Chapter 3
Arithmetic for Computers

Numbers

- Bits are just bits
  — conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: $0...2^n-1$
- Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real numbers
  negative numbers
  e.g., no MIPS subi instruction; addi can add a negative number

Number and Base

- **Number with base b: uses b digits**
  - **Decimal number: base = 10, e.g., 108_{ten}**
  - **Binary number: base = 2, e.g., 1101100_{two}**
  - **Hexadecimal number: base = 16, e.g., $6C_{hex}$ (uses 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)**

- $(d_{n-1}\ d_{n-2} \ldots d_2\ d_0)_b$
  $= d_{n-1} * b^{n-1} + d_{n-2} * b^{n-2} + \ldots + d_1 * b^1 + d_0 * b^0$

$108_{ten} = 1 * 10^2 + 0 * 10^1 + 8 * 10^0$

$1101110_{two} = 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$

$6C_{hex} = 6 * 16^1 + C * 16^0$

Conversion to a different base

Convert 27 $_{ten}$ to a binary number.

Keep dividing by a base 2, and keep track of remainders.

2	**) 27**	
2	**) 13**	**--1**
2	**) 6**	**-- 1**
2	**) 3**	**-- 0**
2	**) 1**	**-- 1**
	0	**-- 1**

Write in bottom up order

11011_{two}

Different representations of signed numbers

- **Assume 3 bit words for simplicity**

- **Sign Magnitude: One's Complement Two's Complement**

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- **Issues: balance, number of zeros, ease of operations**

How to express negative numbers?

- Assume 3 bit words for simplicity
- <u>Sign Magnitude</u> → just set the left most digit to 1 for negative numbers.
  - 100 (=-0)
  - 101 (=-1)
  - 110 (=-2)

- <u>One's complement</u> → just invert all bits (0 -> 1, 1 -> 0)
  - 111 (=-0)
  - 110 (=-1)
  - 101 (=-2)

- <u>Two's complement</u> → invert all bits and add 1
  - 111 + 1 = 000 (=-0)
  - 110 + 1 = 111 (=-1)
  - 101 + 1 = 110 (=-2)
  - 100 + 1 = 101 (=-3)

- We will use Two's complement representation

32 bit signed numbers in MIPS (2’s complement)

- **32 bit signed numbers:**

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 0_{ten}
0000 0000 0000 0000 0000 0000 0000 0001_{two} = $+\ 1_{ten}$
0000 0000 0000 0000 0000 0000 0000 0010_{two} = $+\ 2_{ten}$
...
0111 1111 1111 1111 1111 1111 1111 1110_{two} = $+\ 2{,}147{,}483{,}646_{ten}$
0111 1111 1111 1111 1111 1111 1111 1111_{two} = $+\ 2{,}147{,}483{,}647_{ten}$ *maxint*
1000 0000 0000 0000 0000 0000 0000 0000_{two} = $-\ 2{,}147{,}483{,}648_{ten}$ *minint*
1000 0000 0000 0000 0000 0000 0000 0001_{two} = $-\ 2{,}147{,}483{,}647_{ten}$
1000 0000 0000 0000 0000 0000 0000 0010_{two} = $-\ 2{,}147{,}483{,}646_{ten}$
...
1111 1111 1111 1111 1111 1111 1111 1101_{two} = $-\ 3_{ten}$
1111 1111 1111 1111 1111 1111 1111 1110_{two} = $-\ 2_{ten}$
1111 1111 1111 1111 1111 1111 1111 1111_{two} = $-\ 1_{ten}$

- $(-x_{31} * 2^{31})+(x_{30} * 2^{30})+(x_{29} * 2^{29})+\ldots+(x_1 * 2^{1})+(x_0 * 2^0)$

Exercise

Assume 4 bit words for simplicity.

What is the decimal representation of 1010_{two} if two's complement is used?

Positive → Negative (invert and add 1)

Negative → Positive (subtract 1 and invert)

Thus,

Subtract 1 from 1010, we get 1001.

Then invert 1001, we get 0110. That is 6. So it is -6_{ten}

Unsigned Numbers

- For example, each memory address it not negative, so we should use 32 bits to store just 0 and positive numbers.
- With 32-bits, we can represent numbers from 0 to $2^{32} - 1 = 4{,}294{,}967{,}295_{ten}$

Signed vs. Unsigned Comparison

- **Example**

$s0 has

1111 1111 1111 1111 1111 1111 1111 1111 $_{two}$

And $s1 has

0000 0000 0000 0000 0000 0000 0000 0000 $_{two}$

After "slt $t0, $s0, $s1"

$t0 has: ________________

$t0 has 1

After "sltu $t1, $s0, $s1"   (sltu deals numbers in registers as unsigned)

$t1 has: ___________________

$t1 has 0

Two's Complement Operations

- **Negating a two's complement number: invert all bits and add 1**
- **Advantages of Two's complement representations:**

  1. Only one representation of zero.

  2. All negative numbers have the most significant digit (left most) of 1.

  3. Sign Extension is easy: Extend the number of bits by replicating the most significant bit (most left) into the other bits.

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

  4. Addition and Subtraction (see the next page)

Addition & Subtraction

- Just like in grade school (carry/borrow 1s)

```
   0111          0111          0110
 + 0110        - 0110        - 0101
```

- Two's complement operations easy
  - subtraction using addition of negative numbers

```
   0111
 + 1010
```

Binary Subtraction

- Consider 8-bit numbers for simplicity.
- Direct method:

7 – 6 = 1

```
  0000 0111
- 0000 0110
  0000 0001
```


- Or adding 2's complement number

A – B = A + (2's complement number of B)

7 + (-6) = 1

```
  0000 0111
+ 1111 1010

 1 0000 0001
```

(the left most bit 1 is ignored since it is beyond 8 bit)

Overflow

- **The result of an operation is too large and cannot be represented in the finite computer word:**
  - **e.g., adding two n-bit numbers does not yield an n-bit number**

```
  0111
+ 0001
  1000
```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

If words are 4 bit long,

```
  0111
+ 0110
```

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive

Operation	Operand A	Operand B	Result indicating overflow
A+B	>= 0	>= 0	< 0
A+B	< 0	< 0	>= 0
A-B	>= 0	< 0	< 0
A-B	< 0	>= 0	>= 0

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption

EPC – Exception Program Counter

it contains the address of the instruction that caused the exception.

Multiplication

- **More complicated than addition**
  - **accomplished via shifting and addition**

- **Let's look at how we multiply two positive binary numbers.**

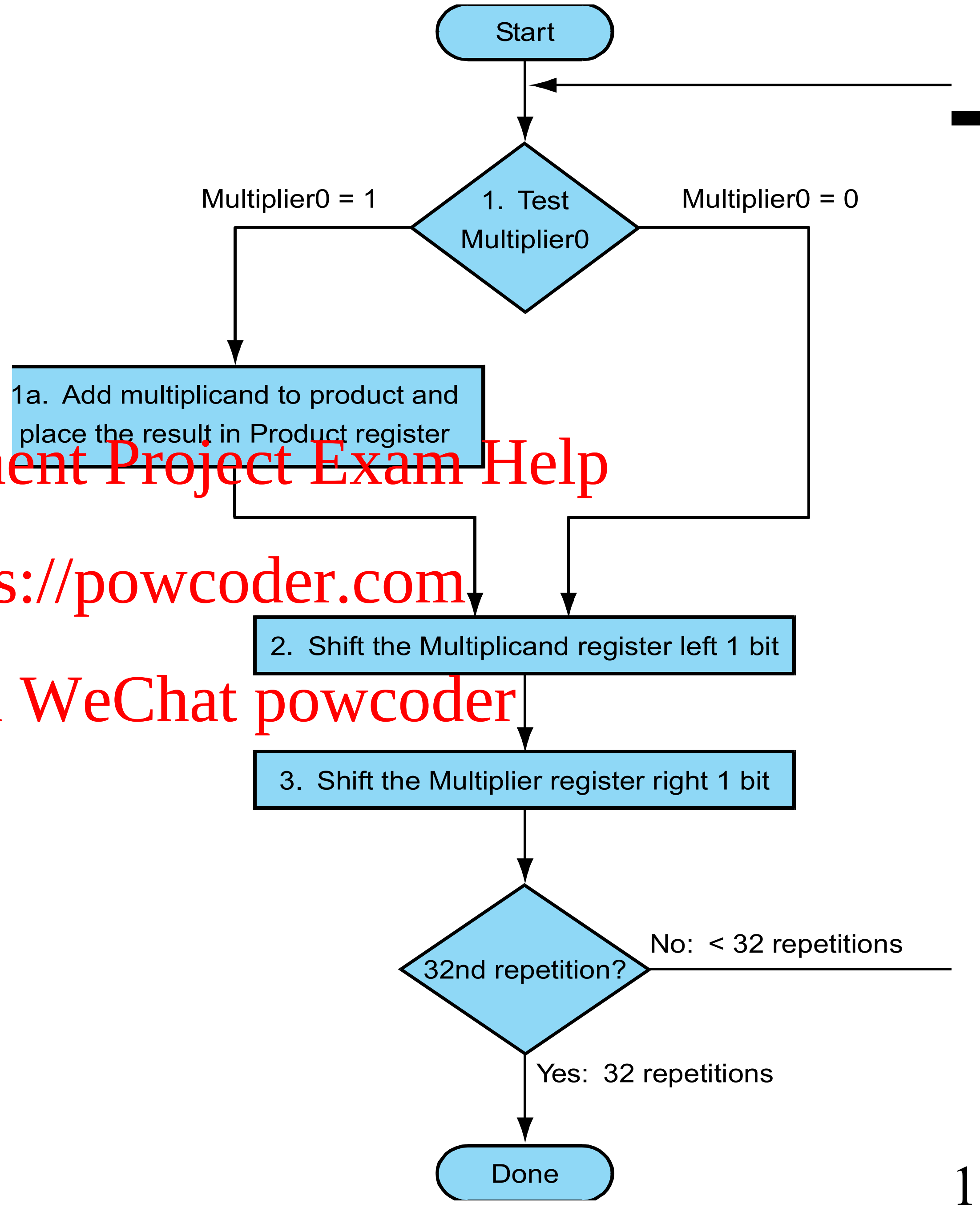

```
      0110   (multiplicand)
  x   0011   (multiplier)
```


- **Negative numbers: convert and multiply**
  - **there are better techniques, we won't look at them**

Control

We use multiplicand register,
multiplier register,
and product register.
Product register's initial value is 0.

Multiplier0 is the right most bit of the multiplier.



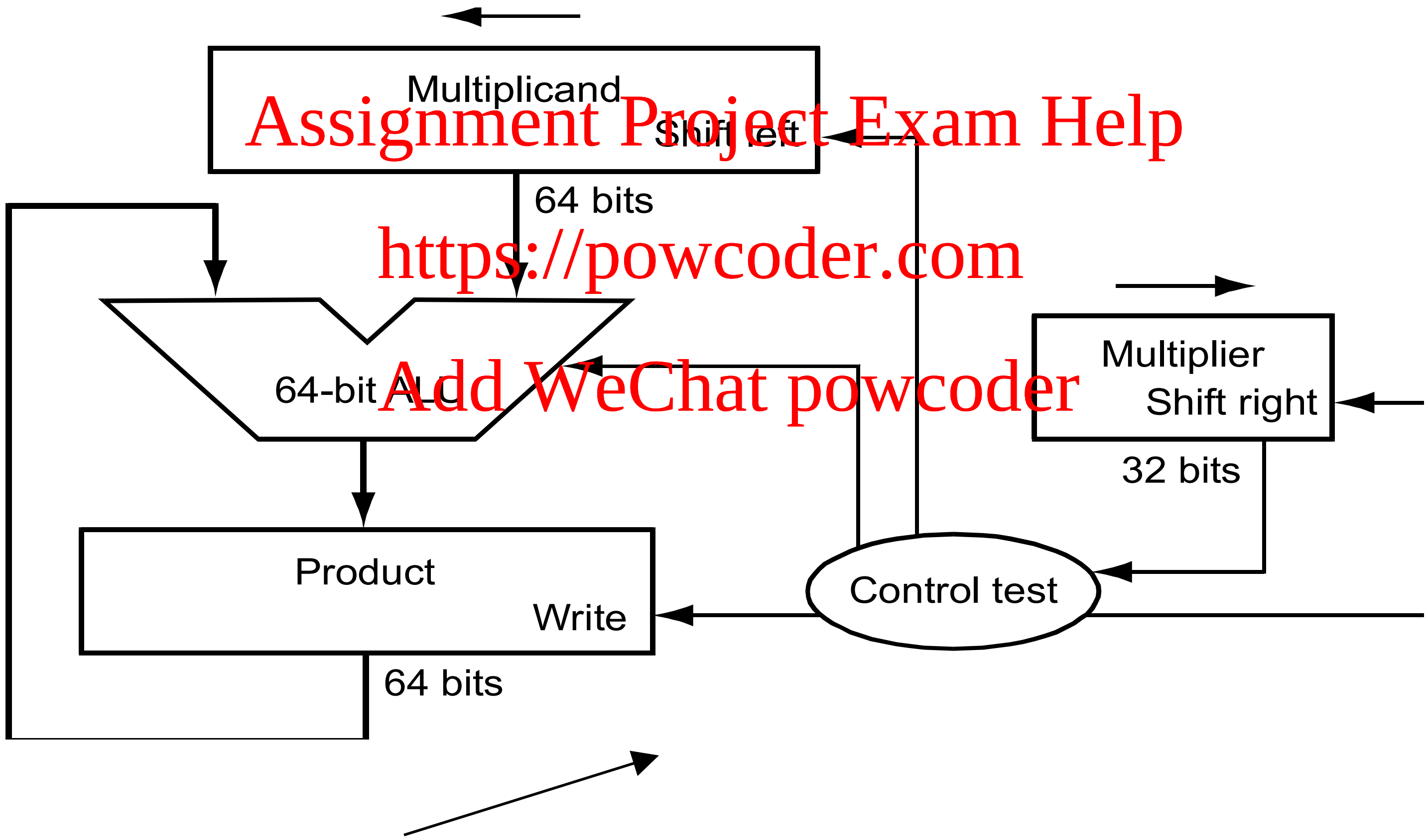
Example: multiply 0110 and 0011
(Assume 4-bit numbers instead of 32-bit numbers)

Iteration	Step	Multiplicand Register value	Multiplier Register value	Product Register value
0	Initial values	0110	0011	0
1st iteration	1a. Prod = Prod +Multiplicand 2.sll Multiplicand by 1 3. srl Multiplier by 1	0 1100	001	0+0110=0110
2nd iteration	1a. Prod = Prod +Multiplicand 2. sll Multplicand by 1 3. srl Multiplier by 1	01 1000	00	0110+01100 = 010010
3rd iteration	2. sll Multplicand by 1 3. srl Multiplier by 1	011 0000	0	010010
4th iteration	2. sll Multplicand by 1 3. srl Multiplier by 1	0110 0000	--	010010

Final Product

Multiplication Hardware

In the previous example, if we use 4-bit multiplicand, we end up 8-bit multiplicand at the end since we kept shifting it 4 times. And the product also ends up in 8 bit. Therefore, if we use 32-bit numbers, then we need 64 bits for multiplicand and product and we need a 64-bit adder to add those two numbers.



Datapath

Signed Multiplication

- **A simple way to make both numbers positive and remember whether to complement the product when finished (leave out the most significant: run for 31 bits)**

Multiply in MIPS

- mult vs. multu
- A pair of registers Hi, Lo to contain the 64 bits product

mfhi: move from Hi
mflo: move from Lo

Division

- How do we perform division on binary numbers?

```
             11      ← Quotient
Divisor 10 ) 111     ← Dividend
            - 10
               11
              -10
                1    ← Remainder
```


Dividend = Quotient x Divisor + Remainder

Dividend and Remainder should have the same sign.

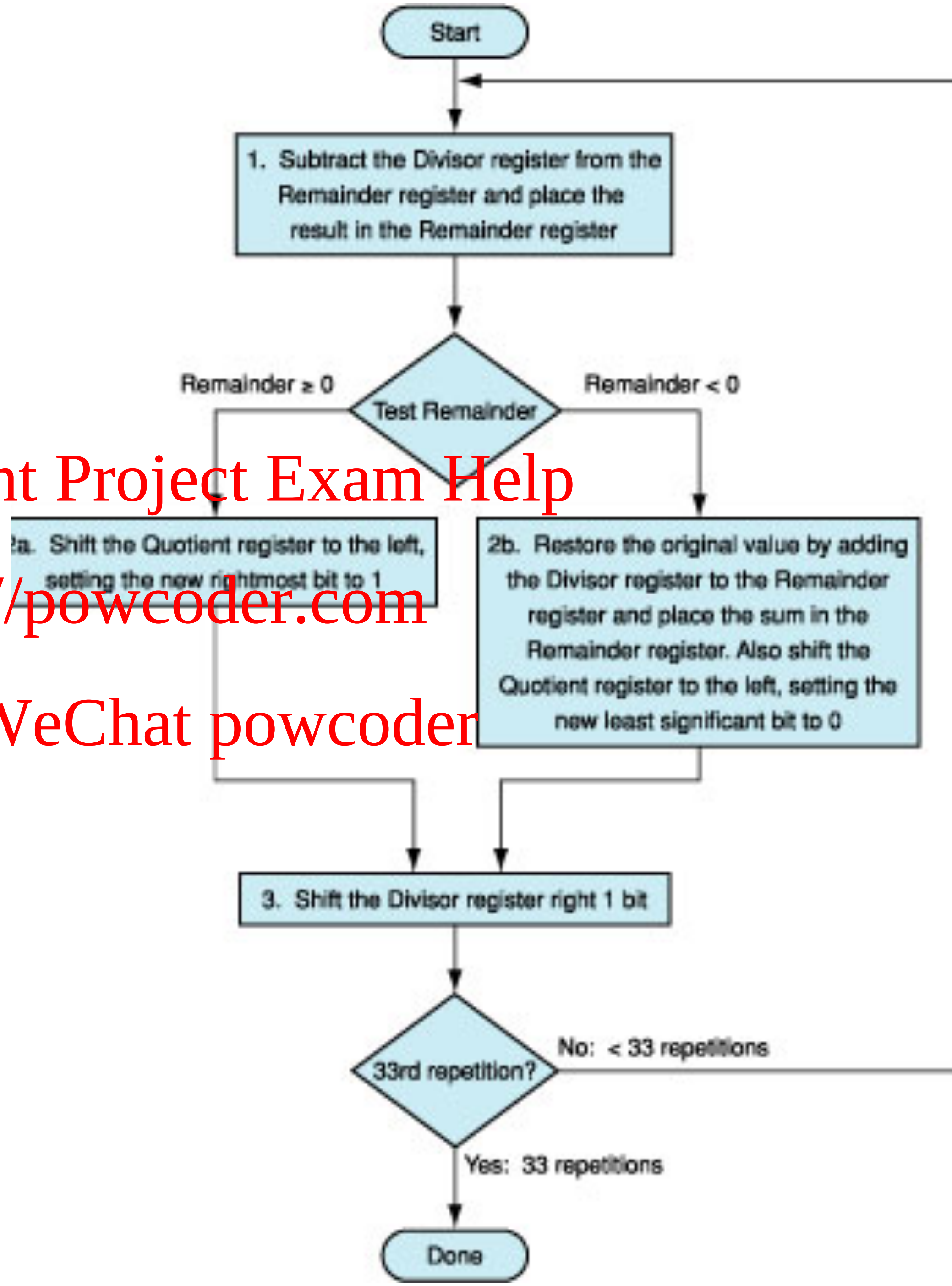
Control for Division

Remainder register is initialized to the Dividend value.

Divisor is shifted to left so that the left most digit is aligned with the left most digit of the dividend.

To decide whether divisor can be subtracted from remainder, we need to subtract and check if the result is negative or positive.

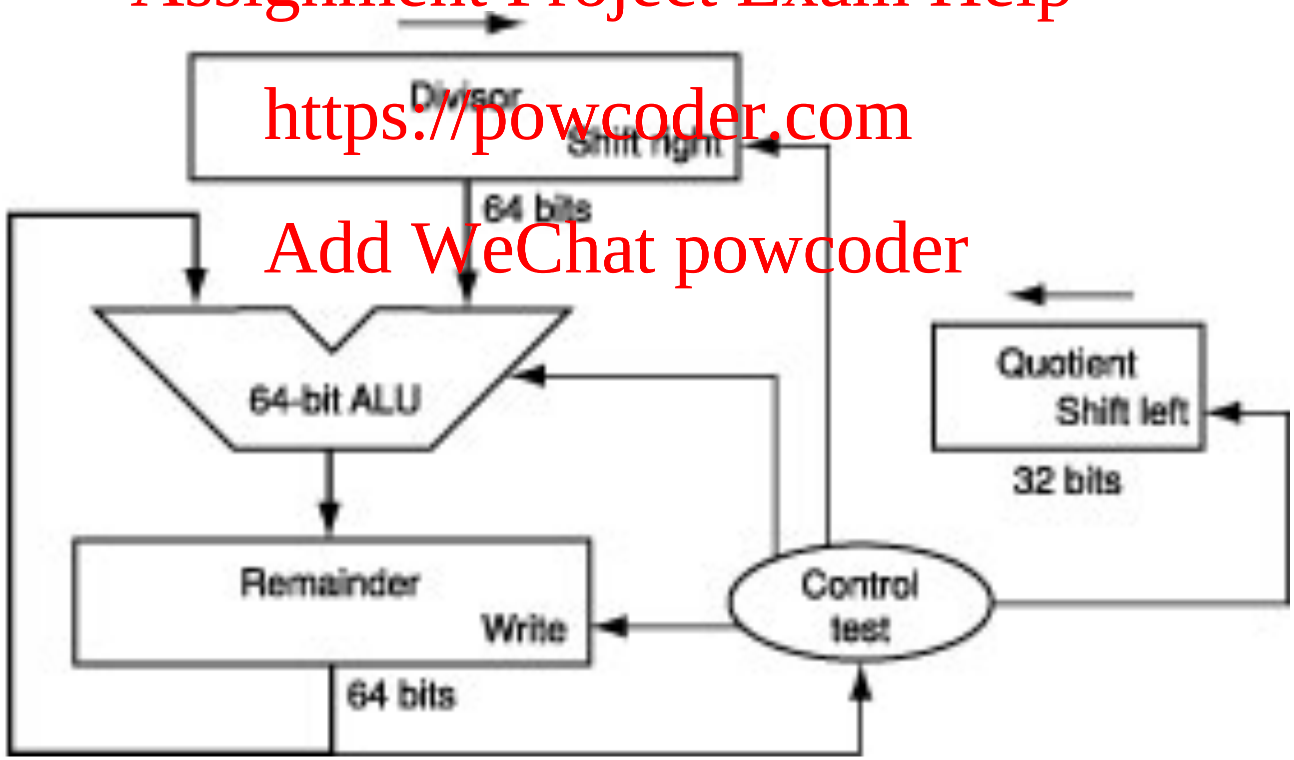
If we cannot subtract, we need to add the subtracted value back to the remainder.



Example: Divide 0000 0111 by 0010

Iteration	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0010 0000 (sll to be 8 bit)	0000 0111 (=Dividend)
1	1.Rem=Rem-Div 2b. Rem<0, Rem+=Div, sll Q, Q0=0 3. srl Div	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	1. Rem=Rem-Div 2b. Rem<0, Rem+=Div, sll Q, Q0=0 3. srl Div	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	1. Rem=Rem-Div 2b. Rem<0, Rem+=Div, sll Q, Q0=0 3. srl Div	0000 0000 0000	0000 1000 0000 1000 0000 0100	1111 1111 0000 0111 0000 0111
4	1. Rem=Rem-Div 2a. Rem>=0, sll Q, Q0=1 3. srl Div	0001 0001 0001	0000 0100 0000 0100 0000 0010	0000 0011 0000 0011 0000 0011
5	1. Rem=Rem-Div 2a. Rem>=0, sll Q, Q0=1 3. srl Div	0001 0011 0011	0000 0010 0000 0010 0000 0001	0000 0001 0000 0001 0000 0001

- W need to move the divisor to the right one digit each time, so we start with the divisor place in the left half of the 64-bit Divisor register and shift it right to w bit each step to align it with the dividend. Divisor and Remainder registers are 64-bit long, and we also need to use 64-bit ALU.



Division in MIPS

- div
- Registers Hi contains remainder, Lo contains the quotient.

mfhi: move from Hi

mflo: move from Lo

Fraction

- 0.011_{two} = ________________________ $_{ten}$?

$$= (0 * 2^{-1}) + (1 * 2^{-2}) + (1 * 2^{-3}) = 0.25 + 0.125 = 0.325_{ten}$$

$$(0.d_1 d_2 d_3 \ldots.)_b$$
$$= (d_1 * b^{-1}) + (d_2 * b^{-2}) + (d_3 * b^{-3}) + \ldots.$$

Converting fractional decimal numbers to binary

- 0.375_{ten} = ______________ $_{two}$?

  $= 0.25 + 0.125 = 2^{-2} + 2^{-3} = 0.011_{two}$

- 0.1_{ten} = ________________ $_{two}$?

  = 0.0625+0.03125+…..

When does a conversion result in a terminating expansion?

Floating Point

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576×10^9

- Scientific notation: a single digit to the left of the decimal point

- Normalized number: no leading 0:

  $1.0 \times 10^{-9} = 0.1 \times 10^{-8} = 10.0 \times 10^{-10}$

Floating Point

- Representation:
  - sign, exponent, significand: $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$

    $-0.11_{\text{two}} = (-1) * 11 * 2^{-2}$

  - more bits for significand gives more accuracy
  - more bits for exponent increases range

One way:

sign	exponent	significand
1 bit	8 bits	23 bits
1	1111 1110	0000 0000 0000 0000 0000 011

(this is not the way MIPS represents floating point.)

Floating Point

- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

IEEE 754 floating-point standard

- Leading "1" bit of significand is implicit
- Exponent is "biased":
  - A number with all 0s is smallest exponent , a number with all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{sign} \times (1+significand) \times 2^{exponent - bias}$
- Example:
  - decimal: -.75 = - (½ + ¼)
  - binary: $-.11 = -1.1 \times 2^{-1}$
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: `10111111010000000000000000000000`

Biasing

[actual (unbiased) exponent] = [exponent written] – bias

(i.e., biased exponent)

Given

sign	exponent written	fraction
1 bit	8 bits	23 bits

The number represented is

$$(-1)^{\text{sign}} \times (1+\text{fraction}) \times 2^{\text{biased exponent} - \text{bias}}$$

'significand' = 1 + 'fraction'

Biasing

Why do we have the exponent field before fraction field?

-It simplifies sorting of floating point numbers using integer comparison instructions, since numbers with biggest exponents look larger than numbers with smaller exponents, as long as both exponents have the same sign.

Negative exponents pose a challenge to simplified sorting.

-biasing solves this problem, by shifting the exponent by the bias 127.

Example

- $(-1)^{\text{sign}} \times (1+\text{fraction}) \times 2^{\text{biased exponent} - \text{bias}}$

$-0.75_{\text{ten}} = -(0.5 + 0.25)_{\text{ten}} = -0.11_{\text{two}}$

$-0.11_{\text{two}} = -1.1 \times 2^{-1}$ (in normalized scientific notation)

Sign: 1 since $(-1)^1 = -1$

Fraction: 0.1_{two}

Biased Exponent: 126 (= exponent + bias, where exponent = -1)

Bias: 127

sign	exponent written	fraction
1 bit	8 bits	23 bits

1	01111110	10000000000000000000000
1	126	.1

Double Precision

sign	exponent written	fraction
1 bit	11 bits	20bits

20+32 bits = 52 bits

Fraction (continued)
32 bits

Bias = 1023

Underflow: A negative component is too large to fit in the exponent field.

What is the range of exponents that can be represented by IEEE 754 encoding in single precision? In double precision?

Single Precision

Almost as small as $2.0_{ten} \times 10^{-38}$ and almost as large as $2.0_{ten} \times 10^{38}$

Double Precision

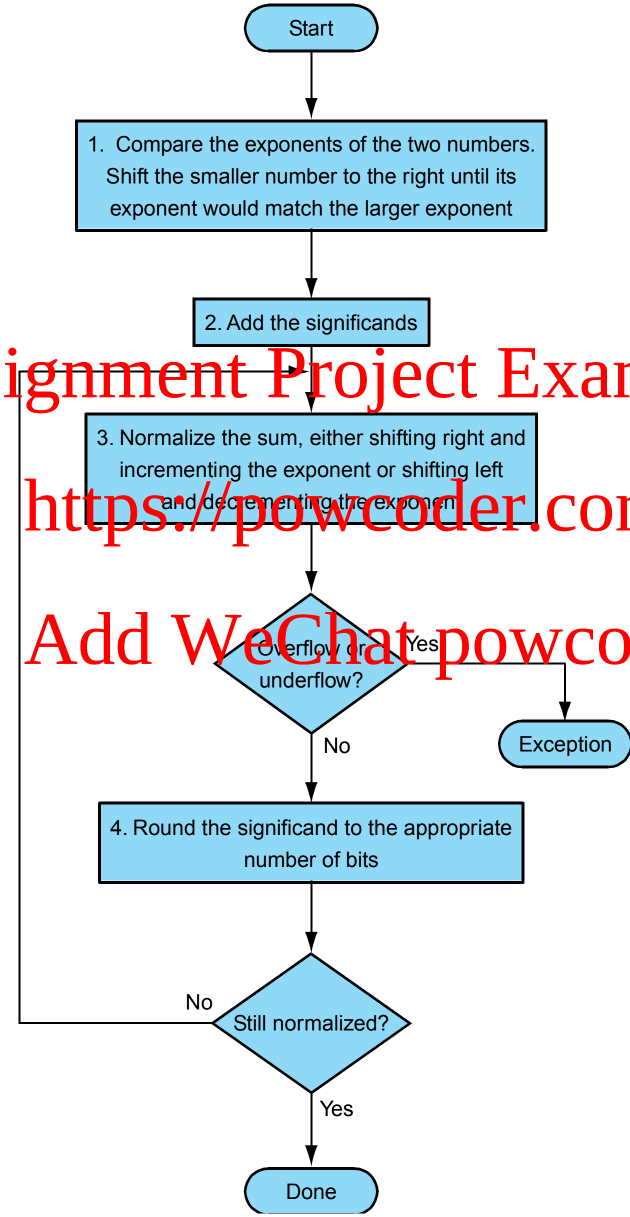
Almost as small as $2.0_{ten} \times 10^{-308}$ and almost as large as $2.0_{ten} \times 10^{308}$

- **How to represent 0, +/- infinity, NaN?**

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 addition

•



Example

- **Assume 4 bits precision: Add 0.5_{ten} and $(-0.4375)_{ten}$**

$0.5_{ten} = ½_{ten} = 2^{-1} = 0.1_{two} = 1.000_{two} \times 2^{-1}$

$-0.4375_{ten} = -7/16_{ten} = -7/2^4 = -111_{two} \times 2^{-4}$

$= -0.0111_{two} = -1.110_{two} \times 2^{-2}$

Step1: The significand of the number with the lesser exponent is shifted right until its exponent matches the larger number:

$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$

Step2: Add the significands:

$(1.000_{two} \times 2^{-1}) + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$

Step3: Normalize the sum, checking for overflow or underflow:

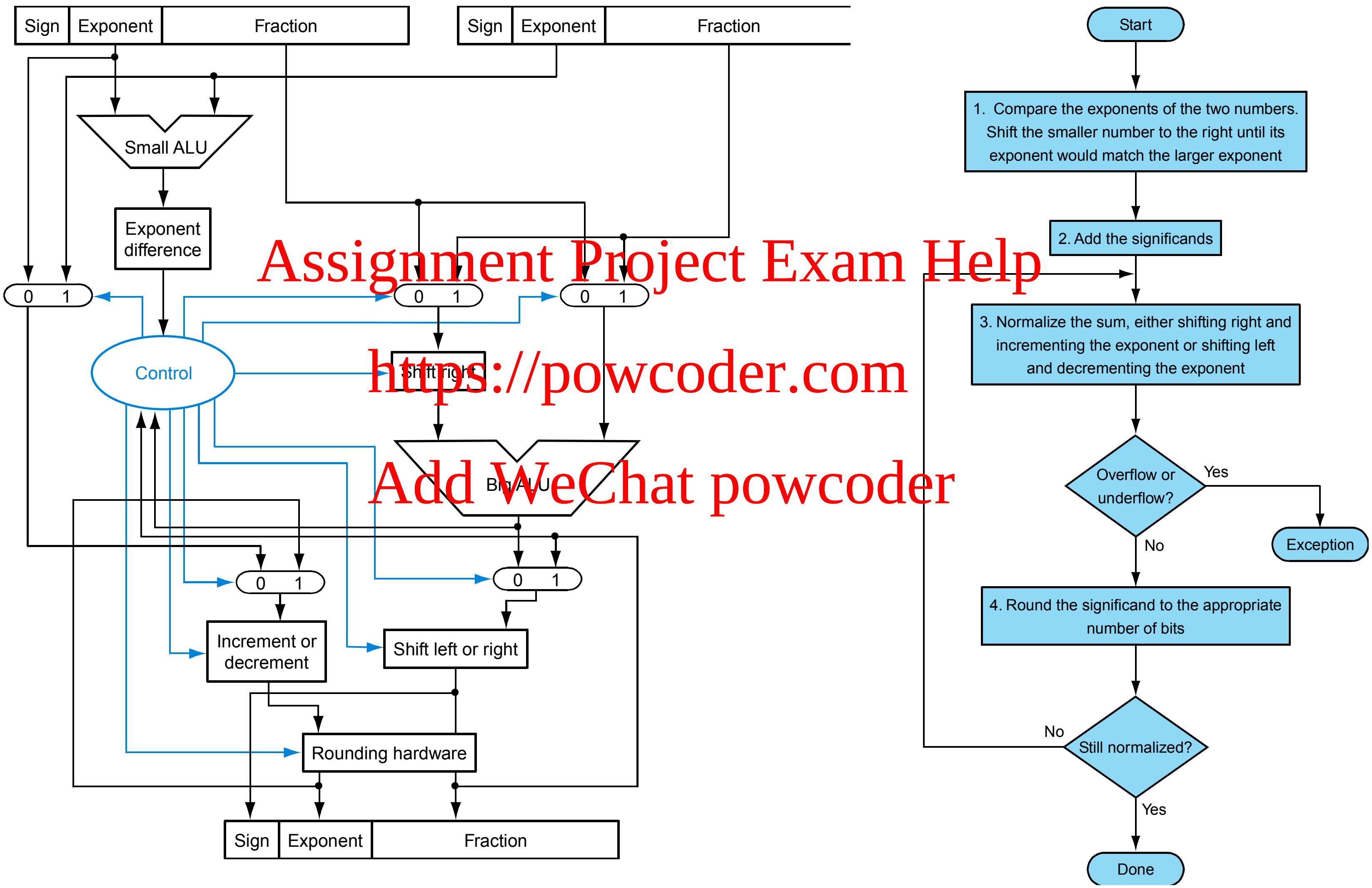
$0.001_{two} \times 2^{-1} = 1.000_{two} \times 2^{-4}$

Since 127 >= -4 >= -126, there is no overflow or underflow.

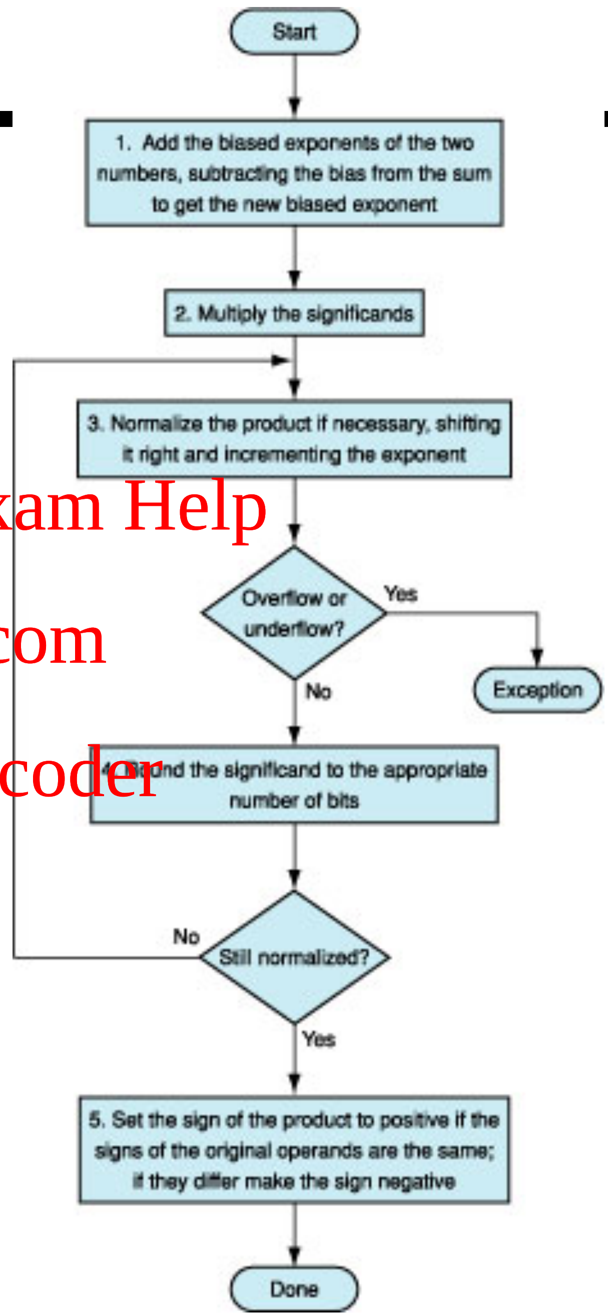
Step4: Round the sum. In this case, it already fits in 4 bits.

$1.000_{two} \times 2^{-4} = 0.0625_{ten}$

Floating point addition

- 

Floating point multiplication



Example

- Assume 4 bits precision: Multiply 0.5_{ten} and -0.4375_{ten}

$0.5_{ten} = 1.000_{two} \times 2^{-1}$

$-0.4375_{ten} = = -1.110_{two} \times 2^{-2}$

Step 1: Adding the exponents without bias:

$-1 + (-2) = -3$

biased representation: $-3 + 127 = 124$

Step2: Multiplying the significands:

$1.000 \times 1.110 = 0000+10000+100000+1000000=111000_{two}$

The product is 1.110000×2^{-3}

Step3: Normalize it and check for overflow or underflow.

127 >= -3 >= -126, there is no overflow or underflow

Step 4: Rounding the product.

$1.110000 \times 2^{-3} = 1.110 \times 2^{-3}$

Step5: Since the signs of the original operands differ, make the sign of the product negative. Hence the product is: $-1.110 \times 2^{-3} = -0.21875_{ten}$

Floating point instructions in MIPS

- green sheet

- add.s – floating point addition single,

  add.s \$f2, \$f4, \$f6   #\$f2 = \$f4+\$f6
- sub.s

  sub.s \$f2, \$f4, \$f6   #\$f2 = \$f4-\$f6
- mul.s
- div.s
- add.d – floating point addition double
- sub.d, ……

32 floating point registers:

\$f0, \$f1, \$f2, …., \$f31

MIPS floating point registers are used in pairs for double precision numbers.

```
                .data
num1:           .float 34.5454
num2:           .double 123.034
num3:           .double 3.545452e+2

                .text
                .globl   main
main:
        la      $s0, num1       # $s0 = address of num1
        lwc1    $f6, 0($s0)     # load the floating pt num at the address $s0

        la      $s1, num2       #$s1 = address of num2
        lwc1    $f2, 0($s1)     # load the floating pt num at the address $s1
        lwc1    $f3, 4($s1)     # get the second part of double
……....
```

lwc1 = load word coprocessor 1

- Is $x + (y + z) = (x + y) + z$?

- Yes, in mathematics
- No, in computer arithmetic (because of rounding errors)

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

Guard digits

- **Guard: The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.**
- **It is used with rounding**
- **Round: Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format.**

Example with Guard

- Assume 3 significant digits.
- Add $2.56_{ten} \times 10^{0}$ and $2.34_{ten} \times 10^{2}$

With Guard:

Shift the smaller num to the right to align the exponent.

$2.56 \times 10^{0} \rightarrow 0.0256 \times 10^{2}$ (we have 2 extra bits for guard)

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

The sum is 2.3656×10^{2}

by rounding to 3 significant digits, we get 2.37×10^{2}

Example without Guard

- Assume 3 significant digits.
- Add $2.56_{ten} \times 10^{0}$ and $2.34_{ten} \times 10^{2}$

Without Guard:

Shift the smaller num to the right to align the exponent.

$2.56 \times 10^{0} \rightarrow 0.02 \times 10^{2}$ (we can have 3 digits only)

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

The sum is 2.36×10^{2}

off by 1 in the last digit compared to the result with Guard.