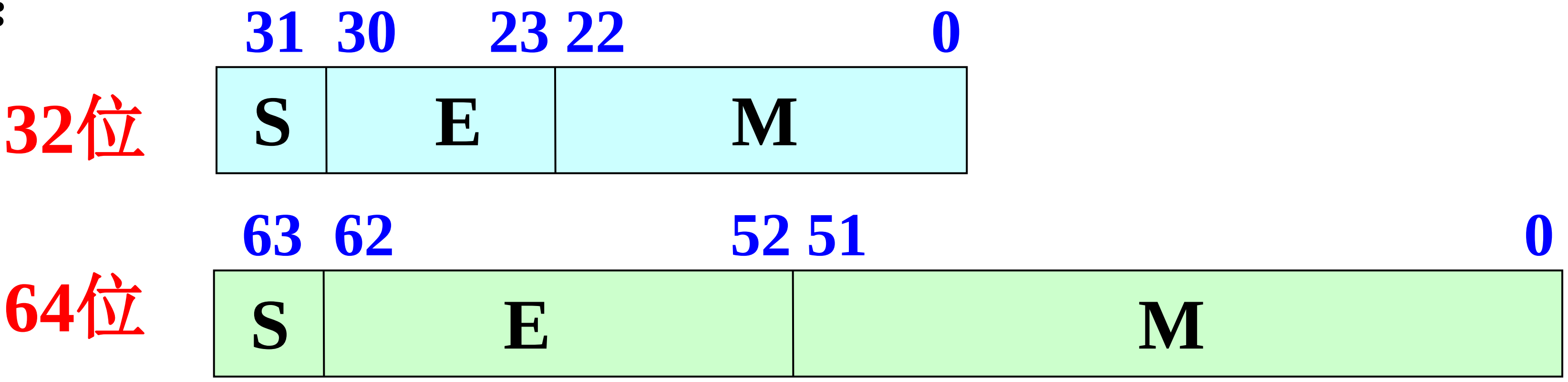
E_s	$E_1\ E_2\ \ldots\ldots\ E_m$	M_s	$M_1\ M_2\ \ldots\ldots\ M_n$
阶符	阶码	数符	尾数

(2) 浮点数的标准格式　　（$N=R^e.m$）

为便于软件移植，使用 IEEE（电气和电子工程师协会）标准

IEEE754 标准：尾数用原码；阶码用移码；基为2

- 按照 IEEE754 的标准，32位浮点数和64位浮点数的标准格式为：



S——尾数符号，0正1负；

M——尾数, 纯小数表示, 小数点放在尾数域的最前面。

采用原码表示。

E——阶码，采用“移码”表示;

阶符采用隐含方式，即采用“移码”方法来表示正负指数。

(3) 浮点数的规格化表示

浮点数是数学中实数的子集合，由一个纯小数乘上一个指数值来组成。

一个浮点数有不同的表示：

0.5；　0.05×10^{1} ；　0.005×10^{2}；　50×10^{-2}

为提高数据的表示精度，需做规格化处理。

浮点数的规格化

规格化目的：

为了提高数据的表示精度

为了数据表示的唯一性

尾数为R进制的规格化：

绝对值大于或等于1/R

二进制原码的规格化数的表现形式：

正数 0.1xxxxxx

负数 1.1xxxxxx

补码尾数的规格化的表现形式：

尾数的最高位与符号位相反

正数 0.1xxxxxx

负数 1.0xxxxxx

规格化处理：

在计算机内，其纯小数部分被称为浮点数的尾数，对非 0 值的浮点数，要求尾数的绝对值必须 >= 1/2，即尾数域的最高有效位应为1，称满足这种表示要求的浮点数为规格化表示：

0.1000101010

把不满足这一表示要求的尾数，变成满足这一要求的尾数的操作过程，叫作浮点数的规格化处理，通过尾数移位和修改阶码实现。

隐藏位技术：

既然非 0 值浮点数的尾数数值最高位必定为 1，则在保存浮点数到内存前，通过尾数左移， 强行把该位去掉，用同样多的尾数位就能多存一位二进制数，有利于提高数据表示精度，称这种处理方案使用了隐藏位技术。

$$0.1100010 \rightarrow 1.100010$$

当然，在取回这样的浮点数到运算器执行运算时，必须先恢复该隐藏位。

(4) 规格化浮点数的真值

32位浮点数格式：

31	30　　23	22　　　　0
S	E	M

移码定义：　$[x]_{移} = x0\ x1\ x2\ \cdots xn$

$= 2^n + x \qquad -2^n \leq x < 2^n$

一个规格化的32位浮点数 *x* 的真值为：

$$x = (-1)^s \times (1.M) \times 2^{E-127} \qquad e = E - 127$$

一个规格化的64位浮点数 *x* 的真值为：

$$x = (-1)^s \times (1.M) \times 2^{E-1023}$$

这里e是真值， E 是机器数

例：若浮点数 x 的二进制存储格式为$(41360000)_{16}$，求其32位浮点数的十进制值。

解：0100,0001,0011,0110,0000,0000,0000,0000

数符：0

阶码：1000,0010

尾数：011,0110,0000,0000,0000,0000

指数e＝阶码－127＝10000010－01111111 ＝00000011=$(3)_{10}$

包括隐藏位1的尾数：

1.M＝1.011 0110 0000 0000 0000 0000＝1.011011

于是有 $x=(-1)^{s}\times 1.M\times 2^{e}$

$=+(1.011011)\times 2^{3}=+1011.011=(11.375)_{10}$

例: 将十进制数20.59375转换成32位浮点数的二进制格式来存储

解：首先分别将整数和分数部分转换成二进制数：

20.59375＝10100.10011

然后移动小数点，使其在第1，2位之间

$10100.10011＝1.010010011 \times 2^4$　　　　$e＝4$

于是得到：　$e = E - 127$

$S＝0$，$E＝4＋127＝131=1000,0011$，$M＝010010011$

最后得到32位浮点数的二进制存储格式为

$0100\ 0001\ 1010\ 0100\ 1100\ 0000\ 0000\ 0000＝(41A4C000)_{16}$

例：将十进制数-0.75表示成单精度的IEEE 754标准代码

解：$-0.75 = -3/4 = -0.11_2 = -1.1\times 2^{-1}$

$=(-1)^1\times(1 + 0.1000\ 0000\ 0000\ 0000\ 0000\ 000)\times 2^{-1}$

$=(-1)^1\times(1 + 0.1000\ 0000\ 0000\ 0000\ 0000\ 000)\times 2^{126-127}$

$s=1$，$E= 126_{10} = 01111110_2$，　F　= 1000 … 000。

1 011,1111,0 100,0000,0000,0000,0000,0000

B　F　4　0　0　0　0　0 H

例： 求如下IEEE 754 单精度浮点数的十进制数值：

1 10000001 01000000000000000000000

解： S=1，*E*=129，F = 1/4 = 0.25，

$(-1)^1 \times (1+0.25) \times 2^{129-127}$

$= -1 \times 1.25 \times 2^2$

$= -1.25 \times 4$

$= -5.0$

例： 对数据123_{10}作规格化浮点数的编码，假定1位符号位，基数为2，阶码5位，采用移码，尾数10位，采用补码。

解： $123_{10}=1111011_2= 0.1111011000_2 \times 2^7$

$[7]_{移}=10000+00111 = 10111$

$[0.1111011000]_{补}=0.1111011000$

$[123]_{浮}=0\ 10111\ 1111011000$

IEEE754浮点数的范围

格式	最小值	最大值
单精度	E=1, M=0, $1.0\times 2^{1-127} = 2^{-126}$	E=254, f=.1111 …, $1.111\ldots1\times 2^{254-127}$ $= 2^{127}\times(2-2^{-23})$
双精度	E=1, M=0, $1.0\times 2^{1-1023} =2^{-1022}$	E=2046, f=.1111 …, $1.111\ldots1\times 2^{2046-1023}$ $=2^{1023}\times(2-2^{-52})$

浮点数标准(IEEE754)

IEEE　　单精度浮点数编码格式

符号位	阶码	尾数	表示
0/1	255	非 0	NaN
0/1	255	非 0	NaN
0	255	0	+INF
1	255	0	-INF
0/1	1-254	f	$(-1)^s \times 1.f \times 2^{e-127}$
0/1	0	f(非 0)	$(-1)^s \times 0.f \times 2^{e-126}$
0/1	0	0	+0/-0

浮点运算方法和浮点运算器

浮点加、减法运算

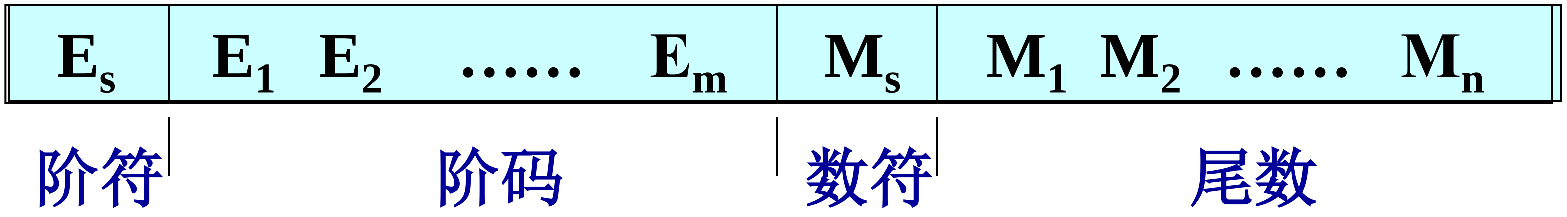
浮点乘、除法运算

浮点数的表示方法

机器浮点数格式：

尾数：用定点小数表示，给出有效数字的位数，
决定了浮点数的表示精度；

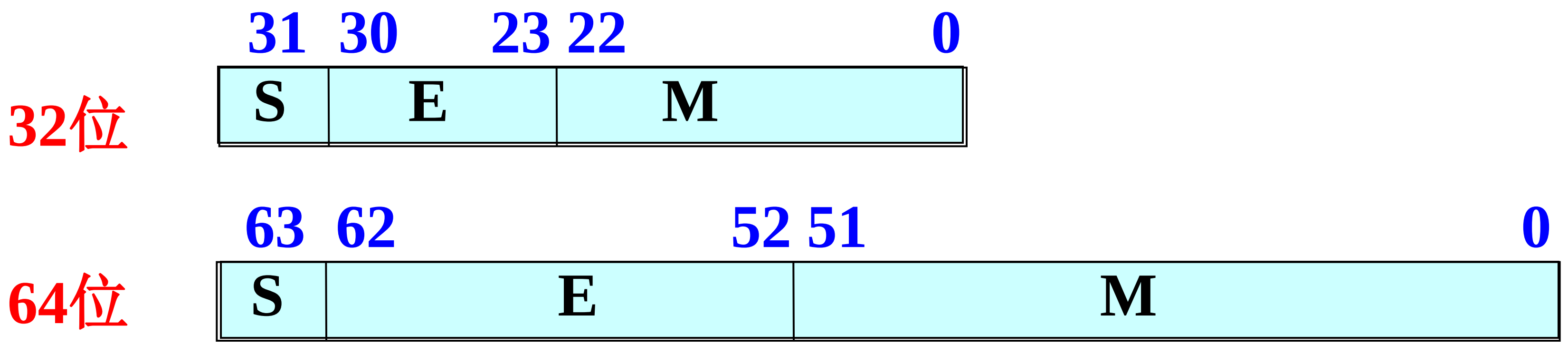
阶码：用整数形式表示，指明小数点在数据中的位
置，决定了浮点数的表示范围。



浮点数的标准格式

- 为便于软件移植，使用 IEEE标准

按照 IEEE754 的标准，32位浮点数和64位浮点数的标准格式为：



IEEE 标准：尾数用原码；阶码用“移码”；基为2。

浮点加、减法运算

设有两个浮点数 x 和 y，它们分别为:

$$x = 2^{Ex} \cdot M_x$$

$$y = 2^{Ey} \cdot M_y$$

其中 E_x 和 E_y 分别为数 x 和 y 的阶码，

M_x 和 M_y 为数 x 和 y 的尾数。

两浮点数进行加法和减法的运算规则是:

$$x \pm y = (M_x 2^{Ex-Ey} \pm M_y) 2^{Ey} \qquad E_x <= E_y$$

完成浮点加减运算的操作过程大体分为四步：

(1) 0 操作数的检查；

(2) 比较阶码大小并完成对阶；

(3) 尾数进行加或减运算；

(4) 结果规格化。

(5) 舍入处理。

(6) 溢出处理。

(1) 0 操作数检查

(2) 对阶

使二数阶码相同（即小数点位置对齐），这个过程叫作对阶。

- **先求两数阶码 E_x 和 E_y之差，即$\triangle E = E_x - E_y$**

若$\triangle E = 0$， 表示 **$E_x=E_y$**

若$\triangle E > 0$, **$E_x>E_y$**

若$\triangle E < 0$, **$E_x<E_y$**

通过尾数的移动来改变E_x或E_y，使其相等.

- 对阶原则

阶码小的数向阶码大的数对齐；

小阶的尾数右移，每右移一位, 其阶码加1 (右规)。

例：$x=2^{01}\times 0.1101$，$y=2^{11}\times(-0.1010)$， 求$x+y=?$

解：为便于直观了解，两数均以**补码**表示，阶码、尾数均采用双符号位。

$[x]_{补}=00\ 01$， 00.1101　　　$[y]_{补}=00\ 11$， 11.0110

$[\triangle E]_{补}=[E_x]_{补}-[E_y]_{补}=00\ 01+11\ 01=11\ 10$

$\triangle E=-2$， 表示E_x比E_y小2，　因此将x的尾数右移两位.

右移一位，　得　$[x]_{补}=00\ 10$， 00.0110

再右移一位，　得　$[x]_{补}=00\ 11$， 00.0011

至此, $\triangle E=0$,　　对阶完毕.

(3) 尾数求和运算

尾数求和方法与定点加减法运算完全一样。

对阶完毕可得：$[x]_{补}$=00 11, 00.0011

$[y]_{补}$=00 11, 11.0110

对尾数求和：

$$\begin{array}{r} 00.0011 \\ +\ 11.0110 \\ \hline 11.1001 \end{array}$$

即得：$[x+y]_{补}$=00 11, 11.1001

(4) 结果规格化

求和之后得到的数可能不是规格化了的数，为了增加有效数字的位数，提高运算精度, 必须将求和的结果规格化.

①规格化的定义： $\frac{1}{2} \le |S| < 1$ （二进制）

采用原码：

正数： S=0.1 ×××…×

负数： S=1.1 ×××…×

采用双符号位的补码：

对正数： S=00. 1×××…×

对负数： S=11. 0×××…×

②向左规格化

若不是规格化的数,需要尾数向左移位,以实现规格化的过程,我们称其为向左规格化。

前例中, 00 11, 11.1001不是规格化数,因而需要左规,即左移一位,阶码减1,得:

$[x+y]_{补}$=00 10, 11.0010

③向右规格化

浮点加减运算时,尾数求和的结果也可能得到:

01.×××…× 或 10.×××…×,

即两符号位不等，即结果的绝对值大于1。向左破坏了规格化。

此时,将尾数运算的结果右移一位,阶码加1,称为向右规格化。

例：两浮点数 $x=0.1101 \times 2^{10}$, $y=(0.1011) \times 2^{01}$，求x+y。

解：　$[x]_{补}$=00 10，00.1101　　　$[y]_{补}$=00 01，00.1011

对阶：　$[\triangle E]_{补}=[E_x]_{补}-[E_y]_{补}$=00 10+ 11 11= 00 01

y向x对齐，将y的尾数右移一位，阶码加1。

$[y]_{补}$=00 10，00.0101

求和：

$$\begin{array}{r} 00.1101 \\ +\ 00.0101 \\ \hline 01.0010 \end{array}$$

$[x+y]_{补}$=00 10，01.0010

右归：运算结果两符号位不同，其绝对值大于1，右归。

$[x+y]_{补}$= 00 11，00.1001

(5) 舍入处理

在对阶或向右规格化时，尾数要向右移位，这样，被右移的尾数的低位部分会被丢掉，从而造成一定误差，因此要进行舍入处理。

- **简单的舍入方法有两种：**

① “0舍1入”法

即如果右移时被丢掉数位的最高位为0则舍去，反之则将尾数的末位加“1”。

② “恒置1”法

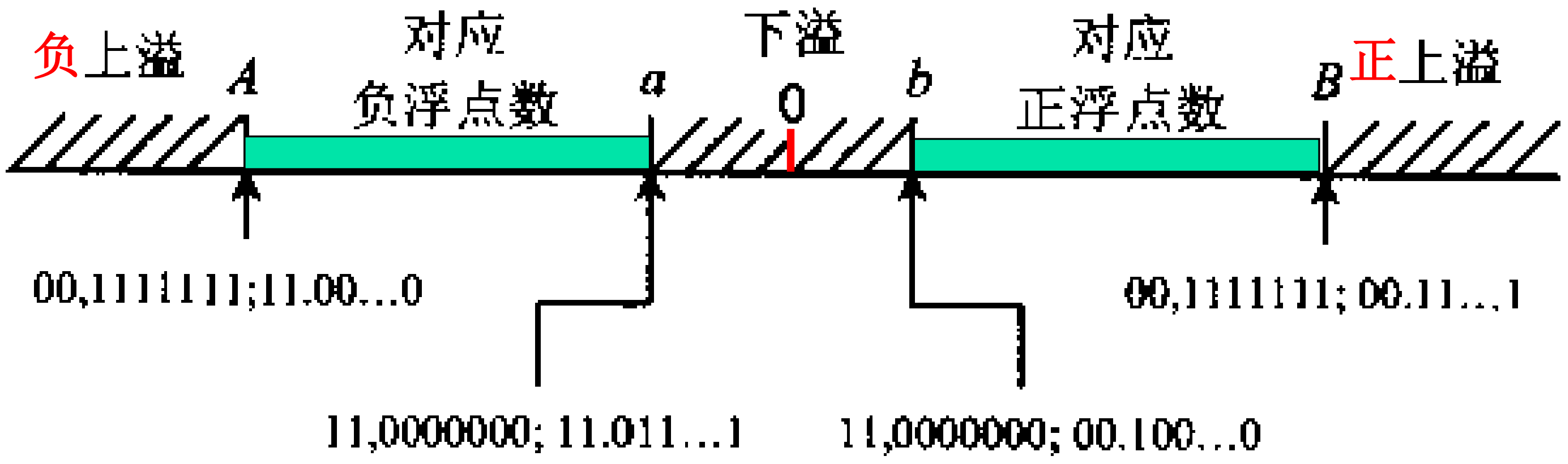
即只要数位被移掉，就在尾数的末位恒置“1”。从概率上来说，丢掉的0和1各为1/2。

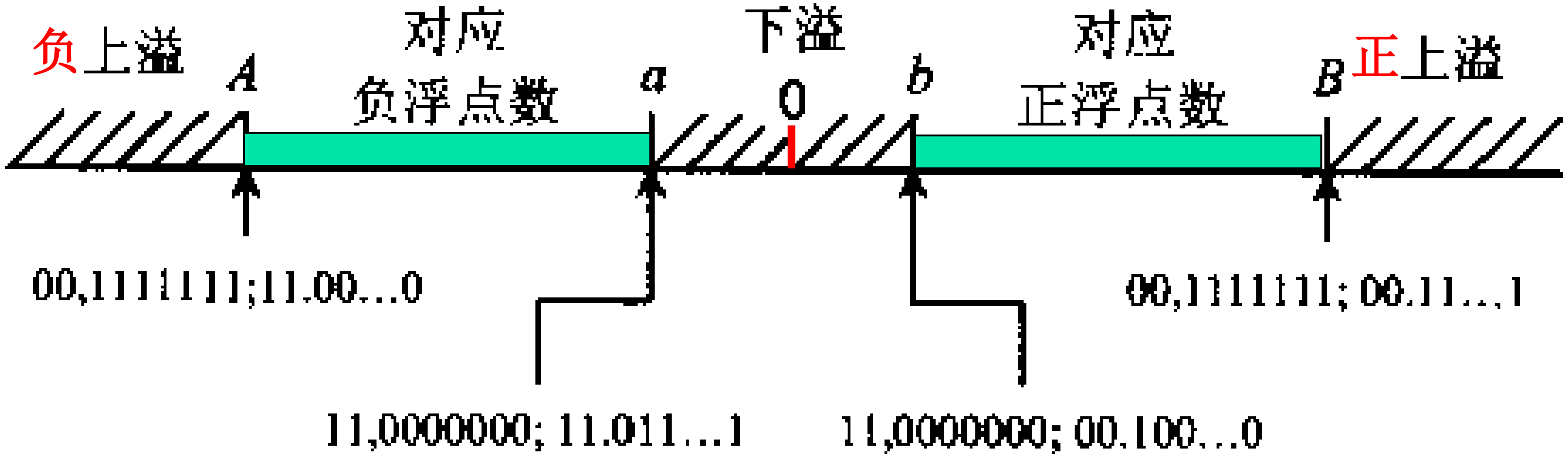
- IEEE754标准中，舍入处理提供了四种可选方法：

(6)溢出处理

与定点加减法一样，浮点加减运算最后一步也需判溢出。在浮点规格化中已指出，当尾数之和(差)出现01．××…×或10．××…×时，并不表示溢出，只有将此数右规后，再根据阶码来判断浮点运算结果是否溢出。

若机器数为补码，尾数为规格化形式，并假设阶符取2位，阶码取7位、数符取2位，尾数取n位，则它们能表示的补码在数轴上的表示范围如图所示。





图中A，B，a，b分别对应最小负数、最大正数、最大负数和最小正数。它们所对应的真值分别是：

A最小负数　$2^{+127}\times(-1)$

B最大正数　$2^{+127}\times(1-2^{-n})$

a最大负数　$2^{-128}\times(-2^{-1}-2^{-n})$

b最小正数　$2^{-128}\times 2^{-1}$

图中a,b之间的阴影部分，对应阶码小于128的情况，叫做浮点数的下溢。下溢时．浮点数值趋于零，故机器不做溢出处理，仅把它作为机器零。

图中的A、B两侧阴影部分，对应阶码大于127的情况，叫做浮点数的上溢。此刻，浮点数真正溢出，机器需停止运算，作溢出中断处理。一般说浮点溢出，均是指上溢。

可见，浮点机的溢出与否可由阶码的符号决定：

阶码$[j]_{补}$=01, ×××××为上溢，机器停止运算，做中断处理；

阶码$[j]_{补}$=10, ×××××为下溢，按机器零处理。

例：若某次加法操作的结果为 $[X+Y]_{补}$=00.111, 10.1011100111

则应对其进行向右规格化操作：

尾数为： 11.0101110011， 阶码加1： 01.000

阶码超出了它所能表示的最大正数（+7），表明本次浮点运算产生了溢出。

例：若某次加法操作的结果为 $[X+Y]_{补}$=11.010, 00.0000110111

则应对其进行向左规格化操作：

尾数为： 00.1101110000， 阶码减4：

$$\begin{array}{r l} 11.010 & \\ +\ 11.100 & [-4]_{补} \\ \hline 10.110 & \end{array}$$

阶码超出了它所能表示的最小负数（-8），表明本次浮点运算产生了溢出。

小结：

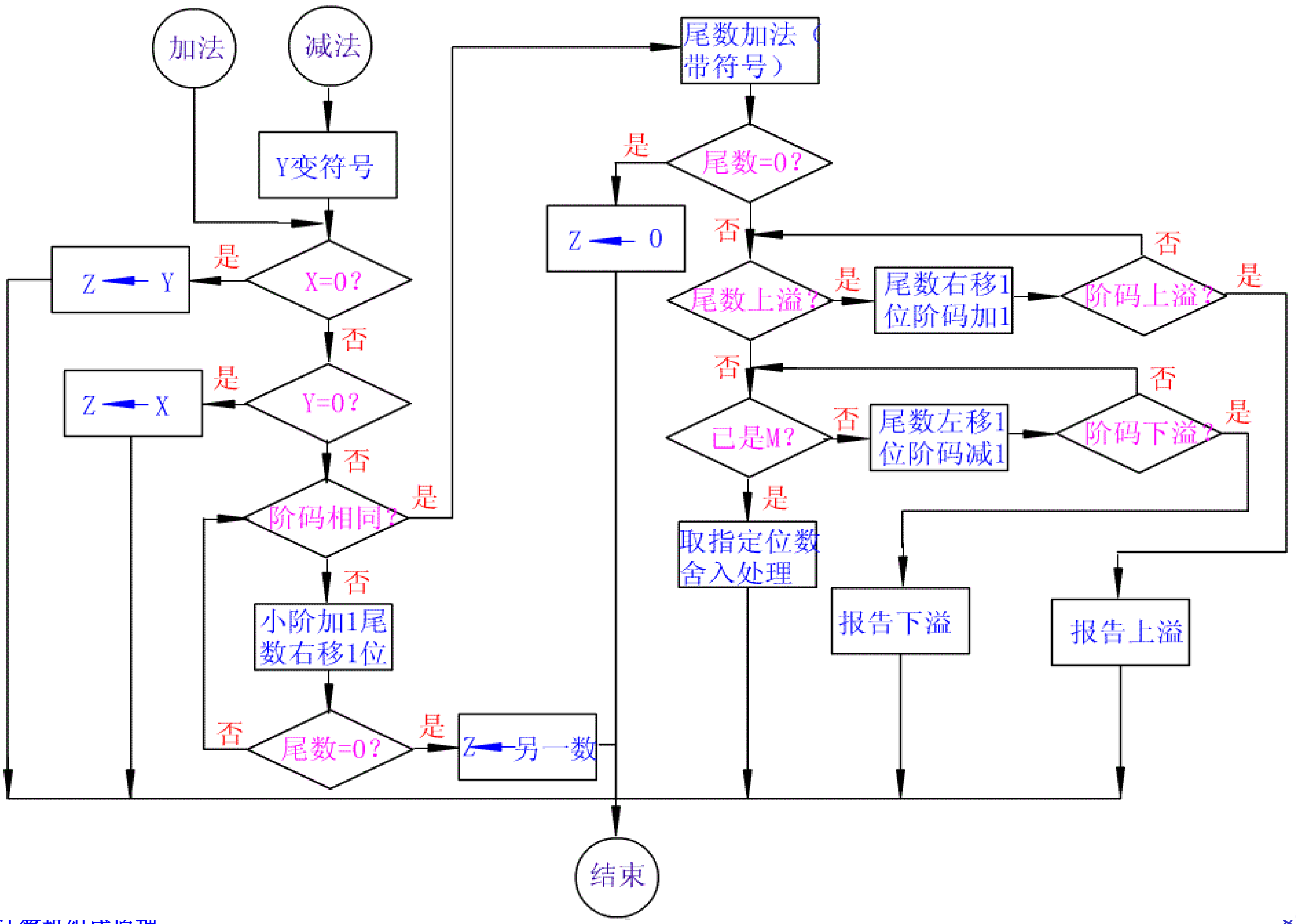
- 浮点数的溢出是以其阶码溢出表现出来的

在加、减运算过程中要检查是否产生了溢出：若阶码正常，加减运算正常结束；若阶码溢出，则要进行相应的处理。

阶码上溢—— 超过了阶码可能表示的最大值的正指数值,一般将其认为是＋∞和－∞。

阶码下溢—— 超过了阶码可能表示的最小值的负指数值,一般将其认为是0。

- 对尾数的溢出也需要处理(上溢—右归，下溢—舍入）。


加法
减法
Y变符号
X=0?
是
Z ← Y
否
Y=0?
是
Z ← X
否
阶码相同?
是
否
小阶加1尾数右移1位
尾数=0?
是
否
Z ← 另一数
尾数加法（带符号）
尾数=0?
是
Z ← 0
否
尾数上溢?
是
尾数右移1位阶码加1
阶码上溢?
是
否
否
已是M?
否
尾数左移1位阶码减1
阶码下溢?
是
否
是
取指定位数舍入处理
报告下溢
报告上溢
结束

例 设 $x = 2^{010} \times 0.11011011$，$y = 2^{100} \times (-0.10101100)$，求 $x + y$ 。

解： 阶码采用双符号位, 尾数采用单符号位，则它们的浮点表示分别为

$[x]_{浮} = 00\ 010,\quad 0.11011011$

$[y]_{浮} = 00\ 100,\quad 1.01010100$

(1) 求阶差并对阶

$\triangle E = E_x - E_y = [E_x]_{补} + [-E_y]_{补} = 00\ 010 + 11\ 100 = 11\ 110$

即△E为−2，x的阶码小，应使 M_x右移两位，E_x加2,

$[x]_{浮} = 00\ 100,\ 0.00110110(11)$

其中(11)表示M_x右移2位后移出的最低两位数。

(2)尾数求和

```
    0.00110110(11)
 +  1.01010100
 ------------------
    1.10001010(11)
```

(3) 规格化处理

尾数运算结果的符号位与最高数值位为同值，应执行左规处理，结果为1.00010101(10)， 阶码为00 011。

(4) 舍入处理

采用0舍1入法处理，则有:

```
    1.00010101
 +           1
 --------------
    1.00010110
```

(5) 判断溢出

阶码符号位为00，不溢出，故得最终结果为

$$x + y = 2^{011} \times (-0.11101010)$$

例 两浮点数$x = 2^{01} \times 0.1101$，$y = 2^{11} \times (-0.1010)$。假设尾数在计算机中以补码表示，可存储4位尾数，2位保护位，阶码以原码表示，求$x+y$。

解：将x, y转换成浮点数据格式

$[x]_{浮} = 00\ 01,\ 00.1101$

$[y]_{浮} = 00\ 11,\ 11.0110$

步骤1：对阶，阶差为11−01=10，即2，因此将x的尾数右移两位，得

$[x]_{浮} = 00\ 11,\ 00.001101$

步骤2：对尾数求和，得：

$[x+y]_{浮} = 00\ 11,\ 11.100101$

步骤3：由于符号位和第一位数相等，不是规格化数，向左规格化，得

$[x+y]_{浮} = 00\ 10,\ 11.001010$

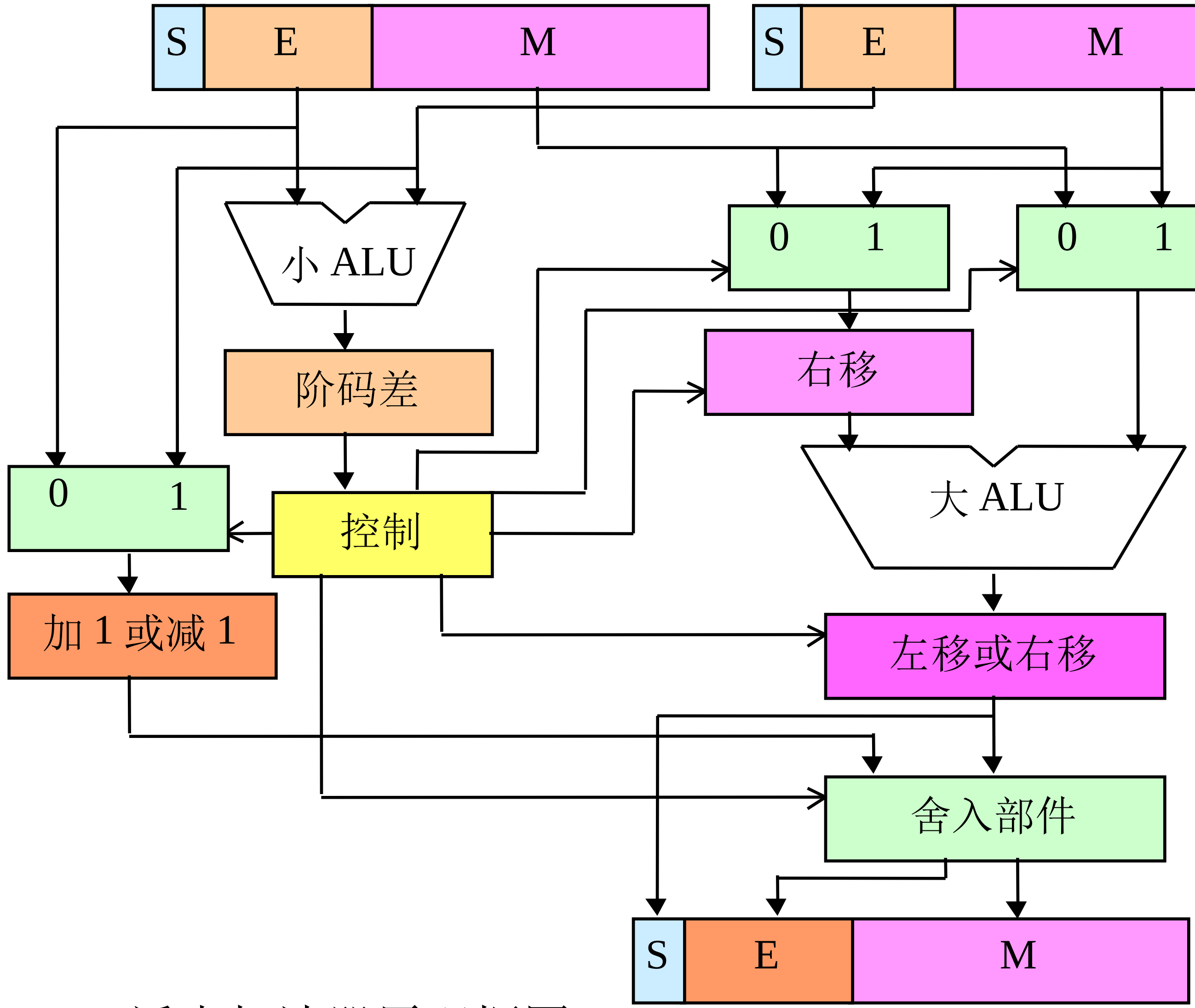
步骤4：截去。

$[x+y]_{浮} = 00\ 10,\ 11.0010$

步骤5：数据无溢出，因此结果为

$x+y = 2^{10} \times (-0.1110)$

浮点运算电路



浮点加法器原理框图

浮点乘、除法运算

1. 浮点乘法、除法运算规则

设有两个浮点数 x 和 y： $x = 2^{Ex} \cdot M_x$

$y = 2^{Ey} \cdot M_y$

浮点乘法运算的规则是: $x \times y = 2^{(Ex + Ey)} \cdot (M_x \times M_y)$

即： 乘积的尾数是相乘两数的尾数之积;
乘积的阶码是相乘两数的阶码之和。

浮点除法运算的规则是: $x \div y = 2^{(Ex - Ey)} \cdot (M_x \div M_y)$

即：商的尾数是相除两数的尾数之商;
商的阶码是相除两数的阶码之差。

2. 浮点乘、除法运算步骤

浮点数的乘除运算大体分为四步：

(1) 0 操作数检查；

(2) 阶码加/减操作；

(3) 尾数乘/除操作；

(4) 结果规格化及舍入处理。

(2) 浮点数的阶码运算

- 对阶码的运算有＋1、－1、两阶码求和、两阶码求差四种，运算时还必须检查结果是否溢出。
- 在计算机中，阶码通常用补码或移码形式表示。

①移码的运算规则和判定溢出的方法

移码的定义为 $[x]_{移} = 2^n + x \qquad -2^n \le x < 2^n$

按此定义，则有 $[x]_{移} + [y]_{移} = 2^n + x + 2^n + y$

$$= 2^n + (2^n + (x + y))$$

$$= 2^n + [x + y]_{移}$$

$$[x + y]_{移} = -2^n + [x]_{移} + [y]_{移}$$

②混合使用移码和补码

考虑到移码和补码的关系：

对同一个数值，其数值位完全相同，而符号位正好完全相反。

$[y]_{补}$的定义为 $[y]_{补} = 2^{n+1} + y$

则求阶码和用如下方式完成：

$$[x]_{移}+[y]_{补} = 2^{n} + x + 2^{n+1} + y$$

$$= 2^{n+1} + (2^{n} +(x + y))$$

即： $[x + y]_{移} = [x]_{移}+[y]_{补} \quad (\mathrm{mod}\ 2^{n+1})$

同理： $[x - y]_{移} = [x]_{移}+[-y]_{补} \quad (\mathrm{mod}\ 2^{n+1})$

③ 阶码运算结果溢出处理

使用双符号位的阶码加法器，并规定移码的第二个符号位，即最高符号位恒用 0 参加加减运算，则溢出条件是结果的最高符号位为1：

- 当低位符号位为 0时，（10） 表明结果上溢，
- 当低位符号位为1时，（11） 表明结果下溢。
- 当最高符号位为0时，表明没有溢出：

  低位符号位为1，（01） 表明结果为正；

  为0，（00） 表明结果为负。

例： $x = +011, y = +110$，求$[x+y]_{移}$ 和 $[x-y]_{移}$，并判断是否溢出。

解：阶码取3位（不含符号位），其对应的真值范围是 -8~+7

$[x]_{移}= 01\ 011$， $[y]_{补}= 00\ 110$， $[-y]_{补}=11\ 010$

$[x+y]_{移}= [x]_{移}+ [y]_{补}=$

```
   01 011
 + 00 110
 --------
   10 001
```

结果上溢。

$[x-y]_{移}= [x]_{移}+ [-y]_{补}=$

```
   01 011
 + 11 010
 --------
   00 101
```

结果正确，为−3。

(3) 尾数处理

浮点加减法对结果的规格化及舍入处理也适用于浮点乘除法。

第一种方法是：

无条件地丢掉正常尾数最低位之后的全部数值。

这种办法被称为截断处理，好处是处理简单，缺点是影响结果的精度。

第二种办法是：

运算过程中保留右移中移出的若干高位的值，最后再按某种规则用这些位上的值修正尾数。

这种处理方法被称为舍入处理。

舍入处理

- 当尾数用原码表示时:

　　最简便的方法是，只要尾数的最低位为1，或移出的几位中有为1的数值位，就使最低位的值为1。

　　另一种是0舍1入法，即当丢失的最高位的值为1时，把这个1加到最低数值位上进行修正，否则舍去丢失的的各位的值。这样处理时,舍入效果对正数负数相同，入将使数的绝对值变大，舍则使数的绝对值变小。

- 当尾数是用补码表示时：

采用0舍1入法时，若丢失的位不全为0时：

对正数来说，舍入的结果与原码分析相同；

对负数来说，舍入的结果与原码分析相反，即“舍”使绝对值变大，“入”使绝对值变小；为使原、补码舍入处理后的结果相同，对负数可采用如下规则进行舍入处理：

当丢失的各位均为0时，不必舍入；

当丢失的最高位为0，以下各位不全为0 时，或者丢失的最高位为1，以下各位均为0时，则舍去丢失位上的值；

当丢失的最高位为1,以下各位不全为0 时,则执行在尾数最低位入1的修正操作。

例 设 $[x_1]_{补}$= 11.01100000, $[x_2]_{补}$= 11.01100001, $[x_3]_{补}$= 11.01101000, $[x_4]_{补}$= 11.01111001，求执行只保留小数点后4位有效数字的舍入操作值。

解：执行舍入操作后,其结果值分别为

$[x_1]_{补}$＝11.0110　(不舍不入)

$[x_2]_{补}$＝11.0110　(舍)

$[x_3]_{补}$＝11.0110　(舍)

$[x_4]_{补}$＝11.1000　(入)

例 设有浮点数 $x = 2^{-5} \times 0.0110011$，$y = 2^{3} \times (-0.1110010)$，阶码用4位移码表示，尾数 (含符号位)用8位补码表示。求$[x \times y]_{浮}$。要求用补码完成尾数乘法运算，运算结果尾数保留高8位(含符号位)，并用尾数低位字长值处理舍入操作。

解:　　移码采用双符号位，尾数补码采用单符号位，则有

$[M_x]_{补} = 0.0110011$，　$[M_y]_{补} = 1.0001110$，

$E_x]_{移} = 00\ 011$，　$[E_y]_{移} = 01\ 011$，　$[E_y]_{补} = 00\ 011$，

$[x]_{浮} = 00\ 011,\ 0.0110011$，　$[y]_{浮} = 01\ 011,\ 1.0001110$

(1) 求阶码和

$[E_x+E_y]_{移}=[E_x]_{移}+[E_y]_{补}=00\ 011+00\ 011=00\ 110$,

值为移码形式−2。

(2) 尾数乘法运算

可采用补码阵列乘法器实现，即有

$$[M_x]_{补}\times[M_y]_{补}=[0.0110011]_{补}\times[1.0001110]_{补}$$

$$=[1.1010010,1001010]_{补}$$

(3) 规格化处理

乘积的尾数符号位与最高数值位符号相同，不是规格化的数，需要左规，阶码变为00 101(−3)，尾数变为 1.0100101,0010100。

(4) 舍入处理

尾数为负数，取尾数高位字长，按舍入规则，舍去低位字长，故尾数为1.0100101 。

最终相乘结果为 $[x \times y]_{浮} = 00\ 101, 1.0100101$

其真值为 $x \times y = 2^{-3} \times (-0.1011011)$

浮点数表示与算法小结：

1） IEEE 754 浮点数的表示

2） 尾数的规格化表示

3） 浮点数计算流程

4） 浮点数计算的硬件实现

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*`xx`*`.s`, `c.`*`xx`*`.d` (*xx* is eq, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - `fahr` in $f12, result in $f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

FP Example: Array Multiplication

- X = X + Y × Z
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double x[][],
         double y[][], double z[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        x[i][j] = x[i][j]
                  + y[i][k] * z[k][j];
}
```

  - Addresses of x, y, z in $a0, $a1, $a2, and `i`, `j`, k in $s0, $s1, $s2

FP Example: Array Multiplication

- MIPS code:

```
    li   $t1, 32       # $t1 = 32 (row size/loop end)
    li   $s0, 0        # i = 0; initialize 1st for loop
L1: li   $s1, 0        # j = 0; restart 2nd for loop
L2: li   $s2, 0        # k = 0; restart 3rd for loop
    sll  $t2, $s0, 5   # $t2 = i * 32 (size of row of x)
    addu $t2, $t2, $s1 # $t2 = i * size(row) + j
    sll  $t2, $t2, 3   # $t2 = byte offset of [i][j]
    addu $t2, $a0, $t2 # $t2 = byte address of x[i][j]
    l.d  $f4, 0($t2)   # $f4 = 8 bytes of x[i][j]
L3: sll  $t0, $s2, 5   # $t0 = k * 32 (size of row of z)
    addu $t0, $t0, $s1 # $t0 = k * size(row) + j
    sll  $t0, $t0, 3   # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
    l.d  $f16, 0($t0)  # $f16 = 8 bytes of z[k][j]
```

…

FP Example: Array Multiplication

```
    …
    sll   $t0, $s0, 5         # $t0 = i*32 (size of row of y)
    addu   $t0, $t0, $s2      # $t0 = i*size(row) + k
    sll    $t0, $t0, 3        # $t0 = byte offset of [i][k]
    addu   $t0, $a1, $t0      # $t0 = byte address of y[i][k]
    l.d    $f18, 0($t0)       # $f18 = 8 bytes of y[i][k]
    mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
    add.d $f4, $f4, $f16      # f4=x[i][j] + y[i][k]*z[k][j]
    addiu $s2, $s2, 1         # $k = k + 1
    bne    $s2, $t1, L3       # if (k != 32) go to L3
    s.d    $f4, 0($t2)        # x[i][j] = $f4
    addiu $s1, $s1, 1         # $j = j + 1
    bne    $s1, $t1, L2       # if (j != 32) go to L2
    addiu $s0, $s0, 1         # $i = i + 1
    bne    $s0, $t1, L1       # if (i != 32) go to L1
```

Accurate Arithmetic

- Infinite variety of real numbers between, say, 0 and 1
  - Only 2^{53} can be represented by double precision FP
- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - guard and round bits are 2 extra bits kept on the right during intermediate additions
  - sticky bit used in rounding in addition to guard and round bits; is set whenever a 1 bit shifts right of the round bit

F = 1 . xxxxxxxxxxxxxxxxxxxxxx G R S

- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Interpretation of Data

The BIG Picture

- Bits have no inherent meaning
  - Same bits can represent a variety of objects
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Programmers, computer systems must minimize gap between computer arithmetic and real world arithmetic

Parallelism and Associativity

- Parallel programs may interleave operations in unexpected orders
  - Integer addition is associative
  - Assumptions of associativity for FP numbers may fail!

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Floating point numbers are approximations of real numbers – not associative!
- Need to validate parallel programs under varying degrees of parallelism

x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64-bit in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental

`FILD mem/ST(i)`
`FISTP mem/ST(i)`
`FLDPI`
`FLD1`
`FLDZ`
	`FIADDP mem/ST(i)`
	`FISUBRP mem/ST(i)`
	`FIMULP mem/ST(i)`
	`FIDIVRP mem/ST(i)`
	`FSQRT`
	`FABS`
	`FRNDINT`
		`FICOMP`
		`FIUCOMP`
		`FSTSW AX/mem`
			`FPATAN`
			`F2XMI`
			`FCOS`
			`FPTAN`
			`FPREM`
			`FPSIN`
			`FYL2X`

- No FP branch – FSTSW sends result of CMP to INT CPU
- Optional variations
  - `I`: integer operand
  - `P`: pop operand from stack
  - `R`: reverse operand order
  - But not all combinations allowed

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit single precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

Fallacy: Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers!
- For signed integers
  - Logical right shift is clearly erroneous
    - e.g., –5 / 4
    - 11111011_2 >>> 2 = 00111110_2 = +62
  - Arithmetic right shift - replicate the sign bit
    - 11111011_2 >> 2 = 11111110_2 = –2
    - Result is -2 instead of -1; close, but no cigar

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug (~1994)
  - Bug in LUT used to guess multiple quotient bits per step; wrong values in some LUT locations
  - Cost Intel $300+ million
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

A Closer Look

Warning: Intel Inside

BUSINESS

MARKETS ◆ HIGH TECH ◆ ECONOMY

Intel to work on software patch for Pentium bug

Quick fix: Program will work around the

Grove says I sorry about Pentium bu

INFO WORLD

The Voice of Personal Computing in the Enterprise

Photo finish: Although Photoshop outdoes Picture P has an in

Bug Dodge Booed

Intel Knocked For Response On Flaw

Flawed

Flawed Chip Bruises Intel

Investors react, stock plunges

INTEL'S MISTAKE

PERSONAL TECHNOLOGY

By Walter S. Mossberg

Chip Shot

Computer Giants' War Over Flaw in Pentium Jolts the PC Industry

Some Scientists Are Angry Over Flaw In Pentium Chip, and Intel's Response

San Francisco Chronicle

BUSINESS

Intel's Pentium Problem Persists

TECHNOLOGY

Intel's Grove Airs Apology for Pentium Over the Internet

Pentium woes continue

Faulty FPU flubs math in certain equations

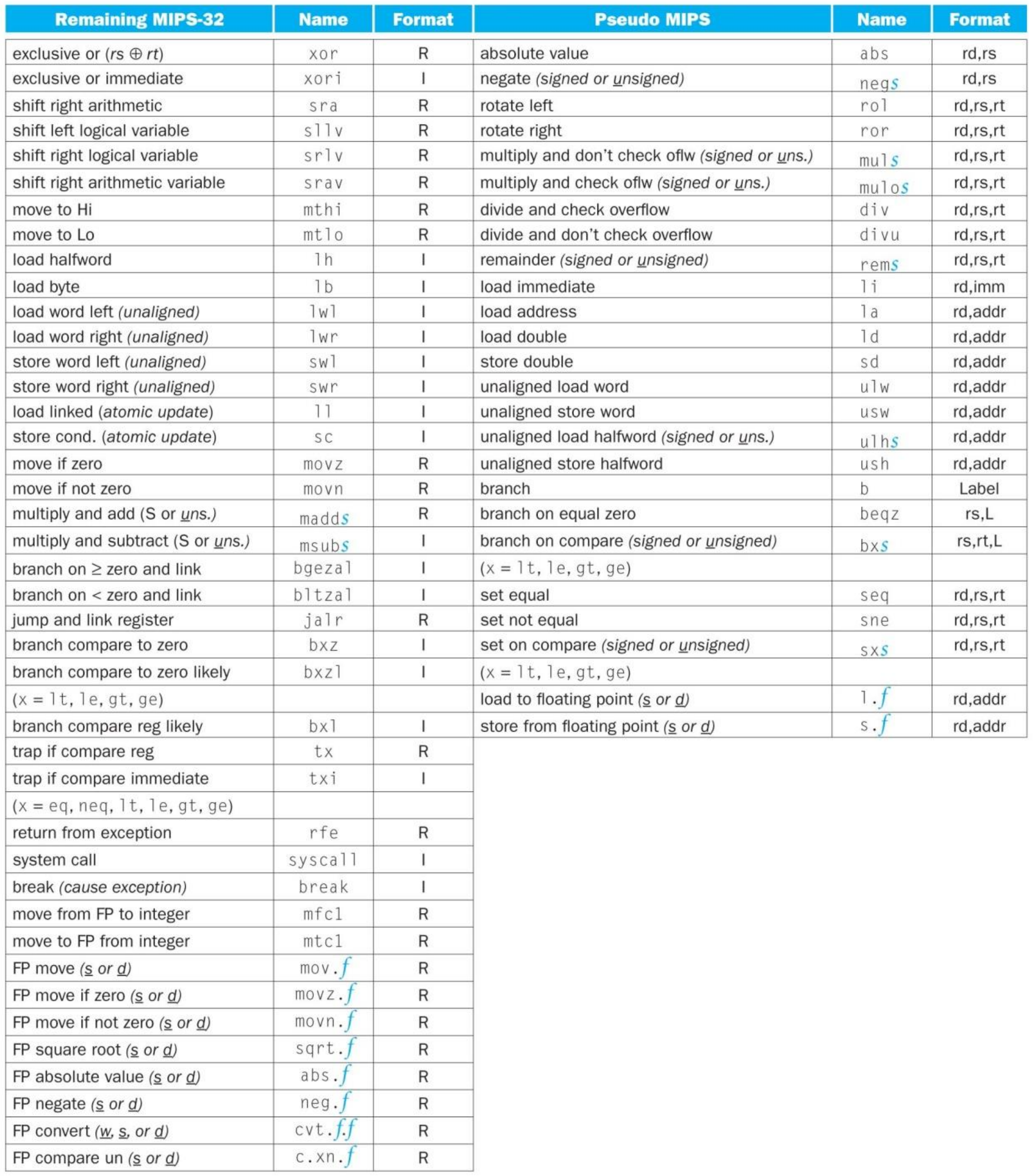
Multithreading gets lost on P100 systems

IBM to stop shipping Pentium PCs

Summary: MIPS Instruction Set

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
AND	AND	R	move from system control (EPC)	mfc0	R
AND immediate	ANDi	I	floating-point add single	add.s	R
OR	OR	R	floating-point add double	add.d	R
OR immediate	ORi	I	floating-point subtract single	sub.s	R
NOR	NOR	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
load linked (*atomic update*)	ll	I	branch on floating-point false	bc1f	I
store cond. (*atomic update*)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

Summary: MIPS Instruction Set



Remaining MIPS-32	Name	Format
exclusive or (*rs* ⊕ *rt*)	xor	R
exclusive or immediate	xori	I
shift right arithmetic	sra	R
shift left logical variable	sllv	R
shift right logical variable	srlv	R
shift right arithmetic variable	srav	R
move to Hi	mthi	R
move to Lo	mtlo	R
load halfword	lh	I
load byte	lb	I
load word left (*unaligned*)	lwl	I
load word right (*unaligned*)	lwr	I
store word left (*unaligned*)	swl	I
store word right (*unaligned*)	swr	I
load linked (*atomic update*)	ll	I
store cond. (*atomic update*)	sc	I
move if zero	movz	R
move if not zero	movn	R
multiply and add (S or *uns.*)	madd*s*	R
multiply and subtract (S or *uns.*)	msub*s*	I
branch on ≥ zero and link	bgezal	I
branch on < zero and link	bltzal	I
jump and link register	jalr	R
branch compare to zero	bxz	I
branch compare to zero likely	bxzl	I
(x = lt, le, gt, ge)		
branch compare reg likely	bxl	I
trap if compare reg	tx	R
trap if compare immediate	txi	I
(x = eq, neq, lt, le, gt, ge)		
return from exception	rfe	R
system call	syscall	I
break (*cause exception*)	break	I
move from FP to integer	mfc1	R
move to FP from integer	mtc1	R
FP move (*s or d*)	mov.*f*	R
FP move if zero (*s or d*)	movz.*f*	R
FP move if not zero (*s or d*)	movn.*f*	R
FP square root (*s or d*)	sqrt.*f*	R
FP absolute value (*s or d*)	abs.*f*	R
FP negate (*s or d*)	neg.*f*	R
FP convert (*w, s, or d*)	cvt.*f*.*f*	R
FP compare un (*s or d*)	c.xn.*f*	R

Pseudo MIPS	Name	Format
absolute value	abs	rd,rs
negate (*signed or unsigned*)	neg*s*	rd,rs
rotate left	rol	rd,rs,rt
rotate right	ror	rd,rs,rt
multiply and don't check oflw (*signed or uns.*)	mul*s*	rd,rs,rt
multiply and check oflw (*signed or uns.*)	mulo*s*	rd,rs,rt
divide and check overflow	div	rd,rs,rt
divide and don't check overflow	divu	rd,rs,rt
remainder (*signed or unsigned*)	rem*s*	rd,rs,rt
load immediate	li	rd,imm
load address	la	rd,addr
load double	ld	rd,addr
store double	sd	rd,addr
unaligned load word	ulw	rd,addr
unaligned store word	usw	rd,addr
unaligned load halfword (*signed or uns.*)	ulh*s*	rd,addr
unaligned store halfword	ush	rd,addr
branch	b	Label
branch on equal zero	beqz	rs,L
branch on compare (*signed or unsigned*)	bx*s*	rs,rt,L
(x = lt, le, gt, ge)		
set equal	seq	rd,rs,rt
set not equal	sne	rd,rs,rt
set on compare (*signed or unsigned*)	sx*s*	rd,rs,rt
(x = lt, le, gt, ge)		
load to floating point (*s or d*)	l.*f*	rd,addr
store from floating point (*s or d*)	s.*f*	rd,addr

Frequency of Common MIPS Instructions

- Only included those with >3% (table 1) and >1% (table 2)

MIPS core	SPECint	SPECfp
`addu`	5.2%	3.5%
`addiu`	9.0%	7.2%
`or`	4.0%	1.2%
`sll`	4.4%	1.9%
`lui`	3.3%	0.5%
`lw`	18.6%	5.8%
`sw`	7.6%	2.0%
`lbu`	3.7%	0.1%
`beq`	8.6%	2.2%
`bne`	8.4%	1.4%
`slt`	9.9%	2.3%
`slti`	3.1%	0.3%
`sltu`	3.4%	0.8%

Arith core + MIPS-32	SPECint	SPECfp
`add.d`	0.0%	10.6%
`sub.d`	0.0%	4.9%
`mul.d`	0.0%	15.0%
`add.s`	0.0%	1.5%
`sub.s`	0.0%	1.8%
`mul.s`	0.0%	2.4%
`l.d`	0.0%	17.5%
`s.d`	0.0%	4.9%
`l.s`	0.0%	4.2%
`s.s`	0.0%	1.1%
`lhu`	1.3%	0.0%

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation for reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - MIPS core and arithmetic core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

Assignment 3

- Homework assignment

  3.3, 3.8 ,3.10, 3.11(.1~.4)

- To be submitted in the next week class

- Read Section 3.10 of Chapter 3

  Historical Perspective and Further

Reading

Project 1

1. 项目目标

(1) 深入掌握二进制数的表示方法以及不同进制数的转换；

(2) 掌握二进制不同编码的表示方法；

(3) 掌握IEEE 754 中单精度浮点数的表示和计算。

2. 具体要求

假设没有浮点表示和计算的硬件，用软件方法采用仿真方式实现IEEE 754单精度浮点数的表示及运算功能，具体要求如下：

(1) 程序需要提供人机交互方式（GUI或者字符界面）供用户选择相应的功能；

(2) 可接受十进制实数形式的输入，在内存中以IEEE 754单精度方式表示，支持以二进制和十六进制的方式显示输出；

(3) 可实现浮点数的加减乘除运算；

(4) 可以使用80X86或MIPS或ARM汇编指令，但是不能使用浮点指令，只能利用整数运算指令来编写软件完成。

3. 开发工具

利用汇编语言来实现，可在一个程序中实现所有功能，也可分解为若干个程序，但是必须完成2中的所有功能。

4. 提交资料

按照学校实验报告的格式，要求提交源程序、设计文档以及编译后可以执行程序的电子版。

5. 提交时间

期末考试前一周。