COMPUTER ORGANIZATION AND ARCHITECTURE (IT 2202)

Lecture 10

Digital Building Blocks

Datapath

Five components of the computer:

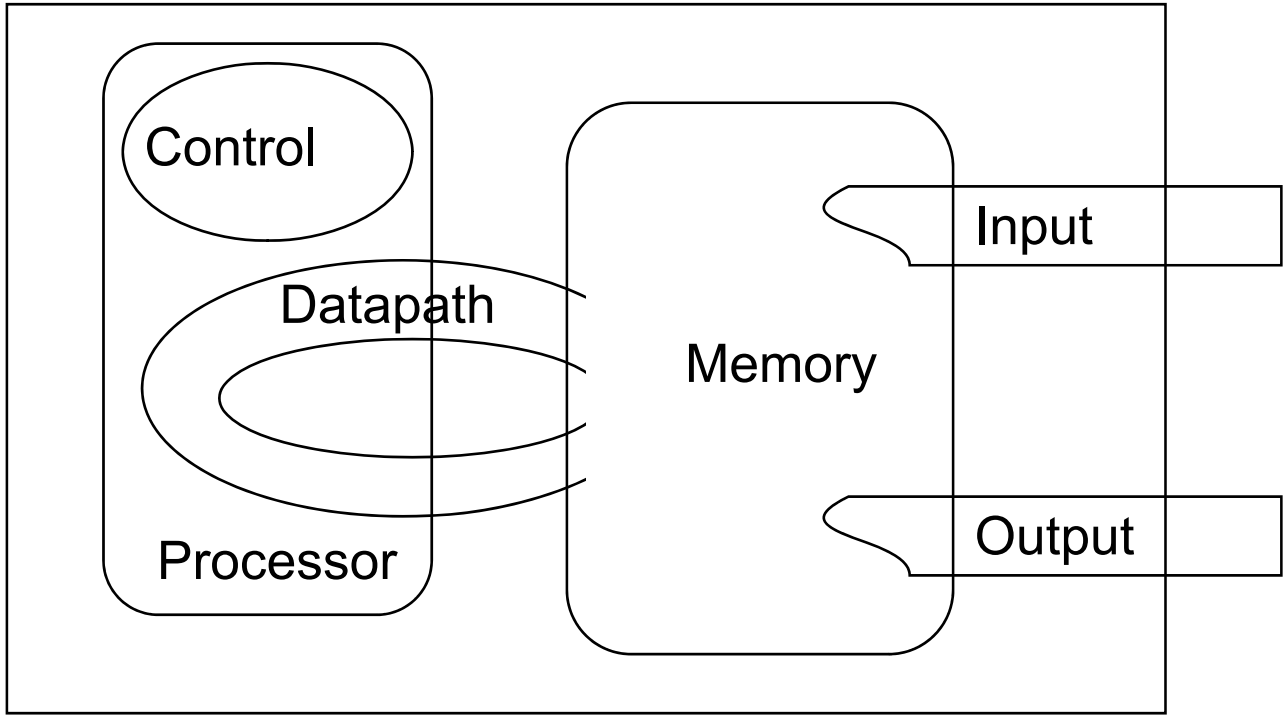
>> Datapath

>> Control

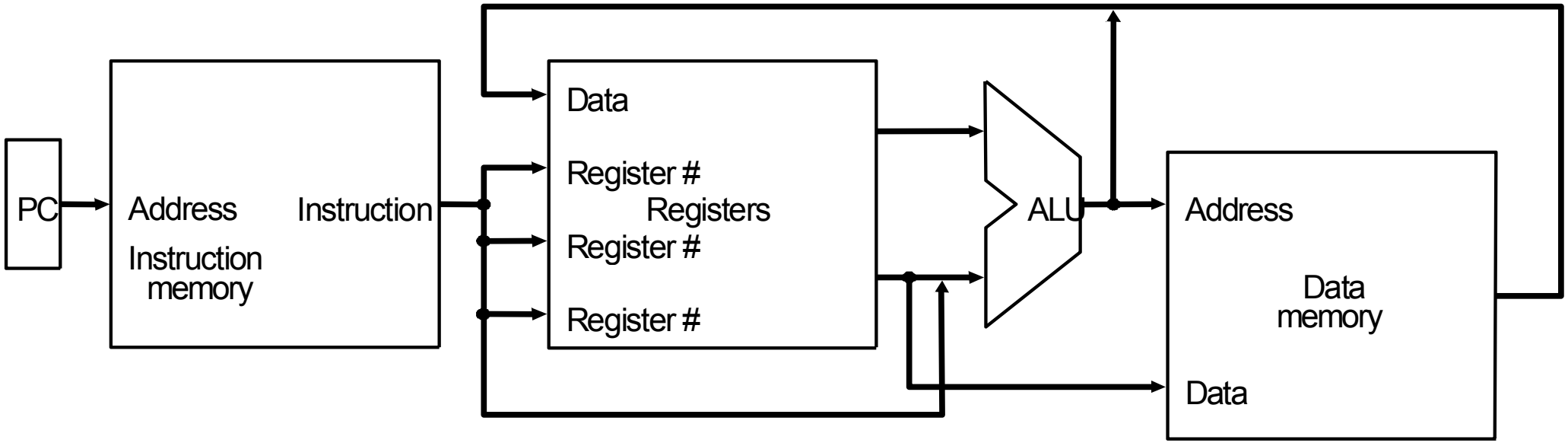
>> Memory

>> Input

>> Output

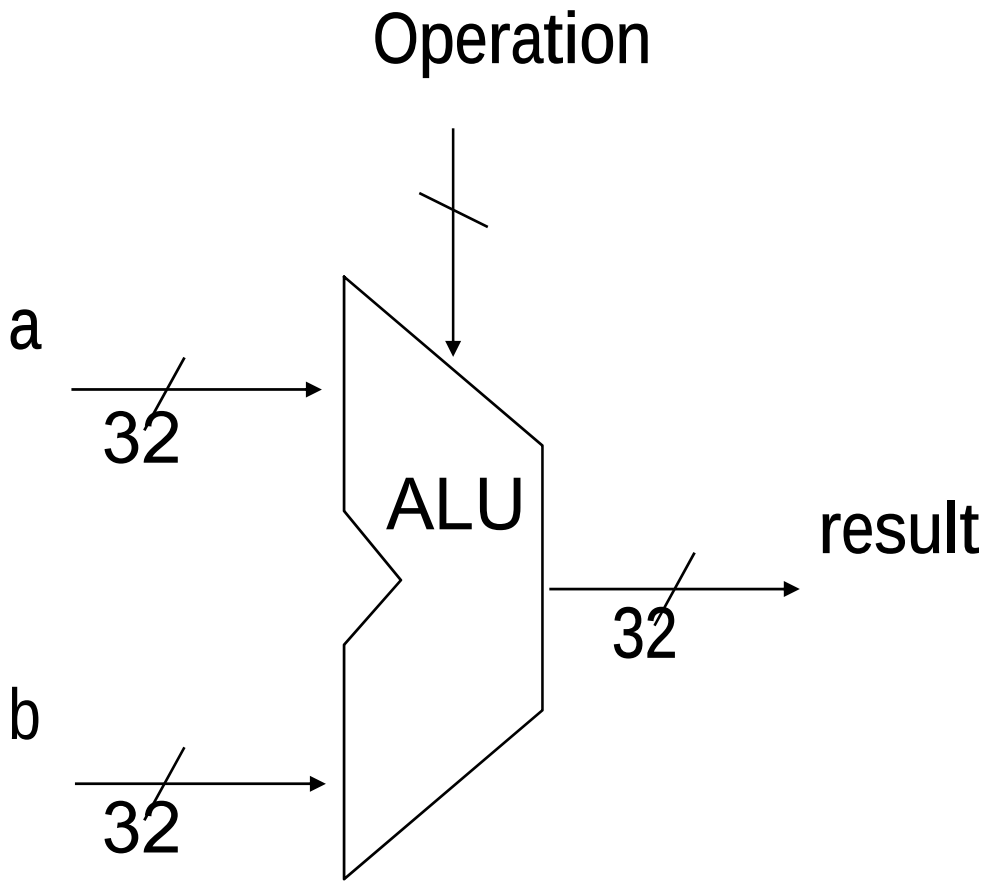


Datapath Overview



Arithmetic

- Arithmetic Circuit: ALU
  - ADDER
  - Subtracter
  - Multiplier
  - Divider



Two's Complement Representation

- Represent "- 3"

	10000 [2^4]	alternatively	1100 [invert 3]
-	0011 [3]		+ 0001 [1]
	1101 [-3]		1101 [-3]

- Representation of -X is nothing but 2^n - X

- Negating a two's complement number (+ve or -ve): invert all bits and add 1

- Subtraction using addition: X-Y = X+Y'+1

Converting n bit Numbers into Numbers with more than n bits

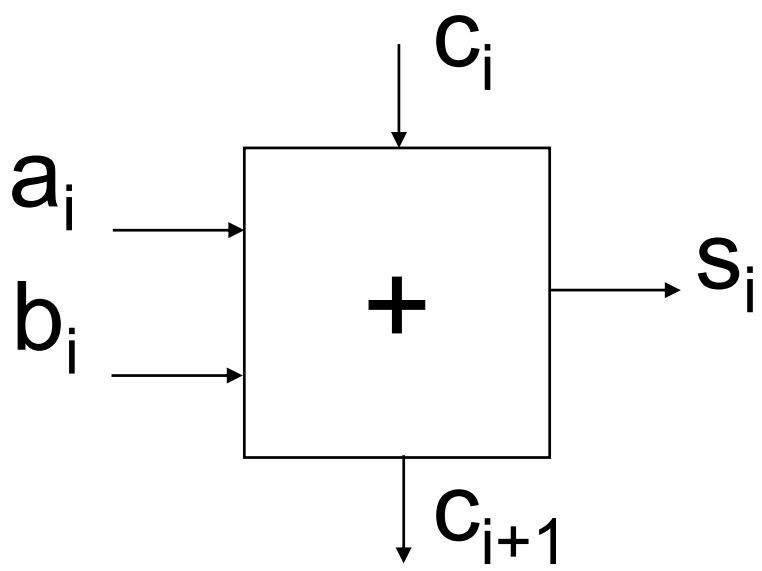
MIPS 16 bit immediate gets converted to 32 bits for arithmetic

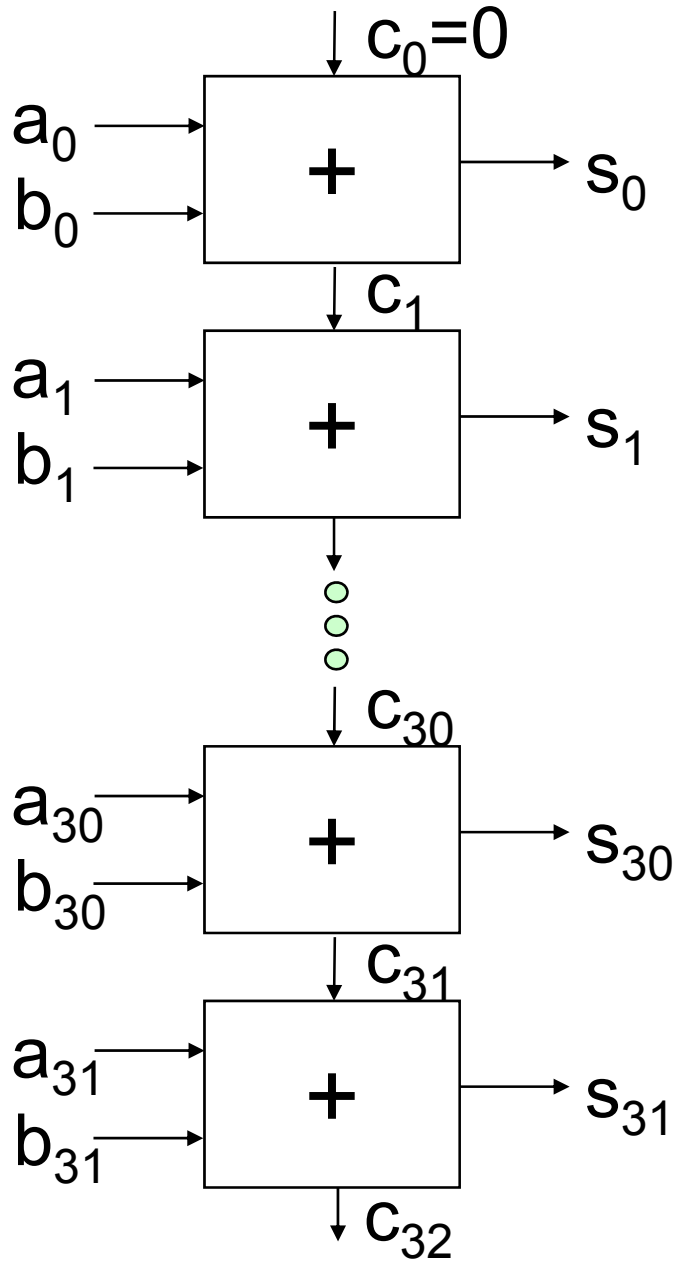
- copy the most significant bit (the sign bit) into the other bits

  0010 $\Rightarrow$ 0000 0010

  1010 $\Rightarrow$ 1111 1010

Adder Circuit





Adder Circuit

Addition of two digits (Bits)

When two bits are added, a sum (S) and a carry (C) are generated as per the following truth table or $S_i = A \oplus B$

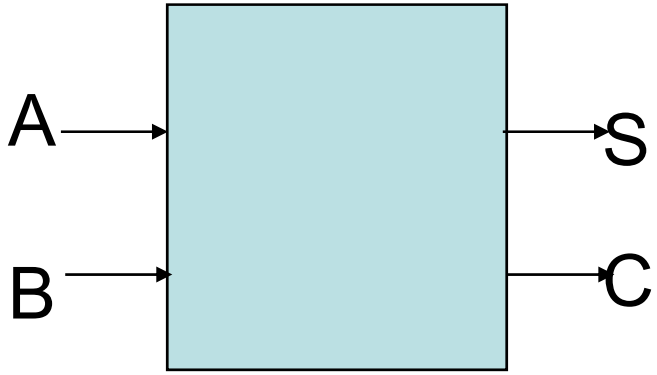
Input		Output	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

0+0=00

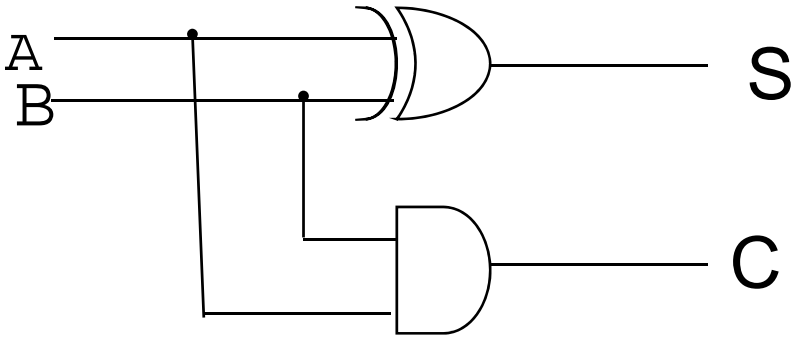
0+1=01

1+0=01

1+1=10



HALF ADDER



$S = A'.B + A.B' = A \oplus B$

$C = A.B$

Adder Circuit

Addition of Multi-bit Binary Numbers

```
  0010110 ←— Carry
  0101011 ←— Number A
+ 0001001 ←— Number B
---------
  0110100 ←— Sum S
```

```
  1111110 ←— Carry
  0111111 ←— Number A
+ 0000001 ←— Number B
---------
  1000000 ←— Sum S
```

- At every bit position (stage), we require to add 3 bits:
  - 1 bit for number A
  - 1 bit for number B
  - 1 carry bit coming from the previous stage

WE NEED A FULL ADDER

Adder Circuit

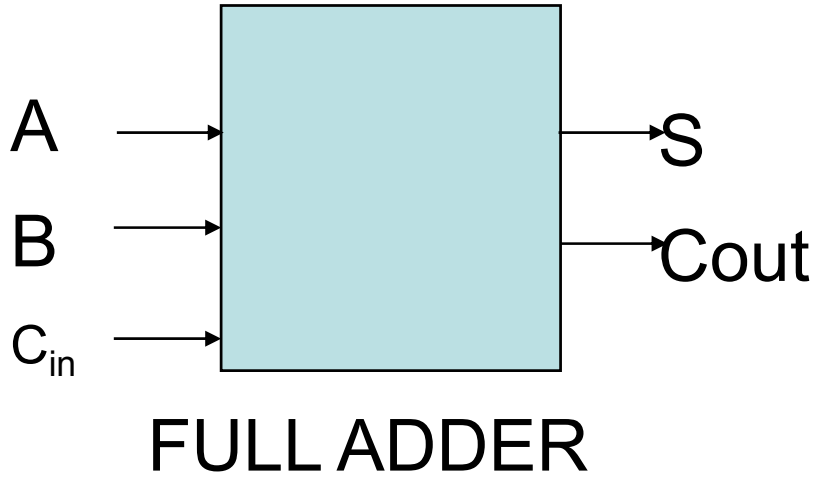
Addition of 3-bit Binary Numbers

Input			Output	
A	B	Cin	S	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

Adder Circuit

Addition of 3-bit Binary Numbers

Input			Output	
A	B	Cin	S	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



$S = A' B' C_{in} + A'.B.C'_{in} + A.B'C'_{in} + A.B.C_{in}$

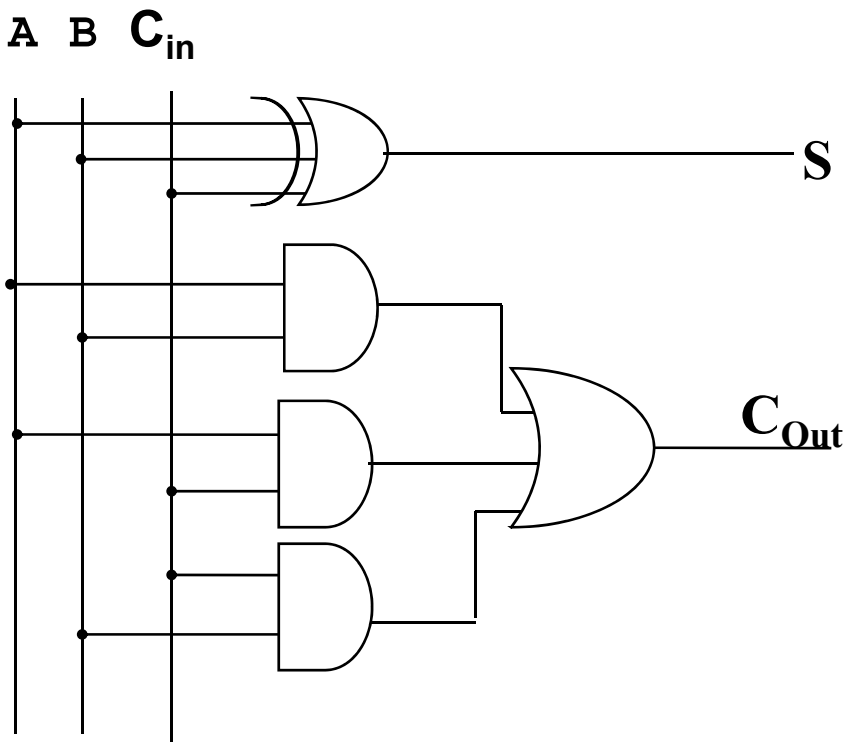
$= A \oplus B \oplus C_{in}$

$C = B.C_{in} + A.C_{in} + A.B + A.B.C_{in}$

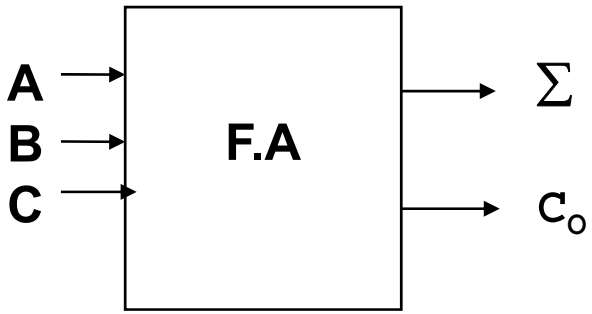
$= AB + B\,C_{in} + A.C_{in}$

Adder Circuit

Various Implementations of Full Adder



Full Adder Circuit realization



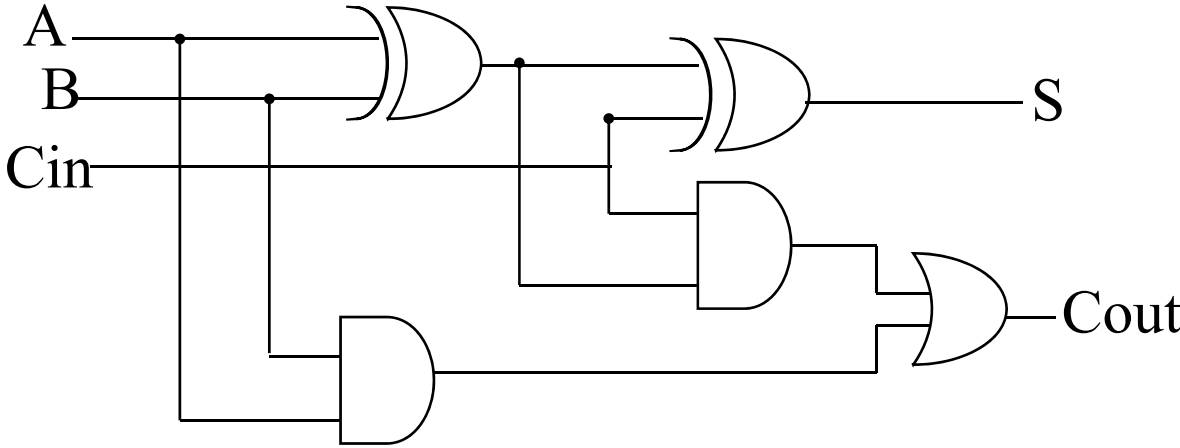
Full Adder symbol

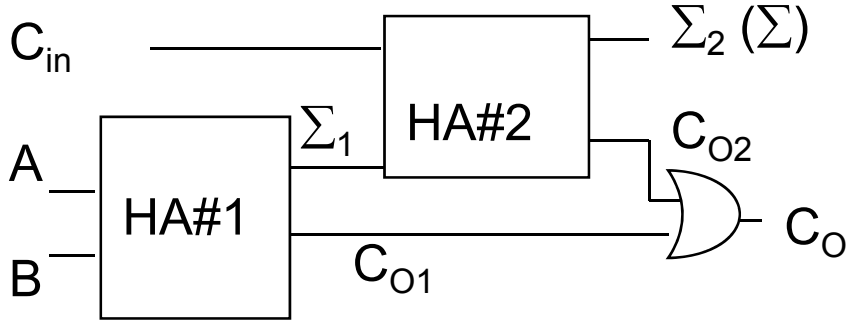
Adder Circuit

Full Adder using half adder

- **Full Adder can be constructed using two half adders**
- **First half adder adds the two bits of the number and their sum is added to the *carry in* bit in another half adder**

Input	$\Sigma_1 C_{in}$		$=C_{O1}+C_{O2}$		
	Output				
	HA#1		HA#2		Carry Out
A B Cin	Σ_1 $A \oplus B$	C_{O1} AB	Σ_2 (Σ) $\Sigma_1 \oplus C_{in}$	C_{O2}	$\Sigma Cin = C_O$
0 0 0	0	0	0	0	0
0 0 1	0	0	1	0	0
0 1 0	1	0	1	0	0
0 1 1	1	0	0	1	1
1 0 0	1	0	1	0	0
1 0 1	1	0	0	1	1
1 1 0	0	1	0	0	1
1 1 1	0	1	1	0	1





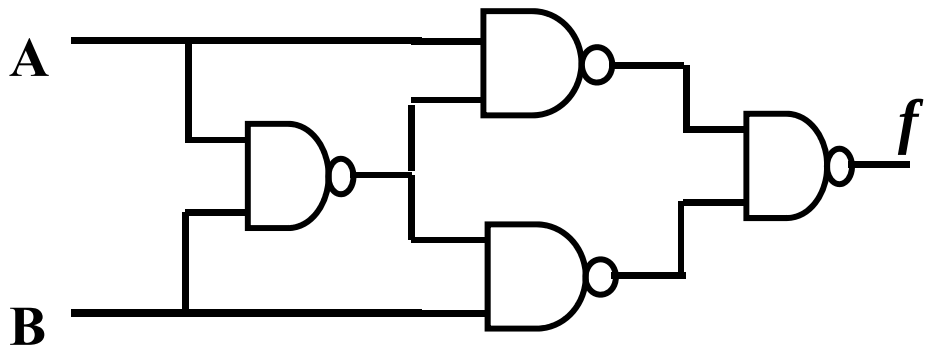
A Full Adder Circuit using two Half Adder

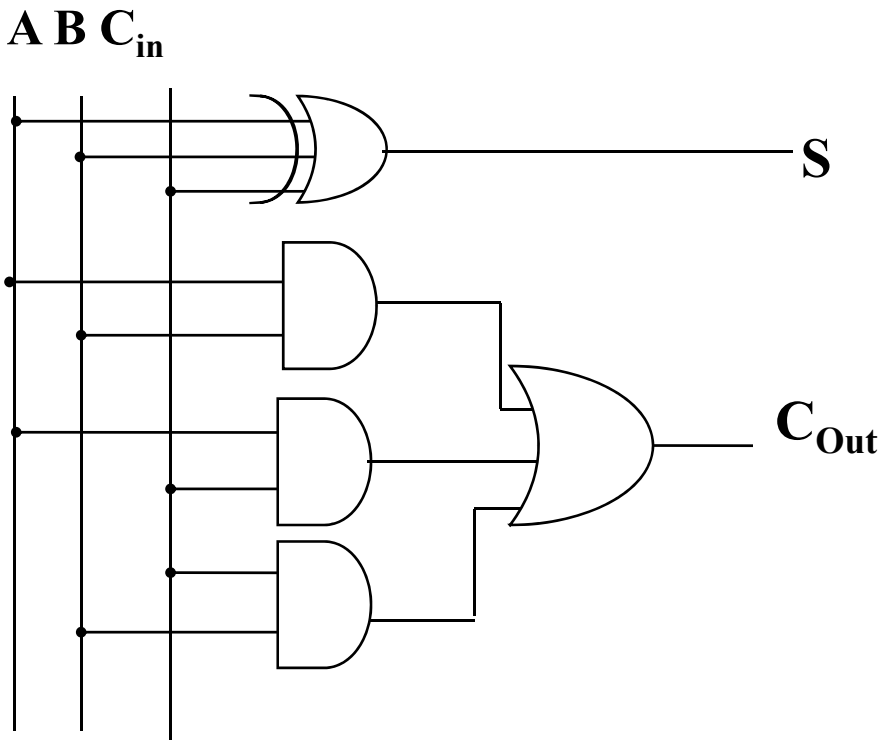
Adder Circuit

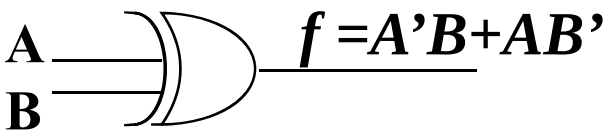
Delay of a Full Adder

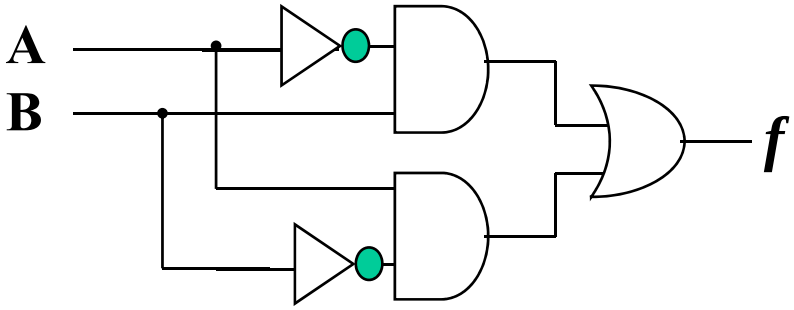
- **Consider that the delay of all basic gates (AND, OR, NAND, NOR, NOT) is δ**
- **Delay for carry = 2 δ**
- **Delay for sum = 3 δ**

(AND-OR delay plus one inverter delay)









Adder Circuit

Parallel Adder Design

- We shall look at the various designs of n-bit parallel adder

  a) Ripple Carry Adder
  b) Carry Look-ahead adder
  c) Carry Save Adder
  d) Carry Select Adder

Adder Circuit

Ripple Carry Adder

- Cascade n full adders to create a n-bit parallel adder
- Carry output from stage-i propagates as the carry input to stage (i+1)
- In the worst-case, carry ripples through all the stages

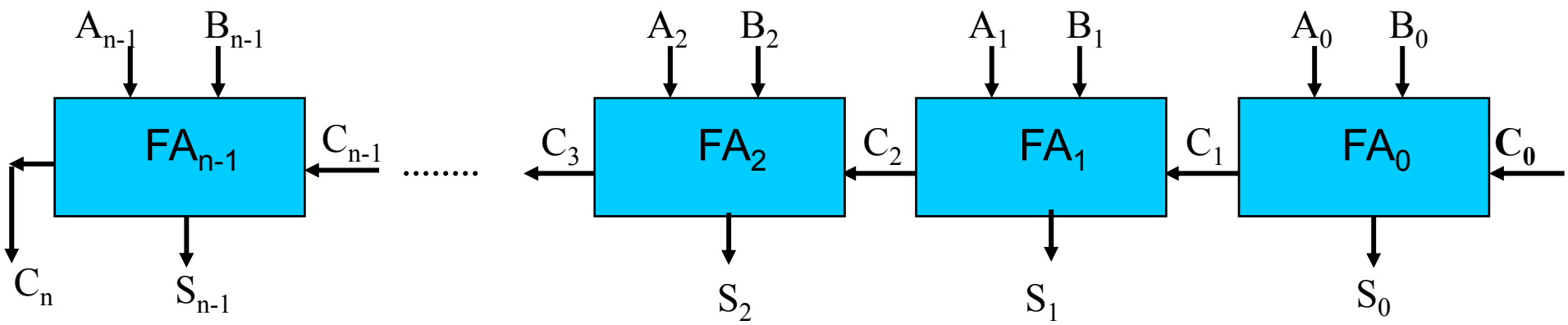
Parallel Adders

C_5	C_4	C_3	C_2	C_1	Carry
	A_4	A_3	A_2	A_1	
	B_4	B_3	B_2	B_1	
	S_4	S_3	S_2	S_1	Sum

$$\begin{array}{rcccc} & 1 & 0 & 0 & 0 \\ & 0 & 1 & 0 & 1 \\ + & 0 & 1 & 1 & 0 \\ \hline & 1 & 0 & 1 & 1 \end{array}$$

Carry in

Adder Circuit (Ripple Carry Adder)



Two numbers : $A_{n-1}\ldots A_2A_1A_0$ and $B_{n-1}\ldots B_2B_1B_0$

Input Carry : C_0

Sum : $S_{n-1}\ldots S_2S_1S_0$

Output Carry : C_n

Delay for $C_1 = 2\delta$

Delay for $C_2 = 4\delta$

Delay for $C_{n-1} = 2(n-1)\delta$

Delay for $C_n = 2n\delta$

Delay for $S_0 = 3\delta$

Delay for $S_1 = 2\delta+3\delta=5\delta$

Delay for $S_2 = 4\delta+ 3\delta=7\delta$

Delay for $S_{n-1} = 2(n-1)\delta+ 3\delta=(2n+1)\delta$

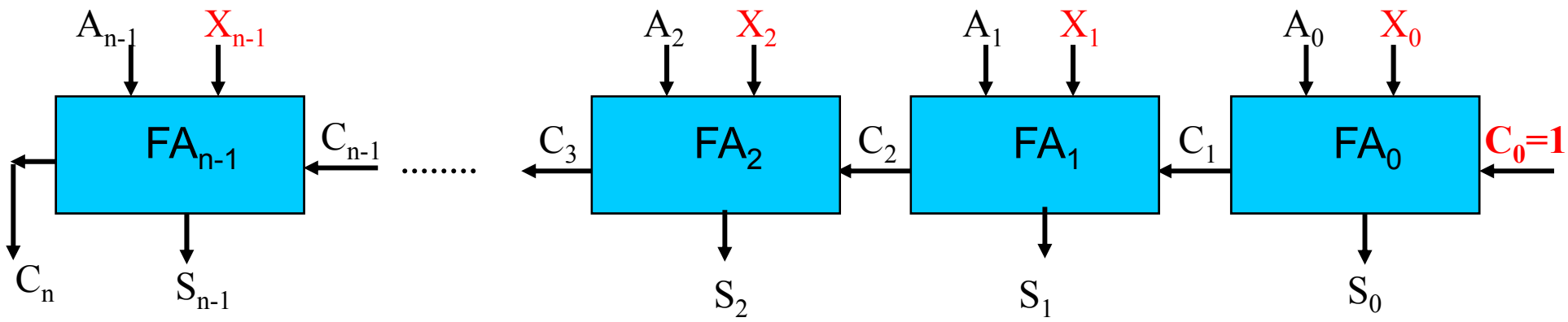
Delay is proportional to n

Adder Circuit

Design a Parallel Subtractor

Observation:

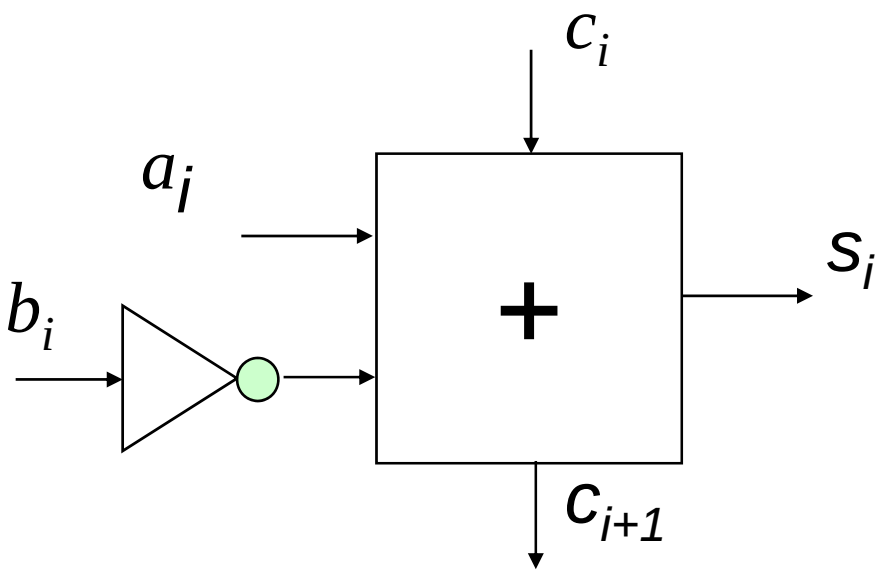
- Computing A-B is the same as adding the 2's complement of B to A.
- 2's complement is equal to 1's complement plus 1
- Let $X_i = B'_i$

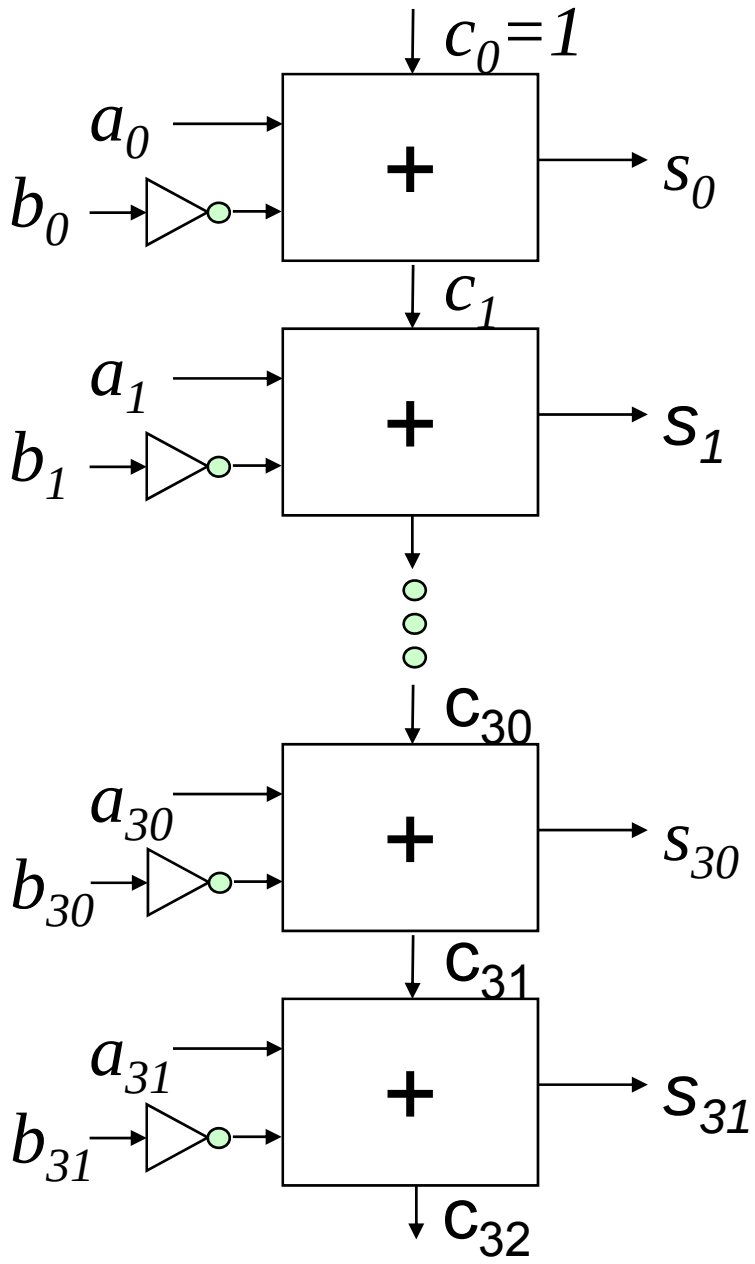


Subtraction Circuit

- Use the formula

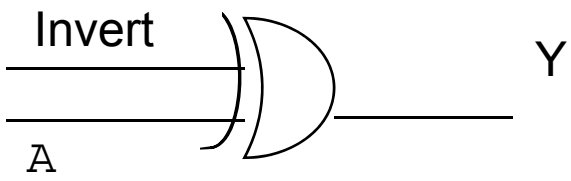
  X-Y = X+Y'+1





Subtraction

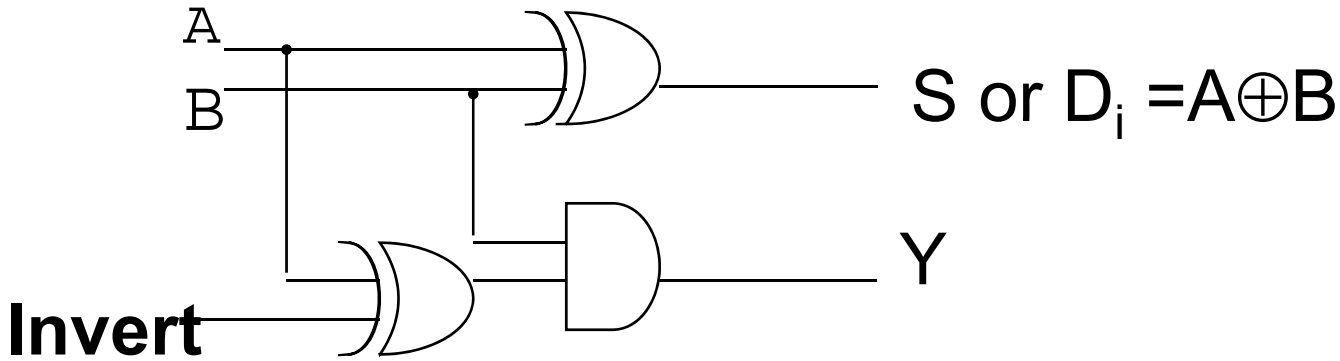
Controlled Inverter



Input		Output	Remark
A	Invert	Y	
0	0	0	Y=A
1	0	1	
0	1	1	Y= $\bar{A}$
1	1	0	

When Invert=0 , then Y=A
and
When Invert=1, then Y= $\bar{A}$.

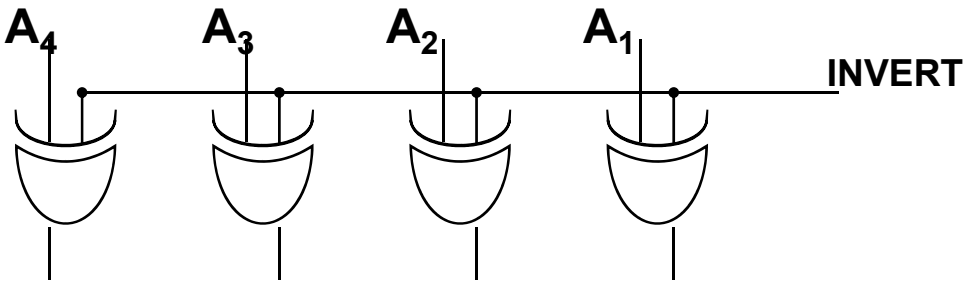
Half Adder/Half Subtracter



If Invert=0, then Y= AB and the circuit performs addition function.

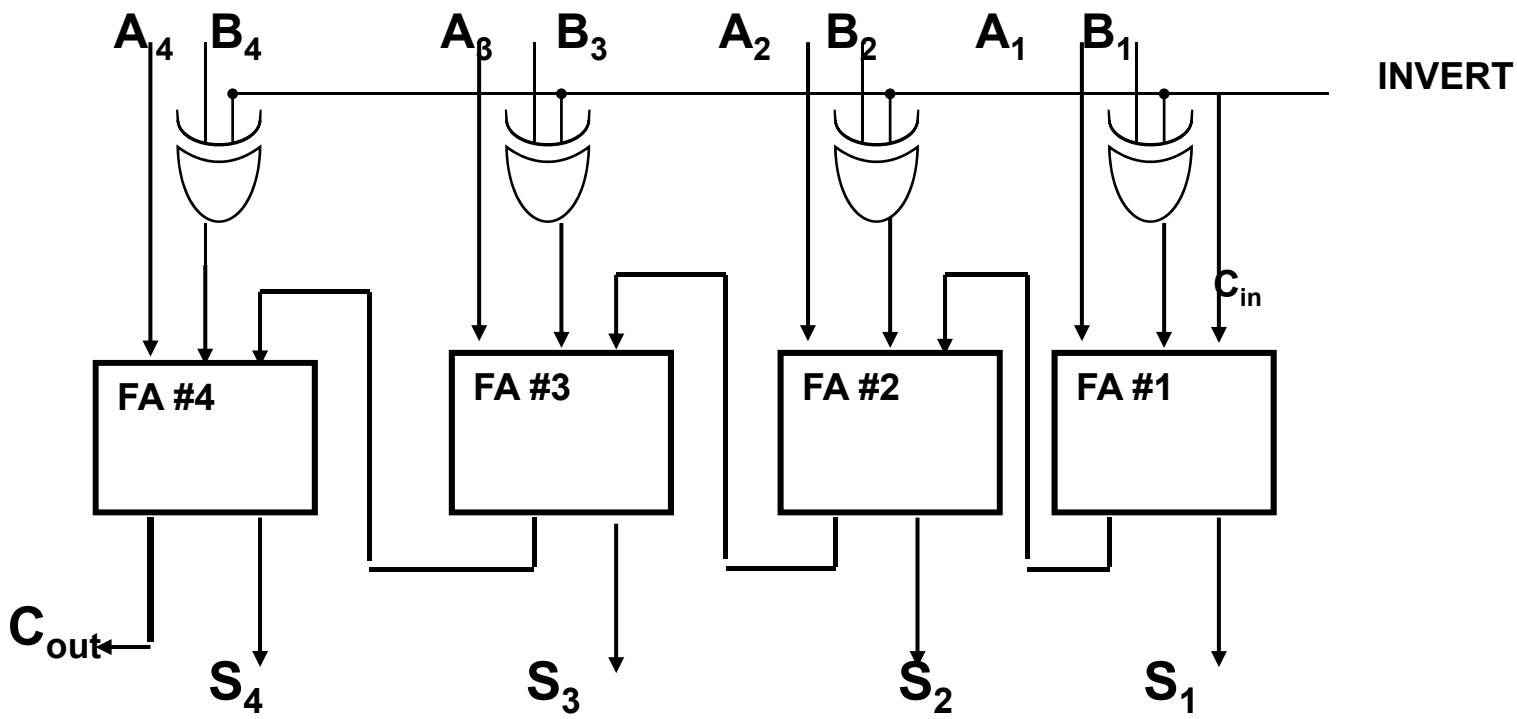
If Invert=1, then Y= $\bar{A}B$ and the circuit performs subtraction.

Adder / Subtracter in 2's Complement System

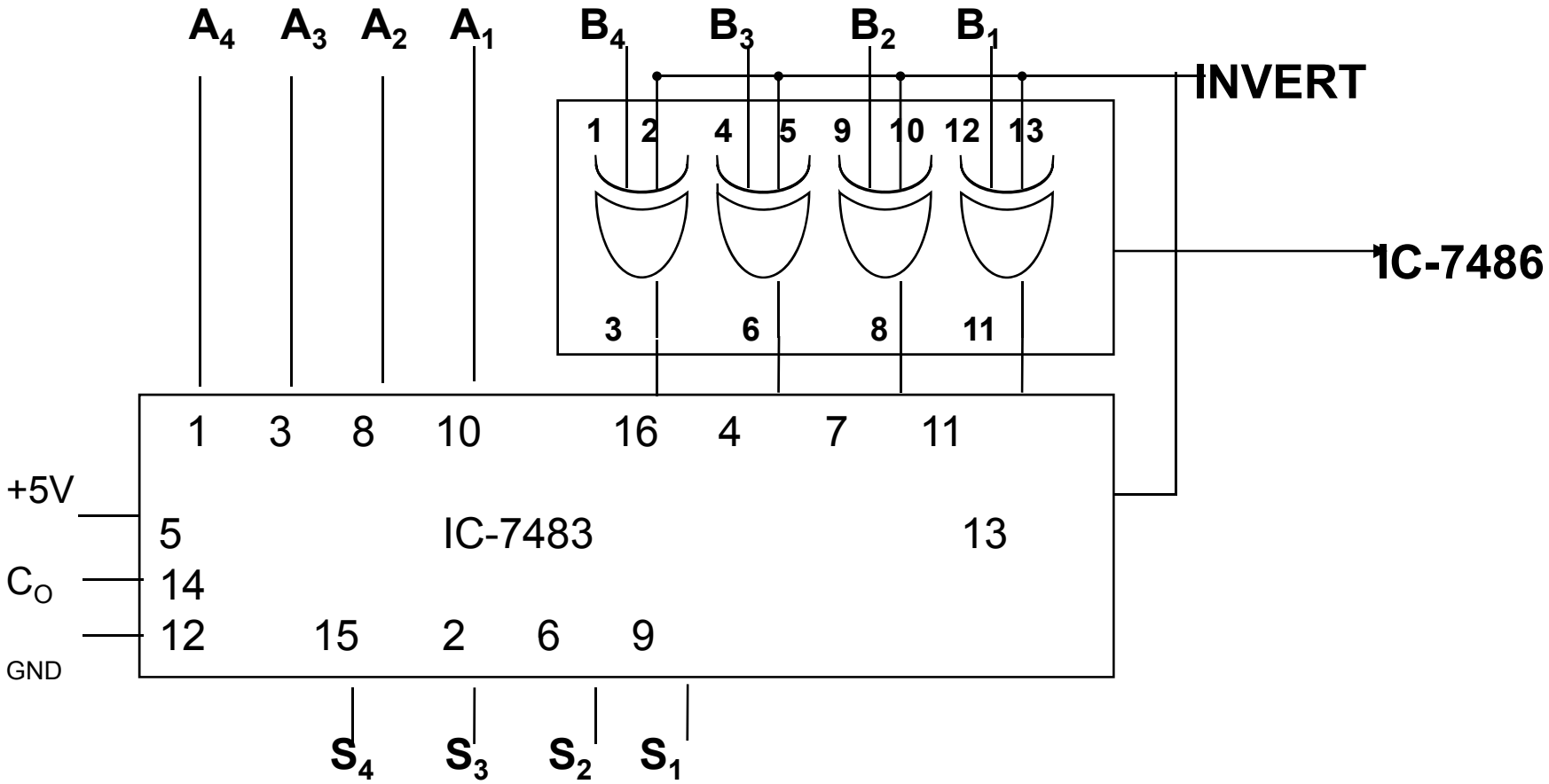


$\bar{A}_4$ $\bar{A}_3$ $\bar{A}_2$ $\bar{A}_1$ If INVERT input =1

A_4 A_3 A_2 A_1 If INVERT input =0

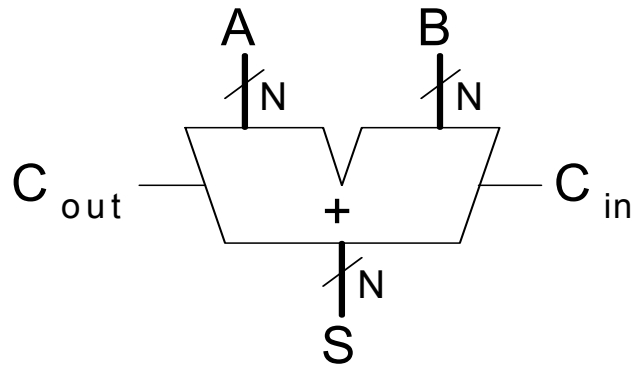


4 Bit 2's complement Adder/ Subtracter

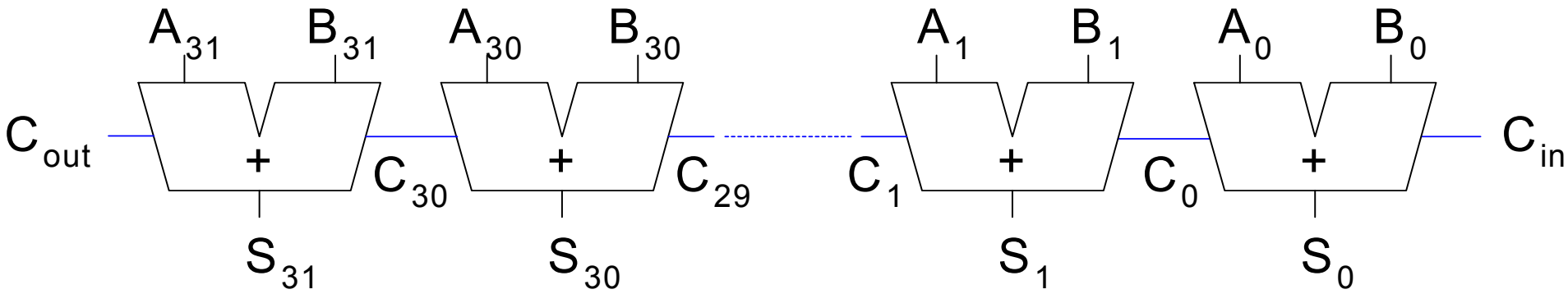


Ripple-Carry Adder

- **Chain 1-bit adders together**
- **Carry ripples through entire chain**
- **Disadvantage: slow**



Symbol



Carry Look-ahead Adder

- **Propagation delay of n-bit ripple-carry adder has been seen to be proportional to n.**
- **Due to the rippling effect of carry sequentially from one stage to the next.**
- **Full adder is unable to start the addition until or unless its carry input is available.**
- **In ripple carry adder, this carry input is coming sequentially through a rippling effect, because of that the different full adders would have to wait till the carry input comes.**

One possible way to speedup the addition

- **Generate the carry signals for the various stages in parallel.**
- **Time complexity reduces from O(n) to O(1).**
- **Hardware complexity increases rapidly with n**

Carry Look-ahead Adder

- Rippling problem can be solved using another kind of adder called carry look ahead adder.
- Name implies we are doing some kind of a look ahead to determine the values of the carry.

Boolean Expressions for Adder

$$Si = ai' bi' ci + ai' bi ci' + ai bi' ci' + ai bi ci$$

$$= ai \oplus bi \oplus ci$$

alternatively,

$$Si = (ai' bi' + ai bi) ci + (ai bi' + ai' bi) ci'$$

$$= ti' ci + ti ci' \quad \text{where } ti = ai bi' + ai' bi$$

$$= (ai \oplus bi) \oplus ci$$

$$c_{i+1} = ai bi + ai ci + bi ci = ai bi + (ai + bi) ci$$

Performance Considerations

Important points about hardware

- the speed of a gate is affected by the number of inputs to the gate

- the speed of a circuit is affected by the number of gates in series

(on “critical path” or “deepest level of logic”)

Speed of Ripple Carry Adder

Ripple is caused because C_{i+1} is generated from C_i, i.e.,

$$c_{i+1} = b_i c_i + a_i c_i + a_i b_i$$

$$c_1 = b_0 c_0 + a_0 c_0 + a_0 b_0$$

$$c_2 = b_1 c_1 + a_1 c_1 + a_1 b_1$$

$$c_3 = b_2 c_2 + a_2 c_2 + a_2 b_2$$

$$c_4 = b_3 c_3 + a_3 c_3 + a_3 b_3$$

Can c_{i+1} be generated directly from a0.. ai, b0.. bi, and c0?

c_{i+1} in terms of $a_0 .. a_i$, $b_0 .. b_i$, and c_0

$c_1 = b0\ c0 + a0\ c0 + a0\ b0$

$c_2 = b1\ c1 + a1\ c1 + a1\ b1$

$=b_0b_1c_0+a_0b_1c_0+a_0b_0b_1+a_1b_0c_0+a_0a_1c_0+a_0a_1b_0+a_1b_1$

$c_3 = b_2c_2+a_2c_2+ a_2b_2 =$

$b_0b_1b_2c_0+a_0b_1b_2c_0+a_0b_0b_1b_2+a_1b_0b_2c_0+a_0a_1b_2c_0+a_0a_1$

$b_0b_2+a_1b_1b_2+a_2b_0b_1c_0+a_0a_2b_1c_0+a_0a_2b_0b_1+a_1a_2b_0c_0$

$+a_0a_1a_2c_0+a_0a_1a_2b_0+a_1a_2b_1+a_2b_2$

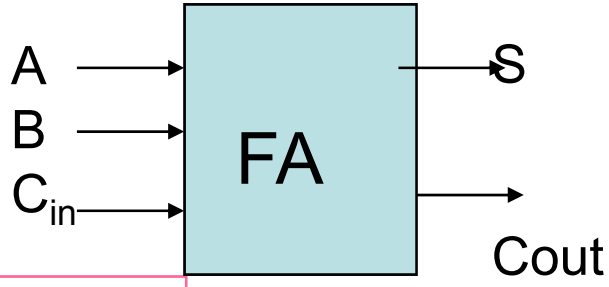
$c_4 = b_3c_3 + a_3c_3 + a_3\ b_3 = ...$

c32 will have 4 billion terms !!

Effect of large fanin and fanout ??

Carry-Look ahead Adder

- Consider the i-th stage in the addition process
- We define two terms Carry Generate and Carry Propagate functions as



$$G_i = A_i B_i$$

$$P_i = A_i \oplus B_i$$

FA implementation using HA, Cout= A.B+ (A $\oplus$ B).Cin

- **A carry out will be generated if A*i* AND B*i* are both 1.**
- **A "carry in" will be propagated to the carry out if a*i* OR *bi* is 1.**
- ***Gi=1 represents the condition when a carry is generated in stage-i independent of the other stages.***
- ***Pi =1 represents the condition when an input carry Ci will be propagated to the output carry Ci+1.***

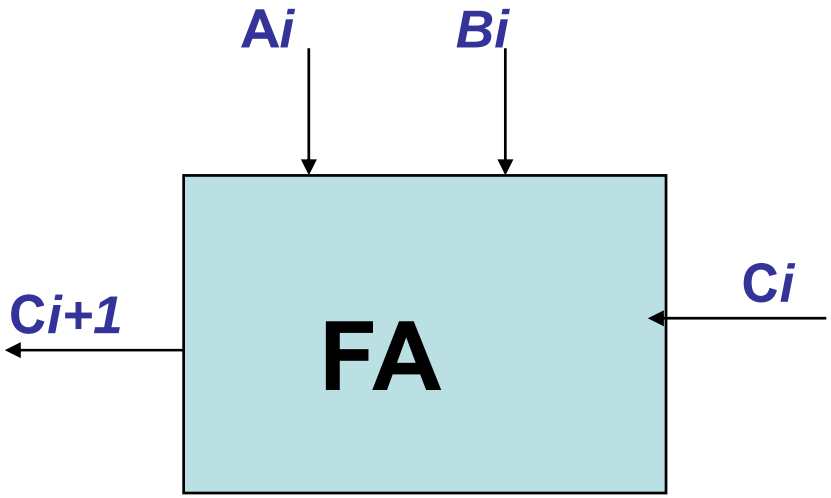
$$C_{i+1} = G_i + P_i . C_i$$

Carry-Look ahead Adder

Generate

Ci+1 =1 will be generated if both Ai =1 Bi = 1 irrespective of Ci value (0 or 1)

[Ci =1, Ai =1 Bi = 1 Ci+1 =1, sum =1
Ci =0, Ai =1 Bi = 1 Ci+1 =1, sum =0]



Propagate

Suppose, C*i* = *1/0*

C_{i+1} *=1/0 will be propagated if [(Ai =1 Bi = 0) or (Ai =0 Bi = 1)]*
i.e. Pi = Ai ⊕ Bi
[Ci =1, Ai =1 Bi = 0 Ci+1 =1, sum =0 , Ci =1, Ai =0 Bi = 1 Ci+1 =1, sum =0]

if [(Ai =1 Bi = 0) or (Ai =0 Bi = 1)] i.e. Pi = Ai ⊕ Bi , C*i+1* =C*i*

We can simply write the following Expression: Ci+1 = Gi + Pi.Ci

Carry out is 1 if either carry (Gi) is generated or propagate function is true and there is an input carry

Carry-Look ahead Adder

$C4 = G3+P3\ C3 = G3+ G2P3+G1P2P3+G0P1P2P3+C0P0P1P2P3$

$C3 = G2+P2C2=G2+G1P2+G0P1P2+C0P0P1P2$

$C2 = G1+P1C1 =G1+G0P1+C0P0P1$

$C1_1 = G0+P0C0=G0+C0P0$

$S0 = A0 \oplus B0 \oplus C0=P0 \oplus C0$

$S1 = P1 \oplus C1$

$S2 = P2 \oplus C2$

$S3 = P3 \oplus C3$

4 AND2 gates

3 AND3 gates

2 AND4 gates

1 AND5 gate

1 OR2, 1 OR3, 1 OR4 AND OR5 gate

4 XOR2 gates

Carry-Look ahead Adder

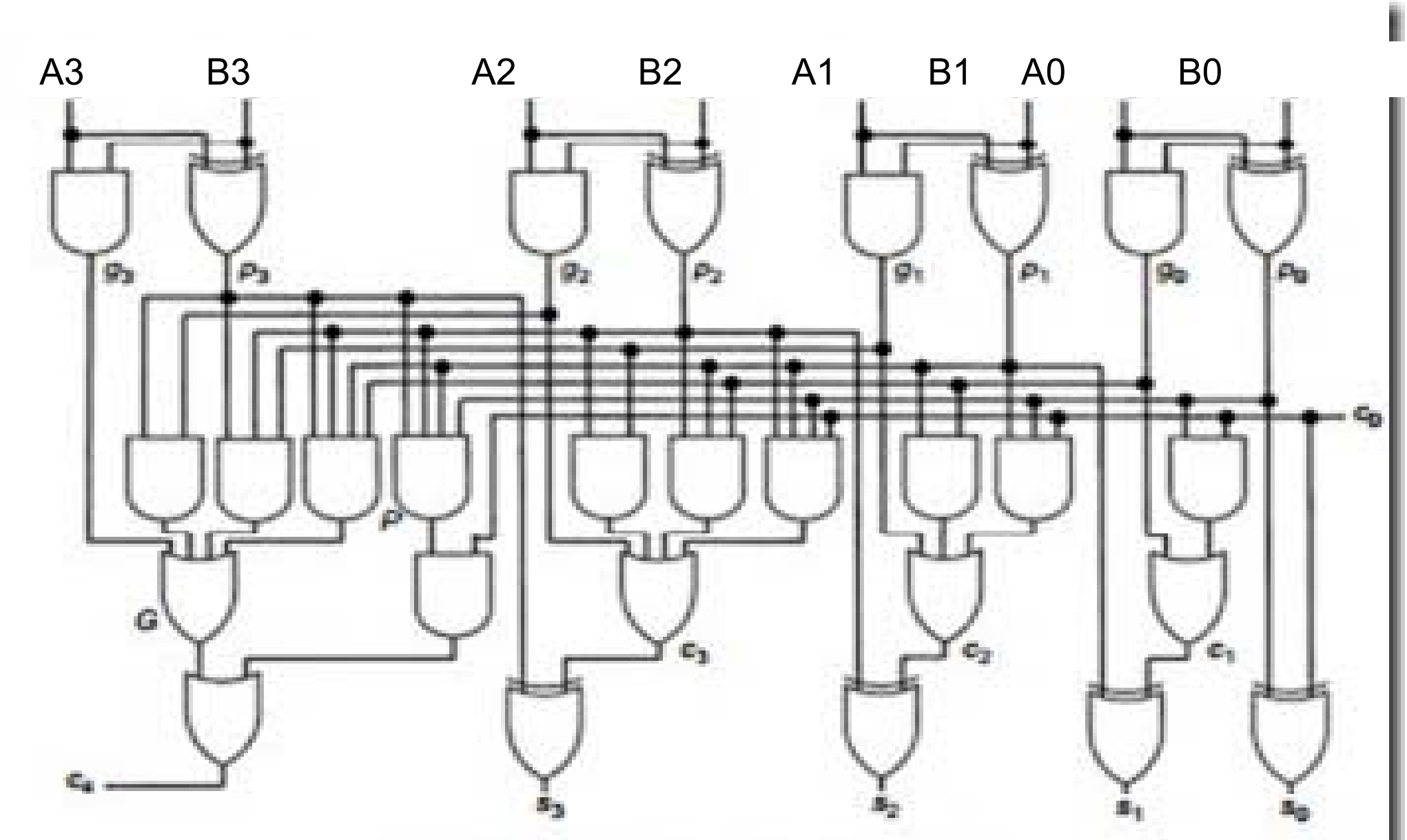
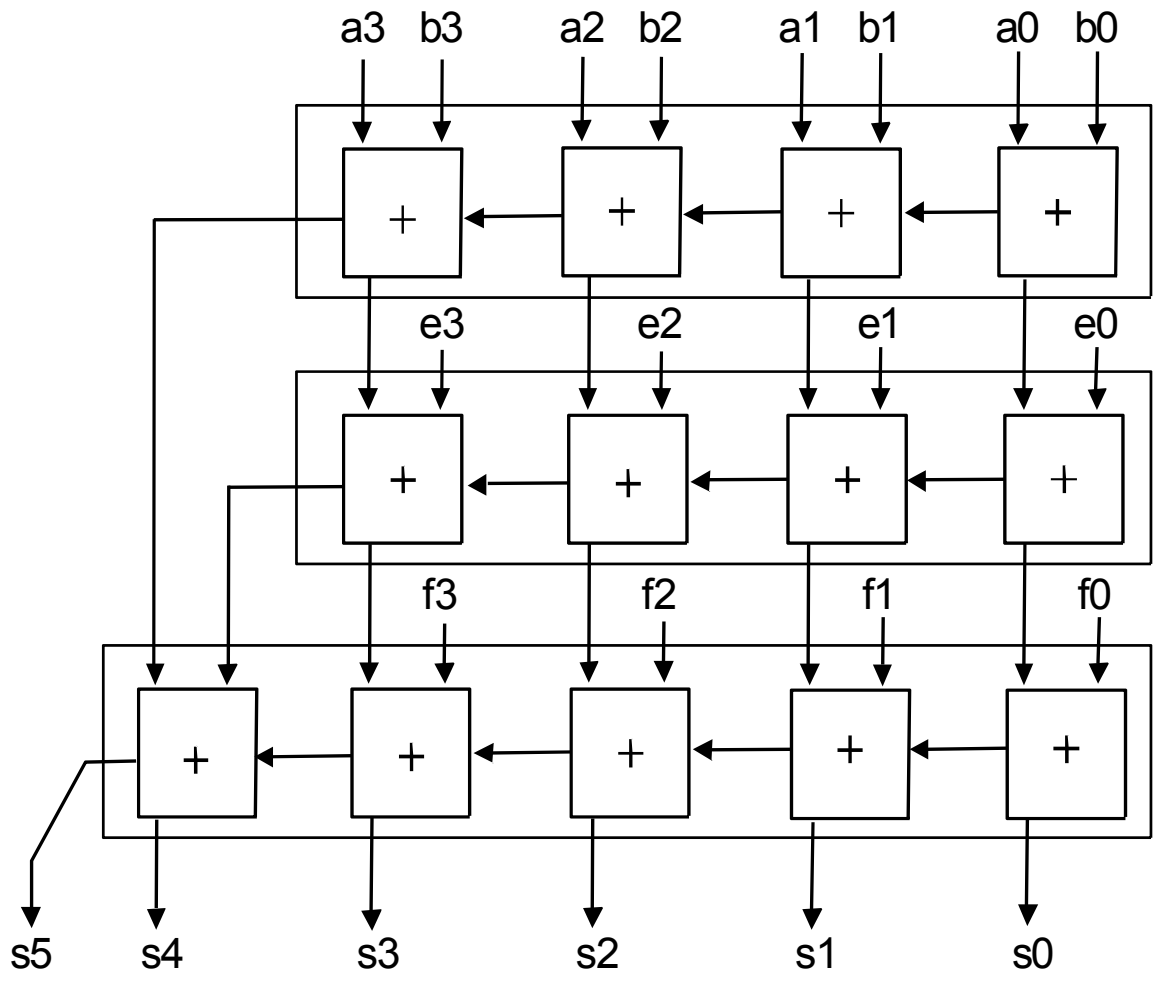
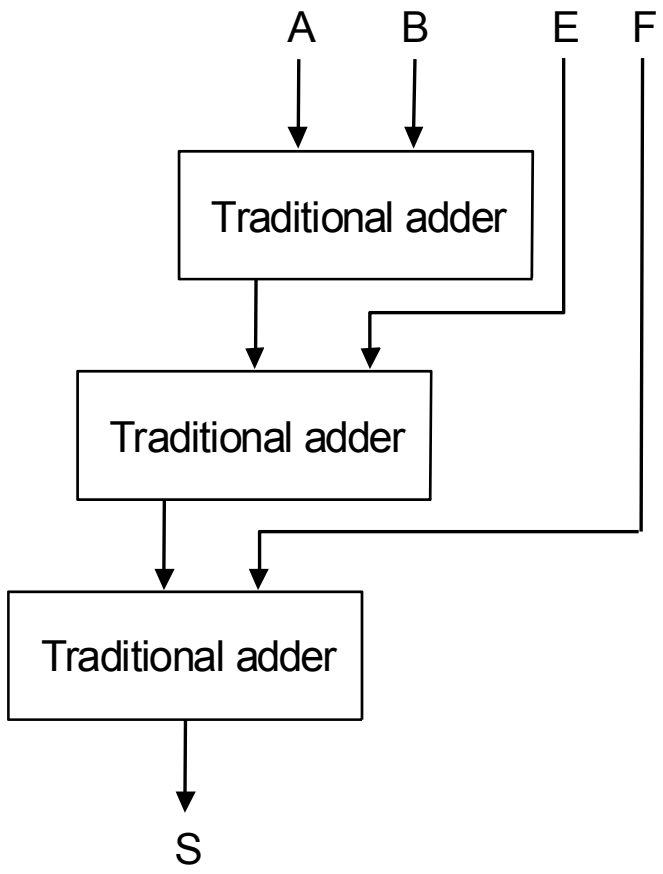


FIGURE 41.10 A 4-bit carry look-ahead adder.

Adding Multiple Operands



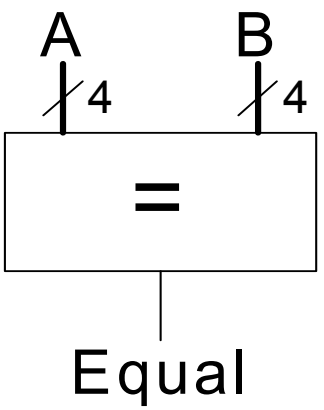


Carry Save Adder

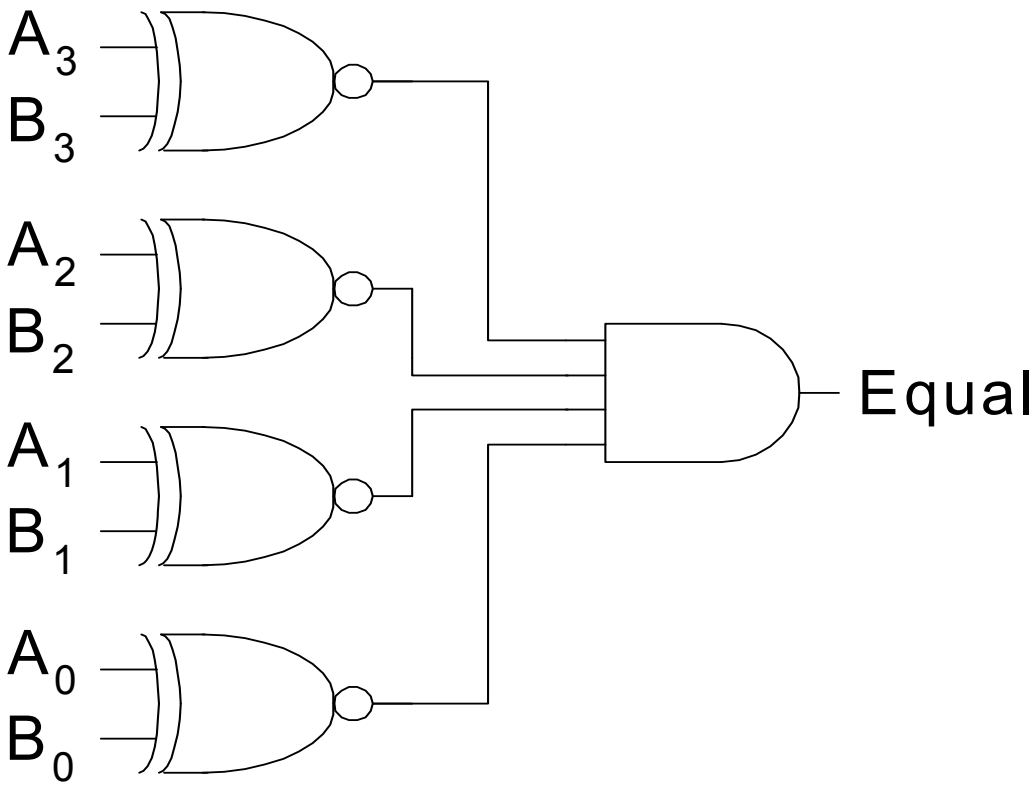
- **Here we add three operands (say, X,Y and Z) together**
- **For adding multiple numbers, we have to construct a tree of carry save adders.**
- **Used in combinational multiplier design.**
- **Each carry save adder is simply an independent full adder without carry propagation.**
- **A parallel adder is required only at the last stage.**

Comparator: Equality

Symbol

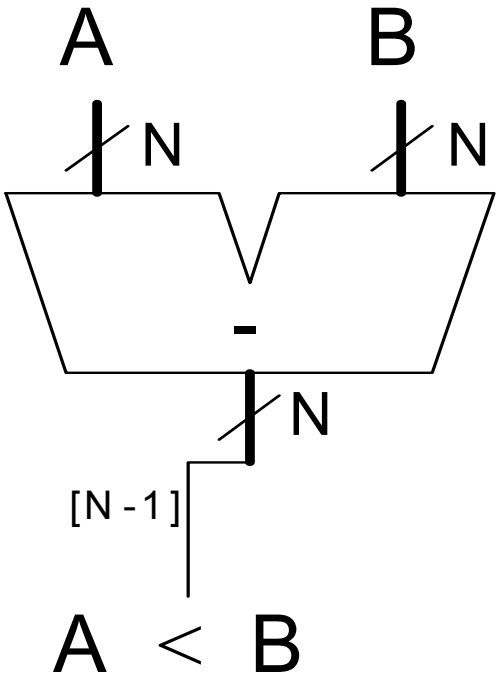


Implementation



Comparator: Less Than

- For unsigned numbers



Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 =
  - Ex: 11001 << 2 =

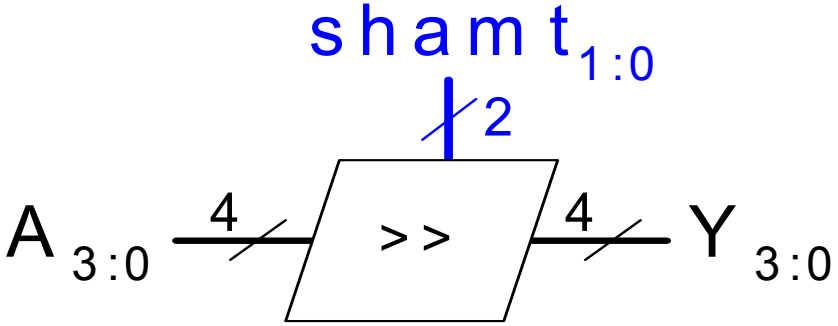
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 =
  - Ex: 11001 <<< 2 =

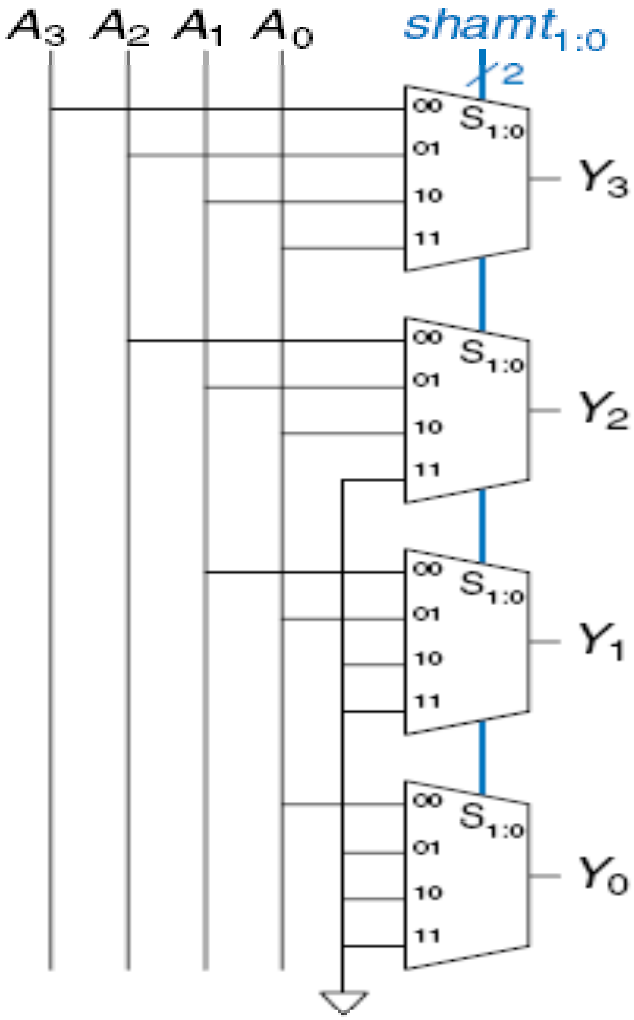
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 =
  - Ex: 11001 ROL 2 =

Shifters

- Logical shifter: shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- Arithmetic shifter: same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 = 11110
  - Ex: 11001 <<< 2 = 00100
- Rotator: rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111

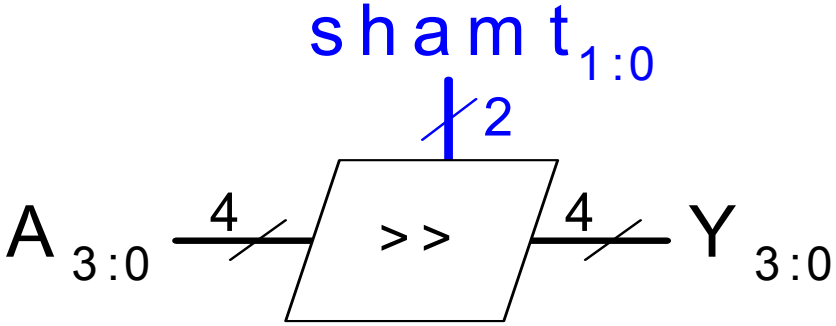
Shifter Design (left)

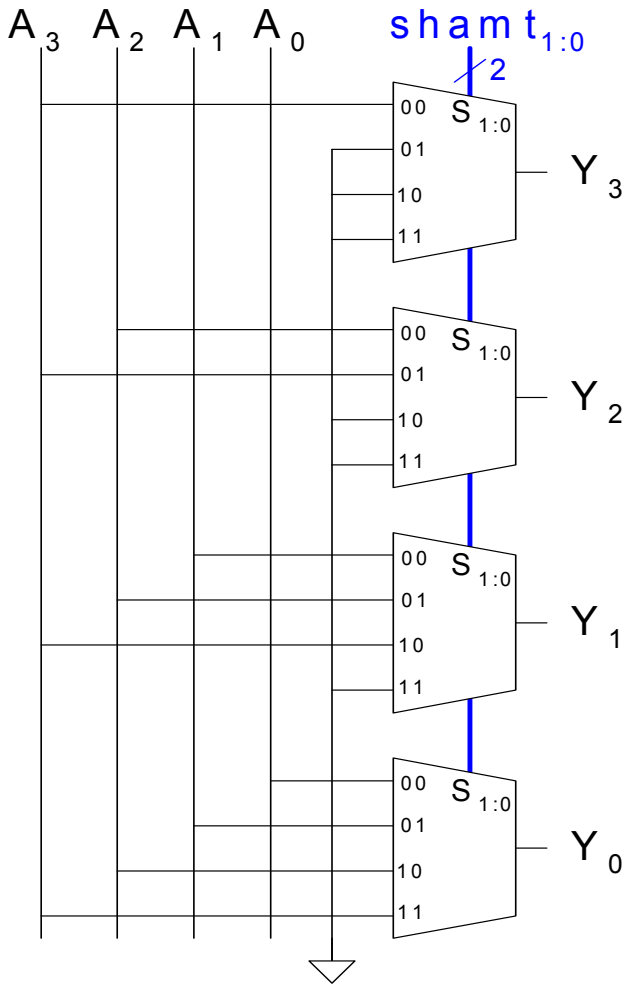




When shamt=00, Y3Y2Y1Y0=A3A2A1A0
When Shamt =10, Y3Y2Y1Y0=A1A0 0 0

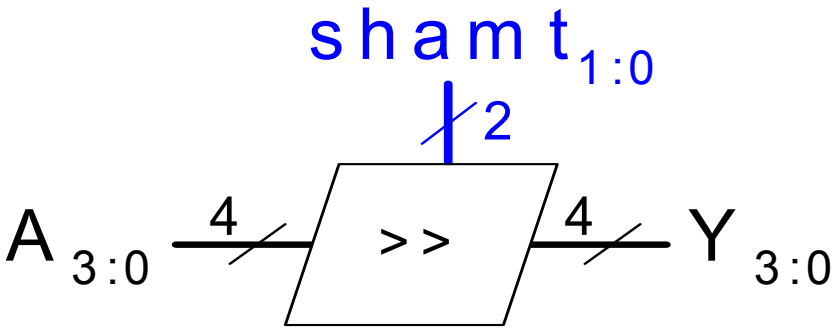
Shifter Design (Right)



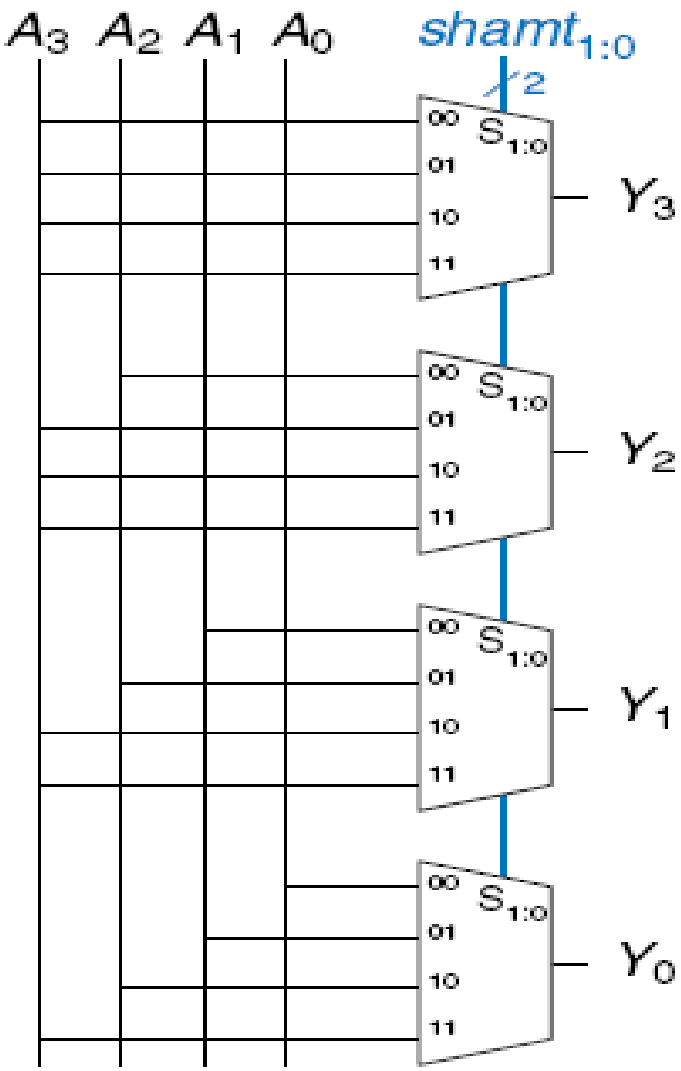


When shamt=00, Y3Y2Y1Y0=A3A2A1A0
When Shamt =10, Y3Y2Y1Y0=00A3A2

Shifter Design (Arithmetic Right)



When shamt=00,
Y3Y2Y1Y0=A3A2A1A0
When Shamt =10,
Y3Y2Y1Y0=A3A3A3A2



Shift operations

shift left logical 3 bits

$a_{31}\ a_{30}\ \ldots\ldots\ a_1\ a_0$

$a_{28}\ a_{27}\ \ldots\ldots\ a_1\ a_0\ 0\ 0\ 0$

shift right logical 3 bits

$a_{31}\ a_{30}\ \ldots\ldots\ a_1\ a_0$

$0\ 0\ 0\ a_{31}\ a_{30}\ \ldots\ldots\ a_4\ a_3$

shift right arithmetic 2 bits

$a_{31}\ a_{30}\ \ldots\ldots\ a_1\ a_0$

$a_{31}\ a_{31}\ a_{31}\ a_{30}\ \ldots\ldots\ a_3\ a_2$

Multipliers

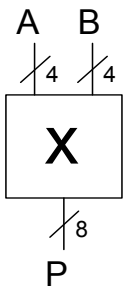
- Steps of multiplication for both decimal and binary numbers:
  - Partial products are formed by multiplying a single digit of the multiplier with the entire multiplicand
  - Shifted partial products are summed to form the result

Decimal		Binary
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
		0101
		+ 0000
9660	result	0100011

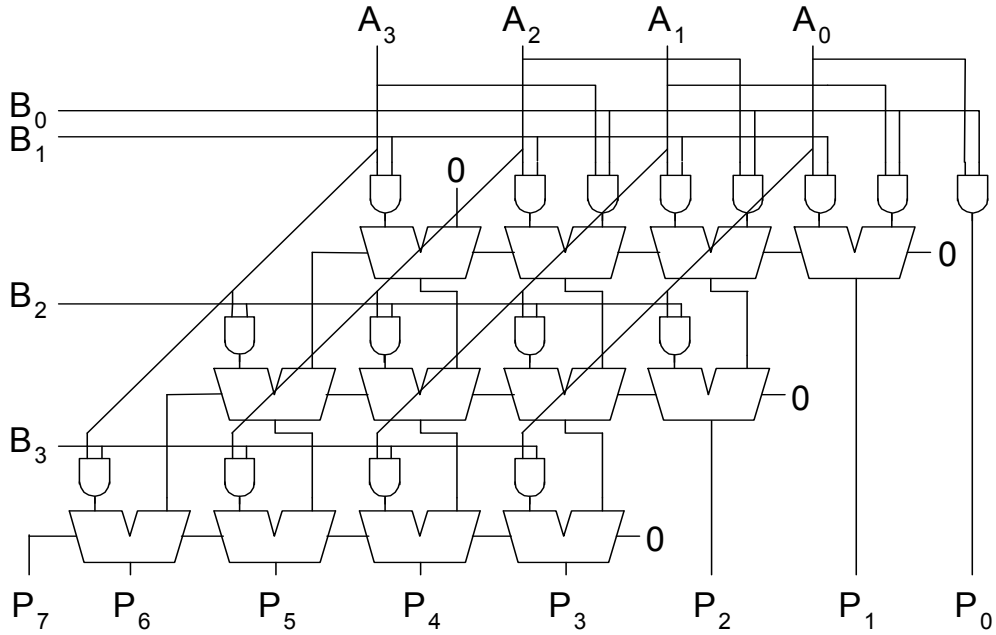
230 x 42 = 9660

5 x 7 = 35

4 x 4 Multiplier



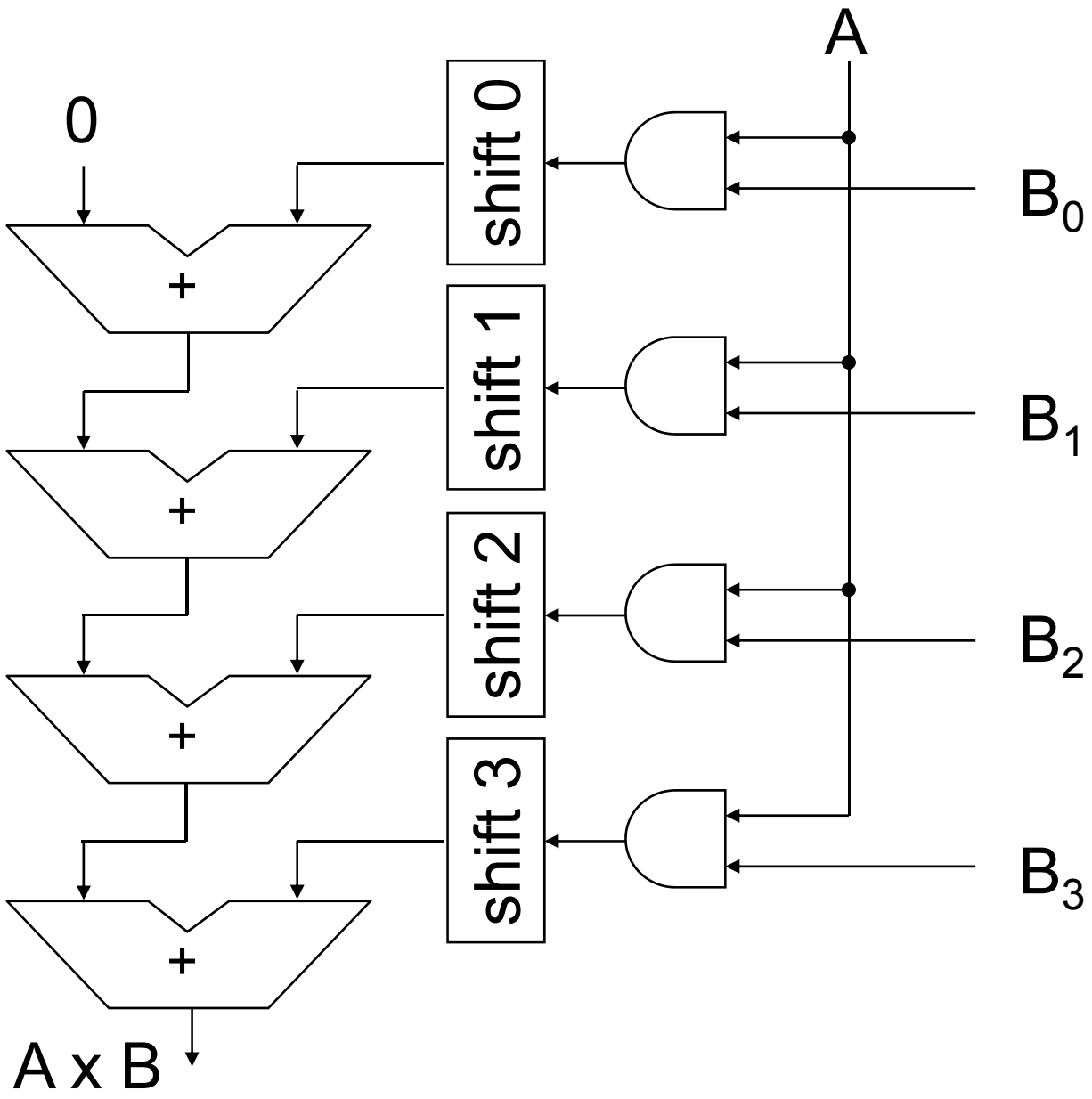
$$
\begin{array}{rcccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 \\
 & & & \times & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



Array Multiplier Adding Many Operands

$$
\begin{array}{cccccccc}
 & & & A_3 & A_2 & A_1 & A_0 \\
 & & \times & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
+ \quad A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
P_7 \quad P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

$$A \times B = \sum_{i=0}^{n-1} A \cdot B_i \times 2^i$$



Assignments

1. Derive the critical delay for

   a) n bit Ripple carry ripple adder, (b) n bit carry-look-ahead adder
   Assume that the delay of all basic gates (AND, OR, NAND, NOR, NOT) is δ.

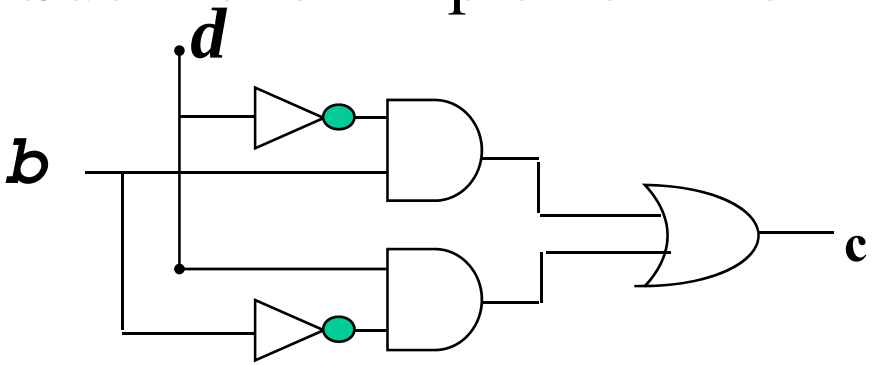
2. Compare the delay of a 64-bit ripple-carry adder and a 64-bit carry-look-ahead adder with 4-bit blocks. Use only two-input gates. Each two-input gate has a 50ps delay.

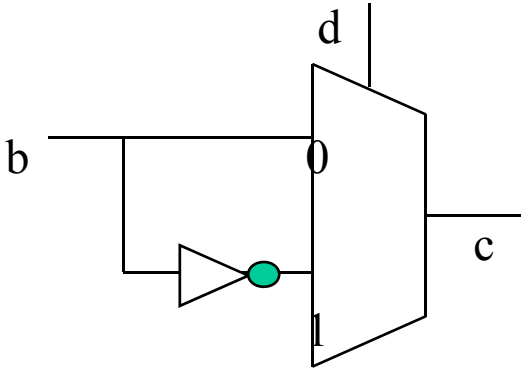
3. A 2-bit left shifter creates the output by appending two zeros to the least significant bits of the input and dropping the two most significant bits.

Arithmetic and Logic Unit (ALU) Design

A 1-bit ALU

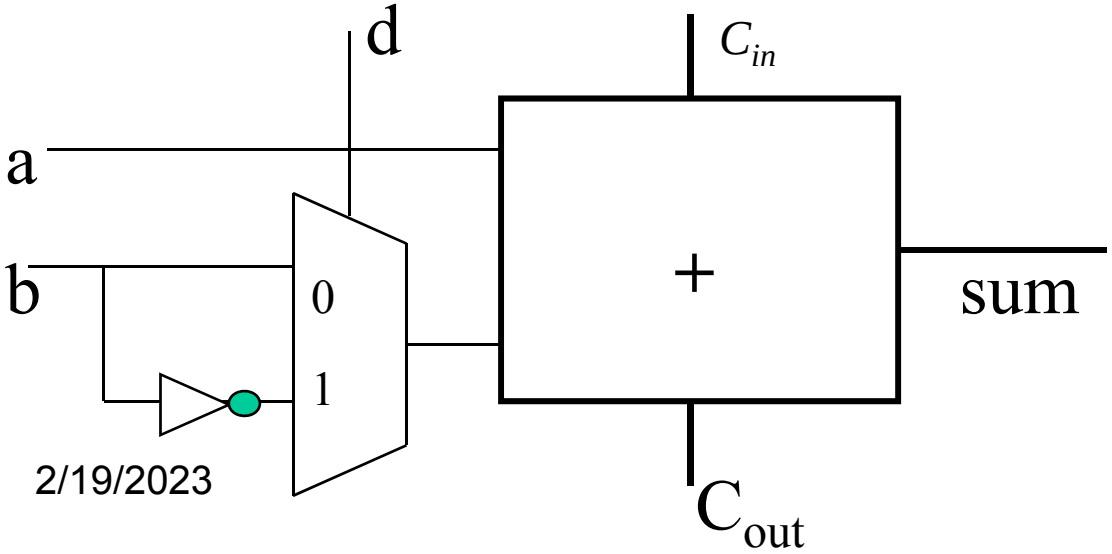
Subtraction implementation





If d= =0, c = b;

If d= =1, c = $\overline{b}$;



If C_{in}= d =1,

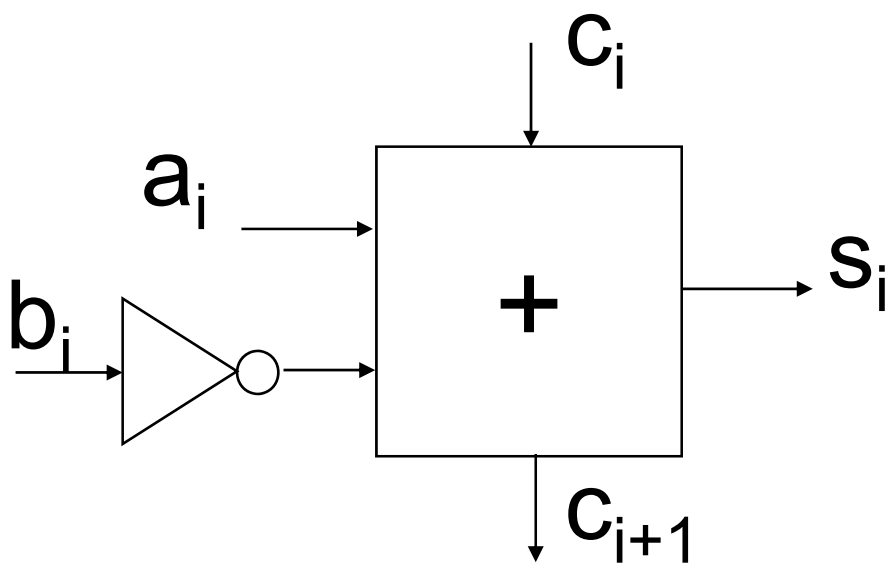
Sum=a + $\overline{b}$ + 1

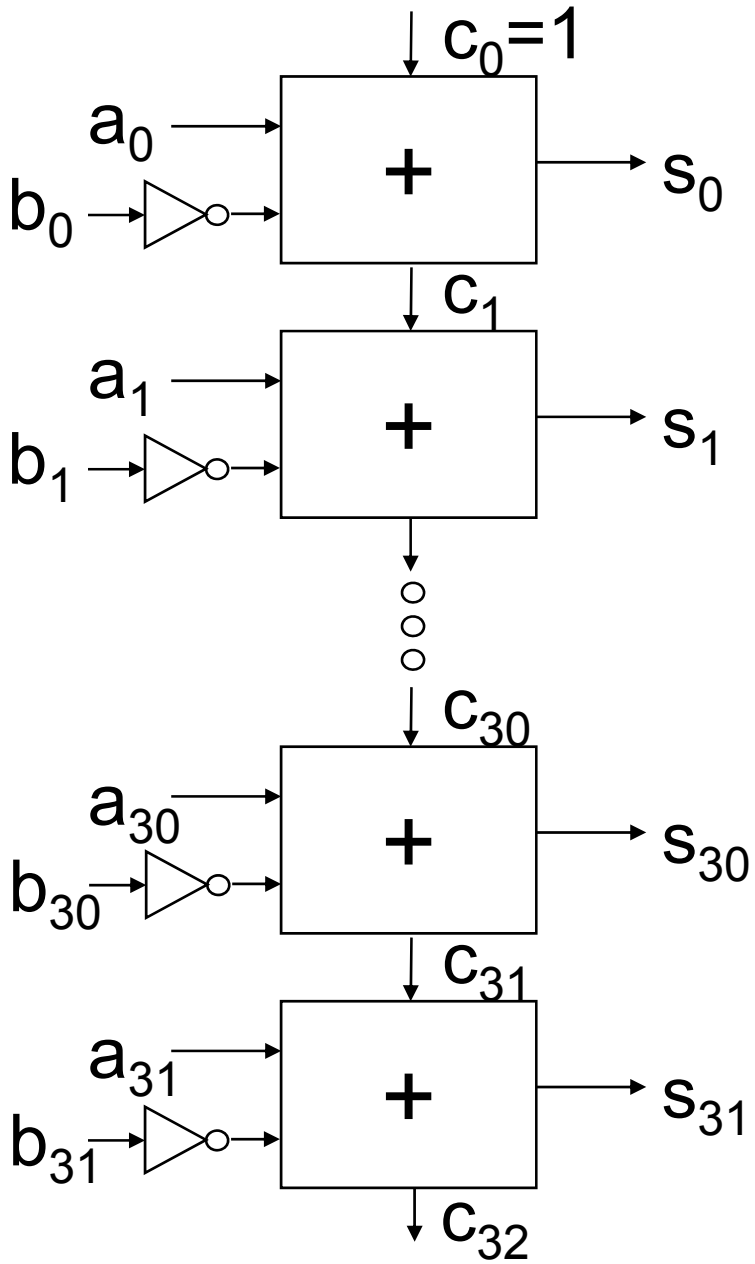
= a +($\overline{b}$ + 1)

= a - b

Subtraction Circuit

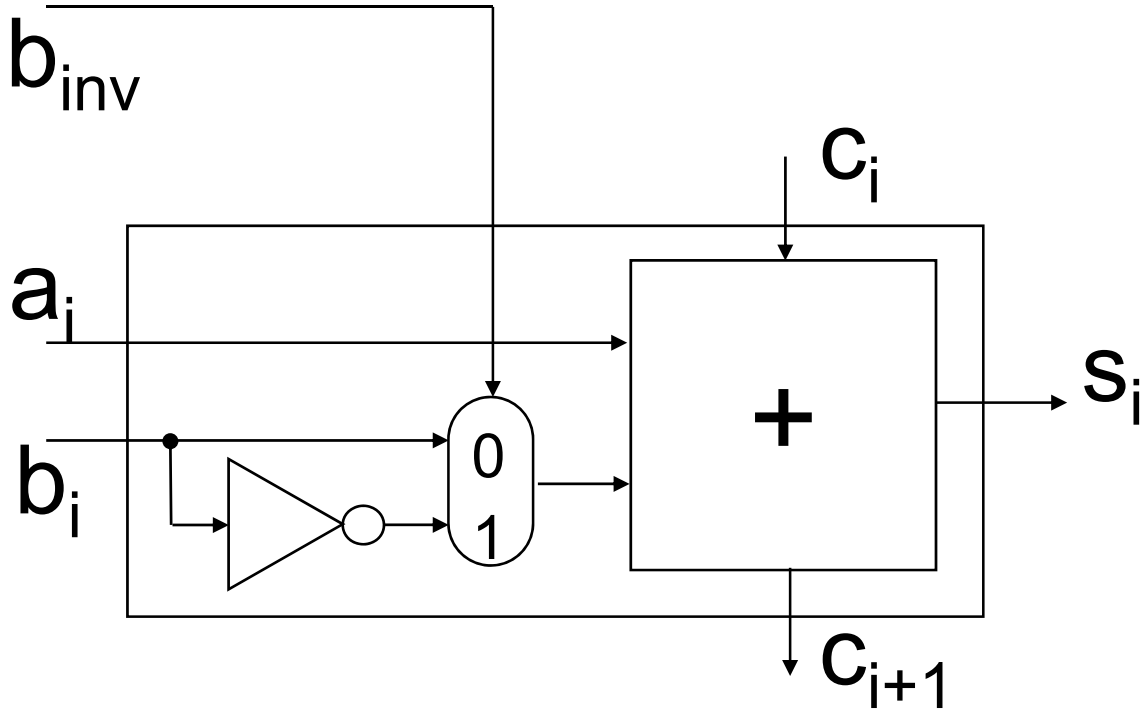
- Use the formula
- X-Y = X+Y'+1

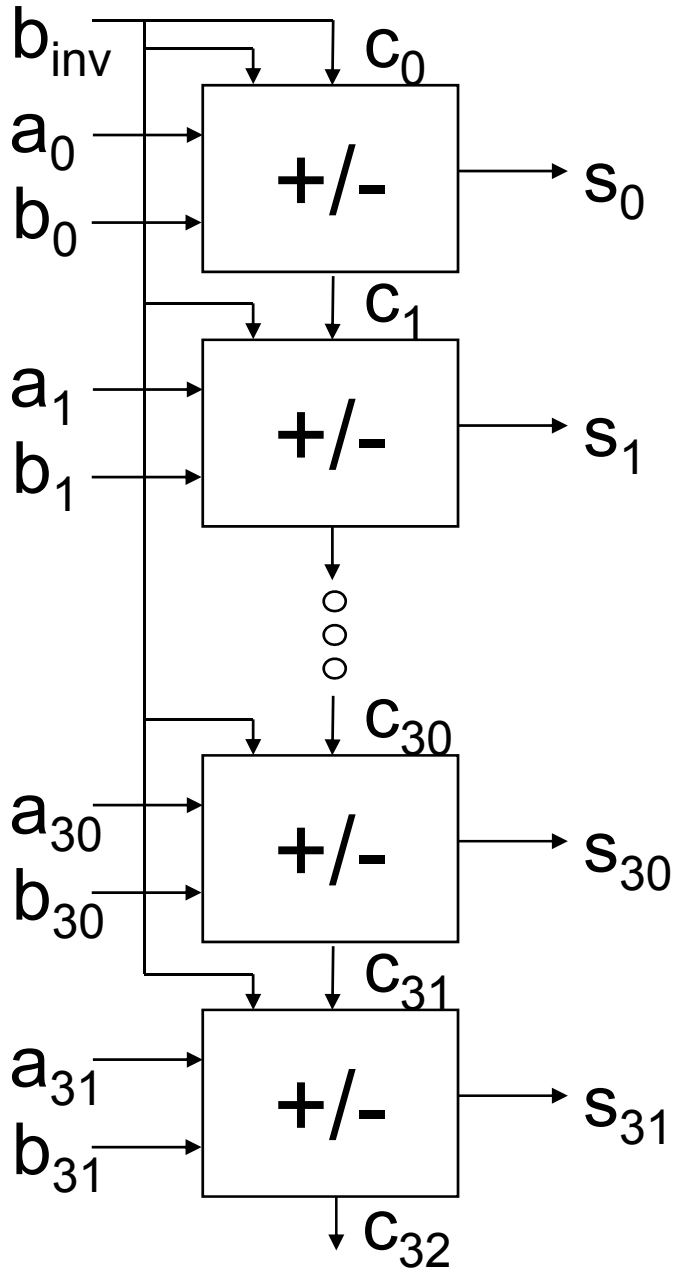




Combining Addition and Subtraction

- **Use a multiplexer circuit**
- **When b_{inv}= 0, S_i=sum**
- **When b_{inv}= 1, S_i=Difference**





Logical Operations: AND, OR

MIPS logical instructions require bit by bit operation on 32 bit strings

AND:

0010 1011 1100 0110 1111 0000 0101 1000

<u>0000 1111 0000 1111 0000 1111 0000 1111</u>

0000 1011 0000 0110 0000 0000 0000 1000

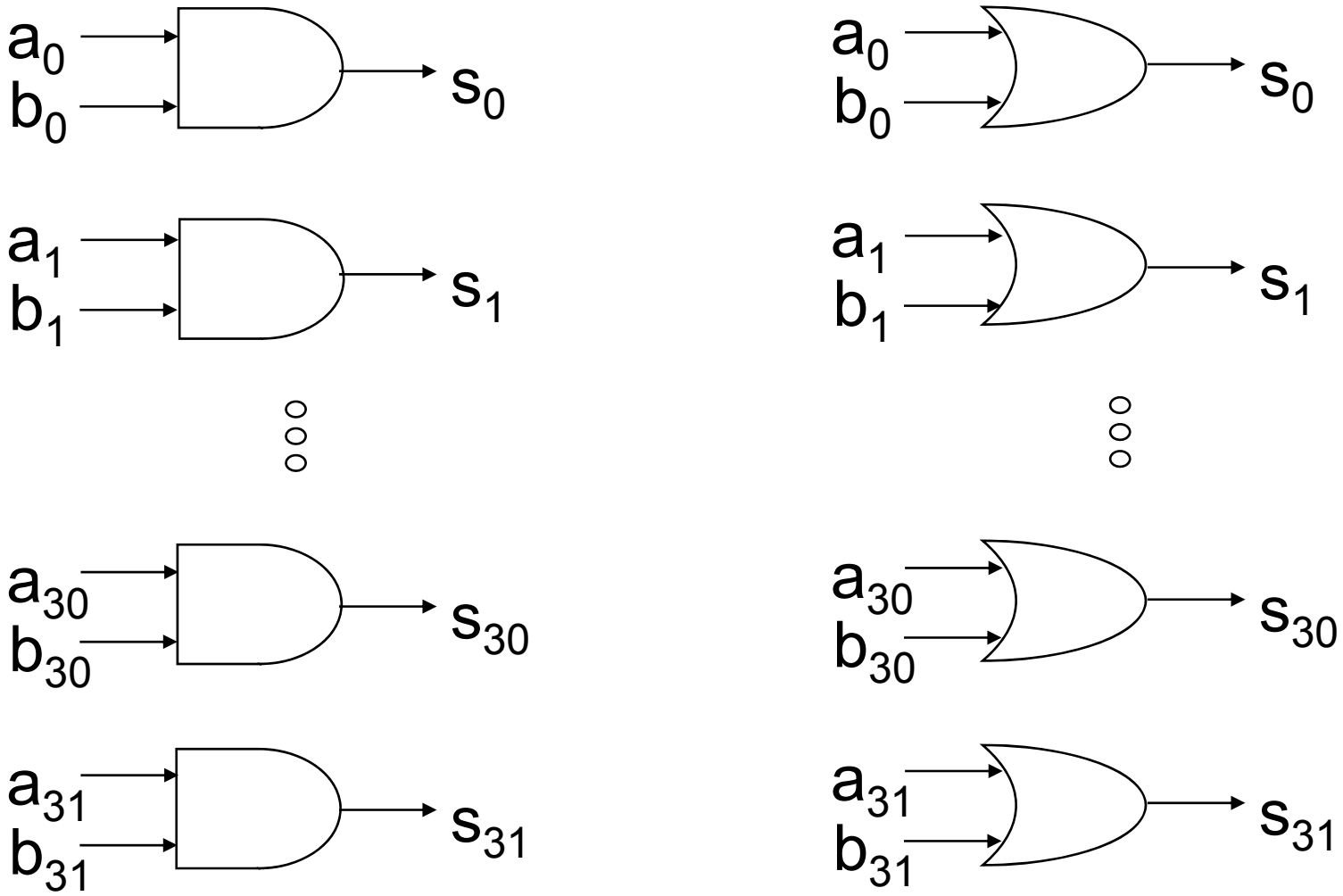
OR:

0010 1011 1100 0110 1111 0000 0101 1000

<u>0000 1111 0000 1111 0000 1111 0000 1111</u>

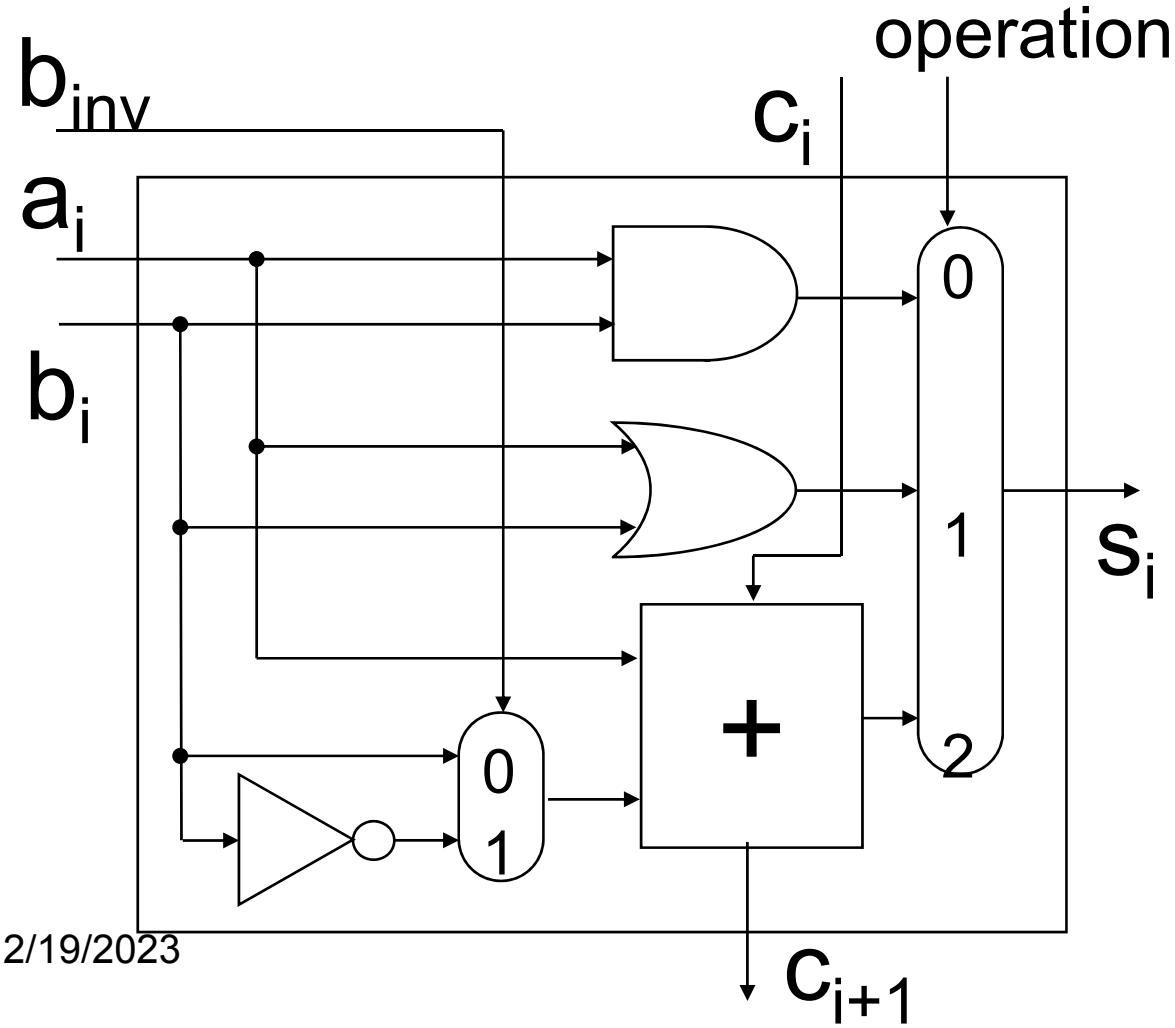
0010 1111 1100 1111 1111 1111 0101 1111

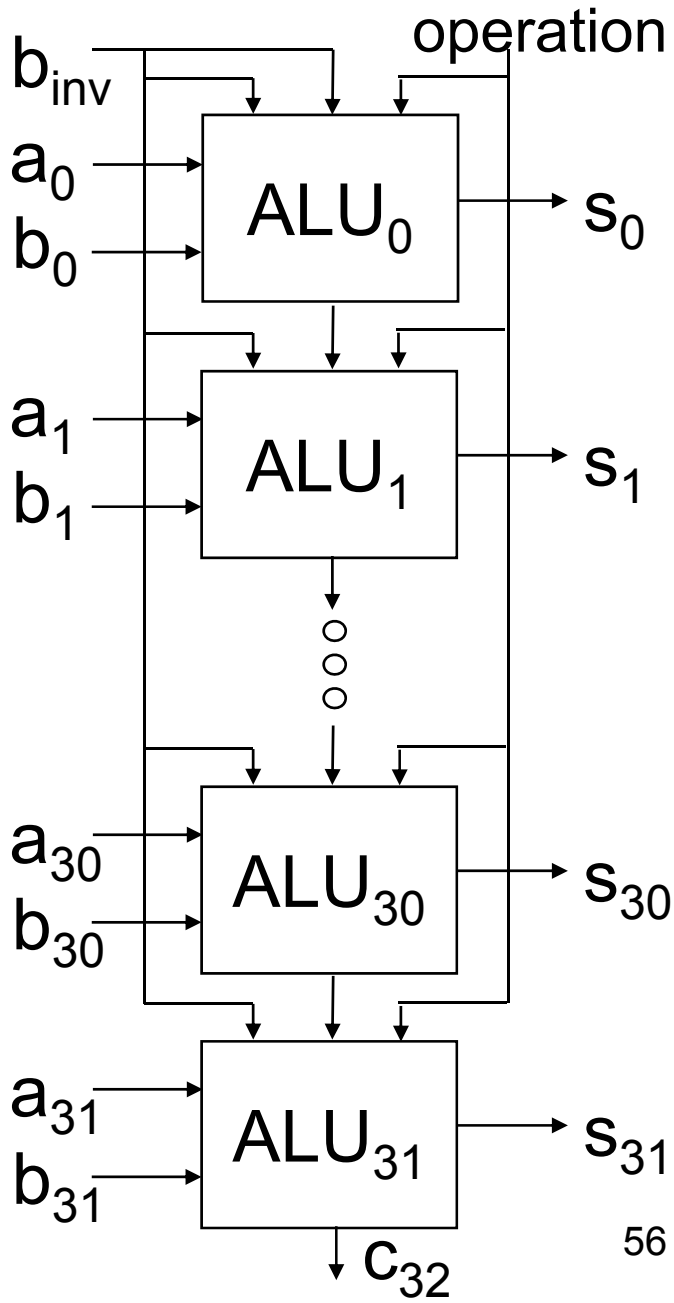
Circuit for ‘AND’ and ‘OR’ Instructions



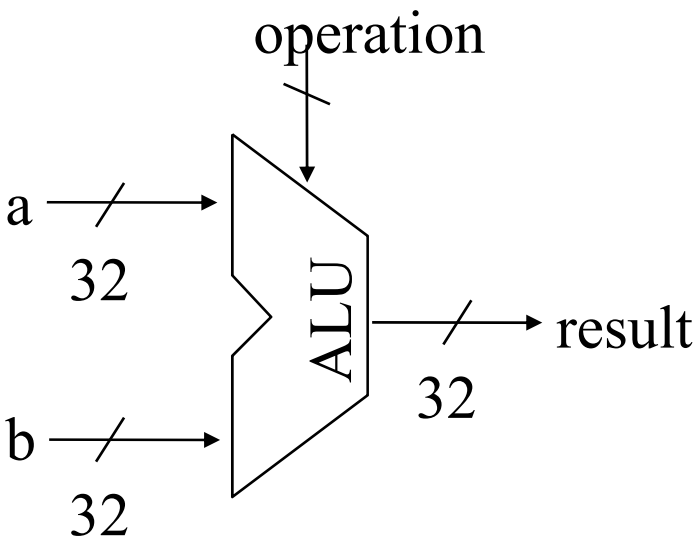
Combining 'AND' , 'OR' , ADD, SUB

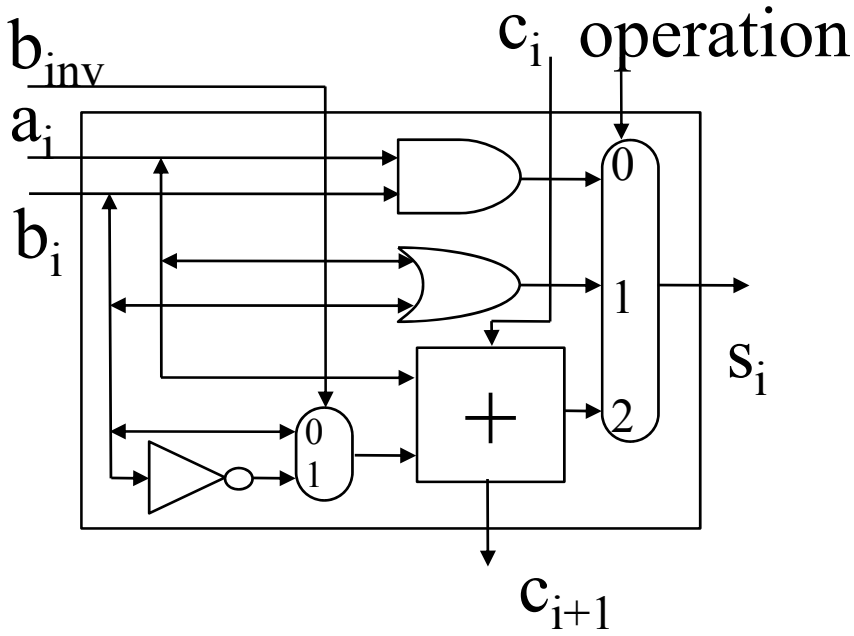
ALU implementing 'AND', 'OR', ADD, SUB

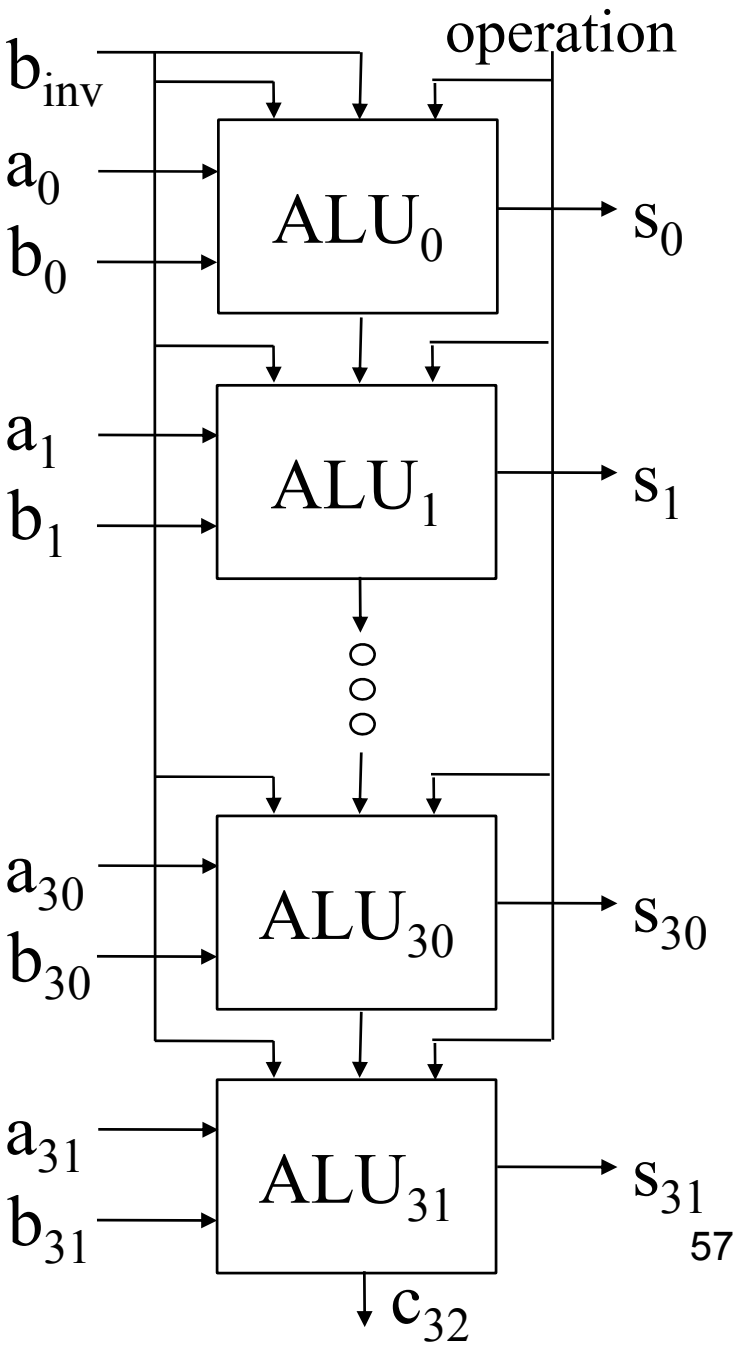




ALU Designed so far

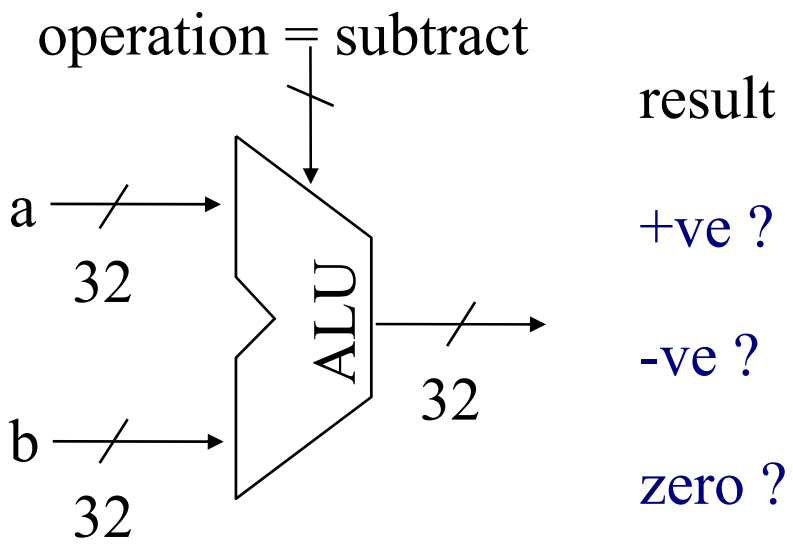




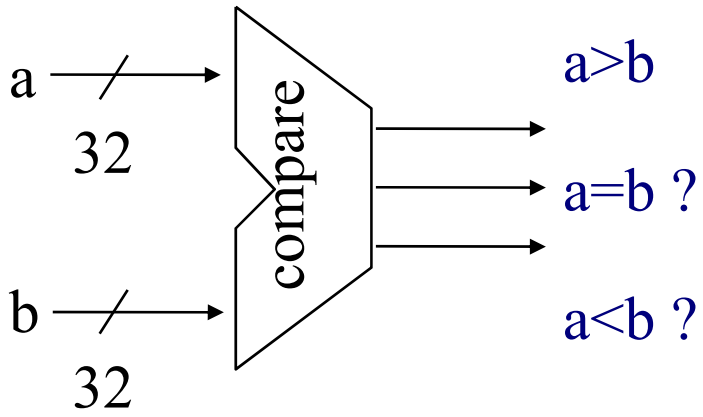


Comparing Two Integers

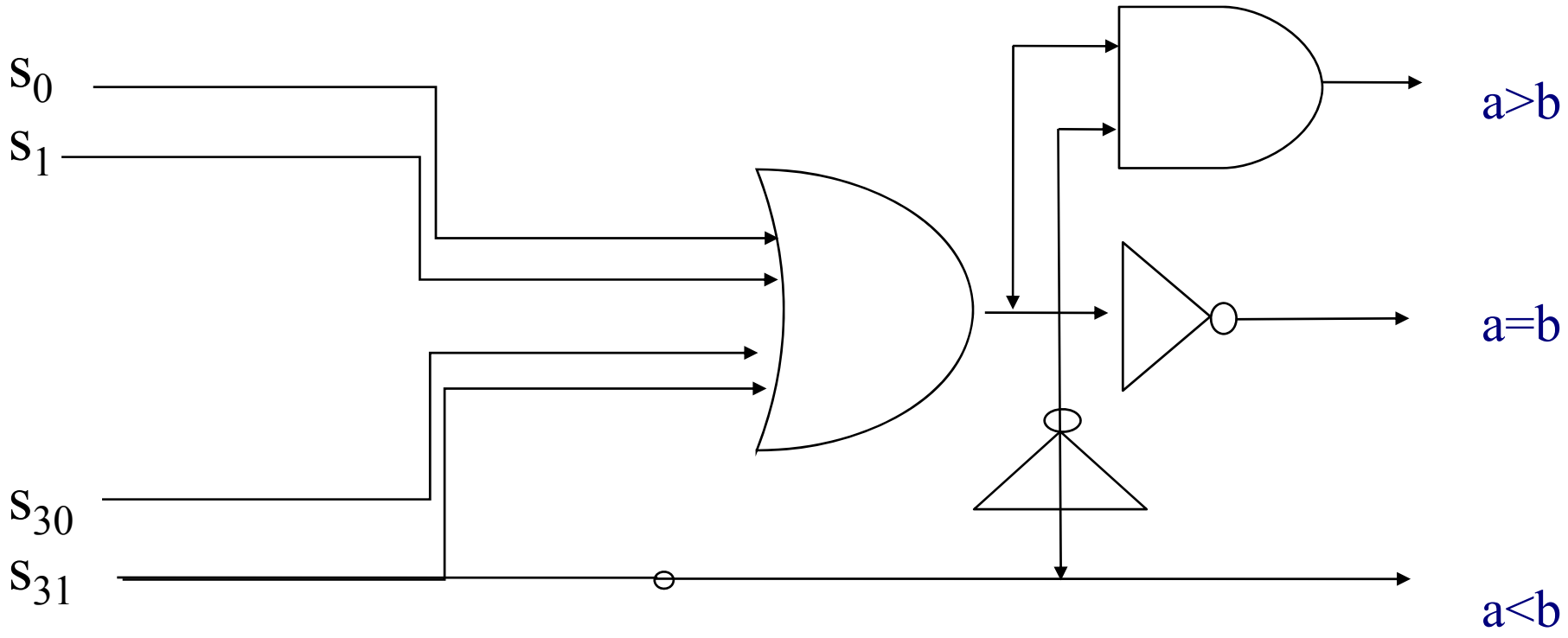
- Subtract and check the result



- Compare directly

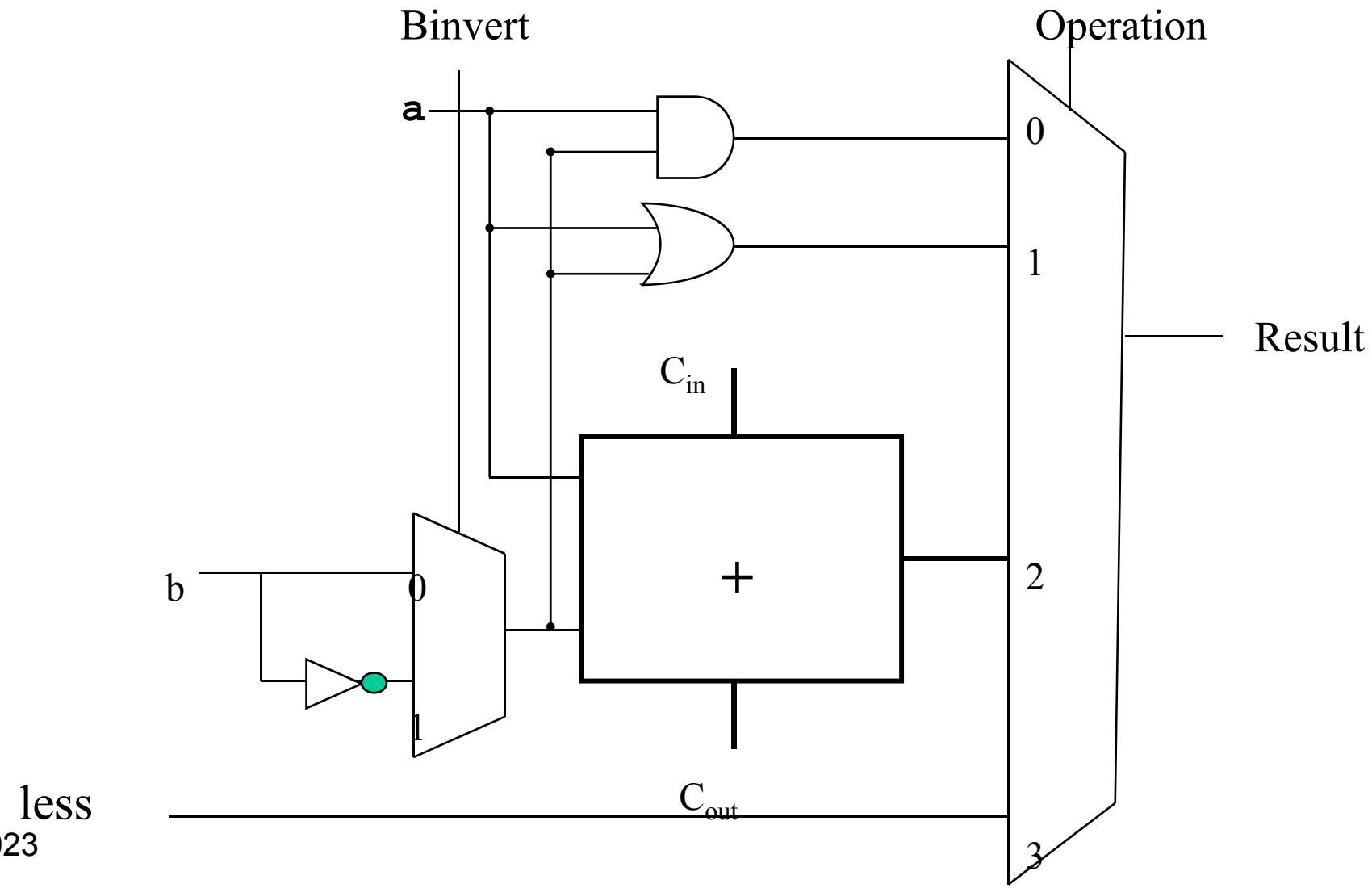


Subtract and Check the result

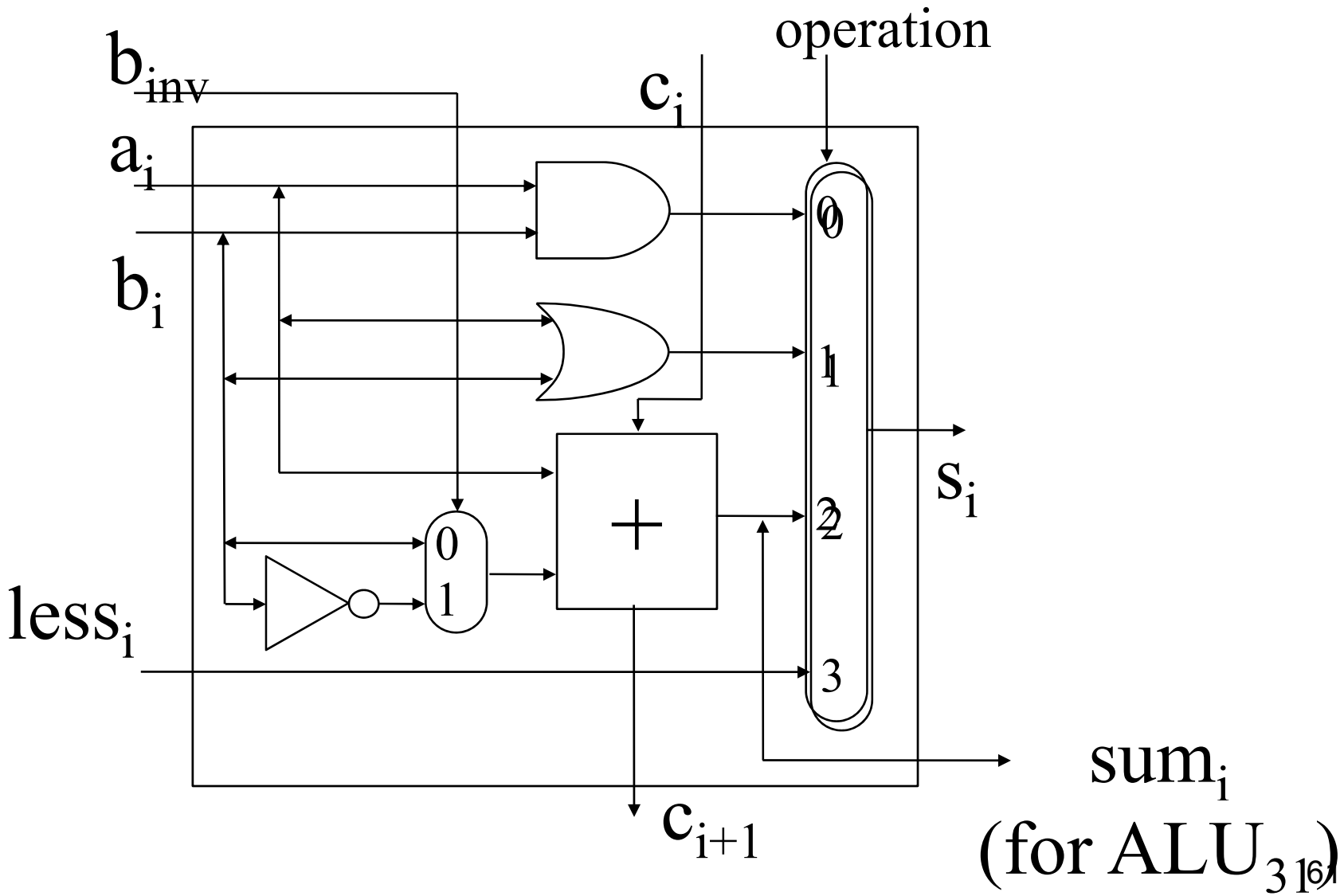


A 1-bit ALU

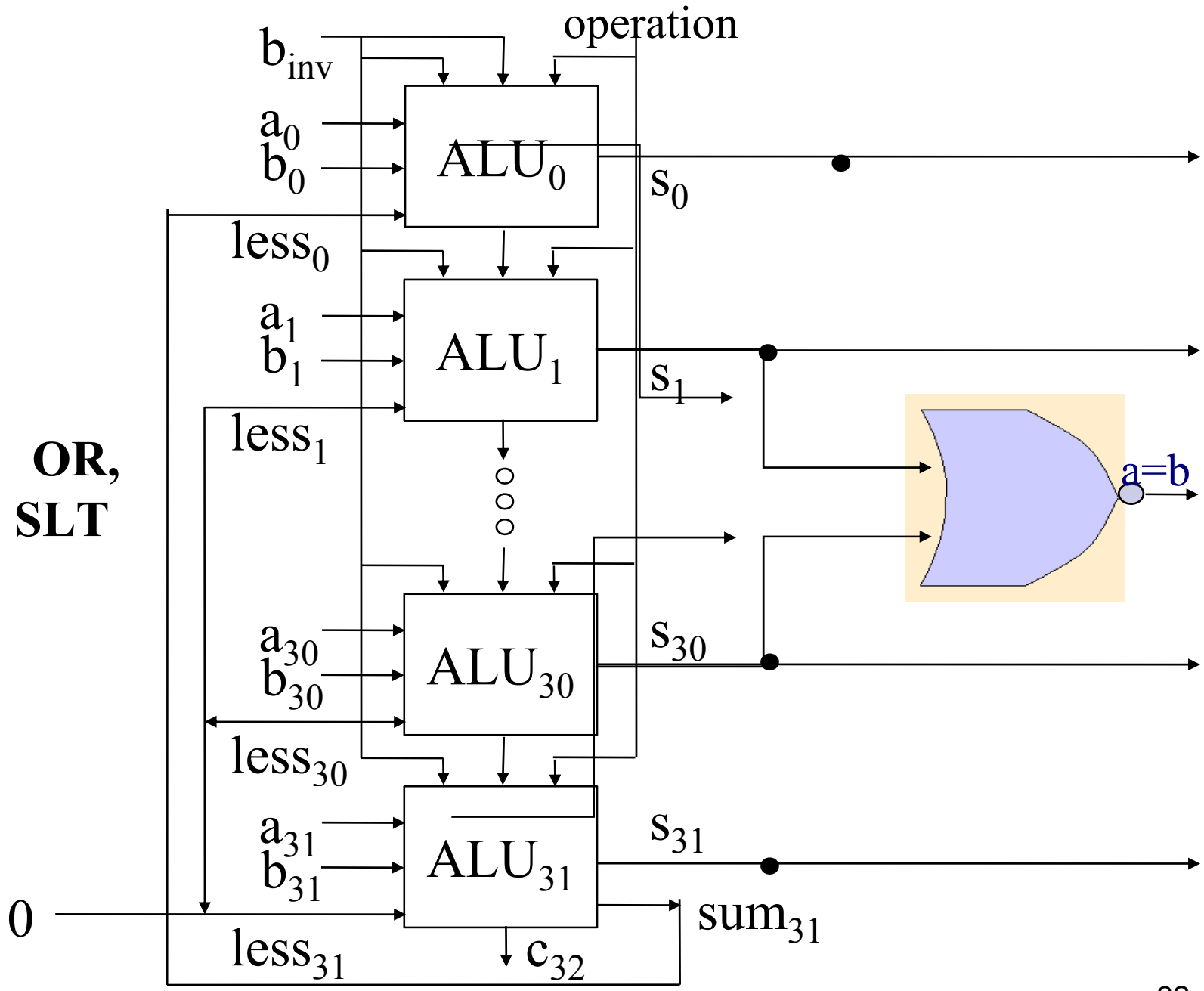
If (a-b) < 0 i.e. a < b, “Set on less” Flag = 1, otherwise 0



Extending ALUi for “slt”

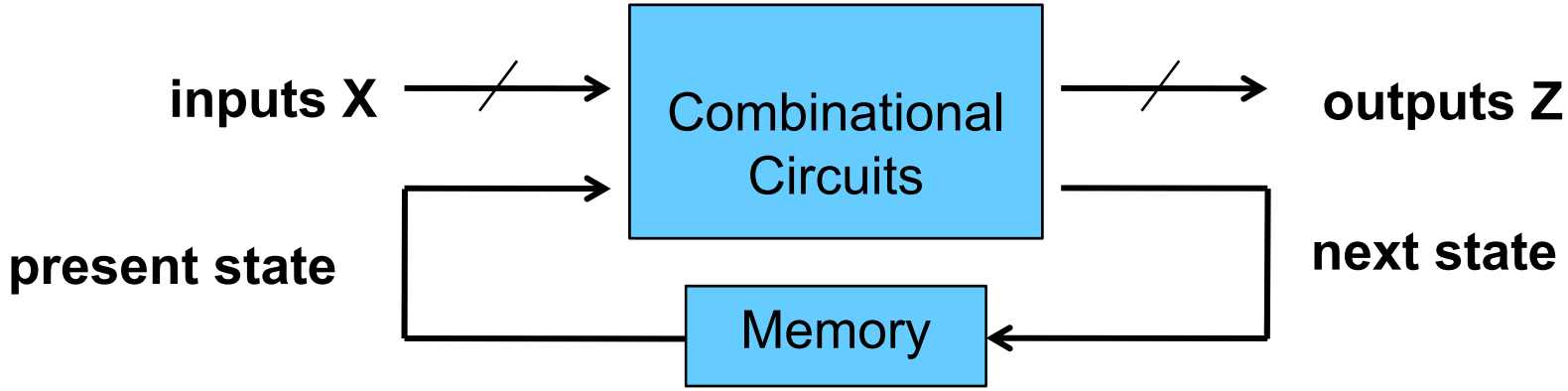


ALU for AND, OR, ADD, SUB, BEQ, SLT



Sequential Building Block

Sequential Block



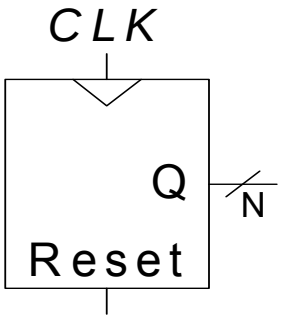
A sequential circuit:

- a combinational circuit with <u>feedback</u> through <u>memory</u>
  - stored information at any time defines a <u>state</u>
- Outputs depends on present inputs and previous outputs
  - Previous inputs are stored as binary information into memory
- Next state depends on inputs and present state

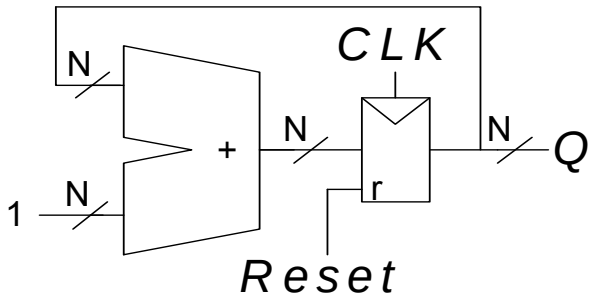
Digital Counters

- Increments on each clock edge.
- Used to cycle through numbers. For example,
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001…
- Example uses:
  - Digital clock displays
  - Program counter: keeps track of current instruction executing

Symbol



Implementation



Excitation Table

- Excitation table lists required inputs of flip-flops that will cause necessary transition from present state to next state.

Present State	Next State	***flip-flop JK***		***SR***		***D***	***T***
Q(t)	Q(t+1)	Input J	K	S	R	D	T
0	0	0	X	0	X	0	0
0	1	1	X	1	0	1	1
1	0	X	1	0	1	0	1
1	1	X	0	X	0	1	0

Digital Counter

Ripple Counter

- Ripple counter consists of cascaded JK F/Fs with JK lead wired high
- Output of each F/F is connected to CLK input of next F/F.
- F/F generating least significant bit of count receives input clock from clock generator

Mod-4 Ripple Counter

- Mod-4 that counts through 4 different states. All J and K inputs are connected to 1.

- Bubbles in clock input of each F/F indicates that F/Fs changes its state on a negative-going transition i.e. when a transition from 1 in present to 0 in next state of a F/F occurs, the state of next F/F will change.

- Sequence of mod-4 ripple counter is 0->1->2->3->0

Design Method

Table: Excitation of D flip-flop

	State Transition			
	$0\rightarrow0$	$0\rightarrow1$	$1\rightarrow0$	$1\rightarrow1$
D input	0	1	0	1

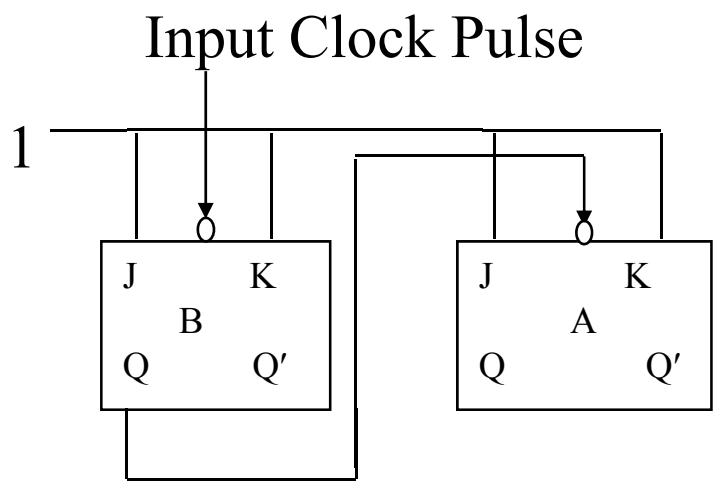
Table: Excitation of JK flip-flop

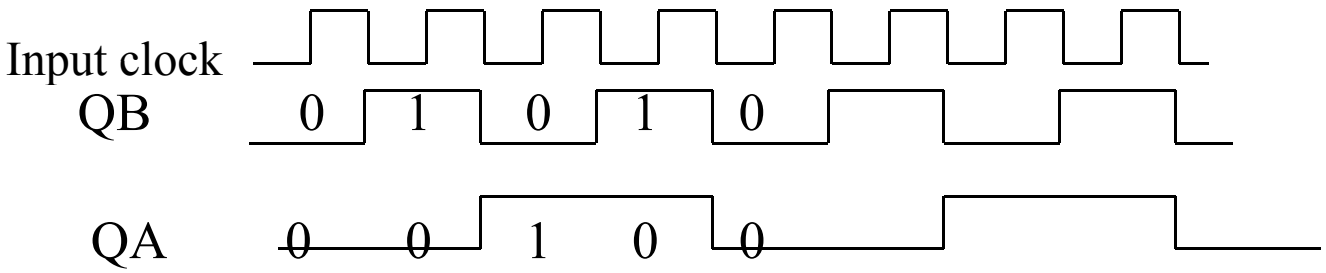
	State Transition			
	$0\rightarrow0$	$0\rightarrow1$	$1\rightarrow0$	$1\rightarrow1$
J	0	1	x	x
K	x	x	1	0

Table: Excitation of T flip-flop

F/F Input	State Transition			
	$0\rightarrow0$	$0\rightarrow1$	$1\rightarrow0$	$1\rightarrow1$
T	0	1	1	0

MOD-4 Ripple Counter



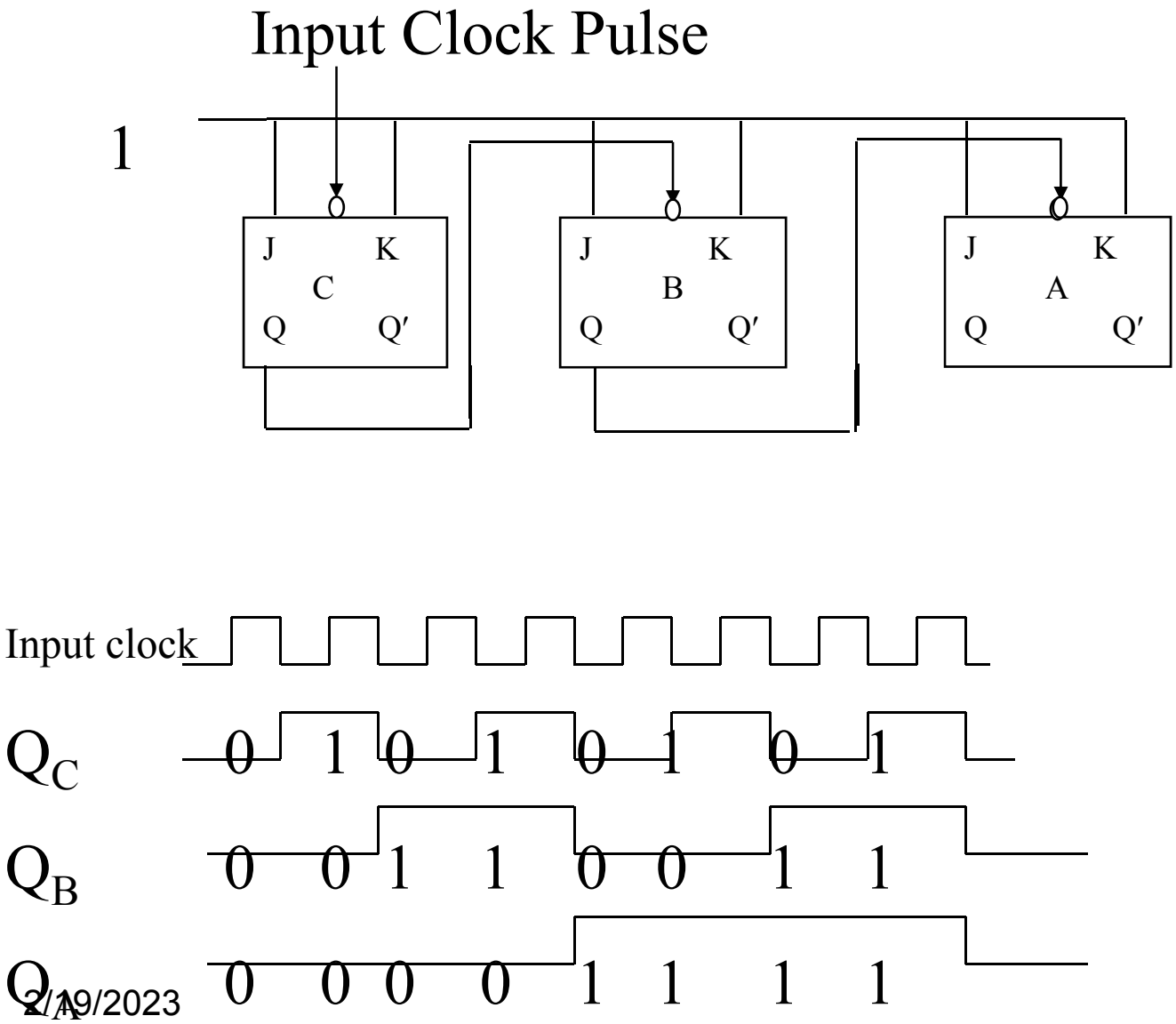


Timing Diagram of MOD-4 ripple counter

Sequence Table

Count Sequence A	B	Decimal
0	0	0
0	1	1
1	0	2
1	1	3

MOD-8 Ripple Counter

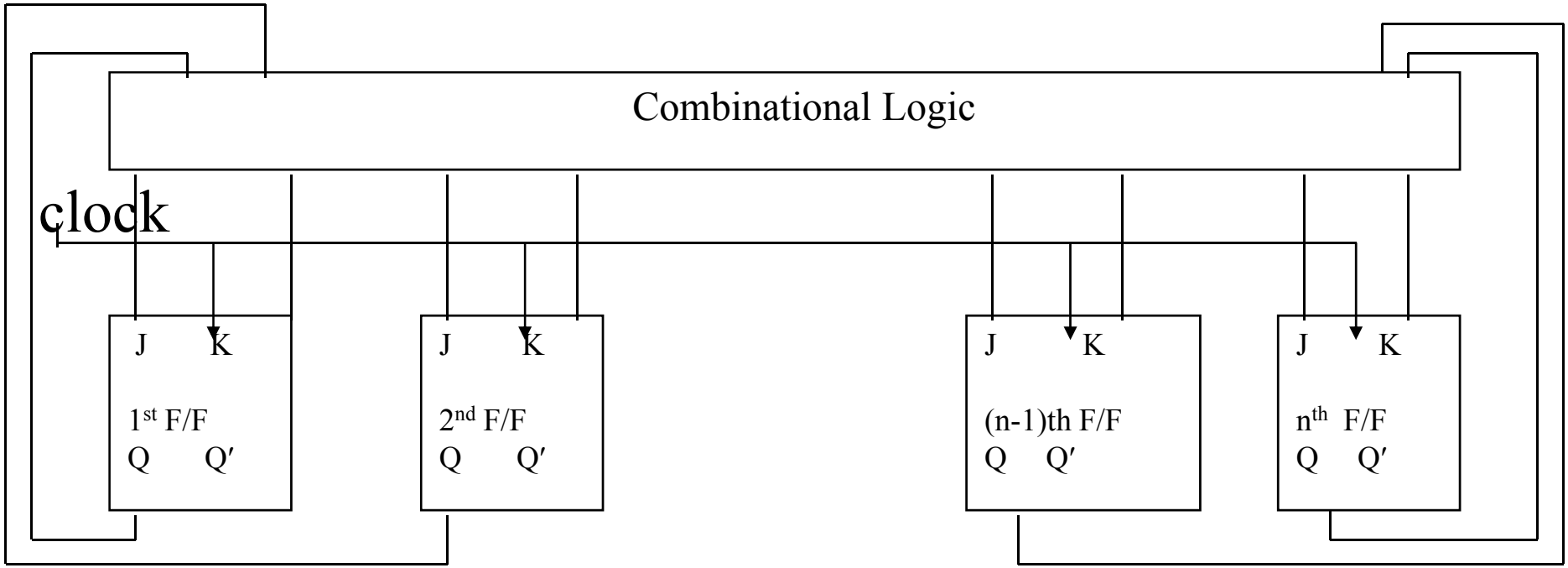


Count Sequence A	B	C	Decimal
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

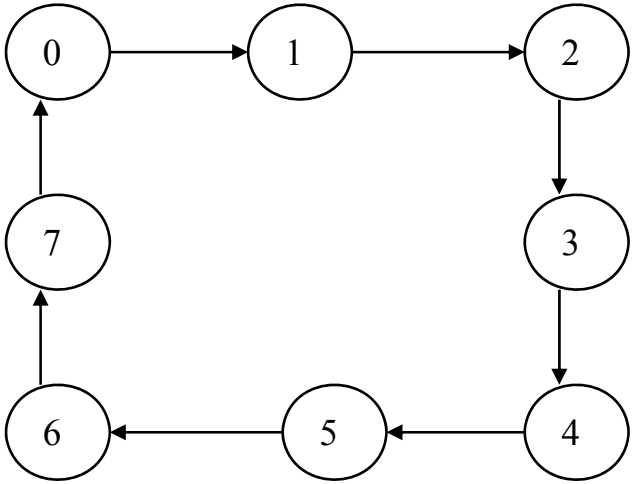
Synchronous Counter

- Normal synchronous state machines
- Normally edge triggered D or JK flip-flops are used as memory devices
- All the flip-flops are triggered synchronously by input pulse from a master clock generator
- Ripple counter connects all CLK leads to an outputs of the previous stage
- Clock signals from a common clock source are applied to CLK leads of all the flip-flops in synchronous counter at a time. For this reason, this type of counter is often called a parallel counter.

Synchronous Counter



Mod-8 Synchronous Counter



State diagram

Present State A B C	Next State A B C	Flip-Flop Inputs A F/F J_A	K_A	B F/F J_B	K_B	C F/F J_C	K_C
0 0 0	0 0 1	0	X	0	X	1	X
0 0 1	0 1 0	0	X	1	X	X	1
0 1 0	0 1 1	0	X	X	0	1	X
0 1 1	1 0 0	1	X	X	1	X	1
1 0 0	1 0 1	X	0	0	X	1	X
1 0 1	1 1 0	X	0	1	X	X	1
1 1 0	1 1 1	X	0	X	0	1	X
1 1 1	0 0 0	X	1	X	1	X	1

Excitation Table of the sequential circuit

0	0	1	0
x	x	x	x

J_A=BC

x	x	x	x
0	0	1	0

K_A=BC

Mod-8 Synchronous Counter

0	1	x	x
0	1	x	x

$J_B = C$

x	x	1	0
x	x	1	0

$K_B = C$

1	x	x	1
1	x	x	1

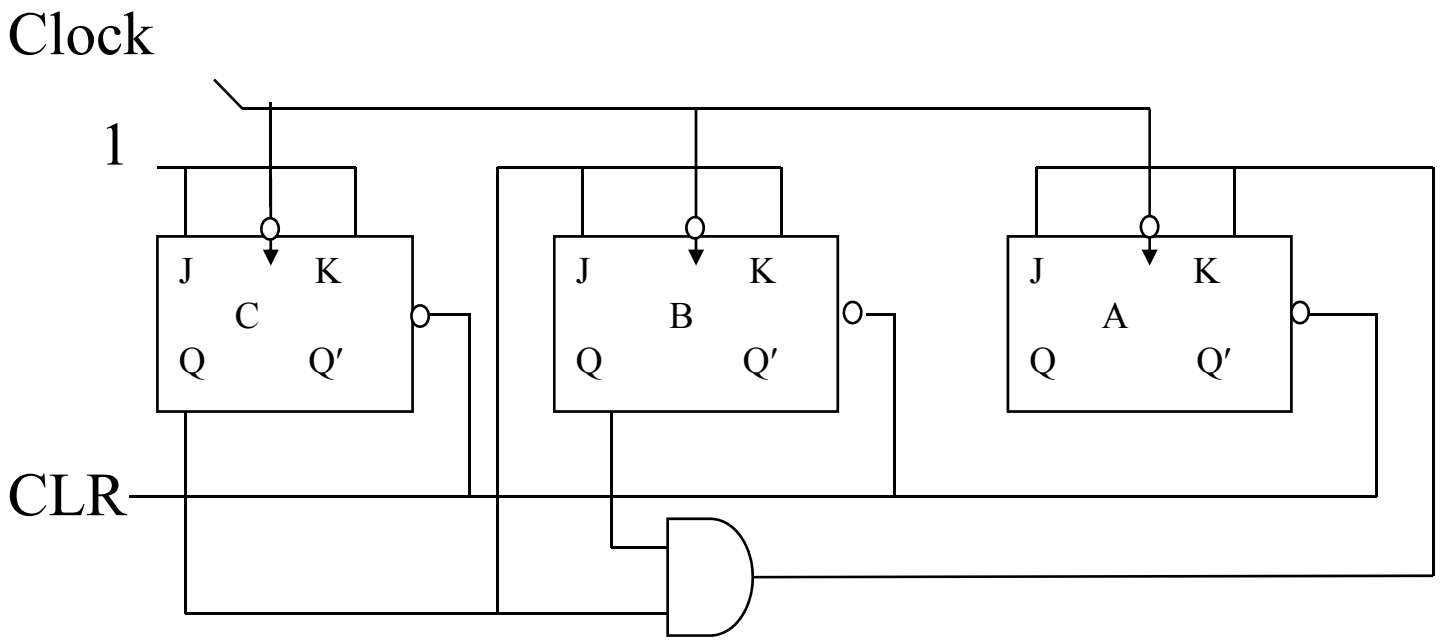
$J_C = 1$

x	1	1	x
x	1	1	x

$K_C = 1$

Mod-8 Synchronous Counter

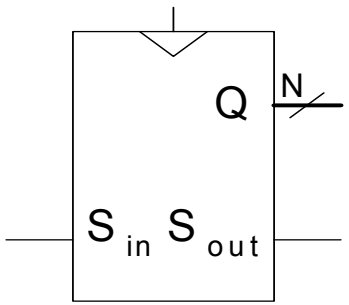
$J_A=K_A=BC$ $J_B=K_B=C$ $J_C=K_C=1$



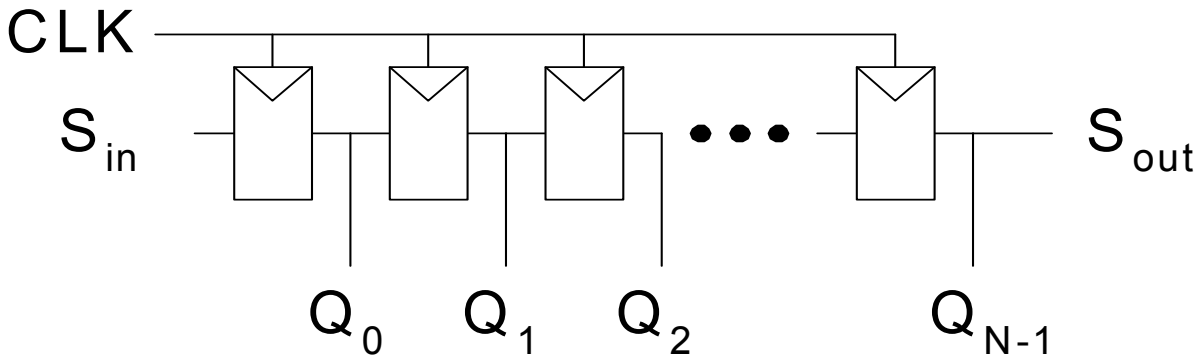
Shift Register

- Shift a new value in on each clock edge
- Shift a value out on each clock edge
- *Serial-to-parallel converter*: converts serial input (S_{in}) to parallel output ($Q_{0:N-1}$)

Symbol:

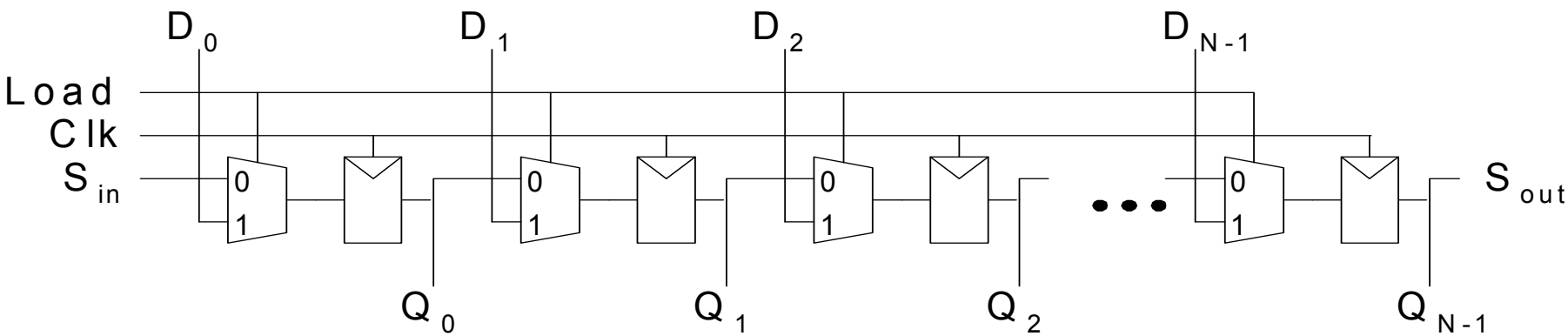


Implementation:



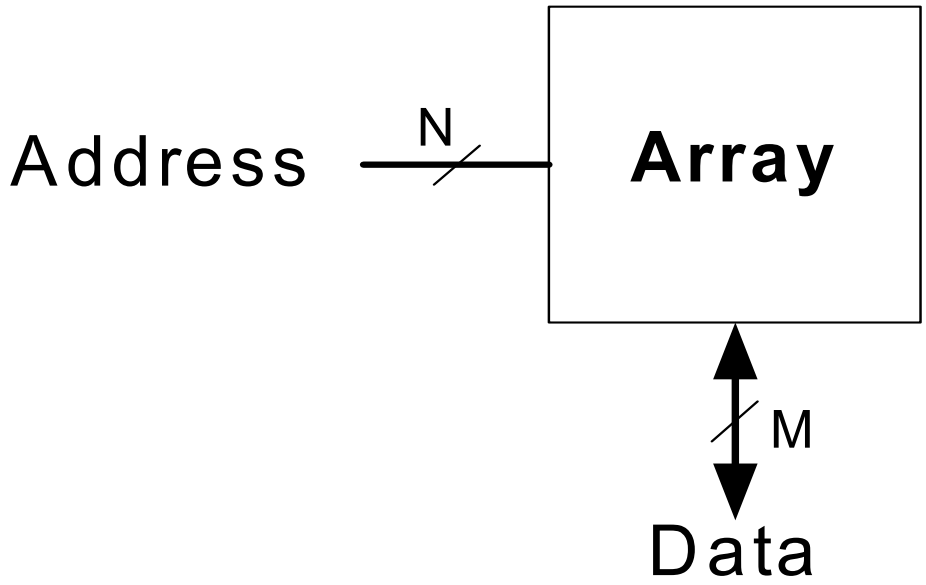
Shift Register with Parallel Load

- When *Load* = 1, acts as a normal *N*-bit register
- When *Load* = 0, acts as a shift register
- Now can act as a *serial-to-parallel converter* (S_{in} to $Q_{0:N-1}$) or a *parallel-to-serial converter* ($D_{0:N-1}$ to S_{out})



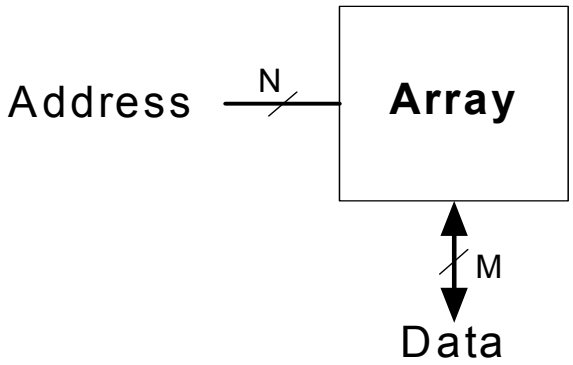
Memory Arrays

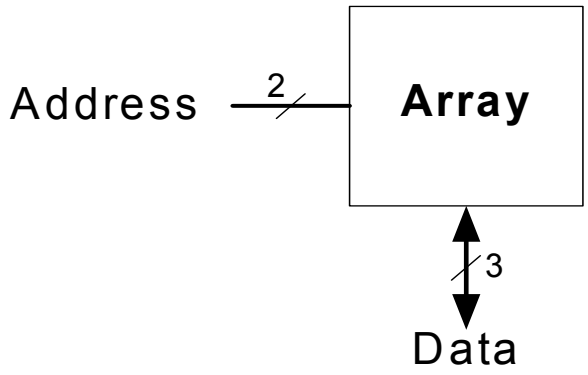
- Efficiently store large amounts of data
- Three common types:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- An *M*-bit data value can be read or written at each unique *N*-bit address.

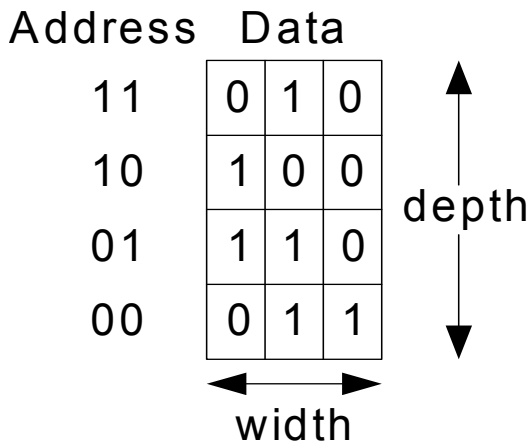


Memory Arrays

- Two-dimensional array of bit cells
- Each bit cell stores one bit
- An array with *N* address bits and *M* data bits:
  - 2^N rows and *M* columns
  - Depth: number of rows (number of words)
  - Width: number of columns (size of word)
  - Array size: depth × width = $2^N \times M$



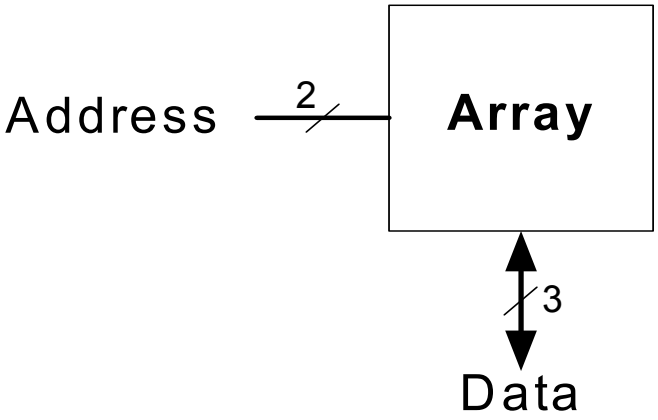




Memory Array: Example

- $2^2 \times$ 3-bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100

Example:

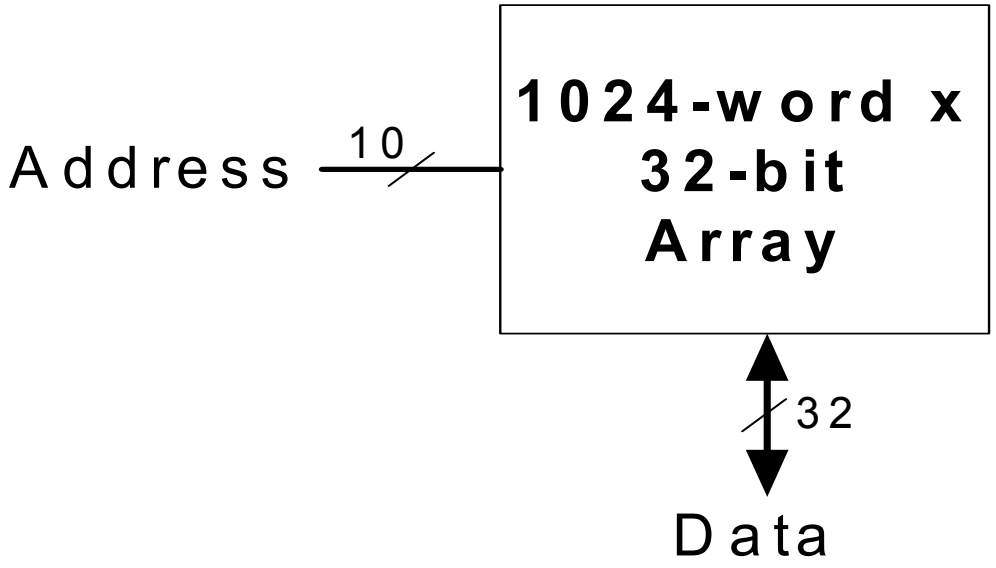


Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

depth

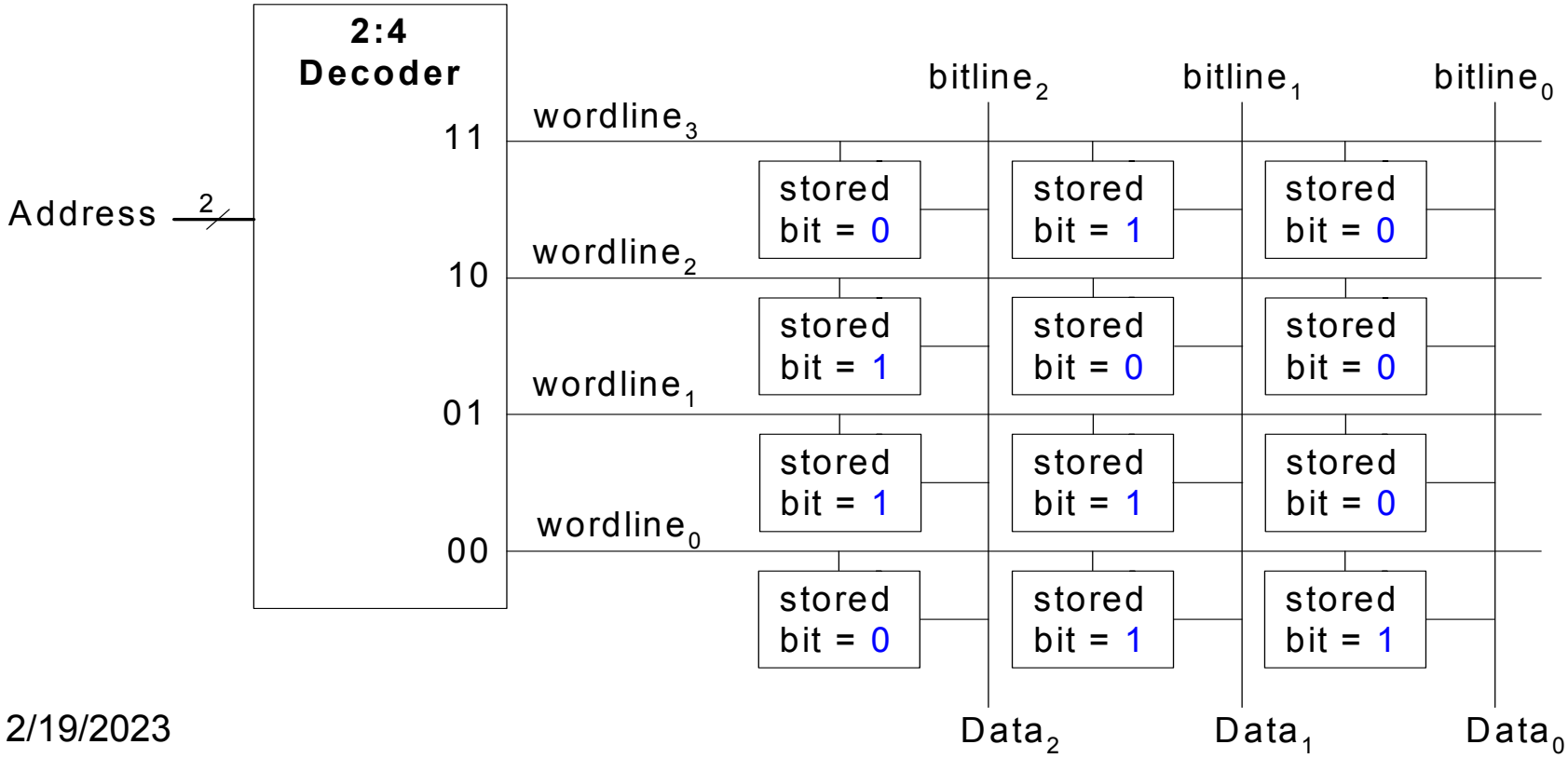
width

Memory Arrays

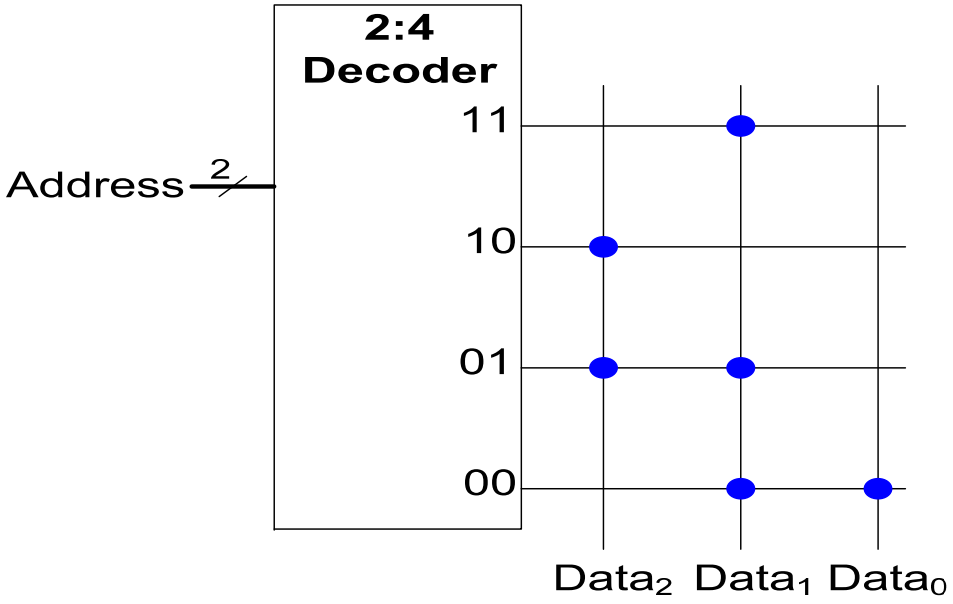


Memory Array

- Wordline:
  - similar to an enable
  - allows a single row in the memory array to be read or written
  - corresponds to a unique address
  - only one wordline is HIGH at any given time



ROM Storage

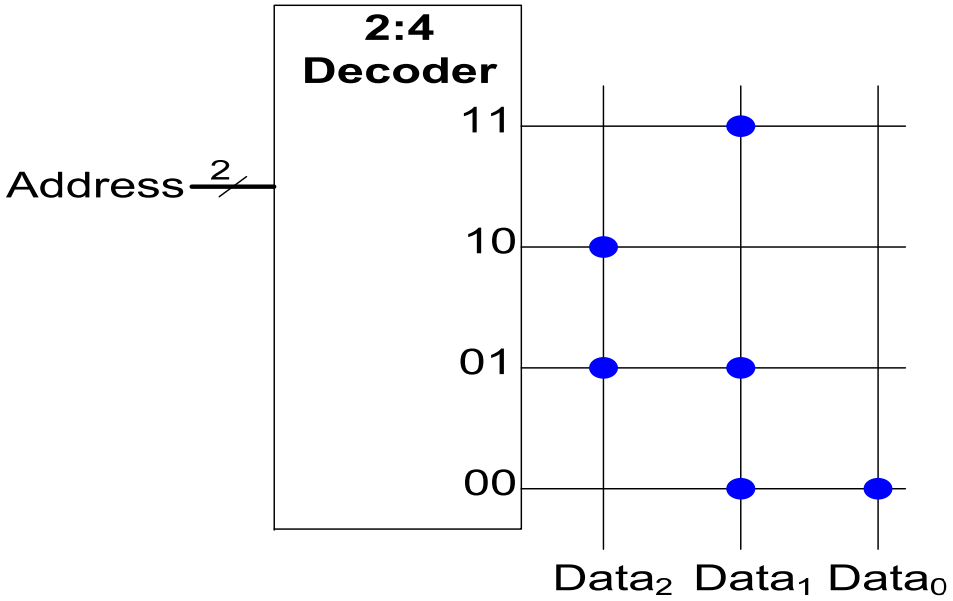


Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

depth

width

ROM Logic



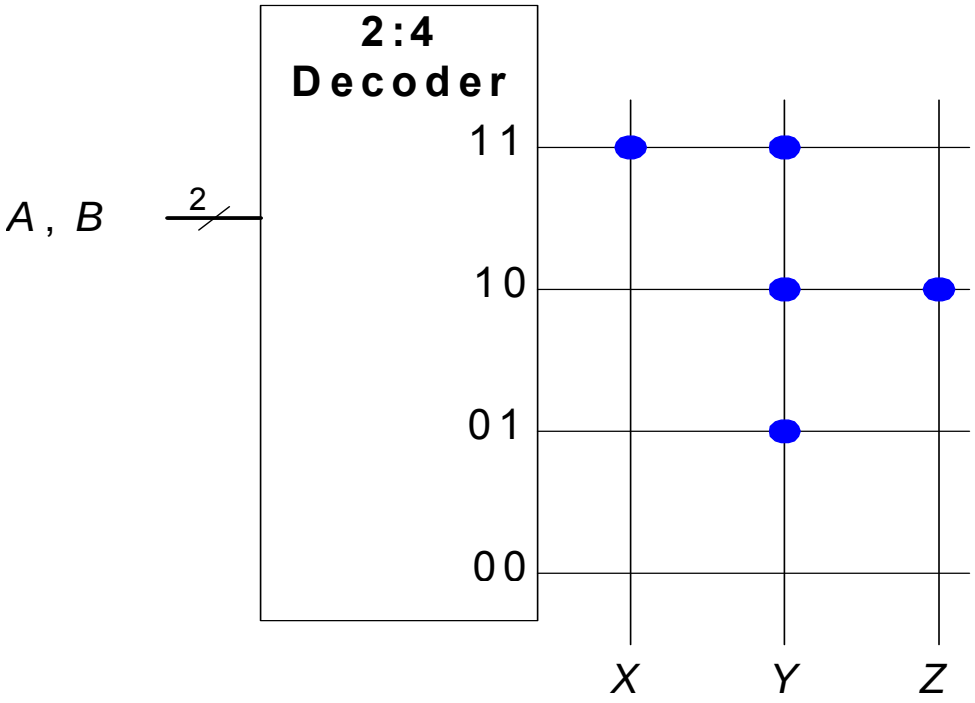
$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \overline{A}_1 + A_0$$

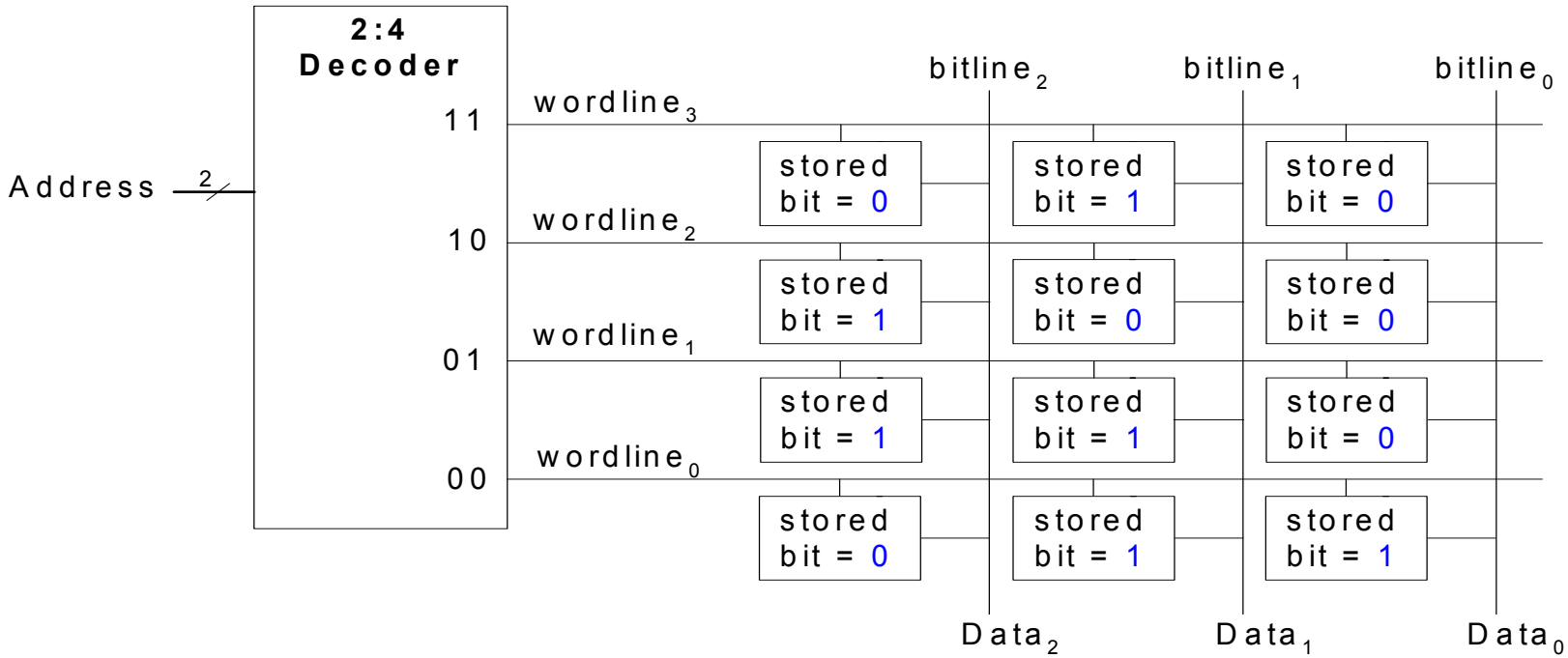
$$Data_0 = \overline{A}_1 \overline{A}_0$$

Example: Logic with ROMs

- Implement the following logic functions using a $2^2 \times 3$-bit ROM:
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$



Logic with Any Memory Array



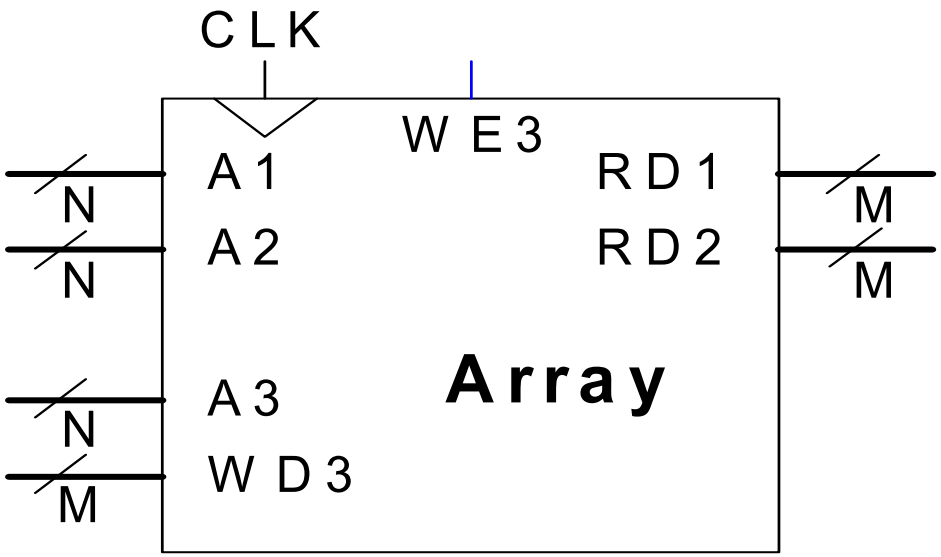
$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \overline{A_1} + \overline{A_0}$$

$$Data_0 = \overline{A_1}\,\overline{A_0}$$

Multi-ported Memories

- Port: address/data pair
- 3-ported memory
  - 2 read ports (A1/RD1, A2/RD2)
  - 1 write port (A3/WD3, WE3 enables writing)
- Small multi-ported memories are called *register files*



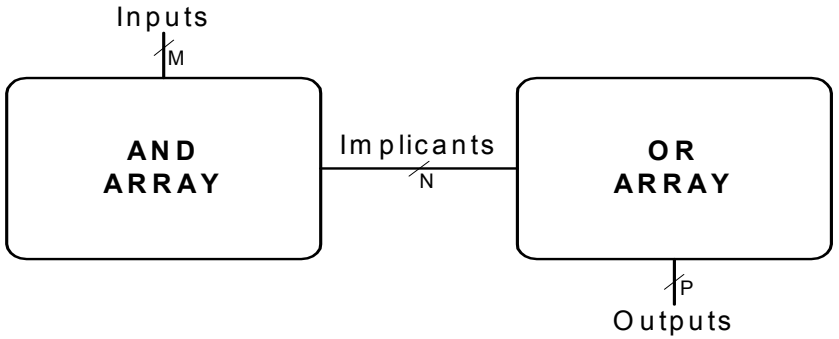
Logic Arrays

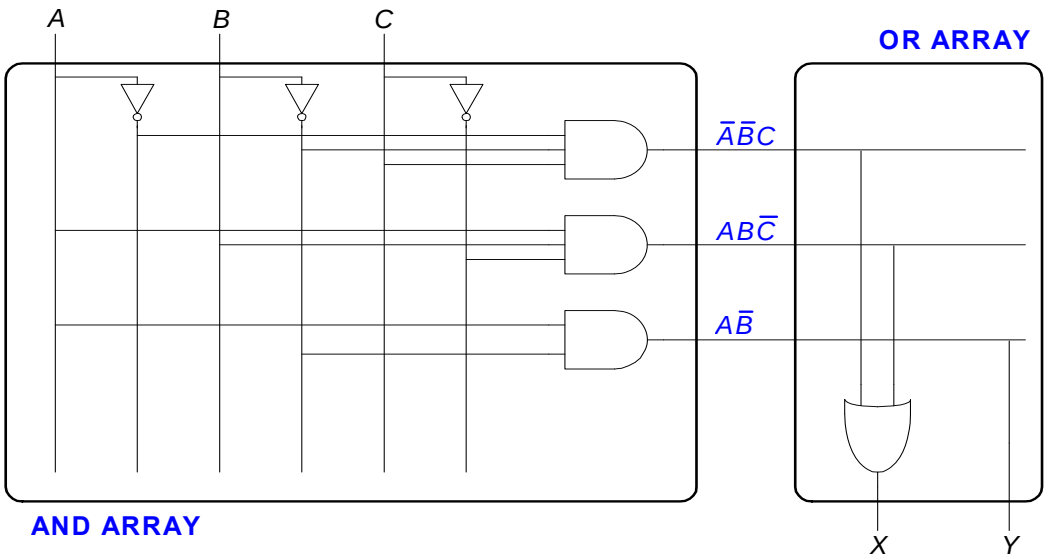
- Programmable logic arrays (PLAs)
  - AND array followed by OR array
  - Perform combinational logic only
  - Fixed internal connections

- Field programmable gate arrays (FPGAs)
  - Array of configurable logic blocks (CLBs)
  - Perform combinational and sequential logic
  - Programmable internal connections

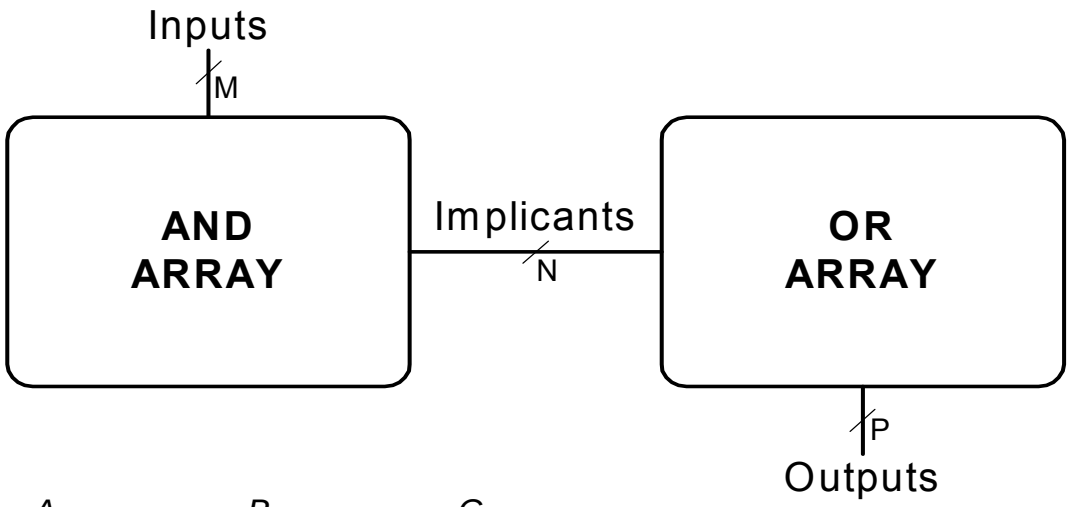
PLAs

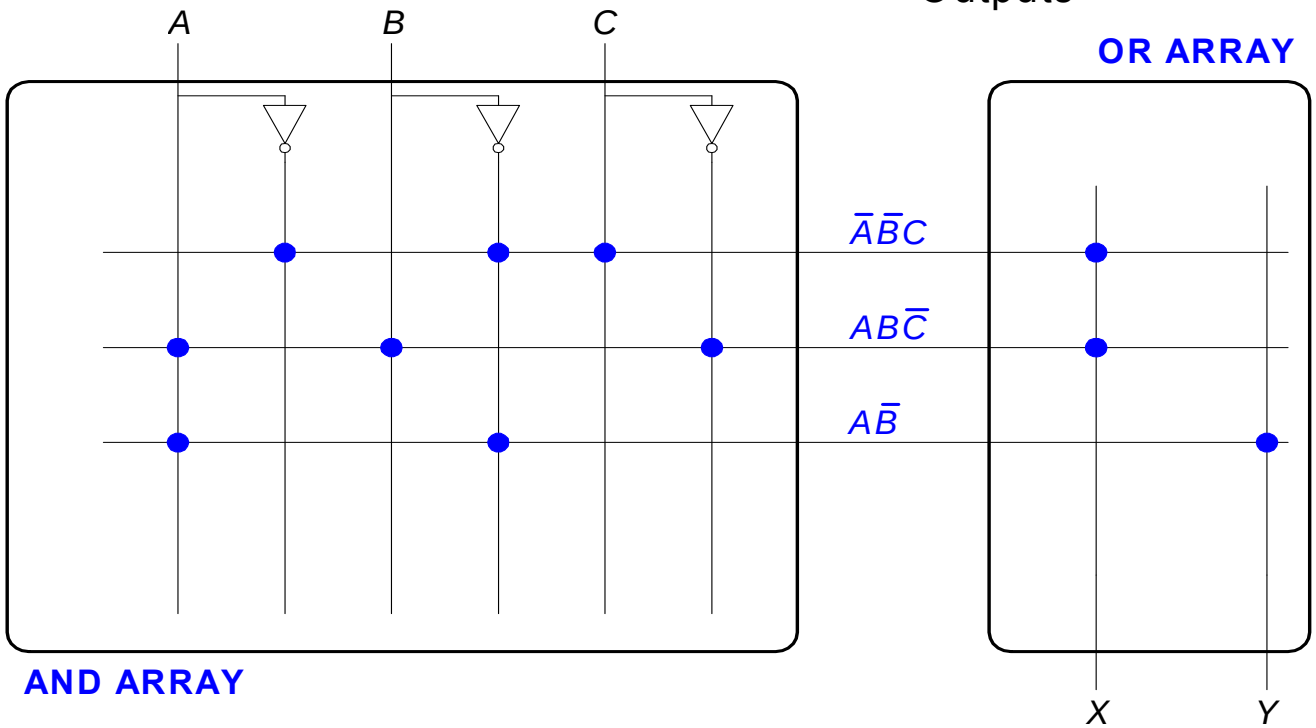
- $X = \bar{A}\bar{B}C + AB\bar{C}$
- $Y = A\bar{B}$





PLAs: DOT Notation





Assignments

1. Specify the size of a ROM that you could use to program each of the following combinational circuits.

   A 16-bit adder/subtractor with C_{in} and $_{Cout}$

2. Implement the following functions using a single 16 x 3 ROM. Use dot notation to indicate the ROM contents.

   (a) $X=AB+BC'D+A'B'$ (b) $Y=AB+BD$ (c) $Z=A+B+C+D$

3. Design a 32 bit ALU implementing MIPS instructions**: AND, OR, ADD, SUB, BEQ, SLT**