# Computer Organization and Design
The Hardware/Software Interface

## Chapter 3 - Arithmetic for Computers

**Dr. Feng Li**
fli@sdu.edu.cn
https://funglee.github.io

# Contents of Chapter 3

## Introduction

- Computer words are composed of bits; thus words can be represented as binary numbers.
- What about fractions and other real numbers?
- What happen if an operation creates a number bigger than can be represented
- And underlying these questions is a mystery: How does hardware really multiply or divide numbers?

# Addition and subtraction

- Adding bit by bit, carries -> next digit

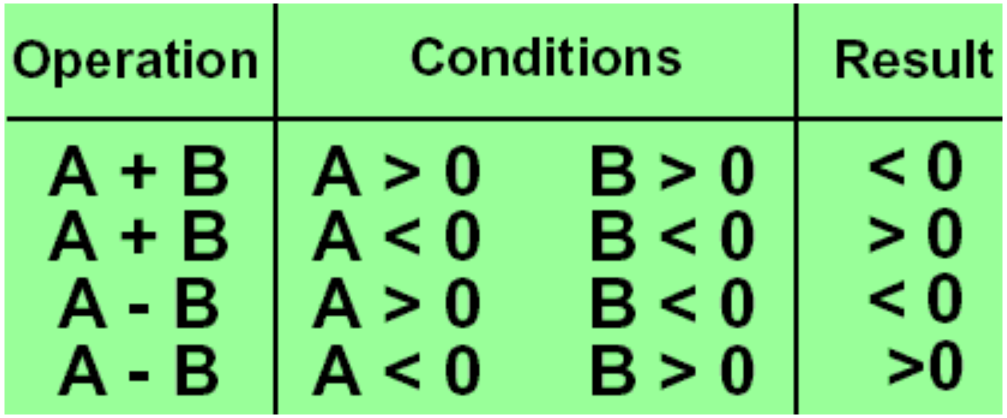
$$\begin{array}{rr} 0000\ 0111 & 7_{10} \\ +\underline{0000\ 0110} & 6_{10} \\ 0000\ 1101 & 13_{10} \end{array}$$

- Subtraction
  - Directly
  - Addition of 2's complement

$$\begin{array}{rr} 0000\ 0111 & 7_{10} \\ -\underline{0000\ 0110} & 6_{10} \\ 0000\ 0001 & 1_{10} \end{array} \qquad \begin{array}{rr} 0000\ 0111 & 7_{10} \\ +\underline{1111\ 1010} & -6_{10} \\ 0000\ 0001 & 1_{10} \end{array}$$

## Overflow

- Overflow occurs when the result from an operation cannot be represented with available hardware
- When adding operands with different signs or subtracting operands with the same sign, overflow will not occur.
- Adding or subtracting two 32-bit numbers can yield a result that needs 33 bits to be fully represented. The lack of the 33rd bit means that when overflow occurs, the sign bit is set with the value of the result instead of the proper sign of the result



| Operation | Conditions | | Result |
|---|---|---|---|
| A + B | A > 0 | B > 0 | < 0 |
| A + B | A < 0 | B < 0 | > 0 |
| A - B | A > 0 | B < 0 | < 0 |
| A - B | A < 0 | B > 0 | >0 |

## Overflow

- Reaction on overflow
  - Ignore ?
  - Reaction of the OS
  - Signalling to application (Ada, Fortran,...)
- Hardware detection in the ALU
- Generation of an exception (interrupt)
- Save the instruction address in special register EPC (Exception Program Counter)
- Jump to specific routine in OS
  - Correct & return to program
  - Return to program with error code
  - Abort program

## Overflow

- Overflows in signed arithmetic instructions cause exceptions:
  - add (add)
  - add immediate (addi)
  - subtract (sub)
- Overflows in unsigned arithmetic instructions don’t cause exceptions:
  - add unsigned (addu)
  - add immediate unsigned (addiu)
  - subtract unsigned (subu)

## Conditional branch that test for overflow

- No conditional branch to test overflow in MIPS
- Sequence to discover  overflow for signed addition

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
                             # so no overflow
xor  $t3, $t0, $t1 # signs =; sign of sum match too?
                   # $t3 negative if sum sign different
slt  $t3, $t3, $zero # $t3 = 1 if sum sign different
bne  $t3, $zero, Overflow # All 3 signs ≠; go to overflow
```

# Arithmetic for multimedia

- Graphical displays
  - 8 bits to represent each of the three primary colors plus 8 bits for a locations of a pixel
- Audio
  - More than 8 bits of precision, but 16 bits are sufficient
- SIMD
  - Single instruction, multiple data

| Instructions category | Operands |
|---|---|
| Unsigned add/sub | Eight 8-bits or four 16 bits |
| saturating add/sub | Eight 8-bits or four 16 bits |
| Max/min/minimum | Eight 8-bits or four 16 bits |
| Average | Eight 8-bits or four 16 bits |
| Shift right/left | Eight 8-bits or four 16 bits |

Multimedia support for desktop computer

# Multiplication

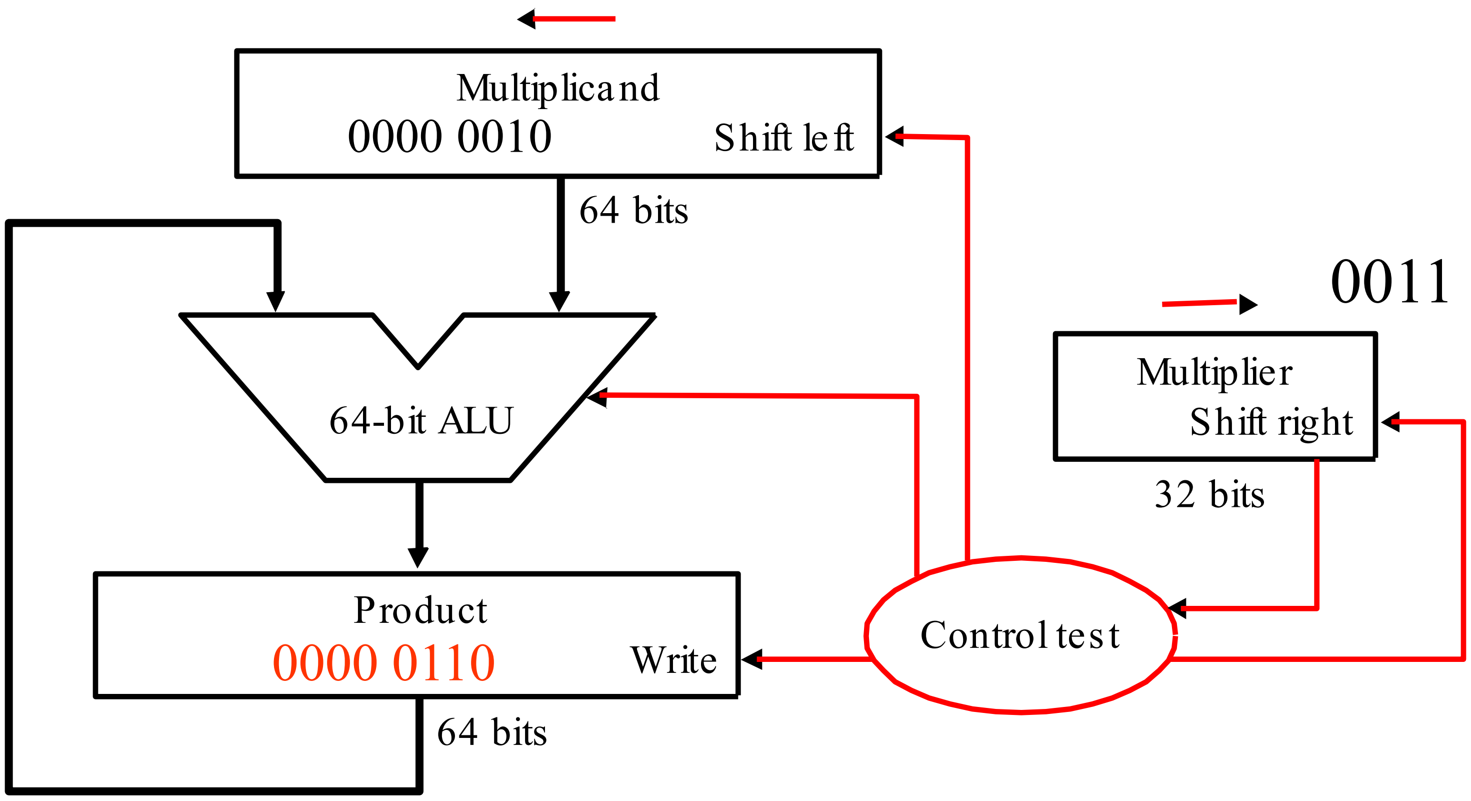
- **Binary multiplication**

$$
\begin{array}{llr}
\text{Multiplicand} & & 1000_{ten} \\
\text{Multiplier} & \times & 1001_{ten} \\
\hline
 & & 1000 \\
 & & 0000\phantom{0} \\
 & & 0000\phantom{00} \\
 & & 1000\phantom{000} \\
\hline
\text{Product} & & 1001000_{ten}
\end{array}
$$

- **Look at current bit position**
  - **If multiplier is 1**
    - **then add multiplicand**
    - **Else add 0**
  - **shift multiplicand left by 1 bit**

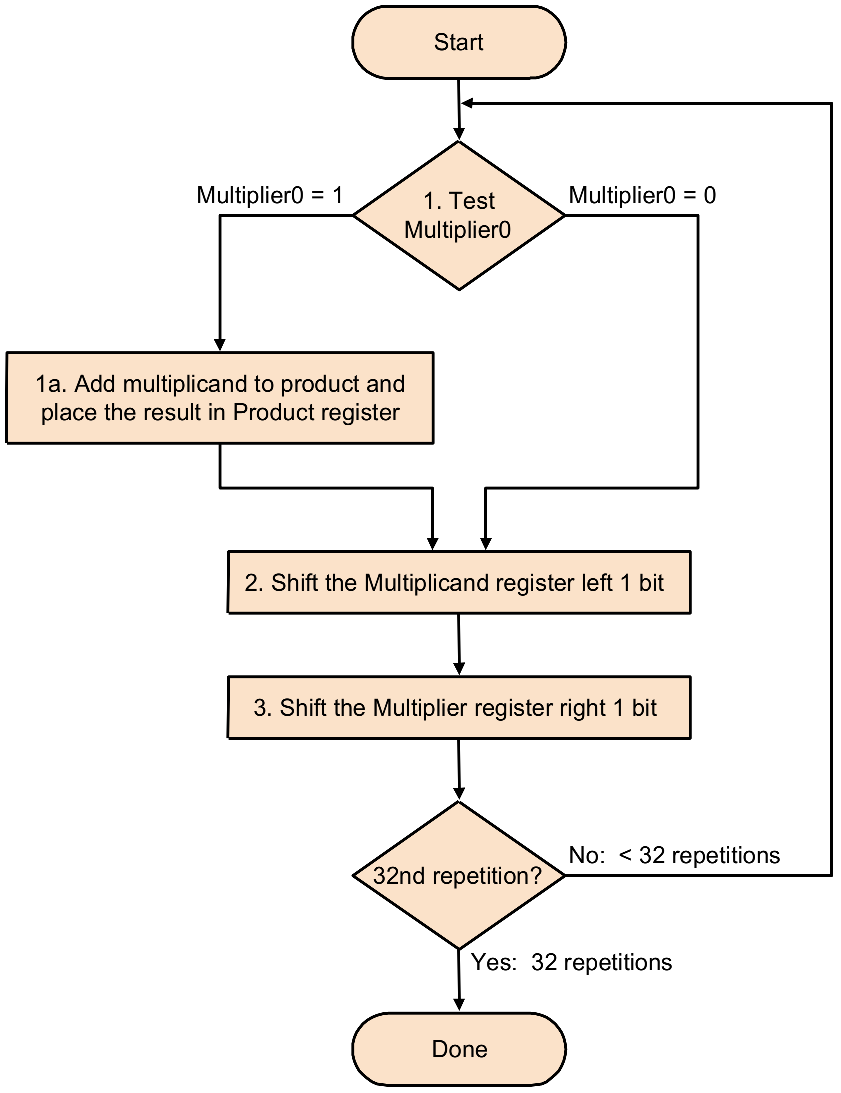
# Multiplier Sequential Version

- 32 bits: multiplier
- 64 bits: multiplicand, product, ALU
- 0010*0011



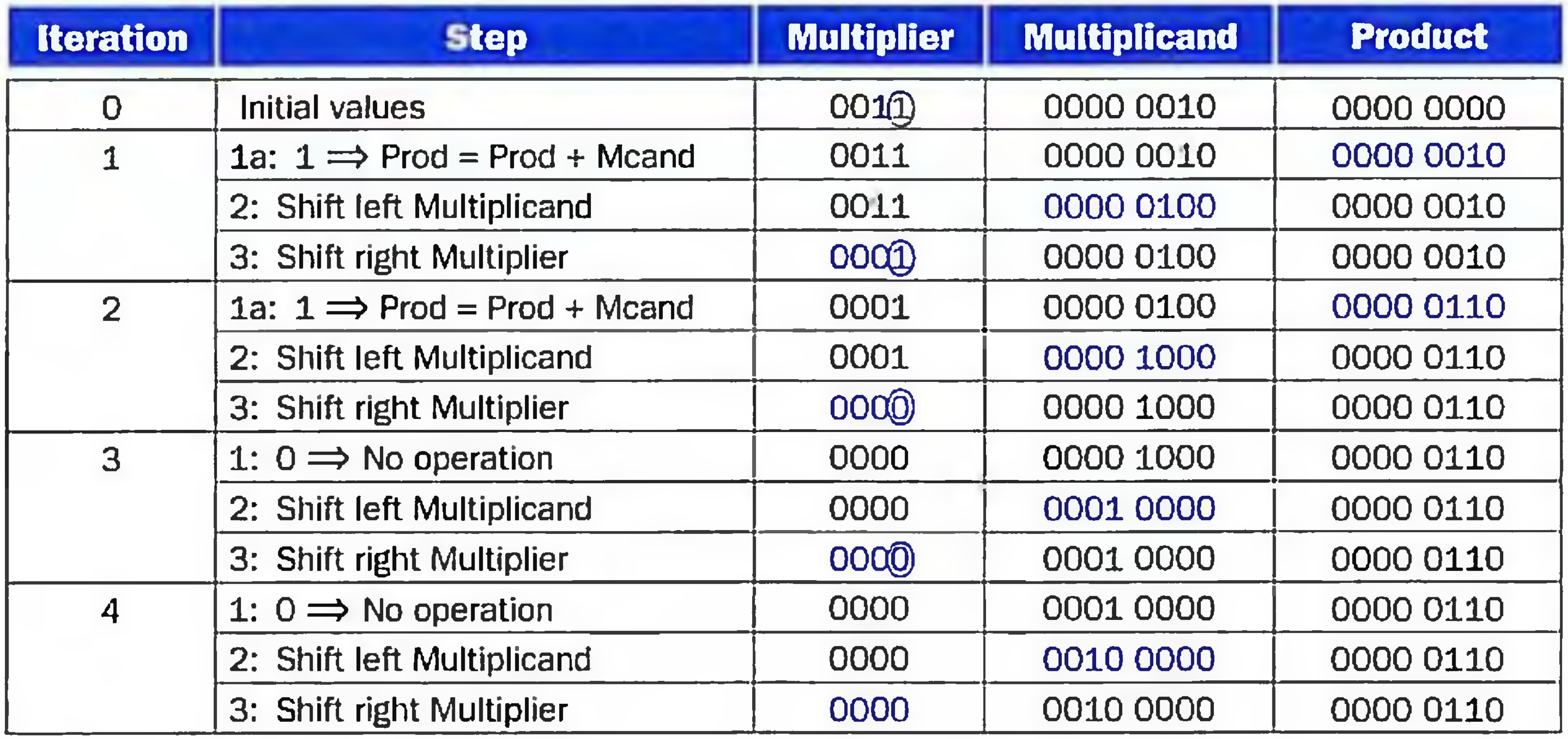
# Multiplier Sequential Version

- **Requires 32 iterations**
  - **Addition**
  - **Shift**
  - **Comparison**
- **Almost 100 cycles**
- **Very big, Too slow!**


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

# Example

- 0010*0011



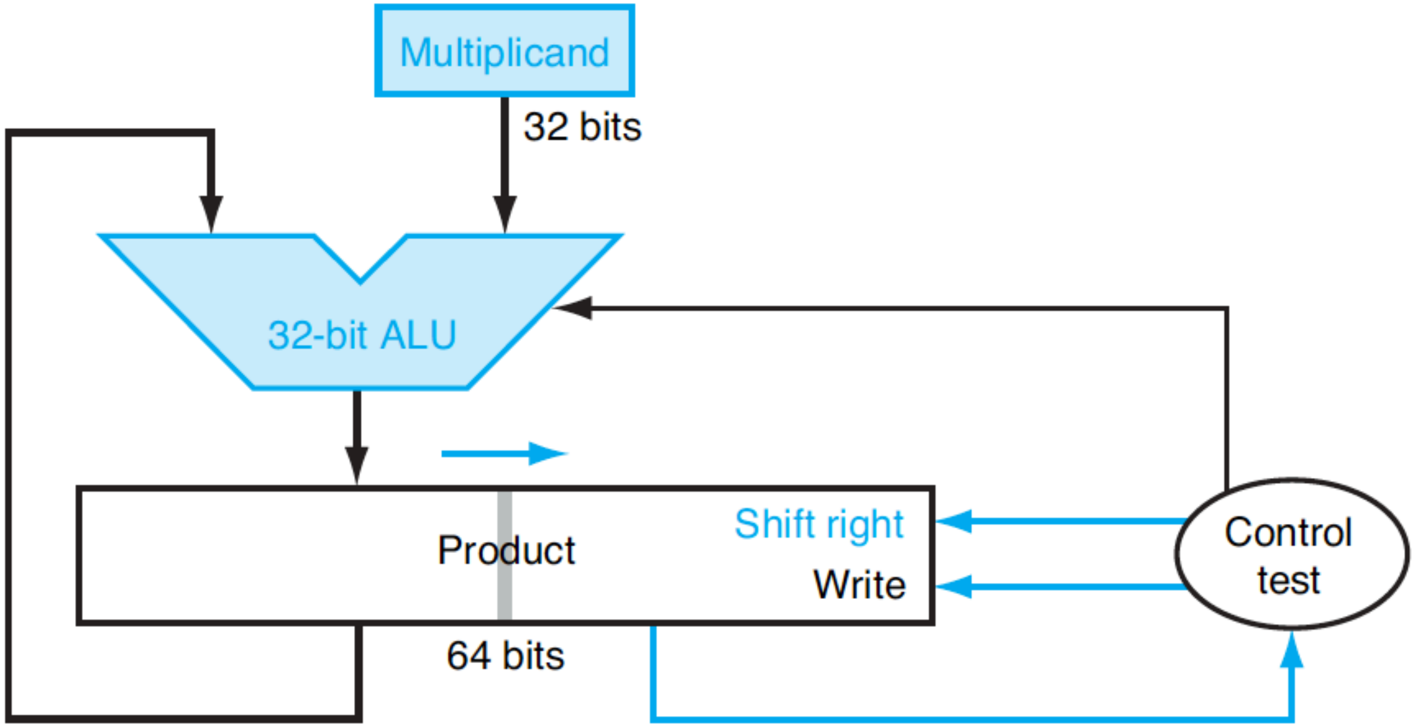
| Iteration | Step | Multiplier | Multiplicand | Product |
|---|---|---|---|---|
| 0 | Initial values | 0011 | 0000 0010 | 0000 0000 |
| 1 | 1a: 1 ⟹ Prod = Prod + Mcand | 0011 | 0000 0010 | 0000 0010 |
| | 2: Shift left Multiplicand | 0011 | 0000 0100 | 0000 0010 |
| | 3: Shift right Multiplier | 0001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

## Multiplier V2

- Real addition is performed only with 32 bits
- Least significant bits of the product don't change
- New idea:
  - Don’t shift the multiplicand
  - Instead, shift the product
  - Shift the multiplier
- ALU reduced to 32 bits!

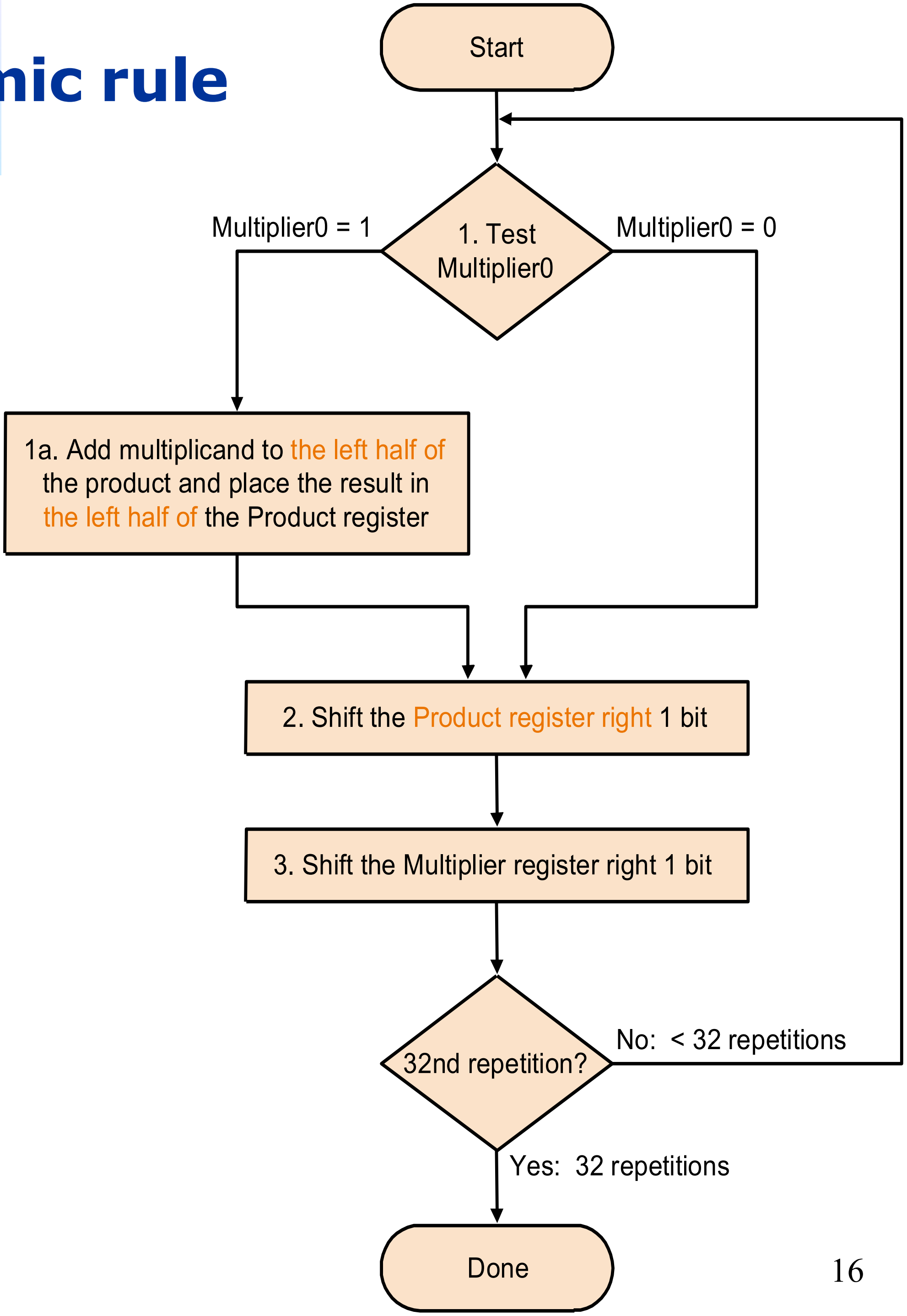
# Multiplier V2-- Logic Diagram

- Diagram of the V2 multiplier
- Only left half of product register is changed



## Multiplier V2----Algorithmic rule

- **Addition performed only on left half of product register**
- **Shift of product register**


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

# Revised 4-bit example with V2

- **Multiplicand x multiplier: 0001 x 0111**

```
Multiplicand:   0001
Multiplier:×    0111
              ------
                00000111              #Initial value for the product
1               00010111              #After adding 0001, Multiplier=1
---------------------------
                00001011      1       #After shifting right the product one bit
                0001
              ------
2               00011011              #After adding 0001, Multiplier=1
---------------------------
                00001101      11      #After shifting right the product one bit
                0001                  #After adding 0001, Multiplier=1
              ------
3               00011101
---------------------------
                00001110      111     #After shifting right the product one bit
                0000
              ------
4               00001110              #After adding 0001, Multiplier=0
---------------------------
                00000111      0111    #After shifting right the product one bit
```

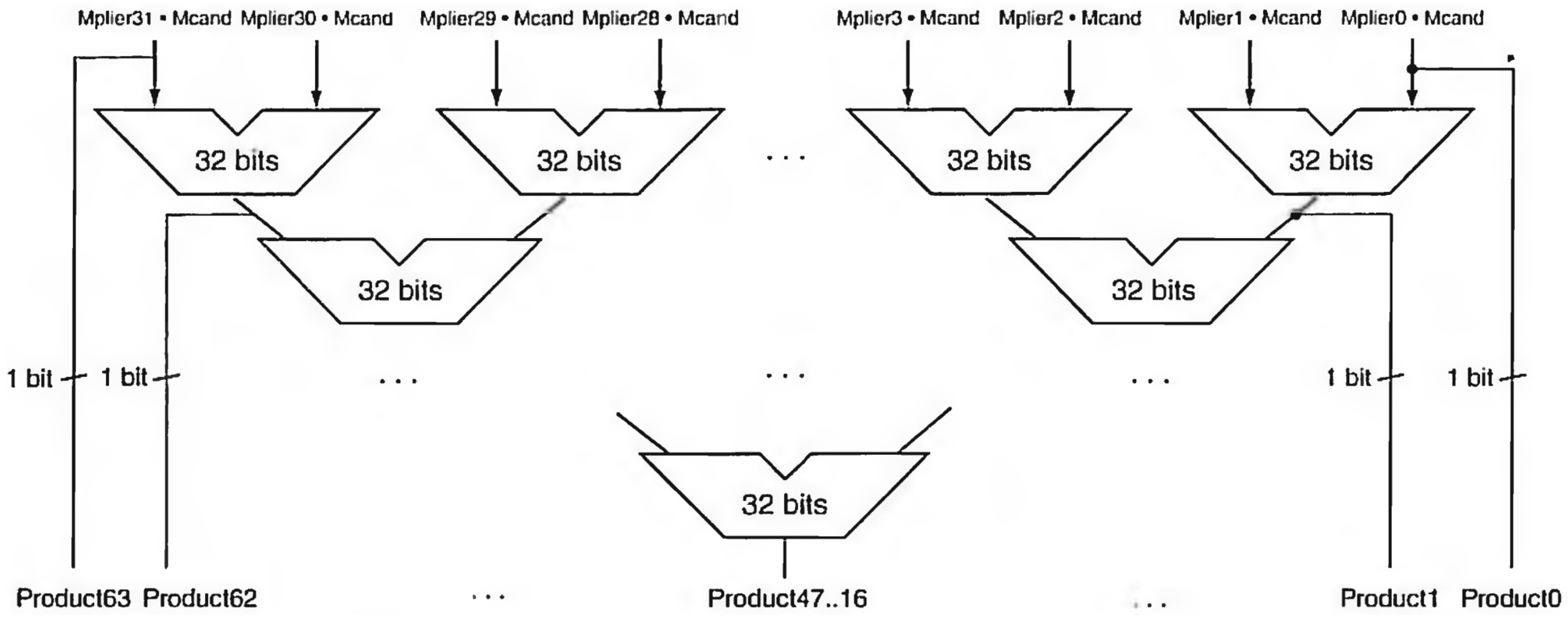
## Signed multiplication

- **Basic approach:**
  - Store the signs of the operands
  - Convert signed numbers to unsigned numbers (most significant bit (MSB) = 0)
  - Perform multiplication
  - If sign bits of operands are equal

```
  sign bit = 0,
else
  sign bit = 1
```

## Faster Multiplication

- One 32-bit adder for each bit of the multiplier
  - One input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder



## Multiply in MIPS

- mult instruction for signed multiply
  - mult reg1, reg2
  - Registers Hi and Lo contain the 64-bit product
  - E.g.

```
mult $s2, $s3  #Hi,Lo=$s2*$s3
mflo  $s1      # $s1=Lo
mfhi  $s2      # $s2=Hi
```

- multu instruction for unsigned multiply
  - multu reg1, reg2 #Hi,Lo=reg1*reg2

## Division

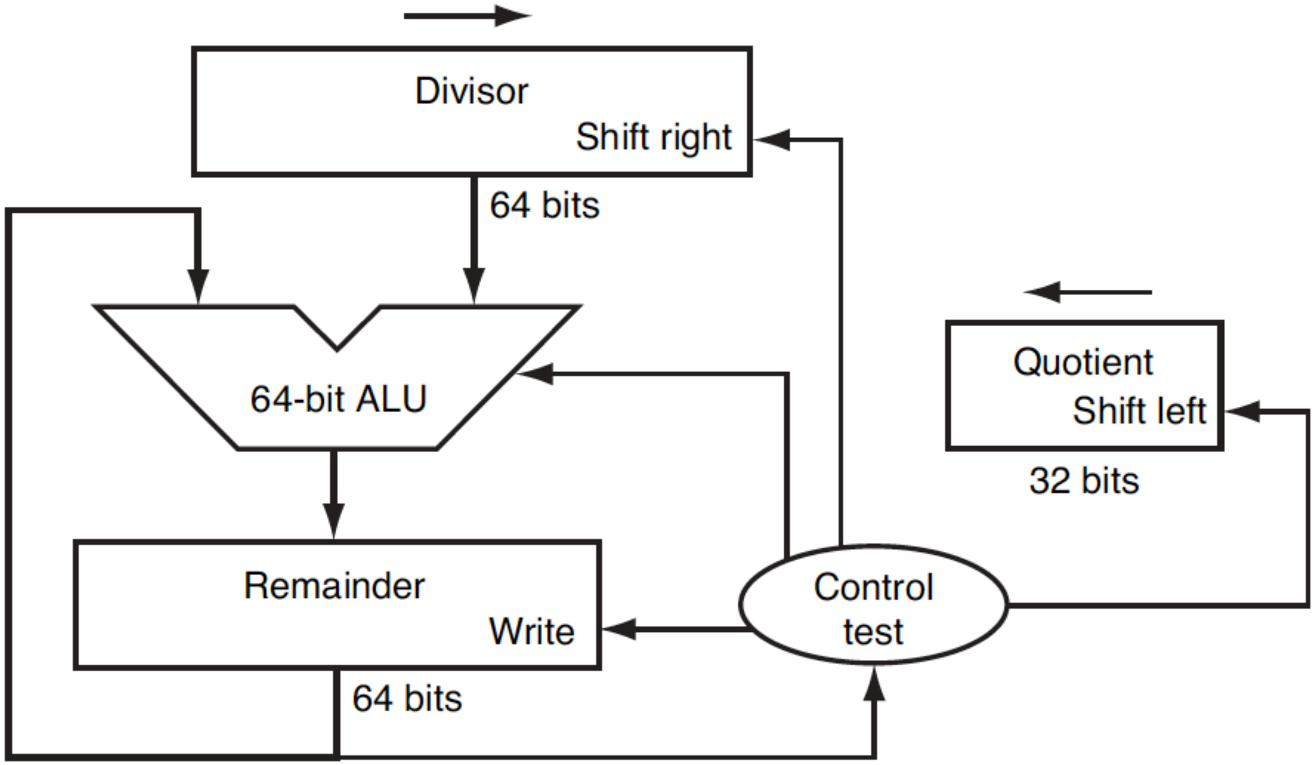
- Dividend = quotient × divisor + remainder
  - Remainder < divisor
  - Iterative subtraction
- Result:
  - Greater than 0: then we get a 1
  - Smaller than 0: then we get a 0

$$
\begin{array}{rrl}
 & 1001_{\text{ten}} & \text{Quotient} \\
\text{Divisor } 1000_{\text{ten}} & \overline{)1001010_{\text{ten}}} & \text{Dividend} \\
 & \underline{-1000\phantom{010}} & \\
 & 10\phantom{10} & \\
 & 101\phantom{0} & \\
 & 1010 & \\
 & \underline{-1000} & \\
 & 10_{\text{ten}} & \text{Remainder}
\end{array}
$$

$$
\begin{aligned}
1001010 - 1000000 &= 0001010 \\
0001010 - 0100000 &< 0 \\
0001010 - 0010000 &< 0 \\
0001010 - 0001000 &= 0000010
\end{aligned}
$$

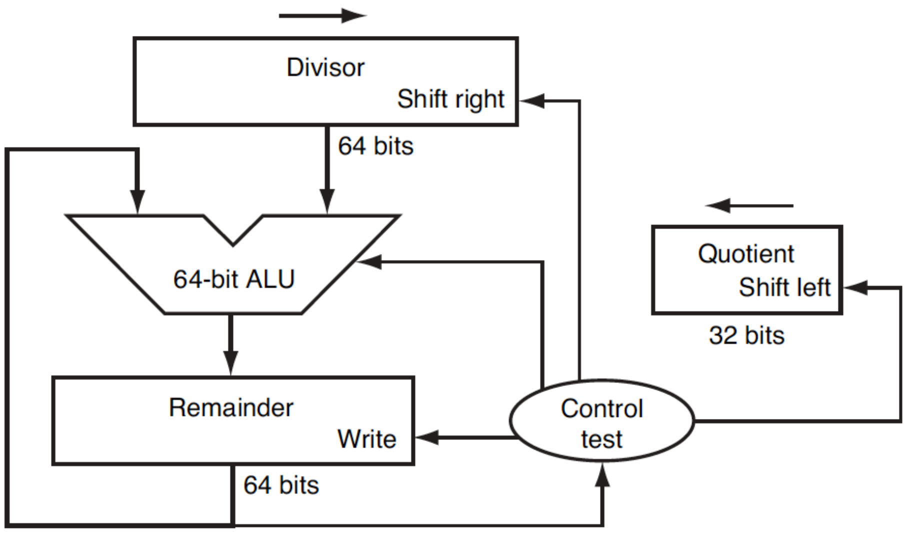
## Division V1 --Logic Diagram

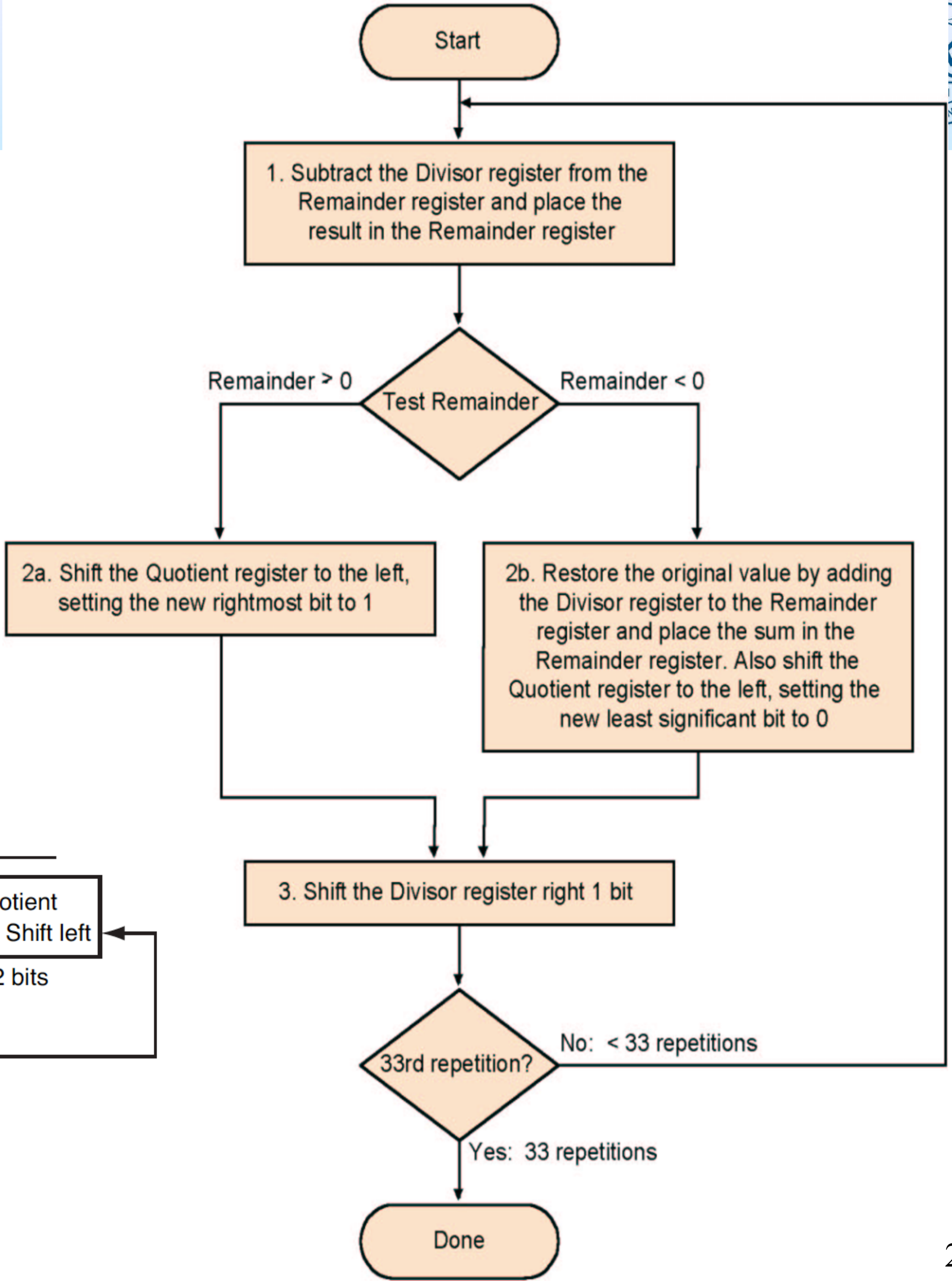
- At first, the divisor is in the left half of the divisor register, the dividend is in the right half of the remainder register.
- Shift right the divisor register each step



# Algorithm V1

- **Each step:**
  - **Subtract divisor**
  - **Depending on Remainder**
    - **if >=0, write '1'**
    - **if <0, restore and write '0'**

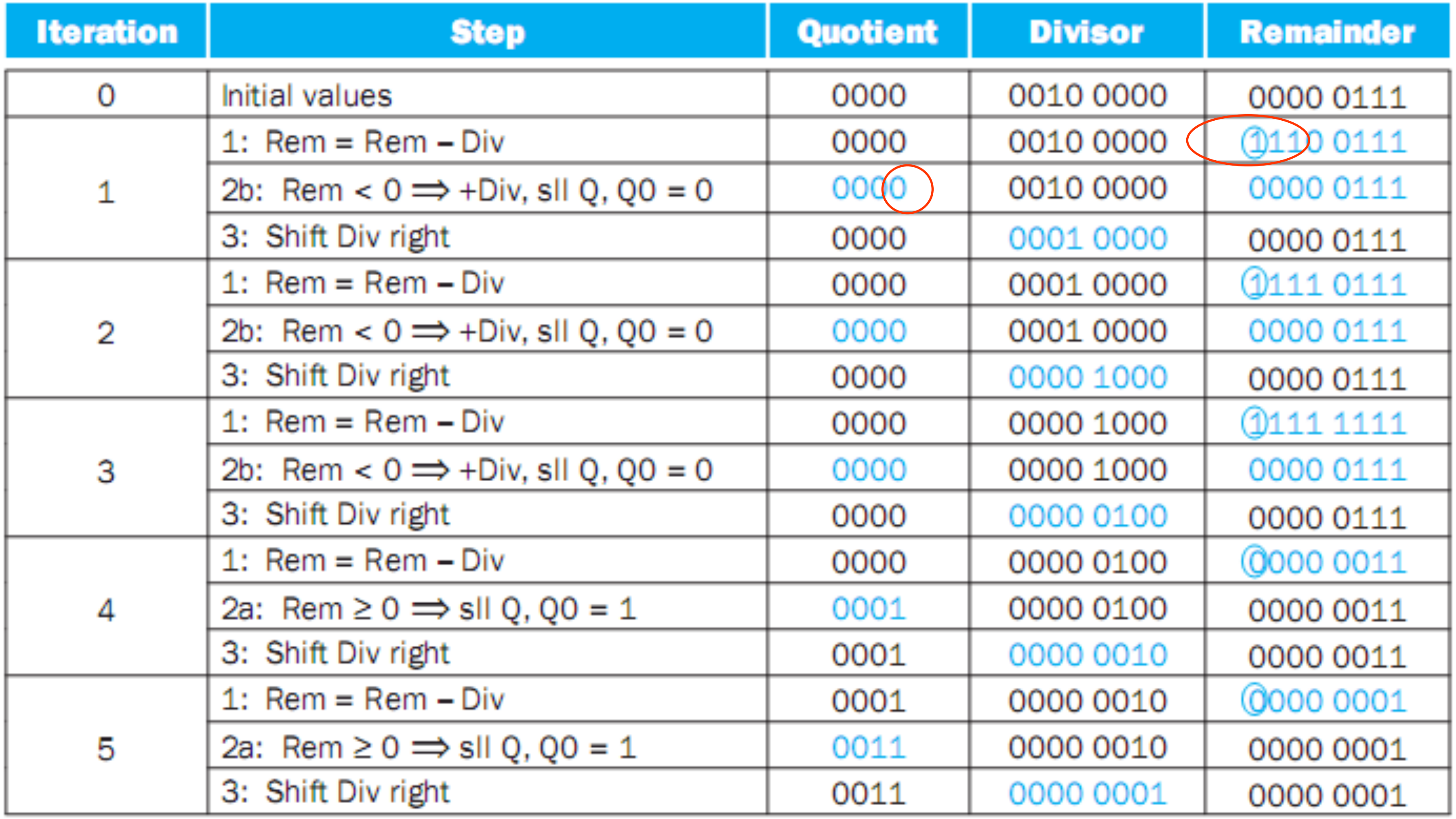




Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder > 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and place 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

# Example

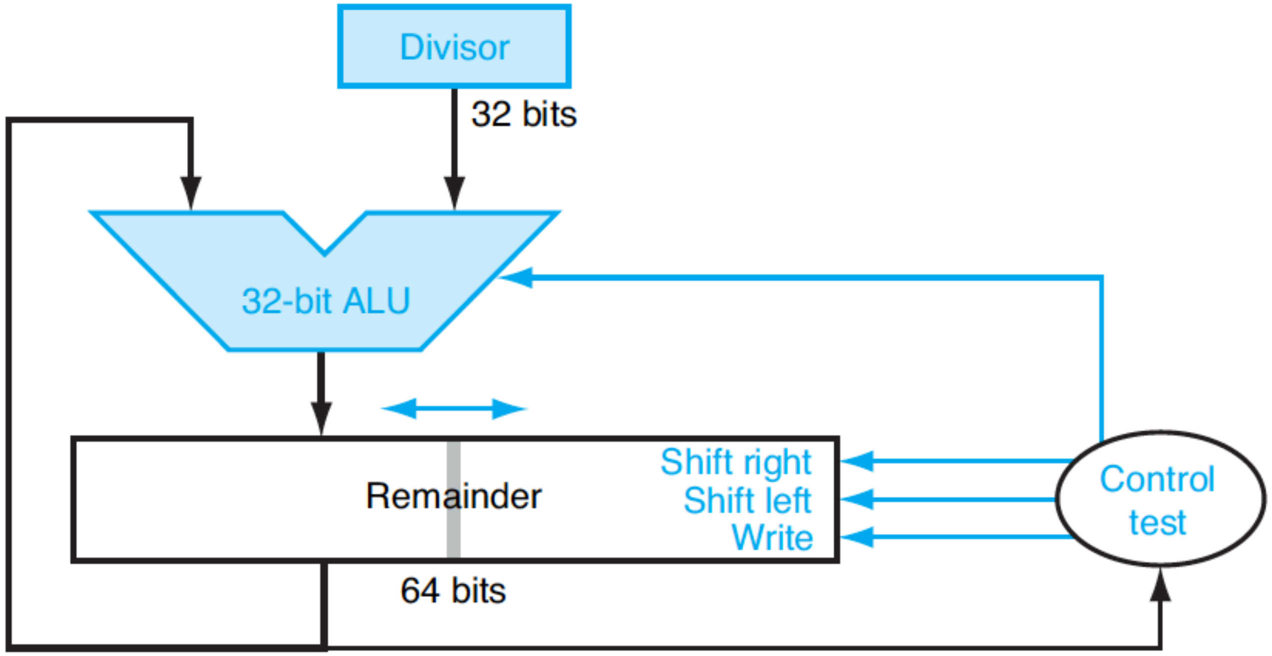
**7/2=0111/0010**



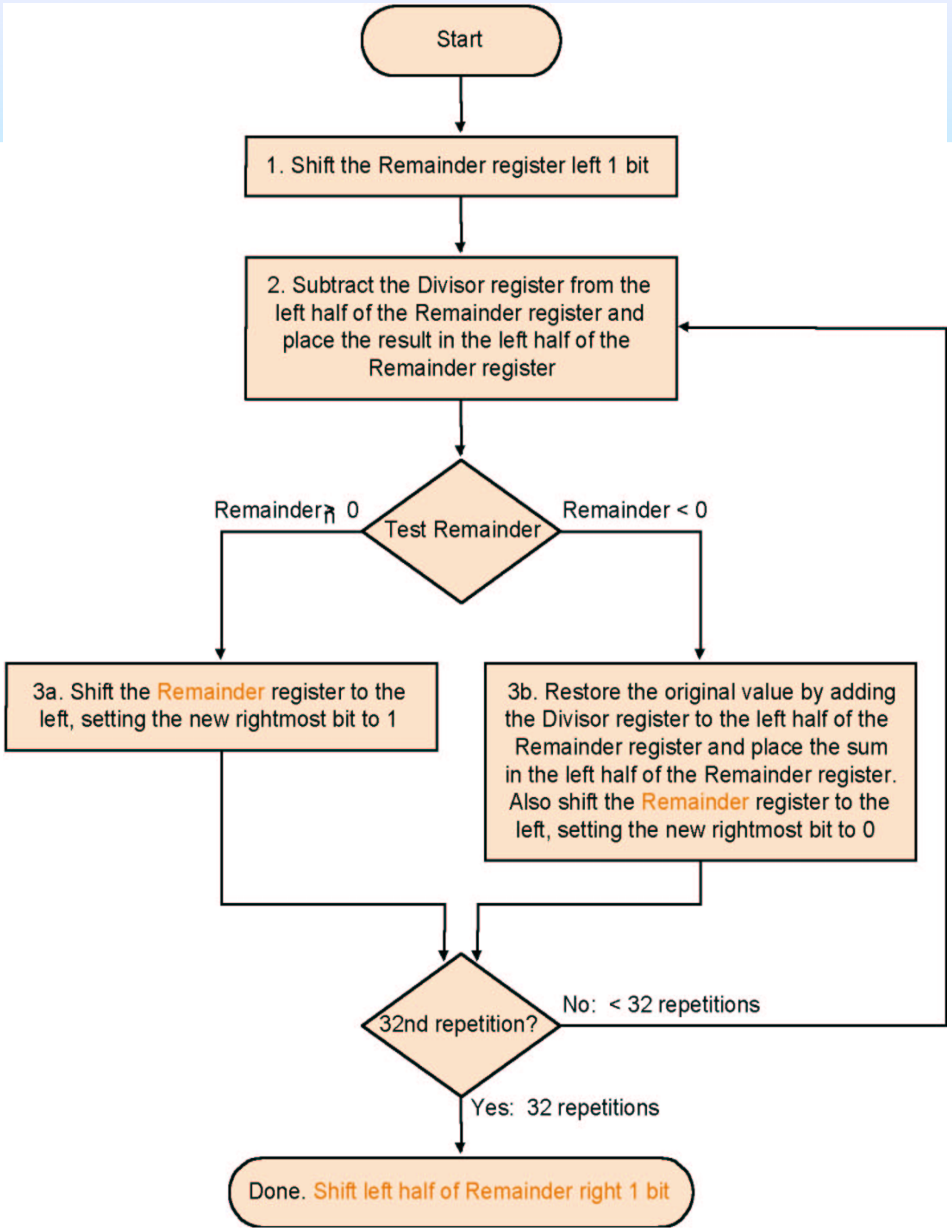
| Iteration | Step | Quotient | Divisor | Remainder |
|---|---|---|---|---|
| 0 | Initial values | 0000 | 0010 0000 | 0000 0111 |
| 1 | 1: Rem = Rem – Div | 0000 | 0010 0000 | 1110 0111 |
| | 2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0 | 0000 | 0010 0000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0001 0000 | 0000 0111 |
| 2 | 1: Rem = Rem – Div | 0000 | 0001 0000 | 1111 0111 |
| | 2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0 | 0000 | 0001 0000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0000 1000 | 0000 0111 |
| 3 | 1: Rem = Rem – Div | 0000 | 0000 1000 | 1111 1111 |
| | 2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0 | 0000 | 0000 1000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0000 0100 | 0000 0111 |
| 4 | 1: Rem = Rem – Div | 0000 | 0000 0100 | 0000 0011 |
| | 2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1 | 0001 | 0000 0100 | 0000 0011 |
| | 3: Shift Div right | 0001 | 0000 0010 | 0000 0011 |
| 5 | 1: Rem = Rem – Div | 0001 | 0000 0010 | 0000 0001 |
| | 2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1 | 0011 | 0000 0010 | 0000 0001 |
| | 3: Shift Div right | 0011 | 0000 0001 | 0000 0001 |

# Improved Division

- Remainder register keeps quotient
- Shifting the operands and quotient simultaneously with subtraction to speed up the division operation
- No quotient register required: quotient register is concatenated with the right half of the remainder register



# Algorithm



Start
1. Shift the Remainder register left 1 bit
2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
3a. Shift the Remainder register to the left, setting the new rightmost bit to 1
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done. Shift left half of Remainder right 1 bit

| Reminder Register | Divisor Register | |
|---|---|---|
| 0000 0111 | 0010 | 0000 – 0010 < 0 |
| 0000 1110 | 0010 | 0000 – 0010 < 0 |
| 0001 1100 | 0010 | 0001 – 0010 < 0 |
| 0011 1000 | 0010 | 0011 – 0010 = 0001 |
| 0011 0001 | 0010 | 0011 – 0010 = 0001 |
| 0001 0011 | | |

## Signed division

- Keep the signs in mind for Dividend and Remainder
  - (+ 7) ÷ ( + 2) = + 3 Remainder = +1
  - 7 = 3 × 2 + (+1) = 6 + 1
  - (- 7 ) ÷ (+ 2) = - 3 Remainder = -1
  - -7 = -3 × 2 + (-1) = - 6 - 1
  - (+ 7 ) ÷ ( - 2) = - 3 Remainder = +1
  - (- 7 ) ÷ ( - 2) = + 3 Remainder = -1
- What if we take – 4 × 2 + 1 = – 7 ?
  - $-(x \div y) \neq -x \div y$

## Divide in MIPS

- instruction for signed divide
  - div reg1, reg2
  - Remainder: Register Hi
  - Quotient: Register Lo
  - E.g.

```
div $s2, $s3  #Lo=$s2/$s3
              #Hi=$s2 mod $s3
mflo  $s1     # $1=Lo
mfhi  $s2     # $s2=Hi
```

- instruction for unsigned divide
  - divu $s2, $s3

```
divu $s2, $s3  #Lo=$s2/$s3
               #Hi=$s2 mod $s3
```

# Floating point numbers

- Reasoning
  - Larger number range than integer rage
  - Fractions
  - Numbers like e (2.71828) and π (3.14159265....)
- Representation (scientific notation)
  - Sign
  - Significant
  - Exponent
  - More bits for significand: more accuracy
  - More bits for exponent: increases the range

# Floating point numbers

- Form
  - Arbitrary 363.4 • $10^{34}$
  - Normalised 3.634 • $10^{36}$
- Binary notation
  - $(-1)^S \times F \times 2^E$
  - F: fraction
  - E: exponent, integer

- **Standard format IEEE 754**
  - **Single precision, $2.0_{10}\times10^{-38}\sim2.0_{10}\times10^{38}$**
  - **Double precision, $2.0_{10}\times10^{-308}\sim2.0_{10}\times10^{308}$**

Single precision

| | | |
|---|---|---|
| 31 | 30 …… 23 | 22 …… 0 |
| S | exponent | fraction |
| 1 bit | 8 bits | 23 bits |

Double precision

| | | |
|---|---|---|
| 31 | 30 …… 20 | 19 …… 0 |
| S | exponent | fraction |
| 1bit | 11 bits | 20 bits |

| | | |
|---|---|---|
| 31 | fraction (continued) | 0 |

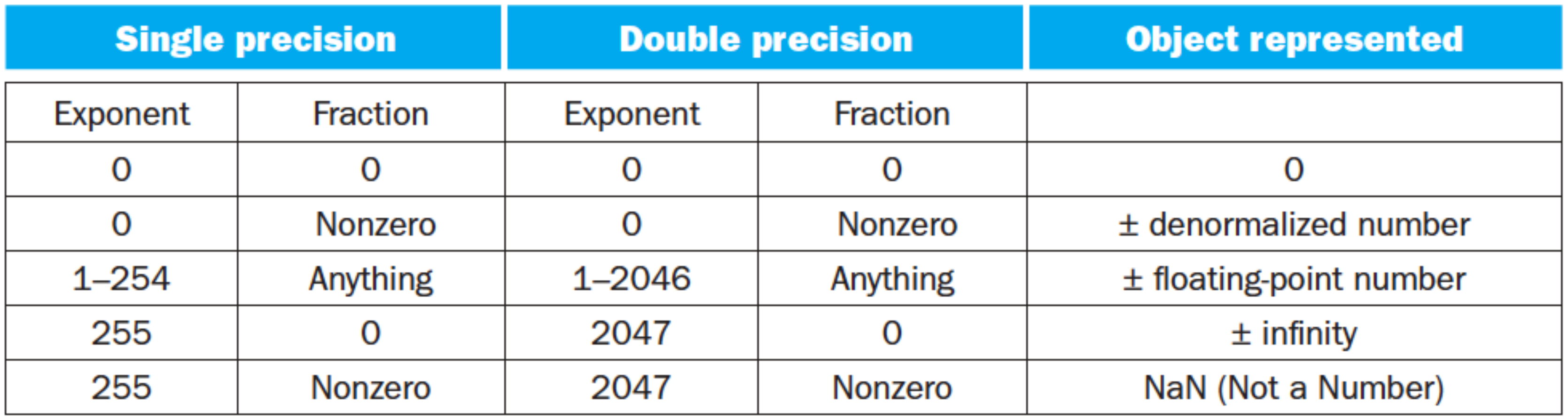
32 bits

# IEEE 754 standard

- **Leading '1' bit of significand is implicit**

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

- **IEEE 754 encoding of floating pioint numbers**



| Single precision | | Double precision | | Object represented |
|---|---|---|---|---|
| Exponent | Fraction | Exponent | Fraction | |
| 0 | 0 | 0 | 0 | 0 |
| 0 | Nonzero | 0 | Nonzero | ± denormalized number |
| 1–254 | Anything | 1–2046 | Anything | ± floating-point number |
| 255 | 0 | 2047 | 0 | ± infinity |
| 255 | Nonzero | 2047 | Nonzero | NaN (Not a Number) |

- Floating-point representation could be easily processed by integer comparisons (e.g., sorting)
  - The sign is put in the most significant bit
  - Exponent is placed before the significant
- Challenges brought by negative exponents
  - $1.0_{two} \times 2^{-1}$

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | . | . | . |

  - $1.0_{two} \times 2^{1}$

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | . | . | . |

## IEEE 754 standard

- Exponent is biased

  00...000 smallest exponent

  11...111 biggest exponent

  - Bias 127 for single precision
  - Bias 1023 for double precision
- Summary:

$$(-1)^{\text{sign}} \bullet (1 + \text{Fraction}) \bullet 2^{\text{exponent - bias}}$$

# Example

- Show the binary representation of -0.75 in IEEE single precision
- Decimal representation: $-0.75_{ten} = -\ 3/4_{ten} = -\ 3/2^2_{ten}$
- Binary representation: $-11_{two}/2^2_{ten} = -0.11_{two} = -\ 1.1_{two} \bullet 2^{-1}$
- Floating point
  - $(-1)^{sign} \bullet (1 + \text{fraction}) \bullet 2^{\text{exponent - 127}}$
    - $(-1)^{sign}$ =-1,so Sign = 1
    - 1+ fraction = 1.1,so fraction=.1000 0000 0000 0000 0000 000
    - Exponent-127=-1 , Exponent = 126= 01111110

Single precision

| 31 | 30 …… 23 | 22 …… 0 |
|---|---|---|
| 1 | 0111 1110 | 100 0000 0000 0000 0000 0000 |
| 1 bit | 8 bits | 23 bits |

Double precision

| 31 | 30 …… 20 | 19 …… 0 |
|---|---|---|
| 1 | 011 1111 1110 | 1000 0000 0000 0000 0000 |
| 1bit | 11 bits | 20 bits |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |

- **Show the binary representation of 3.14 in IEEE single precision**

**3.14=11.001000111101**
**=1.1001000111101 • $2^{1}$**

$$(-1)^{\text{sign}} \bullet (1 + \text{fraction}) \bullet 2^{\text{exponent} - 127}$$

**Sign=0**
**Fraction=1001000111101**
**Exponent=10000000**

**Machine code=0 10000000 100100011110…**

# Converting Binary to Decimal Floating Point

| 31 | 30 …… 23 | 22 …… 0 |
|---|---|---|
| 1 | 1000 0001 | 010 0000 0000 0000 0000 0000 |
| 1 bit | 8 bits | 23 bits |

$$
\begin{aligned}
(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\
&= -1 \times 1.25 \times 2^2 \\
&= -1.25 \times 4 \\
&= -5.0
\end{aligned}
$$

## Limitations

- **Overflow:**
  **The number is too big to be represented**
- **Underflow:**
  **The number is too small to be represented**

## Floating point addition

- Alignment
- The proper digits have to be added
- Addition of significand
- Normalization of the result
- Rounding
- Example in decimal

  system precision 4 digits

  What is $9.999 \cdot 10^{1} + 1.610 \cdot 10^{-1}$ ?

## Example for Decimal

- Aligning the two numbers

  $9.999 \bullet 10^1$

  $0.01610 \bullet 10^1 \rightarrow 0.016 \bullet 10^1$ Truncation

- Addition

  $$\begin{array}{rl} 9.999 & \bullet 10^1 \\ \underline{+\ 0.016} & \underline{\bullet 10^1} \\ 10.015 & \bullet 10^1 \end{array}$$

- Normalisation

  $1.0015 \bullet 10^2$

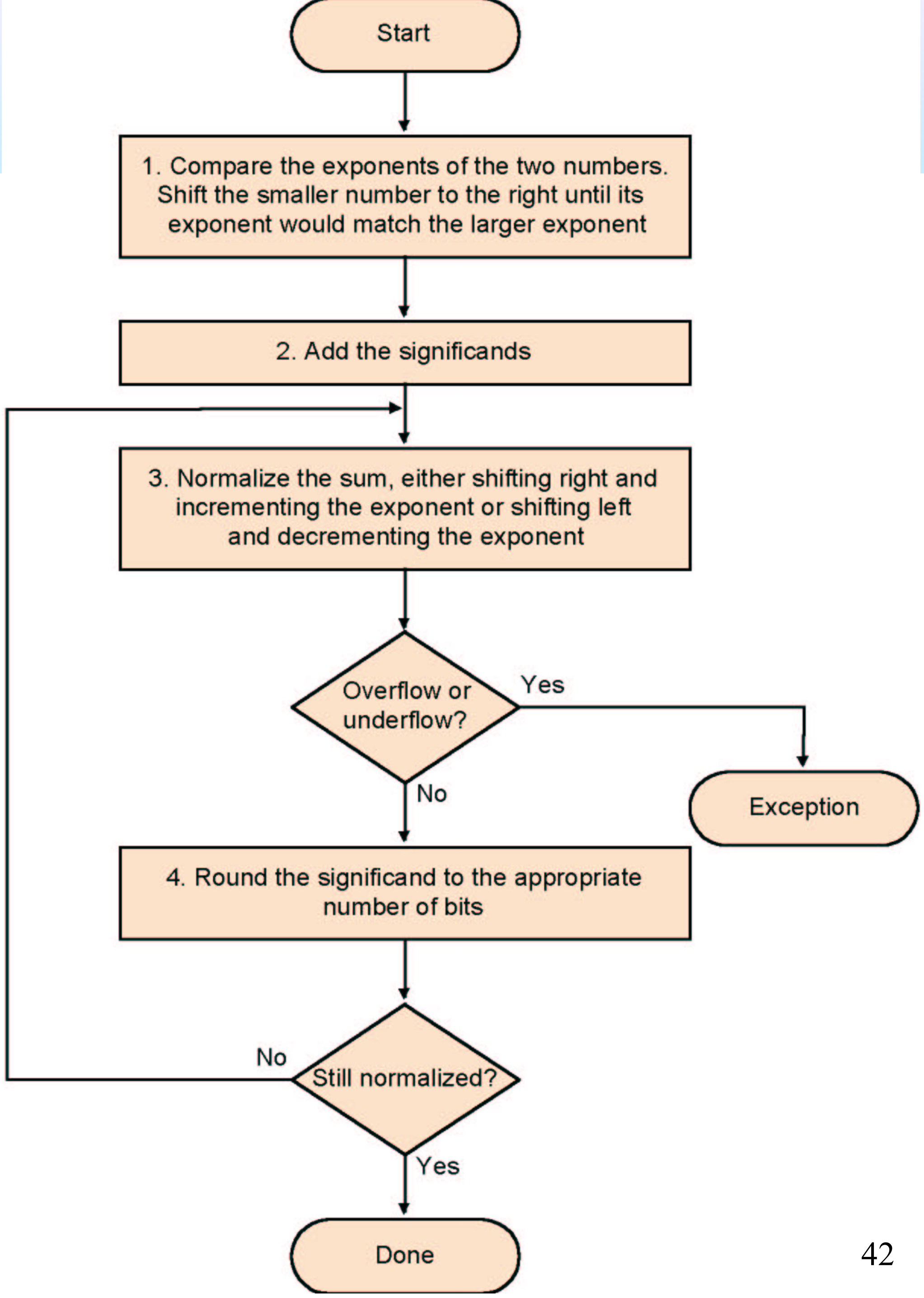
- Rounding

  $1.002 \bullet 10^2$

## Algorithm

- Normalize Significand
- Add Significand
- Normalize the sum
- Over/underflow
- Rounding
- Normalization

Ex. y=0.5+(-0.4375)



Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

# Example y=0.5+(-0.4375) in binary

- $0.5_{10} = 1.000_2 \times 2^{-1}$
- $-0.4375_{10} = -1.110_2 \times 2^{-2}$
- Step1: The fraction with lesser exponent is shifted right until matches

  $-1.110_2 \times 2^{-2} \rightarrow -0.111_2 \times 2^{-1}$
- Step2: Add the significands

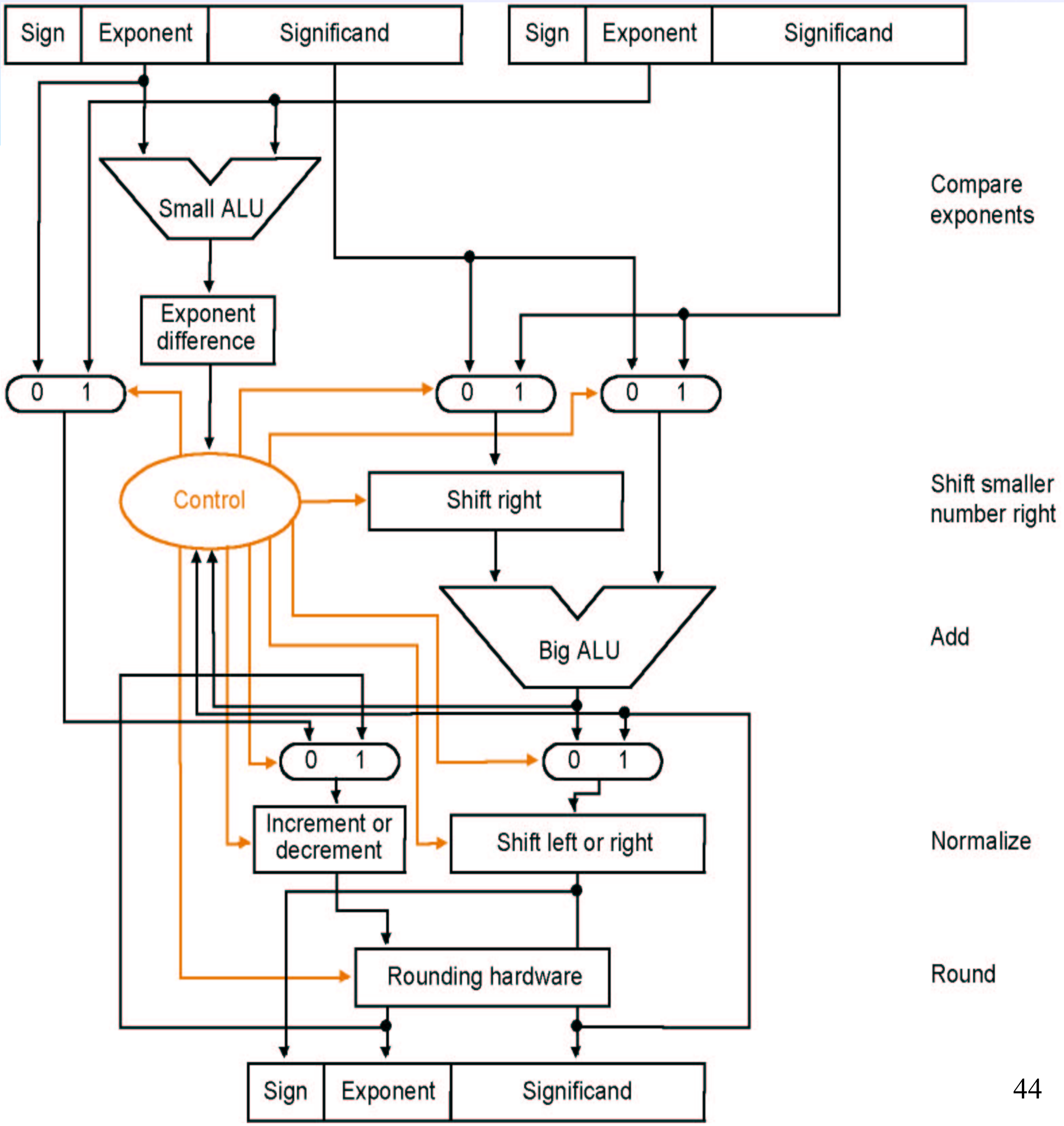
$$\begin{array}{r} 1.000_2 \times 2^{-1} \\ +)\ -0.111_2 \times 2^{-1} \\ \hline 0.001_2 \times 2^{-1} \end{array}$$

- Step3: Normalize the sum and checking for overflow or underflow

  $0.001_2 \times 2^{-1} \rightarrow 0.010_2 \times 2^{-2} \rightarrow 0.100_2 \times 2^{-3} \rightarrow 1.000_2 \times 2^{-4}$
- Step4: Round the sum

  $1.000_2 \times 2^{-4} = 0.0625_{10}$

# Algorithm



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

# Multiplication

- Composition of number from different parts

  $\rightarrow$ separate handling

  $$(s1 \cdot 2^{e1}) \cdot (s2 \cdot 2^{e2}) = (s1 \cdot s2) \cdot 2^{e1+e2}$$

- Example

  1 10000010 000 0000 0000 0000 0000 0000 = $-1 \times 2^3$

  0 10000011 000 0000 0000 0000 0000 0000 = $1 \times 2^4$

- Both significands are 1 $\rightarrow$ product = 1 $\rightarrow$Sign=1
- Add the exponents, bias = 127

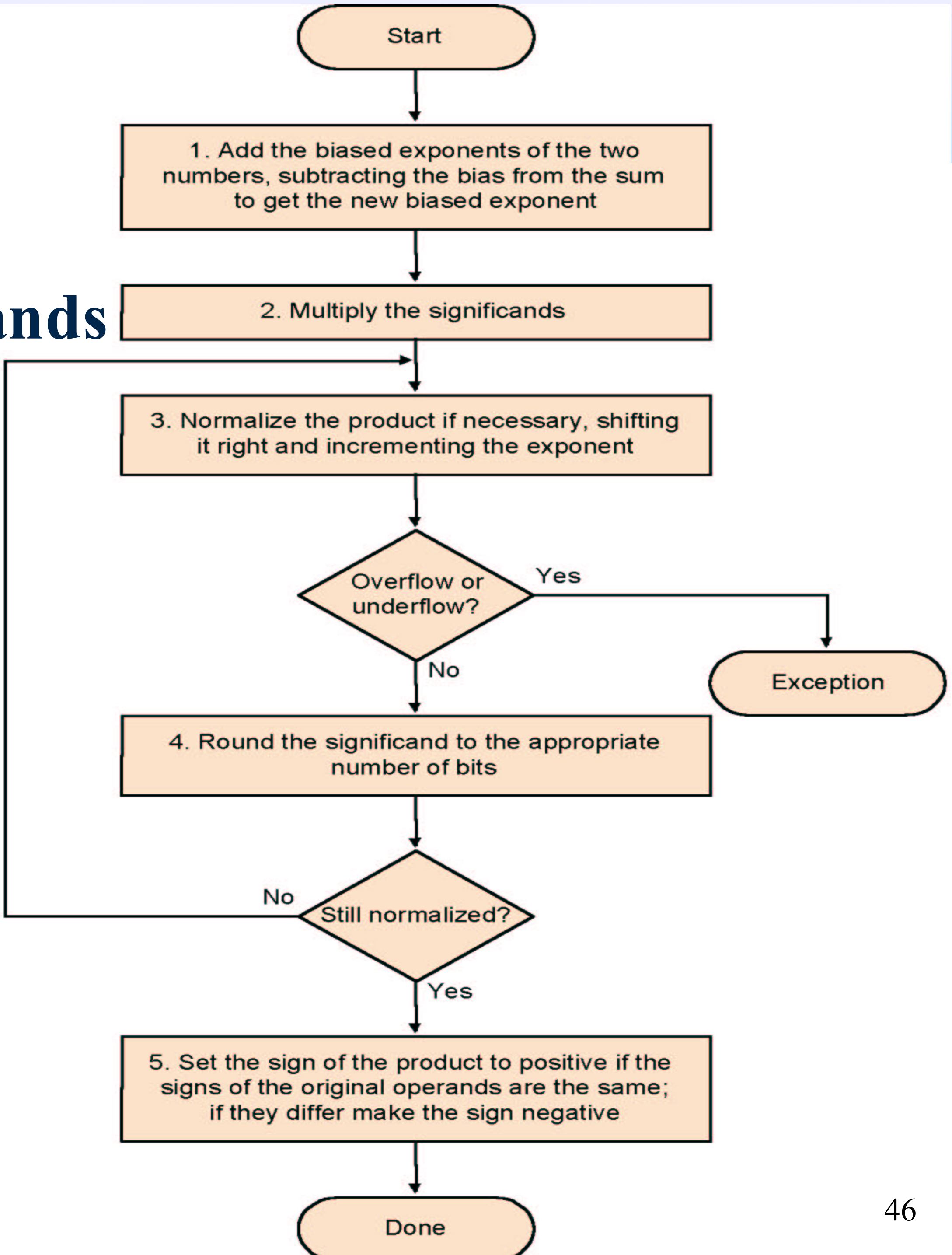
```
  10000010
 +10000011
 100000101
```

Correction: 100000101-01111111=10000110=134=127+3+4

- The result: 1 10000110 000 0000 0000 0000 0000 0000 = $-1 \times 2^7$

# Multiplication

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



Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

## Multiplying the numbers $0.5_{ten}$ and $-0.4375_{ten}$ $\rightarrow 1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$

- Step1:Adding the exponents without bias or using the biased
  - ➢ -1 + (-2)=-3
  - ➢ (-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127- 127) =-3 + 127= 124
- Step 2. Multiplying the significands
  - ➢ $1.110000_{two} x2^{-3}$
- Step 3. normalize
  - ➢ $127 \geq -3 \geq -126$, no overflow or underflow.
- Step 4. Rounding
  - ➢ $1.110_{two} x2^{-3}$
- Step 5. sign
  - ➢ $-1.110_{two} x2^{-3} =-0.21875_{ten}$

$$\begin{array}{r}
1.000_{two} \\
\times \quad 1.110_{two} \\
\hline
0000 \\
1000\phantom{0} \\
1000\phantom{00} \\
1000\phantom{000} \\
\hline
1110000_{two}
\end{array}$$

## Division-- Brief

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

## Floating point instruction in MIPS

- instruction
  - Add（single， double）： add.s， add.d
  - Sub（single， double）： sub.s， sub.d
  - Multiplication（single， double）： mul.s， mul.d
  - division（single， double）： div.s， div.d
  - comparison（single， double）： c.x.s， c.x.d
    - x: eq (equal), neq (not equal), lt (less than), le (less than or equal), gt (greater than) , ge (greater than or equal)
- 32 float registers
  - $f0, $f1, $f2, … , $f31: single precision
  - <$f0,$f1>, …, <$f30,$f31>: double precision
- Load
  - lwcl
- Store
  - swcl

```
lwcl    $f4, x($sp)          # load 32-bit F.P. number into f4
lwcl    $f6, y($sp)          # load 32-bit F.P. number into f6
add.s   $f2, $f4, $f6        # f2 = f4+f6 single pricision
swcl    $f2, z($sp)          # store 32-bit F.P. number from f2
```

## Compiling a Floating-Point C Program into MIPS Assembly Code

5.0 is stored in memory addressed by const5+$gp
9.0 is stored in memory addressed by const9+$gp
32.0 is stored in memory addressed by const32+$gp
fahr is passed in $f12 and the result should go in $f0

- C code

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

- MIPS ASM code

```
F2c:    lwcl      $f16, const5($gp)     # f16 = 5.0
        lwcl      $f18, const9($gp)     # f18 = 9.0
        div.s     $f16, $f16, $f18      # f16 = 5.0/9.0
        lwcl      $f18, const32($gp)    # f18 = 32.0
        sub.s     $f18, $f12, $f18      # f18 = fahr – 32.0
        mul.s     $f0, $f16, $f18       # f0 = (5.0/9.0)*(fahr-32.0)
        jr        $ra                   # return
```

## Compiling Floating-Point C Procedure with Two-Dimensional Matrices into MIPS

- X = X + Y * Z
  - XYZ: 32 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];
}
```

- **MIPS register**

| | |
|---|---|
| x | $a0 |
| y | $a1 |
| z | $a2 |
| i | $s0 |
| j | $s1 |
| k | $s2 |

- **Pseudoinstruction**
  - li: load a constant into a register
  - l. d and s. d: turned into a pair of data transfer instructions, lwcl or swcl, to a pair of floating point registers

- **MIPS ASM code**

```
mm:...
      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*2^5 (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*2^5 (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]
```

- **MIPS ASM code**

```
sll $t0, $s0, 5              # $t0=i*25(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

- IEEE 754 always keeps two extra bits on the right during intermediate additions, called guard and round
- Rounding with Guard Digits
  - Add $2.56_{ten} \times 10^0$ to $2.34_{ten} \times 10^2$

$$\begin{array}{r} 2.3400_{ten} \\ +\ 0.0256_{ten} \\ \hline 2.3656_{ten} \end{array}$$

  - The guard digit holds 5 and the round digit holds 6.
  - Sum=$2.37_{ten} \times 10^2$.
- Rounding without Guard Digits

$$\begin{array}{r} 2.34_{ten} \\ +\ 0.02_{ten} \\ \hline 2.36_{ten} \end{array}$$

  - Sum=$2.36_{ten} \times 10^2$

# IEEE 754 round modes

- Directed roundings
  - Round toward 0 – directed rounding towards zero (also called truncation)
  - Round toward +∞ – directed rounding towards positive infinity
  - Round toward −∞ – directed rounding towards negative infinity.
- Roundings to nearest
  - Round to nearest, ties to even – rounds to the nearest value; if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit, which occurs 50% of the time; this is the default algorithm for binary floating-point and the recommended default for decimal
  - Round to nearest, ties away from zero – rounds to the nearest value; if the number falls midway it is rounded to the nearest value above (for positive numbers) or below (for negative numbers)

| y | round down (towards −∞) | round up (towards +∞) | round towards zero | round away from zero | round to nearest |
|---|---|---|---|---|---|
| +23.67 | +23 | +24 | +23 | +24 | +24 |
| +23.50 | +23 | +24 | +23 | +24 | +23 or +24 |
| +23.35 | +23 | +24 | +23 | +24 | +23 |
| +23.00 | +23 | +23 | +23 | +23 | +23 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| −23.00 | −23 | −23 | −23 | −23 | −23 |
| −23.35 | −24 | −23 | −23 | −24 | −23 |
| −23.50 | −24 | −23 | −23 | −24 | −23 or −24 |
| −23.67 | −24 | −23 | −23 | −24 | −24 |

## ULP

- ULP
  - unit in the last place or unit of least precision
  - The number of bits in error in the least significant bits of the significant between the actual number and the number that can be represented
  - If ULP(x) is less than or equal to 1, then x + 1 > x. Otherwise, x + 1 = x.

# Sticky bit

- A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.
- allows the computer to see the difference between $0.50 \ldots 00_{ten}$ and $0.50 \ldots 01_{ten}$ when rounding.
- examples in the floating point format with guard, round and sticky bits

$$5.01_{10} \times 10^{-1} + 2.34_{10} \times 10^{2} = 0.0050_{10} \times 10^{2} + 2.34_{10} \times 10^{2} = 2.34_{10} \times 10^{2}$$

$$5.01_{10} \times 10^{-1} + 2.34_{10} \times 10^{2} = 0.00501_{10} \times 10^{2} + 2.34_{10} \times 10^{2} = 2.35_{10} \times 10^{2}$$

**1.11000000000000000000100 0 0 0**

**1.11000000000000000000100 (mantissa used, exact representation)**

**1.11000000000000000000000 1 1 0**

**1.11000000000000000000001**

**1.11000000000000000000000 0 1 0**

**1.11000000000000000000000**

**1.11000000000000000000000 1 1 1**

**1.11000000000000000000001**

**1.11000000000000000000000 0 0 1**

**1.11000000000000000000000**

**1.11000000000000000000000 1 0 0 (the “halfway” case)**
**1.11000000000000000000000 （LSB=0）**

**1.11000000000000000000001 1 0 0 (the “halfway” case)**
**1.11000000000000000000010（LSB=1）**

## Fallacies and pitfalls

- Fallacy: Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2
  - The answer is yes for unsigned numbers, but what if the number is signed?

$$-5_{10} \div 4_{10} = -1$$

$$-5_{10} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_2$$

$$\begin{aligned} -5_{10} \div 4_{10} &= 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 \\ &= -2_{10}\ ? \end{aligned}$$

## ● Pitfall: Floating-point addition is not associative

$$\begin{aligned} c + (a + b) &= -1.5_{10}\times 10^{38} + (1.5_{10}\times 10^{38} + 1.0) \\ &= -1.5_{10}\times 10^{38} + 1.5_{10}\times 10^{38} \\ &= 0.0 \end{aligned}$$

$$\begin{aligned} c + (a + b) &= (-1.5_{10}\times 10^{38} + 1.5_{10}\times 10^{38}) + 1.0 \\ &= 0.0_{10} + 1.0 \\ &= 1.0 \end{aligned}$$

- Fallacy: Parallel execution strategies that work for integer data types also work for floating-point data types.

- Pitfall: The MIPS instruction add immediate unsigned (addiu) sign-extends its 16-bit immediate field

- Fallacy: Only theoretical mathematician care about floating-point accuracy

## Further reading

- 3.6 Parallelism and Computer Arithmetic Subword Parallelism
- 3.7 Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86
- 3.8 Going Faster: Subword Parallelism and Matrix Multiply

## END