JGI JAIN DEEMED-TO-BE UNIVERSITY | Powered By Futurense

Revolutionising B.Tech

JGI JAIN DEEMED-TO-BE UNIVERSITY | Powered By Futurense

Module 1: Data Representation and Operations

Course Name: Computer Architecture and organization[22CSE104]

Total Hours : 8

Table of Content

Table of Content

Aim

To equip students in the fundamentals and understanding the Concepts of Data Representation and register and micro Operations.

Objective

a. Discuss on the Data representation methods.
b. Understanding different Data types,
c. Understanding and practice of Register and Micro operations

SIMPLE DIGITAL SYSTEMS

- Combinational and sequential circuits
  can be used to create simple digital systems.

- These are the low-level building blocks of a digital computer.

- Simple digital systems are frequently characterized in terms of
  - the registers they contain, and
  - the operations that they perform.

- Typically,
  - What operations are performed on the data in the registers
  - What information is passed between registers

DATA REPRESENTATION

Data Types

Complements

Fixed Point Representations

Floating Point Representations

Other Binary Codes

Error Detection Codes

DATA REPRESENTATION

Information that a Computer is dealing with

* Data
  - Numeric Data
    Numbers(Integer, real)
  - Non-numeric Data
    Letters, Symbols

* Relationship between data elements
  - Data Structures
    Linear Lists, Trees, Rings, etc

* Program(Instruction)

NUMERIC DATA REPRESENTATION

Data

Numeric data - numbers(integer, real)

Non-numeric data - symbols, letters

Number System

Nonpositional number system

- Roman number system

Positional number system

- Each digit position has a value called a *weight* associated with it

- Decimal, Octal, Hexadecimal, Binary

Base (or radix) R number

- Uses R distinct symbols for each digit
- Example $A_R = a_{n-1}\, a_{n-2} \ldots a_1\, a_0 .a_{-1} \ldots a_{-m}$

Radix point(.) separates the integer portion and the fractional portion

- $V(A_R) = \sum_{i=-m}^{n-1} a_i R^i$

R = 10 Decimal number system, R = 2 Binary

R = 8 Octal, R = 16 Hexadecimal

WHY POSITIONAL NUMBER SYSTEM IN THE DIGITAL COMPUTERS ?

Major Consideration is the *COST* and *TIME*

- Cost of building *hardware*
  Arithmetic and Logic Unit, CPU, Communications
- Time to processing

Arithmetic - Addition of Numbers - *Table for Addition*

* Non-positional Number System
  - Table for addition is infinite
    --> Impossible to build, very expensive even if it can be built

* Positional Number System
  - Table for Addition is finite
    --> Physically realizable, but cost wise the smaller the table size, the less expensive --> Binary is favorable to Decimal

Binary Addition Table

	0	1
0	0	1
1	1	10

Decimal Addition Table

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

REPRESENTATION OF NUMBERS
POSITIONAL NUMBERS

Decimal	Binary	Octal	Hexadecimal
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Binary, octal, and hexadecimal conversion

Octal	1	2	7	5	4	3
Binary	1 0 1 0	1 1 1 1	0 1 1	1 0 0	0 1 1	
Hexa	A	F	6	3		

CONVERSION OF BASES

Base R to Decimal Conversion

$$A = a_{n-1}\ a_{n-2}\ a_{n-3} \ldots a_0 . a_{-1} \ldots a_{-m}$$

$$V(A) = \sum a_k R^k$$

$(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1}$

$= 7 \times 64 + 3 \times 8 + 6 \times 1 + 4/8 = (478.5)_{10}$

$(110110)_2 = \ldots = (54)_{10}$

$(110.111)_2 = \ldots = (6.785)_{10}$

$(F3)_{16} = \ldots = (243)_{10}$

$(0.325)_6 = \ldots = (0.578703703 \ldots\ldots\ldots\ldots)_{10}$

Decimal to Base R number

- Separate the number into its *integer* and *fraction* parts and convert each part separately.
- Convert *integer part* into the base R number

  --> successive divisions by R and accumulation of the remainders.
- Convert *fraction part* into the base R number

  --> successive multiplications by R and accumulation of integer digits

EXAMPLE

Convert 41.6875_{10} to base 2.

Integer = 41

41	
20	1
10	0
5	0
2	1
1	0
0	1

$(41)_{10} = (101001)_2$

Fraction = 0.6875

0.6875 x 2 = 1.3750

0.7500 x 2 = 1.5000

1.5000 x 2 = 1.0000

$(0.6875)_{10} = (0.1011)_2$

$$(41.6875)_{10} = (101001.1011)_2$$

Exercise

Convert $(63)_{10}$ to base 5: $(223)_5$

Convert $(1863)_{10}$ to base 8: $(3507)_8$

Convert $(0.63671875)_{10}$ to hexadecimal: $(0.A3)_{16}$

COMPLEMENT OF NUMBERS

Two types of complements for base R number system:

- R's complement and (R-1)'s complement

The (R-1)'s Complement

Subtract each digit of a number from (R-1)

Example

- 9's complement of 835_{10} is 164_{10}
- 1's complement of 1010_2 is 0101_2(bit by bit complement operation)

The R's Complement

Add 1 to the low-order digit of its (R-1)'s complement

Example

- 10's complement of 835_{10} is $164_{10} + 1 = 165_{10}$
- 2's complement of 1010_2 is $0101_2 + 1 = 0110_2$

FIXED POINT NUMBERS

Numbers: Fixed Point Numbers and Floating Point Numbers

Binary Fixed-Point Representation

$X = x_n x_{n-1} x_{n-2} \ldots x_1 x_0 . x_{-1} x_{-2} \ldots x_{-m}$

Sign Bit(x_n): 0 for positive - 1 for negative

Remaining Bits($x_{n-1} x_{n-2} \ldots x_1 x_0 . x_{-1} x_{-2} \ldots x_{-m}$)

- Following 3 representations

Signed magnitude representation
Signed 1's complement representation
Signed 2's complement representation

Example: Represent +9 and -9 in 7 bit-binary number

Only one way to represent +9 ==> 0 001001

Three different ways to represent -9:

In signed-magnitude: 1 001001

In signed-1's complement: 1 110110

In signed-2's complement: 1 110111

In general, in computers, fixed point numbers are represented either integer part only or fractional part only.

CHARACTERISTICS OF 3 DIFFERENT REPRESENTATIONS

Complement

Signed magnitude: Complement *only* the sign bit
Signed 1's complement: Complement *all* the bits including sign bit
Signed 2's complement: Take the 2's complement of the number,

in

cluding its sign bit.

Maximum and Minimum Representable Numbers and Representation of Zero

$$X = x_n\ x_{n-1} \ldots x_0\ .\ x_{-1} \ldots x_{-m}$$

Signed Magnitude

Max:	$2^n - 2^{-m}$	011 ... 11.11 ... 1
Min:	$-(2^n - 2^{-m})$	111 ... 11.11 ... 1
Zero:	+0	000 ... 00.00 ... 0
	-0	100 ... 00.00 ... 0

Signed 1's Complement

Max:	$2^n - 2^{-m}$	011 ... 11.11 ... 1
Min:	$-(2^n - 2^{-m})$	100 ... 00.00 ... 0
Zero:	+0	000 ... 00.00 ... 0
	-0	111 ... 11.11 ... 1

Signed 2's Complement

Max:	$2^n - 2^{-m}$	011 ... 11.11 ... 1
Min:	-2^n	100 ... 00.00 ... 0
Zero:	0	000 ... 00.00 ... 0

ARITHMETIC ADDITION: SIGNED MAGNITUDE

[1] Compare their signs
[2] If two signs are the *same* ,
ADD the two magnitudes - Look out for an *overflow*
[3] If *not the same* , compare the relative magnitudes of the numbers and then *SUBTRACT* the smaller from the larger --> need a subtractor to add
[4] Determine the sign of the result

6 + 9

```
  6   0110
+) 9   1001
-------------
 15   1111 ->
01111
```

-6 + 9

```
  9   1001
-) 6   0110
-------------
  3   0011 ->
00011
```

6 + (- 9)

```
  9   1001
-) 6   0110
-------------
 - 3   0011 ->
10011
```

-6 + (-9)

```
  6   0110
+) 9   1001
-------------
-15   1111 ->
11111
```

Overflow 9 + 9 or (-9) + (-9)

```
  9   1001
+) 9   1001
-------------
(1)0010
```

overflow

ARITHMETIC ADDITION: SIGNED 2's COMPLEMENT

Add the two numbers, including their sign bit, and discard any carry out of leftmost (sign) bit

Example

```
   6   0   0110
+) 9   0   1001
  15   0   1111
```

```
   -6   1   1010
+) 9    0   1001
   3    0   0011
```

```
   6   0   0110
+) -9  1   0111
  -3   1   1101
```

```
   -9     1   0111
+) -9     1   0111
  -18   (1)0
  1110
```

$x'_{n-1}y'_{n-1}s_{n-1}$
$(c_{n-1} \oplus c_n)$

overflow

```
   9   0   1001
+) 9   0   1001
  18   1   0010
```

$x_{n-1}y_n s'_{n-1}$
$(c_{n-1} \oplus c_n)$

2 operands have the same sign and the result sign changes

$x_{n-1}y_{n-1}s'_{n-1} + x'_{n-1}y'_{n-1}s_{n-1}$

ARITHMETIC ADDITION: SIGNED 1's COMPLEMENT

Add the two numbers, including their sign bits.

- If there is a carry out of the most significant (sign) bit, the result is incremented by 1 and the carry is discarded.

Example

```
      6   0  0110
+)   -9   1  0110
-----------------
     -3   1  1100
```

end-around carry

```
     -6   1  1001
+)    9   0  1001
-----------------
  (1) 0(1)0010
          ->  1
+)
-----------------
      3   0  0011
```

not overflow $(c_{n-1} \oplus c_n) = 0$

```
     -9   1  0110
+)   -9   1  0110
-----------------
  (1)0   1  1100
          ->  1
+)
-----------------
      0  1101
```

overflow $(c_{n-1} \oplus c_n)$

```
      9   0  1001
+)    9   0  1001
-----------------
      1  (1)0010
```

COMPARISON OF REPRESENTATIONS

* Easiness of negative conversion

  S + M > 1's Complement > 2's Complement

* Hardware

  - S+M: Needs an adder and a subtractor for Addition
  - 1's and 2's Complement: Need only an adder

* Speed of Arithmetic

  2's Complement > 1's Complement(end-around C)

* Recognition of Zero

  2's Complement is fast

ARITHMETIC SUBTRACTION

Arithmetic Subtraction in 2's complement

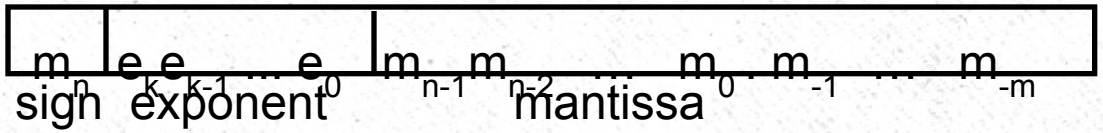
Take the complement of the subtrahend (including the sign bit) and add it to the minuend including the sign bits.

$$(\pm A) - (- B) = (\pm A) + B$$
$$(\pm A) - B = (\pm A) + (- B)$$

FLOATING POINT NUMBER REPRESENTATION

* The location of the fractional point is not fixed to a certain location
* The range of the representable numbers is wide

F = EM



- Mantissa
  Signed fixed point number, either an integer or a fractional number

- Exponent
  Designates the position of the radix point

Decimal Value

$$V(F) = V(M) * R^{V(E)}$$

M: Mantissa
E: Exponent
R: Radix

FLOATING POINT NUMBERS

Example

sign		sign	
0	.1234567	0	04
mantissa		exponent	

==> $+.1234567 \times 10^{+04}$

Note:

In Floating Point Number representation, only Mantissa(M) and Exponent(E) are explicitly represented. The Radix(R) and the position of the Radix Point are implied.

Example

A binary number +1001.11 in 16-bit floating point number representation (6-bit exponent and 10-bit fractional mantissa)

Sign	Exponent	Mantissa
0	0 00100	100111000
or		
0	0 00101	010011100

CHARACTERISTICS OF FLOATING POINT NUMBER REPRESENTATIONS

Normal Form

- There are many different floating point number representations of the same number
  --> Need for a unified representation in a given computer

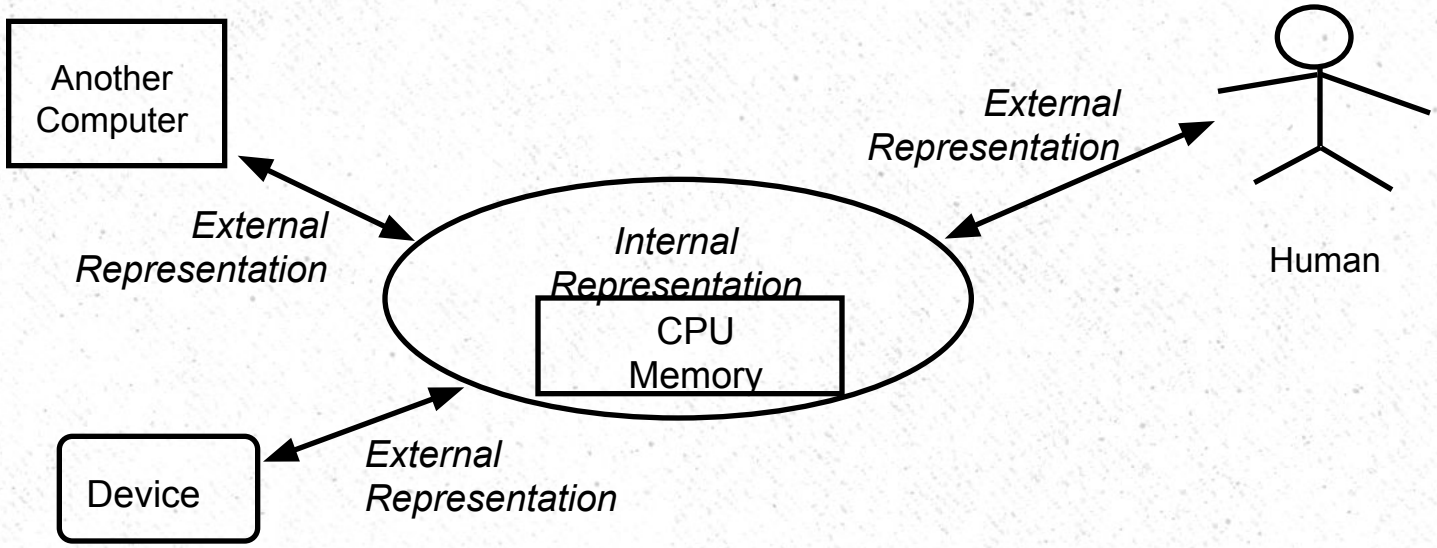
- *the most significant position of the mantissa contains a non-zero digit*

Representation of Zero

- Zero
  Mantissa = 0

- Real Zero
  Mantissa = 0
  Exponent
  = smallest representable number
  which is represented as
  00 ... 0
  <-- Easily identified by the hardware

INTERNAL REPRESENTATION AND EXTERNAL REPRESENTATION



External Representations	Internal Representations
- Presentability - Efficiency Communication Reliability - Easy to handle - BCD, ASCII, EBCDIC	- Efficiency Memory space Processing time - Easy to convert to external representation - Fixed and Floating points

EXTERNAL REPRESENTATION

Numbers

Most of numbers stored in the computer are eventually changed by some kinds of calculations

--> *Internal Representation* for calculation efficiency

--> Final results need to be converted to as *External Representation* for presentability

Alphabets, Symbols, and some Numbers

Elements of these information do not change in the course of processing

--> No needs for Internal Representation since they are not used for calculations

--> External Representation for processing and presentability

Example

Decimal Number: 4-bit Binary Code

BCD(Binary Coded Decimal)

Decimal	BCD Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

OTHER DECIMAL CODES

Decimal	BCD(8421)	2421	84-2-1	Excess-3
0	0000	0000	0000	0011
1	0001	0001	0111	0100
2	0010	0010	0110	0101
3	0011	0011	0101	0110
4	0100	0100	0100	0111
5	0101	1011	1011	1000
6	0110	1100	1010	1001
7	0111	1101	1001	1010
8	1000	1110	1000	1011
9	1001	1111	1111	1100

Note: 8,4,2,-2,1,-1 in this table is the weight associated with each bit position.

d3 d2 d1 d0: symbol in the codes

BCD: d3 x 8 + d2 x 4 + d1 x 2 + d0 x 1
==> 8421 code.
2421: d3 x 2 + d2 x 4 + d1 x 2 + d0 x 1
84-2-1: d3 x 8 + d2 x 4 + d1 x (-2) + d0 x (-1)
Excess-3: BCD + 3

BCD: It is difficult to obtain the 9's complement.
However, it is easily obtained with the other codes listed above.
==> Self-complementing codes

GRAY CODE

* Characterized by having their representations of the binary integers differ in only one digit between consecutive integers

* Useful in analog-digital conversion.

4-bit Gray codes

Decimal number	Gray g_3	g_2	g_1	g_0	Binary b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1

GRAY CODE - ANALYSIS

Letting $g_n g_{n-1} \ldots g_1 g_0$ be the (n+1)-bit Gray code for the binary number $b_n b_{n-1} \ldots b_1 b_0$

$$g_i = b_i \oplus b_{i+1} \quad , \quad 0 \le i \le n-1$$

$$g_n = b_n$$

and

$$b_{n-i} = g_n \oplus g_{n-1} \oplus \ldots \oplus g_{n-i}$$

$$b_n = g_n$$

Note:
The Gray code has a reflection property
- easy to construct a table without calculation,
- for any n: reflect case n-1 about a mirror at its bottom and prefix 0 and 1 to top and bottom halves, respectively

Reflection of Gray codes

ε	0	0 0	0 00
	1	0 1	0 01
		1 1	0 11
		1 0	0 10
			1 10
			1 11
			1 01
			1 00
			1 100
			1 101
			1 111
			1 010
			1 011
			1 001
			1 101
			1 000

0 000
0 001
0 011
0 010
0 110
0 111
0 101
0 100

CHARACTER REPRESENTATION ASCII

ASCII (American Standard Code for Information Interchange) Code

MSB (3 bits)

LSB (4 bits)	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	‘	P
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	I	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	I	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	m	n	~
F	SI	US	/	?	O	n	o	DEL

CONTROL CHARACTER REPRESENTAION (ACSII)

NUL	Null	DC1	Device Control 1
SOH	Start of Heading (CC)	DC2	Device Control 2
STX	Start of Text (CC)	DC3	Device Control 3
ETX	End of Text (CC)	DC4	Device Control 4
EOT	End of Transmission (CC)	NAK	Negative Acknowledge (CC)
ENQ	Enquiry (CC)	SYN	Synchronous Idle (CC)
ACK	Acknowledge (CC)	ETB	End of Transmission Block (CC)
BEL	Bell	CAN	Cancel
BS	Backspace (FE)	EM	End of Medium
HT	Horizontal Tab. (FE)	SUB	Substitute
LF	Line Feed (FE)	ESC	Escape
VT	Vertical Tab. (FE)	FS	File Separator (IS)
FF	Form Feed (FE)	GS	Group Separator (IS)
CR	Carriage Return (FE)	RS	Record Separator (IS)
SO	Shift Out	US	Unit Separator (IS)
SI	Shift In	DEL	Delete
DLE	Data Link Escape (CC)		

(CC) Communication Control
(FE) Format Effector
(IS) Information Separator

ERROR DETECTING CODES

Parity System

- Simplest method for error detection
- One *parity* bit attached to the information
- *Even Parity* and *Odd Parity*

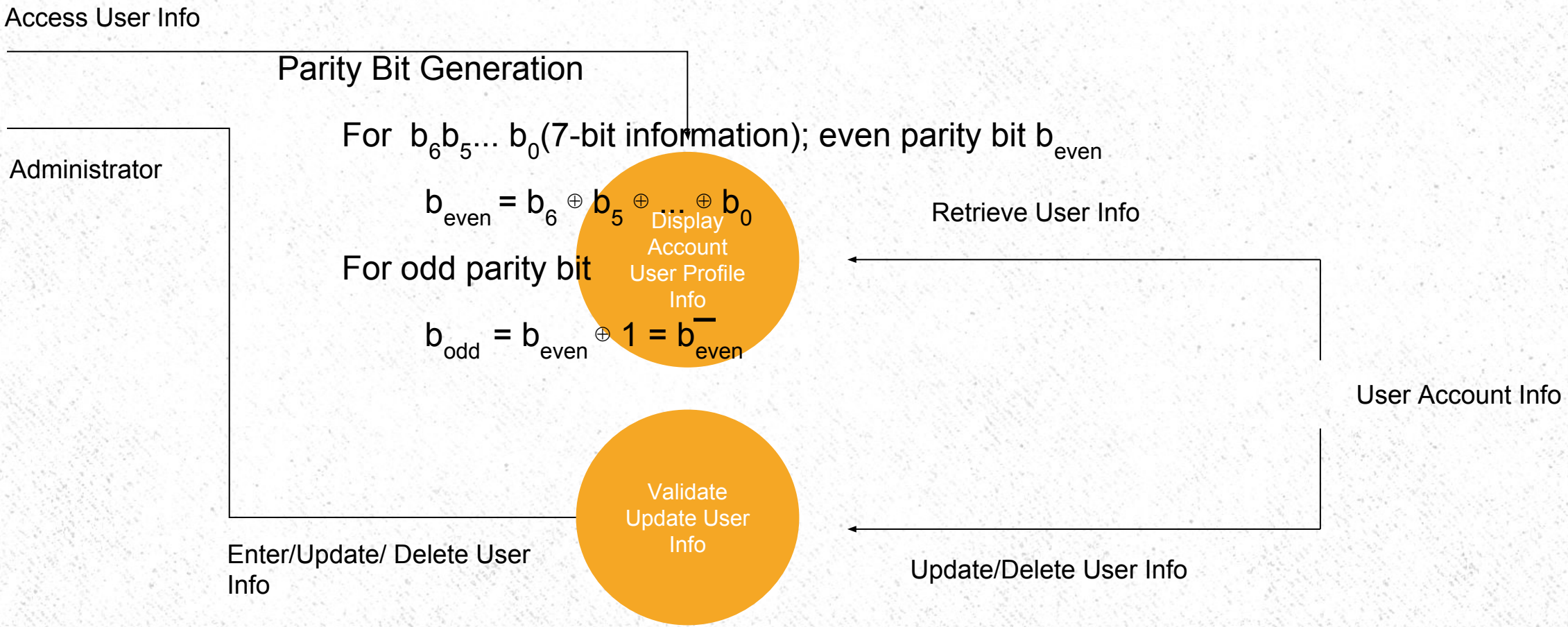
Even Parity
- One bit is attached to the information so that the total number of 1 bits is an even number

1011001 0
1010010 1

Odd Parity
- One bit is attached to the information so that the total number of 1 bits is an odd number

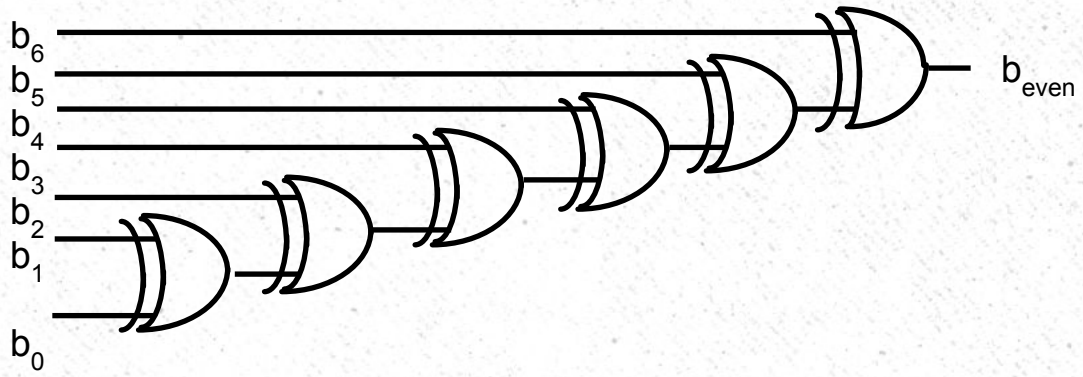
1011001 1
1010010 0

PARITY BIT GENERATION

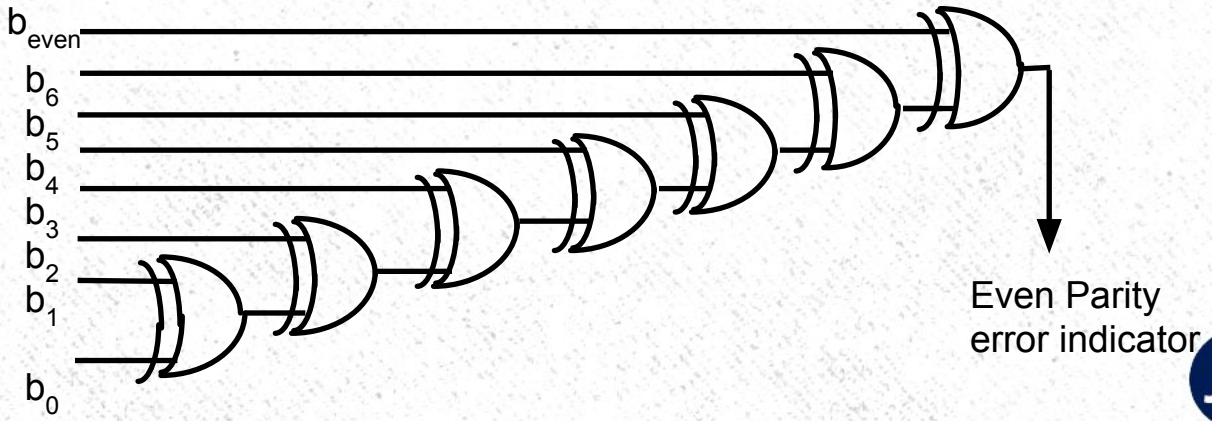


PARITY GENERATOR AND PARITY CHECKER

Parity Generator Circuit(even parity)



Parity Checker



REGISTER TRANSFER AND MICROOPERATIONS

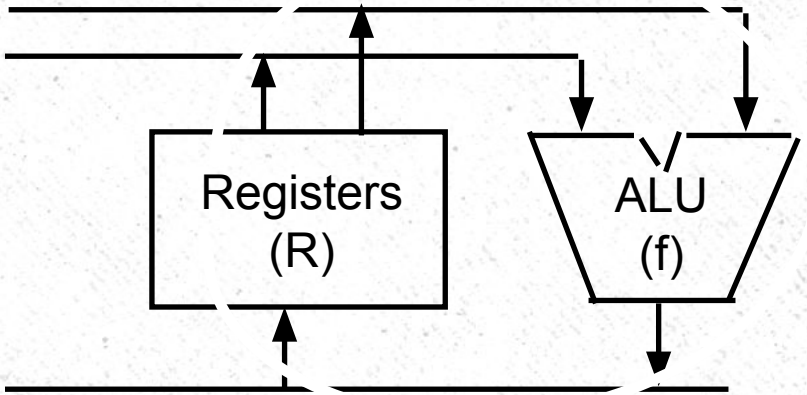
- Register Transfer Language
- Register Transfer
- Bus and Memory Transfers
- Arithmetic Microoperations
- Logic Microoperations
- Shift Microoperations
- Arithmetic Logic Shift Unit

MICROOPERATIONS (1)

- Microoperations refer to the basic operations performed by the control unit of a digital computer on data stored in registers or memory..
- The functions built into registers are examples of microoperations
  - simple arithmetic operations such as addition or subtraction,
  - logical operations such as AND, OR, and NOT, and
  - Shift: moves the bits of a binary word left or right by a specified number of positions, filling the vacated positions with zeros
  - rotate operations: moves the bits of a binary word left or right, but wraps the vacated bits around to the other side of the word. A left rotate moves the bits to the left, with the leftmost bit wrapping around to the rightmost position, while a right rotate moves the bits to the right, with the rightmost bit wrapping around to the leftmost position.

MICROOPERATION (2)

An elementary operation performed (during one clock pulse), on the information stored in one or more registers



1 clock cycle

$R \leftarrow f(R, R)$

f: shift, load, clear, increment, add, subtract, complement, and, or, xor, …

ORGANIZATION OF A DIGITAL SYSTEM

- Definition of the (internal) organization of a computer

  - Set of registers and their functions

  - Microoperations set

    Set of allowable microoperations provided by the organization of the computer

  - Control signals that initiate the sequence of microoperations (to perform the functions)

REGISTER TRANSFER LEVEL

- Viewing a computer, or any digital system, in this way is called the register transfer level

- This is because we're focusing on
  - The system's registers
  - The data transformations in them, and
  - The data transfers between them.

REGISTER TRANSFER LANGUAGE

- Rather than specifying a digital system in words, a specific notation is used, *register transfer language*
- For any function of the computer, the register transfer language can be used to describe the (sequence of) microoperations
- 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.

DESIGNATION OF REGISTERS

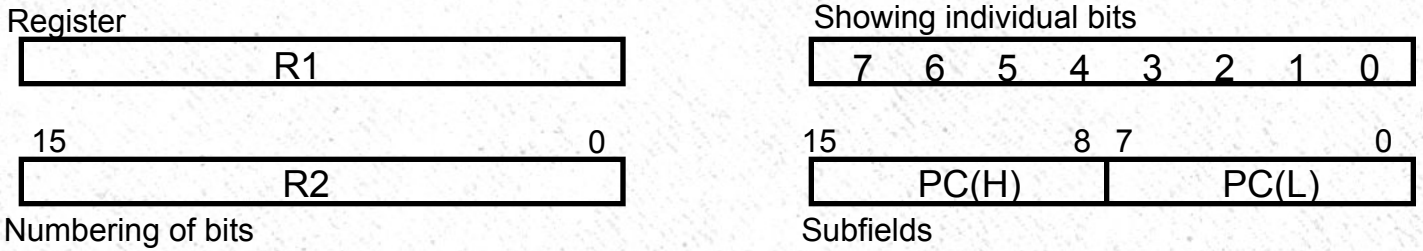
- Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR)
- Often the names indicate function:
  - MAR - memory address register
  - PC - program counter
  - IR - instruction register
- Registers and their contents can be viewed and represented in *various ways*
  - A register can be viewed as a single entity:

  - Registers may also be represented showing the bits of data they contain

DESIGNATION OF REGISTERS

- Designation of a register
  - a register
  - portion of a register
  - a bit of a register

- Common ways of drawing the block diagram of a register



REGISTER TRANSFER

- Copying the contents of one register to another is a register transfer
- A register transfer is indicated as

$$R2 \leftarrow R1$$

  - In this case the contents of register R2 are copied (loaded) into register R1
  - A simultaneous transfer of all bits from the source R1 to the destination register R2, during one clock pulse
  - Note that this is a non-destructive; i.e. the contents of R1 are not altered by copying (loading) them to R2

REGISTER TRANSFER

- A register transfer such as

  R3 ← R5

  Implies that the digital system has

  - the data lines from the source register (R5) to the destination register (R3)
  - Parallel load in the destination register (R3)
  - Control lines to perform the action

CONTROL FUNCTIONS

- Often actions need to only occur if a certain condition is true
- This is similar to an “if” statement in a programming language
- In digital systems, this is often done via a *control signal*, called a *control function*
  - If the signal is 1, the action takes place
- This is represented as:

P: R2 ← R1

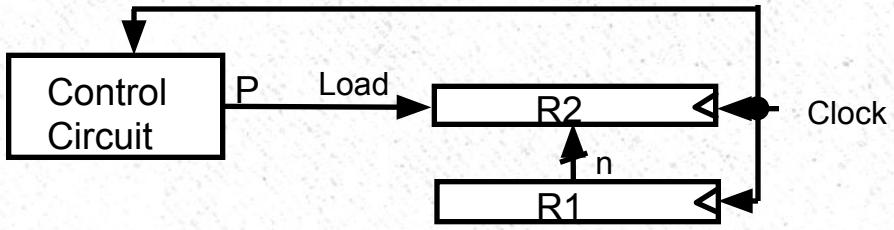
Which means “if P = 1, then load the contents of register R1 into register R2”, i.e., if (P = 1) then (R2 ← R1)

HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS

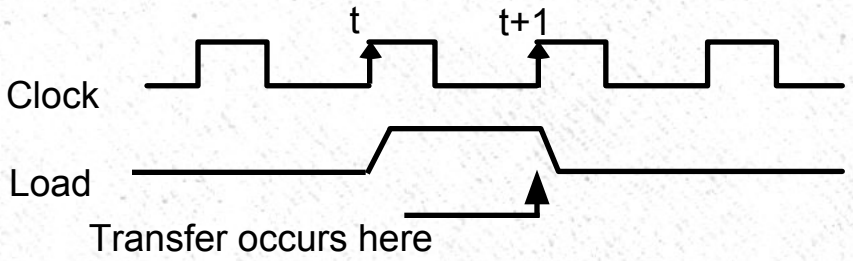
Implementation of controlled transfer

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

SIMULTANEOUS OPERATIONS

- If two or more operations are to occur simultaneously, they are separated with commas

  P: R3 ← R5 MAR ← IR

- Here, if the control function P = 1, load the contents of R5 into R3, and at the same time (clock), load the contents of register IR into register MAR

BASIC SYMBOLS FOR REGISTER TRANSFERS

Symbols	Description	Examples
Capital letters & numerals	Denotes a register	MAR, R2
Parentheses ()	Denotes a part of a register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Colon :	Denotes termination of control function	P:
Comma ,	Separates two micro-operations	A ← B, B ← A

CONNECTING REGISTRS

- In a digital system with many registers, it is impractical to have data and control lines to directly allow each register to be loaded with the contents of every possible other registers

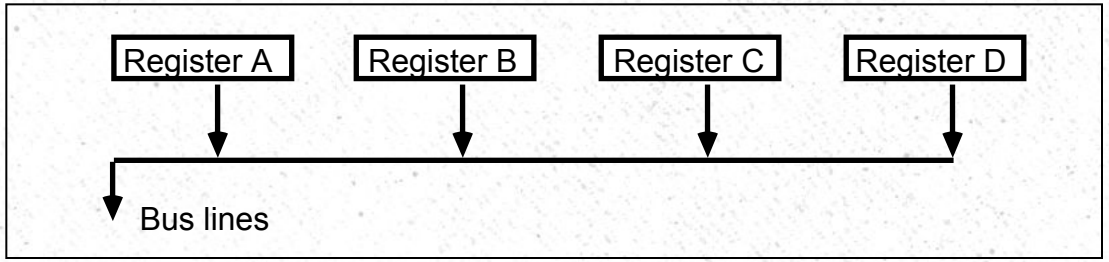
- To completely connect n registers n(n-1) lines
- $O(n^2)$ cost
  - This is not a realistic approach to use in a large digital system

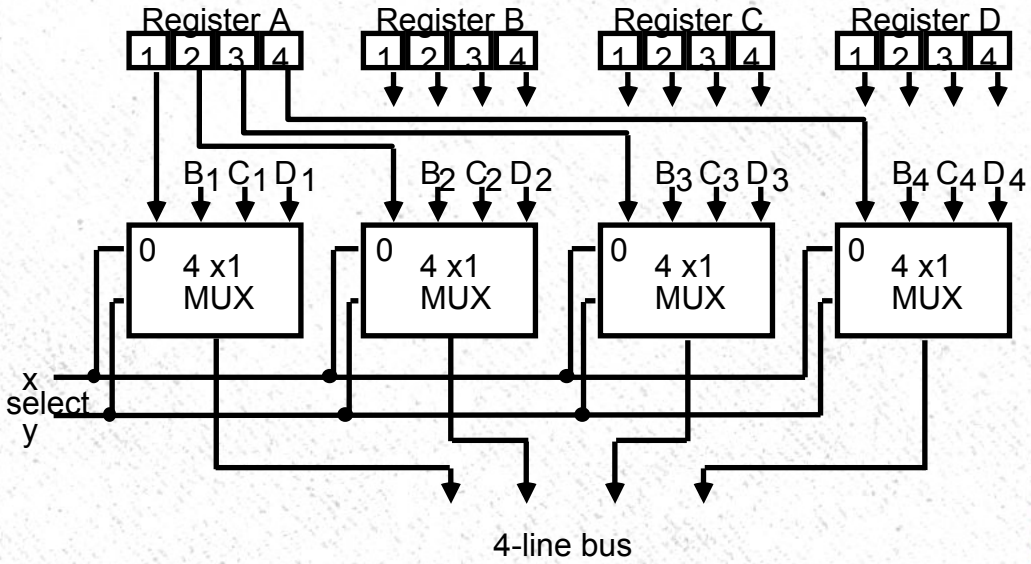
- Instead, take a different approach
- Have one centralized set of circuits for data transfer – the bus
- Have control circuits to select which register is the source, and which is the destination

BUS AND BUS TRANSFER

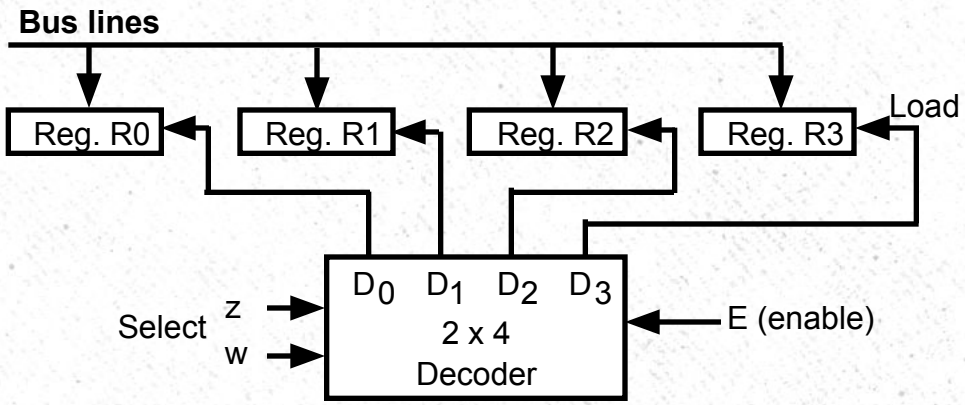
Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

From a register to bus: BUS ← R





TRANSFER FROM BUS TO A DESTINATION REGISTER



Three-State Bus Buffers

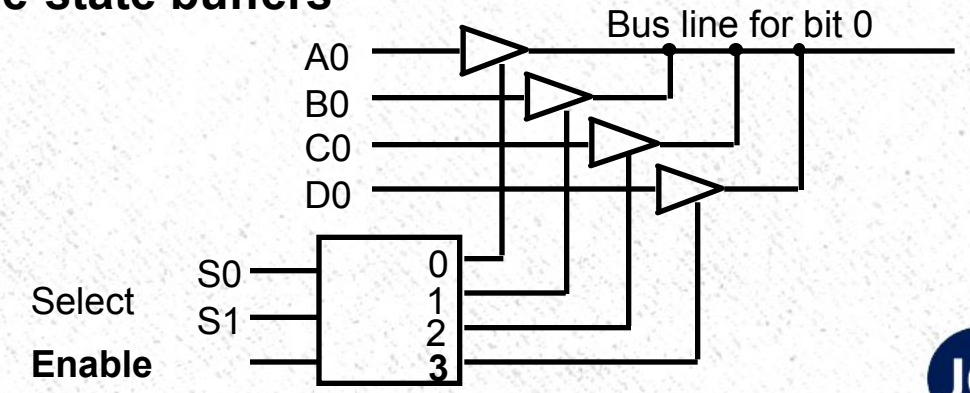
Normal input A

Control input C

Output Y=A if C=1

High-impedence if C=0

Bus line with three-state buffers



BUS TRANSFER IN RTL

- Depending on whether the bus is to be mentioned explicitly or not, register transfer can be indicated as either

R2 ← R1

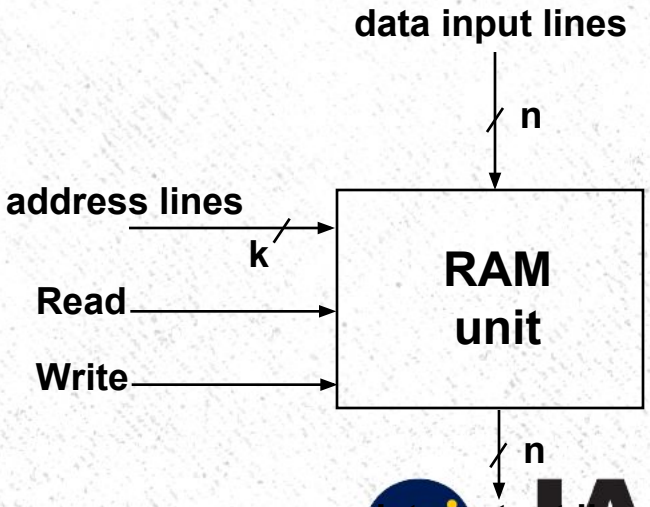
or

BUS ← R1, R2 ← BUS

- In the former case the bus is implicit, but in the latter, it is explicitly indicated

MEMORY (RAM)

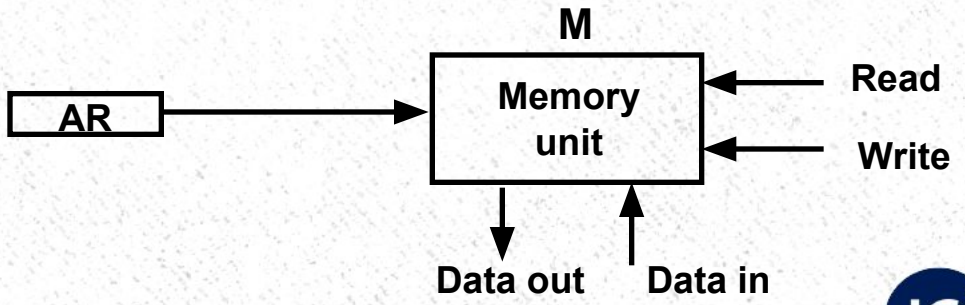
- Memory (RAM) can be thought as a sequential circuits containing some number of registers
- These registers hold the *words* of memory
- Each of the r registers is indicated by an *address*
- These addresses range from 0 to r-1
- Each register (word) can hold n bits of data
- Assume the RAM contains r = 2k 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 usually 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:

$$M[MAR] \leftarrow R1$$

- 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
ABUS ← 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

- Computer system microoperations are of four types:

  - Register transfer microoperations
  - Arithmetic microoperations
  - Logic microoperations
  - Shift microoperations

ARITHMETIC MICROOPERATIONS

- The basic arithmetic microoperations are
  - Addition
  - Subtraction
  - Increment
  - Decrement

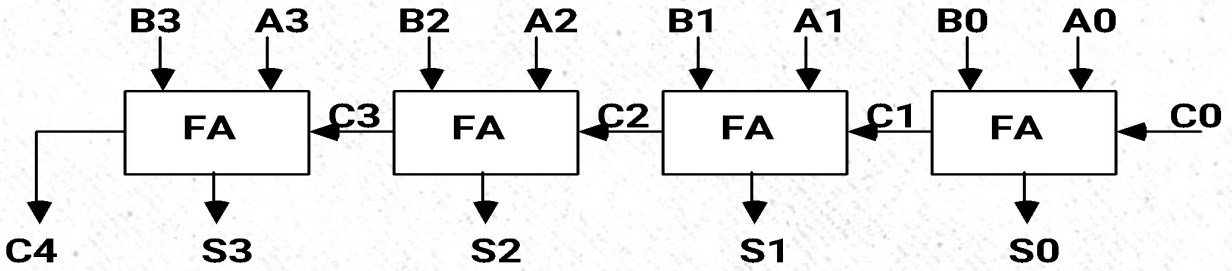
- The additional arithmetic microoperations are
  - Add with carry
  - Subtract with borrow
  - Transfer/Load
  - etc. …

Summary of Typical Arithmetic Micro-Operations

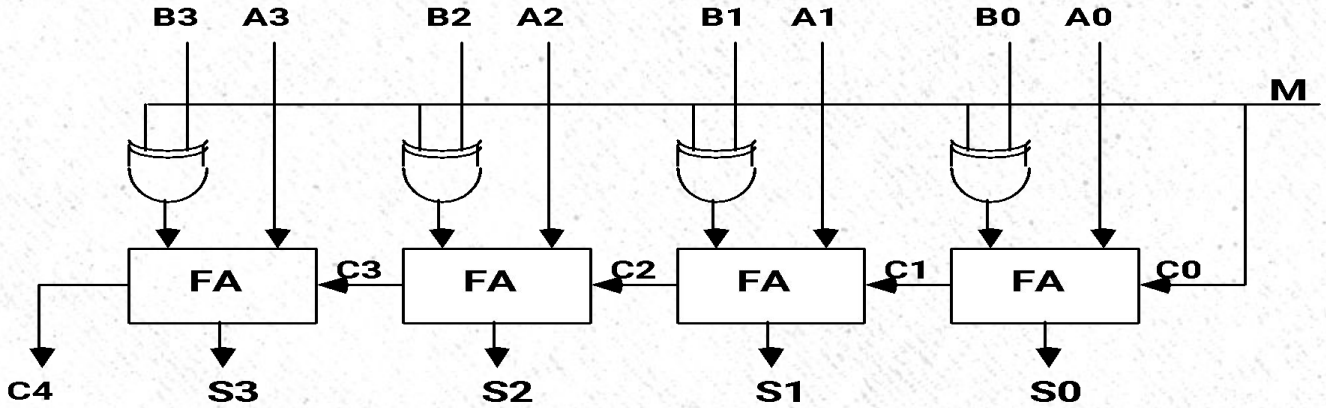
R3 ← R1 + R2	Contents of R1 plus R2 transferred to R3
R3 ← R1 - R2	Contents of R1 minus R2 transferred to R3
R2 ← R2’	Complement the contents of R2
R2 ← R2’+ 1	2's complement the contents of R2 (negate)
R3 ← R1 + R2’+ 1	subtraction
R1 ← R1 + 1	Increment
R1 ← R1 - 1	Decrement

BINARY ADDER / SUBTRACTOR / INCREMENTER

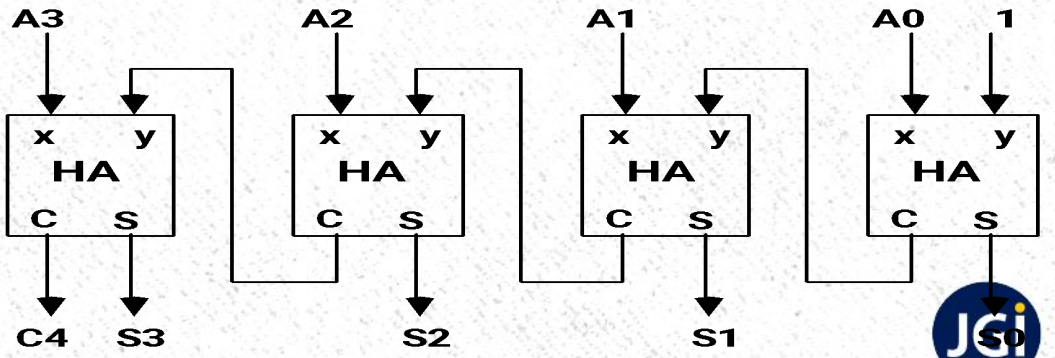
Binary Adder



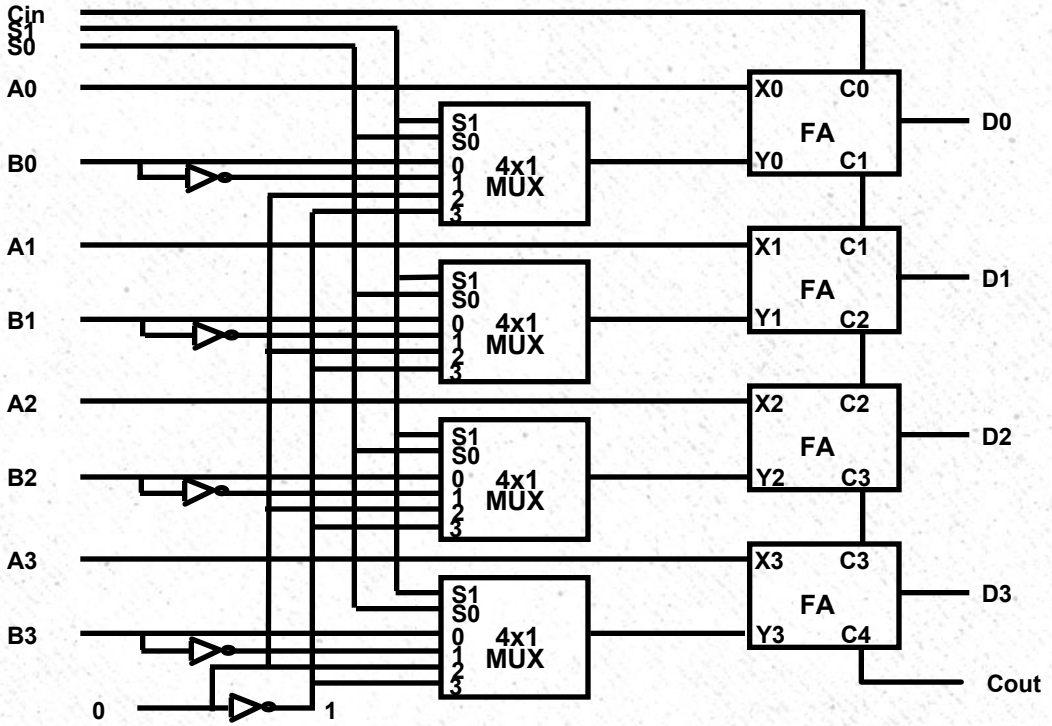
Binary Adder-Subtractor



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

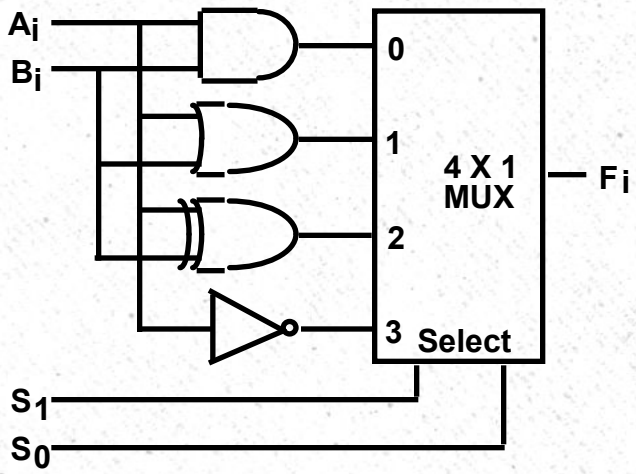
LIST OF LOGIC MICROOPERATIONS

- **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	0 0 1 1	*Boolean Function*	*Micro-Operations*	*Name*
y	0 1 0 1			
	0 0 0 0	F0 = 0	F ← 0	Clear
	0 0 0 1	F1 = xy	F ← A ∧ B	AND
	0 0 1 0	F2 = xy'	F ← A ∧ B’	
	0 0 1 1	F3 = x	F ← A	Transfer A
	0 1 0 0	F4 = x'y	F ← A’∧ B	
	0 1 0 1	F5 = y	F ← B	Transfer B
	0 1 1 0	F6 = x ⊕ y	F ← A ⊕ B	Exclusive-OR
	0 1 1 1	F7 = x + y	F ← A ∨ B	OR
	1 0 0 0	F8 = (x + y)'	F ← (A ∨ B)’	NOR
	1 0 0 1	F9 = (x ⊕ y)'	F ← (A ⊕ B)’	Exclusive-NOR
	1 0 1 0	F10 = y'	F ← B’	Complement B
	1 0 1 1	F11 = x + y'	F ← A ∨ B	
	1 1 0 0	F12 = x'	F ← A’	Complement A
	1 1 0 1	F13 = x' + y	F ← A’∨ B	
	1 1 1 0	F14 = (xy)'	F ← (A ∧ B)’	NAND
	1 1 1 1	F15 = 1	F ← 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 ∧ B	AND
0	1	F = A ∨ B	OR
1	0	F = A ⊕ B	XOR
1	1	F = A’	Complement

APPLICATIONS OF LOGIC MICROOPERATIONS

- Logic microoperations can be used to manipulate individual bits or a portions of a word in a register

- Consider the data in a register A. In another register, B, is bit data that will be used to modify the contents of A

  - Selective-set $A \leftarrow A + B$
  - Selective-complement $A \leftarrow A \oplus B$
  - Selective-clear $A \leftarrow A \cdot B'$
  - Mask (Delete) $A \leftarrow A \cdot B$
  - Clear $A \leftarrow A \oplus B$
  - Insert $A \leftarrow (A \cdot B) + C$
  - Compare $A \leftarrow A \oplus B$
  - . . .

SELECTIVE SET

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

$$\begin{array}{ll} 1\ 1\ 0\ 0 & A_t \\ 1\ 0\ 1\ 0 & B \\ \hline 1\ 1\ 1\ 0 & A_{t+1}\ (A \leftarrow A + B) \end{array}$$

- 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

SELECTIVE COMPLEMENT

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

$$\begin{array}{llll} 1 & 1 & 0 & 0 \quad A_t \\ 1 & 0 & 1 & 0 \quad B \\ \hline 0 & 1 & 1 & 0 \quad A_{t+1} \end{array} \quad (A \leftarrow A \oplus B)$$

- If a bit in B is set to 1, that same position in A gets complemented from its original value, otherwise it is unchanged

SELECTIVE CLEAR

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

$$\begin{array}{ll} 1\ 1\ 0\ 0 & A_t \\ 1\ 0\ 1\ 0 & B \\ \hline 0\ 1\ 0\ 0 & A_{t+1}\ (A \leftarrow A \cdot B') \end{array}$$

- If a bit in B is set to 1, that same position in A gets set to 0, otherwise it is unchanged

MASK OPERATION

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

$$\begin{array}{ll} 1\ 1\ 0\ 0 & A_t \\ 1\ 0\ 1\ 0 & B \\ \hline 1\ 0\ 0\ 0 & A_{t+1}\ (A \leftarrow A \cdot B) \end{array}$$

- If a bit in B is set to 0, that same position in A gets set to 0, otherwise it is unchanged

CLEAR OPERATION

- In a clear operation, if the bits in the same position in A and B are the same, they are cleared in A, otherwise they are set in A

1 1 0 0 A_t

1 0 1 0 B

0 1 ~~1 0 A_{t+1} $(A \leftarrow A \oplus B)$~~

INSERT OPERATION

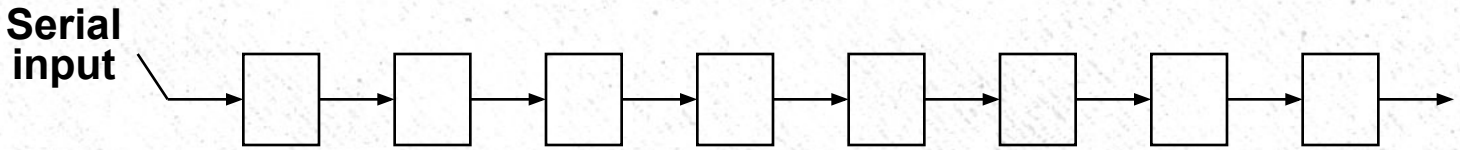
- An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged
- This is done as
  - A mask operation to clear the desired bit positions, followed by
  - An OR operation to introduce the new bits into the desired positions
  - Example
    - Suppose you wanted to introduce 1010 into the low order four bits of A: 1101 1000 1011 0001 A (Original)
      1101 1000 1011 1010 A (Desired)

-
```
1101 1000 1011 0001    A (Original)
1111 1111 1111 0000    Mask
1101 1000 1011 0000    A (Intermediate)
0000 0000 0000 1010    Added bits
1101 1000 1011 1010    A (Desired)
```

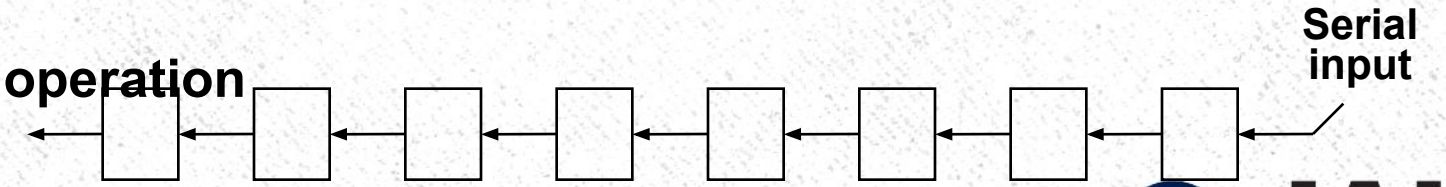
SHIFT MICROOPERATIONS

- 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**

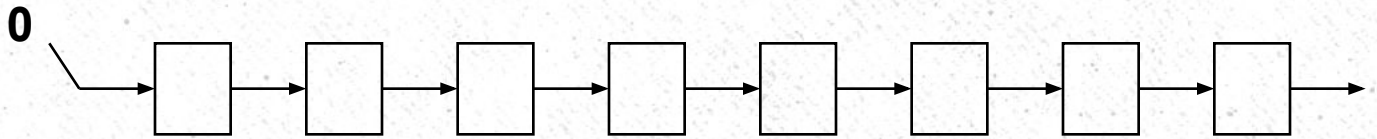


- **A left shift operation**

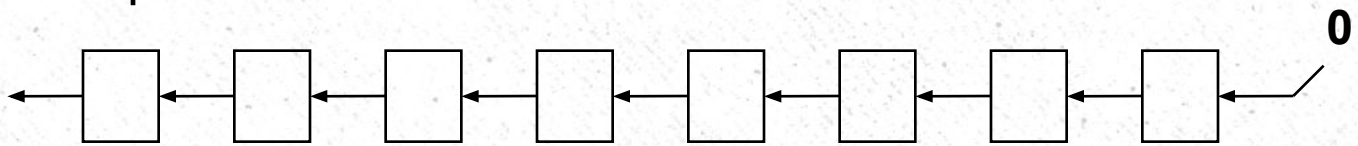


LOGICAL SHIFT

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



- A left logical shift operation:

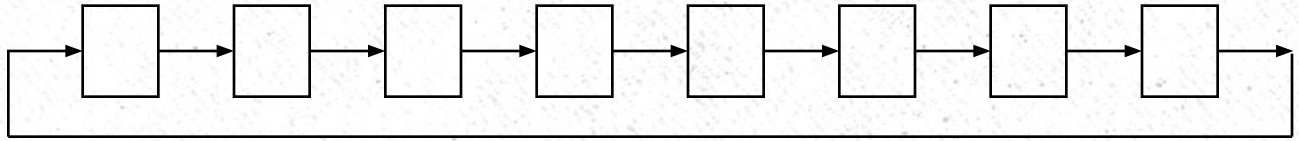


- In a Register Transfer Language, the following notation is used
  - *shl* for a logical shift left
  - *shr* for a logical shift right
  - Examples:
    - R2 ← *shr* R2
    - R3 ← *shl* R3

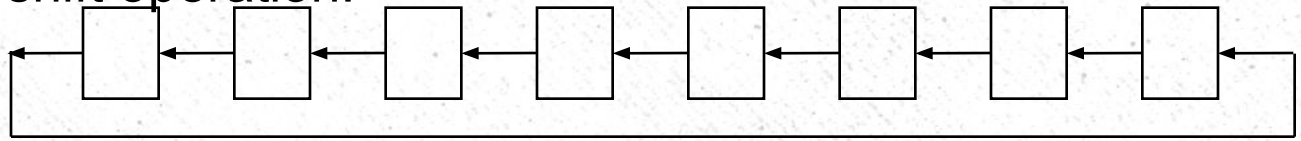
CIRCULAR SHIFT

- 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:

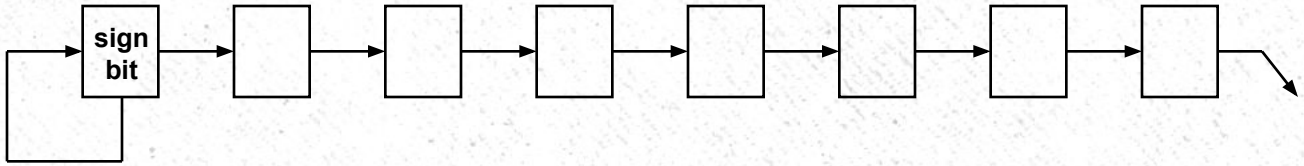


- In a RTL, the following notation is used
  - *cil* for a circular shift left
  - *cir* for a circular shift right
  - Examples:
    - R2 ← *cir* R2
    - R3 ← *cil* R3

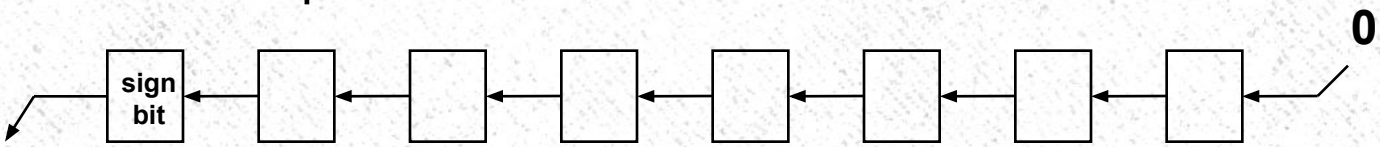
ARITHMETIC SHIFT

- An arithmetic shift is meant for signed binary numbers (integer)
- An arithmetic left shift multiplies a signed number by two
- An 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:

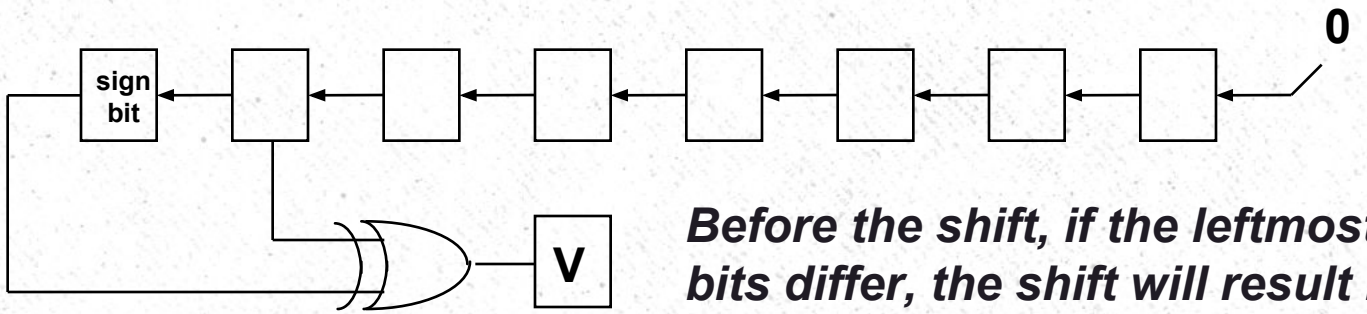


- A left arithmetic shift operation:



ARITHMETIC SHIFT

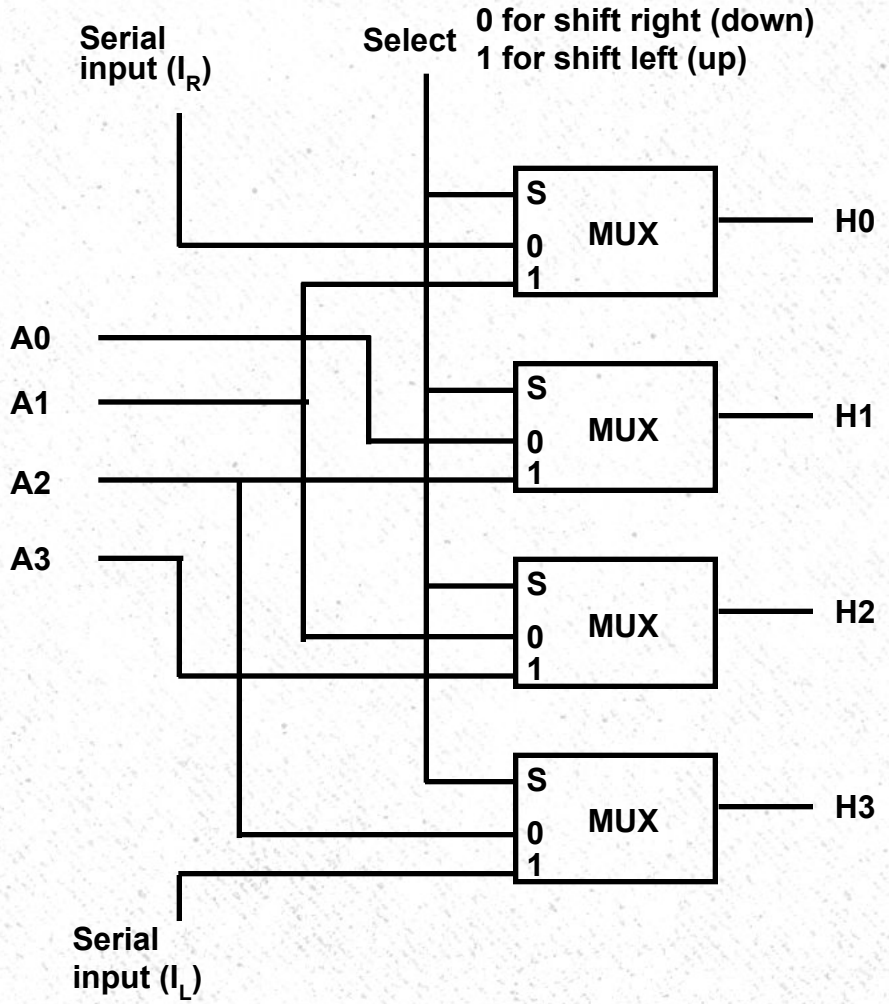
- An left arithmetic shift operation must be checked for the overflow



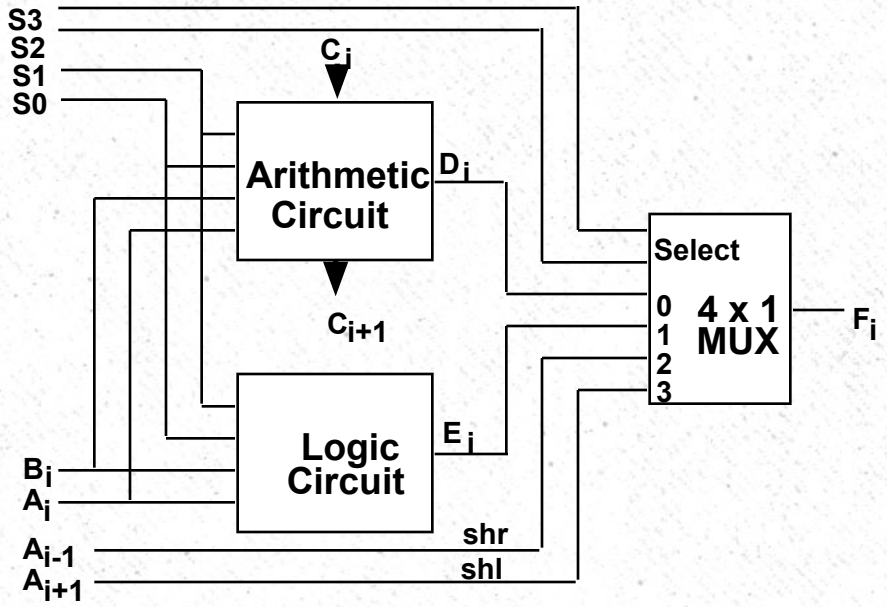
Before the shift, if the leftmost two bits differ, the shift will result in an overflow

- **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 ∧ B	AND
0	1	0	1	X	F = A ∨ B	OR
0	1	1	0	X	F = A ⊕ 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

Did You Know?

1. Data representation in COA is important because it affects the performance and efficiency of the computer system.
2. Efficient data representation can reduce the amount of memory required to store data, and can improve the speed at which data is processed and transmitted.
3. Additionally, understanding data representation is essential for writing efficient and error-free code in programming languages that interact with the computer's hardware.
4. Error detection codes are an important part of COA because they help ensure the integrity of data during transmission or storage.
5. By detecting errors early on, these codes can help prevent data corruption and improve the reliability and performance of computer systems.
6. A register is a group of flip-flops used to store binary information within a digital system. Registers are used to hold data temporarily, to enable processing, or to store data between different operations within a system. Registers are also used to hold addresses and control signals in microprocessors.
7. Micro-operations are combined to form complex operations and instructions that are executed by microprocessors. The design and implementation of registers and micro-operations are important factors in determining the performance and efficiency of digital systems and microprocessors.

Summary

Outcomes:

a. Discuss the theory Data Representation methods
b. Discuss the impact of data types and register data transfer operations on computing system performance.
c. Illustrate the different error detection codes

Terminal Questions

1) What are the different data types?
2) What are the different Arithmetic Micro operations, logic micro operations, Shift micro operations?

Reference Links

- https://www.geekforgeeks.org/computer organization

Reference Material:

- M. Moris Mano, “Computer Systems Architecture”, 4th Edition, Pearson/PHI, ISBN:10:0131755633

Thank you