Computer Architecture
Ch. 3-1: Design Process & ALU Design

Fall, 2014

Sao-Jie Chen (csj@ntu.edu.tw)

The Design Process

"To Design Is To Represent"

- **Design activity yields description/representation of an object**

⇒ Traditional craftsman does not distinguish between the concept-utilization and the artifact

⇒ Separation comes about because of *complexity*

⇒ The concept is captured in one or more *representation languages*

⇒ This process IS design

- ***Design Begins With Requirements***

⇒ Functional Capabilities: what it will do

⇒ Performance Characteristics: Speed, Power, Area, Cost, . . .

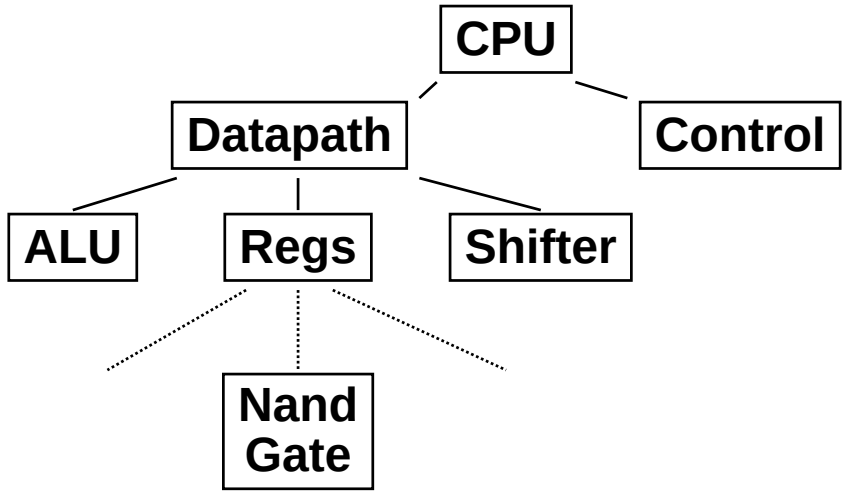
Design Process (cont.)

- *Design Finishes as Assembly*

  ⇒ Design understood in terms of components and how they have been assembled

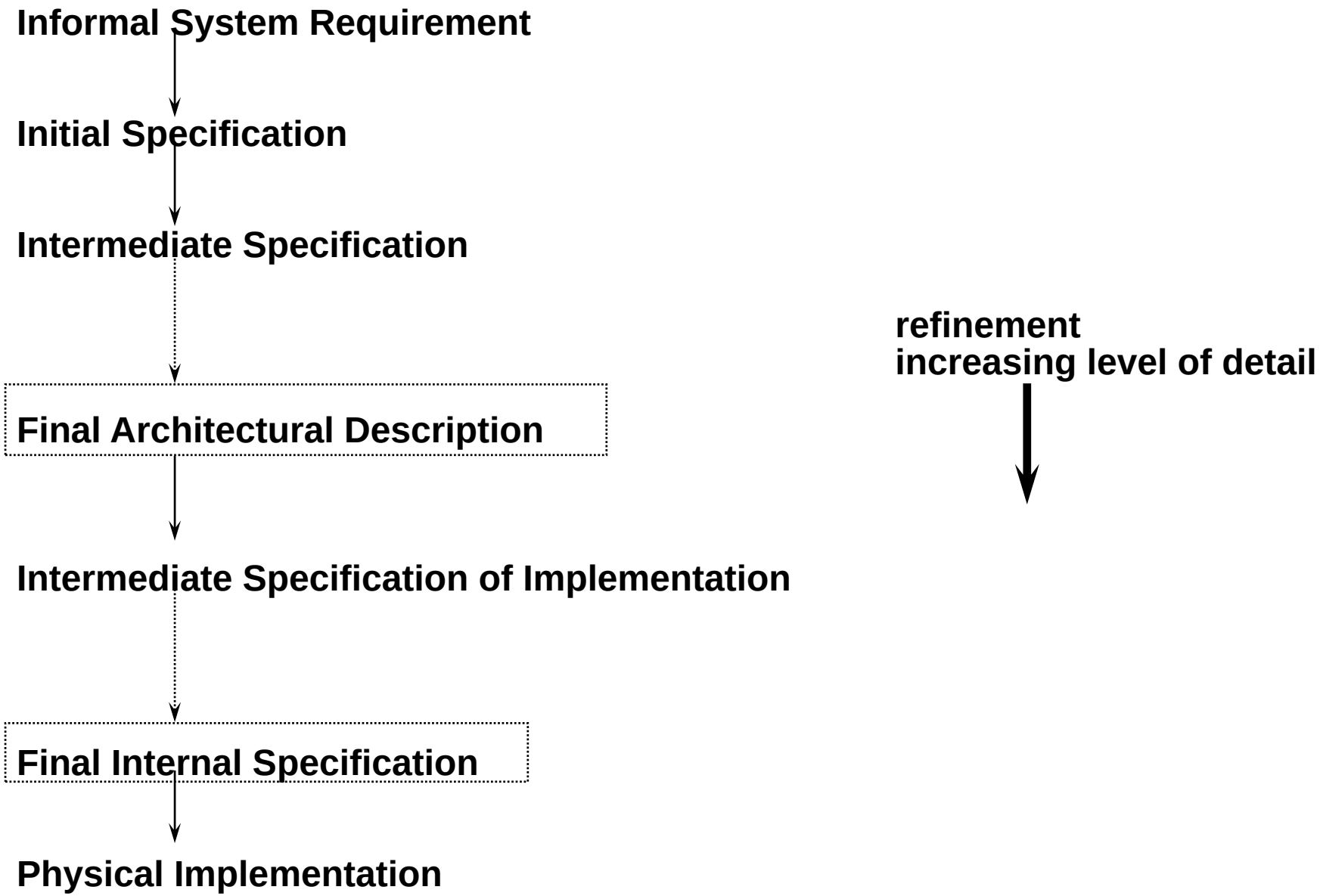
  ⇒ Top Down *decomposition* of complex functions (behaviors) into more primitive functions

  ⇒ bottom-up *composition* of primitive building blocks into more complex assemblies



- *Design is a "creative process," not a simple method*

Design Refinement



Design as Representation (Example)

(1) **Functional Specification** "VHDL Behavior"

Inputs: 2 x 16 bit operands: A, B; 1 bit carry input: Cin.

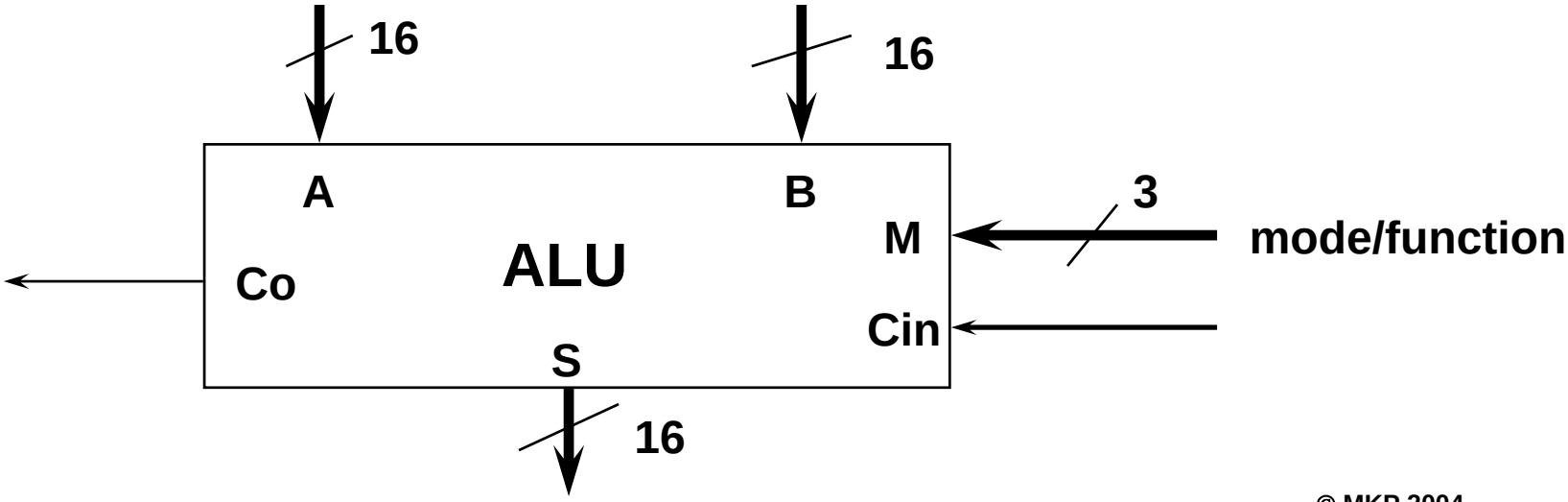
Outputs: 1 x 16 bit result: S; 1 bit carry output: Co.

Operations: PASS, ADD (A plus B plus Cin), SUB (A minus B minus Cin), AND, XOR, OR, COMPARE (equality)

Performance: left unspecified for now!

(2) **Block Diagram** "VHDL Entity"

Understand the data and control flows



Elements of the Design Process

- **Divide and Conquer**
  - **Formulate a solution in terms of simpler components.**
  - **Design each of the components (subproblems)**
- **Generate and Test**
  - **Given a collection of building blocks, look for ways of putting them together that meets requirement**
- **Successive Refinement**
  - **Solve *most* of the problem (i.e., ignore some constraints or special cases), examine and correct shortcomings.**
- **Formulate High-Level Alternatives**
  - **Articulate many strategies to *in mind* while pursuing any one approach.**
- **Work on the Things you Know How to Do**
  - **The unknown will become *obvious* as you make progress.**

Summary of the Design Process

- Hierarchical Design to manage complexity
- Top Down vs. Bottom Up vs. Successive Refinement
- Importance of Design Representations:

  Block Diagrams

  Decomposition into Bit Slices

  Truth Tables (TT), K-Maps

  Circuit Diagrams

  top down

  bottom up

  mux design meets at TT

  Other Descriptions: state diagrams, timing diagrams, reg xfer, . . .

- Optimization Criteria:

  Gate Count — *Area*

  [Package Count] — *Pin Out*

  Logic Levels — *Delay*

  Fan-in/Fan-out — *Delay*

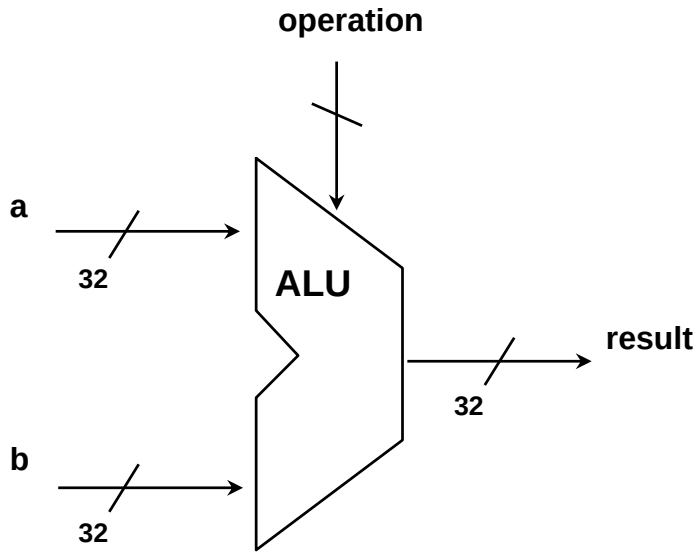
  Power

  Cost

  Design time

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)
  $\Rightarrow$ 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

Signed 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 (2's Complement):**

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

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$ — *minint*

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Two's Complement Representation

- 2's complement representation of negative numbers
  - Bitwise inverse and add 1
  - The MSB is always *1* for negative number => sign bit

- Biggest 4-bit Binary Number: 7 Smallest 4-bit Binary Number: -8

Decimal	Binary	Decimal	Bitwise Inverse	2's Complement
0	0000	0	1111	0000
1	0001	-1	1110	1111
2	0010	-2	1101	1110
3	0011	-3	1100	1101
4	0100	-4	1011	1100
5	0101	-5	1010	1011
6	0110	-6	1001	1010
7	0111	-7	1000	1001
8	1000	-8	0111	1000

"illegal" Positive Number!

Two's Complement Arithmetic

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal	2's Complement
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

- **Examples: 7 - 6 = 7 + (- 6) = 1**

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

3 - 5 = 3 + (- 5) = -2

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

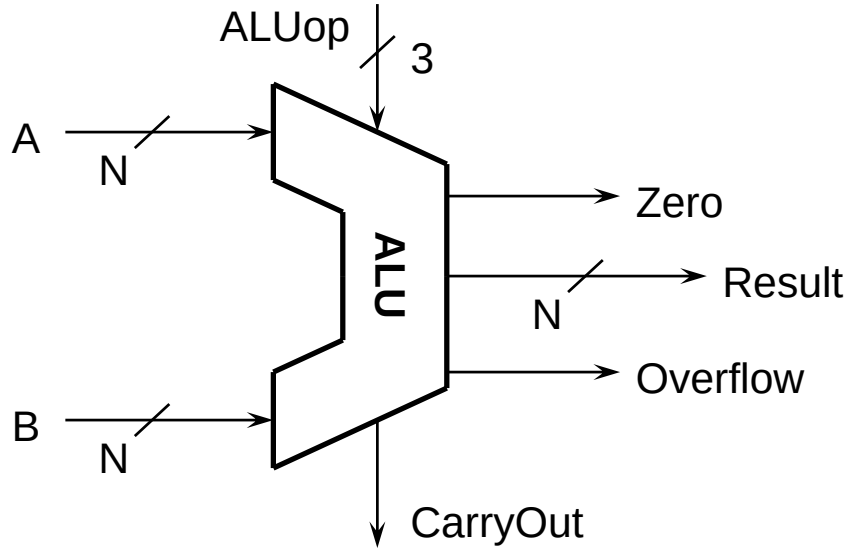
Two's Complement Operations

- 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)

Functional Specification of the ALU

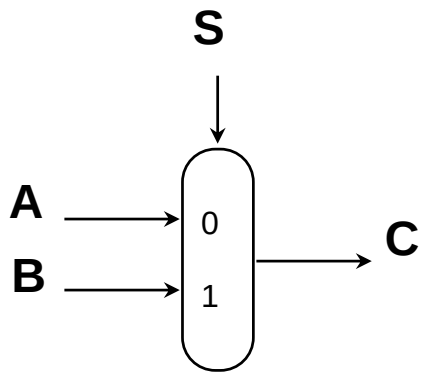


- **ALU Control Lines (ALUop)** — **Function**

ALU Control Lines (ALUop)	Function
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

Review: The Multiplexor

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

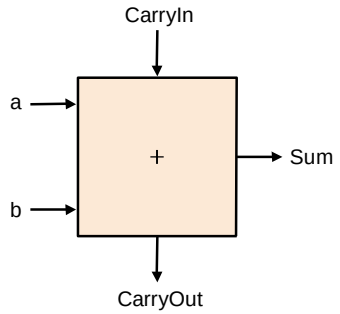


Note: *we call this a 2-input mux even though it has 3 inputs!*

- Let's 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 (FA) for addition:

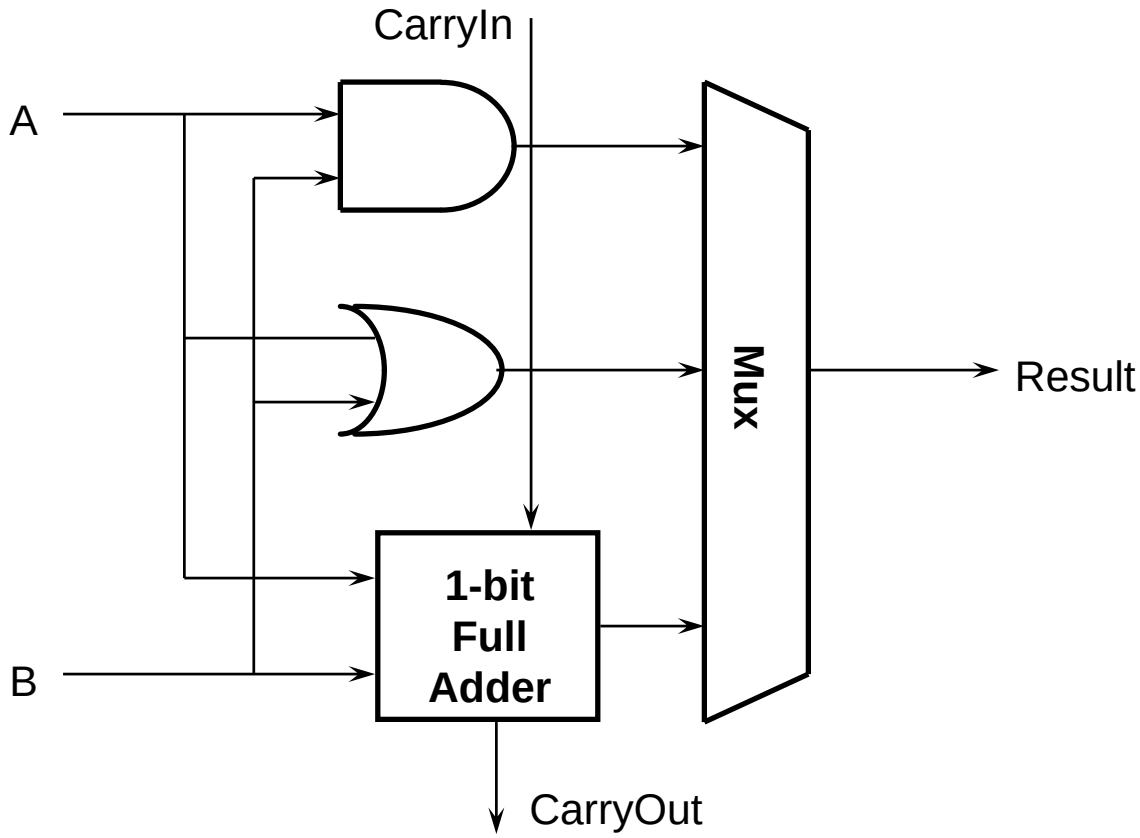


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

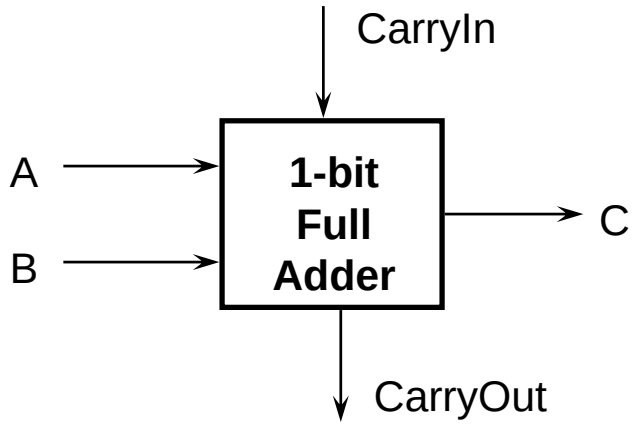
- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

A One-bit ALU

- **This 1-bit ALU will perform AND, OR, and ADD**



A One-bit Full Adder



- This is also called a (3, 2) adder
- Half Adder: No CarryIn nor CarryOut
- Truth Table:

3 Inputs			2 Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
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

Logic Equation for CarryOut

Inputs			Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
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

- **CarryOut = (!A & B & CarryIn) | (A & !B & CarryIn) | (A & B & !CarryIn) | (A & B & CarryIn)**
- **CarryOut = B & CarryIn | A & CarryIn | A & B**

Logic Equation for Sum

Inputs			Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
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

- **Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)**

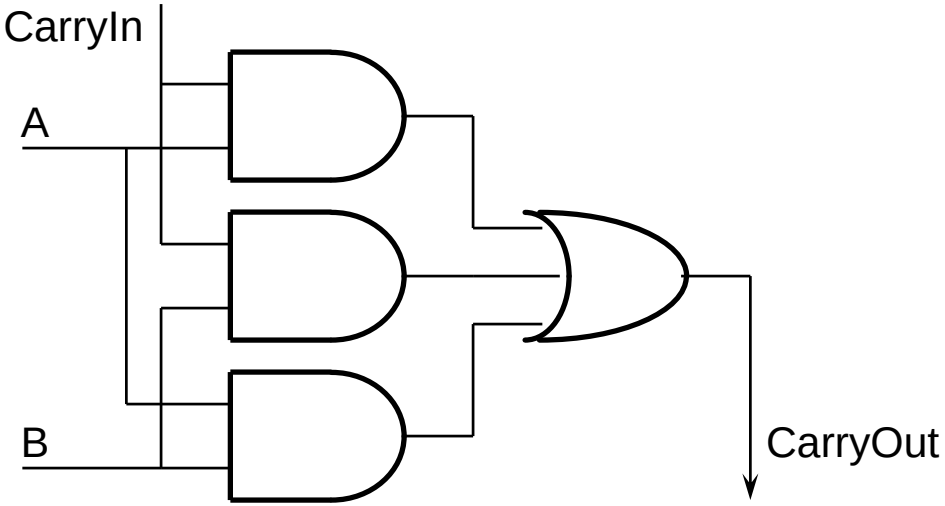
Logic Equation for Sum (continued)

- **Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)**
- **Sum = A XOR B XOR CarryIn**
- **Truth Table for XOR:**

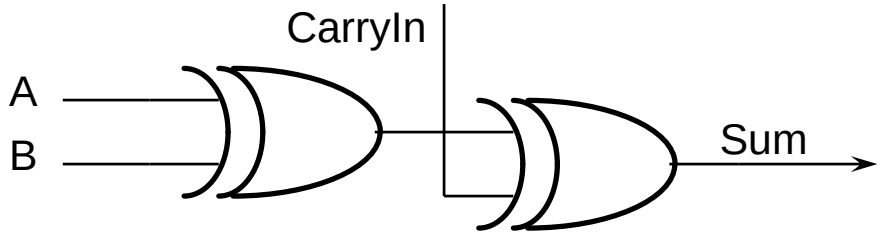
X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Logic Diagrams for CarryOut and Sum

- **CarryOut = B & CarryIn | A & CarryIn | A & B**

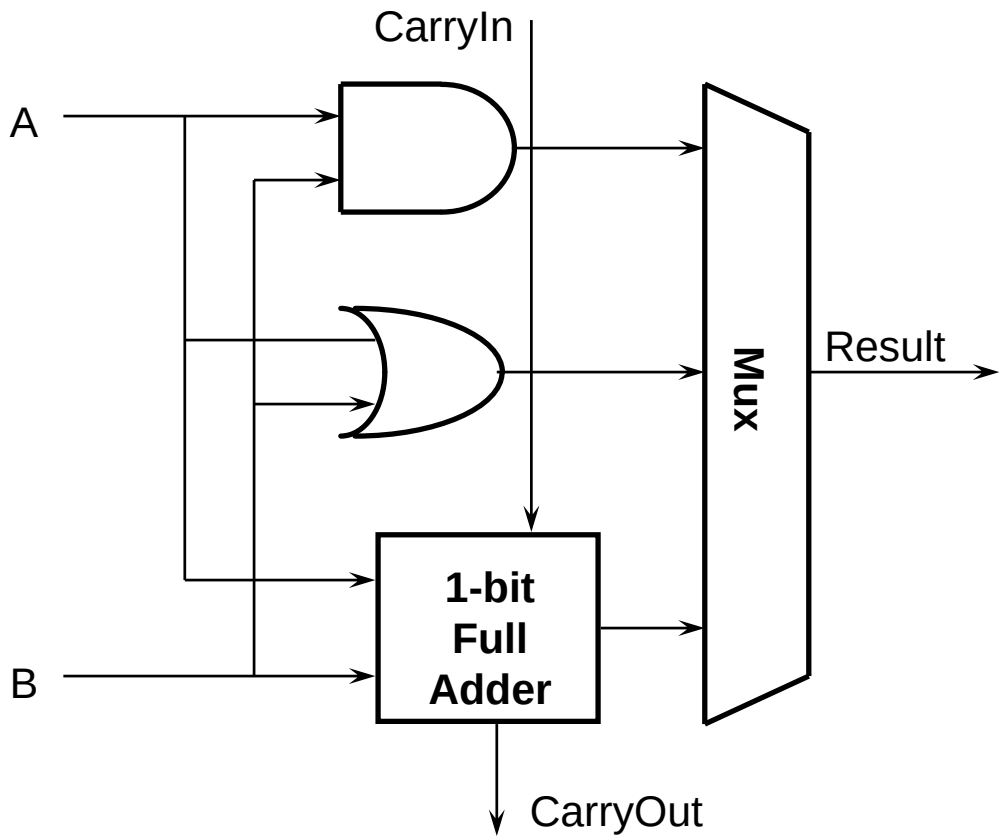


- **Sum = A XOR B XOR CarryIn**

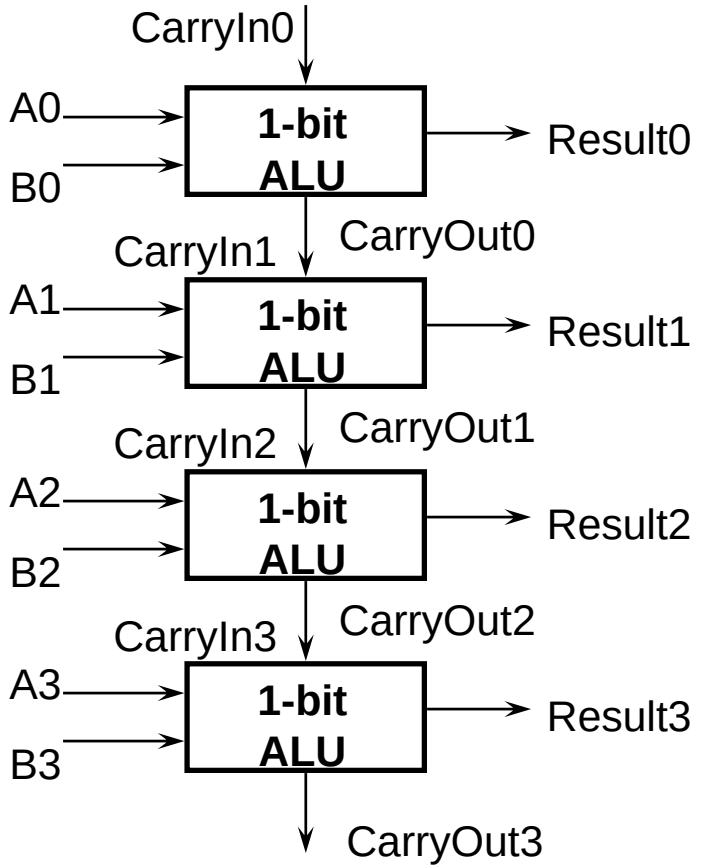


A 4-bit ALU

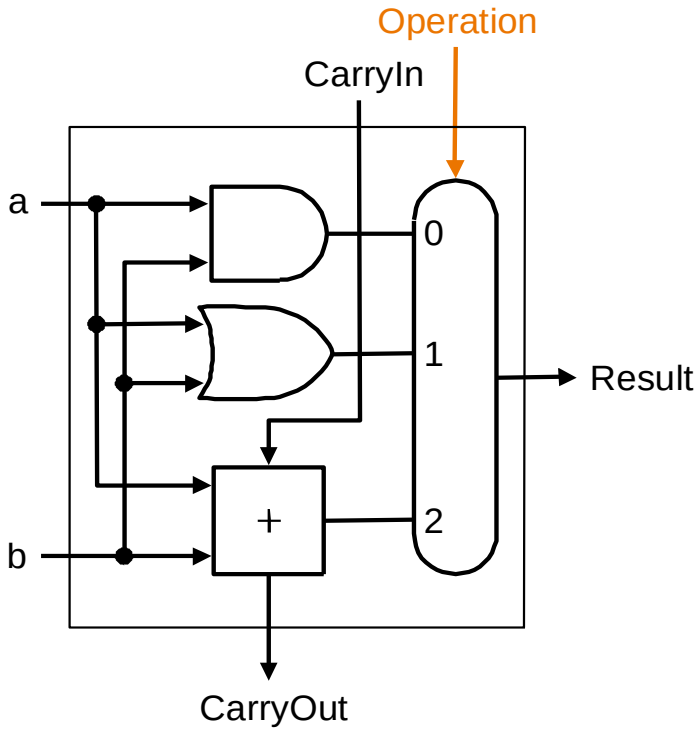
1-bit ALU

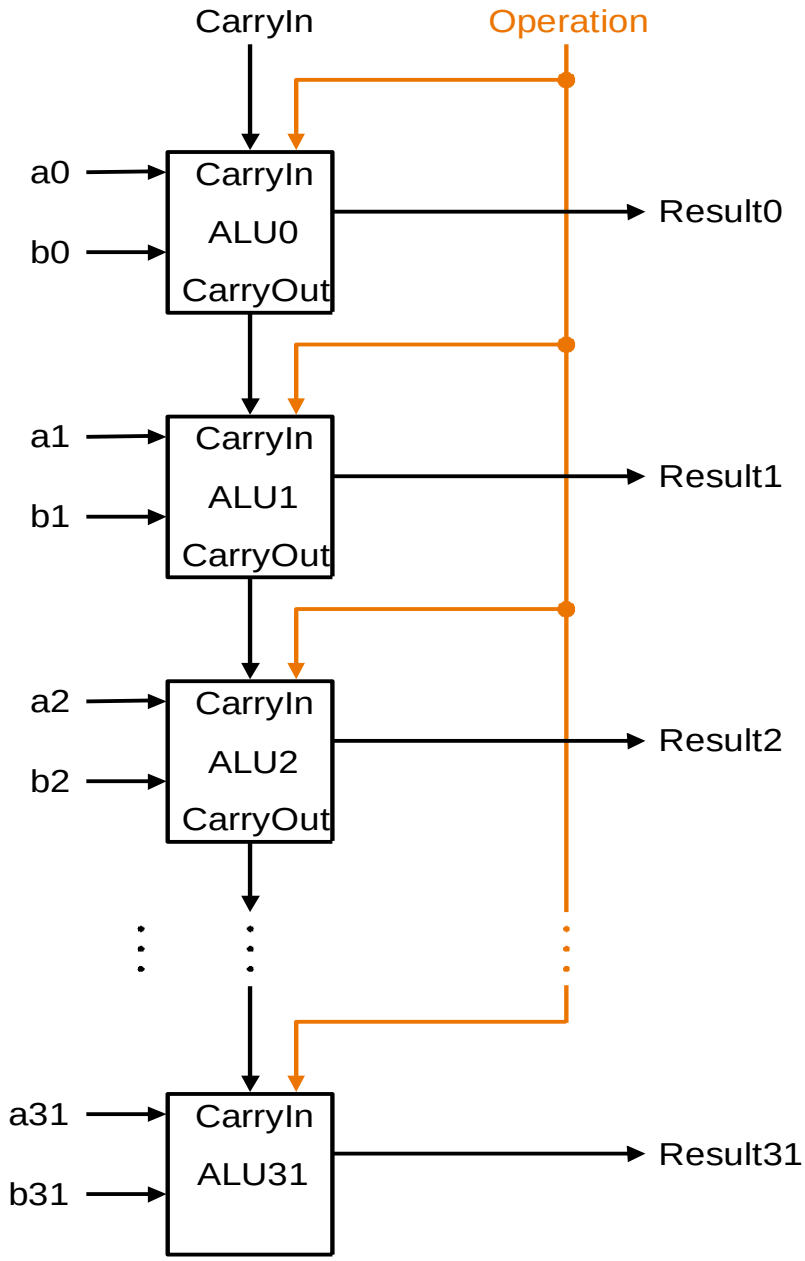


4-bit ALU



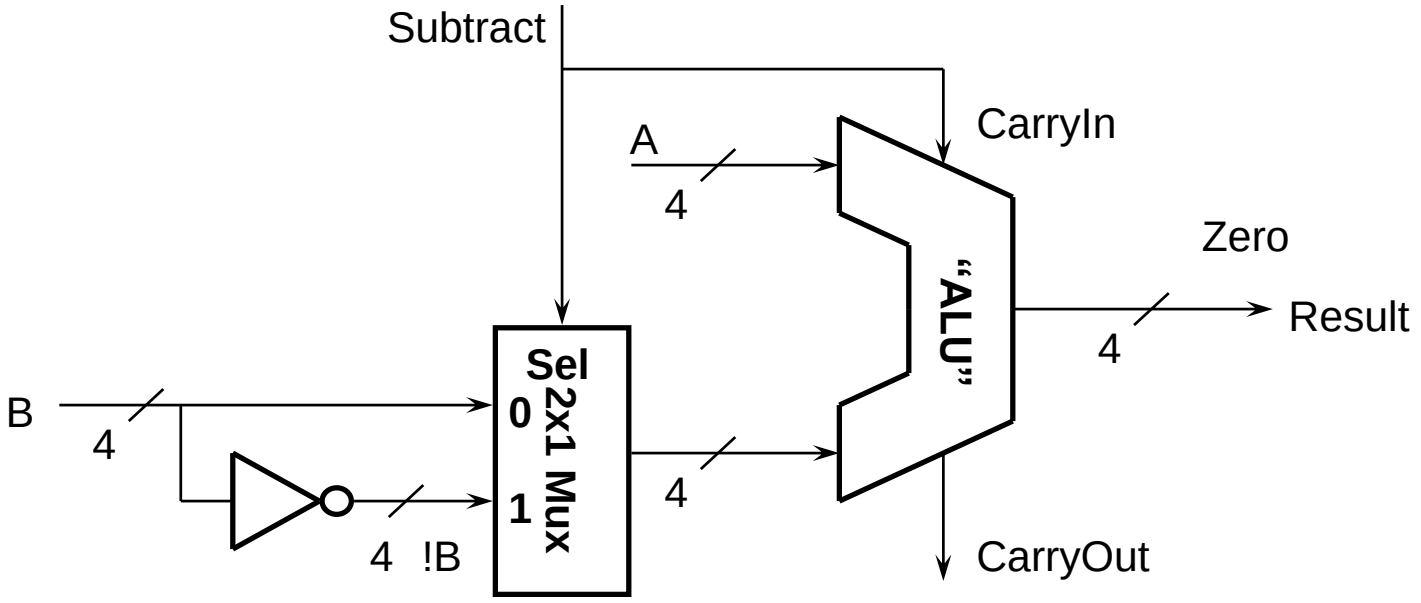
Building a 32-bit ALU





How About Subtraction?

- **Keep in mind the following:**
  - **(A - B) is the same as: A + (-B)**
  - **2's Complement: Take the inverse of every bit and add 1**
- **Bit-wise inverse of B is !B:**
  - **A + !B + 1 = A + (!B + 1) = A + (-B) = A - B**



What about subtraction (a – b) ?

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