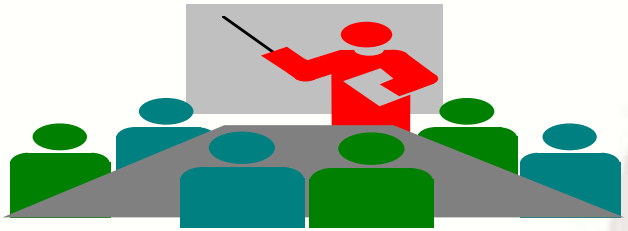


计算机组成与设计

Computer Organization & Design

The Hardware/Software Interface

Chapter 3

Arithmetic for Computer

Haifeng Liu
College of Computer Science and Technology, Zhejiang University
haifengliu@zju.edu.cn

Contents of Chapter 3

3.1 Introduction

- **Computer words are composed of bits;**
  - thus words can be represented as binary numbers
  - there are 32bits/word or 64bits/word in RISC-V
  - Contains four bytes for 32bits word
- **Simplified to contain only in course:**
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, xor, slt…`
  - control flow instructions: `beq, bne, jal`
- **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
- **How do we represent 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

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

Numbers and their representation

Number systems

- Radix based systems are dominating decimal, octal, binary,... $0 \le A \le K$

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

MSD LSD

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

- ***A***: 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

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

Signed 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 & 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 $x = \pm x_{n-1}x_{n-2}x_{n-3}x_{n-4}\ldots x_{0}$

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

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

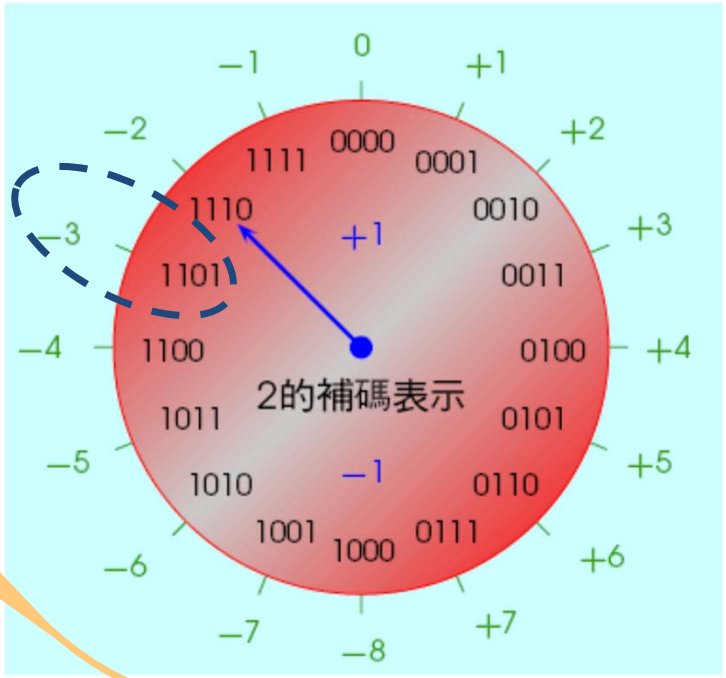
- **Converting n bit numbers into numbers with more than n bits:**
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits

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

2's complement for n=3

The absolute value of the sum of two=2^n

$2^{n+1} + X = 2^{n+1} - |X|$
$= 2^4 - 8$
$= (10000-1000)_2$
$= 01000$



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

biase

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

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

sign extension (lbu vs. lb)

- **Expansion**

  e.g. 8 bit numbers to 64/32 bit numbers

- **Required for operations with registers(32/64 bits) and immediate operands (8 bits)**
- **Sign extension**
  - Take the lower 8 bits as they are
  - Copy the highest bit to the remaining 24/56 bits
  - 0000 0010 → 2

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

    1111 1111 1111 1111 1111 1111 1111 1110

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

$$\begin{array}{rll} & 1111\ 1111 & 255_{10} \\ + & 1111\ 1010 & 250_{10} \\ \hline 1 & 1111\ 1001 & 249_{10} \end{array}$$

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

  Numbers change

  sign and size

$$\begin{array}{rll} & 1000\ 0001 & -127_{10} \\ + & 1111\ 1110 & -2_{10} \\ \hline & 0111\ 1111 & +127_{10} \end{array}$$

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 of Overflow

- **An exception (interrupt) occurs**
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- **Signaling to application (Ada, Fortran)**
- **Ignore, don't always want to detect overflow**

note: `sltu, sltiu` ***for unsigned comparisons***

Overflow process

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

Arithmetic for Multimedia

- **Graphics and media processing operates on vectors of 8-bit and 16-bit data**
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- **Saturating operations**
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

Logical operations

- **Logical shift operations**
  - right (srl)
  - left (sll) ***Filled with '0'***
- **The machine instruction for the instruction**

  slli x11, x9, 3
- **Example:** slli x11, x9, 3
  - x9: 0000 0000 0000 0000 1100 1000 0000 1111
  - x11: 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:**

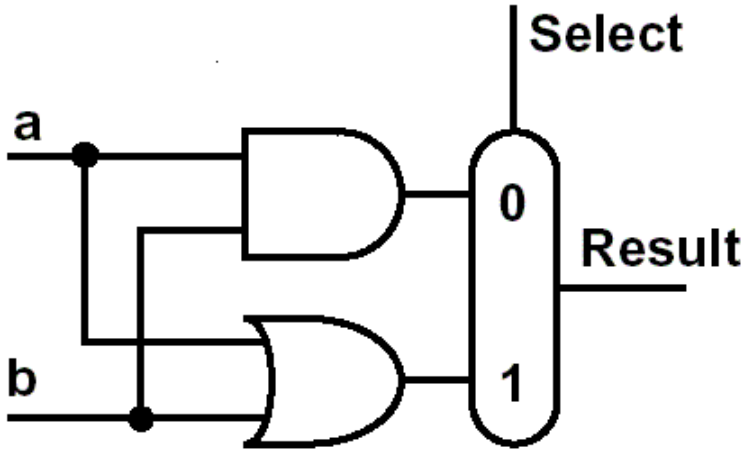
  and x3, x10, x16

  or x4, x10, x16

  - x16: 0000 0000 0000 0000 1100 1000 0000 1111
  - x10: 0000 0000 0000 0110 0100 0000 0111 1000
  - x3: 0000 0000 0000 0000 0100 0000 0000 1000
  - x4: 0000 0000 0000 0110 1100 1000 0111 1111

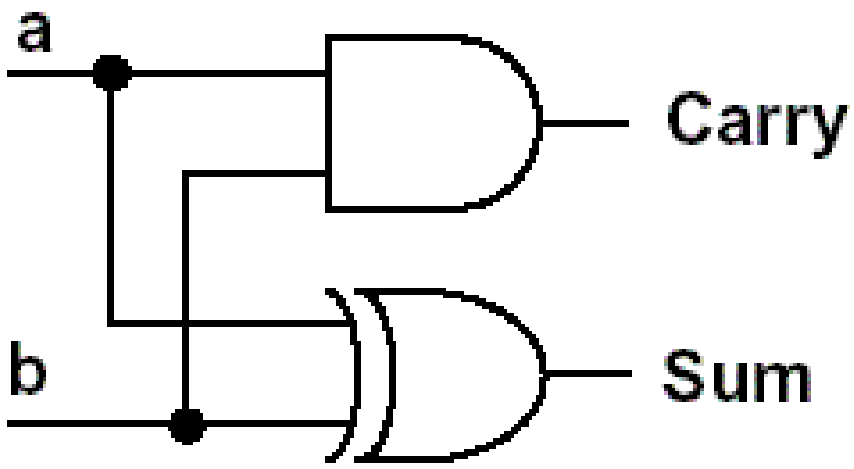
Constructing an ALU

- **Two methods constitute the ALU**
  - extended the adder
  - Parallel redundant select
- **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 = $\bar{a}$ b + a $\bar{b}$
- Carry = a b



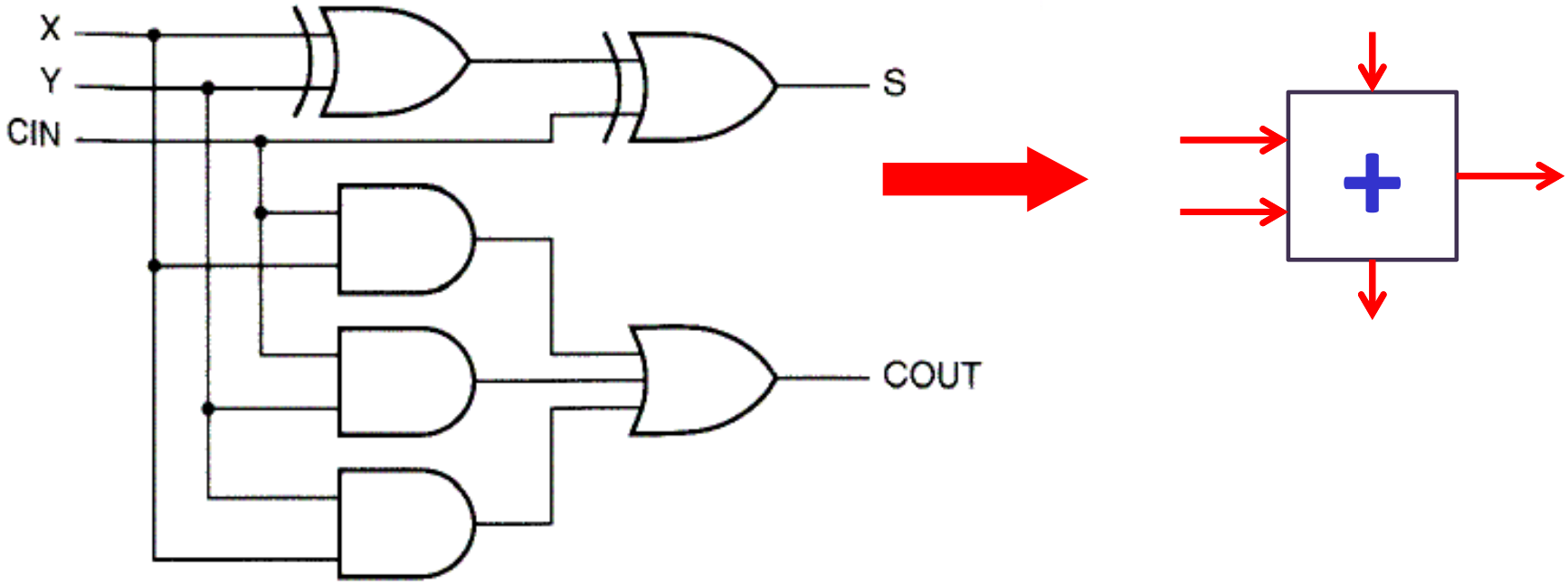
A full adder

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

Inputs			Outputs		Comments
A	B	Carry_{In}	Carry_{Out}	Sum	
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

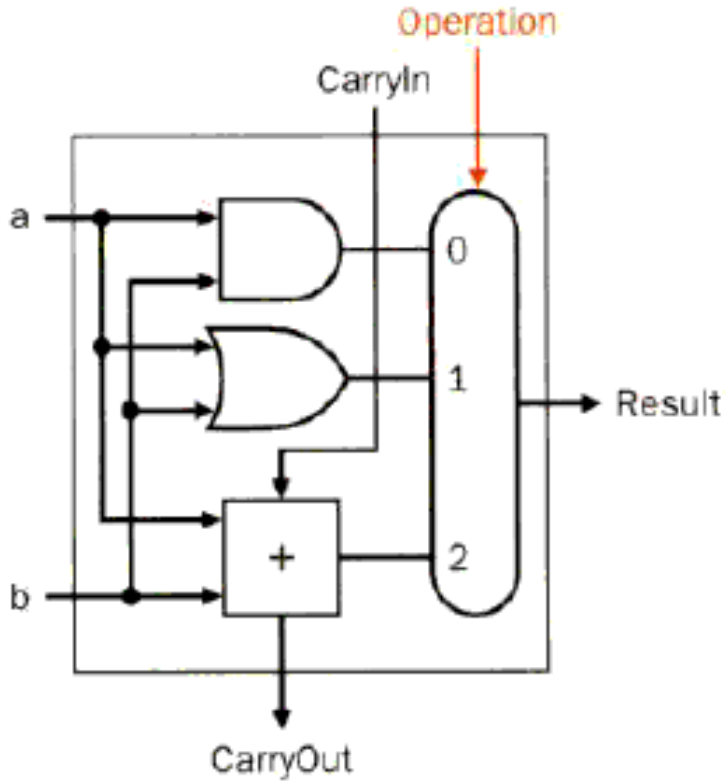
- Full adder in 2-level design



1 bit ALU

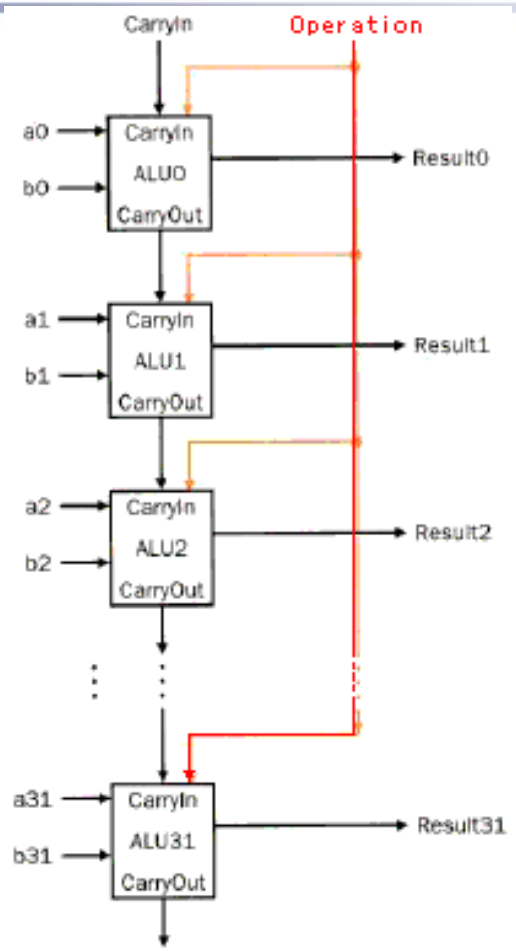
- **ALU**
  - AND
  - OR
  - ADD
- **Cell**

  Cascade Element



Basic 32 bit ALU

- Inputs parallel
- Carry is cascaded
- Ripple carry adder
  行波进位加法器
- Slow, but simple
- 1st Carry In = 0

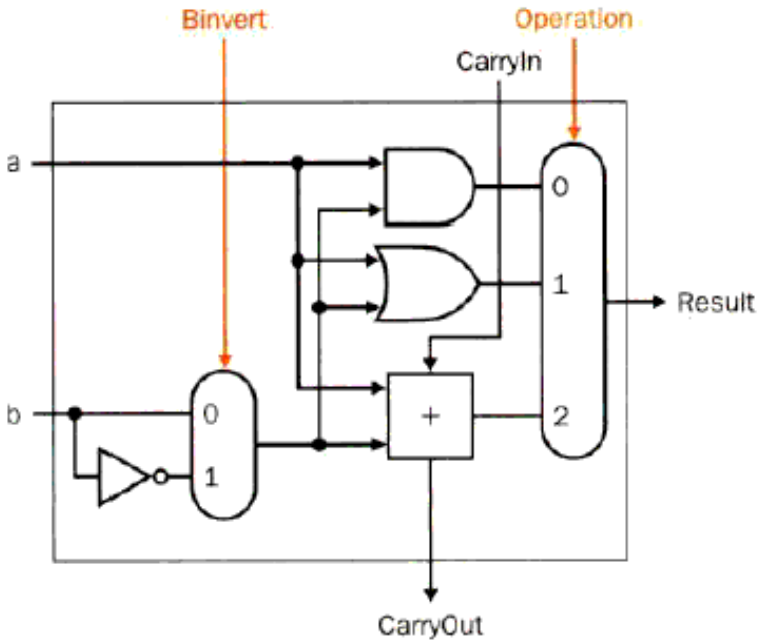


Extended 1 bit ALU-- Subtraction

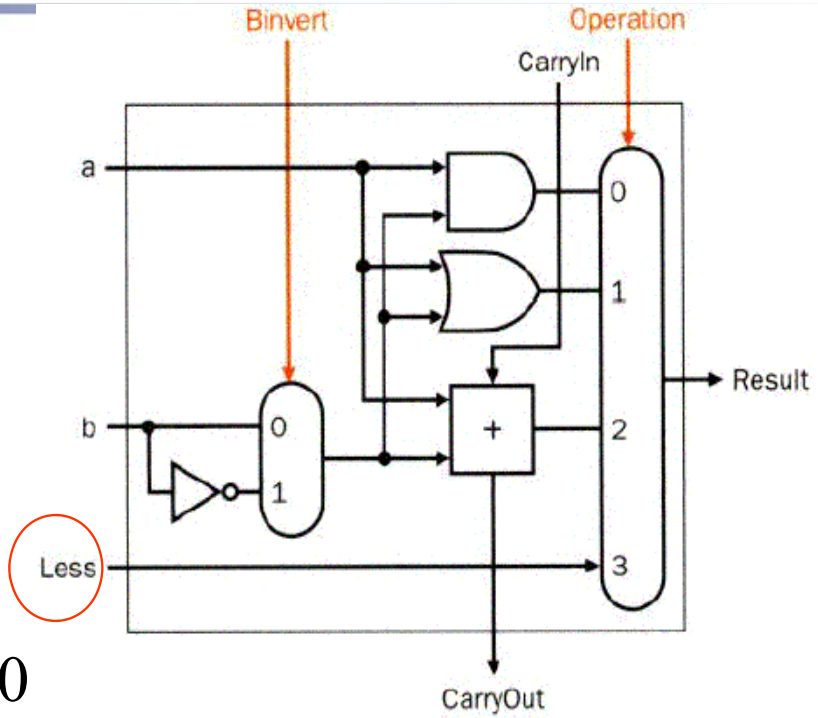
- **Subtraction**

  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
  - For rd, all bits = 0 except the least significant
  - Subtraction (rs - rt), if the result is negative→ rs < rt
  - **Use of sign bit as indicator**

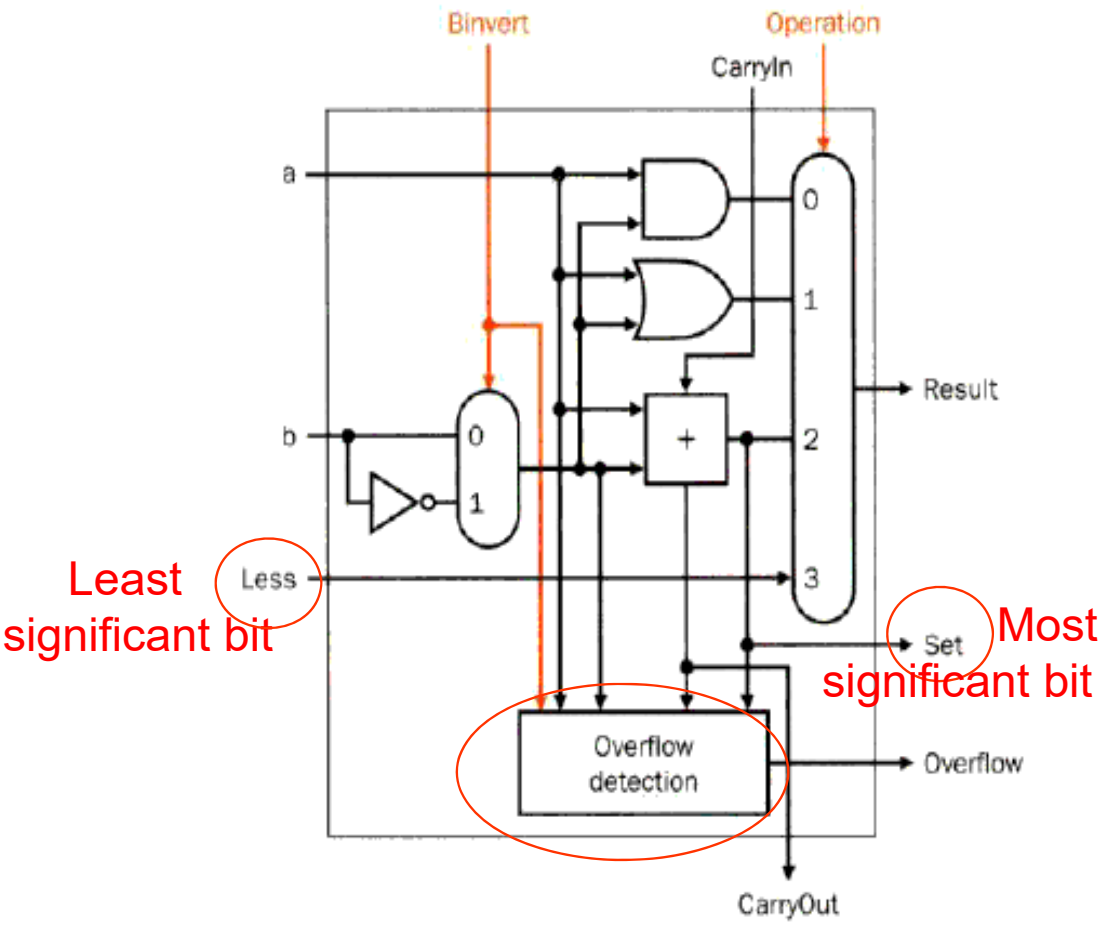
Most significant bit

- Set for comparison
- Overflow detect

Cell

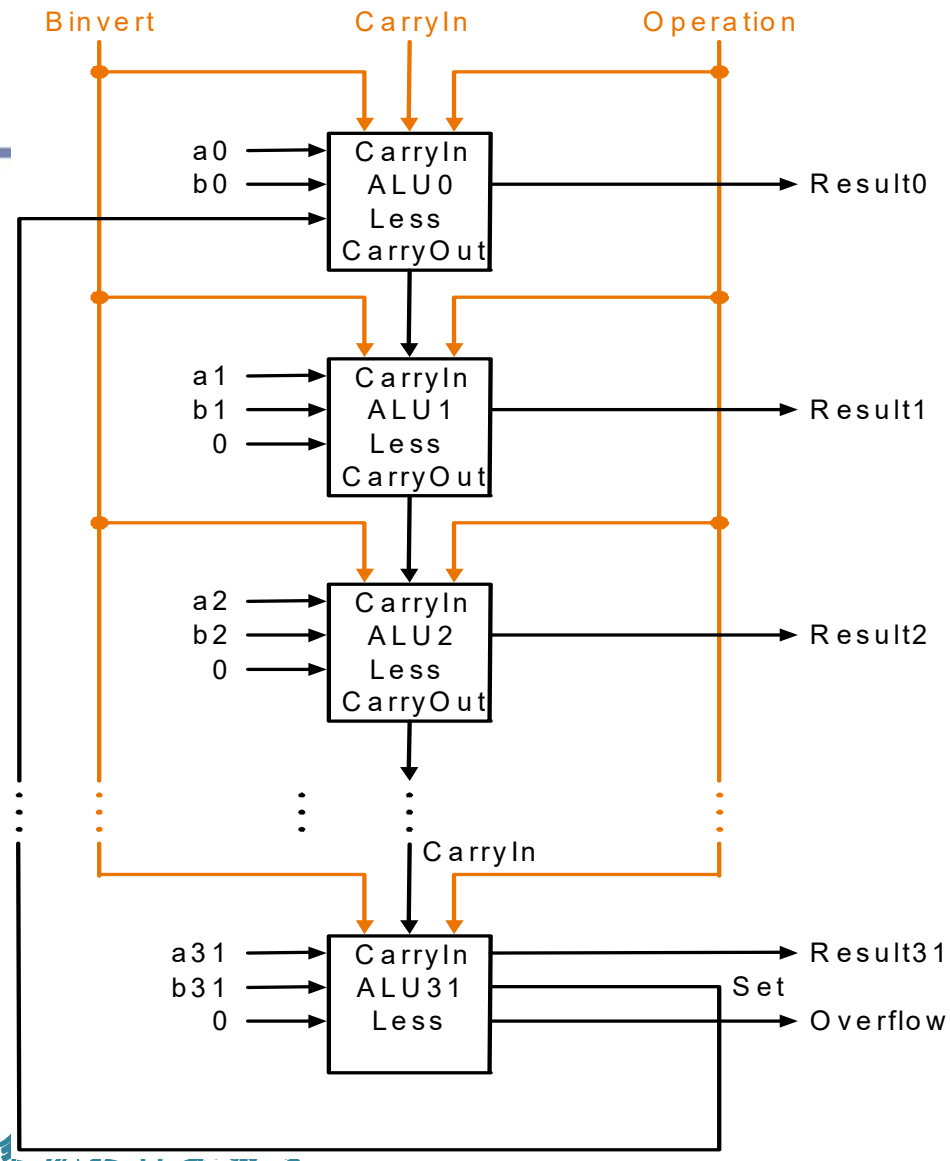
Cascade Element

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)*



Complete ALU

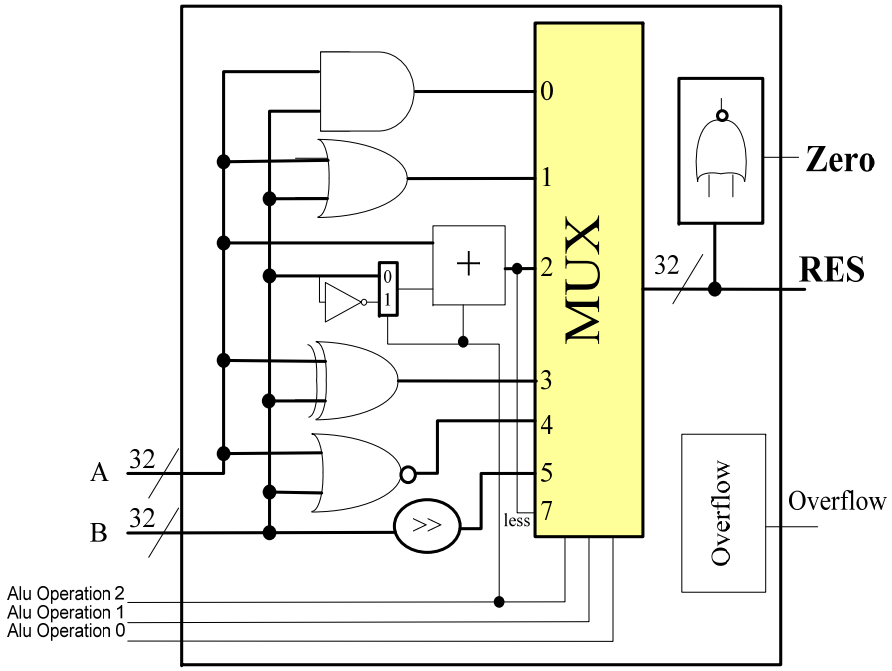
- **Input**
  - A、B
- **Control lines**
  - Binvert
  - Operation
  - Carry in
- **Output**
  - Result
  - Overflow
- **Slow, but simple**
  - Inputs parallel
  - Carry is cascaded
    - Ripple carry adder

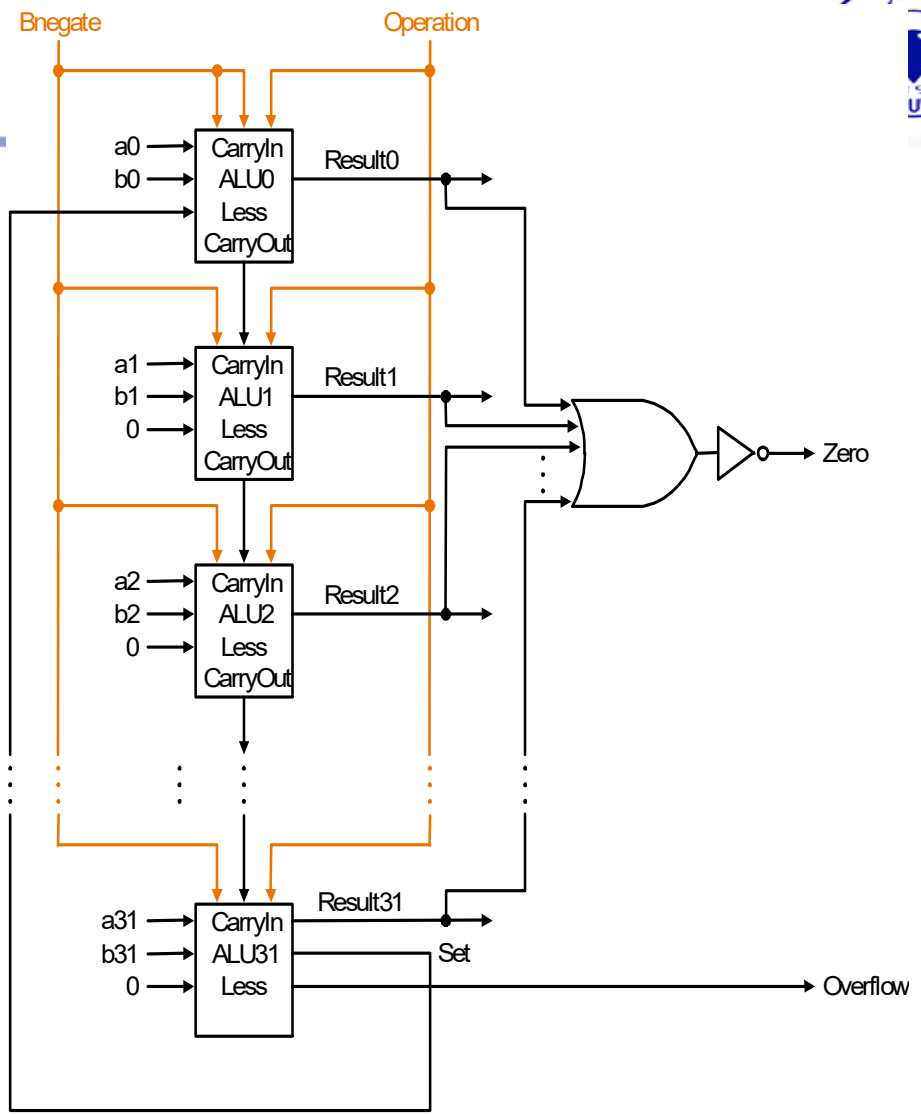


Iterative Circuit

Complete ALU —with Zero detector

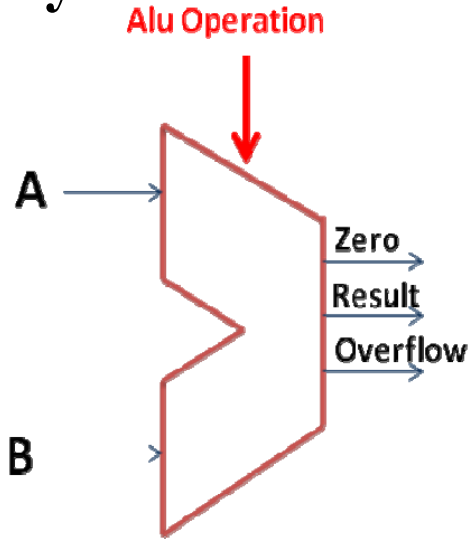
Add a Zero detector





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
100	nor
101	srl
011	xor

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
```

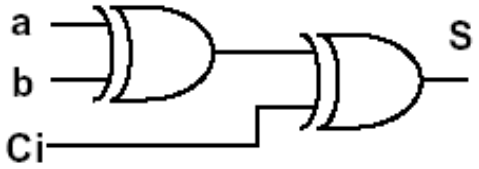
How do you write with overflow code ?

What is the difference The codes in the Synthesize?

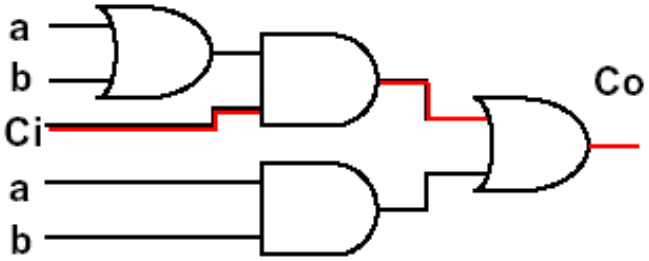
```
always @ (A or B or ALU_operation)
            case (ALU_operation)
                        3'b000: res=A&B;
                        3'b001: res=A|B;
                        3'b010: res=A+B;
                        3'b110: res=A-B;
                        3'b100: res=~(A | B);
 3'b111: res=(A < B)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

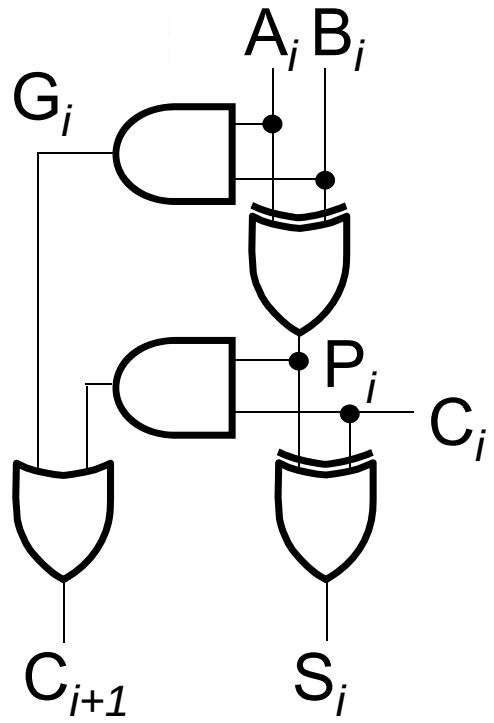
Speed considerations

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



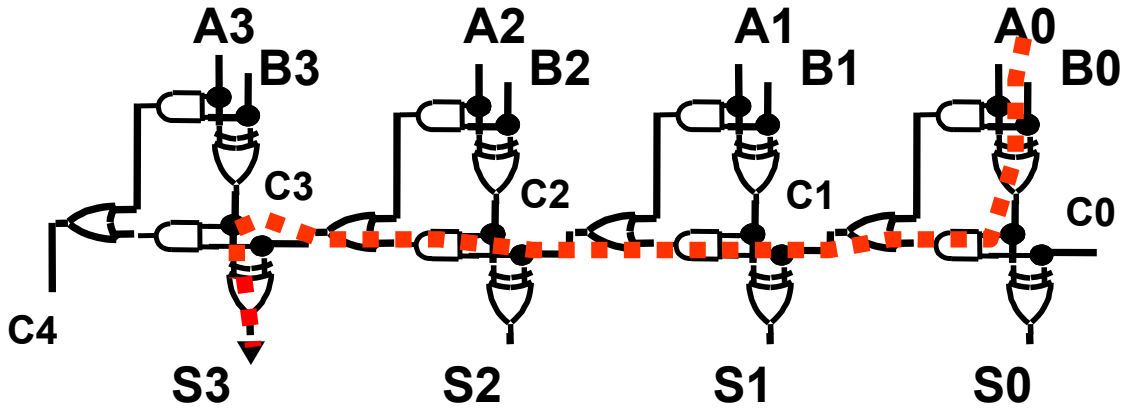
- Delay for the carry:
  - two - three units





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:**

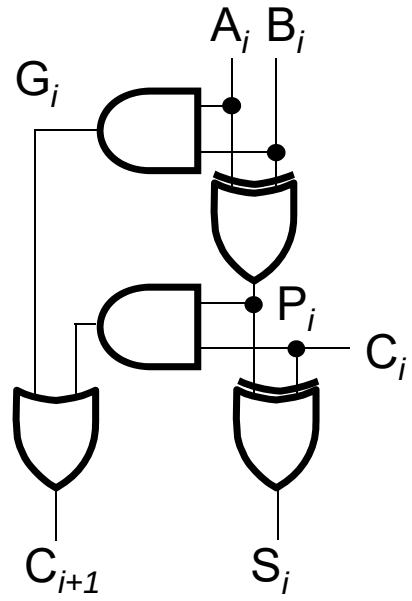
  Optimum between speed and size

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 Lookahead Adder (CLA)

- Given Stage *i* from a Full Adder, we know that there will be a carry generated when $A_i = B_i =$ "1", whether or not there is a carry-in
- Alternately, there will be a carry propagated if the "half-sum" is "1" and a carry-in, C_i occurs, then $C_{i+1}=1$
- These two signal conditions are called
  - *generate*, denoted as G_i
  - *propagate*, denoted as P_i



Addition formula in CLA

- **In the ripple carry adder:**
  - G_i, P_i, and S_i are local to each cell of the adder
  - C_i is also local each cell
- In the carry lookahead adder, in order to reduce the length of the carry chain, Ci is changed to a more global function spanning multiple cells
- Defining the equations for the Full Adder in term of the P_i and G_i:

$$P_i = A_i \oplus B_i \qquad G_i = A_i B_i$$

$$S_i = P_i \oplus C_i \qquad C_{i+1} = G_i + P_i C_i$$

Carry Lookahead Development

- C_{i+1} can be removed from the cells and used to derive a set of carry equations spanning multiple cells.
- Beginning at the cell 0 with carry in C_0:

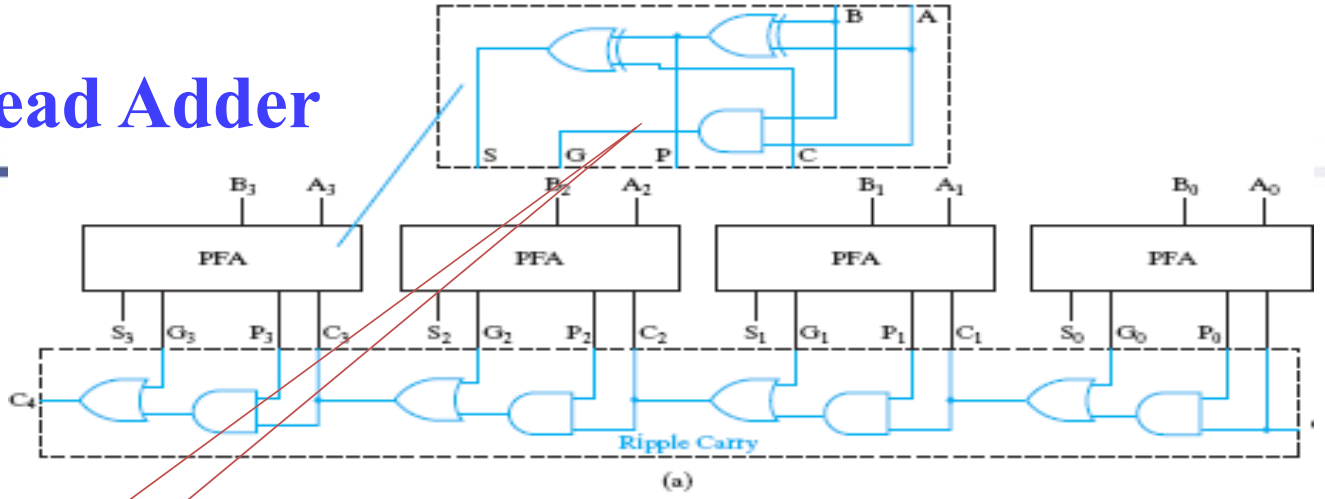
$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0)$$
$$= G_1 + P_1G_0 + P_1P_0 C_0$$

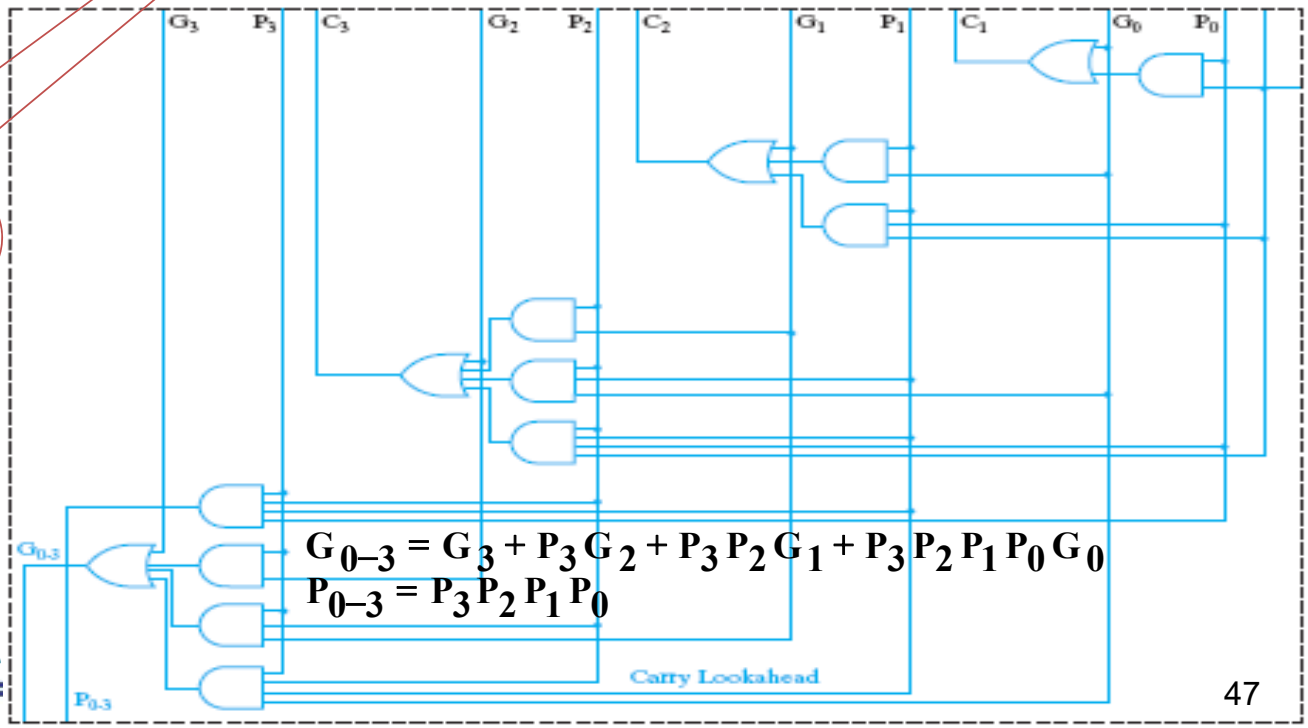
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0 C_0)$$
$$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0 C_0$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0 C_0$$

Carry Lookahead Adder



partial full adder



$$G_{0-3} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 P_0 G_0$$
$$P_{0-3} = P_3 P_2 P_1 P_0$$

Group Carry Lookahead Logic

- Last slide show shows the implementation of these equations for four bits. This could be extended to more than four bits; in practice, due to limited gate fan-in, such extension is not feasible.
- Instead, the concept is extended another level by considering *group generate* (G_{0-3}) and *group propagate* (P_{0-3}) functions:

$$G_{0-3} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 P_0 G_0$$
$$P_{0-3} = P_3 P_2 P_1 P_0$$

- Using these two equations:

$$C_4 = G_{0-3} + P_{0-3} C_0$$

- Thus, it is possible to have four 4-bit adders use one of the same carry lookahead circuit to speed up 16-bit addition

Extended Example: 16 carry lookahead adder

$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 = G_{0\sim3} + P_{0\sim3}C_0$

$C_8 = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4 + P_7P_6P_5P_4C_4 = G_{4\sim7} + P_{4\sim7}C_4$

$C_{12} = G_{11} + P_{11}G_{10} + P_{11}P_{10}G_9 + P_{11}P_{10}P_9G_8 + P_{11}P_{10}P_9P_8C_8 = G_{8\sim11} + P_{8\sim11}C_8$

$C_{16} = G_{15} + P_{15}G_{14} + P_{15}P_{14}G_{13} + P_{15}P_{14}P_{13}G_{12} + P_{15}P_{14}P_{13}P_{12}C_{12} = G_{12\sim15} + P_{12\sim15}C_{12}$

$= G_{12\sim15} + P_{12\sim15}(G_{8\sim11} + P_{8\sim11}(G_{4\sim7} + P_{4\sim7}(G_{0\sim3} + P_{0\sim3}C_0)))$

$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0\,C_0$

$G_{0\sim3} = G_3 + P_3G_2 + P_3P_2\,G_1 + P_3P_2P_1G_0$

$G_{4\sim7} = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4$

$G_{8\sim11} = G_{11} + P_{11}G_{10} + P_{11}P_{1}0G_9 + P_{11}P_{10}P_9G_8$

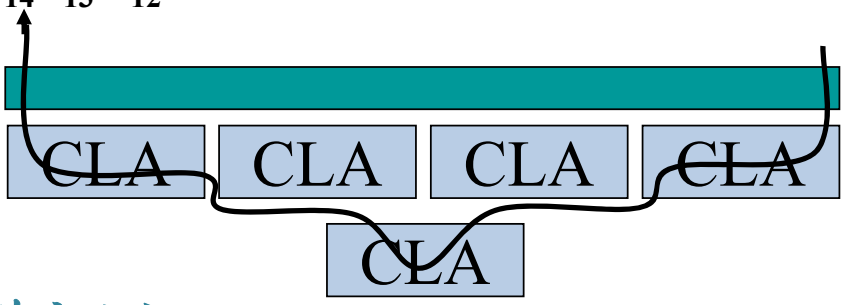
$G_{12\sim15} = G_{15} + P_{15}G_{14} + P_{15}P_{14}G_{13} + P_{15}P_{14}P_{13}G_{12}$

$P_{0\sim3} = P_3\ P_2\ P_1\ P_0$

$P_{4\sim7} = P_7\ P_6\ P_5\ P_4$

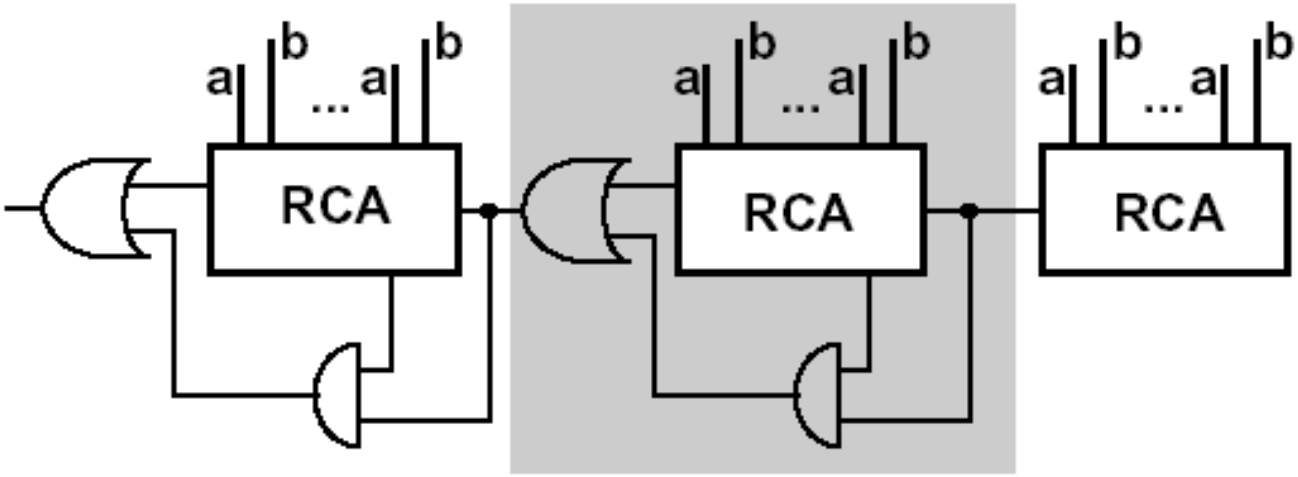
$P_{8\sim11} = P_{11}\ P_{10}\ P_9\ P_8$

$P_{12\sim15} = P_{15}\ P_{14}\ P_{13}\ P_{12}$



Carry skip adder

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



Carry skip adder

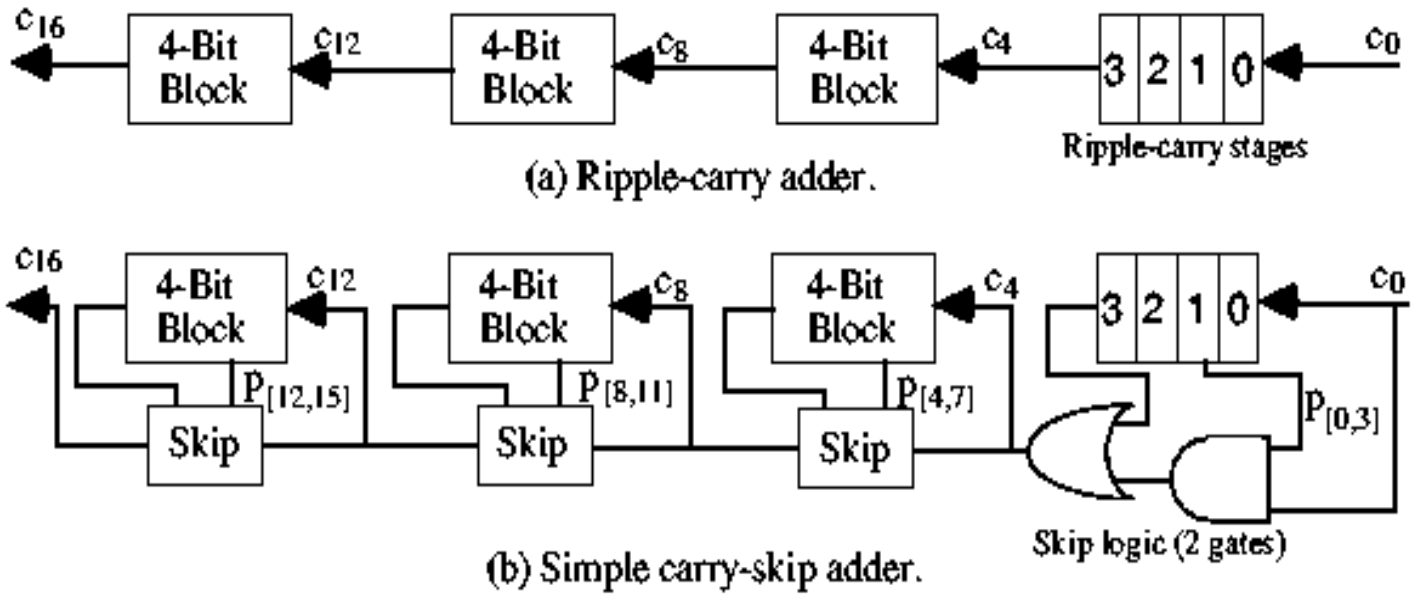
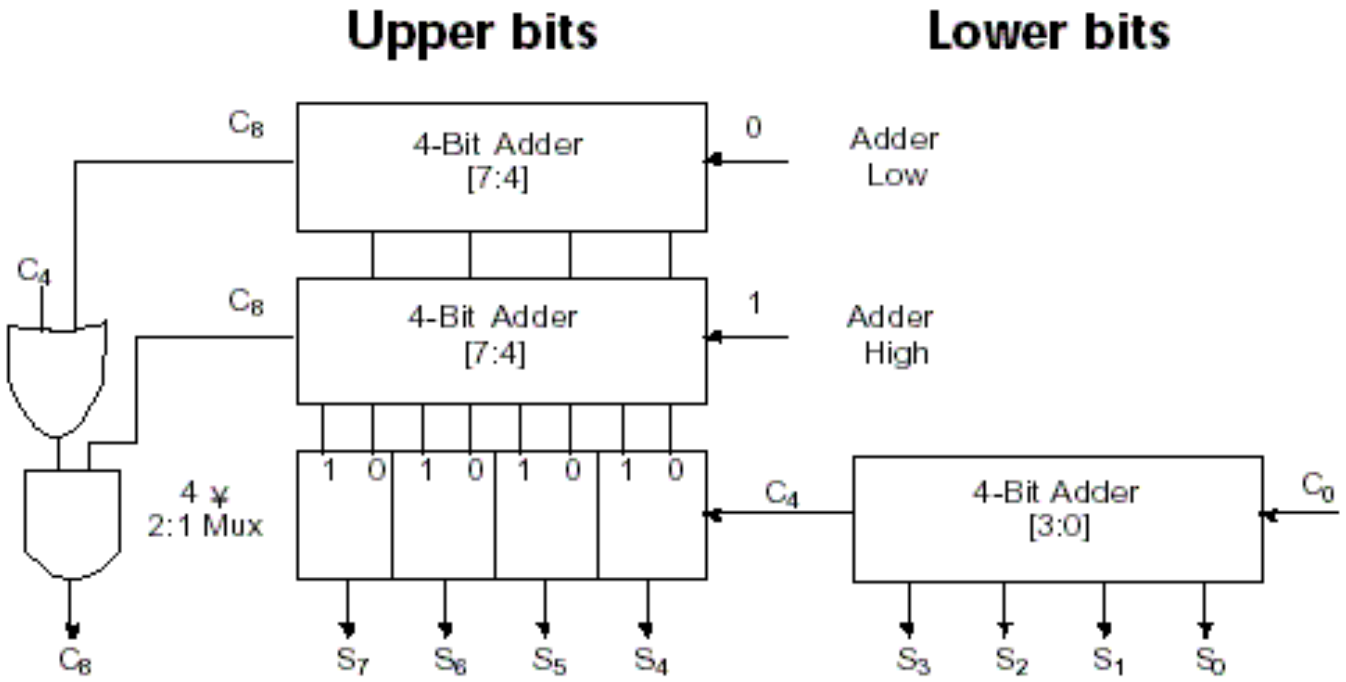


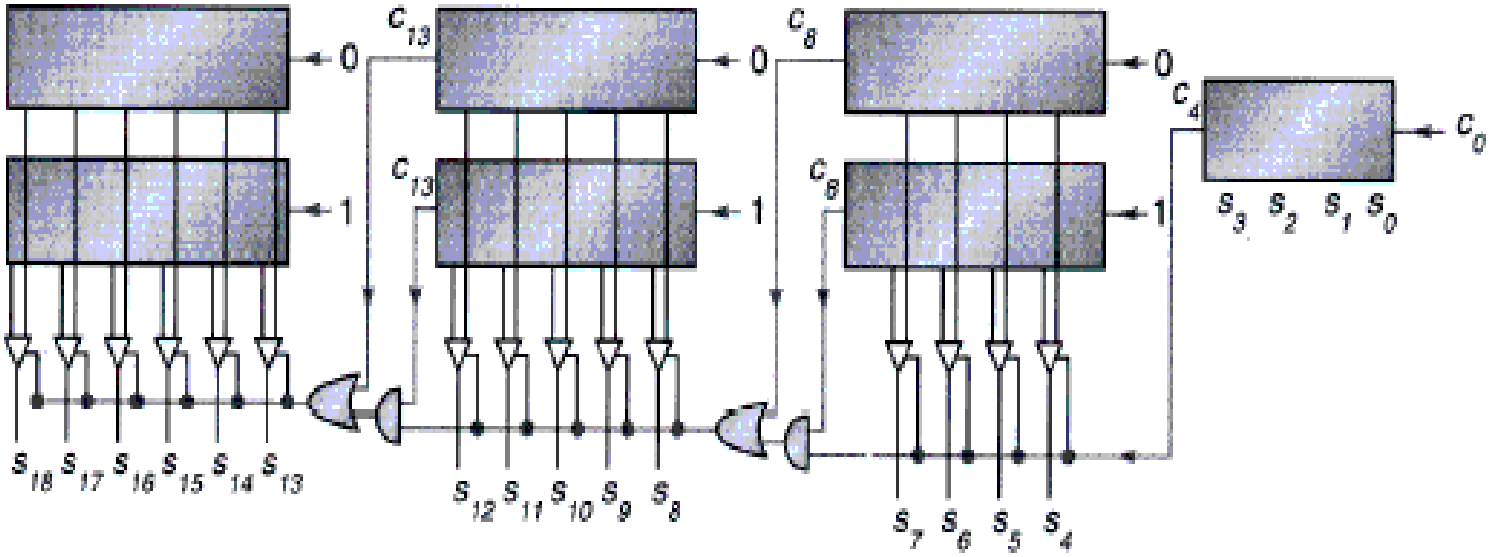
Fig. 7.1 Converting a 16-bit ripple-carry adder into a simple carry-skip adder with 4-bit skip blocks.

Carry select adder (CSA)



Carry select adder

- Carry selection by nibbles



3.4 Multiplication

- **Binary multiplication**

  Multiplicand × Multiplier

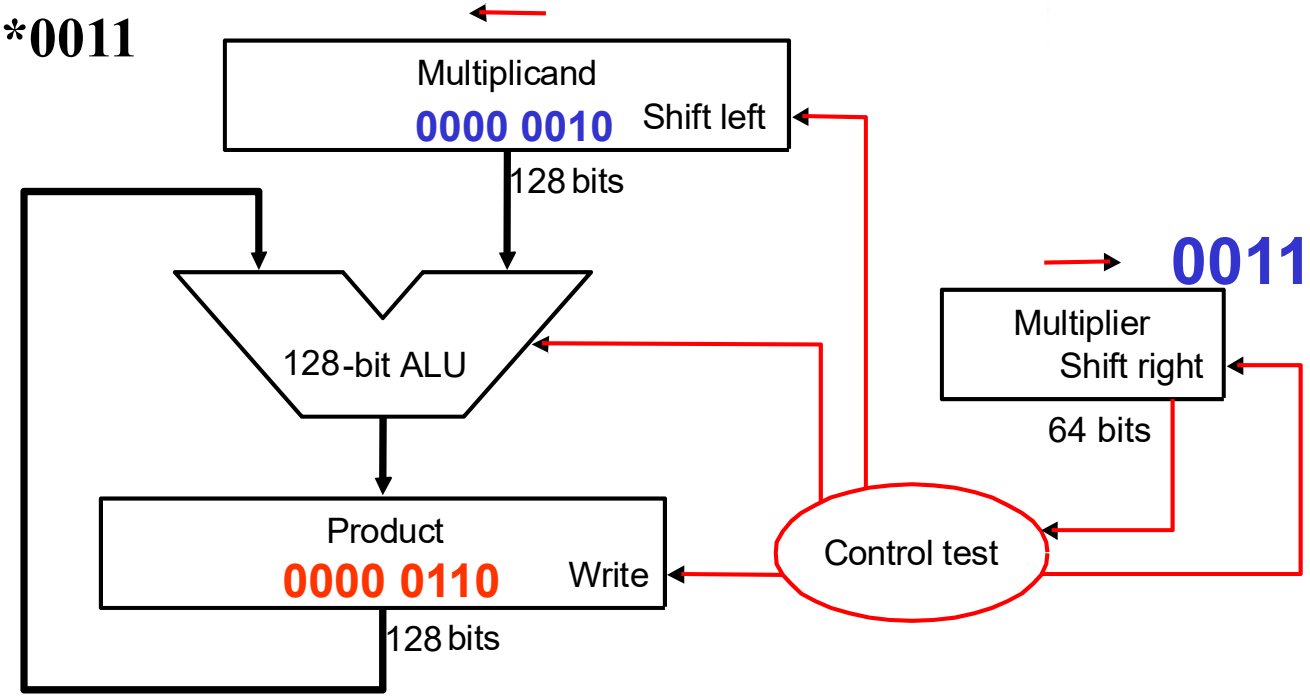
  1000 × 1001

- **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
```

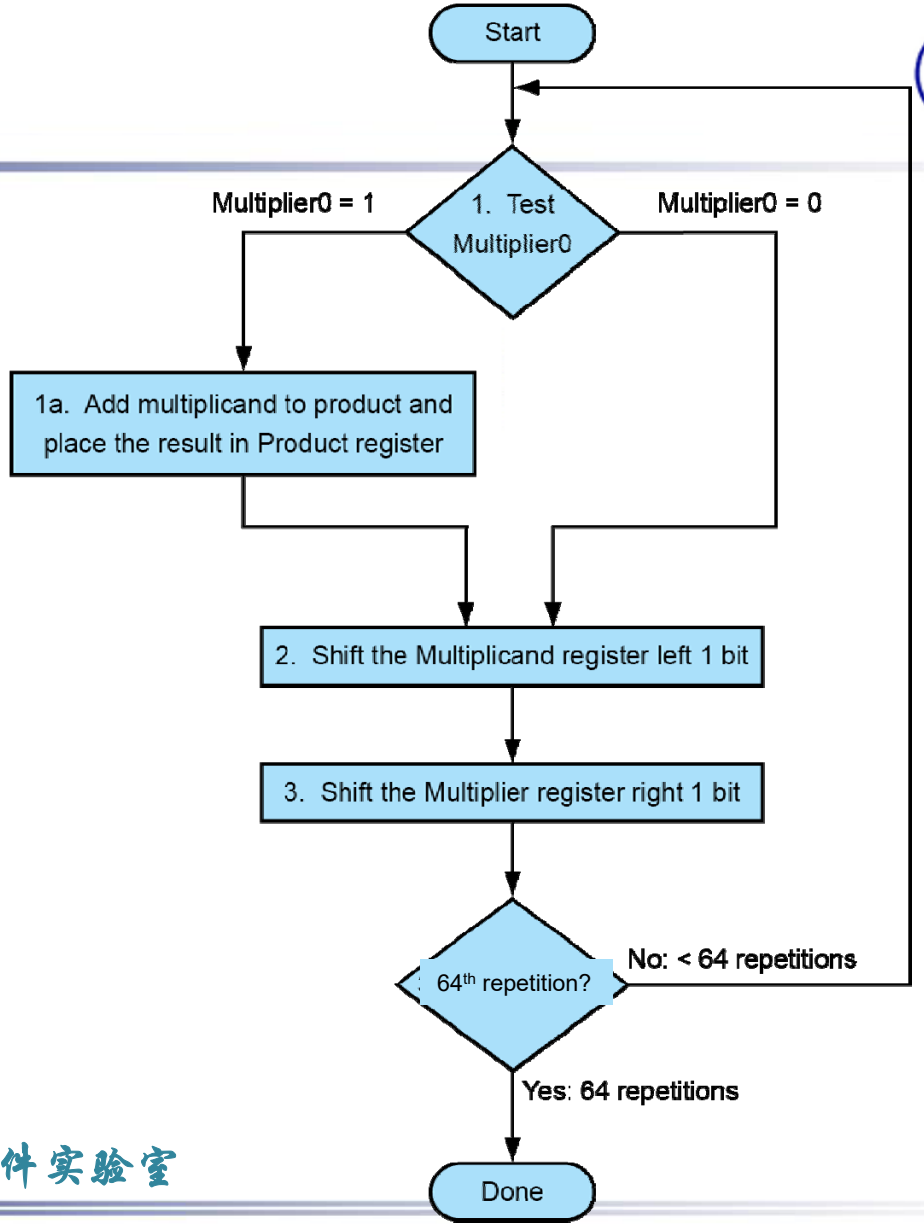
Multiplier V1– Logic Diagram

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



Multiplier V1--Algorithmic rule

- **Requires *64 iterations***
  - Addition
  - Shift
  - Comparison
- **Almost 200 cycles**
- **Very big, Too slow!**



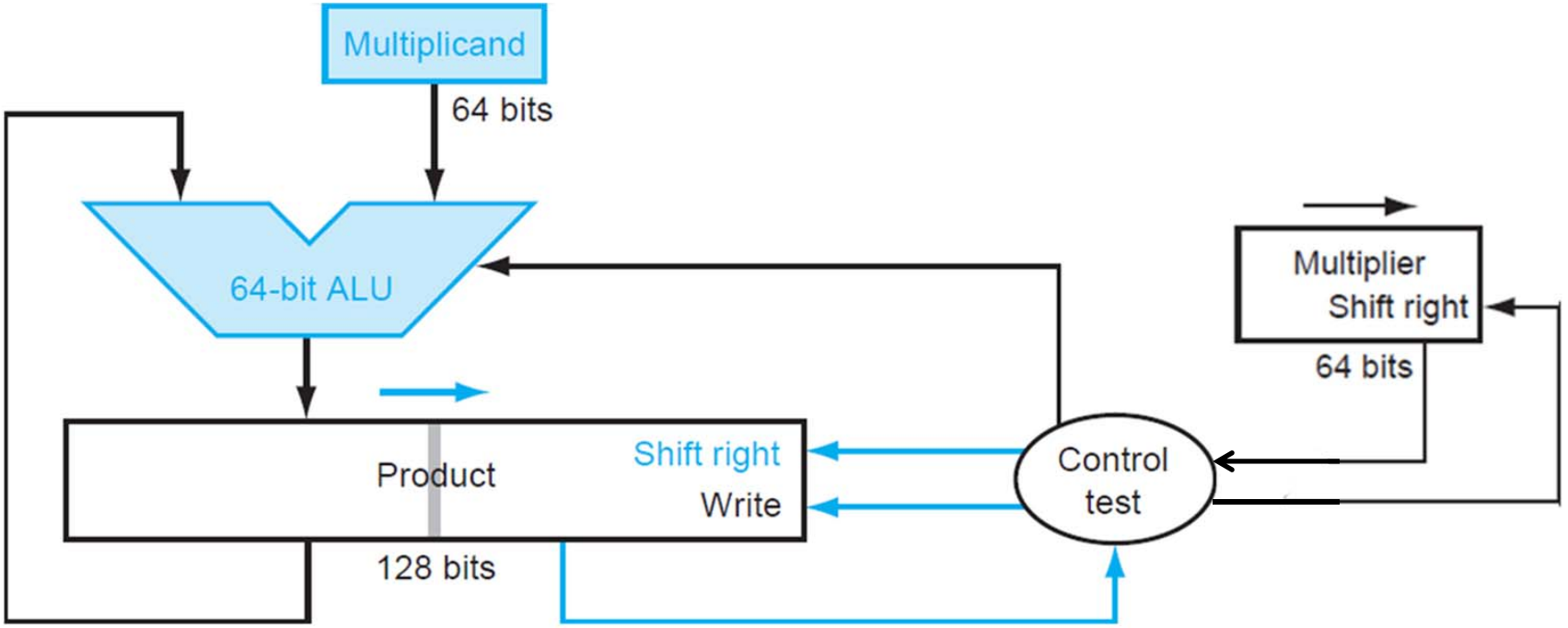
Multiplier V2

- **Real addition is performed only with 64 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 64 bits!**

```
          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
```

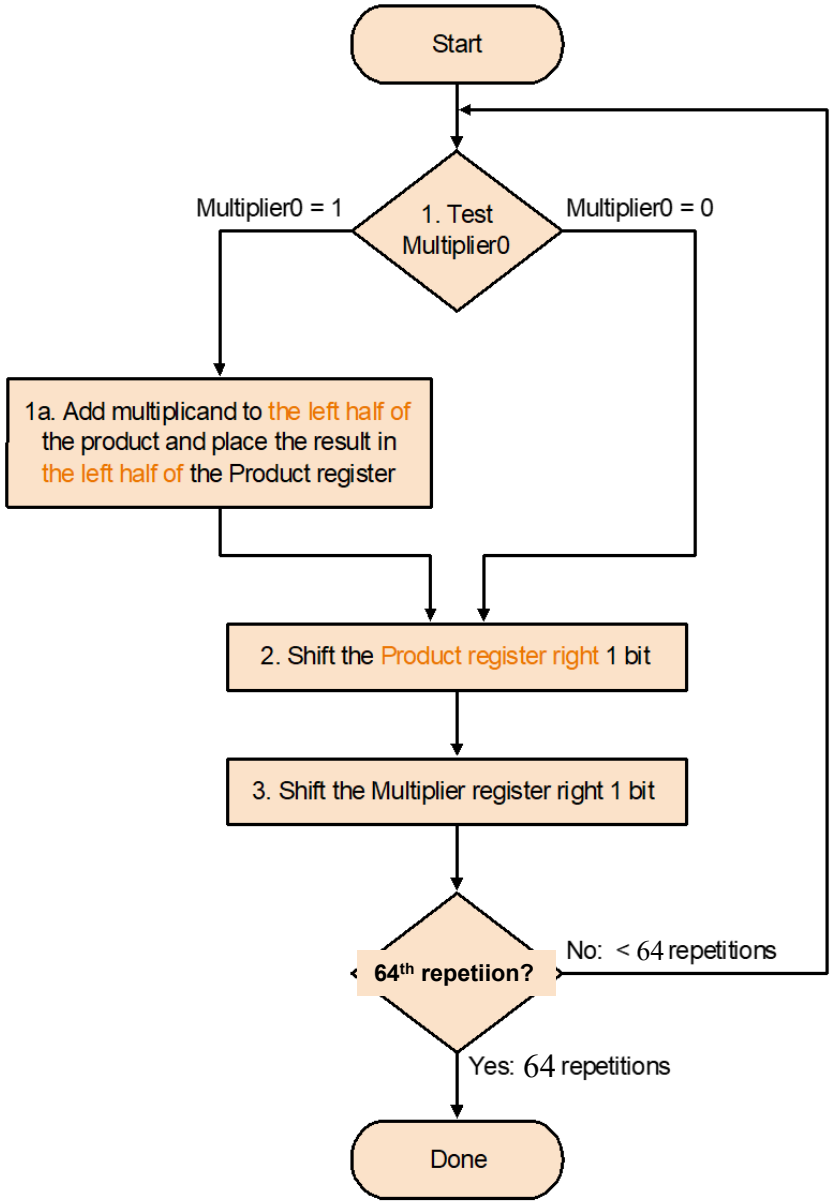
Multiplier V2-- Logic Diagram

- 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



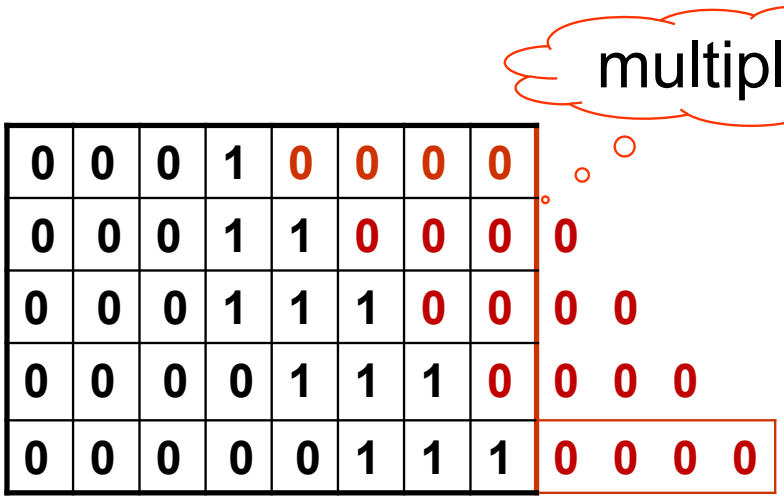
Revised 4-bit example with V2

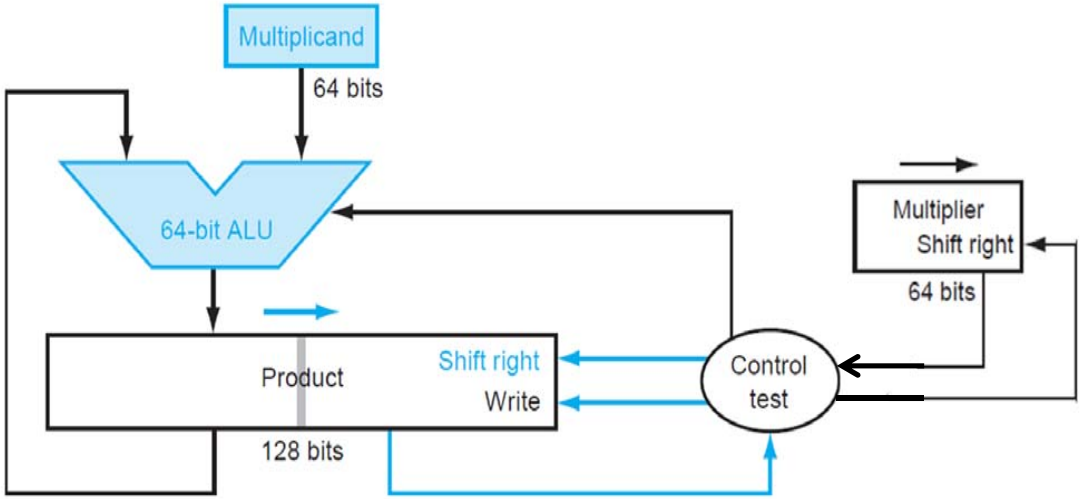
- **Multiplicand x multiplier: 0001 x 0111**

		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

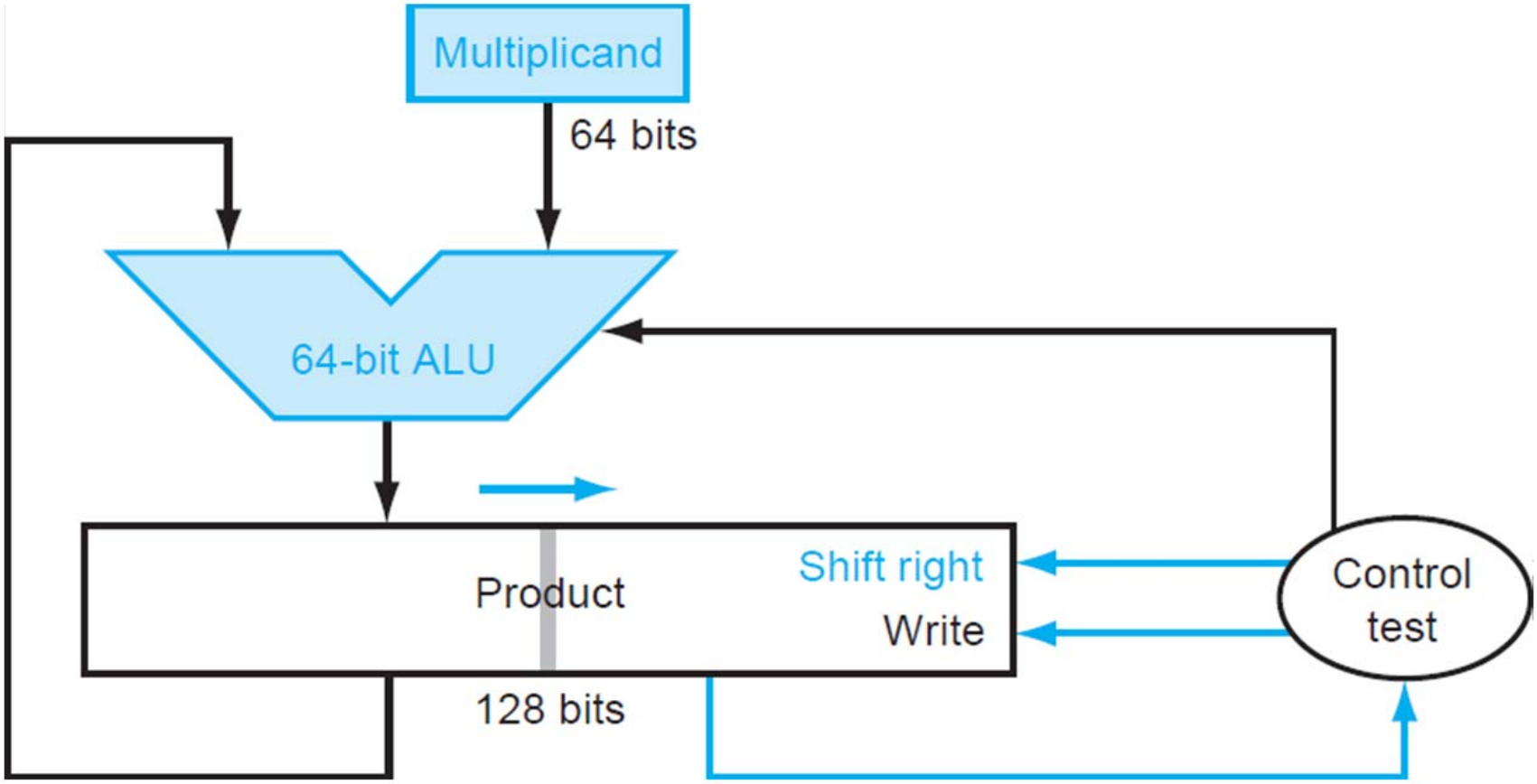
Multiplier V 3

- Further optimization
- At the initial state the product register contains only '0'
- The lower 64 bits are simply shifted out
- Idea: use these lower 64 bits for the multiplier



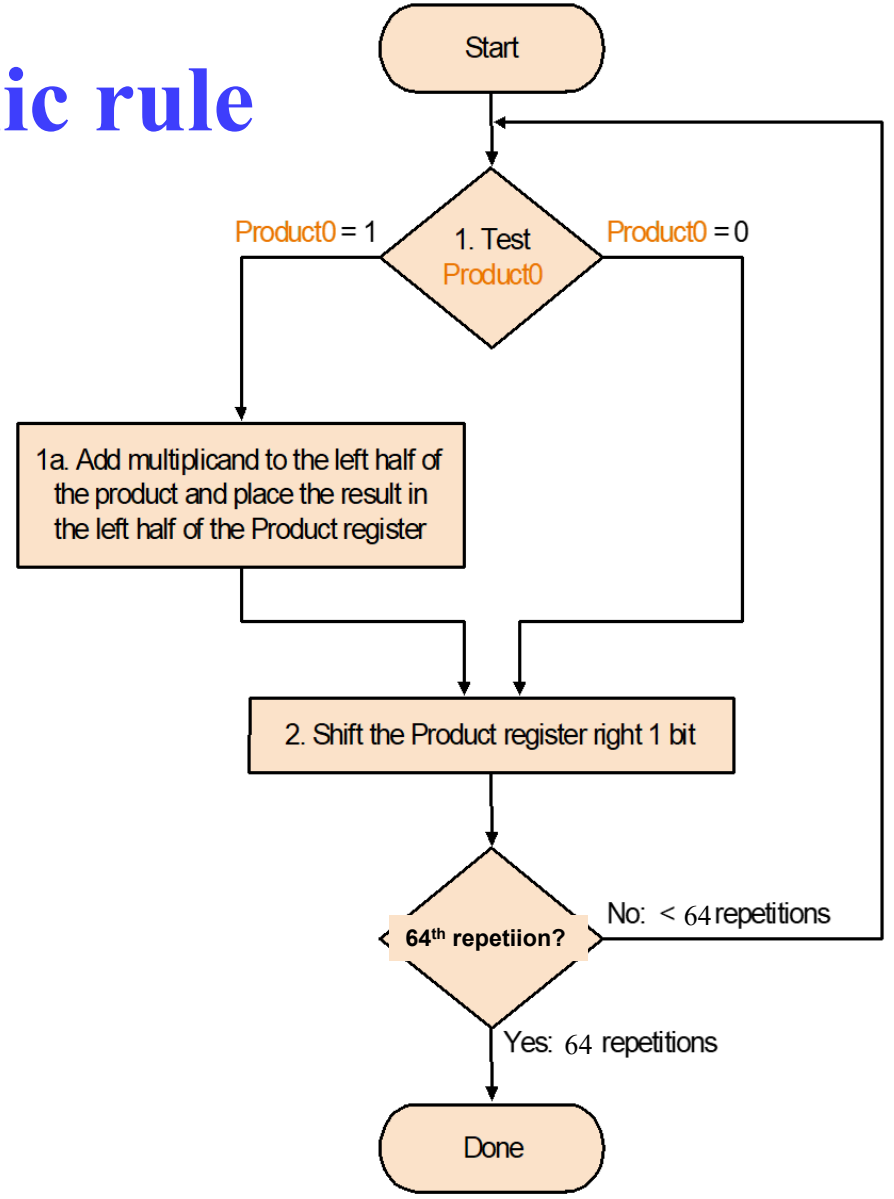


Multiplier V3 Logic Diagram



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: $Z = y \times 10111100$

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

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

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

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

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

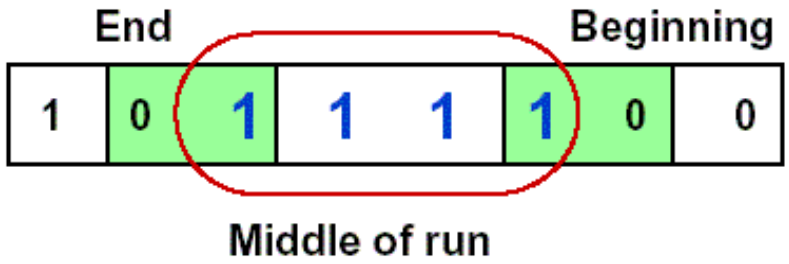
$$= y \times 2^7 + y \times 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 - y \times 2^2 + 0 \times 2^1 + 0 \times 2^0)$$

add | Only shift | sub | Only shift

1 | 01 | 11 | 1 | 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	000011110000
1	1	middle of '1'	000011110000
0	1	End	000011110000
0	0	Middle of '0'	000011110000

- **Action**

1 0	subtract multiplicand from left
1 1	no arithmetic operation-shift
0 1	add multiplicand to left half
0 0	no arithmetic operation-shift

- **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

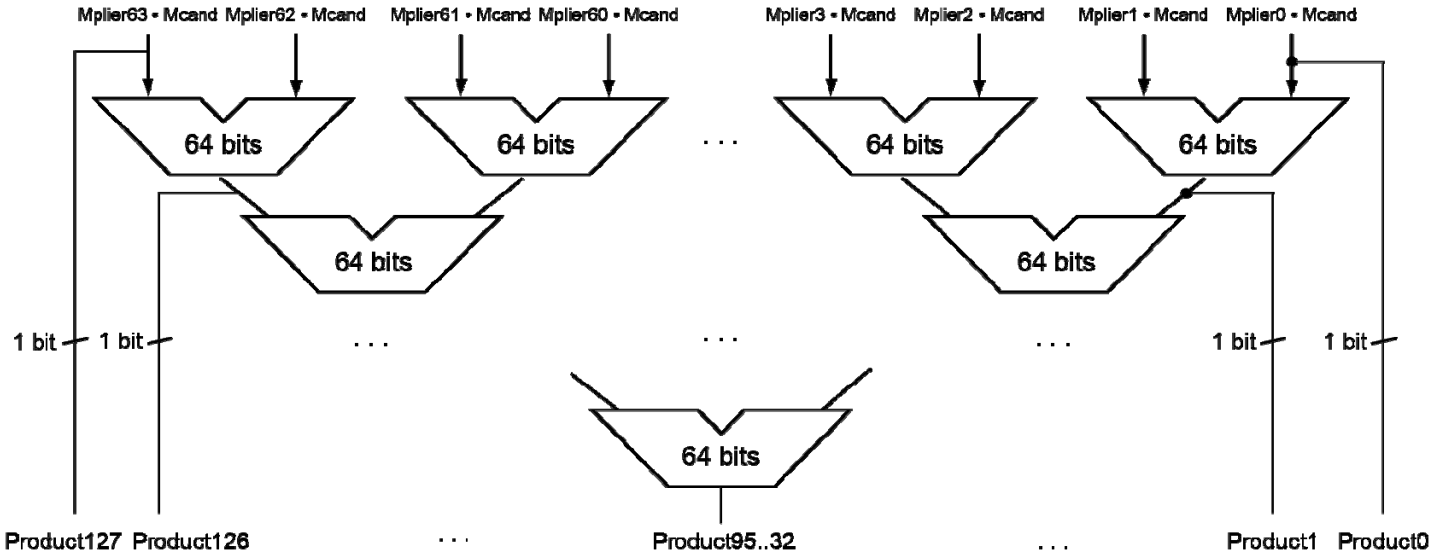
13 * (-11) = - 143

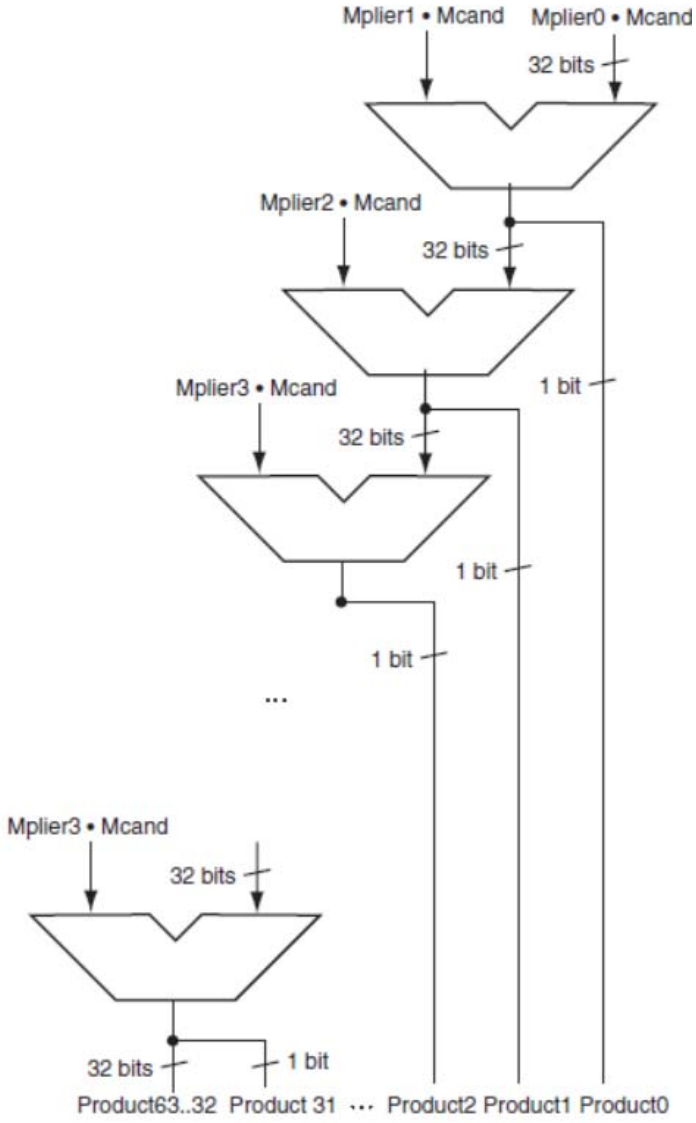
01101 * 10101 = 11011 10001-->00100 01111

	step	Multiplicand	product
0	Initial Values	01101	00000 10101 0
1	1.c:10→Prod=Prod-Mcand	01101	10011 10101 0
	2: shift right Product	01101	11001 11010 1
2	1.b:01→Prod=Prod+Mcand	01101	00110 11010 1
	2: shift right Product	01101	00011 01101 0
3	1.c:10→Prod=Prod-Mcand	01101	10110 01101 0
	2: shift right Product	01101	11011 00110 1
4	1.d:01→Prod=Prod+Mcand	01101	01000 00110 1
	2: shift right Product	01101	00100 00011 0
	1.e:10→Prod=Prod-Mcand	01101	10111 00011 0
	2: shift right Product	01101	**11011 10001** 1

Faster Multiplication

- **Uses multiple adders**
  - Cost/performance tradeoff



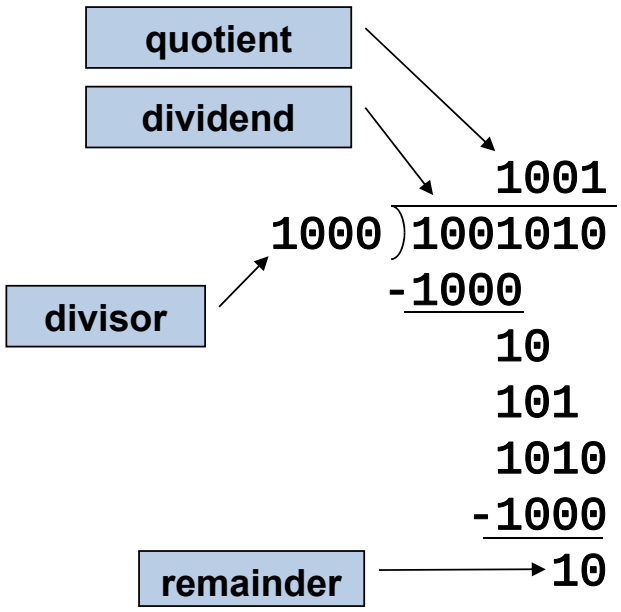


- **Can be pipelined**
  - **Several multiplication performed in parallel**

RISC-V Multiplication

- **Four multiply instructions:**
  - mul: multiply
    - Gives the lower 64 bits of the product
  - mulh: multiply high
    - Gives the upper 64 bits of the product, assuming the operands are signed
  - mulhu: multiply high unsigned
    - Gives the upper 64 bits of the product, assuming the operands are unsigned
  - mulhsu: multiply high signed/unsigned
    - Gives the upper 64 bits of the product, assuming one operand is signed and the other unsigned
  - Use mulh result to check for 64-bit overflow

3.5 Division

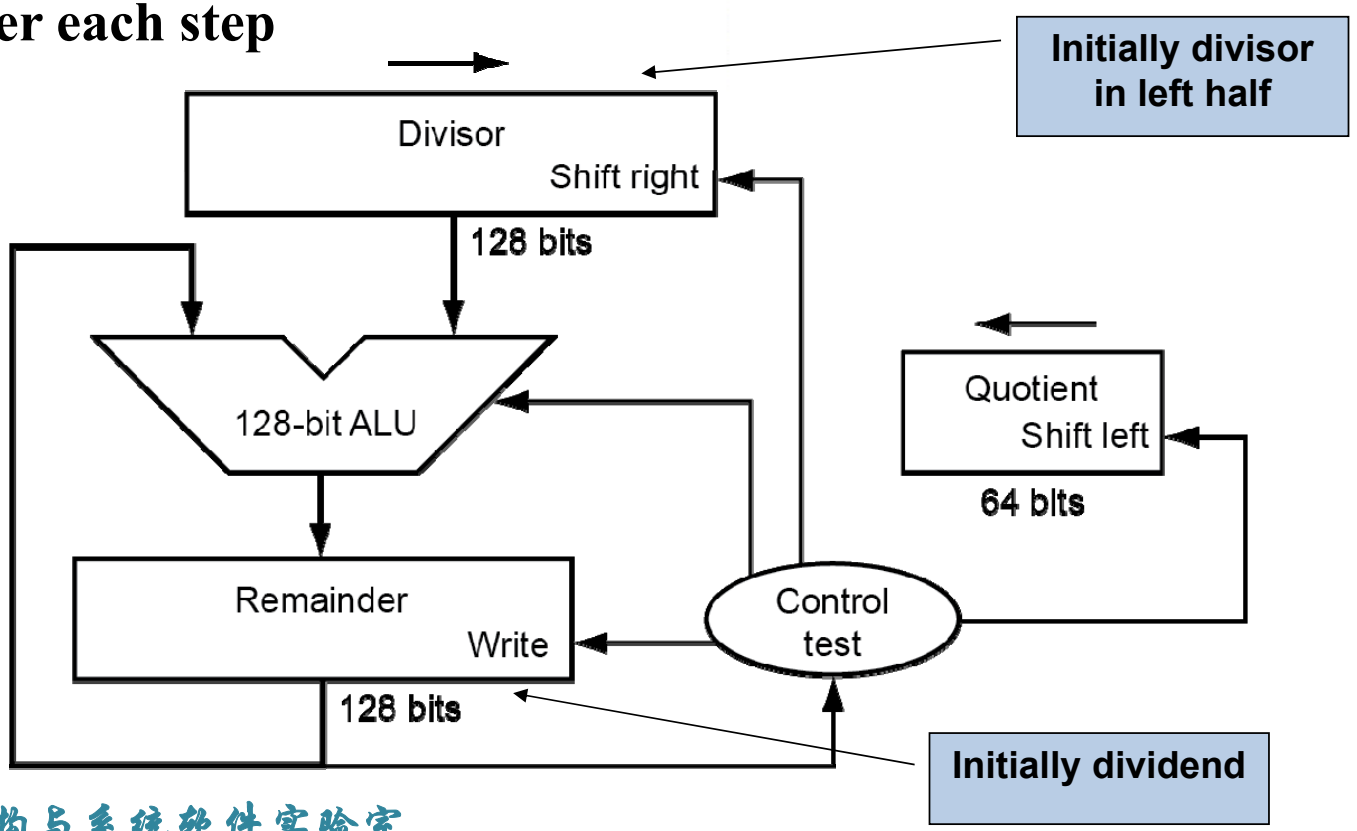


***n*-bit operands yield *n*-bit quotient and remainder**

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

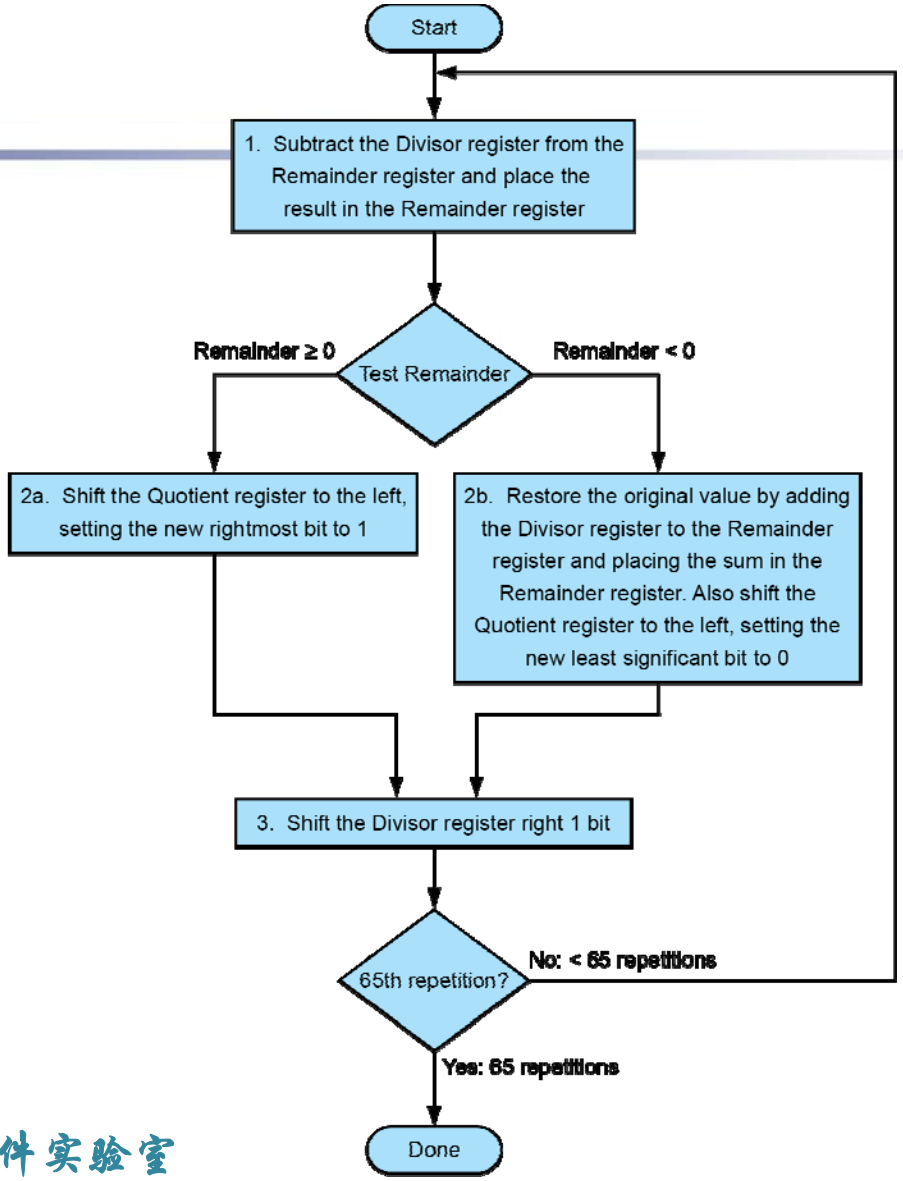
Division V1 --Logic Diagram

- 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
- 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



Algorithm V 1

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



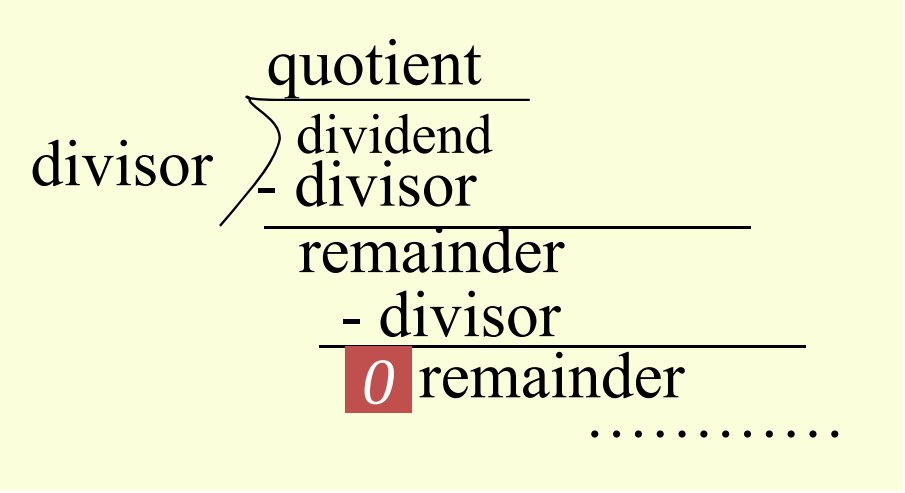
Example (7 ÷ 2) for division v1 00000111 ÷0010

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

Two questions

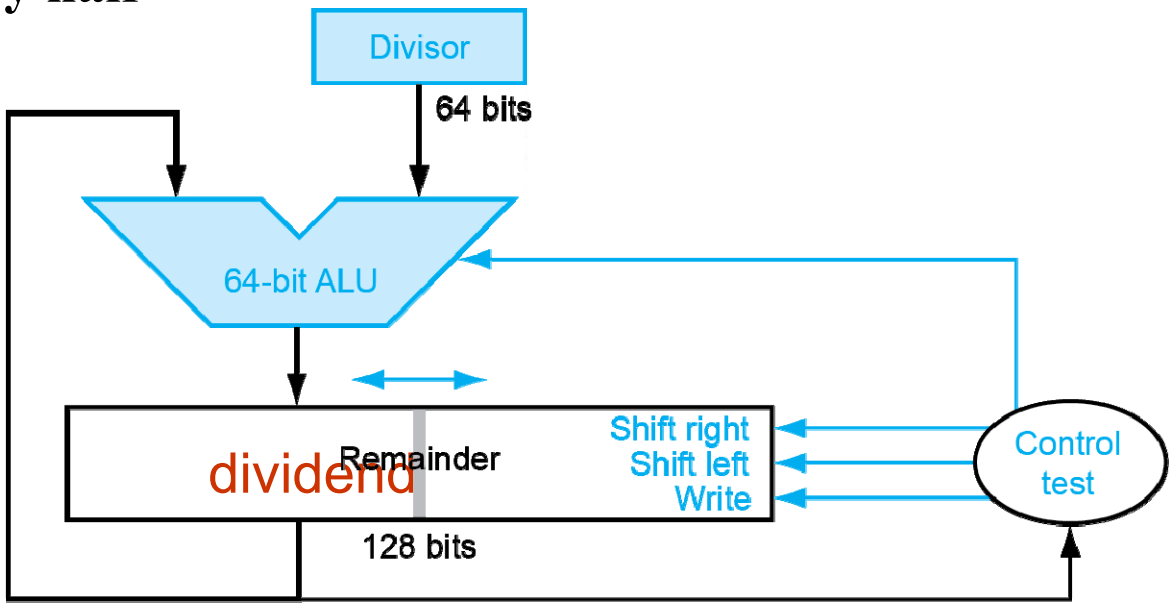
1.Why should the divisor be shifted right one bit each time?

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 ?



Modified Division

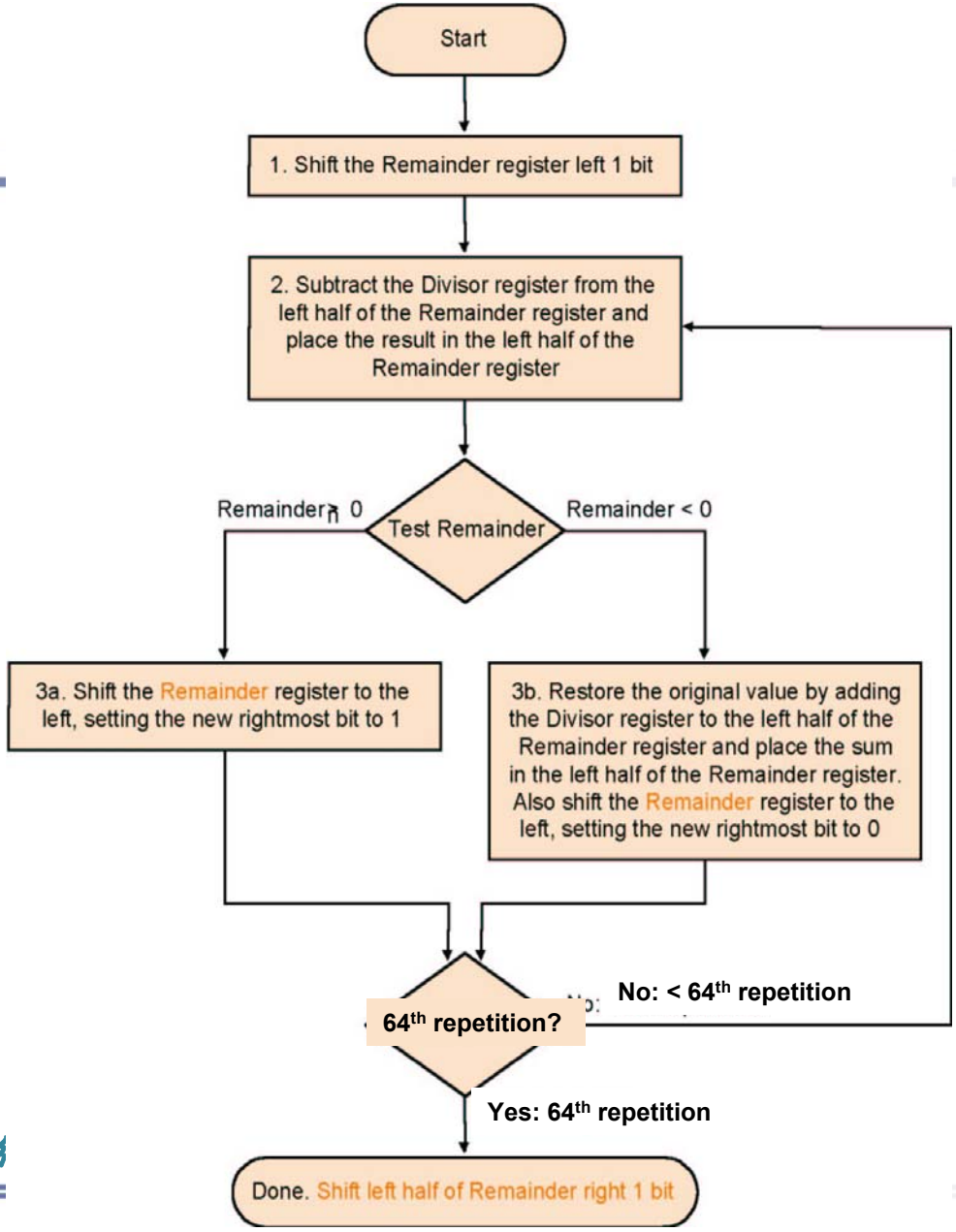
- Reduction of Divisor and ALU width by half
- Shifting of the remainder
- Saving 1 iteration
- Remainder register keeps quotient
  No quotient register required



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

Algorithm V 3

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



Example 7/2 for Modified Division

- 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 1100
	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 0001
	2a: Rem>0 →sll R,R_0=1	0010	**0010 0011**
	Shift left half of Rem right 1		**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
- **Instructions: div, divu**
- **Divide by 0 ⟶ overflow : Check by software**

Faster Division

- **Can't use parallel hardware as in multiplier**
  - Subtraction is conditional on sign of remainder
- **Faster dividers (e.g. SRT division) generate multiple quotient bits per step**
  - Still require multiple steps

RISC-V Division

- **Four instructions:**
  - div, rem: signed divide, remainder
  - divu, remu: unsigned divide, remainder

- **Overflow and division-by-zero don't produce errors**
  - Just return defined results
  - Faster for the common case of no error

3.6 Floating point numbers

- **Reasoning**
  - Larger number range than integer rage
  - Fractions
  - Numbers like e (2.71828) and π (3.14159265....)
- **Representation for non-integral numbers**
  - Including very small and very large numbers
- **Like scientific notation**
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized

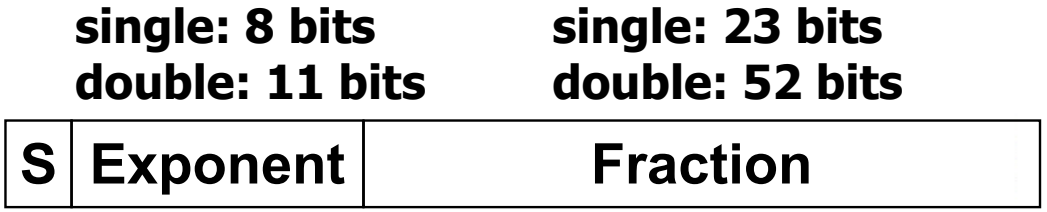
Floating Point

- **In binary**
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- **Types float and double in C**
- **Representation**
  - Sign
  - Fraction
  - Exponent
  - More bits for fraction: more accuracy
  - More bits for exponent: increases the range

Floating Point Standard

- Defined by IEEE Std 754-1980
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format



$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- **S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)**
- **Normalize significand: 1.0 ≤ |significand| < 2.0**
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- **Exponent: excess representation: actual exponent + Bias**
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Floating-Point Example

Represent –0.75

- $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
- S = 1
- Fraction = $1000\ldots00_2$
- Exponent = –1 + Bias
  - Single: –1 + 127 = 126 = 01111110_2
  - Double: –1 + 1023 = 1022 = 01111111110_2

Single precision

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

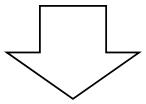
Double precision

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

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



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

Single-Precision Range

- **Exponents 00000000 and 11111111 reserved**
- **Smallest value**
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Largest value**
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- **Exponents 0000…00 and 1111…11 reserved**
- **Smallest value**
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value**
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

Relative precision

- all fraction bits are significant
- Single: approx 2^{-23}
  - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 7$ decimal digits of precision
- Double: approx 2^{-52}
  - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Limitations

- **Overflow:**

  The number is too big to be represented

- **Underflow:**

  The number is too small to be represented

Denormal Numbers

- **Exponent = 000...0 ⇒ hidden bit is 0**

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{1-\text{Bias}}$$

  - Smaller than normal numbers
    - allow for gradual underflow, with diminishing precision
  - Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Infinities and NaNs

- **Exponent = 111...1, Fraction = 000...0**
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- **Exponent = 111...1, Fraction ≠ 000...0**
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

IEEE 754 standard

- $00\ldots00_{two}$ represents 0;
- instead of interrupting on a divide by 0, software can set the result to a bit pattern representing +∞ or -∞;
- *NaN*: Not a Number.
- ±∞ - represented by f=0, E=255, S=0,1 - must obey all mathematical conventions: F+∞=∞, F/∞=0
- denormalized numbers - represented by E=0 -values smaller than smallest normalized number - lowering probability of exponent underflow

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 in decimal： system precision 4 digits

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

- Aligning the two numbers

  $9.999 \bullet 10^{1}$

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

- Addition

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

- Normalization

  $1.0015 \bullet 10^{2}$

- Rounding

  $1.002 \bullet 10^{2}$

Floating point addition steps

- Alignment
- The proper digits have to be added
- Addition of significands
- Normalization of the result
- Rounding

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

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

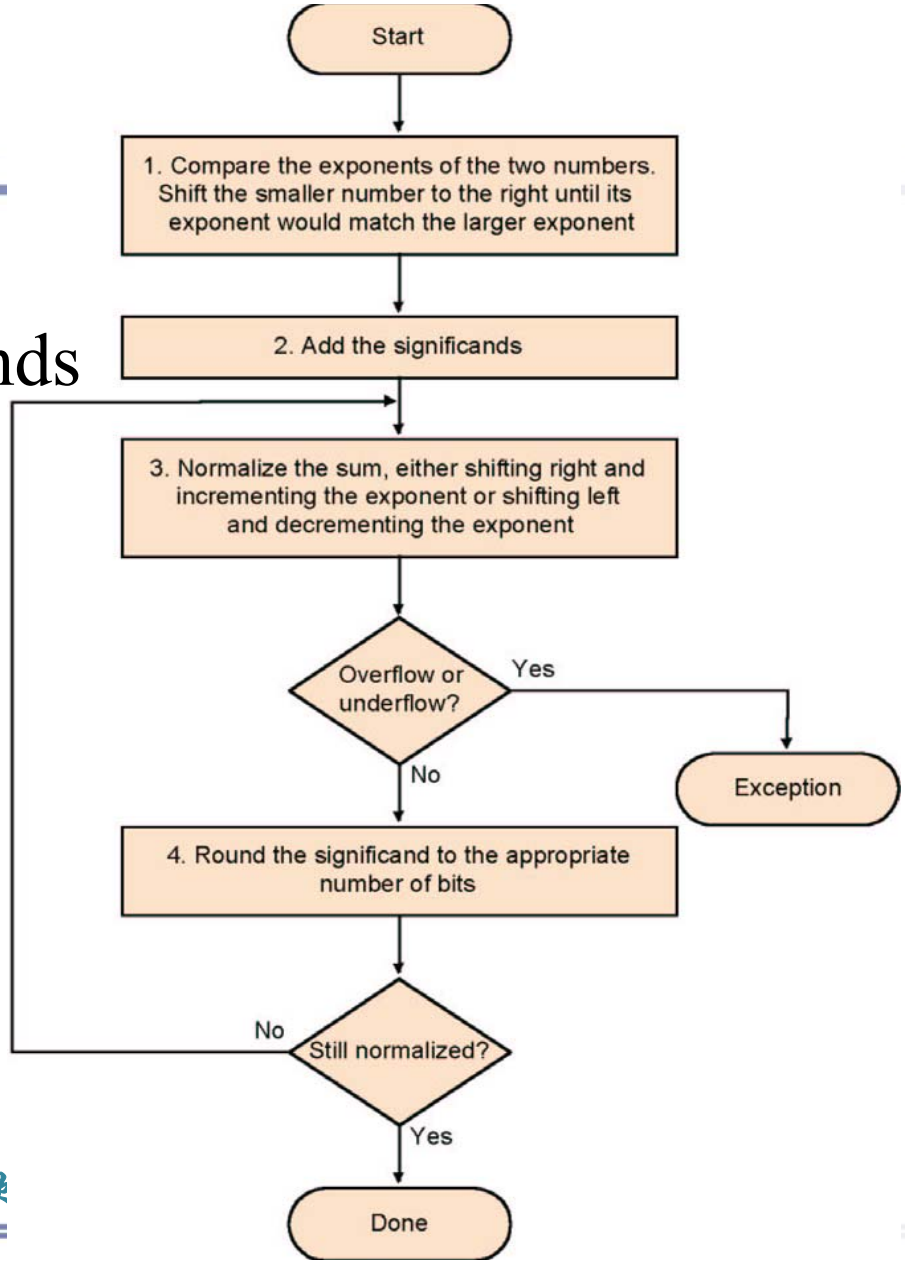
- Step3: Normalize the sum and checking for overflow or underflow

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

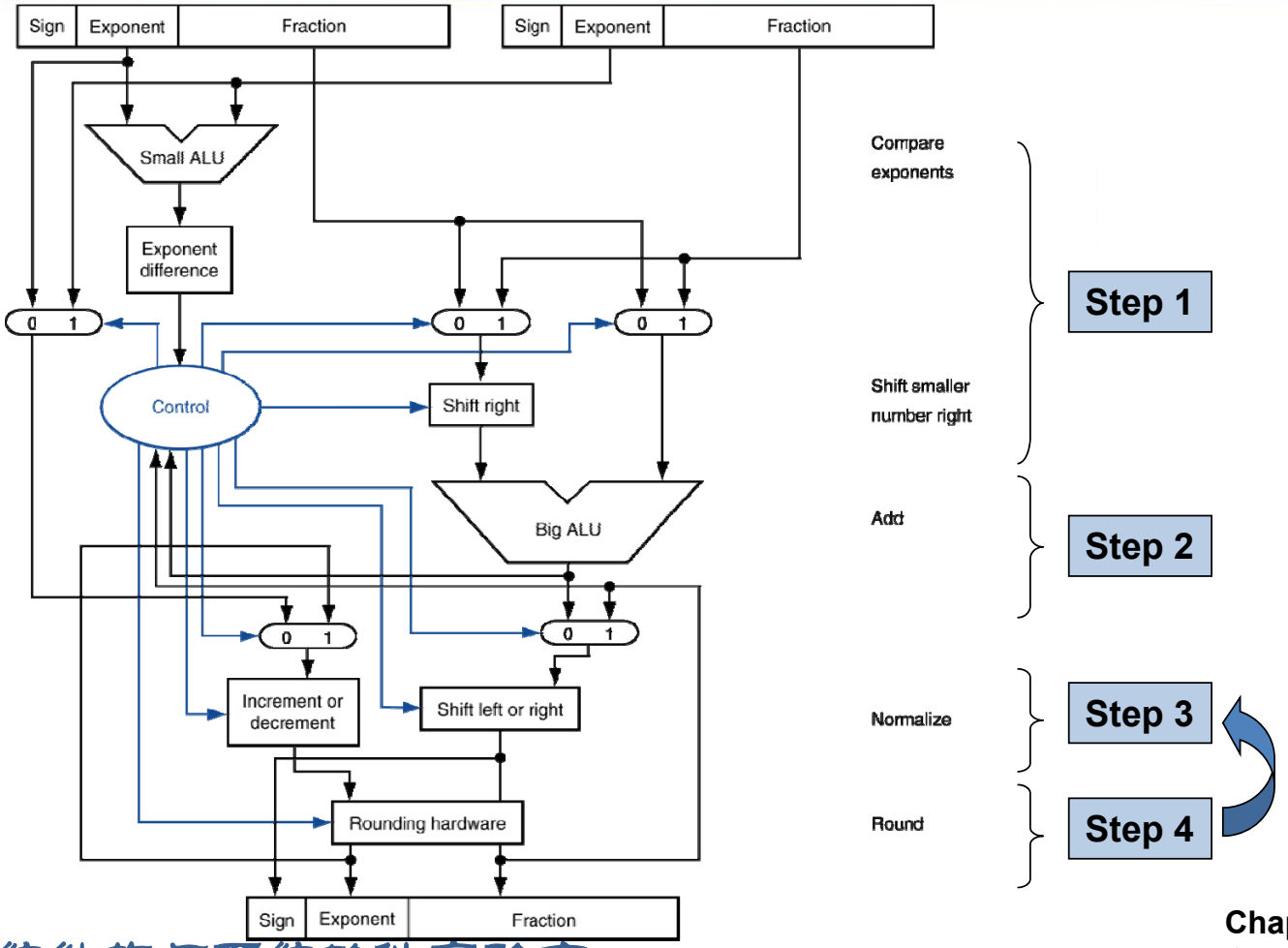
  $1.000_2 \times 2^{-4} = 0.0625_{10}$

Algorithm

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



FP Adder Hardware



FP Adder Hardware

- **Much more complex than integer adder**
- **Doing it in one clock cycle would take too long**
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- **FP adder usually takes several cycles**
  - Can be pipelined

Floating-Point Multiplication

- **Consider a 4-digit decimal example**
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- **1. Add exponents**
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- **2. Multiply significands**
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- **3. Normalize result & check for over/underflow**
  - 1.0212×10^{6}
- **4. Round and renormalize if necessary**
  - 1.021×10^{6}
- **5. Determine sign of result from signs of operands**
  - $+1.021 \times 10^{6}$

Floating Point Multiplication

- Composition of number from different parts

  → separate handling

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

- Example

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

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

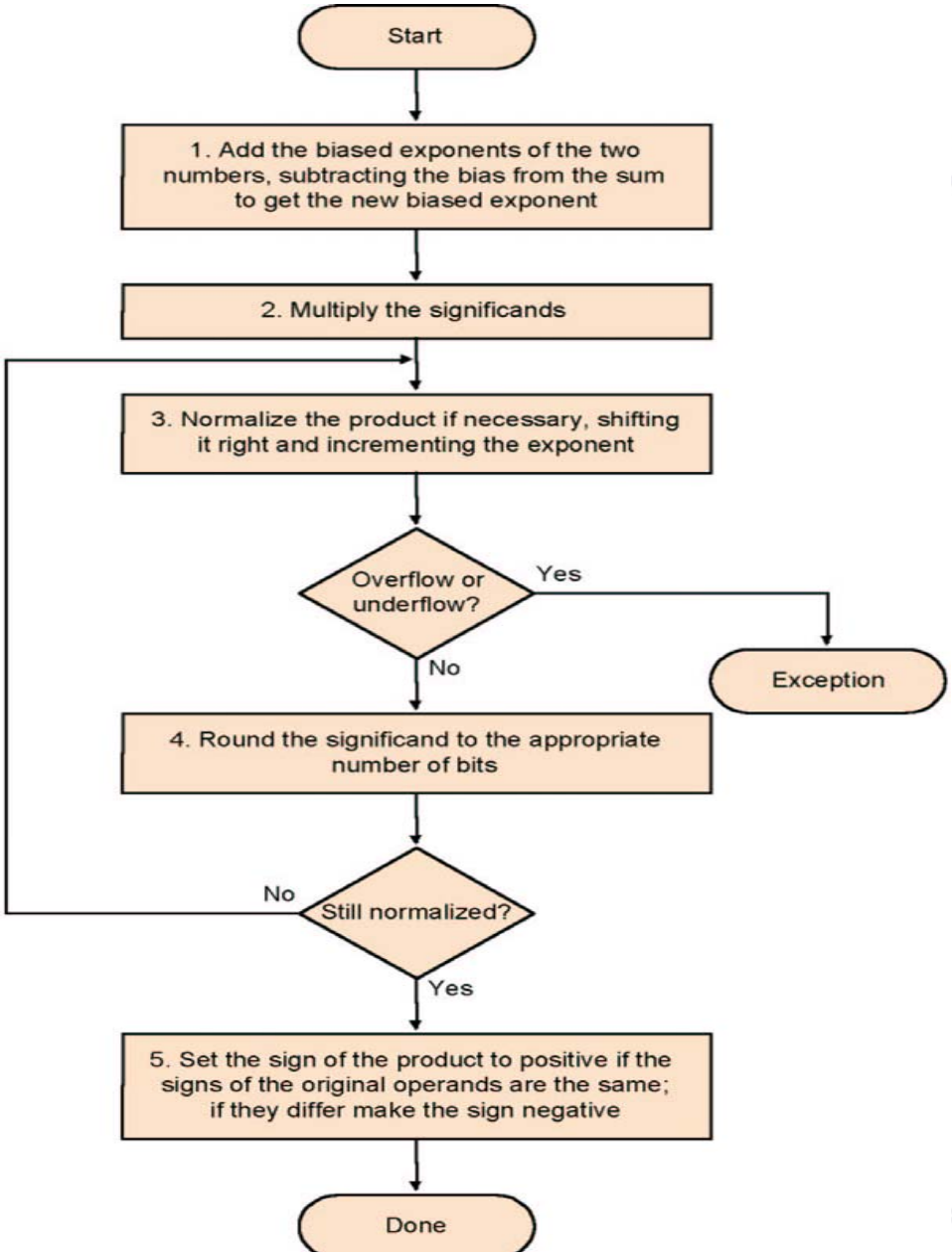
- Both significands are 1 → product = 1 →Sign=1
- Add the exponents, bias = 127

  $$\begin{array}{r} 10000010 \\ +10000011 \\ \hline 110000101 \end{array}$$

  Correction: 110000101-01111111=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

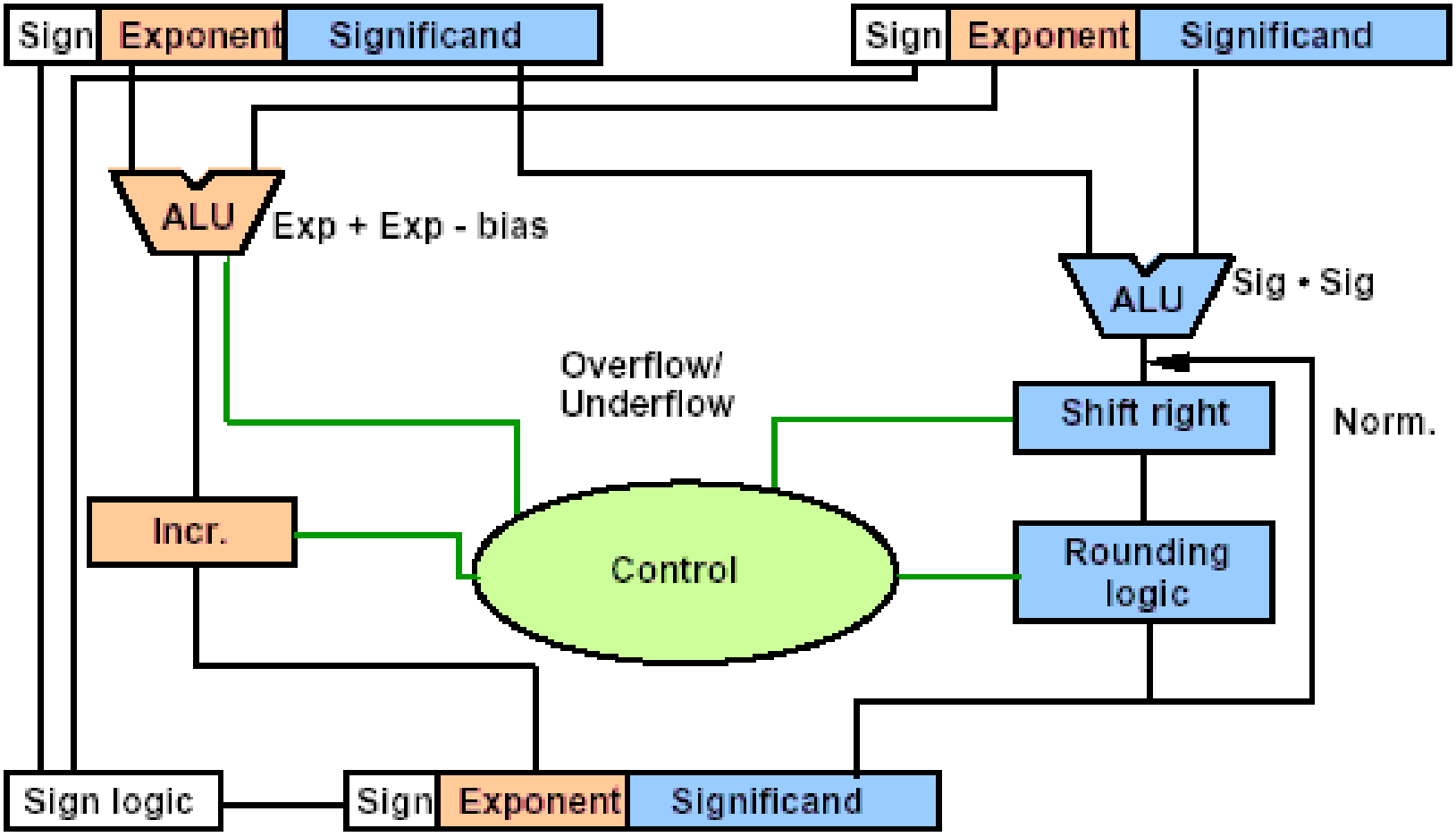


multiplying the numbers 0.5_{ten} and -0.4375_{ten} $\rightarrow 1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$

- **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**
  - $1.110000_{two} \times 2^{-3}$
- **Step 3. normalize**
  - $127 \geq -3 \geq -126$, no overflow or underflow.
- **Step 4. Rounding**
  - $1.110_{two} \times 2^{-3}$
- **Step 5. sign**
  - $-1.110_{two} \times 2^{-3} = -0.21875_{ten}$

$$
\begin{array}{rr}
 & 1.000_{two} \\
\times & 1.110_{two} \\
\hline
 & 0000 \\
 & 1000 \\
 & 1000 \\
 & 1000 \\
\hline
 & 1110000_{two}
\end{array}
$$

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

FP Arithmetic Hardware

- **FP multiplier is of similar complexity to FP adder**
  - But uses a multiplier for significands instead of an adder
- **FP arithmetic hardware usually does**
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- **Operations usually takes several cycles**
  - Can be pipelined

Division-- Brief

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

Parallelism and Computer Arithmetic: Associativity

- **x + (y+ z) = (x + y) + z ?**
  - x = -1.5_{ten} x 10^{38}, y= 1.5_{ten} X 10^{38}, and z = 1.0
  - x + (y + z) = 0.0
  - (x+y) + z = 1.0

FP Instructions in RISC-V

- **Separate FP registers: f0, …, f31**
  - double-precision
  - single-precision values stored in the lower 32 bits
- **FP instructions operate only on FP registers**
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- **FP load and store instructions**
  - `flw, fld`
  - `fsw, fsd`

FP Instructions in RISC-V

- **Single-precision arithmetic**
  - `fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s`
    - e.g., `fadds.s f2, f4, f6`
- **Double-precision arithmetic**
  - `fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d`
    - e.g., `fadd.d f2, f4, f6`
- **Single- and double-precision comparison**
  - `feq.s, flt.s, fle.s`
  - `feq.d, flt.d, fle.d`
  - Result is 0 or 1 in integer destination register
    - Use beq, bne to branch on comparison result
- **Branch on FP condition code true or false**
  - `B.cond`

FP Example: °F to °C

- C code:

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

  - `fahr` in f10, result in f10, literals in global memory space

- Compiled RISC-V code:

```
f2c:
  flw    f0,const5(x3)  // f0 = 5.0f
  flw    f1,const9(x3)  // f1 = 9.0f
  fdiv.s f0, f0, f1  // f0 = 5.0f / 9.0f
  flw    f1,const32(x3) // f1 = 32.0f
  fsub.s f10,f10,f1  // f10 = fahr - 32.0
  fmul.s f10,f0,f10  // f10 = (5.0f/9.0f) * (fahr-32.0f)
  jalr   x0,0(x1)    // return
```

FP Example: Array Multiplication

- C = C + A × B
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double c[][],
         double a[][], double b[][]) {
  size_t 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)
        c[i][j] = c[i][j]
                + a[i][k] * b[k][j];
}
```

  - Addresses of c, a, b in x10, x11, x12, and i, j, k in x5, x6, x7

FP Example: Array Multiplication

- **RISC-V code:**

```
mm:...
        li    x28,32       // x28 = 32 (row size/loop end)
        li    x5,0         // i = 0; initialize 1st for loop
L1:     li    x6,0         // j = 0; initialize 2nd for loop
L2:     li    x7,0         // k = 0; initialize 3rd for loop
        slli  x30,x5,5     // x30 = i * 2**5 (size of row of c)
        add   x30,x30,x6   // x30 = i * size(row) + j
        slli  x30,x30,3    // x30 = byte offset of [i][j]
        add   x30,x10,x30  // x30 = byte address of c[i][j]
        fld   f0,0(x30)    // f0 = c[i][j]
L3:     slli  x29,x7,5     // x29 = k * 2**5 (size of row of b)
        add   x29,x29,x6   // x29 = k * size(row) + j
        slli  x29,x29,3    // x29 = byte offset of [k][j]
        add   x29,x12,x29  // x29 = byte address of b[k][j]
        fld   f1,0(x29)    // f1 = b[k][j]
```

FP Example: Array Multiplication

```
…
    slli   x29,x5,5     // x29 = i * 2**5 (size of row of a)
    add    x29,x29,x7   // x29 = i * size(row) + k
    slli   x29,x29,3    // x29 = byte offset of [i][k]
    add    x29,x11,x29  // x29 = byte address of a[i][k]
    fld    f2,0(x29)    // f2 = a[i][k]
    fmul.d f1, f2, f1   // f1 = a[i][k] * b[k][j]
    fadd.d f0, f0, f1   // f0 = c[i][j] + a[i][k] * b[k][j]
    addi   x7,x7,1      // k = k + 1
    bltu   x7,x28,L3    // if (k < 32) go to L3
    fsd    f0,0(x30)    // c[i][j] = f0
    addi   x6,x6,1      // j = j + 1
    bltu   x6,x28,L2    // if (j < 32) go to L2
    addi   x5,x5,1      // i = i + 1
    bltu   x5,x28,L1    // if (i < 32) go to L1
```

Accurate Arithmetic

- **IEEE Std 754 specifies additional rounding control**
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- **Not all FP units implement all options**
  - Most programming languages and FP libraries just use defaults
- **Trade-off between hardware complexity, performance, and market requirements**

Round modes

- **Rounding: four rounding modes**
  - Round to 0
  - Round to +∞
  - Round to -∞
  - Round to next even number (default)

Round modes Example

y	round down (towards −∞)	round up (towards +∞)	round towards zero	round to nearest even
+23.67	+23	+24	+23	+24
+23.50	+23	+24	+23	+24
+23.35	+23	+24	+23	+23
+23.00	+23	+23	+23	+23
0	0	0	0	0
−23.00	−23	−23	−23	−23
−23.35	−24	−23	−23	−23
−23.50	−24	−23	−23	−24
−23.67	−24	−23	−23	−24

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

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 $0.50 \ldots 0l_{ten}$ when rounding.
- examples in the floating point format with guard, round and sticky bits

Examples for guard, round, and sticky bit

g r s

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）

Concluding Remarks

- **Bits have no inherent meaning**
  - Interpretation depends on the instructions applied
- **Computer representations of numbers**
  - Finite range and precision
  - Need to account for this in programs
- **ISAs support arithmetic**
  - Signed and unsigned integers
  - Floating-point approximation to reals
- **Bounded range and precision**
  - Operations can overflow and underflow

Assignment

Reading: chapter 3

Problems:

3.1, 3.5, 3.8, 3.9, 3.13, 3.19

3.20-24(later after chapter 2), 3.27, 3.29

Note: In 3.21, change MIPS to RISC-V


END