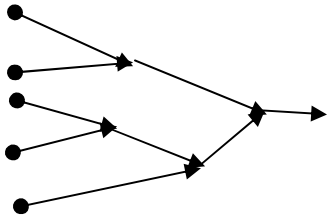
Lecture 1

מעגל צירופי – הגדרה

- אבני הבניין של מעגל צירופי הם שערי and, or, not. הכניסות של השערים הנ"ל סימטריות והם מהווים מערכת אוניברסלית.
- מעגל צירופי בנוי מעץ מכוון עם שורש. הקשתות מכוונות מהעלים לשורש. אין בו מעגלים.



- בכל צומת של העץ ממוקם שער – כך נקבעת הפונקציונליות של המעגל. המעגל מבצע חישוב של הפונקציות הבוליאניות שהשערים ממשים ופלט את ערך היציאה המחושב
- השהיית המעגל – המסלול הארוך ביותר מעלה לשורש.
- מחיר המעגל – מספר הצמתים בעץ. כלומר, מספר השערים בעץ (במעגל).

- הערה: אנו מניחים שלכל השערים אותה השהייה ואותו מחיר. אפשר היה להוסיף משקל לכל שער. במקרה כזה המחיר היה המשקל הכולל וכו'.

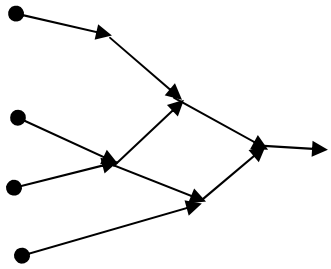
- הערה: העלים יכולים להיות מוזנים על-ידי אותם קלטים.

- דרגת הכניסה – Fan in – מספר הכניסות לשער. בחרנו שיהיה 2 לכל היותר.

(כדוגמא לכל הנ"ל נראה שער and מרובה כניסות)

הגדרה חדשה (הרחבת ההגדרה הקודמת) :

- נשתמש בדג במקום בעץ – DAG (Directed Acyclic Graph)



- דג אינו עץ. מה שנקרא עלה בעץ נקרא כאן מקור.
- מה שנקרא שורש בעץ נקרא כאן בור.

- לדג יכולים להיות מספר בורות.

(נראה דוגמאות במהלך ההרצאה)

1.8 **Gates**

The basic building blocks which are used to implement a logic function are called gates. Any equation can be considered as a system having inputs (the variables) and outputs (the result of applying the functions on the variables). The equation $Y = f(A,B,C)$ describes a system having 3 inputs A,B,C and a single output Y.

1.8.1 **An AND gate**

Its equation is $Y = A \cdot B$. We draw it as in Figure 1.4. Its truth table is given below and is identical to the AND operation.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

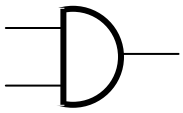


Figure 1.4 – An AND gate

1.8.2 **An OR gate**

Its equation is $Y = A + B$. We draw it as in Figure 1.5. Its truth table is given below and is identical to the OR operation.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

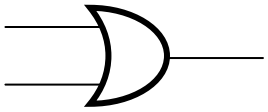


Figure 1.5 –An OR gate

1.8.3 **A NOT gate (usually called an INVERTER)**

Its equation is $Y = \overline{A}$. We draw it as in Figure 1.6. Its truth table is given below and is identical to the NOT operation.

A	Y
0	1
1	0

Figure 1.6 – An Inverter (a NOT gate)

1.8.4 **A NAND gate**

Its equation is $Y = \overline{A \cdot B}$. We draw it as in Figure 1.7. Its truth table is given below.

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

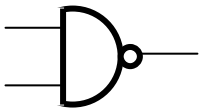


Figure 1.7– A NAND gate

1.8.5 **A NOR gate**

Its equation is $Y = \overline{A + B}$. We draw it as in Figure 1.8. Its truth table is given below.

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

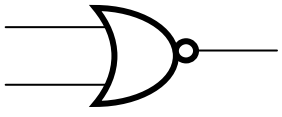


Figure 1.8 – A NOR gate

1.8.6 **A XOR gate**

Its equation is $Y = \overline{A} \cdot B + A \cdot \overline{B} = A \oplus B$. We draw it as in Figure 1.9. Its truth table is given below.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

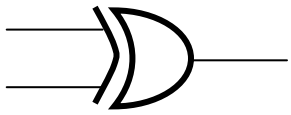


Figure 1.9 – A XOR gate

Figure 1.10 is a simple example of implementing a Boolean function using gates. We here build a XOR gate, using AND, OR and NOT gates (Note: a dot represents a connection of wires):

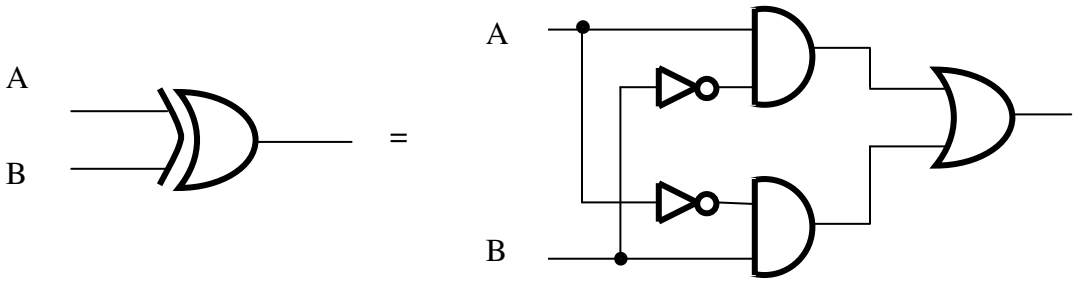


Figure 1.10– Building a XOR gate using AND, OR & NOT gates

1.9 **A Universal system**

Note that since the only operators we defined in Boolean algebra are the AND, OR and NOT operators, it is clear that having these three kind of gates in our hands, enables us to build ***any*** desired function. Therefore, we call the set of AND, OR and NOT gates, a universal system.

A NAND gate is itself a universal system and so it is called a Universal Gate. In order to show that we can build ***any*** desired function using only NAND gates, we will show that we can implement all of the 3 operators AND, OR and NOT using only NAND gates. Let us start with the NOT operation. We want to implement $Y = \overline{A}$ using a NAND gate whose function is $Y = \overline{A \cdot B}$. If we choose B to be 1 or connect the input A to the other input, we create an inverter since $Y = \overline{A \cdot 1} = \overline{A}$ and also $Y = \overline{A \cdot A} = \overline{A}$:

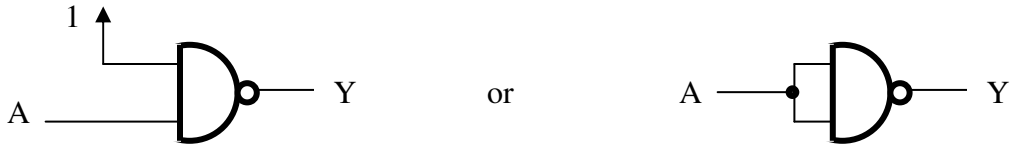


Figure 1.11 – An inverter made of a NAND gate

Since a NAND gate is just an AND gate which is followed by an inverter, all we need in order to convert it "back" to a regular AND gate, is to add one more inverter:

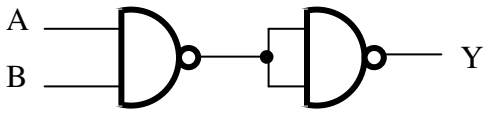


Figure 1.12 – An AND gate made of NAND gates

Building the OR is a little bit more difficult. We need to use DeMorgan's laws:
$Y = A + B = \overline{\overline{A + B}} = \overline{\overline{A} \cdot \overline{B}}$. Now, it is easy to implement the OR using 3 NAND gates, two as inverters and the third one to perform the NAND operation on the first two's outputs.

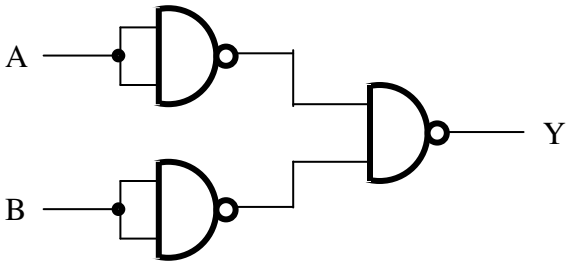


Figure 1.13 – An OR gate made out of NAND gates

Let us now try to implement a XOR gate using only NAND gates. The easiest way is to replace any inverter in Figure 1.10 with the inverter of Figure 1.11, and any AND gate with the AND gate of Figure 1.12, and finally, the OR gate with the OR of Figure 1.13:

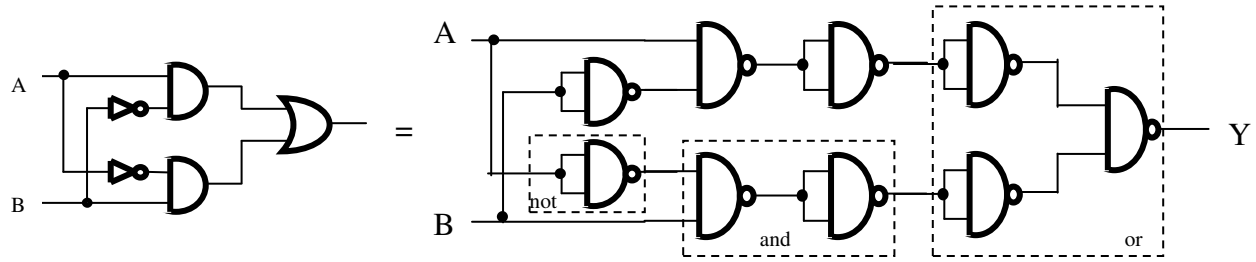


Figure 1.14 – Building a XOR gate using only NAND gates

We can reduce the gate count, if we delete the two redundant pairs of inverters. Those are redundant since $\overline{(\overline{X})} = X$. Eventually we end with:

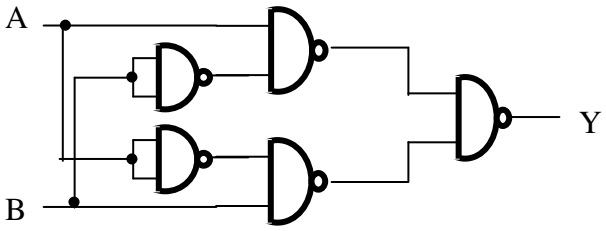


Figure 1.15 – A XOR gate made of NAND gates

1.10 Timing issues of gates

Let us first define some terms.
A signal is a continuous function of the time t.
A logic level "0" is a predefined voltage range that is recognized by a gate as a "0" level. In the well-known TTL 74xx logic family, a "0" level was defined as 0.0v to 0.2v.
A logic level "1" is another predefined voltage range that is recognized by a gate as a "1" level. In the 74xx logic family, a "1" level was defined as 2.0v to 5.0v.
The signal has a logic level when the value of the signal in the range of logic level "0" or in the range of logic level "1". A signal is called stable at a time interval if it stays in the same logic level along the entire time interval.

Let us explore the behavior of a simple gate. We input the signal A(t) to an inverter and receive the signal Y(t) at the inverter's output.

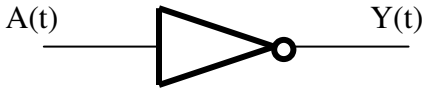


Figure 1.16 – Naming the signals of a NOT gate

The input signal A(t) starts at "1" so Y(t) is "0". At a certain point at time, i.e., at t_0, we change the input signal to be "0". The gate does not respond immediately. Its response is depicted in Figure 1.17 below. We see that it takes some time till the output signal changes. The time period in which the output signal still stays in the initial logic level, i.e., the time in which the gate "does not response" to the input change, is called the ***contamination delay*** and is denoted by t_{cd}. The time required for the output to reach its "final", i.e., stable level, is called the ***propagation delay*** and is denoted by t_{pd}. These two time intervals are described in Figure 1.17 below for the rising and falling of the signals A(t) and Y(t) where A(t) is changing at t_0 and t_1.

When we implement a logical function using gates, we must consider the timing. When we want to know how soon will the output of a logical system be valid, i.e., in its stable logical level, we need to consider the worst case of all the gates. If this is a combinational system, we should take into account the sum of the delays of the maximal path (longest or slowest) between the input and the output signals. So, for our purposes, we can draw the signals as having valid logical values after the maximal t_{pd} of the gates involved.

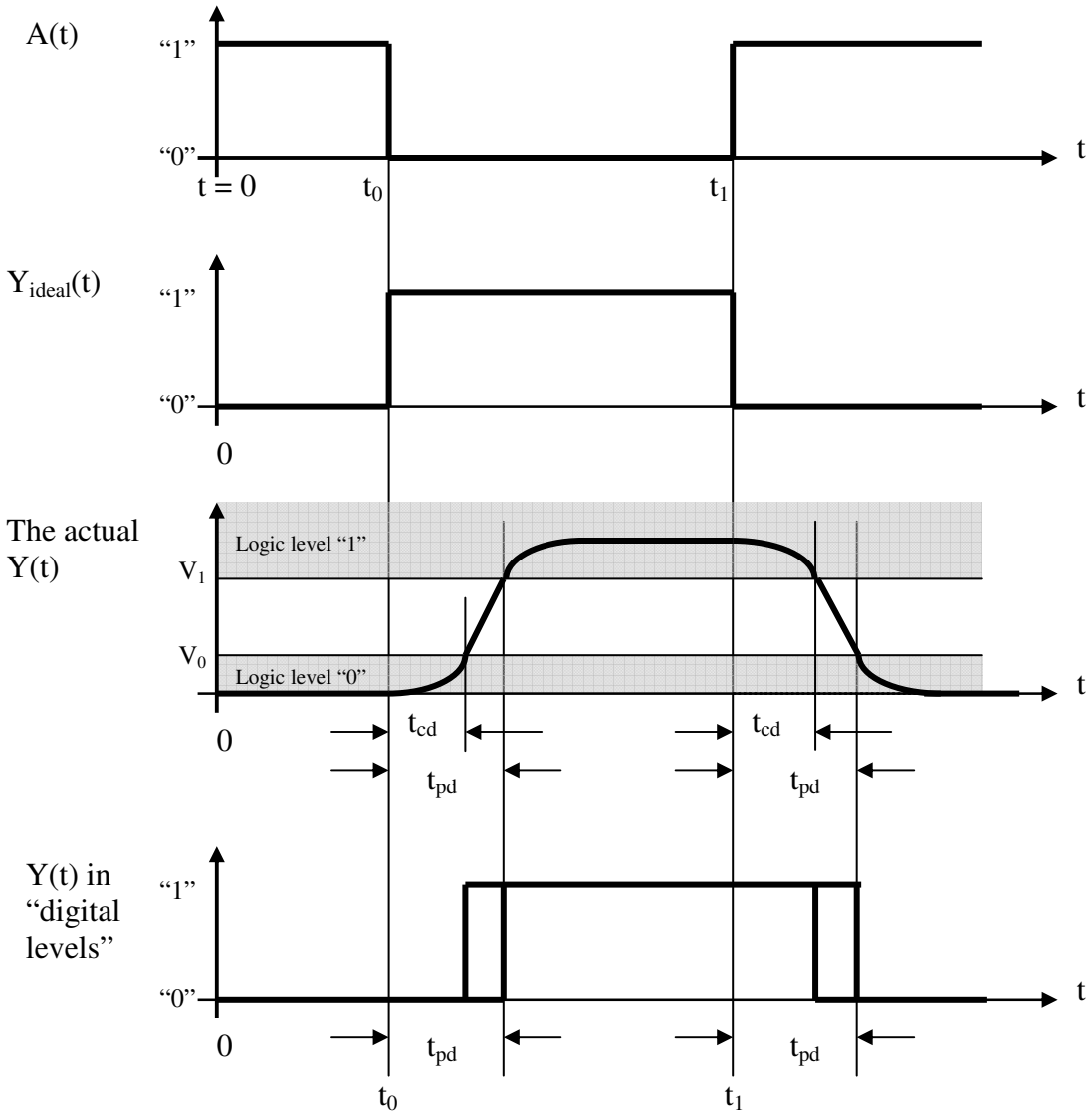


Figure 1.17 – The timing behavior of a NOT gate

In Figure 1.17, we see the input signal A(t) at the top. The response of an ideal gate, i.e., without any delay, is described as $Y_{ideal}(t)$. The actual signal Y(t) at the output appears 2nd from the bottom. For our analysis of digital circuits we can use the "digital levels" picture shown at the bottom of Figure 1.17.

1.11 **Multiple inputs gates**

Now we know that gates have delays. We should take that into account when we build systems that are more complex then a single gate. In computer science, the analysis of an algorithm usually deals with its complexity or performance, expressed as the number of operations required,

and its cost in the memory units required. In analysis of hardware systems, we have similar measures. The performance is measured by the maximum delay of the system and the cost by the number of required gates.

Let us now build an n inputs AND gate using two inputs AND gates only. The simplest way is based on induction. When we want to build a three input AND gate using two inputs AND gate we'll use the rules saying that $Y = A \cdot B \cdot C = (A \cdot B) \cdot C$, i.e., we'll use one gate to produce $A \cdot B$ and another gate to AND the result with C. Using induction, we can quite easily build an n inputs AND gate, adding a single input at a time. This is depicted in Figure 1.18:

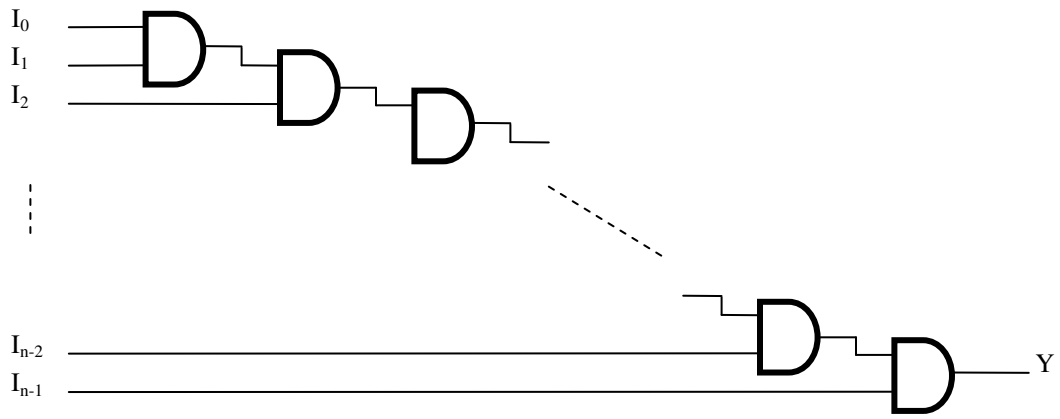


Figure 1.18 – Building an n inputs AND gate using 2 inputs AND gates

We will define such a structure "recursively" by describing an n inputs gate built of an (n-1) input gate and a simple 2 inputs gate:

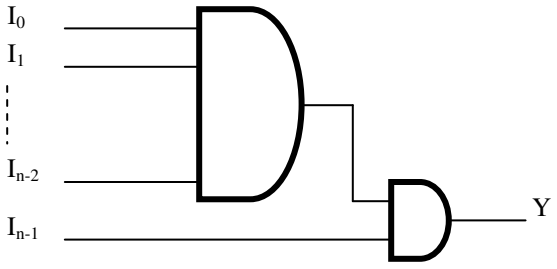


Figure 1.19 – Building an n inputs AND recursively

In this simple way, the recursive equations describing the cost and the delay of the multiple input gate are C(n)=C(n-1)+1 and D(n)=D(n-1)+T respectively. In this case, it is easy to see that the cost of an n inputs AND gate is C(n)=n-1, i.e., we need n-1 gates, 2 input AND gates, in order to build an n inputs AND gate. The delay is given by D(n)=(n-1)T where T is the delay of a single 2 inputs AND gate. The reason for this dependency of the delay on the number of inputs is the chaining of the gates. Because of this structure, a change in the I_0 should "propagate" through n-1 gated until it "reaches", i.e., influences, the output Y. This seems a little exaggerated. There must

be a better way. That way is to use a binary tree structure. The depth of that tree will determine the maximal delay. This can be seen in Figure 1.20 below.

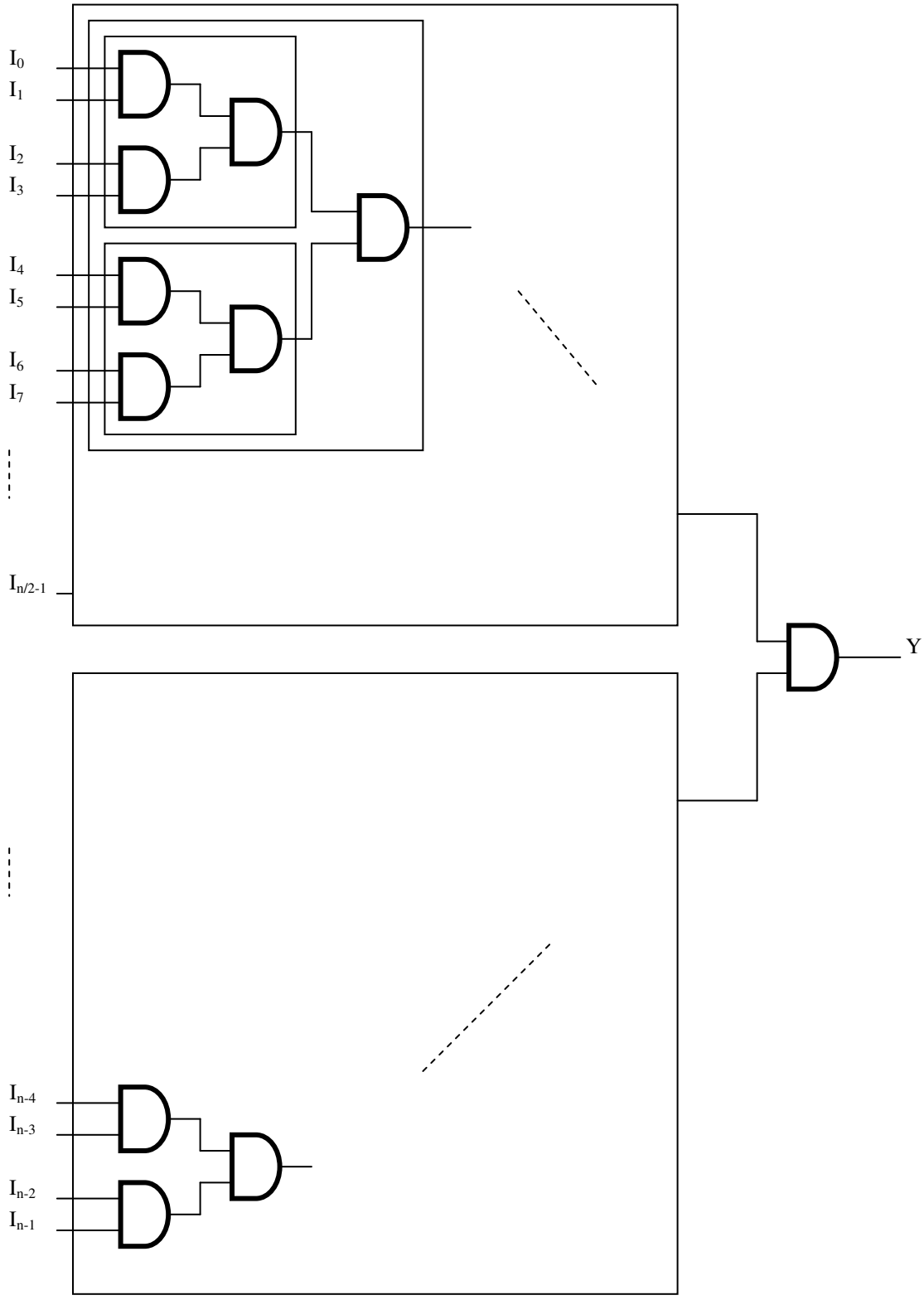


Figure 1.20 – Building an n inputs AND gate using a tree of inputs AND gates

We can define such a structure "recursively" by describing an n inputs gate built of two n/2 input gates:

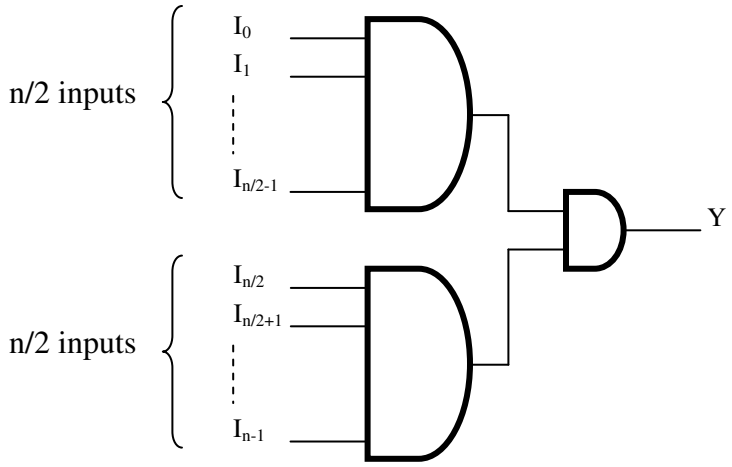


Figure 1.21 –A recursive building of an n inputs AND gate

The cost of such an n inputs gate stays C(n)=n-1. This is so since it really does not matter how we add the inputs, since every new input forces us to add a single gate. The recursive equation describing the cost is C(n)=2C(n/2)+1 having n/2 but also a factor of 2 certifies a linear cost.
It is quite clear from Figure 1.19 that the delay follows the recursive equations $D(n) = D(n/2) + T$. This immediately means that the delay is logarithmic, i.e., $D(n) = T \cdot \lg_2 n$. This is so since we can write:
$D(n) = D(n/2) + T = D(n/4) + T + T = D(n/8) + T + T + T$, etc., so we see that we have to sum $\lg_2 n$ times the delay T.

When n is not an exact power of 2, there are several optional trees, all with depth of $\lceil \lg_2 n \rceil$, to arrange the gates. The delay in such case is given by $D(n) = T \cdot \lceil \lg_2 n \rceil$.

We use basic gates of 2 inputs although in practice gates with more inputs are available.
Note that if we had a basic gate of 3 inputs we would get $D(n) = T \cdot \lceil \lg_3 n \rceil$.

1.12 **Decoders**

It is time now to get to our first useful system. We are going to build a Decoder. A decoder has n inputs and 2^n outputs. Only one of its outputs is "1" at a given time. The combination of the n input lines, each can be "0" or "1", determines which of the outputs is "on", i.e., "1". As a matter of fact, the combination at the input represents a binary number in which the rightmost digit has a value (or weight) of 1, the next digit has a value of 2, the next has a value of 4 and the next of 8 and so on. Thus the combination 0101 has a value of $0\cdot 8+1\cdot 4+0\cdot 2+1\cdot 1=5$ and the combination 0111 has a value of 7 since $0\cdot 8+1\cdot 4+1\cdot 2+1\cdot 1=7$. We would like to build a decoder having only two inputs, I_0 and I_1, forming together a two bit number $[I_1, I_0]$ which can have the values 0,1,2 or 3. And so, the decoder has 4 outputs Y_0, Y_1, Y_2, and Y_3. We would like the i-th output to be "1" when the input has the combination that represent the number i.

How do we do that?

We use a truth table to describe the decoder and then find the equations of the outputs from that table;

$[I_1\ I_0]$	Y_0	Y_1	Y_2	Y_3
00	1	0	0	0
01	0	1	0	0
10	0	0	1	0
11	0	0	0	1

We immediately see that the equations of the outputs are given by:

$$Y_0 = \overline{I_1}\cdot\overline{I_0}$$
$$Y_1 = \overline{I_1}\cdot I_0$$
$$Y_2 = I_1\cdot\overline{I_0}$$
$$Y_3 = I_1\cdot I_0$$

So the decoder can be built as described in Figure 1.22 below:

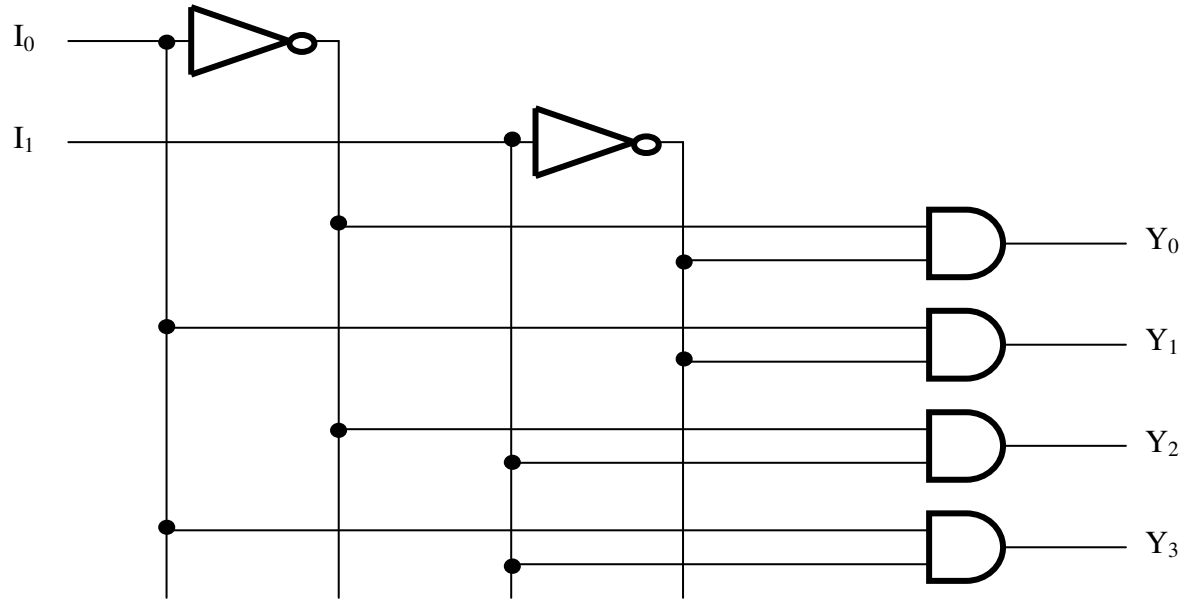


Figure 1.22 – A 2 inputs → 4 outputs decoder

Note that the procedure that we'll always use is: First, define the required device. Then, build its truth table. Then, find its equations from the truth table. Then implement it with gates.

We would now like to recursively build an n inputs decoder using (n-1) inputs decoders. When we design a VLSI chip, we want to get rid of all redundant parts. Another look at Figure 1.23 reveals that the two decoders produce similar outputs. Therefore, a better design is to use a single decoder and duplicate its output as shown in Figure 1.26.

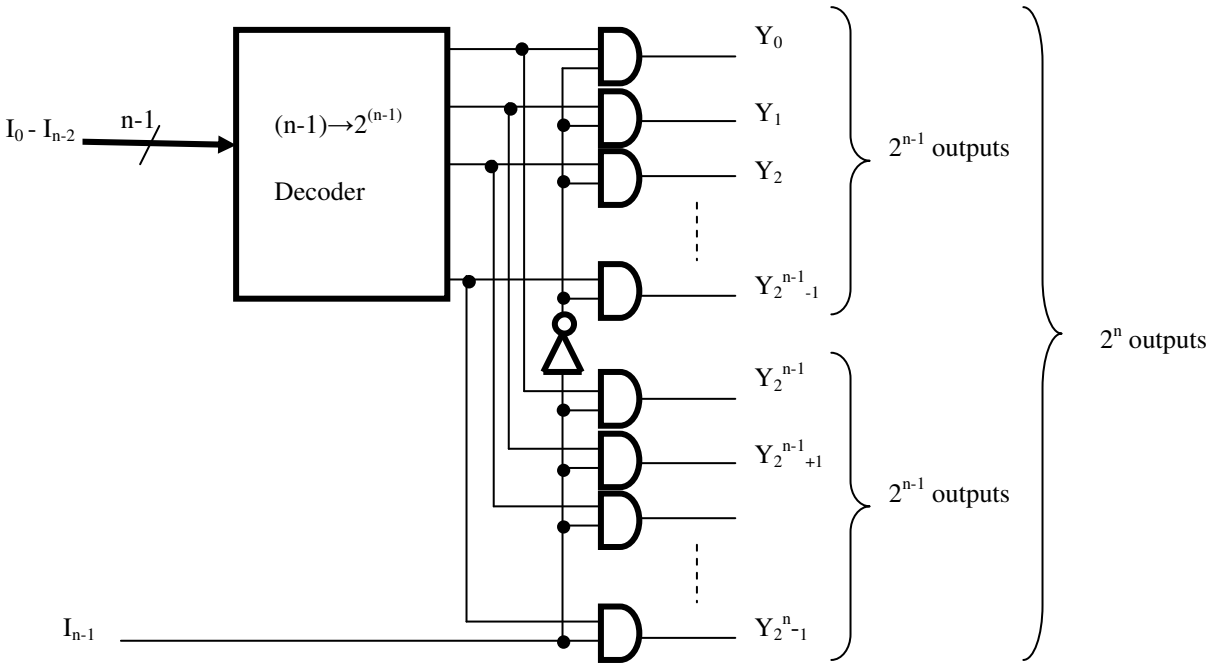


Figure 1.26 – A Recursive Decoder

In Figure 1.27 we show a 3→8 decoder built that way:

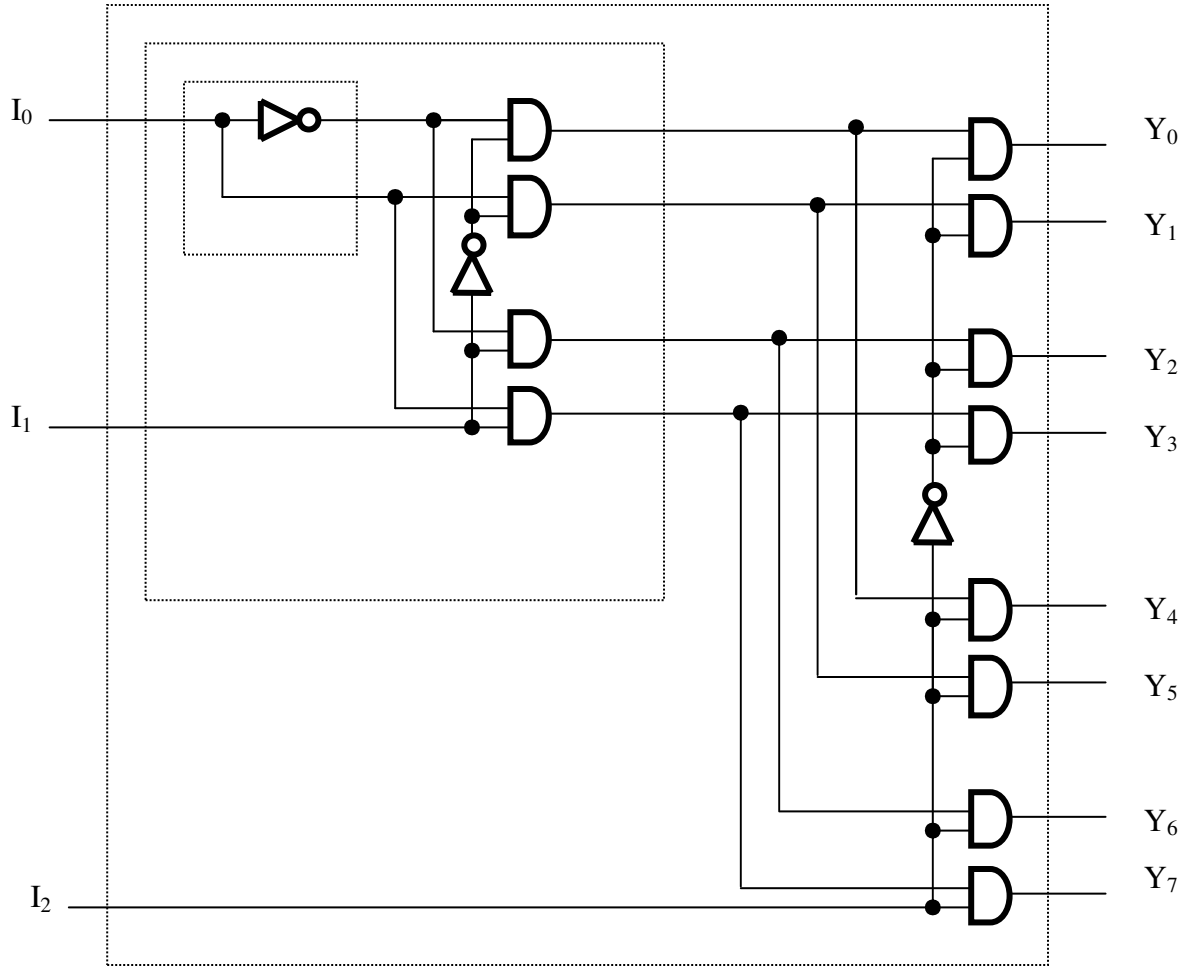


Figure 1.27 – A complete 3→8 recursive decoder

It is quite easy to see that the delay of such a decoder is given by $D(n)=D(n-1)+T$ where T is the delay of a single gate. This means that we have $D(n)=n \cdot T$. Note that this structure is similar to the first way in which we built an n input AND gate (Figures 1.18 and 1.19). We can use a "tree style" approach to get a logarithmic delay. Try to do that as a homework exercise.

The cost of the system in Figure 1.26, is n inverters and much more AND gates so we just count the AND gates as the cost. Since we see that the cost follows the recursive equation $C(n)= 2 \cdot 2^{n-1}+C(n-1)= 2^n + C(n-1)$, we have a geometric sequence with q=2.
Since C(1)=0, C(2)=4, we have $C(n)= 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 4$.

There are two more issues in designing such systems that we did not consider. One is the length of the lines, i.e., the connecting wires. This has to do with the area of silicon that is required in

order to implement the design on silicon. We will not discuss that issue. The other thing is the Fan out of the gates. The gates are electronic devices which have output and input currents. Since the output current of a gate is limited, it can "drive" only a limited number of gates. The number of gate inputs that can be driven by the output of a gate is called the Fan out of that gate. A typical value of the Fan out is 10 to 20. We would like to analyze a much severe case where the fan out of a gate is only 2. (Less then 2 means that we can connect the output of a gate only to a single input. This is too restrictive.)

In our decoder, we see that each AND gate drives two other gates, so there is no problem there. However, the inverters drive up to 2^{n-1} inputs, i.e., the number of the inputs that should be driven by the inverters is exponential! How can we overcome such a problem when the allowed fan out is only two?

The answer is that we should build a "tree" of inverters to produce 2^{n-1} inverted outputs and 2^{n-1} non-inverted outputs from the I_{n-1} input:

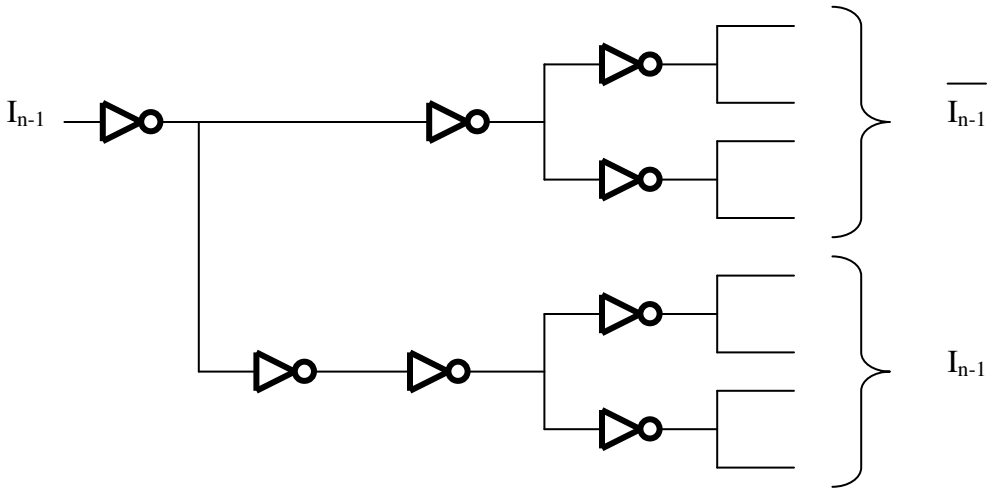


Figure 1.28 – A fan out expansion tree

Note that since the depth of such a tree is about n, we almost did not increase the delay of the decoder.

1.13 **Multiplexers**

A multiplexer (Mux), as a decoder, is one of the basic devices used in building computers. An n→m multiplexer, n>m, is a device with n inputs and m outputs. It also has some select inputs that determine which of the inputs are transferred to the outputs.

1.13.1 A simple mux

We follow our design procedure: First, define the required device. Then, build its truth table. Then, find its equations from the truth table. Then implement it with gates. So, we first define the simplest multiplexer which is a 2→1 multiplexer. It has two data inputs A and B (or I_0 and I_1) and a single data output, Y. It also has a single select input denoted by S. Its drawing and function is given in Figure 1.29.

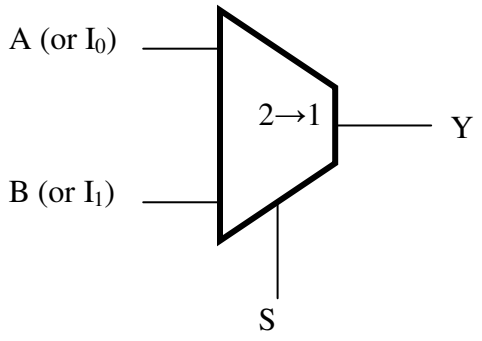


Figure 1.29a – The schematic drawing of 2→1 multiplexer

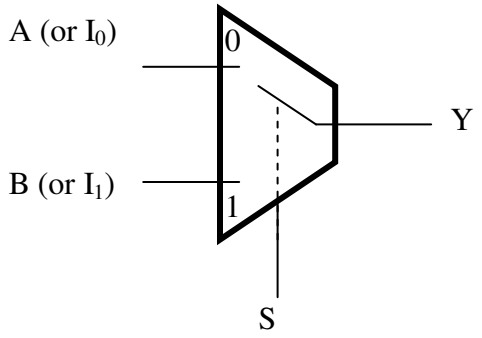


Figure 1.29b – A 2→1 mux selects between the 2 inputs

As shown in Figure 1.29b, the multiplexer functions as a switch. The S input determines which of the two inputs is "connected" to the output Y. When S="0", we have Y=A (or Y=I_0). When S="1", we have Y=B (or Y=I_1). The function of the mux can be written as:

$$Y = \begin{cases} A & \text{if } S=0 \\ B & \text{if } S=1 \end{cases}$$

The truth table is therefore:

S	A (I_0)	B (I_1)	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

The logic function of a mux is very simple:

$Y = A \cdot \overline{S} + B \cdot S$ (or if we use the other notation: $Y = I_0 \cdot \overline{S} + I_1 \cdot S$).

The implementation using gates is also simple:

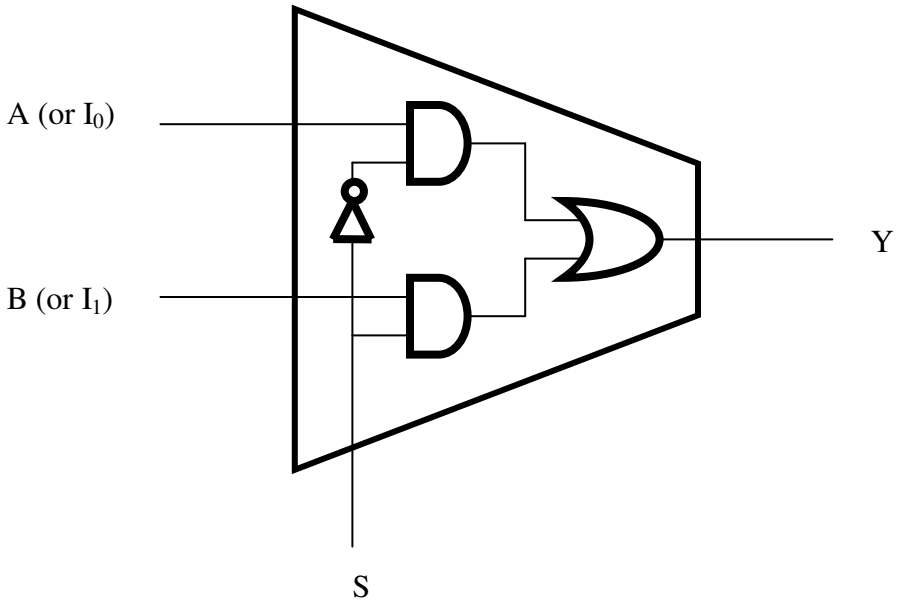


Figure 1.30 – The inside of 2→1 multiplexer

1.13.2 First expansion: A 2n→n mux , also called n*(2→1) mux

The 2n→n mux has 2n inputs and n outputs as shown in Figure 1.31. The data inputs represent two n bit numbers and the S input determines which of them is transferred to the n outputs. We denote the A inputs by A[n-1:0]=[A_{n-1},A_{n-2},…,A_0], the B inputs by B[n-1:0]=[B_{n-1},B_{n-2},…,B_0], and the data outputs, Y by Y[n-1:0]=[Y_{n-1},Y_{n-2},…,Y_0].

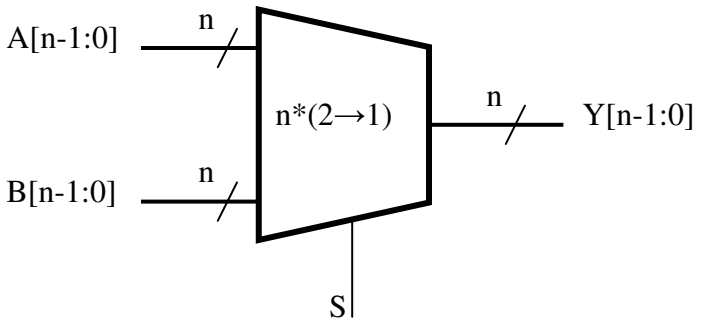


Figure 1.31 – The schematic drawing of an n*(2→1) multiplexor

The function of the mux can is given by:

$$Y[n-1:0] = \begin{cases} A[n-1:0] & \text{if } S=0 \\ B[n-1:0] & \text{if } S=1 \end{cases}$$

This can be implemented using n regular 2→1 muxes, i.e., the A_i, B_i and the Y_i are connected to a single 2→1 mux. So, now we understand why we called that mux an n*(2→1) mux. In Figure 1.32 we see the internal structure of a 3*(2→1) mux.

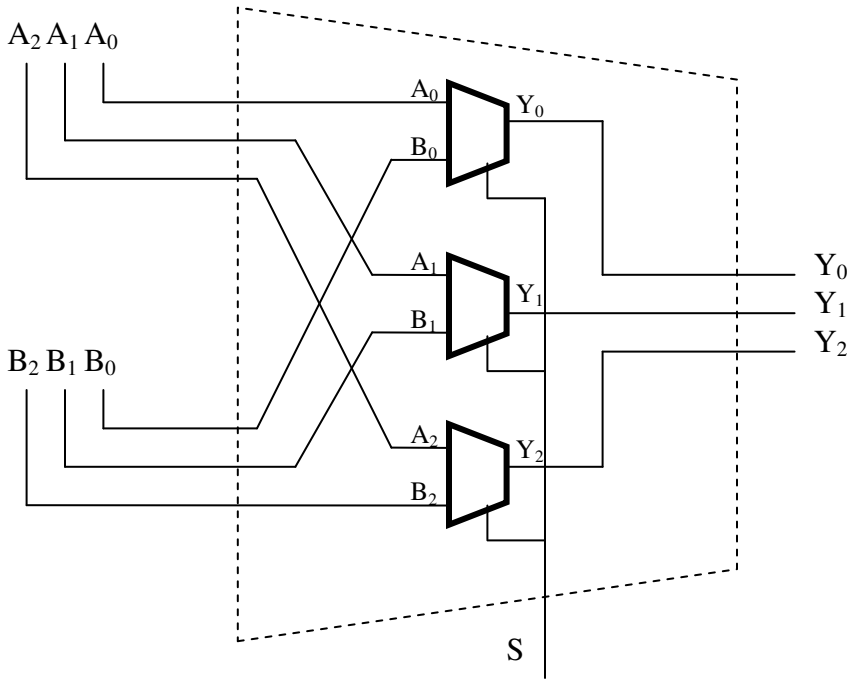


Figure 1.32 – The inside of a 3*(2→1) mux

1.13.3 Second expansion: A $2^k \rightarrow 1$ mux

The $2^k \rightarrow 1$ mux has 2^k inputs and a single output as shown in Figure 1.31. There are also k select inputs denoted S[k-1:0]=[S_{k-1},S_{k-2},…,S_1,S_0]. There are 2^k combinations to the select lines. When S[k-1:0]=i, i.e., the combination of [S_{k-1},…,S_0] represents the number i, the i-th input is transferred to the output Y. Since there are 2^k inputs we have chosen to denote those inputs by I_0, I_1, …, I_{2^k-1}. Note that the simple $2 \rightarrow 1$ mux we studied before, is a particular case with k=1.

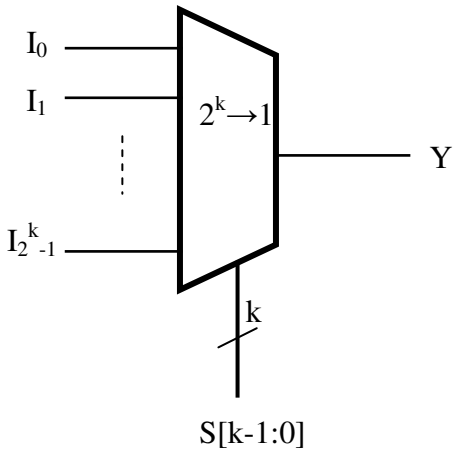
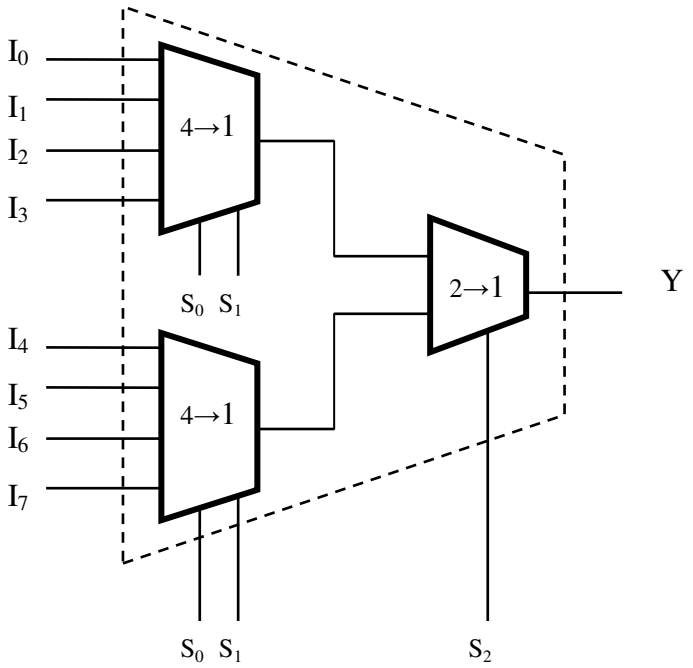


Figure 1.33 – The schematic drawing of a $2^k \rightarrow 1$ multiplexor

We would like to build an $8 \rightarrow 1$ (i.e., a $2^3 \rightarrow 1$) mux using 2 muxes of $4 \rightarrow 1$. This is pretty easy. We have to add another select input, S_2, to the two select inputs, S_1 and S_0, of the $4 \rightarrow 1$ muxes (i.e., $2^2 \rightarrow 1$ muxes). This S_2 input will choose between the two outputs of the two $4 \rightarrow 1$ muxes, as in Figure 1.34 below.



The number represented by S[k-1:0] is the serial number of the input transferred to the output.

S_2 S_1 S_0		
0 0 0	=	0
0 0 1	=	1
0 1 0	=	2
0 1 1	=	3
1 0 0	=	4
1 0 1	=	5
1 1 0	=	6
1 1 1	=	7

Figure 1.34 – A $2^3 \rightarrow 1$ mux build of two $2^2 \rightarrow 1$ muxes

Since adding a select line exactly doubles the number of combinations, we can similarly build a $2^k \rightarrow 1$ mux using two $2^{k-1} \rightarrow 1$ muxes and a single $2 \rightarrow 1$ mux. Thus, we can build a $2^k \rightarrow 1$ mux recursively. In Figure 1.35 we see the recursive definition of such a mux.

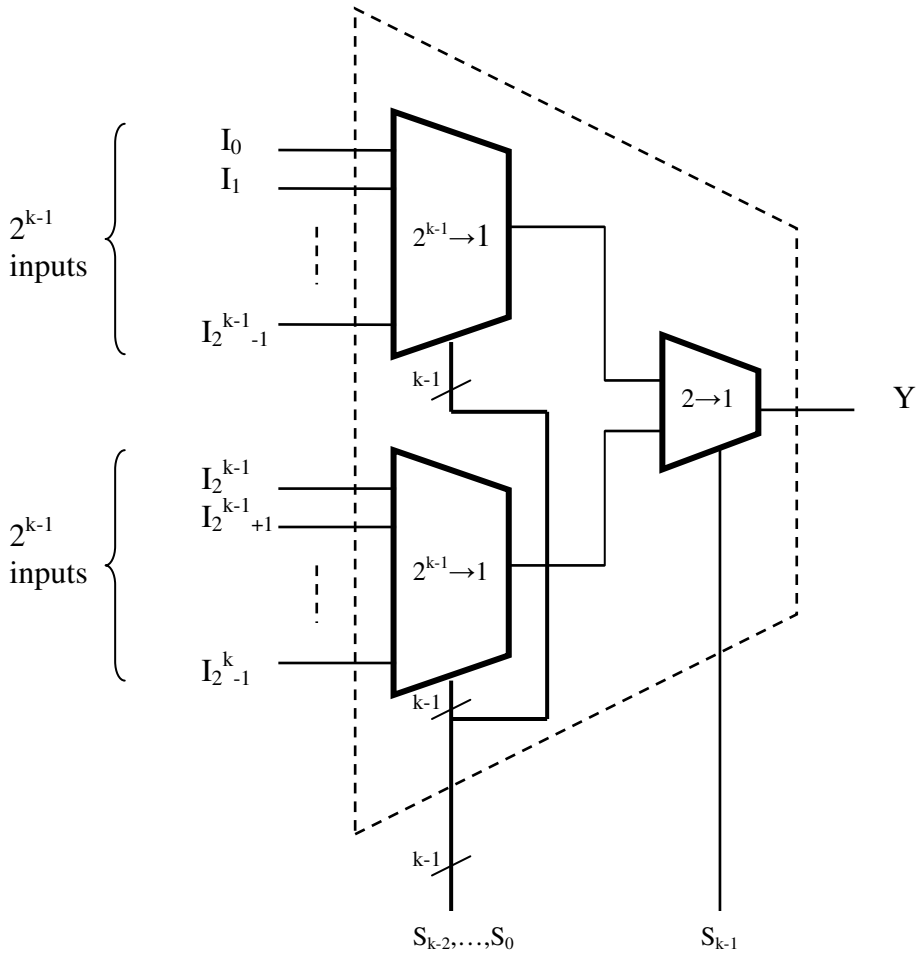


Figure 1.35 – Building a mux recursively

The cost equation is $C(k) = 2 \cdot C(k-1) + C(1)$. This means that the cost is actually: $C(k) = C(1) \cdot [1+ 2+4+\ldots+2^{k-1}] = C(1) \cdot (2^k-1)$. Note that here we look at k instead of n where n is the number of the inputs and follows $n=2^k$. So $C(n)=C_{2\rightarrow1} \cdot (n-1)$.

The delay equation is $D(k) = D(k-1) + D(1)$. This means that the delay is given by $D(k) = k \cdot D(1)$ or $D(n) = \lg_2 n \cdot D_{2\rightarrow1}$.

In Figure 1.36 we show the entire tree of an $8 \rightarrow 1$ mux.

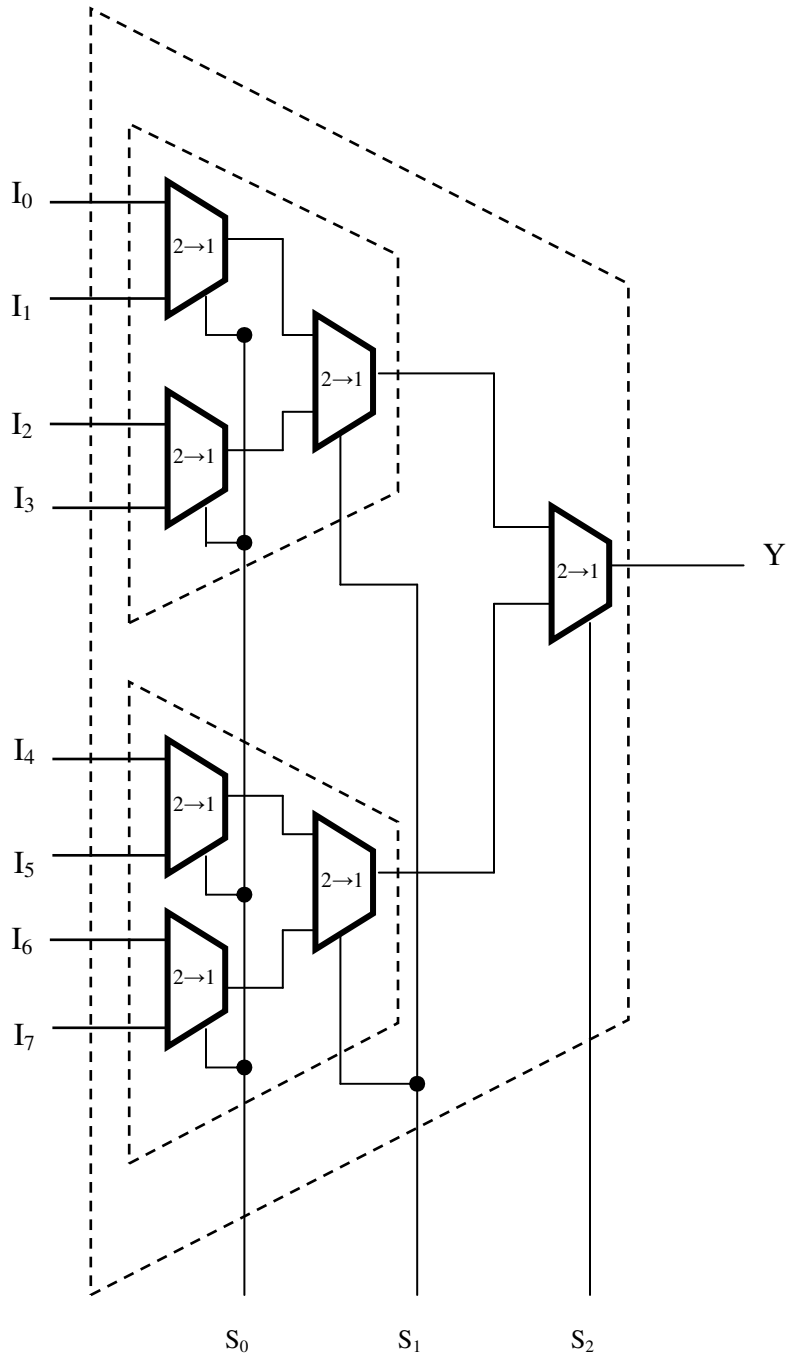


Figure 1.36 – The entire recursion depth in an 8→1 mux

1.13.4 Third expansion: An n*($2^k \rightarrow 1$) mux

The n*($2^k \rightarrow 1$) mux has 2^k inputs, n bits each, i.e., each input represent an n bits binary number, and a single n bits output as shown in Figure 1.37. There are also k select inputs denoted S[k-1:0]=[S_{k-1},S_{k-2},…,S_1,S_0]. There are 2^k combinations to the select lines. When S[k-1:0]=i , the i-th input is transferred to the output Y. Since there are 2^k inputs we have choose to denote those inputs by I_0, I_1, …, I_{2^k-1}, sometimes denoted A,B,…,Z.

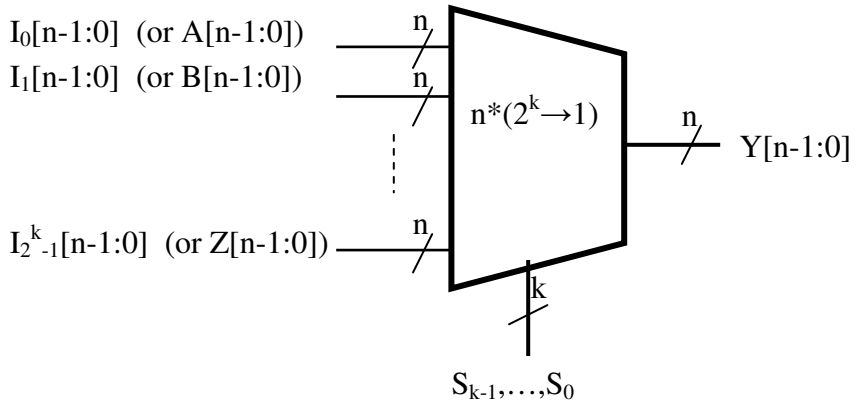


Figure 1.37 – The schematic drawing of an n*($2^k \rightarrow 1$) multiplexer

Similarly to the first expansion, the n*($2^k \rightarrow 1$) mux is built of n muxes of $2^k \rightarrow 1$, each of them takes care for one of the n bits. An example of a $12 \rightarrow 3$ mux, i.e., a 3*($2^2 \rightarrow 1$) mux, is given in Figure 1.38 below.

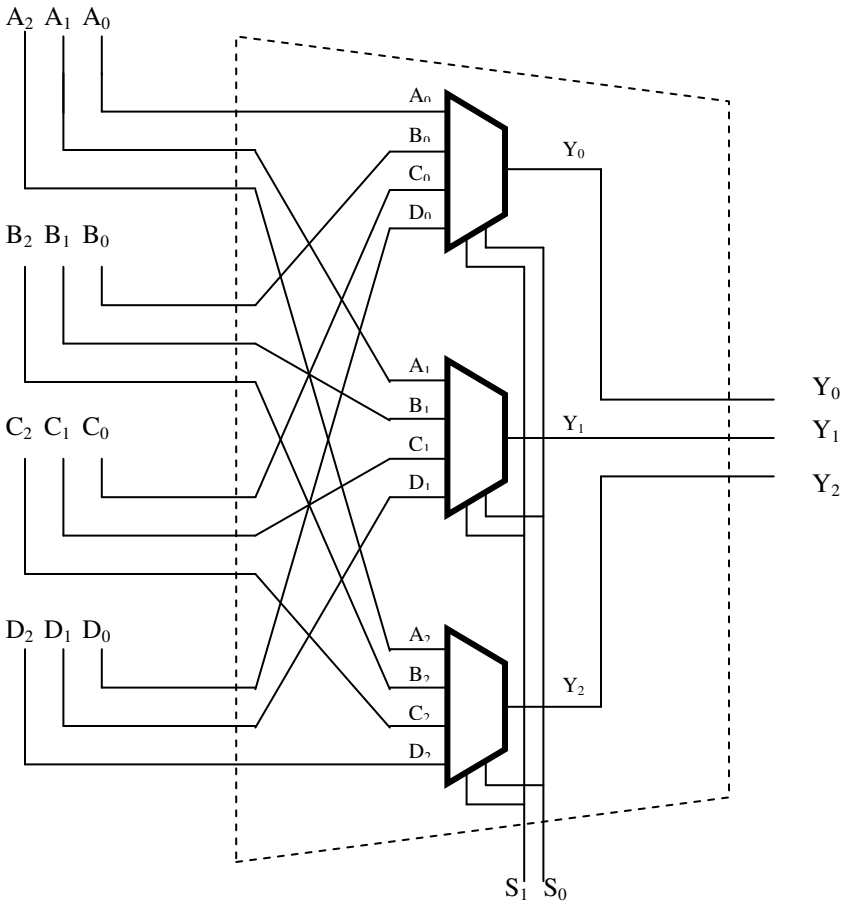


Figure 1.38 – The inside of a 3*($2^2 \rightarrow 1$) mux

We can also take apart the 4→1 muxes, which , as we already know, are built of 2→1 muxes:

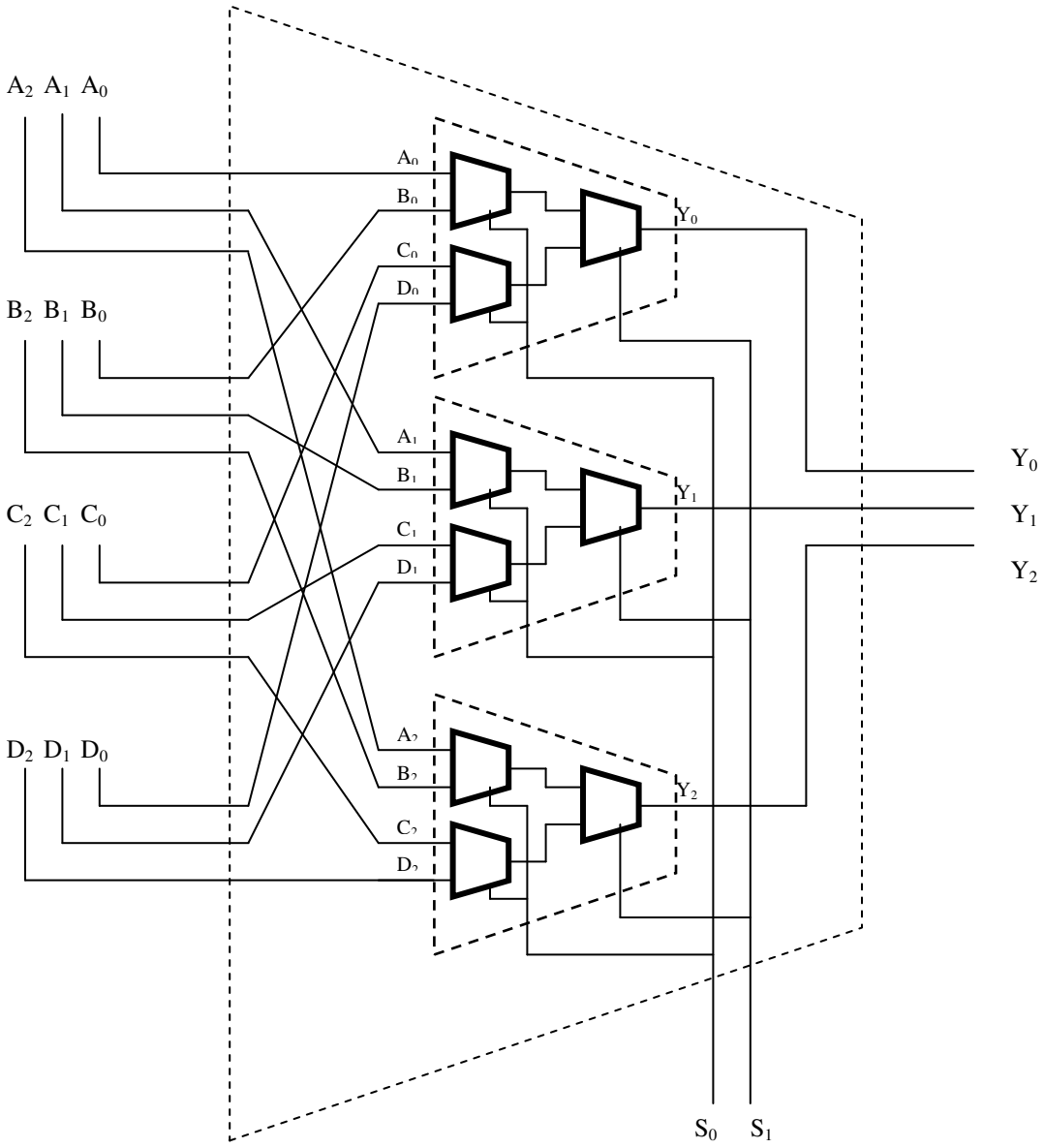


Figure 1.39 – The inside of a 3*($2^2 \rightarrow 1$) mux in detail

Figure 1.40 below shows the entire muxes family:

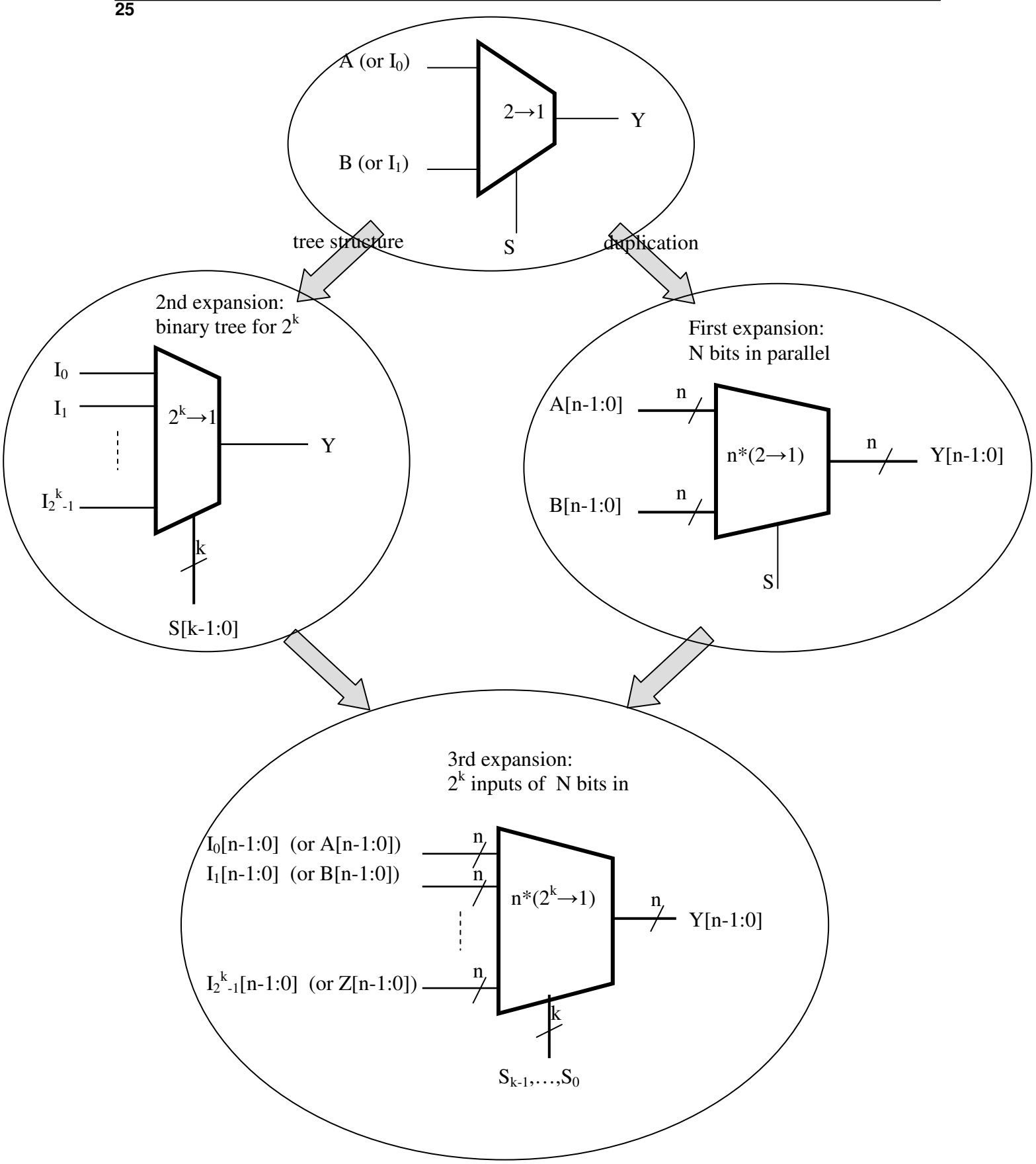


Figure 1.40 – The entire multiplexers family

We have only one last thing to say about muxes and decoders. They complement each other. A mux is the "inverse" of a decoder. To show that, we will change our interpretation of decoders. Let us look at a 2→4 decoder that has an enable input denoted E. If E="0" all outputs of the decoder are "0". If E="1", then the output selected by the code, or combination, of the inputs is "1". So, one can see the decoder as a switch controlled by the inputs that transfers the E data into one of the output as in Figure 1.41.

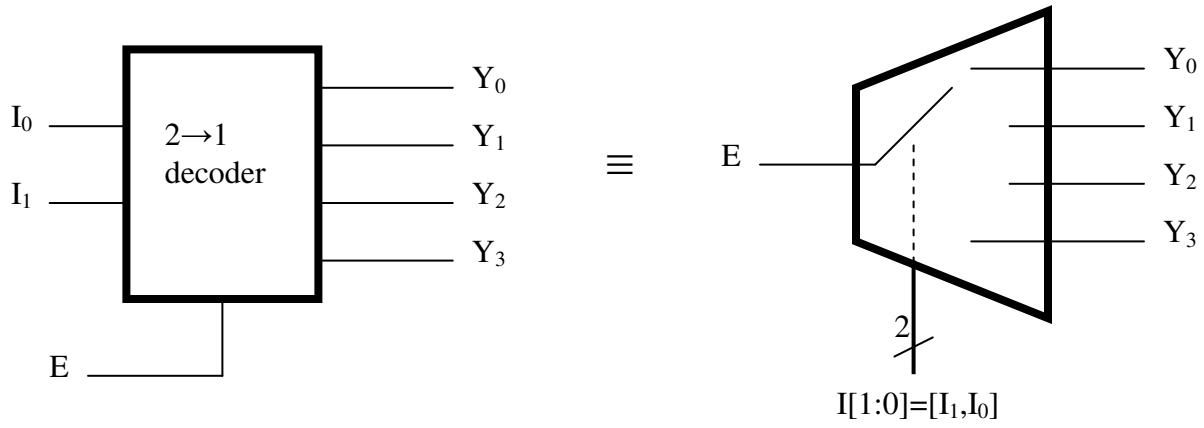


Figure 1.41 – A decoder as a controlled switch

We can use Muxes and Decoders to multiplex multiple data streams on a single line as in Figure 1.42. This is called TDM, Time Division Multiplexing, since when we sequentially change the selection code S[1:0]=0→1→2→3→0→1→… etc., we have a different data stream appearing on the line at different times. Note that the rate of switching the select lines should be 4 times higher than the rate in which the data streams may change.

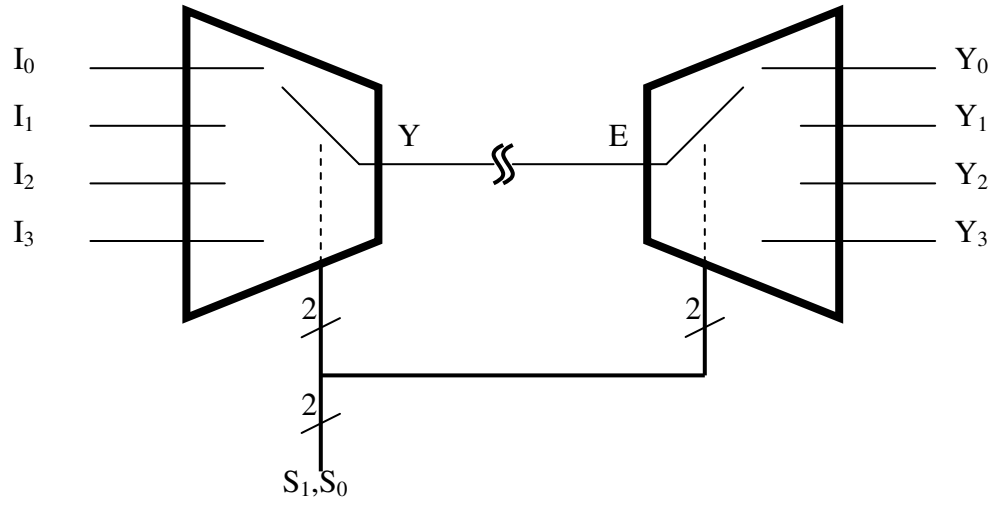


Figure 1.42 – 4 data lines sharing a single line

Suggested homework: 1)A "recursive" comparator 2) "ALT" detector 3) A "tree" decoder

3) **Adders and ALU circuits**

In this section we will use the knowledge we acquired in the previous two chapters to design Adders and ALU.

3.1) **A Half Adder**

We begin with designing a component called a Half Adder. This component, depicted in Figure 3.1 below, as well as its truth-table, is capable of adding two one bit numbers.

A	B	C_o	Y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

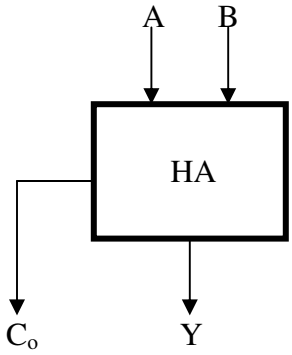


Figure 3.1 – A Half .s

The equations of a Half-Adder are easily found from its truth-tabe:

$$Y = \overline{A} \cdot B + A \cdot \overline{B} = A \oplus B$$

$$C_o = A \cdot B$$

So, the implementation of a Half-Adder (HA) is simple and involves only two gates:

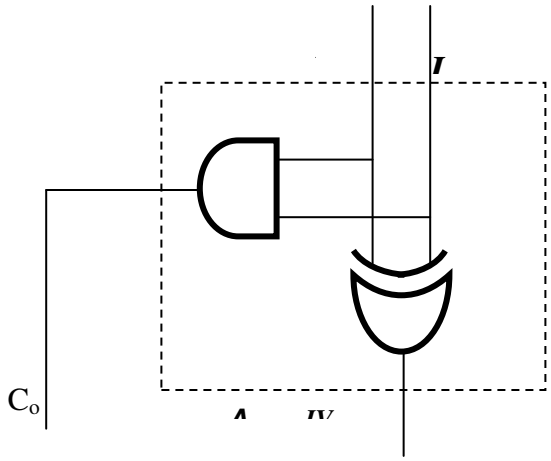


Figure 3.2 – The inside of a Half-Adder

The reason that this device is called a Half-Adder is that we need two of these in order to add longer numbers. Let us demonstrate it. We'll try to build an Adder that will add two 6 bits numbers A[5:0] and B[5:0]. We connect the A_i-th and B_i-th bits to a Half-Adder that produces the Y_i-th output and hope for good:

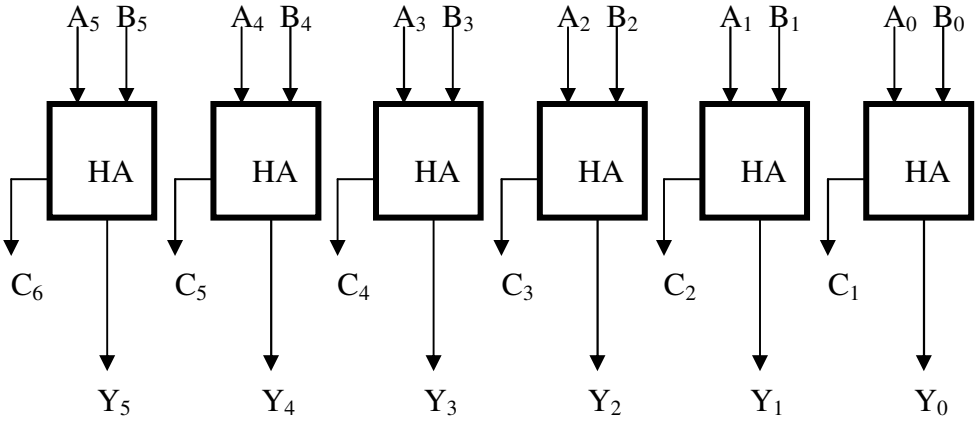


Figure 3.3 – Trying to build an adder .:

Let us now try to use the adder for adding some unsigned numbers. We do not have any problems in adding A=001100 and B=010001. But we seem to have a problem adding A=001100 and B= 001010. The 4 LSBs of Y[5:0], i.e., Y_0, Y_1, Y_2, and Y_3 are OK. But Y_4 is not, since the addition of A_3 and B_3 produces a carry, given by the signal C_4 which is "1", but has no influence on Y_4. Such an adder cannot handle cases of carry. We need to do some modifications to this design as is explained below.

3.2) **A Full Adder**

Let us try to imitate the way we, humans, do the addition of two binary numbers. Let us add the two numbers A=0111100 and B=0101010. We add the numbers bit by bit. When carry is produced, we add it to the next digit; (the result of each step is in red, older results are in blue)

	A = 0111100	
+	B = 0101010	

Calculating Y_0 and C_1:	00	we calc $A_0+B_0=[C_1,Y_0]=0+0=[0,0]$

Calculating Y_1 and C_2:	010	we calc $A_1+B_1+C_1=[C_2,Y_1]=0+1+0=[0,1]$

Calculating Y_2 and C_3	0110	we calc $A_2+B_2+C_2=[C_3,Y_2]=1+0+0=[0,1]$

Calculating Y_3 and C_4	10110	we calc $A_3+B_3+C_3=[C_4,Y_3]=1+1+0=[1,0]$

Calculating Y_4 and C_5	100110	we calc $A_4+B_4+C_4=[C_5,Y_4]=1+0+1=[1,0]$

Calculating Y_5 and C_6	1100110	we calc $A_5+B_5+C_5=[C_6,Y_5]=1+1+1=[1,1]$

Calculating Y_6 and C_7	01100110	we calc $A_6+B_6+C_6=[C_7,Y_6]=0+0+1=[0,1]$

The final 6 bits result:	1100110	

We can build a device that first adds the two digits A_i and B_i and then adds the carry, C_i. The only issue we have left is how to calculate the carry to the next digit, C_{i+1}:

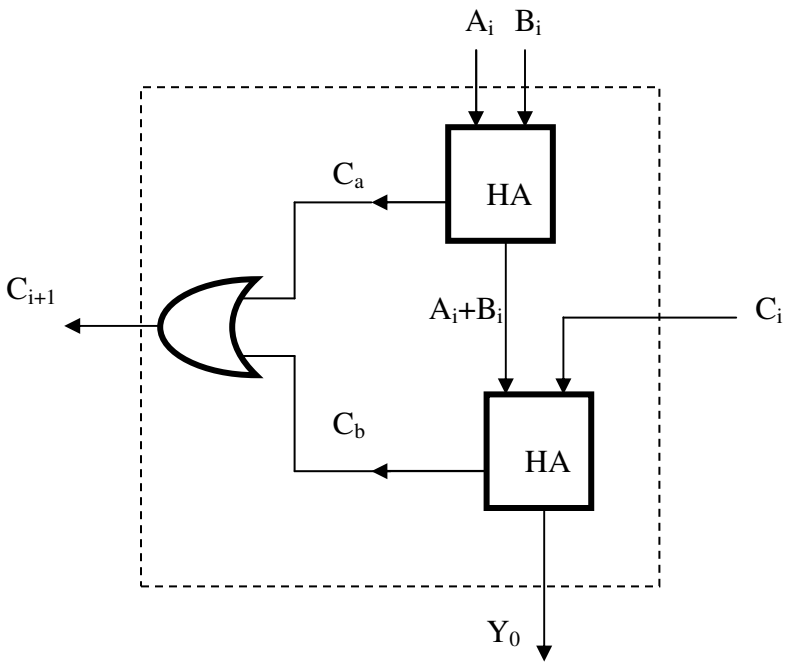


Figure 3.4 – A Full-Adder device

Such a device is called a Full-Adder (FA). In Figure 3.4 we demonstrate building a Full-Adder using two Half-Adders. We see that we first add the two digits and then add also the carry in. It is easy to see that we would have carry out only when two (or more) of the inputs A_i, B_i, C_i are "1"s (see the truth-table below). Let us now make sure that the circuit calculates C_{i+1} correctly:
When $A_i=B_i=$"1", then $C_a=$"1", and so $C_{i+1}=$"1". This is the only case in which $C_a=$"1".
When $C_i=$"0", we have no problem since $C_b=$"0", and so only C_a can cause C_{i+1} to be "1". This of course happens only if $A_i=B_i=$"1".
When $C_i=$"1", we have to worry only about the case in which we have $A_i \neq B_i$, since if they are equal, then if they are "0", A_i+B_i is also "0" and no carry is produced at all. If they are "1"s, then, although $C_b=$"0", we have carry since $C_a=$"1".
When $C_i=$"1" and $A_i \neq B_i$, we know that there must be carry. Since in this case we have $A_i+B_i=$"1", we also have $C_b=$"1" and therefore $C_{i+1}=$"1" as desired.

Thus, the circuit depicted in Figure 3.4 satisfies the truth-table of a Full-Adder (FA) that is shown below.

By the way, note that the unsigned number $[C_{i+1},Y]$ actually represents the number of "1"s in the set $\{ A_i, B_i, C_i \}$.

C_i	A_i	B_i	C_{i+1}	Y_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The schematic drawing of a Full-Adder is shown in Figure 3.5 below.

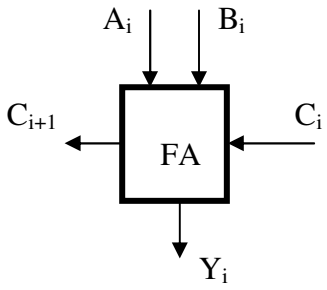


Figure 3.5 – The schematic drawing

We could have used the truth-table and Karnaugh maps to find the equations of Y and C_{i+1}. Those are:

$$Y = \overline{A_i} \cdot B_i \cdot \overline{C_i} + A_i \cdot \overline{B_i} \cdot \overline{C_i} + \overline{A_i} \cdot \overline{B_i} \cdot C_i + A_i \cdot B_i \cdot C_i =$$
$$= (\overline{A_i} \cdot B_i + A_i \cdot \overline{B_i}) \cdot \overline{C_i} + (\overline{A_i} \cdot \overline{B_i} + A_i \cdot B_i) \cdot C_i = (A_i \oplus B_i) \cdot \overline{C_i} + \overline{(A_i \oplus B_i)} \cdot C_i =$$
$$= A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$$

So we can draw the implementation of a Full-Adder using gates:

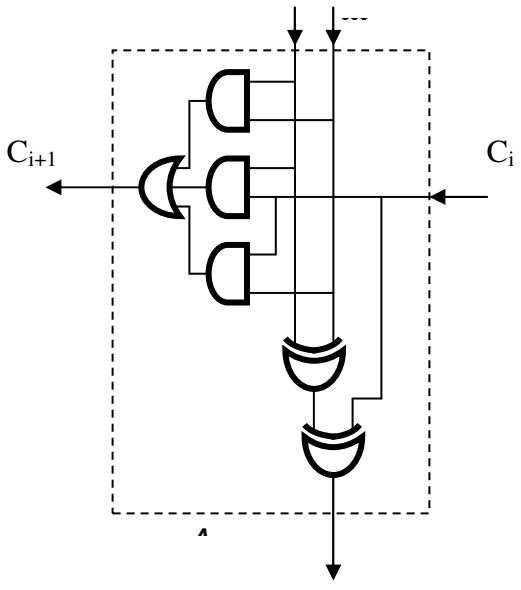


Figure 3.6 – The inside of a Full-Adder

We can use a slightly different equation for C_{i+1}:

$$C_{i+1} = A_i \cdot B_i + (A_i \oplus B_i) \cdot C_i$$

which changes the implementation slightly to:

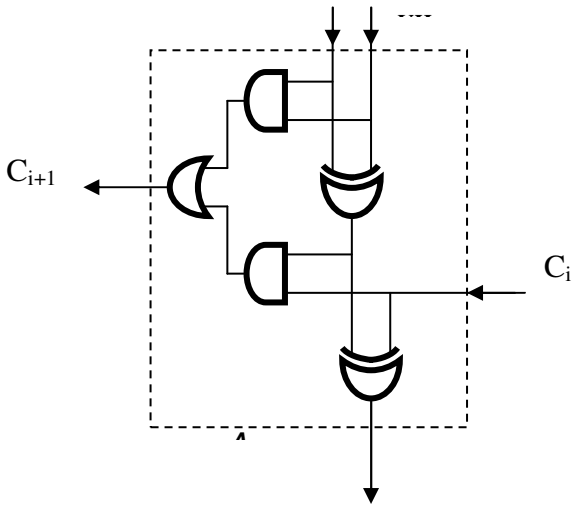


Figure 3.7 – Another implementation of a Full-Adder

Note that this implementation has fewer gates than the previous one, but the delay from A_i or B_i to C_{i+1} is larger since an extra gate is included in this path (if a 3 input gate has the same delay). Note that this implementation is identical to Figure 3.4.

We are ready now to build our first Adder.

3.3) A Ripple Carry Adder

A Ripple Carry Adder is an adder built of Full-Adders very similarly to our first try of Figure 3.3. This time we use Full-Adders instead of the Half-Adders and connect the carry-out of a digit to the carry-in of the next digit:

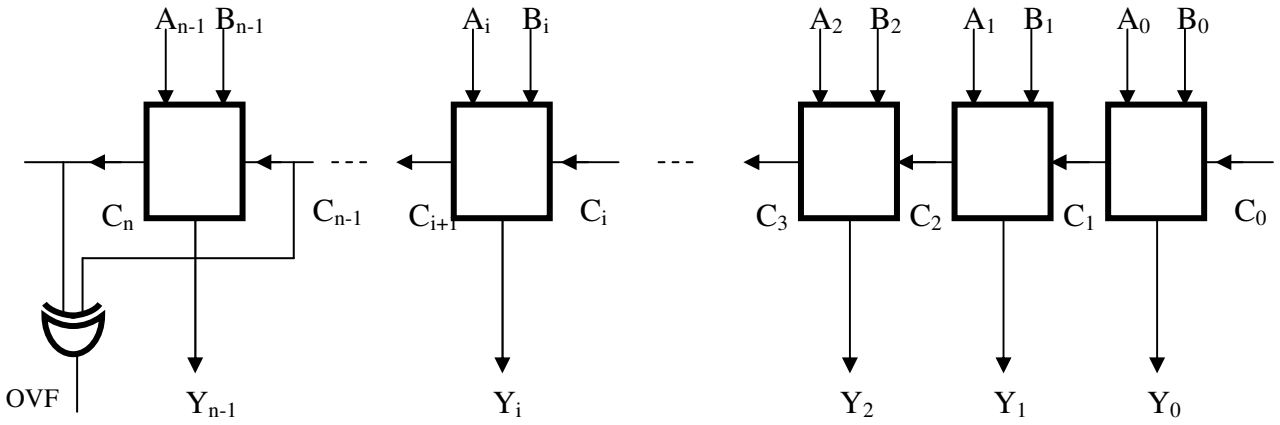


Figure 3.8 – A Ripple Carry Adder

This kind of an adder is called a Ripple Carry Adder (RCA) since the carry propagates through the Full-Adders like a wave.

Note that, as explained in the previous chapter, this adder is capable of adding two unsigned numbers, or two 2's Comp. numbers.

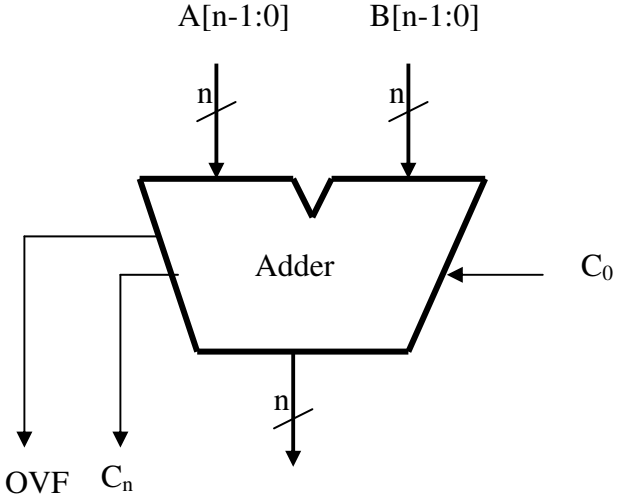


Figure 3.9 - The schematic

The cost of a RCA with width of n bits, i.e., an adder capable of adding two n bits numbers, is n times the cost of a single Full-Adder. Let us choose the implementation of Figure 3.7, i.e., we have 5 gates in a Full-Adder. Thus, we have

$C(n) = n \cdot C_{FA} = 5 \cdot n$

This means that when we double the width of our processor word we will need to pay twice the price.

The delay of a RCA depends on the carry propagation. The first stage calculating Y_0, will have the correct result after the delay of the two XOR gates of the right-hand side FA. The second stage, calculating Y_1, will have the correct result later, since we have to wait for C_1 and then wait another delay of a XOR gate till Y_1 is ready. For Y_2, we have to wait for C_2 which depends on C_1, and so on, as shown in Figure 3.10. Thus, a two gates delay should be added for every bit. Eventually we have:

$D(n) = n \cdot 2 \cdot T + T$, where T is the delay of a single gate.

This is so since in order to calculate C_n, which is required some times, we need to wait 3T until we produce C_1 and then, 2(n-1)T until the carry propagates through the next (n-1) stages, i.e., the total delay of this adder is (2n+1)T.
The time required for Y_{n-1} to be ready is 3T+(n-2)·2T+T=2nT.

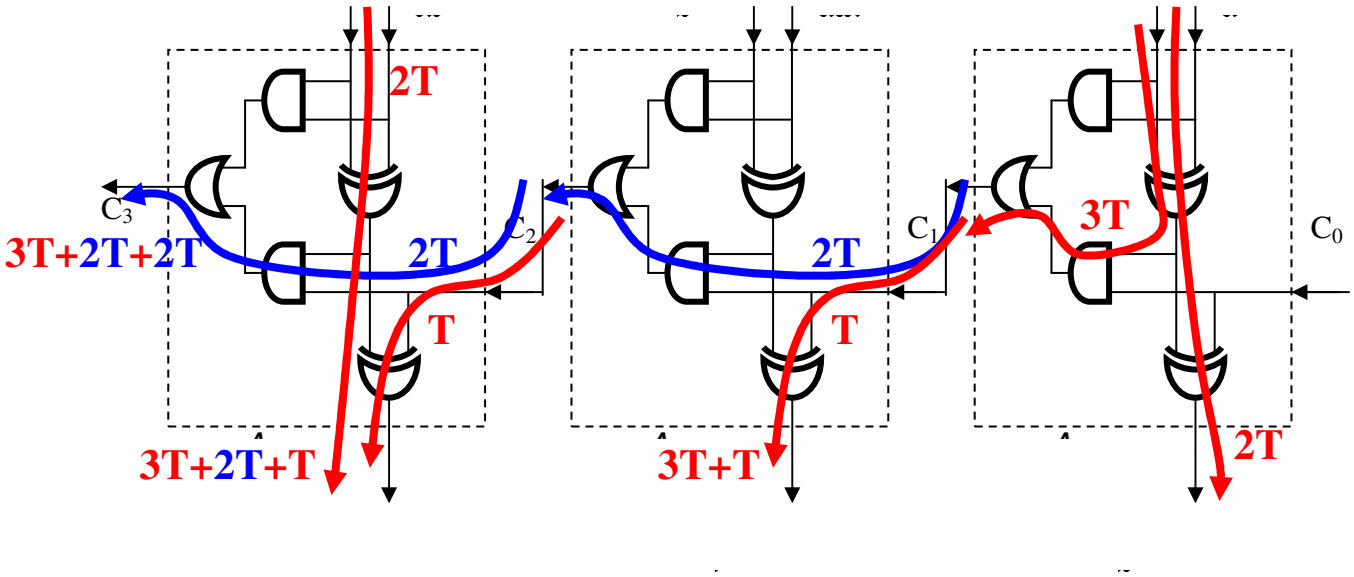


Figure 3.10 – Delay in a RCA

In case we would have used the implementation of Figure 3.6, even only for the 1st stage, we would shorten the time by T to all outputs (except Y_0), and so we would have $D(n) = n \cdot 2 \cdot T$, (for C_n).

We definitely have D(n) = **O(** n **)**, i.e., a linear delay.

This is not a good result. It means that when we double the word width of our computer, we also double the calculation time, i.e., we make our computer two times slower than before. We will later try to improve the performance. We would like to have a linear cost and a logarithmic delay. This is the best we can hope for.

3.4) ALU

Although we are not satisfied with the performance of our adder (the RCA), we are now ready to design the heart of a CPU, the Arithmetic Logic Unit (ALU). The ALU is the part of the CPU that does all the computations.

First, we'll improve the functionality of our adder by adding the capability of performing subtraction.

Since, as we already know, $A - B = A + \overline{B} + 1$, all we have to do in order to subtract B from A is to invert its bits and add the inverted number to A, and also add 1. Adding 1 is very easy. We'll use C_0 for that. This is actually the reason we used a Full-Adder in the 1st stage instead of a Half-Adder which is slightly faster and less expensive. Having the C_0 input is also useful for "add with carry" operations.

The SUB input controls the function of the circuit. In case SUB="0", B is not inverted (the XOR gate actually means that each bit of B is XORed with SUB), and C_0="0", thus the circuit calculates Y=A+B. In case SUB="1", B is inverted and C_0="1", so the circuit calculates Y=A-B. (Note: try to prove that when this circuit is used for unsigned numbers subtraction, the result is negative if C_n="0" !! I.e., C_n is definitely not equal to the borrow we expect when subtracting 2 unsigned numbers).

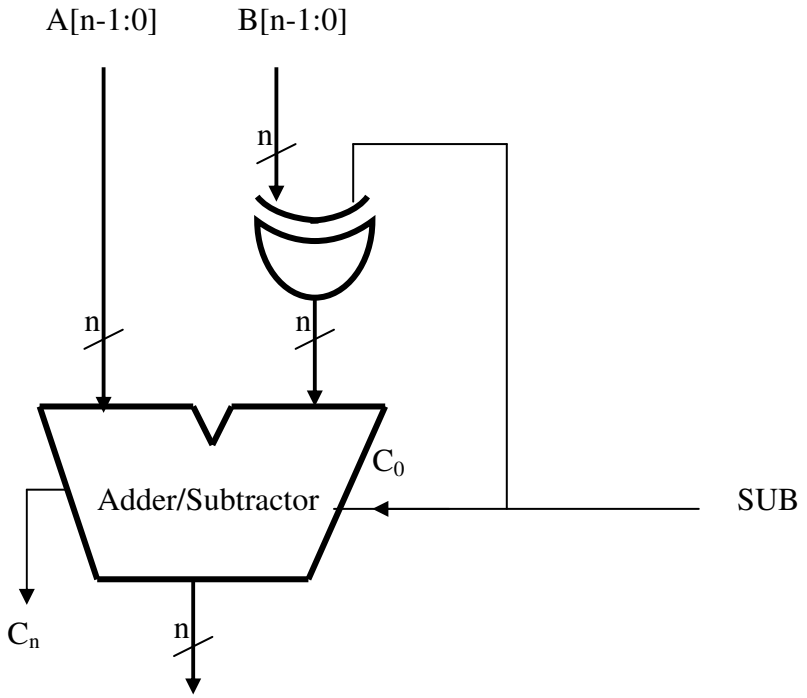


Figure 3.11 - An XVII

It is very simple to add some Logical operation to the Adder/Subtractor, and so, to form an ALU. Let us define the functionality of an ALU we want to design. We want the ALU to have two n bits inputs, the numbers A[n-1:0] and B[n-1:0], and an n bit output called Y[n-1:0]. We need to have some control lines, say 3 select lines, S_2, S_1, S_0, that will determine the operation performed by the ALU. The following table shows the desired functionality of our ALU:

S[2:0] Code			The operation performed by the ALU	The name of the operation
S_2	S_1	S_0		
0	0	0	Y= A plus B	Add
0	0	1	Y= A - B	Sub
0	1	0	Y= A · B	AND
0	1	1	Y= A + B	OR
1	0	0	Y= A $\oplus$ B	XOR
1	0	1	Y= $\overline{A}$	NOT
1	1	0	Y= A	
1	1	1	Y= B	

The way to get such functionality is to perform all those operations (almost) in parallel using 8 (actually 7) h/w units that perform those 8 operations. Some of the units are very simple, just n gates each (e.g., the AND, OR, XOR, NOT, i.e., all the logical operations). Some are much more complicated (e.g., the Adder/Subtractor). We use a multiplexer to select the appropriate output according to the select code. Since the Adder/Subtractor is used for two operations, its output is connected to two inputs of the multiplexer, the ones that are selected by [000] and [001].

We also add some special outputs. The ZR output is "1" if the ALU's output is zero. The C_n is "1" when there is a carry-out. We usually also add the NEG output which is the sign bit of Y, i.e., Y_{n-1}, and the OVF output, which tells us if 2's Comp overflow had occurred. (The last two outputs are not shown in Figure 3.12).

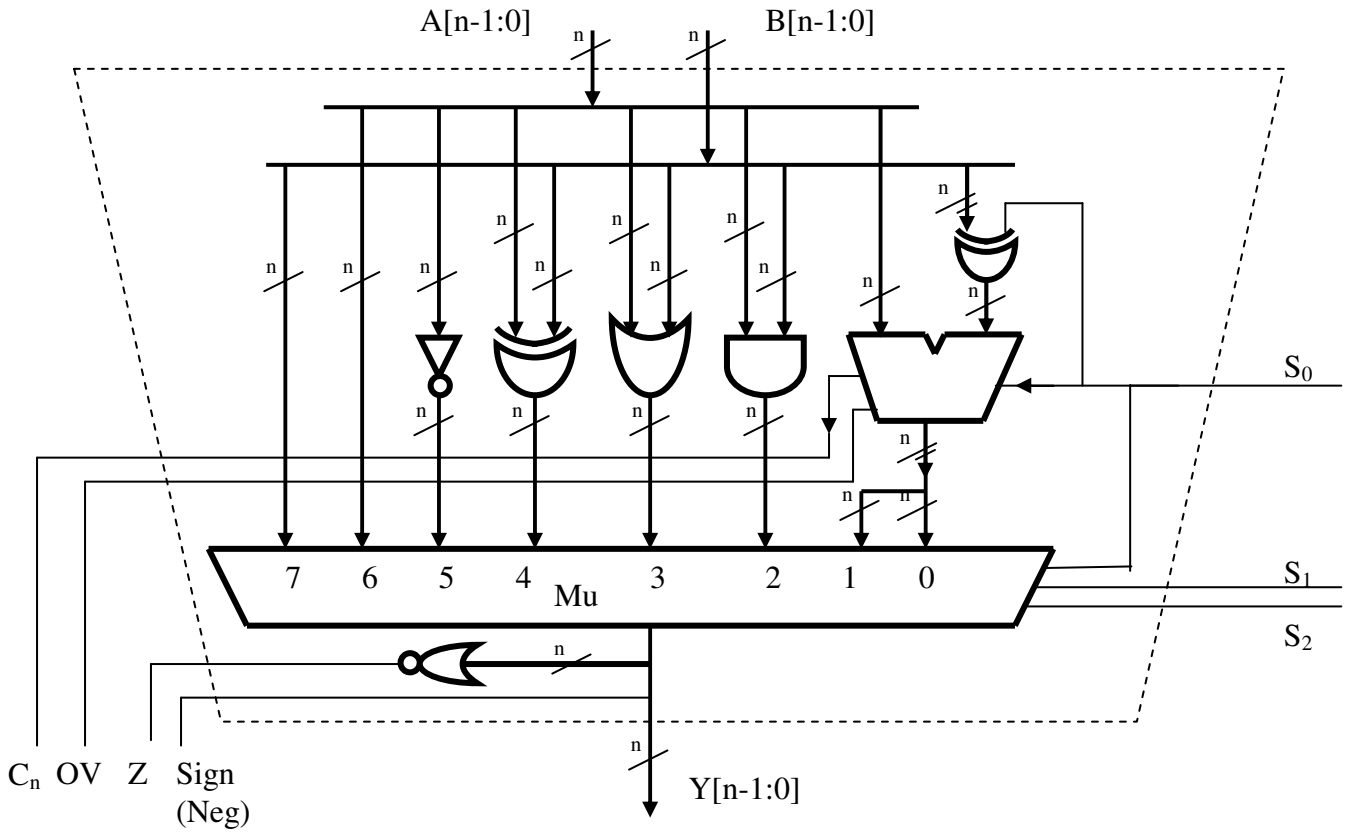


Figure 3.12 – The complete ALU

3.5) **A Conditional Sum Adder**

The idea in a Conditional Sum Adder (CSA) is to compute the addition of A[n-1:0] and B[n-1:0], i.e., two numbers of n bits each, by parts. By that we mean that the lower half of the numbers (lets call it the LSB part of the numbers), A[n/2-1:0] and B[n/2-1:0] are added separately than the upper part of the numbers (the MSB part), A[n-1:n/2] and B[n-1:n/2]. If we split the operation into these two parts, we can use two separate adders (each adding n/2 bits numbers). That way we do the computation faster since we do things in parallel (In both adders the carry propagates only half of the way, at the same time). Instead of having delay of D(n), we'll have a delay of D(n/2). However, there is a problem. For the MSB part, we should add A[n-1:n/2] and B[n-1:n/2] and $C_{n/2}$. But, $C_{n/2}$ is available only after D(n/2), i.e., when the LSB part calculation is finished. If we then start the calculation of the MSB part (adding the $C_{n/2}$ takes also D(n/2)), the total calculation time is D(n/2)+D(n/2), and no improvement had been acquired. The solution is to calculate 3 calculations in parallel: The 1st is A[n/2-1:0]+B[n/2-1:0]. The 2nd is A[n-1:n/2]+B[n-1:n/2] and the 3rd is A[n-1:n/2]+B[n-1:n/2]+1. These 3 calculations last D(n/2). After this time we have the 3 results and $C_{n/2}$ as well. The result of A[n/2-1:0]+B[n/2-1:0] is Y[n/2-1:0]. We'll use $C_{n/2}$ to select the appropriate result of Y[n-1:n/2]. If $C_{n/2}$ is 0, we take the result of A[n-1:n/2]+B[n-1:n/2] as Y[n-1:n/2]. If $C_{n/2}$ is 1, we take the result of A[n-1:n/2]+B[n-1:n/2]+1 as Y[n-1:n/2]. This is done by adding a Mux as depicted in Figure 3.13 below. This is actually a recursive way of building a CSA.

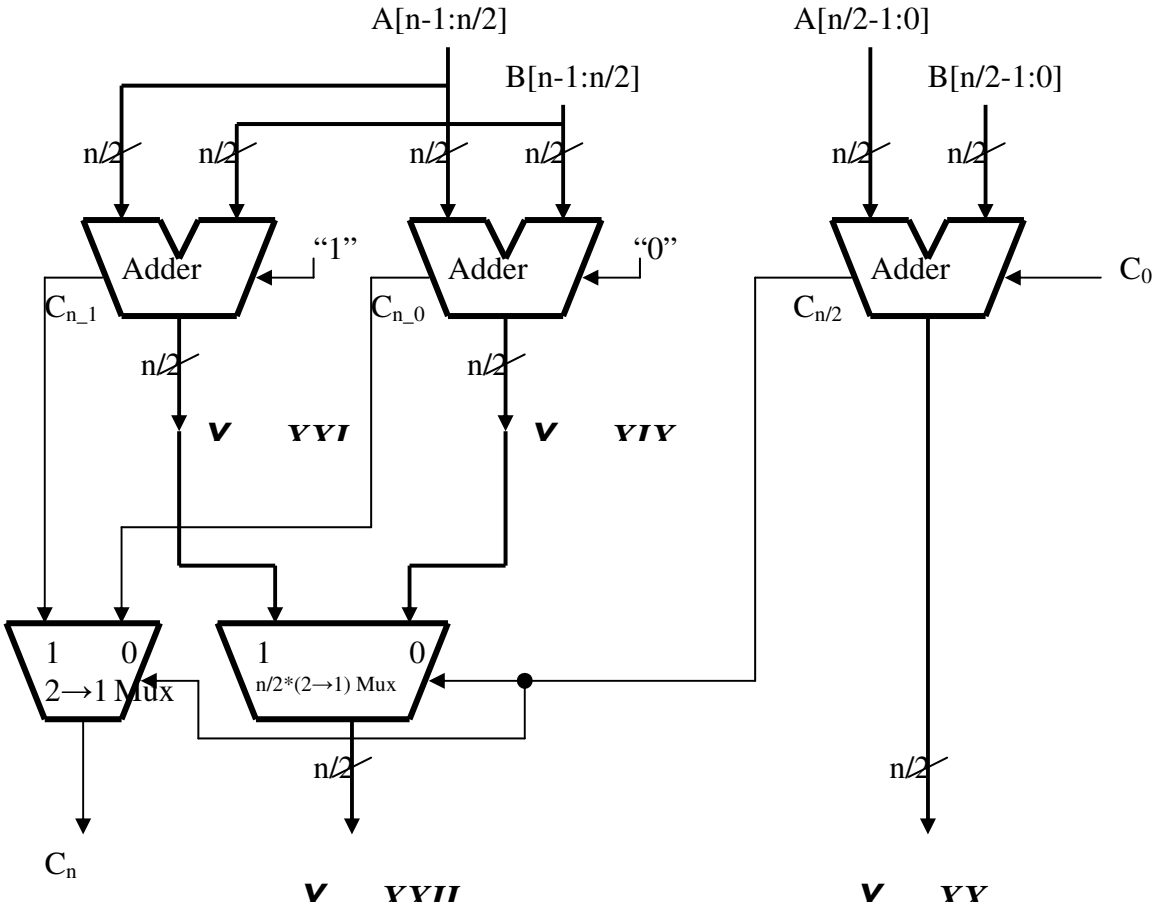


Figure 3.13 - Building a CSA

The delay is given by the recursive equation $D(n) = D(n/2) + T_{mux}$. This means that the delay is logarithmic: $D(n) = T_{mux} \cdot \lg n$. This is so since whenever we double the inputs we add a delay of a mux.

Figure 3.14 describes the entire recursion depth of a 4 bits CSA. The basic parts used are Full-Adders and Muxes.

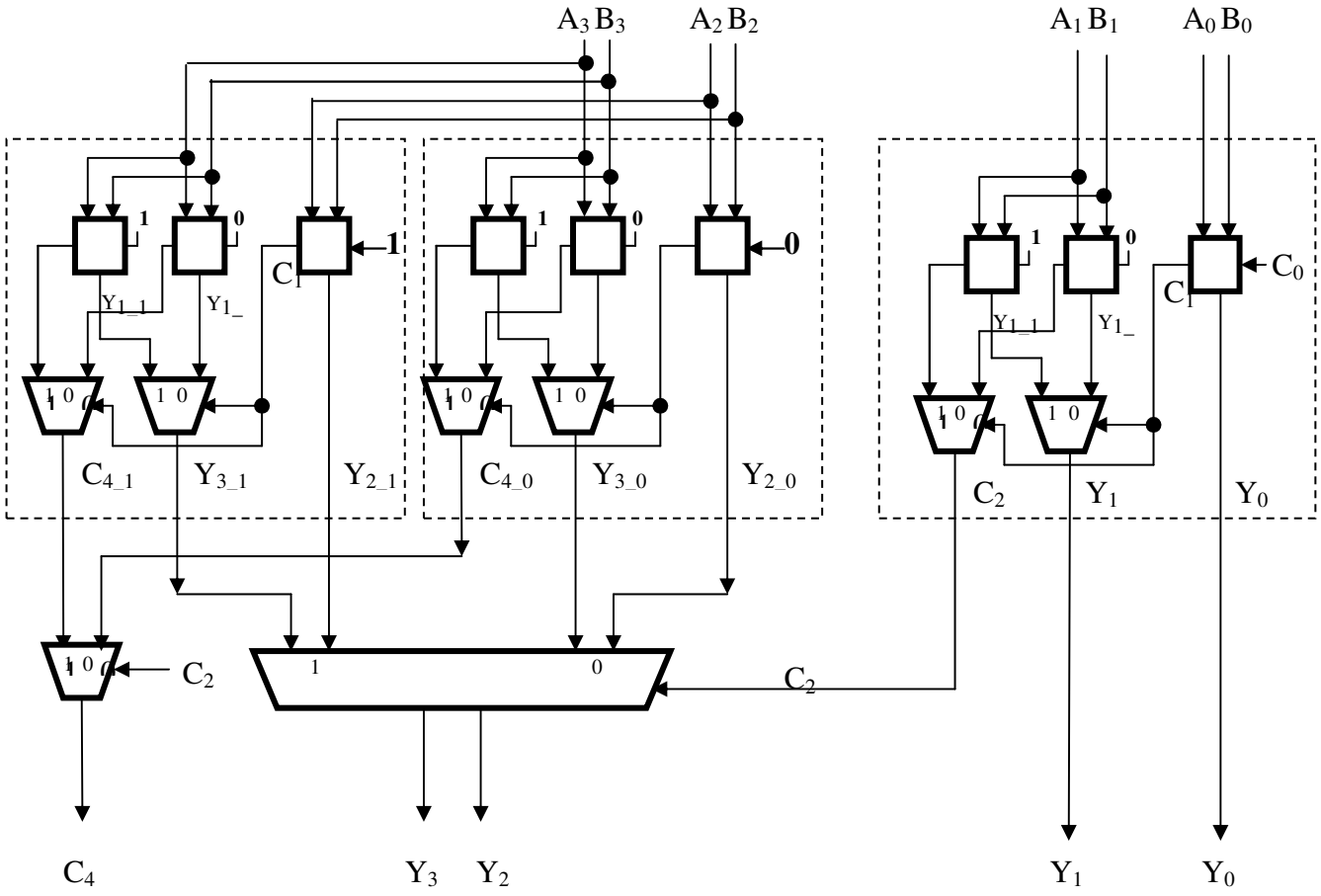


Figure 3.14 – The entire recursion in a 4 bits CSA

Since we have lg n recursive stages, where in each , the number of Full-adders is tripled, we should have about $3^{\lg n} = n^{\lg 3} = n^{1.58}$ Full-Adders in a CSA of n inputs. A similar count, reversing the order, i.e., about (n/2) Muxes at the last stage (not including the carry muxes), $3 \cdot (n/4)$ Muxes at the one before, $3^2 \cdot (n/8)$ Muxes before etc., which sums up to about $n^{1.58}$ Muxes:

$(n/2) + (3/2) \cdot (n/2) + (3/2)^2 \cdot (n/2) + \ldots = (n/2) \cdot [1 + (3/2)^2 + (3/2)^3 + \ldots + (3/2)^{\lg n - 1}] = n \cdot [(3/2)^{\lg n} - 1] = 3^{\lg n} - n = n^{\lg 3} - n = n^{1.58} - n \approx n^{1.58}$

The carry muxes, 1 for the last stage + 3 for the one before the last + $3^2 + \ldots + 3^{\lg n - 1} =$

$= (3^{\lg n} - 1)/2 = (n^{\lg 3} - 1)/2 \approx n^{1.58}/2$

Thus, the cost of such an adder is polynomial, C(n) = **O**($n^{1.58}$). This is not a good result. We employ such a design when we need to build a fast adder using already existing smaller adders. The next adder we'll discuss has a logarithmic delay and a linear cost.

3.6) **A Carry Look Ahead Adder**

Let us try to calculate the carry of all the stages in a logarithmic time. We notice that there are two reasons for having a carry out from the i-th stage of an adder, i.e, from adding A_i, B_i and C_i. (This addition produces a two bits number $[C_{i+1},Y_i]$). The carry C_{i+1} can be generated in the i-th stage or generated in a lower stage and propagate into C_{i+1} (through C_i). Carry is generated only when $A_i=B_i=1$. C_i will propagate through the i-th stage only if $A_i \neq B_i$. If this is the case, we have $C_{i+1}=C_i$. Carry will not be generated or propagated when $A_i=B_i=0$. So we can write the carry equation as:

$$C_{i+1} = A_i \cdot B_i + (A_i \oplus B_i) \cdot C_i$$

We denote the AND of A_i and B_i by G_i which stands for generate. We denote the XOR of A_i and B_i by P_i which stands for propagate. So we have a new equation for the relation of C_i and C_{i+1}:

$C_{i+1} = G_i + P_i \cdot C_i$

Note that since G_i and P_i depend only on A_i and B_i, we can calculate all of them in parallel at the same time. Let us build a device, very similar to a Full-Adder, that calculates Y_i, G_i and P_i (instead of Y_i and C_{i+1} calculated by a Full-Adder). Such a device is depicted in Figure 3.15 below.

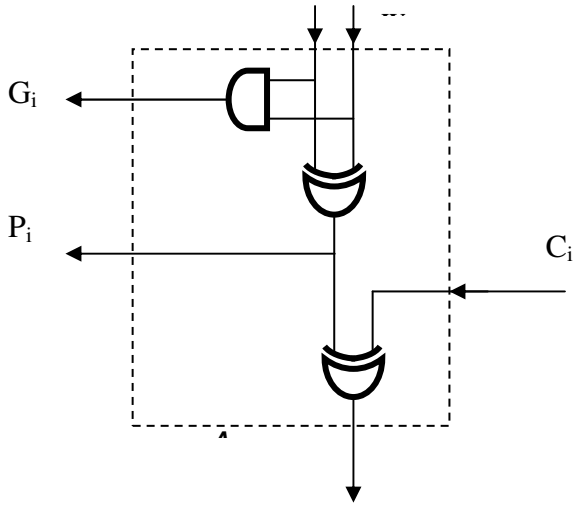


Figure 3.15 – Y and G,P calculation

Let us now build an adder using this device.

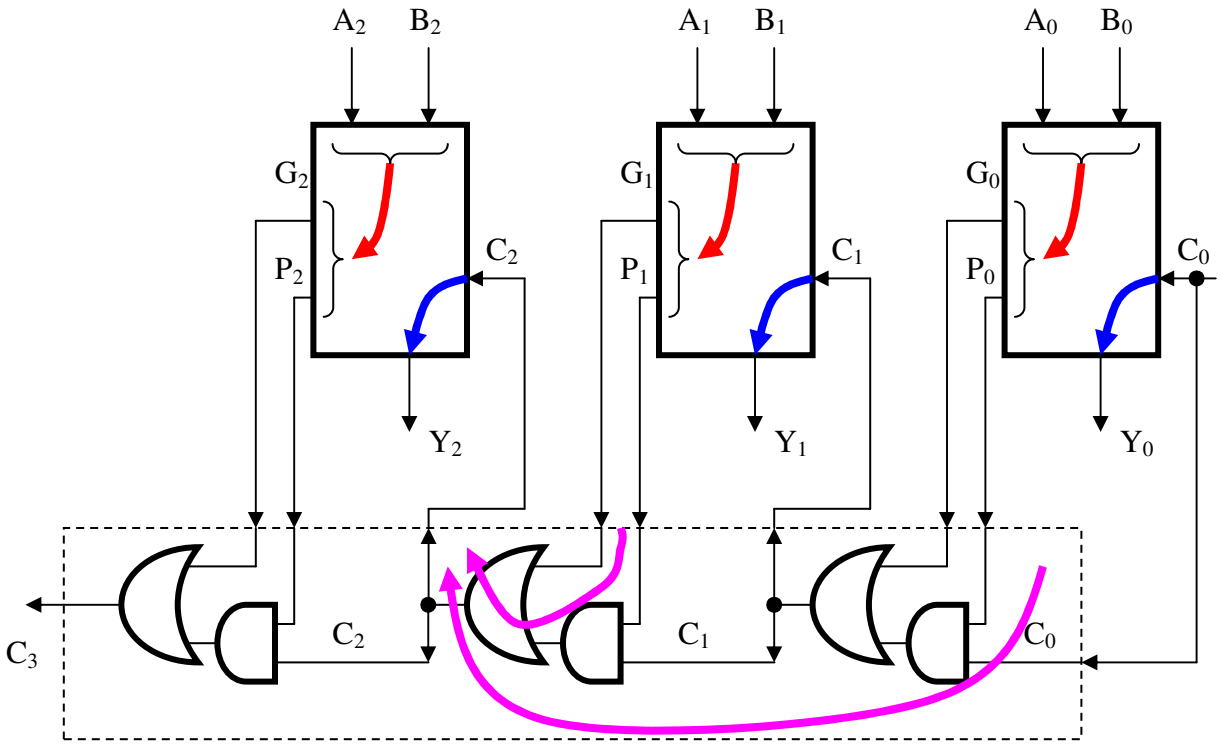


Figure 3.16 – A Carry Look Ahead Adder

Let us calculate the delay of such an adder. First, the G_i-s and P_i-s are all calculated in parallel (see the red lines in Figure 3.16). This takes T seconds, where T is the delay of a single gate. Then, we have to wait till the dotted box calculates all the C_i-s. Let's denote this time as T_{LA}. Following this, all the outputs become valid after an additional delay of a single gate (see the blue lines in Figure 3.16). So the delay of this adder is $T_{total} = 2T + T_{LA}$.

At a first glance, it seems that we earned nothing in this new design of an adder. We calculated C_1 using the equation which gave: $C_1=G_0+P_0 \cdot C_0$. We calculated C_2 similarly by $C_2=G_1+P_1 \cdot C_1$ and C_3 by $C_3=G_2+P_2 \cdot C_2$. As we see from Figure 3.16, this means that we connected the circuit in series. The delay from the minute all the G_i-s and P_i-s and C_0 are ready till the time where all the C_i-s are valid, is the delay of two gates multiplied by the number of bits in the adder, i.e., $T_{LA}=2nT$. Thus the total delay of that adder is $T_{total} = 2T + T_{LA}=(2n+2)T$. (Actually $T_{total} = (2n+1)T$ since C_n is calculated in parallel to Y_{n-1}). Again we reached a situation where the delay of the adder is linear.

So, we should try to improve the performance of the dotted box. This box is called a Carry Look Ahead (CLA) circuit and this is the reason for calling this adder a CLA adder. We can easily see that it is possible to calculate the carry with a logarithmic delay:
We start with calculating C_1 which is given by $C_1=G_0+P_0 \cdot C_0$.

Next, we deal with C_2. Here we have $C_2=G_1+P_1 \cdot C_1$. If we substitute C_1 with the equation above, we get:
$C_2=G_1+P_1 \cdot C_1=G_1+P_1 \cdot (G_0+P_0 \cdot C_0)=G_1+P_1G_0+P_1P_0C_0$.

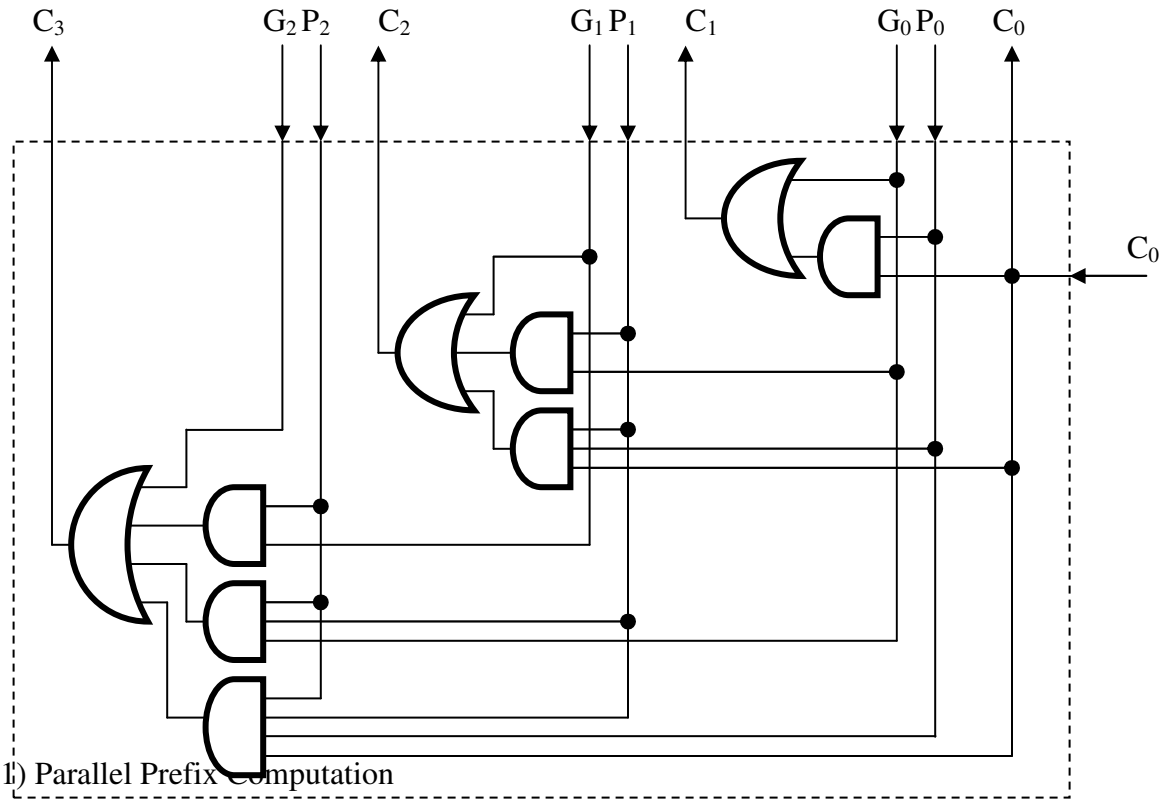
In the same way we get:
$C_3= G_2+P_2 \cdot C_2=G_2+P_2 \cdot (G_1+P_1G_0+P_1P_0 \cdot C_0)=G_2+P_2G_1+P_2P_1G_0+P_1P_0C_0$

and:
$C_4= G_3+P_3 \cdot C_3=G_3+P_3 \cdot (G_2+P_2G_1+P_2P_1G_0+P_1P_0C_0)=$
$= G_3+P_3G_2+P_3P_2G_1+ P_3P_2P_1G_0+ P_3P_1P_0C_0$

And eventually:
$C_n=G_{n-1}+ P_{n-1}G_{n-2}+P_{n-1}P_{n-2}G_{n-3}+ \ldots + P_{n-1}P_{n-2} \cdot \ldots \cdot P_2P_1G_0+ \ldots + P_{n-1}P_{n-2} \cdot \ldots \cdot P_0C_0$

As we see from that equation, we have here AND and OR functions of up to n+1 inputs. We already know that using binary trees, we can compute those in a logarithmic delay. Thus, we are sure that we can build a CLA adder having a logarithmic delay.

Note that there are many identical parts in the product terms of that equation. There is a chance that we can use this in order to produce all of the C_i-s by the same circuit (i.e., by a circuit that shares some partial products). Before we show that this can be done in linear cost, let us replace the "serial" CLA of Figure 3.16 with a "parallel" system which is possible if 3 and 4 input gates are available.



3.6.1) Parallel Prefix Computation

Figure 3.17 – A simple Carry Look Ahead circuit using multiple inputs gates

The algorithm enabling calculation of all of the carry outputs simultaneously, in a linear cost and a logarithmic delay, is called Parallel Prefix Computation (PPC).

We denote the unsigned number $[G_{i-1}, P_{i-1}]$ as σ_i. Actually σ_i equals the sum of A_{i-1} and B_{i-1} and also equals to $2G_{i-1}+P_{i-1}$. (and $\sigma_0= 2C_0$).

We define the operation $\otimes$ as follows:

$$\Pi_a= \sigma_a \otimes \sigma_b = \begin{cases} 0 & \text{if } \sigma_a=0 \\ \sigma_b & \text{if } \sigma_a=1 \\ 2 & \text{if } \sigma_a=2 \end{cases}$$

This operation helps us to detect whether the i-th stage has carry in its input. Let us look at σ_3. If it is 2, we definitely have carry at the input of the 3rd stage, i.e., $C_3=1$. This is so since $G_2=1$ (i.e., carry is generated in the 2nd stage). If $\sigma_3=0$, we are sure that there is no carry, i.e., $C_3=0$, since $G_2=P_2=0$ (no generation or propagation of carry in the 2nd stage). If $\sigma_3=1$, we do not know if we have carry or not. We know that $P_2=1$, i.e., carry will propagate through the 2nd stage. So we should look at σ_2. If $\sigma_2=2$, we have carry. If $\sigma_2=0$, we do not have carry. If $\sigma_2=1$, we must check σ_1. and so on. We therefore conclude that in order to determine C_i, the carry into the the i-th stage, we should actually look at Π_i, where Π_i is defined by:

$$\Pi_i = \sigma_i \otimes \sigma_{i-1} \otimes \sigma_{i-2} \otimes \ldots \otimes \sigma_2 \otimes \sigma_1 \otimes \sigma_0$$

Since σ_0 equals 0 or 2, all of the Π_i-s have values of 0 or 2. None of them can be equal to 1. Eventually, when converting back from Π_i to C_i the rule is therefore C_i="1" if $\Pi_i=2$.

Note that the operation $\otimes$ is associative:

$$(\sigma_a \otimes \sigma_b) \otimes \sigma_c = \sigma_a \otimes (\sigma_b \otimes \sigma_c)$$

This means that the ***order of computation*** does not matter. Note ***that the order of a, b, c*** matters, i.e., $\sigma_a \otimes \sigma_b \otimes \sigma_c \neq \sigma_a \otimes \sigma_c \otimes \sigma_b$!!! (e.g., $2 = 1 \otimes 2 \otimes 0 \neq 1 \otimes 0 \otimes 2 = 0$).
The operation $\otimes$ is ***not*** commutative.

The associativity of the operation $\otimes$ is easily proved by checking all of the possible cases.

Since this is so, we can calculate Π_{n-1} using pairs of σ-s:

$$\Pi_{n-1} = \sigma_{n-1} \otimes \sigma_{n-2} \otimes \sigma_{n-3} \otimes \sigma_{n-4} \otimes \ldots \otimes \sigma_3 \otimes \sigma_2 \otimes \sigma_1 \otimes \sigma_0 =$$
$$= (\sigma_{n-1} \otimes \sigma_{n-2}) \otimes (\sigma_{n-3} \otimes \sigma_{n-4}) \otimes \ldots \otimes (\sigma_3 \otimes \sigma_2) \otimes (\sigma_1 \otimes \sigma_0)$$

Let us denote $\sigma'_{i/2} = (\sigma_{i+1} \otimes \sigma_i)$. So now we have:
$\Pi_{n-1} = \sigma'_{n/2-1} \otimes \sigma'_{n/2-2} \otimes \sigma'_{n/2-3} \otimes \ldots \otimes \sigma'_2 \otimes \sigma'_1 \otimes \sigma'_0$

The following circuit uses the associativity to produce all Π_i-s from the σ_i-s recursively:

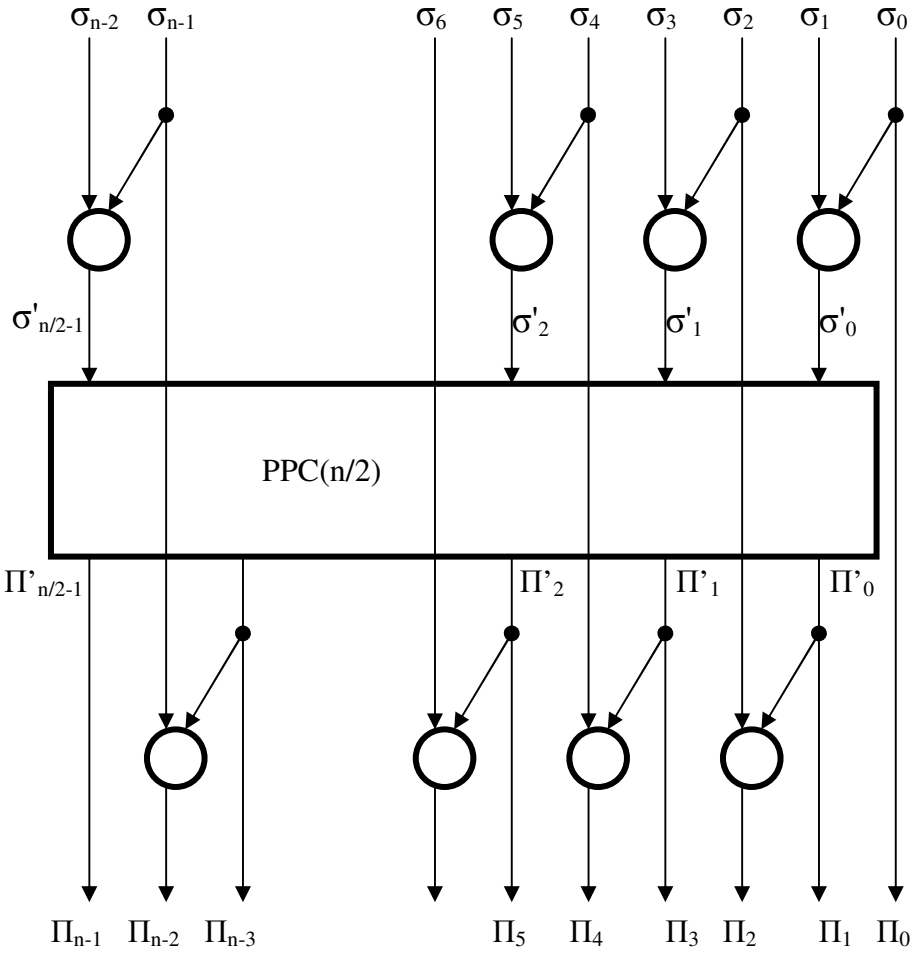


Figure 3.18 – Recursive PPC: PPC(n) built of PPC(n/2)

This circuit calculates the right results since we can see that

$\Pi_0 = \sigma_0$
$\Pi_1 = \sigma'_0 = \sigma_1 \otimes \sigma_0$
$\Pi_2 = \sigma_2 \otimes \Pi_1 = \sigma_2 \otimes \sigma_1 \otimes \sigma_0$
$\Pi_3 = \sigma'_1 \otimes \sigma'_0 = (\sigma_3 \otimes \sigma_2) \otimes (\sigma_1 \otimes \sigma_0) = \sigma_3 \otimes \sigma_2 \otimes \sigma_1 \otimes \sigma_0$

etc., and that we can also use the relation: $\Pi_i = \sigma_i \otimes \Pi_{i-1}$.

We immediately see that the delay follows the equation: D(n)= 2T + D(n/2) where T is the delay of the device performing the operation $\otimes$. Thus, we conclude that the delay is logarithmic: D(n) $\approx 2 \cdot T \cdot \lg n$.

We also see that adding n/2 inputs caused addition of (n-1) devices which is about twice of the number of inputs we add. This means that the cost is given by C(n) ≈ 2n.
This is so since C(n) ≈ n + C(n/2) = 2(n/2)+C(n/2) ≈ 2(n/2) + 2(n/4) + C(n/4) ≈
≈ 2[(n/2) + (n/4) + (n/8) + … + 2 + 1] = 2(n-1) ≈ 2n.
[The exact results are: D(n)= $(2 \cdot \lg n - 1)T$ and C(n)=2n-2-lg(n)]

Thus, the PPC version of the carry look ahead has a logarithmic delay and a linear cost.

Calculating the C_i-s from the Π_i-s is easy:

C_i="0" if Π_i=0, and C_i="1" if Π_i=2. As mentioned above, Π_i can never be 1, since σ_0 can be 0 or 2. (I.e., if we have Π_i=1, we "look" to the right and get the value of 0 or 2 from Π_{i-1}. Even if we get 1 and therefore should continue looking further to the right, eventually we reach σ_0, that by definition can be only 0 or 2).

Figure 3.19 below shows the entire recursion depth of a PPC(16).

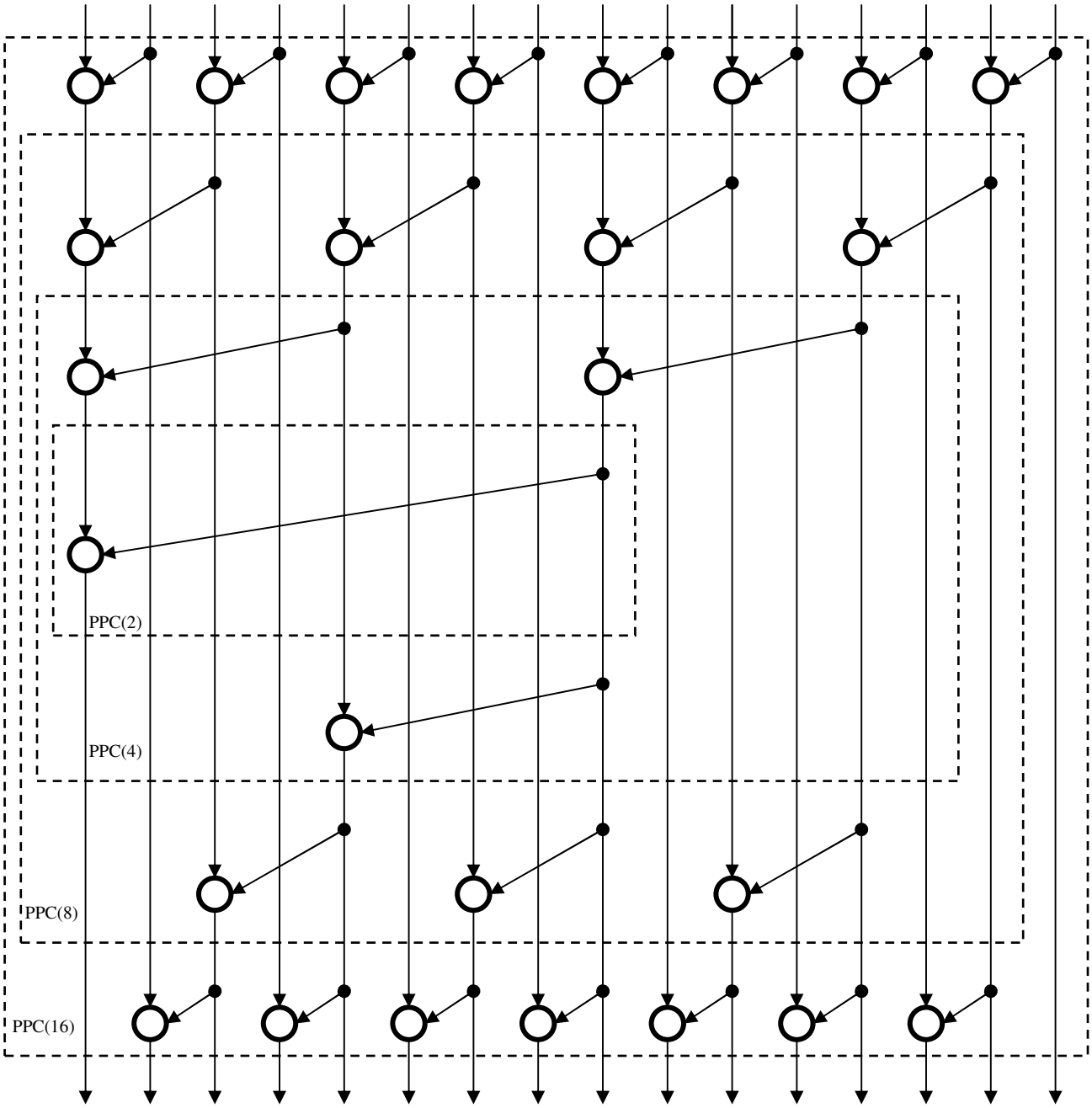


Figure 3.19 – The entire recursion depth in a PPC(16)

The implementation of the $\otimes$ operation is depicted in Figure 3.20 below.

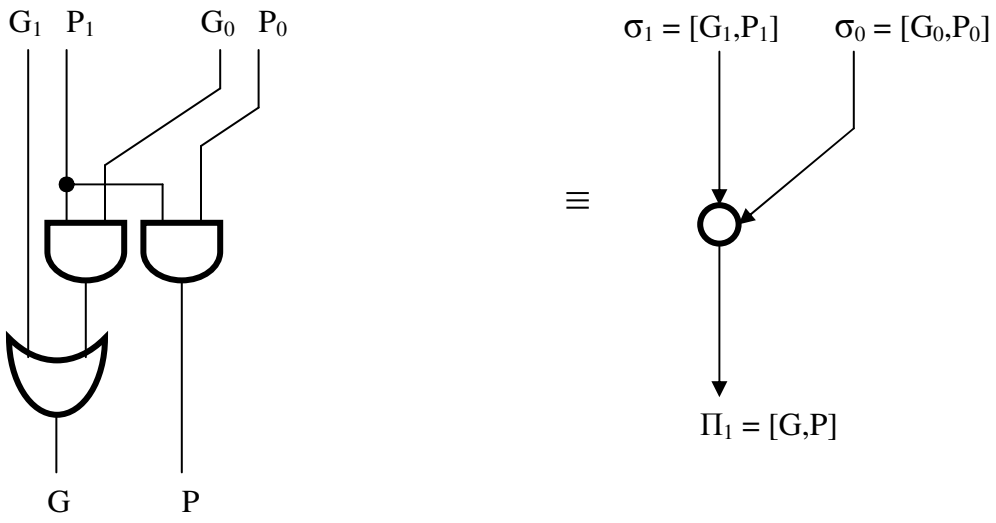


Figure 3.20 – The implementation of the $\otimes$ operation

Let us verify the correctness of the implementation:

- When σ_1=2 (i.e., G_1=1), we have Π_1=2 (i.e., G=1) as desired. (Note that in this case P_1 must be 0 so P is also 0).
- When σ_1=0 (i.e., G_1=P_1=0), we have Π_1=0 as desired.
- When σ_1=1 (i.e., G_1=0, P_1=1), we have Π_1= σ_0 as required (since in this case we have G=G_0 and P=P_0).