Lecture 5

- Topics
  - Character codes
    - Binary-Code Decimal (BCD)
    - Extended Binary Coded Decimal (EBCD)
    - ASCII
    - Unicode
  - Error Detection and Correction

Character Codes

- Calculations aren't useful until their results can be displayed in a manner that is meaningful to people.
- We also need to store the results of calculations, and provide a means for data input.
- Thus, human-understandable characters must be converted to computer-understandable bit patterns using some sort of character encoding scheme.
- Character code schemes
  - Binary-Code Decimal (BCD)
  - Extended Binary Coded Decimal (EBCD)
  - ASCII
  - Unicode

Binary Coded Decimal

- BCD encodes each digit of a decimal number into 4-bit binary form.
- Example: the decimal digits of 146 are encoded to 0001, 0100, 0110, respectively.
- Computers use bytes as the smallest unit of access, most values are stored in eight bits, not four. How to store BCD?

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Zones	
1111	Unsigned
1100	Positive
1101	Negative

Warning: Conversion or Coding?

- Do NOT mix up conversion of a decimal number to a binary number with coding a decimal number with a binary code
- Conversion:
  - repeated division (multiplication) by 2 for integers (fractions)
- BCD coding:
  - Replacing each decimal digit of the number by its equivalent 4 bit BCD code
- $13_{10} = (1101)_2$ This is conversion
- $13 \Leftrightarrow (0001\ 0011)_{BCD}$ This is coding
- In general, coding requires more bits than conversion
- A number with n decimal digits is coded with $4n$ bits in BCD

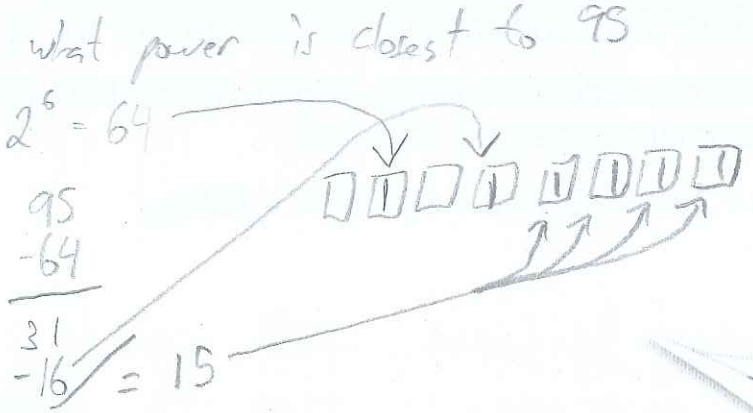
Exercise

Convert $(95)_{10}$ into its binary equivalent value and give its BCD code as well.

divide by 2 & use remainder

Answer:

$\{(0101\ 1111)_2$ and $1001\ 0101\}$



BCD-to-Decimal Conversion

Convert BCD code 1001 0100 0111 0000 to decimal

1001 0100 0111 0000

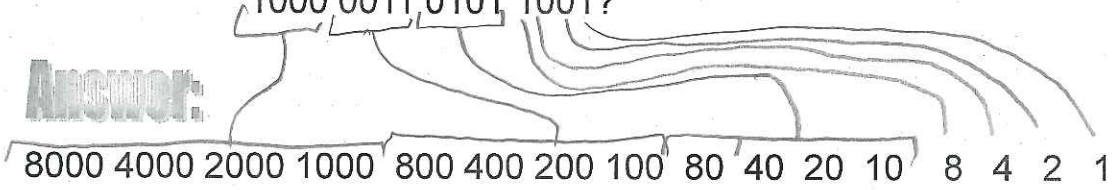
9 4 7 0

Binary Coded Decimal (BCD)

- You can think of BCD in terms of column weights in groups of four bits. For an 8-bit BCD number, the column weights are: 80 40 20 10 8 4 2 1.

Question: What are the column weights for the BCD number 1000 0011 0101 1001?

Answer:



Note that you could add the column weights where there is a 1 to obtain the decimal number. For this case:

$$8000 + 200 + 100 + 40 + 10 + 8 + 1 = 8359_{10}$$

Binary Coded Decimal

- BCD storage
  - computer only access 8-bits at a time
  - Unpacked BCD:
    - padding the high-order nibbles with zeros or ones.
      - 146: 00000001 00000100 00000110
    - wasteful — padded to make byte accessible
  - Packed BCD:
    - stores two digits per byte
    - allows numbers to be signed,
    - the sign (unsigned, positive, or negative) is stored at the end.
      - 146: 00010100 01101100 (8-bit, 8-bit)
        1 4 6 sign

Binary Coded Decimal

- BCD storage (con't)
  - Unpacked BCD:
  - Packed BCD:
  - Zoned decimal format of BCD:
    - Similar to unpacked BCD, except padding with a specific pattern in the high-order nibbles, called zone.
    - Allow signed numbers: sign is located in the high-order nibble of the least significant byte.
    - Extended BCD Interchange Code (EBCDIC) zoned decimal format: pad the zone with all ones (hexadecimal F).
      - 146: 11110001 11110100 11000110 — increases representation
    - ASCII zoned decimal format: pad the zone with 0011 (hexadecimal 3).
      - 146: 00110001 00110100 11000110

BCD Exercise

Represent 1265 using packed BCD and EBCDIC zoned decimal.

Solution:

The 4-bit BCD representation for 1265 is

0001 0010 0110 0101

Packed BCD: adding the sign after the low-order digit and padding the high-order bit with 0000,

00000001 00100110 01011101

EBCDIC zoned decimal: padding the high-order bits of the least significant byte with 1101 and padding the high-order bits of other bytes with 1111.

11110001 11110010 11110110 11010101

EBCDIC

- In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC).
- EBCDIC was one of the first widely-used computer codes that supported upper *and* lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
- EBCDIC and BCD are still in use by IBM mainframes today.

EBCDIC

	Digit															
Zone	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NUL	SOH	STX	ETX	PF	HT	LC	DEL		RLF	SMM	VT	FF	CR	SR	SI
0001	DLE	DC1	DC2	TM	RES	NL	BS	IL	CAN	EM	CC	CU1	IFS	IGS	IRS	IUS
0010	DS	SOS	FS		BYP	LF	ETB	ESC			SM	CU2		ENQ	ACK	BEL
0011			SYN		PN	RS	UC	EOT				CU3	DC4	NAK		SUB
0100	SP										[	.	<	(	+	!
0101	&										]	$	*	)	;	^
0110	–	/									!	,	%	_	>	?
0111										'	:	#	@	'	=	"
1000		a	b	c	d	e	f	g	h	i						
1001		j	k	l	m	n	o	p	q	r						
1010		~	s	t	u	v	w	x	y	z						
1011																
1100	{	A	B	C	D	E	F	G	H	I						
1101	}	J	K	L	M	N	O	P	Q	R						
1110	\		S	T	U	V	W	X	Y	Z						
1111	0	1	2	3	4	5	6	7	8	9						

a: 1000 0001	A: 1100 0001	3: 1111 0011

ASCII → Gives you 7-bit, need 8-bit

- Until recently, ASCII (American Standard Code for Information Interchange) was the dominant character code.
- ASCII: 1 high-order bit for parity, 7 bits for character codes.
- The parity bit is turned "on" or "off" depending on whether the sum of the other bits in the byte is even or odd.
  - "A": lower 7 bits is 1000001, parity bit: 0; 0100 0001 – [redacted] = even 2 (1's)
  - "C": lower 7 bits is 1000011, parity bit: 1; 1100 0011 – 3 (1's) = odd ↓ 1 extra bit makes it even
- Parity bit can be used to detect only single-bit errors.
- ASCII defines 32 control characters, 10 digits, 52 letters, 32 special characters.

Can also be made odd parity by making the examples above odd

- Used to check data when sent & received

ASCII

0	NUL	16	DLE	32		48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	TAB	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Unicode

- Both EBDIC and ASCII were built around the Latin alphabet and cannot provide data representation for the non-Latin alphabets.
- Many of today's systems embrace Unicode, a 16-bit system that can encode the characters of every language in the world.
- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

Unicode

- The Unicode codespace allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

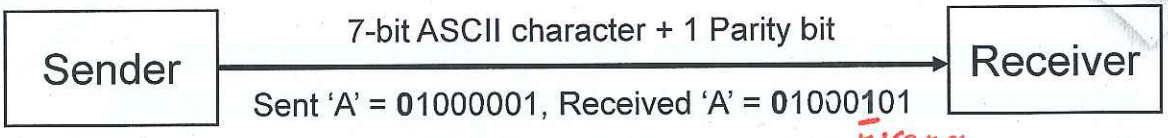
Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

Parity Bit & Error Detection Codes

- Binary data are typically transmitted between computers
- Because of noise, a corrupted bit will change value
- To detect errors, extra bits are added to each data value
- Parity bit: is used to make the number of 1's odd or even
- Even parity: number of 1's in the transmitted data is even
- Odd parity: number of 1's in the transmitted data is odd

7-bit ASCII Character	With Even Parity	With Odd Parity
'A' = 1000001	**0** 1000001	**1** 1000001
'T' = 1010100	**1** 1010100	**0** 1010100

Detecting Errors



- Suppose we are transmitting 7-bit ASCII characters
- A parity bit is added to each character to make it 8 bits
- Parity can detect all single-bit errors
  - If even parity is used and a single bit changes, it will change the parity to odd, which will be detected at the receiver end
  - The receiver end can detect the error, but cannot correct it because it does not know which bit is erroneous
- Can also detect some multiple-bit errors
  - Error in an odd number of bits

Exercise

Assign the proper even parity bit to the binary code 101101011111.

Answer: 1101101011111

An odd parity system receives the following code groups: 10110, 11010, 110011, 110101110100, and 1100010101010. Determine which groups, if any, are in error.

Answer: 110011, 1100010101010.

Error Detection and Correction

- It is physically impossible for any data recording or transmission medium to be 100% perfect 100% of the time over its entire expected useful life.
- Thus, error detection and correction is critical to accurate data transmission, storage and retrieval.
- Check digits, appended to the end of a long number, can provide some protection against data input errors.
- Cyclic redundancy checking (CRC) codes provide error detection for large blocks of data.

Cyclic Redundancy Check

- Checksums and CRCs are examples of systematic error detection.
- In systematic error detection, a group of error control bits is appended to the end of the block of transmitted data.
  - This group of bits is called a syndrome.
- CRCs are polynomials over the modulo 2 arithmetic field.
  - In modulo 2 arithmetic if we add 1 to 1, we get 0. The addition rules are:

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

Know

The mathematical theory behind modulo 2 polynomials is beyond our scope. However, we can easily work with it without knowing its theoretical underpinnings.

22

Modulo 2 Division

Modulo 2 division operates through a series of partial sums using the modulo 2 addition rules.

Example: Find the quotient and remainder when 1001011_2 is divided by 1011_2.

send this

polynomial Key

$x^3 + x^4 + x^3 + x^2$

1 1 1 1

$x^5 + 0^4 + x^3 + x^2$

1 0 1 1

```
1011)1001011
     1011
     0010

     001001
        1011
        0010
        00101
```

1. Write the divisor directly beneath the first bit of the dividend.
2. Add these numbers modulo 2.
3. Bring down bits from the dividend so that the first 1 of the difference can align with the first 1 of the divisor.
4. Copy the divisor as in Step 1.
5. Add as in Step 2.
6. Bring down another bit.
7. 101_2 is not divisible by 1011_2, so this is the remainder.

find remainder

The quotient is 1010_2.

append to original data

Faster than Hamming Code

Cyclic Redundancy Check

Sender side:

1. Suppose we want to transmit the information string: 1111101_2.
2. The receiver and sender decide to use the (arbitrary) polynomial pattern, 1101.
3. The information string is shifted left by one position less than the number of positions in the divisor. 4 - 1 = 3 zeroes
4. The remainder is found through modulo 2 division (at right) and added to the Message M: 1111101000 + 111 = 1111101111_2.

agreed polynomial Key

```
        1011011
1101)1111101000
     1101
     001010
       1101
       01111
        1101
        001000
          1101
          01010
           1101
           0111
```

shift

make 4-bit

Cyclic Redundancy Check

Receiver Side: M is decoded and checked.

- If no bits are lost or corrupted, dividing the received information string by the agreed upon pattern will give a remainder of zero.
- A remainder other than zero indicates that an error has occurred in the transmission of M.

```
        1011011
1101)1111101111
     1101
     001010
       1101
       01111
        1101
        001011
          1101
          01101
           1101
           0000
```

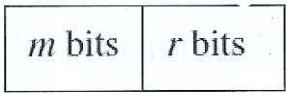
no corruption

Error Correction

- Hamming codes and ~~Reed-Solomon~~ not covered codes are two important error correcting codes.
- Hamming codes, also called check bits or redundant bits, are an adaption of the concept of parity, whereby error detection and correction capabilities are increased in proportion to the number of parity bits added to an information word.
- Reed-Solomon codes are particularly useful in correcting burst errors that occur when a series of adjacent bits are damaged.
  - Because CD-ROMs are easily scratched, they employ a type of Reed-Solomon error correction.

Hamming Codes

- Hamming codes are code words formed by adding redundant check bits, or parity bits, to a data word.



- The Hamming distance between two code words is the number of bits in which two code words differ.

This pair of bytes has a	1 0 0 0 1 0 0 1
Hamming distance of 3:	1 0 1 1 0 0 0 1

- The minimum Hamming distance for a code is the smallest Hamming distance between all pairs of words in the code.
- The minimum Hamming distance for a code, $D(\min)$, determines its error detecting and error correcting capability.

Hamming Codes

- For any code word, X, to be interpreted as a different valid code word, Y, at least D(min) single-bit errors must occur in X.
- Thus, to detect k (or fewer) single-bit errors, the code must have a Hamming distance of $D(\min) = k + 1$.
- Hamming codes can *detect* D(min) - 1 errors and *correct* $\left\lfloor \frac{D(\min) - 1}{2} \right\rfloor$ errors.
- Thus, a Hamming distance of $2k + 1$ is required to be able to correct k errors in any data word.

Hamming Codes

- Assume a memory with 2 data bits and 1 parity bit
- Even parity is used and the parity bit is appended at the end of the code word. (so the number of 1s in the codeword must be even).
- The resulting code words have 3 bits. However, using 3 bits allows for 8 different bit patterns.
- Error correcting codes require more than a single parity bit.
  - If the code word 001 is encountered, it is invalid and thus indicates an error. This error can be detected, but it cannot be corrected.

Data Word	Parity Bit	Code Word
00	0	000
01	1	011
10	1	101
11	0	110

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

Hamming Codes

- Suppose we have a set of *n*-bit code words consisting of *m* data bits and *r* (redundant) parity bits.
- Suppose also that we wish to detect and correct one single bit error only.
- An error could occur in any of the *n* bits, so each code word can be associated with *n* invalid code words at a distance of 1.
- Therefore, we have *n* + 1 bit patterns for each code word: one valid code word, and *n* invalid code words

Hamming Codes

- Using n bits, we have 2^n possible bit patterns. We have 2^m valid code words with r check bits (where $n = m + r$).
- For each valid code word, we have $(n + 1)$ bit patterns (1 legal and n illegal).
- This gives us the inequality:

$$(n + 1) \times 2^m \leq 2^n$$

- Because $n = m + r$, we can rewrite the inequality as:

$(m + r + 1) \times 2^m \leq 2^{m+r}$ or $(m + r + 1) \leq 2^r$

  - This inequality gives us a lower limit on the number of check bits that we need in our code words.

Hamming Codes

- Suppose we have data words of length $m = 4$. Then:
  $(4 + r + 1) \leq 2^r$
  implies that r must be greater than or equal to 3.
  - This means to build a code with 4-bit data words that will correct single-bit errors, we must add 3 check bits.
- Suppose we have data words of length m = 8. Then:
  $(8 + r + 1) \leq 2^r$
  implies that r must be greater than or equal to 4.
  - This means to build a code with 8-bit data words that will correct single-bit errors, we must add 4 check bits, creating code words of length 12.

Hamming Algorithm

1. Determine the number of check bits, r, necessary for the code and then number the n bits (where $n = m + r$), right to left, starting with 1 (not 0)
2. Each bit whose bit number is a power of 2 is a parity bit—the others are data bits.
3. Assign parity bits to check bit positions as follows: Bit b is checked by those parity bits $b_1, b_2, \cdots, b_j$ such that $b_1 + b_2 + \cdots + b_j = b$. (Where "+" indicates the modulo 2 sum.)

Hamming Codes – Example 1

Using the Hamming code and even parity, encode the 8-bit ASCII character *K*. (The high-order bit will be zero.) Induce a single-bit error and then indicate how to locate the error.

Solution:

Step 1: Determine the number of necessary check bits, add these bits to the data bits, and number all n bits.

Because $m = 8$, we have $(8 + r + 1) \leq 2^r$, which implies r must be greater than or equal to 4. We choose $r = 4$.

Step 2: Number the n bits right to left, starting with 1, which results in:

				P				P		P	P
12	11	10	9	8	7	6	5	4	3	2	1

The parity bits are marked with P.

Hamming Codes – Example 1

Step 3: Assign parity bits to check the various bit positons.

- First express all bits positions as sum of those numbers that are powers of 2:

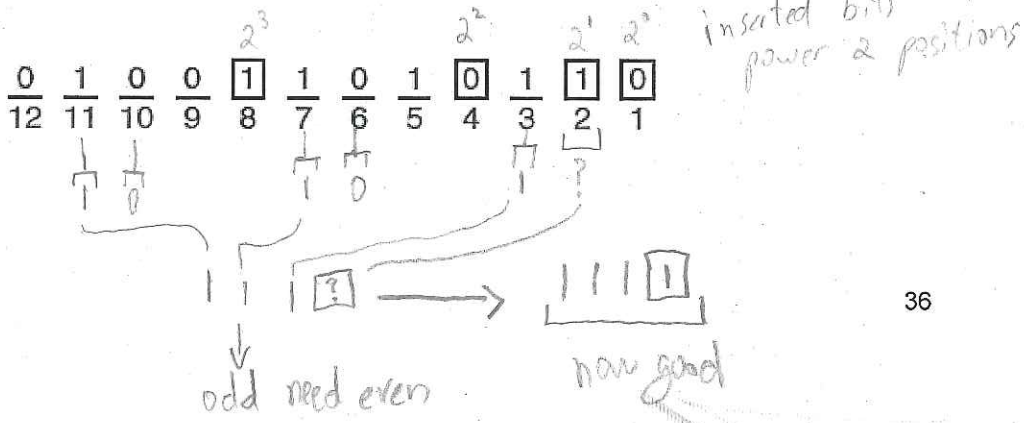
addition by power of 2 #'s

$1 = 2^0$	$5 = 2^2 + 2^0$	$9 = 2^3 + 2^0$
$2 = 2^1$	$6 = 2^2 + 2^1$	$10 = 2^3 + 2^1$
$3 = 2^1 + 2^0$	$7 = 2^2 + 2^1 + 2^0$	$11 = 2^3 + 2^1 + 2^0$
$4 = 2^2$	$8 = 2^3$	$12 = 2^3 + 2^2$

  - Bit 1 contributes to 1, 3, 5, 7, 9, 11.
  - Bit 2 contributes to the digits, 2, 3, 6, 7, 10, and 11.
  - Bit 4 provides parity for 4, 5, 6, 7, and 12
  - Bit 8 provides parity for 8, 9, 10, 11, 12.
- We can use this idea in the creation of our check bits.

Hamming Codes – Example 1

- We can use this idea in the creation of our check bits.
  - Bit 1 contributes to 1, 3, 5, 7, 9, 11.
  - Bit 2 contributes to the digits, 2, 3, 6, 7, 10, and 11.
  - Bit 4 provides parity for 4, 5, 6, 7, and 12
  - Bit 8 provides parity for 8, 9, 10, 11, 12.
- The ASCII character for K is 01001011.
- The code word for K is 010011010110



Hamming Codes – Example 1

- Let introduce an error in bit position b_9, resulting in the code word 010111010110. P P PP P = parity bit
  - Bit 1 checks 1, 3, 5, 7, 9, and 11: With even parity, this produces an error. → got 5, odd, error
  - Bit 2 checks 2, 3, 6, 7, 10, and 11: This is ok.
  - Bit 4 checks 4, 5, 6, 7, and 12: This is ok.
  - Bit 8 checks 8, 9, 10, 11, and 12: This produces an error. → got 3, odd, error
- Parity bits 1 and 8 show errors. These two parity bits both check 9 and 11, so the single bit error must be in either bit 9 or bit 11.
- Since bit 2 checks bit 11 and indicates no error has occurred in the subset of bits it checks, the error must occur in bit 9.

Lecture 7

- Topics
  - Boolean Algebra

0 = False

1 = True

1

Logic and Bits Operation

- Computers represent information by bit
- A bit has two possible values, namely zero and one.
- A bit can be used to represent a truth value, since there are two truth values, true and false.
- Bits operations correspond to the logical operations in Boolean Algebra.

2

Boolean Variables & Boolean Operators

- Boolean variables are variables that can take only binary values: 0 or 1, false or true
  - A,B,C = {0, 1}
- Boolean Operators
  - AND (A AND B, A·B, A ∧ B)
  - OR (A OR B, A+B, A ∨ B)
  - NOT (NOT A, A') ¬A , $\bar{A}$

3

Boolean Algebra

- Boolean algebra is a mathematical system for the manipulation of variables that can have one of two values.
  - In formal logic, these values are "true" and "false."
  - In digital systems, these values are "on" and "off," 1 and 0, or "high" and "low."
- Boolean expressions are created by performing Boolean operations on Boolean variables.
  - Common Boolean operators include AND, OR, and NOT.

4

Truth Table of Boolean Operators

- A Boolean operator can be completely described using a truth table.
- The AND operator is also known as a Boolean product. The OR operator is the Boolean sum.

X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT X

X	$\overline{X}$
0	1
1	0

5

Boolean Function

- A function is a relation that uniquely associates members of one set with members of another set
- A Boolean function :
  - has at least one Boolean variable,
  - has least one Boolean operator, and
  - yield a result, based on input values, in the set {0,1}.
- Example: $F(x, y, z) = x\bar{z} + y$
- Truth table can be used to represent a Boolean function.
- To make evaluation of the Boolean function easier, the truth table contains extra (shaded) columns to hold evaluations of subparts of the function.

$F(x, y, z) = x\bar{z}+y$

x	y	z	$\bar{z}$	$x\bar{z}$	$x\bar{z}+y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

Order of Boolean Operations

- There might be many Boolean operators in one Boolean function. Which operator to apply first?
- Order of Boolean Operations: NOT > AND > OR

precedence
1st = NOT
2nd = AND
3rd = OR

$F(x, y, z) = x\bar{z} + y$

x	y	z	$\bar{z}$	$x\bar{z}$	$x\bar{z}+y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

7

In Class Exercise

Construct a Truth Table for $yz + z(xy)'$

X	Y	Z	YZ	XY	XY'	(XY)'	Z(XY)'	YZ + Z(XY)'
0	0	0	0	0	1	1	0	0
0	0	1	0	0	1	1	1	1
0	1	0	0	0	1	1	0	0
0	1	1	1	0	1	1	1	1
1	0	0	0	0	1	1	0	0
1	0	1	0	0	1	1	1	1
1	1	0	0	1	0	0	0	0
1	1	1	1	1	0	0	0	1

8

Boolean Identities: Simplify a Boolean Function

- Digital computers contain circuits that implement Boolean functions.
- The simpler that we can make a Boolean function, the smaller the circuit that will result.
  - Simpler circuits are cheaper to build, consume less power, and run faster than complex circuits.
- With this in mind, we always want to reduce our Boolean functions to their simplest form.
- There are a number of Boolean identities that help us to simplify a Boolean function.

9

Boolean Identities

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null (or Dominance) Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$xx' = 0$	$x + x' = 1$
Commutative Law	$xy = yx$	$x + y = y + x$
Associative Law	$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$
Distributive Law	$x + (yz) = (x + y)(x + z)$	$x(y + z) = xy + xz$
Absorption Law	$x(x + y) = x$	$x + xy = x$
DeMorgan's Law	$(xy)' = x' + y'$	$(x + y)' = x'y'$
Double Complement Law	$x'' = x$	

TABLE 3.5 Basic Identities of Boolean Algebra

10

How to prove these identities

- All of the above identities can be proved using truth tables .
- To do this, you use truth tables to show all of the possible values of both sides of the equation.
- If they are identical, then the identity is true.

11

Proving the AND Form of DeMorgan's Law

x	y	(xy)	$(xy)'$	x'	y'	$x' + y'$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

TABLE 3.6 Truth Table for the AND Form of DeMorgan's Law

12

Simplification of Boolean Expressions

Example 3.3 Simplify the function:

$$F(x, y, z) = y + \overline{xy}$$

Know

Solution:

$$\begin{aligned} F(x, y, z) &= y + \overline{xy} \\ &= y + (\bar{x} + \bar{y}) && \text{(De Morgan's)} \\ &= y + (\bar{y} + \bar{x}) && \text{(Commutative)} \\ &= (y + \bar{y}) + \bar{x} && \text{(Associative)} \\ &= 1 + \bar{x} && \text{(Inverse)} \\ &= 1 && \text{(Null)} \end{aligned}$$

13

Simplification of Boolean Expressions

Example 3.4 Simplify the function:

$$F(x, y, z) = \overline{xy}(\bar{x} + y)(y + \bar{y})$$

Solution:

$$\begin{aligned} F(x, y, z) &= \overline{xy}(\bar{x} + y)(y + \bar{y}) \\ &= \overline{xy}(\bar{x} + y)(1) && \text{(Inverse)} \\ &= \overline{xy}(\bar{x} + y) && \text{(Identity)} \\ &= (\bar{x} + \bar{y})(\bar{x} + y) && \text{(DeMorgan's)} \\ &= \bar{x} + \bar{y}y && \text{(Distributive over AND)} \\ &= \bar{x} + 0 && \text{(Inverse)} \\ &= \bar{x} && \text{(Identity)} \end{aligned}$$

14

Simplification of Boolean Expressions

Example 3.5 Simplify the function:

$$F(x, y, z) = \bar{x}(x + y) + (y + x)(x + \bar{y})$$

Solution:

$$\begin{aligned}
F(x, y, z) &= \bar{x}(x + y) + (y + x)(x + \bar{y}) \\
&= (x + y)\bar{x} + (x + y)(x + \bar{y}) && \text{(Commutative)} \\
&= (x + y)(\bar{x} + x + \bar{y}) && \text{(Distributive over OR)} \\
&= (x + y)(1 + \bar{y}) && \text{(Inverse)} \\
&= (x + y)(1) && \text{(Null)} \\
&= x + y && \text{(Identity)}
\end{aligned}$$

15

Simplification of Boolean Expressions

Example 3.6 Simplify the function:

$$F(x, y, z) = xy + x'z + yz$$

$$\begin{aligned}
F(x,y,z) &= xy + x'z + yz \\
&= xy + x'z + yz(1) && \text{(Identity)} \\
&= xy + x'z + yz(x + x') && \text{(Inverse)} \\
&= xy + x'z + (yz)x + (yz)x' && \text{(Distributive)} \\
&= xy + x'z + x(yz) + x'(zy) && \text{(Commutative)} \\
&= xy + x'z + (xy)z + (x'z)y && \text{(Associative twice)} \\
&= xy + (xy)z + x'z + (x'z)y && \text{(Commutative)} \\
&= xy(1 + z) + x'z(1 + y) && \text{(Distributive)} \\
&= xy(1) + x'z(1) && \text{(Null)} \\
&= xy + x'z && \text{(Identity)}
\end{aligned}$$

16

Boolean Algebra Properties

- $xy' + y = x + y$

Proof:

$$xy' + y = (x + y)(y + y')$$
$$= x + y$$

In Class Exercise

Simplify the following functional expressions using Boolean algebra and its identities.

$x(yz + y'z) + xy + x'y + xz$ $= (XZ+Y)$

$XZ(Y+Y') + XZ + Y(X+X')$

$XZ(1) + XZ + Y(1)$

$XZ + XZ + Y$

$(XZ+Y)$

18

Complements

- Sometimes it is more economical to build a circuit using the complement of a function than it is to implement the function directly.
- DeMorgan's law provides an easy way of finding the complement of a Boolean function.
- Recall DeMorgan's law states:

$$\overline{(xy)} = \bar{x} + \bar{y} \quad \text{and} \quad \overline{(x+y)} = \bar{x}\bar{y}$$

19

Complements

- Extending to DeMorgan's law: Replace each variable by its complement and change all ANDs to ORs and all ORs to ANDs.
- Thus, the complement of:

$$F(x, y, z) = (xy) + (\bar{x}z) + (y\bar{z})$$

is:

$$\begin{aligned}\overline{F(x, y, z)} &= \overline{(xy) + (\bar{x}z) + (y\bar{z})} \\ &= \overline{(xy)} \cdot \overline{(\bar{x}z)} \cdot \overline{(y\bar{z})} \\ &= (\bar{x} + \bar{y}) \cdot (x + \bar{z}) \cdot (\bar{y} + z)\end{aligned}$$

20

In Class Exercise

Using DeMorgan's Law, write an expression for the complement of F if $F(x, y, z) = xy'(x + z)$

$\overline{x \cdot y' \cdot (x + z)}$

$\bar{x} + \bar{y} + (x + z)$

$x' + y + x' \cdot z'$

$x'(1 + z') + y$

$\rightarrow x'(1) + y$

$x' + y$

21

Representing Boolean Functions

- Many ways to represent a given Boolean function.
  - Truth table
  - Boolean expressions: an infinite number of Boolean expressions that are logically equivalent to one another.
- Two Boolean expressions that can be represented by the same truth table are considered logically equivalent.
- In order to eliminate confusion, designers express Boolean functions in standardized or canonical form.
  - The sum-of-products form: ANDed variables are ORed together, e.g., $F(x, y, z) = xy + yz + xz$.
  - The product-of-sums form: ORed variables are ANDed together: $F(x, y, z) = (x + y)(y + z)(x + z)$

22

Sum-of-Products Form

- Any Boolean expression can be represented in sum-of-products form.
- It is easy to convert a truth table to sum-of-products form: ORed together the values of the variables that result a true function value (1).
- $F(x,y,z) = \bar{x}y\bar{z} + \bar{x}yz + x\bar{y}\bar{z} + xy\bar{z} + xyz$
- Note that it is not in simplest terms.
- Simplify the function and obtain $F(x,y,z) = x\bar{z} + y$

$F(x,y,z) = x\bar{z}+y$

x	y	z	$x\bar{z}+y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

23

In Class Exercise

The true table for a Boolean expression is shown below. Write the Boolean expression in sum-of-products form.

x	y	z	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$x'y'z' + x'y'z + xy'z' + xyz$

24

Summary

- Given a boolean function, construct it's truth table.
- Using truth table to prove a boolean equation is valid or not.
- Applying boolean algebra for boolean function simplification.
- Using DeMorgan's Law, write an expression for the complement.
- Determine the boolean expression in sum-of-products form.

25

Lecture 8

- Topics
  - Switch
  - Transistor
  - CMOS transistor
  - Logic gates
    - AND, OR, NOT
    - Universal gates: NAND, NOR
    - XOR

Build the Binary Computer

- Formally, it is possible to construct a binary computer using any device that meets the following four conditions:
  - It has two stable energy states (for 0 and 1).
  - These two states are separated by a large energy barrier (so a 0 does not accidentally become a 1, or the reverse).
  - It is possible to sense what state the device is in (to see if it is storing a 0 or a 1) without permanently destroying the stored value.
  - It is possible to switch from a 0 to a 1 and vice versa

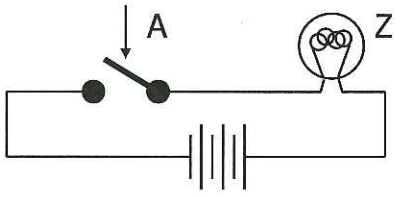
Claude Shannon

- His master's thesis in 1937, A Symbolic Analysis of Relay and Switching Circuits, is considered as "possibly the most important, and also the most famous, master's thesis of the century."

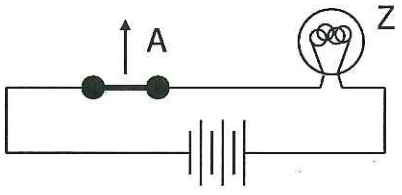
- He came up with the idea that electrical switches can be used to do Boolean logic.

Switches: basic element of physical implementations

- Implementing a simple circuit (arrow shows action if wire changes to "1"):



Close switch (if A is "1" or asserted) and turn on light bulb (Z)

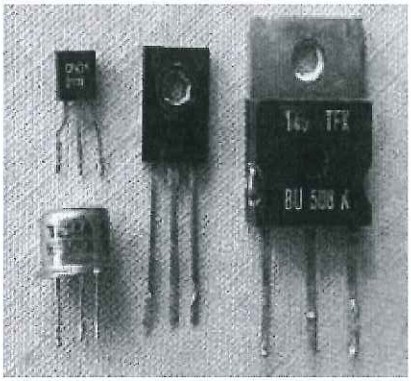


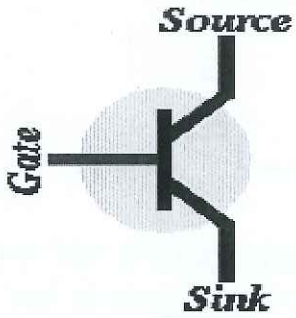
Open switch (if A is "0" or unasserted) and turn off light bulb (Z)

$Z \equiv A$

Transistor

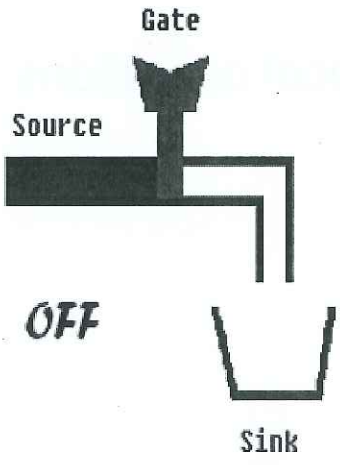
- A transistor is a discrete electronic component that can behave like a switch
- Low cost, flexibility and reliability
- The greatest invention of the twentieth century

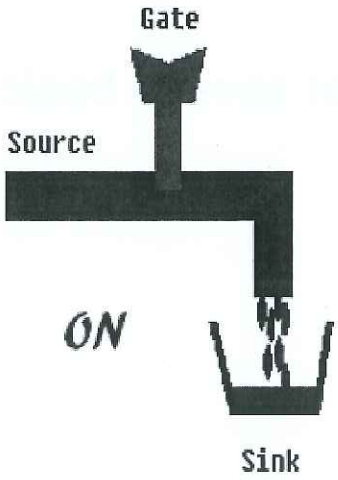


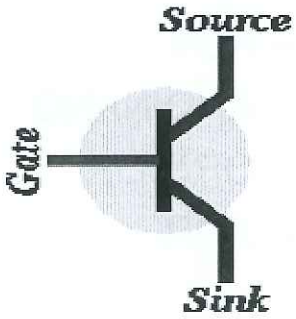


Water Flow Example

- Gate on, Water flow: 1
- Gate off, Water not flow: 0

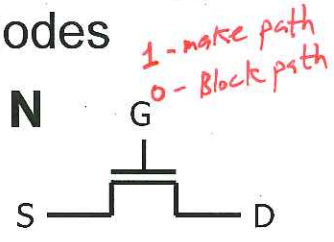




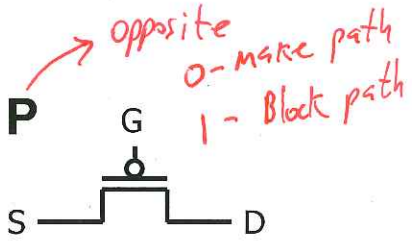


CMOS Transistors

- CMOS
  - Two versions: P-type (positive) and N-type (negative)
  - P and N-type transistors operate in inverse modes



Open (insulating) if gate is "off" = 0
Closed (conducting) if gate is "on" = 1



Open (insulating) if gate is "on" = 1
Closed (conducting) if gate is "off" = 0

From Transistors to Logic Gates

- Using transistors as building blocks, we can build larger circuits that perform logical operations

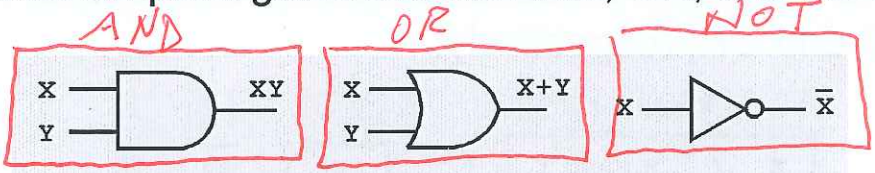
- Next we will look at several basic logical operations
  - NOT
  - AND/NAND
  - OR/NOR
  - XOR

Logic Gates

- Boolean functions are implemented in digital computer circuits called logic gates.
- A gate is an electronic device that produces a result based on two or more input values.
- In order words, a gate implements a simple Boolean function.
- In reality, gates consist of one to six transistors, but digital designers think of them as a single unit.
- Integrated circuits contain collections of gates suited to a particular purpose.

AND, OR and NOT Gates

- The three simplest gates are the AND, OR, and NOT gates.



X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT X

X	$\overline{X}$
0	1
1	0

- They correspond directly to their respective Boolean operations as shown in their truth tables.

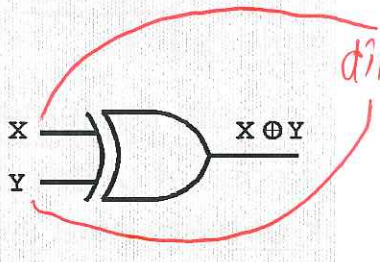
XOR Gate

- The output of the XOR operation, denoted with symbol $\oplus$, is true only when the values of the inputs differ.

X XOR Y

X	Y	X⊕Y
0	0	0 F
0	1	1 T
1	0	1 T
1	1	0 F

opposite
opposite



different input = 1

- $x \oplus y = \bar{x}y + x\bar{y}$ = 1·0 + 0·1 = 0+0 = 0

Sum of products

- $\overline{x \oplus y} = \bar{x}\bar{y} + xy$

NAND and NOR Gates

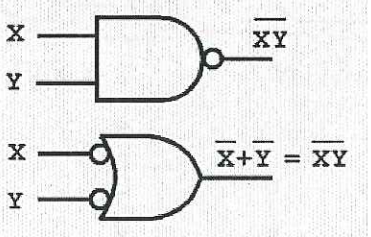
not and not OR

do operation then flip bit

- NAND and NOR produce complementary output to AND and OR, respectively.

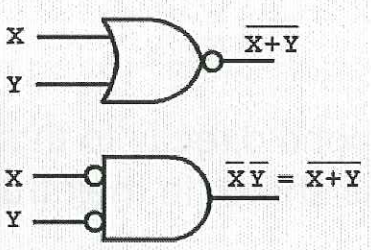
X NAND Y

X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0



X NOR Y

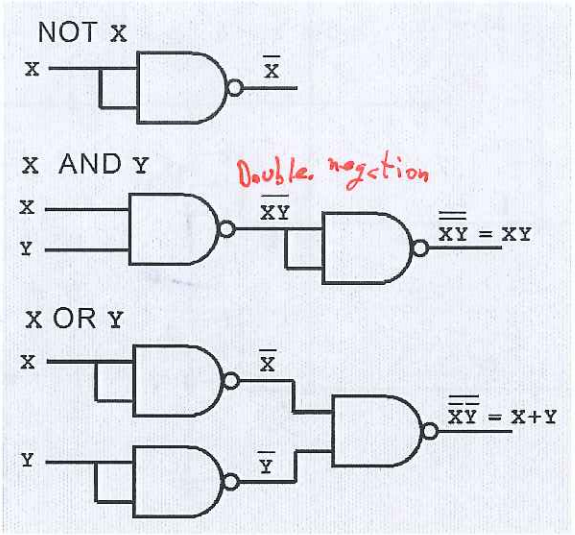
X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0



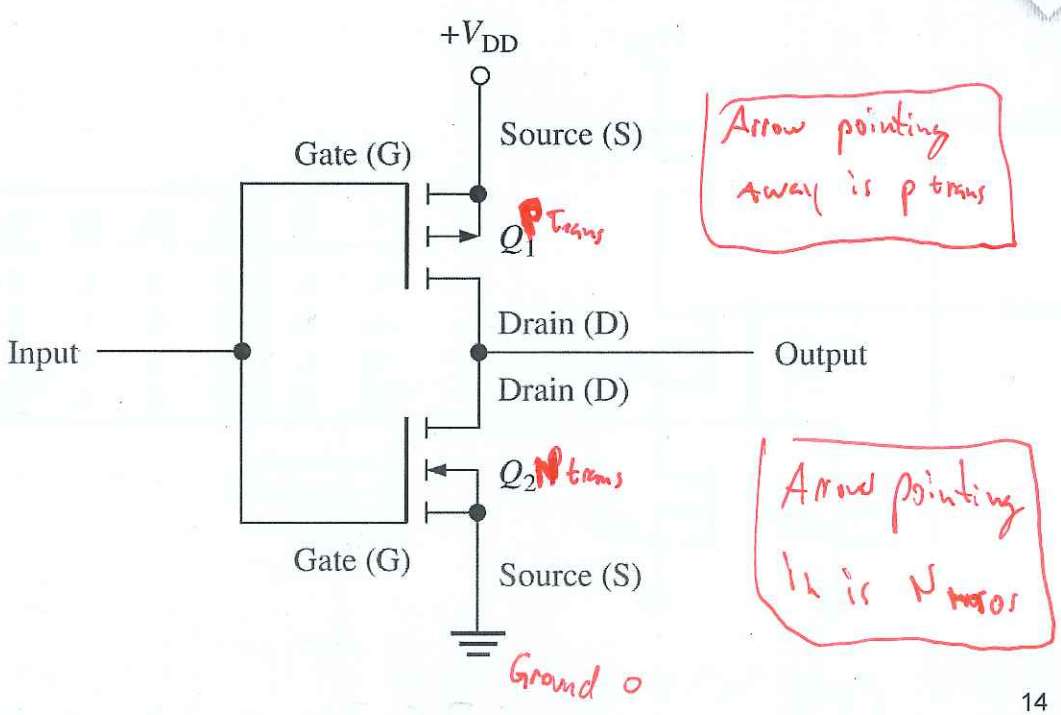
Universal Gates

- NAND and NOR are known as universal gates
  - inexpensive to manufacture
  - any Boolean function can be constructed using only NAND or only NOR gates.

Any gate can be constructed with these two gates



Inverter Gate (NOT)

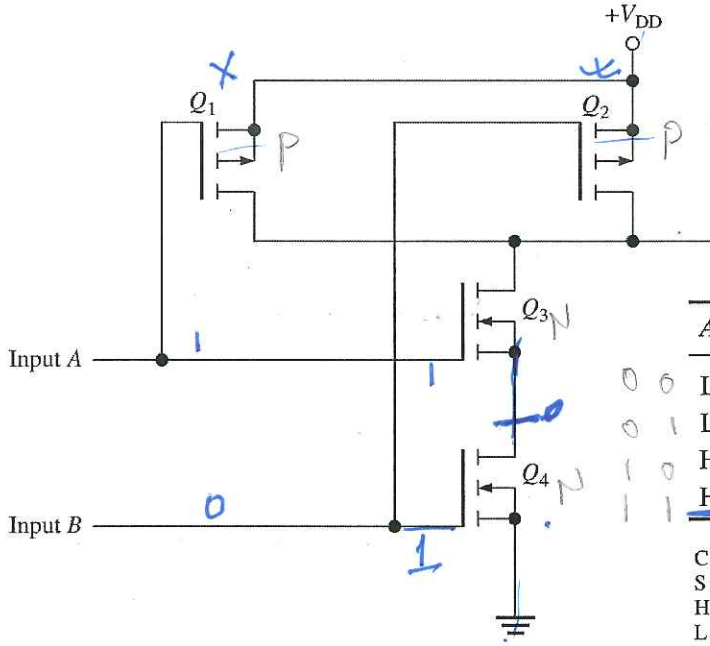


Arrow pointing away is p trans

Arrow pointing in is N trans

AND & NAND Gate

NAND = PMOS in parallel & NMOS in series

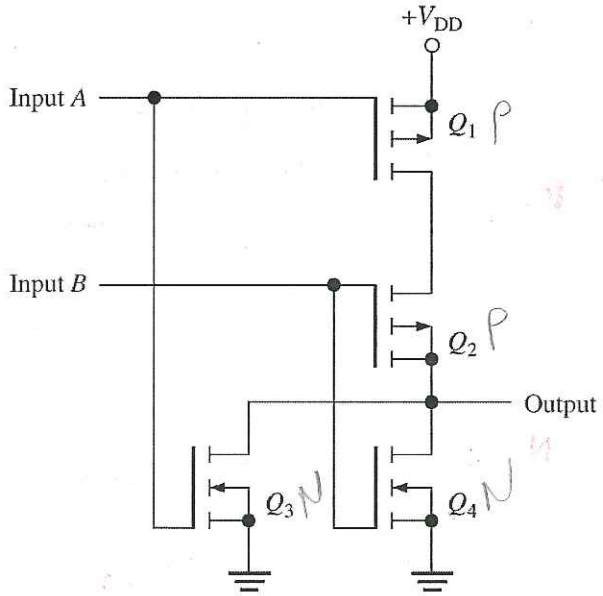


A	B	Q_1	Q_2	Q_3	Q_4	X
L	L	S	S	C	C	H
L	H	S	C	C	S	H
H	L	C	S	S	C	H
H	H	C	C	S	S	L

C = cutoff (off)
S = saturation (on)
H = HIGH
L = LOW

NAND
AND
N (AND)
00 → 0 → 1
01 → 0 → 1
10 → 0 → 1
11 → 1 → 0

OR & NOR Gate



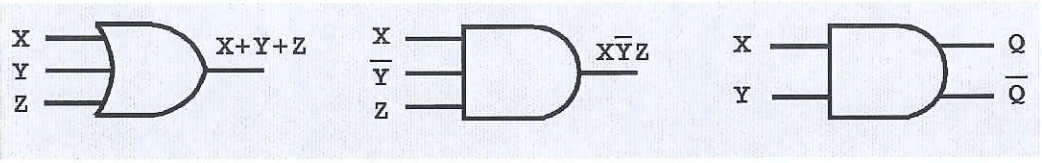
A	B	Q_1	Q_2	Q_3	Q_4	X
L	L	S	S	C	C	H
L	H	S	C	C	S	L
H	L	C	S	S	C	L
H	H	C	C	S	S	L

C = cutoff (off)
S = saturation (on)
H = HIGH
L = LOW

NOR = PMOS in series & NMOS in parallel

Multiple Input Gates

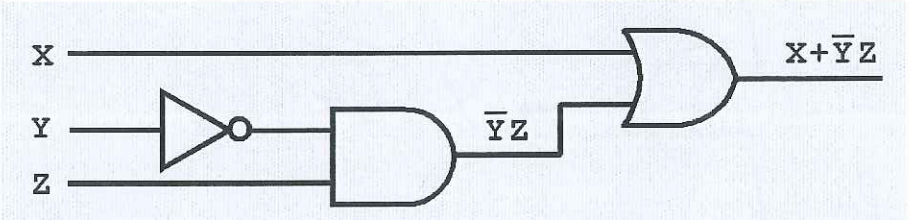
- Gates can have multiple inputs and more than one output.
  - A second output can be provided for the complement of the operation.
  - We'll see more of this later.



Digital Components

- Digital circuits are all constructed using the logical gates, which implement logical operations.
- The combinations of gates implement Boolean functions.
- Boolean algebra: analyze and design digital circuits.
- The circuit below implements the Boolean function:

$$F(x, y, z) = x + \bar{y}z$$



We simplify our Boolean expressions so that we can create simpler circuits.

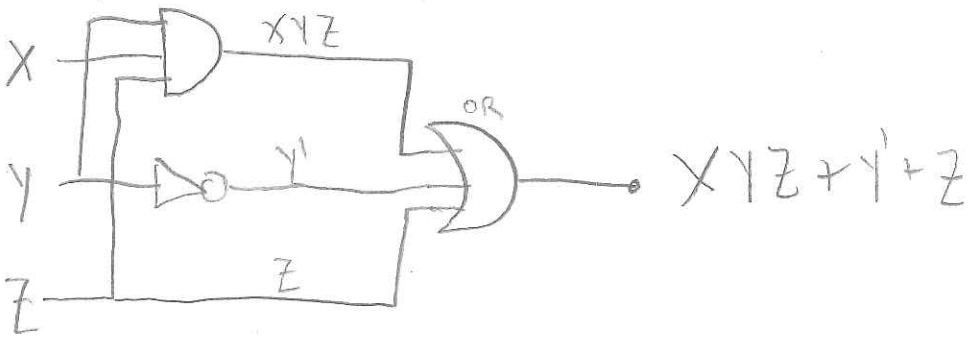
In Class Exercise

Construct the XOR operator using only AND, OR, and NOT gates.

In Class Exercise

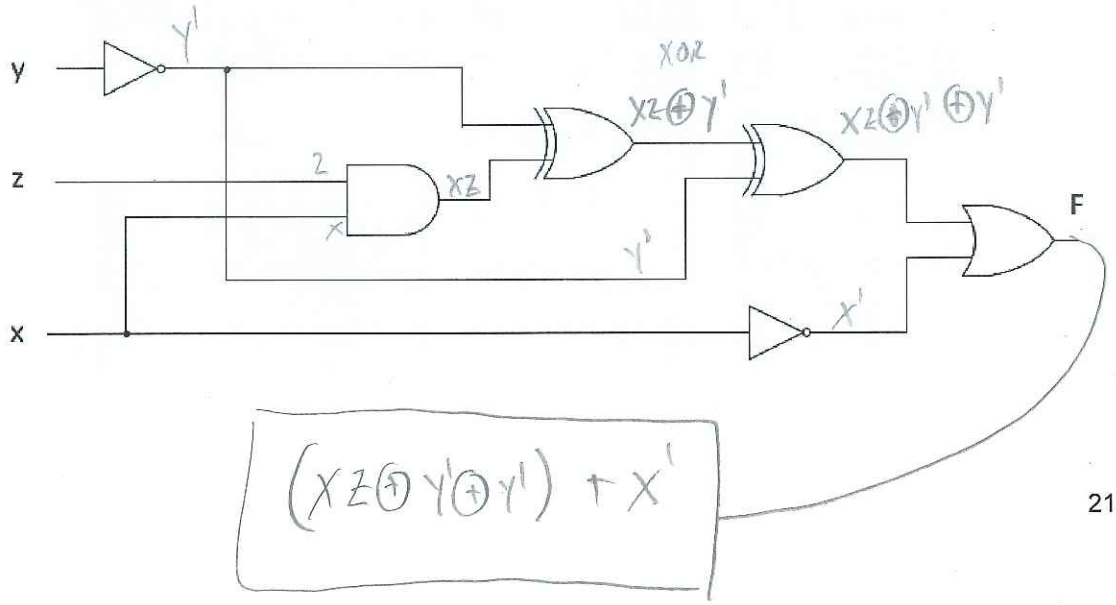
Draw the combinational circuit that directly implements the Boolean expression:

$F(x, y, z) = xyz + y' + z$



In Class Exercise

Find the truth table that describes the following circuit.



Lecture 9

- Topics:
  - Combinational circuits
    - Basic concepts
    - Examples of typical combinational circuits
      - Half-adder
      - Full-adder
      - Ripple-Carry adder
      - Decoder
      - Multiplexer
      - Bit shifter

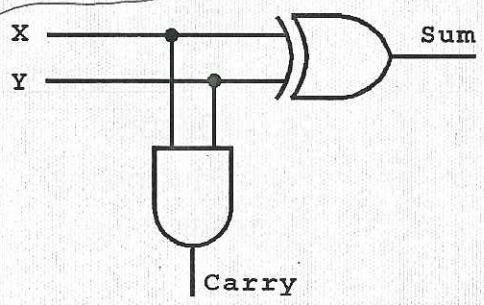
Combinational Circuits

- Digital logic circuits can be categorized as:
  - Combinational circuits
  - Sequential circuits
- Combination logic is used to build circuits that contain basic Boolean operators, inputs and outputs.
- The key to recognize a combinational circuit
  - an output is always based entirely on the given inputs.
- Combinational circuit:
  - The output of a combinational circuit is a function of its inputs,
  - the output is uniquely determined by the values of the inputs at any given moments.

Half-Adder - 2 bits & carry

- Half adder:
  - A typical combinational circuit
  - Adding two binary digits together.
- How to construct a half-adder?
- The truth table reveals that
  - Sum is actually an XOR.
  - Carry is equivalent to an AND gate.
- Combining an XOR gate and an AND gate results in the logic diagram for a half-adder.

Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



X Y AND Gate

X'Y + XY

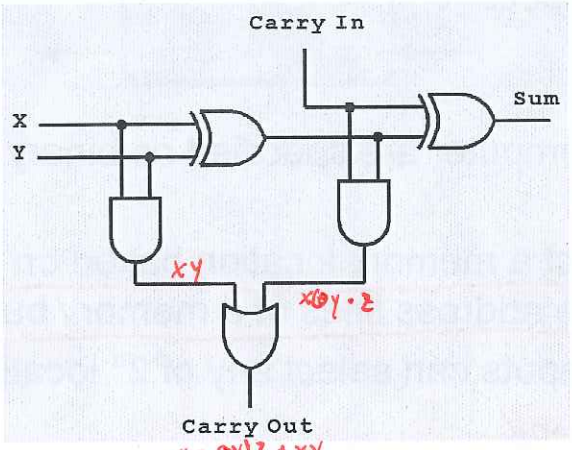
XOR gate

Full-Adder

- How do you add base 10 number?
  - Add up the rightmost column (units digit), and carry the tens digit.
  - Then add that carry to the current column and continue in a similar fashion.
- Binary numbers are added in the same way.
- A circuit that allows three inputs (x, y, and carry−in) and two outputs (Sum and Carry out) is required.
- A half-adder can be changed to to a full adder by including gates for processing the carry bit.

Full-Adder

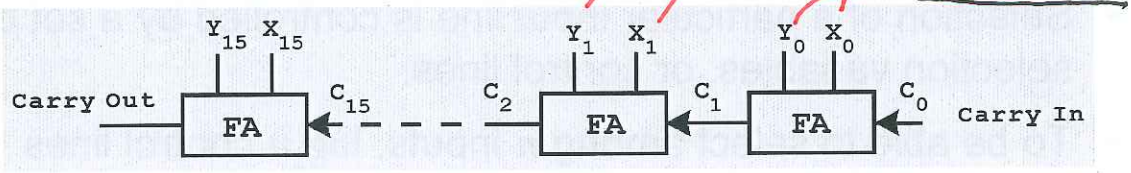
- Sum: $\bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}\bar{z} + xyz = x \oplus y \oplus z$
- Carry-out: $\bar{x}yz + x\bar{y}z + xy\bar{z} + xyz = (x \oplus y)z + xy$
- Note that this full-adder is composed of two half-adder and an OR gate.



Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Ripple-Carry Adder

- A full-adder is capable of adding only three bits.
- Full adders can connected in series to add binary numbers. This type of circuit is called a ripple-carry adder because of the sequential generation of carries that "ripple" through the adder stages.

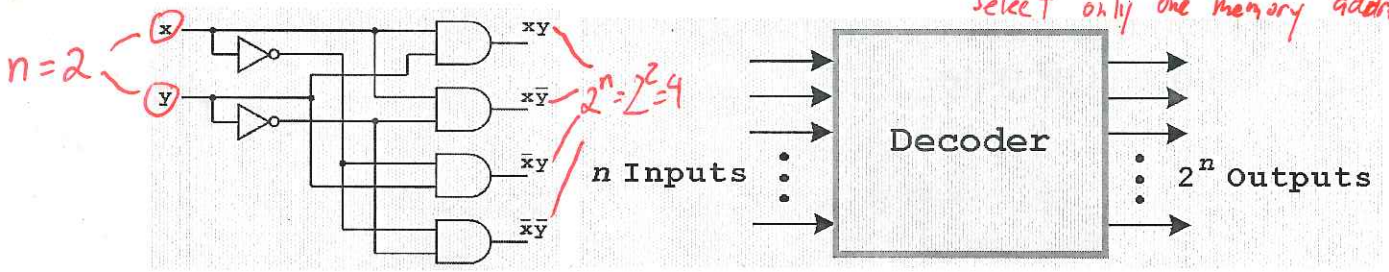


Today's systems employ more efficient adders.

Decoders – Combo of Gates

- Decoders are used to decode binary information from a set of n inputs to a maximum of 2^n outputs.

Decoder used by CPU to select only one memory address



- All memory address in a computer are specified as binary numbers.
- Decoders are used to select a memory location based on the binary values placed on the address lines of a memory bus.
- Address decoders with n inputs can select any of 2^n locations.

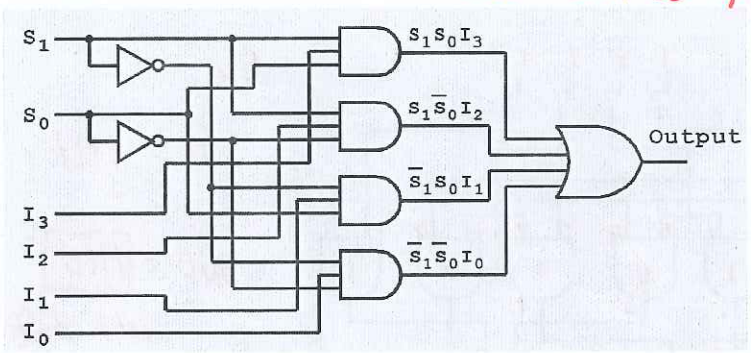
one out of many

Multiplexer

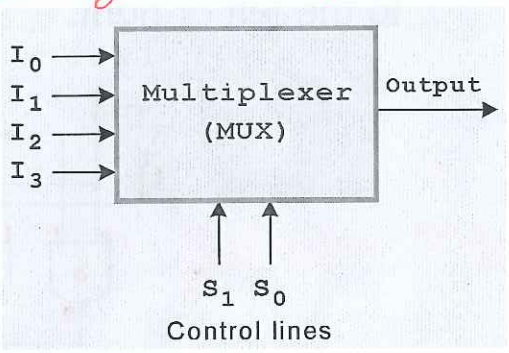
- A multiplexer does just the opposite of a decoder.
  - selects binary information from one of many input lines
  - direct it to a single output.
- At any given time, only one input is routed through the circuit to the output line. All other inputs are "cut-off".
- Selection of a particular input line is controlled by a set of selection variables, or control lines.
- To be able to select among n inputs, $\log_2 n$ control lines are needed.

Multiplexer

according to controler, the path for the selected data will be open to generate output



A look Inside a Multiplexer with four inputs and two control lines

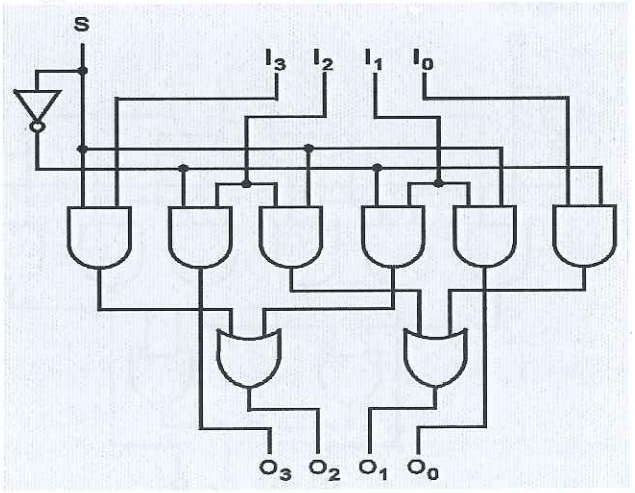


A Multiplexer Symbol

Select one input

Bit Shifting

- Bit Shifting moves the bits of a word or byte one position to the left or right.

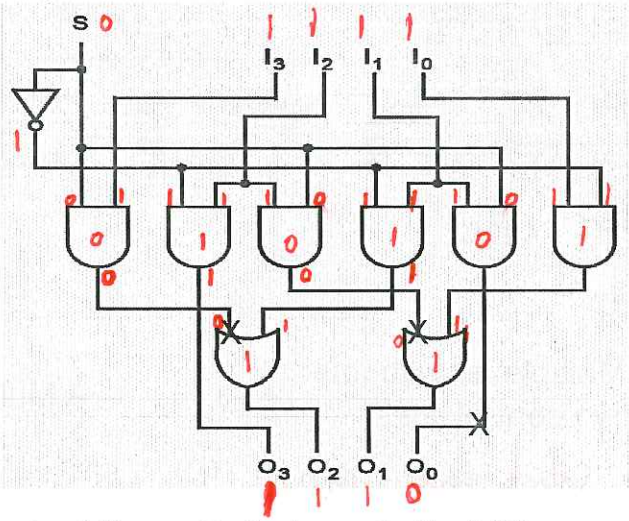


A 4-bit shifter:

- When the control line, S, is low, left shift occurs; when S is high, right shift occurs.

Bit Shifting - left

- Bit Shifting moves the bits of a word or byte one position to the left or right.



Ex

in = 1111

out = 1110

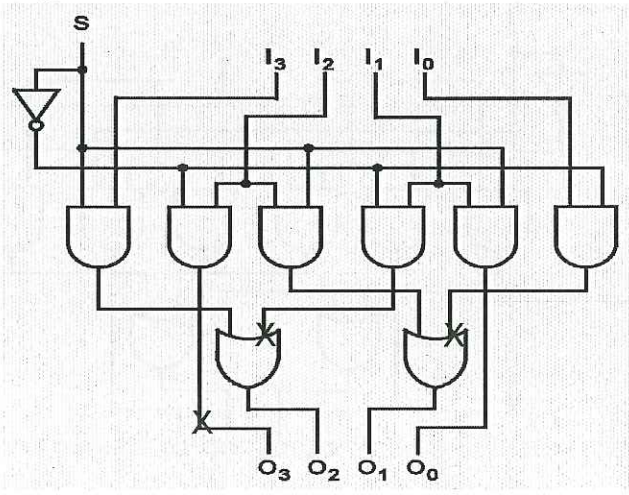
left shift

A 4-bit shifter:

- When the control line, S, is low, left shift occurs; when S is high, right shift occurs.

Bit Shifting - right

- Bit Shifting moves the bits of a word or byte one position to the left or right.



A 4-bit shifter:

- When the control line, S, is low, left shift occurs; when S is high, right shift occurs.

Lecture 10

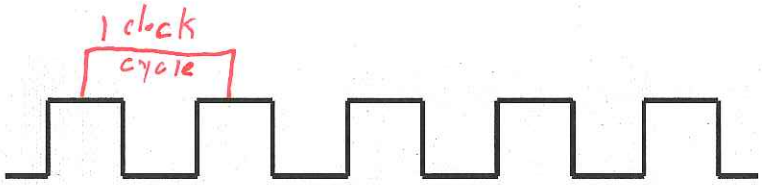
- Topics:
  - Sequential circuits
    - Basic concepts
    - Clocks
    - Flip-flops
      - SR Flip-Flop
      - JK flip-flops
      - D flip-flops
    - Finite state machines
      - Moore machines
      - Mealy machines
    - Examples of sequential circuits
      - Register
      - Binary Counter
      - memory

Sequential Circuits

- Combinational circuits are memory-less: depends sole on the values of the inputs to the Boolean functions.
- A sequential circuit defines its output as a function of both its current inputs and its previous inputs.
- To remember previous inputs, sequential circuits must have some sort of storage element, typically referred to as a flip-flop. The state of the flip-flop is a function of the previous inputs to the circuit.
- Sequential circuits: outputs depends both on current inputs and the current state of the circuit.
- Combinational circuits are generalizations of gates, sequential circuits are generalizations of flip-flop.

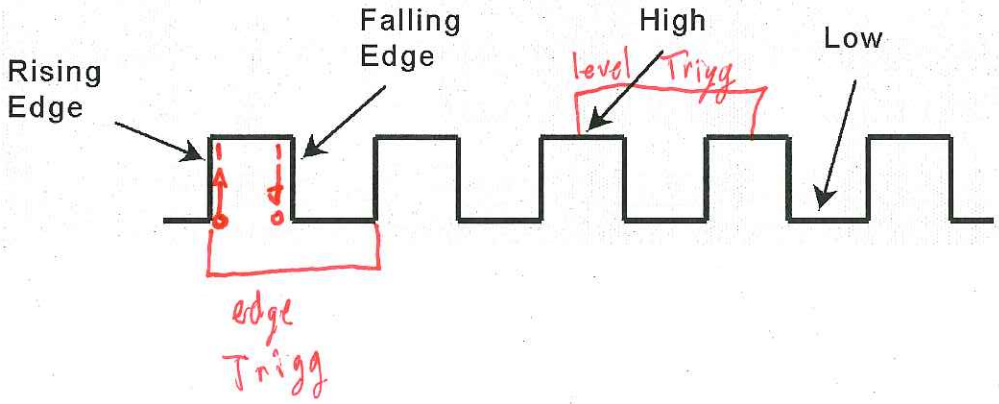
Clocks

- As the name implies, sequential logic circuits require a means by which events can be sequenced.
- State changes are controlled by clocks.
  - A "clock" is a special circuit that sends a series of pulses with a precise pulse width and a precise interval between consecutive pulses.
  - The interval between consecutive pulses is called the clock cycle time.
  - Clock speed is generally measured in megahertz or gigahertz.



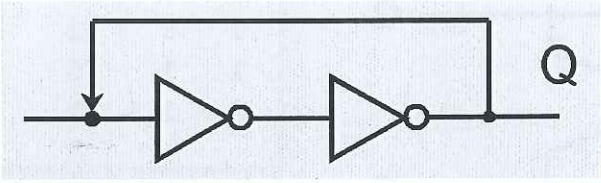
Clocks

- Inputs to the sequential circuit can only affect the storage element at given, discrete instances of time.
- Edge-triggered circuits change state on the rising edge, or falling edge of the clock pulse.
- Level-triggered circuits change state when the clock voltage reaches its highest or lowest level.



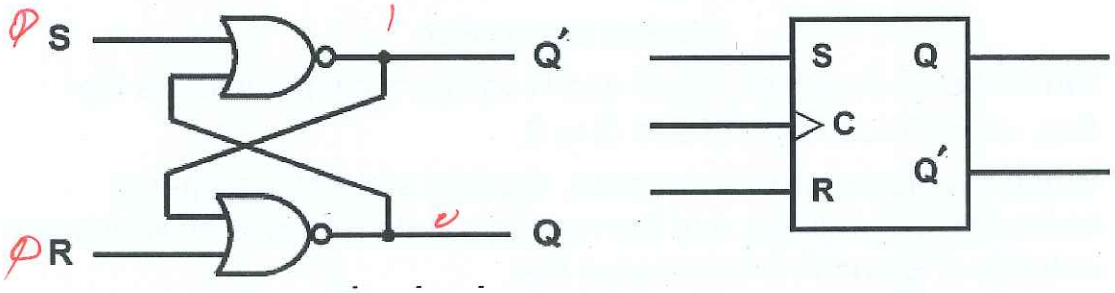
Feedback

- To retain the previous state values, sequential circuits rely on *feedback.*
- Feedback in digital circuits occurs when an output is looped back to the input.
- A simple example of this concept is shown below.
  - If Q is 0 it will always be 0, if it is 1, it will always be 1. (not useful, just describe the feedback concept)



SR Flip-Flop

- SR flip-flop: the most basic sequential logic components.
  - The "SR" stands for set/reset.
- The internals of an SR flip-flop are shown below, along with its block diagram.



How to get current Q if it depends on Q' input? = OR gates only need one state ?

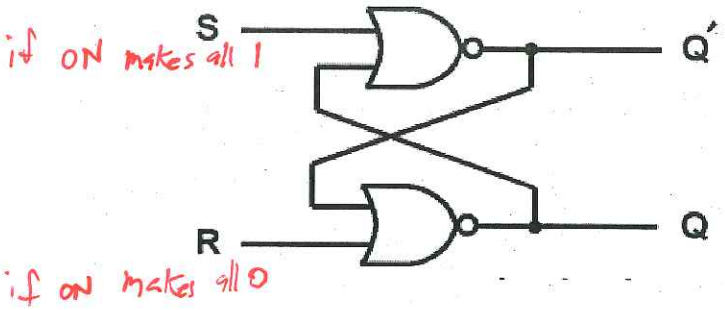
S - 0 > maintain current data
R - 0 >

SR Flip-Flop

- The SR flip-flop actually has three inputs: S, R, and its current output, Q.
- Notice the two undefined values. When both S and R are 1, the SR flip-flop is unstable.

Present State			Next State
S	R	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	undefined
1	1	1	undefined

same as initial Q
" " " "
" " " "
changed state Q

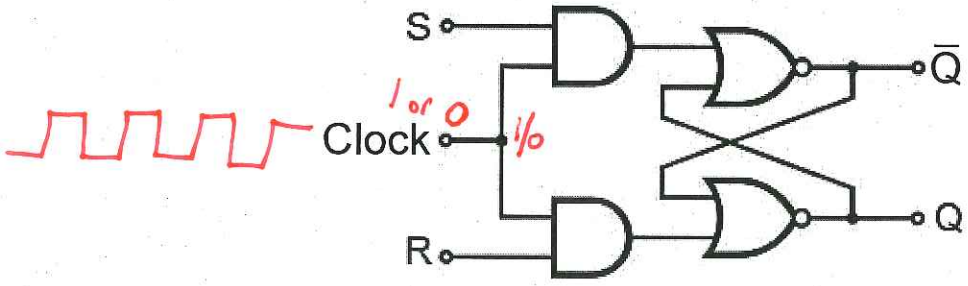


if ON makes all 1
if ON makes all 0

Reset high & set Low will reset

SR Flip-Flop

- What will happen when both S and R are 1?



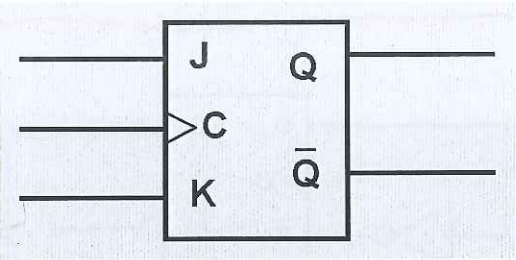
Clocked SR Flip-Flop

- When the clock pulses, the S and R values are input into the flip-flop, which forces both Q and $\bar{Q}$ to 0.
- When the clock pulse is removed, the final state of the flip-flop cannot be determined, and the resulting state depends on which one actually of S and R is terminated first.
- Therefore, S and R are not allowed set to 1 at the same time.

JK Flip-Flop

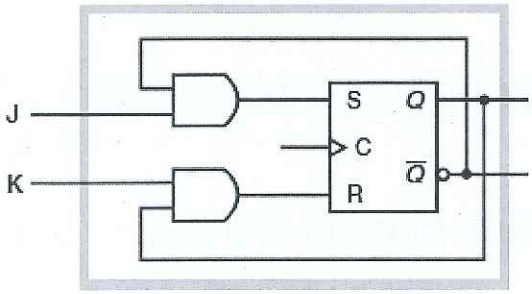
- S and R are not allowed set to 1 at the same time in SR flip-flop circuit, however this may not always be the case.
- The SR flip-flop can be modified to provide a stable state when both inputs are 1.

- This modified flip-flop is called a JK flip-flop, shown at the right.



JK Flip-Flop

- An SR flip-flop can be modified to create a JK flip-flop.
- The characteristic table indicates that the flip-flop is stable for all inputs.

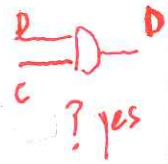


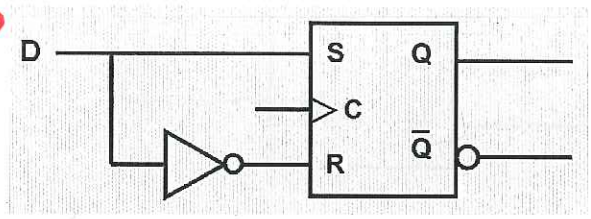
make S/R never have both ON state

S J	R K	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\overline{Q}$(t) – invert previous state

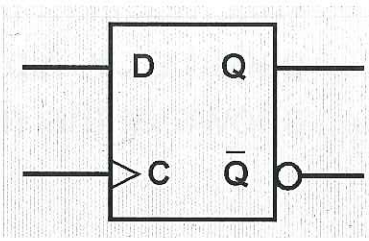
D Flip-Flop

- D(ata) flip-flop is another variant of the SR flip-flop.
- D flip-flop is a true representation of physical computer memory. This sequential circuit stores one bit of information.
- Note that the output of the flip-flop remains the same during subsequent clock pulses.
- The output changes only when the value of D changes.





D	Q(t+1)
0	0
1	1



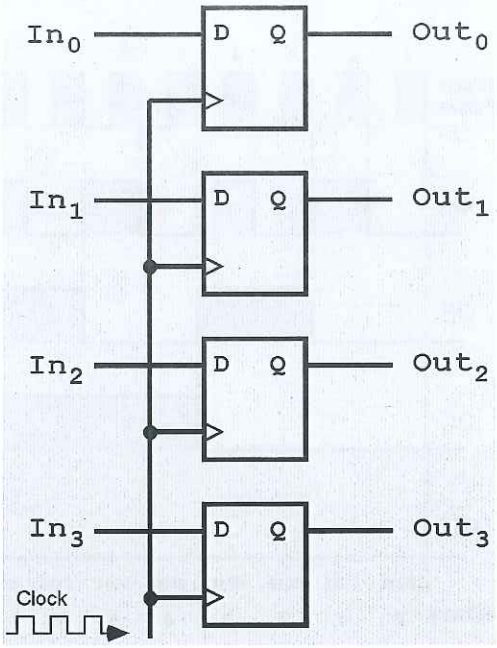
Examples of Sequential Circuits

- Sequential circuits are used anytime that we have a "stateful" application.
  - A stateful application is one where the next state of the machine depends on the current state of the machine and the input.
- A stateful application requires both combinational and sequential logic.
- Several examples of sequential circuits will be discussed.

Register

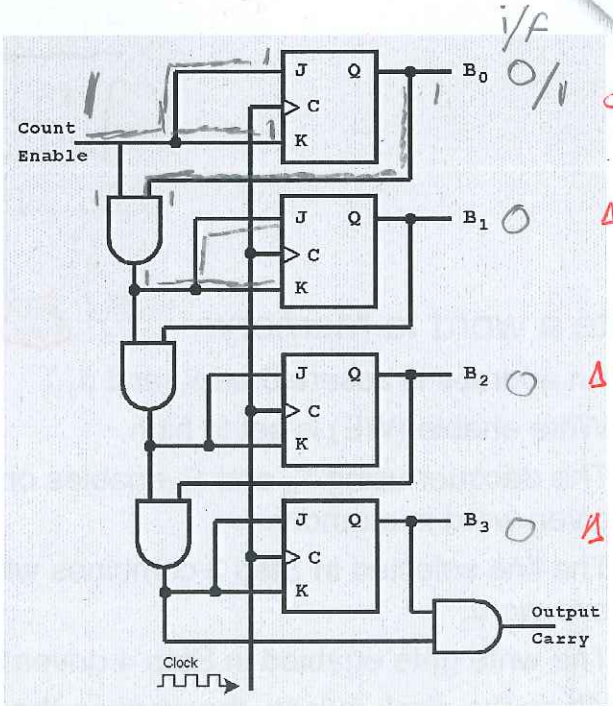
- This illustration shows a 4-bit register consisting of D flip-flops. You will usually see its block diagram (below) instead.

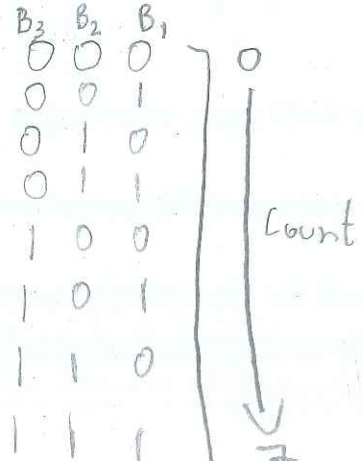
A larger memory configuration is shown on the next slide.



Binary Counter

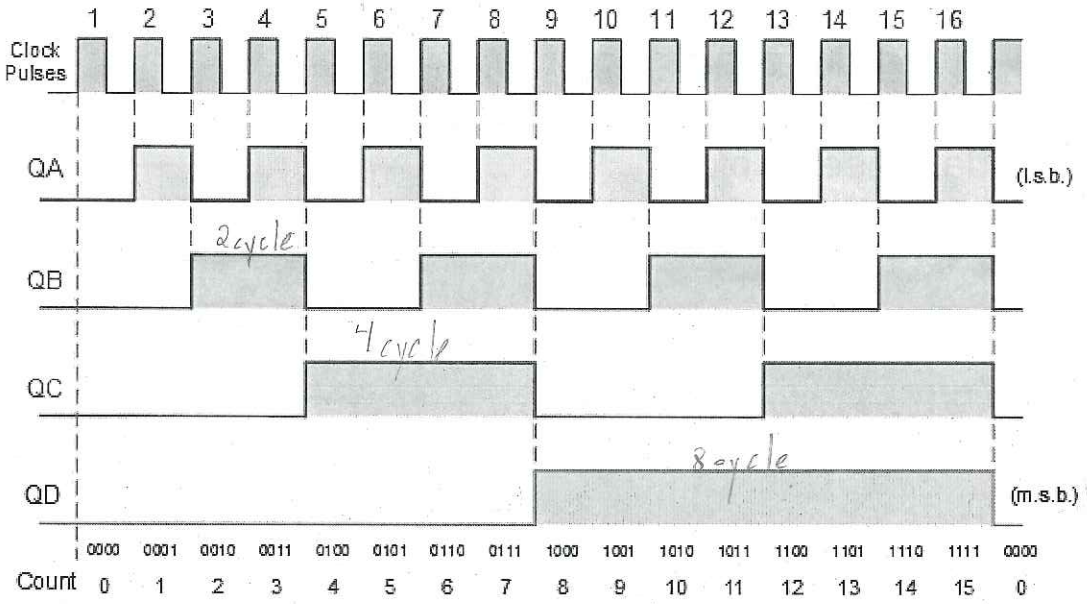
- A binary counter is another example of a sequential circuit.
- The low-order bit is complemented at each clock pulse.





Binary Counter

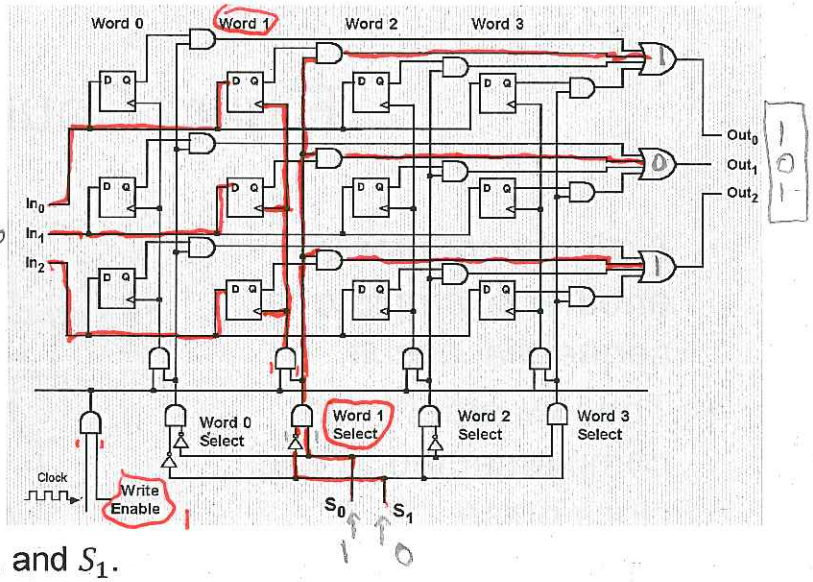
edge cycle



Memory

write here

data



Write a word to memory:

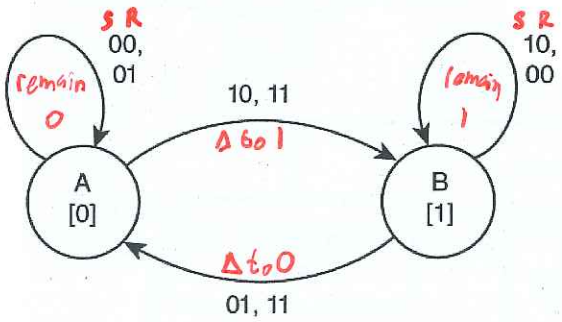
1. An address is asserted on S_0 and S_1.
2. Write enable (WE) is set to high.
3. The decoder using S_0 and S_1 enables only one AND gate, selecting a given word in memory.
4. The line selected in Step 3 combines with the clock and WE select only one word.
5. The write gate enabled in Step 4 drives the clock for the selected word.
6. When the clock pulses, the word on the input lines is loaded into the D flip-flop.

Finite State Machines

- Characteristic tables describe the behavior of flip-flops and sequential circuits.
- An equivalent graphical depiction is provided by finite state machines (FSMs).
  - FSMs consist of a set of nodes that hold the states of the machine and a set of arcs that connect the states.
- FSM is a system that visits a finite number of logically distinct states.
- Moore and Mealy machines are two types of FSMs that are equivalent.
  - Differ only in how they express the outputs of the machine.
- Moore machines place outputs on each node, while Mealy machines present their outputs on the transitions.

Moore Machine

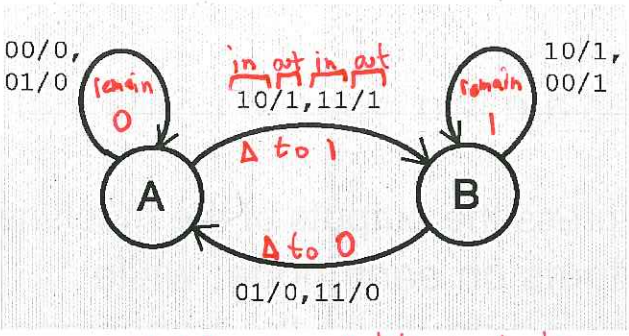
- A Moore machine
  - Each state is associated with the output of the machine.
  - Arcs illustrate the transitions between the states.
  - Outputs are a function of its current state.
- The behavior of a JK flop-flop is depicted below by a Moore machine.



J	K	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}$(t) - invert

Mealy Machine

- A Mealy machine
  - Each transition arc is labeled with its input and output separately by a slash.
  - Each transition is associated with the output of the machine.
  - Outputs are functions of its current sate and its input.
- The behavior of a JK flop-flop is depicted below by a Mealy machine.



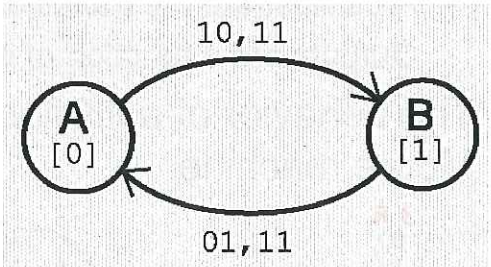
J	K	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}$(t) – invert

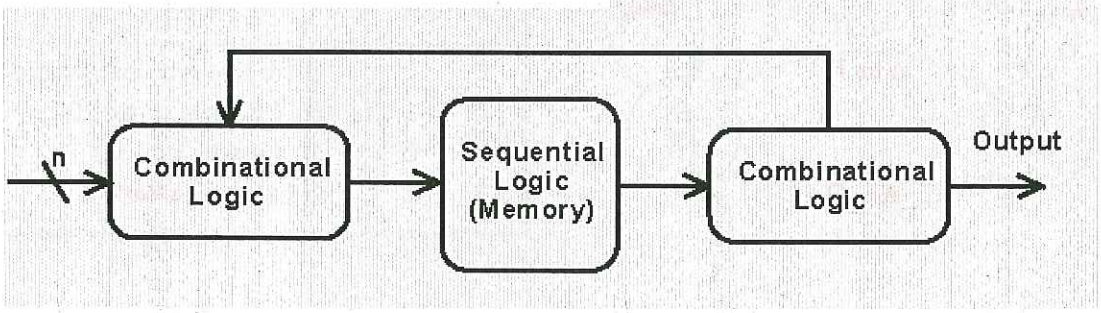
Ability to validate output to allow transition if it matches

Moore vs. Mealy machines

- Although the behavior of Moore and Mealy machines is identical, their implementations differ.

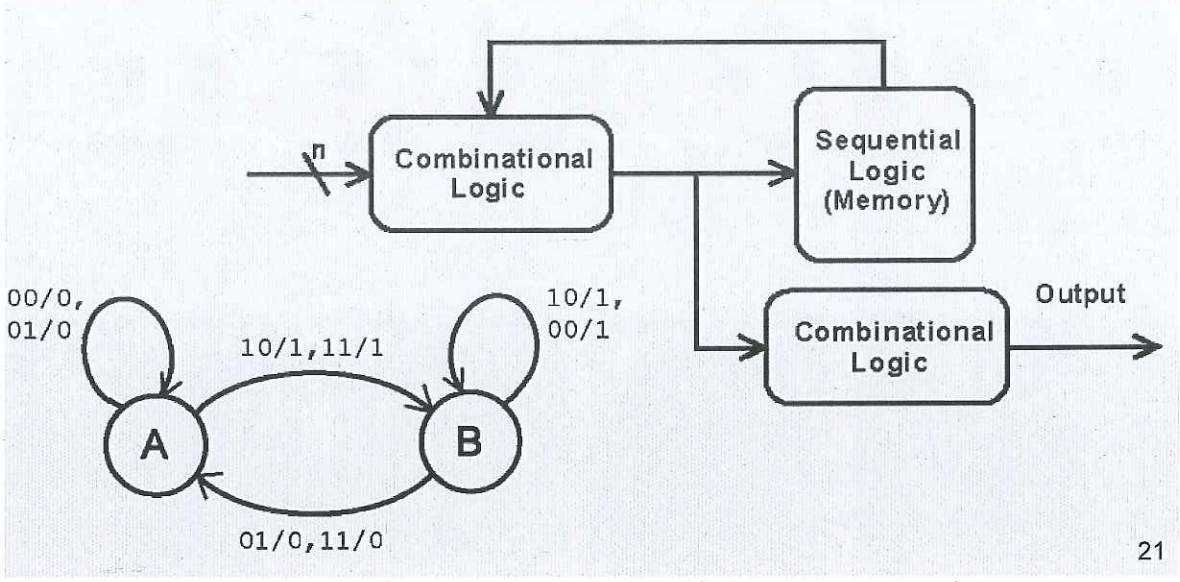
This is our Moore machine: outputs change sequentially with state change.





Moore vs. Mealy machines

- Although the behavior of Moore and Mealy machines is identical, their implementations differ.
- Mealy machine: output depends on its current state and inputs



Lecture 11

- Topics
  - Karnaugh Maps
    - Minterm
    - Rules of Kmap simplification
    - Don't Care Conditions

3A.1 Introduction

- Simplification of Boolean functions leads to simpler (and usually faster) digital circuits.
- Simplifying Boolean functions using identities is time-consuming and error-prone.
- This special section presents an easy, systematic method for reducing Boolean expressions through a graphical way of visualizing and then simplifying Boolean expressions = KMAP

3A.2 Description of Kmaps and Terminology

- A *minterm* is a product term that contains all of the function's variables exactly once, either complemented or not complemented.

like sum of product

For example, the minterms for a function having the inputs *x* and *y* are **x'y, x'y, xy'**, and **xy**.

X' = 0 Y' = 0
X = 1 Y = 1

Minterm	X	Y
X'Y'	0	0
X'Y	0	1
X Y'	1	0
X Y	1	1

3A.2 Description of Kmaps and Terminology

- Similarly, a function having three inputs, has the minterms that are shown in this diagram.

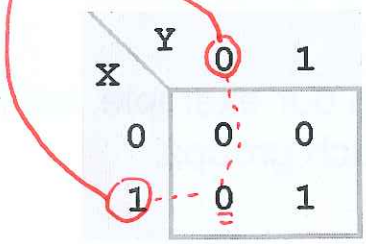
Minterm	X	Y	Z
X'Y'Z'	0	0	0
X'Y'Z	0	0	1
X'Y Z'	0	1	0
X'Y Z	0	1	1
X Y'Z'	1	0	0
X Y'Z	1	0	1
X Y Z'	1	1	0
X Y Z	1	1	1

3A.2 Description of Kmaps and Terminology

- A Kmap is a matrix consisting of rows and columns that represent the output values of a Boolean function.
- The output values placed in each cell are derived from the minterms of a Boolean function.
- A Kmap has a cell for each minterm.
- This means that it has a cell for each line for the truth table of a function.
- The truth table for the function *F(x,y) = xy* is shown at the right along with its corresponding Kmap.

F(X,Y) = XY

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1



3A.2 Description of Kmaps and Terminology

- As another example, we give the truth table and KMap for the function, *F(x,y) = x + y* at the right.
- This function is equivalent to the OR of all of the minterms that have a value of 1. Thus:

$$F(x, y) = x + y$$

$$= x'y + xy' + xy$$

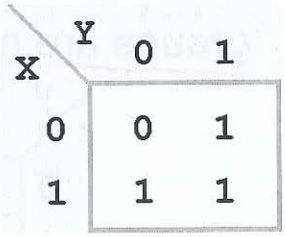
Find all combos of 1's

create truth table 1st

F(X,Y) = X+Y

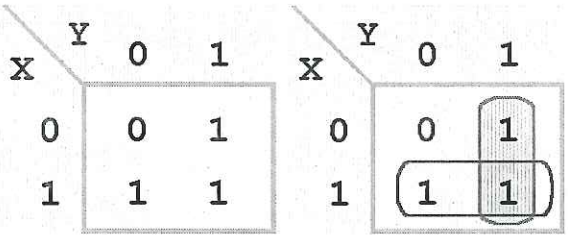
X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

Then



3A.3 Kmap Simplification for Two Variables

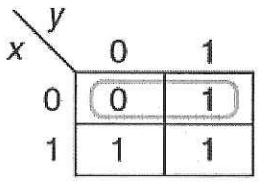
- Of course, the minterm function that we derived from our Kmap was not in simplest terms.
  - That's what we started with in this example.
- We can, however, reduce our complicated expression to its simplest terms by finding adjacent 1s in the Kmap that can be collected into groups that are powers of two.
- In our example, we have two such groups.



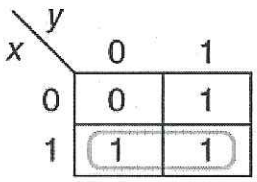
3A.3 Kmap Simplification for Two Variables

The rules of Kmap simplification are:

- Groupings can contain only 1s; no 0s.

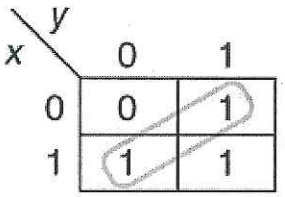


a) Incorrect

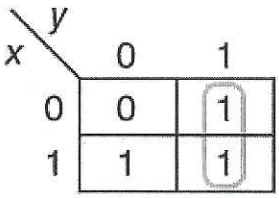


b) Correct

- Groups can be formed only at right angles; diagonal groups are not allowed.



a) Incorrect

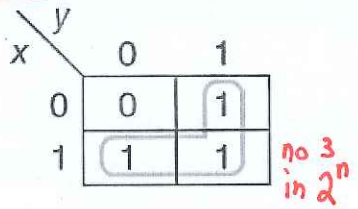


b) Correct

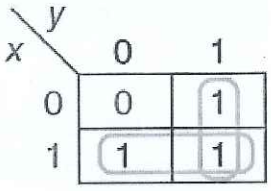
3A.3 Kmap Simplification for Two Variables

The rules of Kmap simplification are:

- The number of 1s in a group must be a power of 2, even if it contains a single 1.

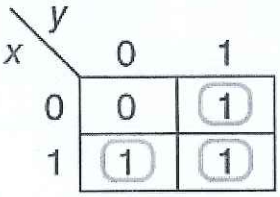


a) Incorrect

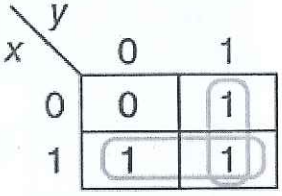


b) Correct

- The groups must be made as large as possible.



a) Incorrect



b) Correct

3A.3 Kmap Simplification for Two Variables

The rules of Kmap simplification are:

- Groups can overlap and wrap around the sides of the Kmap.
- Use don't care conditions when you can.

3A.3 Kmap Simplification for Three Variables

- A Kmap for three variables is constructed as shown in the diagram below.
- We have placed each minterm in the cell that will hold its value.
  - Notice that the values for the *yz* combination at the top of the matrix form a pattern that is not a normal binary sequence.

X \ YZ	00	01	11	10
0	0 pos 000 m_0=X'Y'Z'	1 m_1=X'Y'Z	3 m_3=X'YZ	2 m_2=X'YZ'
1	4 pos 100 m_4=XY'Z'	5 m_5=XY'Z	7 m_7=XYZ	6 m_6=XYZ'

3A.3 Kmap Simplification for Three Variables

- Thus, the first row of the Kmap contains all minterms where *x* has a value of zero.
- The first column contains all minterms where *y* and *z* both have a value of zero.

X \ YZ	00	01	11	10
0	0 m_0=X'Y'Z'	1 m_1=X'Y'Z	3 m_3=X'YZ	2 m_2=X'YZ'
1	4 m_4=XY'Z'	5 m_5=XY'Z	7 m_7=XYZ	6 m_6=XYZ'

3A.3 Kmap Simplification for Three Variables

- Consider the function:

$$F(X,Y,Z) = X'Y'Z + X'YZ + XY'Z + XYZ$$

- Its Kmap is given below.
  - What is the largest group of 1s that is a power of 2?

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

3A.3 Kmap Simplification for Three Variables

- This grouping tells us that changes in the variables *x* and *y* have no influence upon the value of the function: They are irrelevant.
- This means that the function,

$$F(X,Y,Z) = X'Y'Z + X'YZ + XY'Z + XYZ$$

reduces to *F(x)* = *z*.

You could verify this reduction with identities or a truth table.

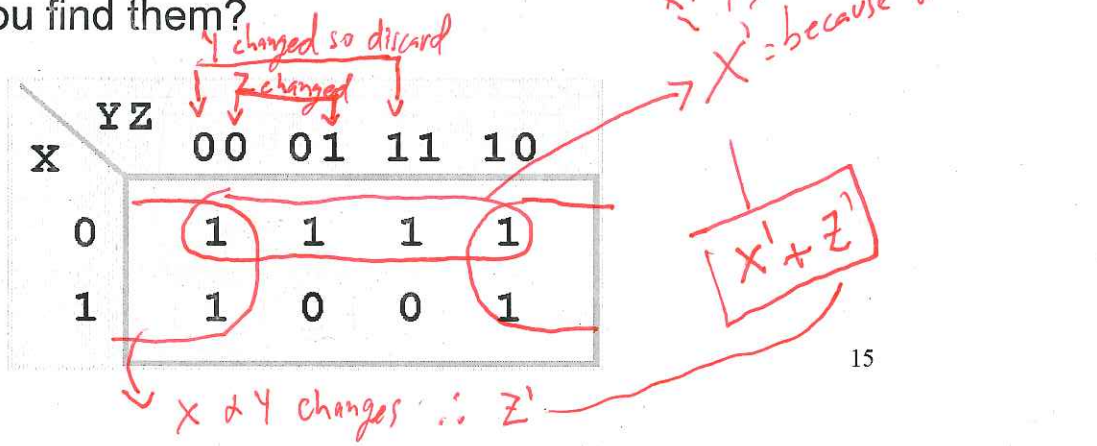
X \ YZ	00	01	11	10
0	0	1	1	0
1	0	1	1	0

3A.3 Kmap Simplification for Three Variables

- Now for a more complicated Kmap. Consider the function:

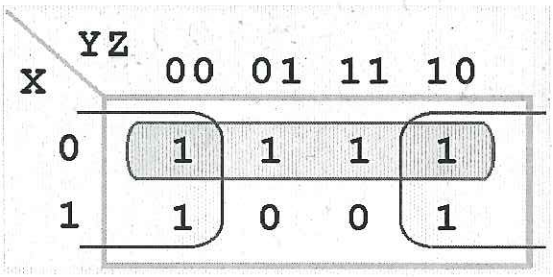
$$F(X,Y,Z) = X'Y'Z' + X'Y'Z + X'YZ + X'YZ' + XY'Z' + XYZ'$$

- Its Kmap is shown below. There are (only) two groupings of 1s.
  - Can you find them?



3A.3 Kmap Simplification for Three Variables

- The green group in the top row tells us that only the value of X' is significant in that group.
- We see that it is complemented in that row, so the other term of the reduced function is X'.
- Our reduced function is $F(X,Y,Z) = X' + Z'$



3A.3 Kmap Simplification for Four Variables

- Our model can be extended to accommodate the 16 minterms that are produced by a four-input function.
- This is the format for a 16-minterm Kmap:

WX \ YZ	00	01	11	10
00	0 m_0=W'X'Y'Z'	1 m_1=W'X'Y'Z	3 m_3=W'X'YZ	2 m_2=W'X'YZ'
01	4 m_4=W'XY'Z'	5 m_5=W'XY'Z	7 m_7=W'XYZ	6 m_6=W'XYZ'
11	12 m_{12}=WXY'Z'	13 m_{13}=WXY'Z	15 m_{15}=WXYZ	14 m_{14}=WXYZ'
10	8 m_8=WX'Y'Z'	9 m_9=WX'Y'Z	11 m_{11}=WX'YZ	10 m_{10}=WX'YZ'

how to create this matrix? Also page 12

3A.3 Kmap Simplification for Four Variables

- We have populated the Kmap shown below with the nonzero minterms from the function:

$$F(W,X,Y,Z) = W'X'Y'Z' + W'X'Y'Z + W'X'YZ' + W'XYZ' + WX'Y'Z' + WX'Y'Z + WX'YZ'$$

  - Can you identify (only) three groups in this Kmap?

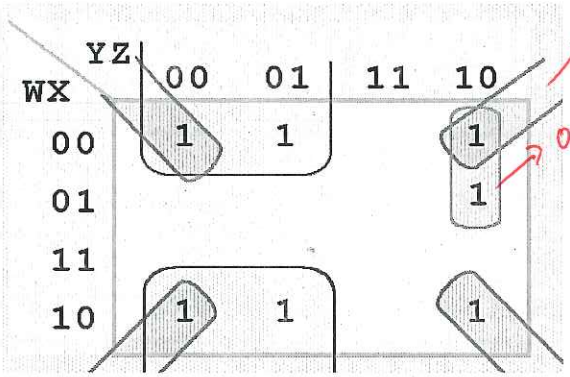
Recall that groups can overlap.

WX \ YZ	00	01	11	10
00	1	1		1
01				1
11				
10	1	1		1

3A.3 Kmap Simplification for Four Variables

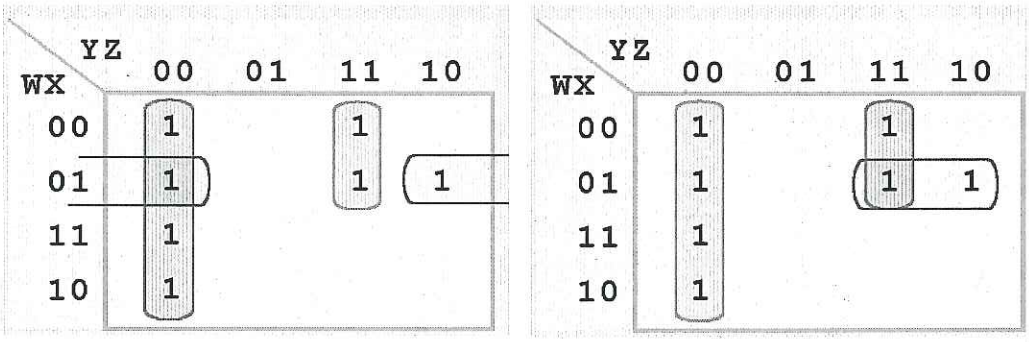
- Our three groups consist of:
  - A purple group entirely within the Kmap at the right.
  - A pink group that wraps the top and bottom.
  - A green group that spans the corners.
- Thus we have three terms in our final function:

$$F(W,X,Y,Z) = X'Y' + X'Z' + W'YZ'$$



3A.3 Kmap Simplification for Four Variables

- It is possible to have a choice as to how to pick groups within a Kmap, while keeping the groups as large as possible.
- The (different) functions that result from the groupings below are logically equivalent.



3A.6 Don't Care Conditions

- Real circuits don't always need to have an output defined for every possible input.
  - For example, some calculator displays consist of 7-segment LEDs. These LEDs can display $2^7 - 1$ patterns, but only ten of them are useful.
- If a circuit is designed so that a particular set of inputs can never happen, we call this set of inputs a *don't care* condition.
- They are very helpful to us in Kmap circuit simplification.

3A.6 Don't Care Conditions

- In a Kmap, a don't care condition is identified by an *X* in the cell of the minterm(s) for the don't care inputs, as shown below.
- In performing the simplification, we are free to include or ignore the *X*'s when creating our groups.

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

3A.6 Don't Care Conditions

- In one grouping in the Kmap below, we have the function:

$$F(W,X,Y,Z) = W'X' + YZ$$

Still group the don't care conditions

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

3A.6 Don't Care Conditions

- A different grouping gives us the function: but logically same

$$F(W,X,Y,Z) = W'Z + YZ$$

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

3A.6 Don't Care Conditions

- The truth table of:

$$F(W,X,Y,Z) = W'X' + YZ$$

differs from the truth table of:

$$F(W,X,Y,Z) = W'Z + YZ$$

- However, the values for which they differ, are the inputs for which we have don't care conditions.

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

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

In Class Exercise

Write a simplified expression for the Boolean function defined by each of the following Kmaps.

wx \ yz	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	0	0	1	0
10	1	0	1	0

→ w'z'

f(w,x,y,z) = wyz + x'y'z' + w'z'

x'y'z'

wyz

Lecture 12

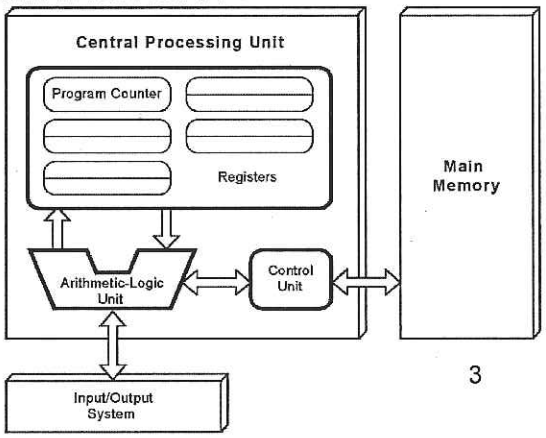
- Today's topics
  - CPU basics
    - Registers
    - ALU
    - Control Unit
  - The bus
  - Clocks
  - Input/output subsystem

Introduction

- Chapter 1 presented a general overview of computer systems.
- In Chapter 2, we discussed how data is stored and manipulated by various computer system components.
- Chapter 3 described the fundamental components of digital circuits.
- Having this background, we can now understand how computer components work, and how they fit together to create useful computer systems.

CPU Basics

- The computer's CPU fetches, decodes, and executes program instructions.
- Two principal parts of the CPU
  - Datapath: consists of an arithmetic-logic unit (ALU) and storage units (registers) that are interconnected by a data bus that is also connected to main memory.
  - Control unit: Responsible for sequencing operations and ensuring the correct data stay at the right location at the right time.

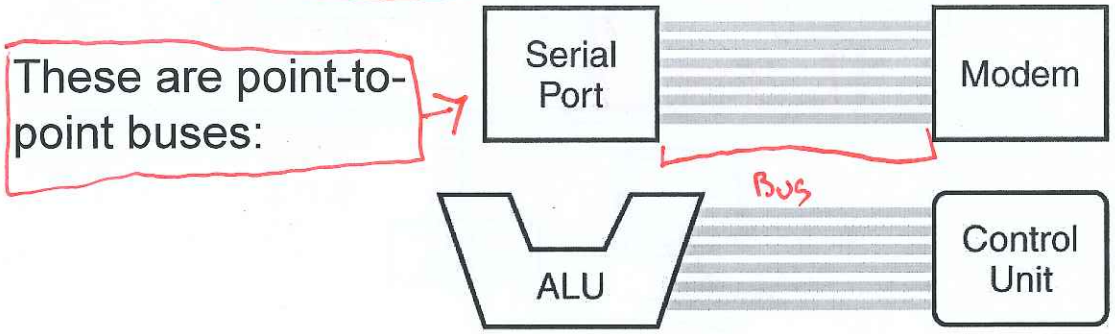


CPU Basics

- Registers hold data that can be readily accessed by the CPU.
- They can be implemented using D flip-flops.
  - A 32-bit register requires 32 D flip-flops.
- The arithmetic-logic unit (ALU) carries out logical and arithmetic operations as directed by the control unit.
- The control unit determines which actions to carry out according to the values in a program counter register and a status register.
  - Program counter: the next instructions for execution
  - Status register: keep track of overflows, carries, borrows, and etc.

The Bus

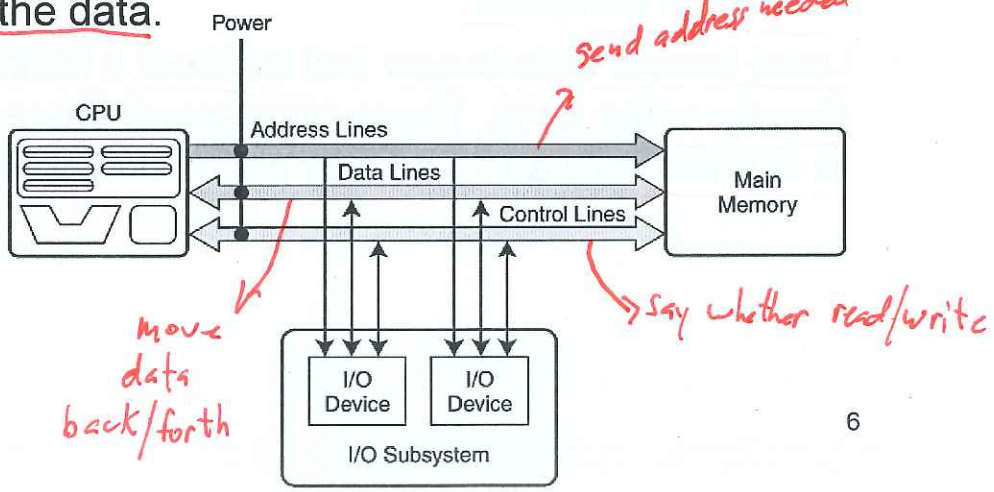
- The CPU shares data with other system components by way of a data bus.
  - A bus is a set of wires that simultaneously convey a single bit along each line.
- Two types of buses are commonly found in computer systems: *point-to-point*, and *multipoint* buses.



The Bus

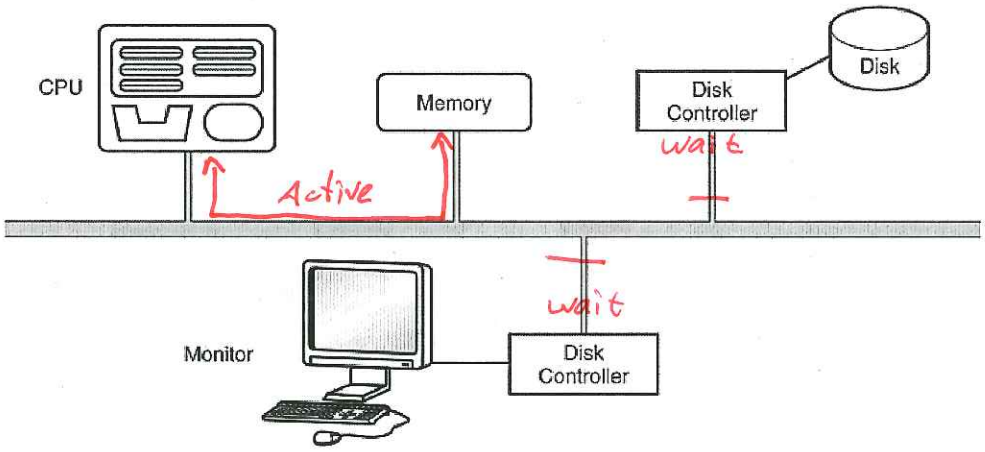
- Buses consist of data lines, control lines, and address lines.
- While the data lines convey bits from one device to another, control lines determine the direction of data flow, and when each device can access the bus.
- Address lines determine the location of the source or destination of the data.

* Know there are 3 buses for Xfer



The Bus

- A multipoint bus is shown below.
- Because a multipoint bus is a shared resource, access to it is controlled through protocols (bus protocol), which are built into the hardware.



Bus Terminologies

Personal computers' terminologies

crucial bus *

- System bus: an internal bus, connecting the CPU, memory and all other internal components.
- Expansion buses: external buses, connecting external devices, peripherals, expansion slots, and I/O ports to the rest of the computer.
- Local buses: data buses that connect a peripheral device directly to the CPU. These high-speed buses can be used to connect only a limited number of similar devices.

Synchronous vs. Asynchronous Buses

Buses are physically little more than bunches of wires, they have specific standards for connectors, timing, and signaling specifications and exact protocols for use.

- Synchronous buses: a sequence of events is controlled by the clock and every device is synchronized by the clock rate.
- Asynchronous buses: control lines coordinate the operations through a handshaking protocol to enforce timing.
  - Bus masters are devices that are allowed to initiate transfer of information (control bus)
  - Bus slaves: modules that are activated by a master and respond to requests to read and write data.
  - Both follow a communication protocol to use the bus, working within very specific timing requirements.

Bus Arbitration

In a master-slave configuration, where more than one device can be the bus master, concurrent bus master requests must be arbitrated.

Four categories of bus arbitration are:

- **Daisy chain:** Permissions are passed from the highest-priority device to the lowest.
  - Simple, not fair.
- **Centralized parallel:** Each device is directly connected to an arbitration circuit.
  - Bottlenecks at the arbitration device.
- **Distributed using self-detection:** Devices decide which gets the bus among themselves.
- **Distributed using collision-detection:** Any device can try to use the bus. If its data collides with the data of another device, it tries again.

Clocks

- Every computer contains at least one clock that synchronizes the activities of its components.
- A fixed number of clock cycles are required to carry out each data movement or computational operation.
- The clock frequency, measured in megahertz or gigahertz, determines the speed with which all operations are carried out.
- Clock cycle time is the reciprocal of clock frequency.
  - An 800 MHz clock has a cycle time of 1.25 ns.

CPU Time

- Clock speed should not be confused with CPU performance.
- The CPU time required to run a program is given by the general performance equation:

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- CPU throughput can be improved by reducing the number of instructions in a program, reducing the number of cycles per instruction, or reduce the number of nanoseconds per clock cycle.

The Input/Output Subsystem

- A computer communicates with the outside world through its input/output (I/O) subsystem.
- I/O devices connect to the CPU through various interfaces,
- Interfaces handle the data transfers:
  - Memory-mapped I/O: the registers in the interface appear in the computer's memory map, and there is no real difference between accessing memory and accessing an I/O device.
    - Advantage in access-speed, drawback in requiring memory space. Not Good
  - Instruction-based I/O: the CPU has a specialized I/O instructions.
    - Not use memory space, but requires specific I/O instruction. Better

We study I/O in detail in chapter 7.