MODULE 2: COMBINATIONAL CIRCUITS

COMBINATIONAL CIRCUITS		
SL. NO.	**CATEGORY**	**CIRCUITS**
1.	**LOGIC GATES**	
2.	**CODE CONVERSION**	
3.	**ADDER**	
4.	**SUBTRACTOR**	
5.	**MULTIPLIER**	
6.	**COMPARATOR**	
7.	**DECODER**	
8.	**ENCODER**	
9.	**MULTIPLEXER**	
10.	**DEMULTIPLEXER**	

CATEGORY-1: LOGIC GATES

PROBLEM 1.1: AND Gate

STRUCTURAL MODEL:	DATA FLOW MODEL:	BEHAVIORAL MODEL:

module ANDGATE (x, y, z);
  input x, y;
  output z;
  and G1(z, x, y);
endmodule
	module ANDGATE (x, y, z);
	  input x,cy;
	  output wire z;
	  assign z = x & y;
	endmodule
		module ANDGATE (x, y, z);
		  input x, y;
		  output reg z;
		  always @(x, y)
		  z=x & y;
		endmodule

PROBLEM 1.2: OR Gate

STRUCTURAL MODEL:	DATA FLOW MODEL:	BEHAVIORAL MODEL:

module ORGATE (x, y, z);
  input x, y;
  output z;
  or G1(z, x, y);
endmodule
	module ORGATE (x, y, z);
	  input x, y;
	  output wire z;
	  assign z = x | y;
	endmodule
		module ORGATE (x, y, z);
		  input x, y;
		  output reg z;
		  always @(x, y)
		  z=x | y;
		endmodule

PROBLEM 1.3: NOT Gate

STRUCTURAL MODEL:	DATA FLOW MODEL:	BEHAVIORAL MODEL:

module NOTGATE (x, z);
  input x;
  output z;
  not G1(z, x);
endmodule
	module NOTGATE (x, z);
	  input x, y;
	  output wire z;
	  assign z = ~x;
	endmodule
		module NOTGATE (x, z);
		  input x, y;
		  output reg z;
		  always @(x, y)
		  z = ~x;
		endmodule

PROBLEM 1.4: NAND Gate

STRUCTURAL MODEL:	DATA FLOW MODEL:	BEHAVIORAL MODEL:

module NANDGATE (x, y, z);
  input x, y;
  output z;
  nand G1(z, x, y);
endmodule
	module NANDGATE (x, y, z);
	  input x, y;
	  output wire z;
	  assign z = ~(x & y);
	endmodule
		module NANDGATE (x, y, z);
		  input x, y;
		  output reg z;
		  always @(x, y)
		  z = ~(x & y);
		endmodule

PROBLEM 1.5: NOR Gate

STRUCTURAL MODEL:	DATA FLOW MODEL:	BEHAVIORAL MODEL:

module NORGATE (x, y, z);
  input x, y;
  output z;
  nor G1(z, x, y);
endmodule
	module NORGATE (x, y, z);
	  input x, y;
	  output wire z;
	  assign z = ~(x | y);
	endmodule
		module ORGATE (x, y, z);
		  input x, y;
		  output reg z;
		  always @(x, y)
		  z = ~(x | y);
		endmodule

PROBLEM 1.6: XOR Gate

STRUCTURAL MODEL:	DATA FLOW MODEL:	BEHAVIORAL MODEL:

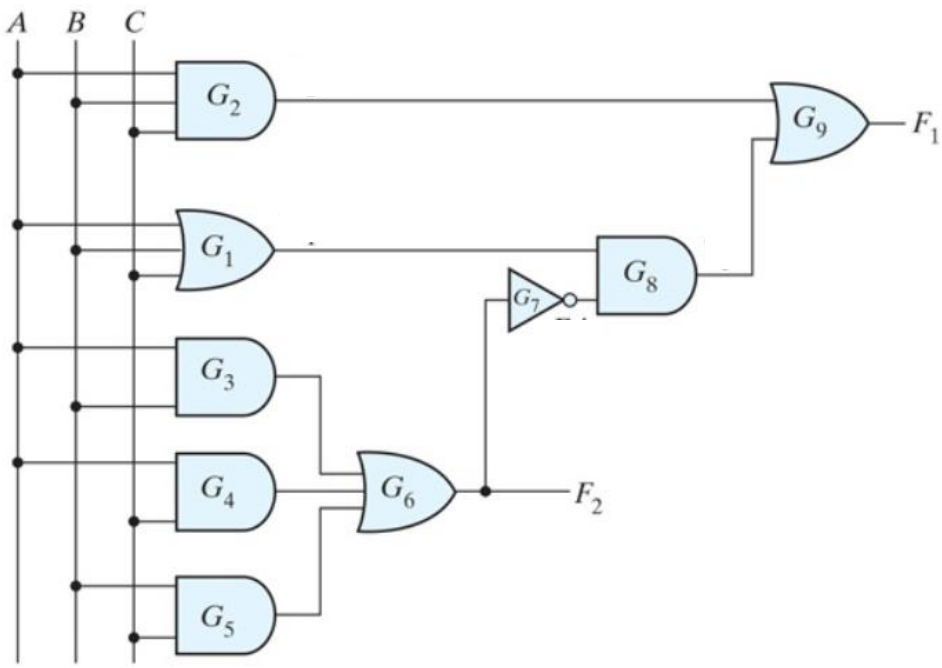
module XORGATE (x, y, z);
  input x, y;
  output z;
  xor G1(z, x, y);
endmodule
	module XORGATE (x, y, z);
	  input x, y;
	  output wire z;
	  assign z = (x ^ y);
	endmodule
		module XORGATE (x, y, z);
		  input x, y;
		  output reg z;
		  always @(x, y)
		  z = (x ^ y);
		endmodule

PROBLEM 1.7: X-NOR Gate

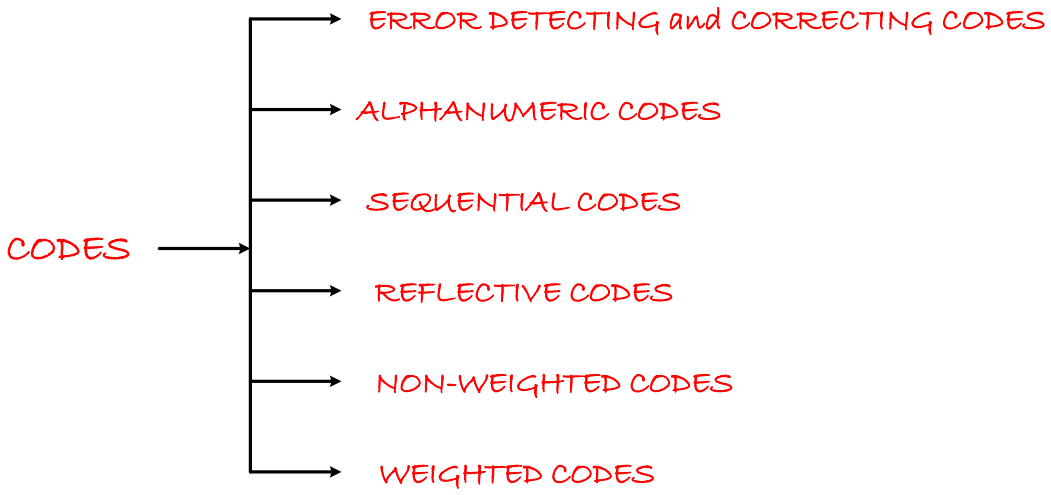
STRUCTURAL MODEL:	DATA FLOW MODEL:	BEHAVIORAL MODEL:

module XNORGATE (x, y, z);
  input x, y;
  output z;
  xnor G1(z, x, y);
endmodule
	module XNORGATE (x, y, z);
	  input x, y;
	  output wire z;
	  assign z = ~(x ^ y);
	endmodule
		module XNORGATE (x, y, z);
		  input x, y;
		  output reg z;
		  always @(x, y)
		  z = ~(x ^ y);
		endmodule

PROBLEM 1.8



CATEGORY-2: CODE CONVERSION



WEIGHTED CODE: Each position of number represents specific weight.
Example: Binary, 8421, 2421
NON-WEIGHTED CODE: No positional weight.
Example: Excess-3, Gray
REFLECTIVE CODE: Self-Complementing Code (Code of 9 = ***Complement*** of Code of 0; 8 = 1 and so on).
Example: Excess-3, 2421
REFLECTIVE CODE: Each succeeding code is 1 binary number greater than preceding code.
Example: Excess-3, 8421
ALPHANUMERIC CODE: Example: ASCII Code
ERROR DETECTING & CORRECTING CODE: Example: Hamming Code

Binary Coded Decimal (BCD)

In this code, a 4-bit binary number represents each decimal digit.

DECIMAL	Binary Coded Decimal (BCD)			
	X3(8)	X2(4)	X1(2)	X0(1)
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	X	X	X	X
11	X	X	X	X
12	X	X	X	X
13	X	X	X	X
14	X	X	X	X
15	X	X	X	X

2421 CODE

DECIMAL DIGIT	SET I				SET II			
	2	4	2	1	2	4	2	1
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	0
3	0	0	1	1	0	0	1	1
4	0	1	0	0	0	1	0	0
5	0	1	0	1	1	0	1	1
6	0	1	1	0	1	1	0	0
7	0	1	1	1	1	1	0	1
8	1	1	1	0	1	1	1	0
9	1	1	1	1	1	1	1	1
					SELF COMPLEMENTING			

OTHER BCD CODE: (MAKE the TABLE BY YOURSELF)

1) 7 4 2 1
2) 5 4 2 1
3) 3 3 2 1
4) 8 4 $\overline{2}$ $\overline{1}$
5) 7 4 $\overline{2}$ $\overline{1}$

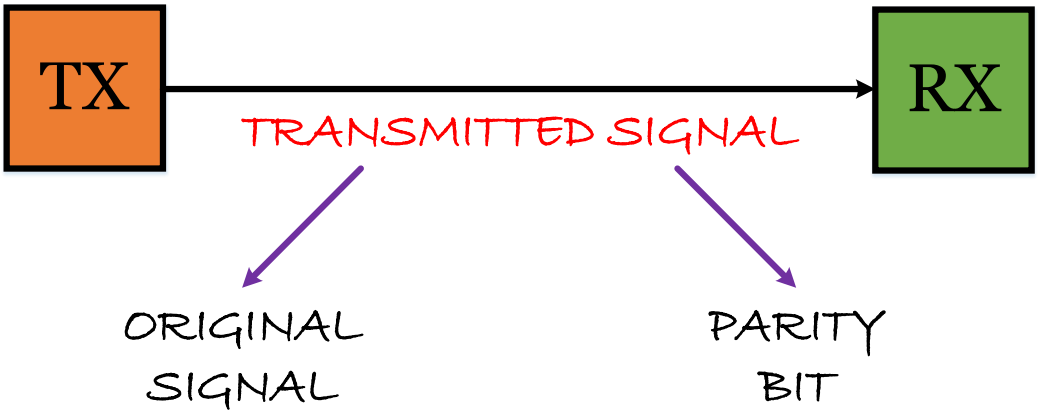
BCD to Excess-3

DECIMAL	INPUT (BCD)				OUTPUT (Excess-3)			
	A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

BINARY to GRAY CODE

DECIMAL	INPUT (BINARY)				OUTPUT (GRAY CODE)			
	A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

BINARY CODE	B_n	B_{n-1}	B_{n-2}	…	…	B_2	B_1	B_0
GRAY CODE	G_n	G_{n-1}	G_{n-2}	…	…	G_2	G_1	G_0
$G_n = B_n$								
$G_{n-1} = B_{n-1} \oplus B_n$								
$G_{n-2} = B_{n-2} \oplus B_{n-3}$								
… … … …								
$G_2 = B_2 \oplus B_3$								
$G_1 = B_1 \oplus B_2$								
$G_0 = B_0 \oplus B_1$								



EVEN PARITY: Transmitted signal contains ***EVEN*** number of 1s		
TYPE	**ORIGINAL SIGNAL**	**PARITY BIT**
EVEN PARITY	Contains ***EVEN*** number of 1s	**0**
	Contains ***ODD*** number of 1s	**1**
ODD PARITY: Transmitted signal contains ***ODD*** number of 1s		
TYPE	**ORIGINAL SIGNAL**	**PARITY BIT**
ODD PARITY	Contains ***EVEN*** number of 1s	**1**
	Contains ***ODD*** number of 1s	**0**

7-BIT HAMMING CODE

Hamming Code consists of two parts: i) ***DATA*** bits ii) ***PARITY*** bits

Position of ***PARITY*** bits in HAMMING CODE = 2^n; where $n = \{0, 1, 2, 3 \ldots\}$

7	6	5	4	3	2	1
D_4	D_3	D_2	P_3	D_1	P_2	P_1

BIT	TYPE	PARITY BIT	CONDITION
P_1	EVEN Parity	$P_1 = 0$	When $D_3D_5D_7$ contains ***EVEN*** no. of 1s
		$P_1 = 1$	When $D_3D_5D_7$ contains ***ODD*** no. of 1s
	ODD Parity	$P_1 = 1$	When $D_3D_5D_7$ contains ***EVEN*** no. of 1s
		$P_1 = 0$	When $D_3D_5D_7$ contains ***ODD*** no. of 1s
P_2	EVEN Parity	$P_2 = 0$	When $D_3D_6D_7$ contains ***EVEN*** no. of 1s
		$P_2 = 1$	When $D_3D_6D_7$ contains ***ODD*** no. of 1s
	ODD Parity	$P_2 = 1$	When $D_3D_6D_7$ contains ***EVEN*** no. of 1s
		$P_2 = 0$	When $D_3D_6D_7$ contains ***ODD*** no. of 1s
P_3	EVEN Parity	$P_3 = 0$	When $D_5D_6D_7$ contains ***EVEN*** no. of 1s
		$P_3 = 1$	When $D_5D_6D_7$ contains ***ODD*** no. of 1s
	ODD Parity	$P_3 = 1$	When $D_5D_6D_7$ contains ***EVEN*** no. of 1s
		$P_3 = 0$	When $D_5D_6D_7$ contains ***ODD*** no. of 1s

ASCII CODE

0	00	NUL
1	01	SOH
2	02	STX
3	03	ETX
4	04	EOT
5	05	ENQ
6	06	ACK
7	07	BEL
8	08	BS
9	09	HT
10	0A	LF
11	0B	VT
12	0C	FF
13	0D	CR
14	0E	SO
15	0F	SI
16	10	DLE
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN

25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
30	1E	RS
31	1F	US
32	20	space
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2

51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L

77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f

103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	DEL

128	80	□
129	81	□
130	82	□
131	83	□
132	84	□
133	85	□
134	86	□
135	87	□
136	88	□
137	89	□
138	8A	□
139	8B	□
140	8C	□
141	8D	□
142	8E	□
143	8F	□
144	90	□
145	91	□
146	92	□
147	93	□
148	94	□
149	95	□
150	96	□
151	97	□
152	98	□

153	99	□
154	9A	□
155	9B	□
156	9C	□
157	9D	□
158	9E	□
159	9F	□
160	A0	
161	A1	¡
162	A2	¢
163	A3	£
164	A4	¤
165	A5	¥
166	A6	¦
167	A7	§
168	A8	¨
169	A9	©
170	AA	ª
171	AB	«
172	AC	¬
173	AD	
174	AE	®
175	AF	¯
176	B0	°
177	B1	±
178	B2	²

179	B3	³
180	B4	´
181	B5	µ
182	B6	¶
183	B7	·
184	B8	¸
185	B9	¹
186	BA	º
187	BB	»
188	BC	¼
189	BD	½
190	BE	¾
191	BF	¿
192	C0	À
193	C1	Á
194	C2	Â
195	C3	Ã
196	C4	Ä
197	C5	Å
198	C6	Æ
199	C7	Ç
200	C8	È
201	C9	É
202	CA	Ê
203	CB	Ë
204	CC	Ì

205	CD	Í
206	CE	Î
207	CF	Ï
208	D0	Ð
209	D1	Ñ
210	D2	Ò
211	D3	Ó
212	D4	Ô
213	D5	Õ
214	D6	Ö
215	D7	×
216	D8	Ø
217	D9	Ù
218	DA	Ú
219	DB	Û
220	DC	Ü
221	DD	Ý
222	DE	Þ
223	DF	ß
224	E0	à
225	E1	á
226	E2	â
227	E3	ã
228	E4	ä
229	E5	å
230	E6	æ

231	E7	ç
232	E8	è
233	E9	é
234	EA	ê
235	EB	ë
236	EC	ì
237	ED	í
238	EE	î
239	EF	ï
240	F0	ð
241	F1	ñ
242	F2	ò
243	F3	ó
244	F4	ô
245	F5	õ
246	F6	ö
247	F7	÷
248	F8	ø
249	F9	ù
250	FA	ú
251	FB	û
252	FC	ü
253	FD	ý
254	FE	þ
255	FF	ÿ

CATEGORY-3: ADDER

PROBLEM 3.1: Half Adder (1 Bit)

Block Diagram:

Truth Table:

A	B	S	C_y
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Boolean Equations:

S = A ^ B

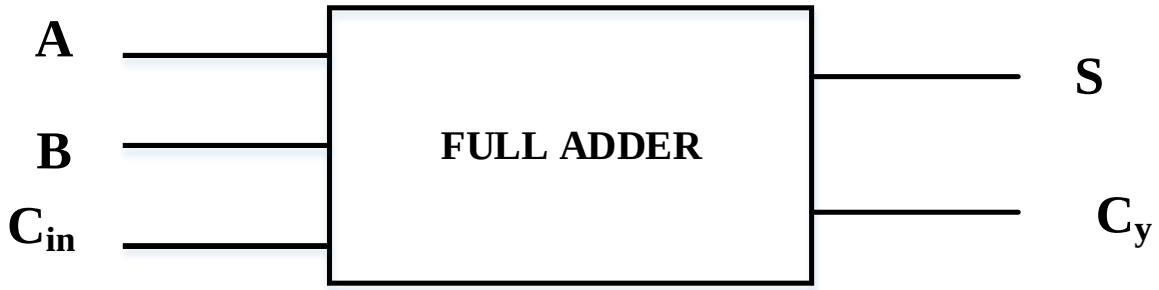
C_y = A . B

Verilog Program:

```
module Half_Adder (A, B, Sum, Carry);
    input A, B;
    output Sum, Carry;
    xor (Sum, A, B);
    and (Carry, A, B);
endmodule
```

PROBLEM 3.2: Full Adder (1 bit)

Block Diagram:



Truth Table:

A	B	C_{in}	S	C_y
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Boolean Equations:

S = A ^ B ^ C

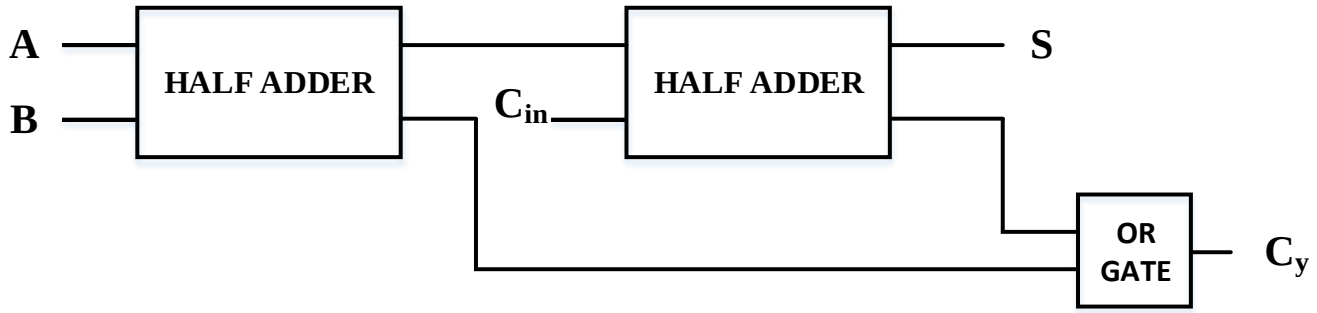
C_y= (A ^ B) C + AB

Verilog Program:

```
module Full_Adder ( A,B,Cin,S,Cy);
    input  A, B, Cin;
    output  S, Cy;
    assign S  = (A ^ B) ^ Cin;
    assign Cy = (A & B) | (B & Cin) | (Cin & A);
endmodule
```

PROBLEM 3.3: Full Adder (1 bit) using Half Adder (1 bit)

Block Diagram:



Verilog Program:

```
module Full_Adder ( A,B,Cin,S,Cy);
    input  A, B, Cin;
    output  S, Cy;
    wire t1, t2, t3;
    Half_Adder HA0 (A, B, t1, t2);
    Half_Adder HA1 (t1, Cin, S, t3);
    or (Cy, t2, t3);
endmodule

module Half_Adder (A, B, Sum, Carry);
      input A, B;
      output Sum, Carry;
      xor (Sum, A, B);
      and (Carry, A, B);
 endmodule
```

PROBLEM 3.4: Adder (Behavioral Design)

Verilog Program:

```
module Adder_4bit (S, Cout, A, B, Cin);
        input [3:0] A, B;
        input Cin;
        output [3:0] S;
        output Cout;
        assign {Cout, S} = A + B + Cin;
endmodule
```

PROBLEM 3.5: BCD ADDITION

CONDITION	CORRECT or NOT?	HOW TO CORRECT?
SUM ≤ 9 & Final Carry = 0	ANSWER (Result of Binary Addition) is ***CORRECT***	
SUM ≤ 9 & Final Carry = 1	ANSWER (Result of Binary Addition) is ***INCORRECT***	***ADD 6 (4'b0110)***
SUM > 9 & Final Carry = 0	ANSWER (Result of Binary Addition) is ***INCORRECT***	***ADD 6 (4'b0110)***

$(2)_{10}$	0	0	1	0
$(6)_{10}$	0	1	1	0
SUM	1	0	0	0

→ **SUM (= 8) < 9 & Final Carry = 0**

VERILOG PROGRAM:

```
module BCD_ADDER (A, B, Cin, Sum, Cout);
        input [3:0] A, B;
        input Cin;
        output reg [3:0] Sum;
        output reg Cout;

always@(A, B, Cin)
begin
        {Sum, Cout} = A + B + Cin;
        if ((Sum <= 4'b1001 && Cout == 1'b1) || (Sum > 4'b1001 && Cout == 1'b0))
        {Sum, Cout} = Sum + 4'b0110;
        else
        {Sum, Cout} = {Sum, Cout};
end
endmodule
```

Signed Integer: 2's Complement

-2^{N-1}	2^{N-2}	2^{N-3}	...	...	2^{2}	2^{1}	2^{0}	2^{-1}	2^{-2}	...	...	...	$2^{-(N-1)}$	2^{-N}

Sign Bit

Decimal Point

Example:

//Binary to Decimal of Signed Integer

11010110 = $-2^7 + 2^6 + 2^4 + 2^2 + 2^1$ = -128 + 64 + 16 + 4 + 2 = -42

1101.0110 = $-2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3}$ = -8 + 4 + 1 + 0.25 + 0.125 = -2.625

//Decimal to Binary of Signed Integer

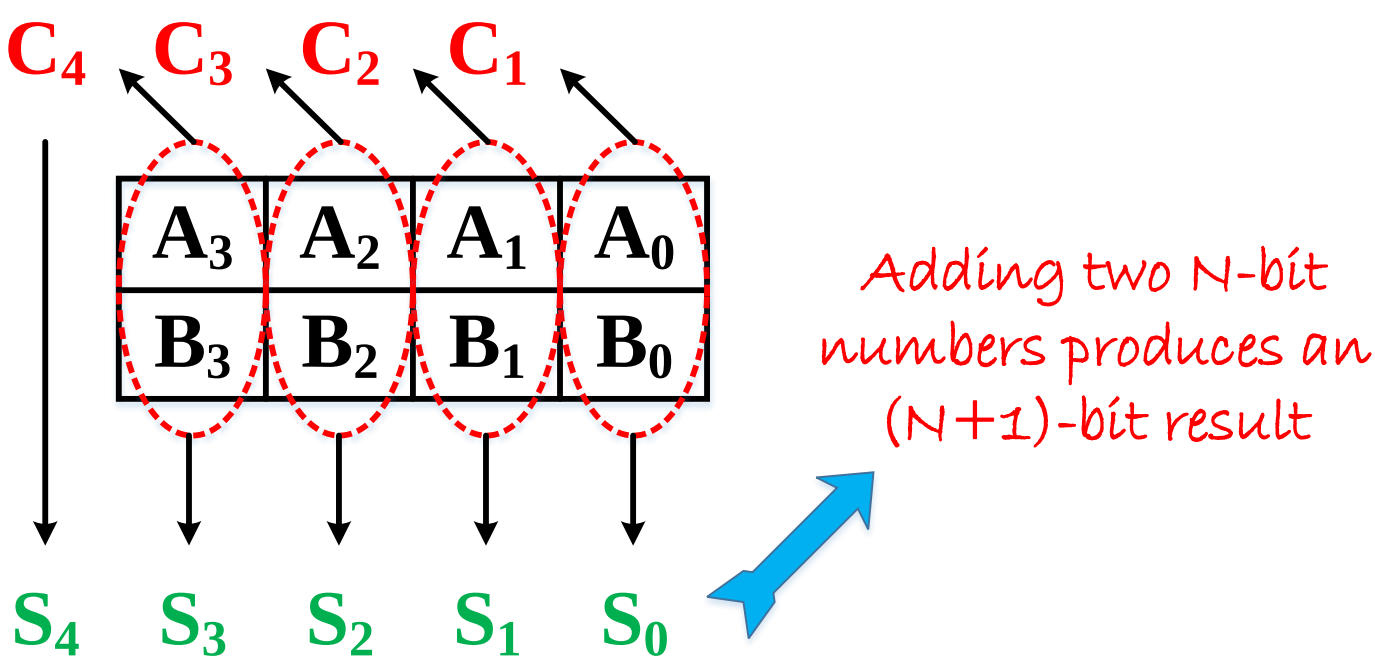
42 = 00101010

-5 = ~ (00000101) + 1 = 11111010 + 1 = 11111011

//Bit Extension of Signed Integer

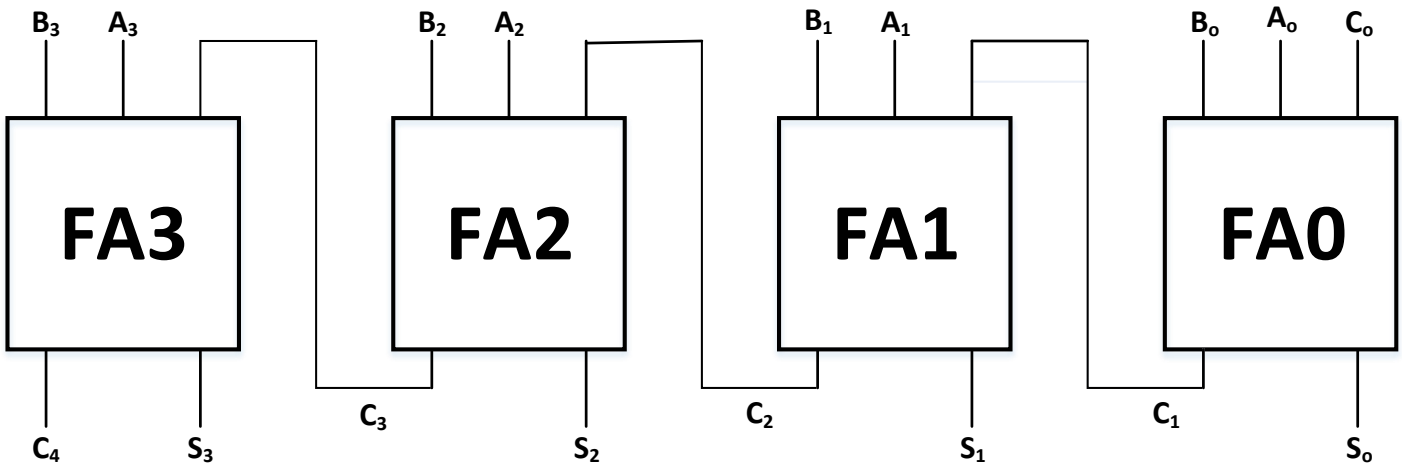
16-bit representation of ***42***: **00000000 00101010**

16-bit representation of ***-5***: **11111111 11111011**



PROBLEM 3.6: Ripple/Parallel Carry Adder

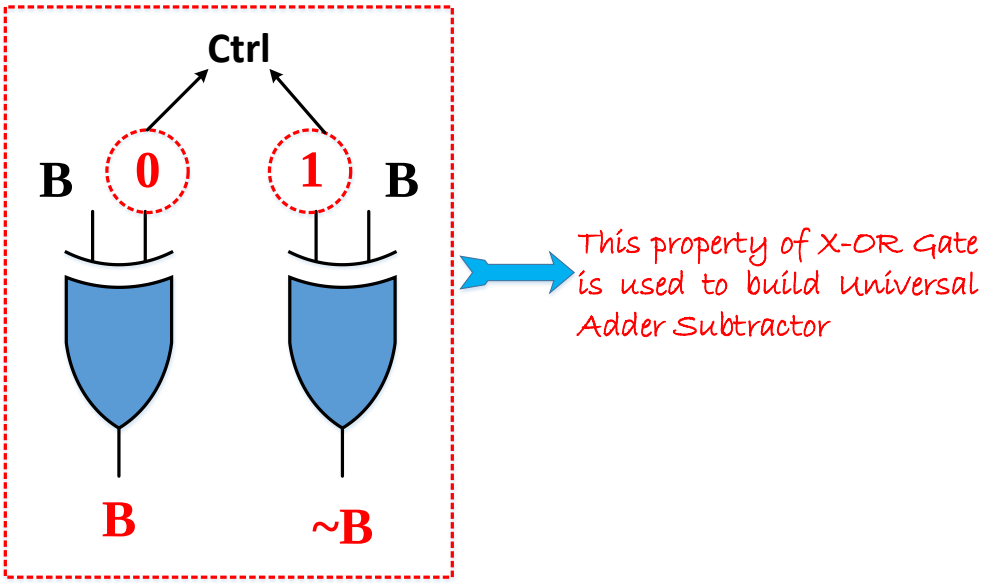
Block Diagram:



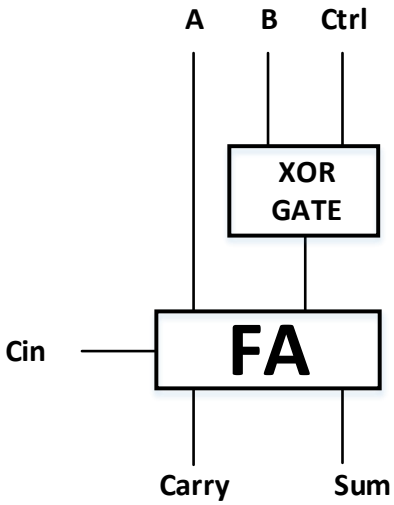
Verilog Program:

```
module Adder_4(Sum, Carry, A, B, Cin)
    input [3:0]A,B;
    input Cin;
    output [3:0]Sum;

    wire C1, C2, C3;
    output Carry;
    Full_AdderFA0(Sum [0], C1, A[0], B[0], Cin);
    Full_AdderFA1(Sum [1], C2, A[1], B[1], C1);
    Full_AdderFA2(Sum [2], C2, A[2], B[2], C2);
    Full_AdderFA3(Sum [3], Cout, A[3], B[3], C3);
endmodule

module Full_Adder ( A,B,Cin,S,Cy);
    input  A, B, Cin;
    output  S, Cy;
    assign S  = (A ^ B) ^ Cin;
    assign Cy = (A & B) | (B & Cin) | (Cin & A);
endmodule
```

PROBLEM 3.7: Universal Adder Subtractor (1 bit)

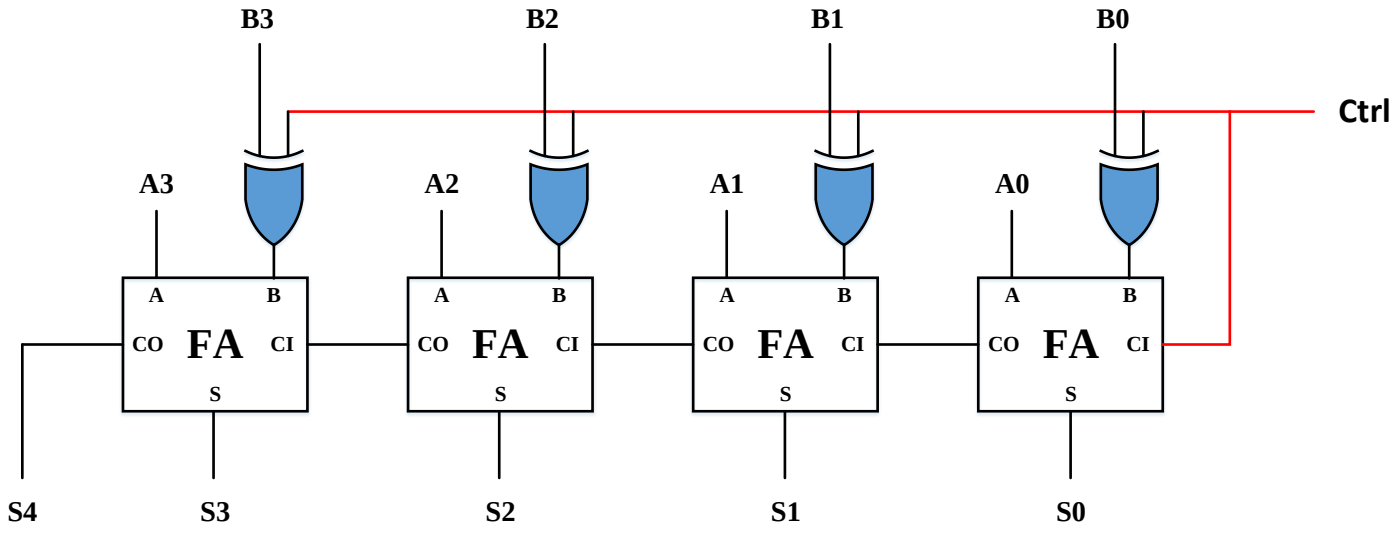


Block Diagram:

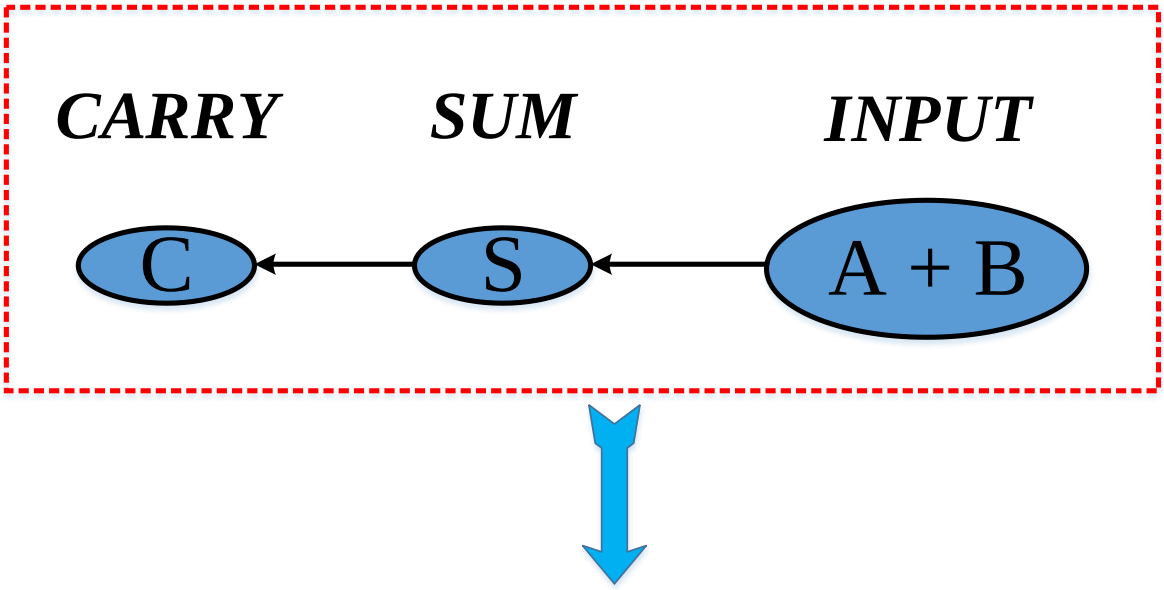


<u>Verilog Program (MODULE)</u>

```
module Univeral_Adder_Subtractor ( A, B, Ctrl, Cin, Sum, Carry);
    input  A, B, Cin, Ctrl;
    output  Sum, Carry;
    wire t1;
    xor (t1, B, Ctrl)
    Full_Adder FA0 (A, t1, Cin, Sum, Carry);
endmodule

module Full_Adder ( A,B,Cin,S,Cy);
    input  A, B, Cin;
    output  S, Cy;
    assign S  = (A ^ B) ^ Cin;
    assign Cy = (A & B) | (B & Cin) | (Cin & A);
endmodule
```


B3
B2
B1
B0
Ctrl
A3
A2
A1
A0
A
B
CO
FA
CI
S
S4
S3
S2
S1
S0

PROBLEM 3.8: Carry Look Ahead Adder



So, the carry mainly occurs due to identify carry. If we can predict carry ahead, it will save time.

A	B	C_{IN}	C_{OUT}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

NO CARRY

$C_{OUT} = (A \oplus B) \,.\, C_{IN}$

Carry Propagate

$C_{OUT} = (A \,.\, B)$

Carry Generate

$$C_{OUT} = (A \oplus B) \,.\, C_{IN} + (A \,.\, B) = P \,.\, C_{IN} + G$$

$$C_i = G + P_i \,.\, C_{i-1}$$

Boolean Equations:

$$G_i = A_i \,.\, B_i$$

$$P_i = A_i \oplus B_i$$

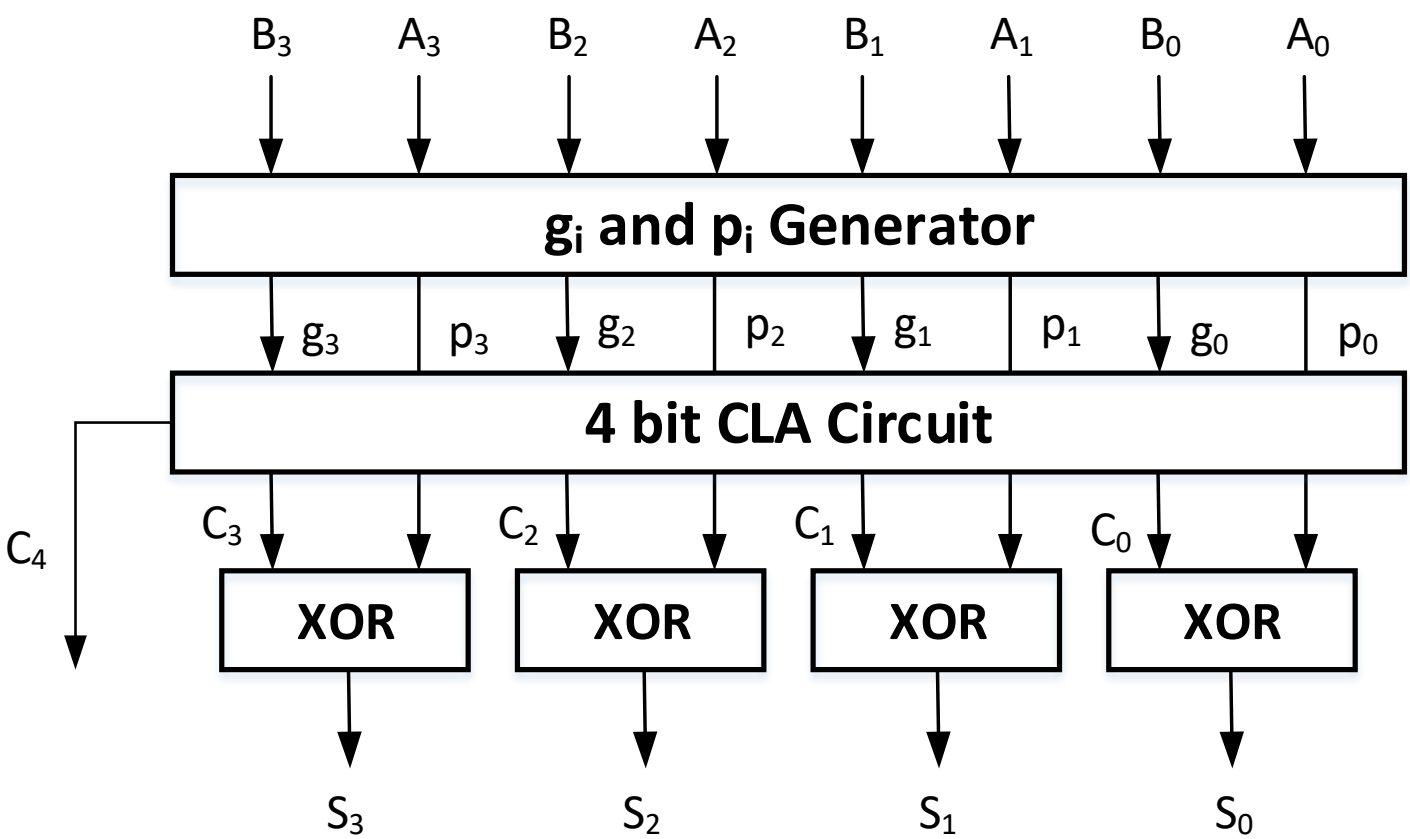
$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$$

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

<u>Verilog Program:</u>

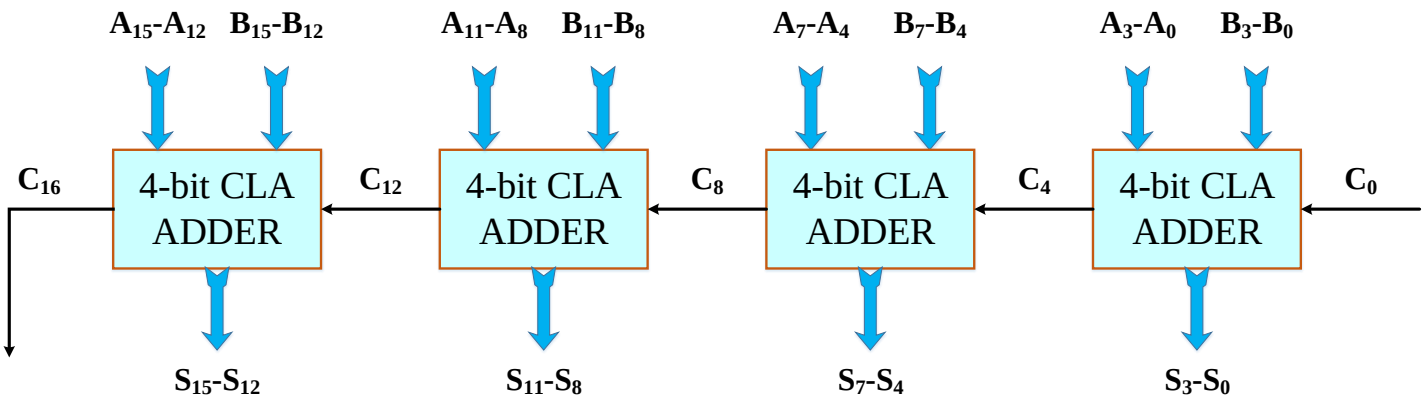
```
//1-bit Carry Look Ahead Adder
module CLAAdder (A, B, Cin, S, Cout);
        input A, B, Cin;
        output S, Cout;
        wire G, P;
        assign G = A & B;
        assign P = A ^ B;
        assign S = P ^ Cin;
        assign Cout = G | (P & Cin);
endmodule
```

Block Diagram:

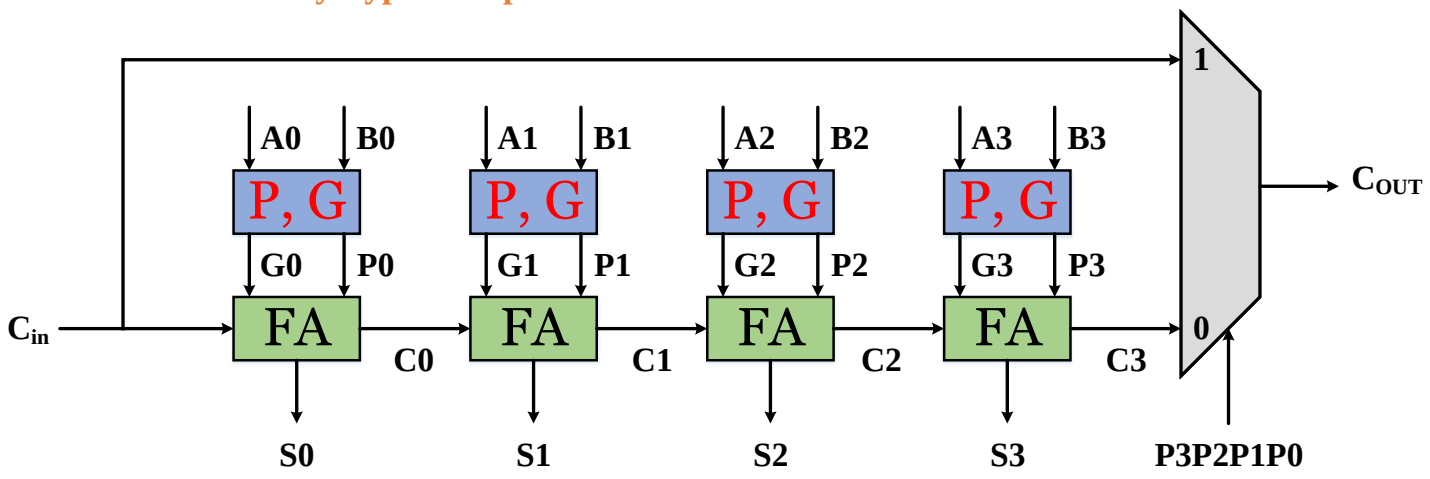


Verilog Program:

```
module carry_look_ahead_4bit(a,b, cin, sum,cout);
        input [3:0] a, b;
        input cin;
        output [3:0] sum;
        output cout;

        wire [3:0] p,g,c;

        assign p=a^b;//propagate
        assign g=a&b; //generate

        //carry=gi + Pi.ci

        assign c[0]=cin;
        assign c[1]= g[0]|(p[0]&c[0]);
        assign c[2]= g[1] | (p[1]&g[0]) | p[1]&p[0]&c[0];
        assign c[3]= g[2] | (p[2]&g[1]) | p[2]&p[1]&g[0] | p[2]&p[1]&p[0]&c[0];
        assign cout= 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[0];
        assign sum=p^c;

endmodule
```



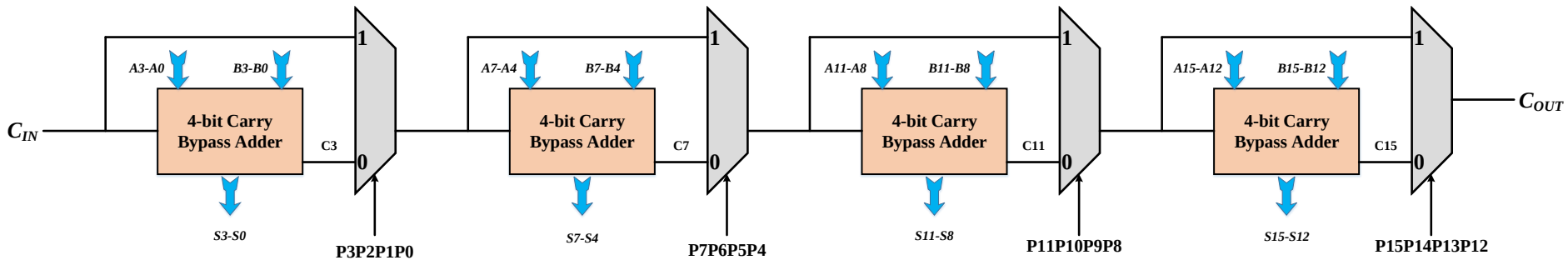
PROBLEM 3.9: Carry Bypass/Skip Adder



Verilog Program:

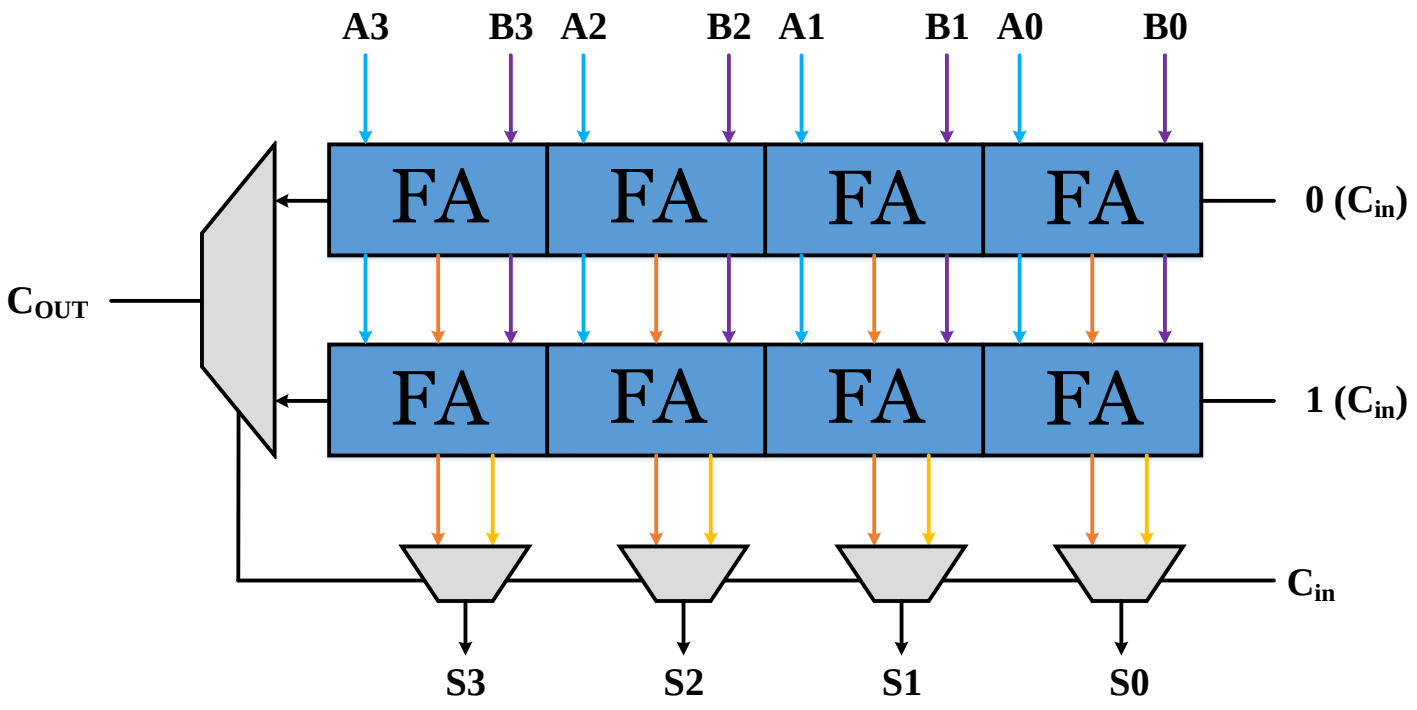
```
module Carry_Skip_4bit (A, B, Cin, Sum, Cout);
        input [3:0] A, B;
        input Cin;
        output [3:0] sum;
        output cout;
        wire [3:0] P;
        wire C0;
        wire BP;

        Ripple_Carry_4bit RCA (.A(A[3:0]), .B(B[3:0]), .Cin(Cin), .Sum(Sum[3:0]), .Cout(C0));
        Generate_Propagate GP (A, B, P, BP);
        MUX2X1 M0 (.In0(C0), .In1(Cin), .Sel(BP), .Out(Cout));
endmodule

// Propagate Generation
module Generate_Propagate (A, B, P, BP);
        input [3:0] A, B;
        output [3:0] P;
        output BP;
        assign P = A^B;                 //Get all propagate bits
        assign BP = &P;                 // and p0p1p2p3 bits
endmodule

//4-bit Ripple Carry Adder
module Ripple_Carry_4bit (A, B, Cin, Sum, Cout);
        input [3:0] A, B;
        input Cin;
        wire C1, C2, C3;
        output [3:0] Sum;
        output Cout;

        Full_Adder FA0 (.A(A[0]), .B(B[0]), .Cin(Cin), .Sum(Sum[0]), .Cout(C1));
        Full_Adder FA1 (.A(A[1]), .B(B[1]), .Cin(C1), .Sum(Sum[1]), .Cout(C2));
        Full_Adder FA2 (.A(A[2]), .B(B[2]), .Cin(C2), .Sum(Sum[2]), .Cout(C3));
```

```
        Full_Adder FA3 (.A(A[3]), .B(B[3]), .Cin(C3), .Sum(Sum[3]), .Cout(Cout));
endmodule

//1bit Full Adder
module Full_Adder ( A,B,Cin,S,Cout);
        input  A, B, Cin;
        output  S, Cout;
        assign S  = (A ^ B) ^ Cin;
        assign Cout = (A & B) | (B & Cin) | (Cin & A);
endmodule

//2X1 Mux
module MUX2X1( In0, In1, Sel, Out);
        input In0, In1;
        input Sel;
        output Out;
        assign Out = (Sel)?In1:In0;
endmodule
```


C_{IN}
A3-A0
B3-B0
4-bit Carry Bypass Adder
C3
S3-S0
1
0
P3P2P1P0
A7-A4
B7-B4
4-bit Carry Bypass Adder
C7
S7-S4
1
0
P7P6P5P4
A11-A8
B11-B8
4-bit Carry Bypass Adder
C11
S11-S8
1
0
P11P10P9P8
A15-A12
B15-B12
4-bit Carry Bypass Adder
C15
S15-S12
1
0
P15P14P13P12
C_{OUT}

PROEBLEM 3.10: Carry Select Adder

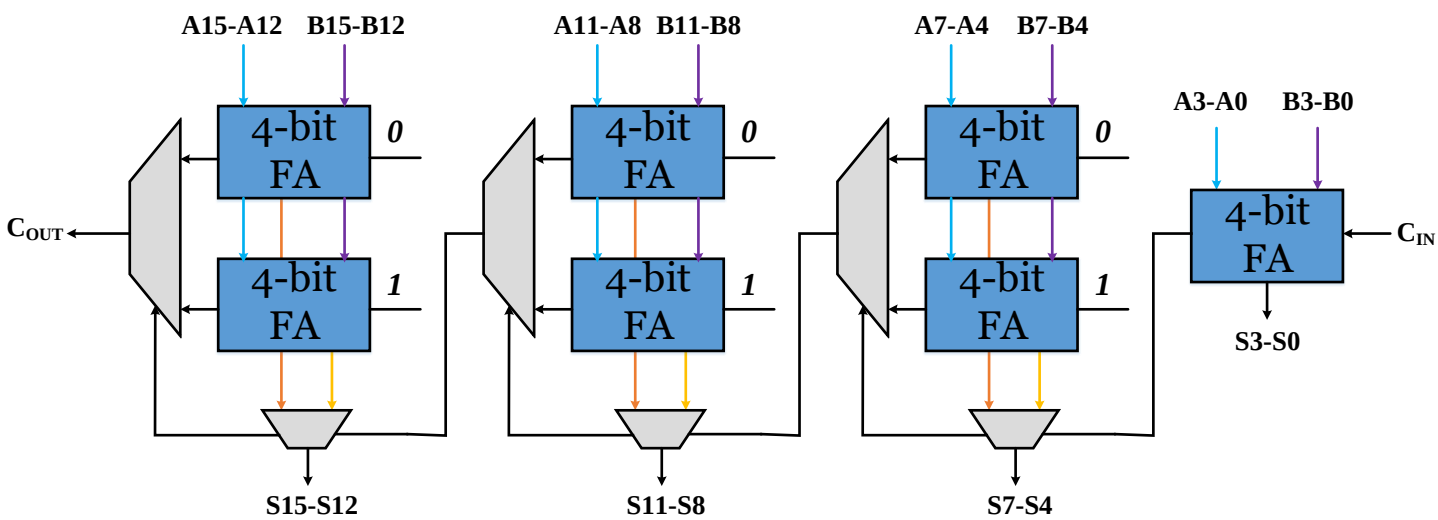
Carry Select Adder basically consists of two parallel adders (e.g. Ripple Carry Adder) and a multiplexer. Here, for two given numbers we carry out addition twice: ***i) With Carry-in as 0 ii) With Carry-in as 1***. Then when the correct ***Carry-in*** is known, the correct sum is selected by a multiplexer.



<u>Verilog Program:</u>

```
//4-bit Carry Select Adder
module Carry_Select_Adder_4bit (A, B, Cin, Sum, Cout);
        input [3:0] A, B;
        input Cin;
        output [3:0] Sum;
        output Cout;

        wire [3:0] S0,S1;
        wire C0,C1;

        Ripple_Carry_4bit RCA1 (.A(A), .B(B), .Cin(1'b0), .Sum(S0), .Cout(C0));
        Ripple_Carry_4bit RCA2 (.A(A), .B(B), .Cin(1'b1), .Sum(S1), .Cout(C1));

        MUX2X1 MS (.In0(S0), .In1(S1), .Sel(Cin), .Out(Sum); // Sum Select
        MUX2X1 MC (.In0(C0), .In1(C1), .Sel(Cin), .Out(Cout)); // Carry Select
endmodule


//2X1 MUX
module MUX2X1( in0,in1,sel,out);
        parameter width=2;
        input [width-1:0] in0,in1;
        input sel;
```

```
        output [width-1:0] out;
        assign Out=(Sel)?In1:In0;
endmodule


//4-bit Ripple Carry Adder
module Ripple_Carry_4bit (A, B, Cin, Sum, Cout);
        input [3:0] A, B;
        input Cin;
        output [3:0] Sum;
        output Cout;
        wire C1, C2, C3;
        Full_Adder FA0( .A(A[0]), .B(B[0]), .Cin(Cin), .Sum(Sum[0]), .Cout(C1));
        Full_Adder FA1( .A(A[1]), .B(B[1]), .Cin(C1), .Sum(Sum[1]), .Cout(C2));
        Full_Adder FA2( .A(A[2]), .B(B[2]), .Cin(C2), .Sum(Sum[2]), .Cout(C3));
        Full_Adder FA3( .A(A[3]), .B(B[3]), .Cin(C3), .Sum(Sum[3]), .Cout(Cout));
endmodule

//1bit Full Adder
module Full_Adder ( A,B,Cin,S,Cout);
        input  A, B, Cin;
        output  S, Cout;
        assign S  = (A ^ B) ^ Cin;
        assign Cout = (A & B) | (B & Cin) | (Cin & A);
endmodule
```



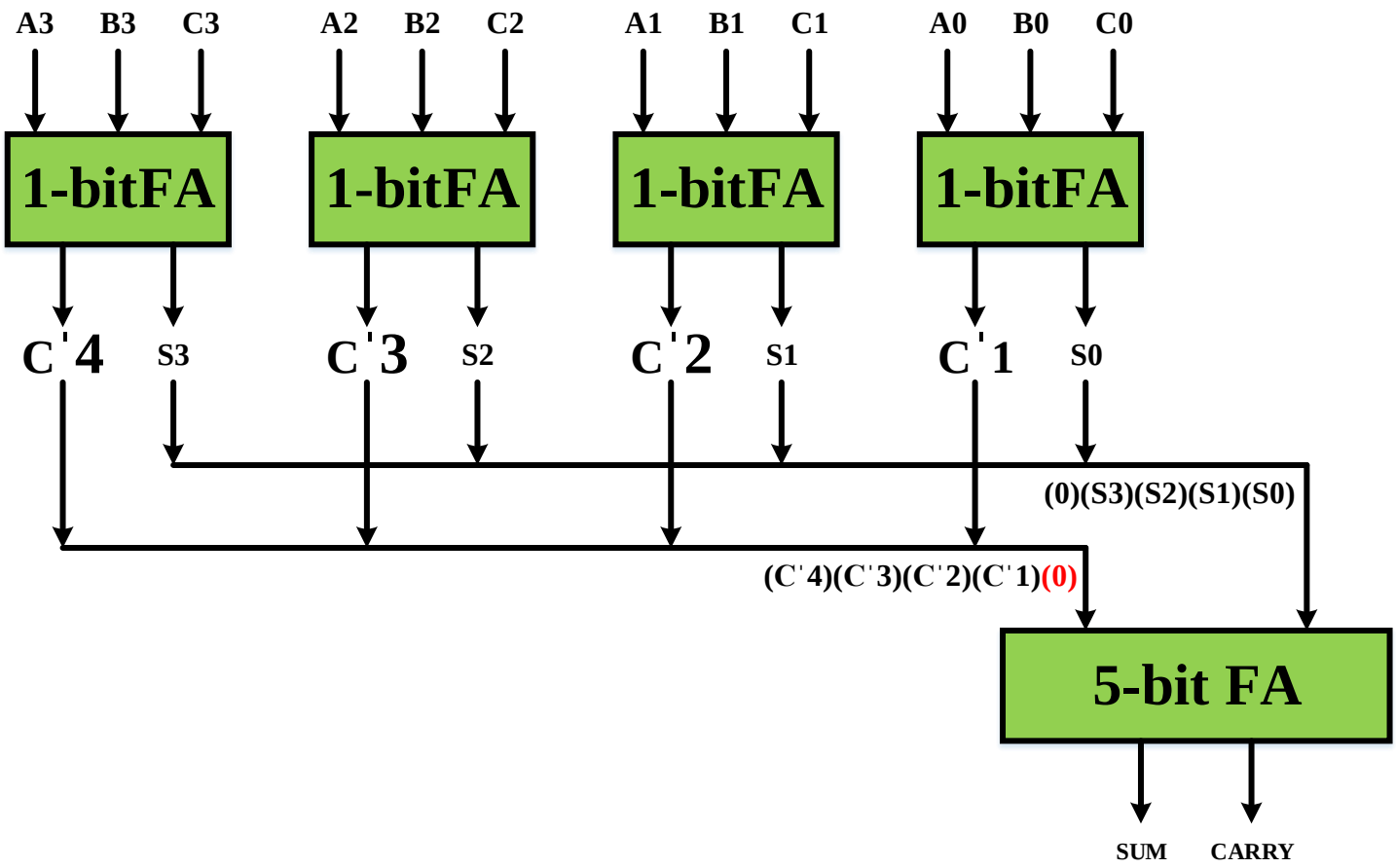
PROEBLEM 3.11: CARRY SAVE ADDER

X		1	0	0	1	1
Y		1	1	0	0	1
Z		0	1	0	1	1
C	1	1	0	1	1	

X		1	0	0	1	1
Y		1	1	0	0	1
Z		0	1	0	1	1
S		0	0	0	0	1

X		1	0	0	1	1
Y		1	1	0	0	1
Z		0	1	0	1	1
S		0	0	0	0	1
C	1	1	0	1	1	
SUM	1	1	0	1	1	1

STEP-1: A set of FAs generates Carry and Sum bits in parallel.
STEP-2: The Sum and Carry vectors are added later with proper shifting.



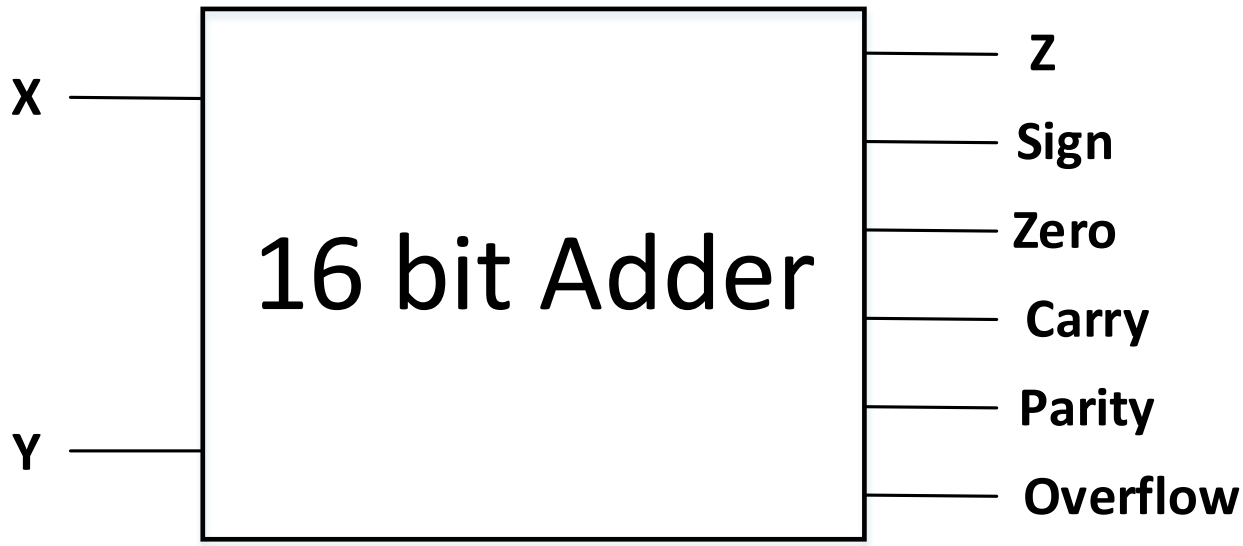
Verilog Program:

```
module Carry_Save_Adder_4bit (A, B, C, Sum, Carry);
        input [3:0] A, B, C;
        output [3:0] Sum;
        output Carry;

        wire [3:0] S, CP;
        wire [4:0] Operand1, Operand2;
```

```
        Full_Adder FA0 (.A (A[0]), .B (B[0]), .Cin (C[0]), .S (S[0]), .Cout (CP[0]));
        Full_Adder FA0 (.A (A[1]), .B (B[1]), .Cin (C[1]), .S (S[1]), .Cout (CP[1]));
        Full_Adder FA0 (.A (A[2]), .B (B[2]), .Cin (C[2]), .S (S[2]), .Cout (CP[2]));
        Full_Adder FA0 (.A (A[3]), .B (B[3]), .Cin (C[3]), .S (S[3]), .Cout (CP[3]));

        Operand1 = {CP , 1'b0};
        Operand2 = {1'b0 , S};
        {Carry, Sum} = Operand1 + Operand2;
endmodule

//1bit Full Adder
module Full_Adder ( A,B,Cin,S,Cout);
        input  A, B, Cin;
        output  S, Cout;
        assign S  = (A ^ B) ^ Cin;
        assign Cout = (A & B) | (B & Cin) | (Cin & A);
endmodule
```

PROBLEM 3.11: 16-bit ADDER including status Flags (Sign, Zero, Carry, Parity, Overflow)

Block Diagram:



Status Flags:

Sign: This status flag determines whether the sum is positive or negative. If the MSB of the result is 0, then the number is positive. Otherwise, the number is negative.

CONDITION	SIGN FLAG	Interpretation	Coding Techniques
MSB = 0	0	Number is Positive	**Just *assign* MSB to the Flag**
MSB = 1	1	Number is Negative	

Zero: This status flag determines whether the sum is zero or not. If the summation is zero, the status flag will be updated to 1.

CONDITION	ZERO FLAG	Interpretation	Coding Techniques
SUM of all bits = 1	0	Result (Summation) is Non-Zero	**Perform *NOR* operation on Result and *assign* to the Flag**
SUM of all bits = 0	1	Result (Summation) is Zero	

Carry: This status flag determines whether there is a carry out of the last stage.

CONDITION	ZERO FLAG	Interpretation	Coding Techniques
Carry = 1	1	Carry generated from MSB	**Just *assign* carry to the Flag**
Carry = 0	0	Carry not generated from MSB	

Parity: This status flag determines whether the number of 1's in the sum is even or odd. If there are even number of 1's / even number of 0's, then then the output of X-NOR gate is 1.

CONDITION	Parity FLAG	Coding Techniques
Number of 1's in Sum = EVEN	1	**Perform *XNOR* operation on Result and *assign* to the Flag**
Number of 1's in Sum = ODD	0	

Overflow: If X, Y represents sign of two numbers and Z represents sign of result then overflow is given by the following equation-

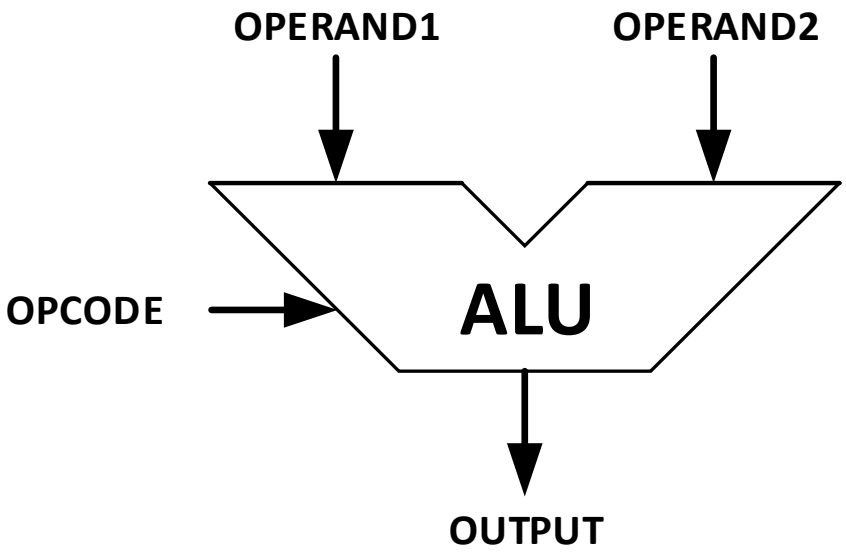
Overflow = $X_{N-1}Y_{N-1}\overline{Z_{N-1}} + \overline{X_{N-1}}\,\overline{Y_{N-1}}Z_{N-1}$

Verilog Program:

```
module ALU (X, Y, Z, Sign, Zero, Carry, Parity, Overflow);
        input [15:0] X, Y;
        output [15:0] Z;
        output Sign, Zero, Carry, Parity, Overflow;
        assign {Carry, Z} = X+Y;
        assign Sign = Z[15];
        assign Zero = ~| Z;
        assign Parity = ~^Z;
        assign Overflow = (X[15] & Y[15] & ~Z[15]) | (~X[15] & ~ Y[15] & Z[15]);
endmodule
```

PROBLEM 3.12: 8-bit Arithmetic Unit (Addition, Subtraction, Multiplication, Division)

Block Diagram:



The ALU works on 8-bit operands. It supports 4 instructions, which are selected by 2-bit opcode.

Serial No.	Opcode	Operation
1	00	Output = Operand1 + Operand2
2	01	Output = Operand1 - Operand2
3	10	Output = Operand1 * Operand2
4	11	Output = Operand1 / Operand2

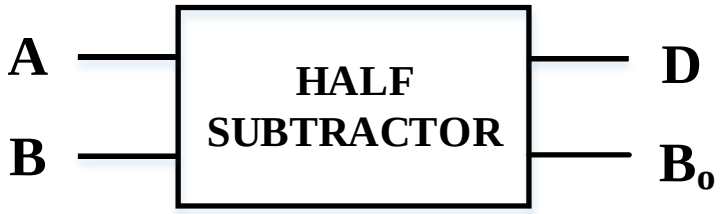
Verilog Program:

```
module ALU (Output, Operand1, Operand2, Opcode);
   input [1:0] Opcode;
   input [7:0] Operand1, Operand2;
   output reg [15:0] Output = 16'b0;
   parameter ADD = 2'b00, SUB = 2'b01, MUL = 2'b10, DIV = 2'b11;
always @ (*)
   case (OP)
      ADD  : Output = Operand1 + Operand2;
      SUB  : Output = Operand1 - Operand2;
      MUL  : Output = Operand1 * Operand2;
      DIV  : Output = Operand1 / Operand2;
   endcase
endmodule
```

CATEGORY-4: SUBTRACTOR

PROBLEM 4.1: Half Subtractor (1 bit)

Block Diagram:



Truth Table:

A	B	D	B_o
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Boolean Equations:

D = A ^ B;

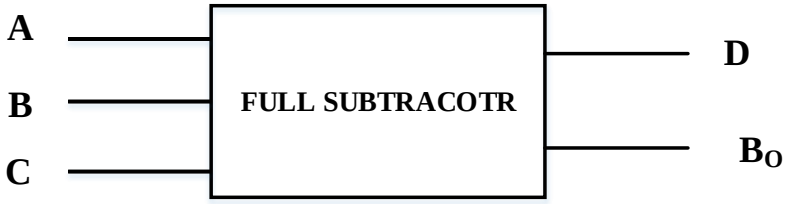
Bo = (~A). B

Verilog Program:

```
module Half_Substractor (A,B,D,Bo);
    input A,B;
    output D, Bo;
    assign D = A ^ B;
    assign Bo = (~A&B);
endmodule
```

PROBLEM 4.2: Full Subtractor (1 bit)

Block Diagram:



Truth Table:

A	B	C_{in}	D	B_o
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Boolean Equations:

D = A ^ B ^ C;

B_o = (~A.B) + (~A.C) + B.C

Verilog Program:

```
module Full_Subtractor (A, B, C, D, Bo);
        input A, B, C;
        output D, Bo;
        assign D = (A ^ B ^ C);
        assign Bout = (~A & B) | (~A & C) | (B & C);
endmodule
```

PROBLEM 4.3: Subtractor (4 bit)

Verilog Program:

```
module Subtractor_4bit (A, B, Bin, Difference, Bout);
        input [3:0] A, B;
        input Bin;
        output [3:0] Difference;
        output Bout;
        assign {Bout, Difference} = A-B-Bin;
endmodule
```

CATEGORY-5: MULTIPLIER

$$
\begin{array}{rccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 \\
 & & & & B_3 & B_2 & B_1 & B_0 \\
 & & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
\end{array}
$$

Multiplying N-bit number by M-bit number gives (N+M)-bit result

SHIFT and ADD MULTIPLIER:

Let's create our algorithm for 4-bit multiplication:

INPUT: A, B
OUTPUT: Result = A * B

```
Load A (A3A2A1A0)
Load B (B3B2B1B0)
while (i < 4)
{
        if (Bi == 1)
                if (i == 0)
                Result = Result + A
                else
                A = A << 1
                Result = Result + A
        else (Bi == 0)
                Result = Result
        i++
}
```

Let's see how we can convert our understanding of algorithm into hardware:

ALGORITHM	EXPLANATION	HARDWARE
Load A	When we activate the device, we have to load A	PIPO
Load B	When we activate the device, we have to load A. However, our algorithm works depending on the single bit of B and which bit will be selected is depending on the iteration number	PISO
while (i < 4) i++	We have to count from 0 to 3	COUNTER
if (B_i == 1) **else** (B_i == 0)	Depending on value of B_i 1 particular operation will be selected	MULTIPLEXER
if (i == 0) **else**	Depending on value of ***i*** 1 particular operation will be selected	MULTIPLEXER
A = A << 1	Shifting A to 1-bit left	LEFT SHIFTER
Result = Result + A	Adding two numbers	ADDER

BOOTH (RADIX-4) MULTIPLIER:

Radix-4 (3-bit recoding) reduces number of partial products to be added by half. Here, 3 bits of multiplier B **(b_{2i+1}, b_{2i}, b_{2i-1})** are examined and corresponding k_i is calculated. Here,

$$K_i = -2b_{2i+1} + b_{2i} + b_{2i-1}$$

B is always appended on the right with zero and n is always even (B is sign extended if needed).

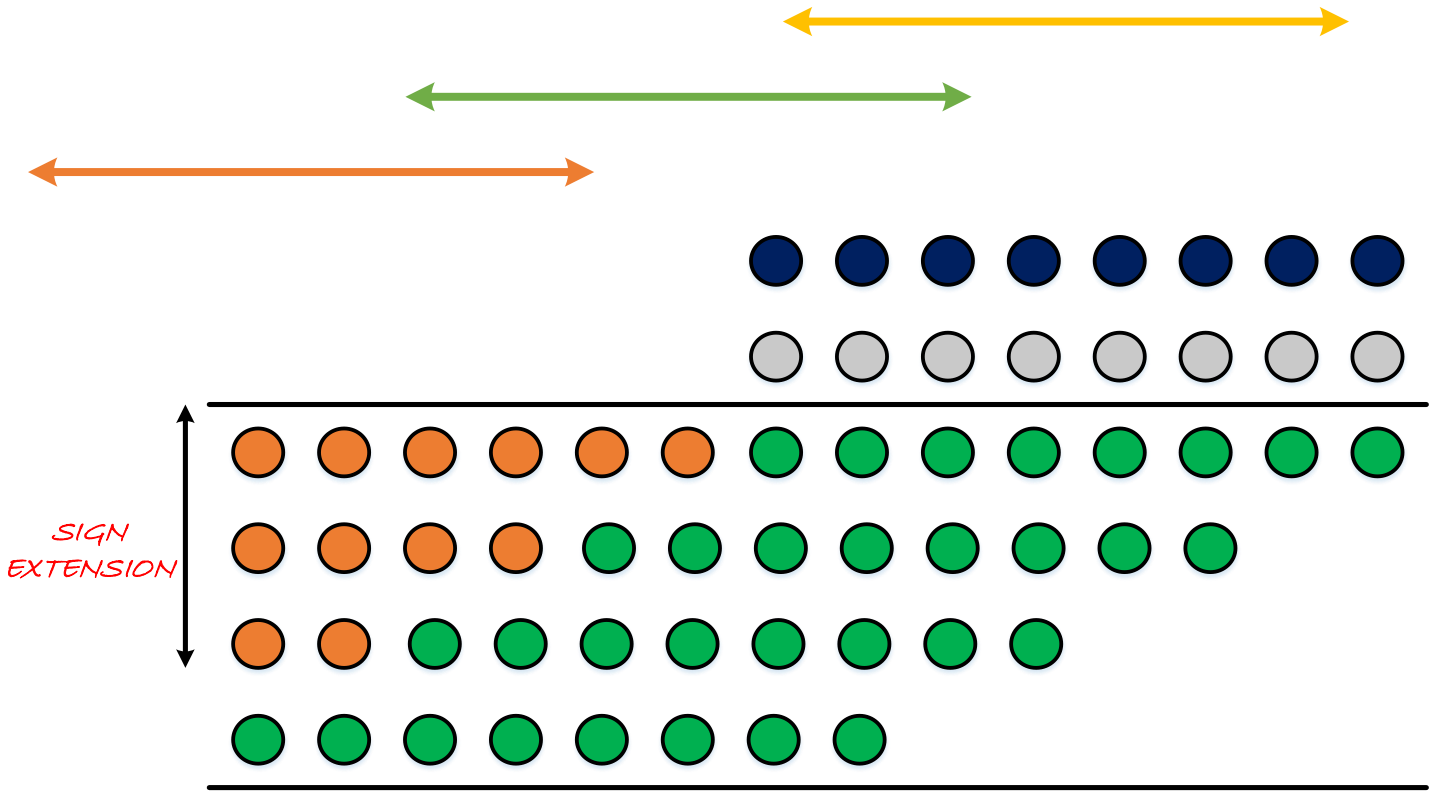
$$Product = A.B = \sum_{i=0}^{i=(n/2)-1} 2^{2i} K_i A$$

$X_{i+1}(2^{-1})$	$X_i(2^{-0})$	$X_{i-1}(2^{-0})$	OP	NEG	ZERO	TWO
0	0	0	0	0	1	0
1	0	1	2	1	0	1
0	1	0	1	0	0	0
1	1	0	1	1	0	0
0	0	1	1	0	0	0
1	0	1	1	1	0	0
0	1	1	2	0	0	1
1	1	1	0	1	1	0

Example:

Multiplicand, X = 000011 = 3
Multiplier, Y = 011101 = 29
Appending "0" to the multiplier, we get-

0	1	1	1	0	1	0



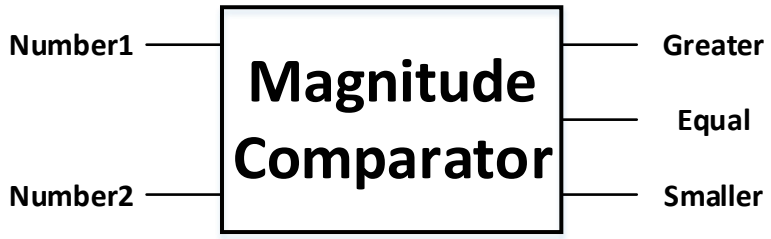
DECODING	MEANING	OPERATION			SHIFTING									
010	+1	X * (+1)	X	000011	0	0	0	0	0	0	0	0	1	1
110	-1	X * (-1)	- X	111101	1	1	1	1	1	1	0	1		
011	+2	X * (+2)	2X	000110	0	0	0	1	1	0				
				RESULT = 1	0	0	0	1	0	1	0	1	1	1

Example: (Do Youself)
Multiplicand, X = 111101 = -3
Multiplier, Y = 011101 = 29

**** For the time being, I am skipping the VERILOG code of the multiplication circuit. However, we will resume once we will finish the concept of FSM****

CATEGORY-6: COMPARATOR

PROBLEM 6.1: 8-BIT MAGNITUDE COMPARATOR



Condition	Status
Number1 > Number2	Greater = 1
Number1 < Number2	Smaller = 1
Number 1 == Number2	Equal = 1

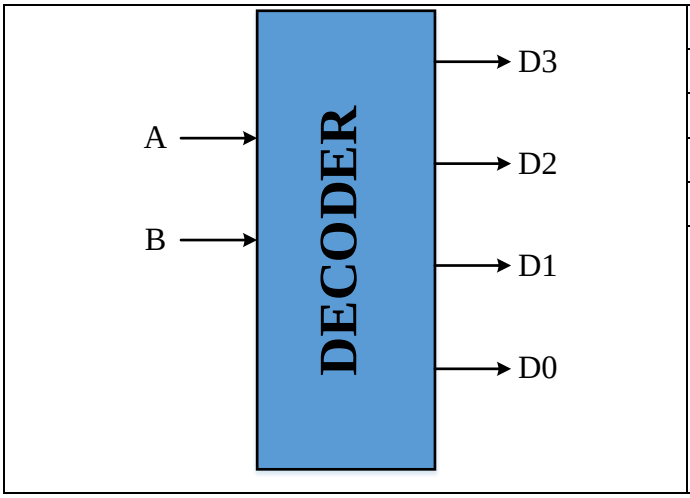
Verilog Program:

```
module Magnitude_Comparator (Number1, Number2, Greater, Smaller, Equal);
        input [7:0] Number1, Number2;
        output reg Greater, Smaller, Equal;
        always @ (Number1 or Number2)
        begin
                Greater <= (Number1 > Number2)? 1'b1 : 1'b0;
                Smaller <= (Number1 < Number2)? 1'b1 : 1'b0;
                Equal   <= (Number1 == Number2)? 1'b1 : 1'b0;
        end
endmodule
```

CATEGORY-7: DECODER

A Decoder has ***n*** inputs and $\mathbf{2^n}$ outputs. Out of all the outputs, only one output will be selected at a time. The usage of decoder are: 1) Selection of a word within a memory 2) Selection of one module connected to a bus when many modules are connected.

PROBLEM 7.1: 2 X 4 DECODER (DATA FLOW STYLE)



A	B	D3	D2	D1	D0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

BOOLEAN EQUATIONS:

$$D0 = \overline{A} \,.\, \overline{B}$$
$$D1 = \overline{A} \,.B$$
$$D2 = A \,.\, \overline{B}$$
$$D3 = \;.B$$

Verilog Program:

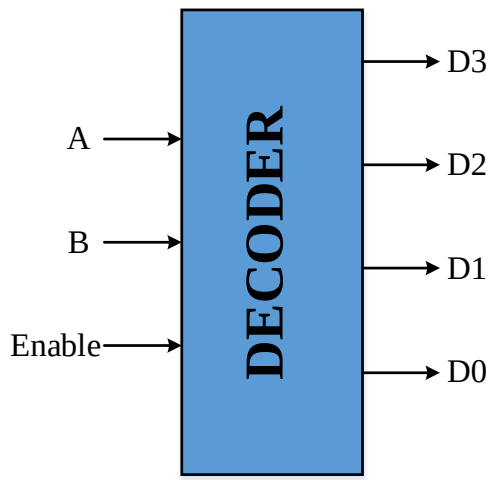
```
module 2X4DECODER (In, Out);
        input [1:0] In;
        output wire [3:0] Out;
        assign Out[0] = ~(In[0]) & ~(In[1]);
        assign Out[1] = ~(In[0]) & (In[1]);
        assign Out[2] = (In[0]) & ~(In[1]);
        assign Out[3] = (In[0]) & (In[1]);
endmodule
```

PROBLEM 7.2: 2 X 4 DECODER (BEHAVIORAL MODELING)

Verilog Program:

```
module 2X4DECODER (In, Out);
        input [1:0] In;
        output reg [3:0] Out;
        always @ (In)
        begin
        case (In)
                2'b00: begin Out = 4'b0001; end
                2'b01: begin Out = 4'b0010; end
                2'b10: begin Out = 4'b0100; end
                2'b11: begin Out = 4'b1000; end
        endcase
        end
endmodule
```

PROBLEM 7.2: 2 X 4 DECODER with Enable (BEHAVIORAL MODELING)




```
module 2X4DECODER (A, B, Enable, Out);
        input A, B;
        input Enable;
        output reg [3:0] Out;
        always @ (Enable, A, B)
        begin
        if (Enable == 0)
        begin
                if (A == 1'b0 & B == 1'b0) Out = 4'b0001;
                else if (A == 1'b0 & B == 1'b1) Out = 4'b0010;
                else if (A == 1'b1 & B == 1'b0) Out = 4'b0100;
                else if (A == 1'b1 & B == 1'b1) Out = 4'b1000;
                else Out = 4'bxxxx;
        end
        else
                Out = 4'b1111;
        end
endmodule
```

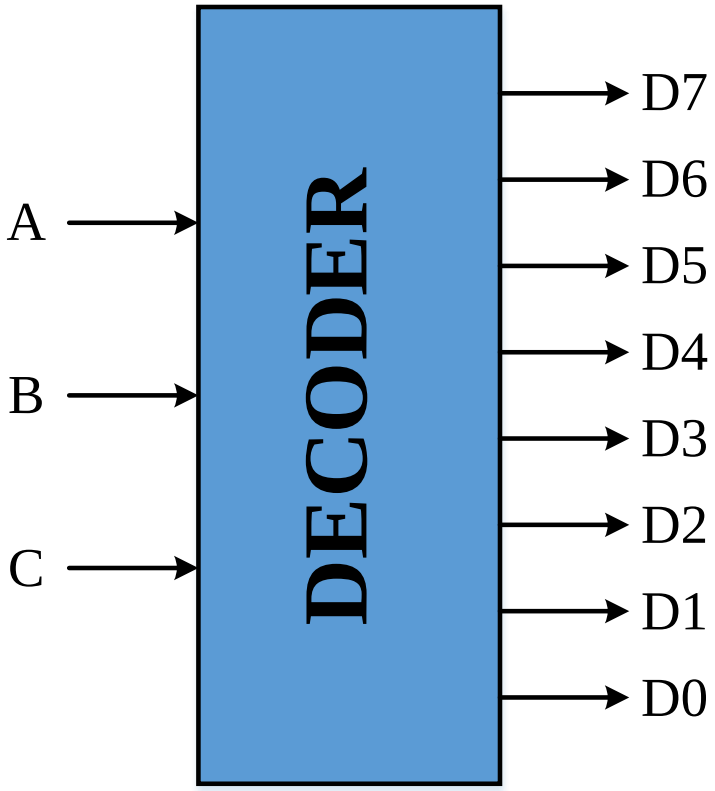
PROBLEM 7.3: 2 X 4 DECODER (Non-Constant Index)

```
module 2X4DECODER (In, Out, Select);
        input In;
        input [0:1] Select;
        output wire [3:0] Out;
        assign Out[Select] = In;
endmodule
```

PROBLEM 7.4: 2 X 4 DECODER (Left Shift Operator)

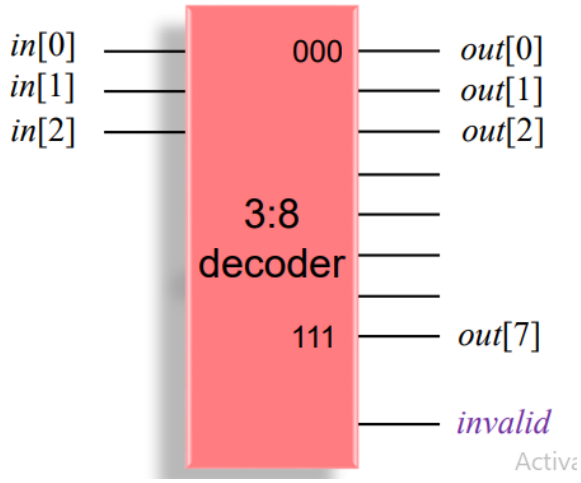
```
module 2X4DECODER (In, Out);
        input [1:0] In;
        output wire [3:0] Out;
        assign Out = 4'b0001 << In;
endmodule
```

```
module 2X4DECODER (In, Out);
        input [1:0] In;
        output reg [3:0] Out;
        always @ (In)
        begin
                Out = 4'b0001 << In;
        end
endmodule
```



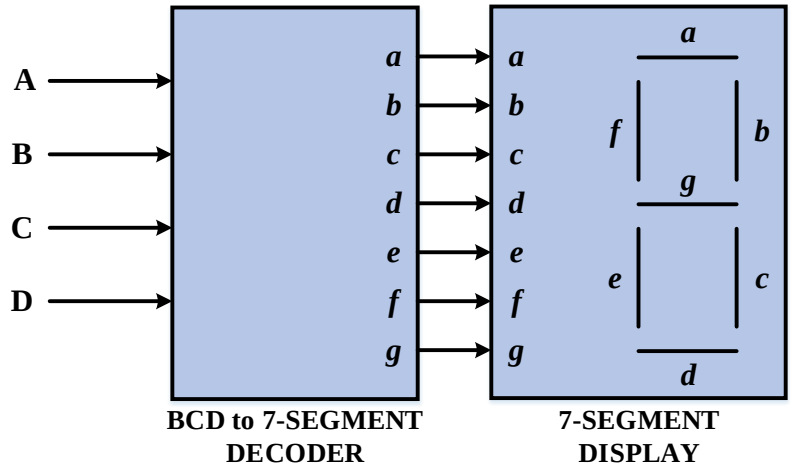
PROBLEM 7.5: Design a 3 X 8 DECODER (DATA FLOW STYLE)
PROBLEM 7.6: Design a 3 X 8 DECODER (BEHAVIORAL STYLE)
PROBLEM 7.7: Design a 3 X 8 DECODER with Enable (BEHAVIORAL STYLE)
PROBLEM 7.8: 3 X 8 DECODER (Non-Constant Index)
PROBLEM 7.9: 3 X 8 DECODER (Left Shift Operator)
PROBLEM 7.10: Design a 4 X 16 DECODER using 3 X 8 DECODER

PROBLEM 7.11: Design a 3 X 8 DECODER with an invalid output signal.




```
module 3X8DECODER (In, Out);
        input [1:0] In;
        output reg [3:0] Out;
        always @ (In)
        begin
                invalid = 1'b0;
        case (In)
                3'b000: begin Out = 8'b00000001; end
                3'b001: begin Out = 8'b00000010; end
                3'b010: begin Out = 8'b00000100; end
                3'b011: begin Out = 8'b00001000; end
                3'b100: begin Out = 8'b00010000; end
                3'b101: begin Out = 8'b00100000; end
                3'b110: begin Out = 8'b01000000; end
                3'b111: begin Out = 8'b10000000; end
               default: begin
                                Out = 8'b00000000;
                                invalid = 1'b1;
                        end
        endcase
        end
endmodule
```

PROBLEM 7.12: Design a BCD to 7-Segment Decoder



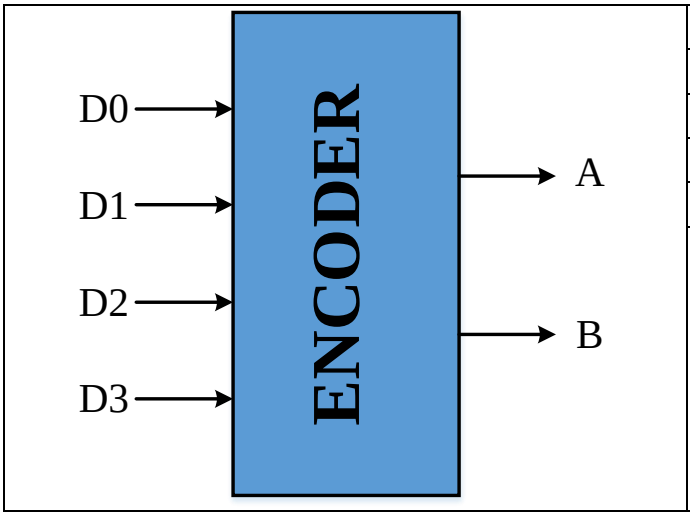
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

```
module BCD_to_7SEG (BCD, SEG);
        input [3:0] BCD;
        output reg [6:0] SEG;

        always@(BCD)
                begin
                        case
                                0: SEG = 7'b1111110;
                                1: SEG = 7'b0110000;
                                2: SEG = 7'b1101101;
                                3: SEG = 7'b1111001;
                                4: SEG = 7'b0110011;
                                5: SEG = 7'b1011011;
                                6: SEG = 7'b1011111;
                                7: SEG = 7'b1110000;
                                8: SEG = 7'b1111111;
                                9: SEG = 7'b1111011;
                                default: SEG = 7'b0000000;
                        endcase
                end
endmodule
```

CATEGORY-8: ENCODER

An encoder has 2n input lines and n output lines. The output lines generate the binary code corresponding to each input value.

PROBLEM 8.1: 4 X 2 ENCODER (DATA FLOW STYLE)



D3	D2	D1	D0	A1	A0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

BOOLEAN EQUATIONS:

A0 = D1 + D3

A1 = D2 + D3

Verilog Program:

```
module 4X2ENCODER (In, Out);
        input [3:0] In;
        output wire [1:0] Out;
        assign Out[0] = In[1] | In[3];
        assign Out[1] = In[2] | In[3];
endmodule
```

PROBLEM 8.2: 4 X 2 ENCODER (BEHAVIORAL MODELING using case STATEMENT)

Verilog Program:

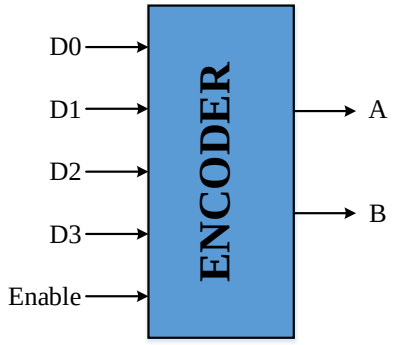
```
module 4X2ENCODER (In, Out);
        input [3:0] In;
        output reg [1:0] Out;
        always @ (In)
        begin
        case (In)
                4'b0001: begin Out = 2'b00; end
                4'b0010: begin Out = 2'b01; end
                4'b0100: begin Out = 2'b10; end
                4'b1000: begin Out = 2'b11; end
                default: begin Out = 2'bZZ; end
        endcase
        end
endmodule
```

PROBLEM 8.3: 4 X 2 ENCODER (BEHAVIORAL MODELING using if else STATEMENT)

Verilog Program:

```
module 4X2ENCODER (In, Out);
        input [3:0] In;
        output reg [1:0] Out;
        always @ (In)
        begin
                if (In == 4'b0001) begin Out = 2'b00; end
                else if (In == 4'b0010) begin Out = 2'b01; end
                else if (In == 4'b0100) begin Out = 2'b10; end
                else if (In == 4'b1000) begin Out = 2'b11; end
                else begin Out = 2'bX; end
        end
endmodule
```

PROBLEM 8.4: 4 X 2 ENCODER with Enable (BEHAVIORAL MODELING)



Verilog Program:

```
module 4X2ENCODER (D, Enable, Out);
        input [3:0] D;
        input Enable;
        output reg [1:0] Out;
        always @ (Enable, D)
        begin
                if (Enable == 0)
                begin
                case (D)
                        4'b0001: begin Out = 2'b00; end
                        4'b0010: begin Out = 2'b01; end
                        4'b0100: begin Out = 2'b10; end
                        4'b1000: begin Out = 2'b11; end
                endcase
                end
               else
                        Out = 2'b11;
        end
endmodule
```

PROBLEM 8.5: 4 X 2 PRIORITY ENCODER

D3	D2	D1	D0	A1	A0
0	0	0	1	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1

Verilog Program:

```
module 4X2PENCODER (In, Out);
        input [3:0] In;
        output reg [1:0] Out;
        always @ (In)
        begin
        casex (In)
                4'b0001: begin Out = 2'b00; end
                4'b001x: begin Out = 2'b01; end
                4'b01xx: begin Out = 2'b10; end
                4'b1xxx: begin Out = 2'b11; end
                default: begin Out = 2'bZZ; end
        endcase
        end
endmodule
```

PROBLEM 8.6: 8 X 3 ENCODER (DATA FLOW STYLE)

PROBLEM 8.7: 8 X 3 ENCODER (BEHAVIORAL MODELING using case STATEMENT)

PROBLEM 8.8: 8 X 3 ENCODER (BEHAVIORAL MODELING using if else STATEMENT)

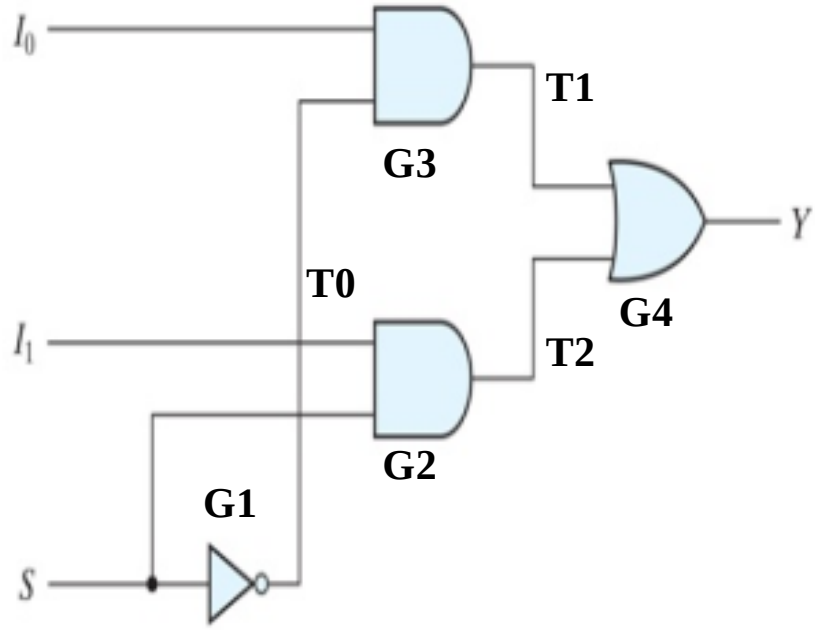
PROBLEM 8.9: 8 X 3 ENCODER with Enable (BEHAVIORAL MODELING)

PROBLEM 8.10: 8 X 3 PRIORITY ENCODER with Enable (BEHAVIORAL MODELING)

CATEGORY-9: MULTIPLEXER

A multiplexer device selects one of the several input signals and forwards the selected inputs to the output. Typical multiplexers follows 2:1, 4:1, 8:1, 16:1 structures. A multiplexer has 2n input lines, n select lines, and 1 output line.

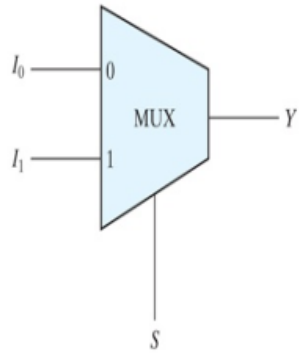
PROBLEM 9.1: 2 X 1 MUX (GATE LEVEL MODELING)



Verilog Program:

```
module 2X1MUX (I, S, Y);
        input [1:0] I;
        input S;
        output Y;

        not G1 (T0, S);
        and G2 (T2, S, I[1]);
        and G3 (T1, I[0], T0);
        or G4 (Y, T1, T2);
endmodule
```

PROBLEM 9.2: 2 X 1 MUX (DATA FLOW/CONTINUOUS MODELING)



S	I0	I1	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

$$Y = \overline{S}I0 + SI1$$

Verilog Program:

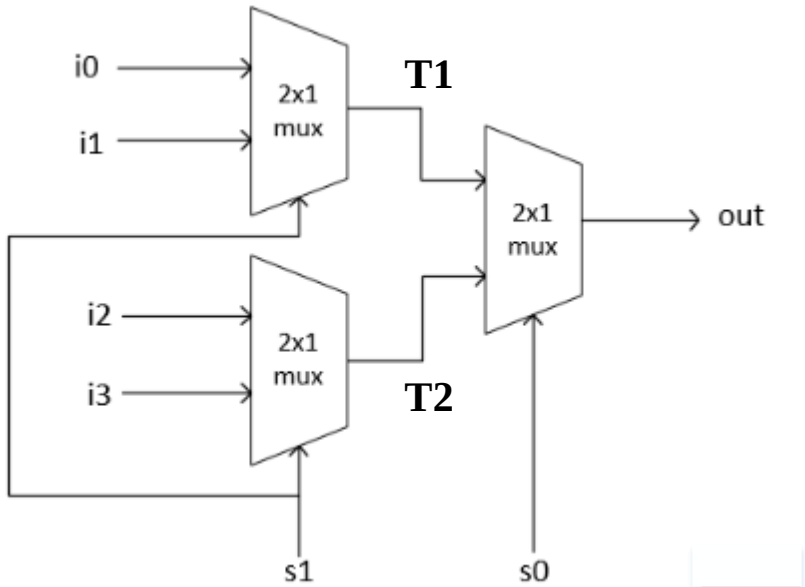
```
module 2X1MUX (I, S, Y);
        input [1:0] I;
        input S;
        output Y;
        assign Y = S ? I[1]: I[0];
endmodule

module 2X1MUX (I, S, Y);
        input [1:0] I;
        input S;
        output Y;
        assign Y = (~S & I0)|(S&I1)
endmodule
```

PROBLEM 9.3: 2 X 1 MUX (BEHAVIORAL MODELING)

Verilog Program:

```
module 2X1MUX (I, S, Y);
        input [1:0] I;
        input S;
        output reg Y;
        always @ (I or S)
                if (S == 1'b1) begin Y = I1; end
                else begin Y = I0; end
endmodule

module 2X1MUX (I, S, Y);
       input [1:0] I;
       input S;
       output Y;
       assign Y = I[S];
endmodule
```

PROBLEM 9.4: 4 X 1 MUX using 2 X 1 MUX



Verilog Program:

```
module 2X1MUX (I, S, Y);
        input [1:0] I;
        input S;
        output Y;
        assign Y = I[S];
endmodule
module 4X1MUX (I, S, OUT);
         input [3:0] I;
         input [1:0] S;
         output OUT;

         2X1MUX M1 (I[1:0], S[1], T1);
         2X1MUX M1 (I[1:0], S[1], T2);
         2X1MUX M1 (T1, T0, S[0], OUT);

endmodule
```

PROBLEM 9.5: 4 X 1 MUX (STRUCTURAL and DATA FLOW and BEHAVIORAL)

PROBLEM 9.6: 8 X 1 MUX (DATA FLOW and BEHAVIORAL)

Verilog Program:

```
module MUX8X1 (In, Sel, Out);
	input [7:0] In;
	input [2:0] Sel;
	output reg Out;
	always@(*)
		begin
			case (Sel)
				3'b000: Out = In[0];
				3'b001: Out = In[1];
				3'b010: Out = In[2];
				3'b011: Out = In[3];
				3'b100: Out = In[4];
				3'b101: Out = In[5];
				3'b110: Out = In[6];
				3'b111: Out = In[7];
				default: Out = 1'bx;
			endcase
		end
endmodule
```

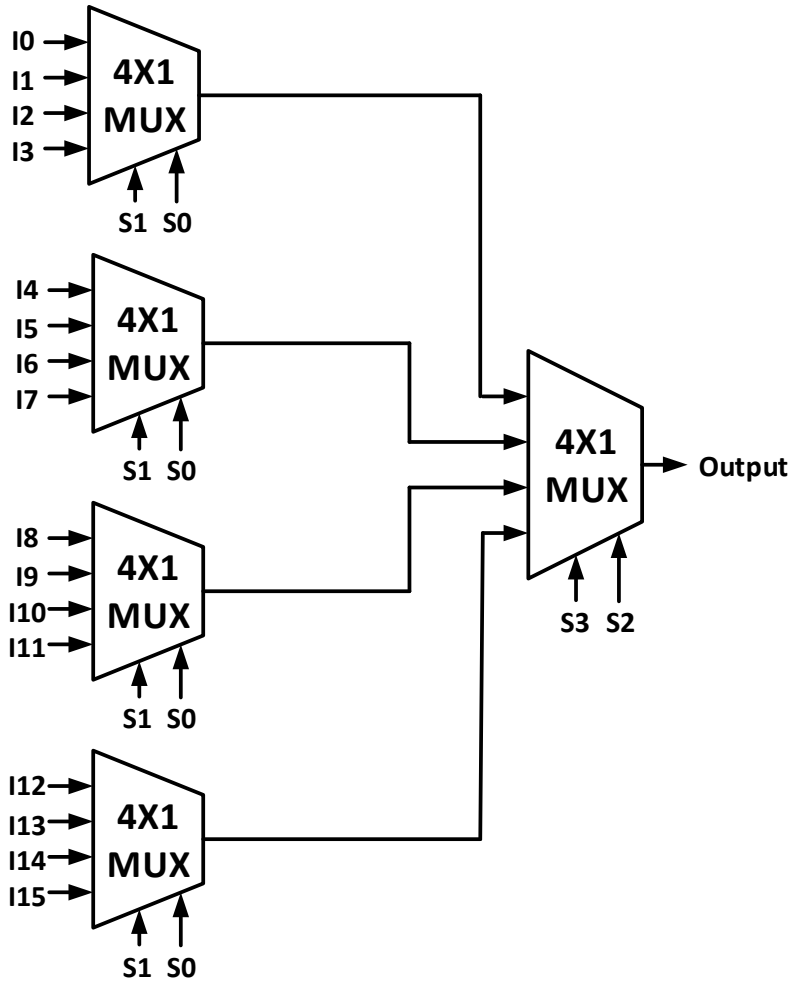
PROBLEM 9.7: 16 X 1 MUX (DATA FLOW)

PROBLEM 9.8: 16 X 1 MUX (BEHAVIORAL)

Verilog Program:

```
module MUX16X1 (In, Sel, Out);
	input [15:0] In;
	input [3:0] Sel;
	output Out;
	assign Out = In [Sel];
endmodule
```

PROBLEM 9.9: 16 X 1 MUX using 4X1 MUX



Verilog Program:

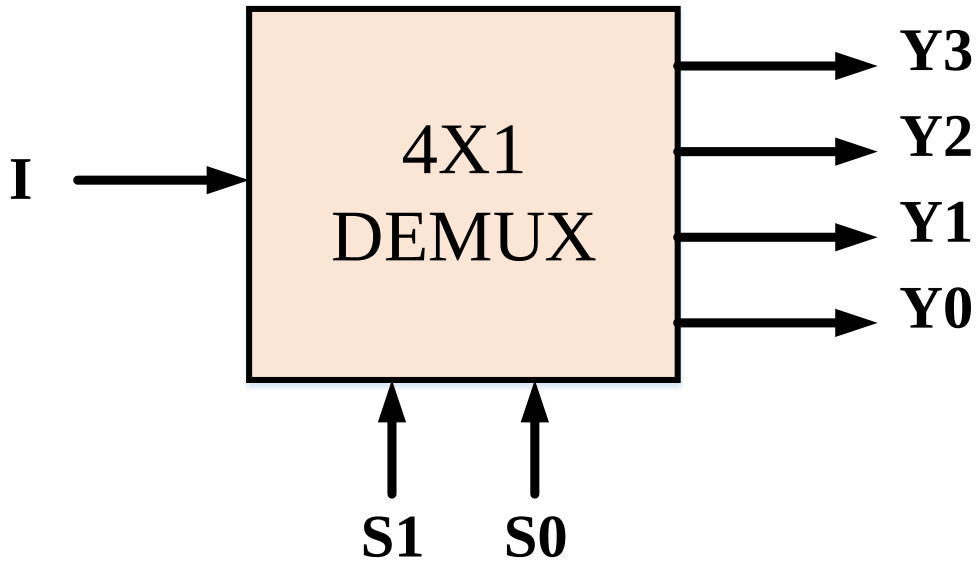
```
module MUX4X1 (In, Sel, Output);
        input [3:0] In;
        input [1:0] Sel;
        output Output;
        assign Output = In [Sel];
endmodule

module MUX16X1 (In, Sel, Output);
        input [15:0] In;
        input [3:0] Sel;
        output Output;
        wire [3:0] t;
        MUX4X1 M0 (In[3:0], Sel[1:0], t[0]);
        MUX4X1 M1 (In[7:4], Sel[1:0], t[1]);
        MUX4X1 M2 (In[11:8], Sel[1:0], t[2]);
        MUX4X1 M3 (In[15:12], Sel[1:0], t[3]);
        MUX4X1 M4 (t, Sel[3:2], Output);
endmodule
```

PROBLEM 9.10: 16 X 1 MUX using 8X1 MUX

PROBLEM 9.11: 32 X 1 MUX (DATA FLOW and BEHAVIORAL)

CATEGORY-10: DEMULTIPLEXER

- DE multiplexer is a combinational circuit that performs reverse operation of the multiplexer.
- It has ***1*** input, ***n*** selection line, and 2^n output lines.
- The input will be connected to one of the output lines based on the value of the selection line.



I	S1	S0	Y3	Y2	Y1	Y0
I	0	0	0	0	0	I
I	0	1	0	0	I	0
I	1	0	0	I	0	0
I	1	1	I	0	0	0

Verilog Program:

```
module DEMUX4X1 (I, S, Y);
        input I;
        input [1:0] S;
        output reg [3:0] Y;
        always @(S)
        begin
        case (S)
                2'b00: Y = {3'b000, I};
                2'b01: Y = {2'b00, I, 1'b0};
                2'b10: Y = {1'b0, I, 2'b00};
                2'b11: Y = {I, 3'b000};
                default: Y = 4'bzzzz;
        endcase
        end
endmodule
```