Enclosed you will find a first version of the lab manual that goes with the LD12 Computer Kit. It is intended as a supplement to a standard text on digital logic such as:

Dietmeyer -- Logic Design of Digital Systems -- Allyn & Bacon

Clare -- Designing Logic Systems Using State Machines -- McGraw-Hill

It can also be profitably used in a course in computer architecture. Texts useful here are:

Hill & Peterson -- Digital Systems: Hardware Organization and Design -- Wiley

Peatman -- The Design of Digital Systems -- McGraw-Hill

Foster -- Computer Architecture -- Van Nostrand

 I am actively engaged in extending the book to cover more advanced design techniques at which time I hope to publish it.

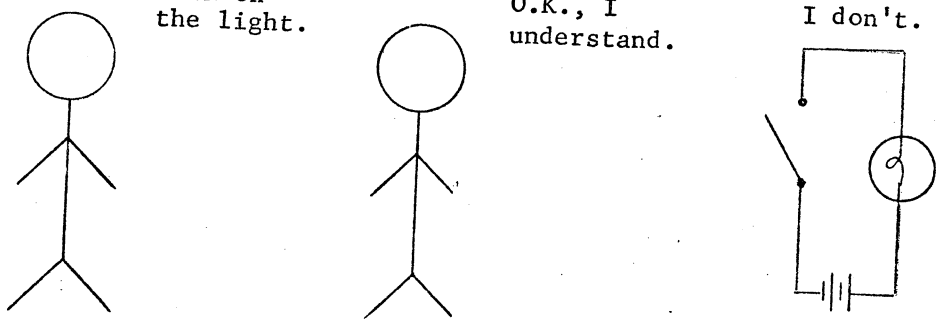
David E. Winkel

(1) INFORMATION REPRESENTATION

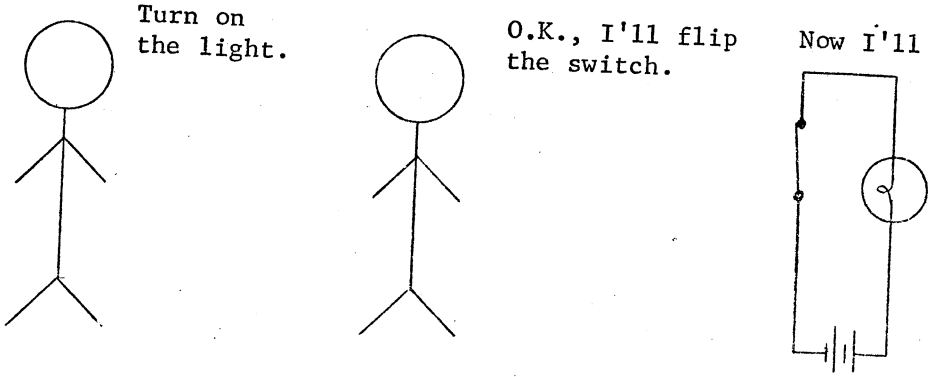
How do we talk to computers? Unfortunately not as easily as we talk to other people. Think of the ways humans can communicate--writing, speaking, hand signs, facial expressions, etc. One of the rude awakenings that new students get when introduced to the subject is how stupid computers are! Human thought must be transcribed onto punched cards or some other machine readable media, and even the most trivial errors (misplaced commas for example) will completely confuse the computer.

But even punched cards hide the inner level of communication within a computer. Internally a computer understands voltages only. That in fact is why we input information on cards. Once a card is punched it is relatively easy to send it through an electromechanical device which will convert the information punched on the card into voltages which can be interpreted by the computer.

Thus, information must be transformed into forms that electronic circuits can understand. To make this perfectly clear let us consider a light switch which in many ways is similar to some of the elementary building blocks used to make a computer. Of course the switch understands only one thing--physical movement--is it on or off? Imagine two people, a room, and a light switch.



Do you see the problem? The verbal command has not been transformed into a form the switch understands.



Now something happens! This may seem trivial but it brings the problem into the open.

What sorts of things do computers (and logic circuits) understand? We repeat--VOLTAGES!

How many different voltages? TWO!

Why only two? Because engineers have been unable to build reliable circuits for anything but two.

What are these voltages?

H = high = approximately 3.0 volts
L = low = approximately 0.2 volts

That only two voltages are recognized by digital logic is fundamental to the whole field. It is so important that a name has been given to it--BINARY--which means that only two stable states exist. Examples of binary devices are:

Switches - either ON or OFF
Punched Cards - any given position either has a hole punched out or it doesn't
Logic Circuits - put out either a high or a low voltage
Magnetic Tape - a given spot has either a north pole or south pole pointing up

These two different states are commonly given names of 1 and 0.* For example if the choices in the left-hand column are arbitrarily made, the meanings of a 1 and 0 are shown in the right two columns.

If a 1 is chosen to mean	then a 1 means	and a 0 means
switch is open	switch open	switch closed
a given spot on a card is punched out	hole at that spot	no hole at that spot
High	High	Low
a given spot is a north pole	north pole	south pole
L	L	H

Again let us emphasize. You, the designer, can choose the meaning of a 1. Of course, you must have some way of telling the world what your

*As the designer you can choose what a 1 means.

choice was so your design can be understood. As we will see a standard notation has been devised to make your choice evident.

Also, understand that 1 and 0 are simply names for the two states of a binary device. 1 and 0 do not have a numerical meaning. You could use X and Y for the two names and everything would work as well. Of course it would be unconventional.

Now we come to an important topic. Can ordinary human (decimal) numbers be represented by only H and L voltages? If not then it will not be possible to build a computer. The answer must be yes.

Let us consider the process of counting, using ordinary decimal digits. Of course there are 10 different digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Suppose we are counting rocks falling out of a chute

* = ROCKS

	DECIMAL NUMBER REPRESENTING THE ROCKS
*	0
**	1
***	2
****	3
*****	4
******	5
*******	6
********	7
*********	8
	9

What do I do now? I have run out of digits. We solve this by a very clever trick called the carry.

We get a carry of 1 out of the units column into the tens column and change the 9 to a 0.

****

10

This process always works. Suppose we have 99 rocks and 1 more falls out of the chute.

```
  9  9
 ____      change 9 to a 0
  1        carry 1 into the next column
+ 9
 ____
           change the 9 to a 0
           carry 1 into the next column
 ____
 1 0 0
```

We have defined a general process for counting using the digits 0 - 9.

The same process will work if we have only two digits 0, 1. Now a 1 will be the largest digit you can have in any column just as a 9 was the

largest digit you could have in a given column in an ordinary number. We call numbers represented in such a manner BINARY numbers. Perhaps a better phrase would be BINARY representation since what we will be doing is representing the number of rocks by 1's and 0's. The same could be said about the decimal representation for the same pile of rocks which uses 0's through 9's.

* = ROCKS	BINARY REPRESENTATION
*	0
	1

What do we do now? We have run out of binary digits!

a) Change the 1 to a 0
b) Put a carry in the next column.

**	
***	10
	11

What do we do now? Well, this is exactly like the case of 99 rocks in decimal. Both 9's were changed to 0's and a 1 carry was put into the next column to make 100. In binary, 11 would change to 100 since in binary 11 is the largest two-digit binary number you can have just as 99 is the largest two-digit decimal number you can have.

****	
*****	100
******	101
*******	110
********	111
*********	1000
**********	1001
etc.	1010

We see that it really is possible to represent numbers using just two symbols, 1 and 0. There is a problem in that a binary 11 is indistinguishable from a decimal 11. But a binary 11 really equals a decimal 3. We need some way to tell what kind of representation we are talking about. For this purpose we use a small subscript:

binary $11 = 11_2$ decimal $11 = 11_{10}$

A more mathematical way of looking at either binary or decimal numbers is the place value system. In this system a digits value depends on the column it is in. For example, in decimal we have a 1's column, 10's column, 100's column, etc. Take the decimal number 432 and label the columns starting from the right with column 0.

column number	2	1	0
	4	3	2

column number	description
0	1's column
1	10's column
2	100's column
etc.	

In fact 432 really means (4 X 100) + (3 X 10) + (2 X 1).

A similar thing can be done in binary except now

column number	description
0	1's column
1	2's column
2	4's column
3	8's column
etc.	

Now 101_2 really means (1 X 4) + (0 X 2) + (1 X 1) = 5_{10}.

Thus we see that a 1 represents the presence of a power of 2 and a 0 its absence.

How the 1's and 0's in a binary number are in turn represented inside the computer by voltages is again the designers choice. Some designers represent a 1 by H, others do the opposite. The symbols used by the designer will make his choice clear.

Now for another piece of jargon. The individual 1's and 0's in a binary number are called BITS (B I nary digiTS).

The only way to feel comfortable with binary numbers is to count to 64_{10} in binary. The student is urged to do this now. So you can check yourself along the way.

$13_{10} = 1101_2$

$19_{10} = 10011_2$

$33_{10} = 100001_2$

$63_{10} = 111111_2$

Let's take a break and discuss a less weighty but related subject--numerical accuracy. As you know from hand held electronic calculators, the more digits the more accuracy you can get. The same is true for binary computers. The smallest have eight bits of accuracy and some of the very large, fast scientific machines have 60 bits. The design in this book is a 12 bit machine.

Another bit of jargon--for our machine 12 bits of data is called a WORD. Thus this machine has 12 bit words whereas the Control Data 6000 series of computers has 60 bit words. The IBM 360-370 series has 32 bit

words. There are more PDP-8 minicomputers in the field than any other single type of computer (>25,000 installed), the PDP-8 has 12 bit words. Most new minicomputers being produced have 16 bit words since it is a nice compromise between cost and accuracy.

There is another parameter besides bits/words that is important in describing memories. That is the number of distinct words of data that can be stored. A fairly typical minicomputer would have four thousand (4k) words of storage. Large scientific machines may have millions of words of storage. Fortunately from a learning standpoint a small memory is as good as a large one and far cheaper.

Now let's get back to binary numbers. These numbers are elegant since they can represent any decimal number with only two symbols, 0 and 1. Remember this is necessary for internal use in a computer. They also have several disadvantages. One of them is it takes more binary digits than decimal digits to represent a given number. For example take 110010_2. That is equal to 50_{10}. It is far easier for a human to keep track of the two digits in 50_{10} than the six in 110010_2. A simple shorthand has been developed for binary numbers to collapse the number of digits a human has to work with. The trick is to start at the right side of the word and group the bits three at a time.

```
110     010
 6       2
```

Now take each 3 bit group and convert it to a single number between 0 and 7. Now you have to remember only the binary numbers between 0 and 7. The resulting number is called an octal number. Why? (Hint: Decimal numbers have 10 distinct digits, 0-9; binary numbers have 2 distinct digits, 0, 1). To show a number is octal we write a small subscript 8 to the right of it. Converting between octal and binary is now simple. The process is best shown by examples:

binary ----> octal		octal ----> binary	
1011	13	6410	110 100 001 000
11001	31	37	011 111
111010	72	14	001 100

All of the commands for the computer will be given in octal because it is such a convenient shorthand.

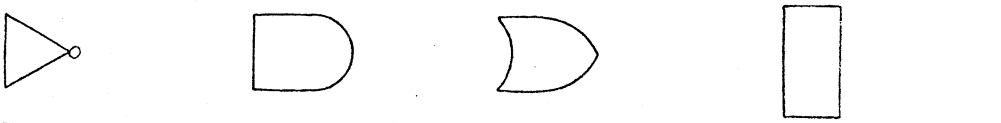
(2) COMBINATIONAL LOGIC (The Logic of Here and Now)

There are two main classes of digital logic circuits, combinational (sometimes called combinatorial) and sequential. We will need both types to accomplish our goal of building hardware that will execute a flow chart.

Combinational circuits act on information represented by H and L voltages and immediately produce an output. Actual circuits take a few nano seconds, which is close enough to immediate (light will travel 1 foot in one nanosecond). This class of circuits is the simplest to understand so we will treat it first.

In general we will not refer to 0's and 1's to represent our data since we are more interested in the voltage which represents it. The reason for this is simple. When you check a circuit the only thing you will be able to test are voltages. As we will see a set of symbols has been devised that gives you these voltages directly at a given point in a digital circuit. It is then a simple matter to place a logic probe at that point and get a visual display of the information. The logic probe used in the lab has small green and red lamps for display. When a given point is L the green lamp will turn on. The red lamp will light for a H input.

Let us now go over the symbols which describe digital logic.



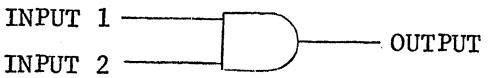
INVERTER AND OR special purpose element

Do not worry at the moment about the function of each of these symbols. Each will be introduced in due course. Let us discuss some of the general properties of the symbols.

1) The shape uniquely identifies its logic function except for the rectangle where this information is supplied by the device name.

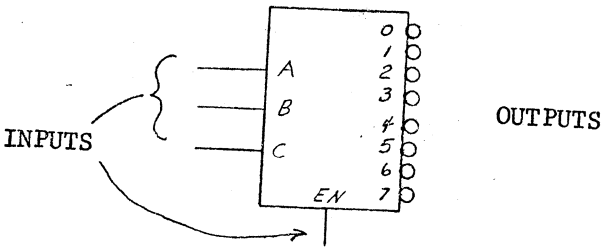
2) Inputs and outputs (copper wires) are represented by lines going into and out of the symbol. For example:

3) Outputs are always on the right; inputs are always on the left, top, or bottom. Thus, we know that in the above example the wires on the left are inputs and the single wire on the right is an output and we can now label it as follows:



Another example would be a decoder. At this point of course you you don't know what a decoder is. Nonetheless if I draw one

for you you should be able to tell which lines are inputs and which outputs.



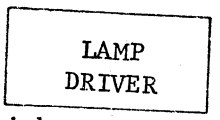
4) The symbols are important since they quickly convey a large amount of information pictorially. The pictoral aspect should be emphasized. The eye can quickly encompass a large drawing with many symbols on it and get the overall picture. Just as easily the eye can zero in on a small portion of a large drawing and extract a great amount of detail. This can be difficult using words or even equations. Very soon you will be thinking of digital logic in pictoral terms since it is so easy and convenient.

5) Voltage polarities are easily shown. The convention is that a small circle represents a low voltage and its absence a high voltage. For example suppose I have a special purpose circuit called a lamp driver. It's function will be to accept an input voltage and turn a lamp on or off on the basis of this input.

Such circuits are very useful in a computer since they can be used to visually monitor a signal.

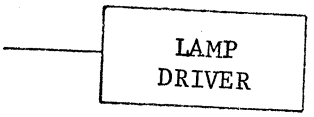
Let us now try to represent a lamp driver graphically:

a) It is a non-standard circuit. Therefore, its symbol will be a rectangle. Of course, a bare rectangle doesn't tell you much so we put a name inside it which describes its function.



b) Its output is on the right side of the box. Since its output is light and not a copper wire no output <u>line</u> has been drawn. An <u>implied</u> output (light) nonetheless exists.

c) One vital piece of information is still missing. What kind of an input voltage turns on the light? In all of the lamp drivers used on your lab kit a <u>high</u> voltage turns on the lamp. Therefore, it would be drawn as below:

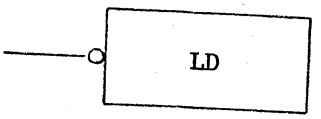


d) At this point review a) - c). You should be able to look at the above symbol and extract the following pieces of information just from the picture:

1.) it is a special purpose circuit (shape)
2.) presumably it turns a light ON or OFF (from its name)
3.) a high voltage will turn the light ON (its input line has no small circle on it)

It is perfectly possible to design a lamp driver that will turn the lamp ON when the input voltage is low. In fact in your lab kit this could be done simply by unplugging the 7406 lamp driver IC (integrated circuit) and substituting a 7407 IC. How would this circuit be represented?

Small circle tells you that a low voltage is required to turn on the lamp

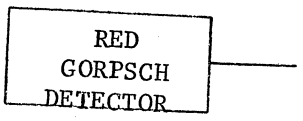


6) Note that the graphical symbol also suppresses much irrelevant information. It does not tell you what's <u>inside</u> the lamp driver. In fact we do not care. It could be transistors or perhaps a gremlin as long as the lamp was reliably turned on or off. The student should get used to looking at symbols as <u>logical</u> building blocks and forget about their internal workings. We are concerned only with the logical process of building a computer by interconnecting such building blocks.

One of the marvels of modern technology is the IC (integrated circuit). They are cheap, compact logical building blocks which have been carefully designed so that very diverse IC's can be interconnected without worrying about impedance matching, and all the other black arts of electrical engineering.

One last word of reassurance. Even professional computer designers are not concerned with the internal construction of an IC. If you are still not convinced take several years of high level electronics courses. But be warned--you won't be a better computer designer by virtue of your knowledge of the inner workings of IC's.

Let's define one more special purpose device and then experiment by connecting them together. The lamp driver required an input to activate it. We will design a special purpose circuit which produces an output which in turn can be input to the lamp driver.



Absence of a small circle indicates the output will be high when a red gorpsch is detected

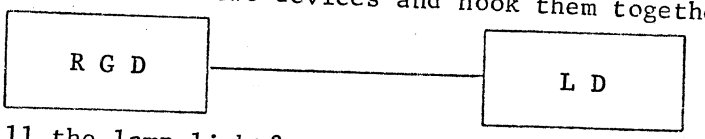
From the above picture we can determine:

1) It is a special purpose circuit.

2) Its input is implied on the left. Since there is no input wire there must be some other means of inputting information to the device. (Perhaps there really is a gremlin sitting inside looking out a window on the left side. When he sees a red gorpsch he throws a switch which drives the output line high.)

3) The output will be H if a Red Gorpsch passes near the detector.

Let's take the two devices and hook them together.

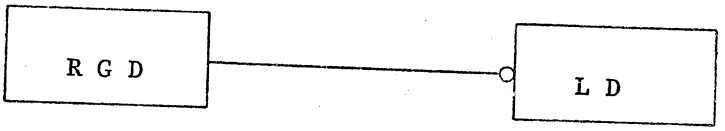


When will the lamp light?

a) Only when a red gorpsch passes near the detector. In that case the RGD output will go H and that is the polarity needed by the lamp driver to turn on the light.

b) Note that only a red gorpsch will produce a H output--a green gorpsch would not, nor would a red elephant. Suppose a green gorpsch passed near the RGD. Its output would be L and the lamp driver would turn the lamp off.

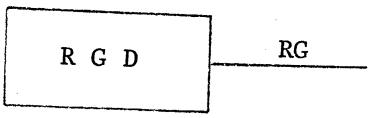
Now suppose we redesign the lamp driver so it will turn on the light when its input goes L. Again we hook the two devices together.



When will the lamp light? The light will be on only when the lamp drivers input is L. Since the RGD is a binary device it always outputs either a H or L. Its output will be L if there is NOT a red gorpsch.

We have just been introduced to the NOT operation of logic. This is a fundamental concept and we must explore it thoroughly.

First let us describe how we talk about the NOT operation. Suppose we have the red gorpsch detector by itself.



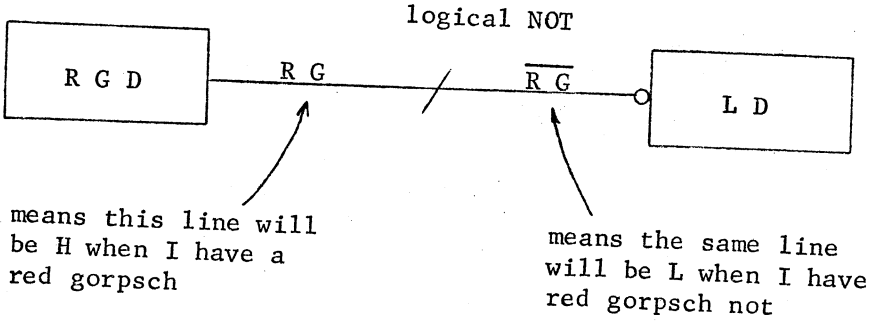
It would be logical to label its output line RG (standing for red gorpsch). A line labeled RG has no meaning by itself since we do not know whether an H or L represents a red gorpsch. To decide that we must trace it back to its source which will tell us its polarity. In this case RG = H.

There are only two voltages this line can have. The other one has to be L which corresponds to red gorpsch not. We write red gorpsch not as $\overline{RG}$.

The crux of the matter is:

A LOGICAL NOT EXISTS WHENEVER A SIGNAL IS USED IN THE OPPOSITE POLARITY FROM ITS GENERATED POLARITY.

To remind ourselves that a logical not has happened we put a slash across the signal line.



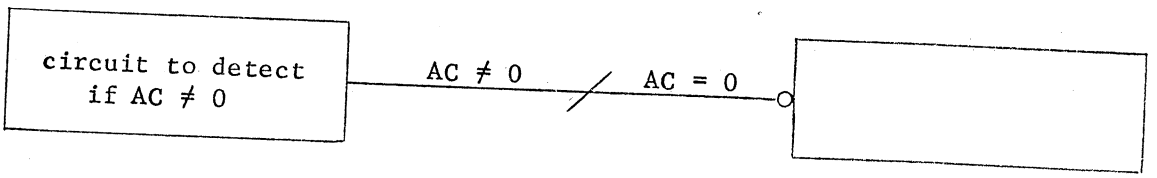
Note: There is no special integrated circuit required to perform the logical not operation. It happens ANY time a signal is used in a polarity opposite from its generated polarity.

The logical not operation is important in digital logic for the same reason it is used in human thought. Many times it is easier to organize your thoughts if a certain condition is not true. An example would be: I will take a trip next week if my car does not break down. Most of us prefer to think that way rather than thinking of all the complications that could arise if it did break down. The concept is sharpened considerably in mathematics or programming where we may wish to alter our course of action on the basis of something being not true.

The same thought processes are used in digital logic. In the computer you will build, it is necessary to determine if the accumulator is equal to 0. (All 12 bits equal 0.) As you will see when you start to wire that section it seems more natural to compute AC $\neq$ 0. It is certainly true that

AC = 0 is the same as $\overline{AC \neq 0}$

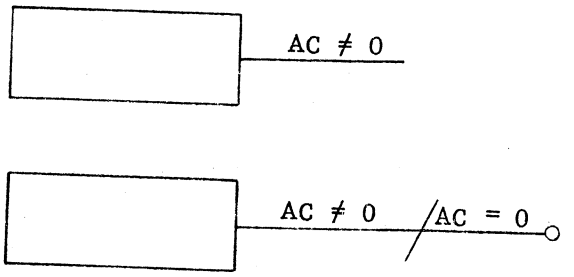
Suppose AC $\neq$ 0 was generated H.



this line high if AC $\neq$ 0

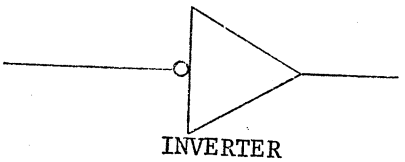
same line will be low if AC = 0

If we want the not of AC ≠ 0 we must draw a slash across the line to remind ourselves that it must be used in a polarity opposite to its generated polarity.



Now we come to another problem. We know that the circuit that is going to use $\overline{AC \neq 0}$ as an input will have to accept it as a L signal. Suppose it expects its inputs H? We need a device (called an inverter) that will fix up the polarity to be what we want. It has the property of outputting a signal of <u>opposite</u> polarity to its input.

Its symbol is:

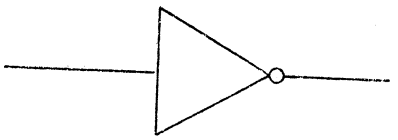


INVERTER

This device will accept an L input and produce an H output. Of course it will also accept an H input and produce an L output.

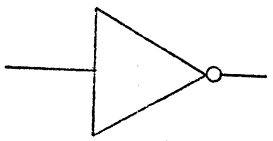
Both behaviors are of course implied in the graphical symbol. As drawn, the behavior for low inputs is emphasized.

An entirely equivalent representation of the <u>same</u> device would be:

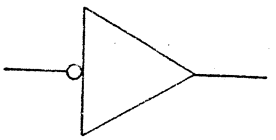


This symbol describes the behavior of an inverter for an H input, which produces an L output. Of course an L input will produce an H output.

Convince yourself that in fact:



is identical to

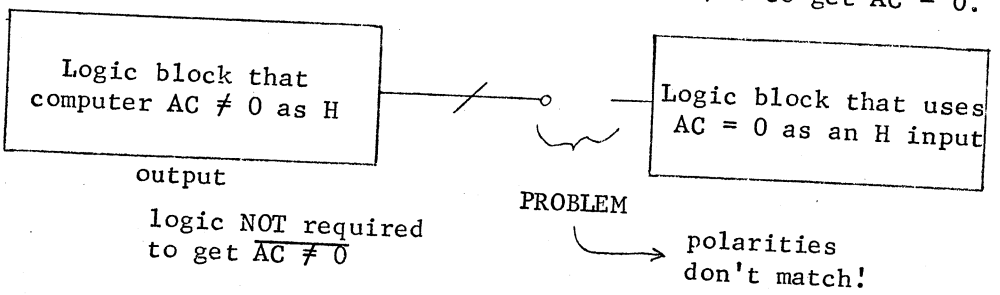


Why draw it two different ways? The logic you are building will naturally dictate which representation to use. Let us reconsider the example of AC = 0. We go through the following steps:

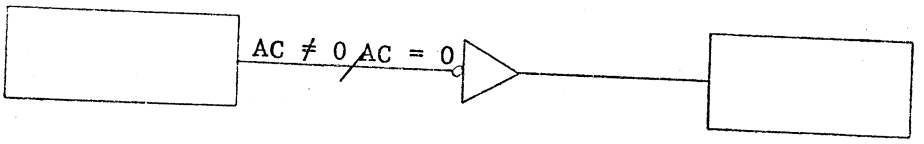
1) We have a logic block which expects AC = 0 to be input as an H signal.

2) With the IC's commonly available we find it more natural to compute AC ≠ 0 which is represented by an H signal.

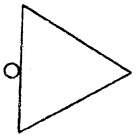
3) We must do a logical NOT operation on AC ≠ 0 to get AC = 0.



4) Solution: Place an inverter in the line to fix up the polarity.



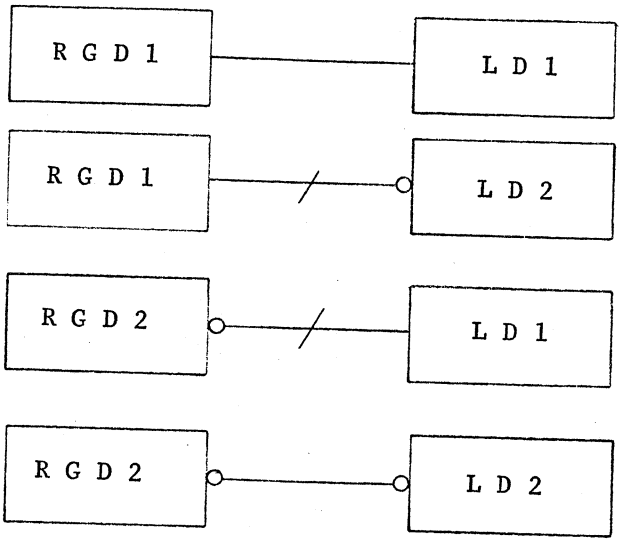
5) In this case there is only <u>one</u> natural way to draw the inverter. Since the signal AC = 0 is produced with an L polarity and we want to use it with an H polarity, we draw the inverter in a form which emphasizes its behavior with L inputs, namely:



To review these concepts suppose we have two different red gorpsch detectors RGD1 and RGD2 with the following properties:

Also assume two different types of lamp drivers, LD1 and LD2:

Let us hook up all four possible combinations and ask when the lamp will be ON.



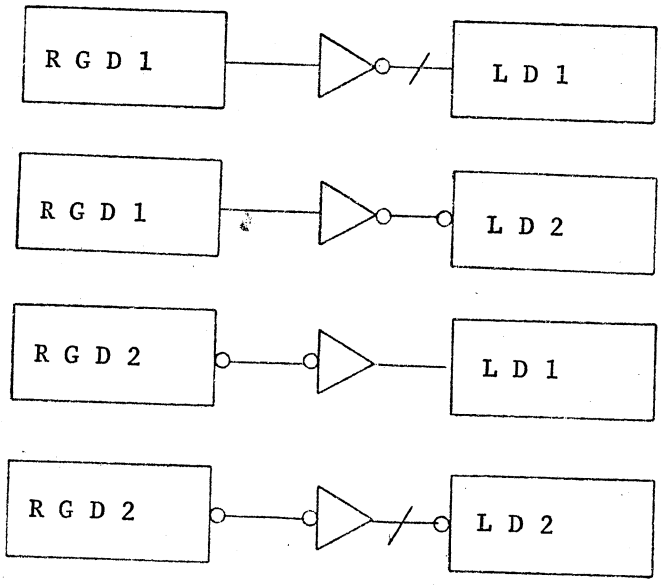
red gorpsch

red gorpsch not

?

?

Do the same four cases but with an inverter inserted between the RGD and LD.



red gorpsch not

red gorpsch

?

?

(3) THE LOGICAL "AND"

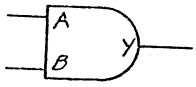
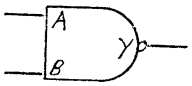
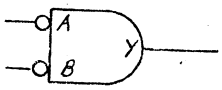
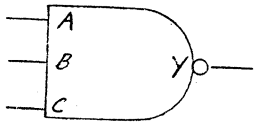
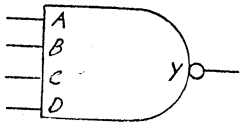
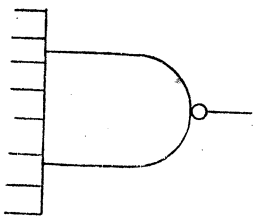
English has several words used to express logical relationships. Without these words it would be difficult to speak. Try it without the following words: NOT, AND, OR, GREATER THAN, LESS THAN, EQUAL, etc.

Logic has similar constructs but a more limited vocabulary. In fact there are just three, NOT, AND, OR. We have covered the logical NOT operation in Chapter 2.

The definition of the logical AND is:

THE OUTPUT WILL BE TRUE IF ALL INPUTS ARE TRUE.

The definition of true follows the usual rules. Small circles represent L voltages. A non-circled line represents an H voltage. AND gates come in many flavors. The common AND gates and their IC catalog numbers are:

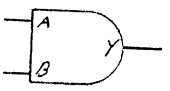
NAME	SYMBOL	CAT. #	BEHAVIOR
2 input AND gate		7408	Y will be H if A AND B are both H
2 input AND gate		7400	Y will be L if A AND B are both H
2 input AND gate		7402	Y will be H if A AND B are both L
3 input AND gate		7410	Y will be L if A AND B AND C are all H
4 input AND GATE		7420	Y will be L if A AND B AND C AND D are all H
8 input AND gate		7430	The output will be L if all inputs are H

There is an equivalent way of describing the logical AND function. It is called the truth table for the device. All possible combinations of inputs are listed in a table and the output for each input combination is listed on the same line. Note that the truth table gives exactly the same information about the 7408 as the graphical symbol. The symbol says that the output will be H only if A AND B are H. All other combinations will give an L output. The symbol has the useful property of emphasizing the last line of the truth table. As a designer if you need a logical AND in fact it is more natural to think only of the last line of the table. Again we see that our symbols help out thought processes by emphasizing only the relevant information.

INPUTS OUTPUT

A	B	Y
L	L	L
L	H	L
H	L	L
H	H	H

7408



The truth tables for the other AND gates are shown below:

7400

A	B	Y
L	L	H
L	H	H
H	L	H
H	H	L

7402

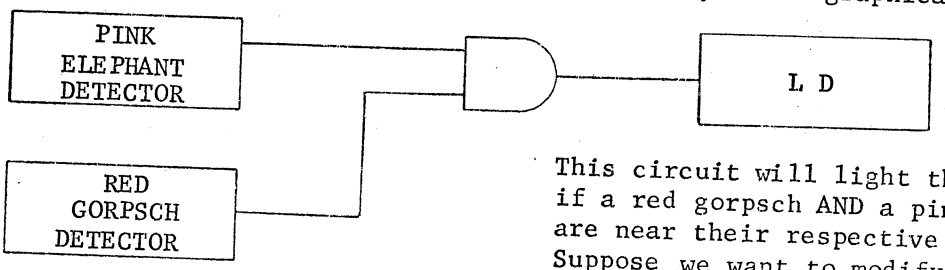
A	B	Y
L	L	H
L	H	L
H	L	L
H	H	L

7410

A	B	C	Y
L	L	L	H
L	L	H	H
L	H	L	H
L	H	H	H
H	L	L	H
H	L	H	H
H	H	L	H
H	H	H	L

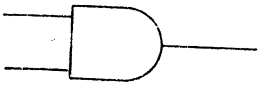
Note that the three input AND truth table has twice as many input combinations as the two input gates. This makes the truth table inconveniently long. As a result, truth tables are seldom used for gates with more than three inputs. How many input combinations are there for the 7430 gate?

Now let us wire up some simple circuits. Assume we have two detectors and a lamp driver with the properties shown by their graphical symbols.



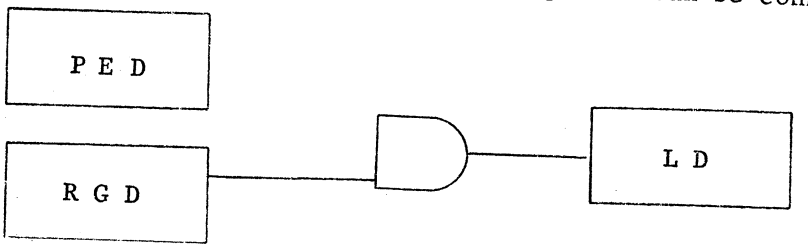
This circuit will light the lamp only if a red gorpsch AND a pink elephant are near their respective detectors. Suppose we want to modify the above circuit to light the lamp only if we have a red gorpsch AND NOT a pink elephant. Go through the following thought processes:

1) We must have a logical AND. Choose a 7408 AND gate and draw it in the middle of the page.

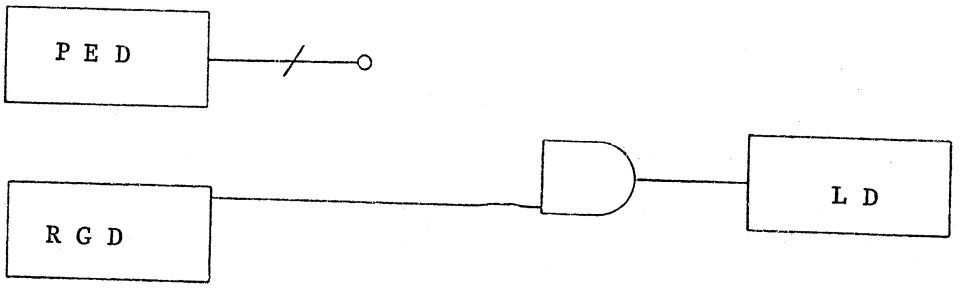


2) The output polarity of the 7408 matches the input polarity of the lamp driver so they can be connected.

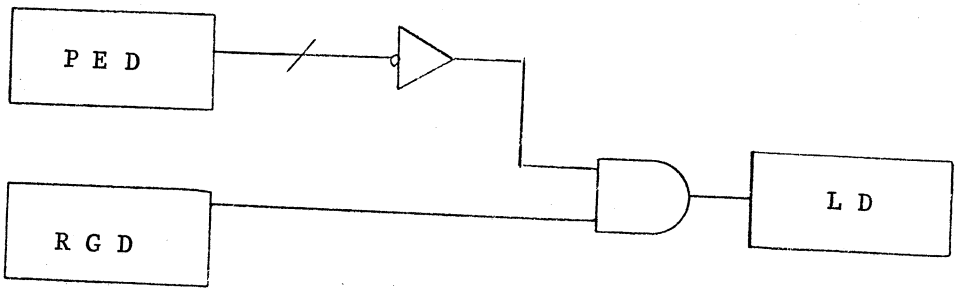
3) The output polarity of the red gorpsch detector matches the input polarity of the 7408 AND gate so these points can be connected.



4) We know we need a logical NOT operation on the output of the pink elephant detector. Draw it.



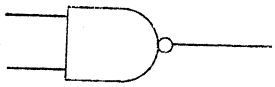
5) Now we need an inverter to change the polarity of $\overline{PE}$ to the high required by the AND gate.



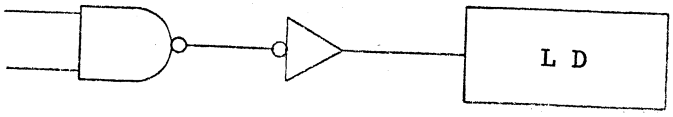
This completes the circuit.

Let's repeat the above example using the 7400 AND gate.

1) Place the 7400 in the middle of the page.

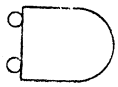


2) The 7400 output doesn't match the input polarity of the lamp driver. It must be changed to the polarity expected by the lamp driver.

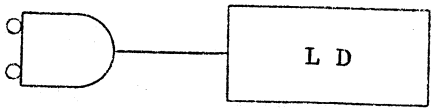


3) The rest of the circuit is identical to the first example. Let's repeat the above example using the 7402 AND gate.

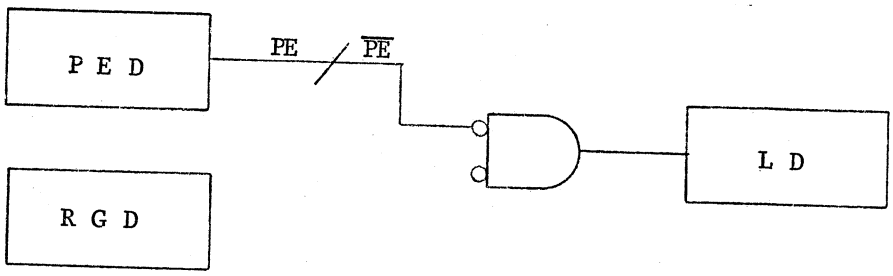
a)



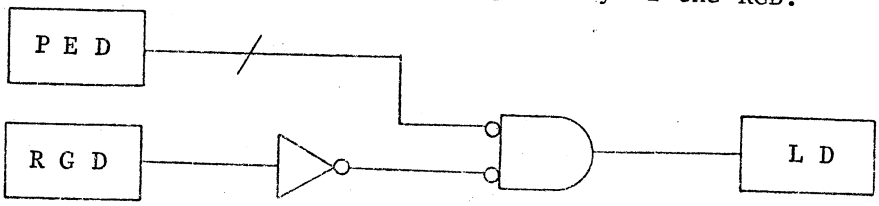
b) The output of the 7402 matches the input polarity of the lamp driver--connect them.



c) The output of the pink elephant detector has to have a slash to create $\overline{PE}$. The resulting polarity matches the input of the 7402.



d) The polarities of the RGD and the 7402 don't match so we must insert an inverter to change the polarity of the RGD.



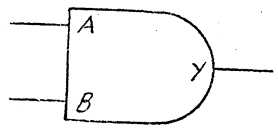
THE BOOLEAN ALGEBRA OF THE AND GATE

There are several fundamental truths about the AND gate which can be neatly summarized by boolean algebra. This is the algebra of binary logic. In all of our previous work we have considered these two values to be H and L voltages. One can conceive of other digital logic systems where the variables are not voltages. An example would be a relay system where we are concerned with a relay contact being OPEN or CLOSED. Boolean algebra abstracts all binary systems by defining two logical variables 1 and 0. The boolean 1 and 0 can then represent H, L voltages or relay contacts being OPEN or CLOSED. A boolean 1 or 0 must not be confused with an ordinary numerical 1 or 0. Instead a boolean 1 is to be interpreted as a logical TRUE and a boolean 0 as a logical FALSE. We have already discussed how the correlation of TRUE and FALSE and voltages is to be made. An output or input with a circle is defined to be TRUE (boolean 1) for an L voltage. An output or input without a circle is defined to be a boolean 1 for an H voltage. The same convention holds for input polarities.

In terms of boolean variables the AND gate is defined by the following truth table:

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Each of the AND gates previously described has an identical boolean truth table when the input voltages are interpreted as boolean 1's or 0's according to the graphical symbol. Let us consider all of the two input AND gates and prove this statement for all of them.



=

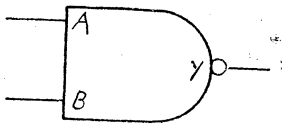
A	B	Y
L	L	L
L	H	L
H	L	L
H	H	H

(if 1=H, for A,B,Y)

=

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Since all of the inputs and outputs are non-circled, a boolean 1 is to be interpreted as an H voltage. If the above truth table is rewritten in terms of 1's and 0's the boolean AND truth table results.



=

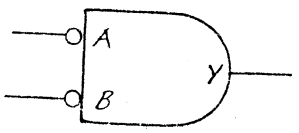
A	B	Y
L	L	H
L	H	H
H	L	H
H	H	L

(if 1=H for A,B) = (and 1=L for Y)

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Rewrite this voltage truth table using an H = 1 for inputs A, B and L = 1 for Y. Again you get the standard truth table for the boolean AND.

The 7402 gate shown on the following page is left as an exercise for the reader.



Now for some boolean algebra. The important theorems are listed below:

$A \cdot B = B \cdot A$ — Look at the truth table for the AND. The only case where $A \cdot B$ could possibly be different from $B \cdot A$ is if B was different from A. In other words A = 1 and B = 0. From the truth table we see that $1 \cdot 0 = 0$ and also $0 \cdot 1 = 0$, QED.

$1 \cdot 1 = 1$ — Verify by looking at the last line of the truth table.

$1 \cdot 0 = 0$ — Verify by looking at the middle lines of the truth table.

$A \cdot 1 = A$

if A = 0, $0 \cdot 1 = 0$
if A = 1, $1 \cdot 1 = 1$ QED

$A \cdot 0 = 0$

if A = 0, $0 \cdot 0 = 0$
if A = 1, $1 \cdot 0 = 0$ QED

$A \cdot A = A$

if A = 0, $0 \cdot 0 = 0$
if A = 1, $1 \cdot 1 = 1$ QED

The last theorem requires a result from a logical NOT operation. The boolean algebra of the NOT operation is so simple it was not covered in that chapter. These results are:

$$\overline{1} = 0$$
$$\overline{0} = 1$$

Now we can state the final theorem on the AND.

$A \cdot \overline{A} = 0$

if A = 1, $A \cdot \overline{A} = 1 \cdot 0 = 0$
if A = 0, $A \cdot \overline{A} = 0 \cdot 1 = 0$ QED

The power of boolean algebra is it allows us to formulate these general results <u>independently</u> from the particular type of logic we are discussing. Although these results are very simple you should memorize them since they will form the basis of some gate simplifications later on.

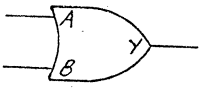
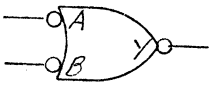
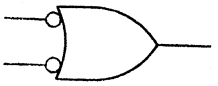
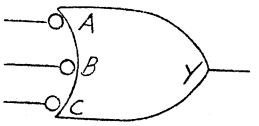
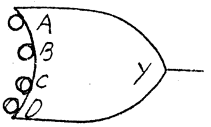
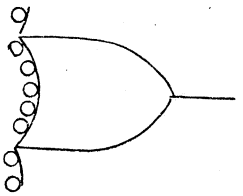
(4) THE LOGICAL OR

The definition of the logical OR is:

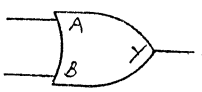
THE OUTPUT WILL BE TRUE IF AT LEAST ONE OF THE INPUTS IS TRUE.

Again the definition of true can be obtained from the symbol for the OR gate. Small circles correspond to L voltages.

The common OR gates and their IC catalog numbers are:

NAME	SYMBOL	CAT. #	BEHAVIOR
2 input OR gate		7432	Y will be H if A OR B (or both) is H
2 input OR gate		7408	Y will be L if one OR more of A, B is L
2 input OR gate		7400	Y will be H if one OR more of A, B is L
3 input OR gate		7410	Y will be H if one OR more of A, B, C is L
4 input OR gate		7420	Y will be H if one OR more of A, B, C, D is L
8 input OR gate		7430	the output will be H if one OR more of the inputs are L

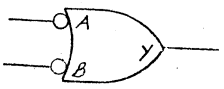
As for the AND gate we can write a truth table for the OR gate. The truth table is an equivalent way of defining the device. Consider the 7432



A	B	Y
L	L	L
L	H	H
H	L	H
H	H	H

Note that the symbol emphasizes the last three rows of the truth table. This says the output will be true if either OR both the inputs are true. The first row is implied by the symbol but if you are designing actual hardware and need an OR gate you are not explicitly concerned with the first row.

The truth table for the 7400 is:



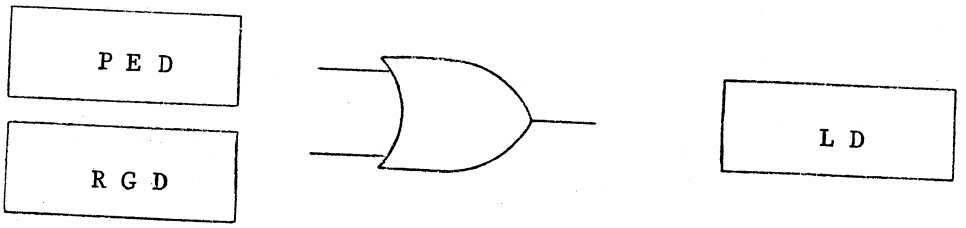
A	B	Y
L	L	H
L	H	H
H	L	H
H	H	L

In this case the symbol emphasizes the first three rows of the truth table. It is just these three rows that embody the logical OR function for this device.

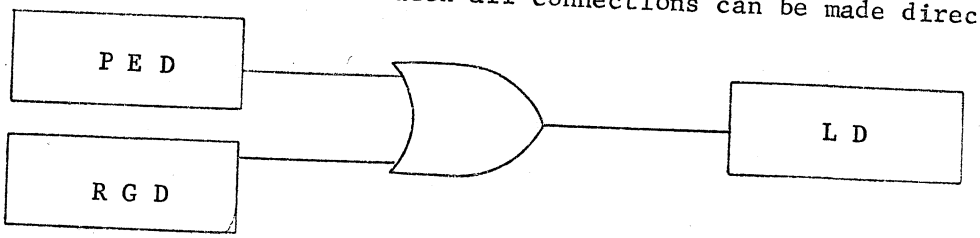
Let us consider some simple examples using OR gates:

Draw a circuit using a 7432 gate that will light a lamp when I have either a red gorpsch OR a pink elephant.

1) Since the OR gate is the central part of the problem draw it in the middle of the page

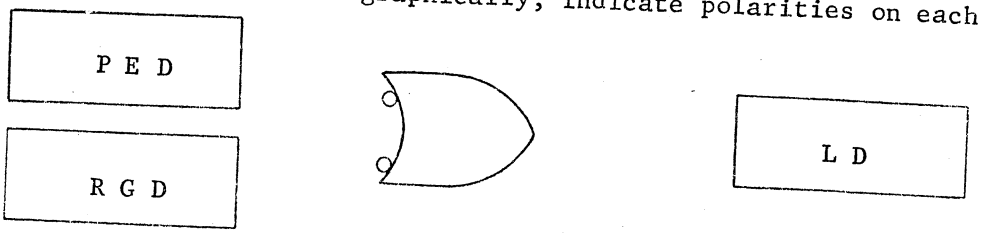


2) Since all polarities match all connections can be made directly.

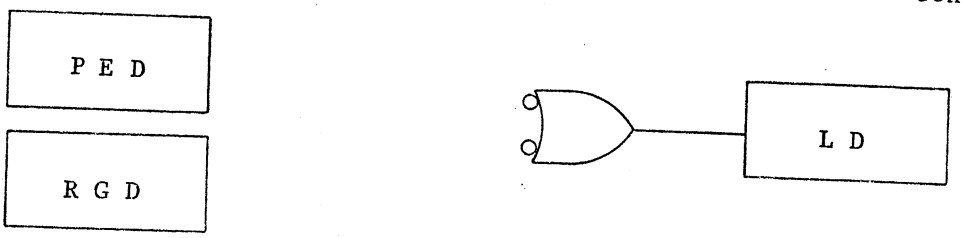


Draw a circuit using a 7400 gate that will light a lamp when we have either NOT a red gorpsch OR a pink elephant.

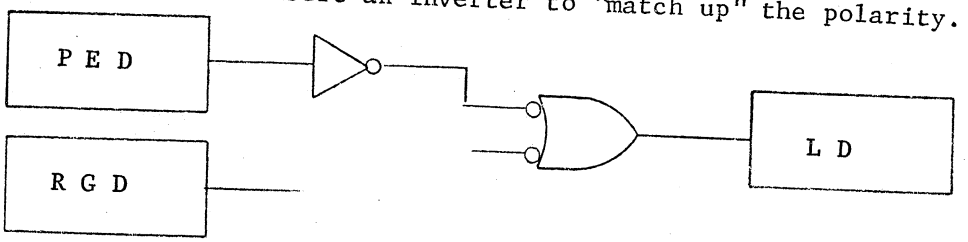
1) Lay out the circuit graphically, indicate polarities on each symbol.



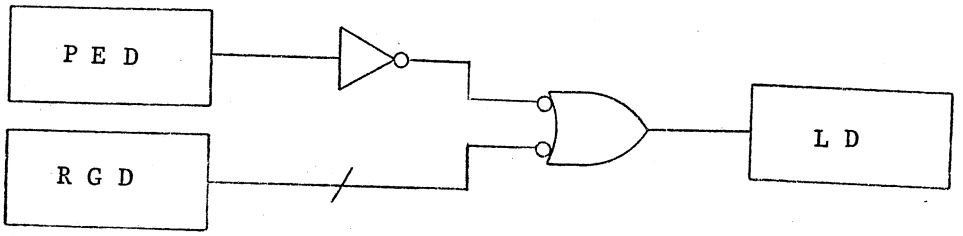
2) The polarities of the 7400 output and lamp driver match--connect them.



3) The polarities of the pink elephant detector and the input of the 7400 don't match. Insert an inverter to "match up" the polarity.



4) Take care of the NOT operation on the red gorpsch line. This means a slash must be put on the output line from the red gorpsch detector. Remember the meaning of the slash--polarities must differ on either side of the slash. This is already true so a director connection can be made.



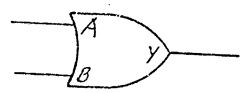
Build the same circuit using the 7402.

THE BOOLEAN ALGEBRA OF THE OR GATE.

The boolean definition of the OR gate is shown by the following truth table:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Each of the OR gates described previously has an identical boolean truth table when the input and output voltages are interpreted as boolean 1's and 0's according to the symbol. Let us prove this for the two input gates.



A	B	Y
L	L	L
L	H	H
H	L	H
H	H	H

input 1=H

output 1=H

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

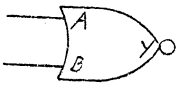
A	B	Y
L	L	H
L	H	H
H	L	H
H	H	L

input 1=L

output 1=H

A	B	Y
1	1	1
1	0	1
0	1	1
0	0	0

Note that the order of the lines in the boolean truth table has been scrambled from the formal definition given above. This of course does not matter since the output depends only on the input combinations.



A	B	Y
L	L	H
L	H	L
H	L	L
H	H	L

input 1=H

output 1=L

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

The student should do the same exercise for the 7408 OR gate.

Let us consider some boolean algebra for the OR gate. The symbol for the boolean OR is a +. This should not be confused with the arithmetic plus. Since in digital logic we deal far more often with the OR than with the plus we give plus a special symbol (+). Thus:

A + B — means A OR B

A (+) B — means arithmetic sum of A and B

The theorems for the OR are listed below:

A + B = B + A — This can be verified by looking at the truth table defining the logical OR

0 + 0 = 0 — First line of defining truth table.

1 + 0 = 1 — Second and third lines of defining truth table.

A + 0 = A — if A = 0, 0 + 0 = 0; if A = 1, 1 + 0 = 1 QED

A + 1 = 1 — if A = 0, 0 + 1 = 1; if A = 1, 1 + 1 = 1 QED

A + A = A — if A = 0, 0 + 0 = 0; if A = 1, 1 + 1 = 1 QED

$A + \bar{A} = 1$ — if A = 0, 0 + 1 = 1; if A = 1, 1 + 0 = 1 QED

DeMorgan's theorems: These are two very important theorems which you will use so often they will become automatic. In each case we will prove them by means of truth tables which is not the most sophisticated proof but is the simplest. Further it will give you more experience handling boolean truth tables.

THEOREM 1: $\overline{A \cdot B} = \overline{A} + \overline{B}$

PROOF a) Form the truth table for $A \cdot B$.
b) Complement the column $\overline{A \cdot B}$ to obtain the left side of the above equation.

Note that complement is another name for the logical NOT. Note that $\overline{A \cdot B} \neq \overline{A} \cdot \overline{B}$ which you should prove by filling in the truth table for $\overline{A} \cdot \overline{B}$.

A	B	A·B	$\overline{A \cdot B}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

c) Form the truth table for $\overline{A} + \overline{B}$ and compare with b).

A	B	$\overline{A}$	$\overline{B}$	$\overline{A}+\overline{B}$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

The last columns of b) and c) agree; therefore we have proved DeMorgan's First Theorem.

THEOREM 2: $\overline{A + B} = \overline{A} \cdot \overline{B}$

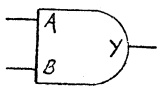
The proof proceeds in the same fashion.

A	B	$\overline{A}$	$\overline{B}$	A+B	$\overline{A+B}$	$\overline{A} \cdot \overline{B}$
0	0	1	1	0	1	1
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	1	0	0

What relationship do DeMorgan's theorems have to AND, OR gates? Said in words the theorems say:

IF ALL INPUTS AND OUTPUTS ARE INVERTED, AND and OR WILL BE INTERCHANGED.

Symbolically:



A	B	Y
L	L	L
L	H	L
H	L	L
H	H	H

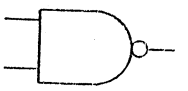
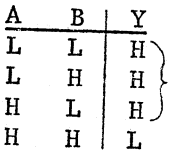
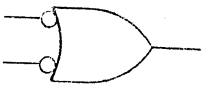
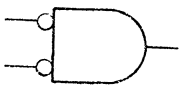
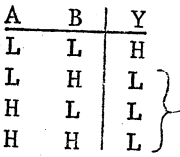
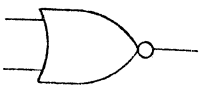
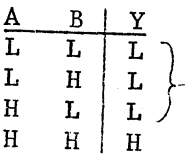
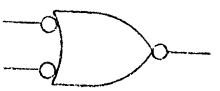
This is true as we'll show by using voltage truth tables: The AND symbol emphasizes the last line of the truth table (the AND line) and implies the other three lines.

Now let us take the first three lines of the same voltage truth table (the OR lines) and represent their behavior with a graphical symbol.

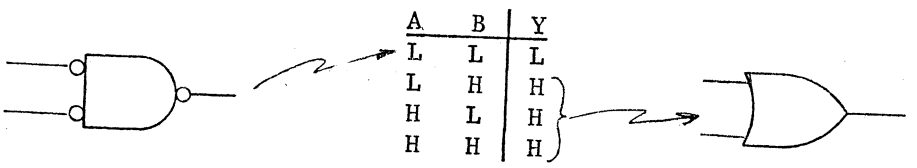
A	B	Y
L	L	L
L	H	L
H	L	L
H	H	H

We see that the AND and OR symbol emphasize different parts of the same truth table. Of course either symbol implies the entire truth table. Still as designers we will be thinking of AND or OR functions. YOUR SYMBOLS SHOULD REPRESENT THE TYPE OF GATE (AND or OR) THAT YOU WERE THINKING OF WHEN YOU DESIGNED THE CIRCUIT. THE SYMBOLS WILL IN TURN INDICATE THE POLARITY REQUIRED TO MAKE THEM ACT THE WAY THEY ARE DRAWN.

This concept is so important that we will illustrate it for all of the commercially available two input gates.

CAT. #	SYMBOL	VOLTAGE TRUTH TABLE	EQUIVALENT SYMBOL
7400			
7402			
7408			

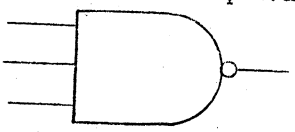
7432

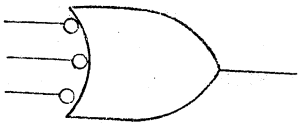


A	B	Y
L	L	L
L	H	H
H	L	H
H	H	H

THUS WE CAN USE ONE AND THE SAME GATE AS A LOGICAL AND or A LOGICAL OR BY GIVING IT INPUTS OF THE PROPER POLARITY!

This is not restricted to two input gates. For example the 7410 can be represented in two equivalent forms.





The student must learn to "change gears" automatically when thinking of AND or OR gates, since this will allow you to simplify circuits.

(5) LOGIC EQUATIONS

As you have seen we have devised a very nice graphical way of representing digital logic circuits. The pictures tell us what operations (NOT, AND, OR) are taking place and also what polarities are at any point in the circuit. This pictoral representation called a LOGIC DIAGRAM is the most useful way of portraying a circuit that is already built.

Unfortunately it is a rather cumbersome way to do the initial design of a circuit. A more compact way to represent the boolean operations is required. This information can be represented by boolean equations. Again we will find that they are tailored to certain applications. They are useful precisely because they emphasize certain things and suppress irrelevant items. One thing suppressed is voltage polarity. Indeed, from a preliminary design viewpoint we are interested in implementing AND's, OR's, NOT's, etc., and are not concerned with polarities. Of course when it comes time to build or debug a circuit then polarities are all important. At that point a logic diagram will be needed. This chapter will be devoted to translating logic equations <u>to</u> logic diagrams.

The symbols used to write equations are divided into two classes:

Variables: These are simply the names of signals. Examples would be: A, A39, AC LOAD. A superficially more complex name would be AC=0. Such a name tells you the condition that will make the line true. For AC=0 it would be true only if all bits in the accumulator were = 0. Note the = sign is <u>part</u> of the name in this case.

Operators:

.	logical AND
+	logical OR
$\overline{\quad}$	logical NOT (this is an overscore)
( )	parenthesis (used as in ordinary algebra)
=	equals (used as in ordinary algebra unless it is part of a name)

Let us consider some simple examples:

$$F = A \cdot B$$

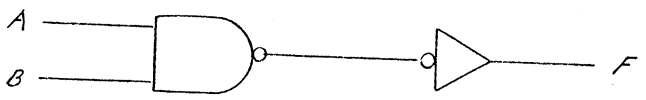
An equivalent way of representing the same information would be a logical truth table:

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

If you want to build a logic circuit to generate F you must provide two additional pieces of information:

a) What type of AND gate you will choose from the IC catalog
b) What polarities A, B, and F will be represented by.

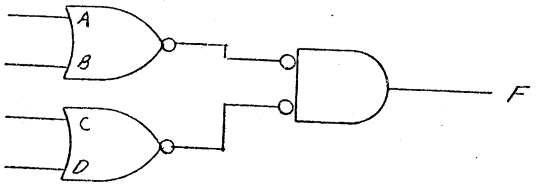
Suppose the gate is a 7400 and TRUE for A, B, F is represented by H. Now the circuit can be drawn



Another example:

$$F = (A + B) \cdot (C + D)$$

The parenthesis tells us that A + B must be generated as an intermediate signal, as must C + D. Then these two intermediate signals must be ANDed. Again suppose F, A, B, C, D are all TRUE when H. Also suppose the 7402 is chosen for the gates. All three gates are 7402's. We have used DeMorgan's theorem to represent the same gate in its most natural form at each place in the circuit.



Such simple examples do not really express the power of boolean equations. Before we can go to more realistic examples it is necessary to consider the implied priority of the different operators. Consider:

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

This could mean:

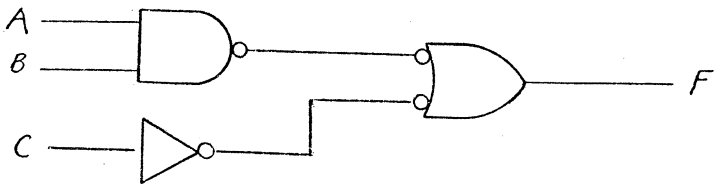
$F = A \cdot (B + C)$
$F = (A \cdot B) + C$

(+ has more priority than ·)
(· has more priority than +)

The proper interpretation is the second. We can formalize this in a table of priorities (hierarchies) the most "powerful" operator being at the left:

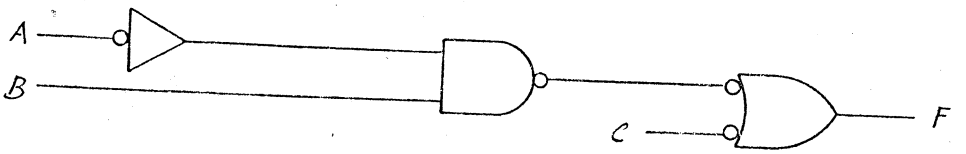
	NOT	AND	OR
( )	$\overline{\quad}$	·	+

Let us now derive a circuit that will generate $F = A \cdot B + C$ where TRUE is represented by H and we choose 7400 gates for implementation.

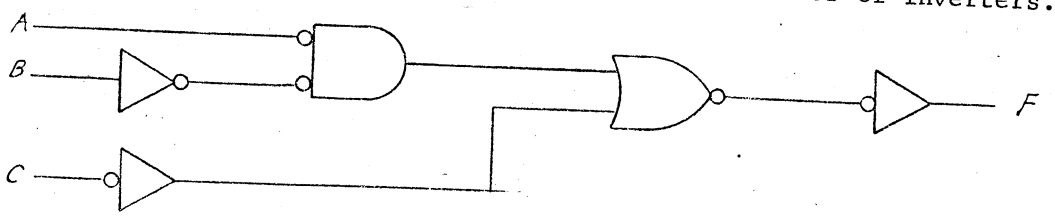


Let us implement $F = A \cdot B + C$ where TRUE for each variable is defined by F = H, A = L, B = H, C = L.

Again let us choose 7400 gates for implementation.



This logic diagram looks very different from the preceding one. Yet, it is the same boolean equation that is being implemented. The difference is caused by the differing polarities chosen to represent TRUE. Still another factor that could change the appearance would be to choose a different gate to implement the circuit. Suppose that 7402's were chosen and the polarities are the same. We see that it takes two more inverters to implement the same boolean equation. It is generally true that one type of gate will be more "natural" for a given situation. The designer should exploit this by choosing the type of gate which minimizes the number of inverters.



Logic equations can also be used to state general theorems. A few of these theorems should be memorized since they will allow you to build circuits with fewer gates.

The important theorems are listed below:

1a) $A + 0 = A$	1b) $A \cdot 1 = A$
2a) $A + 1 = 1$	2b) $A \cdot 0 = 0$
3a) $A + A = A$	3b) $A \cdot A = A$
4a) $A + \overline{A} = 1$	4b) $A \cdot \overline{A} = 0$

These theorems have already been discussed in the chapters on the logical AND and OR. They should be at your fingertips since it is unforgivable to convert one of these equations to gate form.

5a) $A\ (A + B) = A$ 5b) $A + AB = A$

Note that when two single letter variables such as A, B are written side by side the AND is implied thus $AB = A \cdot B$. This is true only if the variables are named by <u>single</u> letters.

These boolean identities can be proved by truth tables. For example 5a):

A	B	A+B	A(A+B)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

We see that the columns for A and $A \cdot (A + B)$ match QED.

Alternatively they could be proved using boolean algebra. For example 5b):

$$A + AB = A(1 + B) = A \cdot (1) = A$$

6a) $A(\overline{A} + B) = AB$ 6b) $A + \overline{A}B = A + B$

6a) is easy to prove using boolean algebra:

$$A(\overline{A} + B) = A\overline{A} + AB = 0 + AB = AB$$

7a) $\overline{AB} = \overline{A} + \overline{B}$ 7b) $\overline{A + B} = \overline{A} \cdot \overline{B}$

These are simply DeMorgan's theorems.

8a) $AB + A\overline{B} = A$

This is an important theorem which is the basis of the Karnough Map simplification. Its proof is simple: $AB + A\overline{B} = A(B + \overline{B}) = A \cdot (1) = A$.

(6) USEFUL STRUCTURES BUILT FROM GATES AND INVERTERS

The human mind works best when irrelevant material can be suppressed. We have already seen how each of our various ways of representing digital logic emphasizes certain things and de-emphasizes others. Consequently, we use the description that best suits the task at hand.

The next step up this ladder is to combine gates into useful structures. Once we do this we can give this structure a name. Once named the human mind can then use the new structure easily. We will describe and name most of the common items used in computer architecture.

A) EXCLUSIVE OR

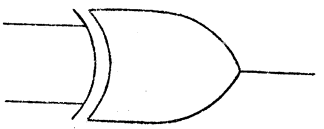
This is a logic function of two inputs and one output. The definition can be given either by a truth table or a boolean equation. We will do both, but we must first give it a boolean operator symbol which is $\oplus$.

which is equivalent to
$A \oplus B = A\bar{B} + \bar{A}B$

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

From the truth table we can see that it is identical to the logical OR except it <u>excludes</u> the case A = B = 1.

The exclusive OR is used enough to warrant its production as a special integrated circuit, the 7486. The graphical symbol is modified from the normal symbol of the logical OR: Note also that it has only two inputs.



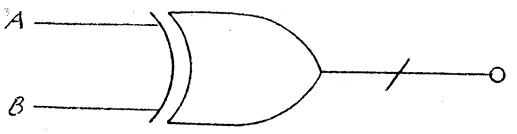
a) Use as a controlled inverter: This technique should be in every designer's bag of tricks, especially since it is so simple. Suppose we wish to pass a signal (A) unchanged when a control signal (C) is 0 and invert A when the control signal is 1.

This is the exclusive OR definition as we can show from the truth table:

when C=0 $A \oplus C = A$
when C=1 $A \oplus C = \bar{A}$

C	A	A⊕C
0	0	0
0	1	1
1	0	1
1	1	0

b) Use as a comparator: Look at the truth table again and you will notice that the output of $A \oplus B$ is a 1 only when $A \neq B$. Therefore, all we have to do to see if both bits are the same is take the logical NOT of the output.



(L only when A = B)

c) Use as a parity generator/checker: Parity is a reliability feature that is used on most computers. The philosophy behind it is to try to give an automatic warning when some component of the computer fails. One common device that is parity checked is memory. The common failure mode of memory is that one bit will always come back a 0, the other failure mode is for it to always come back a 1. Some examples are:

	INPUT	OUTPUT
good memory	0011 1010 0001	0011 1010 0001
bad memory	0011 1010 0001	0011 1010 0000 (last bit always 0)
bad memory	0011 1010 0001	1011 1010 0001 (first bit always 1)

Of course with a good memory you will always read back what you stored. When the memory goes bad you will either "PICK UP" or "DROP" a bit. How could you check for this? Consider the original word stored in memory 0011 1010 0001; it has 5 bits =1. Note that 5 is an ODD number. If the last bit is dropped the number returned from memory will be 0011 1010 0000 which has 4 bits = 1 which is an EVEN number. If a bit is picked up, i.e., 1011 1010 0001, 6 bits = 1, which is STILL an even number. Thus a bit pickup or drop causes the number of bits to change from ODD to EVEN.

SUCH AN ERROR IS CALLED A PARITY ERROR.

Of course we will not always be storing numbers with an odd number of bits; for example we might store 0010 0011 1000 which has 4 bits = 1 (EVEN), and a bit pick up or drop will result in

a number with an ODD number of 1's which is still a parity error. How do you distinguish these cases? You append an extra bit (the parity bit) to the original number so the total number of bits (including the parity bit) is odd. For example:

1) 0011 1010 0001 is already ODD — parity bit = 0
final number is 0 0011 1010 0001 — total bits = 1 is ODD
parity bit original number

2) 0010 0011 1000 has an even number of 1 bits; therefore the parity bit = 1
final number is 1 0010 0011 1000 — total bits = 1 is ODD
parity bit original number

Now let us send these 13 bit numbers to a faulty memory. The parity bit has been appended so that the number of 1 bits is <u>always</u> odd. Now a bit drop or pickup will <u>always</u> result in an even number of 1 bits.

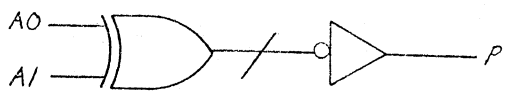
If we had some way of looking at a number and telling whether it had an odd number of bits we could build a circuit that would detect a parity error.

Look at the exclusive OR truth table again. It has two input bits. Its output is a 1 only if the number of 1 bits in the input is <u>ODD</u>.

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

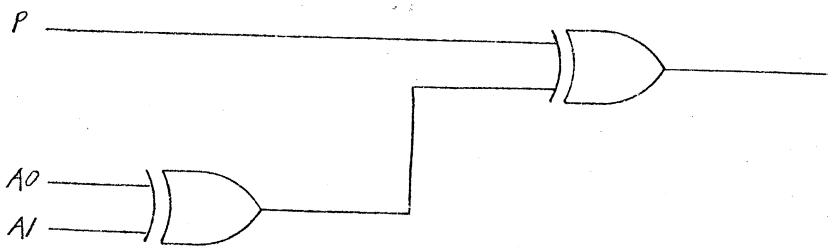
only 1 bit of A or B is = 1 (1 is an odd number)

Thus if we have a 2 bit number (A0, A1), we can calculate the parity bit to append to it (P, A0, A1) with the following circuit.

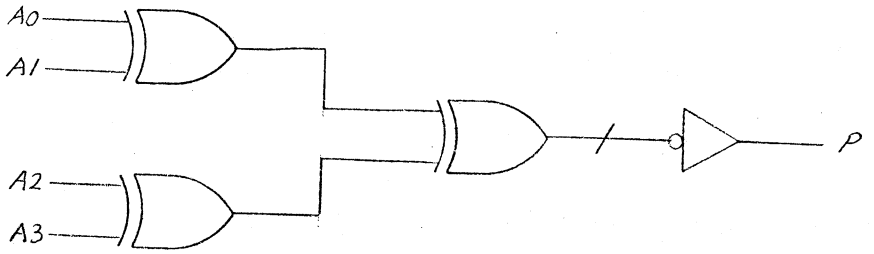


P = H if A0, A1 = L,L
or A0, A1 = H,H

We can ship this modified number P, A0, A1 to a faulty memory which may be picking up or dropping a bit. How do we tell? We take the number we get back from memory and use exclusive OR's to see if it <u>still</u> has an <u>ODD</u> number of bits. If yes, we did not have a parity error. The circuit to check this is shown below:

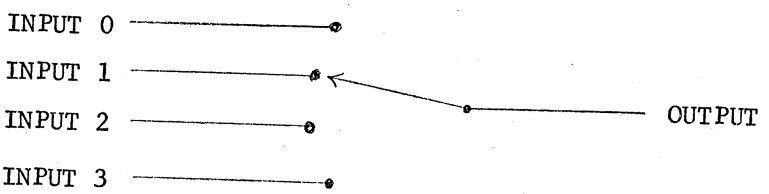


This is a very simple parity checking circuit in that the original number A0, A1 was only 2 bits. A more typical width would be the word size of the computer. For the PDP-8 this is 12 bits. How would you generate the parity bit to append to the 12 bit word? Hint: shown below is the circuit to generate the parity bit for a 4 bit word.



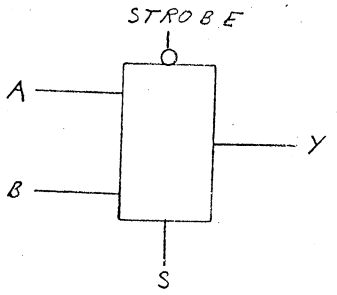
B) THE MULTIPLEXOR (DATA SELECTOR)

The data selector (often abbreviated MUX) is essentially an electronically controlled switch. Its function is to take many inputs, select only one of them, and route the selected one to the output. It is exactly analagous to a mechanical rotary switch as shown below:



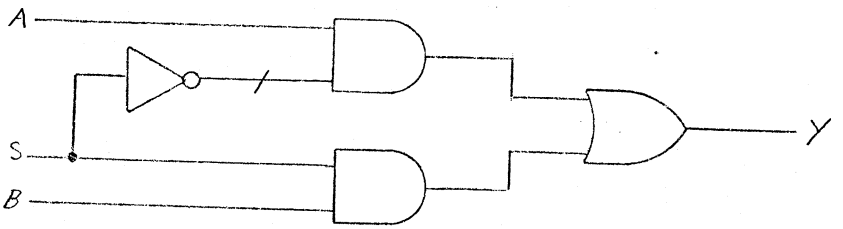
With a manual switch the position is controlled by a knob. MUX's act like the rotary switch but are made of AND, and OR gates. This makes the selection process very fast since gates switch in just a few nano seconds. There are either 1, 2, 3, or 4 switch selection lines in IC MUX's. Therefore, there can be $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, or $2^4 = 16$ different inputs. We will describe these in turn:

1) 2 INPUT MUX catalog number 74157

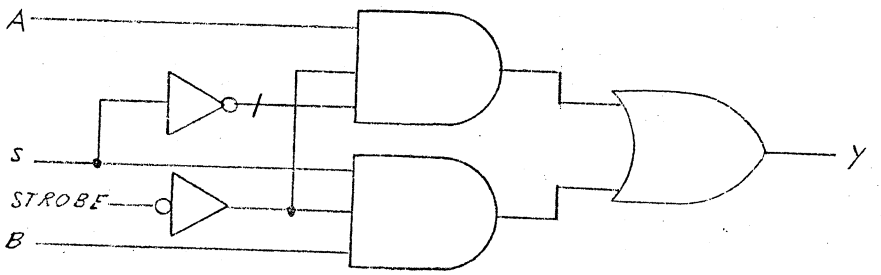


Since it is a special purpose circuit its symbol is a rectangle. A, B are inputs which can be switched to Y by means of the control line S. The strobe line is also important; unless it is L none of the inputs are connected to Y. All of the following discussion assumes strobe = L.

When S = L, A is connected to Y. When S = H, B is connected to Y. For the moment ignore strobe; what combination of AND, OR gates would implement this circuit?

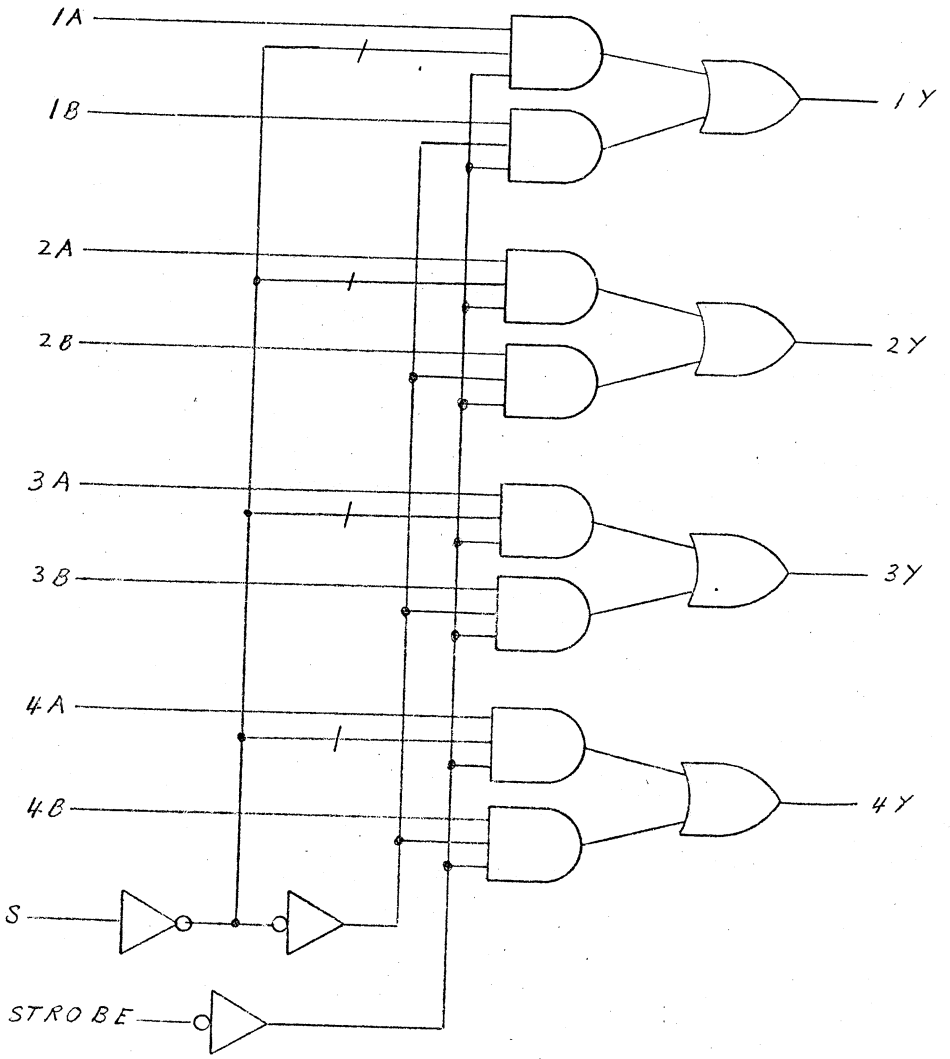


Convince yourself that this circuit does indeed duplicate the above description. How can we implement STROBE? (Remember when strobe = H, all inputs are disconnected from Y).



To get the actual circuit for the 74157 we have to consider one obscure fact. Each gate input absorbs some electrical power. There are four identical two input MUX's per IC package. If we simply took four of the previous circuits and hooked all four select (S) lines to the same pin we would draw four times as much power from the gate which generated the S signal elsewhere in our computer. This would LOAD that gate excessively.

Therefore, we use two inverters on the S line. We take our select signals off of these outputs. Now we draw only one unit of power from the gate which is generating S.



Several comments are in order:

a) AND, OR gates don't look anything like a rotary switch but they will simulate it, and much faster too.

b) You buy four of these switches per IC package at a cost of roughly $1.50.

c) It is much easier to think of a MUX than the gates inside the MUX. After all in a computer we are going to have to route information inside the machine. Even though this is done by gates they clutter up the picture.

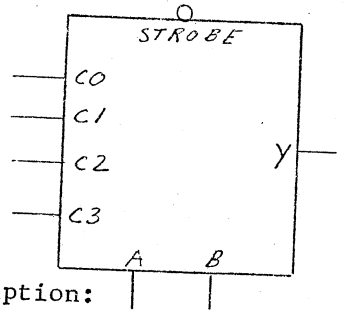
d) We could have described the MUX by means of equations, i.e., $Y = A \cdot STROBE \cdot \overline{S} + B \cdot STROBE \cdot S$.

e) We could have described the device by means of a truth table:

Note the use of X
X = irrelevant, that is H or L

STROBE	S	A	B	Y
H	X	X	X	L
L	L	L	X	L
L	L	H	X	H
L	H	X	L	L
L	H	X	H	H

2) 4 input MUX catalog number 74153



NOTE: the four inputs are called C0--C3. The select lines are now called A, B. Strobe has the same meaning as before.

Description:

a) $Y = (C0 \cdot \overline{A} \cdot \overline{B} + C1 \cdot \overline{B} \cdot A + C2 \cdot B \cdot \overline{A} + C3 \cdot B \cdot A) \cdot STROBE$

b) Truth table

STROBE	B	A	C0	C1	C2	C3	Y
H	X	X	X	X	X	X	L
L	L	L	L	X	X	X	L
L	L	L	H	X	X	X	H
L	L	H	X	L	X	X	L
L	L	H	X	H	X	X	H
L	H	L	X	X	L	X	L
L	H	L	X	X	H	X	H
L	H	H	X	X	X	L	L
L	H	H	X	X	X	H	H

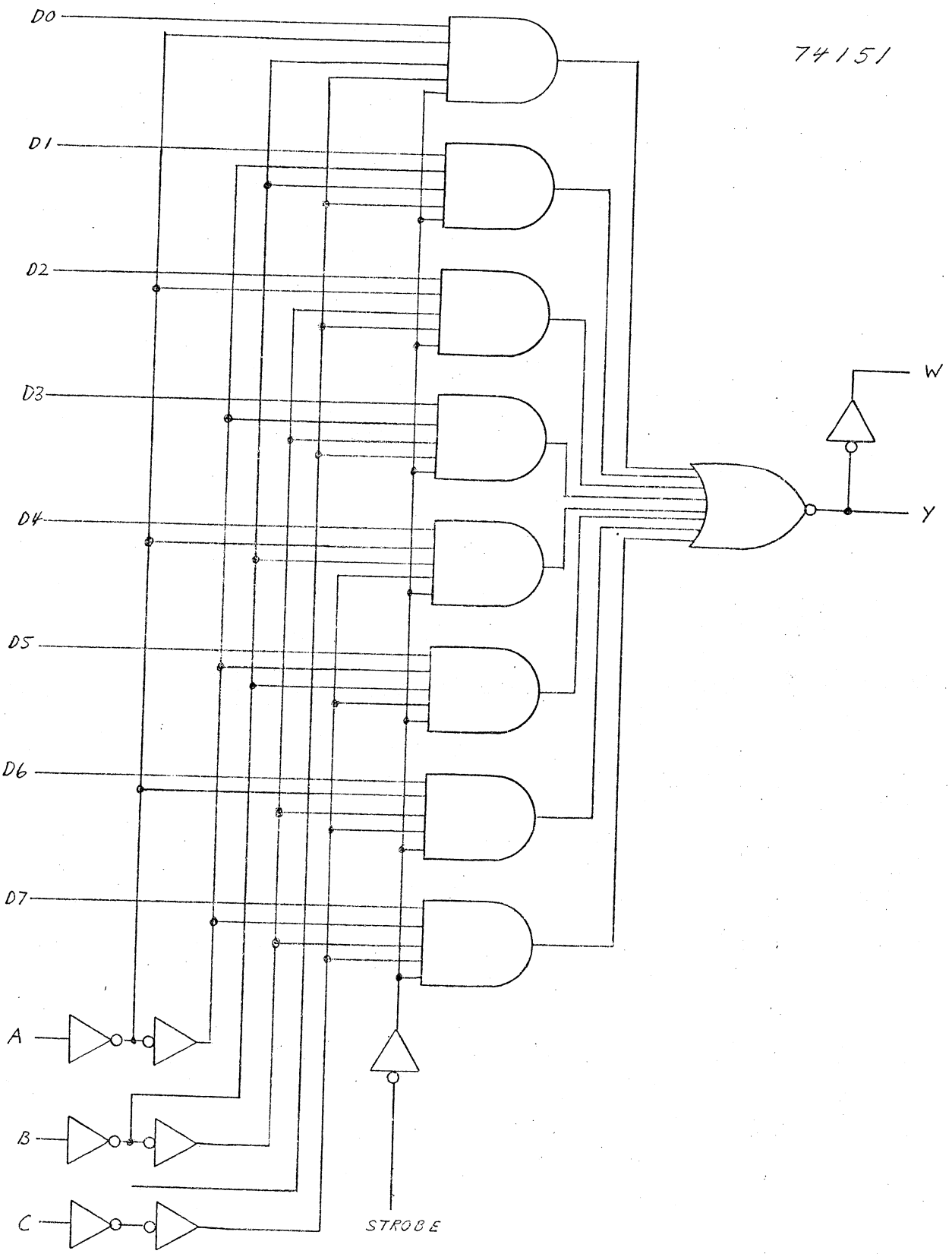
c) As an exercise draw a logic diagram using AND, OR gates that implements a, b above

3) 8 input MUX catalog number 74151

4) 16 input MUX catalog number 74150

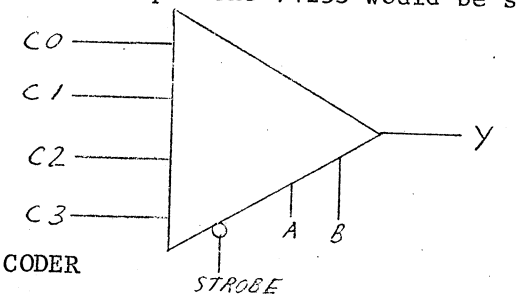
These are identical devices to 1 and 2 except for the increased number of inputs. Figure 6.1 is the logic diagram of the 74151 MUX.

Fig. 6.1



NOTE: Y outputs are circled on 150, 151
Y outputs are uncircled on 153, 157
W output on 151

One last point: Some authors use a triangle to represent a MUX. For example the 74153 would be shown as:



C) THE DECODER

Often we will have to look at a collection of bits and pick out a certain combination. For example in the laboratory computer project the left three bits of a command word tell what kind of instruction it is. Let us label this three bit field IR0, IR1, IR2. The possible combinations and their meanings are:

IR0	IR1	IR2	INSTRUCTION
0	0	0	AND accumulator and memory
0	0	1	TAD accumulator and memory
0	1	0	ISZ increment and skip if zero
0	1	1	DCA store and clear accumulator
1	0	0	JMS jump to subroutine
1	0	1	JMP jump
1	1	0	IØ input output instruction
1	1	1	NMR non-memory reference instruction

Now the question is how do we tell when we have a given instruction? This can be most easily shown by a boolean equation. For example:

NMR = IR0·IR1·IR2 since NMR is the only instruction with all bits TRUE.

Satisfy yourself that each of the following equations is correct:

IØ = IR0·IR1·$\overline{IR2}$

JMP = IR0·$\overline{IR1}$·IR2

JMS = IR0·$\overline{IR1}$·$\overline{IR2}$

DCA = $\overline{IR0}$·IR1·IR2

ISZ = $\overline{IR0}$·IR1·$\overline{IR2}$

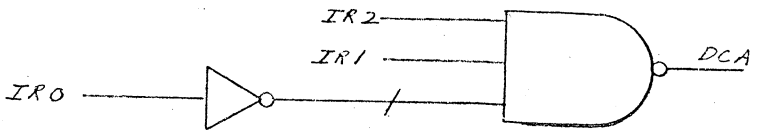
TAD = $\overline{IR0}$·$\overline{IR1}$·IR2

AND = $\overline{IR0}$·$\overline{IR1}$·$\overline{IR2}$

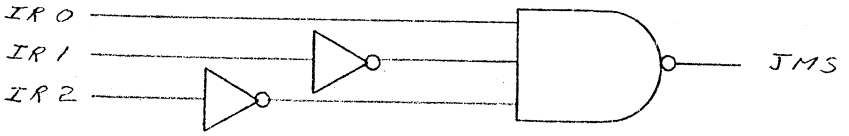
This process is called instruction decoding and the circuits to accomplish it are called decoders.

Let us build a decoder to detect the presence of the DCA instruction. Assume that IR0, IR1, IR2 are TRUE when H and DCA will be true when L.

$$DCA = \overline{IR0} \cdot IR1 \cdot IR2$$



For JMS = $IR0 \cdot \overline{IR1} \cdot \overline{IR2}$



Decoding all eight instructions would take six more circuits analagous to the two above. The integrated circuit manufacturers have been nice enough to package all eight of these circuits into one IC.

The most common decoders (and therefore the cheapest) are the 7442, which decode 10 outputs and the 74154 which decode 16 outputs. Both are used in your lab kit. A schematic of the 7442 is shown on figure 6.2. The schematic of the 74154 is identical except it has six more gates to decode the input combinations from 1010 - 1111. It also has a pair of enable terminals which must both be low to activate the decoder.

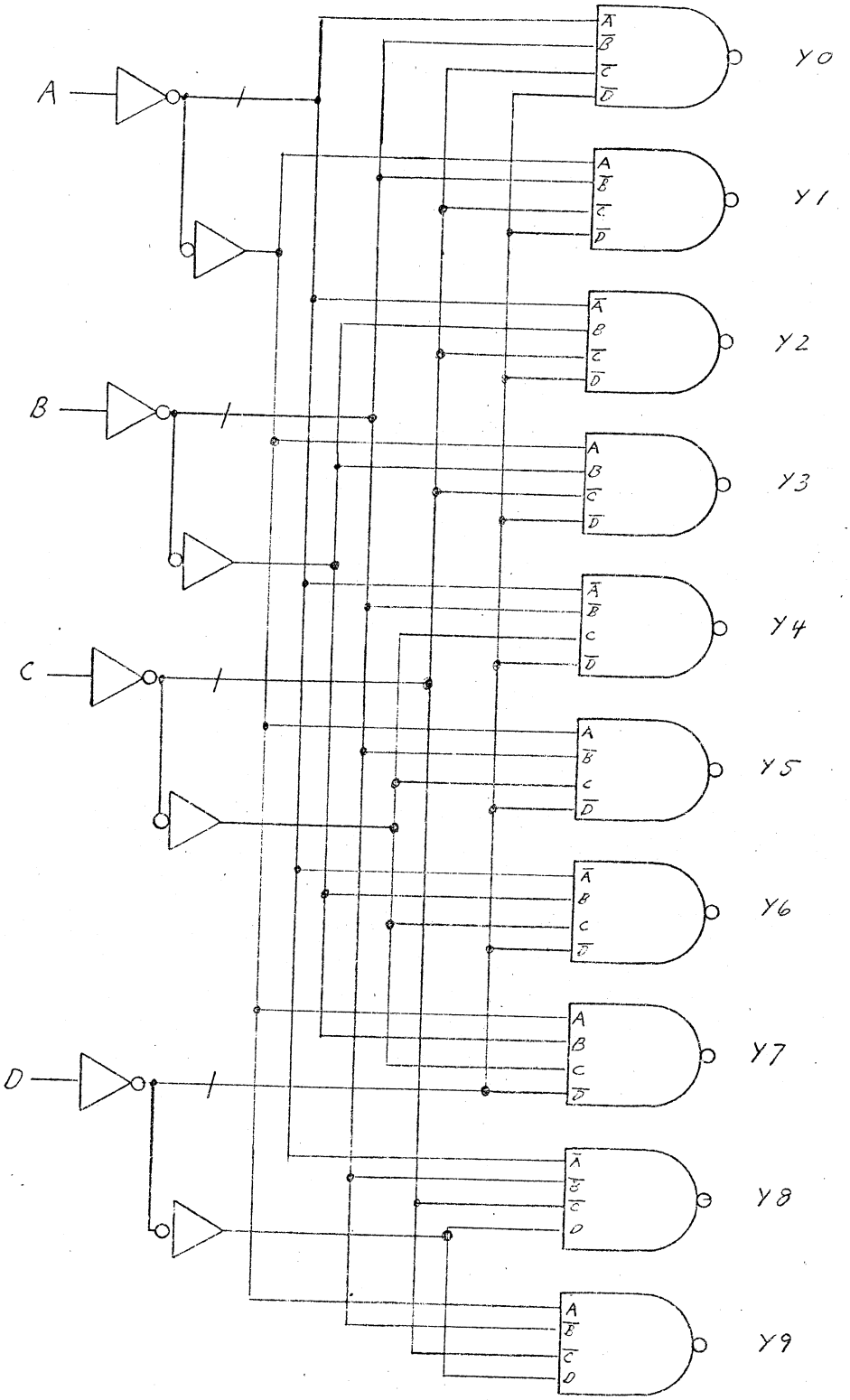
The enable capability of a decoder is very important. For example we discussed the use of a decoder to generate a unique signal for every command. It should be intuitively obvious that we would want to look at these signals only during the execution portion of a computer cycle. In other words we want to <u>enable</u> the instruction decoder only during the execute cycle.

At first sight the 7442 does not have this capability. However, if we choose to look at the first eight outputs only, then the D input will serve as an enable. You can see this by examining the decode gates for outputs 0 - 7. Every one of these gates has a $\overline{D}$ input. Thus they will be enabled only if D = 0.

The student should look in a commercial IC catalog (such as the Texas Instruments TTL data book) for the logic diagram of the 74154.

There is one more very important combinatorial circuit, the binary adder. To allow adequate space for its discussion we have devoted the entire next chapter to it.

Fig. 6.2



7442

(7) ADDITION

A computer that can't add obviously isn't worth much. It is fairly easy to build a combinatorial circuit to add binary numbers. This chapter is devoted to a slow and easy introduction to the process and will culminate in an actual circuit that will add two numbers.

In the introduction we saw how a decimal number can be represented using only 0's and 1's. We call the resulting representation a binary number. To refresh your memory we will give the binary representation of several numbers:

$5_{10} = 101_2$ $\qquad$ $7_{10} = 111_2$

$17_{10} = 10001_2$ $\qquad$ $143_{10} = 1000\ 1111_2$

$34_{10} = 10\ 0010_2$ $\qquad$ $53_{10} = 11\ 0101_2$

We know how to add two decimal numbers but let us review the process since we can use it as a guide to binary addition.

Add without carries:

$$\begin{array}{r} 34 \\ +\ 5 \\ \hline 39 \end{array}$$

Here we can add the 5 + 4 and get a number smaller than 10; therefore, there is no carry into the 10's column.

$$\begin{array}{r} 27 \\ +\ 61 \\ \hline 88 \end{array}$$

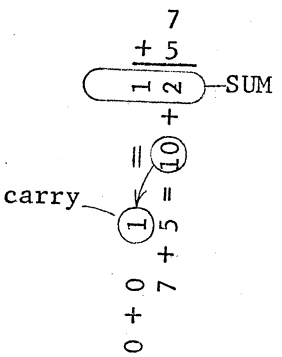
Again the sum in each column is less than 10; there are no carries so each column can be added independently

Add with carries:

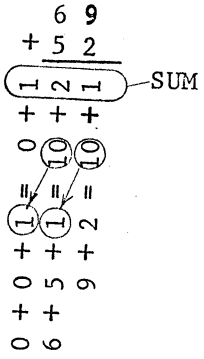
$$\begin{array}{r} 7 \\ +\ 5 \\ \hline 12 \end{array}$$

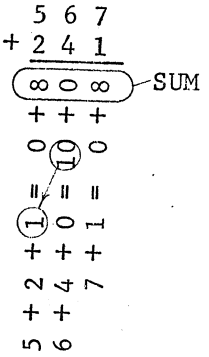
Here the situation is slightly more complex. The largest digit we can have in any column is 9, but 7 + 5 is larger than 9. We solve this by a carry into the 10's column which says 7 + 5 = 10 + 2. Remember the carry = 10 and the 2 is how much we have to add to the 10 to get the answer.

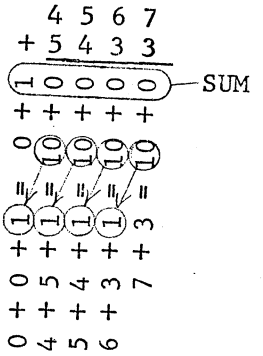
It is somewhat unconventional but we can show this very nicely in the following fashion:



Consider more complex examples:







The rules describing this process are:

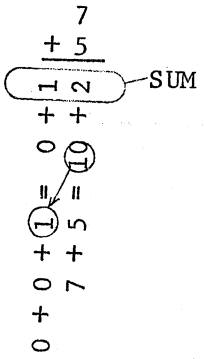
a) The number of different digits is 10 (0, 1, . . . 9).
b) If the sum in a given column is less than 10 there is no carry into the next column.
c) If the sum in a given column is 10 or more there is a carry into the next column.

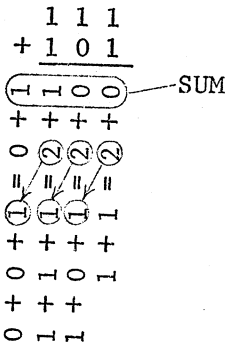
Analagous rules hold for binary addition:

a) The number of different digits is 2 (0, 1)
b) If the sum in a given column is less than 2 there is no carry.
c) If the sum in a given column is 2 or more there is a carry.

To show how similar decimal and binary addition are we will do the same example side by side.

$7_{10} = 111_2$ $\qquad$ $5_{10} = 101_2$

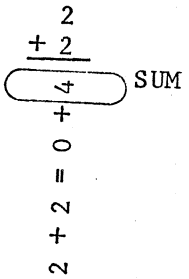


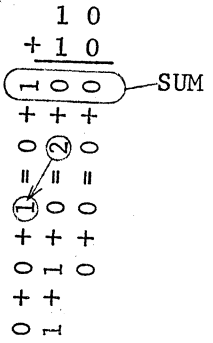


$1100_2 = 12_{10}$

$2_{10} = 10_2$

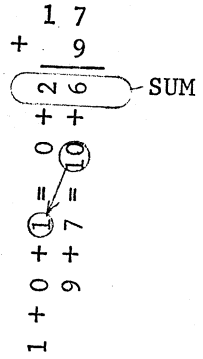
$2_{10} + 2_{10} = 4_{10}$



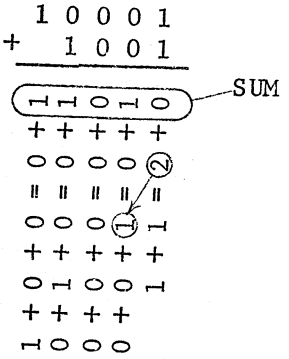


$100_2 = 4_{10}$

$17_{10} = 10001_2$



$9_{10} = 1001_2$



$11010_2 = 26_{10}$

In the preceding examples we assumed you knew the addition tables. For binary addition the tables are very simple:

carry from preceding column	0	0	0	0	1	1	1	1
A	0	0	1	1	0	0	1	1
+ B	0	1	0	1	0	1	0	1
SUM	0	1	1	0	1	0	0	1
carry to next column	0	0	0	1	0	1	1	1

This same information is more usually arranged in a form that looks like a truth table. C_i represents a carry into this column. C_o represents a carry out of this column.

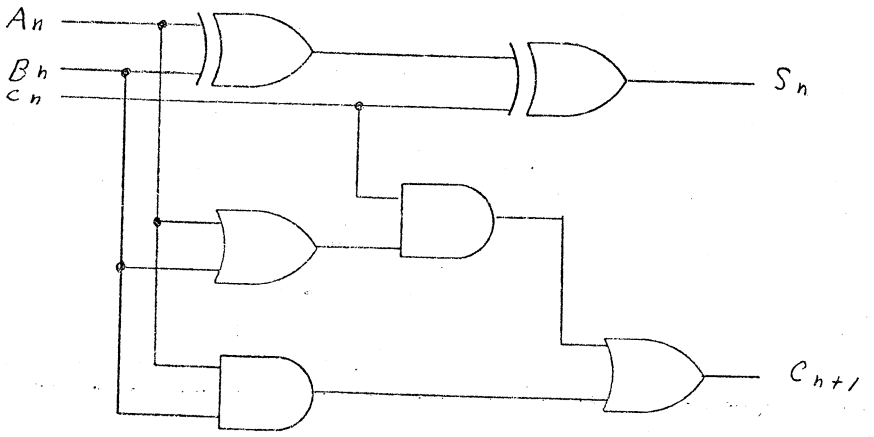
C_i	A	B	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

IMPORTANT PROBLEM!

Prove: $S = A \oplus B \oplus C_i$

$$C_o = A \cdot B + A \cdot C_i + B \cdot C_i$$
$$= A \cdot B + C_i (A + B)$$

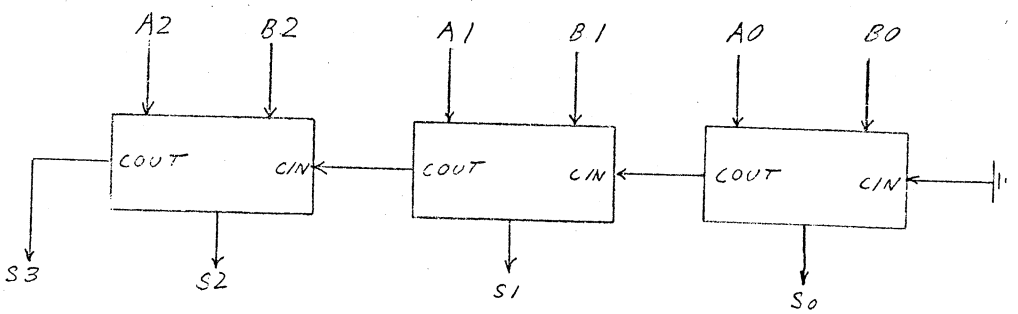
These equations can be translated to hardware as shown below:



We have changed the nomenclature slightly. A, B have been subscripted with a small n to signify that we are working with column n of a binary number. C_n denotes the carry into column n and C_{n+1} denotes the carry out of this column.

Let us enclose the above circuit in a special symbol box and show how adders can be connected together to add two 3 bit numbers. Label the columns as follows:

$$\begin{array}{cccc} & A_2 & A_1 & A_0 \\ + & B_2 & B_1 & B_0 \\ \hline S_3 & S_2 & S_1 & S_0 \end{array}$$



Note that $C_0 = 0$; therefore it is tied to ground to make it always L.

One might think that the full adder would be available as an IC. Indeed it is; but a more common IC (7483) contains 4 full adders in a single package. The carry lines are internally connected so there are only two carry lines; C in to the 4 bit group and C out of the 4 bit group.

SUBTRACTION

We could go through an analagous procedure to derive the algorithm for subtraction and finally wind up with hardware to do it. Fortunately we can use an adder to subtract. We will base our discussion on an ordinary electronic calculator with 6 digits of accuracy. If one is not available you can think of an imaginary 3 digit calculator. Our immediate goal is to use such a calculator to subtract by using only the add key.

To do this we need to define a <u>complement</u> number. In the following discussion we will assume our imaginary 3 digit calculator. If you have an electronic hand calculator by all means use it to do the same examples.

To form a 3 digit complement of a number n, subtract it from 1000.

Complement of 237 = 1000 - 237 = 763
Complement of 64 = 1000 - 64 = 936
Complement of 700 = 1000 - 700 = 300

Let us subtract first using normal arithmetic and secondly by addition of a complement.

$$\begin{array}{r} 514 \\ -\ 237 \\ \hline 277 \end{array} \qquad \begin{array}{r} 514 \\ +\ 763 \\ \hline 1277 \end{array} \qquad\qquad \begin{array}{r} 514 \\ -\ 64 \\ \hline 450 \end{array} \qquad \begin{array}{r} 514 \\ +\ 936 \\ \hline 1450 \end{array}$$

In each case the answer obtained by complement addition is exactly 1000 larger than the number obtained by normal subtraction. This of course must be true since a complement is formed by subtraction from 1000.

$$514 + (1000 - 237) = 1000 + (514 - 237)$$

We see that we could do a subtraction very nicely by complement addition if: a) we had an easy way to reject the 1000,
b) we had an easy way to independently form the complement.

Fortunately the first is automatic. We assumed we had <u>only</u> a 3 digit calculator. The 1000 lies in the fourth column so it is <u>automatically</u> discarded.

The second problem is only slightly more complicated. We need some way of subtracting the number from 1000 that is so easy that is does not burden the process. The main problem in subtraction for humans is the borrow problem. If only we could devise a way to subtract column by column without a borrow we could write the complement almost as fast as we could write the original number. There is an easy way to do this:

$$1000 = 999 + 1 \qquad\qquad 1000 - 237 = (999 - 237) + 1$$

Now it is impossible to get a borrow in subtracting <u>any</u> three digit number from 999. You can see this by considering only one column at a time.

The largest digit you can have is a 9 and a 9-9 = 0 with no borrow. Since we have no borrows we can now subtract from right to left or left to right. Let's use this procedure to form the complement of some numbers.

Complement of 237:

$$\begin{array}{ccc} 9 & 9 & 9 \\ -\,2 & -\,3 & -\,7 \\ \hline 7 & 6 & 2 \end{array} \quad + \; 1$$

So to subtract 237 from 514

add

$$\begin{array}{r} 514 \\ +\,762 \\ \hline 1276 \end{array}$$

then add 1

$$\begin{array}{r} 1276 \\ +\quad 1 \\ \hline 1277 \end{array}$$

└ automatically discarded

NOTE: We have used only the addition process. (We assumed an independent easy way for forming complements).

Complement of 64:

$$\begin{array}{ccc} 9 & 9 & 9 \\ -\,0 & -\,6 & -\,4 \\ \hline 9 & 3 & 5 \end{array} \quad + \; 1$$

$$\begin{array}{r} 514 \\ -\,64 \\ \hline 450 \end{array} \qquad \begin{array}{r} 514 \\ +\,935 \\ \hline 1449 \end{array}$$

add 1

$$\begin{array}{r} 1449 \\ +\quad 1 \\ \hline 1450 \end{array}$$

└ automatically discarded

We call the number obtained by subtracting from 999 the NINE's complement. We define the number obtained by subtracting from 1000 the TEN's complement. Thus: TENS Complement = NINES Complement + 1
Again the reason for breaking the tens complement up into two steps is the ease of doing each step.

Now let us see how we can handle a calculator of more than three digits. The crucial thing is the automatic discard of the one to the left of the difference:

For a 3 digit calculator use $1000 = 10^3$
For a 4 digit calculator use $10000 = 10^4$
For a 6 digit calculator use $1{,}000{,}000 = 10^6$
For an n digit calculator use $1{,}000 \ldots = 10^n$

Now that we have shown how complement addition works for decimal numbers, we can move to binary numbers. The process of taking complements is even easier in binary. By analogy with the calculator example if our binary computer has n bits the complement will be formed by subtracting from 2^n. Definition: 2's complement of $x = 2^n - x$.

For example suppose we have a 3 bit machine:

$$\begin{array}{r} 7 \\ -\,5 \\ \hline 2 \end{array} \qquad \text{2's complement of 5} = 1000 - 101 = 11$$

$$\begin{array}{r} 7 \\ -\,5 \\ \hline 2 \end{array} \qquad \begin{array}{r} 111 \\ +\,11 \\ \hline 1010 \end{array} \qquad 010_2 = 2_{10}$$

↑ automatically discarded since have only a 3 bit machine.

We will find that it much easier to take the complement in two steps as we did for decimal numbers.

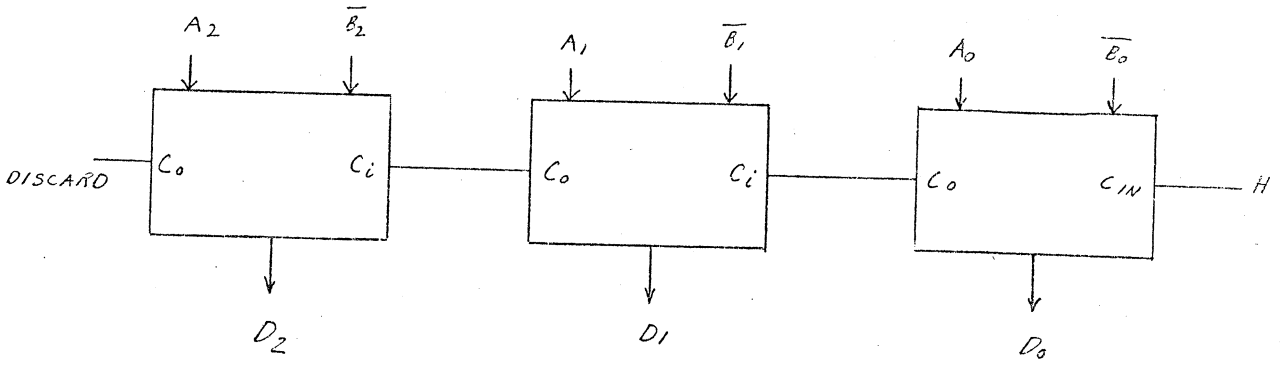
$$2^3 = 1000_2 = 111 + 1 \qquad \begin{array}{r} 111 \\ -\,101 \\ \hline 010 \end{array}$$

NOTE: The final result is a bit by bit NOT (complement) of the denominator.
Definition: The ONE's complement of a binary number is formed by changing each 1 to a 0 and each 0 to a 1.

TWO's complement of X = ONE's complement of X + 1.

As you can guess the ones complement of a binary number is very easy to generate electronically. Further the extra 1 to be added can be generated simply by turning on the carry in to the right most adder (for addition it is always turned off).

We can show this process by means of a logic diagram.

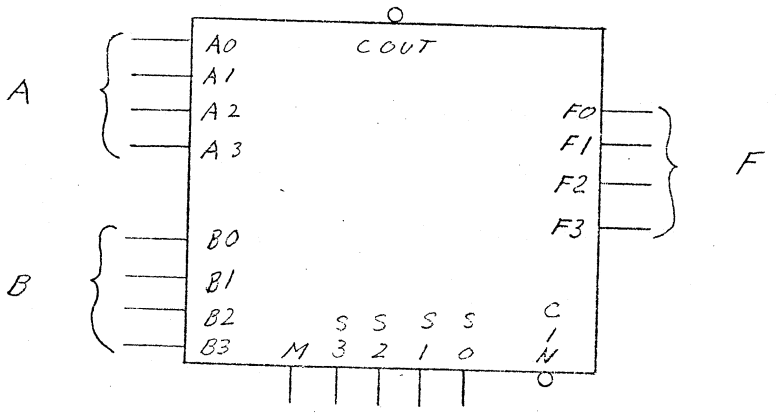


(8) ARITHMETIC UNIT

We have studied most of the combinatorial elements required for a computer. The last and most important one will be covered in this chapter. It is called the arithmetic logic unit (ALU). Its function is to take data words and perform arithmetic operations (such as add) or logical operations (such as AND) in response to instructions from a program stored in memory.

Building an arithmetic unit with early integrated circuits was quite a chore since only simple gates and inverters were available. Such IC's are called SSI (Small Scale Integrated) circuits. Technology has progressed to the point where more complex circuits can now be manufactured. MSI (Medium Scale Integration) contains circuits of approximately 100 gates per IC. LSI (Large Scale Integration) contains approximately 1000 gates per package. The computer you will construct uses all three classes of IC's. SSI is used for implementing simple logic equations. LSI is used in the memory and MSI is used to build the arithmetic unit.

The MSI circuit you will use is a remarkable device. It will be worth studying it in detail since it is the heart of the computer. The device is the 74181 which is capable of performing all possible logical operations (AND, OR, etc.) as well as arithmetic operations on two 4 bit operands. These devices can be strung together to form arithmetic units which will handle more than 4 bits. Since our computer is a 12 bit machine, it will require three 74181's. The symbol for the device is:



A, B, and C_{in} are the inputs; F and C_{out} are the outputs. M and S are control inputs which tell the ALU what operation to perform on A, B, and C_{in}.

Since there are five control inputs 32 (2^5) different operations can be performed by the 74181. These are broken down into two groups of 16 based on the value of M. When M = L it will do arithmetic operations (adding) and when M = H it will do logic operations. You may wonder how these can be 16 such operations since we have discussed only NOT, AND, OR, and EXCLUSIVE OR. Let us imagine a "black box" with two inputs A, B, and one output F. The question we are asking is how many different kinds of "black boxes" can there be. This in turn forces us to ask how

do we know when we have a particular "black box," such as an AND gate. The answer is of course by means of the truth table that the black box produces. For example if we input all four combinations of A, B and obtain this output on F:

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

Then we know that this particular black box is indeed an AND gate. There are exactly 16 different possible truth table outputs as shown below:

A	B	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Boolean equation		0	$A \cdot B$	$A \cdot \overline{B}$	A	$\overline{A} \cdot B$	B	$A \oplus B$	$A + B$	$\overline{A + B}$	$A \odot B$	$\overline{B}$	$A + \overline{B}$	$\overline{A}$	$\overline{A} + B$	$\overline{AB}$	1

Some of these truth tables are familiar to you, for example:

$F_1 = A \cdot B$

$F_3 = A$

$F_5 = B$

$F_6 = A \oplus B$

$F_7 = A + B$

$F_{12} = \overline{A}$

$F_{10} = \overline{B}$

We will now define some of the others:

F_8 = NOR (NOT OR) since $F_8 = \overline{A + B}$

F_{14} = NAND (NOT AND) since $F_{14} = \overline{A \cdot B}$

F_9 = COIN (COINCIDENCE) since $F_9 = 1$ only if $A = B$

Note that $A \odot B = \overline{A \oplus B}$ The symbol for F_9 is $\odot$.

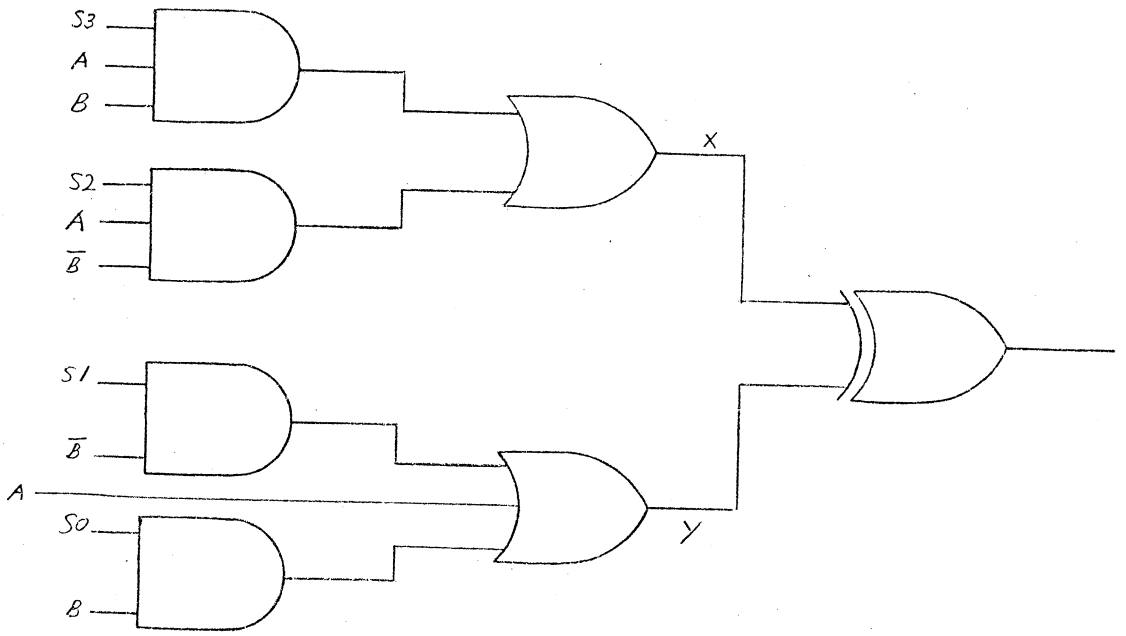
The other entries are of little use and we will not bother to name them. In any event they can be expressed by means of boolean equations, for example:

$$F_2 = A \cdot \overline{B}$$

The 74181 is capable of outputting any one of these 16 truth tables depending on the values of M and S. For example if M = H, S_3 = L, S_2 = H, S_1 = H, S_0 = L, then:

$$F_0 = A_0 \oplus B_0$$
$$F_1 = A_1 \oplus B_1$$
$$F_2 = A_2 \oplus B_2$$
$$F_3 = A_3 \oplus B_3$$

A circuit which will generate all 16 possible truth tables is shown below:



The operation of this circuit can best be analyzed by using boolean algebra. Write the boolean expressions for X and Y and then note that $X \oplus Y = X\overline{Y} + \overline{X}Y$. Let us consider X and Y separately:

S3	S2	$X = A\cdot(S3\cdot B + S2\cdot\overline{B})$
0	0	0
0	1	$A\overline{B}$
1	0	AB
1	1	A

in detail:

S3	S2	$A\cdot(S3\cdot B + S2\cdot\overline{B})$
0	0	$A\cdot(0\cdot B + 0\cdot\overline{B}$
		$A\cdot(0 + 0)$
		$A\cdot(0)$
		0
0	1	$A\cdot(0\cdot B + 1\cdot\overline{B})$
		$A\cdot(0 + \overline{B})$
		$A\cdot\overline{B}$
1	0	$A\cdot(1\cdot B + 0\cdot\overline{B})$
		$A\cdot(B + 0)$
		$A\cdot B$
1	1	$A\cdot(1\cdot B + 1\cdot\overline{B})$
		$A\cdot(B + \overline{B})$
		$A\cdot(1)$
		A

S2	S1	$Y=A + S1\cdot\overline{B} + S0\cdot B$
0	0	A
0	1	A + B
1	0	B + $\overline{B}$
1	1	1

in detail:

0 0

$A + 0\cdot\overline{B} + 0\cdot B$
$A + 01 + 0$
$A + 0$
A

0 1

$A + 0\cdot\overline{B} + 1\cdot B$
$A + 0 + B$
$A + B$

1 0

$A + 1\cdot\overline{B} + 0\cdot B$
$A + \overline{B} + 0$
$A + \overline{B}$

1 1

$A + 1\cdot\overline{B} + 1\cdot B$
$A + (\overline{B} + B)$
$A + 1$
1

Now if we write all 16 combinations of S and tabulate the above values of X and Y we get:

S3	S2	S1	S0	X	Y	$Z = X \oplus Y = X\overline{Y} + \overline{X}Y$
0	0	0	0	0	A	A = F3
0	0	0	1	0	A+B	A+B = F7
0	0	1	0	0	A+$\overline{B}$	A+$\overline{B}$ = F11
0	0	1	1	0	1	1 = F15
0	1	0	0	A$\overline{B}$	A	AB = F1
0	1	0	1	A$\overline{B}$	A+B	B = F5
0	1	1	0	A$\overline{B}$	A+$\overline{B}$	$\overline{A\oplus B}$ = F9
0	1	1	1	A$\overline{B}$	1	$\overline{A}$+B = F13
1	0	0	0	AB	A	A$\overline{B}$ = F2
1	0	0	1	AB	A+B	A$\oplus$B = F6
1	0	1	0	AB	A+$\overline{B}$	$\overline{B}$ = F10
1	0	1	1	AB	1	$\overline{A\cdot B}$ = F14
1	1	0	0	A	A	0 = F0
1	1	0	1	A	A+B	$\overline{A}$B = F4
1	1	1	0	A	A+$\overline{B}$	$\overline{A+B}$ = F8
1	1	1	1	A	1	$\overline{A}$ = F12

Some of the results in the Z column are easy to obtain:

For the first four note that $0 \oplus P = P$
For the last one in each group of four note that $1 \oplus P = \overline{P}$

Also remember DeMorgan's theorems:

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

Some of the others are not so easy! Let us do

S3	S2	S1	S0	X	Y
0	1	0	1	$A\overline{B}$	$\overline{A+B}$

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

$$X\overline{Y} = A\overline{B}\ \overline{(A + B)}$$

$$= A\overline{B}\ (\overline{A}\ \overline{B}) \qquad \text{DeMorgan}$$

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

$$= 0 \cdot \overline{B}$$

$$= 0$$

$$\overline{X}Y = (\overline{AB})\ (A + B)$$

$$= (\overline{A} + B)\ (A + B) \qquad \text{DeMorgan}$$

$$= \overline{A}A + \overline{A}B + BA + BB$$

$$= 0 + \overline{A}B + BA + B$$

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

$$= B\ (1 + 1)$$

$$= B\ (1)$$

$$= B$$

Therefore $Z = X \oplus Y = X\overline{Y} + \overline{X}Y = 0 \oplus B = B$

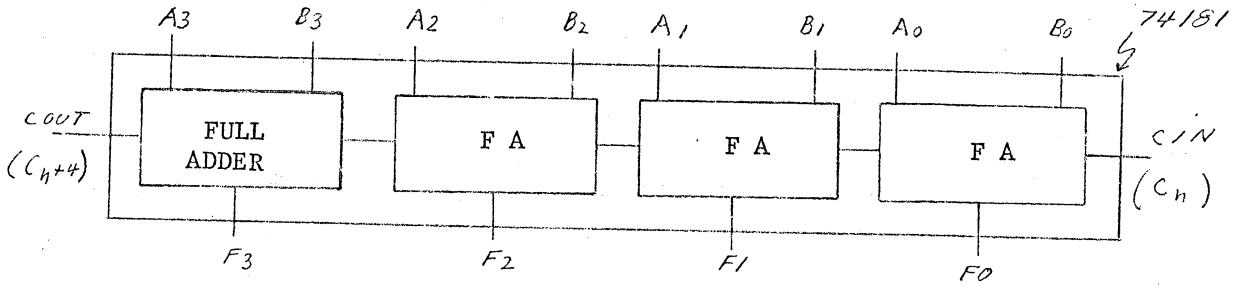
It is not essential that you be able to work through the other cases if you are willing to accept them on faith. The essential things to note are that the 74181 is able to do:

a) $F = A \cdot B$
b) $F = A + B$
c) $F = A$
d) $F = \overline{A}$

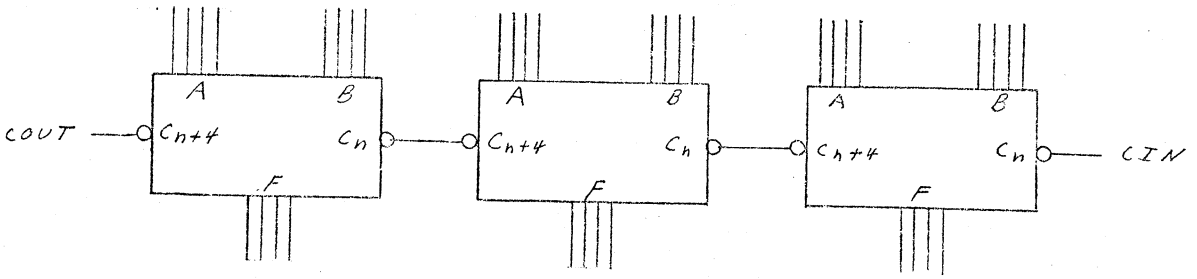
These are the only logical operations required in a PDP-8. More powerful computers might also use $A \oplus B$. The 74181 has four circuits identical to the last one to operate on each bit of the inputs.

We have said nothing of arithmetic operations in the 74181. We shall be interested in only one of the 16 possible, arithmetic addition.

A full treatment involves the theory of carry look ahead addition and is referred to Appendix A1. We ask you to accept for the moment that you can make the 74181 add by setting M = L, S3 = H, S2 = L, S1 = L, S0 = H. When you do so you force the 74181 to act as four full adders connected as follows:



To add more than four bits 74181's can be hooked together in exactly the same fashion as one bit adders.



This is an example of a 12 bit adder and is identical to the arithmetic unit of your lab computer. The control inputs M, S3, S2, S1, S0 are not shown in the above diagram but are nonetheless implied. In fact all three 74181's have their control inputs chained together so they will all be doing the same logical or arithmetic operation at any given time.

The complete table of operations is shown below. The combinations that are used in your lab computer are shown by Ⓕ.

				Logical Operations	Arithmetic Operations
S3	S2	S1	S0	M = H	M = L
L	L	L	L	(F) = $\overline{A}$	(F) = A
L	L	L	H	F = $\overline{A + B}$	F = A + B
L	L	H	L	F = $\overline{A}B$	F = A + $\overline{B}$
L	L	H	H	F = 0	F = minus 1 (2's complement)
L	H	L	L	F = $\overline{AB}$	F = A plus A$\overline{B}$
L	H	L	H	F = $\overline{B}$	F = (A + B) plus A$\overline{B}$
L	H	H	L	F = A $\oplus$ B	F = A minus B minus 1
L	H	H	H	F = A$\overline{B}$	F = A$\overline{B}$ minus 1
H	L	L	L	F = $\overline{A}$ + B	F = A plus AB
H	L	L	H	F = $\overline{A \oplus B}$	(F) = A plus B
H	L	H	L	F = B	F = (A + $\overline{B}$) plus AB
H	L	H	H	(F) = AB	F = AB minus 1
H	H	L	L	F = 1	F = A plus A
H	H	L	H	F = A + $\overline{B}$	F = (A + B) plus A
H	H	H	L	(F) = A + B	F = (A + $\overline{B}$) plus A
H	H	H	H	F = A	F = A minus 1

The results tabulated in the arithmetic column assume there is no carry into the low order bit (rightmost C_{in} in the above figure). If there is a carry in simply add 1 to the results shown in the arithmetic column.

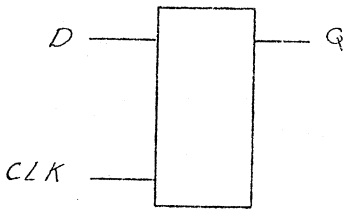
Note that most of the combinations are not used! You might be tempted to build a more specialized arithmetic unit to provide only the circled functions. Ironically such a specialized unit would be more complicated since it would have to be built with many SSI gates and would wind up taking many more packages than the 74181. Incidentally the thing we strive for in a design is to reduce package count since complexity, cost, and indirectly reliability are related to package count.

As you perhaps can sense we are getting close to a full tool kit of IC's. There is one type left to consider--flip-flops which will be discussed in the next chapter.

(9) FLIP FLOPS AND MEMORY

Up until now we have discussed strictly combinatorial logic. Remember its definition--outputs are a function of present inputs only. Some thought should convince you that that alone is not sufficient to build a computer. The essence of computing is the combination of old data to yield new data. The mere use of these terms illustrates the problem. Data must be stored so that it can be operated on; the new results must then be stored for possible later use. How can we do this? The answer is a flip-flop. Flip-flops can do many things besides store data--count for instance. Nonetheless in this chapter we shall discuss only one type of flip-flop which is used to store data in the lab computer. We will do this so we can get a sufficient set of building blocks to construct and discuss a computer. After we have covered this we will come back to flip-flops in more detail.

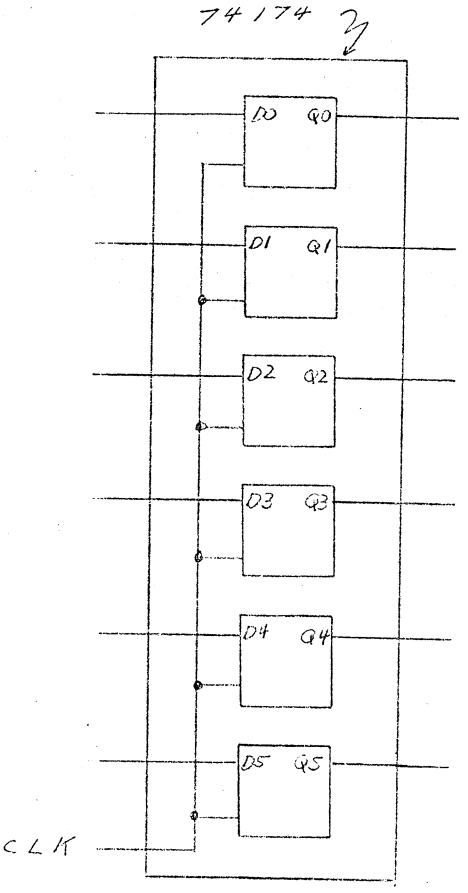
The flip-flop we use for data storage is the "D" (Delay) flip-flop. Since it is a special device its symbol will be a rectangle as shown below:



D is the data input
Q is the flip-flop output
CLK is the clock pulse used to load data into the flip-flop

The function of the device is to store a previous input until a new clock pulse arrives. The stored data appears on the output, Q. When a new clock pulse arrives the voltage on the D input at that time will be stored (entered) into the flip-flop to remain until the next clock pulse arrives. Furthermore the critical event during the clock pulse is its rising edge; this edge is the activating event which enters data into the flip-flop. If the D input is H at that time the Q output, Q, will be H a few nanoseconds later. If the D input is L the output Q will also be L a short time later.

The particular device we use for register storage is the 74174 which contains six identical D flip-flops per package with a common clock line. Since our lab computer is a 12 bit machine it will take two 74174's to hold one word of data. For efficient operation CPU's require several special dedicated words of data to be available at all times. These specialized data words are stored in flip-flop



registers. A register is nothing more than a collection of flip-flops dedicated to storing a particular piece of data. The accumulator is the most familiar example, several others will be defined in the next chapter. The length, or number of bits per register, is determined by the computer architecture. Common sizes range from 8-60 bits.

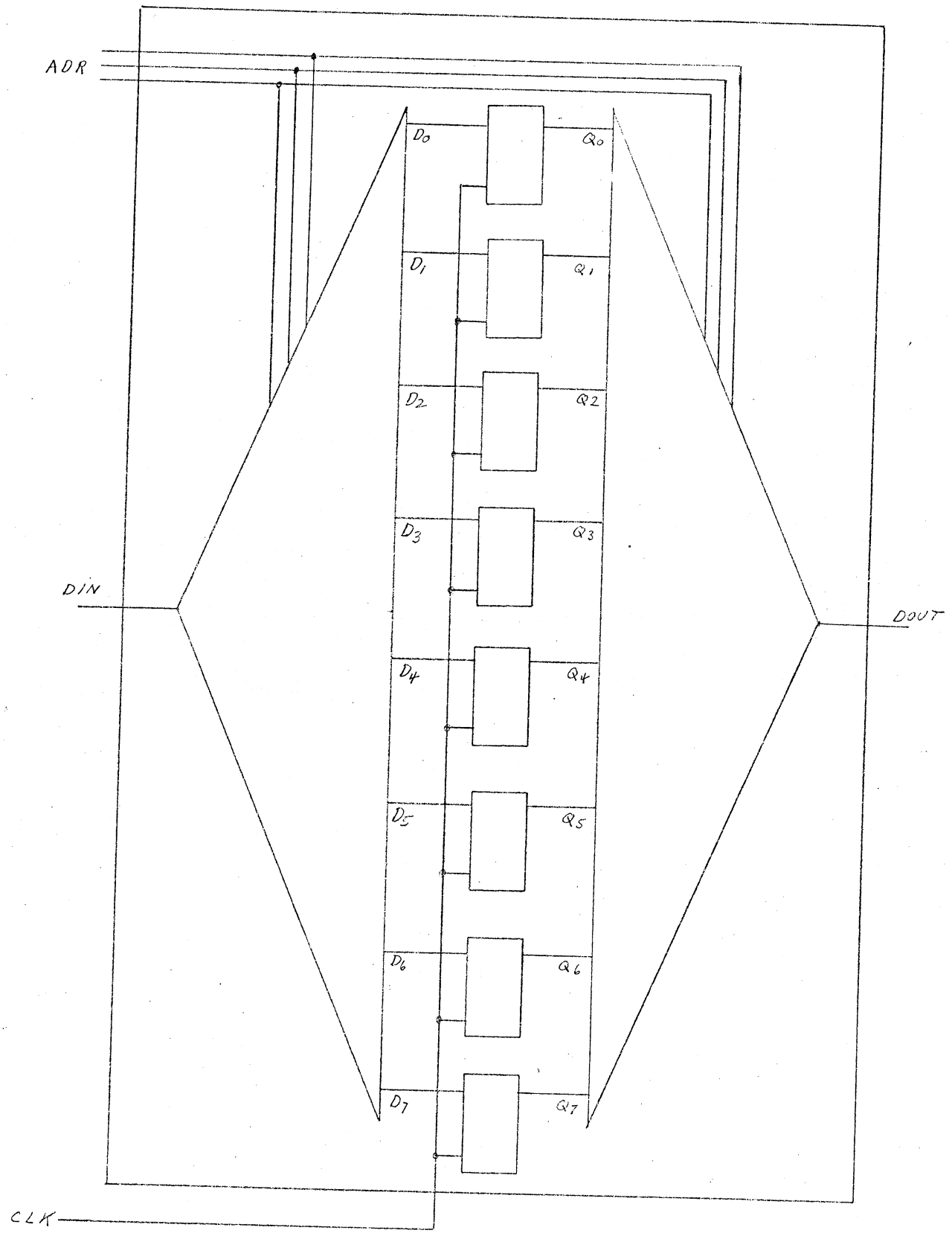
Large computers will have several million bits of main storage. Even minicomputers will have at least 5×10^4 bits. It is obviously impossible to use devices like the 74174 which provide only six bits/package. For the minicomputer example above, this would require 8000 IC's. Clearly some more efficient way must be devised for storing large amounts of data.

There are many different technologies available which can be used for this purpose. The prevalent ones are "core" and "semiconductor" memory. Since it is virtually certain that semiconductor memory will become dominant we will discuss it only.

Semiconductor memory is composed of D type flip-flops. What happens as we put more of them into a single package, such as the 74174? For each new D flip-flop two new pins (one D input, one Q output) must be put on the package. Clearly a package with 100,000 pins would be a mess (the largest common IC packages contain 40 pins).

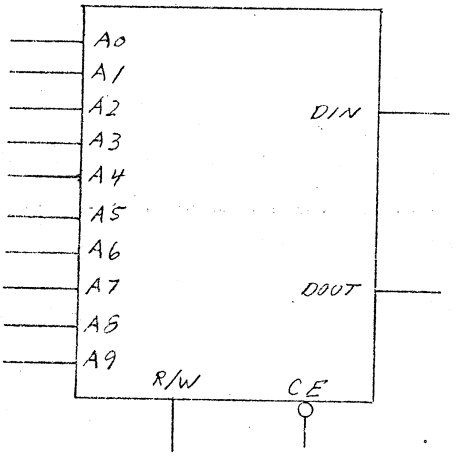
What we need is some way to access several thousand flip-flops with a small number of pins. This is done by using the concept of an address which selects only one flip-flop from the multitude inside the package. Consider a package with eight flip-flops. If we arranged it like the 74174 this would require a minimum of 19 pins (2 power, 1 clock, 8 D, 8 Q). If we relax the requirement of accessing all eight flip-flops simultaneously and are content to access only one at a time the pin count would be 8 (2 power, 1 clock, 3 bits to select which flip-flop, 1D, 1Q). This could be implemented as shown on figure 9.1 using a data distributor (demultiplexor) and a multiplexor.

FIG 9.1



The beauty of this concept is that it is readily extendable. For example a 16 bit memory could be constructed with only one more pin (4 address bits instead of 3). In fact this is exactly what is done with semiconductor memories.

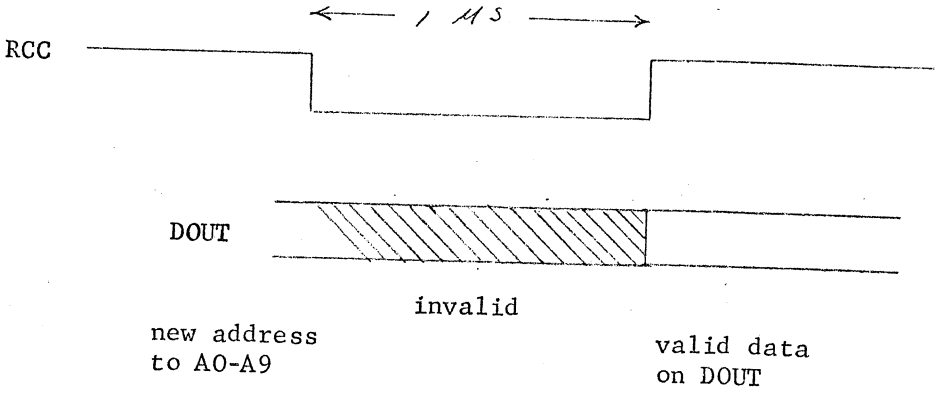
The extended memory in your lab computer is very similar to figure 9.1 except that it contains 1024 bits (flip-flops) of storage. Another difference is the timing pulses required to write a bit of data into it. The device is the 2102 RAM (Random Access Memory). Its symbol is shown below:



A0 - A9 are the 10 address lines required to select a given flip-flop ($2^{10} = 1024$).

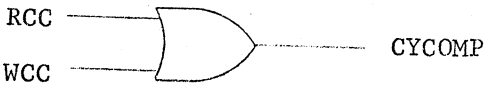
CE (Chip Enable) is a line that acts like the enable (strobe) line of the MUX's discussed in Chapter 6. When CE = H none of the internal flip-flops are connected to DOUT, and the R/W line as well as the DIN line is ignored. CE is very useful in large memories where it is used to select a small subset of the complete memory. We will treat CE again when we discuss assembling large memories from 2102's.

R/W is a line used to set the read or write mode of the memory. In your lab computer we always set R/W = H which sets the 2102 to the read mode unless we wish to do a write operation. For a write, R/W = L only for the duration of the write cycle; at the end we always switch back to read. Unfortunately when a new address is presented to A0 - A9 the new flip-flop output does not immediately appear on DOUT. The delay is called the access time and is about 1 microsecond. We must have a way of telling the computer that the memory is stable so it can trust the DOUT line. When the CPU wants to read a new word of data it must load the new address into a register which continuously presents this address to memory. This register is the MA (Memory Address) register and is composed of two 74174 hex D flip-flop IC's. As soon as a load signal (clock) is issued to the MA register, a 1 microsecond timer is started. The output of this timer will be L for 1 microsecond after which it will go H. This timer output is called RCC (Read Cycle Complete). This is shown on the timing chart below:



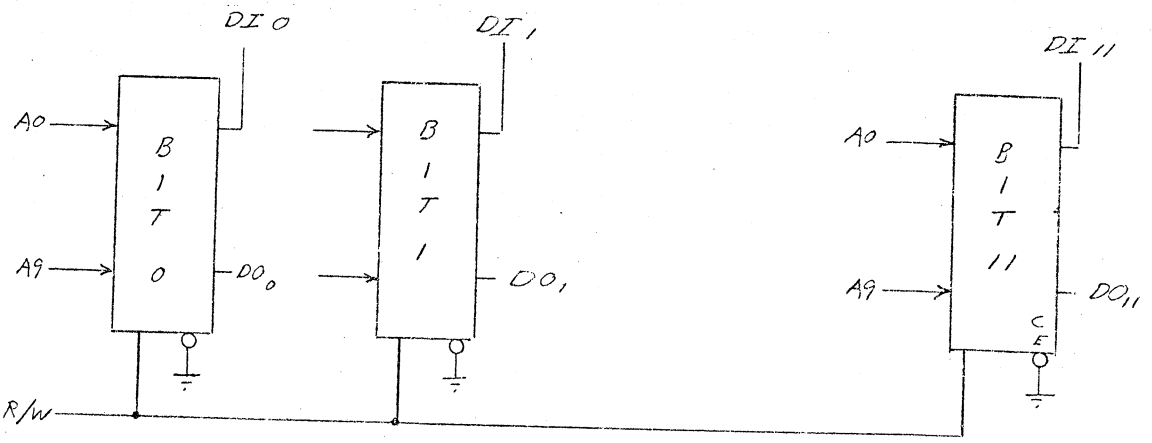
Write cycle timings are more complex. The data on the DIN line will be entered into the flip-flop addressed by A0 - A9 on the rising edge of R/W.

An analogous signal (WCC, Write Cycle Complete) exists for the duration of a write cycle. WCC does not necessarily equal RCC. These two signals are combined to form CYCOMP (CYcle COMPlete).



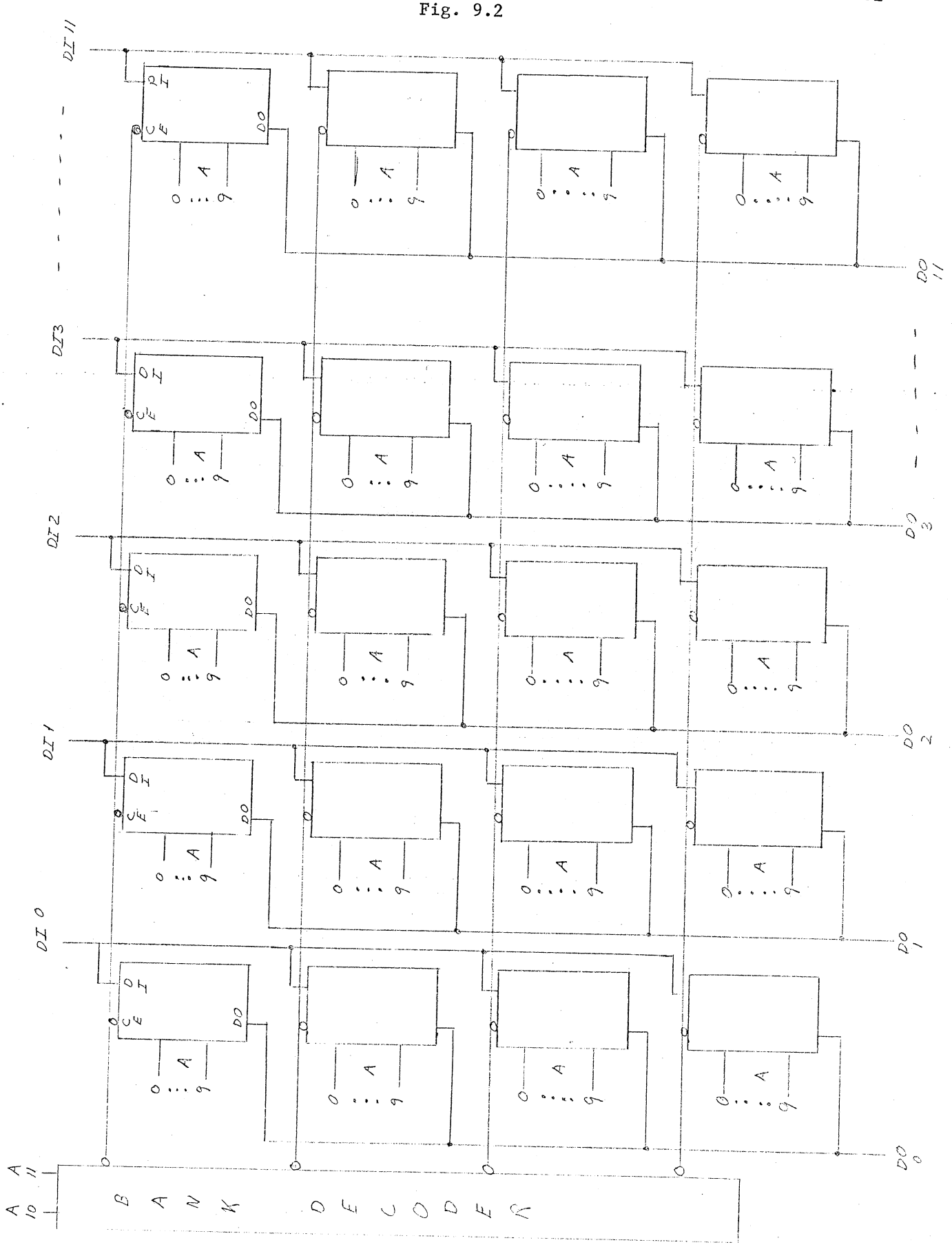
which is then the memory busy signal.

Next we need to discuss how such chips are assembled into a complete memory. Figure 9.1 would be defined as an 8 word by 1 bit memory. The 2102 is a 1024 word by 1 bit memory. To form a 1024 word by 12 bit memory 12 identical memory chips are wired "side by side" one chip per bit. If all chips are fed the same address, corresponding flip-flops in each chip will be accessed at the same time.

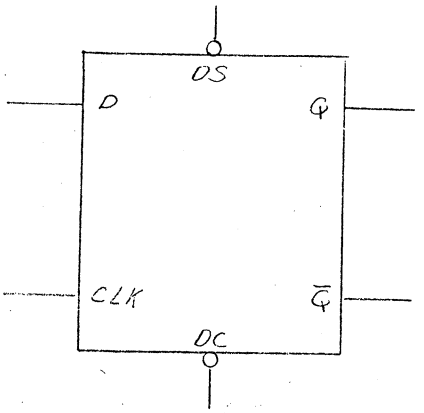


We can construct still larger memories in 1k increments (banks). To do this we need some way to disable all banks except the one that contains the address of interest. The bank enable must be derived from the additional address bits beyond the 10 required by each memory IC. A 4k x 12 memory is shown in figure 9.2. Note that the R/W lines have not been shown for drafting convenience. They are all daisy chained together so that all chips are reading or writing at the same time.

Fig. 9.2



Let us return to a further discussion of flip-flops. We have already introduced the type D flip-flop but have not talked about direct clear (preclear) or direct set (preset). The complete symbol for a common D flip-flop (½ of a 7474) is shown below:

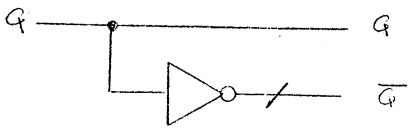


DS and DC are active <u>independent</u> of the clock. This is implied in the terms <u>direct</u> set and <u>direct</u> clear. Asynchronous clear and set is another way of saying the same thing. Whenever DS is L, Q will be H. Whenever DC is L, $\overline{Q}$ will be H. The easy way to remember this is whenever DS is active the output closest to it, Q, becomes active. Whenever DC is active the output closest to it ($\overline{Q}$) will become active. The definition of active is shown on the logic diagram by the presence or absence of small circles (DS, DC are active L; Q, $\overline{Q}$ are active H).

The ability to direct set or clear flip-flops is useful in initializing a machine. For example the CLR push-button is used to direct clear critical flip-flops in your lab computer so the machine assumes a well defined state which can later be left by the action of the CONTinue switch. The use of asynchronous sets and clears should be limited to such initializing functions whenever possible since they are not synchronized to a system clock. The importance of this is discussed at the end of the chapter.

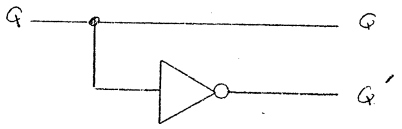
We also need to discuss the $\overline{Q}$ terminal. Because of the internal construction of flip-flops the inverse of Q is always present and can be connected to a pin "for free". We can look at this two ways:

a)



$\overline{Q}$ will be H only when Q = L

b)

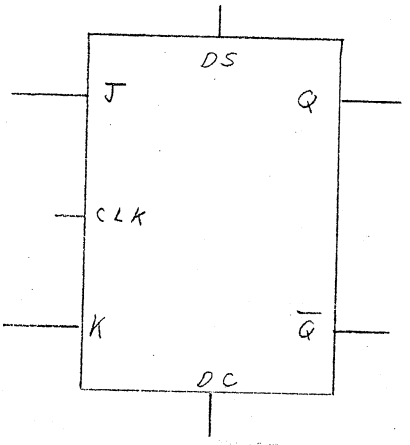


Here we have Q represented by both a high and a low polarity. Rather than label the lower output $\overline{Q}$ (which is universally done) we have labeled it Q prime to emphasize that it is simply the other voltage representation of Q.

The most common way of looking at flip-flops is the first. In any event the designer should be aware that either polarity is available which can save an inverter.

The JK flip-flop

This flip-flop is more versatile than the type D; its only drawback is it has two control terminals (J,K) as compared with one for the type D whose only control is the D line itself. The symbol for the device is shown below:



The Q, $\overline{Q}$ outputs are common to all flip-flops including the JK. (Occasionally the $\overline{Q}$ will be eliminated if there is a shortage of pins on the package, i.e., 74174).

The DS, DC lines have the same function as on the type D.

The flip-flop will make a decision every clock edge. What action is taken depends on J, K at the clock tick. We define the state of the flip-flop after n clock ticks to be Q_n. Q_{n+1} is the state of the flip-flop after one more clock tick. The truth table for the JK is given below:

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q}_n$

Let us discuss this truth table a line at a time.

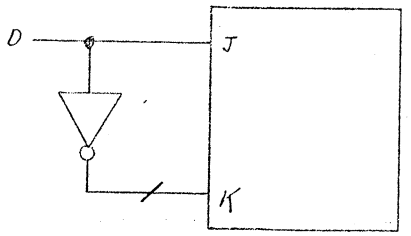
J = 0, K = 0: This is a very useful mode because it <u>stores</u> the previous state of the flip-flop even though clock pulses are continuously presented to it. This should be contrasted to the type D which will always load what is on the D line at every clock tick.

J = 0, K = 1: Clears the flip-flop <u>at</u> the clock tick. In other words it is a <u>synchronous</u> clear.

J = 1, K = 0: Sets the flip-flop synchronously.

J = 1, K = 1: <u>Toggles</u> the flip-flop <u>at</u> the clock tick.

This mode is very useful in designing counters. To demonstrate the versatility of the JK we will show how it can be converted to a type D:



if D = 1	J = 1, K = 0	and $Q_{n+1} = 1$
if D = 0	J = 0, K = 1	and $Q_{n+1} = 0$

In other words Q will always assume the value on the D line at the clock tick.

There are other types of flip-flops but they are seldom used in actual hardware design. Therefore, we will leave their treatment to the reference texts.

There is one important parameter that affects a hardware design a great deal and that is the type of clock that activates the flip-flop. By far the nicest types to use are activated by a clock <u>edge</u>. An edge is simply the voltage rise that occurs when a pulse goes from L to H. This voltage transition is often represented by an up arrow: ↑. The fall of voltage at the trailing edge of a pulse is represented by a down arrow: ↓. Flip-flops activated by either rising or falling edges are called EDGE TRIGGERED. A complete discussion of the advantages of edge triggering is given in your <u>text.</u> Suffice it to say that it allows the maximum time for signals to stabilize before they are used to make branch decisions.

(10) PUTTING IT ALL TOGETHER

We have covered all of the components required in building a computer. How do we put it all together and come up with one? This brings us to the subject of computer architecture. Like any field of engineering design, its essence is the compromise of cost, performance, and esthetics. We will return to this subject at the end of the course when we have gained more perspective. For now we will describe the design process for a simple computer. The major steps of this process are outlined below:

1) Choose the command set.

This is the most important step of a new computer design. Unfortunately, it traditionally has been the most neglected. If designed by programmers the command set tends to be overcomplicated and difficult to implement in hardware. When designed by pure engineers the command set tends to be easy to implement but difficult to program with. Some of the most successful commercial machines have abominable command sets. Some of the points to consider are:

a) Does the command set FORCE good habits on the programmer? It is becoming increasingly clear that the real cost of computing lies in the construction of error free programs. The cost of computer hardware is inconsequential. Anything that helps a programmer comprehend and simplify complex programs will lead to overall lower costs.

b) Is the command set simple? A surprisingly small set of well-chosen commands will be adequate. Commercial machines range from about 20 to 500 unique commands. 500 commands is far too many for even competent programmers to keep at their fingertips.

c) Does the command set force the programmer to build clean subroutine linkages? Subroutines are the most powerful tool available for understanding large complex programs. The programmer should not have to go through an elaborate process to save registers, acquire arguments, etc. upon entering a subroutine. A counter example is a large number of registers which can be used to hold arguments upon entry to, and results upon exit from, a subroutine. The seductive argument is made that this speeds up the machine. This is true but it immensely increases the chance for program bugs by making it too easy for the programmer to leave critical items in registers which can later be destroyed by subroutines. Some of these bugs can be almost impossible to find. The programmer must be saved from himself!

d) Does the command set provide for easy setup and control of loops? If you ever design a machine from scratch you should read and appreciate the concepts of structured programming.

e) Should the command set include facilities for stacks, indexes, and indirection? These decisions can be made only in the context of the machines intended use. Most machines contain facilities for indirection and indexing. Only a few contain stacks even though they make program compilation much easier.

f) Does the command set provide for easy input output?

In summary command sets should be chosen by people who are familiar with modern programming and also computer architecture. A poor second is the team approach where the team is composed of expert programmers and computer architects.

2) A set of registers must be chosen that is sufficient to implement the command set. Some of these registers are standard and exist in nearly every design. The standard ones are discussed first.

a) The program counter, PC. Let us review how a computer executes a program. Suppose we have the following simple program which adds two numbers and leaves the sum in the accumulator.

address	command	operand address	comment
START	LDA	A	Content of a memory address containing A is loaded into the accumulator.
	ADD	B	Content of a memory address containing B is added to the accumulator; the sum remains in the AC (accumulator).
	HLT		Stop the machine.

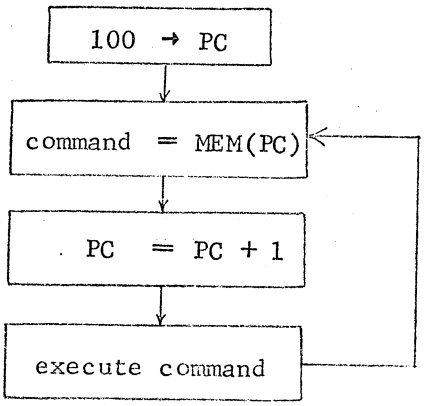
Memory contains two pieces of data A, B and in some other location three commands, LDA, ADD, HLT. This program is written in symbolic form which is done for the convenience of the programmer. The machine understands only binary numbers, so the symbolic addresses (START, A, B) must be converted into binary numbers. This is done by a special program called an assembler. Suppose we run the above program through an assembler and it assigns the first command to memory location 100. The next two commands will then be stored in locations 101, 102. Suppose also the assembler assigns memory location 163 to contain A and location 177 to contain B.

The above program in <u>absolute</u> form would be:

command address	command	operand address
100	LDA	163
101	ADD	177
102	HLT	

Actually the above program is not in pure absolute form since the commands are still represented symbolically. Note the difference between A and the location that contains A. A is <u>NOT</u> equal to 163. A is contained <u>in</u> location 163. The LDA command will access memory address 163; take the <u>contents</u> of that memory location and load the contents into the accumulator. A is called the <u>operand</u>, 163 is the <u>operand address</u>. The ADD

command also must access memory to get an operand. The HLT command does not need an operand since its sole function is to stop the machine. To execute this program the machine must somehow be told where the commands are located. This is the function of the PC. The PC will always point to (contain the address of) the next command to be executed. Thus if we set the PC = 100 and start the machine it will access memory location 100 and execute that command, whatever it is (in this case it is an LDA 163). Then PC will be automatically incremented so PC = 101. Location 101 is now accessed and that command (ADD 177) is executed and PC is incremented. PC now = 102 and that location is accessed. The resulting command (HLT) is executed which stops the machine, the PC is automatically incremented. Thus after the machine stops PC = 103 and the sum of A + B appears in the AC. In flow chart form:

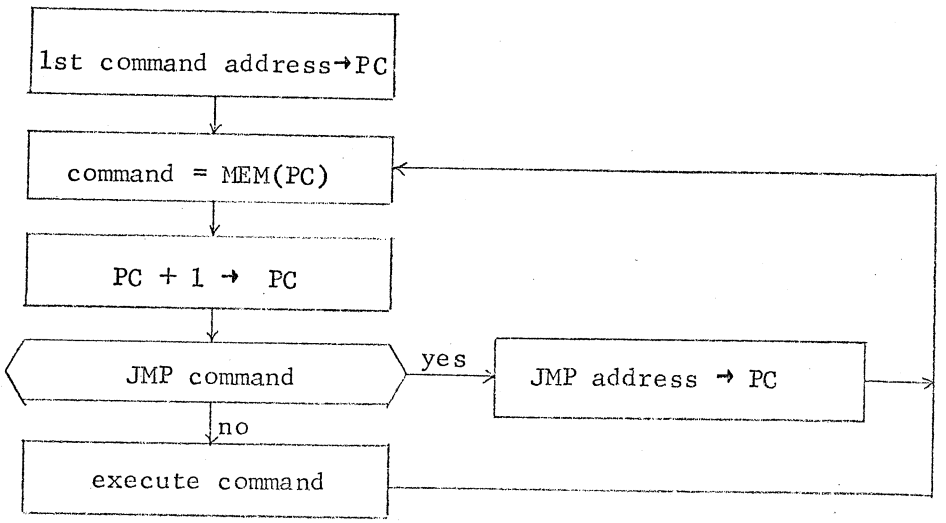


Note that the HLT command stops the machine by means independent of the PC. The PC always points to the next command to be executed.

There is one type of command which will alter the PC and that is a jump command. Consider the following two symbolic commands:

ADD	B
JMP	X

There is a fundamental difference between them. The ADD command requires an operand. The JMP command says jump to an address; no memory access is required. All the JMP command has to do is load the new address into the PC. In flow chart form:



b) The instruction register, IR. When a command is retrieved from memory it must be stored somewhere so it can control the machine during the execution of the command. That somewhere is called the IR. In the above flow chart we said:

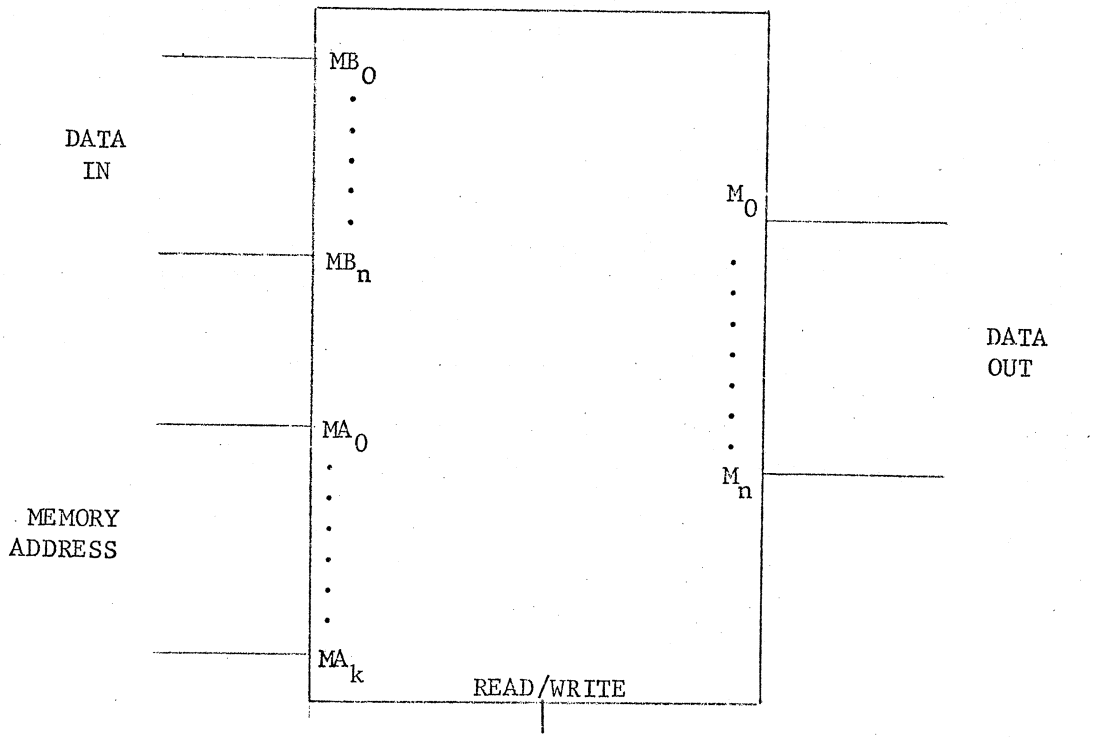
command = MEM (PC)

What actually happens is:

MEM (PC) → IR

How the IR actually controls the machine will become abundantly clear as a result of your lab work.

c) Registers associated with memory, M, MB, MA. Let us review memory. In all of our discussion we will assume semiconductor static RAM's (Random Access Memory). At the "black box" level memory consists of many storage cells (D type flip flops) which are selected by an address, and from which data can be read or written into.



The memory address is held in a register called the MA. The width of this register is determined by the number of words in the memory. A 10 bit register will address $2^{10} = 1024$ unique memory locations. Some of the common sizes are shown below:

number of bits in MA	number of memory words
4	16
8	256
10	1024
12	4096
16	65536

The data to be written into the memory is held in the MB (Memory Buffer) register. The number of bits in this register is equal to the number of bits per word in the memory. Common word sizes are 12, 16, and 32 bits per word. Other sizes have been used such as 24, 36, 48, and 60. The smaller sizes are used for economy, the larger sizes yield arithmetic accuracy.

Strictly speaking M is not an external register although it may be treated as such by the rest of the computer. The normal state of memory is to be reading. As long as this is true new data out will appear on the M lines a short time after a new address is loaded into MA. Further, data out will remain unchanged on the M lines as long as MA stays unchanged. Thus the only precaution the machine designer must exercise is to be sure that sufficient time has elapsed from a change in MA until M is used.

d) Registers associated with the arithmetic unit.

1. AC -- the accumulator. This register is used to hold one of the operands for the arithmetic unit and later to store the results from the arithmetic unit. There is always one of these registers; some machines have several.

2. MQ -- the multiplier quotient register. } not used in your lab machine

3. Index registers. } not used in your lab machine

3) A set of data paths sufficient to make all the necessary data transfers must be chosen. Another name for a data path is a BUS. A bus is defined as a separate wire for each bit of a word to be transferred. The transfer will ordinarily be between a source register and a destination register.

One extreme is a separate bus for every possible transfer path. The rationale for this approach is to provide speed since in many instances simultaneous data transfers can be made. Of course, this approach will be more complex since each bus will require a separate set of control gates.

The other extreme is to have only one bus in the machine and route all transfers over it. Even though some advertising would lead one to believe this is a new concept, it is actually very old. The virtue of this approach

is simplicity and economy. Unfortunately, it will always be slower. We will now go through this process for your laboratory computer:

1. Command Set:

Students are much more enthusiastic about a "real" computer than an original machine designed strictly for teaching purposes in spite of the fact that such a machine can illustrate all of the principles of computer design at a considerable reduction in complexity. Given this fact, the search for a suitable command set narrows to one of simplicity and cost. A near optimum is the PDP 8-I. There are only eight different commands although one of these eight has many variations, so the effective number is about 20. This small set of commands reduces the complexity of the control logic in the CPU. Most small computers have word lengths of 16 bits. The PDP 8-I has a word length of 12 bits which reduces the parts count. From a teaching standpoint this reduces not only the cost but also the number of wires the students must connect.

A very brief description of the command set is given below. A more complete description will be found in the PDP-8 Reference Manual.

AND — Forms the logical AND bit by bit between the AC and a word in memory. The result is left in the AC.

TAD — Forms the arithmetic sum of the AC and a word in memory. The result is left in the AC.

ISZ — Increment and skip if zero--increments a word in memory. If the result is 0, one command is skipped. If not 0, the next sequential command is executed. This command is used for loop control.

DCA — Store and clear accumulator. Stores the AC in a memory location. Clears the AC afterwards.

JMP — Jump. Breaks the sequential flow of a program by setting the PC to a new value.

JMS — Jump to a subroutine. Stores the return address in the first word of the subroutine and jumps to the second word of the subroutine.

IØ — Executes an input/output operation.

Each of the above commands uses an address field in the instruction. The next and last command is fundamentally different. No memory access is required so the bits of the address field can be used for other things. Some of them are listed after the command.

NMR — Non-memory reference. Bits of the address field can be used to:
Increment the AC
Complement the AC
Clear the AC
Clear the Link
Complement the Link
Shift right and left
Skip on various conditions such as AC=0, Link = 0, and minus AC.

2. Choose registers.

This command set requires the minimum number of registers since only one accumulator is required. The registers are:

AC, IR, PC, M, MA, MB.

3. Choose the data path structure.

We will choose the simplest possible structure to implement the computer, namely a single bus system. Our reasons are:

a. A single bus system is easy to understand. This makes it better for a first introduction to computer architecture.

b. Fewer wires are required which makes it easier to build.

c. A smaller amount of control logic is required.

The schematic data path structure for one of the 12 bits is shown in figure LD1.

The 8 input MUX serves to route any of the registers, including the switch register, to the A input of the ALU. The ALU can perform any of the 16 logic functions including F = A. Thus any register can be transferred to any other by a three step process:

a. Address the MUX for the desired source register.

b. Set up the ALU control so that F = A.

c. After F has stabilized, issue a load pulse to the destination register. Note that a D type flip-flop will present the old data on the Q output regardless of the data on the D input. This will continue until a new clock pulse is applied to the CLK terminal on the flip-flop. Thus a register can be <u>both</u> a source and a destination without getting confused.

Also note that two registers can not be loaded directly from the F bus. The switch register is composed of 12 switches on the front panel which can act as sources of data only. The only way to set them is manually. The other register is the M register which is the output of memory. The only way new information can be written into memory is by first loading it into MB and then issuing a write command to memory.

We see that this very simple structure is sufficient to transfer data from any source to any destination. A good portion of computer control consists of doing nothing more. Command execution however, is more complex. Suppose we wish to do an ADD instruction where we must do the following:

AC plus M → AC

Note that AC is always connected to the B input of the ALU. If the MUX is addressed to M the contents of memory will be presented to the A

input of the ALU. If we now set the control bits to make the ALU add A, B the sum will appear on F where it can be loaded back into the AC after it has stabilized.

The simple structure shown above is in fact adequate to implement the entire computer. However, we have ignored how the computer will be guided through its various steps. The control process can be broken down into two major categories:

1) Fetching an instruction from memory and preparing it for execution.
2) Executing it.

An example of an operation that must be done every time a command is executed is PC(+)1 → PC. This can easily be done with our proposed architecture as follows:

a) Address the MUX to pass the PC (octal address = 0). A few nanoseconds later the contents of the PC register will be stable on the output lines of the MUX.

b) Command the ALU to add 1 to its A input. If you look in the defining tables for the ALU (74181) you will find that:

M=L S3=L S2=L S1=L S0=L CIN=L

will do the job.

c) A few nanoseconds after these control signals are applied to the ALU, the output, PC(+)1 becomes stable and can be loaded back into the PC. Since the PC is made from D type flip-flops, its output (= PC) will ignore the input (PC(+)1) until a clock pulse is issued to the flip-flop.

d) The idea is to generate the various control signals with gates, let the signals settle down and load stable ALU output signals back into a register.

e) YOU CAN CONTROL A COMPUTER WITH THESE REGISTER TRANSFERS.

A simple abbreviation for steps a-c is given below:

MUX=PC ALU=A(+)1 PC(load) or PC(L)

Another common operation is to move the PC to the MA to read the next instruction from memory. Remember the PC points to the memory location containing the next instruction to be executed and the MA is a special register that tells memory where its next access should be. Every time we load a new value into MA the contents of that memory location will be read out a short time later (the normal state of memory is to be always reading unless commanded to write). Now the PC must be passed through the ALU unchanged so it can be loaded into the MA. If you look at the data book for the ALU there is indeed such a command:

M=H S3=H S2=H S1=H S0=H CIN=X(H or L)

In short:

MUX=PC ALU=A MA(L)

In our computer after we have read the next instruction from memory we will want to load it into the Instruction Register (IR) so it can be saved for the duration of that command. Control signals will be generated from the (IR) to guide the CPU through to a successful completion of that stored command. The IR can be loaded as follows:

MUX=M ALU=A IR(L)

Again let us emphasize that a computer is nothing but a set of registers which command sub-elements of the computer to do certain operations.

The MA tells memory where to read or write
The M gives the result of a memory read M=mem(MA)
The MB stores data to be written into memory mem(MA)=MB
The PC stores the location of the next instruction
The IR stores the current instruction during its execution
The AC is the accumulator

From the information stored in these registers we must generate control signals to the MUX, ALU, Register Load Signals, etc. to make the computer perform properly. By far the most important control signals are the ones listed in the last sentence.

To get a solid foundation let us go through the first six instructions in detail. We will assume that the fetch portion of the cycle has left the effective address in the MA and the effective operand in the MB. This is the entire purpose of the fetch cycle. The execute cycle can now be called without regard for any of the complexity or past history of processing during the fetch cycle. Not all computers treat the fetch cycle in such a uniform fashion but it is a procedure that any experienced programmer would consider very natural. All that we have done is split the execution of a single instruction into two independent parts (co-routines), the fetch cycle, and the execute cycle. By making them independent we can tackle them independently also.

1) AND X Remember that X is a memory address and the instruction commands the computer to take the contents of that address (the operand) and form the bit by bit AND with the AC. The result is to be left in the AC.

We make the standard assumption that the fetch cycle has left the effective address in the MA and the effective operand in the MB. Thus we form the bit by bit AND between MB and AC and put the result in the AC.

The entire operation can be done in one clock cycle:

MUX=MB ALU=AND AC(L)

2) TAD X The comments on 1) apply here also:

MUX=MB ALU=ADD AC(L)

3) ISZ X This command does two things:

a) Unconditionally increments the contents of memory location X.

mem(X) = mem(X) (+) 1

b) If the result in memory location X=0 then skip one instruction; otherwise execute the next sequential instruction.

Note that this instruction must do a memory write since a new value must be put back in memory. This will require the loading of MB with the incremented value and the issuance of a write command to memory.

After this is completed, we must test the result. If zero, the next instruction must be skipped. This can be done very simply by incrementing the PC in the ALU and loading the result back into the PC if mem(X) = 0. If mem(X) ≠ 0 do not load the new address into the PC; by default the old value will be left in the PC.

This instruction will take three clock cycles to complete.

CLK0) MUX=MB ALU=A (+) 1 MB(L) Get the old effective operand, increment it, store it in MB for memory write.

CLK1) Issue a write pulse to memory, wait until memory has completed the operation.

CLK2) MUX=PC ALU=A (+) 1 Increment the PC
IF MB=0 load the ALU output into the PC.

4) DCA X This instruction stores the contents of the AC into memory location X. Note that we need only the effective address for this command. We do not care what was in mem(X) since the AC will overwrite it.

CLK0) MUX=AC ALU=A MB(L) Move the AC into MB in preparation for writing.

CLK1) Issue a write pulse, wait until done.

5) JMP X Jump to location X. This is probably the simplest command since it only sets X into the PC.

CLK0) MUX=MA ALU=A PC(L)

6) JMS X This command is used to jump to a subroutine. The essence of a subroutine is the ability to return to the next location after the calling location. This return location is stored in location X. A standard jump is then made to location X (+) 1.

CLK0) MUX=PC ALU=A MB(L) Move the return address (PC) to the MB so it can be written into the first word of the subroutine. Remember that address is in MA from the fetch cycle.

CLK1) Issue a write pulse and wait for completion.

CLK2) MUX=MA ALU=A (+) 1 PC(L) Set up the jump to the second word of the subroutine.

The next step is to derive the control signals and their timings that will make the computer execute any one of these six commands. To do this we organize this information into a table with clock times labeling the rows and command types labeling the columns. The clock times will be abbreviated; thus CP2 stands for Clock Pulse 2 (or equivalently clock time 2).

	AND	TAD	ISZ	DCA	JMP	JMS
CP0	MUX=MB ALU=AND AC(L)	MUX=MB ALU=ADD AC(L)	MUX=MB ALU=A+1 MB(L)	MUX=AC ALU=A MB(L)	MUX=MA ALU=A PC(L)	MUX=PC ALU=A MB(L)
CP1			Memory Write	Memory Write		Memory Write
CP2			MUX=PC ALU=A+1 PC(L) if MB=0			MUX=MA ALU=A+1 PC(L)

The information in this table can now be converted to boolean equations. Let us consider the MUX equations in detail:

MUX = MB during: CP0 · (AND + TAD + ISZ)
MUX = AC during: CP0 · DCA
MUX = MA during: CP0 · JMP + CP2 · JMS
MUX = PC during: CP0 · JMS + CP2 · ISZ

Remember that the various MUX inputs are selected by a three bit code. We will label this three bit code as B4 B2 B1. The actual values for a given code are shown on fig. LD1 . We see that:

MUX = MB requires B4 B2 B1 = 001
MUX = AC requires B4 B2 B1 = 011
MUX = MA requires B4 B2 B1 = 010
MUX = PC requires B4 B2 B1 = 000

We can now derive boolean equations for each bit of the select code as follows:

B4 = 0
B2 = AC + MA = CP0 · (DCA + JMP) + CP2 · JMS
B1 = MB + AC = CP0 · (AND + TAD + ISZ + DCA)

It is now a simple matter to build gate circuits that will generate correct signals for B4, B2, B1 which in turn will properly control the MUX for the execution of the above commands. Of course, we will also need a source for CP0, 1, 2, but this is a simple problem which we will consider later.

A similar analysis will give the ALU control signals as shown below:

ALU = AND during CPO · AND
ALU = ADD during CPO · TAD
ALU = A + 1 during CPO · ISZ + CP2 · (ISZ + JMS)
ALU = A during CPO · (DCA + JMP + JMS)

By reference to the data sheets for the ALU (74181) we find the following control codes for the various operations:

	M	S3	S2	S1	S0	CIN
ALU = AND	1	1	0	1	1	X
ALU = ADD	0	1	0	0	1	0
ALU = A + 1	0	0	0	0	0	1
ALU = A	1	1	1	1	1	X

The polarity conventions are:

M = 1 is H S3 .. S0 = 1 is H CIN = 1 is L

Boolean equations for the six ALU control signals can now be derived.

M = (AND + A) ALU = CPO · (AND + DCA + JMP + JMS)
S3 = (AND + TAD + A) = M + ADD = M + CPO · TAD
S2 = A = CPO · (DCA + JMP + JMS)
S1 = (AND + A) = M
S0 = (AND + TAD + A) = S3
CIN = $\overline{S_3}$ (Since we can assign X to be either 0 or 1)

Again we see that the resulting equations are very simple and that the process of deriving them is also simple. All that is required is the complete table of operations required at each clock pulse. This table is shown in fig. LD2 and LD3. Note that six new operations are included in the blueprint that do not correspond to the command set of the PDP 8-I. These are the manual load commands which load the switch register into the corresponding register. These are extra commands added for the convenience of the programmer. They require so little additional hardware that they are well worth it.

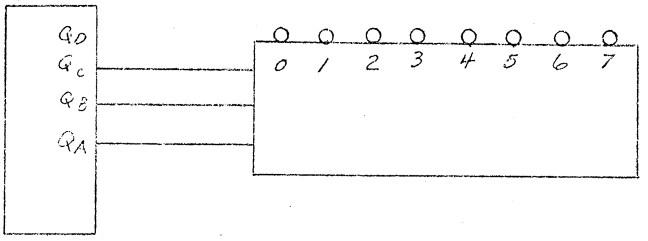
The reader should go through this chart and derive the equations from it to verify the logic equations for the MUX, ALU, and Load signals shown in figure LD6.

Let us return to the clock pulses. As you can see from the logic equations we must have a signal, for example CP1, which is true only during time slot 1. Similar signals must be available to tell you when you are in any given time slot.

Such signals can be supplied by a decoder driven from a counter. The particular counter that we use is the 74163; at this time you should review it in the TI manual. From the "black box" standpoint it has four outputs Q_D, Q_C, Q_B, Q_A, and a clock input. Suppose the initial state of the outputs are all L, i.e., (0000). The arrival of a clock edge (negative to positive transition) will cause the device to COUNT or advance to the next state with Q_A high and Q_D - Q_B low, i.e., 0001. The arrival of another clock pulse

will advance the counter to state 2, i.e., 0010. Each new clock pulse will move the counter into the next state. In each case there is a direct correlation with the sequence of binary numbers from 0-15. The 16th clock pulse will return the counter to 0000 where it is ready to start the counting process again.

Now all we need is some device to tell us what state the counter is in. That IC is of course a decoder. Remember a decoder accepts a binary number and activates a unique terminal corresponding to that input.



Thus if the counter is cleared, i.e., 0000, then only output 0 of the decoder will be L. If the counter is in state 0011 then only output 3 of the decoder will be L. Since we need only eight different time slots to control the execute cycle we use only the three low order bits of the counter to drive the decoder. The decoder actually used (7442) has 10 output terminals (0-9) and will accept 4 bit binary inputs. 0000 through 1001 activate just one output terminal. Inputs 1010 through 1111 are illegal inputs and turn off (inhibit) <u>all</u> outputs. Since we are interested in decoding only eight states we may use the fourth input (D) as in inhibit. If D = 0 we are considering inputs 0000 = 0111 and everything behaves normally. If D = 1 we are considering inputs 1000 = 1111. If we are using only outputs 0 - 7 in our circuits they are switched off (H) whenever D = 1. Thus D can be used as an inhibit signal for outputs 0-7. This is useful since we want to feed signals CP0 - CP7 to gates <u>only</u> during the execute cycle. CP0 - CP7 must be turned off during the fetch cycle to keep the machine from getting confused.

Fig. LD9 shows the actual circuitry for generating CP0 - CP7. B17 is the decoder; note that inverters are used on the outputs so that CP0 - CP7 are available in both high and low polarities. The enable line D, pin 12, is driven from the EXEC flip-flop which turns on only during the execute cycle (EXEC is turned on when the FETCH cycle is exited). Thus the decoder is enabled during the execute cycle <u>only</u>.

B15 is the execute counter. Only the three low order bits of the counter output (Q_C, Q_B, Q_A) drive the decoder. When the EXEC flip-flop is reset a solid clear is applied to B15 continuously holding it in the 0000 state (the $\overline{Q}$ output of EXEC is simultaneously disabling the decoder B17). As the fetch cycle is exited the EXEC flip-flop becomes set (Q = H) which then enables the counter B15 to count through the eight states of the execute cycle and also enables the CP0 - CP7 decoder so these signals become available to the gates which are generating the MUX, ALU, and Load control signals. At the end of CP7 the FETCH flip-flop is set (GTF, Go To Fetch, pin 13 of B47) and the EXEC flip-flop simultaneously reset (pin 3, B46).

Thus the execute cycle always starts at CP0 and runs through CP7. This is stupid since most instructions do not require all eight time slots for their execution and the idle time is simply wasted. It is a small exercise for the reader to generate a new GTF signal which will terminate the execute cycle as soon as possible. The simple design was chosen to keep the parts count to an absolute minimum.

At this point let us review what you should have learned from this chapter:

1) Buss structure of the lab kit;
2) How that structure can be used to move data within the machine;
3) How the machine can be controlled to accomplish the data moves of #2;
4) How boolean equations can be derived to accomplish the control of #3;
5) An introduction to the concept of a time sequence of states used to implement the boolean equations of #4. This topic is so important that we will devote the next chapter to it.

(11) HARDWARE FLOW CHARTS

The essence of digital design is the construction of logic circuits that will implement a flow chart. The reader is most likely familiar with writing a program to implement a flow chart. It sometimes is foreign for a person with such a background to think of hardware doing the same thing. Both techniques should be in the tool kit of any computer architect since each technique has unique advantages.

A) Hardware flow chart implementation.
   1) Advantages
      a) Speed -- properly designed hardware will always execute a flow chart faster than a programmed computer.
      b) Sometimes it is the only way a flow chart can be implemented. For example, although a computer can be programmed to execute the flow chart, how can the internal flow chart of the computer itself be implemented? Clearly this must be done with hardware.
      c) Non-Volatility -- Since the hardware is made up of copper wires, gates, etc., it does not disappear when the power is turned off. As soon as power is reapplied the hardware is ready to go. If the implementation is by means of a stored program it is sometimes necessary to reload the program after a power outage.

   2) Disadvantages
      a) Maintenance -- There are many more programmers in the world than logic designers. If a bug develops it may be easier to find and repair in software.
      b) Inflexibility -- Hardware is difficult to change. It usually means ripping out wires, changing gate types, etc. A program change by contrast is quite easy.

B) Flowcharts implemented by programs (software)
   1) Advantages
      a) Ease of implementation -- If the flowchart is at all complex, and especially if it involves numerical computation, a software implementation will usually be much cheaper.
      b) Ease of change -- Often the flow chart one is implementing will change during the course of a project, usually in an unforeseen way. Such changes can be much more easily handled in software.
      c) Cost -- For one or two copies of a device it can be much cheaper to buy a micro or minicomputer and program the solution. A hardware solution does not pay off until many copies (often hundreds) have to be produced.

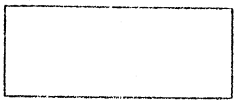
   2) Disadvantages
      a) Interfacing -- We have been very glib when we talk about implementing a flow chart in software since we are considering problems where a physical device must be controlled. An example would be a minicomputer which is controlling a telescope. Somehow or other the telescope must be electrically connected (interfaced) to the computer. This is always a hardware interface which may be more or less difficult.

b) Sometimes the flowchart you are trying to implement may involve an algorithmic process that can be done by a program but too slowly to meet external constraints. An example might be conversion from polar to rectangular coordinates on a rotating radar antenna. Depending on the speed of rotation and conversion accuracy a computer program simply may not be able to keep up. In such a case a hardware solution may be unavoidable.

The point of the preceding discussion is that both methods are useful for solving real problems. Any practicing engineer or computer scientist must be open-minded enough to choose the optimum solution (which may be a combination).

Naturally since we are building a computer we will restrict our discussion to hardware implementations of flowcharts. For obvious reasons we are considering only the simplest systems, which happen to be synchronous sequencers. Synchronous means that the transition from one point to another on the flow chart happens only on the tick of a system wide clock. The basic reason for this simplicity is that every decision can be made at a predictable time. Further, these times can be chosen so all transient behavior in the system has died down and decisions are based on stable signals. The resulting advantages are so strong that the vast majority of digital systems designed are synchronous. Nonetheless, asynchronous systems are important for the professional designer and should be studied in some of the reference texts for those wishing to go deeper into the subject.

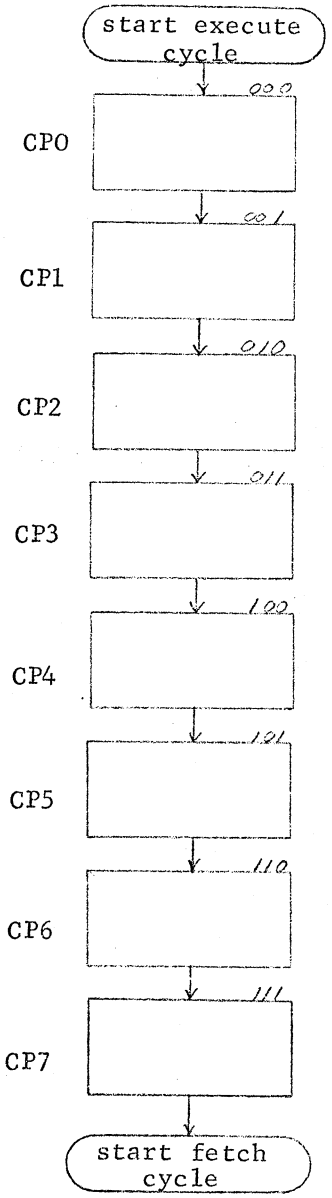
The symbols we use to construct a flow chart are shown below:

STATE 

This symbol represents a unit of time. Time can be visualized as starting at the top of the rectangle, progressing for one clock cycle to the bottom of the rectangle which coincides with a clock tick. At that time a jump to the top of a new box is made. Such a box is called a STATE.

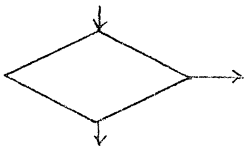
Each box on a flow chart is unique by virtue of its position and must therefore have a unique identifier. This identifier is made up of boolean variables, usually outputs of flip-flops. Thus a state can be identified either by the flip-flop outputs or symbolically. By convention we write a symbolic name for a state to the left of the box, the boolean variables on the upper right.

We have already discussed the flow chart for the execute cycle. Let us show it graphically:

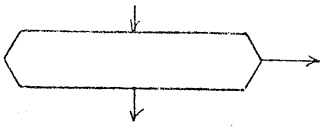


We have chosen state names CP0 through CP7 but there is nothing sacred about choosing them to correspond to the state variables 000 through 111. Other state names such as A - H could have been chosen. Clock Pulse 0 - 7 was chosen because it is descriptive. The state is identified by three flip flop outputs, Q_C, Q_B, Q_A, from the 74163 counter.

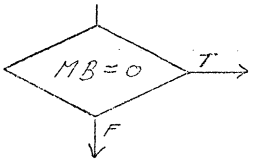
The above flowchart is useful but limited since it has no provisions for a conditional branch. The symbol for a test is shown below:

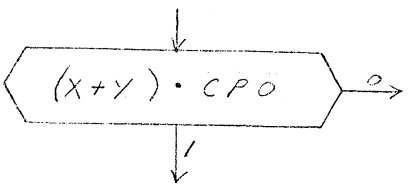


or



The quantity to be tested is written inside the box and is a voltage. Examples are:





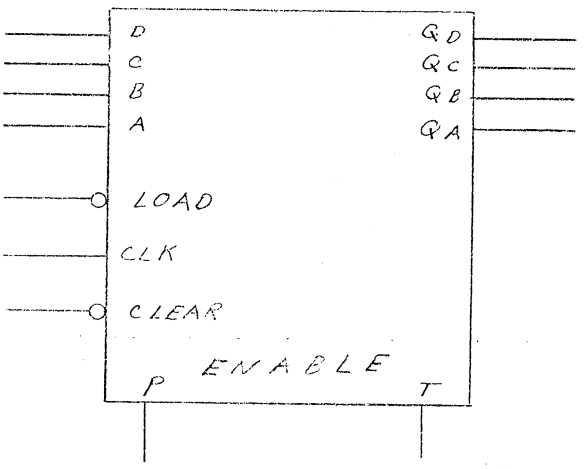
Since the quantity to be tested is a voltage which can assume only two values (H or L) the ordinary conditional branch leads to only two other states.

The actual flow chart for the execute cycle in fact has a conditional branch condition built in to handle the case of a memory write cycle. Memory timings are independent of the main system clock; therefore the memory must have a way of telling the computer that it has finished a write cycle and the computer must have a way to wait until it gets this signal from memory. If you examine the execute table you will see that ISZ, DCA, JMS, LDM, and DEP initiate a write cycle on CP1. After the write cycle is initiated, memory will independently take care of its own operation. During this time we are free to do other CPU operations provided we don't disturb MA or MB. The execute table discloses that MA and MB are not loaded by any command during CP1, CP2, or CP3. We can therefore do these CPU operations in parallel with memory operations and save time. Therefore, we do not test CYCOMP until CP3.

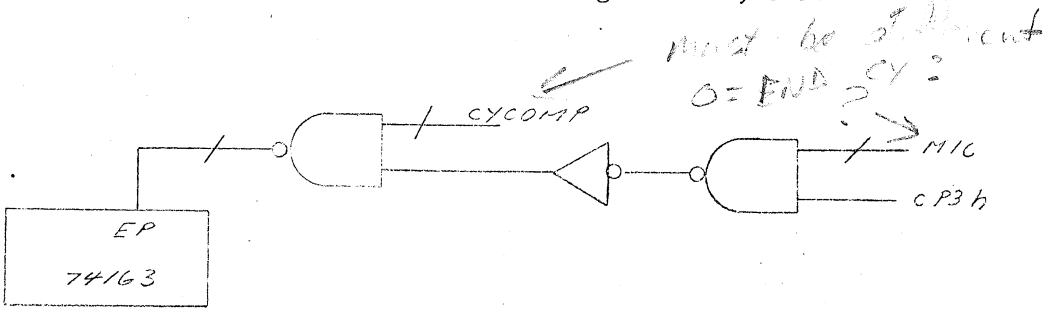
	CYCOMP is the signal that memory returns to the computer
when:	CYCOMP = 0; the memory is still busy
when:	CYCOMP = 1; the memory has completed its previous operation.

The branch test is not made until the clock ticks. The designer must make sure that CYCOMP is stable at the clock tick time or it is possible to confuse the branch test. The test for CYCOMP is handled in a very clever way by a special property of the 74163.

It is now time to consider the 163 in full detail. The student is urged to examine a manufacturers data book in addition to this description:



There are many nice properties of this IC; one of them is controlled by the LOAD terminal. If LOAD = L the IC is disabled as a counter and reconfigured as four ordinary D flip flops whose inputs are A - D and outputs are Q_A - Q_D. If LOAD = H and CLEAR = H the IC functions as a 4 bit binary counter provided both ENABLE P and ENABLE T are H. If either goes L the counter will ignore CLK pulses and stay in its present state. This is used to implement the CYCOMP test during execute, CP3.



It is possible to enter execute with a CYCOMP still pending (CYCOMP=0). The purpose of the $\overline{MIC}$ · CP3h gate is to disable the CYCOMP conditional branch test during the execution of a microinstruction. The reason for this is subtle. If the system clock is much faster then CYCOMP many branches back to state 3 will take place before CYCOMP = 1. If you look at OPG1 during CP3 you see that AC will be incremented at the beginning of CP3 and the new result loaded back into the AC at the end of CP3. Thus it is possible to increment AC many times while waiting for CYCOMP. This is prevented by gate A7 (pin 10, fig. LD9). All other commands will have a conditional branch on CYCOMP at the end of CP3. CYCOMP will normally be 1 unless the memory has been issued a write pulse so those instructions that do not issue a write during CP1 will always take the branch to CP4. Those that do issue a write will branch back to the CP3 state until CYCOMP = 1 and then go to state CP4.

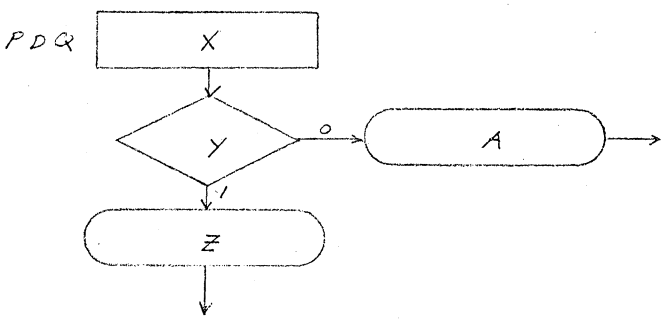
The previous technique will always loop back to the parent state on the failure of the conditional branch test since the counter will be held in its old state. A slightly more complex flow chart will be required to describe the fetch sequence since we will need to branch to any state on the 0 branch of the conditional test.

To describe this flow chart we need to introduce the concept of a conditional output. Its symbol is:

If a state is to output a value regardless of any test conditions that value should be written inside the state box:

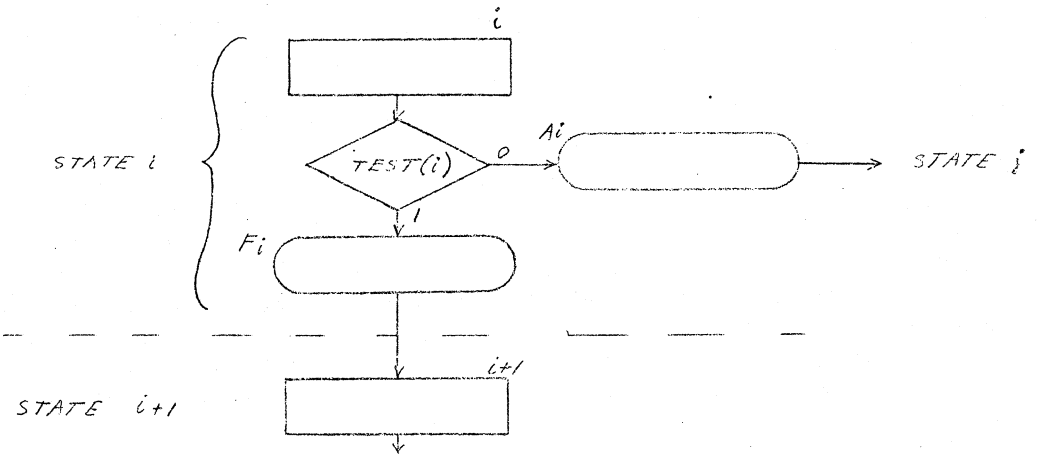
indicates X will = 1 during state PDQ

Sometimes however, we will want signals to be true only during a given state AND a given test variable equal 1 (or 0). We show that in the following fashion:



X will be true for one clock time during state PDQ
A will be true for one clock time during state PDQ
only if Y = 0
Z will be true for one clock time during state PDQ
only if Y = 1

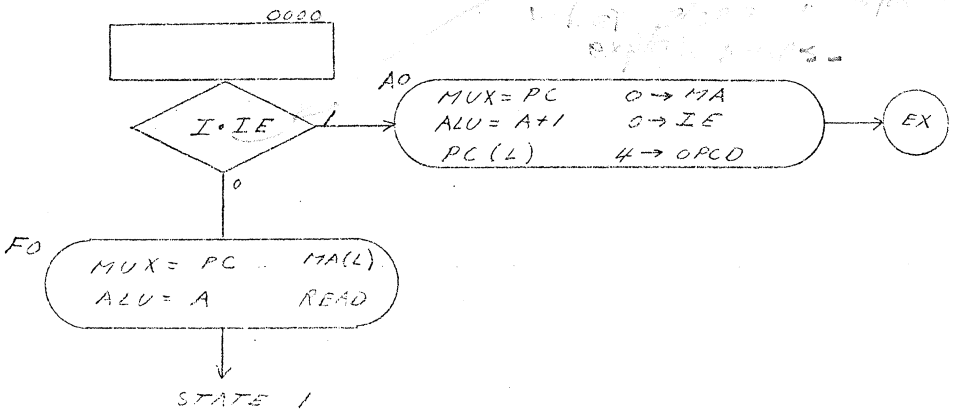
The circuit we shall describe has the following basic structure for each state:



Here we have labeled the conditional outputs as Fi (Fetch i) and Ai (Alternate i). There is nothing sacred about these names and any name that has meaning with respect to the flow chart being implemented may be used by the designer. Note that Fi and Ai are not states; they are conditional outputs associated with state i.

We will now discuss the fetch flow chart state by state:

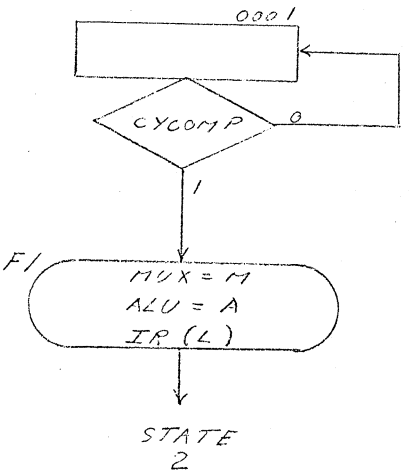
STATE 0



A0: is the external interrupt sent to the CPU by peripheral devices. The CPU has an internal flip flop that can be set by a special command, ION; or reset by another special command, IOF. The CPU will ignore external interrupts when IE = 0 and respond when IE is 1. Response to an interrupt is the execution of a JMS to location 0. At this point reread the description of the JMS command. The net effect will be to store the updated PC in location 0 followed by a JMP to location 1. The execute cycle expects to find the effective address (0) in the MA. A JMS (octal 4) must replace the old Opcode. This is done by the triple two input multiplexors (B13) shown on figure LD10. Conditional output A0 resets the JK flip flop B47, the Q output goes L which is the proper polarity to switch the MUX inputs to (Vcc, gnd, gnd) = (100) = octal 4 to the instruction decoder B14. CP7h sets the flip flop at the end of the execute cycle so that the MUX will pass the opcode (IR0, IR1, IR2) normally for following instructions. Another function of A0 is to turn off the interrupt system by resetting the interrupt enable flip flop. The reason for this is the interrupt subroutine must be free to process the interrupt without being subject to interrupt.

F0: The absence of an interrupt is by far the most common exit taken from state 0. The next instruction to be executed must be read from memory; the PC points to that memory location. Therefore, the MA is loaded from the PC and memory read cycle is started by issuing a read (R) pulse.

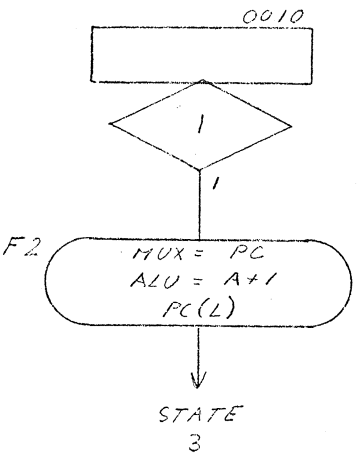
STATE 1



A1: Our standard hardware actually provides an A1 output; but since it is not needed the pin on the generating decoder remains unconnected and the A1 conditional output is not shown on the flow chart.

F1: After CYCOMP = 1 all of the signals shown in conditional output F1 will become active. These signals route the new instruction just read from memory to the instruction register.

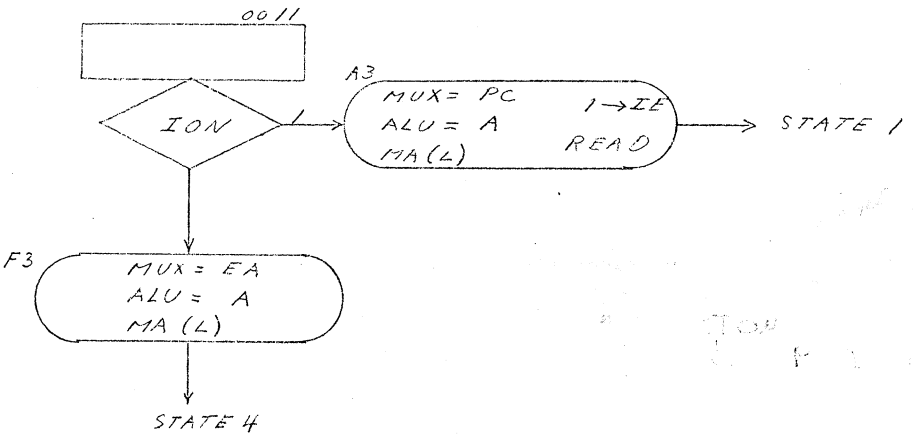
STATE 2



A2: Ignored.

F2: We wish to always increment the PC so we do a conditional branch on an always true variable to in effect generate an unconditional branch. The PC now points to the next instruction so the next time we reach F0, PC will point to the proper instruction.

STATE 3



STATE 3 is used for processing the special instruction ION. This special command is used to turn the interrupt system (IE flip flop) on. In addition it is set up so the next instruction will always be executed even if an interrupt is pending. The reason for this is the interrupt subroutine must disarm the interrupt system so it can process an interrupt without itself responding to following interrupts. The next to last instruction of the interrupt subroutine must be an ION to rearm the interrupt system. The last instruction of the interrupt routine must be a return JMP to the main program. This return jump must always be executed even if a new interrupt is pending.

A3: Must rearm the interrupt system so it sets 1 → IE (A95 fig. LD21). There are many different kinds of flip-flops that could be used for IE. We have chosen one of the more complex kinds, a gated D flip flop. This

special type has so many nice properties that it is becoming available as an IC. In view of this we felt it was worthwhile for the student to construct one from individual components (A97, A95). As long as the INTLD signal is false, data from the flip-flop will be reloaded on the next clock. If INTLD goes true the old flip-flop output is blocked from the D input and a new signal A3 is loaded into IE. The beautiful thing about this circuit is that the clock can be fed to the D flip-flop always without destroying the old contents of the FF. Only when the load signal is applied does the FF accept a new value. The conditions for loading are:

$$INTLD = A0 + A3 + CPO \cdot IOF$$

Remember that state A0 must turn off the IE FF. During A0, A3 will be H which will apply an L to D (A95, pin 12). Thus A0 will set 0 → IE. The same is true for the special command IOF which disarms the interrupt system during CPO. During A3 the input to pin 5, A97 will be L, input to D (A95, pin 12) will be H. Thus A3 will set 1 → IE.

Note from the gates at the top of figure 15 that instructions 6001, 6003, 6005, 6007 (octal) will all be decoded as an ION instruction.

A3: As shown above 1 → IE. Also since the next instruction must always be executed independent of a pending interrupt we must branch to state 1 thereby ignoring state 0 (which tests interrupt). However, we must duplicate the work accomplished in F0.

F3: Assume we have some instruction that requires an operand such as TAD X. We must make a memory reference to get the contents of location X. The PDP 8 can address memory only within PAGES of 128 locations. The reason for this is that only 7 bits of the command (IR5 - IR11) are interpreted as a memory address. Another bit (IR5) is used to select one of two pages for the memory access. If IR5 = 0 bits IR5 - 11 are interpreted as a reference to page 0. If IR5 = 1 the reference is to the page that contains the instruction being executed. Since memory has 4096 (2^{12}) words we must generate a 12 bit memory address which requires that we concatenate an additional 5 bits to the 7 from the instruction. To access the current page these 5 bits must be the 5 most significant bits of the PC. We do this by ANDing IR5 with PC0-4.

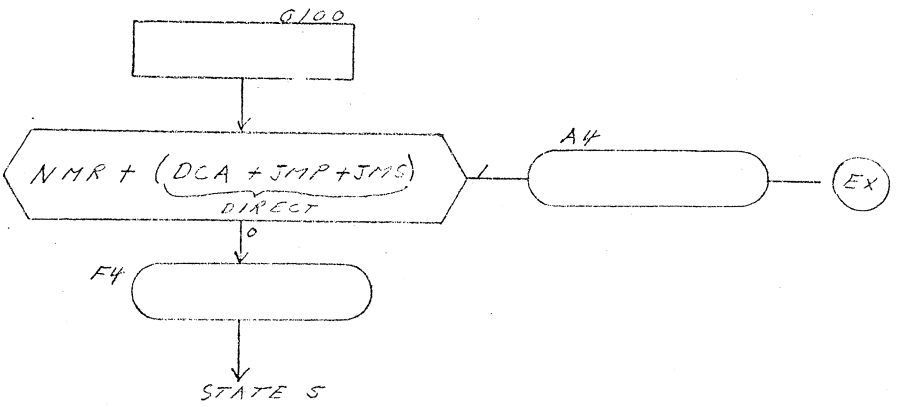
$$IR5 \cdot PC0 = 0 \text{ if } IR5 = 0 \quad = PC0 \text{ if } IR5 = 1$$
$$IR5 \cdot PC1$$
$$\vdots$$
$$IR5 \cdot PC4 = 0 \text{ if } IR5 = 0 \quad = PC4 \text{ if } IR5 = 1$$

We call this composite address the effective address EA.

$$EA\ (0\text{-}4) = IR5 \cdot (PC0\text{-}4)$$
$$EA\ (5\text{-}11) = IR(5\text{-}11)$$

These gates are shown on fig. LD18, A48, 49, 50. The total effect of F3 is to load MA with the EA in preparation for a memory read to retrieve the effective operand.

STATE 4



There is a fundamental difference between an instruction like JMP X and TAD X:

JMP X says jump to location X we do not need an effective operand, the effective address is sufficient and since this was loaded into MA in F3 we can exit directly to the execute cycle.

TAD X however, requires the contents of X so we must do an additional memory read to get the effective operand which must then be stored in MB before the execute cycle is entered.

State 4 is the place where we make the decision that an effective operand is not needed and therefore an immediate exit to the execute cycle can be made. There are two instructions that can be immediately decoded in this class:

IOT X (opcode=6) Initiate an input, output operation to peripheral device X. In the IOT instruction X address a device not memory.

MIC X MIC is a special class of instructions called the microcoded instructions. In this case X does not refer to memory but instead tells what kind of microcoded operation to carry out. An example would be IAC (Increment ACcumulator). None of the MIC operations require a memory reference. Note that CYCOMP = 0 because of MA(L) in F3.

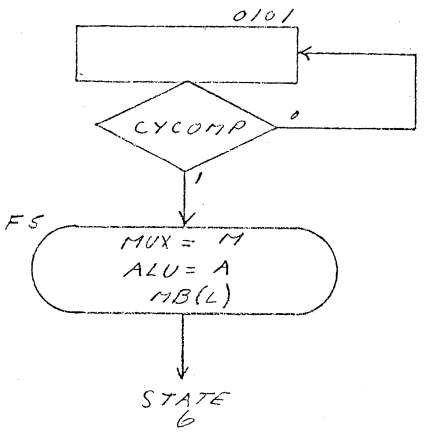
The DCA, JMP, and JMS instructions are slightly more complicated than shown above and require a discussion of INDIRECTION. In an instruction like JMP X, X will be the jump address if IR3 = 0. However, if IR3 = 1 then X is the address of the jump address. Such instructions are called indirect.

Thus if:

IR3 = 0	X is a direct address,
IR3 = 1	X is an indirect address.

Indirect addresses require one more read cycle to get the address which can then be treated in the normal fashion. Thus we can call the execute cycle only if the jumps are direct jumps. The direct DCA instruction is in the same category. Thus the condition for an early exit to the execute cycle is IOT + MIC + (JMS + JMP + DCA) · IR3 = 1.

STATE 5



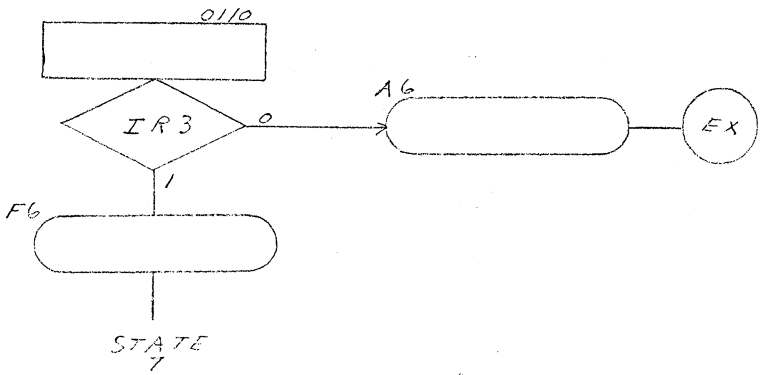
State 5 will be entered only for commands that require an effective operand (such as TAD X) or indirect jumps or DCA. In an indirect JMP *X (indicated by the * before the X) we calculated the address in step F3 but this is only the address of the address so we must read memory again to get the final address which we now call the effective address.

i.e.,

JMP X	jump to loc [X]
JMP *X	jump to loc [mem(X)]
TAD X	operand = contents of [mem(X)]

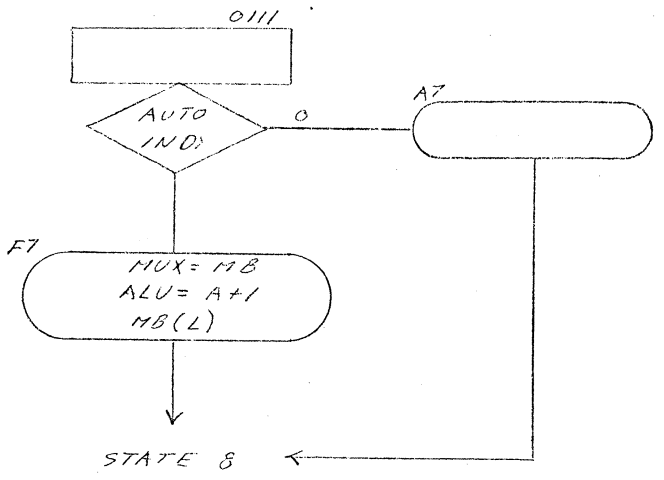
Since a memory access will be required in any case, a read pulse is issued and a test on CYCOMP is made. After the memory is stable the result is transferred to MB on the assumption that it may be the effective operand of an instruction like TAD X.

STATE 6



The function of state 6 is to test for the presence of a direct instruction like ~~ADD~~ TAD X. If the instruction is of that type the execute cycle can be entered. If it is an indirect instruction further processing is required.

STATE 7



If we get to state 7 we know that we have an indirect instruction. These instructions may use the auto index registers. They are eight words of memory, address 0010 = 0017 (octal). If one of these locations is accessed by an indirect instruction its contents are incremented before it is used as an address. This is a very convenient way for instructions to automatically index through memory. For example suppose we wish to clear a table 153_8 words long whose starting location is 321_8.

octal location	octal code	symbolic inst	operand address	comments
110	7200	CLA	---	move the starting address - 1 (320) to auto index register 12
111	1130	TAD	130	mem (130) = 320
112	3012	DCA	12	mem (12) = 320
113	1131	TAD	131	mem (131) = - 153
114	3132	DCA	132	mem (132) = - 153
115	3412	DCA	*12	clear table
116	2132	ISZ	132	
117	5115	JMP	115	repeat if less than 153 locations cleared
118		HLT		153 locs cleared
130	320	data location		must contain (st adr-1) = 320
131	7625	data location		must contain - 153

The first three instructions 110 - 112 set up the table starting address in auto index register 12.

Instructions 113 - 114 set up the loop count (table length) in location 132 for later use by the ISZ instruction. The general rule for a loop that repeats N times is to store the 2's complement of N in a memory location later used with ISZ instruction.

The actual loop is in locations 115 - 117. We will discuss the DCA *12 instruction in detail. DCA does not clear memory location 12. Since it is an indirect instruction location 12 contains the address of the final location to be accessed by the instruction. In addition since it is an

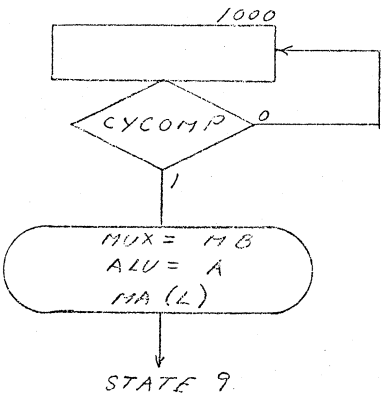
auto index location (0010 - 0017) that is referenced indirectly it is incremented before it is used. To start with it contains 320. The first time DCA *12 is executed the contents of memory location 12 is incremented to 321 and the 321 is used as the effective address. Therefore, location 321 is cleared. The next time a DCA *12 is executed the 321 will be incremented and then used as an address; thus location 322 will be cleared. The nice thing about this technique is the incrementation takes place automatically since the following two conditions were met:

1) The memory reference was indirect,
2) The memory reference was to an auto index register (0010 = 0017)

F7: Increments this address and puts the new value into the MB in preparation for writing it back into memory. A memory write pulse is issued.

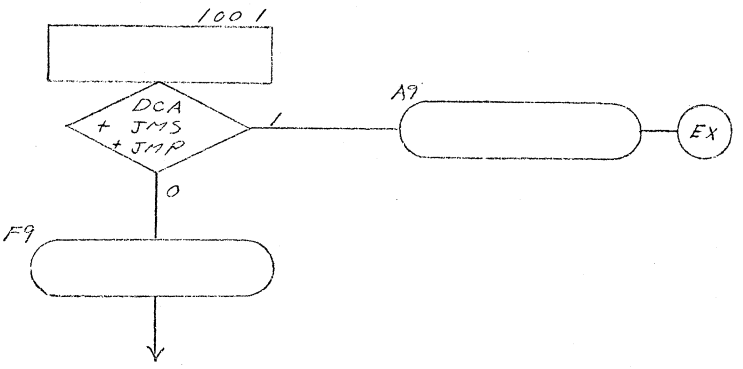
A7: Is the step taken if the indirect memory reference is to any memory location other than an auto index register.

STATE 8



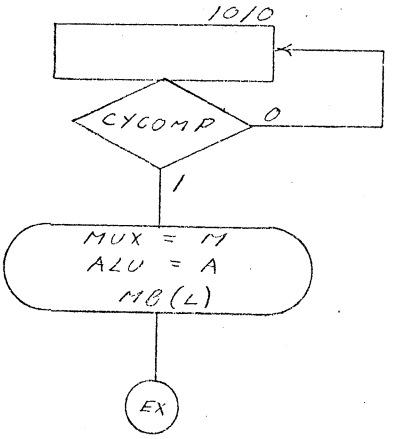
The function of state 8 is to wait until memory is stable at which time the auto index register will have the incremented address written back into it. The incremented address (the effective address) will still reside in MB so it can be transferred to MA as required by the execute cycle.

STATE 9



Again we have to recognize the fundamental difference between JMP *X instruction and a memory reference instruction like AND *X. In state 9 we have the effective address which is all we need to execute a jump instruction which we can do in A9. If it is a memory reference instruction we must do one final read to get the effective operand. This is done in F9.

STATE 10



F10: The sole function of this state is to load the effective operand into the MB and call the execute cycle.

Since the concepts of effective address and effective operand are so important, we will review them again.

EFFECTIVE ADDRESS:

The final address presented to the execute cycle after all indirection and indexing have been performed.

EFFECTIVE OPERAND:

The contents of the memory location pointed to by the effective address.

The fetch sequencer is shown on fig. LD11. The state flip-flops are the Q_D - Q_A outputs of the 74163 counter, B12. The counter is held to 0 by the fetch flip-flop, B47, except when this flip-flop is set (i.e., during FETCH cycle). The setting of B47 removes the clear and also enables the ET input of the counter. B47 also appears on figure 3 for clarity.

The 16 wide MUX (C3) senses the present state of counter and selects a test condition for each state. If the test voltage is H the corresponding Fi output on decoder (C1) becomes active. This also enables the EP input to the counter so it can increment to state i + 1 on the next clock tick. If test voltage (i) is L, output Ai (C2) becomes active and at the same time the count mode is inhibited holding the counter in its present state. A3 and A7 involve jumps to new address; therefore we must calculate the jump address and present it to the D inputs of the 74163. A load signal is simultaneously generated by the OR gate (A93).

Both the execute and fetch sequencers use a binary coding to represent states. Another scheme that is often used is to dedicate one flip-flop to each state. Such a design is called a ONE HOT coding since only one flip-flop should be set at any time. Such systems are very easy to design but will require more hardware for a machine with many states. We have two examples in the lab computer.

1) RUN state: The run/halt state may be represented by a single flip-flop. Run corresponds to the RUN FF set and halt to RUN FF reset. This flip-flop is shown on fig. LD9, B46, pin 9. The RUN FF must be set to permit the completion of execution (CP7) to call the fetch cycle.

$$GTF = RUN \cdot CP7$$

Go To Fetch = RUN · (end of ex cycle)

Thus the RUN FF can be reset any time during the execute cycle and that cycle will finish but the next fetch cycle will not start. The conditions for stopping the computer after the current instruction: are:

STOP switch is manually depressed.
SING INST switch is activated.
HALT command (7402) is executed.

The condition for setting RUN is a signal derived from the CONTinue switch on the control panel. CONT Pulse will be true for only one clock cycle when the continue switch is depressed. At the same time it is necessary to set the FETCH FF to start executing the first instruction.

2) FETCH and EXECUTE flip flops. This is another example of a ONE HOT coding since the machine can be in fetch or execute but not both. If you look at the fetch flow chart the conditions for starting an execute cycle are:

$$GTE \text{ (Go To Execute)} = A0 + A4 + A6 + A9 + F10$$

This is shown in fig. LD9, A10, pin 8. GTE simultaneously sets the EXEC FF (B46, pin 2) and resets the FETCH FF (B47, pin 14). Setting EXEC enables both the EXEC counter (B15, pin 1) and the CP decoder (B17, pin 12). Signals CP0 - CP7 are now generated in sequence to carry the machine through the execute cycle. CP7 will reset the EXEC FF (B46, pin 3) and set the FETCH FF <u>if</u> RUN = 1; GTF (A54, pin 11).

(12) DERIVATION OF FINAL LOGIC EQUATIONS

MUX EQUATIONS

Now that the complete flow charts for both fetch and execute have been explained we can derive the equations for control signals in final form. We do this by locating all occurrences of a given signal on both flow charts. For example MUX = PC occurs in the following places:

AO, FO, F2, A3, JMS·CPO, OPG2·CPO, ISZ·CP2

Therefore, the final equation for MUX = PC is:

$$MUXPC = AO + FO + F2 + A3 + (JMS + OPG2)\cdot CPO + ISZ\cdot CP2$$

Verify each of the following equations:

$$MUXM = F1 + F5 + F10$$
$$MUXEA = F3$$
$$MUXMB = F7 + F8 + (AND + TAD + ISZ) \cdot CPO$$
$$MUXMA = JMP\cdot CPO + JMS\cdot CP2 + (DEP + EX) \cdot CP6$$
$$MUXAC = DCA\cdot CPO + OPG1\cdot (CP2 + CP3)$$
$$MUXSR = \underbrace{(LDMA + LDM + LDMB + LDPC + LDIR + LDAC + DEP)}_{MANSW \cdot \overline{EX}} \cdot CPO + OPG2 \cdot CP2$$

The MUX must in turn be selected by sending it the proper 3 bit code B4, B2, B1. By referring to fig. LD1 the MUX assignments are:

input selected	B4	B2	B1
AC	0	1	1
MA	0	1	0
M	1	1	0
MB	0	0	1
PC	0	0	0
SR	1	0	1
EA	1	1	1

The equations for B4, B2, B1 can now be written by inspection.

$$B4 = MUXM + MUXSR + MUXEA$$
$$B2 = MUXAC + MUXMA + MUXM + MUXEA$$
$$B1 = MUXAC + MUXMB + MUXSR + MUXEA$$

The logic diagram for these equations is shown in fig. LD12. You may wonder why B4 is driven by two identical gates. The reason for this is that one gate output can only drive 10 gate inputs. But there are 12 MUX's that must be identically selected. By paralleling two gates we can drive 20 inputs. We could have generated the signal with one gate and then amplified it with a non-inverting buffer to get the additional power to drive 12 inputs. However, this would have added one more stage of delay and the chain is already long (5 levels).

ALU EQUATIONS

These equations are derived in a similar fashion to the MUX equations. Every occurrence of a given condition is collected together:

$$ALUA = F0 + F1 + A3 + F3 + F5 + F8 + F10 + (DCA + JMS + JMP + MANSW \cdot \overline{EX}) \cdot CP0$$

$$ALUA + 1 = A0 + F2 + F7 + (OPG2 + ISZ) \cdot CP0 + (ISZ + JMS) \cdot CP2 + OPG1 \cdot CP3 + (DEP + EX) \cdot CP6$$

$$ALUAND = AND \cdot CP0$$
$$ALUADD = TAD \cdot CP0$$
$$ALUNA = OPG1 \cdot CP2$$
$$ALUOR = OPG2 \cdot CP2$$

The control bits to achieve these functions are listed below; the conventions are:

M = 1,H S3 .. S0 = 1,H CIN = 1,L

	M	S3	S2	S1	S0	CIN	
ALUA	0	0	0	0	0	0	
ALUA + 1	0	0	0	0	0	1	
ALUAND	1	1	0	1	1	X	X can be either
ALUADD	0	1	0	0	1	0	a 1 or a 0
ALUNA	1	0	0	0	0	X	
ALUOR	1	1	1	1	0	X	

$$S0 = ALUAND + ALUADD = (AND + TAD) \cdot CP0$$
$$S1 = S2 + ALUAND = S2 + AND \cdot CP0$$
$$S2 = ALUOR = OPG2 \cdot CP2$$
$$S3 = S0 + S1 = S0 + S1$$
$$M = S1 + ALUNA = S1 + OPG1 \cdot CP2$$
$$CIN = ALUA + 1 = ALUA + 1$$

These equations are implemented in fig. LD13.

LOAD EQUATIONS

A summary of machine action is:

1) MUX selects a source register,
2) ALU operates on the output of the MUX,
3) A destination register must be loaded with the output of the ALU.

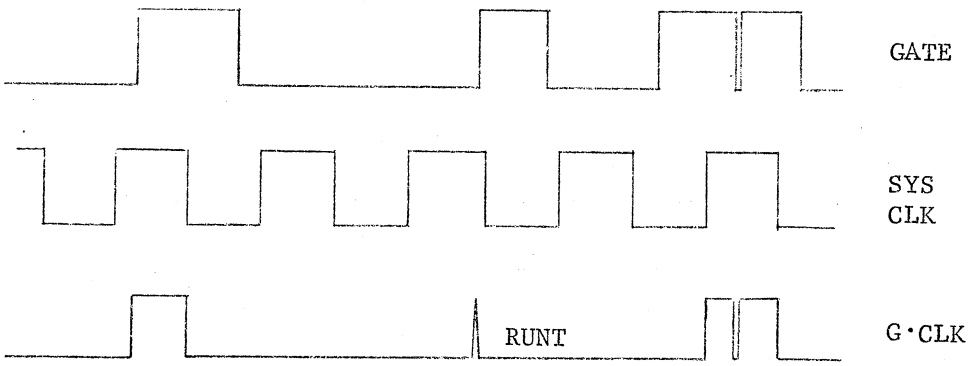
The registers are made from 74174 IC's which are hex D flip-flops with a common clock line. Whenever the clock line sees a positive going edge it loads the D inputs into the flip flops and stores the result until the next clock. Since these registers must hold data over many machine cycles we cannot feed the system clock to the register clock line. Instead we must feed a clock to a register only when we want to load it with the output of the ALU. (Remember that the ALU output is bussed to all register inputs). These special register clock signals are called register LOAD signals.

The design procedure for the load signals is a simple listing of the states and conditions that cause a register to be loaded:

MA(L) = F0 + A3 + F3 + F8 + LDMA·CP0 + (DEP + EX) · CP6
MB(L) = F5 + F7 + F10 + (ISZ + DCA + JMS + LDM + LDMB + DEP)·CP0
IR(L) = F1 + LDIR·CP0
AC(L) = (AND + TAD + LDAC)·CP0 + OPG1·CP2·IR6 + OPG2·CP2·IR9 + OPG1·CP3·IR11
PC(L) = A0 + F2 + (JMP + OPG2·T + LDPC)·CP0 + (ISZ·MB = 0 + JMS) · CP2
T = IR8⊕ (IR5·AC - + IR6·AC = 0 + IR7·L = 1)

These equations are shown on fig. LD14 and LD15. You will notice that a flip-flop is interposed between the gate that generates the load and its final destination at the corresponding register clock.

This special circuit is worth discussing in more detail. We need to present a single positive going edge to the clock input of a register to load it. The immediate thought is simply gate the clock to the register. GATING THE SYSTEM CLOCK IS NOT GOOD PRACTICE! To see this consider the timing diagrams:



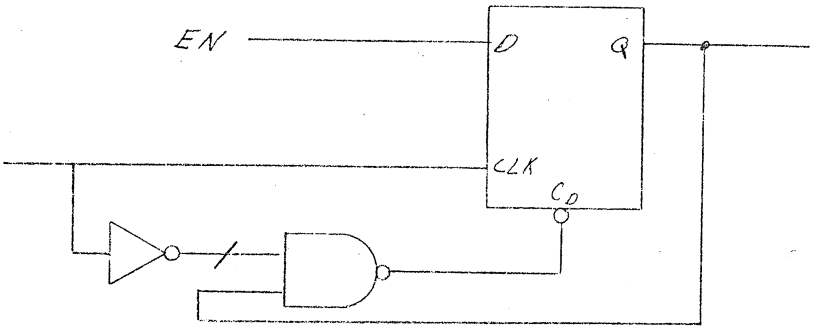
You can see that the first pulse is not the rising edge of the clock but is the rising edge of G. In general this is bad, since now the timing is no longer synchronized with the system clock. Another problem is the possible generation of a "RUNT" pulse if the gate and clock overlap only a small amount. Such a pulse may or may not trigger a flip-flop clock. This can cause the system to be "almost reliable" which is an impossible situation to trouble shoot. There is still a third problem with the gated clock. Real signals are never nice square waves as shown above. Suppose the CLK is clean but G contains a noise spike when both G and CLK are supposed to be true. You will get an extra clock edge on G · CLK! Extra clocks are seldom beneficial.

What we need is a device called an enabled clock passer. At the black box level it has the following properties:

1) The system clock can be continuously presented to the ECP,
2) A separate enable line will be tested by the ECP just before the system clock positive edge.
   a) If enable = H, exactly one clock cycle will appear at the output,
   b) If enable = L, the output will remain low.

3) The above must be accomplished without gating the system clock.

A circuit to accomplish this is shown below:



If EN is present at the clock edge the FF will set making Q = 1. It will remain there for ½ clock cycle at which time $\overline{CLK}$ and Q are both true so the output of the AND gate is L which will reset the flip flop. Thus the FF reproduces the positive half of the clock cycle. Even so the above circuit can be subject to failure if the propagation delay of the inverter is longer than the delay of the flip-flop. This should be demonstrated by drawing a timing diagram. In the lab kit this problem is solved by using a very high speed inverter to produce $\overline{CLK}$. The skew between CLK and $\overline{CLK}$ is 6 ns. The propagation delay of the 74174 is typically 20 ns. To further reduce the sensitivity to the critical timing between $\overline{CLK}$ and Q a low power AND gate (74L00) is used. These gates have propagation delays of 35 ns and are quite insensitive to runt pulses.

The feeling you should get from the above discussion is that gating the system clock is dangerous; that the ECP is a better solution, the ECP is still not the ideal solution.

The ideal solution is to use the gated D flip-flop used in the interrupt system (IE). If these flip-flops were used for registers the system clock could be hooked to the gated D clock line continuously without any gating on that line. The G line involves gating but now it is in the data path. The only requirement is that G be stable at the clock tick. Unfortunately, these IC's are just now becoming commercially available so we are forced to use a less than ideal solution. Future designs should always use the gated D for registers.

ACCUMULATOR EQUATIONS

The accumulator is shown in fig. LD19 and is made from three universal shift registers (74194). These registers are controlled by two bits, ACS0 and ACS1 as shown in the following tables:

S1	S0	
H	H	load new data
H	L	shift left
L	H	shift right
L	L	retain old data

The primary signals for AC control are:

AC(L)	load ALU output into AC
RAR	shift AC right (circular shift with LINK)
RAL	shift AC left (circular shift with LINK)
CLA	clear (asynchronous)

The encodings for ACS0, ACS1 are:

S1 = AC(L) + RAL	gate A92, pin 8
S0 = AC(L) + RAR	gate A92, pin 6

Note that since the clear is asynchronous there must be no noise on that line. The equations for RAR, RAL, and CLA can be derived from the execute flow chart:

RAR = OPG1·IR8 (CP4 + IR10 · CP5)
RAL = OPG1·IR9 (CP6 + IR10 · CP7)
CLA = OPG1·IR4·CP0 + (OPG2·IR4 + DCA)·CP1 + CLRP
(CLRP is derived from the control panel clear button)

LINK EQUATIONS

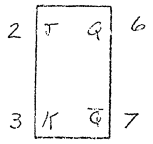
The link is a single FF (B8, fig. LD19) used to store the carry out of the accumulator. Each carry generated by an arithmetic operation (TAD + IAC) must complement the link. The link can also be complemented by a microcoded instruction, CML (7020). There is one other microinstruction which can affect the link, CLL (7100) which clears it.

By reference to the execute flow chart we can derive the link control equations (figure 10)

CLL = OPG1·CP0·IR5 + CLRP
CML = OPG1·CP2·IR7 + (TAD + OPG1·CP3·IR11)·COUT

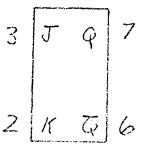
Since we are nearly finished with our discussion of the design we will implement the link in a sophisticated manner. We will lead up to this by easy stages. The result is shown in LD23.

1) A way to look at a JK flip-flop is that the output opposite the input will be the one responding to the input.



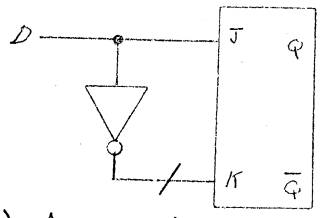
if
if J = 1, Q will be set if K = 0
if K = 1, $\bar{Q}$ will be set if J = 0
2, 3, 6, 7 are pin numbers

2) This allows us to turn the flip-flop "over" and relabel J, K, Q, $\bar{Q}$. It is still true that:



if J = 1, Q will be set if K = 0
if K = 1, $\bar{Q}$ will be set if J = 0

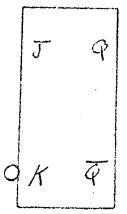
3) A D FF can be made from a JK FF by making $K = \overline{J}$.

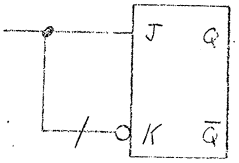


if D = 1, J = 1, K = 0 and Q will set (=1)
if D = 0, J = 0, K = 1 and Q will reset (=0)

i.e., Q = D after a clock

4) A very nice version of a JK would have the K low active and the J high active since this device could be converted into a D simply by connecting J directly to K.





is a D since $K = \overline{J}$.

One manufacturer has recognized this and produces such an FF (Fairchild 9024).

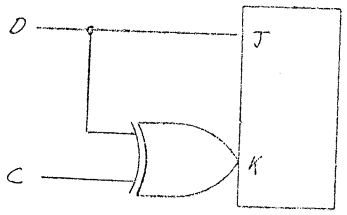
5) An exclusive OR can also be regarded as a controlled inverter as can be seen from its truth table:

C	I	Ø
0	0	0
0	1	1
1	0	1
1	1	0

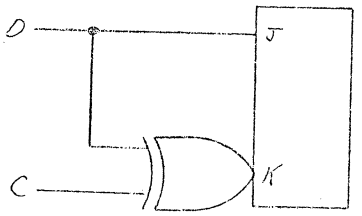
C = control
I = input
Ø = output

When C = 0, Ø = I; when C = 1, $\emptyset = \overline{I}$.

6) We can use this fact to convert a JKFF into a dual purpose FF.



When C = 1 it acts like a D FF



when C = 0 and D = 0 it will retain its old value. When C = 0 and D = 1, it will toggle.

7) Write Karnaugh maps for C, D as functions of RAL, RAR, CML. Note that only one variable of RAL, RAR, CML can be true at one time. This is responsible for the don't care conditions below:

C	RAL, RAR 00	01	11	10
CML 0	0	1	–	1
CML 1	0	–	–	–

R = rt. shift data

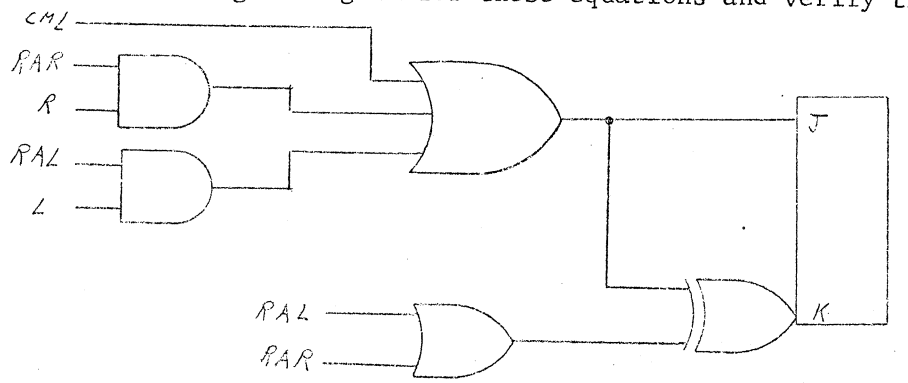
D	RAL, RAR 00	01	11	10
CML 0	0	R	–	L
CML 1	1	–	–	–

L = left shift data

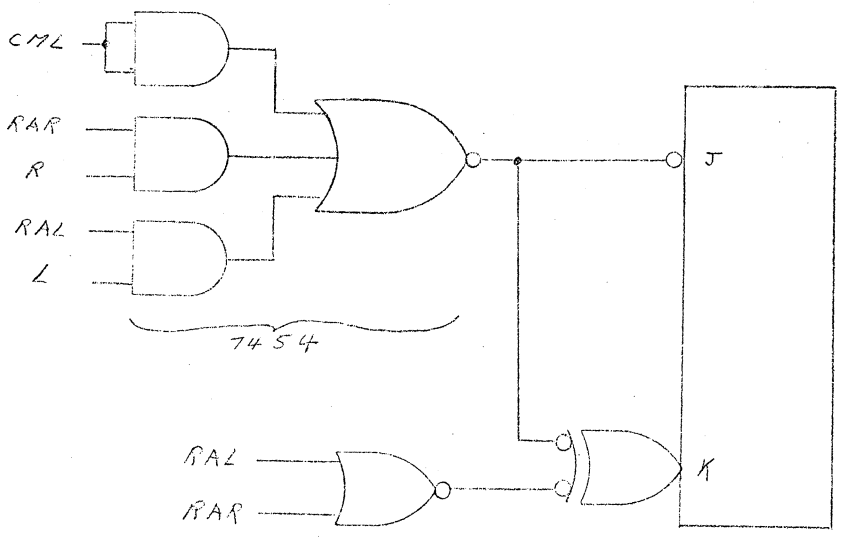
See Clare's book, Sec. 4.4 for the interpretation of the map for D

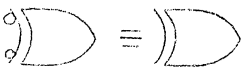
$$C = RAL + RAR$$
$$D = CML + RAR \cdot R + RAL \cdot L$$

8) Draw the logic diagram for these equations and verify that it works.



9) Redraw using 7454 AND, OR, INVERT gate and 9024 JK FF.

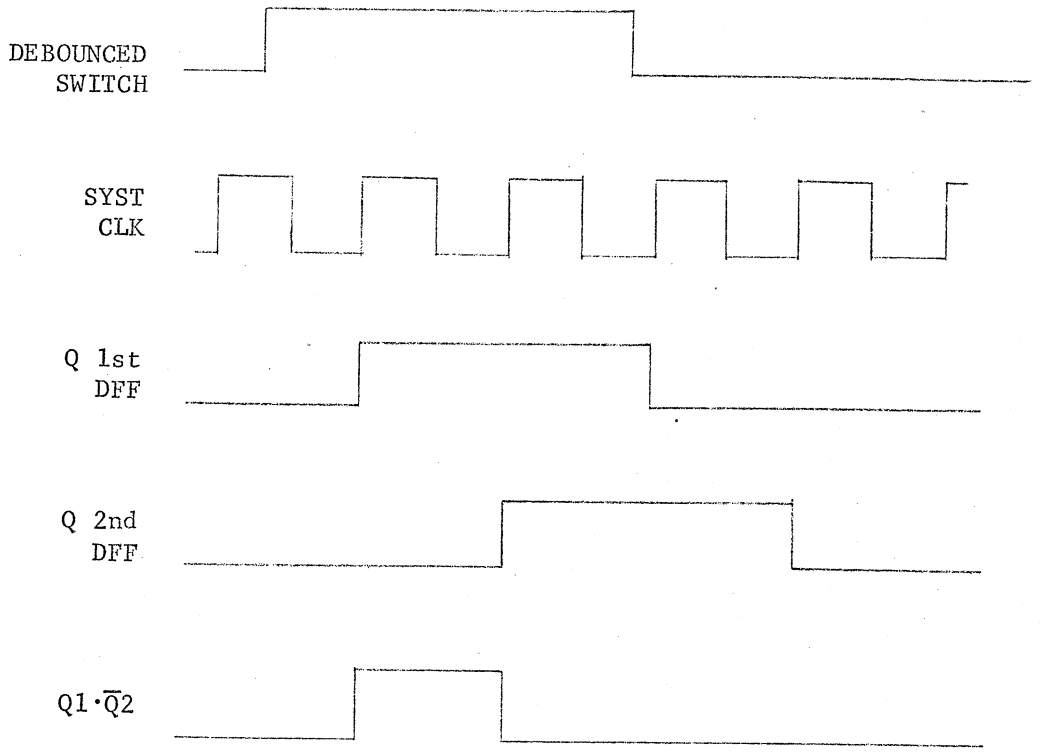


NOTE: 

CONTROL PANEL SIGNALS (Fig. LD8)

All of the pushbutton signals from the panel are debounced by RS flip-flops before being sent to the edge connector. By correct jumpering the polarity of each signal can be chosen H or L when the switch is depressed. Your panels are wired to produce L polarities on switch depression.

Some of these signals must be shortened to one pulse synchronized with the system clock. This can be accomplished by a two bit shift register, (A62, pin 2) and (A62, pin 12). This can be shown by the timing diagram:



Note that the switch must be held down for at least one clock edge. Most of the signals are ANDED with $\overline{RUN}$ so they will be active only when the machine is not running.

	0 AND	1 TAD	2 ISZ	3 DCA	4 JMS	5 JMP	6 IOT	7 • (IR3 OPG1
CP 0	MUX = MB ALU = A•B AC(L)	MUX = MB ALU = A(+)B AC(L)	MUX = MB ALU = A+1 MB(L)	MUX = AC ALU = A MB(L)	MUX = PC ALU = A MB(L)	MUX = MA ALU = A PC(L)		CLA = IR4 CLL = IR5
CP 1			WRITE	WRITE CLA	WRITE			
CP 2			MUX = PC ALU = A+1 PC(L) = (MB=0)		MUX = MA ALU = A+1 PC(L)			CML = IR7 MUX = AC ALU = $\overline{A}$ AC(L) = IR6
CP 3			CYCOMP	CYCOMP	CYCOMP			MUX = AC ALU = A+1 AC(L) = IR11
CP 4								RAR = IR8
CP 5								RAR = IR8 • IR10
CP 6								RAL = IR9
CP 7								RAL = IR9 • IR10

FIG 102

	7·(IR3=1) OPG 2	LDMA	LDM	LDMB	LDPC	LDIR	LDAC	DEP	EX
CP 0	MUX = PC ALU = A+1 PC(L) = TST	MUX = SR ALU = A MA(L)	MUX = SR ALU = A MB(L)	MUX = SR ALU = A MB(L)	MUX = SR ALU = A PC(L)	MUX = SR ALU = A IR(L)	MUX = SR ALU = A AC(L)	MUX = SR ALU = A MB(L)	
CP 1	CLA = IR 4		WRITE					WRITE	
CP 2	MUX = SR ALU = A+B AC(L) = IR9								
CP 3	HLT = IR 10		CYCOMP					CYCOMP	
CP 4									
CP 5									
CP 6								MUX = MA ALU = A+1 MA(L)	MUX = MA ALU = A+1 MA(L)
CP 7									

TST = · IR 8 ⊕ (IR5 · AC-1 + IR6 · AC=0 + IR7 · LINK=1)

FIG 103

FIG LD 4

FETCH 1

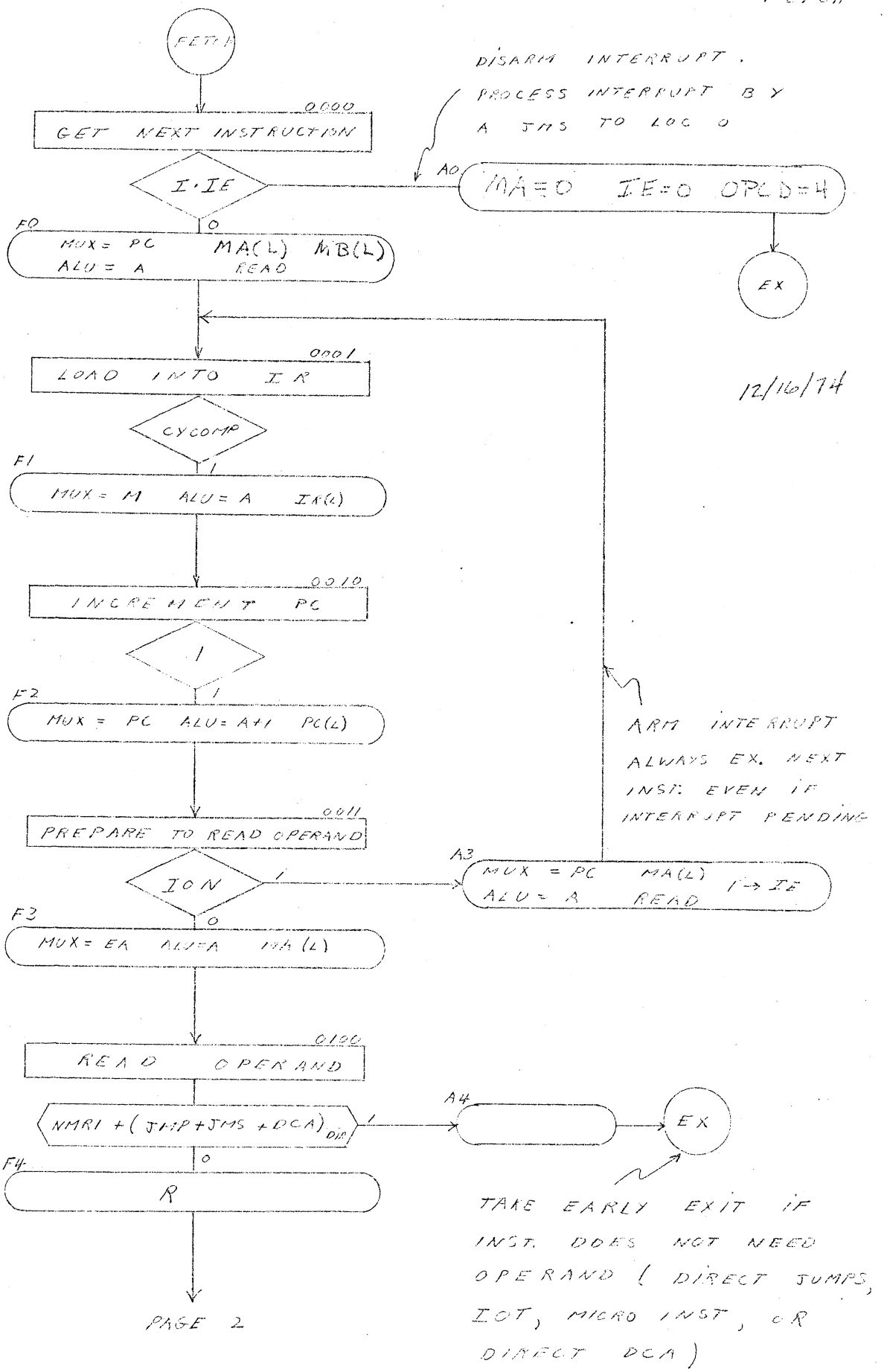
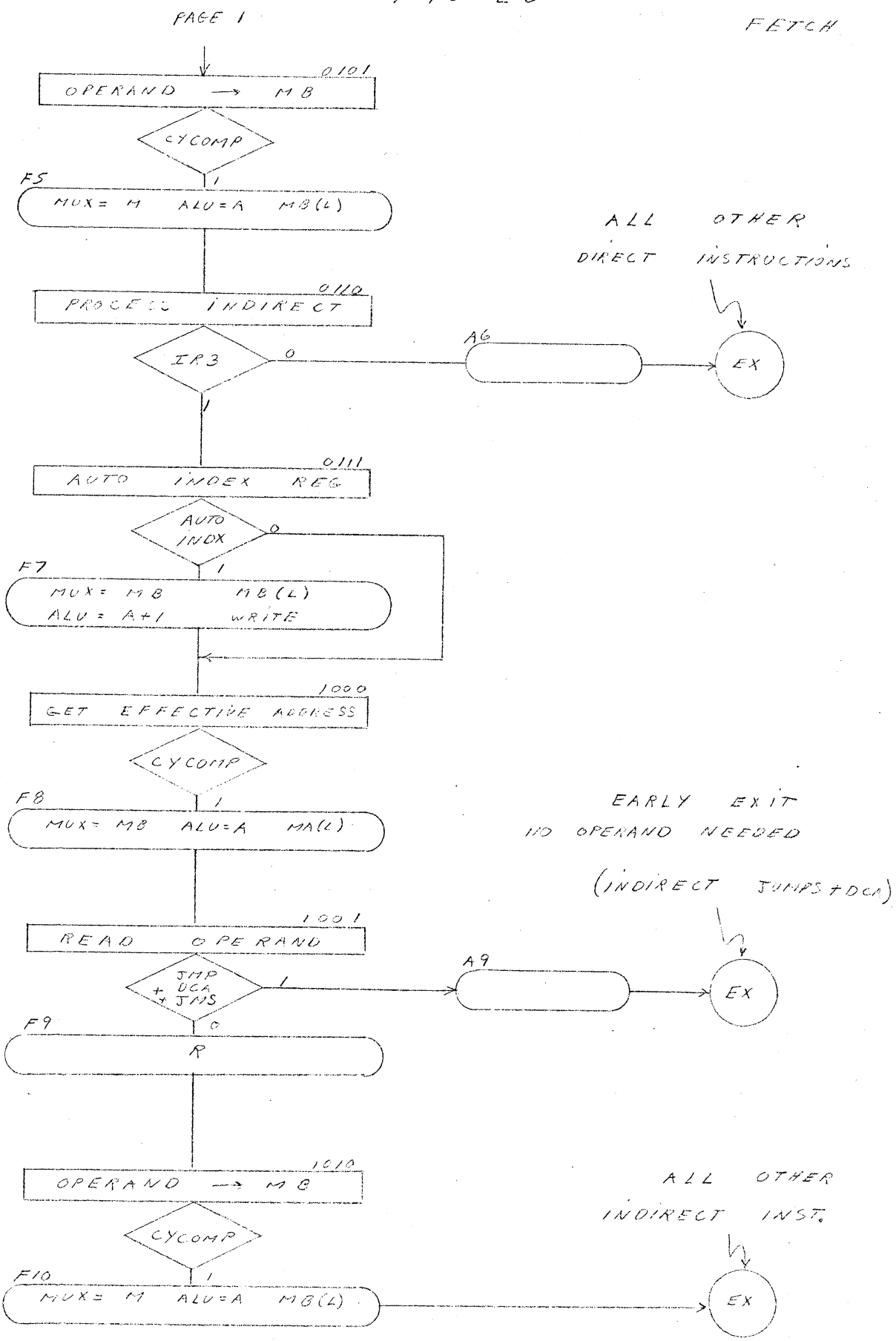


FIG 105


PAGE 1
FETCH 2
0101
OPERAND → MB
CYCOMP
1
F5
MUX = M ALU = A MB(L)
ALL OTHER
DIRECT INSTRUCTIONS
0110
PROCESS INDIRECT
IR3
0
A6
EX
1
0111
AUTO INDEX REG
AUTO INDX
0
1
F7
MUX = MB MB(L)
ALU = A+1 WRITE
1000
GET EFFECTIVE ADDRESS
CYCOMP
1
F8
MUX = MB ALU = A MA(L)
EARLY EXIT
NO OPERAND NEEDED
(INDIRECT JUMPS + DCA)
1001
READ OPERAND
JMP + DCA + JMS
1
A9
EX
0
F9
R
1010
OPERAND → MB
ALL OTHER
INDIRECT INST.
CYCOMP
1
F10
MUX = M ALU = A MB(L)
EX

12/16/74 LD 6

LOGIC EQUATIONS

Don't refer to this
chart until explained + names, etc

MUX:

$B4 = F1 + F3 + F5 + F10 + MANSW \cdot \overline{EX} \cdot CP0 + OPG2 \cdot CP2$

$B2 = F1 + F3 + F5 + F10 + (DCA + JMP) \cdot CP0 + (OPG1 + JMS) \cdot CP2$
$+ OPG1 \cdot CP3 + (DEP + EX) \cdot CP6$

$B1 = F3 + F7 + F8 + (AND + TAD + DCA + ISZ) \cdot CP0$
$+ MIC \cdot CP2 + OPG1 \cdot CP3$

ALU:

$S0 = (AND + TAD) \cdot CP0$ $\quad M = S1 + OPG1 \cdot CP2$

$S1 = S2 + AND \cdot CP0$

$S2 = OPG2 \cdot CP2$

$S3 = S0 + S1$

$CIN = F2 + F7 + (OPG2 + ISZ) \cdot CP0 + (ISZ + JMS) \cdot CP2$
$+ OPG1 \cdot CP3 + (DEP + EX) \cdot CP6$

LOADS:

$MA(L) = F0 + A3 + F3 + F5 + LDMA \cdot CP0 + (DEP + EX) \cdot CP6$

$MB(L) = F0 + F5 + F7 + F10 +$
$(ISZ + DCA + JMS + LDMB + DEP) \cdot CP0$

$IR(L) = F1 + LDIR \cdot CP0$

$AC(L) = (AND + TAD + LDAC) \cdot CP0$
$+ (OPG1 \cdot IR6 + OPG2 \cdot IR9) \cdot CP2 + OPG1 \cdot CP3 \cdot IR11$

$PC(L) = F2 + (JMP + OPG2 \cdot T + LDPC) \cdot CP0$
$+ (ISZ \cdot MB=0 + JMS) \cdot CP2$

$T = IR8 \oplus (IR5 \cdot AC- + IR6 \cdot AC=0 + IR7 \cdot L=1)$

CONNECTORS

C1

FRONT

1-3	4-5	6-7	8-9	10-11	12-13	14-15	16 ← 28	29-30	31-32	33-34	35-38	39-40	41-42	43-44	45-46	47-48	49-50
GND	LINE	SR3	SR2	SR1	SR0	SRW	GND	SRX	SRY	SRE	GND	SING INST	STOP	CLR	CONT	EX	DEP

BACK

100-97	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64	63	62	61	60	59	58	57	56	55	54	53	52-51
CLOCK LOW	MA 3	MB 3	PC 3	IR 3	AC 3	MA 2	MB 2	PC 2	IR 2	AC 2	MA 1	MB 1	PC 1	IR 1	AC 1	MA 0	M 0	MB 0	PC 0	IR 0	AC 0	F 1	CLK LED	SC	RL	DR L	MA LED	F 2	RUN	S1	IE	MA 1	F 4	MB L	S2	KIN	MB 2	AC S0	F 8	MA L	S3	MODE	MUX 4	AC S1

C2

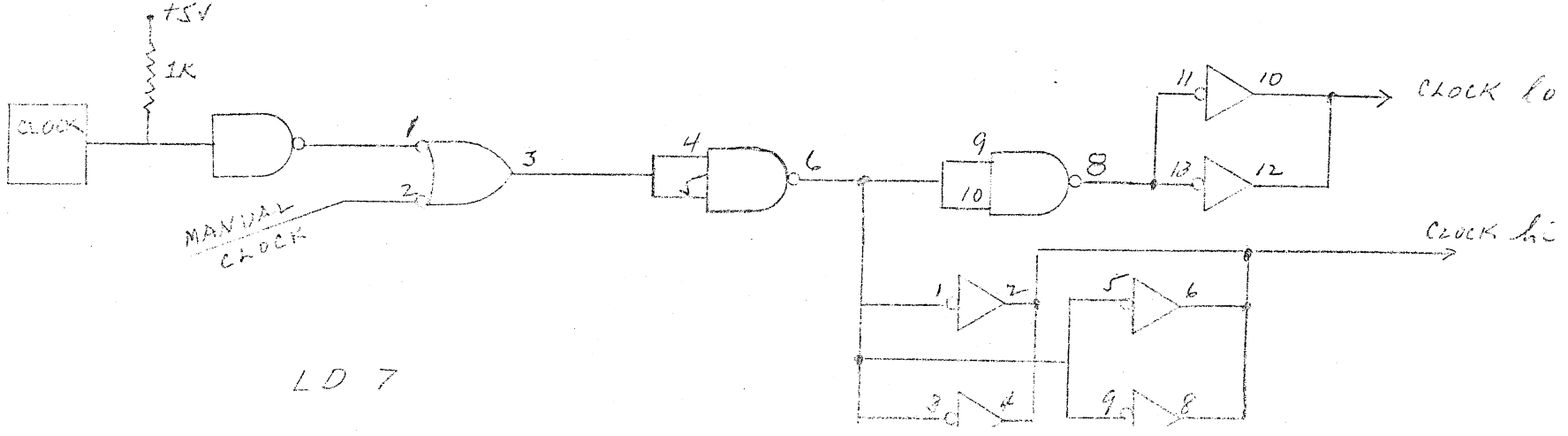
FRONT

1-2	3-4	5-6	7-8	9-10	11-12	13-16	17-20	21-24	25-29	30-31	32-33	34-35	36-37	38-39	40-41	42-43	44-45	46-47	48-50
LDMA	LDM	LDMB	LDPC	LDIR	LDAC	-5V	+5V	+12V	GND	SR 11	SR 10	GND	SR 9	SR 8	SR 7	SR 6	SR 5	SR 2	GND

BACK

100	99	98	97	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64	63	62	61	60	59	58	57	56	55	54	53	52-51
AC 11	IR 11	PC 11	MB 11	M 11	MA 11	AC 10	IR 10	PC 10	MB 10	M 10	MA 10	AC 9	IR 9	PC 9	MB 9	M 9	MA 9	AC 8	IR 8	PC 8	MB 8	M 8	MA 8	AC 7	IR 7	PC 7	MB 7	M 7	MA 7	AC 6	IR 6	PC 6	MB 6	M 6	MA 6	AC 5	IR 5	PC 5	MB 5	MA 5	AC 4	IR 4	PC 4	MB 4	M 4	MA 4	CLOCK HIGH

CLOCK BOOST:



LD 7

LD 7