(b) [25 points] Fill in the blanks for Peregrine G-Class XTreme.

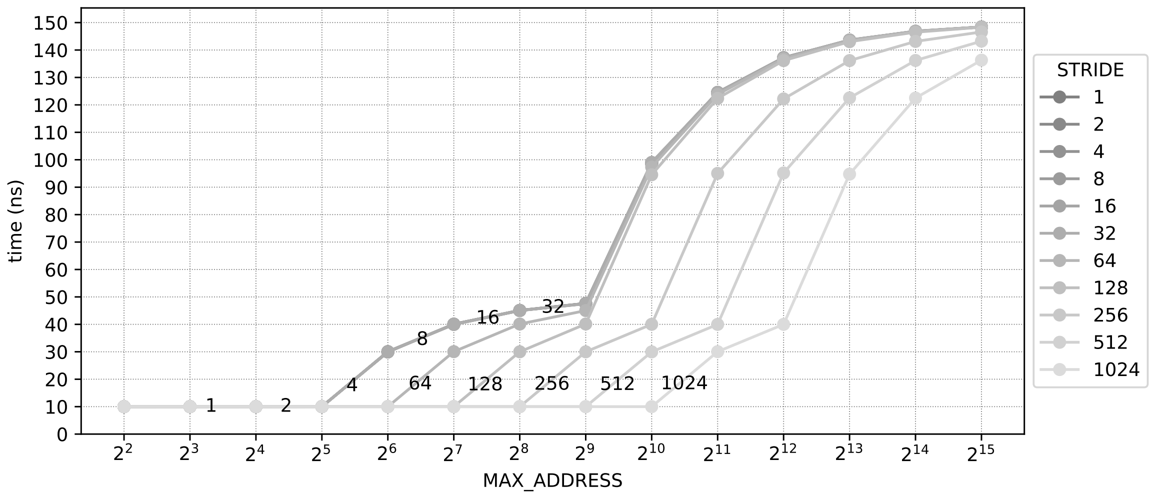


Figure 2: Execution time of the test code on CPU B for various values of *STRIDE* and *MAX_ADDRESS*. *STRIDE* values are labeled on curves themselves for clarity. Note that the curves for strides 1, 2, 4, 8, 16, and 32 overlap in the figure.

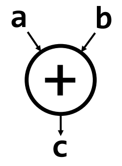
Table 2: Fill in the following table for CPU B (Peregrine G-Class XTreme)

System Parameter	**CPU B: Peregrine G-Class XTreme**			
	L1	L2	L3	DRAM
Cache Line Size (B)	N/A	N/A	N/A	N/A OR X
Cache Associativity	1	4	X	X
Total Cache Size (B)	32	512	X	X
Access Latency (ns) [1]	10	40	X	100

[1] DRAM access latency means the latency of fetching the data from DRAM to L3, *not* the latency of bringing the data from the DRAM all the way down to the CPU. Similarly, L3 access latency means the latency of fetching the data from L3 to L2. L1 access latency is the latency to bring the data to the CPU from the L1 cache.

11 Dataflow Meets Logic [35 points]

We often use the "addition node":



to represent the addition of two input tokens. If we think of the tokens as binary numbers, we can model a simple logic circuit using dataflow graphs.[1] Note that a token can be used as an input to only *one* node. If the same value is needed by more than one node, it first should be replicated using one or more copy nodes, and then each copied token can be supplied to one node only.

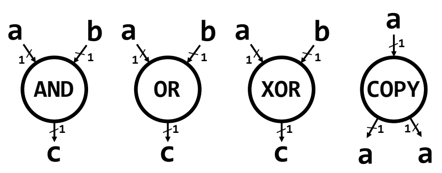
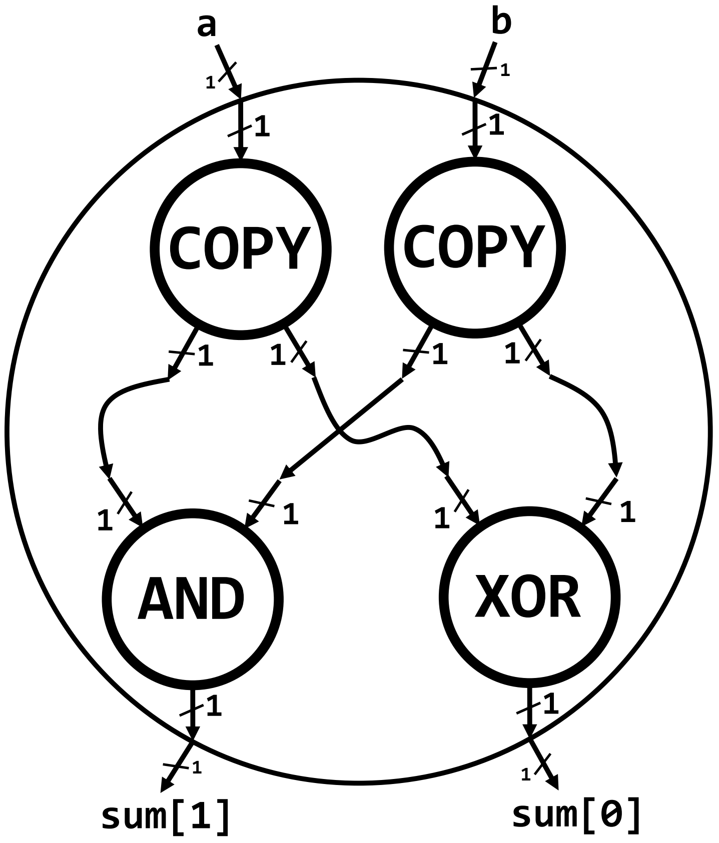


Figure 3: Dataflow nodes of basic bitwise operations allowed in Part (a).

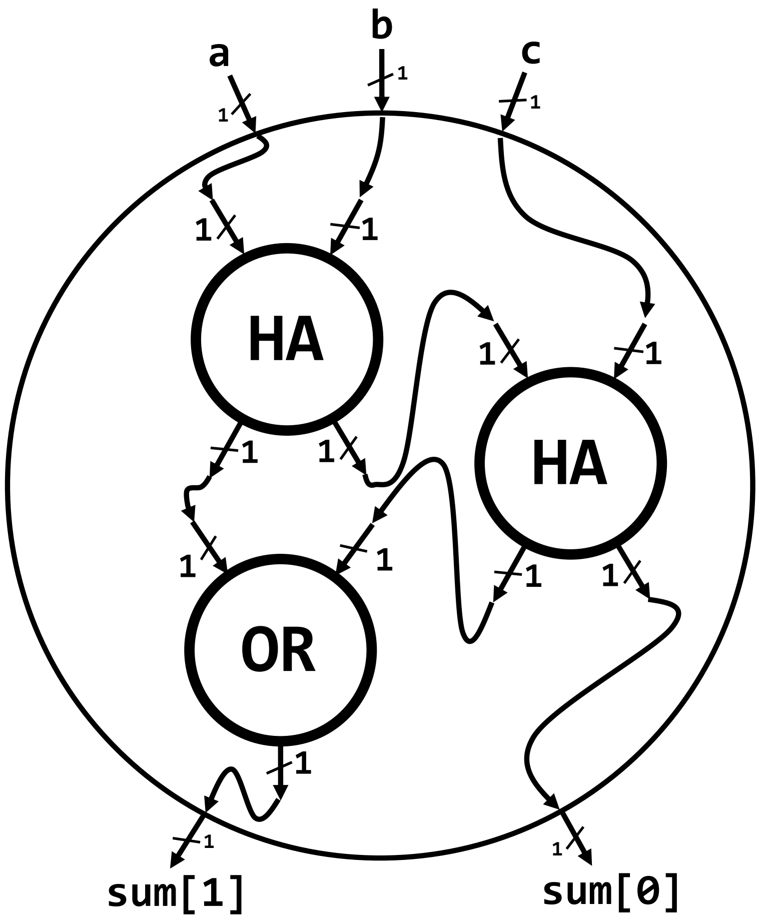
(a) [5 points] Implement the single-bit binary addition of two "1-bit" input tokens a and b as a dataflow graph using *only* 2-input {AND, OR, XOR} nodes and COPY nodes if necessary (illustrated in Figure 3). Fill in the internal implementation below, where inputs and outputs (labeled with their corresponding bit-widths) have been provided:



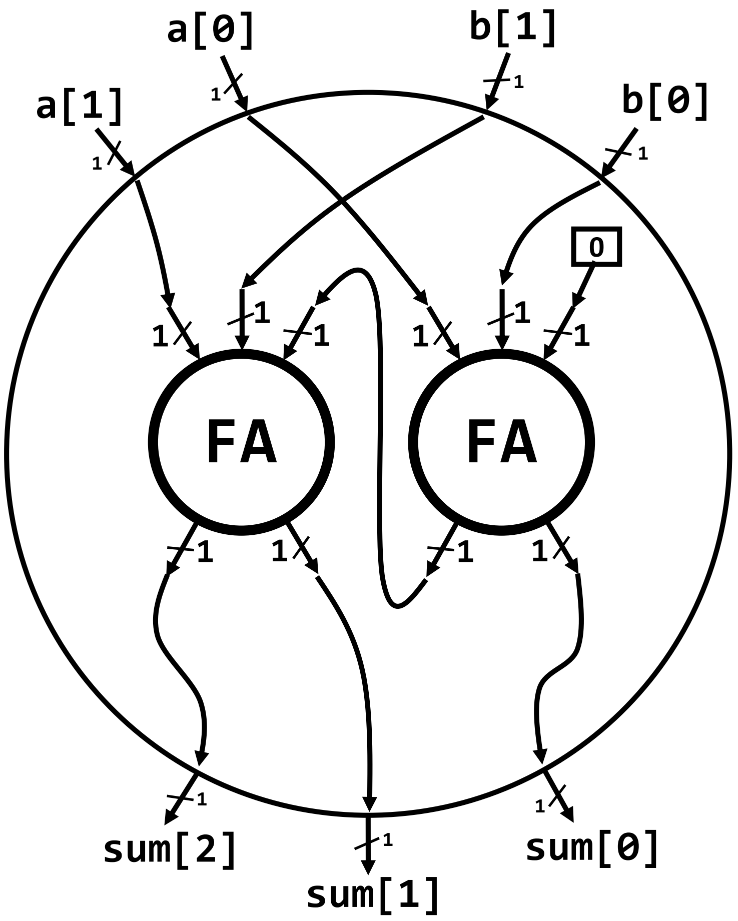
[1]Note: this is not an accurate electrical model of a circuit. Instead, the dataflow analogy is best thought of in terms of the desired flow of *information* rather than physical phenomena.

(b) [5 points] You may recognize the node we designed in part (a) as a model for a so-called "half-adder (HA)", which is not very useful by itself since it is only useful for adding 1-bit input tokens. In order to extend this design to perform binary addition of 2-bit input tokens `a[1:0]` and `b[1:0]`, the `sum[1]` token from half-adding `a[0]` and `b[0]` will have to act as an input token for *another* half-adder node used for adding `a[1]` and `b[1]`. This results in a 3-input adder called a "full-adder (FA)".

Fortunately, we can implement a full-adder (FA) using half-adders (HA) (i.e., the node we designed in part (a). Implement the full-adder using a *minimum* number of half-adders and *at most* 1 additional 2-input {AND, OR, XOR} node.



(c) [5 points] The full-adder (FA) is a versatile design that can be used to implement *n*-bit addition. Show how we might use it to implement 2-bit binary addition of two input tokens `a[1:0]` and `b[1:0]`. Use only a *minimum* number of full-adders (i.e., the dataflow node you designed in Part 2). *Hint: you may use constant input tokens if necessary.*


a[0]
b[1]
a[1]
b[0]
0
1
FA
FA
sum[2]
sum[1]
sum[0]