

Computer Organization and Design

Chapter 3 Part 1

Arithmetic for Computers

(Ch. 2 – Section 2.4; Ch. 3 – Sections 3.1 – 3.2)

(Additional material from Appendix A – Sections A.5 & A.6)

Review of Binary Numbers

- Integers are stored in registers and memory in values of 8-, 16-, 32- or 64-bit words (RISC-V)
 - Byte = 8 bits ($2^8 = 256$ possible values)
 - Halfword = 16 bits ($2^{16} = 65536$ possible values)
 - Word = 32 bits ($2^{32} = 4,294,967,296$ possible values)
 - Doubleword = 64 bits ($2^{64} = \sim 18 \times 10^{18}$ possible values)
- Both positive and negative values must be represented
 - Two's complement for integers is used in all computers
 - Must also be able to interpret and use unsigned integers

Signed vs. Unsigned Integers

Byte: -128 to + 127 (signed)
 0 to 255 (unsigned)

Halfword: - 32,768 to + 32,767 (signed)
 0 to 65,535 (unsigned)

Word: - 2,147,483,648 to + 2,147,483,647 (s)
 0 to 4,294,967,295 (u)

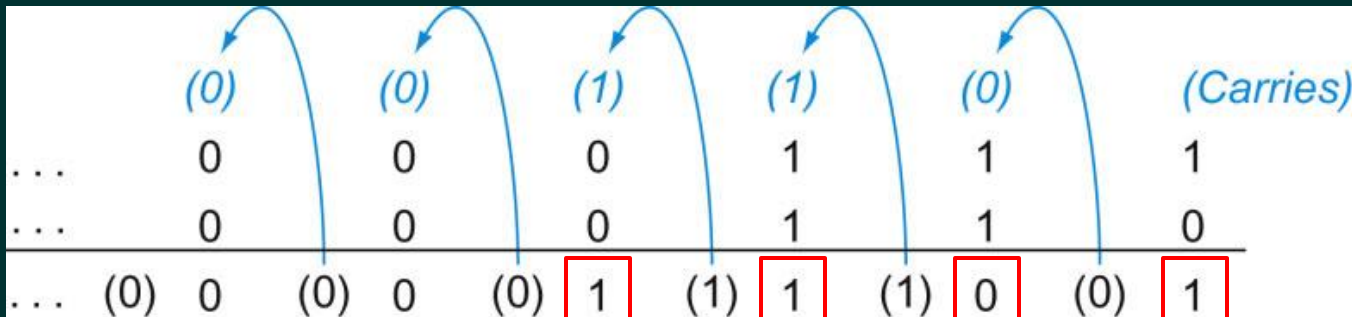
Binary Addition

- 1-bit addition
- Always 2 bits generated
 - Sum
 - Carry

0	0
0	1
<hr/>	
0	0
(c	s)

1	1
0	1
<hr/>	
0	1
(c	s)


- Example: $7 + 6 = 13$



Binary Subtraction

- Complement one of the operands, then add
- Subtraction is converted to an add operation in the hardware

Example: $7 - 6 = 1$



(Carries)	1	1	1	1	1	1	1	0	
	0	0	0	0	0	1	1	1	= + 7
+	1	1	1	1	1	0	1	0	= - 6
<hr/>									
	0	0	0	0	0	0	0	1	= + 1

During the arithmetic operation, the two's complement of 6 is produced converting it from positive to negative, after which the addition operation is performed producing the correct result of +1.

2's Complement Arithmetic

Summary

Adding positives

+69	01000101
+ +12	00001100
----	-----
+81	01010001

Adding positive and negative

+69	01000101
+ -12	11110100
----	-----
+57	00111001

note a carry bit was generated and lost but still got right answer

Subtract positives

+69	01000101
- +12	11110100
----	-----
+57	00111001

becomes same
operation as

complement,
then add.

Adding negatives

-69	10111011
+ -12	11110100
----	-----
-81	10101111

same carry
bit scenario

Subtracting negatives

-69	10111011
- -12	11110100
----	-----
-57	11000111

becomes same
operation as

complement,
then add.

Subtraction is converted to addition by complimenting the second operand then adding.

Examples

$$\begin{array}{r} 19 \\ + 61 \\ \hline 80 \end{array} \quad \begin{array}{r} 111111 \\ 00010011 \\ + 00111101 \\ \hline 01010000 \end{array}$$

$$\begin{array}{r} - 26 \\ + 14 \\ \hline - 12 \end{array} \quad \begin{array}{r} 0001110 \\ 11100110 \\ + 00001110 \\ \hline 11110100 \end{array}$$

$$\begin{array}{r} 120 \\ - 29 \\ \hline 91 \end{array} \quad \begin{array}{r} \textcircled{1} 1100000 \\ 01111000 \\ + 11100011 \\ \hline 01011011 \end{array}$$

$$\begin{array}{r} - 54 \\ - 15 \\ \hline - 69 \end{array} \quad \begin{array}{r} \textcircled{1} 1000000 \\ 11001010 \\ + 11110001 \\ \hline 10111011 \end{array}$$

Positive vs. Negative Integers

- There is a distinction in most architectures that allow you to deal with numbers that can be positive or negative and numbers that can only be positive
 - When would you only want a number to be positive?
- 2's complement is the representation for *signed integers*
- Numbers that must be treated always as positive represent *unsigned integers*
- This is important when comparing and performing arithmetic on numeric values

Signed vs. Unsigned Integers

- RISC-V provides unsigned versions of some instructions that are used when working with unsigned numbers

Loads: lbu – load byte unsigned
 lhu – load halfword unsigned
 lwu – load word unsigned

Branches: bgeu – branch if greater than or equal unsigned
 bltu – branch if less than unsigned

Compares: sltu – set if less than unsigned
 sltiu – set if less than immediate unsigned

Comparing Signed vs. Unsigned

- The 8-bit binary value for -1 = 11111111
- The 8-bit binary value for +1 = 00000001
- The binary value 11111111 = 255 if interpreted as unsigned
- The value +1 is interpreted as +1 for both signed and unsigned interpretations
- 11111111 < 00000001 for signed comparisons
- 00000001 < 11111111 for unsigned comparisons

Arithmetic Limitations

- Consider the following 2's complement example:

$$\begin{array}{r} 0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 \\ + 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 \\ \hline 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \end{array}$$

$$\begin{array}{r} 2,147,483,647 \\ + \quad \quad \quad 1 \\ \hline - 2,147,483,648 \end{array} \quad \begin{array}{l} \text{decimal} \\ \text{equivalent} \end{array}$$

- The first bit of the result is a 1 which indicates a negative number in 2's complement
- Result is incorrect for signed arithmetic!
 - Adding two positive integers should never produce a negative result

Numeric Overflow

- Two n -bit numbers added together yield an $n+1$ -bit result
 - more bits than are available are needed
 - In 64-bit architectures, it is possible to perform an operation that results in a value that needs 65 bits to store it.
- This condition is called *overflow*.
 - Note that overflow does not necessarily mean a carry overflowed
 - It is possible to generate a carry bit from the MSB position and still get a correct result as we saw in previous examples

Overflow

- There are four conditions that result in overflow

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

- Each of the four cases described above affects the sign bit
- Detecting overflow is often implemented in hardware by checking the sign bit of the result and the signs of the input values and generating an exception if overflow is detected

Overflow (continued)

- Overflow is a condition that must be detected and some action taken to prevent an incorrect value from affecting further arithmetic operations
- Computer designer or programmer (compiler) must provide a way to detect overflow
- Detection of overflow should generate an *exception*
 - An exception is the interruption to the normal execution sequence of instructions in a program
 - Exceptions allow proper handling of the overflow either by the program or operating system
 - When an exception occurs, program execution is paused until the exception is processed.

Overflow (continued)

- It is common in many RISC architectures to not implement hardware support for arithmetic overflow checking.
 - RISC designs emphasize simplicity and speed, opting for fewer instructions and a streamlined pipeline.
 - Instead of hardware overflow checks, programmers can handle overflow in software when necessary.
 - Some architectures provide instructions that can help detect overflow, but these are not universal across all RISC designs.
 - The decision to omit overflow checking in hardware allows for faster execution of instructions and reduces the complexity of the hardware.

Overflow (continued)

- RISC-V operations do not provide a means to check for overflow.
- Instead, it is recommended to include branch instructions immediately after arithmetic operations that potentially may produce an overflow condition.
- For unsigned addition, overflow handling can be done with a single bltu instruction.

```
add    t0, t1, t2  
bltu   t0, t1, overflow
```

- The branch target is a subroutine to handle the overflow condition.

Overflow (continued)

- For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands, if and only if the other operand is negative.

```
add    t0, t1, t2
slti   t3, t2, 0
slt    t4, t0, t1
bne    t3, t4, overflow
```

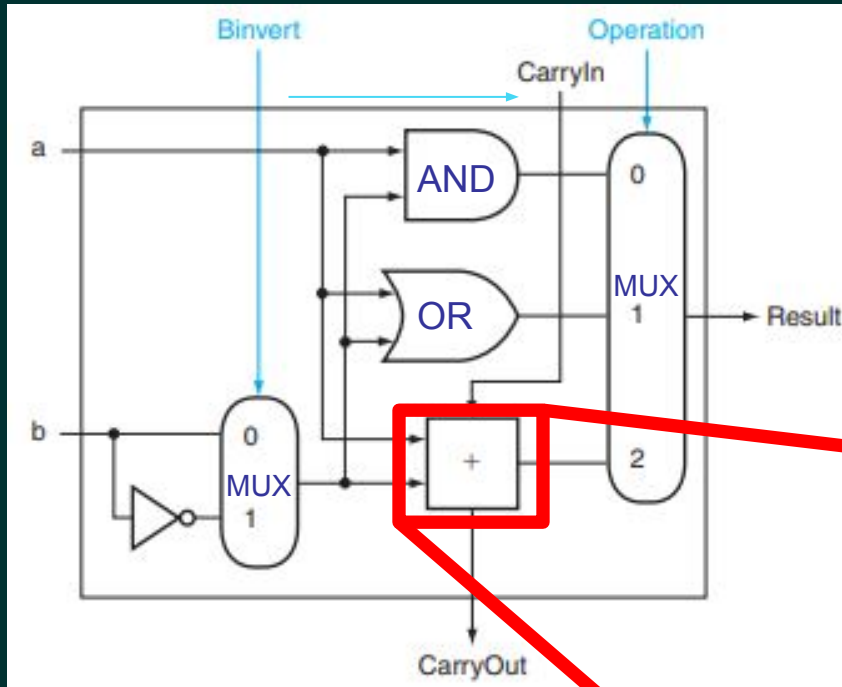
Arithmetic and Logic Unit

(also known as the Integer Unit)

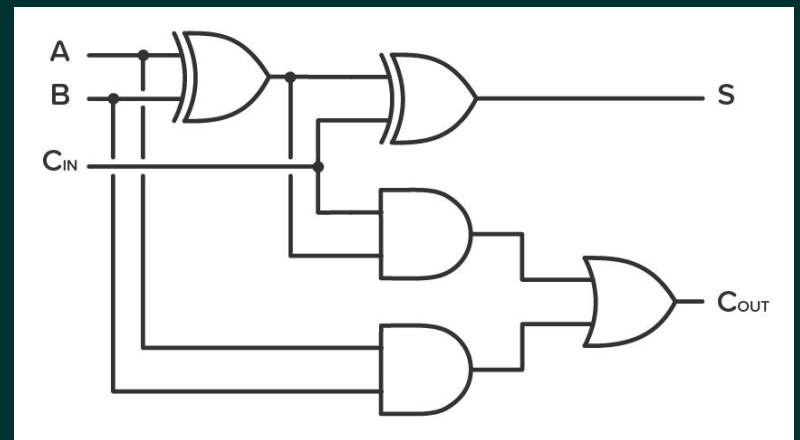
- ALU is the part of the CPU that performs integer arithmetic and logic functions
 - Add, subtract, multiply, divide
 - And, or, not, shifting
- ALU is designed to work with data values based on word size
 - Maximum number of bits in largest allowable numeric type
- ALU implementations vary
 - There are many configurations of circuits to perform arithmetic

A Simple 1-bit ALU

- Implements add, subtract, and, or

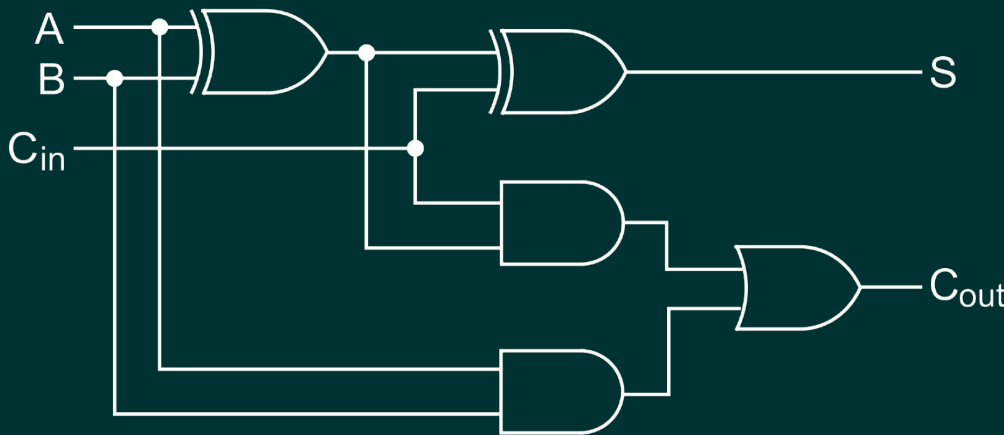


Full Adder Circuit



Full Adder

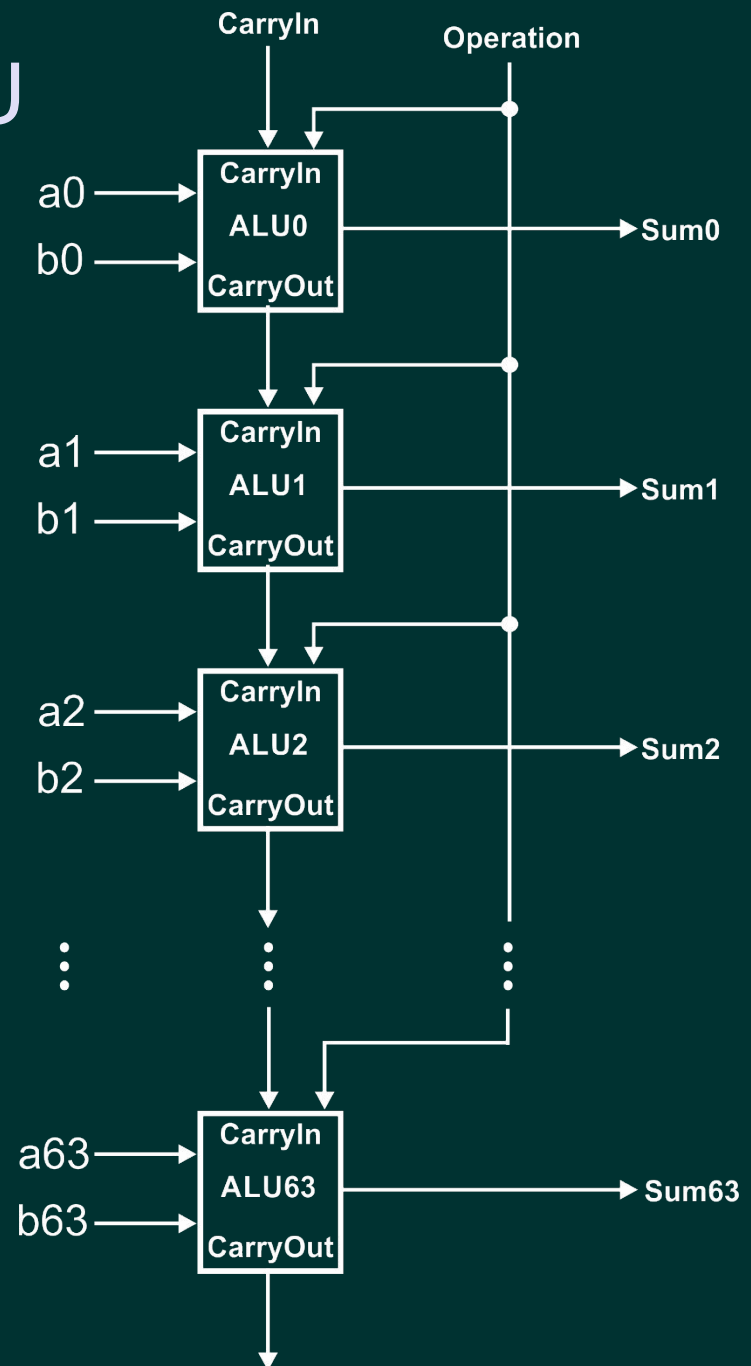
- Full adder has three inputs
 - A, B, C_{in}
- Two outputs
 - S, C_{out}



Truth Table				
Inputs			Outputs	
A	B	C _{in}	S	C _{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

64-bit ALU

- We can create a 64-bit ALU by connecting 64 adjacent 1-bit ALUs such that
 - The CarryOut of each 1-bit ALU is connected to the CarryIn of the next adjacent ALU in series
 - The operation input is connected to every 1-bit ALU in parallel

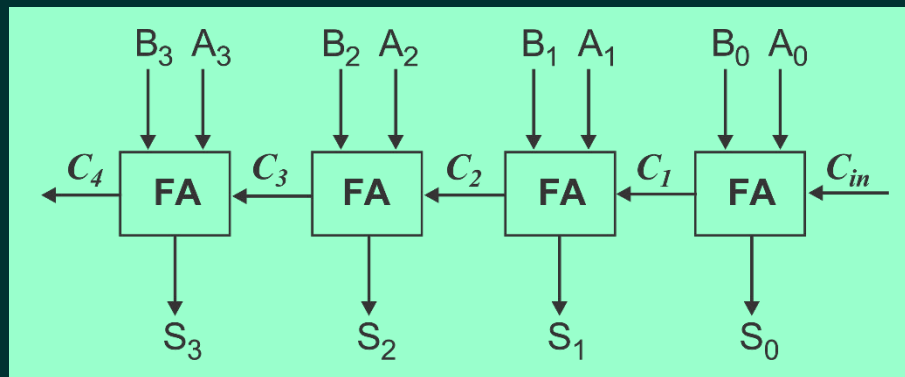


Ripple-Carry Adder

- The design of the ALU on the previous page is called a ripple-carry adder
- The carry out of the least significant bit will ripple all the way through the ALU
- This organization does not result in the fastest ALU design due to a time penalty called propagation delay
- Each 1-bit circuit can not perform the operation on the two input operands until it has received the carry out of the previous circuit in the chain

Propagation of Carries

- Propagation delay slows down the operation of the 64-bit ripple-carry ALU
- The key to making a faster ALU is to do things in parallel (as opposed to the serial nature of the ripple-carry adder)
- Since the generation of the carry is the slow part, it would be good to find a way to generate the carry bits faster – in parallel with the sum



Carry-Lookahead Adder

- Carry-lookahead adder includes a circuit to generate all the carries in an arithmetic operation
 - Alters (expands) the full adder circuit
- The carries are generated directly from the operand inputs to the ALU
- Each stage of the adder has two additional outputs
 - Generate (g_i) – represents logical AND of inputs
 - Propagate (p_i) – represents logical OR of inputs
- These outputs are applied as inputs to the carry-lookahead circuit

1-Bit CLA

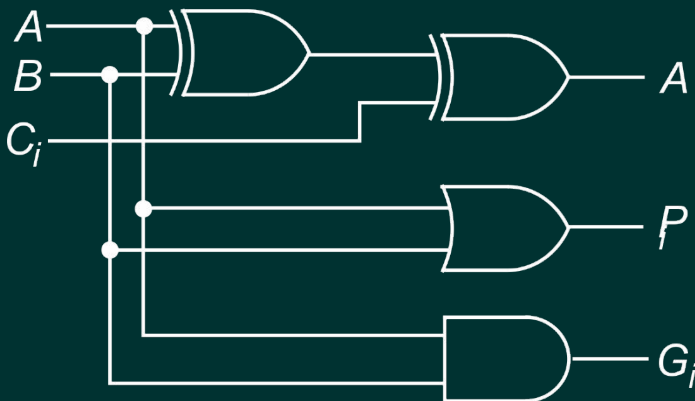
- Definitions:

- Generate: a bit generates a carry if both bits are 1

$$G_i = A_i \cdot B_i$$

- Propagate: a bit propagates a carry if at least one of the bits is 1

$$P_i = A_i + B_i$$



A	B	C_{in}	C_o	Type of Carry
0	0	0	0	None
0	0	1	0	None
0	1	0	0	None
0	1	1	1	Propagate
1	0	0	0	None
1	0	1	1	Propagate
1	1	0	1	Generate
1	1	1	1	Generate/Propagate

Carry-Lookahead Adder (continued)

- A pure carry-lookahead adder has n stages, where n is the number of bits in the operand
 - A pure carry-lookahead becomes very complex and costly to fabricate due to the recursive nature of the logical operations performed
- In real systems, a hybrid circuit is implemented such that the carry-lookahead circuits are organized in groups of either 4 or 8 bits.
 - Ripple-carry effect is present within the group but sufficient gains in speed offset the slightly higher implementation costs
- Variations of CLA have been developed

Derivation Logic: 4-Bit CLA (1)

- Given two input values A ($A_3A_2A_1A_0$) and B ($B_3B_2B_1B_0$), the carry outputs can be derived as follows:
 - Carry for bit 0: $C_1 = G_0 + P_0 \cdot C_0$
 - Carry for bit 1: $C_2 = G_1 + P_1 \cdot C_1$
 - Carry for bit 2: $C_3 = G_2 + P_2 \cdot C_2$
 - Carry for bit 3: $C_4 = G_3 + P_3 \cdot C_3$

Derivation Logic: 4-Bit CLA (2)

- To eliminate the dependency on the previous carry, express all carries in terms of the initial carry C_0 :

- Substitute C_1 into C_2 : $C_2 = G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0)$

- Substitute C_2 into C_3 :

$$C_3 = G_2 + P_2 \cdot (G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0))$$

- Substitute C_3 into C_4 :

$$C_4 = G_3 + P_3 \cdot (G_2 + P_2 \cdot (G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0)))$$

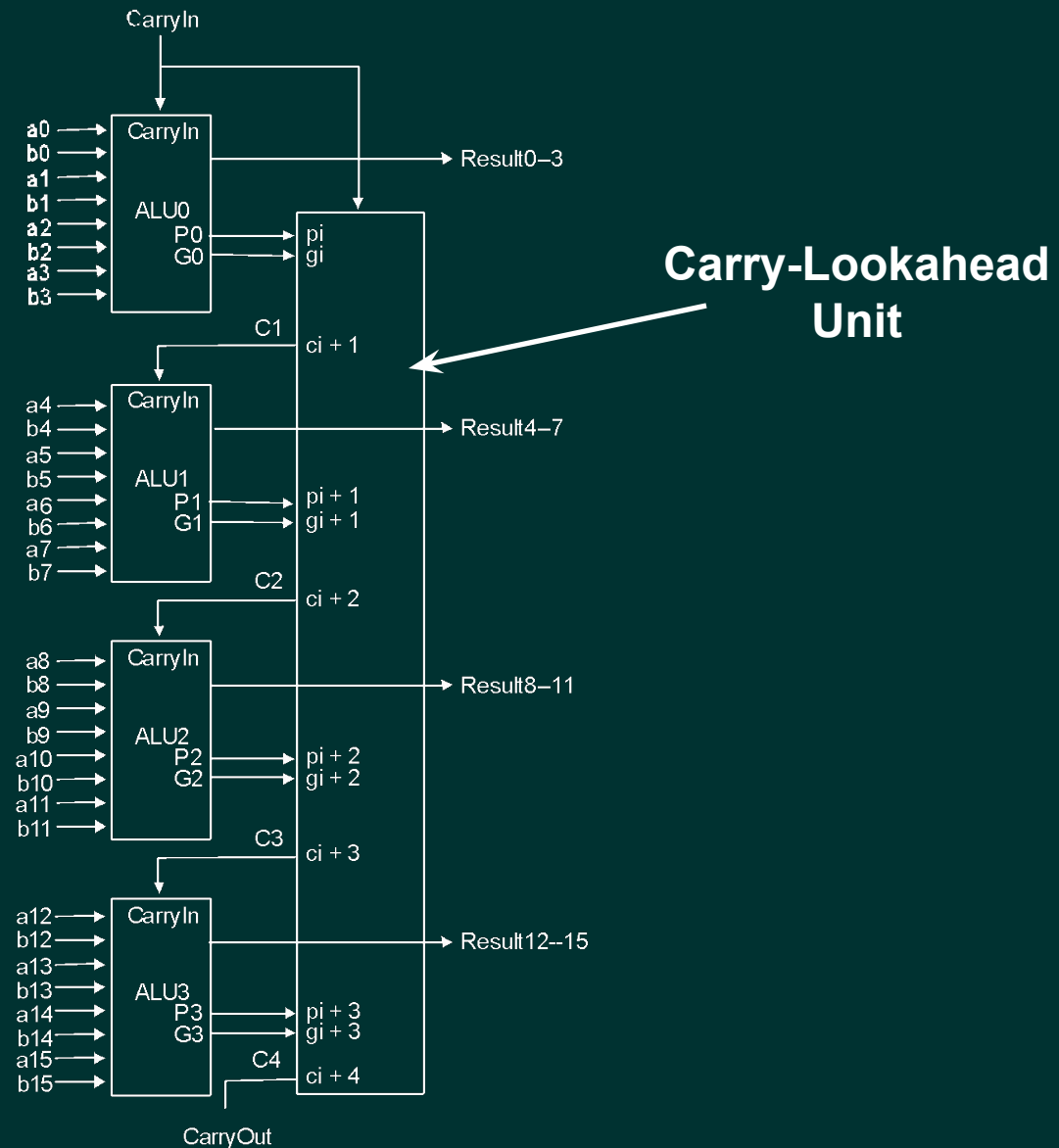
Derivation Logic: 4-Bit CLA (3)

- Simplify the previous equations for implementation:

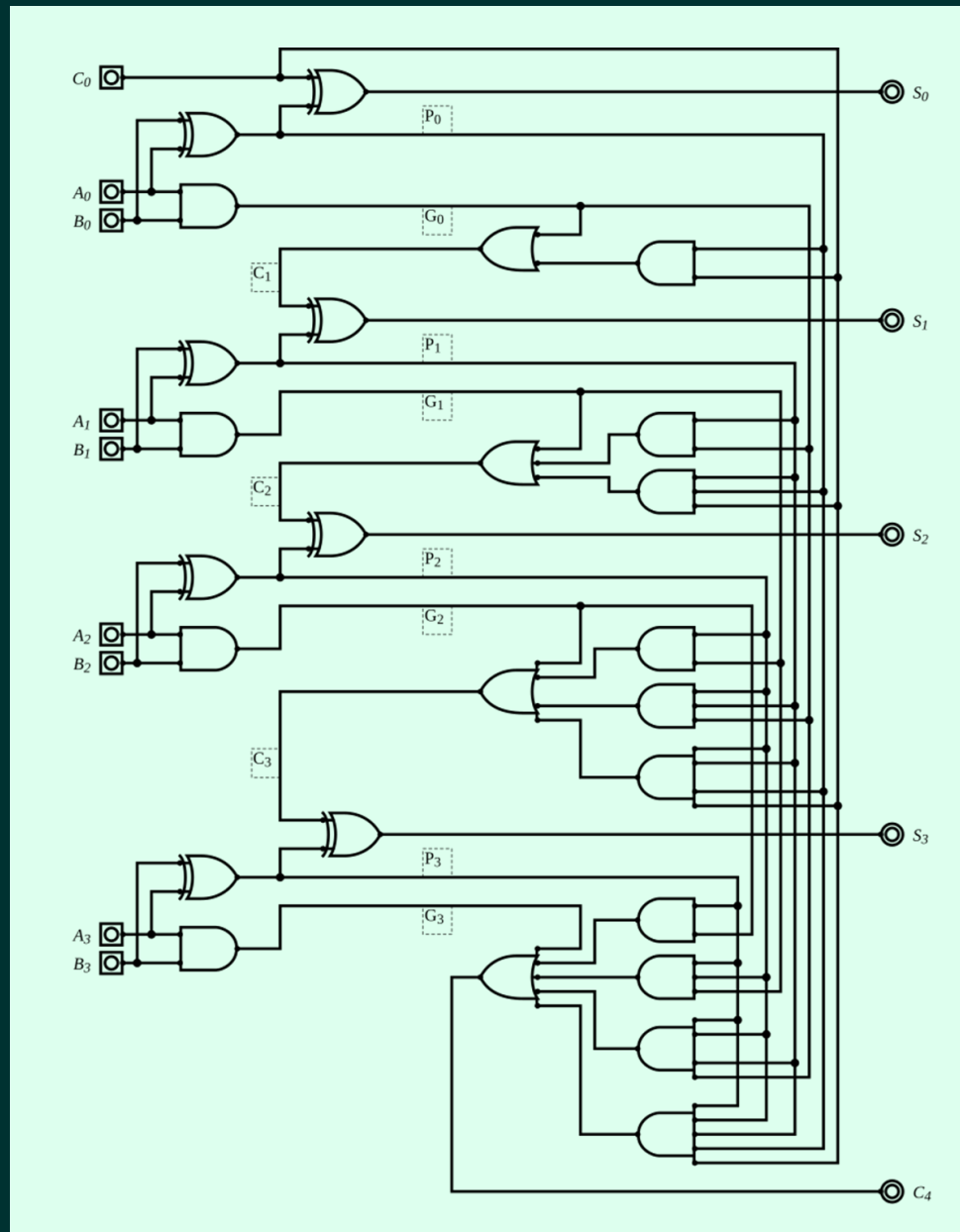
$$C_4 = G_3 + P_3 \cdot G_2 + P_2 \cdot P_2 \cdot G_1 + \\ P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

Carry-Lookahead Adder (continued)

16-bit adder with
carry-lookahead
with 4-bit groups

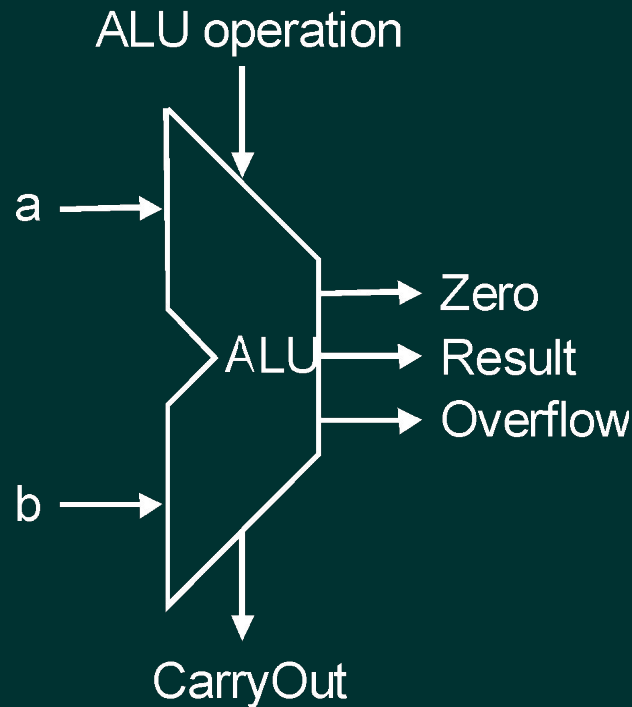


4-Bit CLA Schematic Diagram



ALU Symbol

- ALU is commonly represented in diagrams by a symbol



- This symbol is also used to represent an adder
- Symbols will be labeled appropriately