NUMBERS IN COMPUTERS

So far, we dealt only with natural numbers 0, 1, 2, ...
We need techniques to deal with:
negative numbers,
capacity of a number representation,
fractions and real numbers.

$x = \sum b_i 2^i$ for $i = 0$ to $n - 1$; n being the number of digits

$1011_{two} = 11_{ten}$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

MIPS words are 32 bits long, but we continue the discussion with 16 bits half words. To clarify the notation, the leftmost bit has the highest number and is called the *most significant bit* (MSB). Bit 0 is the *least significant bit* (LSB).

There are several techniques to represent negative numbers. The method almost universally used is the *two's complement representation*.

TWO's COMPLEMENT REPRESENTATION

Zero is uniquely represented by

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Because an even number of numbers that can be represented with a fixed number of bits, we have a skewed representation.

```
0000 0000 0000 0000 = 0ten
0000 0000 0000 0001 = 1 ten
0000 0000 0000 0010 = 2 ten
0000 0000 0000 0011 = 3 ten
...
0111 1111 1111 1110 = +32766 ten
0111 1111 1111 1111 = +32767 ten
1000 0000 0000 0000 = -32768 ten
1000 0000 0000 0001 = -32767 ten
...
1111 1111 1111 1101 = -3 ten
1111 1111 1111 1110 = -2 ten
1111 1111 1111 1111 = -1ten
```

Properties of the two's complement representation:

- $x + (-x) = 2^n$ (which truncates to zero)
- single zero
- all negative numbers have the MSB equal to 1 (sign bit)

- $x = b_{15} (-2)^{15} + b_{14} 2^{14} + b_{13} 2^{13} + ... + b_0 2^0$

SHORTCUTS

•To negate a given number. Invert every bit and add 1.

```
0000  0000  0000  0010  =  2 ten
1111  1111  1111  1101                    <- add 1
1111  1111  1111  1110  =  -2 ten
0000  0000  0000  0001                    <- add 1
0000  0000  0000  0010  =  2 ten
```

•Convert to numbers of different size:

```
                         0000 0000 0000 0010 = 2 ten  (16 bits)
0000  0000  0000  0000  0000  0000  0000  0010  =  2 ten  (32 bits)

                          1111 1111 1111 1110 = -2 ten
1111  1111  1111  1111  1111  1111  1111  1110  =  -2 ten
```

The bit sign is copied to capacity if the new word is longer. The number is truncated if the new word is shorter.

•Shifting left or right performs multiplication or division by powers of two and preserves the sign if zeros are entered on the right and the sign bit propagated on the left:

```
0000  0000  0000  0000  0000  0000  0000  0011  =  3 ten
0000  0000  0000  0000  0000  0000  0000  0110  =  6 ten

1111  1111  1111  1111  1111  1111  1111  1111  =  -1 ten
1111  1111  1111  1111  1111  1111  1111  1110  =  -2 ten
1111  1111  1111  1111  1111  1111  1111  1100  =  -4 ten
1111  1111  1111  1111  1111  1111  1111  1000  =  -8 ten
```

SIGNED AND UNSIGNED

Signed numbers are not always needed. For example, memory addresses run from 0 to the largest. In the C language, integers can be declared to be *unsigned*.

The result of a comparison will depend on whether the numbers are assumed to be signed or unsigned. We need a new compare instruction to account for that. We had:

```
slt   $1, $2, $3
slti  $1,$2,100
```

We add two new instructions:

```
sltu  $1, $2, $3
sltiu $1,$2,100
```

Example:

```
            1111 1111 1111 1111 1111 1111 1111 1111
```

means -1 if signed number, 4,294,967,295 if unsigned.

Comparing these two numbers to 1 will yield different results, whether they are considered to be signed or unsigned.

However, the hardware that performs additions and subtractions does not depend on whether the numbers are signed or unsigned, because the subtraction is achieved by negating a number and then adding. More on that later.

ADDITION AND SUBTRACTION

$$\begin{array}{cl} & 0000\ 0000\ 0000\ 0111 = 7_{ten} \\ + & \underline{0000\ 0000\ 0000\ 0110} = 6_{ten} \\ & 0000\ 0000\ 0000\ 1101 = 13_{ten} \end{array}$$

Subtraction works the same with the manual method or via the two's complement and add.

$$\begin{array}{cl} & 0000\ 0000\ 0000\ 0111 = 7_{ten} \\ + & \underline{1111\ 1111\ 1111\ 1010} = -6_{ten} \\ & 0000\ 0000\ 0000\ 0001 = 1_{ten} \end{array}$$

One problem is that the sum of two numbers (signed or unsigned) may exceed the capacity of the registers.

$$\begin{array}{cl} & 0111\ 1111\ 1111\ 1111 = +32767_{ten} \\ + & \underline{0000\ 0000\ 0000\ 0010} = +\qquad 2_{ten} \\ & 1000\ 0000\ 0000\ 0001 = -32767_{ten} \end{array}$$

Which is obviously wrong. We would need one more bit. The same thing could occur with two negative numbers. However, it cannot occur with numbers of opposite signs. One method to detect occurrence of *overflow* is to check the validity of the sign of the result. Overflow conditions:

Operation	A	B	Result
A + B	>= 0	>= 0	>= 0
A + B	< 0	< 0	< 0
A - B	>= 0	< 0	>= 0
A - B	< 0	>= 0	< 0

ACCOUNTING FOR OVERFLOWS

When an overflow is detected, it may create an *exception,* that is, an abnormal condition. The way the computer handles exceptions is similar to procedures, with the difference that it is unplanned (it jumps to a fixed address outside the control of the user). Later we willl see the hardware needed to handle exceptions. For now, we introduce new instructions which have the property of ignoring exceptions.

```
addu     $1, $2, $3
addiu    $1, $2, 100
subu     $1, $2, $3
```

What is the connection between unsigned numbers and overflow? Unsigned numbers can create overflows like the others but the computer designers chose to ignore them. So actually addu, addiu and subu do exactly the same operation than add, addi and sub, but do not cause exceptions.

LOGICAL OPERATIONS

There is a class of operations to manipulate the bits inside a word without attributing a special meaning to them. There called *logicals*.

First, *shifts*: bits are moved a number of places left of right.

```
0000 0000 0000 0000 0000 0000 0000 1010
```

shifted left eight places yields

```
0000 0000 0000 0000 0000 1010 0000 0000
```

Providing for the shift right requires two new instructions:

```
sll $10, $16, 8         # $10 = ($16 << 8)
srl $10, $16, 8         # $10 = ($16 >> 8)
```

This explains the necessity for the shamt (shift amount) field in the R format instruction.

op	rs	rt	rd	shamt	funct
0	0	16	10	8	0

There exist other kinds of shifts. The most important is the shift right arithmetic which preserves the sign bit (sra) and rotate left and right (rol, ror). Example of shift right arithmetic:

```
1111 0000 0000 0000 0000 1010 0000 0000
1111 1111 0000 0000 0000 0000 1010 0000
```

Other logical operations have to do with the logical operators, AND, OR, (NOR, XOR, NOT). Each bit of the result is the result of a logical operation between each bit of the operands.

Example:

```
        0000 1010 0000 1000 0000 1010 0000 0000
AND     0000 0000 0000 0000 1111 1111 1111 1111
        ---------------------------------------
        0000 0000 0000 0000 0000 1010 0000 0000
```

This is often used to apply a *mask*, which forces all bits to zero except those set to 1 in the mask.

Similar operation for the OR (to combine portions of a word)

```
        0000 1010 0000 1000 0000 1010 0000 0000
OR      0000 0000 0000 0000 1111 1111 1111 1111
        ---------------------------------------
        0000 1010 0000 1000 1111 1111 1111 1111
```

We of course have instructions as well as the "immediate" versions:

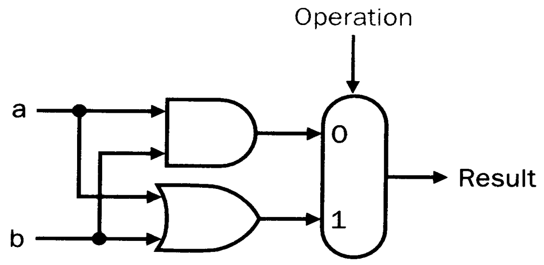
```
and  $1, $2, $3          # $1 = $2 & $3
or   $1, $2, $3          # $1 = $2 | $3
andi $1, $2, 100         # $1 = $2 & 100
ori  $1, $2, 100         # $1 = $2 | 100
```

Note: In addi, the 16 bit constant was signed extended to preserve the sign. In andi and ori, the 16 bit constant is padded with leading 0s.

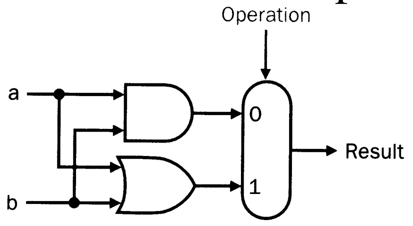
CONSTRUCTION OF ALU

The Arithmetic and Logic Unit is combinatorial logic block which carries out the operations specified by instructions. It take two 32 bit operands and produces a 32 bit result according to the values of control lines.

1 bit ALU AND/OR



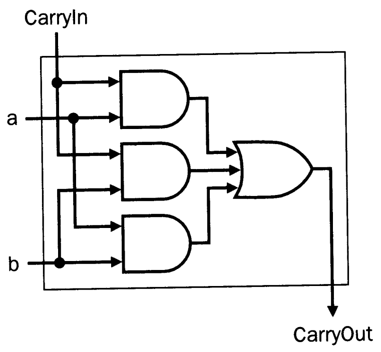
ADDITION: for each bit, the result is found by adding two bits and the carry from the preceding bit. There are two results: the sum and the carry, so there are three inputs two outputs.

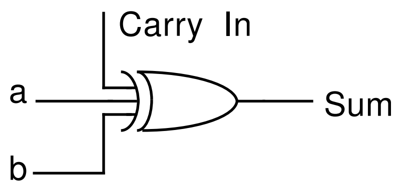


Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	00
0	0	1	0	1	01
0	1	0	0	1	01
0	1	1	1	0	10
1	0	0	0	1	01
1	0	1	1	0	10
1	1	0	1	0	10
1	1	1	1	1	11

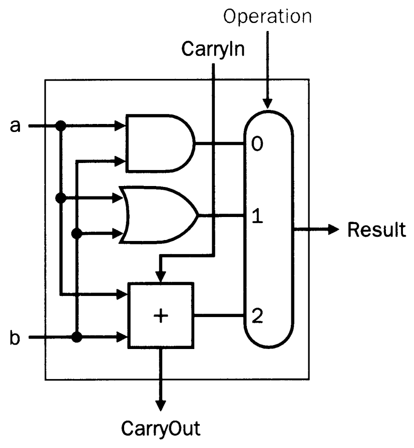
Once simplified: $\text{CarryOut} = (b \,.\, \text{CarryIn}) + (a \,.\, \text{CarryIn}) + (a \,.\, b)$

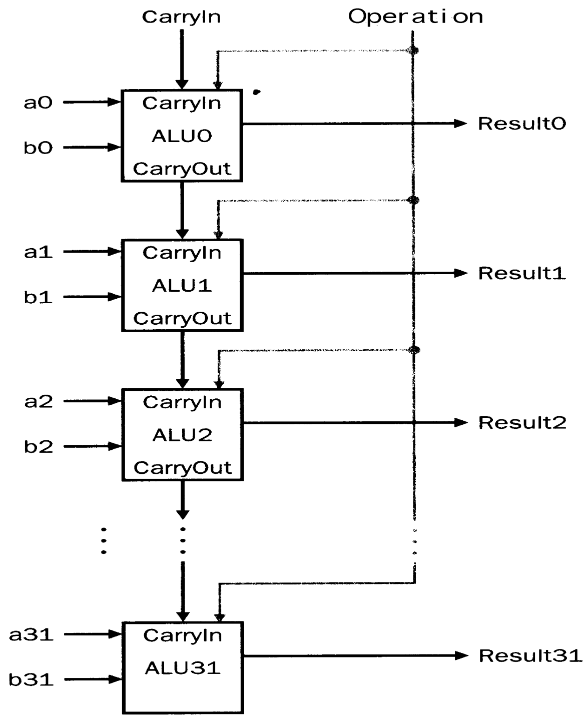
$$\text{Sum} = (\bar{a} \,.\, b \,.\, \overline{\text{CarryIn}}) + (a \,.\, \bar{b} \,.\, \overline{\text{CarryIn}}) + (\bar{a} \,.\, \bar{b} \,.\, \text{CarryIn}) + (a \,.\, b \,.\, \text{CarryIn})$$
$$= (a \text{ xor } b) \text{ xor CarryIn}$$





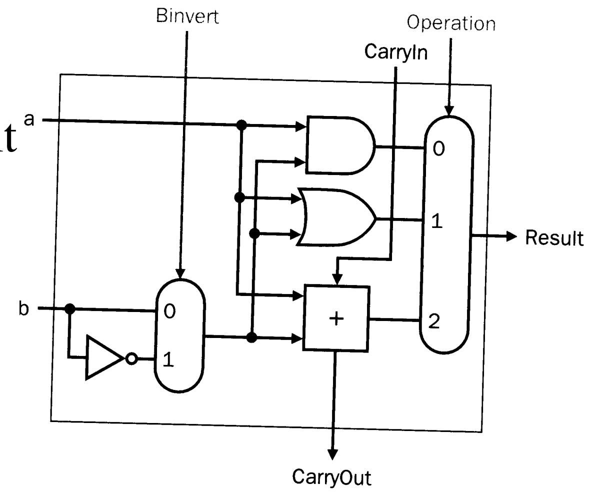
Connecting all the blocks together leads to a *ripple adder* because the carry may propagate all the way through.





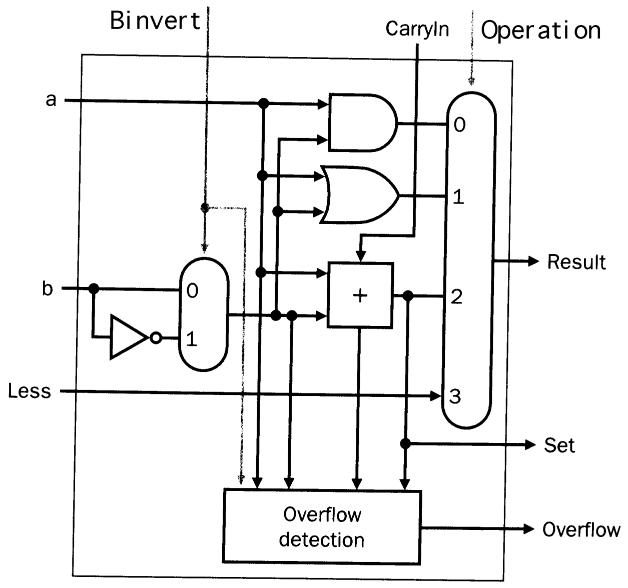
SUBTRACTION

First we change the sign of operand b, and then add 1. It takes an extra inverter and multiplexor. The unused CarryIn input of the least significant bit is used to supply the 1.

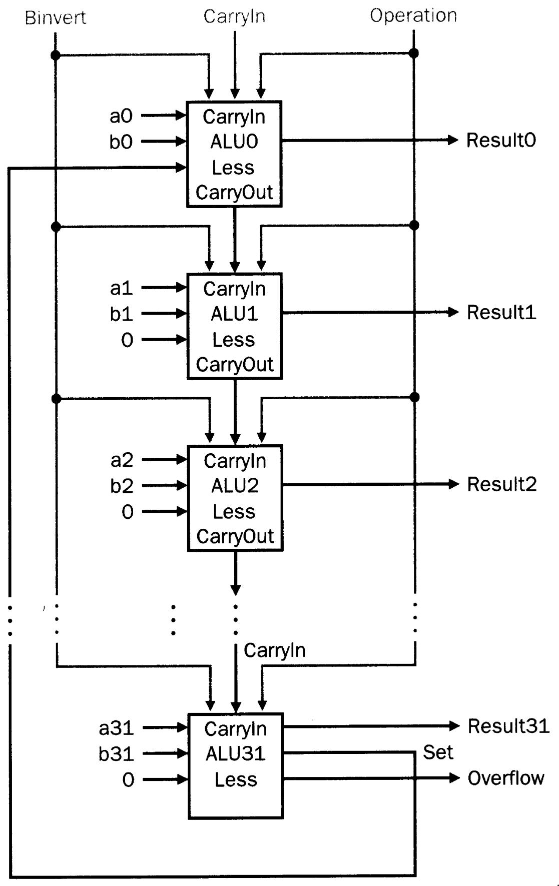


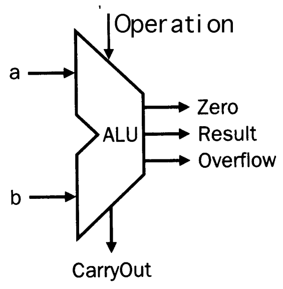
COMPARISON

We need a special 1-bit ALU for the most significant bit so that the result is 1 if the a - b < 0 and 0 otherwise.



Here are the 1-bit cells connected together with a zero-result detector.





ALU and summary of the control lines:

CONTROL LINES	FUNCTION
000	And
001	Or
010	Add
110	Subtract
111	Set-less-than

CARRY LOOK AHEAD

We need to speed up addition, which finishes only when the carry has rippled through. In theory we can calculate the values of any carry using only two gate levels.

$$c2 = (b1 \,.\, c1) + (a1 \,.\, c1) + (a1 \,.\, b1)$$
$$c1 = (b0 \,.\, c0) + (a0 \,.\, c0) + (a0 \,.\, b0)$$

so

$$\begin{aligned} c2 = {} & (a1 \,.\, a0 \,.\, b0) + (a1 \,.\, a0 \,.\, c0) + (a1 \,.\, b0 \,.\, c0) + \\ & (b1 \,.\, a0 \,.\, b0) + (b1 \,.\, a0 \,.\, c0) + (b1 \,.\, b0 \,.\, c0) + (a1 \,.\, b1) \end{aligned}$$

For a wide adder, the size of the logic block would be enormous. Carry look ahead is a more economical technique:

$g_i = a_i \,.\, b_i$ *generates* a carry regardless of CarryIn

$p_i = a_i + b_i$ *propagates* a CarryIn to a CarryOut

We rewrite the carry expression in terms of g_i and p_i

$$c1 = g0 + (p0 \,.\, c0)$$
$$c2 = g1 + (p1 \,.\, g0) + (p1 \,.\, p0 \,.\, c0)$$
$$c3 = g2 + (p2 \,.\, g1) + (p2 \,.\, p1 \,.\, g0) + (p2 \,.\, p1 \,.\, p0 \,.\, c0)$$
$$c4 = g3 + (p3 \,.\, g2) + (p3 \,.\, p2 \,.\, g1) + (p3 \,.\, p2 \,.\, p1 \,.\, g0) + (p3 \,.\, p2 \,.\, p1 \,.\, p0 \,.\, c0)$$

We see that for *n* bits, we have *n+1* terms and the longest term has *n+1* factors, where in the previous case there was a geometrical growth.

For a large adder, it is still not practical.

One idea is to build n-bit adder blocks and to ripple the carry from block to block. These blocks are often 4-bit adders which have only one CarryIn and one CarryOut.

One other idea is to pursue the idea further. The propagate signals can be calculated for 4-bit adder blocks:

P0 = p3 . p2 . p1 . p0
P1 = p7 . p6 . p5 . p4
P2 = p11 . p10 . p9 . p8
P3 = p15 . p14 . p13 . p12

The generate signals will be true if all the bits will generate a carry from the most significant bit of the group:

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)

Then

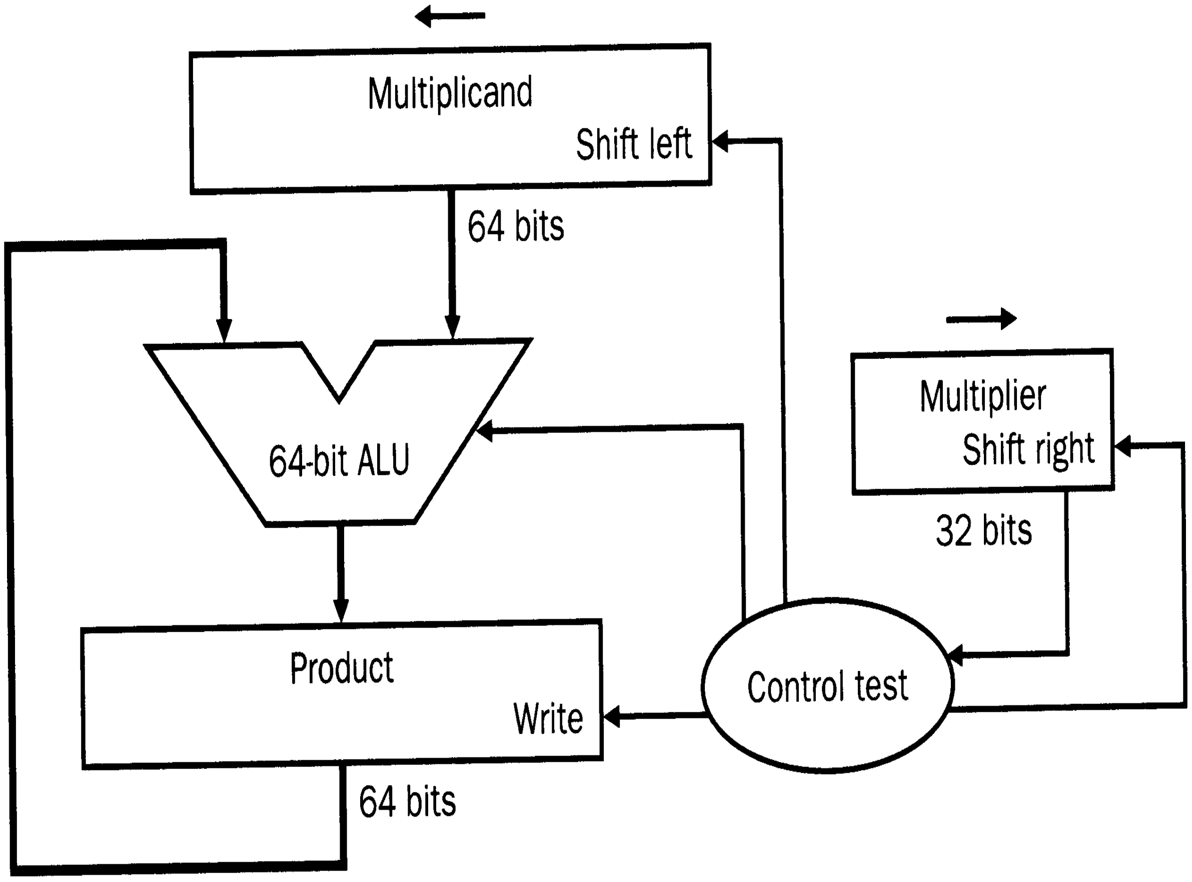
C1 = G0 + (P0 . c0)
C2 = G1 + (P1 . G0) + (P1 . P0 . c0)
C3 = G2 + (P2 . G1) + (P2 . P1 . G0) + (P2 . P1 . P0 . c0)
C4 = G3 + (P3 . G2) + (P3 . P2 . G1) + (P3 . P2 . P1 . G0) + (P3 . P2 . P1 . P0 . c0)

as before.

MULTIPLICATION

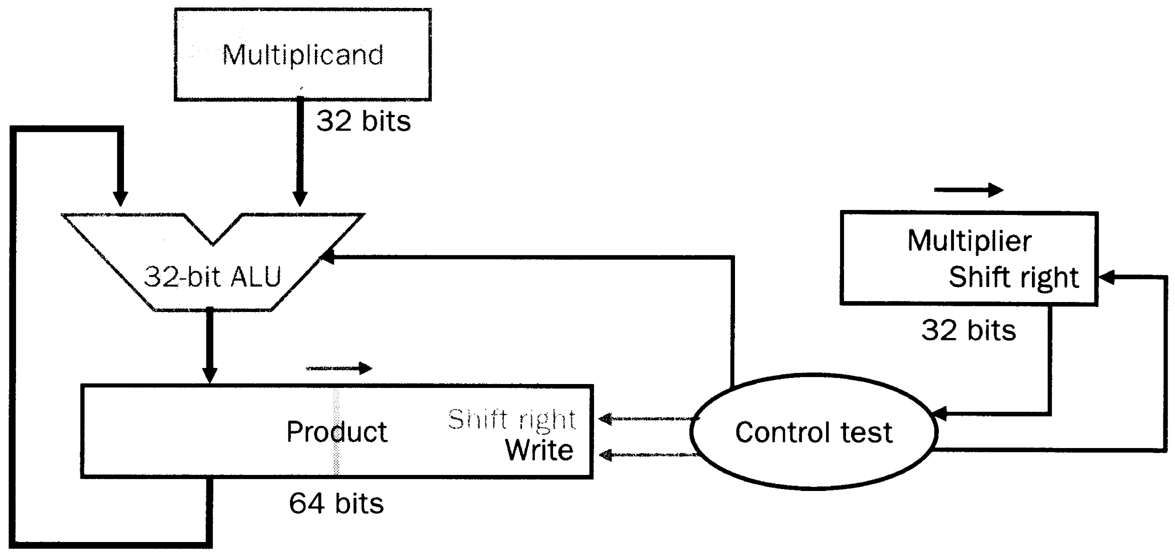
```
   1000          Multiplicand (8)
   1001          Multiplier    (9)
      1000
     0000
    0000
  1000
  01001000       Product        (72)
```

Place the multiplicand if the multiplier's digit is 1 or 0 otherwise. Turn it into hardware by shifting right the multiplier to test the successive bits and shifting left the multiplicand.

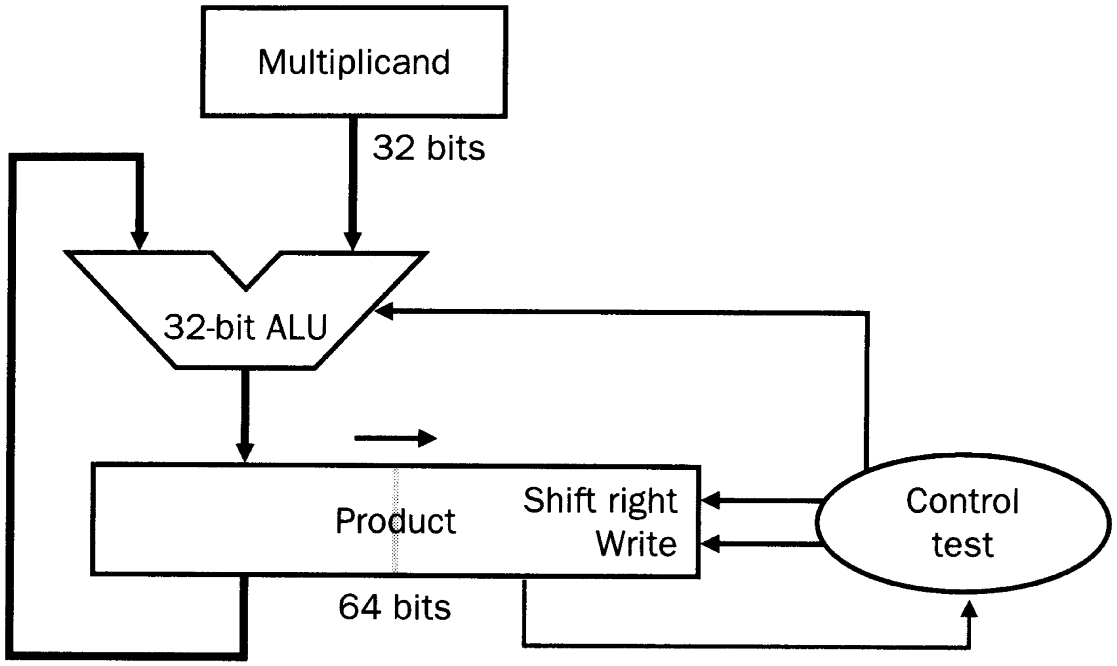


IMPROVEMENT

Instead of shifting the multiplicand left where it ends up using 64 bits, shift the product right. We only need a 32 bit adder.



Further saving is achieved by combining the multiplier and the product.



ALGORITHM

0. **Init.**
   Place Multiplicand in Multiplicand register.
   Place Multiplier in lower half of Product register.
   Clear upper half of Product register.

1. **Test** and **Update**.
   If the LSB of Product register is 1, add Multiplicand register to upper half of Product register and place result back in the upper half of Product register.

2. **Shift.**
   Product register shifted right one 1 bit.

3. **Count.**
   If 32nd repetition then **done** else goto step 1.

SIGNED MULTIPLICATION

Instead of calculating the sign separately from the signs of the operands and multiply positive numbers, there is a way to do it directly.

```
                 action  (cond)                       action  (cond)
      0010                              0010
      0110                              0110
      0000       shift  (0)             0000      shift  (00)
     0010        add    (1)            0010       sub    (10)
    0010         add    (1)           0000        shift  (11)
   0000          shift  (0)          0010         add    (01)
   00001100                          00001100
```

2 x 6 = 12 2 x (-2 + 8) = 12

A string of n 1's is replaced by 1 followed by n 0's. This corresponds to adding 1 to the multiplier.

This is easily compensated by pre-subtracting the multiplicand at the beginning of the sequence of 1's.

Booth's Algorithm:

0000**01**11**10**00 **0111 0**

^ ('bit number -1' initialized at zero)

00: do nothing.
01: add the multiplicand to the left half of the product.
10: subtract the multiplicand from the left half of the product.
11: do nothing.

Pros: Can skip over strings of 1's Cons: Bad for 101010...

Why does this apply to signed numbers? Call *a* the multiplier and *b* the multiplicand. a_i and b_i are the bits of these numbers.

$(a_{i-1} - a_i) = 0, +1$ or -1 (do nothing, add *b*, subtract *b*)

The Product can be written:

$(a_{-1}-a_0)\ b + (a_0-a_1)\ b\ 2^1 + (a_1-a_2)\ b\ 2^2 + \ldots + (a_{30}-a_{31})\ b\ 2^{31}$

$b\ [a_{-1} - a_0 + a_0\ 2^1 - a_1\ 2^1 + a_1\ 2^2 - a_2\ 2^2 + \ldots + a_{30} 2^{31} - a_{31}\ 2^{31}\ ]$

but $- a_i\ 2^{i-1} + a_i\ 2^i = a_i\ 2^{i-1}$

$b\ [a_0 + a_1\ 2^1 + a_2\ 2^2 + \ldots + a_{31}\ (-2^{31})] = b\ a$

The same reasoning could be carried out swaping *a* and *b*.
The Booth algorithm does nothing but implements the encoding of two's complement numbers!

ALGORITHM

0. **Init.**
   Place Multiplicand in Multiplicand register.
   Place Multiplier in lower half of Product register.
   Clear upper half of Product register.
   Clear b_{-1} ("bit number -1").

1. **Test** and **Update**. Check LSB and b_{-1}:
   00: do nothing.
   01: add multiplicand to the left half of product.
   10: subtract the multiplicand from the left half of product.
   11: do nothing.

2. **Shift.**
   Product register arithmetic shift right one 1 bit into b_{-1}

3. **Count.**
   If 32nd repetition then **done** else goto step 1.

EXAMPLE

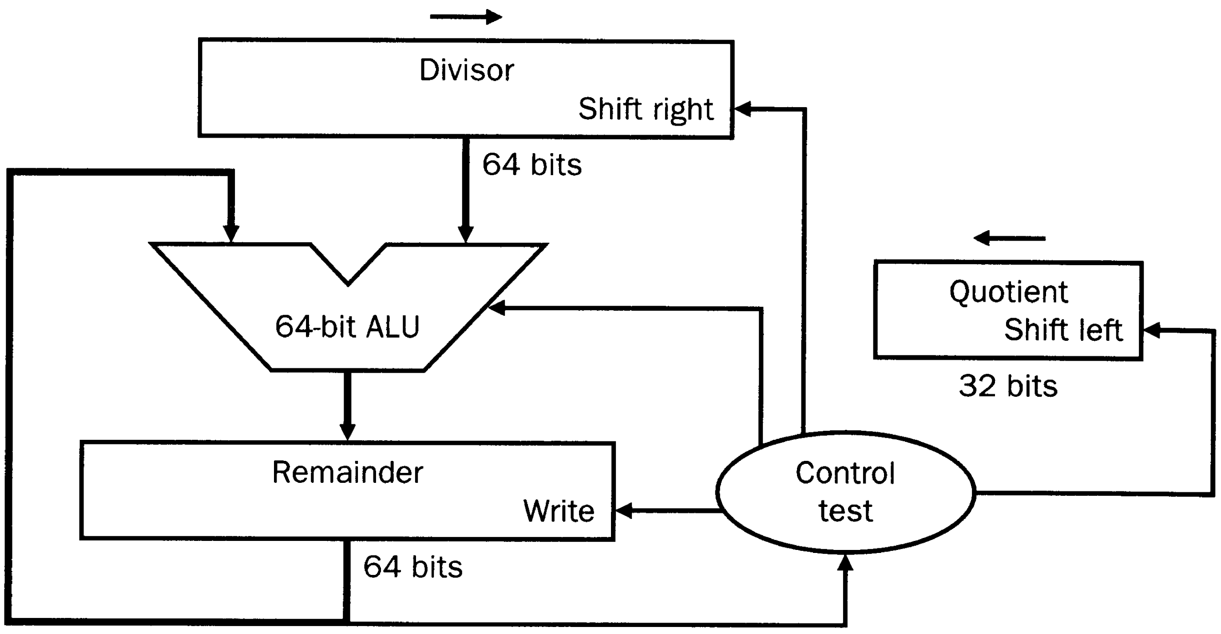
5 (0101)	X -3 (1101)		-5 (1011)	X -6 (1010)	
	0000 1101	0		0000 1010	0
subtract	1011 1101	0	nop	0000 1010	0
shift	1101 1110	1	shift	0000 0101	0
add	0010 1110	1	subtract	0101 0101	0
shift	0001 0111	0	shift	0010 1010	1
subtract	1100 0111	0	add	1101 1010	1
shift	1110 0011	1	shift	1110 1101	0
nop	1110 0011	1	subtract	0011 1101	0
shift	1111 0001	1	shift	0001 1110	1

DIVISION

```
                              1001             Quotient
Divisor   1000       |   1001010               Dividend
                         1000
                           10
                           101
                           1010
                           1000
                              10      Remainder

                       1001010  | 1000
                       1000       1001
                           1010
                           1000
                             10
```

Hardware (divide-by-zero check not represented):



ALGORITHM

0. **Init.**
   Place divisor in the left half of the Divisor register.
   Place dividend in the Remainder register.
   Clear the Quotient register.

1. **Attempt to fit divisor in dividend**.
   Subtract the Divisor register from the Remainder register, place the result in the Remainder register.

2a. **If the result is zero or positive**.
   Shift Quotient register to the left setting rightmost bit to 1.

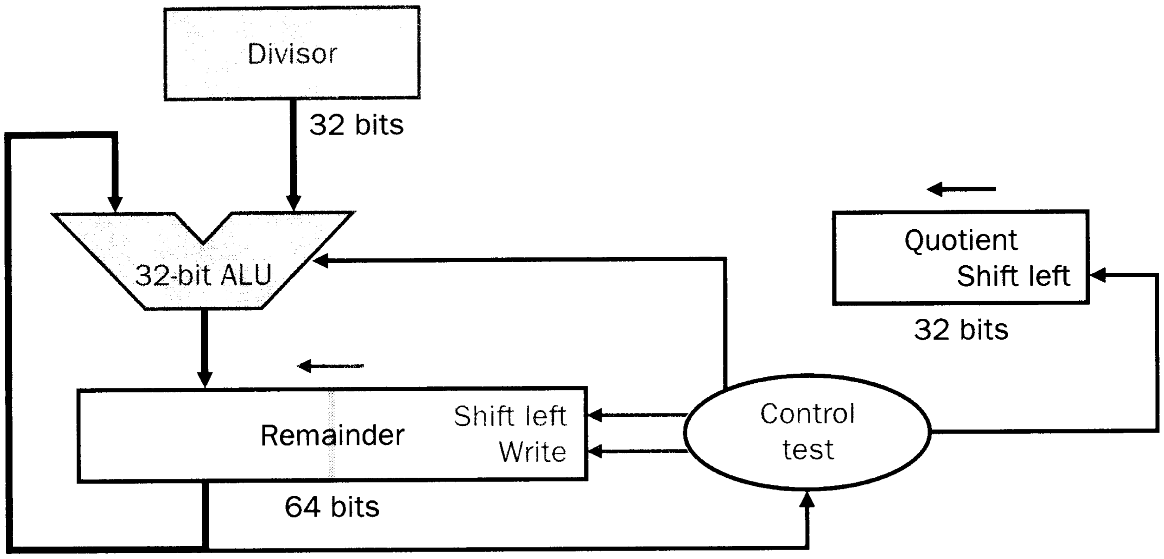
2.b **If the result is negative strictly.**
   Restore initial value by adding Divisor register to the Remainder register leaving the sum there.
   Shift Quotient register to the left setting rightmost bit to 0.

3. **Shift divisor**.
   Shift Divisor register to the right.

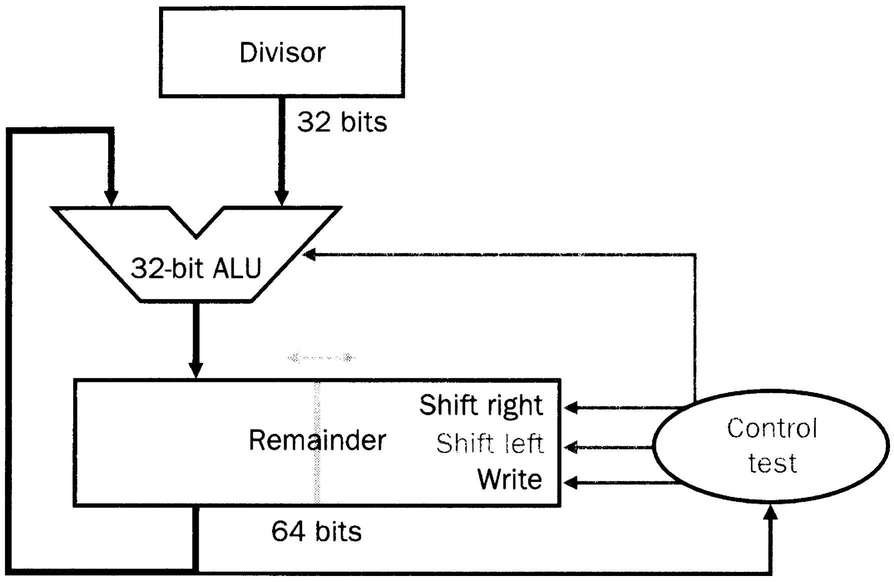
4. **Count.**
   If 33nd repetition then **done** else goto step 1.

IMPROVEMENT

As for the multiplication, only a 32 bit ALU is needed to perform the successive subtractions.



Finally, we can eliminate the Quotient register by putting the result in the same register.



ALGORITHM

0. **Init.**
   Place divisor in the Divisor register.
   Place dividend in the Remainder register.

1. **Shift**.
   Shift Remainder register to the left 1 bit.

2 **Subtract**.
   Subtract Divisor register from the upper half of Remainder register and leave result there.

3a. **If the result is zero or positive**.
   Shift Remainder register left setting righmost bit to 1.

3b. **If the result is negative strictly.**
   Restore initial value by adding Divisor register to the upper part of the Remainder register leaving the sum there.
   Shift Remainder register left setting rightmost bit to 0.

4. **Count.**
   If 32nd repetition then **done** else goto step 2.

Done. Shift upper part of Remainder register right 1 bit.

SIGNED DIVISION

There is no simple Booth algorithm equivalent. First record the sign the divisor and the divident, divide the absolute values, and negate the sign of quotient if the signs are different.

The remainder as the same sign as the sign of the dividend.

Dividend = Quotient x Divisor + Remainder

+	+	+	+
+	-	-	+
-	-	+	-
-	+	-	-

INSTRUCTIONS

The same hardware can be used to perform multiplication and division. Only the control differs. The MIPS machine calls Hi and Lo the respective halfs of the 64 register used to store the 64 bit product or the quotient/remainder. Instructions:

```
mult     $2, $3        # Hi,Lo = $2 x $3
multu    $2, $3        # same but unsigned (no overflow)
div      $2, $3        # Lo = $2 / $3, Hi = $2 mod $3
divu     $2, $3        # same but unsigned (no overflow)
mfhi     $1            # $1 = Hi
mflo     $1            # $1 = Lo
```

So we add two more registers, Hi and Lo, to the 32 general purpose registers.

FLOATING POINT

Real numbers include rationals and irrationals. In either case we must settle for a fractional *approximation*. For a rational, the number of digits needed to represent it may be arbitrarily large. For an irrational, the number of digits is not finite.

A *normalized* number in ordinary scientific notation has just one non zero digit to the left of the dot.

2.997925 10^8 m/s is the speed of light. Same applies to binary:

+-	1.sssssss	2^{eeee}
Sign	Fraction	Exponent

1	8	23
S	E (Exponent)	Z (Significand = Fraction - 1)
s	eeeeeeee	ssssssssssssssssssssssssssssssssssssssssssssss

$F = 1 + Z \qquad X = (-1)^S \; F \; 2^E$

The bits allocated to E and F trade range against precision. Here, the range is about from 2.0 10^{-38} to 2.0 10^{38}. This is by no means infinite, so *overflow* and *underflow* can occur. To offer greater range and precision computer hardware support *double precision* numbers (11 bit exponent and 52 bit fraction on two words). Many computers use different encodings, but this one has become the *IEEE754 Standard*.

In C, a single precision number is a `float` and a double precision number is a `double`.

PROPERTIES

This representation can share some of its operations with integer operations.

The sign bit is in the same place so sign test work the same.

Sorting floating point numbers should be the same operation as sorting fixed point numbers. It would work fine if the exponent could not be negative. But if we use two's complement representation, the scheme breaks down.

The idea is to use a *biased* notation to make the most negative exponent look like the smallest. With an 8 bit exponent, the bias is to be 127 (1023 for 11 bits):

0000 0000	-127
0000 0001	-126
...	
0111 1110	-1
0111 1111	0
1000 0000	+1
...	
1111 1111	128

In general: $X = (-1)^S \ (1 + \text{significand}) \ 2^{(\text{exponent - bias})}$

FLOATING POINT ADDITION

1. **Adjust exponent**.
   Shift the smallest number to the right
   (multiply by powers of 2) until its exponent
   matches the largest.

2. **Add significands**.

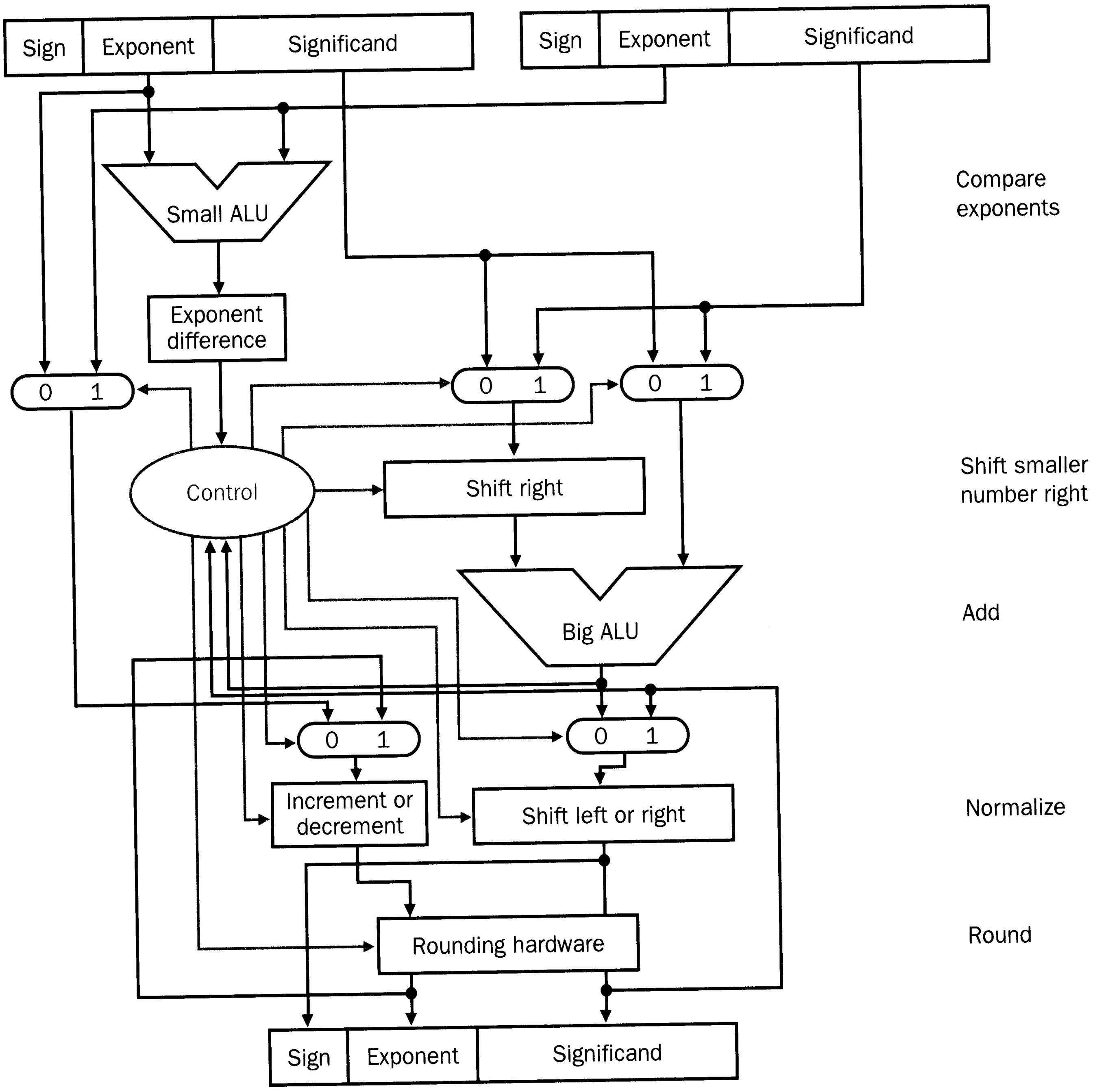
3. **Normalize the sum**. The result must have the form:

   $$+- 1.sssssss\ 2^{eeee}$$

   Shift the significand left (resp. right) while decrementing
   (resp. incrementing) the exponent.
   The biased exponent should be between 0 and 255,
   if it not, **cause an exception** underflow (resp. overflow).

4. **Round**. (discussion later)
   If not Normalized, goto step 3.

HARDWARE TO SUPPORT FLOATING POINT ADDITION



FLOATING POINT MULTIPLICATION

1. **Add exponents**.
   Since they are biased, we would add the bias twice, so the result's biased exponent is:

   E1 + E2 - bias.

2. **Multiply significands**. (unsigned)

3. **Normalize Product**.
   Here normalization will be acheived by shifting right and incrementing the exponent.
   Check for the exponent and **cause an overflow** if capacity exceeded. Note that an **underflow** can occur if both operands have large negative exponents.

4. **Round**. (discussion later)
   If not Normalized, goto step 3.

5. **Sign.**
   Deternine the sign the product from the signs of the operands.

Clearly, there is a lot in common between the hardware for addition and multiplication (rounding and normalizing).

FLOATING POINT INSTRUCTIONS

MIPS supports (like many machines) special registers for storing floating point numbers, named $f0, $f1, ..., $f31.
MIPS supports single and double precision operations and corresponding loads and stores.

To simplify, only even numbered registers can be used for single precision operations. The operands are always even numbered, registers are only used in pairs.

```
add.s     $f2,$f4,$f6     # $f2 = $f4 + $f6
sub.s     $f2,$f4,$f6     # $f2 = $f4 - $f6
mul.s     $f2,$f4,$f6     # $f2 = $f4 * $f6
div.s     $f2,$f4,$f6     # $f2 = $f4 / $f6
add.d     $f2,$f4,$f6     # $f2 = $f4 + $f6
sub.d     $f2,$f4,$f6     # $f2 = $f4 - $f6
mul.d     $f2,$f4,$f6     # $f2 = $f4 * $f6
div.d     $f2,$f4,$f6     # $f2 = $f4 / $f6
```

Loads and stores of single and double precision numbers are all pseudoinstructions (See Appendix A) involving transfers between the coprocessor and the CPU.

```
l.s       $f2,Address
s.s       $f2,Address
l.d       $f2,Address
s.d       $f2,Address

bc1t      100             # if (cond ==1) goto PC+4+100
bc0t      100             # if (cond ==0) goto PC+4+100
c.lt.s    $f2,$f4         # if ($f2 < $f4) cond = 1 else cond = 0
c.lt.d    $f2,$f4         # if ($f2 < $f4) cond = 1 else cond = 0
```

ACCURATE ARITHMETIC

The number of reals between, say 0 and 1, is not finite, but the number of floating point numbers is (there are less than 2^{53} of them). Rounding (finding the closest approximation) can be done in several ways.

First, every intermediate result in floating point addition and multiplication has to be truncated. The IEEE 754 standard specifies that 2 extra bits (*guard* and *round*) must be kept during intermediate calculations.

Here is an example in decimal. Numbers have 3 decimal digits but we keep two more during calculations.

$2.56\ 10^0 + 2.34\ 10^2$

$$\begin{array}{rl} & 2.3400\ \ 10^2 \\ + & \underline{0.0256\ }\,10^2 \\ & 2.3656\ \ 10^2 \end{array}$$

Now we round two digits, 00--49 down and 50--99 up.
Result: $2.37\ 10^2$

Without the extra digits, the result is: $2.36\ 10^2$

The accuracy is measured in the number of bits in error in the LSB's of the significand (*units in the last place*, or *ulp*). The standard guarantees better than one-half ulp.

FEATURES OF IEEE 754

Representation of +∞ or -∞ (result of divide by zero instead of an exception).

Representation of the result of other illicit operations (such as 0/0 or ∞-∞), printed as NaN (Not a Number).

Unnormalized forms. The exponent field is zero but the significand gains zeros after the dot. There is a gradual underflow between the smallest normalized number and 0.

Not all hardwares implement those features, but at least they specify whether they are implemented or not.

The encoding is defined as follows:

Single precision		Double precision		Object
Exponent	Significand	Exponent	Significand	
0	0	0	0	0
0	nonzero	o	nonzero	Denormalized number
1 to 254	anything	1 to 2046	anything	Floating point number
255	o	2047	0	Infinity
255	nonzero	2047	nonzero	NaN

CHARACTERS

The most common standard is the ASCII (American Standard Code for Information Interchange) representation, to associate numbers to characters.

32	20	sp	48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
34	22	“	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
39	27	‘	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	

There are a number of invisible characters such as “escape,” “carriage return,” etc... including the “control characters.” One of particular importance is the “null” character which has the value 0 and which marks the end of a string in C.

HEXADECIMAL NOTATION

16 digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F.

Very practical to represent compactly binary numbers:

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

For example: F A 0 3 is readily translated into
1111 1010 0000 0011 (and vice-versa)