CMSC 411
Computer Architecture

Lecture 6

Arithmetic Logic Unit

Lecture's Overview

Previous Lecture:

- Number representation
  (Binary vs. decimal, Sign and magnitude, Two's complement)
- Addition and Subtraction of binary numbers
  (Sign handling, Overflow conditions)
- Logical operations
  (Right and left shift, AND and OR)

This Lecture:

- Constructing an Arithmetic Logic Unit
- Scaling bit operations to word sizes
- Optimization for carry generation

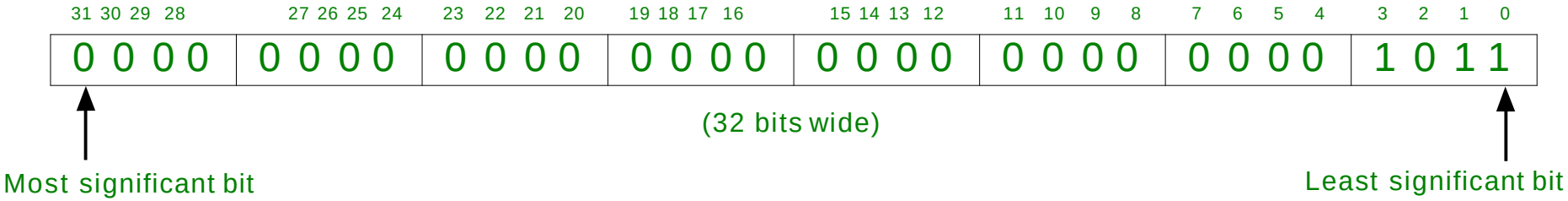
Introduction

- Computer words are composed of bits, thus words can be represented as binary numbers
- Although the natural number can be represented in binary:
  - ➔ How are negative numbers represented?
  - ➔ What is the largest number that can be represented in a computer word
  - ➔ What happens if an operation creates a number bigger than what can be represented?
  - ➔ What about fractions and real numbers?
  - ➔ How does hardware really add, subtract, multiply, or divide numbers?
  - ➔ What are the implications of all of these on instruction sets?

Unsigned Numbers

- Numbers can be represented in any base; humans prefer base 10 and base 2 is best for computers
- The first commercial computer did offer decimal arithmetic (binary decimal coded number) and proved to be inefficient
- In any base the value of the i^{th} digits d is: $d \times base^i$, where i starts at 0 and increases from right to left
- Example: $(1011)_2 = (1 \times 2^3)_{10} + (0 \times 2^2)_{10} + (1 \times 2^1)_{10} + (1 \times 2^0)_{10}$

$$= 8 + 0 + 2 + 1 = (11)_{10}$$



- The MIPS word is 32 bit long → 2^{32} different numbers could be represented

$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000)_2 = (0)_{10}$
$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 = (1)_{10}$
.....
...
$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110)_2 = (4, 294, 967, 294)_{10}$
$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111)_2 = (4, 294, 967, 295)_{10}$

ASCII versus Binary Numbers

- Computers were invented to crunch numbers, but very soon after they were used to process text
- Most computers today use 8-bit bytes to represent characters using the *American Standard Code for Information Exchange* (ASCII)
- If numbers are represented as strings of ASCII digits they will need significantly larger storage and arithmetic operations will be very slow
- Example:

  What is the expansion in storage if the number 1 billion is represented in ASCII versus 32-bit integer?

  1 billion = 1, 000, 000, 000 ➔ it would need 10 ASCII digits (bytes)

  Thus the storage expansion = (10 digits × 8 bits) / 32 = 2.5

Computer professionals are raised to believe that binary is natural

Sign and Magnitude Representation

- Computer programs calculate both positive & negative numbers and thus the the number representation has to distinguish both
- In sign and magnitude representation, a single bit is designated either on the left or the right of the number to indicate its sign
- Although the sign and magnitude representation is very simple, yet it has multiple shortcomings:
  - ➔ It is not obvious where to put the sign bit: to the right or the left?
  - ➔ Adders may need an extra step to set the sign
  - ➔ A separate sign bit means that there will be a positive and negative zero
- Example:

$(+ 13)_{10} = (01101)_{2\ \text{sign/magnitude}}$ $(- 13)_{10} = (11101)_{2\ \text{sign/magnitude}}$

Sign and magnitude was shortly abandoned after their early use

Two's Complement Representation

- The two's complement of a number *X* represented in *n* bits is $2^n - X$
- Negative numbers would always have one in the most significant bit → easy to be tested by hardware
- Advantages:
  - ✓ There is only one zero in the two's complement representation (programmer happy)
  - ✓ Simple hardware design for arithmetic and logical operations (Designer happy)
- Disadvantage:
  - ➢ Most positive number is $2^{n-1}-1$, while least negative number is -2^{n-1} (programmer unhappy)
- To compute the decimal value of a 32-bit two's compliment number the following formula could be used:

$$(X_{31} \times -2^{31}) + (X_{30} \times 2^{30}) + (X_{29} \times 2^{29}) + \ldots + (X_1 \times 2^1) + (X_0 \times 2^0)$$

Example: $(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100)_2$

$= (1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$

$= (-4)_{10}$

Numbers in a MIPS' Word

$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000)_2 = (\quad 0)_{10}$

$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 = (\quad 1)_{10}$

$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 = (\quad 2)_{10}$

....

$(0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101)_2 = (\ 2,147,483,645)_{10}$

$(0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110)_2 = (\ 2,147,483,646)_{10}$

$(0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111)_2 = (\ 2,147,483,647)_{10}$

$(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000)_2 = (-2,147,483,648)_{10}$

$(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 = (-2,147,483,647)_{10}$

$(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010)_2 = (-2,147,483,646)_{10}$

....

$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101)_2 = (-\quad 3)_{10}$

$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110)_2 = (-\quad 2)_{10}$

$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111)_2 = (-\quad 1)_{10}$

- Two's complement does have one negative number that has no corresponding positive number
- The most positive and the least negative number are different in all bits

Quick negation for Two's Complement

Method 1:

- Convert every 1→0 and every 0→1 and then add 1 to the rest

Method 2:

❶ Move from right to left leave every leading 0's until reaching the first 1

❷ Convert every 0→1 and 1→0 afterward until reaching the left end

Example: Negate $(2)_{10}$

$(2)_{10} = (0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010)_2$

Method 1:

```
  1111 1111 1111 1111 1111 1111 1111 1101
+                                       1
-----------------------------------------
  1111 1111 1111 1111 1111 1111 1111 1110
```

Method 2:

0000 0000 0000 0000 0000 0000 0000 0010

1111 1111 1111 1111 1111 1111 1111 1110

Shortcuts for Two's Complement

❑ Sign extension

- When loading numbers in a wide register, the empty bits will be filled with the value of the sign bit
- Example: Convert 16-bit versions of $(2)_{10}$ and $(-2)_{10}$ to 32-bit binary numbers

  ➔ The 16-bit binary version of the number $(2)_{10}$ is $(0000\ 0000\ 0000\ 0010)_{2.}$
  If converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word, we get

  $(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010)_2$

  ➔ For $(-2)_{10}$ the 16-bit binary version is $(1111\ 1111\ 1111\ 1110)_2$ and again by making 16 copies of the value in the most significant bit (1) and placing that in the left-hand half of the word, we get:

  $(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110)_2$

❑ Grouping Binary Numbers

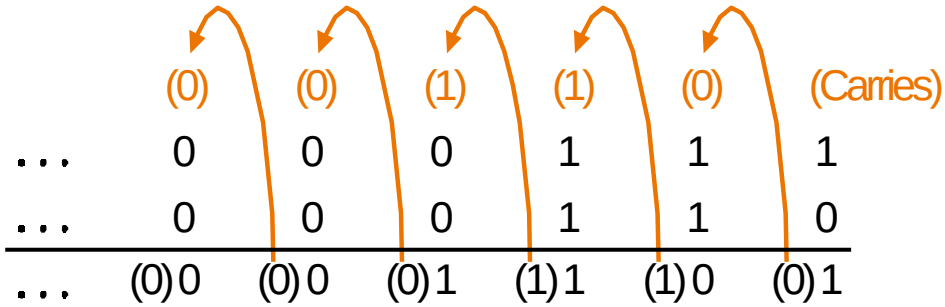
- Grouping every 4 binary digits is equivalent to converting to hexadecimal
- Example: $(1110\ 1100\ 1010\ 1000\ 0110\ 0100\ 0010\ 0000)_2 = (ECA8\ 6420)_{16}$

Addition and Subtraction

- Digits are added bit by bit from right to left, with carries passed to the next digit to the left

- *Example:*

```
  0000 0000 0000 0000 0000 0111 =  7
+ 0000 0000 0000 0000 0000 0110 =  6
  ---------------------------------------
  0000 0000 0000 0000 0000 1101 = 13
```



- Subtraction uses addition: the appropriate operand is simply negated

- *Example:*

```
  0000 0000 0000 0000 0000 0000 0000 0111 =  7
- 0000 0000 0000 0000 0000 0000 0000 0110 =  6
  -------------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0001 =  1
```

Or using two's complement arithmetic

```
  0000 0000 0000 0000 0000 0000 0000 0111 =  7
+ 1111 1111 1111 1111 1111 1111 1111 1010 = -6
  -------------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0001 =  1
```

Arithmetic Overflow

- Overflow occurs when the result of an operation cannot be represented with the available hardware
- Most hardware detects and signals overflow via an exception
- Some high level languages ignore overflow (e.g. C) and some check for and handle it (e.g. Ada and Fortran)
- **Overflow conditions**

If there is either carry-in or carry-out (not both) for the sign bit

Operations	Operand *A*	Operand *B*	Result
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

- Example:

Assuming 4 bits 2's complement numbers, the maximum positive number is 7 and the least number is -8. Adding the numbers 6 and 5 should lead to overflow and similarly for -6 and-5.

0110 + 0101 = 1 011, 1010 + 1001 = 10011

Logical Operations

- Although words are the basic blocks for most computers, it is often needed to operate on fields of bits within a word (check for a character)
- Logical operations are useful for bit-wise handling of words
- AND, OR and shift operations are the most famous supported operations by instruction set architectures
- Shift operations are either right (divide), filling with the sign bit or left (multiply), filling in with zeros

Examples:

$(0000\ 0010)_2 << 2 \rightarrow (0000\ 1000)_2$

$(1111\ 1110)_2 << 2 \rightarrow (1111\ 1000)_2$

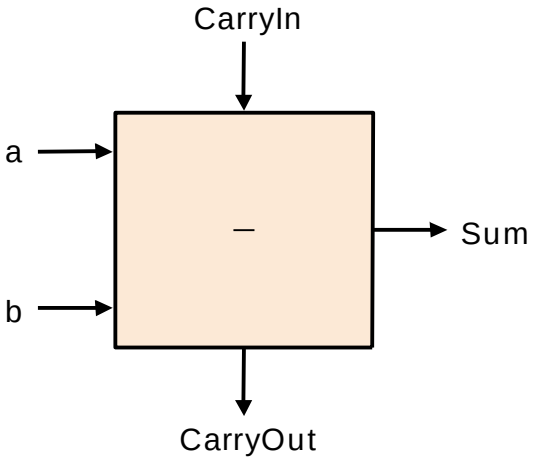
$(0000\ 0010)_2 >> 1 \rightarrow (0000\ 0001)_2$

$(1111\ 1110)_2 >> 1 \rightarrow (1111\ 1111)_2$

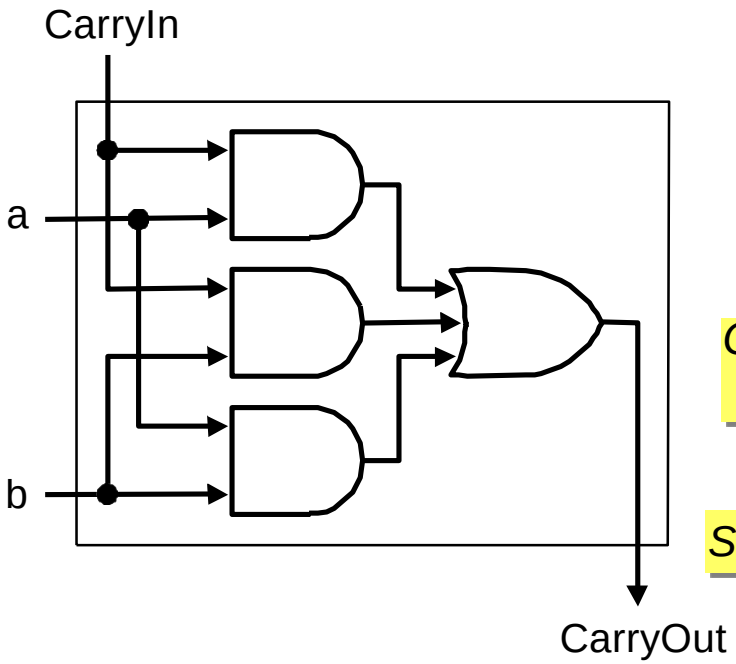
- AND and OR operations are often used to isolate and augment words with certain field of bits

Logical operations can miss up signed numbers ➔ should be carefully used

A 1-Bit Arithmetic Unit



Inputs			Outputs	
a	b	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

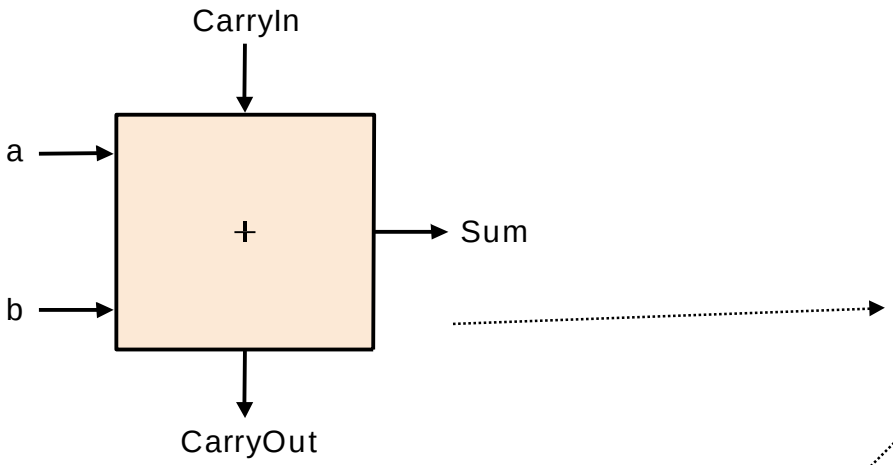


A single bit adder has 3 inputs, two operands and a carry-in and generates a sum bit and a carry-out to be passed to the next 1-bit adder

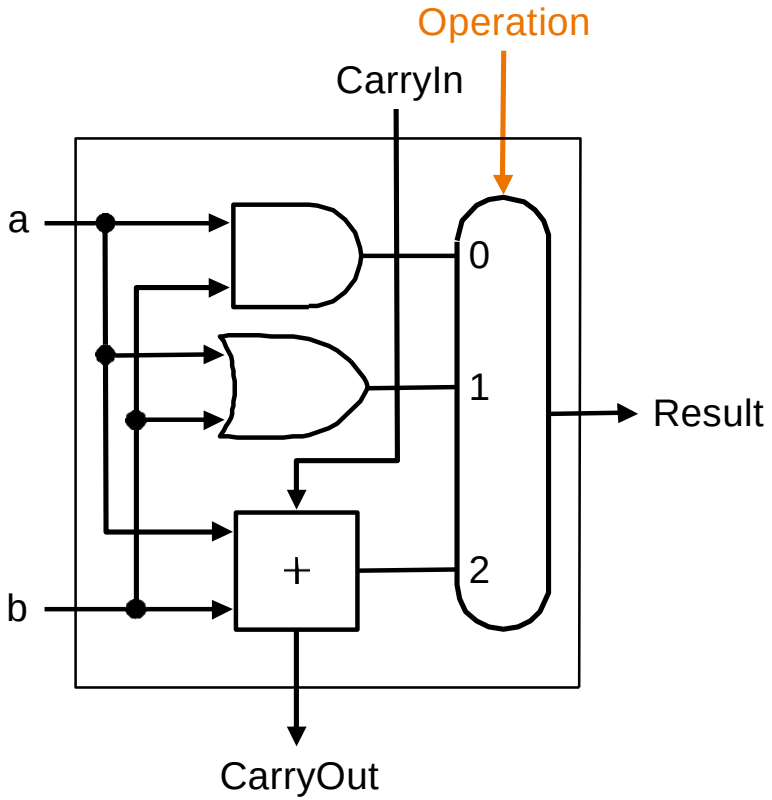
$$CarryOut = (b.CarryIn) + (a.CarryIn) + (a.b) + (a.b.CarryIn)$$
$$= (b.CarryIn) + (a.CarryIn) + (a.b)$$

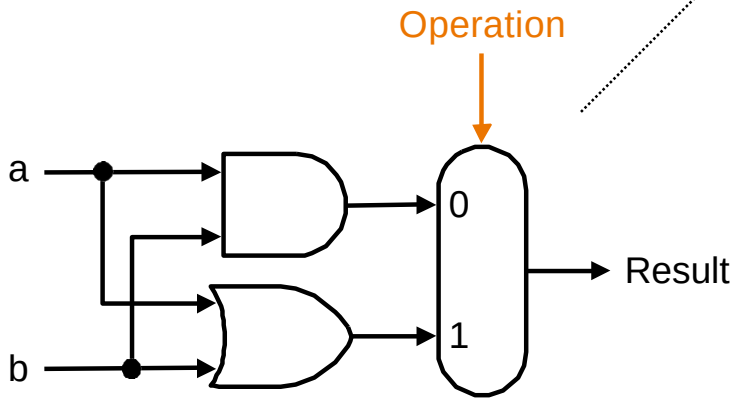
$$Sum = (a.\overline{b}.\overline{CarryIn}) + (\overline{a}.b.\overline{CarryIn}) + (\overline{a}.\overline{b}.CarryIn) + (a.b.CarryIn)$$

A 1-Bit ALU



1-Bit adder





1-Bit logical unit

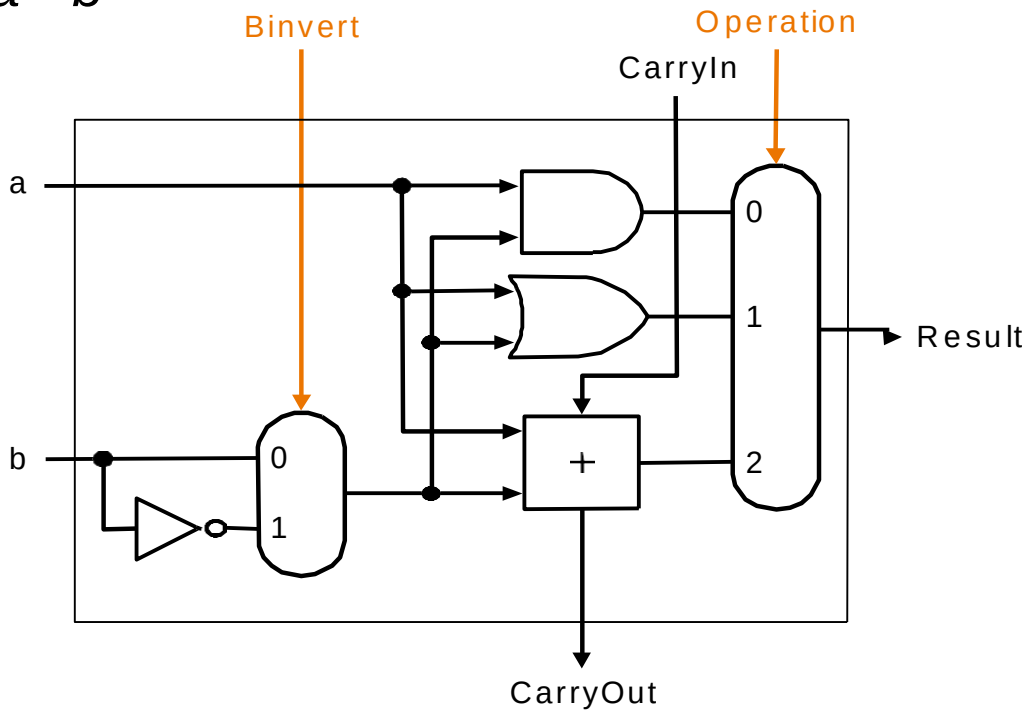
- The multiplexor selects either *a* AND *b*, *a* OR *b or a* + *b* depending on whether the value of *operation* is *0*, *1, or 2*
- To add an operation, the multiplexor has to be expanded & a circuit for performing the operation has to be appended

Supporting Subtraction

- Subtraction can be performed by inverting the operand and setting the "CarryIn" input for the adder to 1 (i.e. using two's complement)
- By adding a multiplexor to the second operand, we can select either b or $\bar{b}$
- The *Binvert* line indicates a subtraction operation and causes the two's complement of b to be used as an input

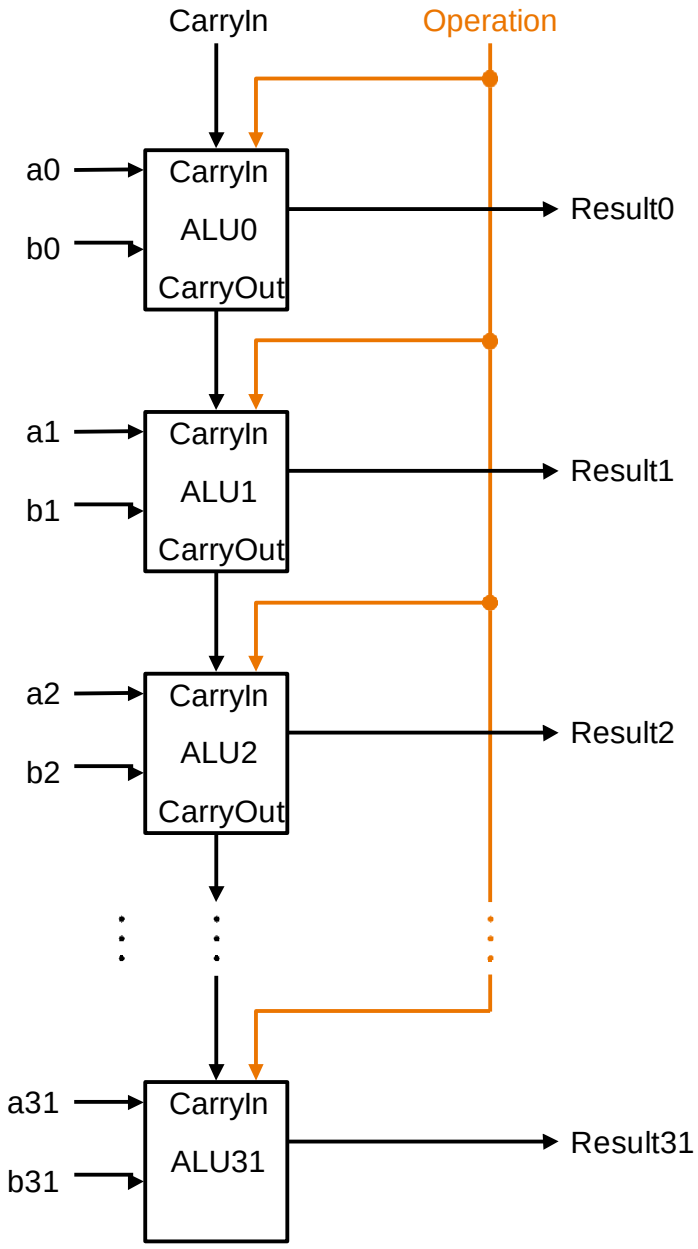
$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

The simplicity of the hardware design of a two's complement adder explains why it is a universal standard for computer arithmetic

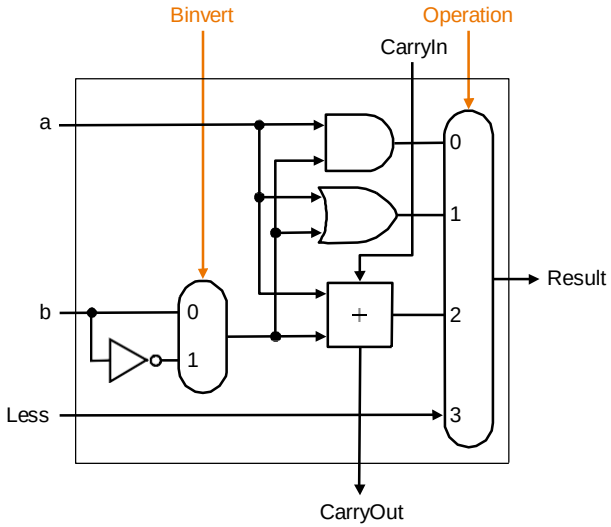


A 32-Bit ALU

- A full 32-bit ALU can be created by connecting adjacent 1-bit ALU's using the Carry in and carry out lines
- The carry out of the least significant bit can ripple all the way through the adder (*ripple carry adder*)
- Ripple carry adders are slow since the carry propagates from a unit to the next sequentially
- Subtraction can be performed by inverting the operand and setting the "CarryIn" input for the whole adder to 1 (i.e. using two's complement)



Supporting MIPS' "*slt*" instruction



Basic 1-Bit ALU

-"Less" input support the "slt instruction

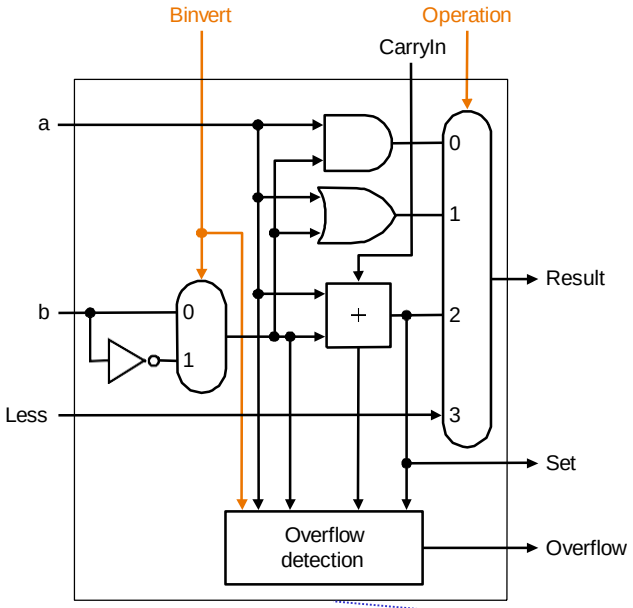
-"slt" produce 1 only if rs<rt and 0 otherwise

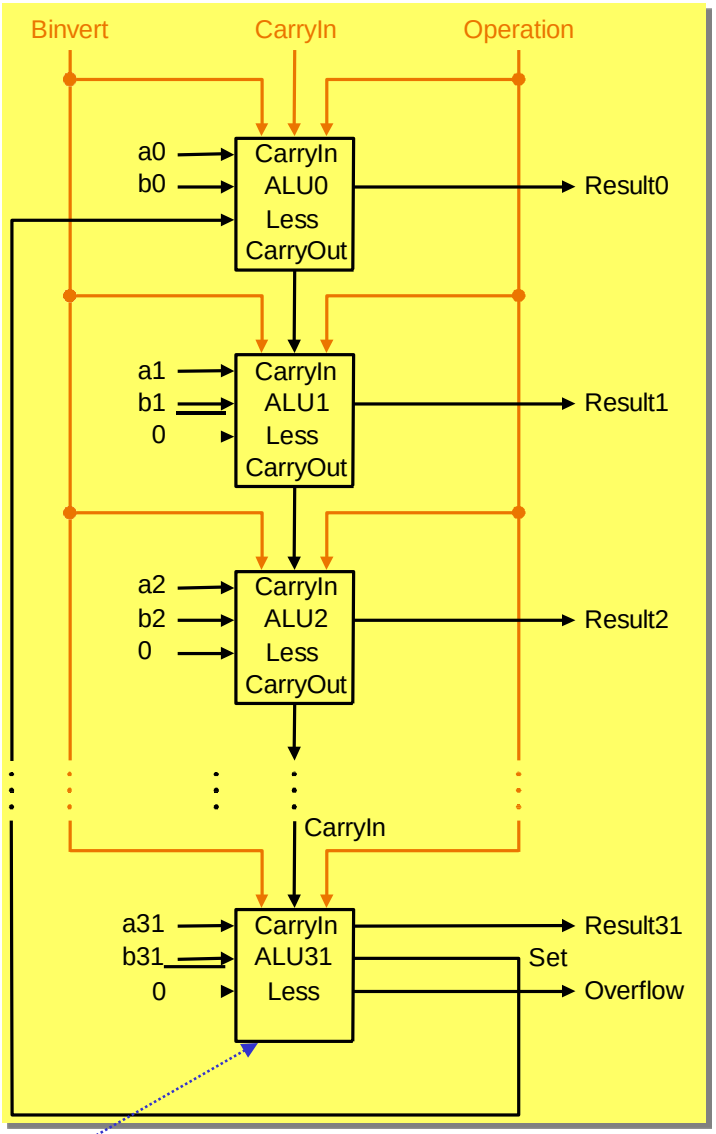
Most Significant Bit

-a < b iff (a-b) < 0 (value of sign bit is 1)

-Checks for overflow

- Sets the least significant bit to the value of sign bit

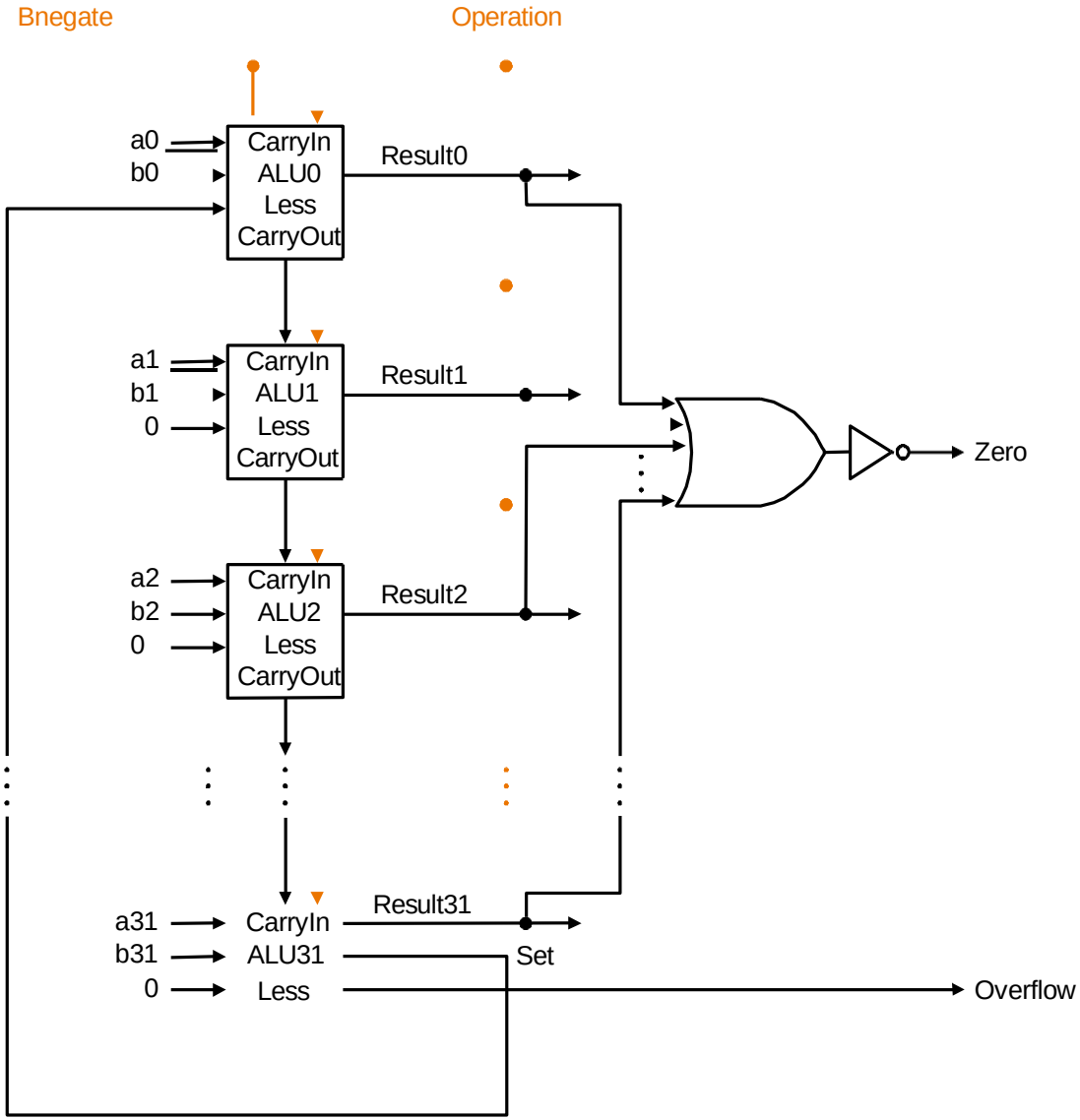




Most Significant Bit

32-Bit Basic MIPS ALU

MIPS' ALU



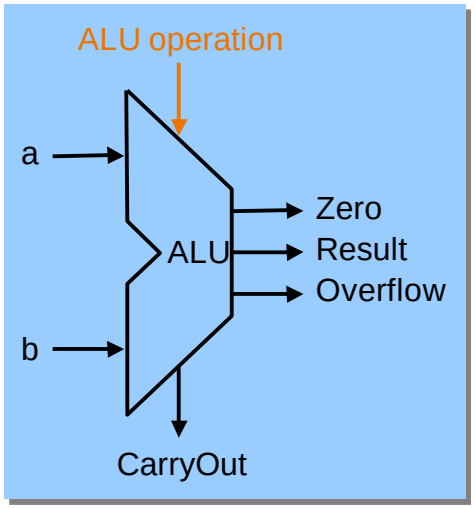
MIPS' ALU Circuits

Conditional Branching

-"bne" and "beq" instruction compares two operands for equality

-a = b iff (a-b) = 0

- *Zero* signal indicates all zero results



ALU Symbol

Optimizing Adder's Design

Ripple Carry Adders

- The CarryIn input depends on the operation in the adjacent 1-bit adder
- The result of adding most significant bits is only available after all other bits, i.e. after *n-1* single-bit additions
- The sequential chain reaction is too slow to be used in time-critical hardware

Carry Lookahead

- Anticipate the value of the carry ahead of time
- Worst-case scenario is a function of $\log_2 n$ (the number of bits in the adder)
- It takes many more gates to anticipate the carry

Fast Carry Using "Infinite" Hardware

Using the equation:

$$C_{Out} = (b.C_{in}) + (a.C_{in}) + (a.b)$$

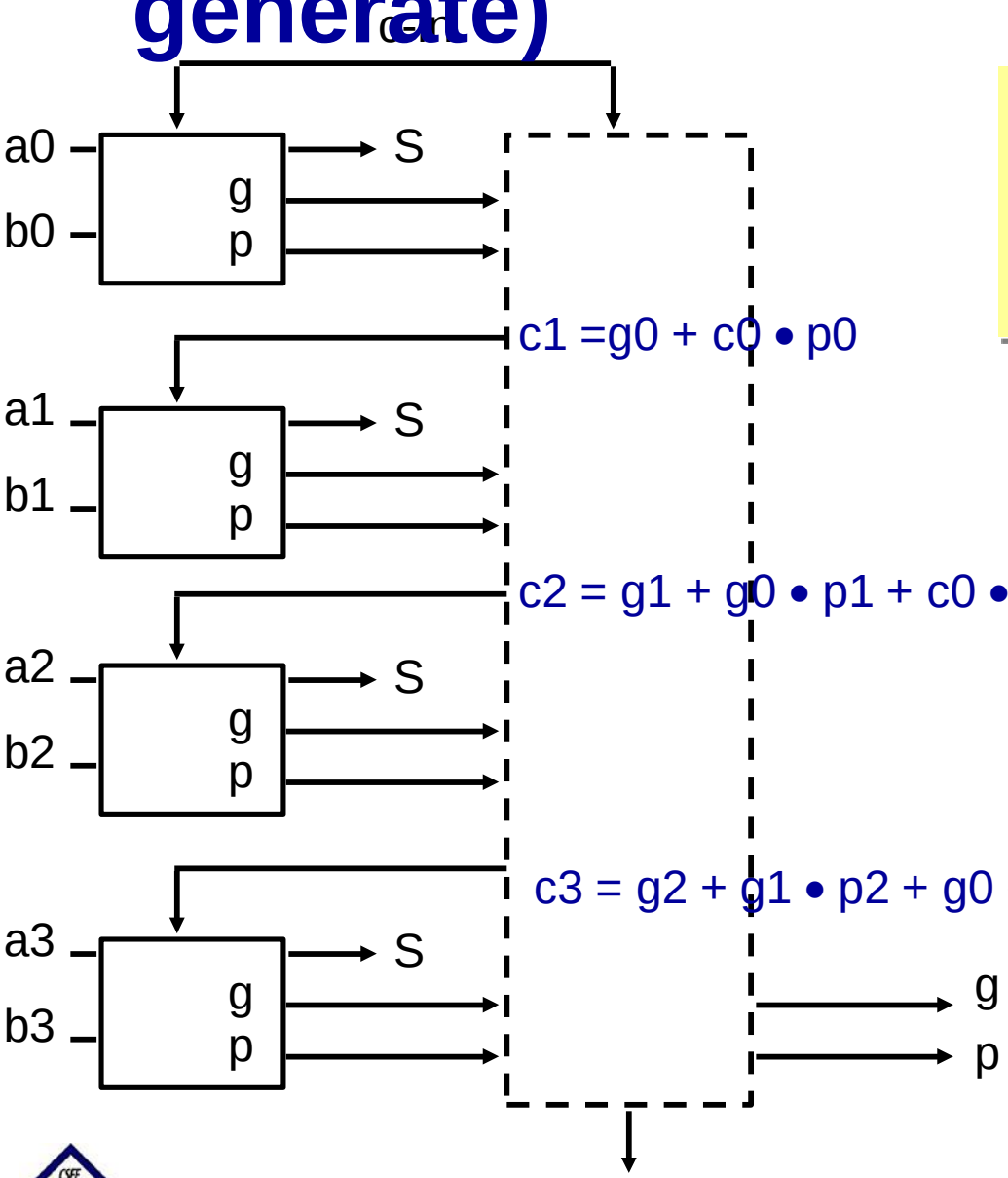
c2 = (b1 . c1) + (a1 . c1) + (a1 . b1)
c1 = (b0 . c0) + (a0 . c0) + (a0 . b0)

Substituting the definition of c1 in c2 equation

c2 = (a1 . a0 . b0) + (a1 . a0 . c0) + (a1 . b0 . c0) + (b1 . a0 . b0) +
(b1 . a0 . c0) + (b1 . b0 . c0) + (a1. b1)

➔ Number of gates grows exponentially when getting to higher bits in the adder

Carry Lookahead (propagate & generate)



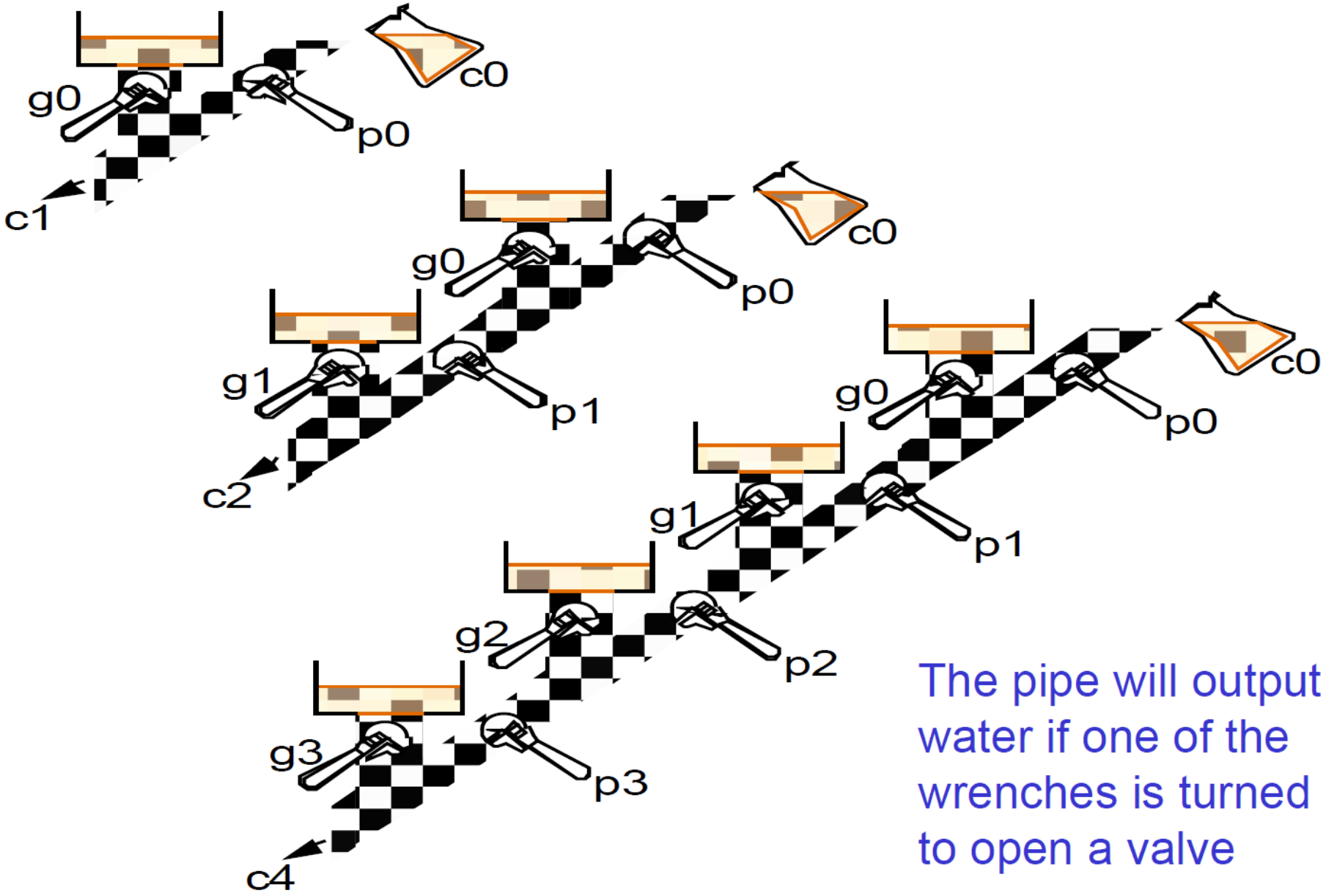
$$c_{i+1} = (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i)$$
$$= (a_i \cdot b_i) + c_i \cdot (a_i + b_i)$$
$$= g_i + c_i \cdot p_i$$

a	b	c-out	
0	0	0	"kill"
1	1	c-in	"propagate"
2	0	c-in	"propagate"
1	1	1	"generate"

p = a or b
g = a and b

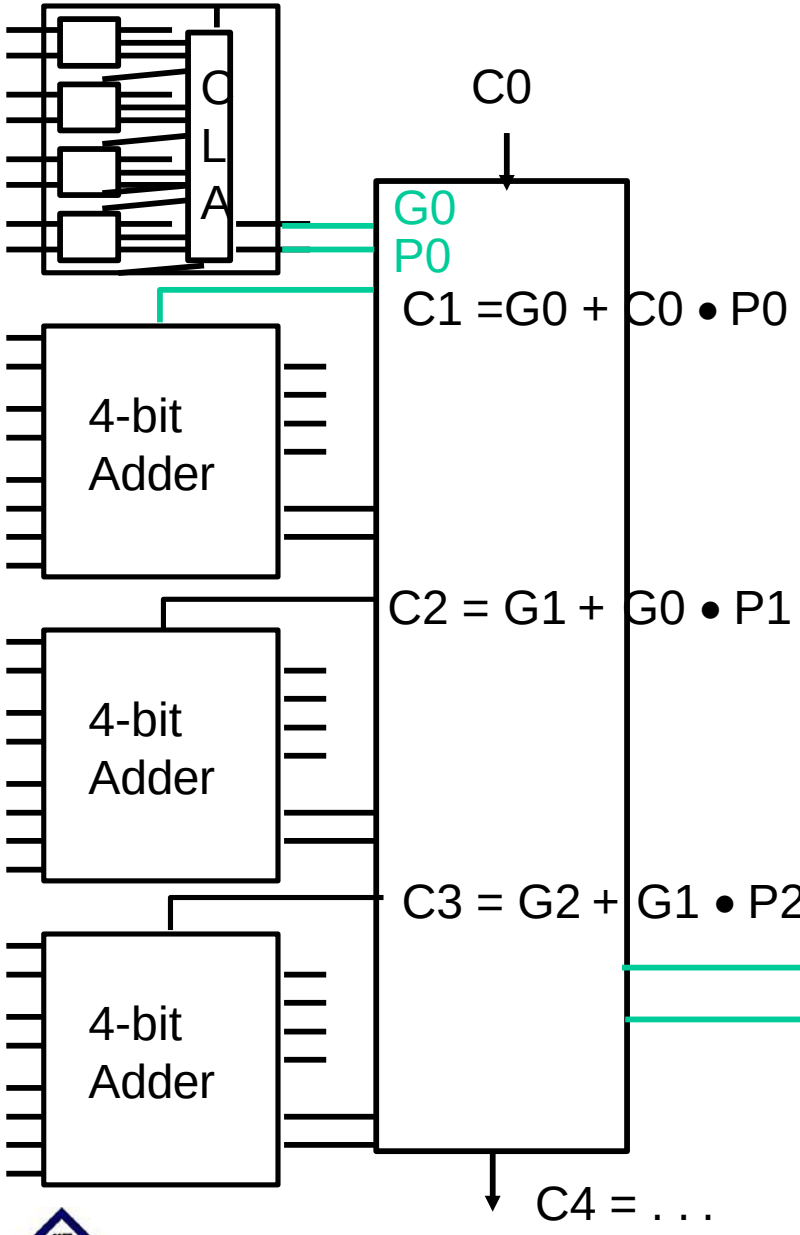
* Figure is courtesy of Dave Patterson

Plumbing as Carry Lookahead Analogy



The pipe will output water if one of the wrenches is turned to open a valve

Cascaded Carry Look-ahead



- Consider a 4-bit adder with its carry lookahead logic as a building block
- Connect the 4-bit adders in ripple carry fashion
- Carry lookahead is done at a high level

$P_0 = p_3 . p_2 . p_1 . p_0$

$P_1 = p_7 . p_6 . p_5 . p_4$

….

$G_0 = g_3 + (p_3.g_2) + (p_3.p_2.g_1) + (p_3.p_2.p_1.g_0)$

$G_1 = g_7 + (p_7.g_6) + (p_7.p_6.g_5) + (p_7.p_6.p_5.g_4)$

….

An Example

Determine g_i, p_i, P_i, G_i and carry out (C_4) values for these two 16-bit numbers:

a: 0001 1010 0011 0011

b: 1110 0101 1110 1011

Answer: Using the formula $g_i = (a_i . b_i)$ *and* $p_i = (a_i + b_i)$

g_i: 0000 0000 0010 0011

p_i: 1111 1111 1111 1011

The "super" propagates (P_0, P_1, P_2, P_3) are calculated as follows:

$P_0 = p_3 . p_2 . p_1 . p_0 = 0$ $P_1 = p_7 . p_6 . p_5 . p_4 = 1$

$P_2 = p_{11} . p_{10} . p_9 . p_8 = 1$ $P_3 = p_{15} . p_{14} . p_{13} . p_{12} = 1$

The "super" generates (G_0, G_1, G_2, G_3) are calculated as follows:

$G_0 = g_3 + (p_3.g_2) + (p_3.p_2.g_1) + (p_3.p_2.p_1.g_0) = 0$

$G_1 = g_7 + (p_7.g_6) + (p_7.p_6.g_5) + (p_7.p_6.p_5.g_4) = 1$

$G_2 = g_{11} + (p_{11}.g_{10}) + (p_{11}.p_{10}.g_9) + (p_{11}.p_{10}.p_9.g_8) = 0$ $G_3 =$

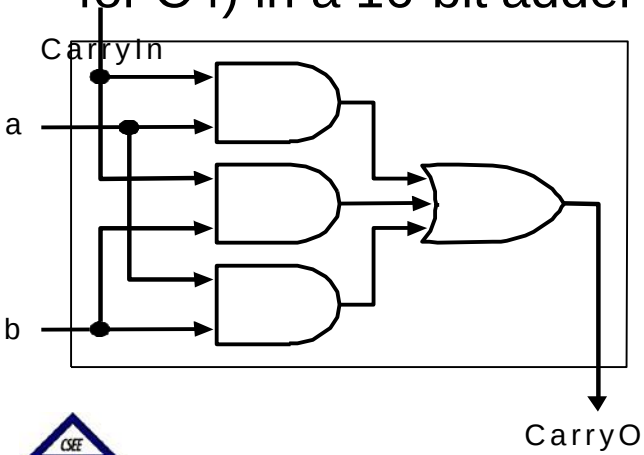
$g_{15} + (p_{15}.g_{14}) + (p_{15}.p_{14}.g_{13}) + (p_{15}.p_{14}.g_{13}.g_{12}) = 0$

Finally carry-out (C_4) *is*:

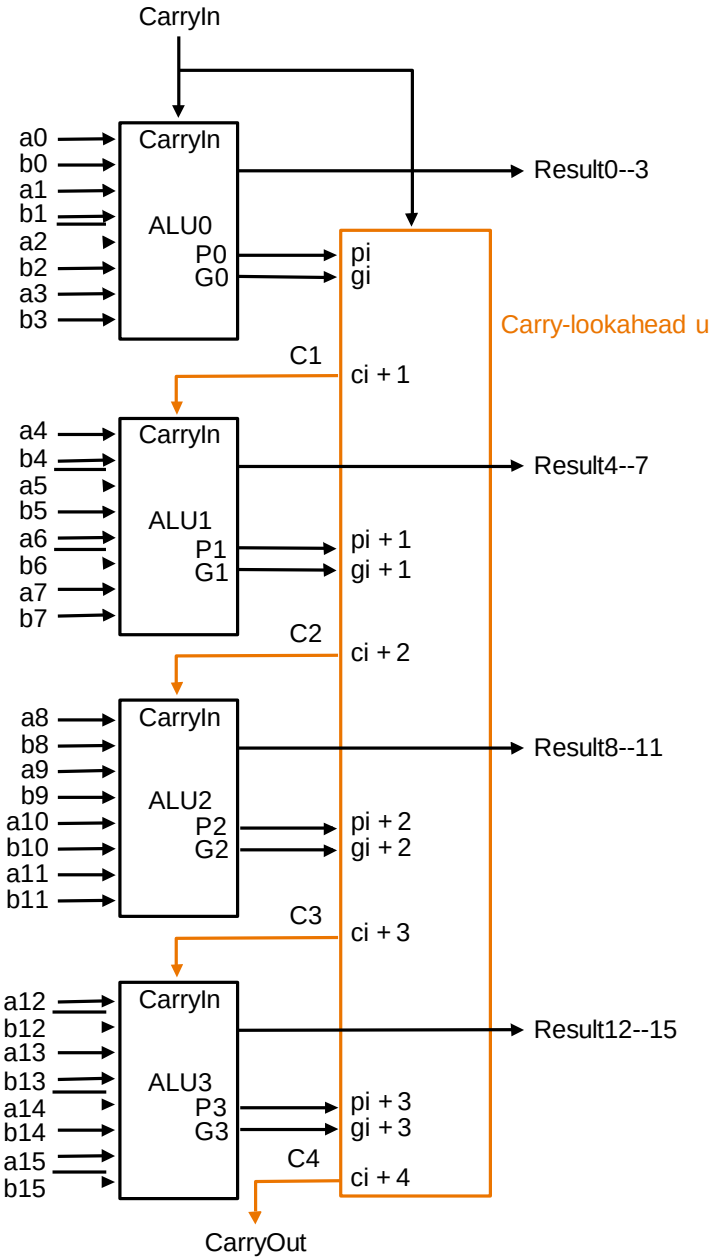
$C_4 = G_3 + (P_3.G_2) + (P_3.P_2.G_1) + (P_3.P_2.P_1.G_0) + (P_3.P_2.P_1.P_0.C_0) = 1$

Speed of Carry Generation

- There is a (gate) delay for an output to be ready once input signals are applied to a gate
- Time is estimated by simply counting the number of gates along the longest path
- Carry lookahead is faster because less cascaded levels of logic gates are used
- For a 16-bit ripple carry adder, carry-out is subject to 32 (16 × 2 for 1 -bit adder) gate
- Cascaded carry lookahead (C4) is delayed by only 5 gates (1 for *p* and *g*, 2 for G and 2 for C4) in a 16-bit adder



It takes two gates delay for carry-out to be available in a single bit adder



Conclusion

❑ Summary

➔ Constructing an Arithmetic Logic Unit

(Different blocks and gluing them together)

➔ Scaling bit operations to word sizes

(Ripple carry adder, MIPS ALU)

➔ Optimization for carry handling

(Measuring performance, Carry lookahead)

❑ Next Lecture

➔ Algorithms for multiplying unsigned numbers

➔ Booth's algorithm for signed number multiplication

➔ Multiple hardware design for integer multiplier

Read sections (B.1 – B.6) in 5rd Ed., or (3.1, C.5-C.6) in 4th Ed. Of the textbook