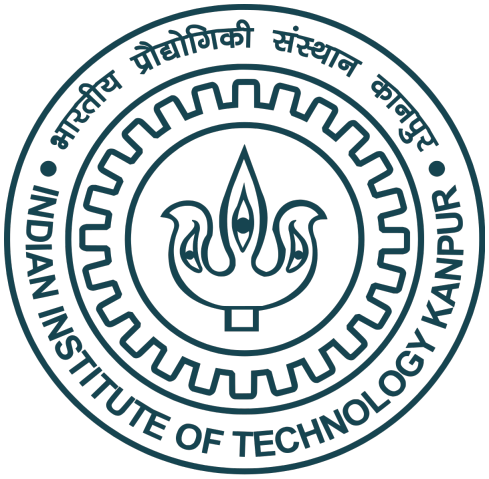
CS220A: Computer Organization



Assignment 2 Report

Aditya Tanwar	Akhil Agrawal	Suket Raj
200057	200076	201013

February 12, 2022

Question 1

Write detailed description of 8 bit Carry Look-Ahead Adder and its working with the proper circuit diagram in a PDF file. Then write the Verilog code module to implement Carry Look-Ahead Adder. Now, add a test bench to test the Carry Look-Ahead Adder. Make sure to display your inputs, sum, and carry out. Your test bench must have fifteen different inputs. Put five-time unit delay between consecutive inputs.

Motivation: Carry Look-Ahead Adder ($\mathcal{CLA}$-$\mathcal{A}$) is a modification of Ripple Carry Adder ($\mathcal{RC}$-$\mathcal{A}$) which aims to remove the time delay due to carry propagation.

Suppose an $\mathcal{RC}$-$\mathcal{A}$ performs addition of *2* n bit numbers. Then, there will be n blocks each having *2* bits as input which are to be added. But, any block needs to receive (and wait for) the *carry-in* from the previous block to complete its own addition. Therefore, the i^{th} block has to wait for the previous $(i-1)$ blocks before calculating its result. This causes a considerable delay *($\mathcal{O}(i)$ gate delays for the i^{th} bit, to be precise).*

In a $\mathcal{CLA}$-$\mathcal{A}$, we introduce a more complex hardware because of which we are able to generate the *carry-out* as a function of inputs A, B and just the inital *carry-in* only (which is different from $\mathcal{RC}$-$\mathcal{A}$ because *carry-out* was a function of inputs A, B and *carry-in* there).
So, at maximum, *2* gate delays are there for calculating a carry, and after the carries have been calculated, the resultant sum can be calculated in parallel.

Concept: For simplifying the logic and notation, we use two more variables, one for *generation* of carries and one for *propagation* of carries.
For any block in the adder, carry can either be *generated* as a result of input bits to that block, or, carry can be *propagated* because of a previous *carry* input to the block.

For any block i, we define the following variables:

a_i : First input bit	g_i : Generated carry
b_i : Second input bit	p_i : Propagated carry
c_i : Input carry	c_{i+1} : Resultant output carry

We introduce and elaborate some notations below-

- We use *bit-wise* "AND" interchangeably with $\wedge$ and $\cdot$. Similarly, with *bit-wise* "OR", $\vee$ and $+$. We denote *bit-wise* "XOR" with $\oplus$.
- *Generating* carry: If both the input bits to the adder block are 1, then a carry will be generated. Essentially, we take *bit-wise* "AND" for g_i,

$$g_i = a_i \cdot b_i$$

- *Propagating* carry: If at least one of the input bits to the adder block is 1, and there is an input carry from previous block, then the carry will be propagated. Essentially, we take *bit-wise* "OR" for p_i,

$$p_i = a_i + b_i$$

- Either a carry is generated from the inputs, or it is propagated from the earlier bit, so for carry-out, we have:

$$c_{i+1} = g_i + p_i \cdot c_i$$

- Sum: Finally, we have all the input bits available, along with the respective carries, so the i^{th} bit of the result can be calculating simply by taking the *bit-wise* "XOR" of those three:

$$s_i = a_i \oplus b_i \oplus c_i$$

- In $\mathcal{CLA}$-$\mathcal{A}$, we make benefit of these notations and expand the expressions for c_i. We have derived some expressions and have written the end results for the others, relevant for a four-bit $\mathcal{CLA}$-$\mathcal{A}$-

$$c_1 = g_0 + p_0 \cdot c_0$$

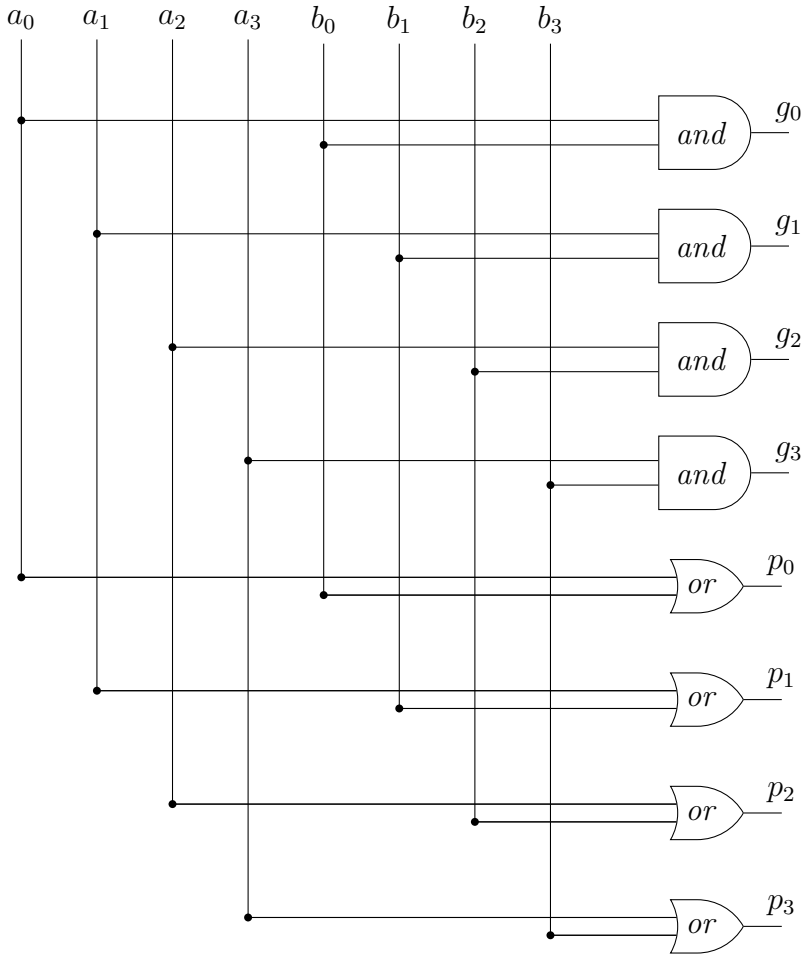
$$c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$$

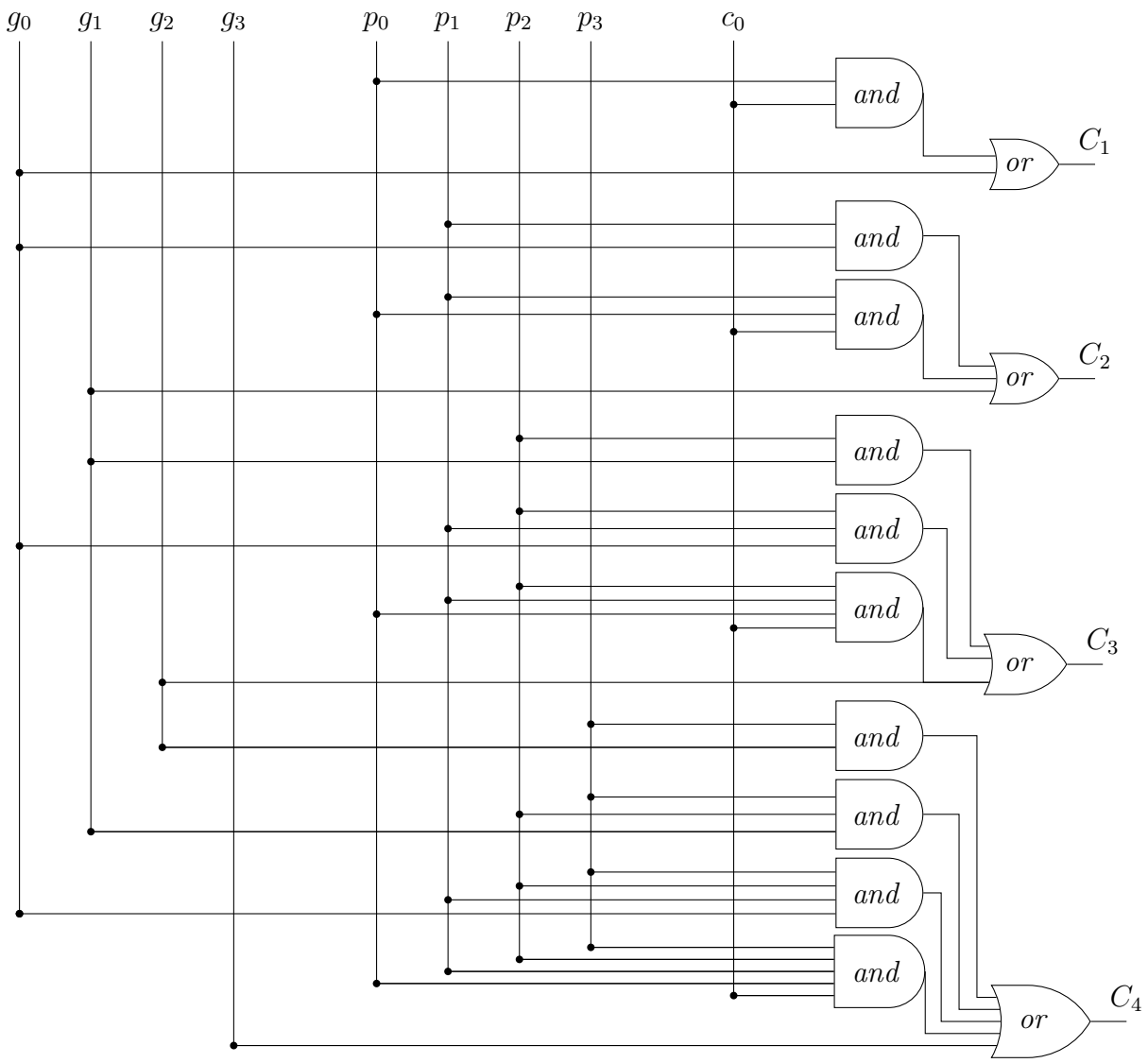
We expand the formulae for c_3 and c_4 in a similar fashion, with the end result written below-

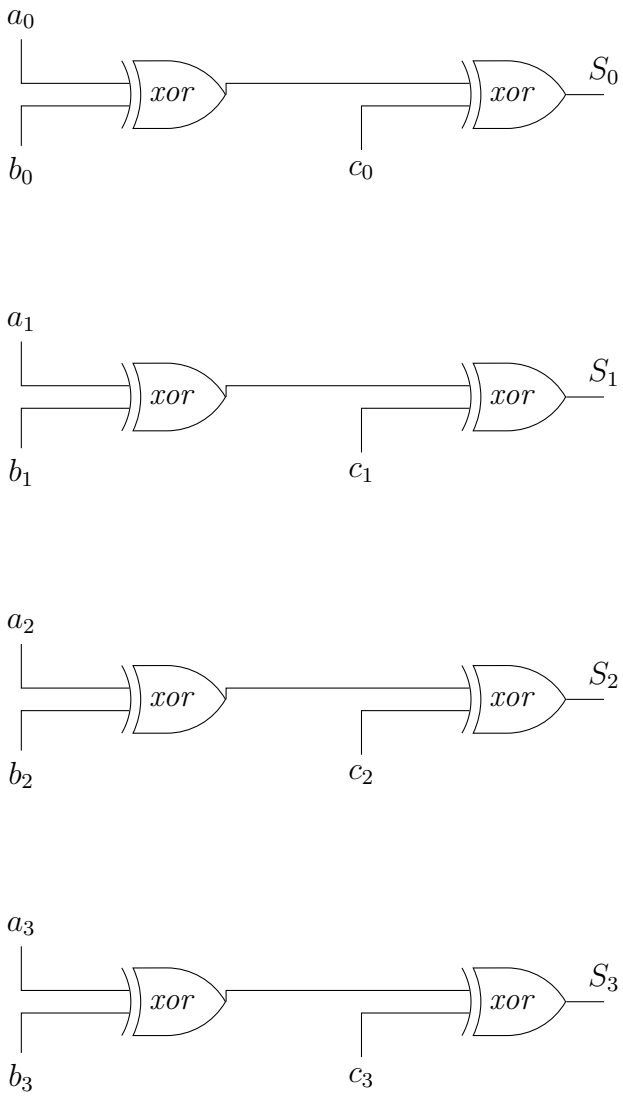
$$c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

$$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

Circuits: Due to the sheer size of the complete circuit, we have broken it down into smaller, multiple parts for ease in representation-




g_0
g_1
g_2
g_3
p_0
p_1
p_2
p_3
c_0
and
or
C_1
C_2
C_3
C_4



Implementation : In the code, we have grouped continuous chunk of *4-bits* together. So, for *8-bit* numbers, we have two chunks of *4-bits*, with the second part (part with more significant bits) having to wait for the first part (part with lesser significant bits) for a carry-in, but the *4-bits* in the same group are operated on concurrently.

Result: Although the asymptotic complexity for addition of *2 n-bit* numbers remain the same in both $\mathcal{RC}$-$\mathcal{A}$ and $\mathcal{CLA}$-$\mathcal{A}$ (both $\mathcal{O}(n)$), $\mathcal{CLA}$-$\mathcal{A}$ improves on the *"constant"* factor by using this trick by *4* times (when 4-bits are grouped together), because an $\mathcal{RC}$-$\mathcal{A}$ processes at most *1* bit at any point of time. Hence, the improvement over $\mathcal{RC}$-$\mathcal{A}$.

Question 2

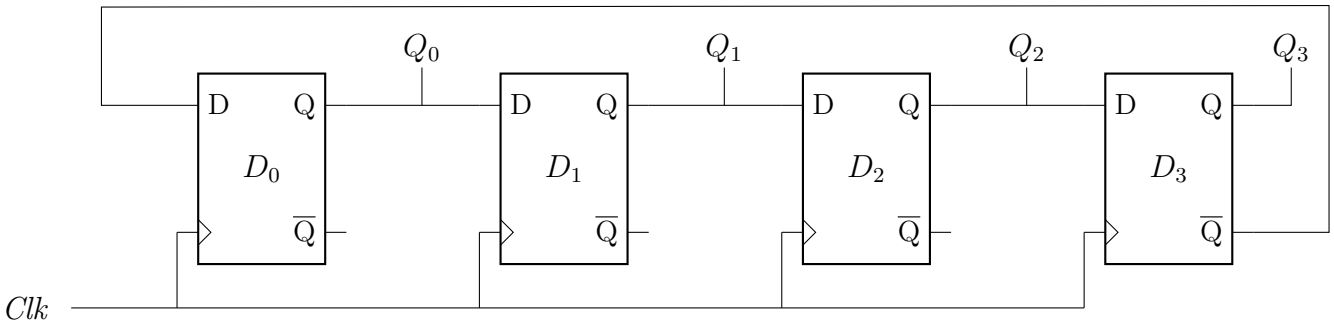
Write detailed description of 8-bit Johnson Counter and its working with the proper circuit diagram and truth table in a PDF file. Then write the Verilog code module to implement 8-bit Johnson Counter. Now, add a test bench to test all the states of the 8-bit Johnson Counter.

Shift Registers : Shift Registers comprise of *D Flip-Flops*, and can be used for reading data bit-wise in a sequential order. This is because each D Flip-Flops can correspond to a bit in the data stream. There are four different modes of operations of Shift Registers, based on the flow of input/ output, whether it is *Parallel* or *Serial*. A data bit is read on the edge of a clock (usually positive edge). This is inferrable from the fact that D-type Flip-Flops are used.

Ring Counters : Ring Counters build upon the concept of Shift Registers, and use feedback to hold data in D Flip-Flops. This is realized by taking feedback from the output of the last Flip-Flop. Essentially, at the (generally positive) edge of the clock, each flip-flop shifts its bit onto the next flip-flop, and since the last flip-flop does not have a following flip-flop, it unloads its data bit into the first one. Thus, the data is circulated among the flip-flops, in a manner reminiscent of a ring, hence the name.
An important limitation of Ring Counters is that for a given input (of say, n bits), they use at most n states, when, in theory, there are 2^n states possible due to the n bits available. But, this does have an interesting consequence in that these Counters can be used to realise $\mod n$ counters.

Johnson Counter : Johnson Ring Counters take the concept one step further by shifting the complement of the data bit stored in the last flip-flop onto the first one. As a consequence, the data stored in the Counter is complemented *1* bit at a time.
As is with Ring Counters, Johnson Counters do not exhaustively use all possible states, but they do improve and go on to use at most $2n$ states for an n bit input and hence, they can be used to realise $\mod (2n)$ counters.

Circuits: The circuit for a *4-bit Johnson Counter*-



Implementation : In the implementation, we have skipped the use of *D Flip-Flops* in favour of cleaner and easier to read code.

References

- Electronics Tutorials

- Wikipedia