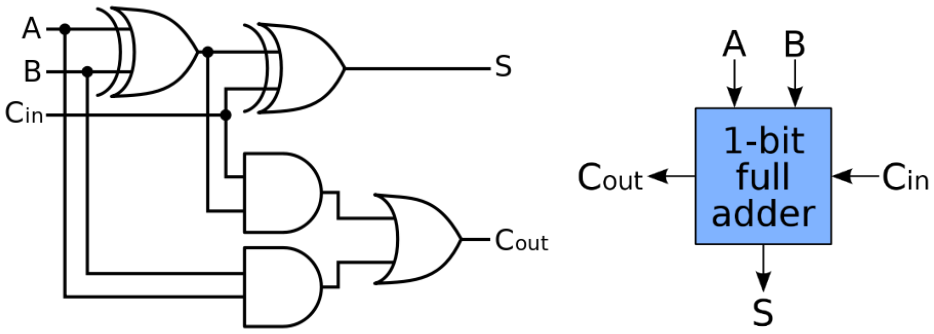
# Carry-Lookahead Adder 16-bits

## Full Adder

A complete adder is a digital circuit that calculates the addition of three bits: two significant bits and one input transport bit. It has three inputs **(A, B, and Cin)** and two outputs **(Sum and Cout).**

The output **Sum** is XOR of all three input bits: **Sum = A^B^Cin**

**Cout** represents an overflow in the next digit of an addition of several bits. Cout is generated by any of the following three conditions and is therefore the OR of three operations: **Cout=(A&B) | (B&Cin) | (A&Cin)**




## CLA4

The complete adder is used in the construction of the 4-bit carry **lookahead adder (CLA).** This arrangement allows sequential addition of multi-bit binary numbers, where the transport bit from one full adder is passed to the next, thus managing multi-bit binary addition correctly.

In the pretransport adder, each addition per bit eliminates dependence on the previous carry-out signal and requires the values of the two input operands to be used instead. It works by generating two new signals (P and G) for each binary rank depending on the state of the inputs.

- G - a signal to generate a bit of carry.
- P - a signal of propagation of one bit of carry.

For a multi-level carry-lookahead adder it is more advantageous to use:

- pi = ai ^ bi
- gi = ai & bi

---

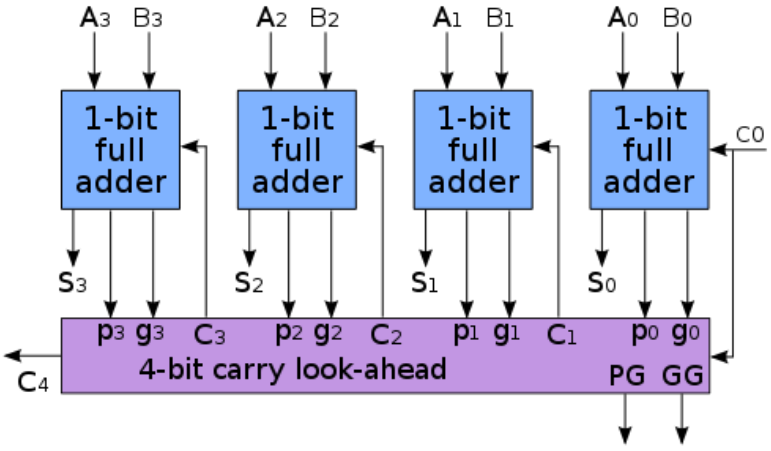
- c0 = c0
- c1 = g0 | (p0 & c0)
- c2 = g1 | (p1 & g0) | (p1 & p0 & c0)
- c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c0)
- c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) | (p3 & p2 & p1 & p0 & c0)

**Quick Transport Calculation (Carry):** The ci formula for each shipment uses logical operators to quickly determine whether there will be a shipment from one position to another, without waiting for the result of operations from previous positions. This reduces delays caused by sequential transport propagation in traditional adders.

**Latency reduction**: The calculation of each shipment depends on the calculation of the previous shipment, which creates significant latency for higher figures. In contrast, in the carry-lookahead adder, shipments are calculated in parallel, thus reducing the total time required to make the amount.

**Parallel Transport Evaluation**: The formulas for ci allow the evaluation of all possible shipments in parallel, using the results of the generated pre-calculations **(gi and pi).** This means that the time required to determine all shipments is approximately the same, regardless of the length of the binary number.

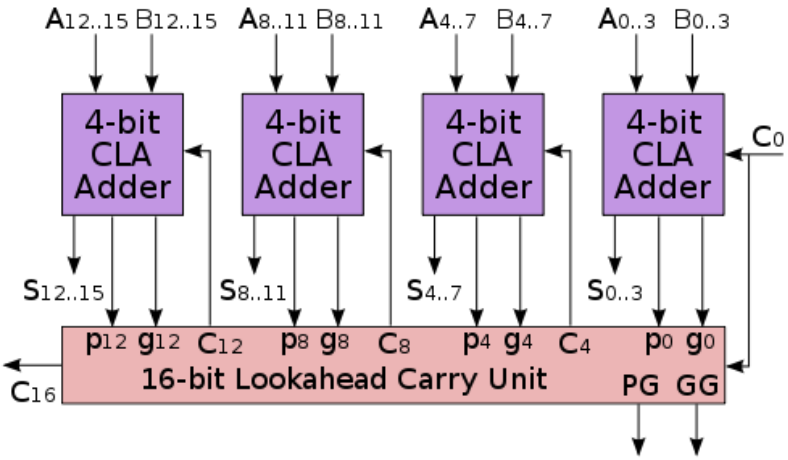
**Scalability and Efficiency**: Because of the way shipments are calculated, this type of adder scales efficiently for larger numbers, maintaining the speed advantage even for high-bit sums.




So for the sum of two numbers A and B, if the n bit in A, together with the n bit in B generates a G signal, it means that these bits generate one carry bit, and in the case of a P signal, the carry bit from n-1 (if any) is propagated by the n bits at position n+1. For a multi-level carry-lookahead adder it is more advantageous to use the XOR version.

## CLA16

The 16-bit implementation uses four 4-bit CLA blocks that are interconnected. Each 4-bit CLA block calculates the sums and transport signals for each 4-bit group. The generation and propagation signals for each group are used to determine transport values for the next 4-bit group. In a 4-bit CLA block, generation (G) and propagation (P) signals:




- Gi is true if and only if the ith group of bits generates a transport bit, that is, both group inputs are 1.
- Pi is true if and only if the ith group of bits can propagate a transport bit, at least one of the inputs is 1.

These signals are used to calculate all transport signals for subsequent groups using the equations above. The basic equations for an ACL16 adder are the same as for an CLA4:

- Pi = ai XOR bi for propagation.
- Gi = ai AND bi for generation.
- Ci+1 = Gi OR (Pi AND Ci) for calculating the transport bit to the next group.

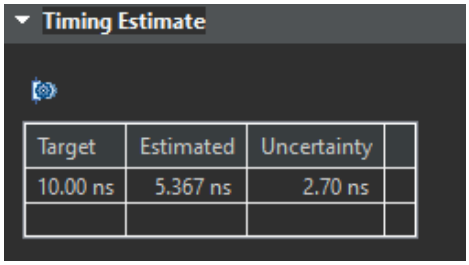
The specific implementation for a 16-bit multi-level CLA adder will divide inputs into four groups of 4 bits. Each group calculates the signals Gi and Pi, and these signals are then used to determine the transport bit for the next group. Typically, implementation will take the form of:

Each CLA4 calculates a group of 4 bits:

- **Group 1**: **Sum[3:0] and Cout0** using **Cin** as the input transport bit.
- **Group 2**: **Sum[7:4] and Cout1** using **Cout0** as the input transport bit.
- **Group 3**: **Sum[11:8] and Cout2** using **Cout1** as the input transport bit.
- **Group 4**: **Sum[15:12] and Cout** (final transport bit) using **Cout2** as the input transport bit.

Such an implementation can be accomplished without loops, meaning that each transport bit is calculated individually by direct logical equations, instead of using repetitive structures such as for loops. This is an effective method for hardware, as it significantly reduces the delay in the transport chain.

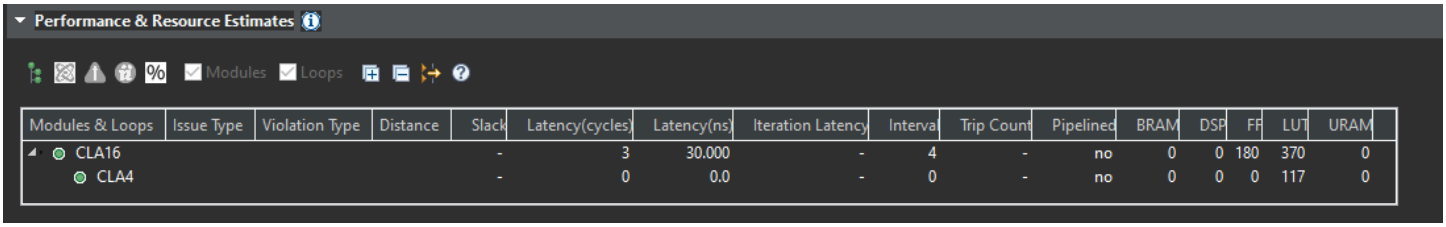
**Product family**: The family of the target FPGA device , which in this case is "artix7.
**Target device**: The specific FPGA model the synthesis uses is. "xc7a100t-csg324-1" identifies a specific Xix-7 FPGA chip from Xilinx.




**Target**: The target time is the design constraint for the clock period, which is the maximum time allowed for a signal to propagate by the combinational logic of a single clock domain between two sequential elements (such as flip-flops). In this case, it is set to 10.00ns, corresponds to a clock frequency of 100 MHz (1/10 ns).

**Estimated Timing:** This is the measured real time, estimated at 5.36ns. Displays the average or average time that the operation took during measurement.

**Uncertainty**: Listed as 2.70ns, refers to statistical uncertainty or the possible range of error in measuring time.




**Latency(s):** For a purely combinational circuit, **cycle latency is usually zero,** since there is no clock cycle dependence for the output to be generated from inputs, except gate delays.

**Latency (ns):** This represents the actual delay, from when inputs change to when outputs stabilize, reflecting those changes. This delay is due to the physical properties of the logic gates used in the circuit and is independent of any clock cycle.

**Iteration Latency:** Not applicable to combinational circuits because there are no iterations or loops.

**Interval:** For combinational logic, the interval is usually zero, since combinational circuits do not have a clock cycle that would limit the rate at which they accept new inputs.

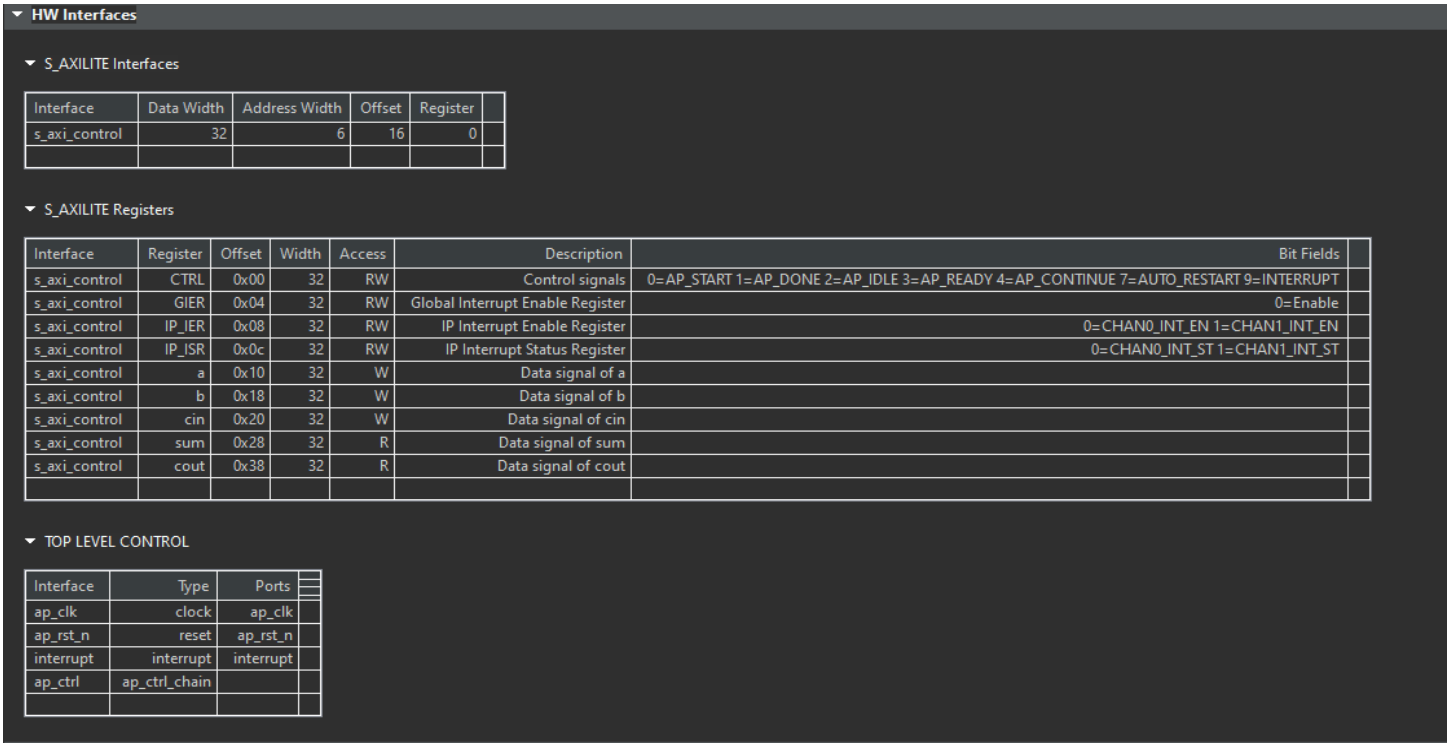
**Trip Count:** Again, it does not apply to combinational circuits as there are no loops.

**Pipelined:** This would usually be "no" for combinational circuits because pipes involve dividing a process into stages, which is a concept applied to sequential, not combinational, circuits .

**BRAM, DSP, FF, LUT, URAM:** These columns indicate the resources used by the circuit. For combined circuits: BRAM (Block RAM) and URAM (Ultra RAM) are types of memory and usually would not be used by purely combinational logic.

**FF (Flip-Flops):** are used in sequential logic to store state information. A purely combinational circuit **should not use flip-flops,** sometimes synthesis tools can use them for certain optimizations to maintain constant values.

**LUTs (Look-Up Tables):** are used in FPGAs to implement logical functions. The number of LUTs used will give an indication of the complexity of combinational logic.



HW Interfaces

S_AXILITE Interfaces

| Interface | Data Width | Address Width | Offset | Register |
|---|---|---|---|---|
| s_axi_control | 32 | 6 | 16 | 0 |

S_AXILITE Registers

| Interface | Register | Offset | Width | Access | Description | Bit Fields |
|---|---|---|---|---|---|---|
| s_axi_control | CTRL | 0x00 | 32 | RW | Control signals | 0=AP_START 1=AP_DONE 2=AP_IDLE 3=AP_READY 4=AP_CONTINUE 7=AUTO_RESTART 9=INTERRUPT |
| s_axi_control | GIER | 0x04 | 32 | RW | Global Interrupt Enable Register | 0=Enable |
| s_axi_control | IP_IER | 0x08 | 32 | RW | IP Interrupt Enable Register | 0=CHAN0_INT_EN 1=CHAN1_INT_EN |
| s_axi_control | IP_ISR | 0x0c | 32 | RW | IP Interrupt Status Register | 0=CHAN0_INT_ST 1=CHAN1_INT_ST |
| s_axi_control | a | 0x10 | 32 | W | Data signal of a | |
| s_axi_control | b | 0x18 | 32 | W | Data signal of b | |
| s_axi_control | cin | 0x20 | 32 | W | Data signal of cin | |
| s_axi_control | sum | 0x28 | 32 | R | Data signal of sum | |
| s_axi_control | cout | 0x38 | 32 | R | Data signal of cout | |

TOP LEVEL CONTROL

| Interface | Type | Ports |
|---|---|---|
| ap_clk | clock | ap_clk |
| ap_rst_n | reset | ap_rst_n |
| interrupt | interrupt | interrupt |
| ap_ctrl | ap_ctrl_chain | |

**SW I/O Information**

Top Function Arguments

| Argument | Direction | Datatype |
|---|---|---|
| a | in | ap_uint<16> |
| b | in | ap_uint<16> |
| cin | in | bool |
| sum | out | ap_uint<16>& |
| cout | out | bool& |

SW-to-HW Mapping

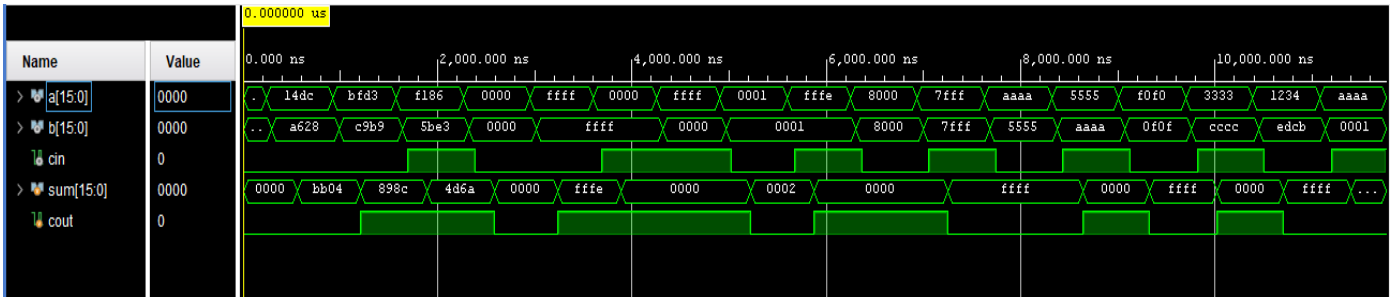
| Argument | HW Interface | HW Type | HW Info |
|---|---|---|---|
| a | s_axi_control | register | name=a offset=0x10 range=32 |
| b | s_axi_control | register | name=b offset=0x18 range=32 |
| cin | s_axi_control | register | name=cin offset=0x20 range=32 |
| sum | s_axi_control | register | name=sum offset=0x28 range=32 |
| cout | s_axi_control | register | name=cout offset=0x38 range=32 |

**Pragma Report**

Valid Pragma Syntax

| Type | Options | Location | Function |
|---|---|---|---|
| interface | ap_none port=a | Adder_16Bits/Adder_16Bits.cpp:57 | cla16 |
| interface | ap_none port=b | Adder_16Bits/Adder_16Bits.cpp:58 | cla16 |
| interface | ap_none port=cin | Adder_16Bits/Adder_16Bits.cpp:59 | cla16 |
| interface | ap_none port=sum | Adder_16Bits/Adder_16Bits.cpp:60 | cla16 |
| interface | ap_none port=cout | Adder_16Bits/Adder_16Bits.cpp:61 | cla16 |
| interface | ap_none port=return | Adder_16Bits/Adder_16Bits.cpp:62 | cla16 |

- **port=a and port=b**: Two operands for addition operation. Since ports are marked with ap_none pragma, this indicates that data is retrieved directly without a specific interface protocol. These will likely be mapped directly to FPGA pins or internal links.
- **port=cin**: The transport input bit (carry-in) to add. Single input signal indicating whether there is a carry from a previous addition to be counted towards the current assembly.
- **port=sum**: This is the output port where the function places the result of the 16-bit addition. It will contain the sum of the two operands a and b, plus any carry-in from the cin, after completion of the addition operation.
- **port=cout**: It is the carry-out bit after adding. It indicates whether the result of the 16-bit addition generated a carry that exceeds the length of the operand and that could be used in a later addition if cla16 is part of a chain addition.
- **port=return**: Indicates that the function does not return a value through a normal mechanism for returning values in C++, but rather that output is handled through interfaces defined by HLS.




The brief logic behind the signals for each test:

**For random tests**, each set of a, b and cin, the adder calculates the sum and determines if there is an carry output. The sum of 0x14DC and 0xA628 without a carryover produces an amount of 0xBB04 and no carryover.

**The boundary conditions test** checks extremes of input space, such as all zeros, all, and cases just before and immediately after the adder spins from its maximum to minimum value. For example, 0xFFFF + 0xFFFF results in 0xFFFE with a transfer, the result exceeds the maximum value of 16 bits.

**Model tests** use specific binary models to verify adder behavior in scenarios where bit patterns can influence transport propagation. Alternative models such as 0xAAAA and 0x5555 test how shipments are generated and propagated across the entire width of the mark-up.