Computer Organization (Register transfer language)

Prepared by:
Prof. Miksha Solanki
miksha.solanki21591@paruluniversity.ac.in
EC dept, PIET

REGISTER TRANSFER LANGUAGE

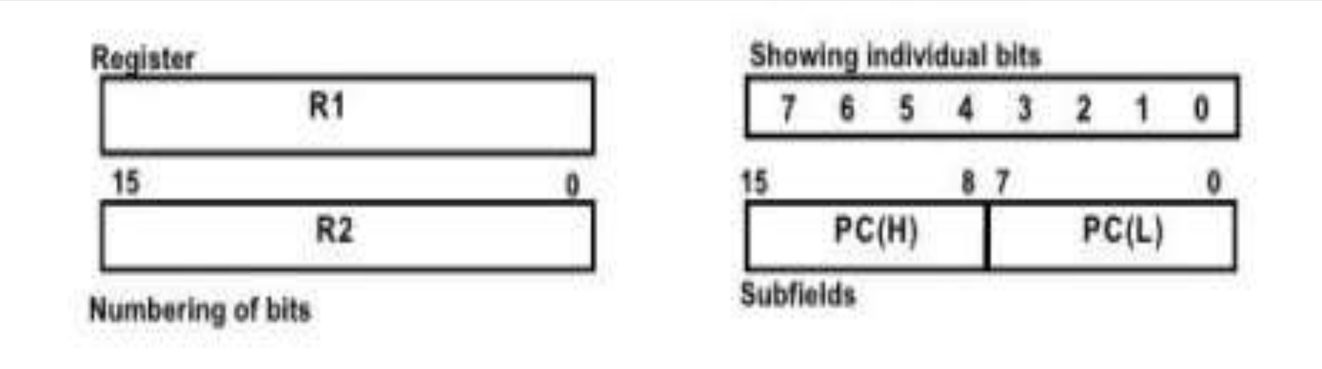
- specific notation used to specify the digital system is called register transfer language.
- For any function of the computer, the register transfer language can be used to describe the (sequence of) micro-operations.
- Register transfer language
  - ✓ A symbolic language
  - ✓ A convenient tool for describing the internal organization of digital computers
  - ✓ Can also be used to facilitate the design process of digital systems

REGISTER TRANSFER MICRO OPERATIONS

- Capital letters are used to designate registers

  Ex: MAR- memory address register (holds an address for memory unit)

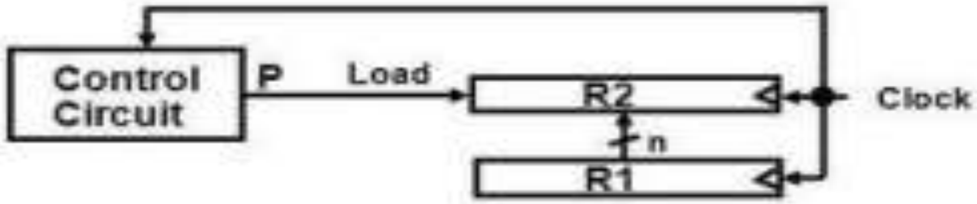
  PC – Program Counter and IR – Instruction Register
- Individual flip-flops are used in n-bit registers
- The symbolic representation of the register transfer is R2 ⟵R1
- The information is transferred from the register R1 (source) to register R2 (destination).
- Common ways of representing a register is shown in figure below:



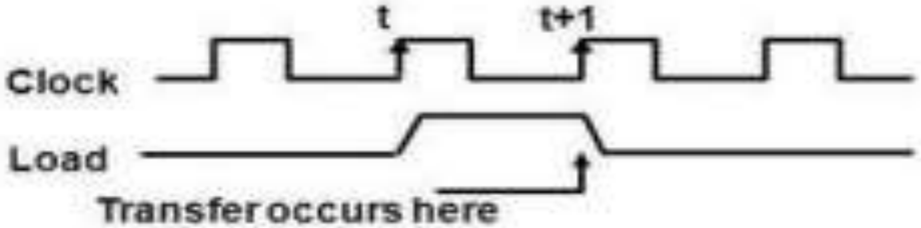
Implementation of controlled transfer

P: R2 ← R1

Block diagram



Timing diagram



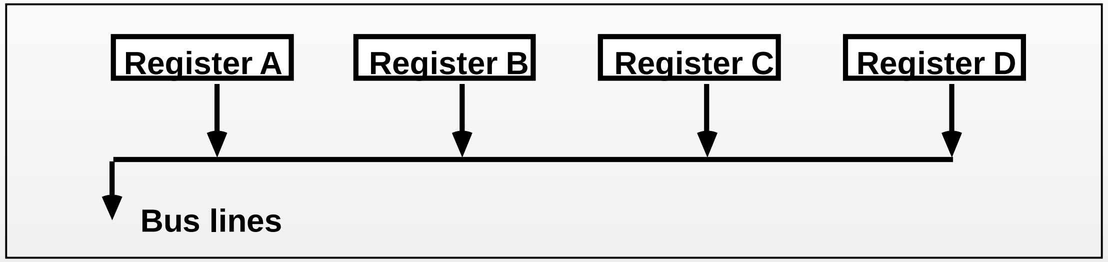
- The same clock controls the circuits that generate the control function and the destination register
- Registers are assumed to use *positive-edge-triggered* flip-flops

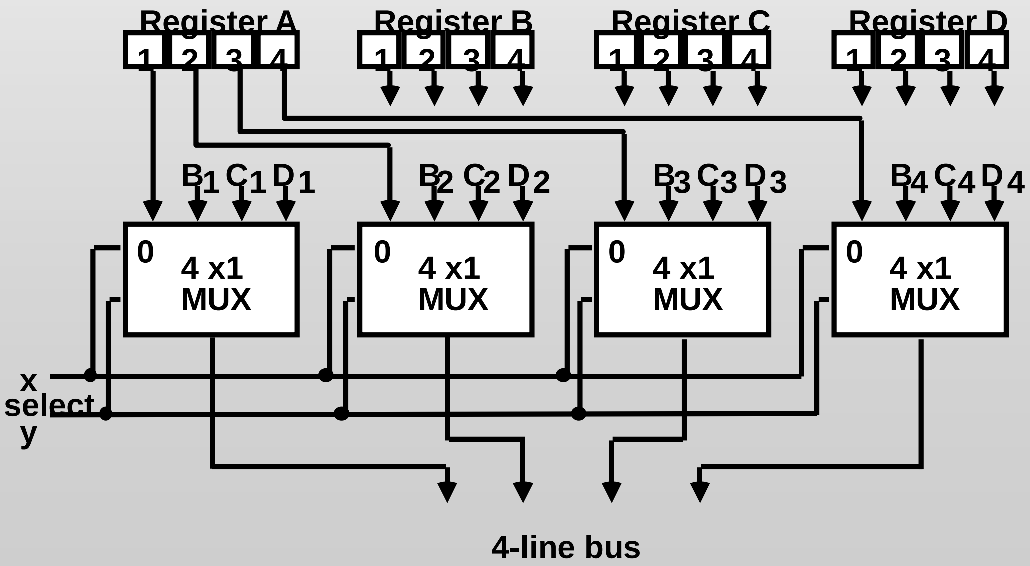
BUS AND MEMORY TRANSFERS

- Since transferring info from one register to another in separate lines becomes complex, a more efficient scheme called common bus system is used.
- A bus consists of a common set of lines, one for each bit of a register through which the binary info is transferred at a time.
- Control signals determine which register is selected by the bus during each particular register transfer.
- One way of constructing the common bus system is using multiplexers
- The bus consists of 4 registers and four 4 by 1 mux each having data inputs 0 through 3 and selection lines S1 and S0
- Ex: o/p of register A is connected to input 0 of mux1 because this i/p is labelled A1.

 When S1S0=00, the four bits of one register are selected and transfer it to four line common bus.

- From a register to bus: BUS ⟵ R



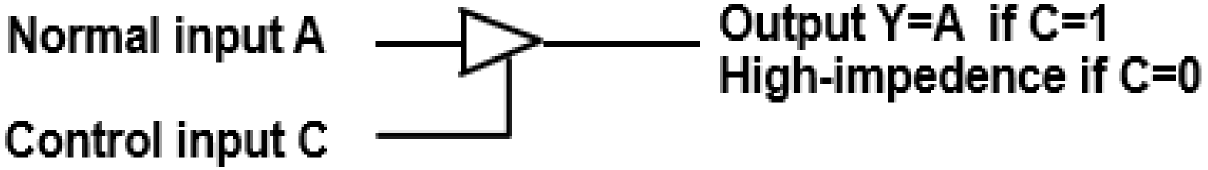


THREE STATE BUFFER

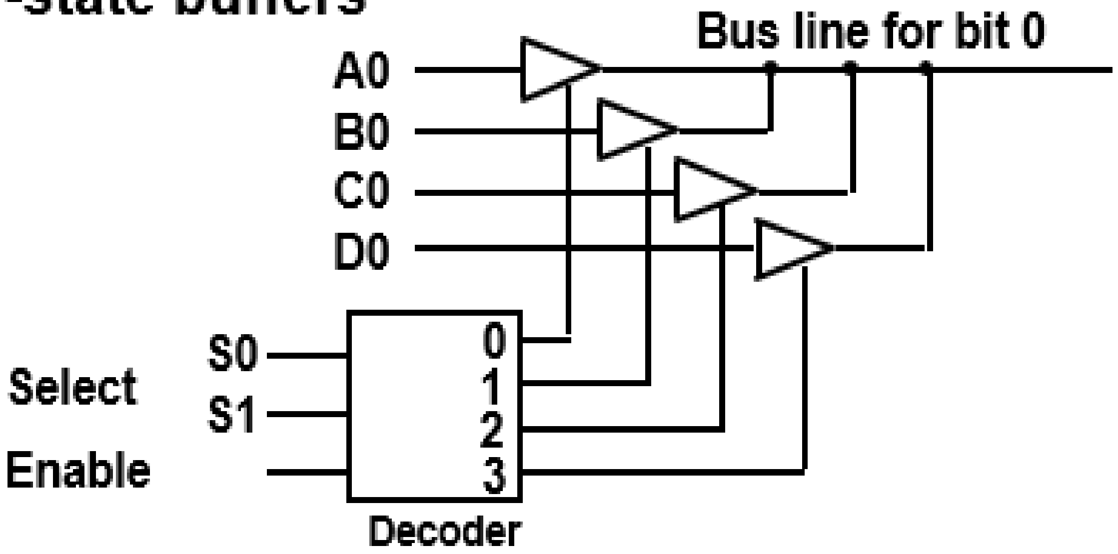
- A three state gate is a digital circuit that exhibits threes states
  - ✓ Two of them being conventional logics 0 and 1
  - ✓ Third is high impedance state which behaves as an open circuit i.e., output disconnected
- They may perform any conventional logics such as AND or NAND
- Where as in three state buffer gate
  - ✓ Control i/p determine the o/p state
  - ✓ When control i/p is 1, the o/p is enabled and behaves as normal i/p
  - ✓ When control i/p is 0, the o/p acts as high impedance state
  - ✓ Because of this high impedance state a large number of 3 state gates o/p's

    can be connected to form a common bus line without loading effect

Three-State (Bus) Buffers

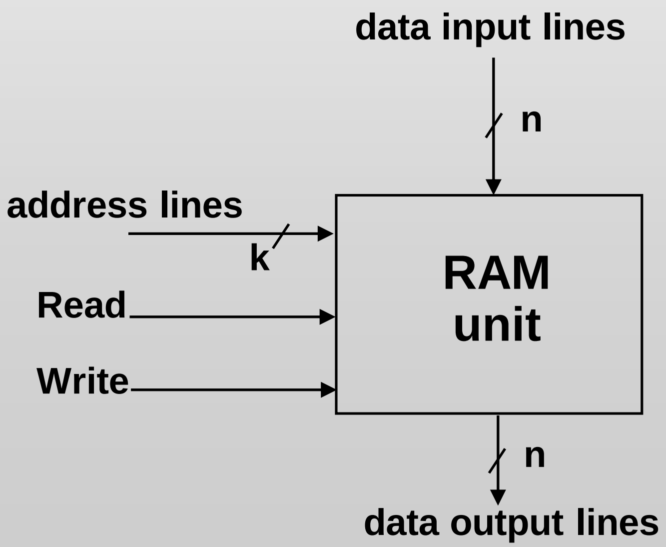


Bus line with three-state buffers



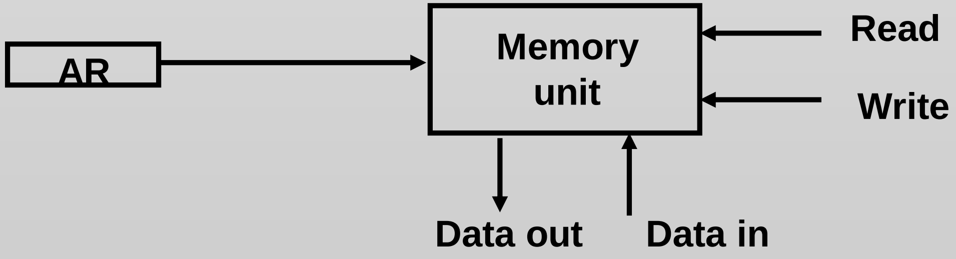
MEMORY (RAM)

- Memory (RAM) can be thought as a sequential circuit containing some number of registers
- These registers hold the words of memory
- Each of the r registers is indicated by an address
- These addresses range from 0 to r-1
- Each register (word) can hold n bits of data
- Assume the RAM contains $r = 2^k$ words. It needs the following
  - n data input lines
  - n data output lines
  - k address lines
  - A Read control line
  - A Write control line



MEMORY TRANSFER

- Collectively, the memory is viewed at the register level as a device, M.
- Since it contains multiple locations, we must specify which address in memory we will be using
- This is done by indexing memory references
- Memory is often accessed in computer systems by putting the desired address in a special register, the Memory Address Register (MAR, or AR)
- When memory is accessed, the contents of the MAR get sent to the memory unit's address lines



MEMORY READ

- To read a value from a location in memory and load it into a register, the register transfer language notation looks like this:

  R1 $\leftarrow$ M[MAR]

- This causes the following to occur
  - The contents of the MAR get sent to the memory address lines
  - A Read (= 1) gets sent to the memory unit
  - The contents of the specified address are put on the memory's output data lines
  - These get sent over the bus to be loaded into register R1

MEMORY WRITE

- To write a value from a register to a location in memory looks like this in register transfer language:

  M[MAR] ← R1

- This causes the following to occur
  - The contents of the MAR get sent to the memory address lines
  - A Write (= 1) gets sent to the memory unit
  - The values in register R1 get sent over the bus to the data input lines of the memory
  - The values get loaded into the specified address in the memory

SUMMARY OF R. TRANSFER MICROOPERATIONS

A ← B	Transfer content of reg. B into reg. A
AR ← DR(AD)	Transfer content of AD portion of reg. DR into reg. AR
A ← constant	Transfer a binary constant into reg. A
A BUS ← R1,	Transfer content of R1 into bus A and, at the same time,
R2 ← ABUS	transfer content of bus A into R2
AR	Address register
DR	Data register
M[R]	Memory word specified by reg. R
M	Equivalent to M[AR]
DR ← M	Memory *read* operation: transfers content of memory word specified by AR into DR
M ← DR	Memory *write* operation: transfers content of DR into memory word specified by AR

MICROOPERATIONS

- The operations executed on the data stored in the registers are called micro operations.
- The functions built into registers are examples of micro- operations - Shift - Load - Clear - Increment ...etc.

Classifications of micro operations:

1. Register transfer micro operations
2. Arithmetic micro operations
3. Logic micro operations
4. Shift micro operations

- This is an elementary operation performed on the information stored in registers. Ex: shift, count, clear and load

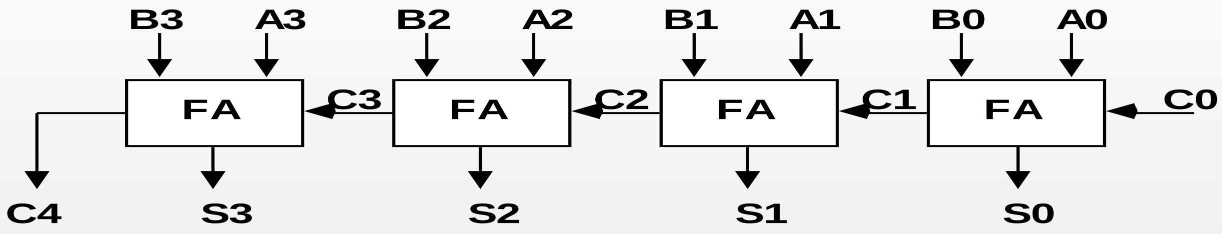
ARITHMETIC MICROOPERATIONS

- The basic arithmetic microoperations are
  - Addition
  - Subtraction
  - Increment
  - Decrement
- The additional arithmetic microoperations are
  - Add with carry
  - Subtract with borrow
  - Transfer/Load

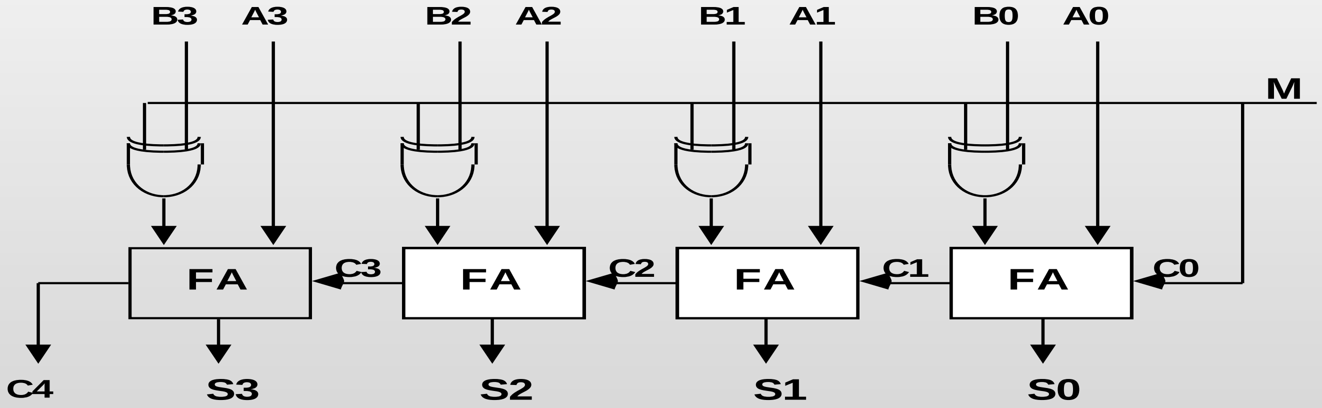
R3 ←	R1 + R2	Contents of R1 plus R2 transferred to R3
R3 ←	R1 - R2	Contents of R1 minus R2 transferred to R3
R2 ←	R2’	Complement the contents of R2
R2 ←	R2’+ 1	2's complement the contents of R2 (negate)
R3 ←	R1 + R2’+ 1	subtraction
R1 ←	R1 + 1	Increment
R1 ←	R1 - 1	Decrement

BINARY ADDER / SUBTRACTOR / INCREMENTER

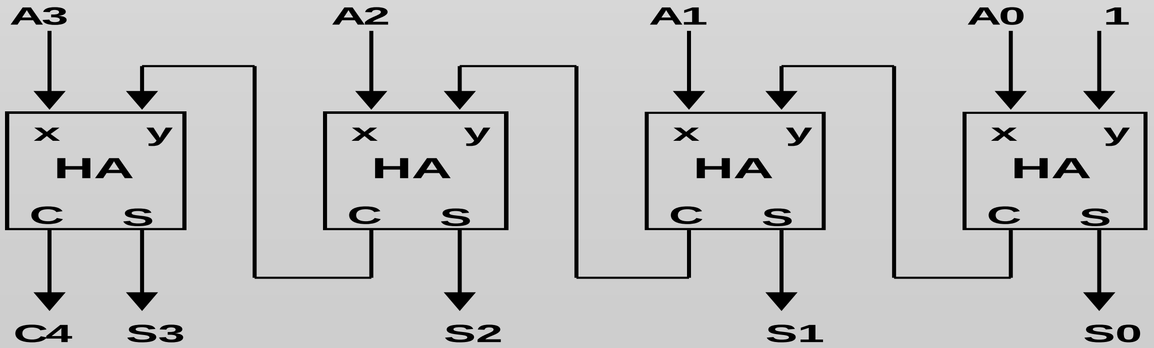
Binary Adder



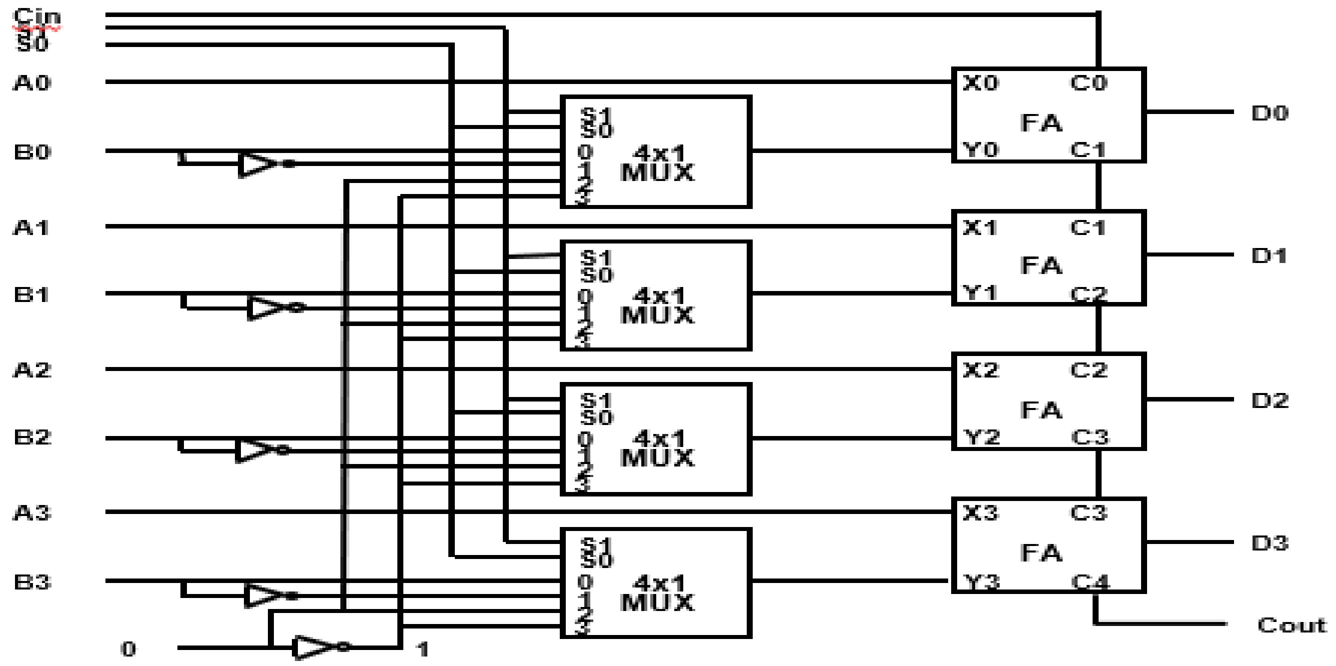
Binary Adder-Subtractor



Binary Incrementer



ARITHMETIC CIRCUIT



S1	S0	Cin	Y	Output	Microoperation
0	0	0	B	D = A + B	Add
0	0	1	B	D = A + B + 1	Add with carry
0	1	0	B'	D = A + B'	Subtract with borrow
0	1	1	B'	D = A + B'+ 1	Subtract
1	0	0	0	D = A	Transfer A
1	0	1	0	D = A + 1	Increment A
1	1	0	1	D = A - 1	Decrement A
1	1	1	1	D = A	Transfer A

LOGIC MICROOPERATIONS

- Specify binary operations on the strings of bits in registers
  - Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data
  - useful for bit manipulations on binary data
  - useful for making logical decisions based on the bit value
- There are, in principle, 16 different logic functions that can be defined over two binary input variables

A	B	F_0	F_1	F_2	…	F_{13}	F_{14}	F_{15}
0	0	0	0	0	…	1	1	1
0	1	0	0	0	…	1	1	1
1	0	0	0	1	…	0	1	1
1	1	0	1	0	…	1	0	1

However, most systems only implement four of these:

AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), Complement/NOT

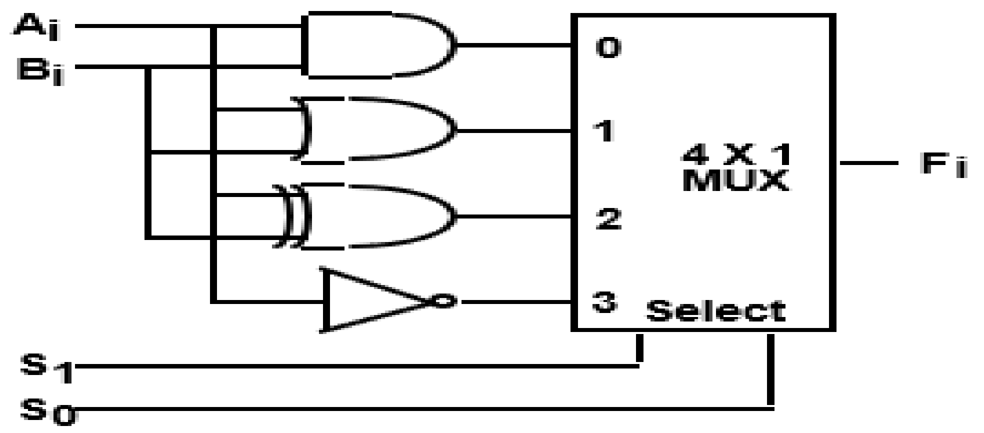
The others can be created from combination of these

LIST OF LOGIC MICROOPERATIONS

- List of Logic Microoperations
  - - 16 different logic operations with 2 binary vars.
  - - n binary vars →2^2^n functions
- Truth tables for 16 functions of 2 variables and the corresponding 16 logic micro-operations

x y	0 0 1 1 0 1 0 1	*Boolean Function*	*Micro-Operations*	*Name*
	0 0 0 0	F0 = 0	$F \leftarrow 0$	Clear
	0 0 0 1	F1 = xy	$F \leftarrow A \wedge B$	AND
	0 0 1 0	F2 = xy'	$F \leftarrow A \wedge B'$	
	0 0 1 1	F3 = x	$F \leftarrow A$	Transfer A
	0 1 0 0	F4 = x'y	$F \leftarrow A' \wedge B$	
	0 1 0 1	F5 = y	$F \leftarrow B$	Transfer B
	0 1 1 0	F6 = $x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
	0 1 1 1	F7 = x + y	$F \leftarrow A \vee B$	OR
	1 0 0 0	F8 = (x + y)'	$F \leftarrow (A \vee B)'$	NOR
	1 0 0 1	F9 = $(x \oplus y)'$	$F \leftarrow (A \oplus B)'$	Exclusive-NOR
	1 0 1 0	F10 = y'	$F \leftarrow B'$	Complement B
	1 0 1 1	F11 = x + y'	$F \leftarrow A \vee B'$	
	1 1 0 0	F12 = x'	$F \leftarrow A'$	Complement A
	1 1 0 1	F13 = x' + y	$F \leftarrow A' \vee B$	
	1 1 1 0	F14 = (xy)'	$F \leftarrow (A \wedge B)'$	NAND
	1 1 1 1	F15 = 1	$F \leftarrow$ all 1's	Set to all 1's

HARDWARE IMPLEMENTATION OF LOGIC MICROOPERATIONS



Function table

S_1	S_0	Output	µ-operation
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	$F = A'$	Complement

SELECTIVE SET

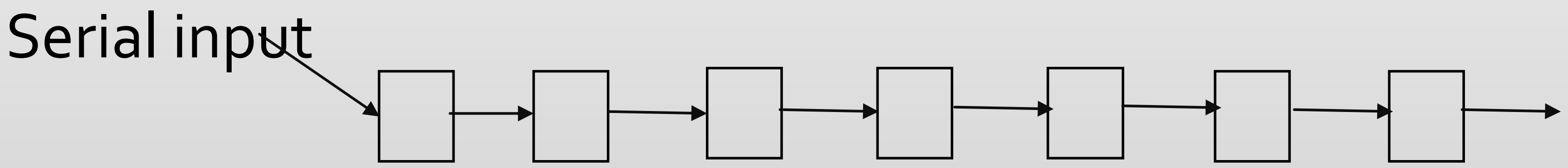
- In a selective set operation, the bit pattern in B is used to *set* certain bits in A

$$
\begin{array}{ll}
1100 & A_t \\
1010 & B \\
\hline
1110 & A_{t+1} \qquad (A \leftarrow A + B)
\end{array}
$$

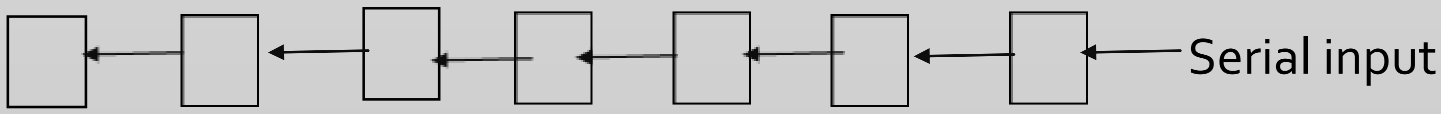
- If a bit in B is set to 1, that same position in A gets set to 1, otherwise that bit in A keeps its previous value

SHIFT MICROOPERATIONS

- There are three types of shifts
  - Logical shift
  - Circular shift
  - Arithmetic shift
- What differentiates them is the information that goes into the serial input
- **A right shift operation**

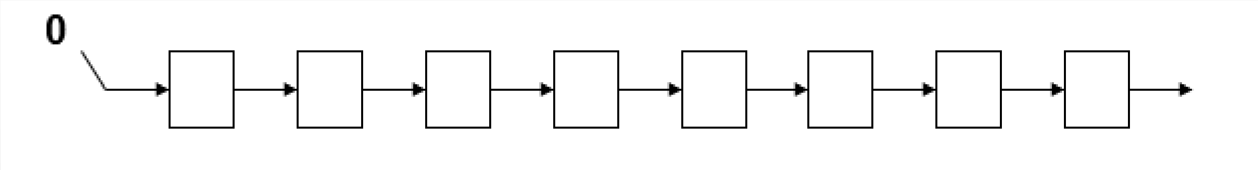


- **A left shift operation**

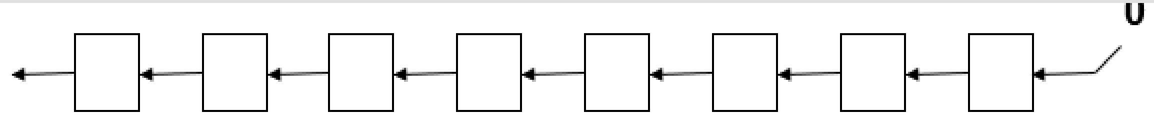


LOGICAL SHIFT

- In a logical shift the serial input to the shift is a 0.
- A right logical shift operation:



A left logical shift operation:



In a Register Transfer Language, the following notation is used

shl for a logical shift left

shr for a logical shift right

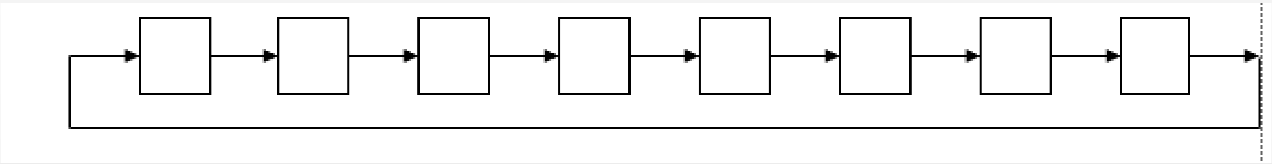
Examples:

$R2 \leftarrow shr\ R2$

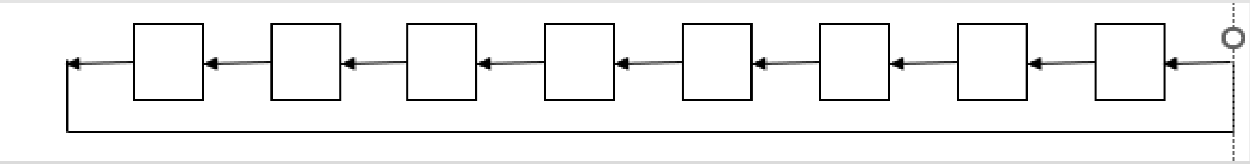
$R3 \leftarrow shl\ R3$

CIRCULAR SHIFT

- In a circular shift the serial input is the bit that is shifted out of the other end of the register.
- A right circular shift operation:



- A left circular shift operation



In a RTL, the following notation is used

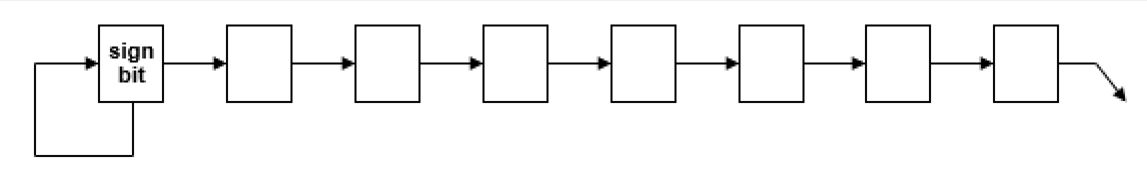
cil for a circular shift left
cir for a circular shift right

Examples:
$R2 \leftarrow$ cir $R2$
$R3 \leftarrow$ cil $R3$

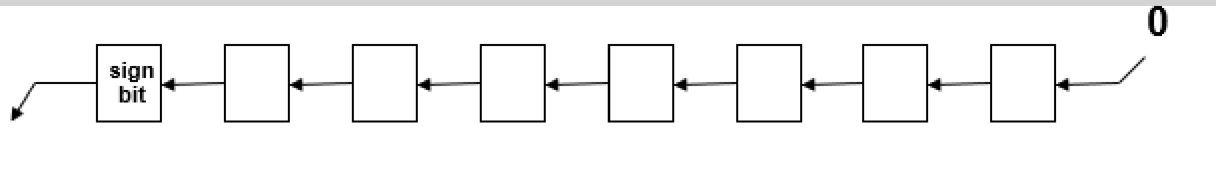
ARITHMETIC SHIFT

- An Arithmetic shift is meant for signed binary numbers (integer)
- An Arithmetic left shift **multiplies** a signed number **by two**
- Arithmetic right shift **divides** a signed number **by two**
- The main distinction of an arithmetic shift is that it must keep the sign of the number the same as it performs the multiplication or division

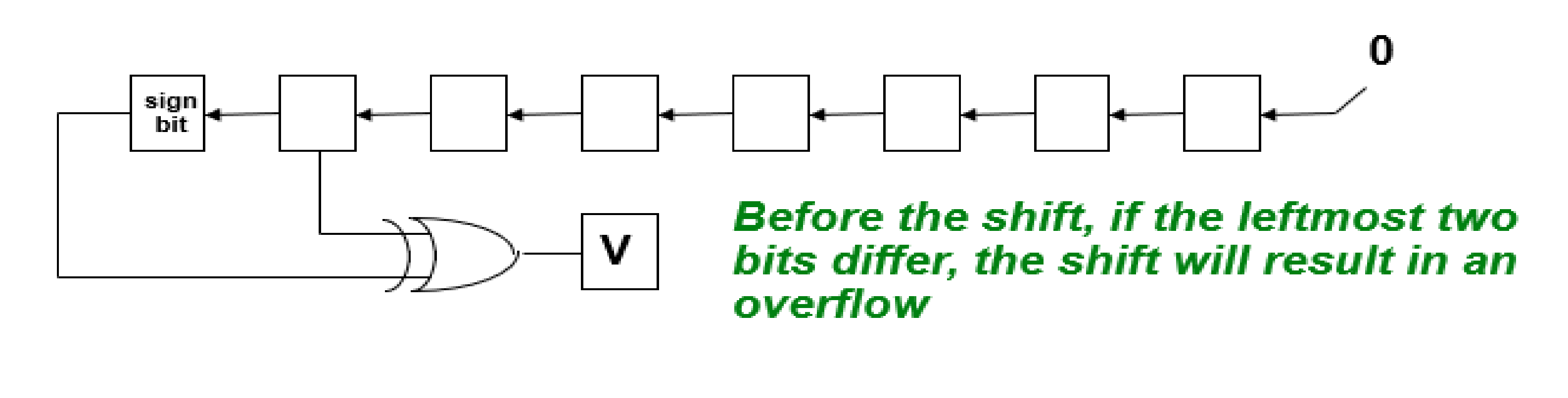
- **A right arithmetic shift operation**



- **A left arithmetic shift operation**

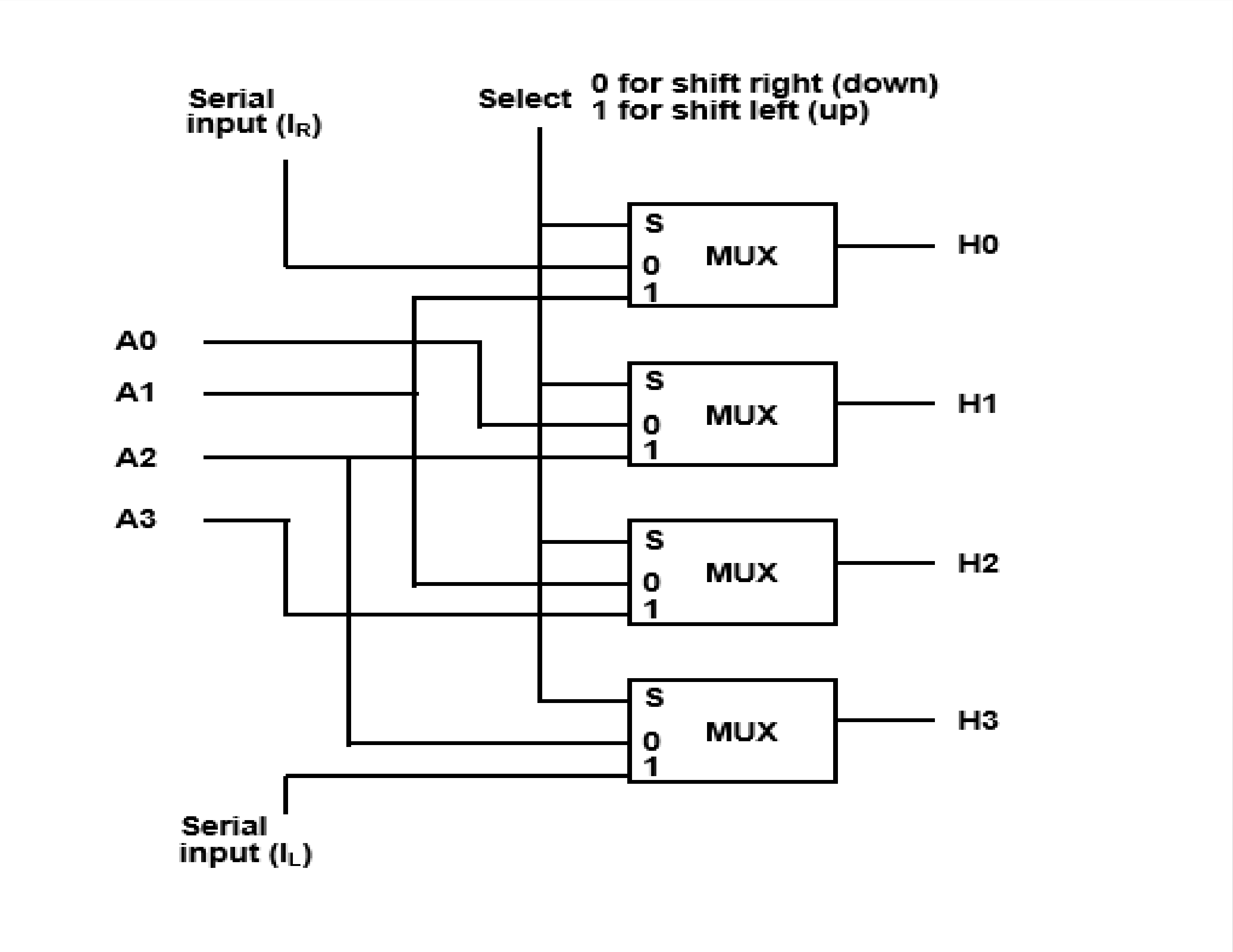


- A left arithmetic shift operation must be checked for the overflow

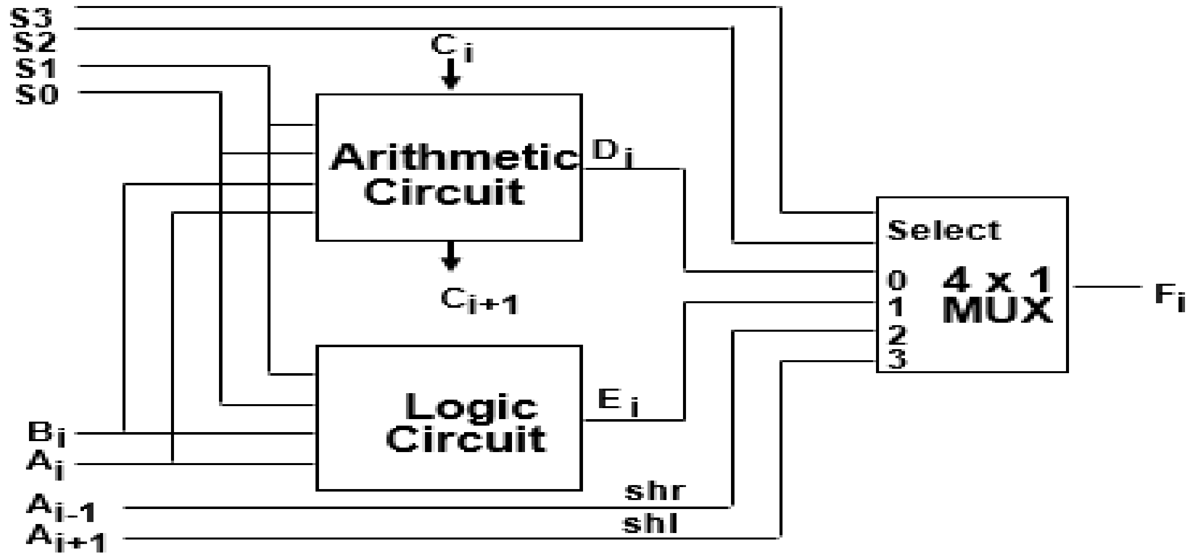


- In a RTL, the following notation is used
  - *ashl* for an arithmetic shift left
  - *ashr* for an arithmetic shift right
  - Examples:
    - R2 ← *ashr* R2
    - R3 ← *ashl* R3

HARDWARE IMPLEMENTATION OF SHIFT MICROOPERATIONS



ARITHMETIC LOGIC SHIFT UNIT



S3	S2	S1	S0	Cin	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	$F = A + B'$	Subtract with borrow
0	0	1	0	1	$F = A + B' + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	TransferA
0	1	0	0	X	$F = A \wedge B$	AND
0	1	0	1	X	$F = A \vee B$	OR
0	1	1	0	X	$F = A \oplus B$	XOR
0	1	1	1	X	$F = A'$	Complement A
1	0	X	X	X	$F = $ shr A	Shift right A into F
1	1	X	X	X	$F = $ shl A	Shift left A into F

REFRENCE

- COMPUTER SYSTEM ARCHITECTURE, MORRIS M. MANO, 3RD EDITION, PRENTICE HALL INDIA.