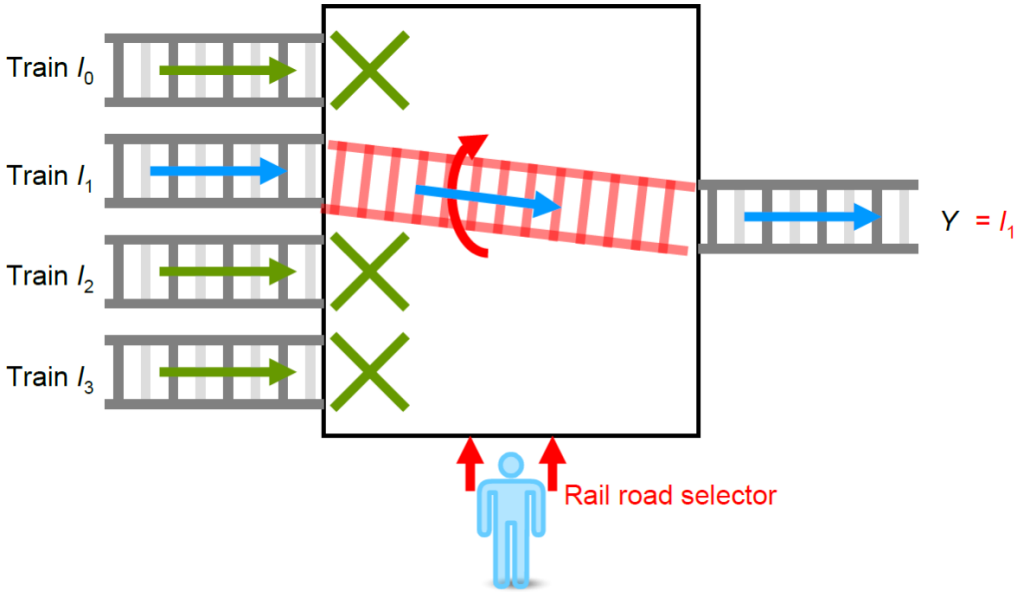
EE2000 Logic Circuit Design

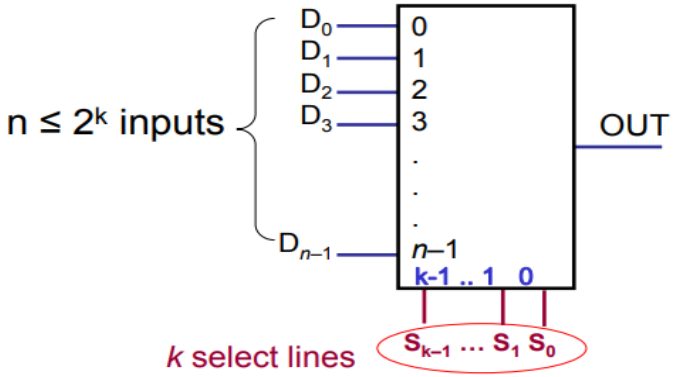
Lecture 5 – Combinational Functional Blocks

electronicscoach.com

Multiplexer (MUX)

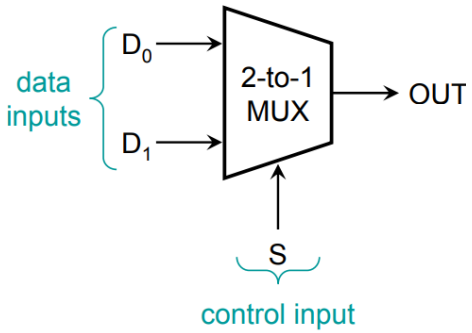
- Basically a digital switch
- Pass one of the inputs to the output
- Selected by the control input.





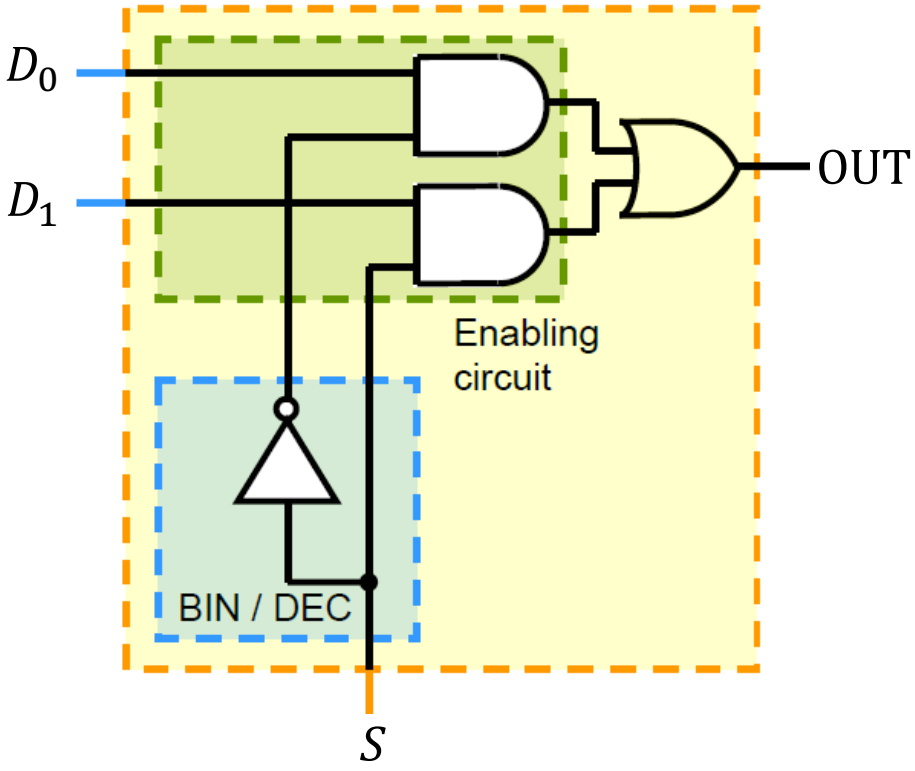
- ***k*** control inputs
- ***n*** data inputs, $n \leq 2^k$
- 1 output

2-to-1 Multiplexer (MUX)



Input			Output
S	D_1	D_0	
0	x	x	D_0
1	x	x	D_1

$$\text{OUT} = S'D_0 + SD_1$$

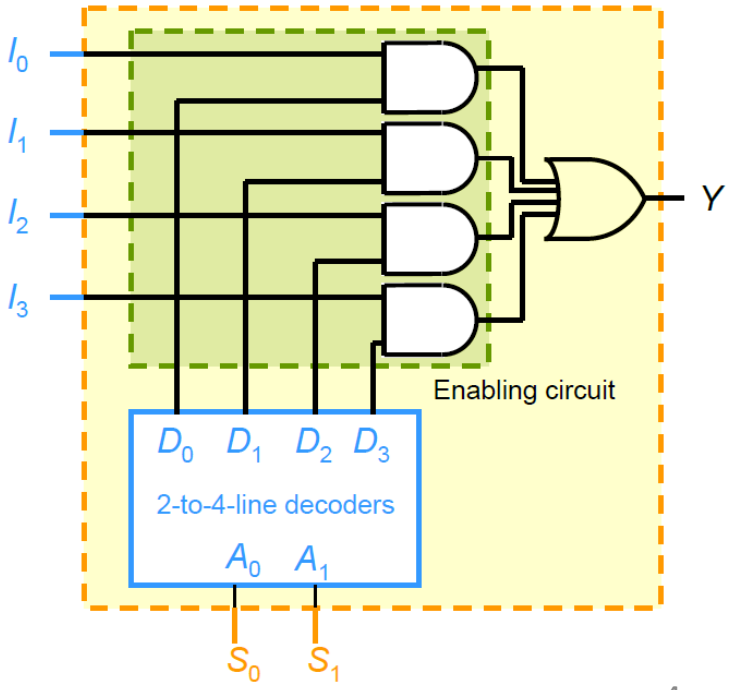


4-to-1 Multiplexer (MUX)

- Specification:
  - $m = 4$
  - $n = \log_2 m = 2$
- Formulation:

Inputs						Output
I_0	I_1	I_2	I_3	S_1	S_0	Y
x	x	x	x	0	0	I_0
x	x	x	x	0	1	I_1
x	x	x	x	1	0	I_2
x	x	x	x	1	1	I_3

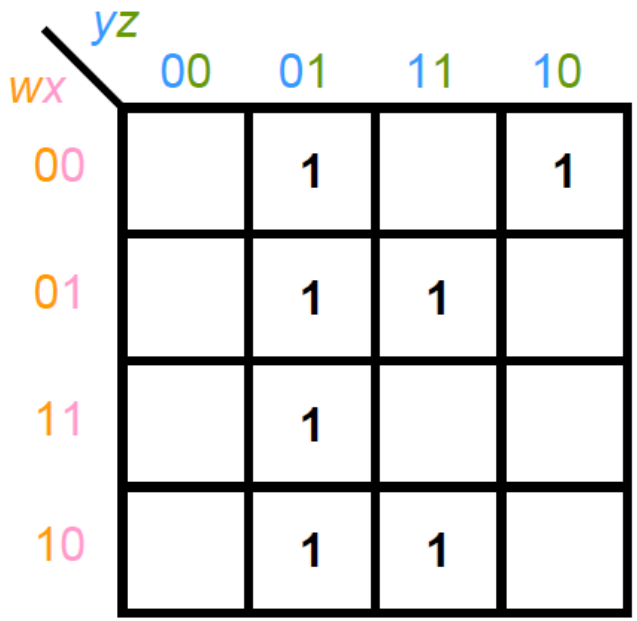
- Optimization:
  - $Y(I_0,I_1,I_2,I_3,S_1,S_0) = S_1'S_0'I0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$



Example

Realize the function $f(w, x, y, z) = \sum m(1, 2, 5, 7, 9, 11, 13)$ using a 4-to-1 MUX

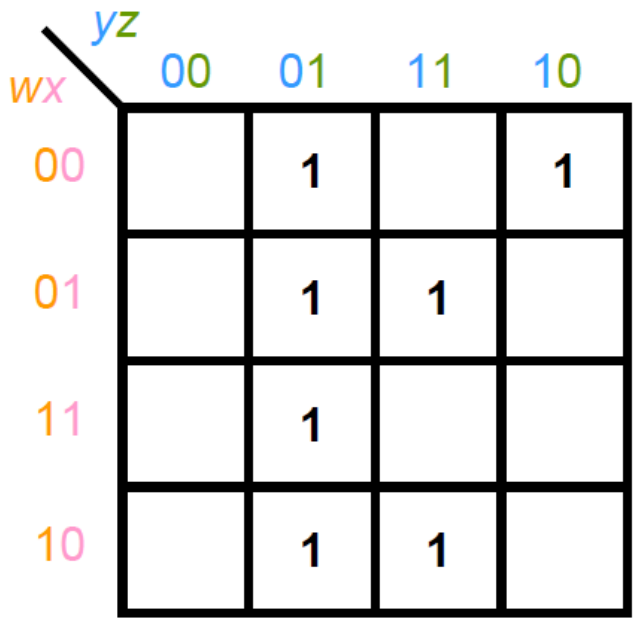
STEP 1: Plot the K-map



Example

STEP 2: Since 4-to-1 MUX has 2 control inputs, choose two variables for these inputs

$$w = S_1 \quad x = S_0$$



STEP 3:

$$f(w = 0, x = 0) = y'z + yz'$$

$$f(w = 0, x = 1) = z$$

$$f(w = 1, x = 1) = y'z$$

$$f(w = 1, x = 0) = z$$

Example

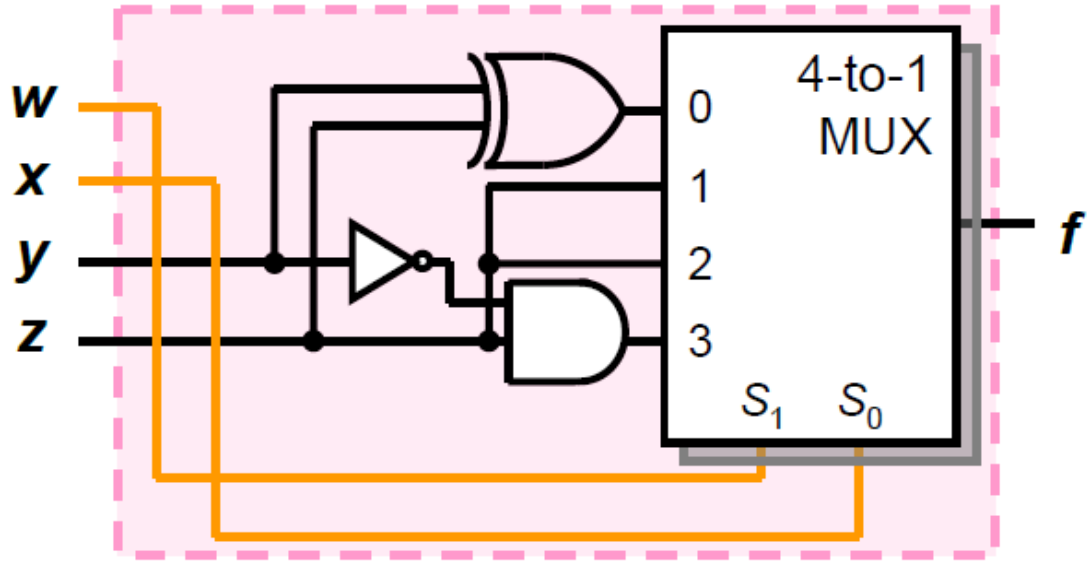
$w = S_1 \quad x = S_0$

$f(w = 0, x = 0) = y'z + yz'$

$f(w = 0, x = 1) = z$

$f(w = 1, x = 1) = y'z$

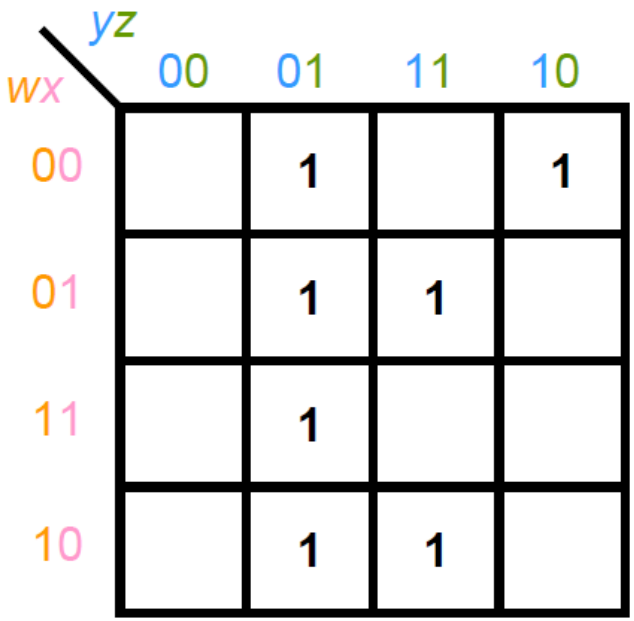
$f(w = 1, x = 0) = z$



Question: How can we realize the function using 2-to-1 muxs?

Exercise

Realize the function $f(w, x, y, z) = \sum m(1, 2, 5, 7, 9, 11, 13)$ using a 2-to-1 MUX



Exercise

Realize the function $f(w, x, y, z) = \sum m(1, 2, 5, 7, 9, 11, 13)$ using an 8-to-1 MUX

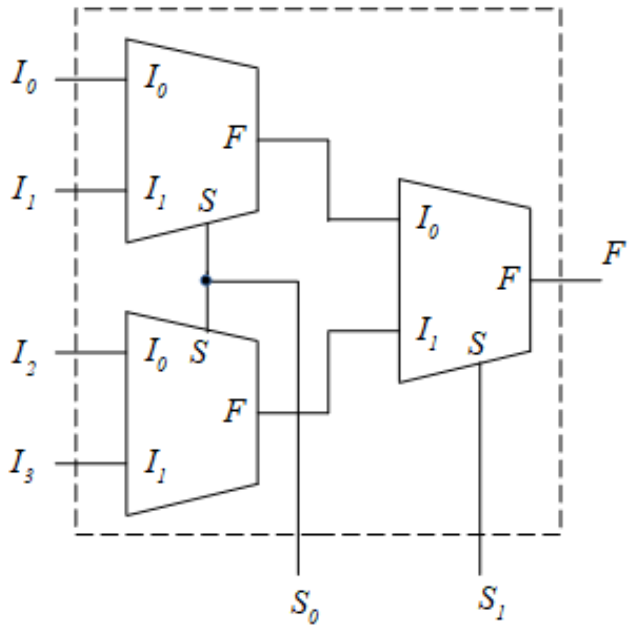
wxyz	*F*
0000	0
0001	1
0010	1
0011	0
0100	0
0101	1
0110	0
0111	1

wxyz	*F*
1000	0
1001	1
1010	0
1011	1
1100	0
1101	1
1110	0
1111	0

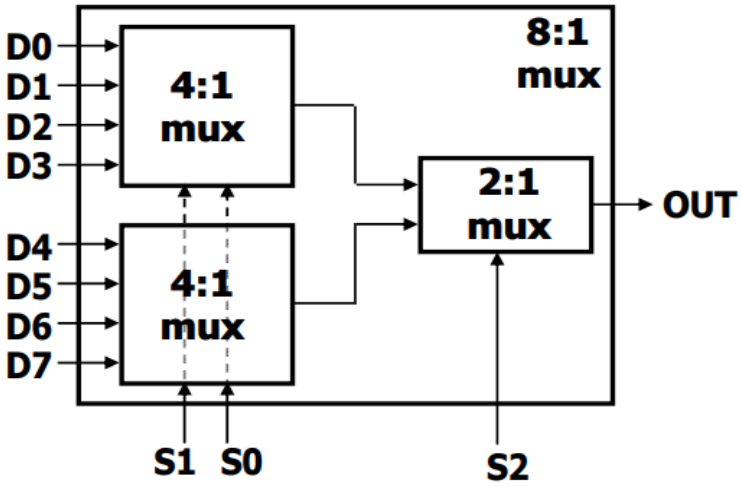
Summary

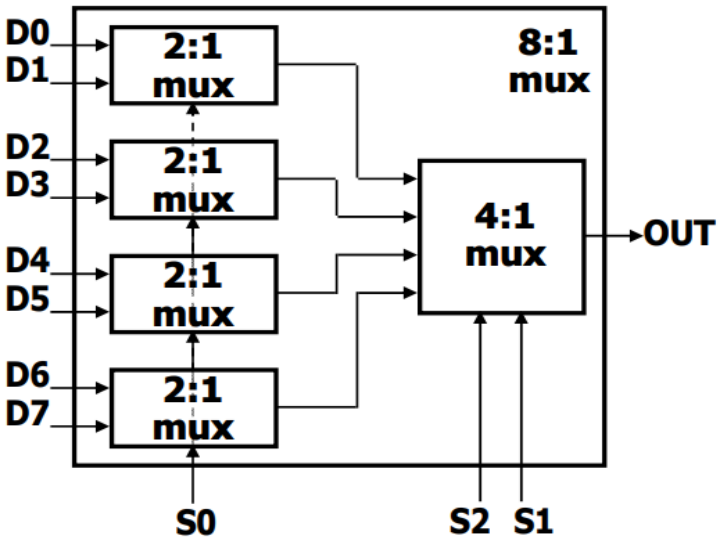
	With	2-to-1 Mux	4-to-1 Mux	8-to-1 Mux
wxyz	*F*	$S_0=w$	$S_1=w, S_0=x$	$S_2=w, S_1=x, S_0=y$
0000	0	$I_0=xz+y'z+x'yz'$	$I_0 = y \oplus z$	$I_0=z$
0001	1			$I_1=z'$
0010	1		$I_1=z$	$I_2=z$
0011	0			$I_3=z$
0100	0		$I_2=z$	$I_4=z$
0101	1			$I_5=z$
0110	0		$I_3=y'z$	$I_6=z$
0111	1			$I_7=0$
1000	0	$I_1=x'z+y'z$		
1001	1			
1010	0			
1011	1			
1100	0			
1101	1			
1110	0			
1111	0			

Cascading Multiplexers



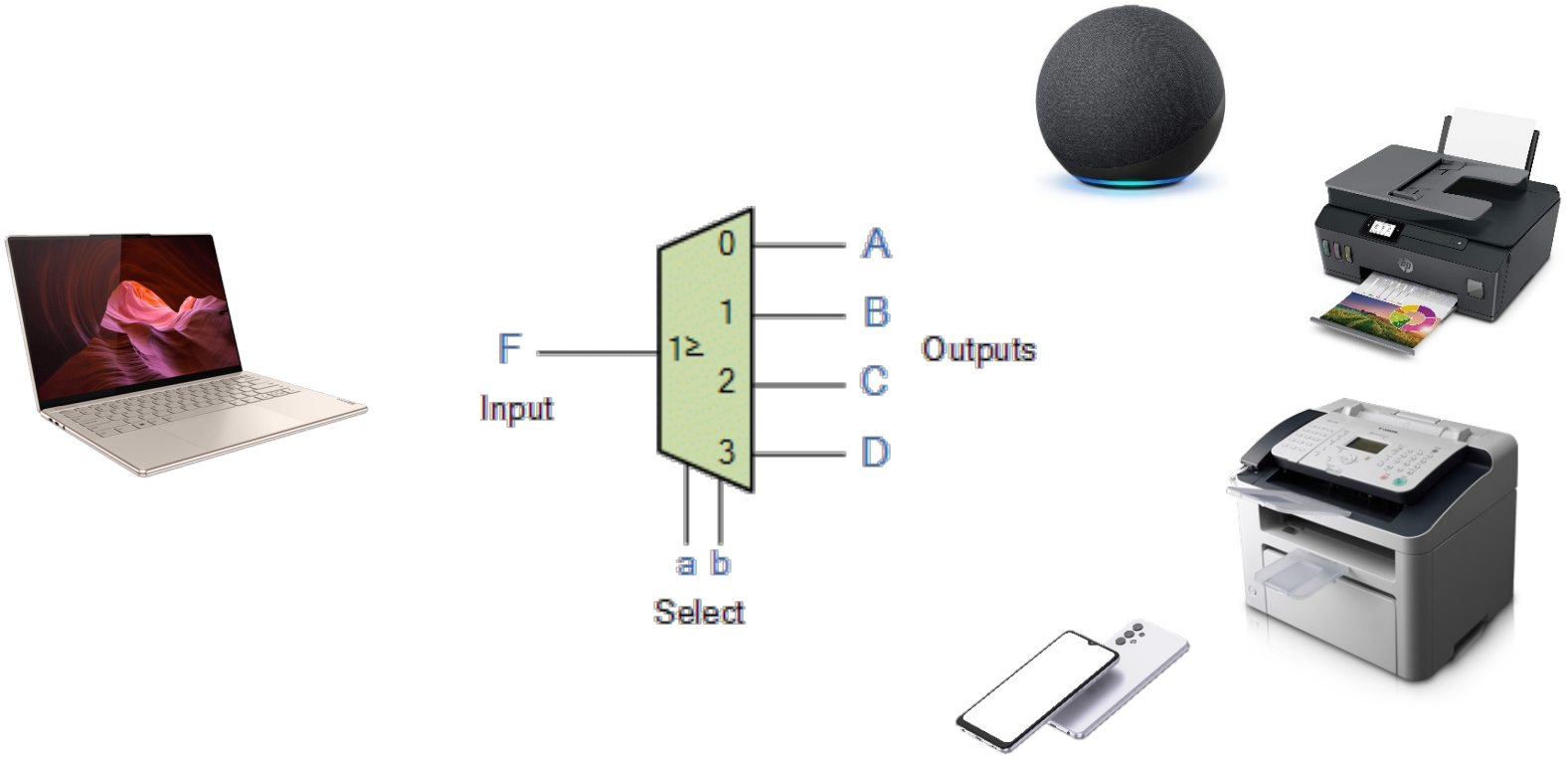
Inputs						Output
I_0	I_1	I_2	I_3	S_1	S_0	Y
x	x	x	x	0	0	I_0
x	x	x	x	0	1	I_1
x	x	x	x	1	0	I_2
x	x	x	x	1	1	I_3





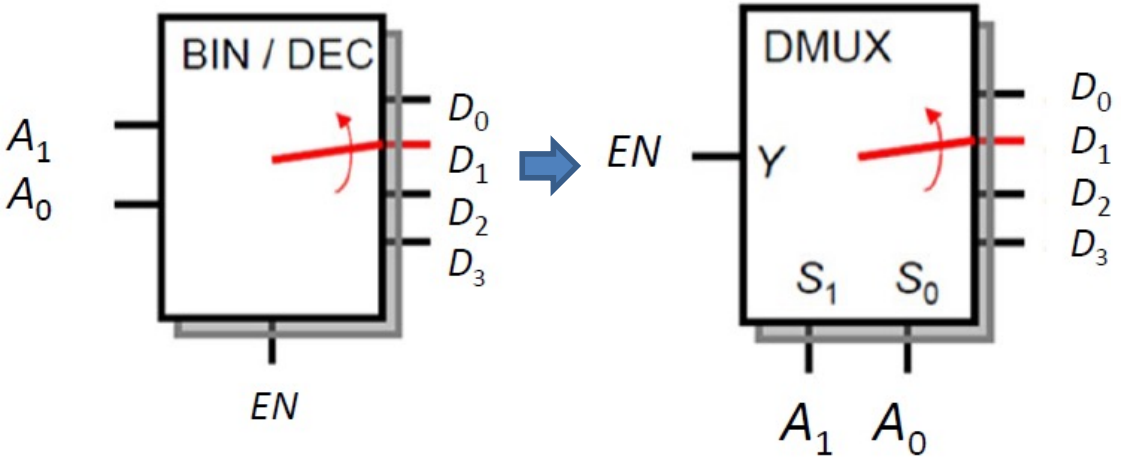
Demultiplexer (DMUX)

- Reverse the function of MUX
- Route a single input to one of the many outputs
- Selected by the control input.



Demultiplexer (DMUX)

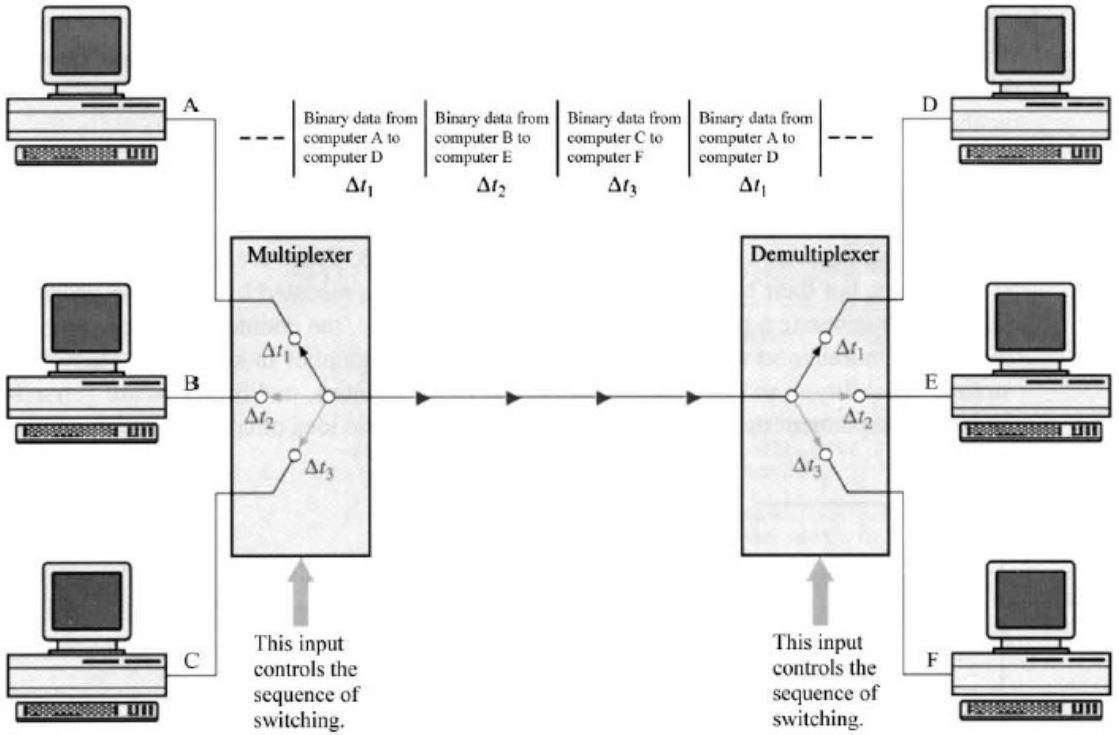
- Remember the decoder with Enable?
- The decoder can perform demultiplexer if we take EN as the input line, A_i (input lines of decoder) as the selection inputs



Input	Select lines		Output			
EN	A_1	A_0	D_3	D_2	D_1	D_0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	X	X	0	0	0	0

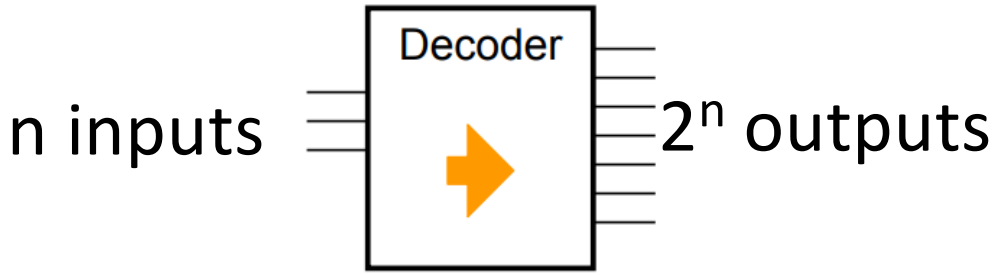
Example of MUX & DMUX Application

- A MUX allows digital information from several sources to be routed onto a single line for transmission over that line to a common destination
- A DMUX basically reverses the multiplexing function

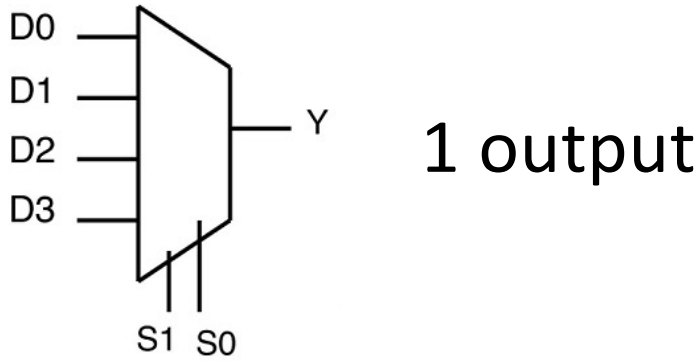


Summary

Only 1 output is ‘1’

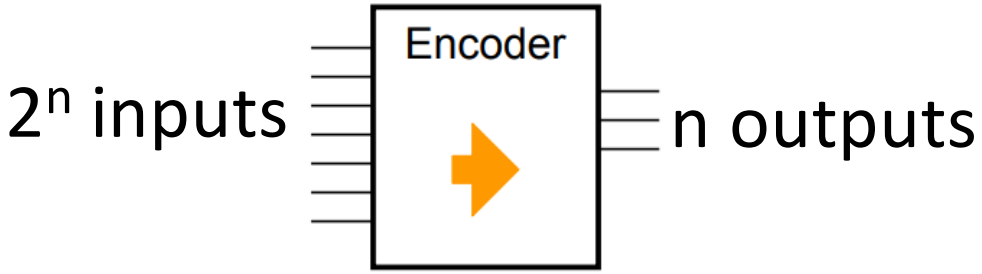


Binary Decoder

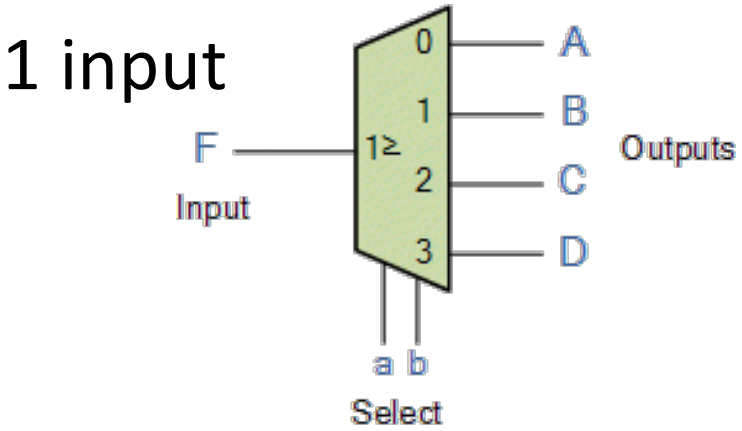


MUX

Only 1 input is ‘1’



Binary Encoder



DMUX

Lab Session 2

Objectives

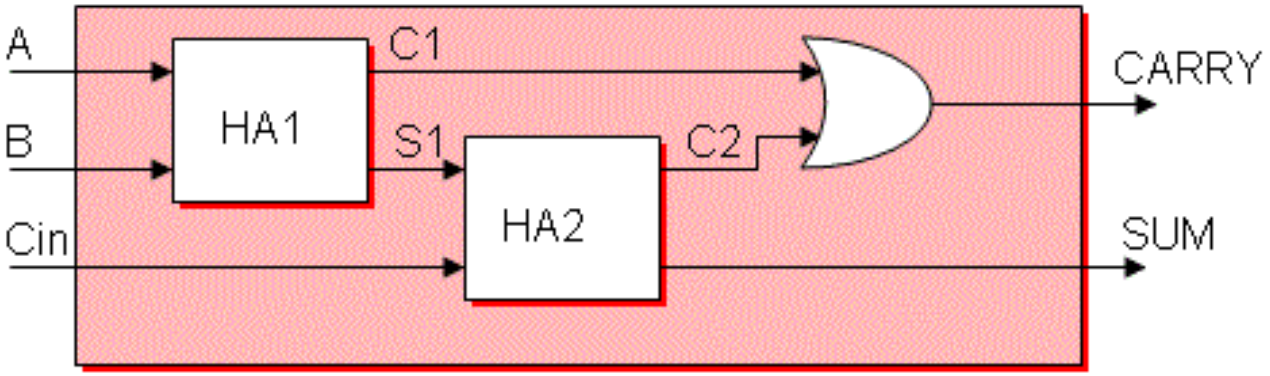
- Learn the modular design flow
- Implement a **1-bit Full-Adder** using VHDL
- Implement a **4-bit Full-Adder** using VHDL

1-bit Full-Adder

Input			Output	
i_Cin	i_A	i_B	o_S	o_Cout
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

Components and Instantiation

- **Structural modeling**: Modular design of a complex project
- When designing a complex project, we can split it into two or more simple designs (sub-modules/sub-circuits/**components**)
- Example: A full adder (FA) contains of 2 half adders (HAs); Half adder can be modeled by a **component**



Structural Modeling

- **Structural modeling** or modular design allows us to pack low-level functionalities into modules
- Allows a designed module to be **reused** without the need to reinvent and re-test the same functions/modules every time
- To include a **component** into a **module**, we need to

  (1) declare the component

  (2) instantiate the component

  in **architecture**

Component Declaration

- An architecture may contain multiple components and they must be declared first

```
architecture [name] …
[signal]

        component XX        -- Component declaration
        …
        end component;
        component YY        -- Component declaration
        …
        end component;
begin
…               -- Component instantiation
end [name];
```

Half Adder

Create the sub-module of half adder first

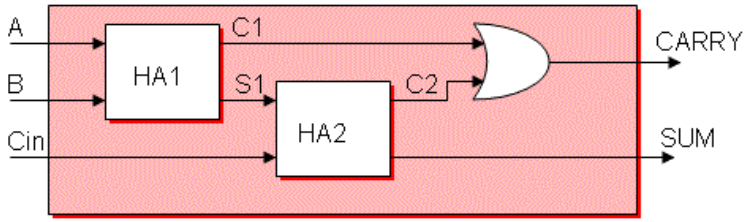
$$\text{sum} = a \oplus b \quad \text{carry} = a \cdot b = ab$$

Inputs		Outputs	
a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

```
--sub module(half adder) entity declaration
entity halfadder is
port (a : in  STD_LOGIC;
      b : in  STD_LOGIC;
      sum : out  STD_LOGIC;
      carry : out  STD_LOGIC
     );
end halfadder;

architecture Behavioral of halfadder is
begin
sum <= a xor b;
carry <= a and b;
end Behavioral;
```

Full Adder

```
--top module(full adder) entity declaration
entity fulladder is
    port (a : in std_logic;
          b : in std_logic;
          cin : in std_logic;
          sum : out std_logic;
          carry : out std_logic
         );
end fulladder;

--top module architecture declaration
architecture behavior of fulladder is

   component halfadder
    port(
         a : in std_logic;
         b : in std_logic;
         sum : out std_logic;
         carry : out std_logic
        );
    end component;
```



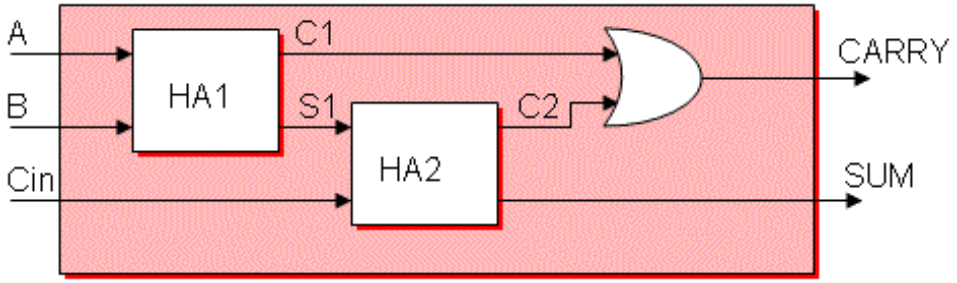
--sub-module(half adder) is declared as a component before the keyword "begin"

Component Instantiation

Differences between a **component** and an **entity** declaration:

- **Entity** declaration declares a circuit model containing one or multiple architectures
- **Component** declaration declares a **virtual circuit** template, which must be **instantiated** to take effect during the design
- **Instantiation** – To map the signals in the entity with the input/output of the component
- **Port map** is required for component **instantiation**

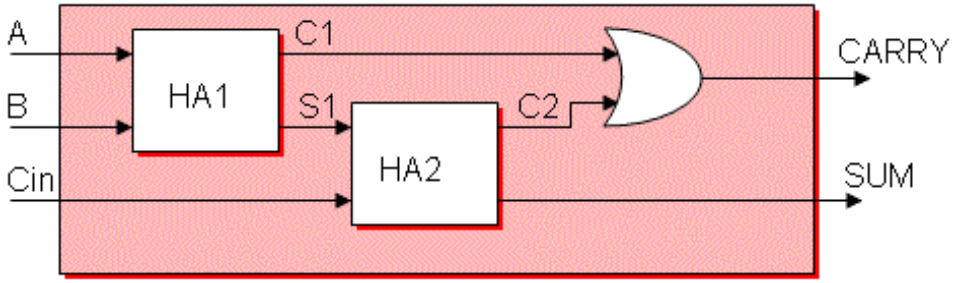
Full Adder



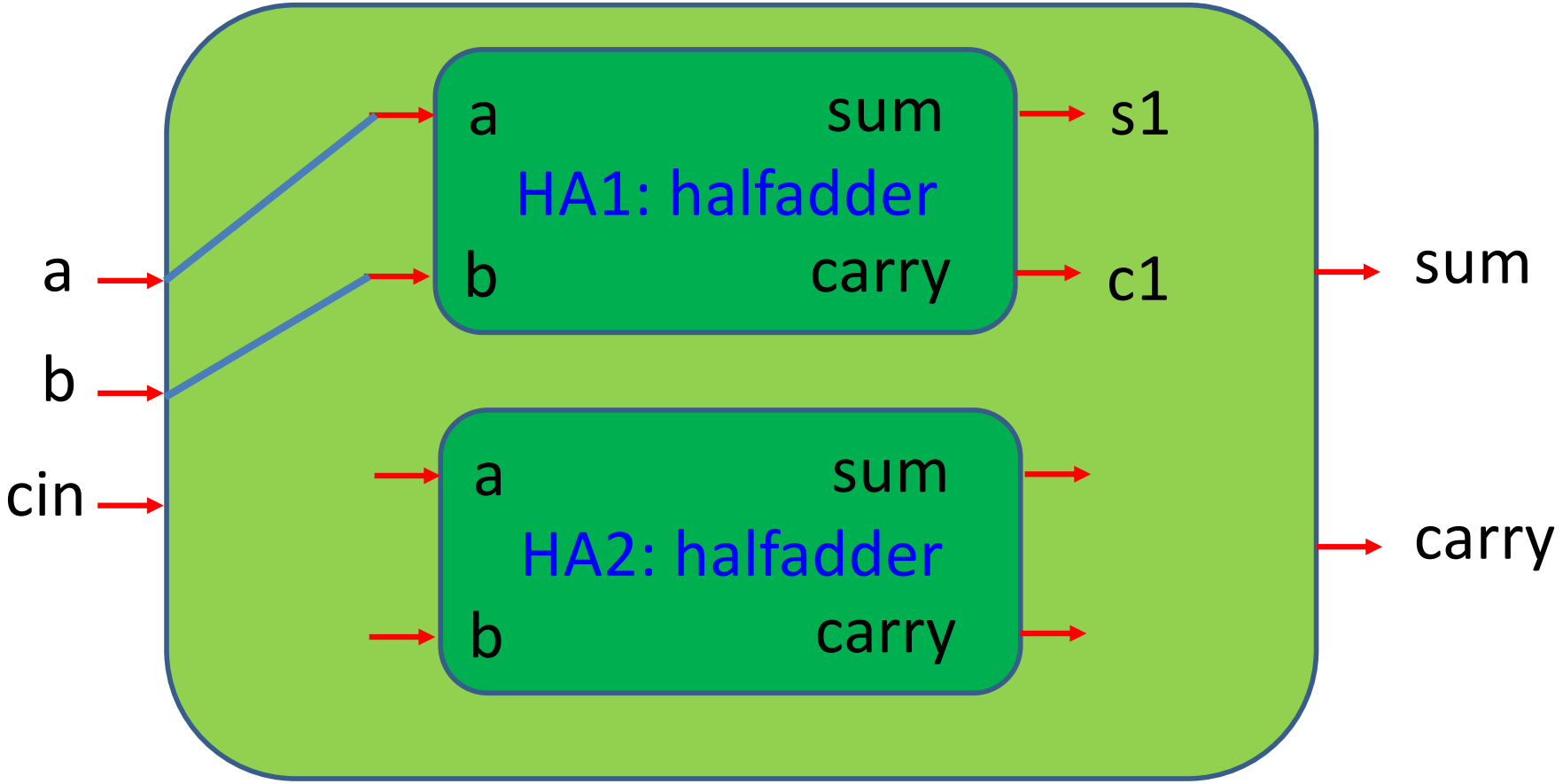
- Two HAs are needed
- Internal signals s1,c1,c2 are used to connect the two HAs
- In HA, we define **port** (a:**in** STD_LOGIC; b:**in** STD_LOGIC; sum:**out** STD_LOGIC; carry:**out** STD_LOGIC);

```
signal s1,c1,c2 : std_logic:='0';  --declare internal signal

begin

--Provide a different name for each half adder.
--instantiate and do port map for the half adders.

HA1 : halfadder port map (a,b,s1,c1);
HA2 : halfadder port map (s1,cin,sum,c2);
carry <= c1 or c2;  --final carry calculation

end;
```

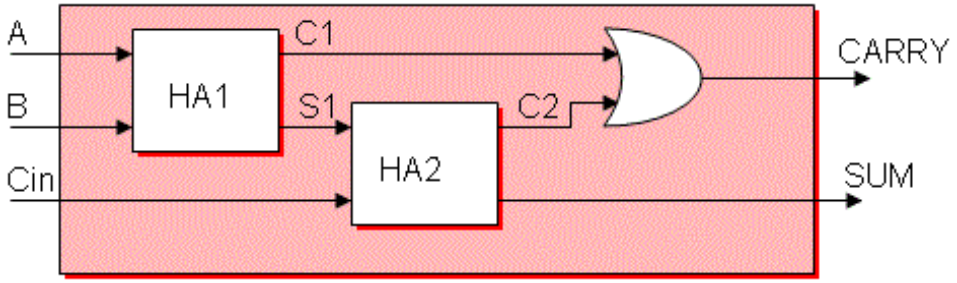
Half-Adder 1



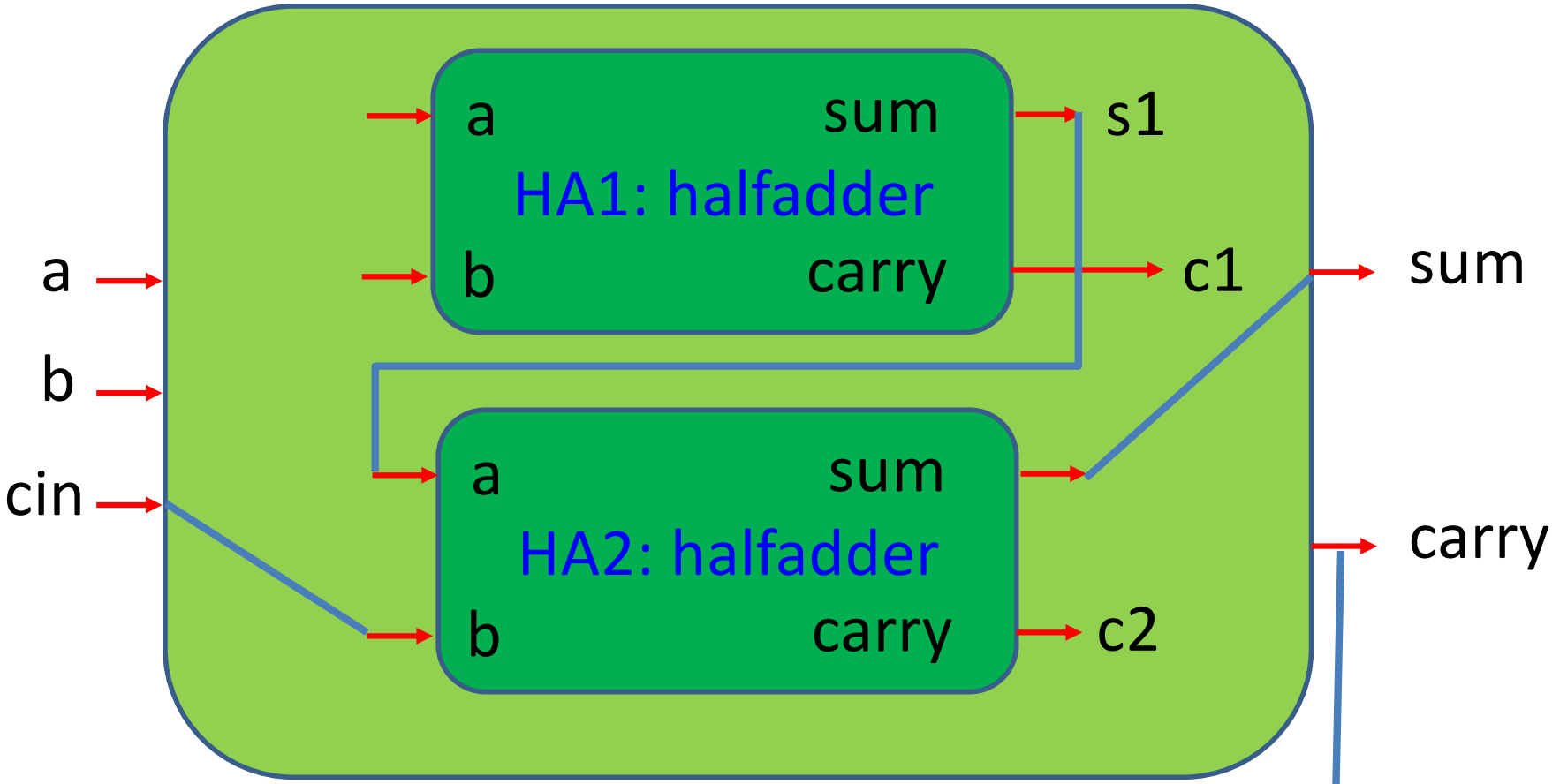

```
HA1 : halfadder port map (a,b,s1,c1);
```



Half-Adder 2




```
HA2 : halfadder port map (s1,cin,sum,c2);
```



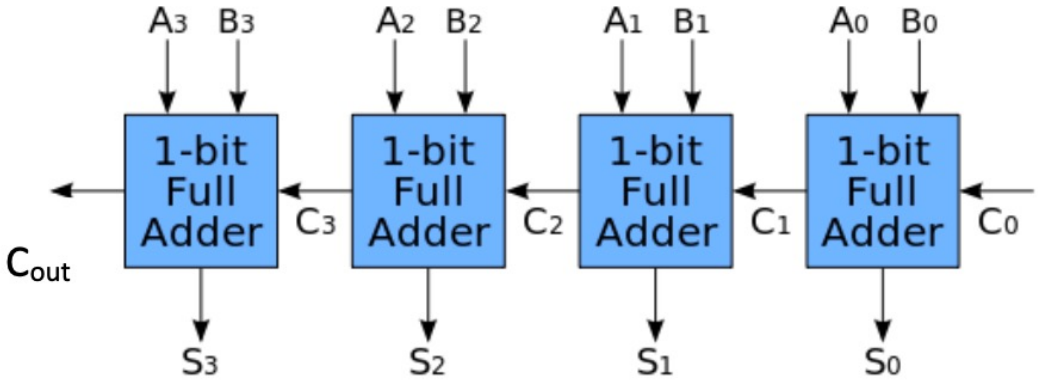

```
carry <= c1 or c2;
```

1-bit Full-Adder

Input			Output	
i_Cin	i_A	i_B	o_S	o_Cout
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

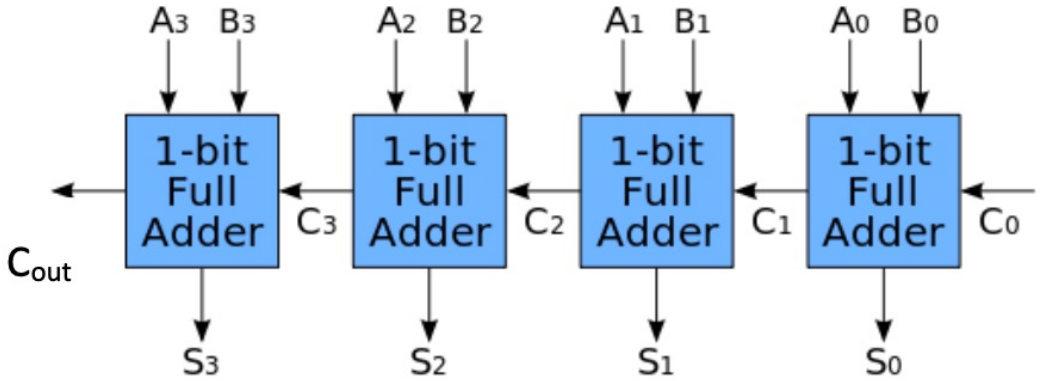
```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;

Entity full_adder_1bit is
   Port (
      i_A: in STD_LOGIC;
      i_B: in STD_LOGIC;
      i_Cin: in STD_LOGIC;
      o_S: out STD_LOGIC;
      o_Cout: out STD_LOGIC);
End full_adder_1bit;

Architecture Behavioral of full_adder_1bit is
Begin
   --Add Your own code here
End Behavioral;
```

4-bit Full-Adder




```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;

Entity full_adder_4bits is
   Port (
      i_A: in STD_LOGIC_VECTOR (3 DOWNTO 0);
      i_B: in STD_LOGIC_VECTOR (3 DOWNTO 0);
      i_Ci: in STD_LOGIC;
      o_S: out STD_LOGIC_VECTOR (3 DOWNTO 0);
      o_Cout: out STD_LOGIC
      );
end full_adder_4bits;
```

4-bit Full-Adder




```
Architecture Behavioral of full_adder_4bits is
   SIGNAL ci: STD_LOGIC_VECTOR (3 DOWNTO 0);

   COMPONENT full_adder_1bit is
      Port (
        i_A: in STD_LOGIC;
        i_B: in STD_LOGIC;
        i_Cin: in STD_LOGIC;
        o_S: out STD_LOGIC;
        o_Cout: out STD_LOGIC
        );
   END COMPONENT;
begin

ci(0) <= i_Ci;
uut0: full_adder_1bit PORT MAP(i_A(0), i_B(0), ci(0), o_S(0), ci(1));
   --Write your own code here to instantiate other three units.
   -- …

end Behavioral;
```

Hardware

Table 12 - Push Button Connections

Signal Name	Subsection	Zynq pin
BTNU	PL	T18
BTNR	PL	R18
BTND	PL	R16
BTNC	PL	P16
BTNL	PL	N15
PB1	PS	D13 (MIO 50)
PB2	PS	C10 (MIO 51)

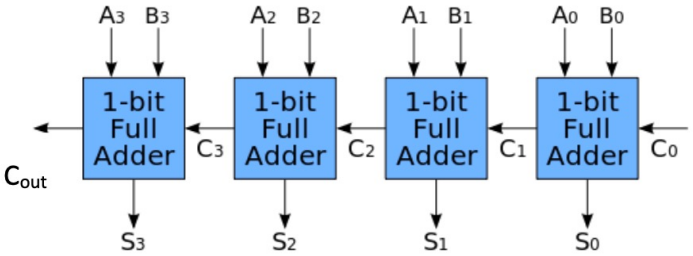


Table 13 - DIP Switch Connections

Signal Name	Zynq pin
SW0	F22
SW1	G22
SW2	H22
SW3	F21
SW4	H19
SW5	H18
SW6	H17
SW7	M15

Table 14 - LED Connections

Signal Name	Subsection	Zynq pin
LD0	PL	T22
LD1	PL	T21
LD2	PL	U22
LD3	PL	U21
LD4	PL	V22
LD5	PL	W22
LD6	PL	U19
LD7	PL	U14
LD9	PS	D5 (MIO7)

- SW0-SW3 as input i_A, SW4-SW7 as input i_B, push button BTNL as input i_Ci
- LD0-LD3 as o_S, LD4 as o_Cout

Results

Run synthesis, implementation and Generate bitstream file

- Program Device
- Observe and verify the result