# Design of a 68-bit Carry Look-Ahead Adder

Mohit Sharma
110031631

## Table of Contents

# Table of Contents

## Table of Contents

# Design of 68-bit Carry Look-Ahead Adder

## What is an Arithmetic Logic Unit?

- Arithmetic Logic Unit is a combinational circuit whose purpose is to perform Arithmetic and Logic instructions on integer binary numbers.

- An Arithmetic Logic Unit or an ALU as the name suggests consists of the Arithmetic Unit and a Logic Unit.

- ALU is a clustered substitute to having distributed combinational circuit i.e. instead of individual registers performing microoperations (like incrementing/shifting etc.) directly, an ALU is amassed to perform all the microoperations like arithmetic and logic operations.

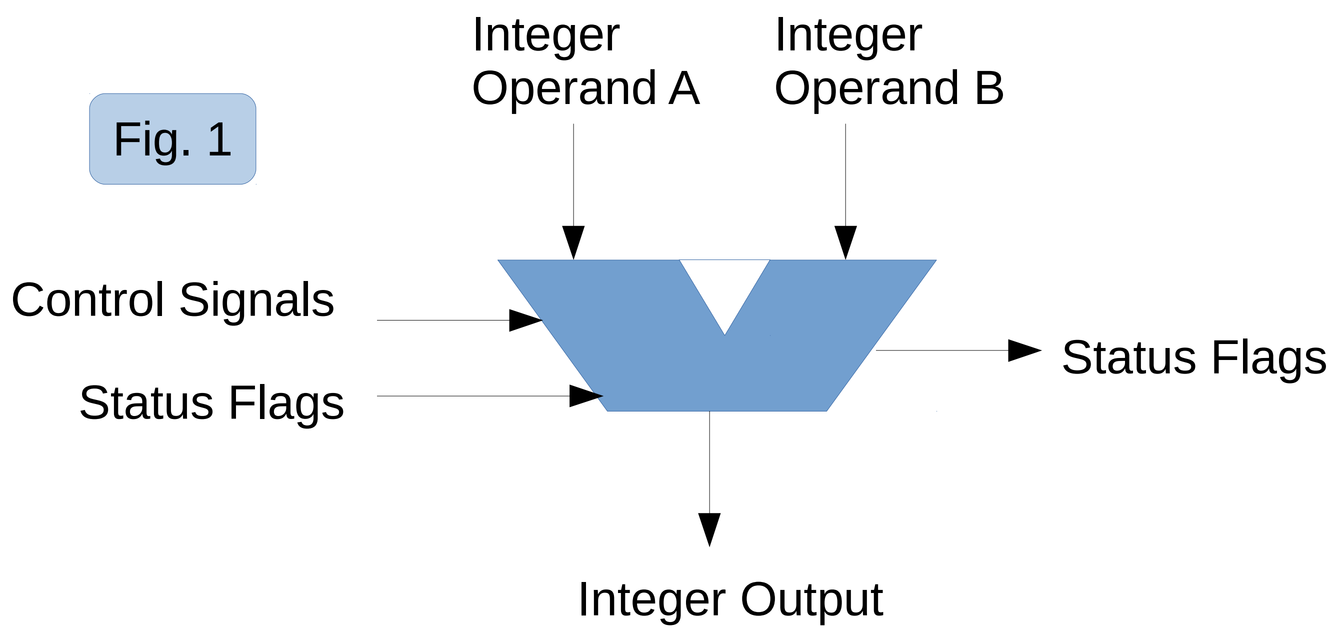
- A microoperation can be performed using an ALU by placing output of selected input registers at the common ALU's input terminals. The ALU performs the microoperation and the result is placed in the destination register.

- Since, ALU is a combinational circuit, the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock period.

# ALU Symbolic Notation

- The symbol for ALU is generally as shown below:

Fig. 1



- An ALU generally takes in two operands since, most microoperatons are binary in nature, and, generates a single integer binary output.
- The direction of arrows indicates whether a signal is input to the ALU or is an output from the ALU. The inputs are the integer operands A and B, control signals and the status flags. Output is the integer output and the status flags.

# ALU Input/Output signals

- Depending on the type of architecture, an ALU's output could be a Stack in Stack based machine, an Accumulator in an accumulator based machine or a general purpose register in register-memory architecture or register-register architecture. On similar grounds, the inputs to the ALU may come from a stack in a stack based machine, from accumulator and memory in an accumulator based machine, one input from a general purpose register and another from memory in a register-memory architecture or  both the inputs from general purpose registers in a register-register architecture.
- The other inputs to the ALU are the control signals decoded from the instruction present in the instruction register by the instruction register. Also, input to the ALU are the status flags generated  by the ALU itself as a result of the Arithmetic or Logic operation. The flags can be the zero flag, the negative flag or the overflow flag for example.

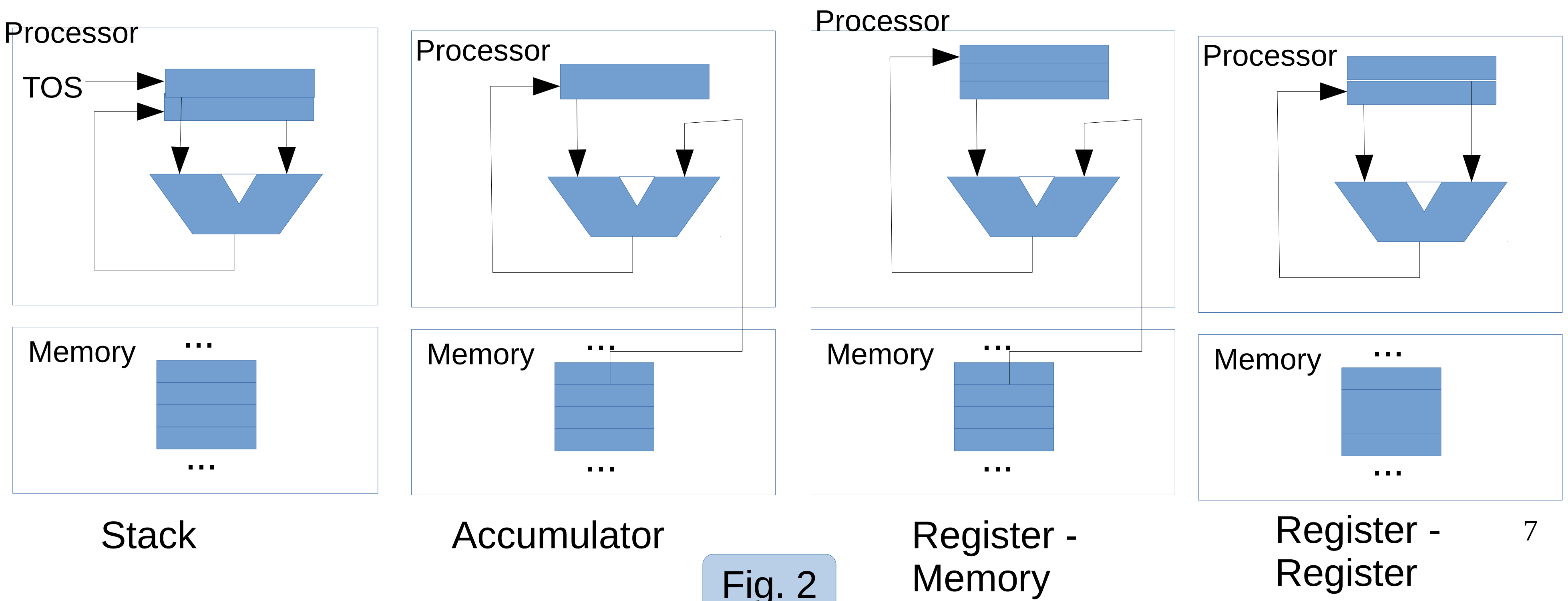


Fig. 2

## ALU Input/Output signals

- Data: The inputs A and B of the ALU and the output Y are connected to three separate data busses which generally connect to registers or memory as shown on previous slide. Typically, the input or output bus widths are same and match the word size of the enclosing CPU or other processor.

- Control Signals / Opcode: The control signals or the opcode (Operation code) decides the microoperation i.e arithmetic or logic operation that is to be performed on the inputs A and B. The opcode size i.e its bus width determines the number of microoperations that the ALU can perform. The opcode is generally encoded in the binary instruction and is decoded by the instruction decoder.

- Status Flags: Status Flags are single bit signals that convey information about the result of the current ALU operation. Common ALU status signals are as follows:

  * Carry-out: Conveys carry, borrow or overflow bit depending on microoperation performed:
  addition, subtraction or binary shift operation.

  * Zero: Indicates that all the bits of the output are zero.

  * Negative: Indicates that the output is negative.

  * Overflow: Indicates that the result of an arithmetic operation has exceeded the numeric range of o/p Y.

  * Parity: Indicates that even or odd number of bits in Y are logic one.

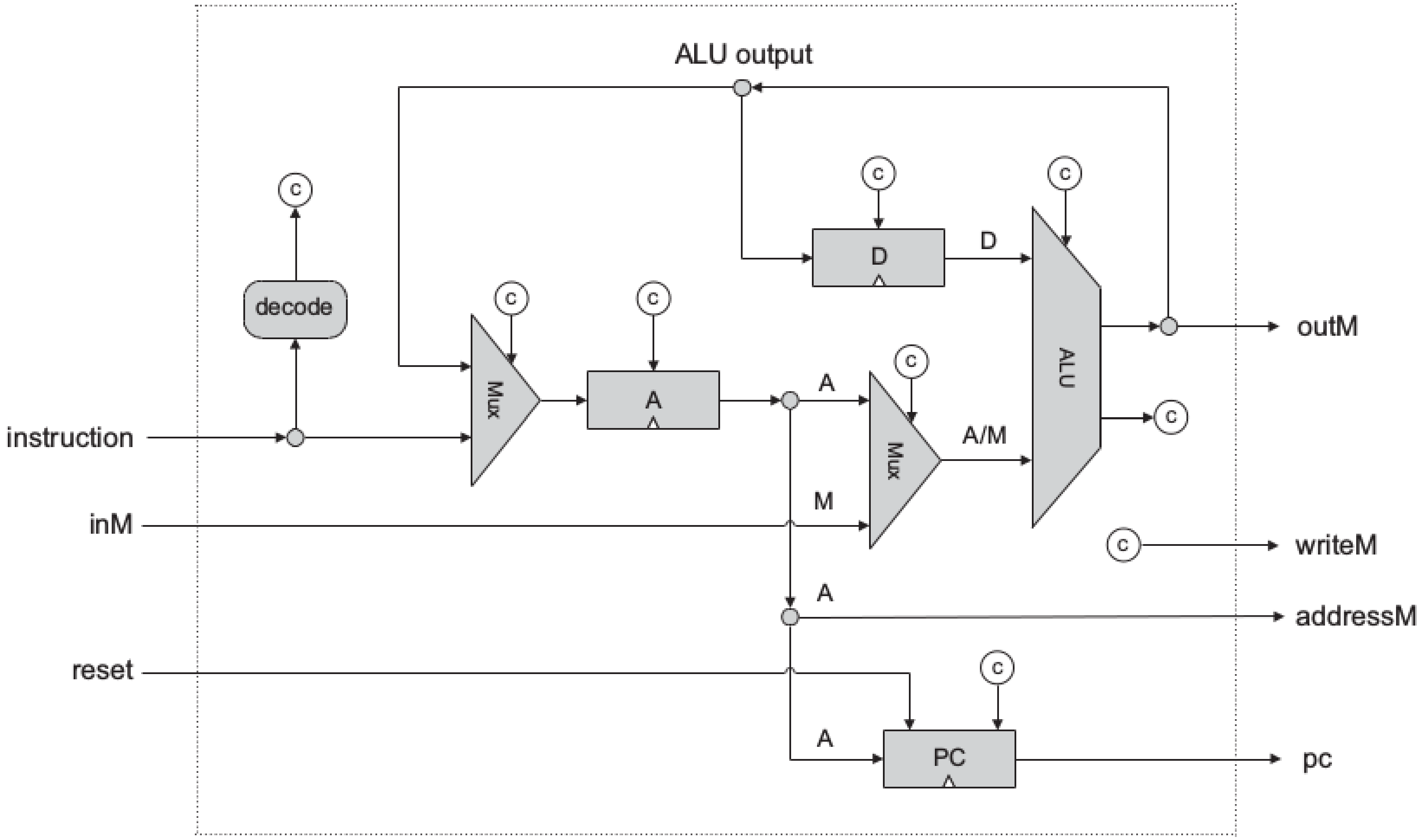
  The status bits are utilized in control the conditional branch decisions generated in the branching logic. Other uses include interrupt enable bits, data transfer flags etc.

# ALU Operation

- During normal operation, stable signals are applied to all of the ALU inputs for enough time known as the propagation delay so that the signals can propagate through the ALU circuitry. The result of this ALU operation gives the output Y and the status flags.
- The task of the external circuitry is to ensure the stability of ALU input signals throughout the microoperation and for ensuring sufficient time for the signals to propagate through the ALU before sampling the ALU result.
- Typically, the external circuitry consists of sequential logic such as registers/memory to control the ALU operation, which is clocked by a clock signal of a sufficiently low frequency to allow enough time for the ALU outputs to settle under worst case or maximum delay conditions.
- Consider the ALU configuration in the CPU of figure 3. Before beginning the ALU operation, the CPU saves the operands into the D register or the A register or the memory. It then routes the operands from these register/memory sources to the ALU's operand inputs as shown in the figure, while decoded control signals from the instruction are fed into the opcode input of the ALU simultaneously configuring the ALU to perform the selected microoperation. At the same instance, CPU also routes the ALU output to a destination register or an external memory to receive the result.
- The ALU's input signals are kept stable, i.e the registers A or D, untill they propagate through the ALU to the destination register while the CPU waits for the next clock. When the next clock arrives, the destination register A's load input or the memory is enabled to store the ALU result.
- Since, the ALU operation has completed, the ALU inputs may be set up for the next ALU operation.

## ALU within a CPU

Fig. 3



## ALU Functions

- Typical arithmetic and logic functions an ALU performs are listed in table 1.

Table 1

| Arithmetic Operations | Logical Operatons | Shift Operations |
|---|---|---|
| Add | AND | Arithmetic shift |
| Add with Carry | OR | Logical shift |
| Subtract | Exclusive-OR | Rotate |
| Subtract with borrow | One's complement | Rotate through carry |
| Negation | | |
| Increment | | |
| Decrement | | |
| Pass through | | |

## Importance of Addition

- One of the fundamental operations performed by an ALU is addition. It is the most common arithmetic operation and also serves as a building block for synthesizing many other operations.

- The commonality of arithmetic operations can be gauged from the fact that addition is performed extensively both in explicitly specified computation steps and as a part of implicit ones. (For example, in computing addresses for index addressing). Also, ALU's which do not have dedicated harware for multiplication and division, these operations are performed as sequences of additions. Similarly, subtraction is performed by negating the subtrahend and adding the result to the minuend.

- Addition is thus a frequent operation since most of the algorithms for complex operations are finally reduced to addition. One of the fundamental principles of computer design states use of technical solutions that favor those parts of the system which are most often requested (“Make the common case fast”) to obtain the best performance possible.

- The quantitative estimation of the acceleration obtained by applying this principle is determined through the Amdahl's law, according to which the addition operation should be favored by finding solutions that make its implementation as easy as possible and implicitly, as efficient as possible.

# Amdahl's Law

- The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's law. Amdahl's law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Amdahl's law gives a quick way to find the speedup from some enhancement, which depends on two factors:
  - The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement.
  - The improvement gained by the enhanced execution mode, that is, how much faster the task would run if the enhanced mode were used for the entire progam.

- The execution time using the original computer with the enhanced mode will be the time spent using the unenhanced portion of the computer plus the time spent using the enhancement:

  $\text{Execution Time}_{\text{new}} = \text{Execution Time}_{\text{old}} * ((1 – \text{Fraction}_{\text{enhanced}}) + \text{Fraction}_{\text{enhanced}} / \text{Speedup}_{\text{enhanced}})$

  $\text{Speedup}_{\text{overall}} = \text{Execution Time}_{\text{old}} / \text{Execution Time}_{\text{new}}$

# Adder and its Types

- Adder, as the name suggests, is a combinational circuit that performs addition operation. Its generally a part of the ALU.

- Following are some of the different types of binary adder circuits based on number of bits/number of inputs :

  - Half Adder
  - Full Adder
  - Ripple-Carry Adder
  - Manchester Adder
  - Carry-Lookahead Adder
  - Conditional Sum Adder
  - Carry Select Adder
  - Carry Skip Adder
  - Carry Save Adder

- Each of the adder types that is mentioned above tries to improve the performance of addition in an ALU so that the most common operation of addition can be made faster.

# Half Adder

- Half adder is a circuit that sums two input bits and produces the sum and the carry-out.
- The arithmetic equations for one-bit adder: $s = (x + y) \bmod 2$, and $c_{out} = [(x + y) / 2]$
- The truth table and Karnaugh maps for the sum(s) and the carry($c_{out}$) out signals is as follows:

Table 2

| x | y | s |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$$\begin{array}{c|cc} x \backslash y & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array}$$

Table 3

| x | y | $c_{out}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$$\begin{array}{c|cc} x \backslash y & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$$

- As per the truth table, the logical equations for sum, $s = xy' + x'y = x \oplus y$ ............ Eq. 1
  carry out ($c_{out}$) $= x.y$ ................................ Eq. 2

# Half adder circuit complexity

- Circuit diagram for half adder would be as follows:

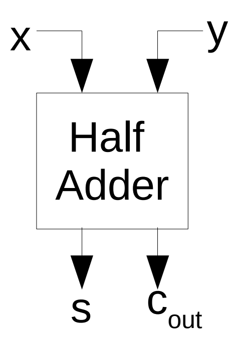
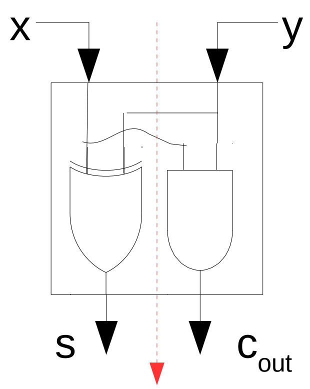


Fig. 4

<=====>



- Space complexity (C):
  - C = 1 XOR + 1 AND
  - 2 gates in total
- Critical path delay (T):
  - x/y => (red dashed line) => s or x/y => (red dashed line) => $c_{out}$
  - $T = 1\Delta_g$

    where $\Delta_g$ is equal to one gate delay and every type of gate has equal delay.

# Full Adder

- Full adder is a circuit that sums two input bits and an input carry in signal and produces the sum and the carry-out.

- The arithmetic equations for one-bit adder: $s = (x + y + c_{in}) \bmod 2$, and $c_{out} = [(x + y + c_{in}) / 2]$

- The truth table and Karnaugh maps for the sum(s) and the carry($c_{out}$) out signals is as follows:

| x | y | $c_{in}$ | s |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

$$\begin{array}{c|cccc} x \backslash yc_{in} & 00 & 01 & 11 & 10 \\ \hline 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{array}$$

Table 4

| x | y | $c_{in}$ | $c_{out}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

$$\begin{array}{c|cccc} x \backslash yc_{in} & 00 & 01 & 11 & 10 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \end{array}$$

Table 5

- As per the truth table, the logical equations for sum(s) $= xy'c_{cin}' + x'y'c_{in} + xyc_{in} + x'yc_{in}'$

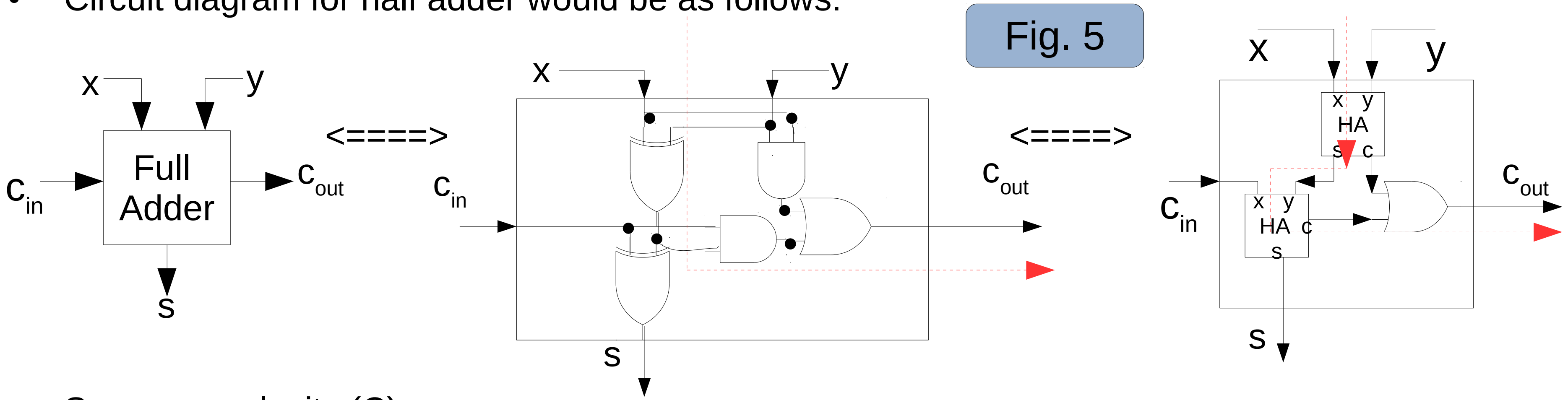
  $= y'(xc_{in}' + x'c_{in}) + y(xc_{in} + x'c_{in}') = y'(x \oplus c_{in}) + y(x \oplus c_{in})' = x \oplus y \oplus c_{in}$ ................................Eq. 3

  carry out ($c_{out}$) $= xy'c_{in} + x'yc_{in} + xy = (x \oplus y)c_{in} + xy$........Eq. 4

# Full adder circuit complexity

- Circuit diagram for half adder would be as follows:

Fig. 5



- Space complexity (C):
  - C = 2 XOR + 2 AND + 1 OR
  - 5 gates in total
- Critical path delay (T):
  - $T = 3\Delta_g$

| Input | Output | Delay |
|---|---|---|
| x/y | s | $2\Delta_g$ |
| x/y | $c_{out}$ | $3\Delta_g$ |
| $c_{in}$ | s | $1\Delta_g$ |
| $c_{in}$ | $c_{out}$ | $2\Delta_g$ |

Table 6

where $\Delta_g$ is equal to one gate delay and every type of gate has equal delay.

# Ripple carry adder

- Untill now, we have seen single bit addition circuits i.e. Half Adder and Full Adder
- Ripple carry adder is an adder circuit for performing n-bit addition for which the building block is a Full Adder (FA).
- As for a full adder, the algorithm for single bit addition would remain the same except that addition would be performed for the ith bit.

$$S_i = x_i \oplus y_i \oplus c_i \; ; \quad c_{i+1} = x_i y_i + c_i (x_i \oplus y_i) \; ; \quad i = 0,1,2,\ldots.,n$$

- Circuit diagram for the n-bit ripple carry adder:

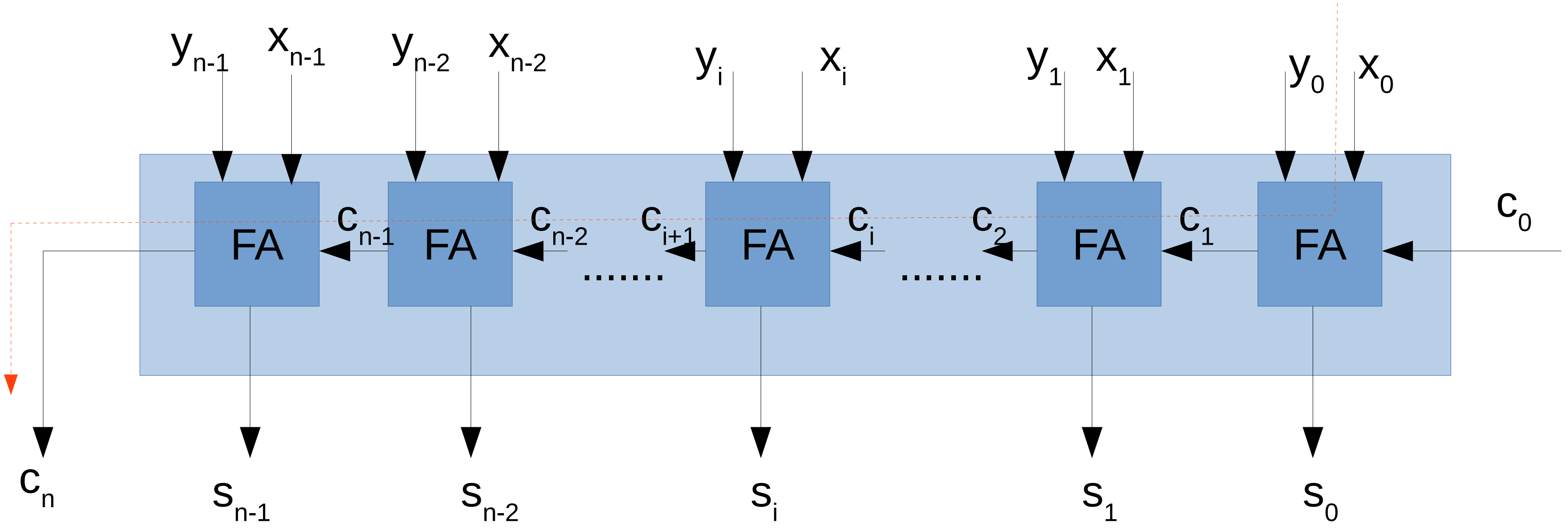


Fig. 6

## RCA circuit complexity

- Space complexity (C):
  - C = n FA

Table 7

| Input | Output | Delay |
|---|---|---|
| $x_i/y_i$ | $s_i$ | $2\Delta_g$ |
| $x_i/y_i$ | $s_j$ (j > i) | $3\Delta_g + 2(j-i-1)\Delta_g + 1\Delta_g$ |
| $x_i/y_i$ | $c_{i+1}$ | $3\Delta_g$ |
| $x_i/y_i$ | $c_j$ (j > i+1) | $3\Delta_g + 2(j-i-1)\Delta_g$ |
| $c_i$ | $s_i$ | $1\Delta_g$ |
| $c_i$ | $s_j$ (j > i) | $2(j-i)\Delta_g + 1\Delta_g$ |
| $c_i$ | $c_j$ (j > i) | $2\Delta_g$ |

- Critical path delay (T):

  $x_i/y_i$ to $s_j$ is max when j = n-1 and i = 0 => delay = $3\Delta_g + 2(n-2)\Delta_g + 1\Delta_g = 2n\Delta_g$

  $x_i/y_i$ to $c_j$ is max when j = n and i = 0 => delay = $3\Delta_g + 2(n-1)\Delta_g = (2n + 1)\Delta_g$

  $c_i$ to $s_j$ is max when j = n-1 and i = 0 => delay = $2(n-1)\Delta_g + 1\Delta_g = (2n - 1)\Delta_g$

  - Hence, T = $(2n + 1)\Delta_g$ when $x_0/y_0$ to $c_n$ path is considered. => O(n)

    where $\Delta_g$ is equal to one gate delay and every type of gate has equal delay.

## Improving performance of RCA

- The carry chain for the ripple carry adder forms the critical path and determines the speed of the adder.
- To improve the adder performance, it is therefore necessary to have low-latency carry network.
- If the carry into position i is known, the sum digit can be determined from the operand digits $x_i$ and $y_i$ and the incoming carry $c_i$ in constant time through modular addition:

$s_i = (x_i + y_i + c_i) \bmod r$ where r is the radix of addition

For r = 2 => $s_i = x_i \oplus y_i \oplus c_i$

- This reduces the design of two-operand adders to the computation of the k carries $c_{i+1}$ based on the 2k operand digits $x_i$ and $y_i$, $0 \le i < k$
- Analysis of the carry network would show that a carry is generated at the $i_{th}$ node of a ripple carry adder if both the inputs to the full adder are one. So, we can define the carry generation signal as the AND of $x_i$ and $y_i$.

=> $g_i = x_i . y_i$

# Carry generate, propagate and kill signals

- Similarly, a carry is propagated through the $i_{th}$ node of the ripple carry adder even if a single input to the ripple carry adder is one at the $i_{th}$ node. So, we can define a carry propagator signal as:

$$p_i = x_i \oplus y_i$$

- A carry ends at the $i_{th}$ node of the ripple carry adder if both the inputs to the ripple carry adder are zero. So, we can define a carry kill signal as:

$$k_i = x_i' \,.\, y_i'$$

- Hence, a carry at i+1th position is created if carry is generated at ith node or is propagated by the ith node. The carry recurrence can then be written as follows:

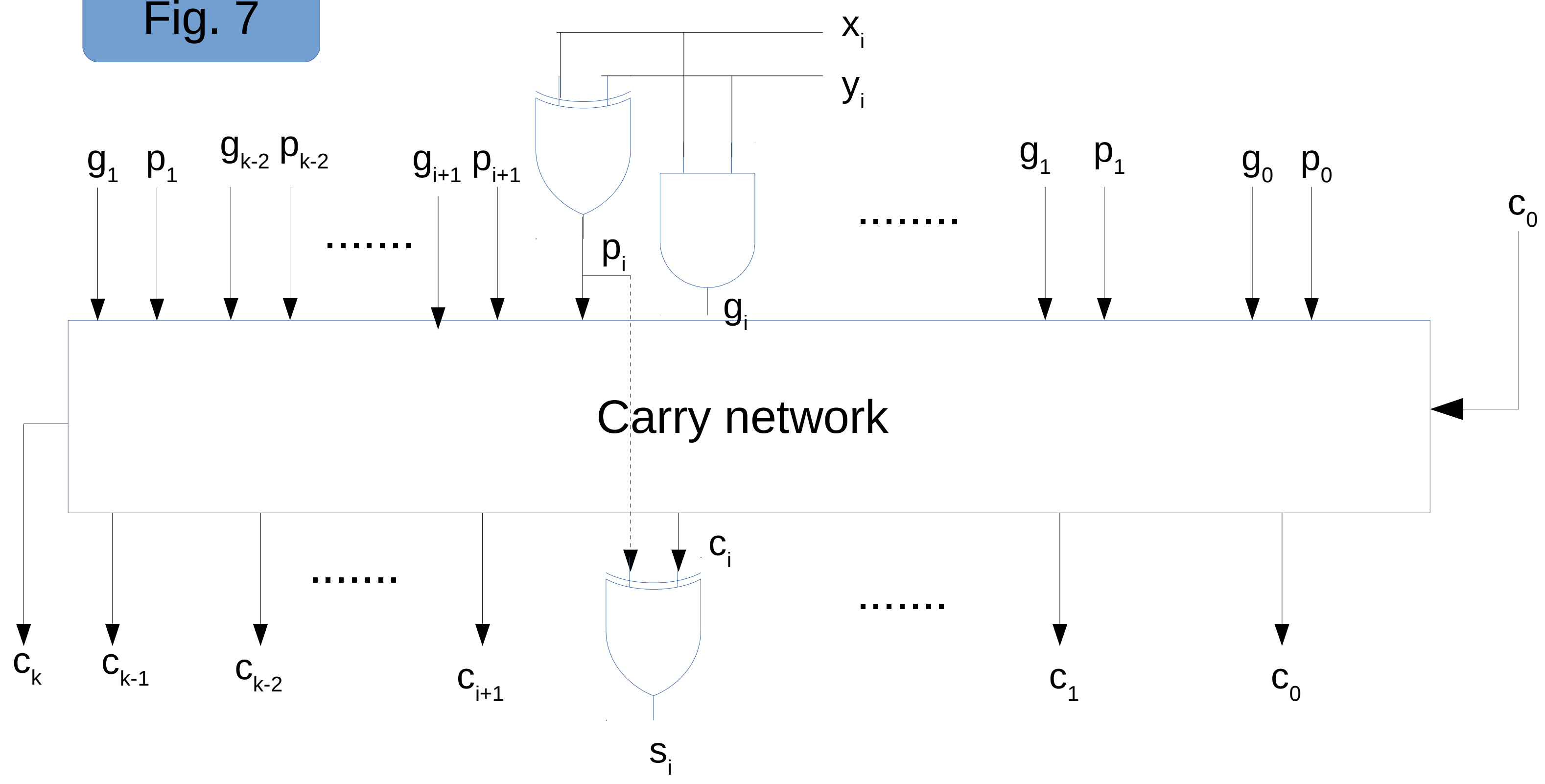
$$\Rightarrow c_{i+1} = g_i + c_i \,.\, P_i$$

- Taking advantage of generate and propagte signals defined above, the adder design can be viewed in the generic form as in the figure 7.

| $g_i$ | $p_i$ | Carry is: |
|---|---|---|
| 0 | 0 | Killed |
| 0 | 1 | Propagated |
| 1 | 0 | Generated |
| 1 | 1 | Not possible |

# Generic structure of a binary adder

Fig. 7



- Any adder will have the two sets of AND and XOR gates at the top to form the $g_i$ and $p_i$ signals, and it will have a set of XOR gates at the bottom to produce the sum bits $s_i$ . It will
differ, however, in the design of its carry network.

# Carry recurrence

- Carry recurrence $c_{i+1} = g_i + c_i \, . \, p_i$ states that a carry will enter stage i + 1 if it is generated in stage i or is propagated by that stage. The recursive carry recurrence relation allows us to obtain carry as a function of operand bits and $c_{in}$

$$
\begin{aligned}
c_1 &= g_1 + c_0 . \, p_1 \\
c_2 &= g_2 + c_1 . \, p_2 = g_2 + (g_1 + c_0 . \, p_1) . \, p_2 = g_2 + g_1 . \, p_2 + c_0 . p_1 . p_2 \\
c_3 &= g_3 + c_2 . \, p_3 = g_3 + (g_2 + g_1 . \, p_2 + c_0 . \, p_1 . \, p_2) . \, p_3 \\
&= g_3 + g_2 . \, p_3 + g_1 . \, p_2 . \, p_3 + c_0 . \, p_1 . \, p_2 . \, p_3 \\
\\
c_4 &= g_4 + c_3 . \, p_4 = g_4 + g_3 . p_4 + g_2 . p_3 p_4 + g_1 . \, p_2 . \, p_3 . p_4 + c_0 . \, p_1 . \, p_2 . \, p_3 . \, p_4
\end{aligned}
$$

- $c_0$ and $c_4$ are the carry in and carry out signals of the adder. The carry network in figure 7 can be replaced with logic corresponding to the above equations. The adder circuit obtained from such a network is called as carry lookahead adder.

- The above equations also suggest that carry generation from the recurrence relation will be difficult to implement as the fanin for the carry signal grows i.e. if the recurrence is applied repeatedly.

## Carry Lookahead Adder

- The carry recurrence relation results in a form of adder known as Carry Lookahead adder.
- The carry network of figure 7 in a carry lookahead adder is described by the carry recurrence relation and is known as a n-bit lookahead carry generator.
- Since, full carry lookahead is impractical for wide words due large fanin of the carry signals, a 4-bit adder is used in a multilevel lookahead configuration for addition of wider words.
- The 4-bit carry lookahead adder is described by the following logic equations:

$G_i = x_i \,.\, y_i$ where $G_i$ is the carry generate and i = 0,1,2,3

$P_i = x_i \oplus y_i$ where $P_i$ is the carry propagate and i = 0,1,2,3

$c_{i+1} = G_i + c_i \,.\, P_i$ where $c_i$ is the carry input to ith adder node and i = 0,1,2,3

$s_i = x_i \oplus y_i \oplus c_i$ where $s_i$ is the sum bit generated by the ith adder node and i = 0,1,2,3

x and y are the 4-bit numbers to be added.

# Implementation of 68-bit CLA Adder

- In this project, we consider the implementation of 68-bit Carry Lookahead Adder.
- The 68-bit CLA is implemented as a hierarchy of smaller units of Carry Lookahead Adder's. Specifically, the implementation consists of a 64-bit CLA cascaded with a 4-bit CLA through a 2-bit lookahead carry generator. The 64-bit CLA is further composed of 16-bit CLA's connected through the 4-bit lookahead carry generator. The 16-bit CLA is composed of 4-bit CLA's connected through the 4-bit lookahead carry generator. The 4-bit CLA is composed of “Carry generator and Carry propagator” circuits connected through the 4-bit lookahead carry generator.
- The inputs to the 68-bit CLA are two 68-bit binary integer operands X and Y, and the output of the 68-bit CLA is the 69 bit sum.
- Following equations describe the 68-bit carry lookahead adder:

$$sum = \{ sum_{68} , sum_{67-0} \}$$

$$sum_{68} = C_{68} \; ; \; sum_{67-0} = \{sum_{67-64} , sum_{63-0}\}$$

$$sum_{67-64} = S_{67-64} = \{s_{67} , s_{66} , s_{65} , s_{64} \}$$

$$sum_{63-0} = S_{63-0} = \{s_{63} , s_{62} , s_{61} , s_{60} , \ldots\ldots , s_{3} , s_{2} , s_{1} , s_{0} \}$$

$$s_i = X_i \oplus Y_i \oplus C_i \quad \text{where } i = 0 \ldots 67$$

$$C_{64} = G_{63-0} + C_0 . P_{63-0}$$

$$C_{68} = G_{67-64} + C_{64} . P_{67-64} = G_{67-64} + (G_{63-0} + C_0 . P_{63-0}) . P_{67-64} = G_{67-64} + G_{63-0} . P_{67-64} + C_0 . P_{63-0} . P_{67-64}$$

# Input/Output of 68-bit CLA Adder

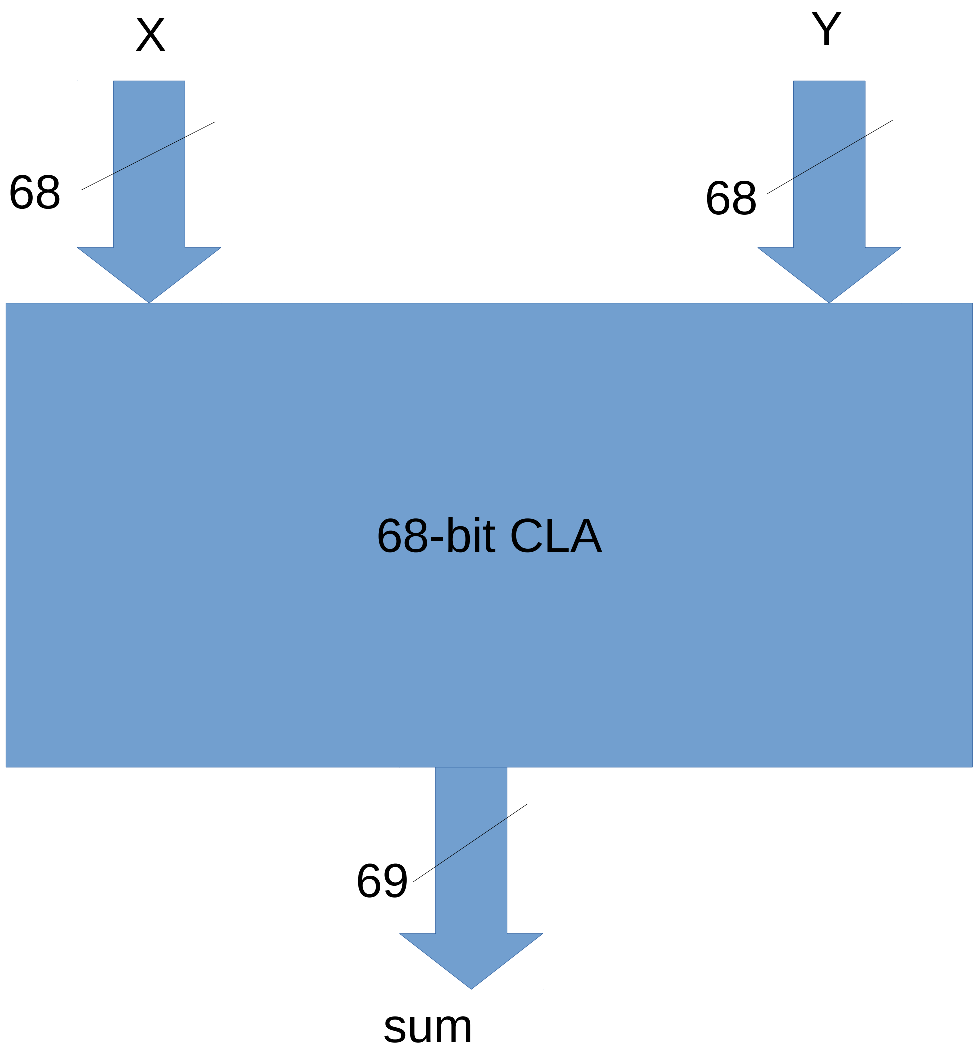


Fig. 8

# 68-bit CLA adder circuit complexity

- Critical path for the 68-bit CLA adder is shown in figure 9 using the red dotted line.

- The critical path for the 68-bit CLA adder is from the input signal $X_{67-0}$ or $Y_{67-0}$ to $S_{63-0}$ through the 64-bit CLA adder.

- Space complexity (C):
  - C = 64-bit CLA adder + 4-bit CLA adder + 4-bit lookahead carry generator =

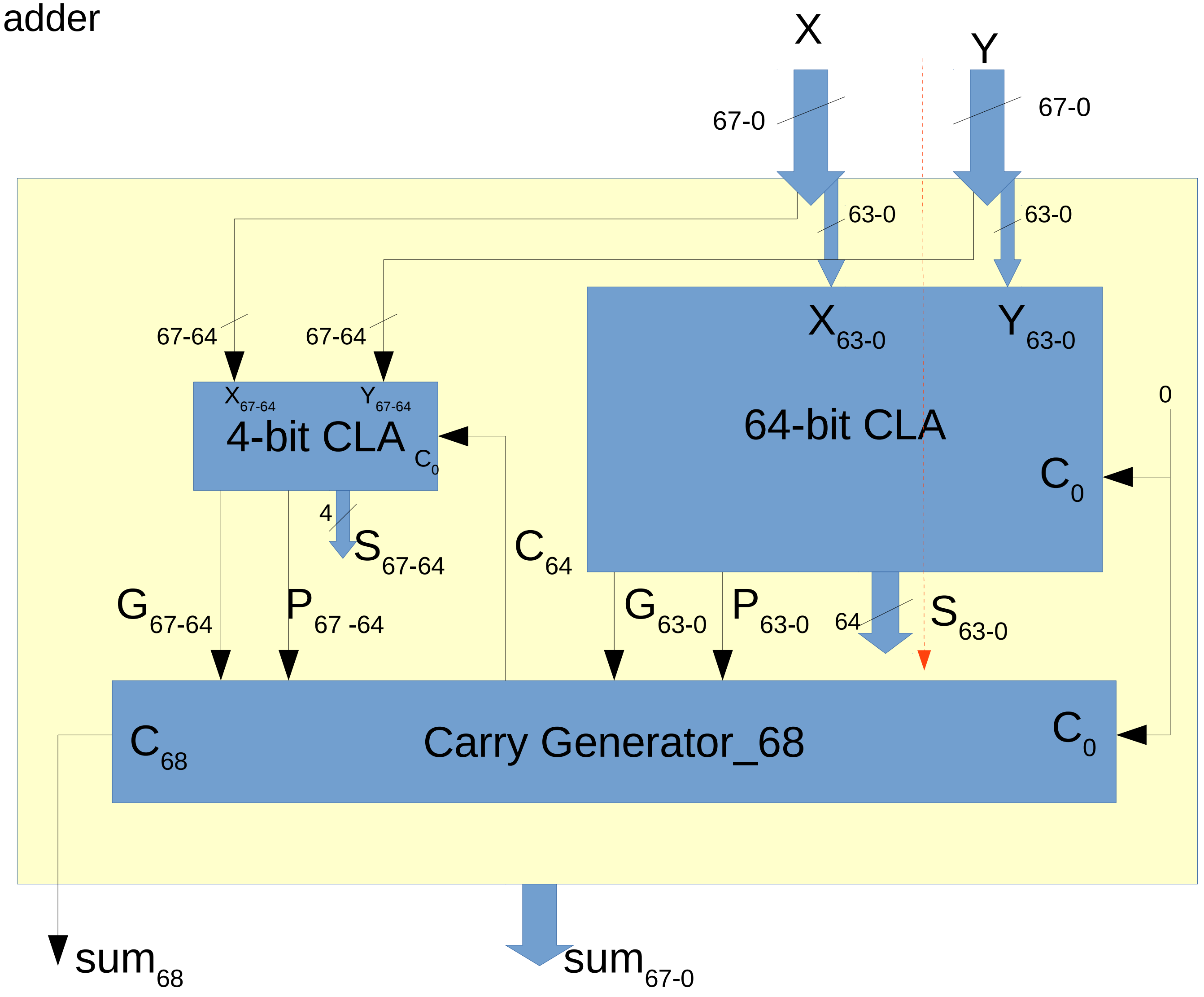
    (212 XOR + 274 AND) + (12 XOR + 14 AND) + (4 XOR + 10 AND) = 228 XOR + 298 AND
  - 526 gates in total

- Critical path delay (T): $T = 5\Delta_g + 2\Delta_g + 5\Delta_g = 12\,\Delta_g$ (Same as for a 64-bit CLA adder as discussed later)

# Block diagram of 68-bit CLA Adder

- 68-bit CLA adder

Fig. 9



# 2-bit lookahead carry generator circuit diagram

- The “Carry generator_68” block has the following circuit diagram:

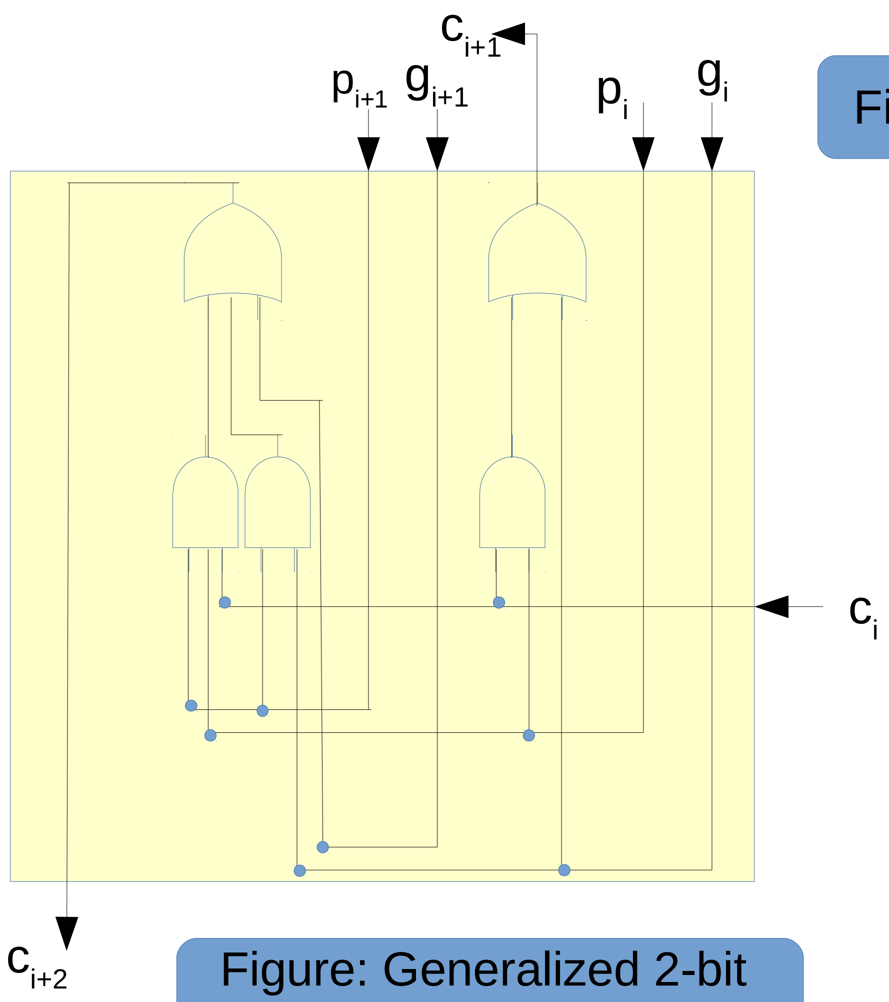


Figure: Generalized 2-bit lookahead carry generator

Fig. 10

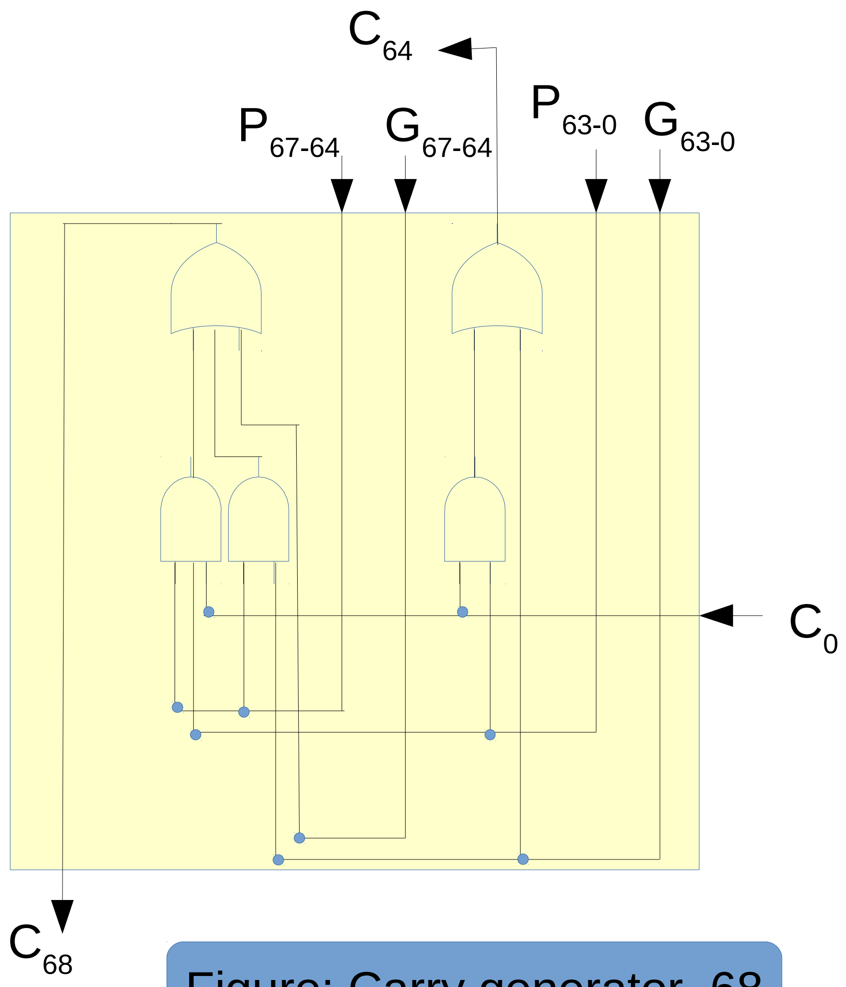


Figure: Carry generator_68

# 64-bit CLA adder

- The multilevel lookahead is realized by designing the adder as a hierarchical design.
- 64-bit CLA adder is one of the blocks inside the 68-bit CLA adder.
- 64-bit CLA adder is described by the following logic equations:
- Block sum bits:

  $S_{63-0} = \{S_{63-48}, S_{47-32}, S_{31-16}, S_{15-0}\}$

  $S_{63-48} = \{s_{63}, s_{62}, s_{61}, \ldots, s_{50}, s_{49}, s_{48}\}$ ; $S_{47-32} = \{s_{47}, s_{46}, s_{45}, \ldots, s_{34}, s_{33}, s_{32}\}$

  $S_{31-16} = \{s_{31}, s_{30}, s_{29}, \ldots, s_{18}, s_{17}, s_{16}\}$ ; $S_{15-0} = \{s_{15}, s_{14}, s_{13}, \ldots, s_{2}, s_{1}, s_{0}\}$

- Block Generate and block propagate signals:

  $G_{63-0} = G_{63-48} + G_{47-32} . P_{63-48} + G_{31-16} . P_{47-32} . P_{63-48} + G_{15-0} . P_{31-16} . P_{47-32} . P_{63-48}$

  $P_{63-0} = P_{63-48} . P_{47-32} . P_{31-16} . P_{15-0}$

- Carry signals generated by the 4-bit lookahead carry generator within the 64-bit adder:

  $C_{16} = G_{15-0} + C_0 . P_{15-0}$

  $C_{32} = G_{31-16} + C_{16} . P_{31-16} = G_{31-16} + (G_{15-0} + C_0.P_{15-0}) . P_{31-16} = G_{31-16} + G_{15-0} . P_{31-16} + C_0.P_{15-0} .P_{31-16}$

  $C_{48} = G_{47-32} + C_{32} . P_{47-32} = G_{47-32} + (G_{31-16} + G_{15-0} . P_{31-16} + C_0 . P_{15-0} . P_{31-16}) . P_{47-32}$

  $= G_{47-32} + G_{31-16} . P_{47-32} + G_{15-0} .P_{31-16} . P_{47-32} + C_0 . P_{15-0} . P_{31-16} . P_{47-32}$

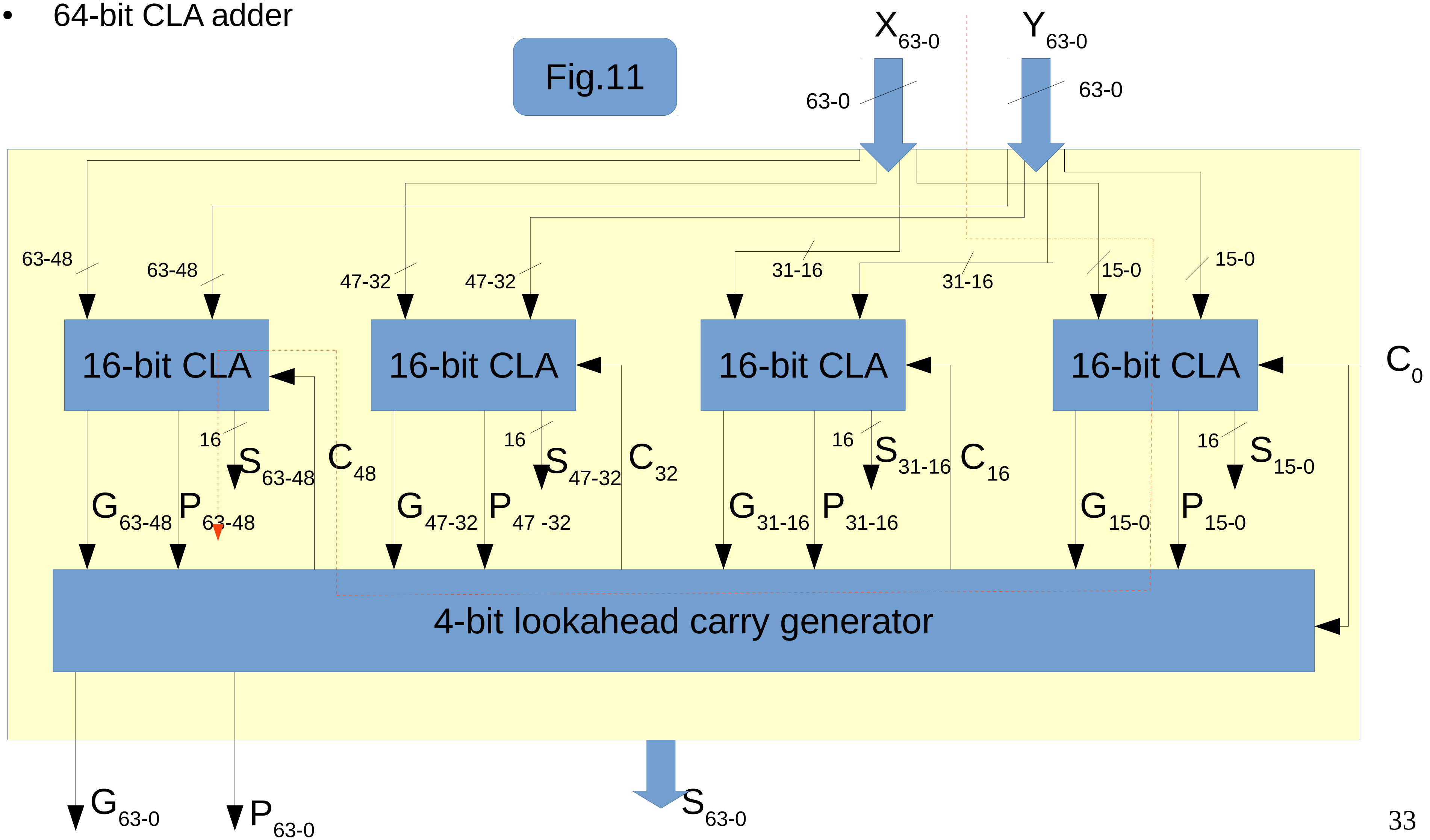
# 64-bit CLA adder circuit complexity

- The critical path for the 64-bit CLA adder is from the input signal $X_{63\text{-}0}$ or $Y_{63\text{-}0}$ to $S_{63\text{-}49}$ or $S_{47\text{-}33}$ or $S_{31\text{-}17}$ through the 4-bit lookahead carry generator.
- Space complexity (C):
  - C = Four 16-bit CLA adder + 4-bit lookahead carry generator = (4 * 52 XOR + 4 * 66 AND) + (4 XOR + 10 AND) = 212 XOR + 274 AND
  - 486 gates in total
- Critical path shown in red dashed line in the block diagram for 64-bit CLA adder and the path is from the inputs $X_{i+15\text{ - }i}$ /$Y_{i+15\text{ - }i}$ to $S_{j+2\text{ - }j}$ where i = 0/16/32; j > i+15 and j = 21/25/29/37/41/45/53/57/61
- Critical path delay (T): $T = 5\Delta_g + 2\Delta_g + 5\Delta_g = 12\ \Delta_g$
- The delay for the block generate signal $= 5\Delta g + 2\Delta g = 7\Delta_g$
- The delay from $C_0$ ($C_{in}$) to sum ($S_{61\text{-}63}$/$S_{57\text{-}59}$/$S_{53\text{-}55}$/$S_{45\text{-}47}$/$S_{41\text{-}43}$/$S_{37\text{-}39}$/$S_{31\text{-}29}$/$S_{27\text{-}25}$/$S_{23\text{-}21}$) $= 2\Delta_g + 5\Delta_g = 7\Delta_g$

# Block diagram of 64-bit CLA Adder

- 64-bit CLA adder

Fig.11



## 16-bit CLA adder

- 16-bit CLA adder is one of the building blocks of the 64-bit CLA adder.
- Similar to a 64-bit CLA adder, the 16-bit CLA adder is described by the following logic equations:
- Block sum bits:
  $S_{15\text{-}0} = \{S_{15\text{-}12}, S_{11\text{-}8}, S_{7\text{-}4}, S_{3\text{-}0}\}$
  $S_{15\text{-}12} = \{s_{15}, s_{14}, s_{13}, s_{12}\}$ ; $S_{11\text{-}8} = \{s_{11}, s_{10}, s_9, s_8\}$
  $S_{7\text{-}4} = \{s_7, s_6, s_5, s_4\}$ ; $S_{3\text{-}0} = \{s_3, s_2, s_1, s_0\}$
- Block Generate and block propagate signals:
  $G_{15\text{-}0} = G_{15\text{-}12} + G_{11\text{-}8} . P_{15\text{-}12} + G_{7\text{-}4} . P_{11\text{-}8} . P_{15\text{-}12} + G_{3\text{-}0} . P_{7\text{-}4} . P_{11\text{-}8} . P_{15\text{-}12}$
  $P_{15\text{-}0} = P_{15\text{-}12} . P_{11\text{-}8} . P_{7\text{-}4} . P_{3\text{-}0}$
- Carry signals generated by the 4-bit lookahead carry generator within the 16-bit adder:
  $C_4 = G_{3\text{-}0} + C_0 . P_{3\text{-}0}$
  $C_8 = G_{7\text{-}4} + C_4 . P_{7\text{-}4} = G_{7\text{-}4} + (G_{3\text{-}0} + C_0.P_{3\text{-}0}) . P_{7\text{-}4} = G_{7\text{-}4} + G_{3\text{-}0} . P_{7\text{-}4} + C_0.P_{3\text{-}0} .P_{7\text{-}4}$
  $C_{12} = G_{11\text{-}8} + C_8 . P_{11\text{-}8} = G_{11\text{-}8} + (G_{7\text{-}4} + G_{3\text{-}0} . P_{7\text{-}4} + C_0 . P_{3\text{-}0} . P_{7\text{-}4}) . P_{11\text{-}8}$
  $= G_{11\text{-}8} + G_{7\text{-}4} . P_{11\text{-}8} + G_{3\text{-}0} .P_{11\text{-}8} . P_{7\text{-}4} + C_0 . P_{11\text{-}8} . P_{7\text{-}4} . P_{3\text{-}0}$

# 16-bit CLA adder circuit complexity

- The critical path for the 16-bit CLA adder is from the input signal $X_{15-0}$ or $Y_{15-0}$ to $S_{7-5}$ or $S_{11-9}$ or $S_{15-13}$ through the 4-bit lookahead carry generator.

- Space complexity (C):
  - C = Four 4-bit CLA adder + 4-bit lookahead carry generator = (4 * 12 XOR + 4 * 14 AND) + (4 XOR + 10 AND) = 52 XOR + 66 AND
  - 118 gates in total

- Critical path shown in red dashed line in the block diagram for 16-bit CLA adder and the path is from the inputs $X_{i+3-i}/Y_{i+3-i}$ to $S_{j+2-j}$ where i=0/4/8; j > i+3 and j = 5/9/13 ( and not 4/8/12 because $C_{in}$ to $s_0$ is 1 $\Delta_g$ for 4-bit CLA and $C_{in}$ to $s_1/s_2/s_3$ is $3\Delta_g$)

- Critical path delay (T): $T = 3\Delta_g + 2\Delta_g + 3\Delta_g = 8\Delta_g$

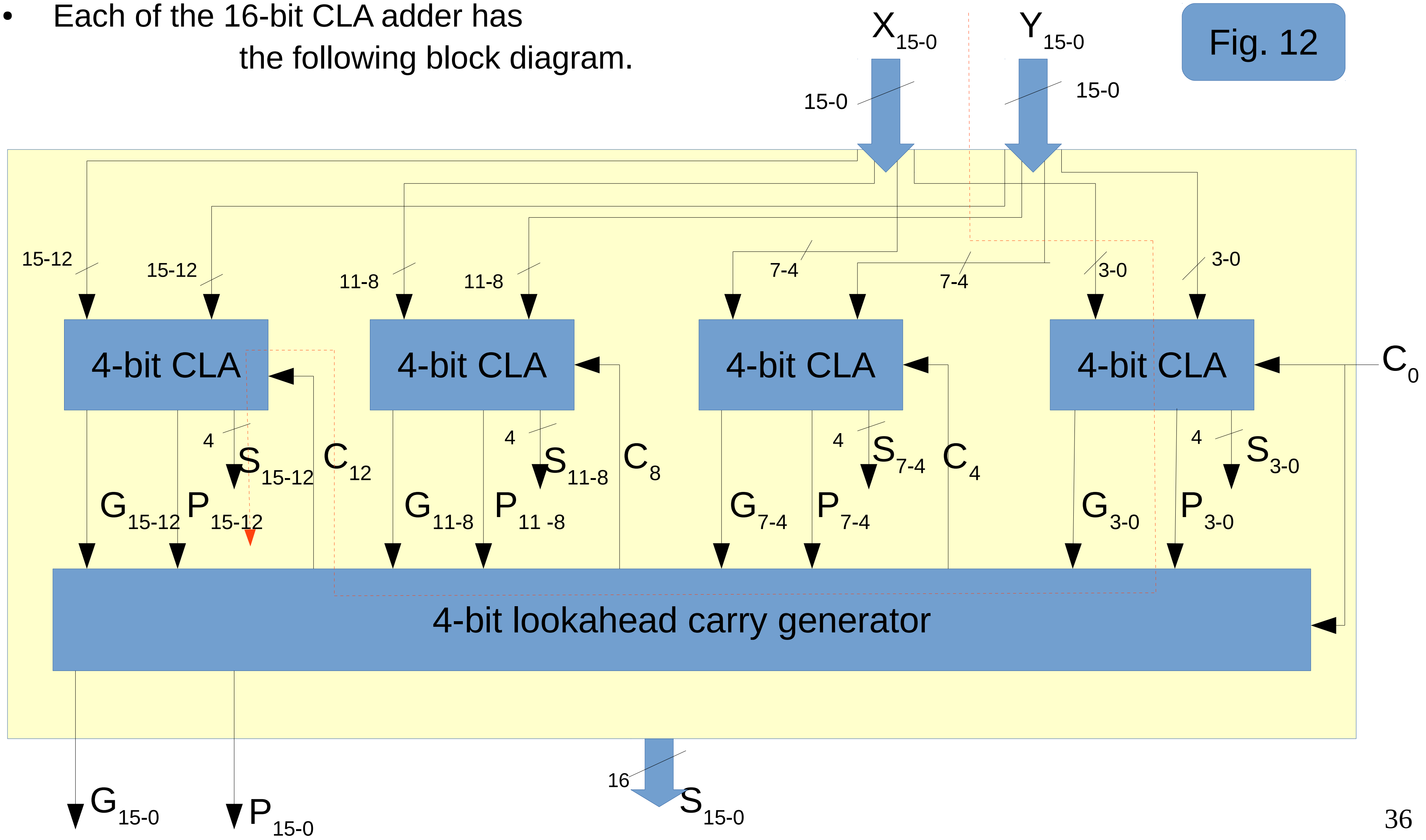
- The delay for the block generate signal = $3\Delta g + 2\Delta g = 5\Delta_g$

- The delay from $C_0$ ($C_{in}$) to sum ($S_{15-13}/S_{11-9}/S_{7-5}$) = $2\Delta_g + 3\Delta_g = 5\Delta_g$

# Block diagram of 16-bit CLA Adder

- Each of the 16-bit CLA adder has the following block diagram.

Fig. 12



# 4-bit CLA Adder

- 4-bit CLA adder is the building block for the implementation of the 68-bit CLA adder. In our design, 4-bit CLA adder is instantiated within the 68-bit CLA adder and is also used to build the 64-bit CLA adder.
- For the 4-bit carry lookahead adder, CG & CP is the building block for which the carry generator and carry propagation signals are described by:

$g_i = x_i \,.\, y_i$ for $i = 0,1,2,3$ (Carry generation)

$p_i = x_i \oplus y_i$ for $i = 0,1,2,3$ (Carry propagation)

$s_i = x_i \oplus y_i \oplus c_i$ for $i = 0,1,2,3$ (Sum)

- The carry signals for the 4-bit lookahead carry generator are described by the following equations:

$$c_1 = g_0 + c_0 p_0$$
$$c_2 = g_1 + c_1 p_1 = g_1 + (g_0 + c_0 p_0)\, p_1 = g_1 + g_0 p_1 + c_0 p_0 p_1$$
$$c_3 = g_2 + c_2 p_2 = g_2 + (g_1 + g_0 p_1 + c_0 p_0 p_1) p_2 = g_2 + g_1 p_{2d}$$

- The block generate and block propagate signals for the 4-bit CLA adderare described by the equation:

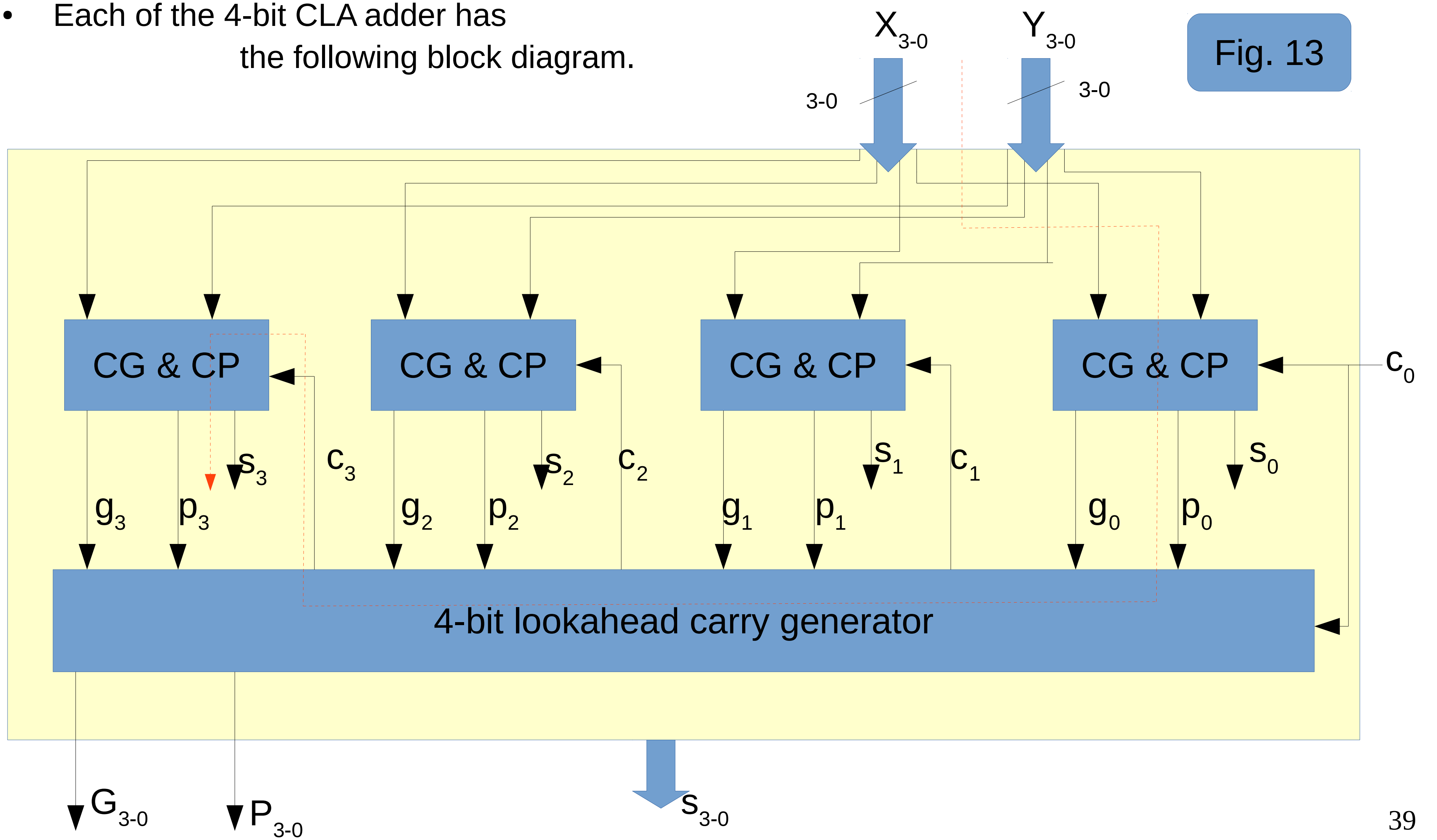
$$G_{3\text{-}0} = g_3 + g_2 \,.\, p_3 + g_1 \,.\, p_2 \,.\, p_3 + g_0 \,.\, p_1 \,.\, p_2 \,.\, p_3$$
$$P_{3\text{-}0} = p_3 \,.\, p_2 \,.\, p_1 \,.\, p_0$$

# 4-bit CLA adder circuit complexity

- The critical path for the 4-bit CLA adder is from the input signals $X_{3-0}$ or $Y_{3-0}$ to $s_1$ or $s_2$ or $s_3$ through the 4-bit lookahead carry generator.
- Space complexity (C):
  - C = 4 CG and CP + 4-bit lookahead carry generator = (4 * 2 XOR + 4 * 1 AND) + (4 XOR + 10 AND) = 12 XOR + 14 AND
  - 26 gates in total
- Critical path shown in red dashed line in the block diagram for 4-bit CLA adder and the path is from the inputs $x_i/y_i$ to $s_j$ where $j > i$
- Critical path delay (T): $T = 1\Delta_g + 2\Delta_g + 1\Delta_g = 4\Delta_g$
- The delay for the block generate signal $= 1\Delta g + 2\Delta g = 3\Delta_g$
- The delay from $c_0$ ($c_{in}$) to sum ($s_3/s_2/s_1$) = $3\Delta_g$

# Block diagram of 4-bit CLA Adder

- Each of the 4-bit CLA adder has the following block diagram.



# Carry generator and carry propagator

- The carry generator and carry propagator is the basic building block of the 4-bit CLA adder.
- It is the logic level realization for the carry generate and carry propagate signals and simply consists of two XOR gates and an AND gate.
- The CG and CP block has $x_i$, $y_i$ and $c_i$ as the input signals and $g_i$, $p_i$ and $s_i$ as the output signals.
- The logic equations describing the CG and CP block are same as that for carry generate, carry propage and sum signals as follows:

$$g_i = x_i \cdot y_i$$
$$p_i = x_i \oplus y_i$$
$$s_i = g_i \oplus c_i$$

- Space complexity (C):
  - C = 2 XOR + 1 AND
  - 3 gates in total
- Critical path delay (T): $T = 2\Delta_g$ for the path from inputs $x_i$ or $y_i$ to $s_i$

# Carry generator and Carry Propagator circuit diagram

- Each of “CG & CP” block, i.e. carry generator and carry propagator block has the following circuit diagram:

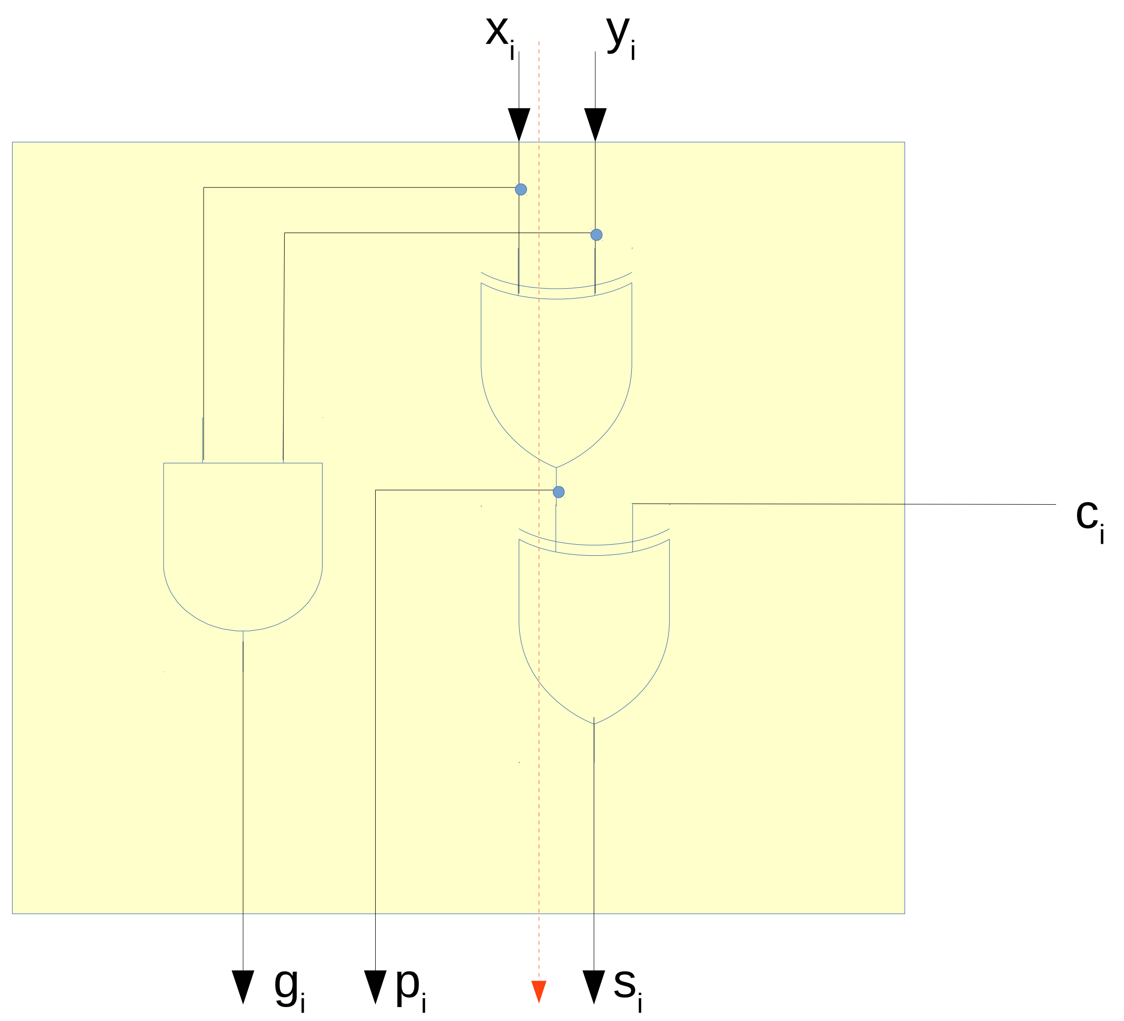


Fig. 14

# 4-bit lookahead carry generator

- The 4-bit lookahead carry generator is described by the carry recurrence relation and is an essential block for multi-level configuration of the carry lookahead adder.
- Following logic equation describes the generalized 4-bit lookahead carry generator circuit:
- According to the carry recurrence relation: $c_{i+1} = g_i + c_i \,.\, p_i$ ( where c is the carry, g is the carry generate and is the carry propagate).
- $c_i$ , $g_i$ , $g_{i+1}$, $g_{i+2}$ , $g_{i+3}$ , $p_i$ , $p_{I+1}$ , $p_{i+2}$ , $p_{i+3}$ are the inputs to the 4-bit lookahead carry generator and $c_{i+1}$ , $c_{i+2}$, $c_{i+3}$ , $g_{i,i+3}$ and $p_{i,i+3}$ are the outputs of the 4-bit lookahead carry generator.
- The 4-bit lookahead carry generator is described by the following logic equations:

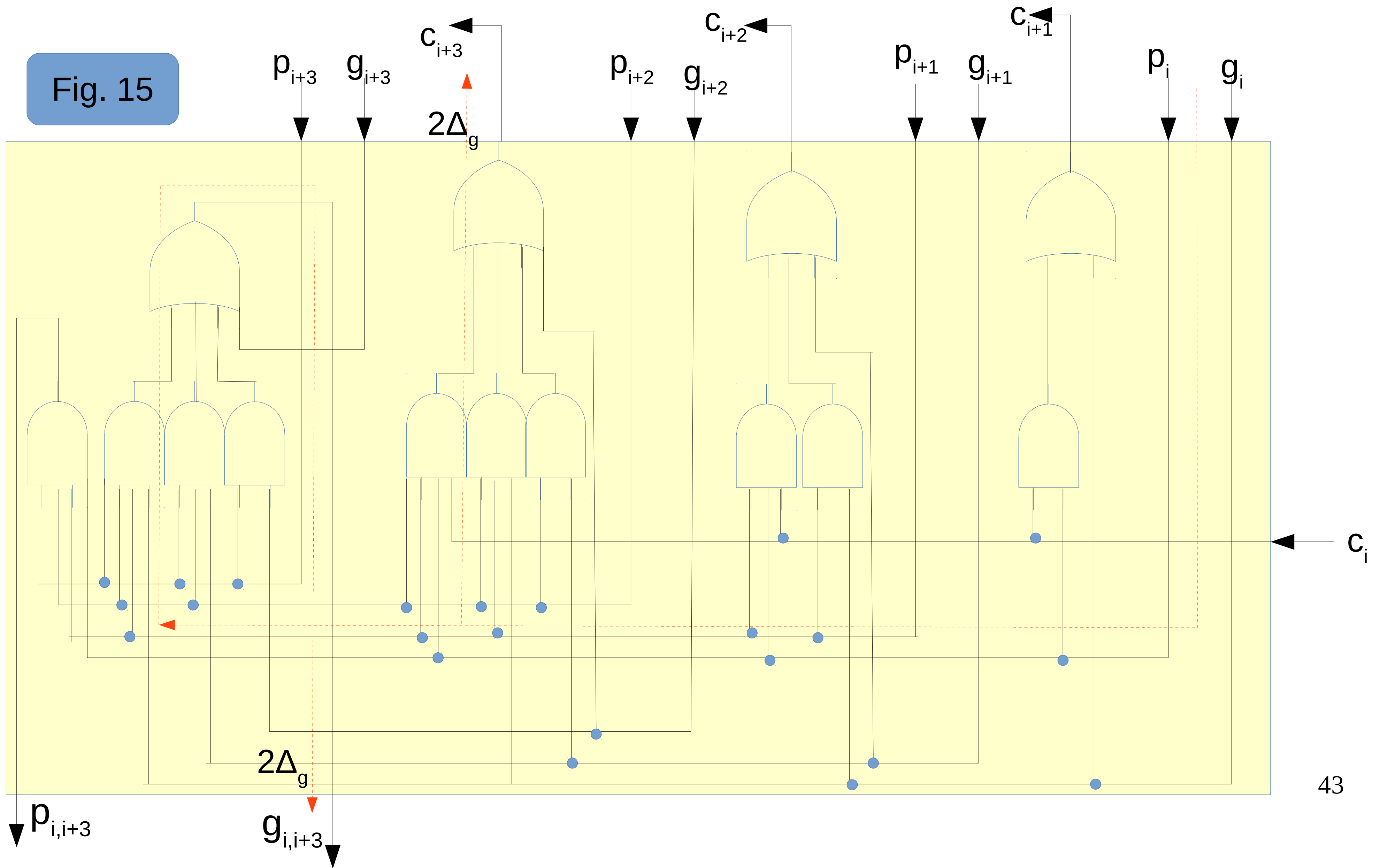
$$c_{i+1} = g_i + c_i \,.p_i$$
$$c_{i+2} = g_{i+1} + c_{i+1} \,.\, p_{i+1} = g_{i+1} + (g_i + c_i \,.\, p_i ) \,.\, p_{i+1} = g_{i+1} + g_i \,.\, p_{i+1} + c_i \,.\, p_i \,.\, p_{i+1}$$
$$c_{i+3} = g_{i+2} + c_{i+2} \,.\, p_{i+2} = g_{i+2} + (g_{i+1} + g_i \,.\, p_{i+1} + c_i \,.\, p_i.\, p_{i+1} ) \,.\, p_{i+2}$$
$$= g_{i+2} + g_{i+1} \,.\, p_{i+2} + g_i \,.\, p_{i+1} \,.\, p_{i+2} + c_i \,.\, p_i \,.p_{i+1} \,.\, P_{i+2}$$

- Space complexity (C):
  - C = 4 XOR + 10 AND
  - 13 gates in total
- Critical path delay (T): T = $2\Delta_g$ for the path from $g_i$ or $p_i$ or $c_i$ to $c_{i+1}$ or $c_{i+2}$ or $c_{i+3}$
  Same is the delay from $p_i$ or $c_i$ to $g_{i,i+3}$

# 4-bit lookahead carry generator circuit diagram

- Each of the “Carry generator” blocks has the following circuit diagram:

Fig. 15



## Verilog source code (3 input AND and OR gates)

- In the following sections , the verilog description for the 68-bit CLA adder is given:

```
module and3 (y, a, b, c);
      input a, b, c;
      output y;
      assign y = a & b & c;
endmodule

module and4 (y, a, b, c, d);
      input a, b, c, d;
      output y;
      assign y = a & b & c & d;
endmodule

module or3 (y, a, b, c);
      input a, b, c;
      output y;
      assign y = a | b | c;
endmodule

module or4 (y, a, b, c, d);
      input a, b, c, d;
      output y ;
      assign y = a | b | c | d;
endmodule
```

# Verilog source code (cg_and_cp and 4-bit lookahead carry generator)

```
module cg_and_cp (g, p, s, x, y);
      input x, y;
      output g, p, s;
      and (g, x, y);
      xor (p, x, y);
      xor (s, p, c);
endmodule

module four_bit_lookahead_carry_generator (c_out, p_block, g_block, p, g, c_in);
      input [3:0] p;
      input [3:0] g;
      input c_in;
      output [2:0] c_out;
      output p_block;
      output g_block;
      wire [8:0] p_g_and_c;

      and (p_g_and_c[0], c_in, p[0]);
      or  (c_out[0], p_g_and_c[0], g[0]);

      and (p_g_and_c[1], p[1], g[0]);
      and3 and3_inst1 (p_g_and_c[2], p[1], p[0], c_in);
      or3  or3_inst1 (c_out[1], p_g_and_c[1], p_g_and_c[2], g[1]);

      and (p_g_and_c[3], p[2], g[1]);
      and3 and3_inst2 (p_g_and_c[4], p[2], p[1], g[0]);
      and4 and4_inst1 (p_g_and_c[5], p[2], p[1], p[0], c_in);
      or4 or4_inst1 (c_out[2], p_g_and_c[3], p_g_and_c[4], p_g_and_c[5], g[2]);

      and (p_g_and_c[6], p[3], g[2]);
      and3 and3_inst3 (p_g_and_c[7], p[3], p[2], g[1]);
      and4 and4_inst2 (p_g_and_c[8], p[3], p[2], p[1], g[0]);
      or4 or4_inst2 (g_block, p_g_and_c[6], p_g_and_c[7], p_g_and_c[8], g[3]);

      and4 and4_inst3 (p_block, p[3], p[2], p[1], p[0]);
endmodule
```

# Verilog source code (4-bit CLA adder and testbench)

```
module four_bit_CLA_adder (G, P, S, X, Y, C_in);
       output G, P;
       output [3:0] S;
       input  [3:0] X;
       input  [3:0] Y;
       input C_in;
       wire [3:0] g;
       wire [3:0] p;
       wire [3:1] c;

       cg_and_cp inst1 (g[0], p[0], S[0], X[0], Y[0], C_in);
       cg_and_cp inst2 (g[1], p[1], S[1], X[1], Y[1], c[1]);
       cg_and_cp inst3 (g[2], p[2], S[2], X[2], Y[2], c[2]);
       cg_and_cp inst4 (g[3], p[3], S[3], X[3], Y[3], c[3]);
       four_bit_lookahead_carry_generator carry_generator_4 (c, P, G, p, g, C_in);
endmodule

module tb ();
       wire [15:0] S;
       wire G, P;
       wire [68:0] sum;
       reg  [15:0] X, Y;
       reg  C_in;
       reg [67:0] A, B;

       sixteen_bit_CLA_adder dut (G, P, S, X, Y, C_in);
       sixty_eight_bit_CLA_adder dut1 (sum, A, B);
     initial
       begin
         //$monitor("G = %b  P = %b  S = %16b  X = %16b  Y = %16b  C_in = %b" ,G, P, S, X, Y, C_in);
         $monitor("sum = %d  B = %d  A = %d" , sum, A, B);
         #5 assign A = 65535;assign B = 65535; assign C_in = 1'b0;
         #5 assign A = 65535;assign B = 0; assign C_in = 1'b1;
         #5 assign A = 0;assign B = 65535; assign C_in = 1'b1;
         #5 assign A = 1065535;assign B = 1; assign C_in = 1'b0;
       end
endmodule
```

# Verilog source code (16-bit CLA adder and 64-bit CLA adder)

```
module sixteen_bit_CLA_adder (G, P, S, X, Y, C_0);
    output G, P;
    output [15:0] S;
    input  [15:0] X, Y;
    input C_0;
    wire [3:0] g, p;
    wire [2:0] c;

    four_bit_CLA_adder inst0 (g[0], p[0], S[3:0],   X[3:0],   Y[3:0],   C_0);
    four_bit_CLA_adder inst1 (g[1], p[1], S[7:4],   X[7:4],   Y[7:4],   c[0]);
    four_bit_CLA_adder inst2 (g[2], p[2], S[11:8],  X[11:8],  Y[11:8],  c[1]);
    four_bit_CLA_adder inst3 (g[3], p[3], S[15:12], X[15:12], Y[15:12], c[2]);
    four_bit_lookahead_carry_generator carry_generator_16 (c, P, G, p, g, C_0);
endmodule

module sixty_four_bit_CLA_adder (G, P, S, X, Y, C_0);
    output G, P;
    output [63:0] S;
    input  [63:0] X, Y;
    input C_0;
    wire [3:0] g, p;
    wire [2:0] c;

    sixteen_bit_CLA_adder inst0 (g[0], p[0], S[15:0],   X[15:0],   Y[15:0],   C_0);
    sixteen_bit_CLA_adder inst1 (g[1], p[1], S[31:16],   X[31:16],   Y[31:16],   c[0]);
    sixteen_bit_CLA_adder inst2 (g[2], p[2], S[47:32],  X[47:32],  Y[47:32],  c[1]);
    sixteen_bit_CLA_adder inst3 (g[3], p[3], S[63:48], X[63:48], Y[63:48], c[2]);
    four_bit_lookahead_carry_generator carry_generator_16 (c, P, G, p, g, C_0);
endmodule
```

# Verilog source code (2-bit lookahead carry generator and 68-bit CLA adder)

```
module two_bit_lookahead_carry_generator (C, P, G, C_0);
      input [1:0] P, G;
      output [1:0] C;
      input C_0;
      wire [2:0] g_p_and_c;

      and (g_p_and_c[0], P[0], C_0);
      or (C[0], g_p_and_c[0], G[0]);

      and (g_p_and_c[1], P[1], G[0]);
      and3 inst0 (g_p_and_c[2], P[1], P[0], C_0);
      or3 inst1 (C[1], g_p_and_c[1], g_p_and_c[2], G[1]);
endmodule

module sixty_eight_bit_CLA_adder (sum, X, Y);
      input [67:0] X, Y;
      output [68:0] sum;
      wire C_64;
      wire [1:0] G, P;
      sixty_four_bit_CLA_adder inst0 (G[0], P[0], sum[63:0], X[63:0], Y[63:0], 1'b0);
      four_bit_CLA_adder inst1 (G[1], P[1], sum[67:64], X[67:64], Y[67:64], C_64);
      two_bit_lookahead_carry_generator carry_generator_68 ({sum[68], C_64}, P, G, 1'b0);
endmodule
```

## Conclusion

- A 68-bit CLA adder was designed using 4-bit CLA adder as a building block in a multilevel lookahead configuration.
- Space complexity for the 68-bit CLA adder was found to be 526 gates.
- The critical path delay was found to be 12 $\Delta_g$ where $\Delta_g$ is the gate delay of one gate.
- Other possible implementations that could be explored are as follows:
  - A serial CLA adder using registers and multiplexers. Since, the multilevel carry lookahead adder is recursive, the 4-bit carry lookahead block can be redesigned to implement a serial CLA adder.
  - Another possible implementation of interest would be where two 4-bit adder blocks are merged into a larger overlapping adder. Such an adder would not require 4-bit CLA adder block separately in a 68-bit CLA adder and would be merged within the 64-bit CLA adder.

# References

- Computer System Architecture, Third Edition, by M. Morris Mano
- Computer Architecture, A Quantative Approach, Fifth Edition by Hennessy and Patterson
- www.nandtotetris.org
- Computer Arithmetic – Algorithms and Hardware Designs (2nd edition) by Behrooz Parhami