College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Introduction

Course Instructor: Mr Mohammad Wakil Ahmad

Office: G75-1

Email: mwakilahmed@ksu.edu.sa

Main Component

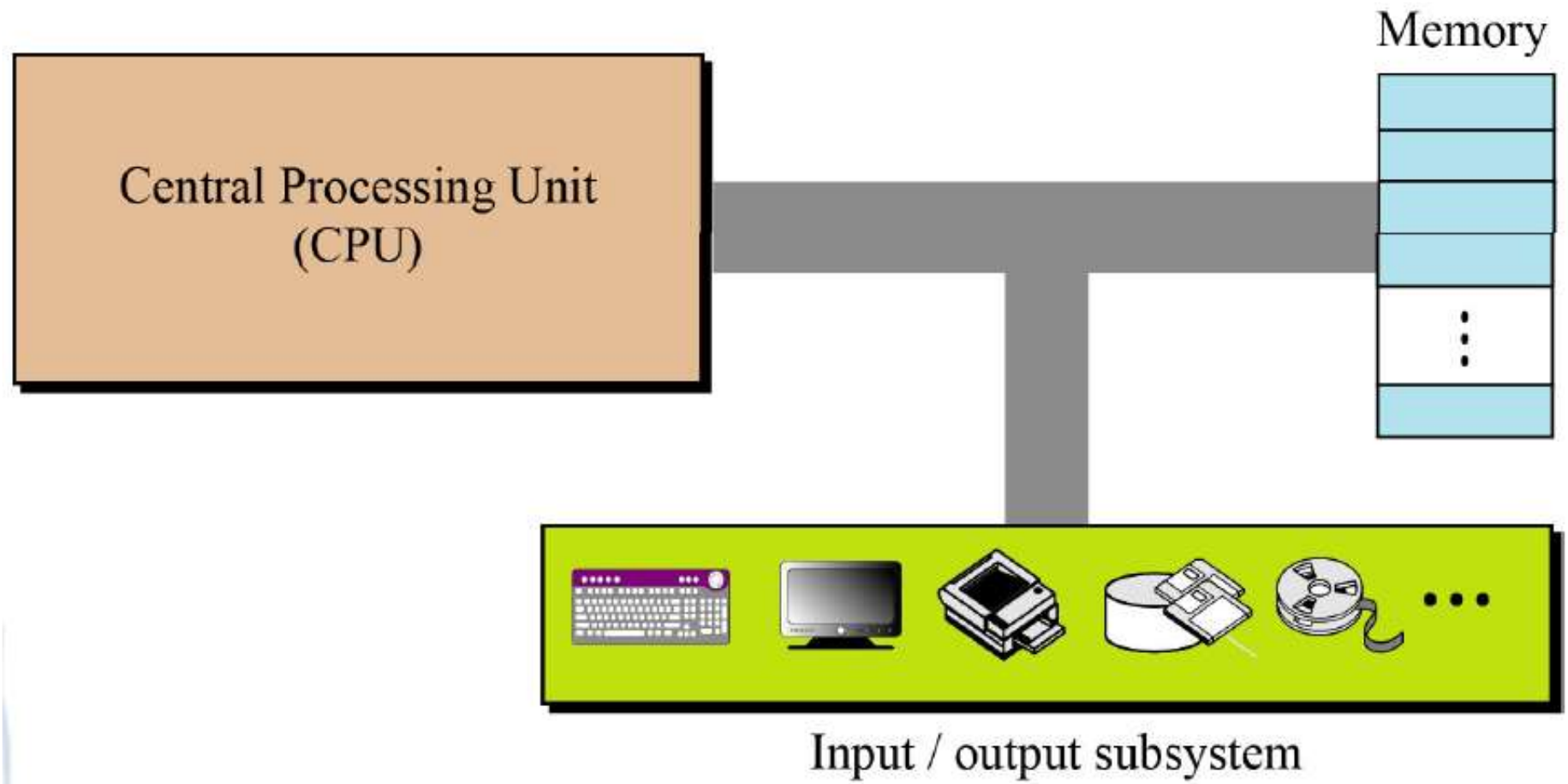
https://www.youtube.com/watch?v=yRmPTbGBqVI

Integrated Circuit (IC) Fabrication

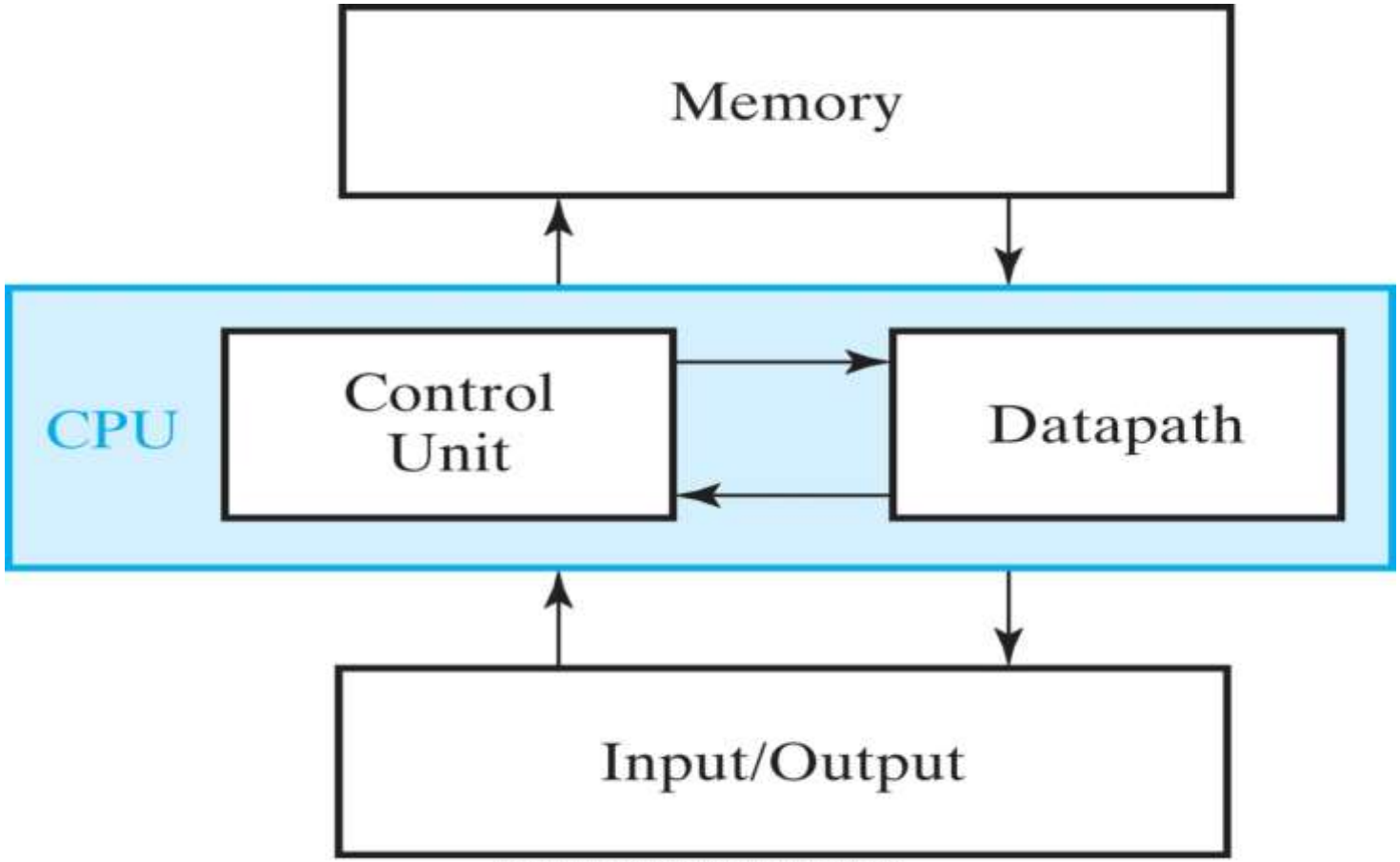
https://www.youtube.com/watch?v=aWVywhzuHnQ

https://www.youtube.com/watch?v=qm67wbB5GmI

Computer Hardware (Subsystems)

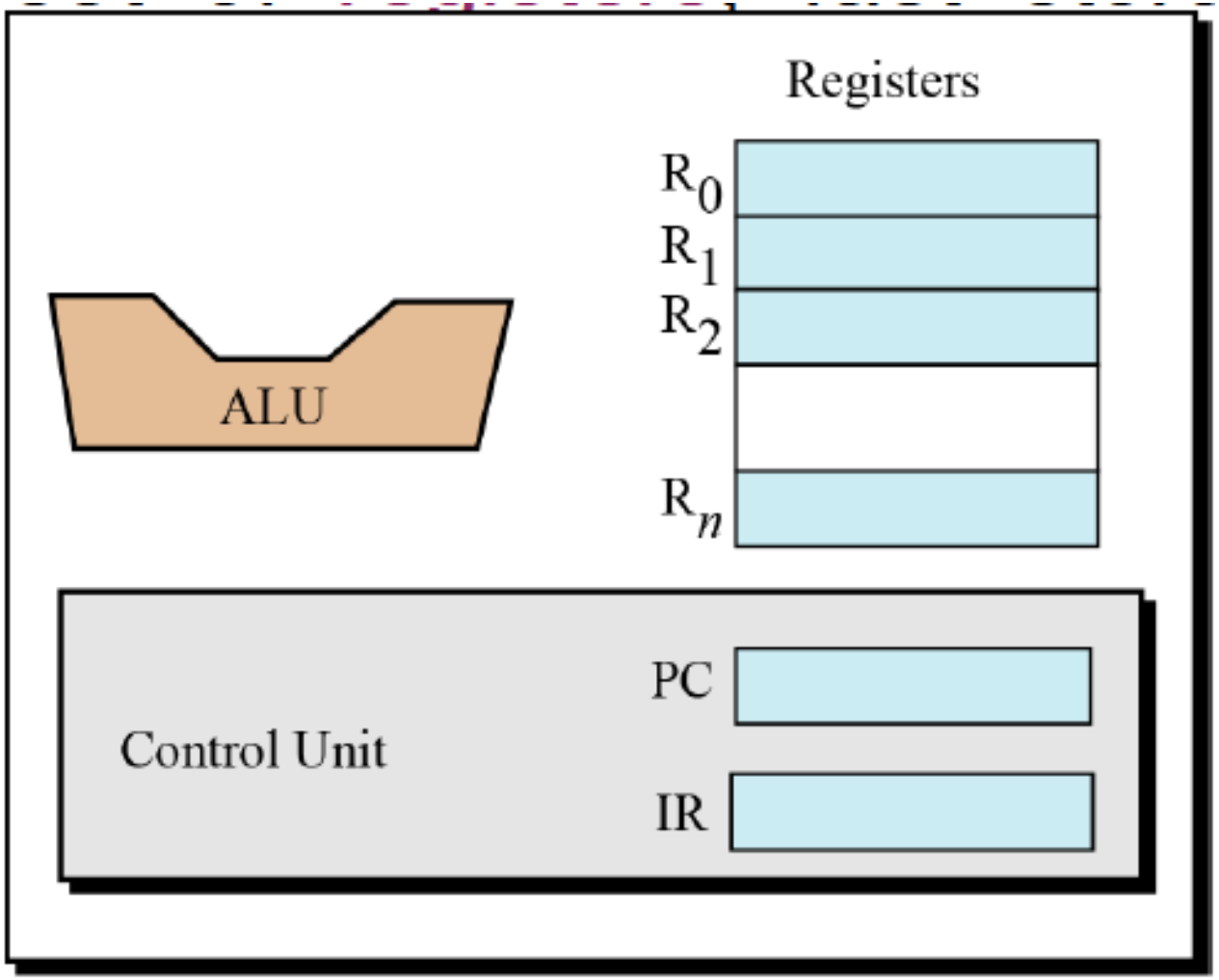


A Digital Computer

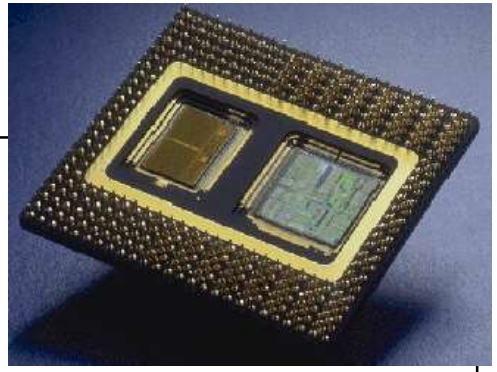


Copyright ©2016 Pearson Education, All Rights Reserved

Central Processing Unit (CPU)



History of Intel x86 processors



- 4004: 4-bit processor; 8-bit processors 8080, 8085.
- 8086: 16-bit processors, introduced in 1979: 20-bit address bus (AB), 16-bit data bus (DB), 5/8/10 MHz.
- 80186: a faster version of the 8086. Also 20-bit AB and 16-bit DB. Never widely used in computer systems.
- 80286: introduced in 1982, It has a 24-bit AB, which implies 16MB of memory address space.
- 80386: Intel introduced its first 32-bit processor in the 1985. It has 32-bit AB and 32-bit DB. It follows their 32-bit architecture known as IA-32.
- Pentium: was introduced in 1993 (20th anniversary). it uses a 64-bit wide DB, 60/66 MHz.
- Itanium: 64-bit processor, released in 2001; formerly called IA-64, the Itanium uses a 64-bit AB to provide substantially larger address space. Its DB is 128 bits wide.

CSC220 - Computer Organization (2-2-1) - Required Course

Pre-requisites: MATH 151 - Discrete Mathematics

Topics

- Logic gates, Logic function, K-Map simplification
- Combinational circuits: Multiplexer, De-multiplexer, Decoder, Arithmetic circuits
- Sequential circuits: register, binary counters, and memory organization
- Data Representation: binary number systems, signed number systems, use of 1's and 2's complements in addition and subtraction
- Arithmetic and Logic Unit design, Central Processing Unit design
- Register, Register Transfer Language, and micro-operations instruction set
- CPU design and programming

CSC 220: Course Learning Outcomes

Upon completing CSC220, students should have the following capabilities:

1. Understanding and simplification of a digital logic function
2. To design combinational circuits such as Adder, Multiplexer, Decoder
3. Identifying necessary logic gates to design different components (e.g. registers, ALU, RAM, Control Unit) of a computer system
4. To design of a basic Arithmetic and Logic Unit (ALU)
5. Identifying appropriate combinational and sequential circuits for Datapath and Control Unit design
6. Understanding of the instruction set architecture
7. Understanding of the register transfer language (RTL) and ability to write simple assembly language program

Text Books

1. M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, 5th (Global) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

2. Morris Mano, **Computer System Architecture** - 3rd Edition, Publisher: McGraw Hill

Class Note based on the main text (Available at the LMS)

Assessment Methods:

Final Exam:	40%
Midterm 1:	20%
Midterm 2:	20%
Lab Projects:	15%
Homeworks:	5%

Important Dates

Last Date to Drop Course:

Midterm 1: Decided by college

Midterm 2: Decided by college

Homework-1 Due:

Homework-2 Due:

Final Exam: Decided by college

CS 220: Attendance Rules

1. Absent if you come after 15 minutes

2. Can't attend final exam if absence rate is more than 25%

3. No medical certificate will be considered for attendance

Week	Lecture 1	Lecture 2	Tutorial	Exams	Homerwork	Remarks
#01 (2 Sep)	Orientation & Introduction					
#02 (Unit-1)	Number System	Number System	Tutorial-1			
#03 (Unit-2)	Logic Gates and Functions	Logic Gates and Functions	Tutorial-2			
#04 (Unit-3)	Simplification (K-map)	Simplification (K-map)	Tutorial-3			
#05 (Unit-4)	Signed Number Representation	Signed Number Representation	Tutorial-4		HW 1	
#06 (Unit-5)	Combinational circuit-1	Combinational circuit-1	Tutorial-5			
#07 (Unit-6)	Combinational circuit-2	Combinational circuit-2	Tutorial-6			
#08 (Unit-7)	Sequential Circuit	Sequential Circuit	Tutorial-7			
#09 (Unit-8)	Registers and RTL	Registers and RTL	Tutorial-8			
#10 (Unit-9)	Counters-RAM	Counters-RAM	Tutorial-9		HW 2	
#11 (Unit-10)	ALU Design	ALU Design	Tutorial-10			
#12 (Unit-11)	Datapath Design	Datapath Design	Tutorial-11			
#13 (Unit-12)	CPU design & Programming	CPU design & Programming	Tutorial-12			
# 14, 15		Revision				
#16 ()	Preparatory Courses Final Exams					
#17 ()	Final Exams					
#18 ()						

College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit1
Number Systems

Unit1: Number Systems

Overview

- Common Number Systems
  - Decimal
  - Binary
  - Octal
  - Hexadecimal
- Conversion Among Bases
- Binary Coded Decimal (BCD)

Chapter-1

M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

Decimal review

- Decimal numbers consist of digits from 0 to 9, each with a weight.

1	6	2	.	3	7	5	digits
100	10	1		1/10	1/100	1/1000	weights

- Notice that the weights are all powers of the base, which is 10.

1	6	2	.	3	7	5	digits
10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}	weights

- To find the decimal value of a number, you can multiply each digit by its weight and sum the products:

$$(1\times10^2) + (6\times10^1) + (2\times10^0) + (3\times10^{-1}) + (7\times10^{-2}) + (5\times10^{-3}) = 162.375$$

Common Number Systems

System	Base	Symbols	Used by humans?	Used in computers?
Decimal	10	0, 1, … 9	Yes	No
Binary	2	0, 1	No	Yes
Octal	8	0, 1, … 7	No	No
Hexa-decimal	16	0, 1, … 9, A, B, … F	No	No

Quantities/Counting (1 of 3)

Decimal	Binary	Octal	Hexa-decimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7

Quantities/Counting (2 of 3)

Decimal	Binary	Octal	Hexa-decimal
8	1000	10	8
9	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

Quantities/Counting (3 of 3)

Decimal	Binary	Octal	Hexa-decimal
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17

Etc.

Binary numbers

- Binary, or base 2, numbers consist of only the digits 0 and 1. The weights are now powers of 2.
- For example, consider the binary number 1101.01:

1	1	0	1	.	0	1	binary digits, or bits
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	weights in decimal

- The decimal value of 1101.01 is computed just like before:

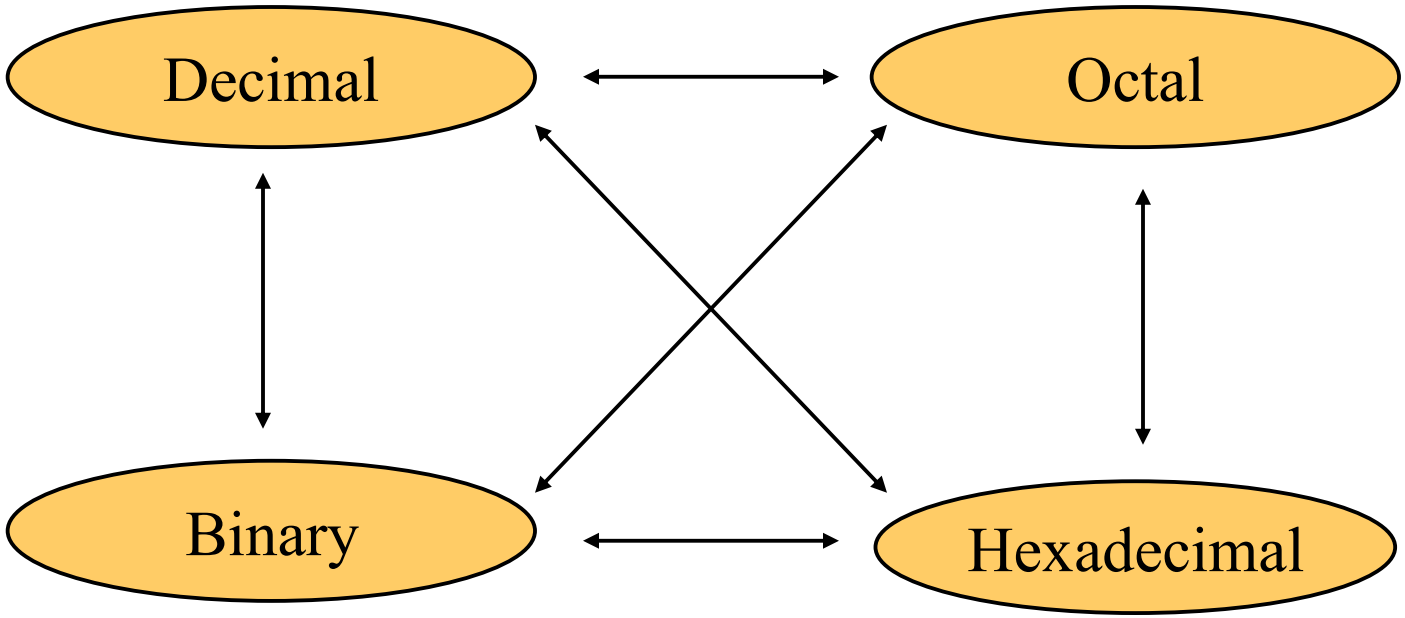
$$(1\times2^3) + (1\times2^2) + (0\times2^1) + (1\times2^0) + (0\times2^{-1}) + (1\times2^{-2}) =$$
$$8 + 4 + 0 + 1 + 0 + 0.25 = 13.25$$

Some powers of 2

$2^0 = 1$	$2^4 = 16$	$2^8 = 256$
$2^1 = 2$	$2^5 = 32$	$2^9 = 512$
$2^2 = 4$	$2^6 = 64$	$2^{10} = 1024$
$2^3 = 8$	$2^7 = 128$	

Conversion Among Bases

- The possibilities:

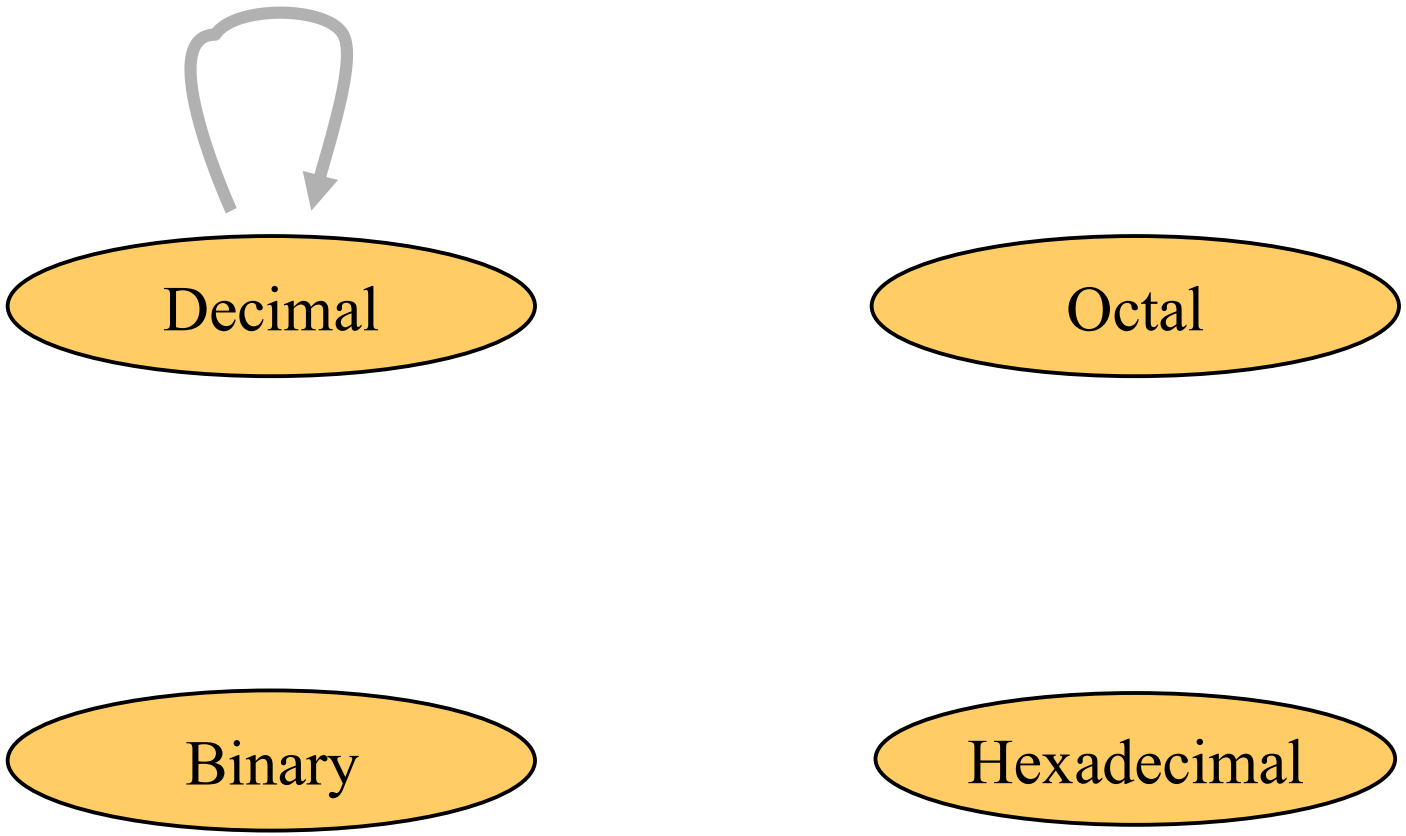


Quick Example

$$25_{10} = 11001_2 = 31_8 = 19_{16}$$

Base

Decimal to Decimal (just for fun)

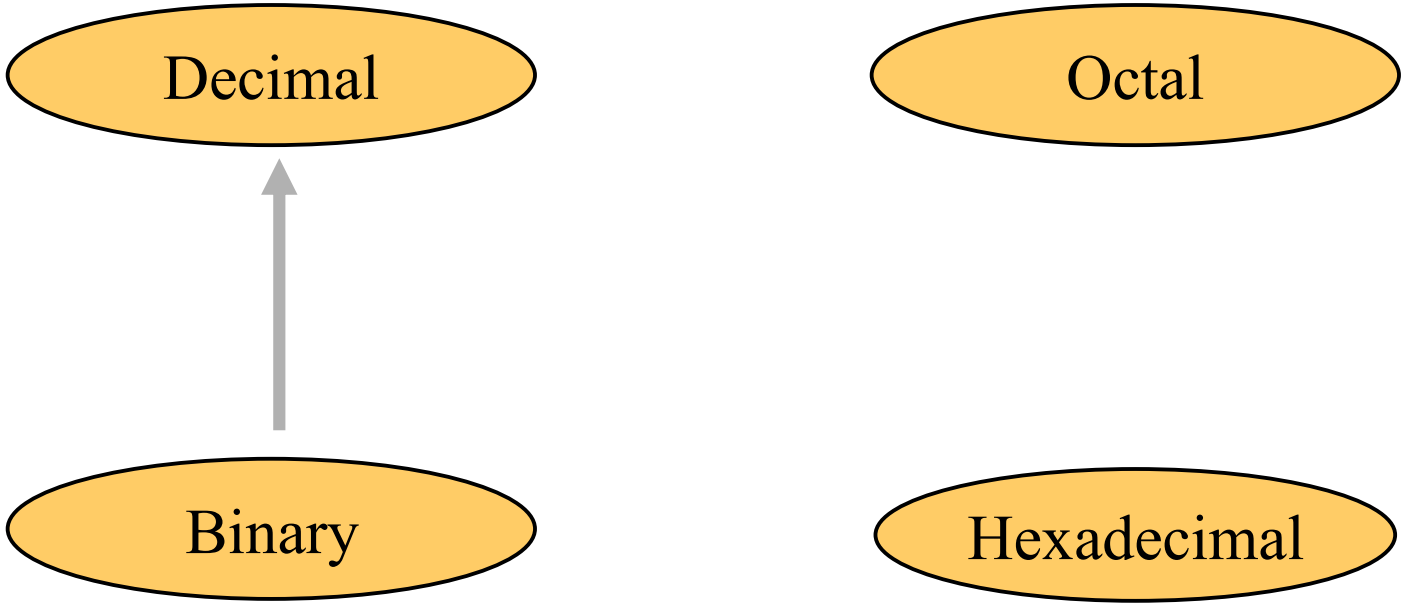


Weight

$$125_{10} \Rightarrow \quad \begin{array}{lcl} 5 \times 10^0 & = & 5 \\ 2 \times 10^1 & = & 20 \\ 1 \times 10^2 & = & 100 \\ \hline & & 125 \end{array}$$

Base

Binary to Decimal



Binary to Decimal

- Technique
  - Multiply each bit by 2^n, where n is the "weight" of the bit
  - The weight is the position of the bit, starting from 0 on the right
  - Add the results

Example

Bit “0”

101011_2 =>

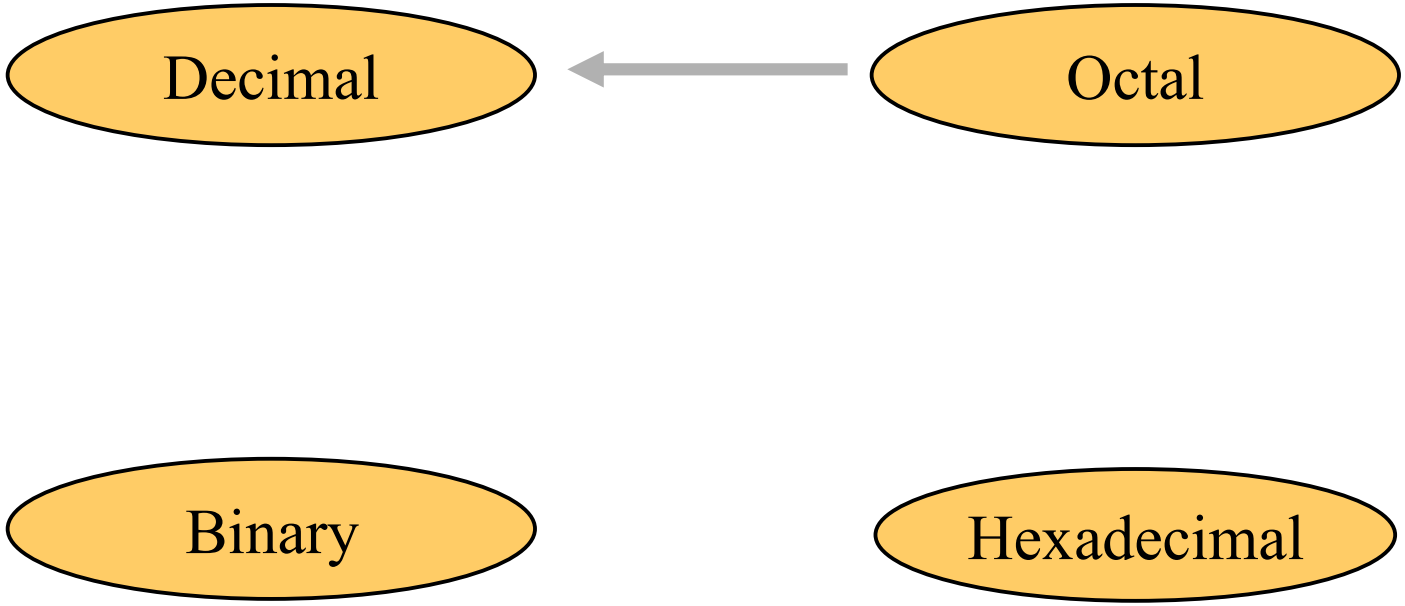
$$
\begin{array}{lcl}
1 \times 2^0 & = & 1 \\
1 \times 2^1 & = & 2 \\
0 \times 2^2 & = & 0 \\
1 \times 2^3 & = & 8 \\
0 \times 2^4 & = & 0 \\
1 \times 2^5 & = & 32 \\
\hline
 & & 43_{10}
\end{array}
$$

Converting Binary to Decimal

What is the decimal equivalent of the binary number 1101110?

$$
\begin{aligned}
& 1 \times 2^6 = 1 \times 64 = 64 \\
+\ & 1 \times 2^5 = 1 \times 32 = 32 \\
+\ & 0 \times 2^4 = 0 \times 16 = 0 \\
+\ & 1 \times 2^3 = 1 \times 8 = 8 \\
+\ & 1 \times 2^2 = 1 \times 4 = 4 \\
+\ & 1 \times 2^1 = 1 \times 2 = 2 \\
+\ & 0 \times 2^0 = 0 \times 1 = 0 \\
& \qquad\qquad\qquad = 110 \text{ in base } 10
\end{aligned}
$$

Octal to Decimal



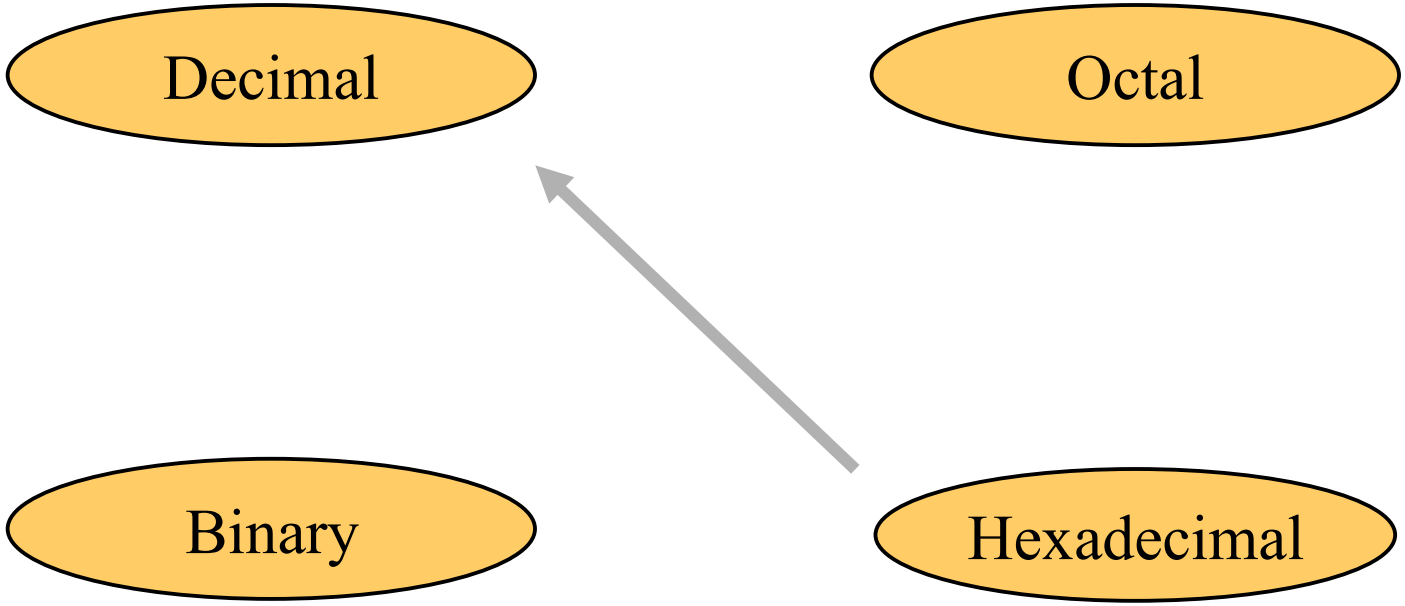
Octal to Decimal

- Technique
  - Multiply each bit by 8^n, where n is the "weight" of the bit
  - The weight is the position of the bit, starting from 0 on the right
  - Add the results

Example

$$
\begin{array}{lrcl}
724_8 \Rightarrow & 4 \times 8^0 & = & 4 \\
 & 2 \times 8^1 & = & 16 \\
 & 7 \times 8^2 & = & 448 \\
\hline
 & & & 468_{10}
\end{array}
$$

Hexadecimal to Decimal



Hexadecimal to Decimal

- Technique
  - Multiply each bit by 16^n, where n is the "weight" of the bit
  - The weight is the position of the bit, starting from 0 on the right
  - Add the results

Base 16 is useful too

- The hexadecimal system uses 16 digits:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

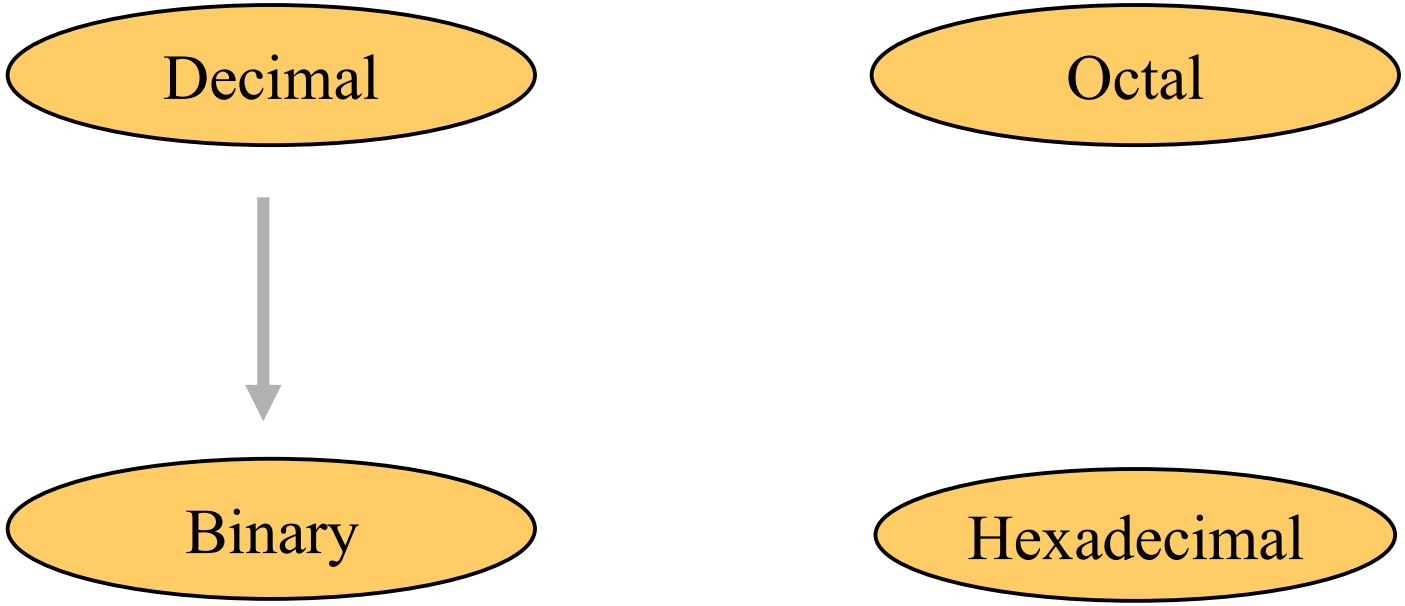
- Hexadecimal is useful as a shorthand for binary numbers.
  - Since 16 = 2^4, one hex digit is equivalent to four bits (including leading 0s).
  - It's often easier to work with numbers like "B4" instead of "10110100".
- Hex shows up in many different contexts.
  - IP addresses, such as "80.AE.05.27".
  - RGB color triplets, like "C0C0FF".
- You can convert between base 10 and base 16 using the same method as for converting from decimal to binary.

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Example

$$
\begin{array}{lrcrcrcr}
ABC_{16} \Rightarrow & C & \times & 16^0 & = & 12 & \times & 1 & = & 12 \\
 & B & \times & 16^1 & = & 11 & \times & 16 & = & 176 \\
 & A & \times & 16^2 & = & 10 & \times & 256 & = & 2560 \\
\hline
 & & & & & & & & & 2748_{10}
\end{array}
$$

Decimal to Binary



Decimal to Binary

- Technique
  - Divide by two, keep track of the remainder
  - First remainder is bit 0 (LSB, least-significant bit)
  - Second remainder is bit 1
  - Etc.

Converting decimal to binary

- To convert a decimal integer into binary, keep dividing by two until the quotient is 0. Then collect the remainders in reverse order.
- To convert a decimal fraction into binary, keep multiplying the fractional part by two until it becomes 0. Collect the integers in forward order.
- An example will make it all clear. Let's convert 162.375 to binary.

$$
\begin{aligned}
162 / 2 &= 81 \quad \text{rem } 0 \\
81 / 2 &= 40 \quad \text{rem } 1 \\
40 / 2 &= 20 \quad \text{rem } 0 \\
20 / 2 &= 10 \quad \text{rem } 0 \\
10 / 2 &= 5 \quad \text{rem } 0 \\
5 / 2 &= 2 \quad \text{rem } 1 \\
2 / 2 &= 1 \quad \text{rem } 0 \\
1 / 2 &= 0 \quad \text{rem } 1
\end{aligned}
$$

$$
\begin{aligned}
0.375 \times 2 &= 0.750 \\
0.750 \times 2 &= 1.500 \\
0.500 \times 2 &= 1.000
\end{aligned}
$$

- So $162.375_{10} = 10100010.011_2$

Why does this work?

- This same idea works for converting from decimal to any other base.
- Think about "converting" 162 from decimal to decimal:

$$
\begin{aligned}
162 \,/\, 10 &= 16 \quad \text{rem } 2 \\
16 \,/\, 10 &= 1 \quad \text{rem } 6 \\
1 \,/\, 10 &= 0 \quad \text{rem } 1
\end{aligned}
$$

- After each division, the remainder contains the rightmost digit of the dividend, while the quotient holds the remaining digits.
- Similarly when converting fractions, each multiplication strips off the leftmost digit as the integer result, leaving the remaining digits in the fractional part.

$$
\begin{aligned}
0.375 \times 10 &= 3.750 \\
0.750 \times 10 &= 7.500 \\
0.500 \times 10 &= 5.000
\end{aligned}
$$

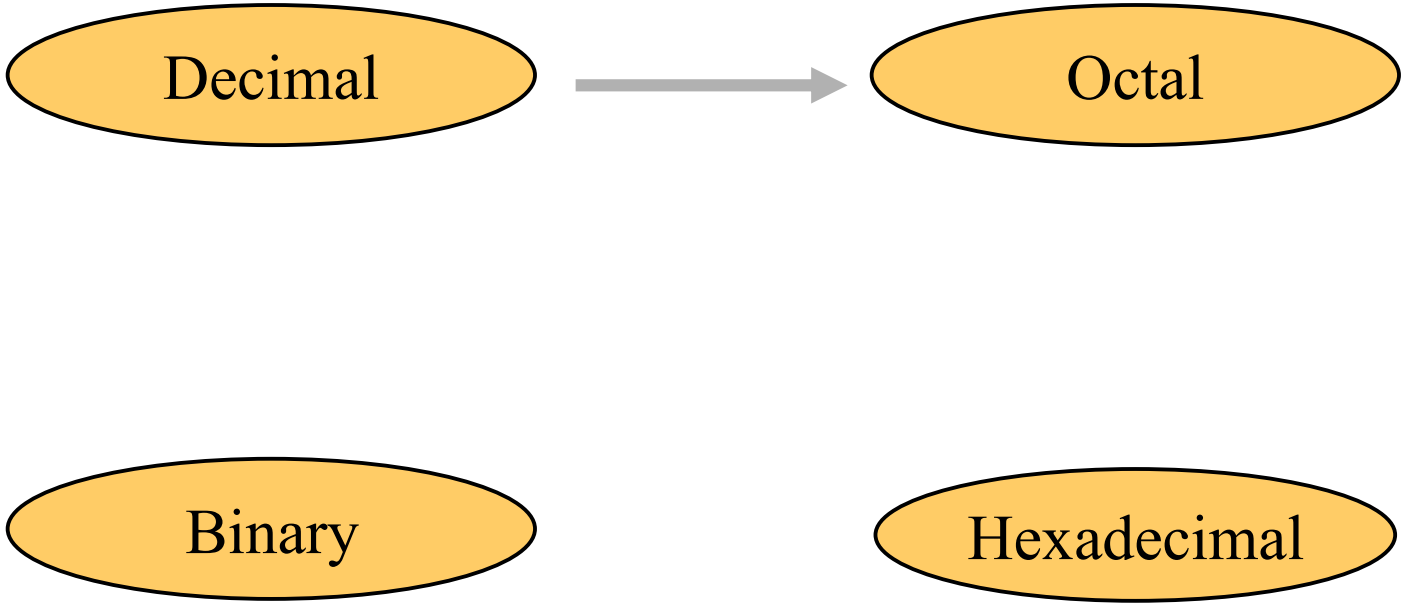
Example

$125_{10} = ?_2$

2	125	
2	62	1
2	31	0
2	15	1
2	7	1
2	3	1
2	1	1
	0	1

$125_{10} = 1111101_2$

Decimal to Octal



Decimal to Octal

- Technique
  - Divide by 8
  - Keep track of the remainder

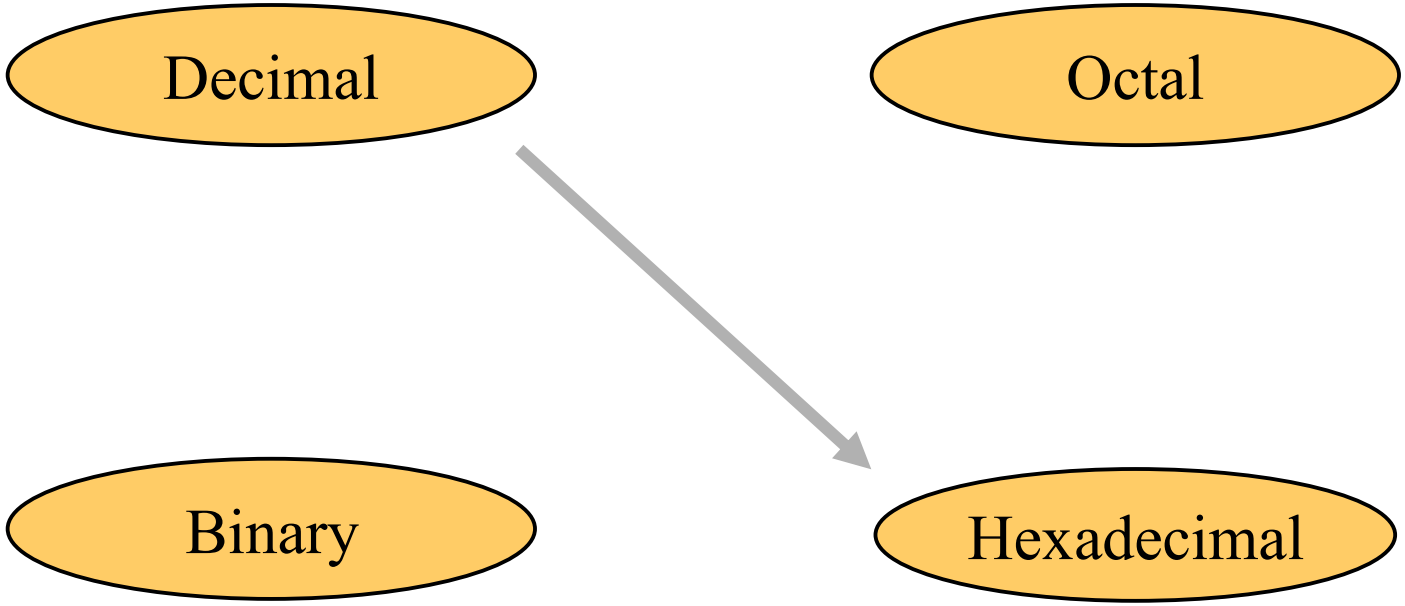
Example

$1234_{10} = ?_8$

8	1234	
8	154	2
8	19	2
8	2	3
	0	2

$1234_{10} = 2322_8$

Decimal to Hexadecimal



Decimal to Hexadecimal

- Technique
  - Divide by 16
  - Keep track of the remainder

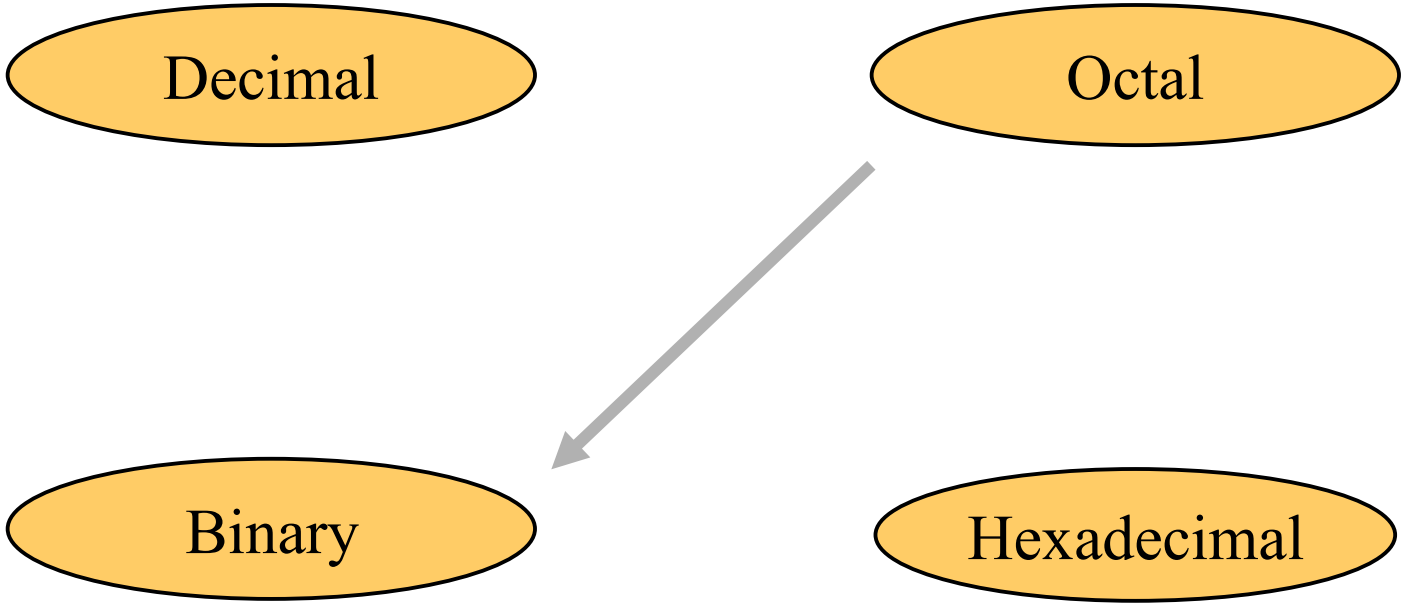
Example

$1234_{10} = ?_{16}$

```
16 | 1234
16 |   77    2
16 |    4    13 = D
        0    4
```

$1234_{10} = 4D2_{16}$

Octal to Binary



Octal to Binary

- Technique
  - Convert each octal digit to a 3-bit equivalent binary representation

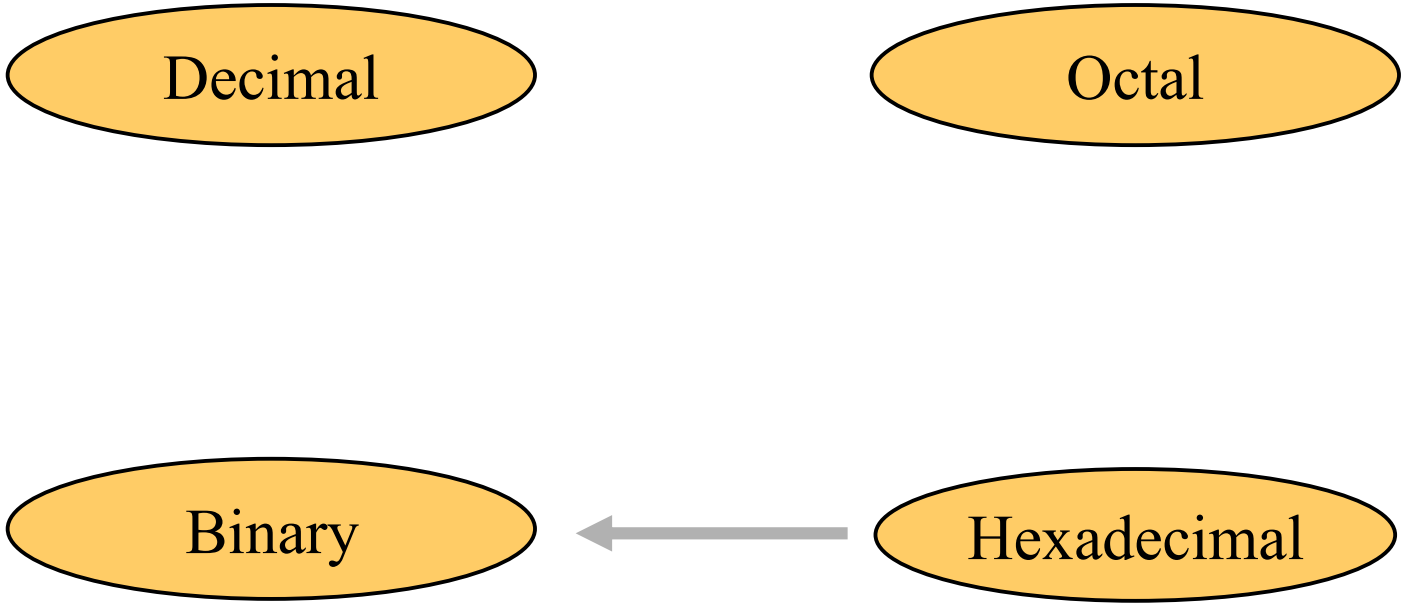
Example

$705_8 = ?_2$

7	0	5
↓	↓	↓
111	000	101

$705_8 = 111000101_2$

Hexadecimal to Binary



Hexadecimal to Binary

- Technique
  - Convert each hexadecimal digit to a 4-bit equivalent binary representation

Binary and hexadecimal conversions

- Converting from hexadecimal to binary is easy: replace each hex digit with its equivalent four-bit binary value.

$$261.A5_{16} = 2\ \ 6\ \ 1\ .\ A\ \ 5_{16}$$
$$= 0010\ \ 0110\ \ 0001\ .\ 1010\ \ 0101_2$$

- To convert from binary to hexadecimal, partition the binary number into groups of four bits, starting from the point. (Add 0s to the ends if needed.) Then replace each four-bit group by the corresponding hex digit.

$$10110100.001011_2 = 1011\ \ 0100\ .\ 0010\ \ 1100_2$$
$$B\ \ 4\ .\ 2\ \ C_{16}$$

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

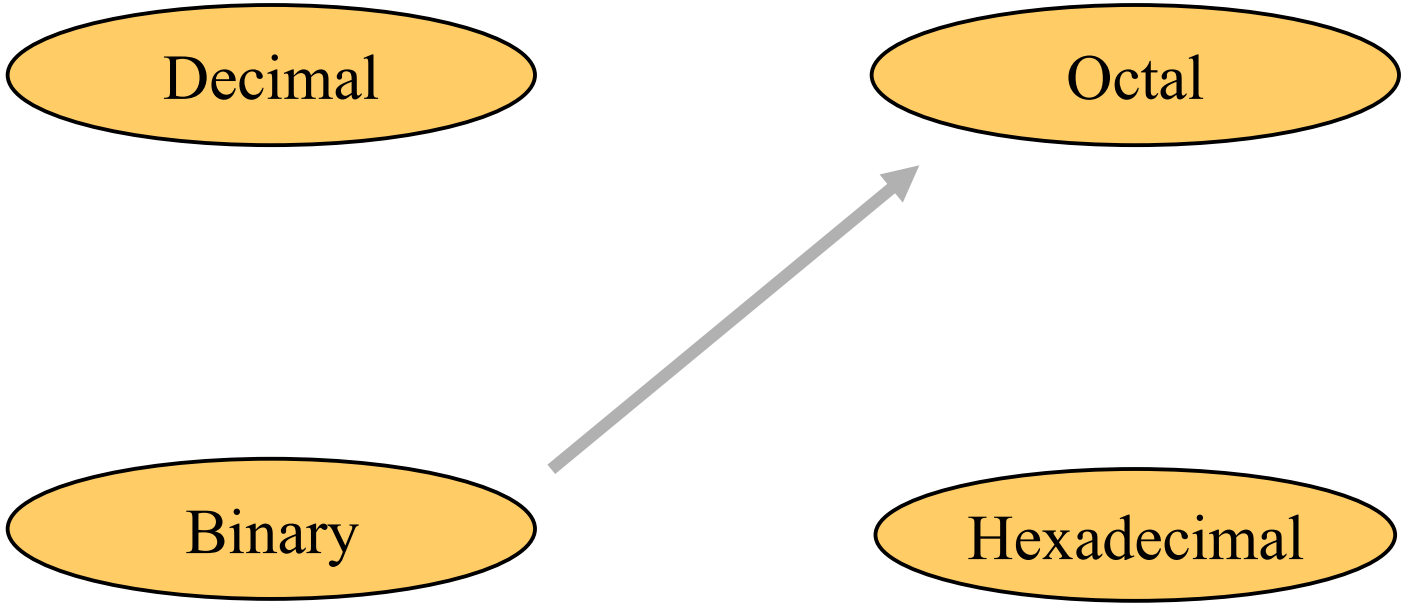
Example

$10AF_{16} = ?_2$

1	0	A	F
↓	↓	↓	↓
0001	0000	1010	1111

$10AF_{16} = 0001000010101111_2$

Binary to Octal



Binary to Octal

- Technique
  - Group bits in threes, starting on right
  - Convert to octal digits

Example

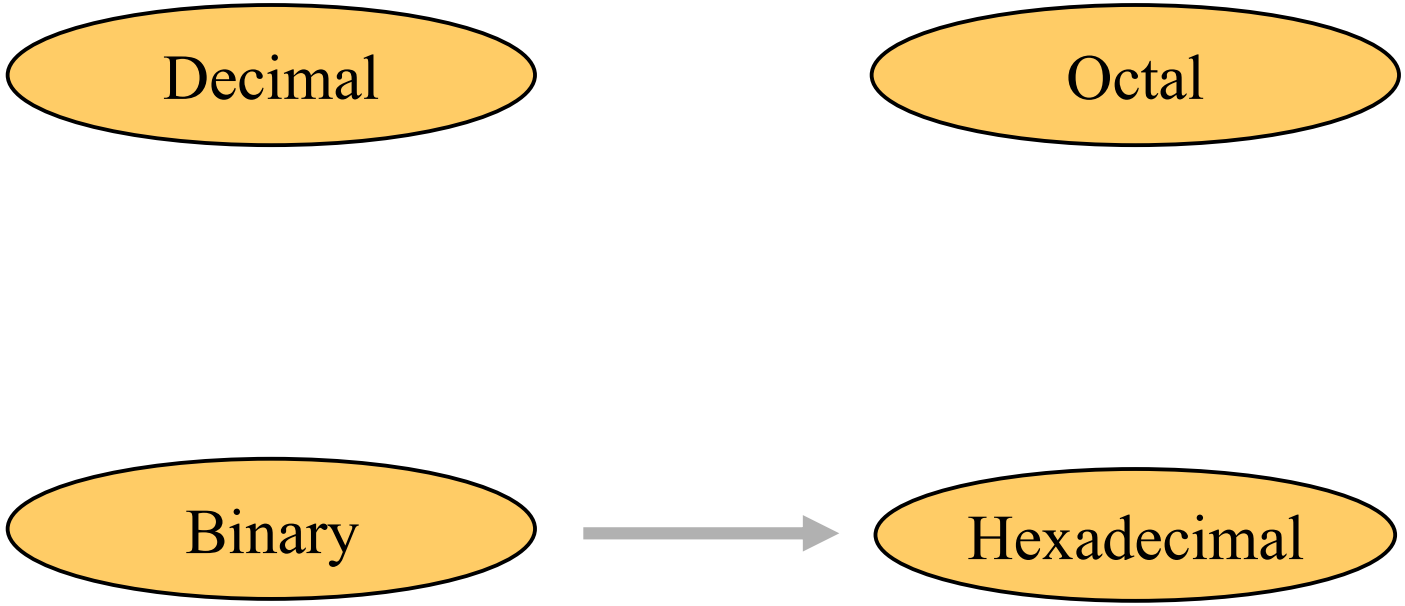
$1011010111_2 = ?_8$

1 011 010 111

1 3 2 7

$1011010111_2 = 1327_8$

Binary to Hexadecimal



Binary to Hexadecimal

- Technique
  - Group bits in fours, starting on right
  - Convert to hexadecimal digits

Example

$1010111011_2 = ?_{16}$

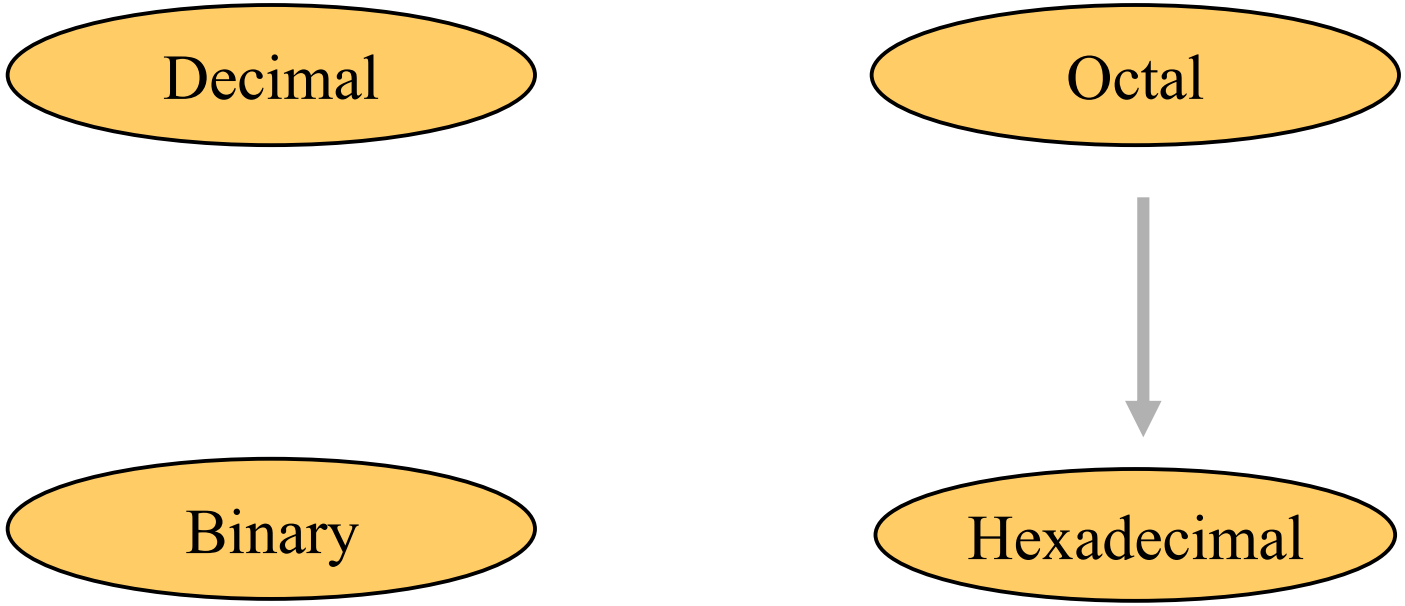
10 1011 1011

↓ ↓ ↓

2 B B

$1010111011_2 = 2BB_{16}$

Octal to Hexadecimal



Octal to Hexadecimal

- Technique
  - Use binary as an intermediary

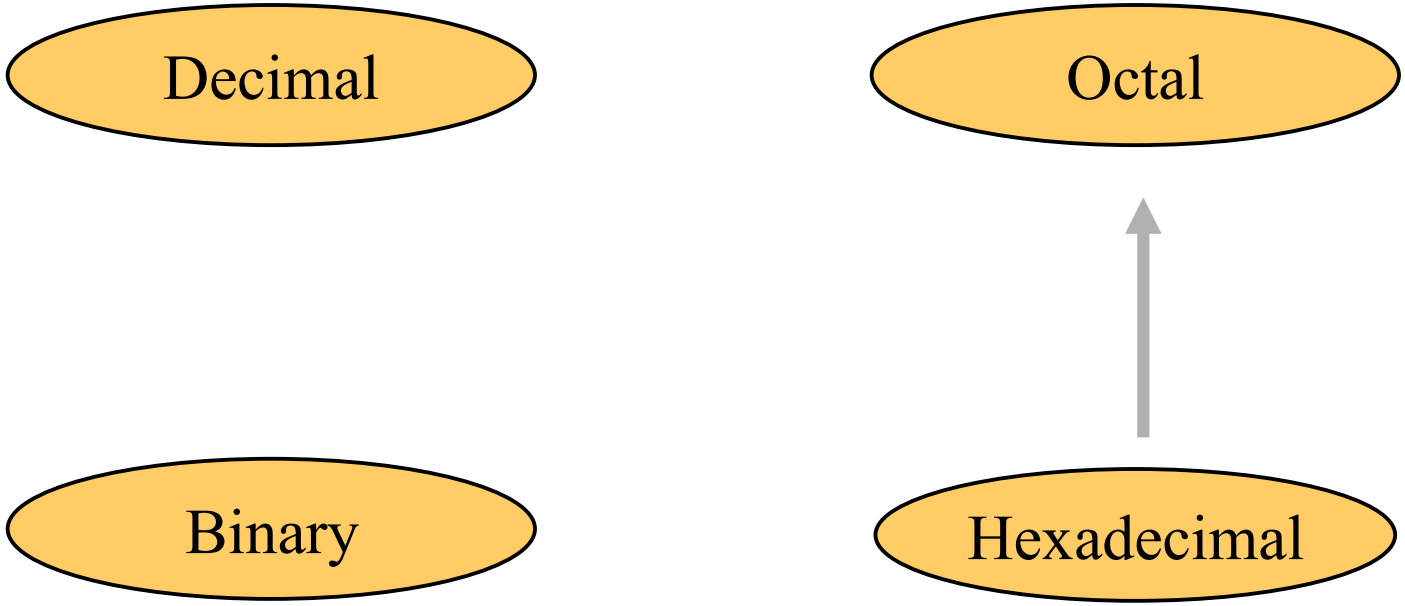
Example

$1076_8 = ?_{16}$

1	0	7	6
↓	↓	↓	↓
001	0|00	11|1	110
2	3		E

$1076_8 = 23E_{16}$

Hexadecimal to Octal



Hexadecimal to Octal

- Technique
  - Use binary as an intermediary

Example

$1F0C_{16} = ?_8$

1	F	0	C
0001	1111	0000	1100
1	7	4	1	4

$1F0C_{16} = 17414_8$

Exercise – Convert ...

Decimal	Binary	Octal	Hexa-decimal
33			
	1110101		
		703	
			1AF

Don’t use a calculator!

Skip answer

Answer

Exercise – Convert …

Answer

Decimal	Binary	Octal	Hexa-decimal
33	100001	41	21
117	1110101	165	75
451	111000011	703	1C3
431	110101111	657	1AF

Common Powers (1 of 2)

- Base 10

Power	Preface	Symbol	Value
10^{-12}	pico	p	.000000000001
10^{-9}	nano	n	.000000001
10^{-6}	micro	μ	.000001
10^{-3}	milli	m	.001
10^{3}	kilo	k	1000
10^{6}	mega	M	1000000
10^{9}	giga	G	1000000000
10^{12}	tera	T	1000000000000

Common Powers (2 of 2)

- Base 2

Power	Preface	Symbol	Value
2^{10}	kilo	k	1024
2^{20}	mega	M	1048576
2^{30}	Giga	G	1073741824

- What is the value of “k”, “M”, and “G”?
- In computing, particularly w.r.t. <u>memory</u>, the base-2 interpretation generally applies

Fractions

- Decimal to decimal (just for fun)

$$
\begin{array}{lrcl}
3.14 \Rightarrow & 4 \times 10^{-2} & = & 0.04 \\
 & 1 \times 10^{-1} & = & 0.1 \\
 & 3 \times 10^{0} & = & 3 \\
\hline
 & & & 3.14
\end{array}
$$

pp. 46-50

Fractions

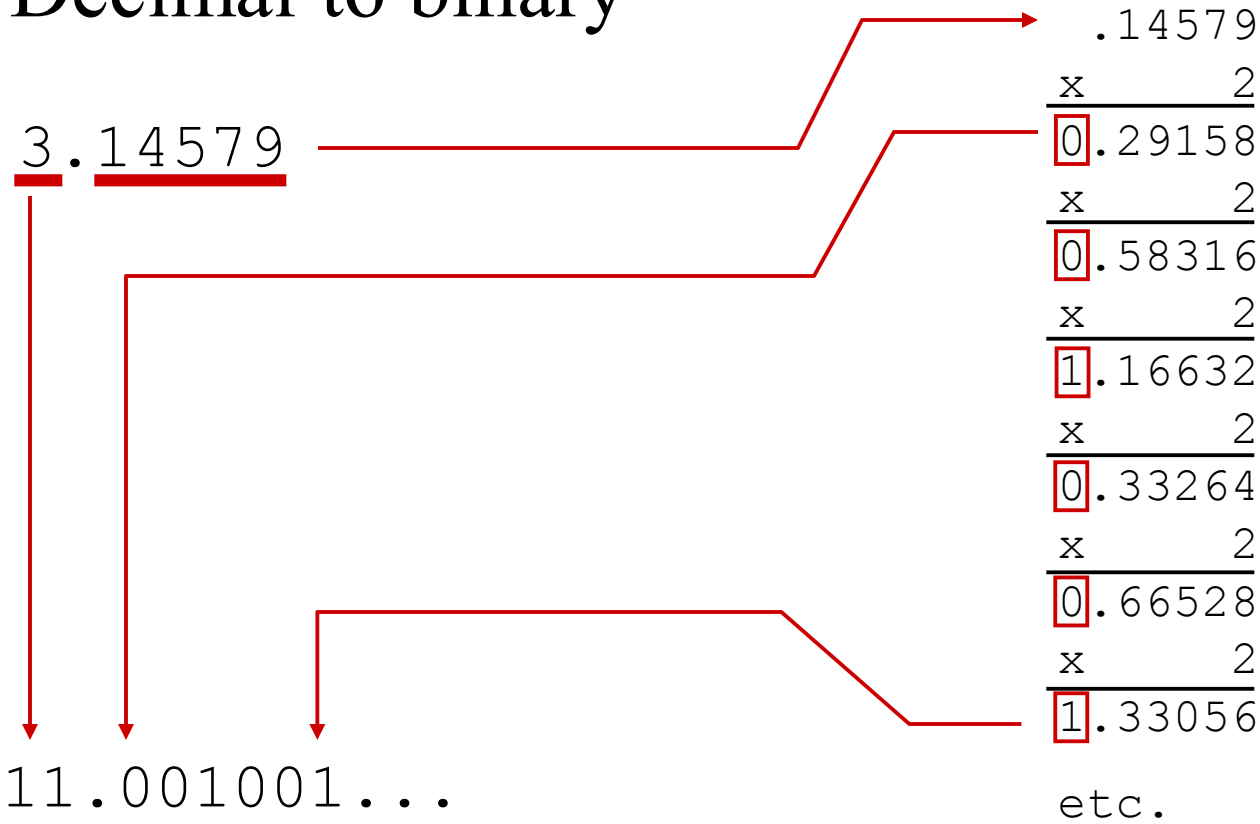
- Binary to decimal

```
10.1011 =>      1 x 2^-4 = 0.0625
                1 x 2^-3 = 0.125
                0 x 2^-2 = 0.0
                1 x 2^-1 = 0.5
                0 x 2^0  = 0.0
                1 x 2^1  = 2.0
                           ------
                           2.6875
```

pp. 46-50

Fractions

- Decimal to binary



Exercise – Convert ...

Decimal	Binary	Octal	Hexa-decimal
29.8			
	101.1101		
		3.07	
			C.82

Don’t use a calculator!

Skip answer

Answer

Exercise – Convert …

Answer

Decimal	Binary	Octal	Hexa-decimal
29.8	11101.110011…	35.63…	1D.CC…
5.8125	101.1101	5.64	5.D
3.109375	11.000111	3.07	3.1C
12.5078125	1100.10000010	14.404	C.82

4-Bit Binary Coded Decimal (BCD) Systems

- The 4-bit BCD system is usually employed by the computer systems to represent and process numerical data only. In the 4-bit BCD system, each digit of the decimal number is encoded to its corresponding 4-bit binary sequence. The two most popular 4-bit BCD systems are:

4-Bit BCD Code

Decimal digits	Weighted 4-bit BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

4-Bit BCD Code

- Represent the decimal number 5327 in BCD code.

The corresponding 4-bit BCD representation of decimal digit 5 is 0101

The corresponding 4-bit BCD representation of decimal digit 3 is 0011

The corresponding 4-bit BCD representation of decimal digit 2 is 0010

The corresponding 4-bit BCD representation of decimal digit 7 is 0111

Therefore, the BCD representation of decimal number 5327 is 0101001100100111.

College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 2

Logic Gates and Functions

Unit 2: Logic Gates and Functions

Overview

- Introduction to Digital Logic Gates
  - Truth table
  - Symbol
  - Universal Gates (NAND/NOR)
  - XOR and XNOR Gates
- Logic Chips
- Logic Functions
  - Logical Equivalence
  - Standard Forms (SOP/POS)

Chapter-2

M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

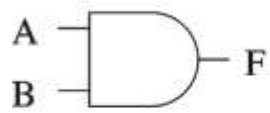
Introduction to Digital Logic Basics

- Hardware consists of a few simple building blocks
  - These are called *logic gates*
    - AND, OR, NOT, …
    - NAND, NOR, XOR, …
- Logic gates are built using transistors
    - NOT gate can be implemented by a single transistor
    - AND gate requires 3 transistors
- Transistors are the fundamental devices
    - Pentium consists of 3 million transistors
    - Compaq Alpha consists of 9 million transistors
    - Now we can build chips with more than 100 million transistors

Basic Concepts

- Simple gates
  - AND
  - OR
  - NOT
- Functionality can be expressed by a truth table
  - A truth table lists output for each possible input combination
- Precedence
  - NOT > AND > OR
  - $F = A\,\overline{B} + \overline{A}\,B$

    $= (A\,(\overline{B})) + ((\overline{A})\,B)$

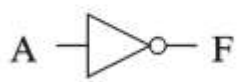


AND gate

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

OR gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1



NOT gate

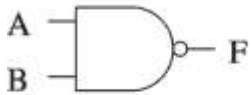
A	F
0	1
1	0

Logic symbol

Truth table

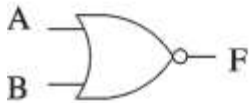
Basic Concepts (cont.)

- Additional useful gates
  - NAND
  - NOR
  - XOR
- NAND = AND + NOT
- NOR = OR + NOT
- XOR implements exclusive-OR function
- NAND and NOR gates require only 2 transistors
  - AND and OR need 3 transistors!



NAND gate

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0



NOR gate

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

XOR gate

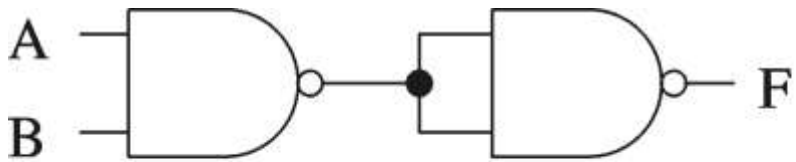
A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Logic symbol

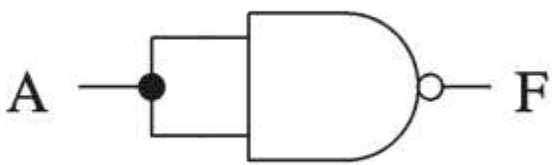
Truth table

Basic Concepts (cont.)

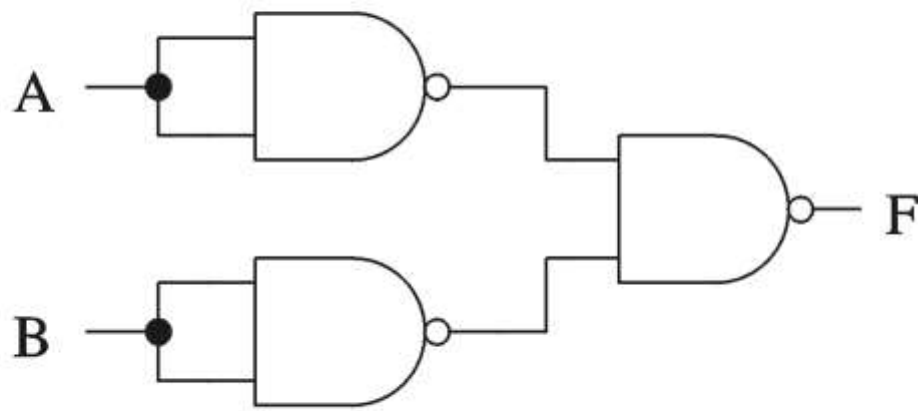
- Proving NAND gate is universal



AND gate



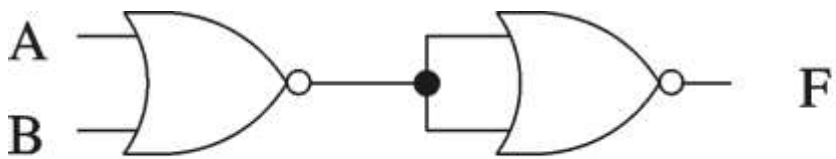
NOT gate



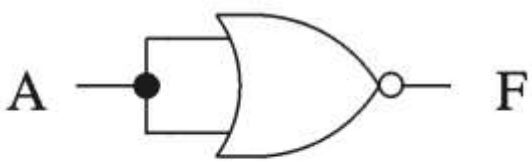
OR gate

Basic Concepts (cont.)

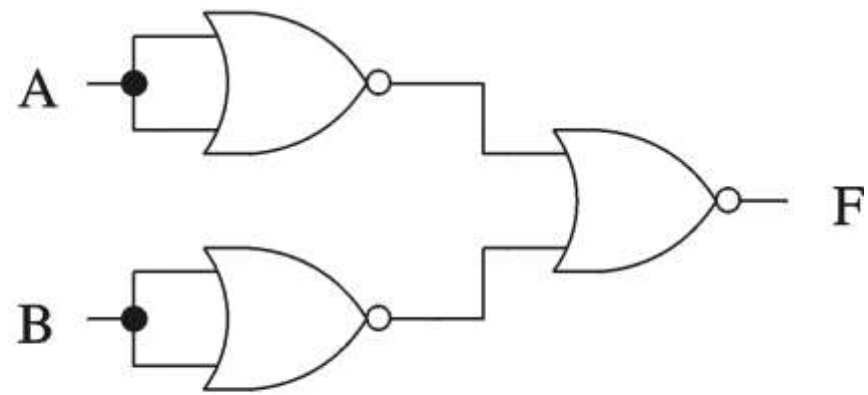
- Proving NOR gate is universal



OR gate



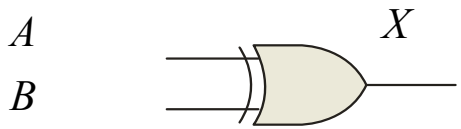
NOT gate

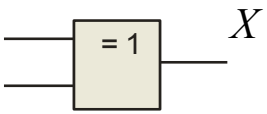


AND gate

XOR and XNOR Gates

The XOR Gate





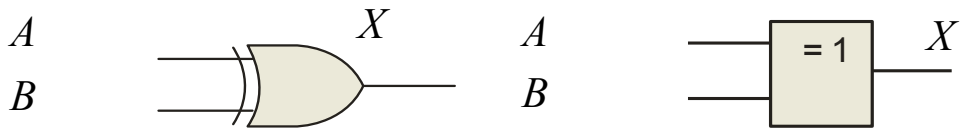
The **XOR gate** produces a HIGH output only when the inputs are at opposite logic levels. The truth table is

Inputs		Output
A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

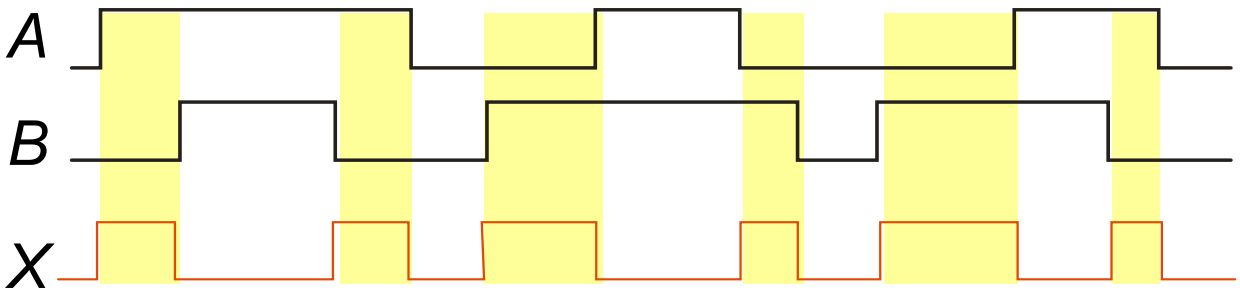
The **XOR** operation is written as $X = \bar{A}B + A\bar{B}$. Alternatively, it can be written with a circled plus sign between the variables as $X = A \oplus B$.

© 2009 Pearson Education, Upper Saddle River, NJ 07458. All Rights Reserved

The XOR Gate



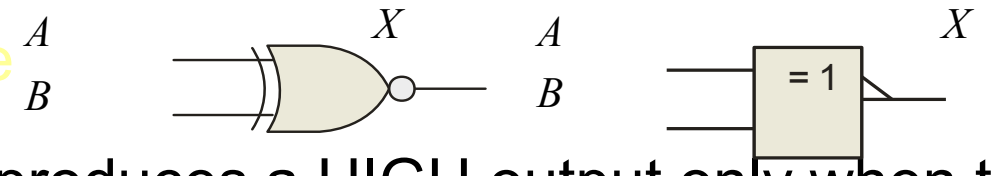
Example waveforms:



Notice that the XOR gate will produce a HIGH only when exactly one input is HIGH.

© 2009 Pearson Education, Upper Saddle River, NJ 07458. All Rights Reserved

The XNOR Gate



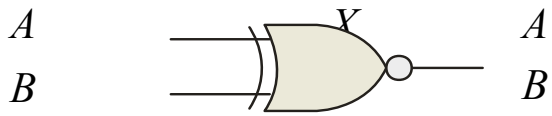
The **XNOR gate** produces a HIGH output only when the inputs are at the same logic level. The truth table is

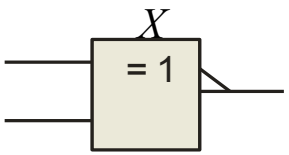
Inputs		Output
A	B	X
0	0	1
0	1	0
1	0	0
1	1	1

The **XNOR** operation can be shown as $X = AB + \bar{A}\bar{B}$.

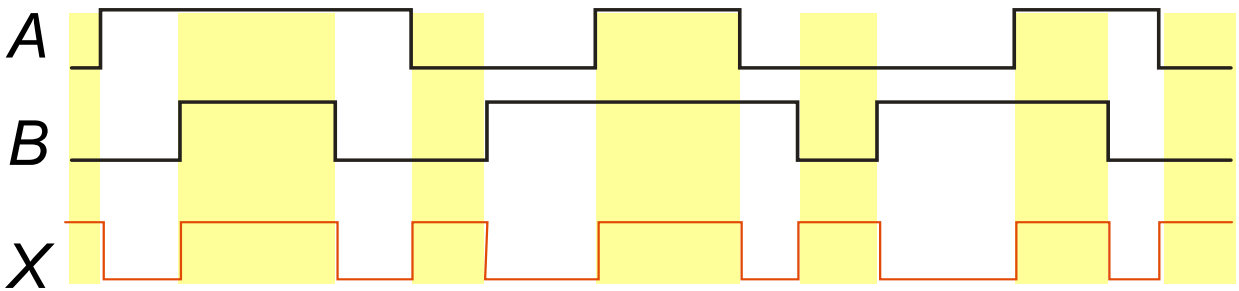
© 2009 Pearson Education, Upper Saddle River, NJ 07458. All Rights Reserved

The XNOR Gate





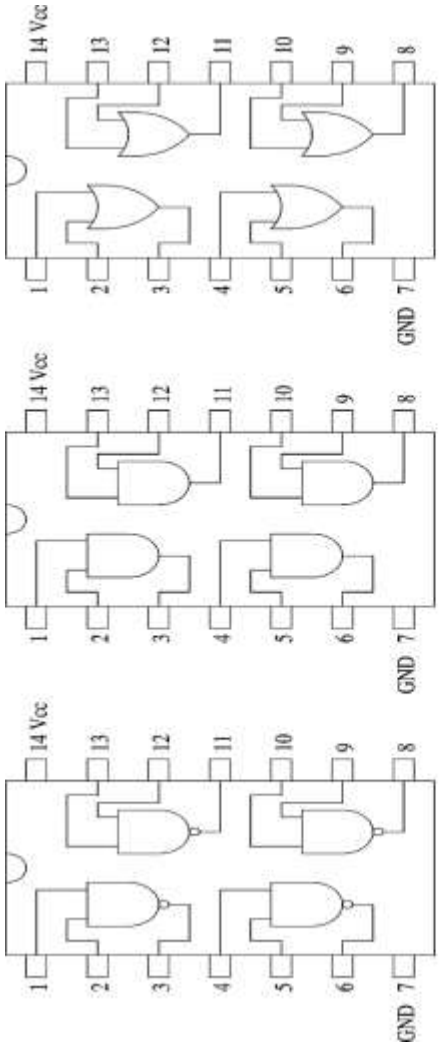
Example waveforms:



Notice that the XNOR gate will produce a HIGH when both inputs are the same. This makes it useful for comparison functions.

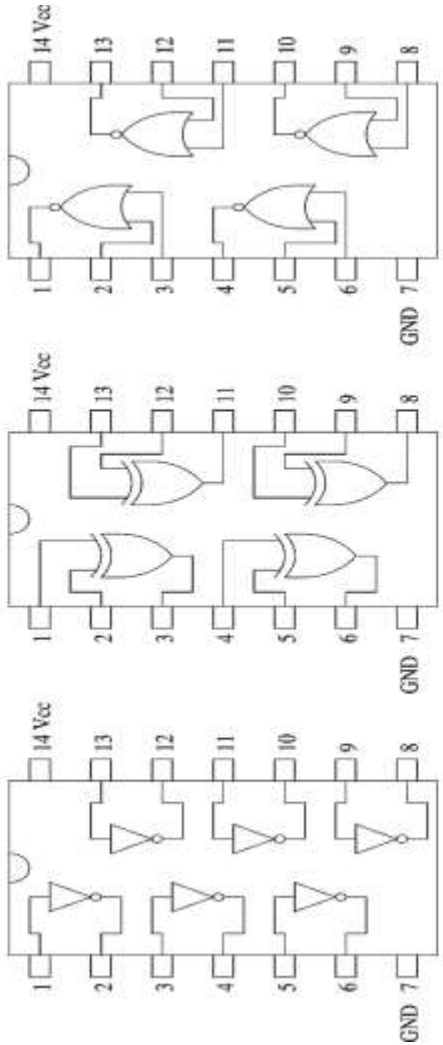
© 2009 Pearson Education, Upper Saddle River, NJ 07458. All Rights Reserved

Logic Chips


14 Vcc
13
12
11
10
9
8
1
2
3
4
5
6
GND 7
7400

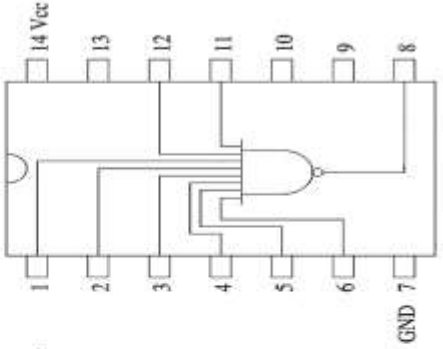
14 Vcc
13
12
11
10
9
8
1
2
3
4
5
6
GND 7
7408

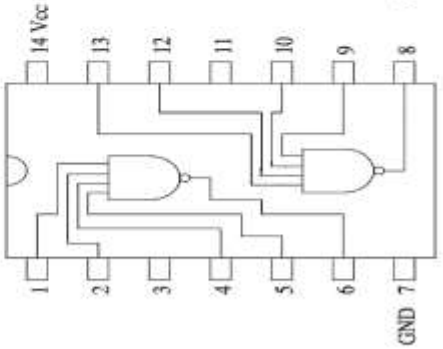
14 Vcc
13
12
11
10
9
8
1
2
3
4
5
6
GND 7
7432


14 Vcc
13
12
11
10
9
8
1
2
3
4
5
6
GND 7
7404

14 Vcc
13
12
11
10
9
8
1
2
3
4
5
6
GND 7
7486

14 Vcc
13
12
11
10
9
8
1
2
3
4
5
6
GND 7
7402


14 Vcc
13
12
11
10
9
8
1
2
3
4
5
6
GND 7
7430


14 Vcc
13
12
11
10
9
8
1
2
3
4
5
6
GND 7
7420

Logic Chips (cont.)

- Integration levels
  - SSI (small scale integration)
    - Introduced in late 1960s
    - 1-10 gates (previous examples)
  - MSI (medium scale integration)
    - Introduced in late 1960s
    - 10-100 gates
  - LSI (large scale integration)
    - Introduced in early 1970s
    - 100-10,000 gates
  - VLSI (very large scale integration)
    - Introduced in late 1970s
    - More than 10,000 gates

Logic Functions

- Number of functions
  - With N logical variables, we can define
    2^N combination of inputs
  - A function relates outputs to inputs
  - Some of them are useful
    - AND, NAND, NOR, XOR, …
  - Some are not useful:
    - Output is always 1
    - Output is always 0

Logic Functions

- Logical functions can be expressed in several ways:
  - Truth table
  - Logical expressions
  - Graphical form
- Example:
  - Majority function
    - Output is one whenever majority of inputs is 1
    - We use 3-input majority function

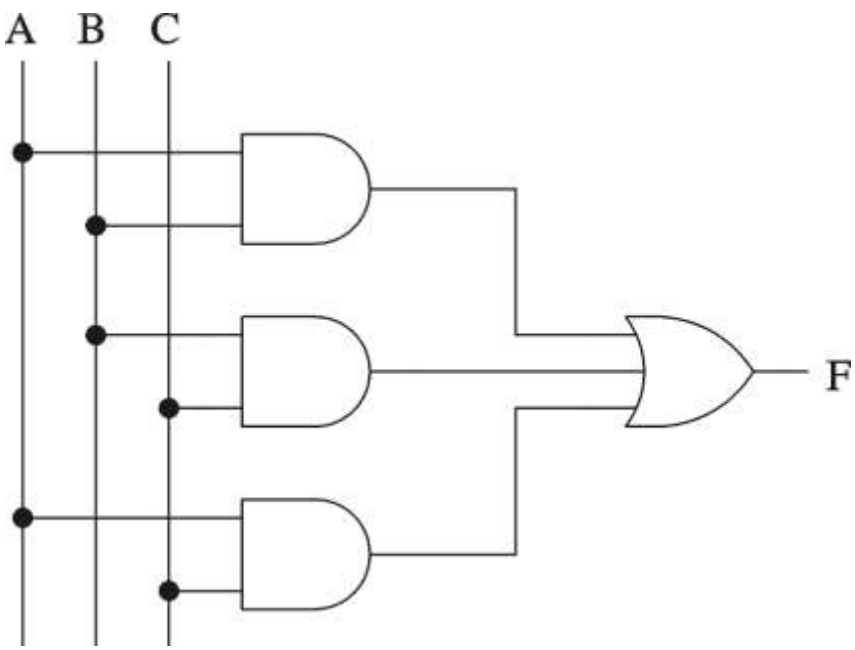
Logic Functions (cont.)

3-input majority function

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

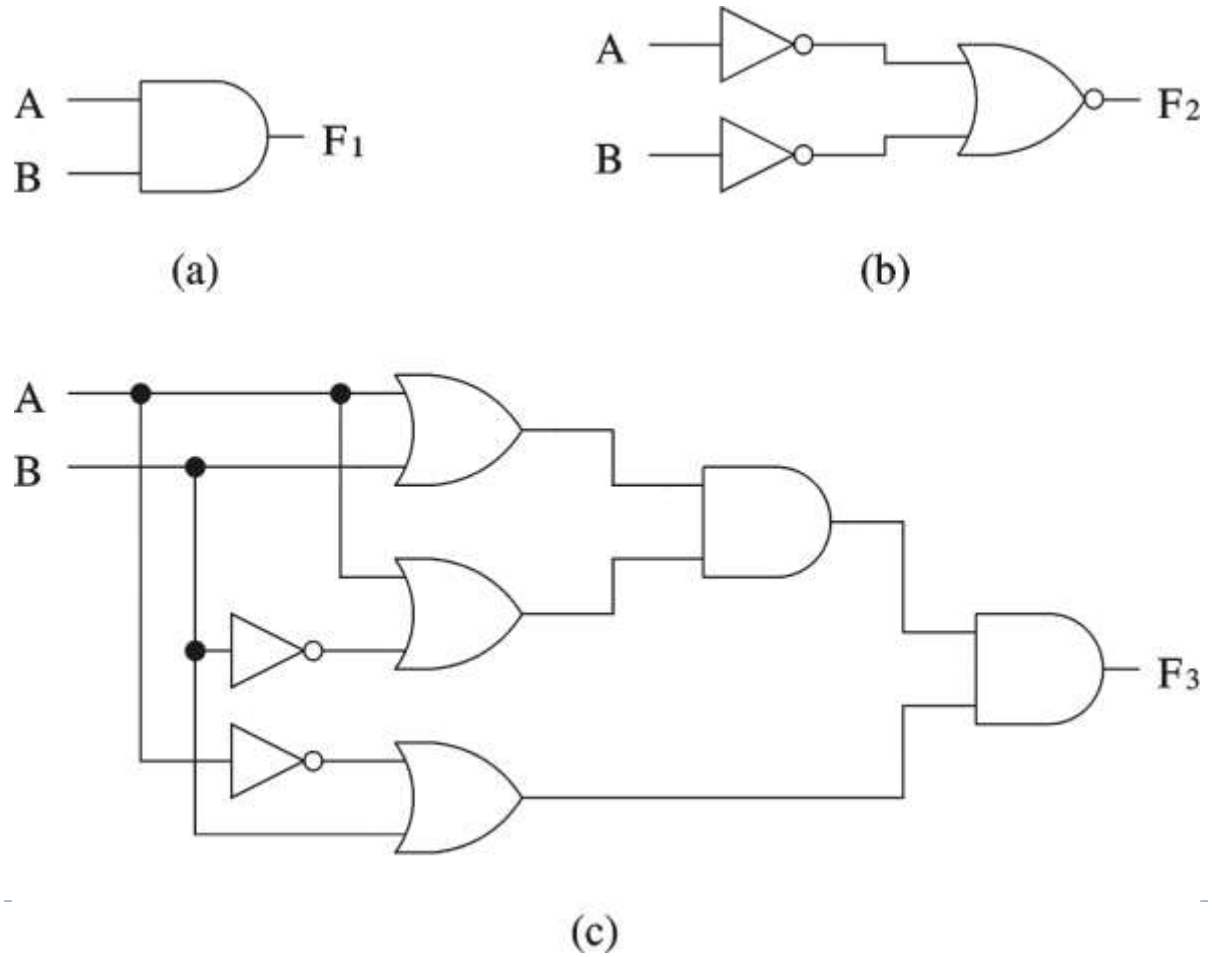
- Logical expression form

$$F = A\,B + B\,C + A\,C$$



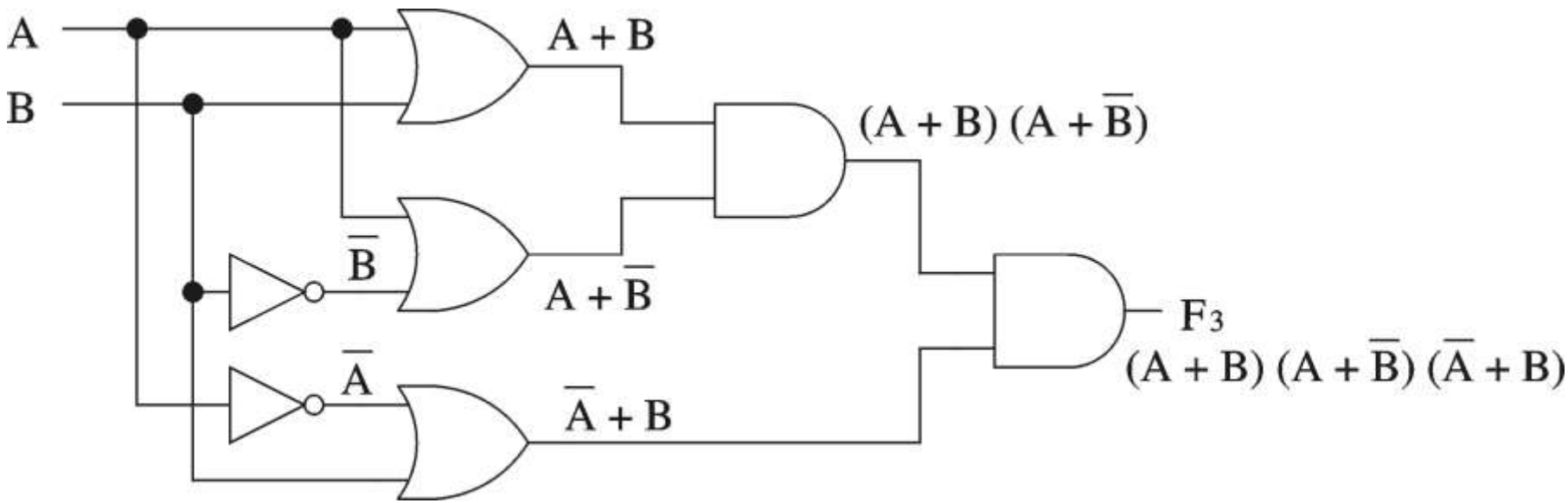
Logical Equivalence

- All three circuits implement F = A B function



Logical Equivalence

- Derivation of logical expression from a circuit
  - Trace from the input to output
    - Write down intermediate logical expressions along the path



Logical Equivalence (cont.)

- Proving logical equivalence: Truth table method

A	B	F1 = A B	F3 = (A + B) ($\overline{A}$ + B) (A + $\overline{B}$)
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Standard Forms for Boolean Expressions

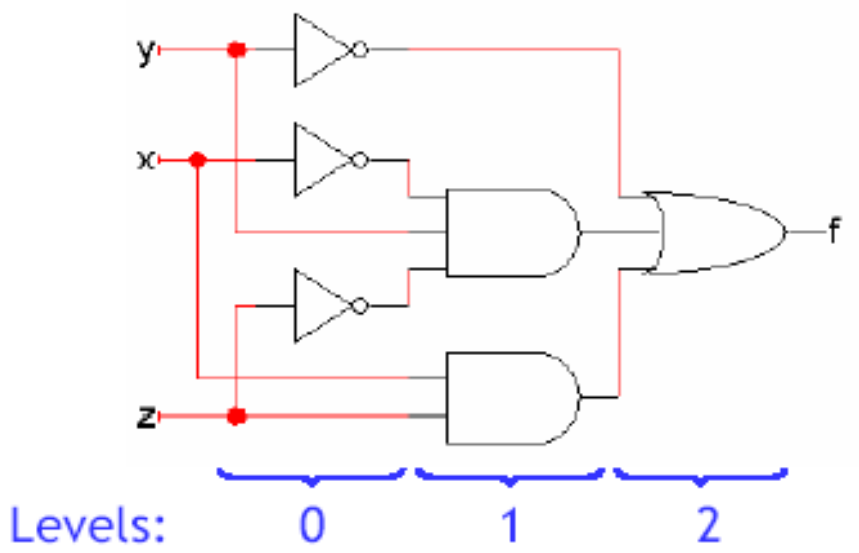
- Sum-of-Products (SOP)
  - Derived from the Truth table for a function by considering those rows for which F = 1.
  - The logical sum (OR) of product (AND) terms.
  - Realized using an AND-OR circuit.
- Product-of-Sums (POS)
  - Derived from the Truth table for a function by considering those rows for which F = 0.
  - The logical product (AND) of sum (OR) terms.
  - Realized using an OR-AND circuit.

1 Sum of products expressions

- There are many equivalent ways to write a function, but some forms turn out to be more useful than others.
- A sum of products or SOP expression consists of:
  - One or more terms *summed* (OR'ed) together.
  - Each of those terms is a *product of literals*.

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

- Sum of products expressions can be implemented with two-level circuits.



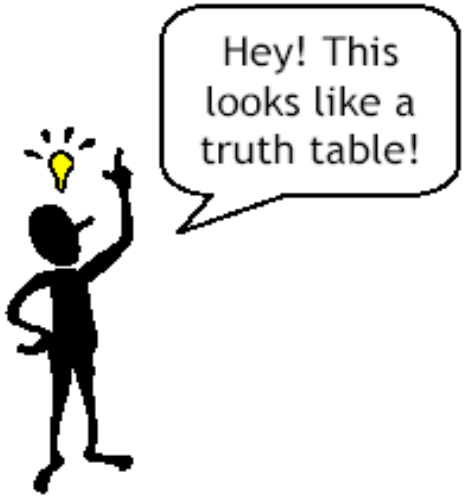
Minterms

- A minterm is a special product of literals, in which each input variable appears exactly once.
- A function with *n* input variables has 2^n possible minterms.
- For instance, a three-variable function f(x,y,z) has 8 possible minterms:

x'y'z'	x'y'z	x'y z'	x'y z
x y'z'	x y'z	x y z'	x y z

- Each minterm is true for exactly one combination of inputs.

Minterm	True when	Shorthand
x'y'z'	xyz = 000	m_0
x'y'z	xyz = 001	m_1
x'y z'	xyz = 010	m_2
x'y z	xyz = 011	m_3
x y'z'	xyz = 100	m_4
x y'z	xyz = 101	m_5
x y z'	xyz = 110	m_6
x y z	xyz = 111	m_7



Minterms

Row number	x_1	x_2	x_3	Minterm	Maxterm
0	0	0	0	$m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \overline{x}_1\overline{x}_2x_3$	$M_1 = x_1 + x_2 + \overline{x}_3$
2	0	1	0	$m_2 = \overline{x}_1x_2\overline{x}_3$	$M_2 = x_1 + \overline{x}_2 + x_3$
3	0	1	1	$m_3 = \overline{x}_1x_2x_3$	$M_3 = x_1 + \overline{x}_2 + \overline{x}_3$
4	1	0	0	$m_4 = x_1\overline{x}_2\overline{x}_3$	$M_4 = \overline{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1\overline{x}_2x_3$	$M_5 = \overline{x}_1 + x_2 + \overline{x}_3$
6	1	1	0	$m_6 = x_1x_2\overline{x}_3$	$M_6 = \overline{x}_1 + \overline{x}_2 + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$

Sum-of-Products

- Any function F can be represented by a sum of minterms, where each minterm is ANDed with the corresponding value of the output for F.
  - $F = \Sigma\ (m_i \cdot f_i)$
    - where m_i is a minterm
    - and f_i is the corresponding functional output
  - Only the minterms for which $f_i = 1$ appear in the expression for function F.
  - $F = \Sigma\ (m_i) = \Sigma\ m(i)$

Denotes the logical sum operation

shorthand notation

Sum of minterms expressions

- A sum of minterms is a special kind of sum of products.
- Every function can be written as a *unique* sum of minterms expression.
- A truth table for a function can be rewritten as a sum of minterms just by finding the table rows where the function output is 1.

x	y	z	C(x,y,z)	C'(x,y,z)
0	0	0	0	1
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	0

$$C = x'yz + xy'z + xyz' + xyz$$
$$= m_3 + m_5 + m_6 + m_7$$
$$= \Sigma m(3,5,6,7)$$

$$C' = x'y'z' + x'y'z + x'yz' + xy'z'$$
$$= m_0 + m_1 + m_2 + m_4$$
$$= \Sigma m(0,1,2,4)$$

C' contains all the minterms *not* in C, and vice versa.

Sum-of-Products

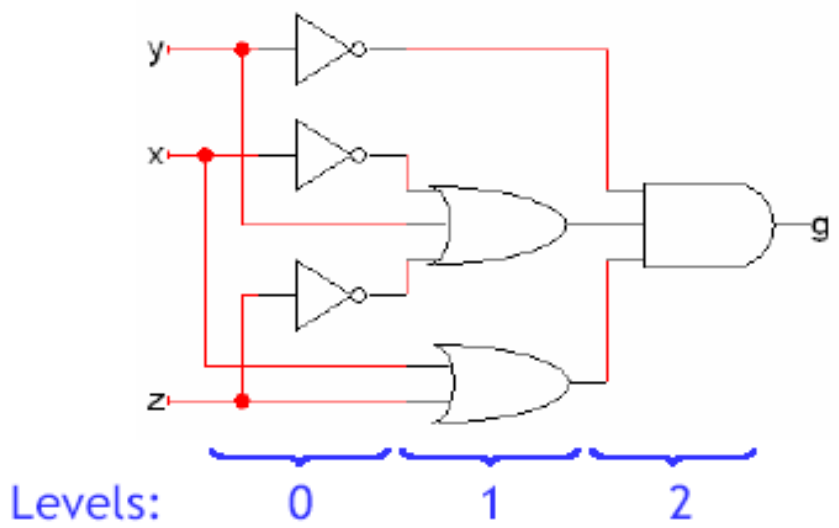
- Sum of minterms are a.k.a. Canonical Sum-of-Products
- Synthesis process
  - Determine the Canonical Sum-of-Products
  - Use Boolean Algebra (and K-maps) to find an optimal, functionally equivalent, expression.

2 Product of sums expressions

- As you might expect, we can work with the duals of these ideas too.
- A product of sums or POS consists of:
  - One or more terms *multiplied* (AND'ed) together.
  - Each of those terms is a *sum of literals*.

$$g(x, y, z) = y'(x' + y + z')(x + z)$$

- Products of sums can also be implemented with two-level circuits.



Maxterms

- A maxterm is a *sum* of literals where each input variable appears once.
- A function with *n* input variables has 2^n possible maxterms.
- For instance, a function with three variables x, y and z has 8 possible maxterms:

$$x + y + z \qquad x + y + z' \qquad x + y' + z \qquad x + y' + z'$$
$$x' + y + z \qquad x' + y + z' \qquad x' + y' + z \qquad x' + y' + z'$$

- Each maxterm is *false* for exactly one combination of inputs.

Maxterm	False when	Shorthand
$x + y + z$	$xyz = 000$	M_0
$x + y + z'$	$xyz = 001$	M_1
$x + y' + z$	$xyz = 010$	M_2
$x + y' + z'$	$xyz = 011$	M_3
$x' + y + z$	$xyz = 100$	M_4
$x' + y + z'$	$xyz = 101$	M_5
$x' + y' + z$	$xyz = 110$	M_6
$x' + y' + z'$	$xyz = 111$	M_7

Maxterms

Row number	x_1	x_2	x_3	Minterm	Maxterm
0	0	0	0	$m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \overline{x}_1\overline{x}_2x_3$	$M_1 = x_1 + x_2 + \overline{x}_3$
2	0	1	0	$m_2 = \overline{x}_1x_2\overline{x}_3$	$M_2 = x_1 + \overline{x}_2 + x_3$
3	0	1	1	$m_3 = \overline{x}_1x_2x_3$	$M_3 = x_1 + \overline{x}_2 + \overline{x}_3$
4	1	0	0	$m_4 = x_1\overline{x}_2\overline{x}_3$	$M_4 = \overline{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1\overline{x}_2x_3$	$M_5 = \overline{x}_1 + x_2 + \overline{x}_3$
6	1	1	0	$m_6 = x_1x_2\overline{x}_3$	$M_6 = \overline{x}_1 + \overline{x}_2 + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$

Product-of-Sums

- Any function F can be represented by a product of Maxterms, where each Maxterm is ANDed with the *complement* of the corresponding value of the output for F.
  - $F = \Pi\ (M_i \,.\, f'_i)$
    - where M_i is a Maxterm
    - and f'_i is the complement of the corresponding functional output
  - Only the Maxterms for which $f_i = 0$ appear in the expression for function F.
  - $F = \Pi\ (M_i) = \Pi\ M(i)$

Denotes the logical product operation

shorthand notation

Product of maxterms expressions

- Every function can also be written as a unique product of maxterms.
- A truth table for a function can be rewritten as a product of maxterms just by finding the table rows where the function output is 0.

x	y	z	C(x,y,z)	C'(x,y,z)
0	0	0	0	1
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	0

$$C = (x + y + z)(x + y + z')(x + y' + z)(x' + y + z)$$
$$= M_0 M_1 M_2 M_4$$
$$= \prod M(0,1,2,4)$$ **When the o/p is Zero**
$$= \Sigma m(3,5,6,7)$$ **When the o/p is 1**

$$C' = (x + y' + z')(x' + y + z')(x' + y' + z)(x' + y' + z')$$
$$= M_3 M_5 M_6 M_7$$
$$= \prod M(3,5,6,7)$$

C' contains all the maxterms *not* in C, and vice versa.

Product-of-Sums

- The Canonical Product-of-Sums for function F is the Product-of-Sums expression in which each sum term is a Maxterm.
- Synthesis process
  - Determine the Canonical Product-of-Sums
  - Use Boolean Algebra (and K-maps) to find an optimal, functionally equivalent, expression.

College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 3
Simplification and K-map

Unit 3: Simplification and K-map

Overview

- Minimal Sum of Product (MSP)
  - K-Map
  - Four Steps
- Don't Care Condition
- Four Variable K-Map

Chapter-2

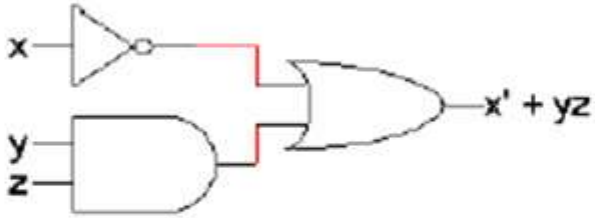
M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

Minimal sums of products

- When used properly, Karnaugh maps can reduce expressions to a minimal sum of products, or MSP, form.
  - There are a minimal number of product terms.
  - Each product has a minimal number of literals.
- For example, both expressions below (from the last lecture) are sums of products, but only the right one is a *minimal* sum of products.

$$x'y' + yz + x'y = x' + yz$$

- Minimal sum of products expressions lead to minimal two-level circuits.



- A minimal sum of products might not be "minimal" by other definitions! For example, the MSP $xy + xz$ can be reduced to $x(y + z)$, which has fewer literals and operators—but it is no longer a sum of products.

Prove?

Organizing the minterms

- Recall that an *n*-variable function has up to 2^n minterms, one for each possible input combination.
- A function with inputs x, y and z includes up to eight minterms, as shown below.

x	y	z	Minterm	
0	0	0	x'y'z'	(m_0)
0	0	1	x'y'z	(m_1)
0	1	0	x'y z'	(m_2)
0	1	1	x'y z	(m_3)
1	0	0	x y'z'	(m_4)
1	0	1	x y'z	(m_5)
1	1	0	x y z'	(m_6)
1	1	1	x y z	(m_7)

- We'll rearrange these minterms into a Karnaugh map, or K-map.

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

x'y'z'	x'y'z	x'y z	x'y z'
x y'z'	x y'z	x y z	x y z'

- You can show either the actual minterms or just the minterm numbers.
- Notice the minterms are almost, but not quite, in numeric order.

Reducing two minterms

- In this layout, any two adjacent minterms contain at least one common literal. This is useful in simplifying the sum of those two minterms.
- For instance, the minterms x'y'z' and x'y'z both contain x' and y', and we can use Boolean algebra to show that their sum is x'y'.

x'y'z'	x'y'z	x'y z	x'y z'
x y'z'	x y'z	x y z	x y z'

$$
\begin{aligned}
x'y'z' + x'y'z &= x'y'(z' + z) \\
&= x'y' \bullet 1 \\
&= x'y'
\end{aligned}
$$

- You can also "wrap around" the sides of the K-map—minterms in the first and fourth columns are considered to be next to each other.

x'y'z'	x'y'z	x'y z	x'y z'
x y'z'	x y'z	x y z	x y z'

$$
\begin{aligned}
x\,y'z' + x\,y\,z' &= xz'(y' + y) \\
&= xz' \bullet 1 \\
&= xz'
\end{aligned}
$$

Reducing four minterms

- Similarly, rectangular groups of four minterms can be reduced as well. You can think of them as two adjacent groups of two minterms each.

x'y'z'	x'y'z	x'y z	x'y z'
x y'z'	x y'z	x y z	x y z'

- These four green minterms all have the literal y in common. Guess what happens when you simplify their sum?

$$
\begin{aligned}
x'yz + x'yz' + xyz + xyz' &= y(x'z + x'z' + xz + xz') \\
&= y(x'(z + z') + x(z + z')) \\
&= y(x' + x) \\
&= y
\end{aligned}
$$

Reducible groups

- Only rectangular groups of minterms, where the number of minterms is a power of two, can be reduced to a single product term.
  - Non-rectangular groups may not even contain a common literal.

x'y'z'	x'y'z	x'y z	x'y z'
x y'z'	x y'z	x y z	x y z'

  - Groups of other sizes cannot be simplified to just one product term.

x'y'z'	x'y'z	x'y z	x'y z'
x y'z'	x y'z	x y z	x y z'

The pattern behind the K-map

- The literal x occurs in the bottom four minterms, while the literal x' appears in the top four minterms.

x'	x'y'z'	x'y'z	x'y z	x'y z'
x	x y'z'	x y'z	x y z	x y z'

- The literal y shows up on the right side, and y' appears on the left.

y'		y	
x'y'z'	x'y'z	x'y z	x'y z'
x y'z'	x y'z	x y z	x y z'

- The literal z occurs in the middle four squares, while z' occurs in the first and fourth columns.

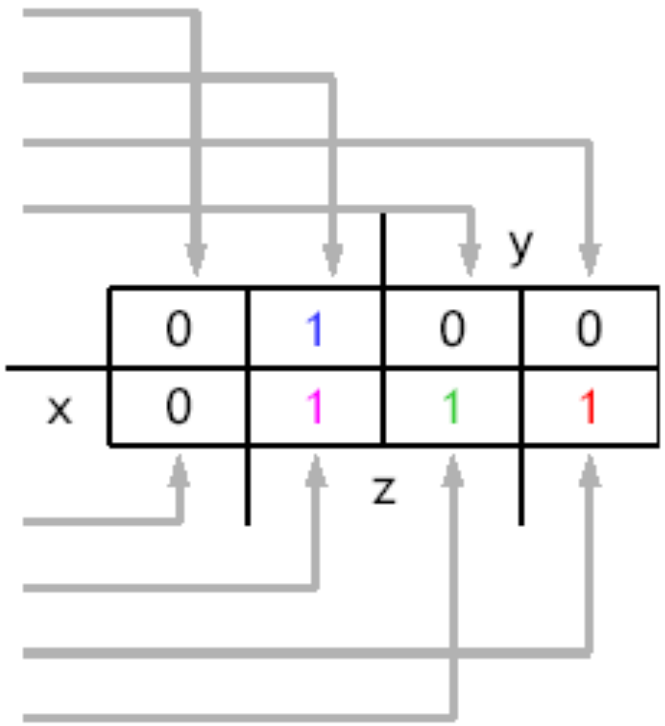
x'y'z'	x'y'z	x'y z	x'y z'
x y'z'	x y'z	x y z	x y z'
z'	z		z'

Filling in the K-map

- Since our labels help us find the correct position of minterms in a K-map, writing the minterms themselves is redundant and repetitive.
- We usually just put a 1 in the K-map squares that correspond to the function minterms, and 0 in the other squares.
- For example, you can quickly fill in a K-map from a truth table by copying the function outputs to the proper squares of the map.

Example 1

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



f(x,y,z) = x'y'z + xy'z + xyz' + xyz

Example 2

Four steps in K-map simplifications

1. Start with a sum of minterms or truth table. $x'y'z' + x'y'z + xyz + xyz'$

2. Plot the minterms on a Karnaugh map.

			y	
	1	1	0	0
x	0	0	1	1
		z		

3. Find rectangular groups of minterms whose sizes are powers of two. Be sure to include all the minterms in at least one group!

			y	
	1	1	0	0
x	0	0	1	1
		z		

4. Reduce each group to one product term. $x'y' + xy$

The tricky part

- The tricky part is finding the best groups of minterms.
  - Each group represents one product term, so *making as few groups as possible* will result in a minimal number of products.
  - *Making each group as large as possible* corresponds to combining more minterms, and will result in a minimal number of literals.
- Which groups would you form in the following example map?

Example 3

			y	
	0	0	1	1
x	1	1	1	1
		z		

Minimizing the number of groups

- The following two possibilities include too many groups, and would result in more product terms than necessary.

			y	
	0	0	1	1
x	1	1	1	1
		z		

			y	
	0	0	1	1
x	1	1	1	1
		z		

- We can put all six minterms into just two groups. Two ways of doing this are shown below.

			y	
	0	0	1	1
x	1	1	1	1
		z		

			y	
	0	0	1	1
x	1	1	1	1
		z		

Solutions for practice K-map 1

- Here is the K-map for $f(x,y,z) = m_1 + m_3 + m_5 + m_6$, with all groups shown.
  - The magenta and green groups overlap, which makes each of them as large as possible.
  - Minterm m_6 is in a group all by its lonesome.

			y	
	0	1	1	0
x	0	1	0	1
		z		

Example 4

MSP
Minimal **S**um of **P**roduct

- The final MSP here is $x'z + y'z + xyz'$.

Don't care conditions

- There are times when we don't care what a function outputs—some input combinations might never occur, or some outputs may have no effect.
- We can express these situations with don't care conditions, denoted with X in truth table rows.
- An expression for this function has two parts.
  - One part includes the function's minterms.
  - Another describes the don't care conditions.

$$f(x,y,z) = m_3,\ d(x,y,z) = m_2 + m_4 + m_5$$

- Circuits *always* output 0 or 1; there is no value called "X". Instead, the Xs just indicate cases where both 0 or 1 would be acceptable outputs.

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	0
0	1	0	X
0	1	1	1
1	0	0	X
1	0	1	X
1	1	0	0
1	1	1	0

Example 5

Don't care simplifications

- In a K-map we can treat each don't care as 0 or 1. Different selections can produce different results.

			y	
	0	0	1	X
x	X	X	0	0
		z		

Example 5

- In this example we can use the don't care conditions to our advantage.
  - It's best to treat the bottom two Xs as 0s. If either of them were 1, we'd end up with an extra, unnecessary term.
  - On the other hand, interpreting the top X as 1 results in a larger group containing m_3.
- The resulting MSP is x'y.

Four-variable Karnaugh maps

- We can do four-variable Karnaugh maps too!
- A four-variable function f(w,x,y,z) has sixteen possible minterms. They can be arranged so that adjacent minterms have common literals.
  - You can wrap around the sides *and* the top and bottom.
  - Again the minterms are almost, but not quite, in numeric order.

			y	
	w'x'y'z'	w'x'y'z	w'x'y z	w'x'y z'
	w'x y'z'	w'x y'z	w'x y z	w'x y z'
w	w x y'z'	w x y'z	w x y z	w x y z'
	w x'y'z'	w x'y'z	w x'y z	w x'y z'
		z		

x

yz / wx			y	
	m_0	m_1	m_3	m_2
	m_4	m_5	m_7	m_6
w	m_{12}	m_{13}	m_{15}	m_{14}
	m_8	m_9	m_{11}	m_{10}
		z		

x

Four-variable example

- Let's say we want to simplify $m_0 + m_2 + m_5 + m_8 + m_{10} + m_{13}$

		y		
	m_0	m_1	m_3	m_2
	m_4	m_5	m_7	m_6
w	m_{12}	m_{13}	m_{15}	m_{14}
	m_8	m_9	m_{11}	m_{10}
		z		

x

		y		
	1	0	0	1
	0	1	0	0
w	0	1	0	0
	1	0	0	1
		z		

x

Example 6

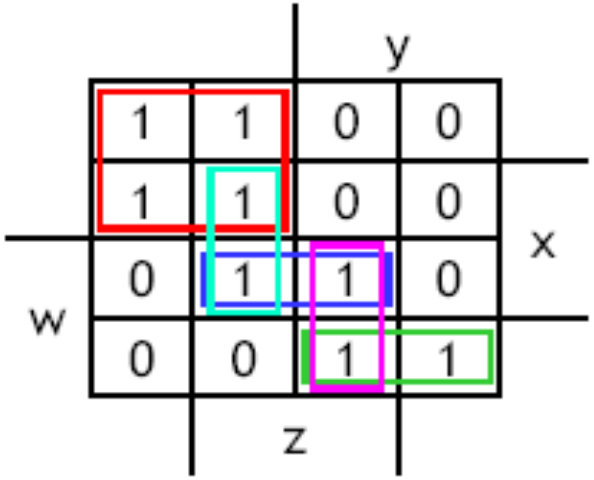
- The following groups result in the minimal sum of products $x'z' + xy'z$.

		y		
	1	0	0	1
	0	1	0	0
w	0	1	0	0
	1	0	0	1
		z		

x

Prime implicants

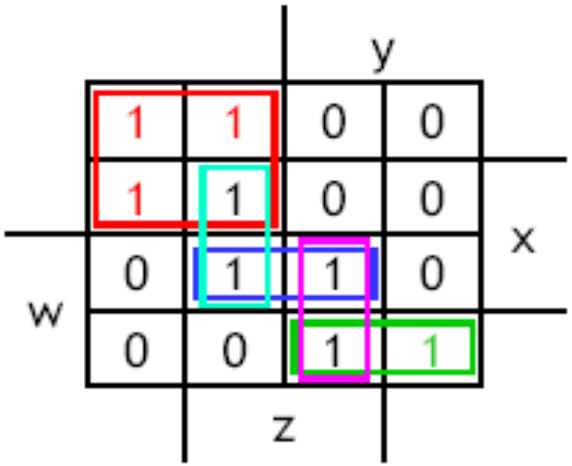
- Finding the best groups is even more difficult in larger K-maps.
- One good approach to deriving an MSP is to first find the largest possible groupings of minterms.
  - These groups correspond to prime implicant terms.
  - The final MSP will contain a subset of the prime implicants.
- Here is an example K-map with prime implicants marked.

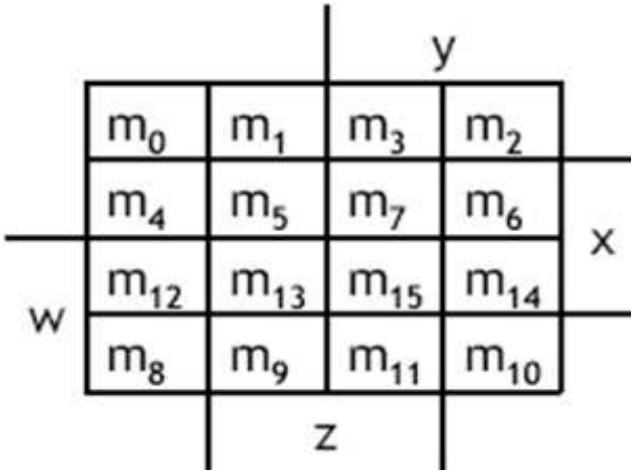


Example 7

Essential prime implicants

- If any minterm belongs to only one group, then that group represents an essential prime implicant.
- Essential prime implicants *must* appear in the final MSP, which has to include all of the original minterms.





- This example has two essential prime implicants.
  - The red group (w'y) is essential, since m_0, m_1 and m_4 are not in any other group.
  - The green group (wx'y) is essential because of m_{10}.

Covering the other minterms

- Finally, pick as few other prime implicants as necessary to ensure that all of the original minterms are included.

		y		
	m_0	m_1	m_3	m_2
	m_4	m_5	m_7	m_6
w	m_{12}	m_{13}	m_{15}	m_{14}
	m_8	m_9	m_{11}	m_{10}
		z		

x

		y		
	1	1	0	0
	1	1	0	0
w	0	1	1	0
	0	0	1	1
		z		

x

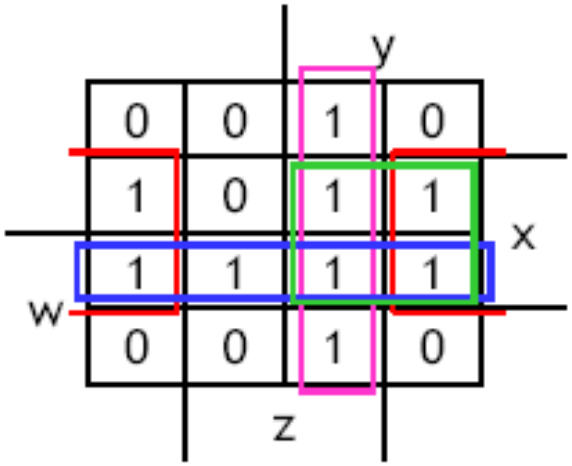
Example 7

- After choosing the red and green rectangles in our example, there are just two minterms remaining, m_{13} and m_{15}.
  - They are both included in the blue prime implicant, wxz.
  - The resulting MSP is w'y' + wxz + wx'y.
- The magenta and sky blue groups are not needed, since their minterms are already included by the other three prime implicants.

Solutions for practice K-map 2

- Simplify the following K-map.

Example 8

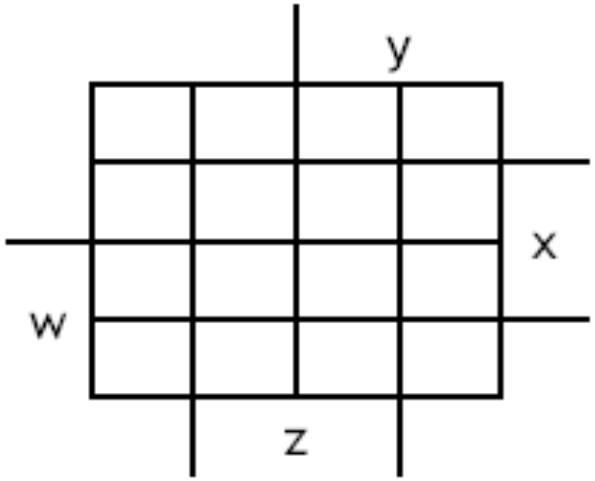


- All prime implicants are circled.
- The essential prime implicants are xz', wx and yz.
- The MSP is xz' + wx + yz. (Including the group xy would be redundant.)

Practice K-map 3

- Find a minimal sum of products for the following.

$f(w,x,y,z) = \Sigma m(0,2,4,5,8,14,15)$, $d(w,x,y,z) = \Sigma m(7,10,13)$

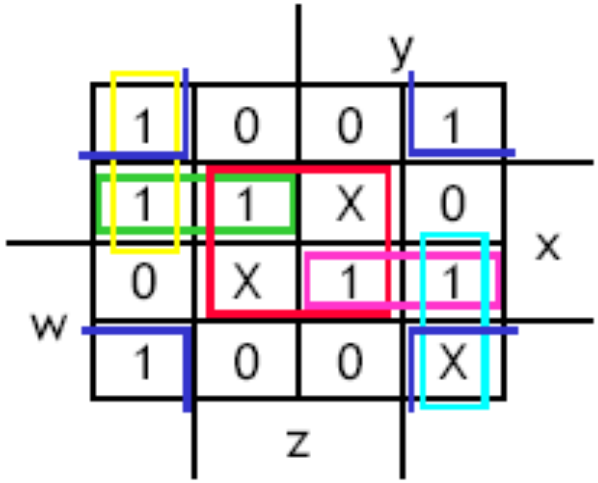


Example 9

Solutions for practice K-map 3

- Find a minimal sum of products for the following.

$$f(w,x,y,z) = \Sigma m(0,2,4,5,8,14,15),\ d(w,x,y,z) = \Sigma m(7,10,13)$$



Example 9

- All prime implicants are circled. We can treat Xs as 1s if we want, so the red group includes two Xs, and the light blue group includes one X.
- The *only* essential prime implicant is x'z'. The red group is not essential because the two minterms in it also appear in other groups.
- The MSP is x'z' + wxy + w'xy'. It turns out the red group is redundant; we can cover all of the minterms in the map without it.

Practice K-map 2

- Simplify the following K-map.

Example 8

			y	
	0	0	1	0
	1	0	1	1
w	1	1	1	1
	0	0	1	0
		z		

x

Conclusion

- Kmaps provide an easy graphical method of simplifying Boolean expressions.
- A Kmap is a matrix consisting of the outputs of the minterms of a Boolean function.
- In this section, we have discussed 2- 3- and 4-input Kmaps. This method can be extended to any number of inputs through the use of multiple tables.

Conclusion

Recapping the rules of Kmap simplification:

- Groupings can contain only 1s; no 0s.
- Groups can be formed only at right angles; diagonal groups are not allowed.
- The number of 1s in a group must be a power of 2 – even if it contains a single 1.
- The groups must be made as large as possible.
- Groups can overlap and wrap around the sides of the Kmap.
- Use don’t care conditions when you can.

College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 4

Signed Number Representation

Overview

- Unsigned Representation
- Representation of signed numbers
  - Signed Magnitude Representation
  - One's Complement Notation
  - Two's Complement Notation
- Two's Complement Addition
- Comparing Signed Number Systems
- Signed Overflow

Representations and algorithms

- Today we'll look at three different representations of signed numbers.
  - The best one will result in the simplest and fastest operations.
  - This is just like choosing a data structure in programming.
- We're mostly concerned with two particular operations.
  1. Negating a signed number, or finding -x from x.
  2. Adding two signed numbers, or computing x + y.

For Review

Unsigned Representation

- Represents positive integers.

  Ex: 8 bit representation of unsigned numbers:

position	7	6	5	4	3	2	1	0
contribution	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
157	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
10	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

- Addition is simple:

  0 0 0 0 1 0 0 1 + 0 0 0 0 0 1 0 1 = 0 0 0 0 1 1 1 0.

Binary Addition (1 of 2)

- Two 1-bit values

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	10

"two"

Binary Addition (2 of 2)

- Two *n*-bit values
  - Add individual bits
  - Propagate carries
  - E.g.,

```
   1     1
 00010101      21
+00011001    + 25
 ---------    ----
 00101110      46
```

Advantages and disadvantages of unsigned notation

Advantages:

One representation of zero

Simple addition

Disadvantages

Negative numbers can not be represented.

The need of different notation to represent negative numbers.

Representation of signed numbers

Is a representation of negative numbers possible?

Unfortunately:

you can not just stick a negative sign in front of a binary number. (it does not work like that)

There are three methods used to represent negative numbers.

- Signed magnitude notation
- One's complement notation
- Two's complement notation

Signed magnitude representation

- Humans use the signed-magnitude system. We add + or - to the front of a number to indicate its sign.
- We can do this in binary too, by adding a sign bit in front of our numbers.
  - A 0 sign bit represents a positive number.
  - A 1 sign bit represents a negative number.

$1101_2 = 13_{10}$		(a 4-bit unsigned number)
0 1101	$= +13_{10}$	(a positive number in 5-bit signed magnitude)
1 1101	$= -13_{10}$	(a negative number in 5-bit signed magnitude)

$0100_2 = 4_{10}$		(a 4-bit unsigned number)
0 0100	$= +4_{10}$	(a positive number in 5-bit signed magnitude)
1 0100	$= -4_{10}$	(a negative number in 5-bit signed magnitude)

Example

- Suppose 10011101 is a signed magnitude representation of a 8 bit number.
- The sign bit is 1, then the number represented is negative
- The magnitude is 0011101 with a value $2^4+2^3+2^2+2^0= 29$
- Then the number represented by 10011101 is –29.

position	7	6	5	4	3	2	1	0
contribution	-			2^4	2^3	2^2		2^0
-29	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**

Exercise 1

37_{10} has 0010 0101 in signed magnitude notation. Find the signed magnitude of -37_{10} in 8 bits?

Using the signed magnitude notation find the 8-bit binary representation of the decimal value 24_{10} and -24_{10}.

Find the signed magnitude of –63 using 8-bit binary sequence?

Disadvantage of Signed Magnitude

- Addition and subtractions are difficult:

  Signs and magnitude, both have to carry out the required operation.

- They are two representations of 0

  00000000 = $+ 0_{10}$

  10000000 = $- 0_{10}$

  To test if a number is 0 or not, the CPU will need to see whether it is 00000000 or 10000000.

  0 is always performed in programs.

  Therefore, having two representations of 0 is inconvenient.

Ones' complement representation

- In a different representation, ones' complement, we negate numbers by complementing each bit of the number.
- We keep the sign bits: 0 for positive numbers, and 1 for negative.
- The sign bit is complemented along with the rest of the bits.

1101_2	$= 13_{10}$	(a 4-bit unsigned number)
0 1101	$= +13_{10}$	(a positive number in 5-bit ones' complement)
1 0010	$= -13_{10}$	(a negative number in 5-bit ones' complement)

0100_2	$= 4_{10}$	(a 4-bit unsigned number)
0 0100	$= +4_{10}$	(a positive number in 5-bit ones' complement)
1 1011	$= -4_{10}$	(a negative number in 5-bit ones' complement)

Why is it called ones' complement?

- Complementing a single bit is equivalent to subtracting it from 1.

x	x'	1 - x
0	1	1
1	0	0

- Similarly, complementing each bit of an *n*-bit number is equivalent to subtracting that number from 2^n-1.
- For example, we can negate the 5-bit number 01101.
  - Here n=5, and 2^5-1 = 11111_2.
  - Subtracting 01101 from 11111 yields 10010.

$$\begin{array}{r} 1\ 1\ 1\ 1\ 1 \\ -\ 0\ 1\ 1\ 0\ 1 \\ \hline 1\ 0\ 0\ 1\ 0 \end{array}$$

Ones' complement addition

- There are two steps in adding ones' complement numbers.
  1. Do unsigned addition on the numbers, *including* the sign bits.
  2. Take the carry out and add it to the sum.

```
      0 1 1 1      (+7)
+     1 0 1 1    + (-4)
-------------   -------
    1 0 0 1 0

      0 0 1 0
+           1
-------------
      0 0 1 1      (+3)
```

```
      0 0 1 1      (+3)
+     0 0 1 0    + (+2)
-------------   -------
    0 0 1 0 1

      0 1 0 1
+           0
-------------
      0 1 0 1      (+5)
```

- This is simpler than signed magnitude addition, but still a bit tricky.

Two's Complement Notation

- The most used representation for integers.

  All positive numbers begin with 0.

  All negative numbers begin with 1.

- One representation of zero

  Ex: 0 is represented as 0000 using 4-bit binary sequence.

Two's complement representation

- Our final idea is two's complement. To negate a number, we complement each bit (just as for ones' complement) and then add 1.

$1101_2 = 13_{10}$		(a 4-bit unsigned number)
0 1101	$= +13_{10}$	(a positive number in 5-bit two's complement)
1 0010	$= -13_{10}$	(a negative number in 5-bit *ones'* complement)
1 0011	$= -13_{10}$	(a negative number in 5-bit two's complement)

$0100_2 = 4_{10}$		(a 4-bit unsigned number)
0 0100	$= +4_{10}$	(a positive number in 5-bit two's complement)
1 1011	$= -4_{10}$	(a negative number in 5-bit *ones'* complement)
1 1100	$= -4_{10}$	(a negative number in 5-bit two's complement)

More about two's complement

- Another way to negate an *n*-bit two's complement number is to subtract it from 2^n.

$$\begin{array}{rrl} & 100000 & \\ - & 01101 & (+13_{10}) \\ \hline & 10011 & (-13_{10}) \end{array} \qquad \begin{array}{rrl} & 100000 & \\ - & 00100 & (+4_{10}) \\ \hline & 11100 & (-4_{10}) \end{array}$$

- You can also complement all of the bits to the left of the rightmost 1.

01101 = $+13_{10}$ (a positive number in two's complement)
10011 = -13_{10} (a negative number in two's complement)

00100 = $+4_{10}$ (a positive number in two's complement)
11100 = -4_{10} (a negative number in two's complement)

Example:

11110101 in Two's Complement (8 bit number)

The most significant bit is 1, hence it is a negative number.

Corresponding number is 00001011 = 8 + 2+1 = 11

the result is then –11.

Two's complement addition

- Negating a two's complement number takes a bit of work, but addition is much easier than with the other two systems.
- To find A + B, you just have to do unsigned addition on A and B (including their sign bits), and *ignore* any carry out.
- For example, we can compute 0111 + 1100, or (+7) + (-4).
  - First add 0111 + 1100 as unsigned numbers.

$$
\begin{array}{r}
0\,1\,1\,1 \\
+\quad 1\,1\,0\,0 \\
\hline
1\;\;0\,0\,1\,1
\end{array}
$$

  - Ignore the carry out (1). The answer is 0011 (+3).

Another two's complement example

- To further convince you that this works, let's try adding two negative numbers—1101 + 1110, or (-3) + (-2) in decimal.
- Adding the numbers gives 11011.

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

- Dropping the carry out (1) leaves us with the answer, 1011 (-5).

An algebraic explanation

- For *n*-bit numbers, the negation of B in two's complement is 2^n - B. (This was one of the alternate ways of negating a two's complement number.)

$$\begin{aligned} A - B &= A + (-B) \\ &= A + (2^n - B) \\ &= (A - B) + 2^n \end{aligned}$$

- If $A \geq B$, then (A - B) has to be positive, and the 2^n represents a carry out of 1. Discarding this carry out leaves us with the desired result, (A - B).
- If $A < B$, then (A - B) must be negative, and 2^n - (A - B) corresponds to the correct result -(A - B) in two's complement form.

Advantages of Two's Complement Notation

- It is easy to add two numbers.
- Subtraction can be easily performed.
- Multiplication is just a repeated addition.
- Division is just a repeated subtraction
- Two's complement is widely used in ***ALU***

Comparing the signed number systems

- Here are all the 4-bit numbers in the different systems.
- Positive numbers are the same in all three representations.
- There are *two* ways to represent 0 in signed magnitude and ones' complement. This makes things more complicated.
- In two's complement, there is one more negative number than positive number. Here, we can represent -8 but not +8.
- However, two's complement is preferred because it has only one 0, and its addition algorithm is the simplest.

Decimal	SM	1C	2C
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	—
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	—	—	1000

Ranges of the signed number systems

- How many negative and positive numbers can be represented in each of the different four-bit systems on the previous page?

	Unsigned	SM	1C	2C
Smallest	0000 (0)	1111 (-7)	1000 (-7)	1000 (-8)
Largest	1111 (15)	0111 (+7)	0111 (+7)	0111 (+7)

- The ranges for general *n*-bit numbers (including the sign bit) are below.

	Unsigned	SM	1C	2C
Smallest	0	$-(2^{n-1}-1)$	$-(2^{n-1}-1)$	-2^{n-1}
Largest	2^n-1	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$

Signed overflow

- With 4-bit two's complement numbers, the largest representable decimal value is +7, and the smallest is -8.
- What if you try to compute 4 + 5, or (-4) + (-5)?

```
    0 1 0 0     (+4)
+   0 1 0 1   + (+5)
-----------   -------
  0 1 0 0 1     (-7)
```

```
    1 1 0 0     (-4)
+   1 0 1 1   + (-5)
-----------   -------
  1 0 1 1 1     (+7)
```

- Signed overflow is very different from unsigned overflow.
  - The carry out is not enough to detect overflow. In the example on the left, the carry out is 0 but there *is* overflow.

Detecting signed overflow

- The easiest way to detect signed overflow is to look at all the sign bits.

```
    0 1 0 0      (+4)        1 1 0 0      (-4)
+   0 1 0 1    + (+5)    +   1 0 1 1    + (-5)
-----------    --------    -----------    --------
  0 1 0 0 1      (-7)      1 0 1 1 1      (+7)
```

- Overflow occurs only in the two situations above.
  1. If you add two *positive* numbers and get a *negative* result.
  2. If you add two *negative* numbers and get a *positive* result.
- Overflow can never occur when you add a positive number to a negative number. (Do you see why?)

Overflow

Example1:

0100000

$$\begin{array}{rl} 0110101_2 & (= 53_{10}) \\ \underline{+0101010}_2 & (= 42_{10}) \\ 1011111_2 & (=-33_{10}) \end{array}$$

Example2:

1000000

$$\begin{array}{rl} 1010101_2 & (=-43_{10}) \\ \underline{+1001010}_2 & (=-54_{10}) \\ 0011111_2 & (= 31_{10}) \end{array}$$

Example3:

1100000

$$\begin{array}{rl} 0110101_2 & (= 53_{10}) \\ \underline{+1101010}_2 & (=-22_{10}) \\ 0011111_2 & (= 31_{10}) \end{array}$$

Example4:

0000000

$$\begin{array}{rl} 0010101_2 & (= 21_{10}) \\ \underline{+0101010}_2 & (= 42_{10}) \\ 0111111_2 & (= 63_{10}) \end{array}$$

Sign extension

- Decimal numbers are assumed to have an infinite number of 0s in front of them, which helps in "lining up" values for arithmetic operations.

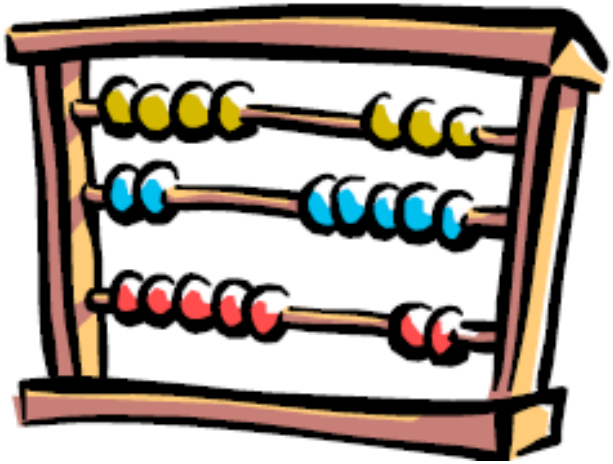
```
   2 2 5
+  0 0 6
---------
   2 3 1
```

- You need to be careful in extending signed binary numbers, because the leftmost bit is the *sign* and not part of the magnitude.
- To extend a signed binary number, you have to replicate the sign bit. If you just add 0s in front, you might accidentally change a negative number into a positive one!
- For example, consider going from 4-bit to 8-bit numbers.

(+5)	0101	→	0000 0101	(+5)
(-4)	1100	→	1111 1100	(-4)

Summary

- Data representations are all-important!
  - A good representation for negative numbers can make subtraction hardware much simpler to design.
  - Using two's complement, it's easy to build a single circuit for both addition and subtraction.
- Working with signed numbers involves several issues.
  - Signed overflow is very different from the unsigned overflow we talked about last week.
  - Sign extension is needed to properly "lengthen" negative numbers.



College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 5
COMBINATIONAL CIRCUITS-1

Unit 5: Combinational Circuits-1

Overview

- Introduction to Combinational Circuits
- Adder
- Ripple Carry Adder
- Subtraction
- Adder/Subtractor

Chapter-3

M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

Combinational circuits

- So far we've only worked with combinational circuits, where applying the same inputs always produces the same outputs.
  - This corresponds to a mathematical function, where every input has a single, unique output.
  - In programming terminology, combinational circuits are similar to "functional programs" that do not contain variables and assignments.
- Such circuits are comparatively easy to design and analyze.

Binary addition by hand

- You can add two binary numbers one column at a time starting from the right, just like you add two decimal numbers.
- But remember it's binary. For example, 1 + 1 = 10 and you have to carry!

The initial carry in is implicitly 0

	1	1	1	0		Carry in
		1	0	1	1	Augend
+		1	1	1	0	Addend
	1	1	0	0	1	Sum

most significant bit (MSB)

least significant bit (LSB)

Adder

- Design an Adder for 1-bit numbers?
- **1. Specification:**

  2 inputs (X,Y)

  2 outputs (C,S)

Adder

- Design an Adder for 1-bit numbers?
- **1. Specification:**

  2 inputs (X,Y)

  2 outputs (C,S)
- **2. Formulation:**

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Adder

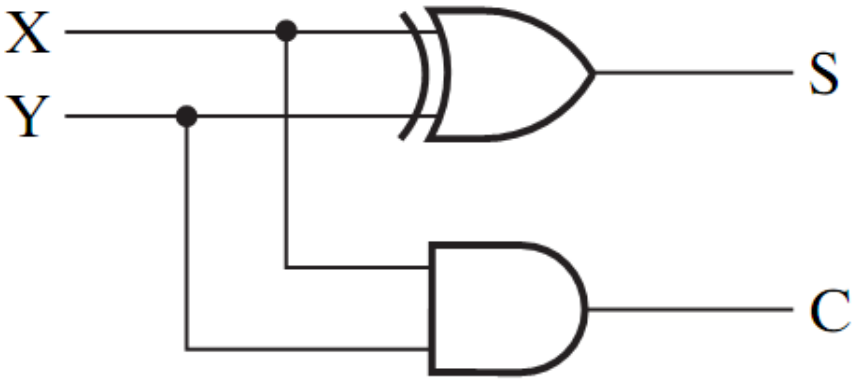
- Design an Adder for 1-bit numbers?
- **1. Specification:**

  2 inputs (X,Y)

  2 outputs (C,S)
- **2. Formulation:**

3. Optimization/Circuit

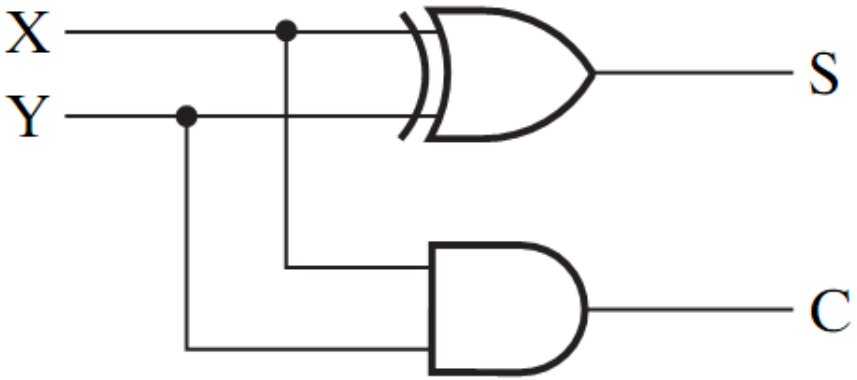
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Half Adder

- This adder is called a Half Adder
- **Q: Why?**

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Full Adder

- A combinational circuit that adds 3 input bits to generate a Sum bit and a Carry bit

- A truth table and sum of minterm equations for C and S are shown below.

	X	Y	Z	C	S
	0	0	0	0	0
	0	0	1	0	1
	0	1	0	0	1
0 + 1 + 1 = 10 →	0	1	1	1	0
	1	0	0	0	1
	1	0	1	1	0
	1	1	0	1	0
1 + 1 + 1 = 11 →	1	1	1	1	1

$C(X,Y,Z) = \Sigma m(3,5,6,7)$
$S(X,Y,Z) = \Sigma m(1,2,4,7)$

Full Adder

- A combinational circuit that adds 3 input bits to generate a Sum bit and a Carry bit

X	Y	Z	C	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

Sum

X \ YZ	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$S = X'Y'Z + X'YZ' + XY'Z' + XYZ$

$= X \oplus Y \oplus Z$

Carry

X \ YZ	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$C = XY + YZ + XZ$

Full Adder = 2 Half Adders

Manipulating the Equations:

$$S = X \oplus Y \oplus Z$$

$$C = XY + XZ + YZ$$

Full Adder = 2 Half Adders

Manipulating the Equations:

$$S = (X \oplus Y) \oplus Z$$

$$C = XY + XZ + YZ$$

$$= XY + XYZ + XY'Z + X'YZ + \cancel{XYZ}$$

$$= XY(1 + Z) + Z(XY' + X'Y)$$

$$= XY + Z(X \oplus Y)$$

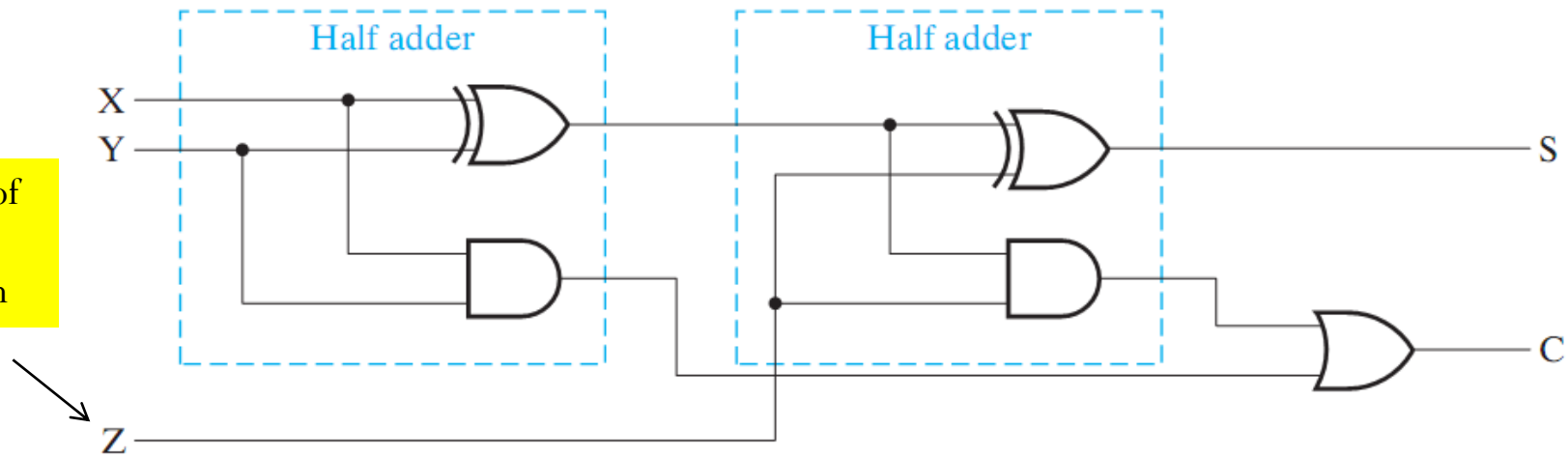
Full Adder = 2 Half Adders

Manipulating the Equations:

$$S = (X \oplus Y) \oplus Z$$

$$C = XY + XZ + YZ = XY + Z(X \oplus Y)$$

Think of Z as a carry in



Src: Mano's Book

Bigger Adders

- How to build an adder for n-bit numbers?
  - Example: 4-Bit Adder
    - Inputs ?
    - Outputs ?
    - What is the size of the truth table?
    - How many functions to optimize?

Bigger Adders

- How to build an adder for n-bit numbers?
  - Example: 4-Bit Adder
    - Inputs ? 9 inputs
    - Outputs ? 5 outputs
    - What is the size of the truth table? 512 rows!
    - How many functions to optimize? 5 functions

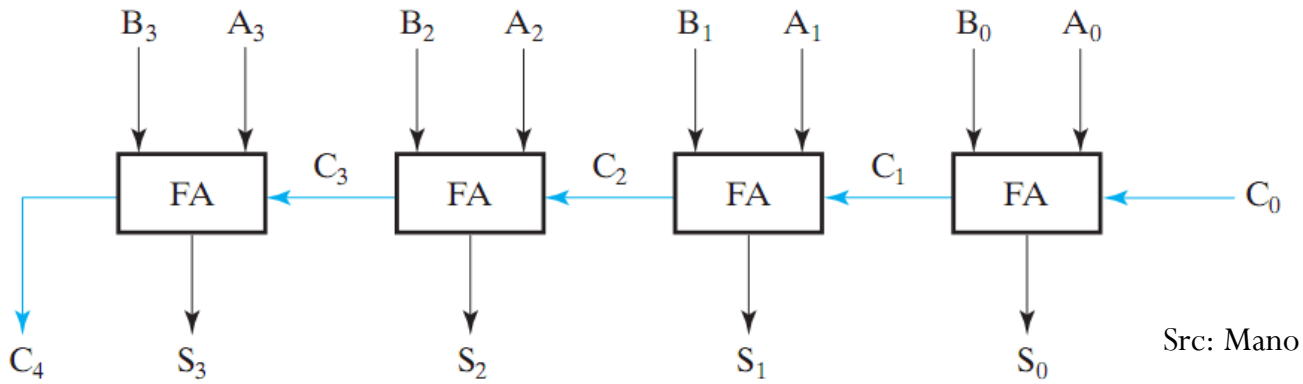
Ripple Carry Adder

- To add n-bit numbers:
- Use n Full-Adders in parallel
- The carries propagates as in addition by hand
- Use Z in the circuit as a C_{in}

```
  1 0 0 0
  0 1 0 1
  0 1 1 0
  -------
  1 0 1 1
```

Binary Parallel Adder

- To add n-bit numbers:
- Use n Full-Adders in parallel
- The carries propagates as in addition by hand



Src: Mano's Book

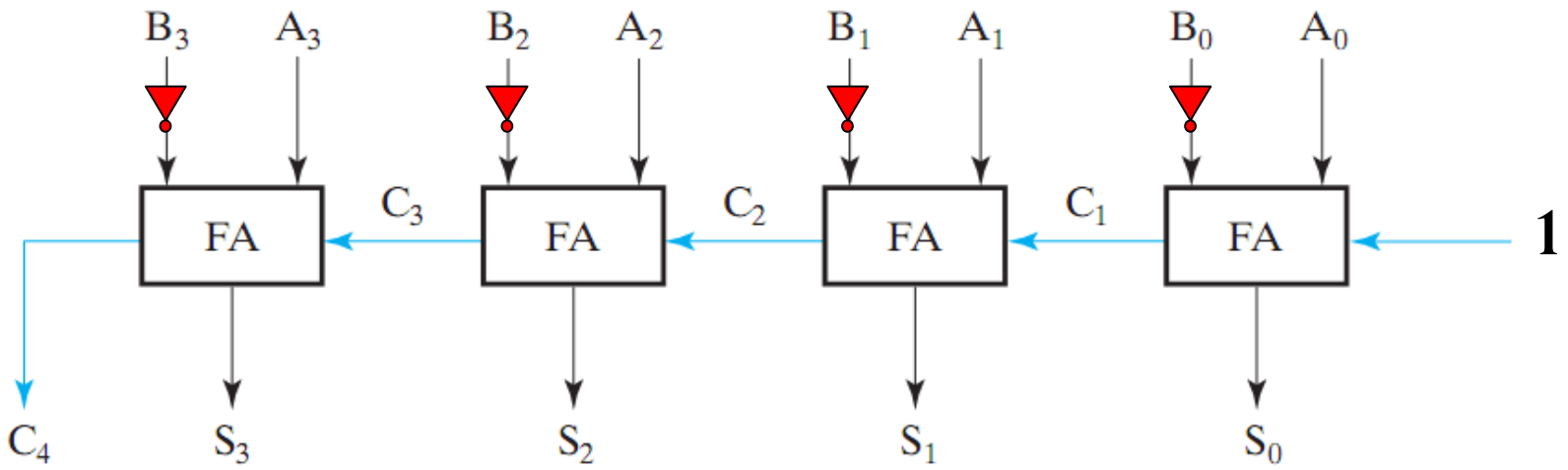
This adder is called ***ripple carry adder***

Subtraction (2's Complement)

- How to build a subtractor using 2's complement?

Subtraction (2's Complement)

- How to build a subtractor using 2's complement?



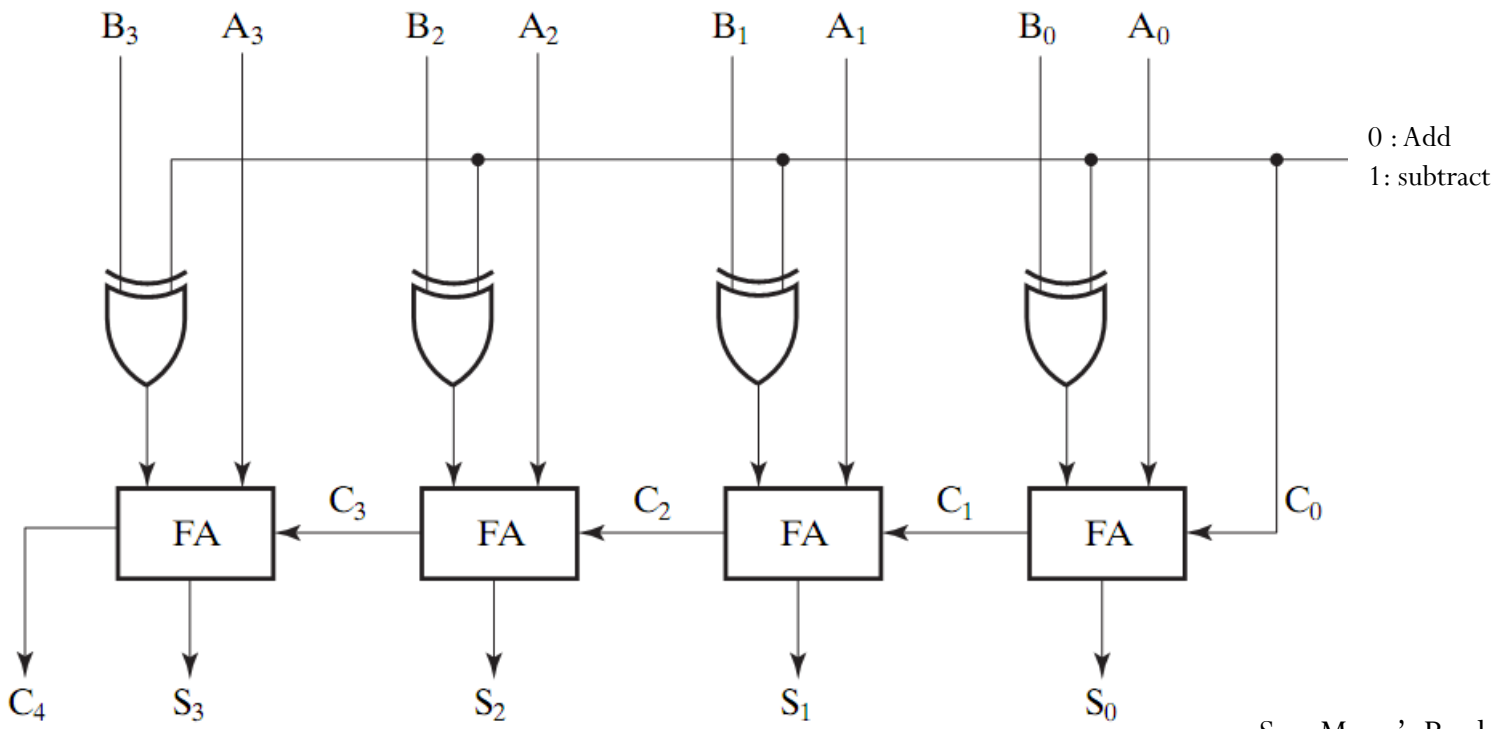
Src: Mano's Book

$$S = A + (-B)$$

Adder/Subtractor

- How to build a circuit that performs both addition and subtraction?

Adder/Subtractor

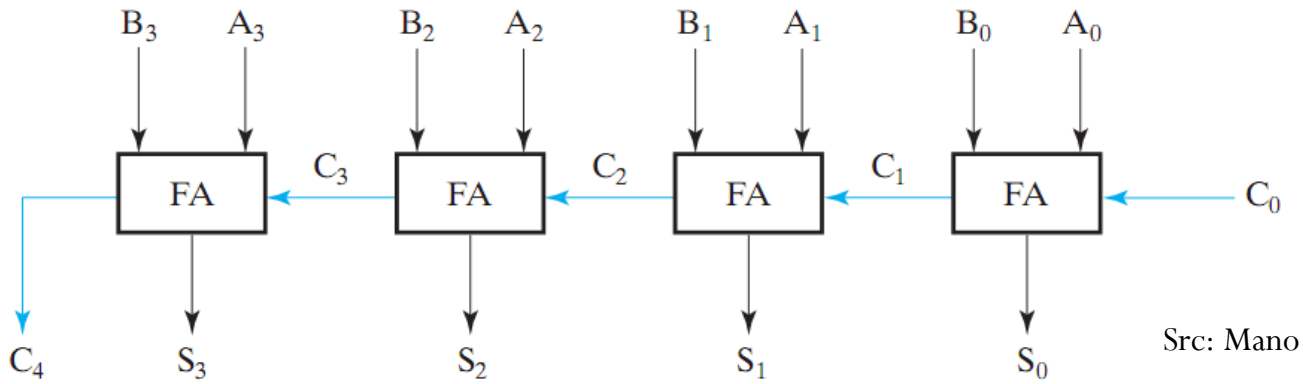


Src: Mano's Book

Using full adders and XOR we can build an Adder/Subtractor!

Binary Parallel Adder (Again)

- To add n-bit numbers:
- Use n Full-Adders in parallel
- The carries propagates as in addition by hand



Src: Mano's Book

This adder is called ***ripple carry adder***

Carry Look Ahead Adder

- How to reduce propagation delay of ripple carry adders?
- **Carry look ahead adder:** All carries are computed as a function of C_0 (independent of n !)
- It works on the following standard principles:
  - A carry bit is generated when both input bits Ai and Bi are 1, or
  - When one of input bits is 1, and a carry in bit exists

Carry bits → Cn Cn-1.....Ci.....C2C1C0

An-1.....Ai.....A2A1A0

Bn-1.....Bi.....B2B1B0

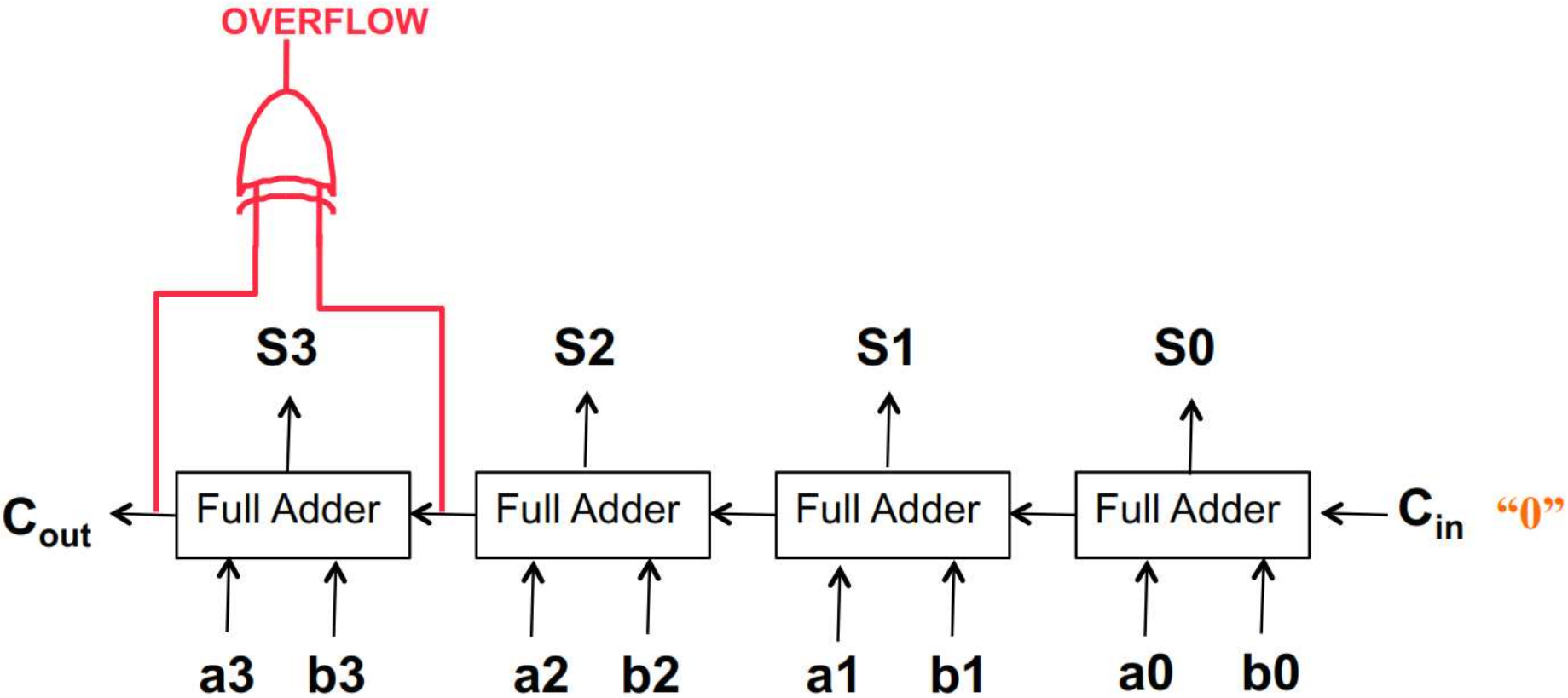
Carry Out → Sn Sn-1.....Si.....S2S1S0

Detecting signed overflow

- The easiest way to detect signed overflow is to look at all the sign bits.

$$\begin{array}{rr} & 0\,1\,0\,0 \\ + & 0\,1\,0\,1 \\ \hline & 0\,1\,0\,0\,1 \end{array} \qquad \begin{array}{r} (+4) \\ +\ (+5) \\ \hline (-7) \end{array} \qquad\qquad \begin{array}{rr} & 1\,1\,0\,0 \\ + & 1\,0\,1\,1 \\ \hline & 1\,0\,1\,1\,1 \end{array} \qquad \begin{array}{r} (-4) \\ +\ (-5) \\ \hline (+7) \end{array}$$

- Overflow occurs only in the two situations above.
  1. If you add two *positive* numbers and get a *negative* result.
  2. If you add two *negative* numbers and get a *positive* result.
- Overflow can never occur when you add a positive number to a negative number. (Do you see why?)


OVERFLOW
S3
S2
S1
S0
Cout
Full Adder
Full Adder
Full Adder
Full Adder
Cin
"0"
a3
b3
a2
b2
a1
b1
a0
b0

College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 6
COMBINATIONAL CIRCUITS-2

Unit 6: Combinational Circuits-2

Overview

Multiplexers

DeMultiplexers

Decoders

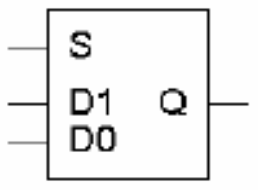
Encoders

Chapter-3

M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

A 2-to-1 multiplexer

- Here is the circuit analog of that printer switch.



- This is a 2-to-1 multiplexer, or mux.
  - There are two data inputs D0 and D1, and a select input called S.
  - There is one output named Q.
- The multiplexer routes one of its data inputs (D0 or D1) to the output Q, based on the value of S.
  - If S=0, the output will be D0.
  - If S=1, the output will be D1.

Building a multiplexer

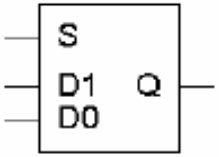
- Here is a truth table for the multiplexer, based on our description from the previous page:

> The multiplexer routes one of its data inputs (D0 or D1) to the output Q, based on the value of S.
>
> — If S=0, the output will be D0.
>
> — If S=1, the output will be D1.

S	D1	D0	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

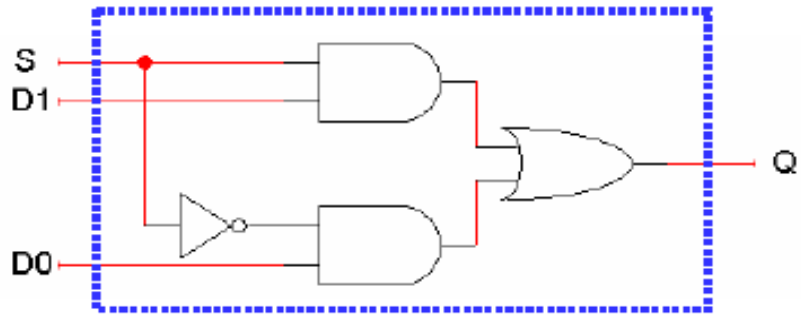
- You can then find an MSP for the mux output Q.

- Note that this corresponds closely to our English specification above—sometimes you can derive an expression without first making a truth table.



Multiplexer circuit diagram

- Here is an implementation of a 2-to-1 multiplexer.



$Q = S'D0 + S\,D1$

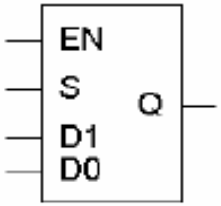
- Remember that a minimal sum of products expression leads to a minimal two-level circuit.

Blocks, abstraction and modularity

- Multiplexers are common enough that we often want to treat them as abstract units or black boxes, as symbolized by our block diagrams.
  - Block symbols make circuit diagrams simpler, by hiding the internal implementation details. You can use a device without knowing how it's designed, as long as you know what it does.
  - Different multiplexer implementations should be interchangeable.
  - Circuit blocks also aid hardware re-use, since you don't have to keep building a multiplexer from scratch every time you need one.
- These blocks are similar to functions in programming languages!

Enable inputs

- Many devices have an additional enable input, which "activates" or "deactivates" the device.
- We could design a 2-to-1 multiplexer with an enable input that's used as follows.
  - EN=0 disables the multiplexer, which forces the output to be 0. (It does *not* turn off the multiplexer.)
  - EN=1 enables the multiplexer, and it works as specified earlier.
- Enable inputs are especially useful in combining smaller muxes together to make larger ones, as we'll see later today.



EN	S	D1	D0	Q
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Truth table abbreviations

EN	S	D1	D0	Q
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

- Notice that when EN=0, then Q is always 0, regardless of what S, D1 and D0 are set to.
- We can shorten the truth table by including Xs in the input variable columns, as shown on the bottom right.

EN	S	D1	D0	Q
0	x	x	x	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Another abbr. 4 U

- Also, when EN=1 notice that if S=0 then Q=D0, but if S=1 then Q=D1.
- Another way to abbreviate a truth table is to list input variables in the output columns, as shown on the right.

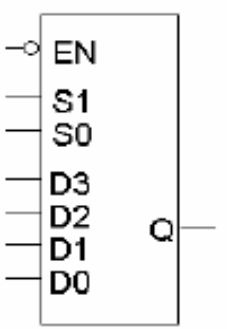
EN	S	D1	D0	Q
0	x	x	x	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

EN	S	Q
0	x	0
1	0	D0
1	1	D1

- This final version of the 2-to-1 multiplexer truth table is much clearer, and matches the equation Q = S'D0 + S D1 very closely.

A 4-to-1 multiplexer

- Here is a block diagram and abbreviated truth table for a 4-to-1 mux, which directs one of four different inputs to the single output line.
  - There are four data inputs, so we need *two* bits, S1 and S0, for the mux selection input.
  - LogicWorks multiplexers have active-low enable inputs, so the mux always outputs 1 when EN' = 1. This is denoted on the block symbol with a bubble in front of EN.

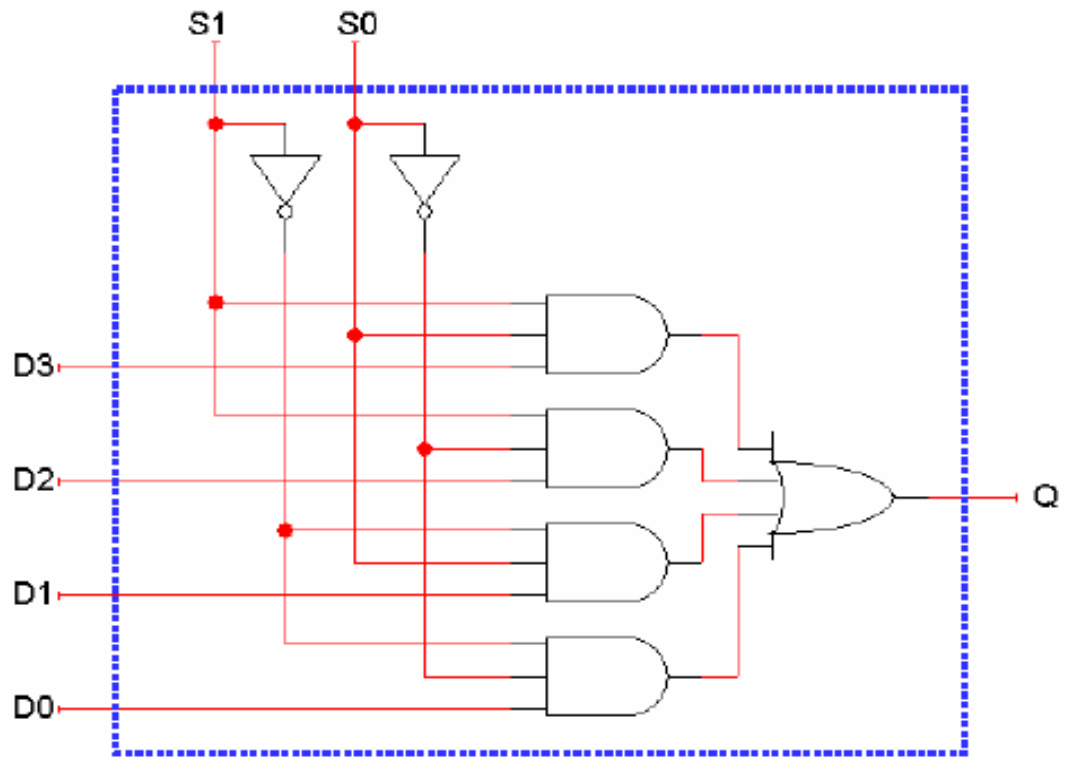


EN'	S1	S0	Q
0	0	0	D0
0	0	1	D1
0	1	0	D2
0	1	1	D3
1	x	x	1

$$Q = S1'S0'D0 + S1'S0\ D1 + S1\ S0'D2 + S1\ S0\ D3$$

A 4-to-1 multiplexer implementation

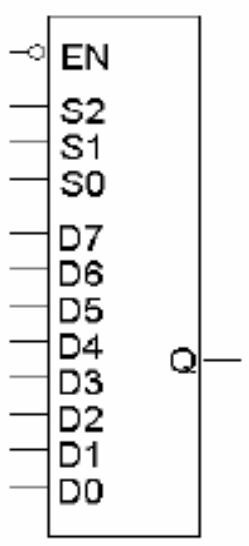
- Again we have a minimal sum of products expression, which leads to a minimal two-level circuit implementation.



$$Q = S1'S0'D0 + S1'S0\,D1 + S1\,S0'D2 + S1\,S0\,D3$$

2^n-to-1 multiplexers

- You can make even larger multiplexers, following the same pattern.
- A 2^n-to-1 multiplexer routes one of 2^n input lines to the output line.
  - There are 2^n data inputs, so there must also be *n* select inputs.
  - The output is a single bit.
- Here is an 8-to-1 multiplexer, probably the biggest we'll see in this class.



Implementing Functions with Multiplexers

Example: addition

- Multiplexers can sometimes make circuit design easier.
- As an example, let's make a circuit to add three 1-bit inputs X, Y and Z.
- We'll need two bits to represent the total.
  - The bits will be called C and S, standing for "carry" and "sum."
  - These are two separate functions of the inputs X, Y and Z.
- A truth table and sum of minterm equations for C and S are shown below.

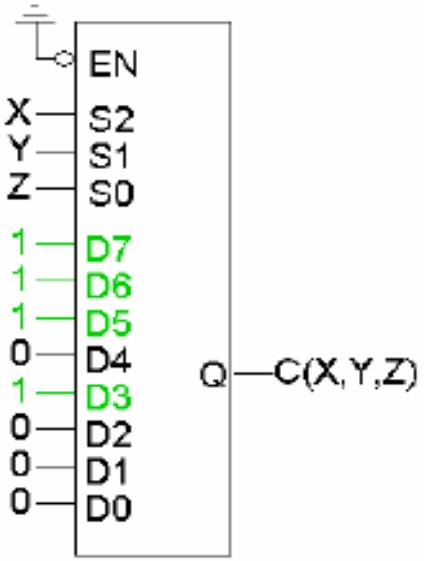
	X	Y	Z	C	S
	0	0	0	0	0
	0	0	1	0	1
	0	1	0	0	1
0 + 1 + 1 = 10 →	0	1	1	1	0
	1	0	0	0	1
	1	0	1	1	0
	1	1	0	1	0
1 + 1 + 1 = 11 →	1	1	1	1	1

$C(X,Y,Z) = \Sigma m(3,5,6,7)$
$S(X,Y,Z) = \Sigma m(1,2,4,7)$

	X	Y	Z	C	S
	0	0	0	0	0
	0	0	1	0	1
	0	1	0	0	1
0 + 1 + 1 = 10 →	0	1	1	1	0
	1	0	0	0	1
	1	0	1	1	0
	1	1	0	1	0
1 + 1 + 1 = 11 →	1	1	1	1	1

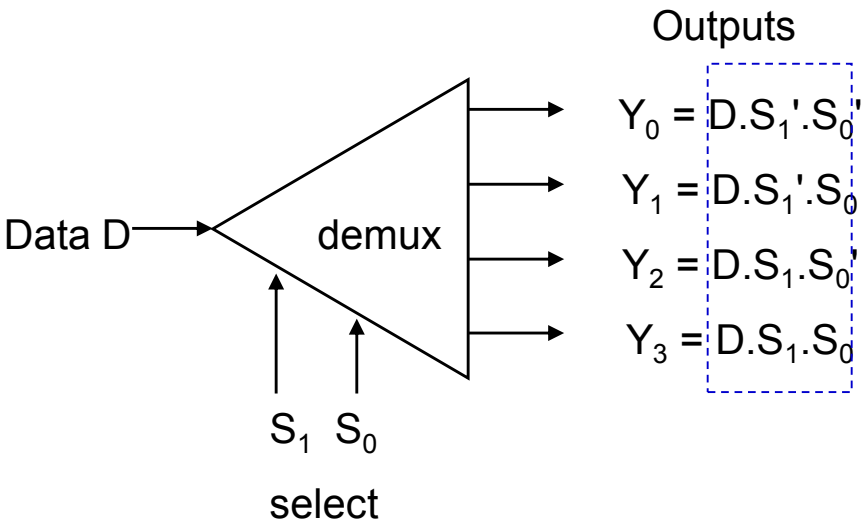
$C(X,Y,Z) = \Sigma m(3,5,6,7)$
$S(X,Y,Z) = \Sigma m(1,2,4,7)$

X	Y	Z	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Demultiplexer

- Given an input line and a set of selection lines, the demultiplexer will direct data from input to a selected output line.
- An example of a 1-to-4 demultiplexer:



S_1	S_0	Y_0	Y_1	Y_2	Y_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

Demultiplexer

- Takes one input
- Out to one of 2^n possible outputs

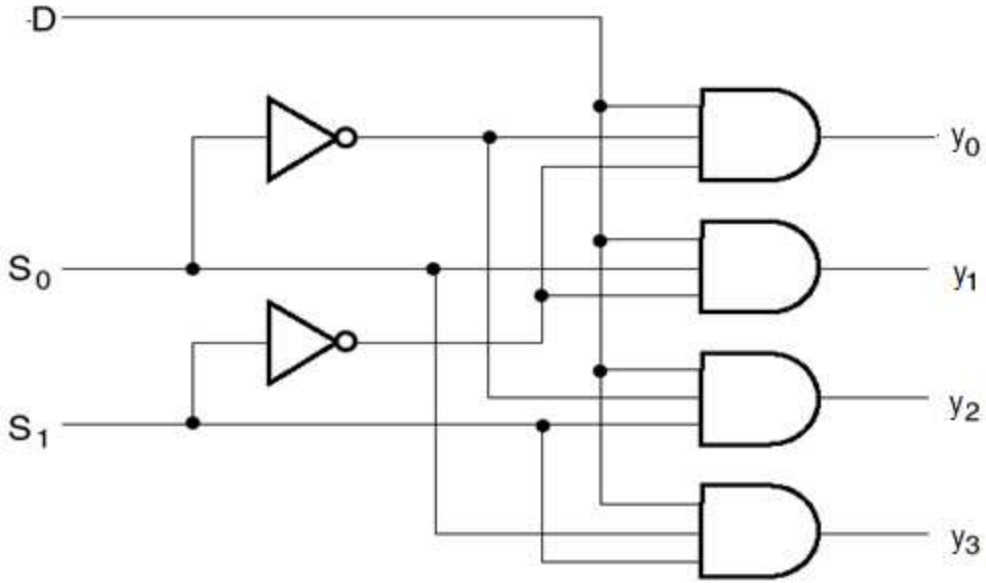
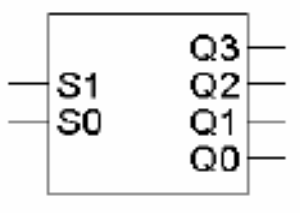


Fig. 3-24 1-to-4-Line Demultiplexer

Decoder

What a decoder does

- A *n*-to-2^n decoder uses its *n*-bit input to determine which of 2^n outputs will be uniquely activated.



S1	S0	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- Here is a block diagram and truth table for a 2-to-4 decoder.
  - The two-bit input is called S1S0, and the four outputs are Q0-Q3.
  - If the input is the binary number i, then output Qi alone will be true.
- This circuit "decodes" a binary number into a "one-of-four" code.

Building a decoder

- We can use the truth table to derive minimal sum of products equations for each of the four outputs (Q0-Q3), based on the two inputs (S0-S1).

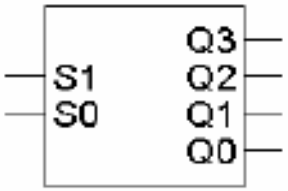
S1	S0	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- In this case there's not much to be simplified. Here are the equations:

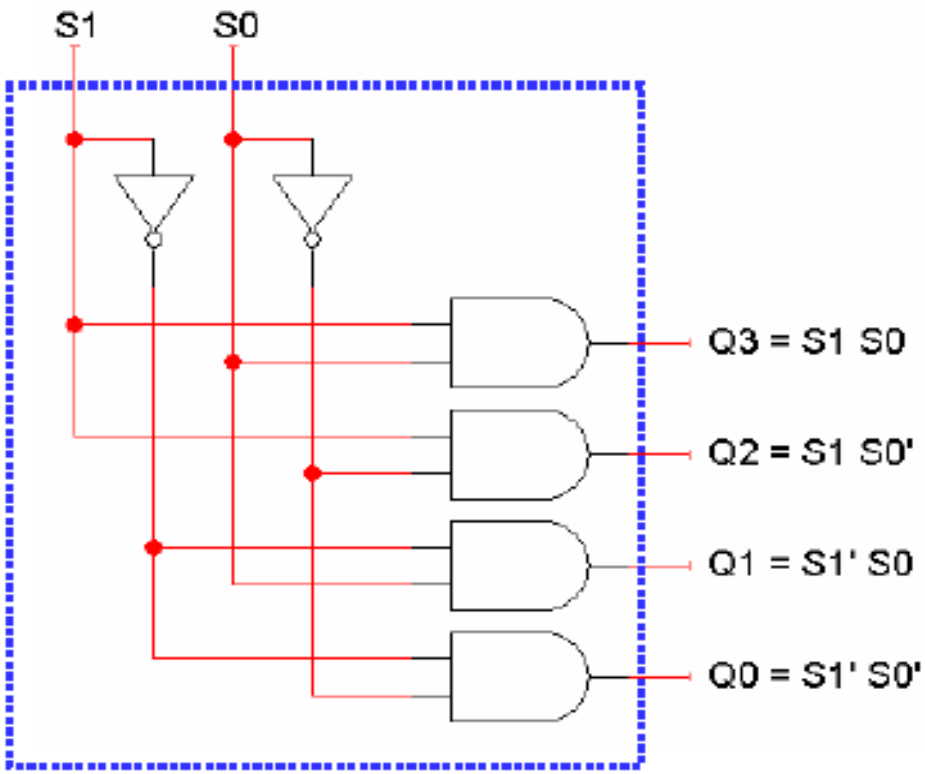
Q0 = S1'S0'
Q1 = S1'S0
Q2 = S1 S0'
Q3 = S1 S0

Decoder circuit diagram

- Here is an implementation of a 2-to-4 decoder.

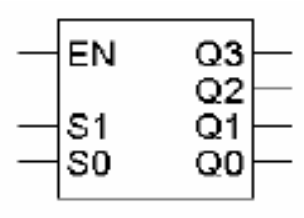


S1	S0	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



Enable inputs

- Just as with multiplexers, decoders can include enable inputs.
  - EN=0 disables the decoder, which by convention means that all of the decoder's outputs are 0.
  - EN=1 enables the decoder so that it behaves as specified earlier, with exactly one of the outputs being 1.



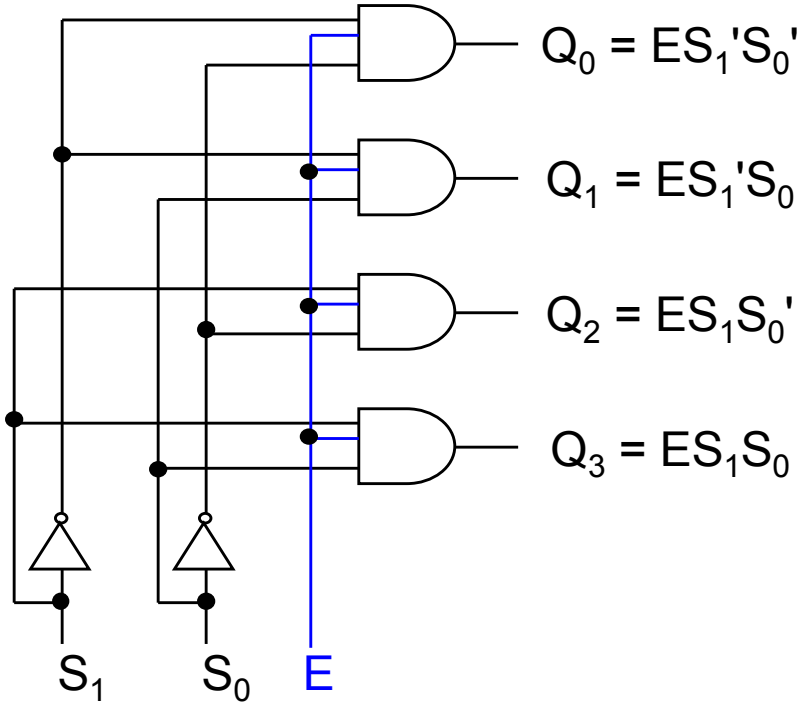
EN	S1	S0	Q0	Q1	Q2	Q3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Decoders with Enable (1/2)

- Decoders often come with an enable signal, so that the device is only activated when the enable, E=1.
- Truth table:

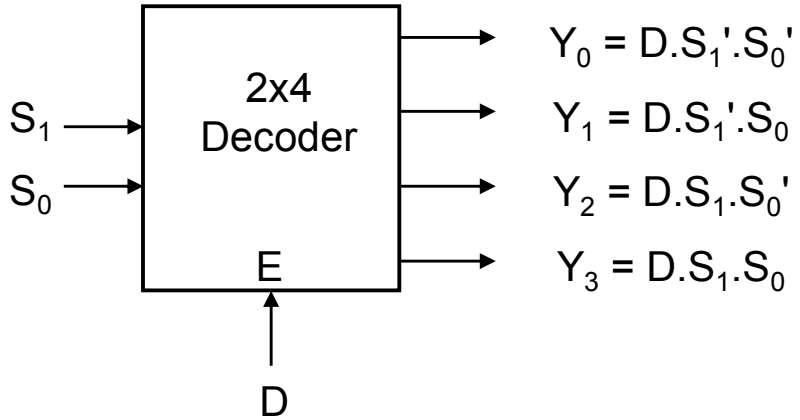
E	S_1	S_0	Q_0	Q_1	Q_2	Q_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	X	X	0	0	0	0

- Circuit:



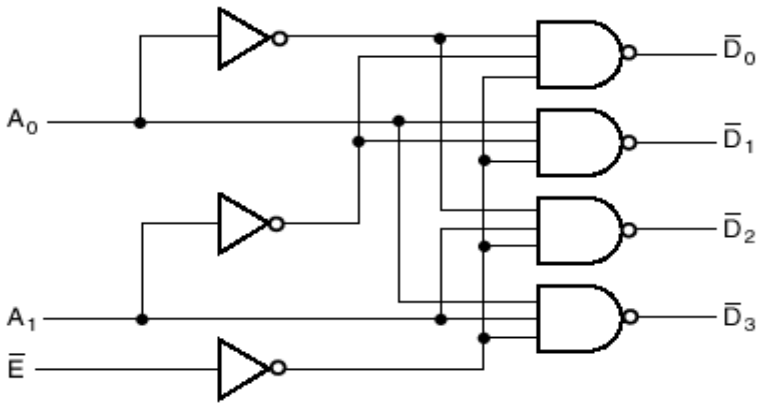
Demultiplexer Vs Decoder

- **The demultiplexer is actually identical to a decoder with enable, as illustrated below:**



Exercise: Provide the truth table for above demultiplexer.

A Demux Using NAND Gates: A Decoder with an Enable

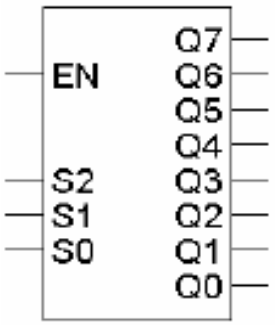


(a) Logic diagram

$\overline{E}$	A_1	A_0	$\overline{D}_0$	$\overline{D}_1$	$\overline{D}_2$	$\overline{D}_3$
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	X	X	1	1	1	1

A 3-to-8 decoder

- Larger decoders are similar. Here is a 3-to-8 decoder.
  - There are three selection inputs S2S1S0, which activate one of eight outputs, Q0-Q7.
  - Again, only one output will be true for any input combination.
- A truth table and output equations for a 3-to-8 decoder (without EN) are given below.



S2	S1	S0	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Q0 = S2'S1'S0'
Q1 = S2'S1'S0
Q2 = S2'S1 S0'
Q3 = S2'S1 S0
Q4 = S2 S1'S0'
Q5 = S2 S1'S0
Q6 = S2 S1 S0'
Q7 = S2 S1 S0

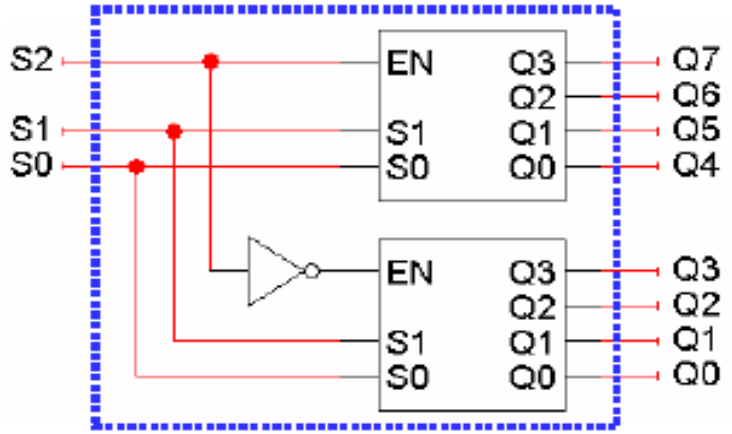
Building a 3-to-8 decoder

- You could build a 3-to-8 decoder from the truth table and MSP equations below, just like we built the 2-to-4 decoder earlier.
- Another way to design a decoder is to break it into smaller pieces.
- Notice some patterns in the table below:
  - When S2 = 0, outputs Q0-Q3 are generated as in a 2-to-4 decoder.
  - When S2 = 1, outputs Q4-Q7 are generated as in a 2-to-4 decoder.

S2	S1	S0	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Decoder expansion

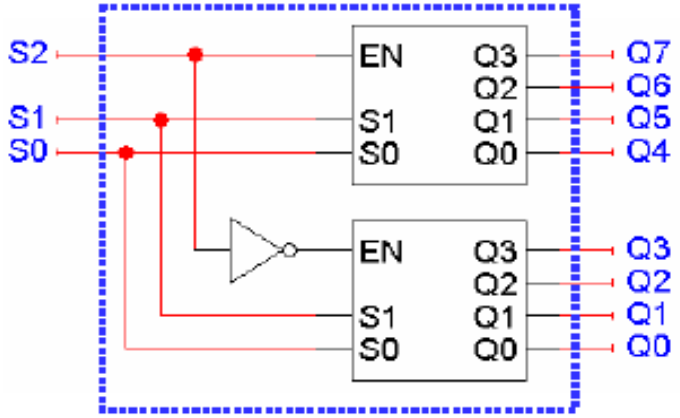
- Here's a 3-to-8 decoder built from two smaller 2-to-4 decoders.
- When S2=0, the bottom 2-to-4 decoder is enabled and generates a 1 for one of outputs Q0, Q1, Q2 or Q3.
- When S2=1, the top 2-to-4 decoder is enabled instead, and a 1 will be output for either Q4, Q5, Q6 or Q7.



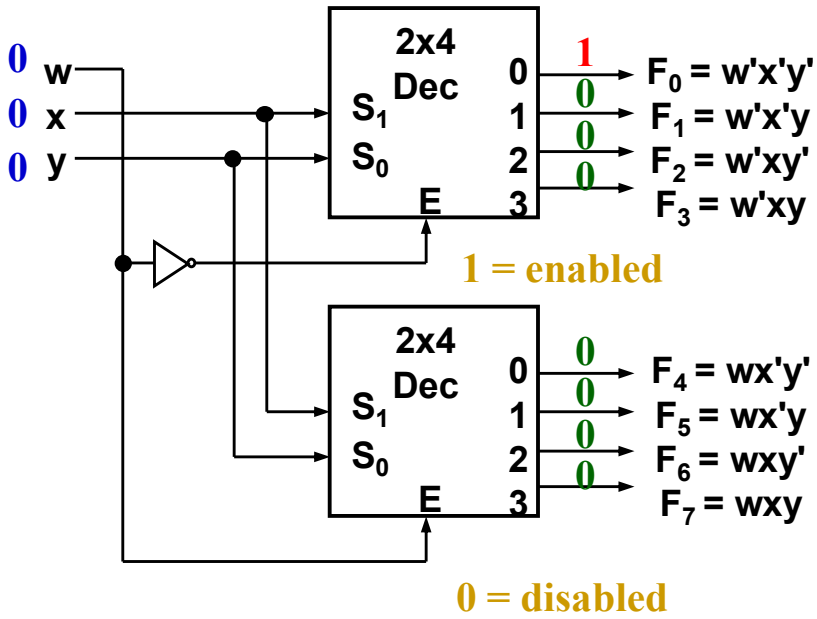
S2	S1	S0	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

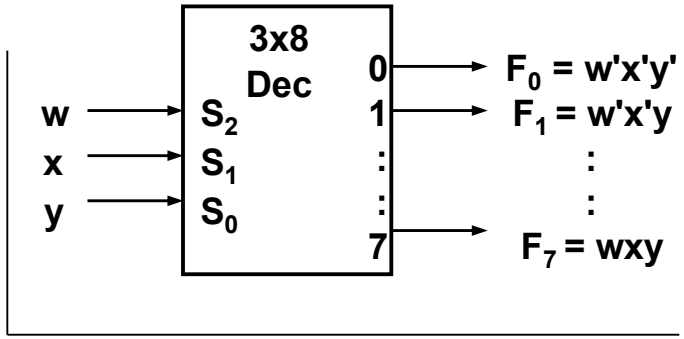
Modularity

- You could verify that this circuit is a 3-to-8 decoder, by using equations for the 2-to-4 decoders to derive equations for the 3-to-8.
- Be careful not to confuse the "inner" inputs and outputs of the 2-to-4 decoders with the "outer" inputs and outputs of the 3-to-8 decoder.
- This is similar to having several functions in a program which all use a formal parameter "x".

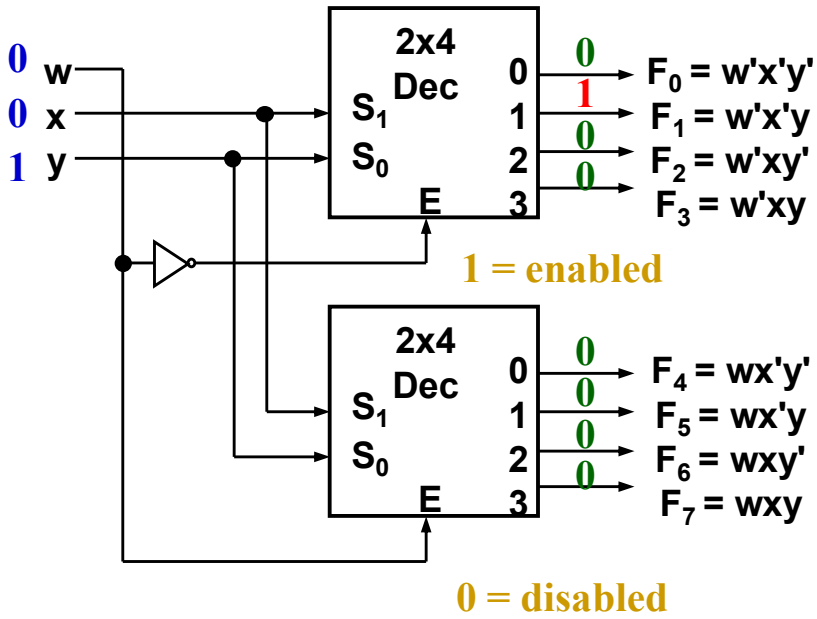


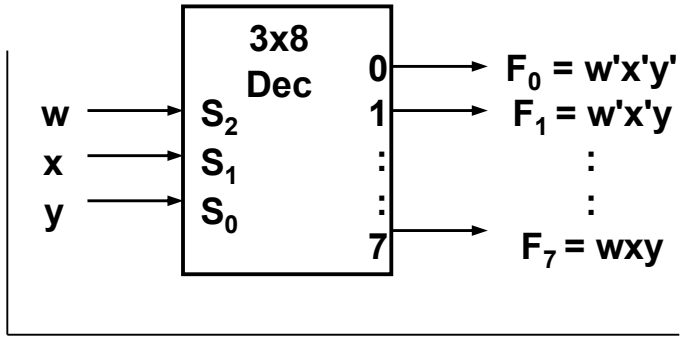
Larger Decoders (2/6)



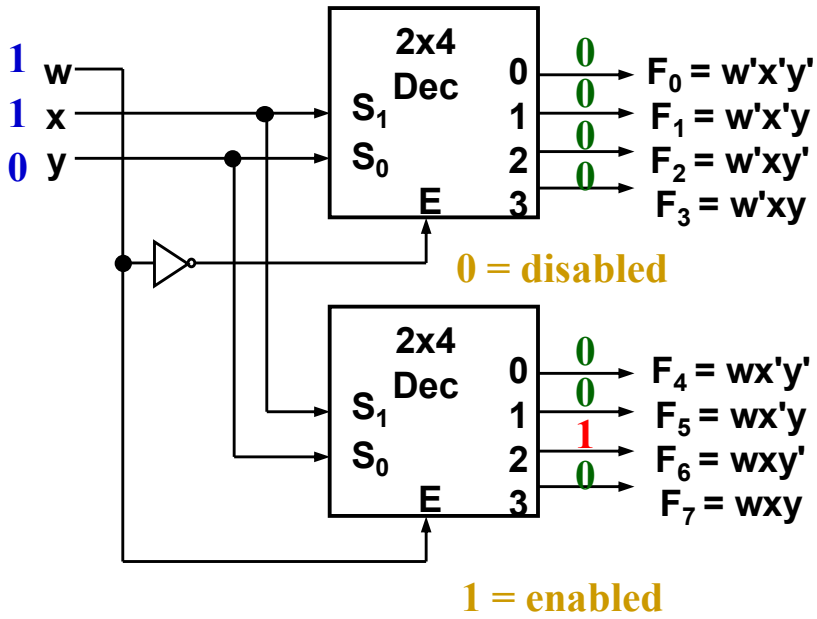


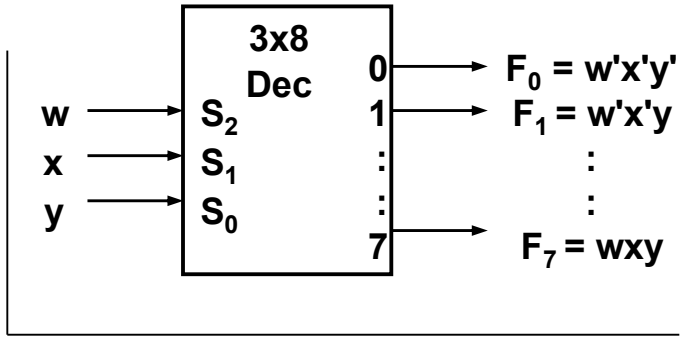
Larger Decoders (3/6)





Larger Decoders (4/6)

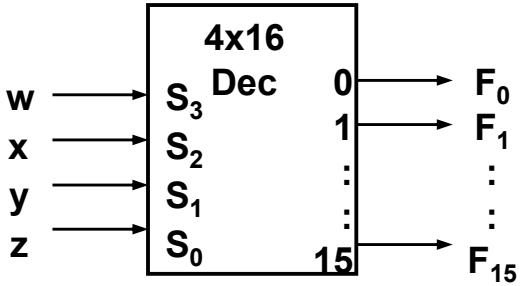


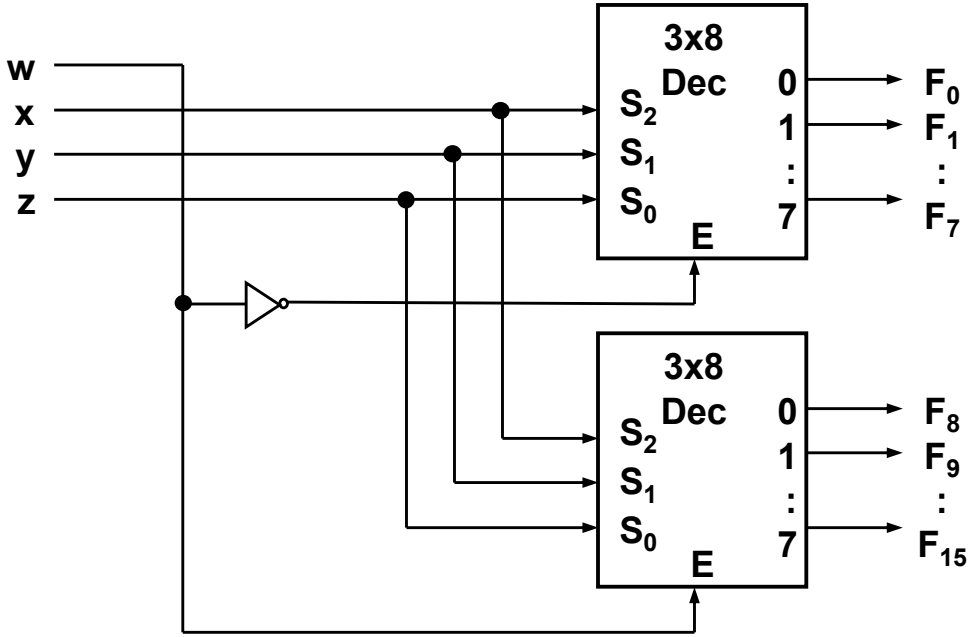


Larger Decoders (5/6)

- Question:

Construct a 4x16 decoder from two 3x8 decoders with 1-enable.





So what good is a decoder?

- Do the truth table and equations look familiar?

S1	S0	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Q0 = S1'S0'
Q1 = S1'S0
Q2 = S1 S0'
Q3 = S1 S0

- Decoders are sometimes called minterm generators.
  - For each input combination, exactly one output is true.
  - Each output equation contains all of the input variables.
- This means that you can easily use a decoder, or a minterm generator, to implement any sum of minterms expression.

Implementing Functions with Decoder

Example: addition

- Yesterday we presented a simple circuit which added three 1-bit inputs X, Y and Z to produce a two-bit output, C ("carry") and S ("sum").
- A truth table and sum of minterm equations for C and S are shown below.

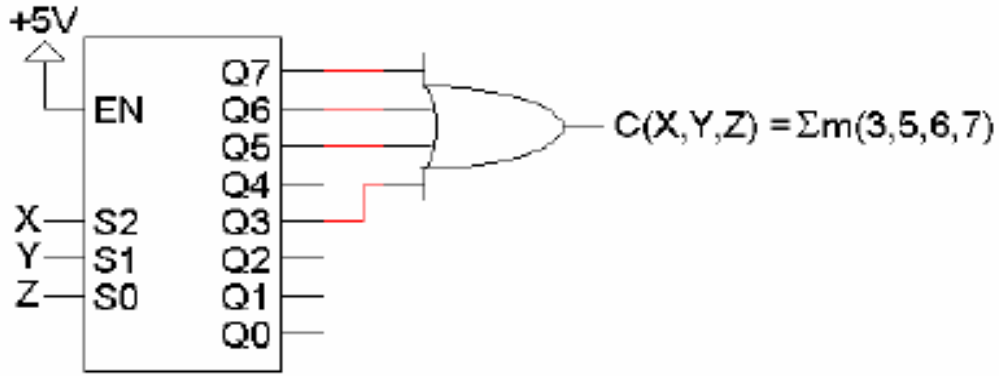
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$C(X,Y,Z) = \Sigma m(3,5,6,7)$
$S(X,Y,Z) = \Sigma m(1,2,4,7)$

- Today we'll implement these two functions using 3-to-8 decoders.

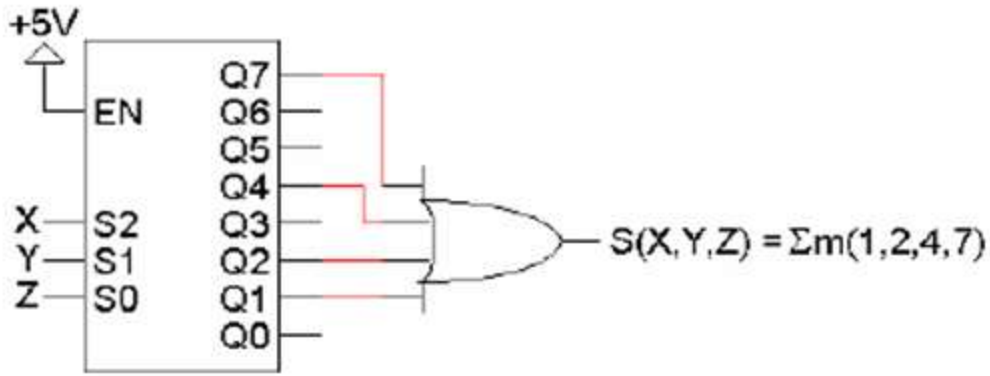
Implementing functions with decoders

- Here, a 3-to-8 decoder implements C as a sum of minterms.



- If XYZ is 011, 101, 110 or 111, then one of the decoder outputs Q3, Q5, Q6 or Q7 will be true, and the output C(X,Y,Z) will also be true.
- The "+5V" symbol ("5 volts") represents a 1 or true in LogicWorks.

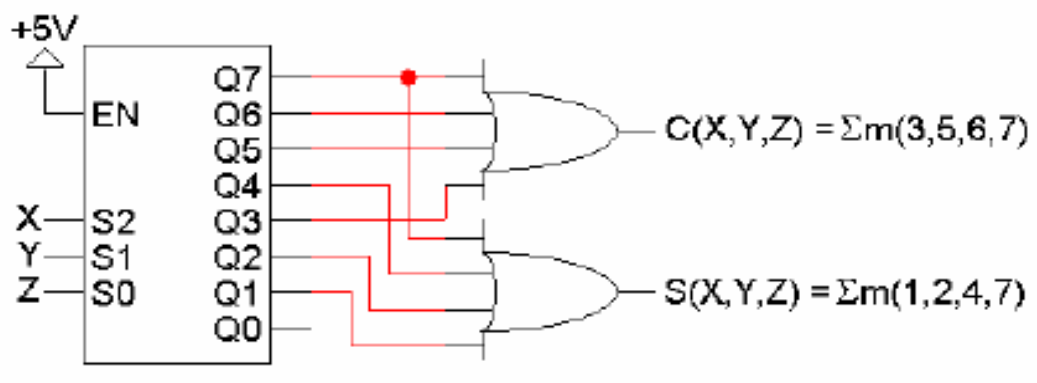
Decoder-based sum



- If XYZ is 001, 010, 100 or 111, then one of decoder outputs Q1, Q2, Q4 or Q7 will be true, and S(X,Y,Z) will be true as well.

Using just one decoder

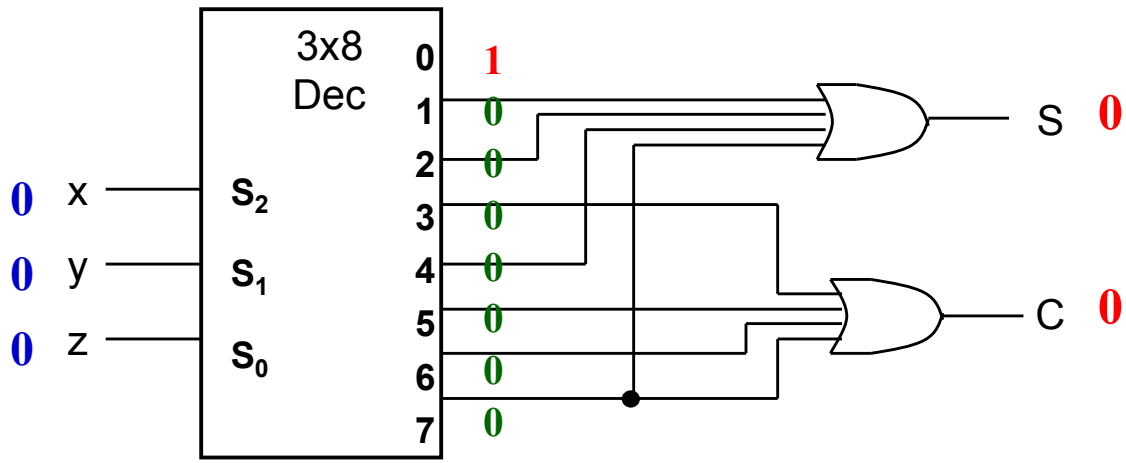
- Since the two functions C and S both have the same inputs, we could use just one decoder instead of two.



- Decoder output Q0 is unused, while Q7 is used multiple times. In general, you can always use circuit outputs as many or as few times as you need.

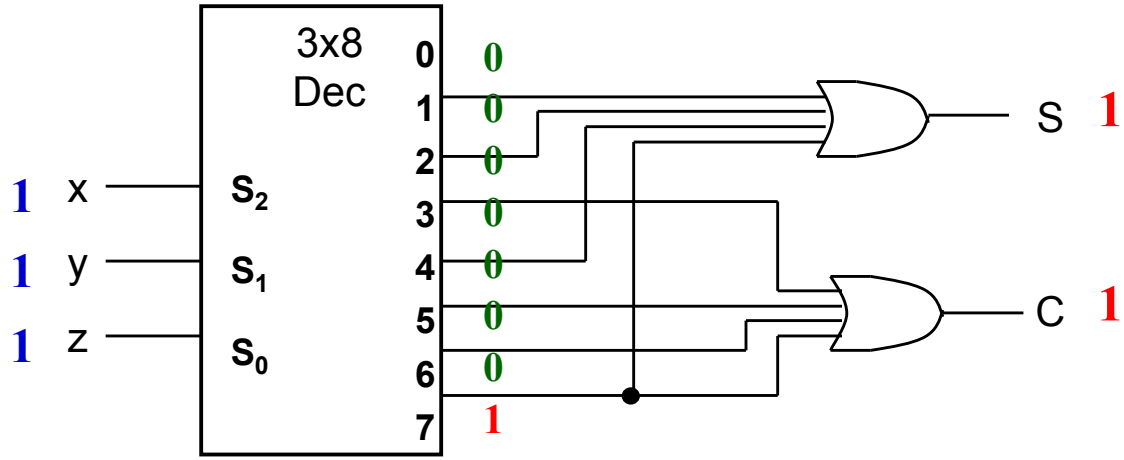
Decoders: Implementing Functions

Example: Full adder



x	y	z	C	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

Decoders: Implementing Functions

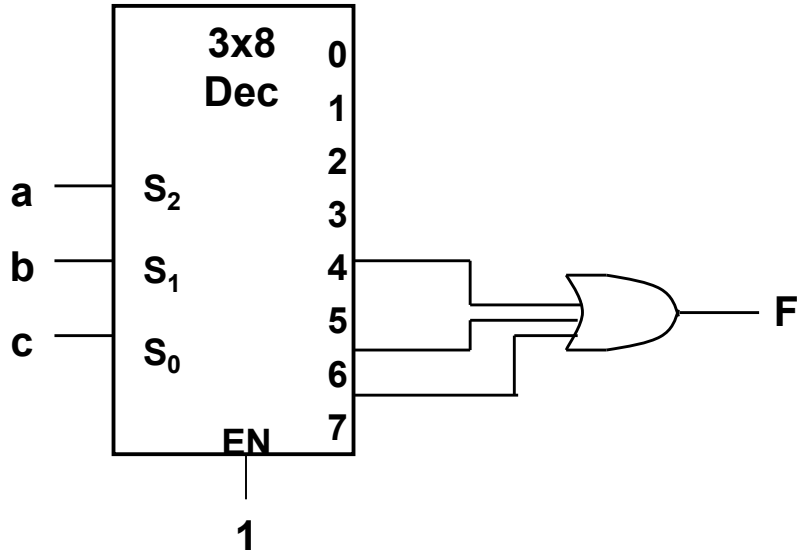


x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- Example:

$$F(a,b,c) = \sum m(4,6,7)$$

- Using a 3×8 decoder (assuming 1-enable and active-high outputs).



Encoder

- Encoder is the opposite of decoder
- **2^n inputs** or less and **n outputs**
- Example: **Decimal to BCD**
  - 10 inputs : $I_0, I_1, I_2, I_3, \ldots, I_9$
  - 4 output lines

Example: Designing an **Octal to binary** Encoder

Inputs: 8 (D_7 ... D_0)

Outputs: 3 (A_2 ... A_0)

Truth Table:

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Inputs are Minterms

- Can OR the minterms appropriately to get each of the outputs A_0, A_1, A_2
- Example: $A_0 = D_1 + D_3 + D_5 + D_7$

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Generating Outputs using OR of Minterms

- $A_0 = D_1 + D_3 + D_5 + D_7$
- $A_1 = D_2 + D_3 + D_6 + D_7$
- $A_2 = D_4 + D_5 + D_6 + D_7$

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

43

College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 7
Sequential Circuits
(Flip Flop)

Unit 7: Sequential Circuits

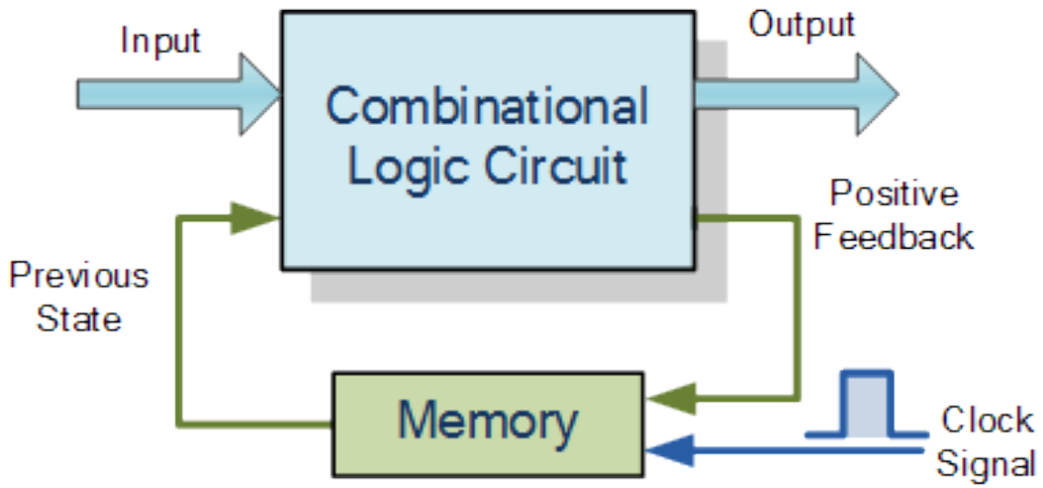
Overview

- SR Latch
  - S′R′ Latch
  - SR Latch with Control Input
  - D Latch
- Clock and Synchronization
- Flip-flops
  - Positive edge-triggered D Flip-flop
  - JK &T Flip-flop
  - Direct Inputs

Chapter-4

M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

Sequential circuits



- In contrast, the outputs of a sequential circuit depend on not only the inputs, but also the state, or the current contents of some memory.
- This makes things more difficult to understand since the same inputs can yield *different* outputs, depending on what's stored in memory.
- The memory contents can also change as the circuit runs, so the *order* in which things occur makes a difference.

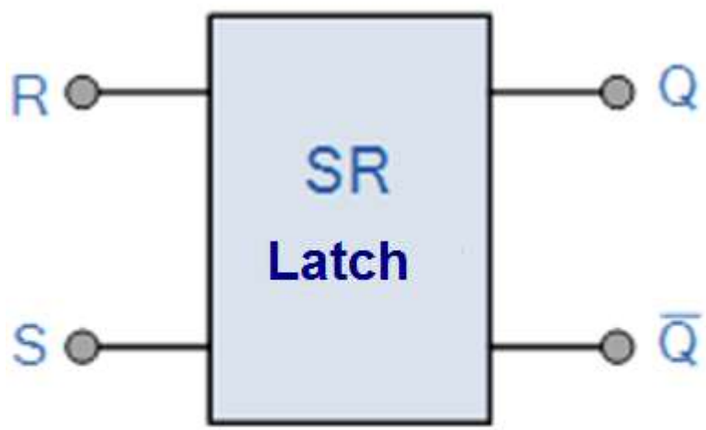
SR Latch

The **SR Latch**, can be considered as one of the most basic sequential logic circuit possible.
This simple circuit is basically a one-bit memory bistable device.

It has two inputs,

- S: will “SET” the device (meaning the output Q = “1”)
- R: will “RESET” the device (meaning the output Q = “0”)

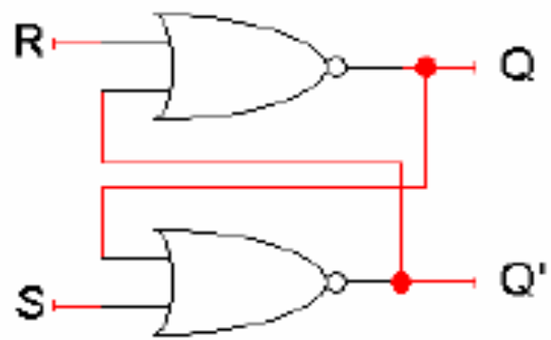
Output Q′ is always **complement of Q**



Symbol

A really confusing circuit

- Let's use NOR gates instead of inverters. The SR latch here has two inputs S and R, which will let us control the outputs Q and Q'.



- Q and Q' feed back into the circuit, so they're not only outputs, they're also inputs!
- To figure out how Q and Q' change, we must look at not only the inputs S and R, but also the *current* values of Q and Q'.

$$Q_{next} = (R + Q'_{current})'$$
$$Q'_{next} = (S + Q_{current})'$$

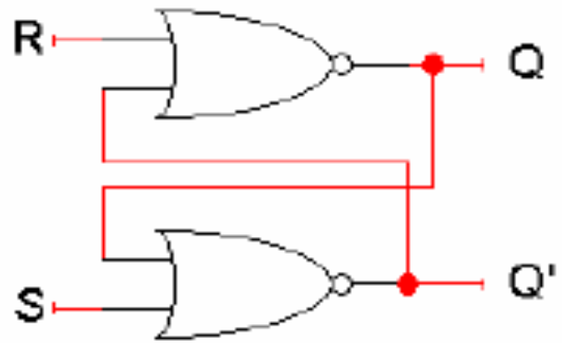
- Let's see how different input values for S and R affect this thing.

Storing a value: SR = 00

- What if S = 0 and R = 0?
- The equations on the right reduce to:

$$Q_{next} = (0 + Q'_{current})' = Q_{current}$$
$$Q'_{next} = (0 + Q_{current})' = Q'_{current}$$

- So when SR = 00, then $Q_{next} = Q_{current}$.
- This is exactly what we need to store values in the latch.



$$Q_{next} = (R + Q'_{current})'$$
$$Q'_{next} = (S + Q_{current})'$$

Setting the latch: SR = 10

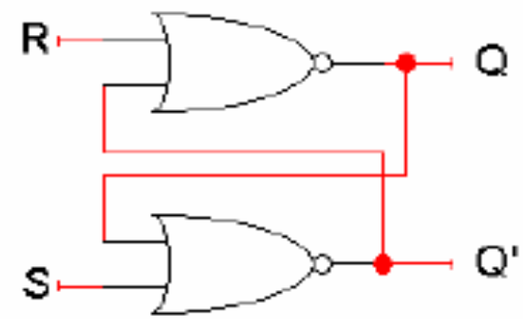
- What if S = 1 and R = 0?
- Since S = 1, Q'_{next} is 0, *regardless* of $Q_{current}$.

$$Q'_{next} = (1 + Q_{current})' = 0$$

- Then this new value of Q' goes into the top NOR gate, along with R = 0.

$$Q_{next} = (0 + 0)' = 1$$

- So when SR = 10, then $Q'_{next} = 0$ and $Q_{next} = 1$. This is how you set the latch to 1; the S input stands for "set."
- Notice it can take up to two steps (two gate delays) from the time S becomes 1 to the time Q_{next} becomes 1.
- But once Q_{next} becomes 1, the outputs will stop changing. This is a stable state.



$$Q_{next} = (R + Q'_{current})'$$
$$Q'_{next} = (S + Q_{current})'$$

SR latches are memories!

- This characteristic table shows that our latch provides everything we need in a memory: we can set it, reset it, or keep the current value.

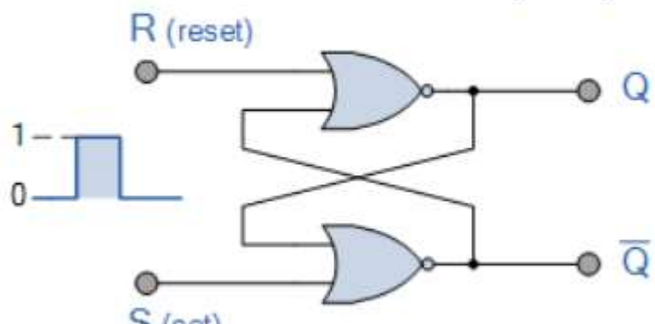
S	R	Q
0	0	No change
0	1	0 (reset)
1	0	1 (set)

- The output Q represents the data stored in the latch. It is also called the state of the latch.

- We can expand the table above into a state table, which explicitly shows that the *next* values of Q and Q' depend on their *current* values, as well as on the inputs S and R.

Inputs		Current		Next	
S	R	Q	Q'	Q	Q'
0	0	0	1	0	1
0	0	1	0	1	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	0	1	0

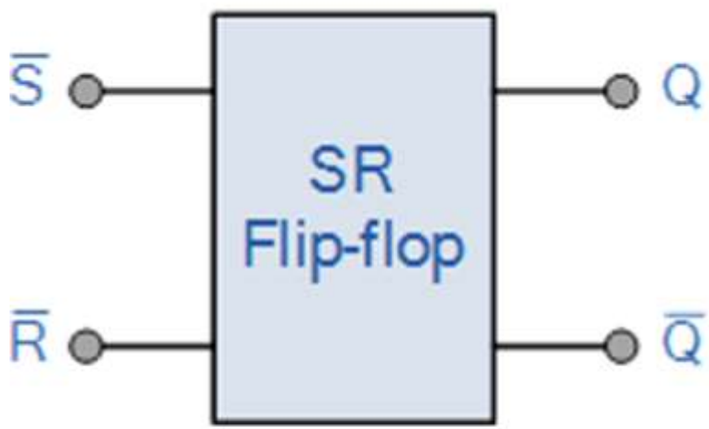
The NOR Gate SR Flip-flop



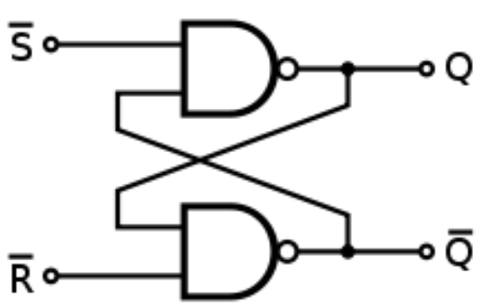
NOR Circuit

S′R′ Latch

The **S′R′ Latch**, is build with two cross-coupled NAND gate.



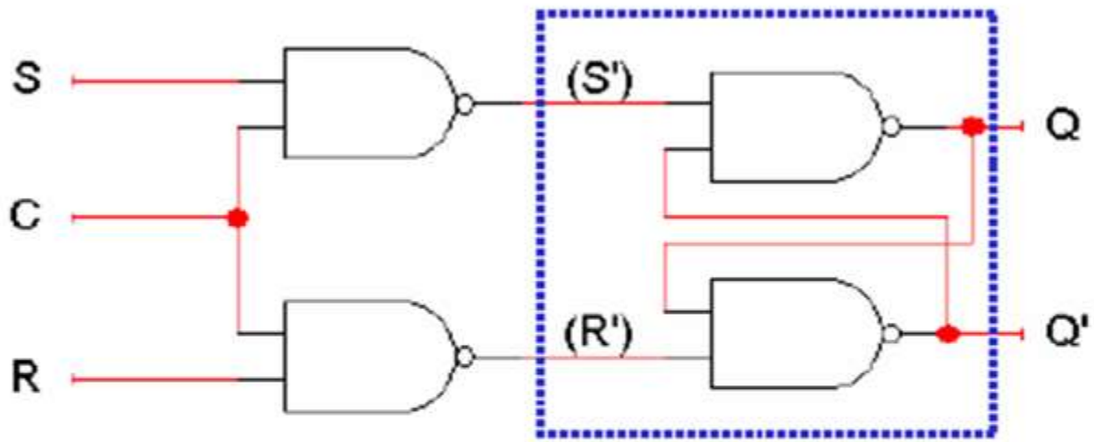
Symbol



S′	R′	Q
0	1	1
1	0	0
1	1	No Change
0	0	!

An SR latch with a control input

- Here is an SR latch with a control input C, which acts like an enable.

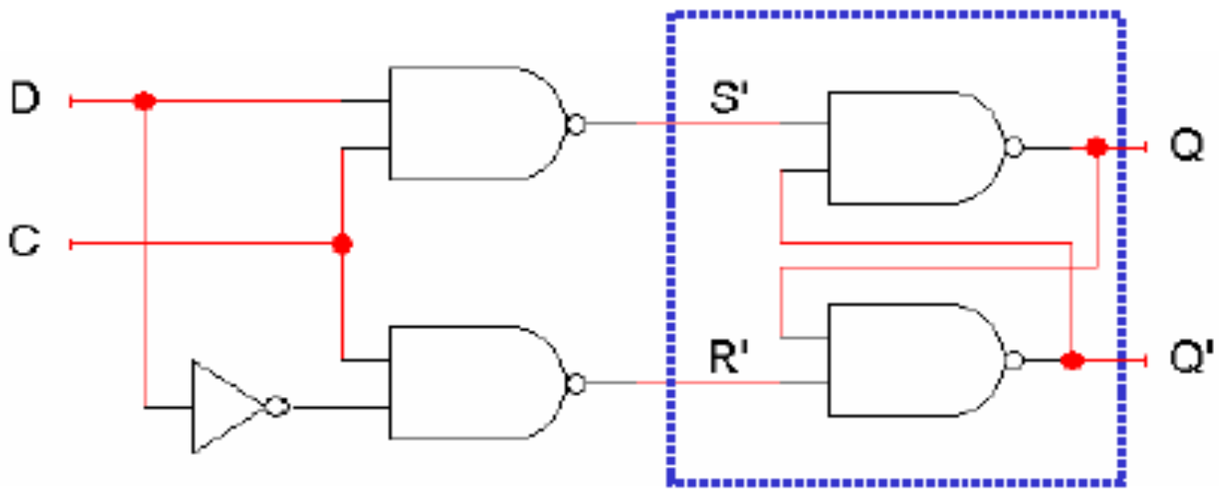


C	S	R	S'	R'	Q
0	x	x	1	1	No change
1	0	0	1	1	No change
1	0	1	1	0	0 (reset)
1	1	0	0	1	1 (set)
1	1	1	0	0	!

- Notice the hierarchical design!
  - The dotted blue box contains the S'R' latch from the previous slide.
  - The additional NAND gates are simply used to generate appropriate inputs for the S'R' latch.
- We'll see more of the control input later today.

D latch

- A D latch is also based on an S'R' latch. The additional gates generate the S' and R' signals, based on inputs D ("data") and C ("control").
  - When C = 0, S' and R' are both 1, so Q does not change.
  - When C = 1, the latch output Q will equal the input D.



C	D	Q
0	x	No change
1	0	0
1	1	1

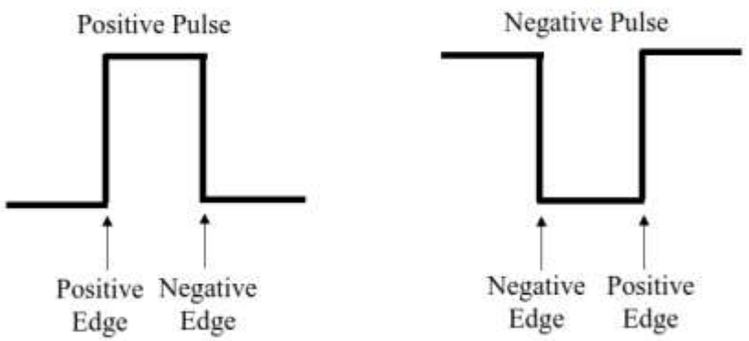
- There are two main advantages of a D latch.
  - No more messing with one input for set and another input for reset!
  - This latch has no "bad" input combinations to avoid. Any of the four possible assignments to C and D are valid.

Clocks and synchronization

- A clock is a special device that continuously outputs 0s and 1s.
  - The time it takes the clock to change from 1 to 0 and back to 1 is called the clock period, or clock cycle time.
  - The clock frequency is the inverse of the clock period. The unit of measurement for frequency is the hertz.

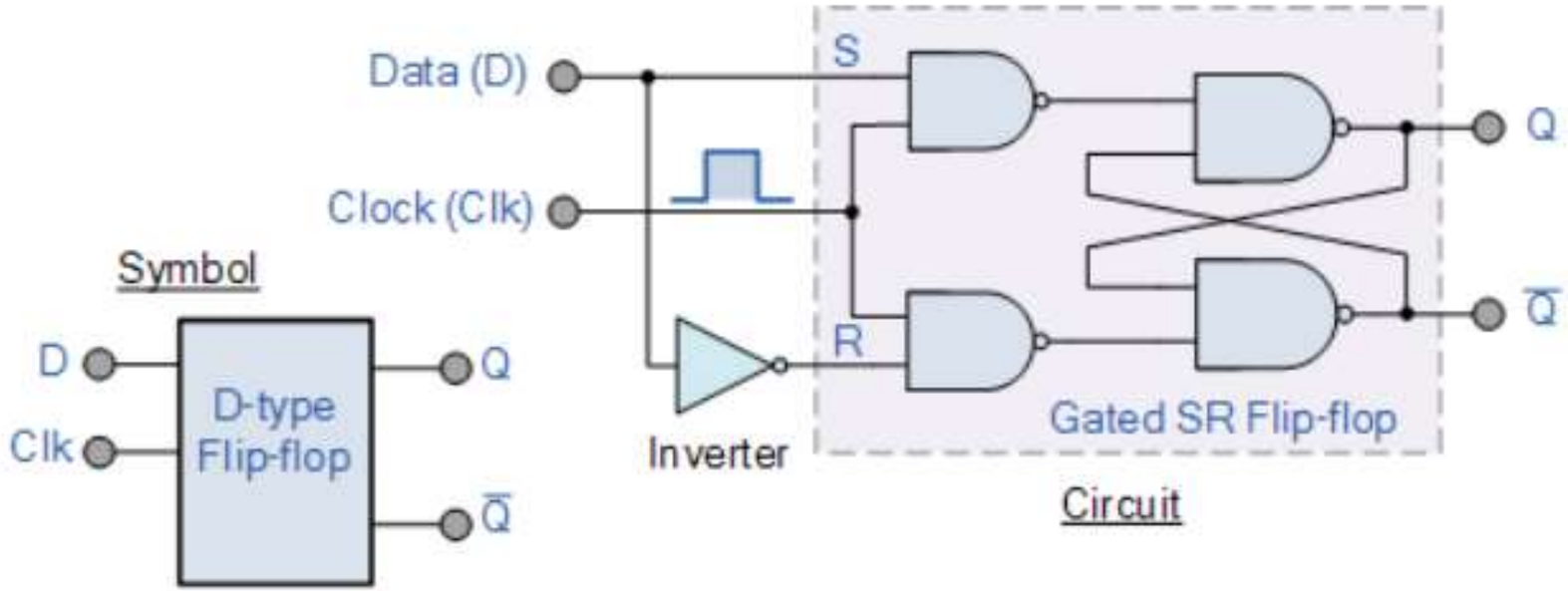
clock period

Clock Pulse Definition



- Clocks are often used to synchronize circuits.
  - They generate a repeating, predictable pattern of 0s and 1s that can trigger certain events in a circuit, such as writing to a latch.
  - If several circuits share a common clock signal, they can coordinate their actions with respect to one another.
- This is similar to how humans use real clocks for synchronization.

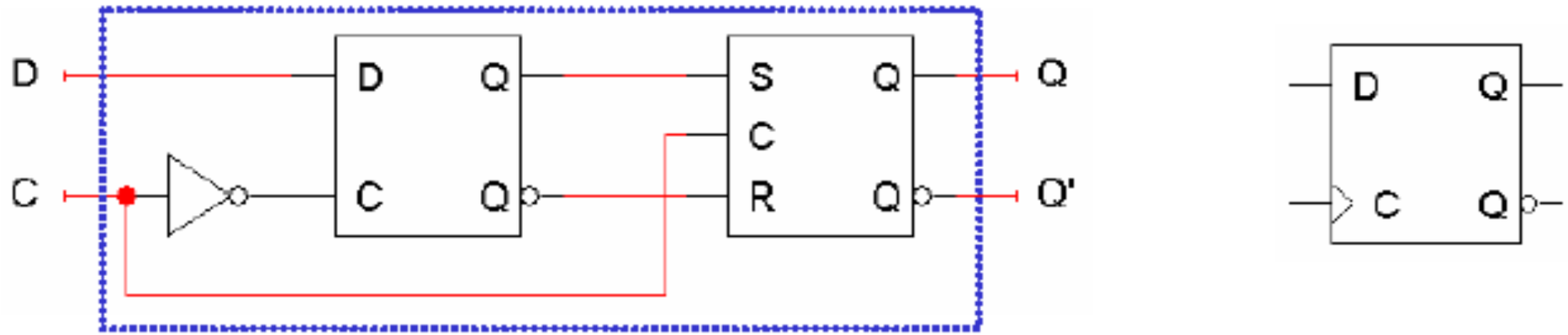
D-type Flip-Flop Circuit



The output of the flip flop would always change on every pulse applied to this data input.

To avoid this an additional input called the “CLOCK” or “ENABLE” input is used to isolate the data input from the flip flop’s latching circuitry after the desired data has been stored. The effect is that D input condition is only copied to the output Q when the clock input is active. This then forms the basis of another sequential device called a **D Flip Flop**.

A positive edge-triggered D flip-flop

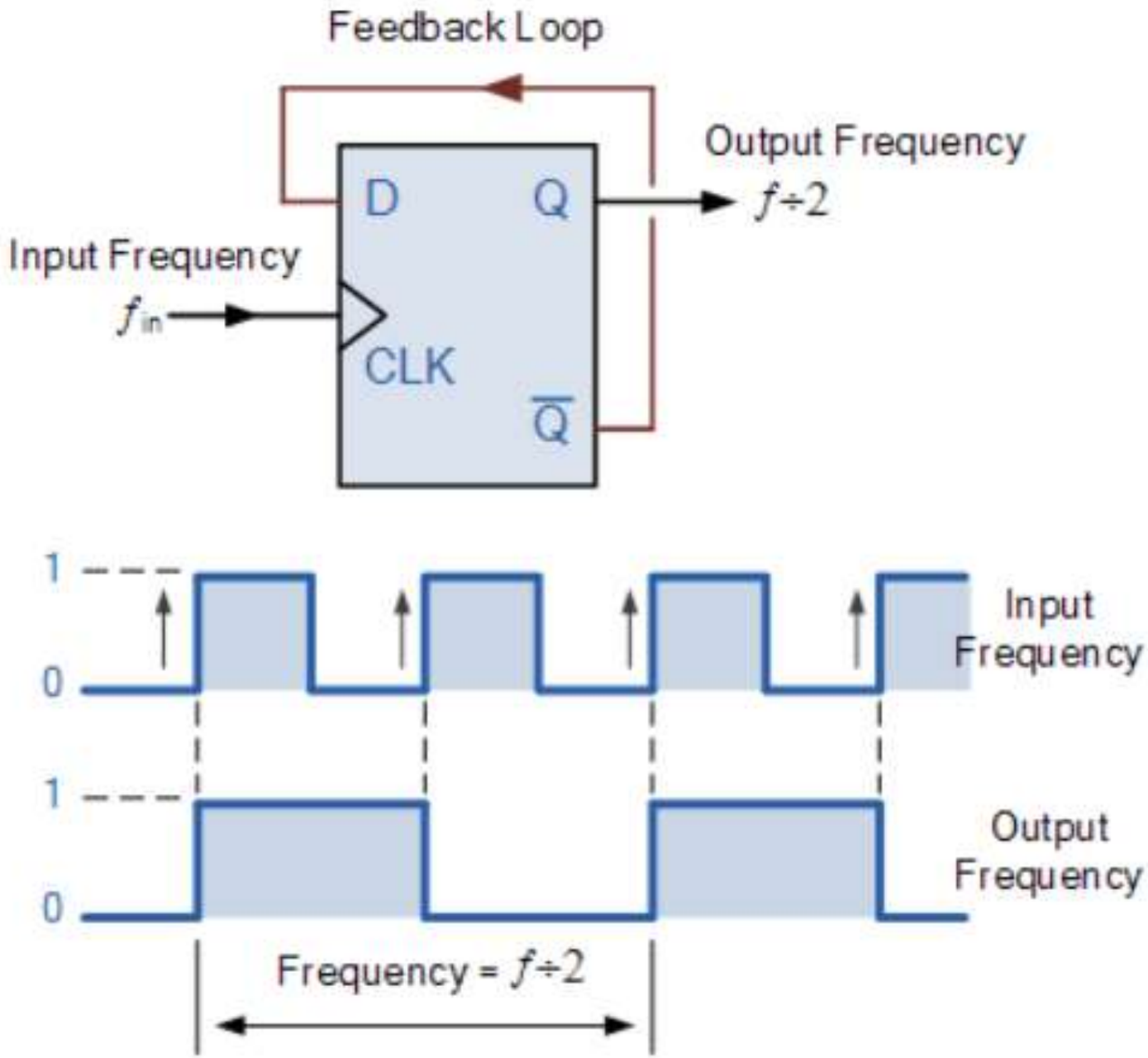


- This positive edge-triggered D flip-flop includes two latches.
  - The flip-flop output Q changes only *after* the positive edge of C.
  - The change is based on the flip-flop input value D that was present at the positive edge of the clock signal.
- A D flip-flop behaves like a D latch except for its positive edge-triggered nature, which is not explicit in the table below.

C	D	Q
0	x	No change
1	0	0 (reset)
1	1	1 (set)

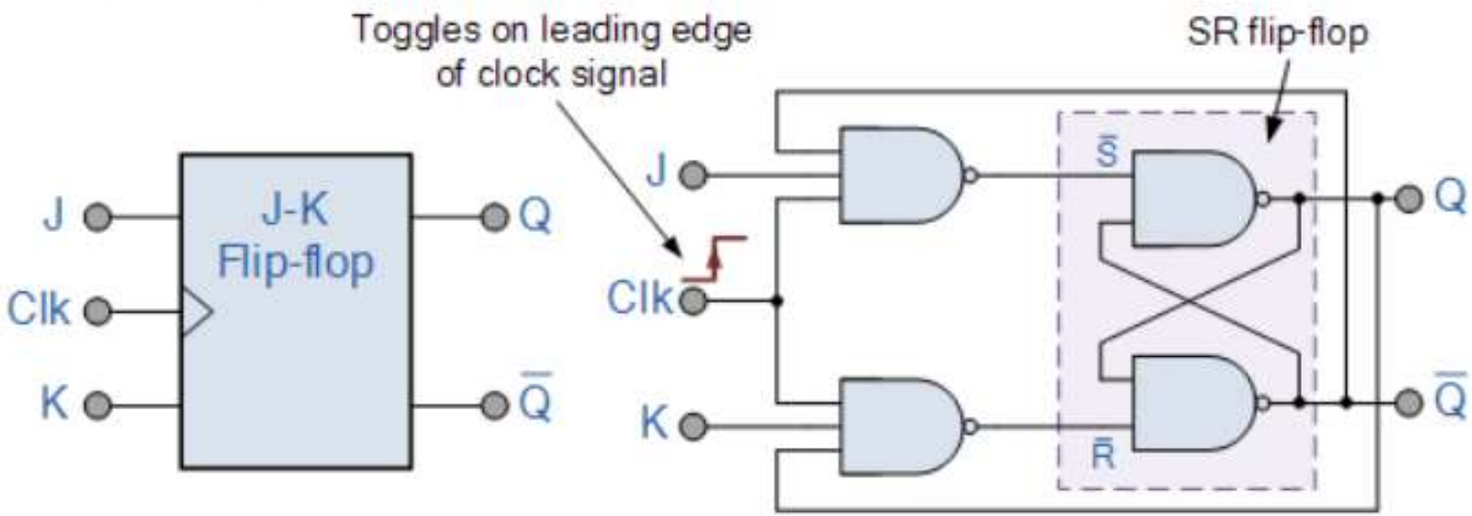
It can be seen from the frequency waveforms above, that by "feeding back" the output from Q to the input terminal D, the output pulses at Q have a frequency that are exactly one half (ƒ/2) that of the input clock frequency, (f_{IN}). In other words the circuit produces **frequency division** as it now divides the input frequency by a factor of two (an octave) as Q = 1 once every two clock cycles.

Divide-by-2 Counter



The JK Flip Flop

The Basic JK Flip-flop



Symbol

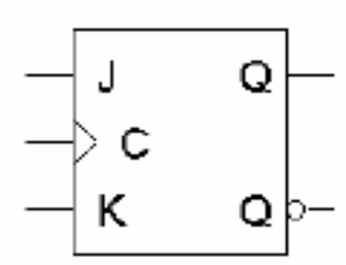
Both the S and the R inputs of the previous SR bistable have now been replaced by two inputs called the J and K inputs, respectively after its inventor Jack Kilby. Then this equates to: J = S and K = R.

The Truth Table for the JK Function

	Input		Output		Description
	J	K	Q	Q_{next}	
same as for the SR Latch	0	0	0	0	Memory no change
	0	0	1	1	
	0	1	1	0	Reset Q » 0
	0	1	0	0	
	1	0	0	1	Set Q » 1
	1	0	1	1	
toggle action	1	1	0	1	Toggle
	1	1	1	0	

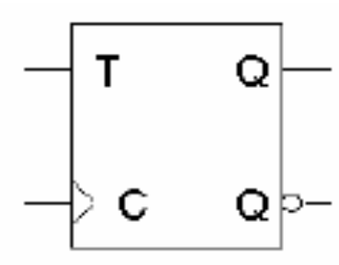
Flip-flop variations

- We can make different versions of flip-flops based on the D flip-flop, just like we made different latches based on the S'R' latch.
- The JK flip-flop has inputs that act like S and R, but JK = 11 *complements* the flip-flop's current state.



C	J	K	Q_{next}
0	x	x	No change
1	0	0	No change
1	0	1	0 (reset)
1	1	0	1 (set)
1	1	1	$Q'_{current}$

- A T flip-flop can only maintain or complement its current state.



C	T	Q_{next}
0	x	No change
1	0	No change
1	1	$Q'_{current}$

Characteristic tables

- The tables that we've made so far are called characteristic tables.
  - They show the next state Q(t+1) in terms of the current state Q(t) and the inputs.
  - For simplicity, the control input C is usually not listed.
  - Again, these tables don't indicate the positive edge-triggered nature of the flip-flops.

D	Q(t+1)	Operation
0	0	Reset
1	1	Set

J	K	Q(t+1)	Operation
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q'(t)	Complement

T	Q(t+1)	Operation
0	Q(t)	No change
1	Q'(t)	Complement

Characteristic equations

- We can also write characteristic equations, where the next state Q(t+1) is defined in terms of the current state Q(t) and the flip-flop inputs.

D	Q(t+1)	Operation
0	0	Reset
1	1	Set

$$Q(t+1) = D$$

J	K	Q(t+1)	Operation
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q'(t)	Complement

$$Q(t+1) = K'Q(t) + JQ'(t)$$

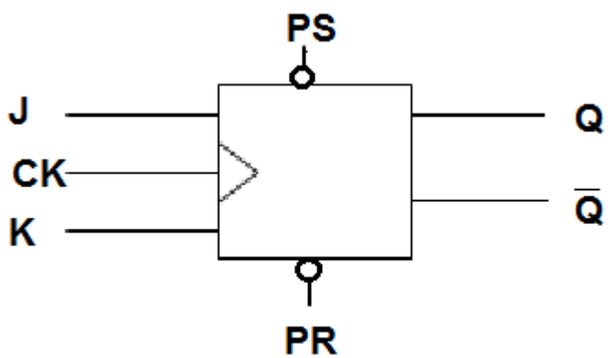
T	Q(t+1)	Operation
0	Q(t)	No change
1	Q'(t)	Complement

$$Q(t+1) = T'Q(t) + TQ'(t)$$
$$= T \oplus Q(t)$$

Direct Inputs

Flip-flops often provides special inputs **PS, PR** for setting and resetting them:

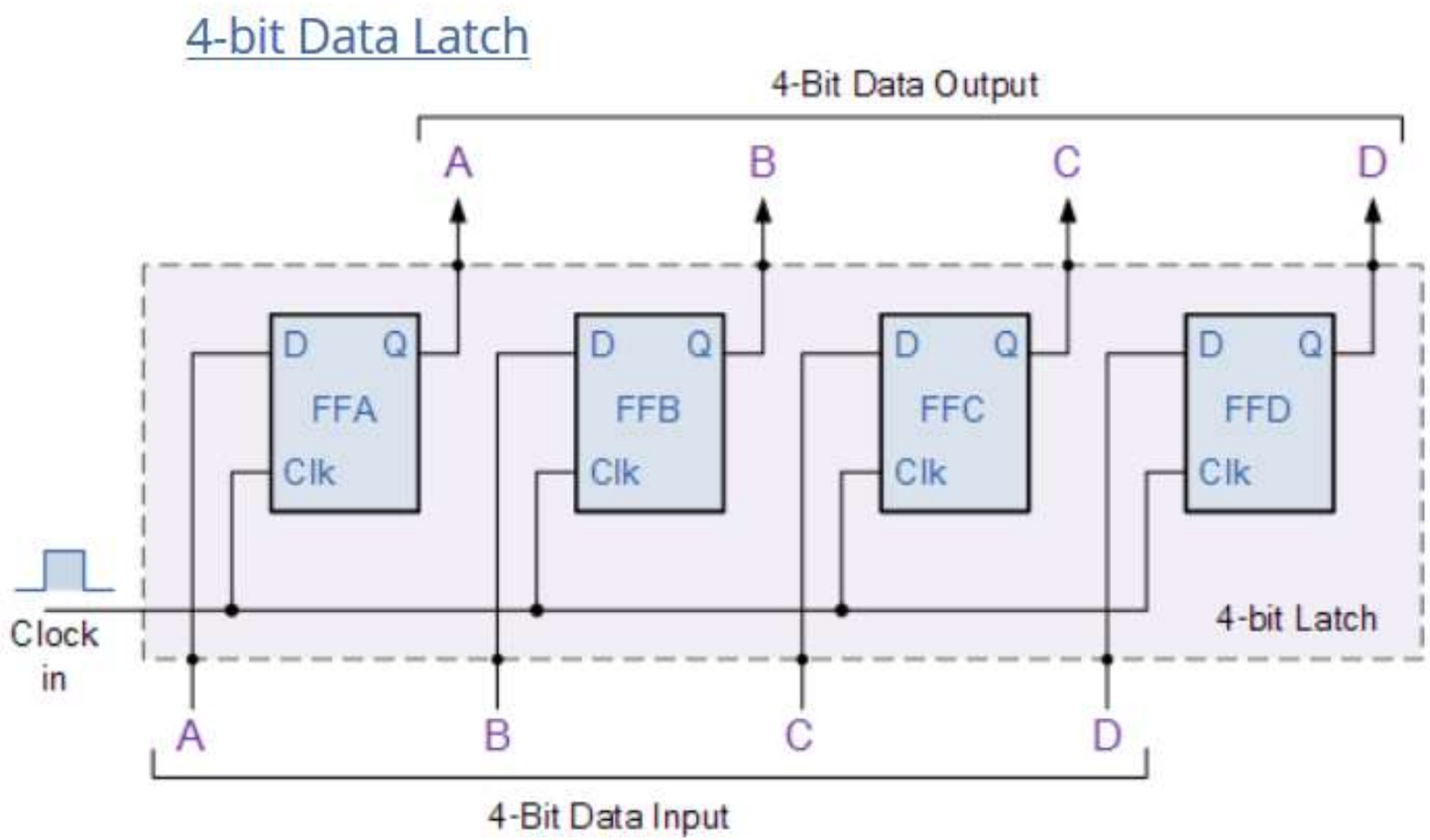
- Independent of clock input
- Initializes the flip-flop



PS′	PR′	CK	J	K	Q(t+1)
1	1	↑	0	0	Q(t)
1	1	↑	0	1	0
1	1	↑	1	0	1
1	1	↑	1	1	Q′(t)
0	1	X	X	X	1
1	0	X	X	X	0
0	0	X	X	X	Undefined

D Flip Flops as Data Latches

A data latch can be used as a device to hold or remember the data present on its data input, thereby acting a bit like a single bit memory device and IC's such as the TTL 74LS74 or the CMOS 4042 are available in Quad format exactly for this purpose. By connecting together four, *1-bit* data latches so that all their clock inputs are connected together and are "clocked" at the same time, a simple "4-bit" Data latch can be made as shown below.



College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 8
Registers and RTL

Unit 8: Registers and RTL

Overview

- Registers
  - Parallel in Parallel out Registers
  - Shift Registers
- Bus Construction
- Register Transfer Language
  - Micro operations

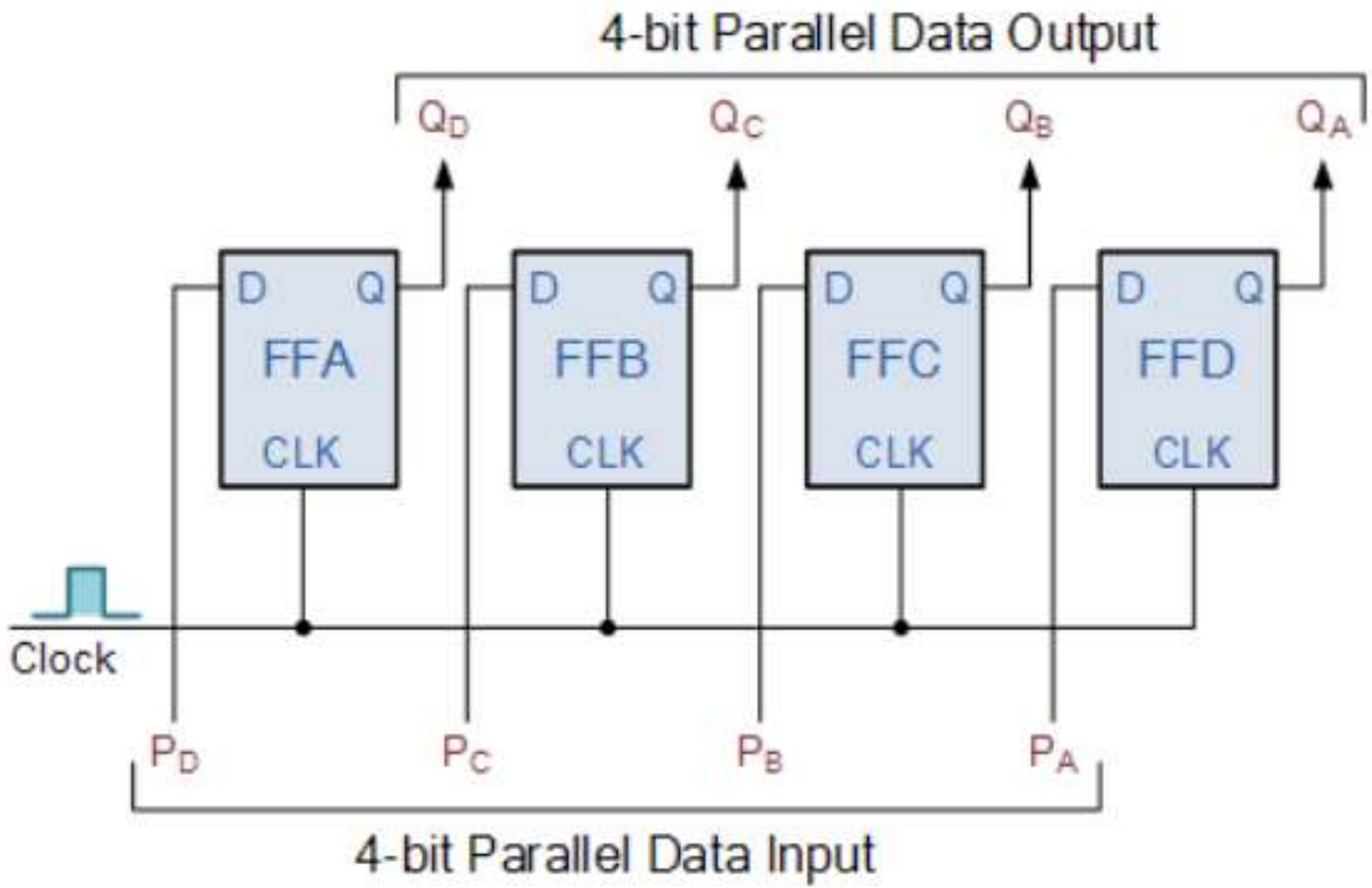
Chapter-6

M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

Registers

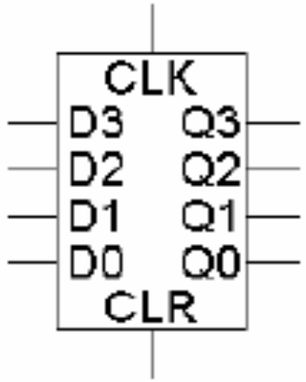
- Flip-flops are limited because they can store only one bit.
  - We had to use two flip-flops for most of our examples so far.
  - Most computers work with integers and single-precision floating-point numbers that are 32-bits long.
- A register is an extension of a flip-flop that can store multiple bits.
- Registers are commonly used as temporary storage in a processor.
  - They are faster and more convenient than main memory.
  - More registers can help speed up complex calculations.
- Later we'll learn more about how registers are used in processors, and some of the differences between registers and random-access memories or RAM.

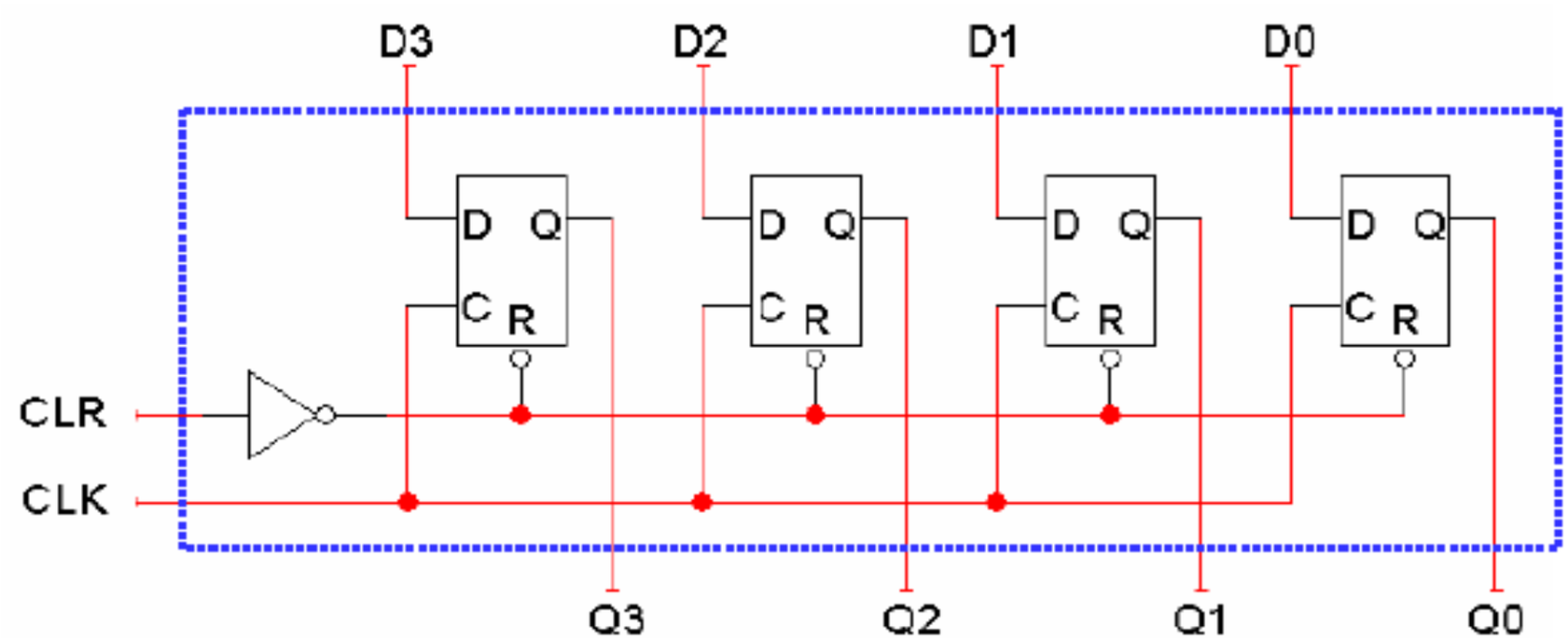
4-bit Parallel-in to Parallel-out Shift Register



A basic register

- Basic registers are easy to build. We can store multiple bits just by putting a bunch of flip-flops together!
- A 4-bit register from LogicWorks, Reg-4, is on the right, and its internal implementation is below.
  - This register uses D flip-flops, so it's easy to store data without worrying about flip-flop input equations.
  - All the flip-flops share a common CLK and CLR signal.

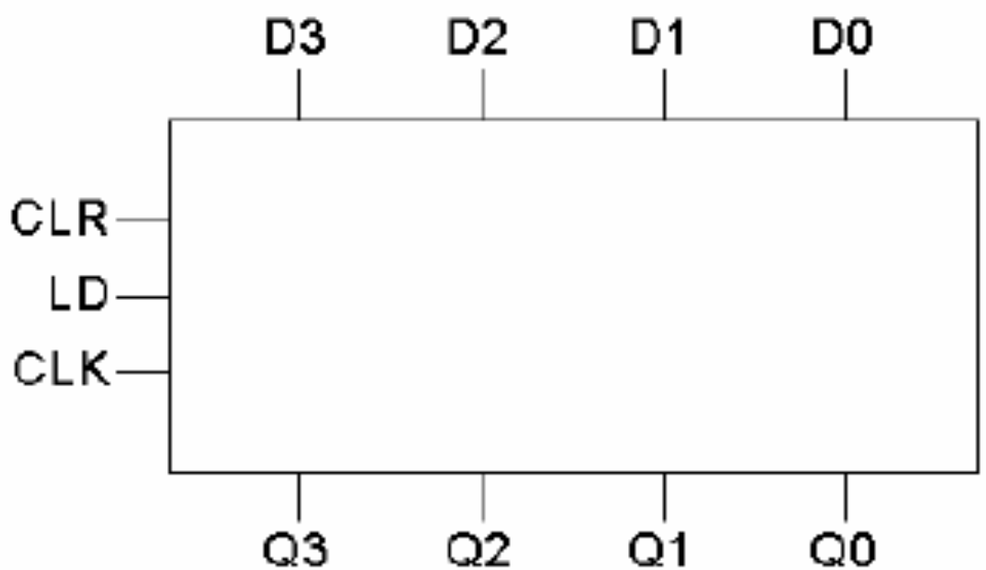




Adding another operation

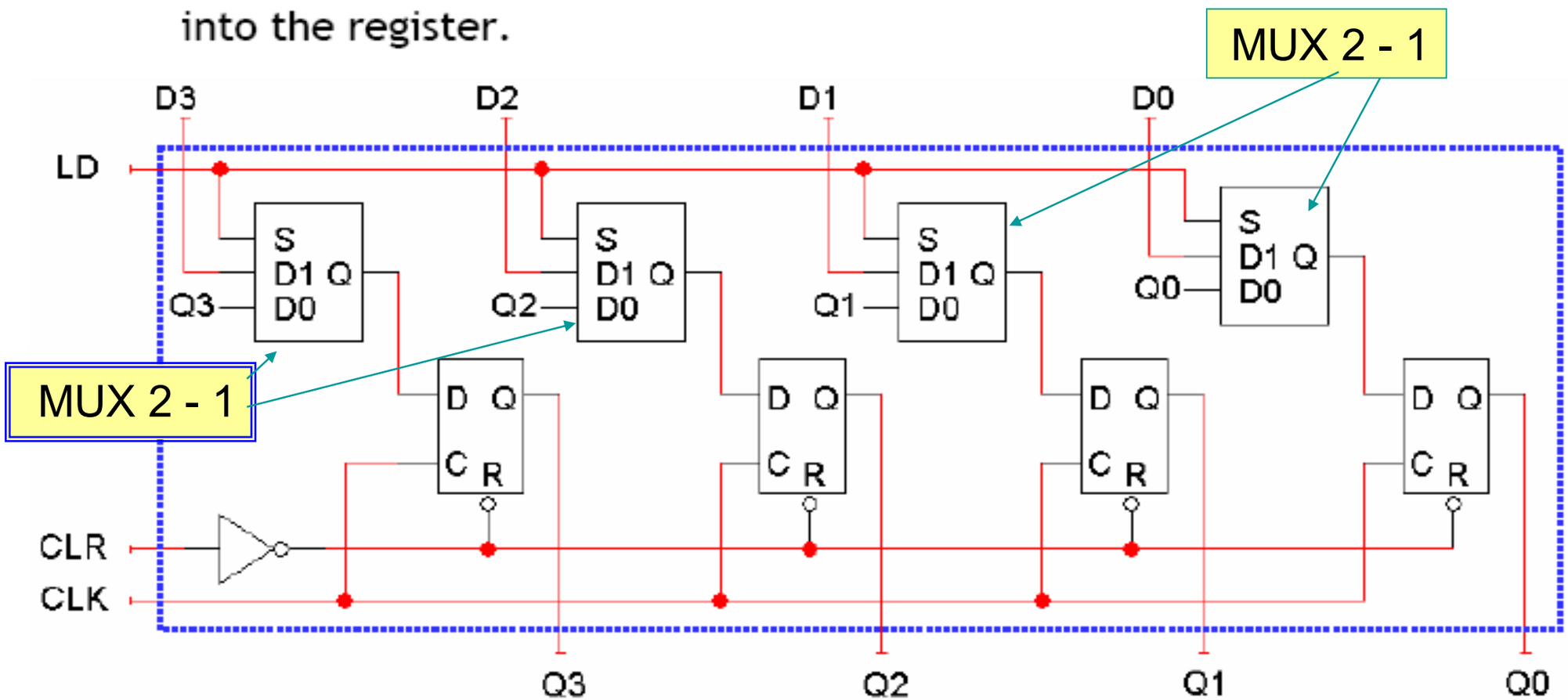
- The input D3-D0 is copied to the output Q3-Q0 on *every* clock cycle.
- How can we store the current value for more than one cycle?
- Let's try to add a load input signal LD to the register.
  - If LD = 0, the register keeps its current contents.
  - If LD = 1, the register stores a new value, taken from inputs D3-D0.

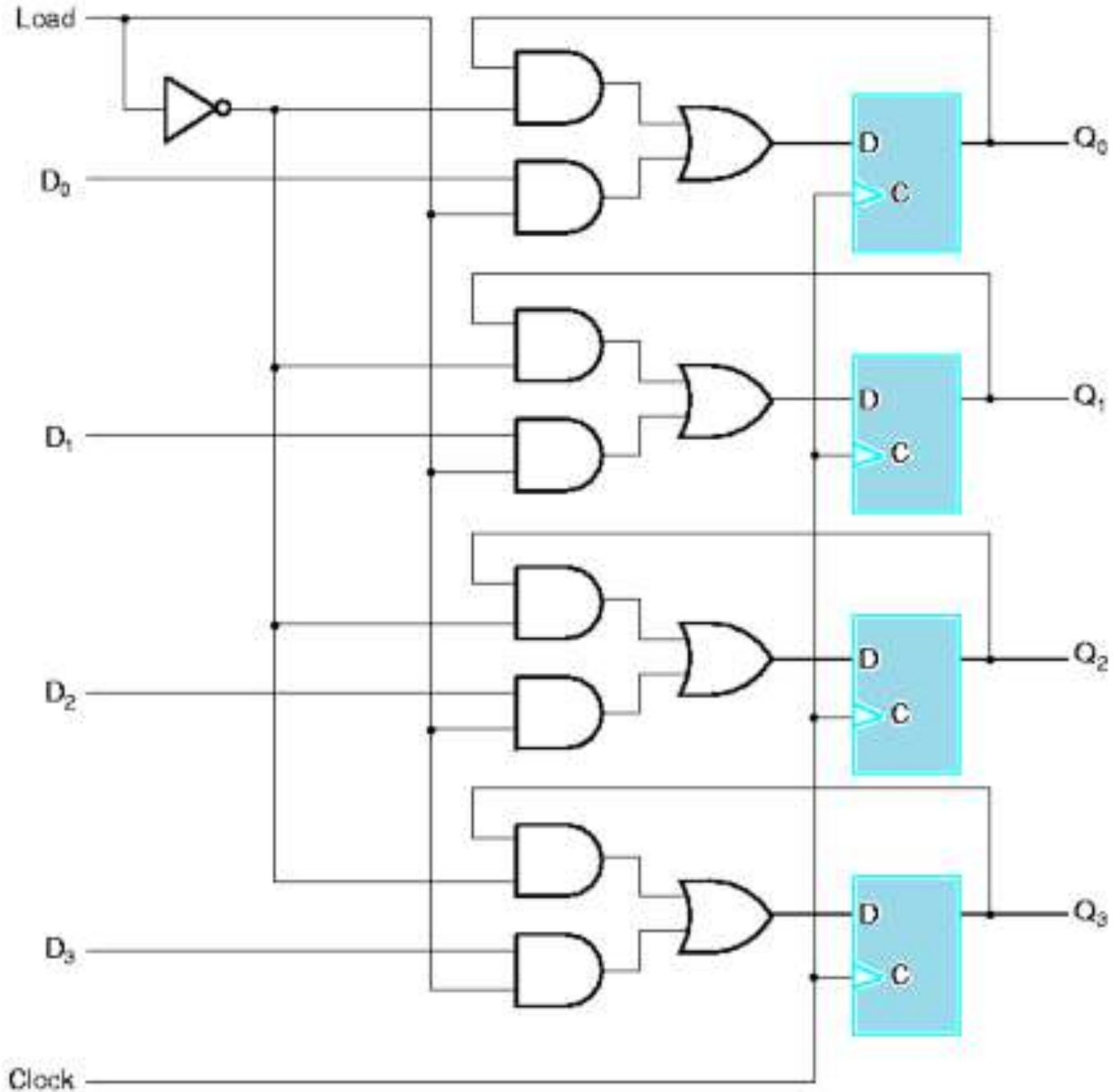
LD	Q(t+1)
0	Q(t)
1	D_3-D_0



A better parallel load

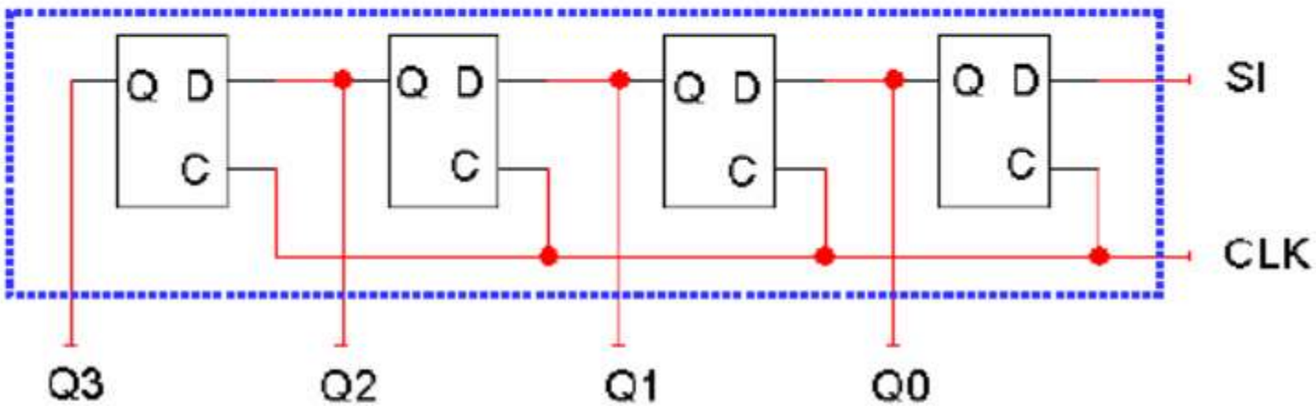
- Another idea is to modify the flip-flop D inputs and not the clock signal.
  - When LD = 0 the flip-flop inputs are Q3-Q0, so each flip-flop keeps its current value.
  - When LD = 1 the flip-flop inputs are D3-D0, so this new value is loaded into the register.




Load
D_0
D_1
D_2
D_3
Clock
D
C
Q_0
Q_1
Q_2
Q_3

Shift registers (Left Shift)

- A shift register "shifts" its output once every clock cycle. SI is an input that supplies a new bit to shift "into" the register.



Q0(t+1) = SI
Q1(t+1) = Q0(t)
Q2(t+1) = Q1(t)
Q3(t+1) = Q2(t)

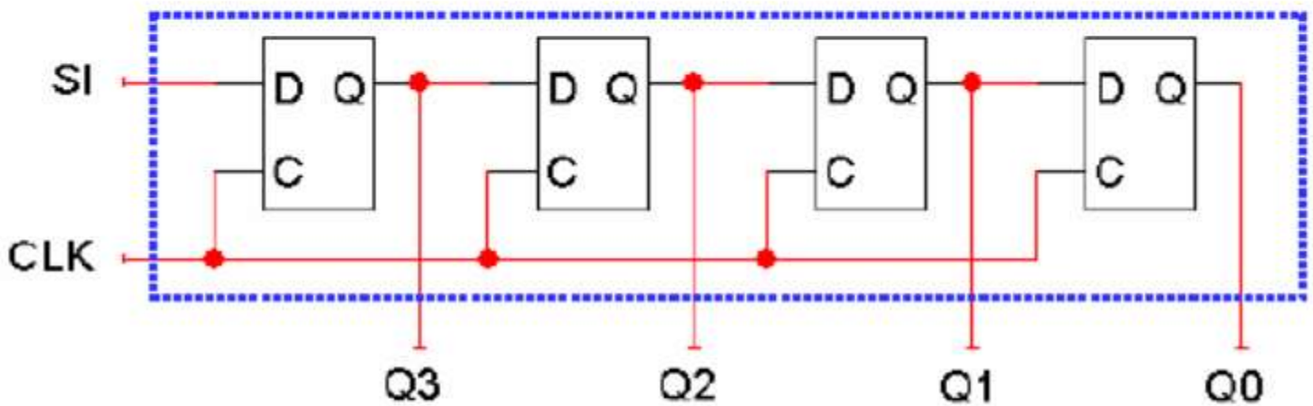
- Here is one example transition.

Present State Q3-Q0	Input SI	Next State Q3-Q0
0110	1	1101

- The current Q3 (0 in this example) will be lost on the next cycle.

Shift registers (Right Shift)

- A shift register "shifts" its output once every clock cycle. SI is an input that supplies a new bit to shift "into" the register.



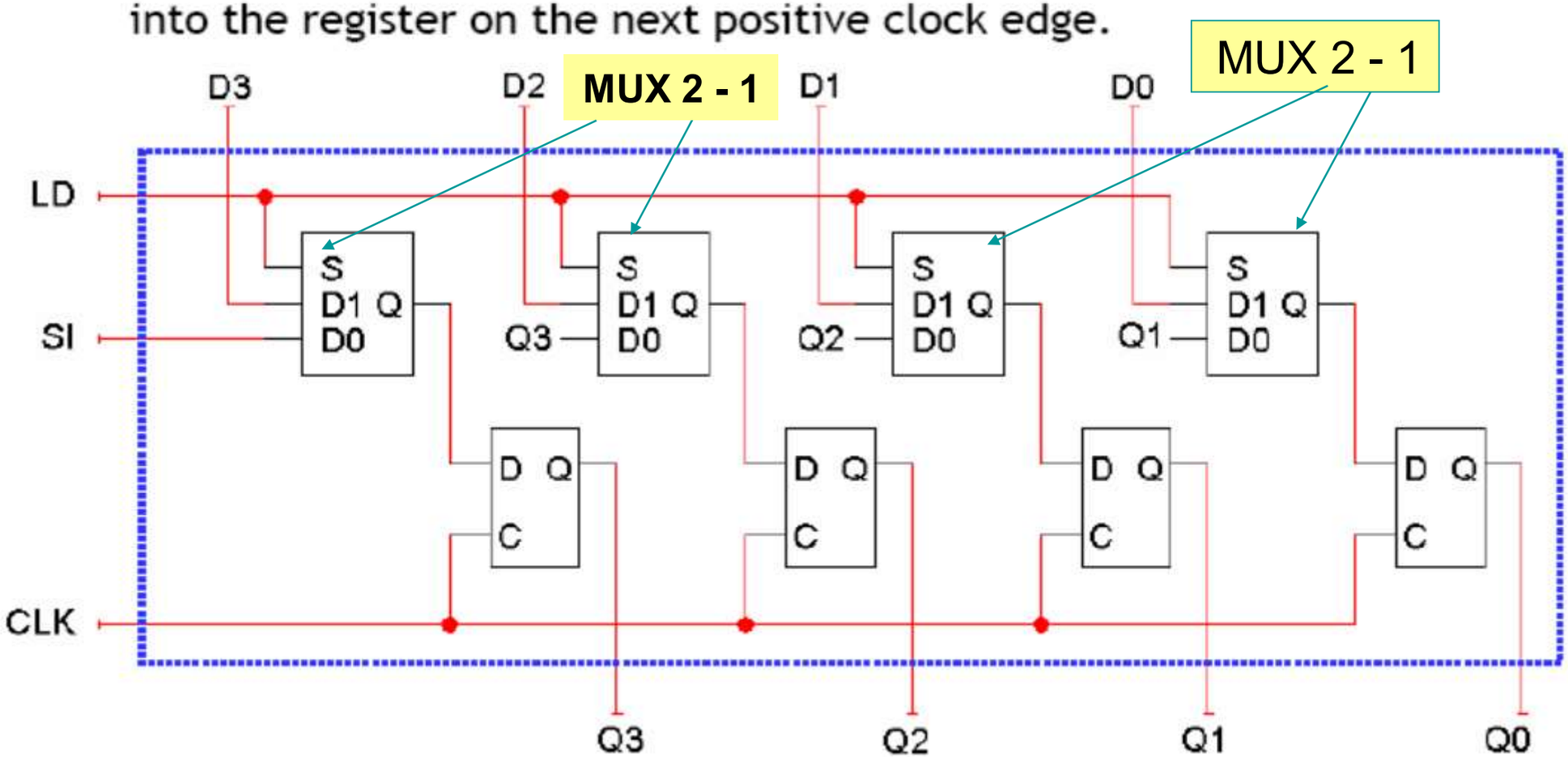
- Here is one example transition.

Present State Q3-Q0	Input SI	Next State Q3-Q0
0110	1	1011

- The current Q0 (0 in this example) will be lost on the next cycle.

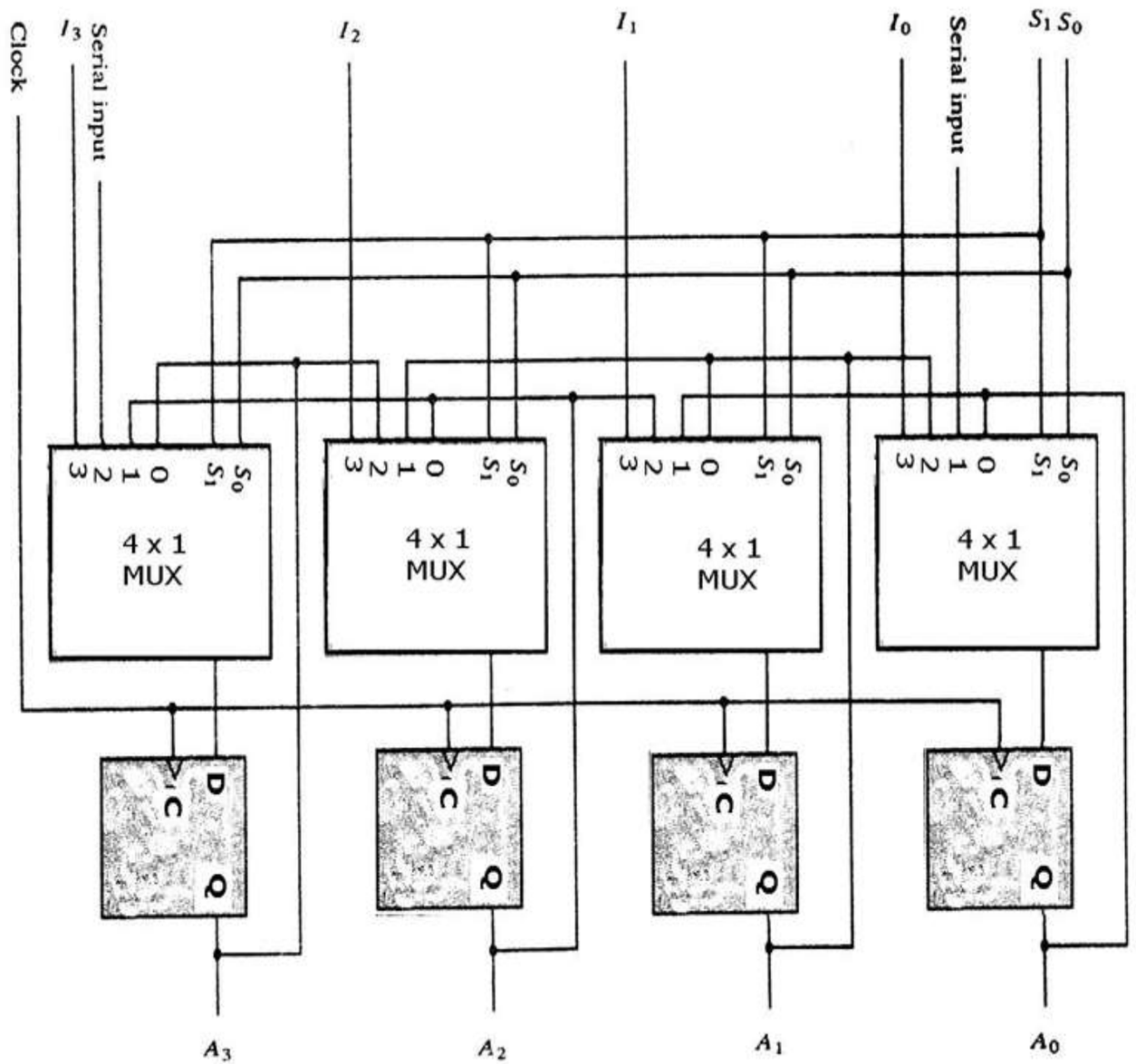
Shift registers with parallel load

- We can add a parallel load operation, just as we did for regular registers.
  - Serial — When LD = 0 the flip-flop inputs will be SIQ3Q2Q1, so the register will shift on the next positive clock edge.
  - Parallel — When LD = 1, the flip-flop inputs are D0-D3, and a new value is loaded into the register on the next positive clock edge.



A Bidirectional Shift Register with parallel load

S1	S0	Operation
0	0	**No change**
0	1	**Shift left**
1	0	**Shift right**
1	1	**Parallel load**



Other types of shift registers

- Logical shifts – Standard shifts like we just saw. In the absence of a SI input, 0 occupies the vacant position.
  - Left: 0110 -> 1100
  - Right: 0110 -> 0011
- Circular shifts (also called ring counters or rotates) – The shifted out bit wraps around to the vacant position.
  - Left: 1001 -> 0011
  - Right: 1001 -> 1100
- Switch-tail ring counter (aka Johnson counter) – Similar to the ring counter, but the serial input is the complement of the serial output.
  - Left: 1001 -> 0010
  - Right: 1001 -> 0100
- Arithmetical shifts – Left shifting is the same as a logical shift. Right shifting however maintains the MSB.
  - Left: 0110 -> 1100
  - Right: 0110 -> 0011; 1011 -> 1101

Example

- Consider a 4-bit register with the following inputs
  - parallel data inputs ABCD = 0011
  - left serial input LSI = 1 (This is the serial input for a left shift.)
  - right serial input RSI = 0 (This is the serial input for a right shift.)
  - The three control inputs S_2, S_1, S_0 operate as shown in the table to the right.
  - Let the initial register content be $Q_A Q_B Q_C Q_D$ = 0101

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Example

S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0				
0	0	1				
0	1	0				
1	0	0				
0	1	1				
1	0	1				
1	1	1				
0	0	0				
1	1	0				

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Example

S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0				
0	0	1				
0	1	0				
1	0	0				
0	1	1				
1	0	1				
1	1	1				
0	0	0				
1	1	0				

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Example

S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0	1	0	1	1
0	0	1				
0	1	0				
1	0	0				
0	1	1				
1	0	1				
1	1	1				
0	0	0				
1	1	0				

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Example

S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0	1	0	1	1
0	0	1	0	1	1	1
0	1	0				
1	0	0				
0	1	1				
1	0	1				
1	1	1				
0	0	0				
1	1	0				

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Example

S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0	1	0	1	1
0	0	1	0	1	1	1
0	1	0	0	0	1	1
1	0	0				
0	1	1				
1	0	1				
1	1	1				
0	0	0				
1	1	0				

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Example

S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0	1	0	1	1
0	0	1	0	1	1	1
0	1	0	0	0	1	1
1	0	0	0	0	1	1
0	1	1				
1	0	1				
1	1	1				
0	0	0				
1	1	0				

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Example

S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0	1	0	1	1
0	0	1	0	1	1	1
0	1	0	0	0	1	1
1	0	0	0	0	1	1
0	1	1	1	0	0	1
1	0	1				
1	1	1				
0	0	0				
1	1	0				

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Example

S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0	1	0	1	1
0	0	1	0	1	1	1
0	1	0	0	0	1	1
1	0	0	0	0	1	1
0	1	1	1	0	0	1
1	0	1	0	0	1	1
1	1	1				
0	0	0				
1	1	0				

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Example

S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0	1	0	1	1
0	0	1	0	1	1	1
0	1	0	0	0	1	1
1	0	0	0	0	1	1
0	1	1	1	0	0	1
1	0	1	0	0	1	1
1	1	1	1	1	1	1
0	0	0				
1	1	0				

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Example

S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0	1	0	1	1
0	0	1	0	1	1	1
0	1	0	0	0	1	1
1	0	0	0	0	1	1
0	1	1	1	0	0	1
1	0	1	0	0	1	1
1	1	1	1	1	1	1
0	0	0	1	1	1	1
1	1	0				

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Example

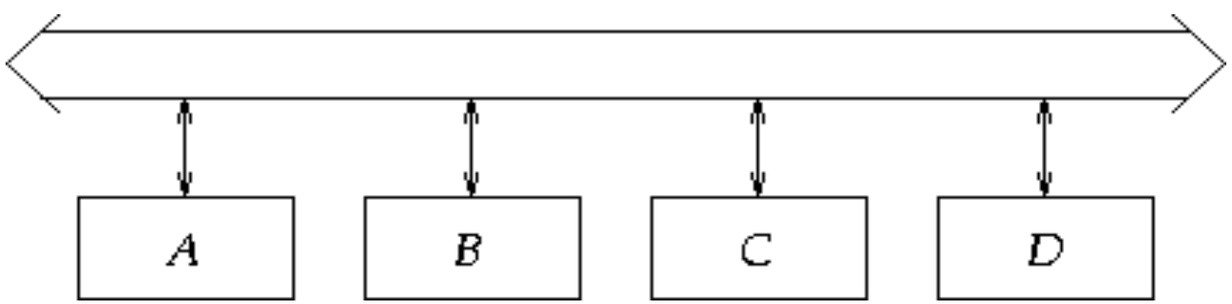
S2	S1	S0	Q_A	Q_B	Q_C	Q_D
			0	1	0	1
0	0	0	1	0	1	1
0	0	1	0	1	1	1
0	1	0	0	0	1	1
1	0	0	0	0	1	1
0	1	1	1	0	0	1
1	0	1	0	0	1	1
1	1	1	1	1	1	1
0	0	0	1	1	1	1
1	1	0	0	0	0	0

S2	S1	S0	Operation
0	0	0	left shift
0	0	1	circular left shift
0	1	0	right shift
0	1	1	circular right shift
1	0	0	no change
1	0	1	parallel load
1	1	0	complement each bit
1	1	1	set to 1111

Other Information	
Parallel Load	ABCD = 0011
LS1	1
RSI	0

Bus Construction

- A bus consists of a set of parallel data lines
- To transfer data using a bus:
  - connect the output of the source register to the bus;
  - connect the input of the target register to the bus;
  - when the clock pulse arrives, the transfer occurs



Bus and Memory transfers

Figure 4-3 Bus system for four registers.

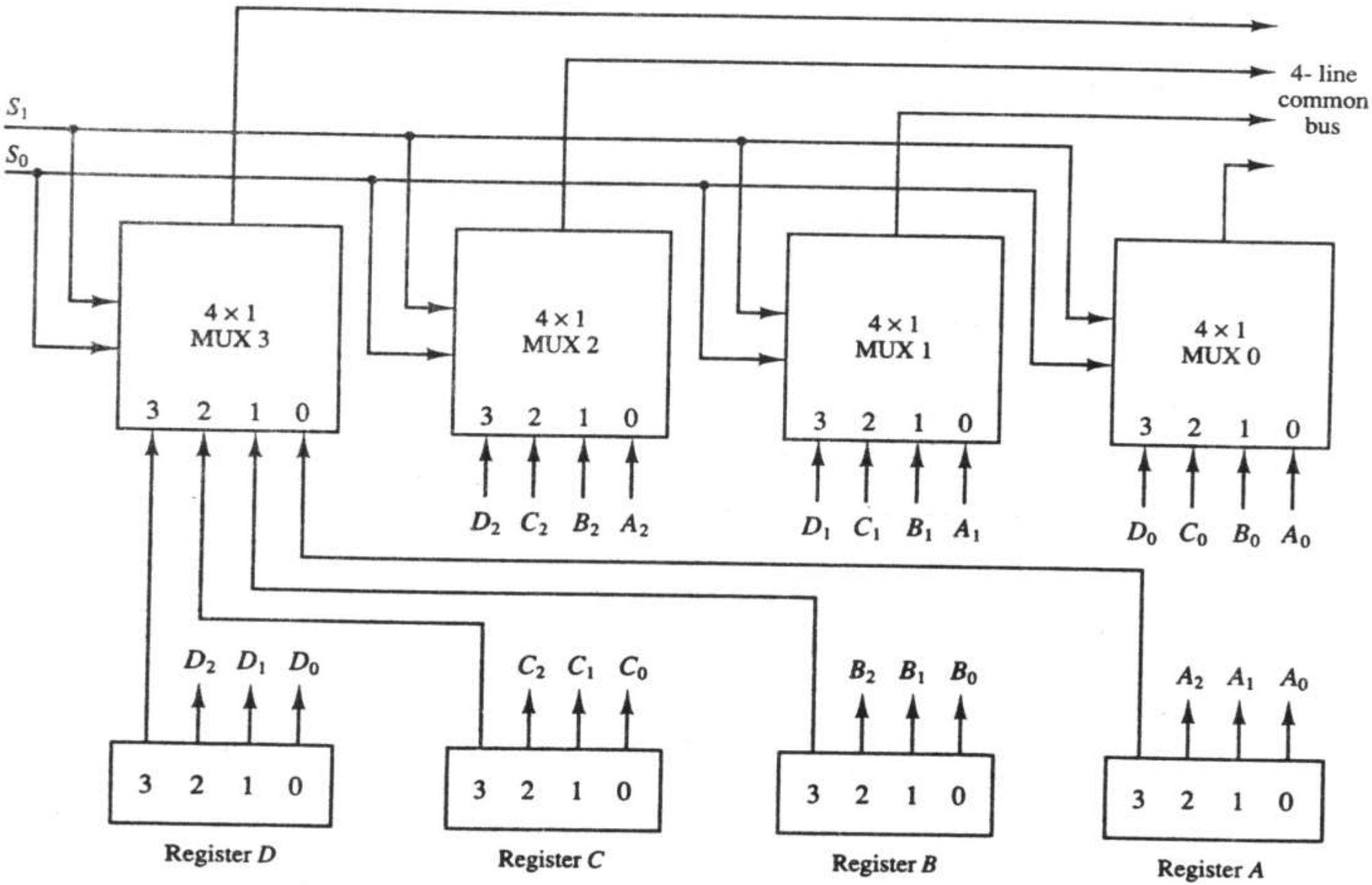


TABLE 4-2 Function Table for Bus of Fig. 4-3

S_1	S_0	Register selected
0	0	*A*
0	1	*B*
1	0	*C*
1	1	*D*

$$A \leftarrow C$$

Step 1: The content of register C is placed on the bus $BUS \leftarrow C,$

Step 2: The content of the bus is loaded into register A by activating its load control input. $A \leftarrow BUS$

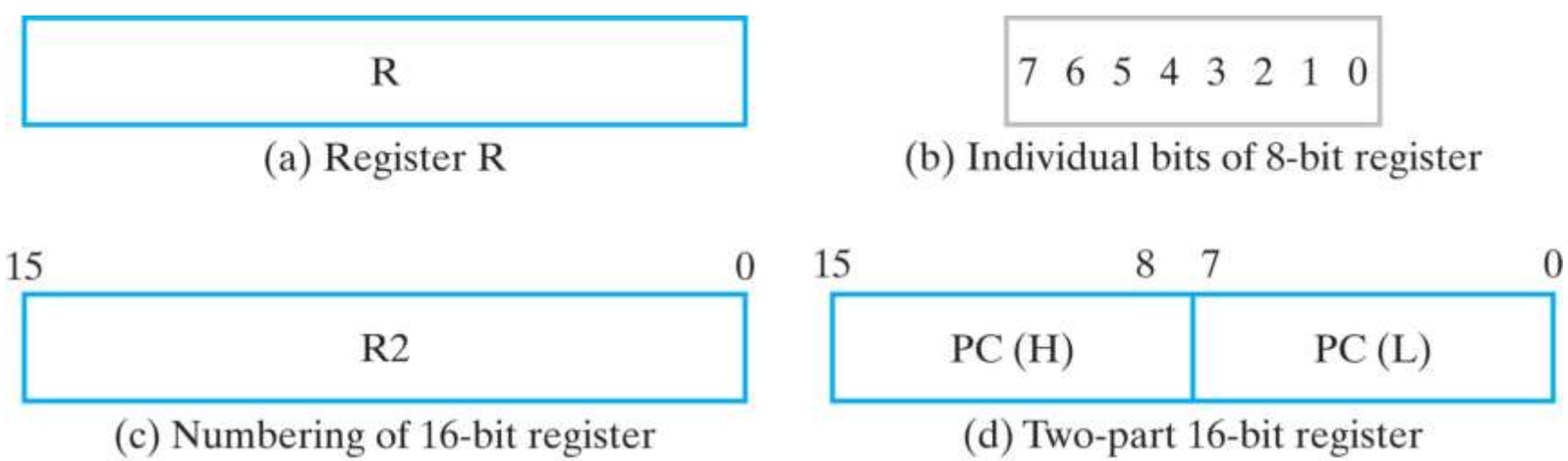
<u>Question</u>

A digital computer has a common bus system for 16 registers of 32 bit each. The bus is constructed with multiplexers.

- How many selection inputs are there is each multiplexer?
- What size of multiplexers is needed?
- How many multiplexers are there in the bus?

Register Transfer Language

- **Micro operations** (micro-ops) are operations on data stored in registers
- **Register transfer language (RTL)** is a concise and precise means of describing those operations
- **RTL expressions** are made up of elements which describe the registers being manipulated, and the micro-ops being performed on them
- Registers are denoted by uppercase letters (sometimes followed by numbers) that indicate the function of the register
  - e.g. R0, R1, AR, PC, MAR, et al.
  - The individual bits can be denoted using parenthesis and bit numbers or labels
    - e.g. R0(0), R0(7:0), PC(L), PC(H)



(a) Register R (b) Individual bits of 8-bit register

(c) Numbering of 16-bit register (d) Two-part 16-bit register

Copyright ©2016 Pearson Education, All Rights Reserved

Register transfer language

- Two-character names denote registers, such as R0, R1, DR, or SA.
- Arrows indicate data transfers. For example, we can copy the contents of source register R2 into the destination register R1 in one clock cycle.

$$R1 \leftarrow R2$$

- A conditional transfer is performed only if the Boolean condition in front of the operation is true. Below, we transfer R3 to R2 only when K = 1.

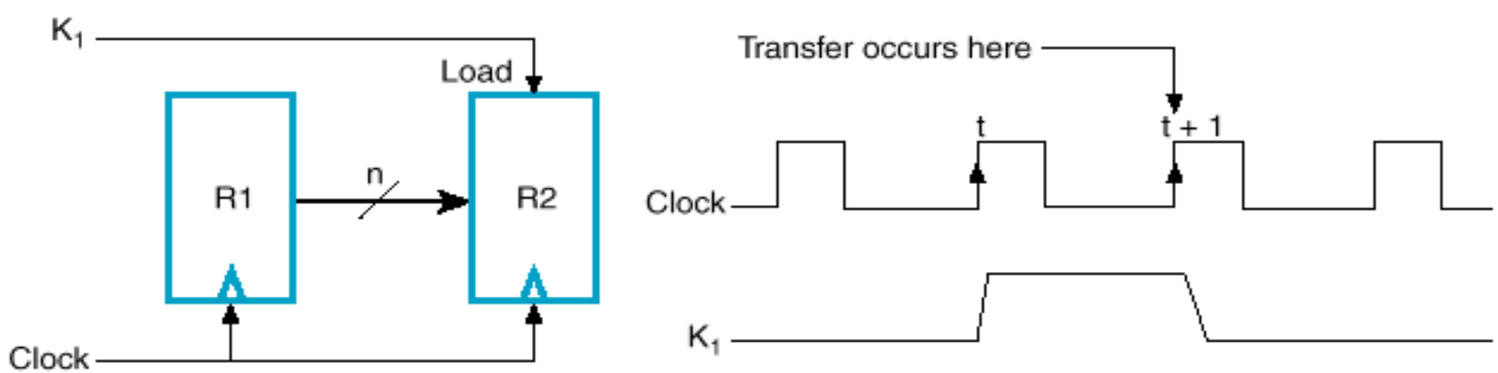
$$K: R2 \leftarrow R3$$

- Multiple transfers on the *same* clock cycle are separated by commas.

$$R1 \leftarrow R2, \quad K: R2 \leftarrow R3$$

Note: The contents of the source register do not change as a result of the transfer— only the contents of the destination register change.

- A statement that specifies a register transfer implies that datapath circuits are available from the outputs of the **source register** to the inputs of the **destination register** and that the destination register has a parallel load capability.
- Every statement written in register-transfer notation presupposes a hardware construct for implementing the transfer.
- All RTL statements occur in response to a clock tick. Normally we want a given transfer to occur not for every clock pulse, but only for specific values of the control signals.
  - RTL conditional statements:
    - e.g. If (K1 = 1) Then (R2 ← R1)
  - Control function notation (Colon, :)



- A comma is used to separate two or more register transfers that are executed at the same time.
  - e.g. Go: R2 ← R1, R4 ← R3

Register transfer operations

- We can apply arithmetic operations to registers.

$$R1 \leftarrow R2 + R3$$
$$R3 \leftarrow R1 - 1$$

- Bitwise logical operations can be expressed. We use special symbols for AND and OR to prevent confusion with arithmetic operations.

$R2 \leftarrow R1 \wedge R2$	bitwise AND
$R3 \leftarrow R0 \vee R1$	bitwise OR

- Finally, we can shift values left or right by one bit. The source register is not modified, and we assume that the shift input is always 0.

$R2 \leftarrow sl\ R1$	left shift
$R2 \leftarrow sr\ R1$	right shift

Bitwise operations

- Most computers also support logical operations like AND, OR and NOT, but extended to multi-bit words instead of just single bits.
- To apply a logical operation to two words X and Y, apply the operation on each pair of bits X_i and Y_i:

```
      1 0 1 1          1 0 1 1          1 0 1 1
AND   1 1 1 0    OR    1 1 1 0    XOR   1 1 1 0
-------------    -------------    -------------
      1 0 1 0          1 1 1 1          0 1 0 1
```

- Single operand logical operation: "complementing" all the bits in a number.

```
NOT   1 0 1 1
-------------
      0 1 0 0
```

TABLE 6-1
Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	$AR, R2, DR, IR$
Parentheses	Denotes a part of a register	$R2(1), R2(7{:}0), AR(L)$
Arrow	Denotes transfer of data	$R1 \leftarrow R2$
Comma	Separates simultaneous transfers	$R1 \leftarrow R2,\ R2 \leftarrow R1$
Square brackets	Specifies an address for memory	$DR \leftarrow M[AR]$

TABLE 6-3
Arithmetic Microoperations

Symbolic Designation	Description
$R0 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R0$
$R2 \leftarrow \overline{R2}$	Complement of the contents of $R2$ (1s complement)
$R2 \leftarrow \overline{R2} + 1$	2s complement of the contents of $R2$
$R0 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus 2s complement of $R2$ transferred to $R0$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ (count up)
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ (count down)

TABLE 6-4
Logic Microoperations

Symbolic Designation	Description
$R0 \leftarrow \overline{R1}$	Logical bitwise NOT (1s complement)
$R0 \leftarrow R1 \wedge R2$	Logical bitwise AND (clears bits)
$R0 \leftarrow R1 \vee R2$	Logical bitwise OR (sets bits)
$R0 \leftarrow R1 \oplus R2$	Logical bitwise XOR (complements bits)

TABLE 6-5
Examples of Shifts

		Eight-Bit Examples	
Type	Symbolic Designation	Source *R2*	After Shift: Destination *R1*
Shift left	$R1 \leftarrow \text{sl } R2$	10011110	00111100
Shift right	$R1 \leftarrow \text{sr } R2$	11100101	01110010

College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 9
Counters & RAM

Overview

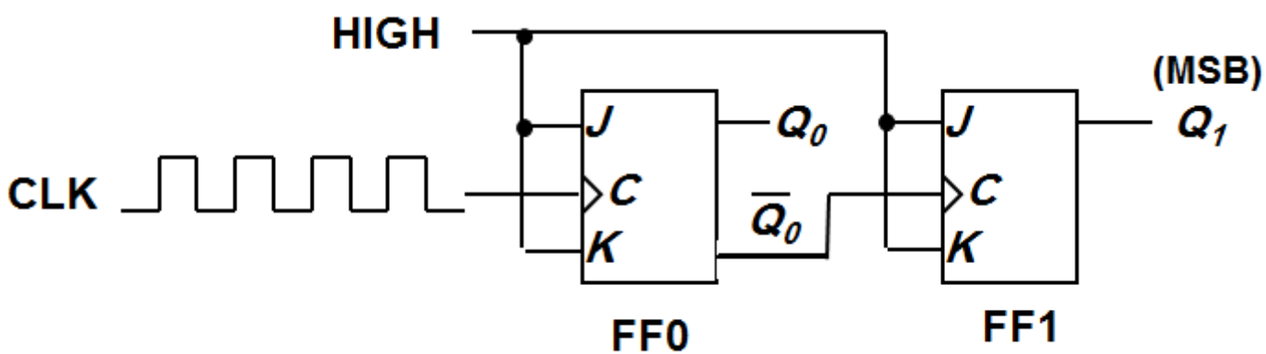
- Asynchronous (Ripple) Counters
- A complex Counter
- Introduction to RAM
  - Size
  - Reading a RAM
  - Writing to a RAM

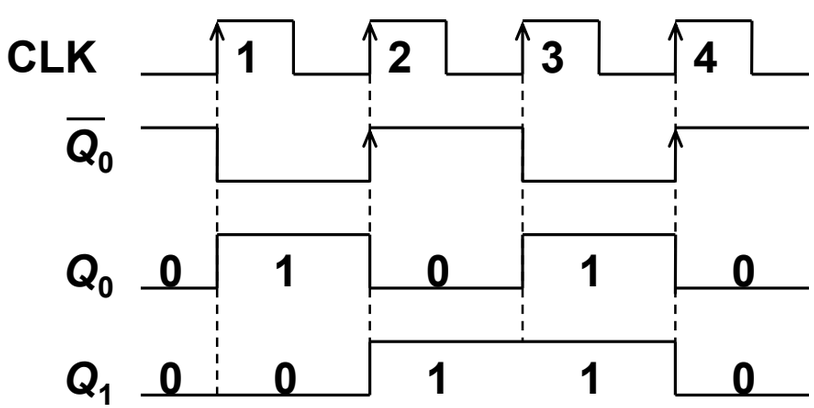
Chapter-6, 7

M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

Asynchronous (Ripple) Counters

- Example: 2-bit ripple binary counter.
- Output of one flip-flop is connected to the clock input of the next more-significant flip-flop.



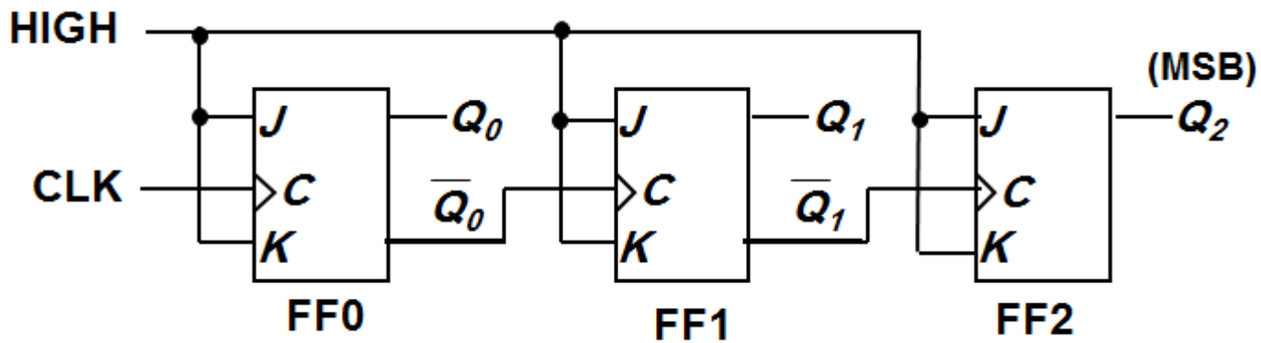


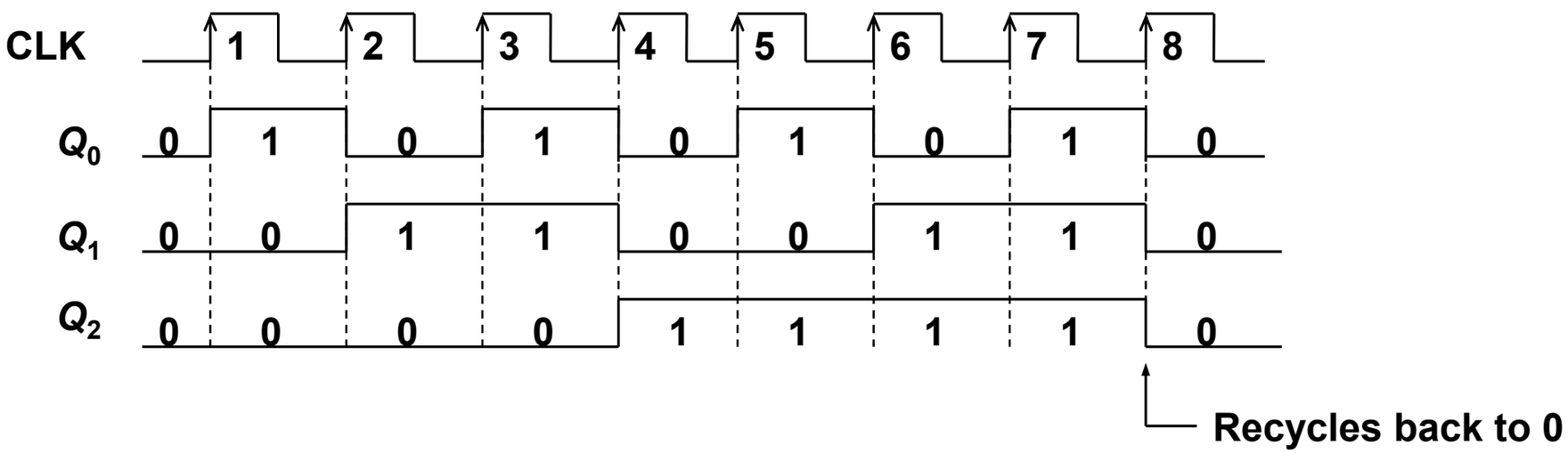
Timing diagram

00 → 01 → 10 → 11 → 00 ...

Asynchronous (Ripple) Counters

- Example: 3-bit ripple binary counter.

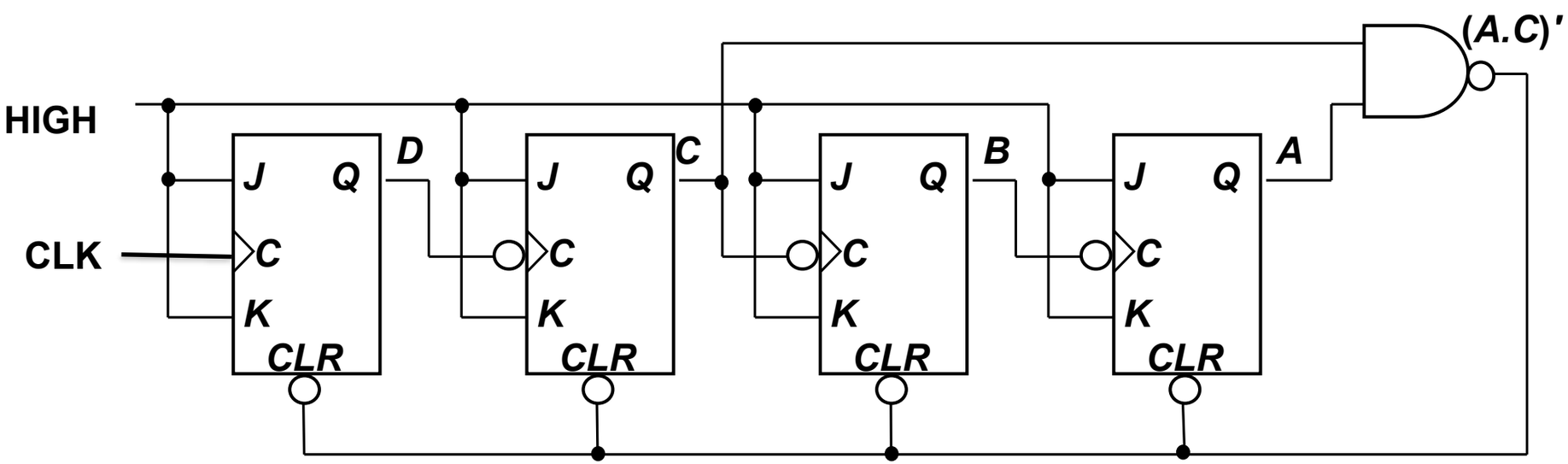




Asyn. Counters with MOD no. < 2^n

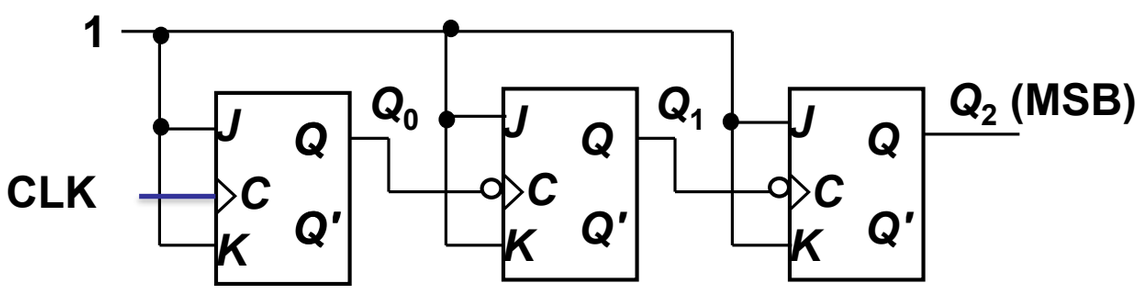
- Decade counters (or BCD counters) are counters with 10 states (modulus-10) in their sequence. They are commonly used in daily life (e.g.: utility meters, odometers, etc.).

- Design an asynchronous decade counter.

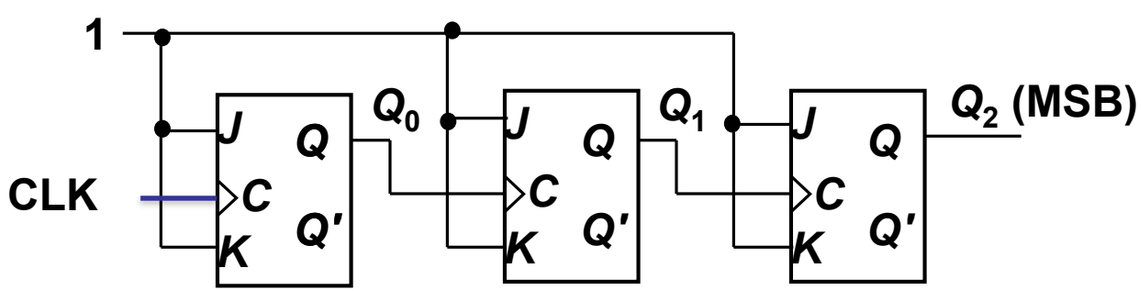


Asynchronous Down Counters

- So far we are dealing with *up counters*. *Down counters*, on the other hand, count downward from a maximum value to zero, and repeat.
- Example: A 3-bit binary (MOD-2^3) down counter.



3-bit binary up counter

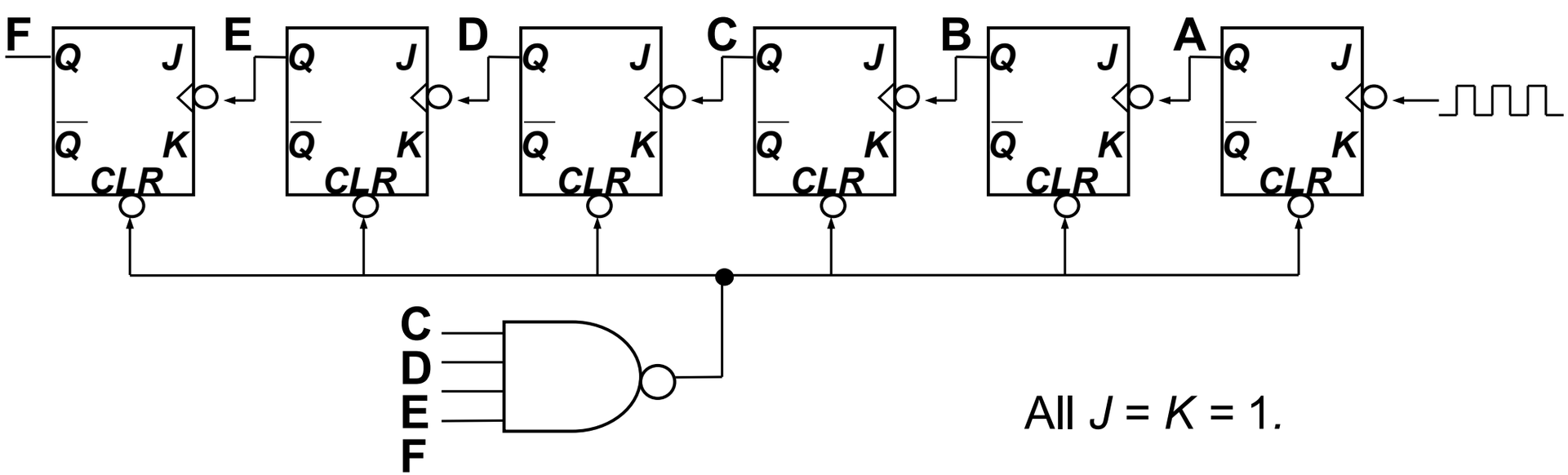


3-bit binary down counter

Asyn. Counters with MOD no. < 2^n

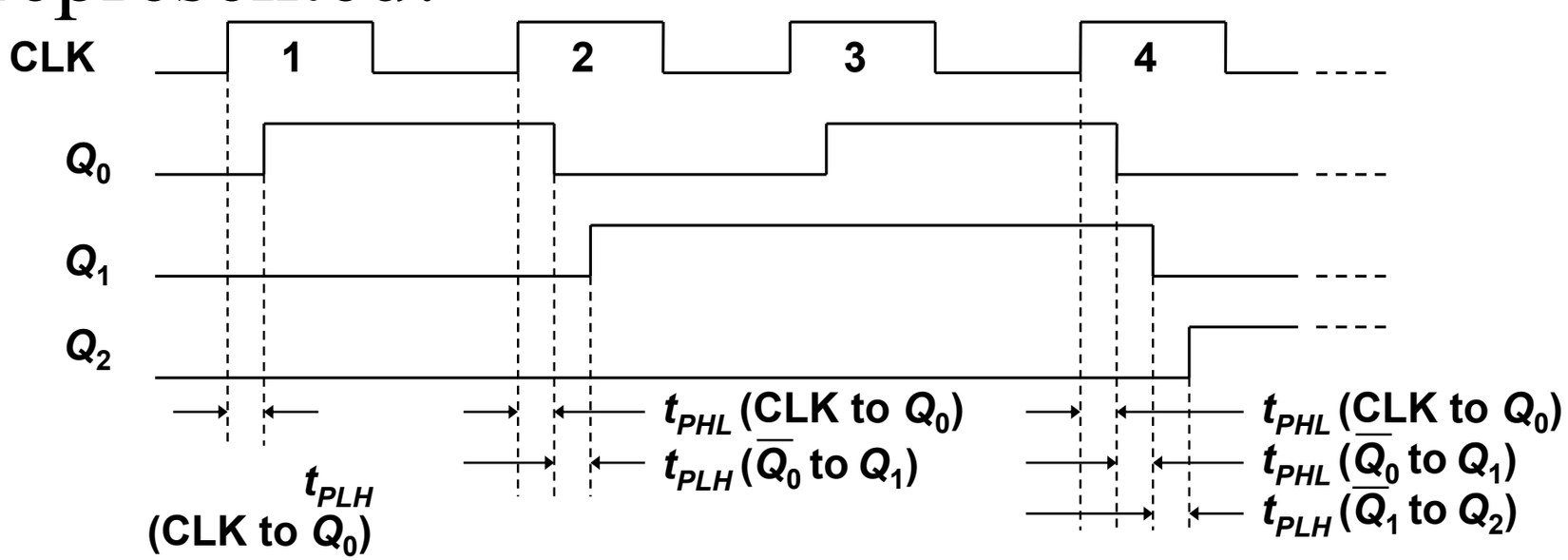
- *Exercise:* How to construct an asynchronous MOD-5 counter? MOD-7 counter? MOD-12 counter?

- *Question:* The following is a MOD-? counter?



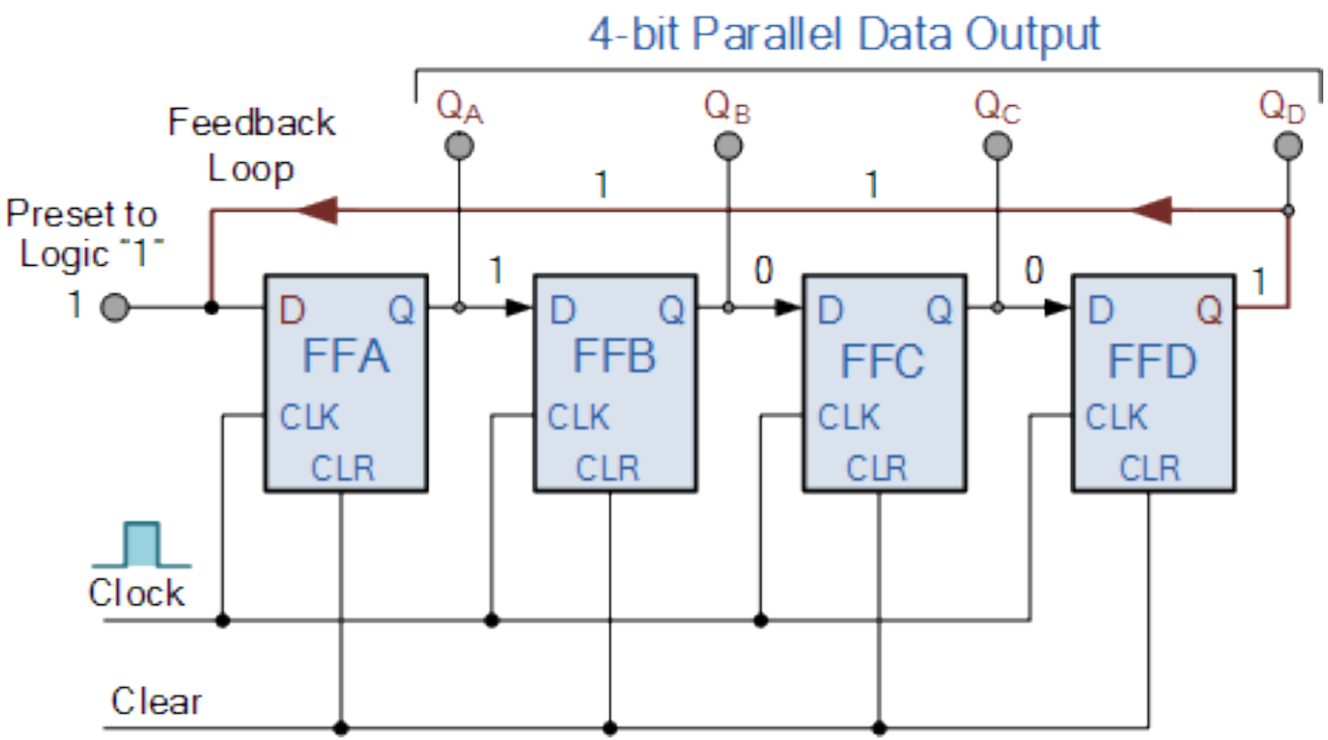
Asynchronous (Ripple) Counters

- Propagation delays in an asynchronous (ripple-clocked) binary counter.
- If the accumulated delay is greater than the clock pulse, some counter states may be misrepresented!



4-bit Ring Counter

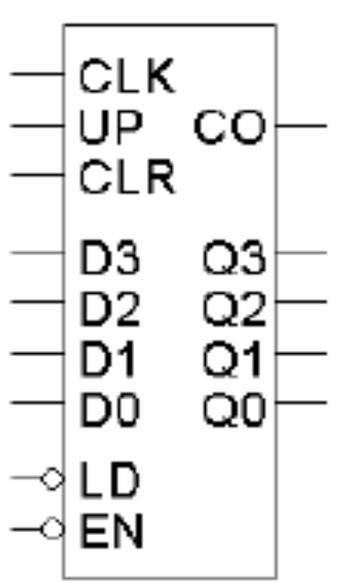
A "CLEAR" signal is firstly applied to all the flip-flops together in order to "RESET" their outputs to a logic "0" level and then a "PRESET" pulse is applied to the input of the first flip-flop (FFA) before the clock pulses are applied. This then places a single logic "1" value into the circuit of the ring counter.



So on each successive clock pulse, the counter circulates the same data bit between the four flip-flops over and over again around the "ring" every fourth clock cycle. But in order to cycle the data correctly around the counter we must first "load" the counter with a suitable data pattern as all logic "0's" or all logic "1's" outputted at each clock cycle would make the ring counter invalid.

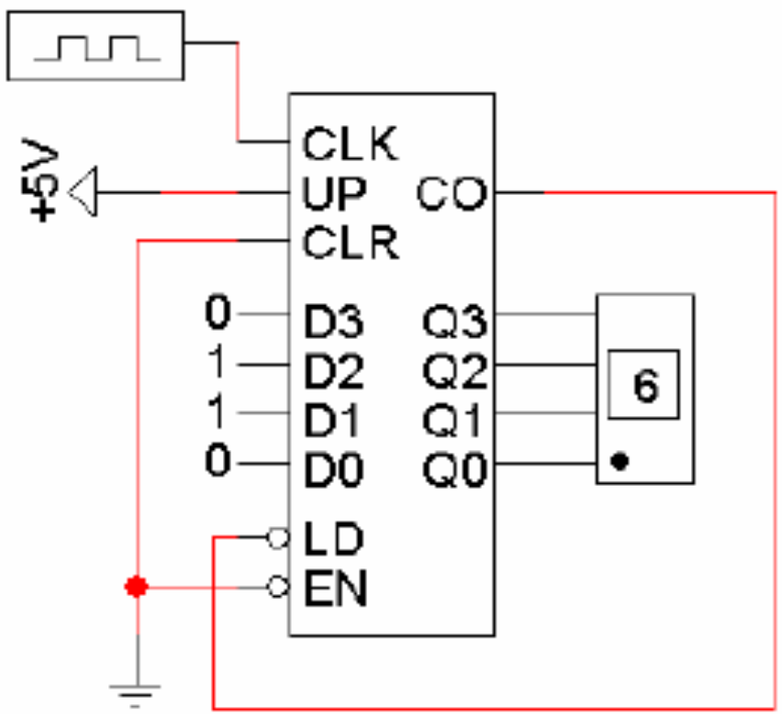
More complex counters

- More complex counters are also possible. The full-featured LogicWorks Counter-4 device below has several functions.
  - It can count up or down, depending on whether the UP input is 1 or 0.
  - You can clear the counter to 0000 asynchronously by setting CLR = 1.
  - You can perform a parallel load of D3-D0 when LD = 0.
  - The active-low EN input enables or disables the counter.
  - The "counter out" CO is normally 1, but becomes 0 when the counter reaches its maximum value of 1111 (if UP = 1) or 0000 (if UP = 0).



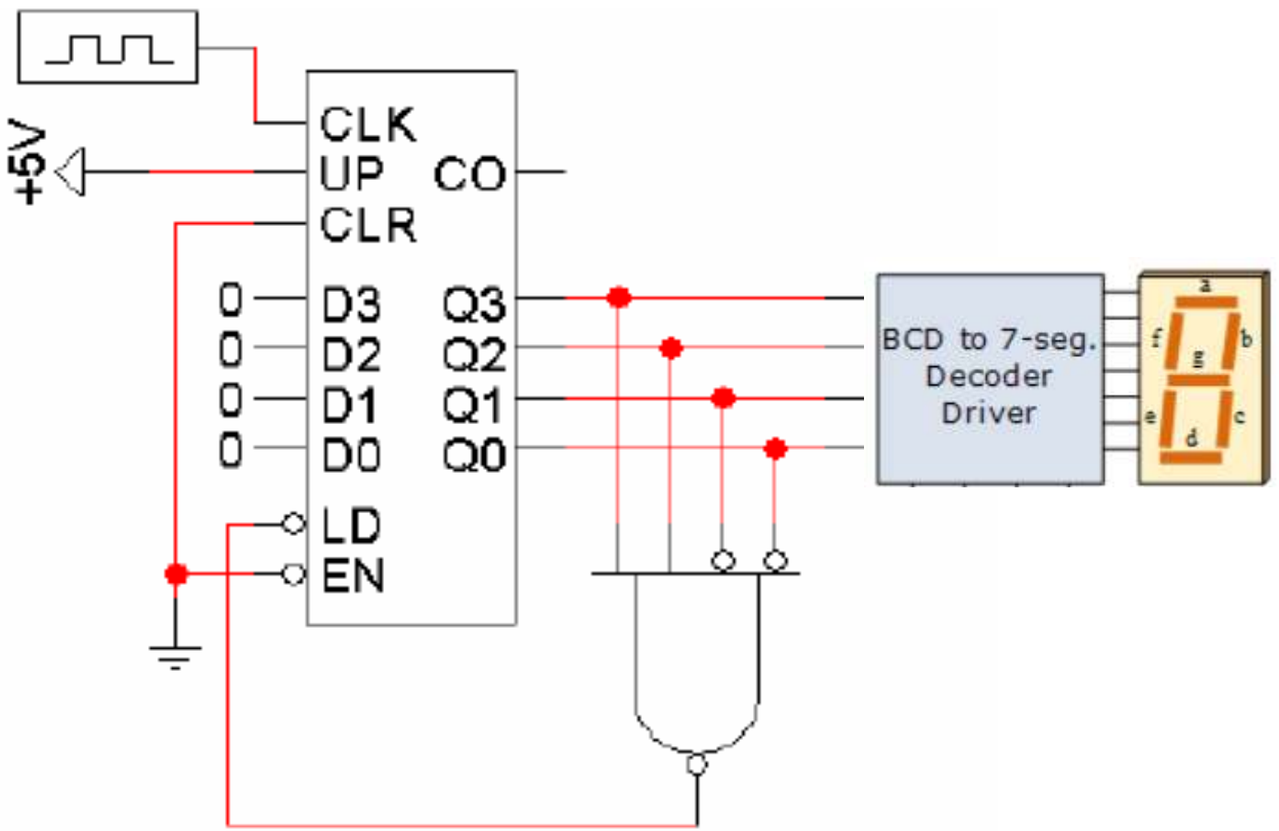
A restricted 4-bit counter

- We can also make a counter that "starts" at some value besides 0000.
- In the diagram below, when CO = 0 the LD signal forces the next state to be loaded from D3-D0.
- The result is this counter wraps from 1111 to 0110 (instead of 0000).



Another restricted counter

- We can also make a circuit that counts up to only 1100, instead of 1111.
- Here, when the counter output reaches 1100, the NAND gate forces the counter to load, so the next state becomes 0000.

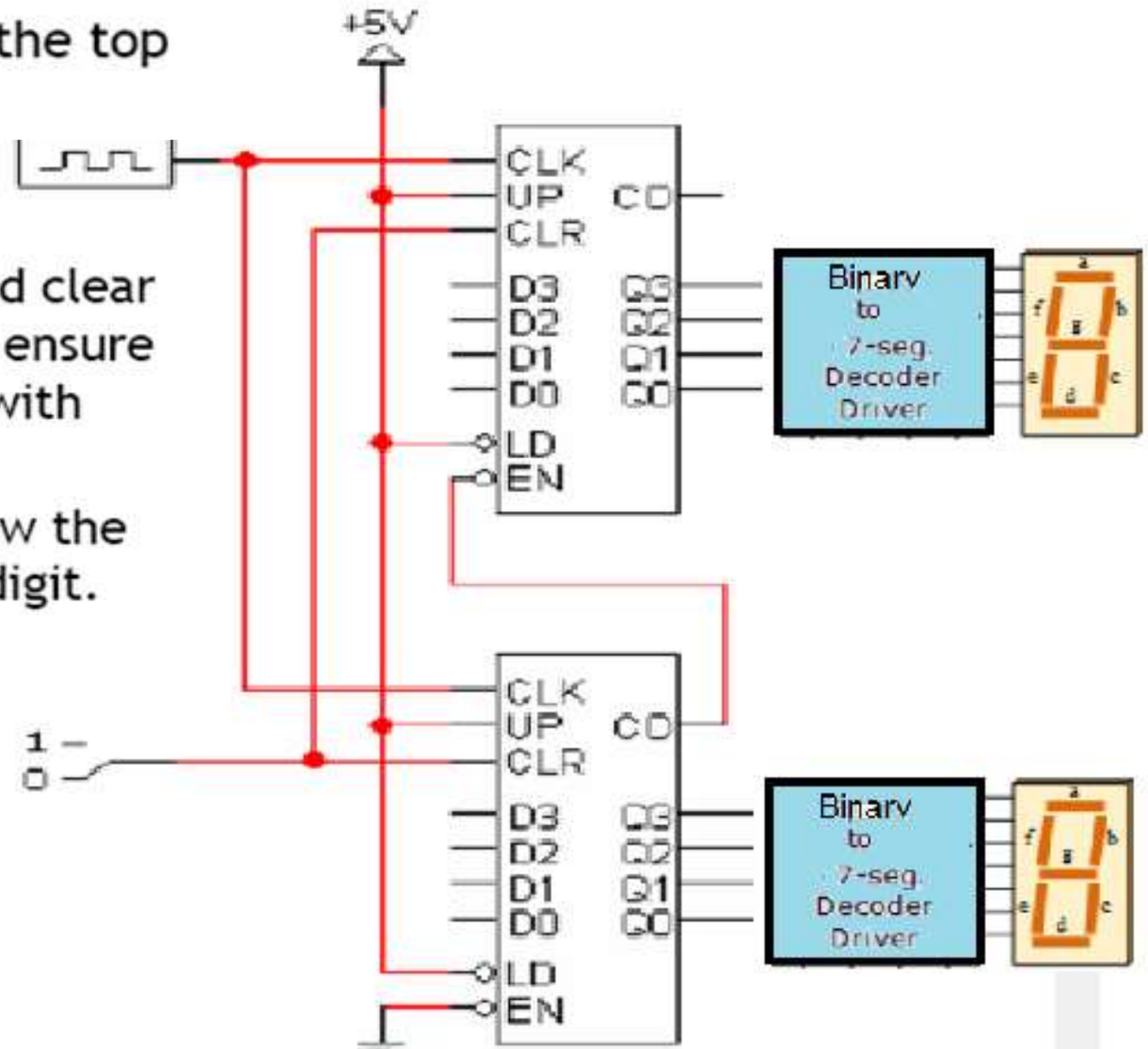


Here is an 8-bit counter made from two 4-bit

- The bottom device represents the least significant four bits, while the top counter represents the most significant four bits.
- When the bottom counter reaches 1111 (i.e., when CO = 0), it enables the top counter for one cycle.

The two four-bit counters share clock and clear inputs. Sharing the clock is important to ensure that the two counters are synchronized with respect to each other.

We've used Hex Display units here to view the four-bit output as a single hexadecimal digit.



Introduction to RAM

- Random-access memory, or RAM, provides large quantities of temporary storage in a computer system.
- Remember the basic capabilities of a memory.
  - It should be able to store a value.
  - You should be able to read the value that was saved.
  - You should be able to change the stored value.
- A RAM is similar, except that it can store *many* values.
  - An address will specify which memory value we're interested in.
  - Each value can be a multiple-bit word (e.g., 32 bits).
- We'll refine the memory properties as follows.

A RAM should be able to:

1. Store many words, one per address
2. Read the word that was saved at a particular address
3. Change the word that's saved at a particular address

Picture of memory

- You can think of computer memory as being one big array of data.
  - The address serves as an array index.
  - Each address refers to one word of data.
- You can read or modify the data at any given memory address, just like you can read or modify the contents of an array at any given index.
- If you've worked with pointers in C or C++, then you've already worked with memory addresses.

Address	Data
00000000	
00000001	
00000010	
.	
.	
.	
.	
.	
.	
.	
.	
.	
.	
.	
FFFFFFFD	
FFFFFFFE	
FFFFFFFF	

Block diagram of RAM

CS	WR	Memory operation
0	x	None
1	0	Read from ADRS
1	1	Write DATA to ADRS

- This block diagram introduces the main interface to RAM.
  - A Chip Select, CS, enables or disables the RAM.
  - ADRS specifies the address or location to read from or write to.
  - WR selects between reading from or writing to the memory.
    - To read from memory, WR should be set to 0.
      OUT will be the *n*-bit value stored at ADRS.
    - To write to memory, we set WR = 1.
      DATA is the *n*-bit value to save in memory.
- This interface makes it easy to combine RAMs together, as we'll see.

Memory sizes

- We refer to this as a $2^k \times n$ memory.
  - There are k address lines, which can specify one of 2^k addresses.
  - Each address contains an n-bit word.

- For example, a $2^{24} \times 16$ RAM contains 2^{24} = 16M words, each 16 bits long.
  - The RAM would need 24 address lines.
  - The total storage capacity is $2^{24} \times 16 = 2^{28}$ bits.

Size matters!

- Memory sizes are usually specified in numbers of bytes (8 bits).
- The 2^{28}-bit memory on the previous page has a capacity of 2^{25} bytes.

$$2^{28} \text{ bits} / 8 \text{ bits per byte} = 2^{25} \text{ bytes}$$

- With the abbreviations below, this is equivalent to 32 megabytes.

$$2^{25} \text{ bytes} = 2^5 \times 2^{20} \text{ bytes} = 32 \text{ MB}$$

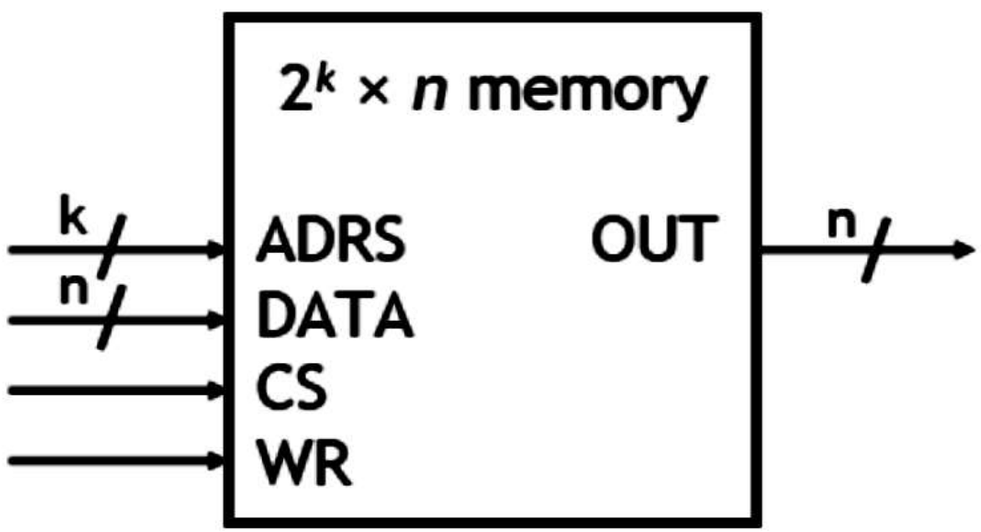
	Prefix	Base 2	Base 10
K	Kilo	2^{10} = 1,024	10^3 = 1,000
M	Mega	2^{20} = 1,048,576	10^6 = 1,000,000
G	Giga	2^{30} = 1,073,741,824	10^9 = 1,000,000,000

Reading RAM

- To *read* from a RAM, the controlling circuit must take several steps.
  1. Enable the chip by ensuring CS = 1.
  2. Select the read operation, by setting WR = 0.
  3. Send the desired address to the ADRS input.
  4. The contents of that address appear on OUT after a little while.
- Notice that the DATA input is unused for read operations.

Writing RAM

- To *write* to this RAM, you need to do the following tasks.
  1. Enable the chip by setting CS = 1.
  2. Select the write operation, by setting WR = 1.
  3. Send the desired address to the ADRS input.
  4. Send the word to store to the DATA input.
- The output OUT is not needed for memory write operations.



RTL and Memory

- While memory transfers are similar to register transfers, we usually identify them differently. Specifically, memory to register transfers are called ***read* operations**, while register to memory transfers are called ***write* operations**. Both require specification of the memory location to be used (which can be done through a special register or a special bus) and a storage location which will hold the result of a read or which holds the data to be written.

- RTL expressions for a **Read** operation, assuming the use of an address registers:

$$AR \leftarrow address$$
$$DR \leftarrow M[AR]$$

- RTL expressions for a **Write** operation, assuming use of a data register:

$$AR \leftarrow address$$
$$DR \leftarrow value$$
$$M[AR] \leftarrow DR$$

- Register to Memory Transfers are denoted using square brackets surrounding the memory address.
  - e.g. DR ← M[AR] (Read operation)
  - e.g. M[AR] ← DR (Write operation)

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*AR, R2, DR, IR*
Parentheses	Denotes a part of a register	*R2*(1), *R2*(7:0), *AR*(*L*)
Arrow	Denotes transfer of data	*R1* ← *R2*
Comma	Separates simultaneous transfers	*R1* ← *R2*, *R2* ← *R1*
Square brackets	Specifies an address for memory	*DR* ← *M*[*AR*]

College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 10
Arithmetic-logic Units

Unit 10: Arithmetic-logic Units

Overview

- Arithmetic Unit Design
  - Primitive gates base implementation
  - MUX-based implementation
- Logic Unit Design
- Arithmetic-logic Unit Design

Chapter-8

M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

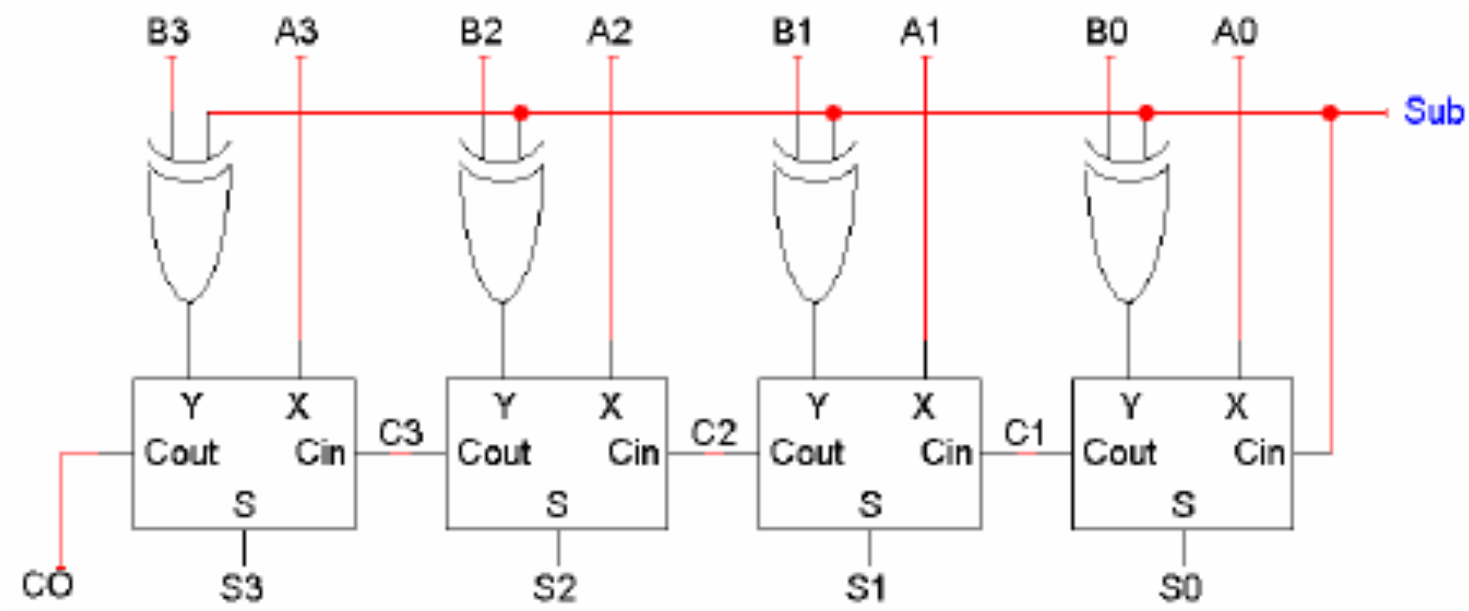
Remember:

An adder-subtractor circuit

- XOR gates let us selectively complement the B input.

$$X \oplus 0 = X \qquad\qquad X \oplus 1 = X'$$

- When Sub = 0, the XOR gates output B3 B2 B1 B0 and the carry in is 0. The adder output will be A + B + 0, or just A + B.
- When Sub = 1, the XOR gates output B3' B2' B1' B0' and the carry in is 1. Thus, the adder output will be a two's complement subtraction, A - B.

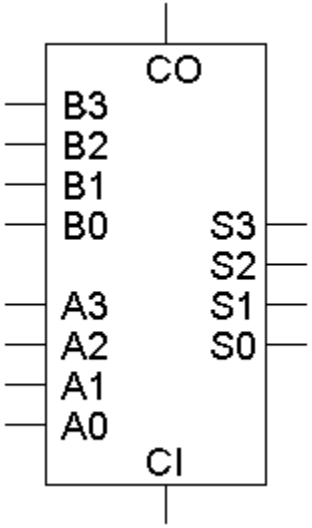


Arithmetic-logic units

- **An arithmetic-logic unit, or ALU, performs many different arithmetic and logic operations. The ALU is the "heart" of a processor—you could say that everything else in the CPU is there to support the ALU.**
- **Here's the plan:**
  - **We'll show an arithmetic unit first, by building off ideas from the adder-subtractor circuit.**
  - **Then we'll talk about logic operations a bit, and build a logic unit.**
  - **Finally, we put these pieces together using multiplexers.**
- **We show the same examples as from the book.**

The four-bit adder

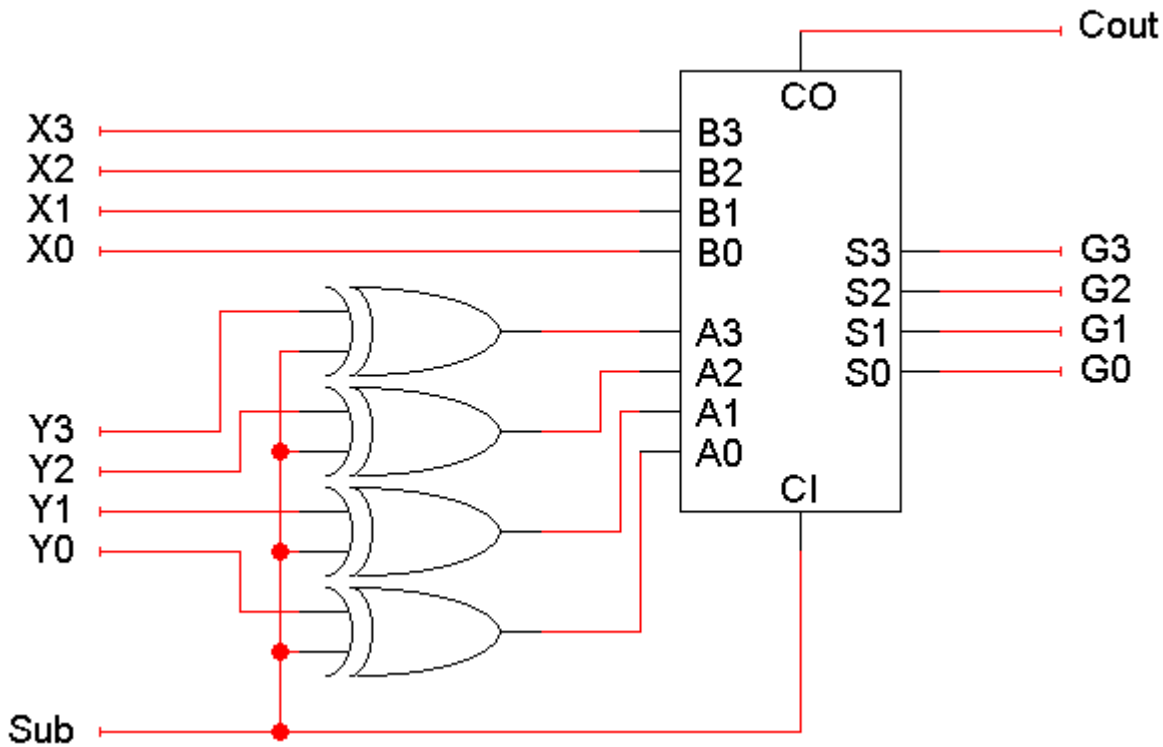
- **The basic four-bit adder *always* computes S = A + B + CI.**



- **But by changing what goes into the adder inputs A, B and CI, we can change the adder output S.**
- **This is also what we did to build the combined adder-subtractor circuit.**

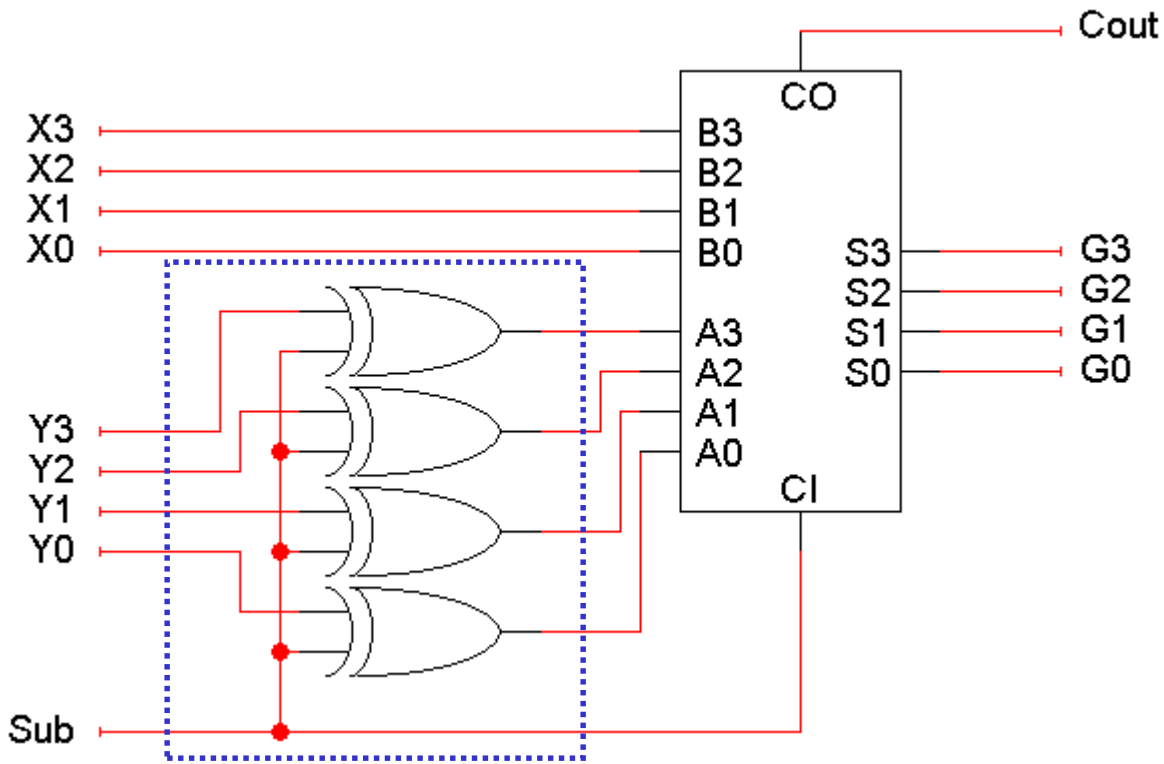
It's the adder-subtractor again!

- Here the signal Sub and some XOR gates alter the adder inputs.
  - When Sub = 0, the adder inputs A, B, CI are Y, X, 0, so the adder produces G = X + Y + 0, or just X + Y.
  - When Sub = 1, the adder inputs are Y', X and 1, so the adder output is G = X + Y' + 1, or the two's complement operation X - Y.



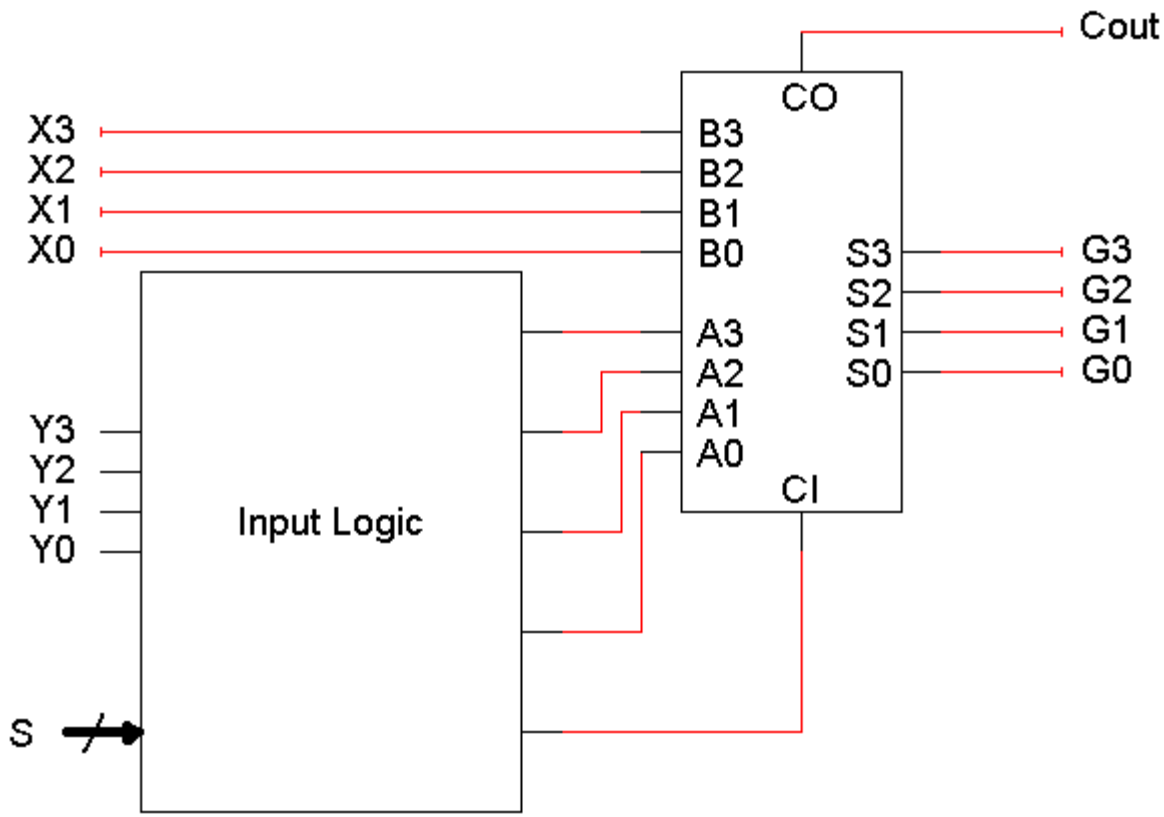
The multi-talented adder

- So we have one adder performing two separate functions.
- "Sub" acts like a function select input which determines whether the circuit performs addition or subtraction.
- Circuit-wise, all "Sub" does is modify the adder's inputs A and CI.



Modifying the adder inputs

- **By following the same approach, we can use an adder to compute *other* functions as well.**
- **We just have to figure out which functions we want, and then put the right circuitry into the "Input Logic" box .**



Some more possible functions

- We already saw how to set adder inputs A, B and CI to compute either X + Y or X - Y.
- How can we produce the <u>increment function</u> G = X + 1?

One way: Set A = 0000, B = X, and CI = 1

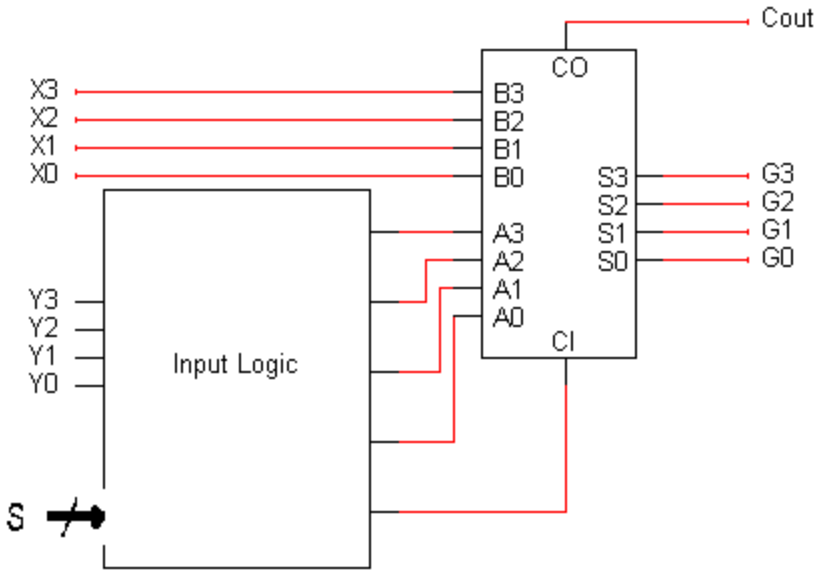
- How about <u>decrement</u>: G = X - 1?

A = 1111 (-1), B = X, CI = 0

- How about <u>transfer</u>: G = X? (This can be useful.)

A = 0000, B = X, CI = 0

This is almost the same as the increment function!



The role of CI

- **The transfer and increment operations have the same A and B inputs, and differ only in the CI input.**
- **In general we can get additional functions (not all of them useful) by using both CI = 0 and CI = 1.**
- **Another example:**
  - **<u>Two's-complement subtraction</u> is obtained by setting A = Y', B = X, and CI = 1, so G = X + Y' + 1.**
  - **If we keep A = Y' and B = X, but set CI to 0, we get G = X + Y'. This turns out to be a ones' complement subtraction operation.**

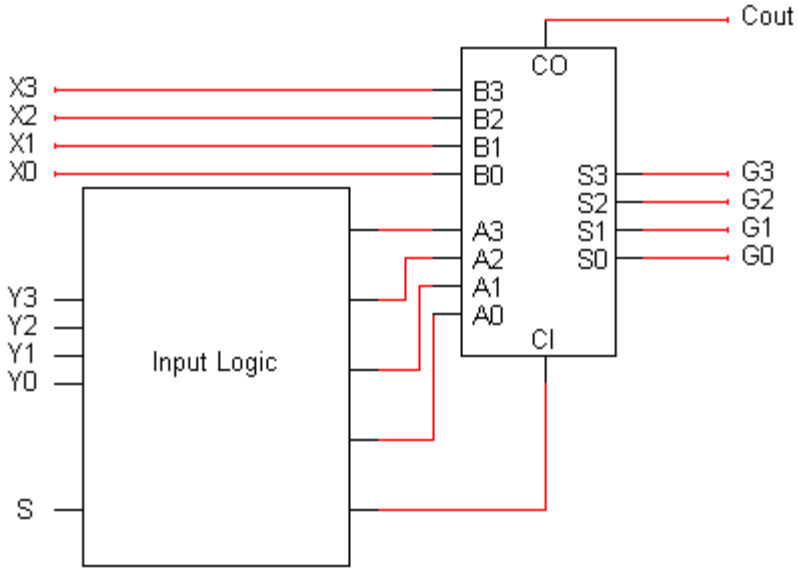
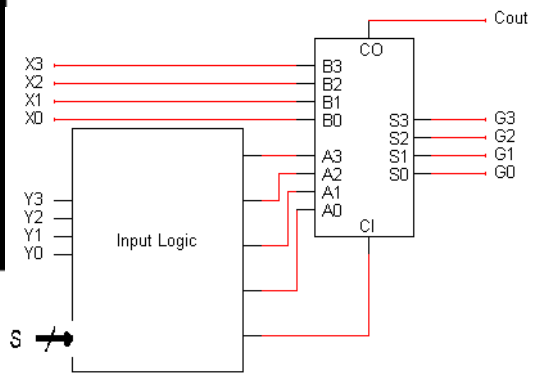


Table of arithmetic functions

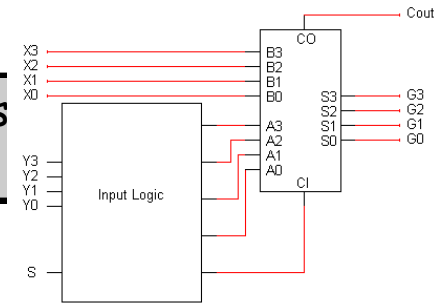
- **Here are some of the different possible arithmetic operations.**
- **We'll need some way to specify which function we're interested in, so we've *randomly assigned* a selection code to each operation.**

S_2	S_1	S_0	Arithmetic operation	
0	0	0	X	(transfer)
0	0	1	X + 1	(increment)
0	1	0	X + Y	(add)
0	1	1	X + Y + 1	
1	0	0	X + Y'	(1C subtraction)
1	0	1	X + Y' + 1	(2C subtraction)
1	1	0	X - 1	(decrement)
1	1	1	X	(transfer)



Mapping the table to an adder

- **This second table shows what the adder's inputs should be for each of our eight desired arithmetic operations.**



Selection code S_2	S_1	S_0	Desired arithmetic operation G (A + B + CI)		Required adder inputs A	B	CI
0	0	0	X	(transfer)	0000	X	0
0	0	1	X + 1	(increment)	0000	X	1
0	1	0	X + Y	(add)	Y	X	0
0	1	1	X + Y + 1		Y	X	1
1	0	0	X + Y′	(1C subtraction)	Y′	X	0
1	0	1	X + Y′ + 1	(2C subtraction)	Y′	X	1
1	1	0	X – 1	(decrement)	1111	X	0
1	1	1	X	(transfer)	1111	X	1

  - **Adder input CI is always the same as selection code bit S_0.**
  - **B is always set to X.**
  - **A depends only on S_2 and S_1.**
- **These equations depend on both the desired operations and the assignment of selection codes.**

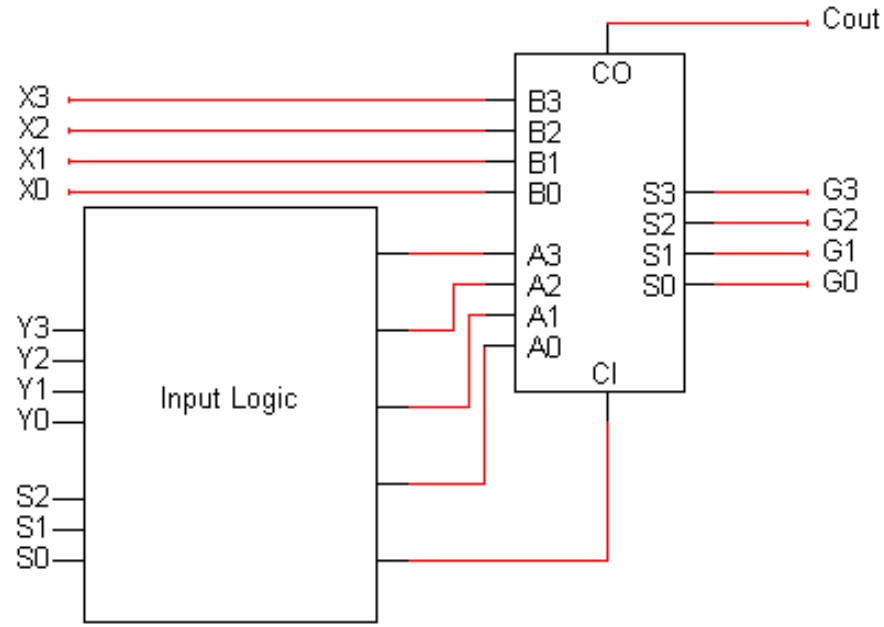
Building the input logic

- **All we need to do is compute the adder input A, given the arithmetic unit input Y and the function select code S (actually just S_2 and S_1).**
- **Here is an abbreviated truth table:**

S_2	S_1	A
0	0	0000
0	1	Y
1	0	Y'
1	1	1111

inputs output

S_2	S_1	Y_i	A_i
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



- **We want to pick one of these four possible values for A, depending on S_2 and S_1.**

Primitive gate-based input logic

- We could build this circuit using primitive gates.
- If we want to use K-maps for simplification, then we should first expand out the abbreviated truth table.
  - The Y that appears in the output column (A) is actually an input.
  - We make that explicit in the table on the right.
- Remember A and Y are each 4 bits long!

S_2	S_1	A
0	0	0000
0	1	Y
1	0	Y'
1	1	1111

inputs output

S_2	S_1	Y_i	A_i
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Primitive gate implementation

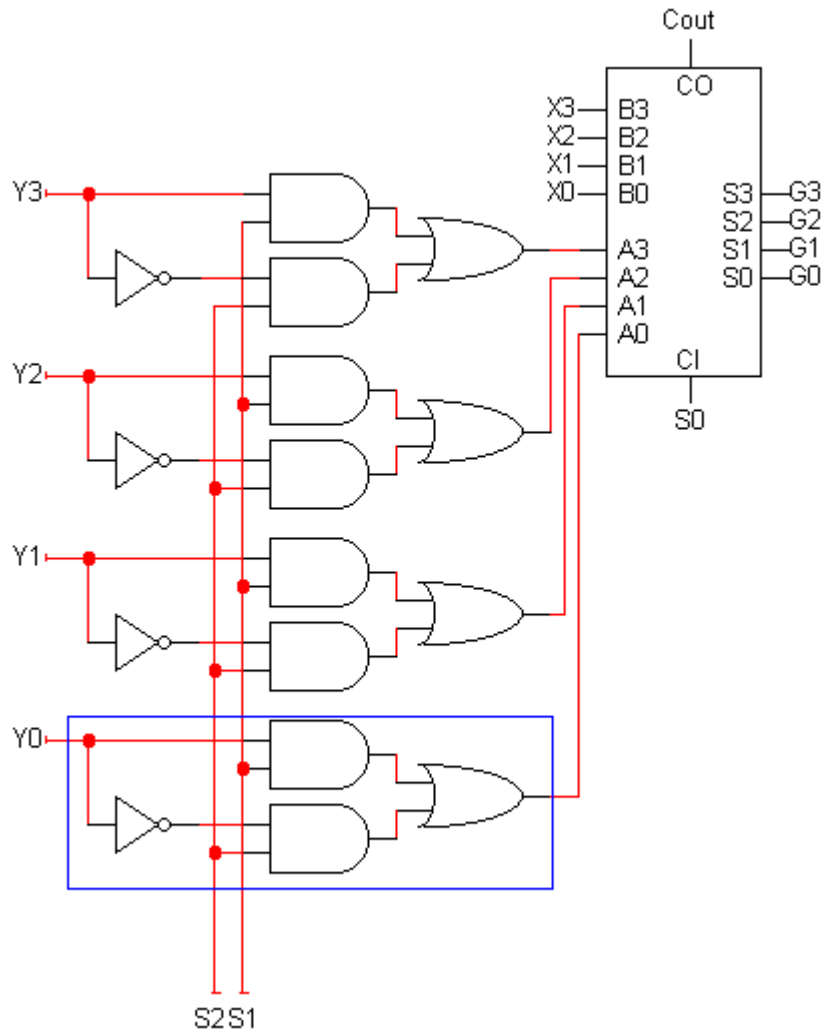
- **From the truth table, we can find an MSP:**

			S_1	
	0	0	1	0
S_2	1	0	1	1
		Y_i		

$$A_i = S_2 Y_i' + S_1 Y_i$$

- **Again, we have to repeat this once for each bit Y3-Y0, connecting to the adder inputs A3-A0.**

- **This completes our arithmetic unit.**

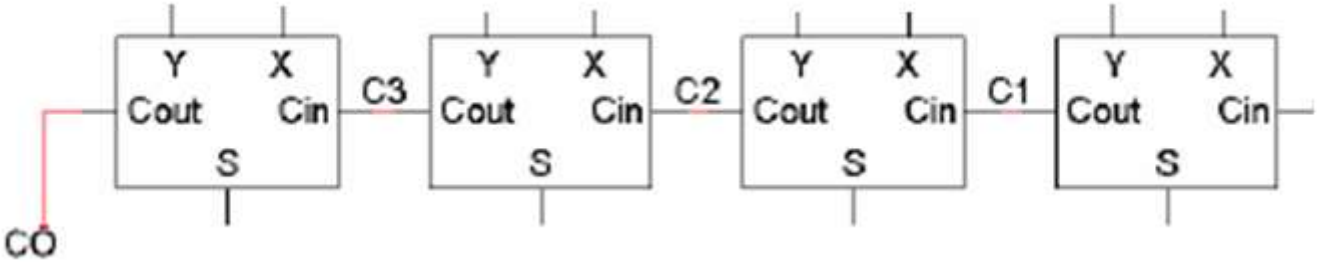


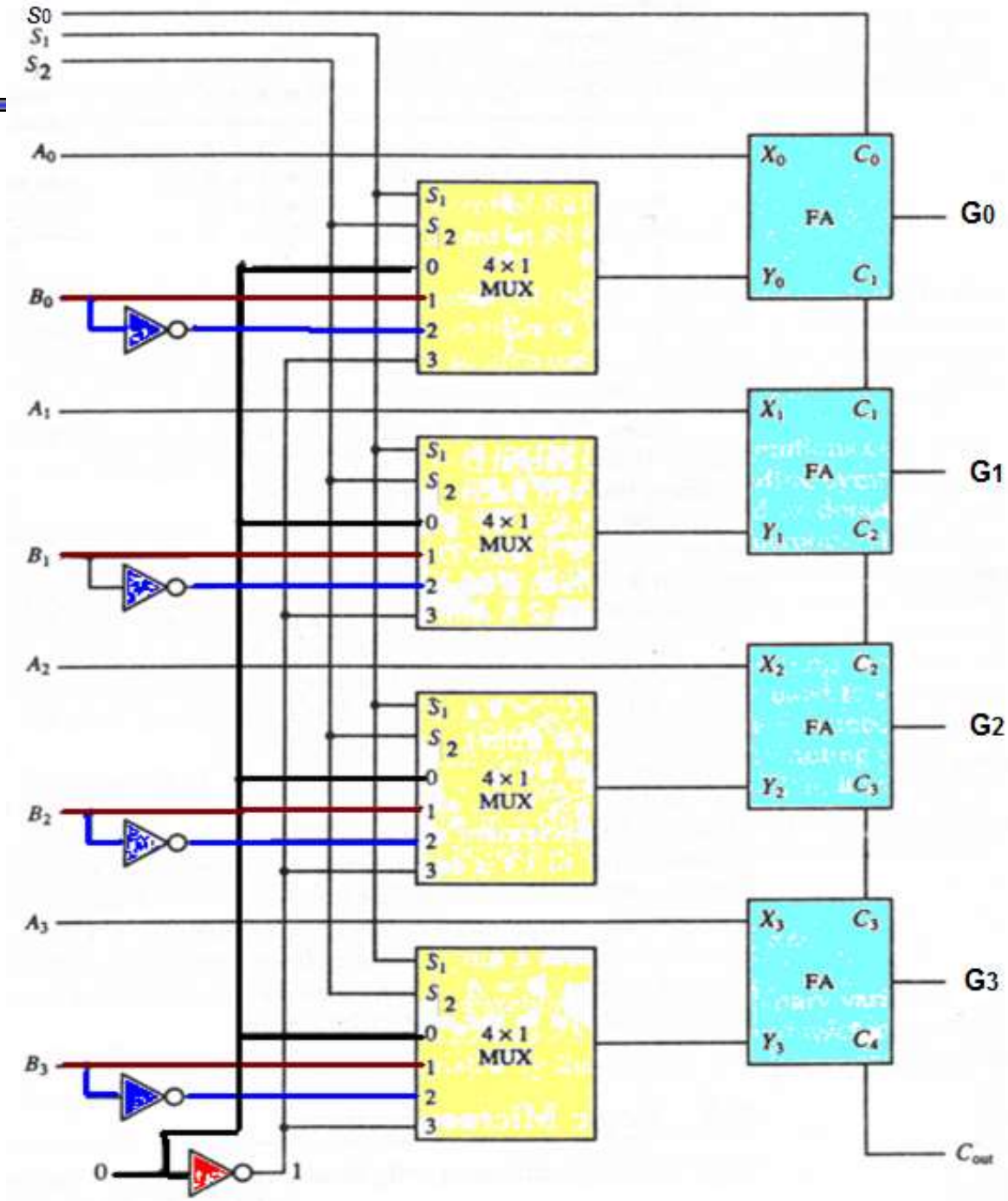
Multiplexer-based implementation

Alternative Implementation using 4 bit adder circuit and multiplexers

Selection code			Desired arithmetic operation		Required adder inputs		
S_2	S_1	S_0	G (A + B + CI)		Y	X	CI
0	0	0	A	(transfer)	0000	A	0
0	0	1	A + 1	(increment)	0000	A	1
0	1	0	A + B	(add)	B	A	0
0	1	1	A + B + 1		B	A	1
1	0	0	A + B'	(1C subtraction)	B'	A	0
1	0	1	A + B' + 1	(2C subtraction)	B'	A	1
1	1	0	A – 1	(decrement)	1111	A	0
1	1	1	A	(transfer)	1111	A	1

S_2	S_1	Y
0	0	0000
0	1	B
1	0	B'
1	1	1111





S_2	S_1	y
0	0	0000
0	1	B
1	0	B'
1	1	1111

Bitwise operations

- **Most computers also support logical operations like AND, OR and NOT, but extended to multi-bit words instead of just single bits.**
- **To apply a logical operation to two words X and Y, apply the operation on each pair of bits X_i and Y_i:**

```
      1 0 1 1          1 0 1 1          1 0 1 1
AND   1 1 1 0    OR    1 1 1 0    XOR   1 1 1 0
-------------    -------------    -------------
      1 0 1 0          1 1 1 1          0 1 0 1
```

- **Single operand logical operation: "complementing" all the bits in a number.**

```
NOT   1 0 1 1
-------------
      0 1 0 0
```

Bitwise operations in programming

- **Languages like C, C++ and Java provide bitwise logical operations:**

  & (AND) | (OR) ^ (XOR) ~ (NOT)

- **These operations treat each integer as a bunch of individual bits:**

  13 & 25 = 9 because 01101 & 11001 = 01001

- **They are *not* the same as the operators &&, || and !, which treat each integer as a single logical value (0 is false, everything else is true):**

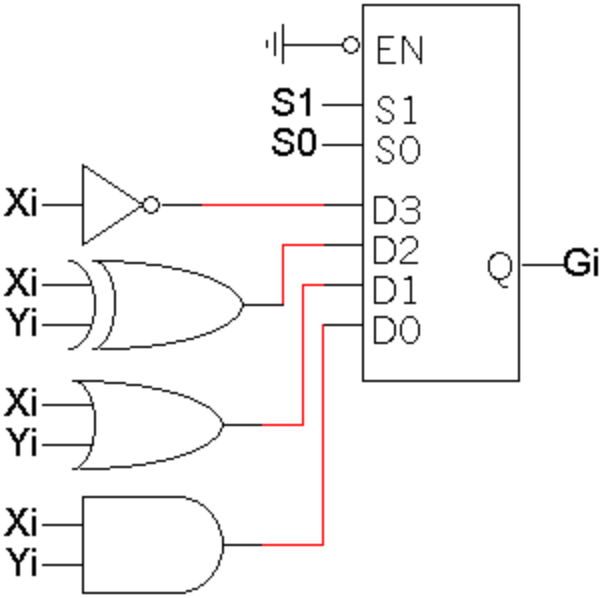
  13 && 25 = 1 because true && true = true

- **Bitwise operators are often used in programs to set a bunch of Boolean options, or flags, with one argument.**
- **Easy to represent sets of fixed universe size with bits:**
  - **1: is member, 0 not a member. Unions: OR, Intersections: AND**

Defining a logic unit

- **A logic unit supports different logical functions on two multi-bit inputs X and Y, producing an output G.**
- **This abbreviated table shows four possible functions and assigns a selection code S to each.**

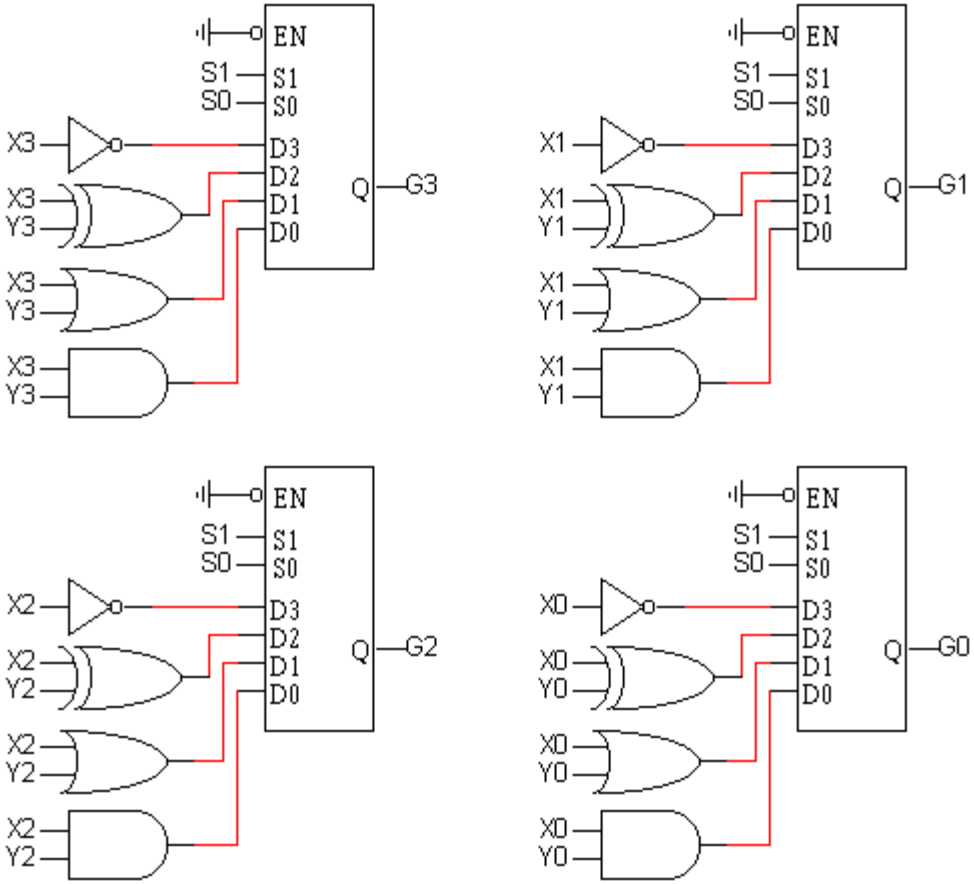
S_1	S_0	Output
0	0	$G_i = X_i Y_i$
0	1	$G_i = X_i + Y_i$
1	0	$G_i = X_i \oplus Y_i$
1	1	$G_i = X_i'$



- **We'll just use multiplexers and some primitive gates to implement this.**
- **Again, we need one multiplexer for *each bit* of X and Y.**

Our simple logic unit

- **Inputs:**
  - **X (4 bits)**
  - **Y (4 bits)**
  - **S (2 bits)**
- **Outputs:**
  - **G (4 bits)**



Combining the arithmetic and logic units

- **Now we have two pieces of the puzzle:**
  - **An arithmetic unit that can compute eight functions on 4-bit inputs.**
  - **A logic unit that can perform four functions on 4-bit inputs.**

- **We can combine these together into a single circuit, an arithmetic-logic unit (ALU).**

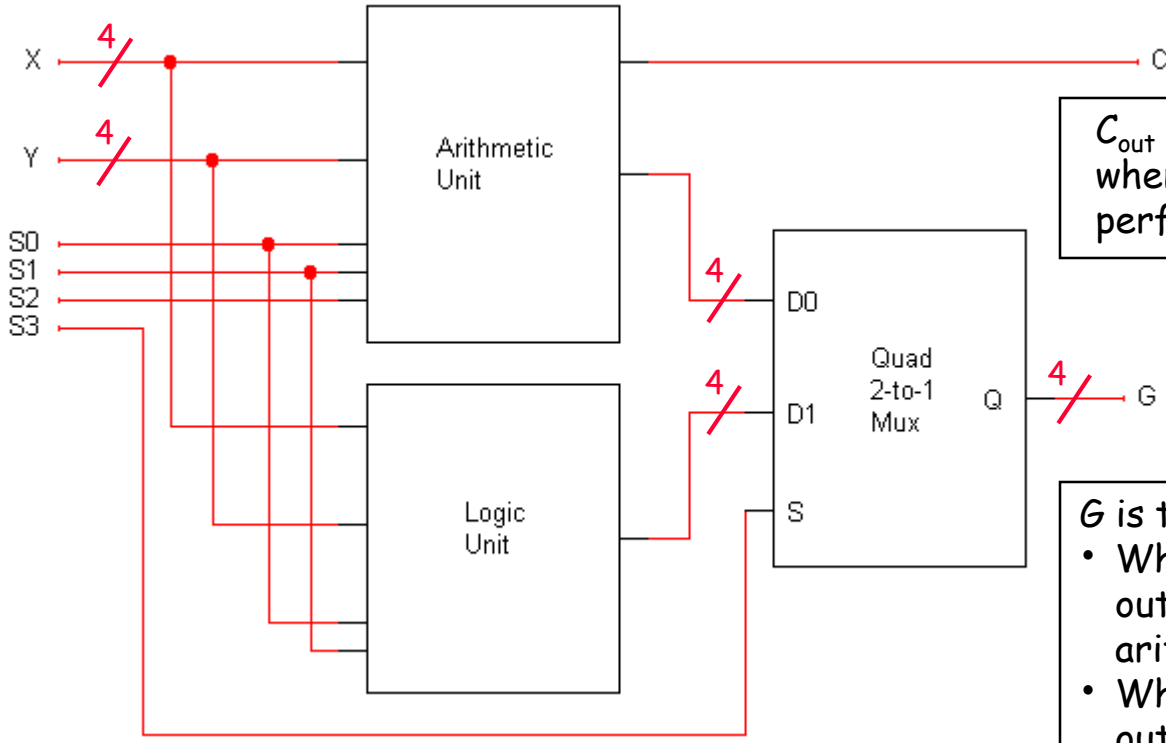
Our ALU function table

- **This table shows a sample function table for an ALU.**
- **All of the arithmetic operations have S_3=0, and all of the logical operations have S_3=1.**
- **These are the same functions we saw when we built our arithmetic and logic units a few minutes ago.**
- **Since our ALU only has 4 logical operations, we don't need S_2. The operation done by the logic unit depends only on S_1 and S_0.**

S_3	S_2	S_1	S_0	Operation
0	0	0	0	$G = X$
0	0	0	1	$G = X + 1$
0	0	1	0	$G = X + Y$
0	0	1	1	$G = X + Y + 1$
0	1	0	0	$G = X + Y'$
0	1	0	1	$G = X + Y' + 1$
0	1	1	0	$G = X - 1$
0	1	1	1	$G = X$
1	x	0	0	$G = X$ and Y
1	x	0	1	$G = X$ or Y
1	x	1	0	$G = X \oplus Y$
1	x	1	1	$G = X'$

A complete ALU circuit

The / and 4 on a line indicate that it's actually *four* lines.



C_{out} should be ignored when logic operations are performed (when S3=1).

G is the final ALU output.

- When S3 = 0, the final output comes from the arithmetic unit.
- When S3 = 1, the output comes from the logic unit.

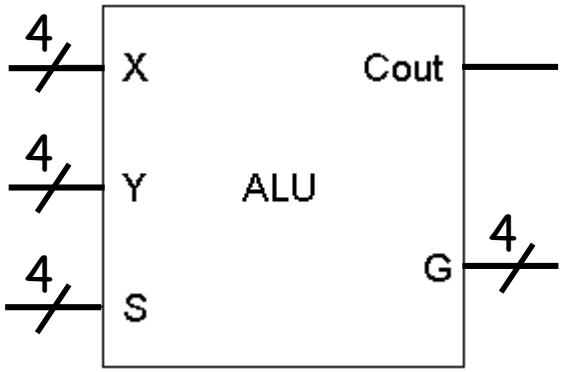
The arithmetic and logic units share the select inputs S1 and S0, but only the arithmetic unit uses S2.

Comments on the multiplexer

- ***Both*** **the arithmetic unit and the logic unit are "active" and produce outputs.**
  - **The mux determines whether the final result comes from the arithmetic or logic unit.**
  - **The output of the other one is effectively ignored.**
- **Our hardware scheme may seem like wasted effort, but it's not really.**
  - **"Deactivating" one or the other wouldn't save that much time.**
  - **We have to build hardware for both units anyway, so we might as well run them together.**
- **This is a very common use of multiplexers in logic design.**

The completed ALU

- **This ALU is a good example of hierarchical design.**
  - **With the 12 inputs, the truth table would have had $2^{12} = 4096$ lines. That's an awful lot of paper.**
  - **Instead, we were able to use components that we've seen before to construct the entire circuit from a couple of easy-to-understand components.**
- **As always, we encapsulate the complete circuit in a "black box" so we can reuse it in fancier circuits.**



College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 11
Datapath Design

Unit 11: Datapath Design

Overview

- Register file
- Function Unit
  - ALU
  - Combinational Shifter
- Basic Datapaths Design
- General Datapath Representation
- Accessing RAM
- The Control Word

Chapter-8

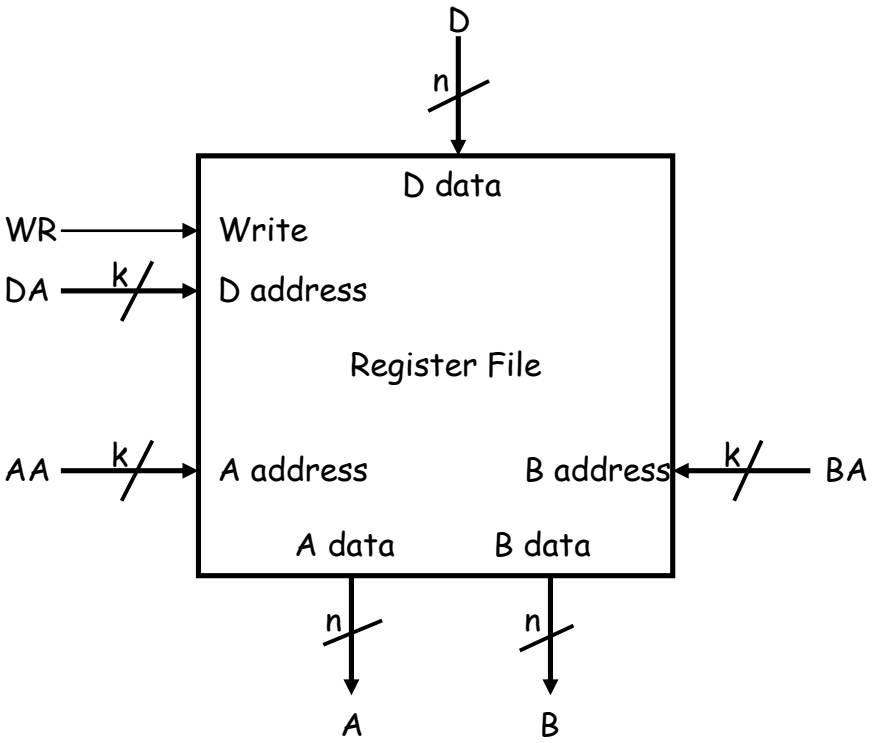
M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

Datapath Design

- A limited set of **registers** serve as fast temporary storage.
- **ALU & Shifter** perform various operations on the data stored in the registers
- A larger, but slower, **random-access memory** is also available.
- The datapath completes a single microoperation **each clock cycle**.
- For given microoperation, we need to specify;
    - Source registers
    - Destination register
    - Operation to perform

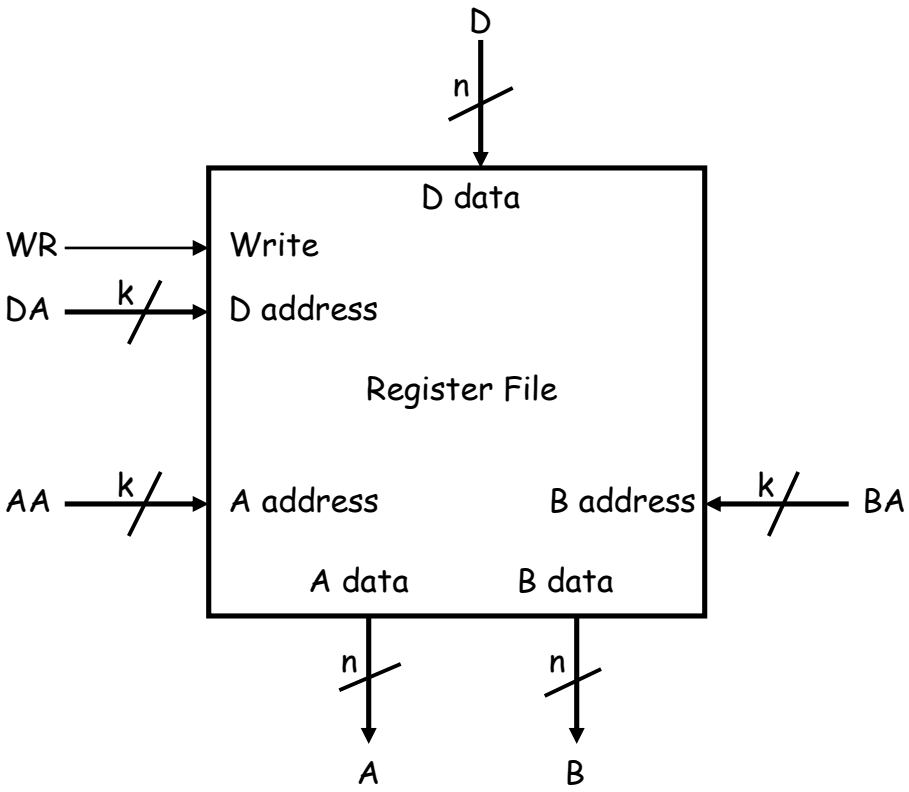
Register file

- Modern processors contain a number of registers grouped together in a register file.
- Much like words stored in a RAM, individual registers are identified by an address.
- Here is a block symbol for a 2^k x n register file.
  - There are 2^k registers, so register addresses are k bits long.
  - Each register holds an n-bit word, so the data inputs and outputs are n bits wide.



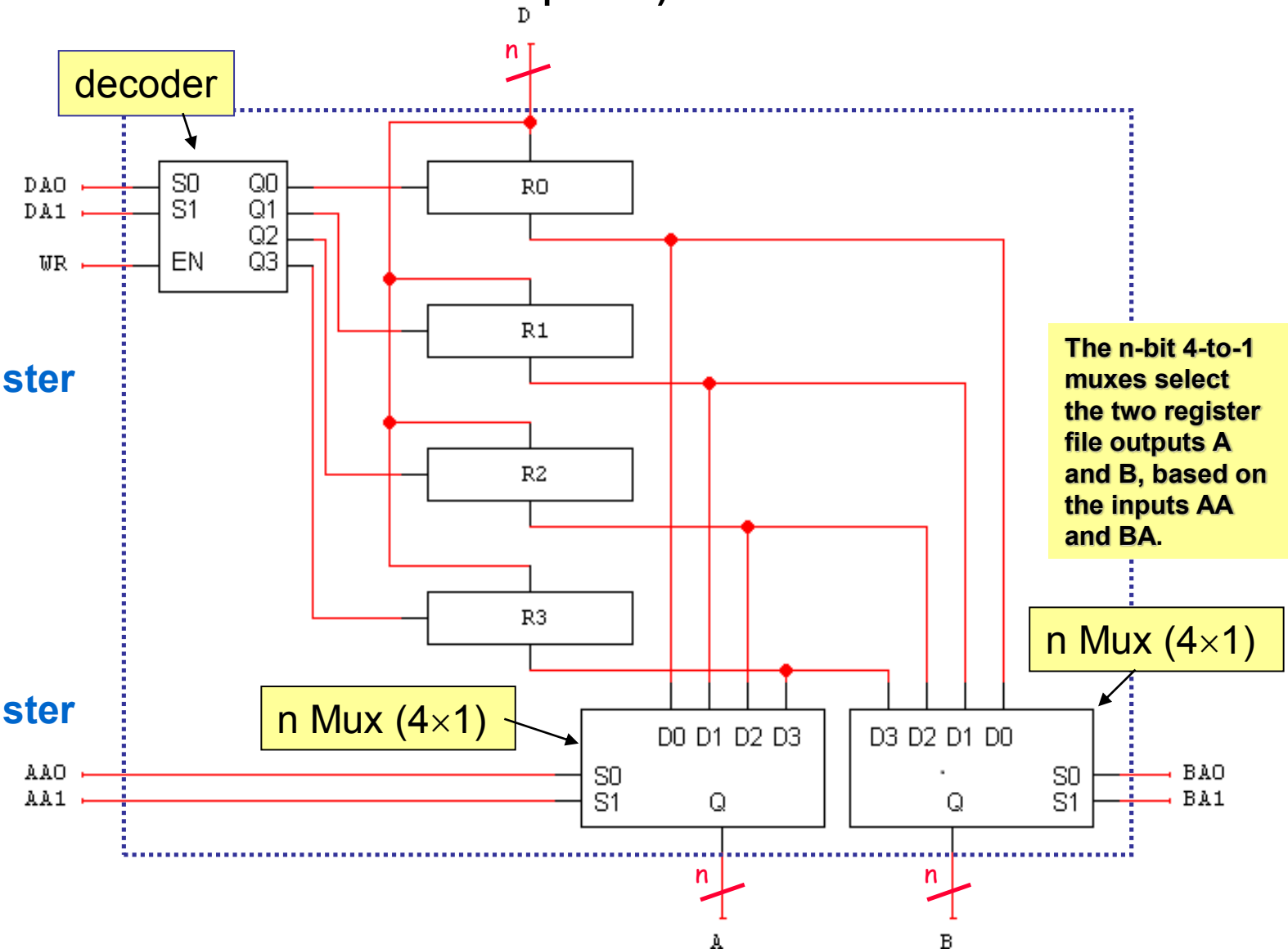
Accessing the register file

- You can read *two* registers at once by supplying the AA and BA inputs. The data appears on the A and B outputs.
- You can write to a register by using the DA and D inputs, and setting WR = 1.
- These are registers so there must be a clock signal, even though we usually don't show it in diagrams.
  - We can read from the register file at any time.
  - Data is written only on the positive edge of the clock.



What's inside the register file

- Here's a 4 x n register file. (We'll assume a 2^k x n = **4 x n** register file for all our examples.)



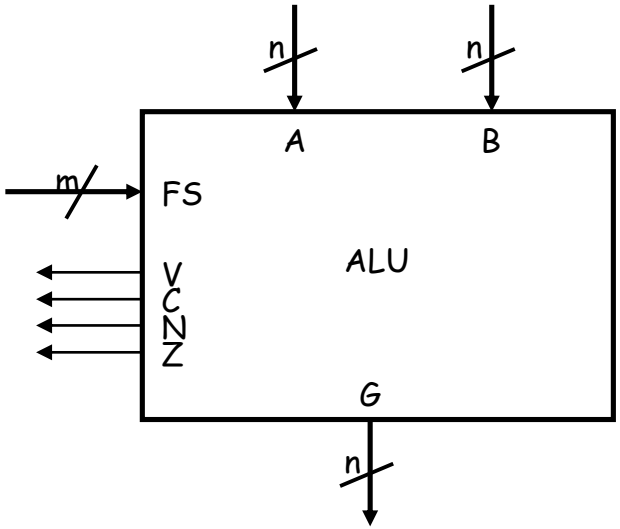
DA0 & DA1
Select the register to write in

AA0 & AA1
BA0 & BA1
Select the register to read from

The n-bit 4-to-1 muxes select the two register file outputs A and B, based on the inputs AA and BA.

The all-important ALU

- The main job of a central processing unit is to "process," or to perform computations....remember the ALU
- We'll use the following general block symbol for the ALU.
  - A and B are two n-bit numeric inputs.
  - FS is an m-bit function select code, which picks one of 2^m functions.
  - The n-bit result is called G.
  - Several status bits provide more information about the output G:
    - V = 1 in case of signed overflow.
    - C is the carry out.
    - N = 1 if the result is negative.
    - Z = 1 if the result is 0.

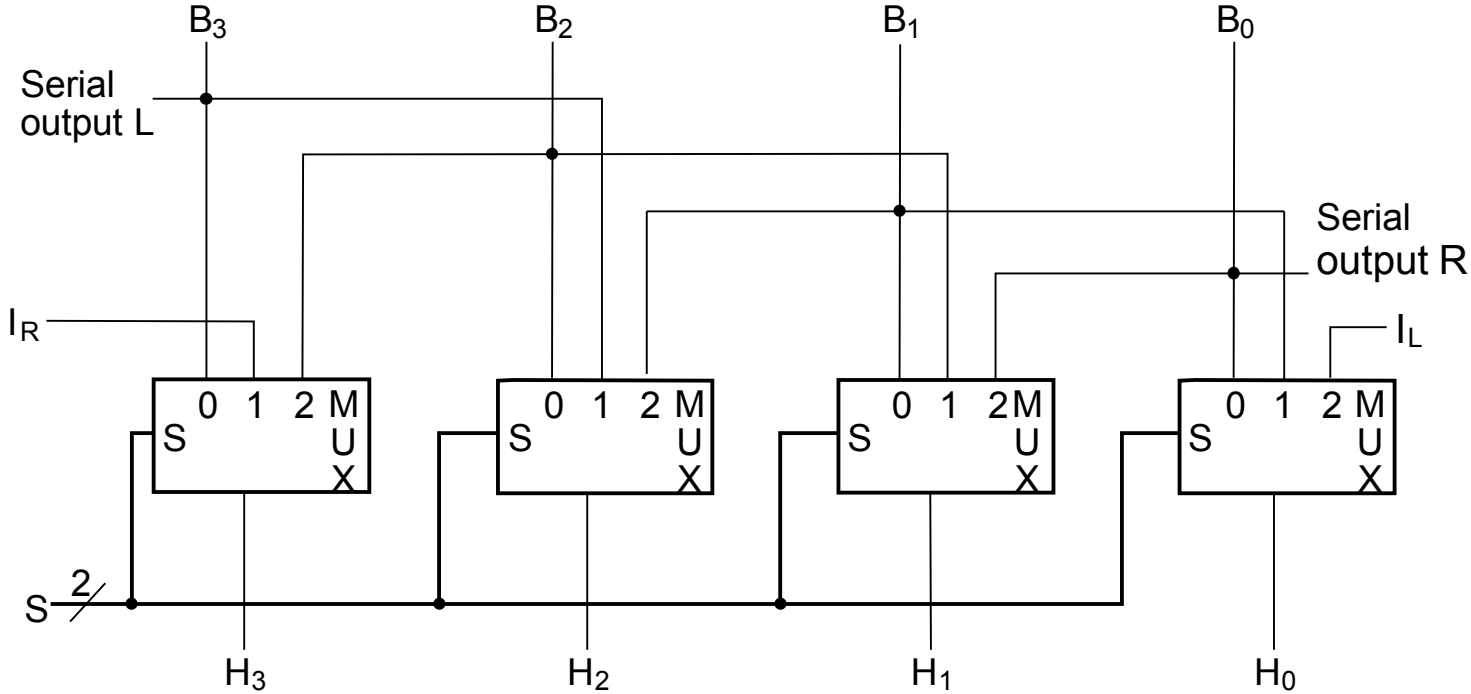


S_3	S_2	S_1	S_0	Operation
0	0	0	0	G = X
0	0	0	1	G = X + 1
0	0	1	0	G = X + Y
0	0	1	1	G = X + Y + 1
0	1	0	0	G = X + Y'
0	1	0	1	G = X + Y' + 1
0	1	1	0	G = X - 1
0	1	1	1	G = X
1	x	0	0	G = X and Y
1	x	0	1	G = X or Y
1	x	1	0	G = X ⊕ Y
1	x	1	1	G = X'

Combinational Shifter

- Bidirectional shift register with parallel load
  - Disadvantage: 3 clock pulses required
- **Combinational Shifter**
  - Transfer from a source to destination register
  - One clock cycle
- Operations: Transfer, Shift Left, Shift Right
- Filling of vacant positions
  - Here, will provide input lines or zero fill
  - Many options depending on instruction set

4-Bit Basic Left/Right Shifter



- Serial Inputs:
  - I_R for right shift
  - I_L for left shift
- Serial Outputs
  - R for right shift (Same as MSB input)
  - L for left shift (Same as LSB input)

- Shift Functions:

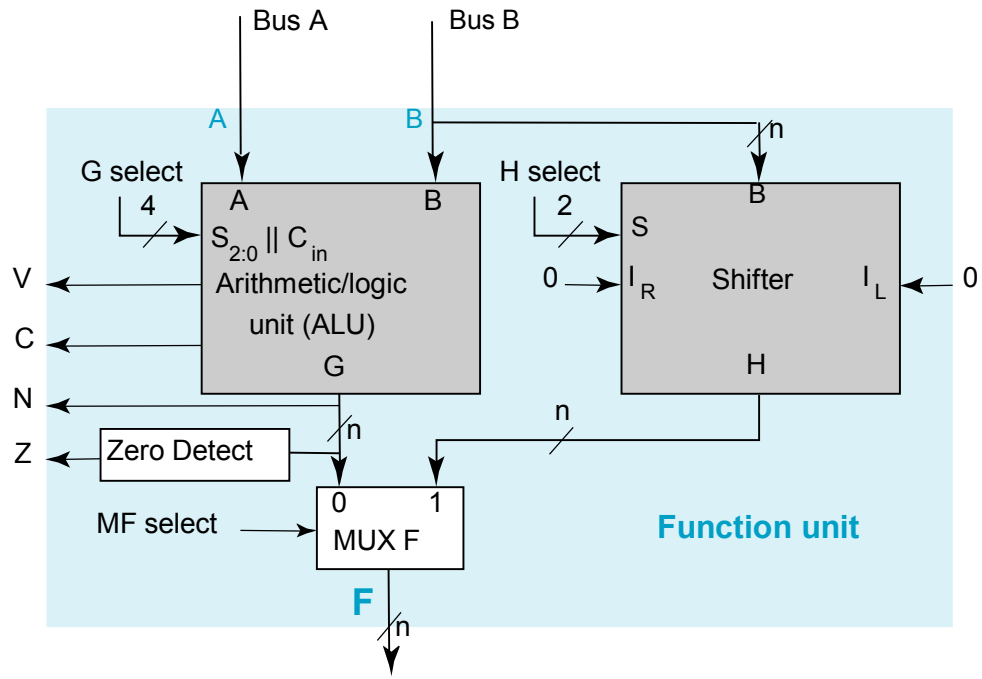
  (S1, S0) = 00 Pass B unchanged
  01 Right shift
  10 Left shift
  11 Unused

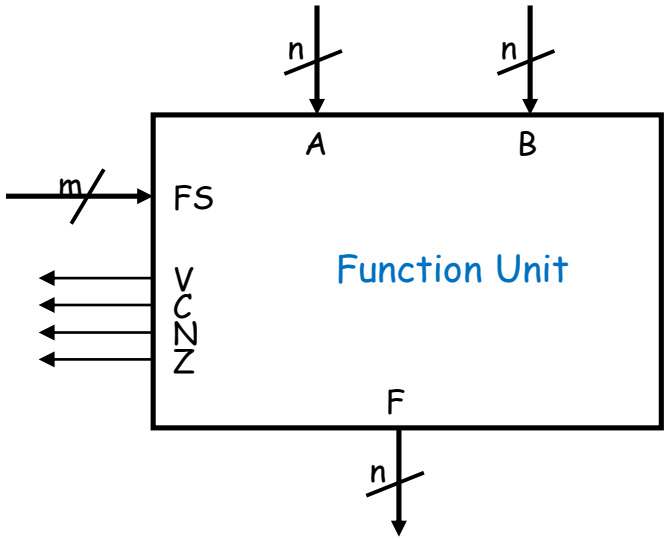
Function Unit Design

Function Unit = ALU + Shifter

- The function select code FS is 4 bits long, but there are only 15 different functions here.

FS	Operation
0000	F = A
0001	F = A + 1
0010	F = A + B
0011	F = A + B + 1
0100	F = A + B'
0101	F = A + B' + 1
0110	F = A - 1
0111	F = A
1000	F = A ∧ B (AND)
1001	F = A ∨ B (OR)
1010	F = A ⊕ B (XOR)
1011	F = A'
1100	F = B
1101	F = sr B (shift right)
1110	F = sl B (shift left)





Definition of Function Unit Select (FS) Codes

FS(3:0)	MF Select	G Select(3:0)	H Select(3:0)	Micr ooperation
0000	0	0000	XX	$F \leftarrow A$
0001	0	0001	XX	$F \leftarrow A + 1$
0010	0	0010	XX	$F \leftarrow A + B$
0011	0	0011	XX	$F \leftarrow A + B + 1$
0100	0	0100	XX	$F \leftarrow A + \overline{B}$
0101	0	0101	XX	$F \leftarrow A + \overline{B} + 1$
0110	0	0110	XX	$F \leftarrow A - 1$
0111	0	0111	XX	$F \leftarrow A$
1000	0	1 X00	XX	$F \leftarrow A \wedge B$
1001	0	1 X01	XX	$F \leftarrow A \vee B$
1010	0	1 X10	XX	$F \leftarrow A \oplus B$
1011	0	1 X11	XX	$F \leftarrow \overline{A}$
1100	1	XXXX	00	$F \leftarrow B$
1101	1	XXXX	01	$F \leftarrow \text{sr } B$
1110	1	XXXX	10	$F \leftarrow \text{sl } B$

Boolean Equations:

$MF = F_3 F_2$

$G_i = F_i$

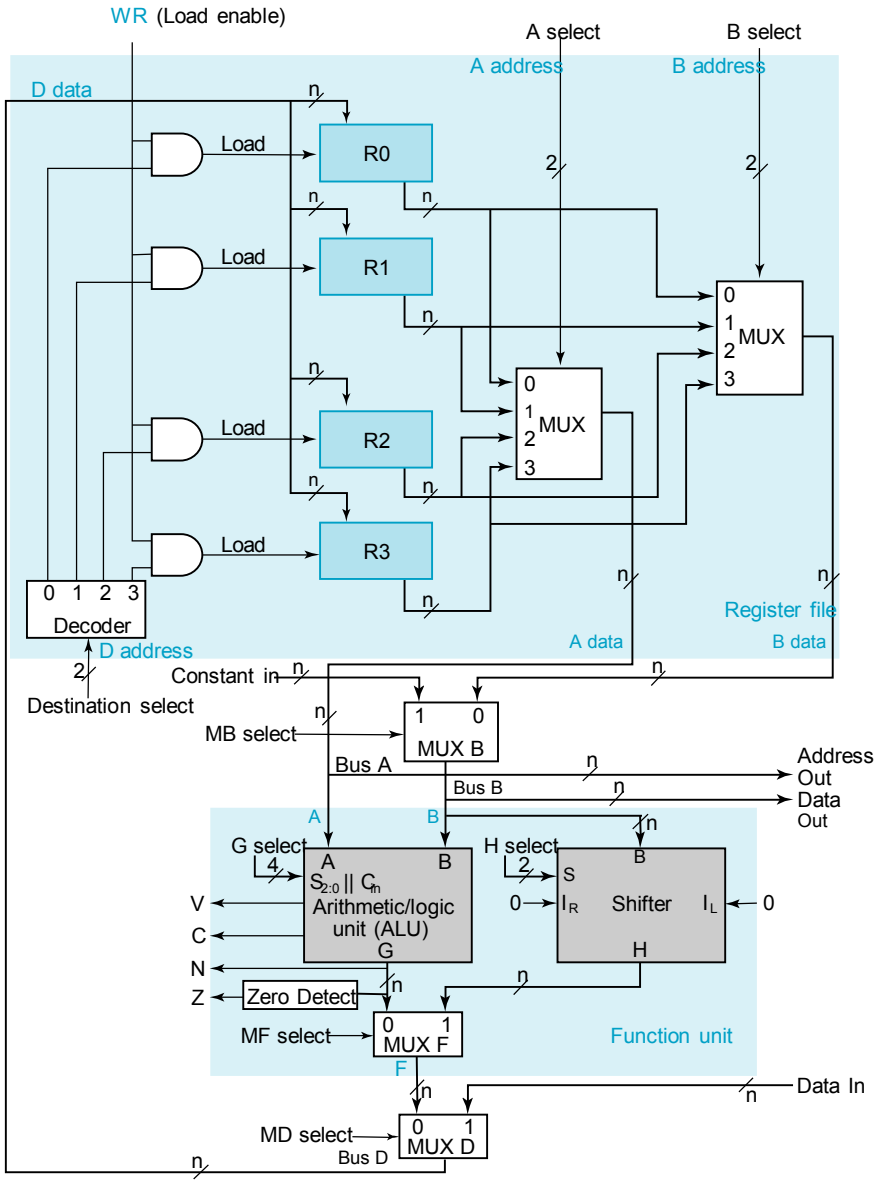
$H_i = F_i$

Basic Datapaths Design

- Guiding principles for basic datapaths:
  - The set of registers
    - A set of registers with common access resources called a *register file*
  - Microoperation implementation
    - One or more shared resources for implementing microoperations
    - **Buses** - shared transfer paths
    - ***Arithmetic-Logic Unit (ALU)*** - shared resource for implementing arithmetic and logic microoperations
    - **Shifter** - shared resource for implementing shift microoperations

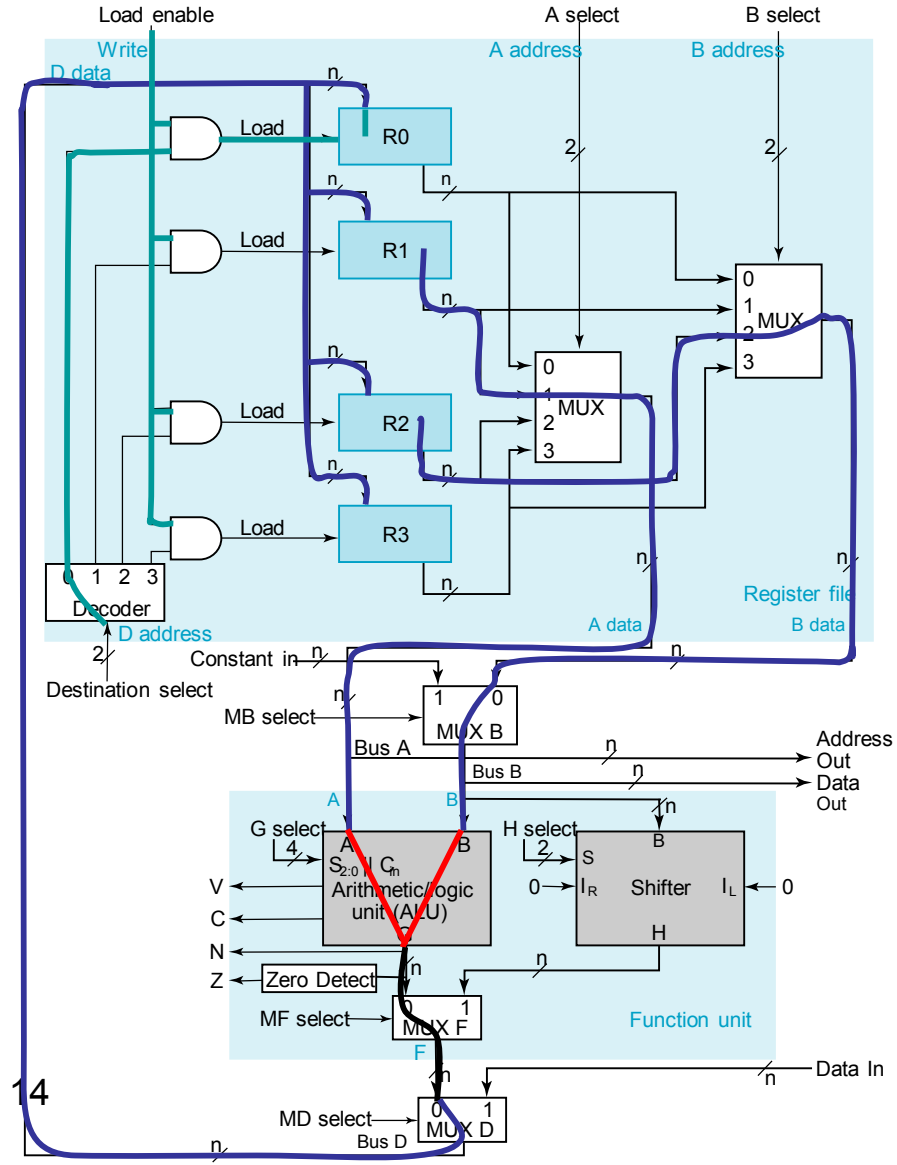
Datapath Example

- Four parallel-load registers (R0-R3)
- Two mux-based register selectors
- Register destination decoder
- Mux B for external constant input
- Buses A and B with external address and data outputs
- ALU and Shifter with Mux F for output select
- Mux D for external data input
- Logic for generating status bits V, C, N, Z



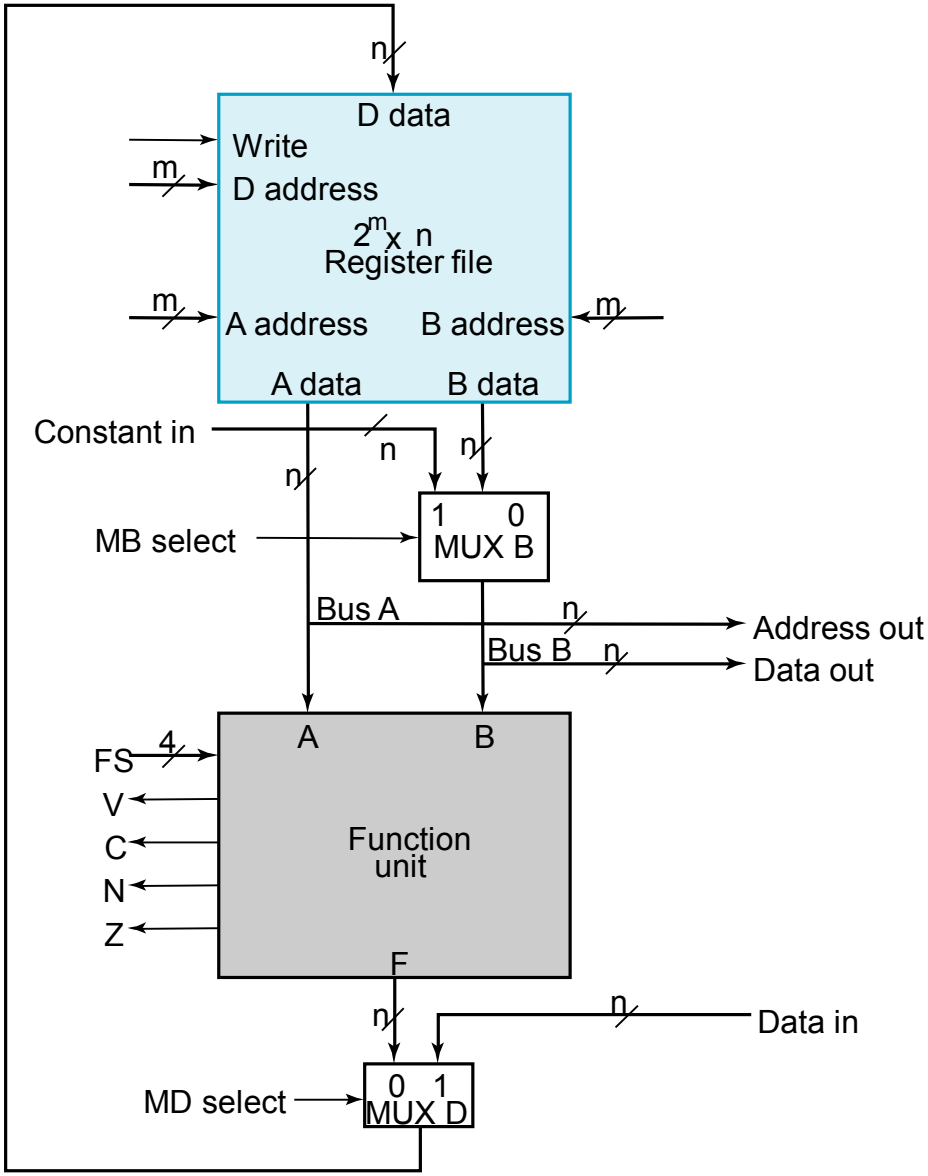
Datapath Example: Performing a Microoperation

- Microoperation: R0 ← R1 + R2
- Apply 01 to A select to place contents of R1 onto Bus A
- Apply 10 to B select to place contents of R2 onto B data and apply 0 to MB select to place B data on Bus B
- Apply 0010 to G select to perform addition G = Bus A + Bus B
- Apply 0 to MF select and 0 to MD select to place the value of G onto BUS D
- Apply 00 to Destination select to enable the Load input to R0
- Apply 1 to Load Enable to force the Load input to R0 to 1 so that R0 is loaded on the clock pulse (not shown)
- The overall microoperation requires 1 clock cycle



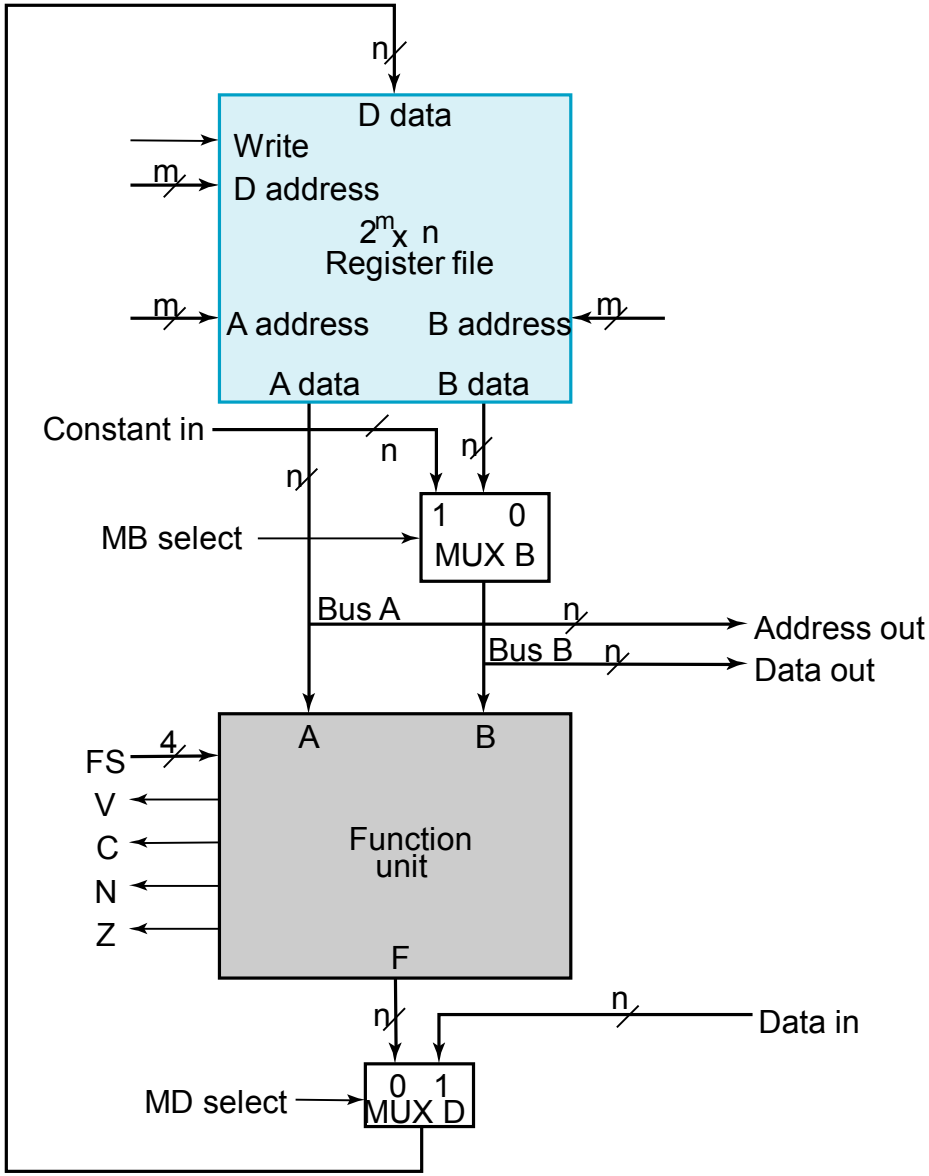
General Datapath Representation

- Here we move up one level in the hierarchy from that datapath
- The registers, and the multiplexer, decoder, and enable hardware for accessing them become a *register file*
- A register file is an array of fast registers
- The ALU, shifter, Mux F and status hardware become a *function unit*
- The remaining muxes and buses which handle data transfers are at the new level of the hierarchy



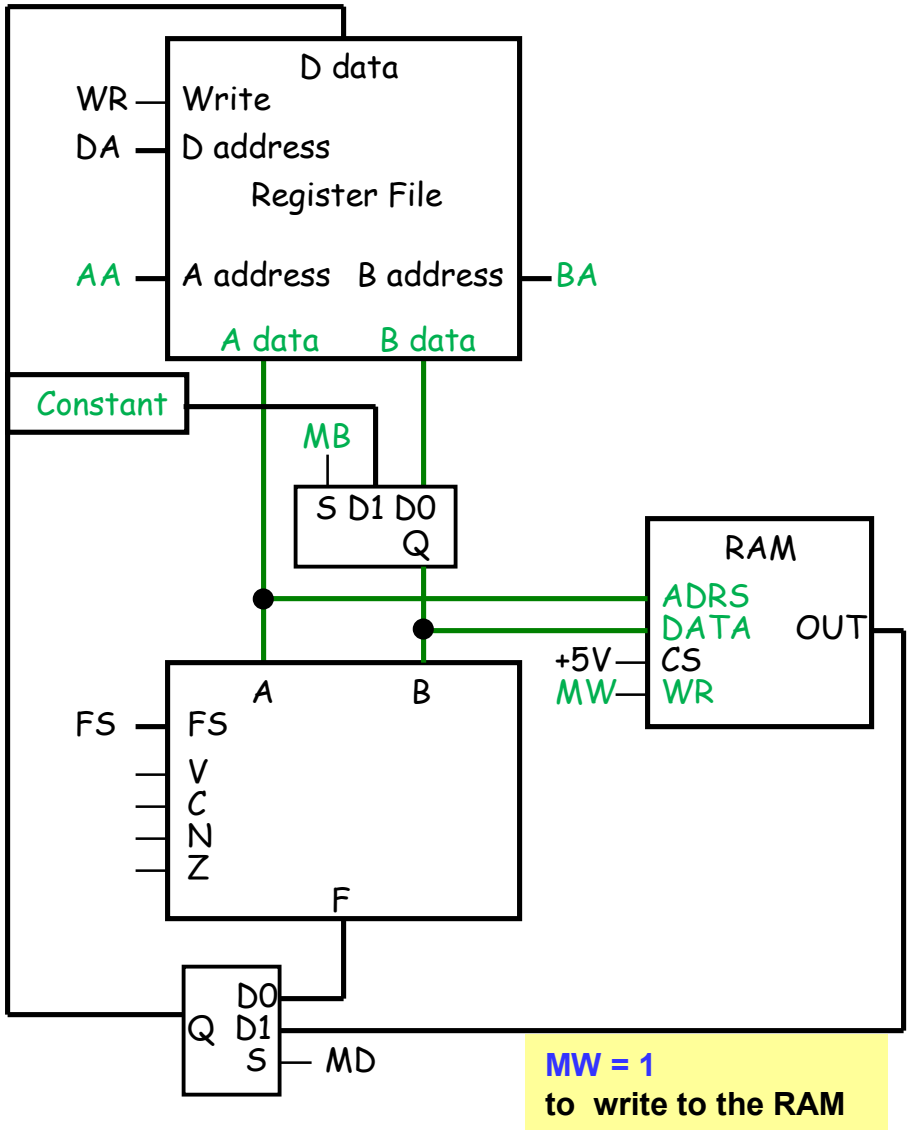
Datapath Representation (continued)

- In the register file:
  - Multiplexer select inputs become A address and B address
  - Decoder input becomes D address
  - Multiplexer outputs become A data and B data
  - Input data to the registers becomes D data
  - Load enable becomes write
- The register file now appears like a memory based on clocked flip-flops (the clock is not shown)
- The function unit labeling is quite straightforward except for FS



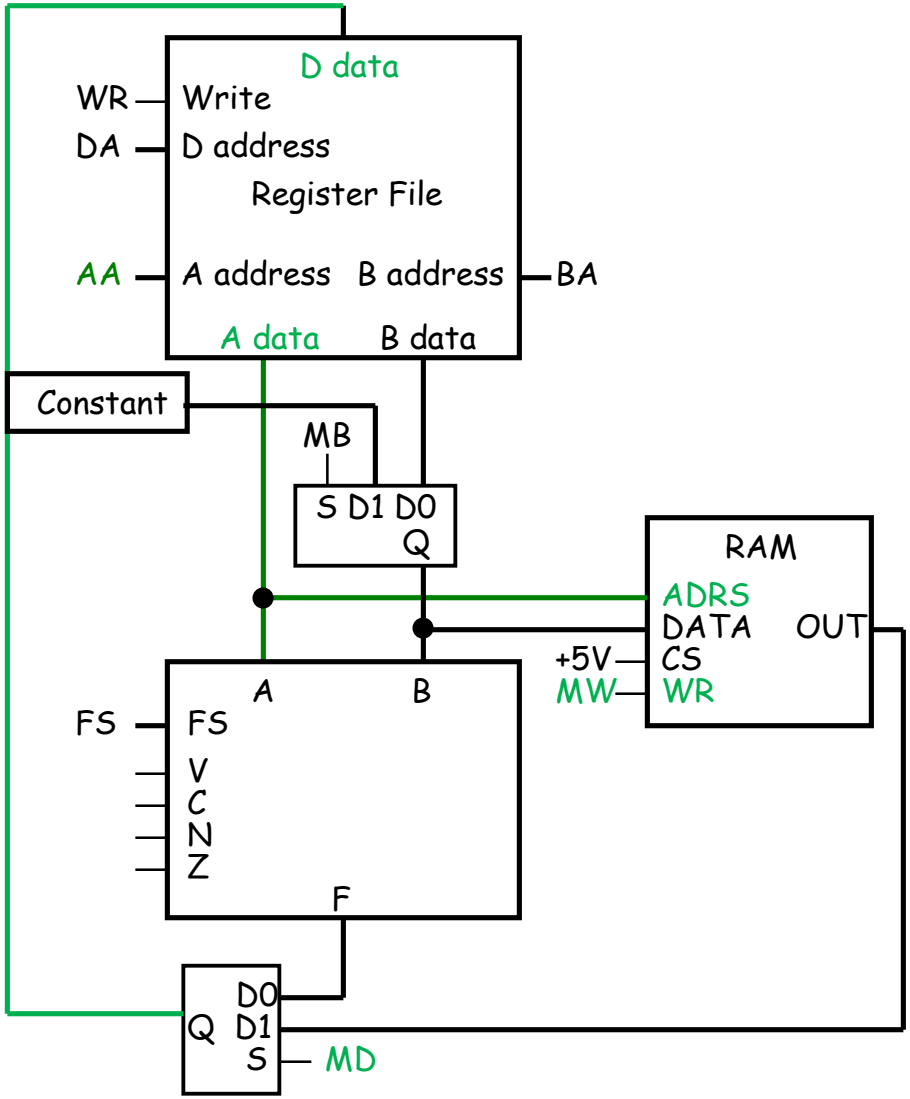
Accessing RAM

- Here's a way to connect RAM into our existing datapath.
- ***Write* to RAM:** we must give an address and a data value.
- These will come from the registers/Constant. We connect A data to the memory's ADRS input, and B data to the memory's DATA input.
- Set MW = 1 to write to the RAM. (It's called MW to distinguish it from the WR write signal on the register file.)



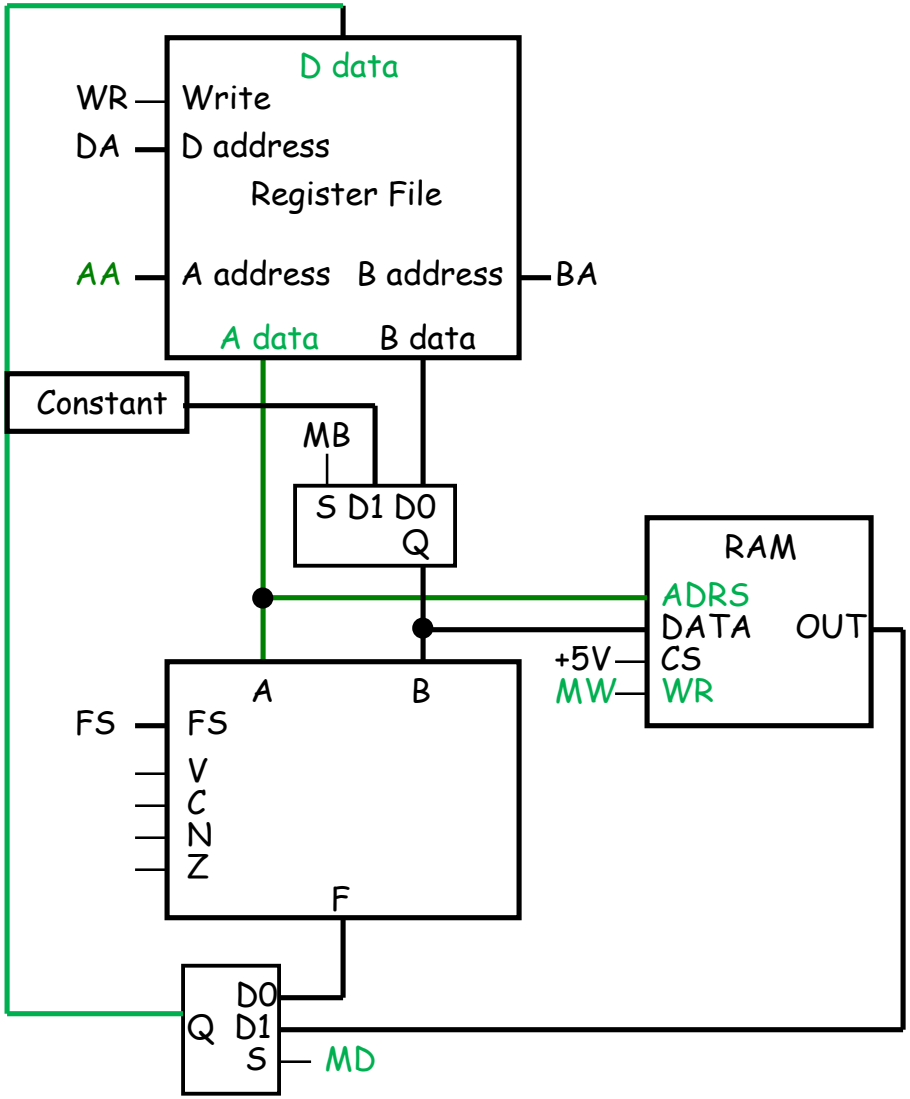
Reading from RAM

- To *read* from RAM, A data must supply the address.
- Set MW = 0 for reading.
- The incoming data will be sent to the register file for storage.
- This means that the register file's D data input could come from *either* the ALU output or the RAM.
- A mux MD selects the source for the register file.
  - When MD = 0, the ALU output can be stored in the register file.
  - When MD = 1, the RAM output is sent to the register file instead.



Notes about this setup

- We now have a way to copy data between our register file and the RAM.
- <u>Notice that there's no way for the Function Unit to directly access the memory—RAM contents *must* go through the register file first.</u>
- Here the size of the memory is limited by the size of the registers; with n-bit registers, we can only use a 2^n x n RAM.



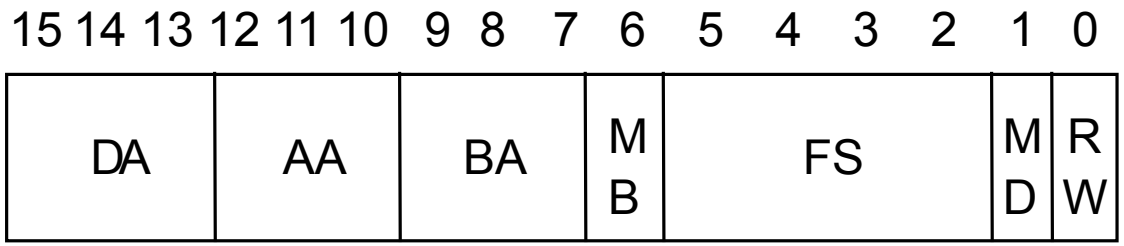
Datapath (Summary)

- The datapath completes a single microoperation each clock cycle.
- For given microoperation, we need to specify;
  - Source registers
  - Destination register
  - Operation to perform
- For the datapath, we also need to specify;
  - External or function data
  - Data B from register or constant
  - Whether the destination is updated

The Control Word

- The datapath has many control inputs
- The signals driving these inputs can be defined and organized into a *control word*
- To execute a microinstruction, we apply control word values for a clock cycle. For most microoperations, the positive edge of the clock cycle is needed to perform the register load
- The datapath control word format and the field definitions are shown on the next slide

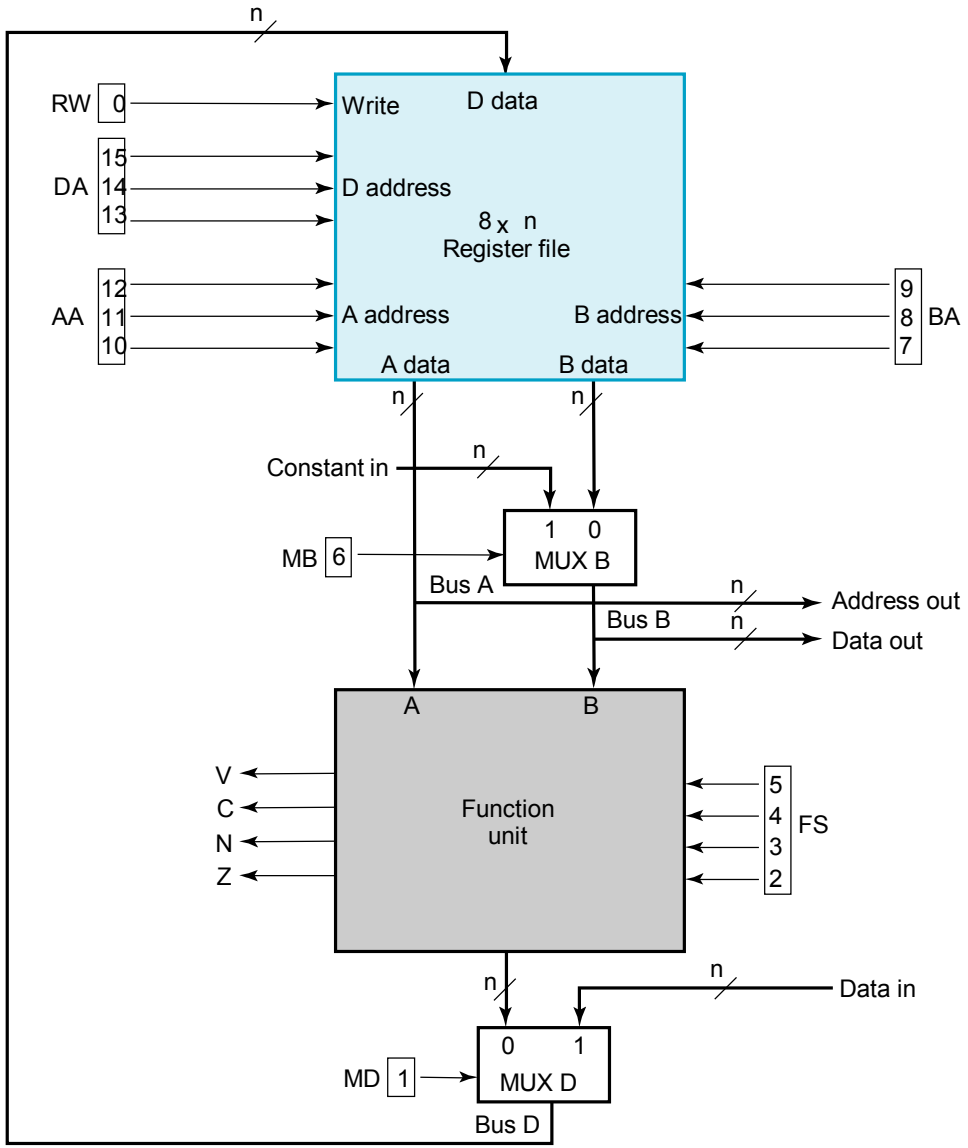
The Control Word Fields



Control word

- Fields
  - DA – D Address (destination)
  - AA – A Address
  - BA – B Address (source for MUXB
  - MB – Mux B (constant/source
  - FS – Function Select
  - MD – Mux D
  - RW – Register Write
- The connections to datapath are shown in the next slide

Control Word Block Diagram



Encoding The Control Word

TABLE 8-5
Encoding of Control Word for the Datapath

DA, AA, BA		MB		FS		MD		RW	
Function	Code	Function	Code	Function	Code	Function	Code	Function	Code
$R0$	000	Register	0	$F = A$	0000	Function	0	No Write	0
$R1$	001	Constant	1	$F = A + 1$	0001	Data in	1	Write	1
$R2$	010			$F = A + B$	0010				
$R3$	011			$F = A + B + 1$	0011				
$R4$	100			$F = A + \overline{B}$	0100				
$R5$	101			$F = A + \overline{B} + 1$	0101				
$R6$	110			$F = A - 1$	0110				
$R7$	111			$F = A$	0111				
				$F = A \wedge B$	1000				
				$F = A \vee B$	1001				
				$F = A \oplus B$	1010				
				$F = \overline{A}$	1011				
				$F = B$	1100				
				$F = \text{sr } B$	1101				
				$F = \text{sl } B$	1110				

Microoperations for the Datapath - Symbolic Representation

TABLE 8-6
Examples of Microoperations for the Datapath, Using Symbolic Notation

Micro-operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	$R1$	$R2$	$R3$	Register	$F = A + \overline{B} + I$	Function	Write
$R4 \leftarrow \text{sl } R6$	$R4$	—	$R6$	Register	$F = \text{sl } B$	Function	Write
$R7 \leftarrow R7 + 1$	$R7$	$R7$	—	—	$F = A + 1$	Function	Write
$R1 \leftarrow R0 + 2$	$R1$	$R0$	—	Constant	$F = A + B$	Function	Write
Data out $\leftarrow R3$	—	—	$R3$	Register	—	—	No Write
$R4 \leftarrow$ Data in	$R4$	—	—	—	—	Data in	Write
$R5 \leftarrow 0$	$R5$	$R0$	$R0$	Register	$F = A \oplus B$	Function	Write

Microoperations for the Datapath - Binary Representation

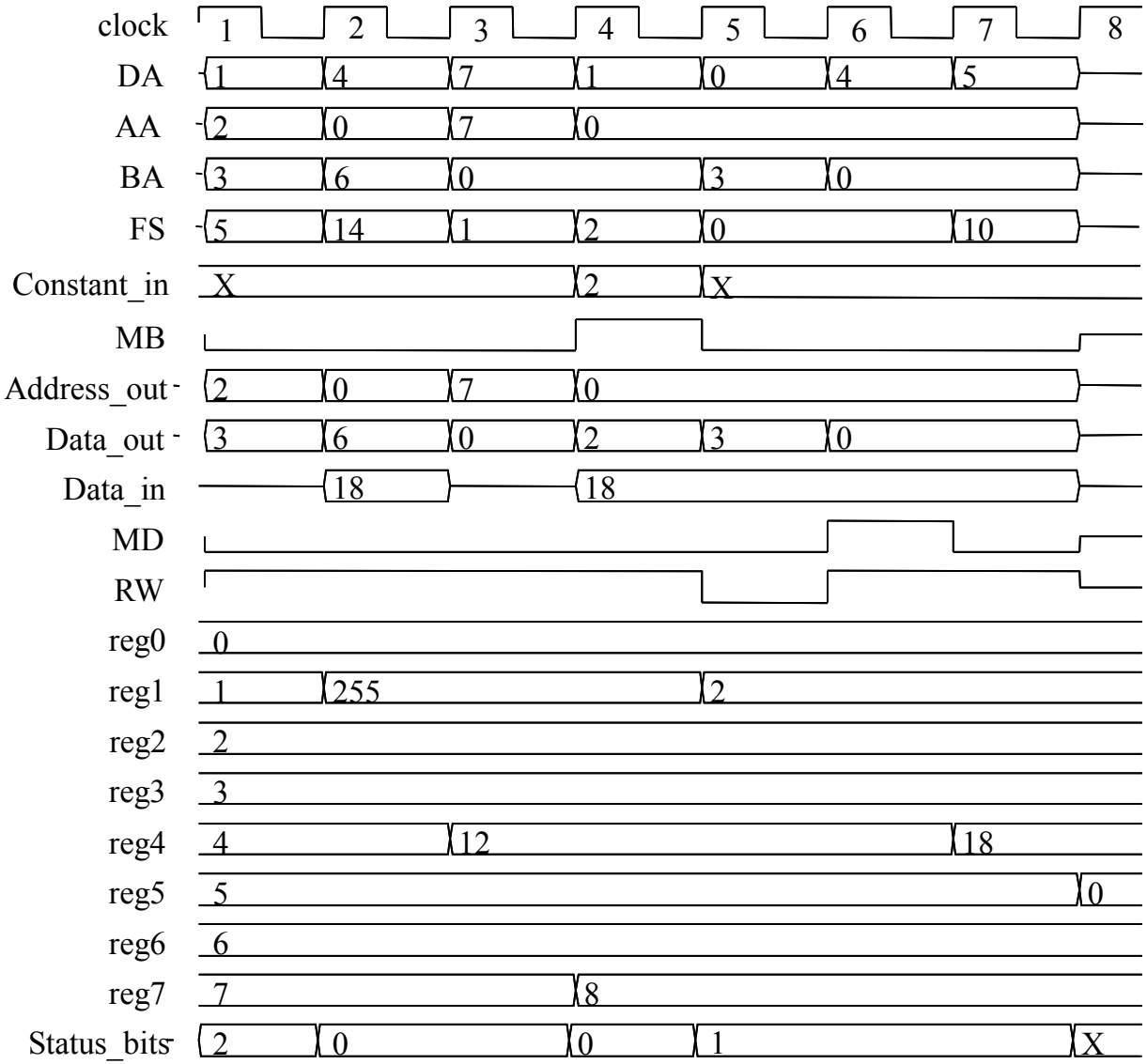
TABLE 8-7
Examples of Microoperations from Table 8-6, Using Binary Control Words

Micro-operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	001	010	011	0	0101	0	1
$R4 \leftarrow$ sl R6	100	XXX	110	0	1110	0	1
$R7 \leftarrow R7 + 1$	111	111	XXX	X	0001	0	1
$R1 \leftarrow R0 + 2$	001	000	XXX	1	0010	0	1
Data out $\leftarrow R3$	XXX	XXX	011	0	XXXX	X	0
$R4 \leftarrow$ Data in	100	XXX	XXX	X	XXXX	1	1
$R5 \leftarrow 0$	101	000	000	0	1010	0	1

Results of simulation of the above on the next slide

Datapath Simulation

Results of simulation of the above



Stored Program Computer

- Control words operate the datapath
- Store control words in memory
  - Sequence through them to perform a series of computational steps
- This is what a computer does

College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 12
CPU Design & Programming

Unit 12: CPU Design & Programming

Overview

- Simple Computer Architecture
  - Instruction Set Architecture (ISA)
  - Storage Resources
  - Instruction Formats
  - Instruction Specifications
- Single-Cycle Hardwired Control Unit
- Assembly and machine languages

Chapter-8

M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

An overview of CPU design

- We can divide the design of our CPU into three parts:
  - The datapath does all of the actual data processing (Unit-11).
  - An instruction set is the programmer's interface to CPU.
  - A control unit uses the programmer's instructions to tell the datapath what to do.

Simple Computer Architecture

- A computer is controlled by a series of ***instructions***. Instructions are binary words that are used to determine both the datapath processing and the control behavior.
- A ***program*** is a collection of instructions that are used to accomplish some task(s).
- A ***program counter*** (PC) is used to keep track of the address of the next instruction in memory. The PC has counting logic, as well as parallel load and other logic to permit changes in the instruction sequence.

Simple Computer Architecture

- Changes in the instruction sequence can be either *conditional* or *unconditional*.
- Instructions are *executed* by activating the necessary microoperations to perform the specified task.
- Note that the computer uses programmable control, but we will use a hardwired control unit to actually implement the programmable control.

Instruction Set Architecture (ISA)

- The instruction set architecture (ISA): a comprehensive description of the instructions a computer can execute.
- The ISA has three major components
  - Storage Resources
  - Instruction Formats
  - Instruction Specification

Storage Resources

- resources available for storing information
  - Register file
  - Program counter (PC)
  - Instruction memory (program memory)
  - Data memory

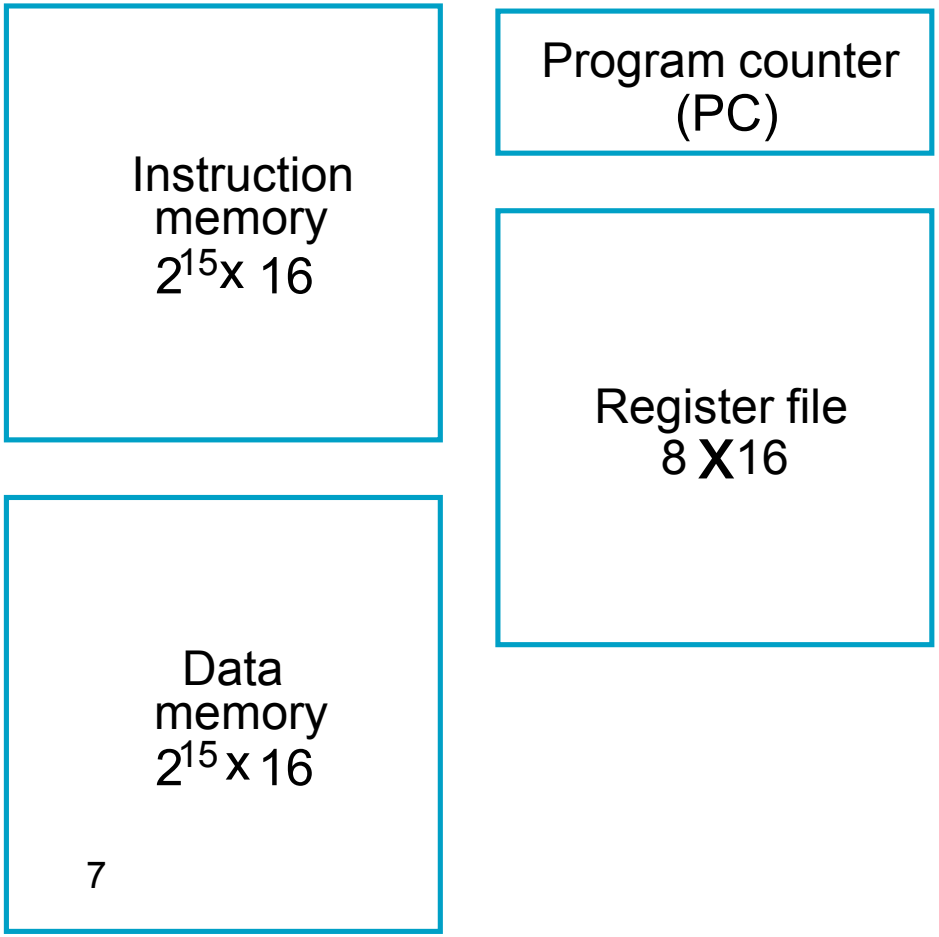


FIGURE 8-13 Storage Resource Diagram for a Simple Computer

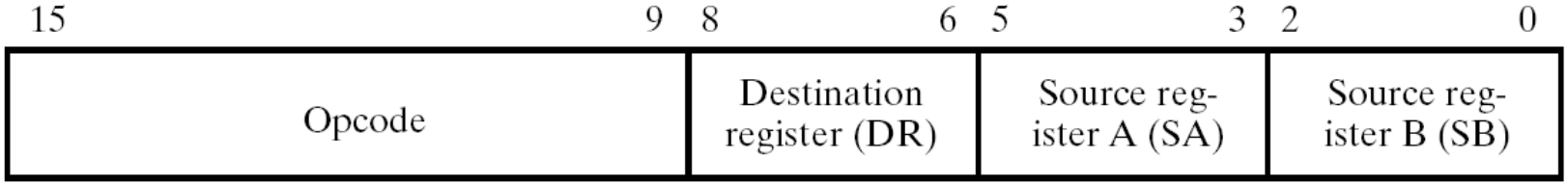
Instruction Formats

- Instructions are divided into bit groupings called *fields*. Each field will contain a specific part of the instruction.
  - Operation code (opcode)
    - The opcode field <u>determines what the actual operation will be.</u>
    - An <u>n-bit opcode</u> can specify <u>up to 2^n different operations</u>.
  - Operand(s)
    - Registers
    - Addresses of Operands
    - Constant data
  - Operands may be specified implicitly as well.

Instruction Formats -1

Register: The source(s) and destination of an instruction can all be registers.

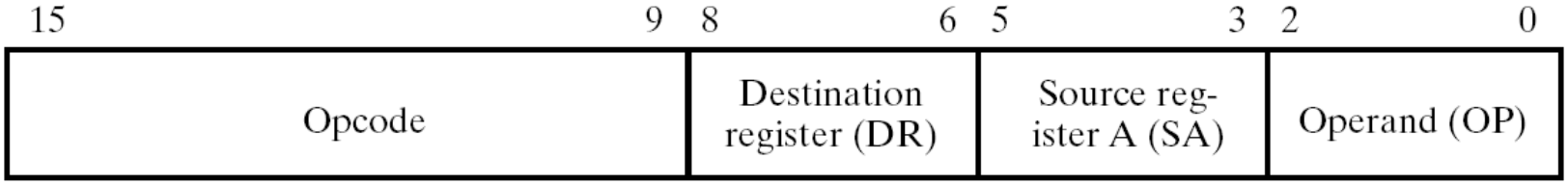
- The source registers are read and the new information is written to the destination register.
- All registers are explicitly identified (usually).



Instruction Formats - 2

Immediate operands: data constants are contained within the instruction.

- Note that only three bits of data can be specified here, so
- It must be extended to the register length by zero-fill or sign-extension.
- Clever encodings enable wider immediates
  - RR format, RI format, B format



Instruction Formats - 3

Jump and Branch: Changes in program flow are caused by jump or branch instructions, affecting only the PC.

1. Can load the PC from source SA.
2. Can add the sign-extended 6-bit offset (AD) to the PC.
3. Can be either *unconditional*, or *conditional* based on some flag value (i.e. Z, N, C, V).

15 … 9	8 … 6	5 … 3	2 … 0
Opcode	Address (AD) (Left)	Source register A (SA)	Address (AD) (Right)

Instruction Specifications

- The *instruction specifications* describe in detail each instruction the system can execute.
- A ***mnemonic*** is written by the programmer to represent the opcode in text.
- ***assembler*** generates the actual binary instruction.
- Not every instruction sets every flag
  - Refer to Table 8-8

ISA: Instruction Specifications

- **The specifications provide**:
  - The name of the instruction
  - The instruction's opcode
  - A shorthand name for the opcode called a *mnemonic*
  - A specification for the instruction format
  - A register transfer description of the instruction, and
  - A listing of the status bits that are meaningful during an instruction's execution (not used in the architectures defined in this chapter)

ISA: Instruction Specifications

TABLE 8-8
Instruction Specifications for the Simple Computer

Instruction	Opcode	Mne-monic	Format	Description	Status Bits
Move A	0000000	MOVA	RD, RA	$R[DR] \leftarrow R[SA]^*$	N, Z
Increment	0000001	INC	RD, RA	$R[DR] \leftarrow R[SA] + 1^*$	N, Z
Add	0000010	ADD	RD, RA, RB	$R[DR] \leftarrow R[SA] + R[SB]^*$	N, Z
Subtract	0000101	SUB	RD, RA, RB	$R[DR] \leftarrow R[SA] - R[SB]^*$	N, Z
Decrement	0000110	DEC	RD, RA	$R[DR] \leftarrow R[SA] - 1^*$	N, Z
AND	0001000	AND	RD, RA, RB	$R[DR] \leftarrow R[SA] \wedge R[SB]^*$	N, Z
OR	0001001	OR	RD, RA, RB	$R[DR] \leftarrow R[SA] \vee R[SB]^*$	N, Z
Exclusive OR	0001010	XOR	RD, RA, RB	$R[DR] \leftarrow R[SA] \oplus R[SB]^*$	N, Z
NOT	0001011	NOT	RD, RA	$R[DR] \leftarrow \overline{R[SA]}^*$	N, Z
Move B	0001100	MOVB	RD, RB	$R[DR] \leftarrow R[SB]^*$	
Shift Right	0001101	SHR	RD, RB	$R[DR] \leftarrow sr\ R[SB]^*$	
Shift Left	0001110	SHL	RD, RB	$R[DR] \leftarrow sl\ R[SB]^*$	

- Here we have arithmetic, logic and shift **instruction sets** correspond to **micro-operations** discussed in Unit 8-11. The **opcode** (7 bit) and **FS** (4 bit) have one-to-one correspondence – first three bits of an opcode are 000.
- RTL instruction (in description) is written in a slightly different way. For example, R[DR] <- R[SA] is a RTL instruction where DR and SA are destination and source register addresses respectively.

ISA: Instruction Specifications (continued)

TABLE 8-8
Instruction Specifications for the Simple Computer

Instruction	Opcode	Mne-monic	Format	Description	Status Bits
Load Immediate	1001100	LDI	RD, OP	R[DR] ← zf OP*	
Add Immediate	1000010	ADI	RD, RA, OP	R[DR] ← R[SA] + zf OP*	N, Z
Load	0010000	LD	RD, RA	R[DR] ← M[SA]*	
Store	0100000	ST	RA, RB	M[SA] ← R[SB]*	
Branch on Zero	1100000	BRZ	RA, AD	if (R[SA] = 0) PC ← PC + se AD, if (R[SA] ≠ 0) PC ← PC + 1	N, Z
Branch on Negative	1100001	BRN	RA, AD	if (R[SA] < 0) PC ← PC + se AD, if (R[SA] ≥ 0) PC ← PC + 1	N, Z
Jump	1110000	JMP	RA	PC ← R[SA]*	

* For all of these instructions, PC ← PC + 1 is also executed to prepare for the next cycle.

- These are additional instructions having no corresponding **micro-operations –** for them first three bits have **non-zero values.**
- For instruction LDI, RTL is R[DR] <- zf OP where **zf** means zero filled and OP is a **3 bit constant** (that is used for the simple computer).
- In description of BRZ and BRN, **se** mean sign extended. This is done to make it 16 bit because the RAM has $2^{16} \times 16$ configuration (refer to slide 8, figure 8-13).

ISA: Memory Representation

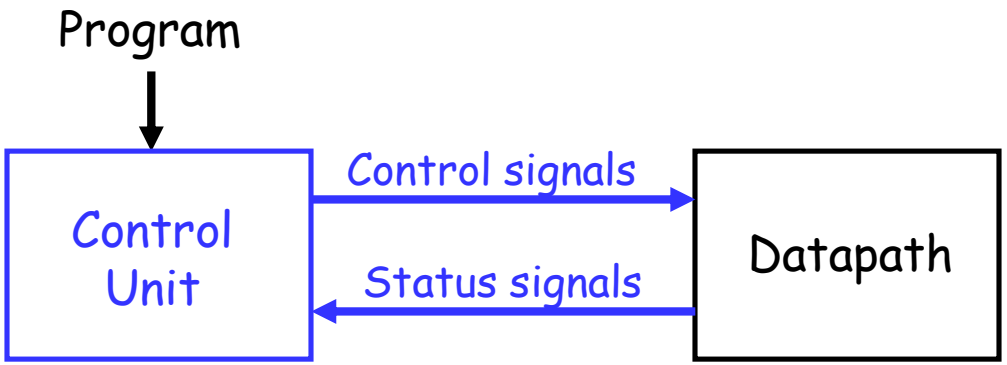
- Table 8-9 shows Instructions and data, in binary, are placed in memory. Here, we assume that we have a program (in machine language) in the RAM and we show instructions at locations 25, 35, 45, and 55 (as examples).
- Corresponding assembly codes for four instructions are as follows.

25: SUB R1, R2, R3
35: ST (R4), R5 [suppose R4=70, R5=80]
45: ADI R2, R7, 3
55: BRZ R6, AD [suppose AD(left)=101, AD(right)=100, se AD = 1111111111101100 = -20]

TABLE 8-9
Memory Representation of Instructions and Data

Decimal Address	Memory Contents (Opcode DR SA SB)	Decimal Opcode	Other Fields	Operation
25	0000101 001 010 011	5 (Subtract)	DR:1, SA:2, SB:3	R1 ← R2 – R3
35	0100000 000 100 101	32 (Store)	SA:4, SB:5	M[R4] ← R5
45	1000010 010 111 011	66 (Add Immediate)	DR:2 SA:7 OP:3 (Immediate operand)	R2 ← R7 + 3
55	1100000 101 110 100 (Jump and Branch; 101100)	96 (Branch on Zero)	AD: 44, SA:6	If R6 = 0, PC ← PC – 20
70	0000000001100000 (before) 0000000001010000 (After)	Data = 192. After execution of instruction in 35, Data = 80.		

Block diagram of a processor



- The control unit connects programs with the datapath.
  - In real computers, the datapath actions are determined by the program that's loaded and running
  - It converts program instructions into control words for the datapath, including **signals WR, DA, AA, BA, MB, FS, MW, MD**.
  - It executes program instructions in the correct sequence.
  - It generates the "constant" input for the datapath.
- The datapath also sends information back to the control unit. For instance, the ALU status bits **V, C, N, Z** can be inspected by branch instructions to alter a program's control flow.

Single-Cycle Hardwired Control

- In single-cycle hardwired control, the PC is updated on each clock cycle. Each instruction is completed in a single cycle.
- The PC is used to select a word from the instruction memory, which is driven to the instruction decoder.
- The instruction decoder then provides the control word to the datapath to activate the desired functionality, and determines how the PC is updated.

The instruction decoder maps the instruction word to a control word.

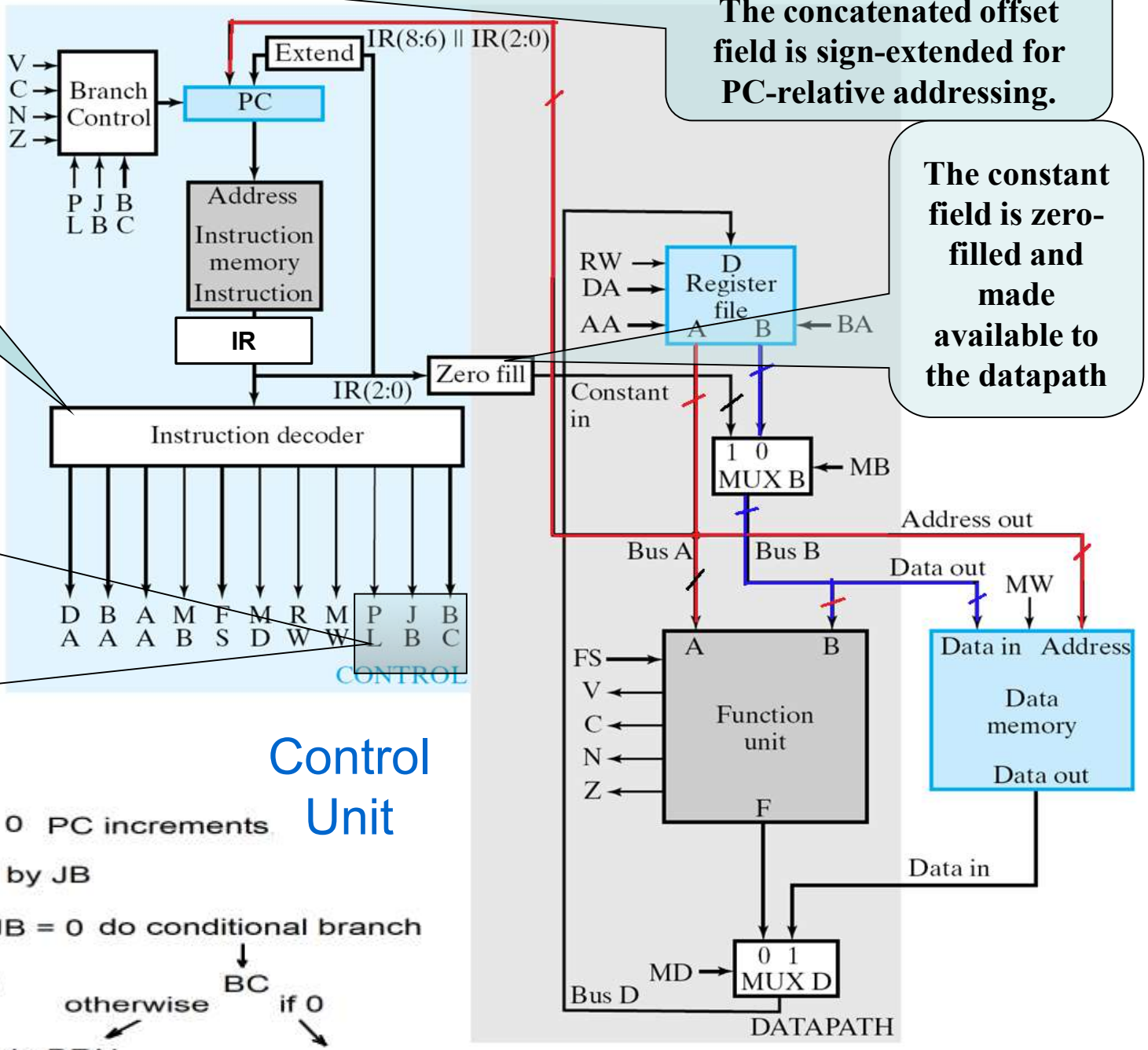
The concatenated offset field is sign-extended for PC-relative addressing.

The constant field is zero-filled and made available to the datapath

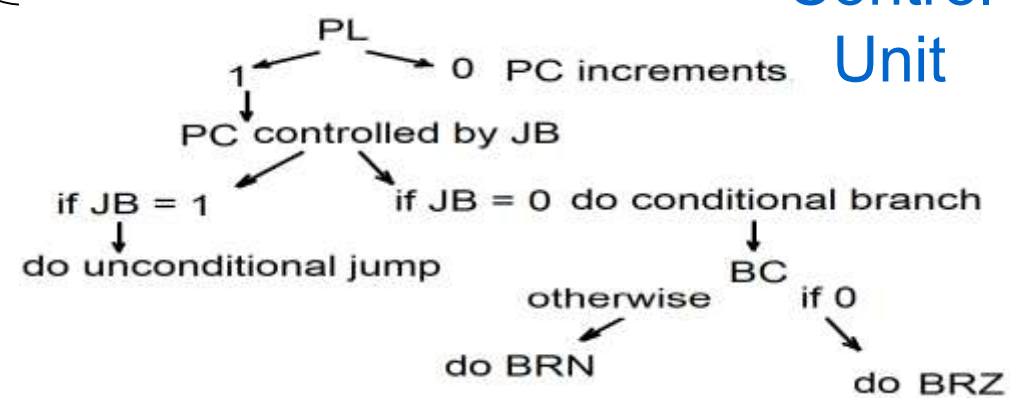
PL – if 0, the PC increments, otherwise the update is controlled by JB

JB – if 0, do conditional branch, otherwise do unconditional jump

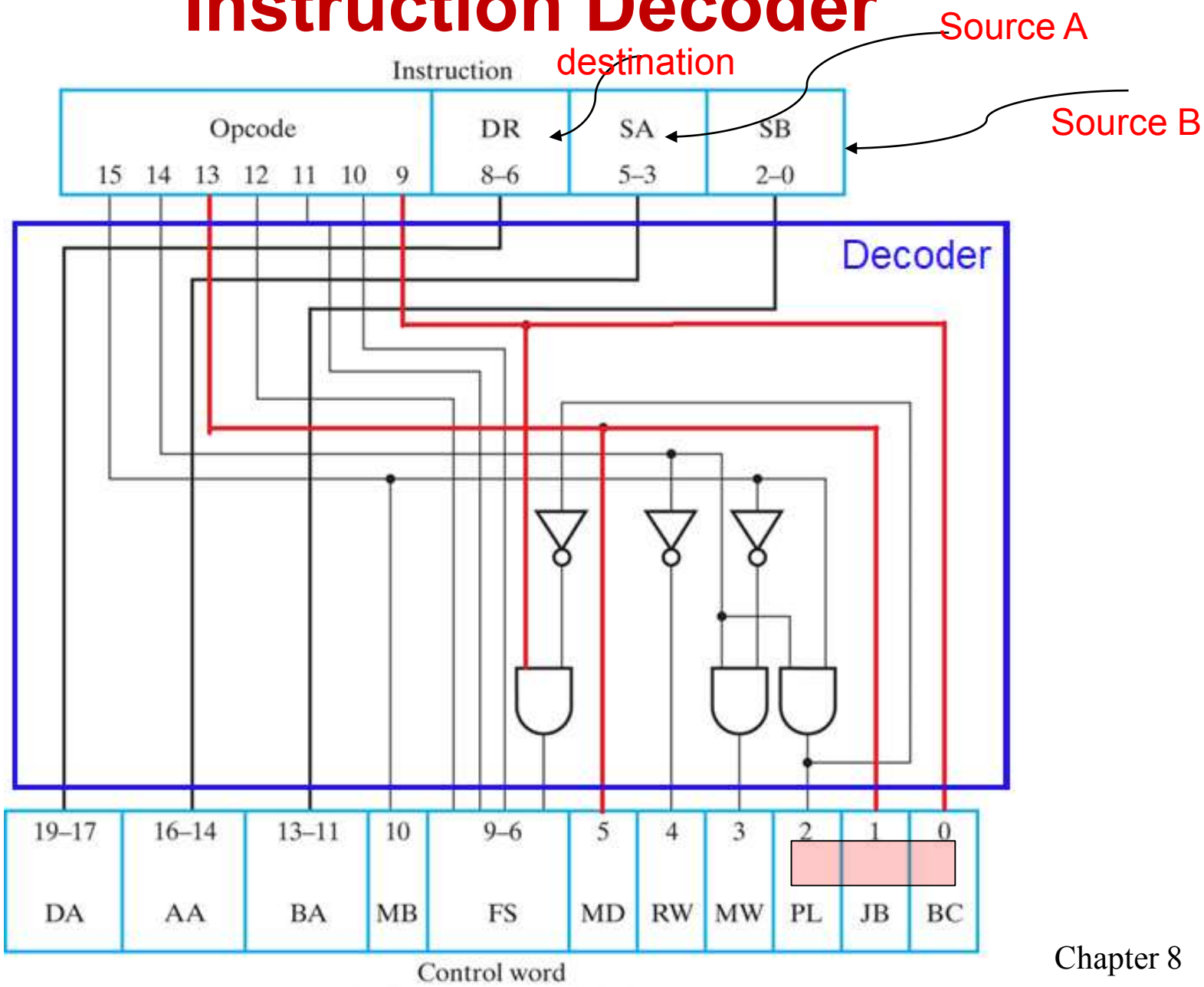
BC – if 0, do BRZ, otherwise do BRN



Control Unit



Instruction Decoder



Copyright ©2016 Pearson Education, All Rights Reserved

Instruction Decoder

- Control-word fields DA, AA, and BA are equal to the instruction fields DR, SA, and SB, respectively
- Control field BC for selection of the branch condition status bits is taken directly from the last bit of Opcode.
- The remaining control-word fields: MB, MD, RW, and MW
- There are two added bits for the control of the PC: PL and JB

TABLE 8-10
Truth Table for Instruction Decoder Logic

	Instruction Bits				Control-Word Bits						
Instruction Function Type	15	14	13	9	MB	MD	RW	MW	PL	JB	BC
Function-unit operations using registers	0	0	0	X	0	0	1	0	0	X	X
Memory read	0	0	1	X	0	1	1	0	0	X	X
Memory write	0	1	0	X	0	X	0	1	0	X	X
Function-unit operations using register and constant	1	0	0	X	1	0	1	0	0	X	X
Conditional branch on zero (Z)	1	1	0	0	X	X	0	0	1	0	0
Conditional branch on negative (N)	1	1	0	1	X	X	0	0	1	0	1
Unconditional jump	1	1	1	X	X	X	0	0	1	1	X

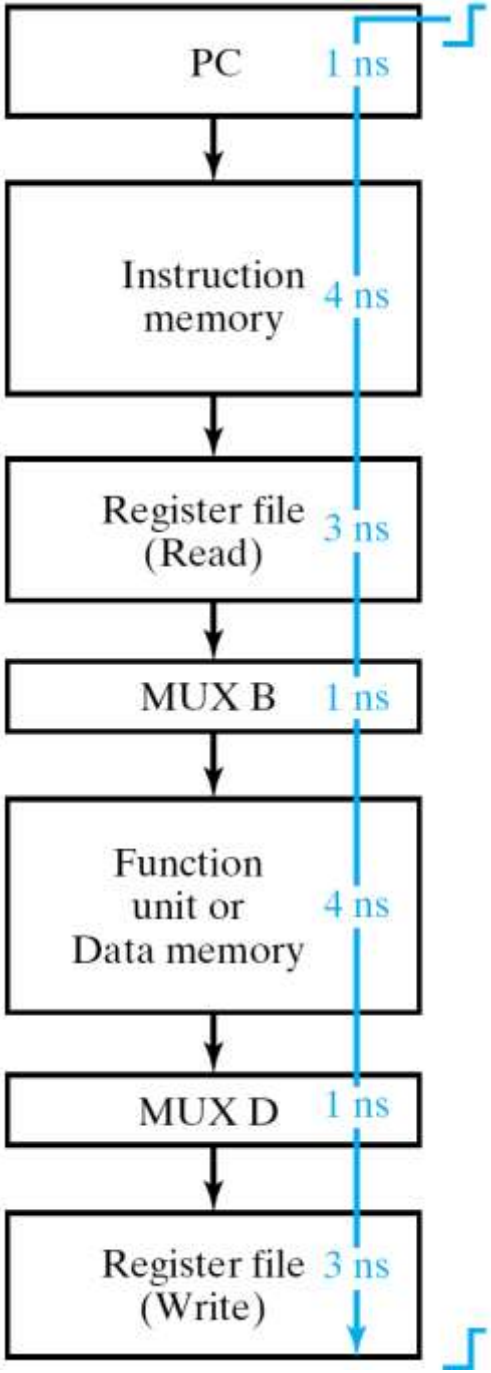
Sample Programs

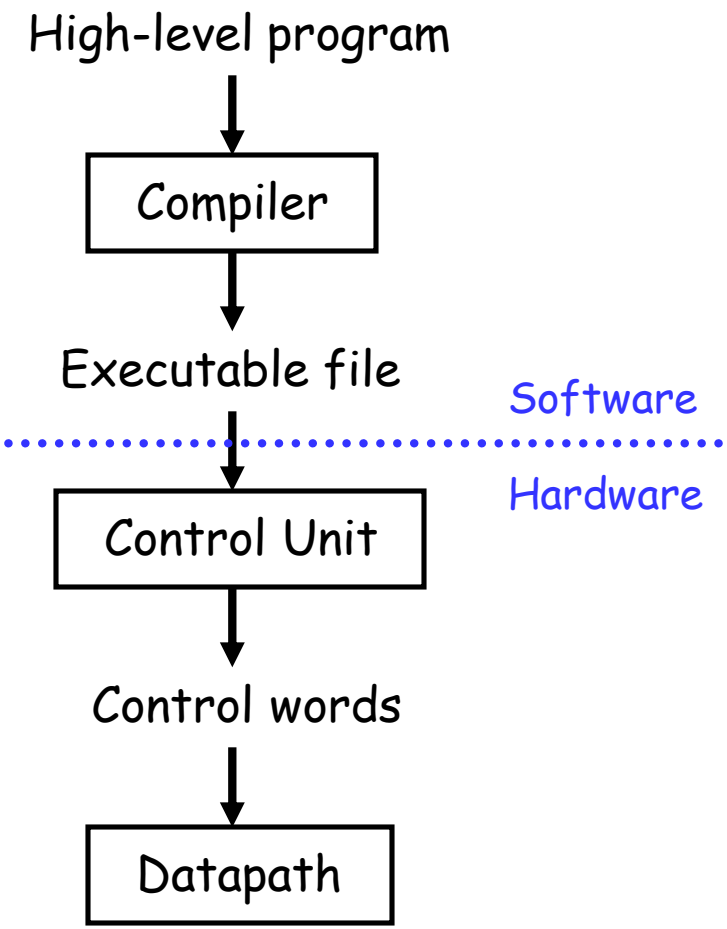
Example Program: Assume address 248 in data memory containing 2, and location 249 contains 83. Evaluation of arithmetic expression 83 - (2 + 3) and place the result in location 250.

LDI	R3, 248	Load address 248 in R3
LD	R1, (R3)	Load R1 with contents of location 248 in memory (R1 becomes = 2)
ADI	R1, R1, 3	Add 3 to R1 (R1 becomes =5)
INC	R3, R3	Increment the contents of R3 (R3 becomes = 249)
LD	R2, (R3)	Load R2 with contents of location 249 in memory (R2 becomes = 83)
SUB	R2, R2, R1	Subtract contents of R1 from contents of R2 (R2 becomes = 78)
INC	R3, R3	Increment the contents of R3 (R3 = 250)
ST	(R3), R2	Store R2 in memory location 250 (M[250] = 78)

Single-Cycle Computer Is

- Complex operations
  - Only combinational logic can be used in data transformation – no sequential logic
    - E.g. no multiplier
- Unified memory
  - If program and data are in one memory, how can you simultaneously access the same memory for an instruction and a data operand?
- Worst-case delay



Programming and CPUs

- Programs written in a high-level language like C++ must be compiled to produce an executable program.
- The result is a CPU-specific machine language program. This can be loaded into memory and executed by the processor.
- CSC 220 focuses on stuff below the dotted blue line, but machine language serves as the interface between hardware and software.
- Machine language instructions are sequences of bits in a specific order.
- To make things simpler, people typically use assembly language. Example assemblers for Intel x86 processor: NASM, YASM, MASM