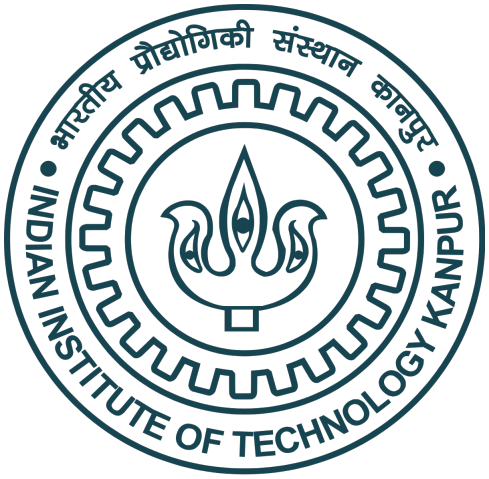
CS220A: Computer Organization



Assignment 4 Report

Aditya Tanwar

200057

March, 2022

Question 1

Construct a finite state machine for the *3-bit Gray code counter*. The counter is to be designed with one input terminal (which receives pulse signals) and one output terminal. It should be capable of counting in the Gray system up to 7 and producing an output pulse for every 8 input pulses. After the count 7 is reached, the next pulse will reset the counter to its initial state, i.e., to a count of zero.

The state transitions will be: $S_0 : 000,\ S_1 : 001,\ S_2 : 011,\ S_3 : 010,\ S_4 : 110,\ S_5 : 111,\ S_6 : 101,\ S_7 : 100 \longrightarrow S_0$. Output will be high only from $S_7 \longrightarrow S_0$.

(a) Construct the state assignment, the state diagram, state table, transition and output table, excitation table.

(b) Then build the K-map for the same and design the circuitry or logic diagram.

(c) Write the Verilog code module to implement the 3-bit Gray code counter.

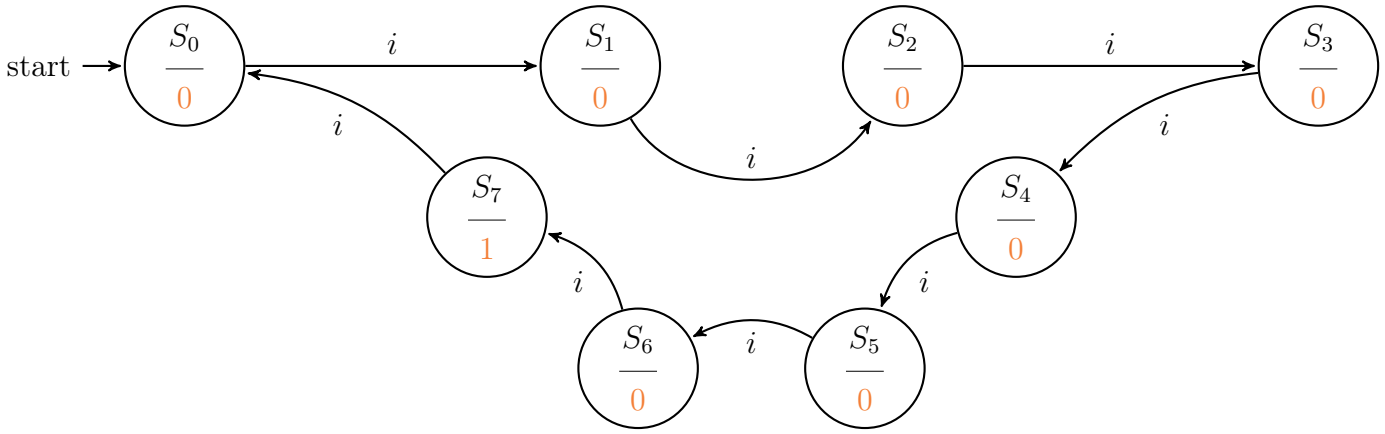
(d) Now, add a test bench to test the 3-bit Gray code counter.

State Assignment: We use gray encoding to encode the state variables $[S_0 : S_7]$, and use *3* state variables (x_1, x_2, x_3) to keep track of the present state of the FSM. The states represented by, and their encoding is the same as that given in the question. The most significant bit of the *3-bit* number used to represent the present state is denoted by x_1, and the least significant bit of it is denoted by x_3.
Further, the positive edge *(posedge)* of the clock is used as the input and this momentary excitation is denoted by i in the state diagram.

The assignment has been represented in a tabular form below-

State	x_1	x_2	x_3
S_0	0	0	0
S_1	0	0	1
S_2	0	1	1
S_3	0	1	0
S_4	1	1	0
S_5	1	1	1
S_6	1	0	1
S_7	1	0	0

State Diagram : The state diagram of the finite state machine (FSM) used for this problem is-



Since, it is a *Moore Machine*, the output is omitted from the transitions and rather, it is denoted in the states themselves, under the horizontal bar. The output has also been coloured. The text above the bar denotes the state.

Note: The input field has been omitted in all the tables since it is simply the positive edge of the clock used.

Note: On reset, the state of the *FSM* is reset to S_0, regardless of the current state. It has thus, been skipped from the tables since behaviour corresponding to it is trivial.

State Table : The state table corresponding to the *FSM* is given below:

P.S.	*N.S./ Output*
S_0	S_1/ 0
S_1	S_2/ 0
S_2	S_3/ 0
S_3	S_4/ 0
S_4	S_5/ 0
S_5	S_6/ 0
S_6	S_7/ 0
S_7	S_0/ 1

Karnaugh Map : A *Karnaugh Map* is made for each of (X_1, X_2, X_3, o), where o denotes the output of the FSM. X_i is used to denote $x_i(t+1)$, while x_i is used to denote $x_i(t)$.

x_1 \ $x_2\,x_3$	00	01	11	10
0	0	0	0	0
1	0	0	1	0

$o = x_1 \wedge x_2 \wedge x_3$

x_1 \ $x_2\,x_3$	00	01	11	10
0	0	0	0	1
1	0	1	1	1

$X_1 = (x_1 \wedge x_3) \vee (x_2 \wedge \neg\, x_3)$

x_1 \ $x_2\,x_3$	00	01	11	10
0	0	1	1	1
1	0	0	0	1

$X_2 = (\neg\, x_1 \wedge x_3) \vee (x_2 \wedge \neg x_3)$

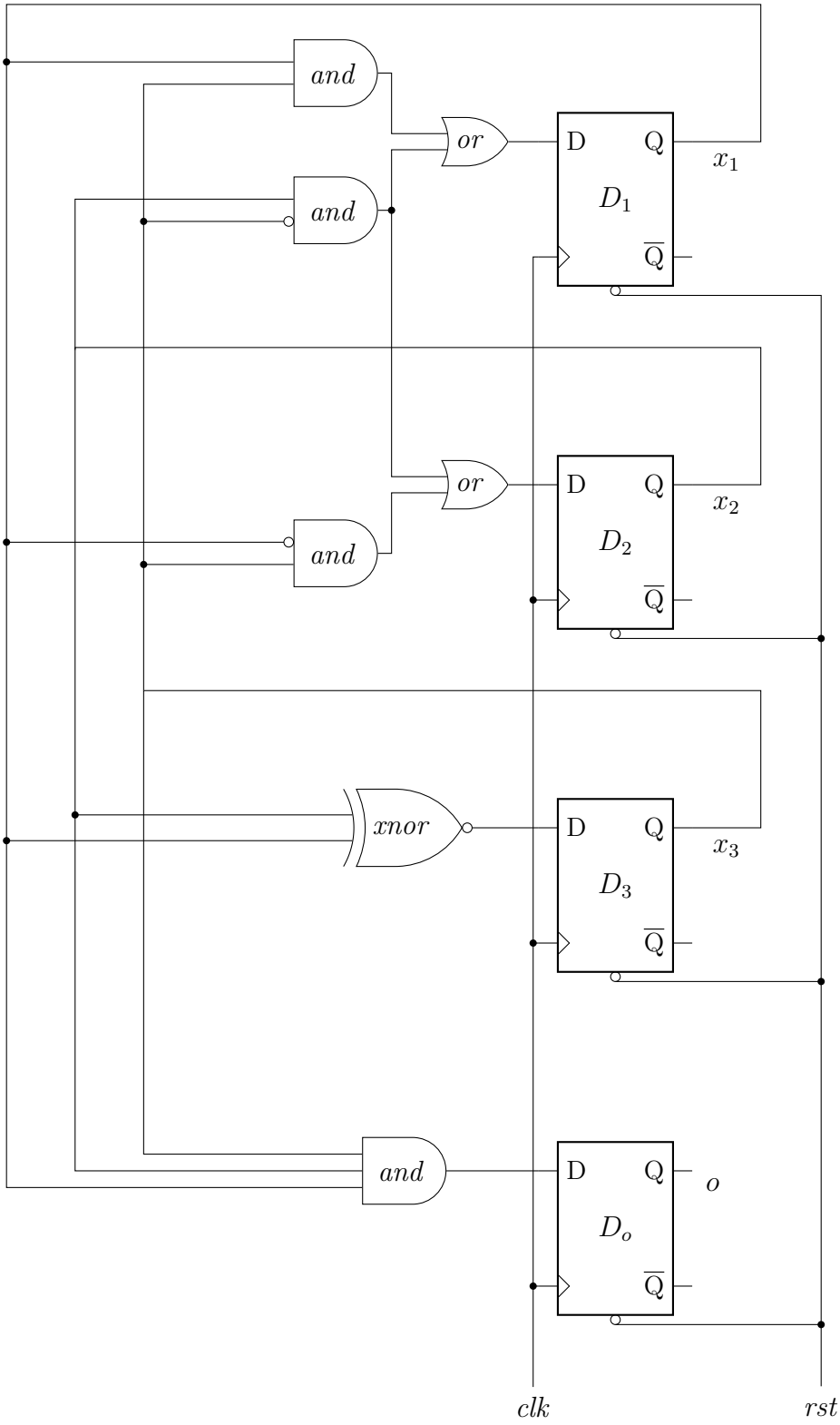
x_1 \ $x_2\,x_3$	00	01	11	10
0	1	1	0	0
1	0	0	1	1

$X_3 = (\neg\, x_1 \wedge \neg\, x_2) \vee (x_1 \wedge x_2)$
$= \neg(x_1 \oplus x_2)$

Transition and Output Table : The transition and output table corresponding to the *FSM* is given below:

P.S.	*N.S.*	*Output*
S_0	S_1	0
S_1	S_2	0
S_2	S_3	0
S_3	S_4	0
S_4	S_5	0
S_5	S_6	0
S_6	S_7	0
S_7	S_0	1

Circuit Diagram : The circuit diagram corresponding to the *FSM* has been drawn below:



Note: The inputs to the *Flip-flops* are the next states/ next outputs, while the output is the present state/ current output.

Excitation Table : The excitation table corresponding to the *FSM* is given below:

x_1	x_2	x_3	*P.S.*	*N.S.*	X_1	X_2	X_3	o
0	0	0	S_0	S_1	0	0	1	0
0	0	1	S_1	S_2	0	1	1	0
0	1	1	S_2	S_3	0	1	0	0
0	1	0	S_3	S_4	1	1	0	0
1	1	0	S_4	S_5	1	1	1	0
1	1	1	S_5	S_6	1	0	1	0
1	0	1	S_6	S_7	1	0	0	0
1	0	0	S_7	S_0	0	0	0	1

Question 2

Write a detailed description of *eight-bit adder/subtractor* to add/subtract two eight-bit two's complement numbers and its working with the proper circuit diagram in a PDF file. Then write the Verilog code module to implement an *eight-bit adder/subtractor*. It will be implemented in two modules. First module implements a *one-bit adder/subtractor* with four inputs a, b, c_{in}, and *opcode*, and two outputs *sum* and *carry*. For the addition operation the input *opcode* will be 0 for addition and 1 for subtraction operation. The top module implements the *eight-bit adder/subtractor* using the *one-bit adder/subtractor* module. There will be two inputs for this module the two input numbers and the *opcode* and produces the *sum*, the *carry out*, and whether there is an *overflow* as the outputs. Now, add a test bench to test the *eight-bit adder/subtractor*. Your test bench must have fifteen different inputs. Put five-time unit delay between consecutive inputs.

We provide two different approaches to implement it, one which we came up with, and the other, which is the traditional one. Both of them have the same implementational cost, but the interpretation and functioning of the *1-bit* $\mathcal{AS}$ at the higher level is easier to explain, since it intends to carry out subtraction in much the same way that we used to do in school, while the functioning of the traditional approach seems to be a bit more technical.

Personal Approach

Conventions : We use the following shorthands for variables-

* a_i/ b_i := The i^{th} bit of the *8-bit* input numbers that are being operated on, in the *8-bit adder/ subtractor*.
* a/ b := The input bits in the *1-bit adder/ subtractor*.
* o := The code for the operation in either of the modules.
* r := The result of the operation on a and b.
* r_i := The i^{th} bit of the result of the operation on *8-bit input numbers*.
* c := The resultant carry/borrow of the operation on a and b.
* c_i := The i^{th} bit of the resultant carry/borrow of the operation on *8-bit input numbers*.
* c^{in} := The input carry/borrow of the operation on a and b.
* f := Overflow.
* It is assumed that when o signifies subtraction operation, then $r = a - b$ has to be calculated by the adder/subtractor.
* $\mathcal{AS}$:= Acronym for adder/subtractor.
* In the subtraction operation, c (and similar variables) is (are) called the borrow, and in the addition operation c (and similar variables) is (are) called the carry.

1-bit $\mathcal{AS}$: We break the analysis of the *1-bit* $\mathcal{AS}$ into two components, one, when it is acting as an adder ($o = 0$), and the other, when it is acting as an subtractor ($o = 1$).

$o = 0$: When it is acting as an adder, we already know the results and the truth tables for r and c. But, we have mentioned them here as well for the sake of completeness,

$$r = (a \oplus b \oplus c^{in})$$

$$c = (a \wedge b) \vee (b \wedge c^{in}) \vee (c^{in} \wedge a)$$

$o = 1$ First of all, we need to observe that the function (behaviour) of c^{in} *(the borrow)* is the same as that of b.
Now, the current digit needs a borrow from the left, if and only if, $a < b + c^{in}$ (where $+$ is the normal addition operation we use in general. Again, it can be split into two cases, one where $a = 0$ and at least one of b or c^{in} is 1, and the other where $a = 1$ and both of b and c^{in} are also 1. Framing the argument into boolean expressions, we get,

$$c = (\neg a \wedge (b \vee c))_{[a=0]} \ \vee (a \wedge b \wedge c)_{[a=1]}$$

Subscripts have been used to denote case splitting.

Since the function (behaviour) of c^{in} and b is the same, we know that the result will be $r = a - b - c^{in}$, if $a \geq b + c^{in}$, and if $a < b + c^{in}$, then the current digit would require a borrow from the more significant bit, in which case, the result will be given by $r = 10_2 + a - b - c^{in}$ where 10_2 was borrowed.
Again, this can be split into two cases, one where $a = 0$, yielding $r = 10_2 + 0 - b - c^{in} = 1 - b + 1 - c^{in} \implies r = \neg b \vee \neg c$, and the other, where $a = 1$, yielding $b = 1$, $c = 1$ ($\because a < b + c$) and so $r = 10_2 + 1 - 1 - 1 = 1_2$. But, forming a concrete expression from these is a bit cumbersome and difficult to observe. We, thus, turn to *K-maps* for a simplified expression. Regardless, these can be used as small explanations/ verifications or seen as interpretations.

a \ $b\ c_{in}$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

By the alternating pattern, we observe that,

$$r = a \oplus b \oplus c^{in}$$

a \ $b\ c_{in}$	00	01	11	10
0	0	1	1	1
1	0	0	1	0

$$c = (\neg a \wedge c^{in}) \vee (\neg a \wedge b) \vee (b \wedge c^{in})$$

We shall prefer to use this expression instead of the one derived above for easier circuital implementation.

And now, we are in a position to combine these two cases together. Firstly, we conclude that

$$r = a \oplus b \oplus c^{in}$$

since the expression holds irrespective of whether the operation is addition or subtraction.
Then, we move on to the expression of c,

$$c = (\neg o \wedge ((a \wedge b) \vee (b \wedge c^{in}) \vee (c^{in} \wedge a)) \vee (o \wedge ((\neg a \wedge c^{in}) \vee (\neg a \wedge b) \vee (b \wedge c^{in})))$$

Clearly, this is a fairly long and complicated expression, and we would prefer to have a shorter version. Our first simplification comes from the observation that $(b \wedge c^{in})$ is *and*-ed with both o and $\neg o$, and the result is *or*-ed. Thus, we can pull it out, in a manner opposite to that of distributive laws.

$$c = (\neg o \wedge ((a \wedge b) \vee (a \wedge c^{in}))) \vee (o \wedge ((\neg a \wedge b) \vee (\neg a \wedge c^{in}))) \vee (b \wedge c^{in})$$

Our second observation requires some jumps, but can be shown with rigour too. Observe how b, for instance, is *and*-ed once with $(\neg o \wedge a)$, once with $(o \wedge \neg a)$, and then these two results are *or*-ed, essentially, giving rise to $(b \wedge (o \oplus a))$, which will be *or*-ed with the rest of the expressions. Similarly, with c^{in}. Thus, our final expression for c is,

$$c = (b \wedge c^{in}) \vee (b \wedge (o \oplus a)) \vee (c^{in} \wedge (o \oplus a))$$

A short proof for the above simplification is provided below-

$$\begin{aligned}(\neg o \wedge ((a \wedge b) \vee \phi)) \vee (o \wedge ((\neg a \wedge b) \vee \psi)) \vee \chi &= (\neg o \wedge a \wedge b) \vee (\neg o \wedge \phi) \vee (o \wedge \neg a \wedge b) \vee (o \wedge \psi) \vee \chi \\ &= (\neg o \wedge a \wedge b) \vee (o \wedge \neg a \wedge b) \vee \dots \\ &= (((\neg o \wedge a) \vee (o \wedge \neg a)) \wedge b) \vee \dots \\ &= ((o \oplus a) \wedge b) \vee \dots\end{aligned}$$

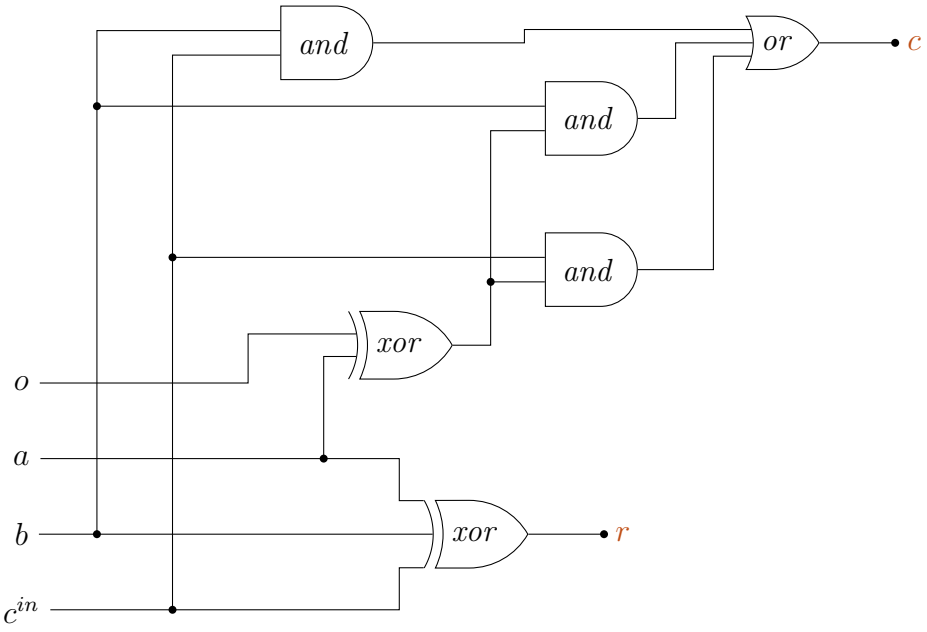
Concluding, our *1-bit* $\mathcal{AS}$ takes 4 bits as input (a, b, c^{in}, o), and outputs 2 bits (r, c).
Note: The calculated c does not have the same interpretation while adding and subtracting. While adding, it holds the carry-out, and while subtracting, it holds the borrow.

8-bit $\mathcal{AS}$: This would take in 2 *8-bit* numbers (represented in 2's complement), and an opcode as input, denoted by (a, b, o), and output an *8-bit* result, the resultant carry-out/ borrow, and report whether there was an overflow, which shall be denoted by (r, c, f).
It uses 8 *1-bit* $\mathcal{AS}$, one corresponding to each bit of the result. The *1-bit* $\mathcal{AS}$'s have been designed in such a way that they do not need to be configured differently for any bit of the input. For the calculation of r_i, we simply need to have a_i, b_i, c_i^{in} ready, along with the opcode. However, for checking overflow, we require operations which are beyond the scope of a *1-bit* $\mathcal{AS}$, namely, we need to *XOR* ($\oplus$) the last two carries.
Concluding, r and c are calculated as dictated in the *1-bit* $\mathcal{AS}$ and $f = c_7 \oplus c_8$.

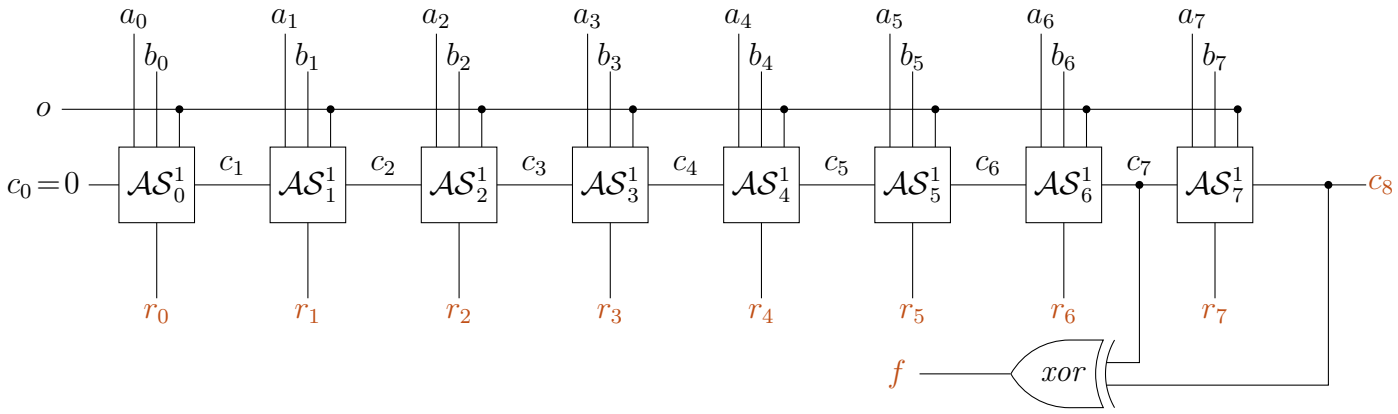
Truth Tables : *K-maps* have not been drawn much in this approach because most of the work for the boolean expressions has been derivational and/or observational because these inferences allow us to notice possible instances of *XOR* ($\oplus$) in an expression, which would otherwise be difficult to notice in a *K-map*.
For the sake of completeness, we have included truth tables for r and c that are calculated by the *1-bit* $\mathcal{AS}$. The input and output have been colored differently for better understanding. The output has further been colored differently for addition ($o = 0$) and subtraction ($o = 1$) for the same reasons.

a	b	c^{in}	r	carry (c)	r	borrow (c)
0	0	0	0	0	0	0
0	0	1	1	0	1	1
0	1	0	1	0	1	1
0	1	1	0	1	0	1
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	1	0	0	1	0	0
1	1	1	1	1	1	1

Circuit Diagrams : Firstly, we draw a detailed diagram of our *1-bit $\mathcal{AS}$*, and then abstract it to use in the *8-bit $\mathcal{AS}$*.



Now, we assume an element of the kind *1-bit $\mathcal{AS}$*, and draw the circuit diagram for an *8-bit $\mathcal{AS}$*. The label $\mathcal{AS}_i^j$ on an adder-subtractor in the diagram denotes that it handles the i^{th} part of the input, and the size of the concerned part is j bits. In our case specifically, i is used to represent the i^{th} bit of the result, while $j = 1$ represents that the adder-subtractors used, are *1-bit $\mathcal{AS}$*.



This constitutes the circuit for our implementation of an *8-bit $\mathcal{AS}$*. All the intermediate carries/ borrows are used as the carry-in/borrow-in for the next $\mathcal{AS}$, and opcode ($= o$) is used in all the $\mathcal{AS}$'s.
Outputs of both the $\mathcal{AS}$'s have been colored.

Traditional Approach

Concept: The concept used is that the *2's complement* of a number is the *1's complement* of it, added with 1. The *1's complement* is gained by flipping each bit of the input.
Thus, when adding two numbers, the $\mathcal{AS}$ acts like a normal adder, whereas while subtracting two numbers, say $a - b$, it converts $-b$ to the *2's complement* form of b, and essentially intends to calculate $a + (-b)$.

1-bit $\mathcal{AS}$: In case of addition ($o = 0$), nothing changes for b, while in the case of subtraction ($o = 1$), we need to flip b, and then calculate $a + (-b)$ in the adder.
Another way to derive the expression for the resultant b, denoted by b_r, is:

o \ b	0	1
0	1	0
1	0	1

By the alternating pattern, we observe that,

$$b_r = b \oplus o$$

Since we only need to add a with b_r, we can use the results obtained from a *1-bit adder*, and directly write $r = a \oplus b_r \oplus c^{in} = a \oplus b \oplus o \oplus c^{in}$ and $= (a \wedge b_r) \vee (b_r \wedge c^{in}) \vee (c^{in} \wedge a)$.

8-bit $\mathcal{AS}$: It uses 8 *1-bit* $\mathcal{AS}$, one corresponding to each bit of the result. The carry-in of the first *1-bit* $\mathcal{AS}$ is wired to the opcode, since in the case of subtraction, we need to add *1* to the *1's complement*. For the calculation of r_i, we simply need to have a_i, b_i, c_i^{in} ready, along with the opcode. However, for checking overflow, we require operations which are beyond the scope of a *1-bit* $\mathcal{AS}$ namely, we need to *XOR* ($\oplus$) the last two carries.
Concluding, r and c are calculated as dictated in the *1-bit* $\mathcal{AS}$ and $f = c_7 \oplus c_8$.

Circuit Diagrams : The circuit diagrams are much the same as that in the previous approach with a single change in each of *1-bit* $\mathcal{AS}$ and *8-bit* $\mathcal{AS}$.
In the *1-bit* $\mathcal{AS}$, we simply exchange a and b, and the rest of the structure remains the same. In the *8-bit* $\mathcal{AS}$, we configure the carry-in for the first *1-bit* $\mathcal{AS}$ to equal opcode, instead of being 0 always.

Truth Tables : These have been omitted in the interest of space. The table remains the same for r, but has minimal changes for c.
