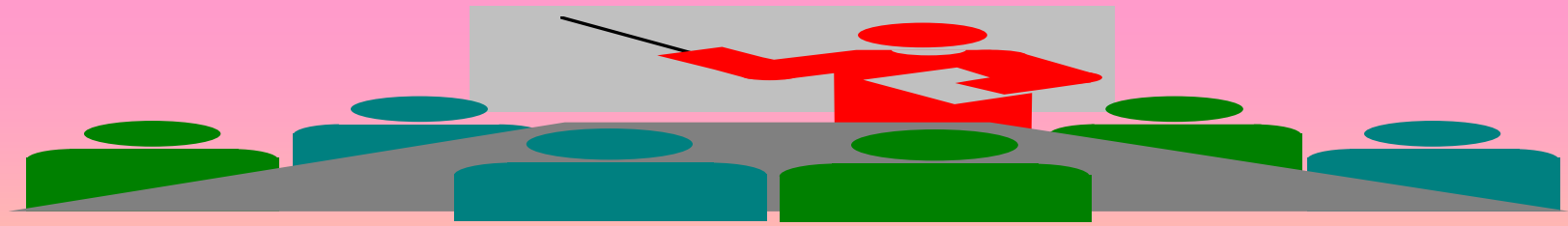


Computer Organization & Design

The Hardware/Software Interface

Qing-song Shi

http://10.214.26.103 **Email:** zjsqs@zju.edu.cn

Chapter 3
Arithmetic for Computer

Contents of Chapter 3

3.1 Introduction

- **Computer words are composed of bits;**
  thus words can be represented as binary numbers.
- **Simplified to contain only in course:**
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- **Generic Implementation:**
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- **All instructions use the ALU after reading the registers**
  Why? memory-reference? arithmetic? control flow?

Numbers

- Bits are just bits (no inherent meaning)— conventions define relationship between bits and numbers

- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: 0 1 2 3...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)

- How do we represent negative numbers?
  i.e., which bit patterns will represent which numbers?

Do you Know?

- What is this about following Digital?

  $00110011110111100000000100000000_2$

  - Don't know! (Do not know, is the right answer !)
- Ah, Why?
  - Because different occasions have different meaning
- The possible meaning is
  - IP Address
  - Machine instructions
  - Values of Binary number :
    - **Integer**
    - **Fixed Point Number**
    - **Floating Point Number**

For binary integer

- The following 4-bit binary integer What does it mean?

$$1001_2$$

  - Don’t know! (Do not know, is the right answer !)
- Ah, still do not know for?
- Integer representation of different methods have different meaning
-
  - Unsigned $1001_2=9_{10}$
  - Signed $1001_2=-1_{10}$ or -7_{10} ?

3.2 Signed and Unsigned Numbers
Possible Representations

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: number of zeros, ease of operations
- **Which one is best? Why?**

Numbers and their representation

- Number systems
  - Radix based systems are dominating
    decimal, octal, binary,…

$$0 \leqslant b \leqslant K$$

$$(N)_k = (A_{n-1}A_{n-2}A_{n-3}\ldots A_1A_0 \bullet A_{-1}A_{-2}A\ldots A_{-m+1}A_{-m})_k$$

MSD LSD

$$(N)_K = \left(\sum_{i=m}^{n-1} b_i \bullet k^{\,i}\right)_k$$

  - ***b***: value of the digit, ***k***: radix, ***n***: digits left of radix point, ***m***: digits right of radix point
  - Alternatives, e.g. Roman numbers (or Letter)
- Decimal (k=10) -- used by humans
- Binary (k=2) -- used by computers

Numbers and their representation

- Representation
  - ASCII - text characters
    - Easy read and write of numbers
    - Complex arithmetic (character wise)
  - Binary number
    - Natural form for computers
    - Requires formatting routines for I/O
- in MIPS:
  - Least significant bit is right (bit 0)
  - Most significant bit is left (bit 31)

Number types

- Integer numbers, unsigned
  - Address calculations
  - Numbers that can only be positive
- Signed numbers
  - Positive
  - Negative
- Floating point numbers
  - numeric calculations
  - Different grades of precision
    - Singe precision (IEEE)
    - Double precision (IEEE)
    - Quadruple precision

Number formats

- Sign and magnitude
- 2's complement
- 1's complement

  similar to 2's complement, + 0 & - 0
- Biased notation

  1000 0000 = minimal negative value(-2^7)

  0111 1111 = maximal positive value (2^7-1)
- Representation
  - Binary
  - Decimal
  - Hexadecimal

Signed number representation

- First idea:

  Positive and negative numbers

  - Take one bit (e.g. 31) as the **sign bit**
    - Problem
    - **0** 0000000 = 0 positive zero!
    - **1** 0000000 = 0 negative zero!
  - Each comparison to 0 requires two steps
- 1's complement
- 2's complement

Two's Complement Operations

- Negating a two's complement number:

  invert all bits and add 1 with end

  - remember: "negate" and "invert" are quite different!
- **Defining :** Assume: $x = \pm 0.x_{-1}x_{-2}x_{-3}\ldots x_{-m}$ OR $x = \pm x_{n-1}x_{-n-2}x_{-n-3}x_{-n-4}\ldots x_{-0}$

$$[X]_C = \begin{cases} X & 0 \leqslant X < 1 \\ 2 + X = 2 - |X| & -1 \leqslant X < 0 \end{cases}$$

fraction

$$\begin{cases} X & 0 \leqslant X < 2^n \\ 2^{n+1} + X = 2^{n+1} - |X| & -2^n \leqslant X < 0 \end{cases}$$

integer

- Converting n bit numbers into numbers with more than n bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits

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

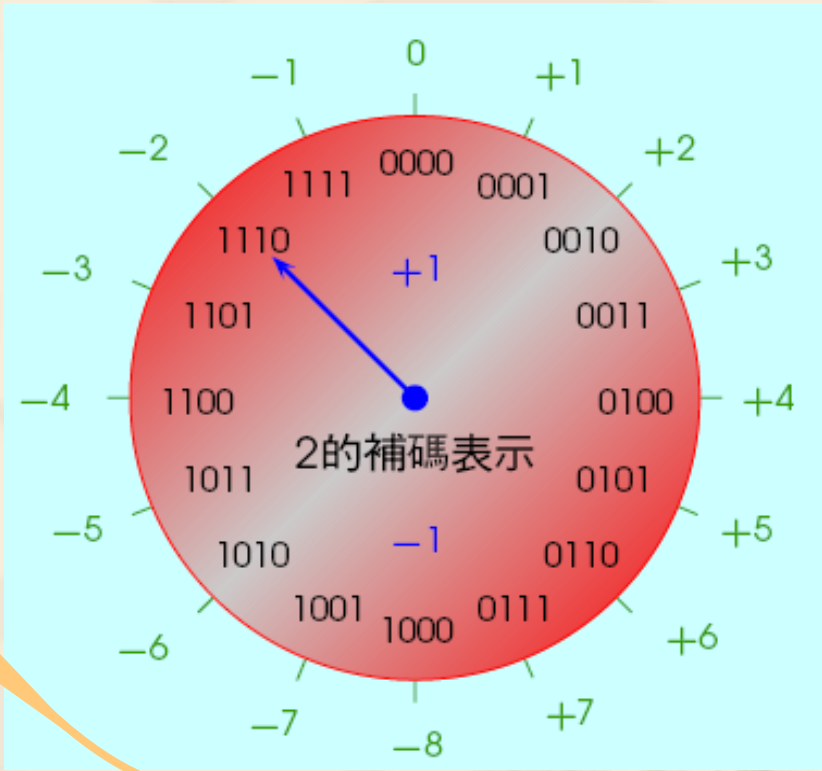
2's complement for n=3

$2^{n+1} + X = 2^{n+1} - |X|$

$= 2^4 - 8$

$= (10000 - 1000)_2$

$= 01000$



0 100 =+4

1 100 =-4

- Only one representation for 0
- One more negative number than positive number

More common: use of 2's complement ---- negatives have one additional number

$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000)_2 = (0)_{10}$

$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 = (1)_{10}$

....

$(0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101)_2 = (2\ ,\ 147\ ,\ 483\ ,\ 645)_{10}$

$(0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110)_2 = (2\ ,\ 147\ ,\ 483\ ,\ 646)_{10}$

$(0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111)_2 = (2\ ,\ 147\ ,\ 483\ ,\ 647)_{10}$

$(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000)_2 = (-2\ ,\ 147\ ,\ 483\ ,\ 648)_{10}$

$(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 = (-2\ ,\ 147\ ,\ 483\ ,\ 647)_{10}$

$(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010)_2 = (-2\ ,\ 147\ ,\ 483\ ,\ 646)_{10}$

....

$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101)_2 = (-3)_{10}$

$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110)_2 = (-2)_{10}$

$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111)_2 = (-1)_{10}$

Two's Biased notation

- Negating Biased notation number:

invert all bits and add 1 with end

Defining : Assume: $x = \pm x_{n-1} x_{-n-2} x_{-n-3} x_{-n-4} \cdots x_{-0}$

$$[X]_b = 2^n + X \qquad -2^n \leqslant X \leqslant 2^n$$

$$[0]_b = 100000\ldots(2^n)$$

```
X= + 1011 [X]b=11011        sign bit "1" Positive
X= - 1011 [X]b=00101        sign bit "0" Negative
```

2's Biased notation VS 2's complement

- **Only reverse sign bit**

e.g.

```
X= + 1011 [X]c=01011 [X]b=11011
X= - 1011 [X]c=10101 [X]b=00101
```

biase

IEEE 754: $[X]_b = 2^{n-1} + X$

sign extension (lbu vs. lb)

- Expansion

  e.g. 16 bit numbers to 32 bit numbers

- Required for operations with registers(32 bits) and immediate operands (16 bits)
- Sign extension
  - Take the lower 16 bits as they are
  - Copy the highest bit to the remaining 16 bits
  - 0000 0000 0000 0010 → 2

    0000 0000 0000 0000 0000 0000 0000 0010
  - 1111 1111 1111 1110 → -2

    1111 1111 1111 1111 1111 1111 1111 1110

Compare operations

- **Different compare operations required for both number types**
  - **Signed integer**
    - slt Set an less than
    - slti Set on less than immediate
  - **Unsigned integer**
    - sltu Set an less than
    - sltiu Set on less than immediate

Example for Compare

- Register $s0

  1111 1111 1111 1111 1111 1111 1111 1111

- Register $s1

  0000 0000 0000 0000 0000 0000 0000 0001

- Compared Operations

  slt $t0, $s0, $s1

  sltu $t0, $s0, $s1

- Results

  $t0 = 1 (-1 < 1)

  $t0 = 0 ($4{,}294{,}967{,}295_{ten} > 1_{ten}$)

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Don't always want to detect overflow
  — new MIPS instructions: `addu, addiu, subu`

  note: `addiu` *still sign-extends!*
  note: `sltu, sltiu` *for unsigned comparisons*

Bounds check Shortcut

- Reduce an index-out-of-bounds check
  - If ($a1>=$t2 & $a1<0) goto IndexOutofBounds

```
sltu $t0, $a1, $t2        #Temp reg $t0=0 if k>=length or k<0
beq $t0, $zero, IndexOutofBounds      # if bad, goto Error
```

```
lw $t2, 4($s6)            #temp reg $s2=length of array save
slt $t0, $s3, $zero       #temp reg $t0=1 if i<0
slt $t3, $s3, $t2         #temp reg $t3=0, if i>=length
slti $t3, $t3, 1          #temp reg $t3=1, if i>=length
or  $t3, $t3, $t0         # $t3=1, if i is out of bounds
bne $t3,$zero, IndexOutOfBounds      # if out of bounds, goto Error
```

3.3 Arithmetic

- Addition and Subtraction
- Logical operations
- Constructing a simple ALU
- Multiplication
- Division
- Floating point arithmetic
- Adding all parts to get an ALU

Addition & subtraction

- Adding bit by bit, carries -> next digit

0000 0111	7_{10}
+0000 0110	6_{10}
0000 1101	13_{10}

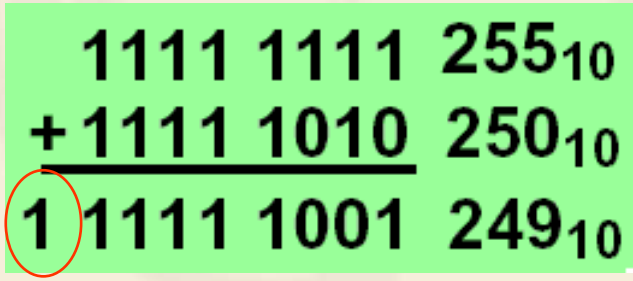
- Subtraction
  - Directly
  - Addition of 2's complement

0000 0111	7_{10}
- 0000 0110	6_{10}
0000 0001	1_{10}

0000 0111	7_{10}
+1111 1010	-6_{10}
0000 0001	1_{10}

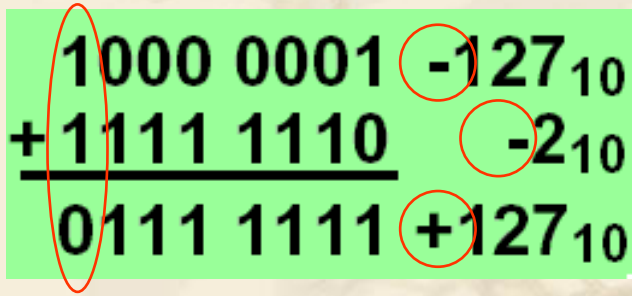
Overflow

- The sum of two numbers can exceed any representation



- The difference of two numbers can exceed any representation
- 2's complement:

  Numbers change sign and size



Overflow conditions

- General overflow conditions

Operation	Operand A	Operand B	Result overflow
A+B	≧0	≧0	<0 **(01)**
A+B	<0	<0	≧0 **(10)**
A-B	≧0	<0	<0 **(01)**
A-B	<0	≧0	≧0 **(10)**

- Reaction on overflow
  - Ignore ?
  - Reaction of the OS
  - Signalling to application (Ada, Fortran,...)

Double sign-bits

Overflow process

- Hardware detection in the ALU
- Generation of an exception (interrupt)
- Save the instruction address (not PC) in special register EPC
- Jump to specific routine in OS
  - Correct & return to program
  - Return to program with error code
  - Abort program

Which instructions cause Overflow

- **Overflows in signed arithmetic instructions cause exceptions:**
  - 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)

Handle with care!

New MIPS instructions

- Byte instructions
  - **lbu**: load byte unsigned
    - Loads a byte into the lowest 8 bit of a register
    - Fills the remaining bits with **'0'**
  - Lb: load byte (signed)
    - Loads a byte into the lowest 8 bit of a register
    - Extends the highest bit into the remaining 24 bits
- Set instructions for conditional branches
  - sltu: set on less than unsigned
  - Sltiu: set on less than unsigned immediate

Logical operations skip

- Logical shift operations
  - right (srl)
  - left (sll)

Filled with '0'

- The machine instruction for the instruction
  sll $t2, $s0, 3 (sll rd,rt,i)

Op	Rs	Rt	Rd	Shamt	Adr/funct
0	0	16	10	3	0

- Example: sll $t2, $s0, 3
  - $s0: 0000 0000 0000 0000 1100 1000 0000 1111
  - $t2: 0000 0000 0000 0110 0100 0000 0111 1000

Logical operations skip

- AND→bit-wise AND between registers

  and register1, register2, register3

- OR →bit-wise OR between registers

  or register1, register2, register3

- Example:

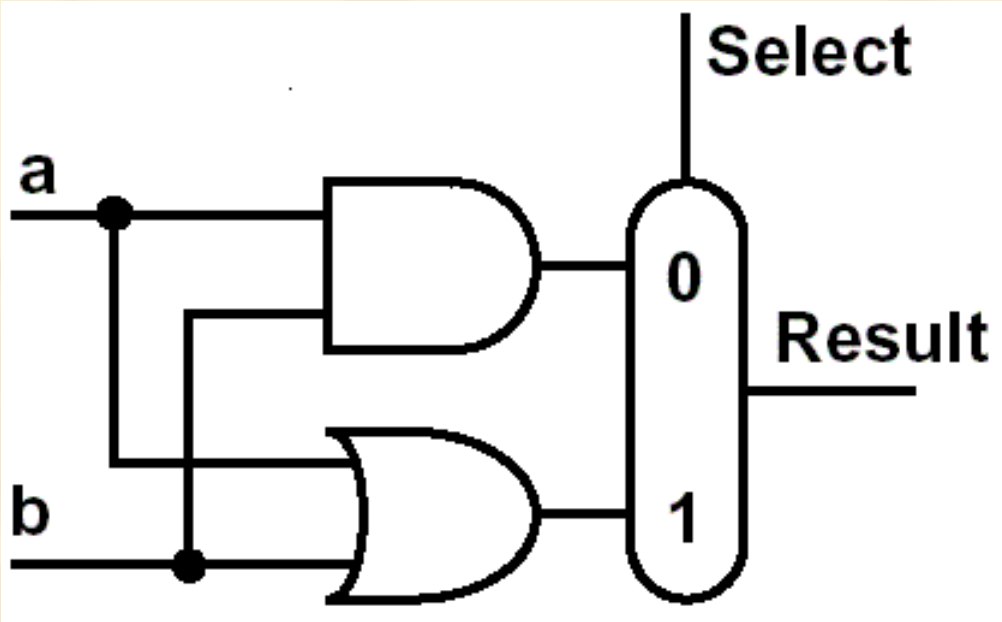
  and $3, $10, $16

  or $4, $10, $16

  - R16: 0000 0000 0000 0000 1100 1000 0000 1111
  - R10: 0000 0000 0000 0110 0100 0000 0111 1000
  - R3: 0000 0000 0000 0000 0100 0000 0000 1000
  - R4: 0000 0000 0000 0110 1100 1000 0111 1111

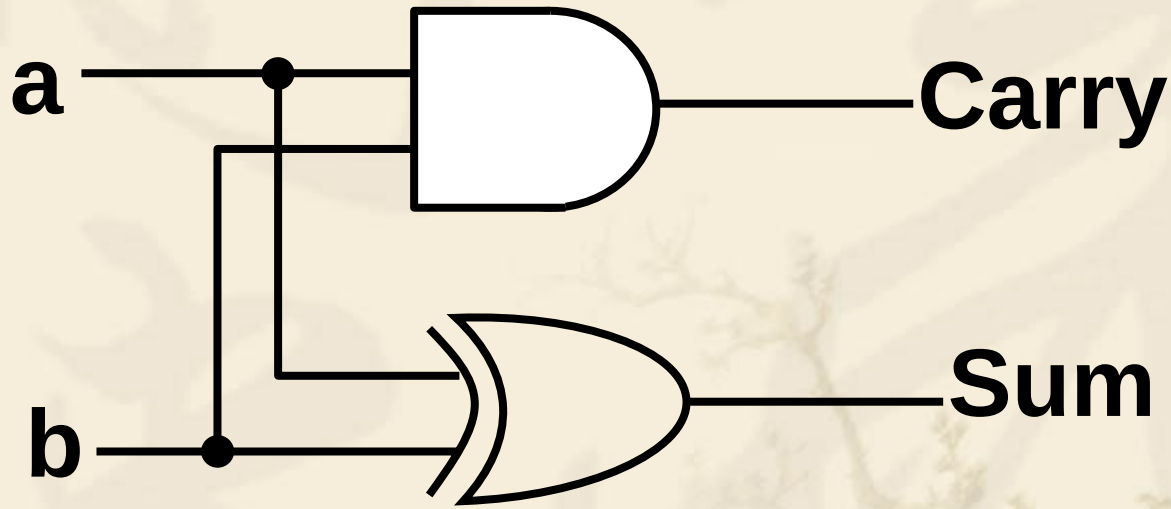
Constructing an ALU

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



A half adder

- Sum = a b + a b
- Carry = a b



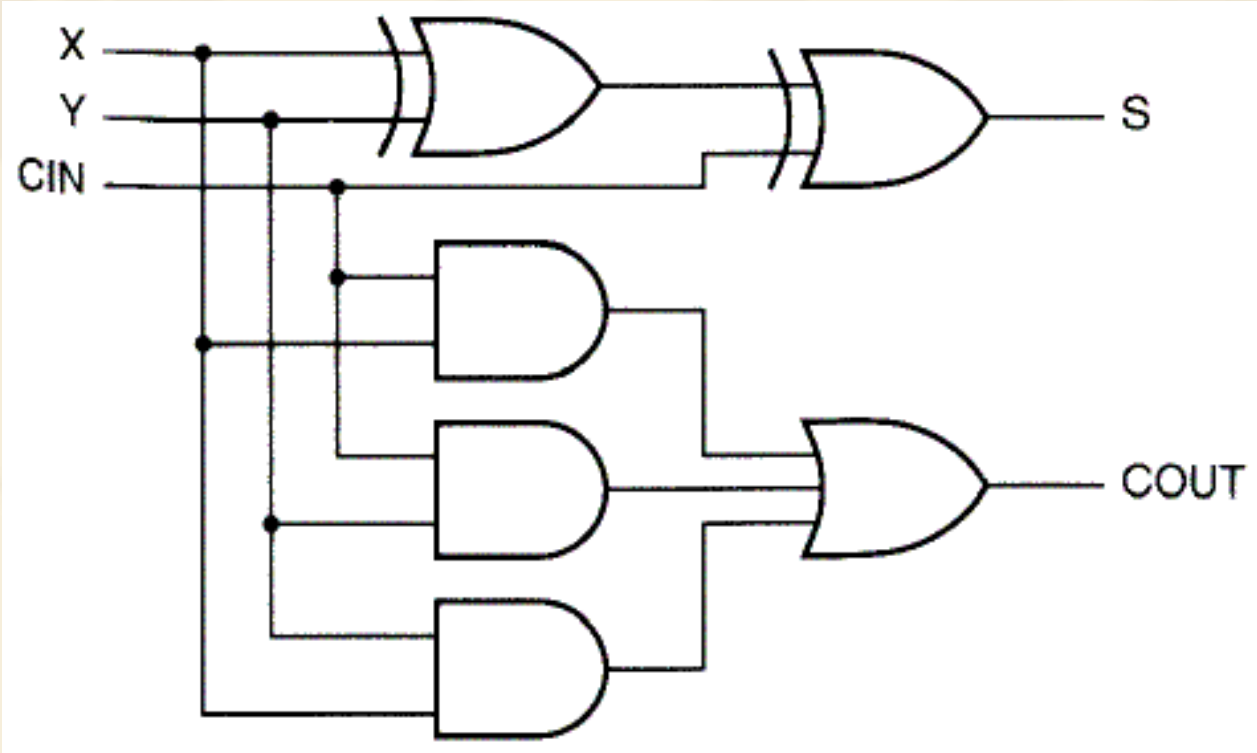
A full adder

- Accepts a carry in
- Sum = $A \oplus B \oplus Carry_{In}$
- $Carry_{Out} = B\ Carry_{In} + A\ Carry_{In} + A\ B$

Inputs			Outputs		Comments
A	B	$Carry_{In}$	$Carry_{Out}$	Sum	(two)
0	0	0	0	0	0+0+0=00
0	0	1	0	1	0+0+1=01
0	1	0	0	1	0+1+0=01
0	1	1	1	0	0+1+1=10
1	0	0	0	1	1+0+0=01
1	0	1	1	0	1+0+1=10
1	1	0	1	0	1+1+0=10
1	1	1	1	1	1+1+1=11

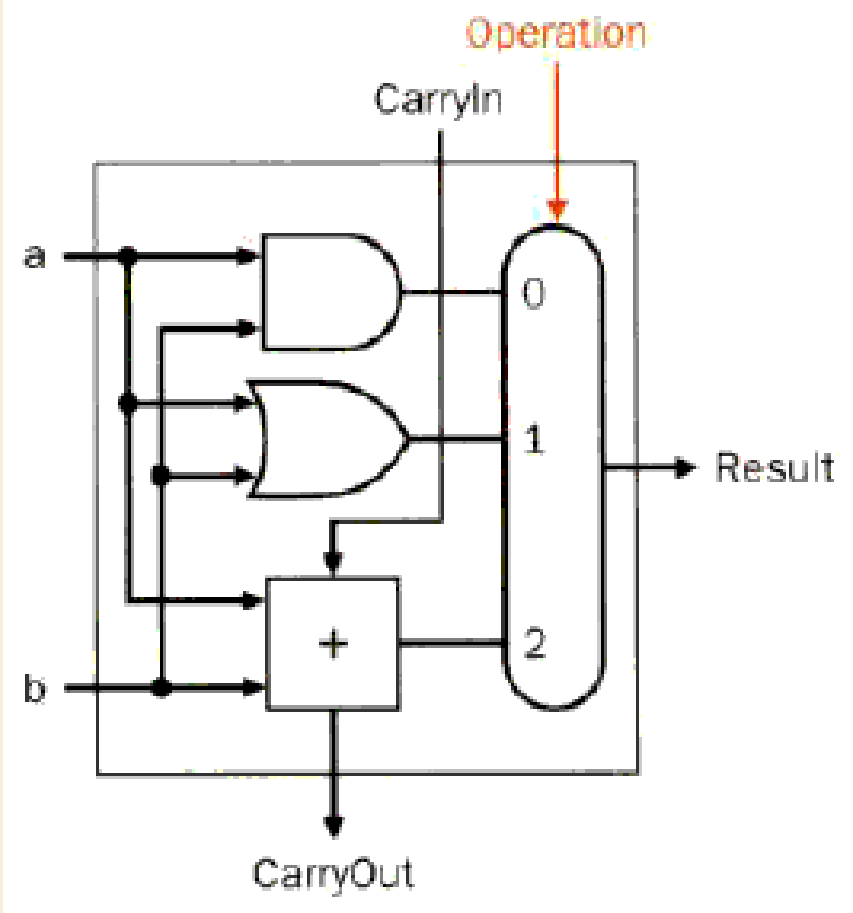
Full adder Logic circuit

- Full adder in 2-level design



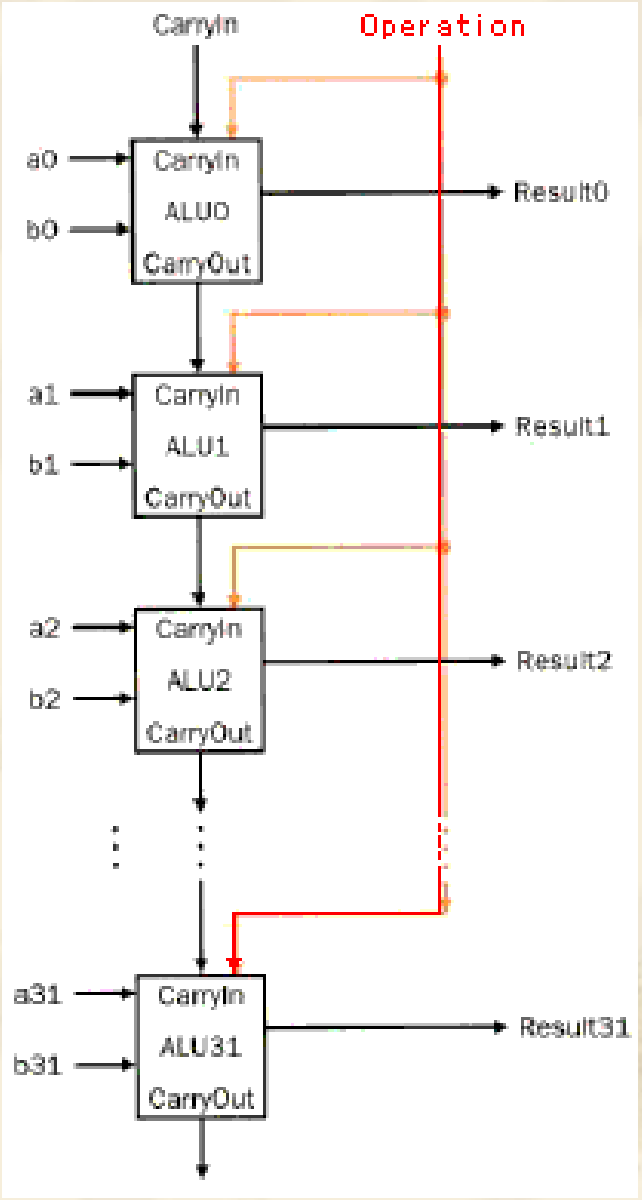
1 bit ALU

- ALU
  - AND
  - OR
  - ADD
- Cell
  Cascade Element



Basic 32 bit ALU

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



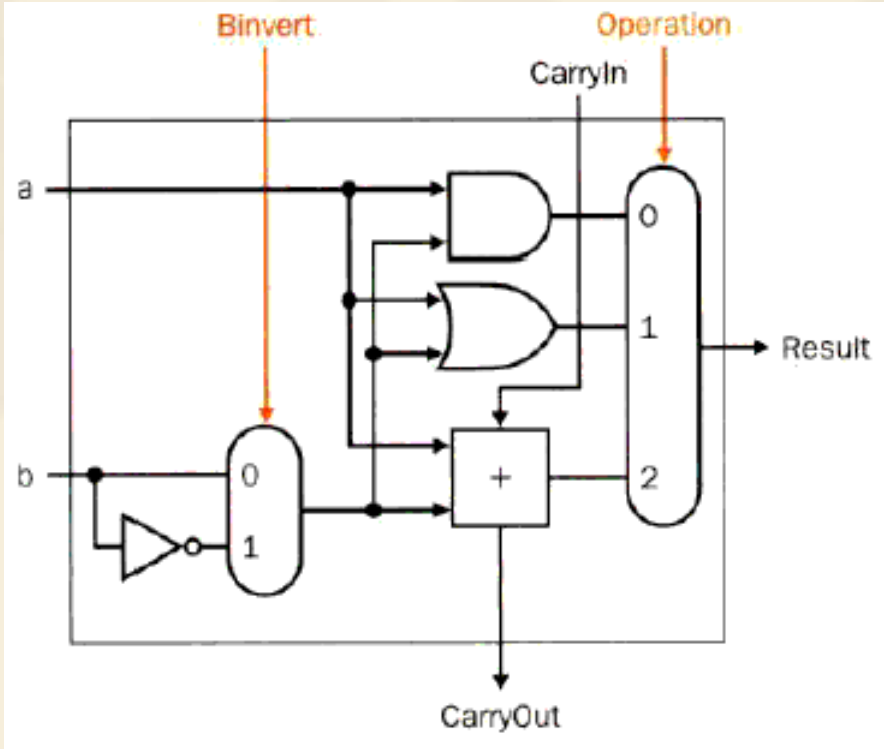
Iterative Circuit

Extended 1 bit ALU--- Subtraction

- Subtraction

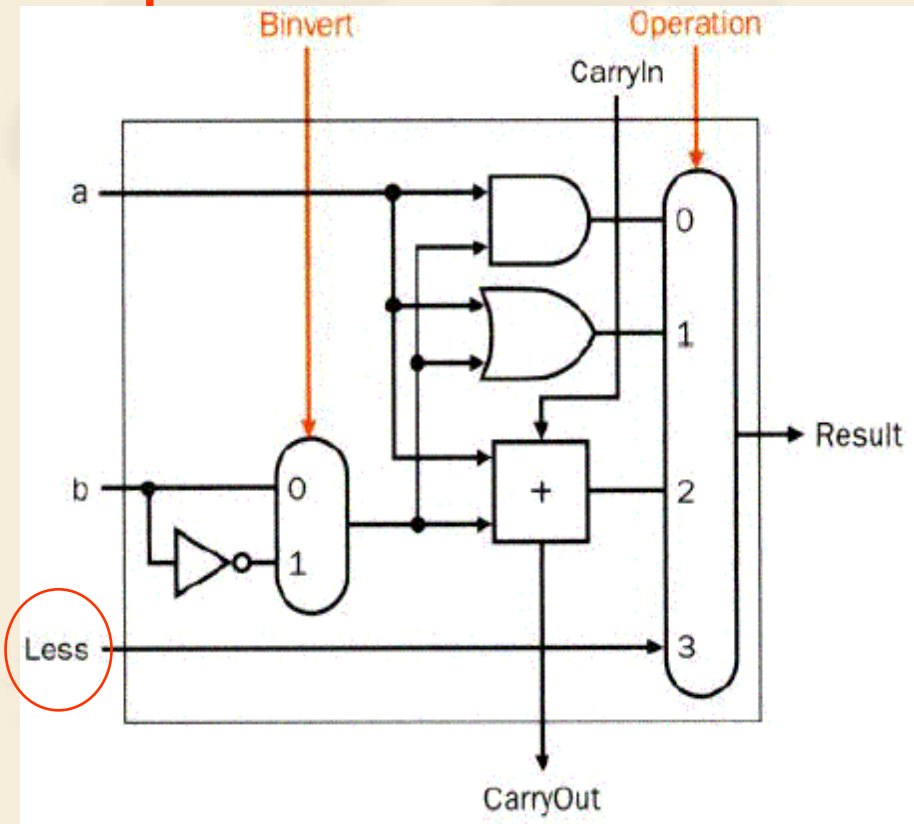
  a - b

  - Inverting b
  - ***1st CarryIn= 1***



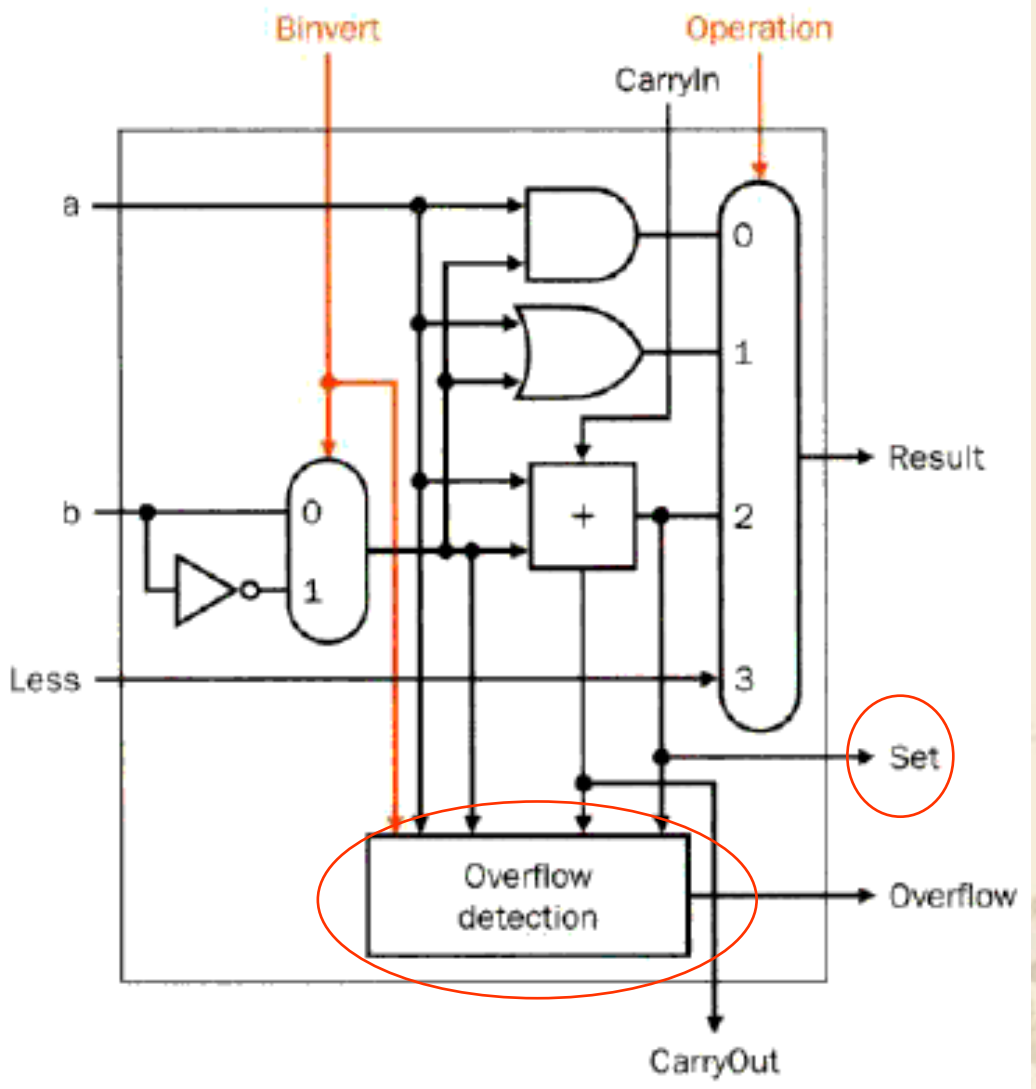
Extended 1 bit ALU-- comparison

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



Most significant bit

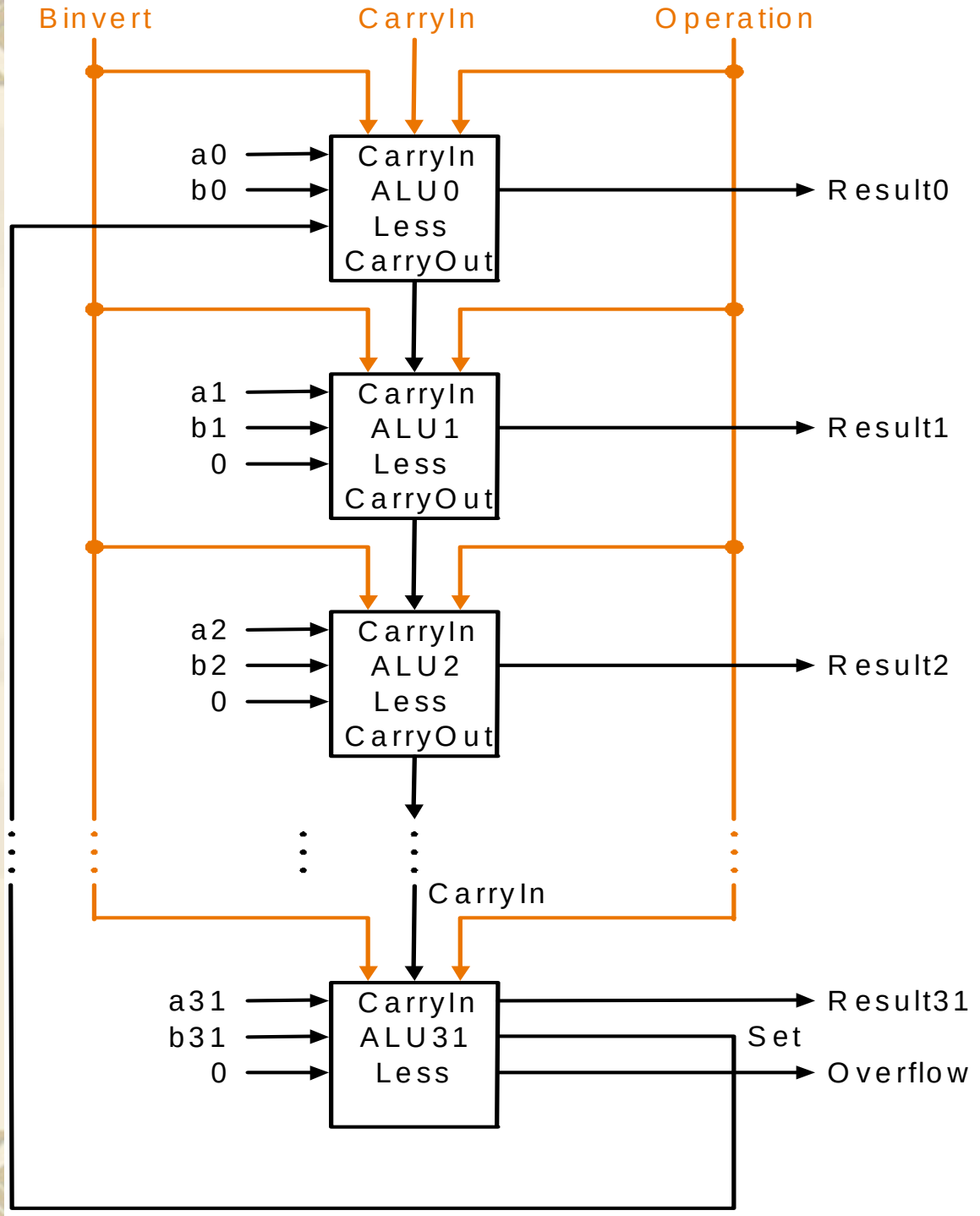
- Set for comparison
- Overflow detect



Complete ALU

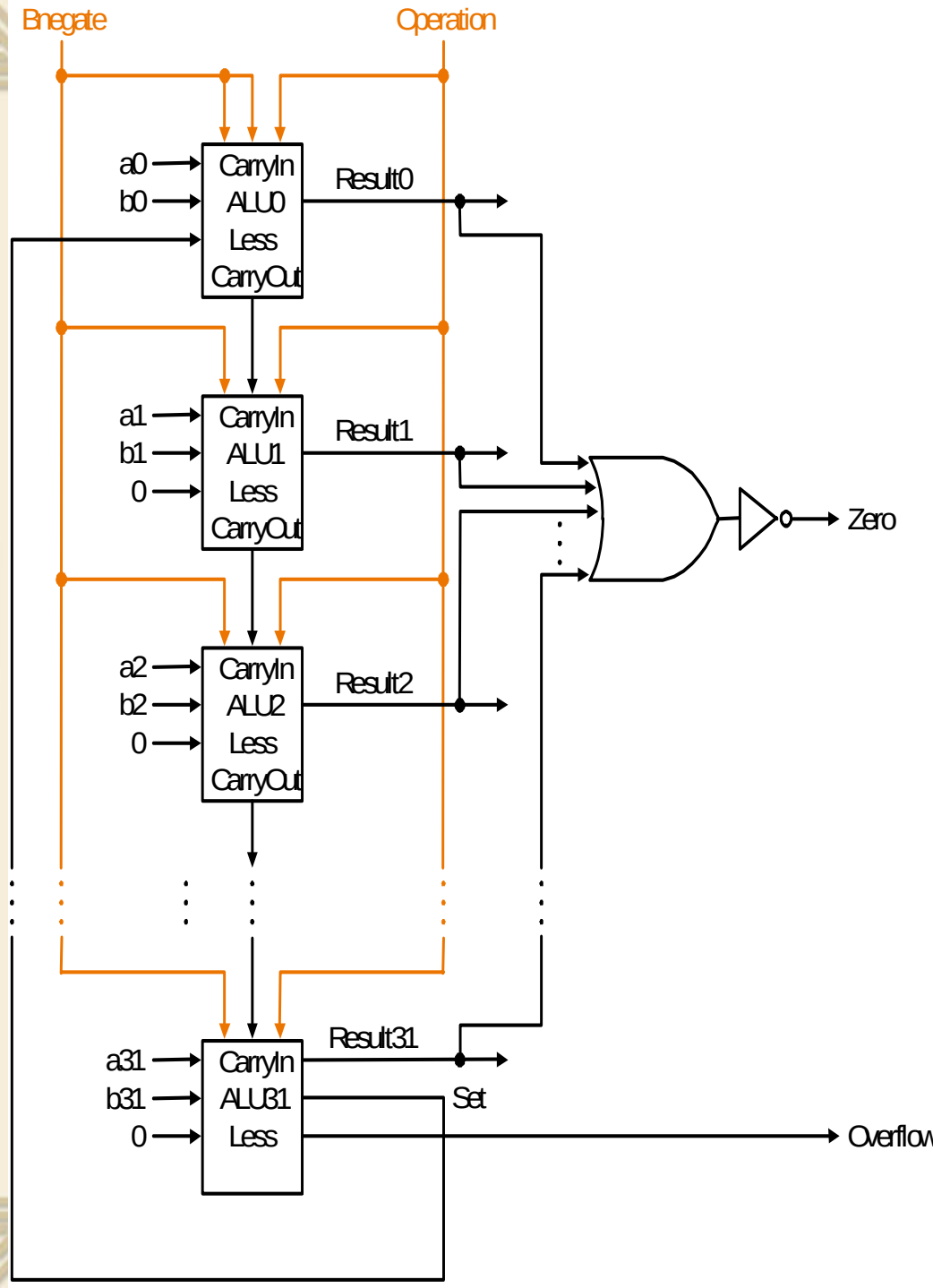
- Input
  - A、B
- Control lines
  - Binvert
  - Operation
  - Carryin
- Output
  - Result
  - Overflow

Iterative Circuit



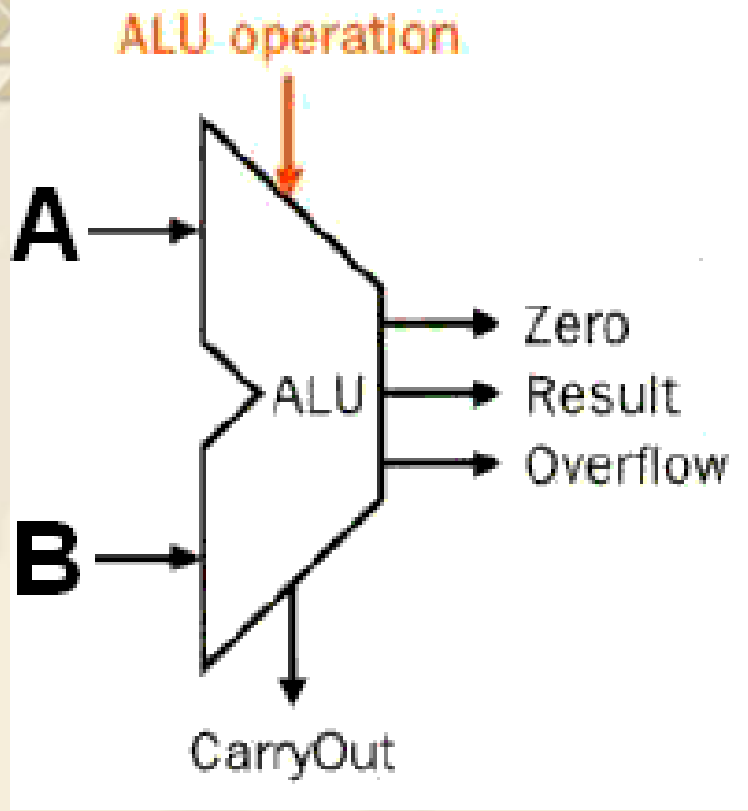
Complete ALU —with Zero detector

- Add a Zero detector



ALU symbol & Control

- Symbol of the ALU



- Control：Function table

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

ALU Hardware Code

```
module alu(A, B, ALU_operation, res, zero, overflow );
    input [31:0] A, B;
   input [2:0] ALU_operation;
   output [31:0] res;
   output zero, overflow ;
   wire [31:0] res_and,res_or,res_add,res_sub,res_nor,res_slt;
   reg [31:0] res;
   parameter one = 32'h00000001, zero_0 = 32'h00000000;
      assign res_and = A&B;
      assign res_or = A|B;
      assign res_add = A+B;
      assign res_sub = A-B;
      assign res_slt =(A < B) ? one : zero_0;
      always @ (A or B or ALU_operation)
            case (ALU_operation)
            3'b000: res=res_and;
            3'b001: res=res_or;
            3'b010: res=res_add;
            3'b110: res=res_sub;
            3'b100: res=~(A | B);
            3'b111: res=res_slt;
            default: res=32'hx;
            endcase
      assign zero = (res==0)? 1: 0;
endmodule
```

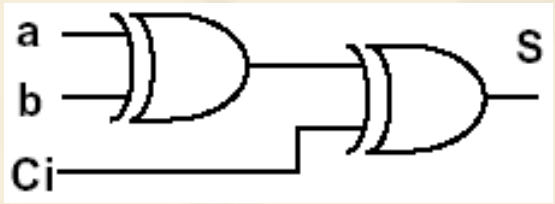
How do you write with overflow code ?

What is the difference The codes in the Synthesize?

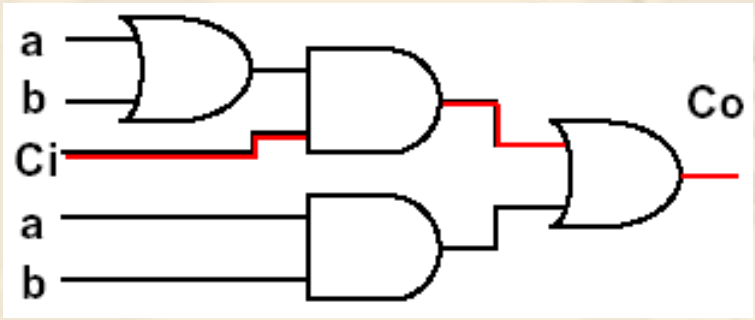
```
always @ (A or B or ALU_operation)
            case (ALU_operation)
          3'b000: res=A&B;
          3'b001: res=A|B;
          3'b010: res=A+B;
          3'b110: res=A-B;
                3'b100: res=~(A | B);
                3'b111: res=(A < B) ? one :
zero_0;
              default: res=32'hx;
       endcase
```

Speed considerations skip

- Previously used: ripple carry adder
- Delay for the sum: two units

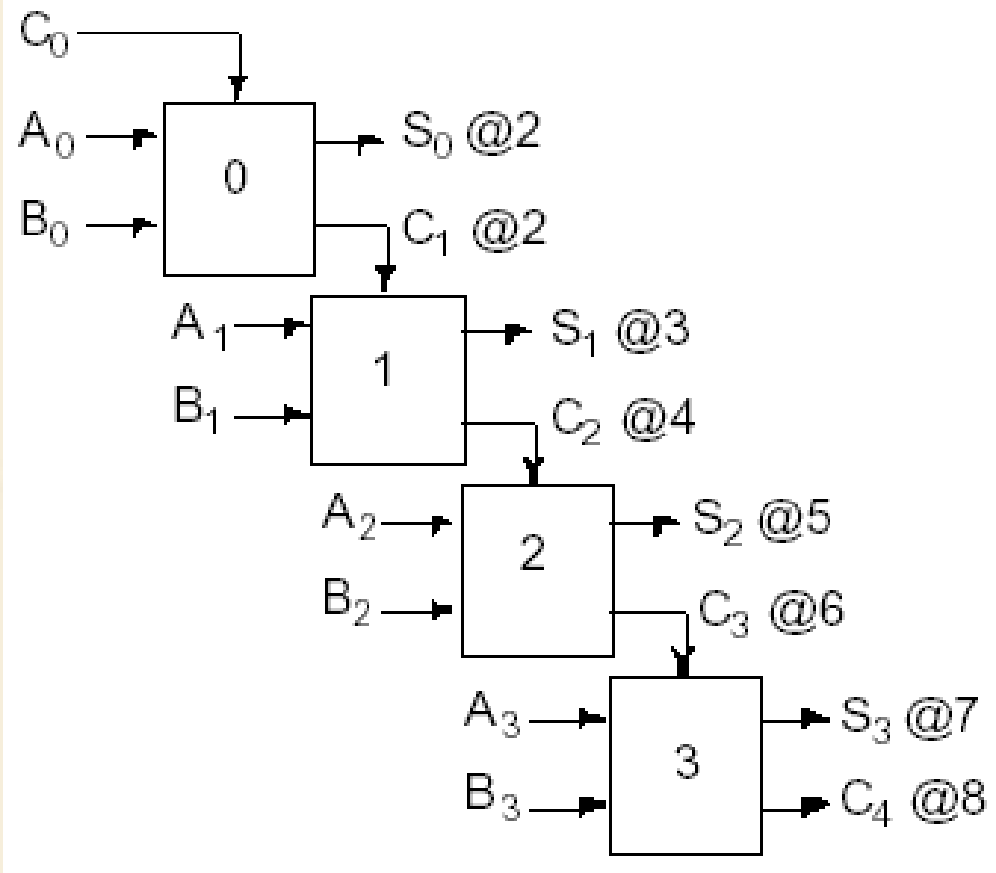


- Delay for the carry: two - three units



Speed considerations

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



Fast adders

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

  Optimum between speed and size

Fast adders

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

Carry look ahead adder (CLA)

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

    $= a_i b_i + a_i + b_i c_i$
  - Generate $g_i = a_i b_i$
  - Propagate $p_i = a_i + b_i$

Carry look ahead adder

- $C_{i+1} = g_i + p_i \ c_i$
- Carry generate: $g_i = a_i \ b_i$
  - If a and b are '1' ->

    we always have a carryout independent of c_i
- Carry propagate: $p_i = a_i + b_i$
  - If only one of a and b is '1' ->

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

Four bit carry look ahead adder

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

COMMENT:

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

PROBLEM:

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

Four bit carry look ahead adder

Let’s consider a 16-bit adder.

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

As we know:

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

So,we can get the following:

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

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

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

Assume: G0= g3 + p3*g2 + p3*p2 *g1 +p3*p2*p1*g0

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

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

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

P0= p3 * p2 * p1 * p0

P1= p7 * p6 * p5 * p4

P2= p11 * p10 * p9 * p8

P3= p15 * p14 * p13 * p12

Four bit carry look ahead adder

Then we get:

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

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

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

Then:

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

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

And,we can further get:

C1=G0+P0*c0 ;

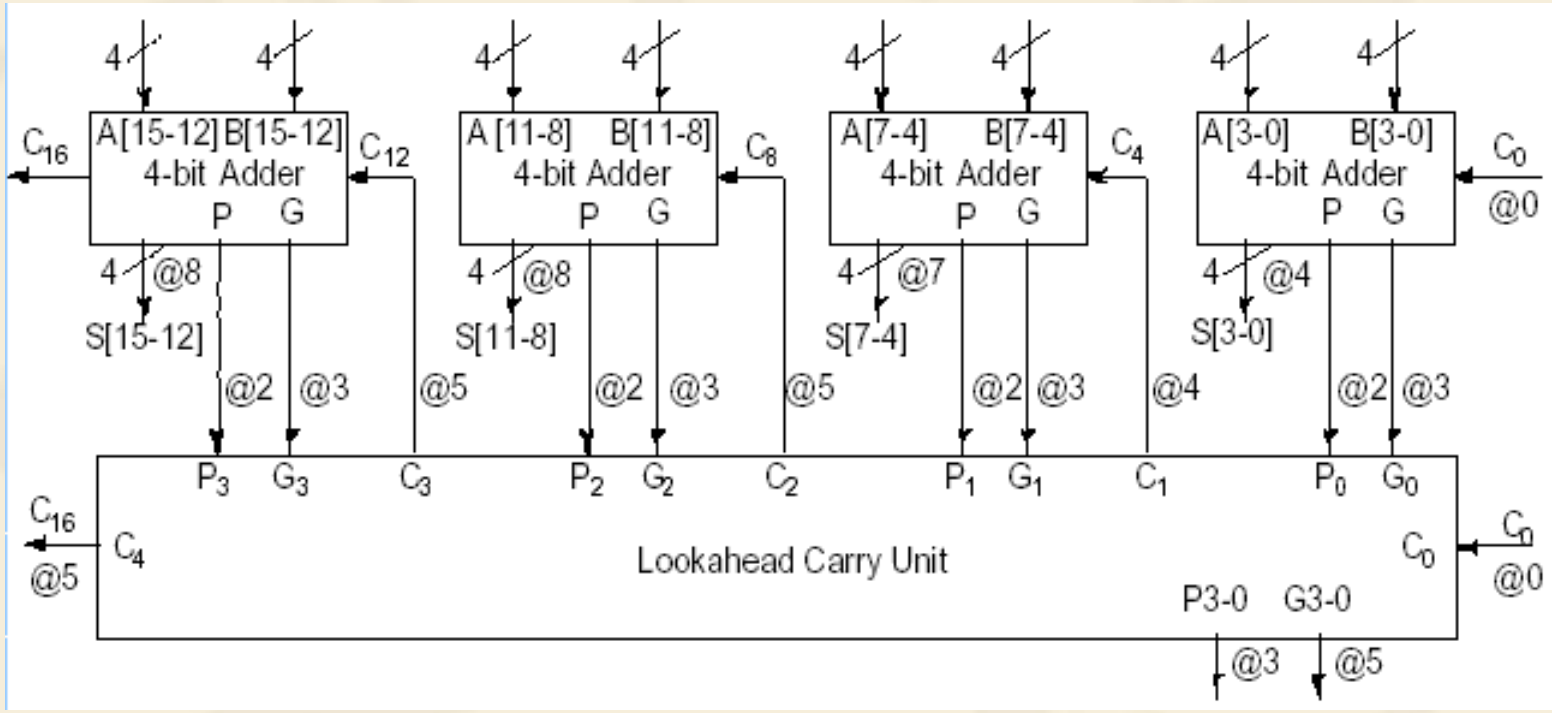
C2= G1+P1*C1 = G1+P1*G0+ P1* P0*c0

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

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

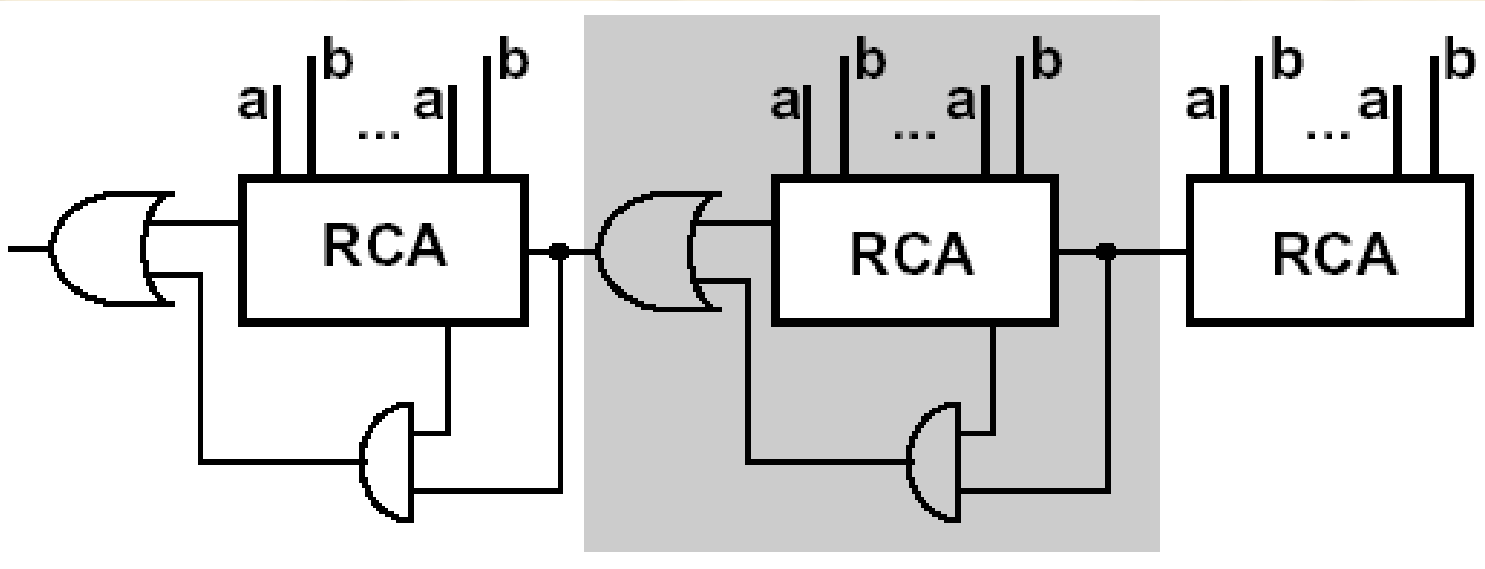
Hybrid CLA + Ripple carry

- Realisation:
  - Ripple carry adders and
  - Carry look ahead logic

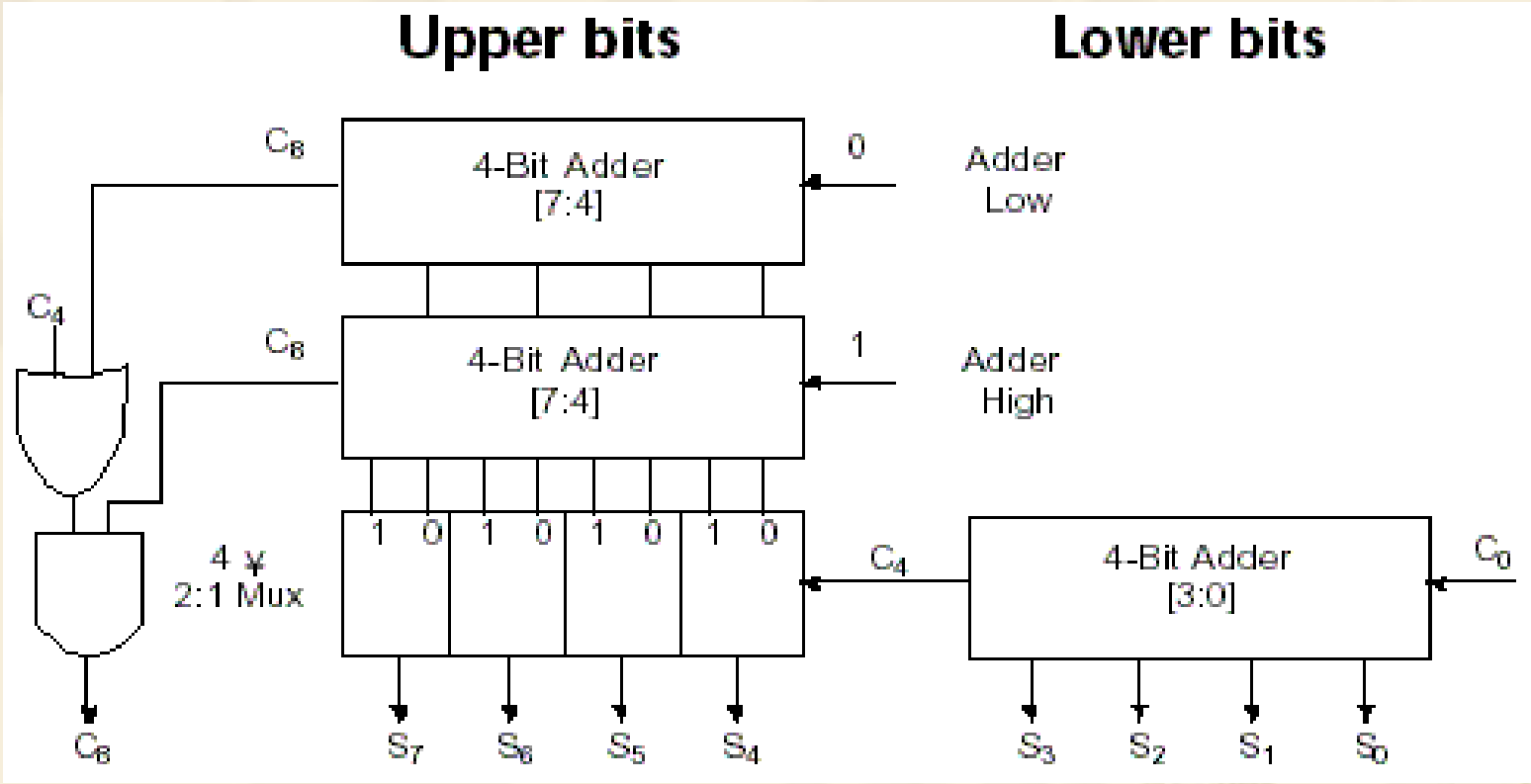


Carry skip adder

- Accelerating the carry by skipping the interior blocks
- Optimal speed with no-equal distribution of block length

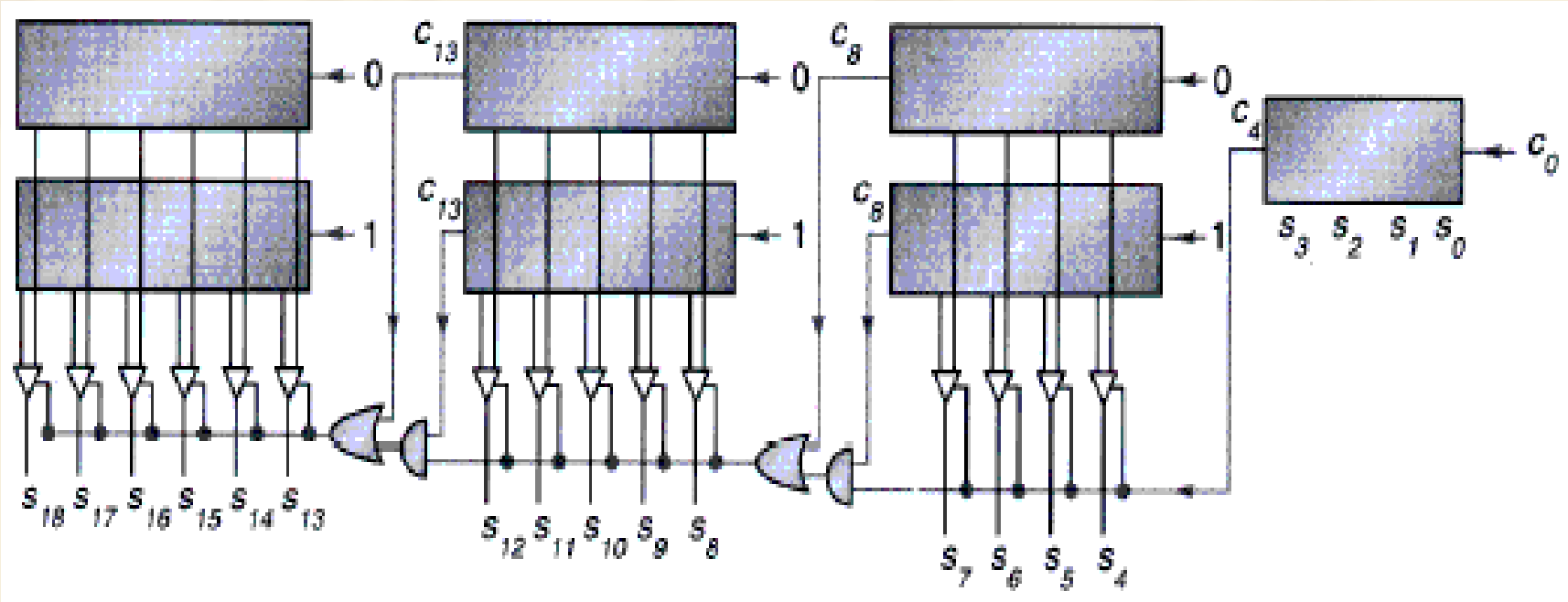


Carry select adder (CSA)



Carry select adder

- Carry selection by nibbles



3.4 Multiplication

- Binary multiplication

  Multiplicand × Multiplier

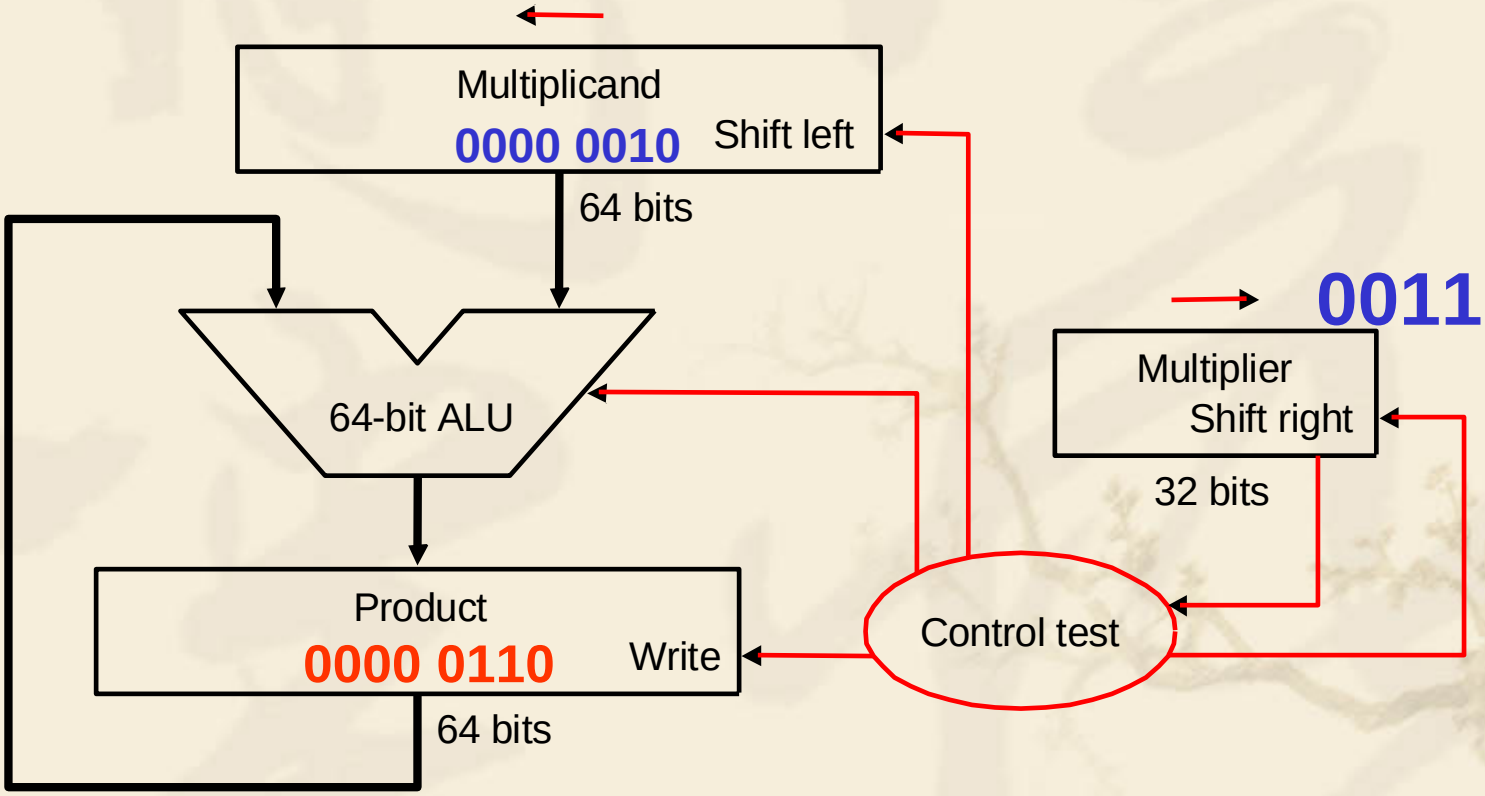
  1000 × 1001

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

```
      1 0 0 0
    × 1 0 0 1
  -----------
      1 0 0 0
    0 0 0 0
  0 0 0 0
+ 1 0 0 0
-----------
1 0 0 1 0 0 0 0
```

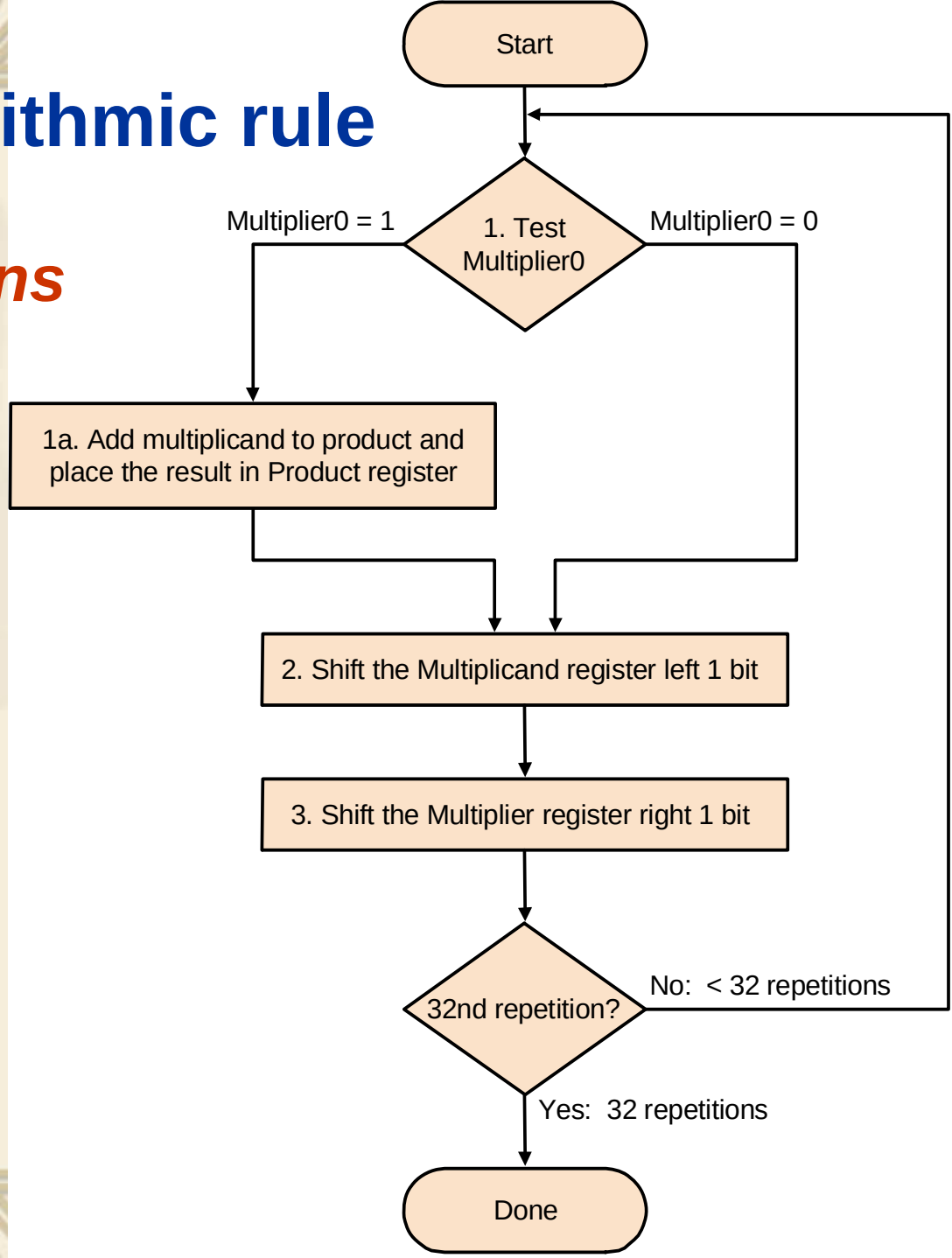
Multiplier V1– Logic Diagram

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



Multiplier V1--Algorithmic rule

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

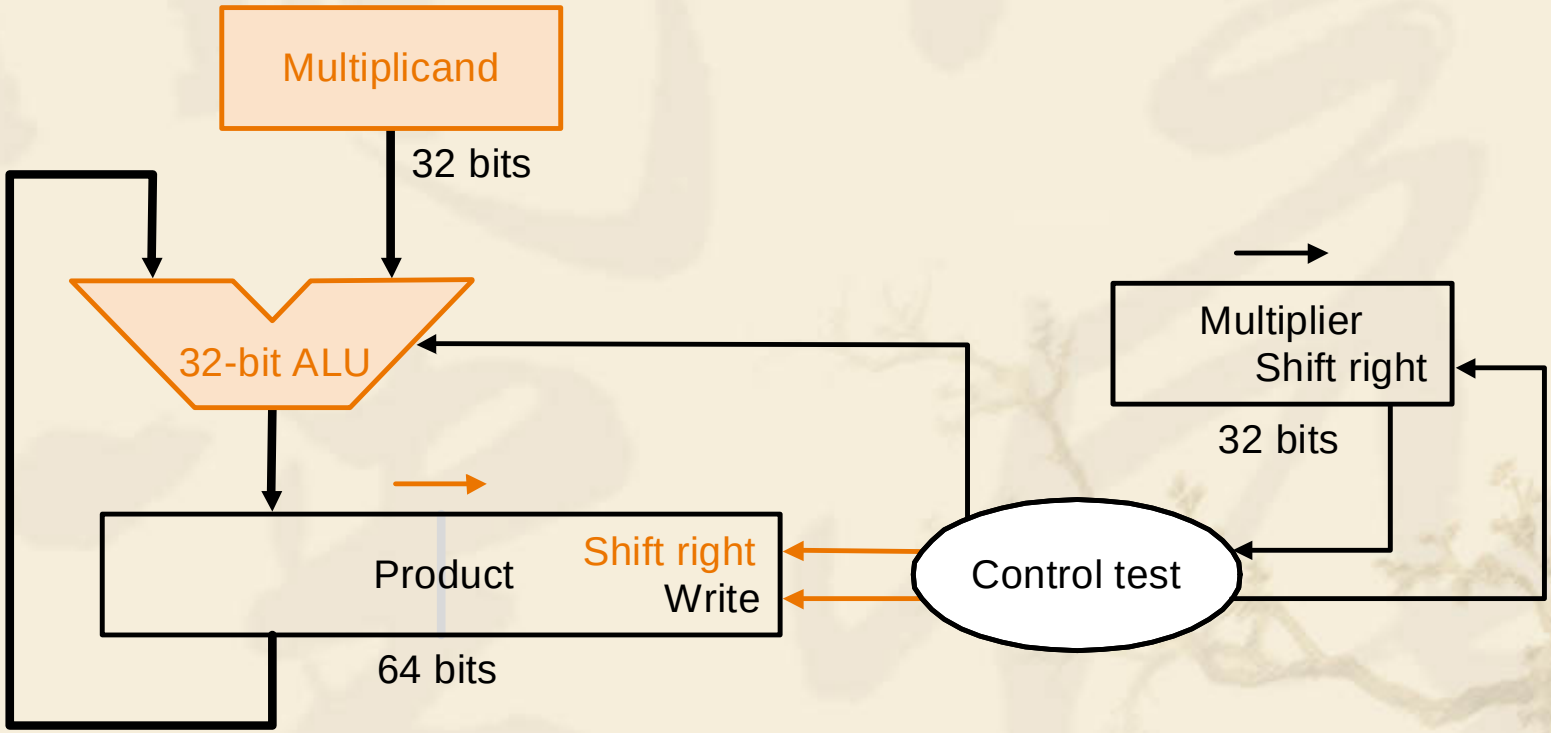


Multiplier V2

- 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!

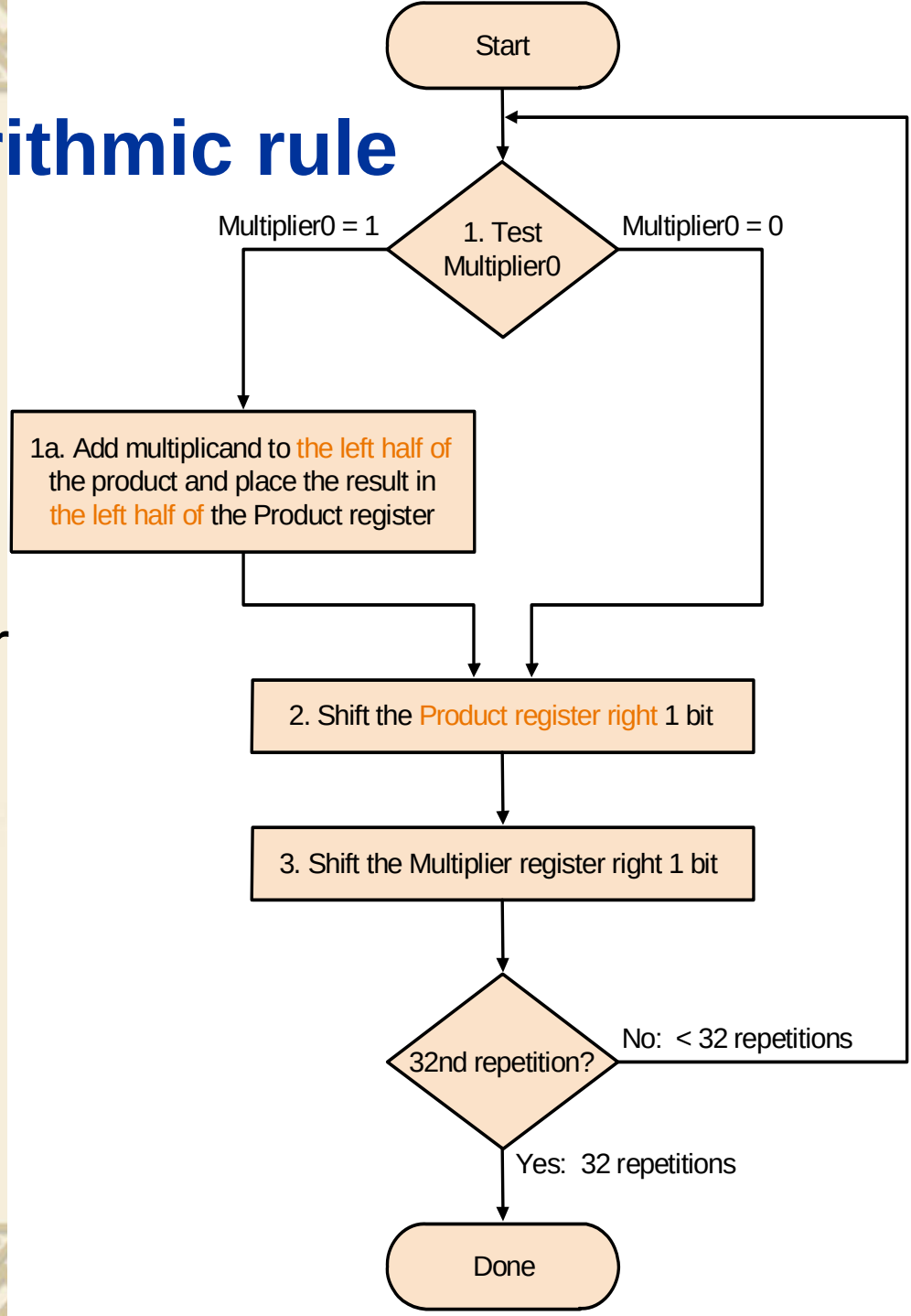
Multiplier V2-- Logic Diagram

- 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



Revised 4-bit example with V2

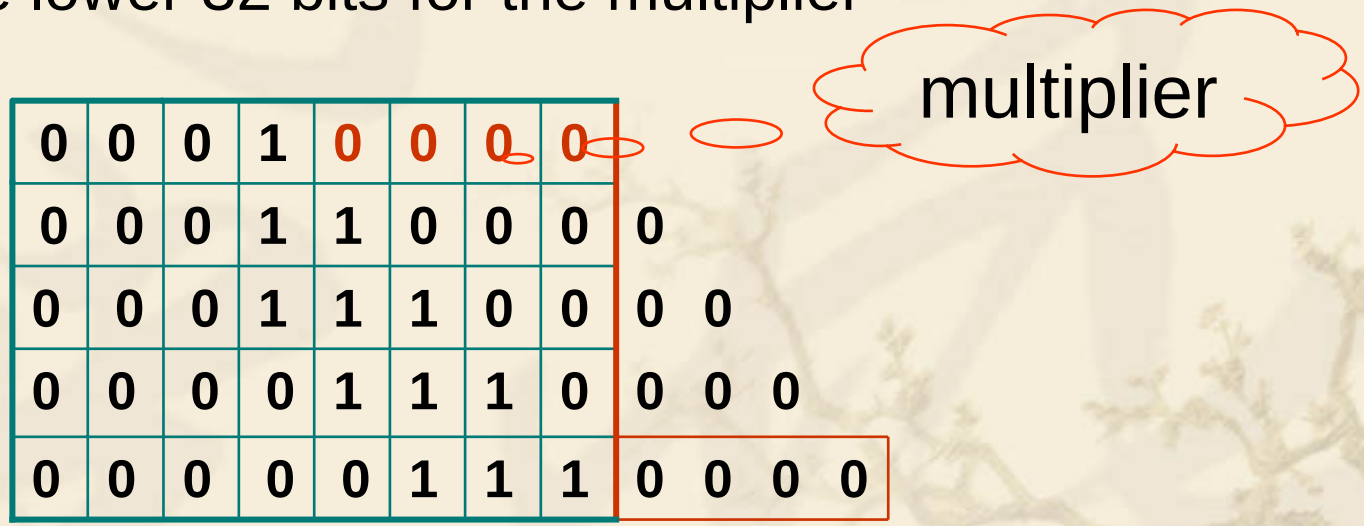
- Multiplicand x multiplier: 0001 x 0111

		Shift out	
Multiplicand:	**0001**		
Multiplier:×	**0111**		
	00000000		#Initial value for the product
1	**00010000**		#After adding 0001, Multiplier=1
	00001000	**0**	#After shifting right the product one bit
	0001		
2	**00011000**		#After adding 0001, Multiplier=1
	00001100	**0**	#After shifting right the product one bit
	0001		#After adding 0001, Multiplier=1
3	**00011100**		
	00001110	**0**	#After shifting right the product one bit
	0000		
4	**00001110**		#After adding 0001, Multiplier=0
	00000111	**0**	#After shifting right the product one bit

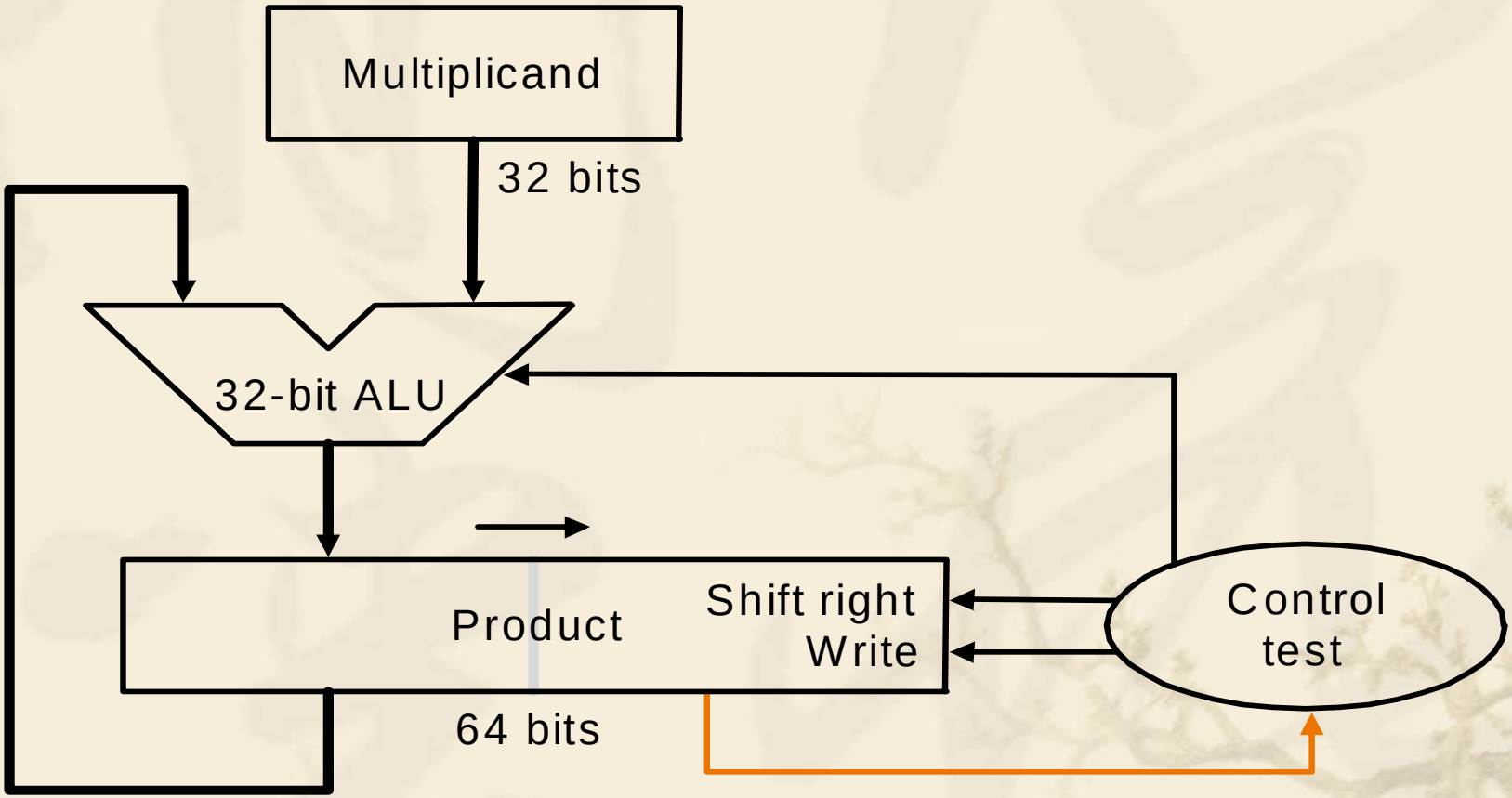
Multiplier V 3

- Further optimization
- At the initial state the product register contains only '0'
- The lower 32 bits are simply shifted out
- Idea:

  use these lower 32 bits for the multiplier

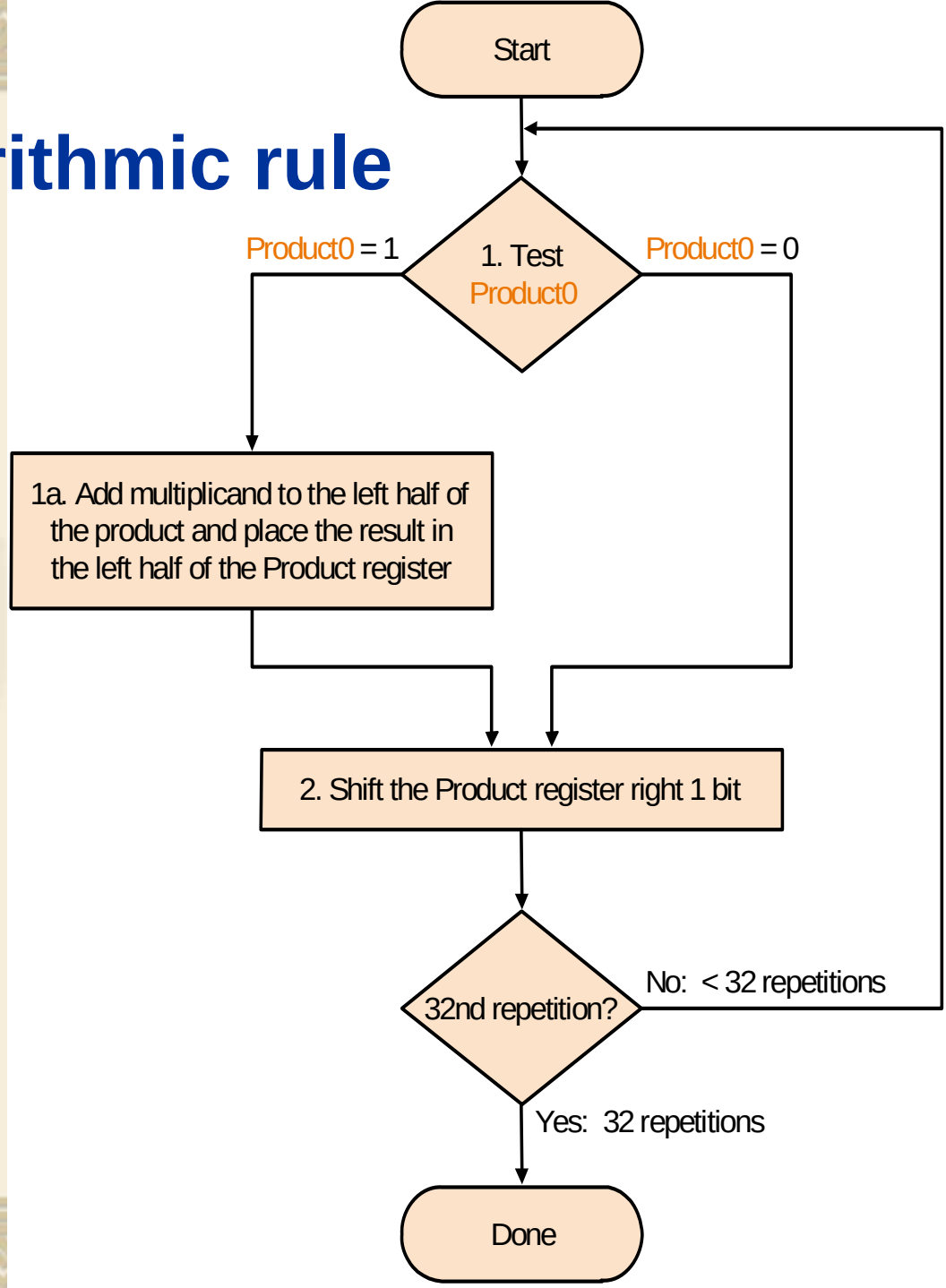


Multiplier V3 Logic Diagram



Multiplier V3--Algorithmic rule

- Set product register to '0'
- Load lower bits of product register with multiplier
- Test least significant bit of product register



Example with V3

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

		Shift out	
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	**1**	#After shifting right the product one bit
	0001		#After adding 0001, Multiplier=1
3	**00011101**		
	00001110	**1**	#After shifting right the product one bit
	0000		
4	**00001110**		#After adding 0001, Multiplier=0
	00000111	**0**	#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
- Improved method:

  Booth's Algorithm

Assumption: addition and subtraction are available

Principle -- Decomposable multiplication

- Assumes： $Z=y\times 10111100$

  $Z=y(10000000+111100+100-100)$

  $=y(1\times 2^7+1000000-100)$

  $=y(1\times 2^7+1\times 2^6-2^2)$

  $=y(1\times 2^7+1\times 2^6+0\times 2^5+0\times 2^4+0\times 2^3+0\times 2^2+0\times 2^1+0\times 2^0-1\times 2^2)$

  $=y(1\times 2^7+1\times 2^6+0\times 2^5+0\times 2^4+0\times 2^3+0\times 2^2-1\times 2^2+0\times 2^1+0\times 2^0)$

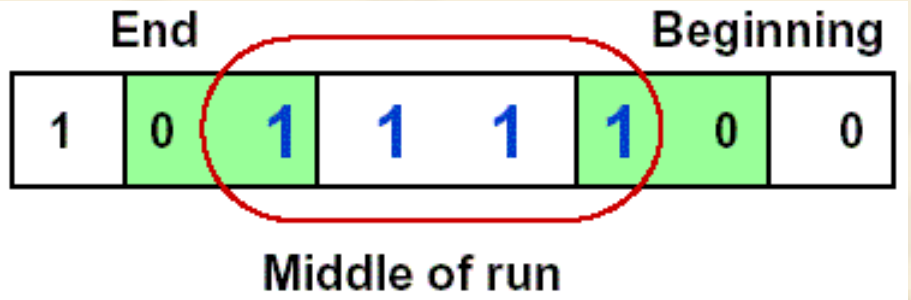
  $=y\times 2^7+y\times 1\times 2^6+0\times 2^5+0\times 2^4+0\times 2^3+0\times 2^2-y\times 2^2+0\times 2^1+0\times 2^0)$

  add　Only shift　sub　Only shift

  1　01　111　00

Booth's Algorithm

- Idea: If you have a sequence of '1's
  - subtract at first '1' in multiplier
  - shift for the sequence of '1's
  - add where prior step had last '1‘



- Result:
  - Possibly less additions and more shifts
  - Faster, if shifts are faster than additions

Example for Booth's Algorithm

- Logic required identifying the run

straight

```
0010 * 0110
      0000   shift
     0010    add
    0010     add
   0000      shift
  00001100
```

Booth

```
0010 * 0110
      0000   shift
     0010    sub
    0000     shift
   0010      add
  00001100
```

Booth's Algorithm rule

- Analysis of two consecutive bits

Current	last	Explanation	Example
1	0	Beginning	0000111**10**000
1	1	middle of '1'	0000**11**10000
0	1	End	000**01**1110000
0	0	Middle of '0'	00**00**11110000

- Action
  - 1 0 subtract multiplicand from left
  - 1 1 no arithmetic operation
  - 0 1 add multiplicand to left half
  - 0 0 no arithmetic operation
- Bit_{-1} = '0'
- Arithmetic shift right:
  - keeps the **leftmost bit constant**
  - no change of sign bit !

Example with negative numbers

- 2 * -3 = - 6
- 0010 * 1101 = 1111 1010

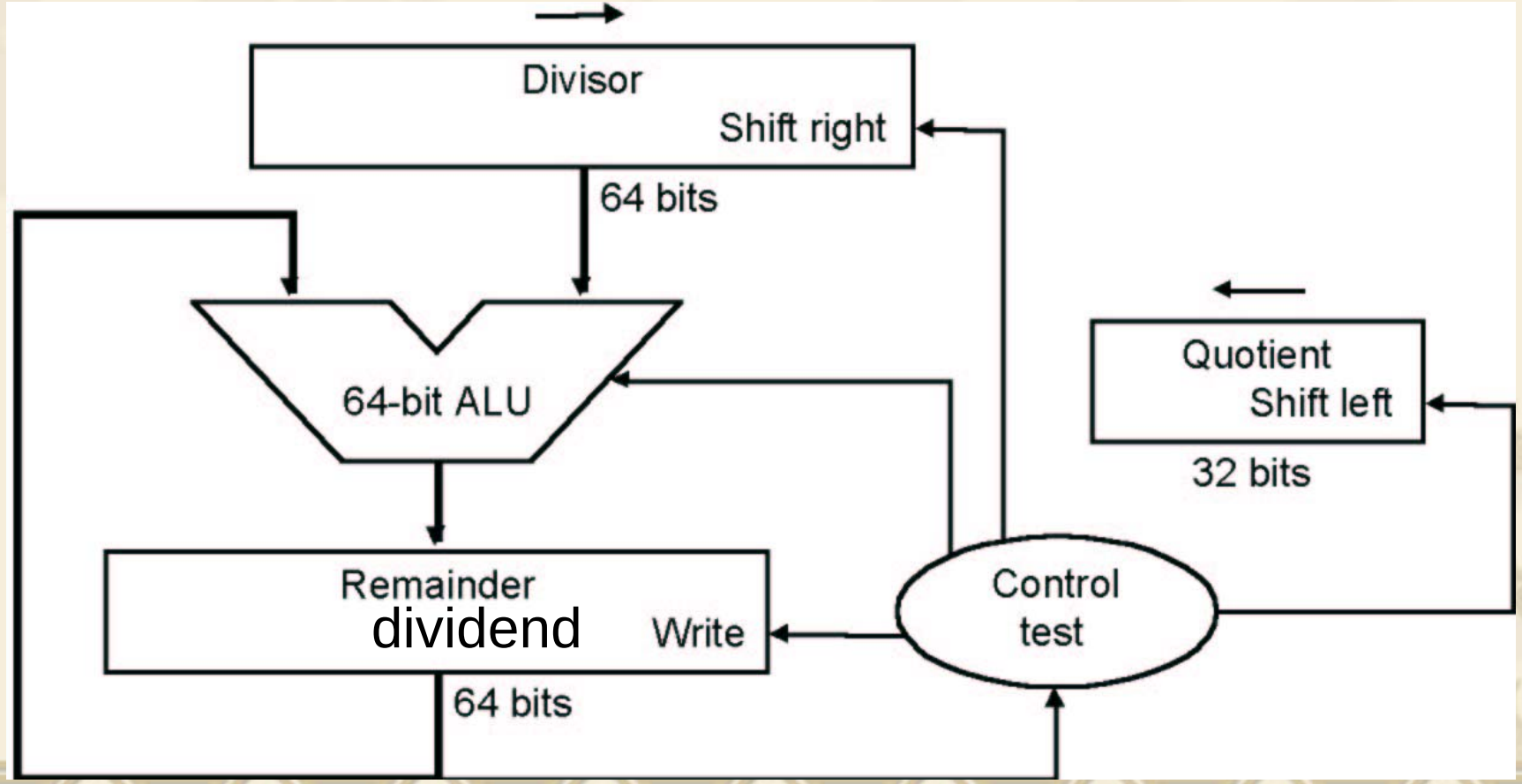
iteration	step	Multiplicand	product
0	Initial Values	0010	0000 1101 0
1	1.c:10→Prod=Prod-Mcand	0010	1110 1101 0
	2: shift right Product	0010	1111 0110 1
2	1.b:01→Prod=Prod+Mcand	0010	0001 0110 1
	2: shift right Product	0010	0000 1011 0
3	1.c:10→Prod=Prod-Mcand	0010	1110 1011 0
	2: shift right Product	0010	1111 0101 1
4	1.d: 11 → no operation	0010	1111 0101 1
	2: shift right Product	0010	**1111 1010** 1

3.5 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

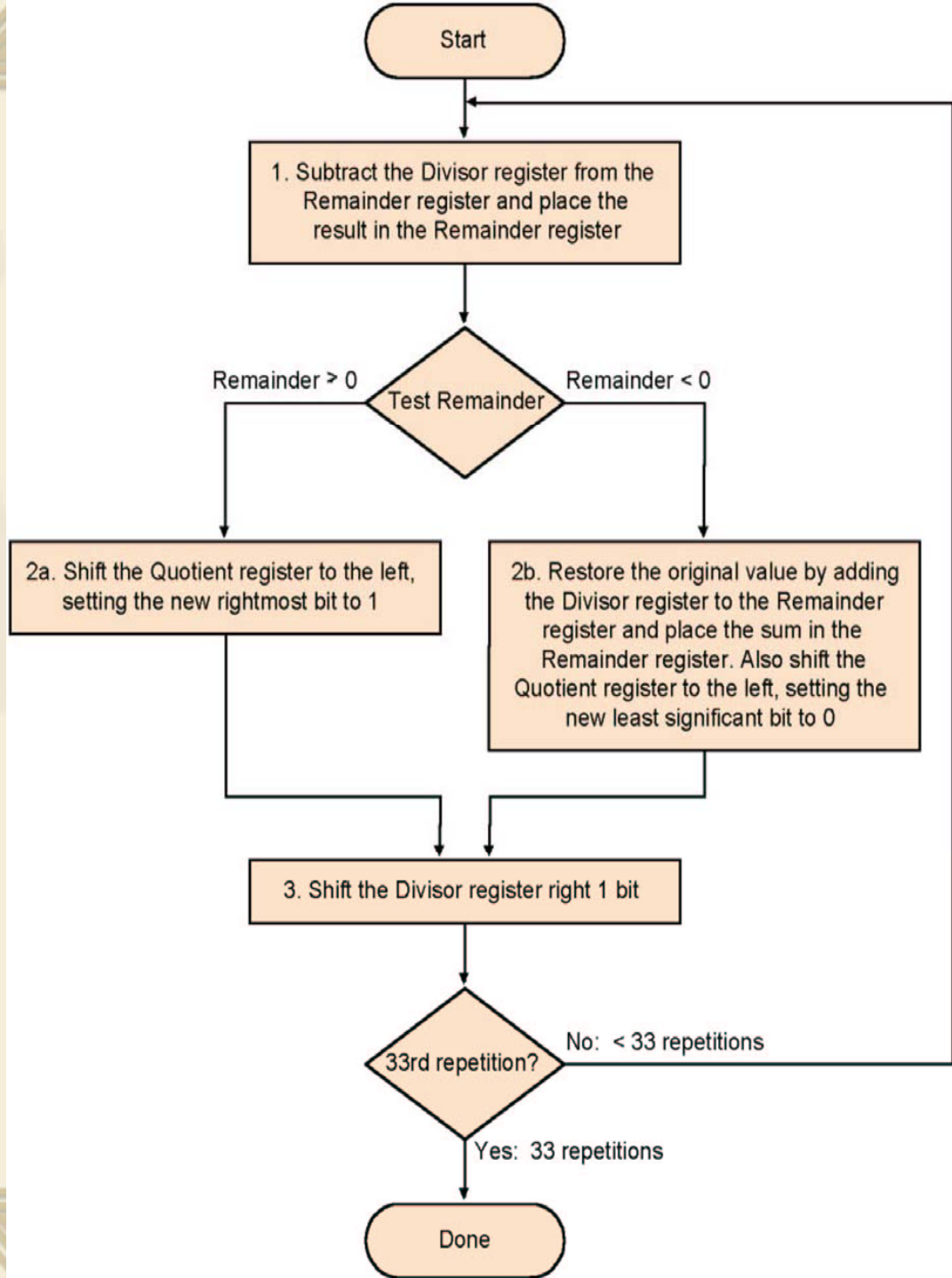
Division V1 --Logic Diagram

- 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 V 1

- Each step:
  - Subtract divisor
  - Depending on Result
    - Leave or
    - Restore
  - Depending on Result
    - Write '1' or
    - Write '0'



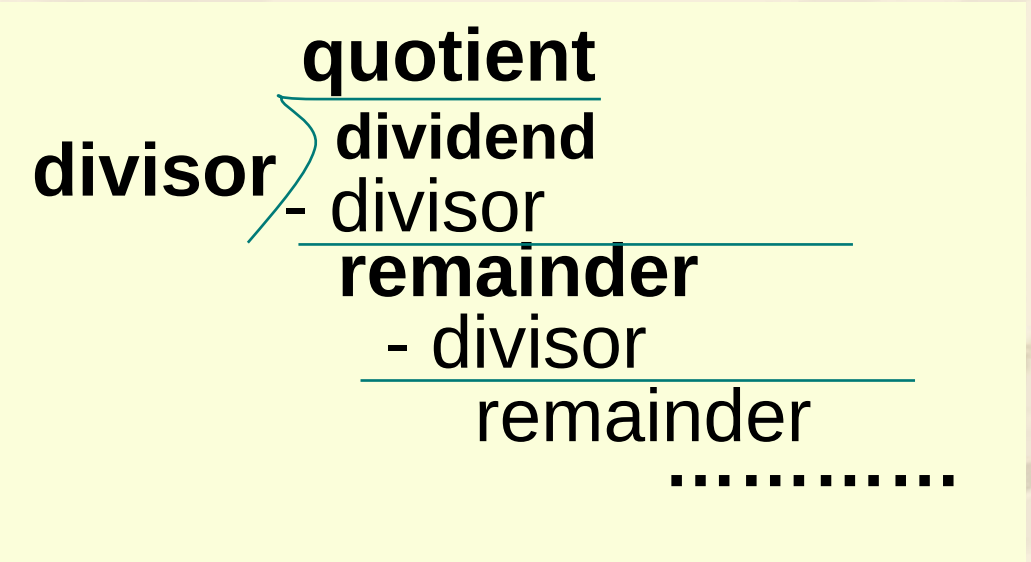
Example 7/2 for Division V1

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	0011	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

Two questions

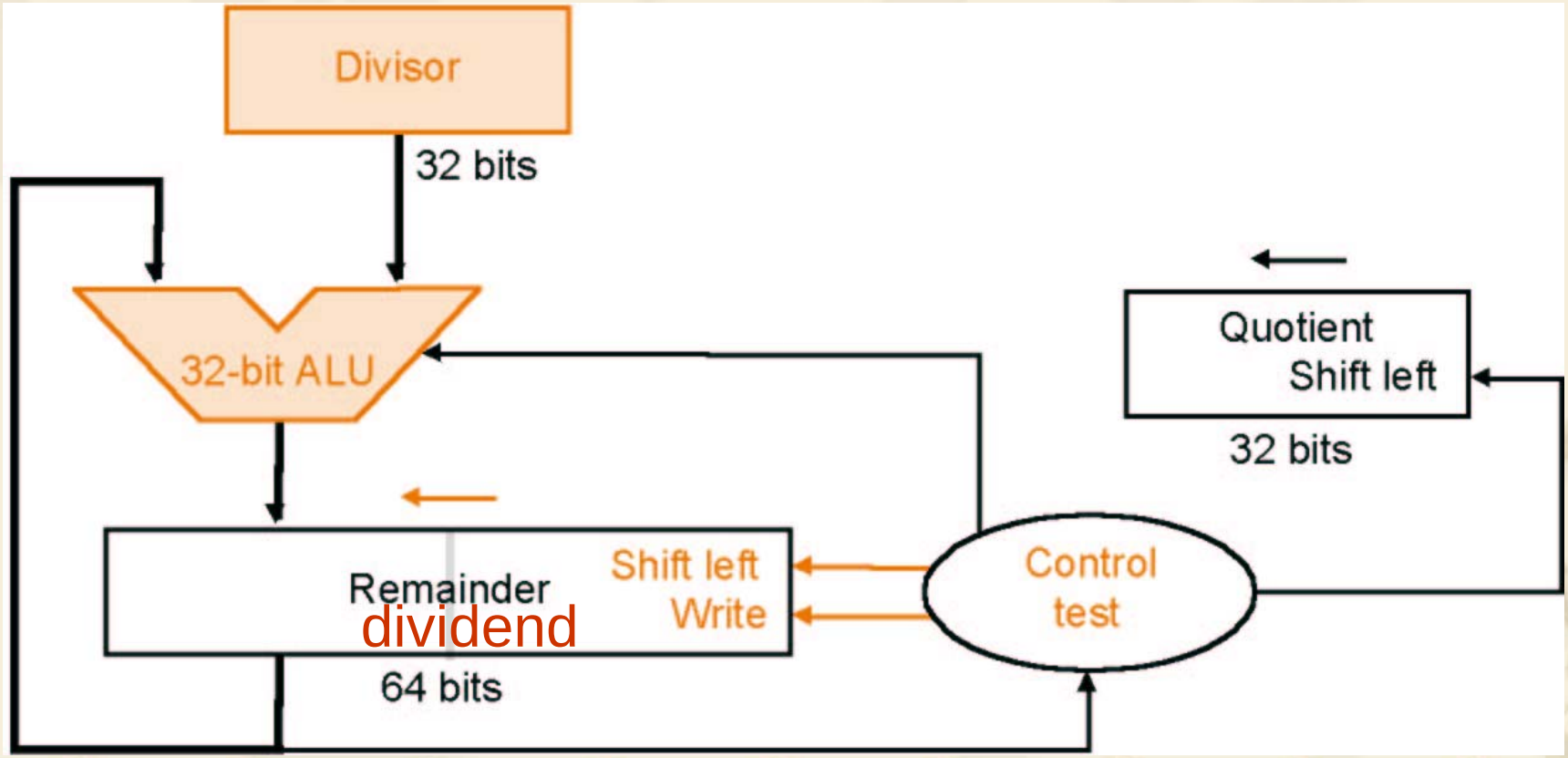
1.Why should the divisor be shifted right one bit each time?

2.Why should the divisor be placed in the left half of the divisor register,and the dividend be placed in the right half of the remainder register at first ?



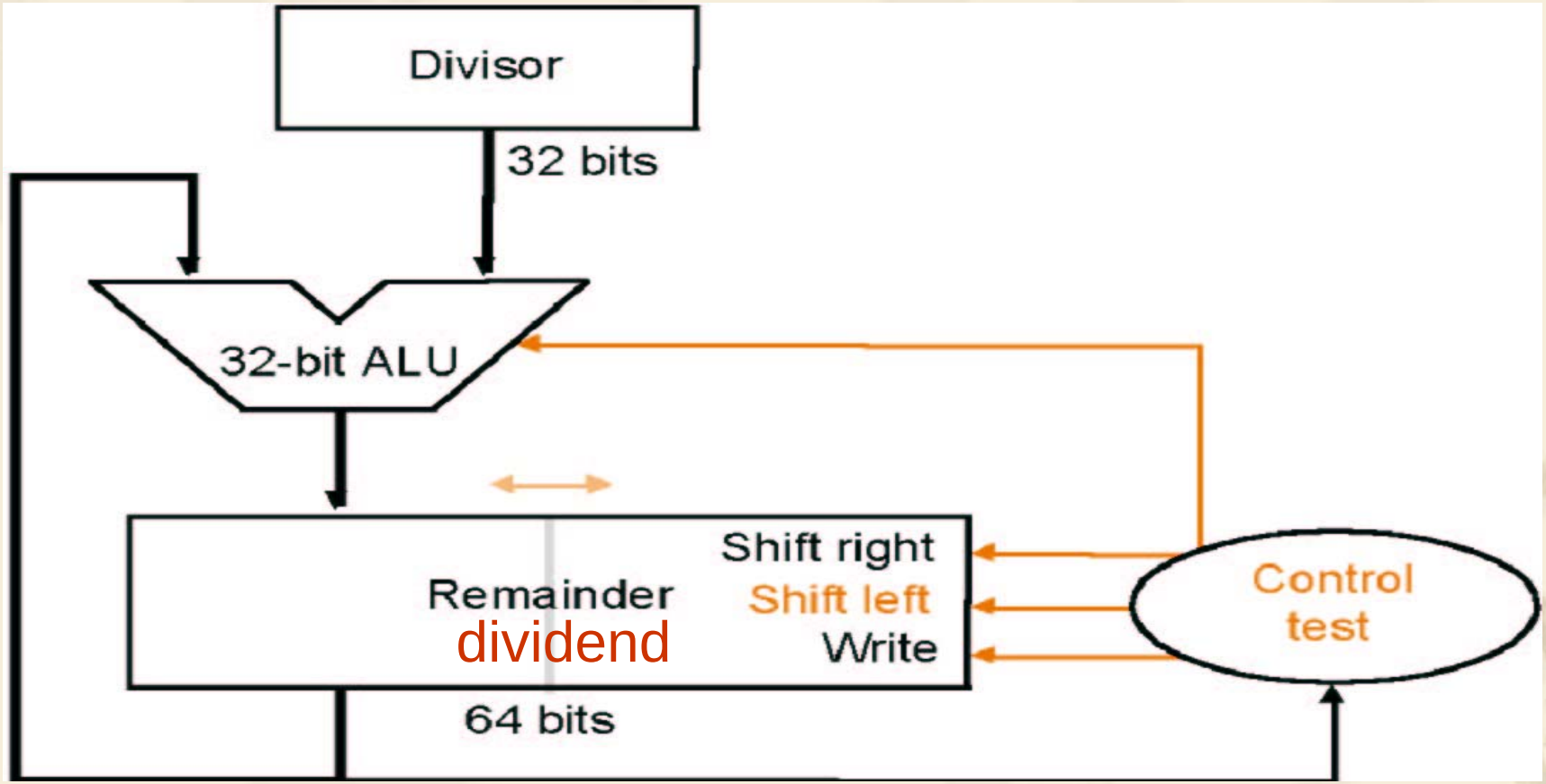
Division V 2

- Reduction of Divisor and ALU width by half
- Shifting of the remainder
- Saving 1 iteration



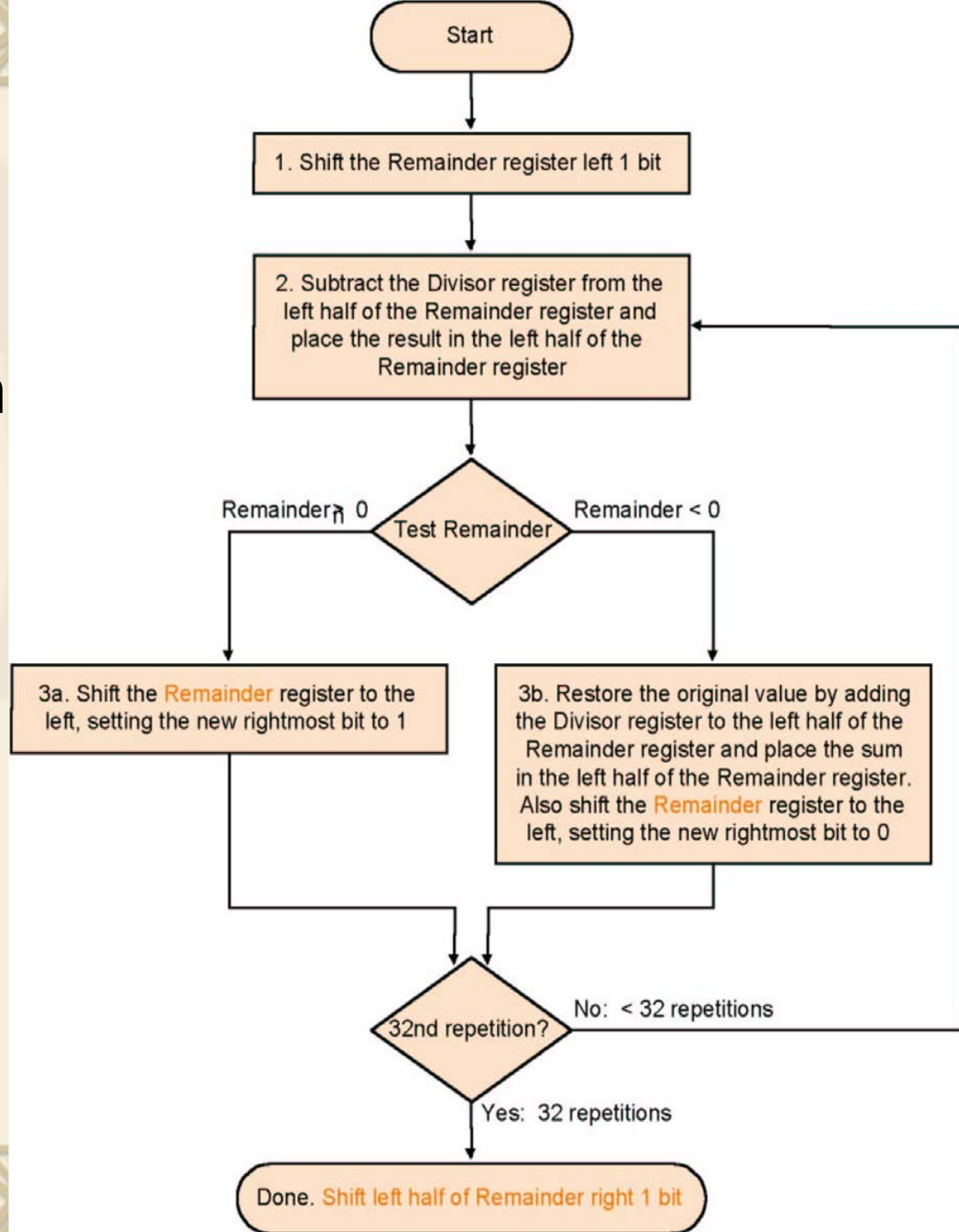
Division V 3

- Remainder register keeps quotient
  No quotient register required



Algorithm V 3

- Much the same than the last one
- Except change of register usage



Example 7/2 for Division V3

- Well known numbers: 0000 0111/0010

iteration	step	Divisor	Remainder
0	Initial Values	0010	0000 0111
	Shift Rem left 1	0010	**0000 1110**
1	1.Rem=Rem-Div	0010	1110 1110
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0001 1100**
2	1.Rem=Rem-Div	0010	1111 0110
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0011 1000**
3	1.Rem=Rem-Div	0010	0001 1000
	2a: Rem>0 →sll R,R_0=1	0010	**0011 0001**
4	1.Rem=Rem-Div	0010	0001 0011
	2a: Rem>0 →sll R,R_0=1	0010	**0010 0011**
	Shift left half of Rem right 1		

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
- One 64 bit register : Hi & Lo
  - Hi: Remainder, Lo: Quotient
- Instructions: div, divu
- Divide by 0 → overflow : Check by software

3.6 Floating point numbers

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

Floating point numbers

- Form
  - Arbitrary $363.4 \cdot 10^{34}$
  - Normalised $3.634 \cdot 10^{36}$
- Binary notation
  - Normalised $1.xxxxxx \cdot 2^{yyyyy}$
- Standardised format IEEE 754
  - Single precision 8 bit exp, 23 bit significand
  - Double precision 11 bit exp, 52 bit significand
- Both formats are supported by MIPS

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

  ->saves one bit
- Exponent is biased:

  00...000 smallest exponent

  11...111 biggest exponent
  - Bias 127 for single precision
  - Bias 1023 for double precision
- Summary:

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

IEEE 754 standard

- Rounding: four rounding modes
  - Round to next even number (default)
  - Round to 0
  - Round to +∞
  - Round to -∞
- Special numbers: NaN, +∞ , -∞
- denormal numbers for results smaller $1.0 \cdot 2^{Emin}$
- Mechanisms for handling exceptions

Example

- Show the binary representation of -0.75 in IEEE single precision format
- Decimal representation: -0.75 = - 3/4 = - $3/2^2$
- Binary representation: - 0.11 = - 1.1 • 2^{-1}
- Floating point
  - $(-1)^{sign}$ • (1 + fraction) • $2^{exponent - bias}$
  - $(-1)^{sign}$ =-1,so Sign = 1
  - 1+ fraction = 1.1,so Significand=.1
  - exponent -127 =-1,so Exponent=(-1 + 127) = 126

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

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 significands
- Normalisation of the result
- Rounding
- Example in decimal

  system precision 4 digits

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

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

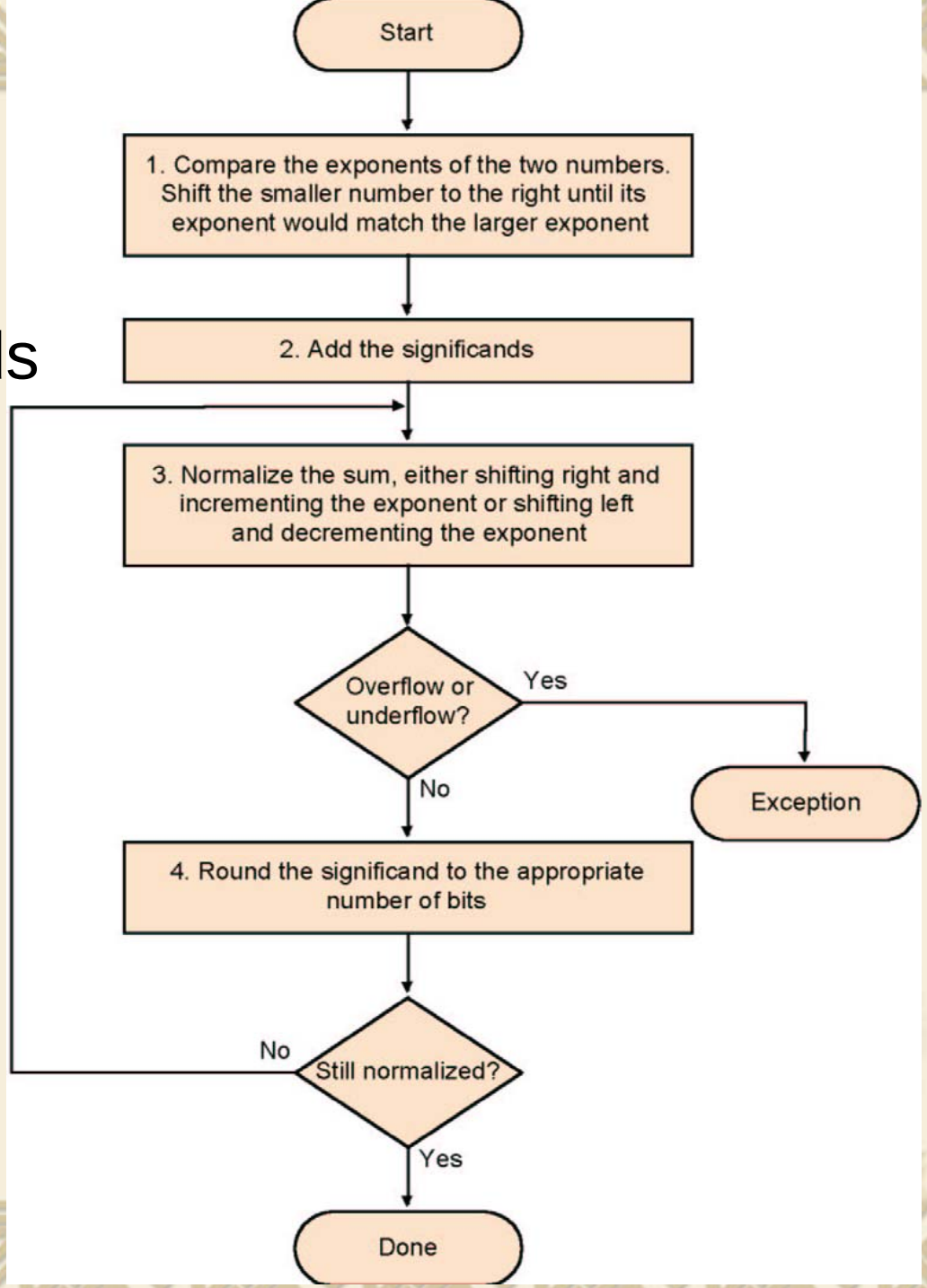
  $1.0015 \bullet 10^2$

- Rounding

  $1.002 \bullet 10^2$

Algorithm

- Normalise Significands
- Add Significands
- Normalise the sum
- Over/underflow
- Rounding
- Normalisation



Example y=0.5+(-0.4375) in binary

- $0.5_{10} = 1.000_2 \times 2^{-1}$
- $-0.4375_2 = -1.110_2 \times 2^{-2}$
- 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

  $\mathbf{1.000_2 \times 2^{-1}}$

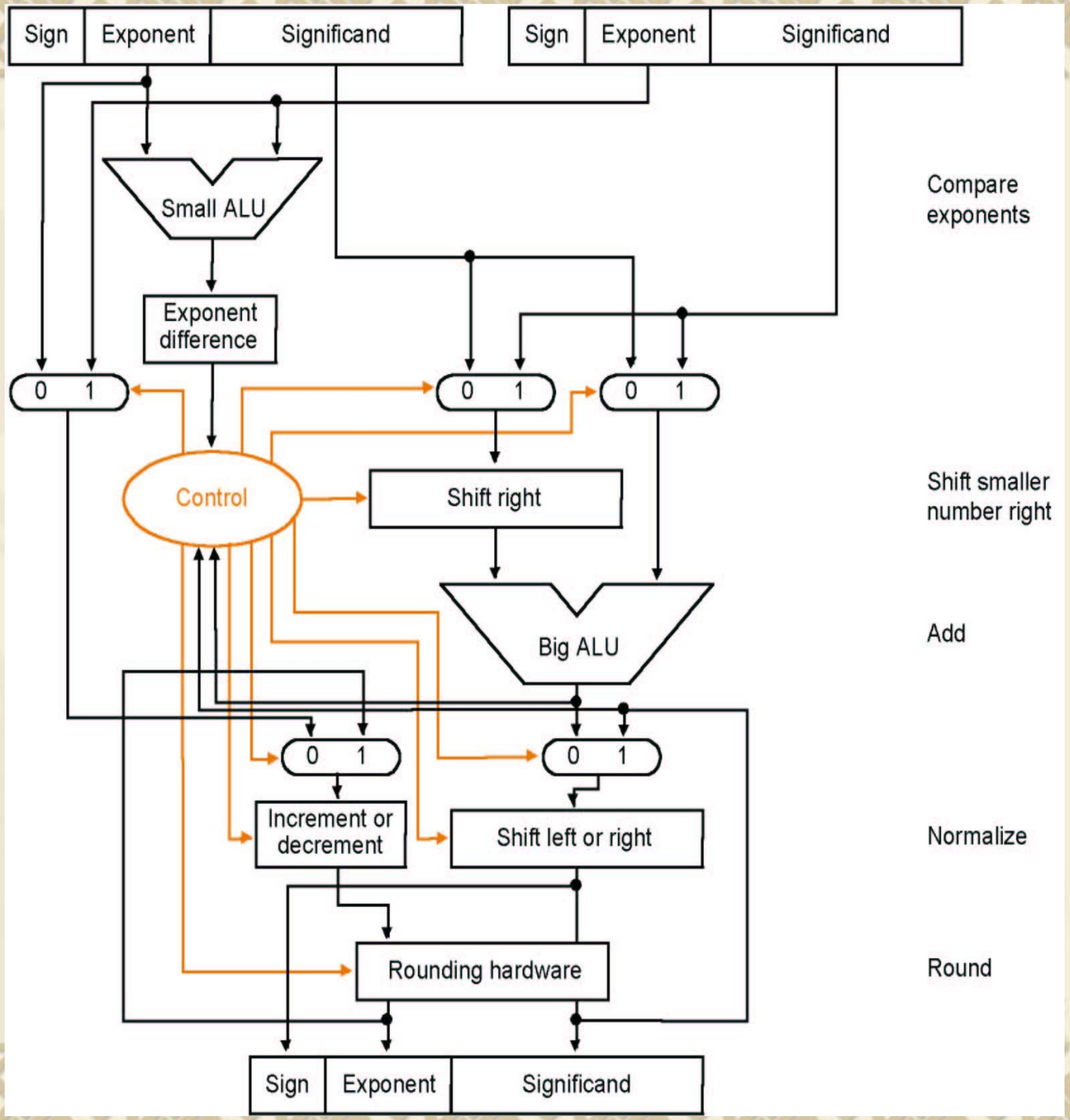
  $\mathbf{+) - 0.111_2 \times 2^{-1}}$

  $\mathbf{0.001_2 \times 2^{-1}}$
- Step3: Normalize the sum and checking for overflow or underflow

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

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

Algorithm


Sign
Exponent
Significand
Sign
Exponent
Significand
Small ALU
Compare exponents
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

Multiplication

- Composition of number from different parts

  → 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 = -1×2^3

  0 10000011 000 0000 0000 0000 0000 0000 = 1×2^4

- Both significands are 1 → product = 1 →Sign=1
- Add the exponents, bias = 127

  10000010

  +10000011

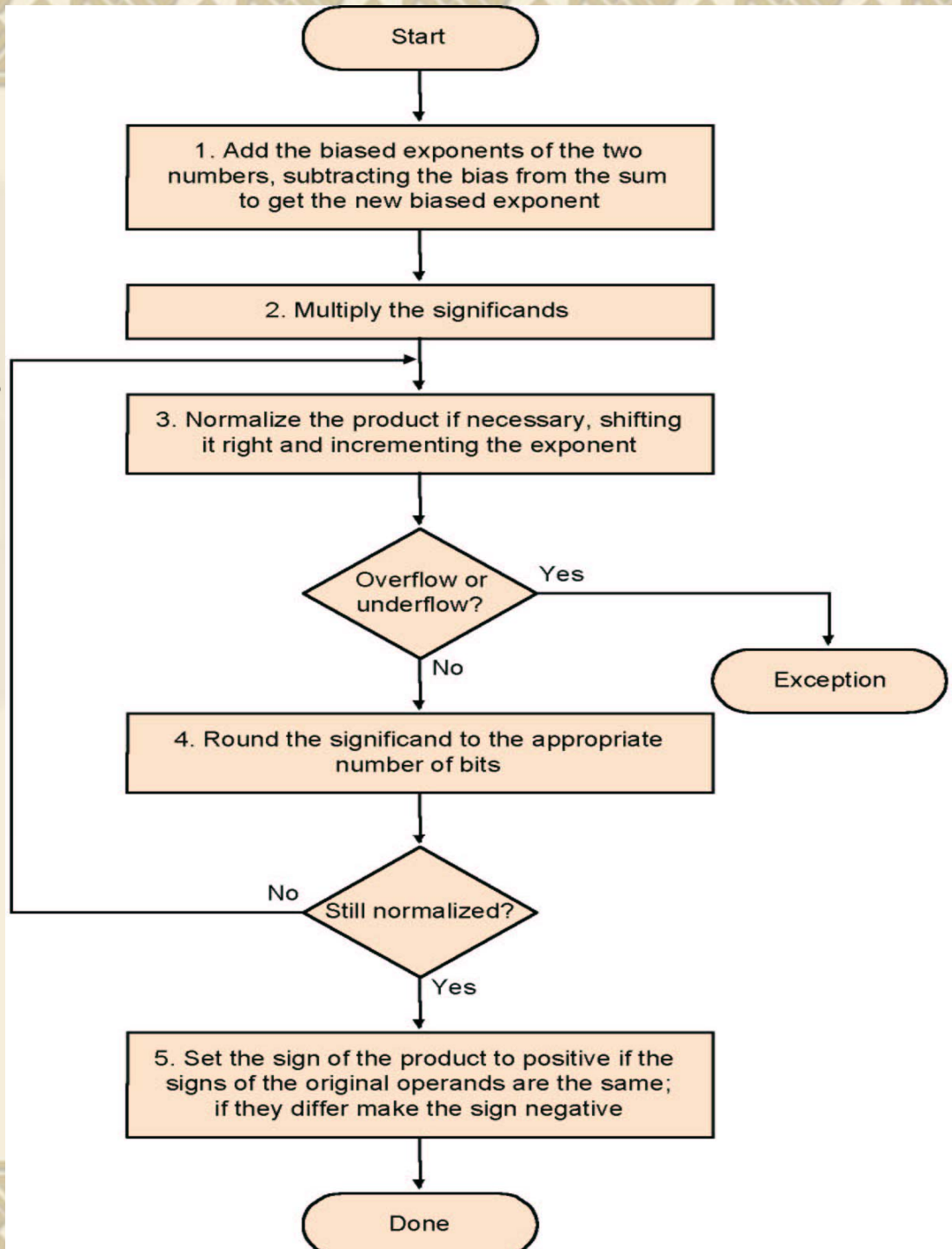
  110000101

  Correction: 110000101-01111111=10000110=134=127+3+4

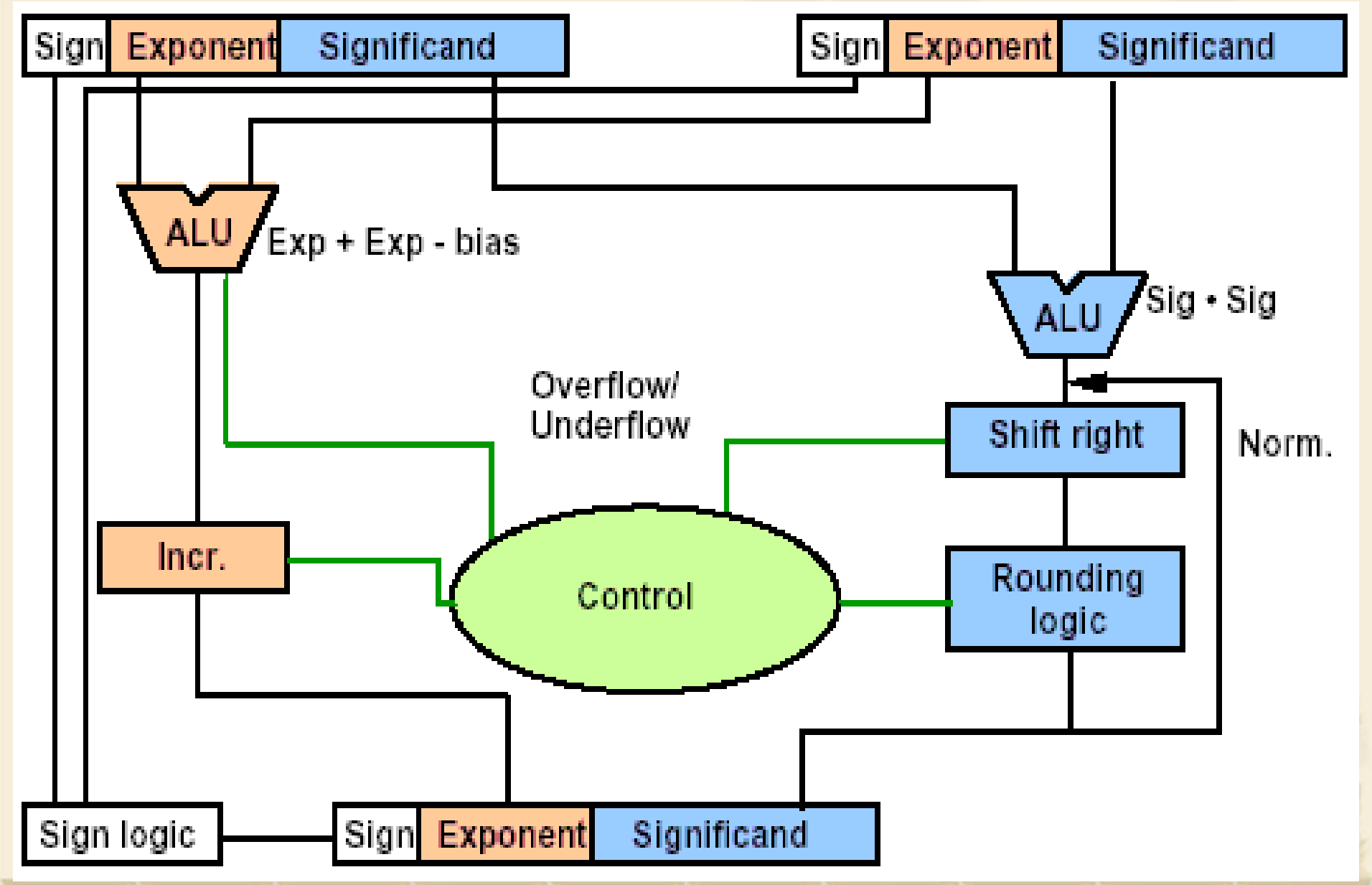
- The result: 1 10000110 000 0000 0000 0000 0000 0000 = -1×2^7

Multiplication

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



Data Flow


Sign
Exponent
Significand
Sign
Exponent
Significand
ALU
Exp + Exp - bias
ALU
Sig • Sig
Overflow/
Underflow
Shift right
Norm.
Incr.
Control
Rounding
logic
Sign logic
Sign
Exponent
Significand

Division-- Brief

- Subtraction of exponents
- Division of the significants
- Normalisation
- Runding
- Sign