Chapter 3
Arithmetic for Computers

Outline

- Signed and unsigned numbers (Sec. 3.2)
- Addition and subtraction (Sec. 3.3)
- Multiplication (Sec. 3.4)
- Division (Sec. 3.5)
- Floating point (Sec. 3.6)

Representation of Unsigned Number

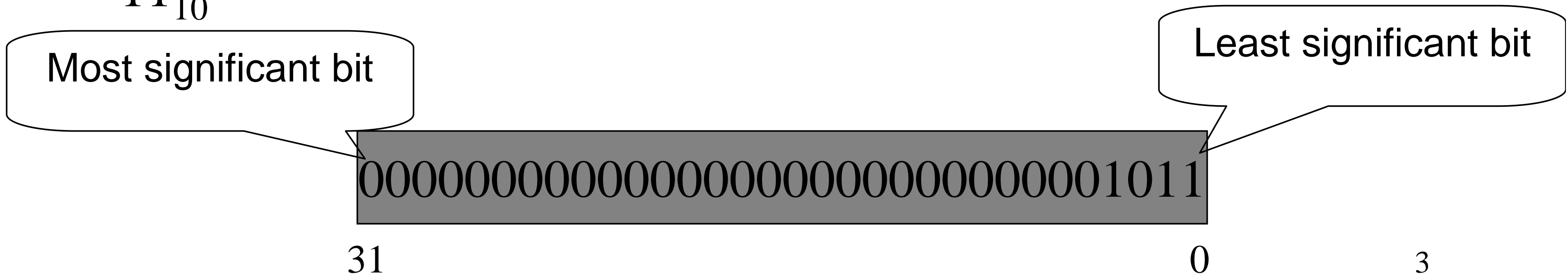
$$d*Base^i$$

- The value of *i*th digital *d*
  - *i* starts at 0 and increase from right to left
- Ex: 1011_{two}

  $(1*2^3) + (0*2^2) + (1*2^1) + (1*2^0)_{10}$

  $=8+0+2+1_{10}$

  $=11_{10}$



Representing Numbers: Review

- *32-bit binary representation of (unsigned) number:*
  - $b_{31} \times 2^{31} + b_{30} \times 2^{30} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$
  - *One billion ($1{,}000{,}000{,}000_{10}$) in binary is*

$$0011\ 1011\ 1001\ 1010\ 1100\ 1010\ 0000\ 0000_2$$

$$2^{28} \quad 2^{24} \quad 2^{20} \quad 2^{16} \quad 2^{12} \quad 2^{8} \quad 2^{4} \quad 2^{0}$$

$$= 1{\times}2^{29} + 1{\times}2^{28} + 1{\times}2^{27} + 1{\times}2^{25} + 1{\times}2^{24} + 1{\times}2^{23} + 1{\times}2^{20} + 1{\times}2^{19}$$
$$+ 1{\times}2^{17} + 1{\times}2^{15} + 1{\times}2^{14} + 1{\times}2^{11} + 1x2^{9}$$

$$= 536{,}870{,}912 + 268{,}435{,}456 + 134{,}217{,}728 + 33{,}554{,}432 +$$
$$16{,}777{,}216 + 8{,}388{,}608 + 1{,}048{,}576 + 524{,}288 + 131{,}072 +$$
$$32{,}768 + 16{,}384 + 2{,}048 + 512 = 1{,}000{,}000{,}000$$

What If Too Big?

- Binary bit patterns are simply representations of numbers.
- Numbers really have an infinite number of digits (non-significant zeroes to the left).
  - with almost all being zero except for a few of the rightmost digits.
  - Don't normally show leading zeros.
- If result of add (or any other arithmetic operation) cannot be represented by these rightmost hardware bits, overflow is said to have occurred.
- Up to Compiler and OS what to do.

How to Avoid Overflow? Allow It Sometimes?

- Some languages detect overflow (Ada, Fortran), some don't (C)
- MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:
  - add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow
  - add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do not cause exceptions on overflow
    - **unsigned integers commonly used for address arithmetic where overflow ignored**
    - **MIPS C compilers always produce `addu`, `addiu`, `subu`**

What If Overflow Detected?

- If "exception" (or "interrupt") occurs
  - Address of the instruction that overflowed is saved in a register
  - Computer jumps to predefined address to invoke appropriate routine for that exception
    - Like an unplanned hardware function call
- Operating System decides what to do
  - In some situations program continues after corrective code is executed
- MIPS hardware support: exception program counter (EPC) contains address of overflowing instruction --- (more in Chpt. 5)

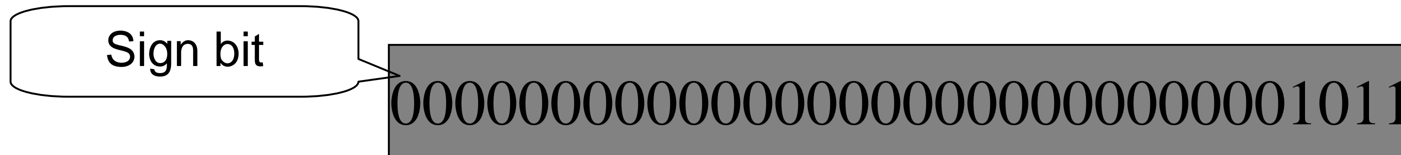
Unsigned Number of 32 bits long

- $00000000000000000000000000000000_2$=0
- $00000000000000000000000000000001_2$=1
- $00000000000000000000000000000010_2$=2
- …
- …
- …
- $11111111111111111111111111111110_2$=4,294,967,294
- $11111111111111111111111111111111_2$=4,294,967,295

$x31*2^{31}+ x30*2^{30}+ x29*2^{29}+\ldots+x1*2^1+x0*2^0$

x*i* means the ith bit of x

Signed Number



- O: mean positive
- 1: mean negative

Representing Negative Numbers

Two's Complement

- What is result for unsigned numbers if subtract larger number from a smaller one?
  - Would try to borrow from string of leading 0s, so result would have a string of leading 1s
  - With no obvious better alternative, pick representation that made the hardware simple:
    - leading 0s ⇒ positive,
    - leading 1s ⇒ negative

000000...*xxx* > 0

111111...*xxx* < 0

- This representation is called <u>two's complement</u>

Two's Complement (32-bit)

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

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

$0111 \ldots 1111\ 1111\ 1111\ 1101_{two} = 2,147,483,645_{ten}$

. . .

$0000 \ldots 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

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

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

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

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

$1111 \ldots 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$

. . .

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

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

Indicates sign of the integer

Two's Complement Formula, Example

- Recognizing role of sign bit, can represent positive and negative numbers in terms of the bit value times a power of 2:

$$- d_{31} \times -2^{31} + d_{30} \times 2^{30} + \cdots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$$

- Example (given 32-bit two's comp. number)

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$$

$$= 1 \times -2^{31} + 1 \times 2^{30} + 1 \times 2^{29} + \cdots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= -2^{31} + 2^{30} + 2^{29} + \cdots + 2^2 + 0 + 0$$

$$= -2{,}147{,}483{,}648_{10} + 2{,}147{,}483{,}644_{10}$$

$$= -4_{10}$$

Ways to Represent Signed Numbers

(1) Sign and magnitude

– separate sign bit   **0001001100101**   **1**

(2) Two's (2's) Complement (*n* bit positions)

– *n*-bit pattern $d_{n-1} \ldots d_2 d_1 d_0$ means:

$$-1 \times d_{n-1} \times 2^{n-1} + \cdots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$$

– also, unsigned sum of *n*-bit number and its negation = 2^n

```
  0001    positive one +

+ 1111   negative one (2's comp)
  ____
 10000          = 2^4 (=zero if only 4 bits)
```

Ways to Represent Signed Numbers

(3) One's (1's) Complement

– unsigned sum of n-bit number and its negation = $2^n - 1$

```
 0001    positive one
+1110    negative one (1's comp)
-----
 1111    (2^4 - 1)
```

– better than sign and magnitude but has two zeros (+0=0000 and -0=1111)

– some scientific computers use 1's comp.

(4) Biased notation

– add negative bias B to signed number; useful in floating point (for the exponent).

– $number = x - B$

Bit-Pattern, $b_3b_2b_1b_0$	Unsigned,	2's Comp,	1's Comp,	Biased
1111	**15**	**-1**	**0**	**7**
1110	**14**	**-2**	**-1**	**6**
1101	**13**	**-3**	**-2**	**5**
1100	**12**	**-4**	**-3**	**4**
1011	**11**	**-5**	**-4**	**3**
1010	**10**	**-6**	**-5**	**2**
1001	**9**	**-7**	**-6**	**1**
1000	**8**	**-8**	**-7**	**0**
0111	**7**	**7**	**7**	**-1**
0110	**6**	**6**	**6**	**-2**
0101	**5**	**5**	**5**	**-3**
0100	**4**	**4**	**4**	**-4**
0011	**3**	**3**	**3**	**-5**
0010	**2**	**2**	**2**	**-6**
0001	**1**	**1**	**1**	**-7**
0000	**0**	**0**	**0**	**-8**

Bias= 8
(Subtract 8)

Set on Less Than

- For signed integers
  - slt (set on less than)
  - slti (set on less than immediate)
- For unsigned integers
  - sltu (set on less than unsigned)
  - sltiu (set on less than immediate unsigned)

Signed *Vs*. Unsigned Comparisons

- Note: memory addresses naturally start at 0 and continue to the largest address – they are unsigned.
  - That is, negative addresses make no sense.
- C makes distinction in declaration.
  - integer (`int`) can be positive or negative.
  - unsigned integers (`unsigned int`) only positive.
- Thus MIPS needs two styles of comparison.
  - Set on less than (`slt`) and set on less than immediate (`slti`) work with signed integers.
  - Set on less than unsigned (`sltu`) and set on less than immediate unsigned (`sltiu`). (Will work with addresses).

Signed *Vs*. Unsigned Comparisons

- $s0 has

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

- $s1 has

$0011\ 1011\ 1001\ 1010\ 1000\ 1010\ 0000\ 0000_2$

- What are $t0, $t1 after:

```
slt     $t0, $s0, $s1     # signed compare
sltu    $t1, $s0, $s1     # unsigned compare
```

- `$t0`: $-4_{ten} < 1{,}000{,}000{,}000_{ten}$?
- `$t1`: $4{,}294{,}967{,}292_{ten} < 1{,}000{,}000{,}000_{ten}$?
  - ***Key Point: <u>Instructions</u> decide what binary bit-patterns mean***

Sign Extension

- Copy the sign repeatedly to fill the rest of the register for a signed load instruction.
- Ex:
  - lb (load byte)
    - Sign-extends to fill the 24 left-most bits of the register
  - lh (load half)
    - Sign-extends to fill the 16 left-most bits of the register
  - lbu (load byte unsigned) and lhu (load half-word unsigned)
    - For unsigned integers

Negation Shortcut for a Two's Complement Binary Number

- Invert every 0 to 1 and every 1 to 0, and then add one to the result.
- The sum of a number x and its inverted representation x' must be $111..111_2$,which is -1.
  - $x + x' \equiv -1$
    - $x + x' + 1 = 0$
    - $x + 1 \equiv -x'$

Two's Complement Shortcut: Negation

- Invert every 0 to 1 and every 1 to 0, then add 1 to the result
  - Unsigned sum of number and its inverted representation must be $111...111_2$
  - $111...111_2 = -1_{10}$
  - Let x' mean the *inverted representation* of x
  - Then $x + x' = -1 \Rightarrow x + x' + 1 = 0 \Rightarrow x' + 1 = -x$

- Example: -4 to +4 to -4

- x : $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$ -4
  x' : $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_2$
  +1: $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100_2$ 4
  $(x)'$: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_2$
  +1: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$ -4

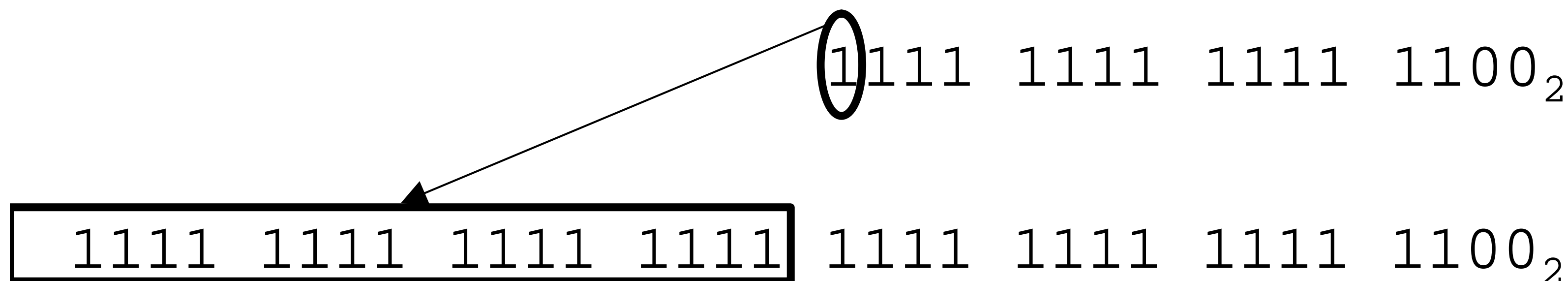
Sign Extension Shortcut for a Two's Complement Binary Number

- Convert a binary number represented in n bits to a number represented with more than n bits.
  - Ex: in order to add an immediate field contains a two's complement 16-bit number to a 32-bit register
  - For load, store, branch, add, and set on less than
- Take the most significant bit (i.e., the sign bit) from the smaller quantity and replicate it to fill the new bits of the larger quantity.

Two's Complement Shortcut

Using Sign extension

- Convert number represented in *k* bits to more than *k* bits
  - *e.g.*, 16-bit immediate field converted to 32 bits before adding to 32-bit register in addi
- Simply replicate the most significant bit (sign bit) of smaller quantity to fill new bits
  - 2's comp. positive number has infinite 0s to left
  - 2's comp. negative number has infinite 1s to left
  - Finite representation hides most leading bits; sign extension restores those that fit in the integer variable
  - 16-bit -4_{10} to 32-bit:



Bounds Check Shortcut for a Two's Complement Binary Number

- Reduce the cost of checking if $0 \leq x < y$, which matches the index out-of-bounds check for arrays.
- The key is that negative integers in two's complement format look like larger numbers in unsigned format.
  - Ex:

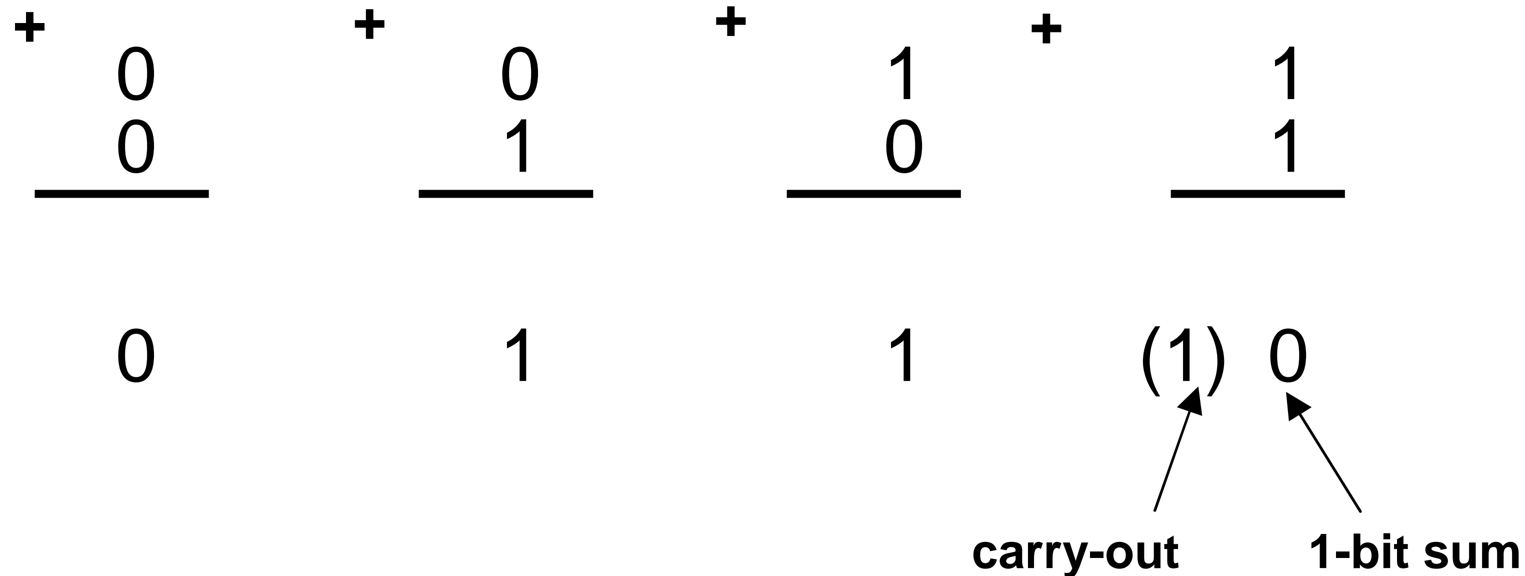
```
sltu $t0, $a1, $t2        //$a1: index, $t2: bound
beq $t0, $zero, IndexOutOfBounds
```

Outline

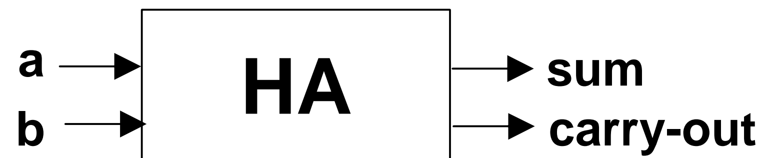
- Signed and unsigned numbers (Sec. 3.2)
- Addition and subtraction (Sec. 3.3)
- Multiplication (Sec. 3.4, CD: 3.23 In More Depth)
- Division (Sec. 3.5)
- Floating point (Sec. 3.6)

1-bit Binary Addition

- two 1-bit values gives four cases:



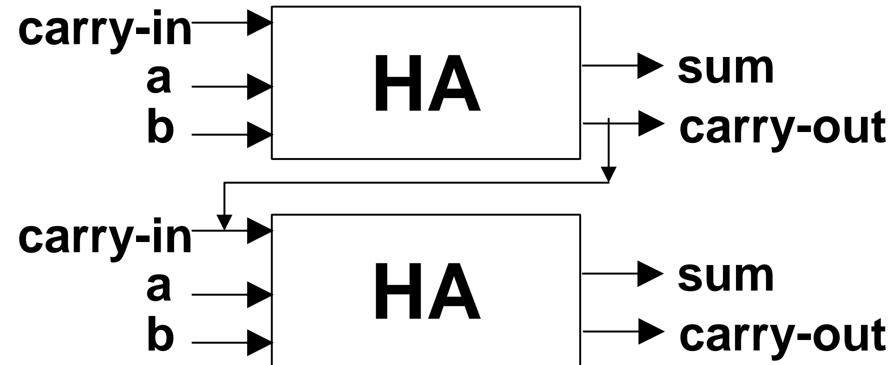
- digital logic?: half-adder circuit

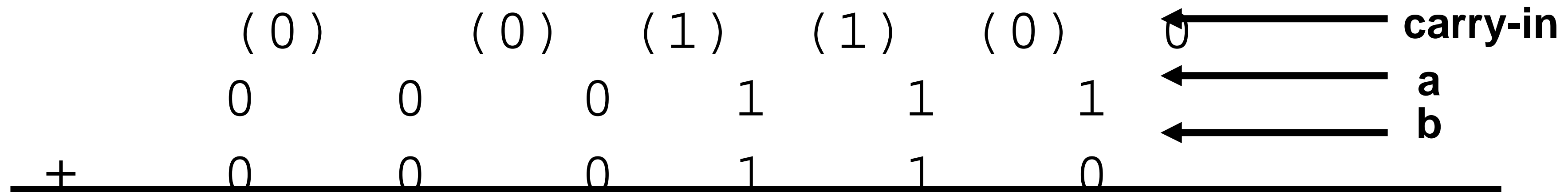


Multi-bit Addition (and Subtraction)

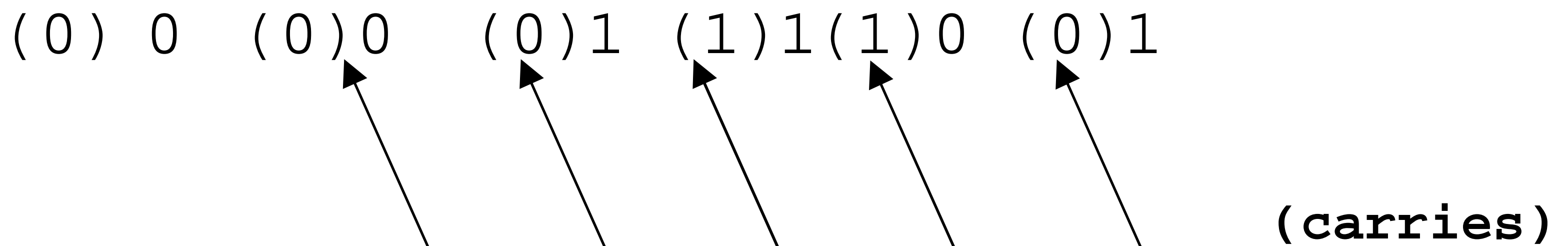
$00\ 0111 = 7_{10}$ +
$00\ 0110 = 6_{10}$

$00\ 1101 = 13_{10}$





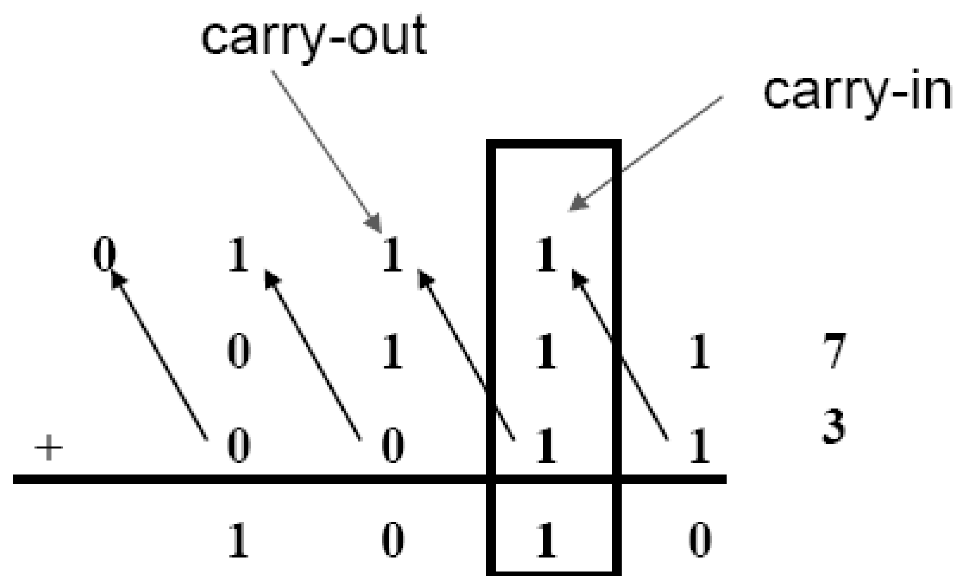
(0) 0 (0)0 (0)1 (1)1(1)0 (0)1



Subtract? Simply negate and add!

Integer Addition and Subtraction

- Two's complement addition/subtraction
  - start at the right
  - sum/subtract the bits and carry in
  - generate a carry out



Overflow

- Overflow occurs when the signed bit is incorrect.
  - 0 on the left of the bit pattern for a negative number
  - 1 on the left of the bit pattern for a positive number
- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction

Detecting Overflow in 2's Complement?

- Adding 2 31-bit positive 2's complement numbers can yield a result that needs 32 bits
  - sign bit set with value of result (1) instead of proper sign of result (0)
  - since need just 1 extra bit, only sign bit can be wrong

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

- **Adding operands with different signs, (subtracting with same signs) overflow cannot occur**

Overflow for Unsigned Numbers?

- Adding 2 32-bit unsigned integers could yield a result that needs 33 bits
  - can't detect from "sign" of result
- Unsigned integers are commonly used for address arithmetic, where overflows are ignored
- Hence, MIPS has unsigned arithmetic instructions, which ignore overflow:
  - addu, addiu, subu
  - Recall that in C, all overflows are ignored, so unsigned instructions are always used (different for Fortran, Ada)

Do It Yourself

- Add 4-bit signed (2's comp.) numbers :

```
   1111  -1₁₀.
 + 1110  -2₁₀
  11101.
```

- Did overflow occur?
  - overflow in 2's complement only if.

    Negative + Negative → "Positive."

    Positive + Positive → "Negative."
  - overflow = carry-out only if numbers considered to be unsigned.

- So: addition works same way for both unsigned, signed numbers.
- But overflow detection is different.

How to handle overflow?

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
- Don't always want to detect overflow
  - new MIPS instructions: addu, addiu, subu

Outline

- Signed and unsigned numbers (Sec. 3.2)
- Addition and subtraction (Sec. 3.3)
- Multiplication (Sec. 3.4 ,CD: 3.23 In More Depth)
- Division (Sec. 3.5)
- Floating point (Sec. 3.6)

MIPS R2000 Organization

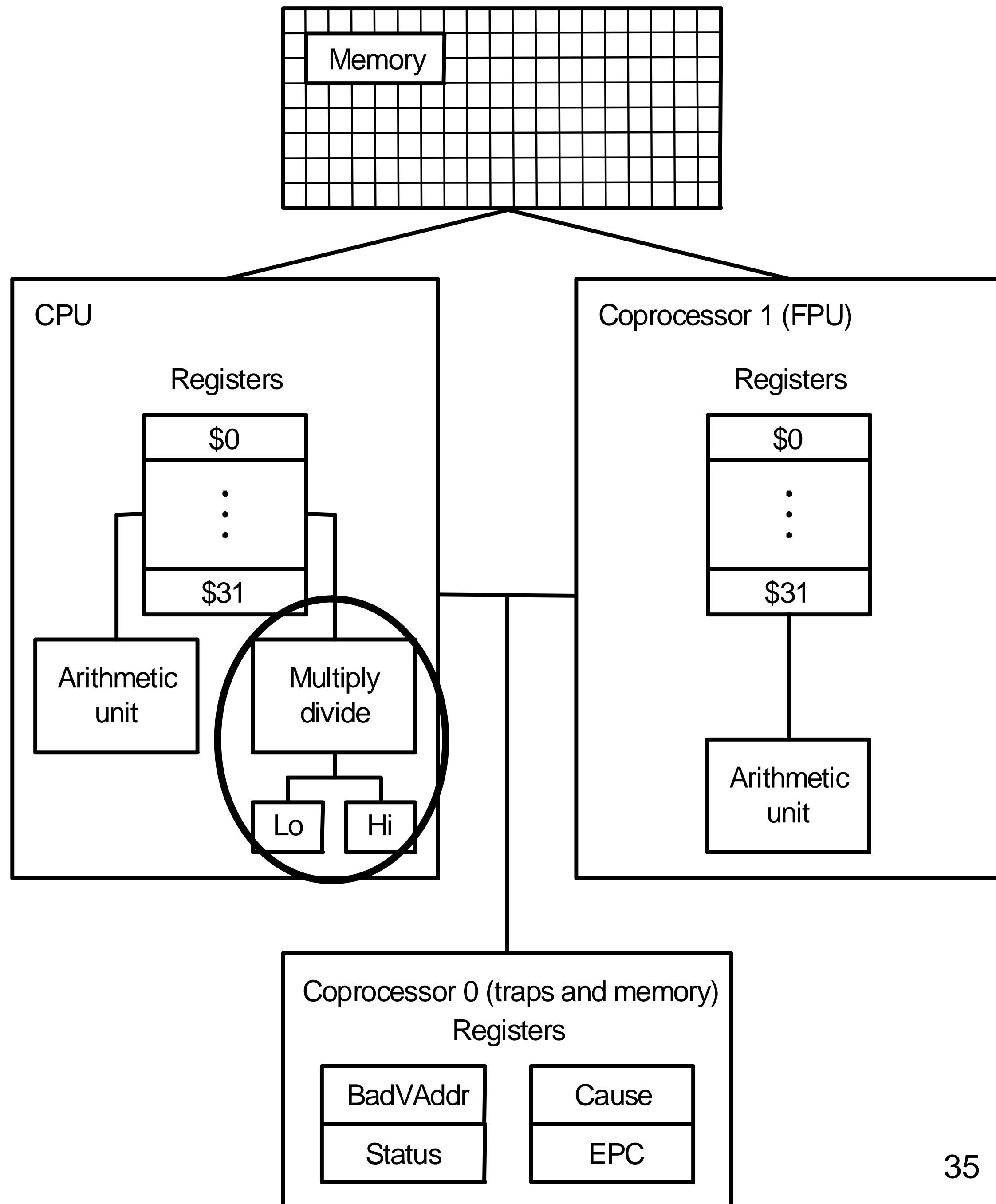


Fig. A.10.1

MULTIPLY

- Paper and pencil example (unsigned):

```
     1000 Multiplicand U
   * 1001 Multiplier   M
     1000
    0000
   0000
+ 1000
 01001000 Product
```

- Binary multiplication is easy:
  - $P_i == 0 \Rightarrow$ place all 0's (0 $*$ multiplicand)
  - $P_i == 1 \Rightarrow$ place a copy of U (1 $*$ multiplicand)
  - Shift the multiplicand left before adding to product
  - *Could we multiply via `add`, `sll`?*

MULTIPLY

- m bits * n bits = m+n bit product

Multiply by Power of 2 via Shift Left

- Number representation: $B = b_{31}b_{30} \cdots b_2b_1b_0$

$$B = b_{31}*2^{31} + b_{30}*2^{30} + \times\times\times + b_2*2^2 + b_1*2^1 + b_0*2^0$$

- What if multiply *B by* 2?

$$B*2 = b_{31}*2^{31+1} + b_{30}*2^{30+1} + \times\times\times + b_2*2^{2+1} + b_1*2^{1+1} + b_0*2$$

$$= b_{31}*2^{32} + b_{30}*2^{31} + \times\times\times + b_2*2^3 + b_1*2^2 + b_0*2^1$$

- What if shift *B left* by 1?

$$B << 1 = b_{31}*2^{32} + b_{30}*2^{31} + \times\times\times + b_2*2^3 + b_1*2^2 + b_0*2^1$$

- Multiply by 2^i often replaced by shift left i

Multiply in MIPS

- Can multiply variable by any constant using MIPS `sll` and `add` instructions:

M I P S

```
i = i * 10; /* assume i: $s0 */

sll $t0, $s0, 3        # i * 2^3
add $t1, $zero, $t0
sll $t0, $s0, 1        # i * 2^1
add $s0, $t1, $t0
```

- MIPS multiply instructions: `mult`, `multu`
- `mult $t0, $t1`
  - puts 64-bit product in pair of new registers `hi`, `lo`; copy to $n by `mfhi`, `mflo`
  - 32-bit integer result in register `lo`

Unsigned Multiply

- Paper and pencil example (unsigned):

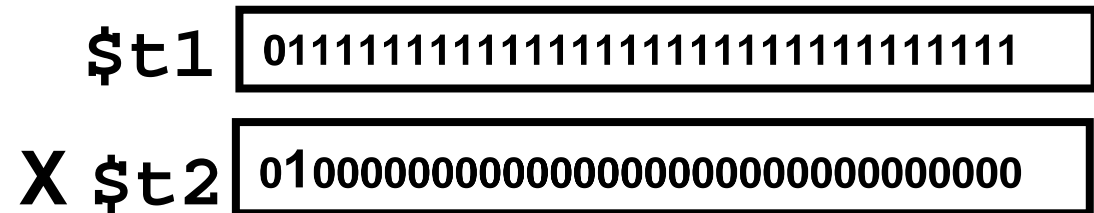
```
Multiplicand              1000ten
Multiplier           X    1001ten
                          1000
                         0000
                        0000
                       1000
Product                01001000ten
```

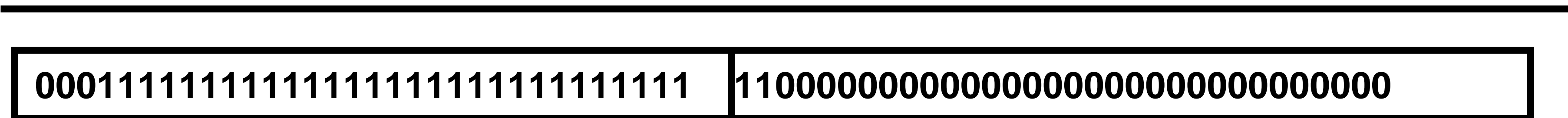
- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)
- 2 versions of multiply hardware and algorithm

Multiplication in MIPS

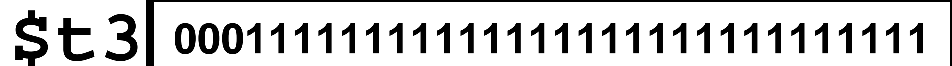
```
mult $t1, $t2          # t1 * t2
```

- No destination register: product could be ~2^{64}; need two special registers to hold it
- 3-step process:





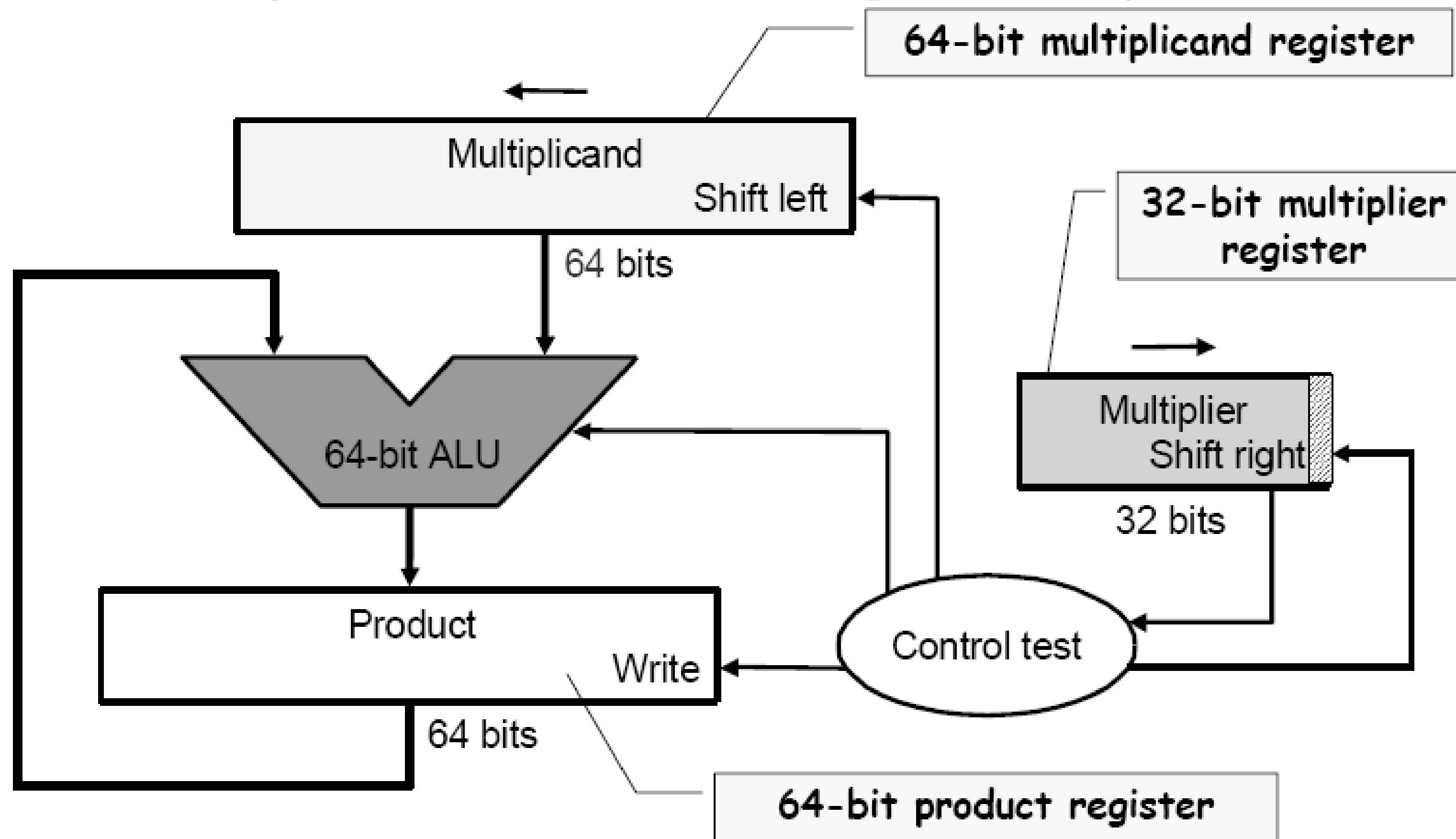

```
mfhi $t3
```



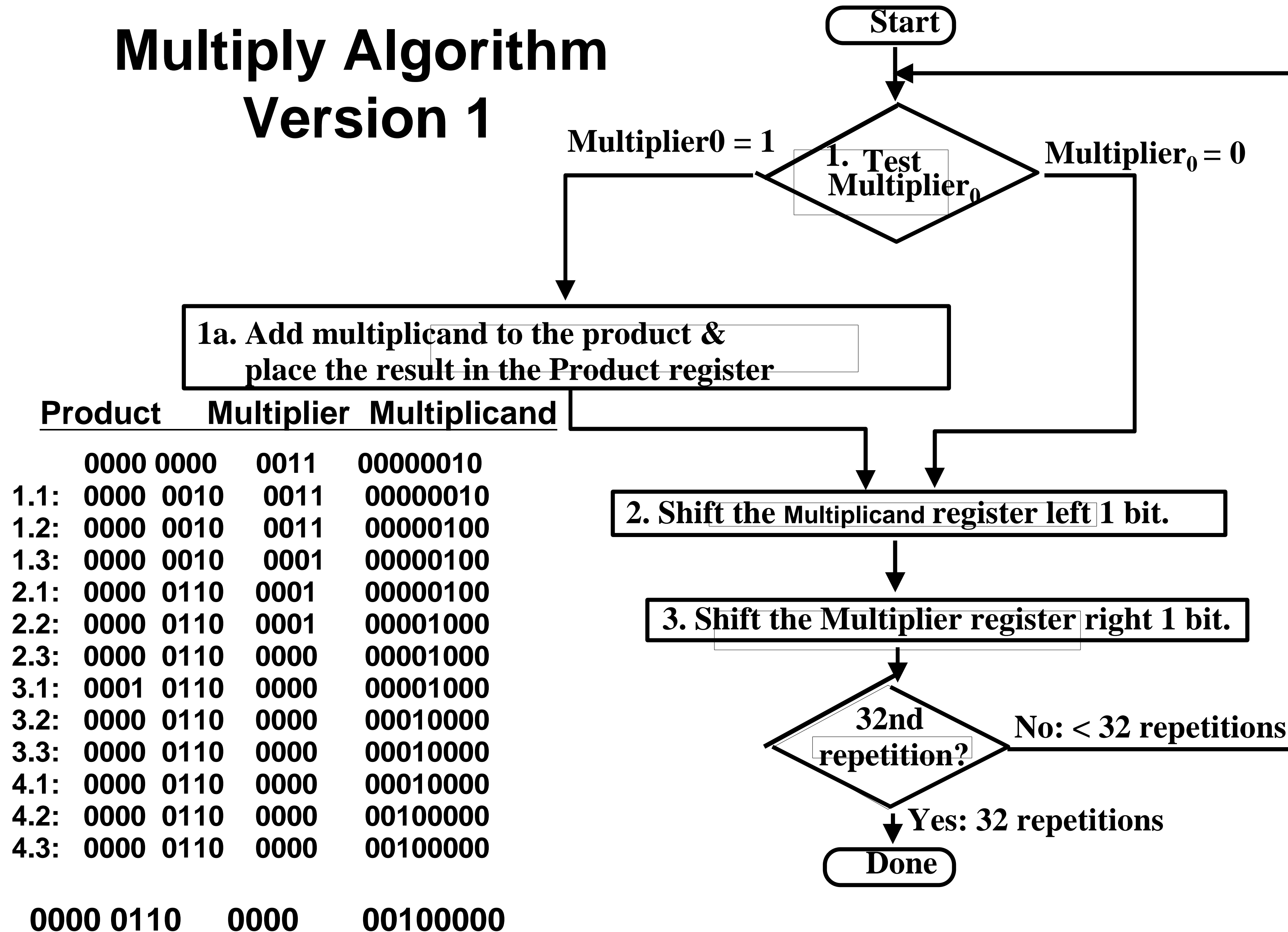

```
mflo $t4
```

Unisigned Multiplier (Ver. 1)

- 64-bit *multiplicand register* (with 32-bit multiplicand at right half), 64-bit ALU, 64-bit *product register*, 32-bit *multiplier register*



Multiply Algorithm Version 1



	Product	Multiplier	Multiplicand
	0000 0000	0011	00000010
1.1:	0000 0010	0011	00000010
1.2:	0000 0010	0011	00000100
1.3:	0000 0010	0001	00000100
2.1:	0000 0110	0001	00000100
2.2:	0000 0110	0001	00001000
2.3:	0000 0110	0000	00001000
3.1:	0001 0110	0000	00001000
3.2:	0000 0110	0000	00010000
3.3:	0000 0110	0000	00010000
4.1:	0000 0110	0000	00010000
4.2:	0000 0110	0000	00100000
4.3:	0000 0110	0000	00100000

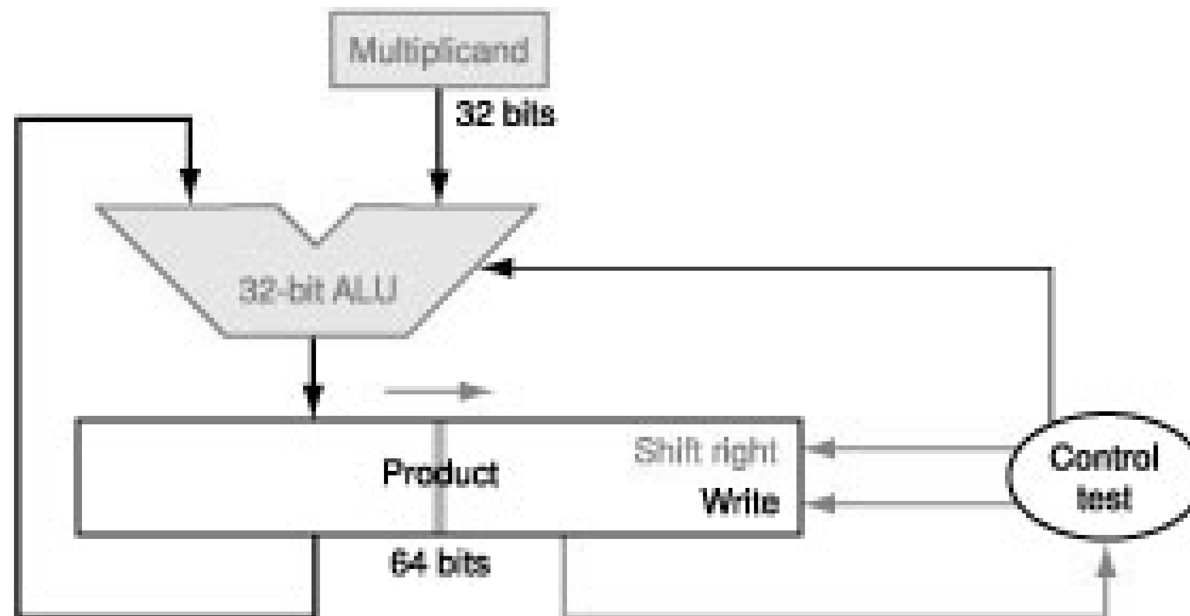
0000 0110 0000 00100000

Observations: Multiply Ver. 1

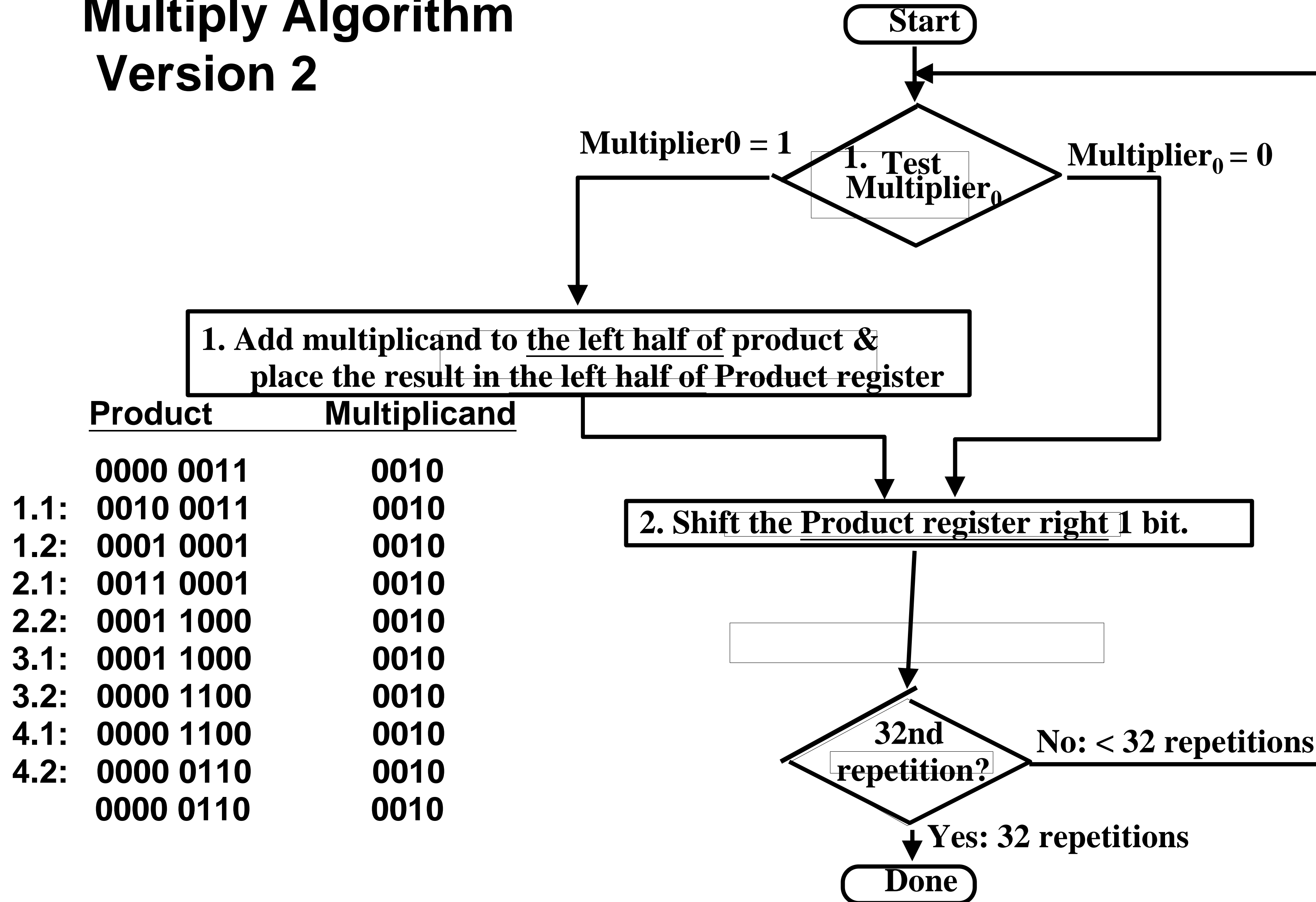
- 1 clock per cycle => ~100 clocks per multiply
  - Ratio of multiply to add 5:1 to 100:1
- Half of the bits in multiplicand always 0
  => 64-bit adder is wasted
- 0's inserted in right of multiplicand as shifted
  => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?
- Product register wastes space => combine Multiplier and Product register

Unsigned Multiplier (Ver. 2)

- 32-bit Multiplicand register, 32 -bit ALU, 64-bit Product register (HI & LO in MIPS), (0-bit Multiplier register)



Multiply Algorithm Version 2



	Product	Multiplicand
	0000 0011	0010
1.1:	0010 0011	0010
1.2:	0001 0001	0010
2.1:	0011 0001	0010
2.2:	0001 1000	0010
3.1:	0001 1000	0010
3.2:	0000 1100	0010
4.1:	0000 1100	0010
4.2:	0000 0110	0010
	0000 0110	0010

Observations: Multiply Ver. 2

- 2 steps per bit because multiplier and product registers combined
- MIPS registers Hi and Lo are left and right half of Product register
  => this gives the MIPS instruction Multu
- What about signed multiplication?
  1. The easiest solution is to make both positive and remember whether to complement product when done (leave out sign bit, run for 31 steps)
  2. Apply definition of 2's complement
     - sign-extend partial products and subtract at end
  3. Booth's Algorithm is an elegant way to multiply signed numbers using same hardware as before and save cycles

Signed Multiply

- Paper and pencil example (signed):

```
Multiplicand              1001 (-7)
Multiplier             X  1001 (-7)
                       --------
                       11111001
               +       0000000
               +      000000
               -      11001
               ----------------
Product               00110001 (49)
```

- Rule 1: Multiplicand sign extended
- Rule 2: Sign bit (s) of Multiplier
  - 0 => 0 x multiplicand
  - 1 => -1 x multiplicand
- Why rule 2 ?
  - $X = s\ x_{n-2}\ x_{n-3} \ldots\ x_1\ x_0$ (2's complement)
  - $\text{Value}(X) = -1 \times s \times 2^{n-1} + x_{n-2} \times 2^{n-2} + \ldots\ldots + x_0 \times 2^0$

Booth's Algorithm

- Booth's Algorithm is a pretty <u>fast</u> <u>algorithm</u> for multipling <u>signed</u> <u>binary</u> numbers.
- Q and Q_{-1}:

Q	Q_{-1}	
0	0	right-<u>shift</u>
0	1	<u>add</u> M to A and right-shift
1	0	<u>substract</u> M from A and right shift
1	1	right-shift

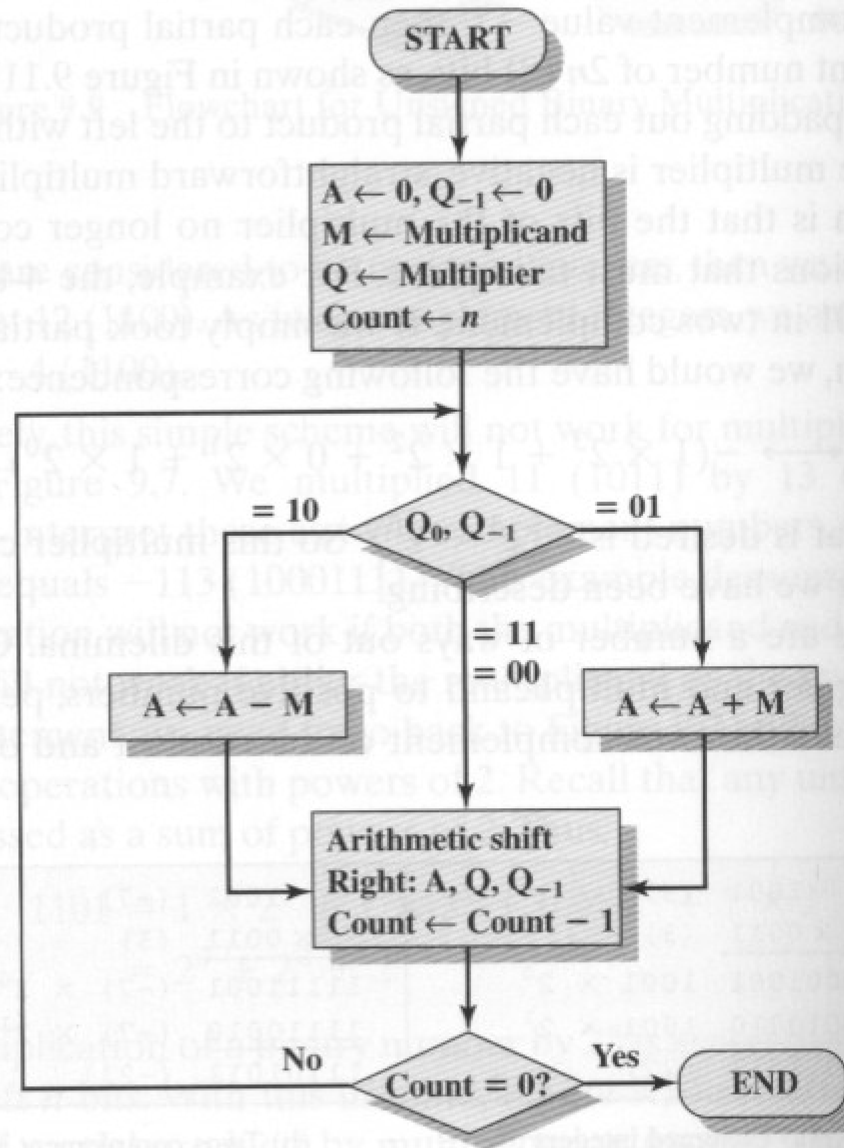


Figure 9.12 Booth's Algorithm for Twos Complement Multiplication

Example 1 for Booth's Algorithm

- 7 * 3 = 0111 * 0011
  M = 0111
  A = 0000
  Q = 0011
  Q_{-1} = 0

-

	M	A	Q	Q_{-1}	Operation #
0	0111	0000	0011	0	
1	0111	1001	0011	0	substract
	0111	1100	1001	1	right shift
2	0111	1110	0100	1	right shift
3	0111	0101	0100	1	add
	0111	0010	1010	0	right shift
4	0110	0001	0101	0	right shift

Example 2 for Booth's Algorithm

- 2 * -3 = 0010 * 1101
  M = 0010
  A = 0000
  Q = 1101
  Q_{-1} = 0

-

	M	A	Q	Q_{-1}	Operation #
0	0010	0000	1101	0	
1	0010	1110	1101	0	subtract
	0010	1111	0110	1	right shift
2	0010	0001	0110	1	add
	0010	0000	1011	0	right shift
3	0010	1110	1011	0	subtract
	0010	1111	0101	1	right shift
4	0010	1111	0101	1	no operation
	0010	1111	1010	1	right shift

Faster Multiplier

- A combinational multiplier
- Carry save adder to construct Wallace tree multiplier

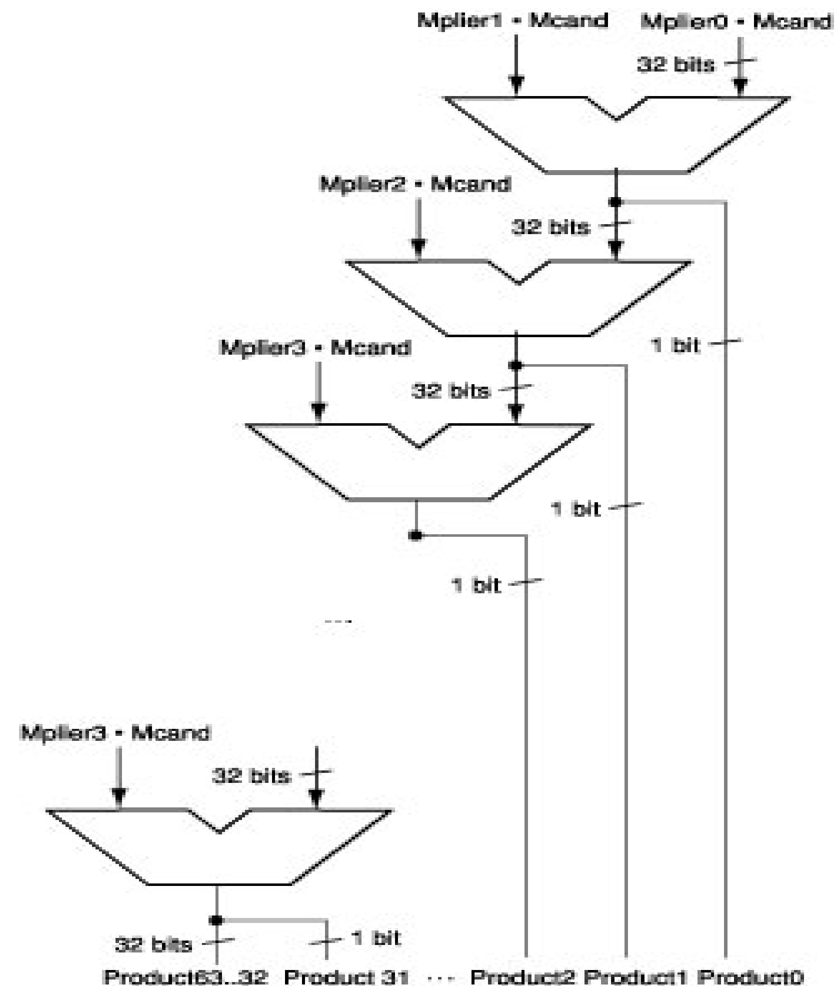


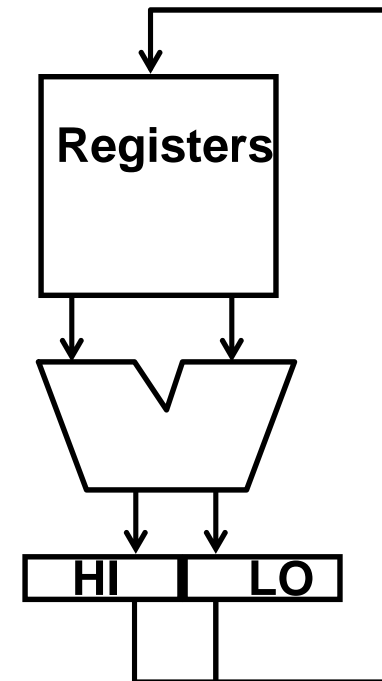
Fig. 3.9

Outline

- Signed and unsigned numbers (Sec. 3.2)
- Addition and subtraction (Sec. 3.3)
- Multiplication (Sec. 3.4)
- Division (Sec. 3.5)
- Floating point (Sec. 3.6)

MIPS Multiply/Divide Summary

- Start multiply, divide
  - MULT rs, rt — HI-LO = rs × rt // 64-bit signed
  - MULTU rs, rt — HI-LO = rs × st // 64-bit unsigned
  - DIV rs, rt — LO = rs ÷ rt; HI = rs mod rt
  - DIVU rs, rt
- Move result from multiply, divide
  - MFHI rd — rd = HI
  - MFLO rd — rd = LO
- Move to HI or LO
  - MTHI rd — HI = rd
  - MTLO rd — LO = rd



Divide: Paper & Pencil

```
                1001   Quotient
              ________
Divisor 1000 |1001010  Dividend
             –1000
                1010
               –1000
                  10 Remainder (or Modulo result)
```

- See how big a number can be subtracted, creating quotient bit on each step

Binary => 1 * divisor or 0 * divisor

Dividend = Quotient × Divisor + Remainder

Division in MIPS

```
div $t1, $t2      # t1 / t2
```

- Quotient stored in Lo, remainder in Hi

```
mflo $t3      #copy quotient to t3
mfhi $t4      #copy remainder to t4
```

- 3-step process

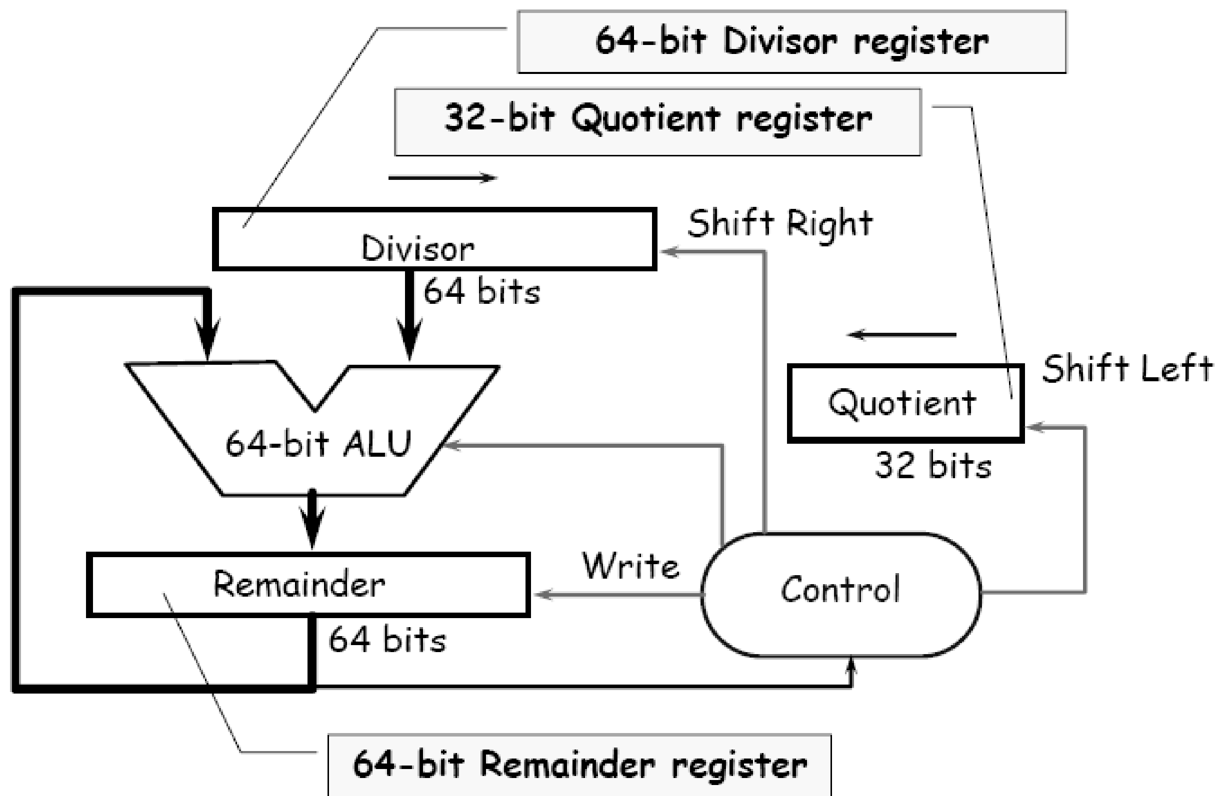
- Unsigned multiplication and division:

```
multu $t1, $t2      # t1 * t2
divu  $t1, $t2      # t1 / t2
```

  - Just like mult, div, except now interpret t1, t2 as unsigned integers instead of signed
  - Answers are also unsigned, use mfhi, mflo to access

Divide Hardware (Version 1)

- 64-bit *Divisor register* (initialized with 32-bit divisor in left half),
- 64-bit ALU,
- 64-bit *Remainder register* (initialized with 64-bit dividend),
- 32-bit *Quotient register*



Divide Algorithm (Version 1)

111 ÷ 0010

Quot.	Divisor	Rem.
1.0000	00100000	00000111
		11100111
	00010000	00000111
2.0000	00010000	00000111
		11110111
		00000111
3.0000	00001000	00000111
		11111111
		00000111
4.0000	00000100	00000111
		00000011
0001		00000011
0001	00000010	00000011
5.		00000001
0011		00000001
0011	00000001	00000001

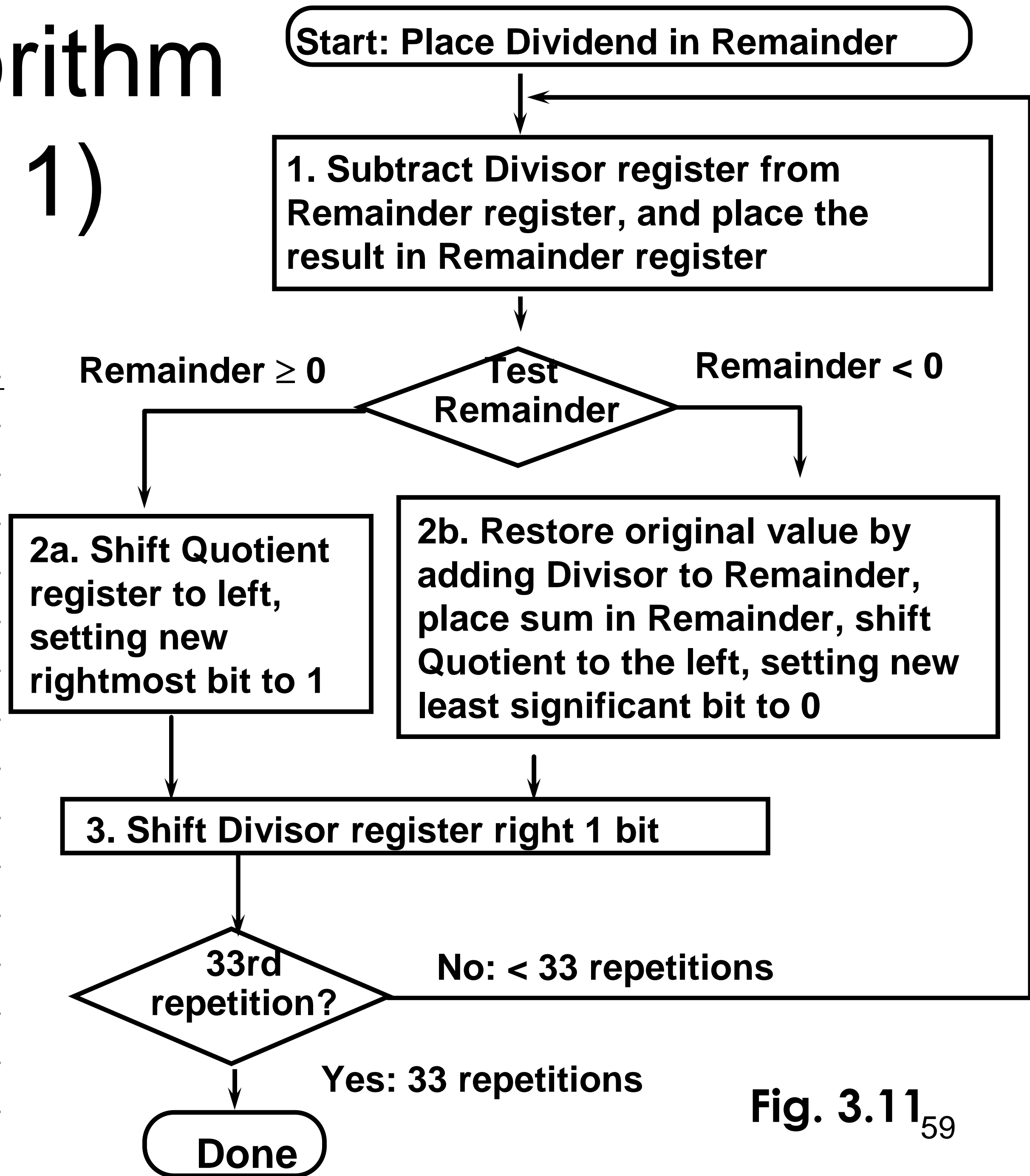


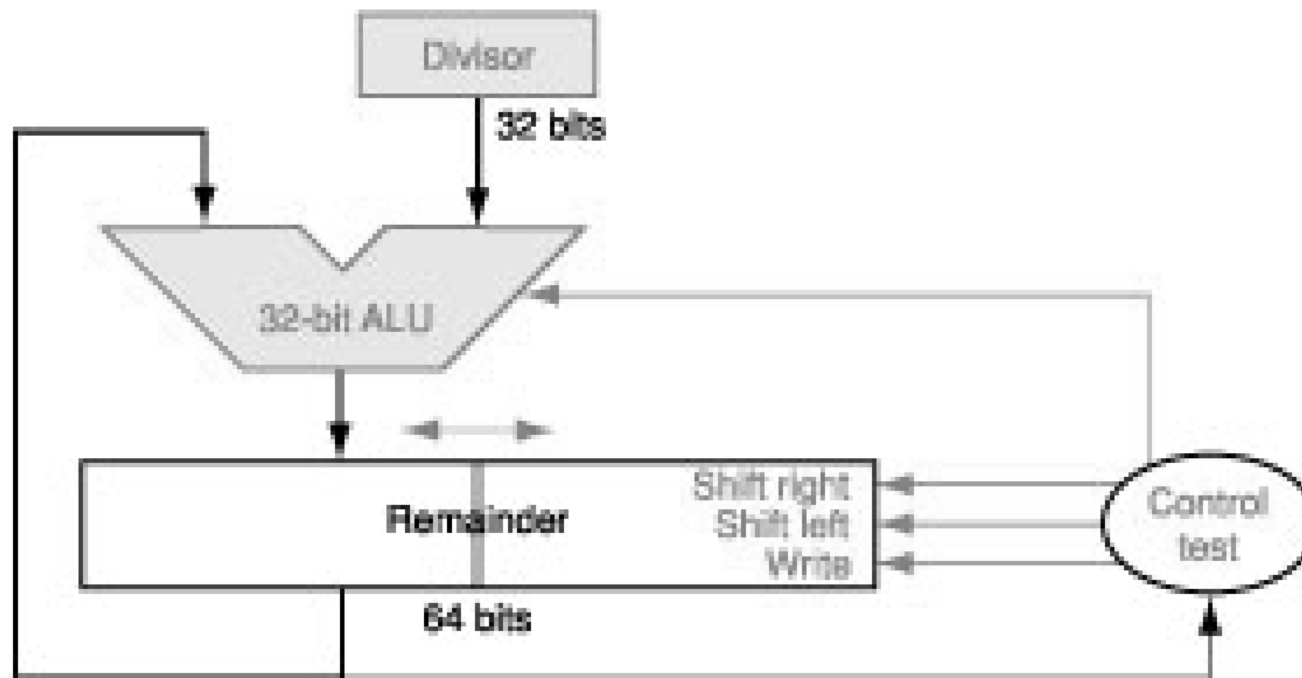
Fig. 3.11

Observations: Divide Version 1

- Half of the bits in divisor register always 0
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted
- Instead of shifting divisor to right,
  shift remainder to left?
- 1st step cannot produce a 1 in quotient bit
  (otherwise quotient is too big for the register)
  => switch order to shift first and then subtract
  => save 1 iteration
- Eliminate Quotient register by combining with Remainder register as shifted left

Divide Hardware (Version 2)

- 32-bit Divisor register,
- 32 -bit ALU,
- 64-bit Remainder register, (<u>0</u>-bit Quotient register)

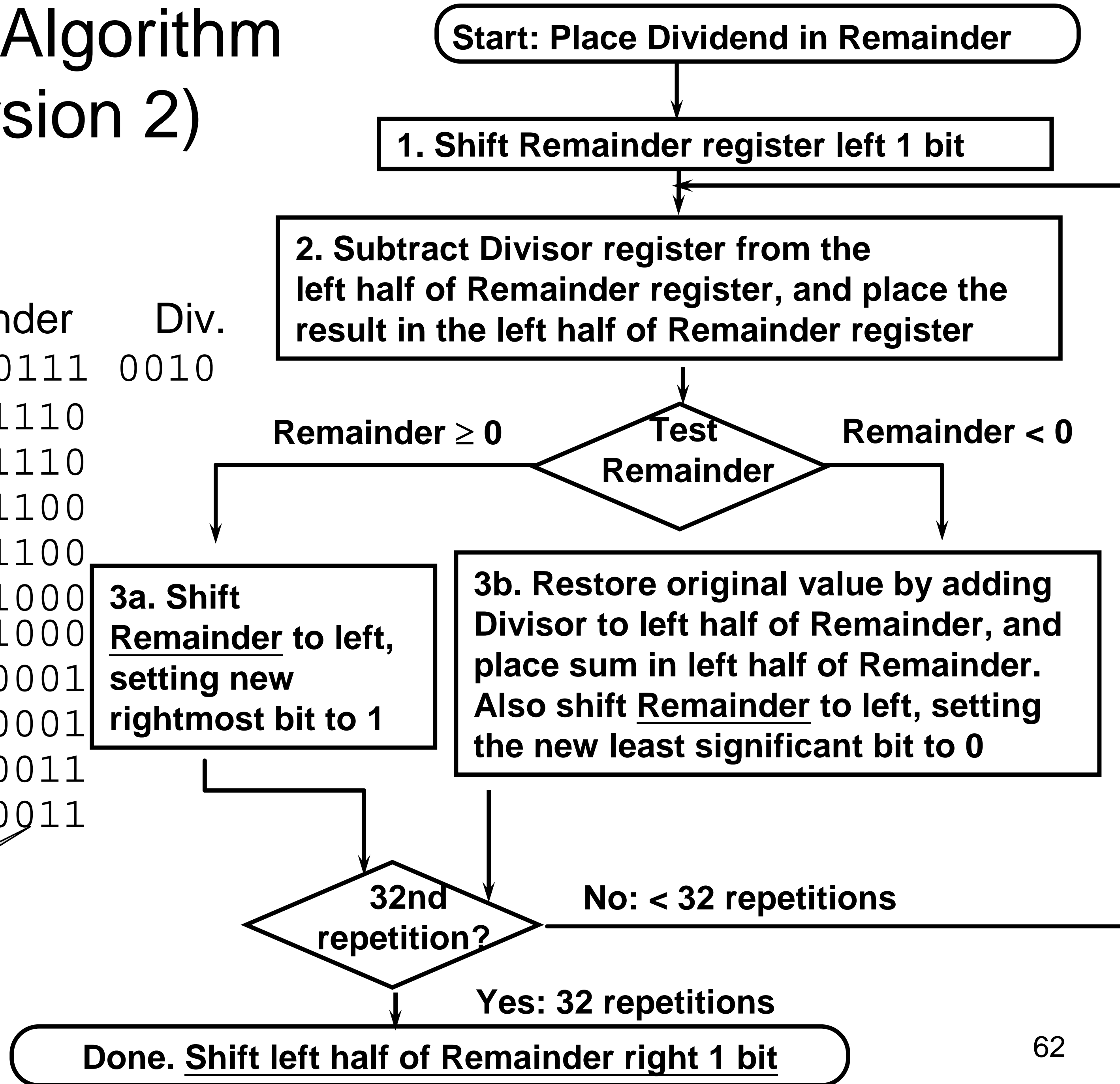


Divide Algorithm (Version 2)

111 ÷ 0010

Step	Remainder	Div.
0	0000 0111	0010
1.1	0000 1110	
1.2	1110 1110	
1.3b	0001 1100	
2.2	1111 1100	
2.3b	0011 1000	
3.2	0001 1000	
3.3a	0011 0001	
4.2	0001 0001	
4.3a	0010 0011	
	0001 0011	

Quotient



Divide

- Signed Divides:
  - Remember signs, make positive, complement quotient and remainder if necessary
  - Let Dividend and Remainder have same sign and negate Quotient if Divisor sign & Dividend sign disagree,
  - e.g., -7÷ 2 = -3, remainder = -1

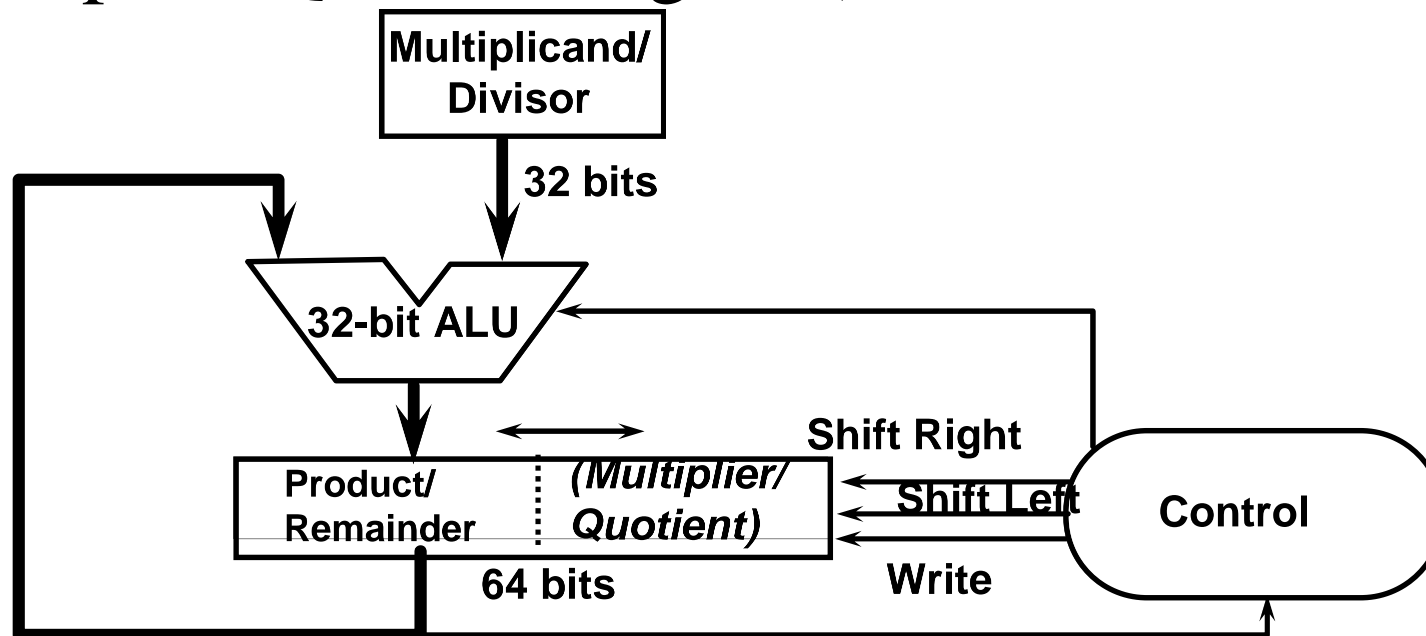
    -7÷- 2 = 3, remainder = -1
  - Satisfy Dividend =Quotient x Divisor + Remainder

Observations: Multiply and Divide

- Same hardware as multiply: just need ALU to add or subtract, and 64-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide

Multiply/Divide Hardware

- 32-bit Multiplicand/Divisor register,
- 32 -bit ALU,
- 64-bit Product/Remainder register, (0-bit Multiplier/Quotient register)



Is Shift Right Arith. Divide by 2?

- Shifting right by n bits would seem to be the same as dividing by 2^n
- Problem is signed integers
  - Zero fill (`srl`) is wrong for negative numbers
- Shift Right Arithmetic (`sra`); sign extends (replicates sign bit); but does it work?
- Divide -5 by 4 via `sra 2`; result should be -1

  1111 1111 1111 1111 1111 1111 1111 1011

  1111 1111 1111 1111 1111 1111 1111 1110
- = -2, not -1; Off by 1, so doesn't work
- Is it always off by 1??

Outline

- Signed and unsigned numbers (Sec. 3.2)
- Addition and subtraction (Sec. 3.3)
- Multiplication (Sec. 3.4)
- Division (Sec. 3.5)
- Floating point (Sec. 3.6)

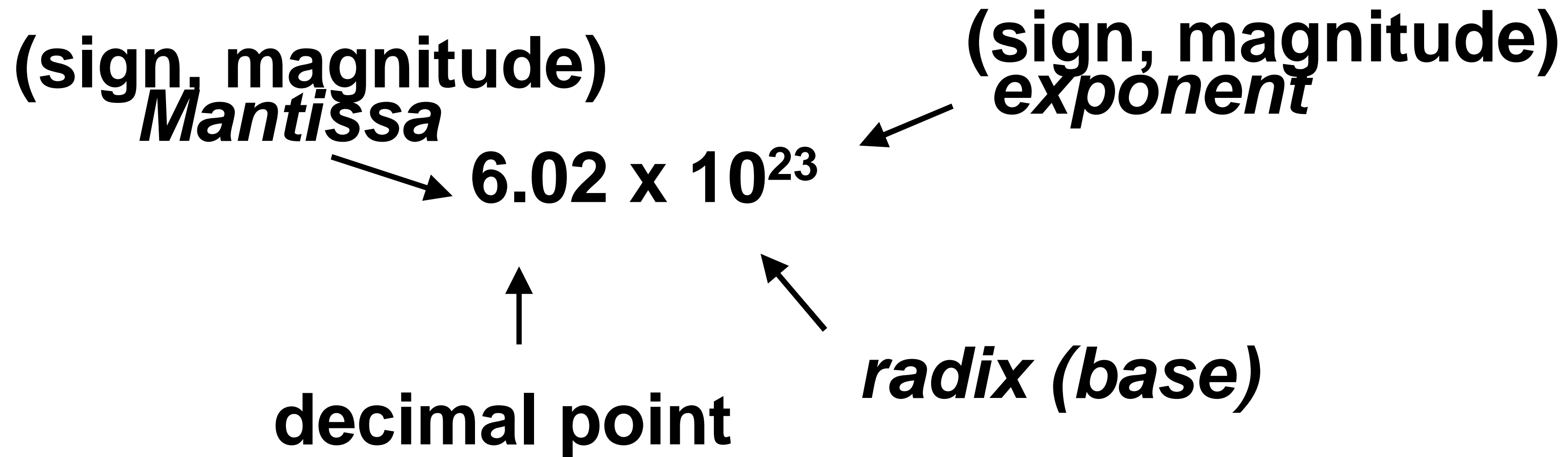
Floating-Point: Motivation

- What can be represented in *n* bits?

Unsigned	0	to	$2^n - 1$
2's Complement	-2^{n-1}	to	$2^{n-1} - 1$
1's Complement	$-2^{n-1}+1$	to	$2^{n-1} - 1$
Bias M	$-M$	to	$2^n - M - 1$

- But, what about ...
  - very large numbers? 9,349,398,989,787,762,244,859,087,678
  - very small number? 0.0000000000000000000000045691
  - rationals 2/3
  - irrationals $\sqrt{2}$
  - transcendentals e, π

Recall Scientific Notation



- Normal form:

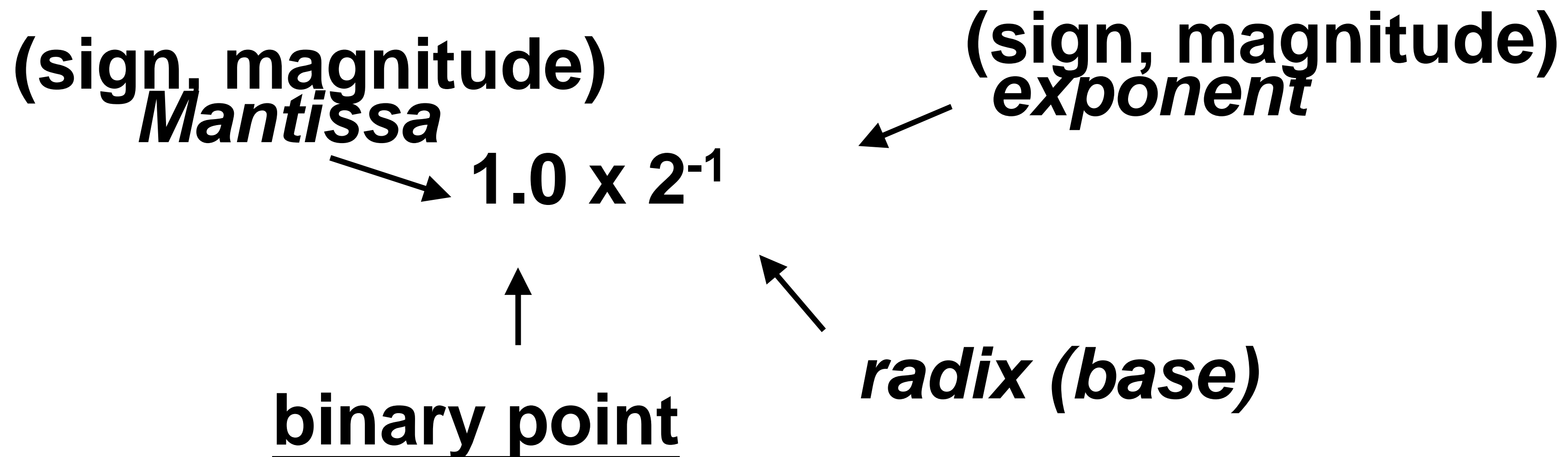
no leading 0s (digit 1 to left of decimal point)

- Alternatives to representing 1/1,000,000,000

  Normalized: 1.0×10^{-9}

  Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

Scientific Notation for Binary Numbers



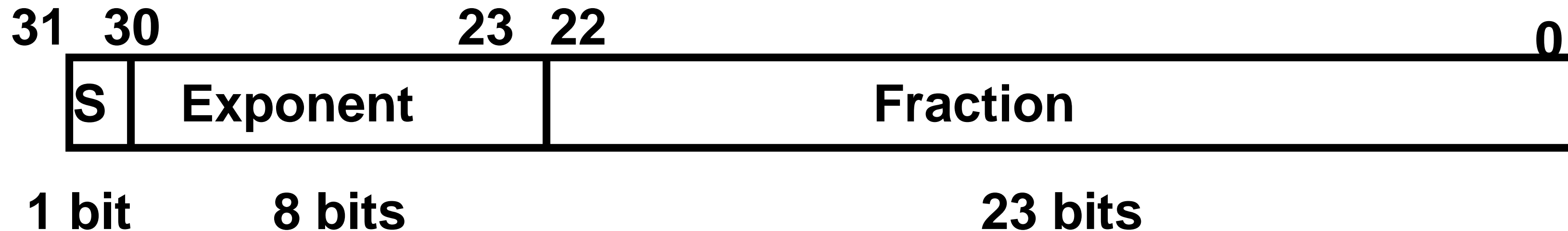
- Computer arithmetic that supports it called floating point, because it represents numbers where binary point is not fixed, as it is for integers
- Declare such a variable in C as `float (double, long double)`
- Normalized form: $1.xxxxxxxxxx_2 * 2^{yyyy_2}$

  Simplifies data exchange, increases accuracy

  4_{10} == 1.0×2^2, 8_{10} == 1.0×2^3

Single Precision FP Representation

- Normal format: $1.xxxxxxxxxx_{two} \times 2^{yyyy_{two}}$
- Want to put it into multiple words: 32 bits for *single-precision* and 64 bits for *double-precision*
- A simple single-precision representation:



- **Meaning: $(-1)^S$ x F x 2^E**
- **Can now represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}**
- **Relationship between Mantissa and Significand bits? Between E and Exponent?**
- **In C type float**

Floating Point Number Representation

- What if result too large? (> 2.0×10^{38})

  Overflow!

  – A positive exponent becomes too large to fit in the 8-bit exponent field

- What if result too small? (>0, < 2.0×10^{-38})

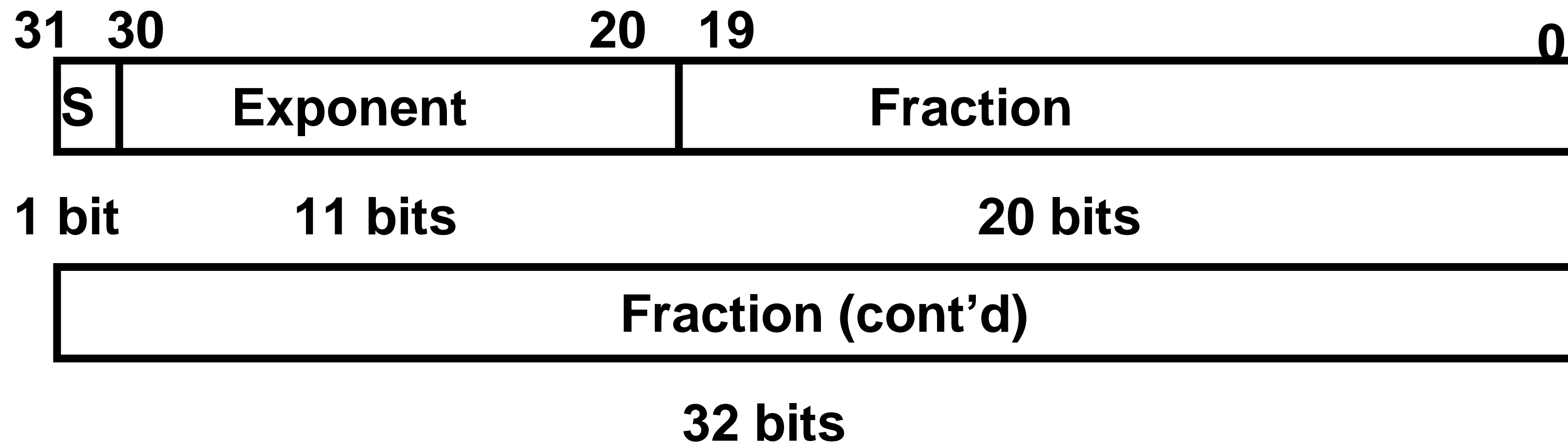
  Underflow!

  – A negative exponent becomes too large to fit in the exponent field

- How to reduce chances of overflow or underflow?

Double Precision FP Representation

- Next multiple of word size (64 bits)



- Double precision (vs. single precision)
  - **C variable declared as double**
  - **Represent numbers almost as small as 2.0 x 10^{-308} to almost as large as 2.0 x 10^{308}**
  - **But primary advantage greater accuracy due to larger fraction**
  - **There is also long double version (16 bytes)**

IEEE 754 Standard - Significand

- Significand:
  - To pack more bits, leading 1 implicit for normalized numbers
  - 1 + 23 bits single, 1 + 52 bits double
  - always true:

    $(-1)^S \times (1 + \text{Significand}) \times 2^{\text{Exponent}}$,

    where $0 < \text{Significand} < 1$

    (for normalized numbers)
  - Note
    - The term significant represents the 24- or 53-bit number that is 1 pus the fraction
    - The term fraction represents the 23- or 52-bit number.

IEEE 754 Standard

- To pack more bits, make leading 1 of mantissa implicit for normalized numbers

  1 + 23 bits single, 1 + 52 bits double

  0 has no leading 1, so reserve exponent value 0 just for number 0.0

  Meaning: (almost correct)

  $(-1)^{S} \times (1 + \text{Significand}) \times 2^{\text{Exponent}}$,

  where $0 < \text{Significand} < 1$

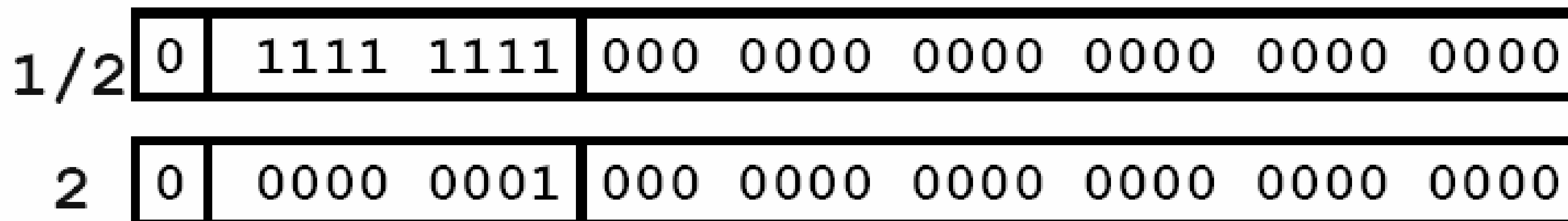
- If label significand bits left-to-right as s_1, s_2, s_3, ... then value is:

  $(-1)^{S} \times (1+(s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_2 \times 2^{-3}) + \cdots) \times 2^{\text{Exponent}}$

IEEE 754 Standard- Exponent

- Exponent:
  - Need to represent positive and negative exponents
  - Also want to compare FP numbers as if they were integers, to help in value comparisons
  - If use 2's complement to represent?

    e.g., 1.0 x 2^{-1} versus 1.0 x2^{+1} (1/2 versus 2)



IEEE 754 Standard- Exponent

- Instead, let notation 0000 0000 be most negative, and 1111 1111 most positive
- Called biased notation, where bias is the number subtracted to get the real number
  - IEEE 754 uses bias of 127 for single precision: Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

IEEE 754 Standard

- Summary (single precision):

31	30 ... 23	22 ... 0
S	Exponent	Significand
1 bit	8 bits	23 bits

$(-1)^S \times (1.\text{Significand}) \times 2^{(\text{Exponent}-127)}$

- Double precision identical, except with exponent bias of 1023

Biased (Excess) Notation

- Biased 7

0000	-7
0001	-6
0010	-5
0011	-4
0100	-3
0101	-2
0110	-1
0111	0
1000	1
1001	2
1010	3
1011	4
1100	5
1101	6
1110	7
1111	8

Example: Converting Decimal to FP

- Show MIPS representation of -0.75

  (show exponent in decimal to simplify)

  $-0.75 = -3/4 = -3/2^2$

  $-11_{two}/2^2 = -11_{two} \times 2^{-2} = -0.11_{two} \times 2^0$

  Normalized to $-1.1_{two} \times 2^{-1}$

  $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

  $(-1)^1 \times (1 + .100\ 0000\ ...\ 0000) \times 2^{(126-127)}$

1	0111 1110	100 0000 0000 0000 0000 0000

$S = 1$; Exponent = 126; Significand = $100\ ...\ 000_2$

Example: Converting FP to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010

- Sign $S = 0 \Rightarrow$ positive

- Exponent E :

  $0110\ 1000_{two} = 104_{ten}$

  Bias adjustment: 104 - 127 = -13

- Mantissa:

  $1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$

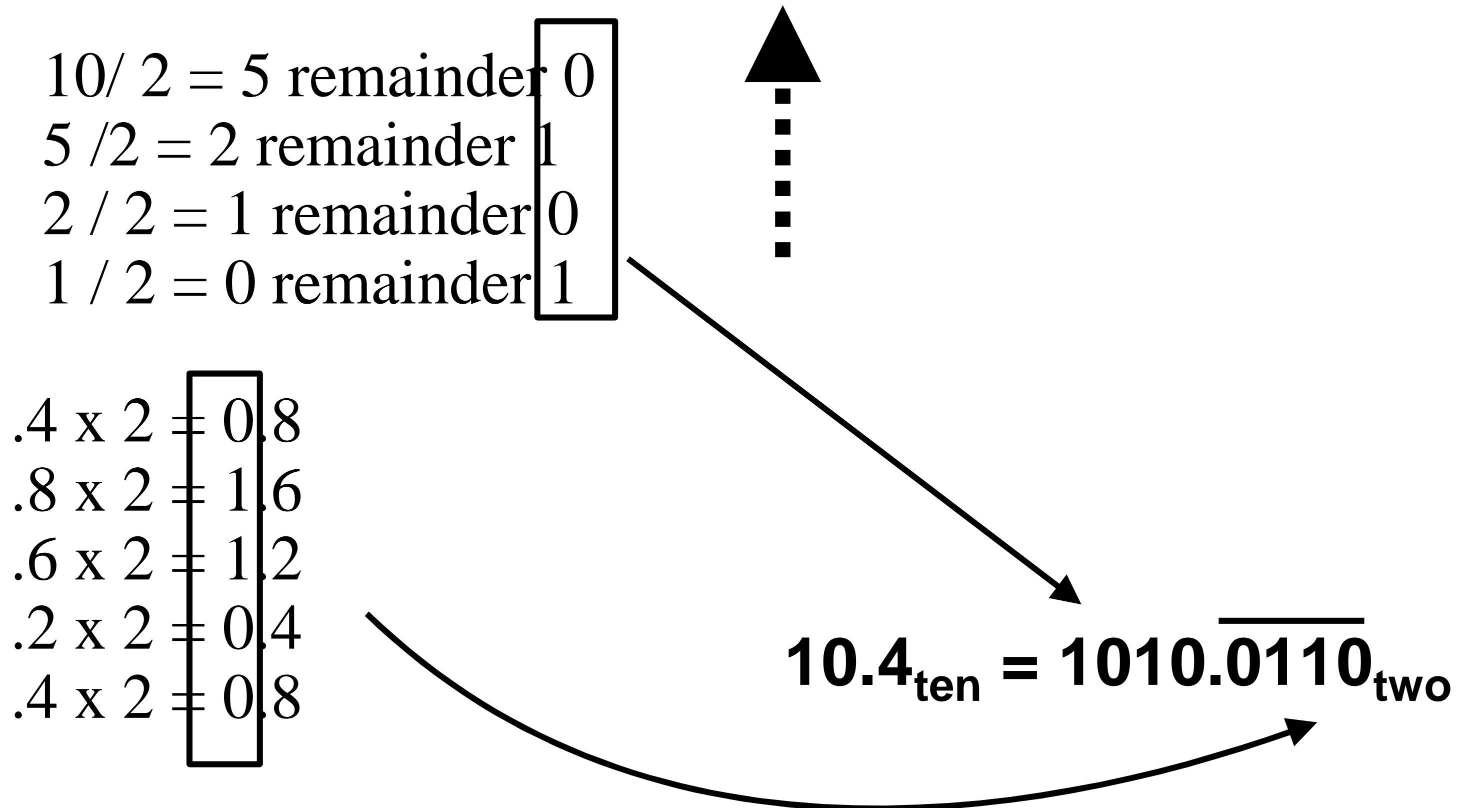
  $= 1+ (5{,}587{,}778 / 2^{23})$

  $= 1+ (5{,}587{,}778 / 8{,}388{,}608) = 1.0 + 0.666115$

- Represents: $1.666115_{ten} \times 2^{-13} \sim 2.034 \times 10^{-4}$

How To Convert Decimal to Binary

- How convert 10.4_{ten} to binary?
- Deal with fraction & whole parts separately:



Zero and Special Numbers

- What have we defined so far? (single precision)

Exponent	Significand	Object
0	0	???
0	nonzero	???
1-254	anything	+/- floating-point
255	0	???
255	nonzero	???

Representation for 0

- Represent 0?
  - exponent all zeroes
  - significand all zeroes too
  - What about sign?
  - `+0: 0 00000000 00000000000000000000000`
  - `-0: 1 00000000 00000000000000000000000`
- Why two zeroes?
  - Helps in some limit comparisons

Special Numbers

- What have we defined so far? (single precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- floating-point
255	0	???
255	nonzero	???

- Range:

$1.0 \times 2^{-126} \approx 1.8 \times 10^{-38}$

What if result too small? (>0, < 1.8×10^{-38} => Underflow!)

$(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$

What if result too large? (> 3.4×10^{38} => Overflow!)

Gradual Underflow

- Represent denormalized numbers (denorms)
  - Exponent : all zeroes
  - Significand : non-zeroes
  - Allow a number to degrade in significance until it become 0 (gradual underflow)

  - The smallest normalized number
    - $1.0000\ 0000\ 0000\ 0000\ 0000\ 0000 \times 2^{-126}$
  - The smallest de-normalized number
    - $0.0000\ 0000\ 0000\ 0000\ 0000\ 0001 \times 2^{-126} = 1.0 \times 2^{-149}$

Special Numbers

- What have we defined so far? (single precision)

Exponent	Significand	Object
0	0	0
0	nonzero	denorm
1-254	anything	+/- floating-point
255	0	???
255	nonzero	???

Representation for +/- Infinity

- In FP, divide by zero should produce +/- infinity, not overflow
- Why?
  - OK to do further computations with infinity, e.g., X/0 > Y may be a valid comparison
- IEEE 754 represents +/- infinity
  - Most positive exponent reserved for infinity
  - Significands all zeroes

S	1111 1111	0000 0000 0000 0000 0000 000

Special Numbers (cont'd)

- What have we defined so far? (single-precision)

Exponent	Significand	Object
0	0	0
0	nonzero	denom
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	???

Representation for Not a Number

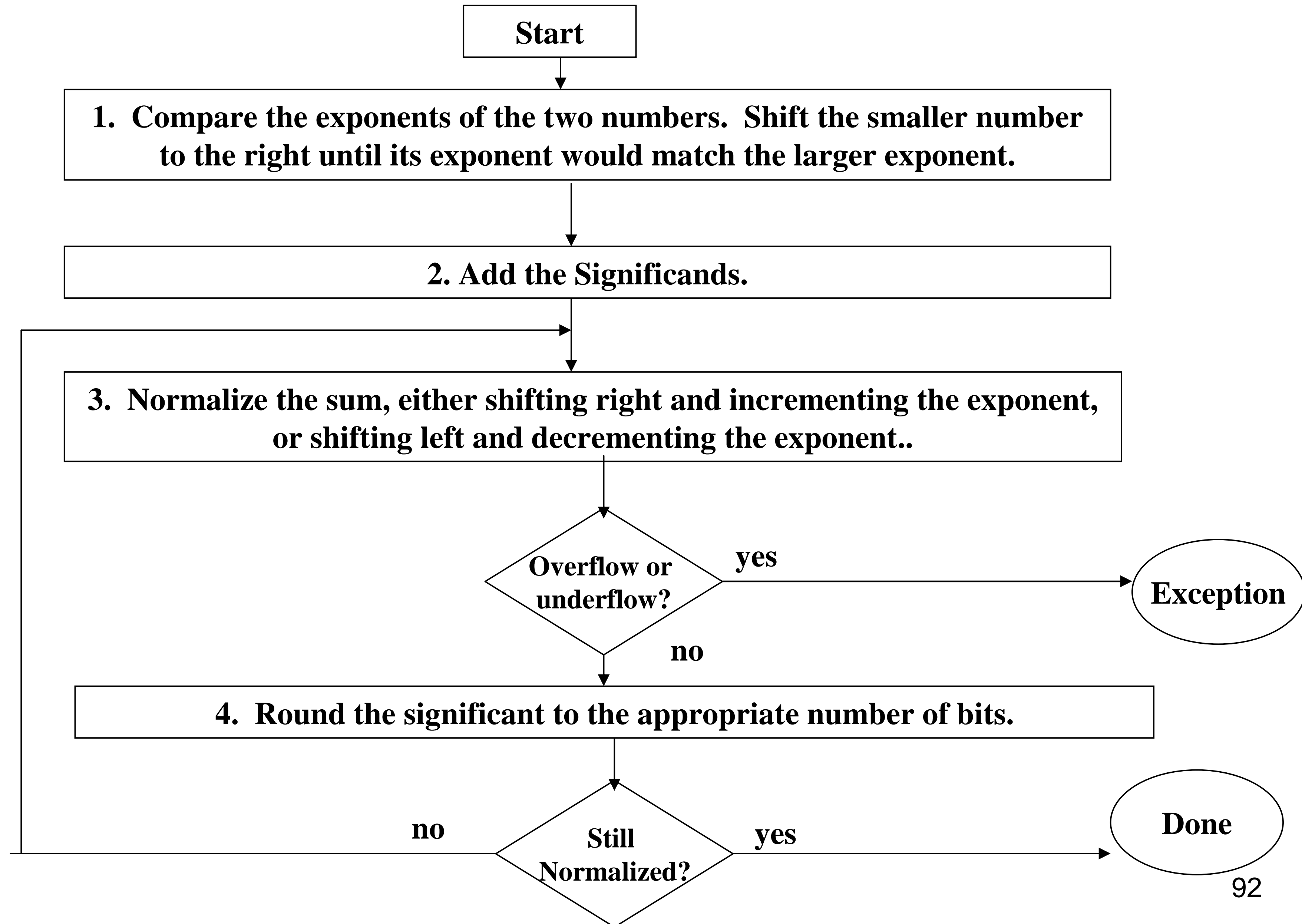
- What do I get if I calculate sqrt(-4.0) or 0/0?
  - If infinity is not an error, these should not be either
  - They are called *Not a Number* (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN,X) = NaN
  - OK if calculate but don't use it

Special Numbers (cont'd)

- What have we defined so far? (single-precision)

Exponent	Significand	Object
0	0	0
0	nonzero	denom
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	NaN

Floating Point Addition



Basic Binary F.P. Addition Algorithm

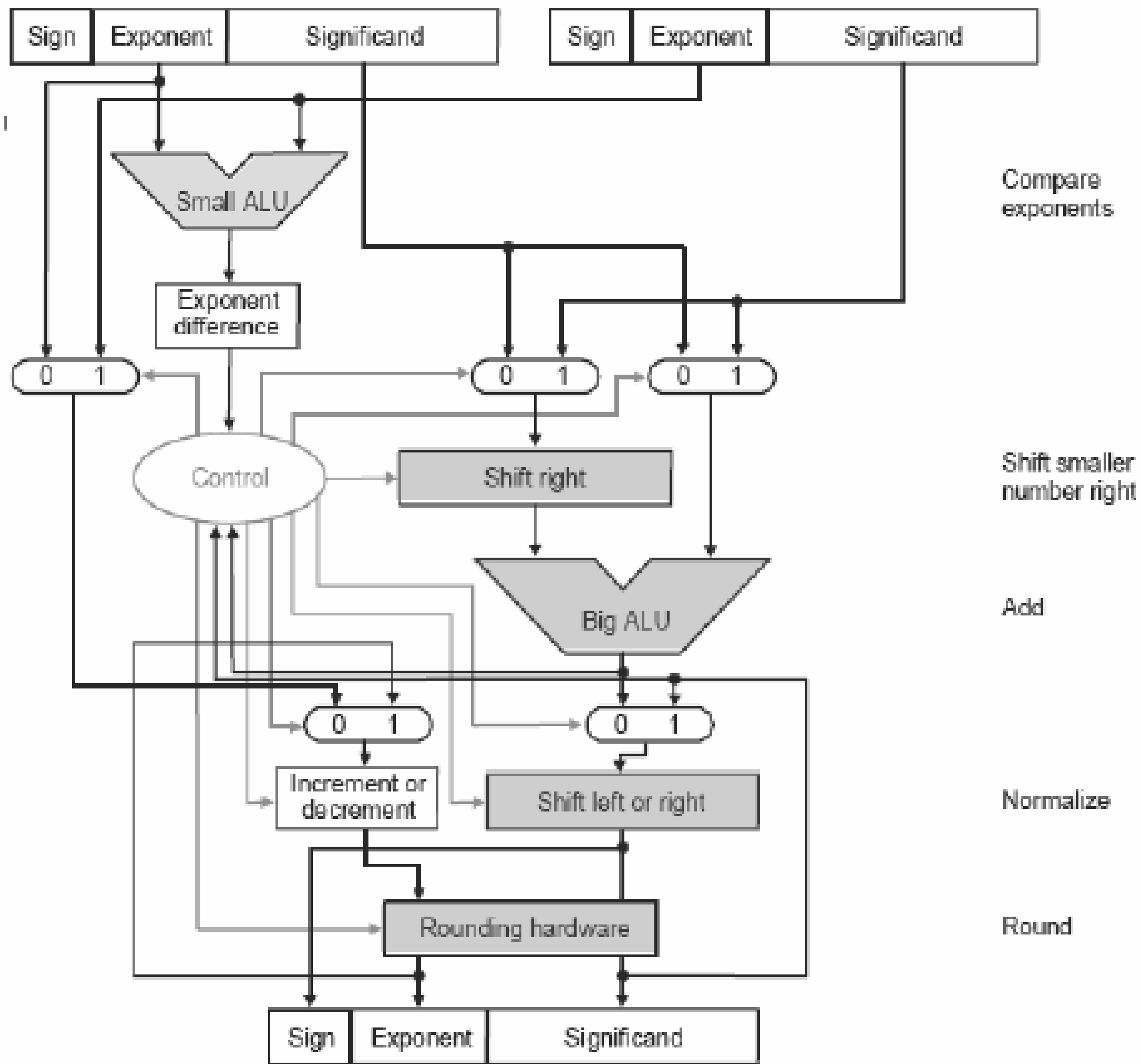
For addition (or subtraction) of X to Y ($X < Y$):

1. **Compute $D = \text{Exp}_Y - \text{Exp}_X$ (align binary points)**
2. **Right shift $(1+\text{Sig}_X)$ D bits $\Rightarrow (1+\text{Sig}_X)*2^{-D}$**
3. **Compute $(1+\text{Sig}_X)*2^{-D} + (1+\text{Sig}_Y)$; Normalize if necessary; continue until MS bit is 1**

4. **Too small (*e.g.*, 0.001xx...) left shift result, decrement result exponent; check for underflow**

4'. **Too big (*e.g.*, 10.1xx...)
right shift result, increment result exponent; check for overflow**

5. **If result significand is 0, set exponent to 0**


Sign
Exponent
Significand
Sign
Exponent
Significand
Compare exponents
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Shift smaller number right
Big ALU
Add
0 1
0 1
Increment or decrement
Shift left or right
Normalize
Rounding hardware
Round
Sign
Exponent
Significand

Floating-Point Addition

- $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
  - Assume we can store only four decimal digits of the significand, and two decimal digits of the exponent
- Basic addition algorithm:
  - (1) align the decimal point

    $1.610 \times 10^{-1} = 0.01610 \times 10^{1}$
  - (2) add two significands

    $9.999 + 0.01610 = 10.015$
  - (3) normalize the sum

    $10.015 \times 10^{1} = 1.0015 \times 10^{2}$
  - (4) round the significand

    1.002×10^{2}

Example: Decimal F.P. Addition

- Assume 4 digit significand, 2 digit exponent
- Let's add $9.999_{ten} \times 10^1 + 1.610_{ten} \times 10^{-1}$
- Exponents must match, so adjust smaller number to match larger exponent

  $1.610 \times 10^{-1} = 0.1610 \times 10^0 = 0.01610 \times 10^1$

- Can represent only 4 digits, so must discard last two:

  0.016×10^1

Example: Decimal F.P. Addition

- Now, add significands:

$$
\begin{array}{r} 9.999 \\ +\ 0.016 \\ \hline 10.015 \end{array}
$$

- Thus, sum is 10.015×10^1
- Sum is not normalized, so correct it, checking for underflow/overflow:

$$10.015 \times 10^1 \Rightarrow 1.0015 \times 10^2$$

- Cannot store all digits, must round. Final result is:

$$\boxed{1.002 \times 10^2}$$

Example

- $0.5_{10}+(-0.4375_{10})$
  - $0.5_{10} = ½_{10} = 0.1^2 = 0.1_2*2^0 = 1.000_2 *2^{-1}$
  - $-0.4375_{10} = -7/16_{10} = -7/2^4{}_{10}= -0.0111_2*2^0 = -1.110_2 *2^{-2}$
- Step 1: $-1.110_2 *2^{-2} =-0.111_2 *2^{-1}$
- Step 2: $1.000_2 *2^{-1} + (-0.111_2 *2^{-1})=0.001_2*2^{-1}$
- Step 3: $0.001_2*2^{-1}=1.000_2*2^{-4}$
- Step 4: $1.000_2*2^{-4}=0.0001_2=1/2^4{}_{10}=1/16_{10}=0.0625_{10}$

F.P. Subtraction

- Similar to addition
- How do we do it?
  - De-normalize to match exponents
  - Subtract significands
  - Keep the same exponent
  - Normalize (possibly changing exponent)
- Problems in implementing FP add/sub:
  - Managing the signs,
  - determining to add or sub,
  - swapping the operands.
- Question: How do we integrate this into the integer arithmetic unit?
- Answer: We don’t!

Floating-Point Multiplication

Basic multiplication algorithm

(1) add exponents of operands to get exponent of product

doubly biased exponent must be corrected:

Xe = 7

Ye = -3

Excess 8

Xe = 1111	**= 15**	**= 7 + 8**
Ye = 0101	**= 5**	**= -3 + 8**
10100	**20**	**4 + 8 + 8**

need extra subtraction step of the bias amount

(2) multiplication of operand mantissa

(3) normalize the product

(3.1) check overflow or underflow during the shift

(3.2) round the mantissa

continue until MSB of data is 1

(4) set the sign of product

Example: Decimal F. P. Multiply (1/2)

- Let's multiply:

$$1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$$

(Assume 4-digit significand, 2-digit exponent)

- First, add exponents:

$$\begin{array}{r} 10 \\ + -5 \\ \hline 5 \end{array}$$

- Next, multiply significands:

$$1.110 \times 9.200 = 10.212000$$

Example: Decimal F. P. Multiply (2/2)

- Product is not normalized, so correct it, checking for underflow / overflow:

  $10.212000 \times 10^5 \Rightarrow 1.0212 \times 10^6$

- Significand exceeds 4 digits, so round:

  1.021×10^6

- Check signs of original operands
  same $\Rightarrow$ positive
  different $\Rightarrow$ negative

  Final result is: $+1.021 \times 10^6$

Basic Binary F.P. Multiplication Algorithm

For multiplication of $P = X \times Y$:

1. Compute Exponent: $\text{Exp}_P = (\text{Exp}_Y + \text{Exp}_X) - \text{Bias}$

2. Compute Product: $(1 + \text{Sig}_X) \times (1 + \text{Sig}_Y)$

   Normalize if necessary; continue until most significant bit is 1

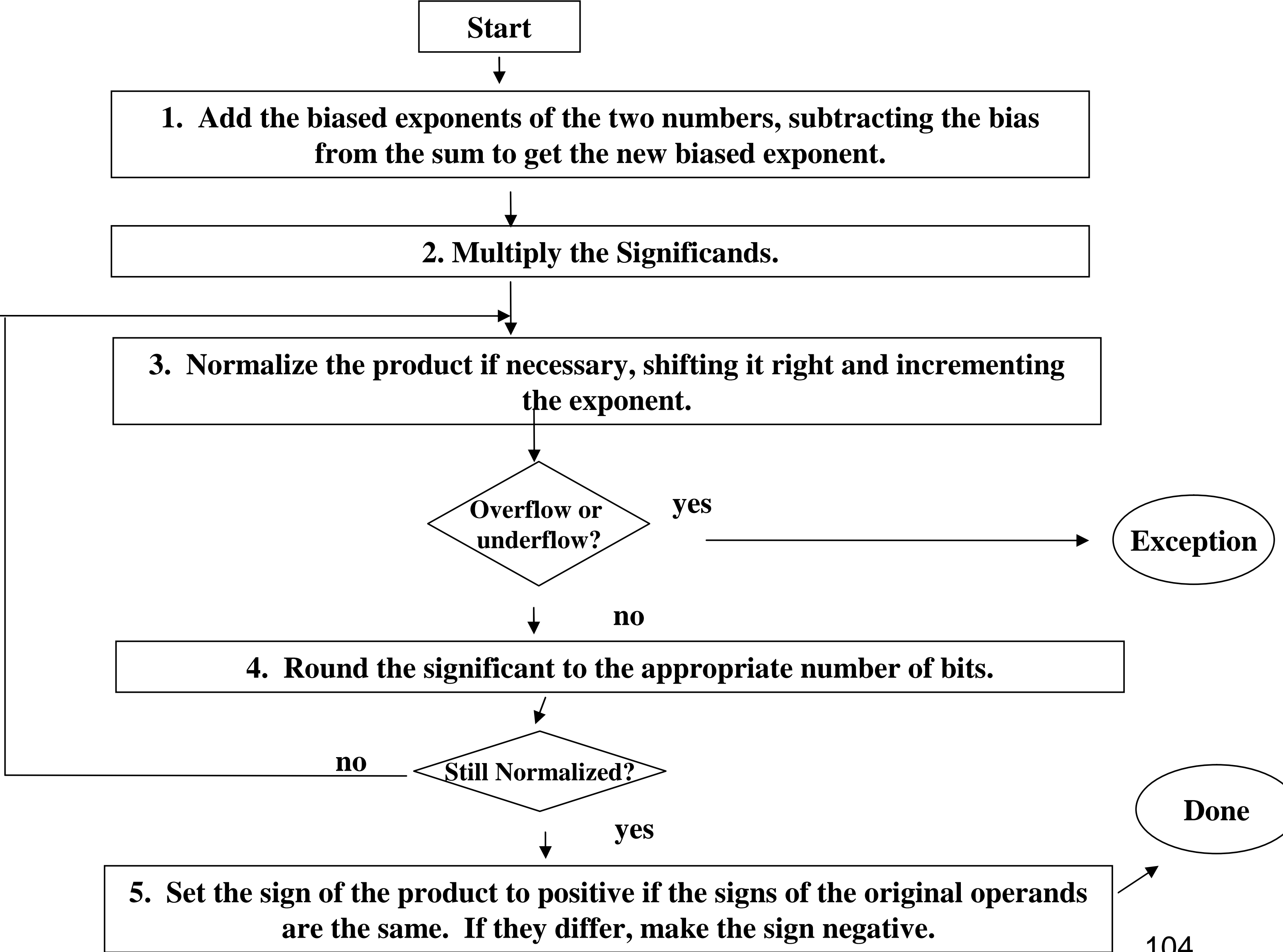
4. Too small (*e.g.*, 0.001xx...) $\rightarrow$
   left shift result, decrement result exponent

4'. Too big (*e.g.*, 10.1xx...) $\rightarrow$
   right shift result, increment result exponent

5. If (result significand is 0) then set exponent to 0

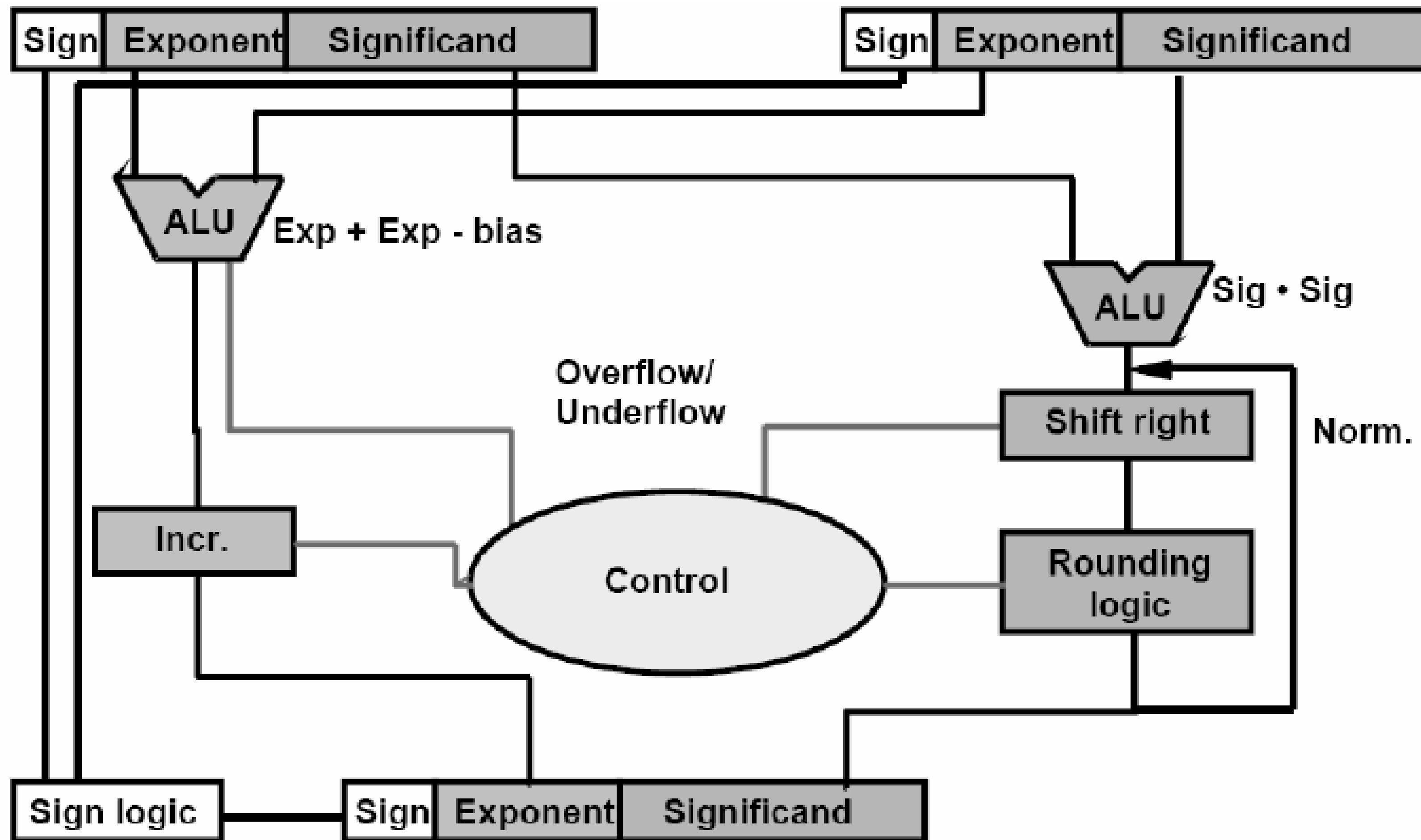
6. if ($\text{Sgn}_X == \text{Sgn}_Y$) then
   Sgn_P = positive (0)
   else
   Sgn_P = negative (1)

F.P. Multiplication Algorithm



- Add exponents
- Multiply the significands
- Normalize
- Over- underflow
- Rounding
- Sign

F.P. Multiplication



Example

- $0.5_{10}*(-0.4375_{10})$
  - $0.5_{10} = ½_{10} = 0.1^2 = 0.1_2*2^0 = 1.000_2 *2^{-1}$
  - $-0.4375_{10} = -7/16_{10} = -7/2^4{}_{10}= -0.0111_2*2^0 = -1.110_2 *2^{-2}$
- Step 1: -1+(-2)= -3
- Step 2: $1.000_2 * 1.110_2$=1.110000

  the product is equal to 1.110_2*2^{-3}
- Step 3: normalization
- Step 4: rounding the product
- Step 5: set the sign bit

  -1.110_2*2^{-3}=-0.001110_2=$-7/2^5{}_{10}$=-0.21875_{10}

F.P. Division

- Subtraction of exponents
- Division of the significands
- Normalization
- Rounding
- Sign

- Remarks
- Divide by zero: check first!

MIPS Floating Point Architecture (1/4)

- Separate floating point instructions:
  - Single Precision:

    `add.s, sub.s, mul.s, div.s`
  - Double Precision:

    `add.d, sub.d, mul.d, div.d`
- These instructions are far more complicated than their integer counterparts, so they can take much longer to execute.

FP Part of the Processor

- contains 32 32-bit registers: \$f0, \$f1, …
- most registers specified in .s and .d instruction refer to this set
- separate load and store: lwc1 and swc1
- Double Precision: by convention, even/odd pair contain one DP FP number
  - \$f0/\$f1, \$f2/\$f3
- Instructions to move data between main processor and coprocessors:
  - **mfc0, mtc0, mfc1, mtc1, etc.**

MIPS Floating Point Architecture (1/3)

- Problems:

  It's inefficient to have different instructions take vastly differing amounts of time.

  Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.

  Some programs do no floating point calculations

  It takes lots of hardware relative to integers to do Floating Point fast

MIPS Floating Point Architecture (2/3)

- 1990 Solution: Make a completely separate chip that handles only FP.

◦ Coprocessor 1: FP chip

1. contains 32 32-bit registers: `$f0`, `$f1`, …
2. most of the registers specified in `.s` and `.d` instruction refer to this set
3. separate load and store: `lwc1` and `swc1` ("load word coprocessor 1", "store …")
4. Double Precision: by convention, even/odd pair contain one DP FP number: `$f0`/`$f1`, `$f2`/`$f3`, … , `$f30`/`$f31`

MIPS Floating Point Architecture (3/3)

- 1990 Computer actually contains multiple separate chips:

  Processor: handles all the normal stuff

  Coprocessor 1: handles FP and only FP;

  more coprocessors?… Yes, later

  Today, FP coprocessor integrated with CPU, or cheap chips may leave out FP HW

- Instructions to move data between main processor and coprocessors:

```
mfc1  rt, rd              Move floating point register rd to
                           CPU register rt.
mtc1  rd, rt              Move CPU register rt to floating
                           point register rd.
mfc1.d  rdest, frsrc1      Move floating point registers
                           frsrc1 & frsrc1 + 1 to CPU
                           registers rdest & rdest + 1.
```

- Appendix pages A-70 to A-74 contain many, many more FP operations.

MIPS R2000 Organization

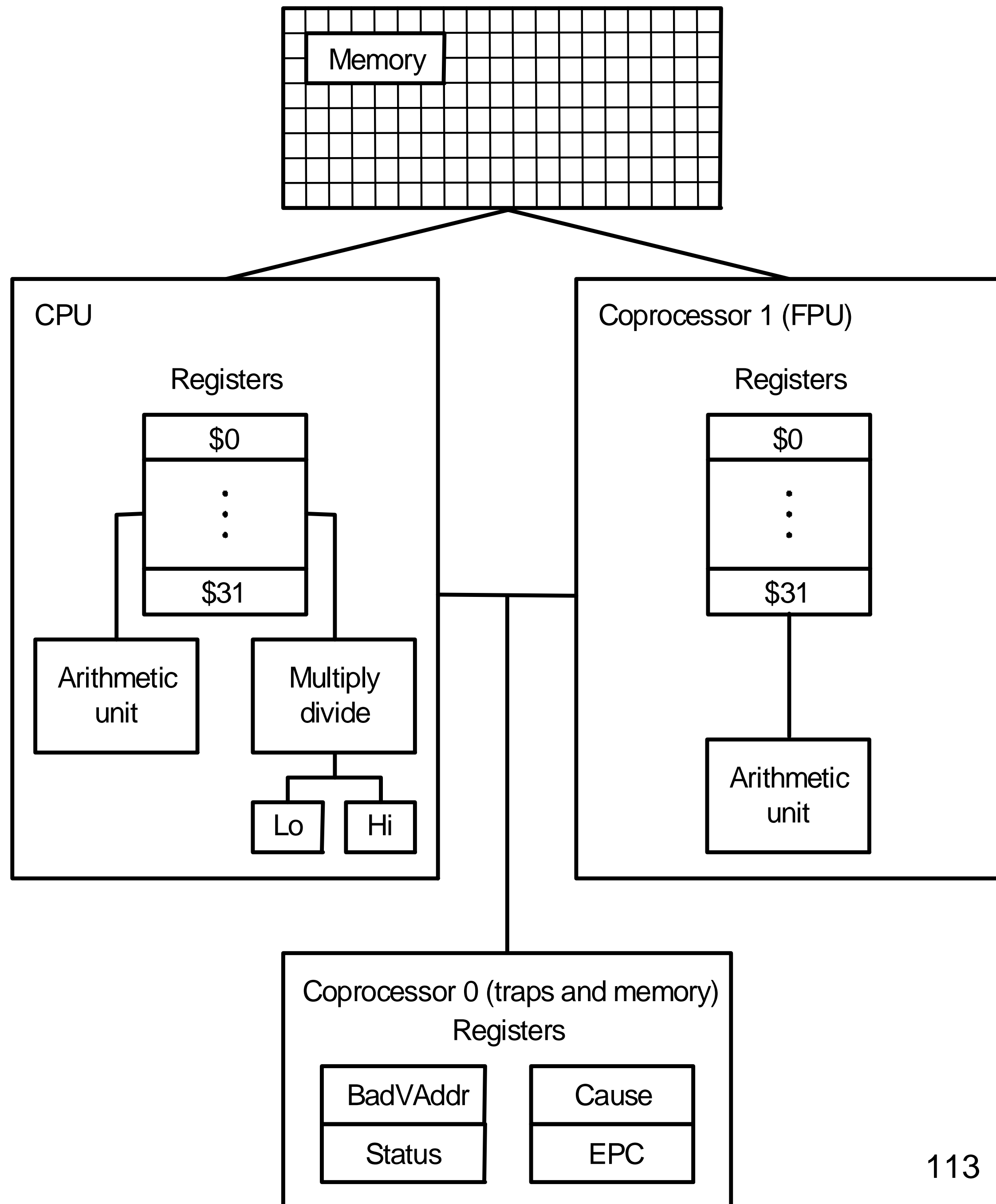


Fig. A.10.1

MIPS Floating Point

- Separate floating point instructions:
  - Single precision: `add.s`,`sub.s`,`mul.s`,`div.s`
  - Double precision: `add.d`,`sub.d`,`mul.d`,`div.d`
- FP part of the processor:
  - contains 32 32-bit registers: `$f0`, `$f1`, …
  - most registers specified in .s and .d instruction refer to this set
  - separate load and store: `lwc1` and `swc1`
  - Double Precision: by convention, even/odd pair contain one DP FP number: `$f0/$f1`, `$f2/$f3`
  - Instructions to move data between main processor and coprocessors:
    - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.
  - See CD A-73 to A-80

Example with F.P.: Matrix Multiply

```
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];

}
```

- Starting addresses are parameters in \$a0, \$a1, and \$a2. Integer variables are in \$t3, \$t4, \$t5. Arrays 32 by 32
- Use pseudoinstructions: `li` (load immediate), `l.d` / `s.d` (load / store 64 bits)

MIPS code 1st piece: initialize `x[ ][ ]`

- Initialize Loop Variables

```
mm:  ...
     li   $t1, 32    # $t1 = 32
     li   $t3, 0     # i = 0; 1st loop
L1:  li   $t4, 0     # j = 0; reset 2nd
L2:  li   $t5, 0     # k = 0; reset 3rd
```

- To fetch x[i][j], skip i rows (i*32), add j

```
 sll      $t2,$t3,5 # $t2 = i * 2^5
 addu     $t2,$t2,$t4     # $t2 = i*2^5 + j
```

- Get byte address (8 bytes), load x[i][j]

```
 sll      $t2, $t2,3# i,j byte addr.
 addu         $t2, $a0,$t2# @ x[i][j]
 l.d      $f4, 0($t2)     # $f4 = x[i][j]
```

MIPS code 2nd piece: z[][], y[][]

- Like before, but load z[k][j] into $f16

```
L3:  sll     $t0, $t5, 5      # $t0 = k * 2^5
     addu    $t0, $t0, $t4    # $t0 = k*2^5 +j
     sll     $t0, $t0, 3      # k,j byte addr.
     addu    $t0, $a2, $t0    # @ z[k][j]
     l.d     $f16, 0($t0)     # $f16 = z[k][j]
```

- Like before, but load y[i][k] into $f18

```
     sll     $t0, $t3, 5      # $t0 = i * 2^5
     addu    $t0, $t0, $t5    # $t0 = i*2^5 +k
     sll     $t0, $t0, 3      # i,k byte addr.
     addu    $t0, $a1, $t0    # @ y[i][k]
     l.d     $f18, 0 ($t0)    # $f18 = y[i][k]
```

- Summary: `$f4: x[i][j], $f16: z[k][j], $f18: y[i][k]`

MIPS code for last piece: add/mul, loops

- Add y*z to x

```
mul.d $f16,$f18,$f16        # y[][]*z[][]
add.d $f4, $f4, $f16        # x[][]+ y*z
```

- Increment k; if end of inner loop, store x

```
addiu $t5, $t5,1            # k = k + 1
bne         $t5, $t1,L3     # if(k!=32) goto L3
s.d         $f4, 0($t2)     # x[i][j] = $f4
```

- Increment j; middle loop if not end of j

```
addiu $t4, $t4,1            # j = j + 1
bne         $t4, $t1,L2     # if(j!=32) goto L2
```

- Increment i; if end of outer loop, return

```
addiu $t3,$t3,1             # i = i + 1
bne         $t3,$t1,L2              # if(i!=32) goto L1
jr          $ra
```

Floating Point Gottchas: Add Associativity?

- $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$
- $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$

  $= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = \underline{0.0}$
- $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$
- $= (0.0) + 1.0 = 1.0$
- Therefore, Floating Point addition <u>not associative!</u>
  - 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ is still 1.5×10^{38}
  - FP result approximation of real result!
- What are the conditions that make smaller arguments "disappear" (rounded down to 0.0)?

Floating Point Fallacy

- FP add, subtract associative?

  FALSE!

  – $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, $z = 1.0$

  – $x + (y + z) = -1.5\text{x}10^{38} + (1.5\text{x}10^{38} + 1.0)$

  $= -1.5\text{x}10^{38} + (1.5\text{x}10^{38}) = 0.0$

  – $(x + y) + z = (-1.5\text{x}10^{38} + 1.5\text{x}10^{38}) + 1.0$

  $= (0.0) + 1.0 = 1.0$

- Therefore, Floating Point add, subtract are not associative!

  – Why? FP result approximates real result!

  – This example: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

Summary

- MIPS arithmetic: successive refinement to see final design
  - 32-bit adder and logic unit
  - 32-bit multiplier and divisor, with HI and LO
  - Booth's algorithm to handle signed multiplies
- Floating point numbers *approximate* values that we want to use
  - IEEE 754 Floating Point Standard is most widely accepted to standardize their interpretation
  - New MIPS registers (`$f0-$f31`) and instructions:
    - Single-precision (32 bits, $2 \times 10^{-38} \ldots 2 \times 10^{38}$): `add.s, sub.s, mul.s, div.s`
    - Double-precision (64 bits , $2 \times 10^{-308} \ldots 2 \times 10^{308}$): `add.d, sub.d, mul.d, div.d`
- Type is not associated with data, bits have no meaning unless given in context