Register Transfer & microoperations; **A**rithmetic **L**ogic **U**nit

uOttawa

L'Université canadienne
Canada's university

Dr. Voicu Groza

SITE Hall, Room 5017
562 5800 ext. 2159
VGroza@uOttawa.ca

Université d'Ottawa | University of Ottawa

www.uOttawa.ca

uOttawa

Outline

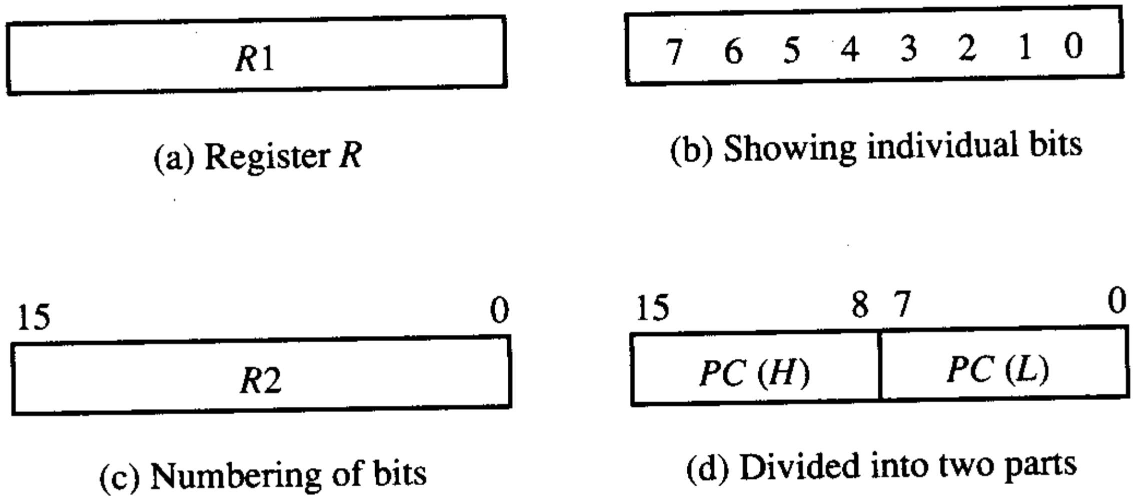
- Register Transfer Language
- Register Transfer
- Bus & memory transfer
- Arithmetic microoperations
- Logic and Shift microoperations
- ALU Hardware Implementation

Register Transfer Language

- The internal hardware organization of a digital system, such as a digital computer for instance, is best defined by specifying:
  1. The set of registers it contains and their functions.
  2. The set of microoperations that can be performed on the binary information stored in each register.
  3. The control that initiates the sequence of such microoperations.
- An operation executed on data stored in registers within **one** clock cycle is called **microoperation**.
- Shift, count, clear, and load, are examples of microoperations.
- The symbolic notation used to describe the micro-operation transfers among registers is called **Register Transfer Language**.

Registers

Figure 4-1 Block diagram of register.



- Computer registers are designated by capital letters, possibly followed by numerals.
  - R1: general purpose data register.
  - MAR (Memory Address Register): a register to hold the address of the word to be fetched from memory.
  - PC: program counter register.
  - IR: instruction register.
- In the figure above, PC(L), or PC(0 -7), refers to the low-order byte, whereas PC(H), or
- PC(8 -15), refers to the high-order byte.

Register Transfer

- The statement R2 ← R1 denotes a transfer of the content of register R1 into register R2.
- By definition, the content of the source register, R1 does not change after the transfer.
- The statement R2 ← R1, R1 ← R2 denotes that the two transfer operations are to be executed at the same time (by the same clock impulse). In this particular case, it leads to the exchange of the contents of R1 and R2.

Basic symbols for register transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	MAR; R2
Parentheses ( )	Denotes a part of a register	R2(0-7), R2(L)
Left arrow	Denotes transfer of information	R2 R1
Comma ,	Separates two microoperations	R2 R1 , R5 MAR

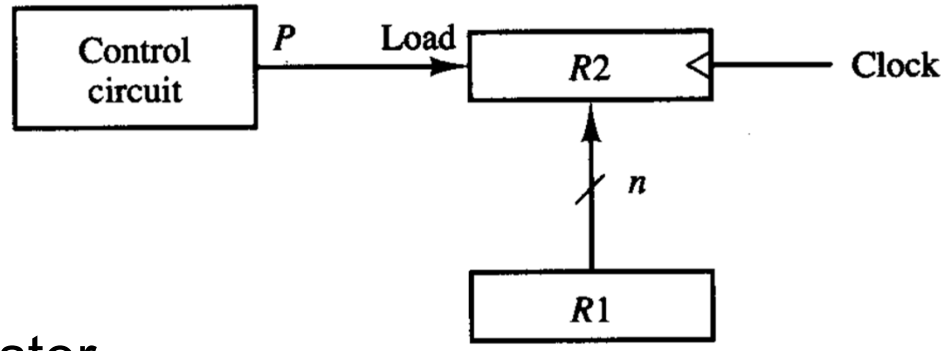
Control Function

- A control function is a Boolean variable. It is included in a register transfer statement as follows:

  P : R2 ← R1

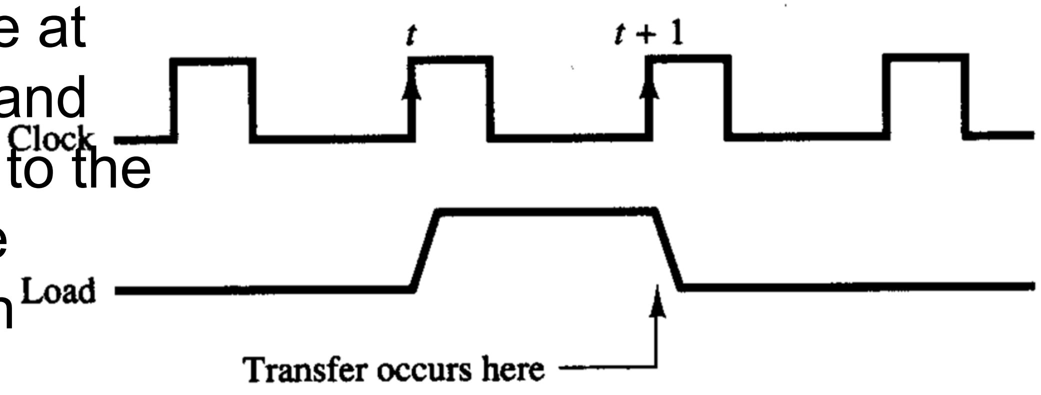
- The above statement means:

  if (P = 1) then (R2 ← R1)

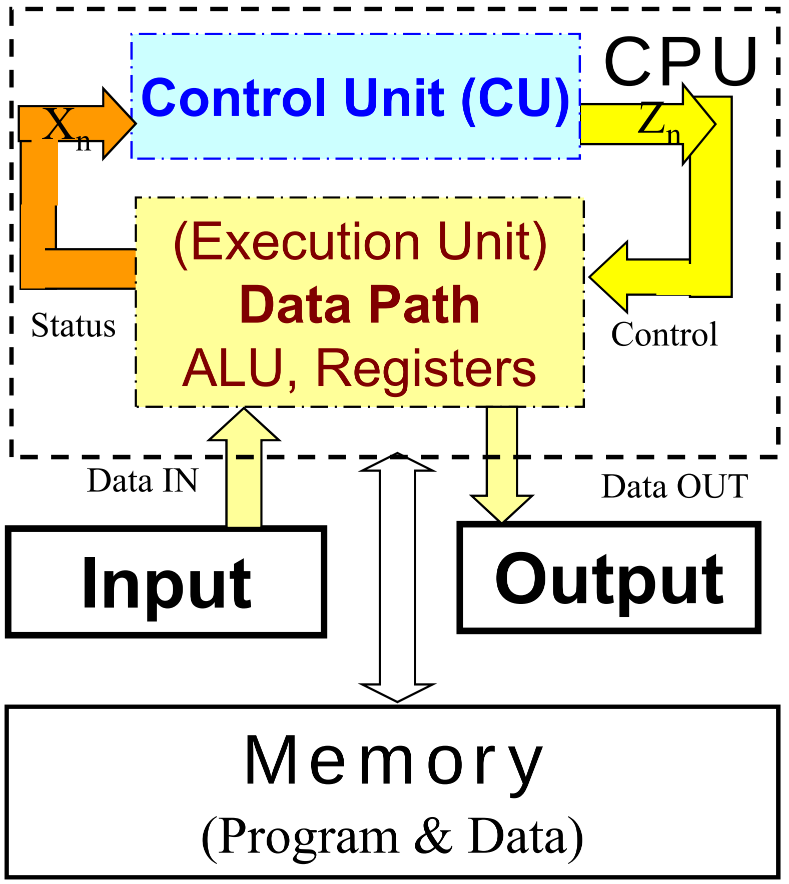


(a) Block diagram

- *n* denotes the number of bits in a register.
- P is activated by the rising edge of a clock pulse at time t.
- The next transition of the clock pulse at time t + 1 finds the load input active and the data input lines of R2 connected to the data output lines R1. At this time, the contents of R1 gets loaded into R2 in parallel (parallel loading).



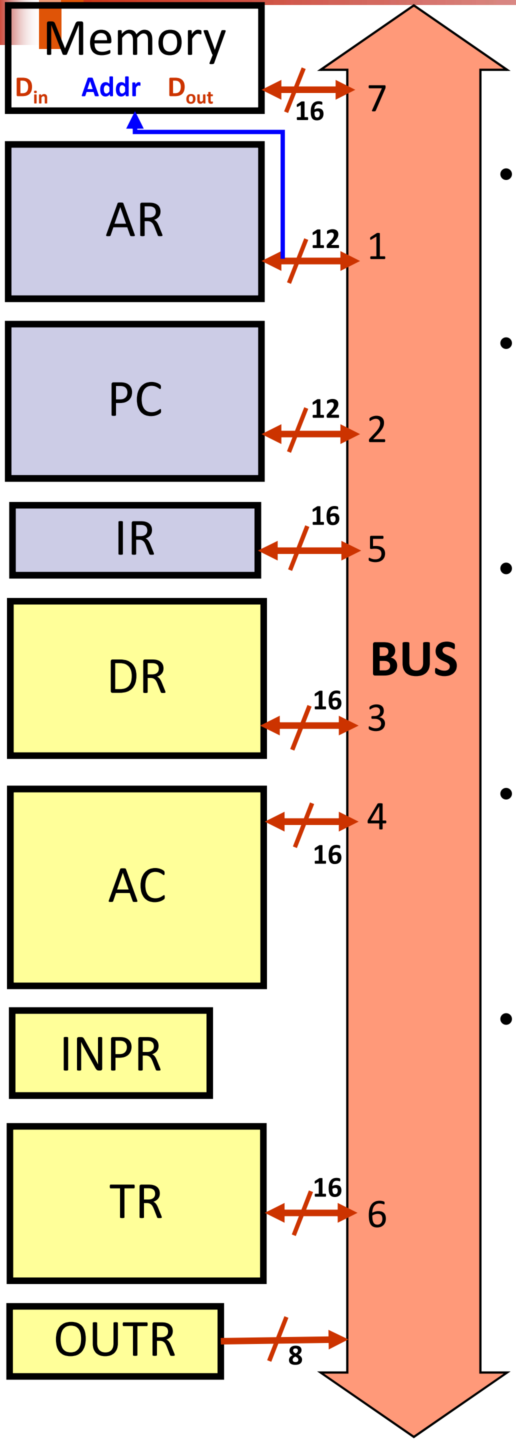
Structure of a Basic Computer



1. The computer *accepts* external information in the form of sequence of **instructions** (programs) and **data** through an input unit and *stores* them in the memory.
2. In the **CPU** (Central Processing Unit): the *data* stored in the memory is *fetched*, under the *program* control, and *processed* in the ALU of the **Data Path**.
3. The processed data *leaves* the computer through an *output unit*
4. All activities inside the computer are *directed* by the Control Unit (CU).

The **Control Unit (CU)** of the CPU is a “***programmable***” *sequential circuit* which changes its transition function depending on the instruction to be performed.

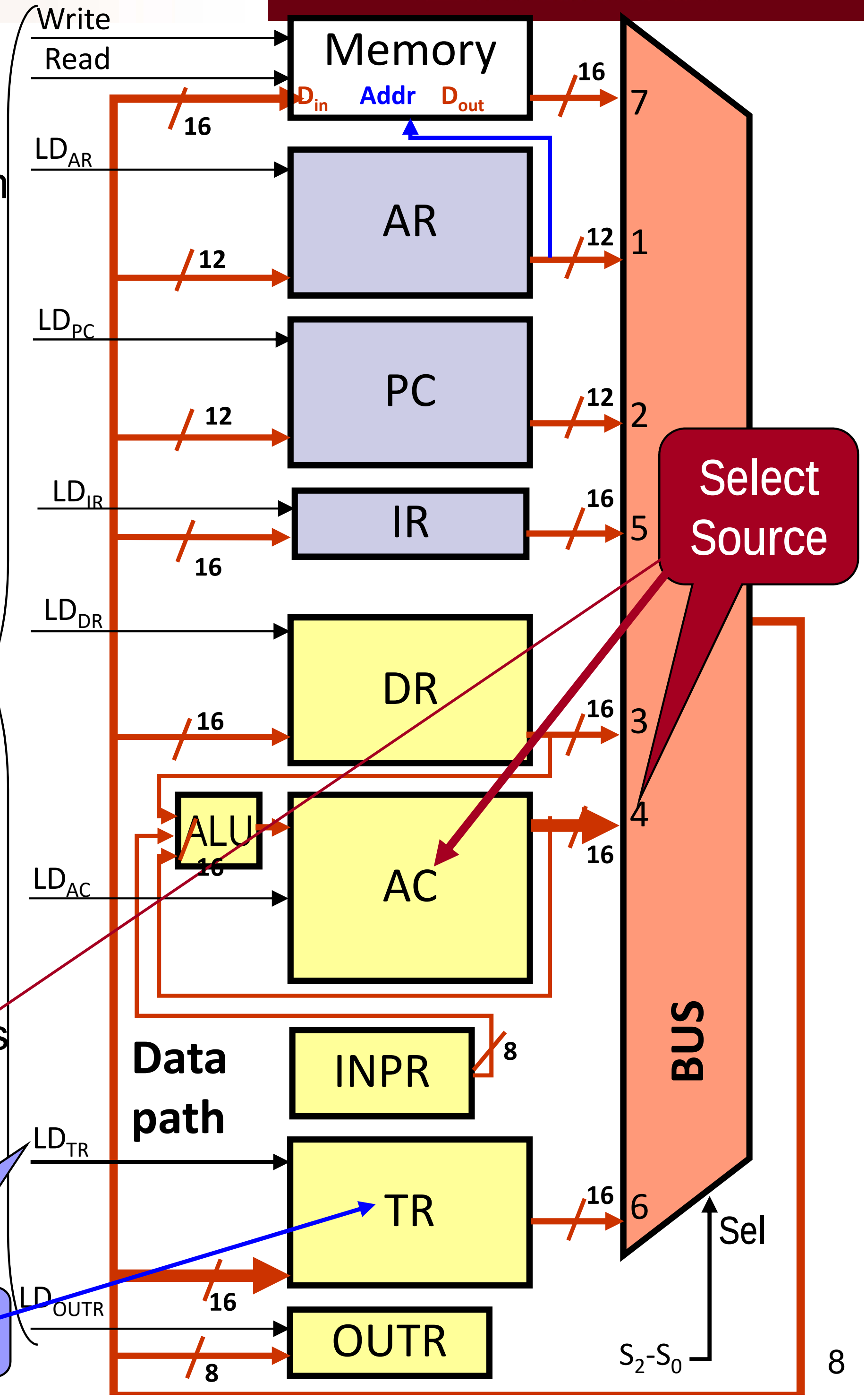
Common Bus



- A typical digital computer system contains a large number of registers.
- The number of wires will be excessive if separate wires are used to connect each register to all the other registers.
- A more efficient scheme of transferring information between registers is a common bus system.
- A bus structure consists of common lines, one for each bit of a register, through which binary information is transferred.
- Control signals, or bus selection bits, determines which register is selected by the bus during each particular register transfer.

TR ← AC

(destination ← source)



uOttawa

Info Transfer

- The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control bit of the selected **destination** register.
- The transfer of information between two registers through the bus can be symbolized explicitly by
  1. $BUS \leftarrow AC$ (Sel = $S_2S_1S_0 = 100_2$)
  2. $TR \leftarrow BUS$ (LD_{TR})
- or implicitly by

$$TR \leftarrow AC$$

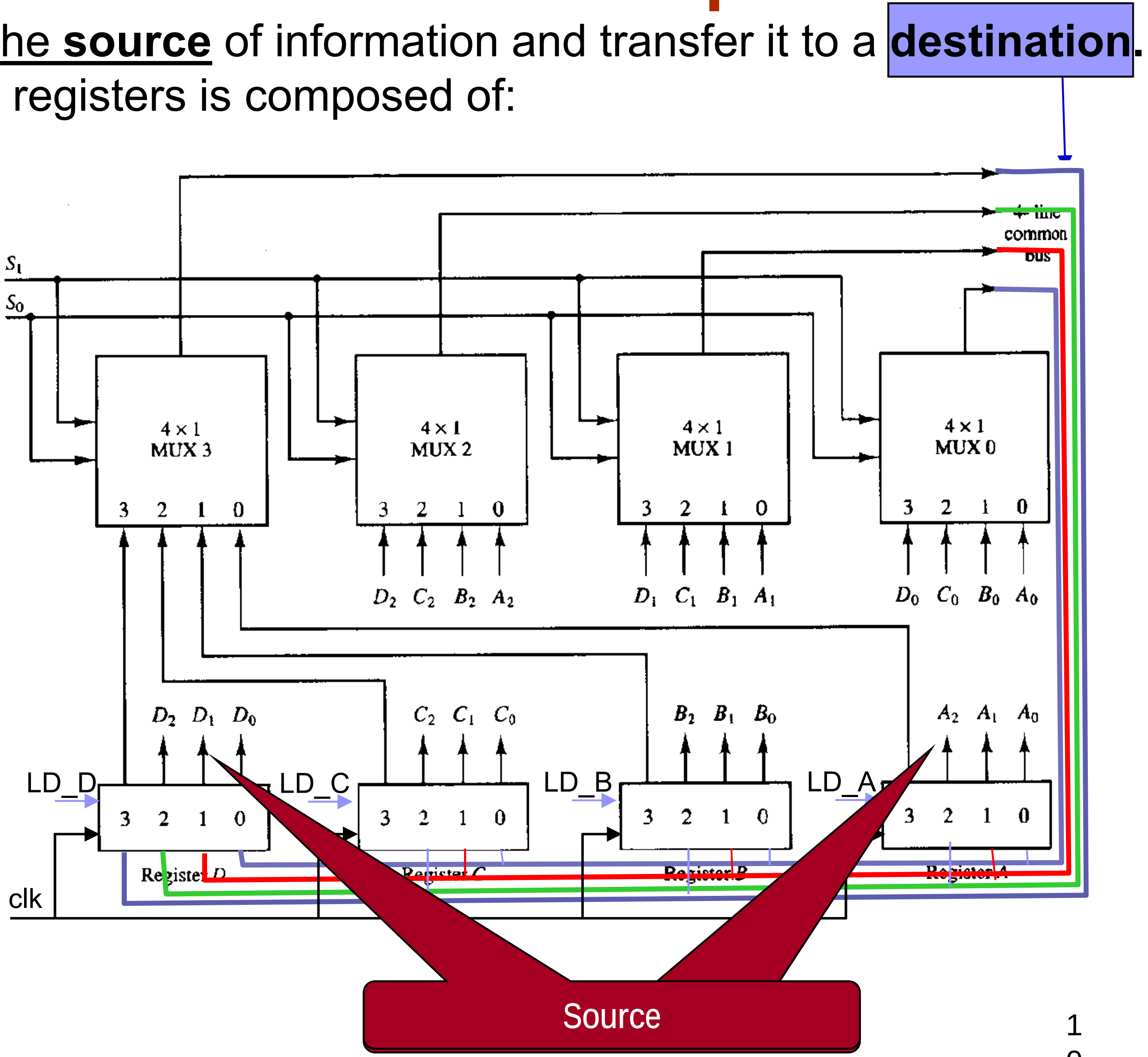
Common Bus With Multiplexers

A **bus** is employed to select the **source** of information and transfer it to a **destination**.
An *n*-line bus multiplexing 2^k registers is composed of:

- 2^k registers with *n* bits each (since it is an *n*-line bus)
- *n* $2^k \times 1$-multiplexers: each multiplexer has 2^k data bits, *k* selection bits, and 1 output bit.

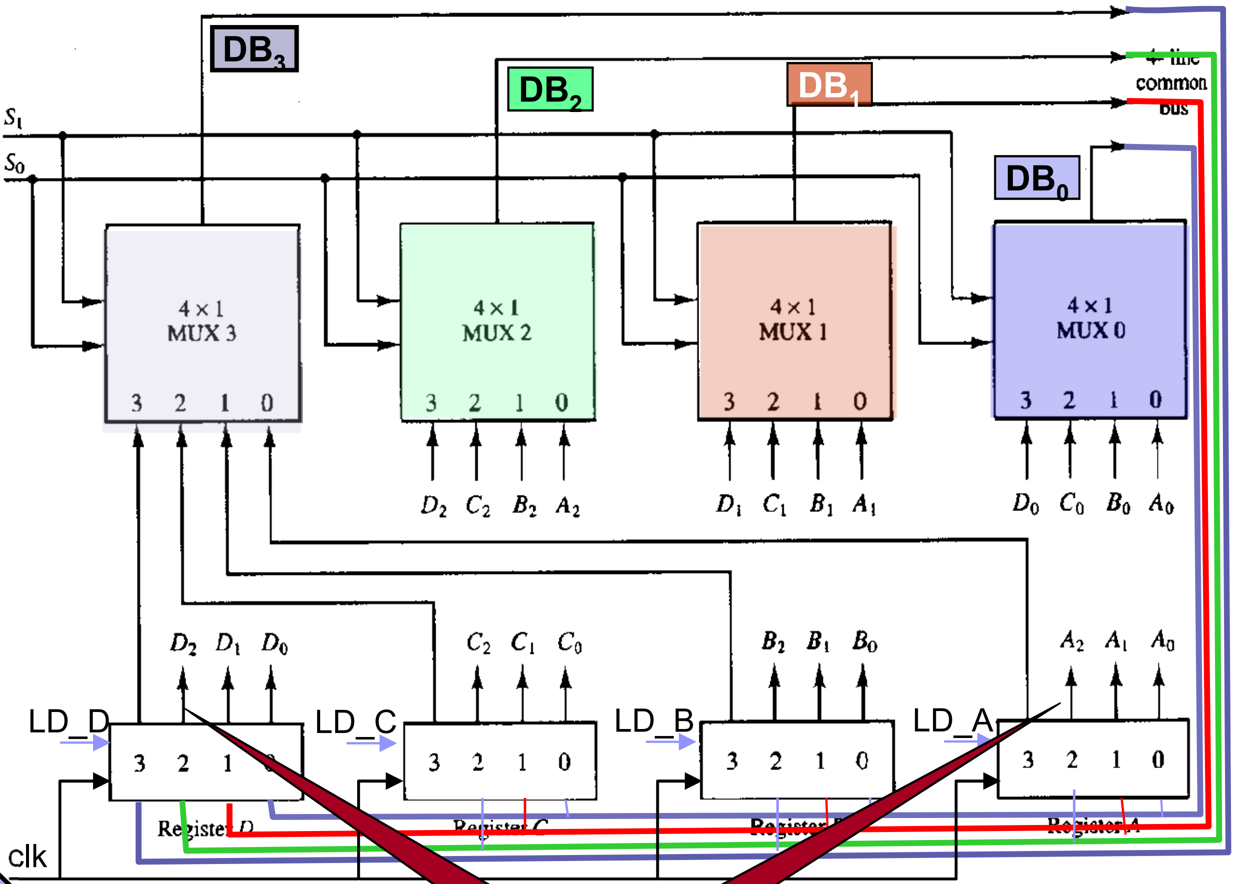
e.g., n=4 k=2:

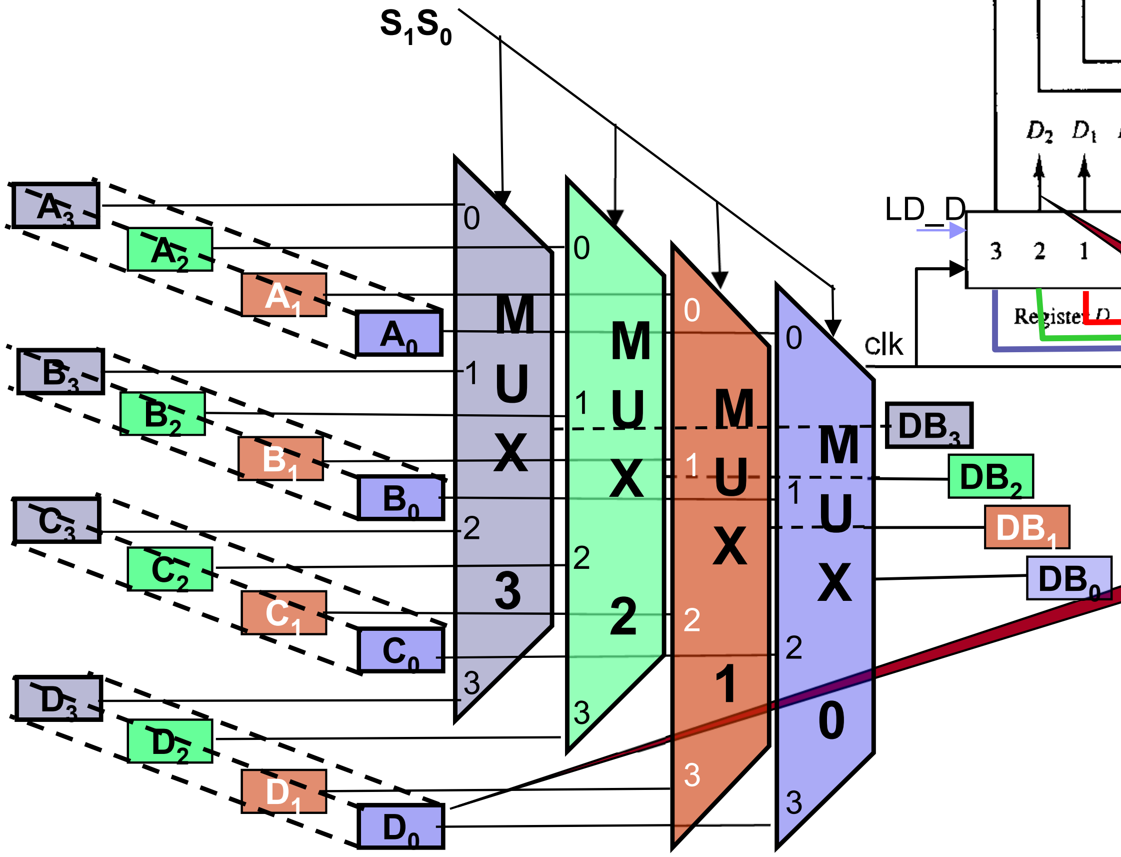
S1	S0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D



A bus system for four registers using multiplexers

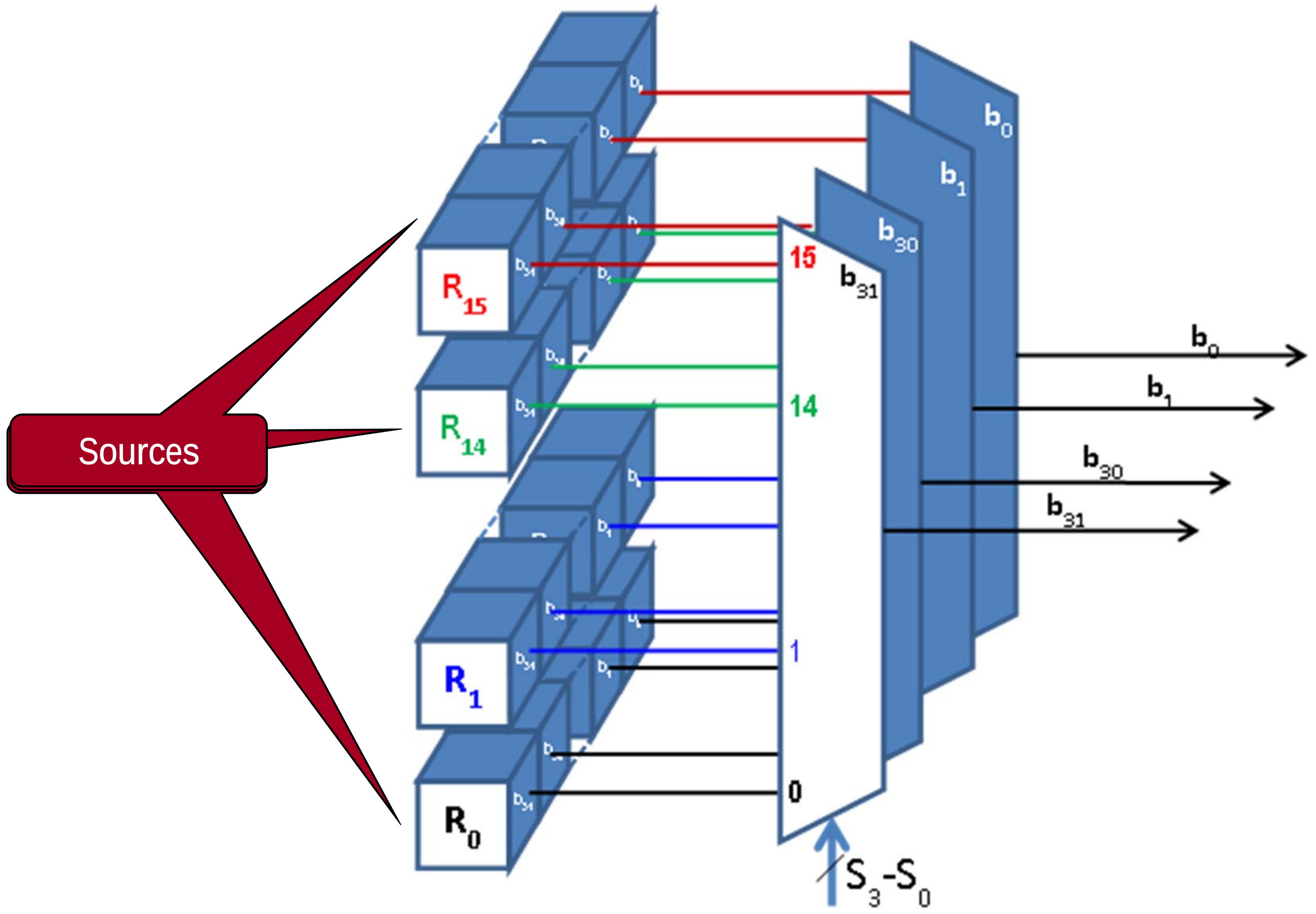
Bus with Multiplexers (“2D”/ ”3D” view)





S1	S0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

Common Bus System for 16 registers of 32 bits each



Three-State Buffers

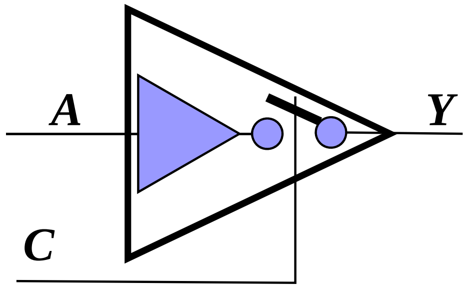
- A bus system may also be constructed with three-state gates instead of multiplexers.
- A three-state gate is a digital circuit that exhibits three states:
  - Two of these states correspond to signals equivalent to logic 1 and 0, as in a conventional gate.
  - The third state is high-impedance state.
- The **high-impedance** state makes the gate behaves like an open circuit (think of this state as if the output of the gate is “disconnected”).

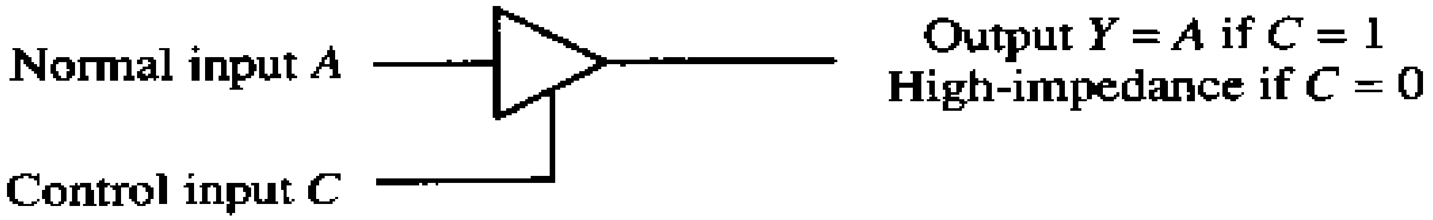
uOttawa

Three-State Buffers (cont.)

- Three-state gates can come in several forms, such as AND gates, NAND gates, etc.
- However, there is one that is most commonly used in the design of bus systems: the **three-state buffer** gate.

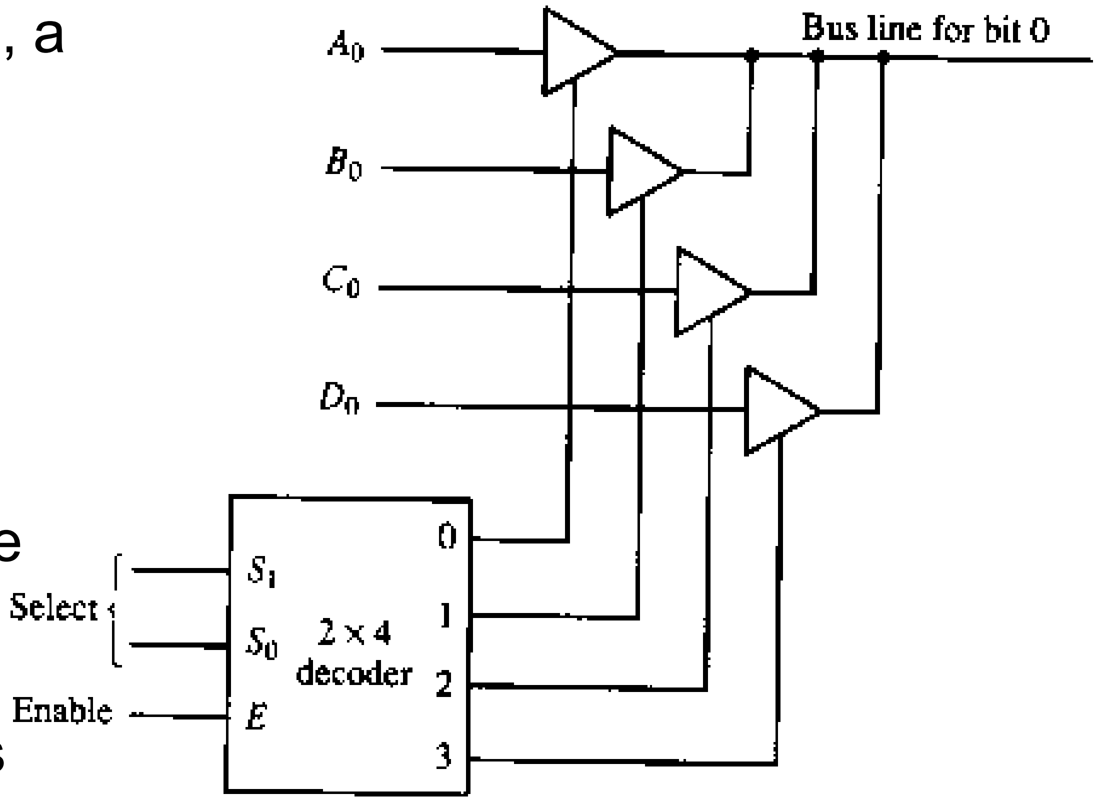
Switch	Control Input (C)	Data Input (A)	Output (Y)
ON	1	0	0
	1	1	1
OFF	0	x	High-impedance





A bus line with three-state buffers

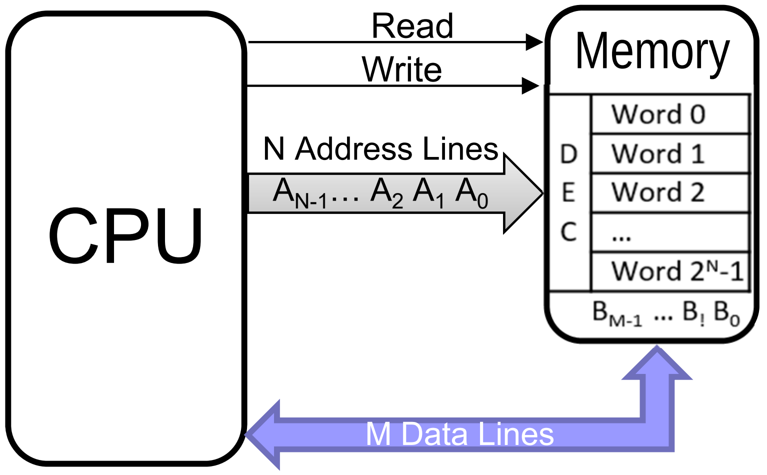
- To ensure that no more than one control input is active at any given time, a decoder is used.
- When the *Enable* input of the decoder is 0, all of its four outputs are reset to 0 ⇒ The bus line is in a high-impedance state (logically “disconnected”).
- When the *Enable* input is 1, the input of the selected buffer is transferred to the bus line. In this case, the above circuit acts like a 4 × 1 multiplexer.



- To construct a common bus system for four registers of *n* bits each using three-state buffers, *n* circuits like the one in figure are needed. However, only one decoder is required as common decoder for all of them.

Memory

- The memory stores the data while the CPU can read (extract or fetch) the data from computer memory for processing, or, it can write (store) the data onto the computer memory.
- The memory can be seen as a set of registers of the same length that store binary data, called "words". Each memory location (i.e., such a register) stores a word and has an address.



- The computer uses the address lines to locate the specific data word in the computer memory. Also, it uses the data lines to write a word into the memory location specified by the address lines, or to read a stored word from a memory location, also specified by the address lines, as shown in the figure.

- It can be noted from the figure above that the address lines run from CPU to memory, i.e., the address lines are unidirectional, and the data lines are bidirectional, i.e. CPU uses the data lines to both read from memory, or to write the data onto the memory.

Memory Transfer

- In the following, we will symbolize a memory word by the letter *M* .
- The *address register* holding the memory address of the word to be accessed will be symbolized by *AR*.
- The *data register* to hold word to be read from or written to the memory unit will be symbolized by *DR* .
- A memory **read** operation is stated as follows:

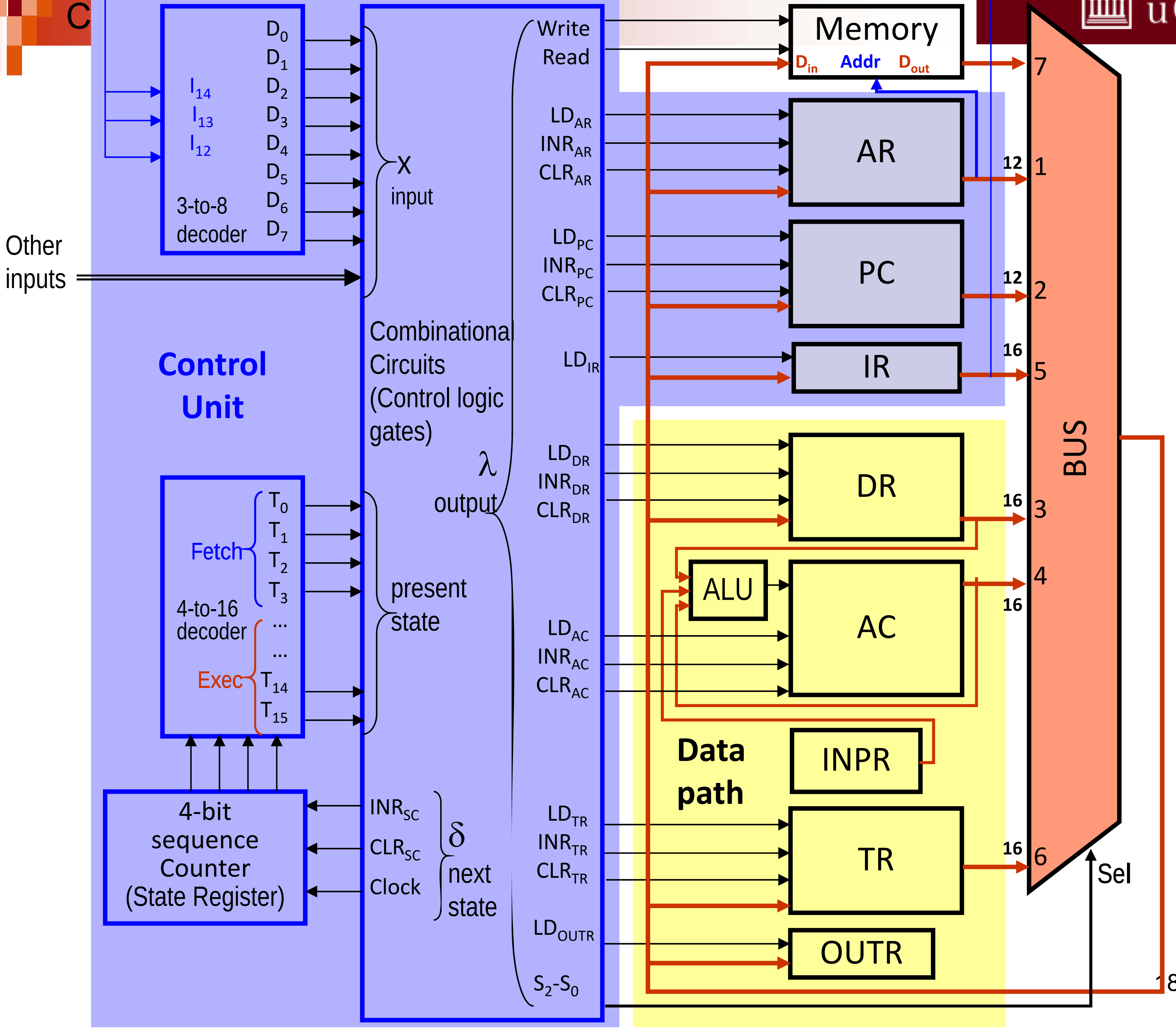
  Read: $DR \leftarrow M[AR]$, to mean

  if *the control function (memory control signal)* **Read** *is 1,* **then** *transfer the contents of the word* ***from*** *the memory location specified by address stored in register AR* ***into*** *register DR .*
- A memory **write** operation is stated as follows:

  Write: $M[AR] \leftarrow DR$, to mean

  if *the control function (memory control signal)* ***Write*** *is 1,* **then** *transfer the word stored in register DR to the memory word located at the address which is stored in register AR .*

C

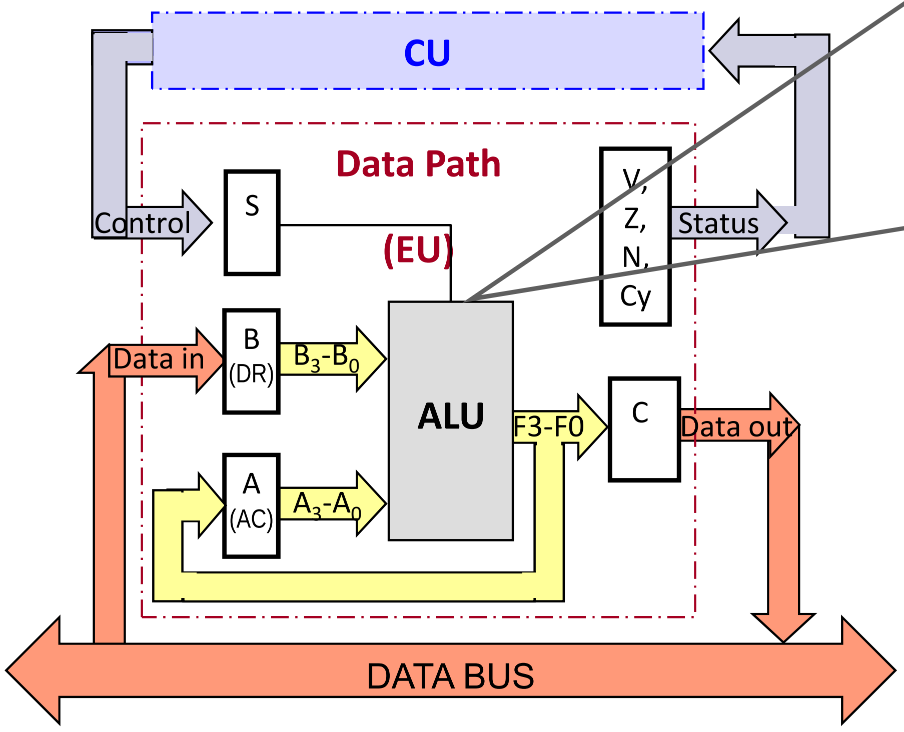


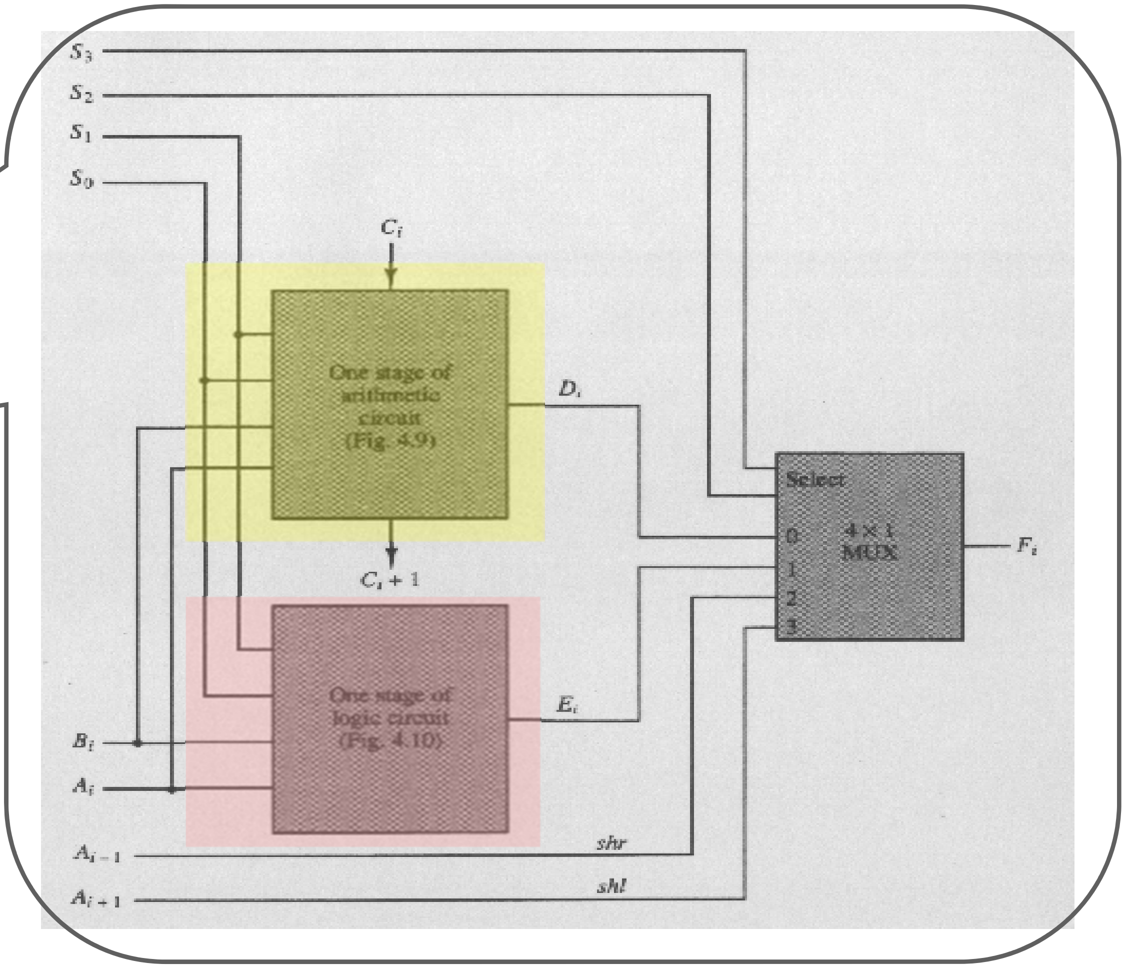
uOttawa

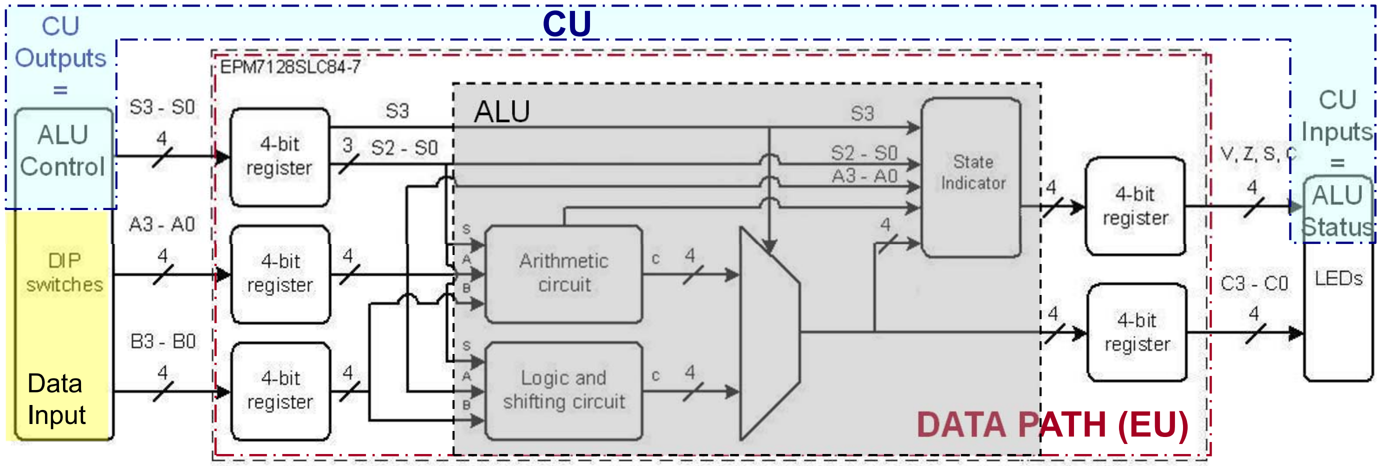
BASIC COMPUTER

uOttawa

ALU Block Diagram







Arithmetic microoperations

The microoperations most often encountered in digital computers are classified into four categories:

1. Register **transfer** microoperations: they transfer binary data from one register to another.
2. **Arithmetic** microoperations: they perform arithmetic microoperations, such as additions and subtractions for instance, on digital data stored in registers.
3. **Logic** microoperations: they perform bit manipulation operations on data stored in registers.
4. **Shift** microoperations: they perform shift operations on data stored in registers.

- We have just seen register **transfer** microoperations.
- In what follows, we will introduce each of the remaining three types of microoperations.

Add Micro-operation

- ... The micro-operation defined by the statement:

$$R3 \leftarrow R1 + R2$$

denotes an **add** micro-operation.

- It states that the content of register $R1$ is added to the content of register $R2$ and the sum is transferred to register $R3$.

Subtract Micro-operation

- Subtracting the content of register $R2$ from the content of register $R1$ and storing the result in register $R3$ can be symbolized by the following two possible notations:

$$R3 \leftarrow R1 - R2 \qquad \text{or} \qquad R3 \leftarrow R1 + \overline{R2} + 1$$

- $R2$ denotes the complement of the data stored in register $R2$.
- **Next** Table lists the most common basic microoperations

Common Arithmetic microoperations

#	Symbolic Designation (RTL)	Description
1.	$R3 \leftarrow R1 + R2$	Content of R1 plus R2 transferred to R3
2.	$R3 \leftarrow R1 - R2$	Content of R1 minus R2 transferred to R3
3.	$R2 \leftarrow R2$	Complement the content of R2 (1's complement)
4.	$R2 \leftarrow \overline{R2} + 1$	2's complement the content of R2 (negate)
5.	$R3 \leftarrow R1 + \overline{R2} + 1$	R 1 plus the 2's complement of R2 (subtraction)
6.	$R1 \leftarrow R1 + 1$	Increment the content of R1 by 1
7.	$R1 \leftarrow R1 - 1$	Decrement the content of R1 by 1

Binary Adder

- To implement the add micro-operation in hardware, we need at least two registers to hold the two numbers to be added, possibly a third register to store the result of the operation, and a combinational circuit to perform the arithmetic addition.
- A combinational circuit that adds two bits and a carry bit is called a full-adder (FA) {refer to Chapter 1}.
- The combinational circuit that performs the arithmetic sum of two n-bit numbers is called an *n*-bit binary adder.
- Typically, an *n*-bit binary adder is constructed with *n* full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder.

ADDING BINARY NUMBERS

- **Adding two bits:**

0+	0+	1+	1+
0	1	0	1
0	1	1	1 0

Carry (over) — Sum

The binary number **10** is equivalent to the decimal **2**

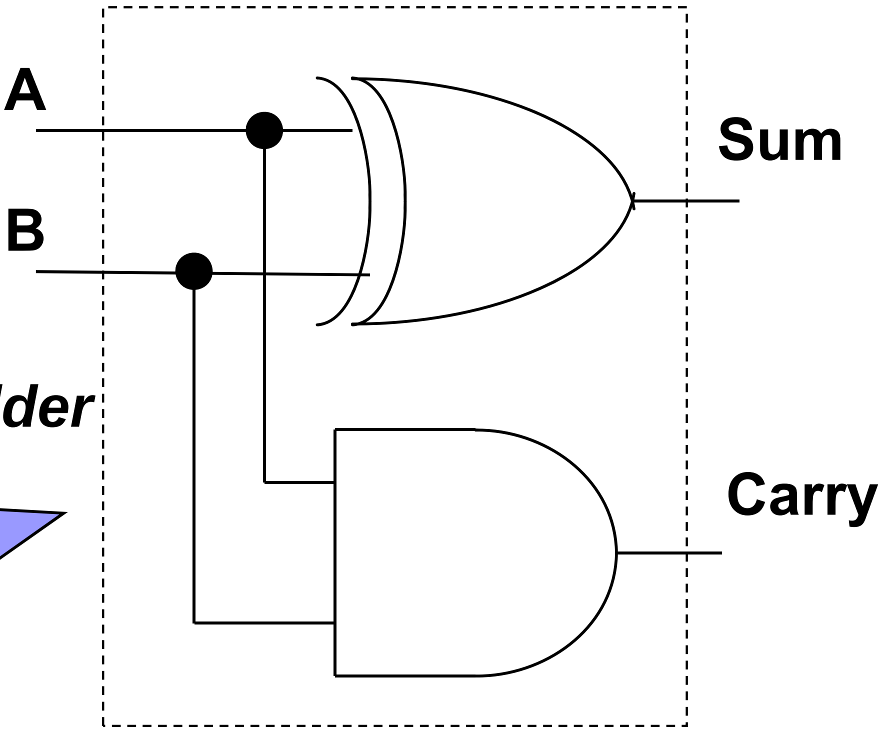
Truth table

Inputs		Outputs	
A	**B**	**Carry**	**Sum**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Sum = A ⊕ B

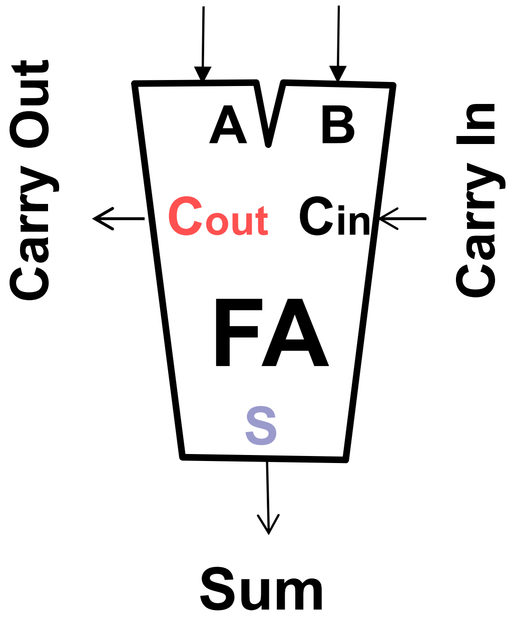
Carry = A · B

Half-Adder circuit



Full Adder

Bits of the same rank of the two numbers

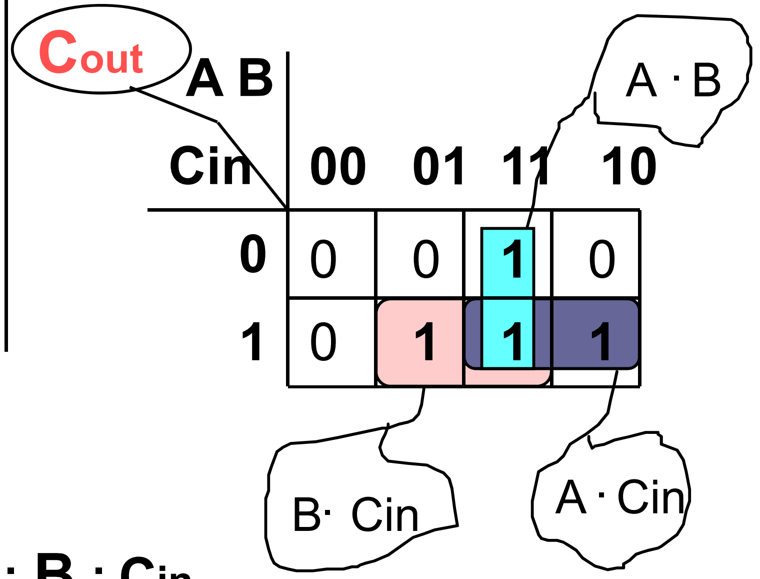


Inputs			Outputs	
A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S

Cin \ AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Cout



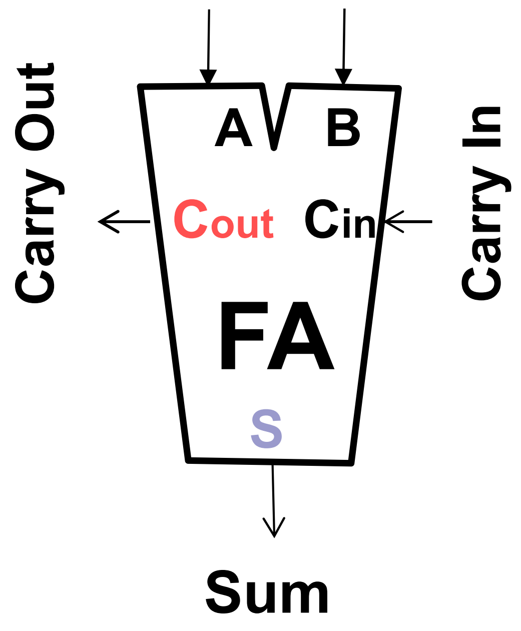
Cin \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$S = \overline{A} \cdot \overline{B} \cdot Cin + \overline{A} \cdot B \cdot \overline{Cin} + A \cdot \overline{B} \cdot \overline{Cin} + A \cdot B \cdot Cin$$

$$Cout = A \cdot B + B \cdot Cin + A \cdot Cin$$

Full Adder (FA)

employing Half Adders (HA)

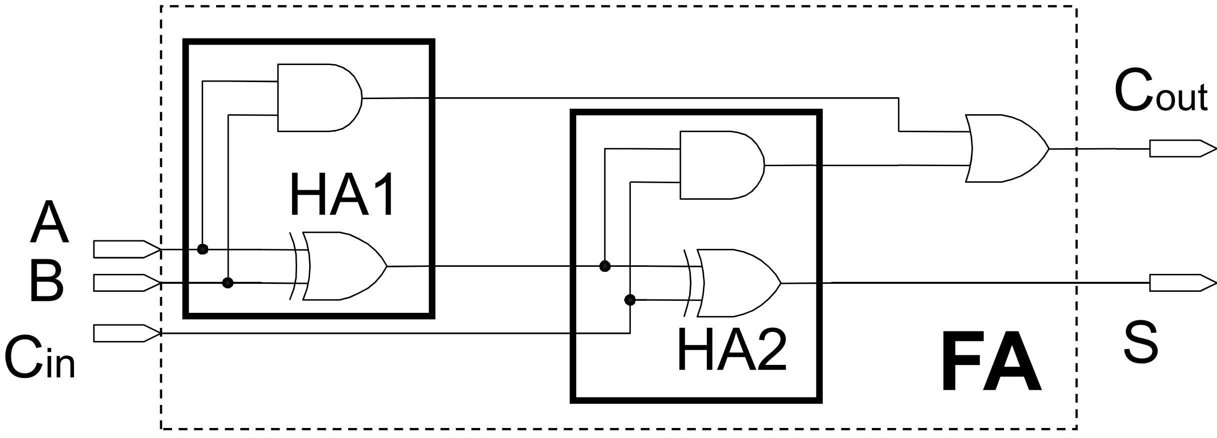


A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

•Equations are modified as follows:

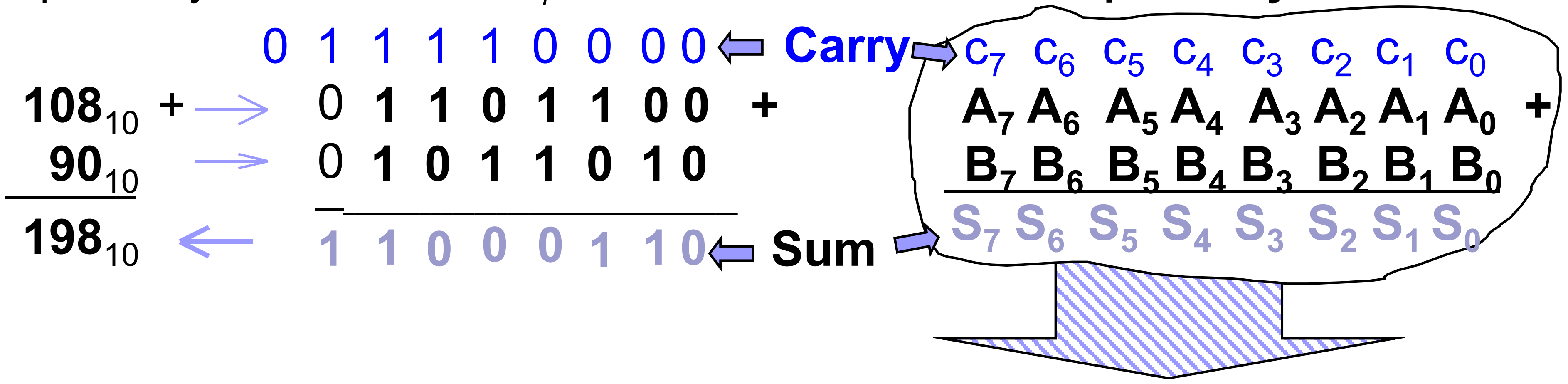
$$C_{out} = ((A \oplus B) \,.\, C_{in}) + (A \,.\, B)$$

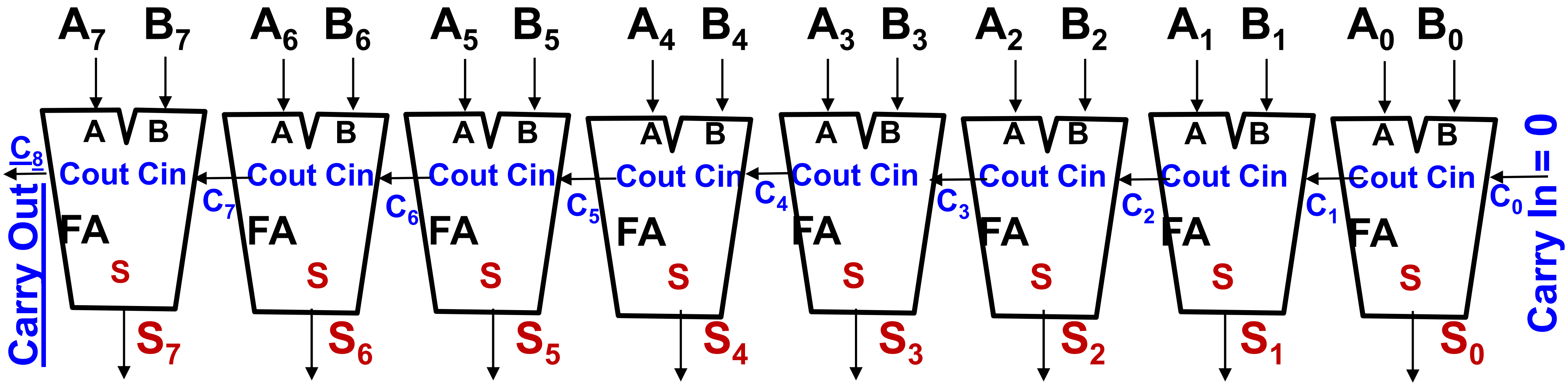
$$S = A \oplus B \oplus C_{in}$$



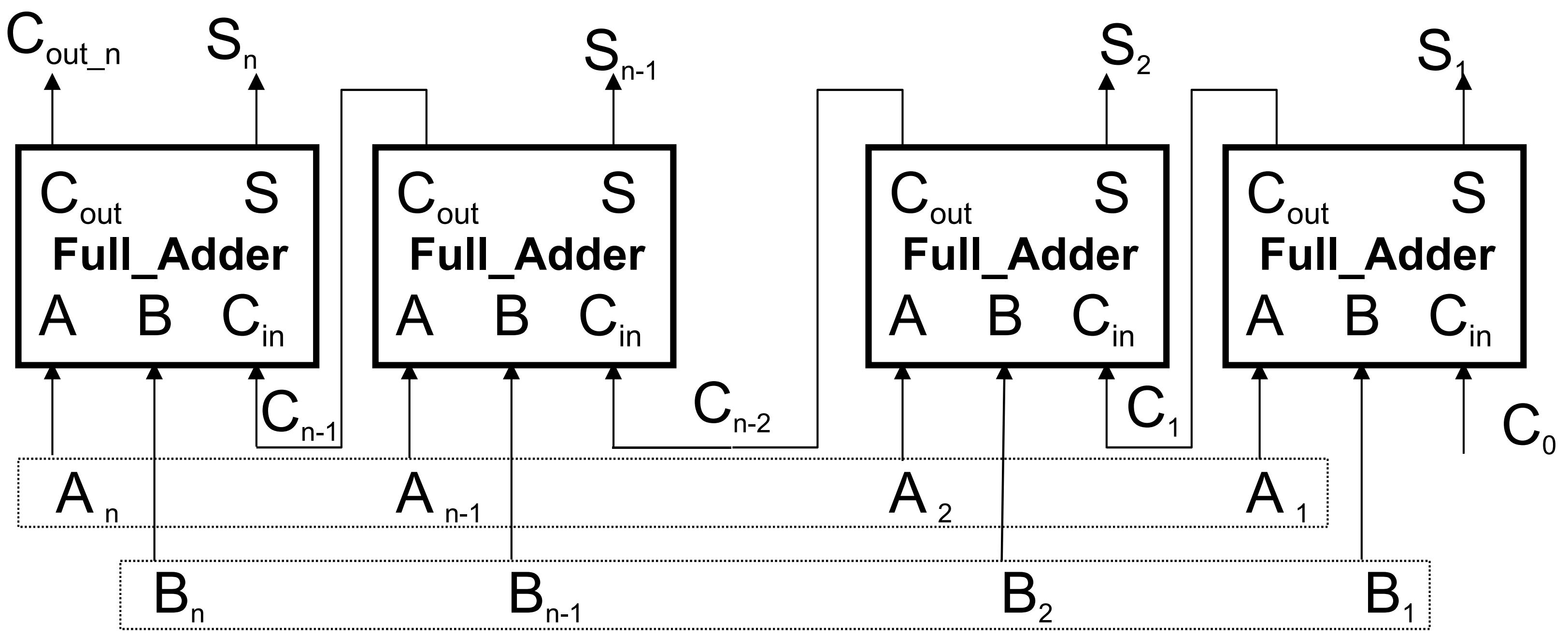
Adding multi-bit numbers:

An adder is a combinational circuit made of full adders FA – see the block diagram below
In this example the added numbers are $A_7 \ldots A_2A_1A_0$ and $B_7 \ldots B_2B_1B_0$, which are supposed to be stored in two 8-bit registers A and B.
The sum is represented by $S_7 \ldots S_2S_1S_0$, & is to be fed into the input of another register.
The input carry to the *i*-th bit is c_i, with i = 0, 1, 2, …7; the **output carry** of the bit 7 is C_8.

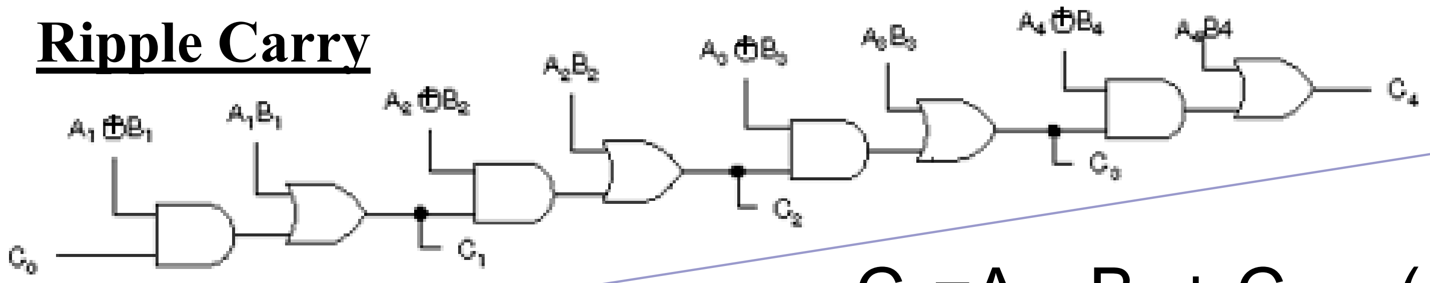

0 1 1 1 1 0 0 0 0
Carry
108_{10} +
90_{10}
198_{10}
0 1 1 0 1 1 0 0 +
0 1 0 1 1 0 1 0
1 1 0 0 0 1 1 0
Sum
c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0
A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 +
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
S_7 S_6 S_5 S_4 S_3 S_2 S_1 S_0


A_7 B_7 A_6 B_6 A_5 B_5 A_4 B_4 A_3 B_3 A_2 B_2 A_1 B_1 A_0 B_0
A B
Cout Cin
FA
S
C_8
Carry Out
C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0
Carry In = 0
S_7 S_6 S_5 S_4 S_3 S_2 S_1 S_0

Parallel Binary Adder



Ripple Carry



$$C_n = A_n \cdot B_n + C_{n-1} \cdot (A_n \oplus B_n)$$

Look-ahead carry generator

$$s_i = \bar{a}_i\bar{b}_ic_i + \bar{a}_ib_i\bar{c}_i + a_i\bar{b}_i\bar{c}_i + a_ib_ic_i$$
$$c_{i+1} = b_ic_i + a_ic_i + a_ib_i$$
$$c_{i+1} = a_ib_i + (a_i + b_i)c_i$$
$$c_{i+1} = G_i + P_ic_i$$

$G_i = a_i\, b_i$ => G_i = generate,

$P_i = a_i + b_i$ => P_i = propagate,

The carry c_1 of stage 0 is

$c_1 = G_0 + P_0C_0$

Since $c_0 = 0$ then $c_1 = G_0$

The carry c_2 out of stage 1 is

$c_2 = G_1 + P_1\, c_1$

Since $c_1 = G_0$ then $c_2 = G_1 + P_1G_0$

The carry c_3 out of stage 2 is

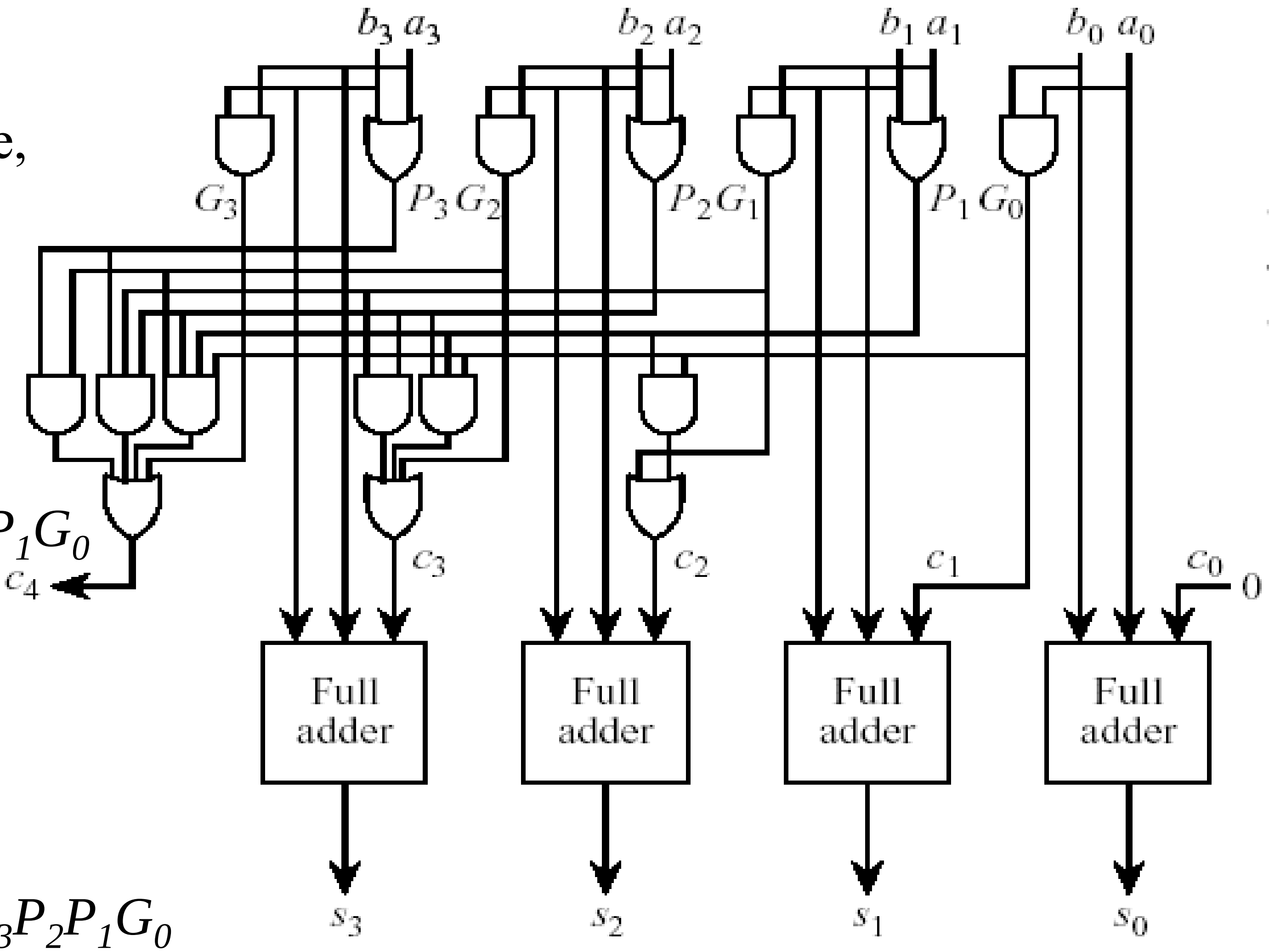
$c_3 = G_2 + P_2c_2$

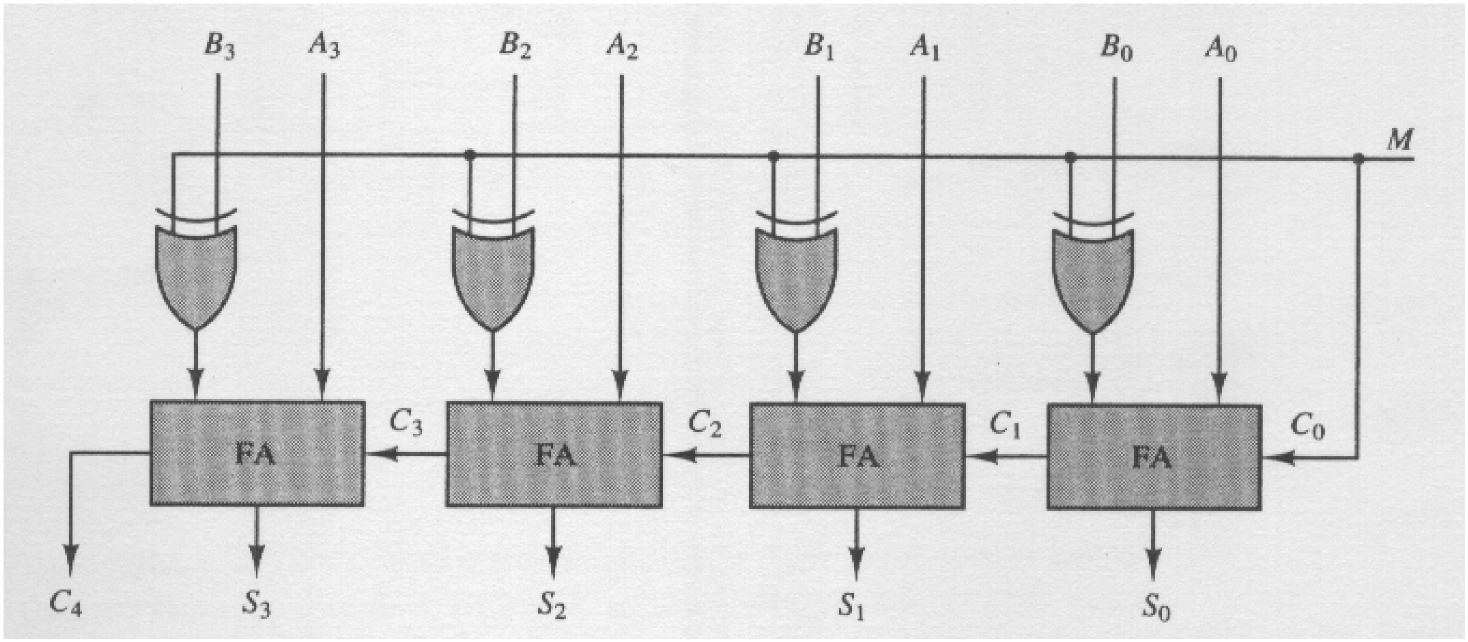
$c_3 = G_2 + P_2G_1 + P_2P_1G_0$

The carry c_4 out of stage 3 is

$c_4 = G_3 + P_3c_3$

$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$



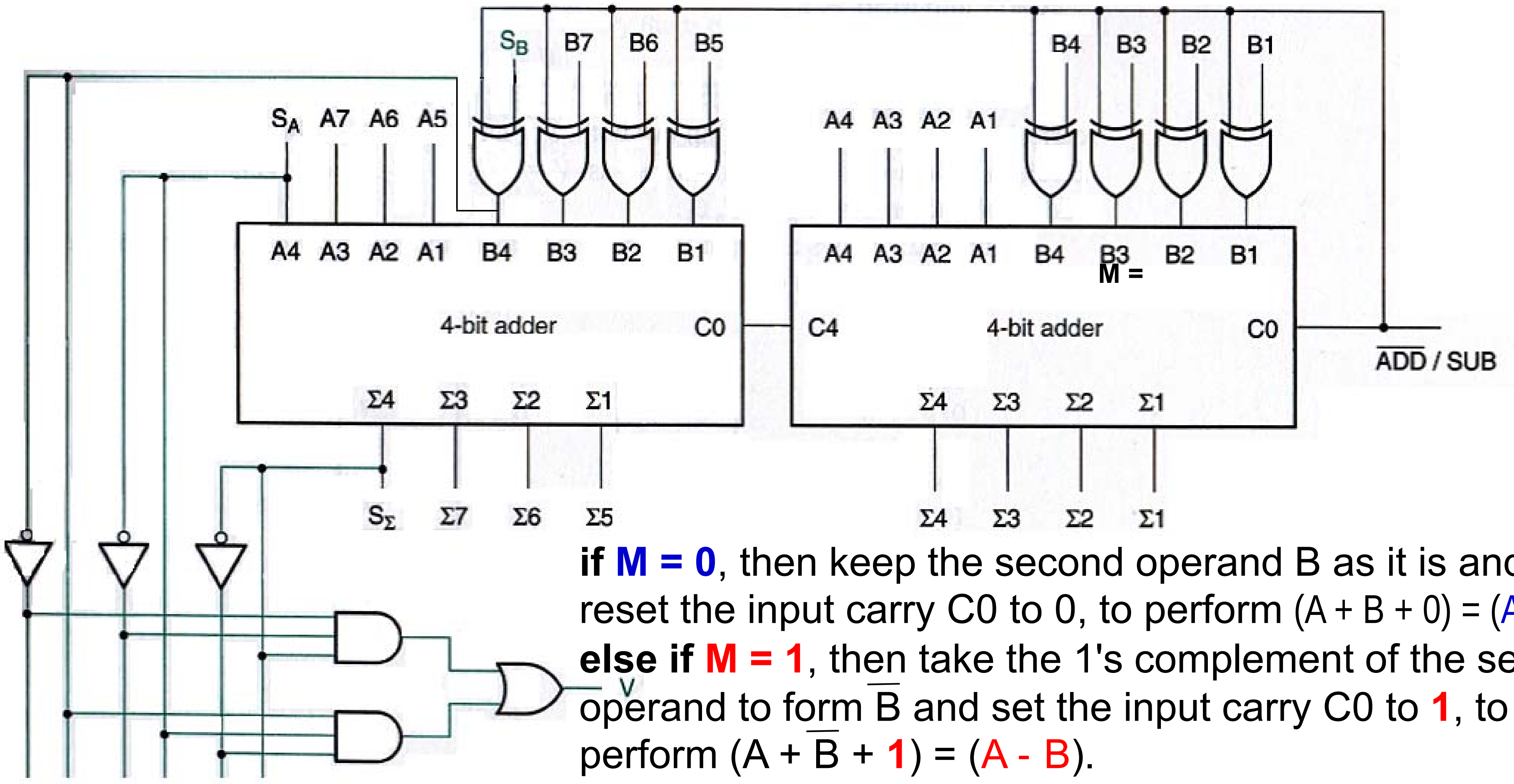


- For unsigned numbers,
  - When $M = 0$ (**addition** micro-operation), $S_{n-1} \ldots S_1 S_0$ gives the result of ($A + B$). An overflow occurs when $C_n = 1$.
  - When $M = 1$ (**subtraction** micro-operation) and if $A \geq B$, then $S_{n-1} \ldots S_1 S_0$ gives the result of ($A - B$) . An overflow cannot possibly exist in this case.
  - When $M = 1$ (subtraction micro-operation) and if $A < B$, then $S_{n-1} \ldots S_1 S_0$ gives the result of the 2's complement of ($B - A$) . An overflow cannot possibly exist in this case.

- For signed numbers,
  - When $M = 0$ (addition micro-operation), $S_{n-1} \ldots S_1 S_0$ gives the result of ($A + B$).
    An overflow occurs when $C_n \oplus C_{n-1} = 1$.
  - When $M = 1$ (subtraction micro-operation), $S_{n-1} \ldots S_1 S_0$ gives the result of ($A - B$).
    An overflow occurs when $C_n \oplus C_{n-1} = 1$.

8 – bit Binary Adder / Subtractor with Overflow Detector

The operation (A-B) is accomplished by performing the operation (A + $\overline{B}$ + 1), which is in essence an addition operation => the addition and subtraction operations can be combined into one common circuit that uses an input control bit, say M, to indicate the type of the desired operation.



if M = 0, then keep the second operand B as it is and reset the input carry C0 to 0, to perform (A + B + 0) = (A + B)

else if M = 1, then take the 1's complement of the second operand to form $\overline{B}$ and set the input carry C0 to 1, to perform (A + $\overline{B}$ + 1) = (A - B).

Binary Incrementer

•One way of implementing a binary incrementer is using a binary counter.

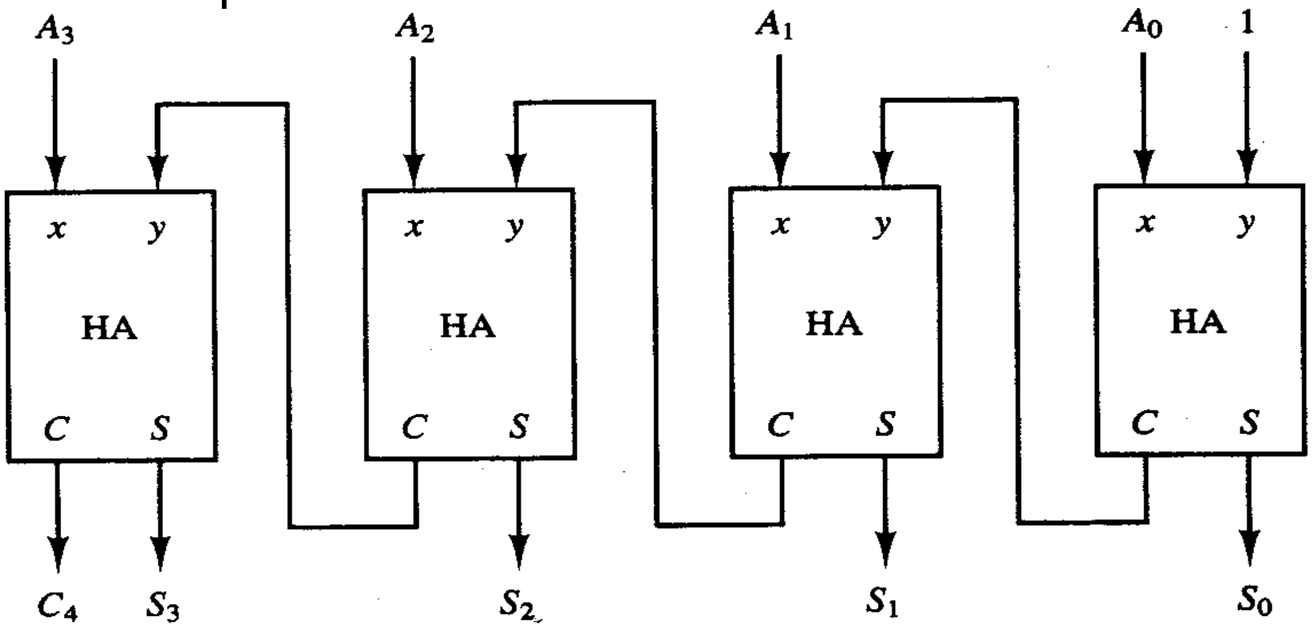
•Another way of implementing a binary incrementer is using half-adders (HA) connected in cascade.

•A combinational circuit of n-bit binary incrementer that operates on a binary number $A = A_{n-1} \ldots A_0$ is composed of:

- n half-adders
- n + 1 input bits: $A_0, \ldots, A_{n-1}$, and 1
- n + 1 output bits: the sum $S_{n-1} \ldots, S_0$, and the output carry Cn

•The two input bits of the first half-adder (HA_0) are A0 and 1.

•The two input bits of each other half-adder (HA_i), where i = 1; : : : ; n - 1, are A_i and the output carry C_{i-1} from the previous half-adder.

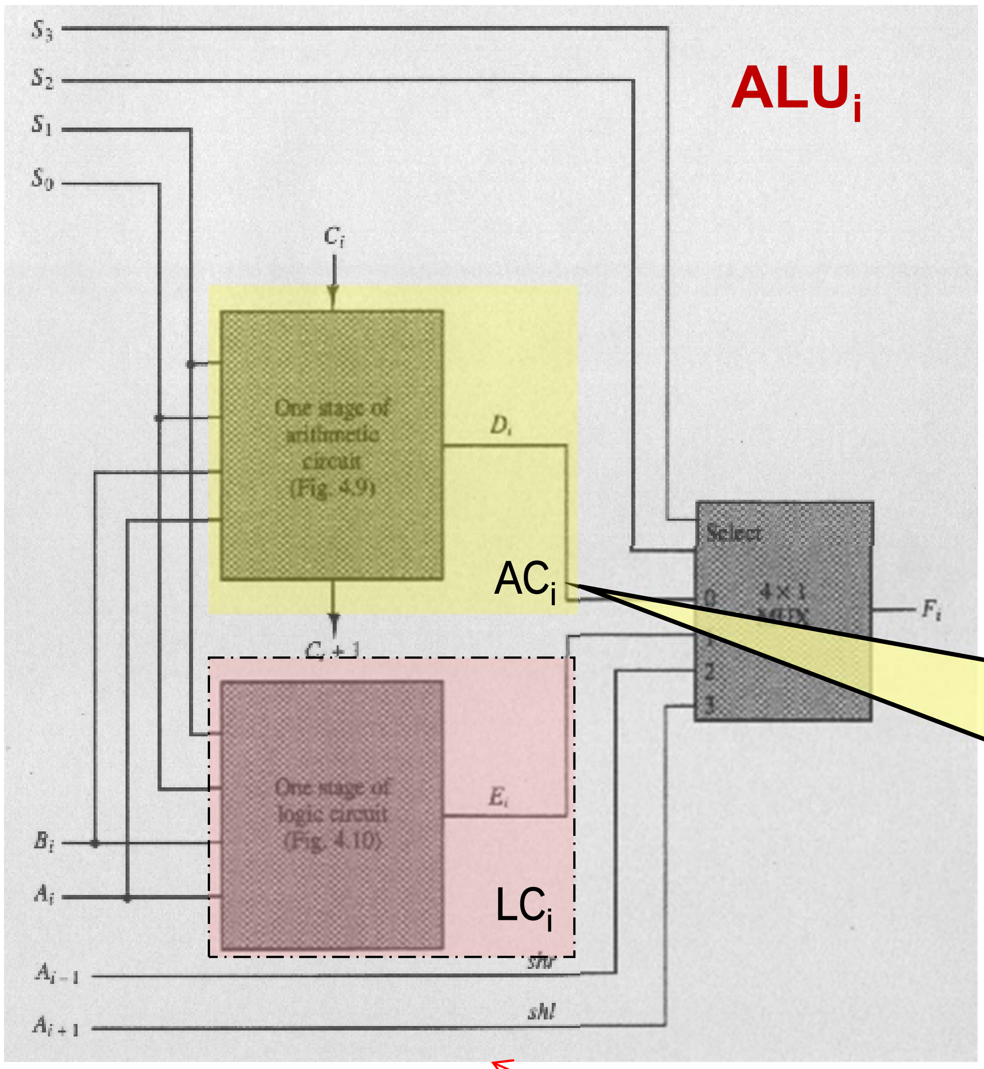


Arithmetic Circuit (AC)

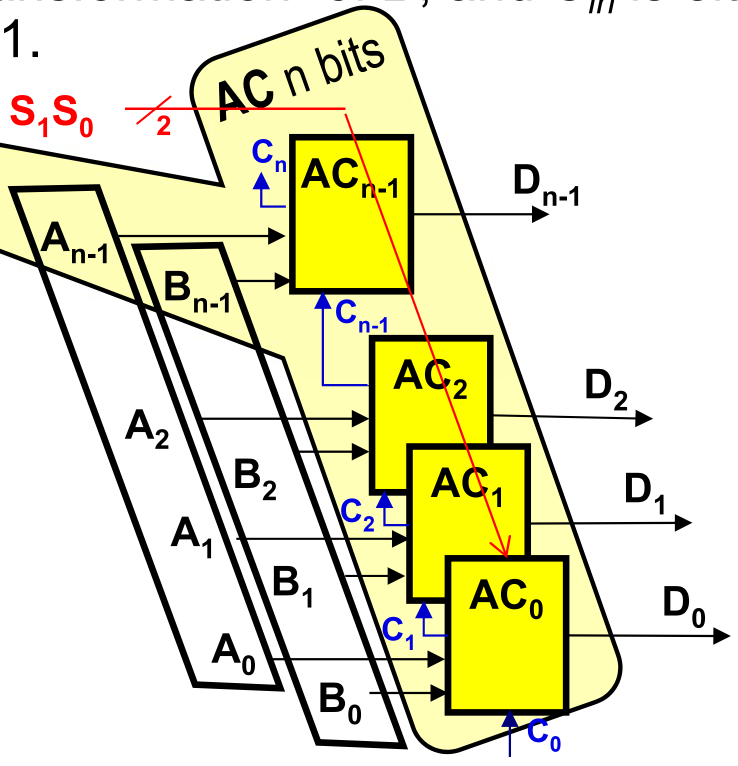
- Different microoperations can be implemented in one common combinational circuit called an **arithmetic circuit (AC)**, or an **arithmetic unit**.
- The idea behind designing an *n*-bit arithmetic unit (AC) that operates on two *n*-bit numbers *A* and *B* is to write the result *D* of each micro-operation in the form of

$$D = A + Y + C_{in} ,$$

where *Y* is the result of some sort of a "transformation" of *B*, and C_{in} is either 0 or 1.

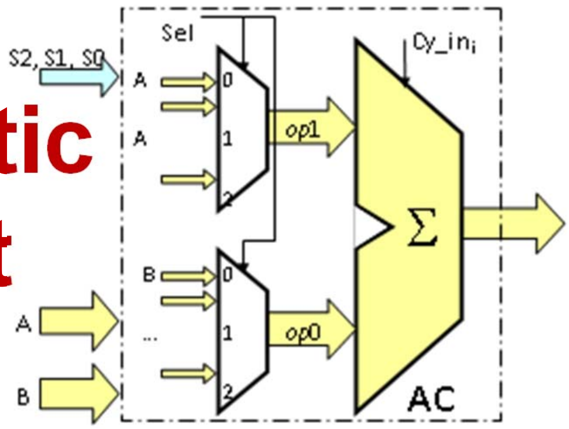


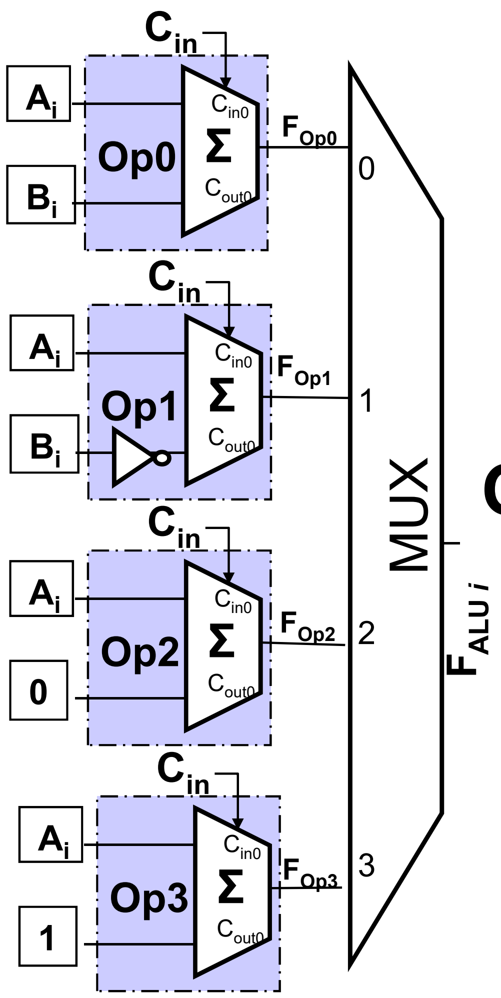
ALU Block Diagram (for 1 bit, *i* only!)



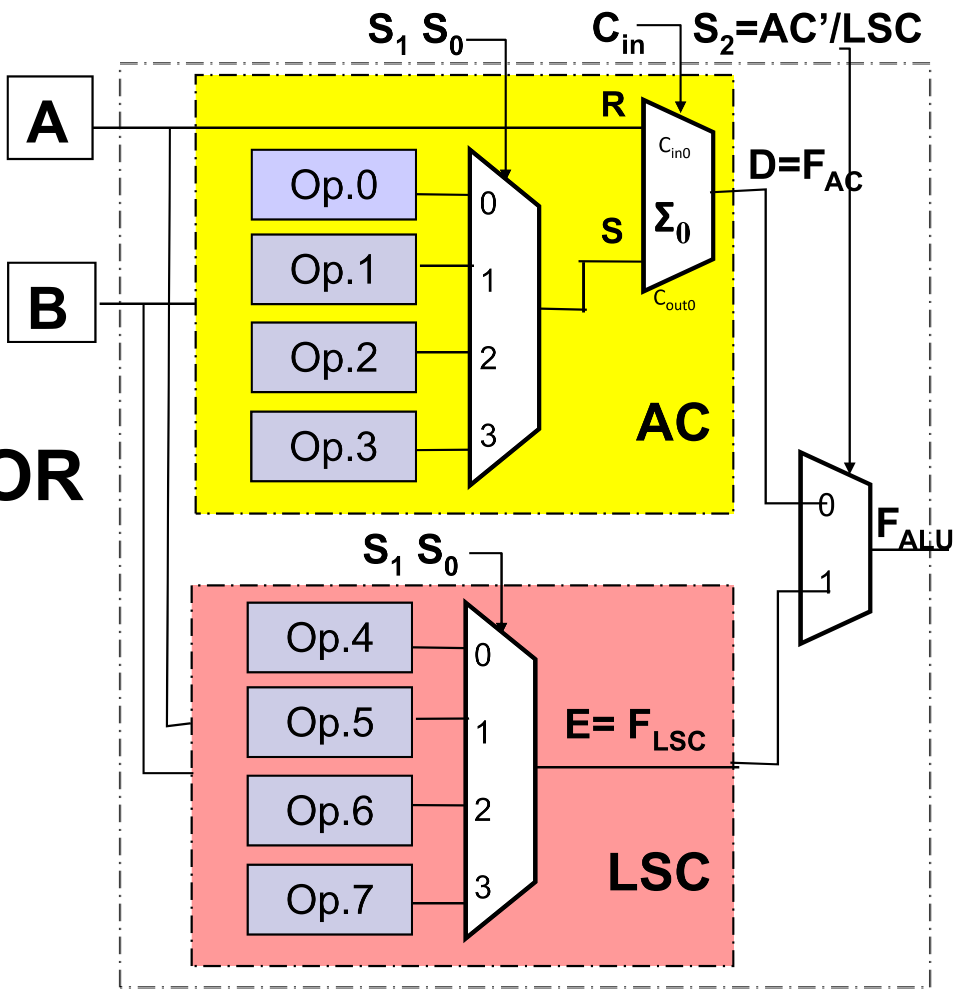
	Select Bits	Output	Micro-operation
	S_1 S_0 C_{in}	$D = R+S+C_{in}$	$= R_{n-1}...R_2R_1R_0 + S_{n-1}...S_2S_1S_0$
Op.0	0 0 0	D = A+B	**Add** = $A_{n-1}...A_2A_1A_0 + B_{n-1}...B_2B_1B_0$
	0 0 1	D = A+B+1	Add with carry = $A_{n-1}...A_2A_1A_0 + B_{n-1}...B_2B_1B_0$+00000001
Op.1	0 1 0	D = A−B−1	**Subtract with borrow** $A_{n-1}...A_2A_1A_0 + B'_{n-1}...B'_2B'_1B'_0$
	0 1 1	D = A−B	**Subtract** =$A_{n-1}...A_2A_1A_0$ + ($B'_{n-1}...B'_2B'_1B'_0$ + 0000 0001)
Op.2	1 0 0	D = A	**Transfer A** = $A_{n-1}...A_2A_1A_0$ + 0000 0000
	1 0 1	D = A+1	**Increment A** = $A_{n-1}...A_2A_1A_0$+ 0000 0000 + 0000 0001
Op.3	1 1 0	D = A−1	**Decrement A** = A + 2's Compl(1) = A + 1111 1111
	1 1 1	D = A	**Transfer A** = A + 1111 1111 + 0000 0001

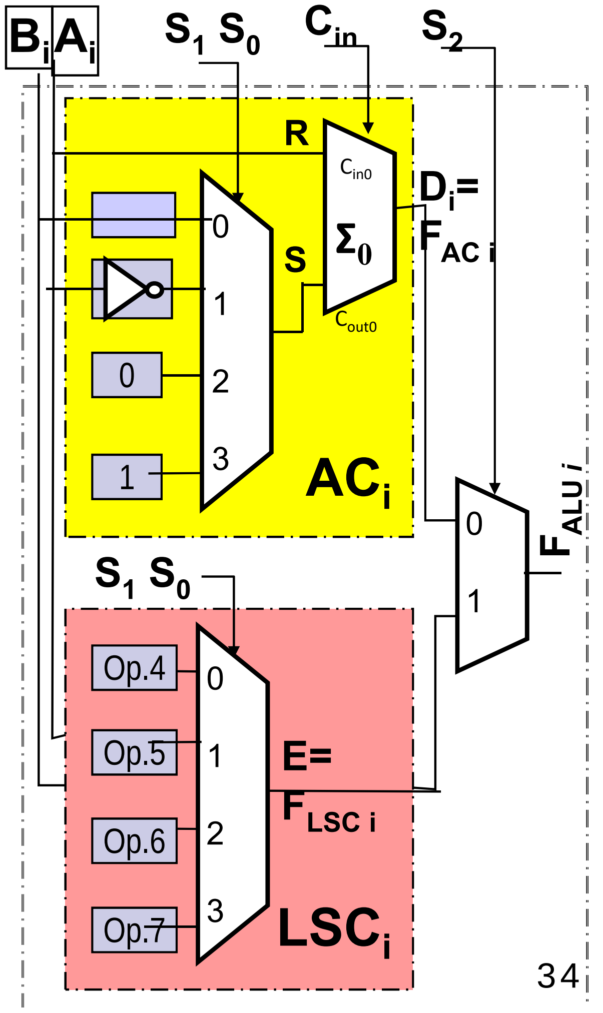
Arithmetic Circuit





OR



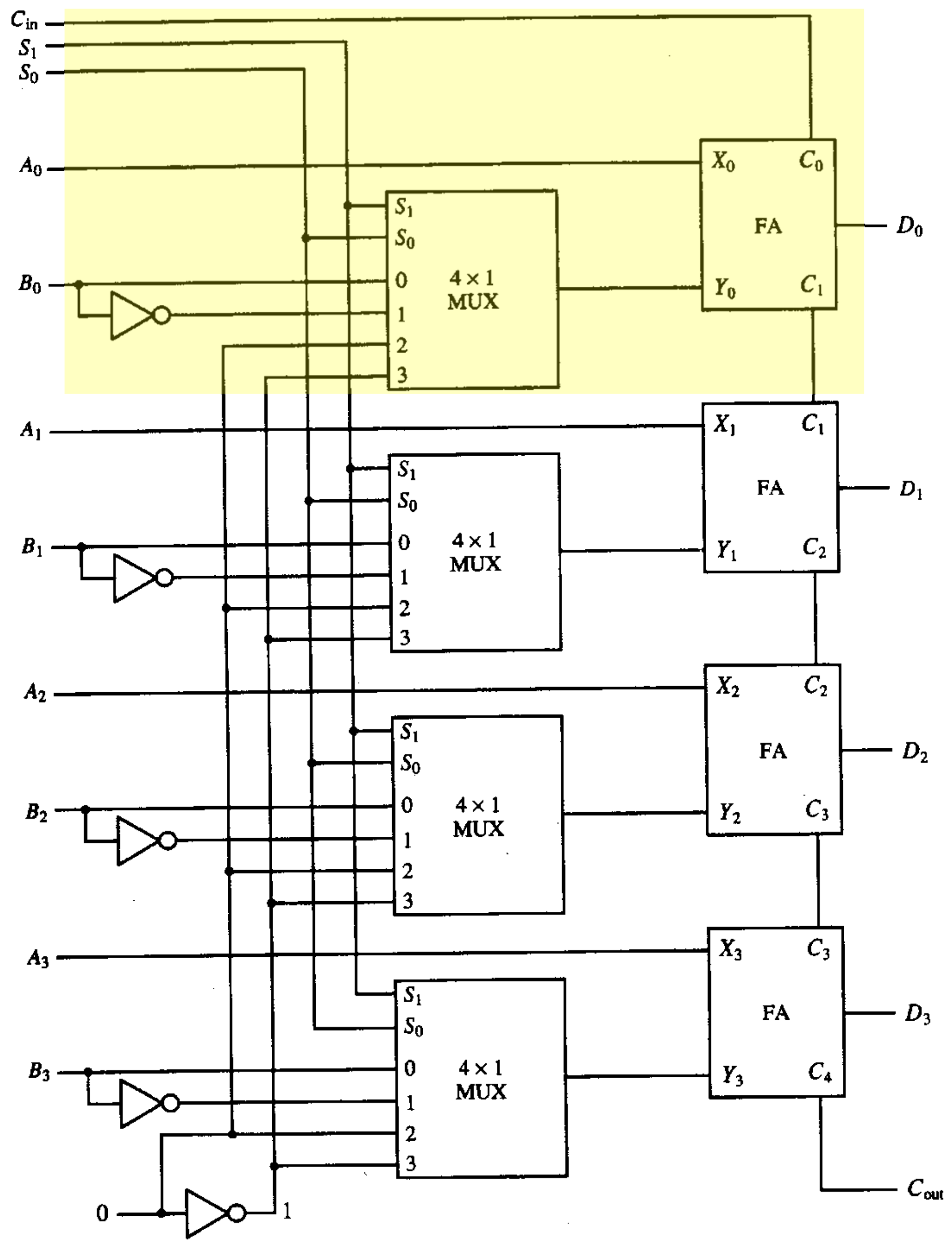


Arithmetic Circuit

(for 4 bit integers)

- An n-bit arithmetic unit performing 2^k arithmetic microoperations is composed of:
  - Two n-bit data input lines, A and Y
  - n full-adders, with each full-adder operating on one bit of A, one bit of Y , and the output carry bit from the previous full-adder
  - n multiplexers of type $2^k \times 1$, with each multiplexer generating one bit of Y
  - k selection bits $S_{n-1} \ldots S_0$ that are common for all the multiplexers
- Sometimes, and depending on the microoperations to be performed by the arithmetic unit, it is convenient to assign C_{in} to one of the selection bits.

Select Bits			Output	
S_1	S_0	C_{in}	$D = A + Y + C_{in}$	Micro-operation
0	0	0	$D = A + B$	Add
0	0	1	$D = A + B + 1$	Add with carry
0	1	0	$D = A + \overline{B}$	Subtract with borrow ($D =$
0	1	1	$D = A + \overline{B} + 1$	Subtract ($D = A - B$)
1	0	0	$D = A$	Transfer A
1	0	1	$D = A + 1$	Increment A
1	1	0	$D = A - 1$	Decrement A
1	1	1	$D = A$	Transfer A



Logic microoperations

- Logic microoperations perform ordinary boolean operations on data stored in two registers.
- An example of a logic microoperations is

$$R1 \leftarrow R1 \oplus R2,$$

which performs an XOR operation on the contents of registers *R* 1 and *R* 2 and stores the result back into *R* 1.

- To distinguish an arithmetic addition from a logic OR micro-operation
  - a logic OR micro-operation will be symbolized by a "$\vee$".
  - Similarly, a logic AND micro-operation will be symbolized by a "$\wedge$".
- When a "+" symbol appears in a control function, it will still denote an OR operation (as it cannot be confused with an addition).

Example 38. The statement

$P + Q: \; R1 \leftarrow R2 + R3, \; R4 \leftarrow R5 \vee R6$ means:

if (P = 1 or Q = 1), then perform the two microoperations and load their results into their respective registers at the same clock pulse:

- add *R*2 and *R*3 and store the result into *R*1, and
- perform a logic OR operation on *R*5 and *R*6 and store the result in *R*4.

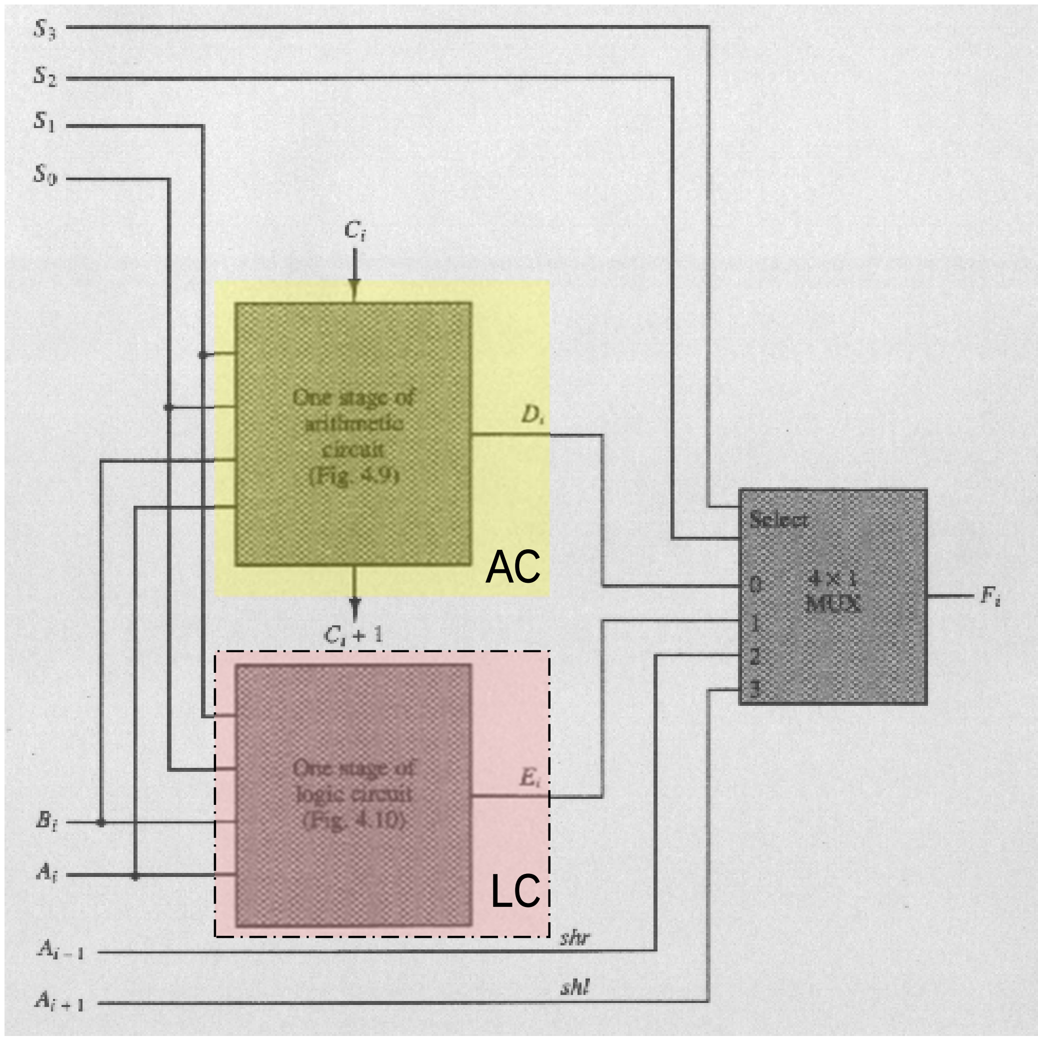
List of all 16 logic microoperations

A_i	B_i	f_0	f	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	1	1	0	1	1	0	1	1	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

that can be performed on pairs of bits (A_i and B_i) of 2 registers (A & B).

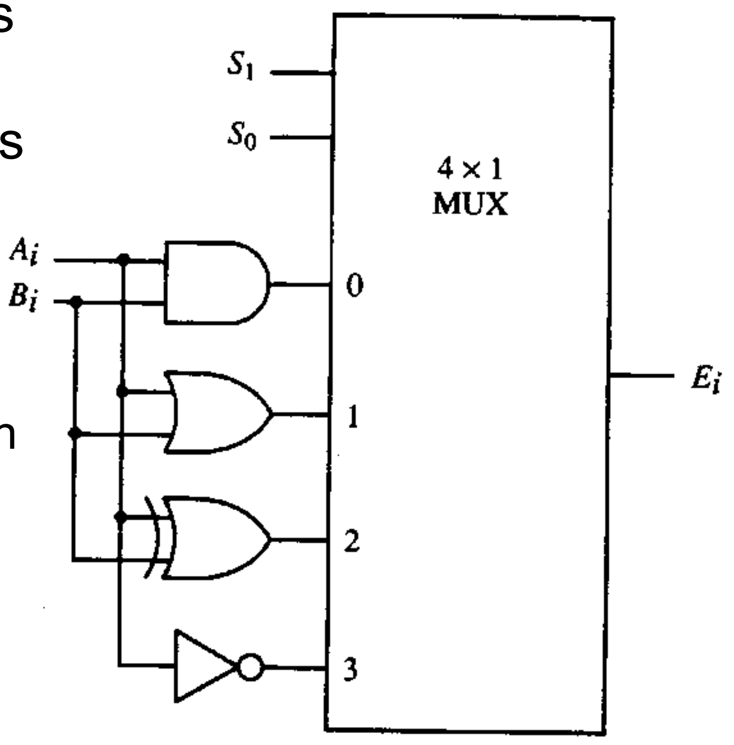
The block diagram implements 4 of them.

Boolean function	Microoperation	Name
$f_0 = 0$	f←0	Clear
$f_1 = xy$	f←A ∧ B	AND
$f_2 = xy'$	f←A ∧ B’	
$f_3 = x$	f←A	Transfer A
$f_4 = x'y$	f←A’ ∧ B	
$f_5 = y$	f ←B	Transfer B
f_6 = xE9y	f ←A ⊕ B	Exclusive-OR
$f_7 = x+y$	f ← A V B	OR
$f_8 = (x + y)'$	f ← (A V B)’	NOR
f_9 = (x EB y)'	f ←(A ⊕ B)’	Exclusive-NOR
$f_{10} = y'$	f←B’	Complement B
$f_{11} = x+y'$	f←A V B’	
$f_{12} = x'$	f←A’	Complement A
$f_{13} = x' + y$	f←A’ V B	
$f_{14} = (xy)'$	f←(A ∧ B)’	NAND
$f_{15} = 1$	f←all 1's	Set to all 1's



Logic Unit (LC)

- The circuit performing a set of logic microoperations is called a **logic unit**, or a **logic circuit (LC)**.
- An *n*-bit logic unit that can perform 2*k* logic operations on two *n*-bit registers *A* and *B* is composed of:
  - Two *n*-bit input lines for *A* and *B*
  - *n* multiplexers of type 2*k* × 1, with each multiplexer operating on one bit from each register
  - The particular logic operation to be performed is chosen by the *k* selection bits that are common to all *n* multiplexers.



Logic diagram for the *i*-th bit only!

Example 39 (2-Bit Logic Unit).
Design a 2-bit logic unit to perform the following four microoperations on two 2-bit registers $A = A_1\ A_0$ and $B = B_1\ B_0$.

- 2-bit logic unit ⇒ two multiplexers, one for each bit.
- Four microoperations ⇒ Two selection bits S_1 and S_0 that are common for both multiplexers.
- First, build the circuit that operates on A_0 and B_0 (one-stage circuit). Then build as many replicates of this circuit as needed for the rest of the bits. In this case, one more replicate is needed for *A*1 and *B*1 .
- Stack all these circuits together to form an *n*-bit logic unit.

Select S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	E = A’	Complement

uOttawa

Applications

- Logic microoperations are mainly used to manipulate individual bits or portions of a word stored in a register.
- Examples of such bit manipulation operations are: resetting selected bits to 0 or setting them to 1.
- In the following, we will see some examples of how to manipulate a set of bits in a certain register (register *A*).
  - Selective-Set
  - Selective-Complement
  - Selective-Clear
  - Mask
  - Insert
  - Clear

Selective-Set

- The **selective-set** micro-operation sets certain bits of *A* to 1 and keeps the remaining bits unchanged.
- The position of the bits in *A* to be affected by this transformation are specified by another register *B* .
- The bits in *A* where there are corresponding 1's in *B* are the bits that are going to be set to 1.

1	0	1	0	*A* before
1	1	0	0	*B*
1	1	1	0	*A* after

- Such a transformation can be performed using a logic OR (∨) micro-operation, and it can be symbolized by the following register-transfer language (RTL) statement:

$$A \leftarrow A \vee B \,.$$

Selective-Complement

- The **selective-complement** micro-operation complements bits in *A* where there are corresponding 1's in *B* . It keeps the remaining bits of *A* unchanged.

1	0	1	0	*A* before
1	1	0	0	*B*
0	1	1	0	*A* after

- Such a transformation can be performed using a logic XOR ($\oplus$) micro-operation, and it can be symbolized by the following RTL statement:

$$A \leftarrow A \oplus B \,.$$

Selective-Clear

- The **selective-clear** micro-operation resets to 0 bits in *A* where there are corresponding 1's in *B* . It keeps the remaining bits of *A* unchanged.

1	0	1	0	*A* before
1	1	0	0	*B*
0	0	1	0	*A* after

- Such a transformation can be performed using a logic AND ($\wedge$) micro-operation, and it can be symbolized by the following RTL statement:

$$A \leftarrow A \wedge \overline{B} \,.$$

Mask

- The **mask** micro-operation resets to 0 bits in *A* where there are corresponding 0's in *B* . It keeps the remaining bits of *A* unchanged.

1	0	1	0	*A* before
1	1	0	0	*B* (logic operand)
1	0	0	0	*A* after masking

- Such a transformation can be performed using a logic AND (∧) micro-operation, and it can be symbolized by the following RTL statement: $A \leftarrow A \wedge B$.

Clear

- The **clear** micro-operation compares the words in *A* and *B* and produces an all 0's result if the two numbers are equal.
- This can be achieved by an XOR logic operation.

1	**0**	**1**	**0**	**A before**
1	0	1	0	**B**
0	**0**	**0**	**0**	$\mathbf{A \leftarrow A \oplus B}$

Insert

- The **insert** micro-operation inserts a new value into a group of bits.
- This is done by first masking that group of bits and OR-ing it with the required value.
- If, for instance, the value of *A* is 0110 1010 and we would like to replace the leftmost four bits by 1001, we first mask the bits we want to replace

0110 1010	*A* before
$\underline{0000\ 1111}$	*B* (mask)
0000 1010	*A* after masking

and then insert the new value:

0000 1010	*A* before
$\underline{1001\ 0000}$	*B* (insert)
1001 1010	*A* after insertion

- Such a transformation can be performed using the following RTL statements

$A \leftarrow A \wedge B$

$B \leftarrow$ the appropriate operand

$A \leftarrow A \vee B$

Shift microoperations

There are three types of shift microoperations:

- logical shift,
- circular shift, and
- arithmetic shift

The following is a list of the different shift microoperations and their notations.

Table 23: Shift microoperations

Symbolic designation	**Description**
$R \leftarrow shl\ R$	Shift-left register R
$R \leftarrow shr\ R$	Shift-right register R
$R \leftarrow cil\ R$	Circular shift-left register R
$R \leftarrow cir\ R$	Circular shift-right register R
$R \leftarrow ashl\ R$	Arithmetic shift-left register R
$R \leftarrow ashr\ R$	Arithmetic shift

uOttawa

Logical Shift

- A **logical** shift is one that inserts a ’0’ into the register through the serial input.
- *shl* and *shr* will be used to symbolize shift-left and sfift-right microoperations.

Example 40.

1001	*A* before	1001	*A* before
0100	*A* after ($A \leftarrow$ ***shr*** A)	0010	*A* after ($A \leftarrow$ ***shl*** A)

Circular Shift

- The **circular** shift (also known as **rotate** micro-operation) circulates the bit of the register around the two ends without loss of information,
- This is accomplished by connecting the serial output of the shift register to its serial input.

Example 41.

1001	*A* before	1001	*A* before
0011	*A* after ($A \leftarrow$ ***cil*** A)	1100	*A* after ($A \leftarrow$ ***cir*** A)

Arithmetic Shift

- An **arithmetic** shift is a micro-operation that shifts a **signed** number left or right.
- An arithmetic shift-left performs a signed multiplication by 2.
- An arithmetic shift-right performs a signed division by 2.
- An arithmetic shift has to be done in a particular way so as not to change the value of the sign bit

Arithmetic Shift-Right

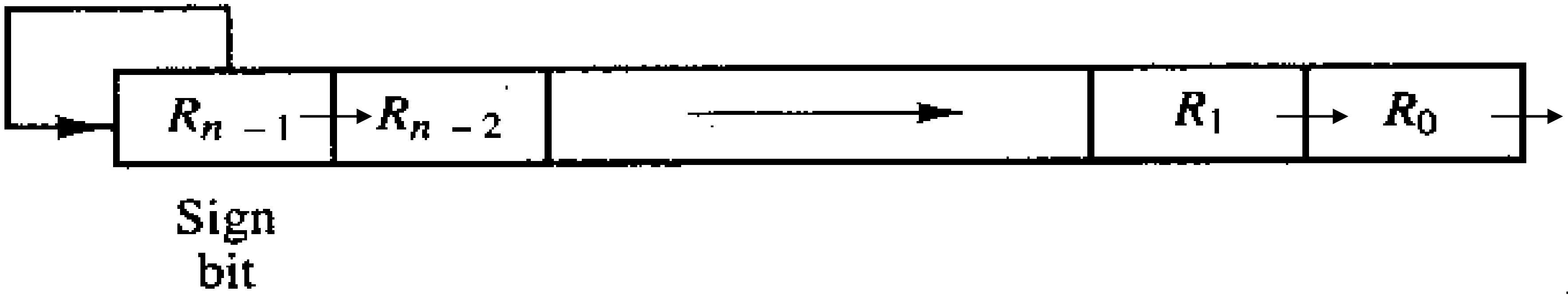
The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Equivalent with division by 2.

Example 42.

-7	1001	signed number *A* before shift
-4	1100	*A* after ($A \leftarrow$ ***ashr*** *A*)

+6	0110	signed number *A* before shift
+3	0011	*A* after ($A \leftarrow$ ***ashr*** *A*)

Figure 24 depicts and arithmetic shift-right micro-operation.



Arithmetic Shift-Left

- The arithmetic shift-left inserts a '0' into the register's least significant position and shifts all the other bits to the left (including the sign bit). It is equivalent with multiplication by 2.
- In this case, a sign reversal may happen, which is then considered as an overflow.
- An overflow occurs in an *n*-bit register *R* after an arithmetic shift-left if **initially**, before the shift operation, the sign bit R_{n-1} is different from its precedent bit R_{n-2} .
- An overflow D flip-flop *Vs* can then be used to detect an arithmetic shift-left overflow.

$$V_s \leftarrow R_{n-1} \oplus R_{n-2} \;,\quad R \leftarrow ashl\ R\,.$$

- For the overflow detection to be successful, both microoperations have to be triggered with the **same** clock pulse.

NOTE: The „carries rule" from addition cannot be applied here since there is no adder involved and no carries are generated!

Example 43.

-7	1001	signed number *A*
+2	0010	*A* after ($A \leftarrow$ *ashl* *A*) [overflow]

-2	1110	signed number *A*
-4	1100	*A* after ($A \leftarrow$ *ashl* *A*) [no overflow]

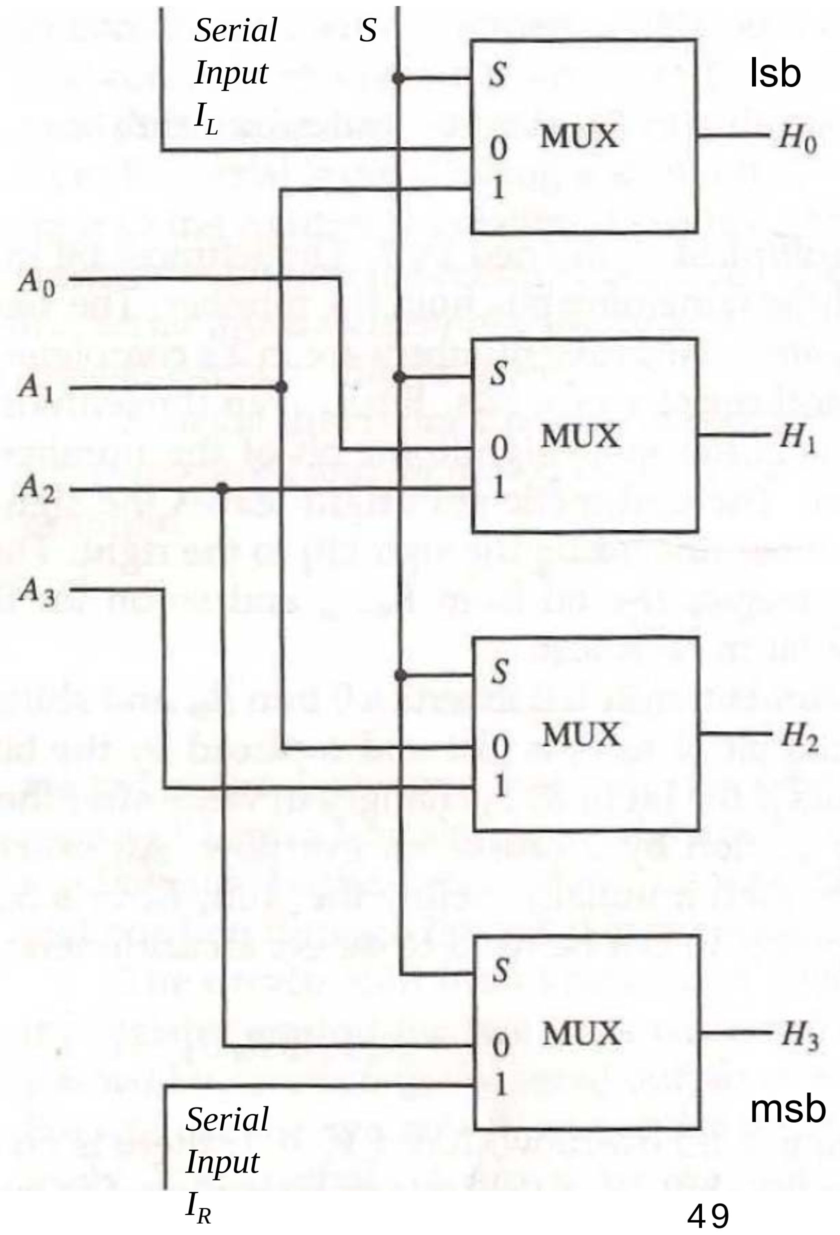
Hardware Implementation

Left/Right Shift (logical/arithmetic/circular)

- A combinational shifter is shown next.
- In this case:
  - if S = 0, then H = *shl A* (lsb → msb, down in the figure)
  - if S = 1, then H = *shr A* (msb → lsb , up in the figure)
- The designer can choose what to feed into the serial inputs depending on the desired functionality of the circuit.

Select	msb	Out		lsb	
S	H_3	H_2	H_1	H_0	
0	I_R	A_3	A_2	A_1	*shr A*
1	A_2	A_1	A_0	I_L	*shl A*

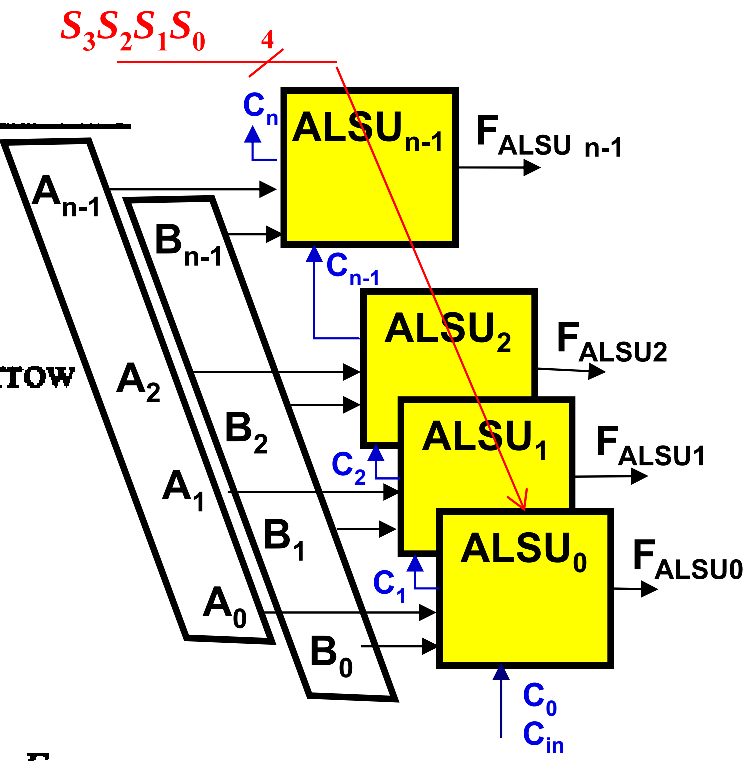
Note: Fig. 4-12 of the textbook presents a 4-bit shifter assuming that msb = H_0 and lsb = H_3!



Arithmetic Logic Shift Unit

- An **arithmetic logic unit** (**ALU**) is a combinational circuit that performs arithmetic and logic microoperations.
- An **arithmetic logic shift unit** (**ALSU**) is an ALU that also performs shift microoperations.
- One way of implementing an *n*-bit ALSU is to build a one-stage ALSU (for one bit only) and then stack *n* replicates of it together (one for each bit) to form the full ALSU.
- All replicates of the one-stage ALSU will use the same selection bits.

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F = \text{shr } A$	Shift right A into F
1	1	×	×	×	$F = \text{shl } A$	Shift left A into F

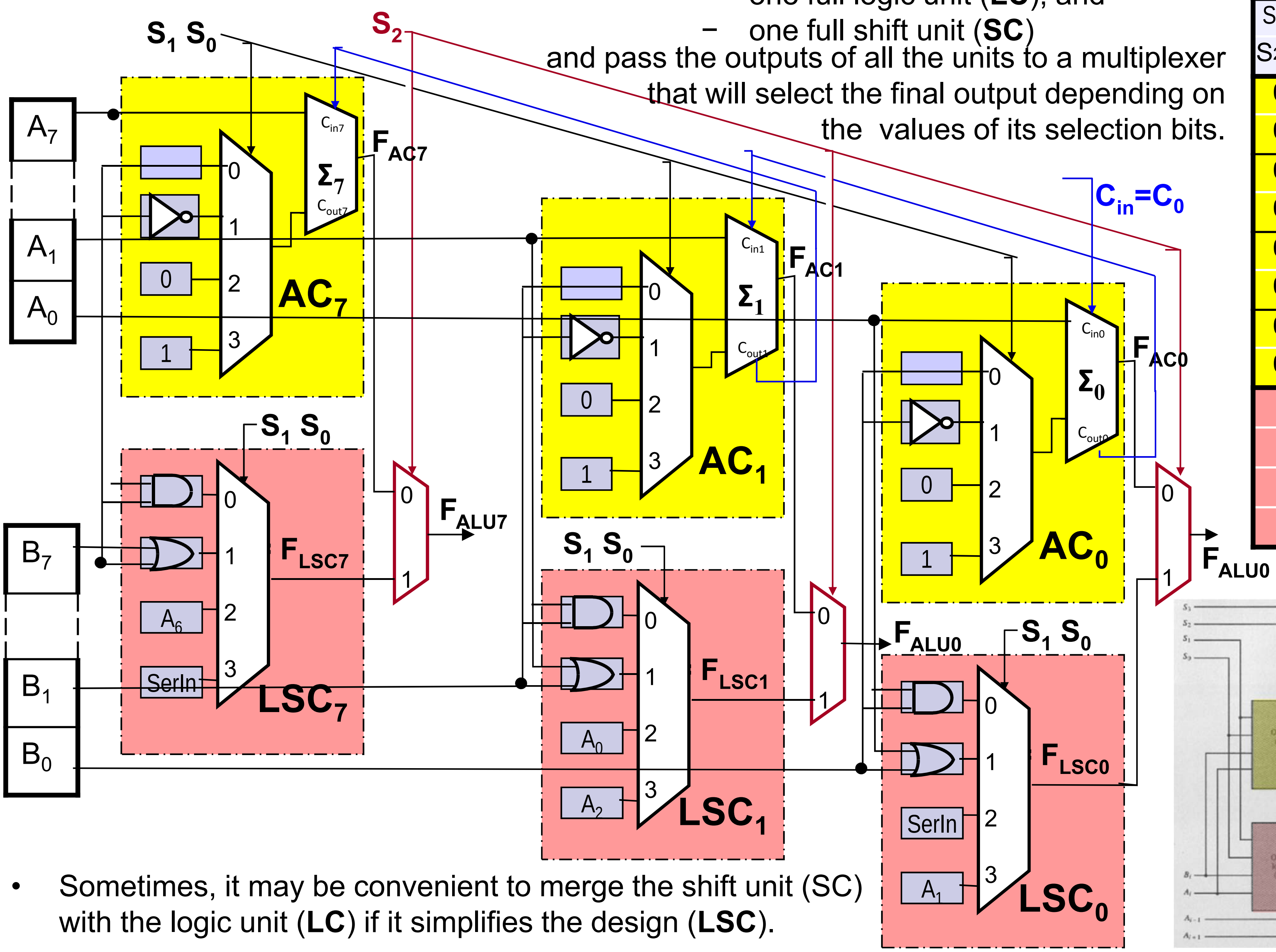


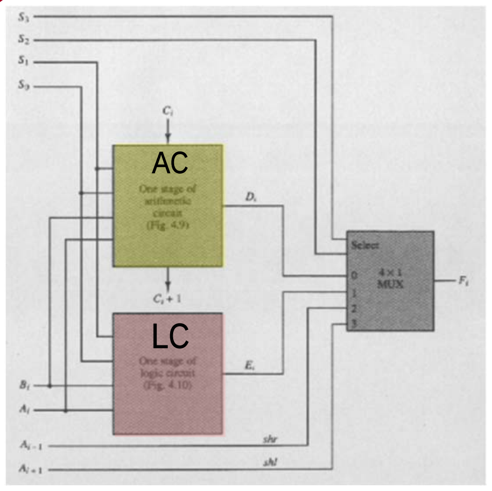
ALSU(for 8 bit)

- Another way of implementing an ALSU is to build
  - one full arithmetic unit (**AC**)
  - one full logic unit (**LC**), and
  - one full shift unit (**SC**)

  and pass the outputs of all the units to a multiplexer that will select the final output depending on the values of its selection bits.

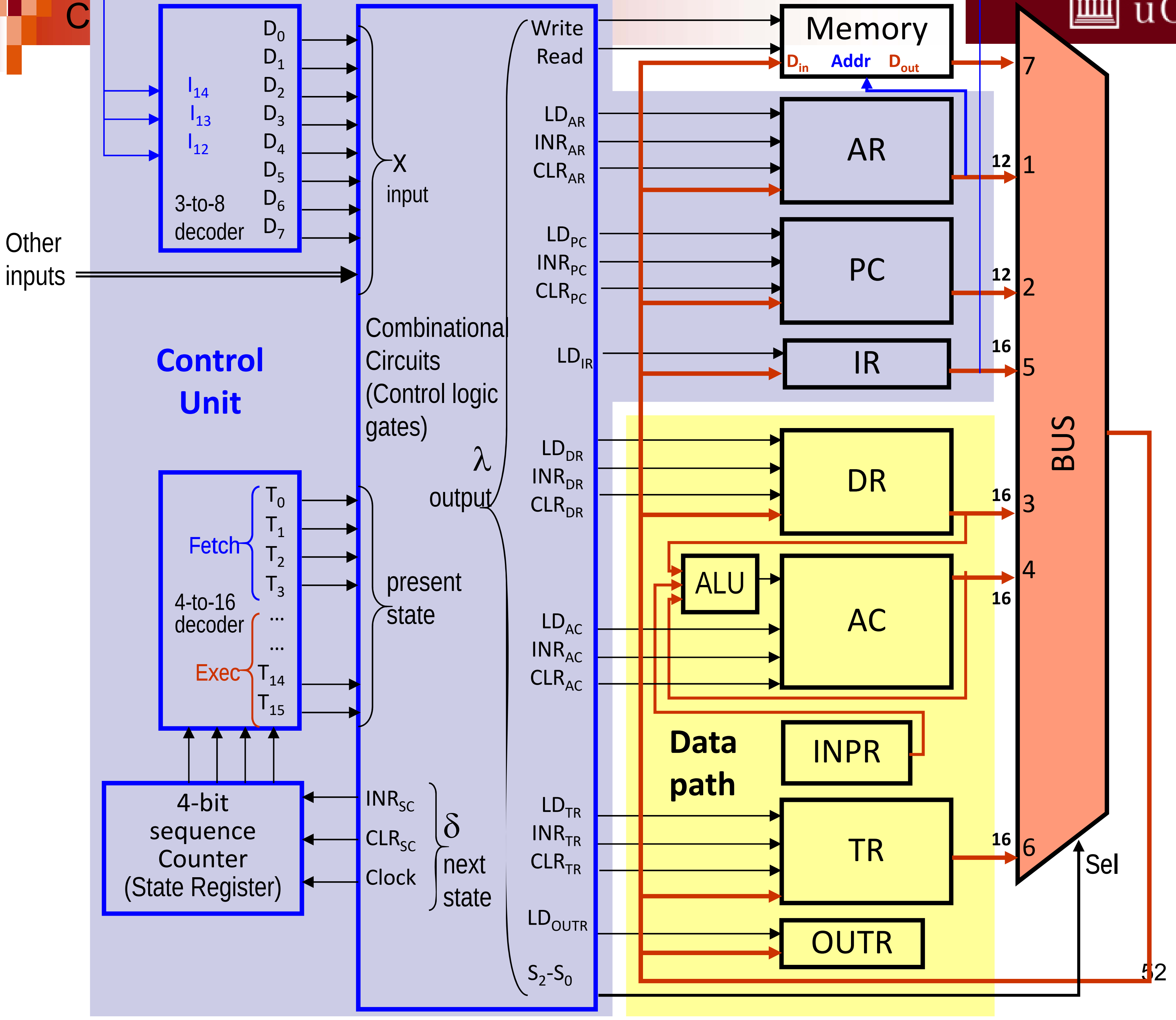
Select Bits	**Output**
$S_2S_1S_0C_{in}$	F_{ALU}
0 0 0 0	F_{AC} = A+B
0 0 0 1	F_{AC} = A+B+1
0 0 1 0	F_{AC} = A−B−1
0 0 1 1	F_{AC} = A−B
0 1 0 0	F_{AC} = A
0 1 0 1	F_{AC} = A+1
0 1 1 0	F_{AC} = A−1
0 1 1 1	F_{AC} = A
1 0 0 x	F_{LSC} = A ∧ B
1 0 1 x	F_{LSC} = A ∨ B
1 1 0 x	F_{LSC} = shift left
1 1 1 x	F_{LSC} = shift right




S3
S2
S1
S0
Ci
AC
One stage of arithmetic circuit (Fig. 4.9)
Di
Ci+1
LC
One stage of logic circuit (Fig. 4.10)
Ei
Select
4 × 1 MUX
Fi
Bi
Ai
Ai-1
Ai+1
shr
shl

- Sometimes, it may be convenient to merge the shift unit (SC) with the logic unit (**LC**) if it simplifies the design (**LSC**).

C



BASIC COMPUTER