CSCI 350

Shefali Emmanuel

Shefali Emmanuel

big deh

The College of Charleston
CSCI 350 Digital Logic and Computer Organization
Spring 2020

Instructor	George J. Pothering (potheringg@cofc.edu)
Office	Room 327, Harbor Walk East (HWE)
Office Hours	MW 3:30 - 4:30 pm, Room 327 Harbor Walk East Other hours by appointment
Telephone	843-953-8156 (office)

Section Meetings TR 8:30am – 9:45 am in Room 300 of Harbor Walk East (Section 1)
TR 9:55am – 11:10 am in Room 301 of Harbor Walk East (Section 2)

Course Description: This course introduces the formal study of programming language specifications and develops an understanding of the structure and run-time organization of imperative programming languages. Topics include data types, control structures, functional programming, logic programming, procedure mechanisms, and data abstraction.
Prerequisite: CSCI 250 with a grade of C- or better; *Co-requisite or prerequisite*: MATH 207.

Textbook Tanenbaum. Andrew and Austin, Todd. *Structured Computer Organization* (6th ed)
Pearson Education, Inc. 2013. ISBNs:0-13-291652-5 or 978-0-13-291652-3.

Web Page The class will use resources available through Oaks

Important Dates for This Course:

January 9	First class
March 17 – 23	Spring Break
April 18	Last day of class
April 25	Final Exam 8:00 am – 11:00 am Section 2; [12:00 pm – 3:00 pm Section 1]
May 6	Final grades due at noon.

Class Policies

I. ***Missed Classes***: Attendance will be taken each day for the first few classes only after which no record of attendance will be kept. You are responsible for all work done or required for any class that you miss, i.e. doing the preparatory work for class, getting notes, turning in assignments, etc. On average you should expect to work six to nine hours *outside of class* each week on this course.

II. ***Exams:*** You will take two semester exams and a comprehensive final examination. The dates of the semester exams are:

Thursday, February 20
Thursday, April 9

The final examination is currently scheduled for:

Saturday, April 25 from 12:00 pm to 3:00 pm (Section 1).
Saturday, April 25 from 8:00 am to 11:00 am (Section 2).

You may also be given quizzes from time-to-time on material I will ask you to read, but will not cover in class. Results of these quizzes will be incorporated into your grade for the next test that you take. No make-ups will be given for missed quizzes. If you miss a quiz for any reason, you will be required to write a 500-word paper on the material covered on the quiz. This will be due by the class after the one of the missed quiz.

Lab Work — has to scan the sheet NOT phone pictures

III. ***Assignments, etc.:*** Written homework or programming assignments will be issued throughout the semester and a due date will be associated with the assignment. Unless otherwise indicated, your work on these assignments must be submitted via OAKS. After the due date solutions to the assignment will be posted on OAKS.

The solutions will give you an opportunity to compare your work to what I have posted and to see me if there is any clarification or other related question you want to ask me. I will examine your effort on a problem or group of problems on each assignment and evaluate it on a 0-3 scale, and also give you a rating for assignment overall. The ratings should be interpreted as follows:

3	You made a serious attempt at all of the problems and successfully completed the vast majority of them. There may be minor errors in some of the problems.
2	You attempted all of the problems, but showed some significant misunderstandings or errors in some of them. If I had assigned point values to these problems showing misunderstanding you would probably have received no more than 60% of the points on them. All of the attempts clearly reflect your own work, however.
1	For some of the problems you either did not attempt them, gave minimal effort, or appear to have copied some, or all, of the solution from the Internet or from the textbook's solutions manual.
0	You did not submit anything for the assignment.

I will not post the solutions earlier than 11:59pm on the assignment due date You have until I actually post the solutions to submit your work, however, but no work will be accepted after the solutions have been posted. If you do not understand something about a posted solution you may come to see me in my office and I will gladly explain anything you do not understand. **I will not explain the solutions via e-mail.** If it seems appropriate, I will also try to find a time and a room where I can work them on the board in detail for the class as a whole.

At the end of the semester I will take the accumulated results of your efforts and incorporate them into a score that will count as ½ of a test toward determining you semester grade.

IV. ***Grading Scale:*** I do not use a strict grading scale such as the 10-point scale (90-100, 80-89, etc) commonly used by other instructors Instead, scales derived from clusters of student performance will be given for each test. Your status after every test can be determined by adding the boundaries of the individual scales and finding where your accumulated scores lie within these intervals. The exact way this works will become clear after the first test.

V. ***Course Grades:*** Students who complete every assignment and are awarded an effort score of 2 or more for each of those efforts will have their best exam grade counted again in determining their course grade. Your efforts in the course will be weighted as follows:

Semester Exams	2/9 each	(2/11 each for those who qualify)
Final Examination	4/9	(4/11 for those who qualify)
Assignments	1/9	(1/11 for those who qualify)
Best Test/Exam Grade		(2/11 for those who qualify)

do all of the assignments w/ 2 or 3's

VI. ***Classroom Behavior***: To maintain a classroom environment that is conducive to learning, I expect certain behavior of students in my classes. Students who, during class, check and send e-mail or text messages, browse the Web, Tweet, giggle, sleep, yawn audibly, whisper, whine, groan, arrive late, leave early, or come unprepared are disrupting to me and detract from their fellow students' learning experiences. At the end of the semester you may be just below the cut-off for a higher grade. If the impression of you that comes to mind is of

someone who is uninterested, disruptive, rude or otherwise lacking in classroom etiquette and deportment, do you really think anyone is going to reward you for this?

VII. ***Honor Code and Academic Integrity:*** Lying, cheating, attempted cheating, and plagiarism are violations of our Honor Code that, when identified, are investigated. Each instance is examined to determine the degree of deception involved.

Incidents where the professor believes the student's actions are clearly related more to ignorance, miscommunication, or uncertainty, can be addressed by consultation with the student. We will craft a written resolution designed to help prevent the student from repeating the error in the future. The resolution, submitted by form and signed by both the professor and the student, is forwarded to the Dean of Students and remains on file.

Cases of suspected academic dishonesty will be reported directly to the Dean of Students. **A student found responsible for academic dishonesty will receive a XF in the course, indicating failure of the course due to academic dishonesty. This grade will appear on the student's transcript for two years after which the student may petition for the X to be expunged.** The student may also be placed on disciplinary probation, suspended (temporary removal) or expelled (permanent removal) from the College by the Honor Board.

It is important for students to remember that unauthorized collaboration--working together without permission--is a form of cheating. Unless a professor specifies that students can work together on an assignment and/or test, no collaboration is permitted. Other forms of cheating include possessing or using an unauthorized study aid (such as a PDA), copying from another's exam, fabricating data, and giving unauthorized assistance.

Remember, research conducted and/or papers written for other classes cannot be used in whole or in part for any assignment in this class without obtaining prior permission from the professor.

Students can find a complete version of the Honor Code and all related processes in the Student Handbook at http://www.cofc.edu/studentaffairs/general_info/studenthandbook.html.

VIII. ***Disability:*** Accommodation: Any student who feels he or she may need an accommodation based on the impact of a disability should contact me individually to discuss your specific needs. Also, please contact the College of Charleston, Center for Disability Services http://www.cofc.edu/~cds/ for additional help.

Shefali Emmanuel
Jan 9, 2020

CHAPTER 1 DIGITAL LOGIC

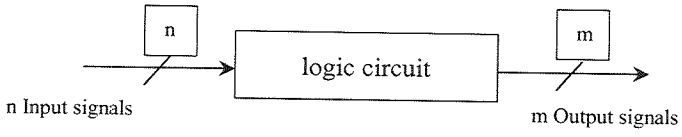
Section 1. Introduction -- Logic Circuits

The electronics inside a modern computer are comprised of circuits in which only two voltage levels are of interest: a high voltage (typically in the range 3 – 5 volts) and a low voltage (typically 0 – 1.5 volts). Rather than refer to the voltage levels, however, we talk about *signals* that are (logically) true, or 1, or are *asserted*; or signals that are (logically) false, or 0, or are *deasserted*. Because of the association of digit values with these various voltage levels such circuits are known as *digital circuits*. The further adaptation of the terms "true" and "false" from logic has resulted in such circuits being referred to as ***logic circuits***.

When we associate the value *true/1/asserted* with a high voltage and the value *false/0/deasserted* with a low voltage we employ what is known as a ***positive logic***-based system. This is the system we shall assume in this course. The values *false/0/deasserted* and *true/1/asserted* are ***complements*** of one another.

everything we do

In their most abstract form, we can regard a digital circuit as one that accepts one or more digital values as input values (or input signals) from an external source and produces one or more output values (or output signals). We can illustrate such a circuit abstractly as follows:



input will give you the same output

Logic circuits are classified as being either *combinational* circuits or *sequential* circuits. In ***combinational circuits*** the output values are uniquely determined by the input values. ***Sequential circuits***, on the other hand, contain memory elements that can store a 0 or 1, and which collectively comprise the ***state*** of the sequential circuits. Unlike combinational circuits the output values are uniquely termined by both the input values and the current state of the circuit. Moreover the input values and the current state together iquely determine a, possibly different, *next state* for the circuit.

input can give you diff output every time bc its like a memory box

Section 2. Truth Tables, Logic Functions and Logical Expressions

Because the output values for a combinational circuit are uniquely determined by the input values, when there are a small number of input signals it is possible to represent the associations between input and output values using a ***truth table***, a structure once again adopted from logic.

Examples: Examples of truth tables for representing combinational circuits with one output signal, X, and 2, 3 and 4 inputs respectively.

Done in class

Example: A 1-bit, ***half adder*** is a combinational circuit with two input values A and B representing 1-bit values to be added, and two output values ***sum*** and ***carry*** representing the sum and carry values from doing binary addition on the values of A and B. Represent the input-output associations with a truth table.

Done in class

Example: A 1-bit, ***full adder*** is a combinational circuit with three input values A, B and carry-in (C_{in}) representing 1-bit values to be added together with a third value representing a value to be carried into the addition, and two output values ***sum*** and ***carry-out*** (C_{out}) once again representing the sum and carry values from doing binary addition on the values of A, B and C_{in}. Represent the input-output associations with a truth table.

Done in class

Logic Functions

ıth tables work fine for describing combinational circuits when there are a small number of input signals, say 2, 3 or 4, but beyond ıis the truth tables have too many rows. An alternative approach for representing combinational circuits is to regard the various

patterns of input signal associations as binary numbers and to give the binary numbers of only those input combinations that have the value 1 associated with them, the understanding being that the entries not listed have the value 0 associated with them

	A	B	f
0	0	0	**1**
1	0	1	0
2	1	0	0
3	1	1	**1**

Represented as $f(A,B) = \Sigma(0,3)$

Of course there is no particular reason for favoring those entries that have ones, so with a slight adjustment in notation, we can show those entries that have the value 0 associated with them. For the above case this would be

$$f(A,B) = \Pi(1,2)$$

It is imperative that one show the number of input variables since otherwise we have no way of knowing how many rows are in the table being represented.

Examples of More Logic Functions:

Done in class ended on Jan 9

Logical (Boolean) Expressions

Another approach is to express the logic function with logic expressions. This is done with the use of notation and language borrowed *Boolean algebra* (named after George Boole, a 19th-century mathematician), and hence the resulting expressions are sometimes known as ***Boolean expressions.***

In Boolean algebra, all the variables have the values 0 or 1 and, in typical formulations, there are three operators:

- The OR operator, written as +, as in $A + B$, yields a result of 1 if the value of either (or both) of the variables is 1; otherwise the result is 0. The OR operation is also called a *logical sum*. We can represent the OR operations via the truth table

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

F is only when $0 \wedge 0$

- The AND operator, written as × as in $A \times B$ has result of 1 only if both inputs are 1; otherwise the result is 0. The AND operator is also called *logical product*. We can represent the AND operation via the truth table

A	B	A×B
0	0	0
0	1	0
1	0	0
1	1	1

T is only when $1 \wedge 1$

Alternatively a dot is sometimes used for logical AND as in A·B, and even simple adjacency positioning when there is no danger of confusion and thus not using an operator symbol, whence instead of A × B or A·B one would simply write AB.

* alternate the 2 values for the last one — then square

Ch 1: Digital Logic

Jan 9, 2020

Sec 2

Ex 1

circuits
- 0's
- 1's

voltage: low / high

2 input table

A	B	X
1	1	
1	0	
0	1	
0	0	

3 input table

A	B	C	X
1	1	0	
1	1	1	
1	0	1	
1	0	0	
0	1	1	
0	1	0	
0	0	1	
0	0	0	

4 input table

hex-a-decimal	A	B	C	D	X
0	0	0	0	0	
1	0	0	0	1	
2	0	0	1	0	
3	0	0	1	1	
4	0	1	0	0	
5	0	1	0	1	
6	0	[redacted]	1	0	
7	0	[redacted]	1	1	
8	1	0	0	0	
9	1	0	0	1	
10	1	0	1	0	
11	1	0	1	1	
12	1	[redacted]	0	0	
13	1	[redacted]	0	1	
14	1	[redacted]	1	0	
15	1	[redacted]	1	1	

Ex 2 half adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$Sum(A,B) = \sum(1,2)$

$\prod(0,3)$

Ex 3 full adder

Truth Table

	A	B	C_{in}	C_{out}	Sum
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

$C_{out}(A,B,C_{in}) = \sum(3,5,6,7)$ (where 1's are)

$Sum(A,B,C_{in}) = \sum(1,2,4,7)$ ← same thing

$\prod(0,1,2,4)$ (where 0's are)

- The unary operator NOT, written as $\bar{A}$ for a value A, returns the complement of the value of A; that is it returns 1 if A is 0 and 0 if A is 1. For completeness we give its truth table, even though it is simple.

A	$\bar{A}$
0	1
1	0

) always the opposite

We note that while we have defined OR (+) and AND (·) for just two operands, we can easily extend it to n operands as follows:

- The OR of operands $x_1, x_2, \ldots, x_n$ is 1 only if at least one the values of an operand $x_1,\ldots,x_n$ is 1; otherwise the value is 0. We represent the OR of $x_1,x_2 \ldots,x_n$ as $x_1+x_{2+} \ldots+x_n$

- The AND of operands $x_1, x_2, \ldots, x_n$ is 1 only if the values of all the operands $x_1,\ldots,x_n$ are 1; otherwise the value is 0. We represent the AND of $x_1,x_2 \ldots,x_n$ as $x_1 \times x_2 \cdot \times \ldots \times x_n$ or $x_1 \cdot x_2 \cdot \ldots \cdot x_n$ or even $x_1x_2 \ldots x_n$

The reason we can use such representations when there are more than two operands is because both the OR and the AND operators have the associativity property.

Starting with our above definitions and notations for NOT, AND, OR, we can combine them into more complex Boolean expressions using parenthesis for grouping, if necessary for clarity, in a manner familiar to you from standard algebra. An example of such an expression is

$$(x1 \cdot x2) + \left(x1 + \overline{(x2 \cdot x3)}\right) + x3$$

Canonical Expressions

We now show how one can associate Boolean expressions with a truth table so that the function defined by the Boolean expression has the given truth table as its associated truth table. In particular we show how to derive two such Boolean expressions known as the *canonical Boolean expressions* for the truth table. There are two forms of canonical expressions – *canonical sum-of-products* form and *canonical product-of-sums* form.

1. ***sum of products form*** (SOP):
   a. Each row where the value of the table is 1 is a product term (or *minterm*); all the product terms are to be ORed (although in deference to the use of the symbol +, we use the word "summed" even though there is no addition going on).
   b. For each product term, leave the individual variables uncomplemented if the value of the variable in that row is 1, otherwise complement it.

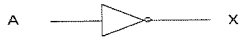
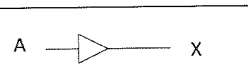
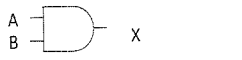
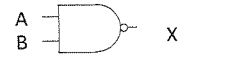
2. ***product of sums form*** (POS):
   a. Each row where the value of the table is 0 is a sum term (or *maxterm*); all the product terms are to be ANDed (although again in deference to the notation used we sometimes say "multiplied," even though no arithmetic multiplication is going on).
   b. For each sum term, leave the individual variables uncomplemented if the value of the variable in that row is 0, otherwise complement it.

Examples: Done in class ended on Jan 14

Section 3. Logic Gates and Digital Circuits

Below

One way to implement digital circuits is to build them from simpler logic circuits known as ***logic gates***. ~~On the following page~~ we list some of the fundamental logic gates that we will use in this chapter, and indeed in the rest of the course. All are combinational circuits. Rather than describe the input-output relationship with cumbersome truth-tables, however, we use Boolean expressions instead.

Name	Symbol	Expression
inverter		$X = \overline{A}$
buffer		$X = A$ Stronger signal amplifier
AND		$X = A \bullet B$ or $X = AB$
OR		$X = A + B$
NAND		$X = \overline{A \cdot B}$
NOR		$X = \overline{A + B}$
XOR		$X = A\,\overline{B} + \overline{A}B$ (also denoted $A \oplus B$)

can implement any circuit

Same value = 0
diff val = 1

Although we don't show them here, one can visualize AND, OR, NAND and NOR gates that have three or more inputs, and indeed we shall assume these gates exist for any desired number of inputs, even if no such gates are actually manufactured.

Read!! Pages 148-149 of your textbook have a discussion of how the inverter, NAND, and NOR gates might be implemented using transistors and resistors.

Simplier circuits than AND & OR

Examples of Implementations of Logic Circuits

Done in Class

Section 4. Circuit Equivalence, Boolean Algebra

In the previous section we found that both the canonical sum of products form and the canonical product of sums form can be used to express the same function, as evidenced by the fact that both expressions have the same truth table. Two logical expressions are said to be ***logically equivalent*** if they both generate the same truth table, or define the same logic function.

Logical equivalence is particularly important when used in connection with logic circuits as simpler logical expressions generally result in simpler circuit which may involve fewer a gates overall and fewer inputs in some of these gates.

Example: Consider the logic function $f(x,y,z) = \Sigma(0,2,4,6,7)$.

The canonical expressions associated with this function are:

Canonical SOP: $f(x,y,z) = \bar{x}\bar{y}\bar{z} + \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z} + xyz$ same as $\bar{z} + xy$

Canonical POS: $f(x,y,z) = (x + y + \bar{z})(x + \bar{y} + \bar{z})(\bar{x} + y + \bar{z})$

$\bar{16}\ \bar{8}\ \bar{4}\ \bar{2}\ \bar{0}$

Jan 16

Truth Tables > $\underbrace{\Sigma \text{ or } \Pi}_{\text{logical functions}}$ (all the 1's / all the 0's) > $\underbrace{\text{SOP or POS}}_{\text{canonical / logical expressions}}$

ex 1

SOP

$f(A,B,C) = \sum (1,3,4,6)$

$= \bar{A}\bar{B}C + A\bar{B}\bar{C} + \bar{A}BC + AB\bar{C}$] based off of the truth table

(everything becomes a 1)

POS

$f(A,B,C) = \Pi (2,5,7)$

$= \underbrace{(A+B+C)}\ (A+\bar{B}+C)(\bar{A}+B+\bar{C})(\bar{A}+\bar{B}+\bar{C})$

(everything becomes a 0)

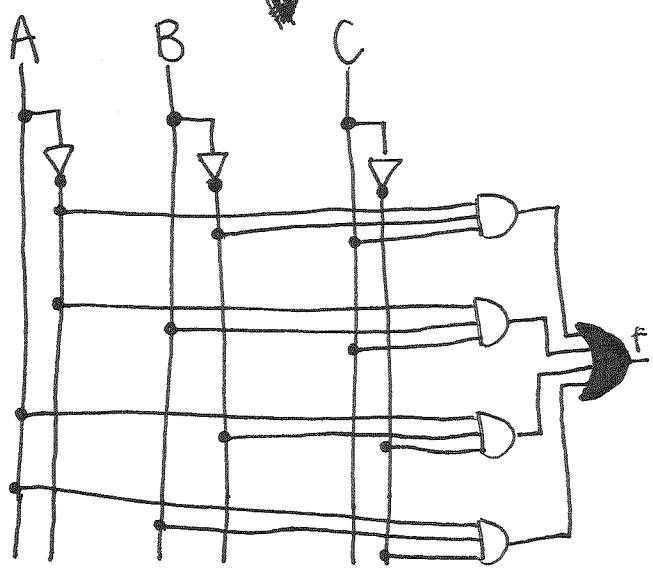
ex 2

00010 10001 10100 10101

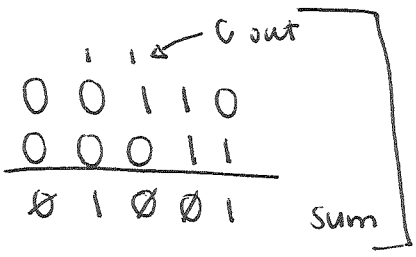
SOP

$f(A,B,C,D,E) = \Sigma(2,17,20,21)$

$= \bar{A}\bar{B}\bar{C}D\bar{E} + A\bar{B}\bar{C}\bar{D}E + A\bar{B}CD\bar{E} + A\bar{B}C\bar{D}E$



Jan 14, 2020



FULL ADDER EXAMPLE

Logical (Boolean) Expressions

OR (+) $A + B + C$
AND (x) $A \times B \times C = A \cdot B \cdot C = ABC$ } Associative $\therefore (A \times B) \times C \equiv A \times (B \times C)$
NOT ($\bar{}$)

Canonical Expressions

if you follow this pattern there is only 1 answer

① SOP Sum of Products $\leftrightarrow$ 1s

$$Sum(A, B, C_{in}) = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C_{in}} + A\bar{B}\bar{C_{in}} + ABC_{in}$$

* put a bar over 0 value in full adder truth table
* you list 4 because there are 4 1's in the Sum Column

$$Cout(A, B, C_{in}) = \bar{A}BC_{in} + A\bar{B}C_{in} + AB\bar{C_{in}} + ABC_{in}$$

* goal is to make everything 1 (one)

② POS Product of Sums

$$Sum(A, B, C_{in}) = (A+B+C_{in})(A+\bar{B}+\bar{C_{in}})(\bar{A}+B+\bar{C_{in}})(\bar{A}+\bar{B}+C_{in})$$

* put a bar over 1 value

$$Cout(A, B, C_{in}) = (A+B+C_{in})(A+B+\bar{C_{in}})(A+\bar{B}+C_{in})(\bar{A}+B+C_{in})$$

* pick SOP or POS based on which has less 1's or 0's

Shefali Emmanuel
Jan 16, 2020

Note, however, that the Boolean expression $\bar{z} + xy$ is logically equivalent to both of the above expressions but is structurally much mpler than either f the above, and hence yields a much simpler implementation as a logic circuit.

Boolean Algebra MATH 207

Determining that two logical expressions are logically equivalent can be tedious and time-consuming, even for expressions with a small number of inputs, if the expressions have many or complex terms. Even more challenging, however, is to find an expression that is indeed simpler, but logically equivalent to, a given expression. Fortunately there is an abstract mathematical system – *Boolean algebra* – that assists us greatly, if not invaluably, in this effort. Although originally based on formal logic, Boolean algebras can be generalized beyond this particular case, and it is this general system that we define now.

Definition of a Boolean Algebra: A ***Boolean algebra*** is a mathematical system $< B, +, \bullet, \bar{} >$, where B is a set and

question on Test 1

$+ : B \times B \to B$

$\bullet : B \times B \to B$

$\bar{} : B \to B$

Are operators such that for all x, y, z in B

1. $x + y = y + x;\ x \bullet y = y \bullet x$ *commutativity* of + and $\bullet$
2. $(x + y) + z = x + (y + z);\ (x \bullet y) \bullet z = x \bullet (y \bullet z)$ *associativity* of + and $\bullet$
3. $x + (y \bullet z) = (x + y) \bullet (x + z);\ x \bullet (y + z) = x \bullet y + x \bullet z$ *distribution* of + over $\bullet$ and $\bullet$ over + diff from normal algebra

In addition there exist distinguished elements 0 and 1 in B such that for all x in B

4. $x + 0 = x;\ x \bullet 1 = x$

   $x + \bar{x} = 1;\ x \bullet \bar{x} = 0$

As we've noted earlier, often adjacency of operands is used instead of the dot notation, whence we write xy instead of $x \bullet y$.

Of special interest to us is the Boolean algebra with $B = \{0,1\}$ and where $+ \leftrightarrow$ OR, $\bullet \leftrightarrow$ AND, and $\bar{} \leftrightarrow$ NOT. This particular Boolean algebra is more properly called the *logic algebra* or *switching algebra*, and was in fact the prototype Boolean algebra. It is often referred to as **the** Boolean algebra. However **it is not the only Boolean algebra**.

Example: Let S be an arbitrary set and let B be the power set of S; that is, the set of all subsets of S. If we now make the following associations $0 \leftrightarrow \phi$; $1 \leftrightarrow S$; $+ \leftrightarrow \cup$; $\bullet \leftrightarrow \cap$; and $\bar{} \leftrightarrow$ set complement, then B is a Boolean algebra. Moreover, if S has n elements, then B has 2^n elements, thus we can generate infinitely many examples of Boolean algebras with more than 2 elements.

The following results can be derived about all Boolean algebras (not just the logic algebra). things that can be proven

a. (idempotency) $x + x = x;\ x \bullet x = x$
b. $x + 1 = 1;\ x \bullet 0 = 0$
c. (absorption 1) $x + x\,y = x;\ x\,(x + y) = x$
d. (absorption 2) $x + \bar{x}y = x + y;\ x\,(\bar{x} + y) = xy$
e. (DeMorgan's laws) $\overline{x + y} = \bar{x} \bullet \bar{y};\ \overline{x \bullet y} = \bar{x} + \bar{y}$

Boolean algebras are also ***dual*** with respect to + and $\bullet$, i.e. if one relationship is true in the Boolean algebra then the relationship derived by interchanging + and $\bullet$ and 0 and 1 is also true in the algebra.

Section 5. Circuit Simplification and Gate Homogeneity

The real power of the relationship between Boolean algebra and digital circuits manifests itself in the following way: As we have already seen, any combinational circuit can be represented by a Boolean expression and vice-versa. Using Boolean algebra one can transform this Boolean expression (and hence the combinational circuit) into one which is different but logically equivalent. Two particular types of transformations are particularly desirable.

1. Transform a given combinational circuit into one which requires a smaller number of gates (*circuit simplification*).
2. Transform a combinational circuit into one which uses gates chosen from a restricted set of gate types. We have already shown that any combinational circuit can be implemented using only AND gates, OR gates, and inverters. We shall show later that in fact any circuit can be implemented using just NAND gates or just NOR gates (*gate homogeneity*).

We now consider each of these issues.

Circuit Simplification

As we noted earlier, the complexity of a logic circuit that implements a Boolean function is directly related to the complexity of the Boolean expression that defines the function. It is to the advantage of a circuit designer therefore to strive for a design which simplifies a combinational circuit in the following ways:

1. The length of the chain of gates input signals must traverse to produce the desired output signal should be as short as possible, since each gate through which signals are propagated causes a delay between the arrival of the signals and the appearance of the appropriate output signal. The existence of canonical POS and SOP expressions for a truth table show us that any combinational circuit can be realized as a two level circuit if input signal are available both in their true form and in their inverted form; if both of these forms are not available, an additional level of inverters may be needed to generate them.
2. The number of gates and gate inputs should be as small as possible since these influence the cost of implementing the circuit.

In the following discussion we first show how Boolean expressions can be simplified using relations from Boolean algebra. Unfortunately, for complex expressions it is usually difficult to tell whether a simplified expression meets criterion 2 above. We follow this with a gate minimization technique based on a structure known as a *Karnaugh map* which allows one to create a SOP o. POS expression for a truth table which uses a minimum number of terms and variables. Further simplification may still be possible, but usually the expression derived with by Karnaugh maps is sufficient.

A. **Simplification Using Boolean Algebra Relations:** We use examples to illustrate how Boolean algebra can be used to simplify an expression defining a Boolean function.

Examples: Done in class ended on Jan 21

B. **Simplification Using Karnaugh Maps:** A *Karnaugh map* is an arrangement of cells, each one representing a combination of variables in a truth table.

2 input Karnaugh map

AB	
00	
01	
11	
10	

3 input Karnaugh map

A \ BC	00	01	11	10
0				
1				

4 input Karnaugh map

AB \ CD	00	01	11	10
00				
01				
11				
10				

Jan 21

$f(c,d,e) = \sum(0,2,4,6,7)$ (all 1's)

$SOP = (\bar{c}\bar{d}\bar{e}) + (\bar{c}d\bar{e}) + (c\bar{d}\bar{e}) + (cd\bar{e}) + (cde)$

Boolean Algebra Simplification

$= \left((\bar{c}\bar{d}\bar{e}) + (\bar{c}d\bar{e})\right) + c\bar{d}\bar{e} + cd\bar{e} + cde$

$= \bar{c}\bar{e} + (\bar{d} + d) + c\bar{d}\bar{e} + cd\bar{e} + cde$

1

$= \bar{c}\bar{e} + (c\bar{d}\bar{e} + c\bar{d}\bar{e}) + cde$

$= \bar{c}\bar{e} + c\bar{e} + (\bar{d} + d) + cde$

1

$= \bar{c}\bar{e} + c\bar{e} + cde$

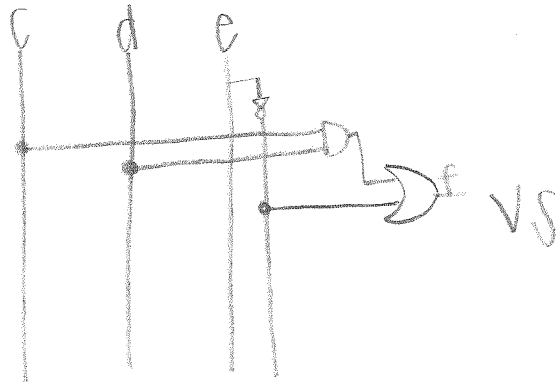
$= \bar{e}(\bar{c} + c) + cde$

1

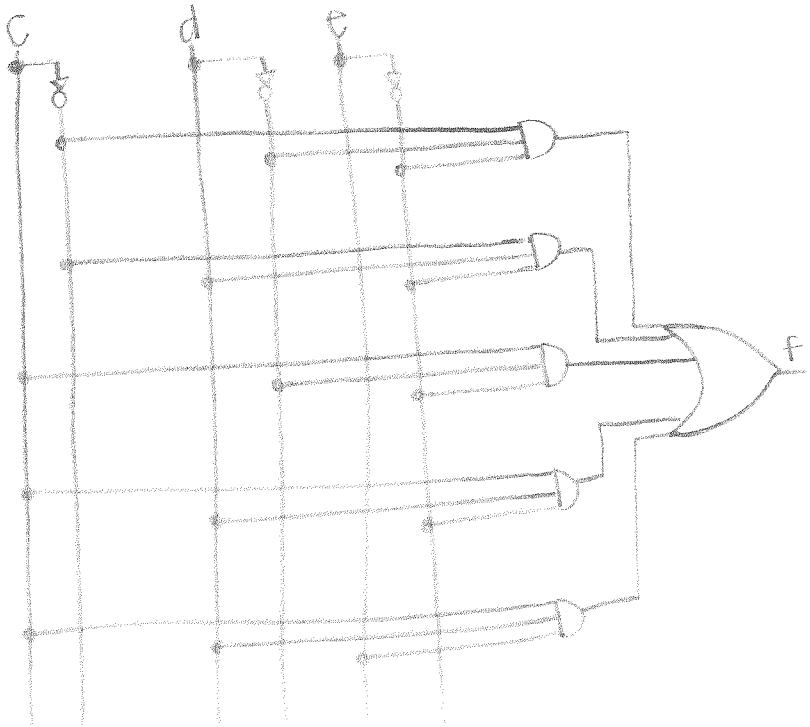
$= \bar{e} + cde$ — $e = \bar{\bar{e}}$

$= \bar{e} + cd\bar{\bar{e}}$ — absorbtion

$f = \bar{e} + cd$



VS



Set $S = \{a, b, c\}$

power set $PS = \{\{\emptyset\}, \{a\}, \{b\}, \{c\}, \{ab\}, \{ac\}, \{bc\}, \{abc\}\}$

$+ \longleftrightarrow \cup$

$\cdot \longleftrightarrow \cap$

$- \longleftrightarrow$ compliment

$0 \longleftrightarrow \emptyset$

$1 \longleftrightarrow \{abc\}$

---

ex 1

$x = x + 0$

$= x + (x \cdot \bar{x})$

$= (x+x) \cdot (x+\bar{x})$

$= (x+x) \cdot 1$

$= (x+x)$

$= x$

ex 2

$x = x \cdot 1$

$= x(x+\bar{x})$

$= (x \cdot x) + \underbrace{(x \cdot \bar{x})}_{\emptyset}$

$= (x \cdot x) + 0$

$= x \cdot x$

$= x$

ex 3

$x = x + xy$

$= x \cdot 1 + xy$

$= x(1+y)$

$= x \cdot 1$

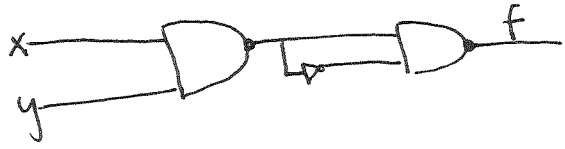
$= x$

ex 4

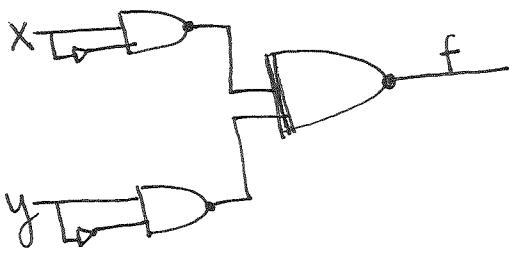
$\stackrel{=}{x} = x$

but how will you prove it??

$x + y = \overline{\overline{x+y}}$



$\overline{x+y} = \bar{x} \cdot \bar{y}$



$$f(x,y,z) = \left[\bar{x}\bar{y}z + \bar{x}y\bar{z}\right] + \left[x\bar{y}\bar{z} + xy\bar{z}\right] + xyz$$

$$= \bar{x}z\underbrace{(\bar{y}+y)}_{1} + x\bar{z}\underbrace{(\bar{y}+y)}_{1} + xyz$$

logically adjacent

this will work for groups that are powers of

$$= \bar{x}\bar{z} + x\bar{z} + xyz = \bar{z}\underbrace{(\bar{x}+x)}_{1} + xyz$$

$$= \bar{z} + xyz$$

$$= \bar{z} + xy$$

Jan 28, 2020

Boolean Algebra Simplifications

x	y	
0	0	1
0	1	1
1	0	0
1	1	0

$$\bar{x}\bar{y} + \bar{x}y = \bar{x}\underbrace{(\bar{y}+y)}_{1} = \bar{x}$$

x	y	
0	0	0
0	1	0
1	0	1
1	1	1

$$x = x\bar{y} + xy = x\underbrace{(\bar{y}+y)}_{1}$$

$\bar{x}y + \bar{y}x$

x	y	
0	0	0
0	1	1
1	0	1
1	1	0

physically adjacent but not logically adjacent

x	y	
0	0	0
0	1	1
1	0	0
1	1	1

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

logically adjacent but not physically

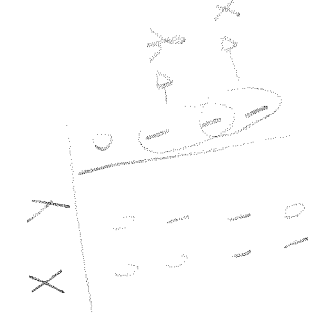
Reorder

x	y	y
0	0	0
0	1	1
1	1	1
1	0	0

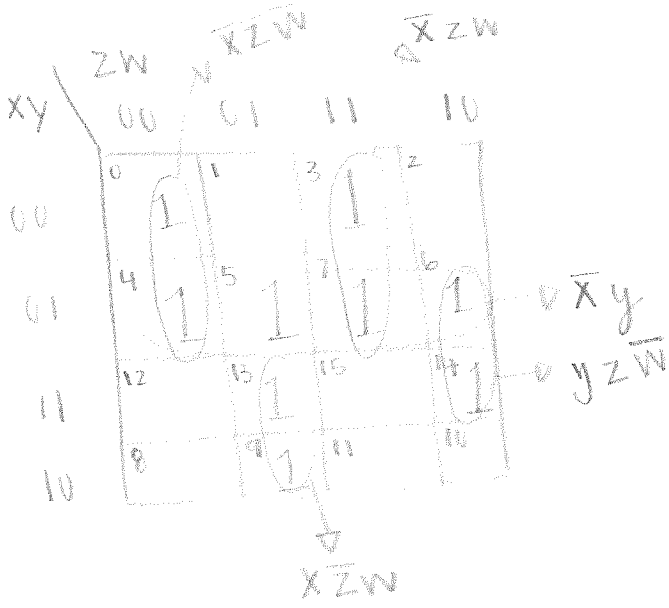
better version

adjacent like a top & bottom

KARNAUGH

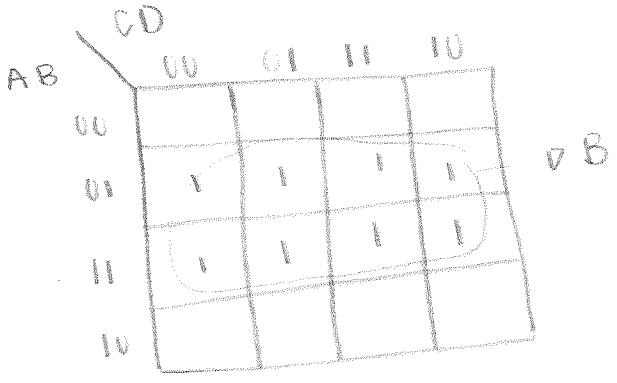


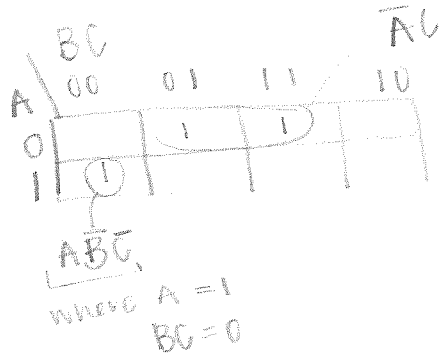
$$f(X,Y,Z,W) = \sum(14,9,1,3,4,13,5,6,0)$$



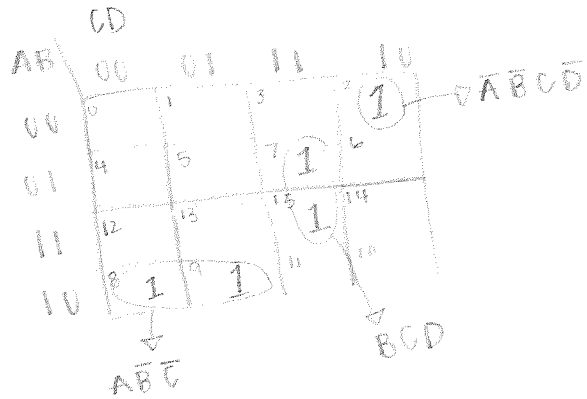
$$\bar{X}\bar{Z}\bar{W} + \bar{X}ZW + \bar{X}y + yZ\bar{W} + X\bar{Z}W$$

Single term	4 var
Couple	3 var
4	2 var
8	1 var
all	none var

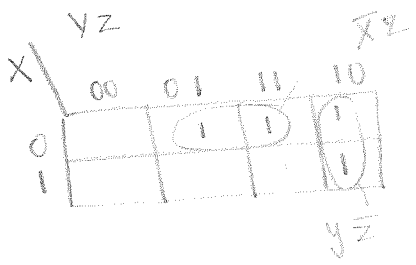




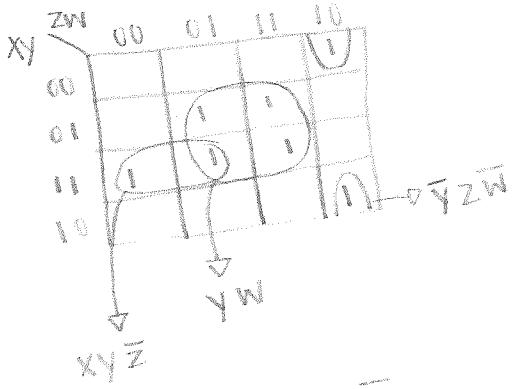
$A\bar{B}\bar{C} + \bar{A}C$



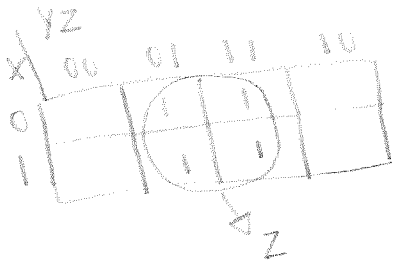
$A\bar{B}\bar{C} + BCD + \bar{A}B\bar{C}\bar{D}$



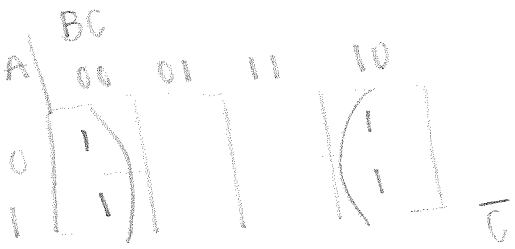
$\bar{X}z + y\bar{z}$



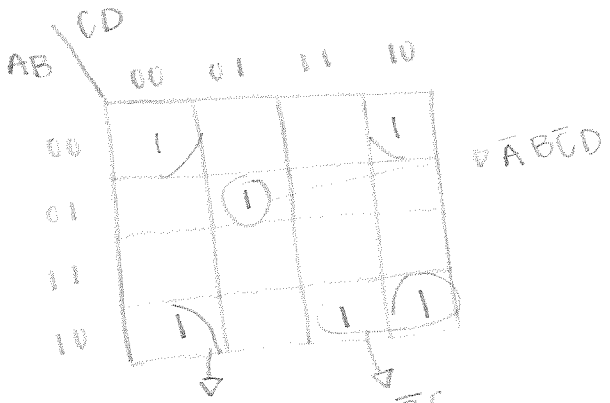
$xy\bar{z} + yw + \bar{y}z\bar{w}$



What is constant?

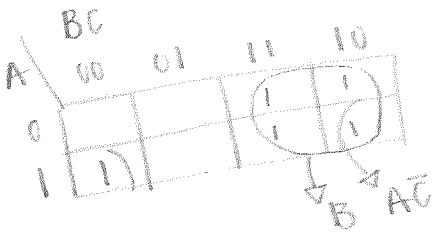


1 group of 4



$\bar{B}\bar{D}$

$\bar{B}\bar{D} + A\bar{B}C + \bar{A}B\bar{C}D$



$B + A\bar{C}$

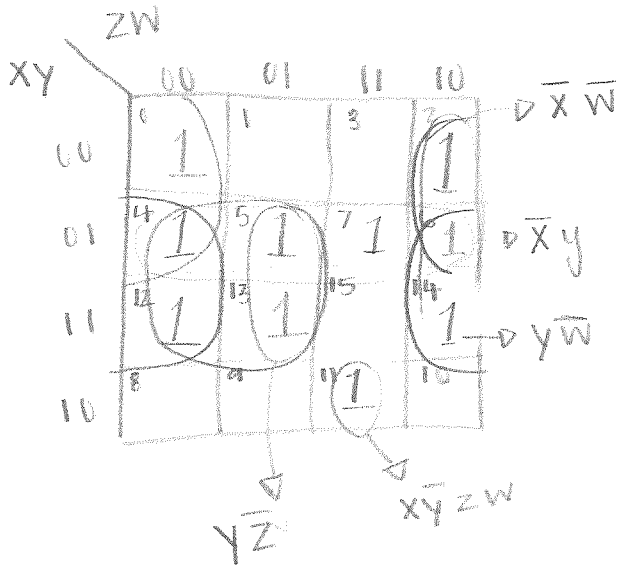
3 Input

4 Input

4 corner attempt

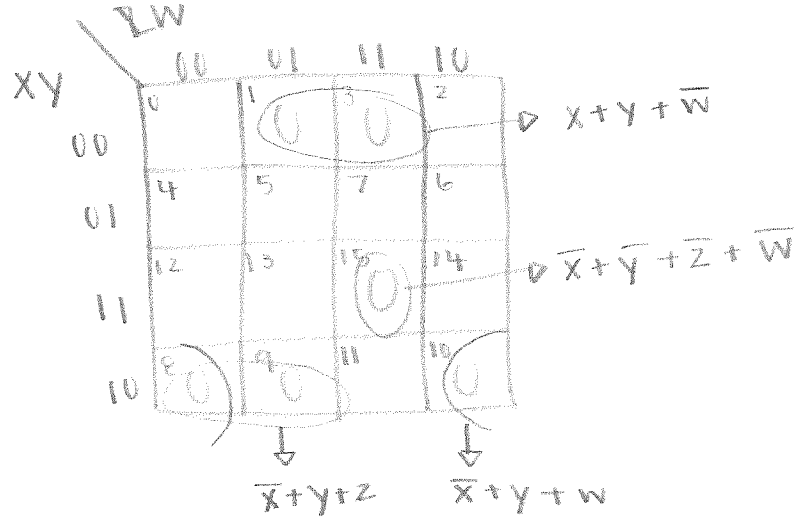
SOP

$$f(x,y,z,w) = \sum(0,6,13,7,14,12,11,2,5,4)$$
$$= y\bar{z} + \bar{x}\bar{w} + \bar{x}y + y\bar{w} + x\bar{y}zw$$



POS

$$= (\bar{x}+y+z)(\bar{x}+y+w)(x+y+\bar{w})(\bar{x}+\bar{y}+\bar{z}+\bar{w})$$
$$f(x,y,z,w) = \prod(1,3,8,9,10,15)$$

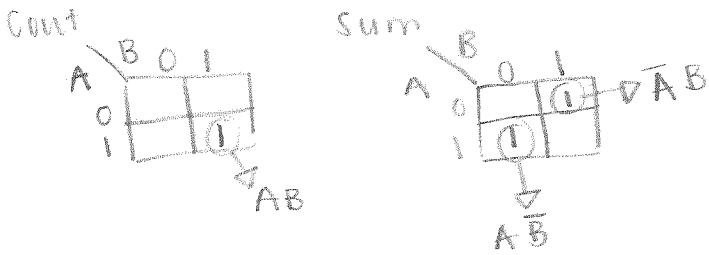


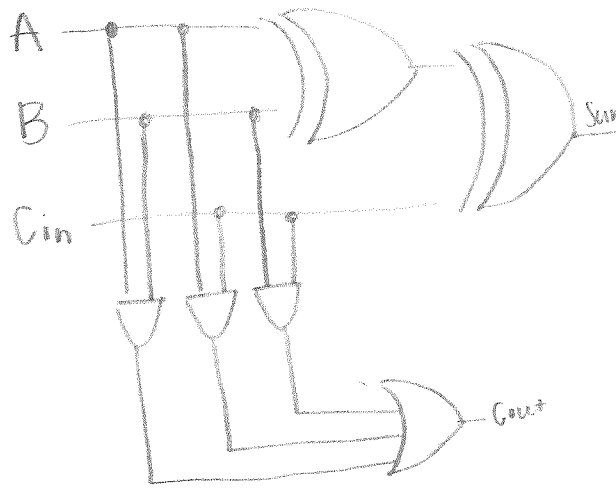
Jan 30, 2020

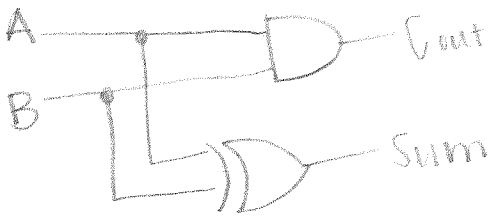
half adder

A	B	Cout	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

exclusive or
$A \oplus B$







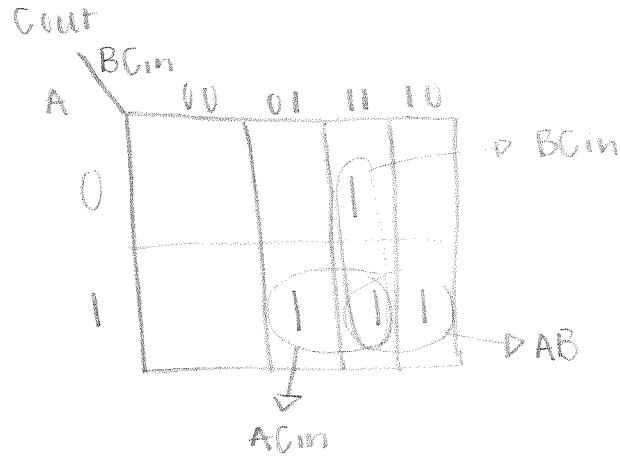
$S_1 = C_0 \oplus (A_0 + B_0)$

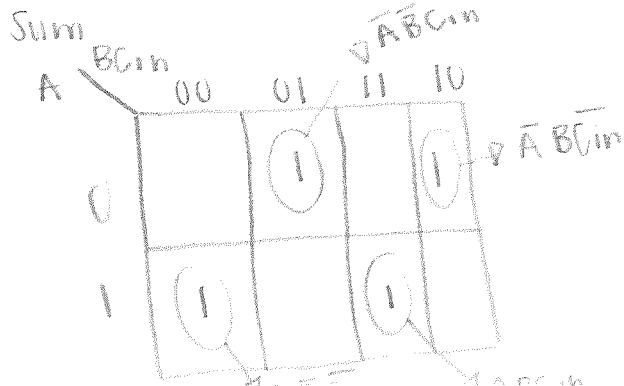
$C_1 = A_1 B_1 + B_1 C_0$

$C_0 = A_0 B_0 + B_0 C_{in}$

full adder

A	B	Cin	Cout	Sum
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





$= \bar{A}\bar{B}C_{in} + (\bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in}) + ABC_{in}$

$= \bar{A}\bar{B}C_{in} + \bar{C}_{in}(\bar{A}B + A\bar{B}) + ABC_{in}$

$= \bar{C}_{in}(A \oplus B) + (\bar{A}\bar{B} + AB)C_{in}$

$= (A \oplus B)\bar{C}_{in} + \overline{(A \oplus B)}C_{in}$

demorgans

$$\overline{\bar{A}B + A\bar{B}} = \overline{\bar{A}B} \cdot \overline{A\bar{B}}$$

$$= (\bar{\bar{A}} + \bar{B})(\bar{A} + \bar{\bar{B}})$$

$$= (A + \bar{B})(\bar{A} + B)$$

$$= A\bar{A} + AB + \bar{A}\bar{B} + \bar{B}B$$

$$= AB + \bar{A}\bar{B}$$

$\bar{\bar{X}} = X$

prove for set of

4 ? 3 ? 16 ? 32 ? 64

only use first 5 properties

not demorgans

(1) start out w/ 1 ? replace w/ equiv

(2) do stuff w/ both then stop when they look the same

To minimize a SOP expression the output values from the truth table are inserted in the table in squares determined by the input combination which produced the output value and the input combination represented by each square. One then seeks out and circles groups of "logically adjacent" 1's of the largest possible sizes that are powers of two. We define *logically adjacent* as follows:

1. Two squares of 1's are logically adjacent if the product terms they represent are the same except in one variable, where they are complementary. Such a pair of cells can be described by a product term comprised of the variables where they agree.

Examples:

2. Four squares of 1's are logically adjacent if they can be decomposed into two sets of logically adjacent pairs so that the product terms describing these pairs are the same except in one variable, where they are complementary. Such a quadruple of cells can be described by a product term comprised of the variables that agree in the product terms of the pairs into which the quadruple was decomposed.

Examples:

3. In general 2^n squares of 1's ($n > 1$) are logically adjacent if they can be decomposed into two (disjoint) sets of 2^{n-1} logically adjacent squares of 1's such that the product terms which describe each of these sets are the same except in one variable, where they are complementary.

Such a group of 2^n cells can be described by a product term comprised of the variables that agree in the product terms of the sets of 2^{n-1} cells into which the given group was decomposed.

Examples:

C. **Quine-McKluskey Tables**: Karnaugh maps are effective for generating minimimal SOP expression involving up to four variables, and with some additional effort can be made to work for expressions requiring up to six variables. For expressions involving more than 6 variables, it is impractical to use Karnaugh methods. Fortunately there is an expression minimization method that is practical for more than 6 variables; in fact this method can be described algorithmically and hence implemeted by a computer. The method involves the use of Quine-McKluskey tables. Since we shall have no need for the method in this course, however, we do not discuss it here, but rather leave it to the student to explore this technique.

Gate Homogeneity in Logic Circuits

In the following discussion we show how any Boolean expression can be implemented using only NAND gates or only NOR gates. The ability to employ such gate homogeneity in a circuit is attractive from an engineering viewpoint since NAND gates and NOR gates have simple electronic realizations. Unfortunately logic design with NAND and NOR gates is not as straightforward as design with AND and OR gates and inverters (primarily because associativity does not hold for NOR and NAND), whence achieving gate minimization and gate homogeneity is more difficult. For our purposes we concentrate on showing how (minimal) POS and SOP expressions can be implemented using only NAND gates or only NOR gates. In logic diagrams a small circle is used to denote complementation as is observable in the graphic symbols for inverters, NOR gates and NAND gates.

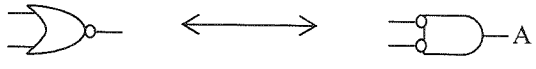
Consider now the Boolean expressions

$\overline{x+y}$ and $\overline{x \bullet y}$

which can be implemented in a digital circuit by a NOR gate and a NAND gate respectively. If we apply DeMorgan's laws to these expressions however we get

$$\overline{x+y} = \bar{x} \bullet \bar{y} \quad \text{and} \quad \overline{x \bullet y} = \bar{x} + \bar{y}$$

This suggests that in addition to representing a NOR gate with an OR symbol followed by a circle, we can represent it by an AND symbol preceded by circles in all inputs.



Similarly a NAND gate can be represented by an OR symbol preceded by circles in all inputs.

Finally we note that by idempotency

$$\overline{x+x} = \bar{x} \text{ and } \overline{x \bullet x} = \bar{x}$$

suggesting that inverters can be implemented with a NAND gate or a NOR gate as follows:

We now show by examples how these observations can be used to convert any digital circuit to one which uses only NAND gates or only NOR gates.

Examples:

Done in class

Section 6. Basic Logic Circuits I - Combinational Circuits

Integrated Circuits

Logic gates are available commercially in integrated circuit (IC) form. Here the gates are manufactured on a silicon wafer and sealed in a protective package that has pins protruding from it in order to make connections to the gates. Depending on the number of gates they contain, ICs are roughly classified as follows:

small scale integrated circuit (SSI)	1 to a few 10s of gates
medium scale integrated circuit (MSI)	a few 10s to a few 100s of gates
large scale integrated circuit (SLSI)	a few 100s to a few 100,000 gates
very large scale integrated circuit (VLSI)	more than a few 100,000 gates

Combinational Circuits

For the remainder of this chapter we shall discuss some fundamental design principles of digital circuits, emphasizing those that relate directly to computer organization. In keeping with our hierarchical approach, we shall use gates as the primitive components for our components, and use these to develop other SSI and MSI circuits (most of which are themselves available as integrated circuits). These will in turn become primitive components of sub-layers within the digital logic level.

As we have seen, combinational circuits can be described by a truth table showing the binary relations between n inputs and m outputs or by m Boolean functions of n input variables, thus allowing many of the techniques of the previous section to come into play. Our first three examples below illustrate this approach to combinational circuit design, though it works well primarily for relatively simple circuits (SSI and some MSI) and is generally not widely used with more complex circuits (MSI and above).

Examples of combinational circuits

1. The *full adder*: A full adder accepts three bits as input - 2 addend bits, A and B, and a carry-in bit , C_{in}, and produces as output a sum bit, S, and a carry-out bit, C_{out}. This circuit is not new, as we discussed it earlier. What we do here, however, is provide an alternative implementation of the circuit.

C_{in}	A_0	B_0	C_0	S_0
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

Feb 4

If we represent a one-bit adder by the following symbol

then we can represent a two-bit adder by the following symbol

Moreover we can implement this two-bit adder using two one-bit adders as follows

This implementation is known as a ***ripple-carry adder*** for the manner in which it determines the final carry-out. The carry-out of the first full adder must feed (or "ripple") into the second full adder as its carry-in. As you will see in an assignment, when incorporated into higher bit adders it can result in a significant delay in calculating the adder's final sum and carry-out over other approaches such as carry-select (discussed in your textbook on pages 165-166) and carry-lookahead (Exercise 14 at the end of Chapter 3).

2. A 2-bit left/right ***shifter*** is a circuit that has two inputs D0 and D1, a control line C that determines the direction of the shift (0 for left and 1 for right), and two output lines S0 and S1. On a left shift, S0s gets the value 0 and S1 gets the value of S0; on a right shift S1 gets the value 0 and S0 gets the value of S1. A truth table for the shifter is shown below, following by (simplified) expressions for S0 and S1 in terms of C, D0, and D1.

C	D1	D0	S1	S0
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

$S1 = \bar{C}D0$

$S0 = CD1$

Similarly we can specify a 3-bit left-right shifter with the following truth table and expressions for S0, S1 and S2 in terms of C, D0, D1 and D2

C	D2	D1	D0	S2	S1	S0
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	0	0
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	1	0
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	1
1	0	1	1	0	0	1
1	1	0	0	0	1	0
1	1	0	1	0	1	0
1	1	1	0	0	1	1
1	1	1	1	0	1	1

$S2 = \bar{C}D1$

$S1 = \bar{C}D0 + CD2$

$S0 = CD1$

Finally, analyzing what we have seen in the expressions for the S's above in terms of C and the D's, we extrapolate that in an 8-bit left-right shifter, we get the following expressions for the S's in terms of C and the D's

$S7 = \bar{C}D6$

$Si = \bar{C}D_{i-1} + CD_{i+1}$ for i = 2,3,4,5,6

$S0 = CD1$

A circuit that implements the above expressions is shown on page 164 of your textbook. Recall that shifting provide a simple means for carrying out multiplication by 2 (shift left) or division by 2 (shift right to get the quotient).

Designing the circuits for the next set of examples using truth tables and minimization techniques such as Karnaugh maps would be impractical. Instead we use more intuitive approach, similar to what we did for the shifter, based on what we know about how the circuit should work.

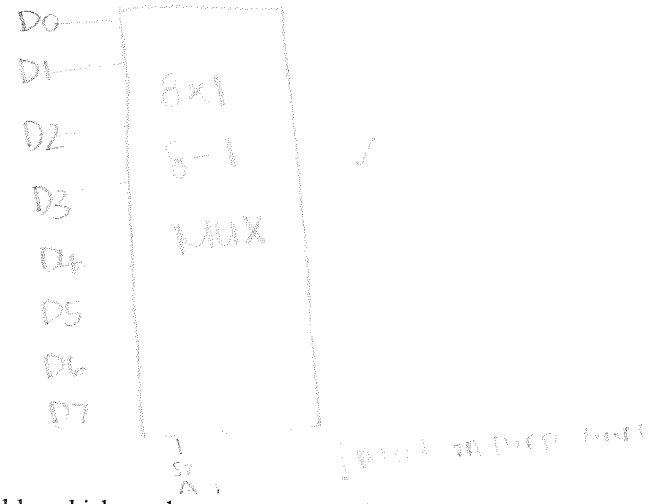
3. *Multiplexers*: A $2^n \times 1$ *multiplexer* receives input on 2^n input lines and transfers the value of one of these input lines to a single output line. The input line to be selected is determined from the bit combination of n additional input lines known as the *selection lines*.

Example: On page 161, Figure 3-12 (a) we have a graphic symbol for an 8-input multiplexer with 8 input lines D0 , D1 ,..., D7, one output line F, and 3 select lines A , B , C (although I would prefer S2, S1, S0 repsectively).

- Multiplexers are available as integrated circuits.

A truth table for a 2 × 1 multiplexer, with associated expression for F would be the following

A	D1	D0	F
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



$$F = \bar{A}D0 + AD1$$

Representing a 4 × 1 multiplexer requires a 6-input truth table, which we do not want to work with, , but if we want to have the following associations between select values (A,B) and output F

(0,0) → F=D0, (0,1) → F=D1, (1,0) → F=D2, (1,1) → F=D3

A moment's refection should convince one of the validity of the following expression for representing F

$$F = \bar{A}\bar{B}D0 + \bar{A}BD1 + A\bar{B}D2 + ABD3$$

Similarly we would get the following expression for the output F of an 8 × 1 multiplexer

$$F = \bar{A}\bar{B}\bar{C}D0 + \bar{A}\bar{B}CD1 + \bar{A}B\bar{C}D2 + \bar{A}BCD3 + A\bar{B}\bar{C}D4 + A\bar{B}CD5 + AB\bar{C}D6 + ABCD7$$

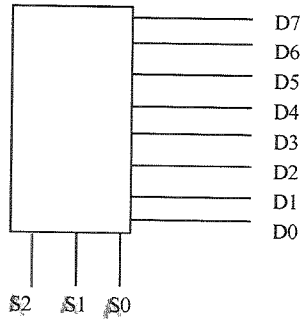
An implementation of this expression is given in Figure 3-11 of your textbook.

In the following example we show how Boolean functions can be implemented using multiplexers.

Example:

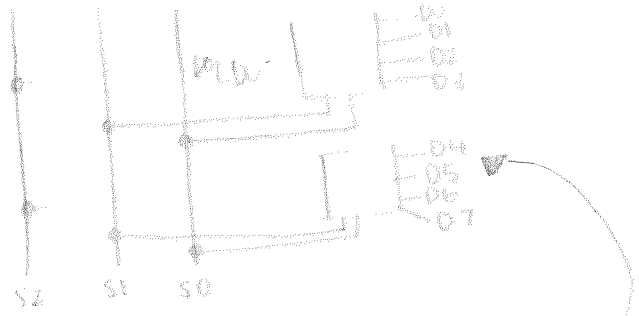
4. *Decoders*: An n × 2^n *decoder* receives input on n input lines A_0 ,..., A_{n-1} and by treating these signals as a binary number, sets the signal on exactly one of 2^n output lines to 1 (known as *selecting the line*), and makes the rest 0. We label the output lines as D_0 ,..., D_{2n-1}.

Example: In a 3 × 8 decoder such as the one below, input signals of 1 0 1 might select line D_5 (generally one must use a data sheet to find the correspondence between given input signals and the line selected).



- Often an *enable input* is included on a decoder to control its operation. Here all of the outputs will be 0 (be *disabled*) if the enable input has one value, say 0, while if the enable input has the other value, in this case 1, the decoder will operate as expected (be *enabled*).

Examples of 3-to-8 decoders with enable.



Decoders are available as integrated circuits.

In the following example we show how to use two 2 × 4 decoders with enables to construct a 3 × 8 decoder.

Example: Done in Class

5. *Arithmetic logic units*: In this example we show how some of the previous circuits can be used to design a simple 1-bit arithmetic/logic unit (ALU). This ALU will be capable of performing the following operations with operands A and B:

$$A \text{ AND } B,\ A \text{ OR } B,\ \overline{B},\ A + B + \text{carry-in}$$

An operation is selected according to the following input signals on the select lines F0 and F1:

F1	F0	operation
0	0	A AND B
0	1	$\overline{B}$
1	0	A OR B
1	1	A + B + carry-in (produces carry-out)

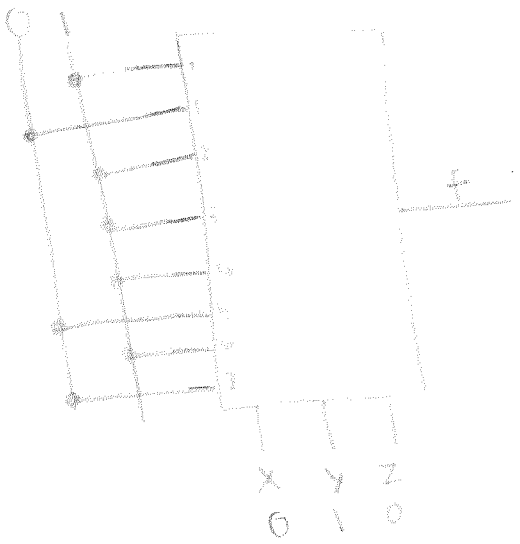
If we represent this 1bit alu by the graphic symbol

then we can combine eight such alu's to produce an 8-bit alu for operands $A_7 \ldots A_0$ and $B_7 \ldots B_0$ - a design technique known as *bit-slicing*. An example of bit-slicing is shown on page C-30 of your textbook.

$f(x,y,z) = \sum_1(4,3,7,6,0)$

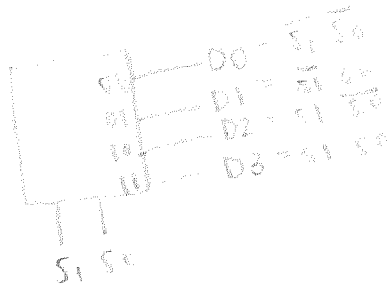
the pattern should be in this form

4 x 1 MUX



S0	D1	DC
0	0	1
1	1	0

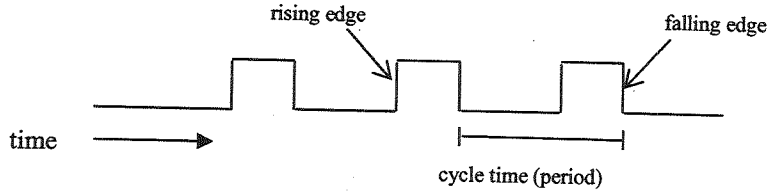
D1 = S0
DC = S1



Feb 11

Latch

1. An *asynchronous sequential circuit* is one in which the circuit output depends on the order in which the input signals change; the changes can occur at any time.
2. A *synchronous sequential circuit* is one in which the memory elements can change values only at discrete instances of time. flipflop
   - Clock pulses are generated throughout the circuit so that changes can only take place upon the arrival of a pulse.
   - A clock is a circuit that emits a series of pulses (i.e. 0-1 and 1-0 state transitions) with a precise pulse length (width) and precise interval between consecutive pulses (*the clock cycle time,* or *period*).
   - The following diagram illustrates some of the common terminology used in association with clocks



   - The *frequency* of a clock is the reciprocal of the clock cycle time. The most common measure of frequency is the *hertz*, where 1 hertz = 1 cycle per second. Some example of frequencies are 500 MHz and 4 GHz, yielding cycle times of 2 nsec to .25 nsec respectively.

Memory Elements

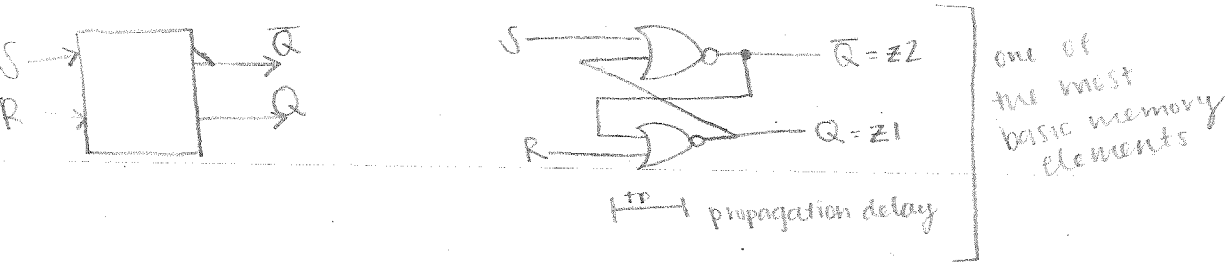
By a "memory element" in a sequential circuit we mean a circuit component capable of maintaining a given value for an indefinite period of time. A typical memory element that stores a single bit will feature one or more inputs that allow one to change the value stored as well as at least one output to indicate the value stored. Often such memory elements will have a second output carrying the complement of the stored value.

simple way to implement a memory element is through a "feedback" circuit such as the one on page 170 of your textbook.

To understand how this circuit works, let tp be the propagation delay for the NOR gate being used . Initially let us consider only those cases where the output lines labelled Q and $\bar{Q}$ do indeed have complementary values abut let's relabel the output lines to where z1↔Q and z2↔$\bar{Q}$. The following table summarizes the changes in value of z1 and z2 over time, starting at time a given time t0.

R	S	t0		t0+tp		t0+2tp		t0+3tp		comments
		z1↔Q	z2↔$\bar{Q}$	z1	z2	z1	z2	z1	z2	
0	0	0	1	0	1	0	1	0	1	stable
0	0	1	0	1	0	1	0	1	0	stable
0	1	0	1	0	0	1	0	1	0	stable Q=1
0	1	1	0	1	0	1	0	1	0	stable Q=1
1	0	0	1	0	1	0	1	0	1	stable Q = 0
1	0	1	0	0	0	0	1	0	1	stable Q = 0
1	1	0	1	0	0	0	0	0	0	stable* Q = 0
1	1	1	0	0	0	0	0	0	0	stable* Q = 0

*Notice in the last row, where R = S = 1 that while the values of z1 and z2 are stable, they are no longer complementary to each other. This will present a problem as we shall see shortly.



We summarize the above behavior more as follows:

R(t)	S(t)	Q(t)	$Q^+=Q(t0+2tp)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	undesirable, it forces z1 =z2
1	1	1	

or more succinctly

book doesn't show this

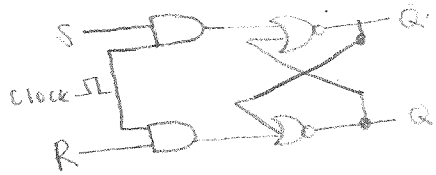
R	S	Q^+
0	0	Q
0	1	1
1	0	0
1	1	undesirable

The memory element we described is for one known as an ***RS-latch.***

La place to put signals before 1 or 0

changes when

Types of memory elements - latched and flip-flops



One bit memory elements can be classified by two criteria:

1. Input conditions that change the value stored.
2. Control signals necessary for the input conditions to take effect.

Let us consider this second criterion first. Here there are three ways to control the time when input conditions are allowed to change the state of a memory element:

1. Any time the input signals change they "immediately" alter the stored value to its appropriate new value (here "immediately" means up to a delay, known as a *propagation delay*, which reflects the time it takes the input signals to work their way through the components of the memory element to produce and store the proper new value). Here there is no control per se over the input conditions.
2. Any time a special control signal is active (high or low, depending on the design of the memory element) the state of the memory element can be changed via its inputs. The control signal is variously known as the *strobe*, *enable*, or *clock pulse*.
3. Any time the control signal changes signal levels appropriately, that is, either from low to high (*positive edge-triggered*) or, alternatively, from high to low (*negative edge-triggered*).

We shall use the term *latch* for those memory elements whose control signals operate as in 1. or 2. above, and the term *flip-flop* for memory elements which behave as in 3.

- Note, while widely used, this terminology is by no means universal. One can pick up textbooks and find references to memory elements described by such terms as "level-triggered flip-flops," or "level-triggered latches." Some authors refer to any memory element which uses an enabling signal as a "flip-flop," leaving the term "latch" only for those which respond according to the first characteristic above. Rarely does this confusion in terminology cause problems.

- There is a special type of flip-flop known as a *master-slave* flip-flop which specifies not only when input signals are captured, but also when output signals take effect. We take up master-slave flip-flops in more detail when we discuss how to implement various types of latches and flip-flops.

Feb 13 Now let us examine some of the commonly used types of input signals used with latches/flip-flops and how these inputs determine the new state of the memory element. For simplicity we initially restrict our discussion to latches.

1. The *RS-latch* (for reset-set, but also known as the SR latch), whose implementation we just saw, has two inputs, R and S, and changes its current value Q, to its new value Q+ as shown in the following table (known as its *characteristic table*).

R	S	Q+
0	0	Q
0	1	1
1	0	0
1	1	not allowed

UNSTABLE

So we use JK latch

To see why R=1, S=1 is not allowed for an RS-latch, first note that it puts the latch in a stable state, but with z1 = z2 = 0. By itself, this causes no problems, but if we later set R = S = 0, which is the "resting" state for the latch (since when z1 and z2 are complementary these values are retained indefinitely) it causes these values to oscillate between 0 and 1 as the following table shows

R	S	t0		t0+tp		t0+2tp		t0+3tp		comments
		z1	z2	z1	z2	z1	z2	z1	z2	
0	0	0	0	1	1	0	0	1	1	unstable

As we shall see in our discussion of implementations, the RS latch can be used to implement the other types of latches.

When one assigns the value "1" to a latch (or flip-flop) the latch is said to be *set*, while giving the latch the value "0" is known as *reset*ting or *clear*ing the latch, hence the use of the identifiers S and R for this latch.

2. The *JK-latch* is similar to the RS-latch except the unstable condition is avoided.

start at ∅

STABLE

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

start @ 1

J	K	Q+
1	1	$\overline{Q}$
1	0	1
0	1	0
0	0	Q

3. The *D-latch* (for "data") has only one data input, which is called D.

D	Q+
0	0
1	1

Implementation of a Memory Elements

Clocked Latches

Most applications require that the time at which a latch is set or reset be controlled more precisely via an enabling input. Such an input can be incorporated into that of the basic latch as shown on page 171.

Note that any changes that occur in the R and S inputs to a clocked RS latch while the clock pulse is high will be followed "immediately" (up to propagation delay) by the appropriate change in the output signal.

Implementation of D Latches and JK Latches

As we noted when we introduced the *D latch* it accepts a single input signal, D, which causes the output signal, Q, to have the same value as D. An implementation for the D latch, also using an RS-latch implementation, is given on page 172. The *JK latch* is similar to the RS latch-flop (with inputs J and K acting as the set and reset lines respectively), except the inputs J=1 and K=1 cause the latch to change state (making it more useful). The following diagram shows how to implement a JK latch using an RS-latch (done in class).

pos edge trigger

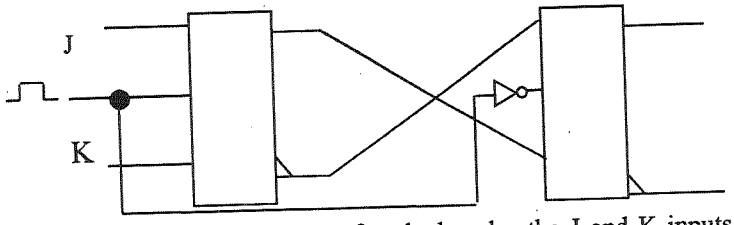
Edge-Triggered Flip-Flops and Master-Slave Flip-Flops

In some circuits the immediate changes in output that follow changes in input signals in latches cannot be allowed, especially in those where the value of Q at a time t0 + t depends on the value of Q at t0 (we will see examples where this arises later). Two types of memory circuitry which provide more control over times when they change states are the *edge-triggered flip-flop* and the *master-slave flip-flop*.

For edge-triggering, additional circuitry is placed at the clock input of a clocked latch so that a prescribed edge of the input pulse, either the rising edge or the falling edge, causes a brief pulse to be generated (see page 173)

This brief time when the latch is enabled allows its inputs to be accepted, with the appropriate output signal then following.

In the master-slave flip-flop, two clocked-latches (a master and a slave) are organized as follows:



Under this arrangement, prior to the rising edge of a clock pulse the J and K inputs should have reached the desired signal levels. While the *clock* is 1 the master latch will assume the appropriate next state while the slave latch maintains the old output value. When the falling edge of the pulse arrives the new state will be transferred to the slave latch.

Registers

While occasionally latches or flip-flops are used by themselves, it is more typical that a group of flip-flops acts together in a coordinated manner. Such a grouping is known as a *register*. More precisely, a *register* is a collection of flip-flops together with combinational circuitry to control when and how information is transferred to the individual flip-flops. One of the simplest schemes is the parallel load register.

Example: an eight-bit register with parallel load (see textbook, page 175) The register is called a *parallel load* register because all the register's bits can be loaded simultaneously.

As we noted above, the example of the register just given is one of the simplest types of registers, and is to the flip-flop what a word gate is to a logic gate.

In order to design more complex registers than what we have seen here we will need a systematic method for designing sequential circuits.

Design Procedure for Sequential Circuits (this is not in your textbook)

1. **Draw a state diagram from the problem statement.** A *state diagram* a graphic method that uses circles and arcs between circles to present the output values of the circuit and the next states of the memory elements as functions of the inputs and present states. Here
   - circle = a state, with state of flip-flops A and B represented inside by the binary number AB
   - arc = a state transition; the notation, I/O, by the arc gives the input value(s), I, that caused the state change and the resulting output value(s), O.

   We will see examples of state diagrams shortly when we actually design some sequential circuits.
2. **Derive a state table from the state diagram.** A *state table* for a sequential circuit is a table which presents the outputs of the circuit and the next states of the flip-flops as functions of the inputs and present states. A state diagram is a graphic method for representing the information in a state table.
3. **Determine the type of latch/flip-flops to be used in the circuit and for each latch/flip-flop determine the value each input of the latch/flip-flop must have to cause the necessary change in state.**
4. **Derive an input equation for each flip-flop input and the circuit output equations.**
5. **Draw the circuit diagram.**

The derivation of the input tables for each flip-flop in the circuit can be simplified by using an *excitation table* for the class(es) of flip-flop(s) being used. This is a table which gives the input signals needed to change the latch from one state to another. It is derived from the characteristic table for the latch.

Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

JK flip-flop

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

D flip-flop

Excitation Tables for JK and D flip-flops
(X = "don't care;" can use 0 or 1)

Examples:

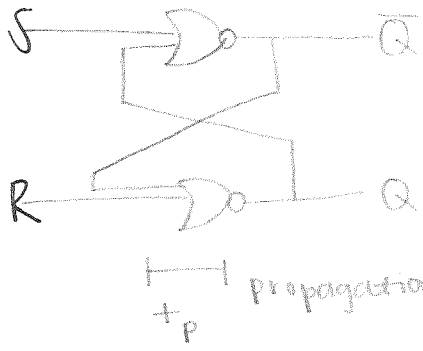
1. A ***serial (binary) adder*** is a circuit which accepts two numbers as input in serial (i.e. bit-by-bit) and generates their binary sum, which is sent out serially. We now apply the design principles for sequential circuits to design a serial adder using JK flip-flops.

2. A ***modulo-n synchronous counter*** is a register that is often used to keep track of the number of times an event occurs, with the occurrence of the event being signaled by an input value of 1 and the state of the counter updated accordingly to reflect the occurrence of the event. Upon the occurrence of the nth event, an output value of 1 is generated and the counter reset to indicate 0 events; in all other cases the output value is 0. Design a modulo-4 counter using JK flipflops.

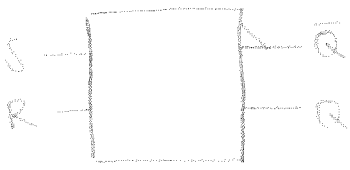
3. A ***shift-right register*** is a register capable of shifting its binary information the right by one bit upon the arrival of a signal 1, while storing the value 0 in the leftmost bit; otherwise the values of the bits remain unchanged. There is no output value. Design a 3-bit shift-right register using D flip-flops.

4. A ***sequence recognizer*** is a circuit designed to generate an output value of 1 when the last bit in an n-bit sequence of input values has arrived, and to generate an output value of 0 otherwise. The sequences may be overlapping. Use D flip flops to implement a sequence recognizer for the sequence 0,1,0,0. Here assume the order of arrival of the bits is from left-to-right.

   Example: the sequence of input values 1,0,0,1,0,0,1,0,0,1 would produce the output sequence: 0,0,0,0,0,1,0,0,1,0



Latches



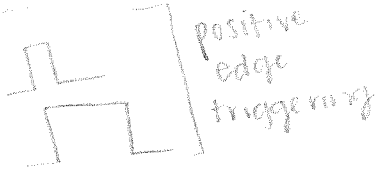
S	R	Q^+
0	0	Q
0	1	0
1	0	1
1	1	UNSTABLE

start@0

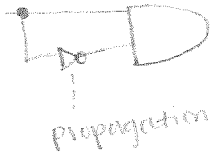
J	K	Q^+
0	0	Q
0	1	0
1	0	1
1	1	$\bar{Q}$

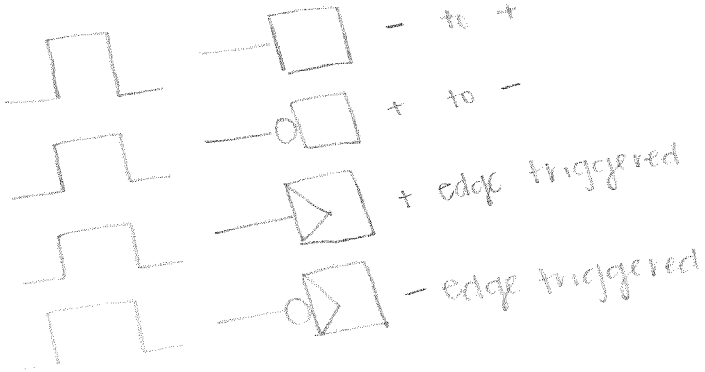
data

D	Q
0	0
1	1



Clock used by flipflops





Excitation Table

Q	Q^+	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Stay at 0

Switch to 1

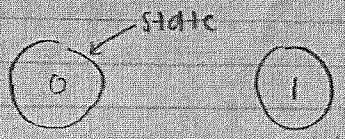
SOP $(ab)+(bc)$ Σ

POS $(a+b)(a+b)$ Π

Chloe Stapleton

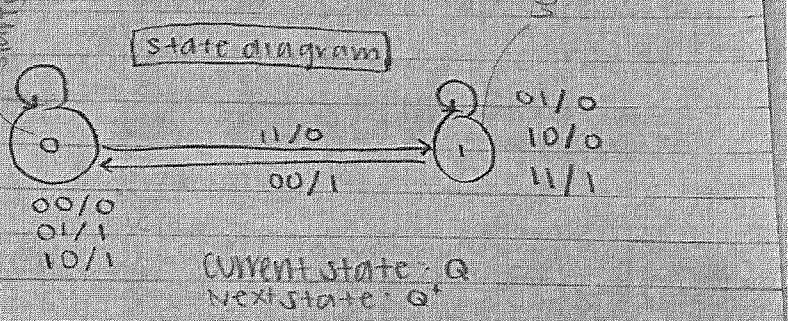
2/13/20

Chapter 1 - Digital Logic p 17



carry-in

inputs / outputs



A	B	S
0	0	
0	1	
1	0	
1	1	

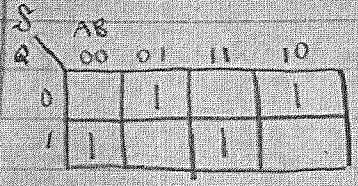
inputs / output

Current state: Q

Next state: Q^+

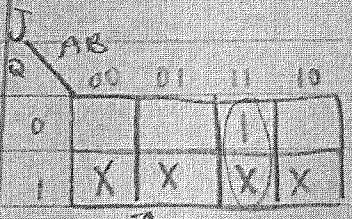
State Table

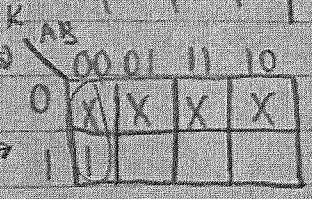
Q	A	B	Q^+	S	J	K
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0



$S = Q \oplus (A \oplus B)$

(see full adder design)

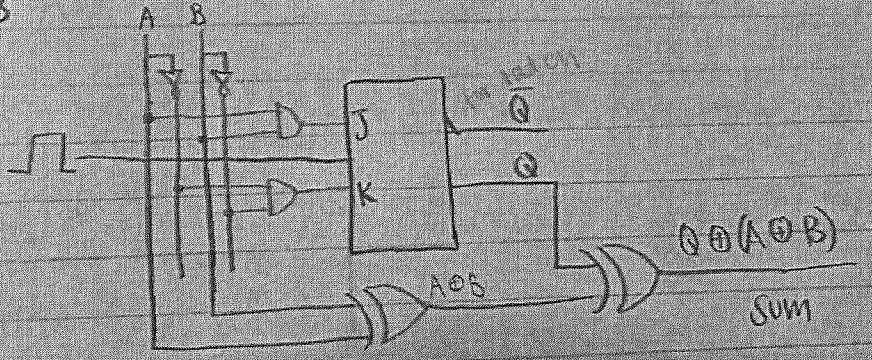




least amt of var ? don't cares

$J = AB$ $K = \overline{A}\,\overline{B}$

here's the circuit



Problem #1

A serial (binary) adder

- *accepts 2 numbers as input
- * Sum
- * JK

Start @ ∅

J	K	Q^+
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

J	K	Q^+
1	1	$\overline{Q}$
1	0	1
0	1	0
0	0	Q

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

STATE TABLE

Carry in Q	A	B	Carry out Q^+	Sum
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

J	K
0	0
0	0
0	0
1	0
0	1
1	0
0	1
1	0

J	K
0	0
0	0
0	0
0	0
0	0
0	0
1	1
1	1

$S = \{a, b\}$

$PS = \{\{\emptyset\}, \{a\}, \{b\}, \{a, b\}\}$

$\emptyset \leftrightarrow 0$

$1 \leftrightarrow 1$

$- \leftrightarrow$ set comp

$+ \leftrightarrow \cup$

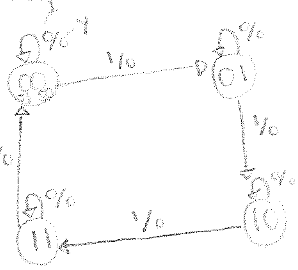
$\cdot \leftrightarrow \cap$

problem #1 of Test 1 will be on final

Problem #2 Module-n synchronous counter

design when n=4 using JK flipflops

(A) Draw a state diagram



if 0 = input ∴ 0 = output

if 1 = input ∴ 0 = output

(B) Derive the state table

S1	S0	I	S1⁺ (next state)	S0⁺ (next state)	Y	J1 (S1→S1⁺)	K1 (S1→S1⁺)	JØ (S0→S0⁺)	KØ (S0→S0⁺)
0	0	0	0	0	0	Ø	X	Ø	X
0	0	1	0	1	0	Ø	X	1	X
0	1	0	0	1	0	Ø	X	X	Ø
0	1	1	1	0	0	1	X	X	1
1	0	0	1	0	0	X	Ø	Ø	X
1	0	1	1	1	0	X	Ø	1	X
1	1	0	1	1	0	X	Ø	X	Ø
1	1	1	0	0	1	X	1	X	1

$Y = S1 \cdot S0 \cdot I$

(C) Determine type of latch

Q	Q⁺	J	K
0	0	Ø	X
0	1	1	X
1	0	X	1
1	1	X	0

(D) derive an input equation

J1

S1 \ S0 I	00	01	11	10
0			1	
1	X	X	X	X

$J1 = S0 \cdot I$

K1

S1 \ S0 I	00	01	11	10
0	X	X	X	X
1			1	

$K1 = SØ \cdot I$

J0

S1 \ S0 I	00	01	11	10
0		1	X	X
1		1	X	X

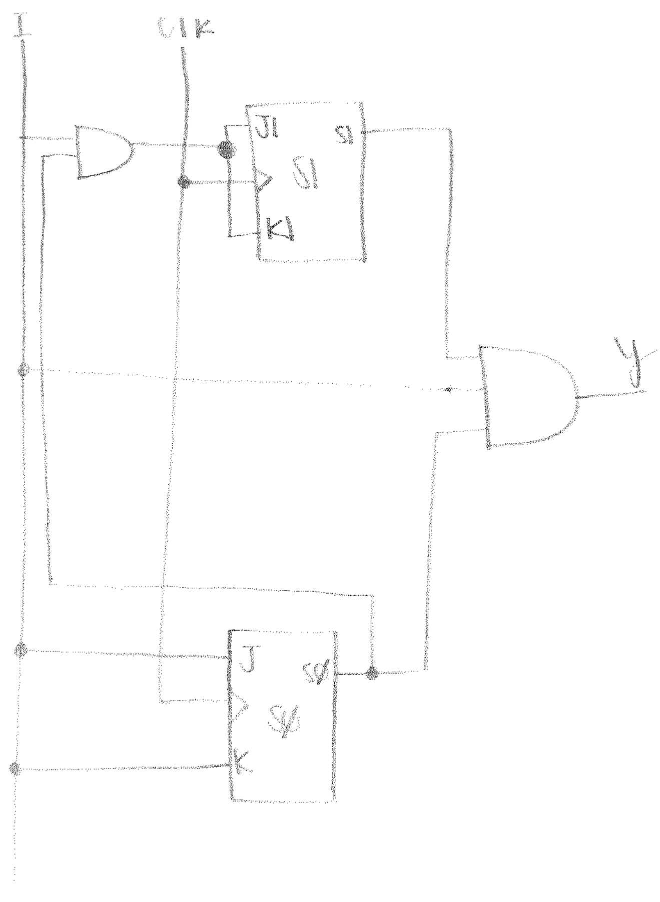
$J0 = 1$

K0

S1 \ S0 I	00	01	11	10
0	X	X	1	
1	X	X	1	

$KØ = 1$

(E) Draw the circuit



Problem #4

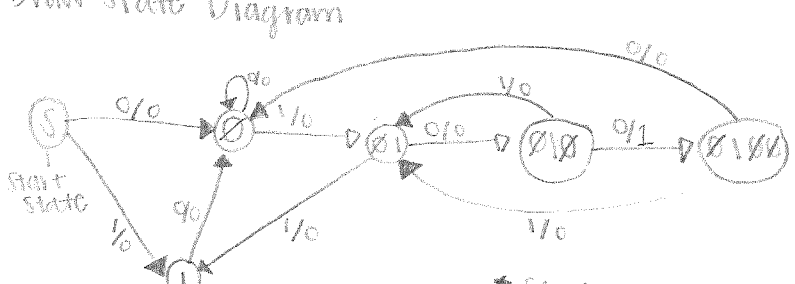
Sequence Recognizer

if (0 1 0 0 is input) : {output = output value}
else {output = x}

* they can overlap

goal: 0 1 0 0

(A) Draw State Diagram



* Crux of the problem *

Binary

B2 B1 B0	
0 0 0	S0 ↔ S
0 0 1	S1 ↔ '0'
0 1 0	S2 ↔ '1'
0 1 1	S3 ↔ '01'
1 0 0	S4 ↔ '010'
1 0 1	S5 ↔ '0100'

(B) Draw a state table

B2	B1	B0	I	B2+	B1+	B0+	Y	Next State D2	D1	D0
0	0	0	0	0	0	1	0			
0	0	0	1	0	1	0	0			
0	0	1	0	0	0	1	0			
0	0	1	1	0	1	1	0			
0	1	0	0	0	0	1	0			
0	1	0	1	0	1	0	0	= B2+	B1+	B0+
0	1	1	0	1	0	0	0			
0	1	1	1	0	1	0	0			
1	0	0	0	1	0	1	0			
1	0	0	1	0	1	1	0			
1	0	1	0	0	0	1	0			
1	0	1	1	0	1	1	0			
1	1	0	0	X	X	X	X			
1	1	0	1	X	X	X	X			
1	1	1	0	X	X	X	X			
1	1	1	1	X	X	X	X			

(C) Determine type of latch

D flip flop

(D) derive input equation

D2

B2 B1 \ B0 I	00	01	11	10
00				
01				1
11	X	X	X	X
10	1			

$D2 = B1B0\bar{I} + B2\bar{B0}\bar{I}$

D1 = you work it

D0 = you don't work it

Y

B2 B1 \ B0 I	00	01	11	10
00				
01				
11	X	X	X	X
10	1			

$Y = B2\bar{B0}\bar{I}$

Shefali Emmanuel

no ch1 in back
ch2 in back

Logic Circuits:

1. Combinational- input will give you the same output
2. Sequential- input can give you a different output every time bc it's like a memory box
    a. output is determined by 2 input values & current state

* dont to min sop & pos in the same K map *

4 corners L & R side

test is like assignments

State of the circuit = 0 or 1

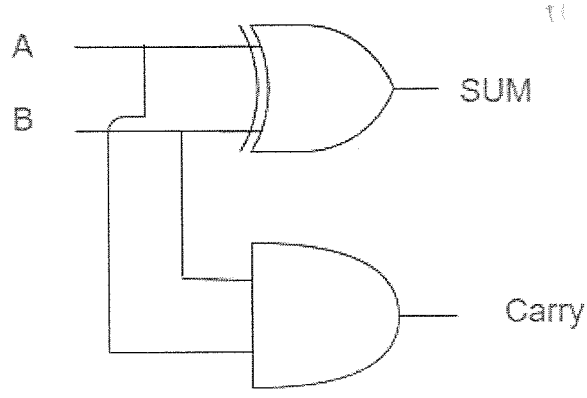
Truth Table > Logical Function > Canonical Expression > Circuit Diagram

1-bit Half Adder

- Two input Values A,B
- Represents 1 bit addition
- Sum(A,B) = $\Sigma(1,2)$
- Sum(A,B) = $\pi(0,3)$

No sequential circuit taking de bugging to the ↓ level

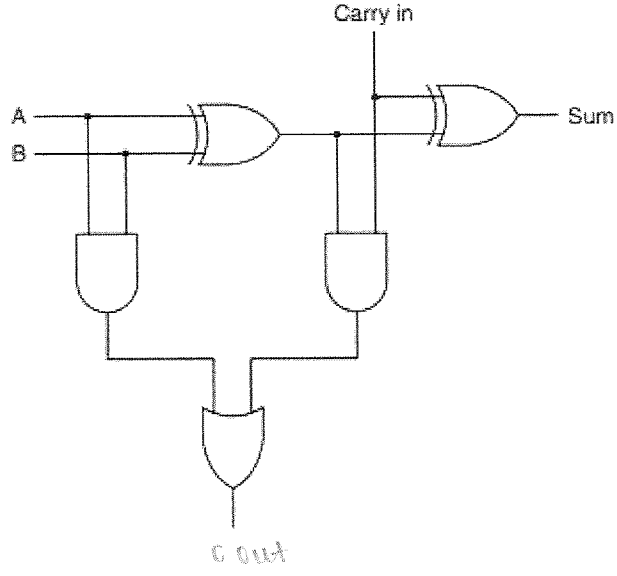
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



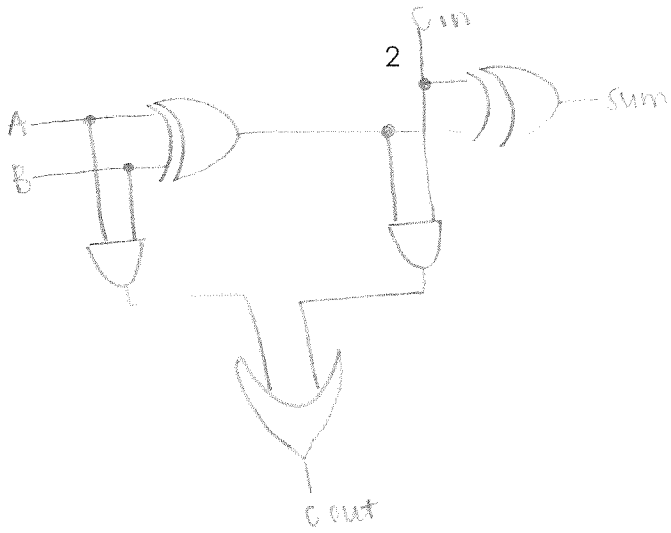
1-bit Full Adder

- 3 inputs A,B,C_{in}
- Sum(A,B,C_{in}) = $\Sigma(1,2,4,7)$
- Sum(A,B,C_{in}) = $\pi(0,3,5,6)$

A	B	Cin	Sum	Carry
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



Logical Operators (Boolean Expressions)

- AND (“.”): returns true if all values are true

A	B	A x B
0	0	0
0	1	0
1	0	0
1	1	1

only T when 1 x 1 AND

- OR (“+”): returns true if any value is true

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

only F when 0 OR 0

- NOT (“~”): returns the compliment/ opposite of the value

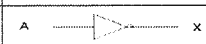
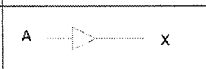
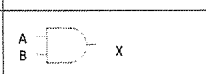
Canonical Expressions

- Sum Of Products (**SOP**):
  - Denotes which values in a standard Truth Table are 1
  - Ex: F(x,y,z) = Σ(1,4,5,6)
    - (~x * ~y * z) + (x * ~y * ~z) + (x * ~y * z) + (x * y * ~z)
- Product Of Sums (**POS**):
  - Denotes which values in a standard truth table are 0
    - Compliment all values that are normally 1

- Ex: f(x,y,z) = π(0,2,3,7)
  - (x + y + z)*(x + ~y + z) * (x + ~y + ~z)*(~x + ~y + ~z)

Section 3. Logic Gates and Digital Circuits

One way to implement digital circuits is to build them from simpler logic circuits known as ***logic gates***. On the following page we list some of the fundamental logic gates that we will use in this chapter, and indeed in the rest of the course. All are combinational circuits. Rather than describe the input-output relationship with cumbersome truth-tables, however, we use Boolean expressions instead.

Name	Symbol	Expression
inverter		$X = \bar{A}$
buffer		X = A
AND		X = A•B or X = AB
OR		X = A + B
NAND		$X = \overline{A \cdot B}$
NOR		$X = \overline{A + B}$
XOR		$X = A\bar{B} + \bar{A}B$ (also denoted A⊕B)

Stronger signal amplifier

Can implement any circuit

same val = 0
diff val = 1

anything can be considered LOGICALLY EQUIVALENT if they both generate the same truth table or define the same logic function

Boolean Algebra

A mathematical system < B, +, *, ~> where B is a set and

- + :
- * :
- ~ : B → B

DUAL with respect to . and + : if 1 relationship is true then the relationship is derived by interchanging the . and +

Examples that satisfies definition:

- set B = {a, b ,c}
- power set PS = {{empty}, {a}, {b}, {c}, {ab}, {bc}, {ac}, {abc}}
- + ↔ U(nion)
- * ↔ ∩ (intersection)

- ~ ↔ compliment
- 0 = { } (null set)
- 1 = {a, b, c}

Circuit Simplification (less # of gates == less cost) && Gate Homogeneity (use just NAND or just NOR)

Circuit Simplification #1: Boolean Algebra Relations

Boolean Algebra Rules:

1. Commutative
   X + Y = Y + X ; X*Y = Y *X
2. Associative
   (X+Y) + Z = X + (Y + Z) ; (X*Y) * Z = X *(Y * Z)
3. Distribution
   X + (Y*Z) = (X+Y) * (X + Z) ; X * (Y + Z) = (X * Y) + (X*Z)

   With distinguished Elements 0 and 1:
4. X + 0 = X

   X * 1 = X
5. X + ~X = 1

   X * ~X = 0

Rules for all Boolean Algebras:

- Idempotency
  - X + X = X
  - X * X = X
- X + 1 = 1

  X * 0 = 0
- Absorption 1
  - X + XY = X
  - X(X+Y) = X
- Absorption 2:
  - X + ~XY = X + Y
  - X(~X + Y) = XY
- DeMorgan's Law:
  - ~(X+Y) = ~X * ~Y ; ~(X*Y) = ~X + ~Y

Circuit Simplification #2: Karnaugh Maps

- An arrangement of cells, each representing a combination of variables in a truth table
  - Logically Adjacent
    - 2 squares of 1 are ____ if the product terms they represent are the same expect 1 variable (where they are complimentary)

    &&
  - Physically Adjacent

Ex: Σ(1,3,4) = ~A * C + A* ~B * ~C

A/BC	00	01	11	10
0	0	1	1	0
1	1	0	0	0

Grouping Rules:

- Singleton: 4 variables
- Couple: 3 variables
- Group of 4: 2 variables
- Group of 8: 1 variable
- ALL: no variables

Values of cells in Karnaugh Maps

AB/CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Circuit Simplification #3: Quine-McKluskey Tables

- minimal SOP for 6+ Variables

Integrated Circuits

Small Scale Integrated Circuits (SSI)	1 - a few 10's of gates
Medium Scale Integrated Circuits (MSI)	a few 10's - a few 100's of gates
S Large Scale Integrated Circuits (SLSI)	a few 100's - a few 100,000's of gates
Very Large Scale Integrated Circuits (VLSI)	> 100,000 of gates

Examples of Combinatorial Circuits:

1. Full Adder

   Represent a 1-bit adder

   Represent a 2-bit adder

   Represent a 2-bit adder using 1-bit adders

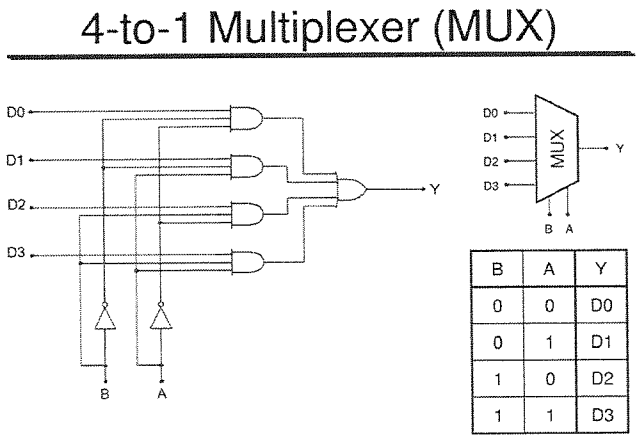
   a. Ripple Carry Adders
      i. Adder carry out is connected to the carry out of the previous adder so that as one completes it "ripples" into the next one
2. Shifters
   a. C : input to control direction of shift
      i. 0 left shift
      ii. 1 right shift
   b. Shifts input 1 bit by placing a 0 from either right or left

C	D1	D0	S1	S0
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

S1 = ~CD0

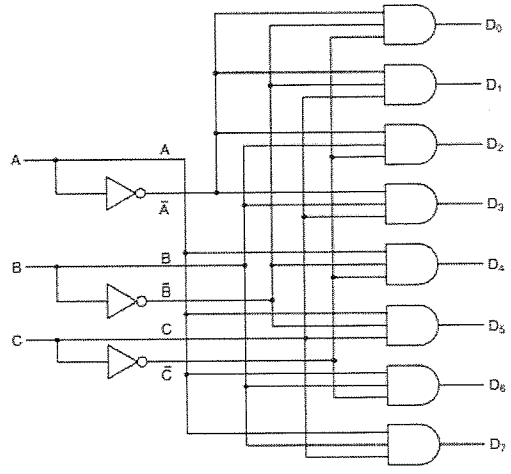
S0 = CD1

3. Multiplexers
   a. 2^n x 1 mux receives 2^n inputs, and selects one input as the output based on the value of the selection lines



B	A	Y
0	0	D0
0	1	D1
1	0	D2
1	1	D3

4. Decoders
   a. n x 2^n decoder receives n input lines and sets the value of one 2^n output lines to 1, and the rest to 0
   b. Enable: controls the output of decoders
      i. Outputs 0 if disabled
      ii. Outputs normally if enabled
      iii. Enabled low (denoted as a circle): flips operation of enable



5. ALU

a. definition

F1	F0	Operation
0	0	A AND B
0	1	~B
1	0	A OR B
1	1	A + B + carry in

Fancy Circuit Stuff

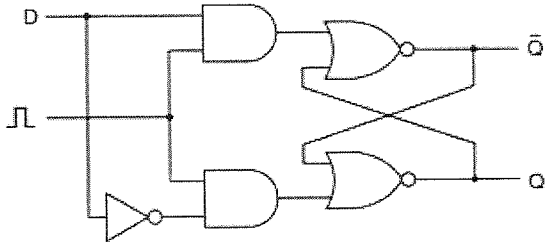
- Asynchronous sequential circuit: output depends on order that input signals change
    - Latches are Asynchronous, can change as soon as inputs can
- Synchronous sequential circuit: memory values can change only at discrete instances of time
    - Flip-flops are synchronous, triggered by an edge
    - Clocks emit a series of 0-1 pulses with a precise length and interval between pulses
    - Clock pulses allow circuits to only change when clock pulse arrives
    - Clock frequency: how often the clock pulses
        - Measured in Hertz; 1 hertz = 1 cycle per second
- Memory element: can maintain a given value indefinitely

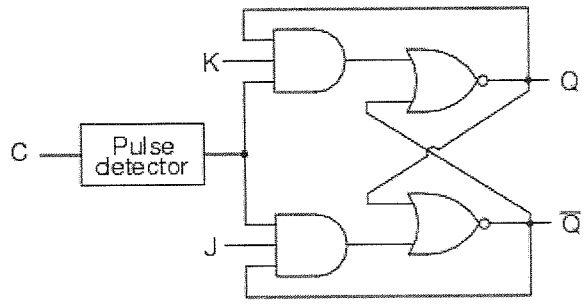
Latches (input conditions) & Flip Flops (control signals)

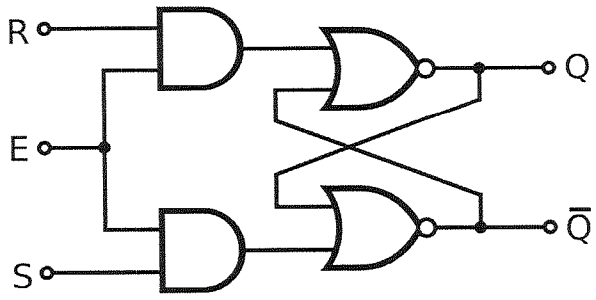
S	R	Q+
0	0	Q
0	1	0
1	0	1
1	1	N/A

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	~Q

D	Q
0	0
1	1







*All diagrams are of flip-flops, pothering really likes his flip flops with and gates. The clocks are the input signal that is not an R, S, K, J, or D for the diagrams

- Propagation Delay: delay caused by gate operations
- Latches:
  -  changes negative to positive
  -  changes positive to negative
- Flip-Flops
  -  positive edge triggered
    - Only changes state when input becomes high
  -  negative edge triggered
    - Only changes state when input becomes low
- Excitation Table: shows minimum inputs required to generate next state when current state is known

JK Flip Flop Excitation Table

Q	Q+	j	k
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

D Flip Flop Excitation Table

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

*X: Don't care what the value is

*Q: current value
*Q+: desired value

Design Procedures for Sequential Circuits:

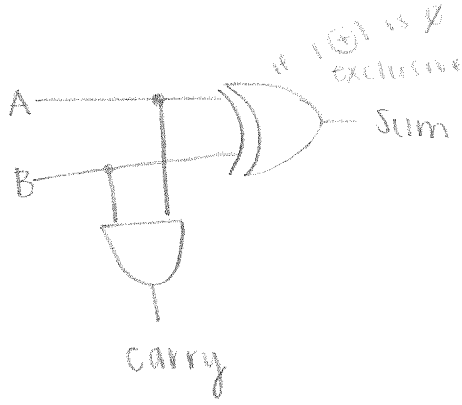
1. Draw state diagram
    a. Circle = state of flip flops represented by binary number
    b. arc= state change based on input/output
2. Derive state table from state diagram
3. Determine appropriate latch/flip flop and corresponding input values needed to cause state changes
4. Derive an equation for each flip-flop
5. Draw the circuit

Logic Circuits: digital circuits w/ T or F

- **Combinational**
  - output values are UNIQUELY DETERMINED by input
- **Sequential**
  - contains num elements $\neq 1$ AND state of circuit UNIQUELY determined by inputs & state
    - Asynchronous
    - Synchronous (button)

1/2 Adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



if 1 ⊕ 1 is 0 exclusive only accepts 1

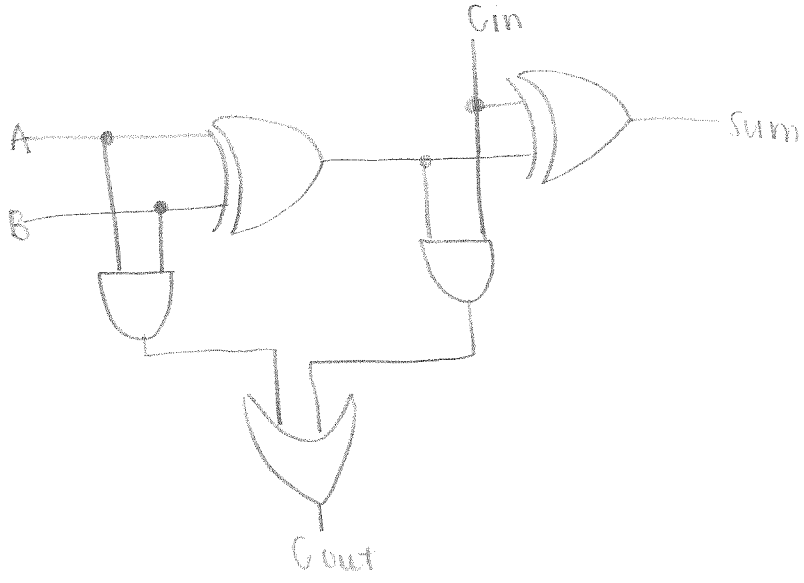
Sum(A,B)

$= \sum_1 (1,2)$

$= \prod_0 (0,3)$

Full Adder

A	B	Cin	Sum	Carry
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



LOGICAL OPERATORS

① x AND

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

② + OR

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

③ NOT

Canonical Expressions

what value are

SOP

$f(x,y,z) = \sum(1,2)$

$= \bar{x}\bar{y}z + \bar{x}y\bar{z}$

1

POS

$f(x,y,z) = \prod(0,3)$

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

∅

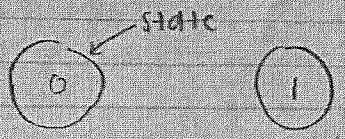
SOP $(ab)+(bc)$ Σ

POS $(a+b)(a+b)$ Π

Chloe Stapleton

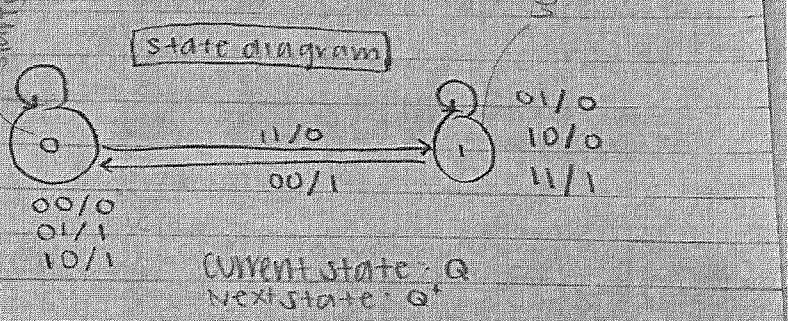
2/13/20

Chapter 1 - Digital Logic p 17



carry-in

inputs / outputs



A	B	S
0	0	
0	1	
1	0	
1	1	

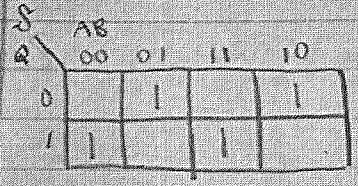
inputs / output

Current state: Q

Next state: Q^+

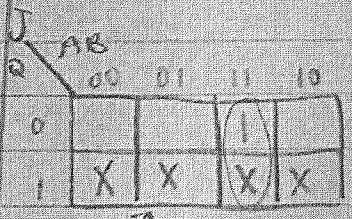
State Table

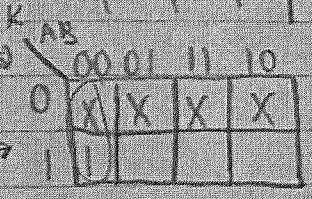
Q	A	B	Q^+	S	J	K
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0



$S = Q \oplus (A \oplus B)$

(see full adder design)

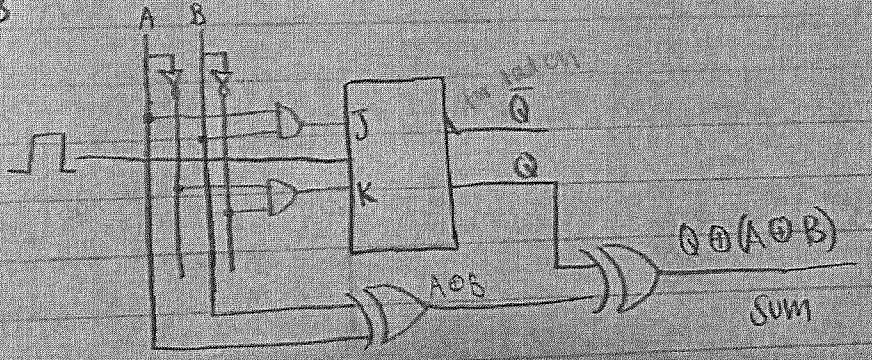




least amt of var ? don't cares

$J = AB$ $K = \overline{A}\,\overline{B}$

here's the circuit



Problem #1

A serial (binary) adder

- *accepts 2 numbers as input
- * Sum
- * JK

Start @ ∅

J	K	Q^+
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

J	K	Q^+
1	1	$\overline{Q}$
1	0	1
0	1	0
0	0	Q

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

STATE TABLE

Carry in Q	A	B	Carry out Q^+	Sum
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

J	K
0	0
0	0
0	0
1	0
0	1
1	0
0	1
1	0

J	K
0	0
0	0
0	0
0	0
0	0
0	0
1	1
1	1

$S = \{a, b\}$

$PS = \{\{\emptyset\}, \{a\}, \{b\}, \{a, b\}\}$

$\emptyset \leftrightarrow 0$

$1 \leftrightarrow 1$

$- \leftrightarrow$ set comp

$+ \leftrightarrow \cup$

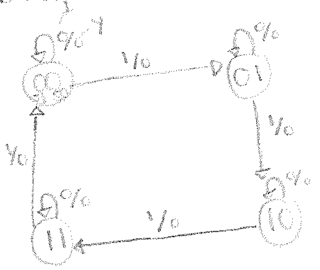
$\cdot \leftrightarrow \cap$

problem #1 of Test 1 will be on final

Problem #2 Modulo-n synchronous counter

design when n=4 using JK flipflops

Ⓐ Draw a state diagram



if 0 = input ∴ 0 = output

if 1 = input ∴ 0 = output

Ⓑ Derive the state table

S1	S0	I	$S1^+$	$S0^+$	Y	J1	K1	J0	K0
0	0	0	0	0	0	Ø	X	Ø	X
0	0	1	0	1	0	Ø	X	1	X
0	1	0	0	1	0	Ø	X	X	Ø
0	1	1	1	0	0	1	X	X	1
1	0	0	1	0	0	X	Ø	Ø	X
1	0	1	1	1	0	X	Ø	1	X
1	1	0	1	1	0	X	Ø	X	Ø
1	1	1	0	0	1	X	1	X	1

(next state: $S1^+$, $S0^+$; S1→S1⁺: J1, K1; S0→S0⁺: J0, K0)

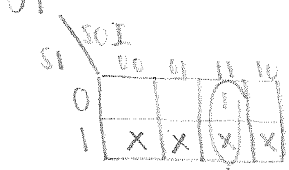
Y = S1·S0·I

Ⓒ Determine type of latch

Q	Q+	J	K
0	0	Ø	X
0	1	1	X
1	0	X	1
1	1	X	0

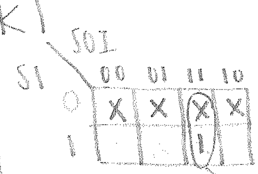
Ⓓ derive an input equation

J1



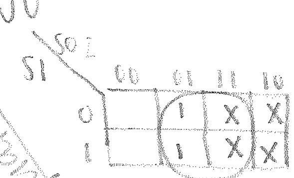
J1 = S0·I

K1



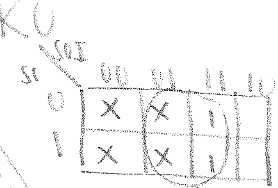
K1 = SØ·I

J0



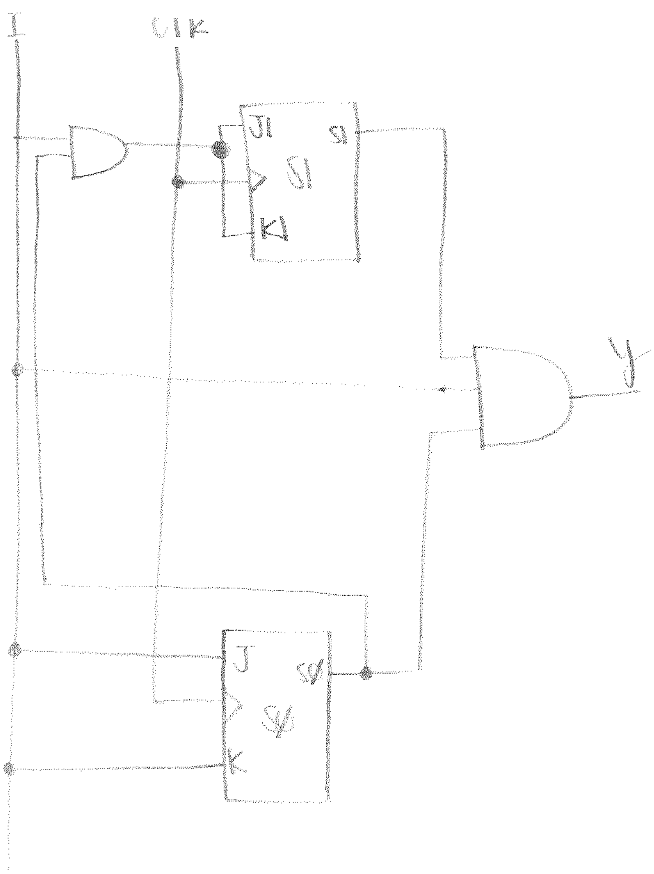
J0 = 1

K0



KØ = I

Ⓔ Draw the circuit



Problem #4

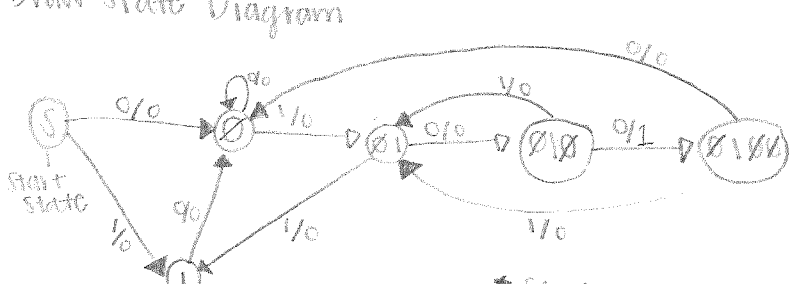
Sequence Recognizer

if (0 1 0 0 is input) : {output = output value}
else {output = x}

* they can overlap

goal: 0 1 0 0

(A) Draw State Diagram



* Crux of the problem *

Binary

B2 B1 B0	
0 0 0	S0 ↔ S
0 0 1	S1 ↔ '0'
0 1 0	S2 ↔ '1'
0 1 1	S3 ↔ '01'
1 0 0	S4 ↔ '010'
1 0 1	S5 ↔ '0100'

(B) Draw a state table

B2	B1	B0	I	B2+	B1+	B0+	Y	Next State D2	D1	D0
0	0	0	0	0	0	1	0			
0	0	0	1	0	1	0	0			
0	0	1	0	0	0	1	0			
0	0	1	1	0	1	1	0			
0	1	0	0	0	0	1	0			
0	1	0	1	0	1	0	0	= B2+	B1+	B0+
0	1	1	0	1	0	0	0			
0	1	1	1	0	1	0	0			
1	0	0	0	1	0	1	0			
1	0	0	1	0	1	1	0			
1	0	1	0	0	0	1	0			
1	0	1	1	0	1	1	0			
1	1	0	0	X	X	X	X			
1	1	0	1	X	X	X	X			
1	1	1	0	X	X	X	X			
1	1	1	1	X	X	X	X			

(C) Determine type of latch

D flip flop

(D) derive input equation

D2

B2 B1 \ B0 I	00	01	11	10
00				
01				1
11	X	X	X	X
10	1			

$D2 = B1B0\bar{I} + B2\bar{B0}\bar{I}$

D1 = you work it

D0 = you don't work it

Y

B2 B1 \ B0 I	00	01	11	10
00				
01				
11	X	X	X	X
10	1			

$Y = B2\bar{B0}\bar{I}$

Shefali Emmanuel

no ch1 in back
ch2 in back

Logic Circuits:

1. Combinational- input will give you the same output
2. Sequential- input can give you a different output every time bc it's like a memory box
    a. output is determined by 2 input values & current state

* dont to min sop & pos in the same K map *

4 corners L & R side

test is like assignments

State of the circuit = 0 or 1

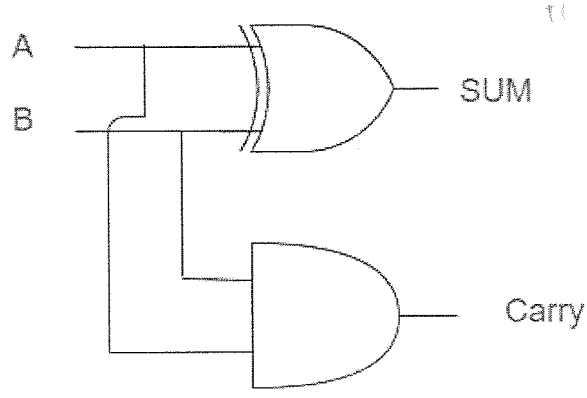
Truth Table > Logical Function > Canonical Expression > Circuit Diagram

1-bit Half Adder

- Two input Values A,B
- Represents 1 bit addition
- Sum(A,B) = $\Sigma(1,2)$
- Sum(A,B) = $\pi(0,3)$

No sequential circuit taking de bugging to the ↓ level

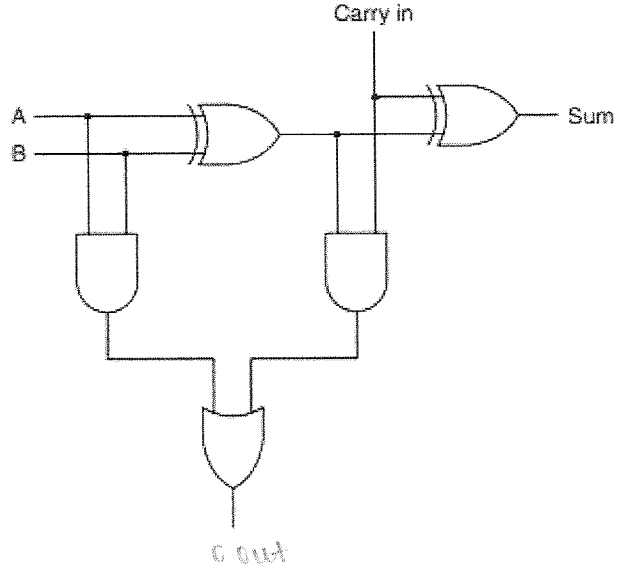
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



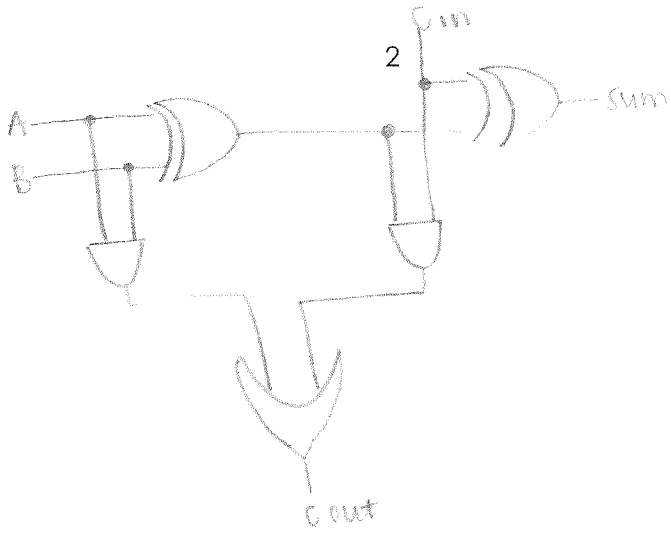
1-bit Full Adder

- 3 inputs A,B,C_{in}
- Sum(A,B,C_{in}) = $\Sigma(1,2,4,7)$
- Sum(A,B,C_{in}) = $\pi(0,3,5,6)$

A	B	Cin	Sum	Carry
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



Logical Operators (Boolean Expressions)

- AND (“.”): returns true if all values are true

A	B	A x B
0	0	0
0	1	0
1	0	0
1	1	1

only T when 1 x 1 AND

- OR (“+”): returns true if any value is true

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

only F when 0 OR 0

- NOT (“~”): returns the compliment/ opposite of the value

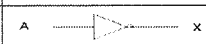
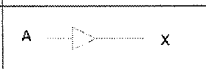
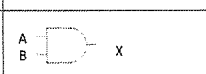
Canonical Expressions

- Sum Of Products (**SOP**):
  - Denotes which values in a standard Truth Table are 1
  - Ex: F(x,y,z) = Σ(1,4,5,6)
    - (~x * ~y * z) + (x * ~y * ~z) + (x * ~y * z) + (x * y * ~z)
- Product Of Sums (**POS**):
  - Denotes which values in a standard truth table are 0
    - Compliment all values that are normally 1

- Ex: f(x,y,z) = π(0,2,3,7)
  - (x + y + z)*(x + ~y + z) * (x + ~y + ~z)*(~x + ~y + ~z)

Section 3. Logic Gates and Digital Circuits

One way to implement digital circuits is to build them from simpler logic circuits known as ***logic gates***. On the following page we list some of the fundamental logic gates that we will use in this chapter, and indeed in the rest of the course. All are combinational circuits. Rather than describe the input-output relationship with cumbersome truth-tables, however, we use Boolean expressions instead.

Name	Symbol	Expression
inverter		$X = \bar{A}$
buffer		X = A
AND		X = A•B or X = AB
OR		X = A + B
NAND		$X = \overline{A \cdot B}$
NOR		$X = \overline{A + B}$
XOR		$X = A\bar{B} + \bar{A}B$ (also denoted A⊕B)

Stronger signal amplifier

Can implement any circuit

same val = 0
diff val = 1

anything can be considered LOGICALLY EQUIVALENT if they both generate the same truth table or define the same logic function

Boolean Algebra

A mathematical system < B, +, *, ~> where B is a set and

- + :
- * :
- ~ : B → B

DUAL with respect to . and + : if 1 relationship is true then the relationship is derived by interchanging the . and +

Examples that satisfies definition:

- set B = {a, b ,c}
- power set PS = {{empty}, {a}, {b}, {c}, {ab}, {bc}, {ac}, {abc}}
- + ↔ U(nion)
- * ↔ ∩ (intersection)

- ~ ↔ compliment
- 0 = { } (null set)
- 1 = {a, b, c}

Circuit Simplification (less # of gates == less cost) && Gate Homogeneity (use just NAND or just NOR)

Circuit Simplification #1: Boolean Algebra Relations

Boolean Algebra Rules:

1. Commutative
   X + Y = Y + X ; X*Y = Y *X
2. Associative
   (X+Y) + Z = X + (Y + Z) ; (X*Y) * Z = X *(Y * Z)
3. Distribution
   X + (Y*Z) = (X+Y) * (X + Z) ; X * (Y + Z) = (X * Y) + (X*Z)

   With distinguished Elements 0 and 1:
4. X + 0 = X

   X * 1 = X
5. X + ~X = 1

   X * ~X = 0

Rules for all Boolean Algebras:

- Idempotency
  - X + X = X
  - X * X = X
- X + 1 = 1

  X * 0 = 0
- Absorption 1
  - X + XY = X
  - X(X+Y) = X
- Absorption 2:
  - X + ~XY = X + Y
  - X(~X + Y) = XY
- DeMorgan's Law:
  - ~(X+Y) = ~X * ~Y ; ~(X*Y) = ~X + ~Y

Circuit Simplification #2: Karnaugh Maps

- An arrangement of cells, each representing a combination of variables in a truth table
  - Logically Adjacent
    - 2 squares of 1 are ____ if the product terms they represent are the same expect 1 variable (where they are complimentary)

    &&
  - Physically Adjacent

Ex: Σ(1,3,4) = ~A * C + A* ~B * ~C

A/BC	00	01	11	10
0	0	1	1	0
1	1	0	0	0

Grouping Rules:

- Singleton: 4 variables
- Couple: 3 variables
- Group of 4: 2 variables
- Group of 8: 1 variable
- ALL: no variables

Values of cells in Karnaugh Maps

AB/CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Circuit Simplification #3: Quine-McKluskey Tables

- minimal SOP for 6+ Variables

Integrated Circuits

Small Scale Integrated Circuits (SSI)	1 - a few 10's of gates
Medium Scale Integrated Circuits (MSI)	a few 10's - a few 100's of gates
S Large Scale Integrated Circuits (SLSI)	a few 100's - a few 100,000's of gates
Very Large Scale Integrated Circuits (VLSI)	> 100,000 of gates

Examples of Combinatorial Circuits:

1. Full Adder

   Represent a 1-bit adder

   Represent a 2-bit adder

   Represent a 2-bit adder using 1-bit adders

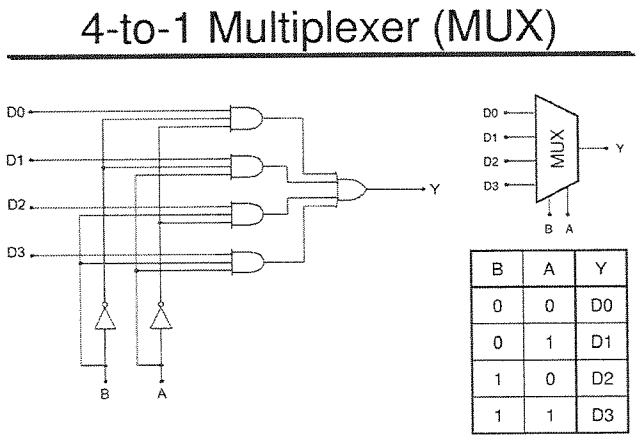
   a. Ripple Carry Adders
      i. Adder carry out is connected to the carry out of the previous adder so that as one completes it "ripples" into the next one
2. Shifters
   a. C : input to control direction of shift
      i. 0 left shift
      ii. 1 right shift
   b. Shifts input 1 bit by placing a 0 from either right or left

C	D1	D0	S1	S0
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

S1 = ~CD0

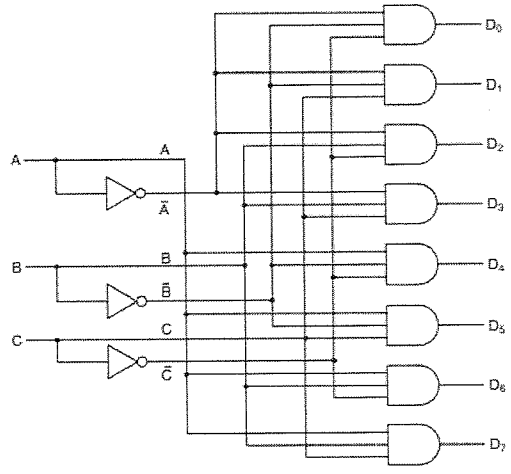
S0 = CD1

3. Multiplexers
   a. 2^n x 1 mux receives 2^n inputs, and selects one input as the output based on the value of the selection lines



B	A	Y
0	0	D0
0	1	D1
1	0	D2
1	1	D3

4. Decoders
   a. n x 2^n decoder receives n input lines and sets the value of one 2^n output lines to 1, and the rest to 0
   b. Enable: controls the output of decoders
      i. Outputs 0 if disabled
      ii. Outputs normally if enabled
      iii. Enabled low (denoted as a circle): flips operation of enable



5. ALU

a. definition

F1	F0	Operation
0	0	A AND B
0	1	~B
1	0	A OR B
1	1	A + B + carry in

Fancy Circuit Stuff

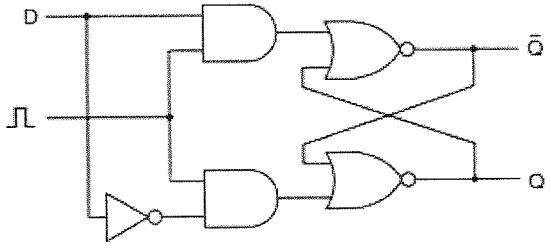
- Asynchronous sequential circuit: output depends on order that input signals change
    - Latches are Asynchronous, can change as soon as inputs can
- Synchronous sequential circuit: memory values can change only at discrete instances of time
    - Flip-flops are synchronous, triggered by an edge
    - Clocks emit a series of 0-1 pulses with a precise length and interval between pulses
    - Clock pulses allow circuits to only change when clock pulse arrives
    - Clock frequency: how often the clock pulses
        - Measured in Hertz; 1 hertz = 1 cycle per second
- Memory element: can maintain a given value indefinitely

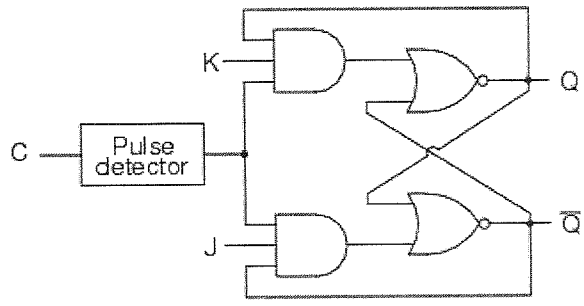
Latches (input conditions) & Flip Flops (control signals)

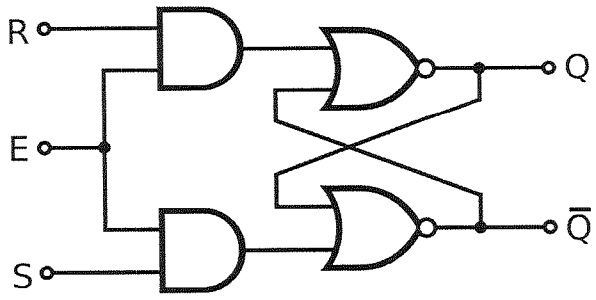
S	R	Q+
0	0	Q
0	1	0
1	0	1
1	1	N/A

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	~Q

D	Q
0	0
1	1







*All diagrams are of flip-flops, pothering really likes his flip flops with and gates. The clocks are the input signal that is not an R, S, K, J, or D for the diagrams

- Propagation Delay: delay caused by gate operations
- Latches:
  -  changes negative to positive
  -  changes positive to negative
- Flip-Flops
  -  positive edge triggered
    - Only changes state when input becomes high
  -  negative edge triggered
    - Only changes state when input becomes low
- Excitation Table: shows minimum inputs required to generate next state when current state is known

JK Flip Flop Excitation Table

Q	Q+	j	k
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

D Flip Flop Excitation Table

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

*X: Don't care what the value is

*Q: current value
*Q+: desired value

Design Procedures for Sequential Circuits:

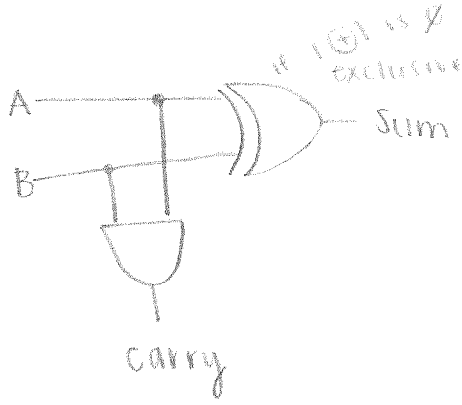
1. Draw state diagram
    a. Circle = state of flip flops represented by binary number
    b. arc= state change based on input/output
2. Derive state table from state diagram
3. Determine appropriate latch/flip flop and corresponding input values needed to cause state changes
4. Derive an equation for each flip-flop
5. Draw the circuit

Logic Circuits: digital circuits w/ T or F

- **Combinational**
  - output values are UNIQUELY DETERMINED by input
- **Sequential**
  - contains num elements $\neq 1$ AND state of circuit UNIQUELY determined by inputs & state
    - Asynchronous
    - Synchronous (button)

1/2 Adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



if 1 ⊕ 1 is 0 exclusive only accepts 1

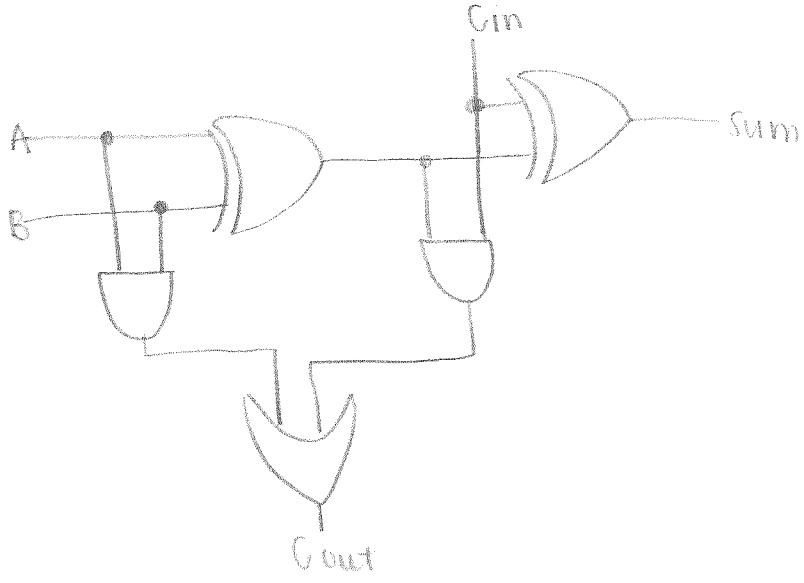
Sum(A,B)

$= \sum_1 (1,2)$

$= \prod_0 (0,3)$

Full Adder

A	B	Cin	Sum	Carry
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



LOGICAL OPERATORS

① x AND

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

② + OR

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

③ NOT

Canonical Expressions

what value are

SOP

$f(x,y,z) = \sum(1,2)$

$= \bar{x}\bar{y}z + \bar{x}y\bar{z}$

1

POS

$f(x,y,z) = \prod(0,3)$

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

∅

Boolean Algebra

mathematical system $\langle B, +, *, \sim \rangle$ (+ and *)

$B \to \{\}\{\}$

$\sim : B \to B$

$+, * : B \times B \to B$

DUAL

with respect to $\cdot$ & $+$

if 1 relationship is T then relationship derived by interchanging $\cdot$ & $+$

Examples

$B = \{a, b, c\}$

$PS = \{\{a\}, \{b\}, \{c\}, \{\emptyset\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$

$+ \leftrightarrow \cup$ union

$* \leftrightarrow \cap$ intersection

$\sim \leftrightarrow$ compliment

$\emptyset \leftrightarrow$ null set

$1 \leftrightarrow \{a, b, c\}$

Circuit Simplification

↓ gates = ↓$$$

Gate Homogeneity

use just NAND || just NOR

#1 Boolean Algebra Relations

BA RULES

Commutative

$X + Y = Y + X$

$X * Y = Y * X$

Associative

$(X + Y) + Z = X + (Y + Z)$

$(X * Y) * Z = X * (Y * Z)$

Distribution

$X + (Y * Z) = (X + Y) * (X + Z)$

$X * (Y + Z) = (X * Y) + (X * Z)$

$X + 0 = X$ $\quad X * 1 = X$

$X + \overline{X} = 1$ $\quad X * \overline{X} = 0$

$X \underset{\cdot}{+} 0 = X$ $\quad X \underset{+}{\cdot} 1 = X$ $\quad X \underset{+}{\cdot} \overline{X} = 0$ $\quad X \underset{\cdot}{+} \overline{X} = 1$

Idempotency

$X + X = X$

$X * X = X$

$X + 1 = 1$

$X \cdot 0 = 0$

Absorption 1

$X + XY = X$

$X(X + Y) = X$

Absorption 2

$X + \overline{X} Y = X + Y$

$X(\overline{X} + Y) = XY$

DeMorgan's Law $\quad \overline{X + Y} = \overline{X}\,\overline{Y}$ $\quad \overline{XY} = \overline{X} + \overline{Y}$

#2 Karnaugh Maps

for 3 to 4 vars

an arrangement of cells, each representing a combo of var in TT

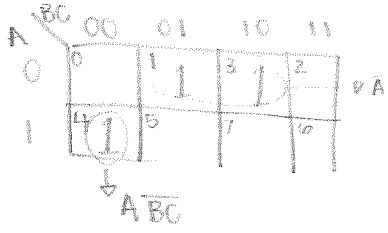
differ by one literal

LOGICALLY product terms are the same Except 1 var, where they are complementary

xy & xz

PHYSICALLY right next to each other in a traditional TT

EX/ $\sum(1,3,4) = \bar{A}C + A\bar{B}\bar{C}$



Grouping Rules

singleton	4
couple	3
group 4	2
group 8	1
all	NONE

variables

	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

values of cells

#3 Quine McKluskey Tables

Minimal SOP for 6+ var

TruthTables > Logical Functions > Canonical Expression > Circuit

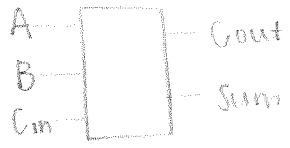
$\sum$ & $\prod$ POS / SOP

Examples of Combinatorial Circuits

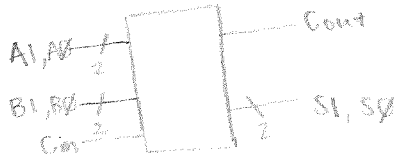
① Full Adder

↑ hardware - ↑ efficiency

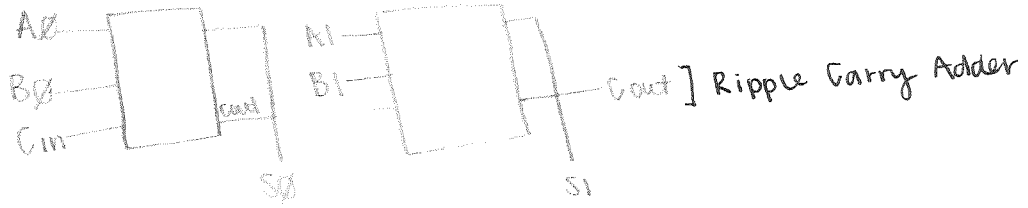
Represent a 1 bit adder



Represent a 2 bit adder



Implement a 2 bit adder using 2 1 bit adders



] Ripple Carry Adder

② Shifters

a circuit that has 2 inputs D0, D1 } control C (determines direction of shift)

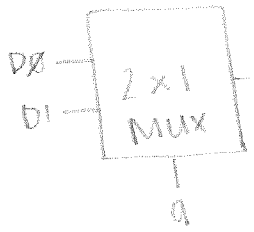
Shifts input by 0 or 1 L or R

C	D1	D0	S1	S0
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

L shift
add R

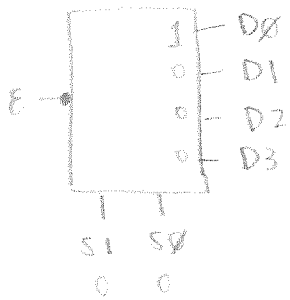
③ Multiplexer

$2^n \times 1$ MUX recieves 2^n inputs & selects 1 input based on selection line val



④ Decoders

$n \times 2^n$ recieves n inputs & sets the value of one 2^n to 1 & rest to 0



Enable controls output
└ disable = everything 0

⑤ ALU

F1	F0	Operation
0	0	A AND B
0	1	$\overline{B}$
1	0	A OR B
1	1	A + B + Cin

Asynchronous

output depends on order of input signals

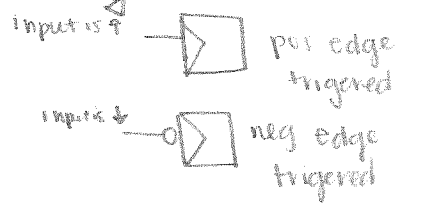
LATCHES — change inputs when change | level triggering

Synchronous

mem values change @ discrete instances of time allowed

FLIPFLOP allowed by edge

CLOCKS continuous may to do it

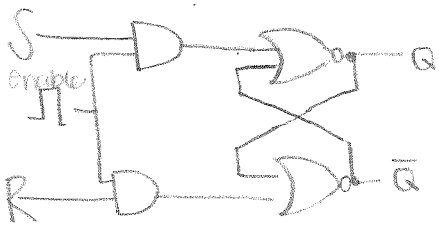


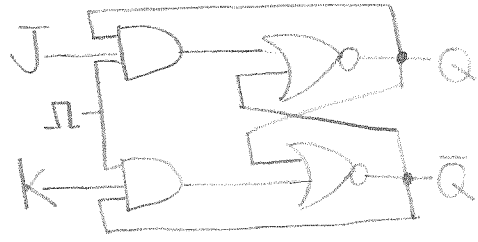
neg edge triggered

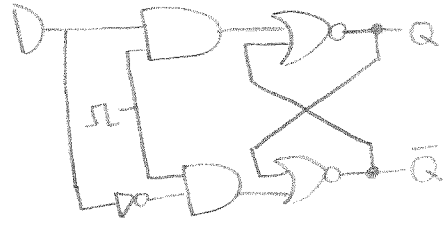
S	R	Q+
0	0	Q
0	1	0
1	0	1
1	1	NOT STABLE

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	$\bar{Q}$

D	Q+
0	0
1	1







Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Excitation Table

↓

Shows min inputs required for next state when current state is shown

battery [red = positive
black = negative

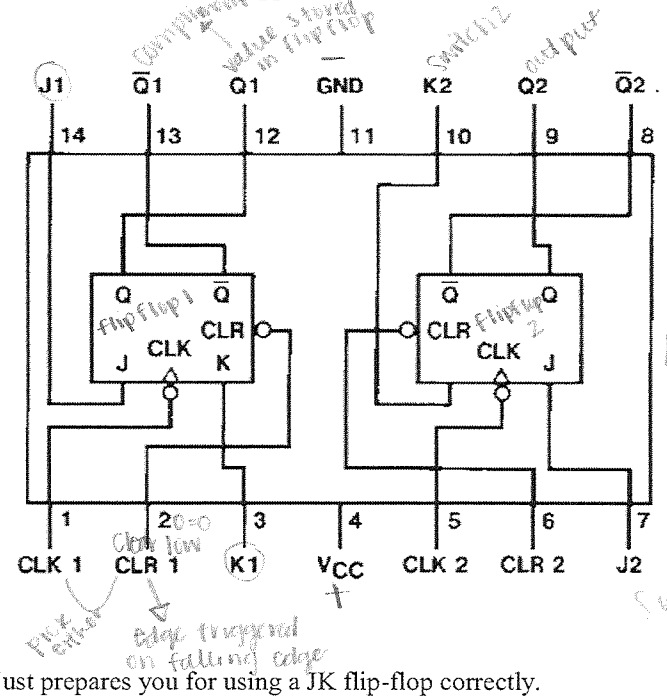
long side of light goes in negative

Feb 17

CSCI350 - Spring, 2020
In-class Laboratory - Working with the 7473 chip

Shown below is a copy of the connection diagram of the 7473 chip, which contains two ***negative edge-triggered*** JK flip-flops. Here we will address the role of the pins found on the leftmost flip-flop (flip-flop 2)and the chip overall.

- Pin 4 gets assigned to + (Vcc) and Pin 11 is assigned to - (ground-GND). **This is different from what we have seen on the other chips we have used** (where pin 14 was Vcc and pin 7 was GND).
- Pin 14 is where the J input is found and pin 3 is where the K input is found.
- Pin 2 is where the clock input signal gets sent(for our purposes, we will use a pushbutton, although this is not ideal as we will see). Note the ^ on the flip-flop diagram with CLK above it. This indicates that this flip-flop is edge-triggered. Moreover, the small circle below is means that the edge-triggering is on the falling edge.
- Pin 12 gives us the value currently stored in the flip-flop and pin 13 gives us its complement.
- Pin 2 represents an asynchronous "clear-low" setting. This means that when a 0 appears on this input (the "low" in "clear-low") the flip-flop will immediately (up to propagation delay) be placed in the 0 (cleared) state. This setting of the value is independent of what is happening on the clock.



Lab activities:

A. The following activity just prepares you for using a JK flip-flop correctly.

1. Set up a pushbutton switch and use an LED to show that it is working.
   a. Insert one of the pushbutton switch somewhere in the leftmost area of your circuit board (be sure it does not straddle the center channel) and connect one of the pins so it receives a + signal. One of the small grey wires is good for this.
   b. Insert an LED near the leftmost side of the circuit board), placing the shorter connector into the − rail, and inserting the longer connectorsomewhere in the column next to the one receiving the + signal.
   c. The column next to the one where you inserted the LED (but not the one receiving the + signal) should be the same column as the second pin in the pushbutton switch. columns with a resistor. Connect both.
   a. Apply power to the circuit board and then depress the button on the push button switch. The LED should light when the button is pushed and go out when the button is released.

make circuit w/ cdm music

2. Now we will do something similar with two of the four switches (say switches 1 and 2) on one of the DIP (dual in-line package) switch packages.
   a. Insert one of these DIP switch packages on the far right side of the circuit board, but here be sure that the package straddles the center channel). I am going to assume that the ON side of the switch package is on top as you look at the circuit board from above.
   b. Using either a small brown wire, or small grey wire, connect pins 8 and 7 (the ON sides for switches 1 and 2 respectively) so they receive a + signal.
   c. On the bottom side of the circuit board insert two LEDs so the shorter ends receive - signals and so the longer ends are not in any of the columns used by the pins of the switch package.. It is also best to leave at least two columns between the LEDs.
   d. Now run one resistor between pins 1 of the switch package (aligned with switch 1 of the package) and the positive connector of one of the LEDs. Then do the same with the pin for switch 2 and the other LED.
   e. Apply power to the circuit board and then move the levers for switched 1 and 2 so they are in the ON position. The LED associated with each switch should light. When the levers are moved back, each LED should go off.

B. Now we are ready to start working with one of the flip-flops on the 7473 chip

1. Insert a 7473 chip into the circuit board (say somewhere in the middle). You are going to wire this so that flip-flop 2 receives its J and K signals from the switches you hooked up in step 2 of Activity A above, and which displays the value stored in the flip-flop using an LED that is wired to the appropriate Q output (pin 9) for the flip-flop 2. Don't forget to use a resistor in connecting the Q output to the long connector of the LED. Finally connect the output of the pushbutton switch to pin 5, which is the appropriate CLK input for the flip-flop 2.

   J → pin 7 switch 1
   K → pin 10 switch 2
   Q → pin 9
   switch output → pin 5

   Before testing your set-up you will need to connect the Vcc pin to + and the GND to -. Be sure you use the correct pins. Also, you should connect pin 6 to receive a + signal. This is the asynchronous CLEAR signal for flip-flop 2, which is activated low and automatically store a 1 in the flip-flop when a "0" signal appears on the line. Since we don't want to do this in this lab, we need to keep a "1" signal on the line.

   4 & 6 to +
   11 to −
   └ send to a +

2. Test your wiring of the flip-flop by activating the switches so the J input receives a "1" and the "K" input receives a "0". When you press the pushbutton switch, you should see the LED for representing the value currently being stored in the flip-flop change from 0 to 1 **when the button is released**. The releasing of the button gives the 1-to-0 transition that represents the falling edge of a clock pulse.

   J → +
   K → −
   button 0 to 1

3. Further test the flip-flop by using various settings of 00, 01, and 10 (**but not 11**) for the J and K values and checking that your flip-flop value assume the value you expect.

   RS latch?

   J K
   0 0
   0 1
   1 0

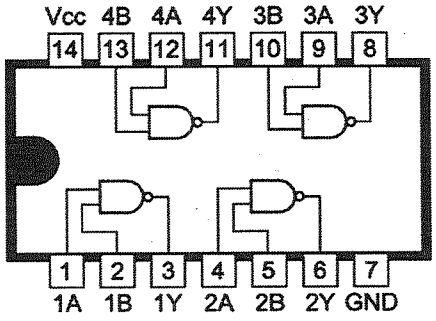
4. Finally place both switches ON, so J and K both receive input values of 1. Now press the pushbutton a number of times, and examine the output of the flip-flop. While you may be expecting the LED to alternately go on and off each time you release the pushbutton, this may not happen because of the phenomenon known as "switch bounce."

   J K
   1 1

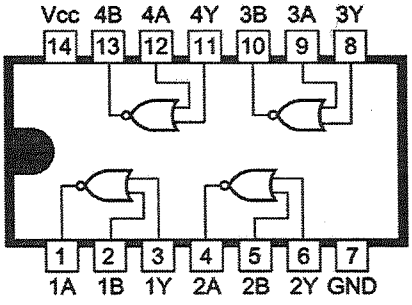
C. For the final activity of this lab, you should wire the circuit for the serial adder that we designed in class. Use flip-flop 2 to represent the current value of the carry-in for the circuit. Note, there are two ways you can implement the expression $\overline{A}\,\overline{B}$. You can use two inverters and a 2-input AND gate, or you can use just a 2-input NOR gate (7402 chip) since by DeMorgan's law $\overline{A + B} = \overline{A}\,\overline{B}$.

- Note, you will have to wire up the SUM output as part of this activity. You can use the XOR implementation for the full-adder sum to do this and using the 7486 chip. The carry-in part will be the current value of the JK flip-flop. You should connect this to an LED so you can check that it is working as expected.

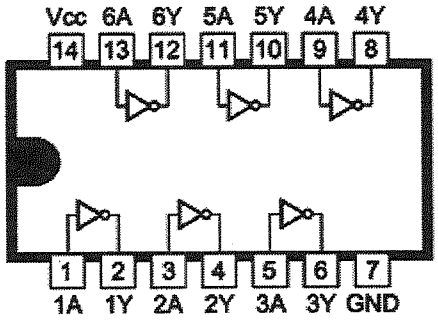
7400 Quad 2-input NAND Gates

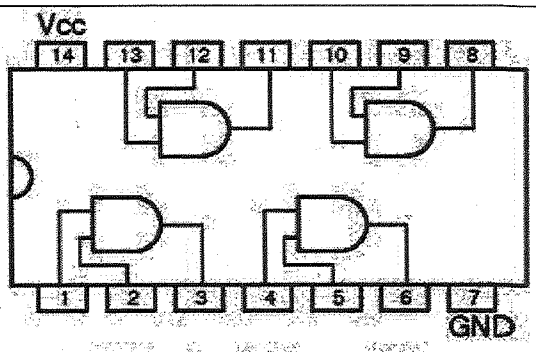


7402 Quad 2-input NOR Gates



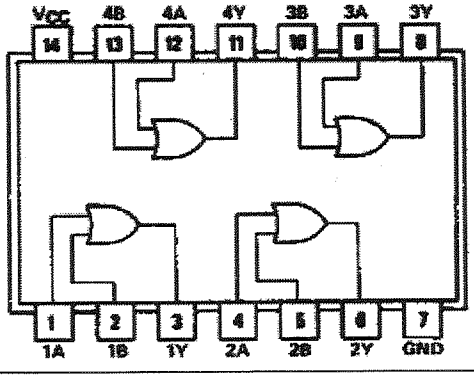
7404 Hex Inverters



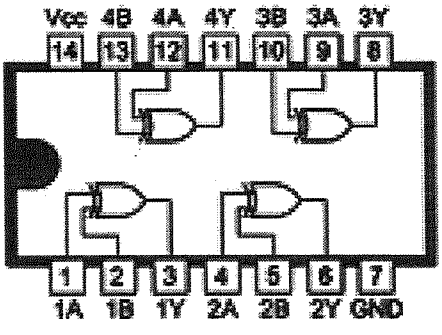


7408 Quad 2 Input AND

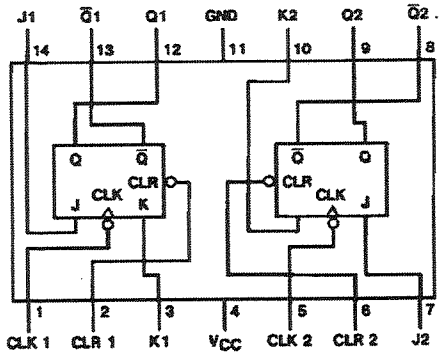
7432



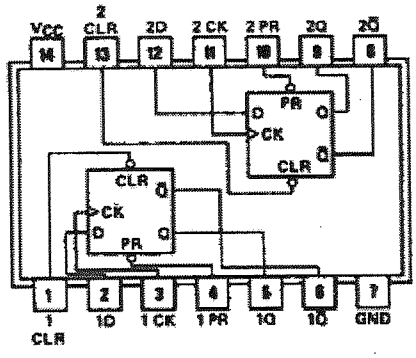
7486 Quad 2-input ExOR Gates



7473


J1
Q̄1
Q1
GND
K2
Q2
Q̄2
14
13
12
11
10
9
8
Q
Q̄
CLR
CLK
J
K
Q̄
Q
CLR
CLK
J
1
2
3
4
5
6
7
CLK 1
CLR 1
K1
VCC
CLK 2
CLR 2
J2

7474


VCC
2
CLR
2D
2 CK
2 PR
2Q
2Q̄
14
13
12
11
10
9
8
D
PR
Q
CK
Q̄
CLR
CLR
Q
CK
D
PR
Q
1
2
3
4
5
6
7
1
CLR
1D
1 CK
1 PR
1Q
1Q̄
GND

Feb 27

CHAPTER 2
MEMORY ORGANIZATION

Section 1. Organization of Main Memory

Memory is one of the major components of a computer system. More precisely, we mean a relatively large, fast memory used for program and data storage during the computer's operation. To help satisfy the desire for speed, this memory is organized so that every storage unit can be accessed with a speed that is constant and independent of the unit's location. Memory with such an organization is known as *random access memory* (or RAM).

- The memory's storage units, known as *words*, normally have the same size; a word comprised of 8 bits is called a *byte*.

- Communication between a memory unit and its environment is achieved through control lines, address selection lines, data input lines and data output lines.

CL a. The *control lines* specify (at least) whether a read or write operation is to be performed; R / W

AL b. The *address lines* select a particular word out of all those available; location

IL c. The *input lines* supply information to be stored;

OL d. The *output lines* supply the information coming out of memory.

These differences in functions among lines should be viewed logically rather than physically however since in some implementations it is common for some of the lines to be used for several functions (e.g. using the same lines for input and output, using one line for designating a read or write operation; or using the same lines for data and addresses).

Now that we know what a register is we can view RAM (once again, logically, but not necessarily physically) as a collection of registers together with the associated circuitry needed to transfer information in and out of these registers. The main components of memory now become are:

1. the registers. collection of mem cells

2. a decoder for selecting one register out of all those in the memory.

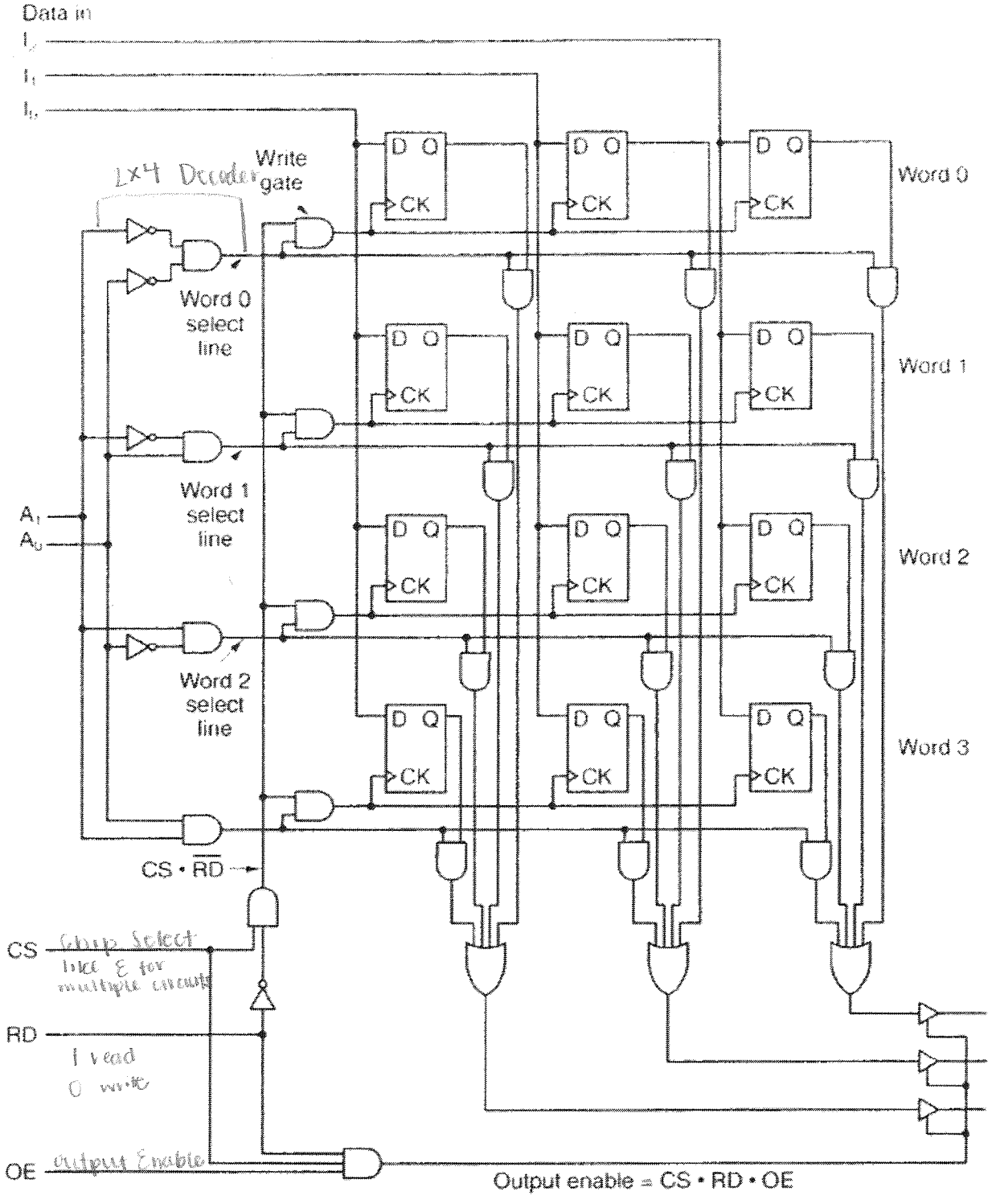
We then need to combine the control lines, address lines, and data lines with these components to make the memory unit function as desired. The diagram on page 2 of these notes gives a possible organization for a small memory of four 3-bit words and the same as that on page 176 of your textbook.

A memory of this type, essentially implemented from latches or flip-flops, is known as a ***static random access memory***, or SRAM. There is another way of implementing memory, known as dynamic RAM, or DRAM, that is actually the way most computers' memory is implemented, that we describe later. Operationally, however, the two technologies are the same.

The details of the registers implementing each word are shown and from this we see these registers are simply parallel-in, parallel-out registers with the loading of data being controlled by simultaneous edge-triggering of each input bit..

The address lines A_1 and A_0 give the binary address (0 - 3) of the word on which a given operation is to be performed, while the control lines CS, RD, and OE whether a read or write operation is to be performed, or neither. In this configuration the control line CS, (for Chip Select, also known as chip enable), in effect disables the operation of the memory unit by preventing both read and write operations. The signal RD is used when a read operation is to be performed; a read operation will be performed when the line has value 1. While a value of 0 on the RD line could suggest a write operation, this is not what happens. Instead a write is performed only when RD = 0 and a value of 1 appears on the OE line (which stands for Output Enable). Likewise, CS must be active (= 1).

4 x 3


Data in
I_2
I_1
I_0
2x4 Decoder
Write gate
Word 0 select line
Word 1 select line
Word 2 select line
A_1
A_0
D Q
CK
Word 0
Word 1
Word 2
Word 3
CS • $\overline{RD}$
CS
Chip Select
Like E for multiple circuits
RD
1 read
0 write
OE
output Enable
Output enable = CS • RD • OE

Feb 27

Kilo	K	1024	2^{10}
Mega	M		2^{20}
Ja	G		2^{30}
Tera	T		2^{40}

2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

1 1 1 1 1 1 1 1 1 1

512 256 128 64 32 16 8 4 2 1

binary digit

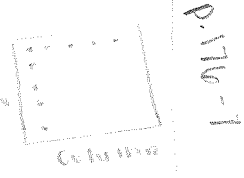
bit

4c96k x 1

4 M ⟷ 2^{22}

RAS = Row Activate

CAS = Column Activate



A curious feature of the memory example is the use of the "buffer looking devices" connecting the four-input OR gates to the output lines O_0, O_1, and O_2. These are examples of what are known as" tri-state buffers" and their purpose is to make the buffer act in the normal manner when a control input is high, but to act like an open circuit when the control input is low. Technically we don't need these tri-state buffers since we show separate lines for input and output, but in practice the same lines are used for both, in which case the tri-state buffers prevent the output values from interfering with the input values on a read operation. The following tables show some of the various tri-state devices available and their operation.

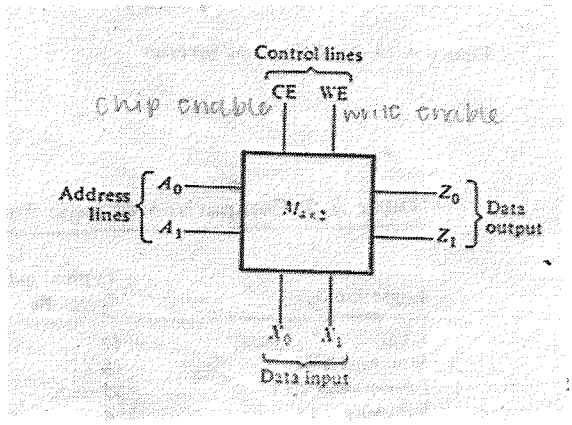
tri-state buffer (active high)		
E	A	X
0	X	HI-Z
1	0	0
1	1	1

tri-state inverter (active high)		
E	A	X
0	X	HI-Z
1	0	1
1	1	0

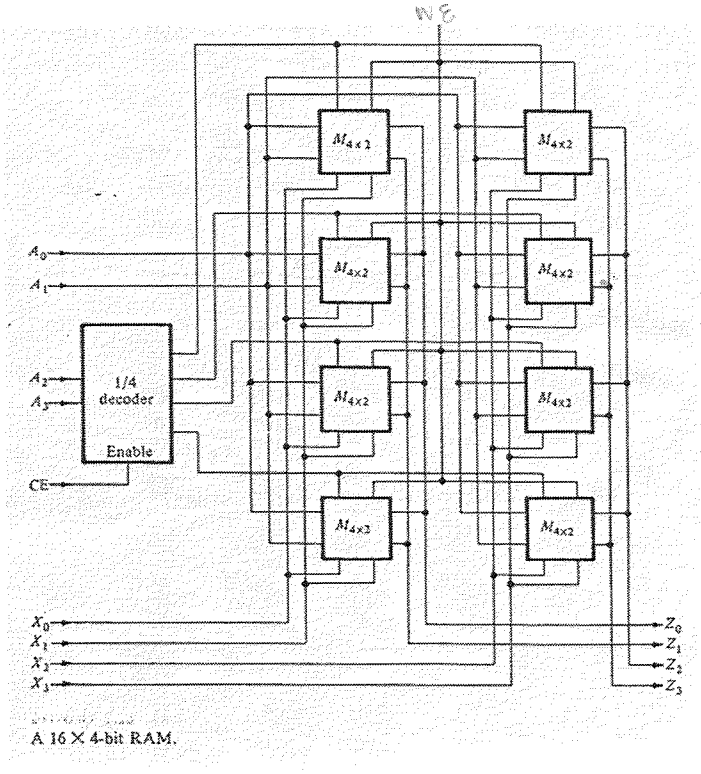
tristate buffer (active low)		
E	A	X
0	0	0
0	1	1
1	X	HI-Z

Forming Larger Memory Units from Smaller Ones

The following diagram shows a 4 × 2 RAM that uses a chip enable (CE) line and a write enable(WE) line to carry out memory operations, with a low signal on WE indicating a read operation.



We can then combine these to form a 16 x 4 RAM as follows:



A 16 × 4-bit RAM.

In another organization, the memory addressing logic regards the address space as two-dimensional rather than linear.

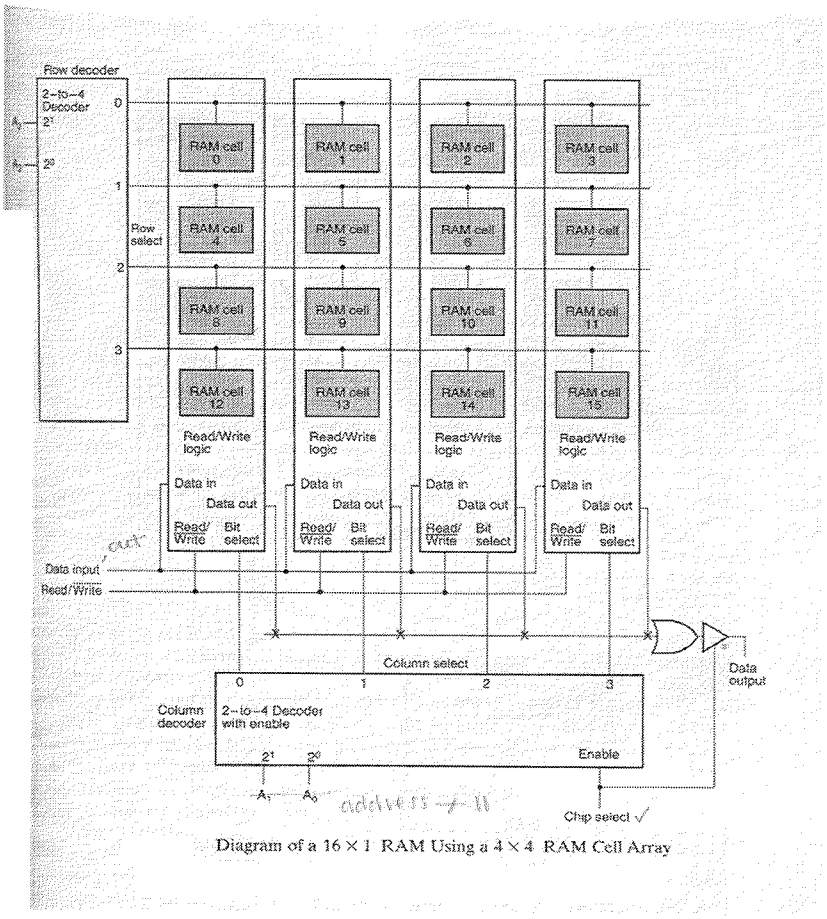


Diagram of a 16 × 1 RAM Using a 4 × 4 RAM Cell Array

Although we do not supply descriptions for all of the lines, by now you should be able to discern their role. You will notice, however, that both the row decoder and the column decoder use address lines labeled A_1 and A_0 rather than employ four address lines A3, A2, A1, A0 and distribute them between the row and column decoders. This is not unusual, as large memories will use the same lines for row and column addresses to save pins and reduce package costs. To distinguish the two types of addresses, a pair of signals called *row access strobe* (RAS) and *column access strobe* (CAS) are also provided to signal the memory that a row or column address is being supplied.

Example:. Consider the 4M × 1 RAM (2×2^{22} words of 1 bit1 each) and 512K × 8 RAM (2×2^{19} words of 8 bits each) diagrammed on page179 of your textbook. Note that each uses 4Mbits, but their internal organization is different as the book explains.

What are some of the ways 4 of the first type can be combined to form larger memory units? Before considering this case

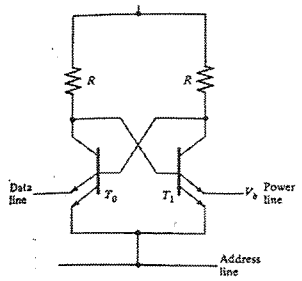
Done in class

Operational Properties of RAM

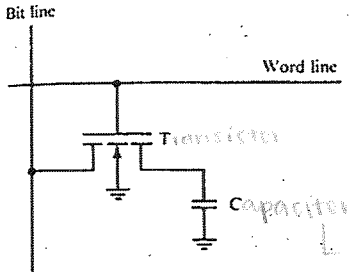
As our discussion above indicated, RAM is usually constructed as an n × m arrangement of *bits*, not an arrangement of n *words* of m bits each.

When fabricated from semiconductors two classes of RAM emerge, depending on the technology used in the construction of each.

1. *static RAM* (SRAM) - Here the bits are implemented essentially as flip-flops.



2. *dynamic RAM* (DRAM) - Here each bit is implemented using a single transistor and a single capacitor.

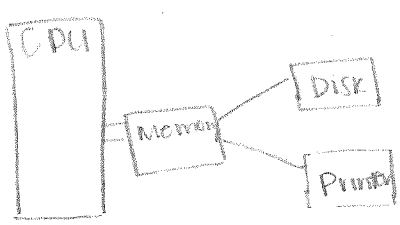


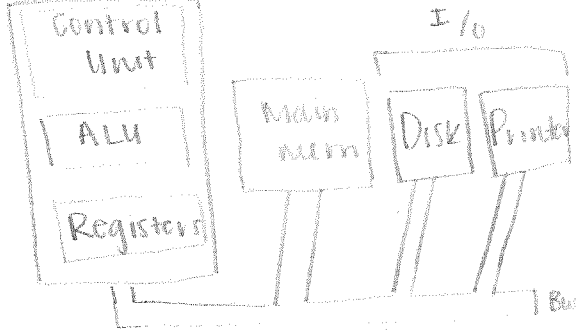
Operationally, they differ in that over time the capacitors used in dynamic RAM lose their charge, requiring that they be periodically "refreshed." At one time, to carry out a read operation, the capacitors must be discharged, destroying the value stored - a phenomenon known as *destructive read-out* (DRO). DRO requires that each read be followed by a restoration of the bit's original value. Static RAM is non-destructive during read operations (NDRO) and requires no refreshing. Both types of memory however are *volatile*, meaning the values stored in each bit will be destroyed if power to the memory is disrupted.

Static RAM is faster than dynamic RAM. Not only does static RAM require less time to complete a read or write operation (a property known as the RAM's *access time*), but as we noted above, static RAM requires neither refreshing nor restoration after a read. One consequence of this NDRO property is that the time which must elapse between successive read or write operations (known as the memory's *cycle time*) is the same as its access time; for dynamic RAM the cycle time is necessarily greater than its access time.

The major advantage of dynamic RAM over static RAM is that a bit of dynamic RAM requires only one transistor and one capacitor so that one can pack significantly more bits of DRAM into a given area than SRAM, and DRAM consumes less power than a bit of static RAM.

Because of the high-bit density achievable with dynamic RAM, it is currently the type used for a computer's main memory. Where greater speed, but fewer bits, are required (e.g. in *cache memories*, or in virtual memory tables) static memory is preferred.





CHAPTER 3
BASIC COMPUTER ORGANIZATION

Section 1. Introduction

In Chapter 1 we studied circuit design at what is commonly referred to as the digital logic (or gate) level. Here the fundamental unit for processing is a binary, digital, electronic signal, and the basic components of circuits are gates. Connections among gates made with individual conducting lines.

As part of our study of digital logic circuits, however, we designed circuits capable of accepting multi-bit inputs, producing multi-bit output, and undergoing multi-bit changes of state. Among some of these circuits are multi-bit gates, multiplexers, decoders, bit-sliced adders, various forms of registers (parallel in/out, counters, shift registers), etc. These circuits are among the fundamental components for a level of circuit design immediately above the digital logic level in a digital system design hierarchy. In recognition of the omnipresence of registers at this level, it is commonly referred to as the *register level* (or *register-transfer level*). In Chapter 2 we discussed the organization of main memory, one of the critical components of a computer system. In Chapter 4 we shall discuss the organization and design of a simple central processing unit,

In this chapter we will examine how these fundamental system components can be brought together in a basic computer system. Missing from our discussion, however, will be any detailed discussion of any of the input/output (I/O or IO) devices or secondary storage devicesof a computer system. We do this because there are just too many varieties of such devices. As a result we just note their presence in a system and address other issues as they are pertinent to our discussion. Sections 2.2 and 2.3 of your textbook describe many of these devices and issues involved in transferring data among them and memory. We leave this for you to read to become more familiar with these devices. We will discuss some of the issues involved with data transfers in this chapter, however.

Recall that the central processing unit (CPU) performs many of the processing tasks of the of the computer system and generally controls the system. Memory is used to store programs being executed by the CPU as well as any data immediately needed by, or generated by, the programs. The IO devices generally allow the CPU to interact with the computer's environment (where by "environment" we mean anything other than memory). The overall functioning of a computer system involves the transfer of data among the system components and the exchange of control signals to coordinate these transfers.

While it is possible to organize the connections among the register-level components so that each pair of elements has its own connections, as the number of components increases the number of dedicated connections soon becomes unwieldy. Instead register-level design uses sets of shared connection, known as *buses,* for transferring data to or from an associated set of register-level components. We will discuss buses in more detail in the next section, but for now we show some possible interconnection structures for computer systems.

Section 2. Buses

As we just noted a *bus* is a common electrical pathway between two or more devices. When applied to computer system (and also the design of computer components) buses are intended to transfer all bits of an n-bit word from a specified source to a specified destination, or destinations. Although our emphasis in this chapter is on data transfers at the register level, bus structures are applicable for data transfers at both the register level and at the system level. While data transfers among system level components are generally more complex because of the greater variety of system level components, many of the issues we consider at the register level also exist at the system level and are dealt with similarly.

A bus may be *unidirectional* (capable of word transfers in one direction only) or *bidirectional.* Although there are various approaches to bus design, in any bus design we can identify the following types of lines:

what do we need for a bus??

Tri State Buffer

E = 1 — enable

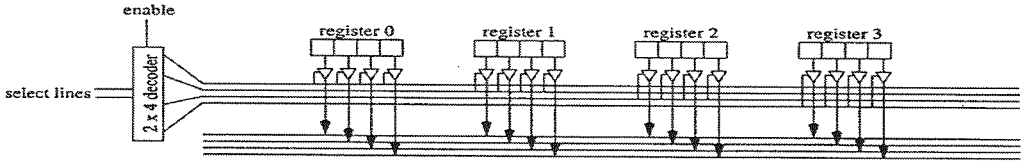
E = 0 — disable

1. *data lines* for carrying the data being transferred between the registers.
2. *address* (or select) *lines* to indicate the source/destination registers of a transfer
3. *control lines* to control access to, and the use of, the data and address lines. Minimally the control lines indicate the direction of the data transfer (from source to bus, or from bus to destination). In addition there may be an additional line/lines to coordinate the activities of a transfer.

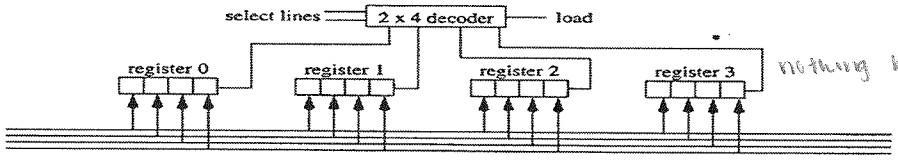
We now indicate how it is possible for multiple devices to be attached to common data lines without causing their respective signals to become mixed. We shall focus exclusively on connecting a number of registers (rather than entire components) to common data lines. This is not unnecessarily restrictive for our discussion since in practice all transfers of data within a computer system take place at the register level.

Implementation of Register-level Buses

The first problem we encounter is: If several potential sources are physically connected to the bus, how can we keep the data from all of them from becoming intermingled when the data from only one source is to be placed on the bus? A solution is to attach the output lines from each device to the data lines of the system bus with *tri-state devices* (typically tri-state buffers or tri-state inverters). We introduced tri-state devices in the previous chapter.

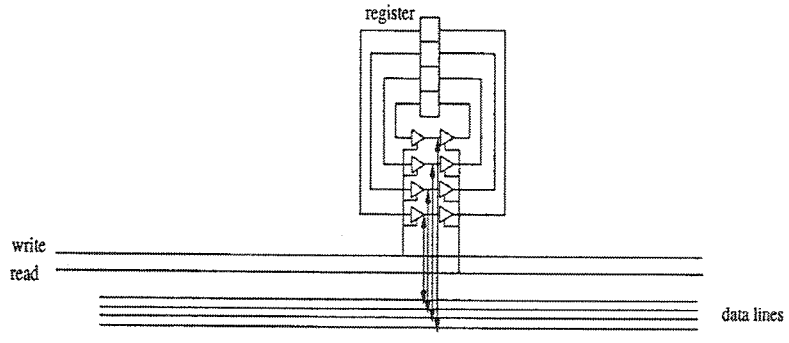


Now we show how it is possible to transfer data from a single register to a bus. Transferring data from a bus to a register can be done without using tri-state devices, as shown below



nothing happens until you enable the decoder to accept info passed to register

For bi-directional transfers, however, a more complex arrangement is required. We show below the connections for one register. For multiple registers the output of as decoder (for register selection) could be ANDed with the write line and with the read line to read from or write to a selected register.



Section 3. Bus Operations

In this section we examine more technical issues related to bus implementations. We begin by making a distinction between those devices that can initiate bus transfers, so-called *bus masters*, and other devices that respond to request – *bus slaves.*

- Some devices may be bus masters in some instances and bus slaves in others.
- See the textbook (page 189) for examples of bus masters and bus slaves.

Bus Clocking – Synchronous and Asynchronous Buses

Buses are classified as either synchronous or asynchronous

- *Synchronous bus* – one of the control lines is a (master) clock line which receives its signals from a clock source (typically a crystal oscillator). All bus activities between devices attached to the bus are coordinated with pulses (clock cycles) on this line and take an integral number of such cycles. Note that typically some activities take place on the rising edge of a clock pulse, while others take place on the falling edge.

  One complication that must be considered in using a synchronous bus is that of *clock skew*. This arises when the lengths of some the paths between two different data elements are sufficiently long that there are delays between the times when a clock signal reaches the different elements. If this delay is significant it may be possible for one element to change and cause input values to a second element to be changed before that element has received the clock signal.

- *Asynchronous bus* – does not use a clock line

Example – Asynchronous bus transfer from a master device to a slave device – a read operation from memory to a CPU.

Again using general concepts, the activities involved in performing a memory read to a CPU over an asynchronous bus are as follows:

1. The CPU places an address on the bus' address lines (ADDR)
2. The CPU indicates the address is in memory (MREQ = 0) and that a read operation will take place (RD = 0).
3. The CPU (as transfer master) asserts a master synchronization signal (MSYNC) to alert the slave device (memory) to commence the transfer
4. Memory reads data from the appropriate word and places it on the bus' data lines.
5. Once the data is on the data lines, memory activates a slave synchronization signal (SSYNC) to let the CPU know that data is available.

Example – Synchronous bus transfer from a master device to a slave device -- a read operation from memory to a CPU.

The accompanying handouts illustrate, and then describe a protocol for transferring synchronously from memory to a cpu. Note that some events occur with rising edges of clock pulses and some with falling edges.

Broadly speaking, the activities involved in performing a memory read to a CPU over a synchronous bus are as follows:

1. The CPU (master) places an address on the bus' address lines (ADDR). This address becomes active with the rising edge of the clock
2. Control lines for the device (in this case $\overline{MREQ}$ and desired operation ($\overline{RD}$) is asserted (low in our cases).

- Note, in some buses there may be special lines such as MEMR (memory read), MEMW (memory write), IOR (IO device read), IOW (IO device write), etc. for this task.

3. The CPU waits for an appropriate number of clock pulses for the memory operation to complete. A "wait state" signal may be activated so the CPU can (continue to) wait.
4. Memory (the slave device) reads data from the appropriate word and places it on the bus' data lines.
5. Once the memory data has been read into a CPU register the CPU (master) can deactivate the read and memory request lines.

Because all of these activities must happen in coordination with clock cycles, the frequency of the master clock, the read/write times for memory, and other timing factors must all be taken into account.

The accompanying handout gives a detailed discussion of a synchronous memory read transfer..

6. Once the CPU has captured the data from the bus it negates the MSYNC, MREQ and RD lines.
7. Once it recognizes that that MSYNC has been deactivated, memory (the slave device) can negate the SSYNC line

Block Transfers

Where it is necessary to transfer multiple words between a master device and a slave device it is possible to modify the synchronous scheme to allow a count value to be transferred to the slave device and then to transfer a block of data values at a time (say one word per clock cycle). This can reduce the time to transfer a block of data significantly compared to transferring the block one word at a time.

Bus Arbitration

Bus arbitration is a mechanism for resolving simultaneous requests from several bus masters for control of the bus for a data transfer. Special bus arbitration circuitry is needed for this, often as a special chip.
In the simplest scheme a master device asserts a bus request line that goes to the bus arbiter. Bus requests from each possible master go to the same request line to the bus arbiter, Upon recognizing that a bus request signal has arrived, the bus arbiter returns a bus grant signal on a separate line to which the bus masters are attached in a daisy chain manner (see page 185 of the text). In this way the device that sent a bus request that is closest to the bus arbiter in this daisy chain arrangement will be given control of the bus.

- In a variation of this arrangement, the bus arbiter may accept signals on several (prioritized) bus request lines, each of which has an associated bus grant line. As in the above case, the devices associated with a given bus request/grant pair are attached to the bus grant line in a daisy-chain manner. Simultaneous requests for a bus are resolved first by the priority of the request line and secondarily (if necessary) by position in the daisy chain.
- Because of the role of a bus arbiter, the above mechanism is known as *centralized bus arbitration*. It is also possible to use a simpler, decentralized scheme in which the devices compete among themselves and monitor a busy line before attempting to gain bus access..

Bus Standards

Computer components such as CPUs, memory, and various IO devices will be manufactured by different companies, each having their own unique internal characteristics. In order that they might communicate among themselves via buses within a computer system, however, various bus standards (or protocols) have been established and used over the years. These standards include not only specifications for the number of address and data lines, and types of control lines present, but also encompass mechanical and electrical specifications so that when the bus is incorporated into printed circuit boards the physical connectors, voltages, and timing signal will all be compatible. Here we review briefly some noteworthy bus standards:

- ISA (Industry Standard Architecture) bus: A bus standard adopted by most of the personal computer industry in the early 1980s. Originally it had 20 address lines and 8 data lines, later expanded to 16 data lines. It used an 8.33 MHz clock. Among its control signals are ones for asserting memory reads, memory

writes, IO reads, IO writes, etc. ISA was extended to 32 bits in the late 80s, giving us Extended Industry Standard Architecture (EISA). The maximum data transfer rate (bandwidth) on ISA was 16.7 MB/sec, and with EISA it was 33.3 MB/sec.

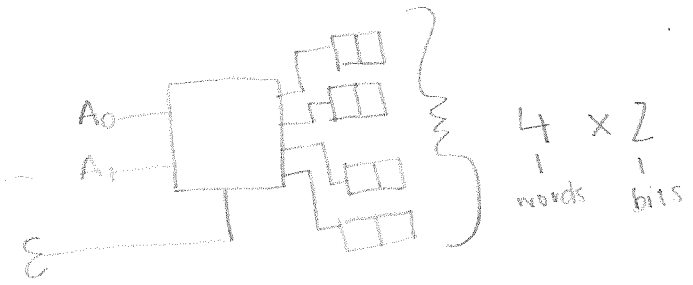
- PCI (Peripheral Component Interface): A standard developed in the early 1990s to have a higher bandwidth than EISA. It is intended for attaching high-speed peripherals to a computer. Later versions of PCI supported up to 133MHz clock speeds and either 32 bit or 64 bit address and data lines (shared). This standard is described in your book in detail starting on page 204.
- USB (Universal Serial Bus): A standard developed in the early 1990s for attaching low-speed IO devices (such as keyboard, mice, scanners, etc.) to a computer. It is intended for serial transmission (bit-at-a-time) and uses only four lines – two for data, one for power, and one for ground. UISB 2.0, adopted in 1998, allows a data transfer speed of up to 480 Mbps.

Mem: 15 nano sec

Clock: 10 nano sec

March 5

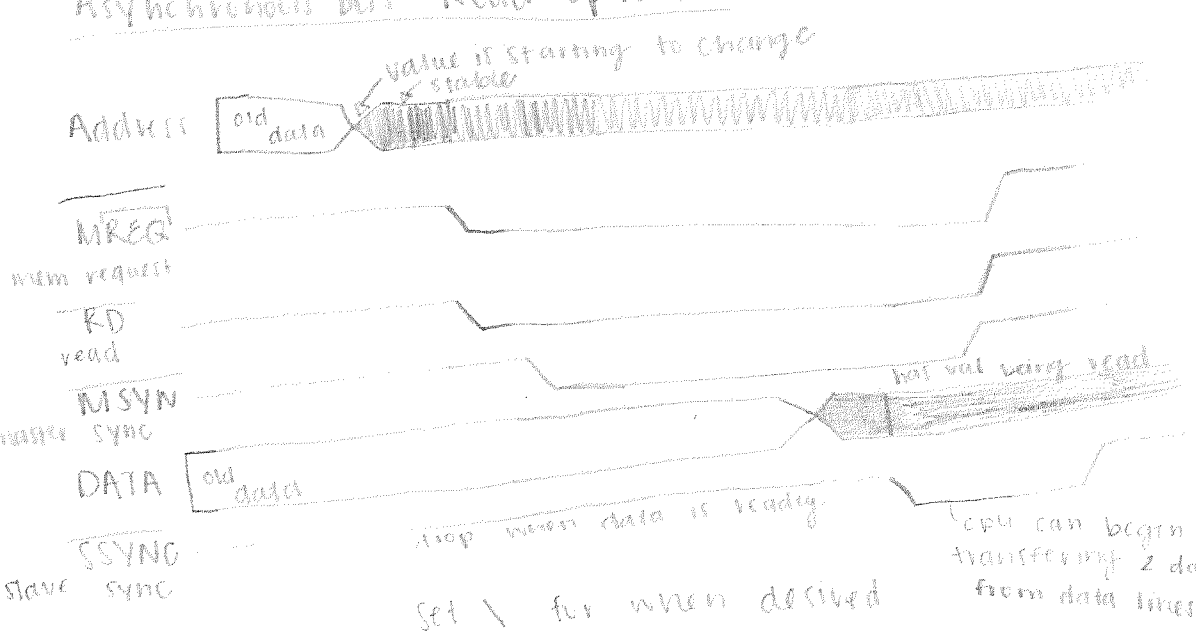
crystal oscillator



4 x 2
1 1
words bits

address?

Asynchronous Bus Read Operation



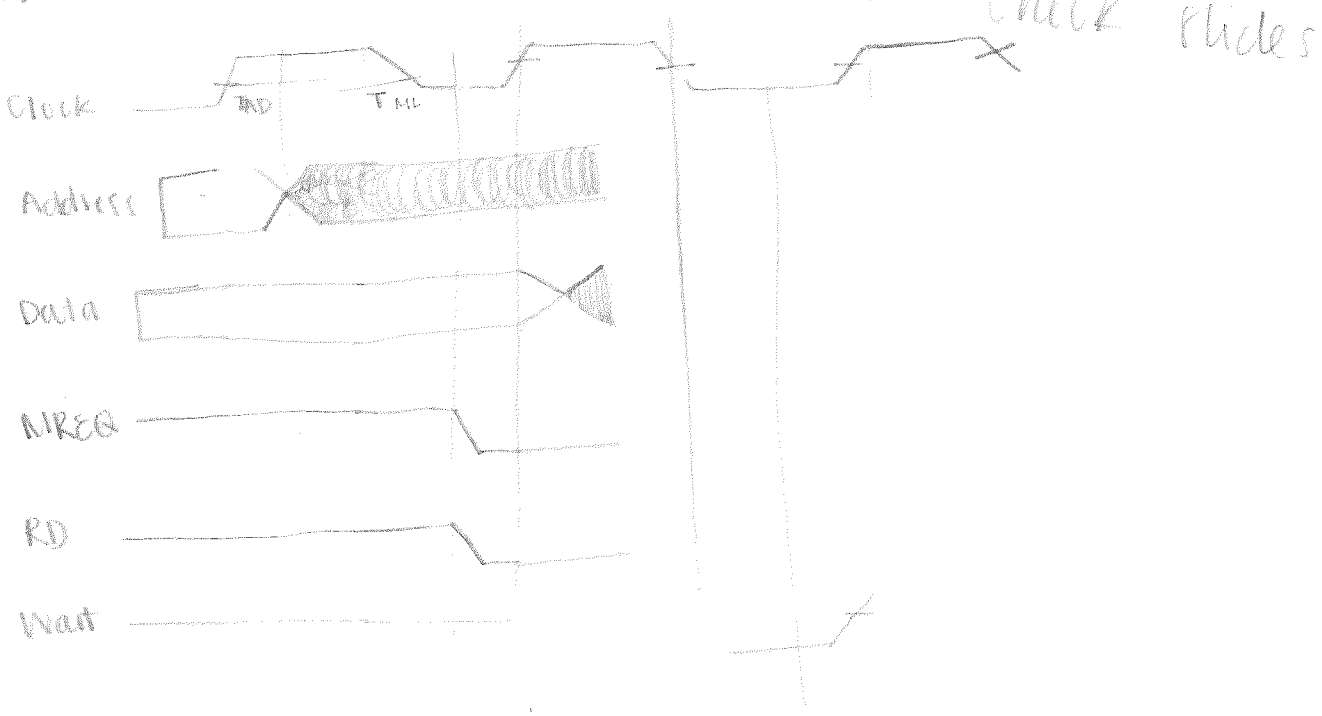
set \ for when derived

\ now stable ? complete

MSYN now start bus transaction

once CPU has data - done so put in orig state

Synchronous Bus Read Operation No Waiting



Example – Synchronous bus transfer from a master device to a slave device -- a read operation from memory to a CPU.

Assume the bus transfer are coordinated by a by a 100MHz clock. This means that the clock goes through 100 million cycles per second (= 10^6 cycles per second), which in turn means each cycle must last $\frac{1}{10^6}$seonds = $\frac{10}{10^7}$seconds, or 10 nanoseconds. Assume further that a memory read operation requires 15 nsecs from the time the read address value is available until one can assume a stable output value is available.

The overall specifications for the synchronous transfer protocol that we will use are as follows:

- The rising edge of the clock pulse that represents the start of cycle T_1 is the "go" signal for establishing the correct values on the address lines. We will assume that the proper values for the address have been reached after a delay of time $\mathbf{T_{AD}}$.

- While the clock pulse in T_1 is still high (or even earlier for that matter) signals indicating the type of data transfer to take place (a read signal, RD, in this case) and the device involved (a memory request signal, MREQ, for this example) can be activated. The actual realization of these signals will not commence however until the falling edge of the clock pulse for T1 arrives, and then we still experience delays T_{RL} (for the RD signal) and T_M (for the MREQ signal) until the signal values stabilize on their respective lines. Note, some memory designs may also require that the address values be established for a certain time T_{ML} before the memory enable signal is activated. If this is the case, this constraint must be accounted for as well. Such a delay may be necessary, for example so that the address decoders inside main memory have a chance to get the proper address before the enable signal for memory, is activated.

- Although the data to be transferred to the master device (in this example a CPU) may be available earlier, the actual transfer of the data will not commence until the falling edge of the clock pulse of the cycle in which the data values became stable on the data lines. Furthermore we will require that this stability of data values signals must have been achieved at least for time T_{DS} prior to the falling edge.

We now become more specific about the sequences of activities involved in performing a memory read to a CPU over a synchronous bus and their timings.

1. At the "beginning" of a clock cycle T_1, the CPU places an address on the bus' address lines (ADDR)
   - Clock pulses do not change immediately so we will take the midpoint of the rising edge of the clock pulse to represent the beginning point of a clock pulse. Likewise the midpoint of the falling edge will be regarded as the end of the clock pulse. We also assume it takes 1 nsec for the clock pulse to change.
   - The address is not immediately stable on the address lines. There is an "address output" delay which represents how long one must wait until the address signals are stable on the address lines. We denoted this delay above as T_{AD}. If we suppose T_{AD} is at most 4 nsecs then after waiting 4 nsecs from the beginning of T_1 we can assume the address lines contain the correct address.
2. The CPU indicates the address on the address lines is a memory address by activating the MREQ line and the RD control lines. The bars over the lines in their specification means activation requires a low (0) signal.
   - Before dropping the signal on the MREQ line from 1 to 0 we shall agree to wait for $\mathbf{T_{ML}}$ nsecs after the address has stabilized.
   - The actual change in signals for MREQ and RD will be triggered by the falling edge of the clock pulse. In order to give the MREQ and RD times to stabilize, however, we must wait for a period of time denoted by T_M and T_{RL}, respectively. For the purposes of this example we shall assume this delay is no more than 3 nsecs for each.
   - In some buses there may be special lines such as MEMR (memory read), MEMW (memory write), IOR (IO device read), IOW (IO device write), etc. for this task that allows one signal line to be used instead of two.
3. Ideally the speed of memory is such that it can have the data available for the CPU and the CPU can start to store it on the falling edge of the next clock cycle T_2 (meaning a new bus operation could commence as early as cycle T_3). This ideal circumstance may not be realizable, however, and the CPU must then be told that the data will not be available during the next cycle and so should not be allowed to store the data on the data lines during that cycle. This signaling is done by activating (with a low value) a $\overline{WAIT}$ signal.
   - In our specific example, by the time all of the setup times and delay times are accommodated, we are close to the rising edge of the next clock pulse T_2. Our assumption is that reading from memory requires 15 nsec from the time

the address values become stable. Since our clock period is 10 nsecs this means the next clock cycle, T_2, will not afford us enough time to complete this memory operation; consequently we need a way to signal the CPU that it will have to wait at least another cycle for the data to be available. This is done by control signal WAIT drop to 0 from its normal value 1 at the start of T_2. Since the additional clock cycle will afford us more than enough time to for memory to place the correct value on the data lines and for the CPU to store it, we can reverse the value of WAIT at the start of T_3 so the CPU knows it store the value on the data lines during cycle T_3.

4. Memory (the slave device) reads data from the appropriate word and places it on the bus' data lines so the CPU can access it and store it. The timing specifications are that the CPU can commence storing the value on the falling edge of the pulse beginning cycle T3.
   - Given what we know about memory's operating times, we also know that the read operation will complete sometime during the first half of T_3.since the read operation commenced prior to the start of T_2. If the timing specification requires that data is to be allowed to enter the CPU on the falling edge of T3 then we have to know the setup time for the data to stabilize on the bus before the falling edge of T3. Let us call this time T_{DS} and assume it must be at least 2 nsec. This means that we have to be assured that the data read must be completed 2 nsec or more before the falling edge of the clock pulse in order for this to happen
5. Once the memory data has been read into a CPU register the CPU (master) can deactivate the read and memory request lines.
   - Usually one requires waiting for times t_{MH} and t_{RH} after the time the data has started loading into the CPU before deactivating the READ and MREQ lines. Likewise there should be a delay of t_{DH} seconds after the RD signal has been deactivated before the values on the data lines can be changed again.

no test 2 - only final exam
questions that would be on test 2 could come
find out tonight if CotC is pass/fail

Pass → all the assignments ≥ 1 > on all

CHAPTER 4
DESIGN Of A SIMPLE CPU

Section 1. Introduction

In Chapter 1 we studied circuit design at what is commonly referred to as the digital logic (or gate) level. Here the fundamental unit for processing is a binary, digital, electronic signal, and the basic components of circuits are gates. Connections among gates made with individual conducting lines.

As part of our study of digital logic circuits, however, we designed circuits capable of accepting multi-bit inputs, producing multi-bit output, and undergoing multi-bit changes of state. Among some of these circuits are multi-bit gates, multiplexers, decoders, bit-sliced alu's, various forms of registers (parallel in/out, counters, shift registers), etc. These circuits are among the fundamental components for a level of circuit design immediately above the digital logic level in a digital system design hierarchy. In recognition of the omnipresence of registers at this level, it is commonly referred to as the register level (or register-transfer level).

In Chapter 2 we introduced one register-level structure when we introduced main memory as an integrated collection of registers (which we call words or bytes when they are used in reference to memory). In this chapter we take on another register-level structure when we introduce important concepts behind the design of central processing units (CPU's), and incorporate these concepts into the design of a simple processor.

Recall that the primary function of a CPU is to execute programs expressed in the processor's own machine language. During their execution programs and their accompanying data are stored wholly or in part in a main memory M which lies outside the CPU. To actually execute the program the CPU must perform the following actions:

1. Fetch I from M by performing one or more memory read operations.
2. Determine the address in M of the next machine language instruction I (sometimes included in step 1).
3. Decode I to determine the operation to be performed.
4. If the instruction uses a word in memory, determine where it is.
5. If necessary from step 4, retrieve the word from memory and copy it into a CPU register. .
6. Perform the operations specified by I. This may involve writing a word to memory.
7. Go back to step 1 to begin executing the next instruction.

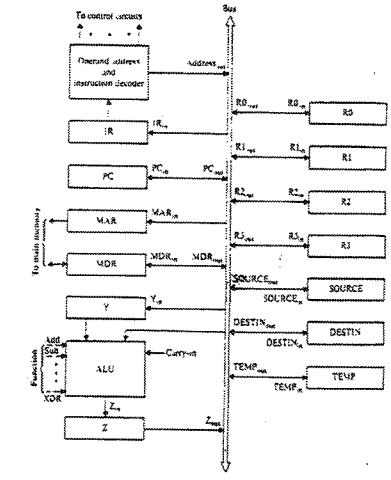
These steps comprise what is known as the *instruction cycle* or *fetch-execute cycle*.

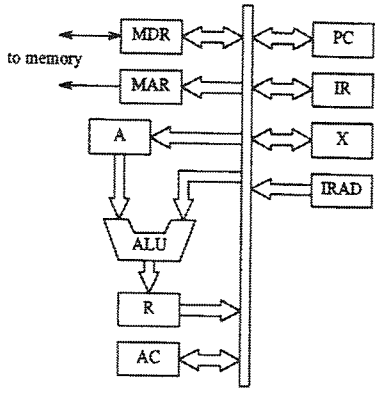
During normal execution of a program the CPU repeatedly goes through the instruction cycle. The circuitry within the CPU to implement this process consists of:

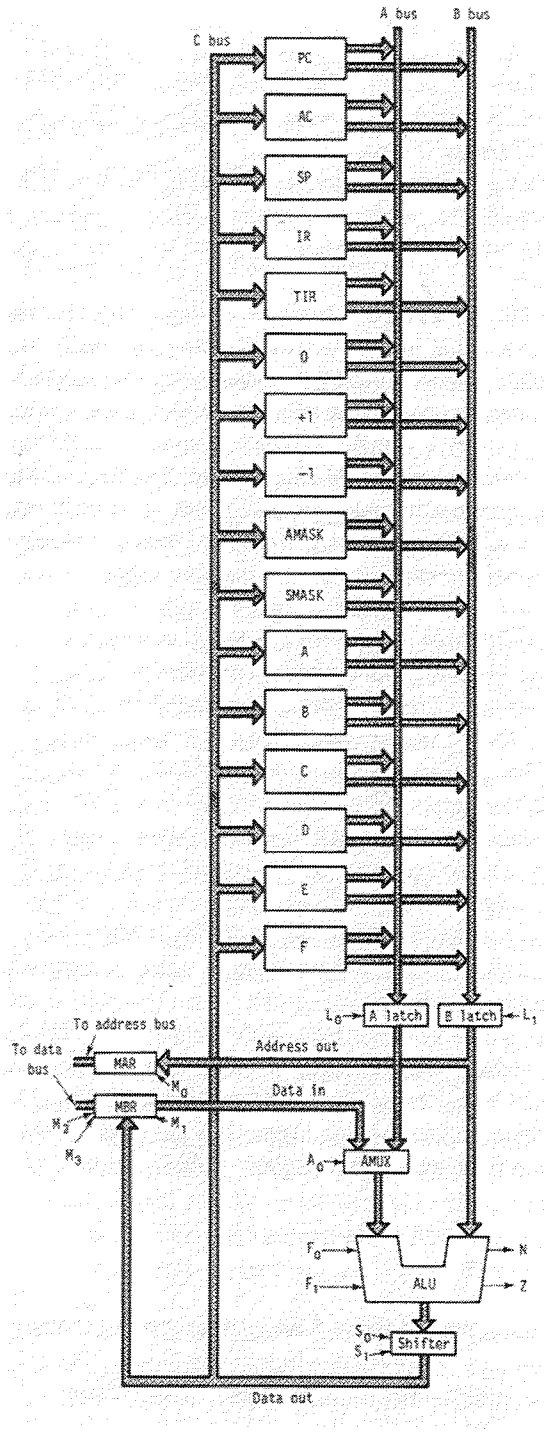
1. An appropriate sized alu.
2. A variety of registers for the temporary storage of addresses, instructions and data.
3. Control circuitry to properly sequence the transfers of data among the CPU's alu and internal registers that are needed to implement the steps of the instruction cycle for each machine language instruction.

The organization used to connect the registers and CPU is often referred to as the *data path* of the CPU. On the next page we show the data paths for some simple, hypothetical, CPU's. In section 2 we shall describe the data path for our own hypothetical CPU and use this CPU as a vehicle for introducing processor organization principles.

Single Bus Data Path







Section 2. Data Path for a Hypothetical CPU

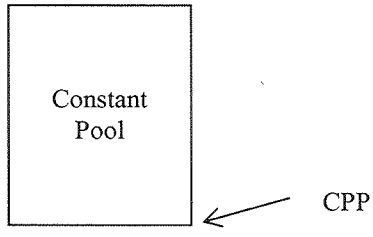
On page 245 of your textbook is the data path for a hypothetical CPU we shall design for this course.

We note that the data path for our CPU uses a triple bus organization with all of the registers allowing only uni-directional transfers either to or from the bus. (Note, one of the buses, bus A, only goes from register H to the alu). The register-level components visible in our data path are:

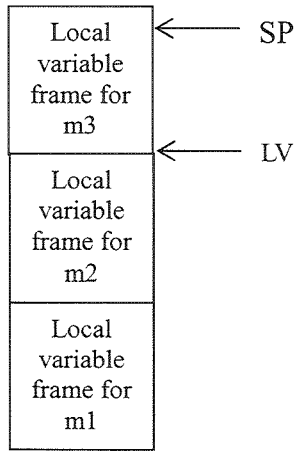
1. *A 32-bit ALU* capable of performing sixteen different functions, as determined by six control lines – F_1, F_0, ENA, ENB, INVA, and INC. A table showing the relationship between control signal values and ALU functions is given on page 246. The ALU also has two output lines N and Z that tell us whether the result of an operation (when regarded as a signed integer) is negative (N = 1) or zero (Z = 1).
2. A holding register (H) that is connected to the left input of the ALU. H is a 32-bit register and is connected to the ALU by a data bus A with 32 data lines.
3. A 32-bit shift register (Shifter) that is connected to the output lines of the ALU. The shift register has two control lines SLL8 and SRA1. When active, the signal SLL8 will cause the shift contents of the register to shift left by one byte and fill the rightmost 8 bits with zeros; When the control signal SRA1 is active it will cause the register to shifts its bits one bit to the right, but leaves the leftmost bit unchanged.
4. A 32-bit memory address register (MAR) and an-bit program counter (PC) to address values in main memory. The MAR addresses 32-bit words in memory while the PC addresses 8-bit bytes.
5. A 32-bit memory data register (MDR) that contains a 32-bit word that has been read from memory, or which will be written to memory.
6. An 8-bit memory buffer register (MBR) which contains a byte of memory.
7. A 32-bit stack pointer (SP). The purpose of this register will become clear later.
8. A 32-bit local variable register (LV) The purpose of this register will become clear later.
9. A 32-bit top of stack register (TOS). The purpose of this register will become clear later.
10. A 32-bit old program counter register (OPC). The purpose of this register will become clear later.
11. A 32-bit constant pool pointer (CPP). The purpose of this register will become clear later.

We are going to assume that our CPU will implement a small subset—IJVM— of the Java virtual machine (JVM). In addition we assume the following memory model for IJVM:

1. ***Constant Pool***: There is an area of main memory known as the "constant pool" that stores all constants, strings, and references to (other) areas of main memory. The register CPP in our data path contains the address of the first *word* of this constant pool.

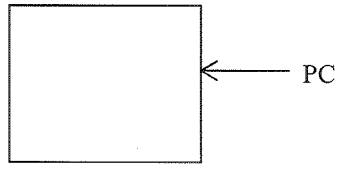


2. ***Local Variable Frame***: Whenever a method of a Java class is invoked, an area known as a *local variable frame* is created for the method's parameters and local variables. The values of any parameters that are passed are stored at the beginning of the local variable frame. The register LV in our data path contains the address of the word that begins the local variable frame of the currently executing method, while the register SP contains the address of the word of the last of the local variables of the currently executing method. Note, as methods invoke other methods the local variable frames of the called methods are stacked atop those of the calling methods, with LV referencing the starting address of the most recently called method. Thus, if method m1 called method m2, which in turn called method m3, then the local variable frames would look like the following:



3. ***Operand Stack*:** The space immediately above that for the local variable frame of the currently executing method is known as the ***operand stack***. The operand stack is used to hold operands during the computation of an arithmetic expression and the value of SP will point at the word immediately before that where the next operand to be placed on the stack will go. We will also assume that at the beginning and end of each machine language instruction that the register TOS will contain a copy of the word being referenced by SP

4. ***Method Area:*** The (byte-addressable) region of memory that contains the code of the program being executed is known as the *method area*. The register PC in our data path stores a byte indicating the address of the program instruction to be addressed next.



We now give the IJVM instruction set (that is, the machine language for the CPU we are designing)

IJVM Instruction Set for Hypothetical CPU			
Op-code (hex)	Mnemonic	Meaning	Notes
10	`BIPUSH` *`byte`*	Push byte onto stack	*`byte`* is 1 byte
59	`DUP`	Copy top word on stack and push onto stack	
A7	`GOTO` *`offset`*	Unconditional branch	*`offset`* is 2 bytes
60	`IADD`	Pop two words from operand stack and push their sum	
7E	`IAND`	Pop two words from operand stack and push their bit-wise AND	
99	`IFEQ` *`offset`*	Pop one word from the operand stack and branch if it is zero	*`offset`* is 2 bytes
9B	`IFLT` *`offset`*	Pop one word from the operand stack and branch if it is less than zero	*`offset`* is 2 bytes
9F	`IF_ICMPEQ` *`offset`*	Pop two words from the operand stack and branch if they are equal	*`offset`* is 2 bytes
84	`IINC` *`varnum const`*	Add a constant to a local variable	*`varnum`* is 1 byte; *`const`* is 1 byte

15	ILOAD *varnum*	Push local variable onto the operand stack	
B6	INVOKEVIRTUAL *disp*	Invoke a method	*disp* is 2 bytes
80	IOR	Pop two words from the operand stack and push their bit-wise OR	
AC	IRETURN	Return from a method with an integer return value	
36	ISTORE *varnum*	Pop a words from the operand stack and store it in a local variable	*varnum* is 1 byte
64	ISUB	Pop two words from the operand stack and push their difference onto the stack	
13	LDC_W *index*	Push a constant from the constant pool onto the operand stack	*index* is 2 bytes
00	NOP	Do nothing	
57	POP	Pop the word on top of the operand stack	
5F	SWAP	Swap the top two words on the operand stack	
C4	WIDE	A prefix instruction; the next instruction has a 16-bit index	

Data Path Timing and Register Transfers for IJVM

In this subsection we develop the data transfers that will be needed to implement each of the IJVM instructions using the given data path. We assume that these data transfers will be coordinated by a clock pulse being transmitted throughout the data path. It is important to know, however, just how much can be done in our data path in a single clock pulse. This is shown in Figure 4-3 on page 248 of your textbook. What is most significant for us at this time about this clock cycle is that it is significantly long to allow us to read from and write to the same register in one clock cycle.

Another point to remember is that it is possible to access memory in two ways from our data path:

- 4-byte words can be read from or written to memory using the MAR and MDR registers. Here MAR contains the address of the word in memory involved in the transaction and MDR either receives the word read or holds the word to be written. The appropriate control signal **read** (or **rd)** or **write** (or **wr)** will indicate the memory operation to perform.

- 1-byte words can be read (only) using the PC and MBR registers in a manner similar to that above. To distinguish a read operation using PC/MBR from one using MAR/MDR, we use the designation **fetch** for a PC/MBR read.

Register Transfer Notation: We complete this section by giving the sequences of registers transfers needed to implement each of the IJVM instructions. In showing these registers transfers we adopt the following notational conventions (this is slightly different from your author's, but I think it makes certain things clearer):

1. D = [S]

   Transfer the content of source register S to destination registers D.

2. D1=D2= …=Dn = [S]

   Transfer the content of source register S to destination registers D1, D2,...,Dn.

3. Op1; Op2

   Perform operations op1 and op2 at the same time.

4. D := f([R1],...,[Rn])

   The function f is performed using the contents of registers R1 ,..., Rn as operands and its value is transferred to register D. If f is a binary operator (i.e. n = 2) we use infix notation rather than prefix notation.

5. a symbol for a data line, e.g. X

   Generate a signal on the indicated line: 1 if the symbol is uncomplemented (e.g., X) and 0 if the symbol is complemented (e.g. $\overline{X}$).

6. if *condition* then *RT-statement*

   Here *condition* has the form data-line or register = signal value(s). The statement indicates that the given register-transfer (RT) statement is to be performed if the indicated data line or register has the specified value.

Examples of registers transfers implementing the instruction cycle for IJVM: We now give examples of the register transfers needed to implement the instruction cycle for our CPU. In doing so we note the following:

- At the beginning and end of each IJVM instruction the register TOS will contain a copy of the word being referenced by SP; otherwise it can be used as a temporary storage register.
- The register OPC also can be used as a temporary (or scratch) register. It gets its name from its use in storing previous values ("old") values of the PC register.
- If a memory read is indicated (by either the **read** or **fetch**) the read operation starts at the end of the data path cycle using the value in the MAR or PC as is appropriate. Our discussion of the data path cycle showed that it is possible for PC or MAR to attain new values before the end of the cycle, however, meaning it is possible to change the value of PC or MAR and initiate a read operation with this new value.
- Continuing our discussion of memory reads, we shall assume that memory completes its operation within one cycle. This means that following the initiation of a memory read, we can assume the data being read is available in the MDR or MDR at the *end* of the next data path cycle, but not at the beginning of this cycle. Consequently we must wait until the cycle after that before it can be used.
- Whenever the value of the MBR (which is only 8 bits) is placed on the B bus two options will be available:
  a. Put the value in MBR in the low order bits and append twenty-four 0s for the high order bits.
  b. Treat the value of the MBR as an 8-bit signed integer and generate a 32-bit word with the same value by extending the sign bit of the 8-bit value into the twenty-four high order bits (meaning either twenty four 0s or twenty four 1s are copied).

Examples of Register Transfers Needed for various IJVM Instructions:

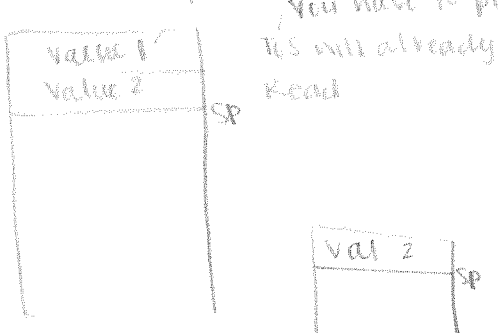
① Fetch Step of Instruction Cycle

Fetch; PC = [PC] + 1

└ PC has the addy 4 the instruction to be fetched

FETCH
POP
IADD
DUP
ILOAD
GOTO
IFLT
INVOKEVIRTUAL

② IADD



(A) SP = [SP] - 1 ; MAR = [SP] - 1 ; Read

SP ; MAR = [SP] - 1 ; Read

(B) H = [TOS] [not MDR = M[MAR] at the end of the clock cycle]

(C) MDR = [H] + [MDR] ; [TOS] = [H] + [MDR] ; write

Let us now look at some of the full microinstruction sequences for these instructions:

Control store address	Next Address	JAM	ALU	C bus	Memory	B bus	hex
Main Loop (fetch)							
100	000000000	100	00110101	000000100	001	0001	004350211
NOP							
000	100000000	000	00000000	000000000	000	0000	800000000
ILOAD							
015	000010110	000					
016	000010111	000	00111100	000000001	010	0011	0B83C0063
017	000011000	000					
018	000011001	000					
019	100000000	000	00010100	001000000	000	0000	800142000
IAND							
07E	001111111	000	00010101	000001001	010	0100	3F8150464
07F	011100000	000					
0E0	100000000	000					
GOTO							
0A7	010101000	000					
0A8	010101001	000					
0A9	010101010	000	10010100	100000000	000	0010	550948002
0AA	010101011	000					
0AB	011010000	000					
0D0	100000000	000	00000000	000000000	000	0000	800000000
IFLT							
09B	010011100	000					
09C	010011101	000					
09D	010011110	000					
09E	011110000	010	00010100	000000000	000	1000	782140008
1F0	010101000	000					
F0	011110001	000					
F1	011110010	000					
F2	100000000	000	00000000	000000000	000	0000	800000000
WIDE							
C4	100000000	100	00110101	000000100	001	0001	804350211
WIDE_ILOAD							
115	100010110	000					
116	100010111	000					
117	100011000	000					
118	000010111	000					

The final schematic for our cpu is given on page 254 of your textbook. There are two features of this code we need to consider:

- The box labeled "high bit" contains combinational circuitry that takes as input the values of the JAMN and JAMZ bits of the microinstruction, the value of the μ_8 bit of the microinstruction (bit 35), and the values of the N and Z flip-flops. It produces the following output value F. which will become the value of the leftmost bit of the MPC register:

$$F = (JAMZ * Z) + (JAMN * N) + \mu_8$$

- The box labeled OR contains combinational circuitry that takes as input the 8 bits of the MBR, the rightmost 8 bits of the next address field (that is bits $\mu_7\mu_6\mu_5\mu_4\mu_3\mu_2\mu_1\mu_0$), and the value of the JMPC bit. If JMPC = 1 then the box produces as output the (8-bit) bitwise-or of the MBR bits with$_7\mu_6\mu_5\mu_4\mu_3\mu_2\mu_1\mu_0$; if JMPC = 0 then the box merely passes the value of $\mu_7\mu_6\mu_5\mu_4\mu_3\mu_2\mu_1\mu_0$ through to the rightmost 8 bits of MPC.

Section 4. A Hardwired Control Unit

The cpu that we designed in the previous section is an example of what is known as a ***microprogrammed control unit*** because of the use of microinstructions and the microprogram storage unit. There is, however and alternative approach to implementing the control unit known as a ***hardwired control unit***. We indicate the overall approach for such a control unit here. This approach is especially useful for machine architectures that do not have a large number of machine instructions, or instructions as complex as INVOKEVIRTUAL.

With a hardwired control unit the correct control signals for each register transfer step of each phase of the instruction cycle are generated at the proper time by means of a clock-driven counter interfaced with a decoder as shown below.

Each output line of this sequence corresponds to a step number in a register-transfer sequence, where 2^n is equal to, or greater than, the maximum number of steps that will ever be required.

The required control signals to be activated for each register transfer will then be generated by additional ***combinational*** circuitry that uses the following input signals:

a. The output of the step-sequences, so we know what step we are on.
b. The content of the MBR at the end of fetch, so we know which instruction we are executing
c. The values of any condition codes (such as the N and Z signals) that tell us whether the result of an alu operation was negative, or zero, for example.

In our examples at the end of Section 2. of this chapter we derived some of the sequences of control signals needed to implement the fetch phase of the instruction cycle as well as each machine language instruction. We note that at most 23 steps are needed to carry out any instruction (1 for fetching it, and at most 22 to implement it). If we let T0,...,T22 denote the first 23 output lines of the counter-decoder circuit given above, representing lines that are active (i.e. = 1) on the successive clock ticks, then we can specify the values that the control signals must have to correctly implement the instruction cycle for our machine language instruction set by control equations such as the following

```
fetch =


 mbru =


  mar =
```

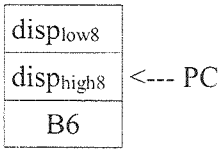
Sequence of steps for INVOKEVIRTUAL

The instruction format and what it means

INVOKEVIRTUAL disp (16 bits)

here **disp** is a 16-bit unsigned integer that indicates the position in the constant pool (accessible by CPP) that contains the starting address within the method area of the method being invoked. When stored as a machine instruction in the method area of memory, disp will be split into two 8-bit bytes and stored as a big-endian value.

Method area when invoked



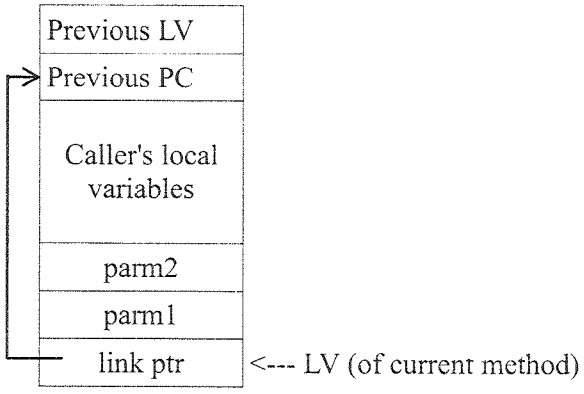
Prior to invoking a method:

1. Caller pushes onto the stack a reference OBJREF to the object being called. This also serves as parameter 0 (or $parm_0$)

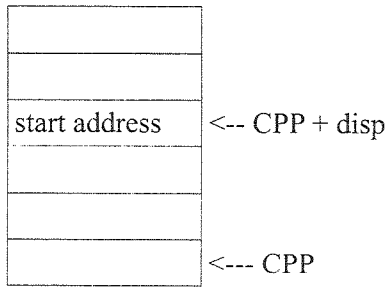
   - See the discussion in your book on page 263 for why this is not really needed for IJVM but is retained for consistency with the Java Virtual Machine (JVM).

2. The caller pushes the remaining parameters onto the stack – say $parm_1$, $parm_2$..$parm_n$.

The local variable area (referenced by LV) before INVOKEVIRTUAL is called. The current method is presumed to have had 2 parameters. INVOKEVIRTUAL will have to construct something similar for the called method.



Constant pool area when INVOKEVIRTUAL is called,



Here: #parms is a 16 bit number giving the number of parameters; the parameters are presumed to have been previously pushed onto the stack. Recall that $parm_0$ represents OBJREF.

Sequence that occurs for INVOKEVIRTUAL:

1. The two ***unsigned*** index bytes that follow the op-code are used to construct an index into the constant pool (with the first value representing the high-order byte of this index).

2. The instruction computes the base address of the new local variable frame by subtracting off the number of parameters from the stack pointer and setting LV to point to OBJREF.

3. At this location the implementation stores the address of the location where the old PC is to be stored, overwriting OBJREF.

    - This address is computed by adding the size of the local variable frame to the address contained in LV

4. Immediately above the address where the old PC is to be stored is the address where the old LV is to be stored.

5. Immediately above that address is the beginning of the stack for the newly called procedure.
    - SP is set to point to the old LV, which is the address immediately below the first empty location on the stack.

    - Our stacks always grow upward, toward higher addresses.

6. The PC is set to point to the 5^{th} byte in the method code space.

See page 264 for a graphic of the changes that occurred.

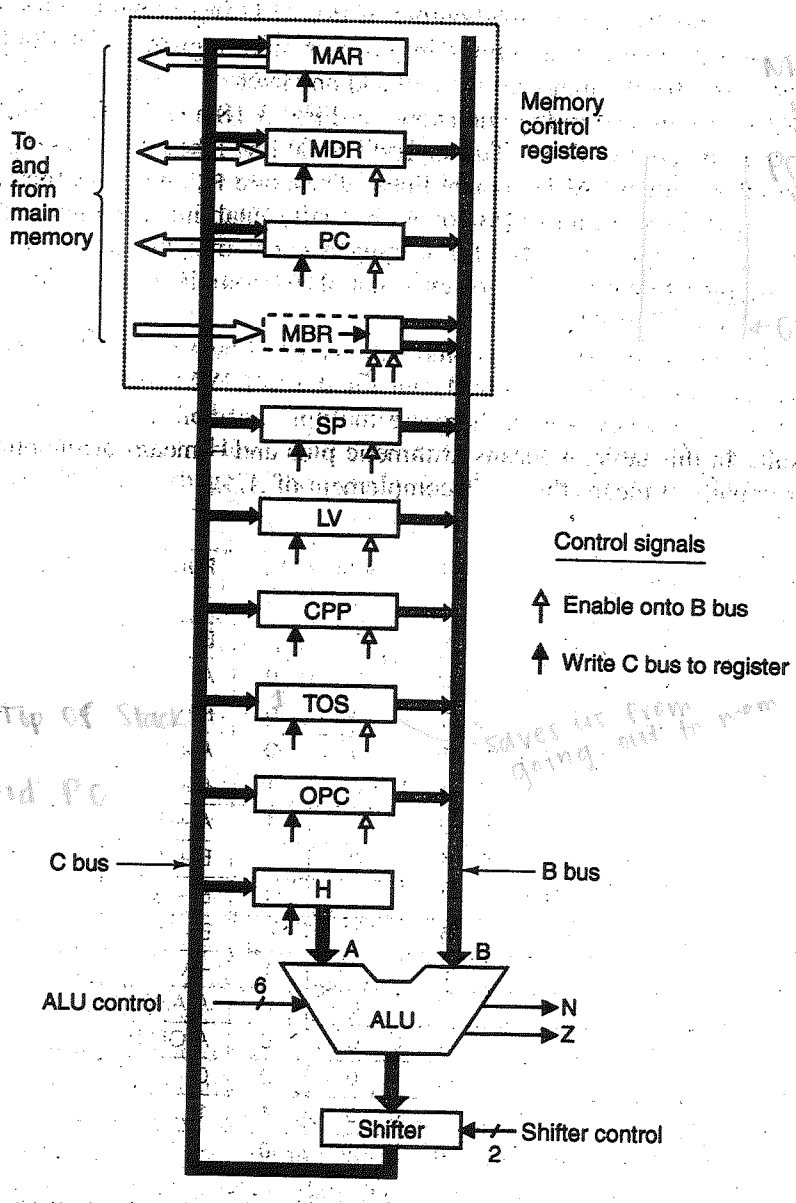
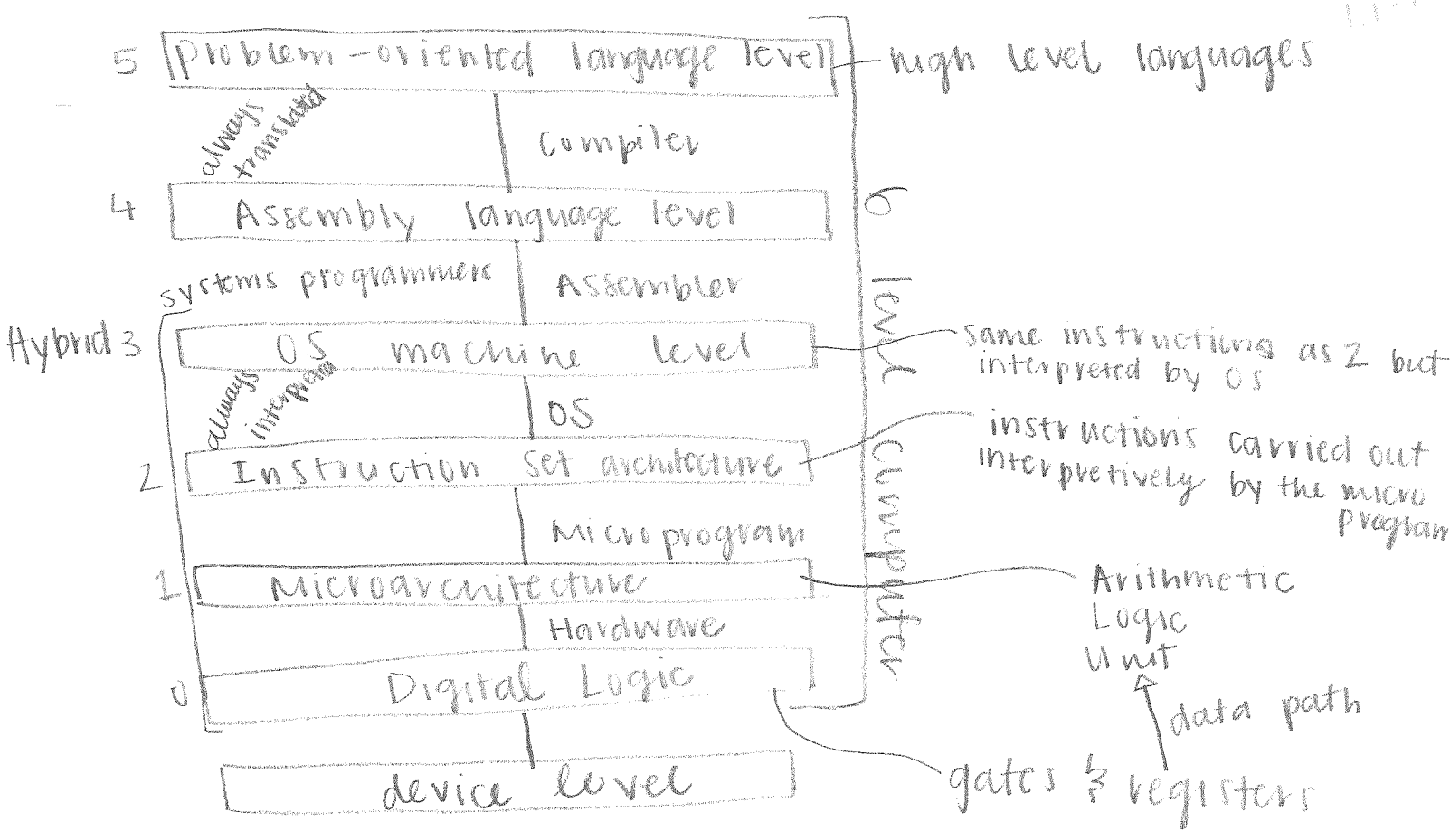


Figure 4-1. The data path of the example microarchitecture used in this chapter.

registers, to which we have assigned symbolic names such as PC, SP, and MDR. Though some of these names are familiar, it is important to understand that these registers are accessible only at the microarchitecture level (by the microprogram). They are given these names because they usually hold a value corresponding to the variable of the same name in the ISA level architecture. Most registers can drive

program - sequence of instructions

Book Reading 1.1-1.3



architecture - set of data types, operations, features the programmer sees

* Hardware & software are logically equivalent *

Microprogramming

1st digital computers 1940s → ISA, digital Logic

Maurice Wilkes 1951 → ISA, microprogram, digital logic
enhance reliability w/ less tubes needed

1970's ISA | microprogram instead of directly by electronics

OS

FORTRAN MONITOR SYSTEM on IBM 709

↓

System call (OS Macros / Supervisor calls)

↓

Batch Processing

↓

Timesharing systems

Microprogramming

features added to increase OS productivity & efficiency

Gen 0 : Mechanical Comp 1642-1945

Blasie Pascal - working calculator (+ −)

Baron Gttfried Wilhelm von Leibniz - new cal (+ − • %)

~ 150 yrs later

Charles Babbage - difference engine (naval navigation)

- analytical engine = general purpose
  1. store : (memory)
  2. mill (computation unit)
  3. input (punch card reader)
  4. output (punched & printed)

programmed in assembly by Ada

Gen1 : Vacuum Tubes

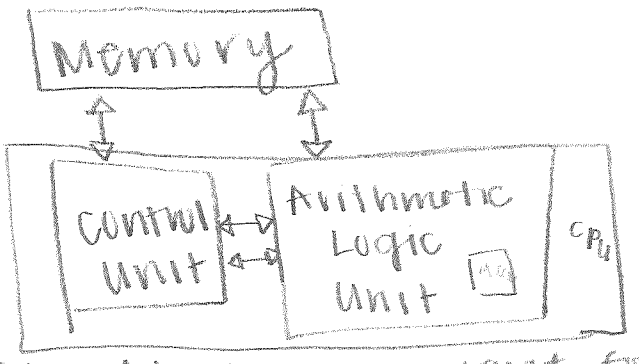
messages were encoded using ENIGMA

Colossus - 1st electronic digital computer by Alan Turring

Electronic Numerical Integrator & Computer ENIAC war was over

Electronic Discrete Variable Automatic Computer EDVAC

Von Neuman



CPU

Accumulator

IAS Machine - meant for heavy # crunching

Gen 2: Transistors

TX-0 1st transistored computer

bus - collection of parallel wires
PDP1 fastest computer in the world @ the time
Supercomputers = 6600 ⅁ 7600 ⅁ Cray 1

Gen 3: Integrated Circuits

multiprogramming] several programs in memory at once

Gen 4: Very Large Scale Integration
Computer Centers
~~GUI~~
1980's personal computers

Gen 5: Low Power ⅁ Invisible Computers

Personal Digital Assistants

1st Smart Phone - Simon

Codesigning hardware ⅁ Software

Ubiquitous / pervasive computing