Computer Arithmetic

COE608: Computer Organization and Architecture

Dr. Gul N. Khan
http://www.ee.ryerson.ca/~gnkhan
Electrical and Computer Engineering
Ryerson University

Overview

- Computer Arithmetic: Overview
  - ♦ 2'complenet numbers
  - ♦ Addition, Subtraction and Logical Operations
- Constructing an Arithmetic Logic Unit
- Multiplication and Division
- Floating Point Arithmetic

Chapter 3 of the text

Digital Arithmetic

Arithmetic operations in digital computers are performed on binary numbers.

Main Arithmetic Operations

- Addition, Subtraction, Multiplication, Division

Binary Addition

Operand-1	Operand-2		Sum	Carry
0	0		0	0
0	1		1	0
1	0		1	0
1	1		0	1

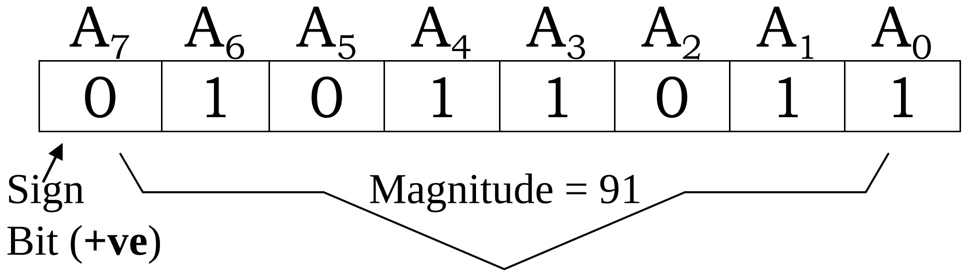
```
    10110          11.101
 +  00111        + 10.011
 ------------------------
```

Sign-Magnitude System

Magnitude and Sign is represented distinctively.

For an eight-bit signed number

- MSB represents sign of the number.
- 7-bits represent magnitude ≤ 127.



A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	1	0	1	1	0	1	1

2's Complement System

Sign-Magnitude based arithmetic is hard to implement in hardware
2's complement system is mostly used for signed binary numbers

1's Complement

2's Complement

(22)	10110
1'complement	

Negation

Converting a +ve number to its -ve equivalent or a -ve number to its +ve equivalent

- ***2's complement*** conversion is only needed for -ve numbers

2's Complement Representation

Positive Number

Add a sign bit ‘0’ in front of the MSB

Negative Number

1. Obtain the binary representation of the magnitude of number
2. Obtain 2's complement of the magnitude
3. Add a sign bit 1 (for -ve number) in front of the MSB of the 2's complement obtained in step 2.

A short cut method

Begin from LSB and move left bit-by-bit towards the MSB.

i) If the bit is 0, simply copy down the bit.

ii) Repeat step i) until the first bit of 1 is encountered, now copy down this bit.

iii) For all subsequent bits, simply invert each one of them.

Example: 2’s complement of – 42

Decimal Values of 2's Complement

Positive Numbers

- MSB is the sign bit = 0
- decimal value equals to binary equivalent

2's complement of 0 10010 = 18

Negative Numbers

- sign bit is 1
- obtain 2’s complement of N-bit magnitude part
- decimal value is equal to –ve of the 2's complement

Decimal value of 1 01110 ?

Sign bit = 1

2's complement of 01110

2's complement number 1 01110 = –18

Special Case

When sign bit = 1 and all other bits equal zero decimal value of the 2's complement = -2^N

e.g. 1 00000

Addition in 2's Complement System

Addition of two positive numbers

+9	0	1 0 0 1
+4	0	0 1 0 0

Addition of +ve and smaller -ve number

+9	0	1 0 0 1
-4	1	1 1 0 0

Addition of a +ve and larger -ve number

-9	1	0 1 1 1
+4	0	0 1 0 0

Addition of two negative numbers

-9	1	0 1 1 1
-4	1	1 1 0 0

Subtraction in
2's Complement System

Subtraction of 2's complement numbers is carried out in the same way as addition

➢ No need of separate hardware for addition and subtraction

- get 2’s complement (negate) of subtrahend
- add it to minuend, result of this addition represent the difference

For example

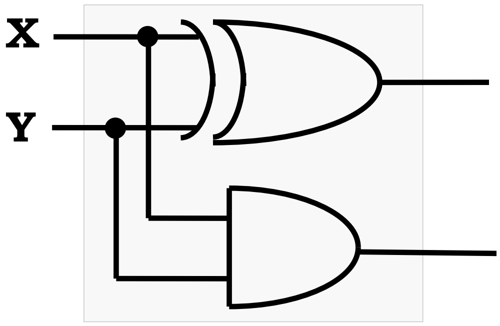
9 – 6 = 9 + (-6)
= 9 + (2’s complement of 6)
=

```
 +9      0   1 0 0 1
 (-6)    1   1 0 1 0
____________________
```

Adder Circuits

Half adder performs addition of 2 bits.

Operand-1, **X**	Operand-2, **Y**		Sum	Carry
0	0		0	0
0	1		1	0
1	0		1	0
1	1		0	1



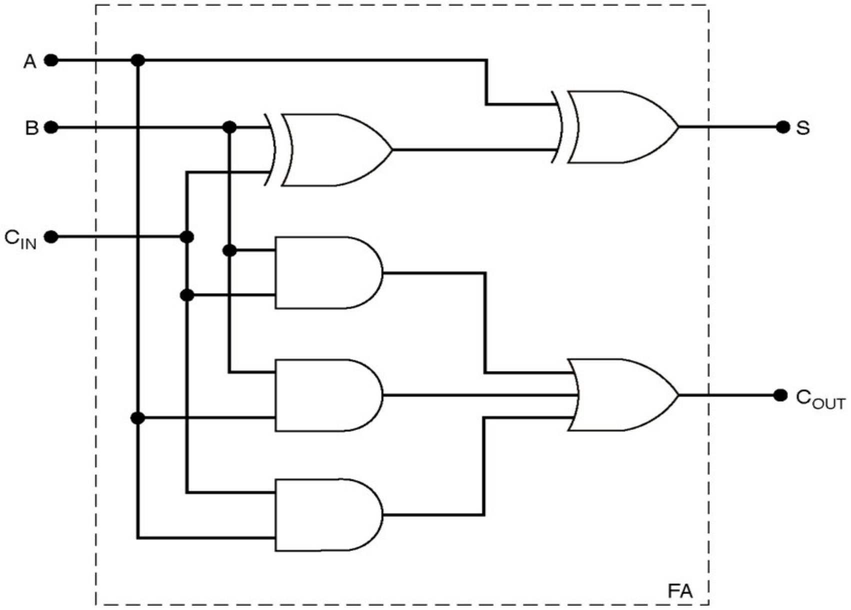
Sum = s = $\overline{x} \cdot y + x \cdot \overline{y}$

Carry = c = $x \cdot y$

VHDL Code for Half Adder

```
library ieee ;
use ieee.std_logic_1164.all ;

entity half_adder is
      port ( x, y : in std_logic ;
                s, c : out std_logic ) ;
end  half_adder ;

architecture dataflow_ha of half_adder is
begin
      s <= x xor y  ;
      c <= x and y ;
end dataflow_ha;
```

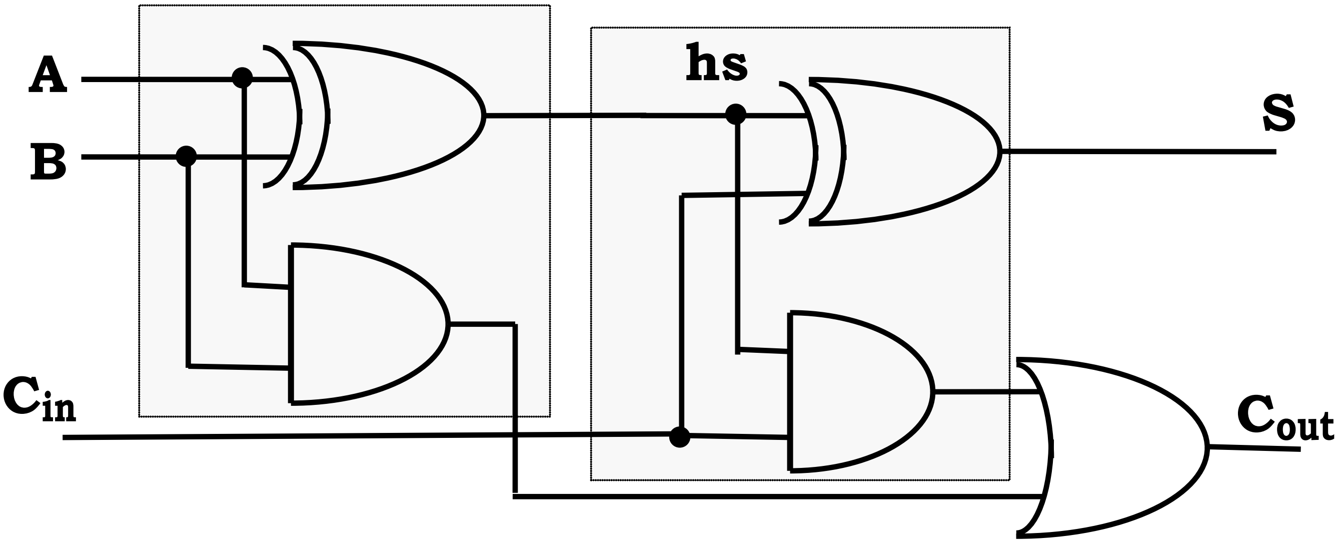
1-Bit Full Adder



Two half adders and one OR gate can also implement a Full Adder.

$S = (A \oplus B) \oplus C_{IN}$

$C_{OUT} = A.B + C_{IN}. (A \oplus B)$



Full Adder

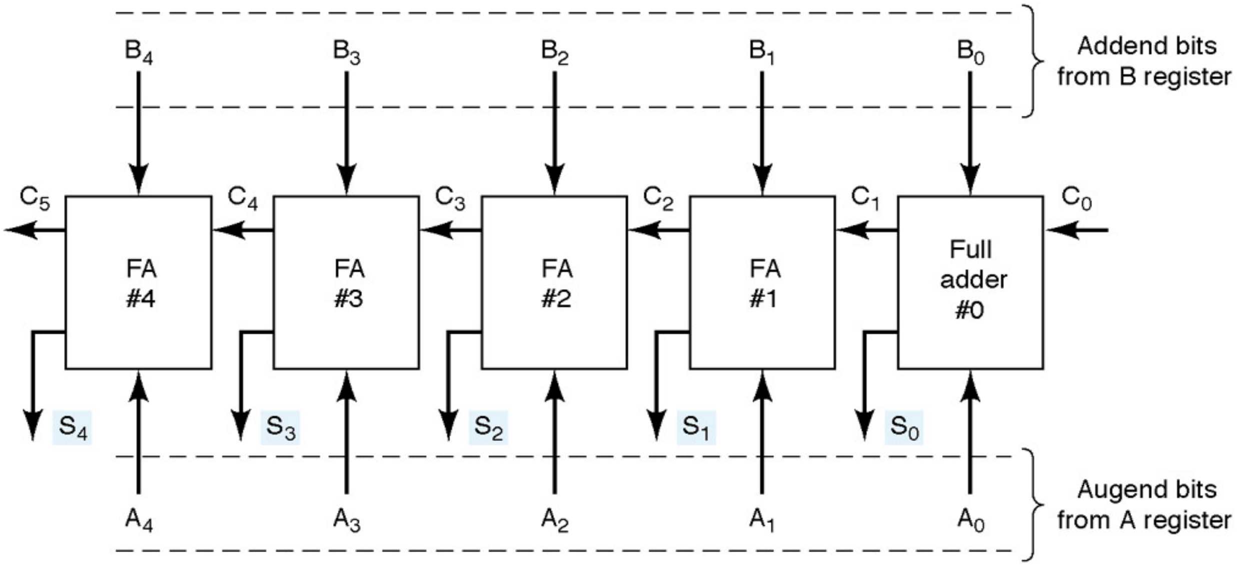
$$S = (A \oplus B) \oplus C_{IN}$$

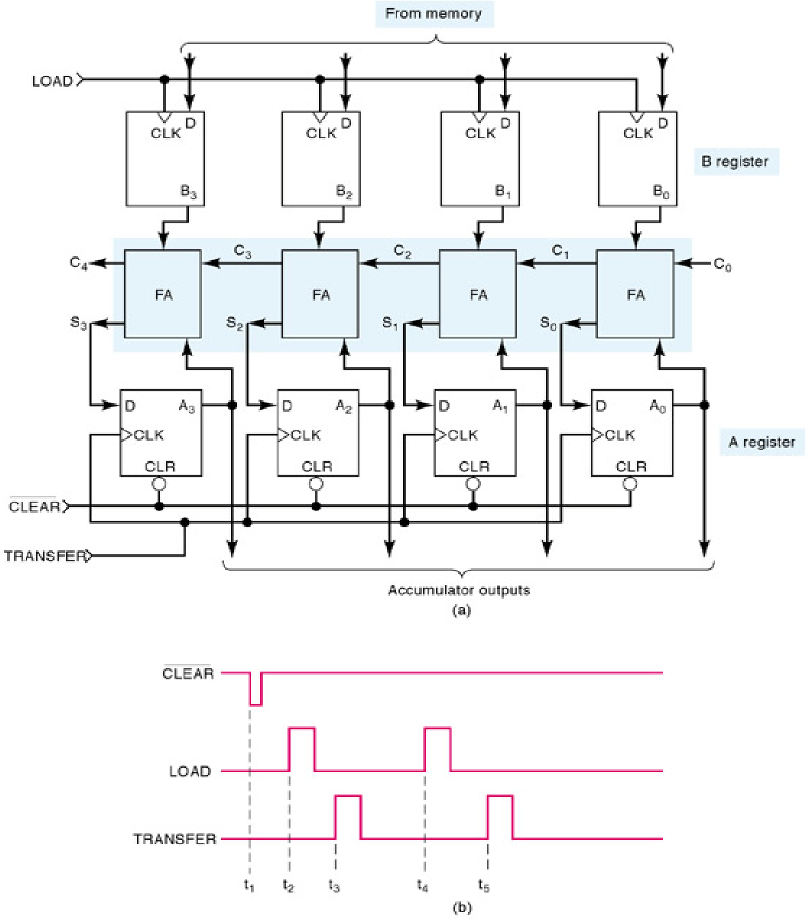
$$C_{OUT} = A \bullet B + C_{IN} \bullet (A \oplus B)$$

VHDL Code of Full Adder

```
library ieee ;
use ieee.std_logic_1164.all ;

entity full_adder is
      port ( a, b, cin : in std_logic ;
               s, cout : out std_logic ) ;
end  full_adder ;

architecture dataflow_fa of full_adder is

      component half_adder is
             port ( x, y : in std_logic ;
                      s, c : out std_logic ) ;
      end  component ;
      signal hs, hc, tc : std_logic

begin
      HA1: half_adder
                    port map(a, b, hs, hc) ;
      HA2: half_adder
                    port map(hs, cin, s, tc) ;
      cout <= tc or hc ;
end dataflow_fa ;
```

Multi-bit Adder





(a)

(b)

Speed limited by carry chain

4-bit Adder VHDL Code

```
library ieee ;
use ieee.std_logic_1164.all ;
entity adder_4 is
      port  ( A, B : in std_logic_vector (3 downto 0) ;
             C0 : in std_logic ;
             S : out std_logic_vector (3 downto 0) ;
             C4 : out std_logic ) ;
end  adder_4 ;
architecture dataflow_add4 of adder_4 is
    component full_adder is
        port ( a, b, cin : in std_logic ;
              s, cout : out std_logic ) ;
    end  component ;
    signal C : std_logic_vector (4 downto 0) ;
begin
    C(0) <= C0
    BIT0: full_adder
             port map(B(0), A(0), C(0), S(0), C(1)) ;
    BIT1: full_adder
             port map(B(1), A(1), C(1), S(1), C(2)) ;
    BIT2: full_adder
             port map(B(2), A(2), C(2), S(2), C(3)) ;
    BIT3: full_adder
             port map(B(3), A(3), C(3), S(3), C(4)) ;
    C4 <= C(4) ;
end dataflow_add4 ;
```

Ripple Addition

Typical Ripple Carry Addition is a Serial Process:

- Addition starts by adding LSBs of the augend and addend.
- Then next position bits of augend and addend are added along with the carry (if any) from the preceding bit.
- This process is repeated until the addition of MSBs is completed.

Carry Propagation

- Speed of a ripple adder is limited due to carry propagation or carry ripple.
- Sum of MSB depends on the carry generated by LSB.

Carry Lookahead Adder

Faster Adders Limit the Carry Chain

- 2-level AND-OR logic.

  2^n product terms
- 3 or 4 levels of logic, carry look-ahead

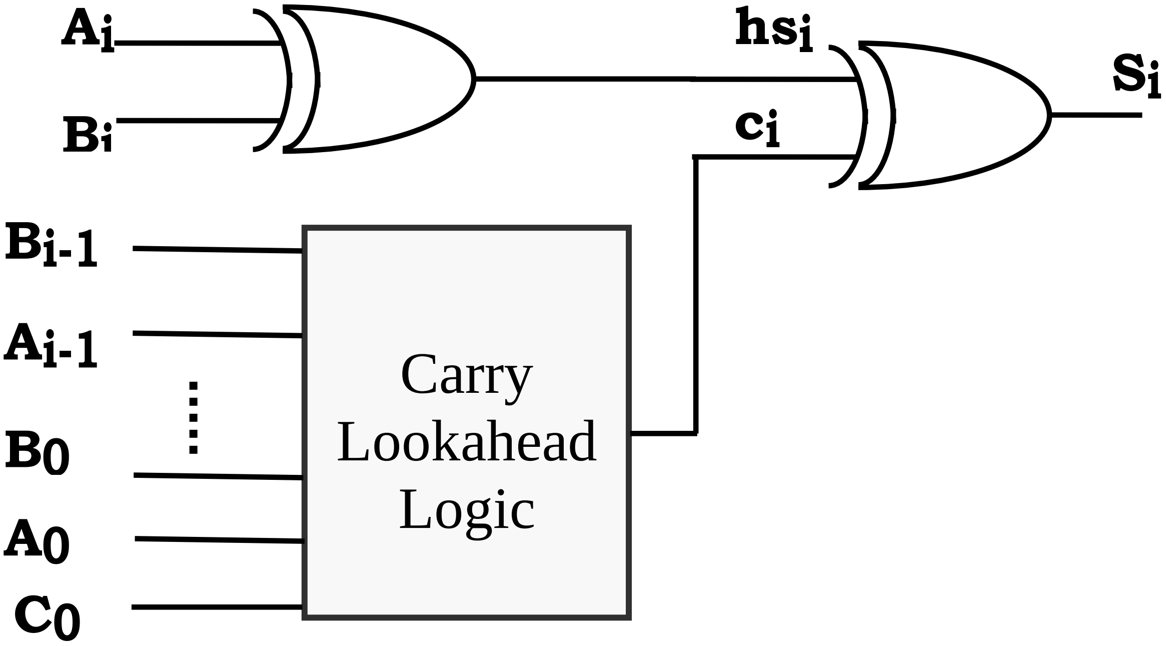
A Carry look-ahead adder avoids carry propagation delay by using additional logic circuit.

Looks at the lower order bits of operands and determine if a higher-order carry is to be generated

The Sum bit $S_i = A_i \oplus B_i \oplus C_i$

$$C_{i+1} = g_i + p_i C_i$$

Carry Lookahead Adder



$C_1 = g_0 + p_0C_0$

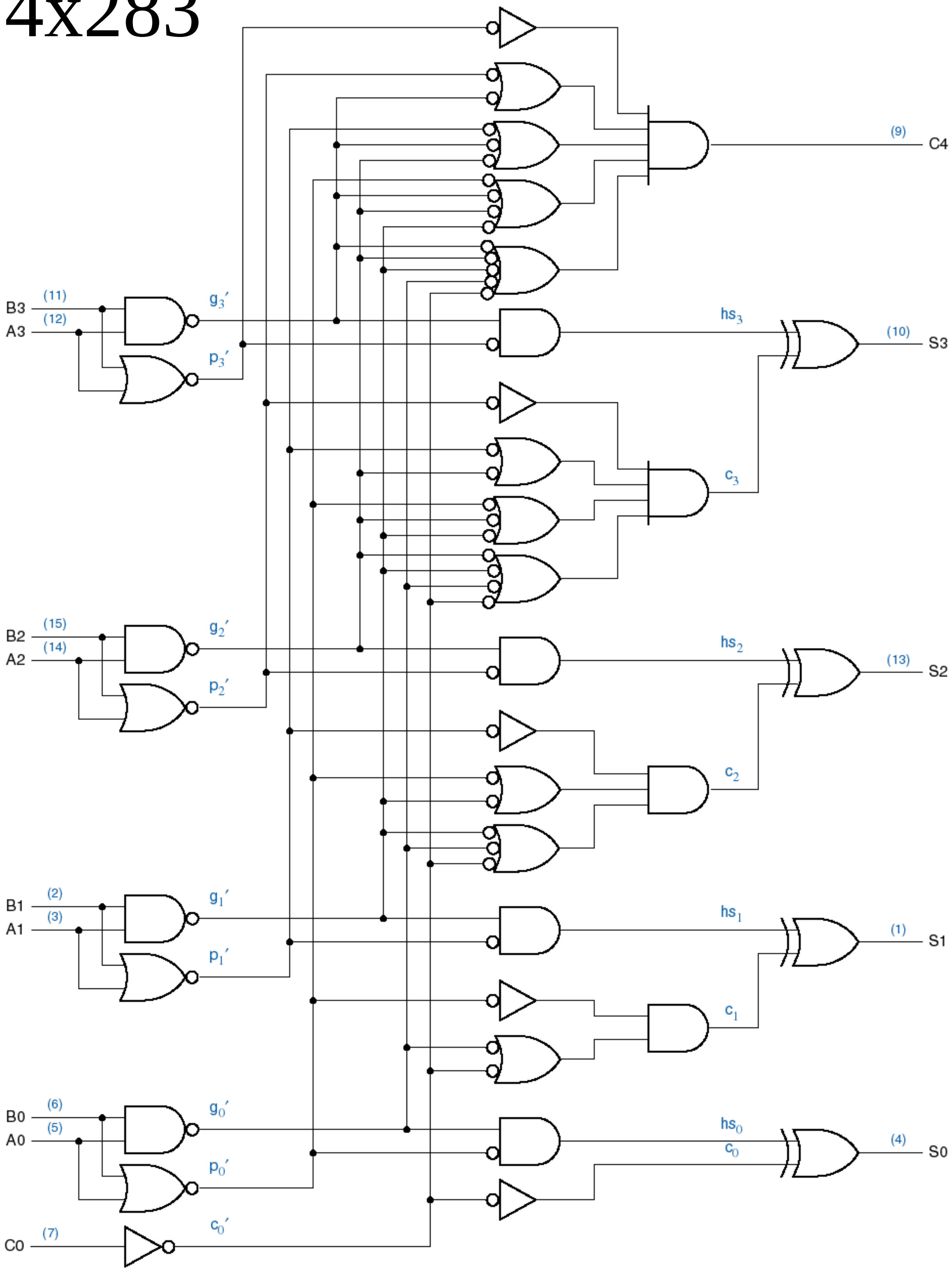
$C_2 = g_1 + p_1C_1 = g_1 + p_1g_0 + p_1p_0C_0$

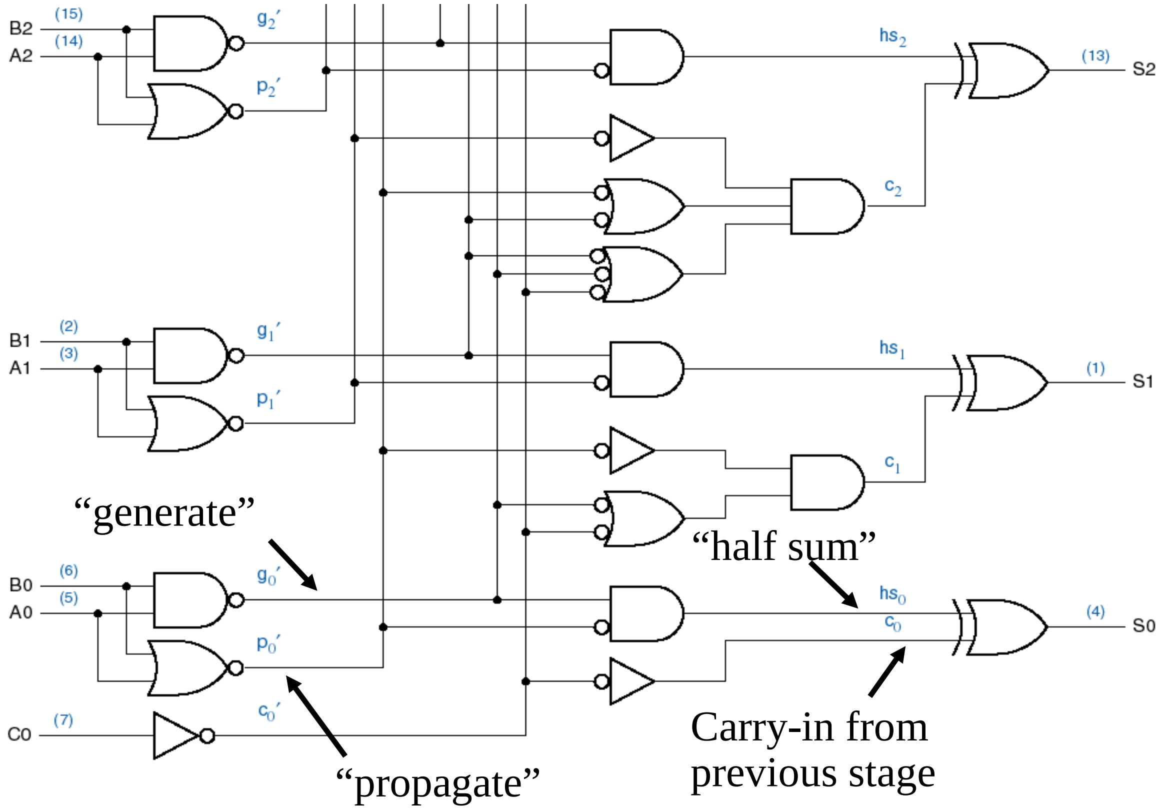
$C_3 = g_2 + p_2C_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0C_0$

$C_4 = C_{OUT} = g_3 + p_3g_2 + p_3p_2g_1 + p_2p_1g_0 + p_2p_1p_0C_0$

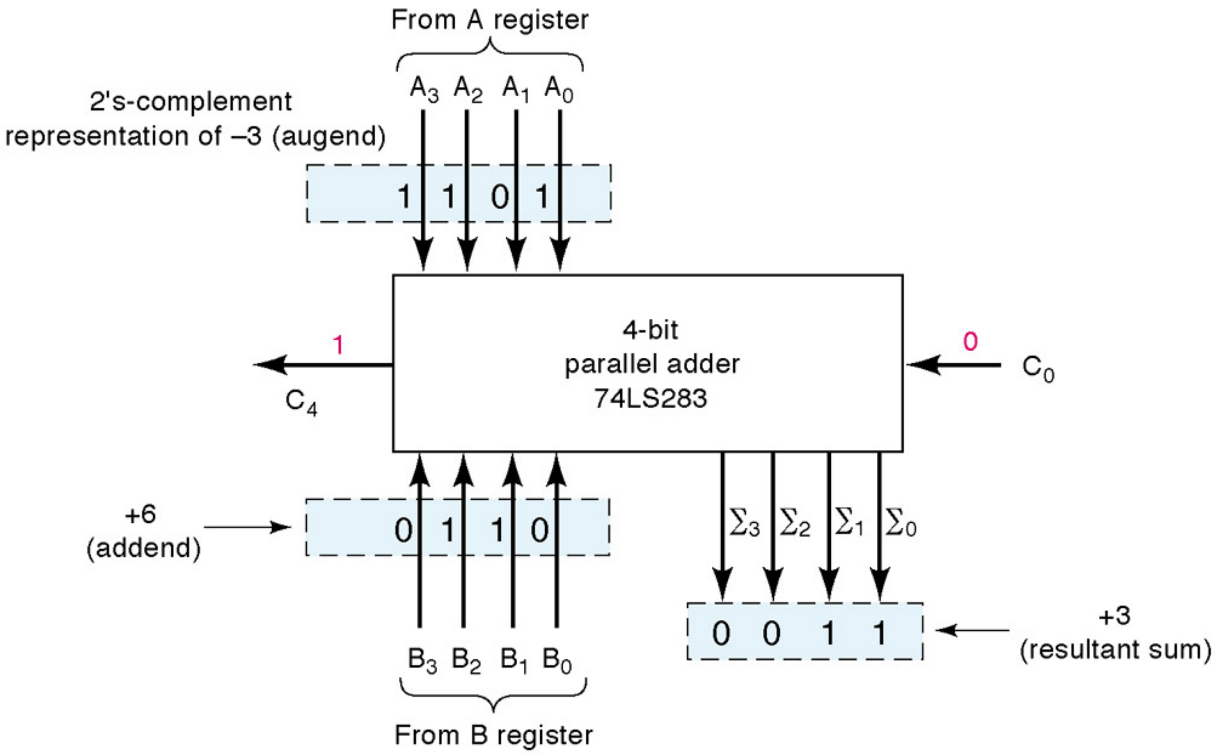
4-bit Carry Look-ahead Adder

74x283





Addition of +ve/-ve numbers in 2's-complement



Subtraction

Negative numbers can be added by first converting them to 2’s complement form.

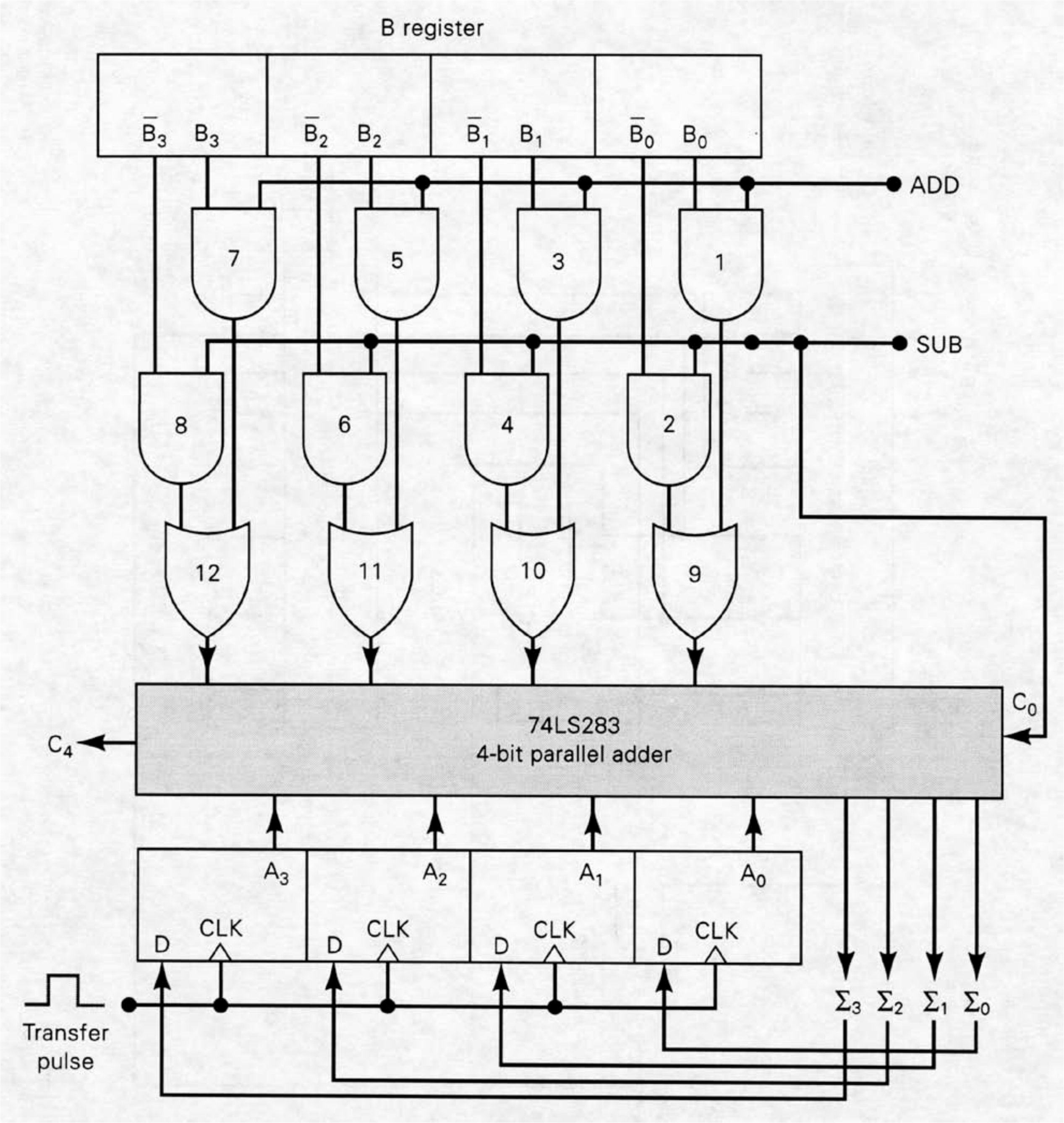
Subtraction is the same as addition of the two’s complement numbers.

- The two’s complement is a bit-by-bit complement plus 1.
- Therefore: X – Y = X + Y' + 1

If the result is negative then get 2's complement of the result.

Adder circuit can be modified to perform both addition and subtraction in a 2’s complement system

2's Complement Addition and Subtraction



Adder-Subtractor

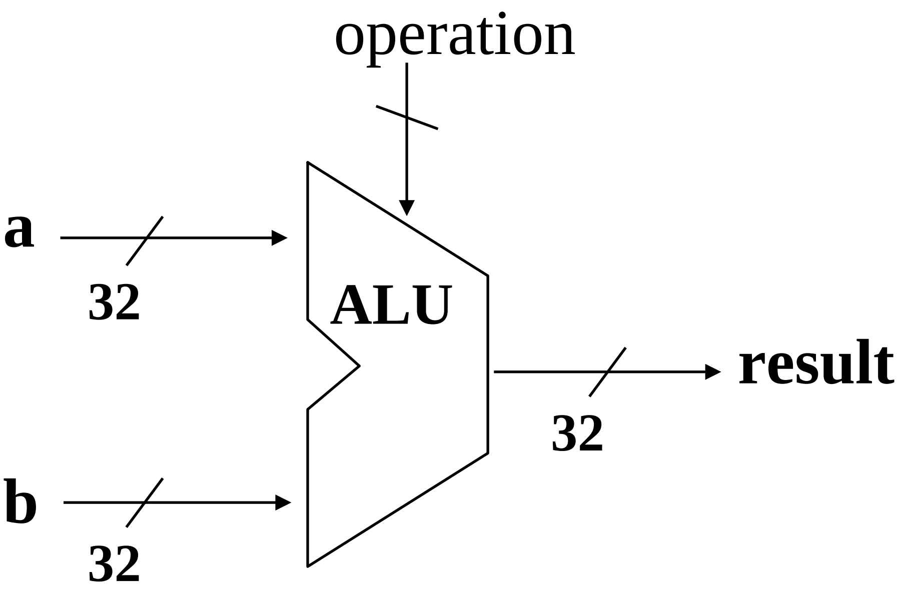
2's Complement System Addition

Straight forward

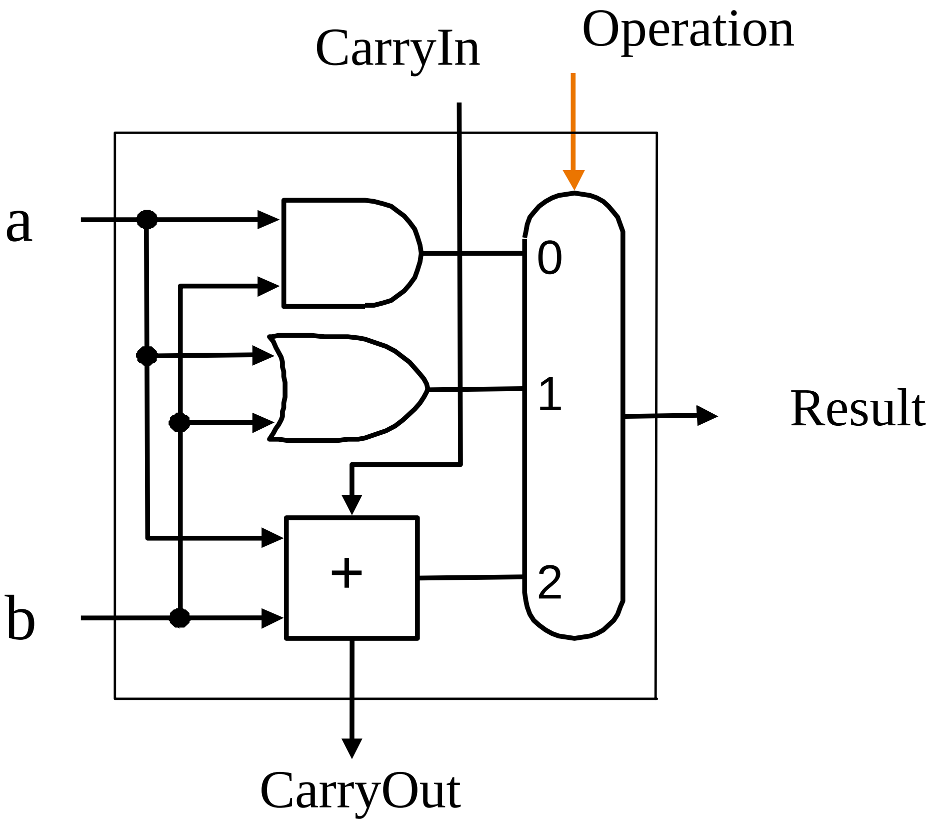
2's Complement System Subtraction

- Minuend and Subtrahend are in registers A and B respectively
- SUB = 1 enables AND gates 2, 4, 6, 8
- ADD = 0 disables AND gates 1, 3, 5, 7
- It connects the complement of subtrahend to port B of LS283
- C_0 = 1 produces 2's complement of subtrahend during addition
- Transfer pulse adds minuend and 2's complement of subtrahend (i.e. equivalent to subtraction

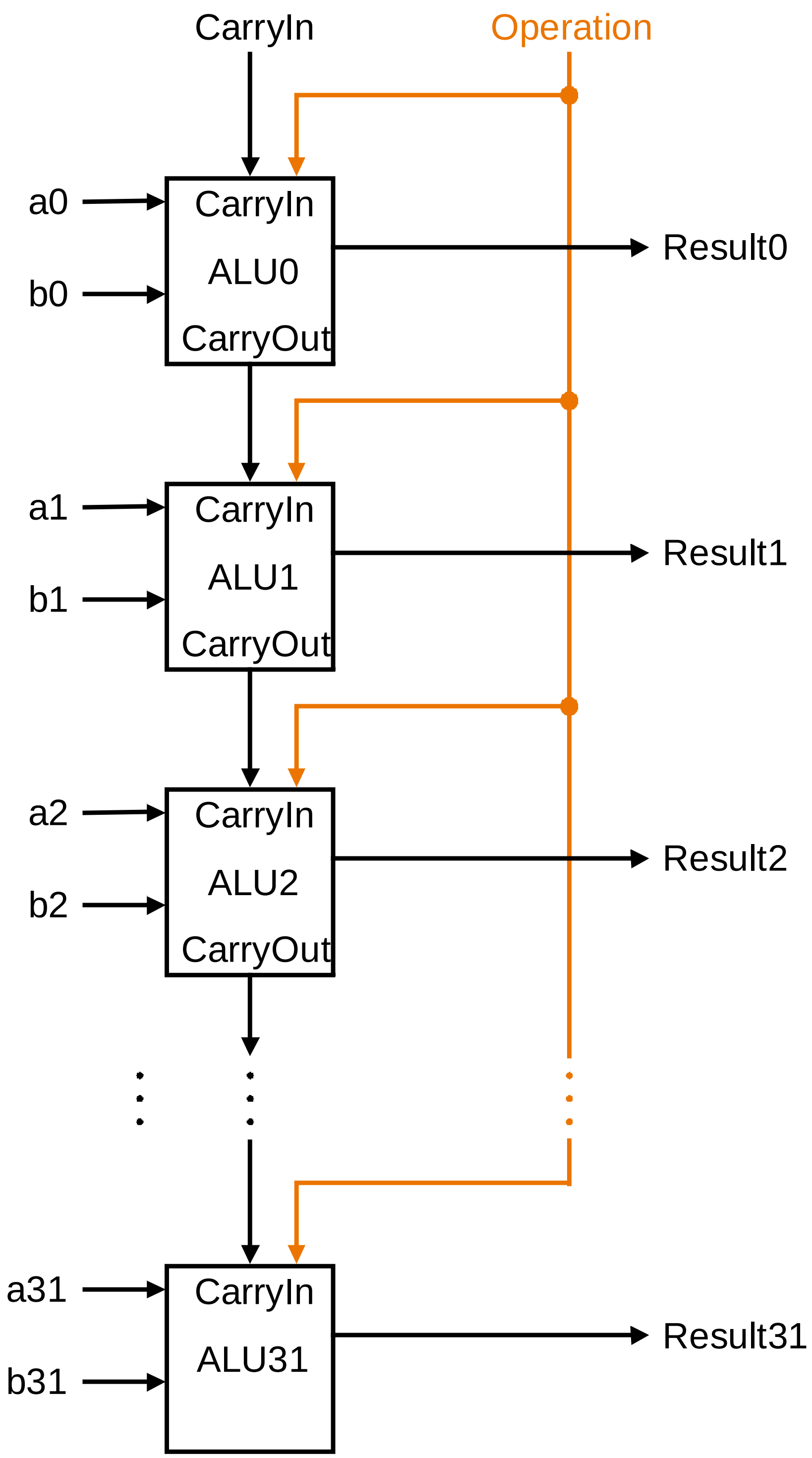
ALU Architecture



1-bit ALU that performs AND, OR and Addition

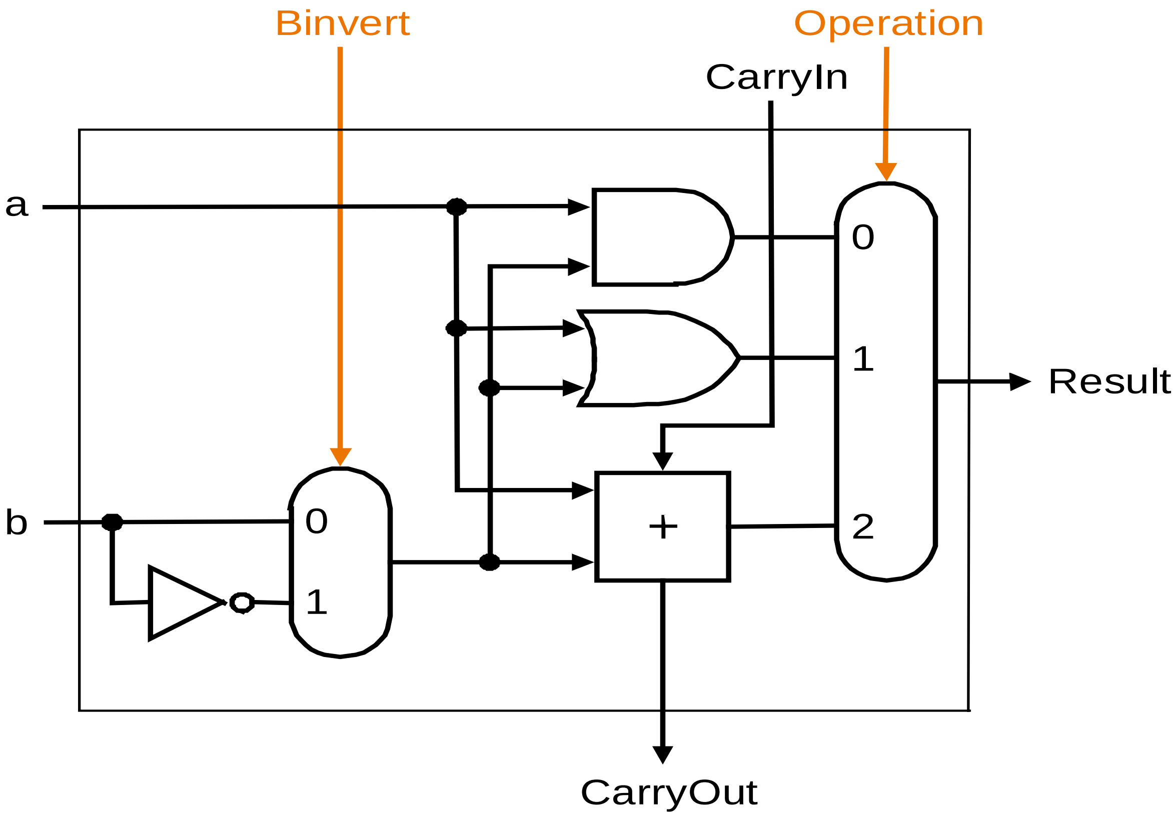


32-bit ALU


CarryIn
Operation
a0
b0
CarryIn
ALU0
CarryOut
Result0
a1
b1
CarryIn
ALU1
CarryOut
Result1
a2
b2
CarryIn
ALU2
CarryOut
Result2
a31
b31
CarryIn
ALU31
Result31

ALU for MIPS-Processor

Subtraction Option: $a - b = a + \overline{b} + 1$



Support the set-on less-than instruction (**slt**)

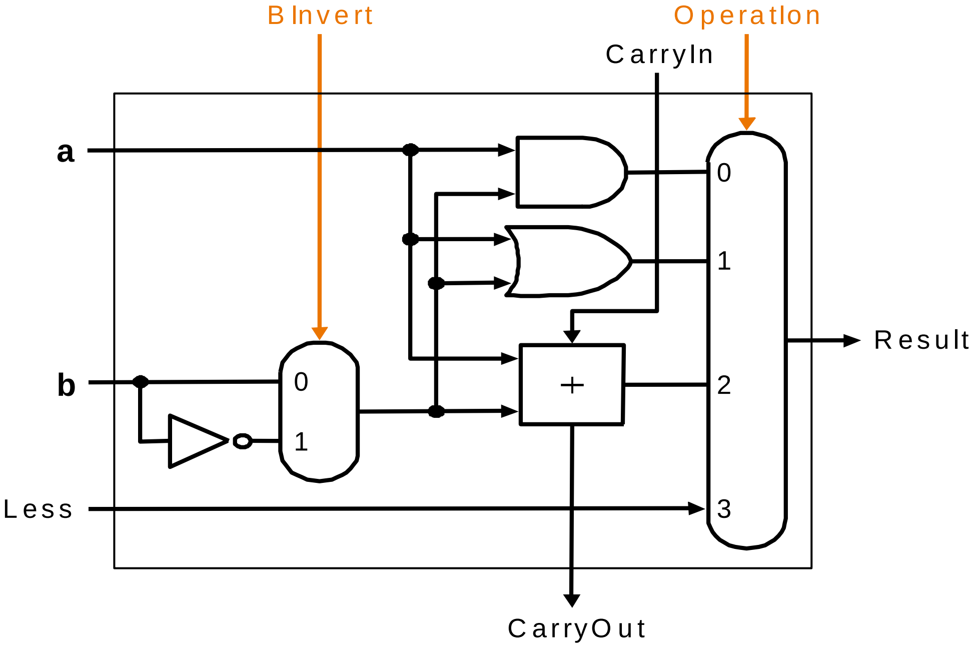
- **slt** is an arithmetic instruction.

  slt $t5, $t6, $t7

Need to test for equality (**beq** $t6, $t7, Label)

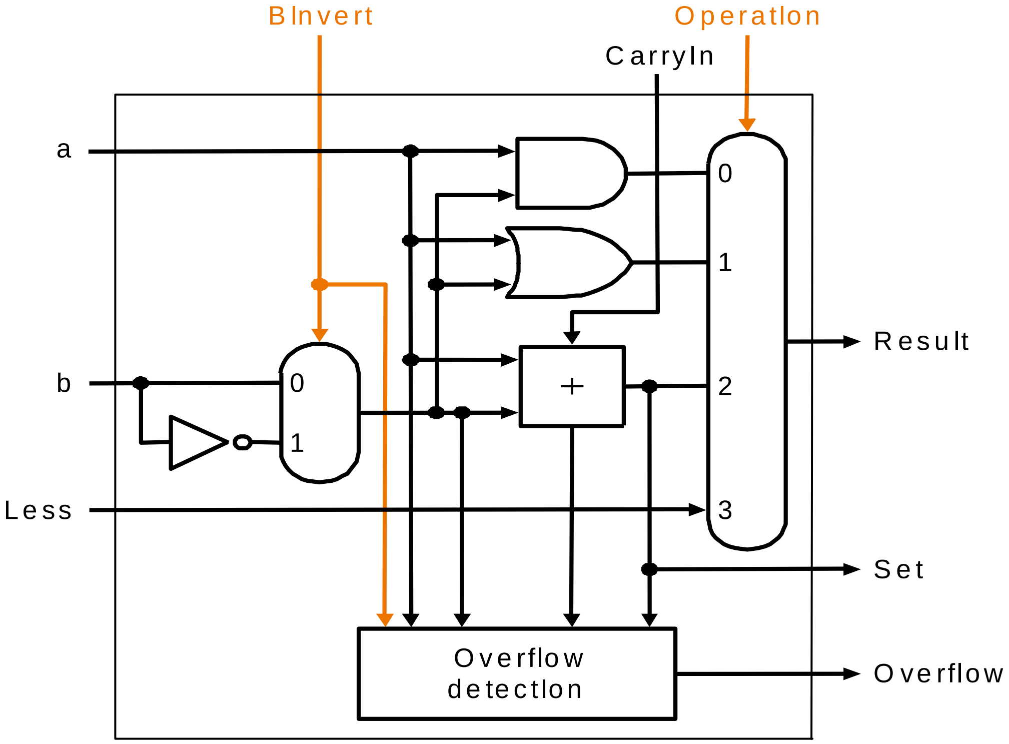
Use subtraction: (a - b) = 0 implies a = = b

Supporting the **slt** Instruction



Overflow Detection

Set Overflow output to 1 when a < b



Detecting Overflow

No Overflow

- By adding a positive (+ve) and a negative (-ve) number.
- subtracting numbers of the same signs.

Overflow occurs when value affects the sign.

- By adding two positives yields a negative result.
- Adding two negatives gives a positive.
- Subtract a -ve from a +ve and get a negative.
- Subtract a +ve from a -ve and get a positive.

Consider the operations A + B and A – B

- Can overflow occur if B is 0 ?
- Can overflow occur if A is 0 ?

Effects of Overflow

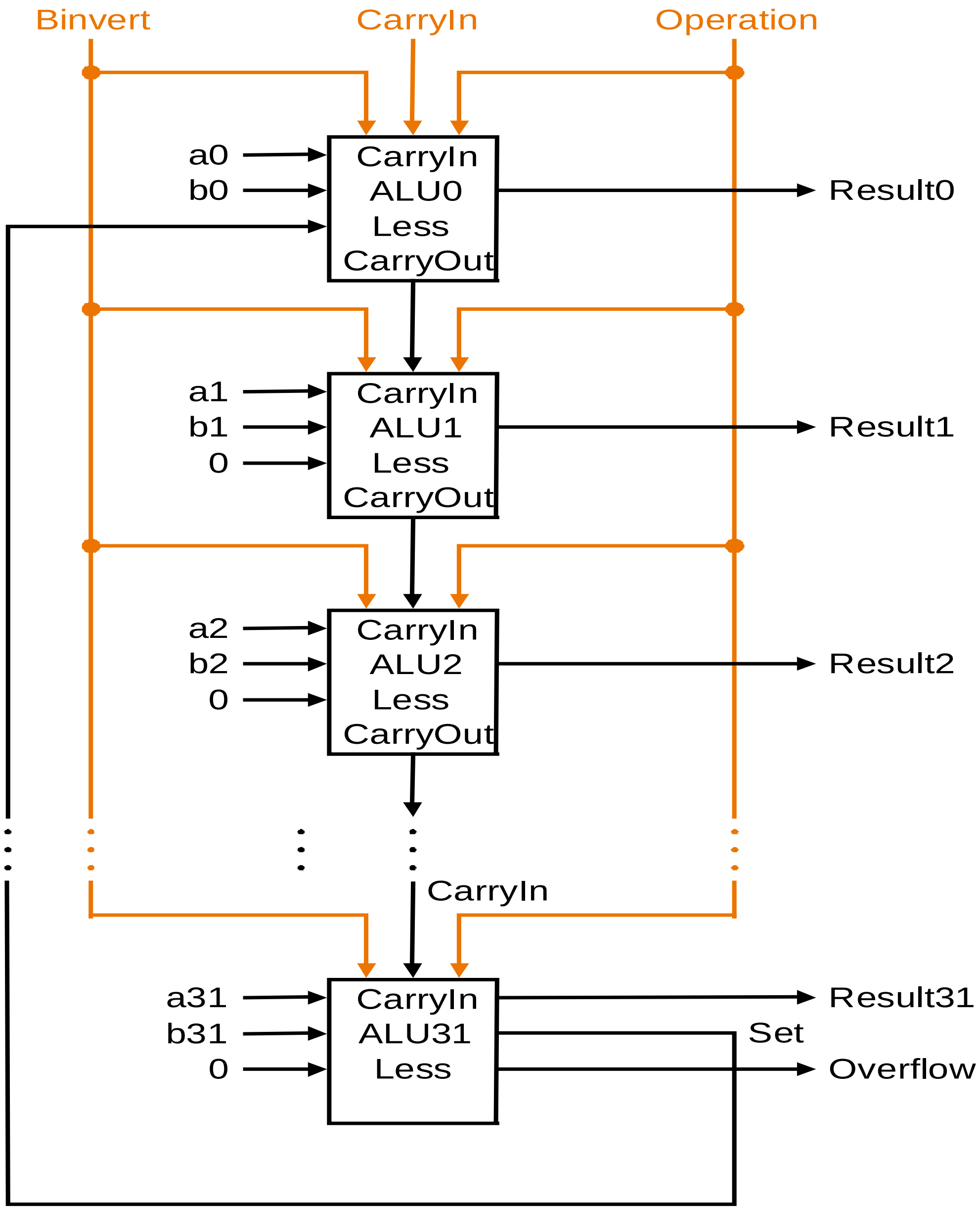
- An exception (interrupt) occurs.
  - Control jumps to predefined address for exception.
  - Interrupted address is saved for resumption.

If there is no need to detect overflow

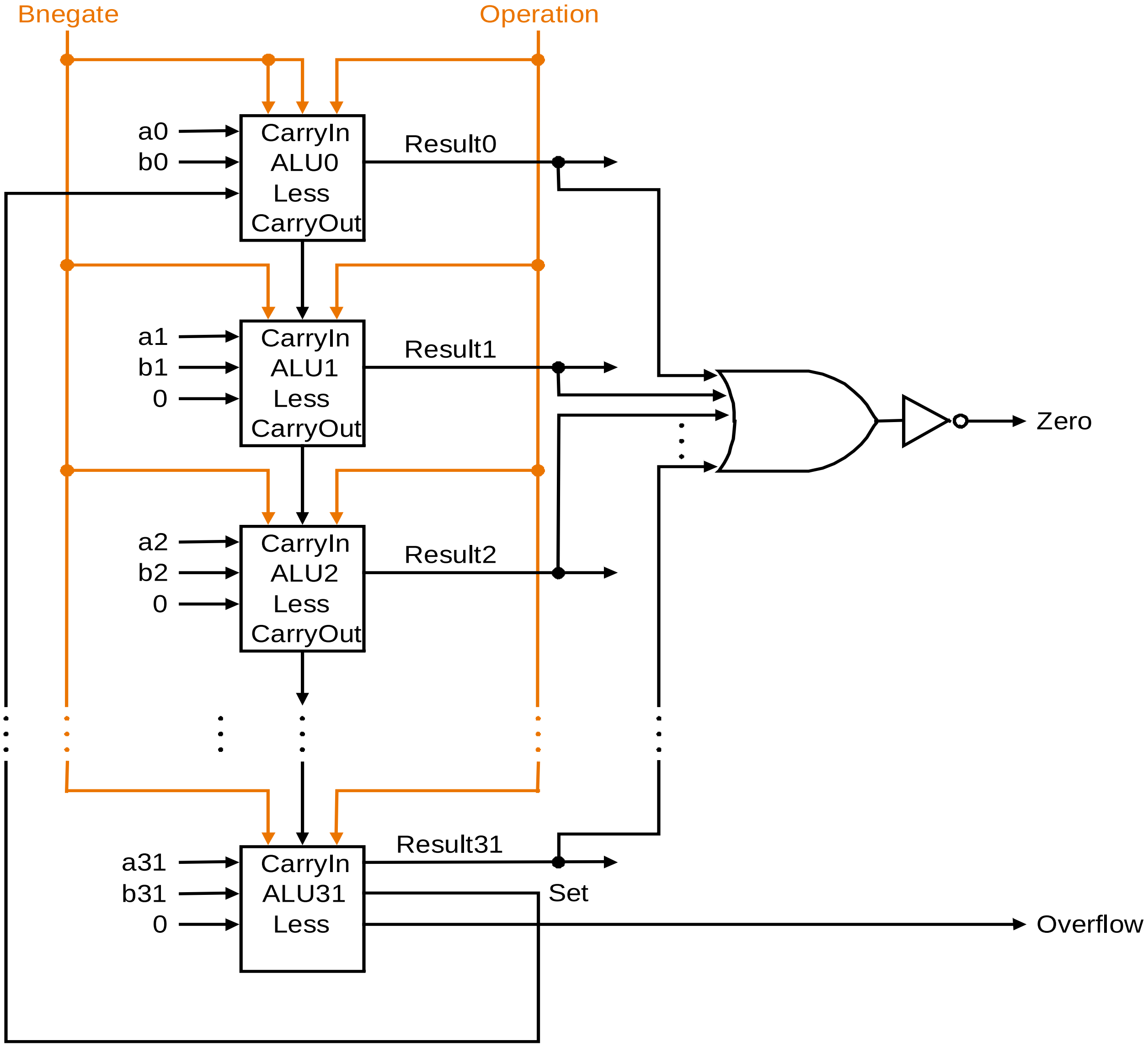
- new MIPS-CPU instructions: addu, addiu, subu

32-bit ALU

Overflow and **slt** Supported



Test for Equality



Notice the control lines

000 = and
001 = or
010 = add
110 = subtract
111 = slt

Multiplication

Binary Multiplier

Product of two 4-bit numbers is an 8-bit number

```
multiplicand          1101   (13)
multiplier          * 1011   (11)
                    ___________




                 __________
                 10001111    (143)
```

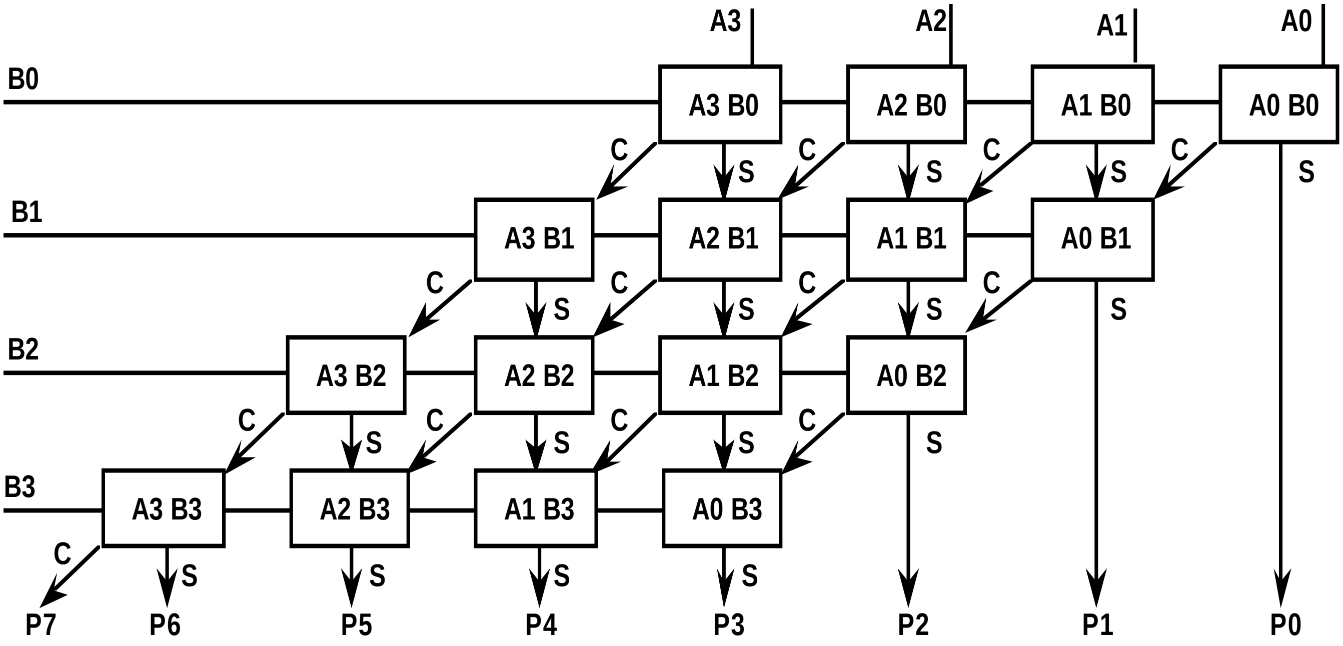
4-bit Multiplication

				A3	A2	A1	A0
				B3	B2	B1	B0
				A3B0	A2B0	A1B0	A0B0
			A3B1	A2B1	A1B1	A0B1	
		A3B2	A2B2	A1B2	A0 B2		
	A3B3	A2 B3	A1 B3	A0 B3			
S7	S6	S5	S4	S3	S2	S1	S0

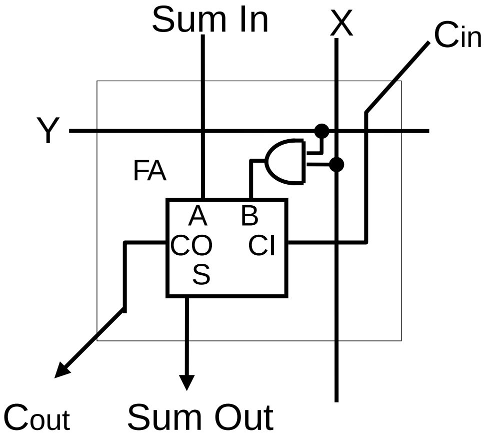
4-Bit Multiplier Circuit

Unsigned Combinational Multiplier

4 X 4 Array of Building Blocks (Cells)

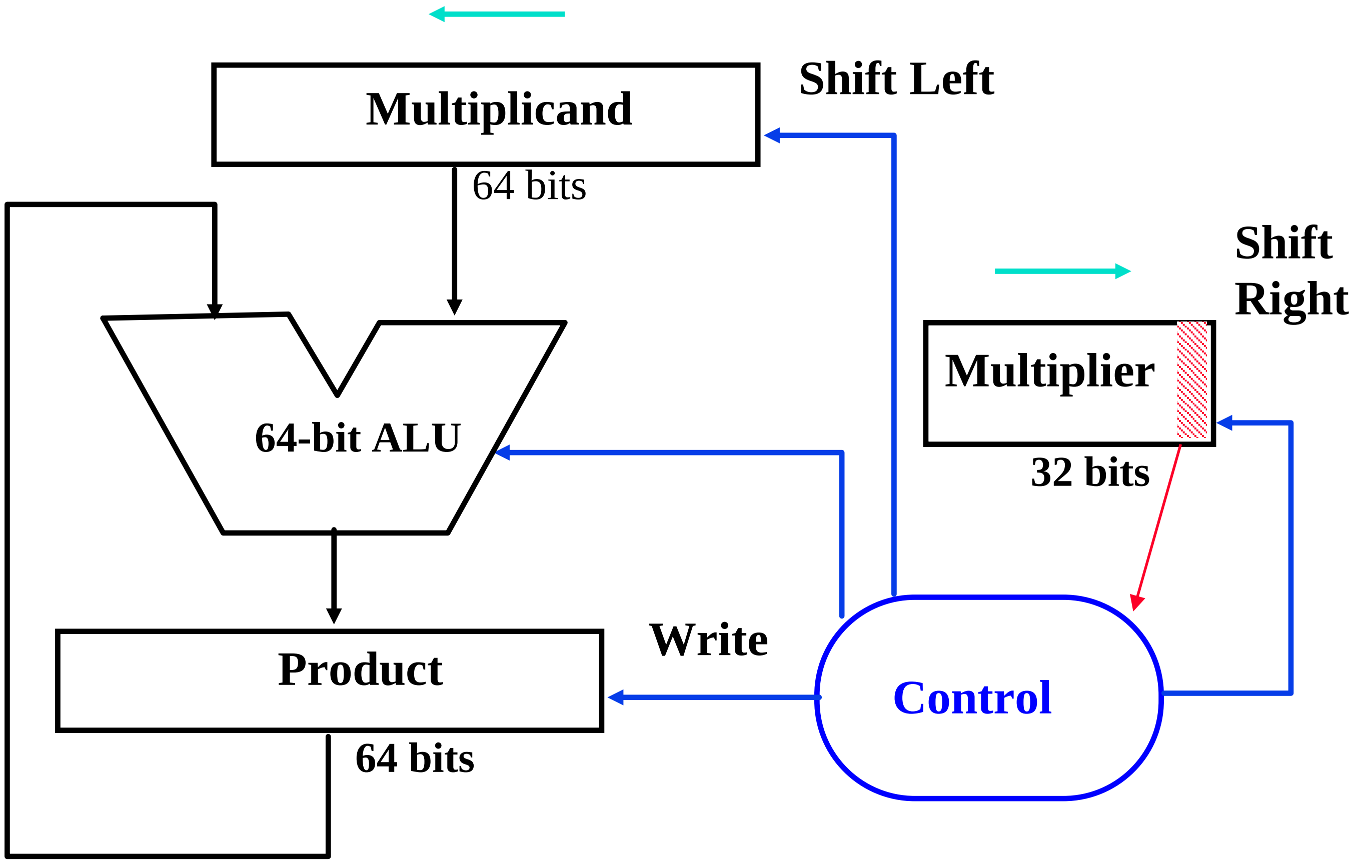


Building Block/Cell



Unsigned Shift-Add Multiplier

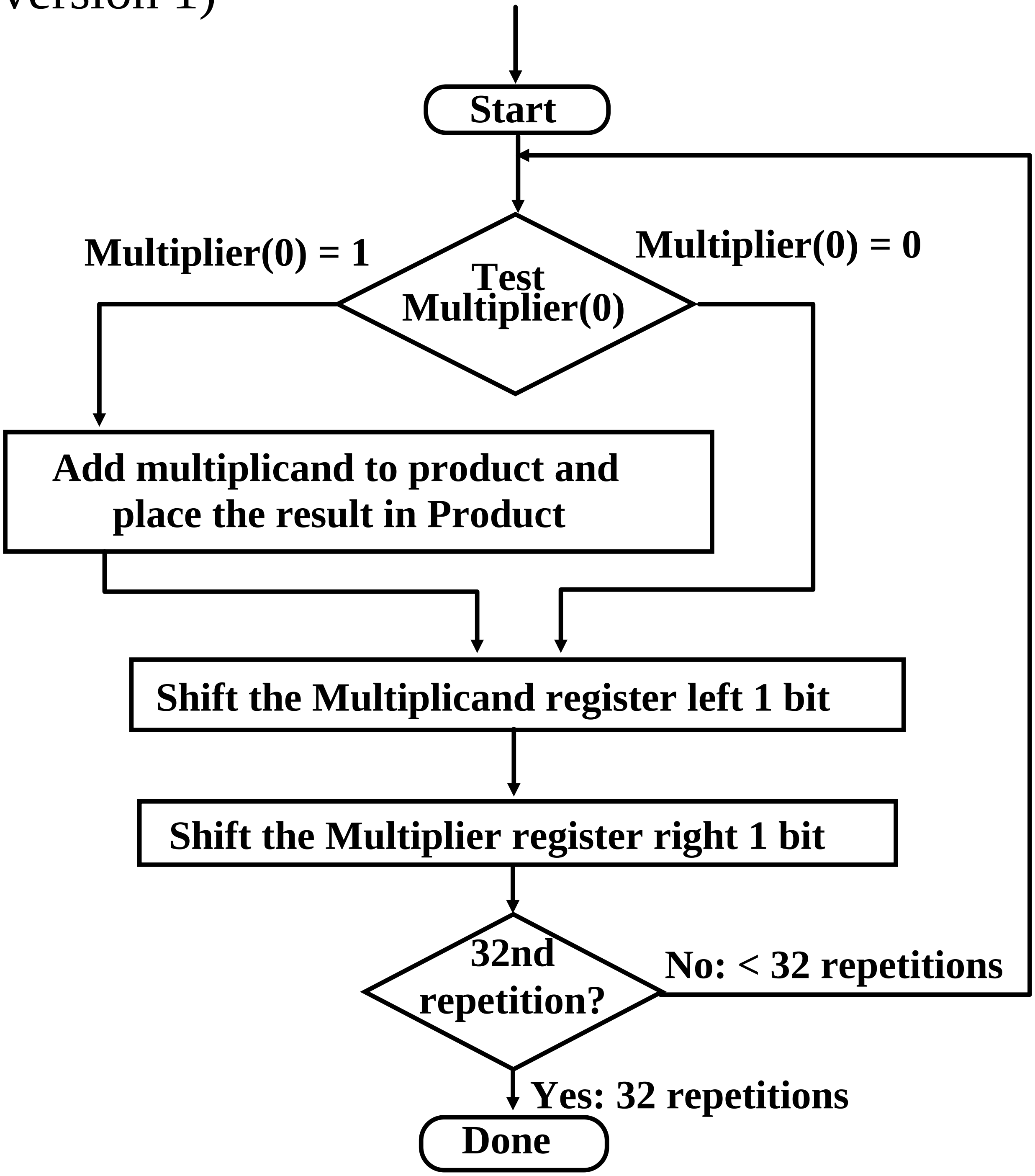
(version 1)



Multiplier = datapath + control

Unsigned Shift-Add Multiplier

(version 1)



Start
Multiplier(0) = 1
Test Multiplier(0)
Multiplier(0) = 0
Add multiplicand to product and place the result in Product
Shift the Multiplicand register left 1 bit
Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Unsigned Shift-Add Multiplier

Product	**Multiplier**	**Multiplicand**
0000 0000	0011	0000 0010

1 clock per cycle => 100 clocks/multiply.

Ratio of multiply to add 5:1 to 100:1

1/2 bits in multiplicand always 0.

=> 64-bit adder is wasted.

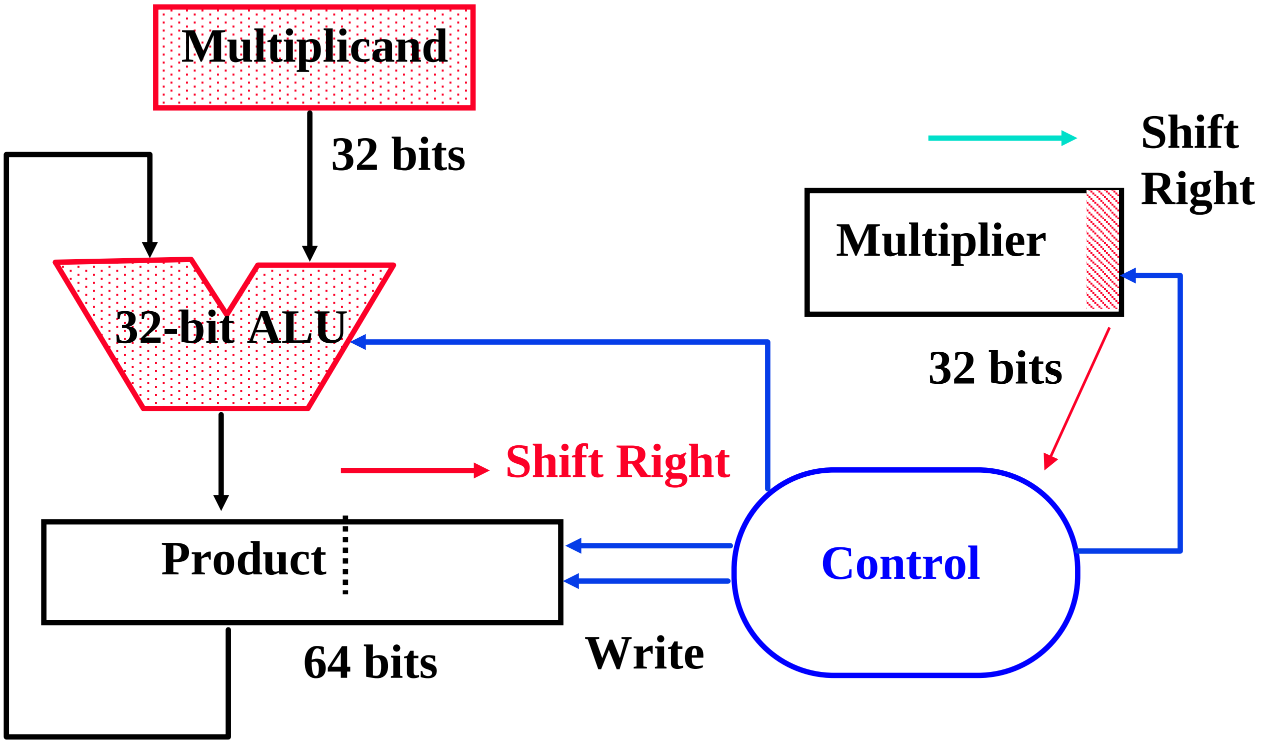
0 bits are inserted in left of the multiplicand as shifted.

=> least significant bits of product never changed once formed

Multiplier

(version 2)

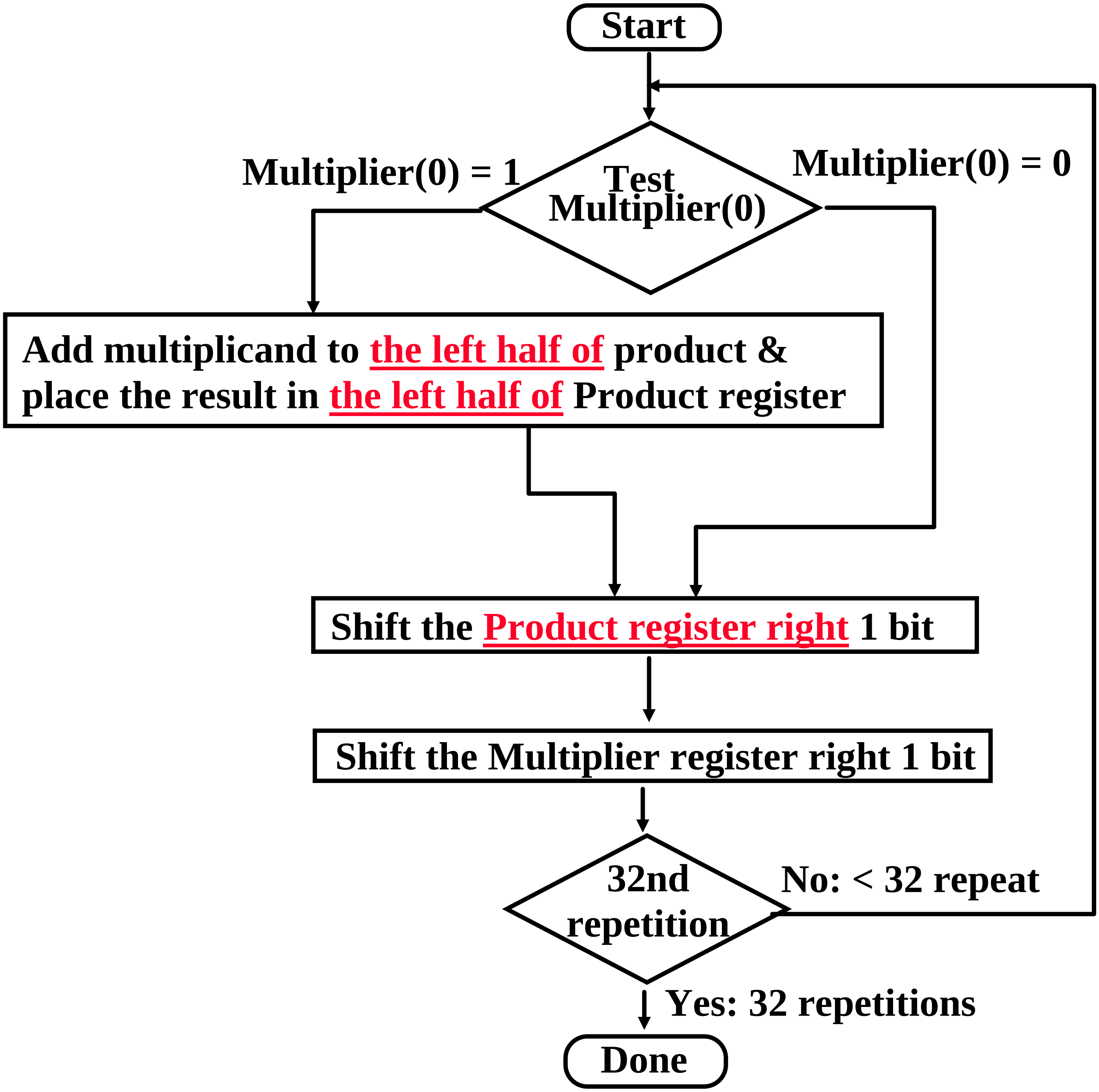
32-bit Multiplicand register
32-bit ALU
64-bit Product register
32-bit Multiplier register



Multiplier Algorithm

(version 2)

Multiplier	Multiplicand	Product
0011	0010	0000 0000



Start

Test Multiplier(0)

Multiplier(0) = 1

Multiplier(0) = 0

Add multiplicand to the left half of product & place the result in the left half of Product register

Shift the Product register right 1 bit

Shift the Multiplier register right 1 bit

32nd repetition

No: < 32 repeat

Yes: 32 repetitions

Done

Multiplication Process

Product	Multiplier	Multiplicand
0000 0000	0011	0010

Product register wastes space that exactly matches with the size of multiplier

=> combine Multiplier register and Product register.

Integer Multiplication in MIPS

In MIPS, we multiply registers:
32-bit value x 32-bit value = 64-bit value
Syntax of Multiplication (signed):

mult *register1*, *register2*

- Multiplies 32-bit values in those registers & puts 64-bit product in special registers hi & lo (separate from the 32 general purpose registers)
- Use mfhi move from hi and move from mflo.

Example in C: a = b * c;Let b be $s2; let c be $s3; and let a be $s0 and $s1 (since it may be up to 64 bits)

```
mult $s2, $s3      # b*c
mfhi $s0       # upper half of product into $s0
mflo $s1       # lower half of
               # product into $s1
```

Integer Division

```
                 1001              Quotient
Divisor 1000|1001010            Dividend
            -1000
                 10
                 101
                 1010
                 -1000
                      10   Remainder
                      (or Modulo result)Dividend
```

= Quotient x Divisor + Remainder

MIPS Division

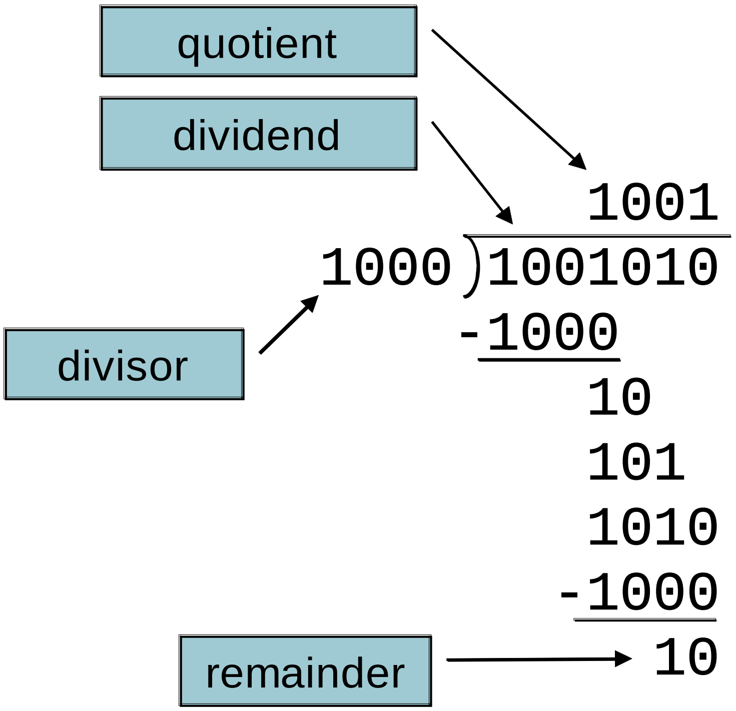
div *register1*, *register2*

Divides 32-bit register-1 by 32-bit register-2 and puts remainder of division in ***hi***, quotient in ***lo***Implements C division (/) and modulo (%)

Example in C: a = c / d; b = c % d;In MIPS:

```
a«$s0;b«$s1;c«$s2;d«$s3 div  $s2,$s3     # lo=c/d, hi=c%d
     mflo $s0          # get quotient
     mfhi $s1          # get remainder
```

Division



```
              1001
1000 )1001010
        -1000
            10
            101
            1010
           -1000
              10
```

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

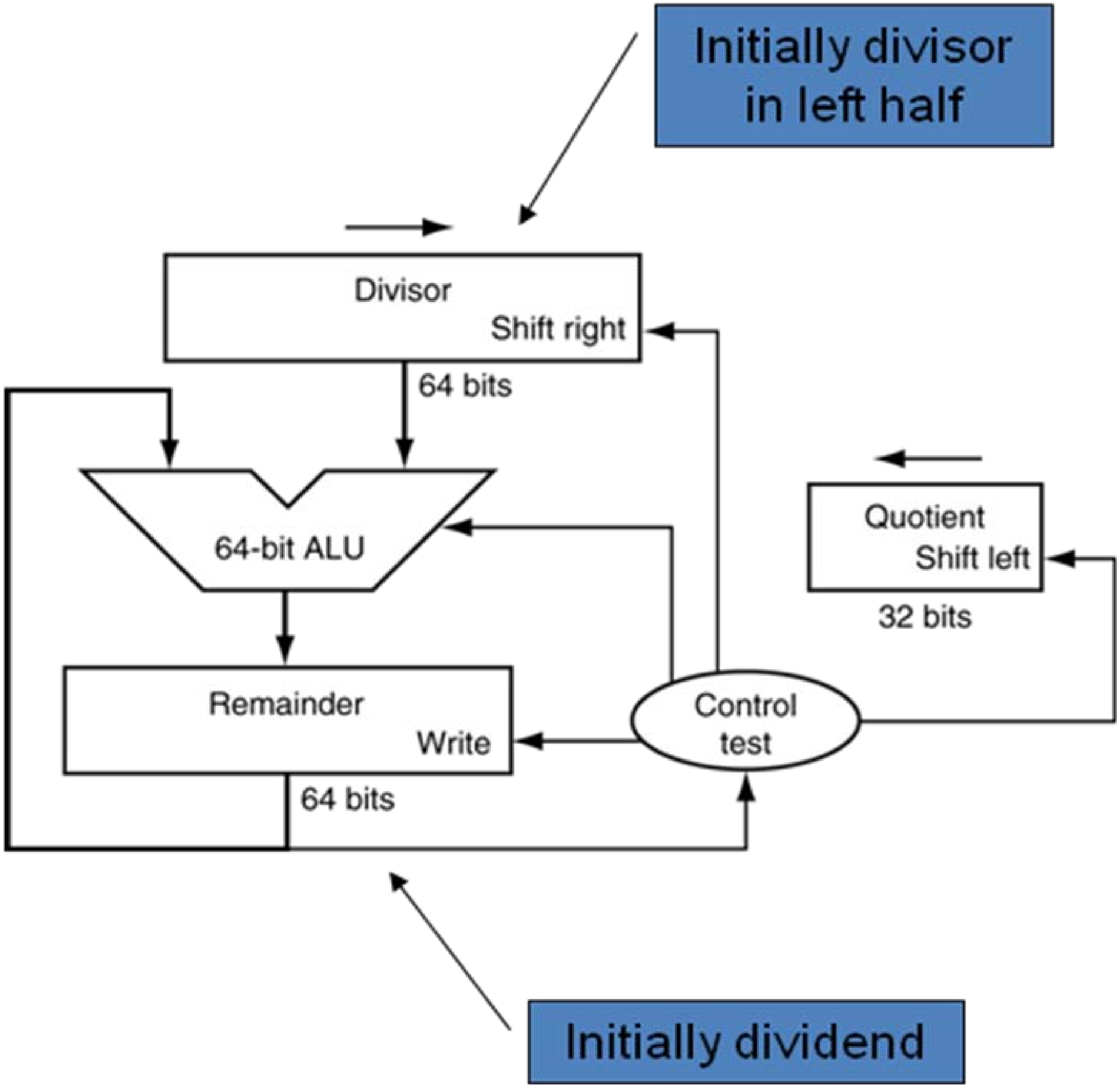
Check for 0 divisor
Long division approach
If divisor ≤ dividend bits
- * 1 bit in quotient, subtract
- * Otherwise 0 bit in quotient, bring down next dividend bit

Restoring division
- * Subtract, and if remainder is < 0, then add divisor back

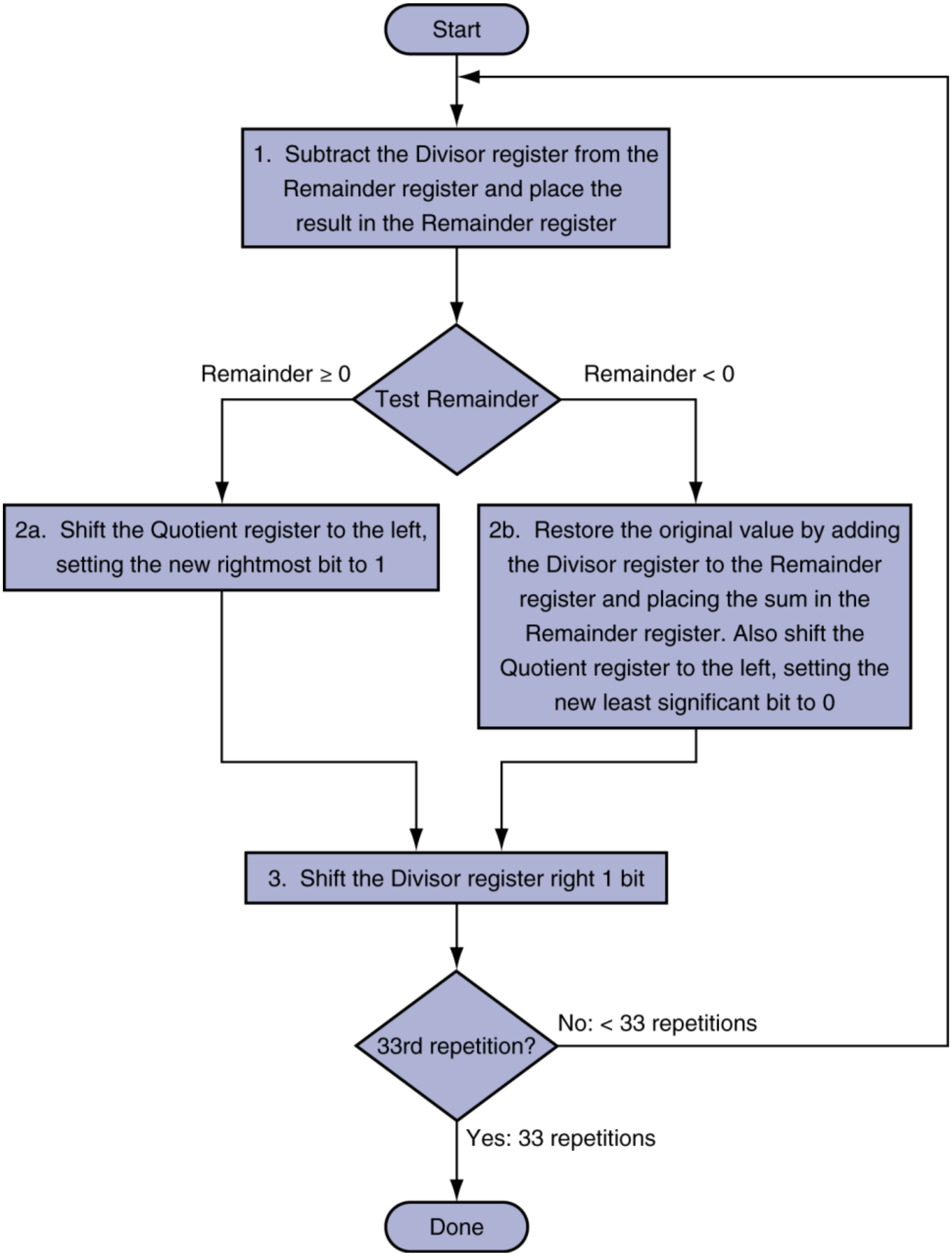
Signed division

- * Divide using absolute values
- * Adjust sign of quotient and remainder as required

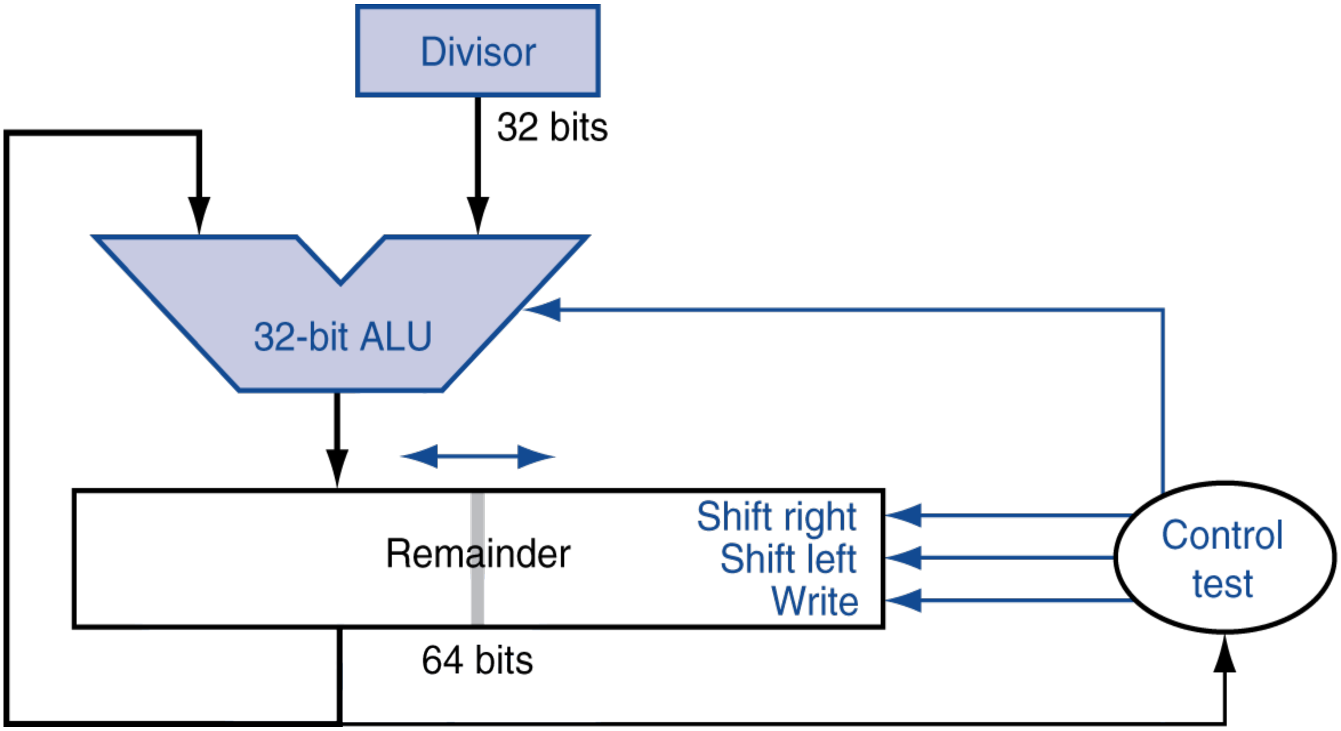
Divisor Hardware



Division Algorithm


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
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
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

Optimal Division Hardware



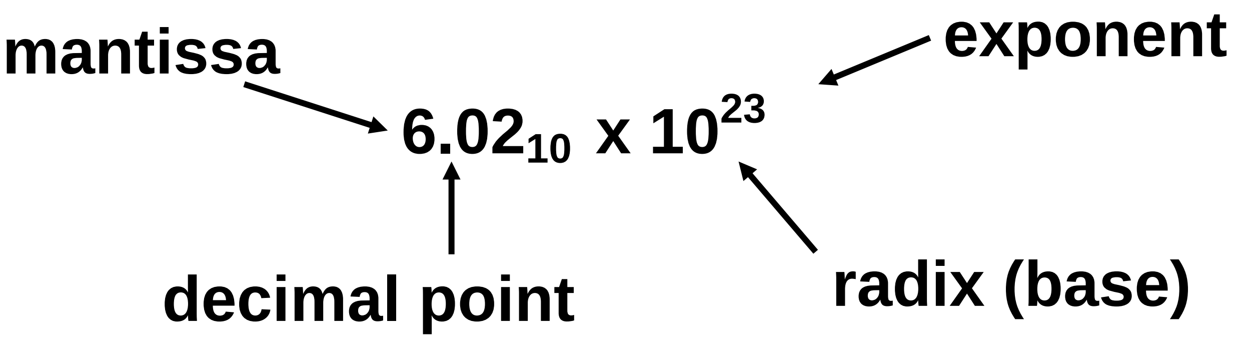
One cycle per partial-remainder subtraction

Looks a lot like a multiplier!

Same hardware can be used for both multiplication and division.

Floating Point Arithmetic

Scientific Notation (in Decimal)



mantissa

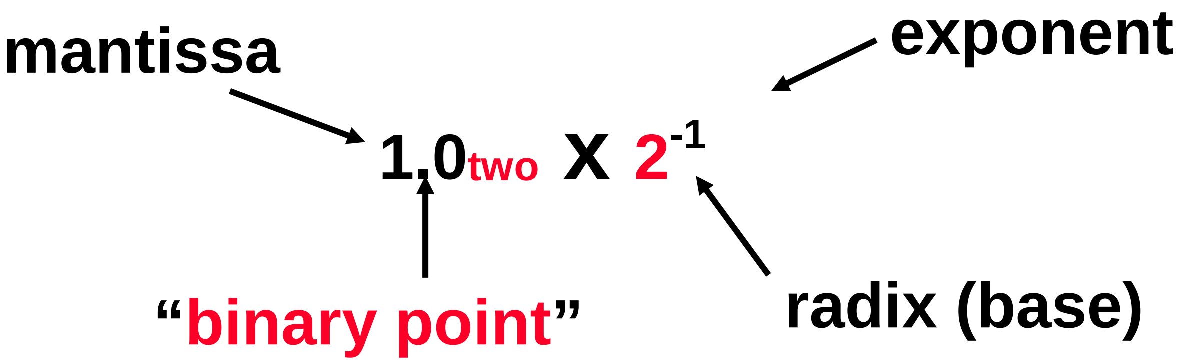
exponent

6.02_{10} x 10^{23}

decimal point

radix (base)

Normalized form: no leadings 0s
(exactly one digit to left of decimal point)
Alternatives to representing 1/1,000,000,000
Normalized: 1.0 x 10^{-9}
Not normalized: 0.1 x 10^{-8}, 10.0 x 10^{-10}



mantissa

exponent

1.0_{two} X 2^{-1}

“binary point”

radix (base)

Computer arithmetic that supports it called floating point, because it represents numbers where binary point is not fixed, as it is for integers

- Declare such variable in C as float

Floating Point Arithmetic

- Floating Point numbers approximate values that we want to use.
- IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers
- Every desktop or server computer sold since ~1997 follows these conventions

Single Precision Floating Point

Normal Format: $+1.xxxxxxxxxx_{two} \times 2^{yyyy}{}_{two}$

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

$$(-1)^{S} \times (1 + \text{Significand}) \times 2^{\textbf{(Exponent-127)}}$$

S represents Sign, Exponent represents y's

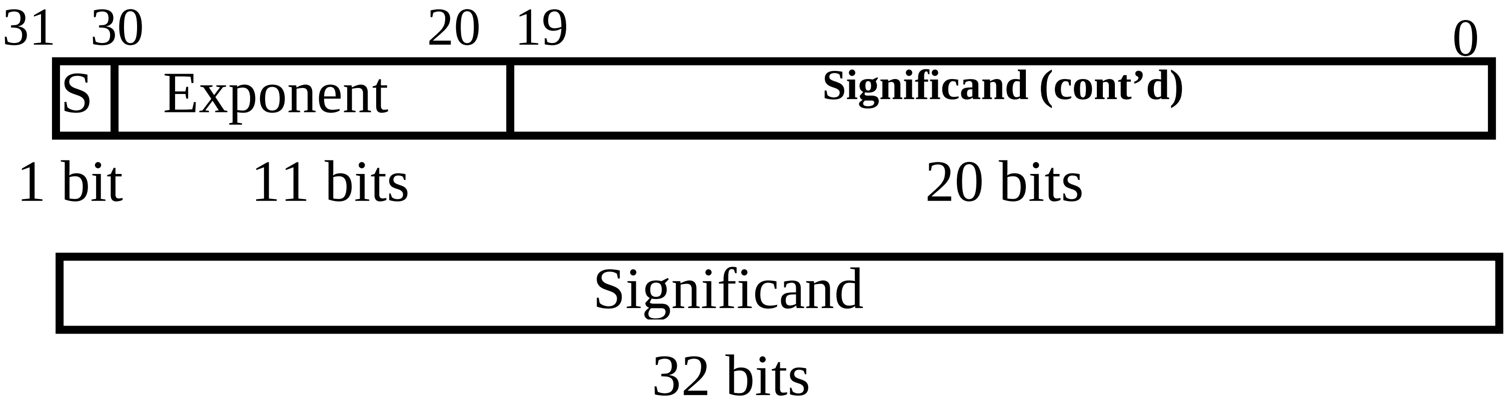
Significand represents x's

Represent numbers as small as 2.0×10^{-38}

to as large as 2.0×10^{38}

Double Precision Floating Point Representation

Next Multiple of Word Size (64 bits)



31 30 20 19 0

S Exponent Significand (cont’d)

1 bit 11 bits 20 bits

Significand

32 bits

Double Precision (vs. Single Precision)
C variable declared as doubleRepresent numbers almost as small as
2.0 x 10^{-308}

to almost as large as 2.0 x 10^{308}

But primary advantage is greater accuracy due to larger significand

IEEE 754 Floating Point Standard

Single Precision, Double Precision similar

Sign bit: 1 means -ve and 0 means +ve

Significand: To pack more bits, leading 1 implicit for normalized numbers

1 + 23 bits single, 1 + 52 bits double

Always true: Significand < 1

(for normalized numbers)

Called Biased Notation, where bias is number to be subtracted to get the real number

IEEE 754 uses bias of 127 for single precision.

Subtract 127 from Exponent field to get actual value for exponent.

$$(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent-127})}$$

1023 is bias for double precision

Understanding FP Numbers

Ist Method (Fractions):

In decimal: $0.340_{10} \Rightarrow 340_{10}/1000_{10}$

$\Rightarrow 34_{10}/100_{10}$

In binary: $0.110_2 \Rightarrow 110_2/1000_2 = 6_{10}/8_{10}$

$\Rightarrow 11_2/100_2 = 3_{10}/4_{10}$

Advantage: Less purely numerical, more thought oriented; this method usually helps people understand the meaning of the significand better.

2nd Method (Place Values):

Convert from scientific notation

In decimal: $1.6732 = (1 \text{ x } 10^{0}) + (6 \text{ x } 10^{-1}) + (7 \text{ x } 10^{-2}) + (3 \text{ x } 10^{-3}) + (2 \text{ x } 10^{-4})$

In binary: $1.1001 = (1 \text{ x } 2^{0}) + (1 \text{ x } 2^{-1}) + (0 \text{ x } 2^{-2}) + (0 \text{ x } 2^{-3}) + (1 \text{ x } 2^{-4})$

Interpretation of value in each position extends beyond the decimal/binary point

Advantage: Good for quickly calculating the significand value; use this method for translating FP numbers

Example

Converting Binary FP to Decimal

0	**0110 1000**	**101 0101 0100 0011 0100 0010**

Sign: 0 => positive

Exponent:
$0110\ 1000_{two} = 104_{ten}$

Bias adjustment: 104 - 127 = -23

Significand:
$1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$

$=1+2^{-1}+2^{-3} +2^{-5} +2^{-7} +2^{-9} +2^{-14} +2^{-15} +2^{-17} +2^{-22}$

$= 1.0_{ten} + 0.666115_{ten}$

Represents: 1.666115_{ten} x 2^{-23} ~ 1.986 x 10^{-7}
(about 2/10,000,000)

Converting Decimal to FP

Simple Case: If denominator is exponent of 2 (2, 4, 8, 16, etc.), then it's easy.

Show MIPS representation of -0.75

$= -3/4$

$-11_{two}/100_{two} = -0.11_{two}$

Normalized to $-1.1_{two} \times 2^{-1}$ $(-1)^S \times$

$(1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

$(-1)^1 \times (1 + .100\ 0000\ ...\ 0000) \times 2^{(126-127)}$

1	**0111 1110**	**100 0000 0000 0000 0000 0000**

Converting Decimal to FP

Not So Simple Case:
If denominator is not an exponent of 2.

Then we can not represent number precisely, but that's why we have so many bits in significand: for precision

Once we have significand, normalizing a number to get the exponent is easy.
So how do we get the significand of a never-ending number?

- *All rational numbers have a repeating pattern when written out in decimal.*
- *This also applies in binary.*

To finish conversion:

- Write out binary number with repeating pattern.
- Cut it off after correct number of bits (Different for single vs. double precision)
- Derive Sign, Exponent and Significand fields.

Example

What is the decimal equivalent of the floating point number given below:

1	1000 0001	111 0000 0000 0000 0000 0000

S Exponent Significand

$$(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

$$(-1)^{1} \times (1 + .111) \times 2^{(129-127)}$$

$$-1 \times (1.111) \times 2^{(2)}$$

$$-111.1$$

$$-7.5$$