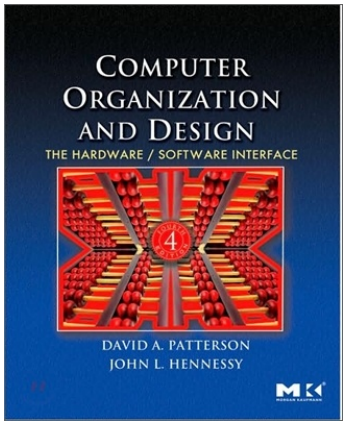
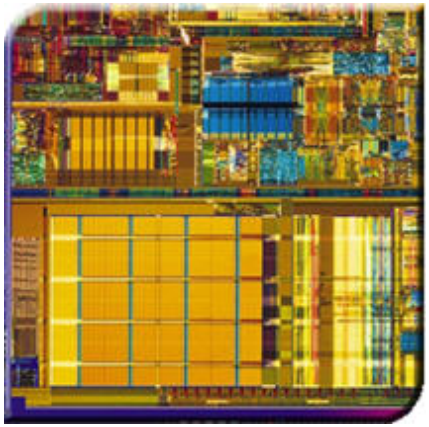
# Computer Architecture

## Lecture 4  Numbers & ALU



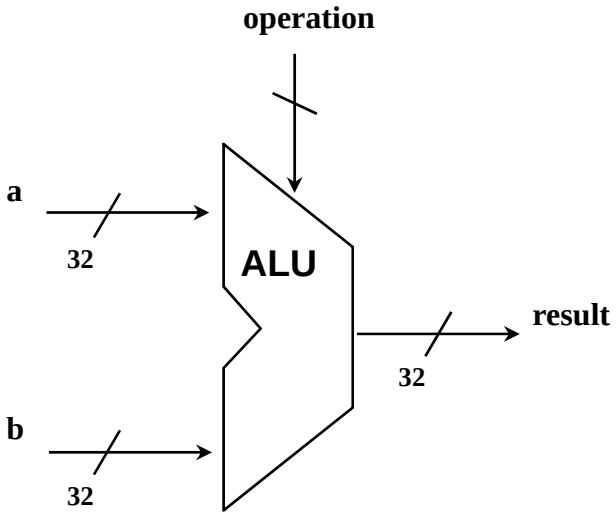



**Prof. Jongmyon Kim**

울산대학교
UNIVERSITY OF ULSAN

# Arithmetic

- Where we've been:
  - Performance (seconds, cycles, instructions)
  - Abstractions:
    Instruction Set Architecture
    Assembly Language and Machine Language
- What's up ahead:
  - Implementing the Architecture



# 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: $0...2^n-1$
- Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - negative numbers (e.g., no MIPS subi instruction; addi can add a negative number)
- How do we represent negative numbers? i.e., which bit patterns will represent which numbers?

# Possible Representations

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

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

# MIPS

- 32 bit signed numbers:

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

# Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  - remember: “negate” and “invert” are quite different!
- 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
```

  - "sign extension" (lbu vs. lb)

# Addition & Subtraction

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

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

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

```
  0111
+ 1010
```

- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
  0111
+ 0001
  1000
```

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

# Detecting Overflow

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

# Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow
  — new MIPS instructions: `addu, addiu, subu`

  *note:* `addiu` *still sign-extends!*
  - *same as addi except for no overflow exception*

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

# Review: Boolean Algebra & Gates

- Problem: Consider a logic function with three inputs: A, B, and C.

  Output D is true if at least one input is true
  Output E is true if exactly two inputs are true
  Output F is true only if all three inputs are true

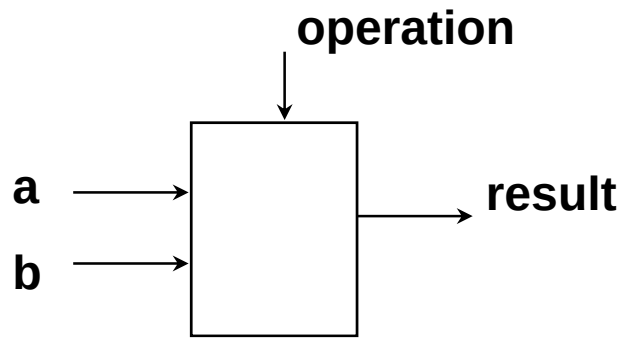
- Show the truth table for these three functions.

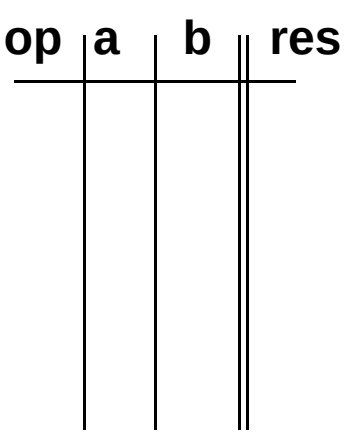
- Show the Boolean equations for these three functions.

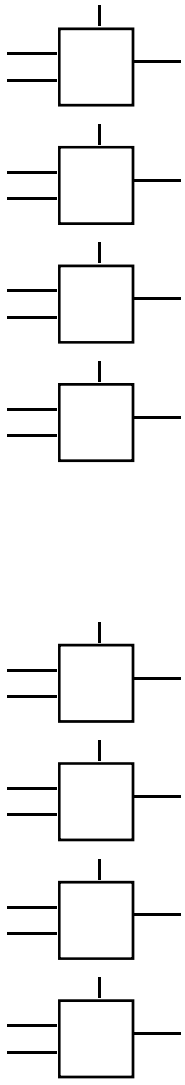
- Show an implementation consisting of inverters, AND, and OR gates.

# ALU (arithmetic logic unit)

- Let's build an ALU to support the "and" and "or" instructions
  - we'll just build a 1 bit ALU, and use 32 of them (bit-slice)

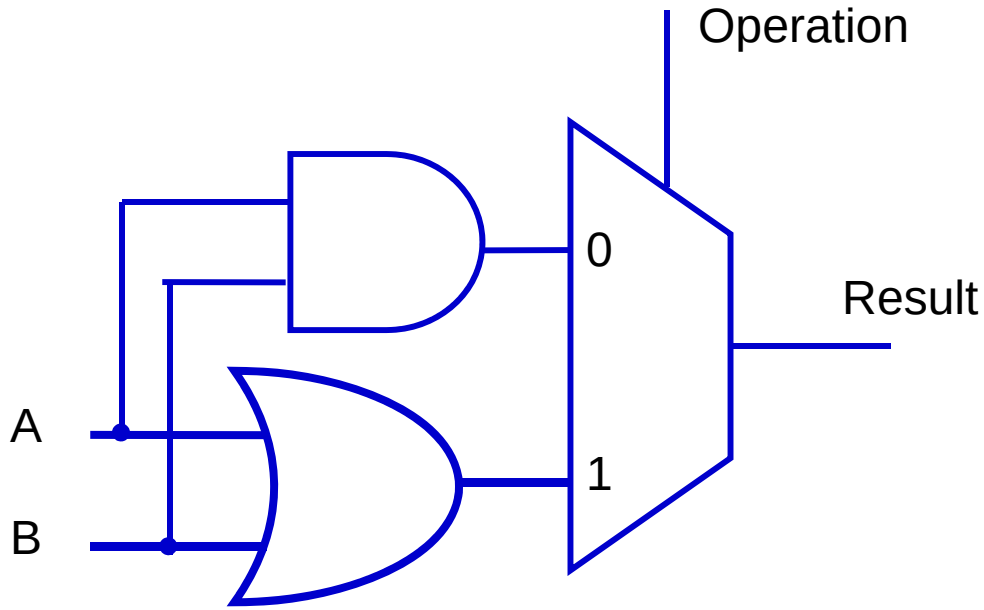


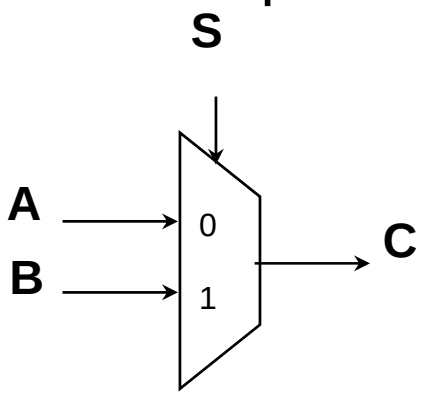
- Possible Implementation (sum-of-products):

# AND and OR ALU



# Review: The Multiplexer

- Selects one of the inputs to be the output, based on a control input

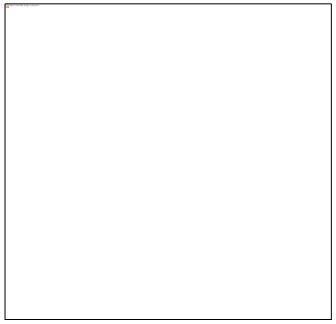



***note: we call this a 2-input mux even though it has 3 inputs!***

- Lets build our ALU using a MUX:

# Different Implementations

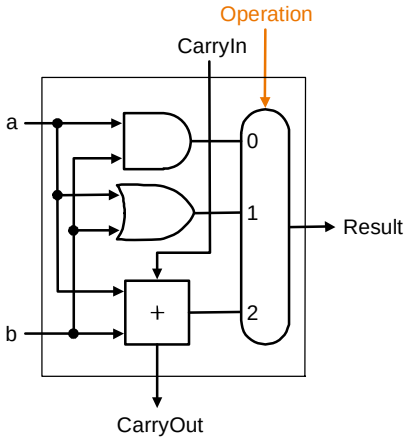
- Not easy to decide the “best” way to build something
  - Don't want too many inputs to a single gate
  - Don’t want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:

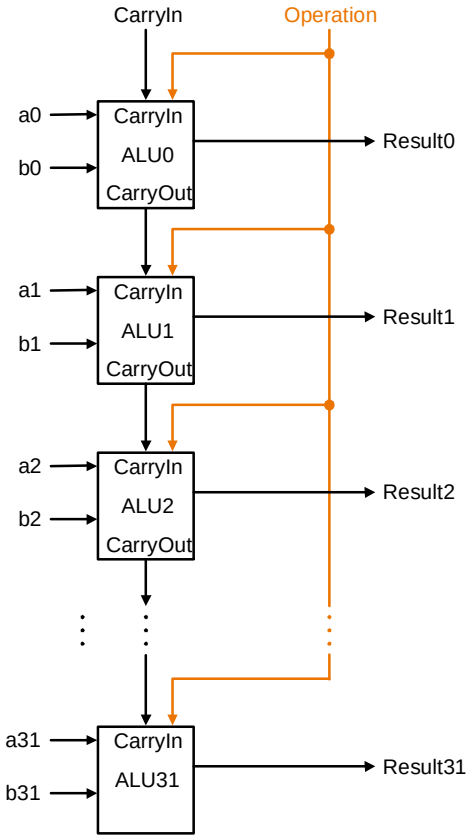


$$c_{out} = ab + ac_{in} + bc_{in}$$
$$sum = a \oplus b \oplus c_{in}$$

- How could we build a 1-bit ALU for *add*, *and*, and *or*?
- How could we build a 32-bit ALU?

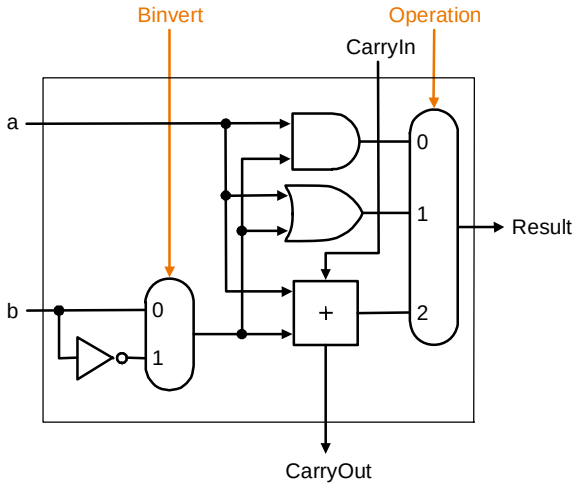
# Building a 32-bit ALU

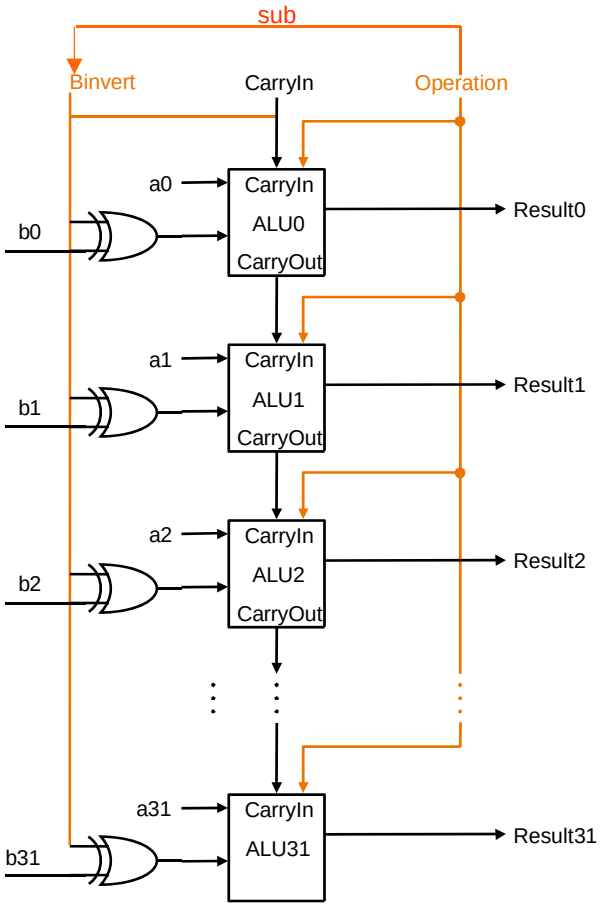


# Subtraction (a – b) ?

- Two's complement approach: just negate b and add 1.
- How do we negate?
- A clever solution:

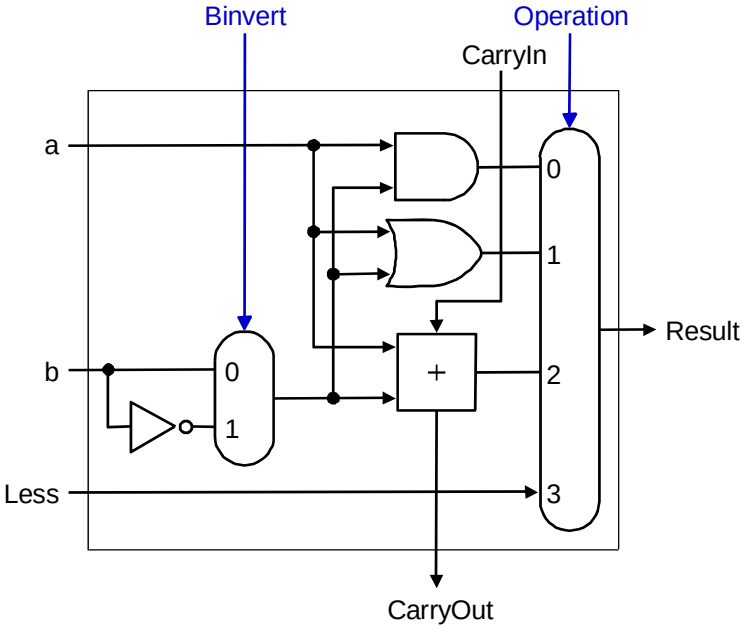


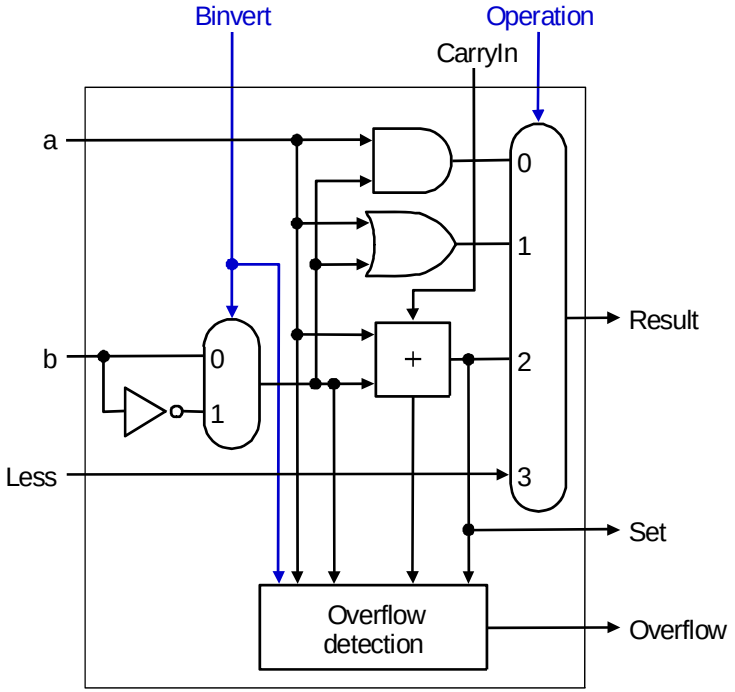



# Tailoring the ALU to the MIPS

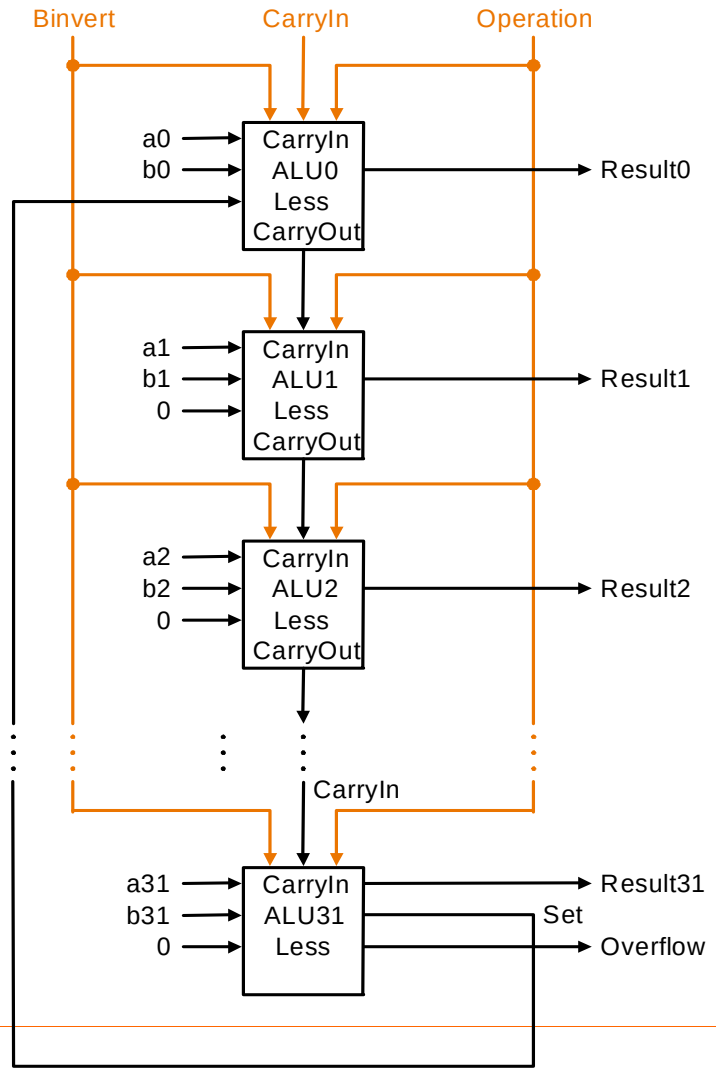
- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: (a-b) = 0 implies a = b

# Supporting slt



# What Result31 is when (a-b)<0?

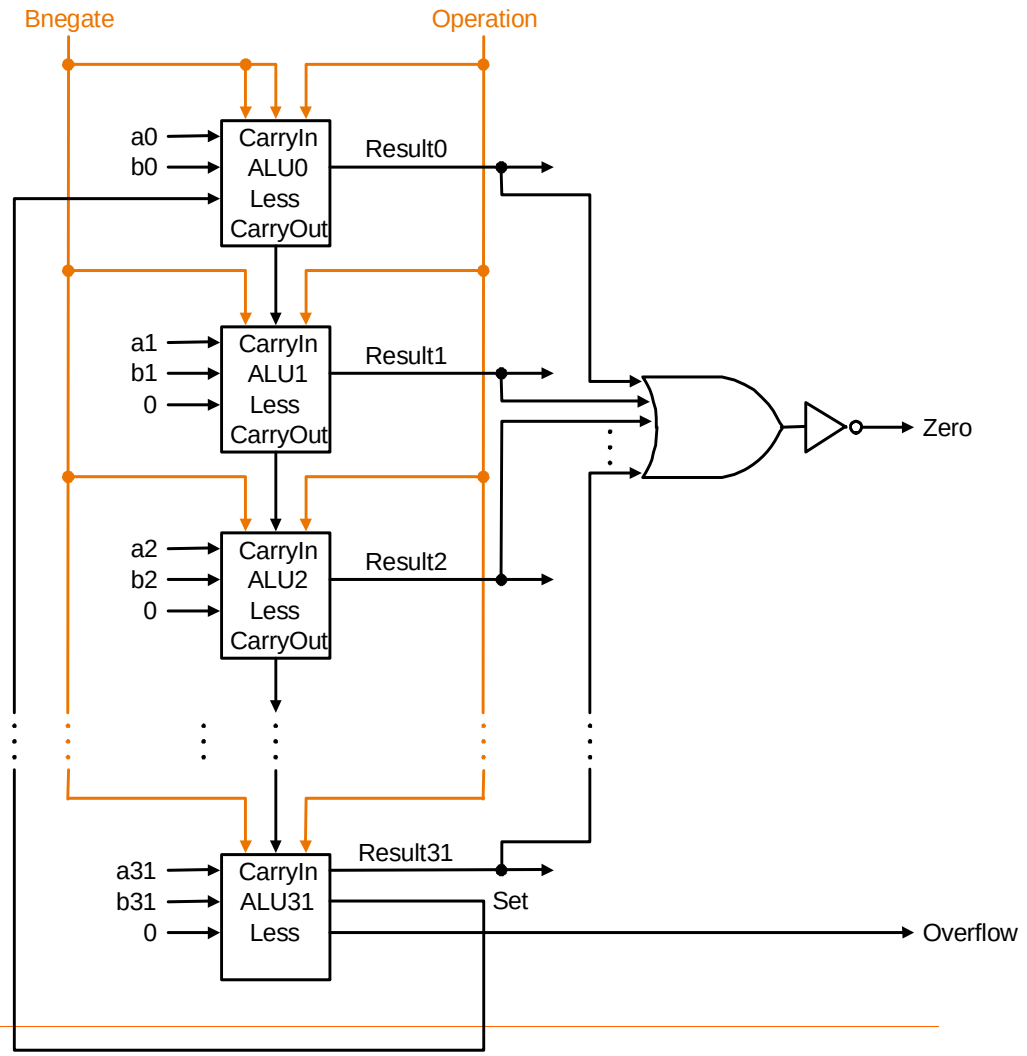


# Test for equality

- Notice control lines:

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

•***Note: zero is a 1 when the result is zero!***



# Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexer to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at two examples for addition and multiplication

# Problem: Ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?

Can you see the ripple? How could you get rid of it?

An approach in-between our two extremes

- Motivation:
  - If we didn't know the value of carry-in, what could we do?
    - $c_{i+1} = a_i\ b_i\ +\ a_i\ c_i\ +\ b_i\ c_i = a_i\ b_i\ +\ c_i\ (a_i\ +\ b_i)$
  - When would we always generate a carry? $g_i\ =\ a_i\ b_i$
  - When would we propagate the carry? $p_i\ =\ a_i\ +\ b_i$
- Did we get rid of the ripple?

# Carry Lookahead

$$C_{i+1} = A_i B_i + C_i (A_i + B_i)$$

$$g_i = A_i B_i \qquad \text{(generate)}$$

$$p_i = A_i + B_i \qquad \text{(propagate)}$$

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

$$C_1 = g_0 + p_0 C_0$$

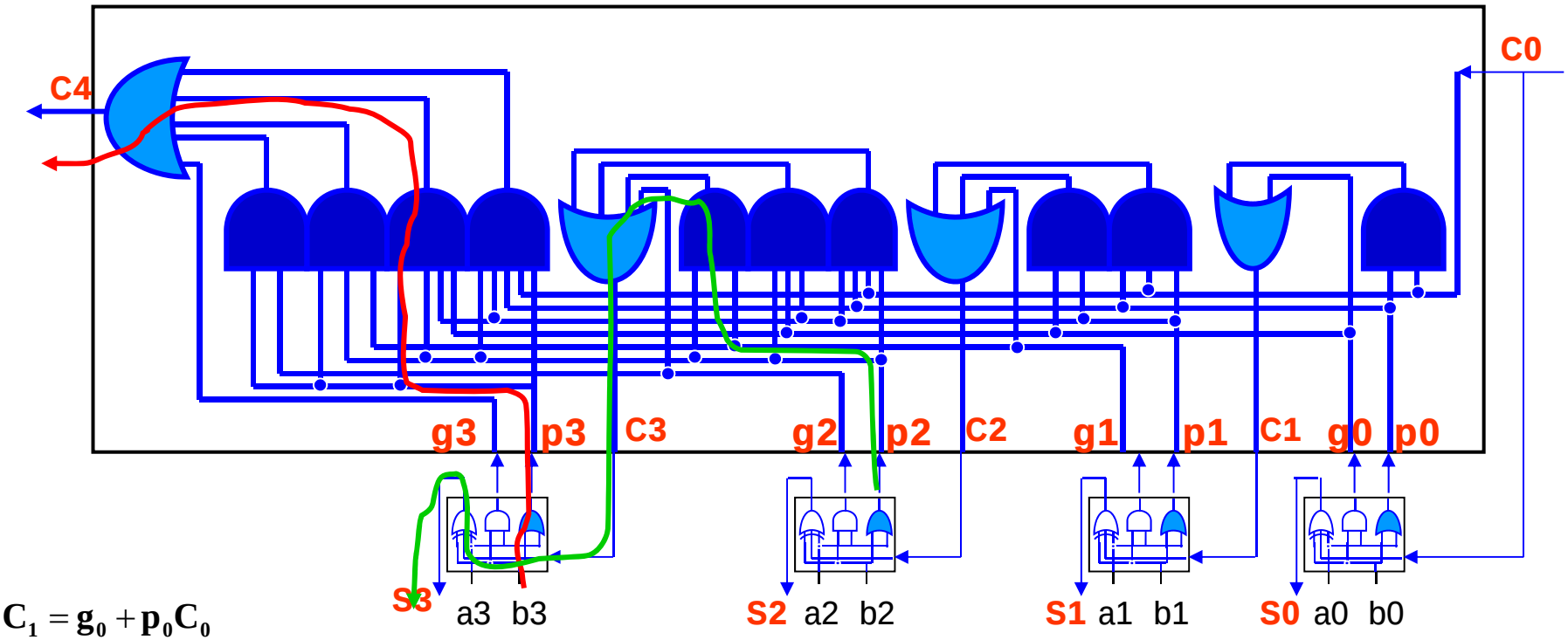
$$C_2 = g_1 + p_1 C_1 = g_1 + p_1 g_0 + p_1 p_0 C_0$$

$$C_3 = g_2 + p_2 C_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0$$

$$C_4 = g_3 + p_3 C_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0$$

Note that all the carry's are only dependent on input A and B and C

# 4-bit Carry Lookahead Adder




$$C_1 = g_0 + p_0 C_0$$

$$C_2 = g_1 + p_1 C_1 = g_1 + p_1 g_0 + p_1 p_0 C_0$$

$$C_3 = g_2 + p_2 C_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0$$

$$C_4 = g_3 + p_3 C_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0$$

**Only 3 Gate Delay for each Carry $C_i$**
**$= D_{AND} + 2*D_{OR}$**

**4 Gate Delay for each Sum $S_i$**
**$= D_{AND} + 2*D_{OR} + D_{XOR}$**

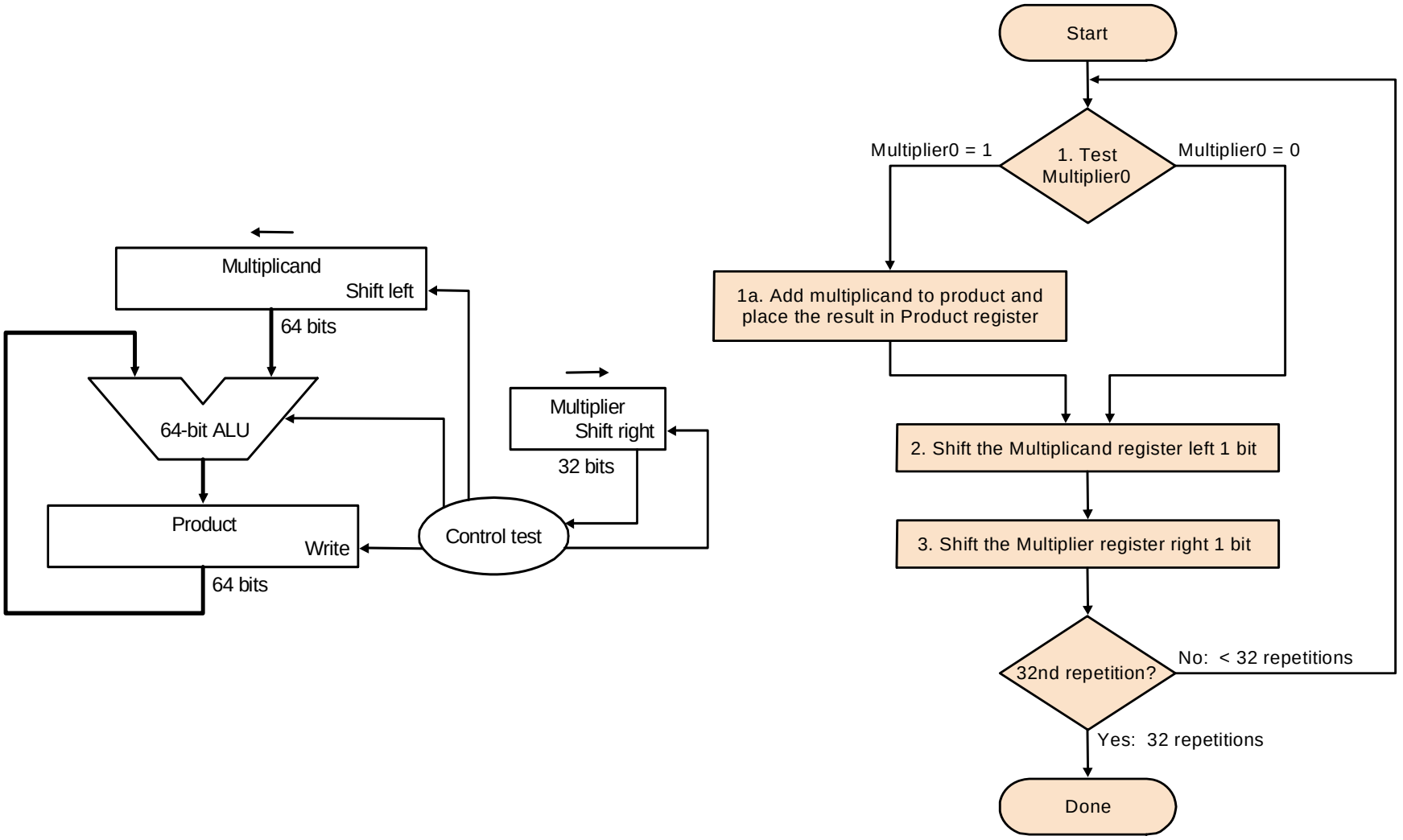
# Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

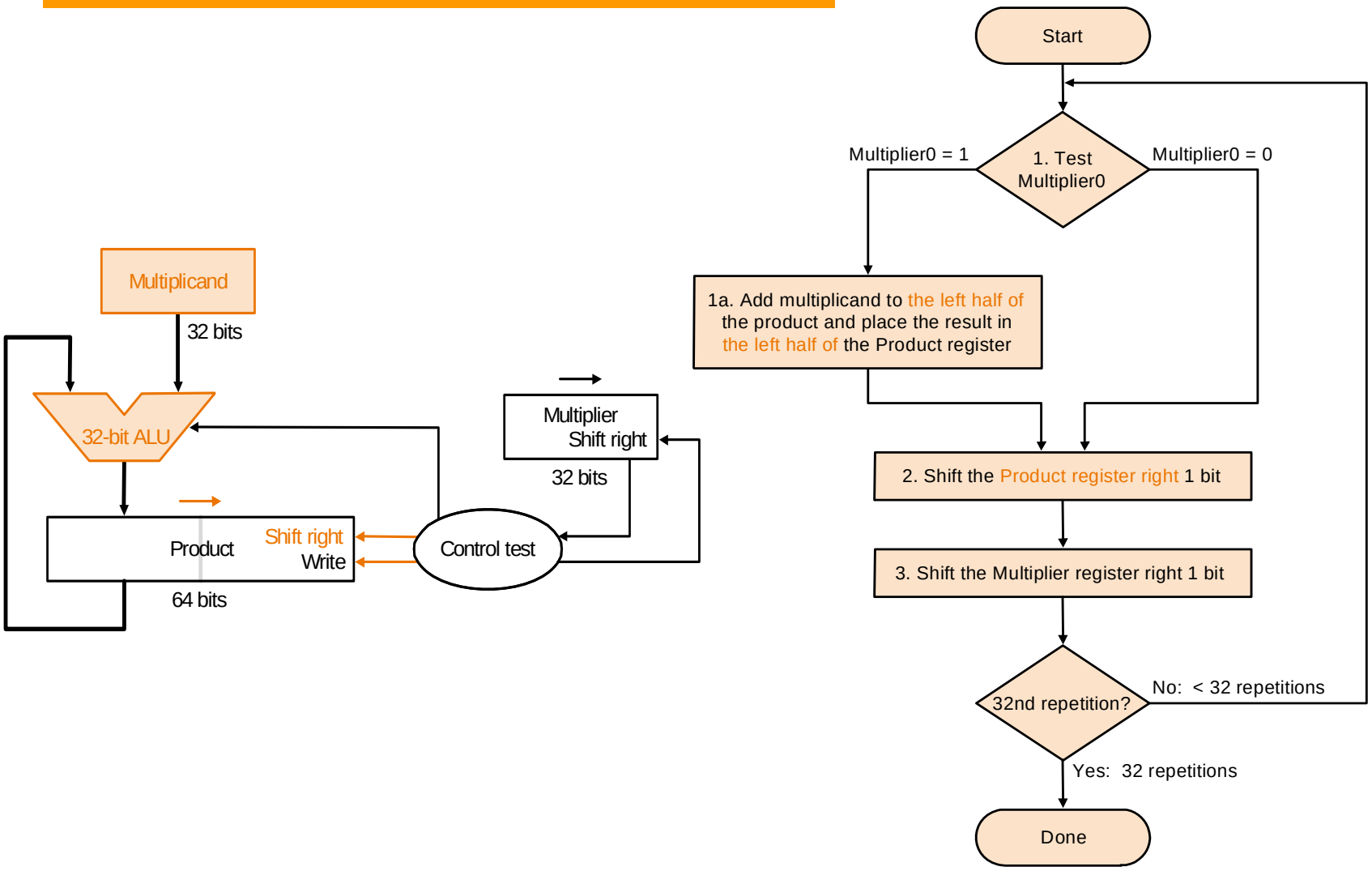
```
      0010   (multiplicand)
__x_1011   (multiplier)
```

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

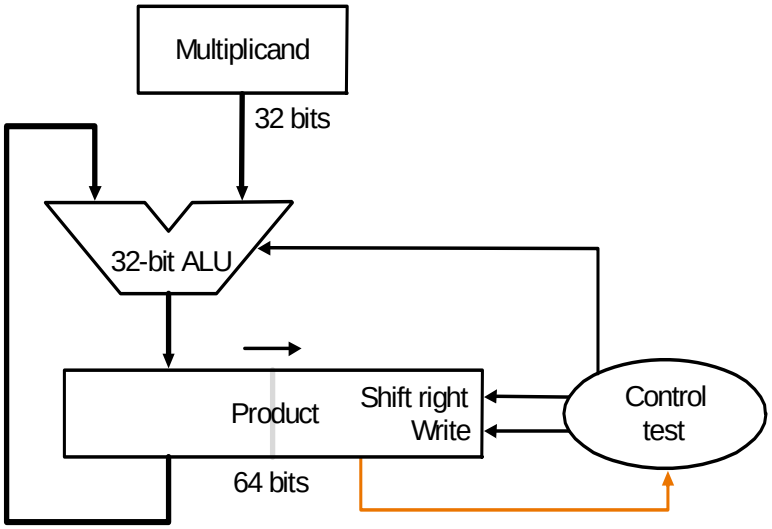
# Multiplication: Implementation

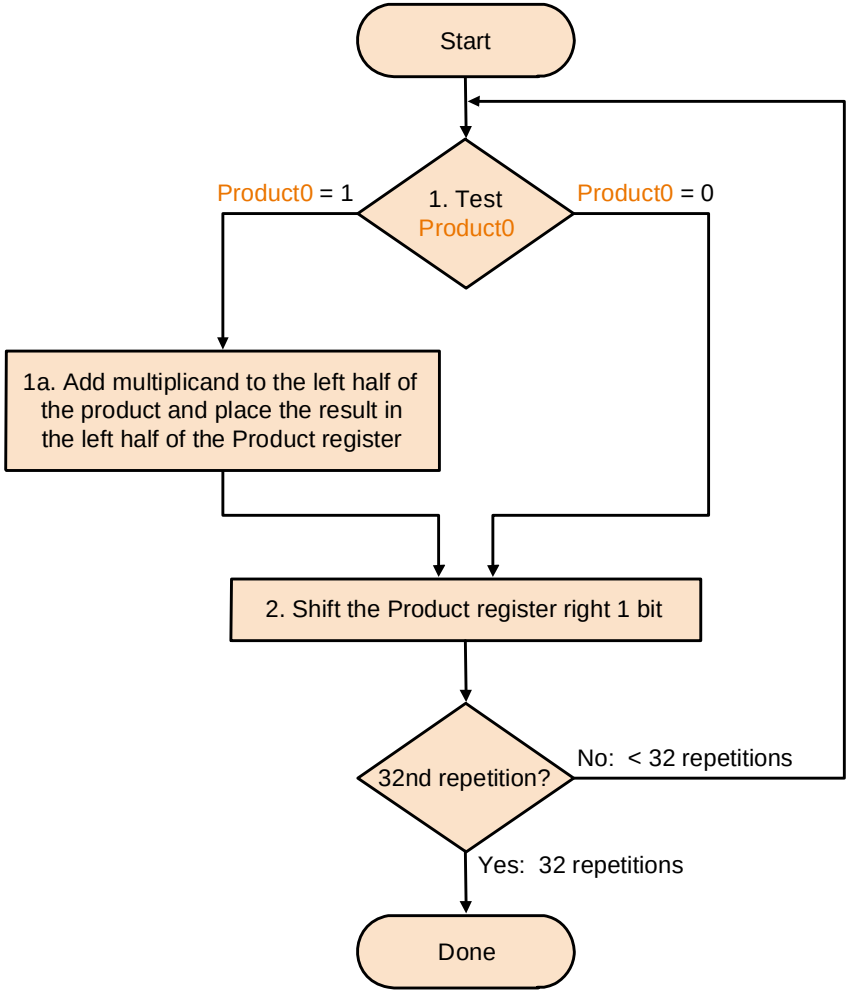


# Second Version



# Final Version



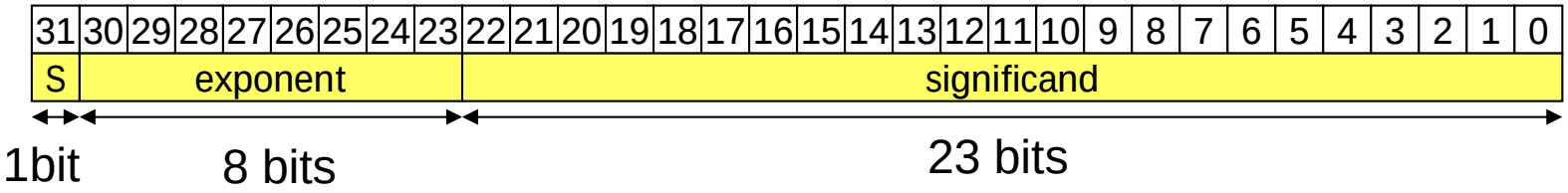



# Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., $3.15576 \times 10^9$
- Representation:
  - sign, exponent, significand: $(-1)^{sign} \times significand \times 2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand
  - Signficand = (1 + fraction), defined implicitly

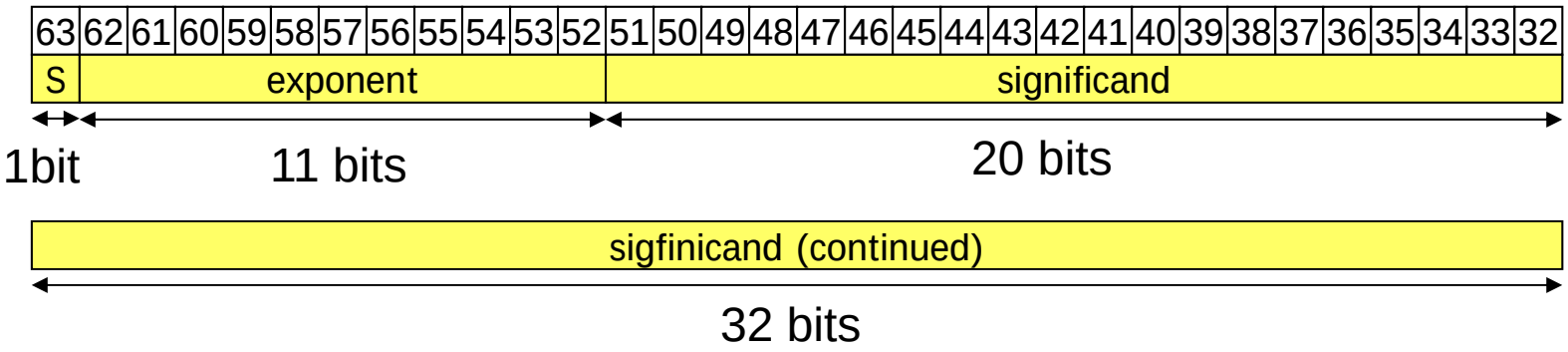
# IEEE 754 Standard Floating-point Representation

## Single Precision (32-bit)




$$(-1)^{sign} \times (1+fraction) \times 2^{exponent-127}$$

## Double Precision (64-bit)




$$(-1)^{sign} \times (1+fraction) \times 2^{exponent-1023}$$

# IEEE 754 floating-point standard

- Leading “1” bit of significand is implicit

- Exponent is “biased” to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{sign} \times (1+fraction) \times 2^{exponent - bias}$ (a.k.a. a normalized number – because of the 1 for scientific notation)

- Example:
  - decimal: -.75 = -3/4 = $-3/2^2$
  - binary: -.11 = $-1.1 \times 2^{-1}$
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: `10111111010000000000000000000000`

# IEEE 754 Standard Example (single precision)

울산대학교 UNIVERSITY OF ULSAN

$$(-1)^{sign} * (1 + Fraction) * 2^{(\exp - 127)}$$

$$= (-1)^{sign} * (1 + s1 * 2^{-1} + s2 * 2^{-2} + s3 * 2^{-3} + \ldots + s23 * 2^{-23}) * 2^{(\exp - 127)}$$

$(-0.75)_{10} = (????????)_{16}$

$(-0.75)_{10} = (BF400000)_{16}$

# IEEE 754 Standard Example (single precision)

울산대학교 UNIVERSITY OF ULSAN

$$(-1)^{sign} * (1 + Fraction) * 2^{(\exp - 127)}$$

$$= (-1)^{sign} * (1 + s1 * 2^{-1} + s2 * 2^{-2} + s3 * 2^{-3} + \ldots + s23 * 2^{-23}) * 2^{(\exp - 127)}$$

$(23.15625)_{10} = (????????)_{16}$ $(23.15625)_{10} = (41B94000)_{16}$

# IEEE 754 Standard Encoding

| Single Precision | | Double Precision | | Object Represented |
|---|---|---|---|---|
| Exponent | Fraction | Exponent | Fraction | |
| 0 | 0 | 0 | 0 | 0 (zero) |
| 0 | Non-zero | 0 | Non-zero | ±Denormalized number |
| 1-254 | Anything | 1-2046 | Anything | ±Floating-point number |
| 255 | 0 | 2047 | 0 | ± Infinity |
| 255 | Non-zero | 2047 | Non-zero | NaN (Not a Number) |

- **NaN : (infinity – infinity), or 0/0**
- **Denormalized number = $(-1)^{sign} * 0.f * 2^{1-bias}$**

# Precision Issues

- Cannot represent all possible real numbers, they are infinite !
- Must be sacrifice precision when representing FP numbers in some cases
  - Precision lost when integer portion is too large
  - Precision lost when fraction portion is too small
- Example
  - How to represent $2^{24}$ and $2^{24}+1$ ?
    - Both = 4B800000 in single precision
  - How to represent $2^{-127}$ ? (use denormalized number ?? $0.1*2^{-126}$)
  - How about $2^{-150}$? (use denormalized number ?? What is the smallest number by denormalized number?)

# Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!