浙江大學
ZHEJIANG UNIVERSITY

系统结构与网络安全研究所

计算机组成与设计

Computer Organization & Design

The Hardware/Software Interface

Chapter 3

Arithmetic for Computer

林 芃
Lin Peng
penglin@zju.edu.cn

Outline

3.1 Introduction

- **Computer words are composed of bits;**
  - thus one word is a vector of binary numbers
  - there are 32bit/word or 64bits/word in RISC-V
  - 32 bits contains four bytes

- **Generic Implementation**
  - use program counter (PC) to link to instruction address
  - fetch the instruction from memory
  - the instruction tells what needs to be done
  - ALU will perform the specified arithmetic operations

Numbers and their representation

- **Number systems**
  - Radix based systems are dominating

    decimal, octal, binary,…

$0 \leq b \leq K$

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

MSD LSD

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

  - ***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 (External)**
    - Great for printable symbols and numbers
    - Complex arithmetic (character wise)
    - “0” 48_{10}, 00110000_2, $0x30_{16}$
    - “SPACE” 32_{10}, 00100000_2, $0x20_{16}$
    - “!” 33_{10}, 00100001_2, $0x21_{16}$
  - **Binary number (Internal)**
    - Natural form of 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 574)
    - Double precision (IEEE 574)
    - Quadruple precision

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 negative numbers?**
  i.e., which bit patterns will represent which numbers?

Do you Know?

- **What does this number mean in a Digital computer?**

  $00110011110111100000000100000000_2$

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

Computer uses binary numbers just like how we use decimal numbers

For binary integer

- **The following is a 4-bit binary integer, what does it mean?**

  1001_2

  - Don’t know! Do not know, is still the right answer !

- **Ah, we still do not know?**
- **Different representations have different meanings**
  - Unsigned $1001_2=9_{10}$
  - Signed $1001_2=-1_{10}$ or -7_{10} ?

Signed Number Representations

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

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

Outline

Addition & subtraction

- **Adding bit by bit, carries → next digit**

```
  0000 0111      7₁₀
+ 0000 0110      6₁₀
-----------
  0000 1101     13₁₀
```

- **Subtraction**
  - Directly
  - Addition of 2's complement

```
  0000 0111      7₁₀
- 0000 0110      6₁₀
-----------
  0000 0001      1₁₀
```

```
  0000 0111      7₁₀
+ 1111 1010     -6₁₀
-----------
  0000 0001      1₁₀
```

Overflow

- The sum of two numbers can exceed any representation

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

- The difference of two numbers can exceed any representation

- 2's complement: Numbers change sign and size

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

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)***

Double sign-bits

Reaction on overflow

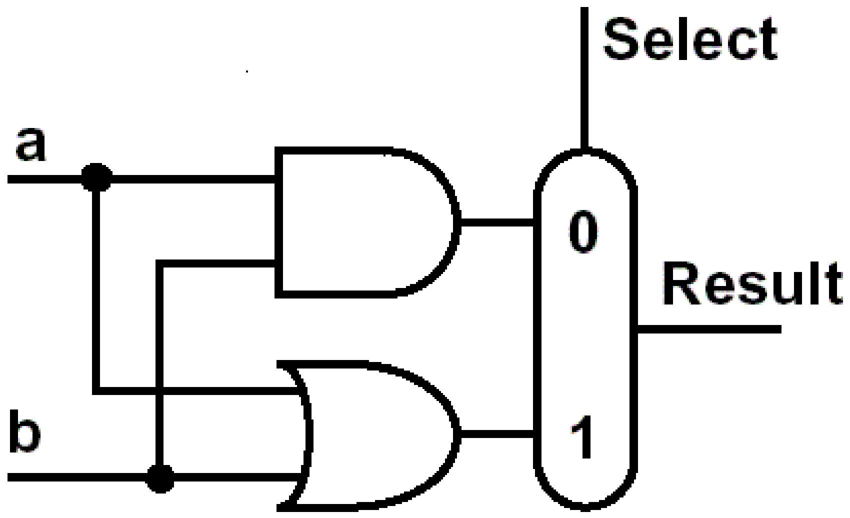
- Ignore ?
- Reaction from the OS
- Signaling to application (Python,...)

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

Constructing an ALU

- Two methods constitute the ALU
  - Modular design (e.g. add extensions to support add)
  - Sharable logic with “select”
- Step by step:
  - build a single bit ALU
  - and expand it to the desired width
- First function:
  - logic AND and OR



Second function

- Arithmetic
  - A half adder to A full adder

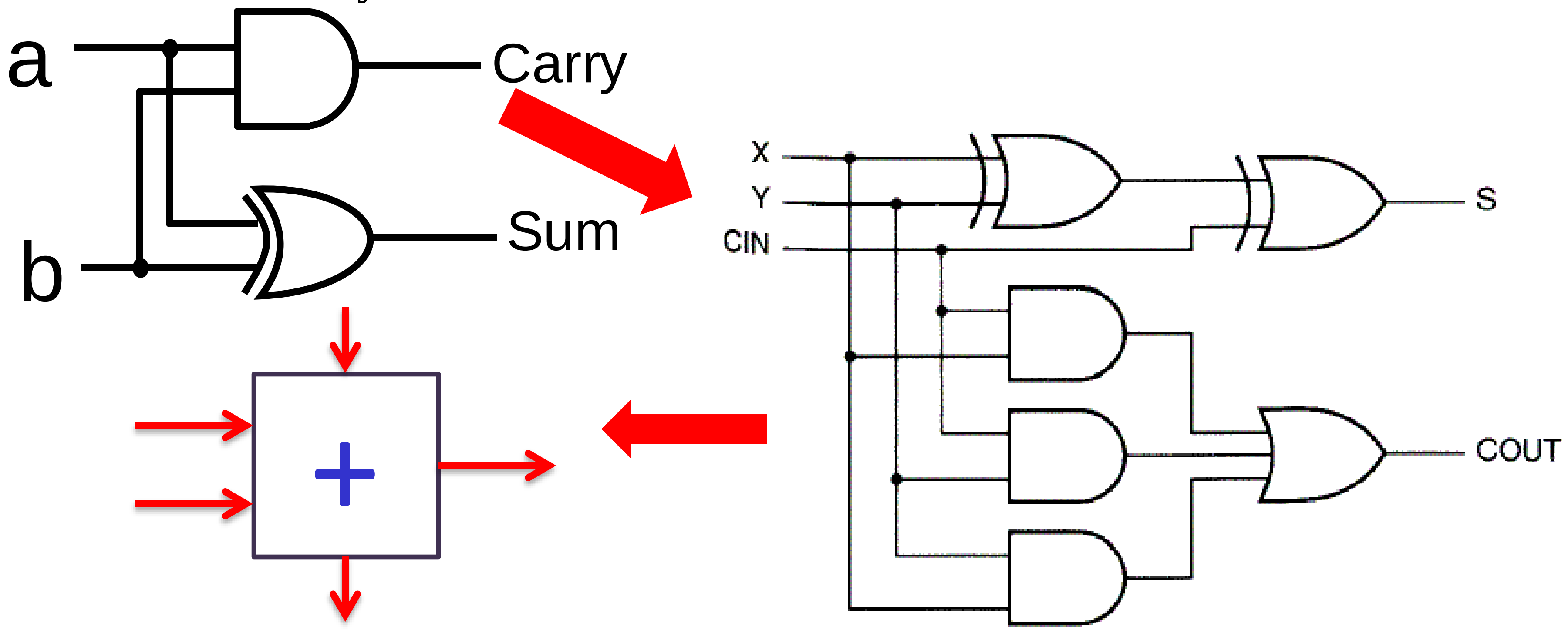
    Sum = a b + a b

    Carry = a b

- Accepts a carry in

  $\text{Sum} = a \oplus b \oplus C_{In}$

  $C_{Out} = b\, C_{In} + a\, C_{In} + ab$



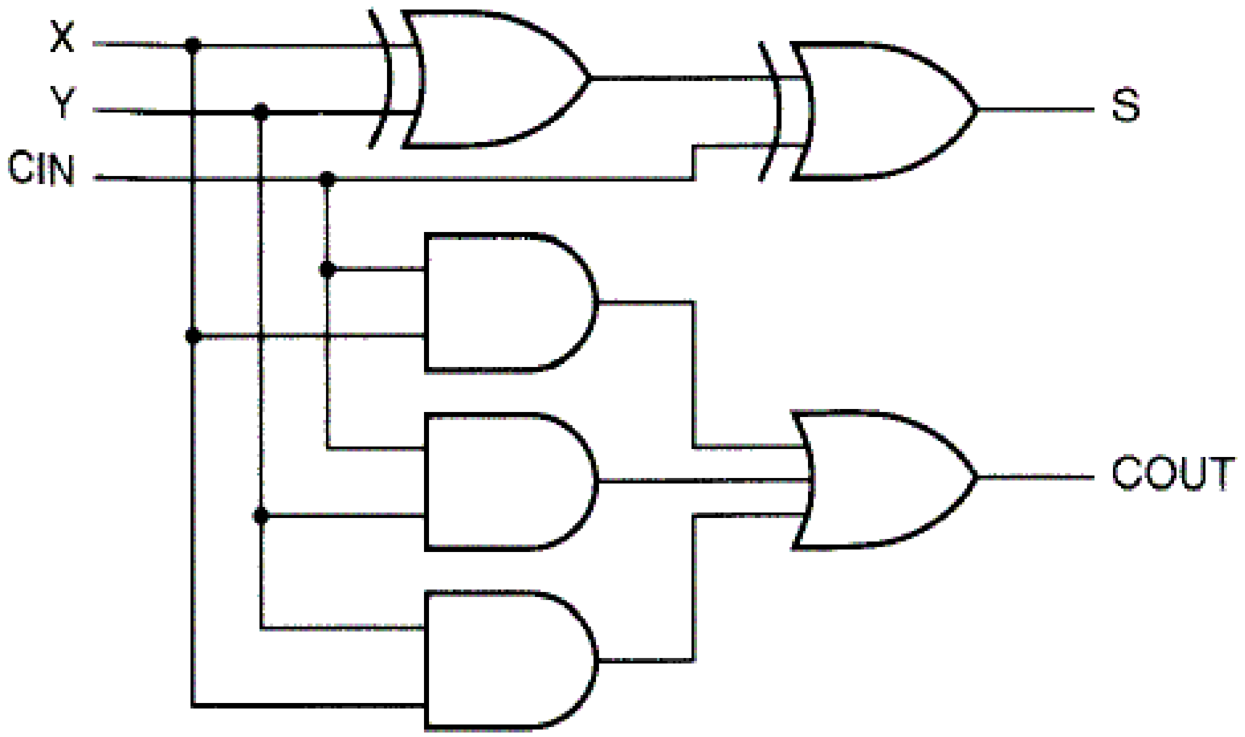
A full adder

- 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	
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

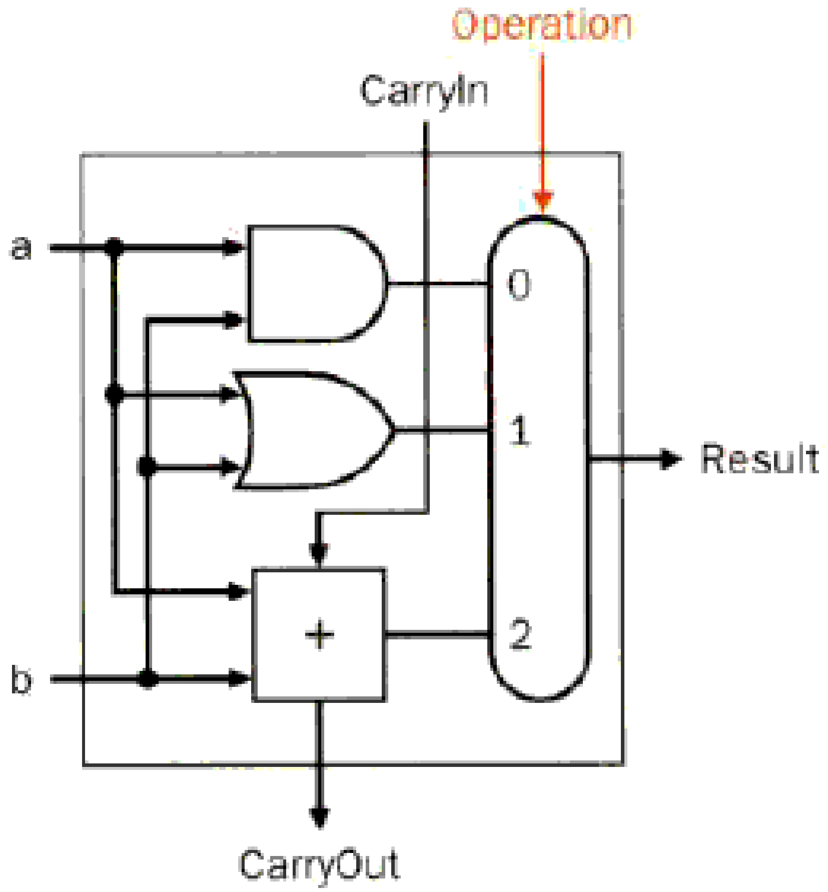
- Full adder in 2-level design



1 bit ALU

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

  Cascade Element

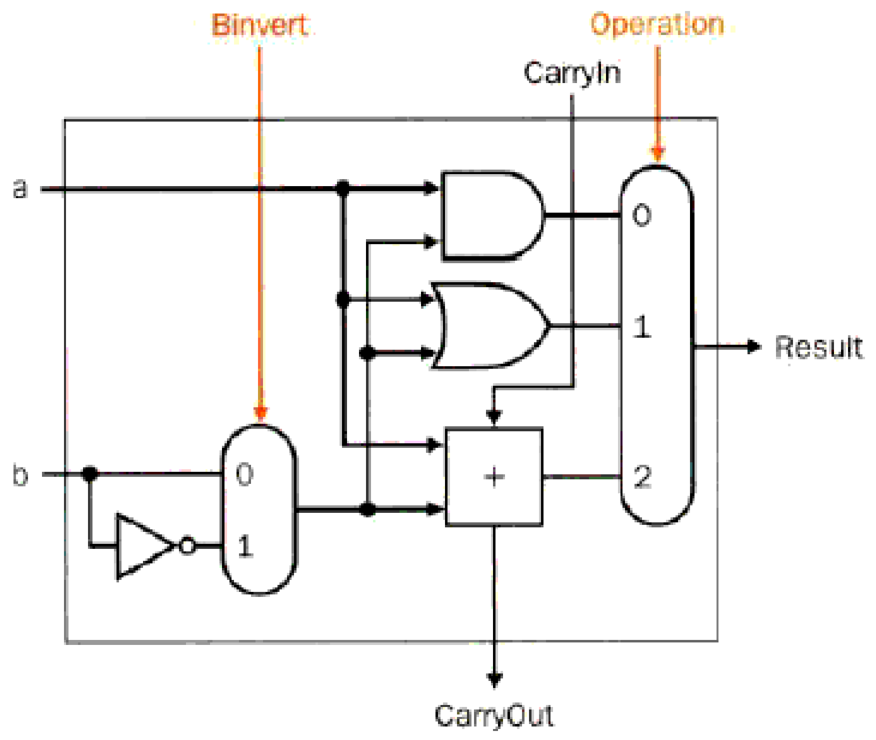


Extended 1 bit ALU-- Subtraction

- Subtraction

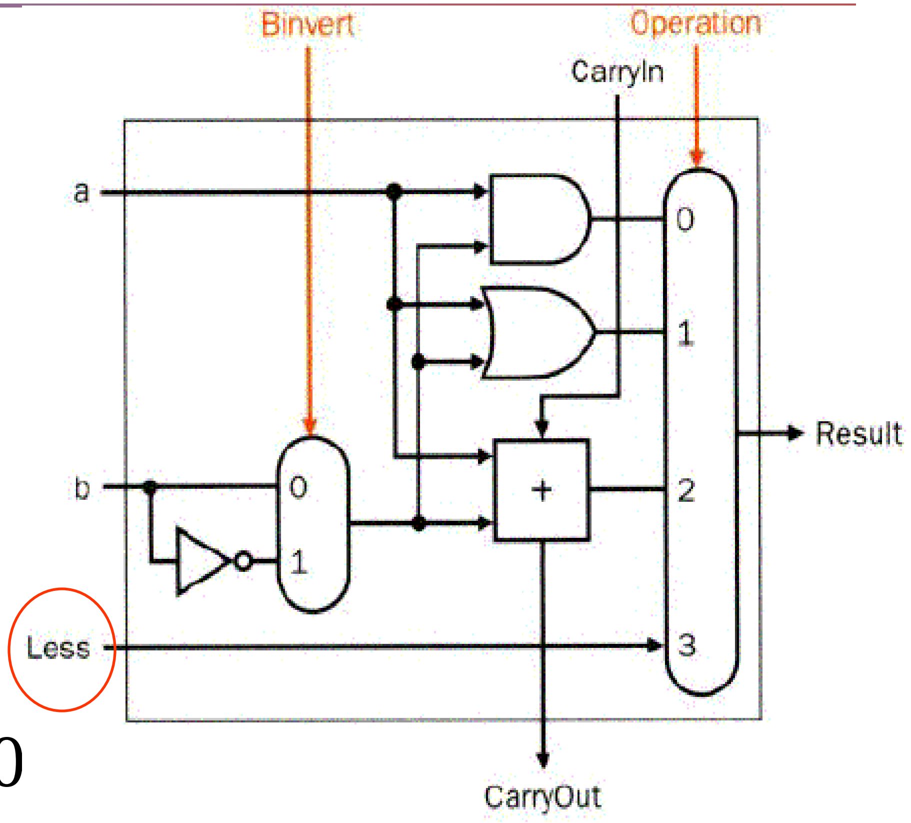
  a - b

  - 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**

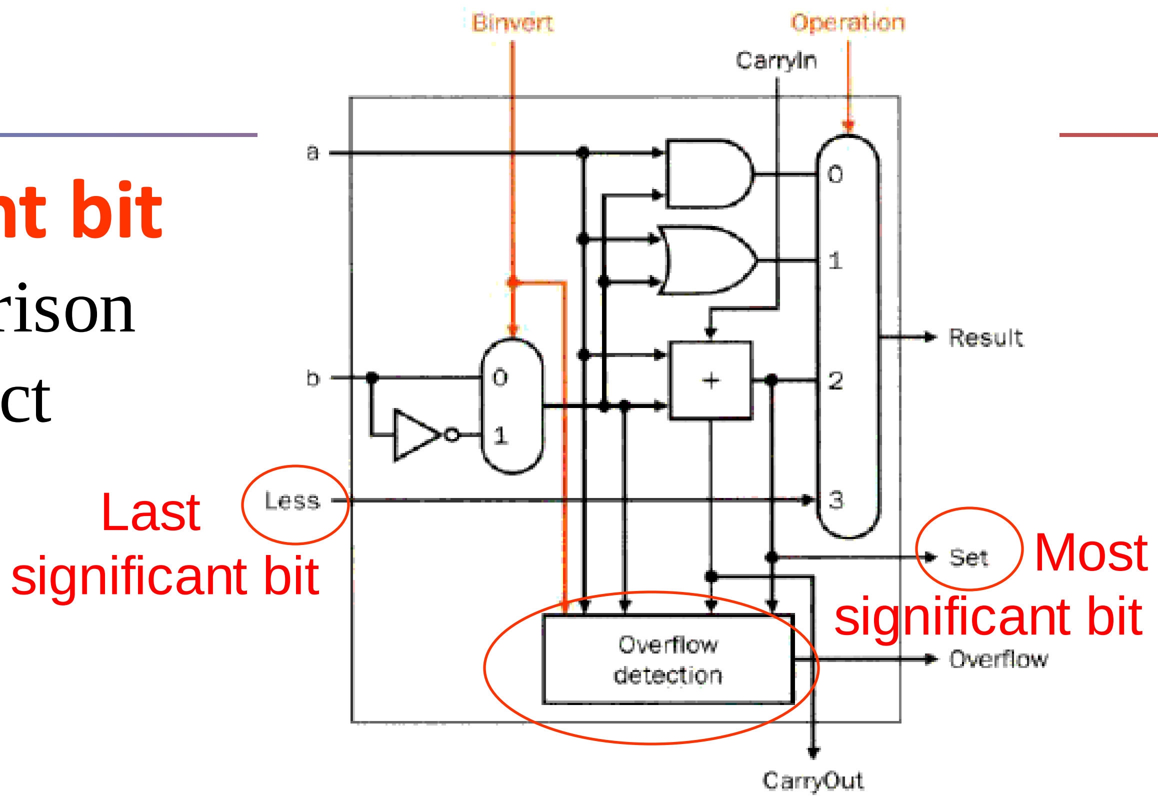


Most significant bit

- Set for comparison
- Overflow detect

Cell

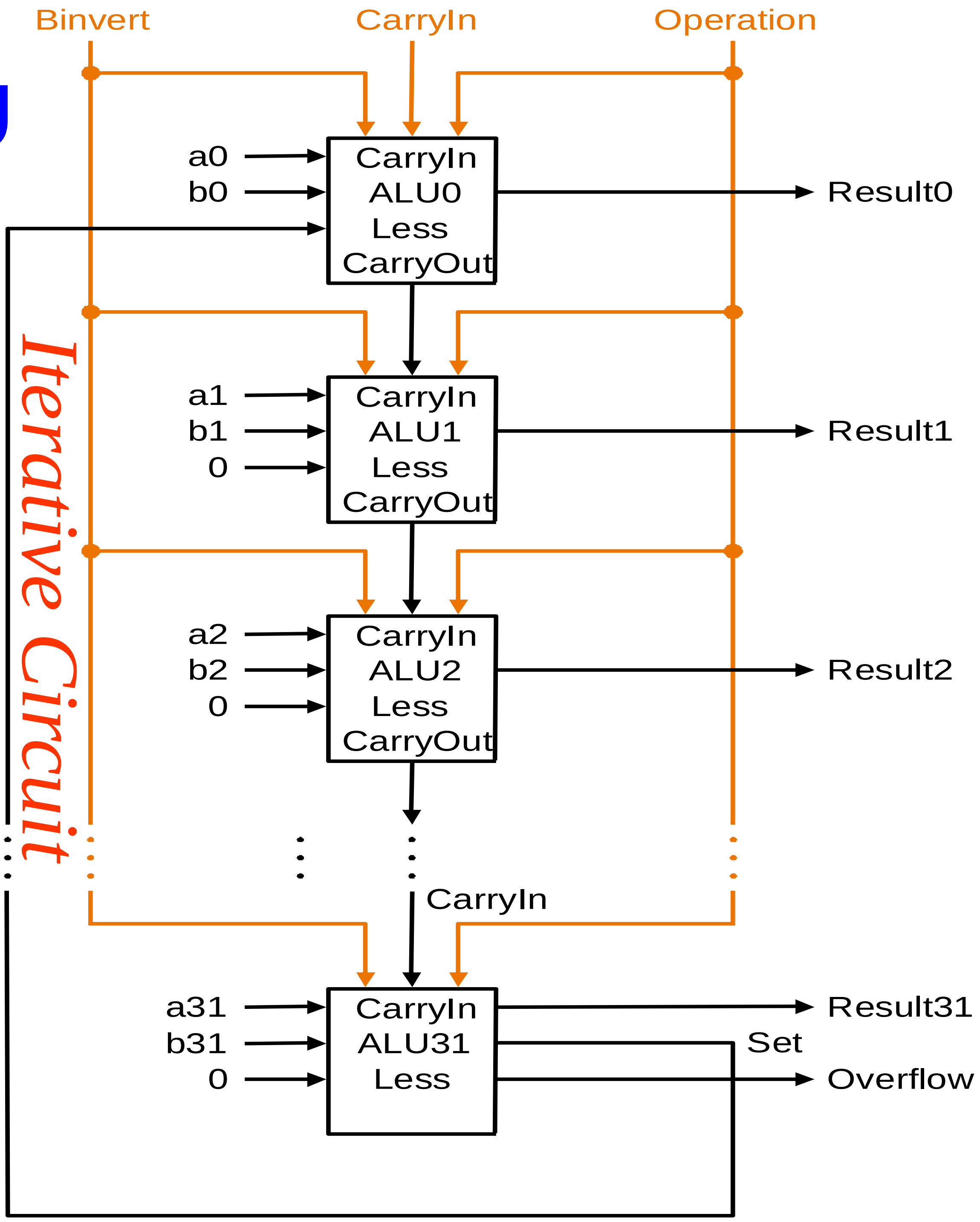
Cascade Element



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)***

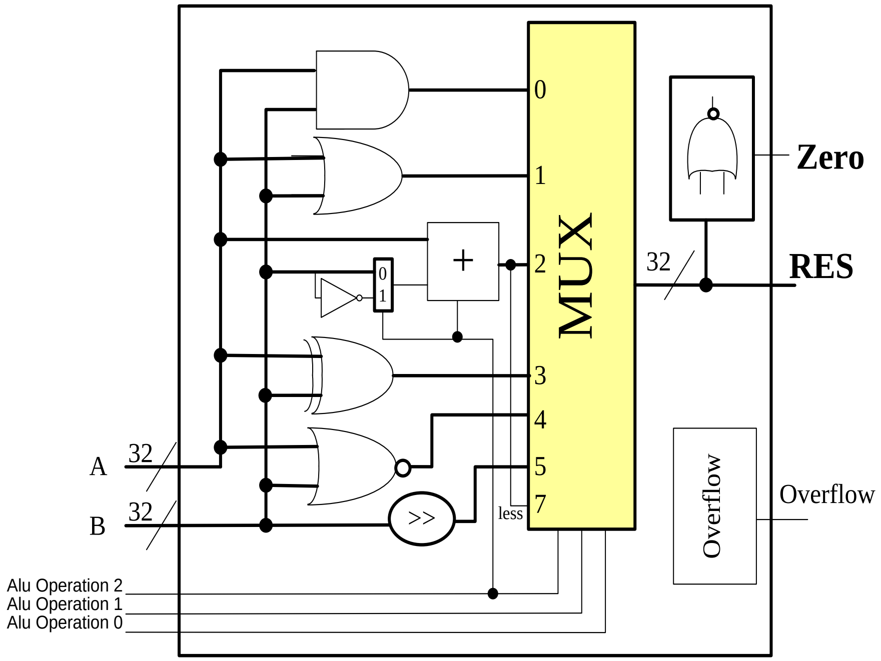
Complete ALU

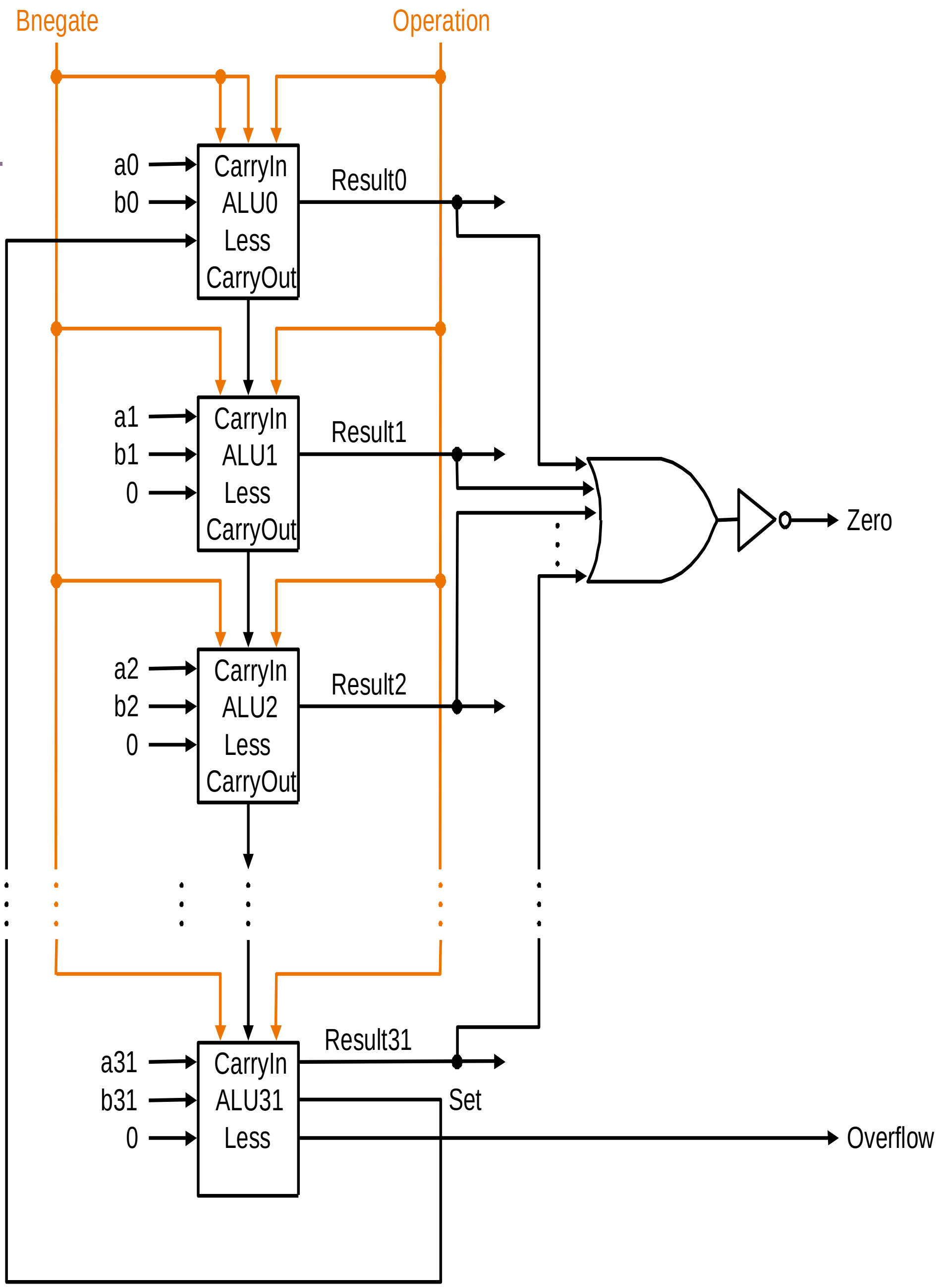
- **Input**
  - A、B
- **Control lines**
  - Binvert
  - Operation
  - Carry in
- **Output**
  - Result
  - Overflow
- **Slow, but simple**
  - Inputs parallel
  - Carry is cascaded
    - Ripple carry adder



Complete ALU —with Zero detector

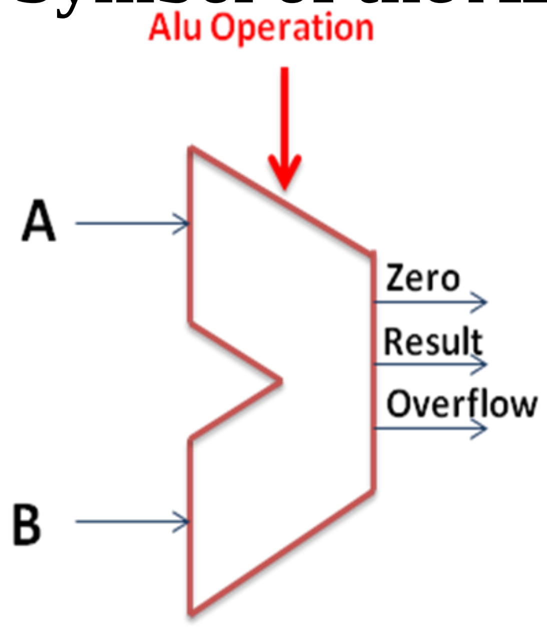
- Add a Zero detector





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
```

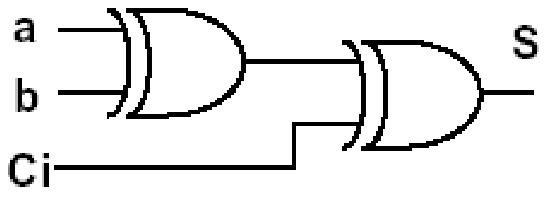
How do you write with overflow code ?

What is the difference The codes in the Synthesize?

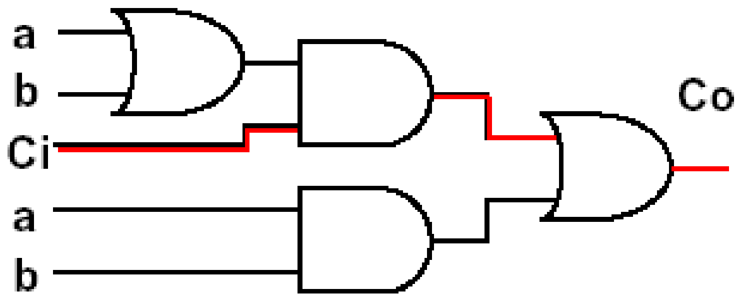
```
always @ (A or B or ALU_operation)
            case (ALU_operation)
                        3'b000: res=A&B;
                         3'b001: res=A|B;
                         3'b010: res=A+B;
                        3'b110: res=A-B;
                         3'b100: res=~(A | B);
 3'b111: res=(A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

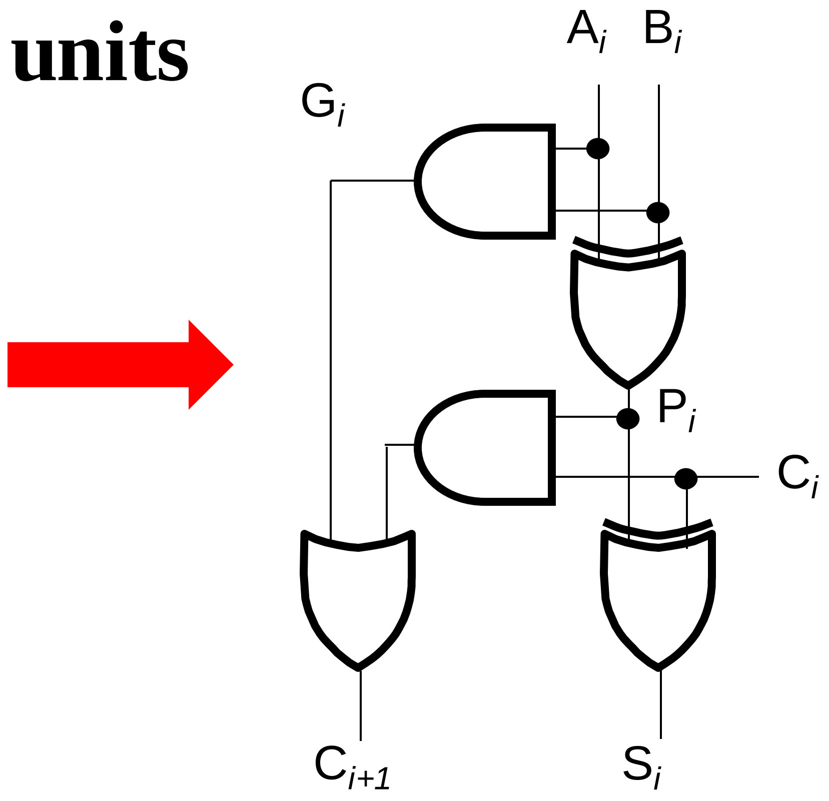
Speed considerations

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



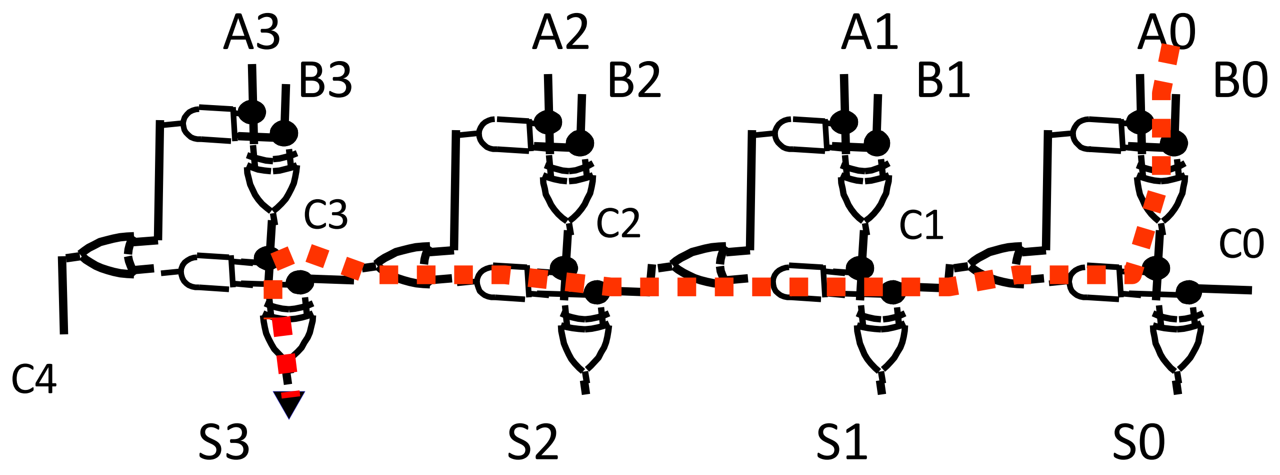
- Delay for the carry:
  - two - three units





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:**

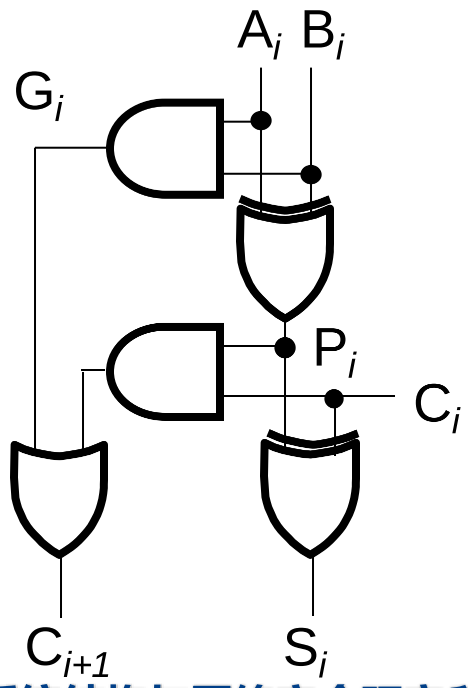
  Optimum between speed and size

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 look ahead 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:

$$C_1 = G_0 + P_0 C_0$$

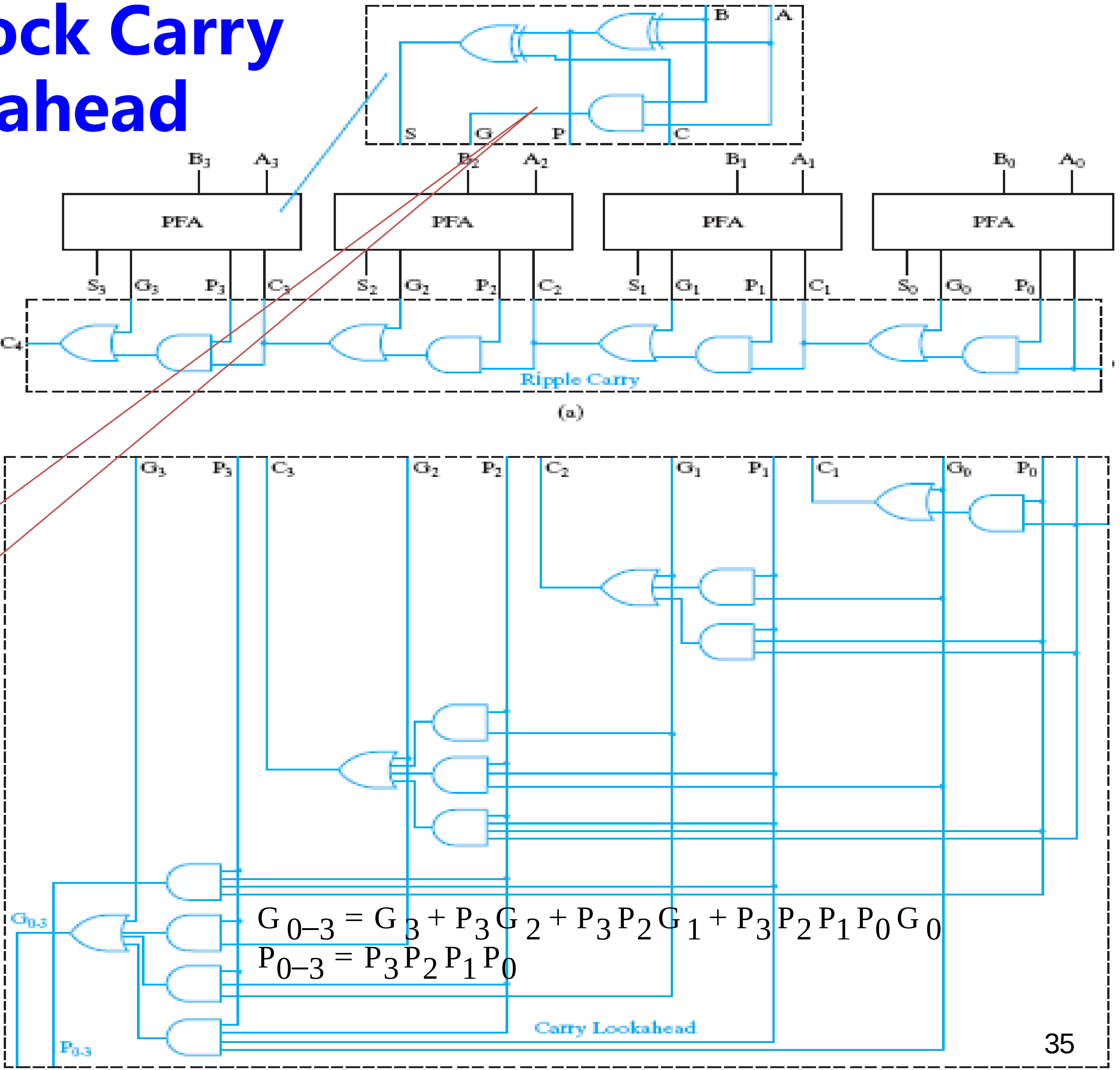
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0)$$
$$= G_1 + P_1G_0 + P_1P_0 C_0$$

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

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

Group Block Carry Lookahead

Delay？



$$G_{0-3} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1P_0G_0$$
$$P_{0-3} = P_3P_2P_1P_0$$

partial full adder

浙江大学 ZHEJIANG UNIVERSITY

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* ($G_{0\text{-}3}$) and *group propagate* ($P_{0\text{-}3}$) functions:

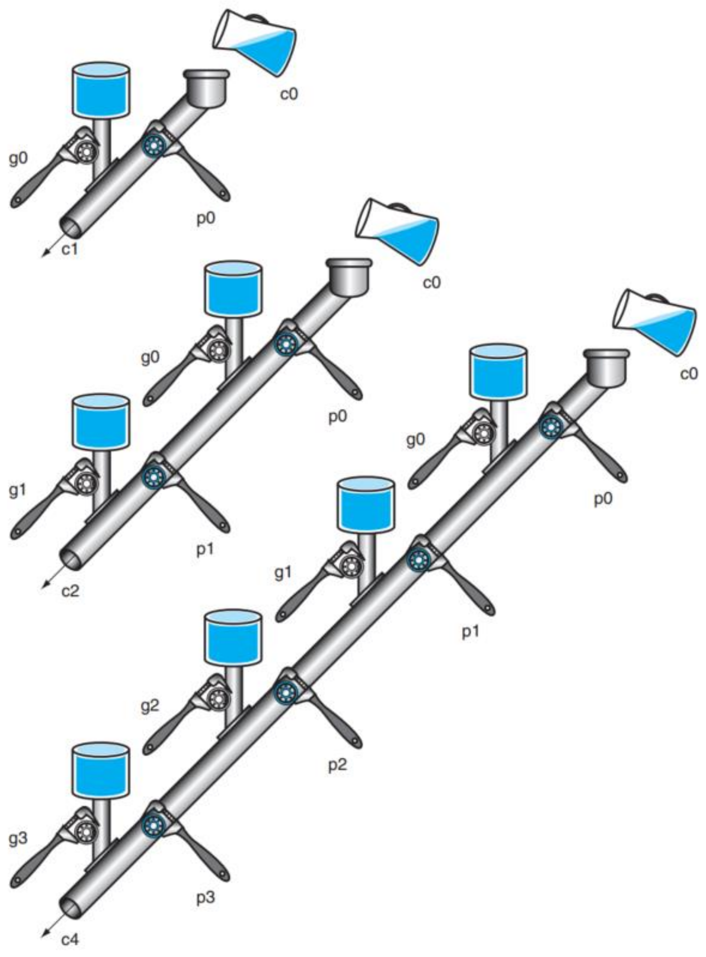
$$G_{0-3} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 P_0 G_0$$
$$P_{0-3} = P_3 P_2 P_1 P_0$$

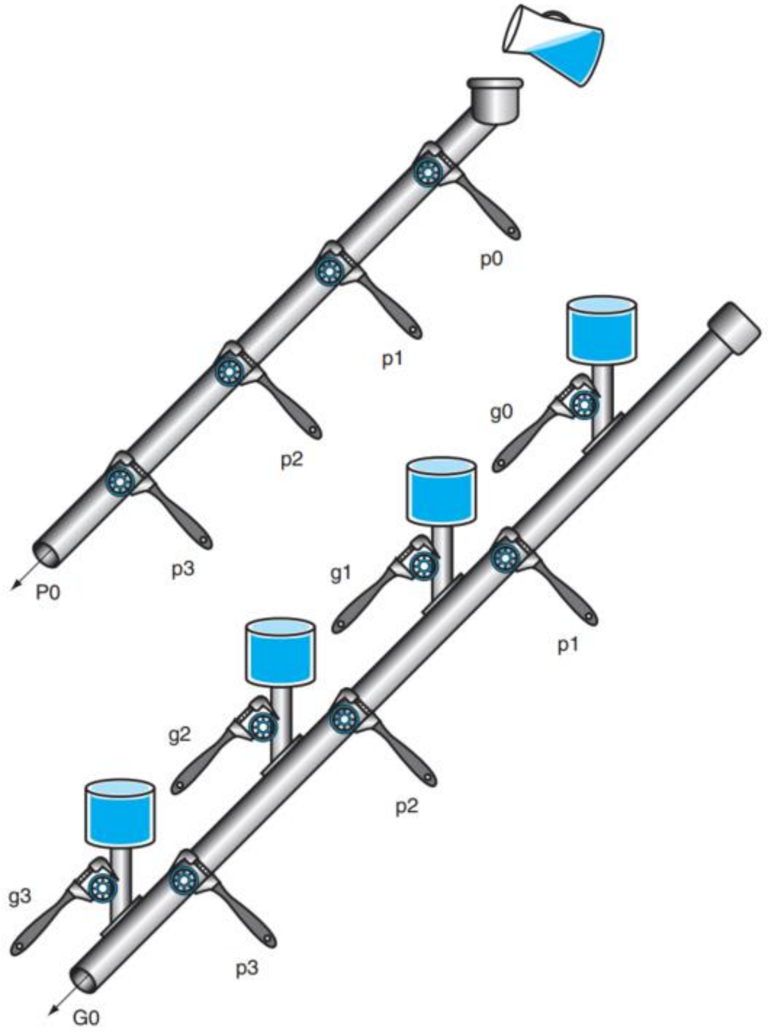
- Using these two equations:

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

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

A plumbing analogy





Extended Example: 16 carry lookahead adder

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

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

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

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

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

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

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

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

$G_{8\sim11} = G_{11}+P_{11}G_{10}+P_{11}P_10G_9+P_{11}P_{10}P_9G_8$

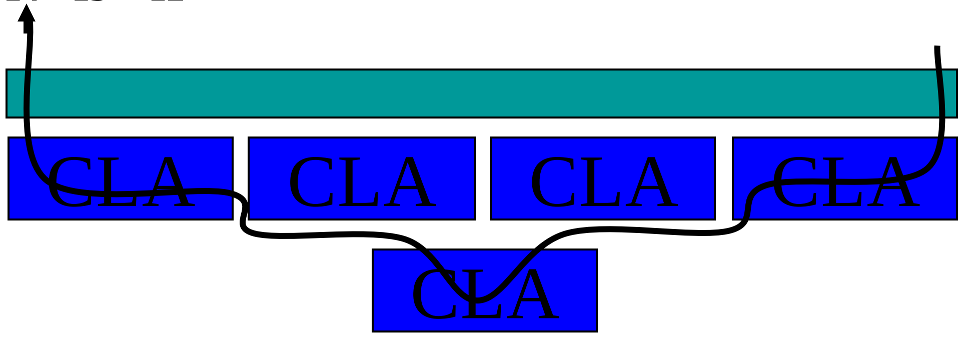
$G_{12\sim15}= G_{15}+P_{15}G_{14}+P_{15}P_{14}G_{13}+P_{15}P_{14}P_{13}G_{12}$

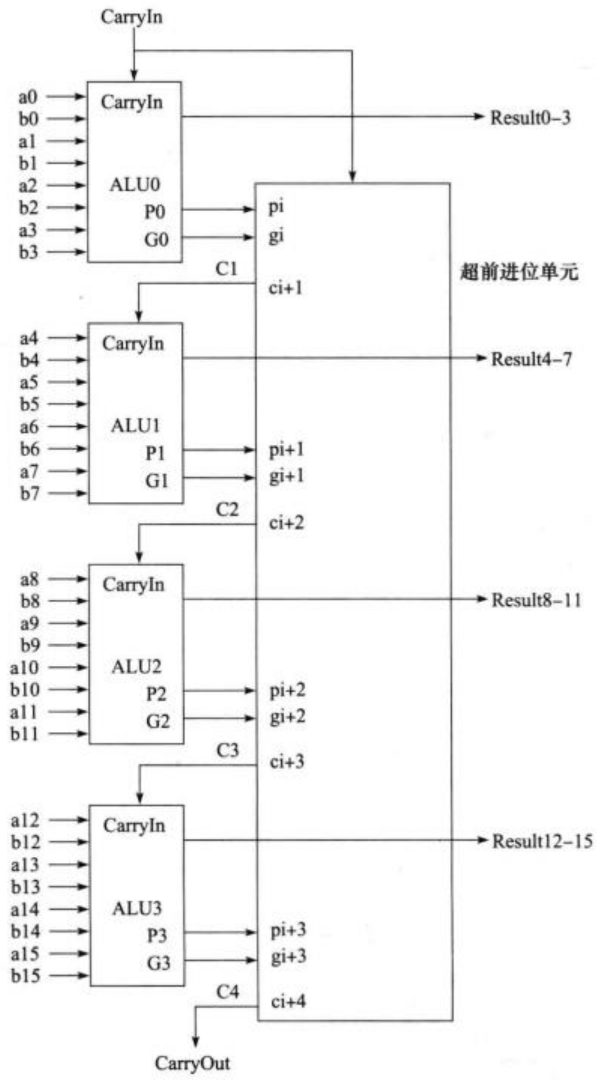
$P_{0\sim3} = P_3\ P_2\ P_1\ P_0$

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

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

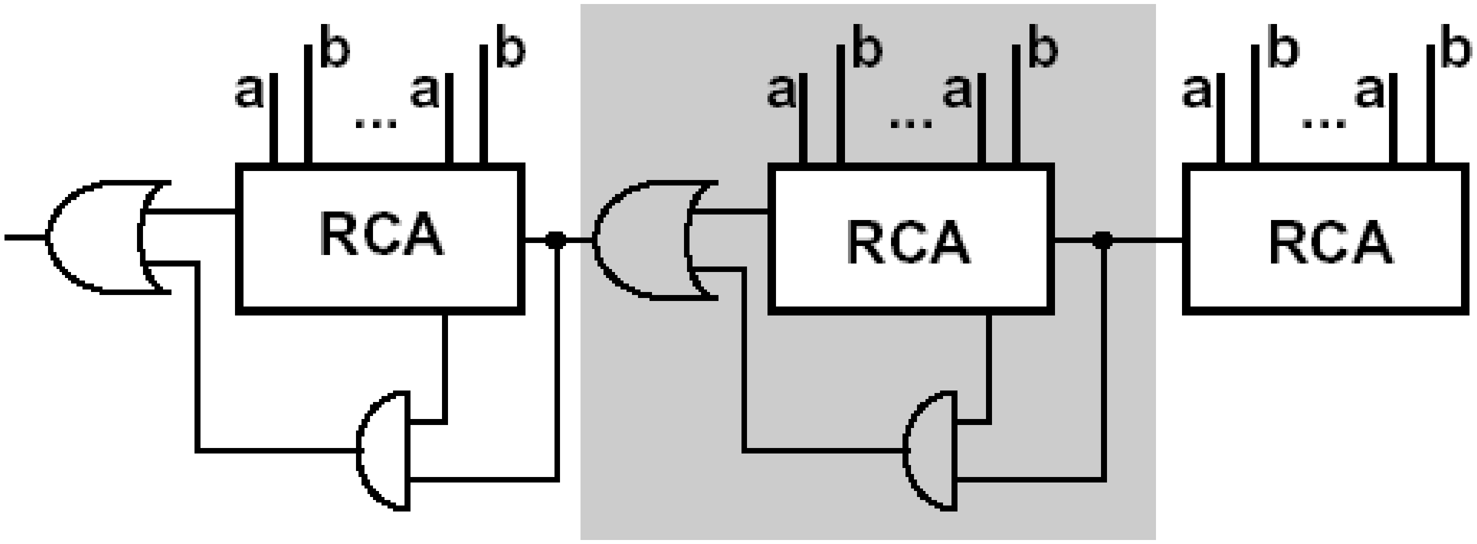
$P_{12\sim15}= P_{15}\ P_{14}\ P_{13}\ P_{12}$



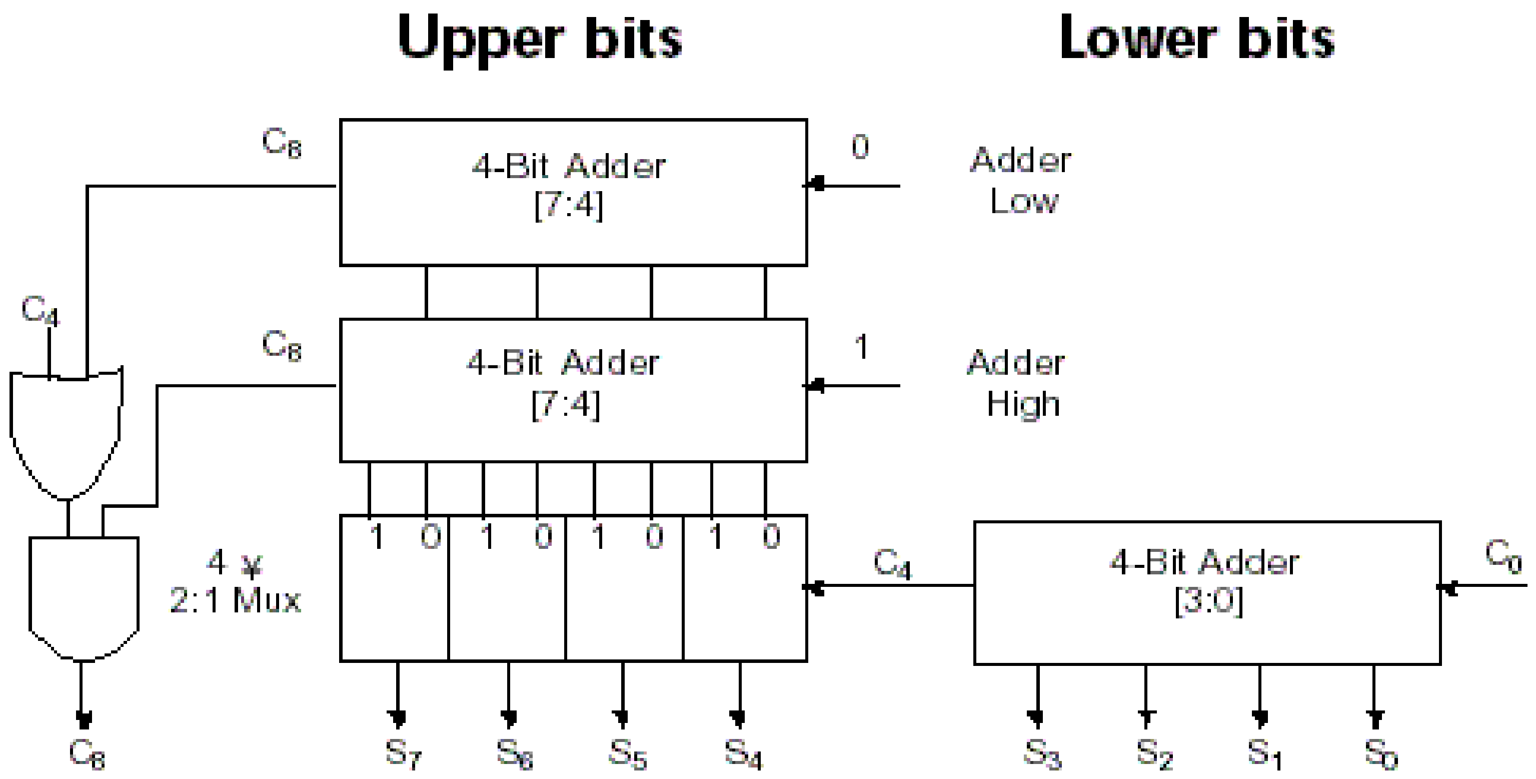

CarryIn
a0
b0
a1
b1
a2
b2
a3
b3
CarryIn
ALU0
P0
G0
Result0–3
pi
gi
C1
ci+1
超前进位单元
a4
b4
a5
b5
a6
b6
a7
b7
CarryIn
ALU1
P1
G1
Result4–7
pi+1
gi+1
C2
ci+2
a8
b8
a9
b9
a10
b10
a11
b11
CarryIn
ALU2
P2
G2
Result8–11
pi+2
gi+2
C3
ci+3
a12
b12
a13
b13
a14
b14
a15
b15
CarryIn
ALU3
P3
G3
Result12–15
pi+3
gi+3
C4
ci+4
CarryOut

Carry skip adder

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

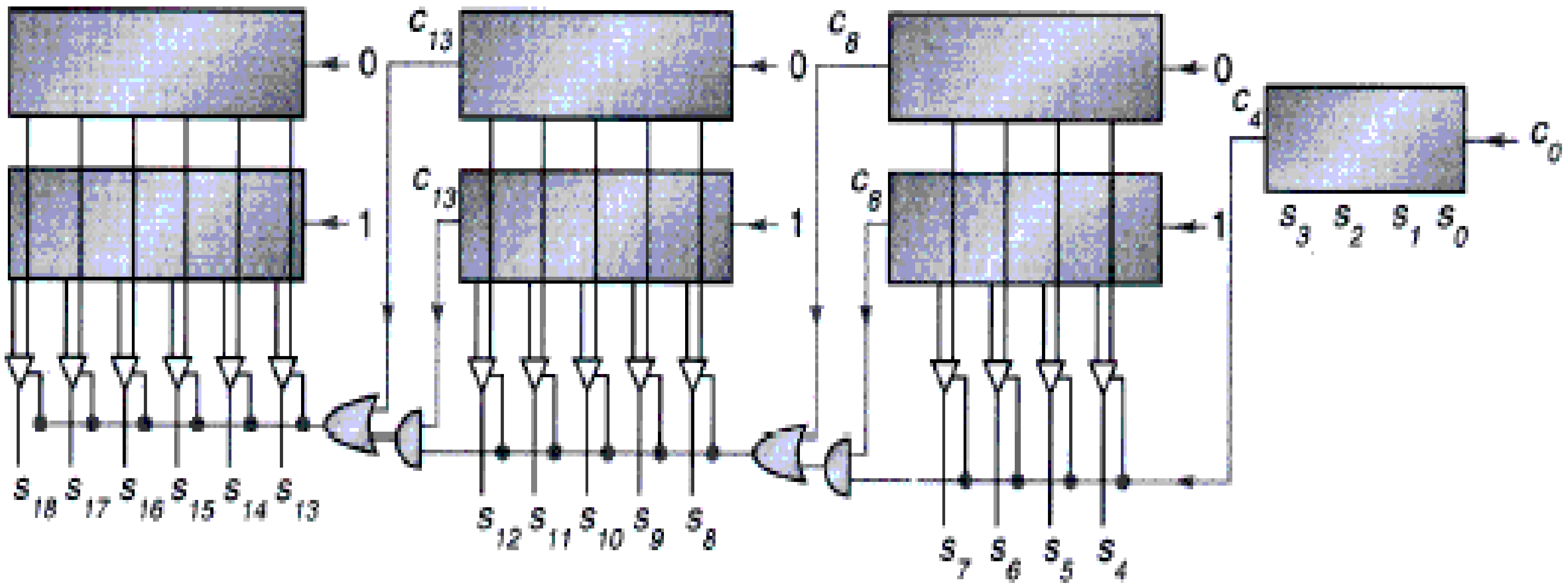


Carry select adder (CSA)



Carry select adder

- Carry selection by nibbles



0
1
c_{13}
c_{13}
0
1
c_8
c_8
0
1
c_4
c_0
s_{18} s_{17} s_{16} s_{15} s_{14} s_{13}
s_{12} s_{11} s_{10} s_9 s_8
s_7 s_6 s_5 s_4
s_3 s_2 s_1 s_0

3.4 Multiplication

- **Binary multiplication**

  Multiplicand × Multiplier

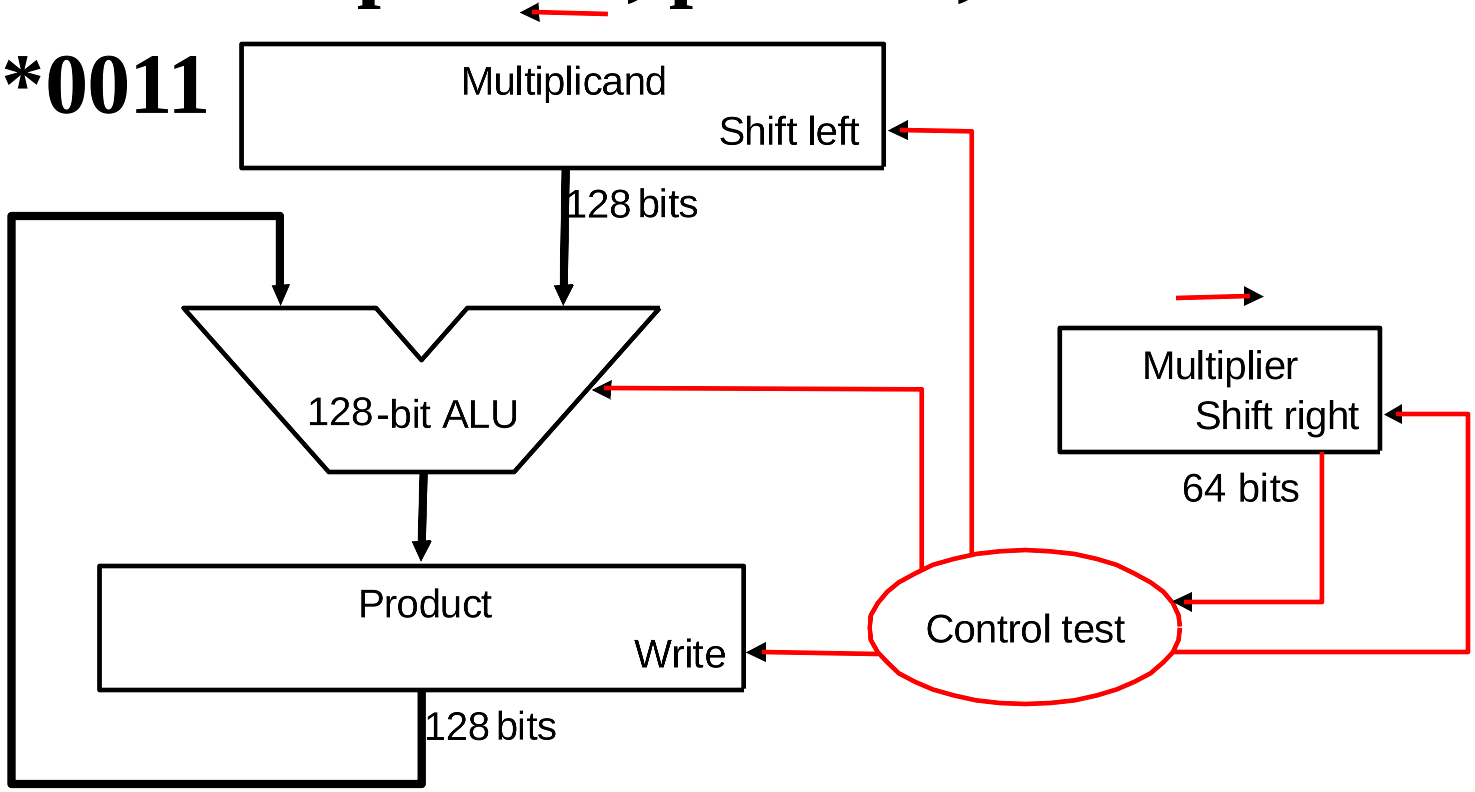
  1000 × 1001

- **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
```

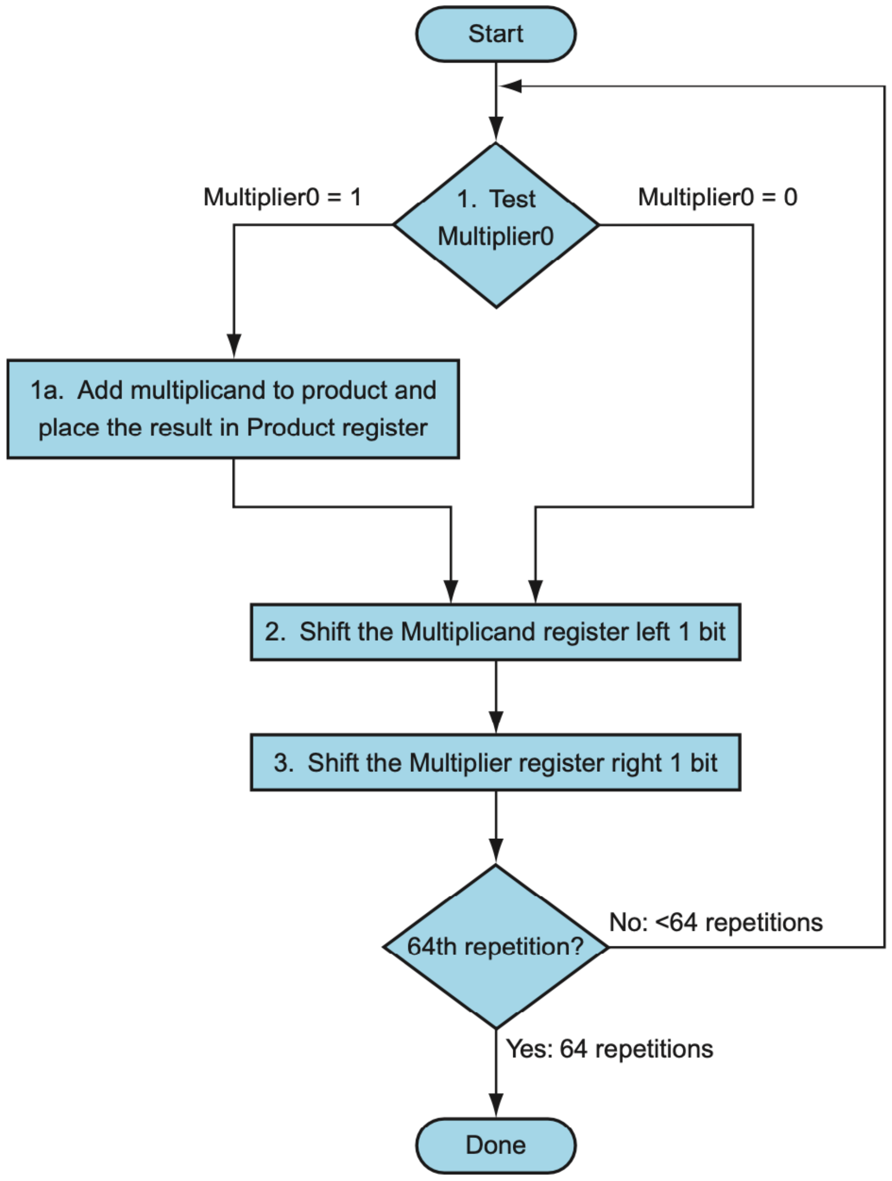
Multiplier V1– Logic Diagram

- 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!**



Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
64th repetition?
No: <64 repetitions
Yes: 64 repetitions
Done

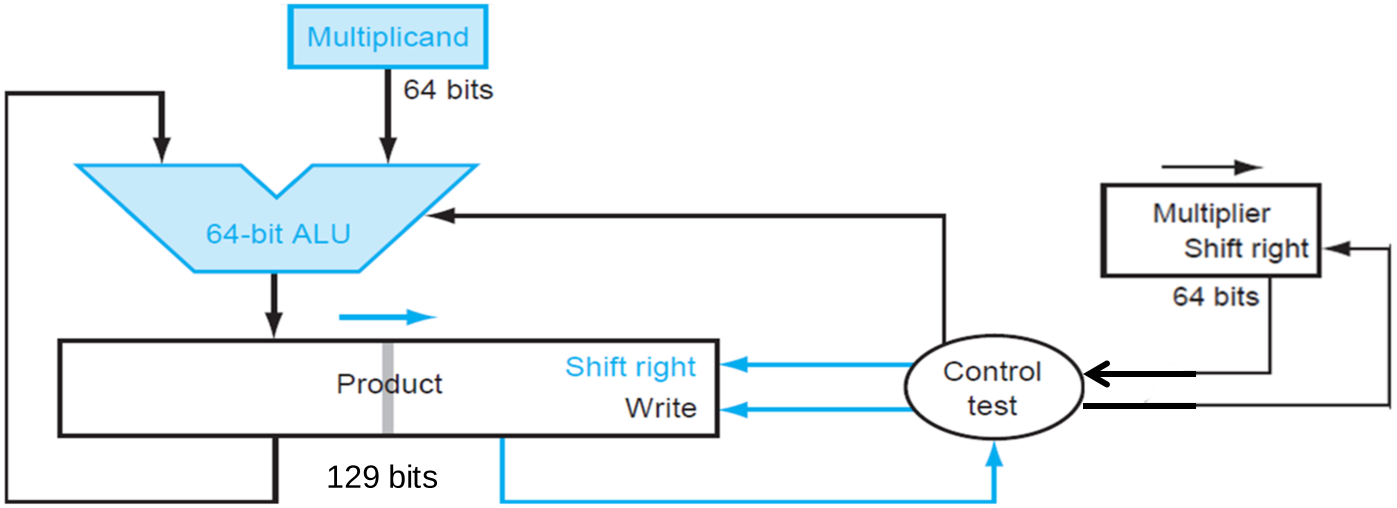
Multiplier V2

- 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
```

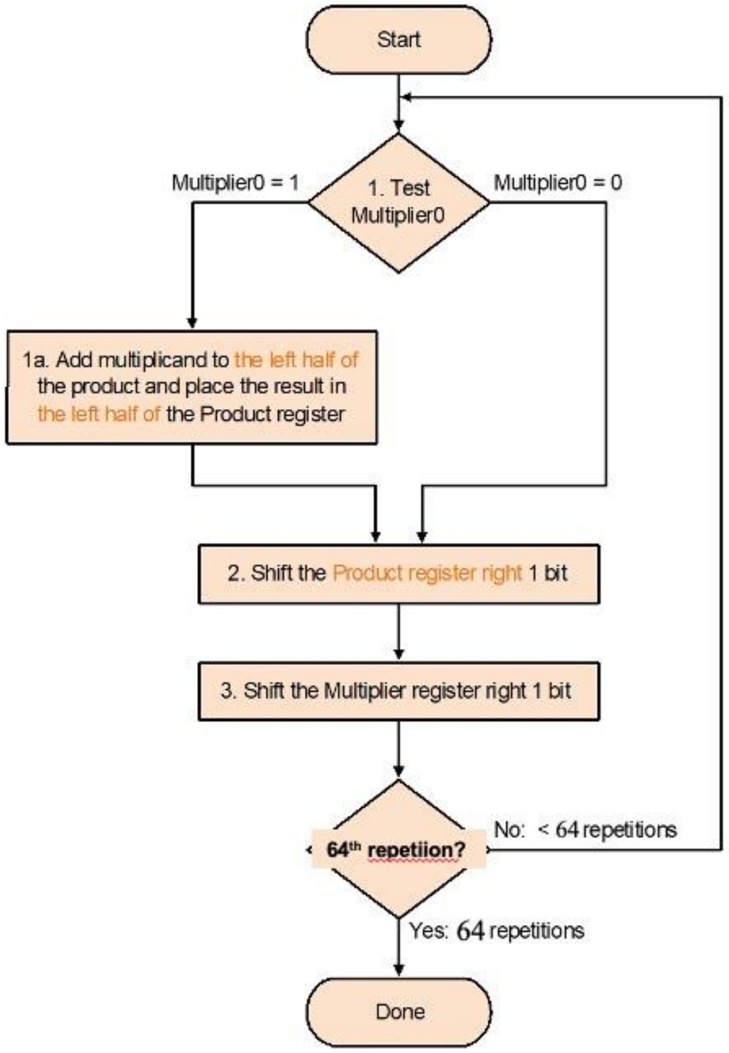
Multiplier V2-- Logic Diagram

- 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


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
3. Shift the Multiplier register right 1 bit
64th repetiion?
No: < 64 repetitions
Yes: 64 repetitions
Done

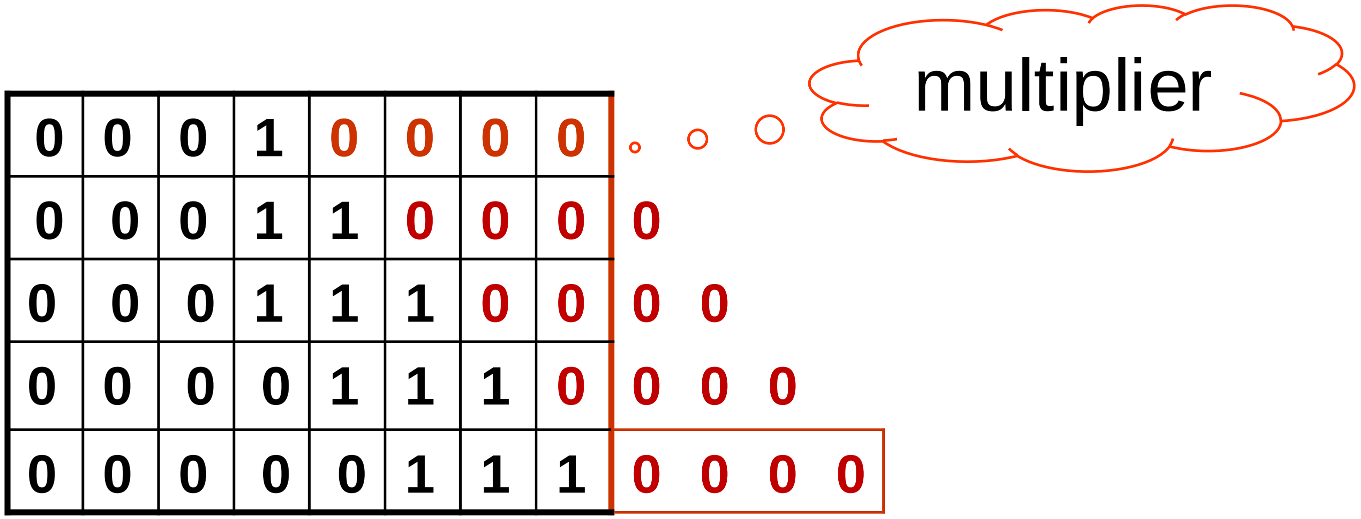
Revised 4-bit example with V2

- Multiplicand x multiplier: 0001 x 0111

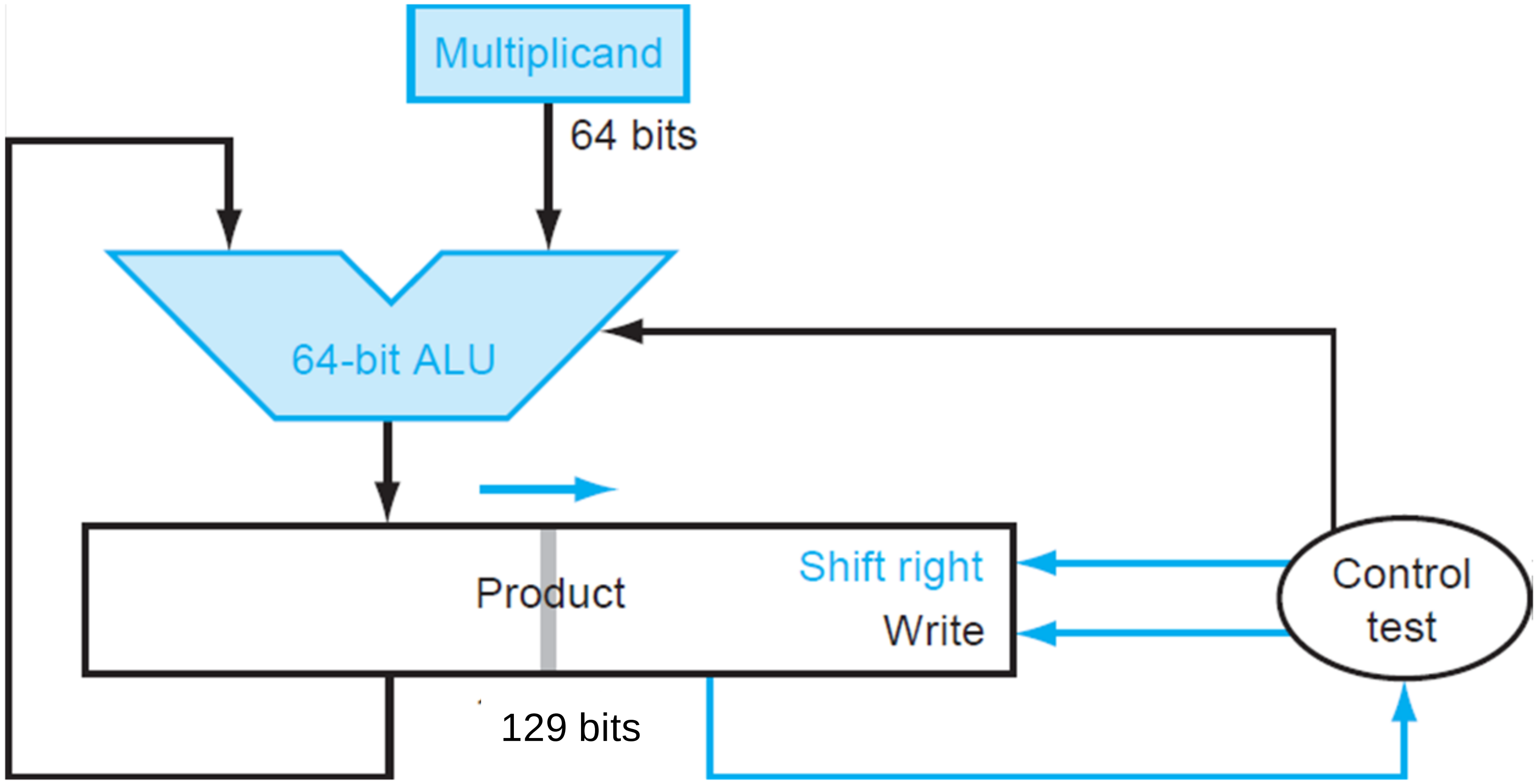
```
Multiplicand:      0001
Multiplier:×       0111
              ----------
                   00000000          #Initial value for the product
1                  00010000          #After adding 0001, Multiplier=1
---------------------------
                   00001000     0    #After shifting right 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 0000, Multiplier=0
---------------------------
                   00000111     0    #After shifting right the product one bit
```

Multiplier V3

- 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

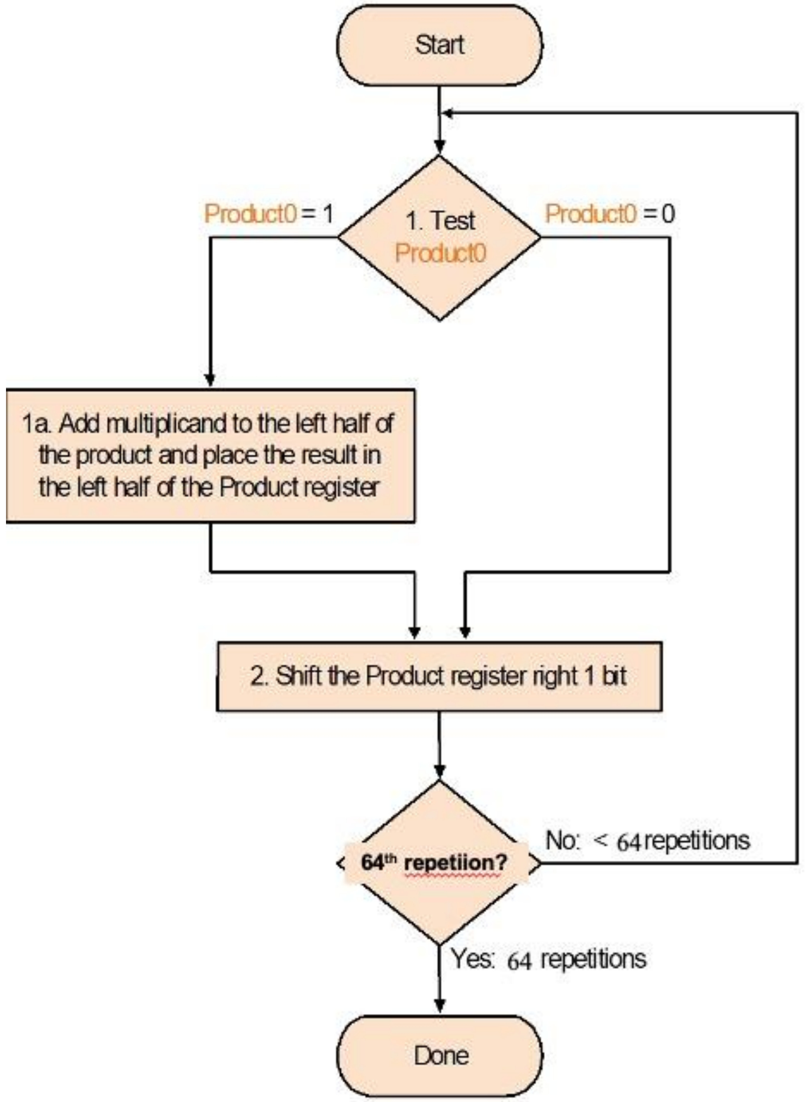


Multiplier V3 Logic Diagram



Multiplier V3--Algorithmic rule

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


Start
1. Test Product0
Product0 = 1
Product0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
64th repetiion?
No: < 64 repetitions
Yes: 64 repetitions
Done

浙江大学 ZHEJIANG UNIVERSITY

系统结构与网络安全研究所

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: $Z=y\times 10111100$

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

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

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

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

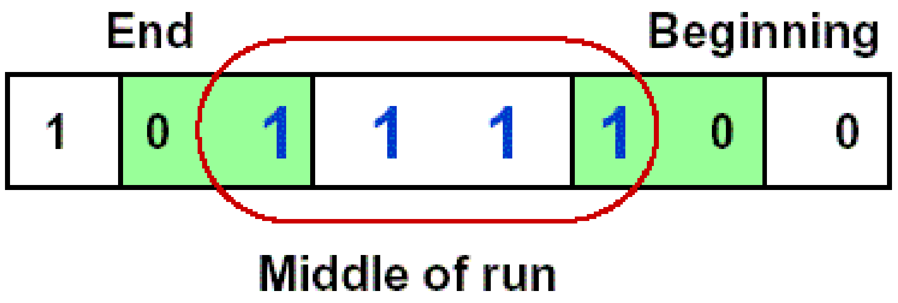
$=y(1\times 2^7+1\times 2^6+0\times 2^5+0\times 2^4+0\times 2^3+0\times 2^2-1\times 2^2+0\times 2^1+0\times 2^0)$

$$= y\times 2^7+\underbrace{y\times 1\times 2^6}_{\text{add}}+\underbrace{0\times 2^5+0\times 2^4+0\times 2^3+0\times 2^2}_{\text{Only shift}}\underbrace{-y\times 2^2}_{\text{sub}}+\underbrace{0\times 2^1+0\times 2^0}_{\text{Only shift}})$$

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	0000111**10**000
1	1	middle of '1'	00001**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-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 11*01* 0
	2: shift right Product	0010	1111 011*0 1*
2	1.b:01→Prod=Prod+Mcand	0010	0001 01*10* 1
	2: shift right Product	0010	0000 101*1 0*
3	1.c:10→Prod=Prod-Mcand	0010	1110 10*11* 0
	2: shift right Product	0010	1111 010*1 1*
4	1.d: 11 → ***no operation***	0010	1111 0101 1
	2: shift right Product	0010	1111 1010 1

浙江大学 ZHEJIANG UNIVERSITY

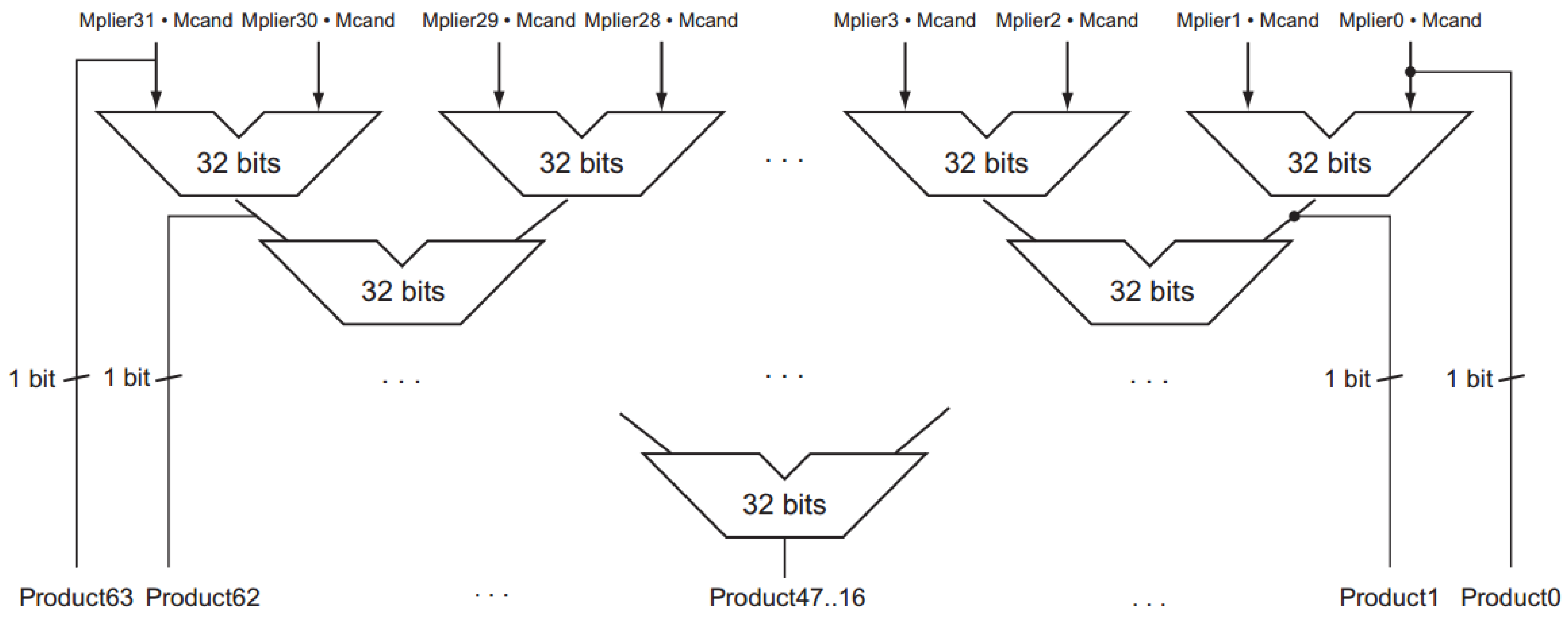
系统结构与网络安全研究所

13 * (-11) = - 143 -13= +10011
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

- Unrolls the loop

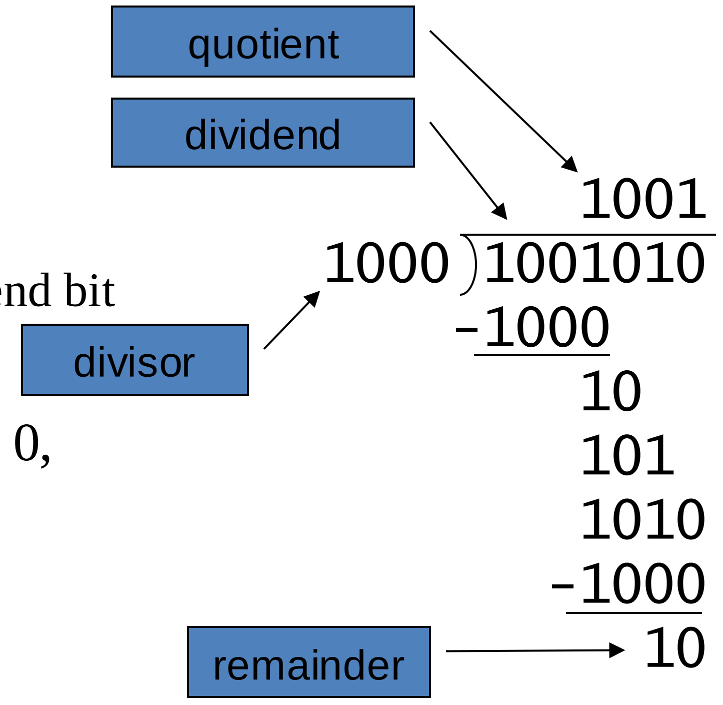


RISC-V Multiplication

- **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

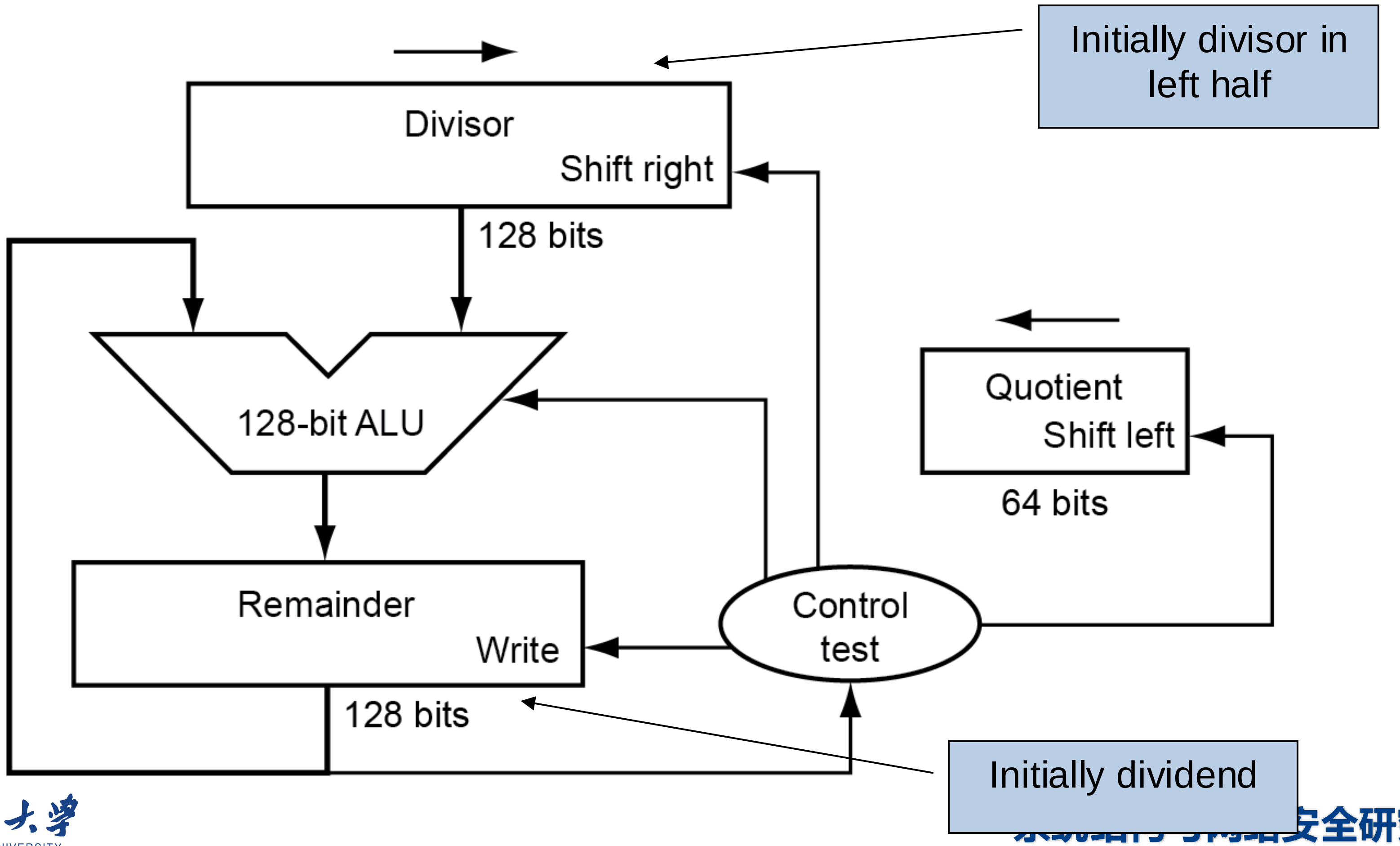
- **Check for 0 divisor**
- **Long division approach**
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- **Restoring division**
  - Do the subtract, and if remainder goes < 0, add divisor back
- **Signed division**
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



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

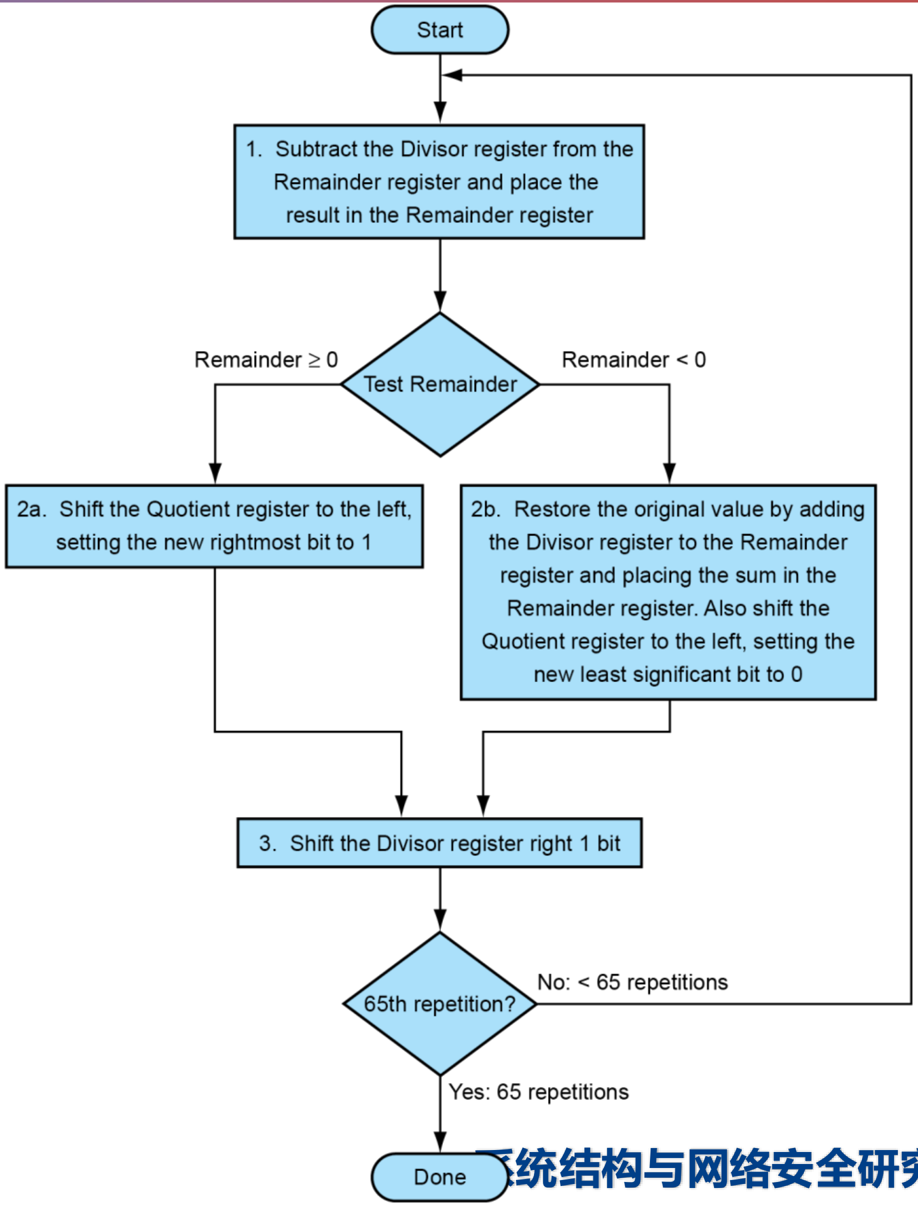
Division V1 --Logic Diagram

- At first, the divisor is in the left half of the divisor 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'



Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
65th repetition?
No: < 65 repetitions
Yes: 65 repetitions
Done

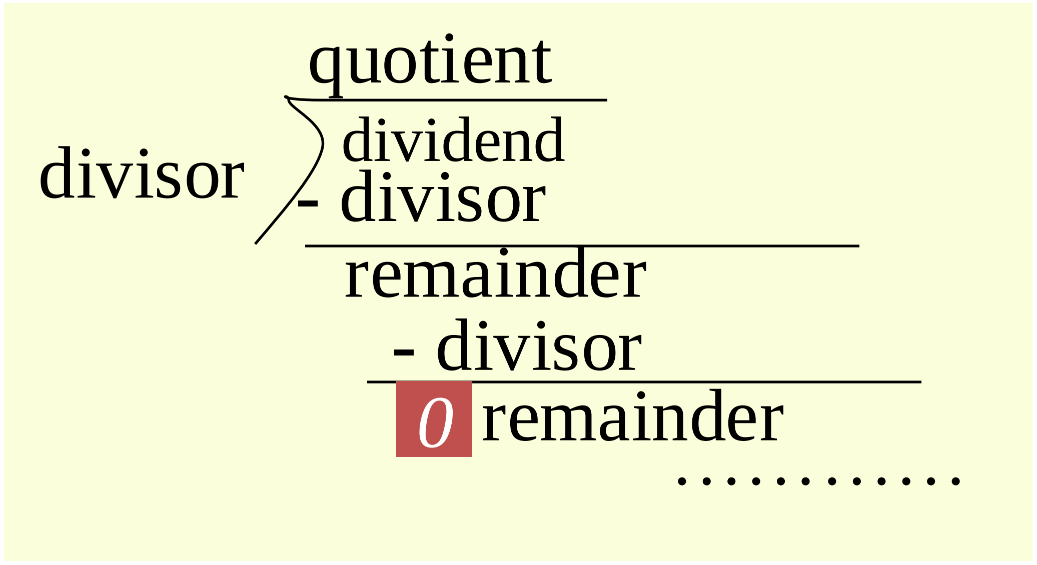
Example 7/2 for Division V1

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	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

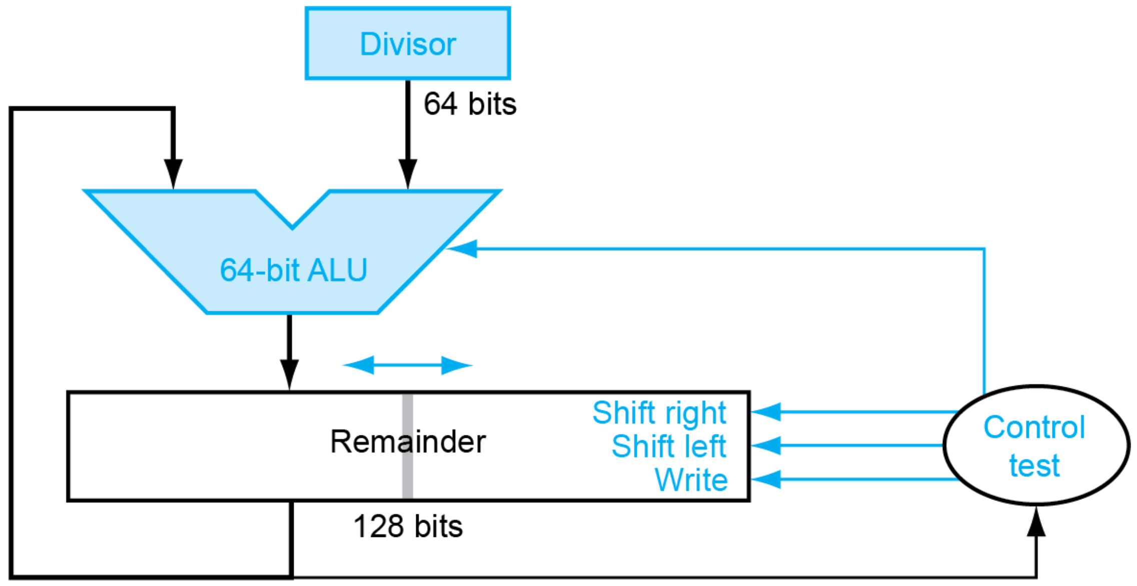
1.Why should the divisor be shifted right one bit each time?

2.Why should the divisor be placed in the **left half** of the divisor register.



Modified Division

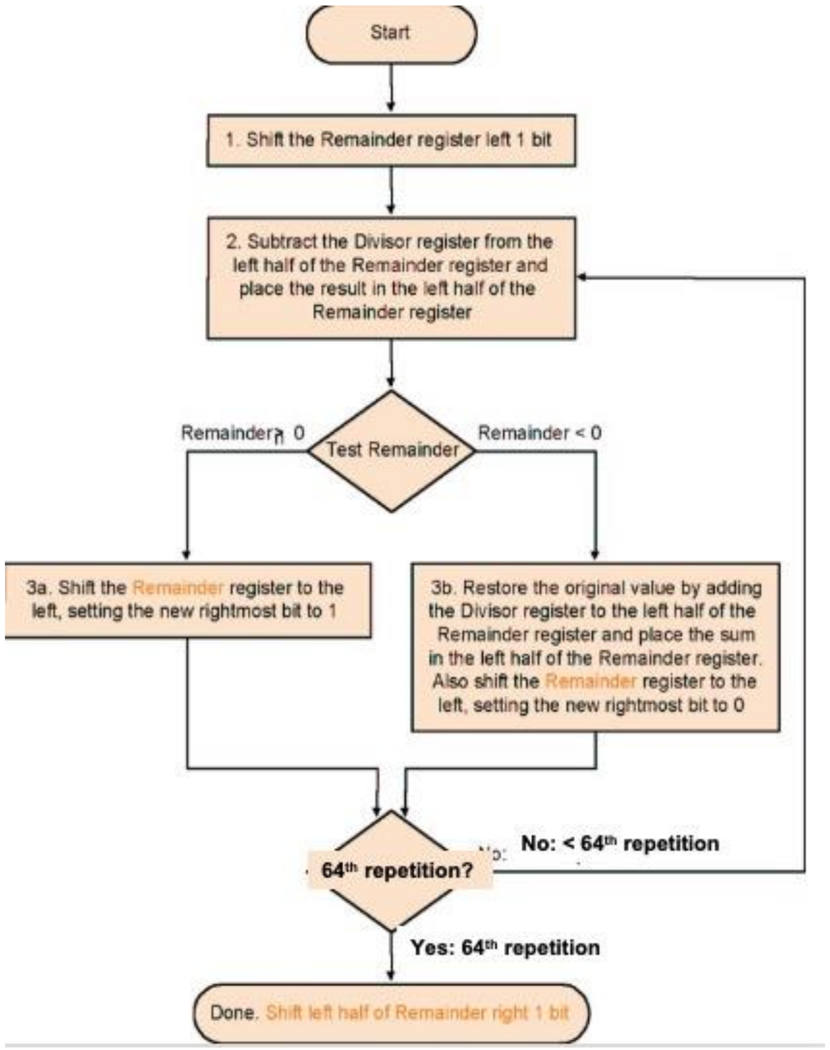
- Reduction of Divisor and ALU width by half
- Shifting of the remainder
- Saving 1 iteration
- Remainder register keeps quotient No quotient register required



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

Algorithm V3

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



Start
1. Shift the Remainder register left 1 bit
2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
3a. Shift the Remainder register to the left, setting the new rightmost bit to 1
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0
64th repetition?
No: < 64th repetition
Yes: 64th repetition
Done. Shift left half of Remainder right 1 bit

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 110*0***
2	1.Rem=Rem-Div	0010	1111 1100
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0011 10*00***
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
- **One 64 bit register : Hi & Lo**
  - Hi: Remainder, Lo: Quotient
- **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 range
  - Fractions
  - Numbers like e (2.71828) and π (3.14159265....)
- **Representation**
  - Sign
  - Significand
  - Exponent
  - More bits for significand: more accuracy
  - More bits for exponent: increases the range

Floating point numbers

- **Form**
  - Arbitrary 363.4 • 10^{34}
  - Normalised 3.634 • 10^{36}
- **Binary notation**
  - Normalised 1.xxxxxx • 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 RISC-V**

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

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:

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

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

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
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value**
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 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 $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ 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

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

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

  $9.999 \bullet 10^1$

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

- Addition

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

- Normalisation

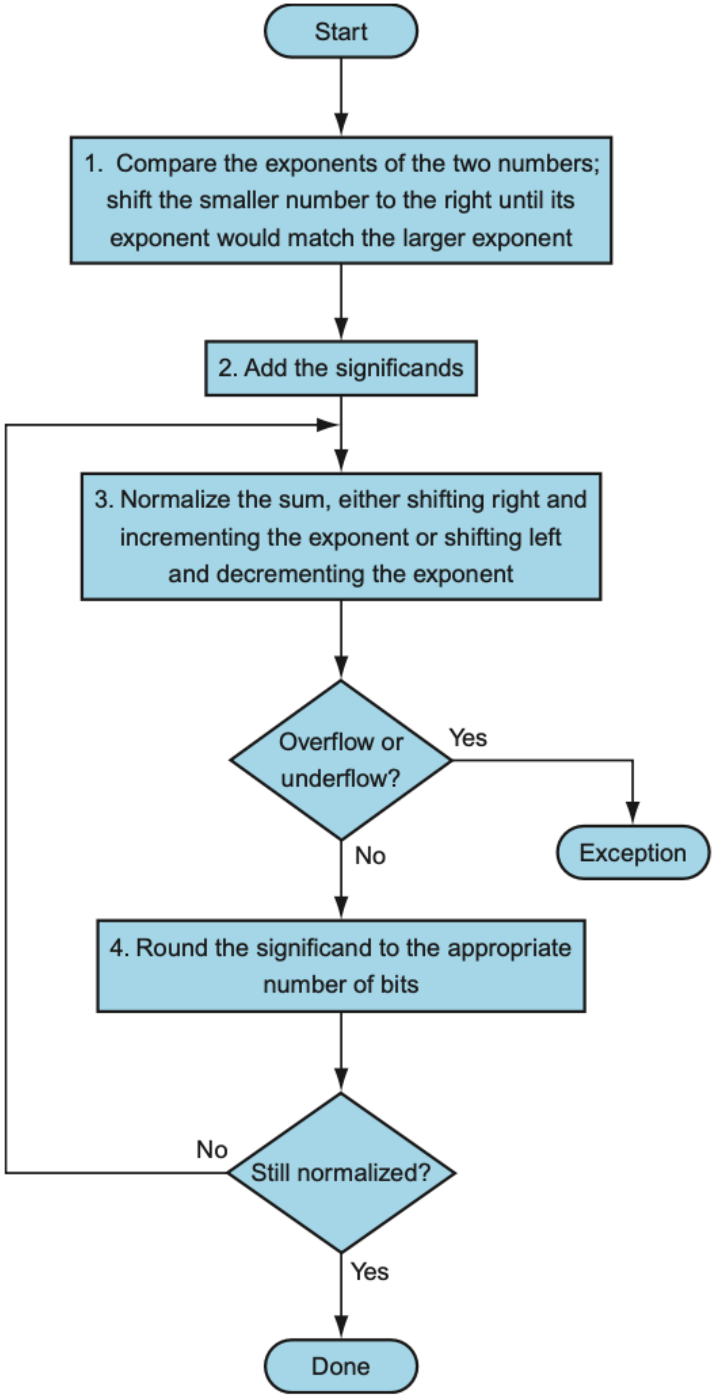
  $1.0015 \bullet 10^2$

- Rounding

  $1.002 \bullet 10^2$

Algorithm

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


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

Example y=0.5+(-0.4375) in binary

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

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

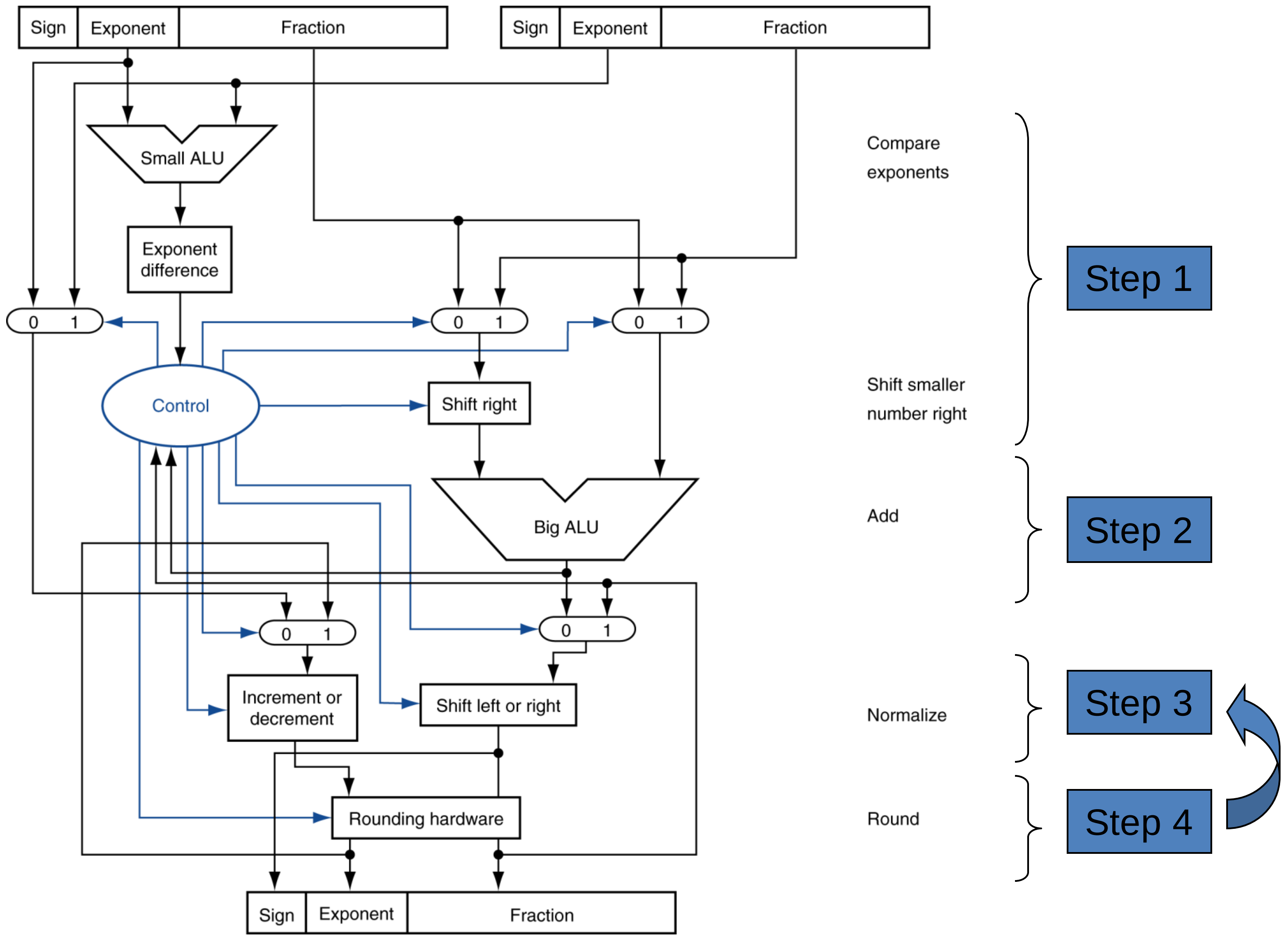
$$\begin{array}{rl} & 1.000_2 \times 2^{-1} \\ +) & -\ 0.111_2 \times 2^{-1} \\ \hline & 0.001_2 \times 2^{-1} \end{array}$$

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

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

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

Algorithm


Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Big ALU
0 1
0 1
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Fraction
Compare exponents
Shift smaller number right
Add
Normalize
Round
Step 1
Step 2
Step 3
Step 4

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 $\times$ 2^3

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

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

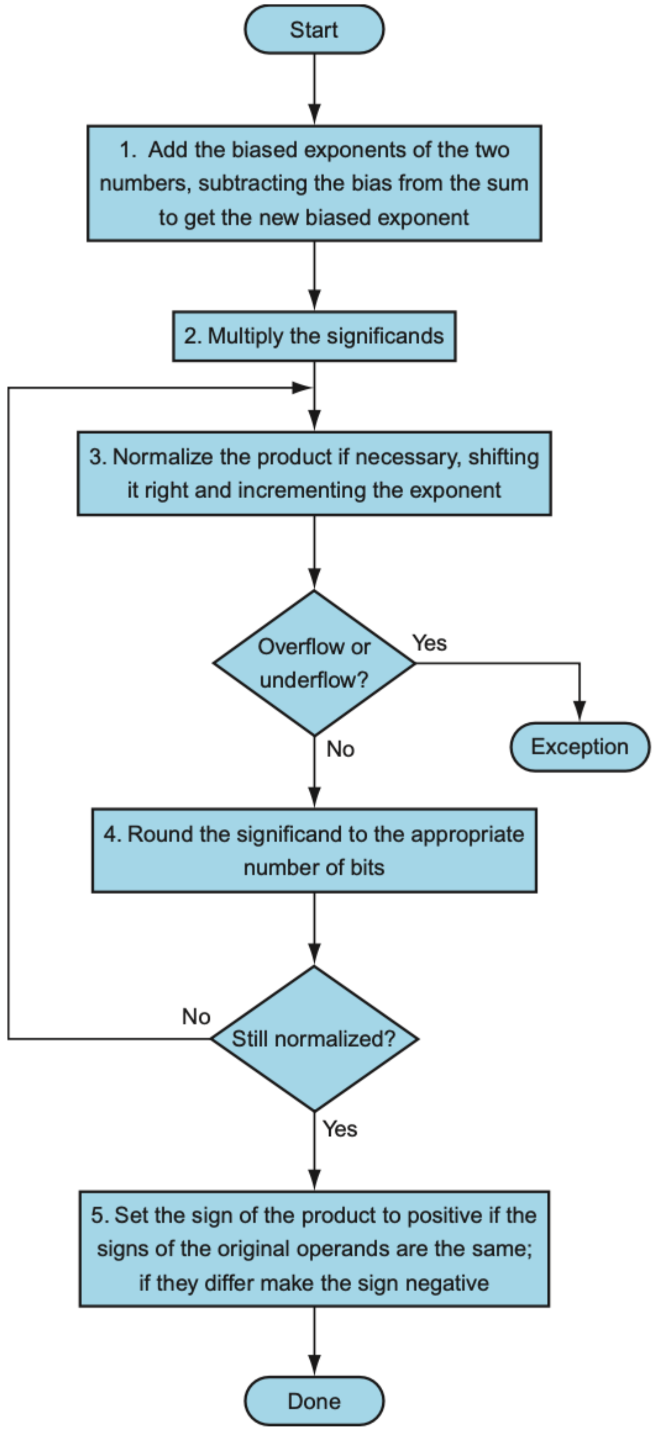
```
  10000010
 +10000011
 110000101
```

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

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

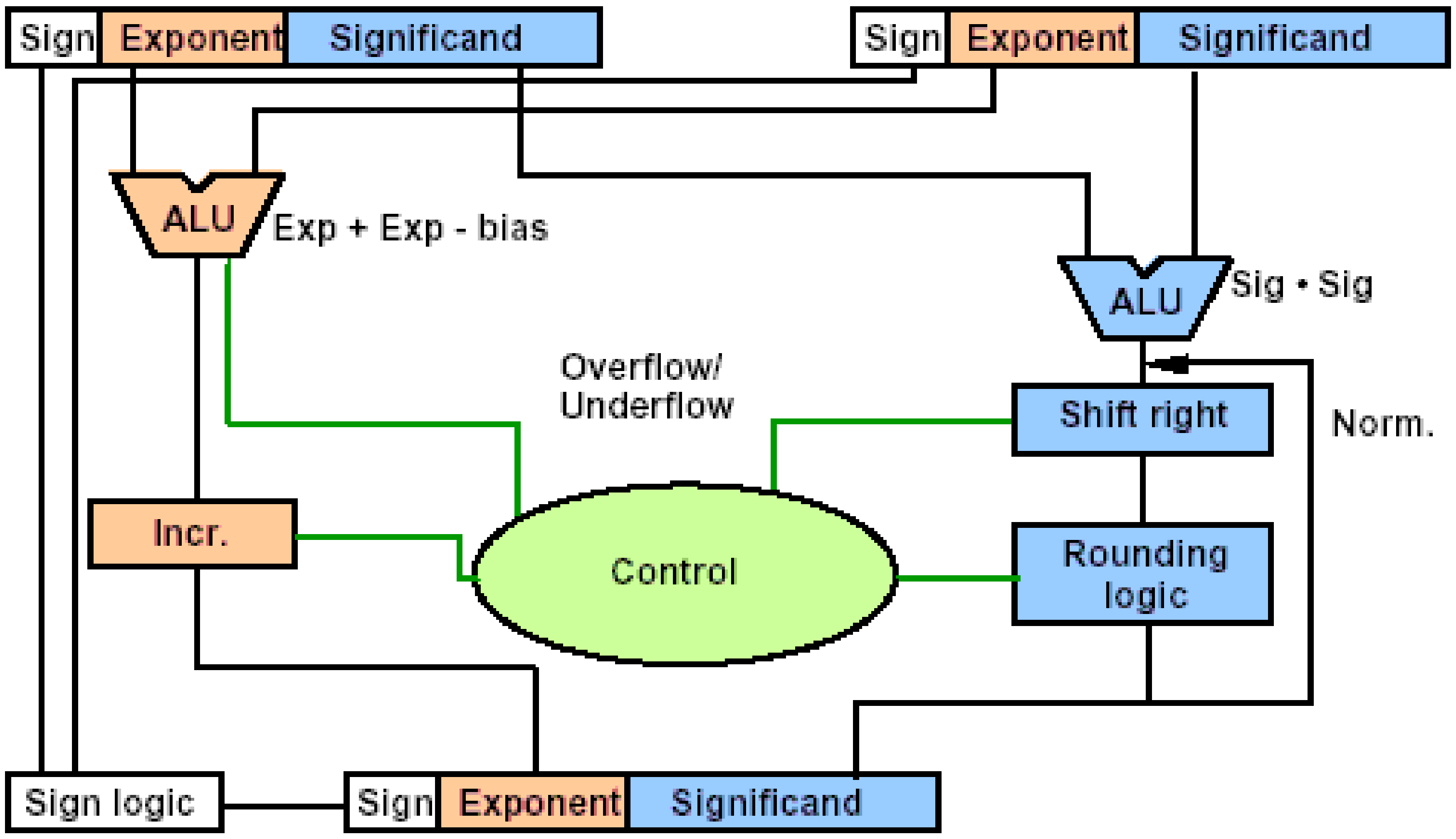
Multiplication

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



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

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 significands
- Normalisation
- Rounding
- Sign

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**

Accurate Arithmetic

- **Guard**: the first of two extra bits.
- **Round**: method to make the immediate floating-point result fit the floating-point format.
- **Units in the last place(ulp):**The number of bits in error in the least significant bits of the significant between the actual number and the number that can be represented.

$$\begin{array}{r} 2.3400_{\text{ten}} \\ +\ 0.0256_{\text{ten}} \\ \hline 2.3656_{\text{ten}} \end{array} \qquad\qquad \begin{array}{r} 2.34_{\text{ten}} \\ +\ 0.02_{\text{ten}} \\ \hline 2.36_{\text{ten}} \end{array}$$

Round to nearest even

- **Rounding modes:**
  - **Always round up (to 正无穷）**
  - **Always round down（to 负无穷）**
  - **Truncate**
  - **Round to nearest even**
- **Rounding to nearest even (Keep LSB to 0 when extra bits are 100)**

```
0101010100 | 011 ->      0101010100   (+0)
0101010101 | 011 ->      0101010101   (+0)
-------------------------------------------------------
0101010100 | 100 ->      0101010100   (+0, keep LSB to 0)
0101010101 | 100 ->      0101010110   (+1, keep LSB to 0)
-------------------------------------------------------
0101010100 | 101 ->      0101010101   (+1)
0101010101 | 101 ->      0101010110   (+1)
```

精度：0.5ulp

Homework

- 3.7, 3.20, 3.26, 3.27, 3.32

END