# INDIAN INSTITUTE OF ENGINEERING SCIENCE AND TECHNOLOGY, SHIBPUR

Howrah, West Bengal, India - 711103

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY


Computer Architecture & Organization Laboratory [CS2272]

## LAB REPORT

*SUBMITTED BY*

**NAME - GOURAV KUMAR SHAW**
**ENROLLMENT ID. - 2020CSB010**
**SECTION - Gx**

**4th Semester, Academic Year: 2021-2022**

# Contents

# Experiment No: 1

## Organization of (m × d) memory with latches

### 1.1 Objective

To design memory module with the features of Read/Write, using RS latch, considering
(a) m = 1; d = 1
(b) m = 2; d = 1
(c) m = 2; d = 2, using the design resulted out of (b)

### 1.2 Theoretical Basis

A $2^m$ X d Memory Module can be drawn as follows

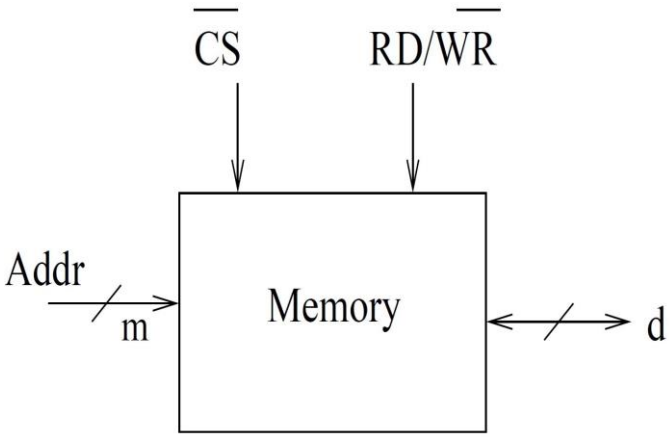
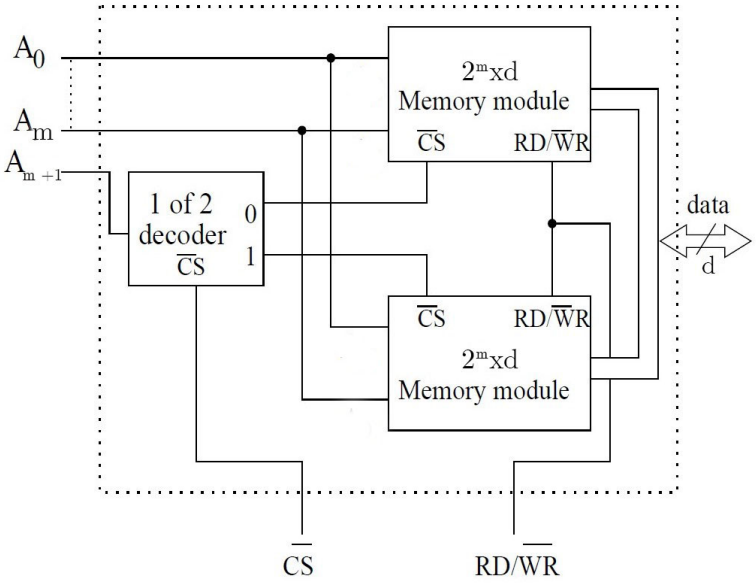
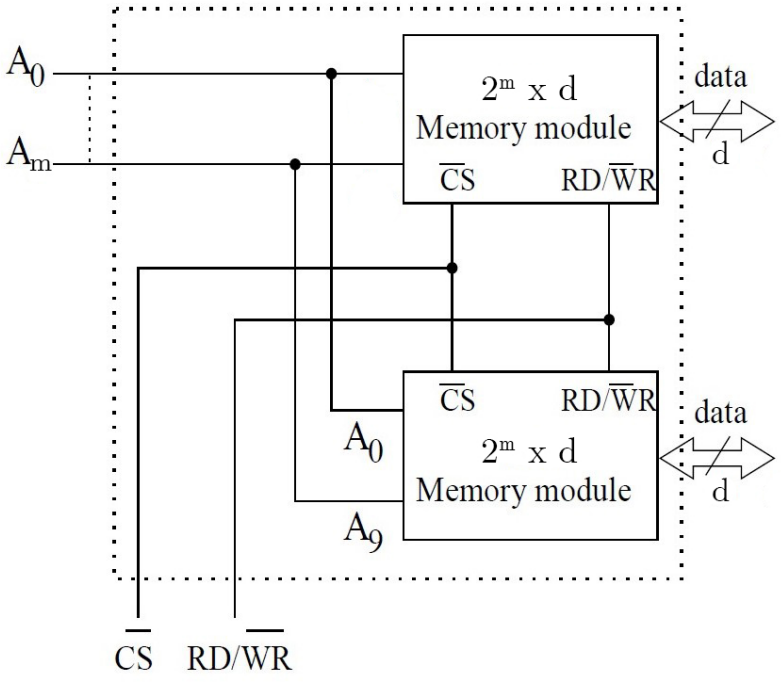



Figure 1: A m X d Memory Module

Here CS is Chip Select and RD/WR' is Read or Write input.
Given a $2^m$ X d Memory Module, we can make $2^{m+1}$ X d and $2^m$ x2d Memory Modules as follows




(a)A $2^{m+1} \times d$ Memory Module




(b) A $2^{m} \times 2d$ Memory Module

Figure 2: Expansion of Memory Modules

## 1.3 Task

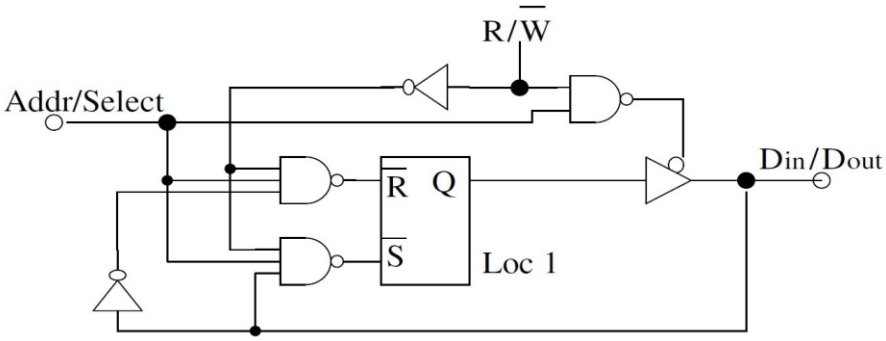
To Realize Circuit of:
(a) 1 bit 1 word memory
(b) 1 bit 2 word memory
(c) 2 bit 2 word memory

## 1.4 Major Components/Modules Used

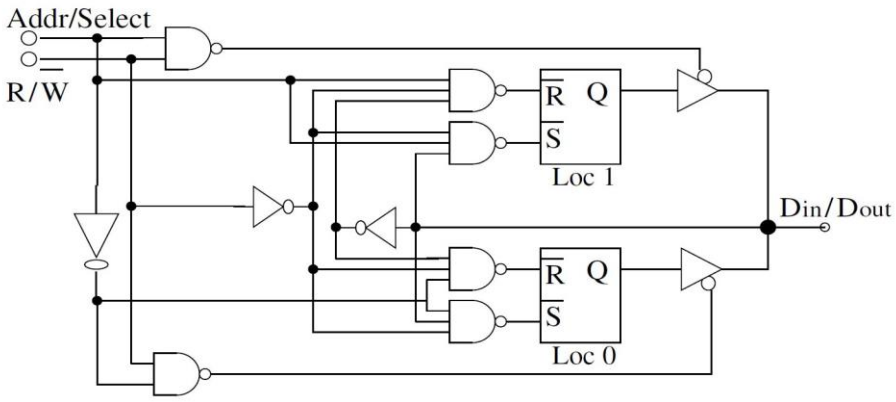
1. RS Flip Flop
2. Tri-state logic
3. Triple-input NAND Gate
4. Inverter

## 1.5 Design

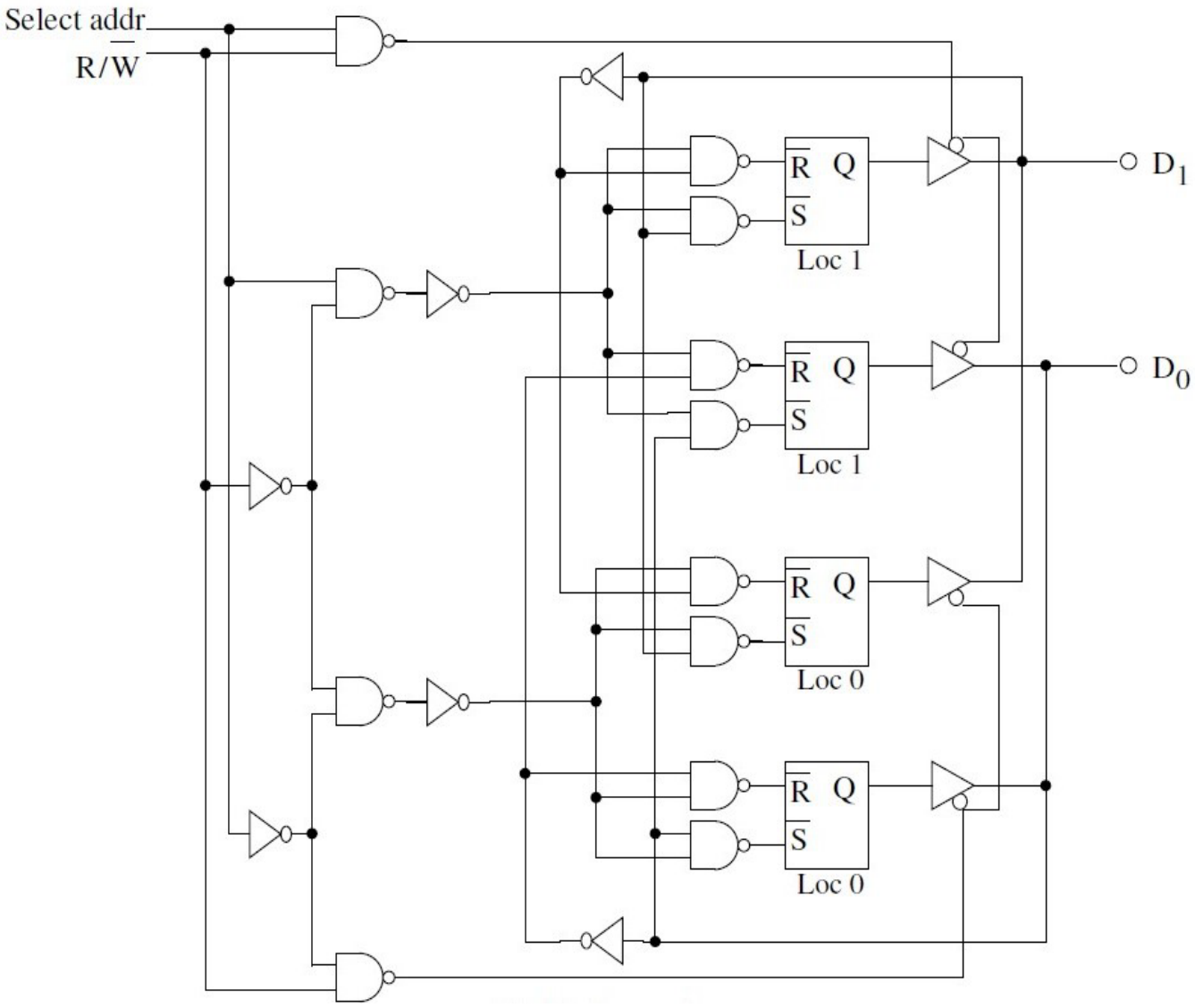
Following the above Ideas, we construct required memory modules as follows




(a) 1 bit 1 word memory



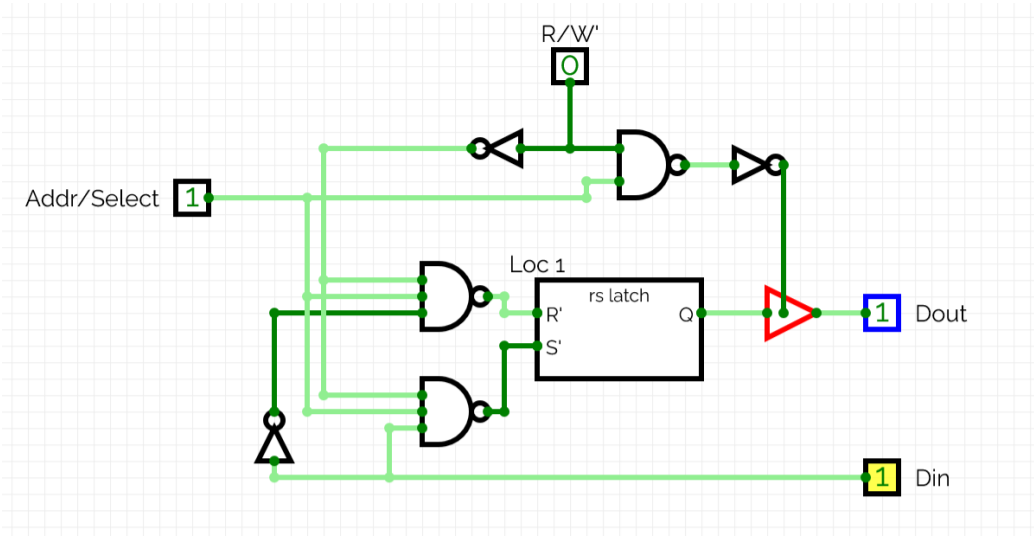

(b) 1 bit 2 word memory



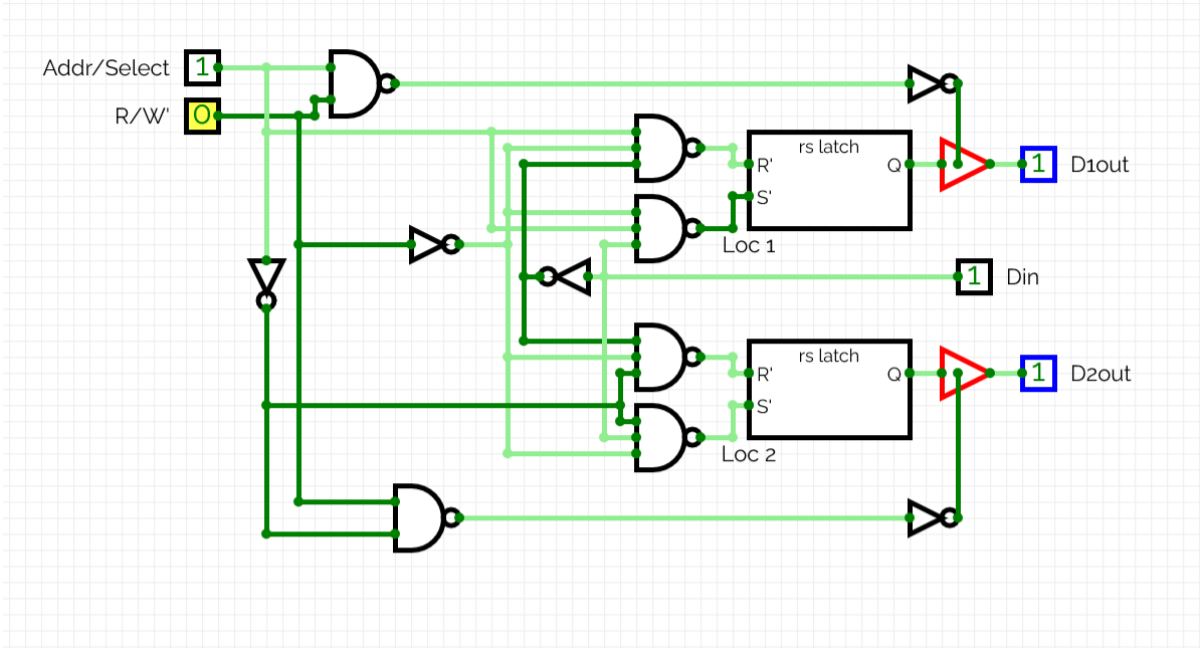

(c) 2 bit 2-word memory

Figure 3: Design of Memory

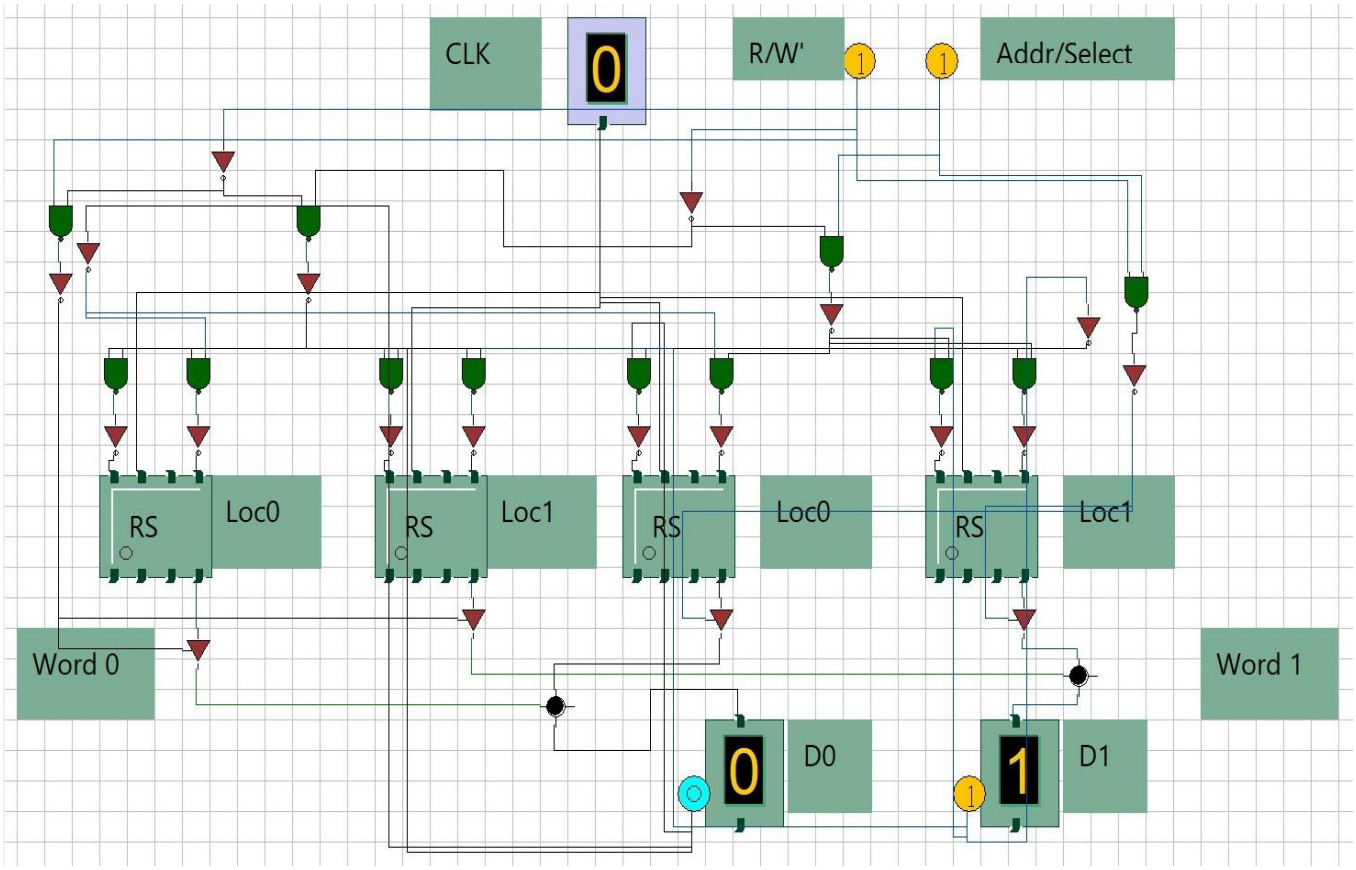
## 1.6 Circuit Diagram




(a) 1 bit 1 word memory




(b) 1 bit 2 word memory

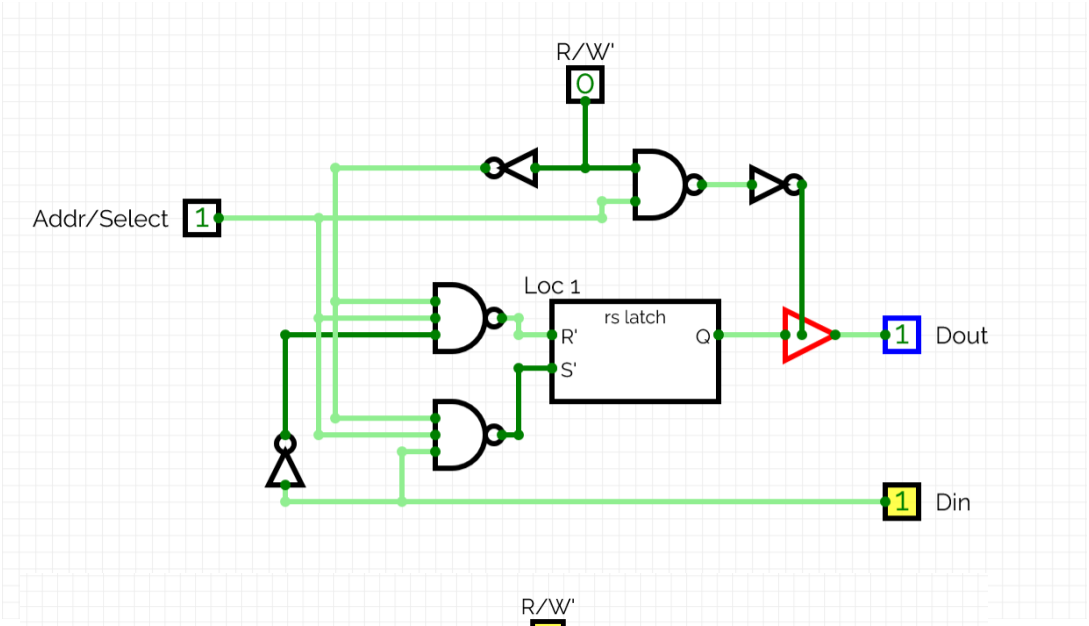


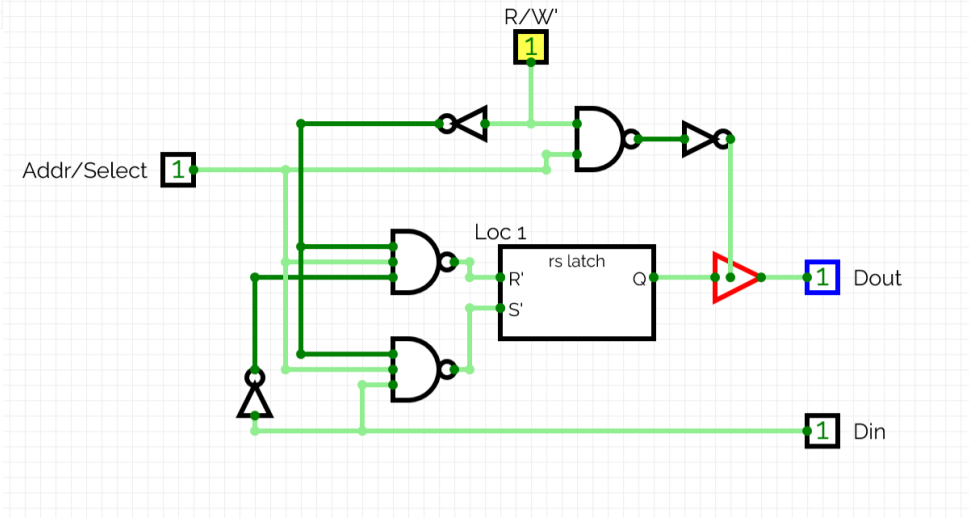

(c) 2 bit 2 word memory

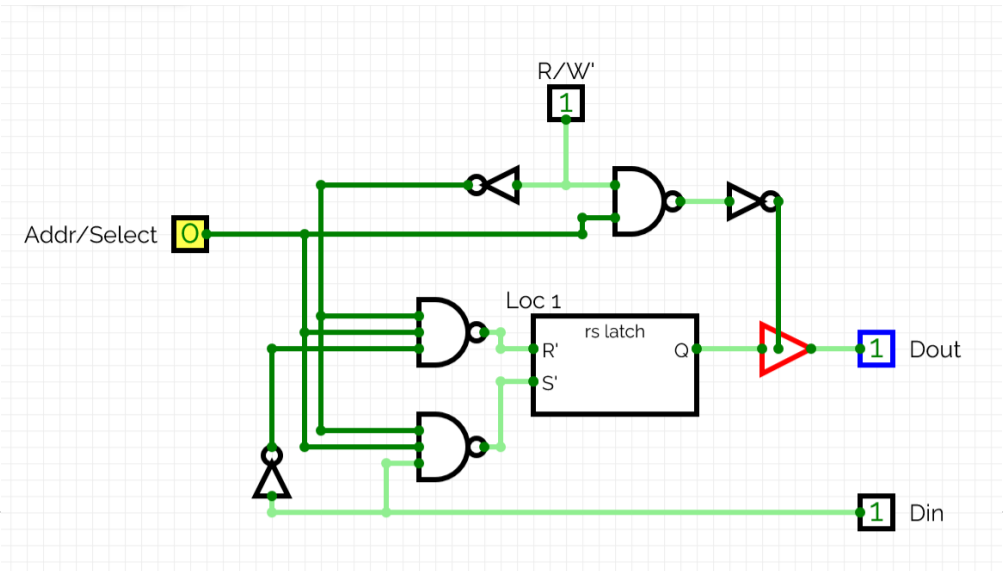
Figure 4: Implementation of Memory

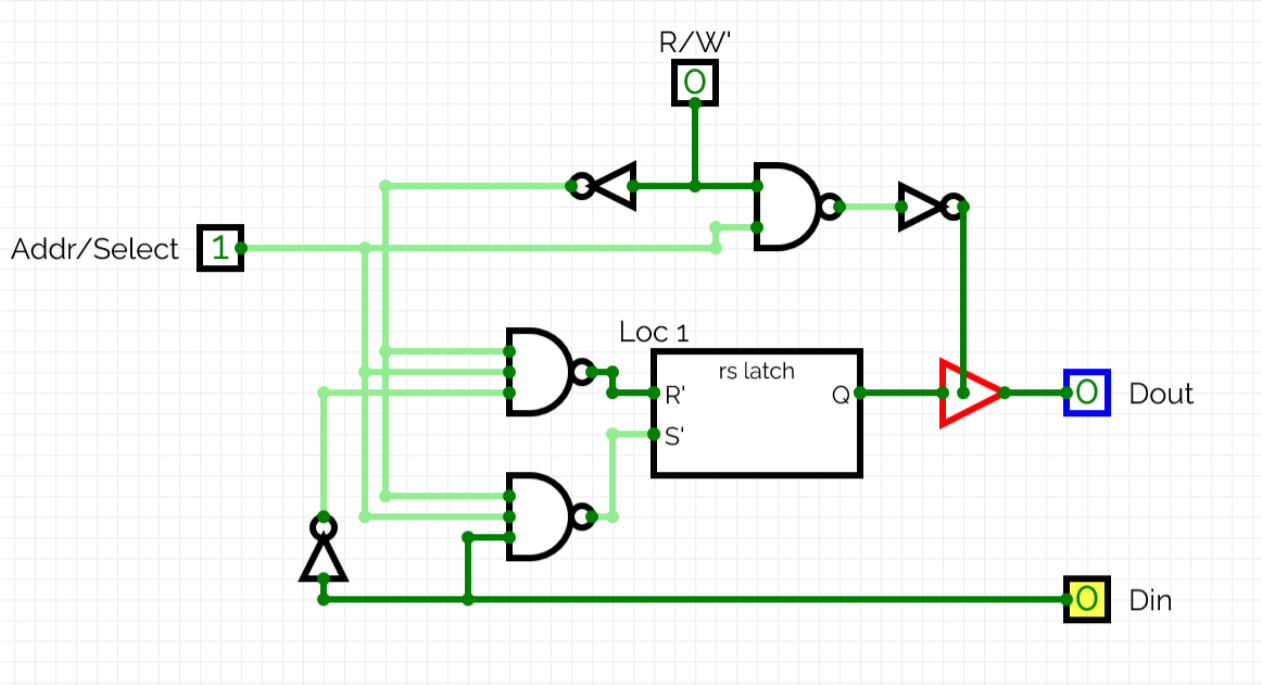
## 1.7 Simulation Results

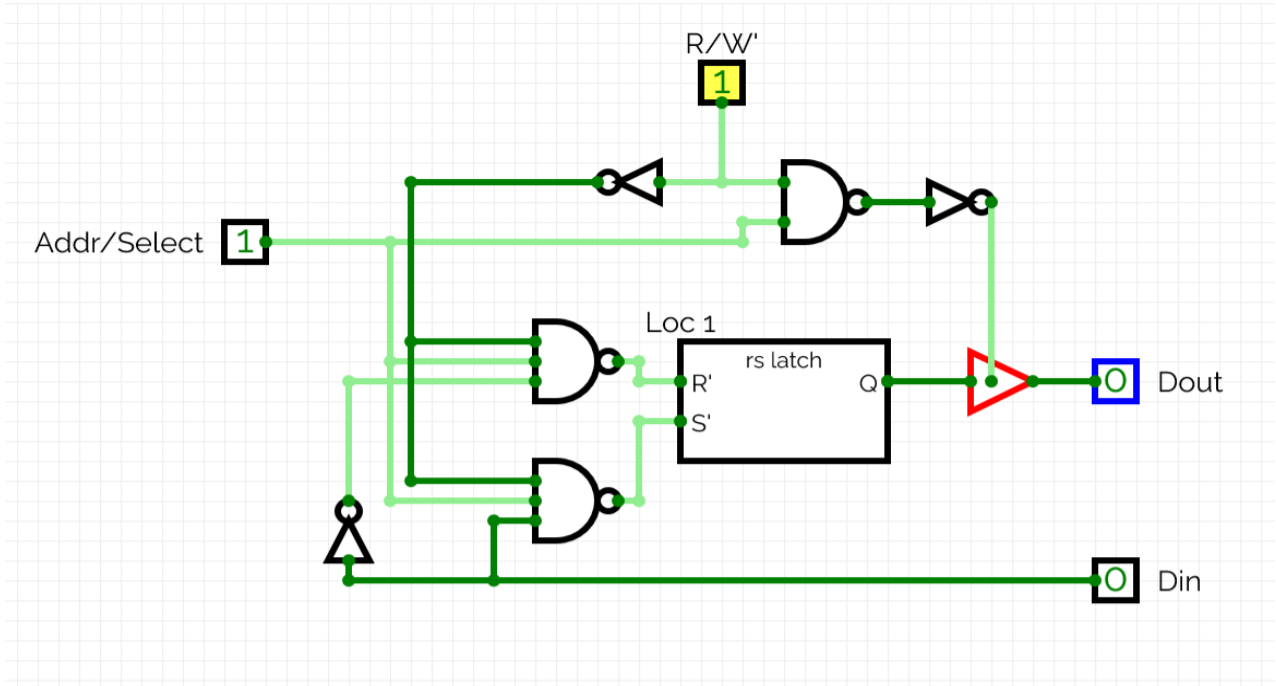
(a) 1 bit 1 word memory



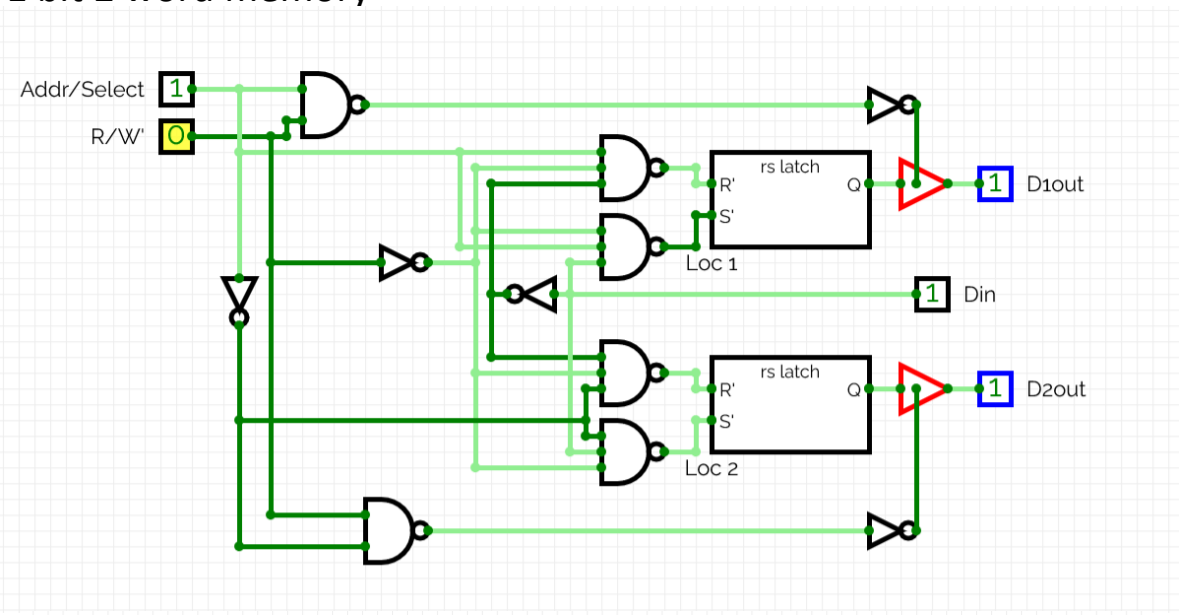



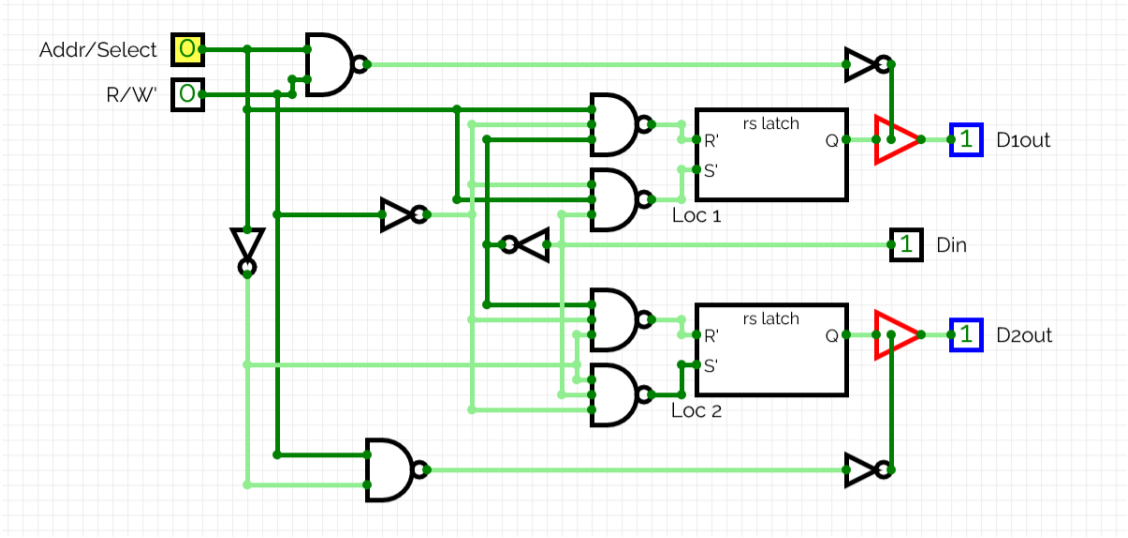



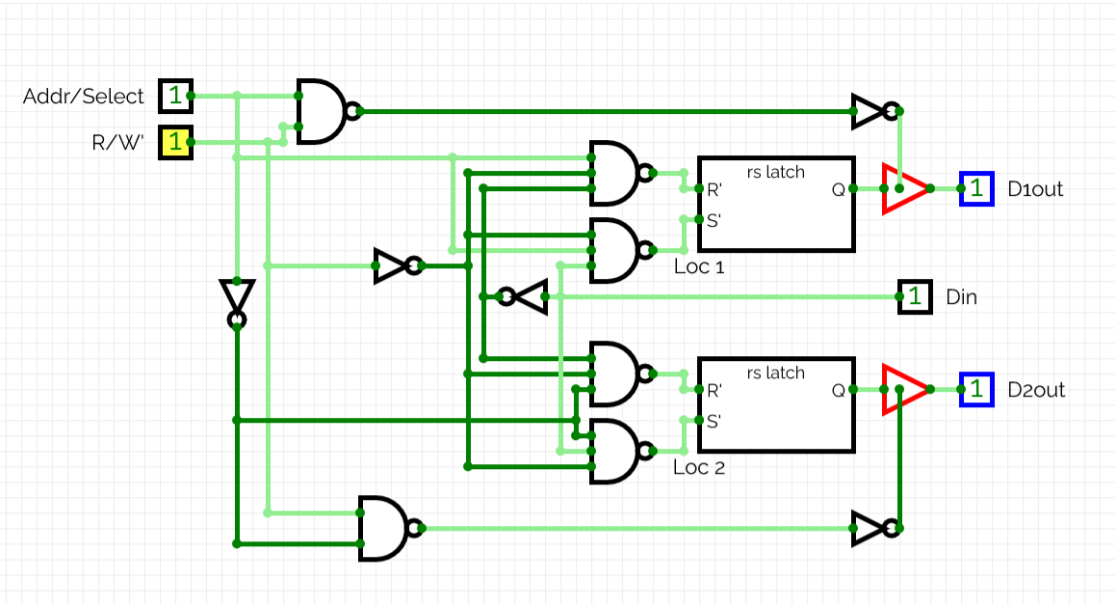

R/W'
0
Addr/Select
1
Loc 1
rs latch
R'
S'
Q
0
Dout
0
Din

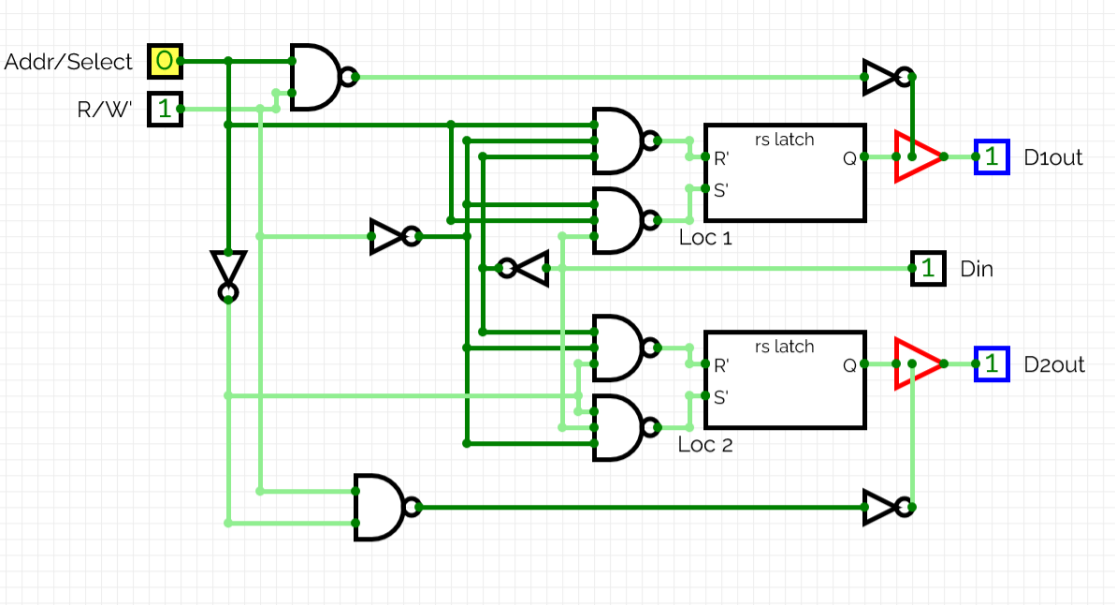

R/W'
1
Addr/Select
1
Loc 1
rs latch
R'
S'
Q
0
Dout
0
Din

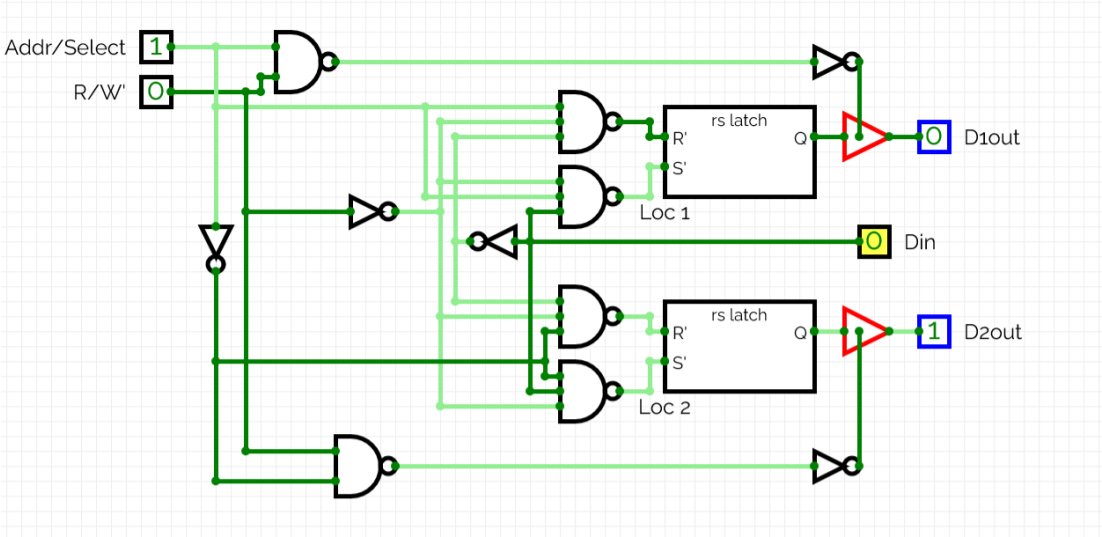
## (b) 1 bit 2 word memory

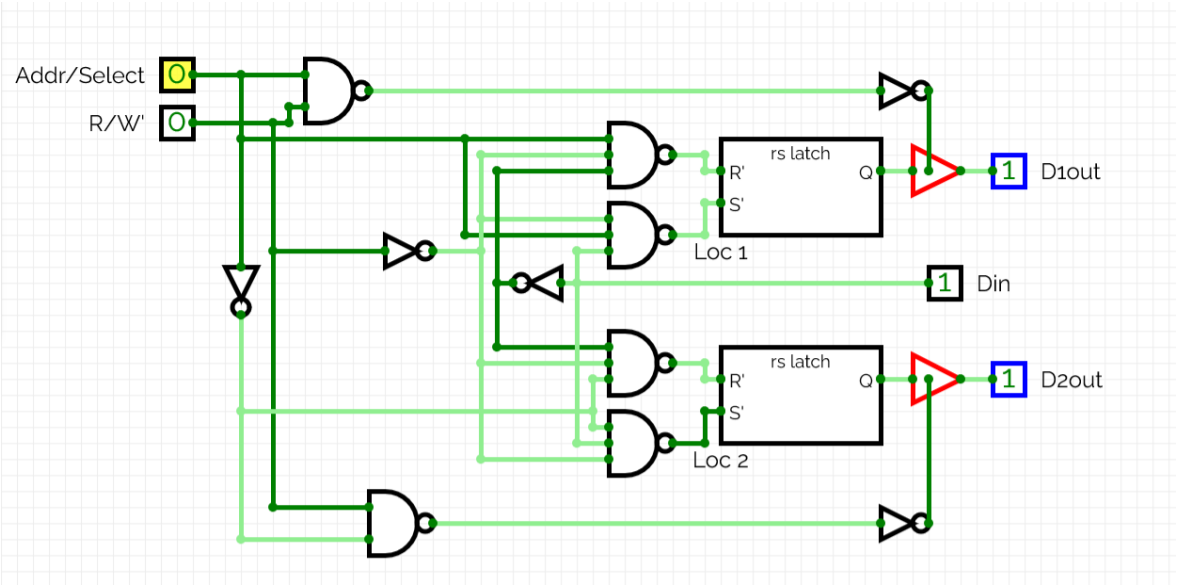


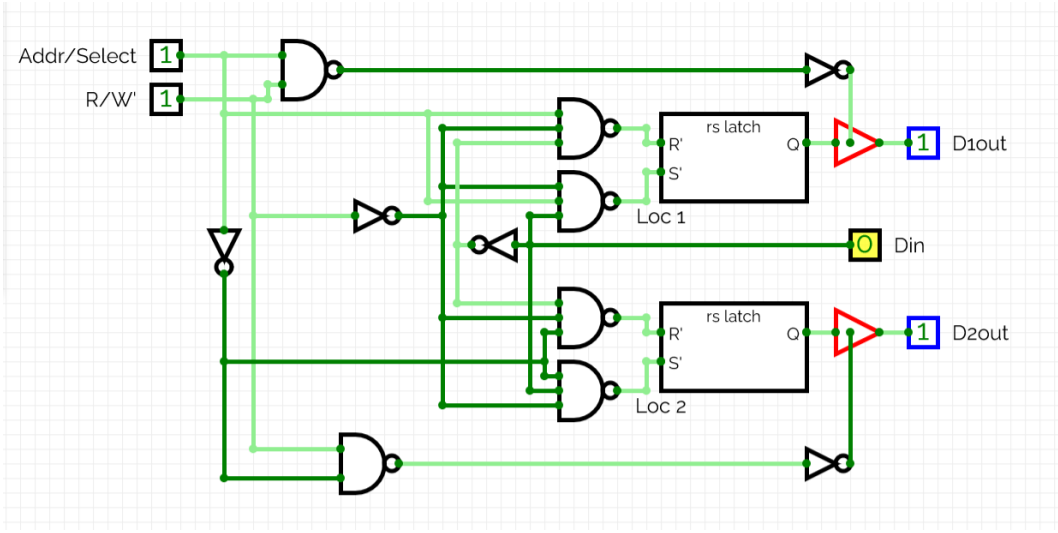



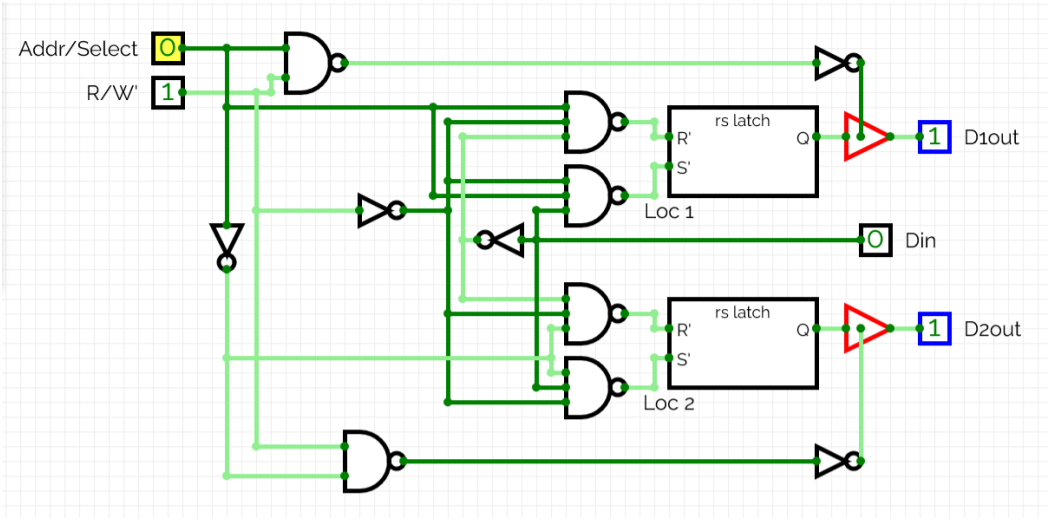



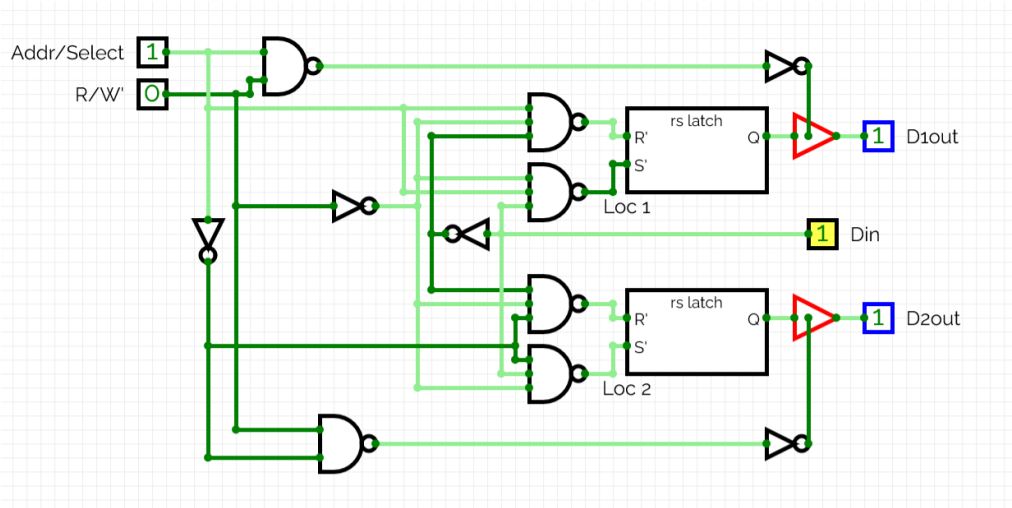

Addr/Select
0
R/W'
1
rs latch
R'
S'
Q
Loc 1
1
D1out
1
Din
rs latch
R'
S'
Q
Loc 2
1
D2out

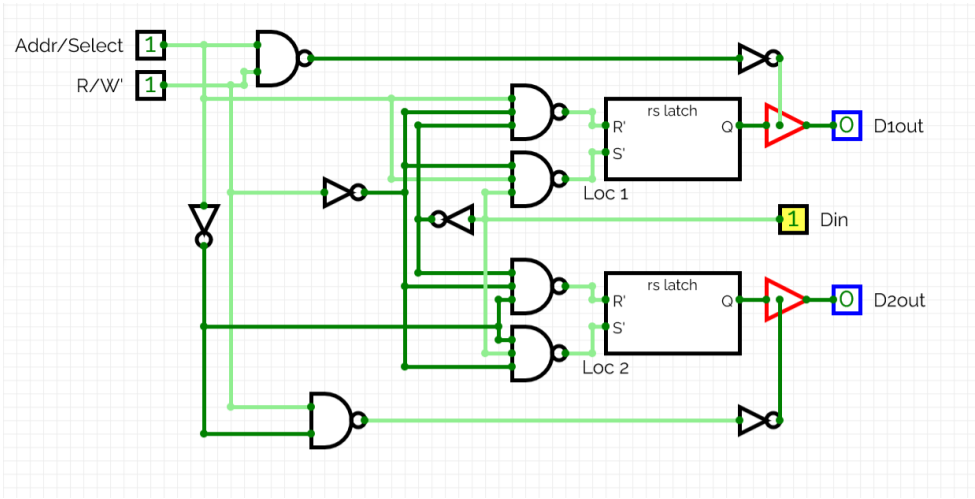

Addr/Select
1
R/W'
0
rs latch
R'
S'
Q
Loc 1
0
D1out
0
Din
rs latch
R'
S'
Q
Loc 2
1
D2out

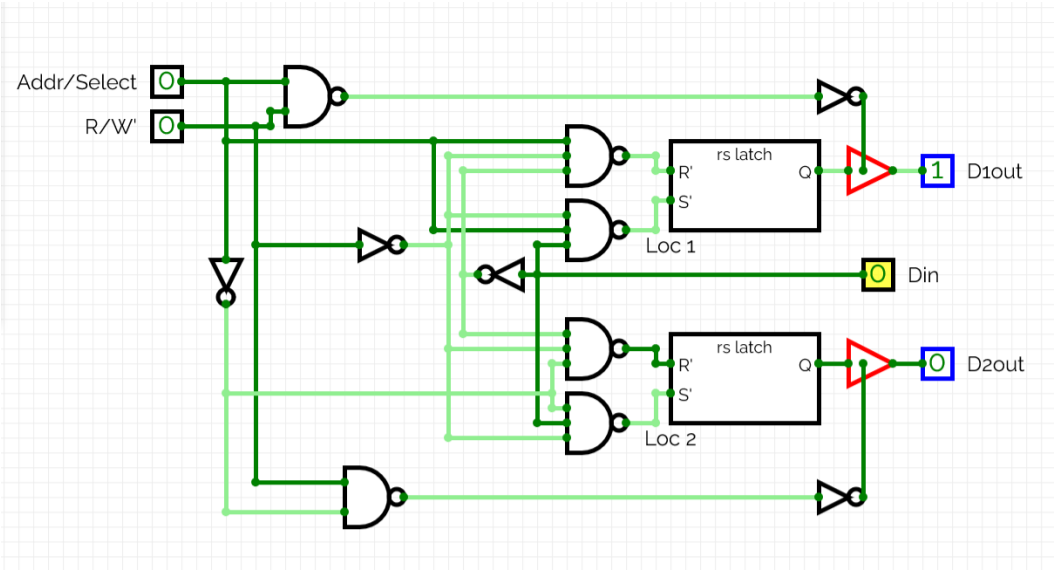

Addr/Select
0
R/W'
0
rs latch
R'
S'
Q
Loc 1
1
D1out
1
Din
rs latch
R'
S'
Q
Loc 2
1
D2out

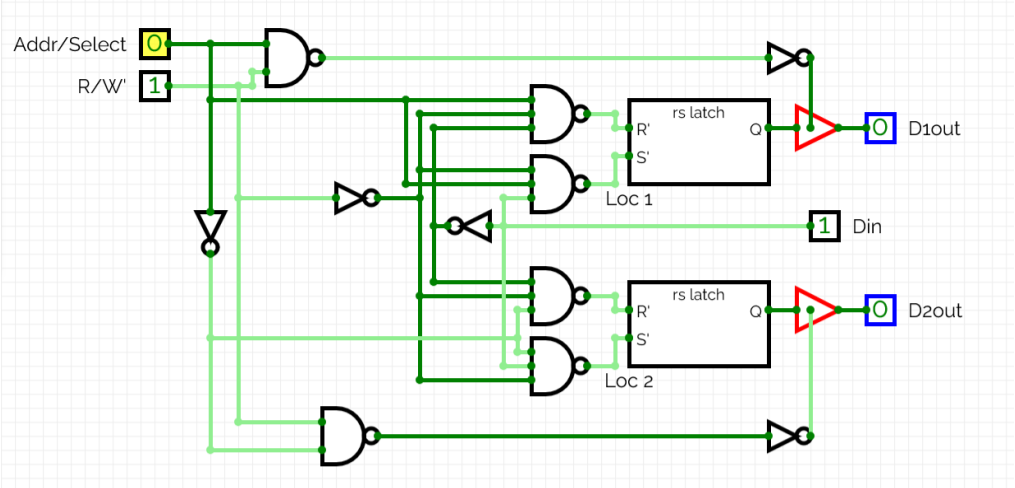

Addr/Select
1
R/W'
1
rs latch
R'
S'
Q
1
D1out
Loc 1
0
Din
rs latch
R'
S'
Q
1
D2out
Loc 2


Addr/Select
0
R/W'
1
rs latch
R'
S'
Q
1
D1out
Loc 1
0
Din
rs latch
R'
S'
Q
1
D2out
Loc 2

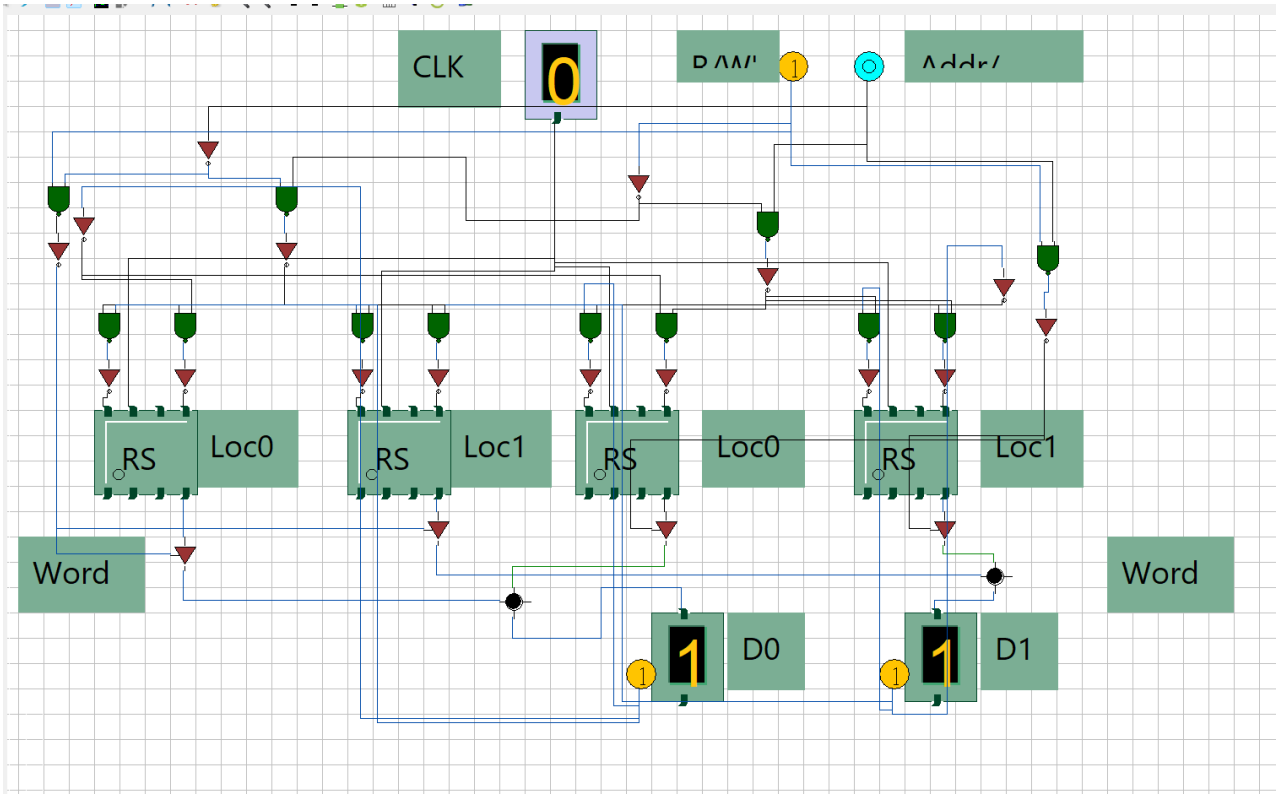

Addr/Select
1
R/W'
0
rs latch
R'
S'
Q
1
D1out
Loc 1
1
Din
rs latch
R'
S'
Q
1
D2out
Loc 2

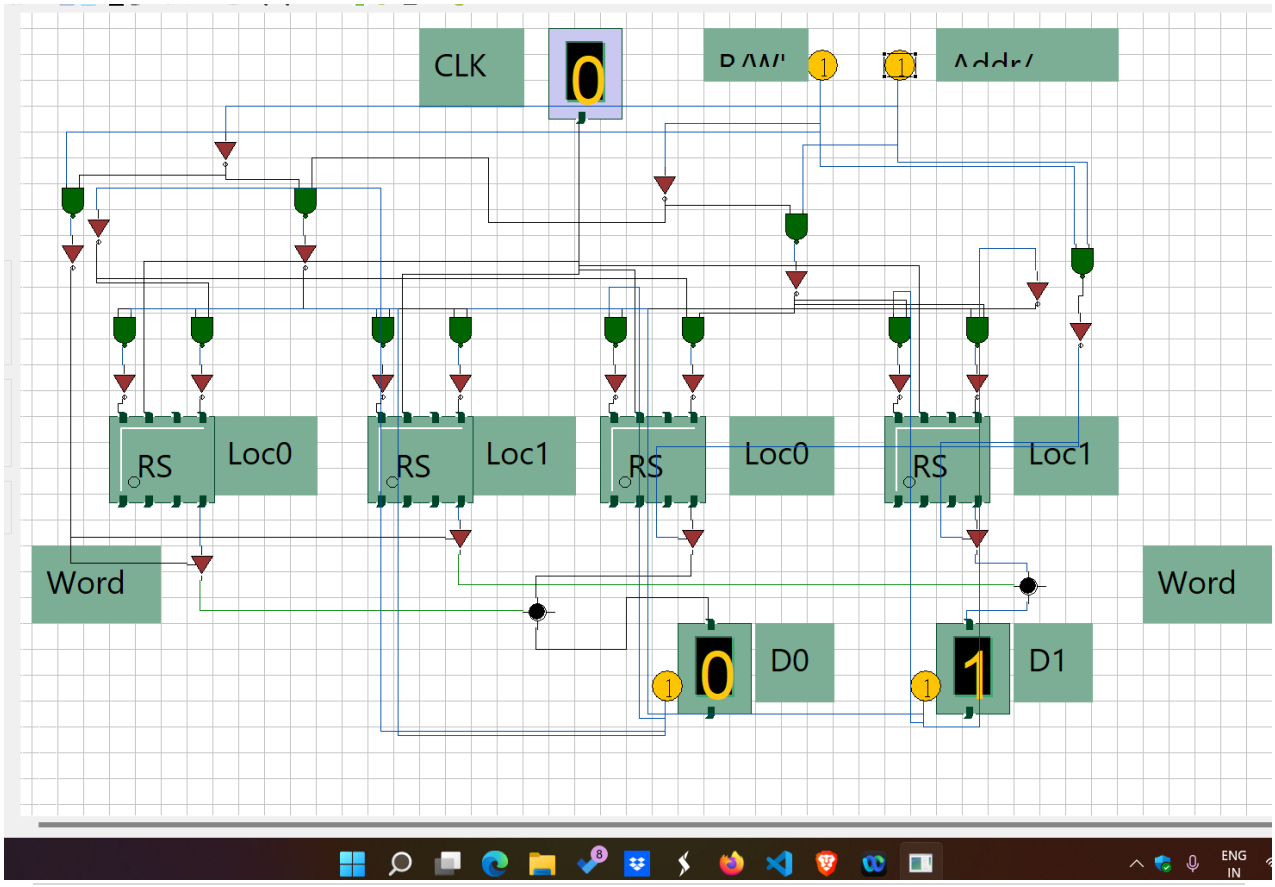

Addr/Select
1
R/W'
1
rs latch
R'
S'
Q
0
D1out
Loc 1
1
Din
rs latch
R'
S'
Q
0
D2out
Loc 2


Addr/Select
0
R/W'
0
rs latch
R'
S'
Q
1
D1out
Loc 1
0
Din
rs latch
R'
S'
Q
0
D2out
Loc 2


Addr/Select
0
R/W'
1
rs latch
R'
S'
Q
0
D1out
Loc 1
1
Din
rs latch
R'
S'
Q
0
D2out
Loc 2

(c) 2 bit 2 word memory



## 1.8 Verification

| Sr. No | Select | R/$\overline{W}$ | Data In | Data Out | Activity |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | - | Write 1 in Loc-1 |
| 2 | 1 | 1 | - | 1 | Read 1 from Loc-1 |
| 3 | 0 | x | - | - | No operation |
| 4 | 1 | 0 | 0 | - | Write 0 in Loc-1 |
| 5 | 1 | 1 | 1 | 0 | Read 0 from Loc-1 |

Table 1: 1 bit 1 word memory

| Sr. No | Select | R/$\overline{W}$ | Data In | Data Out | Activity |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 ($d_1$) | - | Write 1 in Loc-1 |
| 2 | 0 | 0 | 1 ($d_2$) | - | Write 1 in Loc-0 |
| 3 | 1 | 1 | - | 1 ($d_1$) | Read 1 from Loc-1 |
| 4 | 0 | 1 | - | 1 ($d_2$) | Read 1 from Loc-0 |
| 5 | 1 | 0 | 0 | - | Write 0 in Loc-1 |
| 6 | 0 | 0 | 1 | - | Write 1 in Loc-0 |
| 7 | 1 | 1 | - | 0 | Read 0 from Loc-1 |
| 8 | 0 | 1 | - | 1 | Read 1 from Loc-0 |
| 9 | 1 | 0 | 1 | - | Write 1 in Loc-1 |
| 10 | 0 | 0 | 0 | - | Write 0 in Loc-0 |
| 11 | 1 | 1 | - | 1 | Read 1 from Loc-1 |
| 12 | 0 | 1 | - | 0 | Read 0 from Loc-0 |

Table 2: 1 bit 2 word memory

| Sr. No | Select | R/$\overline{W}$ | $D1_{in}$ | $D0_{in}$ | $D1_{out}$ | $D0_{out}$ | Verify |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | - | - | Write 10 in Loc-1 |
| 2 | 0 | 0 | 0 | 1 | - | - | Write 01 in Loc-0 |
| 3 | 1 | 1 | - | - | 1 | 0 | Read 10 from Loc-1 |
| 4 | 0 | 1 | - | - | 0 | 1 | Read 01 from Loc-0 |
| 5 | 1 | 0 | 0 | 0 | - | - | Write 00 in Loc-1 |
| 6 | 0 | 0 | 1 | 1 | - | - | Write 11 in Loc-0 |
| 7 | 1 | 1 | - | - | 0 | 0 | Read 00 from Loc-1 |
| 8 | 0 | 1 | - | - | 1 | 1 | Read 11 from Loc-0 |

Table 3: 2 bit 2 word memory

## 1.9 Conclusion

We can design $2^{m1}$ X $d_1$ Memory Module where $m_1 >= m$ and $d_1 >= d$

# Experiment No: 2

## Design of Carry-Look-Ahead Adder

### 2.1 Objective

To Design
1. 4-bit carry-lookahead adder (CLA) using half adders/full adders.
2. A 16-bit CLA using 4-bit CLAs.

### 2.2 Theoretical Basis and Design

Basic Unit of Carry Look-Ahead Adder is as follows:

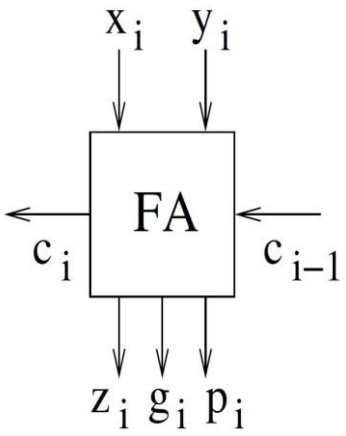



Figure 5: Basic Unit of CLA

where,

$$p_i = x_i \oplus y_i$$
$$g_i = x_i y_i$$

Using this building blocks, we can construct an n-bit CLA as follows

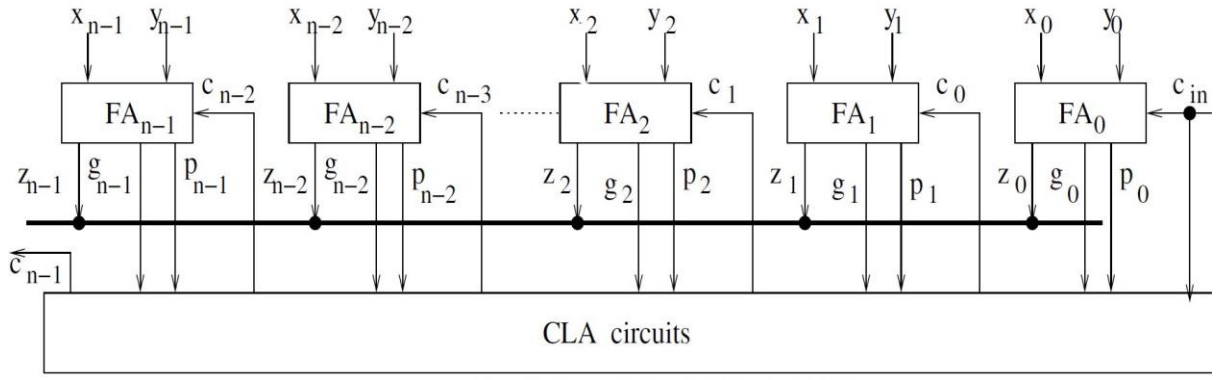



Figure 6: n-bit CLA

Using these we can form this for 4-bit CLA

$$
\begin{aligned}
c_0 &= g_0 + p_0 c_{in} \\
c_1 &= g_1 + p_1 c_0 = g_1 + p_1 g_0 + p_1 p_0 c_{in} \\
c_2 &= g_2 + p_2 c_1 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in} \\
c_3 &= g_3 + p_3 c_2 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in}
\end{aligned}
$$

Now from 4-bit CLA, we can construct 16-bit CLA as follows:

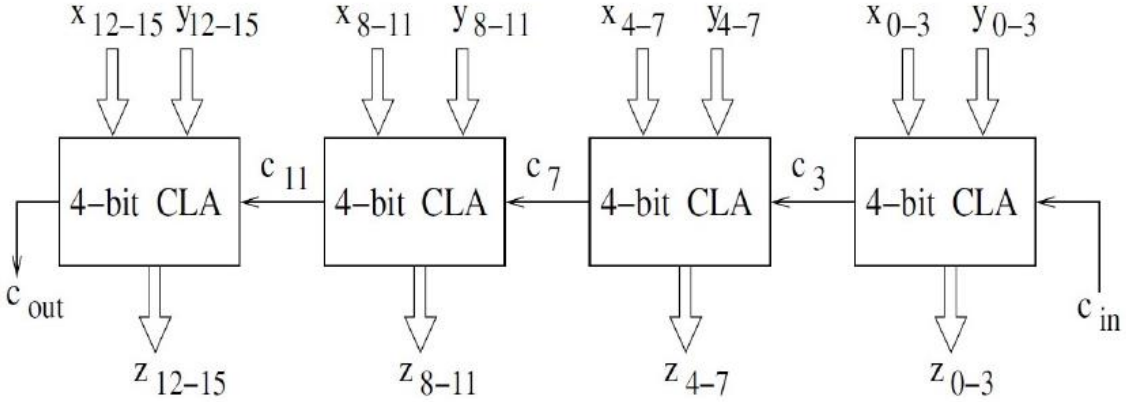



Figure 7: 16-bit CLA

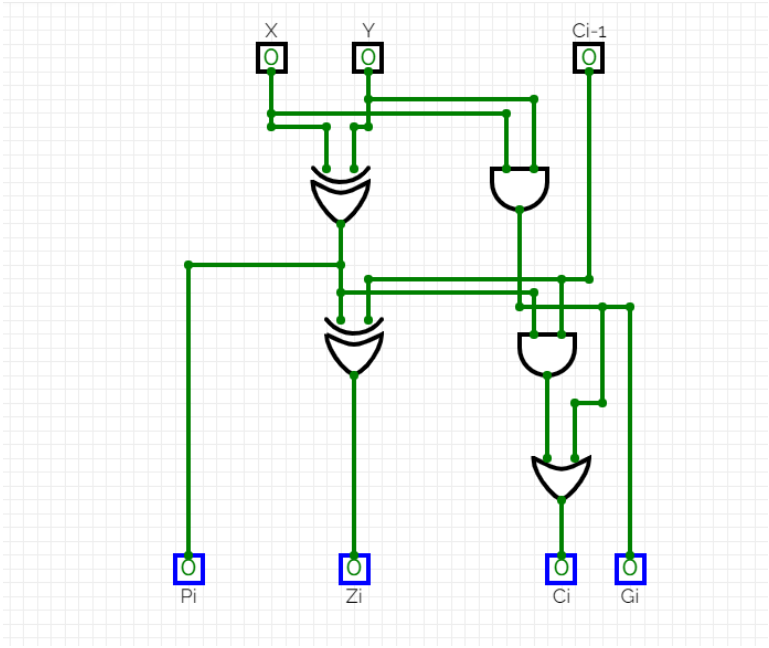
## 2.3 Task

1. Construct 4 modules as in Figure 5 (can use half adder).
2. Simplify the expressions for c0, c1, c2 and c3 assuming cin = 0.
3. Design CLA circuits i.e., realize c0, c1, c2, and c3.
4. Design the CLA as in Figure 6 and verify a part of its truth table.
5. Report on the design of a 16-bit CLA as in Figure 7.

## 2.4 Major Components/Modules Used

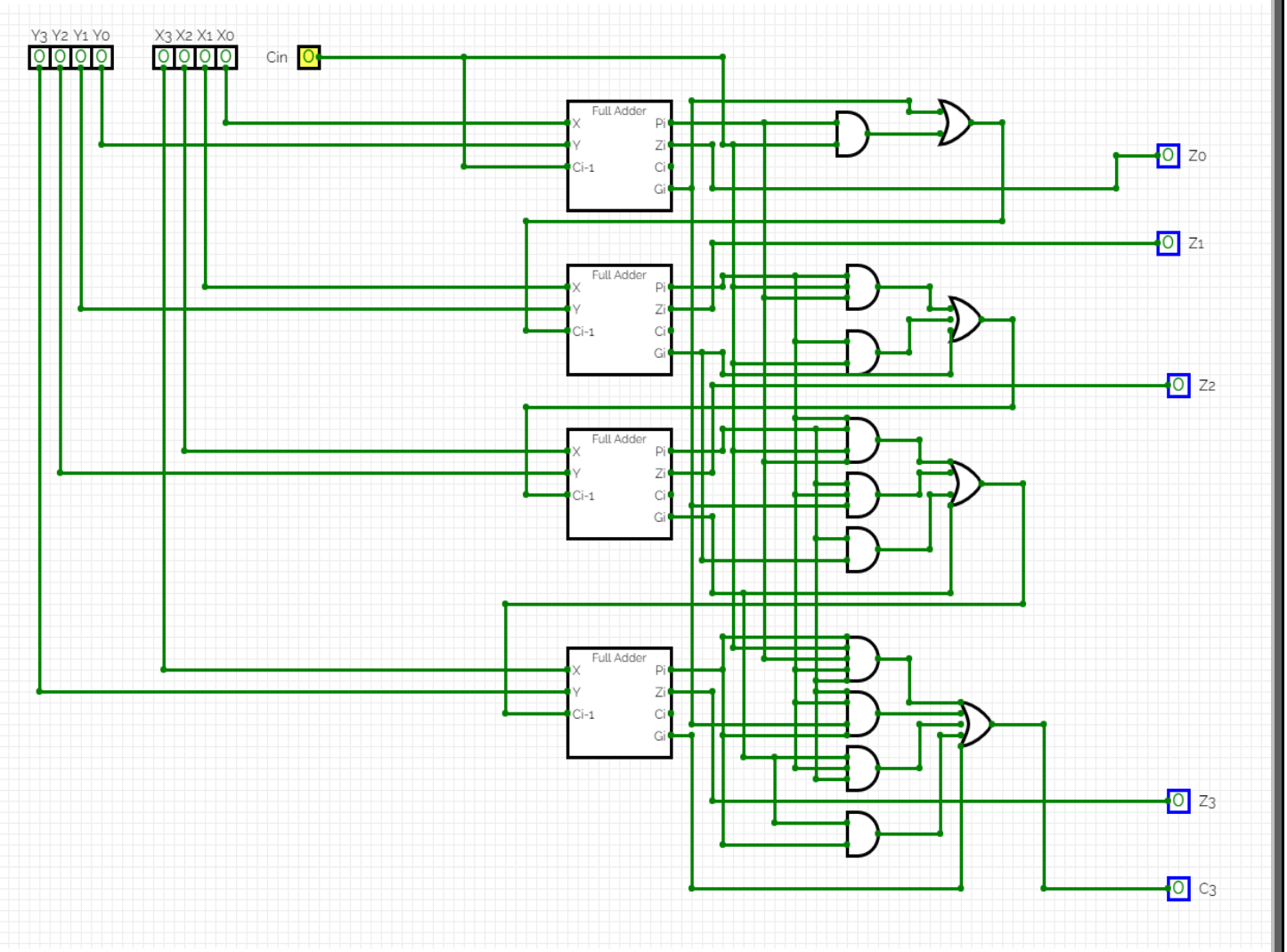
1. AND Gate
2. OR Gate
3. Half Adder
4. Full Adder

## 2.5 Circuit diagram




**(a)CLA full adder block**

## (b) 4 bit CLA


Y3 Y2 Y1 Y0
X3 X2 X1 X0
Cin
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
Z0
Z1
Z2
Z3
C3

## (c) 16 bit CLA

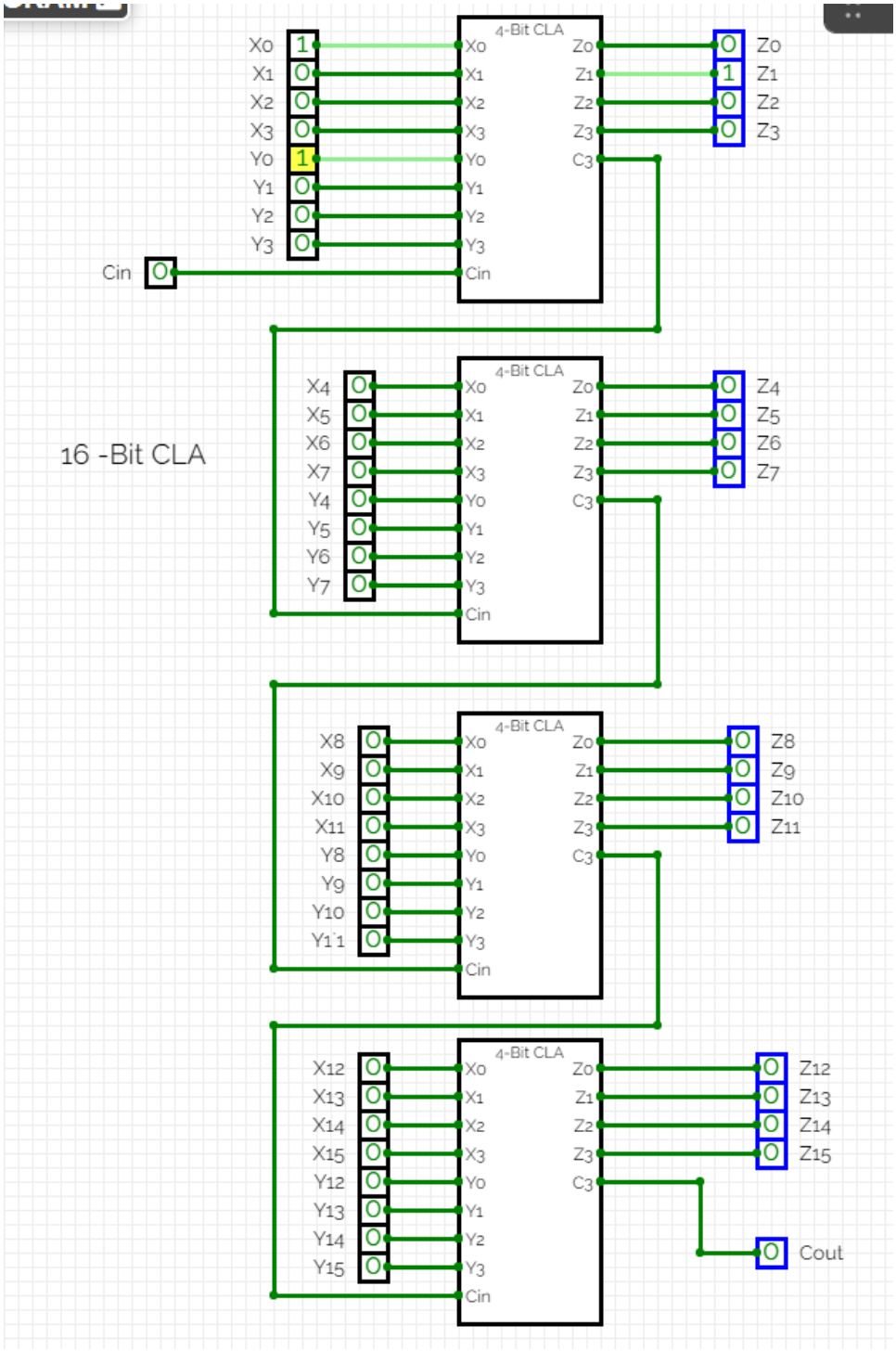
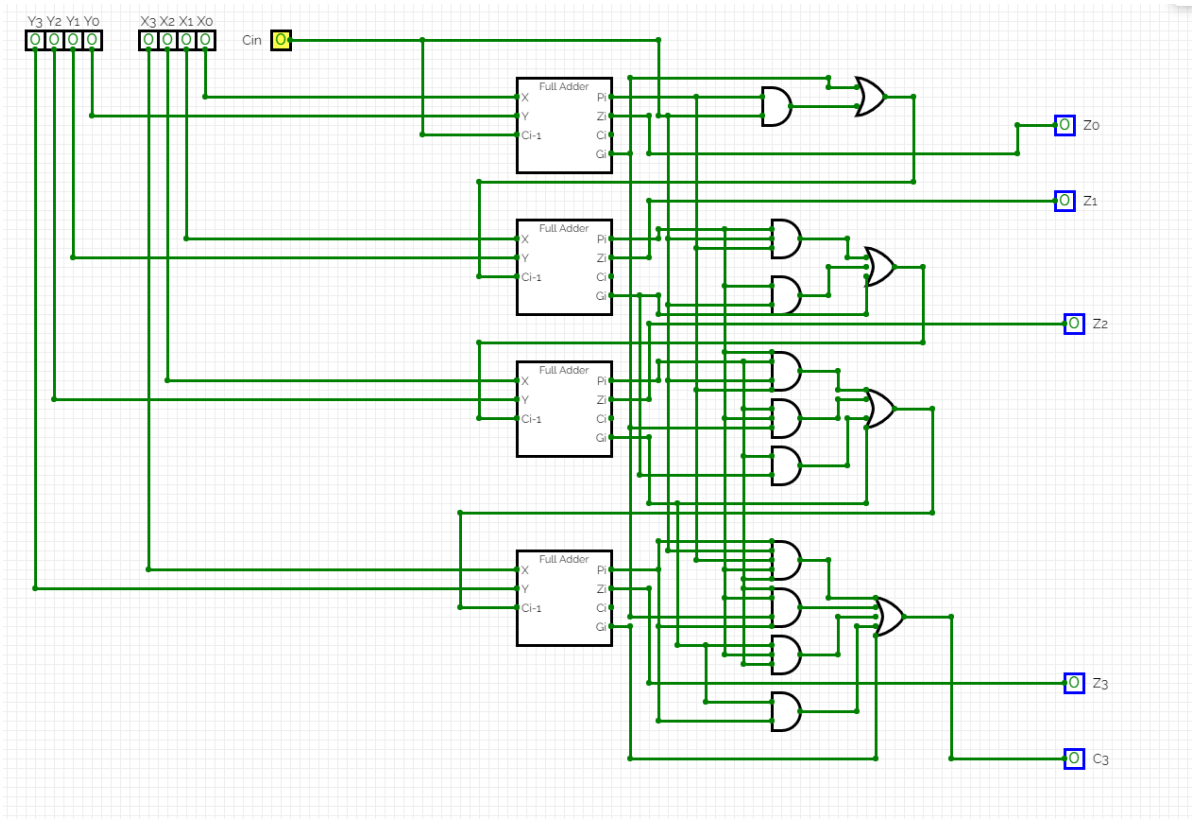
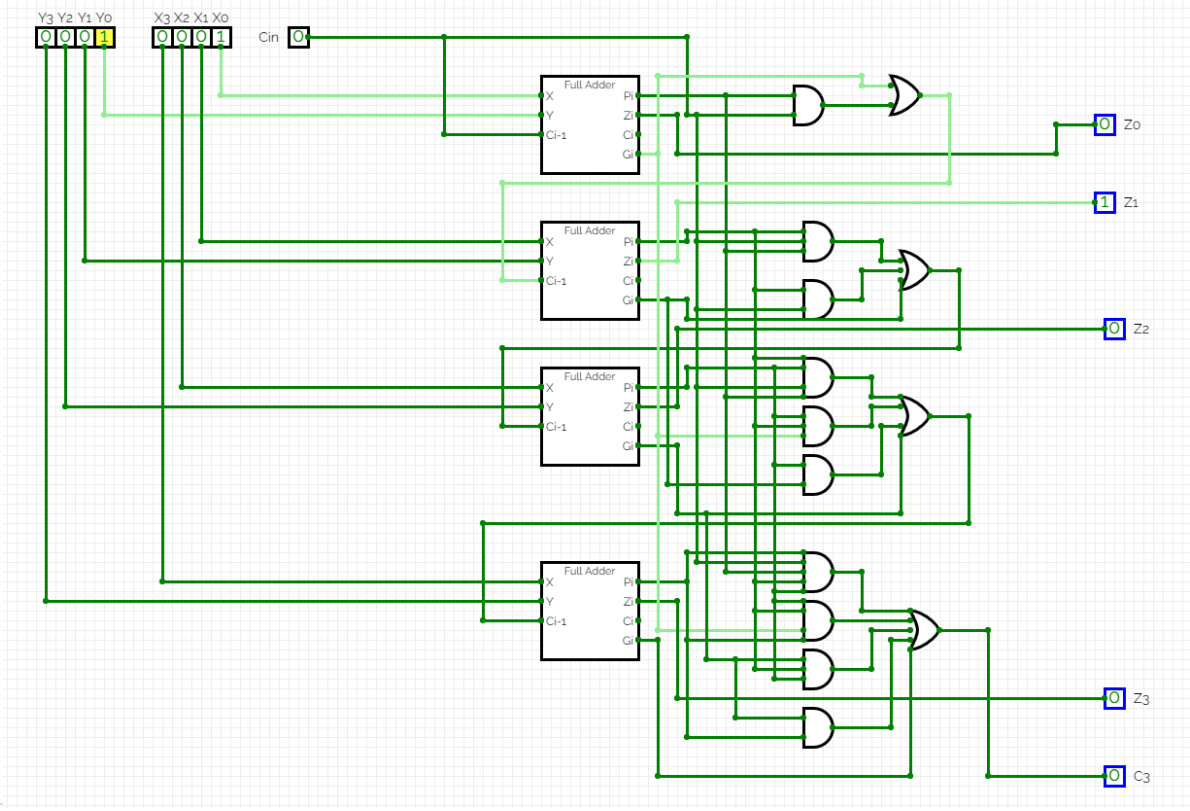


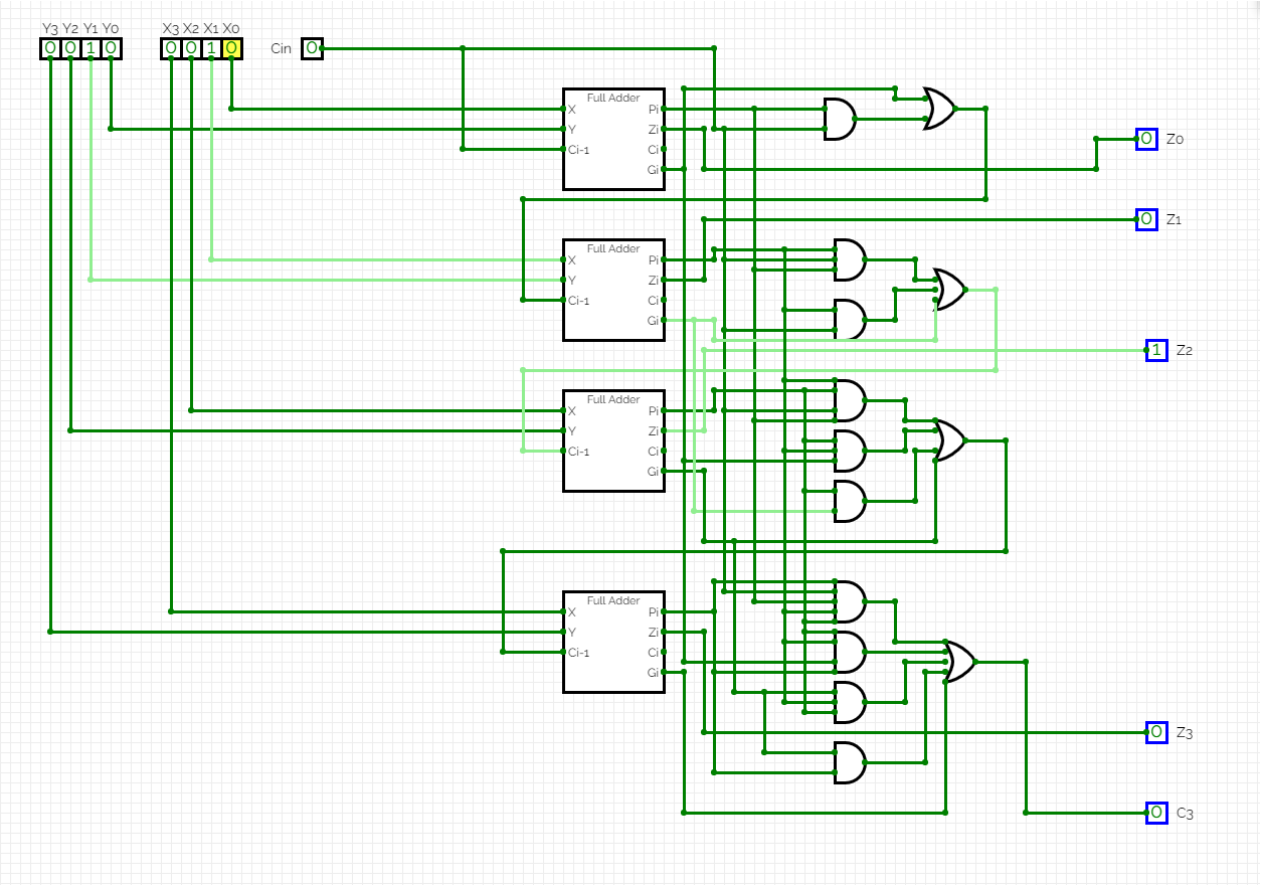

Figure 8: Carry Look Ahead Adder

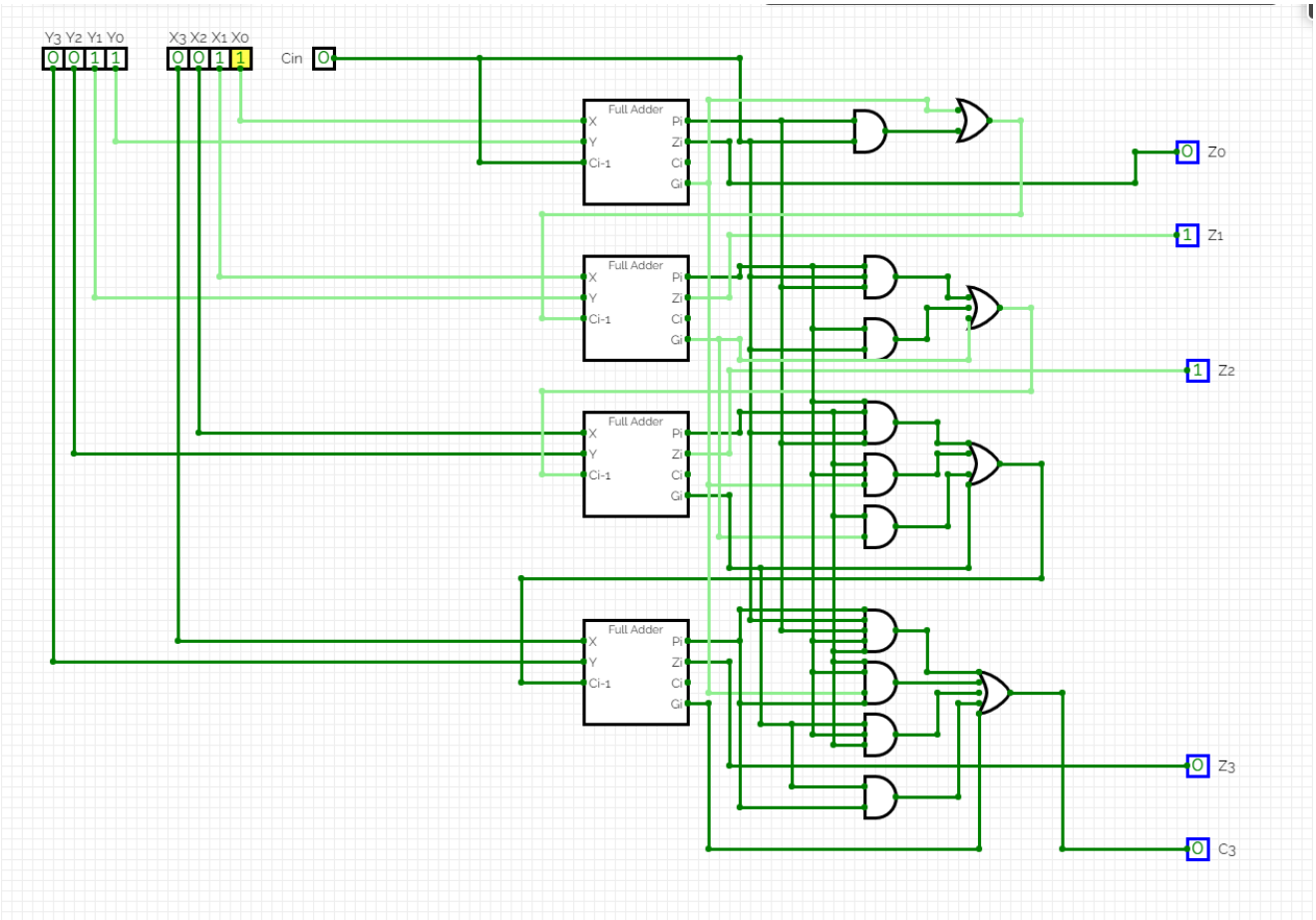
## 2.6 Simulation Results

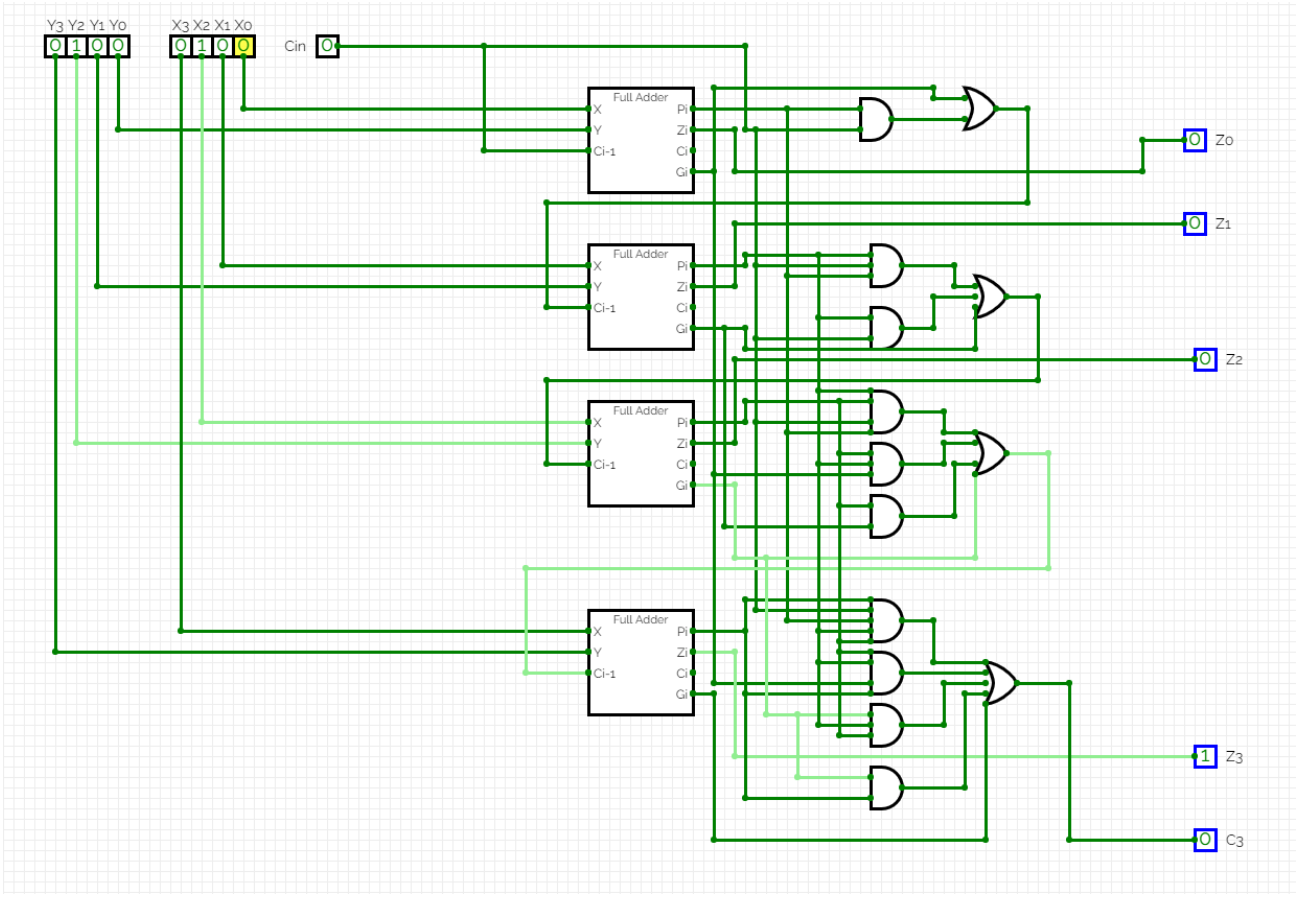
(a)4-bit CLA

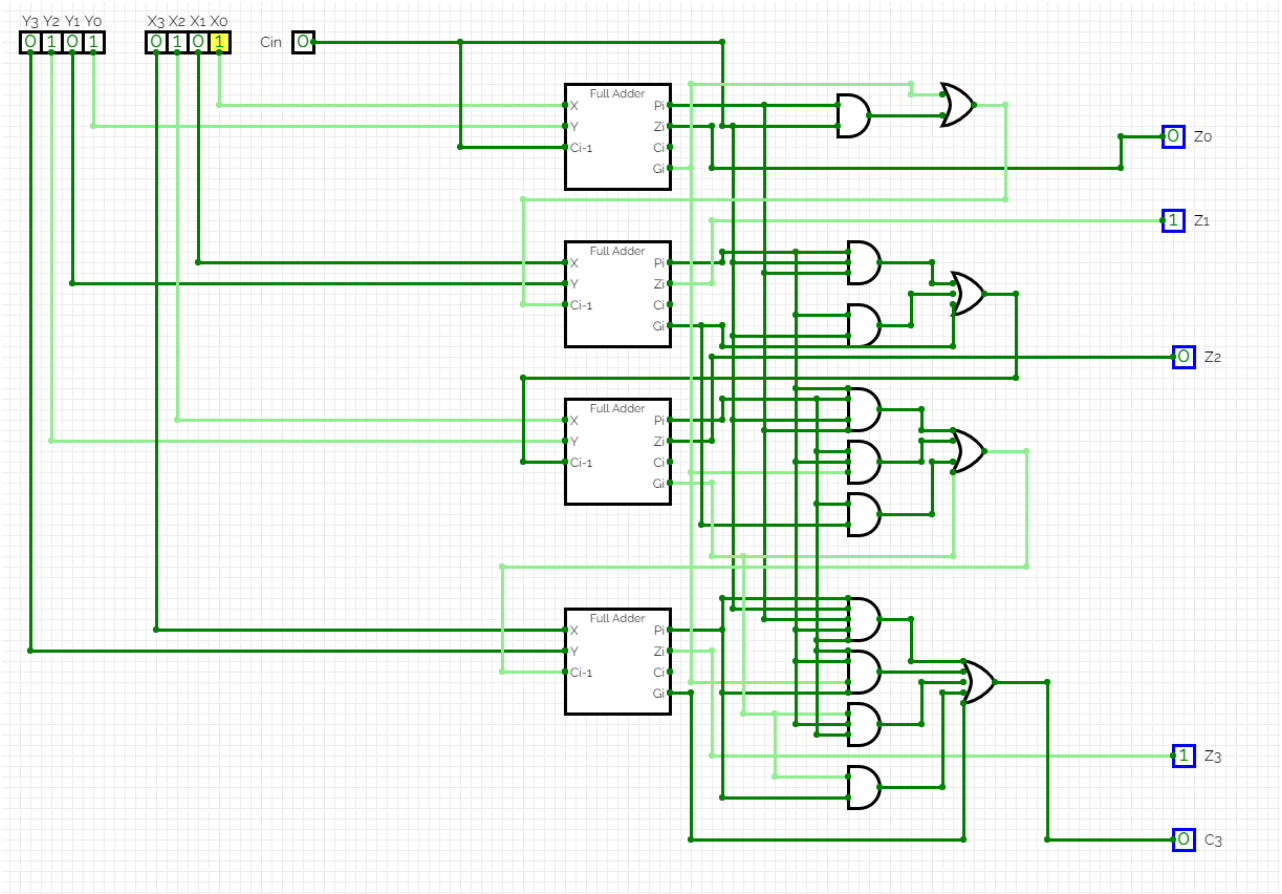


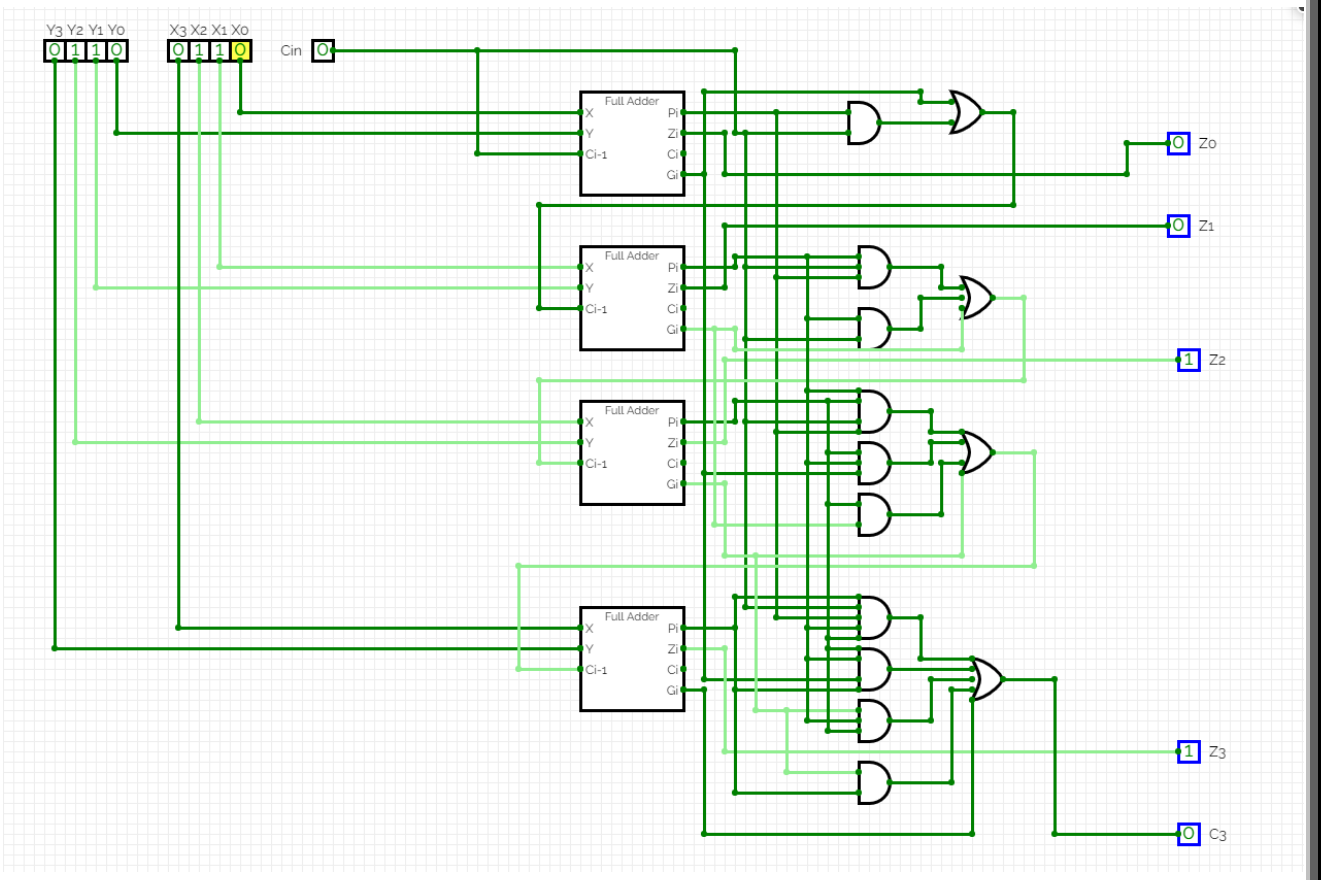



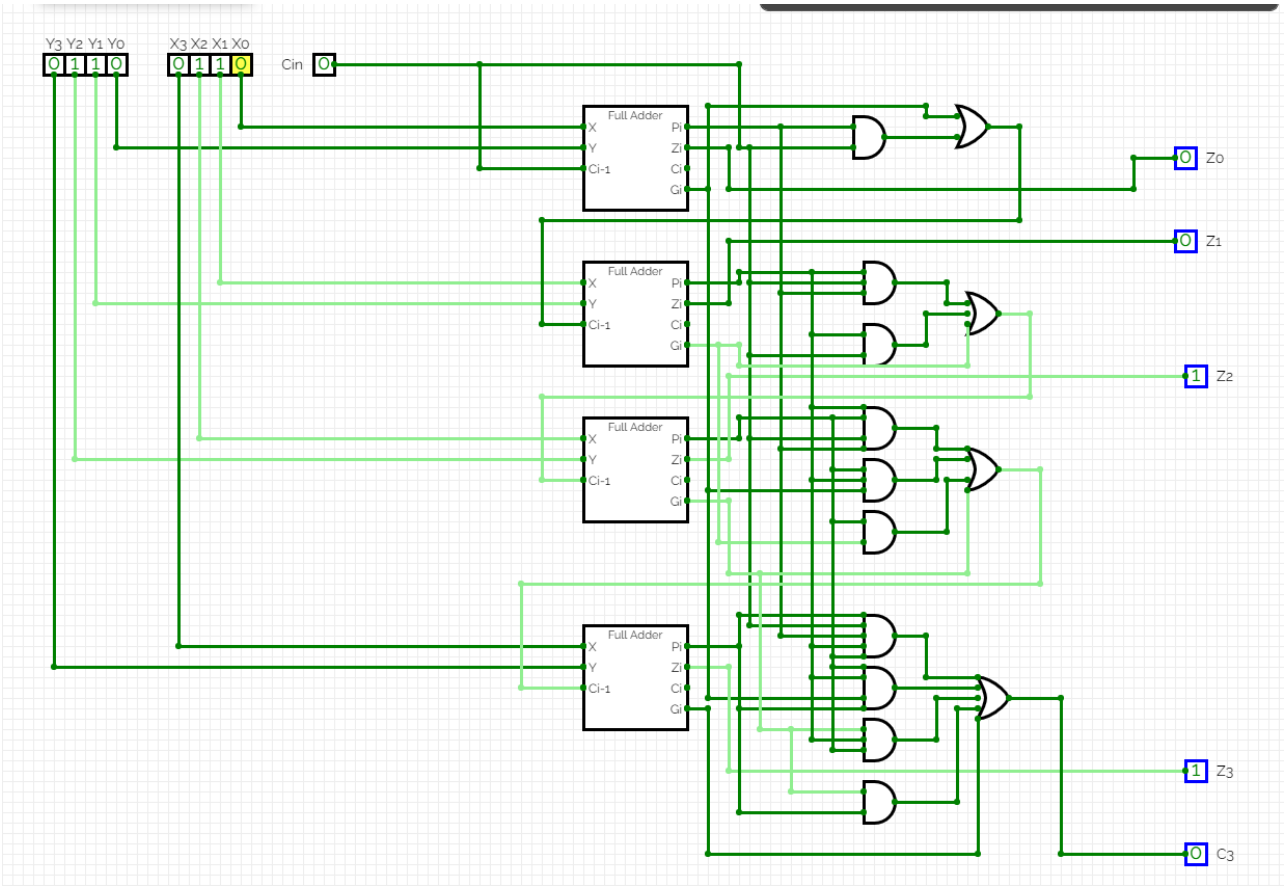

Y3 Y2 Y1 Y0
0 0 1 0
X3 X2 X1 X0
0 0 1 0
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
Z0
Z1
Z2
Z3
C3

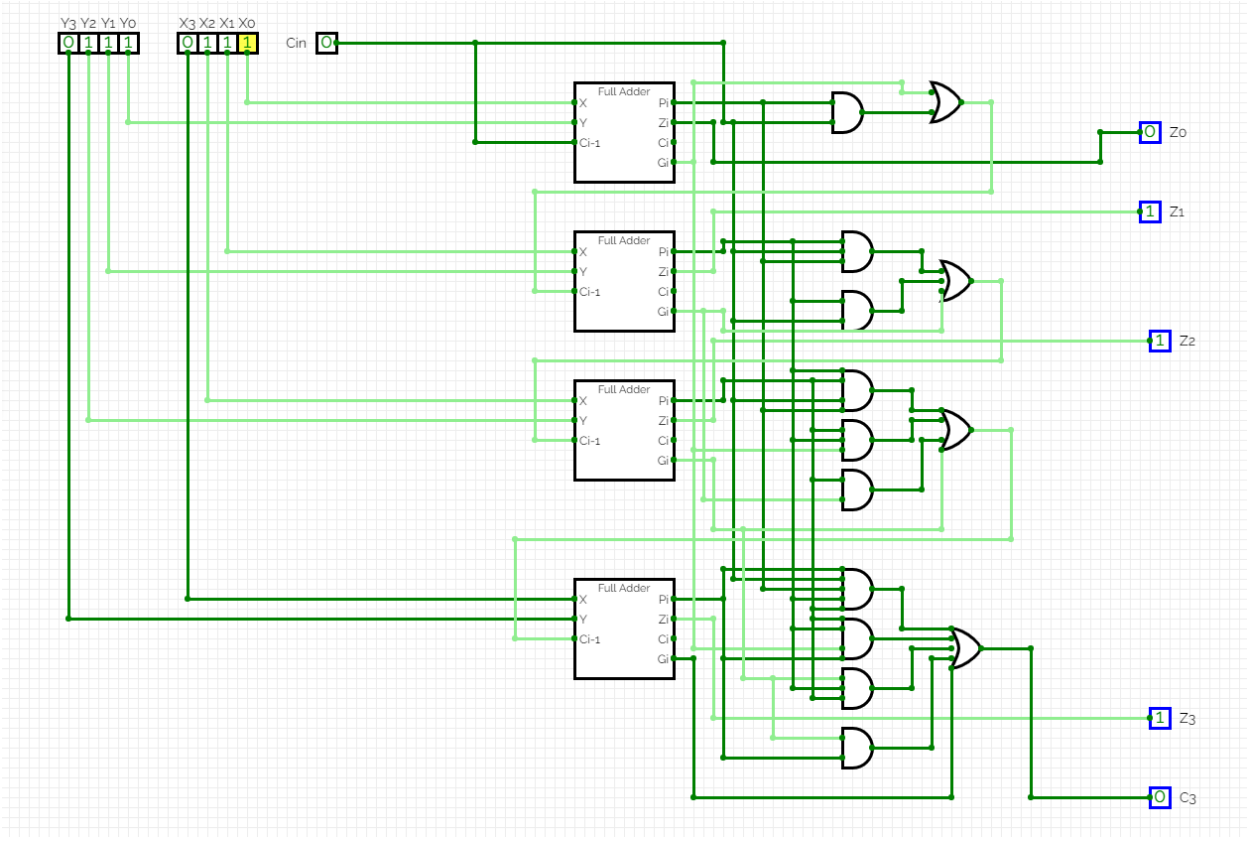

Y3 Y2 Y1 Y0
0 0 1 1
X3 X2 X1 X0
0 0 1 1
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
Z0
Z1
Z2
Z3
C3

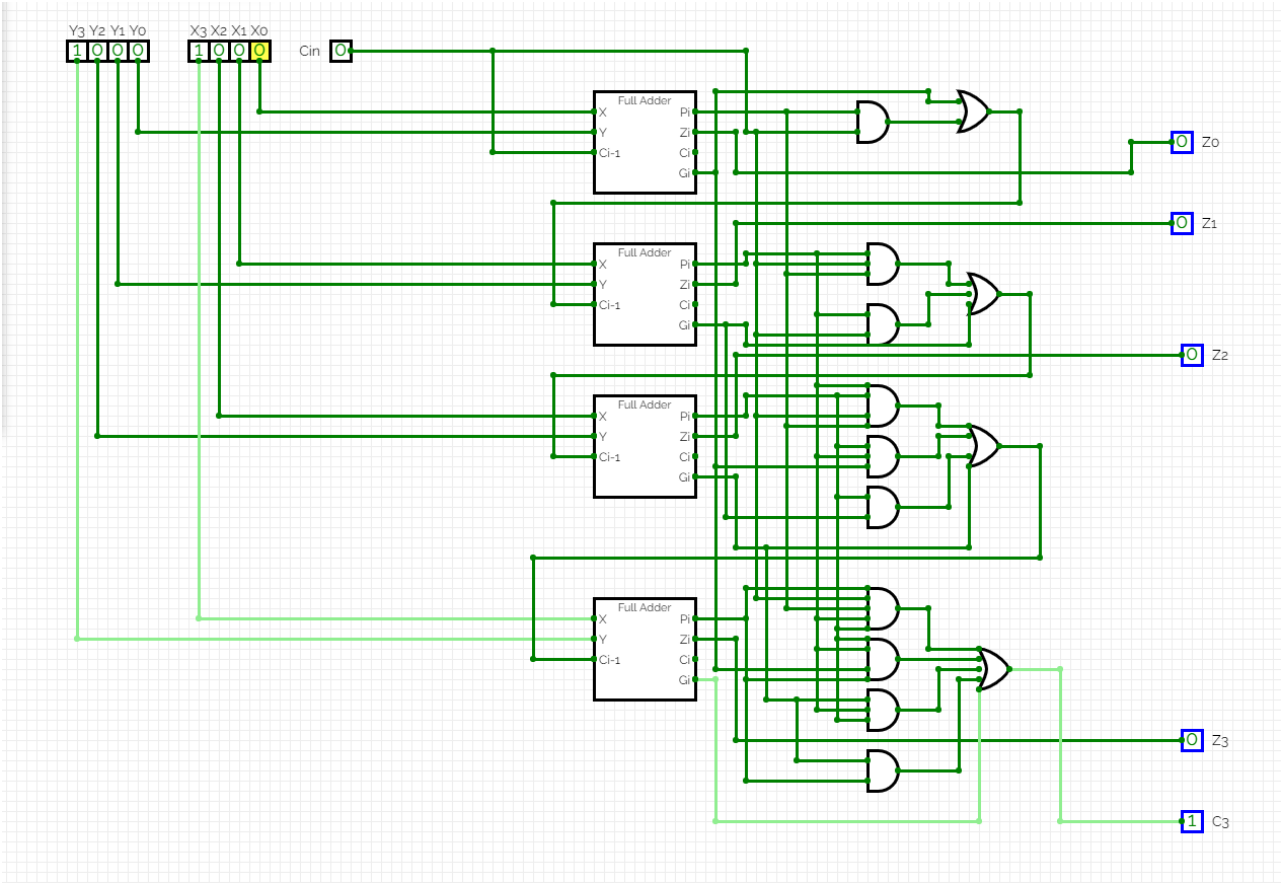

Y3 Y2 Y1 Y0
0 1 0 0
X3 X2 X1 X0
0 1 0 0
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
Z0
Z1
Z2
Z3
C3

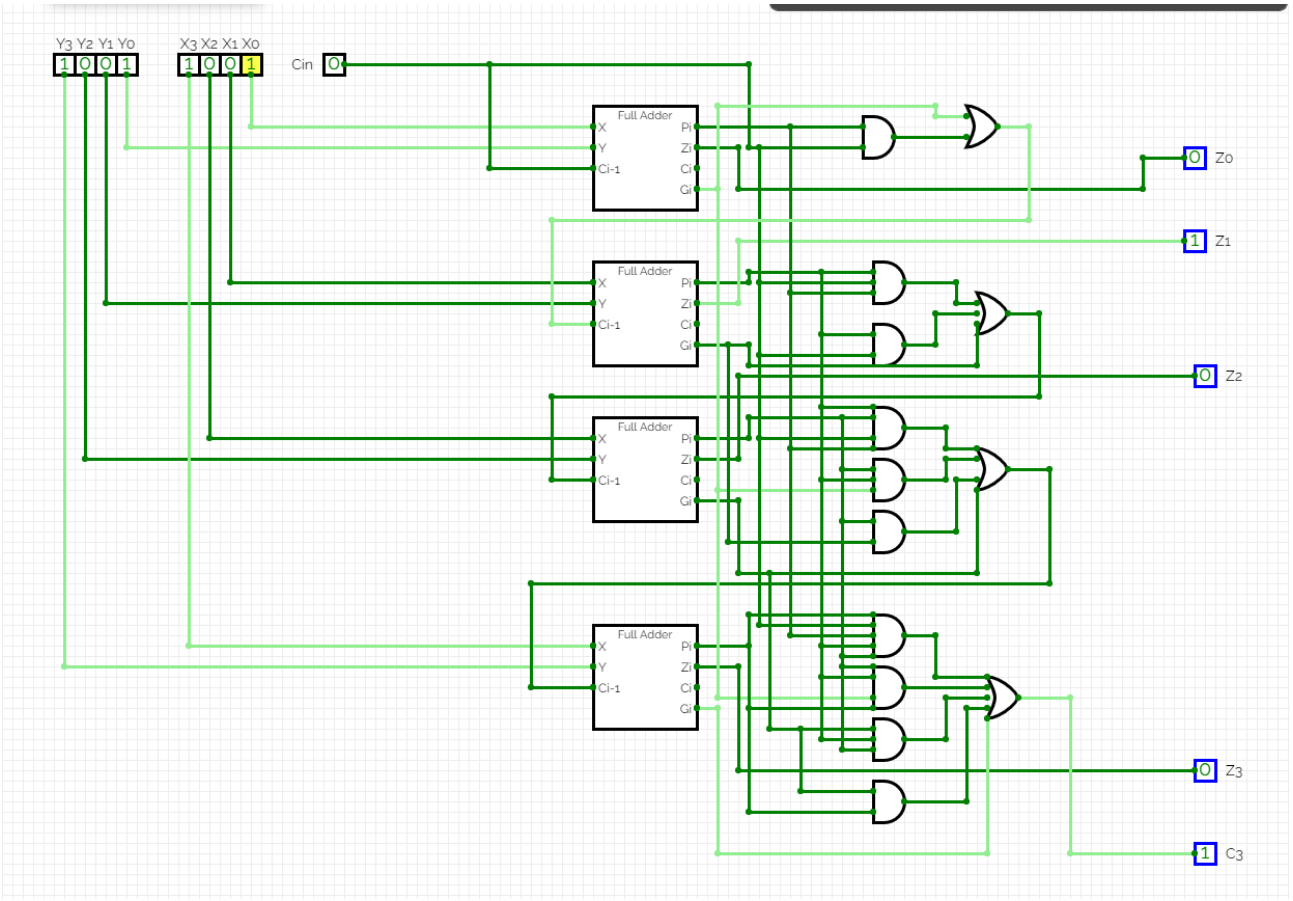

Y3 Y2 Y1 Y0
0 1 0 1
X3 X2 X1 X0
0 1 0 1
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
Z0
Z1
Z2
Z3
C3

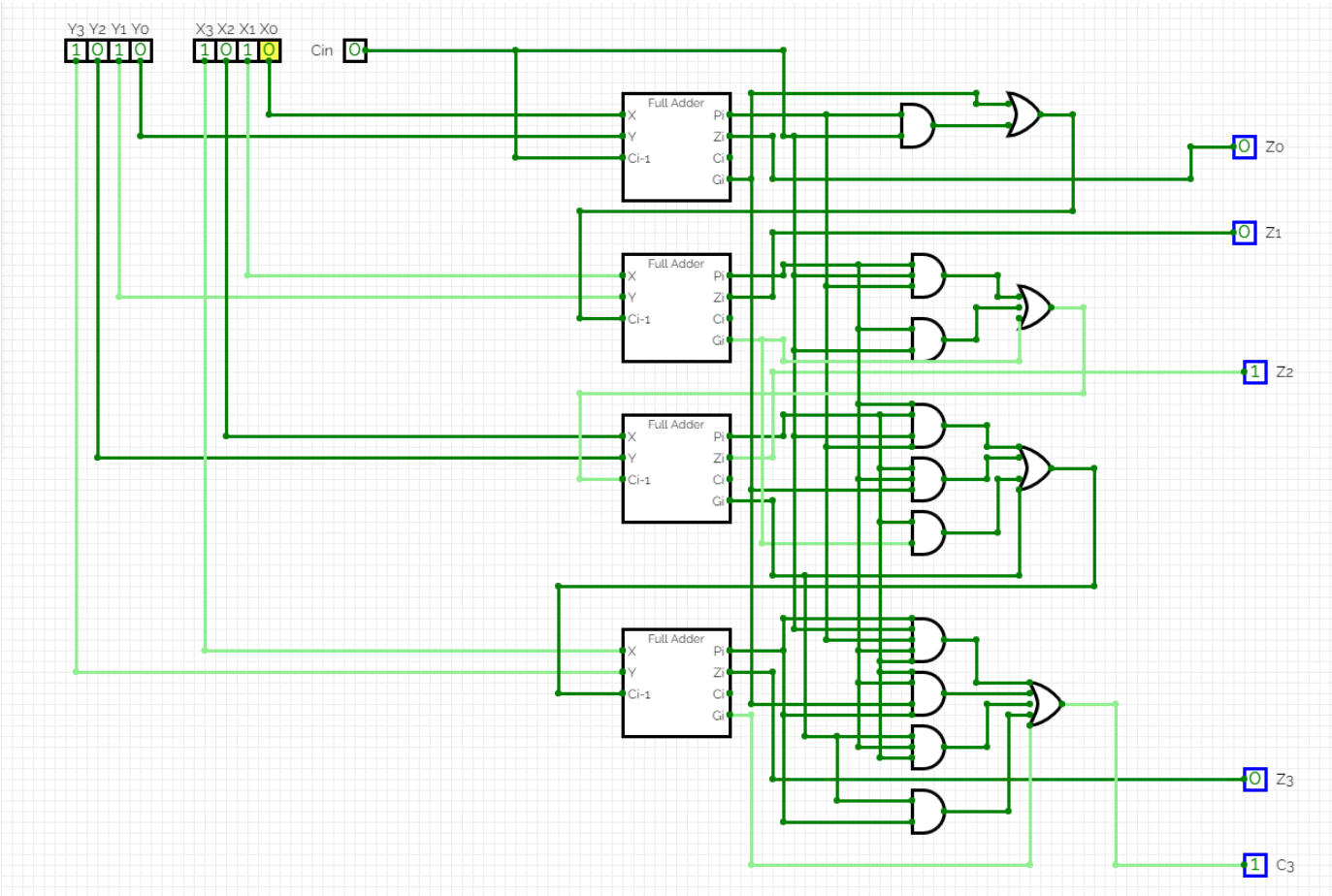

Y3 Y2 Y1 Y0
0 1 1 0
X3 X2 X1 X0
0 1 1 0
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
0 Z1
1 Z2
1 Z3
0 C3

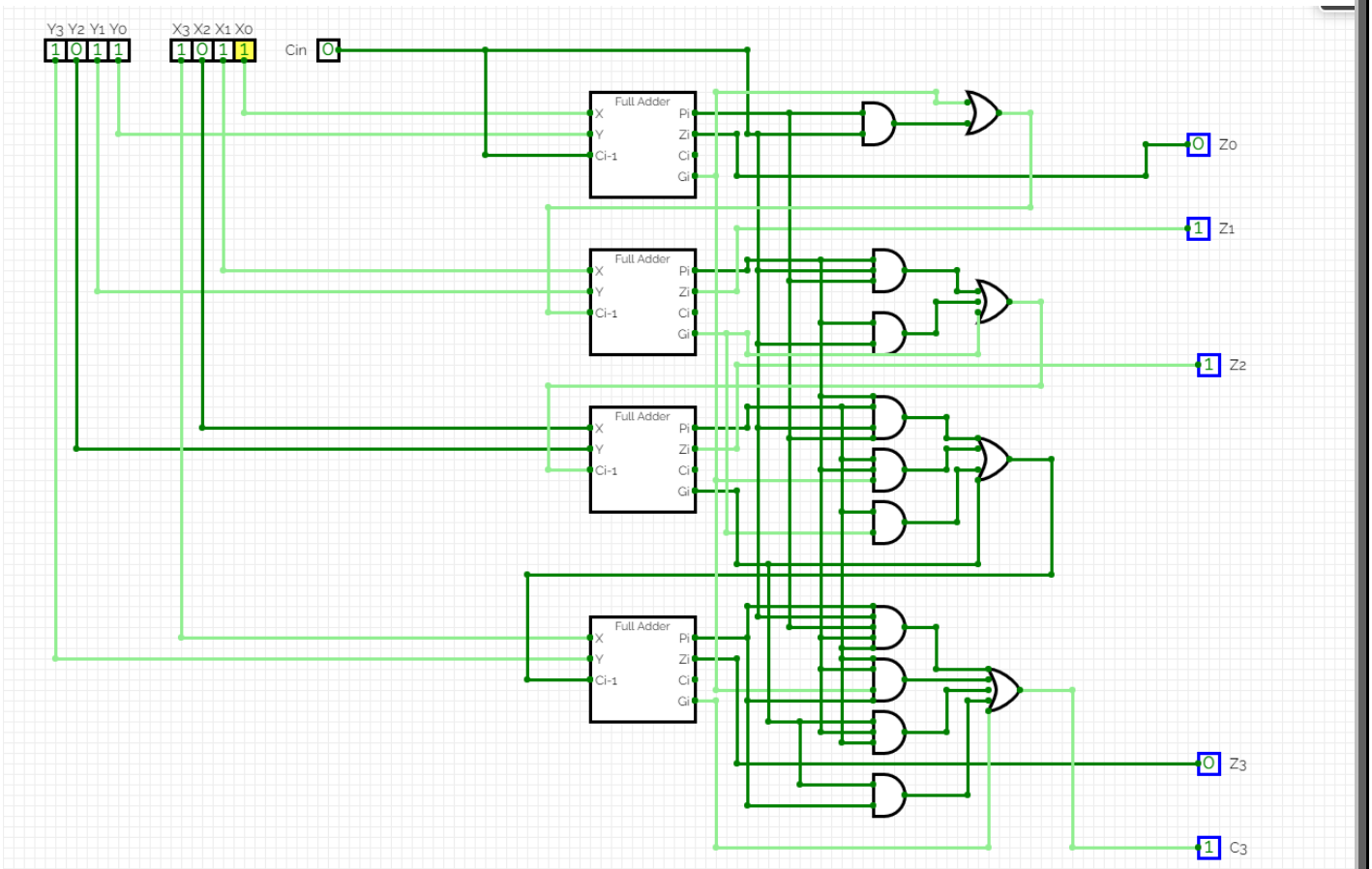

Y3 Y2 Y1 Y0
0 1 1 0
X3 X2 X1 X0
0 1 1 0
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
0 Z1
1 Z2
1 Z3
0 C3

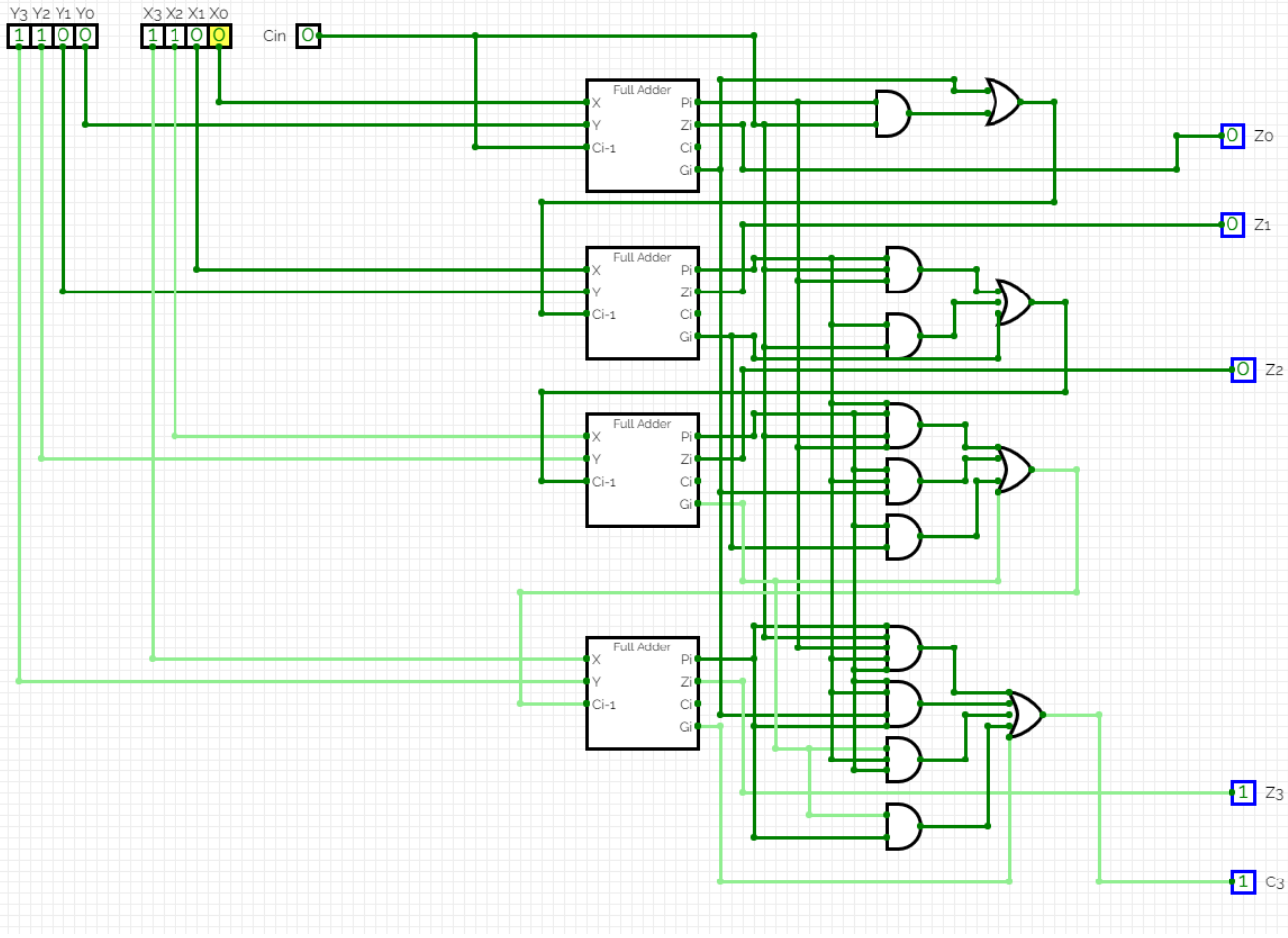

Y3 Y2 Y1 Y0
0 1 1 1
X3 X2 X1 X0
0 1 1 1
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
1 Z1
1 Z2
1 Z3
0 C3

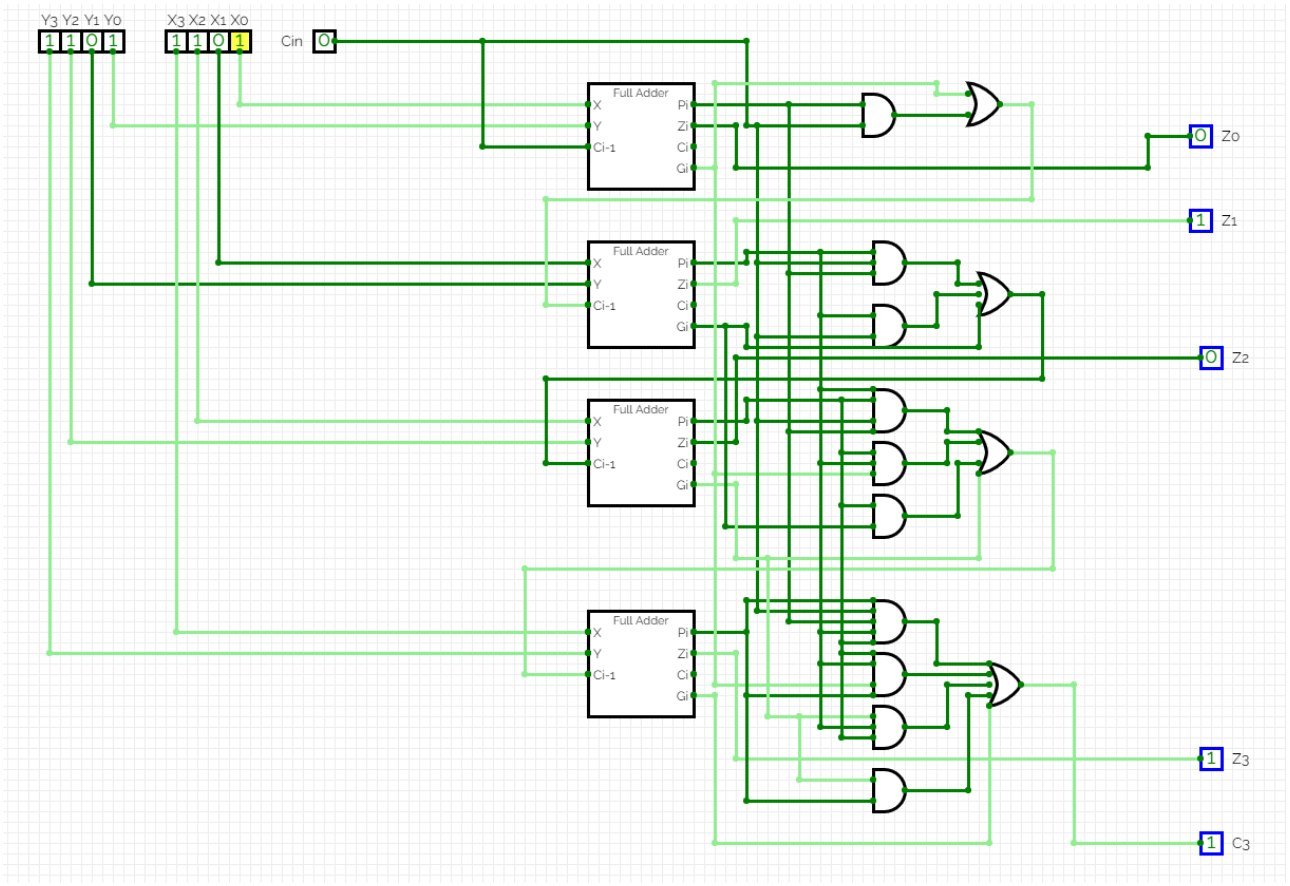

Y3 Y2 Y1 Y0
1 0 0 0
X3 X2 X1 X0
1 0 0 0
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
0 Z1
0 Z2
0 Z3
1 C3

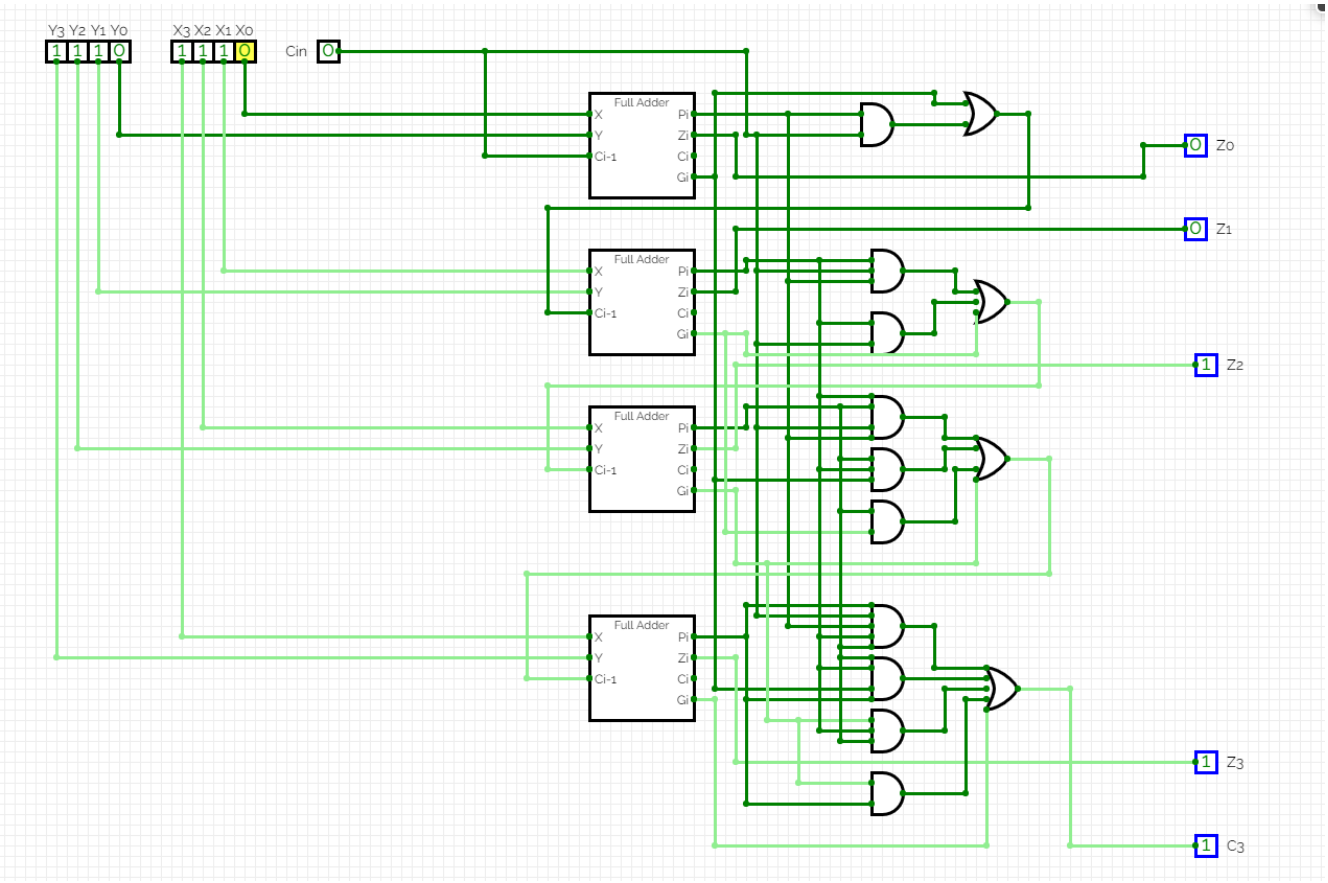

Y3 Y2 Y1 Y0
1 0 0 1
X3 X2 X1 X0
1 0 0 1
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
1 Z1
0 Z2
0 Z3
1 C3

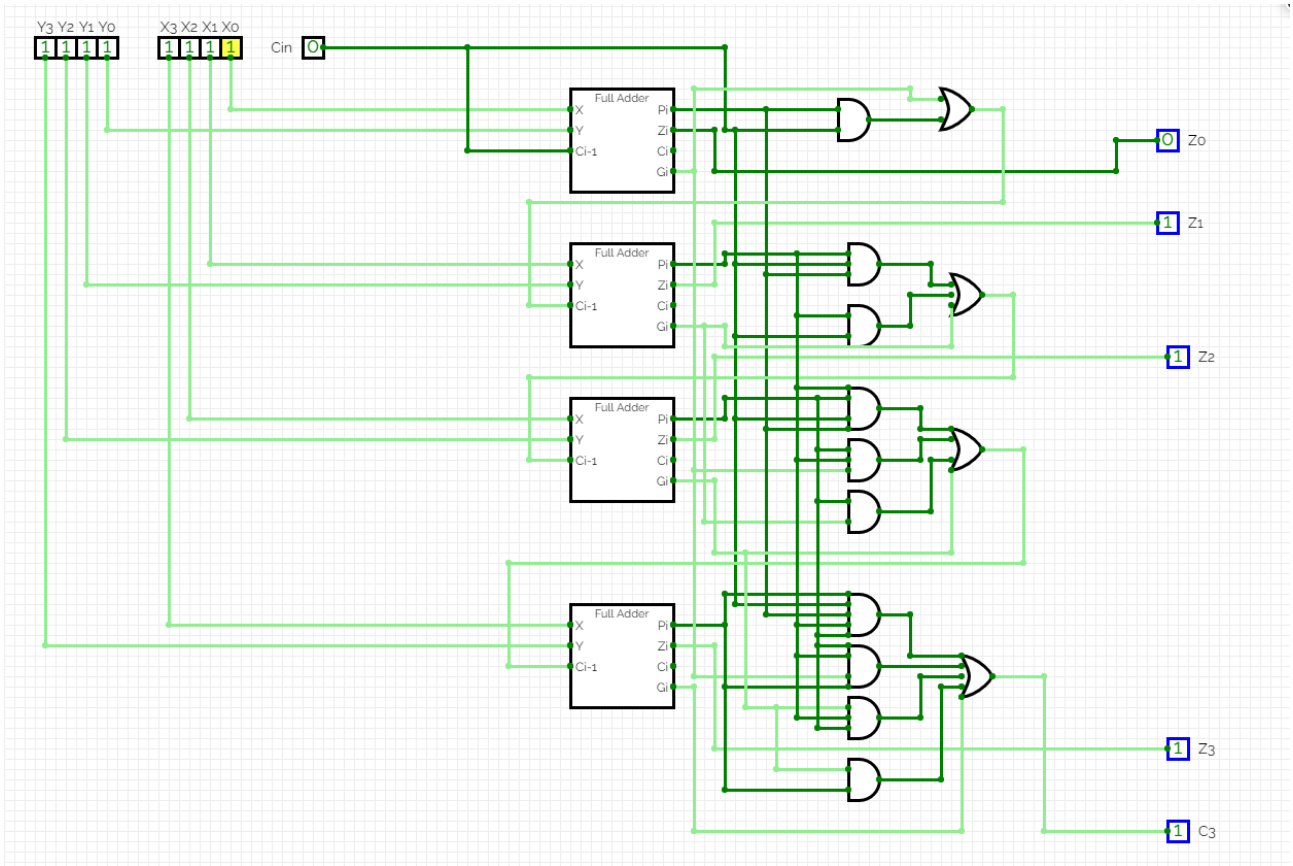

Y3 Y2 Y1 Y0
1 0 1 0
X3 X2 X1 X0
1 0 1 0
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
0 Z1
1 Z2
0 Z3
1 C3

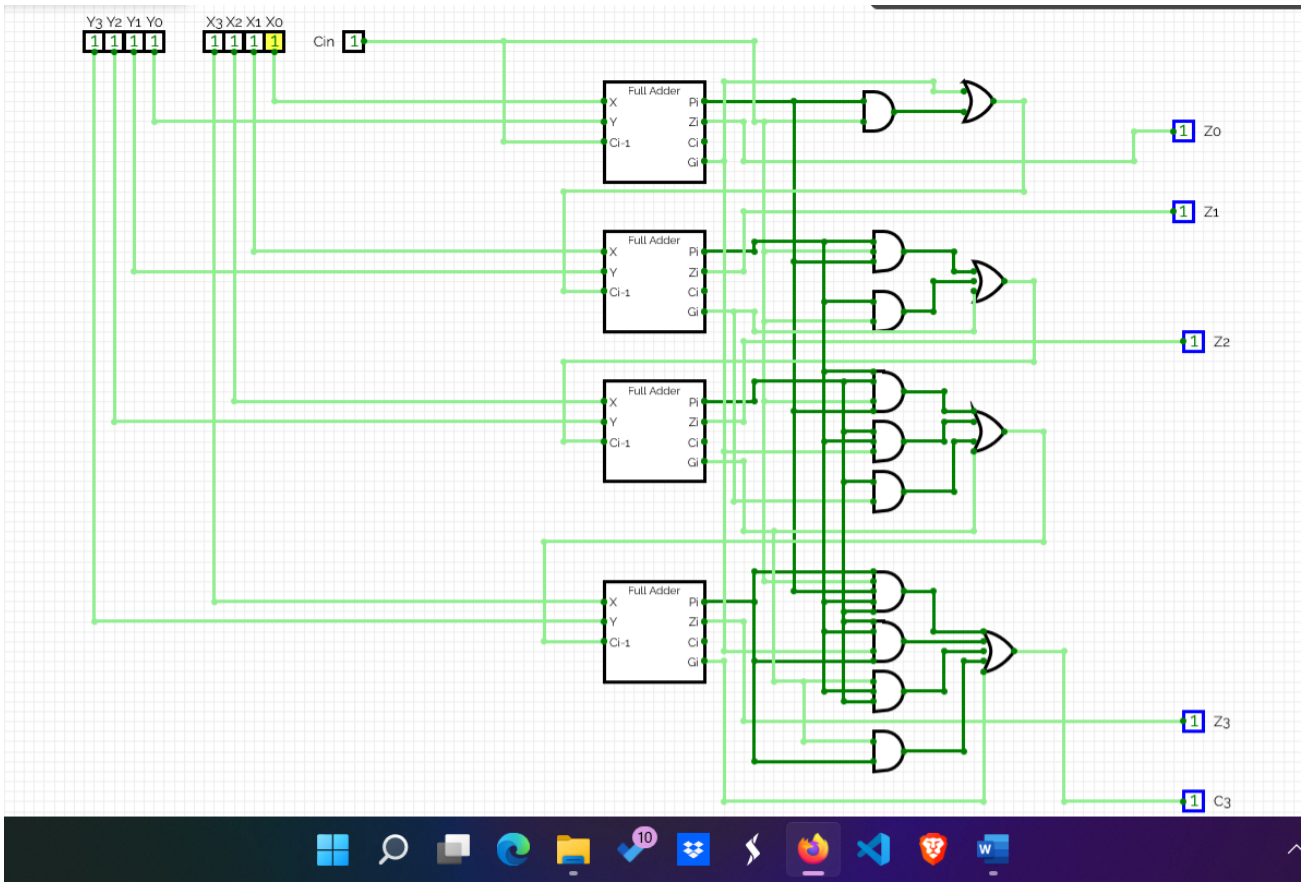

Y3 Y2 Y1 Y0
1 0 1 1
X3 X2 X1 X0
1 0 1 1
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
1 Z1
1 Z2
0 Z3
1 C3


Y3 Y2 Y1 Y0
1 1 0 0
X3 X2 X1 X0
1 1 0 0
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
0 Z1
0 Z2
1 Z3
1 C3

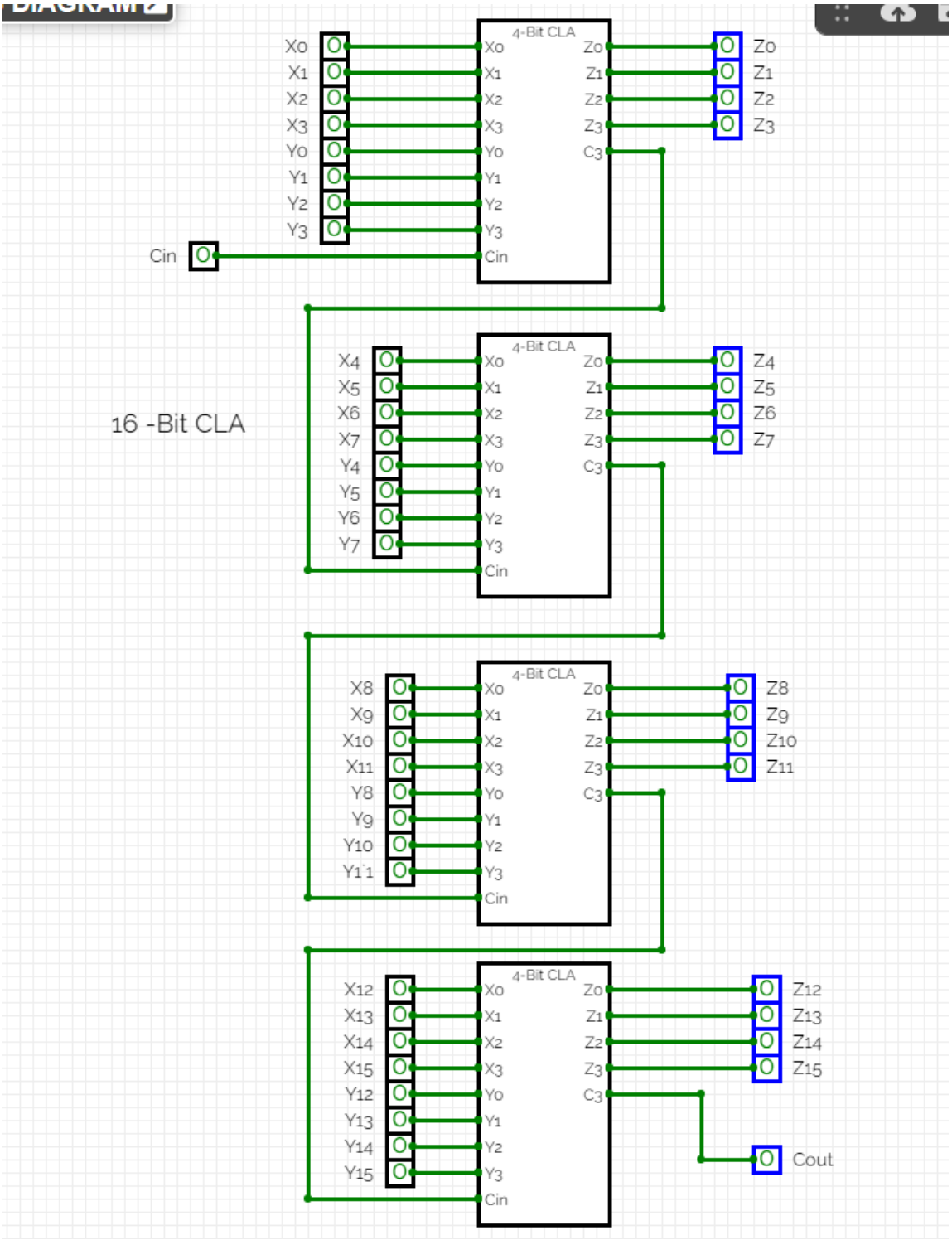

Y3 Y2 Y1 Y0
1 1 0 1
X3 X2 X1 X0
1 1 0 1
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
1 Z1
0 Z2
1 Z3
1 C3

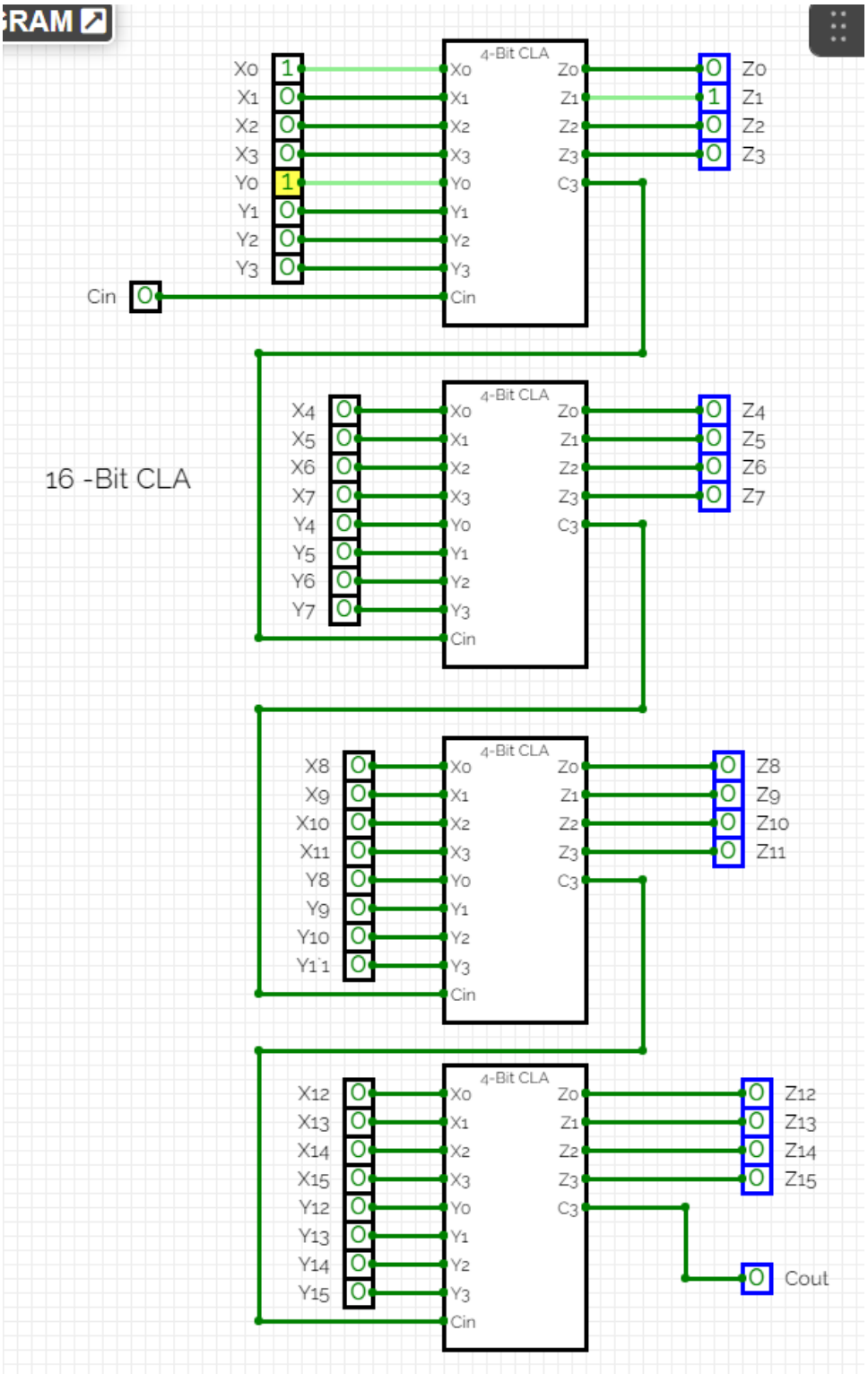

Y3 Y2 Y1 Y0
1 1 1 0
X3 X2 X1 X0
1 1 1 0
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
0 Z1
1 Z2
1 Z3
1 C3

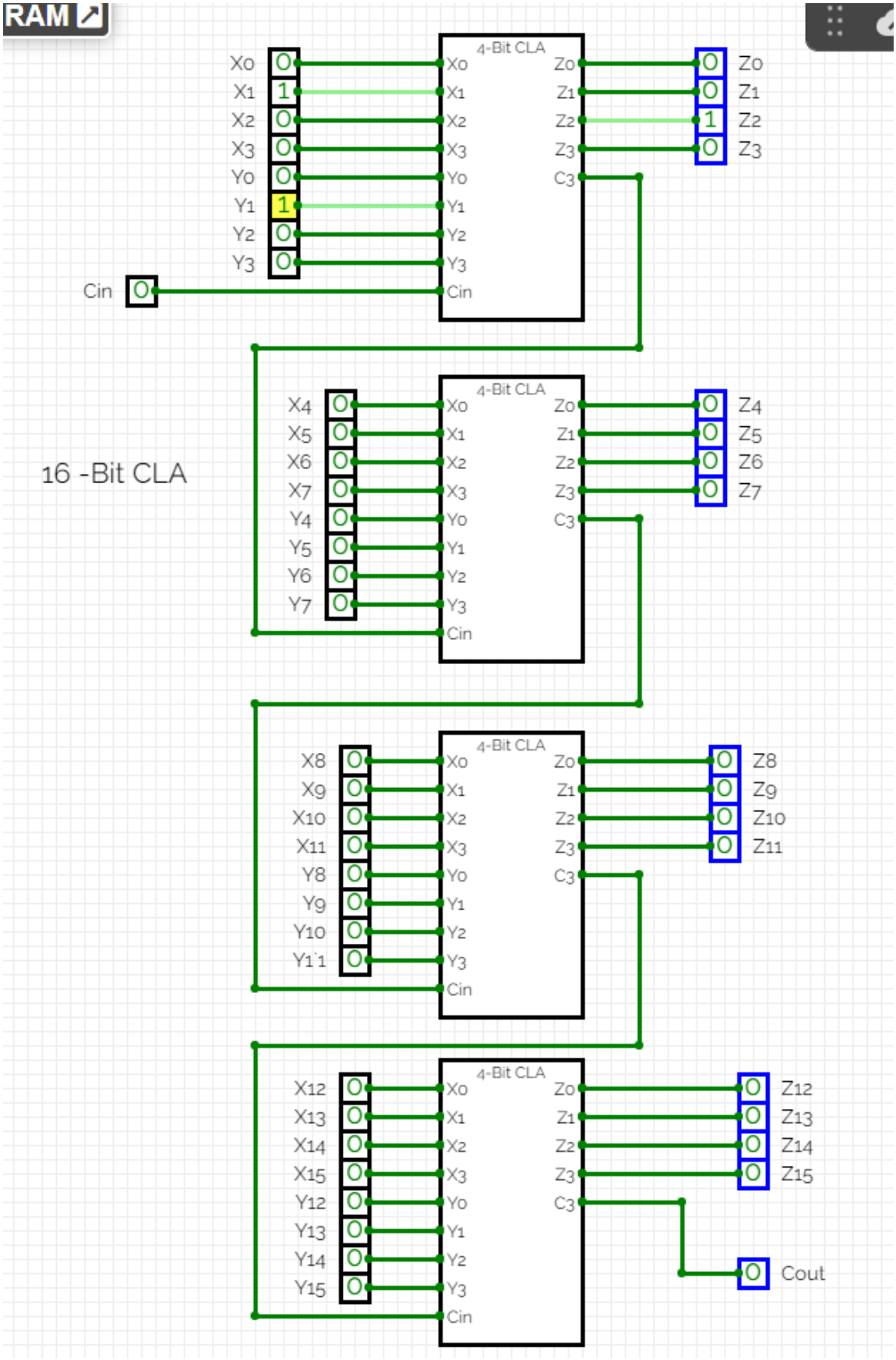

Y3 Y2 Y1 Y0
1 1 1 1
X3 X2 X1 X0
1 1 1 1
Cin
0
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
0 Z0
1 Z1
1 Z2
1 Z3
1 C3

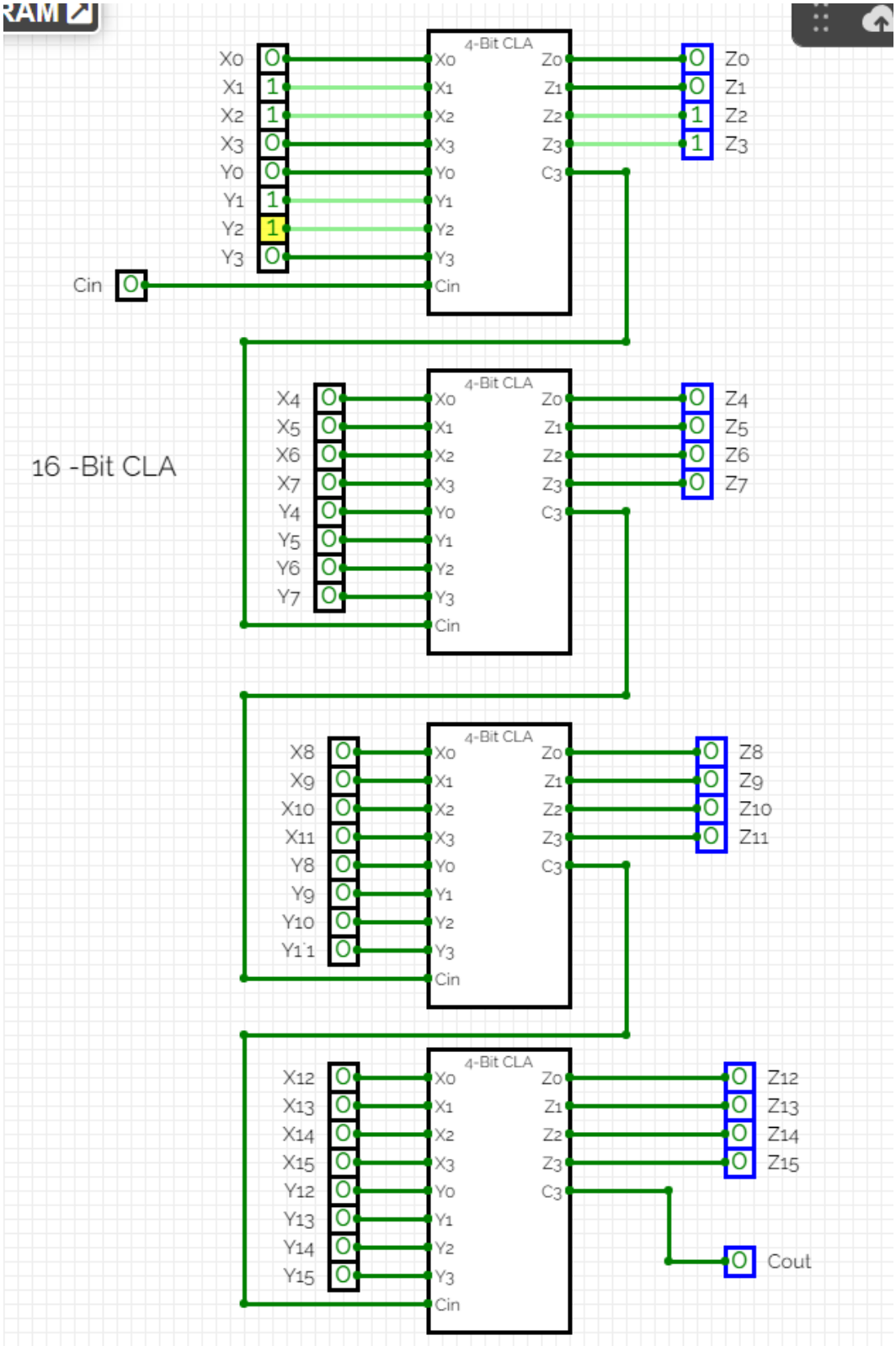

Y3 Y2 Y1 Y0
1 1 1 1
X3 X2 X1 X0
1 1 1 1
Cin
1
Full Adder
X
Y
Ci-1
Pi
Zi
Ci
Gi
1 Z0
1 Z1
1 Z2
1 Z3
1 C3

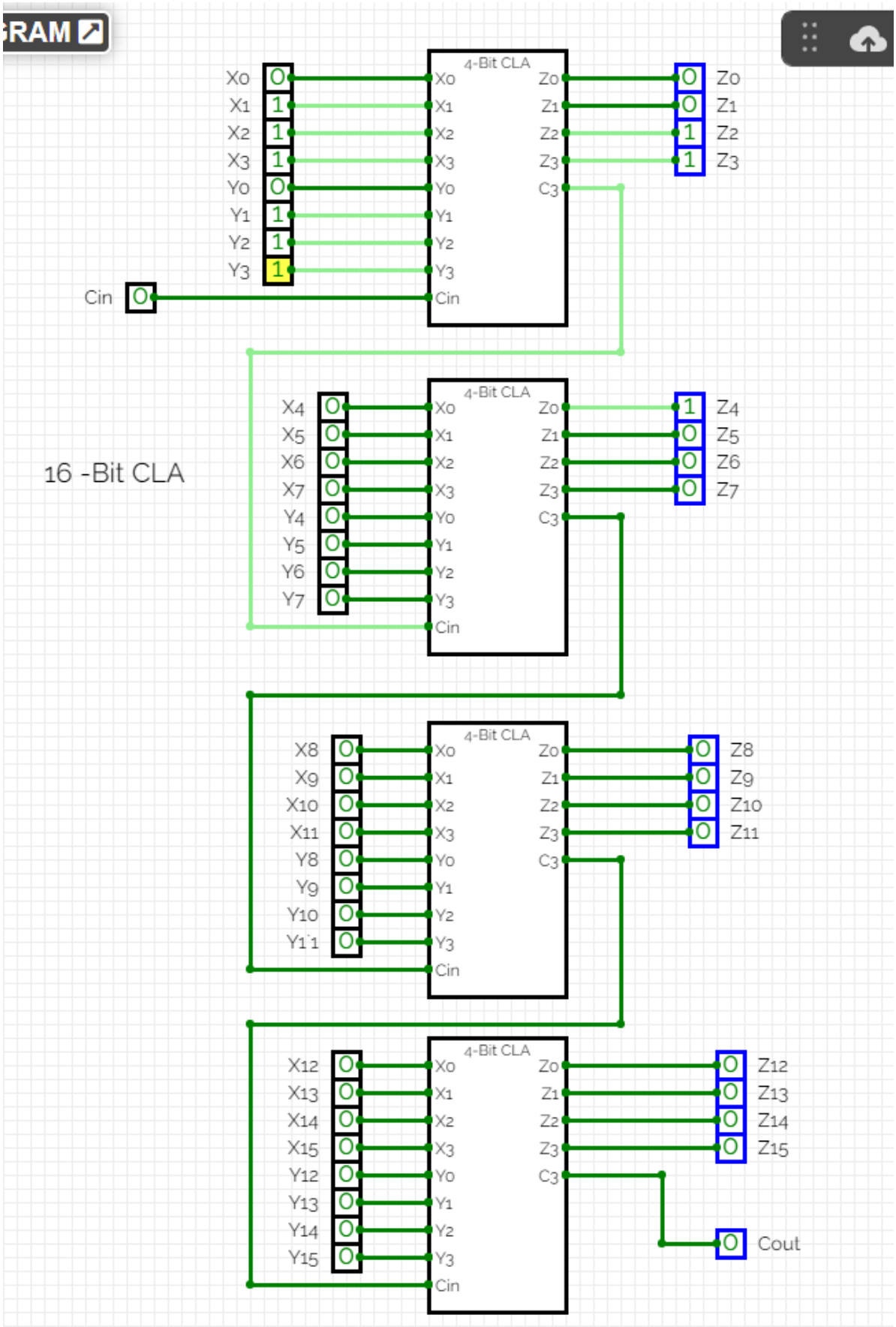
## 16-bit CLA

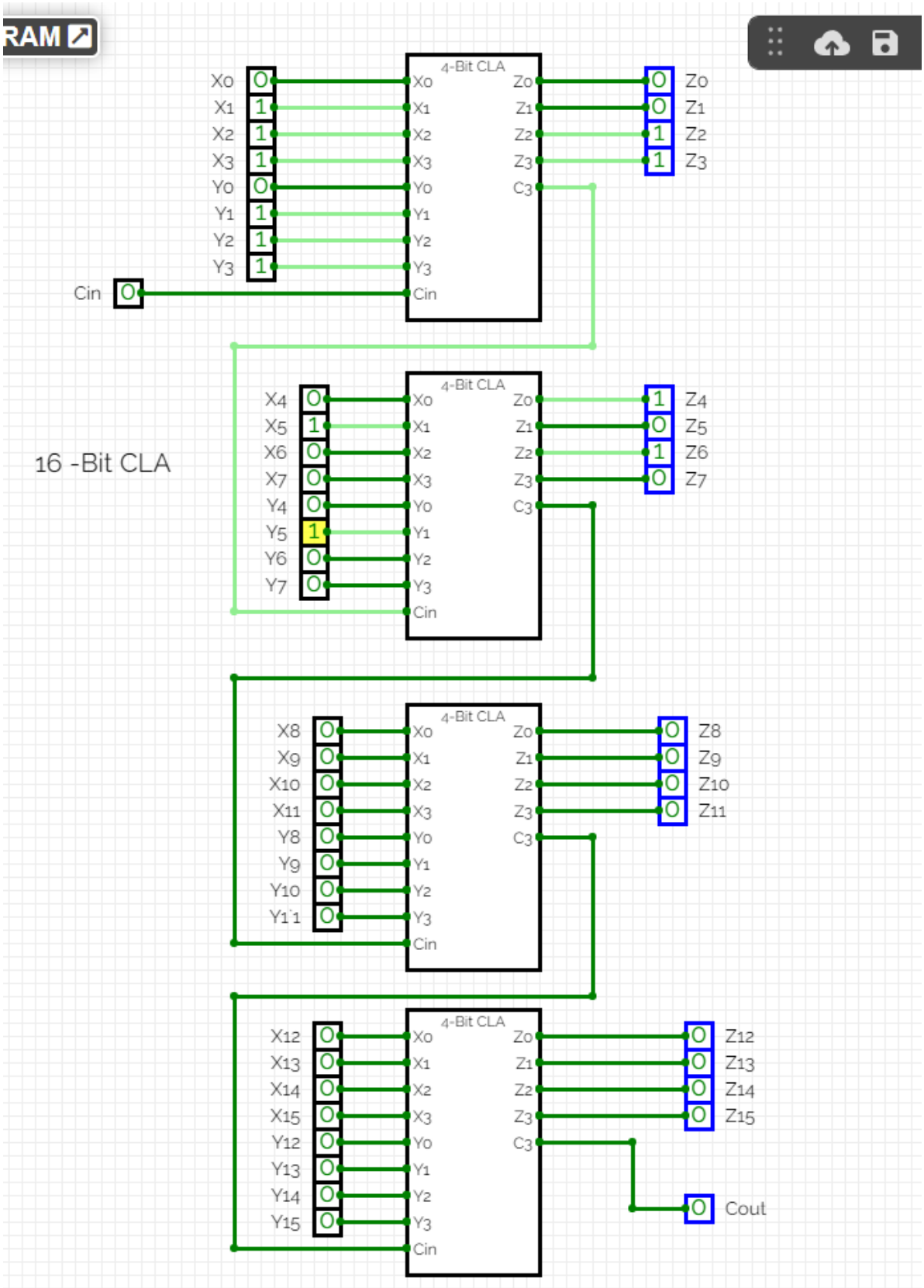

16 -Bit CLA
4-Bit CLA
X0
X1
X2
X3
Y0
Y1
Y2
Y3
Cin
Z0
Z1
Z2
Z3
C3
X4
X5
X6
X7
Y4
Y5
Y6
Y7
Z4
Z5
Z6
Z7
X8
X9
X10
X11
Y8
Y9
Y10
Y11
Z8
Z9
Z10
Z11
X12
X13
X14
X15
Y12
Y13
Y14
Y15
Z12
Z13
Z14
Z15
Cout

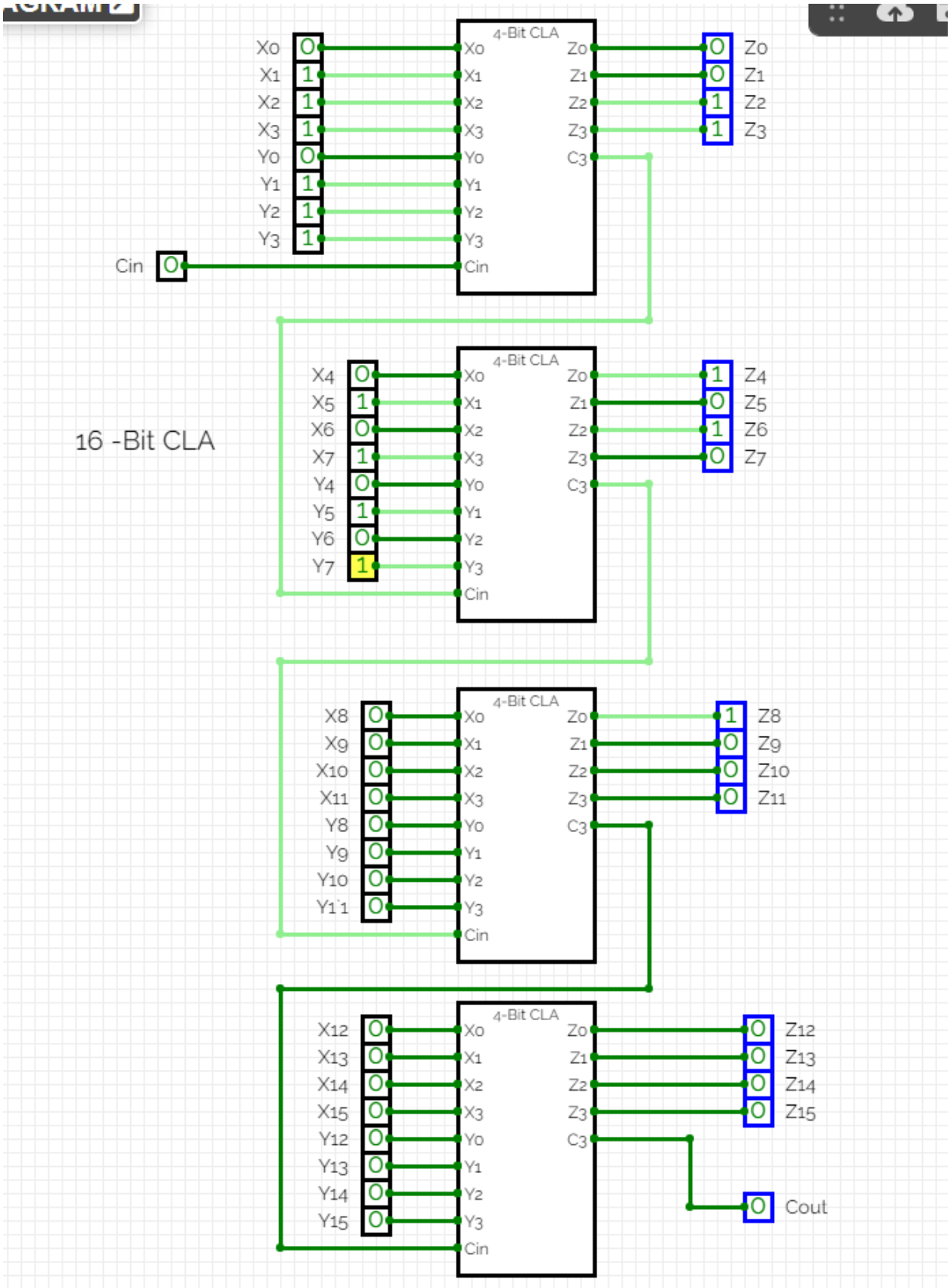

RAM
16 -Bit CLA
4-Bit CLA
X0 1
X1 0
X2 0
X3 0
Y0 1
Y1 0
Y2 0
Y3 0
Cin 0
Z0 0
Z1 1
Z2 0
Z3 0
C3
X4 0
X5 0
X6 0
X7 0
Y4 0
Y5 0
Y6 0
Y7 0
Z4 0
Z5 0
Z6 0
Z7 0
X8 0
X9 0
X10 0
X11 0
Y8 0
Y9 0
Y10 0
Y11 0
Z8 0
Z9 0
Z10 0
Z11 0
X12 0
X13 0
X14 0
X15 0
Y12 0
Y13 0
Y14 0
Y15 0
Z12 0
Z13 0
Z14 0
Z15 0
Cout 0

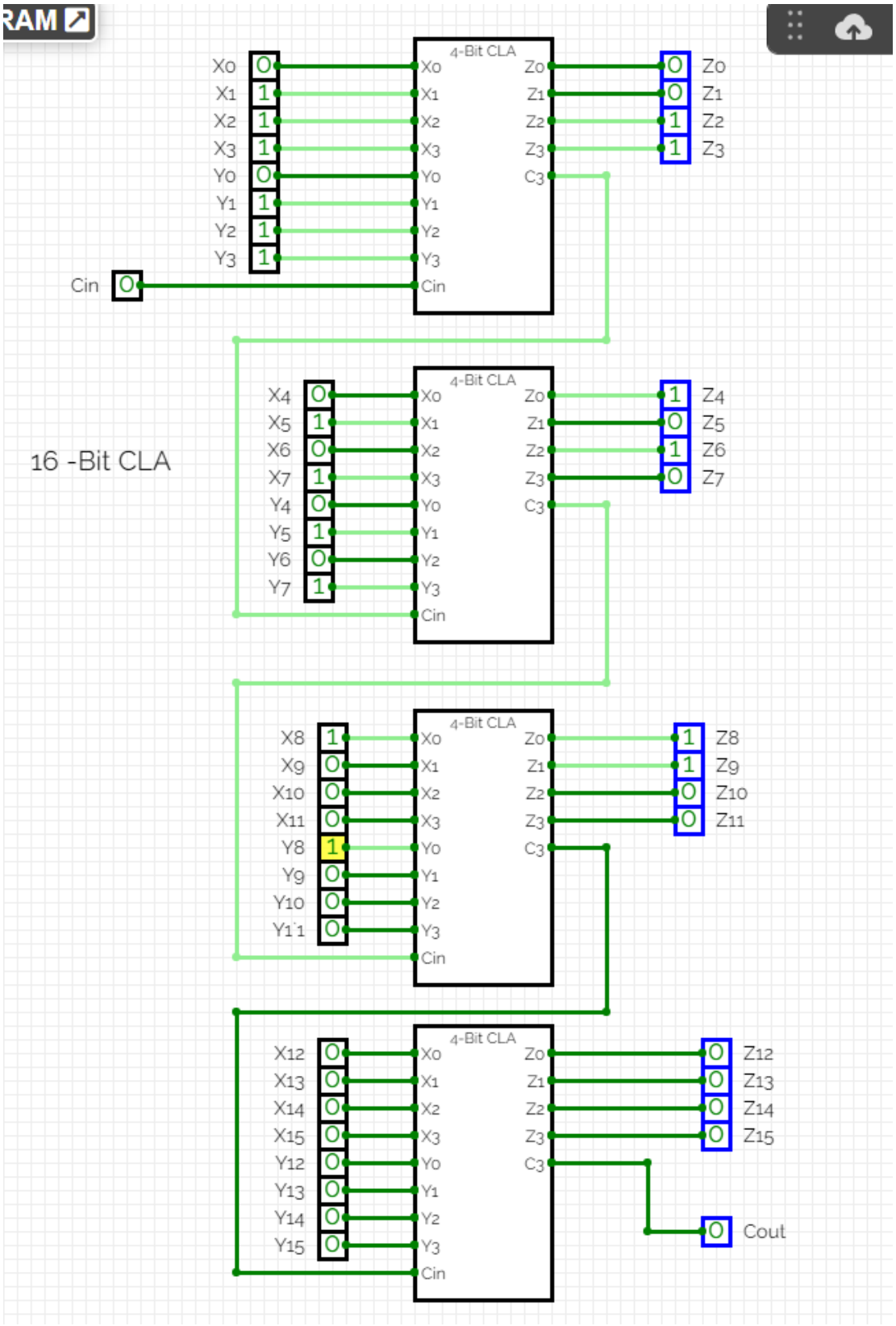

RAM
16 -Bit CLA
4-Bit CLA
X0
X1
X2
X3
Y0
Y1
Y2
Y3
Cin
Z0
Z1
Z2
Z3
C3
X4
X5
X6
X7
Y4
Y5
Y6
Y7
Z4
Z5
Z6
Z7
X8
X9
X10
X11
Y8
Y9
Y10
Y11
Z8
Z9
Z10
Z11
X12
X13
X14
X15
Y12
Y13
Y14
Y15
Z12
Z13
Z14
Z15
Cout

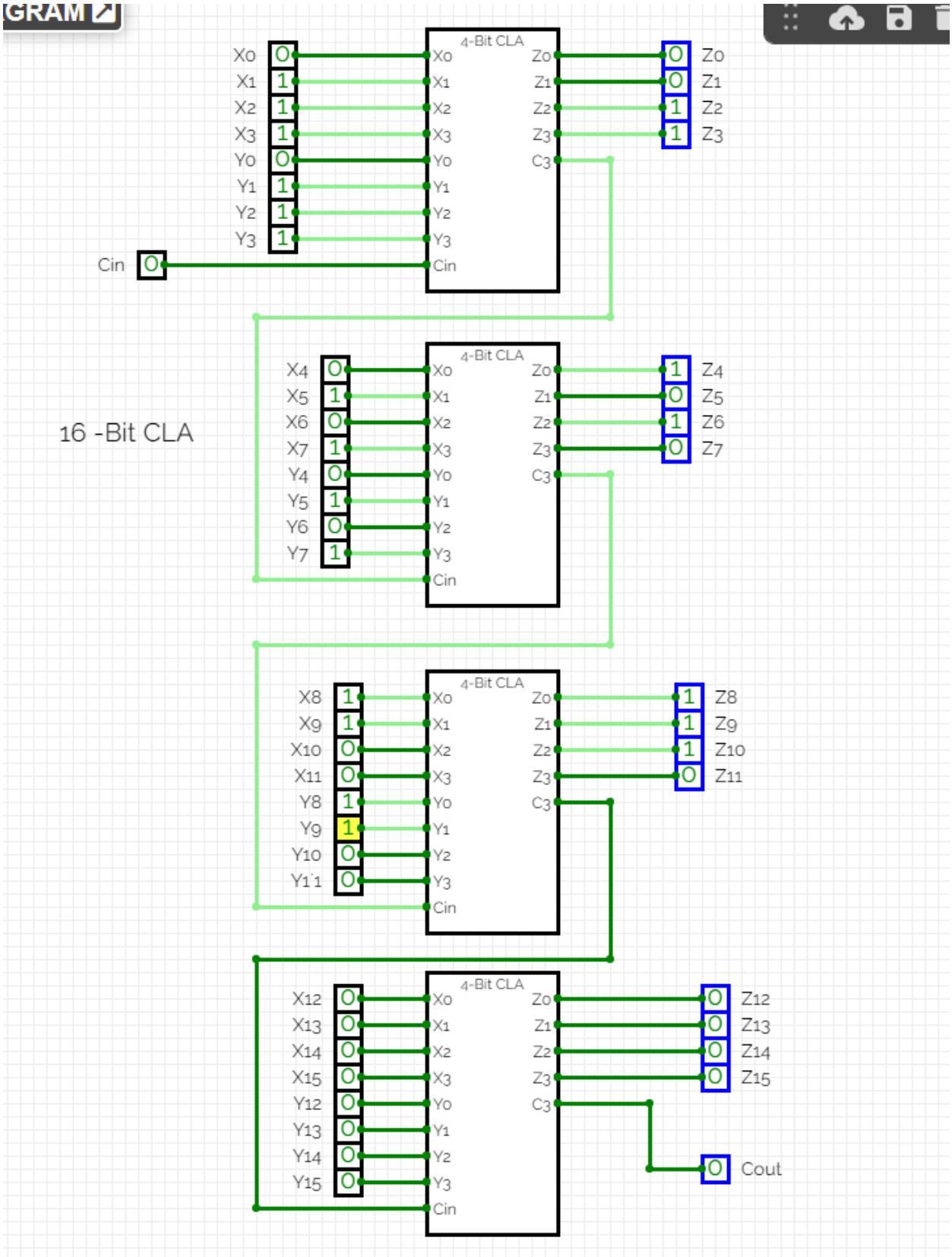

RAM
16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
Z4 Z5 Z6 Z7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
Z8 Z9 Z10 Z11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Z12 Z13 Z14 Z15
Cout

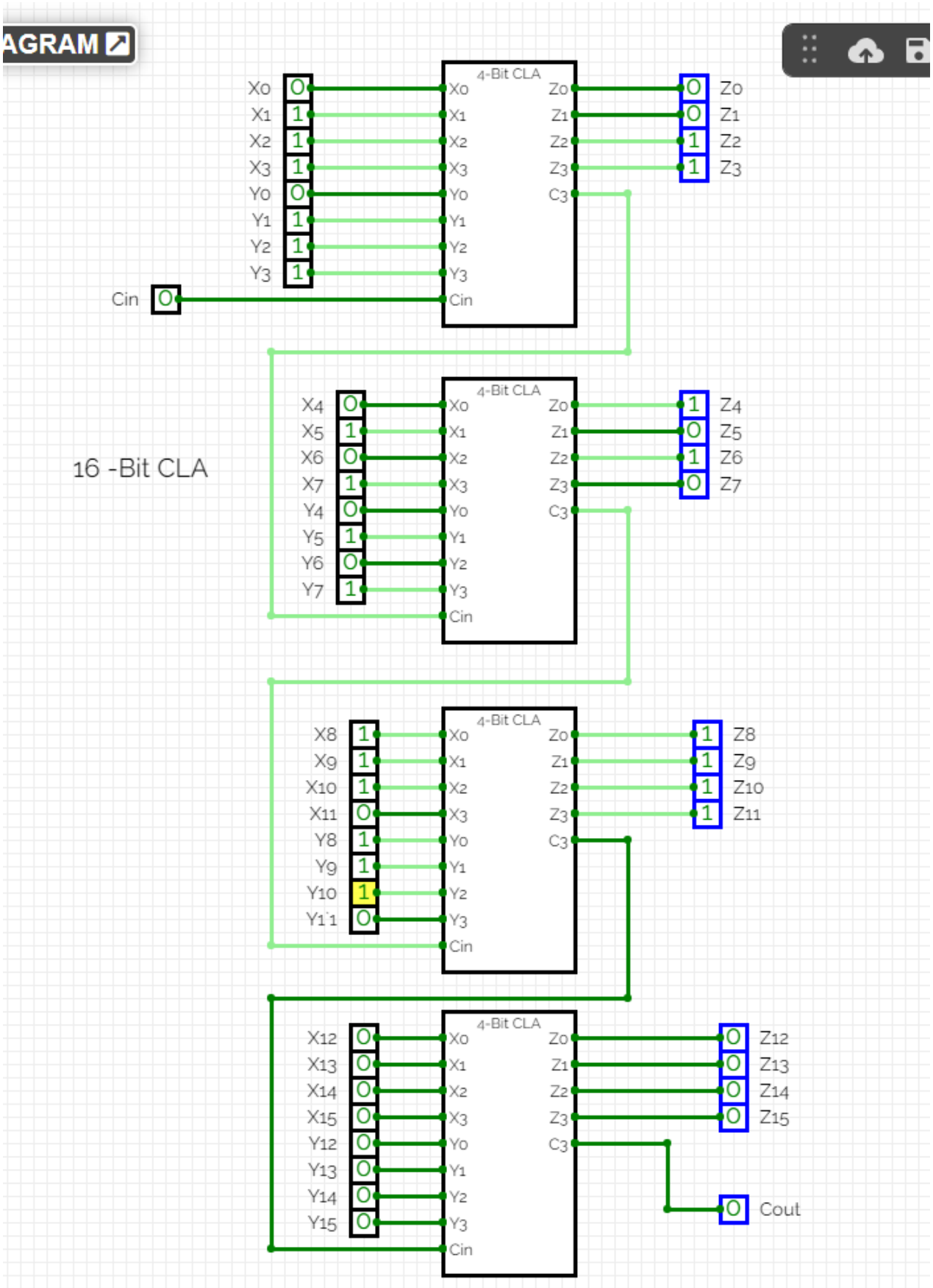

RAM
16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
Z4 Z5 Z6 Z7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
Z8 Z9 Z10 Z11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Z12 Z13 Z14 Z15
Cout

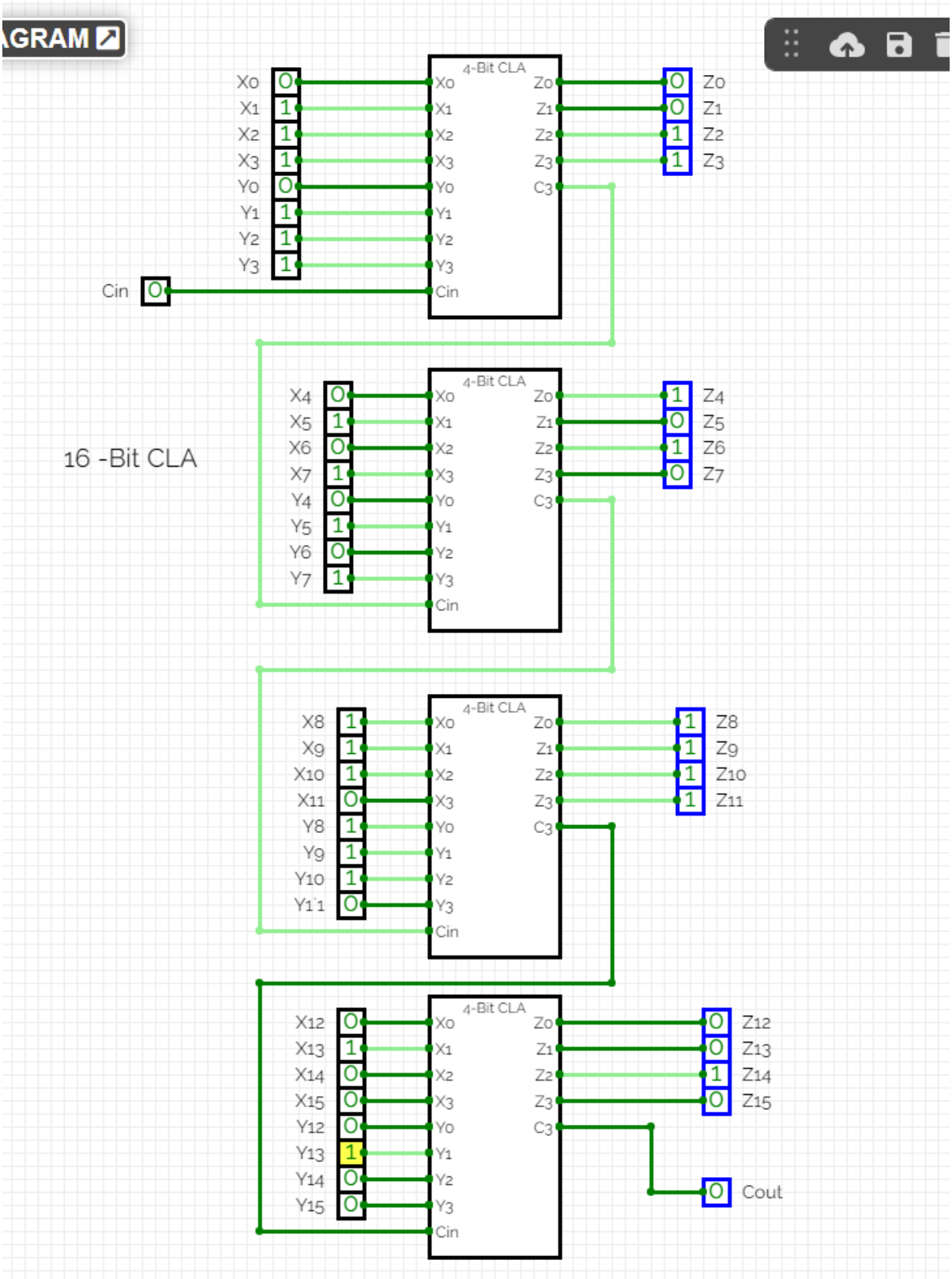

RAM
16 -Bit CLA
4-Bit CLA
X0 0
X1 1
X2 1
X3 1
Y0 0
Y1 1
Y2 1
Y3 1
Cin 0
Z0 0
Z1 0
Z2 1
Z3 1
C3
X4 0
X5 1
X6 0
X7 0
Y4 0
Y5 1
Y6 0
Y7 0
Z4 1
Z5 0
Z6 1
Z7 0
X8 0
X9 0
X10 0
X11 0
Y8 0
Y9 0
Y10 0
Y11 0
Z8 0
Z9 0
Z10 0
Z11 0
X12 0
X13 0
X14 0
X15 0
Y12 0
Y13 0
Y14 0
Y15 0
Z12 0
Z13 0
Z14 0
Z15 0
Cout 0

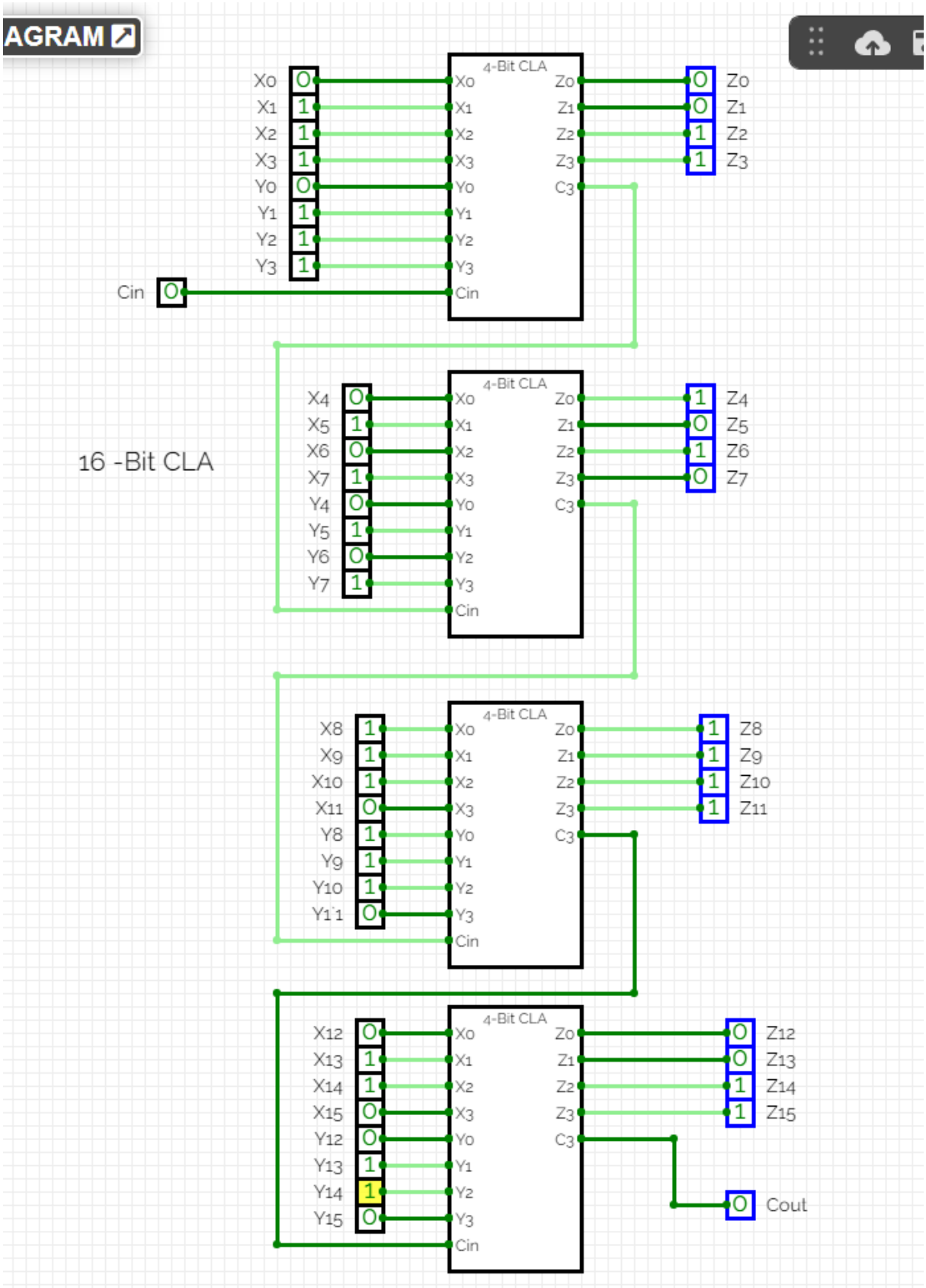

16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
Z0 Z1 Z2 Z3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
Z4 Z5 Z6 Z7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
Z8 Z9 Z10 Z11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Z12 Z13 Z14 Z15
Cout

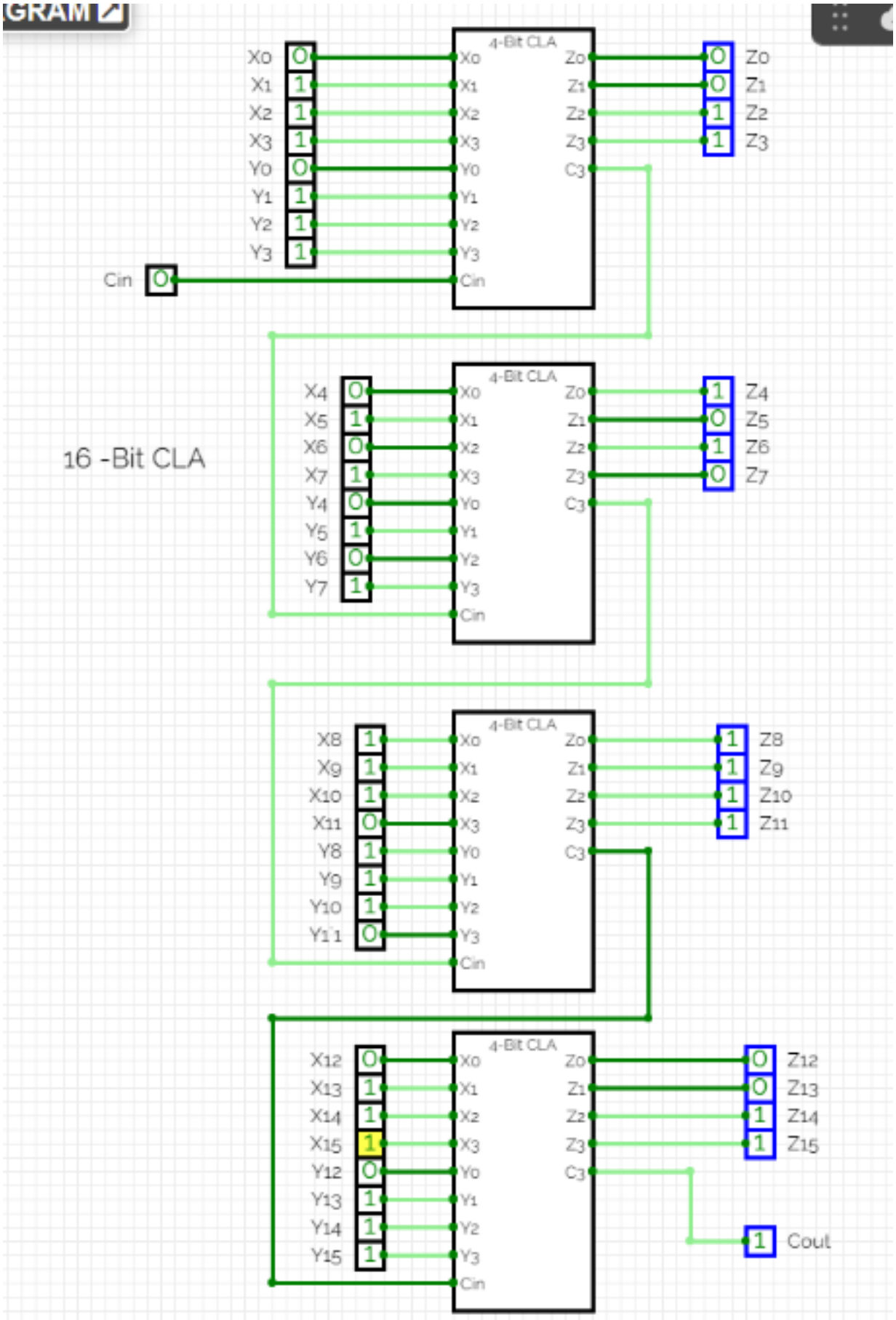

RAM
16 -Bit CLA
4-Bit CLA
X0 0
X1 1
X2 1
X3 1
Y0 0
Y1 1
Y2 1
Y3 1
Cin 0
Z0 0
Z1 0
Z2 1
Z3 1
C3
X4 0
X5 1
X6 0
X7 1
Y4 0
Y5 1
Y6 0
Y7 1
Z4 1
Z5 0
Z6 1
Z7 0
X8 1
X9 0
X10 0
X11 0
Y8 1
Y9 0
Y10 0
Y11 0
Z8 1
Z9 1
Z10 0
Z11 0
X12 0
X13 0
X14 0
X15 0
Y12 0
Y13 0
Y14 0
Y15 0
Z12 0
Z13 0
Z14 0
Z15 0
Cout 0

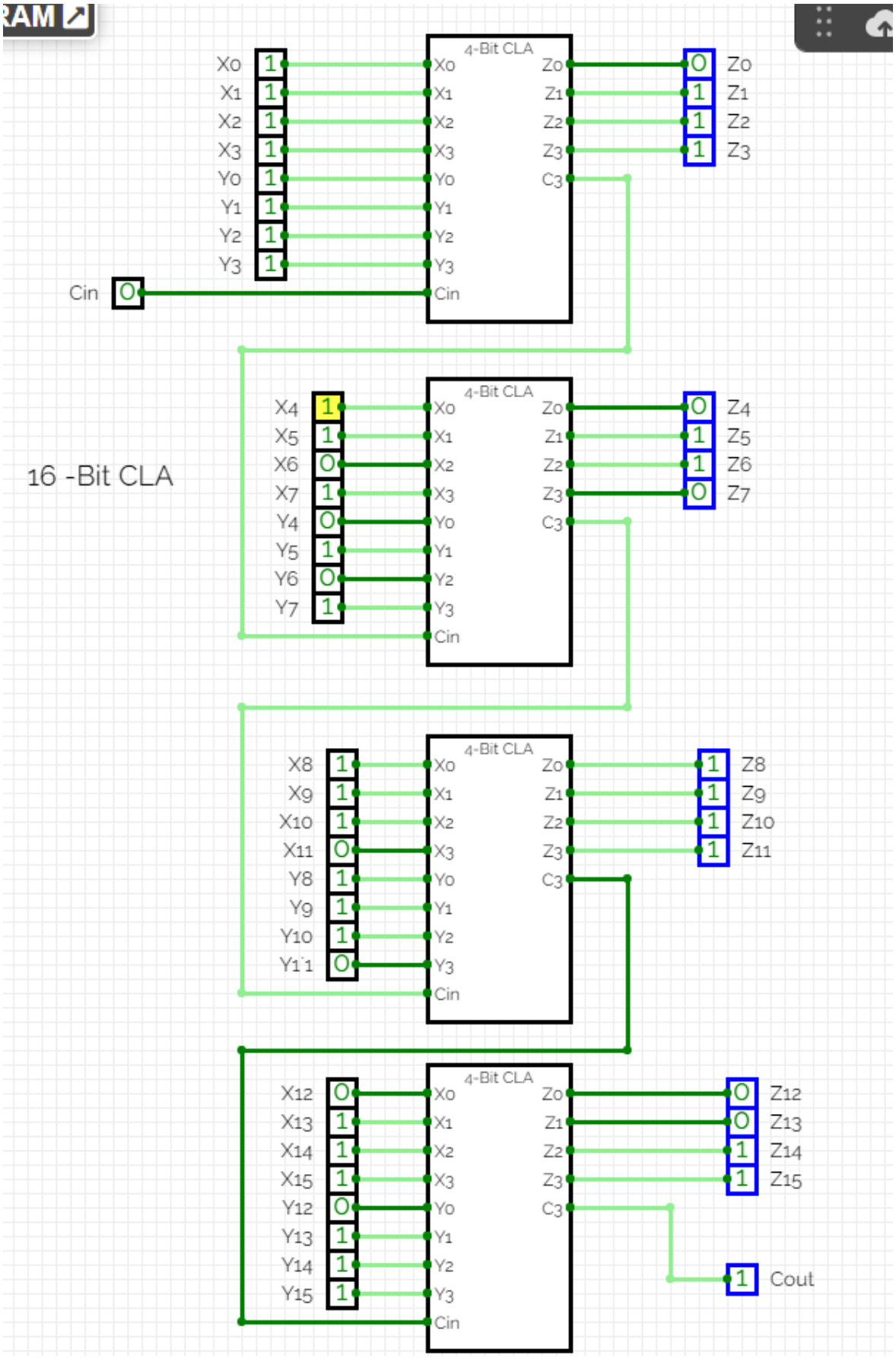

GRAM
16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Cout

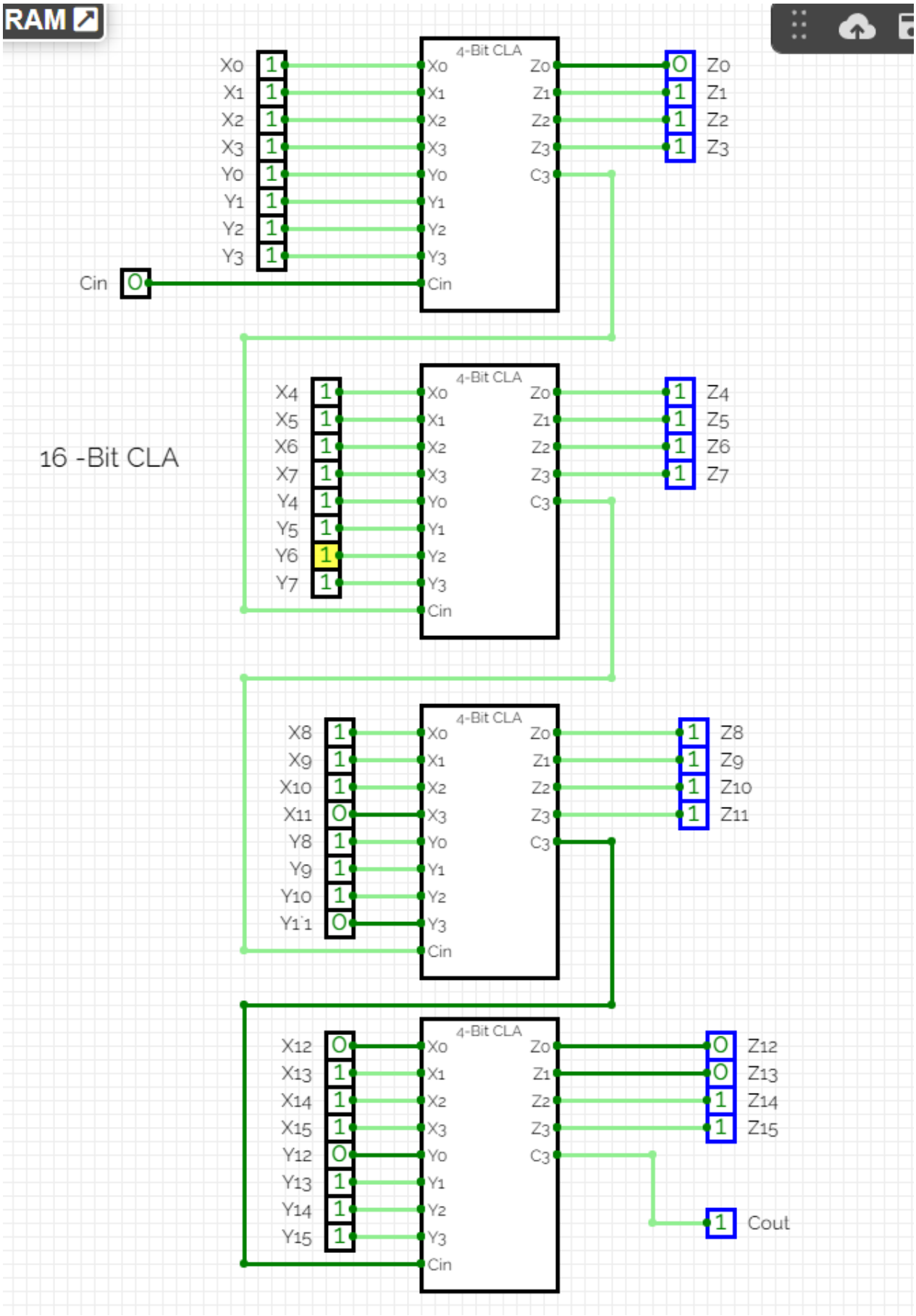

AGRAM
16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Cout

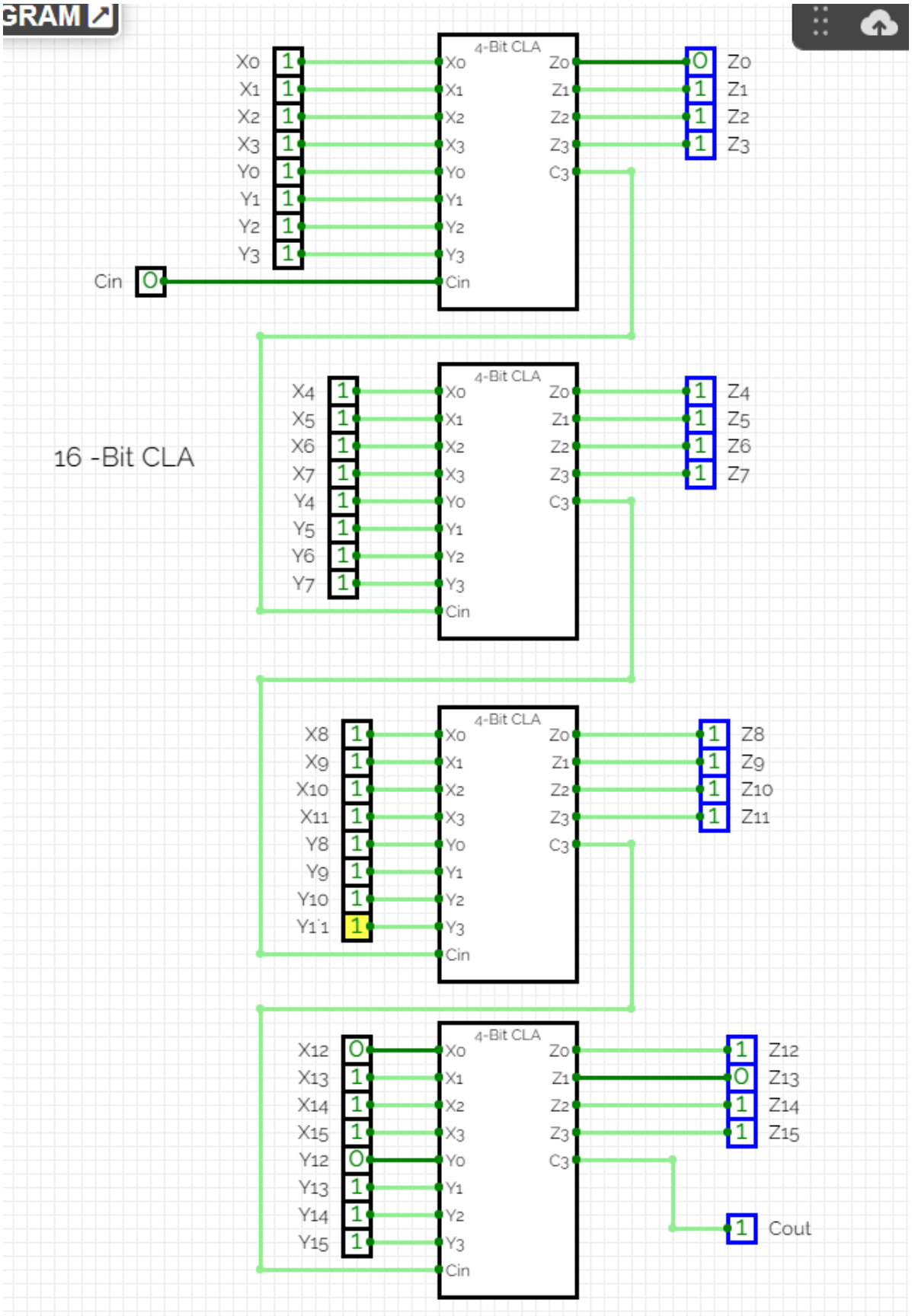

GRAM
16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Cout

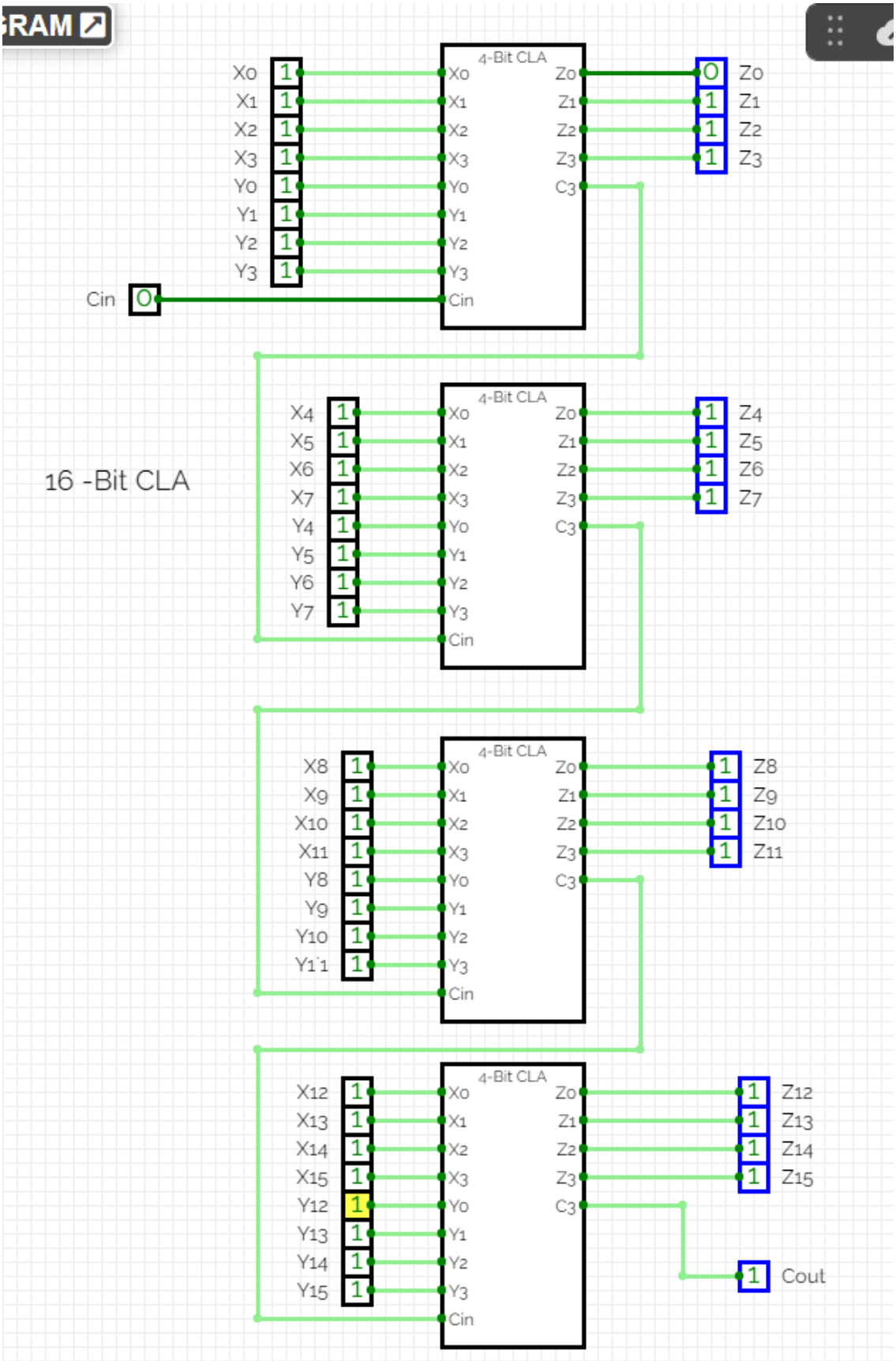

AGRAM
16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
Z4 Z5 Z6 Z7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
Z8 Z9 Z10 Z11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Z12 Z13 Z14 Z15
Cout

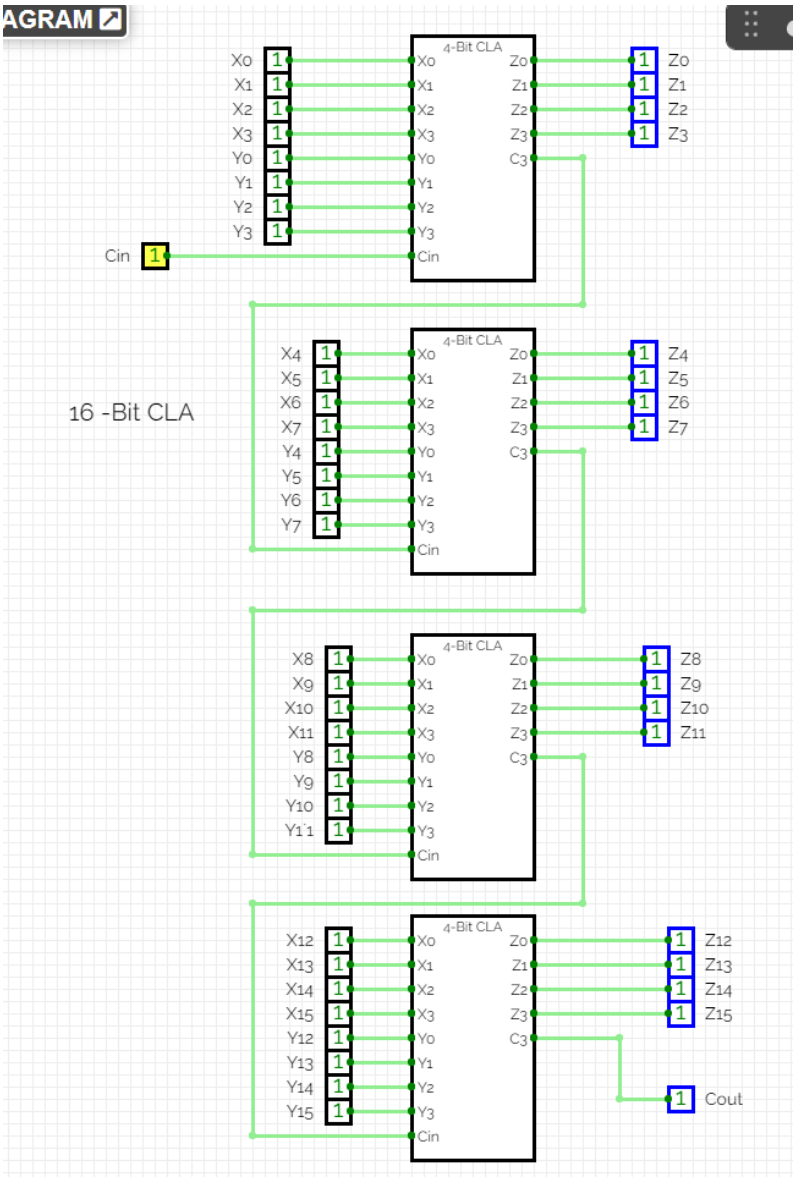

GRAM
16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Z4 Z5 Z6 Z7
Z8 Z9 Z10 Z11
Z12 Z13 Z14 Z15
Coul


16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
Z4 Z5 Z6 Z7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
Z8 Z9 Z10 Z11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Z12 Z13 Z14 Z15
Cout


RAM
16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Z4 Z5 Z6 Z7
Z8 Z9 Z10 Z11
Z12 Z13 Z14 Z15
Cout


GRAM
16 -Bit CLA
4-Bit CLA
X0 X1 X2 X3 Y0 Y1 Y2 Y3 Cin
Z0 Z1 Z2 Z3 C3
X4 X5 X6 X7 Y4 Y5 Y6 Y7
X8 X9 X10 X11 Y8 Y9 Y10 Y11
X12 X13 X14 X15 Y12 Y13 Y14 Y15
Cout


RAM
16 -Bit CLA
4-Bit CLA
X0 1
X1 1
X2 1
X3 1
Y0 1
Y1 1
Y2 1
Y3 1
Cin 0
Z0 0
Z1 1
Z2 1
Z3 1
C3
X4 1
X5 1
X6 1
X7 1
Y4 1
Y5 1
Y6 1
Y7 1
Z4 1
Z5 1
Z6 1
Z7 1
X8 1
X9 1
X10 1
X11 1
Y8 1
Y9 1
Y10 1
Y11 1
Z8 1
Z9 1
Z10 1
Z11 1
X12 1
X13 1
X14 1
X15 1
Y12 1
Y13 1
Y14 1
Y15 1
Z12 1
Z13 1
Z14 1
Z15 1
Cout 1




## 2.7 Verification

• We are successfully able to add two 4-bit numbers using circuit in Figure 8b.
• We are successfully able to add two 16-bit numbers using circuit in Figure 8c

## 2.8 Conclusion

We are able to add two 4-bit number in constant delay time and are able to add two 16-bit numbers
in some delay, which is not proportional to number of bits.

# Experiment No: 5

Implementation of data movement instructions

## 5.1 Objective

To realize data movement instructions in a CPU having four 2-bit registers.

## 5.2 Theoretical Basis

Data movement instructions move data from one place, called the source operand, to another place, called the destination operand.
Data is transferred directly from a source register in a register file connected to a data bus from a destination register in the register file, through a data bus bypass mechanism.
The figure shown below shows how to implement such data movement instructions when the source register and the destination register belong to the same set of registers.

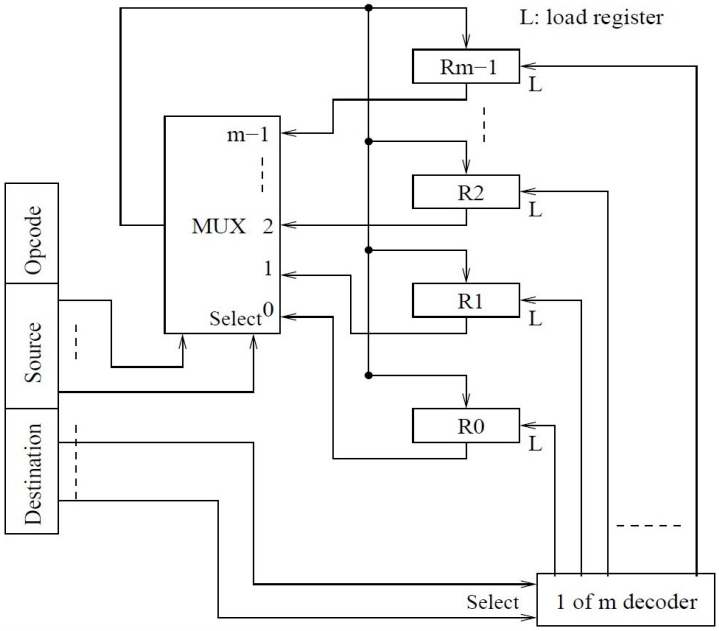



Figure 9: Data Transfer between same set of Registers

## 5.3 Task

Given four 2-bit registers $R_0$, $R_1$, $R_2$, and $R_3$. We need to realize the implementation of data movement instruction like

*MOV* ⟨*dest reg addr*⟩ ⟨*source reg addr*⟩

Here we set source and destination address manually

## 5.4 Major Components/Modules Used

1. Four 2-bit Registers
2. Two 4 to 1 Multiplexer
3. One 1 of 4 Decoder

## 5.5 Design

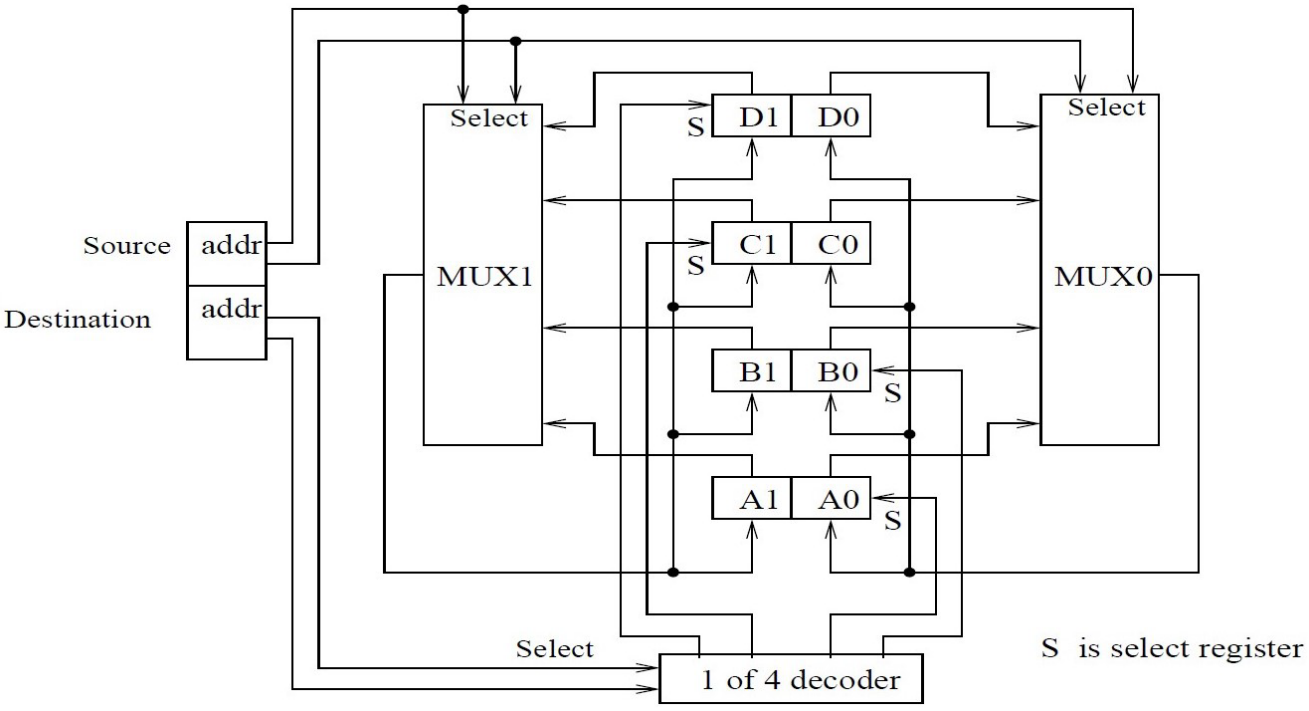
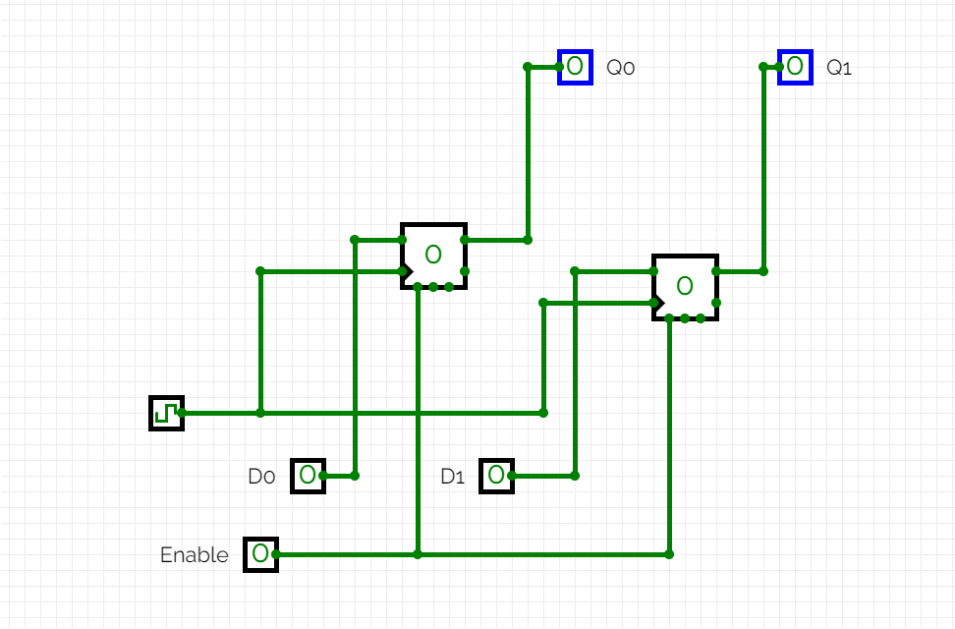


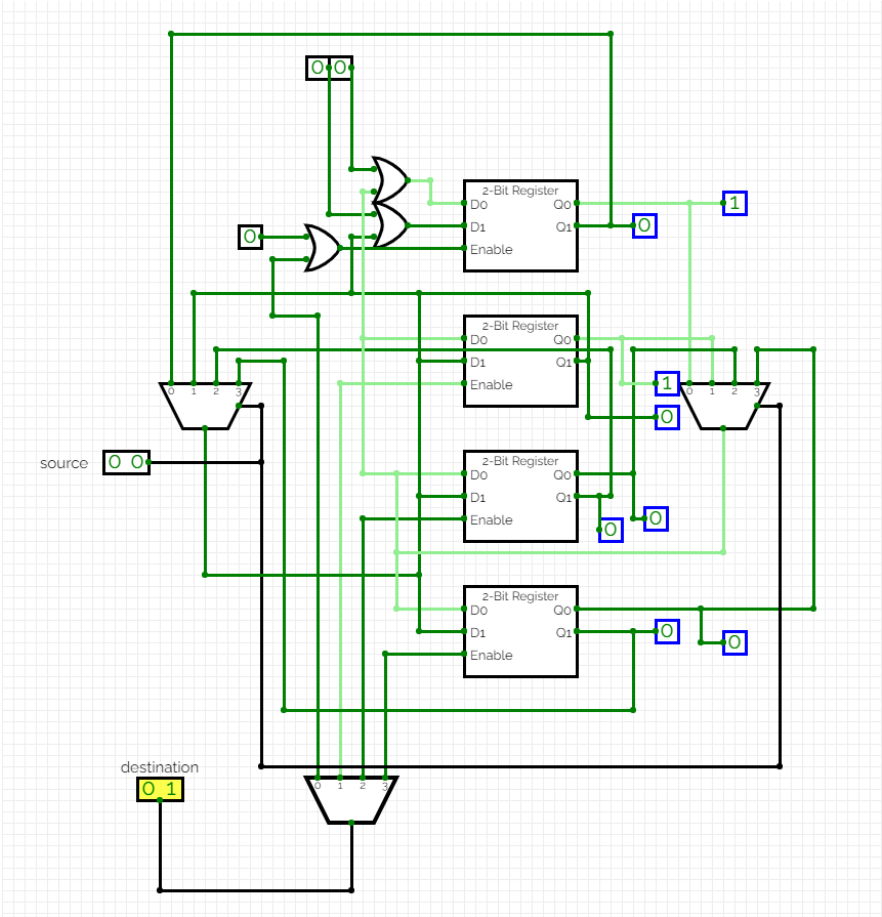

Figure 10: Design Idea to implement Data Transfer between four Registers

## 5.6 Circuit Diagram

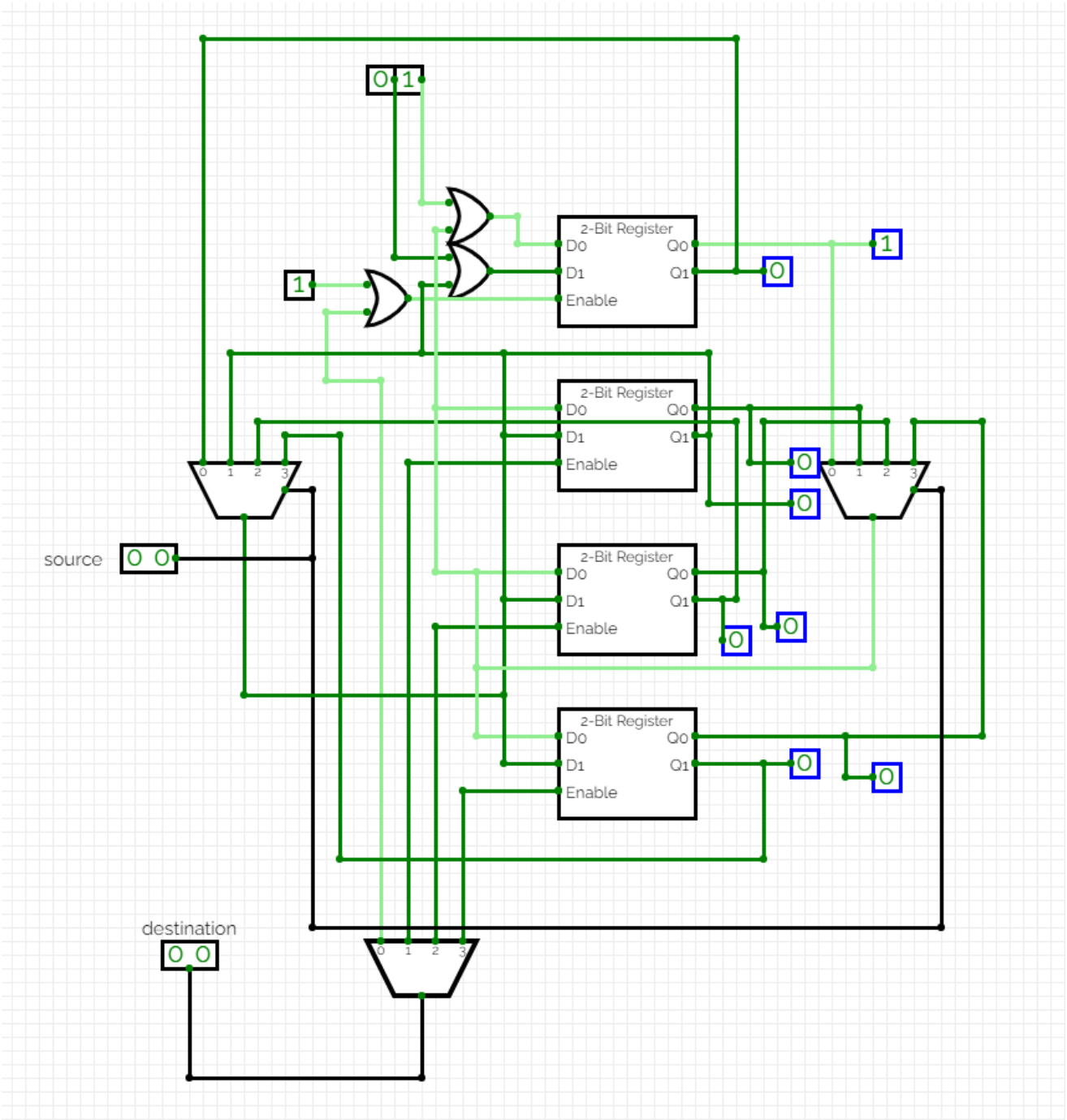
(a)2-bit Register

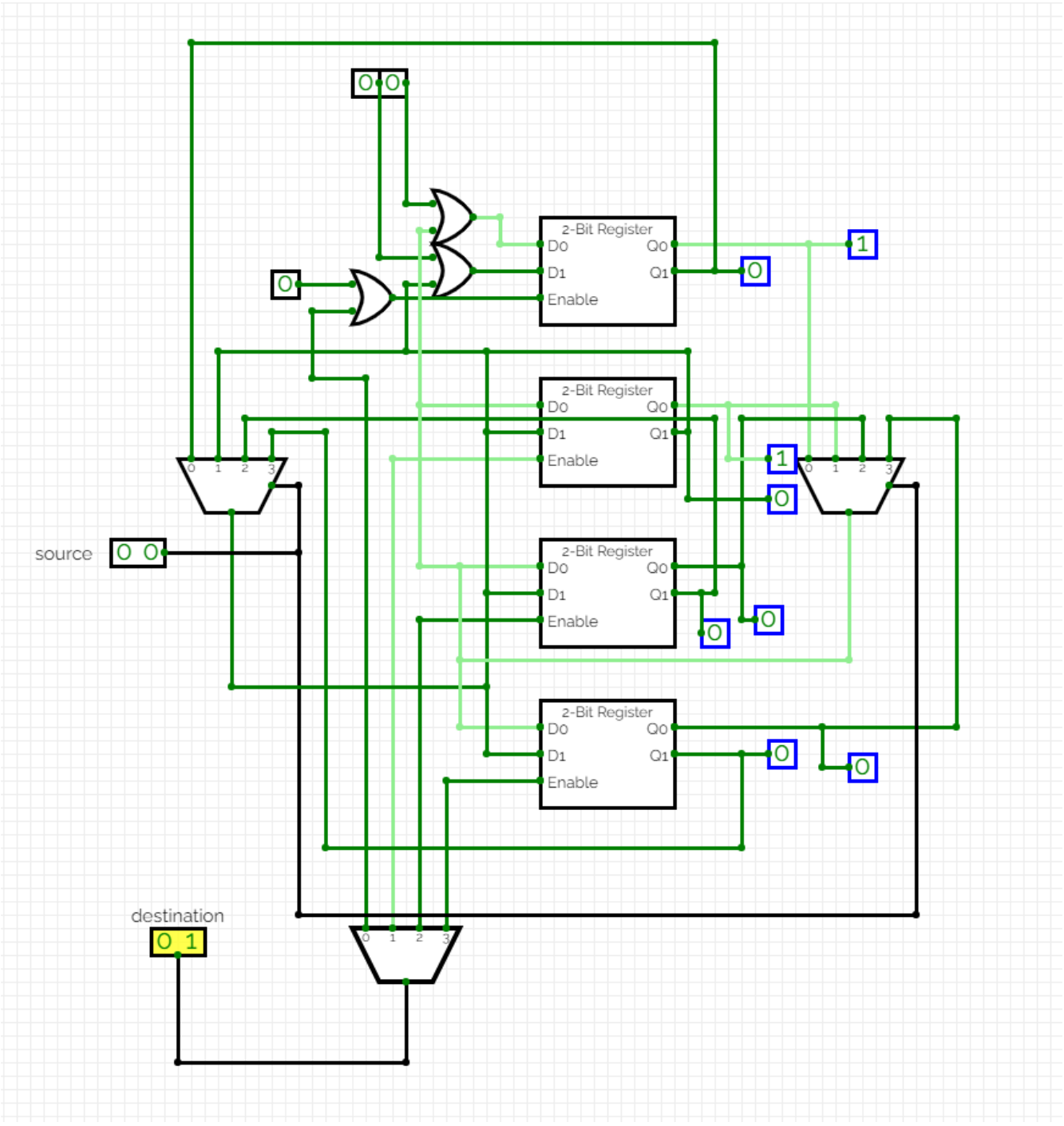


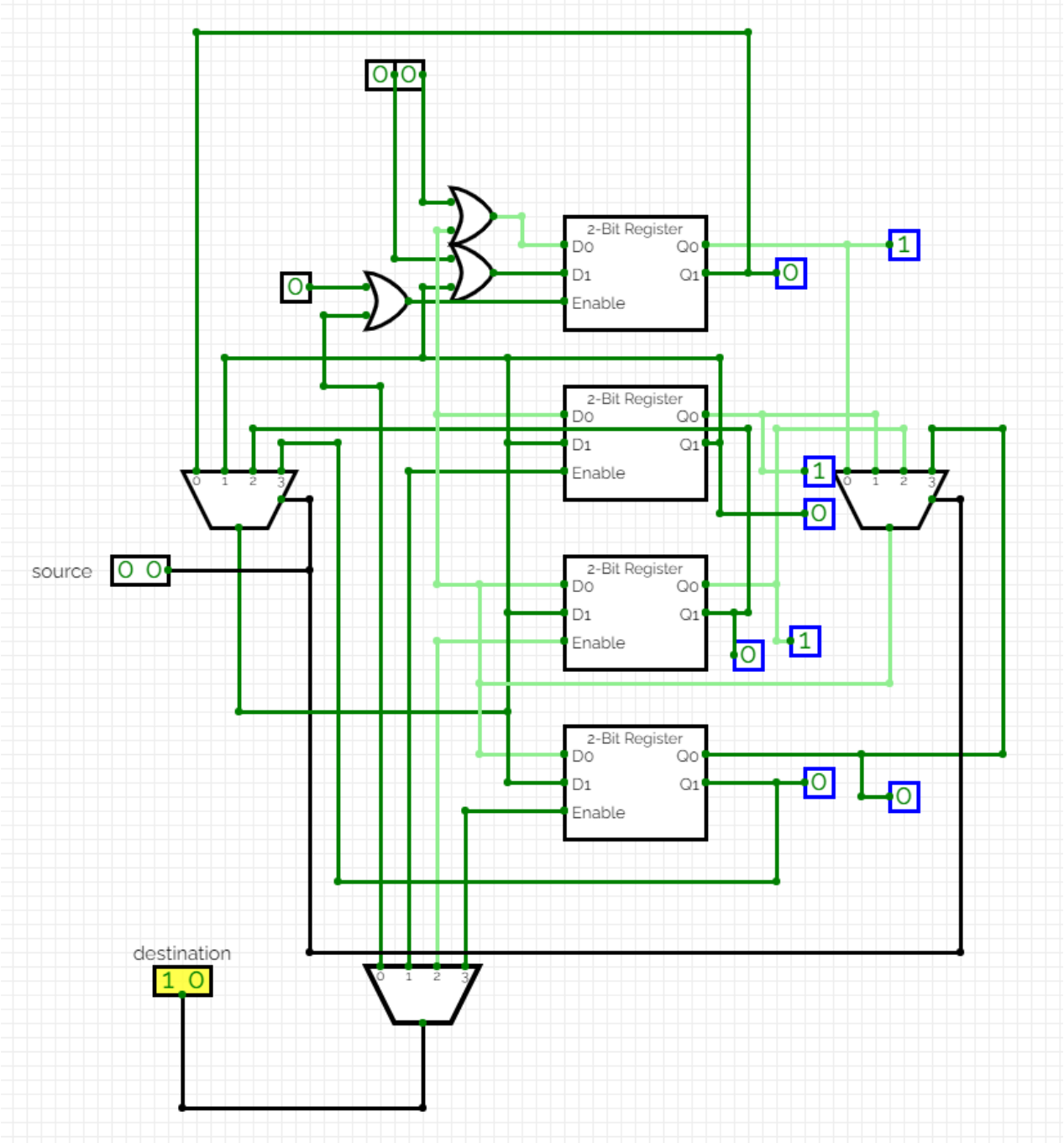

(b) Implementation of Data Transfer between four Registers

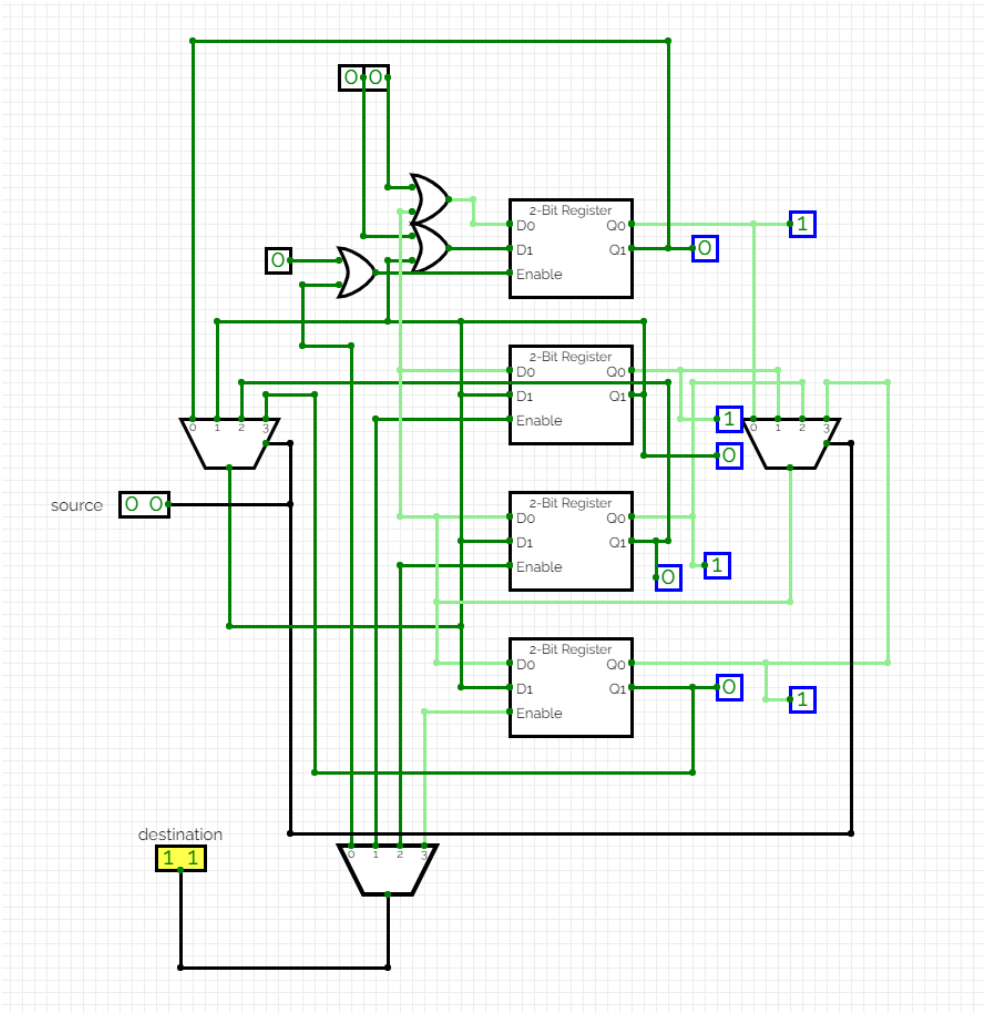


## 5.7 Simulation Results




0 0
0
2-Bit Register
D0
Q0
D1
Q1
Enable
1
0
2-Bit Register
D0
Q0
D1
Q1
Enable
1
0
0 1 2 3
source
0 0
2-Bit Register
D0
Q0
D1
Q1
Enable
0
0
2-Bit Register
D0
Q0
D1
Q1
Enable
0
0
destination
0 1
0 1 2 3


0 0
0
2-Bit Register
D0
Q0
D1
Q1
Enable
1
0
2-Bit Register
D0
Q0
D1
Q1
Enable
1
0
0 1 2 3
source
0 0
2-Bit Register
D0
Q0
D1
Q1
Enable
0
1
2-Bit Register
D0
Q0
D1
Q1
Enable
0
0
destination
1 0
0 1 2 3




## 5.8 Verification

The circuit developed for the implementation of Data Transfer between Registers works as expected.
i.e. it moves 2-bit data from one register from the source register to the destination register.

## 5.9 Conclusion

A circuit for the implementation of data movement between registers has been successfully designed and verified using simulation. Hence the assigned task is complete.

# Experiment No: 4

Swapping contents of registers using Microprogram

## 4.1 Objective

To become familiar with

(a) Register level data transfer through common bus

(b) Microprogrammed realization of the register level data transfer

## 4.2 Theoretical Basis

The µ-programming is a method of designing control unit in which control signal selection and sequencing information is stored in a ROM/RAM. The control signals to be activated at time t are specified by a µ-instruction.

A sequence of µ-instructions related to a task is the µ-program. In the present design, the µ-program is stored in memory/RAM.

## 4.3 Task and Design

***1. Setting registers and control switches (manually) to realize swapping of data:***

(a) Realize registers A, B and T with D latches/registers as in Figure 11a.
(b) Set the switches (control signals) C1, C2, and C3 to realize data swap.
(c) Identify additional control signals required to realize swapping through common bus.

## *2. Automated swapping of data:*

(i). Realize bi-directional bus (Figure (11b)).

(ii). Identify the control signals required to realize the following sequence of µ-instructions

a) Copy T←A

b) No operation

c) Copy A←B

d) No operation

e) Copy B←T

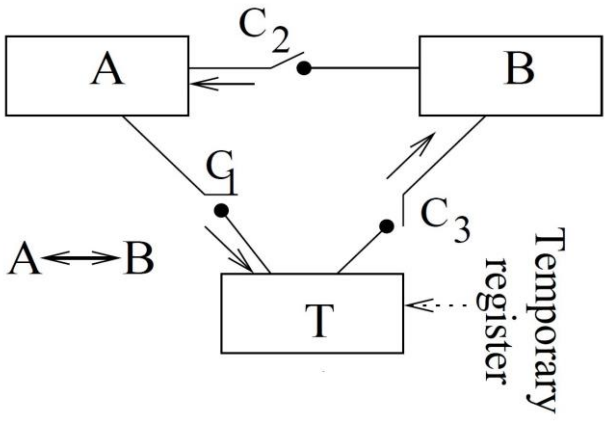
## *3. Setting up control memory:*

(i) Write µ-instructions in memory. Generate memory address using counter. Supply clock to generate next

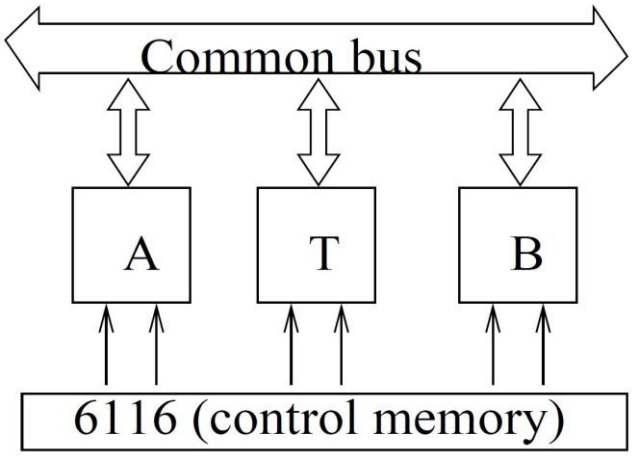
addresses (Figure (11c)).

(ii) Run µ-program.

(a) Data Swap




(b) Common Bus




(c) Control Memory

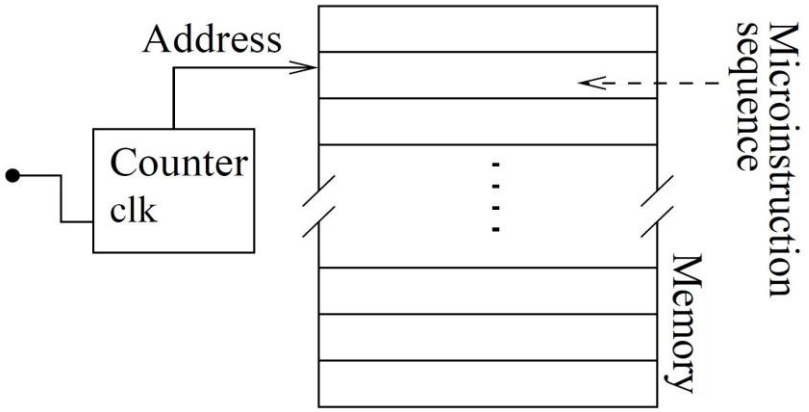


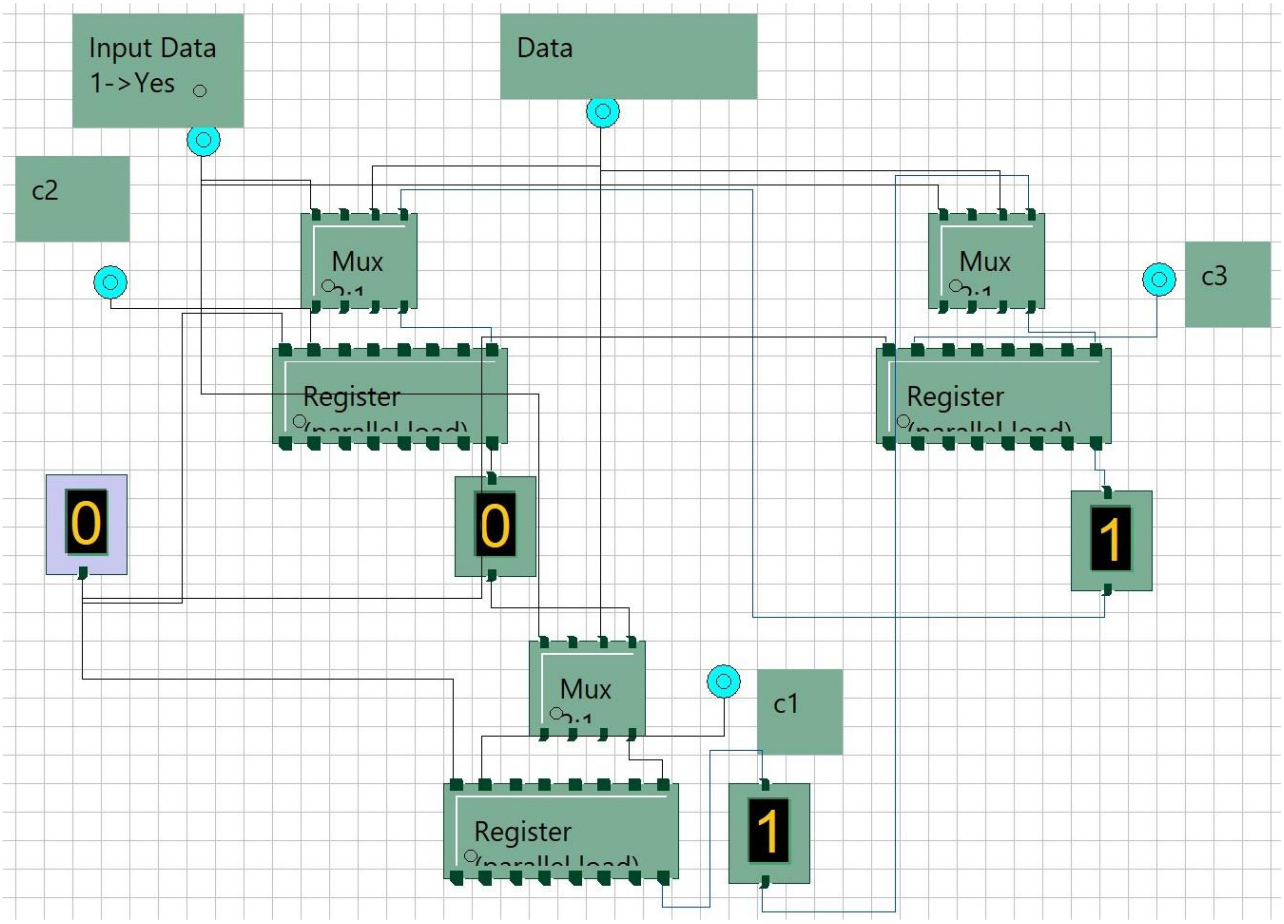

Figure 11: Data Swap between A and B

## 4.4 Major Components/Modules Used

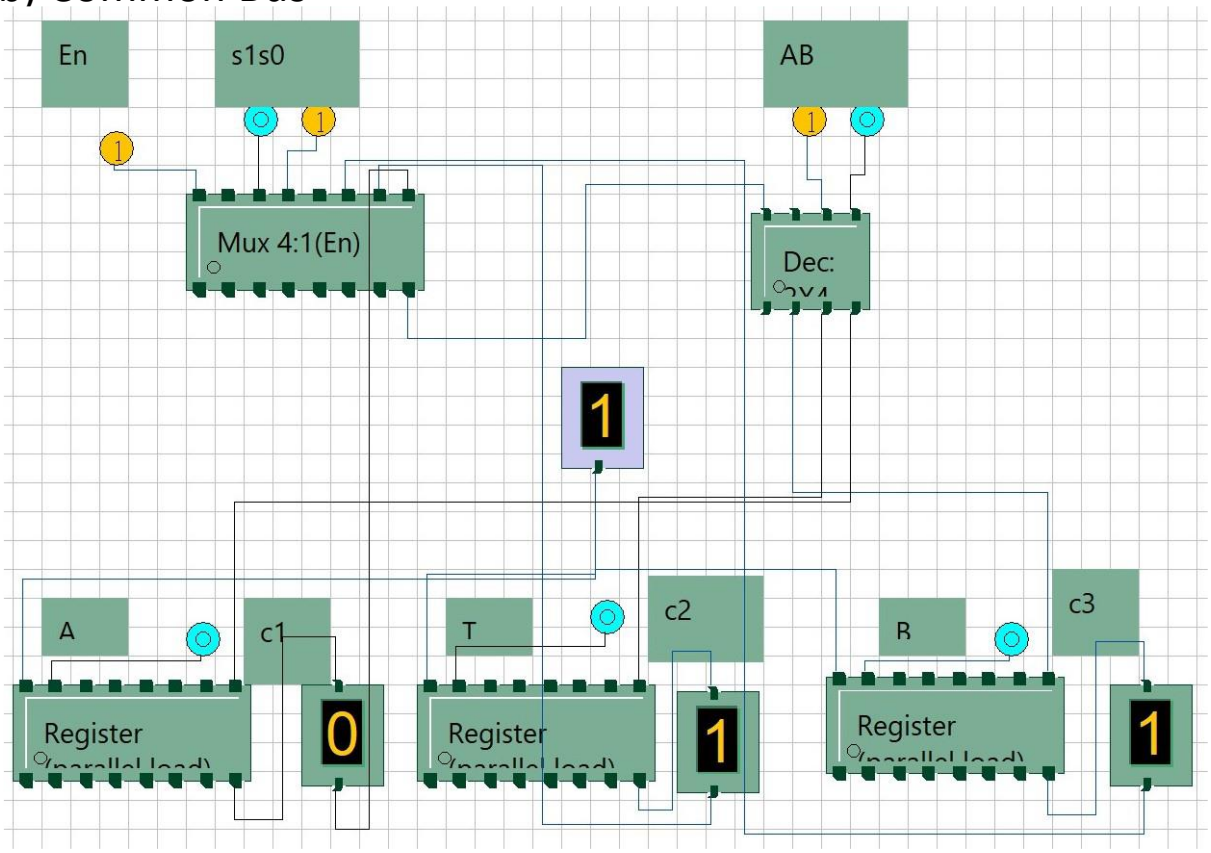
1. Registers
2. Multiplexers and Decoders
3. D Flip Flops
4. Memory Module

## 4.5 Implementation

(a) Data Swap



(b) Common Bus




(c) Control Memory

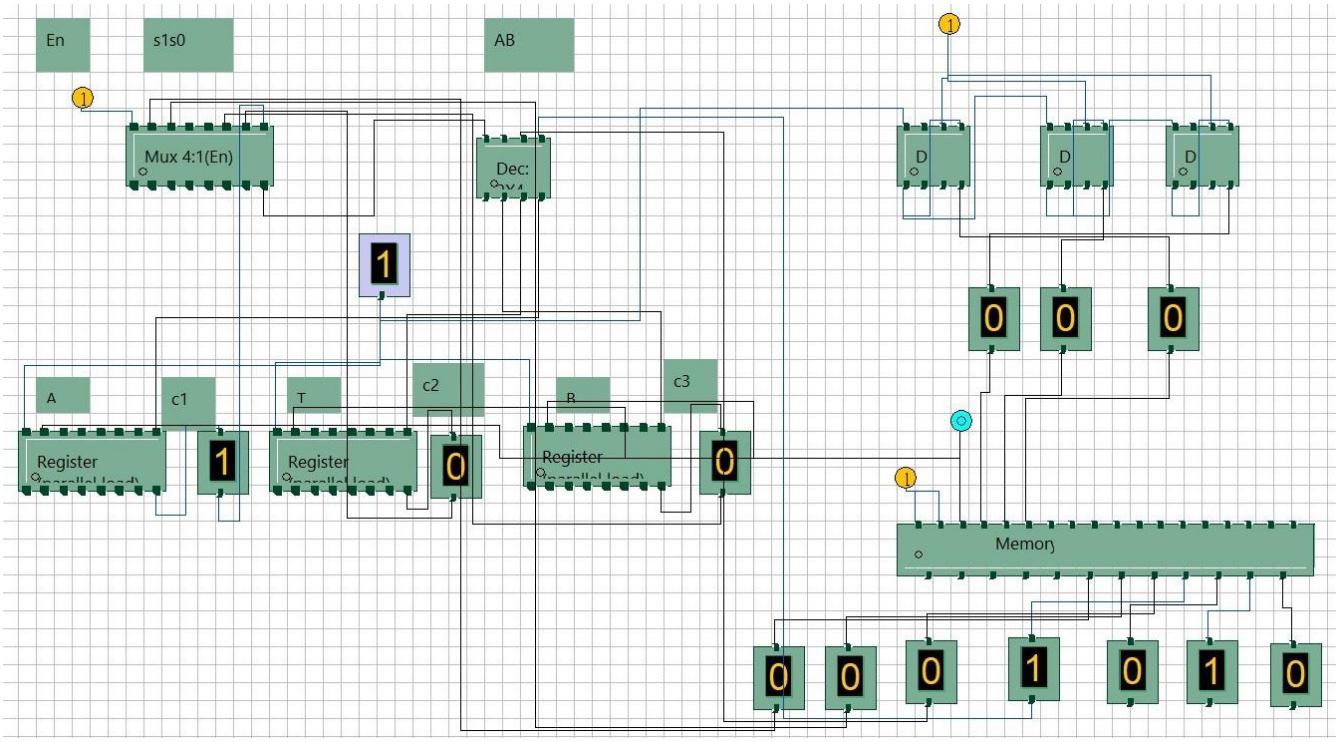



Figure 12: Implementation of Data Swap between A and B

### 4.6 Verification

1. Successfully swapped content of A and B manually, using T as a temporary register.

2. Successfully implemented common bus and implemented data swap on it.

3. Successfully used Memory to automate the swapping procedure.

## 4.7 Conclusion

We hence realized register level data transfer using common bus and control memory.

# Experiment No: 6

Design of sequence counter for processor control unit.

## 6.1 Objective

To build a mod-8 sequence counter for generation of control signals of a processor ALU.

## 6.2 Theoretical Basis

The instruction cycle of a CPU is shown in Figure 13. Assuming that each step is performed in an appropriately chosen clock period. A control unit for this CPU can be built around a modulo-8 sequence counter.

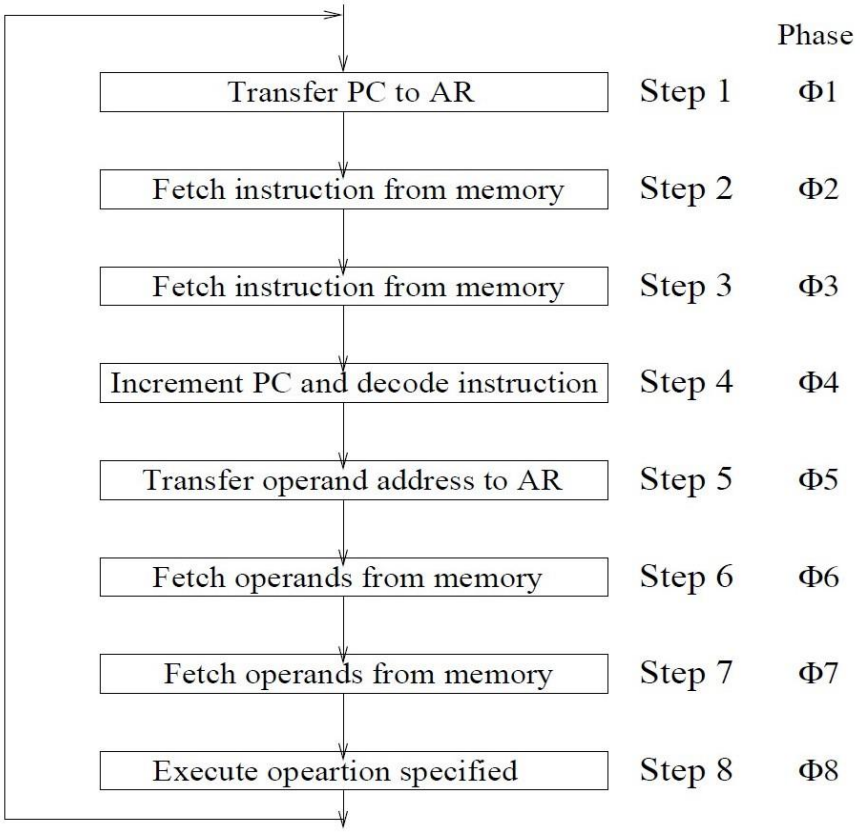



Figure 13: Instruction Cycle of CPU

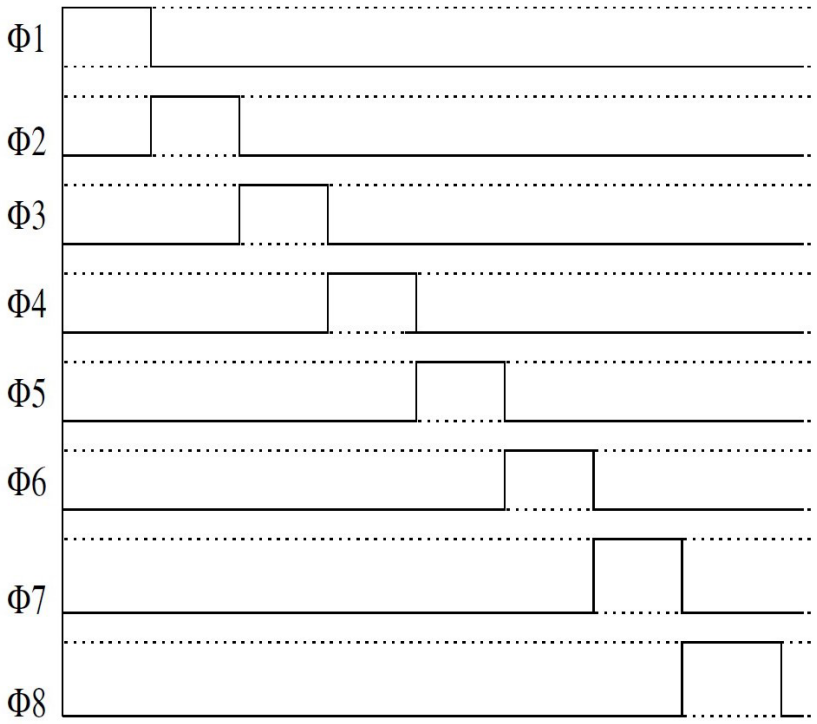
A modulo-8 counter counts input clock pulses from 000 to 111 and then sets to 000. The modulo-8 sequence counter can be designed with a modulo-8 counter and a 3-to-8 decoder. The decoder decodes the 3-bit binary code into a 1-bit binary code.

## 6.3 Task and Design

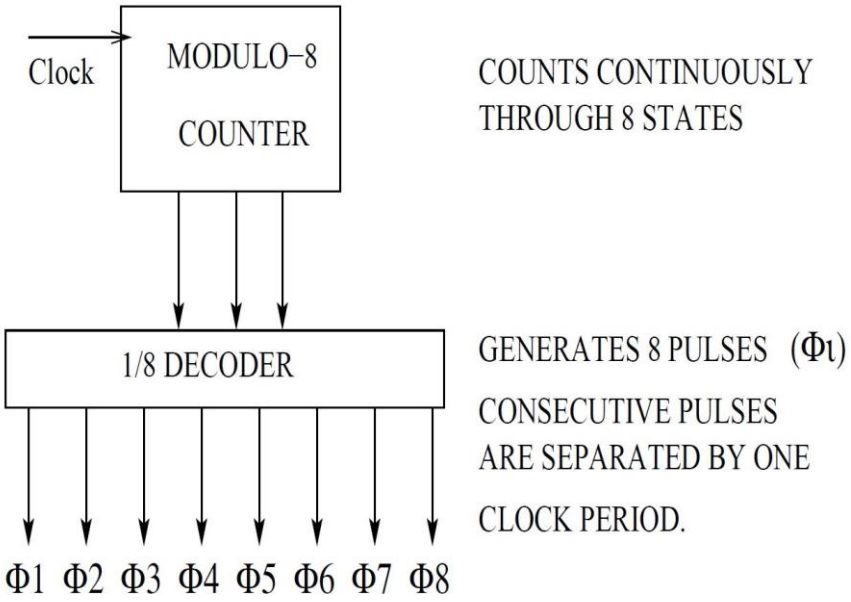
1. Design Modulo-8 sequence counter

2. Design a combinational logic that takes the phase signals (ϕ1, ϕ2, ϕ3, ϕ4, ϕ5, ϕ6, ϕ7, ϕ8) as the input and

generates a sequence of signals to control the primitive operations of the ALU.




(a) Clock Pulses of $\phi_i$




(b) Modulo-8 Sequence Counter

Figure 14: Design Of Modulo-8 Sequence Counter

## 6.4 Major Components/Modules Used

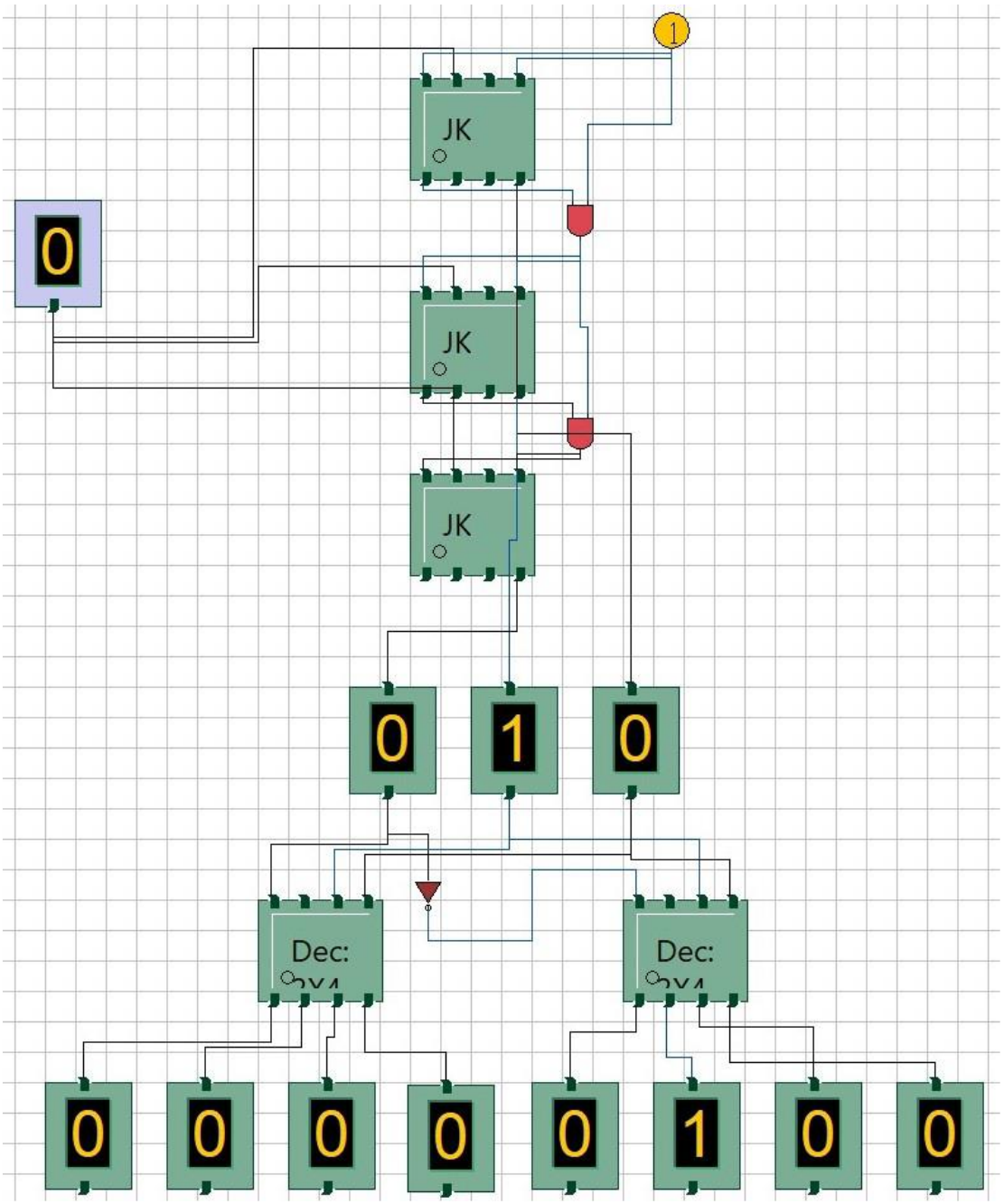
1. A modulo-8 counter. The counter was constructed using 3 JK flip flops and 2 AND gate as there was no counter available in the simulator.

2. One 3-to-8 decoder. The decoder was constructed using two 2-to-4 decoders as there were no 3-to-8 decoder available in the simulator.

3. ALU Chip

## 6.5 Implementation

(a) Implementation of Modulo-8 Sequence Counter



(b) ALU running from Modulo-8 Sequence Counter

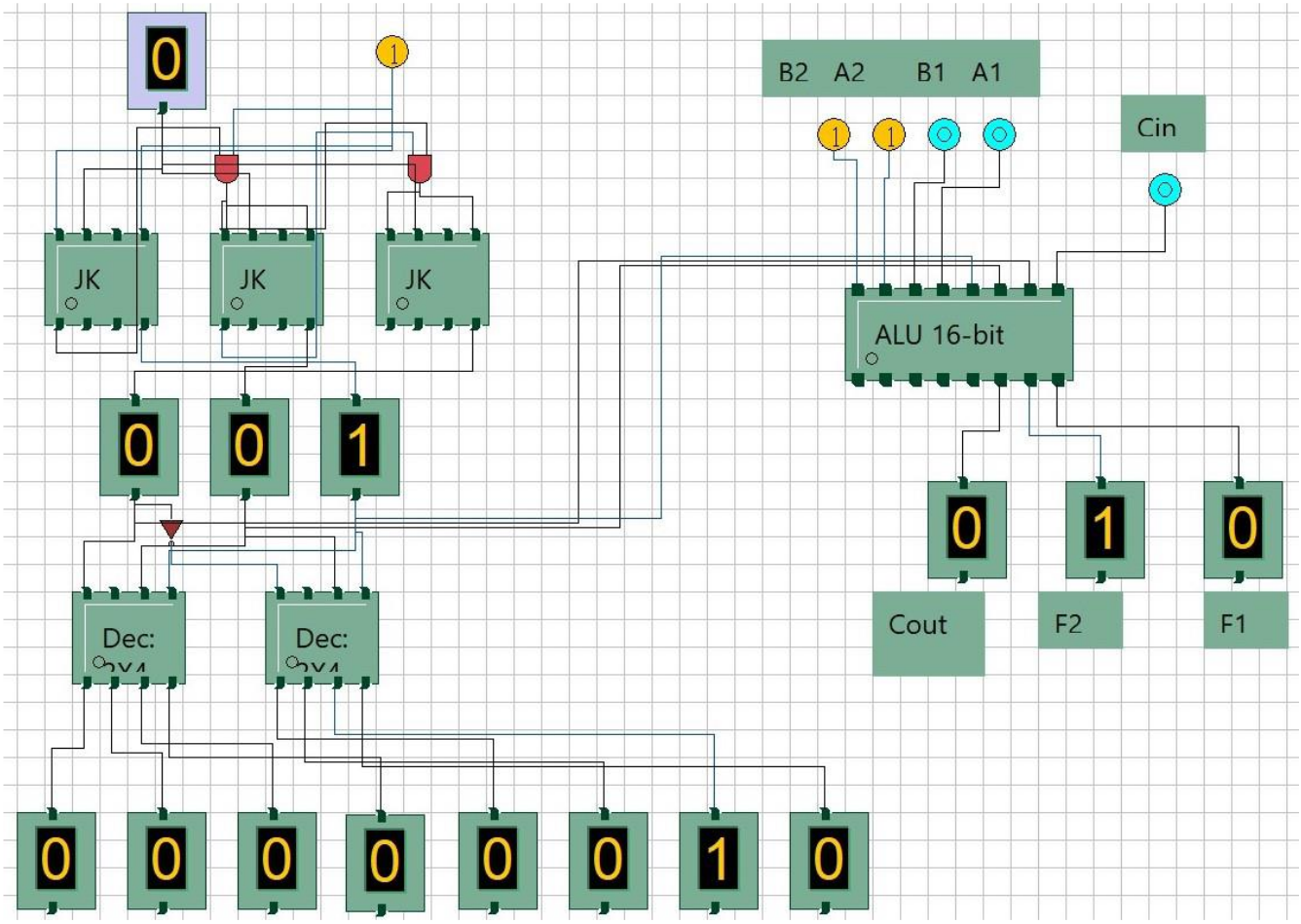



Figure 15: Implementation of Sequence Counter to Drive ALU

## 6.6 Verification

• The modulo-8 counter designed in the simulator counts from 000 to 111 properly when simulated i.e. when the clock is turned on. It then sets it back to 000. The 3-to-8 decoder decodes the 3-bit binary code (000-111) and produces a single control signal at a time.

• The Control signal then properly drives the ALU

## 6.7 Conclusion

A modulo-8 sequence counter with ALU is successfully designed and verified using the simulator. Hence, the given task is completed.