

L1 - PRIORITY ENCODER

A priority encoder is a circuit or algorithm that compresses multiple binary inputs into a smaller number of outputs. The output of a priority encoder is the binary representation of the index of the most significant activated line, starting from zero. They are often used to control interrupt requests by acting on the highest priority interrupt input.

Priority encoders can be easily connected in arrays to make larger encoders, such as one 16-to-4 encoder made from six 4-to-2 priority encoders - four 4-to-2 encoders having the signal source connected to their inputs, and the two remaining encoders take the output of the first four as input. The priority encoder is an improvement on a simple encoder circuit, in terms of handling all possible input configurations.

LSI Example: 74XX148 8:3 priority encoder (<https://www.ti.com/lit/ds/symlink/sn54ls148.pdf>)

Lab Exercise: implement a 4:2 priority encoder using:

1. The conditional assignment
2. The selected assignment
3. The if statement
4. The case statement

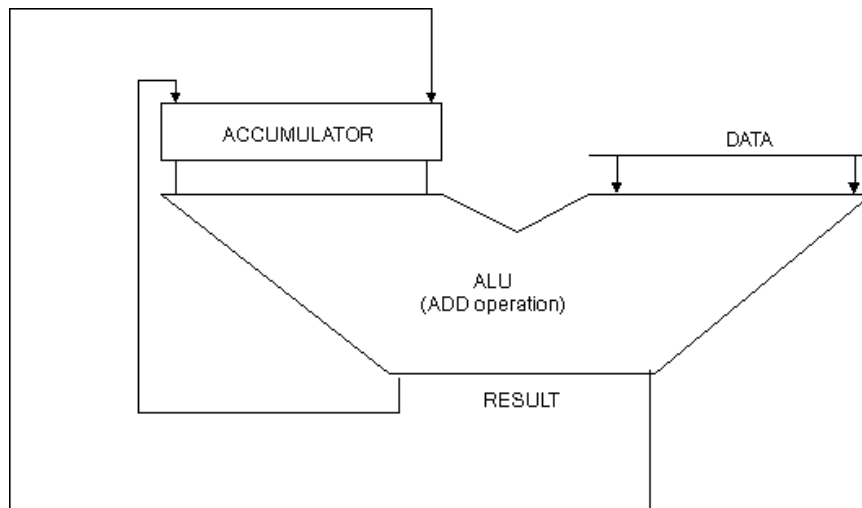
For sake of completeness, the encoder truth table is reported in the following:

| I ₃ | I ₂ | I ₁ | I ₀ | O ₁ | O ₀ | Active |
|----------------|----------------|----------------|----------------|----------------|----------------|--------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | x | 0 | 1 | 1 |
| 0 | 1 | x | x | 1 | 0 | 1 |
| 1 | x | x | x | 1 | 1 | 1 |

After creating the description of the 4:2 priority encoder, modify the code in order to add a {1,2,3,4}ns delay depending on the different statements adopted.

L2 - Arithmetic Logic Unit (ALU)

An arithmetic logic unit (ALU) is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. The inputs to an ALU are the data to be operated on, called operands, and a code indicating the operation to be performed; the ALU's output is the result of the performed operation. In simplest microprocessors, the ALU output is connected to an ACCUMULATOR register, permitting to reuse a previous result as an input operand.



LSI example: 74XX181 4bit ALU (<http://www.righto.com/2017/01/die-photos-and-reverse-engineering.html>)

Lab Exercise: implement a simple ALU

Implement an ALU capable of performing the following operations:

| OPCODE | OPERATION |
|--------|--|
| 0000 | Result_out = Data_in |
| 0001 | Result_out = Accumulator_in |
| 0010 | Result_out = Accumulator_in + Data_in |
| 0011 | Result_out = Accumulator_in - Data_in |
| 0100 | Result_out = Accumulator_in AND Data_in |
| 0101 | Result_out = Accumulator_in OR Data_in |
| 0110 | Result_out = Accumulator_in XOR Data_in |
| 0111 | Result_out = NOT Accumulator_in |
| 1000 | Result_out = NOT Data_in |
| 1001 | Result_out = "0000 0000" (Zero) |
| 1010 | Result_out = Accumulator_in << 1 [use shift_left()] |
| 1011 | Result_out = Accumulator_in >> 1 [use shift_right()] |
| 1100 | Result_out = Accumulator_in NAND Data_in |
| 1101 | Result_out = Accumulator_in NOR Data_in |
| 1110 | Result_out = Accumulator_in XNOR Data_in |
| 1111 | Result_out = Accumulator_in + 1 |

L3 - CARRY LOOK AHEAD ADDER

Carry Look Ahead Adder is fastest adder compared to Ripple carry Adder.

The ripple carry adder indeed produces carry propagation delay while performing other arithmetic operations like multiplication and divisions as it uses several additions or subtraction steps. This is a major problem for the adder and hence improving the speed of addition will improve the speed of all other arithmetic operations. Hence reducing the carry propagation delay of adders is of great importance. There are different logic design approaches that have been employed to overcome the carry propagation problem. One widely used approach is to employ a carry look-ahead which solves this problem by calculating the carry signals in advance, based on the input signals. This type of adder circuit is called a carry look-ahead adder.

Here a carry signal will be generated in two cases:

- Input bits A and B are 1
- When one of the two bits is 1 and the carry-in is 1.

In ripple carry adders, for each adder block, the two bits that are to be added are available instantly. However, each adder block waits for the carry to arrive from its previous block. So, it is not possible to generate the sum and carry of any block until the input carry is known.

For the Purpose of carry Propagation, Carry look Ahead Adder construct Partial Full Adder, Propagation (P) and generation (G) Carry block. It avoids Carry propagation through each adder.

For simplicity, let:

- $G_i = A_i B_i$ where G is called carry generator (i.e., a “locally” generated carry out)
- $P_i = A_i \oplus B_i$ where P is called carry propagator (i.e., a carry out must be propagated if carry in = ‘1’)

Then, re-writing the above equations, we have-

$$C_1 = C_0 P_0 + G_0 \quad (1)$$

$$C_2 = C_1 P_1 + G_1 \quad (2)$$

$$C_3 = C_2 P_2 + G_2 \quad (3)$$

$$C_4 = C_3 P_3 + G_3 \quad (4)$$

Clearly, C_1 , C_2 and C_3 are intermediate carry bits; substituting (1) in (2), we get C_2 in terms of C_0 . Then, substituting (2) in (3), we get C_3 in terms of C_0 and so on. Finally, we have the following equations:

$$C_1 = C_0 P_0 + G_0 \quad (5)$$

$$C_2 = C_0 P_0 P_1 + G_0 P_1 + G_1 \quad (6)$$

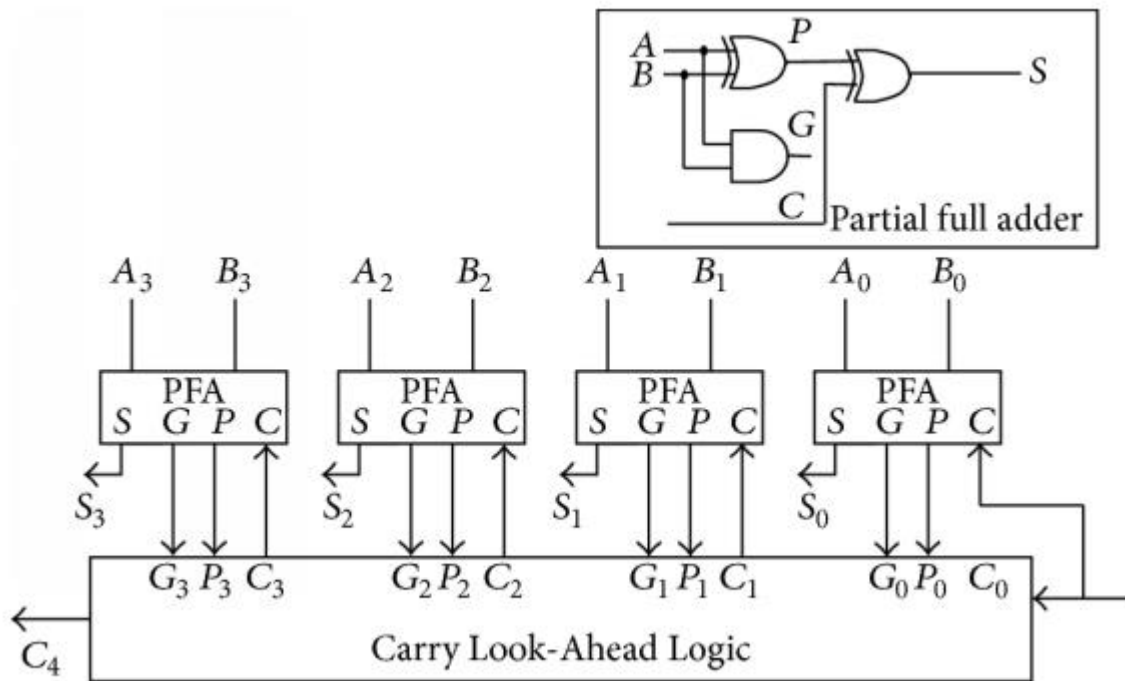
$$C_3 = C_0 P_0 P_1 P_2 + G_0 P_1 P_2 + G_1 P_2 + G_2 \quad (7)$$

$$C_4 = C_0 P_0 P_1 P_2 P_3 + G_0 P_1 P_2 P_3 + G_1 P_2 P_3 + G_2 P_3 + G_3 \quad (8)$$

These equations show that the carry-in of any stage full adder depends only on:

- Bits being added in the previous stages
- Carry bit which was provided in the beginning

In order to implement Carry Look Ahead Adder, first implement Partial Full Adder and then Carry Look-Ahead logic using Propagation and Generation equations. Partial Full Adder consists of inputs (A, B, C) and Outputs (S, P, G) where P is Propagate Output and G is Generate output.



The disadvantage of the carry-lookahead adder is that the carry logic is getting quite complicated for more than 4 bits. For that reason, carry-look-ahead adders are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4 bits. Following Figure shows the block diagram for a 16-bit CLA adder. The circuit makes use of the same CLA Logic block as the one used in the 4-bit adder. Notice that each 4-bit adder provides a group Propagate PG and Generate GG Signal, which is used by the CLA Logic block. The group Propagate PG of a 4-bit adder will have the following expressions:

$$PG = P_3.P_2.P_1.P_0 \quad (9)$$

$$GG = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 \quad (10)$$

